

# **BBM 201**

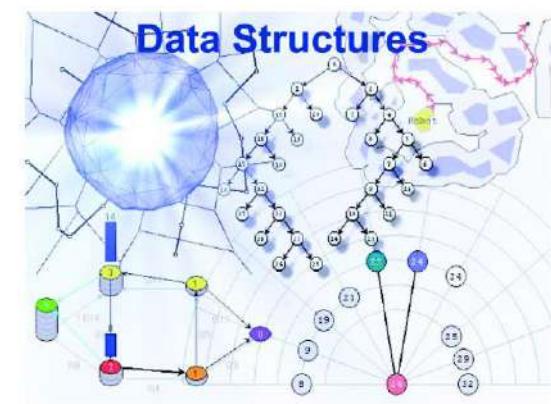
# **DATA STRUCTURES**

---

**Lecture 1:**  
**Basic concepts for data structures**



**2018-2019 Fall**



# About the course

- This course will help students understand the **basic data structures** such as matrices, stacks, queues, linked lists, etc.
- **BBM 203 Programming Laboratory:** The students will gain hand-on experience via a set of programming assignments supplied as complementary.
- **Requirements:** You must know basic programming (i.e. BBM101).

# References

- Data Structures and Algorithm Analysis in C++, 4<sup>th</sup> edition, Mark Allen Weiss, Pearson, 2014
- Problem Solving and Program Design in C, 7th Edition. Jeri Hanly and Elliot Koffman, Pearson, 2013
- Fundamentals of Data Structures in C. Ellis Horowitz and Sartaj Sahni, 1993.
- Fundamentals of Data Structures in C++. Ellis Horowitz and Sartaj Sahni, 1995.
- Data Structures Notes, Mustafa Ege.

# Communication



- The course web page will be updated regularly throughout the semester with lecture notes, programming assignments, announcements and important deadlines.

<http://web.cs.hacettepe.edu.tr/~bbm201>

# Getting Help

- **Office hours**

See the web page for details

- **BBM 203 Programming Laboratory**

Course related recitations, practice with example codes, etc.

- **Communication**

Announcements and course related discussions through



BBM 201: <https://piazza.com/hacettepe.edu.tr/fall2018/bbm201>

BBM 203: <https://piazza.com/hacettepe.edu.tr/fall2018/bbm203>

# Course Work and Grading

- **2 midterm exams (52%)**
  - Closed book and notes
- **Final exam (40%)**
  - Closed book
  - To be scheduled by the registrar
- **Quizzes & Homeworks (8%)**
  - Attempting to create false attendance (e.g., signing in the attendance list on behalf of someone else) will be punished.
  - Attendance is mandatory – students who fail to attend more than **%30** of the lectures will fail from the course (**≈if you do not attend 4 lectures, you will fail**).



The joy of learning

# Course Overview

| Week | Date   | Topic   |
|------|--------|---|
| 1    | Oct 11 | Introduction to Data Structures                       |
| 2    | Oct 18 | Performance Analysis                                  |
| 3    | Oct 25 | Representation of Multidimensional Arrays             |
| 4    | Nov 1  | Band, Sparse and Triangular Arrays                    |
| 5    | Nov 8  | Stacks and Queues                                     |
| 6    | Nov 15 | Evaluation of Expressions                             |
| 7    | Nov 22 | Array-based Linked Lists                              |
| 8    | Nov 29 | <b>Midterm Exam 1</b>                                 |
| 9    | Dec 6  | Linked Lists Applications (Stacks, Queue, Hashtables) |
| 10   | Dec 13 | Doubly Linked Lists                                   |
| 11   | Dec 20 | Binary Trees, Tries                                   |
| 12   | Dec 27 | Graph Representation                                  |
| 13   | Jan 3  | <b>Midterm Exam 2</b>                                 |
| 14   | Jan 10 | Examples  |

# BBM 203 Programming Laboratory I

- **Programming assignments (PAs)**
  - Four assignments throughout the semester.
  - Each assignment has a well-defined goal such as solving a specific problem.
  - You **must work alone** on all assignments stated unless otherwise.
- **Important Dates**
  - Programming Assignment 1 22 October 2018
  - Programming Assignment 2 5 November 2018
  - Programming Assignment 3 26 November 2018
  - Programming Assignment 4 17 December 2018

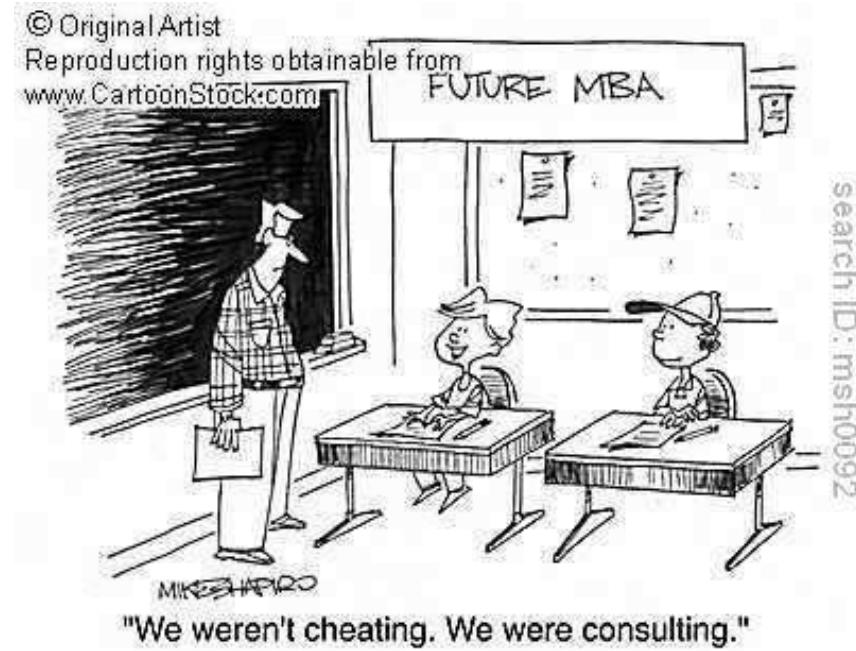
# Policies

- **Work groups**
  - You must work alone on all assignments stated unless otherwise
- **Submission**
  - Assignments due at 23:59 (no extensions!)
  - Electronic submissions (no exceptions!)
- **Lateness penalties**
  - No late submission is accepted
- **Attendance**
  - Attendance is mandatory – If you do not attend **4 recitation classes**, you will fail F1 grade.
  - If you do not get more than 25 points for 3 out of 4 assignments will fail with F1 grade.

# Cheating

- **What is cheating?**

- Sharing code: by copying, retyping, looking at, or supplying a file
- Coaching: helping your friend to write a programming assignment, line by line
- Copying code from previous course or from elsewhere on WWW



- **What is NOT cheating?**

- Explaining how to use systems or tools
- Helping others with high-level design issues

# Cheating

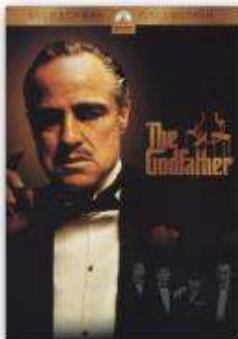
- **Penalty for cheating:**
  - Suspension from school for 6 months (minimum)



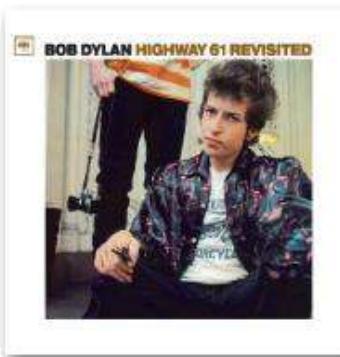
- **Detection of cheating:**
  - We do check: Our tools for doing this are much better than most cheaters think!

# **BASIC CONCEPTS FOR DATA STRUCTURES**

# Digital Data



Movies



Music



Photos

DNA

```
gatctttta tttaaacat ctcttatta gatctttat taggatcatg atccctgtgt  
gataagtat tattcacatg gcagatcata taattaagga ggatcgtttg ttgtgagtga  
ccgggtatcg tattggat aagctggat ctaaatggca tggatgcac agtcactcg  
cagaaatcaag gttgttatgt ggatatctac tggtttacc ctgttttaa gcatagttat  
acacattcgt tcgcgcgatc tttgagctaa ttagagtaaa ttaatccaat ctttgaccca
```



Protein  
Shapes



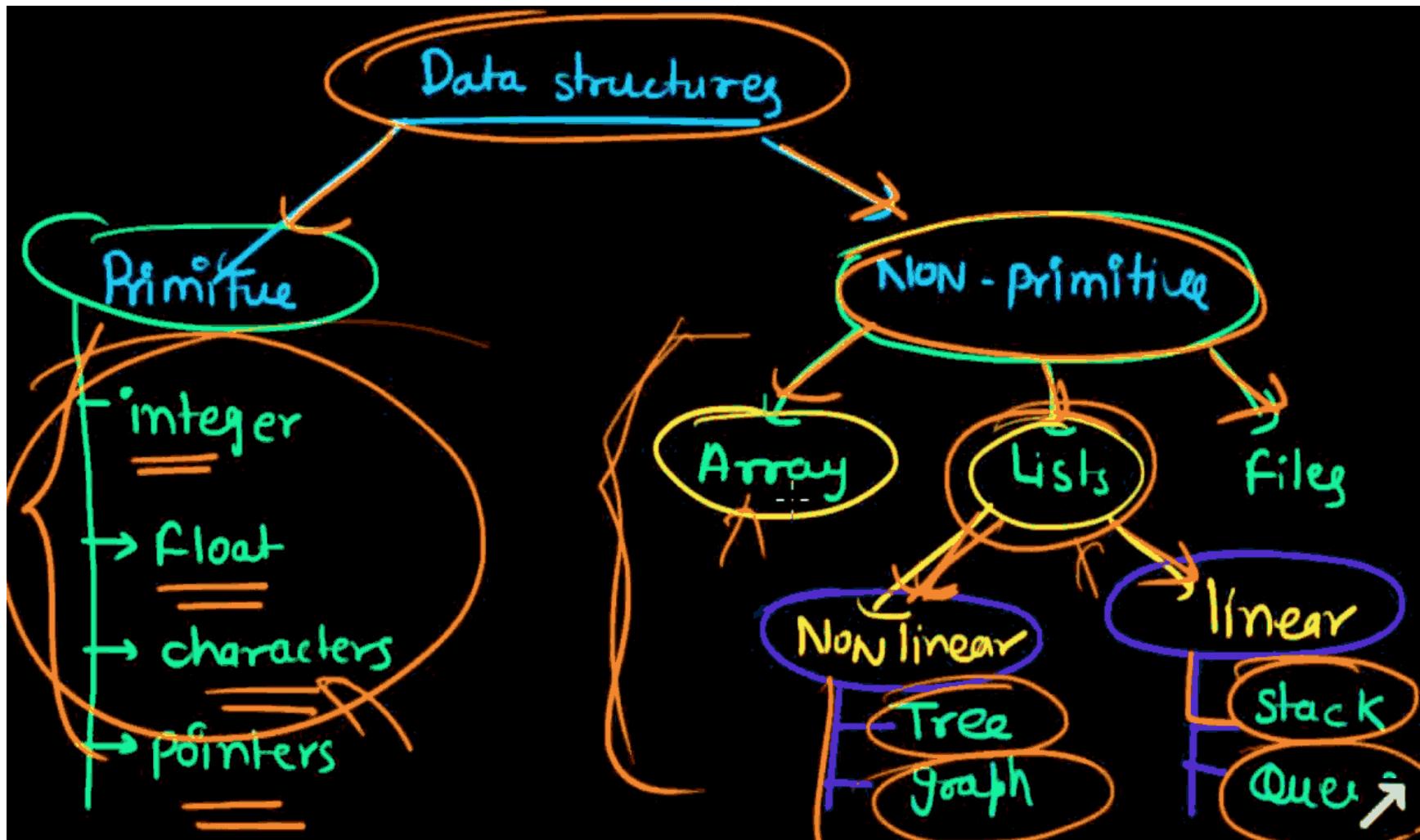
Maps

00101010010101010100100101010000010010010100....



Evolution of the Desk

1980





# Digital Data Must Be ...

- **Encoded** (e.g. 01001001 <-> 
  - **Arranged**
    - Stored in an orderly way in memory / disk
  - **Accessed**
    - Insert new data
    - Remove old data
    - Find data matching some condition
  - **Processed**
    - Algorithms: shortest path, minimum cut, FFT, ...
- The focus of this class

## Data Structures → Data StructurING

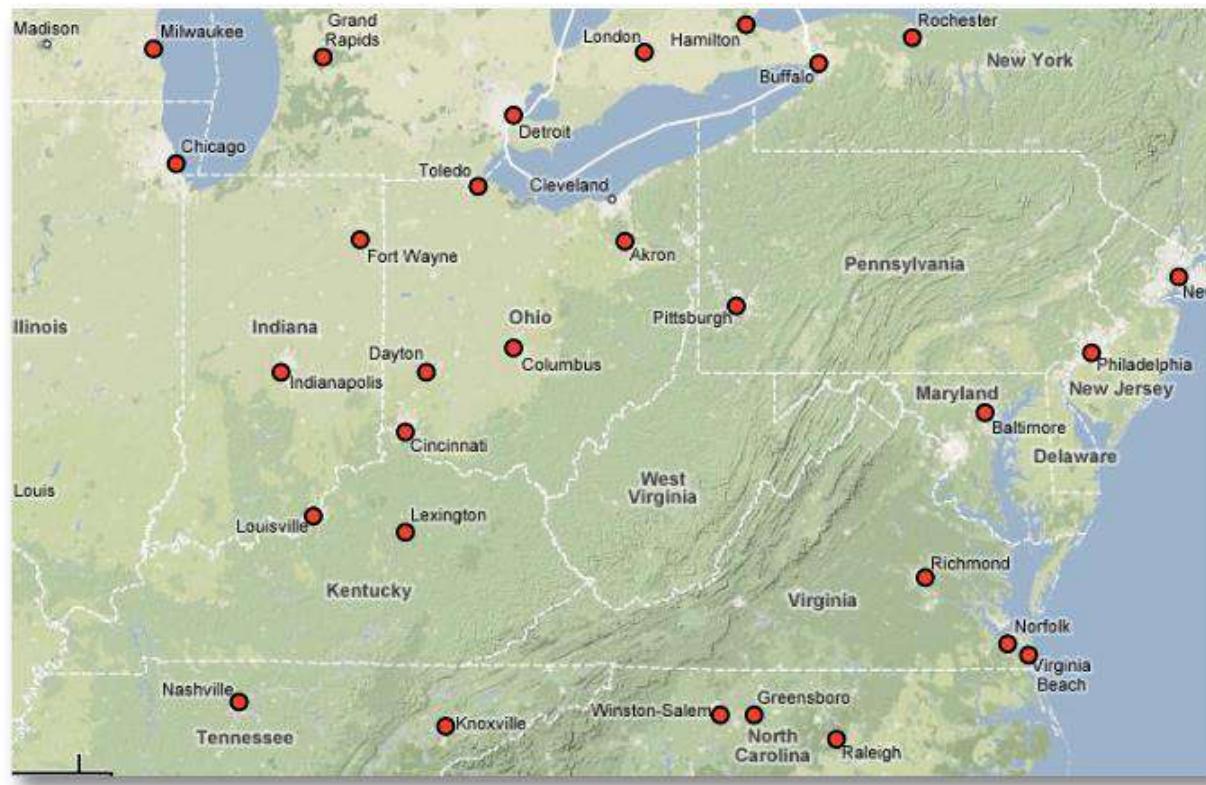
How do we organize information so that we can find, update, add, and delete portions of it efficiently?

# Data Structure Example Applications

- How does Google quickly find web pages that contain a search term?
- What's the fastest way to broadcast a message to a network of computers?
- How can a subsequence of DNA be quickly found within the genome?
- How does your operating system track which memory (disk or RAM) is free?
- In the game Half-Life, how can the computer determine which parts of the scene are visible?

# Suppose You're Google Maps...

- You want to store data about cities (location, elevation, population)...



What kind of operations should your data structure(s) support?

# Operations to support the following scenario...

Finding addresses on map?

- *Lookup city by name...*

Mobile user?

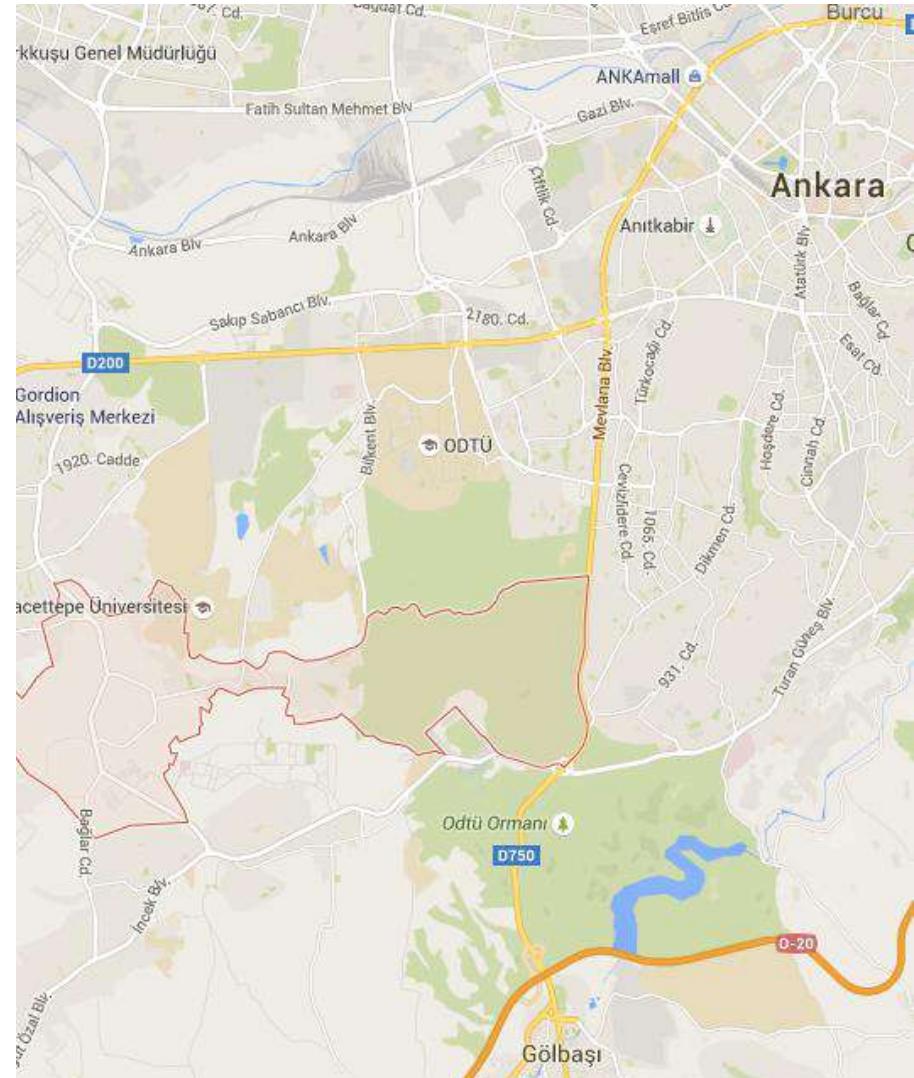
- *Find nearest point to me...*

Car GPS system?

- *Calculate shortest-path between cities...*
- *Show cities within a given window...*

Political revolution?

- *Insert, delete, rename cities*



# How will you count user views on YouTube?

- Lets write a userViewCount() function



```
int userViewCount (int  
current_count)  
{  
    int new_count;  
    new_count = current_count + 1;  
    return new_count;  
}
```

Will this implementation  
work all the time?

# How will you count user views on YouTube?

%99.9 times yes.



PSY - GANGNAM STYLE (강남스타일) M/V



officialpsy

Subscribe

7,605,627

2,153,880,168

+ Add to

Share

\*\*\* More

1,142,528

1,142,528

# How will you count user views on YouTube?

BBC | Sign in

News Sport Weather Shop Earth Travel

## NEWS

Home Video World UK Business Tech Science Magazine Entertainment & Arts

Asia China India

Asia

### Gangnam Style music video 'broke' YouTube view limit

① 4 December 2014 · Asia



<http://www.bbc.com/news/world-asia-30288542>

The Economist

World politics Business & finance Economics Science & technology Culture

### The Economist explains

Explaining the world, daily

Previous Next Latest The Economist explains All latest updates

### The Economist explains

### How “Gangnam Style” broke YouTube’s counter

Dec 10th 2014, 23:50 BY G.F. | SEATTLE

Timekeeper Like 6.1k Tweet 114

<http://www.economist.com/blogs/economist-explains/2014/12/economist-explains-6>

**YouTube's counter previously used a 32-bit integer**

YouTube said the video - its most watched ever - has been viewed more than 2,147,483,647 times.

It has now changed the maximum view limit to 9,223,372,036,854,775,808, or more than nine quintillion.

# How bad can it be?

- June 4, 1996
- Ariane 5 rocket launched by the European Space Agency
- After a decade of development costing \$7 Billion (~21 Billion in Turkish Liras, just for comparison Istanbul's third bridge cost estimates are 4.5 Billion TL)
- Exploded just 40 seconds after its lift-off
- The destroyed rocket and its cargo were valued at \$500 million
- Reason?



# How bad can it be?

- Reason?
- Inertial reference system error: specifically a 64 bit floating point number relating to the horizontal velocity of the rocket with respect to the platform was converted to a 16 bit signed integer.
- The number was larger than 32,767, the largest integer storable in a 16 bit signed integer, and thus the conversion failed.
- \$500 Million rocket/cargo
- Time and effort



# Floating Point Representation

Format of Floating points  
IEEE754

64bit = double, double precision



32bit = float, single precision



16bit = half, half precision



# Floating Point Representation

| Nvidia Tesla Workstation GPU Performance Comparison |              |             |             |
|---|--------------|-------------|-------------|
|   | P100         | M40         | K40         |
| Architecture  | Pascal       | Maxwell     | Kepler      |
| Double Precision (FP64)                             | 5.3 Tflop/s  | 0.2 Tflop/s | 1.4 Tflop/s |
| Single Precision (FP32)                             | 10.6 Tflop/s | 7 Tflop/s   | 4.3 Tflop/s |
| Half Precision (FP16)                               | 21.1 Tflop/s | N/A         | N/A         |
| Memory Bandwidth                                    | 720GB/s      | 288GB/s     | 288GB/s     |
| Memory Size   | 16GB         | 12GB / 24GB | 12GB        |
| Release Date  | 2016         | Nov-15      | Nov-13      |

# Goals

“I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships.”

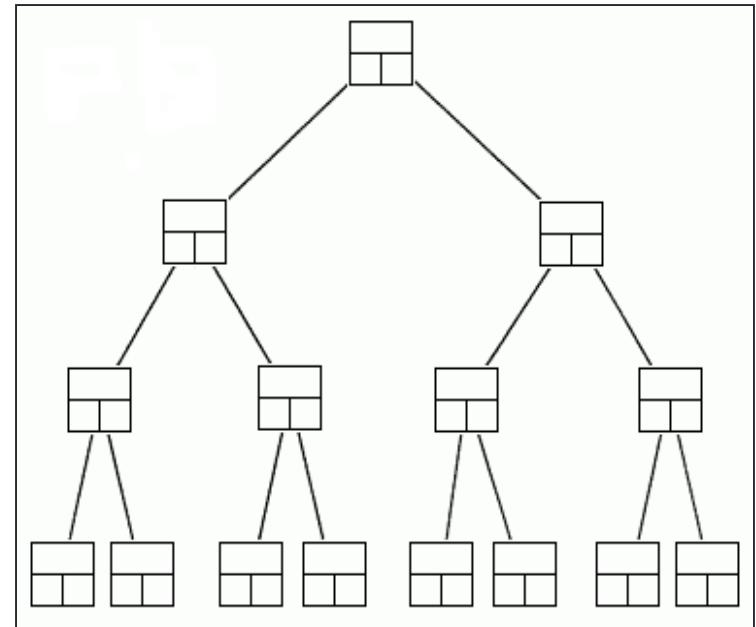
Linus Torvalds, 2006



# Data Structures

A data structure is a way to store and organize data in computer, so that it can be used efficiently.

Some of the more commonly used data structures include lists, *arrays*, *stacks*, *queues*, *heaps*, *trees*, and *graphs*.



Binary Tree

# What are data structures?

- Data structures are software artifacts that allow data to be stored, organized and accessed.
- Ultimately data structures have two core functions: put stuff in and take stuff out.

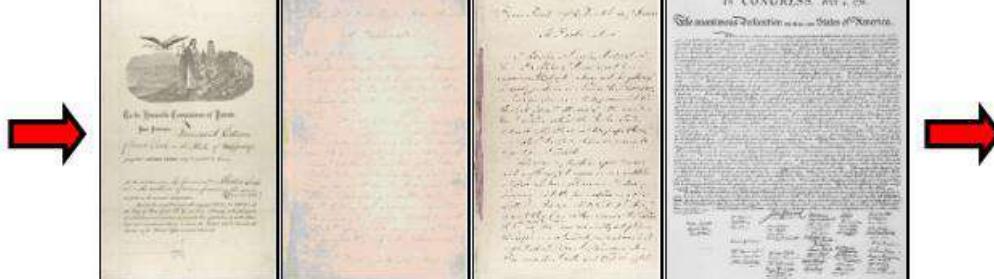
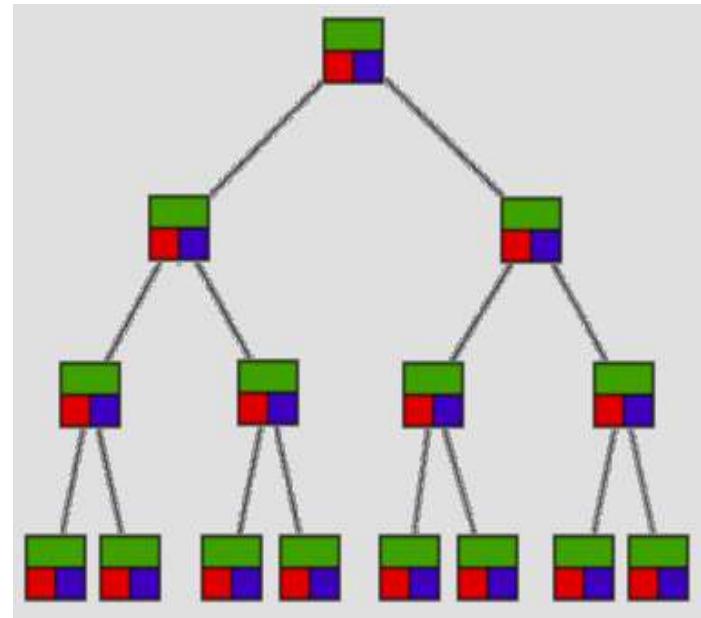
# Why so many?

- Space efficiency
- Time efficiency:
  - Store
  - Search
  - Retrieve
  - Remove
  - Clone etc.

# Choosing Data Structures

Queue vs Binary Tree

---Which one to use for what task?



# Why So Many Data Structures?

- Ideal data structure:
  - “fast”, “elegant”, memory efficient
- Generates tensions:
  - time vs. space
  - performance vs. elegance
  - generality vs. simplicity
  - one operation’s performance vs. another’s

*The study of data structures is the study of tradeoffs. That's why we have so many of them!*

# **BBM 201**

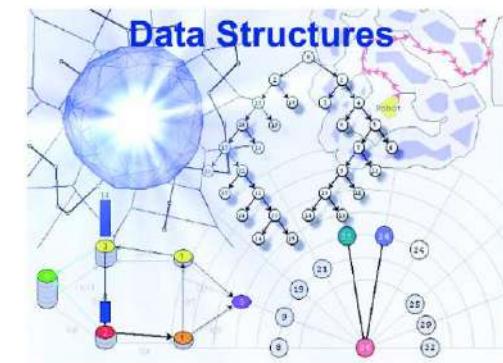
# **DATA STRUCTURES**

---

**Lecture 2:**  
**Recursion & Performance analysis**



**2018-2019 Fall**



# System Life Cycle

- Programs pass through a period called ‘system life cycle’, which is defined by the following steps:
  - **1. Requirements:** It is defined by the inputs given to the program and outputs that will be produced by the program.
  - **2. Analysis:** The first job after defining the requirements is to analyze the problem. There are two ways for this:
    - Bottom-up
    - Top-down

# System Life Cycle (cont')

- **3. Design:** Data objects and the possible operations between the objects are defined in this step. Therefore, abstract data objects and the algorithm are defined. They are all independent of the programming language.
- **4. Refinement and Coding:** Algorithms are used in this step in order to do operations on the data objects.
- **5. Verification:** Correctness proofs, testing and error removal are performed in this step.

# Cast of Characters



**Programmer** needs to develop  
a working solution.



**Client** wants to solve  
problem efficiently.



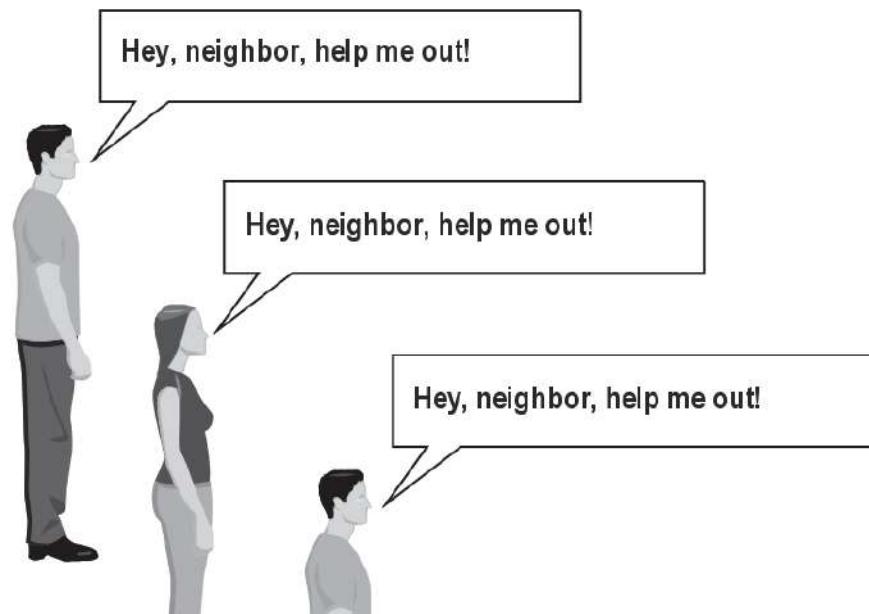
**Theoretician** wants  
to understand.



# Recursion

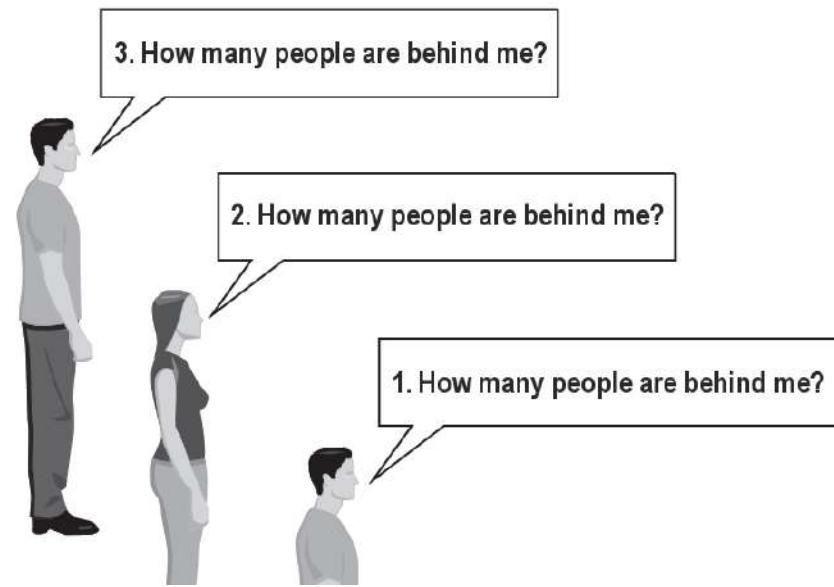
# Recursion

- Recursion is all about breaking a big problem into smaller occurrences of that same problem.
  - Each person can solve a small part of the problem.
    - What is a small version of the problem that would be easy to answer?
    - What information from a neighbor might help me?



# Recursive Algorithm

- Number of people behind me:
  - If there is someone behind me, ask him/her how many people are behind him/her.
    - When they respond with a value **N**, then I will answer **N + 1**.
  - If there is nobody behind me, I will answer **0**.



# Recursive Algorithms

- Functions can call themselves (**direct recursion**)
- A function that calls another function is called by the second function again. (**indirect recursion**)

# Recursion

- Consider the following method to print a line of \* characters:

```
// Prints a line containing the given number of stars.  
// Precondition: n >= 0  
void printStars(int n) {  
    for (int i = 0; i < n; i++) {  
        System.out.print("*");  
    }  
    System.out.println(); // end the line of output  
}
```

- Write a recursive version of this method (that calls itself).
  - Solve the problem without using any loops.
  - Hint: Your solution should print just one star at a time.

# A basic case

- What are the cases to consider?
  - What is a very easy number of stars to print without a loop?

```
public static void printStars(int n) {  
    if (n == 1) {  
        // base case; just print one star  
        System.out.println("*");  
    } else {  
        ...  
    }  
}
```

# Handling more cases

- Handling additional cases, with no loops (in a bad way):

```
public static void printStars(int n) {  
    if (n == 1) {  
        // base case; just print one star  
        System.out.println("*");  
    } else if (n == 2) {  
        System.out.print("*");  
        System.out.println("*");  
    } else if (n == 3) {  
        System.out.print("*");  
        System.out.print("*");  
        System.out.println("*");  
    } else if (n == 4) {  
        System.out.print("*");  
        System.out.print("*");  
        System.out.print("*");  
        System.out.println("*");  
    } else ...  
}
```

# Handling more cases 2

- Taking advantage of the repeated pattern (somewhat better):

```
public static void printStars(int n) {  
    if (n == 1) {  
        // base case; just print one star  
        System.out.println("*");  
    } else if (n == 2) {  
        System.out.print("*");  
        printStars(1);      // prints "*"  
    } else if (n == 3) {  
        System.out.print("*");  
        printStars(2);      // prints "**"  
    } else if (n == 4) {  
        System.out.print("*");  
        printStars(3);      // prints "***"  
    } else ...  
}
```

# Using recursion properly

- Condensing the recursive cases into a single case:

```
public static void printStars(int n) {  
    if (n == 1) {  
        // base case; just print one star  
        System.out.println("*");  
    } else {  
        // recursive case; print one more star  
        System.out.print("*");  
        printStars(n - 1);  
    }  
}
```

# Even simpler

- The real, even simpler, base case is an n of 0, not 1:

```
public static void printStars(int n) {  
    if (n == 0) {  
        // base case; just end the line of output  
        System.out.println();  
    } else {  
        // recursive case; print one more star  
        System.out.print("*");  
        printStars(n - 1);  
    }  
}
```

# Recursive tracing

- Consider the following recursive method:

```
public static int mystery(int n) {  
    if (n < 10) {  
        return n;  
    } else {  
        int a = n / 10;  
        int b = n % 10;  
        return mystery(a + b);  
    }  
}
```

- What is the result of the following call?

```
mystery(648)
```

# A recursive trace

mystery(648) :

- int a = 648 / 10; // 64
- int b = 648 % 10; // 8
- return mystery(a + b); // mystery(72)

mystery(72) :

- int a = 72 / 10; // 7
- int b = 72 % 10; // 2
- return mystery(a + b); // mystery(9)

mystery(9) :

- return 9;

# Recursive tracing 2

- Consider the following recursive method:

```
public static int mystery(int n) {  
    if (n < 10) {  
        return (10 * n) + n;  
    } else {  
        int a = mystery(n / 10);  
        int b = mystery(n % 10);  
        return (100 * a) + b;  
    }  
}
```

- What is the result of the following call?

mystery(348)

# A recursive trace 2

mystery(348)

- int a = mystery(34);
  - int a = mystery(3);  
return (10 \* 3) + 3; // 33
  - int b = mystery(4);  
return (10 \* 4) + 4; // 44
  - return (100 \* 33) + 44; // 3344
  
- int b = mystery(8);  
return (10 \* 8) + 8; // 88
  
- return (100 \* 3344) + 88; // 334488

- What is this method really doing?

# Selection Sort

- Let's sort an array that consists of randomly inserted items.

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

# Selection Sort

```
#include <stdio.h>
#include <math.h>
#define MAX_SIZE 101
#define SWAP(x, y, z) (z = x; x = y; y = z)
void sort(int [], int);

void main()
{
    int i, n;
    int list[MAX_SIZE];
    printf("array size");
    scanf("%d", &n);

    for(i = 0; i < n; i++){
        list[i] = rand() % 100;
        printf("%d ", list[i]);
    }
    sort(list,n);
}
```

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

# SWAP

- prototype:

```
void swap(int *, int *)  
  
void swap(int *x, int *y)  
{  
    int temp = *x;  
    *x = *y ;  
    *y = temp;  
}
```

# Binary Search

- A value is searched in a sorted array. If found, the index of the item is returned, otherwise -1 is returned.

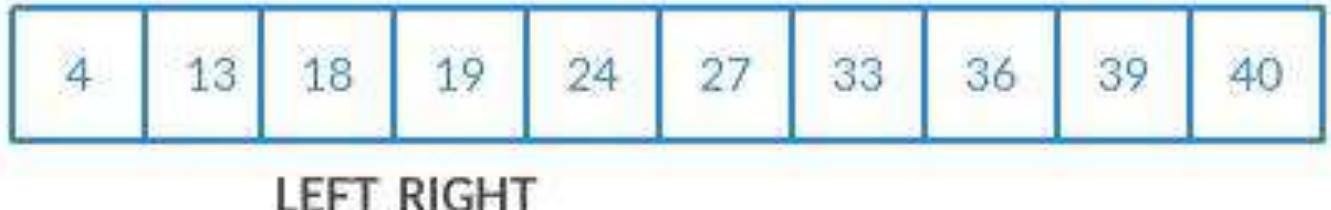
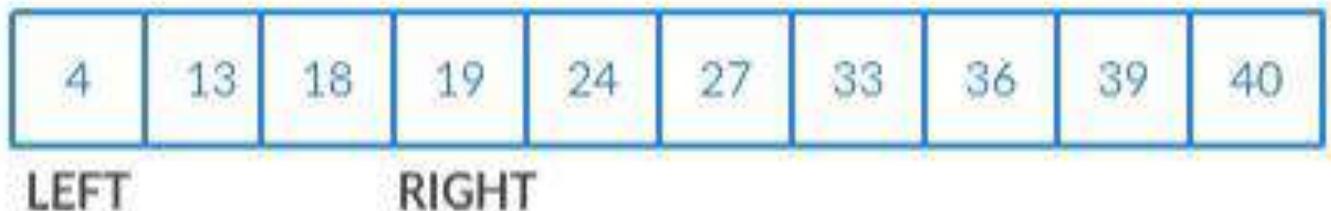
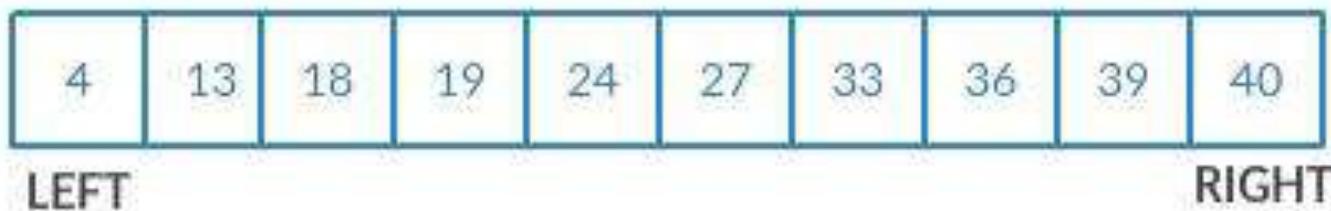
```
int binsearch(int list[], int searchnum, int left, int right)
{
    int middle;
    while(left <= right){
        middle = (left + right)/2;
        switch(compare(list[middle], searchnum)){
            case -1 : left = middle + 1; break;
            case 0; return middle;
            case 1: right = middle -1;
        }
    }
    return -1;
}
```

- Macro for ‘COMPARE’:

```
#define COMPARE(x,y) ((x)<(y))? -1 ((x)==(y))?0:1)
```

# Binary Search

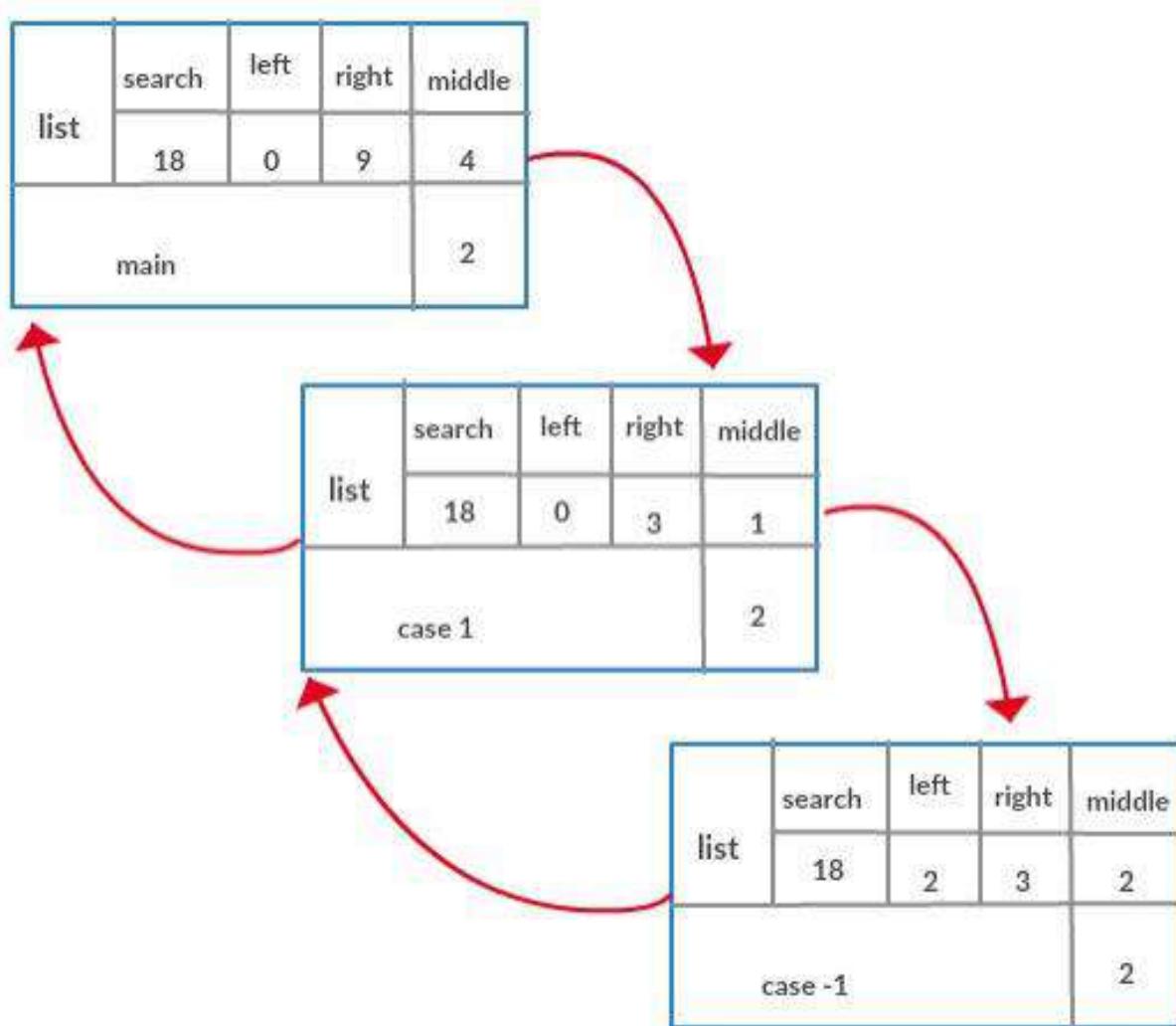
## Let's search for 18:



# Binary Search (revisited)

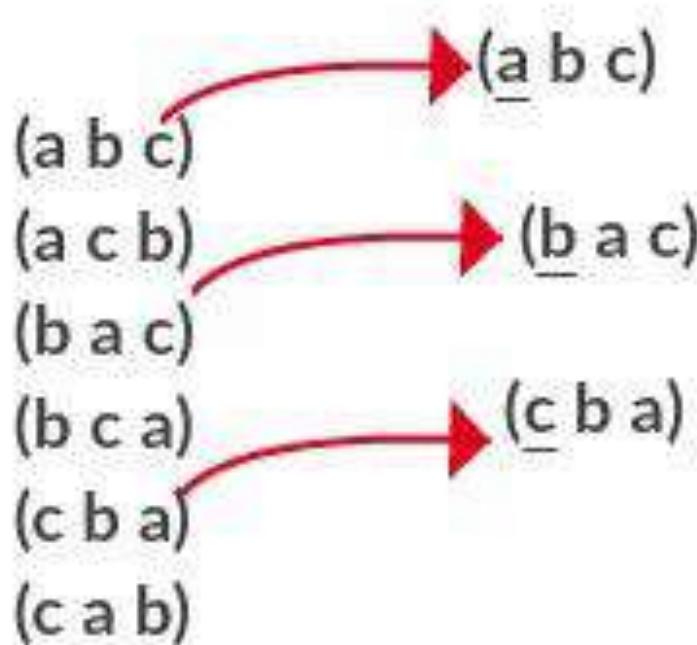
```
int binsearch(int list[], int searchnum, int left, int right)
{
    int middle;
    if(left<=right) {
        middle = (left+right)/2;
        switch(COMPARE(list[middle], searchnum)) {
            case -1: return binsearch(list, searchnum, middle+1, right);
            case 0: return middle;
            case 1: return binsearch(list, searchnum, left, middle-1);
        }
    }
    return -1;
}
```

# Binary Search (revisited)



# Permutation problem

- Finding all the permutations of a given set with size  $n \geq 1$ .
  - Remember there are  $n!$  different sequences of this set.
  - Example: Find all the permutations of  $\{a,b,c\}$



# Permutation problem

- Example: Find all the permutations of {a,b,c,d}
- All the permutations of {b,c,d} follow ‘a’
- All the permutations of {a,c,d} follow ‘b’
- All the permutations of {a,b,d} follow ‘c’
- All the permutations of {a,b,c} follow ‘d’

# Factorial Function

- $n! = n * (n - 1) * (n - 2) * \dots * 2 * 1$  for any integer  $n > 0$

$$0! = 1$$

- **Iterative Definition:**

- 

```
int fval=1;  
  
for(i=n; i>=1; i--)  
    fval = fval*i;
```

# Factorial Function

## Recursive Definition

- To define  $n!$  recursively,  $n!$  must be defined in terms of the factorial of a smaller number.
- Observation (problem size is reduced):

$$n! = n * (n - 1)!$$

- Base case:

$$0! = 1$$

- We can reach the base case by subtracting 1 from  $n$  if  $n$  is a positive integer.

# Factorial Function

## Recursive Definition

### Recursive definition

$$n! = 1 \quad \text{if } n = 0$$

$$n! = n * (n - 1)! \quad \text{if } n > 0$$

```
int fact(int n)
{
    if(n==0)
        return (1);
    else
        return (n*fact(n-1));
}
```

This fact function satisfies the four criteria of a recursive solution.

# Four Criteria of a Recursive Solution

1. A recursive function calls itself.
  - This action is what makes the solution recursive.
2. Each recursive call solves an identical, but smaller, problem.
  - A recursive function solves a problem by solving another problem that is identical in nature but smaller in size.
3. A test for the base case enables the recursive calls to stop.
  - There must be a case of the problem (known as *base case* or *stopping case*) that is handled differently from the other cases (without recursively calling itself.)
  - In the base case, the recursive calls stop and the problem is solved directly.
4. Eventually, one of the smaller problems must be the base case.
  - The manner in which the size of the problem diminishes ensures that the base case is eventually reached.

# Fibonacci Sequence

- It is the sequence of integers:

$$t_0 \quad t_1 \quad t_2 \quad t_3 \quad t_4 \quad t_5 \quad t_6$$

$$0 \quad 1 \quad 1 \quad 2 \quad 3 \quad 5 \quad 8 \quad \dots$$

- Each element in this sequence is the sum of the two preceding elements.
- The specification of the terms in the Fibonacci sequence:

$$t_n = \begin{cases} n & \text{if } n \text{ is 0 or 1 (i.e. } n < 2) \\ t_{n-1} + t_{n-2} & \text{otherwise} \end{cases}$$

# Fibonacci Sequence

- Iterative solution

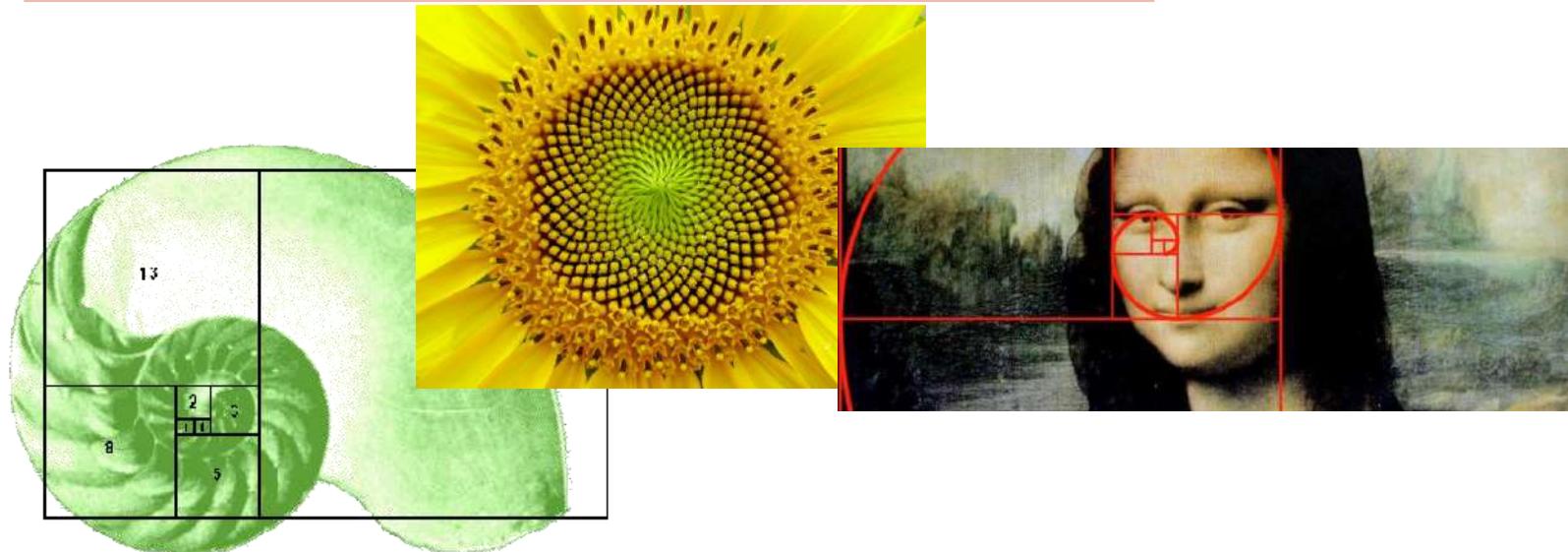
```
int Fib(int n)
{
    int prev1, prev2, temp, j;

    if(n==0 || n==1)
        return n;
    else{
        prev1=0;
        prev2=1;
        for(j=1;j<=n;j++)
        {
            temp = prev1+prev2;
            prev1=prev2;
            prev2=temp;
        }
        return prev2;
    }
}
```

# Fibonacci Sequence

- Recursive solution

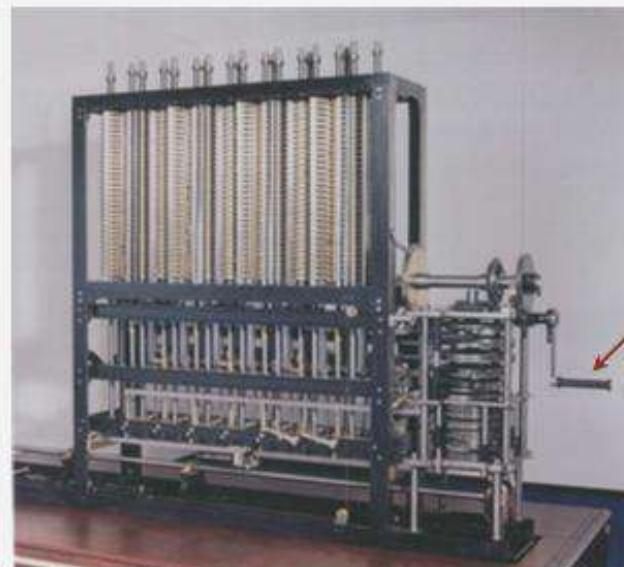
```
int fibonacci(int n)
{
    if(n<2)
        return n;
    else
        return (fibonacci(n-2)+fibonacci(n-1));
}
```



# Performance Analysis

# Running Time

*"As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise—By what course of calculation can these results be arrived at by the machine in the shortest time?"* — Charles Babbage (1864)



Analytic Engine

# Observations

## Example-1

3-SUM. Given  $N$  distinct integers, how many triples sum to exactly zero?

```
% more 8ints.txt  
8  
30 -40 -20 -10 40 0 10 5  
  
% java ThreeSum 8ints.txt  
4
```

|   | a[i] | a[j] | a[k] | sum |
|---|------|------|------|-----|
| 1 | 30   | -40  | 10   | 0   |
| 2 | 30   | -20  | -10  | 0   |
| 3 | -40  | 40   | 0    | 0   |
| 4 | -10  | 0    | 10   | 0   |



Context. Deeply related to problems in computational geometry.

# Observations

## Example-1: 3-SUM brute force algorithm

```
public class ThreeSum
{
    public static int count(int[] a)
    {
        int N = a.length;
        int count = 0;
        for (int i = 0; i < N; i++)
            for (int j = i+1; j < N; j++)
                for (int k = j+1; k < N; k++) ← check each triple
                    if (a[i] + a[j] + a[k] == 0) ← for simplicity, ignore
                        count++;                   integer overflow
        return count;
    }

    public static void main(String[] args)
    {
        In in = new In(args[0]);
        int[] a = in.readAllInts();
        StdOut.println(count(a));
    }
}
```

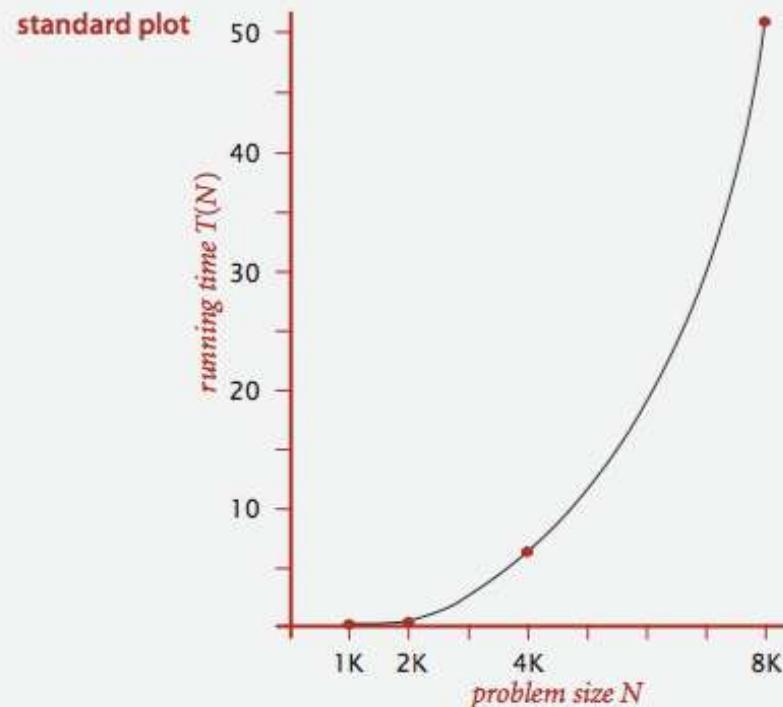
# Empirical Analysis

Run the program for various input sizes and measure running time.

| N      | time (seconds) † |
|--------|------------------|
| 250    | 0.0              |
| 500    | 0.0              |
| 1,000  | 0.1              |
| 2,000  | 0.8              |
| 4,000  | 6.4              |
| 8,000  | 51.1             |
| 16,000 | ?                |

# Data Analysis

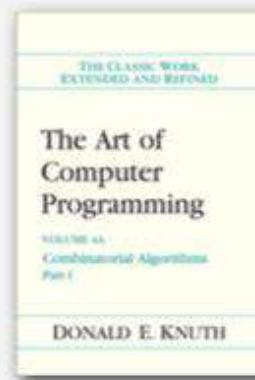
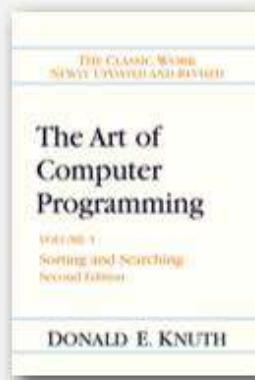
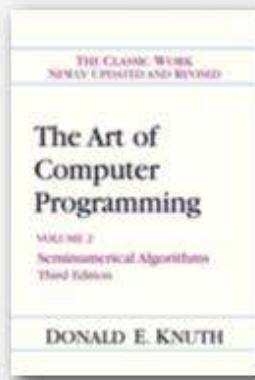
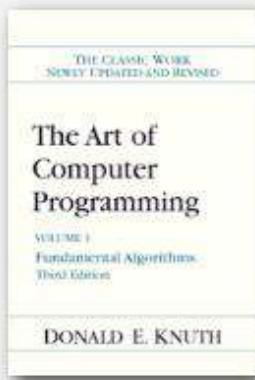
Standard plot. Plot running time  $T(N)$  vs. input size  $N$ .



# Mathematical Models for Running Time

Total running time: sum of cost  $\times$  frequency for all operations.

- Need to analyze program to determine set of operations.
- Cost depends on machine, compiler.
- Frequency depends on algorithm, input data.



Donald Knuth  
1974 Turing Award

In principle, accurate mathematical models are available.

# Cost of Basic Operations

Challenge. How to estimate constants.

| operation               | example                       | nanoseconds † |
|-------------------------|-------------------------------|---------------|
| integer add             | $a + b$                       | 2.1           |
| integer multiply        | $a * b$                       | 2.4           |
| integer divide          | $a / b$                       | 5.4           |
| floating-point add      | $a + b$                       | 4.6           |
| floating-point multiply | $a * b$                       | 4.2           |
| floating-point divide   | $a / b$                       | 13.5          |
| sine                    | <code>Math.sin(theta)</code>  | 91.3          |
| arctangent              | <code>Math.atan2(y, x)</code> | 129.0         |
| ...                     | ...                           | ...           |

† Running OS X on Macbook Pro 2.2GHz with 2GB RAM

# Cost of Basic Operations

**Observation.** Most primitive operations take constant time.

| operation            | example                    | nanoseconds † |
|----------------------|----------------------------|---------------|
| variable declaration | <code>int a</code>         | $c_1$         |
| assignment statement | <code>a = b</code>         | $c_2$         |
| integer compare      | <code>a &lt; b</code>      | $c_3$         |
| array element access | <code>a[i]</code>          | $c_4$         |
| array length         | <code>a.length</code>      | $c_5$         |
| 1D array allocation  | <code>new int[N]</code>    | $c_6 N$       |
| 2D array allocation  | <code>new int[N][N]</code> | $c_7 N^2$     |

**Caveat.** Non-primitive operations often take more than constant time.

novice mistake: abusive string concatenation

# Example: 1-Sum

Q. How many instructions as a function of input size  $N$ ?

```
int count = 0;  
for (int i = 0; i < N; i++)  
    if (a[i] == 0)  
        count++;
```

N array accesses

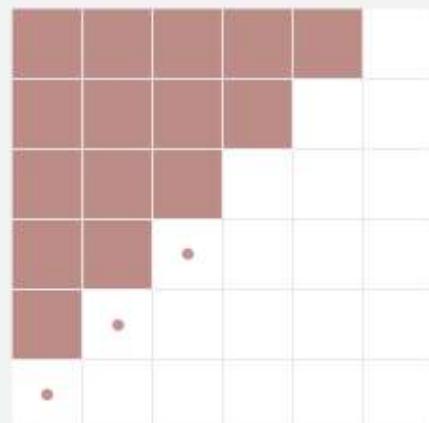
| operation            | frequency   |
|----------------------|-------------|
| variable declaration | 2           |
| assignment statement | 2           |
| less than compare    | $N + 1$     |
| equal to compare     | $N$         |
| array access         | $N$         |
| increment            | $N$ to $2N$ |

# Example: 2-Sum

Q. How many instructions as a function of input size  $N$ ?

```
int count = 0;  
for (int i = 0; i < N; i++)  
    for (int j = i+1; j < N; j++)  
        if (a[i] + a[j] == 0)  
            count++;
```

Pf. [n even]



$$\begin{aligned}0 + 1 + 2 + \dots + (N-1) &= \frac{1}{2}N(N-1) \\&= \binom{N}{2}\end{aligned}$$

$$0 + 1 + 2 + \dots + (N-1) = \frac{1}{2}N^2 - \frac{1}{2}N$$

half of      half of  
square          diagonal

# Example: 2-Sum

Q. How many instructions as a function of input size  $N$ ?

```
int count = 0;  
for (int i = 0; i < N; i++)  
    for (int j = i+1; j < N; j++)  
        if (a[i] + a[j] == 0)  
            count++;
```

$$\begin{aligned}0 + 1 + 2 + \dots + (N-1) &= \frac{1}{2}N(N-1) \\&= \binom{N}{2}\end{aligned}$$

| operation            | frequency                       |
|----------------------|---------------------------------|
| variable declaration | $N + 2$                         |
| assignment statement | $N + 2$                         |
| less than compare    | $\frac{1}{2}(N+1)(N+2)$         |
| equal to compare     | $\frac{1}{2}N(N-1)$             |
| array access         | $N(N-1)$                        |
| increment            | $\frac{1}{2}N(N-1)$ to $N(N-1)$ |



tedious to count exactly

# Simplifying the Calculations

*"It is convenient to have a measure of the amount of work involved in a computing process, even though it be a very crude one. We may count up the number of times that various elementary operations are applied in the whole process and then given them various weights. We might, for instance, count the number of additions, subtractions, multiplications, divisions, recording of numbers, and extractions of figures from tables. In the case of computing with matrices most of the work consists of multiplications and writing down numbers, and we shall therefore only attempt to count the number of multiplications and recordings." — Alan Turing*

## ROUNDING-OFF ERRORS IN MATRIX PROCESSES

By A. M. TURING

(National Physical Laboratory, Teddington, Middlesex)

[Received 4 November 1947]

### SUMMARY

A number of methods of solving sets of linear equations and inverting matrices are discussed. The theory of the rounding-off errors involved is investigated for some of the methods. In all cases examined, including the well-known 'Gauss elimination process', it is found that the errors are normally quite moderate: no exponential build-up need occur.



# Simplification: cost model

Cost model. Use some basic operation as a proxy for running time.

```
int count = 0;
for (int i = 0; i < N; i++)
    for (int j = i+1; j < N; j++)
        if (a[i] + a[j] == 0)
            count++;
```

$$\begin{aligned}0 + 1 + 2 + \dots + (N-1) &= \frac{1}{2}N(N-1) \\&= \binom{N}{2}\end{aligned}$$

| operation            | frequency                       |
|----------------------|---------------------------------|
| variable declaration | $N+2$                           |
| assignment statement | $N+2$                           |
| less than compare    | $\frac{1}{2}(N+1)(N+2)$         |
| equal to compare     | $\frac{1}{2}N(N-1)$             |
| array access         | $N(N-1)$                        |
| increment            | $\frac{1}{2}N(N-1)$ to $N(N-1)$ |

cost model = array accesses

(we assume compiler/JVM do not optimize any array accesses away)

# Asymptotic Notation: 2-SUM problem

Q. Approximately how many array accesses as a function of input size  $N$ ?

```
int count = 0;
for (int i = 0; i < N; i++)
    for (int j = i+1; j < N; j++)
        if (a[i] + a[j] == 0)
            count++;
```

"inner loop"

$$\begin{aligned} 0 + 1 + 2 + \dots + (N - 1) &= \frac{1}{2} N(N - 1) \\ &= \binom{N}{2} \end{aligned}$$

A.  $\sim N^2$  array accesses.

# Asymptotic Notation: 3-SUM problem

Q. Approximately how many array accesses as a function of input size  $N$ ?

```
int count = 0;
for (int i = 0; i < N; i++)
    for (int j = i+1; j < N; j++)
        for (int k = j+1; k < N; k++)
            if (a[i] + a[j] + a[k] == 0) ← "inner loop"
                count++;
```

A.  $\sim \frac{1}{2}N^3$  array accesses.

$$\binom{N}{3} = \frac{N(N-1)(N-2)}{3!}$$
$$\sim \frac{1}{6}N^3$$

# Binary search (java implementation)

Invariant. If key appears in array  $a[]$ , then  $a[lo] \leq \text{key} \leq a[hi]$ .

```
public static int binarySearch(int[] a, int key)
{
    int lo = 0, hi = a.length - 1;
    while (lo <= hi)           why not mid = (lo + hi) / 2 ?
    {
        int mid = lo + (hi - lo) / 2;
        if      (key < a[mid]) hi = mid - 1;
        else if (key > a[mid]) lo = mid + 1;
        else return mid;
    }
    return -1;
}
```

one "3-way compare"

# Binary search: mathematical analysis

**Proposition.** Binary search uses at most  $1 + \lg N$  key compares to search in a sorted array of size  $N$ .

**Def.**  $T(N) = \# \text{ key compares to binary search a sorted subarray of size } \leq N$ .

**Binary search recurrence.**  $T(N) \leq T(N/2) + 1$  for  $N > 1$ , with  $T(1) = 1$ .

$\uparrow$                                $\uparrow$   
left or right half                      possible to implement with one  
(floored division)                      2-way compare (instead of 3-way)

**Pf sketch.** [assume  $N$  is a power of 2]

$$\begin{aligned} T(N) &\leq T(N/2) + 1 && [\text{given}] \\ &\leq T(N/4) + 1 + 1 && [\text{apply recurrence to first term}] \\ &\leq T(N/8) + 1 + 1 + 1 && [\text{apply recurrence to first term}] \\ &\vdots \\ &\leq T(N/N) + 1 + 1 + \dots + 1 && [\text{stop applying, } T(1) = 1] \\ &= 1 + \lg N && \underbrace{\hspace{2cm}}_{\lg N} \end{aligned}$$

# Types of Analyses

**Best case :** Lower bound on cost.

- Determined by “easiest” input.

- Provides a guarantee for all inputs.

**Worst case :** Upper bound on cost.

- Determined by “most difficult” input..

- Provides a guarantee for all inputs.

**Average case :** Expected cost for random input.

- Need a model for “random” input.

- Provides a way to predict performance.

# Types of Analyses

Ex 1. Array accesses for brute-force 3-SUM.

Best:  $\sim \frac{1}{2} N^3$

Average:  $\sim \frac{1}{2} N^3$

Worst:  $\sim \frac{1}{2} N^3$

Ex 2. Comparisons for binary search.

Best:  $\sim 1$

Average:  $\sim \lg N$

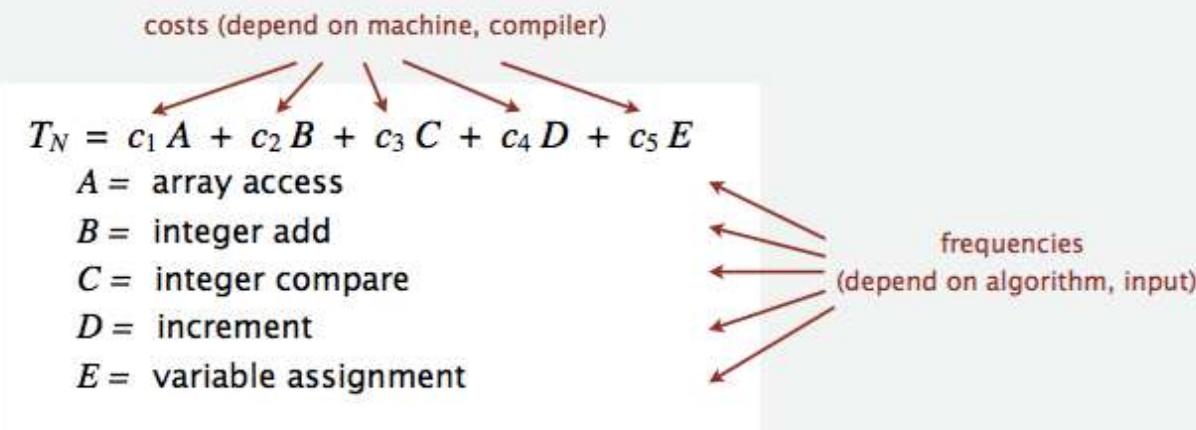
Worst:  $\sim \lg N$

# Mathematical Models for Running Time

In principle, accurate mathematical models are available.

In practice,

- Formulas can be complicated.
- Advanced mathematics might be required.
- Exact models best left for experts.



Bottom line. We use approximate models in this course:  $T(N) \sim c N^3$ .

# Asymptotic notation: O (Big Oh)

$f(n) = O(g(n))$  iff there exist positive constants  $c$  and  $n_0$  such that

$$f(n) \leq cg(n) \text{ for all } n, n \geq n_0$$

$$3n+2 = O(n) \quad \text{as } 3n+2 \leq 4n \quad \text{for all } n \geq 2$$

$$10n^2 + 4n + 2 = O(n^2) \quad \text{as } 10n^2 + 4n + 3 \leq 11n^2 \quad \text{for } n \geq 5$$

$$6*2^n + n^2 = O(2^n) \quad \text{as } 6*2^n + n^2 \leq 7*2^n \quad \text{for } n \geq 4$$

$$3n+3 = O(n) \quad \text{Correct, OK.}$$

$$3n+3 = O(n^2) \quad \text{Correct, NO!}$$

# Question

How many array accesses does the following code fragment make as a function of  $N$ ?

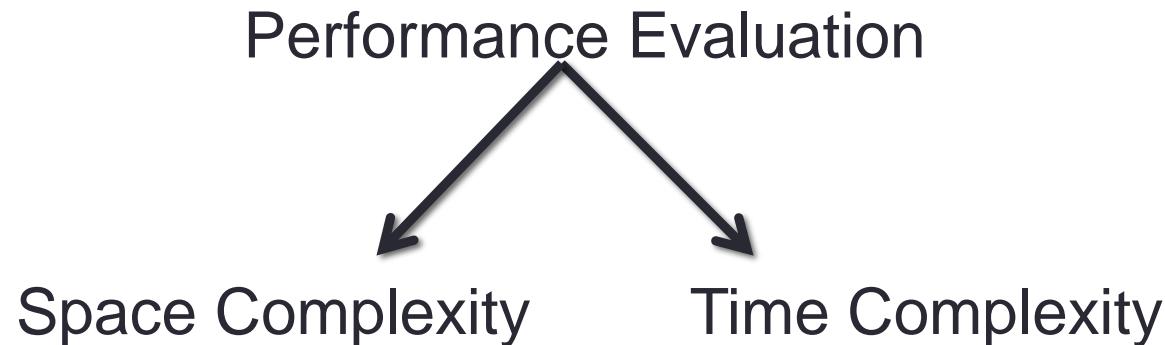
```
int count = 0;
for (int i = 0; i < N; i++)
    for (int j = i+1; j < N; j++)
        for (int k = 1; k < N; k = k*2)
            if (a[i] + a[j] >= a[k])
                count++;
```

- A.  $\sim 3N^2$
- B.  $\sim 3/2 N^2 \lg N$
- C.  $\sim 3/2 N^3$
- D.  $\sim 3 N^3$
- E. *I don't know.*

# Performance Analysis

- There are various criteria in order to evaluate a program.
- The questions:
  - Does the program meet the requirements determined at the beginning?
  - Does it work correctly?
  - Does it have any documentation on the subject of how it works?
  - Does it use the function effectively in order to produce the logical values: TRUE and FALSE?
  - Is the program code readable?
  - Does the program use the primary and the secondary memory effectively?
  - Is the running time of the program acceptable?

# Performance Analysis



# Space Complexity

- Memory space needed by a program to complete its execution:
  - A fixed part →  $c$
  - A variable part →  $S_p(\text{instance})$

$$S(P) = c + S_p(\text{instance})$$

```
float abc(float a, float b, float c){  
    return a+b+b*c+(a+b+c)/(b+c);  
}
```

$$S_{abc}(I)=0$$

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

$$S_{\text{sum}}(n)=0$$

# Space Complexity

- Recursive function call:

```
float rsum(float list[], int n){  
    if(n)  
        return rsum(list, n-1)+list[n-1];  
    return 0;  
}
```

- What is the total variable memory space of this method for an input size of 2000?

# Time Complexity

- Total time used by the program is:  $T(P)$ 
  - Where  $T(P)=\text{compile time} + \text{run (execution) time } (T_P)$

```
float sum(float list[], int n){  
    float tempsum=0; ..... 1  
    int i;  
    for(i=0;i<n;i++){ ..... n+1  
        tempsum += list[i]; ..... n  
    return tempsum; ..... 1  
}  
  
Total step number = 2n+3
```

# Time Complexity

```
float rsum(float list[], int n){  
    if(n)  
        return rsum(list, n-1)+list[n-1];  
    return 0;  
}
```

- What is the number of steps required to execute this method?

$$2n+2$$

# Time Complexity

```
void add(int a[MAX_SIZE]){
    int i,j;
    for(i=0; i<rows; i++)
        for(j=0; j<cols; j++)
            c[i][j]=a[i][j]+b[i][j];
```

- What is the number of steps required to execute this method?

$$2\text{rows} * \text{cols} + 2\text{rows} + 1$$

# O (Big Oh) Notation

- Time complexity = algorithm complexity
- $f(n) < cg(n) \rightarrow f(n) = O(g(n))$
- $T_{\text{sum}}(n) = 2n + 3 \rightarrow T_{\text{sum}}(n) = O(n)$
- $T_{\text{rsum}}(n) = 2n + 2 \rightarrow T_{\text{sum}}(n) = O(n)$
- $T_{\text{add}}(\text{rows}, \text{cols}) = 2\text{rows} * \text{cols} + 2\text{rows} + 1 \rightarrow T_{\text{add}}(n) = O(\text{rows} * \text{cols}) = O(n^2)$

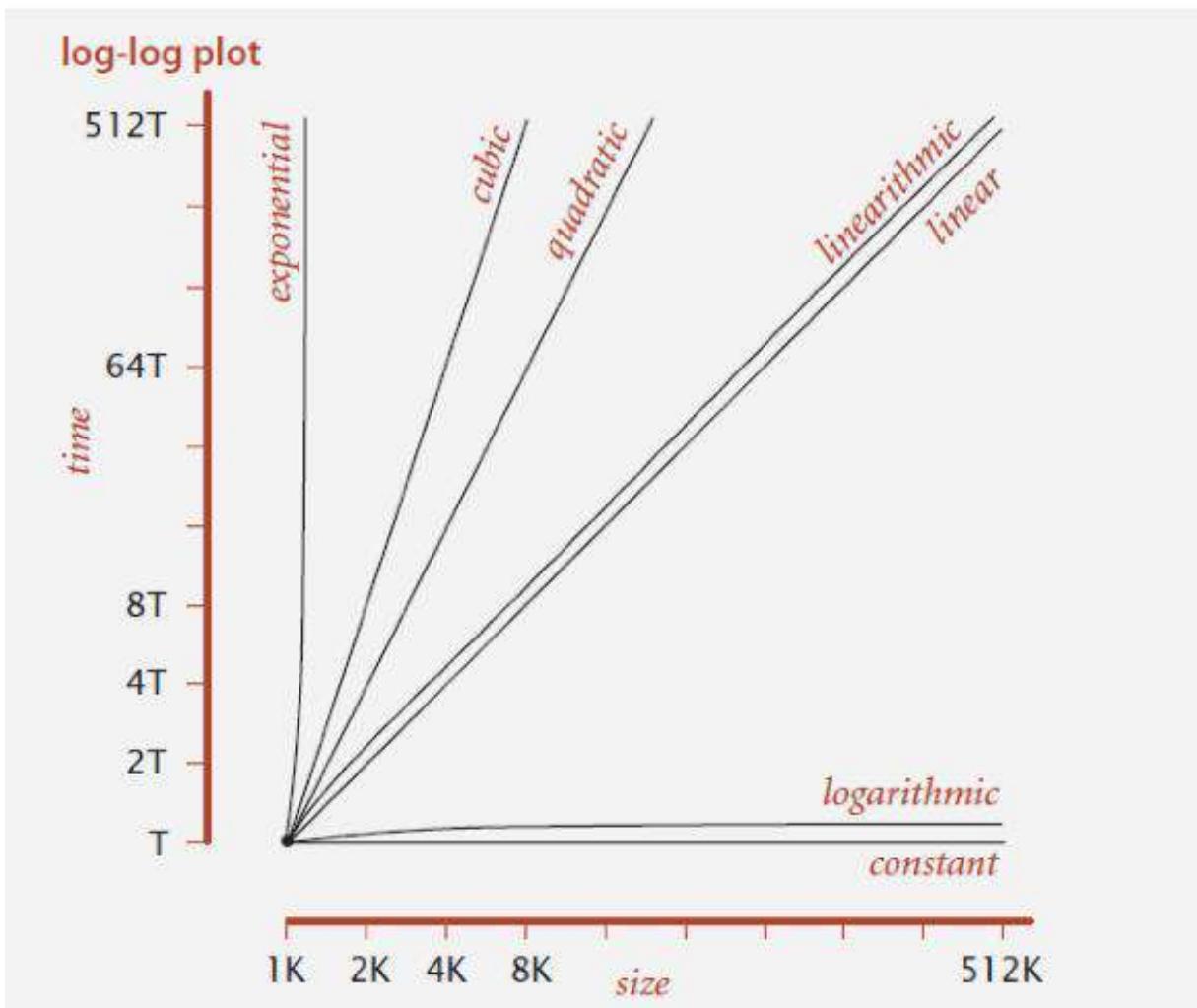
# Asymptotic Notation

- Example:  $f(n)=3n^3+6n^2+7n$
- What is the algorithm complexity in O notation?

# Practical Complexities

| Complexity<br>(time) | Name        | 1 | 2 | 4  | 8     | 16             | 32                     |
|----------------------|-------------|---|---|----|-------|----------------|------------------------|
| 1                    | constant    | 1 | 1 | 1  | 1     | 1              | 1                      |
| log n                | logarithmic | 0 | 1 | 2  | 3     | 4              | 5                      |
| n                    | linear      | 1 | 2 | 4  | 8     | 16             | 32                     |
| n log n              | log linear  | 0 | 2 | 8  | 24    | 64             | 160                    |
| $n^2$                | quadratic   | 1 | 4 | 16 | 64    | 256            | 1024                   |
| $n^3$                | cubic       | 1 | 8 | 64 | 512   | 4096           | 32768                  |
| $2^n$                | exponential | 2 | 4 | 16 | 256   | 65536          | 4294967296             |
| $n!$                 | factorial   | 1 | 2 | 24 | 40326 | 20922789888000 | $26313 \times 10^{33}$ |

# Practical Complexities



| $n$       | $f(n)=n$ | $f(n)=\log_2 n$ | $f(n)=n^2$ | $f(n)=n^3$ | $f(n)=n^4$            | $f(n)=n^{10}$            | $f(n)=2^n$            |
|-----------|----------|-----------------|------------|------------|-----------------------|--------------------------|-----------------------|
| 10        | 0.01μ    | 0.03μ           | 0.1μ       | 1μ         | 10μ                   | 10 sec                   | 1μ                    |
| 20        | 0.02μ    | 0.09μ           | 0.4μ       | 8μ         | 160μ                  | 2.84 hr                  | 1 ms                  |
| 30        | 0.03μ    | 0.15μ           | 0.9μ       | 27μ        | 810μ                  | 6.83 d                   | 1 sec                 |
| 40        | 0.04μ    | 0.21μ           | 1.6μ       | 64μ        | 2.56ms                | 121.36 d                 | 18.3 min              |
| 50        | 0.05μ    | 0.28μ           | 2.5μ       | 125μ       | 6.25ms                | 3.1 yr                   | 13 d                  |
| 100       | 0.10μ    | 0.66μ           | 10μ        | 1ms        | 100ms                 | 3171 yr                  | $4 \times 10^{13}$ yr |
| 1000      | 1μ       | 9.96μ           | 1ms        | 1sec       | 16.67min              | $3.17 \times 10^{13}$ yr |                       |
| 10,000    | 10μ      | 130.03μ         | 100ms      | 16.67min   | 115.7d                | $3.17 \times 10^{23}$ yr |                       |
| 100,000   | 100μ     | 1.66ms          | 10sec      | 11.57d     | 3171yr                | $3.17 \times 10^{33}$ yr |                       |
| 1,000,000 | 1ms      | 19.92ms         | 16.67min   | 31.71yr    | $3.17 \times 10^7$ yr | $3.17 \times 10^{43}$ yr |                       |

# **Survey of Common Running Times**

# Tilde notation

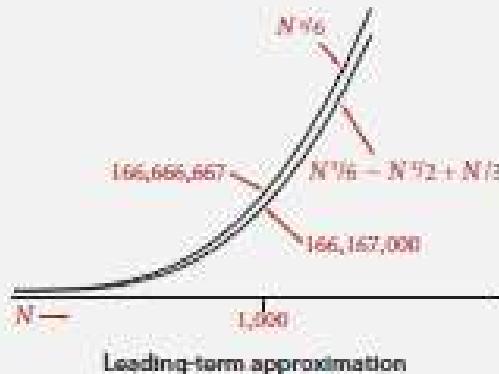
- Estimate running time (or memory) as a function of input size  $N$ .
- Ignore lower order terms.
  - when  $N$  is large, terms are negligible
  - when  $N$  is small, we don't care

Ex 1.  $\frac{1}{6}N^3 + 20N + 16 \sim \frac{1}{6}N^3$

Ex 2.  $\frac{1}{6}N^3 + 100N^{4/3} + 56 \sim \frac{1}{6}N^3$

Ex 3.  $\frac{1}{6}N^3 - \underbrace{\frac{1}{2}N^2}_{\text{discard lower-order terms}} + \frac{1}{3}N \sim \frac{1}{6}N^3$

(e.g.,  $N = 1000$ : 166.67 million vs. 166.17 million)



Technical definition.  $f(N) \sim g(N)$  means  $\lim_{N \rightarrow \infty} \frac{f(N)}{g(N)} = 1$

# Tilde Notation ( $\sim$ ) (same as $\Theta$ -notation)

- Estimate running time (or memory) as a function of input size  $N$ .
- Ignore lower order terms.
  - when  $N$  is large, terms are negligible
  - when  $N$  is small, we don't care

| operation            | frequency                              | tilde notation                       |
|----------------------|--|--------------------------------------|
| variable declaration | $N + 2$                                | $\sim N$                             |
| assignment statement | $N + 2$                                | $\sim N$                             |
| less than compare    | $\frac{1}{2} (N + 1) (N + 2)$          | $\sim \frac{1}{2} N^2$               |
| equal to compare     | $\frac{1}{2} N (N - 1)$                | $\sim \frac{1}{2} N^2$               |
| array access         | $N (N - 1)$                            | $\sim N^2$                           |
| increment            | $\frac{1}{2} N (N - 1)$ to $N (N - 1)$ | $\sim \frac{1}{2} N^2$ to $\sim N^2$ |

# Asymptotic notation: $\Omega$ (Omega)

$f(n) = \Omega(g(n))$  iff there exist positive constants  $c$  and  $n_0$  such that

$$f(n) \geq cg(n) \text{ for all } n, n \geq n_0$$

$$3n+2 = \Omega(n) \quad \text{as } 3n+2 \geq 3n \quad \text{for all } n \geq 1$$

$$10n^2 + 4n + 2 = \Omega(n^2) \quad \text{as } 10n^2 + 4n + 2 \geq n^2 \quad \text{for } n \geq 1$$

$$6*2^n + n^2 = \Omega(2^n) \quad \text{as } 6*2^n + n^2 \geq 2^n \quad \text{for } n \geq 1$$

$$3n+3 = \Omega(n) \quad \text{Correct, OK.}$$

$$3n+3 = \Omega(1) \quad \text{Correct, NO!}$$

# Asymptotic notation: $\Theta$ (theta) (same as “ $\sim$ ” notation)

$f(n) = O(g(n))$  iff there exist positive constants  $c_1, c_2$ , and  $n_0$  such that

$$c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n, n \geq n_0$$

$$3n+2 = \Theta(n) \quad \text{as } 3n+2 \geq 3n \quad \text{for all } n \geq 1$$

$$10n^2 + 4n + 2 = \Theta(n^2) \quad \text{as } 10n^2 + 4n + 2 \geq n^2 \quad \text{for } n \geq 1$$

$$6*2^n + n^2 = \Theta(2^n) \quad 6*2^n + n^2 \geq 2^n \quad \text{for } n \geq 1$$

$$3n+3 = \Omega(n) \quad \text{Correct, OK.}$$

$$3n+3 = \Omega(2n) \quad \text{Correct, not used!}$$

# Linear time: $O(n)$

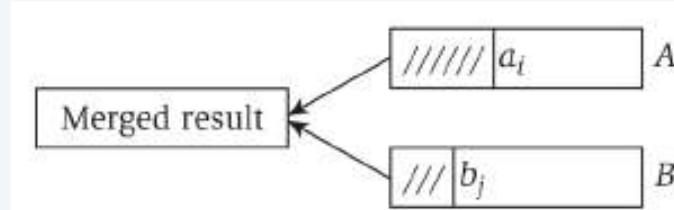
Linear time. Running time is proportional to input size.

Computing the maximum. Compute maximum of  $n$  numbers  $a_1, \dots, a_n$ .

```
max ← a1
for i = 2 to n {
    if (ai > max)
        max ← ai
}
```

# Linear time: $O(n)$

Merge. Combine two sorted lists  $A = a_1, a_2, \dots, a_n$  with  $B = b_1, b_2, \dots, b_n$  into sorted whole.



```
i = 1, j = 1
while (both lists are nonempty) {
    if ( $a_i \leq b_j$ ) append  $a_i$  to output list and increment i
    else            append  $b_j$  to output list and increment j
}
append remainder of nonempty list to output list
```

Claim. Merging two lists of size  $n$  takes  $O(n)$  time.

Pf. After each compare, the length of output list increases by 1.

# Linearithmic time: $O(n \log n)$

$O(n \log n)$  time. Arises in divide-and-conquer algorithms.

**Sorting.** Mergesort and heapsort are sorting algorithms that perform  $O(n \log n)$  compares.

**Largest empty interval.** Given  $n$  time-stamps  $x_1, \dots, x_n$  on which copies of a file arrive at a server, what is largest interval when no copies of file arrive?

**$O(n \log n)$  solution.** Sort the time-stamps. Scan the sorted list in order, identifying the maximum gap between successive time-stamps.

# Quadratic time: $O(n^2)$

|Ex. Enumerate all pairs of elements.

Closest pair of points. Given a list of  $n$  points in the plane  $(x_1, y_1), \dots, (x_n, y_n)$ , find the pair that is closest.

$O(n^2)$  solution. Try all pairs of points.

```
min ← (x1 - x2)2 + (y1 - y2)2
for i = 1 to n {
    for j = i+1 to n {
        d ← (xi - xj)2 + (yi - yj)2
        if (d < min)
            min ← d
    }
}
```

Remark.  $\Omega(n^2)$  seems inevitable, but this is just an illusion. [see Chapter 5]

# Cubic time: $O(n^3)$

Cubic time. Enumerate all triples of elements.

Set disjointness. Given  $n$  sets  $S_1, \dots, S_n$  each of which is a subset of  $1, 2, \dots, n$ , is there some pair of these which are disjoint?

$O(n^3)$  solution. For each pair of sets, determine if they are disjoint.

```
foreach set  $S_i$  {
    foreach other set  $S_j$  {
        foreach element  $p$  of  $S_i$  {
            determine whether  $p$  also belongs to  $S_j$ 
        }
        if (no element of  $S_i$  belongs to  $S_j$ )
            report that  $S_i$  and  $S_j$  are disjoint
    }
}
```

# Polynomial time: $O(n^k)$

Independent set of size  $k$ . Given a graph, are there  $k$  nodes such that no two are joined by an edge?

$k$  is a constant

$O(n^k)$  solution. Enumerate all subsets of  $k$  nodes.

```
foreach subset S of k nodes {  
    check whether S is an independent set  
    if (S is an independent set)  
        report S is an independent set  
    }  
}
```

- Check whether  $S$  is an independent set takes  $O(k^2)$  time.
- Number of  $k$  element subsets =  $\binom{n}{k} = \frac{n(n-1)(n-2) \times \dots \times (n-k+1)}{k(k-1)(k-2) \times \dots \times 1} \leq \frac{n^k}{k!}$
- $O(k^2 n^k / k!) = O(n^k)$ .

$\nwarrow$   
poly-time for  $k=17$ ,  
but not practical

# Exponential time

**Independent set.** Given a graph, what is maximum cardinality of an independent set?

**$O(n^2 2^n)$  solution.** Enumerate all subsets.

```
S* ← φ
foreach subset S of nodes {
    check whether S is an independent set
    if (S is largest independent set seen so far)
        update S* ← S
}
}
```

# Sublinear time

Search in a sorted array. Given a sorted array  $A$  of  $n$  numbers, is a given number  $x$  in the array?

$O(\log n)$  solution. Binary search.

```
lo ← 1, hi ← n
while (lo ≤ hi) {
    mid ← (lo + hi) / 2
    if      (x < A[mid]) hi ← mid - 1
    else if (x > A[mid]) lo ← mid + 1
    else return yes
}
return no
```

# References

- Kevin Wayne, “Analysis of Algorithms”
- Sartaj Sahni, “Analysis of Algorithms”
- BBM 201 Notes by Mustafa Ege
- Marty Stepp and Helene Martin, Recursion

# **BBM 201**

# **DATA STRUCTURES**

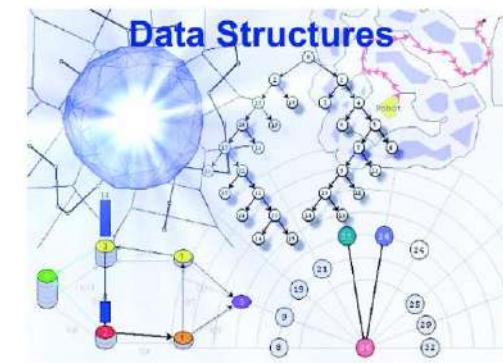
---

## **Lecture 3:**

## **Representation of Multidimensional Arrays**



**2018-2019 Fall**



# What is an Array?

- An array is a fixed size sequential collection of elements of identical types.
- A multidimensional array is treated as an array of arrays.
  - Let  $a$  be a  $k$ -dimensional array; the elements of  $a$  can be accessed using the following syntax:

$a[i_1][i_2] \dots [i_k]$

The following loop stores 0 into each location in two dimensional array  $A$ :

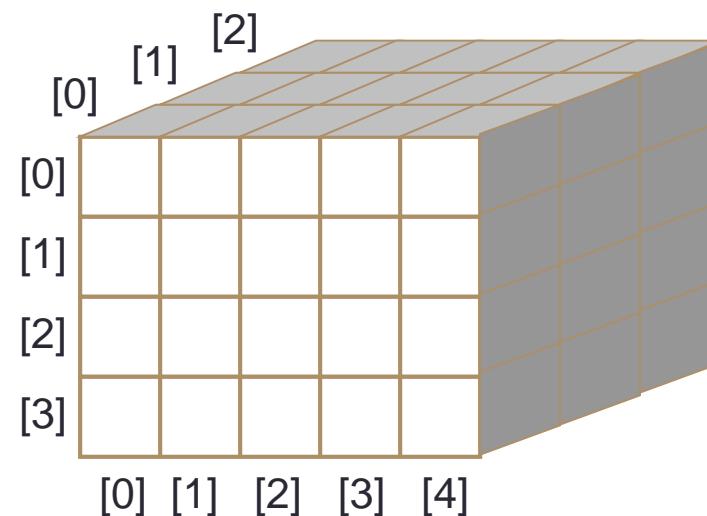
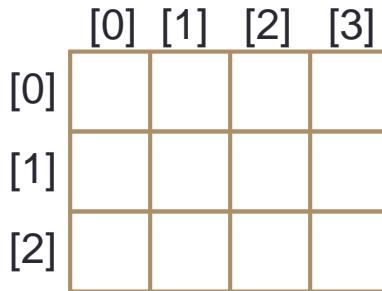
```
int row, column;
int A[3][4];
for (row = 0; row < 3; row++)
{
    for (column = 0; column < 4; column++)
    {
        A[row][column] = 0;
    }
}
```

# Definition of a Multidimensional Array

- One-dimensional arrays are linear containers.



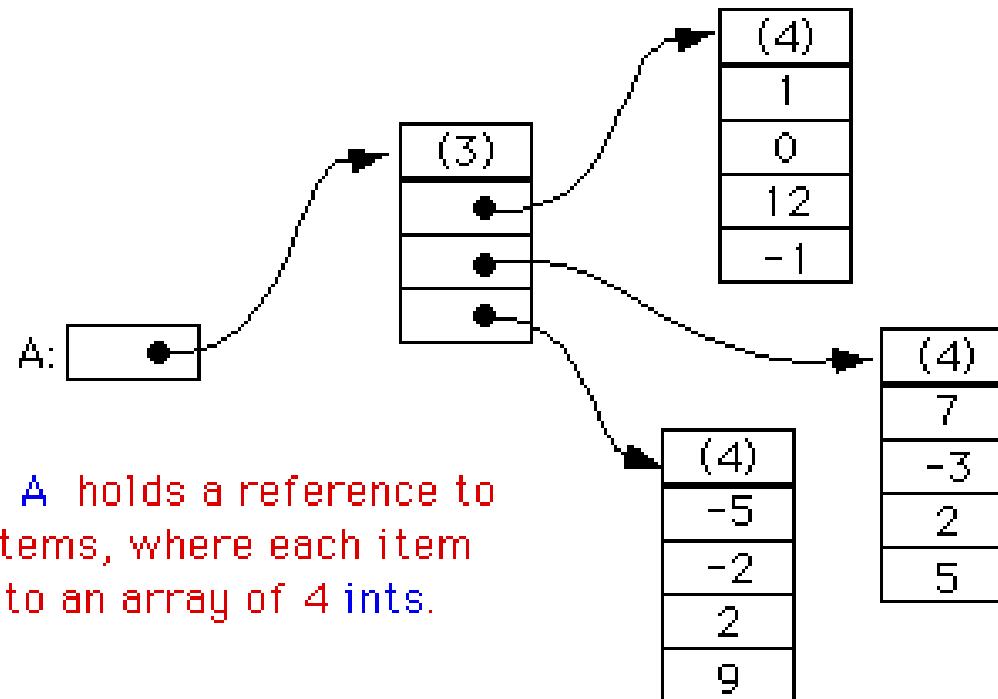
## Multi-dimensional Arrays



# Two-Dimensional Array

|    |    |    |    |    |
|----|----|----|----|----|
| A: | 1  | 0  | 12 | -1 |
|    | 7  | -3 | 2  | 5  |
|    | -5 | -2 | 2  | 9  |

If you create an array `A = new int[3][4]`, you should think of it as a "matrix" with 3 rows and 4 columns.



But in reality, `A` holds a reference to an array of 3 items, where each item is a reference to an array of 4 ints.

# Storage Allocation

The storage arrangement shown in this example uses the array subscript, also called the array indices.

Array declaration: int a[3][4];

Array elements:

|         |         |         |         |
|---------|---------|---------|---------|
| a[0][0] | a[0][1] | a[0][2] | a[0][3] |
| a[1][0] | a[1][1] | a[1][2] | a[1][3] |
| a[2][0] | a[2][1] | a[2][2] | a[2][3] |

# Array size

- In a matrix which is defined as

$a[upper_0] [upper_1] \dots [upper_{n-1}]$ ,

the number of items is:

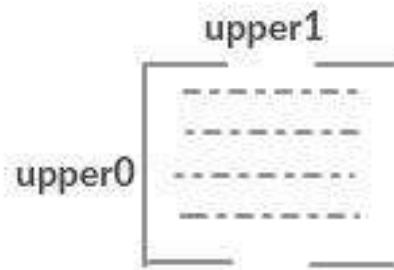
$$\sum_{i=0}^{n-1} upper^i$$

Example: What is the number of items in  $a[20][20][1]$ ?

# Memory Storage

- There are two types of placement for multidimensional arrays in memory:
  - Row major ordering
  - Column major ordering

Example: In an array which is defined as  $A[\text{upper}_0][\text{upper}_1]$ , if the memory address of  $A[0][0]$  is  $\alpha$ , then what is the memory address of  $A[i][0]$  (according to row major ordering)?

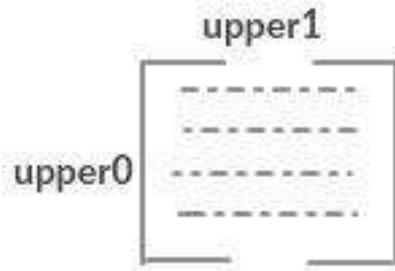


# Memory Storage

- There are two types of placement for multidimensional arrays in memory:
  - Row major ordering
  - Column major ordering

Example: In an array which is defined as  $A[\text{upper}_0][\text{upper}_1]$ , if the memory address of  $A[0][0]$  is  $\alpha$ , then what is the memory address of  $A[i][0]$  (according to row major ordering)?

$$\alpha + i * \text{upper}_1$$

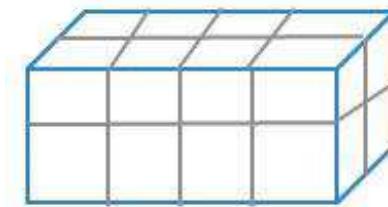


# Memory Storage

- For a three-dimensional array  $A[upper_0][upper_1][upper_2]$  what is the memory storage like?

- Example: char  $y[2][2][4]$

which slice?      which row?      which column?



- What is the memory address of  $y[1][1][3]$  if the memory address of  $y[0][0][0]$   $\alpha$ ?

# Memory Storage

The memory address of  $a[i][0][0]$  is:

$$\alpha + i * \text{upper}_1 * \text{upper}_2$$

if the memory address of  $a[0][0][0]$  is  $\alpha$ . Therefore, the memory address of  $a[i][j][k]$  becomes:

$$\alpha + i * \text{upper}_1 * \text{upper}_2 + j * \text{upper}_2 + k$$

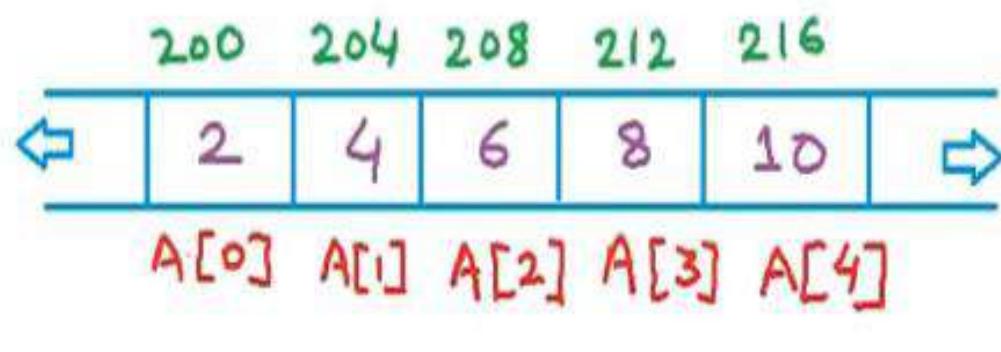
The memory address of  $a[i_0] [i_1] [i_2] \dots [i_{n-1}]$  is:

$$\alpha + \sum_{j=0}^{n-1} i_j a_j = \sum_{k=j+1}^{n-1} \text{upper}_k \quad 0 \leq j \leq n - 1$$

$a_{n-1} = 1$

# Pointers and Multi-dimensional Arrays

int A[5]



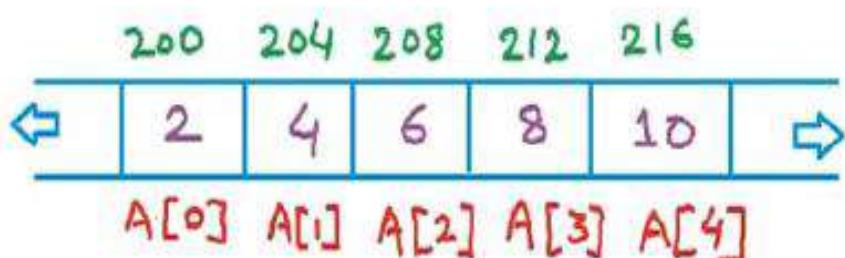
int \*P = A;

Print P // 200

Print \*P // 2

Print \*(P+2) // 6

`int A[5]`



`int *p = A;`

`Print A // 200`

`Print *A // 2`

`Print *(A+2) // 6`

$*(A+i)$  is same as  $A[i]$

$(A+i)$  is same as  $\&A[i]$

int A[5]



int \*p = A;

Print A // 200

Print \*A // 2

Print \*(A+2) // 6

\*(A+i) is same as A[i]

(A+i) is same as &A[i]

p = A ; ✓

A = p ; X

```
int A[5]
```

A[0] } → int  
A[1]  
;

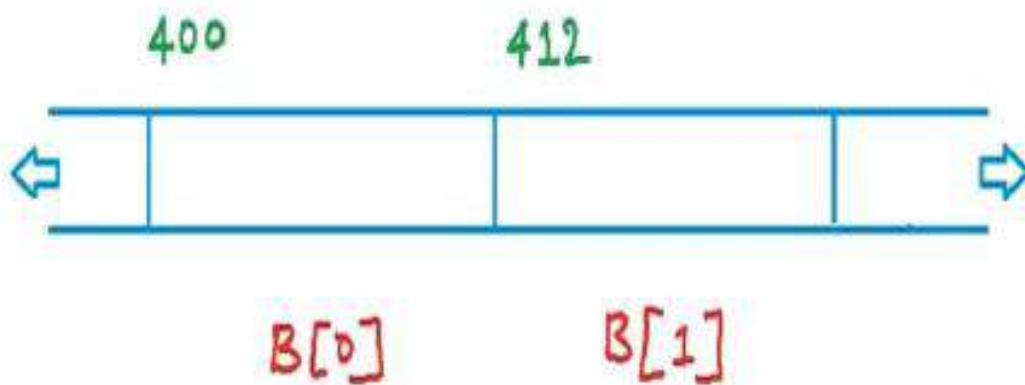
```
int B[2][3]
```

B[0] } → 1-D arrays  
B[1] } of 3 integers

|      |      |      |      |      |  |
|------|------|------|------|------|--|
| 200  | 204  | 208  | 212  | 216  |  |
| 2    | 4    | 6    | 8    | 10   |  |
| A[0] | A[1] | A[2] | A[3] | A[4] |  |

`int B[2][3]`

`B[0]` } → 1-D arrays  
`B[1]` } of 3 integers

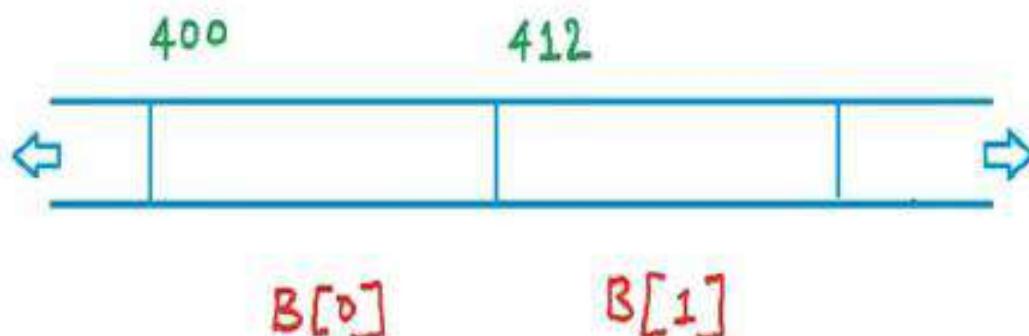


`int *p = B; X`

↓  
will return a pointer  
to 1-D array of 3 integers

`int B[2][3]`

`B[0]`      } → 1-D arrays  
`B[1]`      of 3 integers



`int *P = B; X`

↓  
will return a pointer  
to 1-D array of 3 integers

`int (*P)[3] = B; ✓`

```
int B[2][3]
```

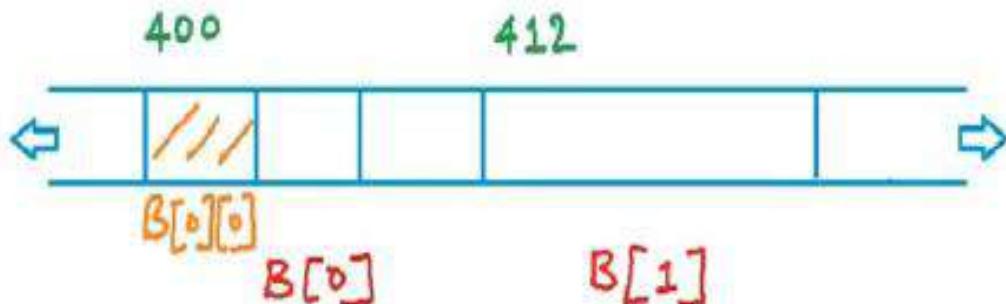
B[0] } → 1-D arrays  
B[1]    of 3 integers

```
int (*P)[3] = B;
```

↓  
will return a pointer  
to 1-D array of 3 integers

```
Print B or &B[0] // 400
```

```
Print *B or B[0] or &B[0][0] // 400
```



```
int B[2][3]
```

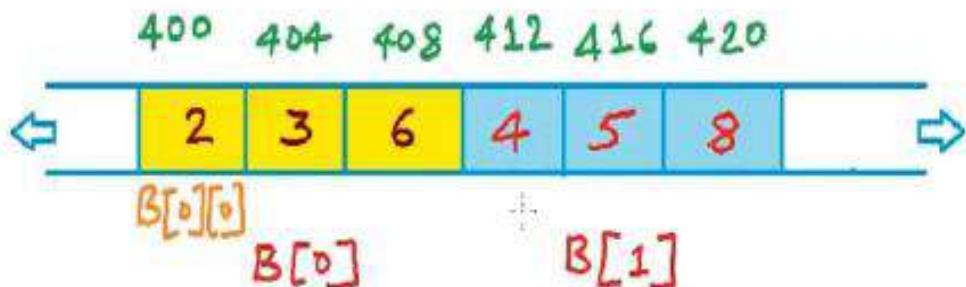
B[0] } → 1-D arrays  
B[1]    of 3 integers

```
int (*P)[3] = B;
```

```
Print B or &B[0] // 400
```

```
Print *B or B[0] or &B[0][0] // 400
```

Print B+1 // 400 + 12 = 412  
or  
&B[1]



```
int B[2][3]
```

B[0] } → 1-D arrays  
B[1]      of 3 integers

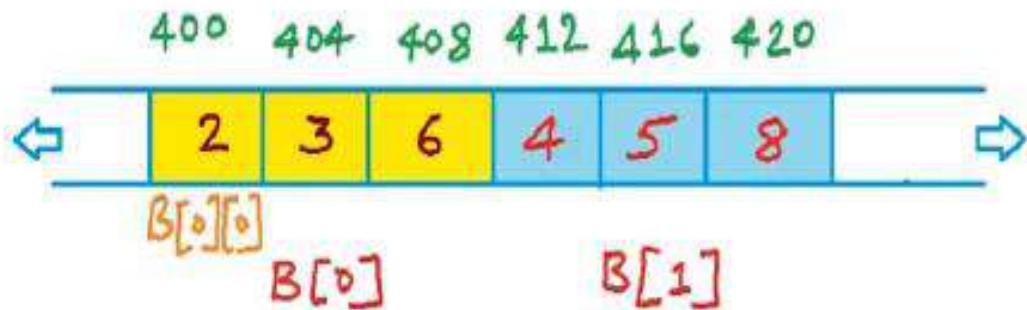
```
int (*P)[3] = B;
```

Print B or &B[0] // 400

Print \*B or B[0] or &B[0][0] // 400

Print B+1 or &B[1] // 412

Print \*(B+1) or B[1] or &B[1][0] // 412



```
int B[2][3]
```

B[0] } → 1-D arrays  
B[1]    of 3 integers



```
int (*P)[3] = B;
```

Print B or &B[0] // 400

Print \*B or B[0] or &B[0][0] // 400

Print B+1 or &B[1] // 412

Print \*(B+1) or B[1] or &B[1][0] // 412

Print \*(B+1)+2 or B[1]+2 or &B[1][2] // 420  
                    → returning int \*

```
int B[2][3]
```

B[0] } → 1-D arrays  
B[1]      of 3 integers

```
int (*P)[3] = B;
```

Print B or &B[0] // 400

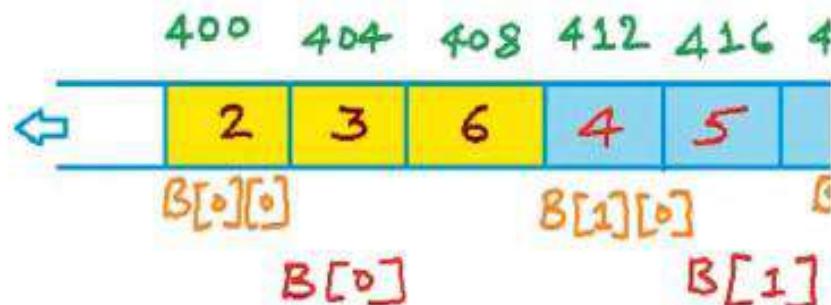
Print \*B or B[0] or &B[0][0] // 400

Print B+1 or &B[1] // 412

Print \*(B+1) or B[1] or &B[1][0] // 412

Print \*(B+1)+2 or B[1]+2 or &B[1][2] // 420

Print \*(\*B+1)  
          B → int (\*)[3]  
          B[0] → int \*



```
int B[2][3]
```

B[0] } → 1-D arrays  
B[1] of 3 integers

```
int (*P)[3] = B;
```

Print B or &B[0] // 400

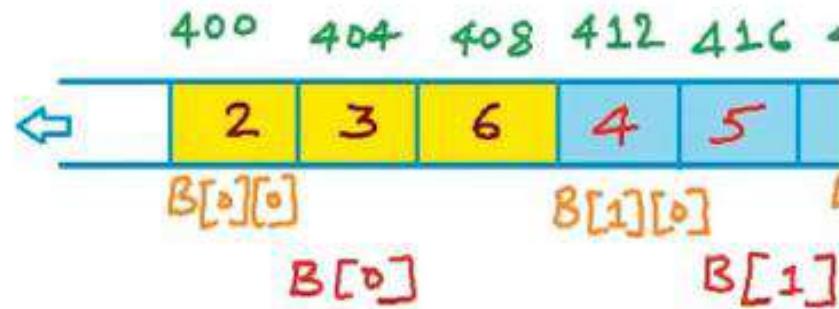
Print \*B or B[0] or &B[0][0] // 400

Print B+1 or &B[1] // 412

Print \*(B+1) or B[1] or &B[1][0] // 412

Print \*(B+1)+2 or B[1]+2 or &B[1][2] // 420

Print \*(\*B+1)  
↓  
&B[0][1]



```
int B[2][3]
```

B[0] } → 1-D arrays  
B[1]      of 3 integers

```
int (*P)[3] = B;
```

Print B or &B[0] // 400

Print \*B or B[0] or &B[0][0] // 400

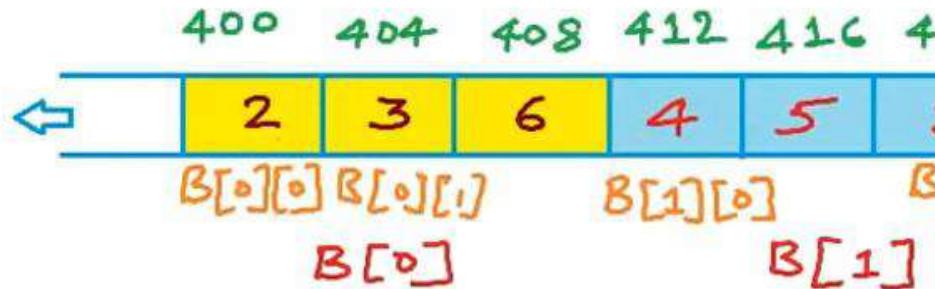
Print B+1 or &B[1] // 412

Print \*(B+1) or B[1] or &B[1][0] // 412

Print \*(B+1)+2 or B[1]+2 or &B[1][2] // 420

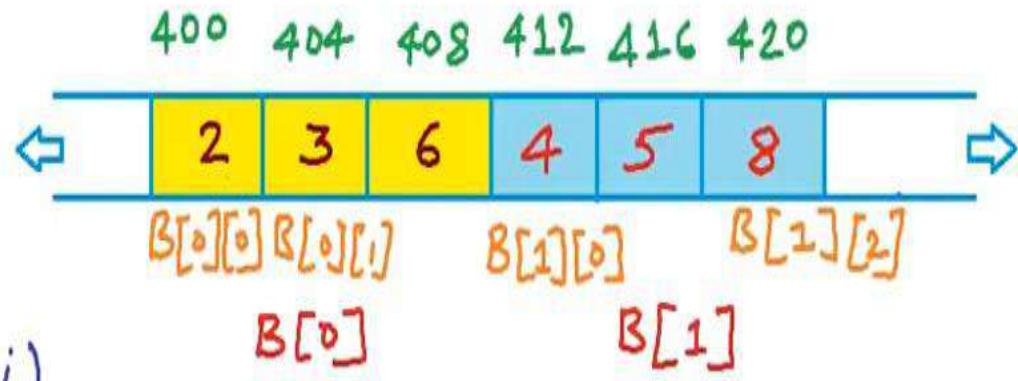
Print \*(\*B+1) // 3

↓  
B[0][1]



int B[2][3]

For 2-D array



$$\begin{aligned}B[i][j] &= *(B[i]+j) \\&= *(*(B+i)+j)\end{aligned}$$

## Pointers and multi-dimensional arrays

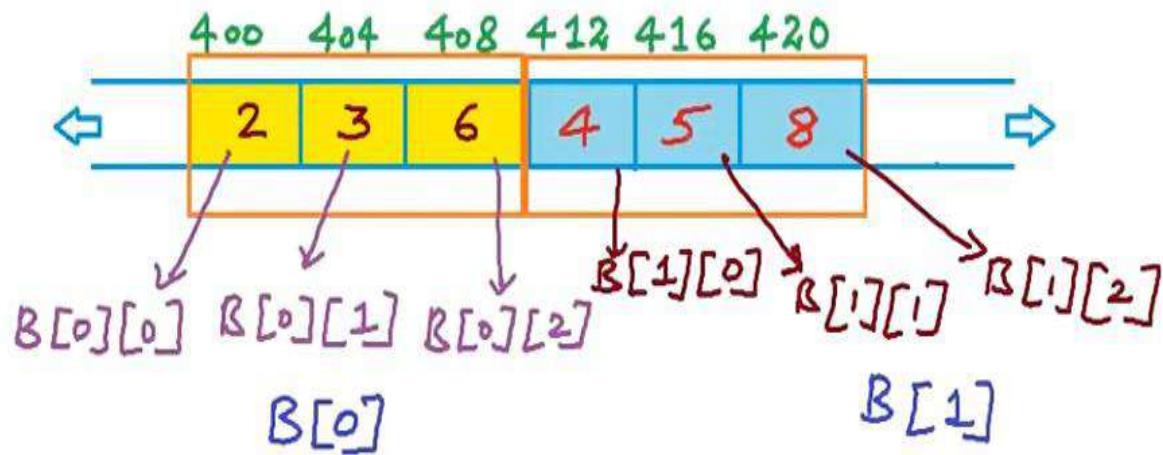
int B[2][3]

int (\*P)[3] = B; ✓

↓  
declaring

pointer to 1-D  
array of 3 integers

int \*P = B; X



## Pointers and multi-dimensional arrays

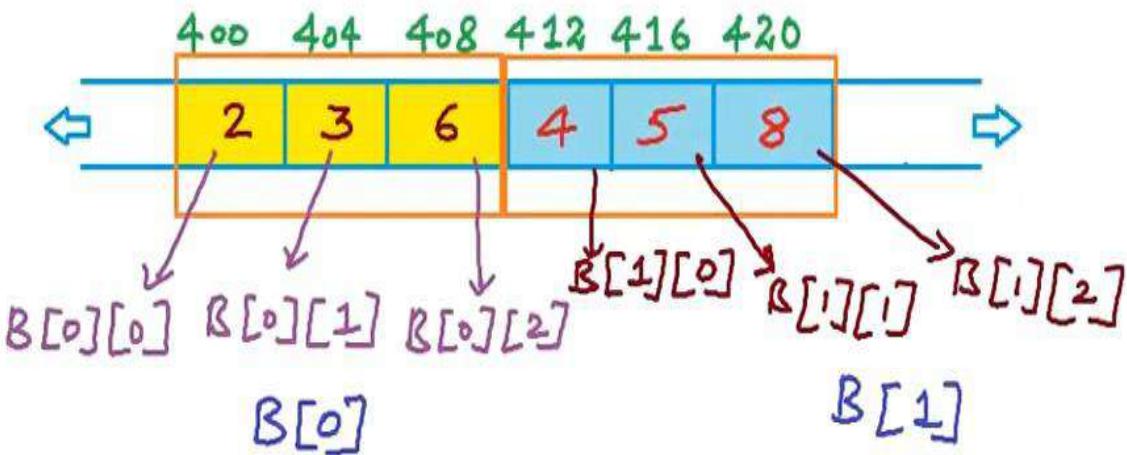
```
int B[2][3]
```

```
int (*P)[3] = B; ✓
```

```
Print B //400
```

```
Print *B //400
```

```
Print B[0] //400
```



## Pointers and multi-dimensional arrays

```
int B[2][3]
```

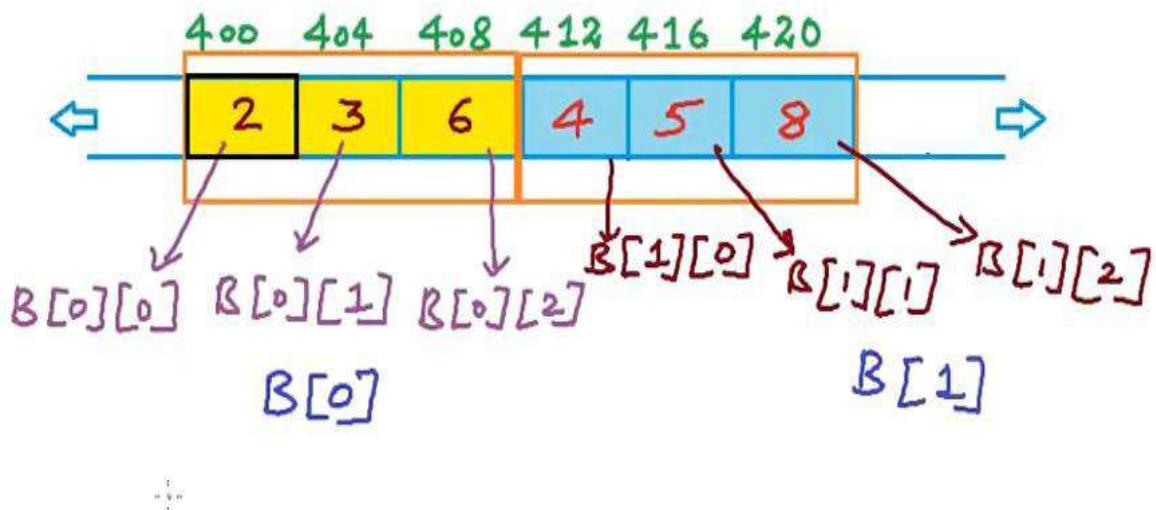
```
int (*P)[3] = B; ✓
```

```
Print B //400
```

```
Print *B //400 }
```

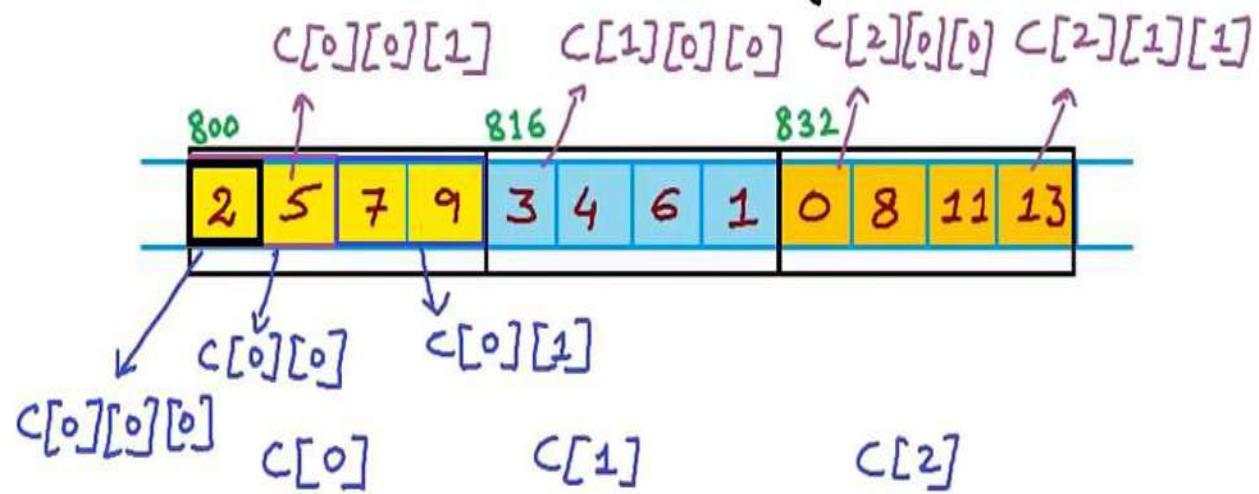
```
Print B[0] //400 }
```

```
Print &B[0][0] //400
```



## Pointers and multi-dimensional arrays

int c[3][2][2]



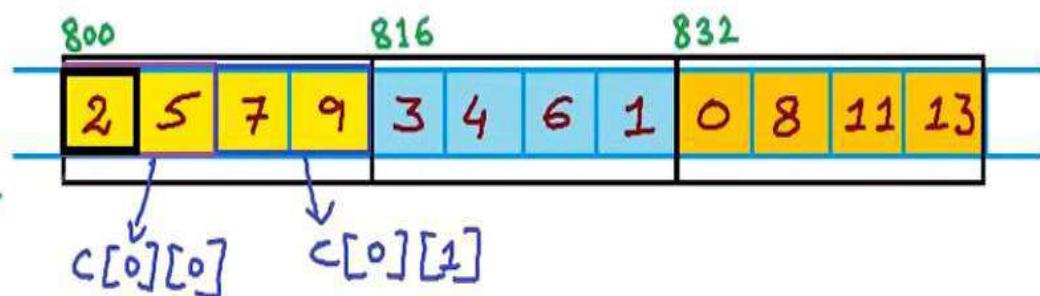
## Pointers and multi-dimensional arrays

```
int C[3][2][2]
```

```
int (*P)[2][2] = C; ✓
```

Print  $\hookrightarrow$   $\text{int}(*[2][2])$      $C[0]$      $C[1]$      $C[2]$   
 $// 800$

Print  $\underbrace{*C \text{ or } C[0] \text{ or } \&C[0][0]}_{\downarrow}$      $// 800$   
 $\text{int}(*[2])$

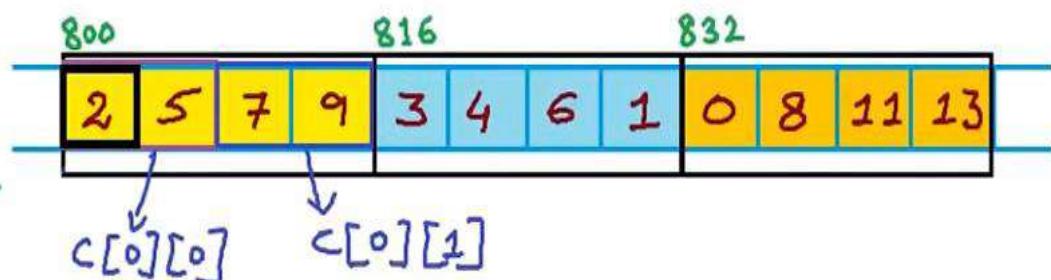


## Pointers and multi-dimensional arrays

int C[3][2][2]

int (\*P)[2][2] = C; ✓

Print C // 800  
    ↳ int (\*)[2][2]      C[0]      C[1]      C[2]  
Print \*C or C[0] or &C[0][0]



$$\begin{aligned} C[i][j][k] &= * (C[i][j] + k) = *(*(c[i] + j) + k) \\ &= *(*(*(c + i) + j) + k) \end{aligned}$$

## Pointers and multi-dimensional arrays

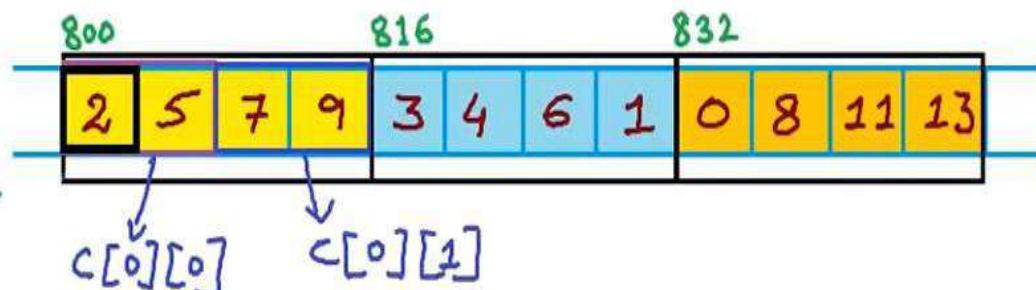
int C[3][2][2]

int (\*P)[2][2] = C; ✓

Print C // 800  
    ↳ int (\*P)[2][2]      C[0]      C[1]      C[2]

Print \*C or C[0] or &C[0][0]

Print \*(C[0][1] + 1) or C[0][1][1] // 9



## Pointers and multi-dimensional arrays

int C[3][2][2]

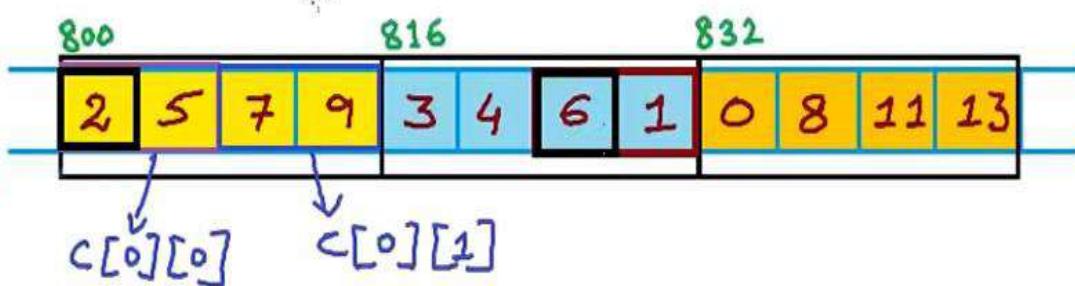
int (\*P)[2][2] = C; ✓

Print C // 800  
int (\*P)[2][2] C[0] C[1] C[2]

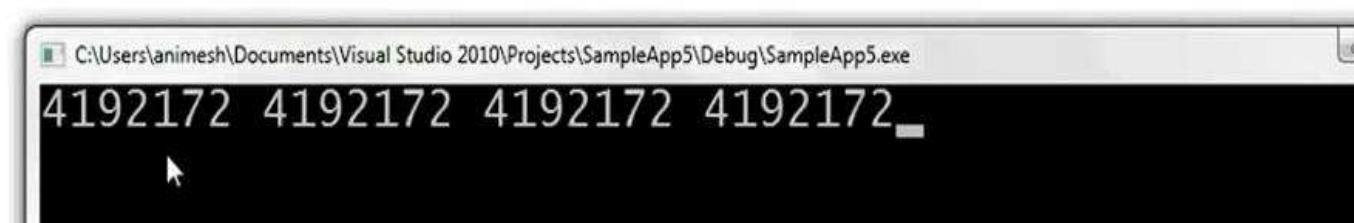
Print \*C or C[0] or &C[0][0] // 800

Print \*(C[0][1] + 1) or C[0][1][1] // 9

Print \*(C[1] + 1) or C[1][1] or &C[1][0] // 824



```
// Pointers and multi-dimensional arrays
#include<stdio.h>
int main()
{
    int C[3][2][2]={{ {2,5},{7,9} },
                    {{3,4},{6,1} },
                    {{0,8},{11,13}}};
    printf("%d %d %d %d", C, *C, C[0], &C[0][0]);
}
```



# References

- BBM 201 Notes by Mustafa Ege
- Lecture Videos: [www.mycodeschool.com/videos/pointers-and-arrays](http://www.mycodeschool.com/videos/pointers-and-arrays)

# **BBM 201**

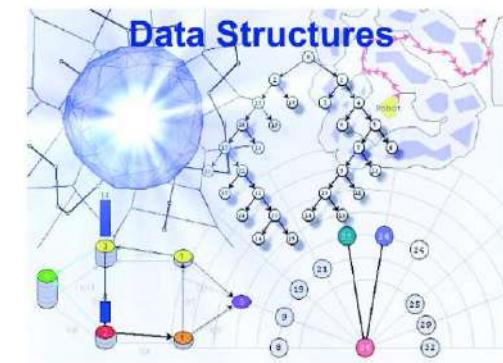
# **DATA STRUCTURES**

---

**Lecture 4:**  
**Lower/Upper Triangular Matrix**  
**Band Matrix**  
**Sparse Matrix**



**2018-2019 Fall**




$$\begin{bmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{bmatrix}$$

WELCOME .... TO  
THE MATRIX!!!!

# Lower Triangular Matrix

Triangular matrix

Upper triangular matrix

$$U = \begin{bmatrix} u_{11} & u_{12} & u_{13} & \dots & u_{1n} \\ 0 & u_{22} & u_{23} & & \cdot \\ 0 & 0 & u_{33} & & \cdot \\ \cdot & & \cdot & & \cdot \\ \cdot & & \cdot & & \cdot \\ \cdot & & \cdot & & \cdot \\ 0 & \cdot & \cdot & \cdot & 0 & u_{nn} \end{bmatrix}$$

Lower triangular matrix

$$L = \begin{bmatrix} l_{11} & 0 & 0 & \dots & 0 \\ l_{21} & l_{22} & 0 & & \cdot \\ l_{31} & l_{32} & l_{33} & & \cdot \\ \cdot & & \cdot & & \cdot \\ \cdot & & \cdot & & \cdot \\ \cdot & & \cdot & & 0 \\ l_{n1} & \cdot & \cdot & \cdot & \cdot & l_{nn} \end{bmatrix}$$

# Lower Triangular Matrix

- Does the definition of a special data structure for triangular matrix provide any benefits over a typical matrix in terms of **memory** and **processing time?**
- We can insert the items in a single dimensional array:

• **ALT**

|          |          |          |          |          |          |          |          |          |          |
|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| $a_{00}$ | $a_{10}$ | $a_{11}$ | $a_{20}$ | $a_{21}$ | $a_{22}$ | $a_{30}$ | $a_{31}$ | $a_{32}$ | $a_{33}$ |
| [0]      | [1]      | [2]      | [3]      | [4]      | [5]      | [6]      | [7]      | [8]      | [9]      |

- Number of items in the array becomes:

$$1 + 2 + \dots + (n - 1) + n = \frac{n(n+1)}{2}$$

$$a \begin{bmatrix} a_{00} & 0 & 0 & 0 \\ a_{10} & a_{11} & 0 & 0 \\ a_{20} & a_{21} & a_{22} & 0 \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix}$$

# Lower Triangular Matrix

- How can we find the position of  $u[i][j]$  in the array?
  - Answer:  $i=1$ , there is one item in the 0<sup>th</sup> row, 2 items in the 1<sup>st</sup> row.
  - $i=2$ , there is one item in the 0<sup>th</sup> row, 2 items in the 1<sup>st</sup> row, 3 items in the 2<sup>nd</sup> row.
  - Therefore the address of  $u[i][j]$  in the array is calculated as below:

$$\begin{aligned} k &= \sum_{t=0}^i a(t) + (j) = (0 + 1 + 2 + \dots + i) + (j) \\ &= \frac{i(i+1)}{2} + (j) \end{aligned}$$

# Lower Triangular Matrix

```
void main(void) {
    int alt[MAX_SIZE];
    int i, n;
    scanf("%d", &n); //matrix size
    readtriangularmatrix(alt,n);
    for(i=0; i<=n*(n+1)/2-1; i++)
        printf(" %d", alt[i]);

    i=gettriangularmatrix(3,0,n);
    if(i==-2)
        printf("\n invalid index\n");
    else if(i==-1)
        printf("\n access to the upper triangular\n");
    else
        printf("\n the position in 'alt' matrix: %d value: %d \n", i, alt[i]);
```

# Lower Triangular Matrix

```
void readtriangularmatrix(int alt[], int n)
{
    int i, j, k;
    if(n*(n+1)/2 > MAX_SIZE){
        printf("\n invalid array size \n");
        exit(-1);
    }
    else
        for(i=0; i<=n-1; i++){
            k=(i+1)*i/2;
            for(j=0; j<=i; j++)
                scanf("%d", &alt[k+j]);
        }
}
```

```
int gettriangularmatrix(int i, int j, int n){

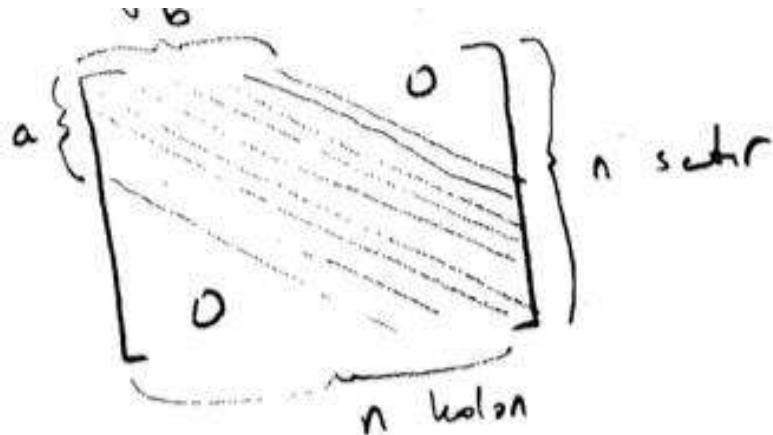
    if(i<0 || i>=n || j<0 || j>=n){
        printf("\n invalid index\n");
        exit(-2);
    }
    else if(i>=j) //valid index
        return (i+1)*i/2+j;
    else return -1; //outside of the triangular
                    //value is zero
```

# Band Matrix

$$\begin{bmatrix} B_{11} & B_{12} & 0 & \cdots & \cdots & 0 \\ B_{21} & B_{22} & B_{23} & \ddots & \ddots & \vdots \\ 0 & B_{32} & B_{33} & B_{34} & \ddots & \vdots \\ \vdots & \ddots & B_{43} & B_{44} & B_{45} & 0 \\ \vdots & \ddots & \ddots & B_{54} & B_{55} & B_{56} \\ 0 & \cdots & \cdots & 0 & B_{65} & B_{66} \end{bmatrix}$$

Matrix (n, a) : n by n matrix, non-zero entries are confined to a diagonal band, comprising the main diagonal and zero or more diagonals (a-1) on either side.

# Band Matrix

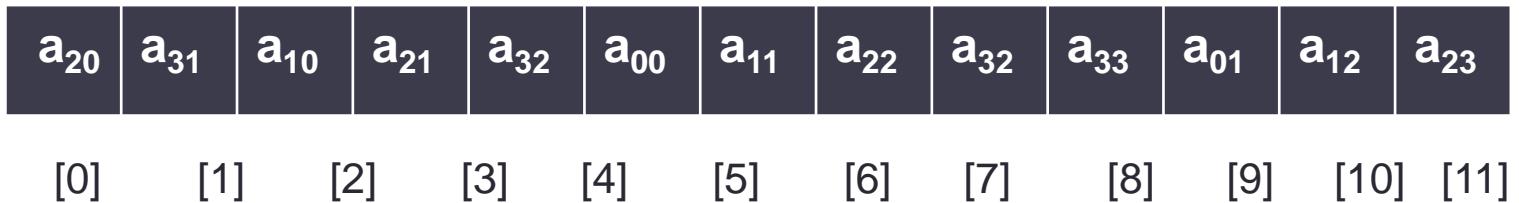


$$\begin{bmatrix} d_{0,0} & d_{0,1} & 0 & 0 \\ d_{1,0} & d_{1,1} & d_{1,2} & 0 \\ d_{2,0} & d_{2,1} & d_{2,2} & d_{2,3} \\ 0 & d_{3,1} & d_{3,2} & d_{3,3} \end{bmatrix}$$

$$b = 2$$
$$a = 3$$
$$n = 4$$

# Band Matrix

- What kind of a data structure can we use?
- We can insert the items in a single dimensional array:

- *ALT* 

|          |          |          |          |          |          |          |          |          |          |          |          |          |
|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| $a_{20}$ | $a_{31}$ | $a_{10}$ | $a_{21}$ | $a_{32}$ | $a_{00}$ | $a_{11}$ | $a_{22}$ | $a_{32}$ | $a_{33}$ | $a_{01}$ | $a_{12}$ | $a_{23}$ |
| [0]      | [1]      | [2]      | [3]      | [4]      | [5]      | [6]      | [7]      | [8]      | [9]      | [10]     | [11]     |          |

- What is the number of items in the array?

# Band Matrix

- What is the number of items in the array?

- Number of items **on** and **below** the diagonal:

$$n + (n - 1) + (n - 2) + \dots + n - (a - 1)$$

- Number of items **above** the diagonal:

$$(n - 1) + (n - 2) + \dots + n - (b - 1)$$

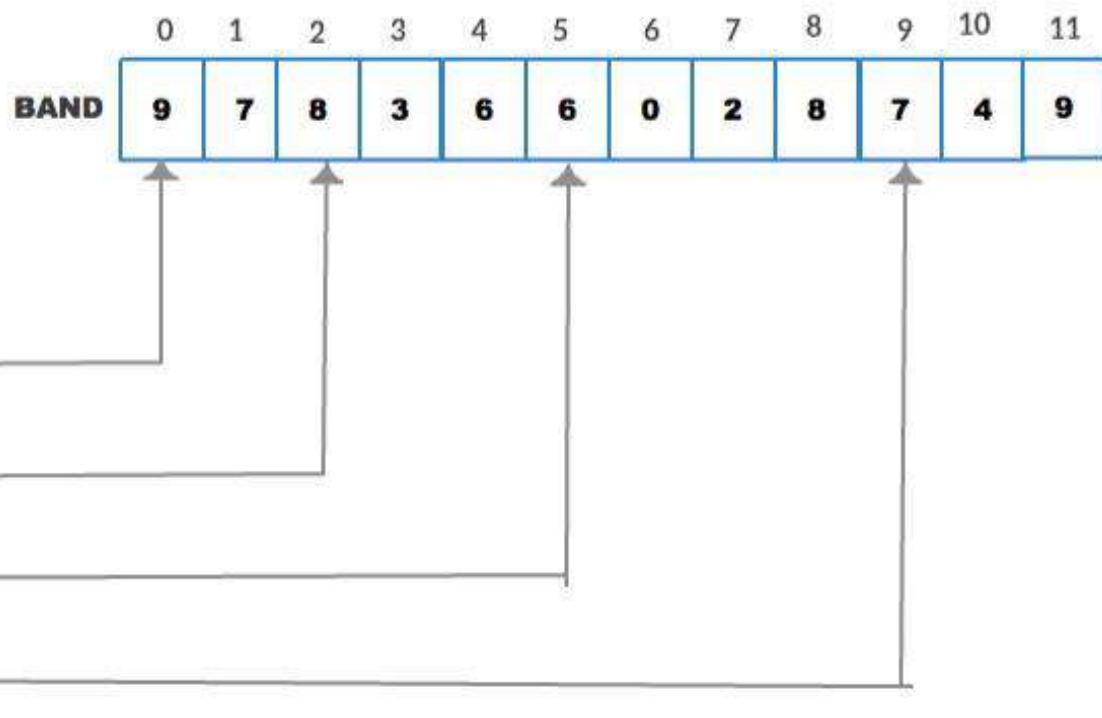
- Sum of these becomes:

$$\text{Sum} = n + (n - 1) + (n - 2) + \dots + n - (a - 1) + (n - 1) + (n - 2) + \dots + n - (b - 1)$$

$$= n(a + b - 1) - \frac{(a - 1)a}{2} - \frac{(b - 1)b}{2}$$

# Band Matrix

$$\begin{bmatrix} 6 & 7 & 0 & 0 \\ 8 & 0 & 4 & 0 \\ 9 & 3 & 2 & 9 \\ 0 & 7 & 6 & 8 \end{bmatrix}$$



# Band Matrix

```
void main(void) {
    int band[MAX_SIZE];
    int search[MAX_SIZE];

    int i, n, a, b;
    printf(" n:", &n); scanf("%d", &n);
    printf(" a:", &a); scanf("%d", &a);
    printf(" b:", &b); scanf("%d", &b);

    buildbandmatrix(band, search, a, b);

    for(i=0; i<=n*(a+b-1)-a*(a-1)/2-b*(b-1)/2-1; i++)
        printf(" %d ",band[i]);

    printf("\n");
    for(i=0; i<=a+b-2; i++)
        printf(" %d ", search[i]);

    i=getbandmatrix(3,3,n,a,b,search);

    if(i==2)
        printf("\n invalid index");
    else if(i==1)
        printf("\n item to be searched: 0");
    else
        printf("\n item to be searched: %d->%d", i, band[i]);
```

# Band Matrix

```
void buildbandmatrix(int band[], int search[], int n, int a, int b){  
    int i, k, itemnum;  
  
    if(n*(a+b-1)-a*(a-1)/2-b*(b-1)/2 > MAX_SIZE) {  
  
        printf("\n not enough memory");  
        exit(-1);  
    }  
    else{  
        itemnum=0;  
        for(i=-a+1; i<=b-1; i++){ //for each diagonal  
            search[i+a-1]=itemnum;  
  
            for(k=0; k<= n-abs(i)-1; k++) //for the current diagonal  
                scanf("%d", &band[search[i+a-1]+k]);  
            itemnum = itemnum+(n-abs(i));  
        }  
    }  
}
```

# Band Matrix

```
void getbandmatrix(int i, int j, int n, int a, int b, int search[]){  
  
    if(i>=n || i<0 || j>=n || j<0){ //index overflow  
        printf("\n invalid index\n");  
        return -2;  
    }  
    else{  
        if(j>i) //above the diagonal  
            if(j-i<b) //above the upper band  
                return(search[a-1+j-i]+i); //yes  
            else //no  
                return -1;  
        else if(i-j<a) //below or on the diagonal  
            return(search[j-i+a-1]+j);  
        else //not on the band  
            return -1;  
    }  
}
```

# Sparse Matrix

- Most of the elements are zero.
- It wastes space.

**Sparsity:** the fraction of zero elements.

Basic matrix operations:

1. Creation
2. Addition
3. Multiplication
4. Transpose

| A | 0  | 1  | 2  | 3  | 4 | 5   |
|---|----|----|----|----|---|-----|
| 0 | 15 | 0  | 0  | 22 | 0 | -15 |
| 1 | 0  | 11 | 3  | 0  | 0 | 0   |
| 2 | 0  | 0  | 0  | -6 | 0 | 0   |
| 3 | 0  | 0  | 0  | 0  | 0 | 0   |
| 4 | 91 | 0  | 0  | 0  | 0 | 0   |
| 5 | 0  | 0  | 28 | 0  | 0 | 0   |

# Sparse Matrix

## Data Structure

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

- a[0].row: row index
- a[0].col: column index
- a[0].value: number of items in the sparse matrix



Rows and columns are in ascending order!

# Sparse Matrix

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 0 & 15 & 0 & 0 & 22 & 0 & -15 \\ 1 & 0 & 11 & 3 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & -6 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & 91 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 28 & 0 & 0 & 0 \end{bmatrix}$$

|      | Row | Column | 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[8] | 5   | 2      | 28    |

# Matrix Transpose

- Replacement of rows and columns in a matrix is called the transpose of the matrix:

$$A = \begin{pmatrix} 1 & 3 \\ 0 & 4 \end{pmatrix} \quad A' = \begin{pmatrix} 1 & 0 \\ 3 & 4 \end{pmatrix}$$

- The item  $a[i][j]$  becomes  $a[j][i]$ .

# Matrix Transpose

```
void transpose(term a[],term b[])
{
    int n,i,j,currentb;
    n=a[0].value; //number of items
    b[0].row=a[0].col; //number of rows
    b[0].col=a[0].row; //number of columns
    b[0].value=n;

    if(n>0){
        currentb=1;
        for(i=0; i<a[0].col; i++)
            for(j=1; j<=n; j++) //find the ones with col i in a
                if(a[j].col==i){
                    b[currentb].row=a[j].col;
                    b[currentb].col=a[j].row;
                    b[currentb].value=a[j].value;
                    currentb++;
                }
    }
}
```

**Question:** What is the complexity of this method?

# Fast Transpose

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

void fastTranspose(term a[], term b[])
{
    int ItemNum[MAX_COL], StartPos[MAX_COL];
    int i,j,ColNum=a[0].col,TermNum=a[0].value;
    b[0].value=TermNum;
    if(TermNum>0){ //does the item exist?
        for(i=0;i<ColNum;i++)
            ItemNum[i]=0;
        for(i=1;i<=TermNum;i++)
            ItemNum[a[i].col]++;
        StartPos[0]=1;
        for(i=1;i<ColNum;i++)
            StartPos[i]=StartPos[i-1]+ItemNum[i-1];
        for(i=1;i<=TermNum;i++){
            j=StartPos[a[i].col]++;
            b[j].row=a[i].col; b[j].col=a[i].row;
            b[j].value=a[i].value;
        }
    }
}
```

# Fast Transpose

- Execute the fastTranspose method.
- **Question:** What is the complexity of the method?
- Compare its complexity with the previous transpose method.

# **BBM 201**

# **DATA STRUCTURES**

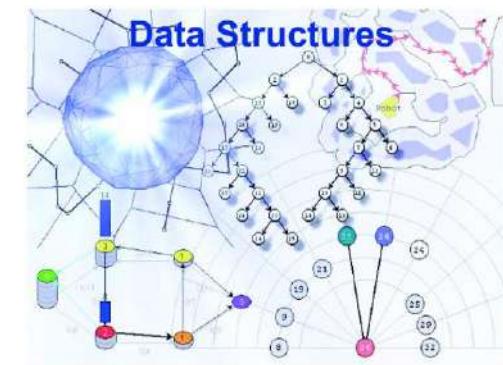
---

## Lecture 5:

## Stacks and Queues



2018-2019 Fall

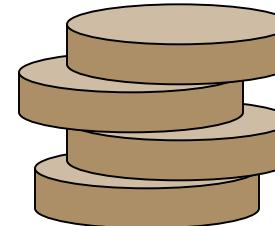
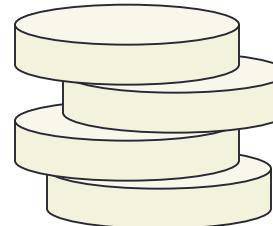
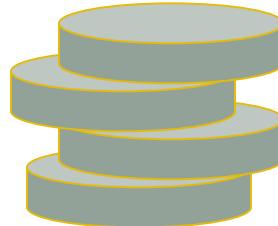




# Stacks

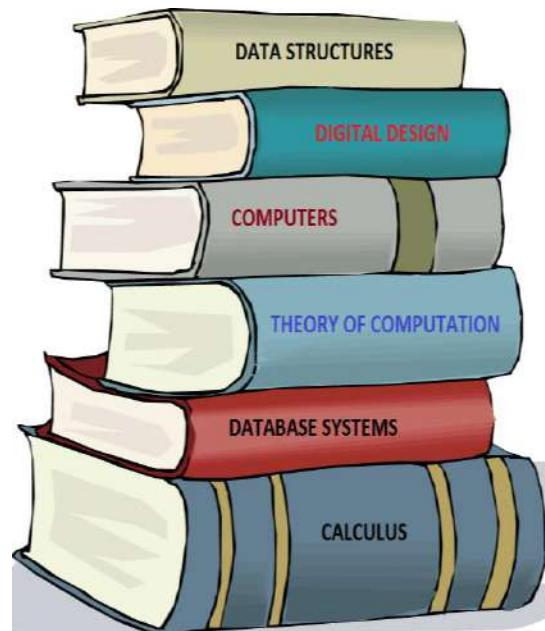
- A list on which insertion and deletion can be performed.
  - **Based on Last-in-First-out (LIFO)**
- Stacks are used for a number of applications:
  - Converting a decimal number into binary
  - Program execution
  - Parsing
  - Evaluating postfix expressions
  - Towers of Hanoi

...

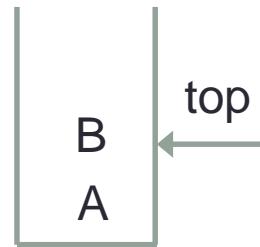
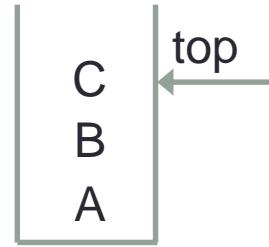
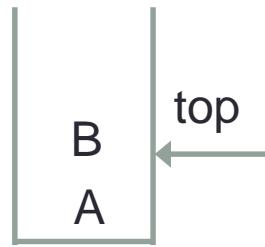
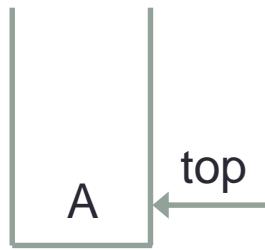


# Stacks

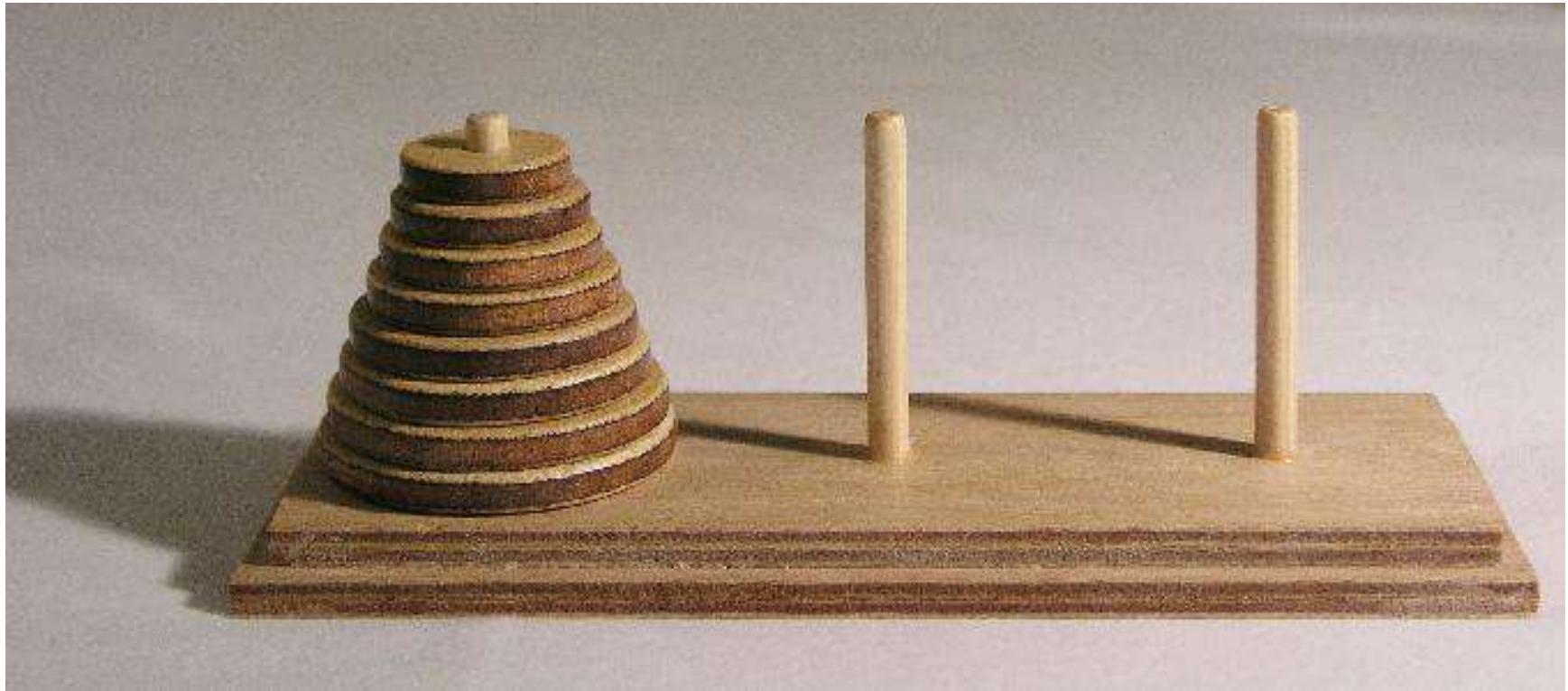
A stack is an ordered lists in which insertions and deletions are made at one end called the **top**.



# Stacks

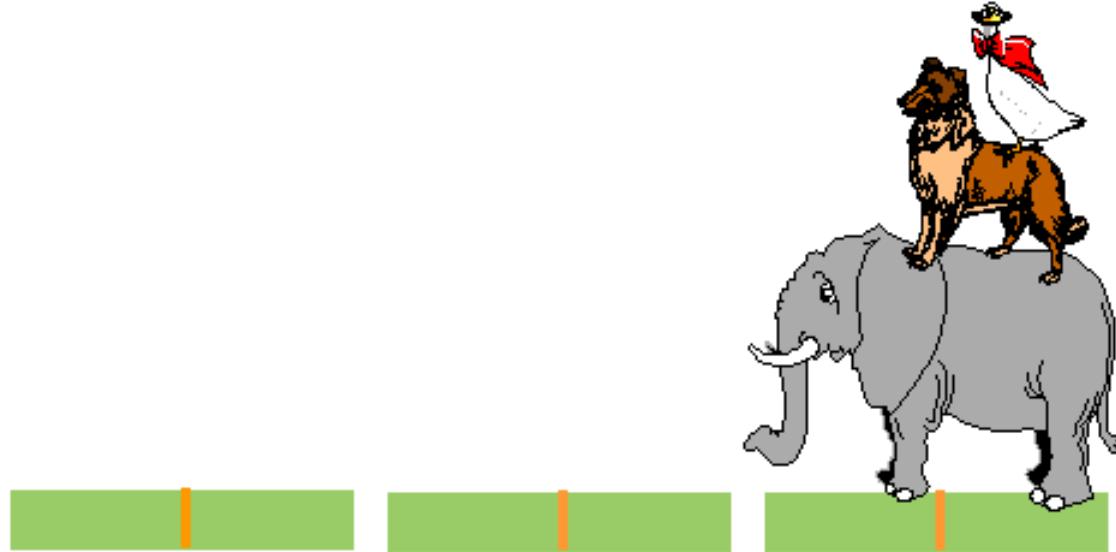


# Towers of Hanoi

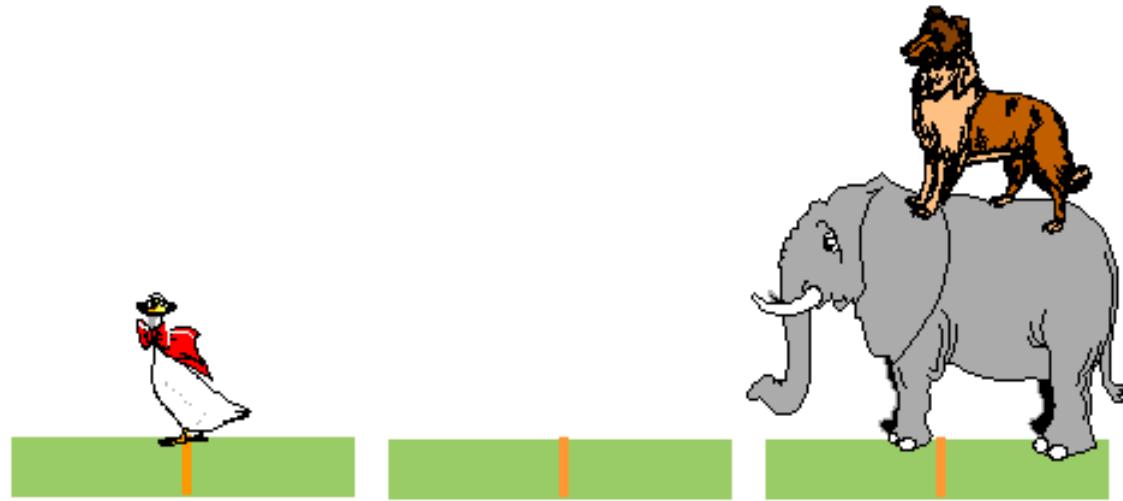


Object of the game is to move all the disks (animals) over to Tower 3.  
But you cannot place a larger disk onto a smaller disk.

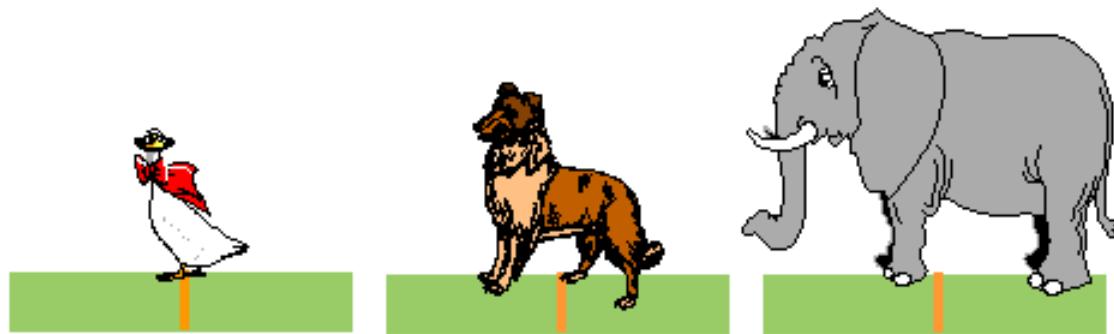
# Towers of Hanoi



# Towers of Hanoi



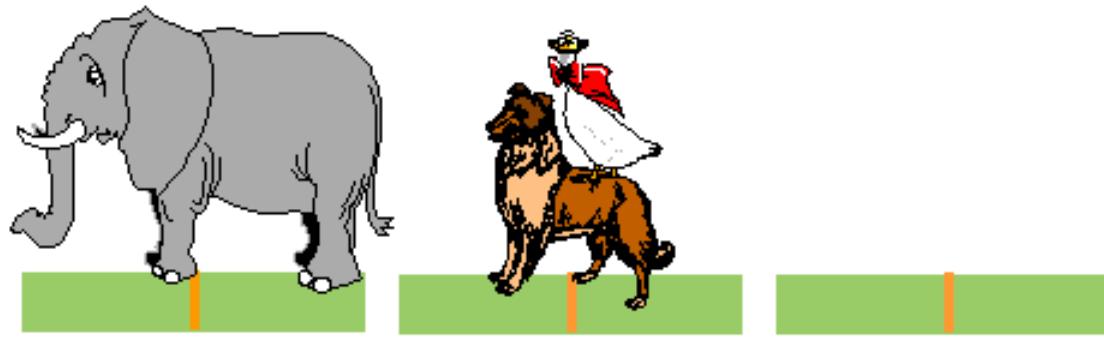
# Towers of Hanoi



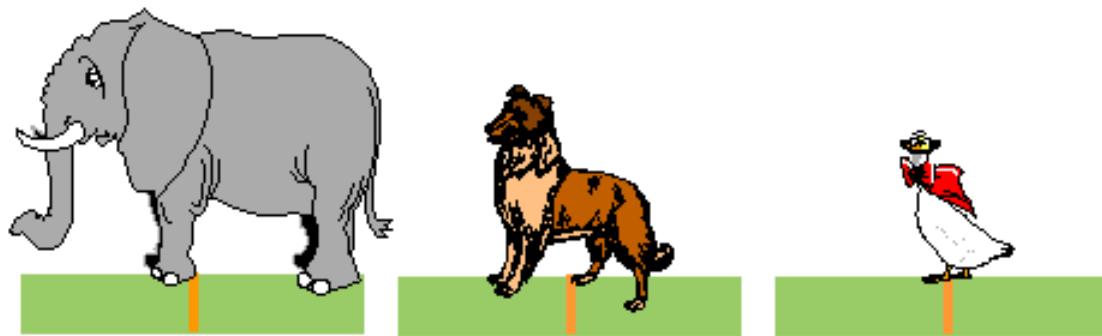
# Towers of Hanoi



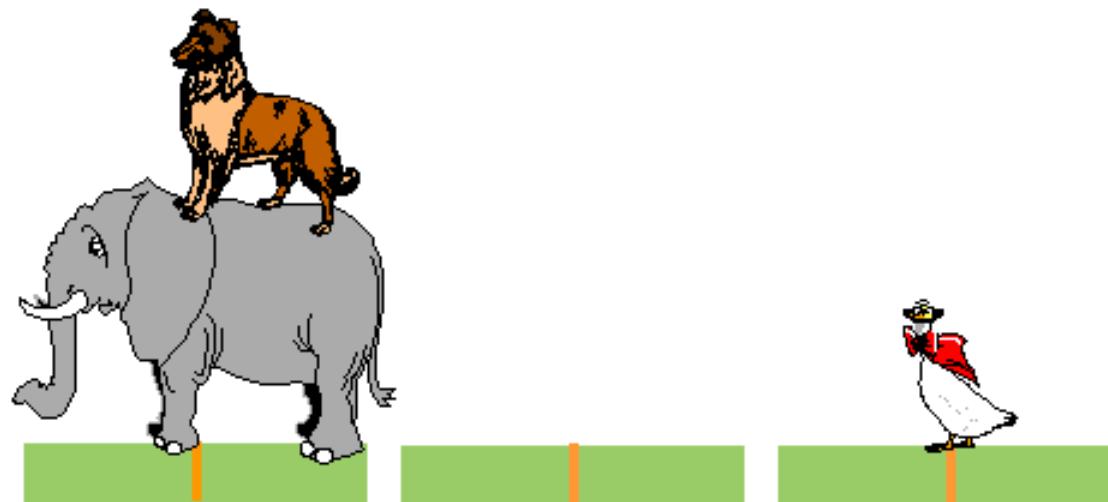
# Towers of Hanoi



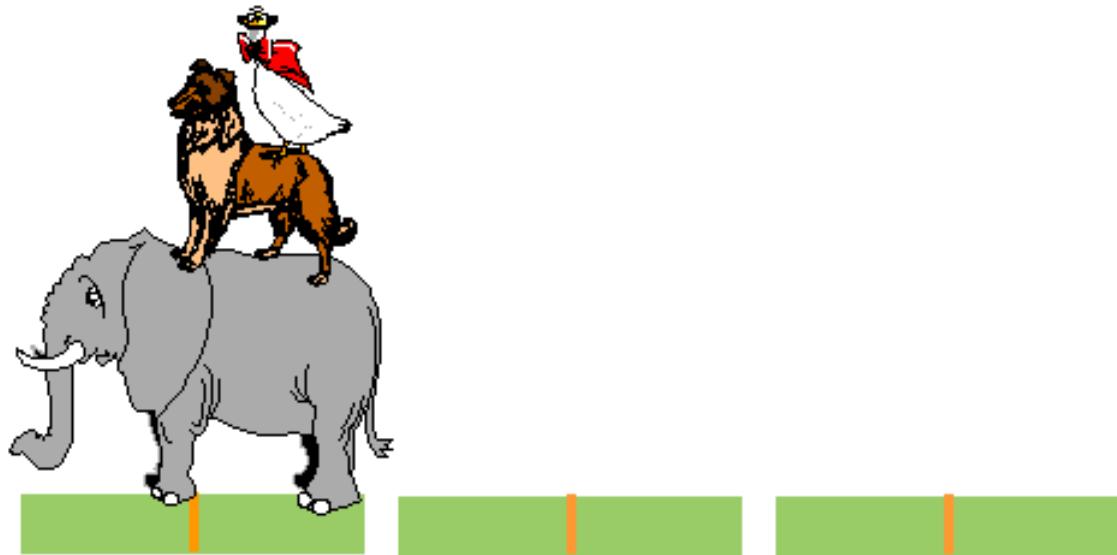
# Towers of Hanoi



# Towers of Hanoi



# Towers of Hanoi



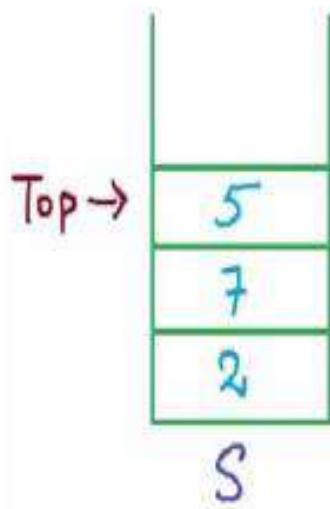
# Stack Operations

1. Pop()
2. Push(x)
3. Top()
4. IsEmpty()

- An insertion (of, say x) is called **push** operation and removing the most recent element from stack is called **pop** operation.
- **Top** returns the element at the top of the stack.
- **IsEmpty** returns true if the stack is empty, otherwise returns false.

*All of these take constant time - O(1)*

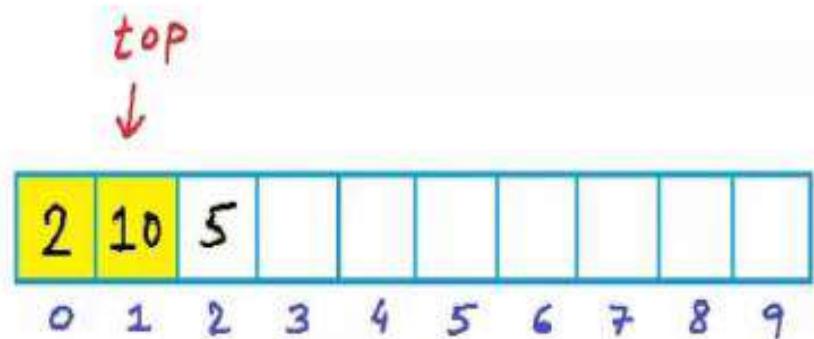
# Example



- Push(2)
- Push(10)
- Pop()
- Push(7)
- Push(5)
- Top(): 5
- IsEmpty(): False

# Array implementation of stack (pseudocode)

```
int A[10]
top <- -1 //empty stack
Push(x)
{
    top <- top + 1
    A[top] <- x
}
Pop()
{
    top <- top - 1
}
```

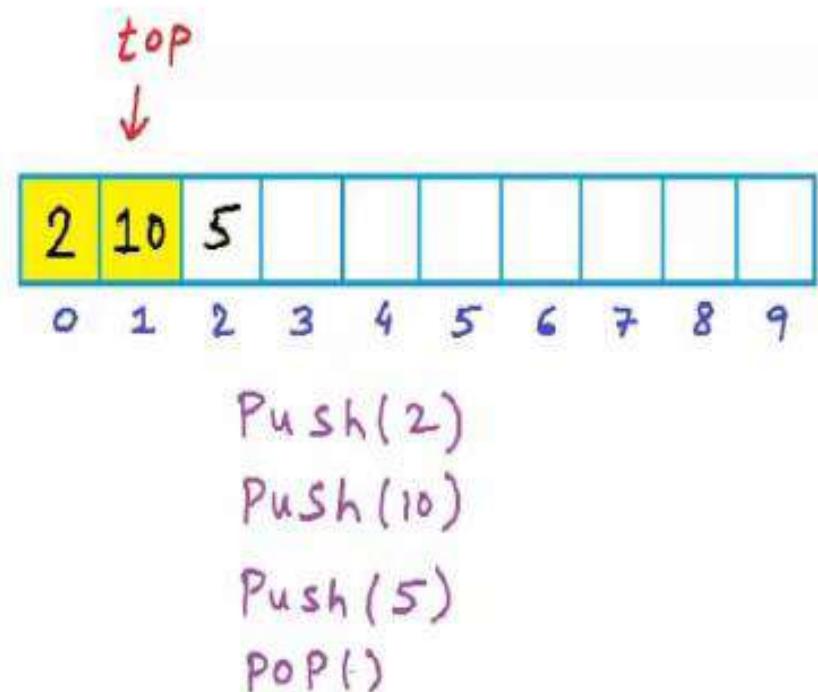


Push(2)  
Push(10)  
Push(5)  
POP()

For an empty stack, top is set to -1.  
In push function, we increment top.  
In pop, we decrement top by 1.

# Array implementation of stack (pseudocode)

```
Top ()  
{  
    return A[top]  
}  
IsEmpty ()  
{  
    if (top == -1)  
        return true  
    else  
        return false  
}
```



# Stack

## Data Structure

```
#define MAX_STACK_SIZE 100

typedef struct{
    int VALUE;
}element;

element stack[MAX_STACK_SIZE];
int top=-1;
```

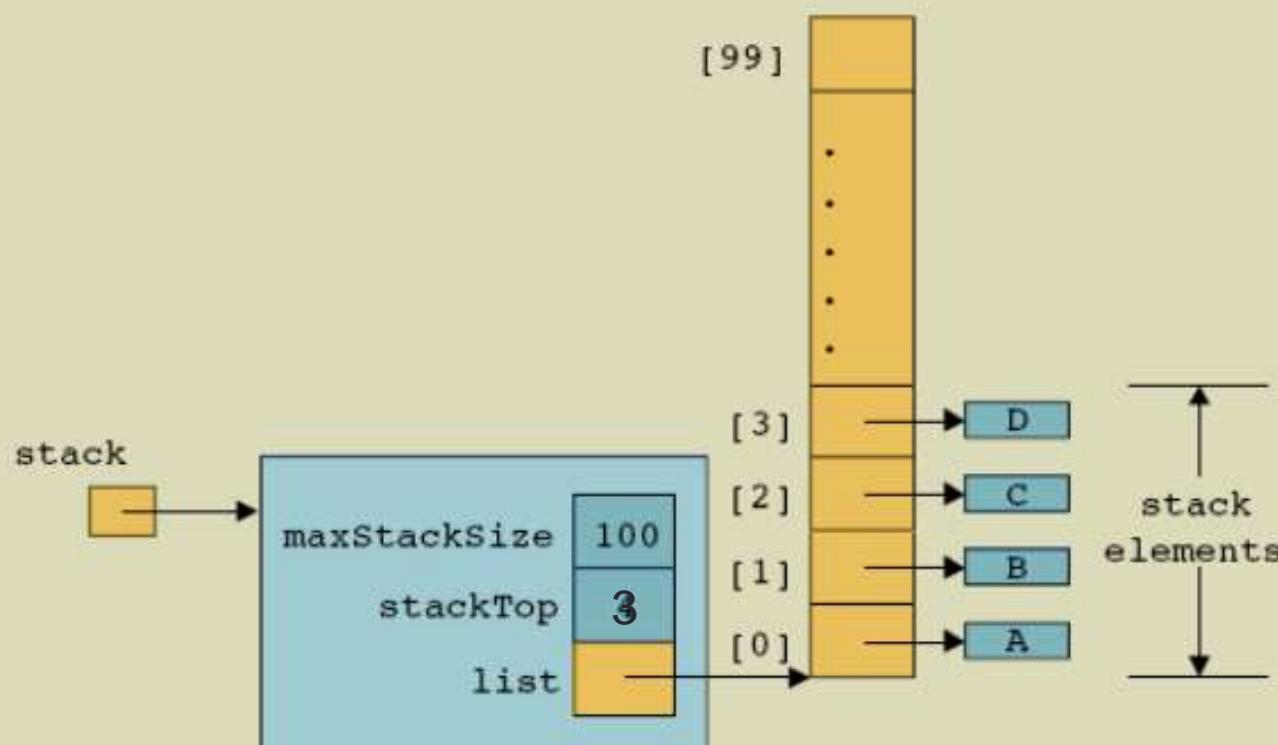
# Push Stack

```
void push (element item)
{
    if (top >= MAX_STACK_SIZE-1) {
        isFull();
        return;
    }
    stack[++top]=item;
}
```

# Pop Stack

```
element pop()
{
    if(top== -1)
        return empty_stack();
    return stack[top--];
}
```

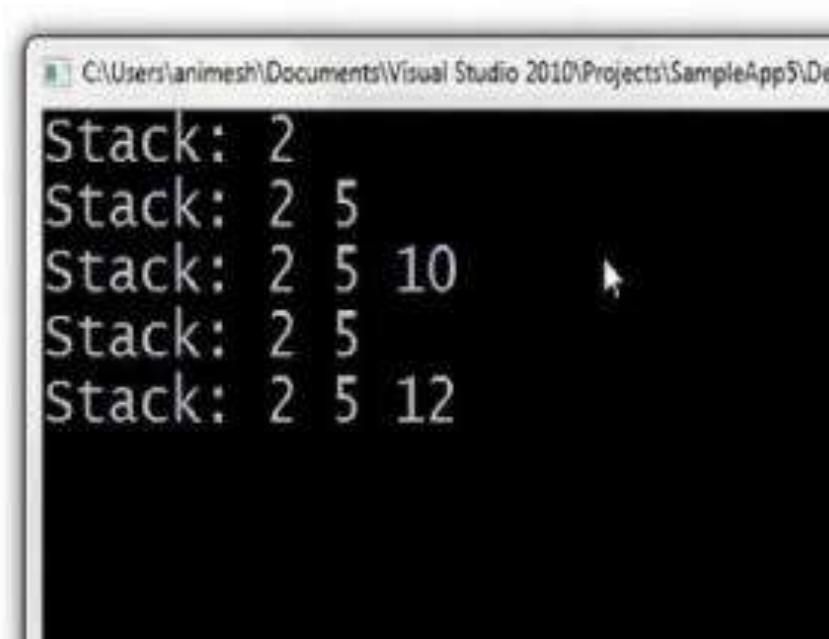
# Implementation of Stacks Using Arrays



# More array implementation

```
// Stack - Array based implementation.  
#include<stdio.h>  
#define MAX_SIZE 101  
int A[MAX_SIZE];  
int top = -1;  
void Push(int x) {  
    if(top == MAX_SIZE -1) {  
        printf("Error: stack overflow\n");  
        return;  
    }  
    A[++top] = x;  
}  
void Pop() {  
    if(top == -1) {  
        printf("Error: No element to pop\n");  
        return;  
    }  
    top--;  
}  
int Top() {  
    return A[top];  
}  
int main() {}
```

```
void Print() {
    int i;
    printf("Stack: ");
    for(i = 0;i<=top;i++)
        printf("%d ",A[i]);
    printf("\n");
}
int main() {
    Push(2);Print();
    Push(5);Print();
    Push(10);Print();
    Pop();Print();
    Push(12);Print();
}
```



# Check For Balanced Parentheses using Stack

| Expression    | Balanced? |
|---------------|-----------|
| (A+B)         |           |
| {(A+B)+(C+D)} |           |
| {(x+y)*(z)}   |           |
| [2*3]+(A)]    |           |
| {a+z)         |           |

# Check For Balanced Parentheses using Stack

| Expression | Balanced? |
|------------|-----------|
| ( )        | Yes       |
| {(())()}   | Yes       |
| {(())( )}  | No        |
| [ ]( )]    | No        |
| { )        | No        |

The count of opening should be equal to the count of closings.  
AND  
Any parenthesis opened last should be closed first.

# Idea: Create an empty list

- Scan from left to right
  - If opening symbol, add it to the list
    - Push it into the stack
  - If closing symbol, remove last opening symbol of the same type
    - using Pop from the stack
- Should end with an empty list

# Check For Balanced Parentheses: Pseudocode

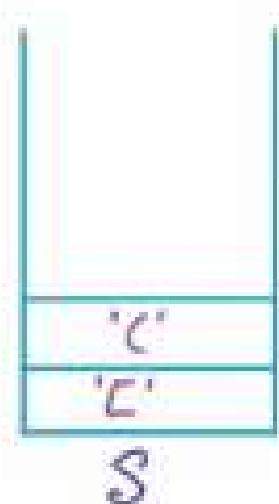
```
CheckBalancedParanthesis(exp) {
    n ← length(exp)
    Create a stack: S
    for i ← 0 to n-1{
        if exp[i] is '(' or '{' or '['
            Push(exp[i])
        else if exp[i] is ')' or '}' or ']'{
            if (S is empty or
                top does not pair with exp[i])
                return false
            else
                pop()
        }
    }
    return S is empty?
}
```

Create a stack of characters and scan this string by using push if the character is an opening parenthesis and by using pop if the character is a closing parenthesis. (See next slide)

# Examples

$\text{exp} = [()$

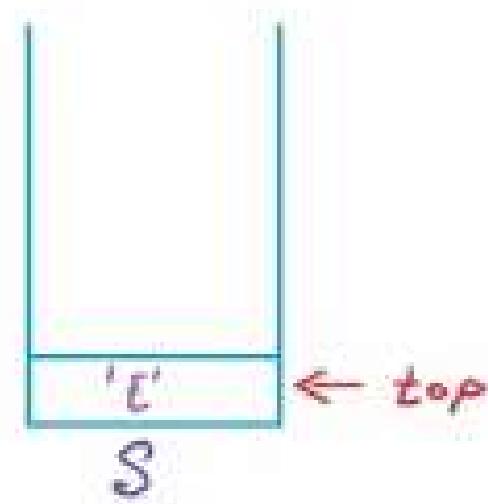
$i = 2$



The pseudo code will return false.

$\text{exp} = \{()\}$

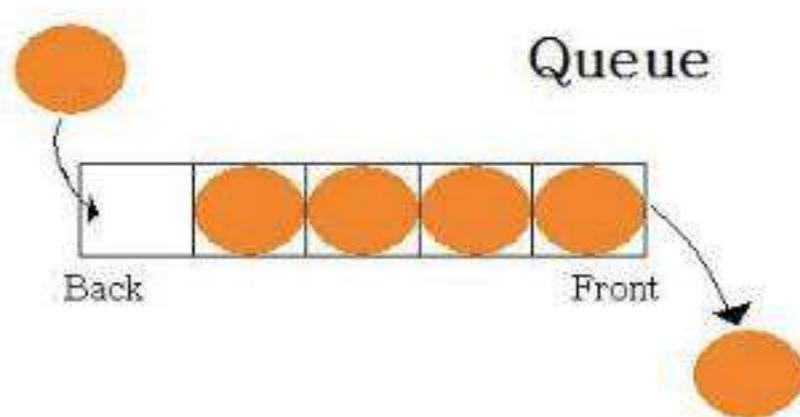
$i = 5$



The pseudo code will return true.

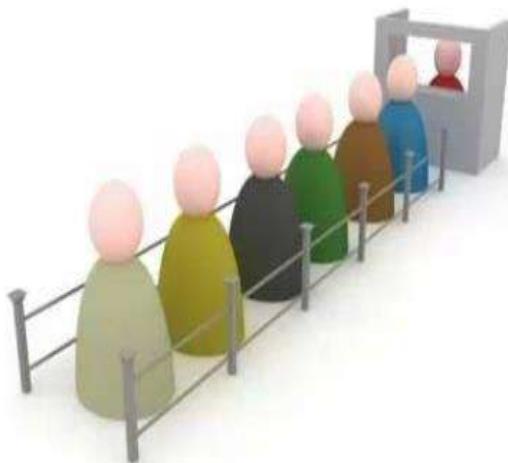
# Queues

- A queue is an ordered list on which
  - all insertions take place at one end called the **rear/back** and
  - all deletions take place at the opposite end called the **front**.
  - Based on **First-in-First-out (FIFO)**



# Comparison of Queue and Stack

Queue ADT



Queue - First-In-First-Out  
(FIFO)

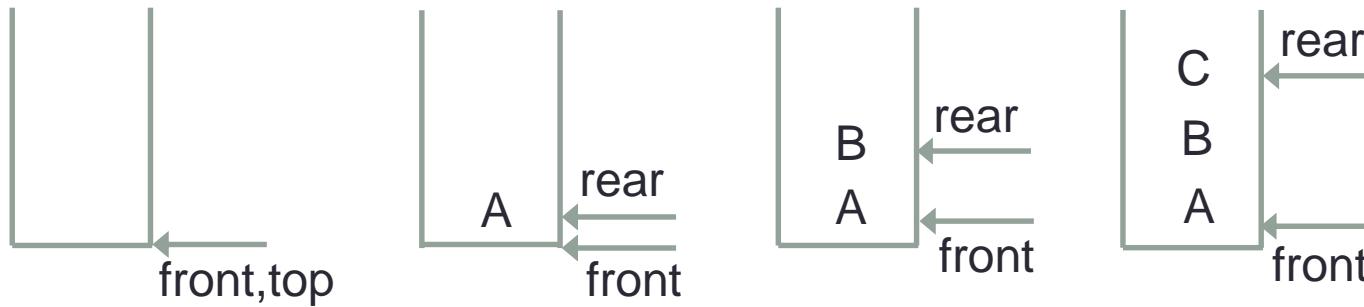


Stack - Last-In-First-Out  
(LIFO)



© Alamy

# Queues



Queue is a list with the restriction that insertion can be made at one end (**rear**)  
And deletion can be made at other end (**front**).

# Built-in Operations for Queue

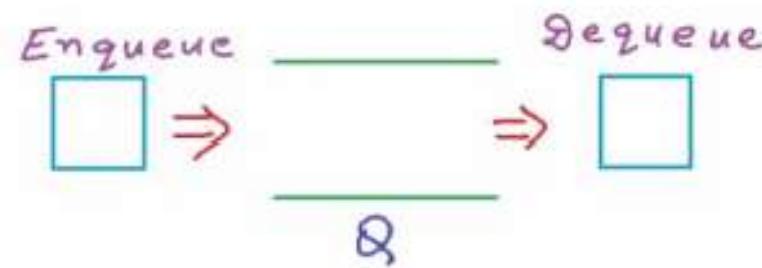
Enqueue(x) or Push(x)

Dequeue() or Pop()

Front(): Returns the element in the front without removing it.

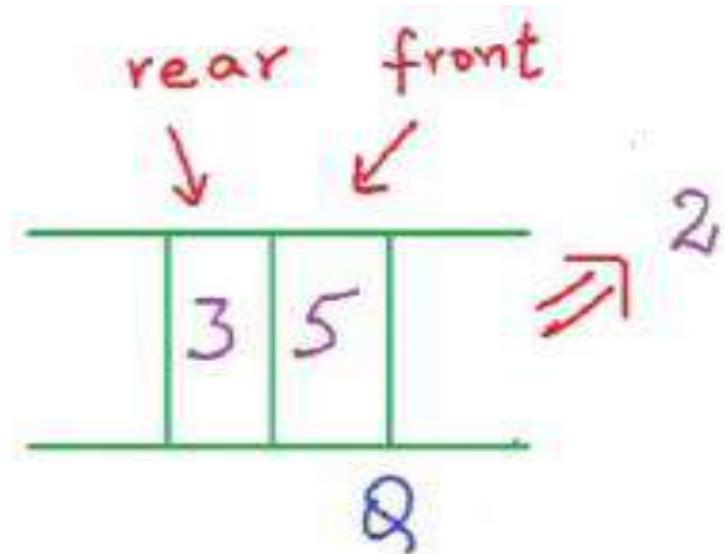
IsEmpty(): Returns true or false as an answer.

IsFull()



Each operation takes constant time, therefore has O(1) time complexity.

# Example



**Enqueue (2)**

**Enqueue (5)**

**Enqueue (3)**

**Dequeue () → 2**

**Front () → 5**

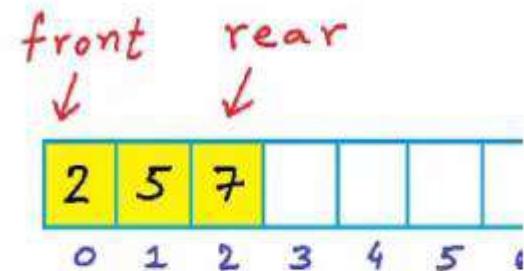
**IsEmpty () → False**

## Applications:

- Printer queue
- Process scheduling

# Array implementation of queue (Pseudocode)

```
int A[10]
front ← -1
rear ← -1
IsEmpty() {
    if (front == -1 && rear == -1)
        return true
    else
        return false}
Enqueue(x) {
    if IsFull()
        return
    else if IsEmpty()
        front ← rear ← 0
    else
        rear ← rear+1
    A[rear]← x
}
```



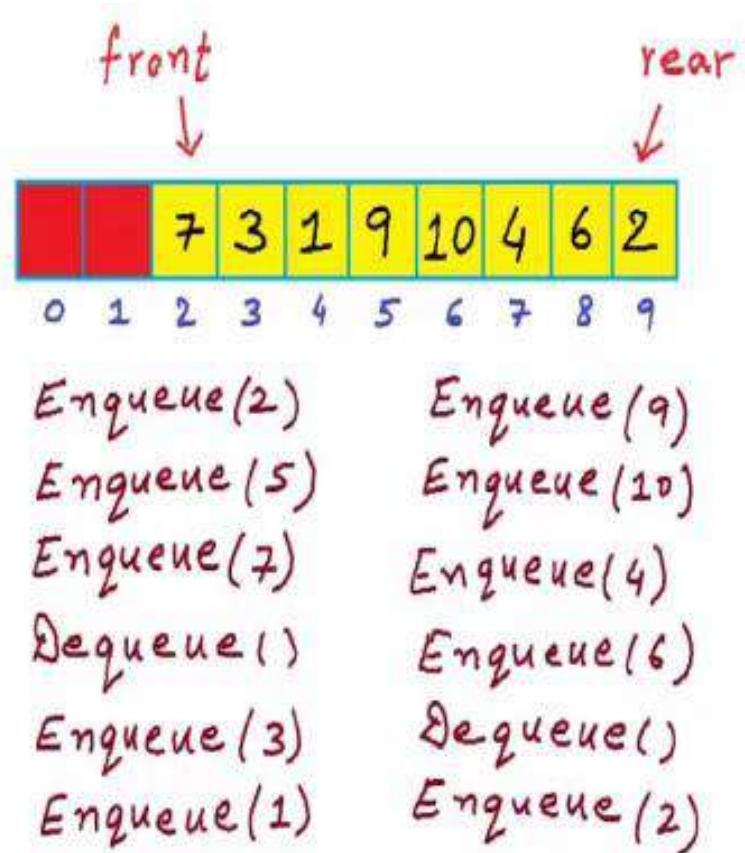
Enqueue(2)

Enqueue(5)

Enqueue(7)

# Array implementation of queue (Pseudocode)

```
Dequeue () {  
    if IsEmpty () {  
        return  
    else if (front == rear)  
        front ← rear ← -1  
    else{  
        front ← front+1  
    }  
}
```



At this stage, we cannot Enqueue an element anymore.

# Queue

## Data Structure

```
#define MAX_QUEUE_SIZE 100

typedef struct{
    int value;
}element;

element queue[MAX_QUEUE_SIZE];
int front=-1;
int rear=-1;
```

# Add Queue

```
void addq( element item)
{
    if(rear == MAX_QUEUE_SIZE-1) {
        isFull();
        return;
    }
    queue[++rear]=item;
}
```

# Delete Queue

```
element deleteq(element item)
{
    if(front == rear)
        return isEmpty();
    return queue[++front];
}
```

# Circular Queue

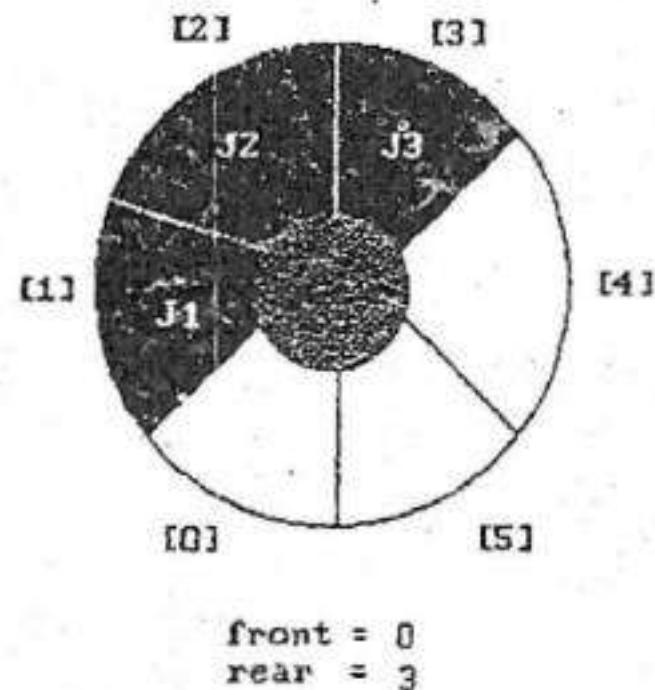
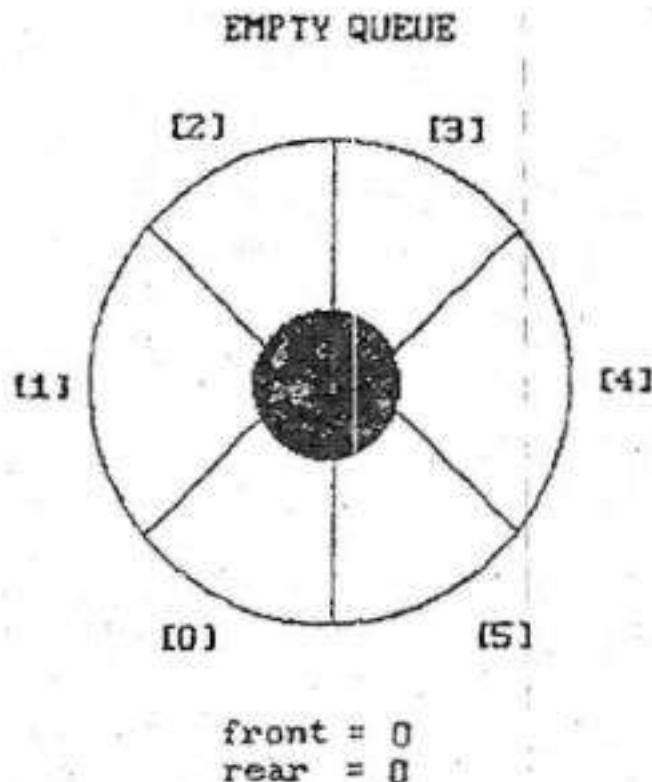
- When the queue is full  
(the rear index equals to MAX\_QUEUE\_SIZE)
  - We should move the entire queue to the left
  - Recalculate the rear

Shifting an array is time-consuming!

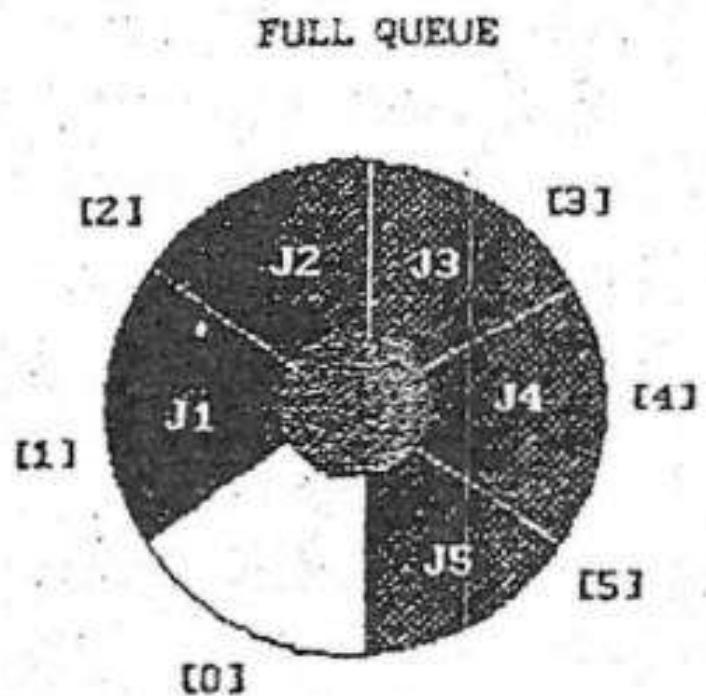
- $O(\text{MAX\_QUEUE\_SIZE})$

# Circular Queue

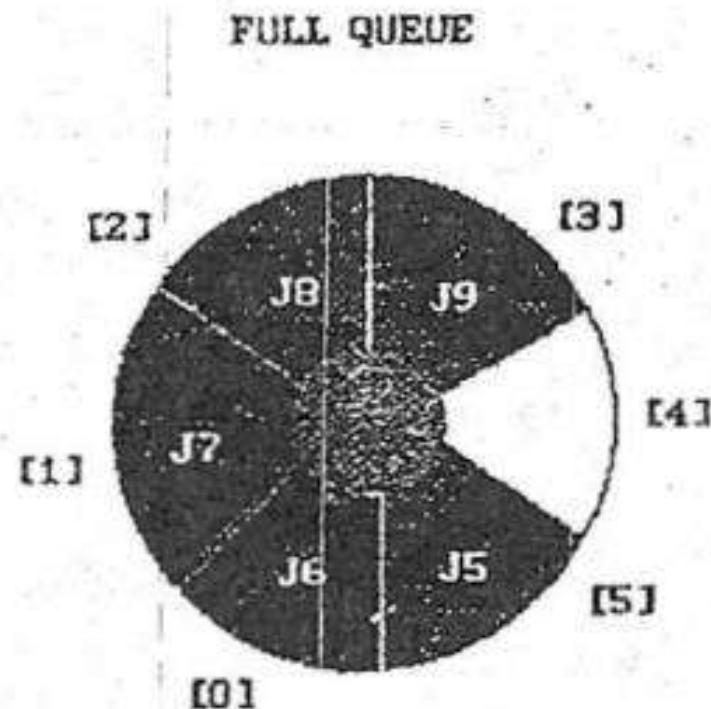
- More efficient queue representation



# Full Circular Queue



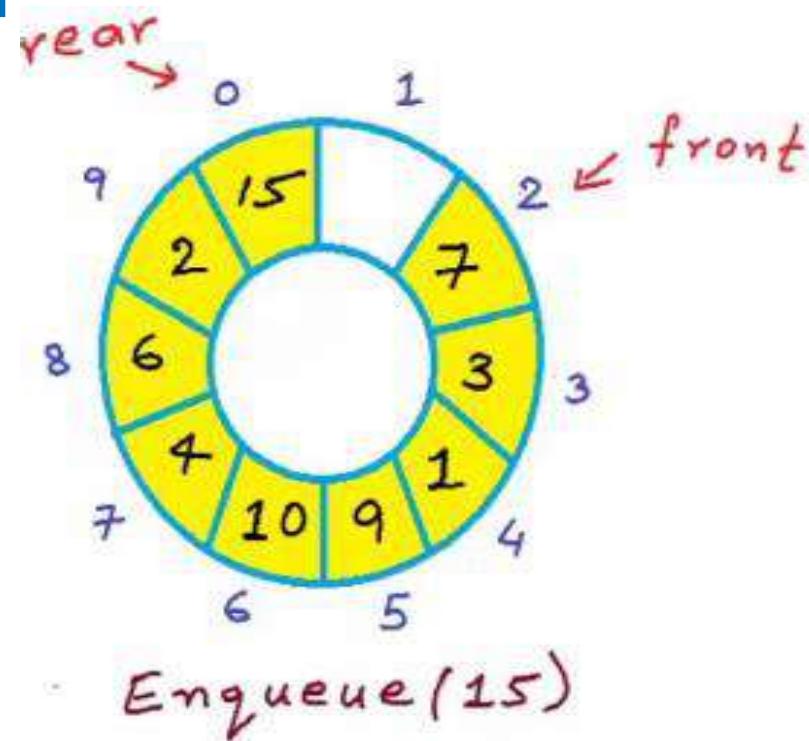
front = 0  
rear = 5



front = 4  
rear = 3

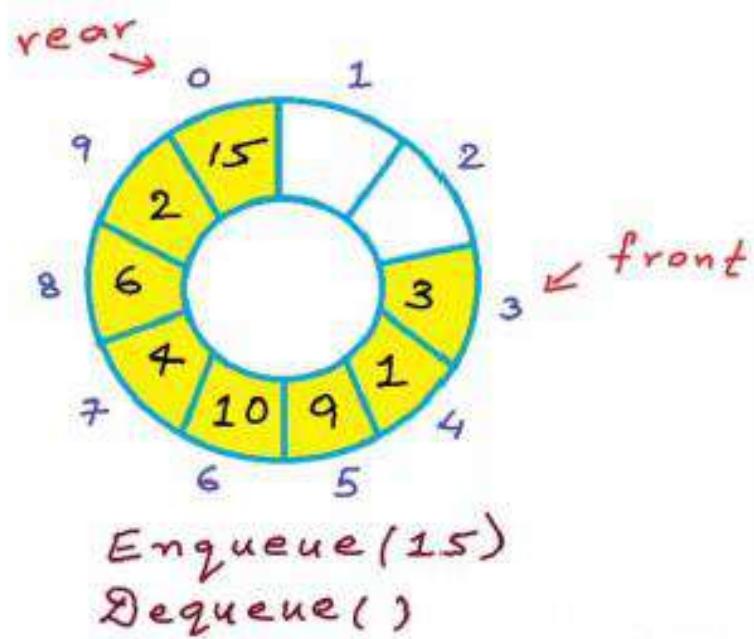
# Enqueue for circular array (Pseudocode)

```
Current position = i  
Next position = (i+1) % N  
previous position = (i+N-1) % N  
Enqueue(x) {  
    if (rear+1) % N == front  
        return  
    else if IsEmpty()  
        front ← rear ← 0  
    else  
        rear ← (rear+1) % N  
    A[rear] ← x  
}
```



# Dequeue for circular array (Pseudocode)

```
Dequeue(x) {  
    if IsEmpty()  
        return  
    else if(front == rear)  
        front ← rear ← -1  
    else  
        front ← (front+1)%N  
}
```



# Add Circular Queue

```
void addcircularq( element item)
{
    rear=(rear+1)% MAX_QUEUE_SIZE;
    if(front == rear){
        isFull(rear);
        return;
    }
    queue[rear]=item;
}
```

# Delete Circular Queue

```
void deletecircularq()
{
    if(front == rear)
        return isEmpty();
    front=(front+1)% MAX_QUEUE_SIZE;
    return queue[front];
}
```

# **BBM 201**

# **DATA STRUCTURES**

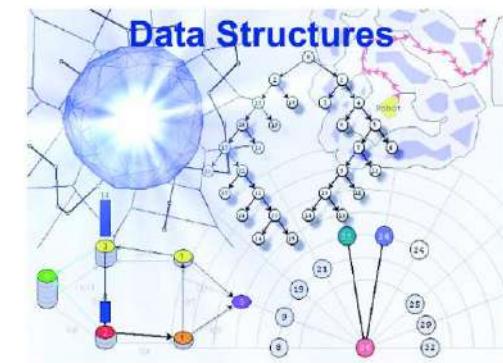
---

## **Lecture 6:**

## EVALUATION of EXPRESSIONS



**2018-2019 Fall**



# Evaluation of Expressions

- Compilers use stacks for the arithmetic and logical expressions.
- **Example:**  $x=a/b-c+d^*e-a^*c$
- If  $a=4$ ,  $b=c=2$ ,  $d=e=3$  what is  $x$ ?
  - $((4/2)-2)+(3^3)-(4^2)$ , ('/' and '\*' have a priority)
- There may be also parenthesis, such as:
  - $a/(b-c)+d^*(e-a)^*c$
  - **How does the compiler solve this problem?**

# Infix, prefix, postfix

- Normally, we use ‘infix’ notation for the arithmetic expressions:
  - Infix notation:  $a+b$
- However, there is also ‘prefix’ and ‘postfix’ notation:
  - Prefix notation:  $+ab$
  - Postfix notation:  $ab+$
- Infix :  $2+3*4$
- Postfix:  $234*+$
- Prefix:  $+2*34$

# Prefix

$$+ 2 * 3 5 =$$

$$= + 2 \underline{* 3 5}$$

$$= + 2 \underline{15} = 17$$

$$* + 2 3 5 =$$

$$= * \underline{+ 2 3 5}$$

$$= * \underline{5 5} = 25$$

# Postfix

$$\begin{aligned} 2 & \ 3 \ 5 \ * \ + \ = \\ & = 2 \ \underline{3 \ 5 \ *} \ + \\ & = \underline{2 \ 15 \ +} \ = 17 \end{aligned}$$

$$\begin{aligned} 2 & \ 3 \ + \ 5 \ * \ = \\ & = \underline{2 \ 3 \ +} \ 5 \ * \\ & = \underline{5 \ 5 \ *} \ = 25 \end{aligned}$$

# How to convert infix to prefix?

Move each operator to the left of the operands:

$$((A + B) * (C + D))$$


$$(+ A B * (C + D))$$

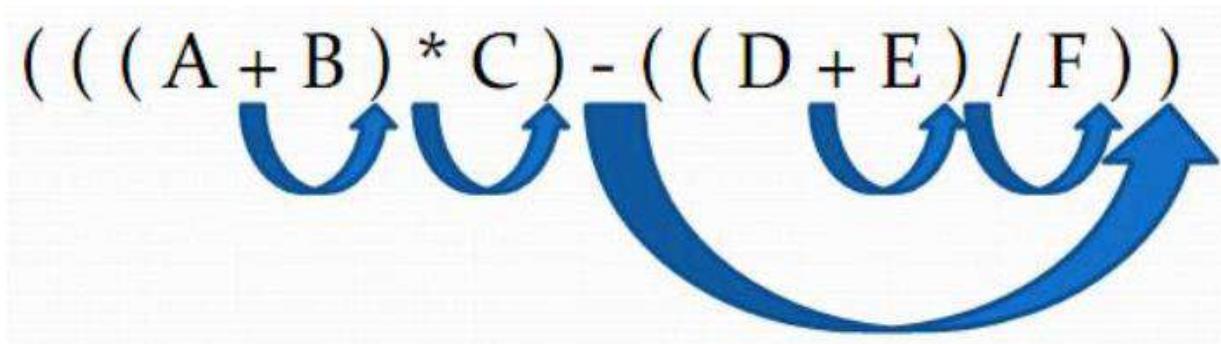

$$* + A B (C + D)$$


$$* + A B + C D$$

Operand order does not change!

# How to convert infix to postfix?

Move each operator to the right of the operands:



- $(( AB+* C) - ( ( D + E ) / F ))$
- $(AB+C* - ( ( D + E ) / F ))$
- $AB+C* ( ( D + E ) / F )-$
- $AB+C* (DE+ / F )-$
- $A B + C * D E + F / -$

Operand order still does not change!

# Example - 1

- Infix:  $(a+b)^*c-d/e$
- Postfix: ???
- Prefix: ???

# Example – 1 (solution)

- Infix:  $(a+b)^*c-d/e$
- Postfix:  $ab+c^*de/-$
- Prefix:  $-^*+abc/de$

## Example - 2

- Infix:  $a/b-c+d^*e-a^*c$
- Postfix: ???
- Prefix: ???

# Example – 2 (solution)

- Infix:  $a/b-c+d^*e-a^*c$
- Postfix:  $ab/c-de^*+ac^*-$
- Prefix: $-+-/abc^*de^*ac$

# Example – 3

- Infix:  $(a/(b-c+d))^*(e-a)^*c$
- Postfix: ???
- Prefix: ???

# Example – 3 (solution)

- Infix:  $(a/(b-c+d))^*(e-a)^*c$
- Postfix: abc-d+/ea-\*c\*
- Prefix: \*\*/a+-bcd-eac

# Expressions

| Infix             | Postfix       | Prefix          | Notes  |
|-------------------|---------------|-----------------|--|
| $A * B + C / D$   | $AB * CD / +$ | $+ * A B / C D$ | multiply A and B,<br>divide C by D,<br>add the results |
| $A * (B + C) / D$ | $ABC + * D /$ | $/ * A + B C D$ | add B and C,<br>multiply by A,<br>divide by D          |
| $A * (B + C / D)$ | $ABCD / + *$  | $* A + B / C D$ | divide C by D,<br>add B,<br>multiply by A              |

# Infix, prefix, postfix

| Infix                     | Postfix           | Prefix            |
|---------------------------|-------------------|-------------------|
| $A+B-C$                   | $AB+C-$           | $-+ABC$           |
| $(A+B)^*(C-D)$            | $AB+CD-*$         | $*+AB-CD$         |
| $A^B*C-D+E/F/(G+H)$       | $AB^C*D-EF/GH+/+$ | $+-*^ABCD//EF+GH$ |
| $((A+B)^*C-(D-E))^*(F+G)$ | $AB+C*DE-FG+^*$   | $^-*+ABC-DE+FG$   |
| $A-B/(C*D^E)$             | $ABCDE^*/-$       | $-A/B*C^DE$       |

# Why postfix?

- For the infix expressions we have two problems:
  - Parenthesis
  - Operation precedence
- Example:  $((4/2)-2)+(3*3)-(4*2)$  (infix)
  - $42/2-33^+42^-$  (postfix)

# Operator PRECEDENCE

| Operators |    |    |    |    |        | Associativity | Type           |
|-----------|----|----|----|----|--------|---------------|----------------|
| ++        | -- | +  | -  | !  | (type) | right to left | unary          |
| *         | /  | %  |    |    |        | left to right | multiplicative |
| +         | -  |    |    |    |        | left to right | additive       |
| <         | <= | >  | >= |    |        | left to right | relational     |
| ==        | != |    |    |    |        | left to right | equality       |
| &&        |    |    |    |    |        | left to right | logical AND    |
|           |    |    |    |    |        | left to right | logical OR     |
| ?:        |    |    |    |    |        | right to left | conditional    |
| =         | += | -= | *= | /= | %=     | right to left | assignment     |
| ,         |    |    |    |    |        | left to right | comma          |

Fig. 4.16 Operator precedence and associativity.

Parentheses are used to override precedence.

# EVALUATION OF INFIX OPERATIONS

## (fully Parenthesized)

1. Read one input character
2. Actions at the end of each input

|                    |  |
|--------------------|--|
| Opening brackets   | (2.1) <i>Push</i> into stack and then Go to step (1)   |
| Number             | (2.2) <i>Push</i> into stack and then Go to step (1)   |
| Operator           | (2.3) <i>Push</i> into stack and then Go to step (1)   |
| Closing brackets   | <p>(2.4) <i>Pop</i> from character stack</p> <p>(2.4.1) if it is opening bracket, then discard it,<br/>Go to step (1)</p> <p>(2.4.2) <i>Pop</i> is used four times</p> <p>The first popped element is assigned to op2</p> <p>The second popped element is assigned to op</p> <p>The third popped element is assigned to op1</p> <p>The fourth popped element is the remaining<br/>opening bracket, which can be discarded</p> <p>Evaluate op1 op op2</p> <p>Convert the result into character and<br/><i>push</i> into the stack</p> <p>Go to step (1)</p> |
| New line character | (2.5) <i>Pop</i> from stack and print the answer<br><b>STOP</b>  |

$$(((2 * 5) - (1 * 2)) / (11 - 9))$$

| Input Symbol | Stack (from bottom to top) | Operation             |
|--------------|----------------------------|-----------------------|
| (            | (                          |                       |
| (            | ((                         |                       |
| (            | ((()                       |                       |
| 2            | ((()2                      |                       |
| *            | ((()2*                     |                       |
| 5            | ((()2*5                    |                       |
| )            | ((10                       | $2 * 5 = 10$ and push |
| -            | ((10-                      |                       |
| (            | ((10-(                     |                       |
| 1            | ((10-(1                    |                       |
| *            | ((10-(1*                   |                       |
| 2            | ((10-(1*2                  |                       |
| )            | ((10-2                     | $1 * 2 = 2$ & Push    |
| )            | (8                         | $10 - 2 = 8$ & Push   |
| /            | (8/                        |                       |
| (            | (8/(                       |                       |
| 11           | (8/(11                     |                       |
| -            | (8/(11-                    |                       |
| 9            | (8/(11-9                   |                       |
| )            | (8/2                       | $11 - 9 = 2$ & Push   |
| )            | 4                          | $8 / 2 = 4$ & Push    |
| New line     | Empty                      | Pop & Print           |

# EVALUATION OF INFIX OPERATIONS

## (Not fully Parenthesized)

(1) Read an input character

(2) Actions that will be performed at the end of each input

|                     |   |
|---------------------|---|
| Opening parentheses | (2.1) <i>Push</i> it into character stack and then Go to step 1   |
| Number              | (2.2) <i>Push</i> into integer stack then Go to step 1  |
| Operator            | (2.3) Do the comparative priority check<br><br>(2.3.1) If the character stack's <i>top</i> contains an operator with equal or higher priority,<br>Then <b>process</b><br>(2.3.1.1) If the character stack is empty, Go to step 2.3.2<br>(2.3.1.2) Else, Go to step 2.3<br><br>(2.3.2) Else, <i>Push</i> the input character to the character stack and Go to step 1 |
| Closing par.        | (2.4) <i>Peek</i> the <i>top</i> element of character stack<br><br>(2.4.1) If it is an opening parentheses then <i>Pop</i> from character stack and Go to step 1<br>(2.4.2) Else, <b>process</b> Go to the step 2.4   |
| New line character  | (2.5) If the character stack is not empty<br><br>(2.5.1) Then <b>process</b> and Go to step 2.5<br>(2.5.2) Else, print the result after popping from the integer stack <b>STOP</b>  |

**Process:** (1) *Pop* from character stack to op

- (2) *Pop* from integer stack to op2
- (3) *Pop* from integer stack to op1
- (4) Calculate op1 op op2 and *Push* the result into the integer stack

$$(2^*5-1^*2)/(11-9)$$

| Input Symbol | Character Stack (from bottom to top) | Integer Stack (from bottom to top) | Operation performed                             |
|--------------|--------------------------------------|------------------------------------|---|
| (            | (                                    |                                    |   |
| 2            | ( 2                                  |                                    |   |
| *            | ( * 2                                |                                    | Push as * has higher priority                   |
| 5            | ( * 2 5                              |                                    |   |
| -            | ( * ( - 10                           |                                    | Since '-' has less priority, we do $2 * 5 = 10$ |
|              | ( - 10                               |                                    | We push 10 and then push '-'                    |
| 1            | ( - 10 1                             |                                    |   |
| *            | ( - * 10 1                           |                                    | Push * as it has higher priority                |
| 2            | ( - * 10 1 2                         |                                    |   |
| )            | ( - 10 2                             |                                    | Perform $1 * 2 = 2$ and push it                 |
|              | ( 8                                  |                                    | Pop - and $10 - 2 = 8$ and push, Pop (          |
| /            | / 8                                  |                                    |   |
| (            | / ( 8                                |                                    |   |
| 11           | / ( 8 11                             |                                    |   |
| -            | / ( - 8 11                           |                                    |   |
| 9            | / ( - 8 11 9                         |                                    |   |
| )            | / 8 2                                |                                    | Perform $11 - 9 = 2$ and push it                |
|              | 4                                    |                                    | Perform $8 / 2 = 4$ and push it                 |
| New line     | 4                                    |                                    | Print the output, which is 4                    |

# Evaluation of a prefix operation

**Input:** / - \* 2 5 \* 1 2 - 11 9

**Output:** 4

Data structure requirement: a character stack and an integer stack

1. Read one character input at a time and keep pushing it into the character stack until the new line character is reached
2. Perform *pop* from the character stack. If the stack is empty, go to step (3)

Number (2.1)      *Push* into the integer stack and then go to step (2)

Operator (2.2)      Assign the operator to op

*Pop* a number from integer stack and assign it to op1

*Pop* another number from integer stack and assign it to op2

                        Calculate op1 op op2 and push the output into the int. stack.

                        Go to step (2)

3. *Pop* the result from the integer stack and display the result

**/ - \* 2 5 \* 1 2 -11 9**

|    |                            |                        |  |
|----|----------------------------|------------------------|--|
| /  | /                          |                        |  |
| -  | /-                         |                        |  |
| *  | / - *                      |                        |  |
| 2  | / - * 2                    |                        |  |
| 5  | / - * 2 5                  |                        |  |
| *  | / - * 2 5 *                |                        |  |
| 1  | / - * 2 5 * 1              |                        |  |
| 2  | / - * 2 5 * 1 2            |                        |  |
| -  | / - * 2 5 * 1 2 -          |                        |  |
| 11 | / - * 2 5 * 1 2 - 11       |                        |  |
| 9  | / - * 2 5 * 1 2 - 11 9     |                        |  |
| \n | / - * 2 5 * 1 2 - 11 9     |                        |  |
|    | / - * 2 5 * 1 2 - 9 11     |                        |  |
|    | / - * 2 5 * 1 2 2          | 11 - 9 = 2             |  |
|    | / - * 2 5 * 1 2 2          |                        |  |
|    | / - * 2 5 * 1 2 2 1        |                        |  |
|    | / - * 2 5 2 2              | 1 * 2 = 2              |  |
|    | / - * 2 5 2 2 5            |                        |  |
|    | / - * 2 5 2 2 5 2          |                        |  |
|    | / - 2 2 10 5 * 2 = 10      |                        |  |
|    | / 2 8 10 - 2 = 8           |                        |  |
|    | Stack is empty 4 8 / 2 = 4 |                        |  |
|    |                            | Stack is empty Print 4 |  |

# POSTFIX

Compilers typically use a parenthesis-free notation (postfix expression).

The expression is evaluated from the left to right using a stack:

- when encountering an operand: push it
- when encountering an operator: pop two operands, evaluate the result and push it.

# Evaluation of a postfix expression

| Token | Stack                 |     |     | Top |
|-------|-----------------------|-----|-----|-----|
|       | [0]                   | [1] | [2] |     |
| 4     | 4                     |     |     | 0   |
| 2     | 4                     | 2   |     | 1   |
| /     | 4/2                   |     |     | 0   |
| 2     | 4/2                   | 2   |     | 1   |
| -     | (4/2)-2               |     |     | 0   |
| 3     | (4/2)-2               | 3   |     | 1   |
| 3     | ((4/2)-2)             | 3   | 3   | 2   |
| *     | ((4/2)-2)             | 3*3 |     | 1   |
| +     | ((4/2)-2)+(3*3)       |     |     | 0   |
| 4     | ((4/2)-2)+(3*3)       | 4   |     | 1   |
| 2     | ((4/2)-2)+(3*3)       | 4   | 2   | 2   |
| *     | ((4/2)-2)+(3*3)       | 4*2 |     | 1   |
| -     | ((4/2)-2)+(3*3)-(4*2) |     |     | 0   |

**62/3-42\*+**

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

# How to evaluate a postfix evaluation?

```
typedef enum
{left_parent,right_parent,add,subtract,multiply,divide,eos,operand}
precedence;

char expr[] = "422-3+/34-*2*";

precedence get_token(char* symbol, int* n){

    *symbol=expr[(*n)++];
    switch(*symbol){
        case '(': return left_parent;
        case ')': return right_parent;
        case '+': return add;
        case '-': return subtract;
        case '/': return divide;
        case '*': return multiply;
        case '\0': return eos;
        default: return operand;
    }
}
```

# How to evaluate a postfix evaluation?

```
float eval(void){  
    char symbol;  
    precedence token;  
    float op1, op2;  
    int n=0;  
    int top=-1;  
  
    token = get_token(&symbol, &n); //take a token  
  
    while(token!=eos){ //end of string?  
        if(token==operand)  
            push(&top, symbol- '0');  
        else{  
            op2=pop(&top);  
            op1=pop(&top);  
            switch(token){  
                case add: push(&top,op1+op2);  
                case subtract: push(&top,op1-op2);  
                case multiply: push(&top,op1*op2);  
                case divide: push(&top,op1/op2);  
            }  
        }  
        token=get_token(&symbol,&n);  
    }  
    return pop(&top);  
}
```

# CONVERT an INFIX to POSTFIX

**a+b\*c**

| Token | Stack |     |     | Top | Output |
|-------|-------|-----|-----|-----|--------|
|       | [0]   | [1] | [2] |     |        |
| a     |       |     |     | -1  | a      |
| +     | +     |     |     | 0   | a      |
| b     | +     |     |     | 0   | ab     |
| *     | +     | *   |     | 1   | ab     |
| c     | +     | *   |     | 1   | abc    |
| eos   |       |     |     | -1  | abc*+  |

**a\*(b+c)\*d**

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

# How to convert infix to postfix?

```
char expr[]=" (4/(2-2+3))*(3-4)*2";
static int stack_pre[]={0,19,12,12,13,13,13,0};
static int pre[]={20,19,12,12,13,13,13,0};

void postfix(void){
    char symbol;
    precedence token;
    int n=0;
    int top=0;
    stack[0]=eos;

    for(token=get_token(&symbol,&n);token!=eos;token=get_token(&symbol,&n)){
        if(token==operand)
            printf("%c", symbol);
        else if(token==right_parent)
            while(stack[top]!=left_parent)
                print_token(pop(&top));
            pop(&top);
        }
        else{
            while(stack_pre[stack[top]]>=pre[token])
                print_token(pop(&top));
            push(&top, token);
        }
    }
    while((token=pop(&top))!=eos)
        print_token(token)
}
```

Exercise to do at home:

1. Write the code that converts infix to prefix.
2. Write the code that evaluates a prefix expression.

# **BBM 201**

# **DATA STRUCTURES**

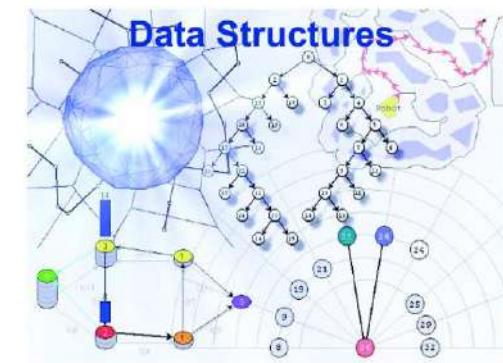
---

## Lecture 7:

Introduction to the Lists  
(Array-based linked lists)



2018-2019 Fall

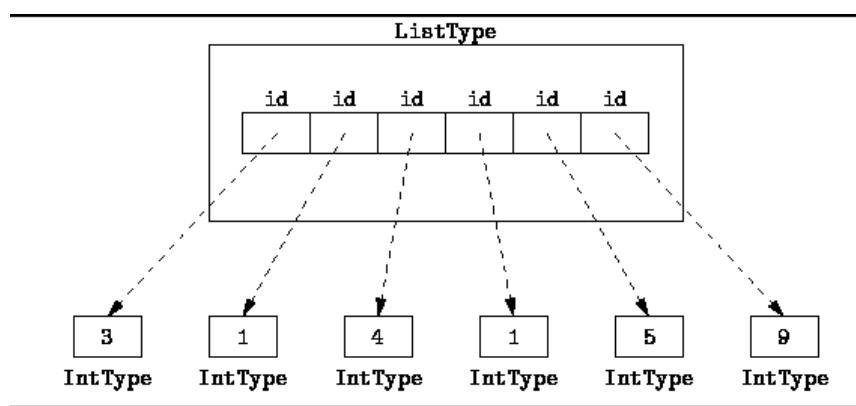


# Lists



# Lists

- We used successive data structures up to now:
  - If  $a_{ij}$  in the memory location  $L_{ij}$ , then  $a_{ij+1}$  is in  $L_{ij}+c$  ( $c$ : constant)
  - In a queue, if the  $i^{\text{th}}$  item is in  $L_i$ ,  $i+1^{\text{th}}$  item is in  $(L_i+c) \% n$ . (i.e. circular queue)
  - In a stack, if the top item is in  $L_T$ , the below item is in  $L_T-c$ .



Insertion and deletion:  
 $O(1)$

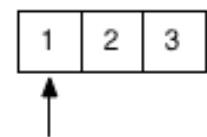
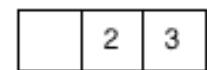
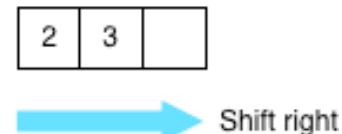
# Sequential Access (ascending or descending)

## Example 1:

- Alphabetically ordered lists:

|     |           |     |     |       |
|-----|-----------|-----|-----|-------|
| Ape | Butterfly | Cat | Dog | Mouse |
|-----|-----------|-----|-----|-------|

- Delete 'Ape', what happens?
- Delete 'Cat', what happens?
- Add 'Bear', what happens?
- Add 'Chicken', what happens?



Save new  
element

# Sequential Access (ascending or descending)

## Example 2:

- The result of the multiplication of two polynomials
  - $(x^7 + 5x^4 - 3x^2 + 4)(3x^5 - 2x^3 + x^2 + 1)$

|    |    |   |   |    |     |   |   |    |   |    |    |     |     |
|----|----|---|---|----|-----|---|---|----|---|----|----|-----|-----|
| 3  | -2 | 1 | 1 | 15 | -10 | 5 | 5 | -9 | 6 | -3 | 12 | ... | ... |
| 12 | 10 | 9 | 7 | 9  | 7   | 6 | 4 | 7  | 5 | 4  | 5  | ... | ... |

- Powers are not ordered. So either we need to sort or shift in order to solve this problem.

# Sorted items

- We want to keep the items sorted, and we want to avoid the sorting cost.
  - We may need to sort after each insertion of a new item.
  - Or we need to do shifting.

What is the solution?

# Towards the Linked List

- Each item has to have a second data field – link.
  - Each item has two fields: **data** and **link**.

# Linked List

```
#define MAX_LIST 10  
#define TRUE 1  
#define FALSE 0  
#define NULL -1
```

```
typedef struct{  
    char name[5];  
    //other fields  
    int link;  
}item;  
  
item linkedlist[MAX_LIST];  
int free_;
```

# Linked List

## -make empty list

```
void make emptylist(void)
{
    int i;
    for(i=0;i<MAX_LIST-1;i++)
        list[i].link=i+1; //every item points the next

    linkedlist [MAX_LIST-1].link=NULL; //last item
    free_=0;
}
```

# Linked List

## -get item

*Returns a free item from the list:*

```
int get_item(int* r)
{
    if(free_== NULL) //there is no item to get
        return FALSE;
    else{
        *r=free_; //get the item which is pointed by free_
        free_=linkedlist[free_].link;//points next free item
        return TRUE;
    }
}
```

# Linked List

## -return item

*Free the item:*

```
void return_item(int r)
{
    linkedlist[r].link=free_;      //return item that is pointed by r
    free_=r; //free the item
}
```

|     | <b>name</b> | <b>link</b> |
|-----|-------------|-------------|
| [0] |             | 1           |
| [1] |             | 2           |
| [2] |             | 3           |
| [3] |             | 4           |
| [4] |             | 5           |
| [5] |             | 6           |
| [6] |             | 7           |
| [7] |             | 8           |
| ... | ....        | ...         |
| ... | ....        | ...         |
|     |             | -1          |

**free\_ = 0**

|     | <b>name</b> | <b>link</b> |
|-----|-------------|-------------|
| [0] | Arzu        | 1           |
| [1] | Ayse        | 2           |
| [2] | Aziz        | 3           |
| [3] | Bora        | 4           |
| [4] | Kaan        | 5           |
| [5] | Muge        | 6           |
| [6] | Ugur        | -1          |
| [7] |             | 8           |
|     |             | ...         |
|     |             | -1          |

**free\_ = 7**  
**List starts at 0 (\*list=0)**

|      | <b>name</b> | <b>link</b> |
|------|-------------|-------------|
| [0]  | Arzu        | 1           |
| [1]  | Ayse        | 2           |
| [2]  | Aziz        | 3           |
| [3]  | Bora        | 4           |
| [4]  | Kaan        | 7           |
| [5]  | Muge        | 6           |
| [6]  | Ugur        | -1          |
| [7]  | Leyla       | 5           |
| [8]  | .....       | 9           |
| .... | .....       | ....        |
|      |             | -1          |

|     | <b>name</b> | <b>link</b> |
|-----|-------------|-------------|
| [0] | Eyup        | 4           |
| [1] | Ayse        | 2           |
| [2] | Aziz        | 3           |
| [3] | Bora        | 0           |
| [4] | Kaan        | 7           |
| [5] | Muge        | 6           |
| [6] | Ugur        | -1          |
| [7] | Leyla       | 5           |
| [8] |             | 9           |
|     |             | -1          |

free\_ = 8 (“Leyla” added)  
 \*list = 0

free\_ = 0 (“Arzu” deleted)  
 free\_ = 8 (“Eyup” added)  
 \*list = 1

# Linked List

## –insert item

```
void insert_item(char name[], int* list)
{
    int r, q, p;
    if(get_item(&r)){
        strcpy(linkedlist[r].name, name);
        q = NULL;
        p = *list;
        while( p != NULL && strcmp(linkedlist[p].name, name) < 0 ) { //search right position
            q = p;
            p = linkedlist [p].link;
        }
        if(q == NULL){ //new item is inserted to the front of the list.
            *list = r;
            linkedlist [r].link = p;
        }
        else{ //new item is inserted in the middle
            linkedlist [q].link = r;
            linkedlist [r].link = p;
        }
    }
    else printf( "\n not enough free space!!" );
}
```

# Linked List

## —delete item

```
void delete_item(char name[], int* list)
{
    int q,p;
    q = NULL;
    p = *list;
    int l;
    while( p != NULL && (l = strcmp(linkedlist[p].name, name)) < 0 ) { //search for the item
        q = p;
        p = linkedlist [p].link;
    }
    if(p == NULL || l>0) //end of the list
        printf( "\n %s cannot be found!! ", name);
    else if( q == NULL){ //the first item of the list will be deleted.
        *list = linkedlist [p].link;
        return_item(p);
    }
    else{ //get the item pointed by 'p'
        linkedlist [q].link = linkedlist [p].link;
        return_item(p);
    }
}
```

# **BBM 201**

# **DATA STRUCTURES**

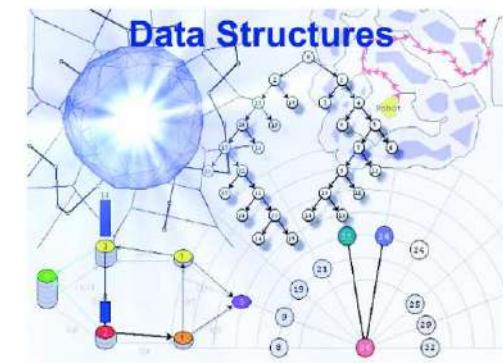
---

## Lecture 8:

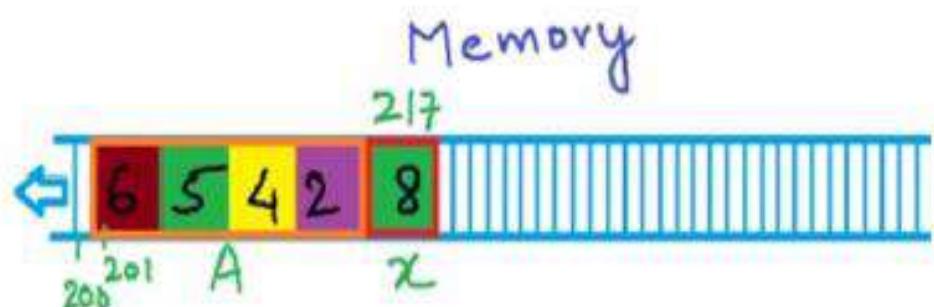
Dynamically Allocated Linked Lists



2018-2019 Fall



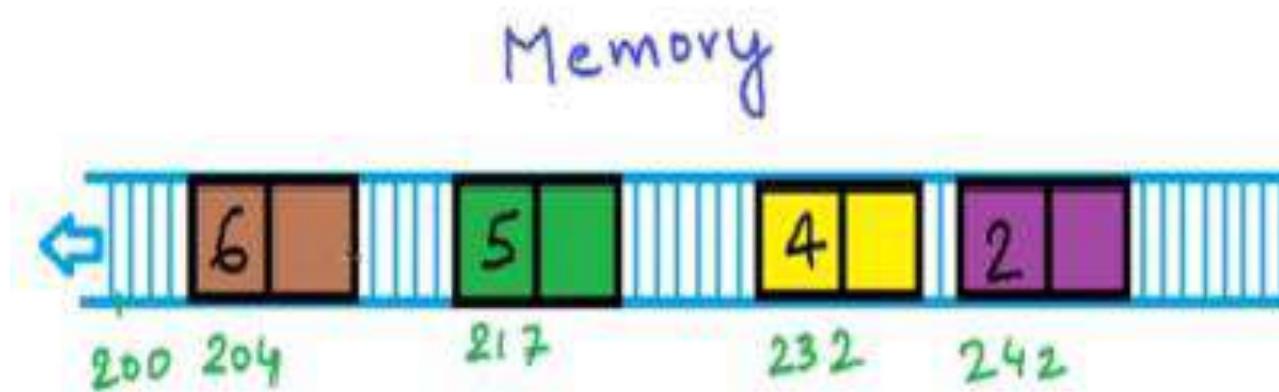
```
int x;  
x = 8;  
int A[4];
```



Memory  
Manager

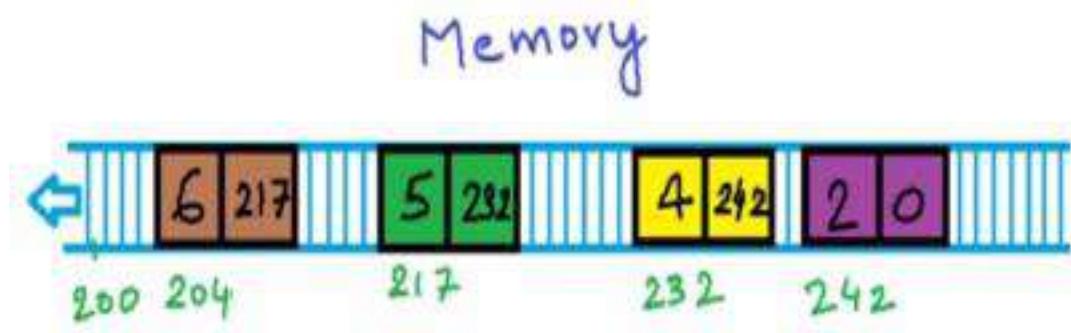
An array is stored as one contiguous block of memory.  
How can we add a fifth element to the array A above???

If we used dynamic memory allocation, we need to use realloc.  
Otherwise, we can use a linked list.



To input elements to the linked list, the memory manager finds an address for each element one by one.

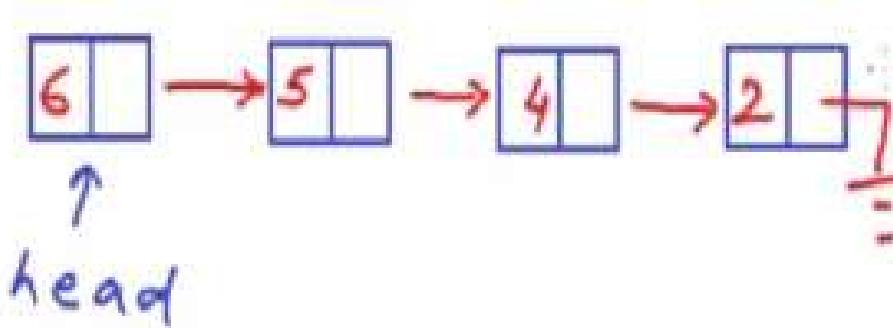
We store not only the value of each element but also the **address of the following element** for each element.



## Struct Node

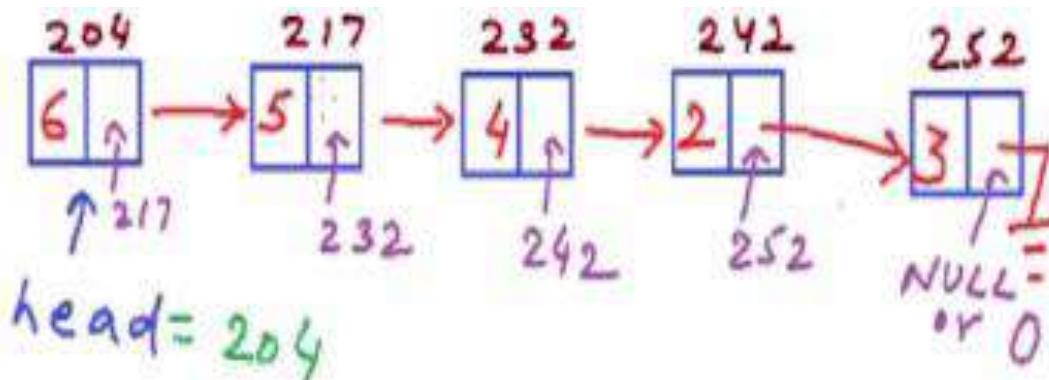
```
{  
    int data; // 4 bytes  
    Node* next; // 4 bytes  
}
```

- For each element (or **node**) **two fields** are stored, each costing four bytes.



- The first node is called the **head node**. The address of the **last node** is **NULL or 0**.
- The address of the head node gives us access to the complete list.
- To access a node, we need to traverse ALL nodes with smaller index.

# Adding a new node

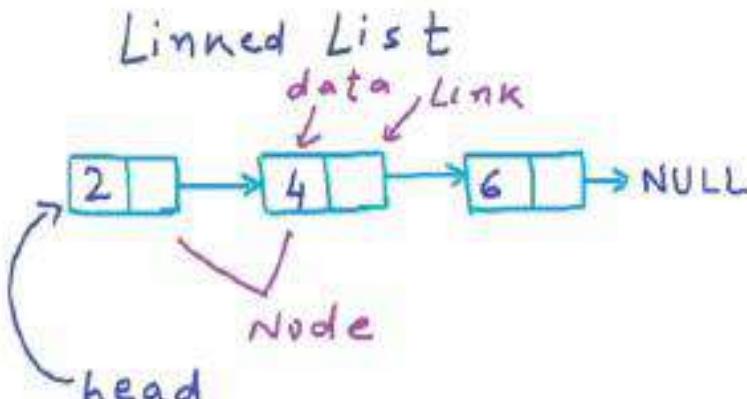
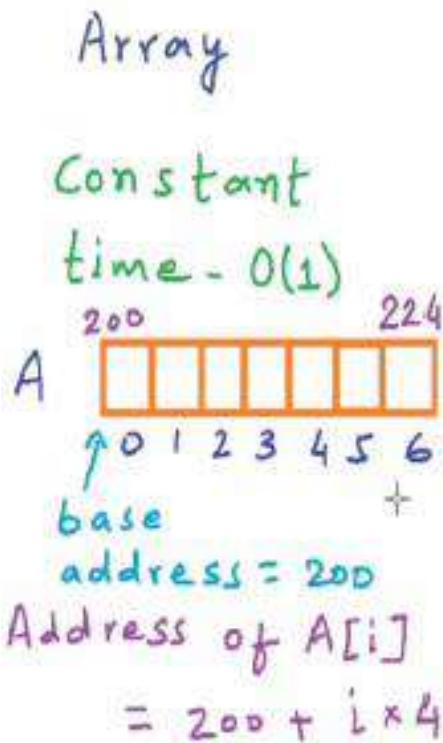


Create a node independently with some memory location and adjust the links properly. Say the **node 3** gets **address 252**.

- The linked list is always **identified** by the address of the head node.
- Unlike array, it costs **O(n)** to access an element of a linked list.

# Array vs. Linked List:

## 1) Cost of accessing an element



Average case :  $O(n)$

- If we know the starting address of the array, we can calculate the address of the  $i$ 'th element in the array. This takes **constant ( $O(1)$ ) time** for any element in the array!
- To find the address of the  $i$ 'th element in the linked list, we need to traverse all elements until that element **( $O(n)$  time)**.

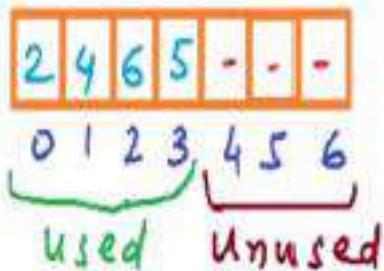
# Array vs. Linked List:

## 2) Memory requirements

Array

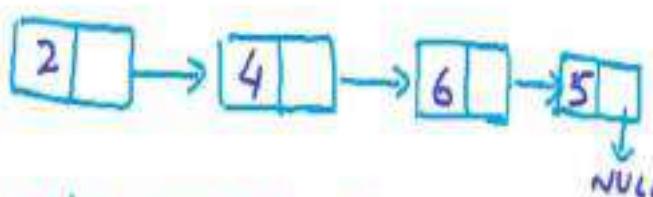
- Fixed size

A



$$7 \times 4 = 28 \text{ bytes}$$

Linked List



- NO unused memory
- extra memory for pointer variables

$$8 \times 3 = 24 \text{ bytes}$$

32

- Before creating an array, we need to know its possible size.
- For linked list, we ask from memory one node at a time, but we use twice the size since also the address of each consecutive node is stored.

If we have a data of complex type, then for each element in the linked list, 20 bytes are used.

The strategy to decide which one to use depends on the case.

# Array vs. Linked List:

## 2) Memory requirements

| Array  | Linked List                                      |
|--|--|
| Has fixed size                                 | No unused memory                                 |
|  | Extra memory for pointer variables               |
| Memory may not be available as one large block | Memory may be available as multiple small blocks |

# Array vs. Linked List:

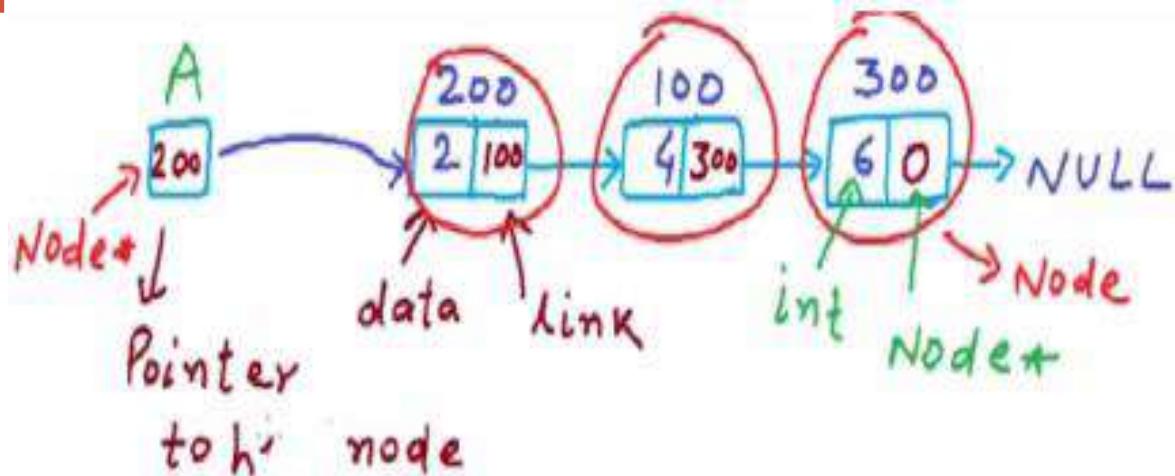
## 3) Cost of inserting an element

| Cost of inserting a new element (worst case): | Array                       | Linked List |
|---|-----------------------------|-------------|
| a) At the beginning                           | O(n)                        | O(1)        |
| b) At the end                                 | O(1) (if array is not full) | O(n)        |
| c) At i'th position                           | O(n)                        | O(n)        |

- To insert an element to the beginning of an array all elements need to be shifted by one to the next address.
- To add an element to the end of a linked list all elements need to be traversed.

# Linked List: Implementation in C and C++

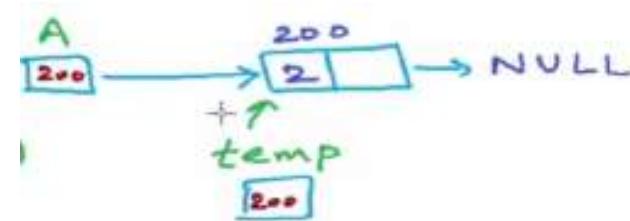
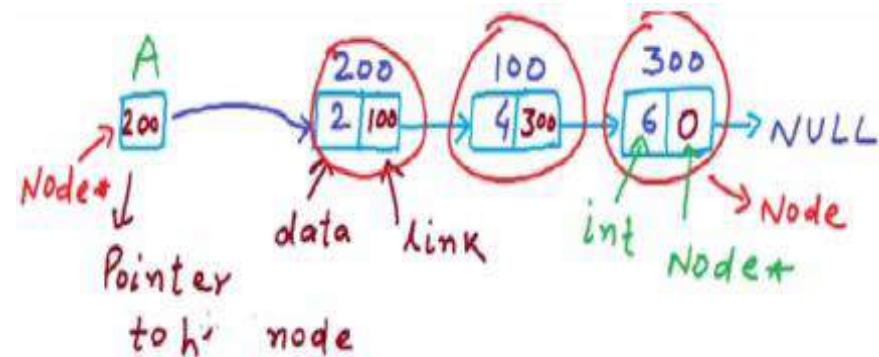
```
Struct Node  
{  
    int data;  
    Node* next;  
}  
Node* A;  
A = NULL; //empty list  
Node* temp = (Node*)malloc(sizeof(Node))
```



- We define such data type using **Structure**. Integer is the type of the variable data stored as the element of the linked list.
- The second field called **link** is of type pointer.
- The last line creates one node in the memory.

# Dereferencing

```
Struct Node  
{  
    int data;  
    Node* next;  
}  
  
Node* A;  
A = NULL; //empty list  
Node* temp = (Node*)malloc(sizeof(Node))  
(*temp).data = 2;  
(*temp).link = NULL;  
A = temp;
```

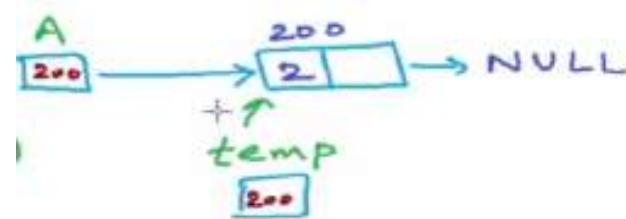
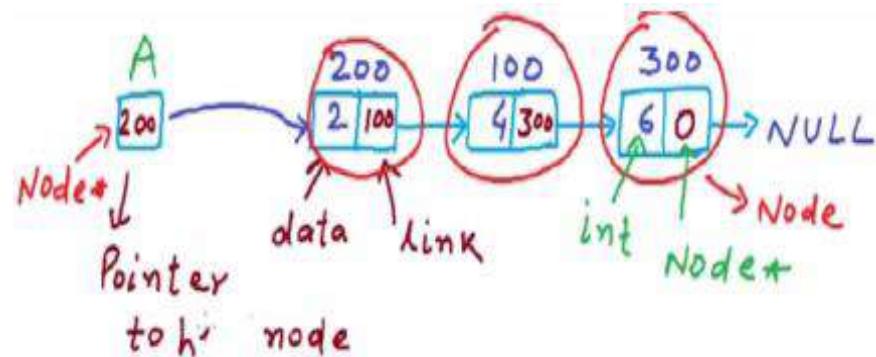


- The **data part** of this node is **2** and the temp variable is pointing to it.
- The **link part** is **NULL** since it is the last node.
- Finally, write the **address** of the newly created node to **A**.

# C++ implementation

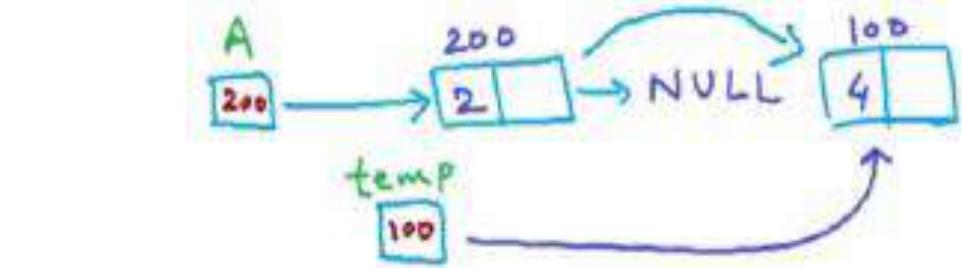
Struct Node

```
{  
    int data;  
    Node* next;  
}  
  
Node* A;  
A = NULL;  
Node* temp = new Node();  
temp->data = 2;  
temp->link = NULL;  
A = temp;
```

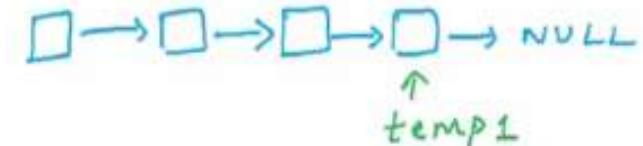


# Traversal of a list

```
Node* A;  
A = NULL;  
Node* temp = new Node();  
(*temp).data = 2;  
(*temp).link = NULL;  
A = temp;  
temp = new Node();  
temp->data = 4;  
temp->link = NULL;
```



```
Node* templ = A;  
while(templ != NULL) {  
    print "templ->data";  
    templ = templ->link;  
}
```



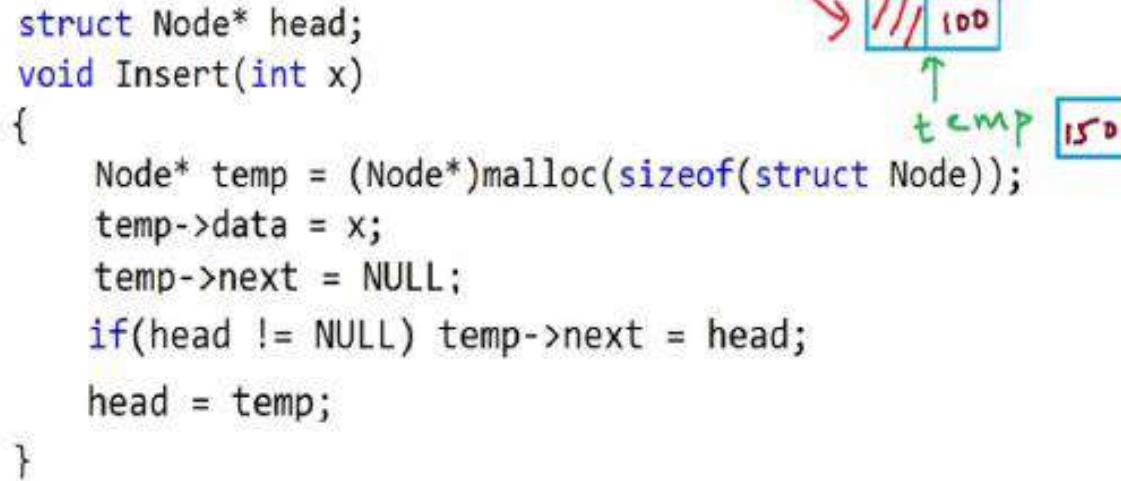
- **temp** stores the **address of the new node**.
- We need to **record the address of the new node** and to do so we need to traverse the whole list to go to the end of the list.
- While traversing, if the link of the node is not **NULL**, we can move to the next node.
- Finally, the loop prints the elements in this list.

# Inserting a node at the beginning

```
struct Node {  
    int data;  
    struct Node* next;  
};  
struct Node* head; // global variable, can be accessed anywhere  
void Insert(int x);  
void Print();  
int main() {  
    head = NULL; // empty list;  
    printf("How many numbers?\n");  
    int n,i,x;  
    scanf("%d",&n);  
    for(i = 0;i<n;i++){  
        printf("Enter the number \n");  
        scanf("%d",&x);  
        Insert(x);  
        Print();  
    }  
}
```

- Pointer storing the address of the next node is called next. (In C++, only `Node* next;` is written.)
- Insert each number  $x$  into the linked list by calling a method insert and print it.

# Implementing the insert function



- If the list is empty as above, we want `head` to point to this new node.
- If the list is not empty, by the line `temp->next = head`, the new node points to the address 100.
- Next, to cut the link from `head` to 100, we have the line `head = temp`.

# Implementing the print function

```
struct Node* head; // global variable, can be accessed anywhere
void Insert(int x)
{
    struct Node* temp = (Node*)malloc(sizeof(struct Node));
    temp->data = x;
    temp->next = head;
    head = temp;
}
void Print()
{
    struct Node* temp = head;
    printf("List is: ");
    while(temp != NULL)
    {
        printf(" %d",temp->data);
        temp= temp->next;
    }
    printf("\n");
}
```

# Implementing the print function

```
void Print() {
    struct Node* temp = head;
    printf("List is: ");
    while(temp != NULL)
    {
        printf(" %d",temp->data);
        temp= temp->next;
    }
    printf("\n");
}

int main() {
    head = NULL; // empty list;
    printf("How many numbers?\n");
    int n,i,x;
    scanf("%d",&n);
    for(i = 0;i<n;i++){
        printf("Enter the number \n");
        scanf("%d",&x);
        Insert(x);
        Print();
    }
}
```

```
C:\Users\animesh\documents\visual studio 2010\Projects\SampleAp
5
Enter the number
2
List is: 2
Enter the number
5
List is: 5 2
Enter the number
2
List is: 5 2
Enter the number
8
List is: 8 5 2
Enter the number
1
List is: 8 5 2
Enter the number
1
List is: 1 8 5 2
Enter the number
10
List is: 10 1 8 5 2
```

Note that each newly entered number is stored as the beginning of the list.

# Implementing the print function

```
void Print(Node* head) {
    printf("List is: ");
    while(head != NULL)
    {
        printf(" %d",head->data);
        head= head->next;
    }
    printf("\n");
}

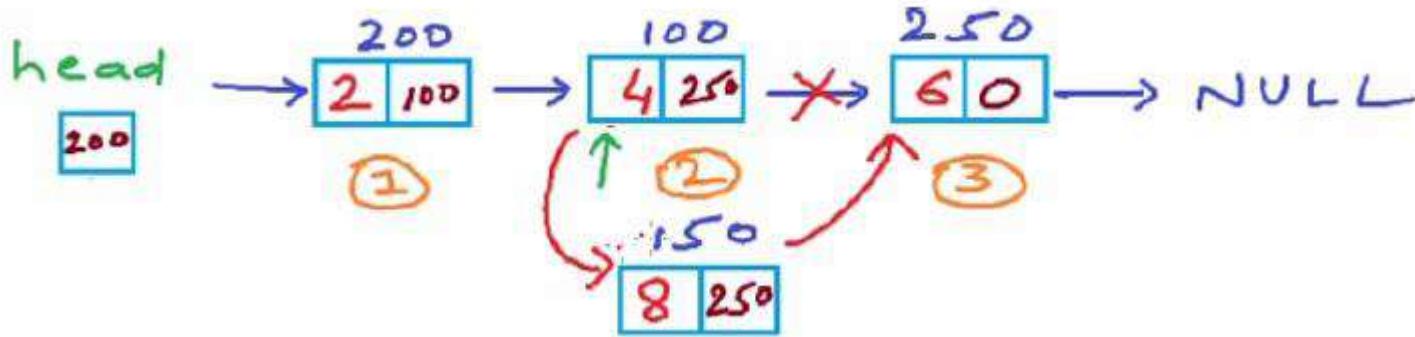
int main() {
    Node* head = NULL; // empty list;
    printf("How many numbers?\n");
    int n,i,x;
    scanf("%d",&n);
    for(i = 0;i<n;i++){
        printf("Enter the number \n");
        scanf("%d",&x);
        head = Insert(head,x);
        Print(head);
    }
}
```

Head will be passed to print as local variable and therefore an argument of the print function.

Since head is a local variable in print, we can change the value of head inside the print function instead of defining a temporary variable temp.

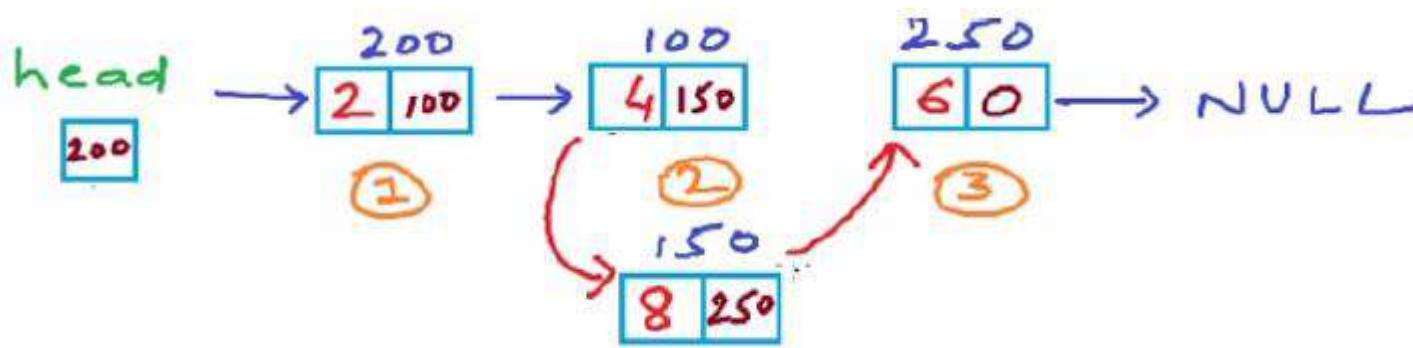
The insert method also passes the return as head.

# Inserting a node at the n'th position



Say, we want to insert a value 8 at 3rd position.

The link field of the new node should be 250 and the link field of the second node should be the address of the new node as shown below.



```
void Insert(int data,int n) {  
    Node* temp1 = new Node();  
    temp1->data = data;  
    temp1->next = NULL;  
    if(n == 1) {  
        temp1->next = head;  
        head = temp1;  
        return;  
    }  
    Node* temp2 = head;  
    for(int i =0;i<n-2;i++) {  
        temp2 = temp2->next;  
    }  
    temp1->next = temp2->next;  
    temp2->next = temp1;  
}
```

In this implementation of **Insert function**, the new node is defined **using C++ syntax** (without using malloc).

Finally, we set the link of the new node to the link of the (n-1)st node and then we set the link of the (n-1)st node to the new node.

```
struct Node {  
    int data;  
    struct Node* next;  
};  
struct Node* head;  
void Insert(int data, int n);  
void Print();  
int main() {  
    head = NULL; //empty list  
    Insert(2,1); //List: 2  
    Insert(3,2); //List: 2,3  
    Insert(4,1); //List: 4,2,3  
    Insert(5,2); //List: 4,5,2,3  
    Print();  
}
```

In the implementation above, again the pointer to the beginning of the linked list, called head, is defined as a global variable.

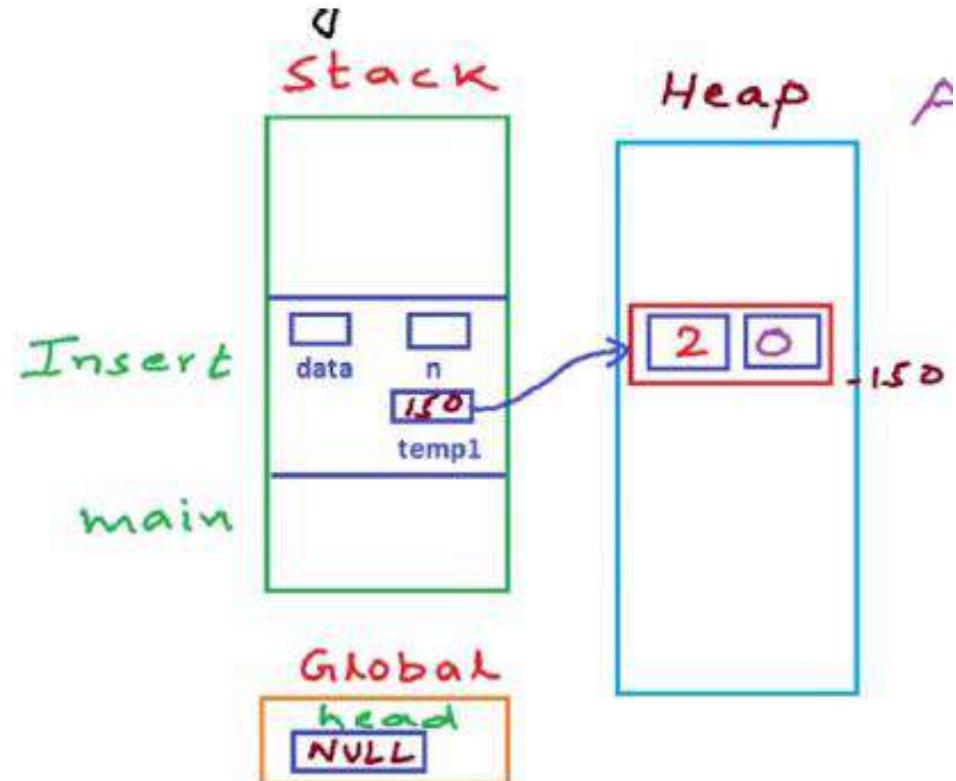
The insert function takes the value of the new node and the position we want to insert the new node.  
Print will print all the numbers in the linked list.

```

void Insert(int data,int n) {
    Node* temp1 = new Node();
    ✓temp1->data = data;
    temp1->next = NULL;
    ✓if(n == 1) {
        temp1->next = head;
        head = temp1;
        return;
    }
    Node* temp2 = head;
    for(int i =0;i<n-2;i++) {
        temp2 = temp2->next;
    }
    temp1->next = temp2->next;
    temp2->next = temp1;
}
int main() {
    ✓head = NULL; //empty list
    ✓Insert(2,1); //List: 2
    Insert(3,2); //List: 2,3
    Insert(4,1); //List: 4,2,3
    Insert(5,2); //List: 4,5,2,3
    Print();
}

```

## Location of data in the memory

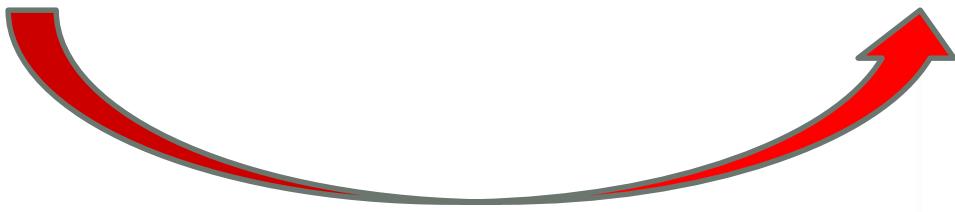


When we store something in the heap using `new` or `malloc`, we do not have a variable name for this and we can only access it through a pointer variable as seen above.

**Deleting a node at n'th position**

```
//Linked List: Delete a node at nth position
#include<stdio.h>
#include<stdlib.h>
struct Node {
    int data;
    struct Node* next;
};
struct Node* head; // global
void Insert(int data); // insert an integer at end of list
void Print(); // print all elements in the list
void Delete(int n) // Delete node at position n
int main()
{
```

The user is asked to enter a position and the program will delete the node at this particular position.



```
int main()
{
    head = NULL; // empty list
    Insert(2);
    Insert(4);
    Insert(6);
    Insert(5); //List : 2,4,6,5
    Print();
    int n;
    printf("Enter a position\n");
    scanf("%d",&n);
    Delete(n);
    Print();
}
```

```

// Deletes node at position n
void Delete(int n)
{
    struct Node* temp1 = head;
    if(n ==1){
        head = temp1->next; //head now points to second node.
        free(temp1);
        return;
    }
    int i;
    for(i = 0;i<n-2;i++)
        temp1 = temp1->next;
    // temp1 points to (n-1)th Node
    struct Node* temp2 = temp1->next; // nth Node
    temp1->next = temp2->next; // (n+1)th Node
    free(temp2); //delete temp2;
}

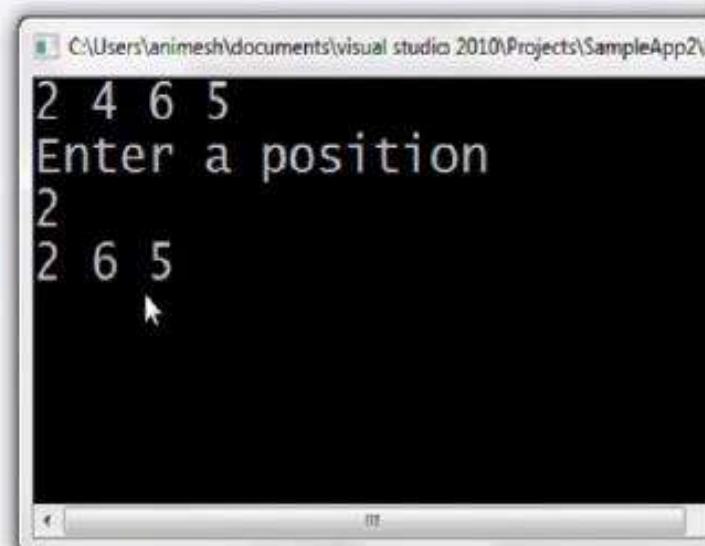
```

In the first case, we handle the case when there is a node before the node we want to delete.

Create a temporary variable **temp1** and point this to head.

Create a temporary variable **temp2** that points to the nth node.

```
int main()
{
    head = NULL; // empty list
    Insert(2);
    Insert(4);
    Insert(6);
    Insert(5); //List : 2,4,6,5
    Print();
    int n;
    printf("Enter a position\n");
    scanf("%d",&n);
    Delete(n);
    Print();
}
```

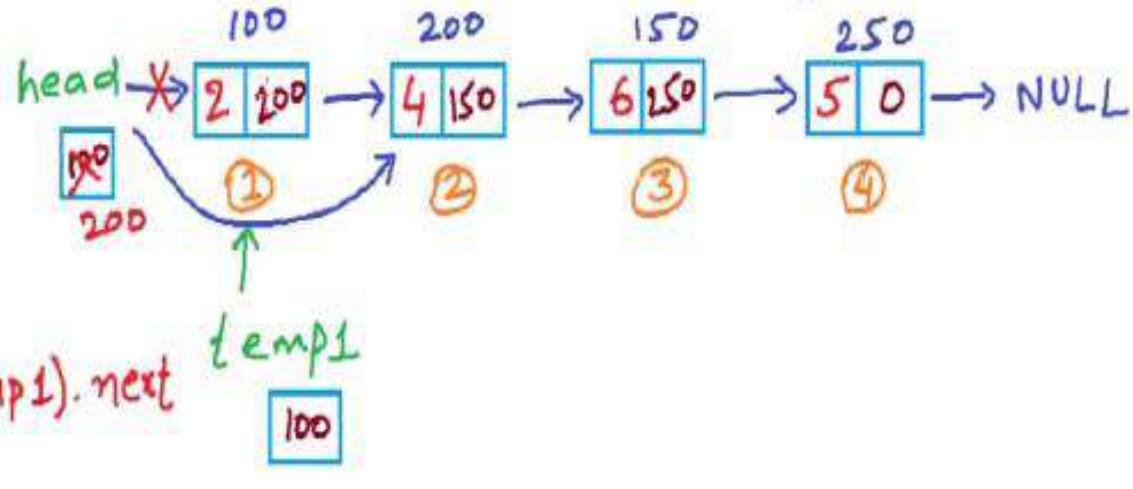


# Example

Linked List: Delete a node at  $n^{\text{th}}$  position

```
void Delete(int n)
{
    struct Node* temp1 = head;
    if(n == 1){
        head = temp1->next;
        free(temp1);
        return;
    }
    int i;
    for(i = 0; i < n-2; i++)
        temp1 = temp1->next;

    struct Node* temp2 = temp1->next;
    temp1->next = temp2->next;
    free(temp2);
}
```

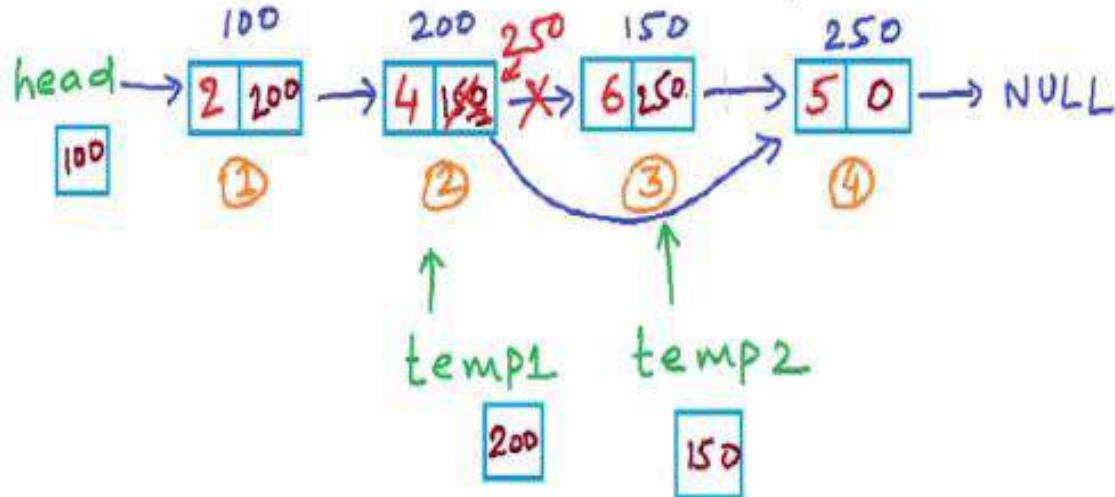


temp1 points to the first node (first position).

# Example

Linked List: Delete a node at  $n^{th}$  position

```
void Delete(int n)
{
    struct Node* temp1 = head;
    if(n == 1){
        head = temp1->next;
        free(temp1);
        return;
    }
    int i;
    for(i = 0; i < n-2; i++)
        temp1 = temp1->next;
    struct Node* temp2 = temp1->next;
    temp1->next = temp2->next;
    free(temp2);
}
```

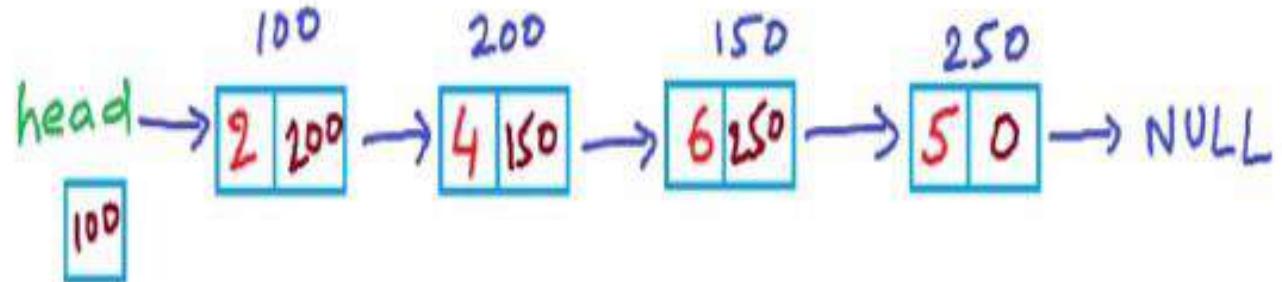


In the final step, **temp1** points to the second node and **temp2** points to the third node.

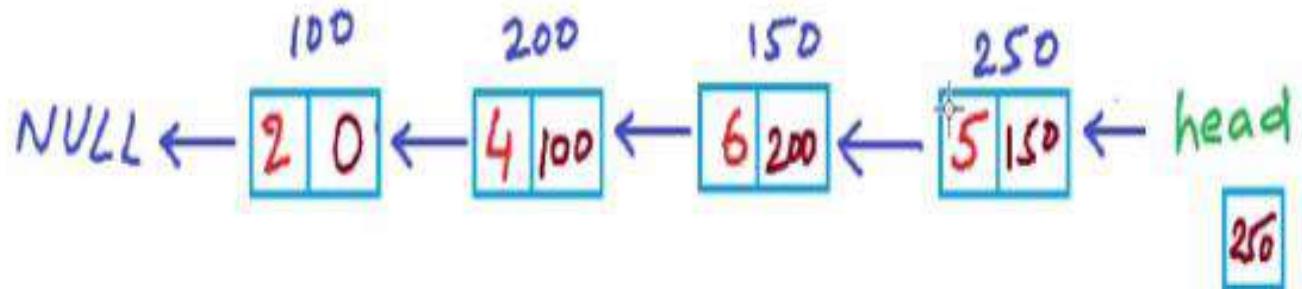
At the end, the address stored by **temp1**, which is 150, changes to **temp2.next**, which is 250.

Reverse a linked list using  
iterative method

Input:



Output:



Links should be changed. Head node should point to the node at address 250. For the first node, we cut the link from head and build a link to NULL.

Two solutions: iterative approach and recursion.

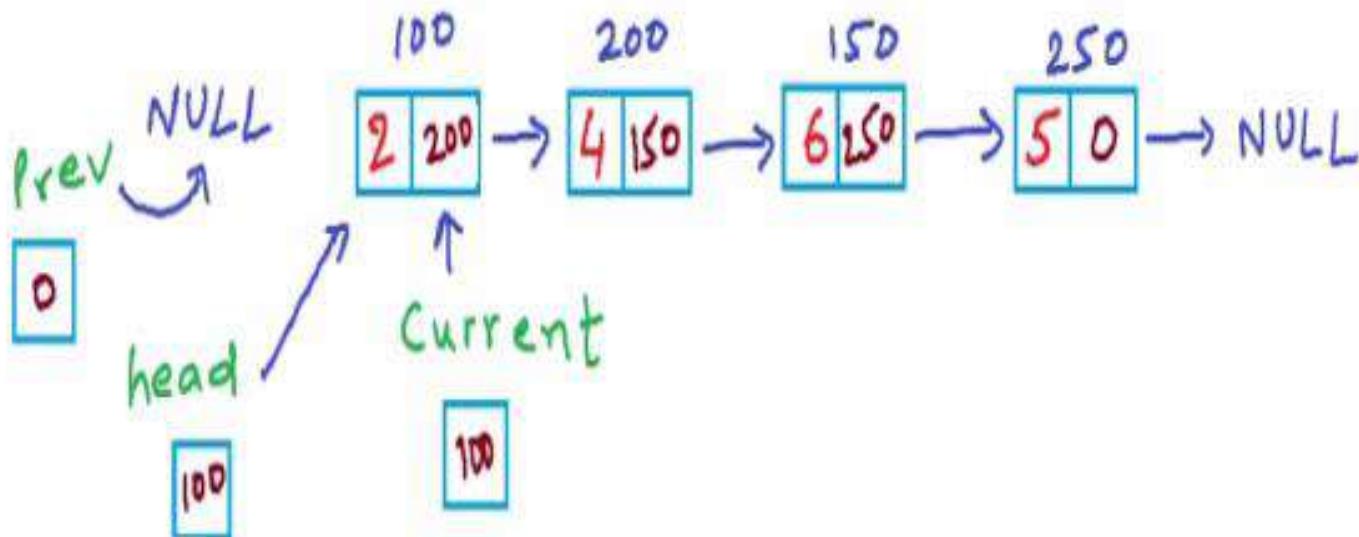
In the iterative solution, we write a loop that traverses each node and make it point to the preceding node instead of the next node.

```

struct Node
{
    int data;
    struct Node* next;
};

struct Node* head;
void Reverse()
{
    struct Node *current, *prev, *next;
    current = head;
    prev = NULL;
    while(current != NULL)
    {
        ...
    }
}

```



To traverse a list, create a temporary node pointing to the currently traversed node, called **current**, first point it to the head node and then run a loop as shown in the code.

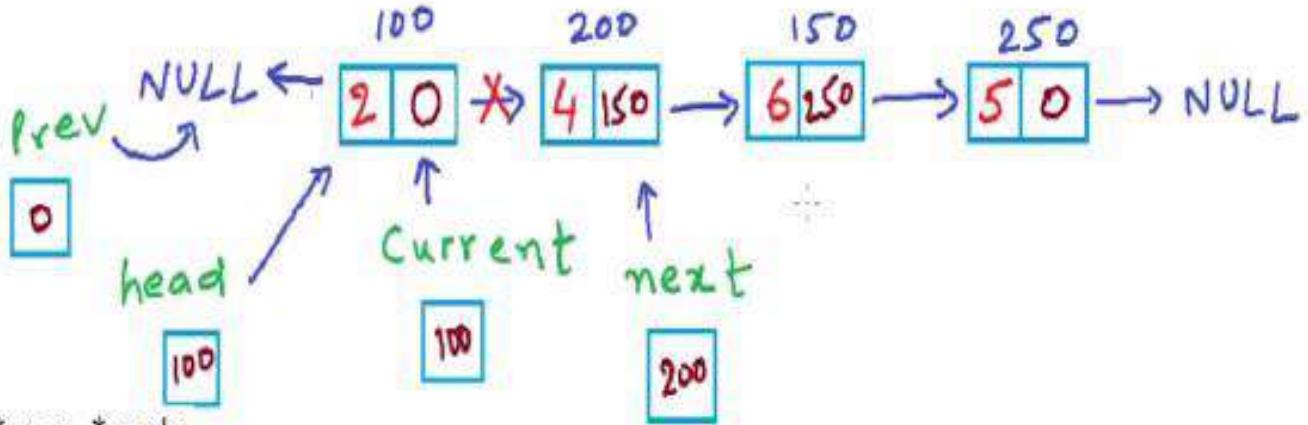
We will have to keep track of the previous node of each node while traversing. Call this temporary node **prev**.

```

struct Node
{
    int data;
    struct Node* next;
};

struct Node* head;
void Reverse()
{
    struct Node *current, *prev, *next;
    current = head;
    prev = NULL;
    while(current != NULL)
    {
        next = current->next;
        current->next = prev;
}

```



And at each step of the traversal, we need to store the address of the old next node of the current node using a temporary variable, otherwise we loose this link. So, call this temporary variable **next**.

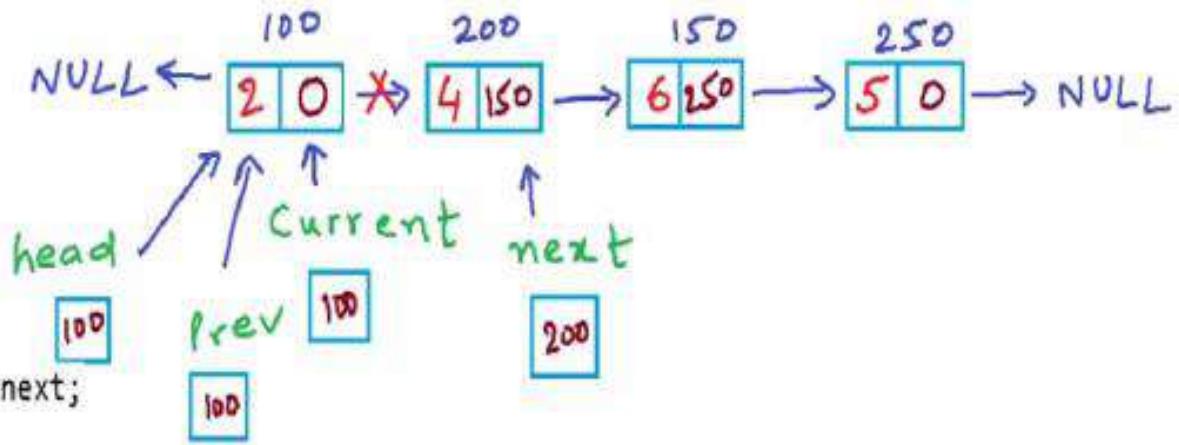
Initially, **prev** points to **NULL**, **current** points to the first node. After the first iteration of the while loop, **next** points to the second node and the address field of first node stores **NULL** after using the dereferencing **current->next=prev**.

```

struct Node
{
    int data;
    struct Node* next;
};

struct Node* head;
void Reverse()
{
    struct Node *current, *prev, *next;
    current = head;
    prev = NULL;
    while(current != NULL)
    {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
}

```



In the last two lines of the while loop, we update where `prev` and `current` are pointing to complete the traversal of one node.

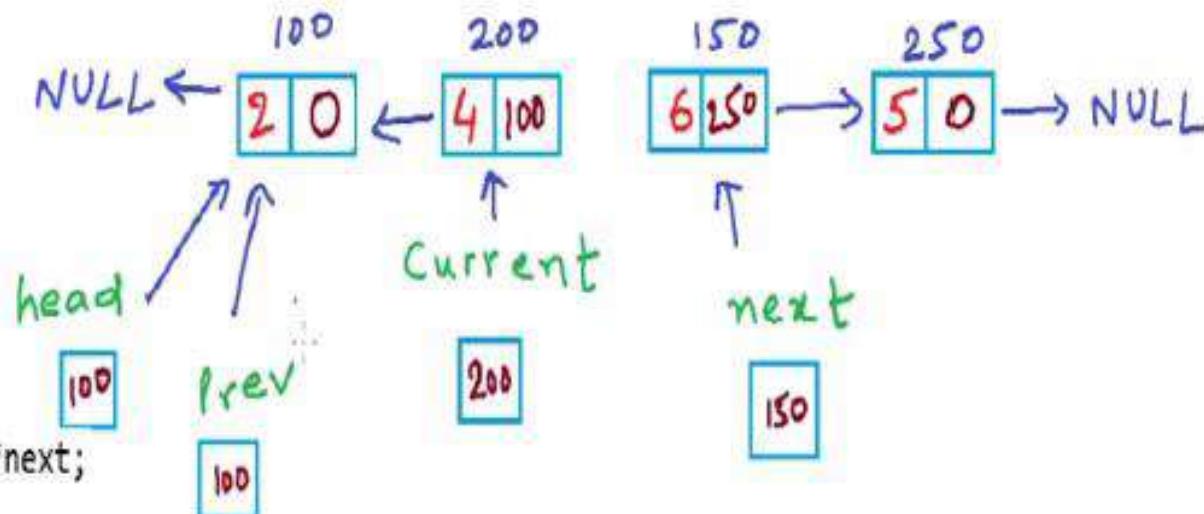
Note that the `next` in `current->next` and the local variable `next` are different variables!!!

```

struct Node
{
    int data;
    struct Node* next;
};

struct Node* head;
void Reverse()
{
    struct Node *current,*prev,*next;
    current = head;
    prev = NULL;
    while(current != NULL)
    {
        next = current->next;
        current->next = prev;
        prev = current; ✓
        current = next; ✓
    }
}

```

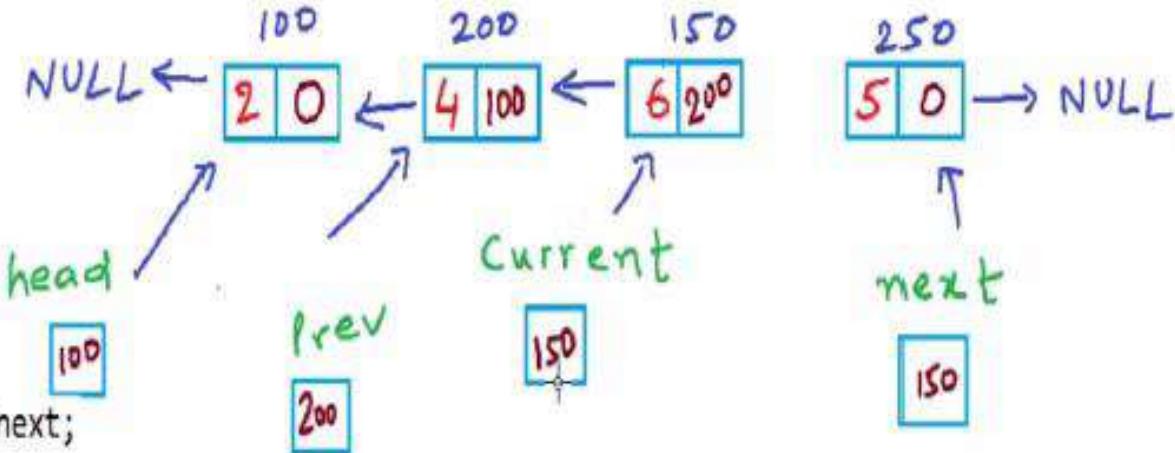


```

struct Node
{
    int data;
    struct Node* next;
};

struct Node* head;
void Reverse()
{
    struct Node *current, *prev, *next;
    current = head;
    prev = NULL;
    while(current != NULL)
    {
        next = current->next;
        current->next = prev;
        prev = current; ✓
        current = next; ✓
    }
}

```

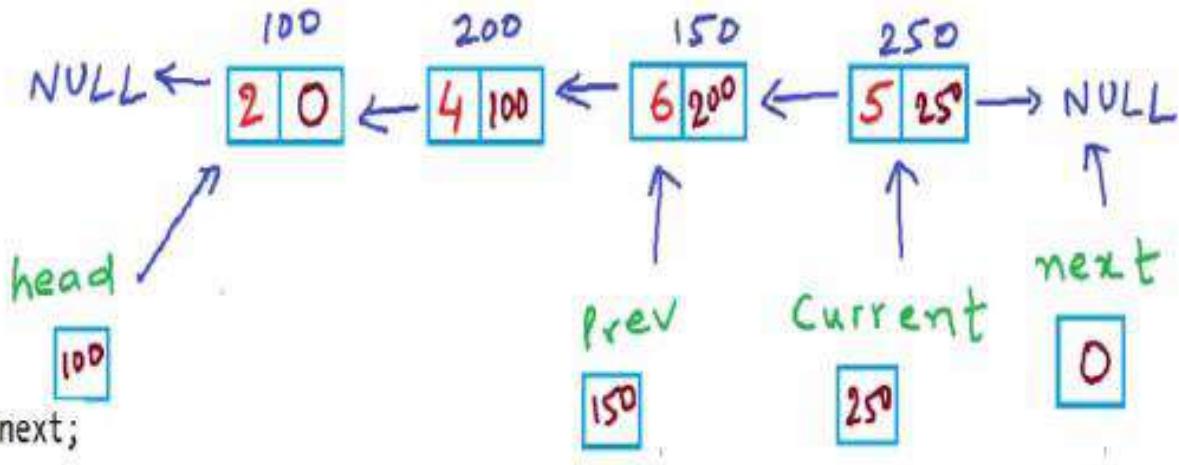


```

struct Node
{
    int data;
    struct Node* next;
};

struct Node* head;
void Reverse()
{
    struct Node *current,*prev,*next;
    current = head;
    prev = NULL;
    while(current != NULL)
    {
        next = current->next;
        current->next = prev;
        prev = current; ✓
        current = next; ✓
    }
}

```

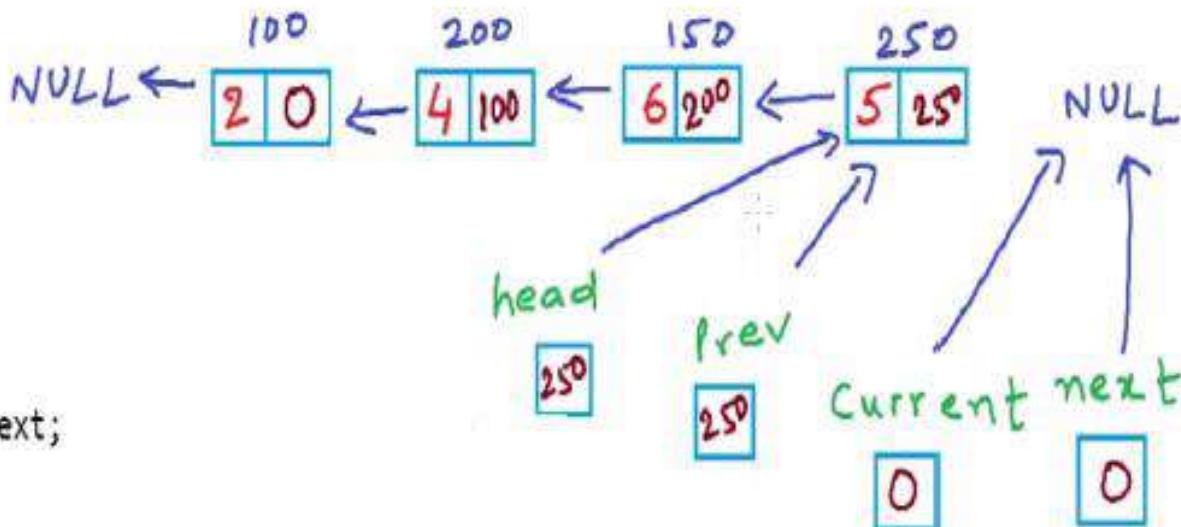


```

struct Node
{
    int data;
    struct Node* next;
};

struct Node* head;
void Reverse()
{
    struct Node *current, *prev, *next;
    current = head;
    prev = NULL;
    while(current != NULL)
    {
        next = current->next;
        current->next = prev;
        prev = current; ✓
        current = next; ✓
    }
    head = prev;
}

```



```
// Reverse a nexted list
#include<stdio.h>
#include<stdlib.h>
struct Node {
    int data;
    struct Node* next;
};
struct Node* Reverse(struct Node* head) {
    struct Node *current,*prev,*next;
    current = head;
    prev = NULL;
    while(current != NULL)
    {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    head = prev;
    return head;
}
```

The **reverse function** takes the address of the head node as argument and returns the address of the head node.

```
int main()
{
    struct Node* head =NULL; // local variable
    head = Insert(head,2); // Insert: struct Node* Insert
    head = Insert(head,4);
    head = Insert(head,6);
    head = Insert(head,8);
    Print(head);
    head = Reverse(head);
    Print(head);
}
```



In the main method, head is defined as a local variable. The insert function takes two **arguments**: the **address of the head node** and the **data to be inserted**.

The **insert function** returns the address of the head node. **Print** function prints the elements in the list.

Print elements of a linked list in forward and  
reverse order using recursion

```
// Print Linked List using Recursion
#include<stdio.h>
#include<stdlib.h>
struct Node {
    int data;
    struct Node* next;
};
void Print(struct Node* p)
{
    if(p == NULL) return; // Exit condition
    printf("%d ",p->data); // First print the value in the node
    Print(p->next); // Recursive call
```

The print function takes the address of a node, so the argument is of type pointer.

For now forget about how we input the linked list, assume it is already entered. Printf will print the value at the node p.

Then we make a recursive call to the print function passing the address of the next node without forgetting the exit condition for the recursion.

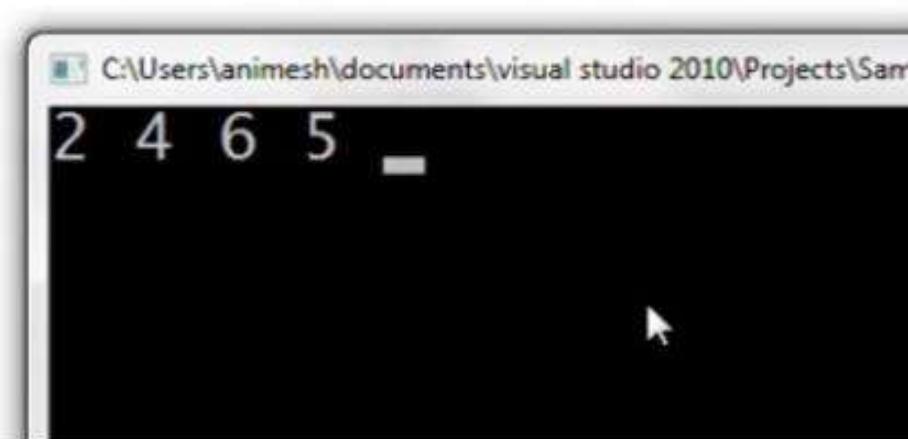
In the [insert function](#), the insert function returns the current address of the head node after insertion. (Note that, head is a local variable in the main method.)

Here, the insert function [inserts a node at the end of the list](#).

```
struct Node* Insert(Node* head,int data) {  
    Node *temp = (struct Node*)malloc(sizeof(struct Node));  
    temp->data = data;  
    temp->next = NULL;  
    if(head == NULL) head = temp;  
    else {  
        Node* temp1 = head;  
        while(temp1->next != NULL) temp1 = temp1->next;  
        temp1->next = temp;  
    }  
    return| head;  
}  
  
int main()  
{  
    struct Node* head = NULL; // local variable  
    head = Insert(head,2);  
}
```

By using the recursive print function, we were able to print the linked list in forward order. See next slide for steps.

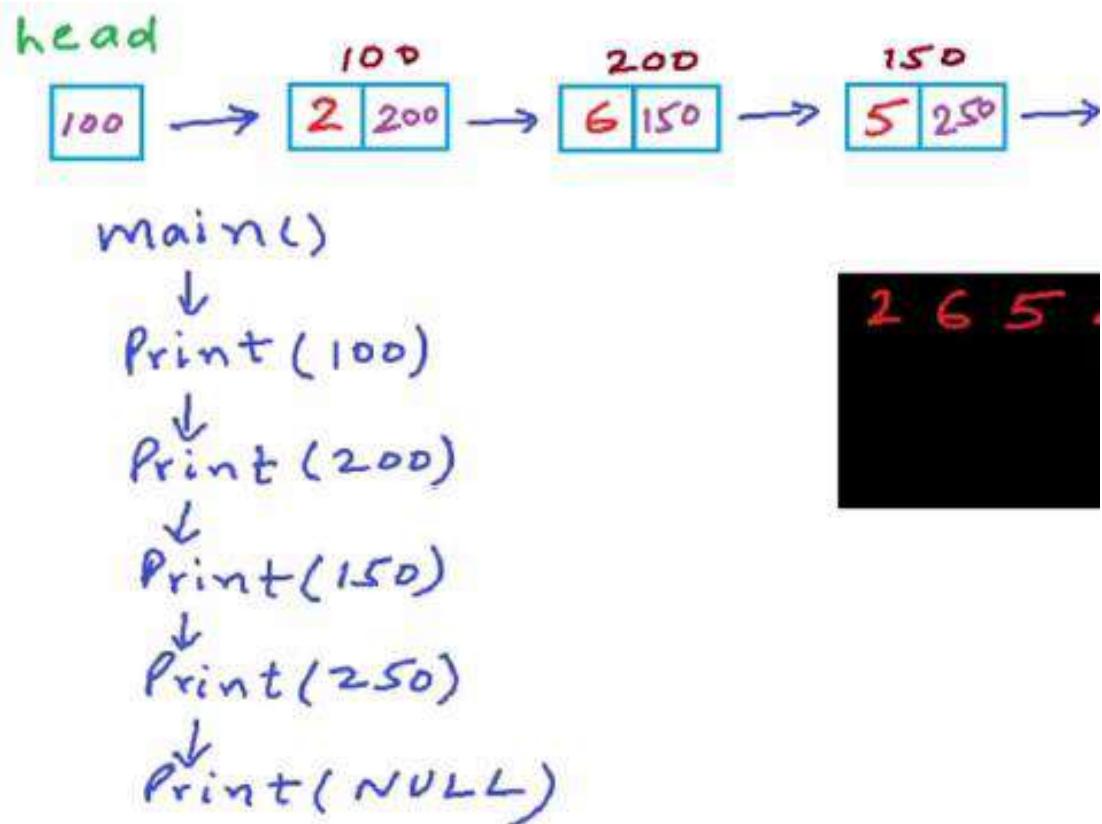
```
int main()
{
    struct Node* head = NULL; // local variable
    head = Insert(head,2);
    head = Insert(head,4);
    head = Insert(head,6);
    head = Insert(head,5);
    Print(head);
}
```



Each time print is visited, it prints the data stored in the address input as an argument.

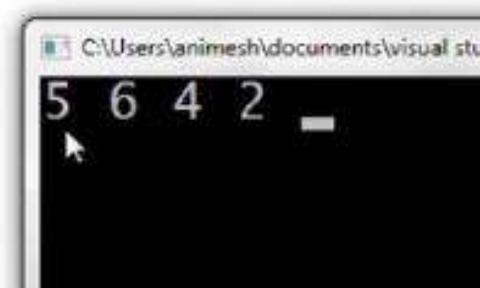
The arrows showing the steps of the recursion is called a **recursion tree**.

```
struct Node {  
    int data;  
    struct Node* next;  
};  
  
void Print(struct Node* p)  
{  
    ✓ if(p == NULL)  
    {  
        printf("\n");  
        return;  
    }  
    ✓ printf("%d ", p->data);  
    ✓ Print(p->next);  
}
```



```
// Print Linked List using Recursion
#include<stdio.h>
#include<stdlib.h>
struct Node {
    int data;
    struct Node* next;
};
void Print(struct Node* p)
{
    if(p == NULL) return; // Exit condition
    Print(p->next); // Recursive call
    printf("%d ",p->data); // First print the value in the node
}
```

```
int main()
{
    struct Node* head = NULL; // local variable
    head = Insert(head,2);
    head = Insert(head,4);
    head = Insert(head,6);
    head = Insert(head,5);
    Print(head);
}
```

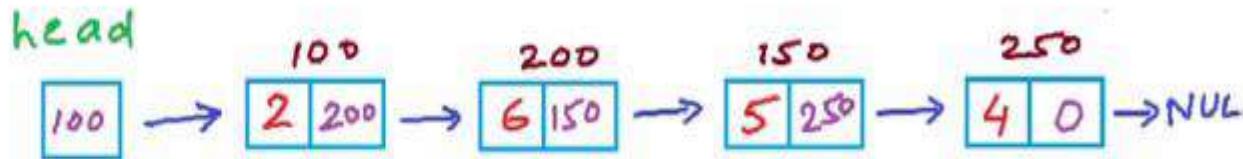


```

struct Node {
    int data;
    struct Node* next;
};

void ReversePrint(struct Node* p)
{
    ✓ if(p == NULL)
    {
        return;
    }
    ReversePrint(p->next);
    ✓ printf("%d ",p->data);
}

```



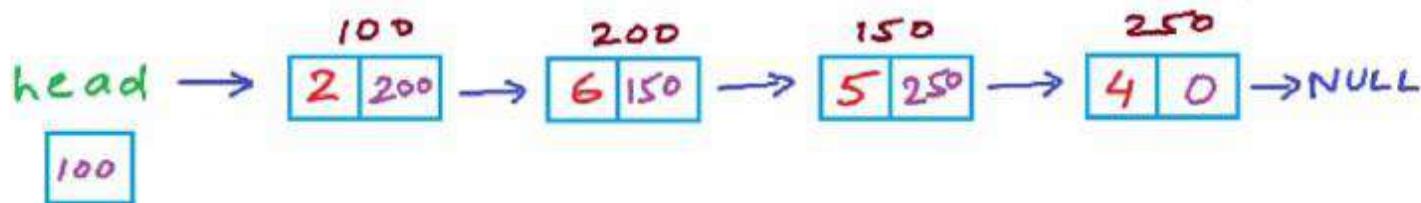
✓ main()

4 5 6 2

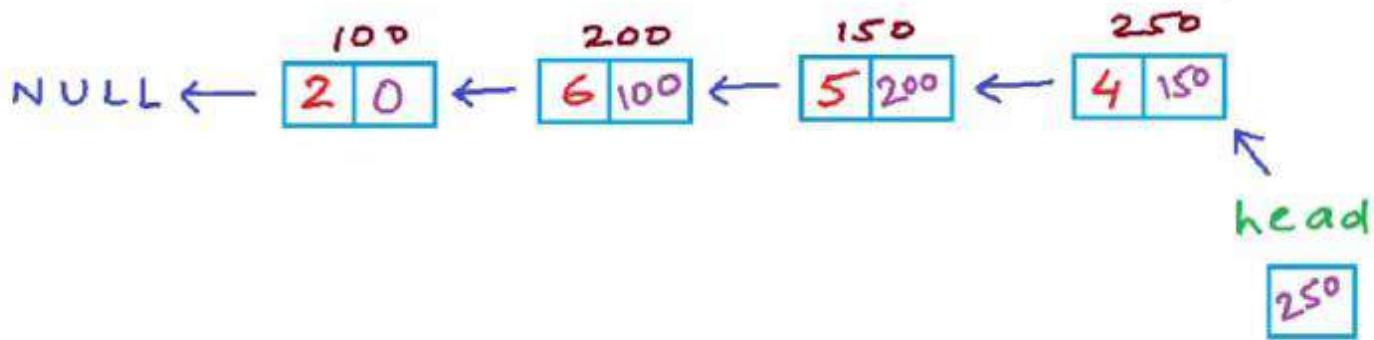
Reverse a linked list using  
recursion

Reverse a linked list using recursion

Input:



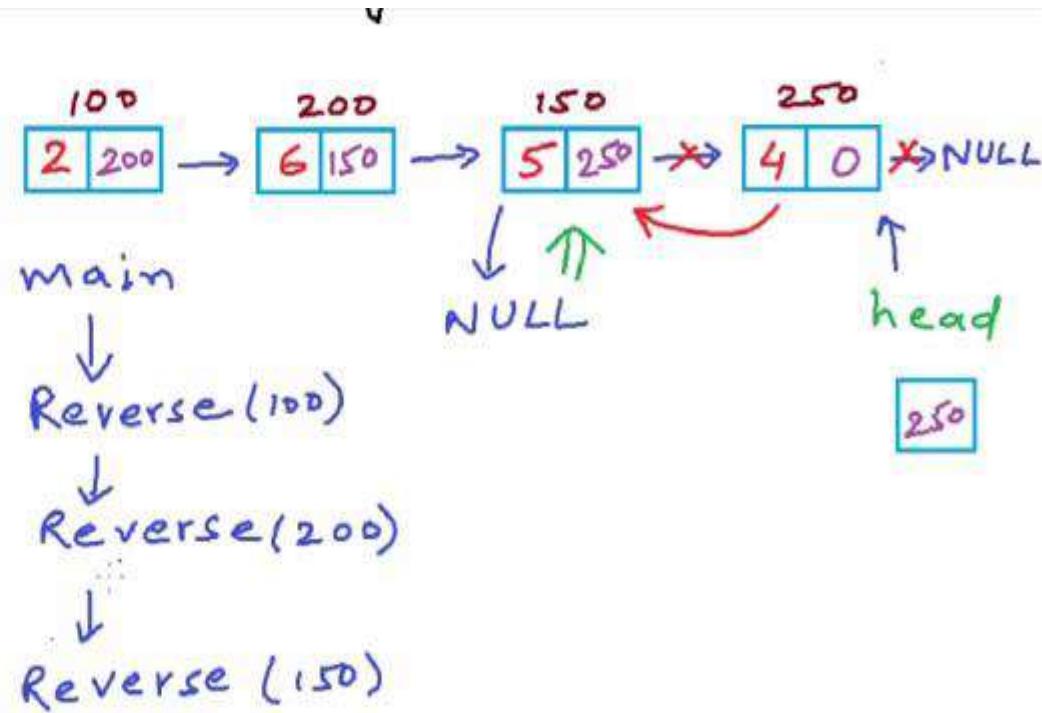
Output:



The address passed to the reverse function is the address of the first node. As soon as we reach the last node, the program modifies the head pointer to point to the fourth node.

Below, we see the links after **Reverse(250)** and **Reverse(150)** are finished.

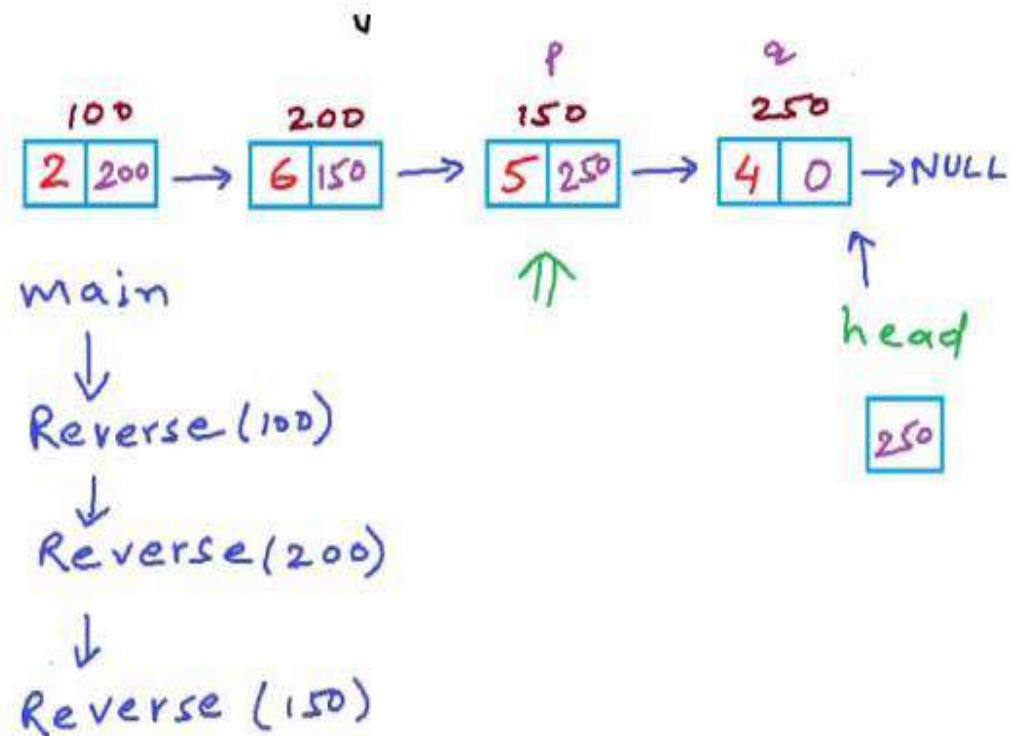
```
struct Node {  
    int data;  
    struct Node* next;  
};  
  
struct Node* head;  
void Reverse(struct Node* p)  
{  
    ✓ if(p->next == NULL)  
    {  
        head = p;  
        return;  
    }  
    Reverse(p->next);  
    // Statement  
}
```



The last three lines will execute after the recursive calls are finished and we are traversing the list in backwards direction.

When **Reverse(150)** is executed, p would be 150 and q would be p.next. See next slide.

```
struct Node {  
    int data;  
    struct Node* next;  
};  
  
struct Node* head;  
void Reverse(struct Node* p)  
{  
    ✓ if(p->next == NULL)  
    {  
        head = p;  
        return;  
    }  
    Reverse(p->next);  
    ✓ struct Node* q = p->next;  
    q->next = p;  
    p->next = NULL;  
}
```



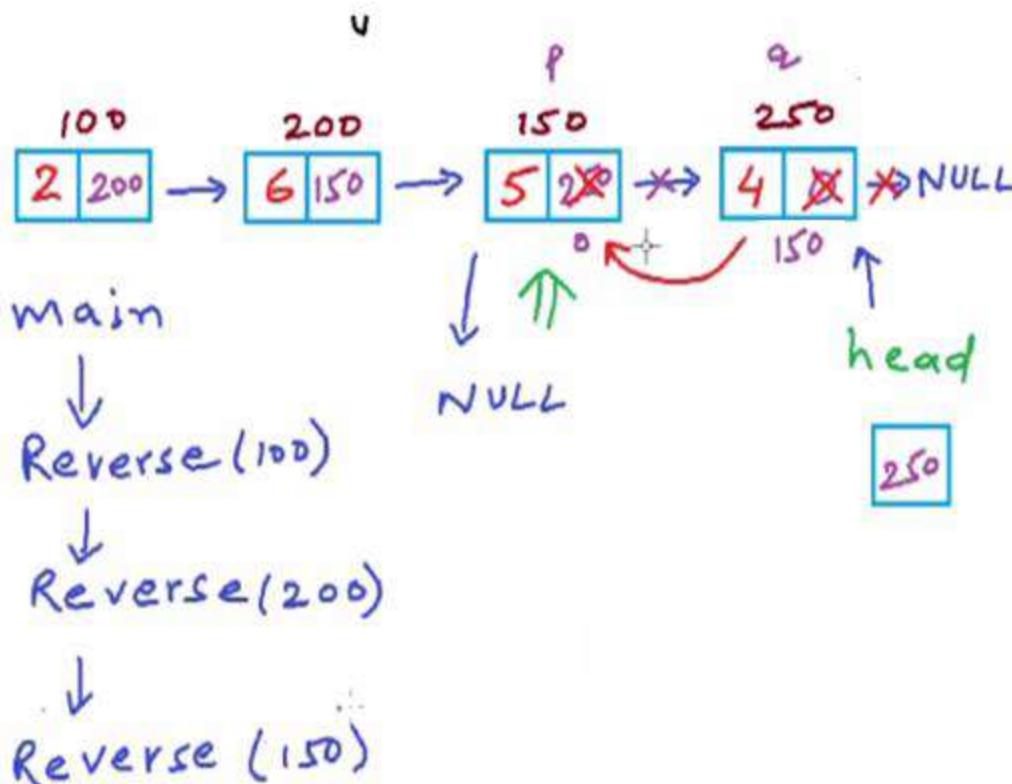
```

struct Node {
    int data;
    struct Node* next;
};

struct Node* head;
void Reverse(struct Node* p)
{
    ✓ if(p->next == NULL)
    {
        head = p;
        return;
    }
    Reverse(p->next);
    ✓ struct Node* q = p->next;
    q->next = p;
    p->next = NULL;
}

```

*Exit Condition*



## **Question 1: Print the contents of a given linked list pointed with a list pointer (list \*)?**

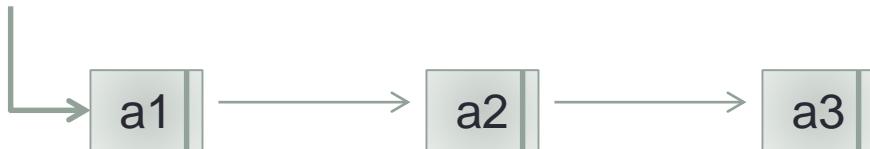
```
void print_items ( listnode * list )
{
    printf (" \n list contents: " );
    for(;list; list= list-> link)
        printf (" %d ", list->data );
    printf (" \n ");
}
```

## **Question 2: Count the number of items in a given linked list pointed with a list pointer (list \*)?**

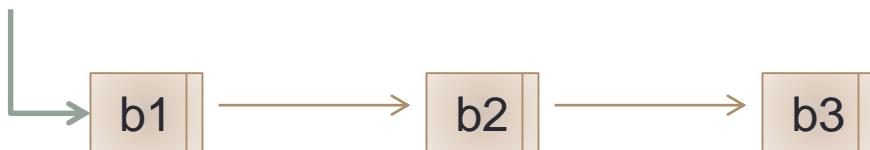
```
int count_items ( listnode * list )
{
    int n=0;
    for(;list; list= list-> link, n++);
    return n;
}
```

**Question 3: list1 and list2 are two pointers pointing to two separate linked list. Append list2 to the end of list1.**

list1



list2



list1



```
void appendlists ( listnode * list1, listnode * list2 )
{
    list * p;
    if( list1 ) {
        for(p=list; p->link; p = p -> link);
        p->link = list2;
    }
    else
        list1=list2;
    list2=NULL;
}
```

# **BBM 201**

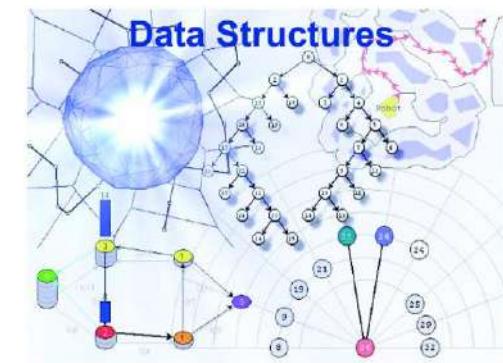
# **DATA STRUCTURES**

---

**Lecture 10:**  
Implementation of Linked Lists  
(Stacks, Queue, Hashtable)



**2018-2019 Fall**



# **Linked list implementation of stacks**

The cost of insert and delete at the beginning of a linked list is constant, that is  $O(1)$ .

So, in the linked list implementation of stack, we insert and delete a node at the beginning so that time complexity of Push and Pop is  $O(1)$ .

Insert/delete

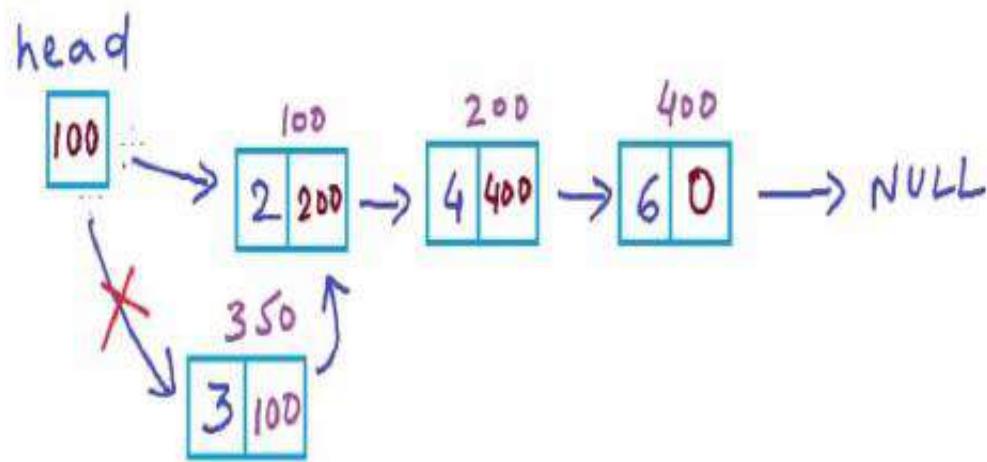
(1) at end X

$\downarrow$   
 $O(n)$

(2) at beginning

$\downarrow$

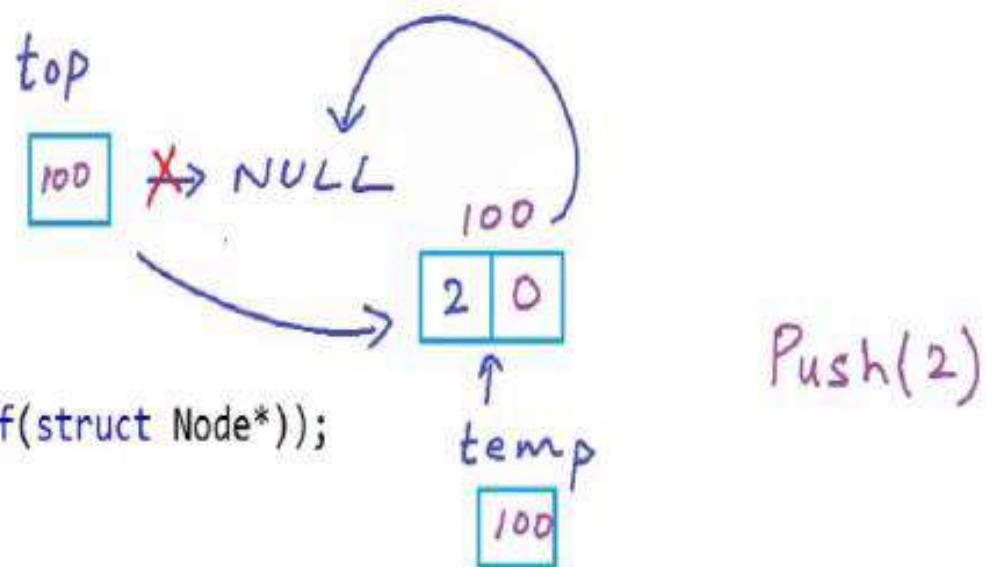
$O(1)$



Instead of head, we use variable name **top**. When top is NULL, the stack is empty.

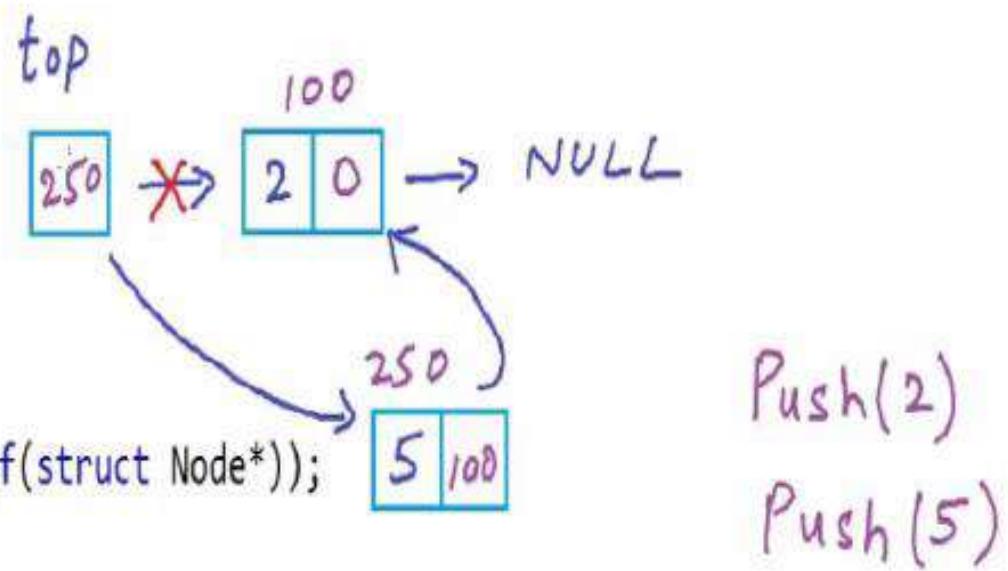
With dereferencing **temp->data=x** and **temp->link = top**, we fill the fields of the new node, called **temp**. Finally, with **top=temp**, top points to the new node.

```
struct Node {  
    int data;  
    struct Node* link;  
};  
struct Node* top = NULL;  
void Push(int x) {  
    struct Node* temp =  
        (struct Node*)malloc(sizeof(struct Node*));  
    temp->data = x;  
    temp->link = top;  
    top = temp;  
}
```



Stack data structure uses the memory efficiently by using push when something is needed and pop when not needed in an array or linked list.

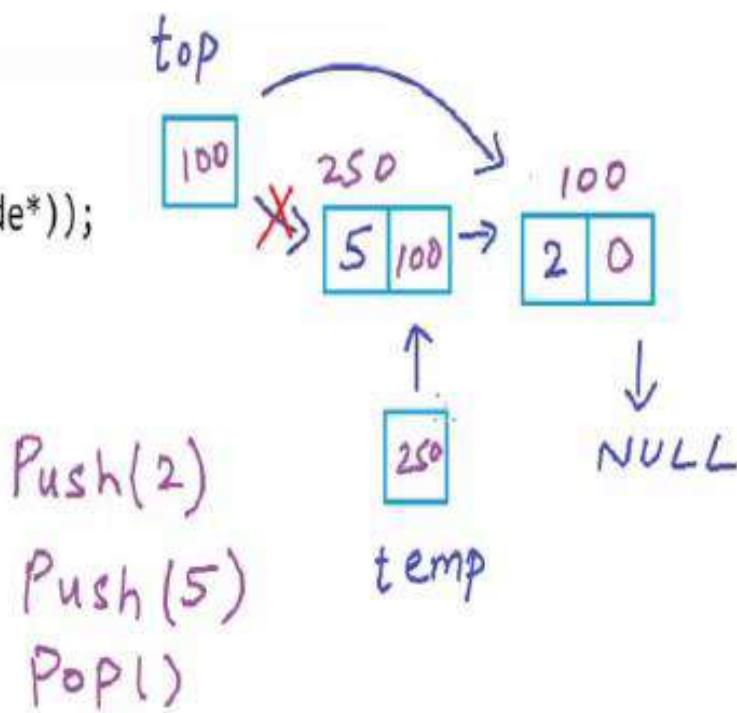
```
struct Node {  
    int data;  
    struct Node* link;  
};  
struct Node* top = NULL;  
void Push(int x) {  
    struct Node* temp =  
        (struct Node*)malloc(sizeof(struct Node*));  
    temp->data = x;  
    temp->link = top;  
    top = temp;  
}
```



```

struct Node* top = NULL;
void Push(int x) {
    struct Node* temp =
        (struct Node*)malloc(sizeof(struct Node*));
    temp->data = x;
    temp->link = top;
    top = temp;
}
void Pop() {
    struct Node *temp;
    if(top == NULL) return;
    temp = top;
    top = top->link;
    free(temp);
}

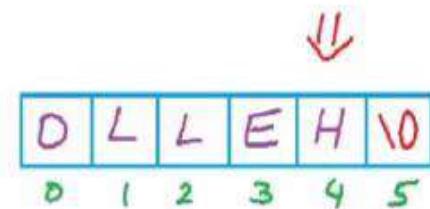
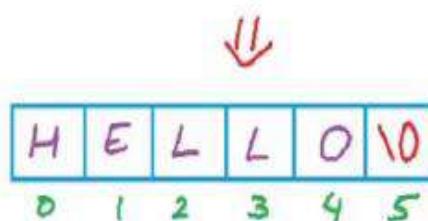
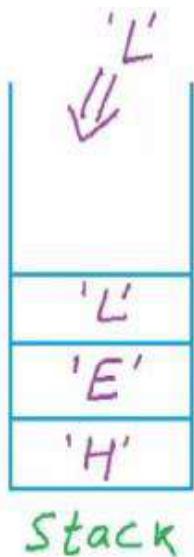
```



# **Reverse a string using stack**

We can create a stack of characters by traversing the string from left to right and using the push operation.

Then, we can pop the elements from the stack and overwrite all the positions in the string.



We pass the string and the length of the string to the [Reverse function](#).

```
// STRING REVERSAL USING STACK
#include<iostream>
using namespace std;

void Reverse(char C[], int n)
{
}

int main() {
    char C[51];
    printf("Enter a string: ");
    gets(C);
    Reverse(C, strlen(C));
    printf("Output = %s", C);
}
```

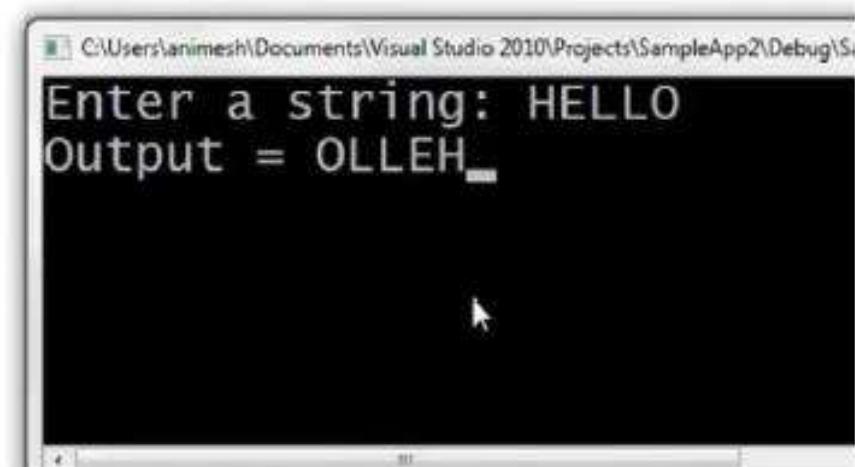
In C++, we can use these operations available from the standard temporary library called stack.

```
// string reversal using stack
#include<iostream>
using namespace std;
class Stack
{
private:
    char A[101];
    int top;|
public:
    void Push(int x);
    void Pop();
    int Top();
    bool IsEmpty();
};
```

As we traverse the string from left to right, we use the push function. In the second loop, we use the top and pop functions for each character in the stack to determine the string C. See next slide for output.

```
#include<iostream>
#include<stack> // stack from standard template library (STL)
using namespace std;
void Reverse(char *C,int n)
{
    stack<char> S;
    //loop for push
    for(int i=0;i<n;i++){
        S.push(C[i]);
    }
    //loop for pop
    for(int i =0;i<n;i++){
        C[i] = S.top(); //overwrite the character at index i.
        S.pop(); // perform pop.
    }
}
```

```
#include<stack> // stack from standard template library (STL)
using namespace std;
void Reverse(char *C,int n)
{
    stack<char> S;
    //loop for push
    for(int i=0;i<n;i++){
        S.push(C[i]);
    }
    //loop for pop
    for(int i =0;i<n;i++){
        C[i] = S.top(); //overwrite the character at index i.
        S.pop(); // perform pop.
    }
}
int main() {
    char C[51];
    printf("Enter a string: ");
    gets(C);
    Reverse(C,strlen(C));
    printf("Output = %s",C);
}
```



## Problem: Reverse a string

|   |   |   |   |   |    |
|---|---|---|---|---|----|
| H | E | L | L | O | \0 |
| 0 | 1 | 2 | 3 | 4 | 5  |

```
void Reverse(char *C,int n)
```

```
{
```

```
stack<char> S;
```

```
//loop for push
```

```
O(n) { for(int i=0;i<n;i++){ S.push(C[i]); }
```

```
//loop for pop
```

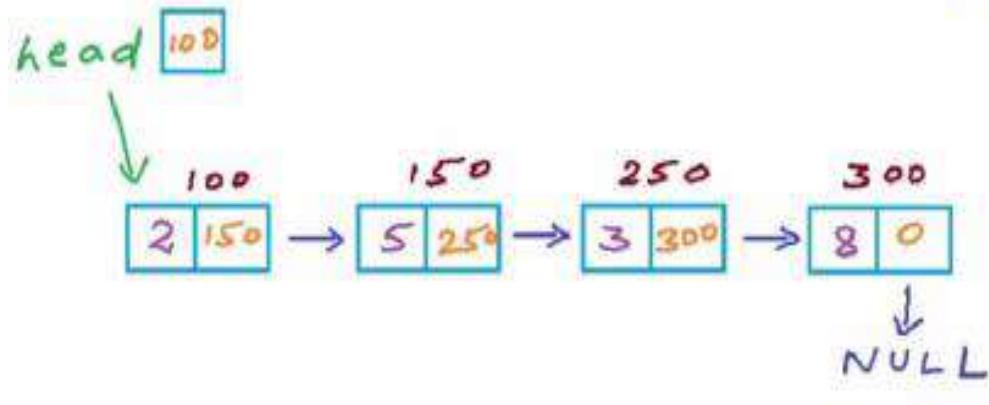
```
O(n) { for(int i =0;i<n;i++){ C[i] = S.top(); } S.pop(); }
```

```
}
```

- All operations on stack take constant time,  $O(1)$ .
- Thus, for each loop, time cost is  $O(n)$ .
- Space complexity is also  $O(n)$ .

**Reverse a linked list using  
stack**

# Time complexity of reversing a linked list



Iterative Solution

Time -  $O(n)$

Space -  $O(1)$

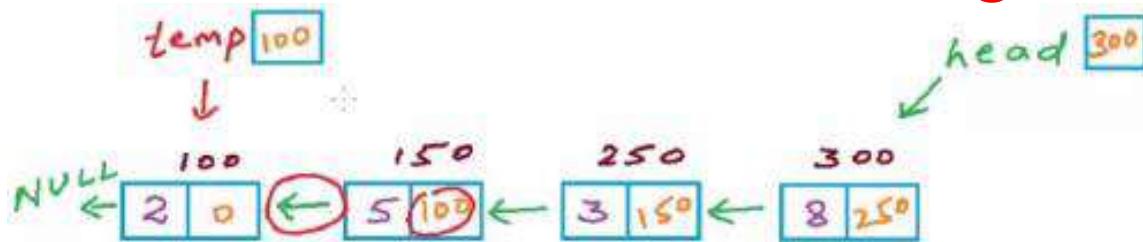
Recursive Solution

(Implicit Stack)

Time -  $O(n)$

Space -  $O(n)$

# Reverse a linked list using stack



```
Node *temp = S.top();
head = temp;
S.pop();
while(!S.empty()){
    temp->next = S.top();
    S.pop();
    temp = temp->next;
}
temp->next = NULL;
```

```
stack<struct Node*> S;
struct Node {
    int data;
    Node* next;
};
```

The values we are pushing into the stack are the pointers to the node. Initially, `S.top()` returns `300`. We store head as this address, so `head` stores `300`.

In the loop, while stack is not empty, set `temp->next` to the address at the top of the stack, which is `250`.

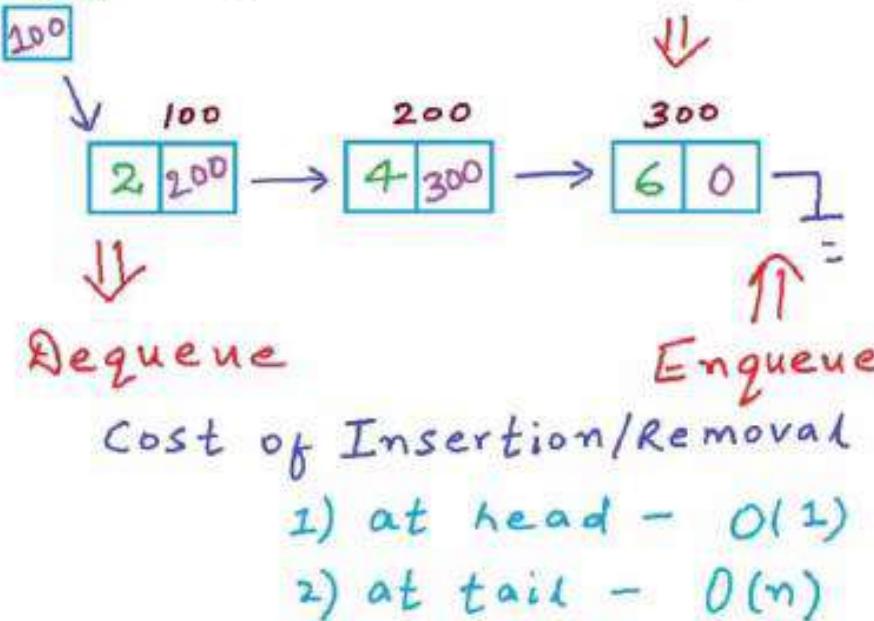
And finally, `temp` points to the node at address `250`.

The last line is needed so that the end of the reversed linked list points to `NULL` and not to `150`.

# **Linked list implementation of queue**

head

## Queue - Linked List implementation



### Operations

- (1) Enqueue( $x$ )
- (2) Dequeue()
- (3) front()
- (4) IsEmpty()

Constant time  
or  
 $O(1)$

In this case above, Dequeue will take constant time, but Enqueue will take  $O(n)$  time.

As before, the traversal of the linked list to learn the address of the last node takes  $O(n)$  time.

**Solution:** Keep a pointer called rear that stores the address of the node at the rear position.

```

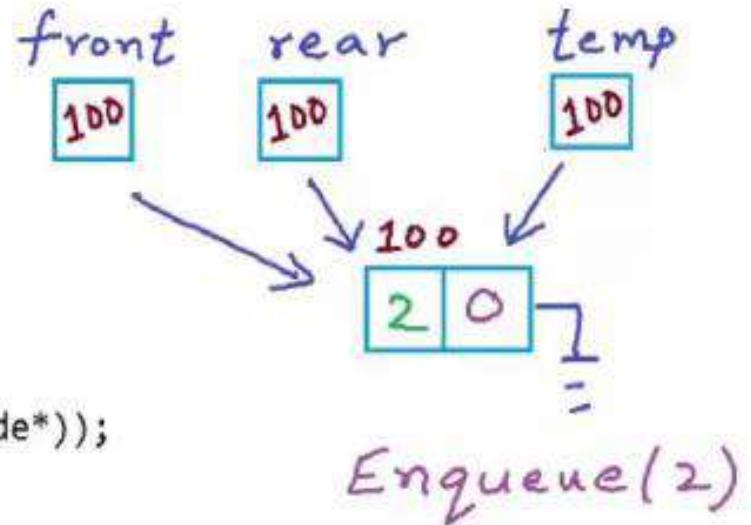
struct Node {
    int data;
    struct Node* next;
};

struct Node* front = NULL;
struct Node* rear = NULL;

void Enqueue(int x) {
    struct Node* temp =
        (struct Node*)malloc(sizeof(struct Node*));
    temp->data = x;
    temp->next = NULL;

    if(front == NULL && rear == NULL){
        front = rear = temp;
        return;
    }
    rear->next = temp;
    rear = temp;
}

```



Instead of declaring a pointer variable pointing to head, we **declare two pointer variables** one pointing to the front node and another pointing to the rear node.

For the **Enqueue**, we again use a pointer called `temp` to point to the new node created dynamically.

If the list is empty, both `front` and `rear` point to the new node.

```

struct Node {
    int data;
    struct Node* next;
};

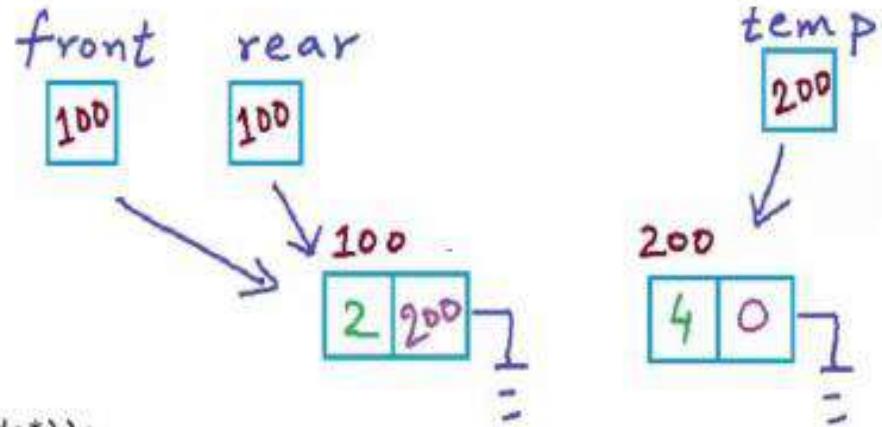
struct Node* front = NULL;
struct Node* rear = NULL;

void Enqueue(int x) {
    struct Node* temp =
        (struct Node*)malloc(sizeof(struct Node*));
    temp->data = x;
    temp->next = NULL;

    if(front == NULL && rear == NULL){
        front = rear = temp;
        return;
    }

    rear->next = temp;
    rear = temp;
}

```



If the list is not empty, first, set the address part of the current **rear node** to the address of the new node.

Then let the rear pointer point to the new node.

```

struct Node {
    int data;
    struct Node* next;
};

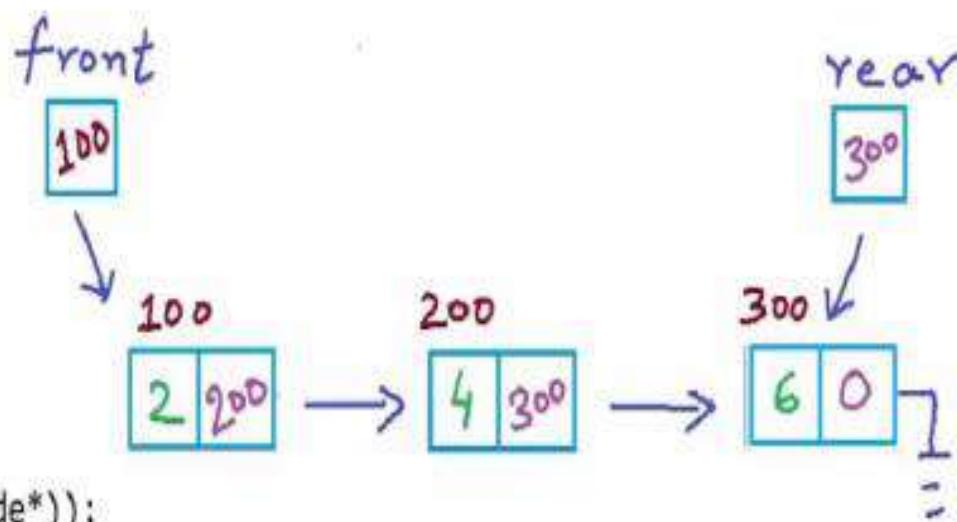
struct Node* front = NULL;
struct Node* rear = NULL;

void Enqueue(int x) {
    struct Node* temp =
        (struct Node*)malloc(sizeof(struct Node));
    temp->data = x;
    temp->next = NULL;

    if(front == NULL && rear == NULL){
        front = rear = temp;
        return;
    }

    rear->next = temp;
    rear = temp;
}

```



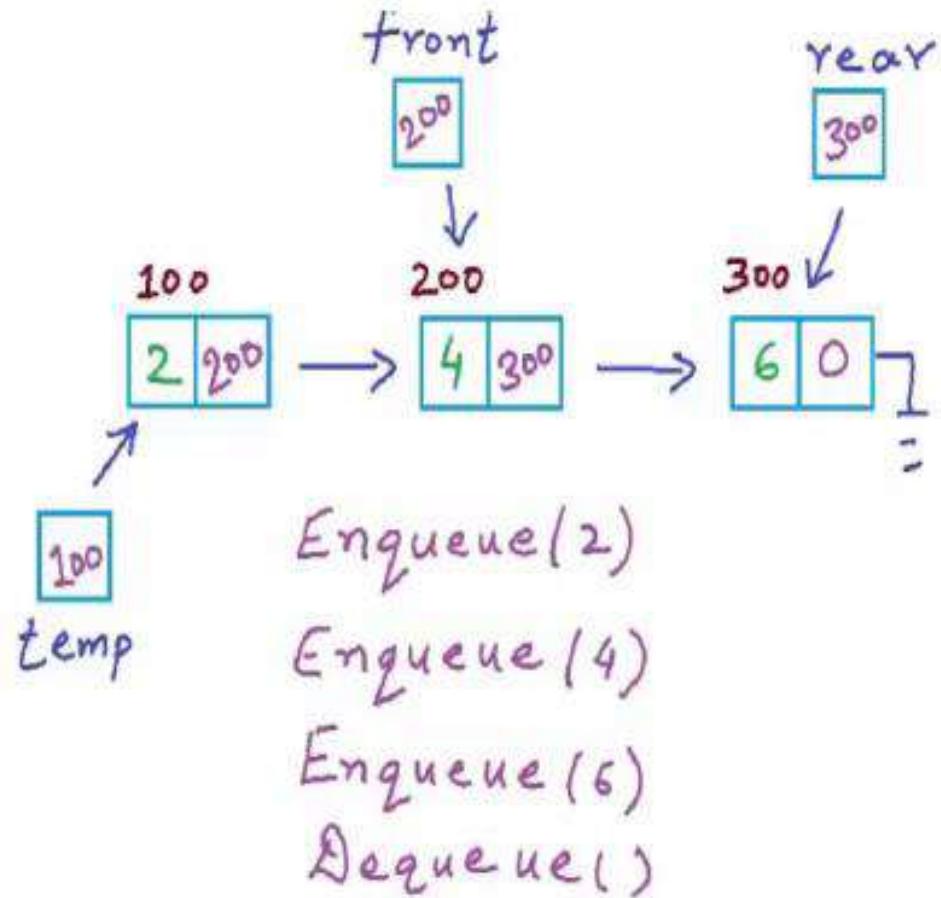
Enqueue(2)

Enqueue(4)

Enqueue(6)

# Implementation of Dequeue

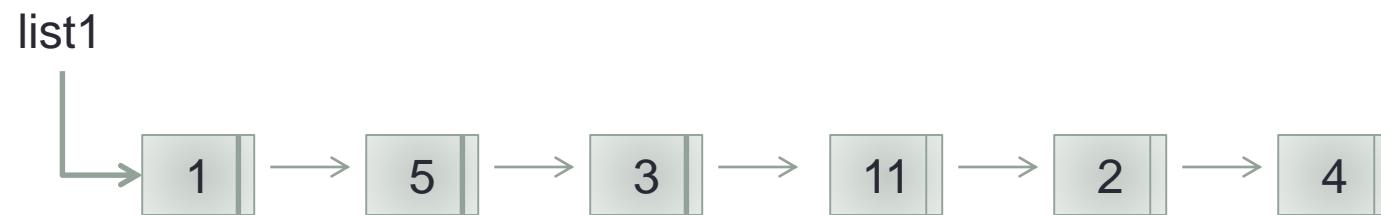
```
void Dequeue() {  
    struct Node* temp = front;  
    if(front == NULL) return;  
    if(front == rear) {  
        front = rear = NULL;  
    }  
    else {  
        front = front->next;  
    }  
    free(temp);  
}
```



# **Multiple Stack Implementation Using Linked List**

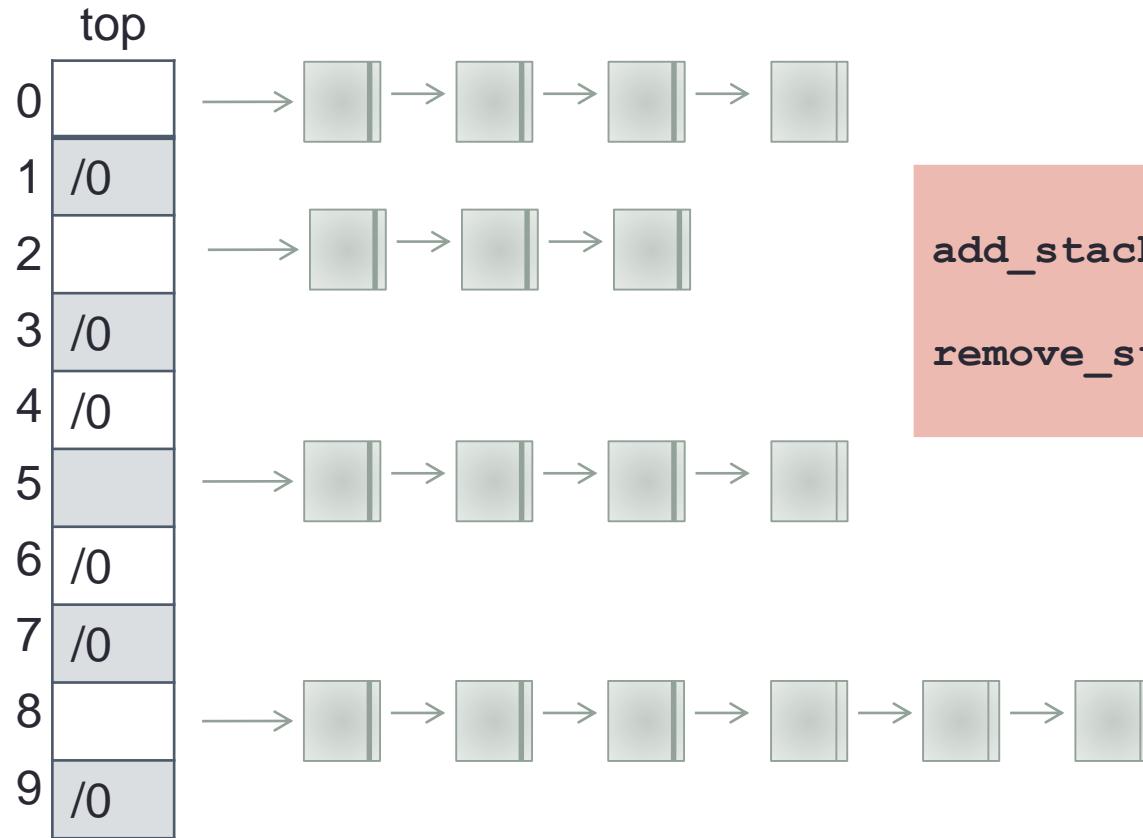
# Multiple Stack Implementation Using Linked List

A single stack:



# Multiple Stack Implementation Using Linked List

## Multiple stacks:



```
add_stack(&top[stack_i], x);  
  
remove_stack(&top[stack_i]);
```

# Linked List

```
#define MAX_STACK 10 // number of max stacks

typedef struct{
    int value;
    //other fields
}item;
typedef struct stack *stack_pointer;
typedef struct stack{
    item x;
    stack * next;
};

stack_pointer top [MAX_STACK];
```

# Linked List

```
#define ISFULL (p) (!p)
#define ISEMPY (p) (!p)

for(int i=0;i<MAX_STACK;i++)
    top[i] = NULL;
```

# Linked List

```
void add_stack(stack_pointer *top, item x)
{
    stack_pointer p = (stack *)malloc (sizeof(stack));
    if(ISFULL(p))
    {
        printf("\n Memory allocation failed");
        exit(1);
    }
    p->x = x;
    p->next = *top;
    *top = p;
}
```

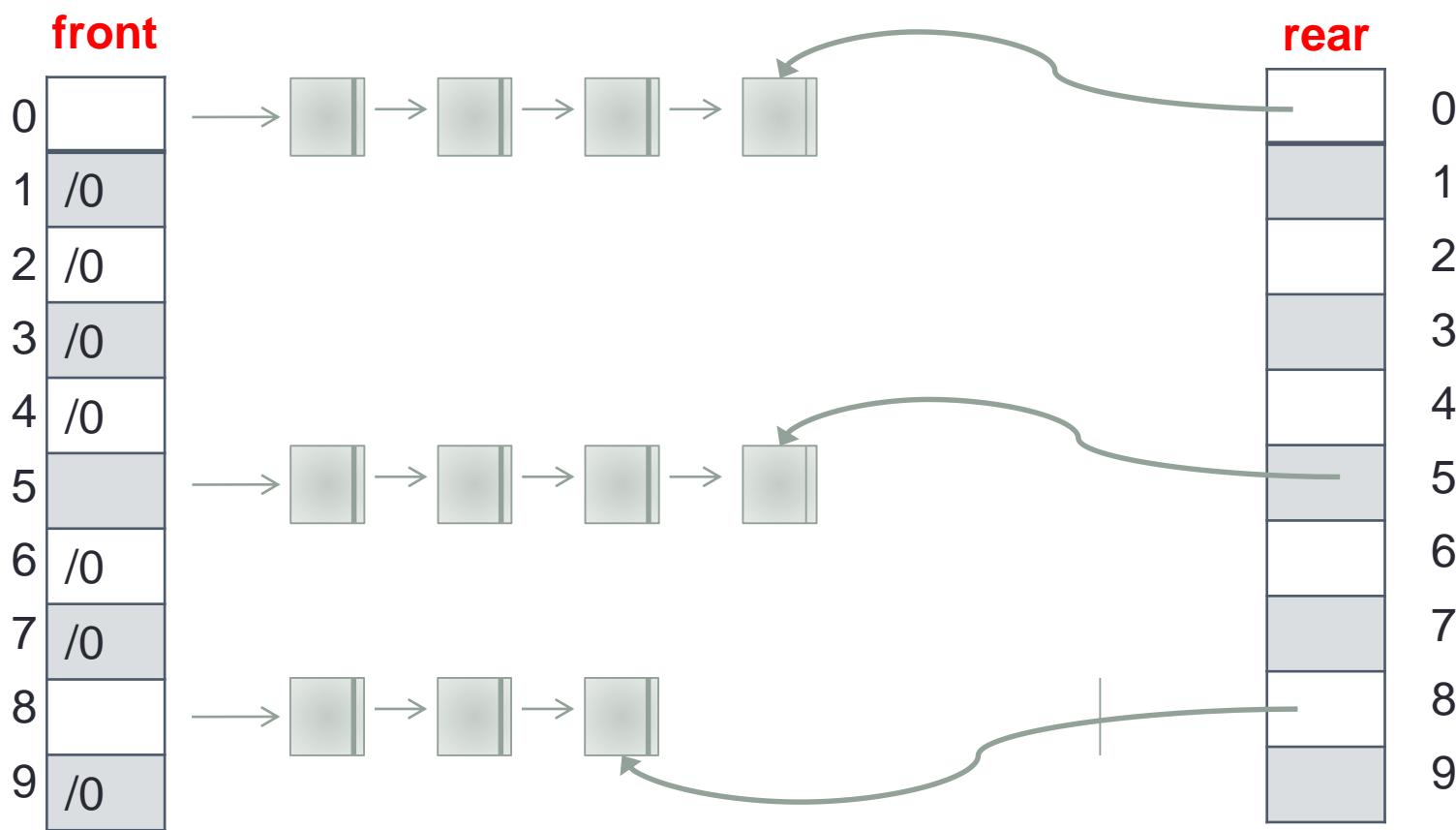
# Linked List

```
item remove_stack(stack_pointer *top)
{
    stack_pointer p = *top;
    item x;

    if (ISEMPTY(p))
    {
        printf("\n Stack is empty!");
        exit(1);
    }
    x = p->x;
    *top = p->next;
    free(p);
    return(x);
}
```

# **Multiple Queue Implementation Using Linked List**

# Multiple Queue Implementation Using Linked List



# Linked List

```
#define MAX_QUEUE 10 // number of max queues

typedef struct queue *queue_pointer;
typedef struct queue{
    item value;
    queue_pointer link;
} ;

queue_pointer front[MAX_QUEUE], rear[MAX_QUEUE];
for(i=0;i<MAX_QUEUE;i++)
    front[i]=NULL;
```

# Linked List

```
void add_queue(queue_pointer *front, queue_pointer* rear, item x)
{
    queue_pointer p=(queue_pointer)malloc(sizeof(queue));
    if(ISFULL(p)){
        printf("\n Memory allocation failed!!!\n");
        exit(1);
    }
    p->value=x;
    p->link=NULL;
    if(*front)
        (*rear)->link=p;
    else
        *front=p;
    *rear=p;
}
```

# Linked List

```
item remove_queue(queue_pointer *front)
{
    queue_pointer p=*front;
    item x;
    if(ISEMPTY(*front)) {
        printf("\nQueue is empty!!!!");
        exit(1);
    }

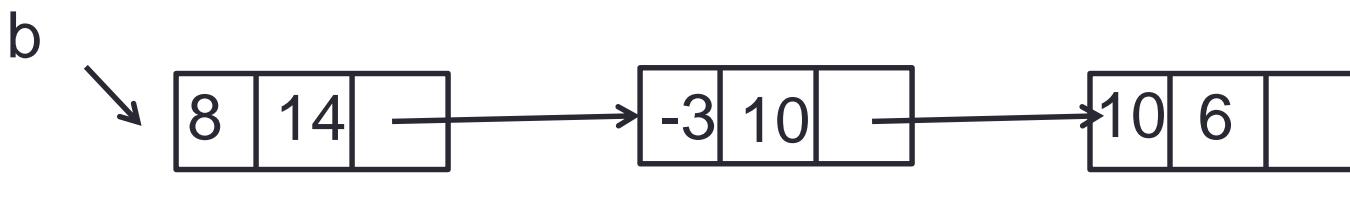
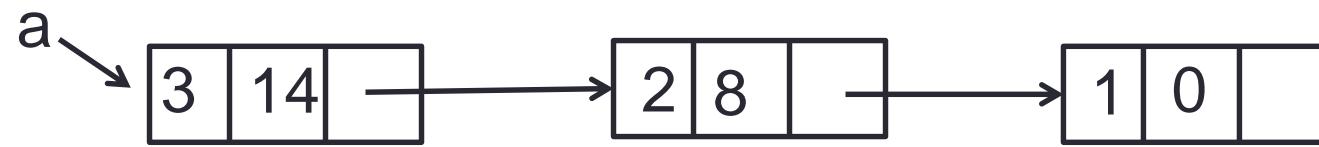
    x=p->value;
    *front=p->link;
    free(p);
    return(x);
}
```

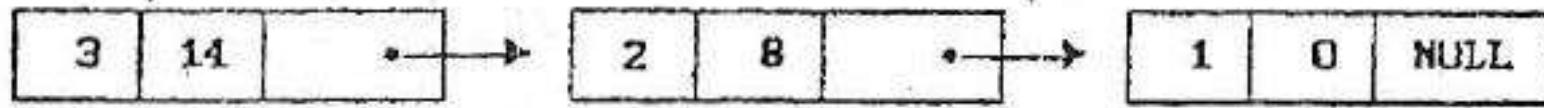
# **Addition of polynomials by using singly linked lists**

# Adding Polynomials using linked list

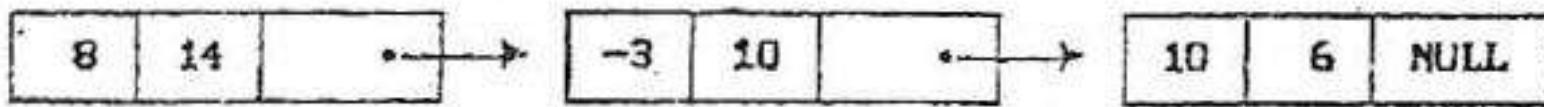
```
typedef struct poly_node* poly_pointer;
typedef struct poly_node{
    int coefficient;
    int expon;
    poly_pointer link;
};
poly_pointer a,b,d;
```

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

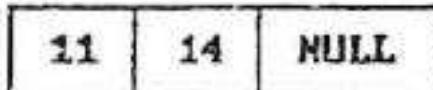




↑  
a



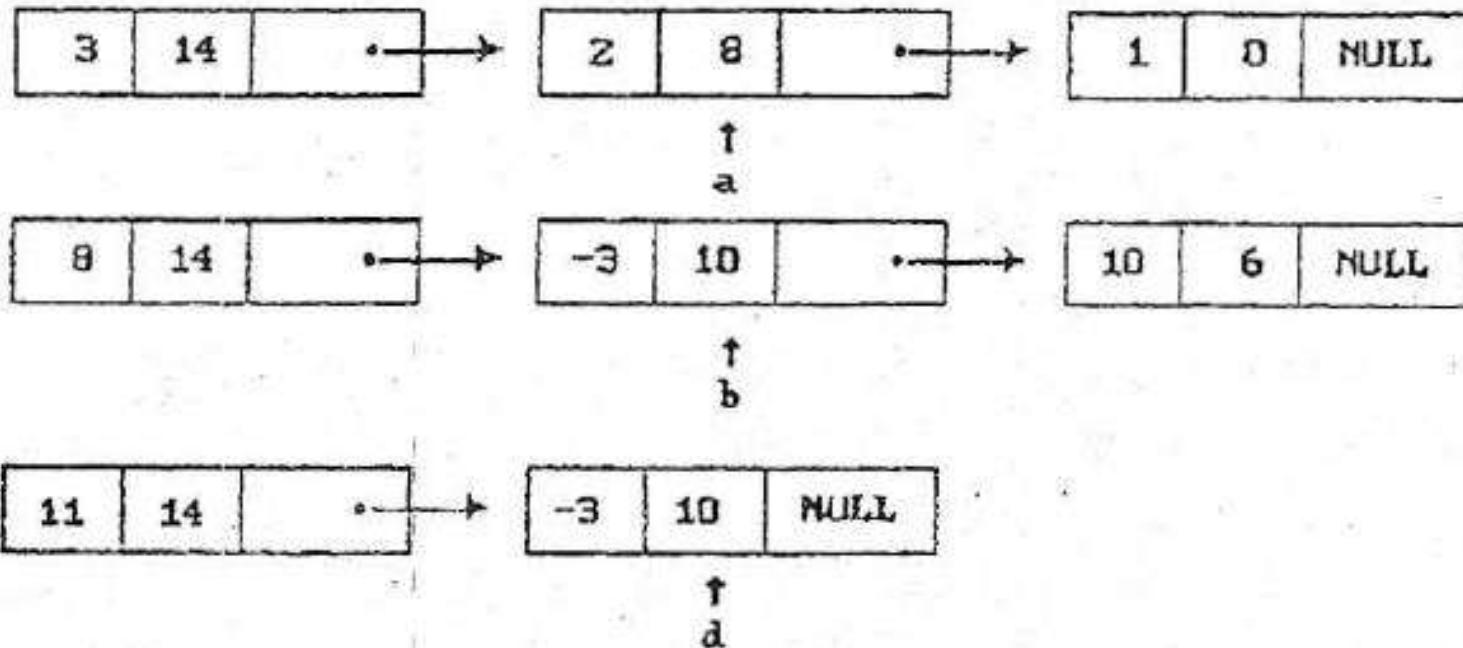
↑  
b



↑  
d

(a) a  $\rightarrow$  expon == b  $\rightarrow$  expon

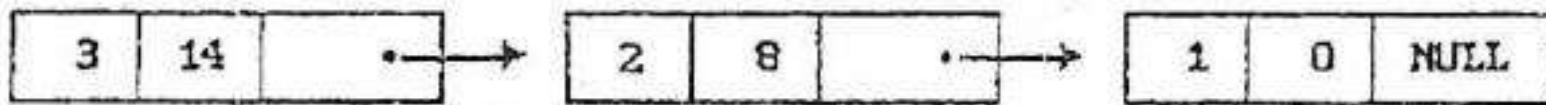
If the exponents of two terms are equal, we add the two coefficients and create a new term, also move the pointers to the next nodes in a and b.



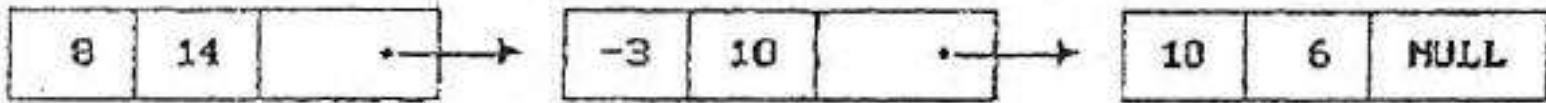
(b)  $a \rightarrow \text{expon} < b \rightarrow \text{expon}$

If the exponent of the current term in a is less than the exponent of the current term in b, then we create a duplicate term of b, Attach this term to the result, called d. Then move the move the pointer to the next term in b.

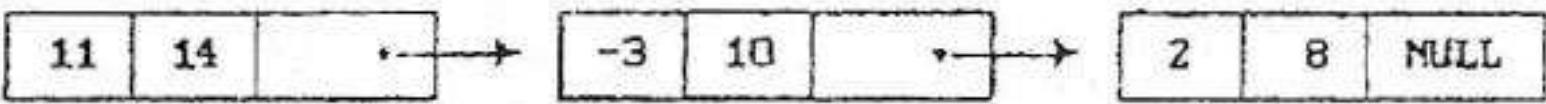
(Similarly, if the exponent of the current term in a is larger than the exponent of the current term in b.)



↑  
a



↑  
b



↑  
d

(c) a → expon > b → expon

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

Complexity of padd: O(m+n).,  
 m and n, the number of terms in the  
 polynomials a and b. Why?

```
poly_pointer padd(poly_pointer a, poly_pointer b) {
/* return a polynomial which is the sum of a and b */
    poly_pointer front, rear, temp;
    int sum;
    rear = (poly_pointer)malloc(sizeof(poly_node));
    if (IS_FULL(rear)){
        fprintf(stderr, "The memory is full\n");
        exit(1);
    }
    front = rear;
    while (a && b)
        switch (COMPARE(a->expon,b->expon)) {
            case -1: /* a->expon < b->expon */
                attach(b->coef,b->expon, &rear);
                b = b->link; break;
            case 0: /* a->expon = b->expon */
                sum = a->coef + b->coef;
                if(sum) attach(sum,a->expon,&rear);
                a = a->link; b = b->link; break;
            case 1: /* a->expon > b->expon */
                attach(a->coef,a->expon,&rear);
                a = a->link;
        }
}
```

```
/* copy rest of list a and then list b */
for (; a; a=a->link) attach(a->coef,a->expon,&rear);
for (; b; b=b->link) attach(b->coef,b->expon,&rear);
rear->link = NULL;
/* delete extra initial node */
temp = front; front = front->link; free(temp);
return front; }

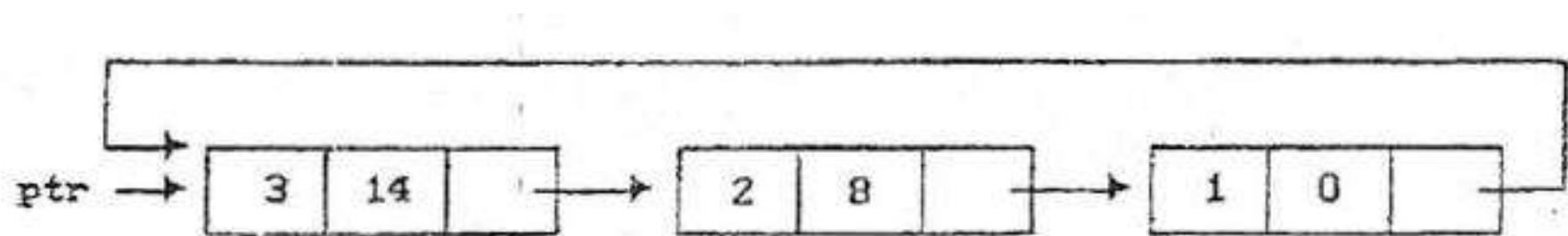
void attach(float coefficient,int exponent,poly_pointer *ptr){
/* create a new node with coef = coefficient and expon =
exponent,attach it to the node pointed to by ptr. ptr is
updated to point to this new node */
poly_pointer temp;
temp = (poly_pointer)malloc(sizeof(poly_node));
if (IS_FULL(temp)){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

# Singly-linked circular lists

# Representing polynomials as circularly linked List

We can free the nodes of a polynomial more efficiently if we have a circular list:

In a circular list, the last node is linked to the first node.



$$\text{ptr}(x) = 3x^{14} + 2x^8 + 1$$

## Methodology:

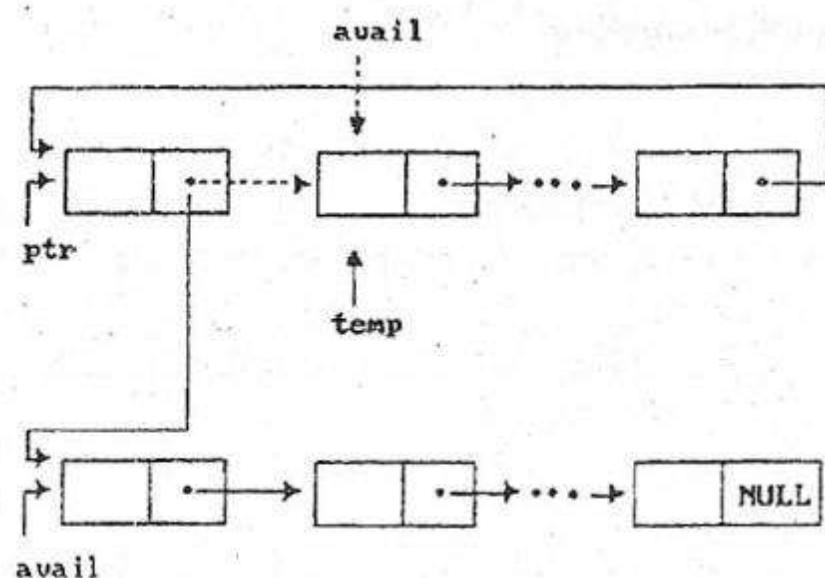
We free nodes that are no longer in use so that we may reuse these nodes later by:

- Obtaining an efficient erase algorithm for circular lists
- Maintaining our own list (as a chain) of nodes that have been “freed”
- Examining this list, when we need a new node and using one of the nodes if this list is not empty
- And using “malloc” only if this list is empty

- Let `avail` be a variable of type `poly_pointer` that points to the first node in our list of free nodes.

- Instead of using “malloc” and “free”, we use `get_node` and `ret_node`.

(See next slide)



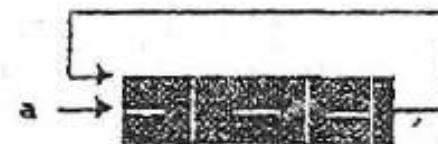
```
poly_pointer get_node() {
/*provide a node for use*/
    poly_pointer node;
    if (avail) {
        node = avail;
        avail = avail->link;
    }
    else{
        node = (poly_pointer) malloc(sizeof(poly_node));
        if(IS_FULL(node)){
            fprintf(stderr, The memory is full\n);
            exit(1);
        }
    }
    return node;
}
```

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

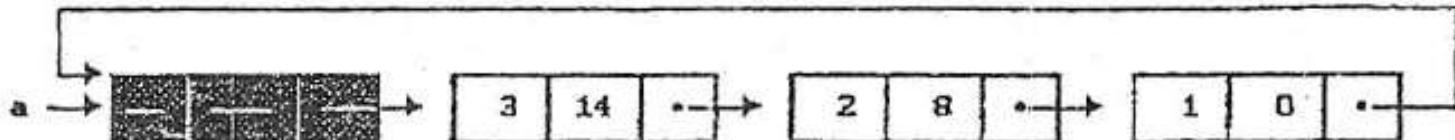
## Erasing in a circular list:

- We may erase a circular list in a fixed amount of time independent of the number of nodes in the list using `cerase`.
- To avoid the special case for **zero polynomial**, a head node is introduced for each polynomial. Therefore, also the zero polynomial contains a node.

```
void cerase(poly_pointer *ptr) {
/*erase the circular list ptr*/
    poly_pointer temp;
    if(*ptr) {
        temp = (*ptr)->link;
        (*ptr)->link = avail;
        avail = temp;
        *ptr = NULL;
    }
}
```



(a) Zero polynomial



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

# To change padd for circularly lists

- Add two variables, `starta = a` and `startb = b`.
- Prior to the while loop, assign `a = a->link` and `b = b->link`.
- Change the while loop to `while(a!=starta && b!=startb)`
- Change the first for loop to `for (;a!=starta; a=a->link)` .
- Change the second for loop to `for (;b!=startb; b=b->link)` ..
- Delete the lines:  
`rear->link = NULL;`  
`/*delete extra initial node */`
- Change the lines:  
`temp = front;`  
`front = front->link;`  
`free(temp);`  
to `rear->link = front;`

```
poly_pointer cpadd(poly_pointer a, poly_pointer b) {
/* polynomials a and b are singly linked circular lists with a head node. Return
a polynomial which is the sum of a and b */
    poly_pointer starta, d, lastd;
    int sum, done = FALSE;
    starta = a;                      /*record start of a*/
    a = a->link;                    /*skip headnode for a and b*/
    b = b->link;
    d = get_node();                  /*get a head node for sum*/
    d->expon = -1; lastd = d;
    do{
        switch (COMPARE(a->expon,b->expon)) {
            case -1: /* a->expon < b->expon */
                attach(b->coef, b->expon, &lastd);
                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,&lastd);
                    a = a->link; b = b->link; }
                break;
            case 1: /* a->expon > b->expon */
                attach(a->coef,a->expon,&lastd);
                a = a->link;
        }
    }while(!done);
    lastd->link = d;
    return d;}
```

# Hashtable

# Hashing

- Another important and widely useful technique for implementing dictionaries
- Constant time per operation (on the average)
- Worst case time proportional to the size of the set for each operation (like linked list and arrays).

# Basic Idea

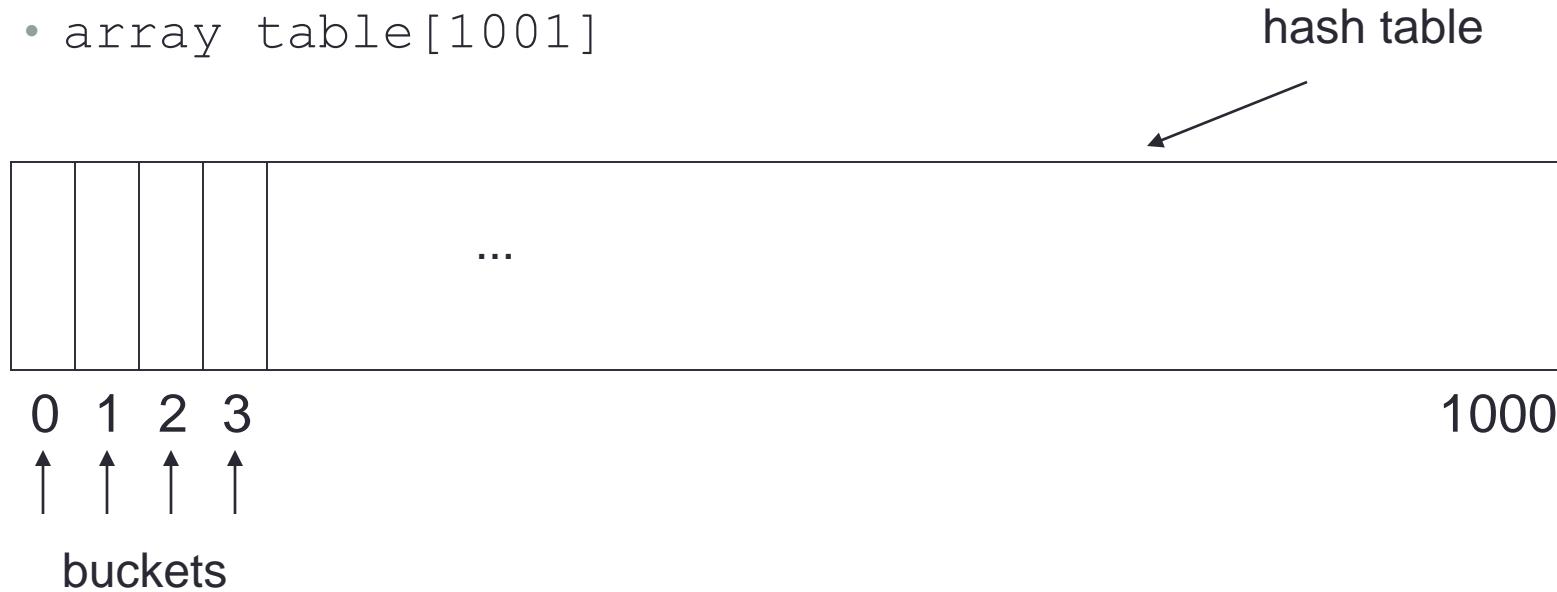
- Use *hash function* to map keys into positions in a *hash table*.

## Ideally

- If element  $e$  has key  $k$  and  $h$  is hash function, then  $e$  is stored in position  $h(k)$  of table
- To search for  $e$ , compute  $h(k)$  to locate position. If no element, dictionary does not contain  $e$ .

# Example

- Dictionary Student Records
  - Keys are ID numbers (951000 - 952000), no more than 100 students
  - Hash function:  $h(k) = k - 951000$  maps ID into distinct table positions 0-1000
  - array table[1001]



# Analysis (Ideal Case)

- $O(b)$  time to initialize hash table ( $b$  number of positions or buckets in hash table)
- $O(1)$  time to perform *insert*, *remove*, *search*

# Ideal Case is Unrealistic

- Works for implementing dictionaries, but many applications have key ranges that are too large to have 1-1 mapping between buckets and keys!

## Example:

- Suppose key can take on values from 0 .. 65,535 (2 byte unsigned int)
- Expect  $\approx$  1,000 records at any given time
- Impractical to use hash table with 65,536 slots!

# Hash Functions

- If key range too large, use hash table with fewer buckets and a hash function which maps multiple keys to same bucket:  
$$h(k_1) = \beta = h(k_2)$$
:  $k_1$  and  $k_2$  have **collision** at slot  $\beta$
- Popular hash functions: hashing by division  
$$h(k) = k \% D$$
, where  $D$  number of buckets in hash table
- Example: hash table with 11 buckets  
$$h(k) = k \% 11$$
  
80 → 3 ( $80 \% 11 = 3$ ), 40 → 7, 65 → 10  
58 → 3 collision!

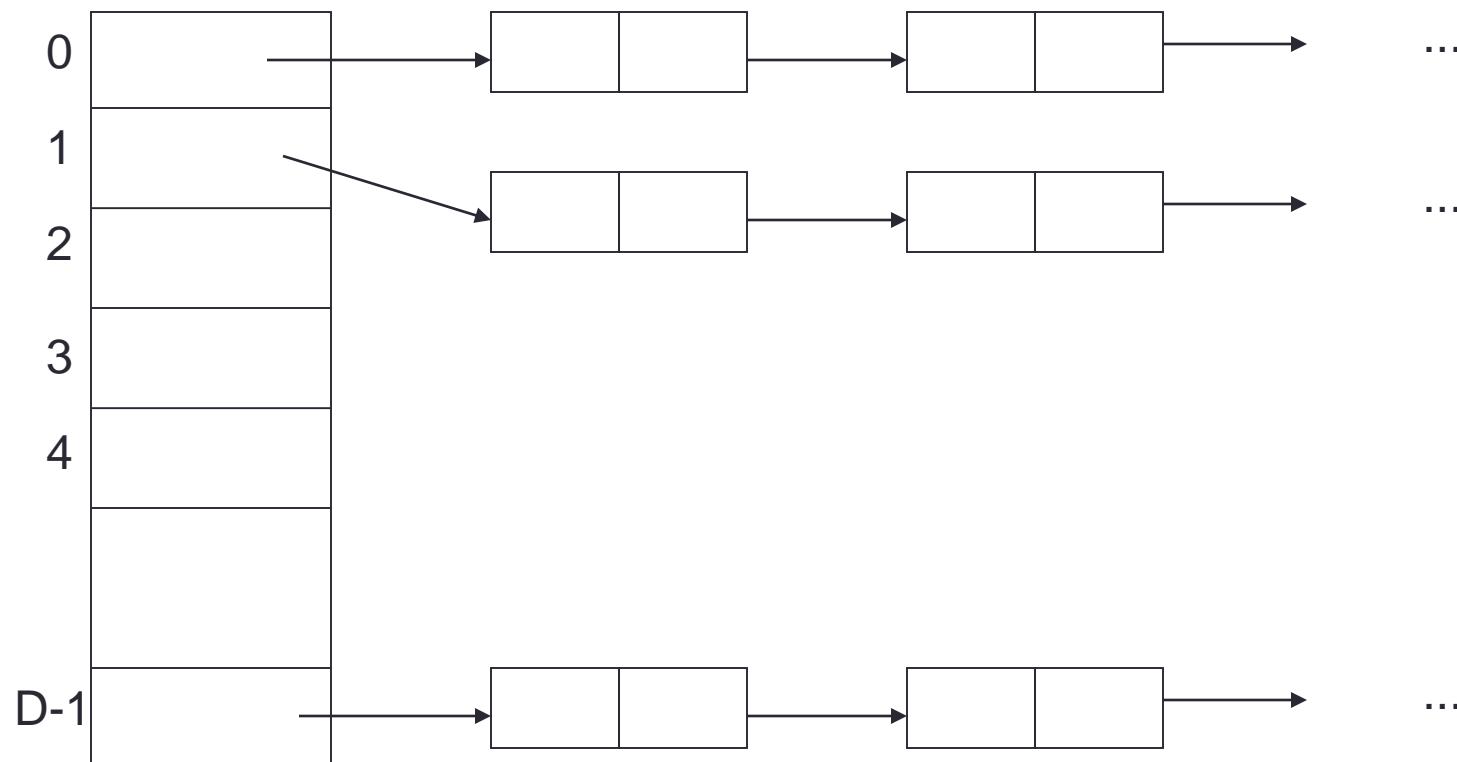
# Collision Resolution Policies

- Two classes:
  - (1) Open hashing, a.k.a. separate chaining
  - (2) Closed hashing, a.k.a. open addressing
- Difference has to do with whether collisions are stored *outside the table* (open hashing) or whether collisions result in storing one of the records at *another slot in the table* (closed hashing)

# Open Hashing

- Each bucket in the hash table is the head of a linked list
- All elements that hash to a particular bucket are placed on that bucket's linked list
- Records within a bucket can be ordered in several ways
  - by order of insertion, by key value order, or by frequency of access order

# Open Hashing Data Organization



# Analysis

- Open hashing is most appropriate when the hash table is kept in main memory, implemented with a standard in-memory linked list
- We hope that number of elements per bucket roughly equal in size, so that the lists will be short
- If there are  $n$  elements in set, then each bucket will have roughly  $n/D$
- If we can estimate  $n$  and choose  $D$  to be roughly as large, then the average bucket will have only one or two members

# Analysis Cont'd

## Average time per dictionary operation:

- $D$  buckets,  $n$  elements in dictionary  $\Rightarrow$  average  $n/D$  elements per bucket
- *insert, search, remove* operation take  $O(1+n/D)$  time each
- If we can choose  $D$  to be about  $n$ , constant time
- Assuming each element is likely to be hashed to any bucket, running time constant, independent of  $n$

# Data Structure for Chaining

```
#define MAX_CHAR 10
#define TABLE_SIZE 13
#define IS_FULL(ptr) (!!(ptr))

typedef struct {
    char key[MAX_CHAR];
    /* other fields */
} element;

typedef struct list *list_pointer;
typedef struct list {
    element item;
    list_pointer link;
};

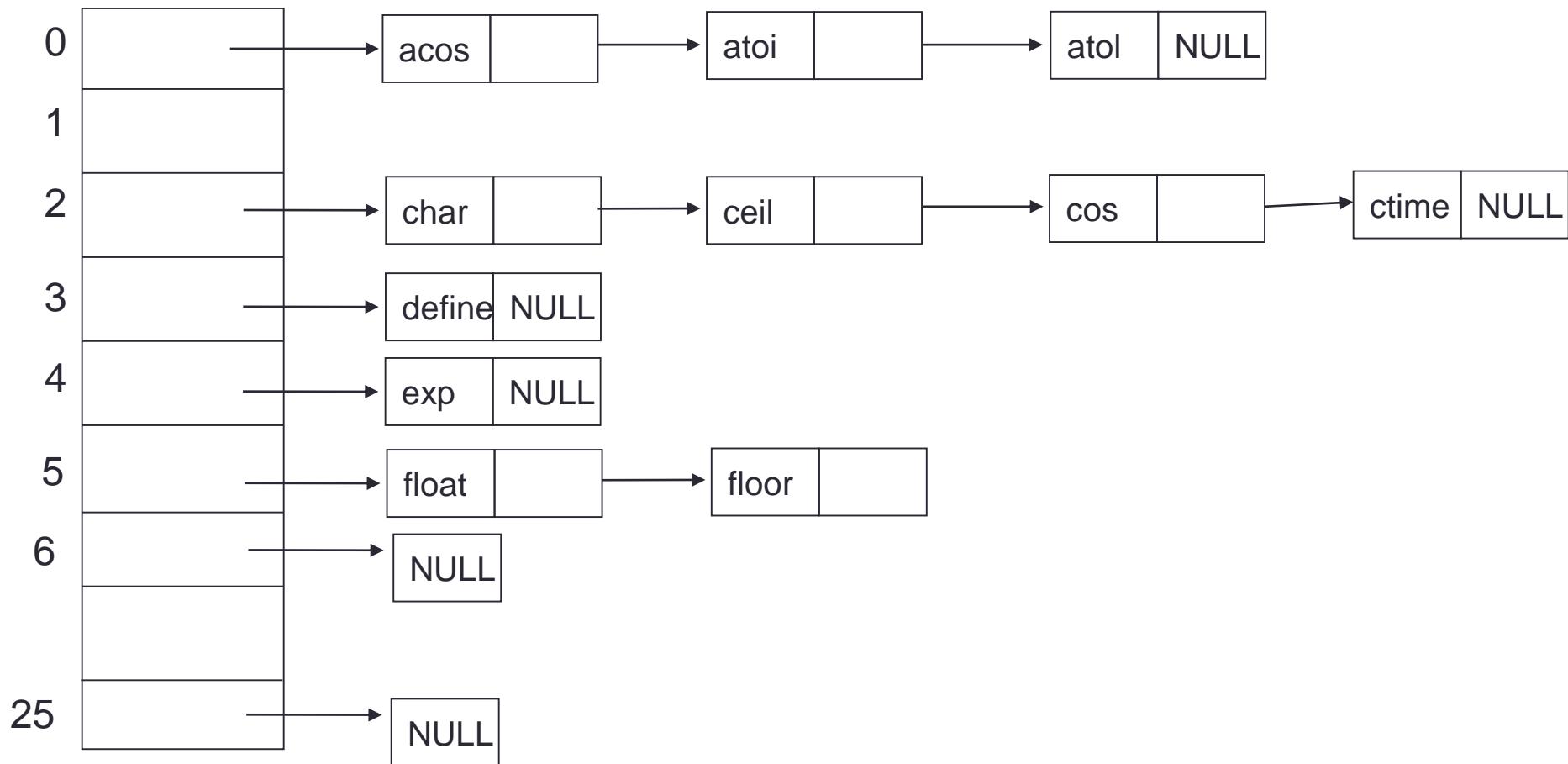
list_pointer hash_table[TABLE_SIZE];
```

# Chain Insert

```
void chain_insert(element item, list_pointer ht[])
{
    int hash_value = hash(item.key);
    list_pointer ptr, trail=NULL, lead=ht[hash_value];
    for (; lead; trail=lead, lead=lead->link)
        if (!strcmp(lead->item.key, item.key)) {
            fprintf(stderr, "The key is in the table\n");
            exit(1);
        }
    ptr = (list_pointer) malloc(sizeof(list));
    if (IS_FULL(ptr)) {
        fprintf(stderr, "The memory is full\n");
        exit(1);
    }
    ptr->item = item;
    ptr->link = NULL;
    if (trail) trail->link = ptr;
    else ht[hash_value] = ptr;
}
```

# Results of Hash Chaining

acos, atoi, char, define, exp, ceil, cos, float, atol, floor, ctime  
 $f(x)$ =first character of  $x$



# **BBM 201**

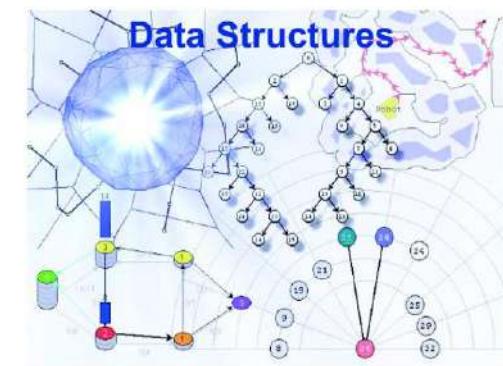
# **DATA STRUCTURES**

---

**Lecture 10:**  
Doubly Linked Lists



**2018-2019 Fall**

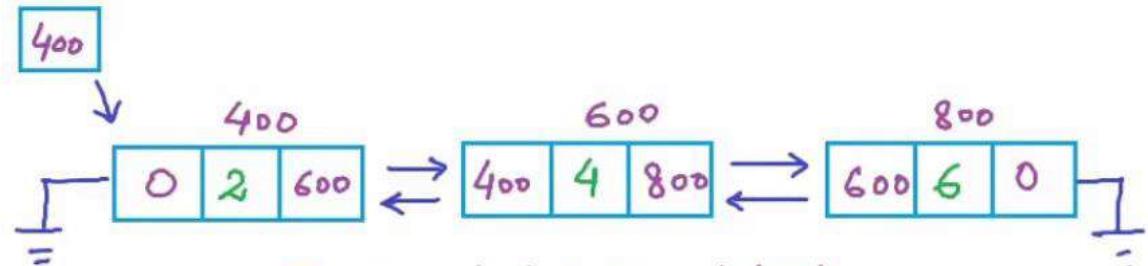


# Doubly Linked Lists

Doubly Linked List - Implementation

head

```
struct Node {  
    int data;  
    struct Node* next;  
    struct Node* prev;  
};
```



Insert At Head(x)

Insert At Tail(x)

Print()

ReversePrint()

Each node stores not only the address of the next node, but also the address of the previous node. So, each node stores three fields.

**Advantage of doubly linked list:** Reverse look-up that we could not do using a linked list. For example, deletion is much faster than for a linked list.

For **temp** being 600, **temp->next** points to the address 800 and **temp->prev** points to the address 400.

```
/* Doubly Linked List implementation */
#include<stdio.h>
#include<stdlib.h>
struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
};
struct Node* head; // global variable - pointer to head node.
void InsertAtHead(int x) {
    // local variable
    // Will be cleared from memory when function call will finish
    struct Node myNode;
    myNode.data = x;
    myNode.prev = NULL;           I
    myNode.next = NULL;
}
```

Note: **head** is a global variable. Each node inside the **InsertAtHead** function is created locally and the node **myNode** does not exist after the function is executed.

Therefore, local node allocation is NOT preferred.

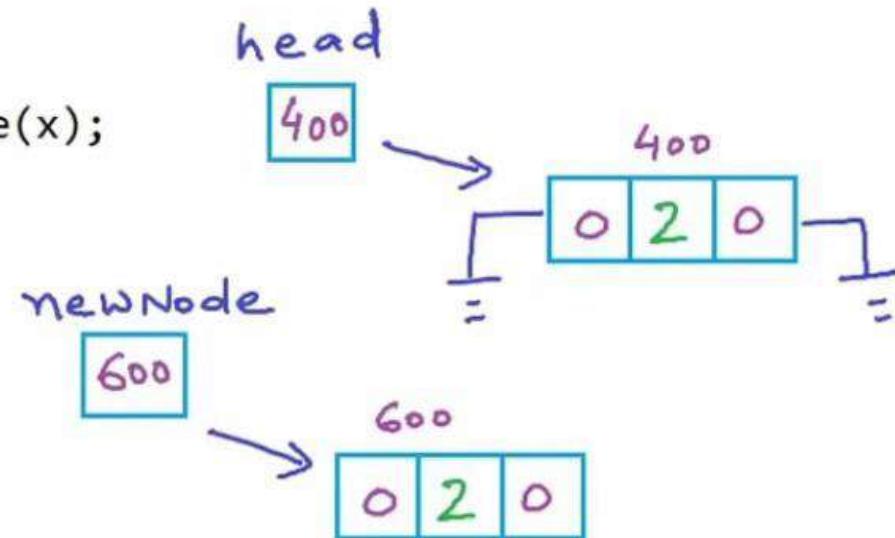
```
struct Node {  
    int data;  
    struct Node* next;  
    struct Node* prev;  
};  
struct Node* head; // global variable - pointer to head node.  
struct Node* GetNewNode(int x) {  
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));  
    newNode->data = x;  
    newNode->prev = NULL;  
    newNode->next = NULL;  
    return newNode;  
}  
void InsertAtHead(int x) {  
}
```

Each node “**newNode**” is created in the dynamic memory and the node exists after the function is executed.

Now, we create a new node in a separate function, called **GetNewNode**.

# Doubly Linked List - Implementation

```
void InsertAtHead(int x) {  
    struct Node* newNode = GetNewNode(x);  
    if(head == NULL) {  
        head = newNode;  
        return;  
    }  
    head->prev = newNode;  
    newNode->next = head;  
    head = newNode;  
}
```



Insert At Head (2)

Insert At Head (4)

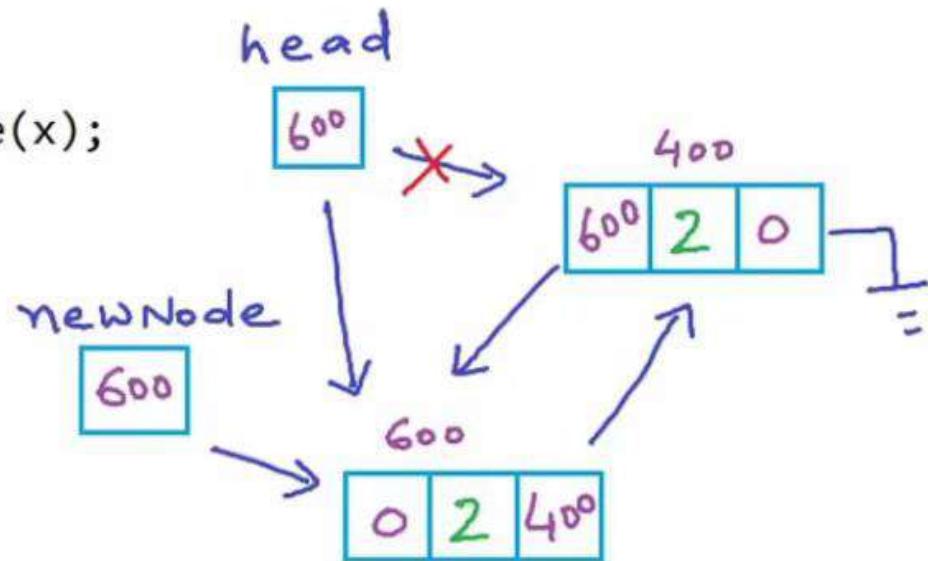
Now, one node is created in the list with head pointing to it using the line **head = newNode**.

We have two nodes, **head** is pointing to the node at address 400 and **newNode** is pointing to the node at address 600.

Typo: The data-field at address 600 has value 4 not 2.

# Doubly Linked List - Implementation

```
void InsertAtHead(int x) {  
    struct Node* newNode = GetNewNode(x);  
    if(head == NULL) {  
        head = newNode;  
        return;  
    }  
    → head->prev = newNode;  
    newNode->next = head;  
    head = newNode;  
}
```



Insert At Head (2)

Insert At Head (4)

Set the **prev-field** of the **head** node as 600 (address of the new node).

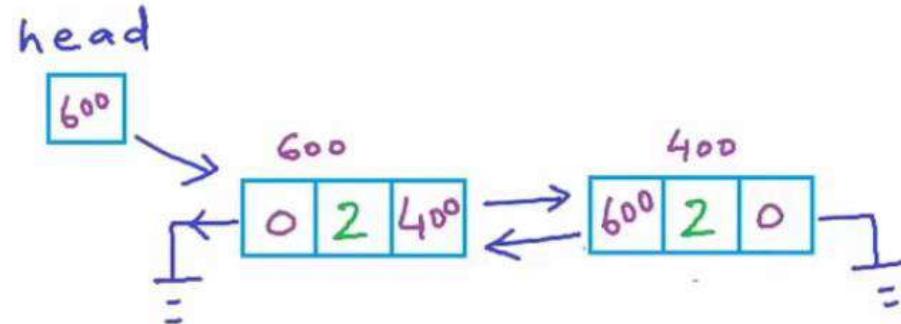
Then, set the **next-field** of the **new node** as 400 (the address of the head node).

And now, head can point to 600, that is the address of the final head node.

Typo: The data-field at address 600 has value 4 not 2.

## Doubly Linked List - Implementation

```
void InsertAtHead(int x) {  
    struct Node* newNode = GetNewNode(x);  
    if(head == NULL) {  
        head = newNode;  
        return;  
    }  
    head->prev = newNode;  
    newNode->next = head;  
    head = newNode;  
}
```



Insert At Head (2)

Insert At Head (4)

# Reverse Printing

```
void ReversePrint() {  
    struct Node* temp = head;  
    if(temp == NULL) return; // empty list, exit  
    // Going to last Node  
    while(temp->next != NULL) {  
        temp = temp->next;  
    }  
    // Traversing backward using prev pointer  
    printf("Reverse: ");  
    while(temp != NULL) {  
        printf("%d ",temp->data);  
        temp = temp->prev;  
    }  
    printf("\n");  
}
```

In Reverse-printing, the code first goes to the end of the list and then traverses backwards.

```
void ReversePrint() {
    struct Node* temp = head;
    if(temp == NULL) return; // empty list, exit
    // Going to last Node
    while(temp->next != NULL) {
        temp = temp->next;
    }
    // Traversing backward using prev pointer
    printf("Reverse: ");
    while(temp != NULL) {
        printf("%d ",temp->data);
        temp = temp->prev;
    }
    printf("\n");
}

int main() {
    head = NULL; // empty list.
    InsertAtHead(2); Print(); ReversePrint();
    InsertAtHead(4); Print(); ReversePrint();
    InsertAtHead(6); Print(); ReversePrint();
}
```

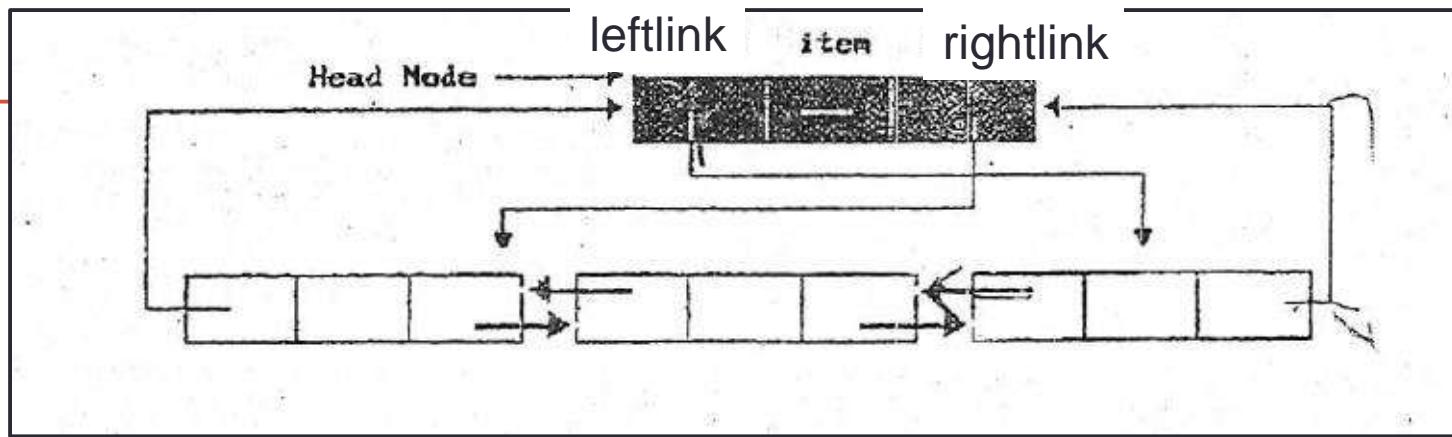
```
C:\Users\animesh\Documents\Visual Studio 2010\Projects\SampleApp
Forward: 2
Reverse: 2
Forward: 4 2
Reverse: 2 4
Forward: 6 4 2
Reverse: 2 4 6
```

# Doubly Circular Linked List

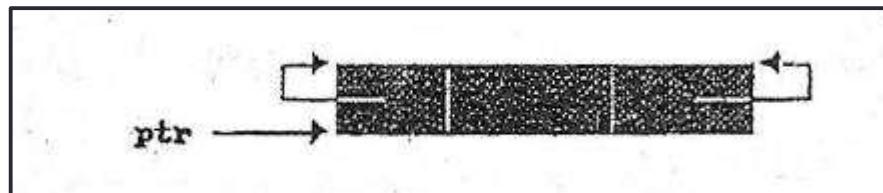
```
typedef struct node *node_pointer;
typedef struct node{
    node_pointer leftlink;
    element item;
    node_pointer rightlink;};
```

```
ptr = ptr->leftlink->rightlink = ptr->rightlink->leftlink
```

Doubly linked circular linked list with head node:



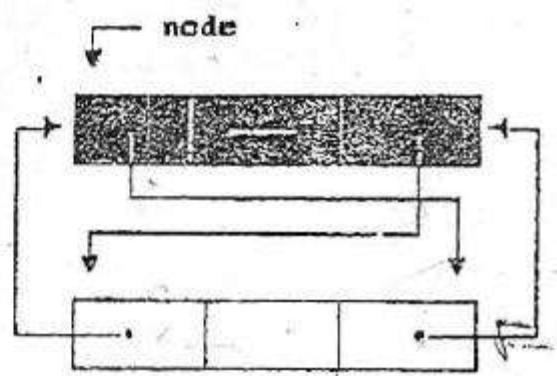
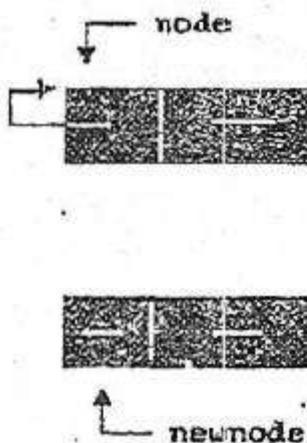
Empty doubly linked circular linked list with head node:



### Inserting into a doubly-linked circular list:

```
void dinsert(node_pointer node, node_pointer newnode)
{
    /* insert newnode to the right of node */
    newnode->leftlink = node;
    newnode->rightlink = node->rightlink;
    node->rightlink->leftlink = newnode;
    node->rightlink = newnode; }
```

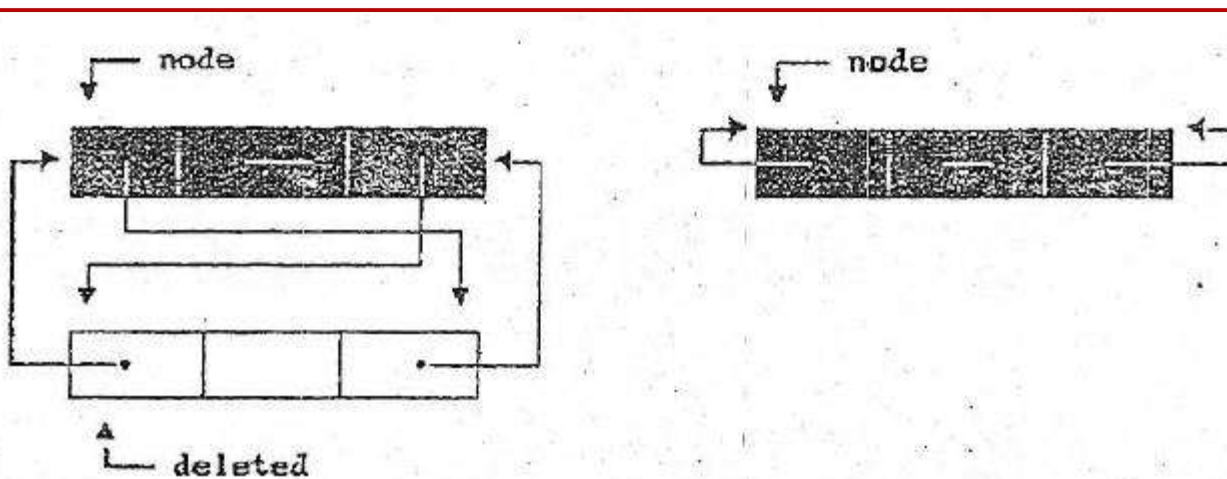
### Insertion into an empty doubly linked circular linked list:



### **Deletion from a doubly-linked circular list:**

```
void ddelete(node_pointer node, node_pointer deleted)
{
    if(node == deleted)
        printf("Deletion of head node not permitted.\n");
    else
        deleted->leftlink->rightlink = deleted->rightlink;
        deleted->rightlink->leftlink = deleted->leftlink;
        free(deleted) ;
}
```

### **Deletion from a doubly linked circular linked list:**



## Doubly vs. Singly Linked List

It uses extra space for previous pointers.

Insertion/Deletion has extra work.

You have ready access\insert on both ends.

It can work as a Queue and a Stack at the same time.

Node deletion requires no additional pointers.

# Reverse a doubly linked list

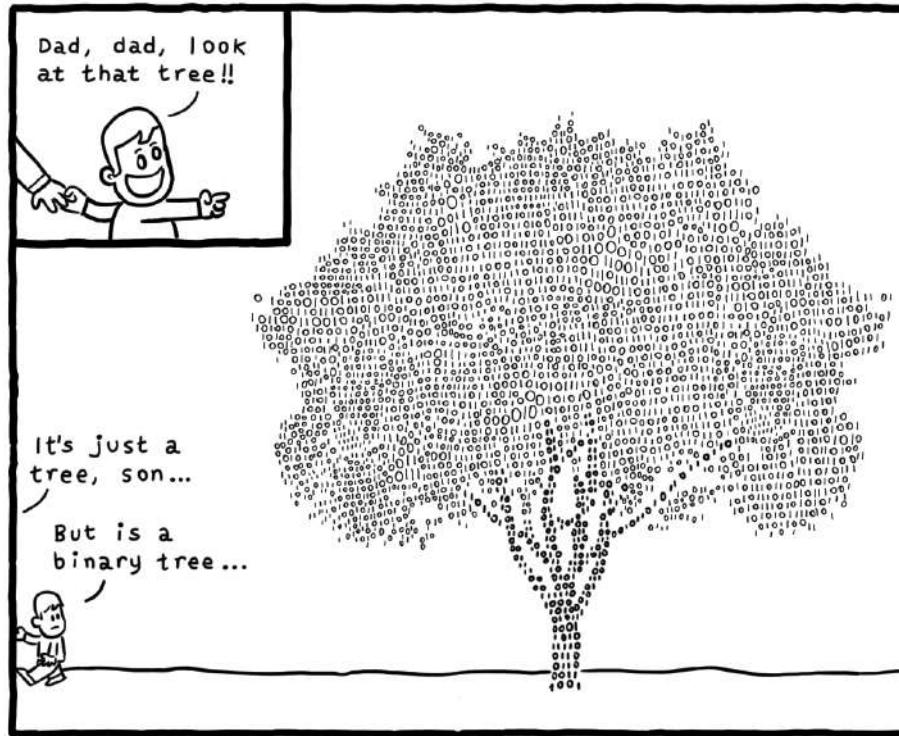
```
Node reverse(Node head)
{
    Node n = head, next;
    while(n.next != null) {
        next = n.next;
        n.next = n.prev;
        n.prev = next;
        n = next;
    }
    //for the last node
    next=n.next;
    n.next = n.prev;
    n.prev = next;
    // n is the new head.
    return n;
}
```

# BBM 201

# Data structures

## Lecture 11:

## Trees



Daniel Storl {turnoff.us}

2018-2019 Fall



# Content

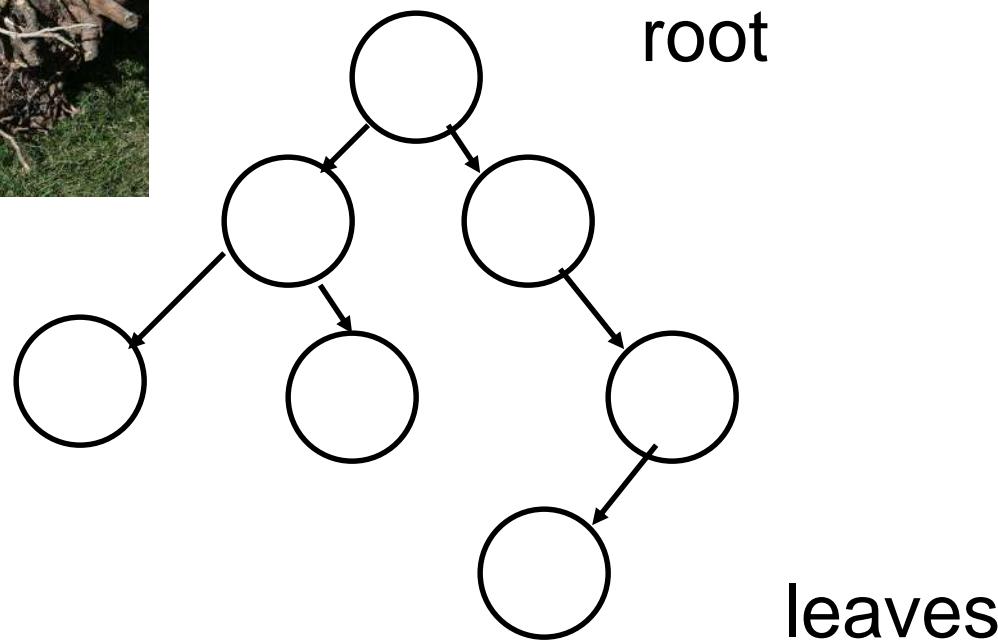
- Terminology
- The Binary Tree
- The Binary Search Tree

# Terminology

- Trees are used to represent relationships
- Trees are hierarchical in nature
  - “Parent-child” relationship exists between nodes in tree.
  - Generalized to ancestor and descendant
  - Lines between the nodes are called edges
- A **subtree** in a tree is any node in the tree together with all of its descendants

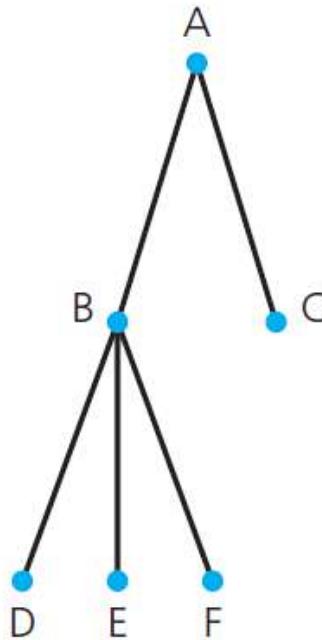
# Terminology

- Only access point is the root
- All nodes, except the root, have one parent
  - like the inheritance hierarchy in Java
- Traditionally trees are drawn upside down

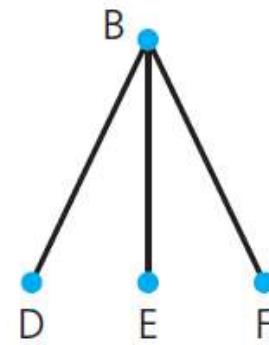


# Terminology

(a)



(b)



(a) A tree;

(b) a subtree of the tree in part a

# Terminology

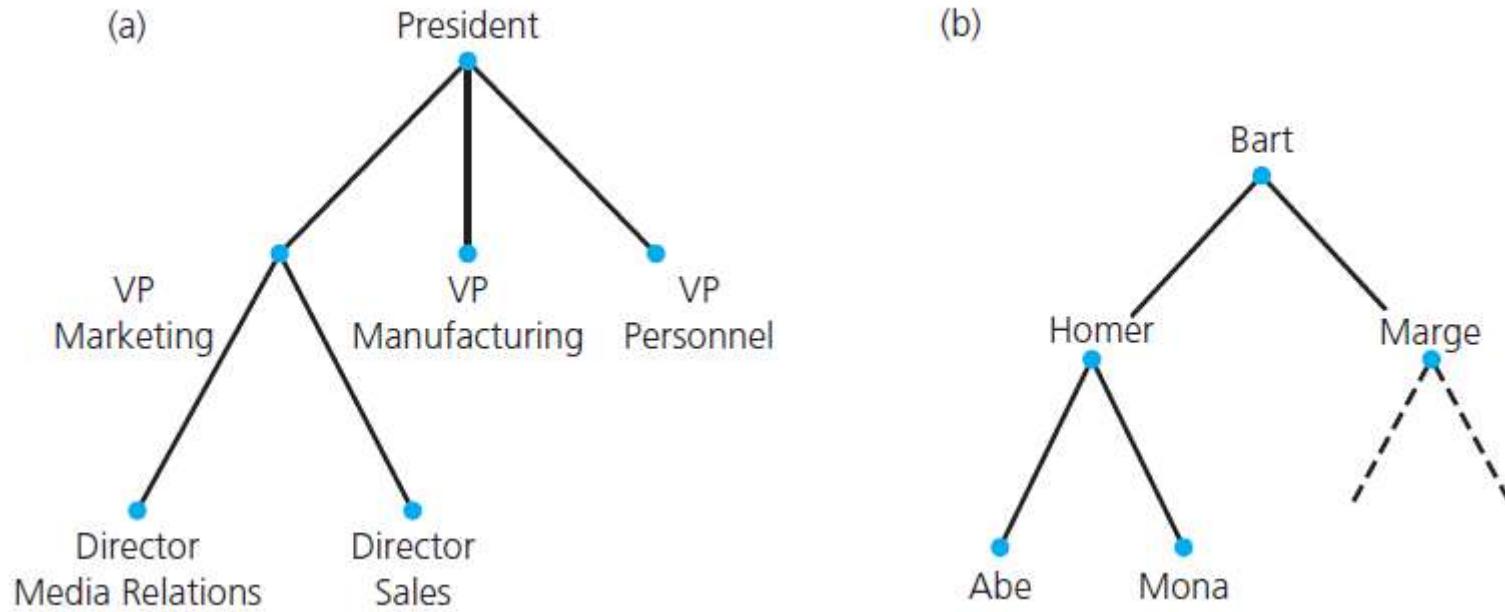
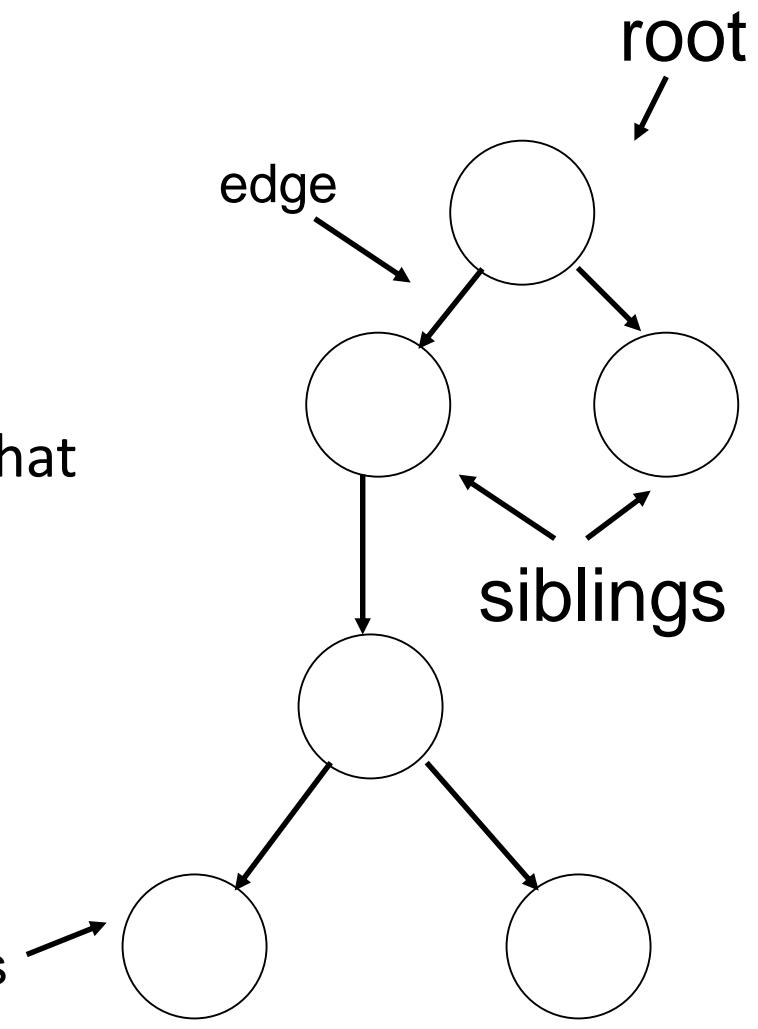


FIGURE 15-2 (a) An organization chart; (b) a family tree

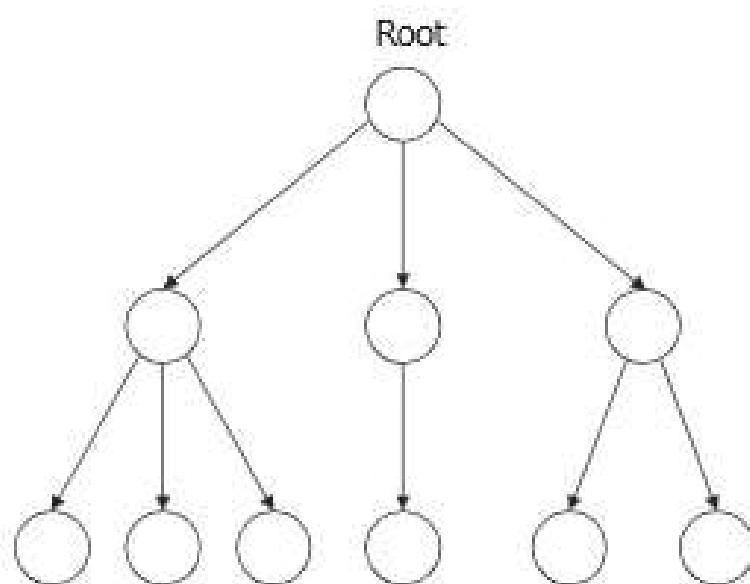
# Properties of Trees and Nodes

- *siblings*: two nodes that have the same parent
- *edge*: the link from one node to another
- *path length*: the number of edges that must be traversed to get from one node to another



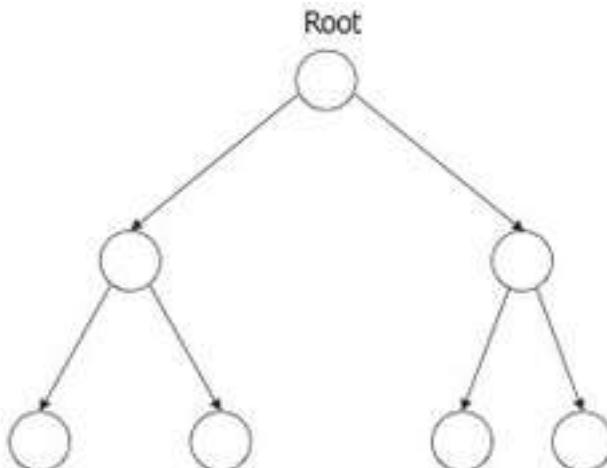
# General Tree

- A general tree is a data structure in that each node can have infinite number of children
- A general tree cannot be empty.



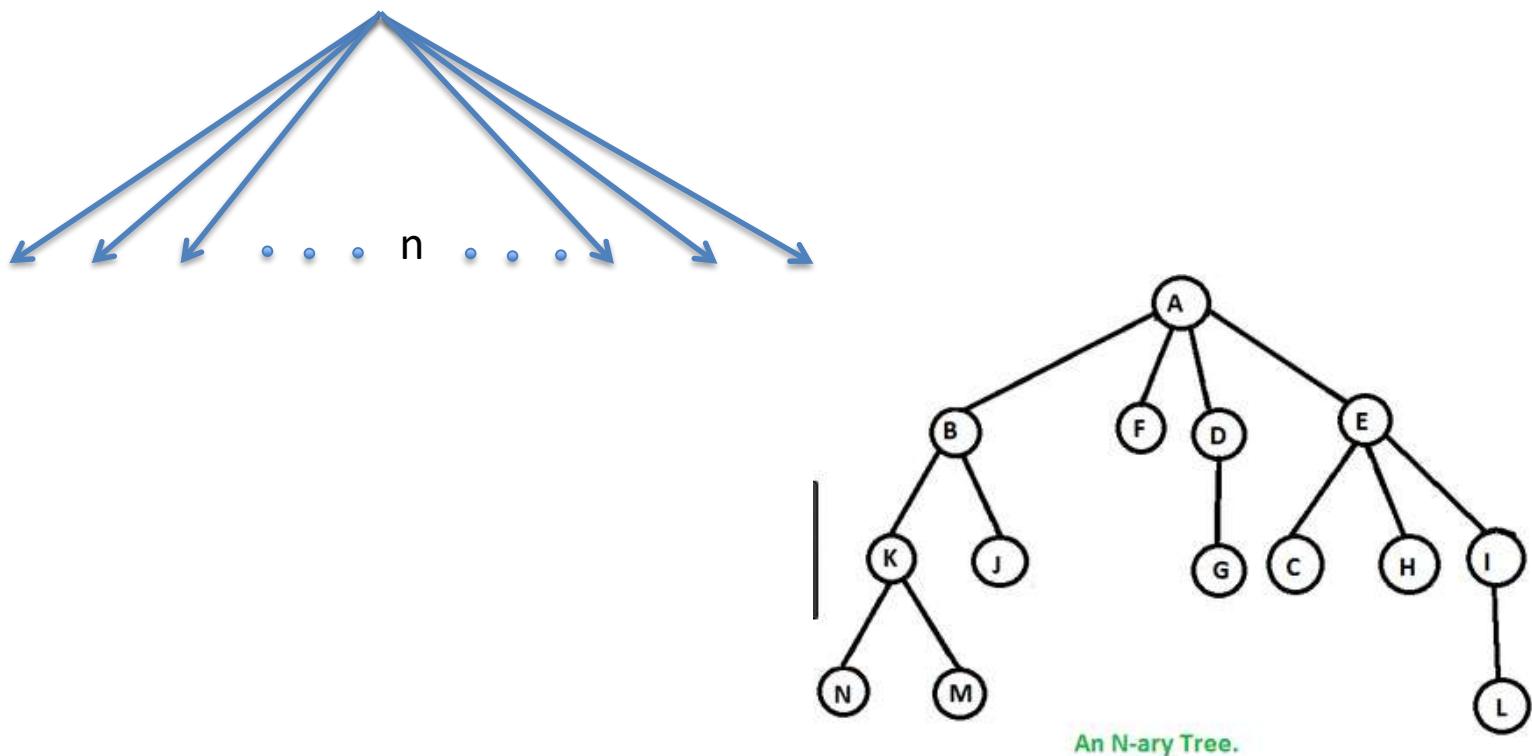
# Binary Tree

- A Binary tree is a data structure in that each node has at most **two nodes** left and right.
- A Binary tree can be **empty**.



# n -ary tree

- A generalization of a binary tree whose nodes each can have no more than n children.



# Example: Algebraic Expressions.

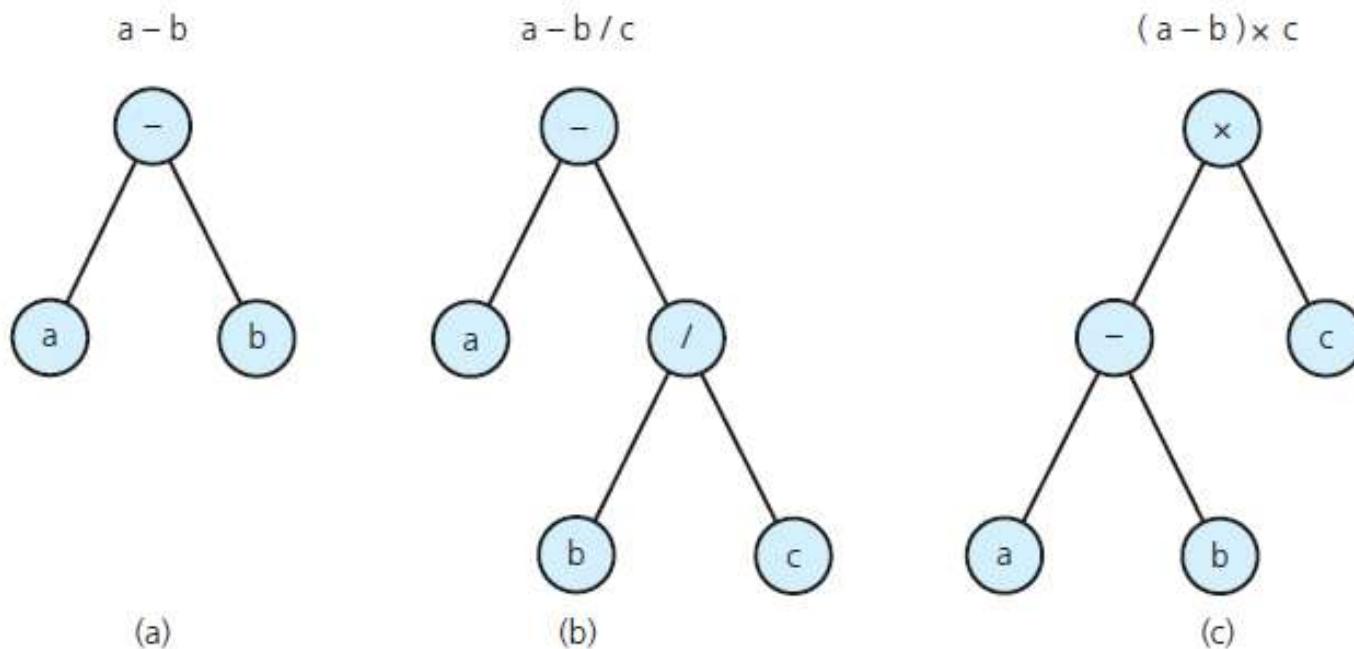
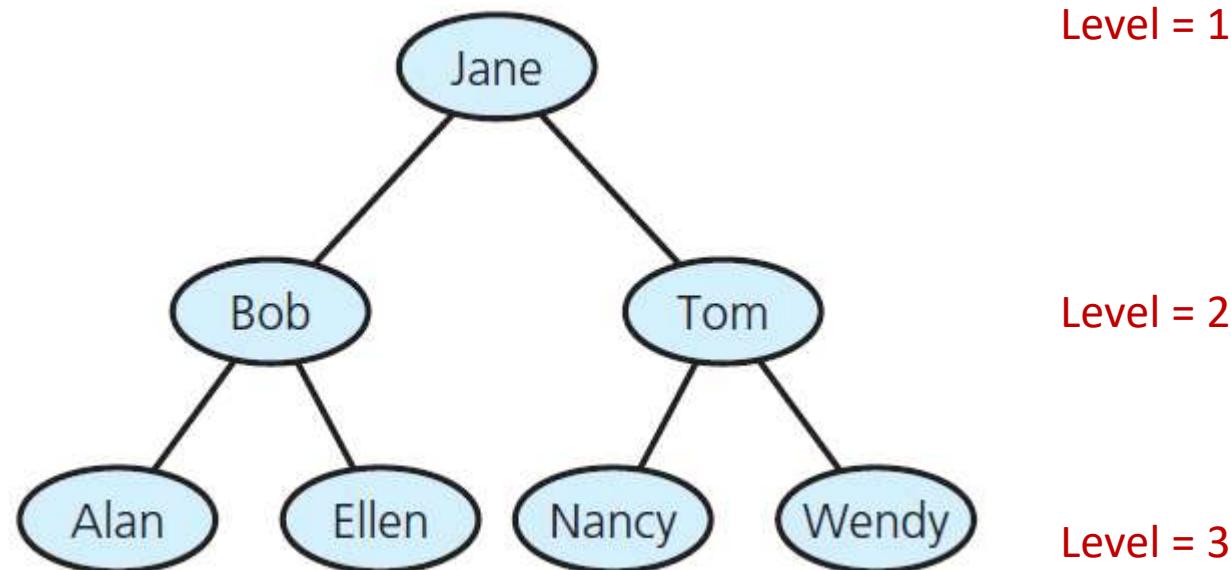


FIGURE 15-3 Binary trees that represent algebraic expressions

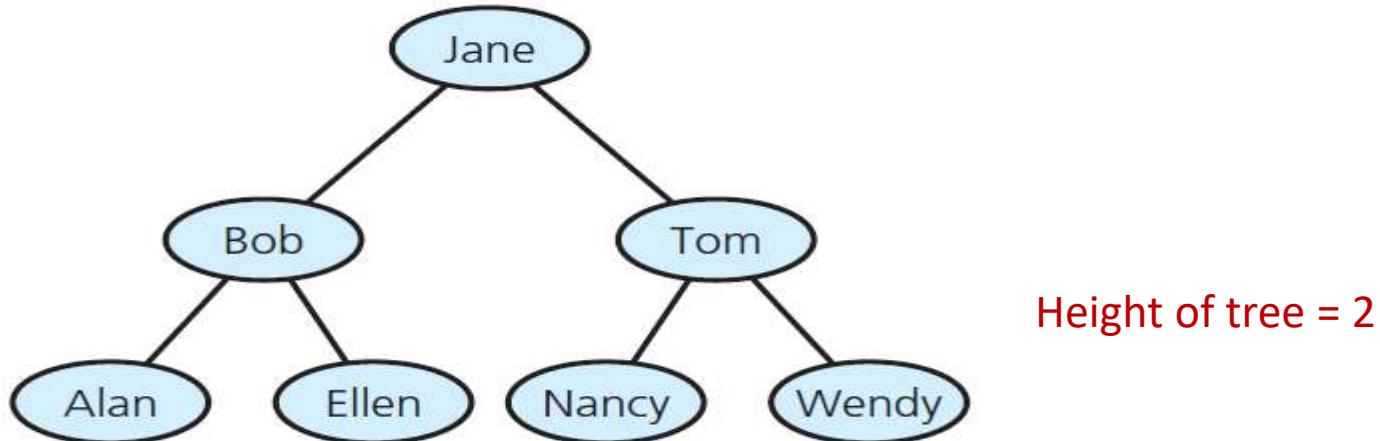
# Level of a Node

- Definition of the level of a node  $n$  :
  - If  $n$  is the root of  $T$ , it is at level 1.
  - If  $n$  is not the root of  $T$ , its level is 1 greater than the level of its parent.

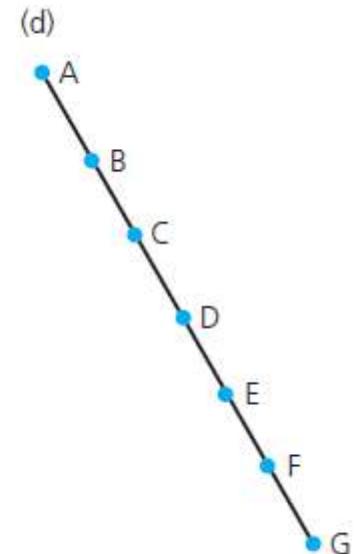
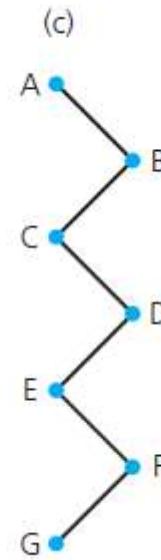
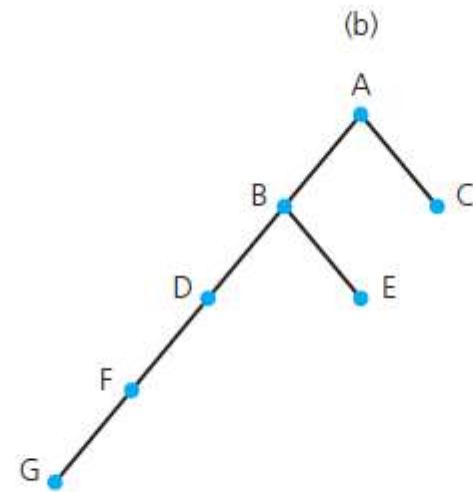
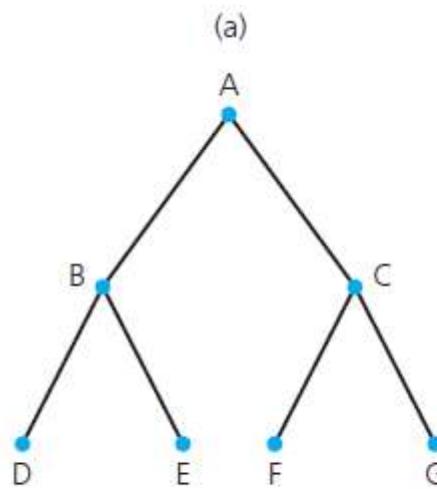


# Height of Trees

- *The height of a node is the number of edges on the longest downward path between that node and a leaf.*



# The Height of Trees



Height 2

Height 4

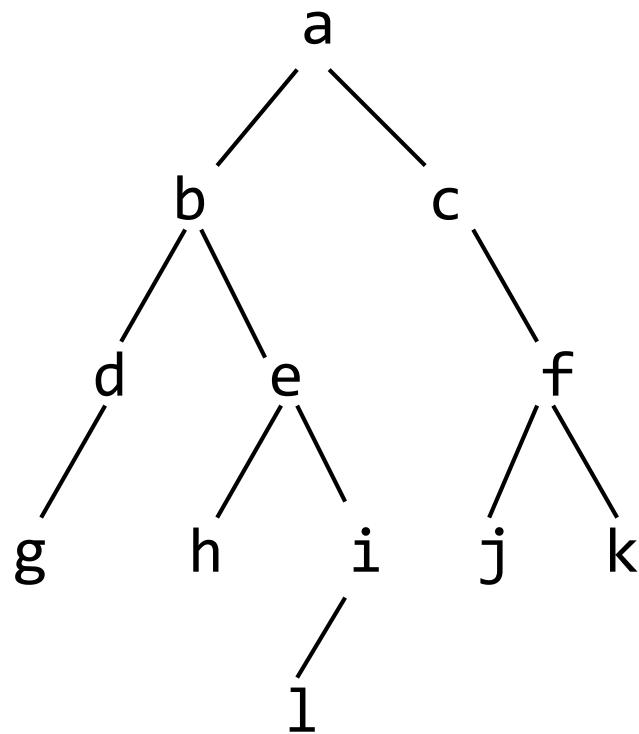
Height 6

Height 6

Binary trees with the same nodes but different heights

# Depth of a Tree

- The path length from the root of the tree to this node.



The **depth** of a node is its distance from the root

a is at depth zero

e is at depth 2

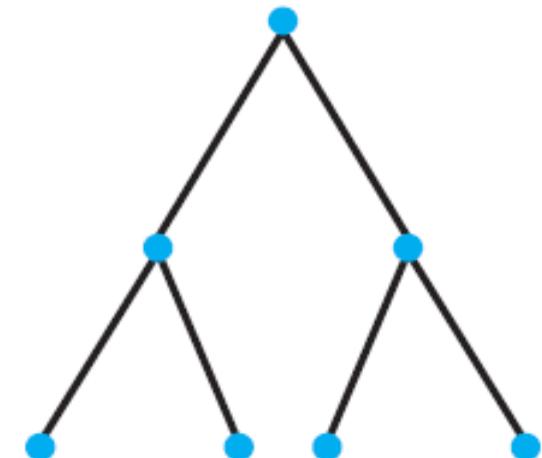
The **depth** of a binary tree is the depth of its deepest node

This tree has depth 4

# Full, Complete, and Balanced Binary Trees

# Full Binary Trees

- Definition of a full binary tree
  - If  $T$  is empty,  $T$  is a full binary tree of height 0.
  - If  $T$  is not empty and has height  $h > 0$ ,  $T$  is a full binary tree if its root's subtrees are both full binary trees of height  $h - 1$ .
  - Every node other than the leaves has two children.



# Facts about Full Binary Trees

- You cannot add nodes to a full binary tree without increasing its height.
- The number of nodes that a full binary tree of height  $h$  can have is  $2^{(h+1)} - 1$ .
- The height of a full binary tree with  $n$  nodes is  
$$\log_2(n+1)-1$$
- The height of a complete binary tree with  $n$  nodes is  
$$\text{floor}(\log_2 n)$$

# Complete Binary Trees

Every level, except possibly the last, is completely filled, and all nodes are as far left as possible

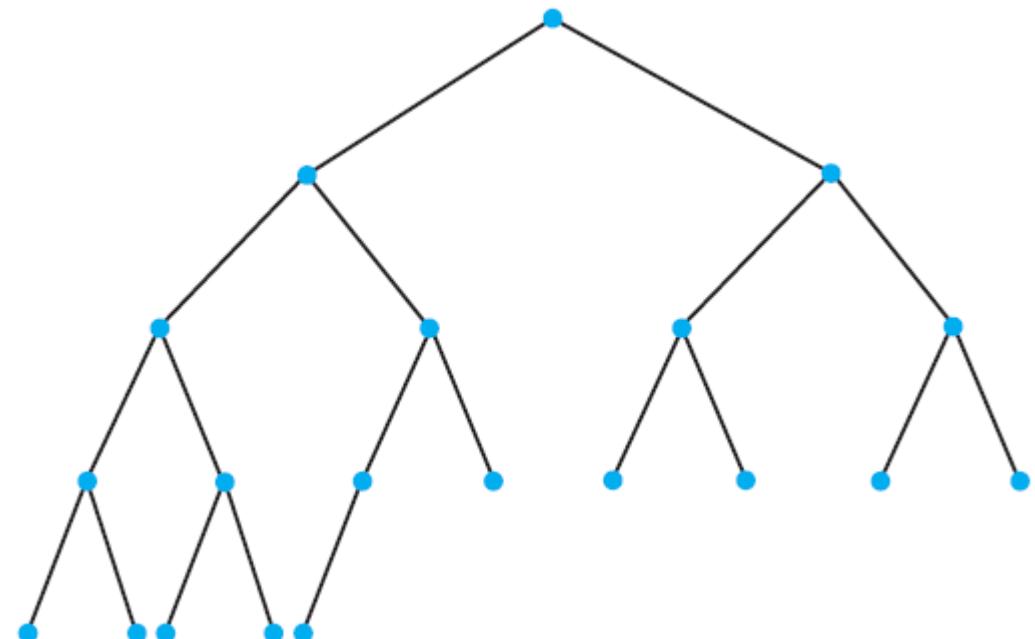
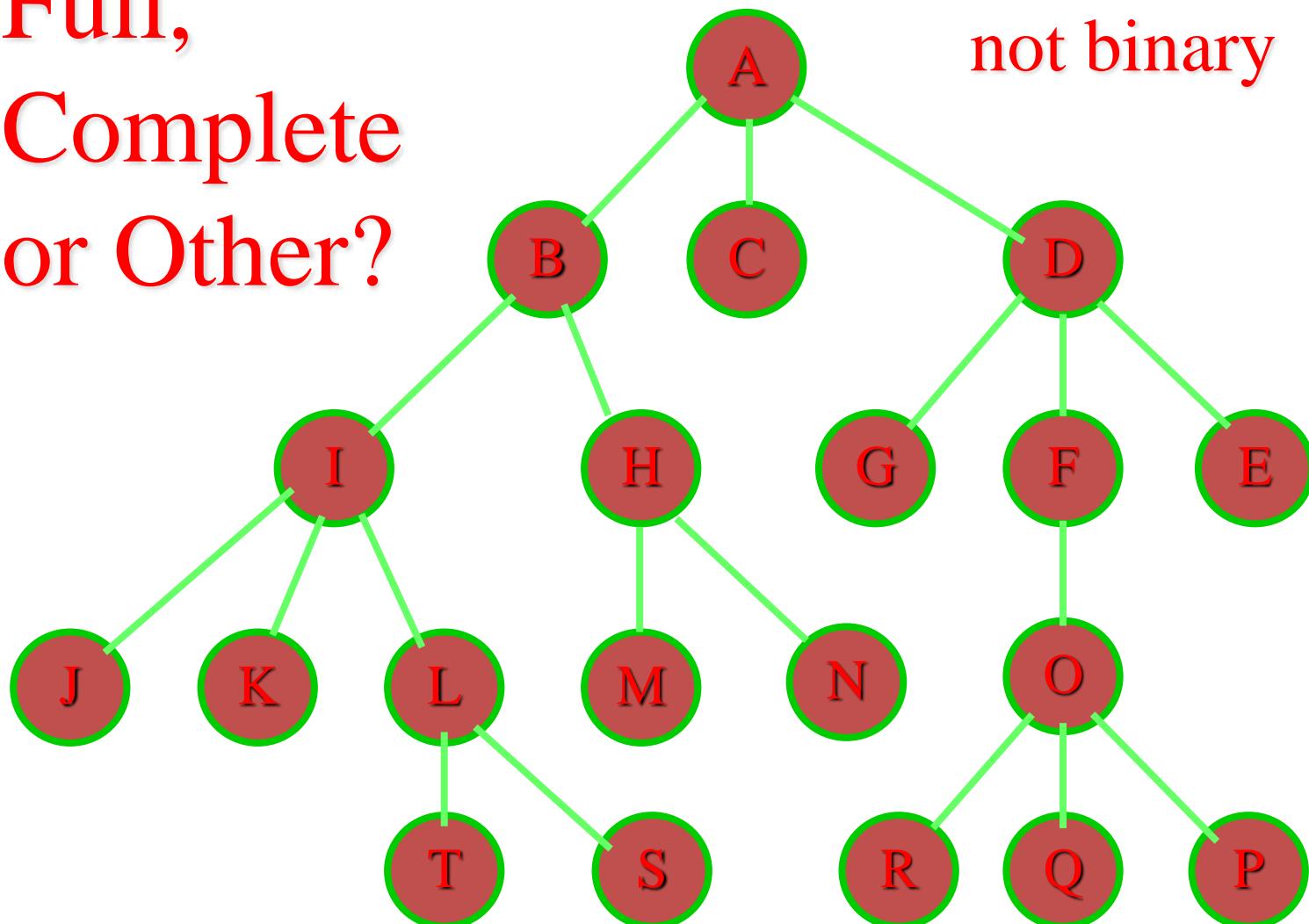


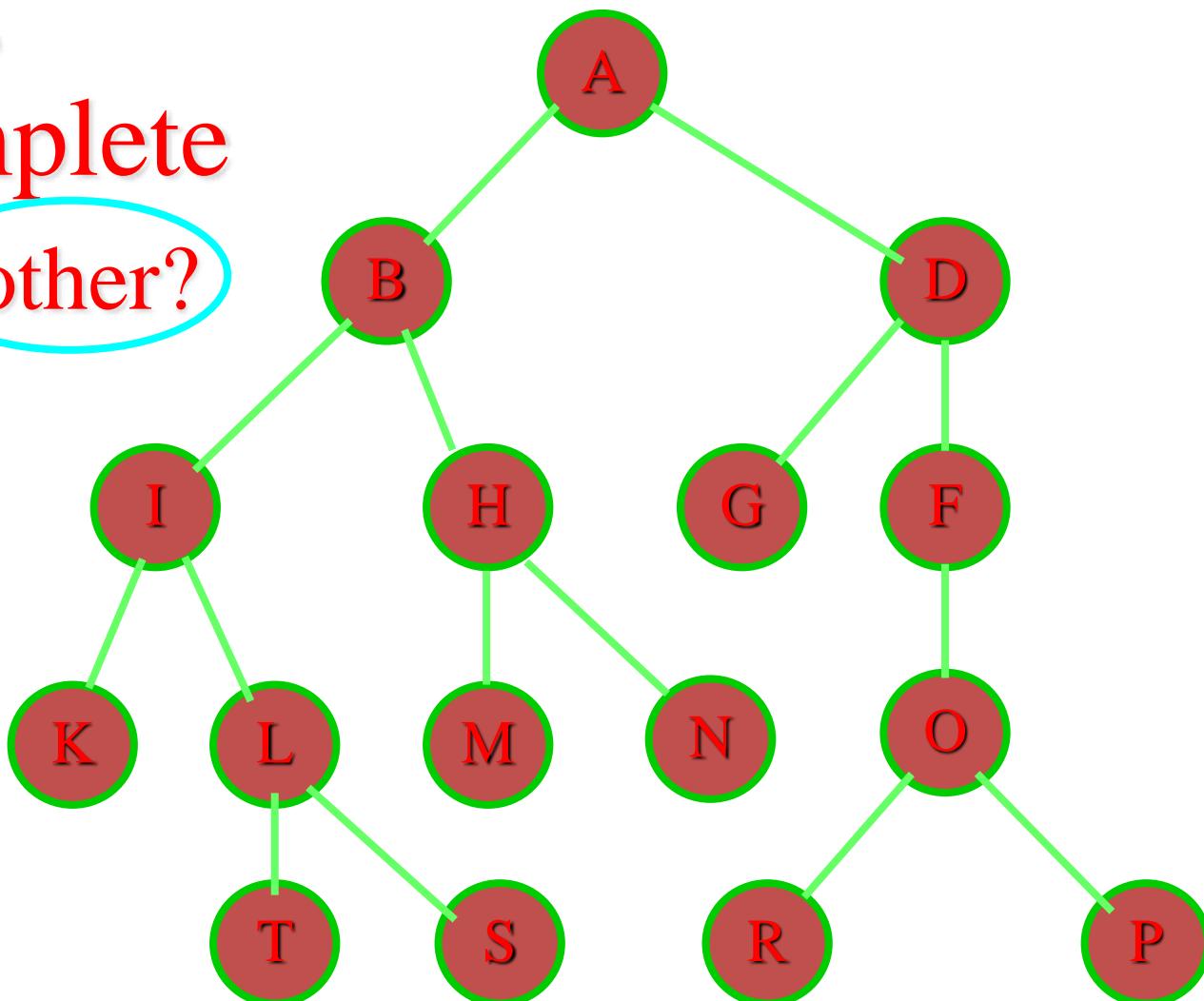
FIGURE 15-7 A complete binary tree

Full,  
Complete  
or Other?

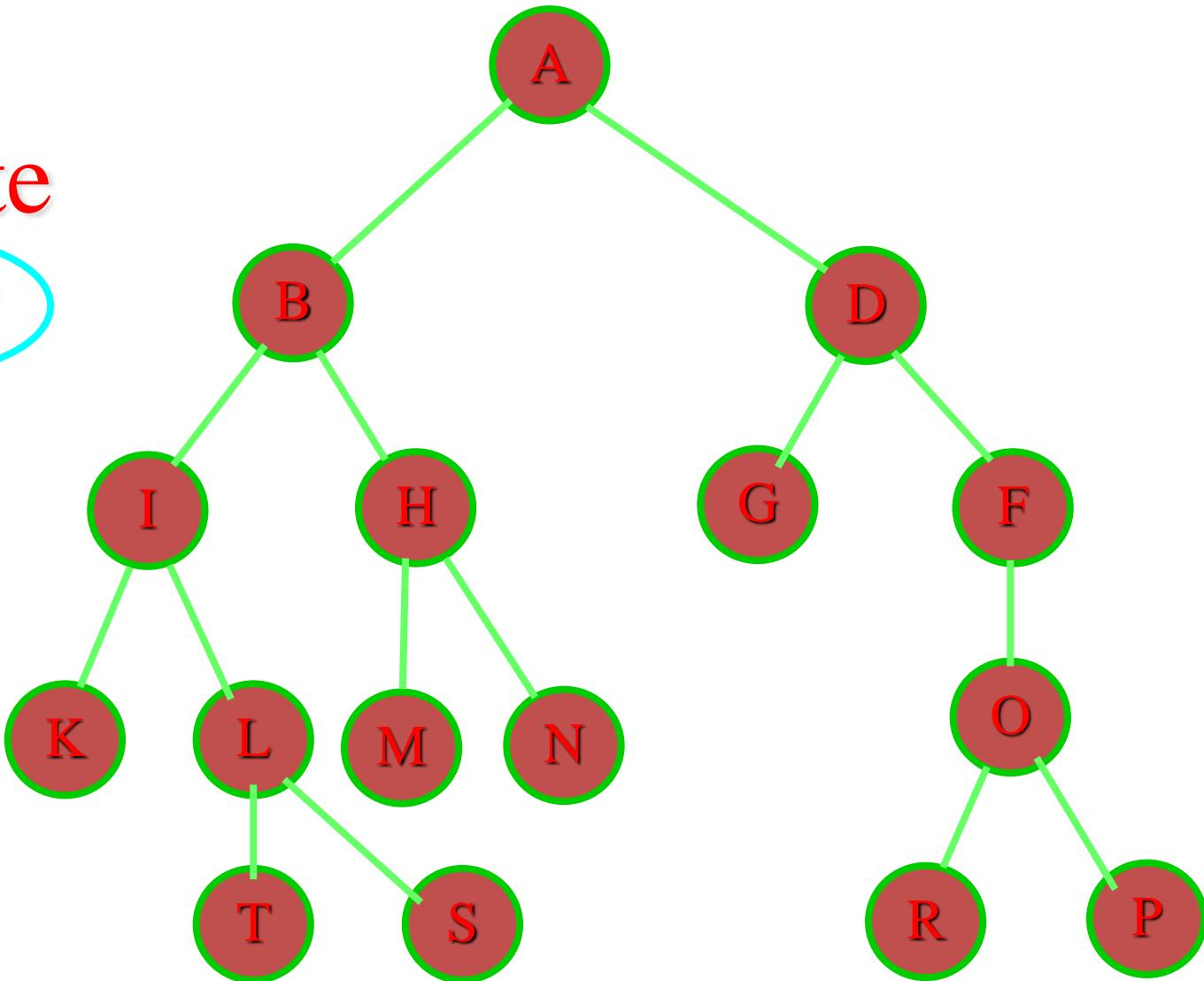
not binary



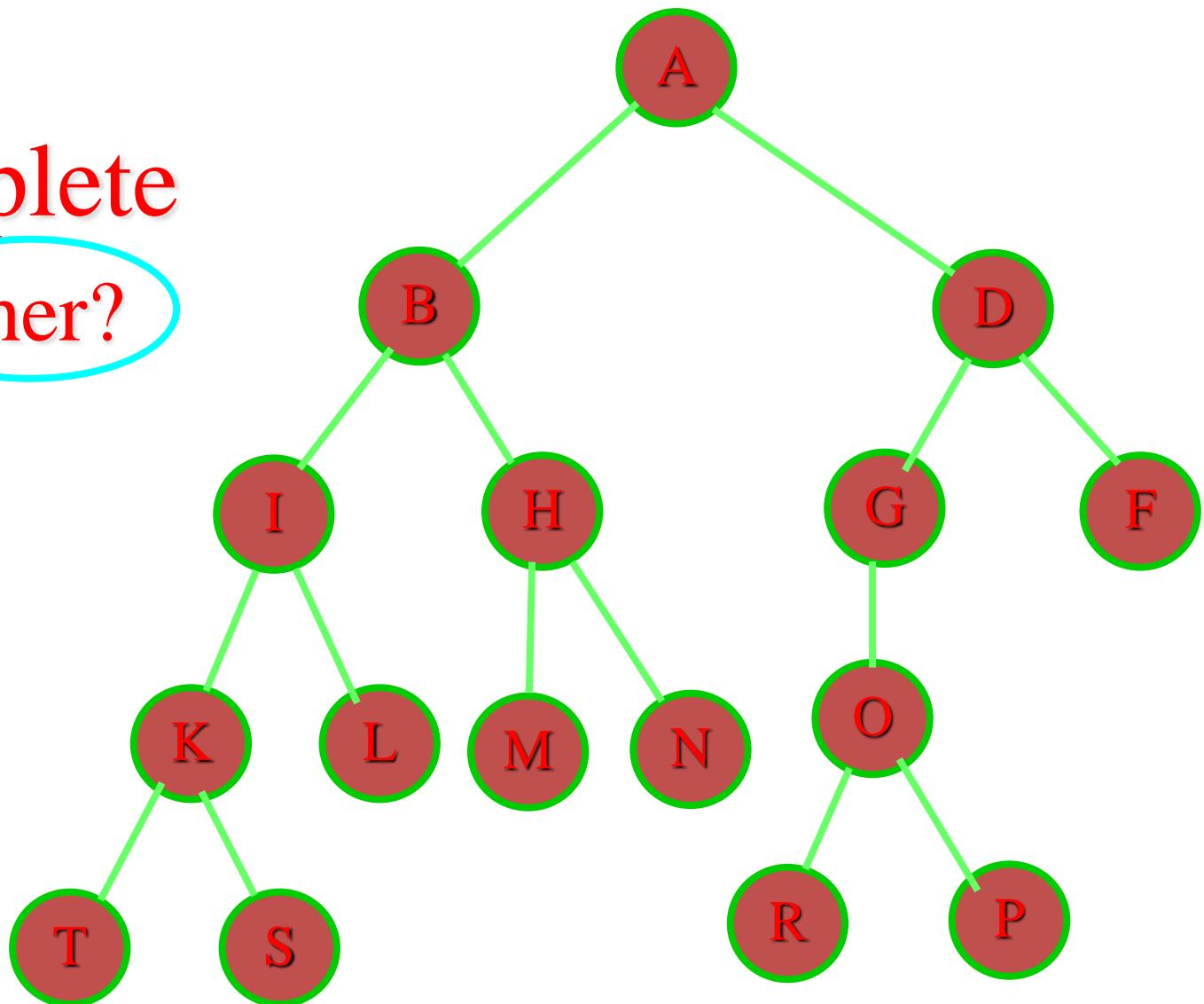
Full,  
Complete  
or other?



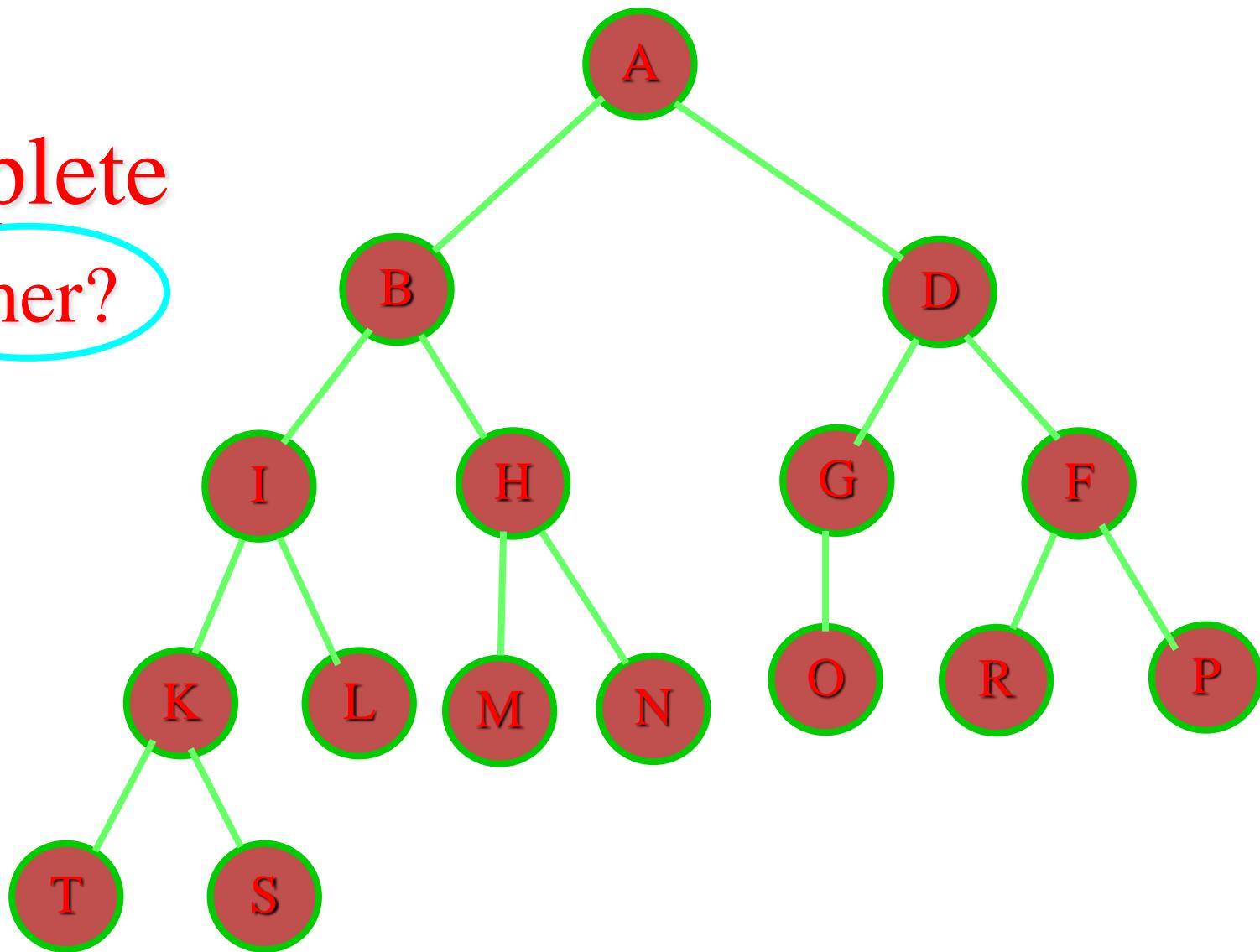
Full,  
Complete  
or other?



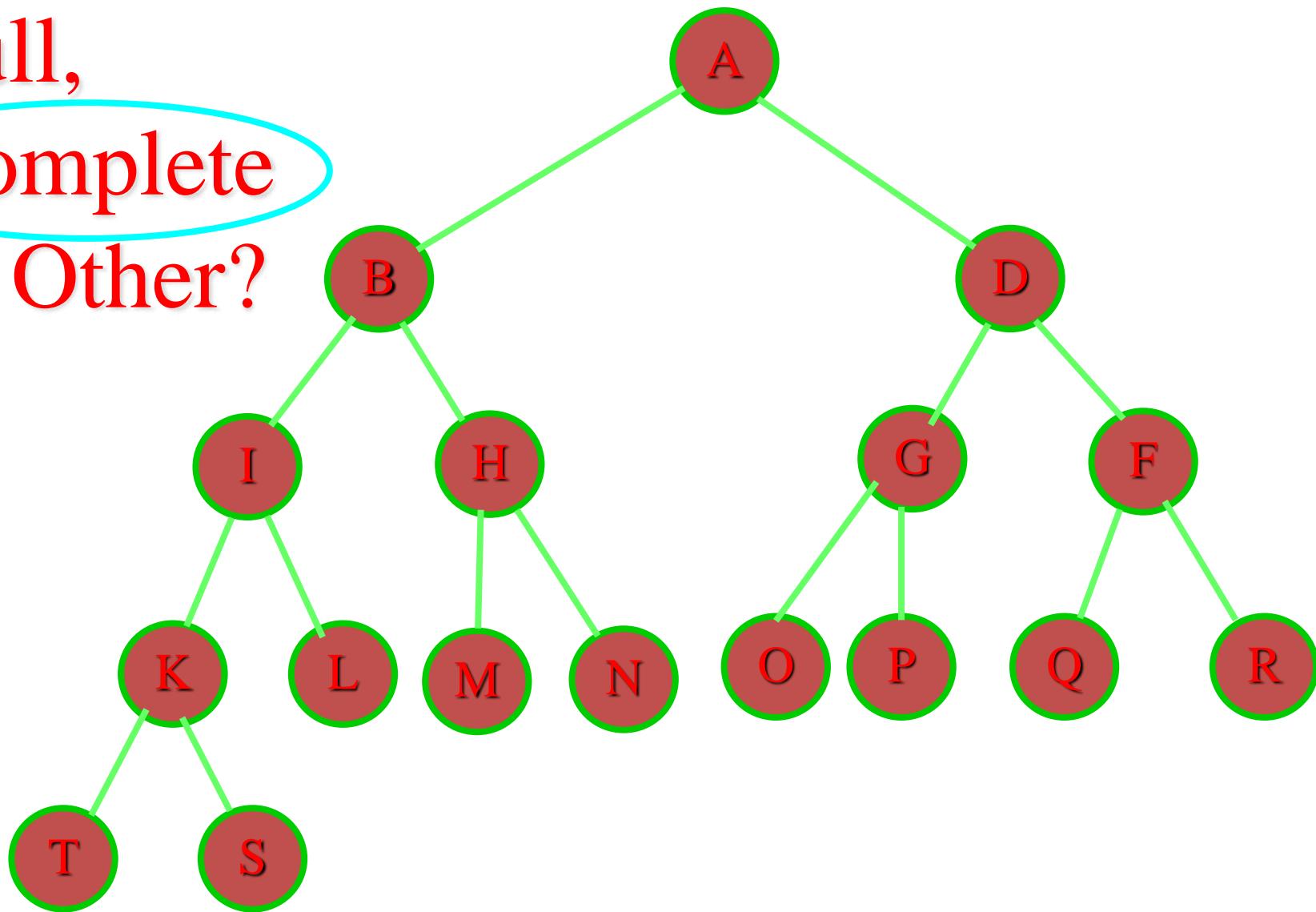
Full,  
Complete  
or other?



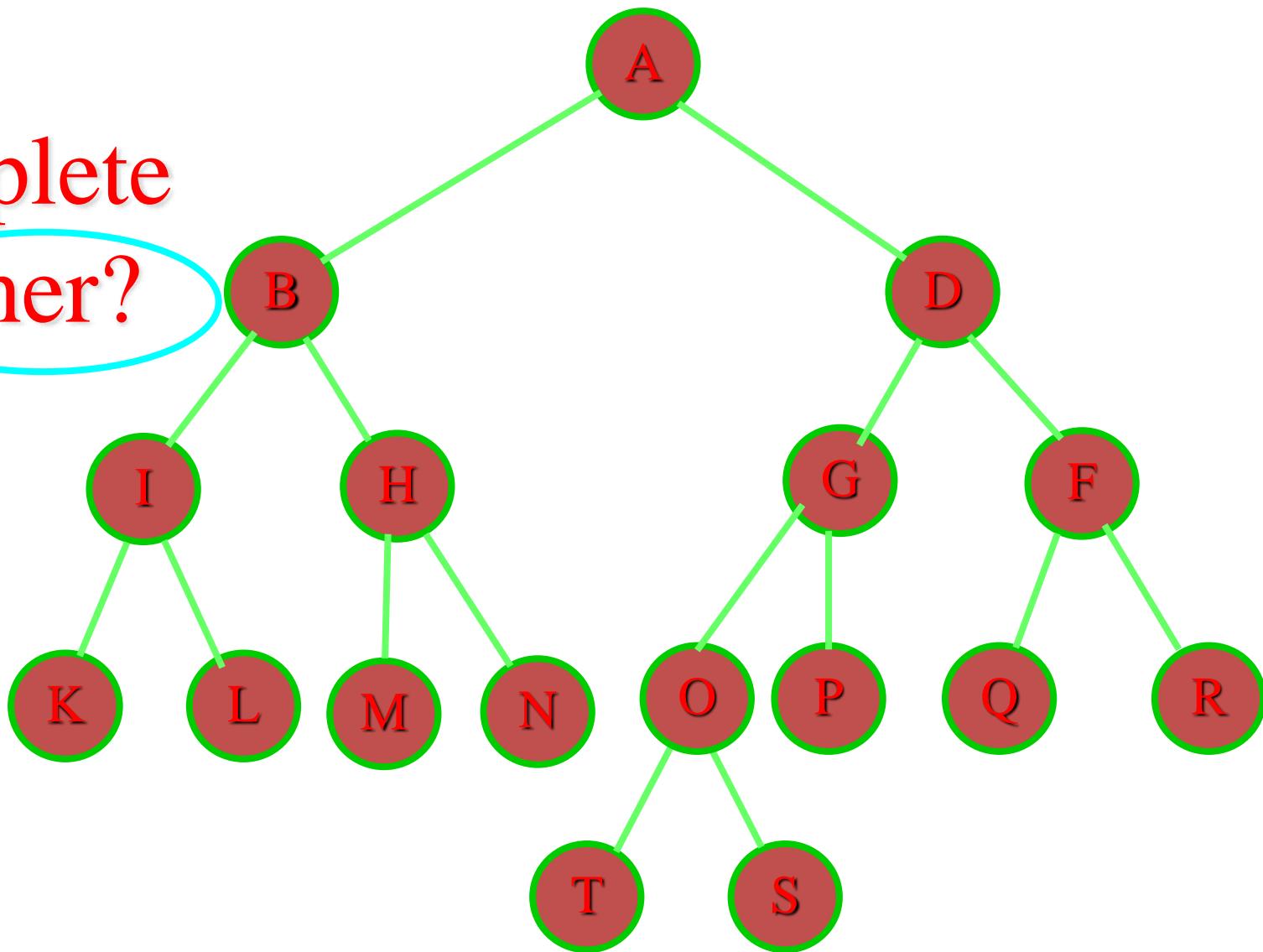
Full,  
Complete  
or other?



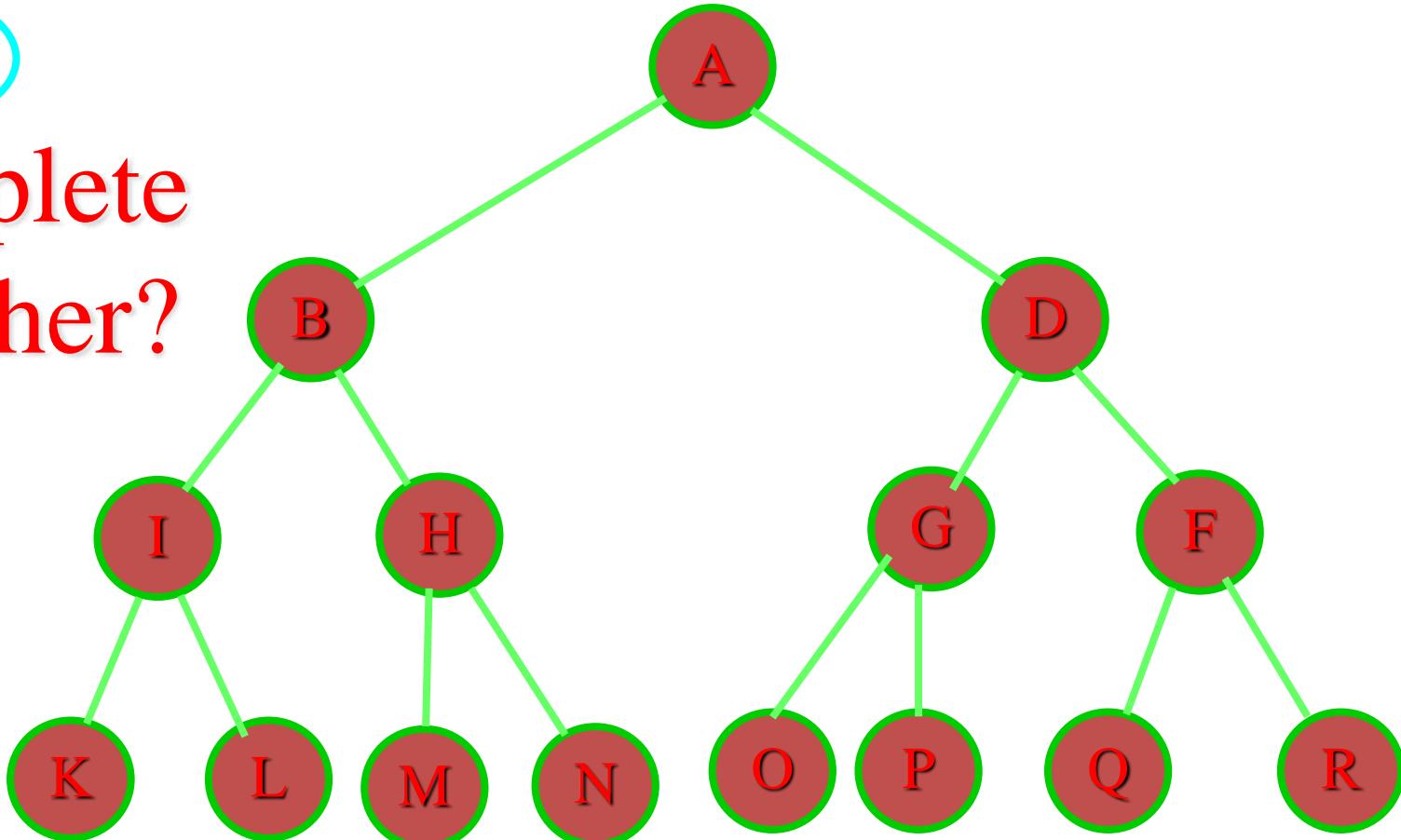
Full,  
Complete  
or Other?



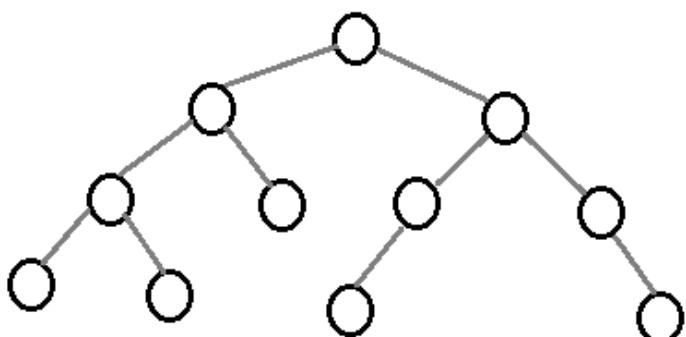
Full,  
Complete  
or other?



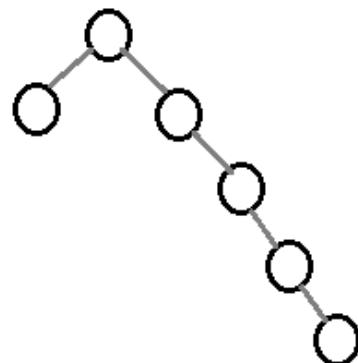
Full,  
Complete  
or Other?



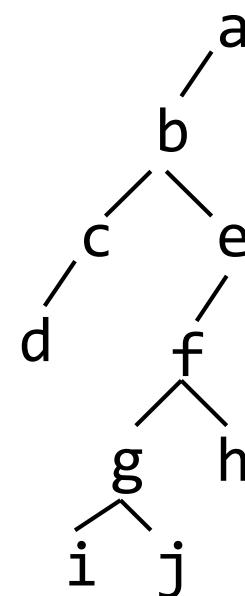
- A balanced binary tree has the minimum possible height for the leaves



Balanced Binary Tree



Unbalanced Binary Tree

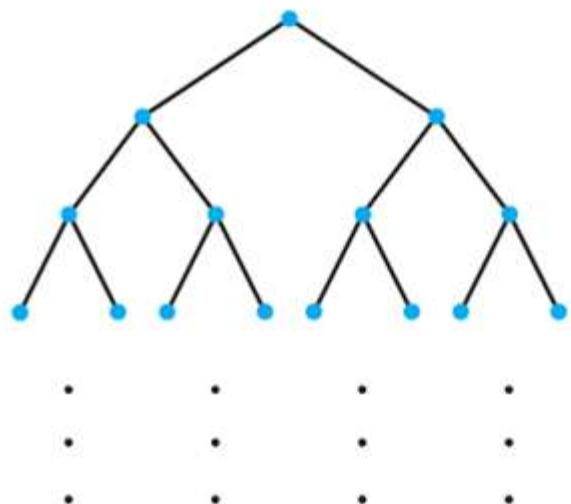


An unbalanced binary tree

# Number of Nodes in a Binary Tree

| Level | Number of nodes at this level | Total number of nodes at this level and all previous levels |
|-------|-------------------------------|---|
| 1     | $1 = 2^0$                     | $1 = 2^1 - 1$   |
| 2     | $2 = 2^1$                     | $3 = 2^2 - 1$   |
| 3     | $4 = 2^2$                     | $7 = 2^3 - 1$   |
| 4     | $8 = 2^3$                     | $15 = 2^4 - 1$  |
| •     | •                             | •   |
| •     | •                             | •   |
| •     | •                             | •   |
| $h$   | $2^h$                         | $2^{h+1} - 1$   |

depth: h                              level: h+1



# Traversals of a Binary Tree

- General form of recursive traversal algorithm

## 1. Preorder Traversal

Each node is processed before any node in either of its subtrees

## 2. Inorder Traversal

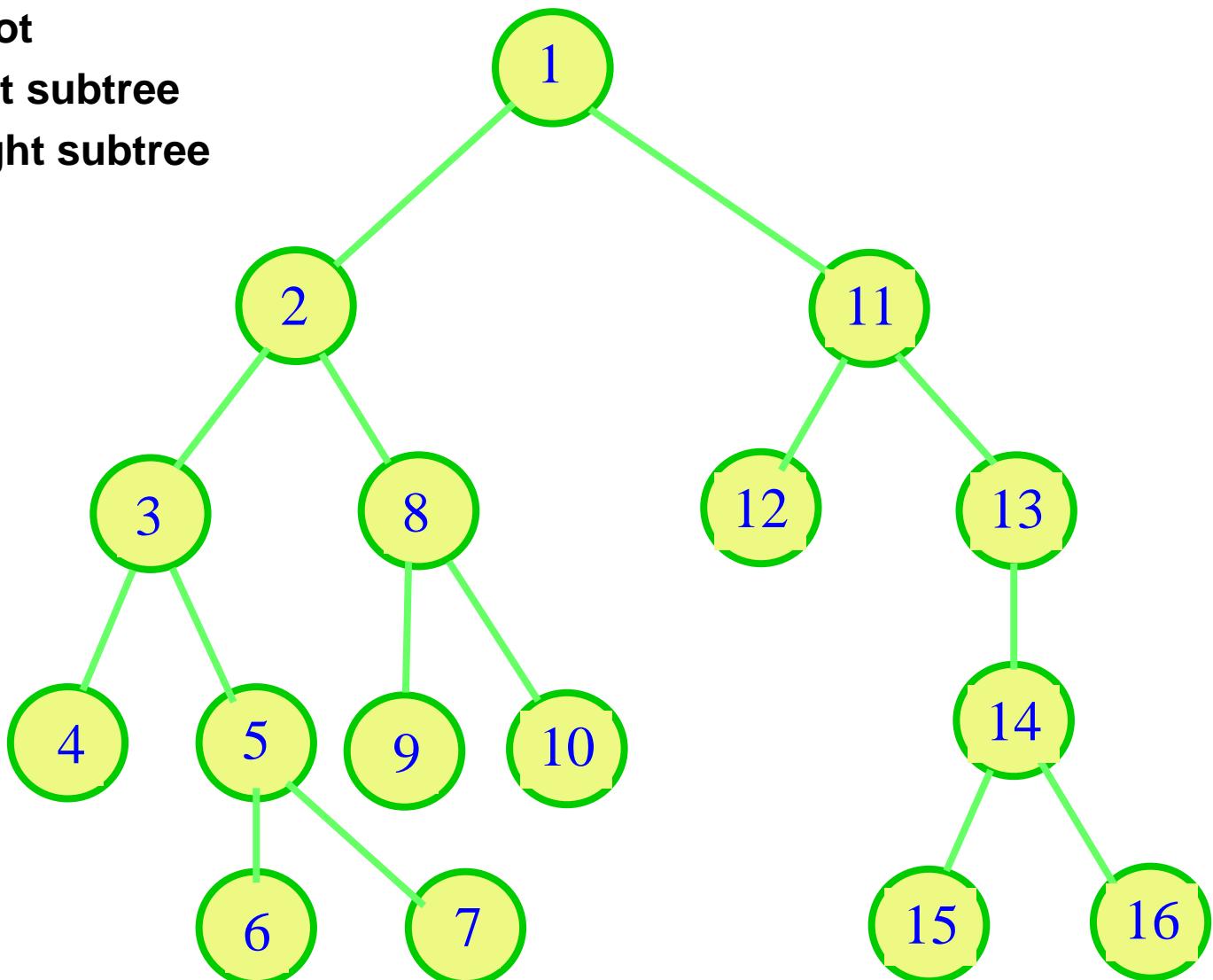
Each node is processed after all nodes in its left subtree and before any node in its right subtree

## 3. Postorder Traversal

Each node is processed after all nodes in both of its subtrees

# Preorder Traversal

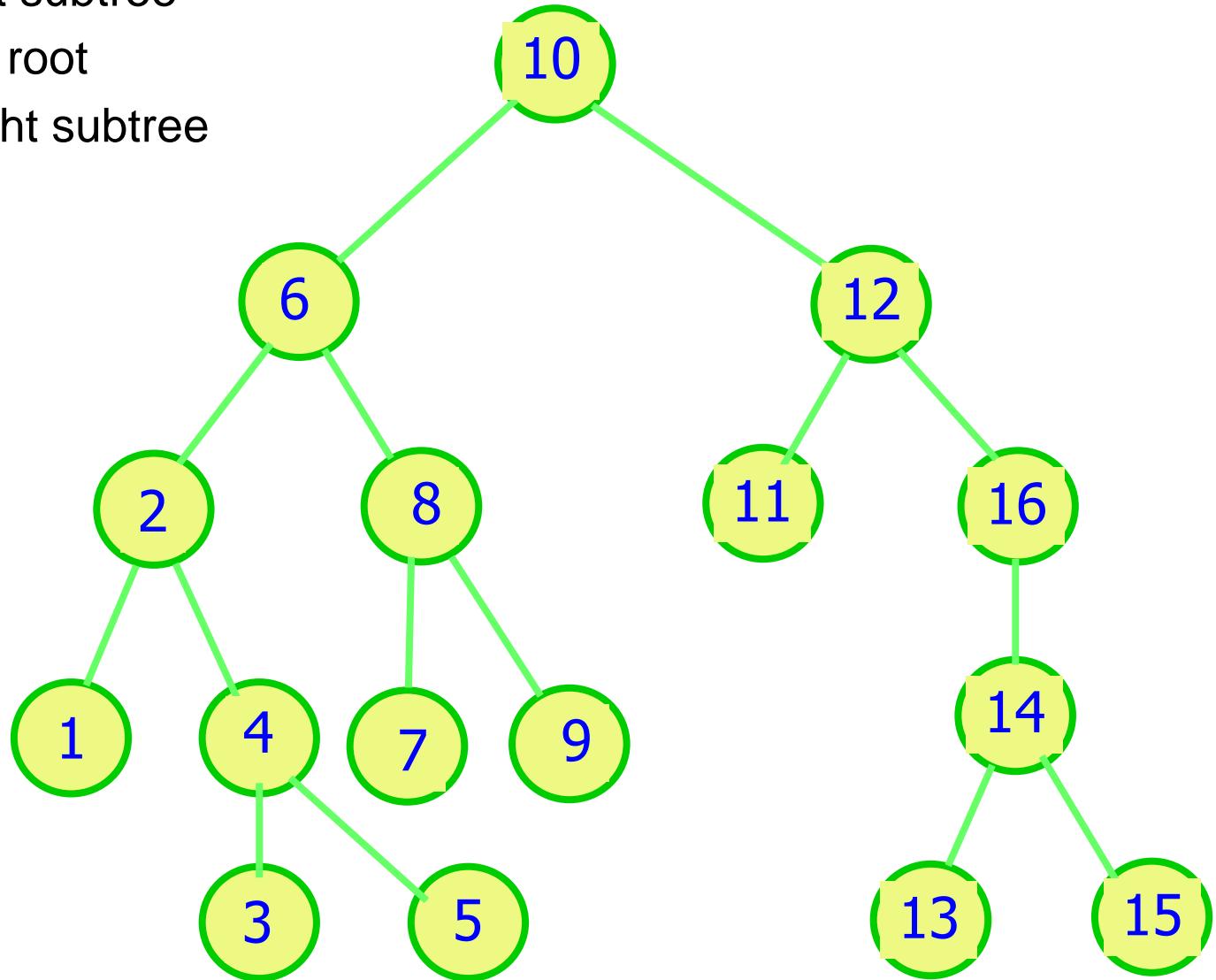
1. Visit the root
2. Visit the left subtree
3. Visit the right subtree



```
Algorithm TraversePreorder(n)
    Process node n
    if n is an internal node then
        TraversePreorder( n -> leftChild)
        TraversePreorder( n -> rightChild)
```

# Inorder Traversal

1. Visit Left subtree
2. Visit the root
3. Visit Right subtree



```
Algorithm TraverseInorder(n)

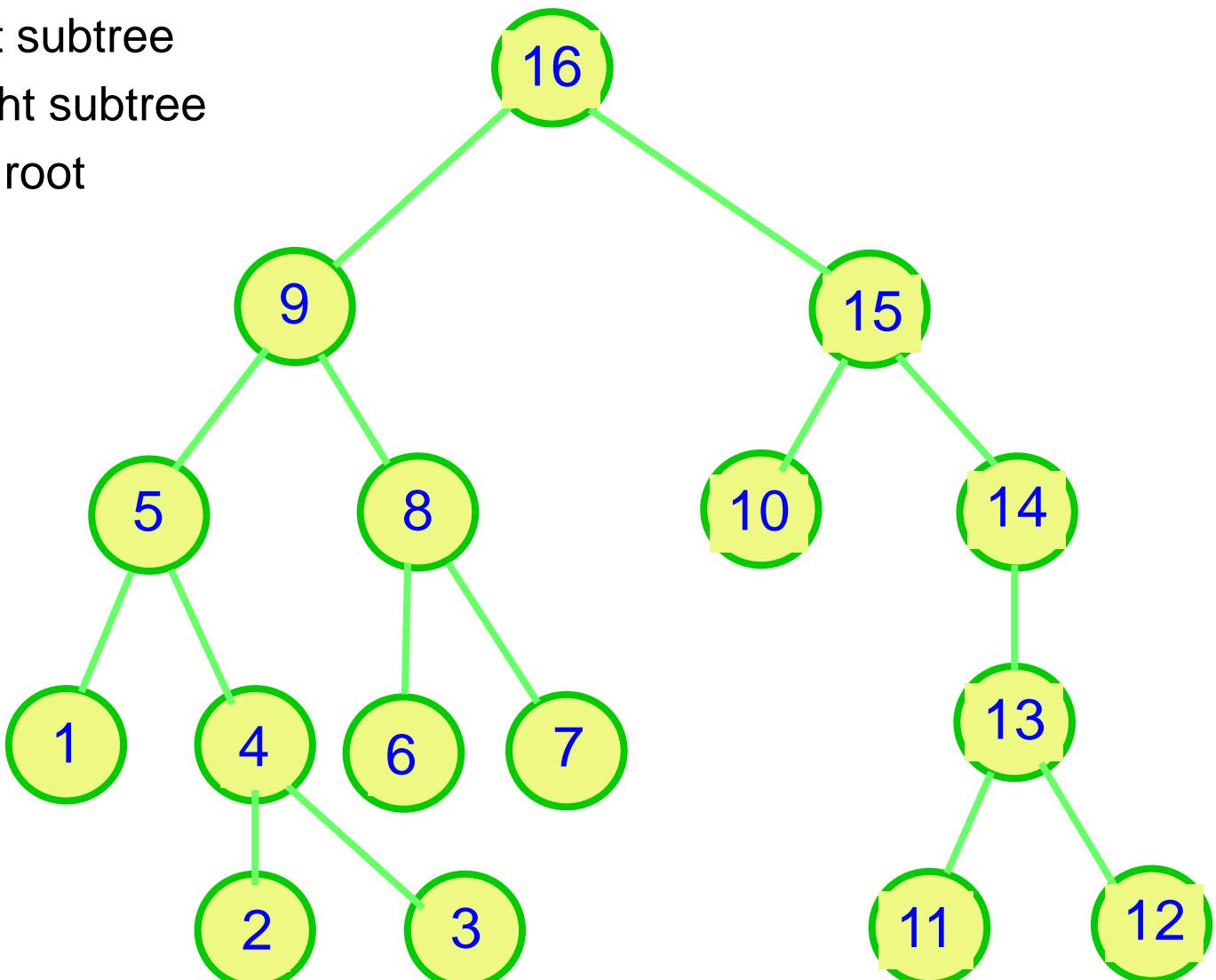
    if n is an internal node then
        TraverseInorder( n -> leftChild)

    Process node n

    if n is an internal node then
        TraverseInorder( n -> rightChild)
```

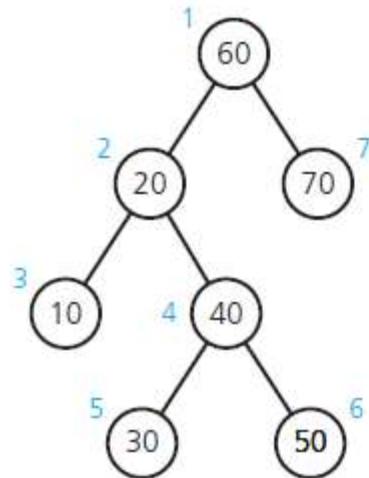
# Postorder Traversals

1. Visit Left subtree
2. Visit Right subtree
3. Visit the root

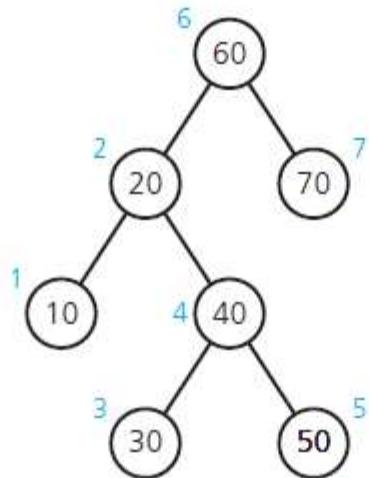


```
Algorithm TraversePostorder(n)
    if n is an internal node then
        TraversePostorder( n -> leftChild)
        TraversePostorder( n -> rightChild)
    Process node n
```

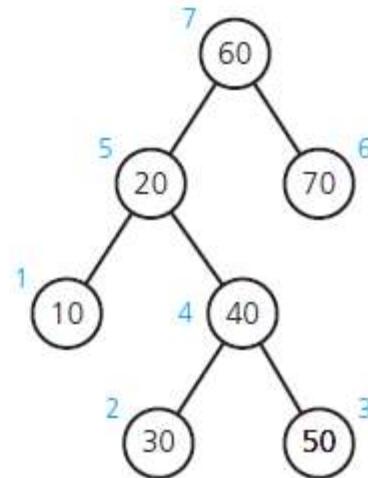
# Traversals of a Binary Tree



(a) Preorder: 60, 20, 10, 40, 30, 50, 70



(b) Inorder: 10, 20, 30, 40, 50, 60, 70

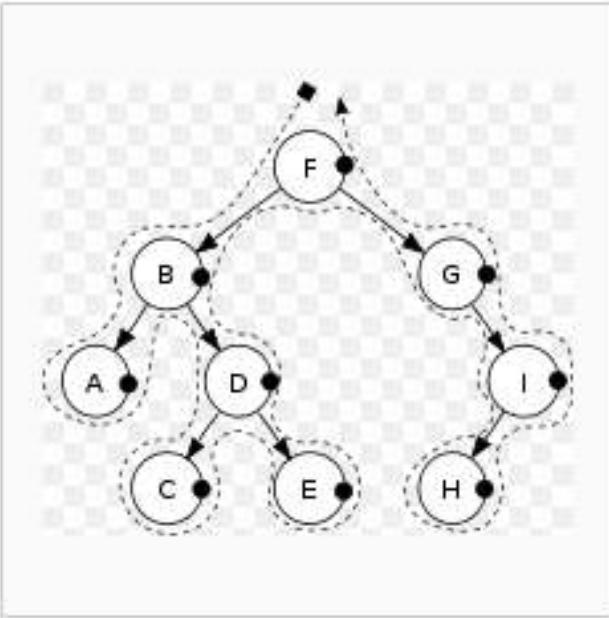
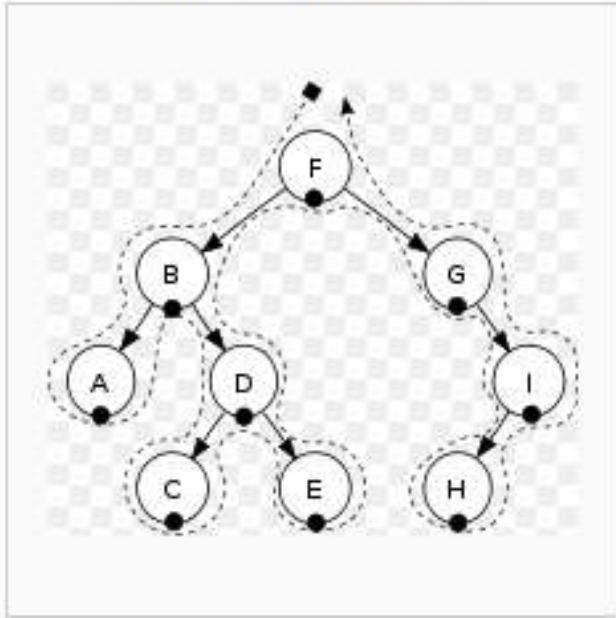
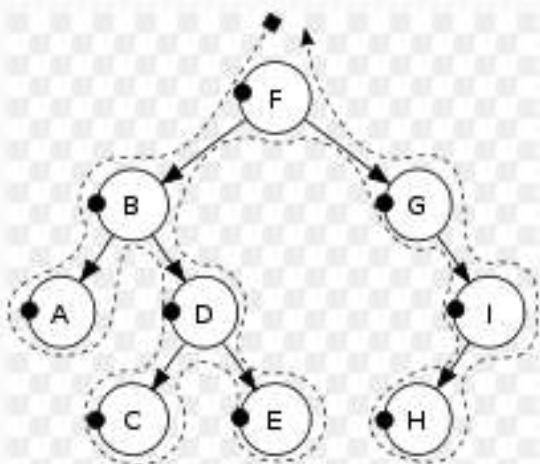


(c) Postorder: 10, 30, 50, 40, 20, 70, 60

(Numbers beside nodes indicate traversal order.)

FIGURE 15-11 Three traversals of a  
binary tree

### The 3 different types of traversal



Pre-order Traversal

FBADCEGIH

In-order Traversal

ABCDEFGHI

Post-order Traversal

ACEDBHIGF

# Binary Tree Operations

- Test whether a binary tree is empty.
- Get the height of a binary tree.
- Get the number of nodes in a binary tree.
- Get the data in a binary tree's root.
- Set the data in a binary tree's root.
- Add a new node containing a given data item to a binary tree.

# Binary Tree Operations

- Remove the node containing a given data item from a binary tree.
- Remove all nodes from a binary tree.
- Retrieve a specific entry in a binary tree.
- Test whether a binary tree contains a specific entry.
- Traverse the nodes in a binary tree in preorder, inorder, or postorder.

# **Representation of Binary Tree ADT**

**A binary tree can be represented using**

- **Linked List**
- **Array**

**Note :** Array is suitable only for full and complete binary trees

```
struct node
{
    int key_value;
    struct node *left;
    struct node *right;
};
```

```
struct node *root = 0;
```

```
void inorder(node *p)
{
    if (p != NULL)
    {
        inorder(p->left);
        printf(p->key_value);
        inorder(p->right);
    }
}
```

```
void preorder(node *p)
{
    if (p != NULL)
    {
        printf(p->key_value);
        preorder(p->left);
        preorder(p->right);
    }
}
```

```
void postorder(node *p)
{
    if (p != NULL)
    {
        postorder(p->left);
        postorder(p->right);
        printf(p->key_value);
    }
}
```

```
void destroy_tree(struct node *leaf)
{
    if( leaf != 0 )
    {
        destroy_tree(leaf->left);
        destroy_tree(leaf->right);
        free( leaf );
    }
}
```

# The Binary Search Tree

- Binary tree is ill suited for searching a specific item
- Binary *search* tree solves the problem
- Properties of each node,  $n$ 
  - $n$ 's value is **greater** than all values in the left subtree  $T_L$
  - $n$ 's value is **less** than all values in the right subtree  $T_R$
  - Both  $T_R$  and  $T_L$  are binary search trees.

# The Binary Search Tree

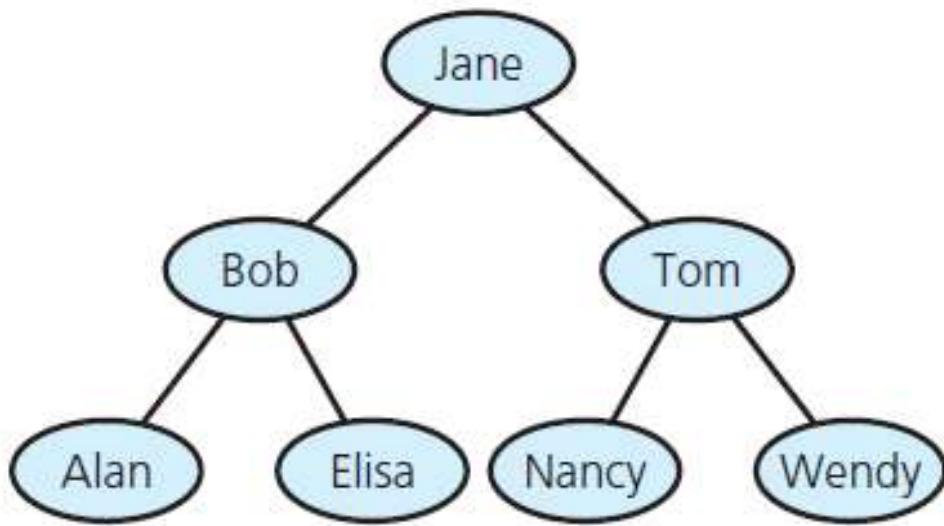


FIGURE 15-13 A binary search tree of  
names

# The Binary Search Tree

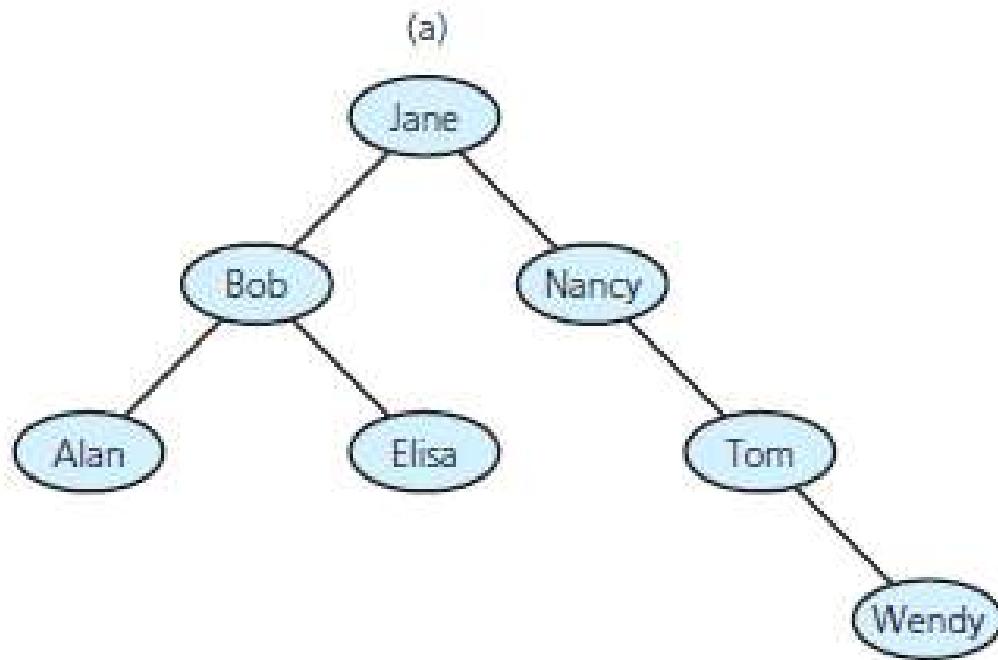


FIGURE 15-14 Binary search trees  
with the same data as in Figure 15-13

# The Binary Search Tree

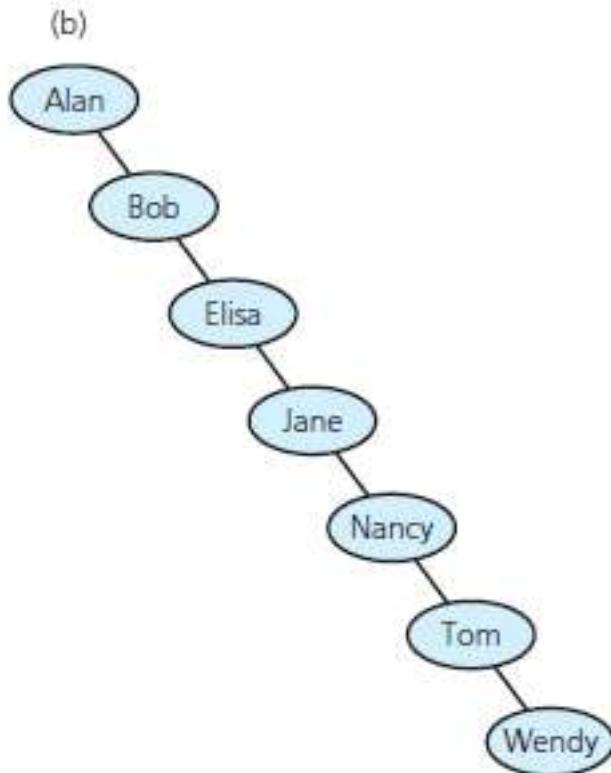
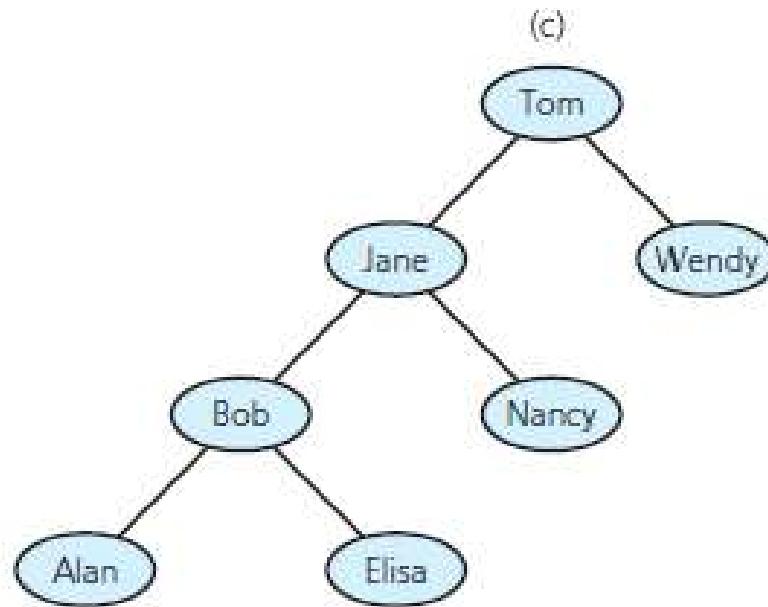


FIGURE 15-14 Binary search trees  
with the same data as in Figure 15-13

# The Binary Search Tree



Binary search trees with the same data as in Figure 15-13

# Binary Search Tree Operations

- Test whether a binary search tree is empty.
- Get height of a binary search tree.
- Get number of nodes in a binary search tree.
- Get data in binary search tree's root.
- Insert new item into the binary search tree.
- Remove given item from the binary search tree.

# Binary Search Tree Operations

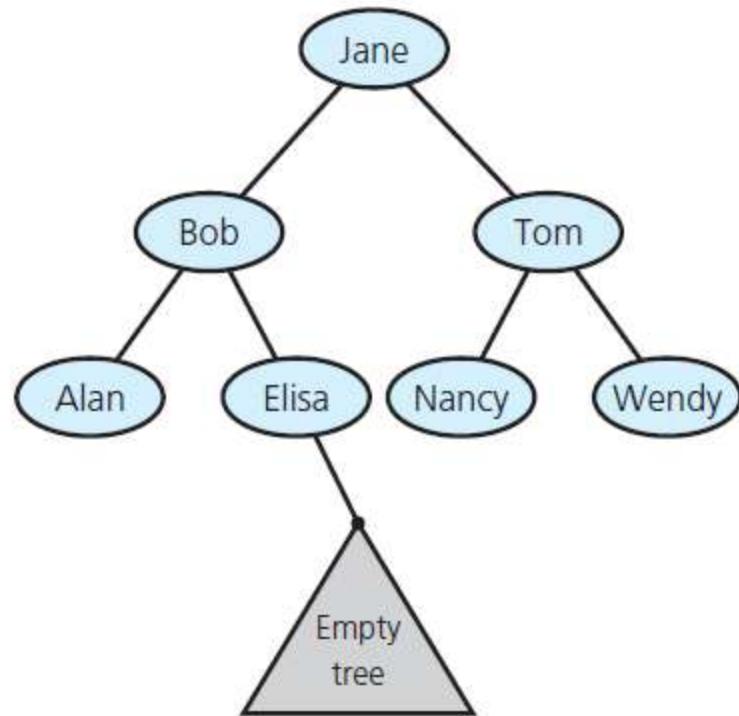
- Remove all entries from a binary search tree.
- Retrieve given item from a binary search tree.
- Test whether a binary search tree contains a specific entry.
- Traverse items in a binary search tree in
  - Preorder
  - Inorder
  - Postorder.

# Searching a Binary Search Tree

- Search algorithm for binary search tree

```
struct node *search(int key, struct node *leaf)
{
    if( leaf != 0 )
    {
        if(key==leaf->key_value)
        {
            return leaf;
        }
        else if(key<leaf->key_value)
        {
            return search(key, leaf->left);
        }
        else
        {
            return search(key, leaf->right);
        }
    }
    else return 0;
}
```

# Creating a Binary Search Tree



Empty subtree where the `search` algorithm terminates when looking for Frank

```
void insert(int key, struct node **leaf)
{
    if( *leaf == 0 )
    {
        *leaf = (struct node*) malloc( sizeof( struct node ) );
        (*leaf)->key_value = key;
        /* initialize the children to null */
        (*leaf)->left = 0;
        (*leaf)->right = 0;
    }
    else if(key < (*leaf)->key_value)
    {
        insert(key, &(*leaf)->left );
    }
    else if(key > (*leaf)->key_value)
    {
        insert(key, &(*leaf)->right );
    }
}
```

# Efficiency of Binary Search Tree Operations

| <u>Operation</u> | <u>Average case</u> | <u>Worst case</u> |
|------------------|---------------------|-------------------|
| Retrieval        | $O(\log n)$         | $O(n)$            |
| Insertion        | $O(\log n)$         | $O(n)$            |
| Removal          | $O(\log n)$         | $O(n)$            |
| Traversal        | $O(n)$              | $O(n)$            |

The Big O for the retrieval, insertion, removal, and traversal operations of the ADT binary search tree

# BBM 201 – DATA STRUCTURES



HACETTEPE UNIVERSITY

**DEPT. OF COMPUTER ENGINEERING**

**TRIES**

**Acknowledgement:** The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.

# TODAY

- ▶ Tries
- ▶ R-way tries

# TRIES

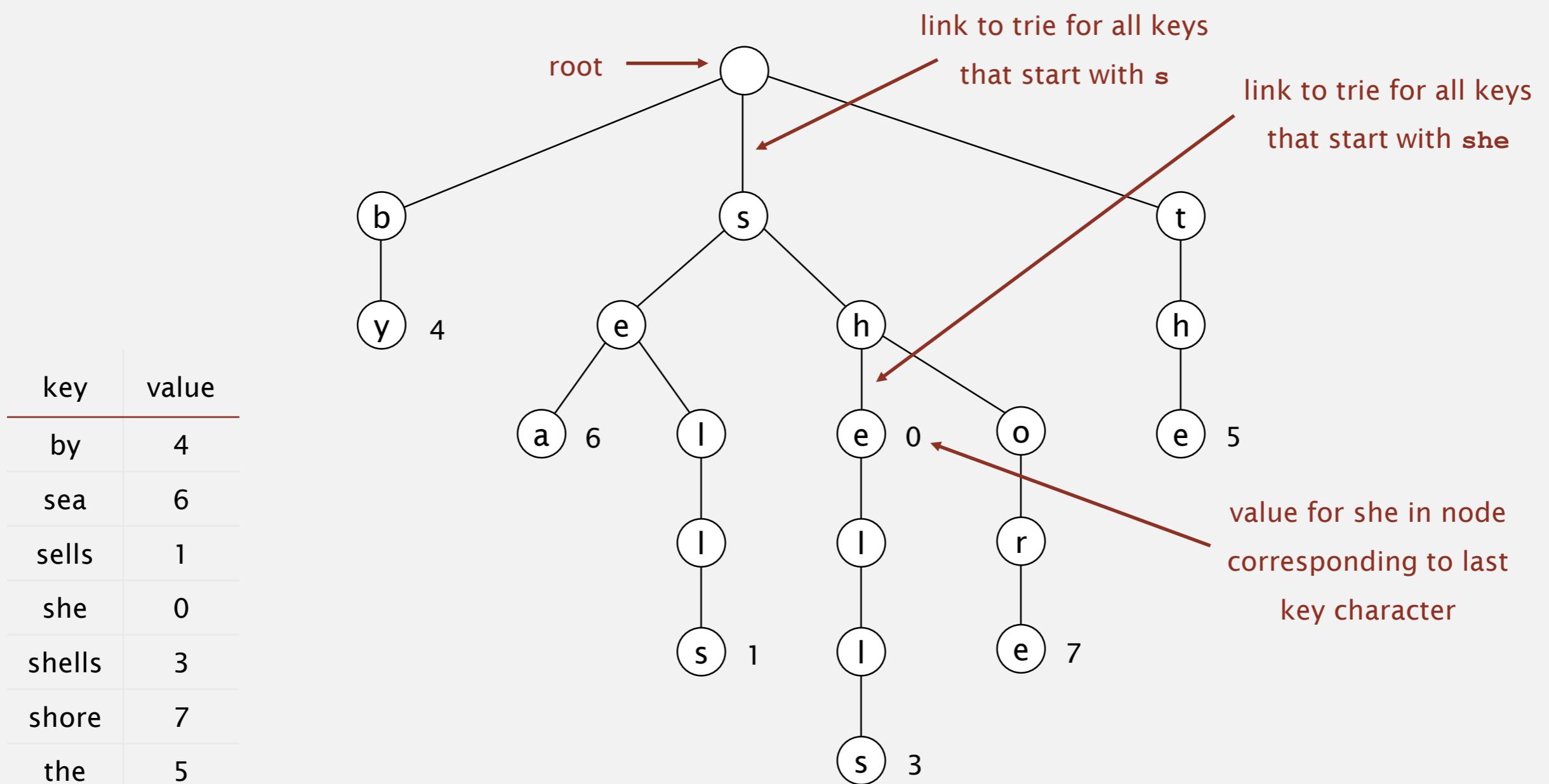
- ▶ **R-way tries**

# Tries

**Tries.** [from retrieval, but pronounced "try"]

- Store characters in nodes (not keys).
- Each node has  $R$  children, one for each possible character.
- Store values in nodes corresponding to last characters in keys.

for now, we do not  
draw null links

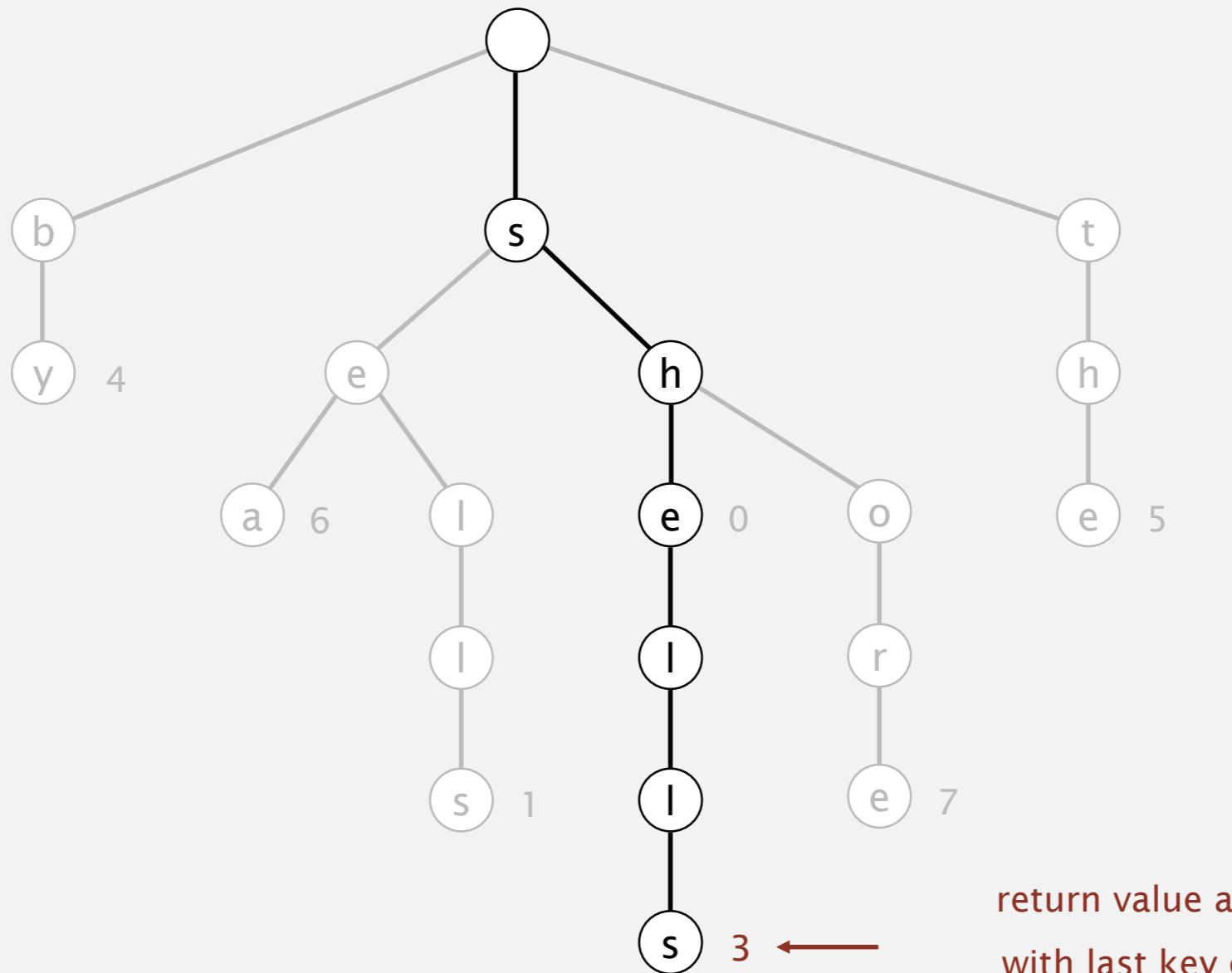


# Search in a trie

Follow links corresponding to each character in the key.

- **Search hit:** node where search ends has a non-null value.
- Search miss: reach a null link or node where search ends has null value.

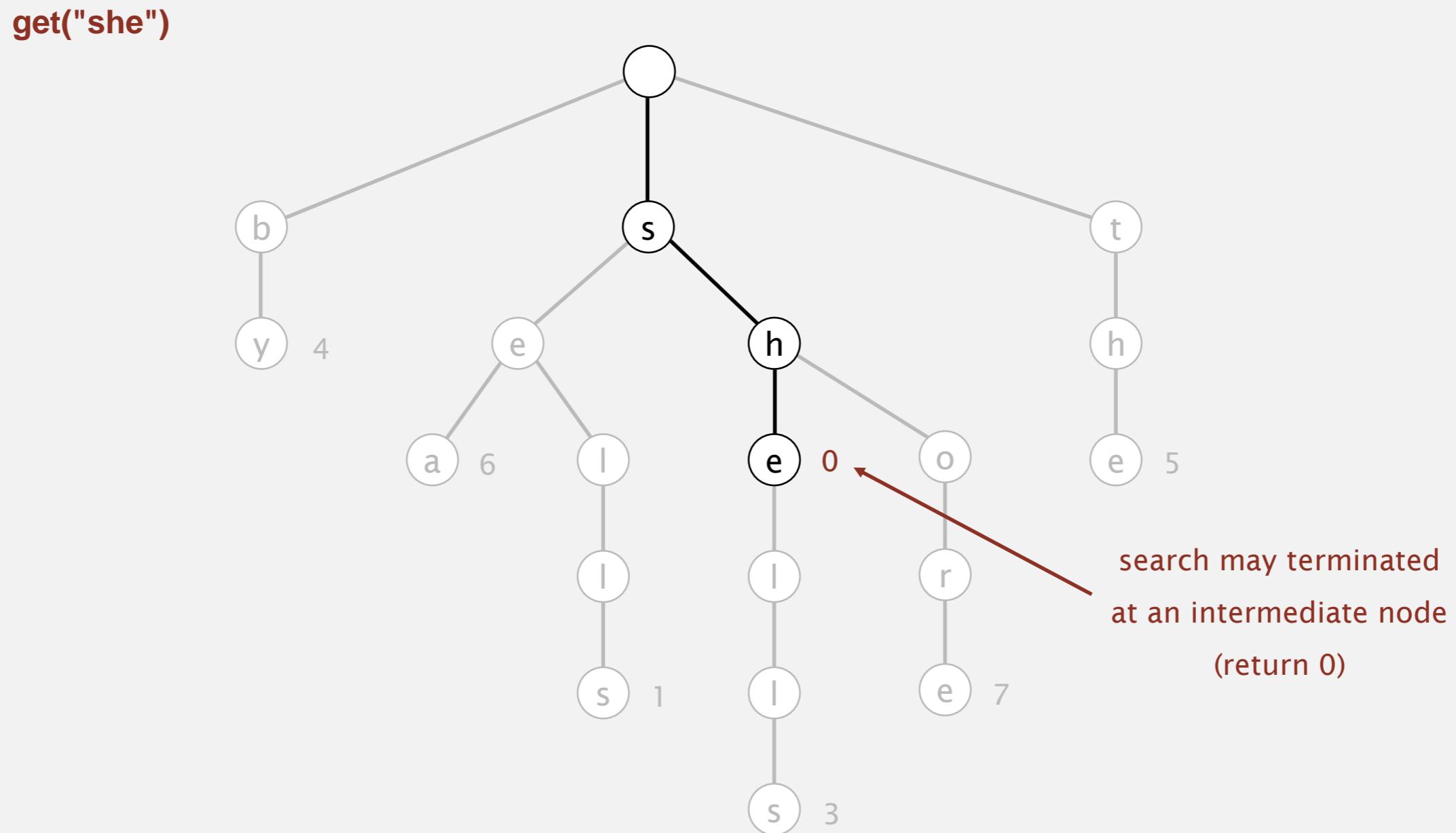
`get("shells")`



# Search in a trie

Follow links corresponding to each character in the key.

- **Search hit:** node where search ends has a non-null value.
- Search miss: reach a null link or node where search ends has null value.

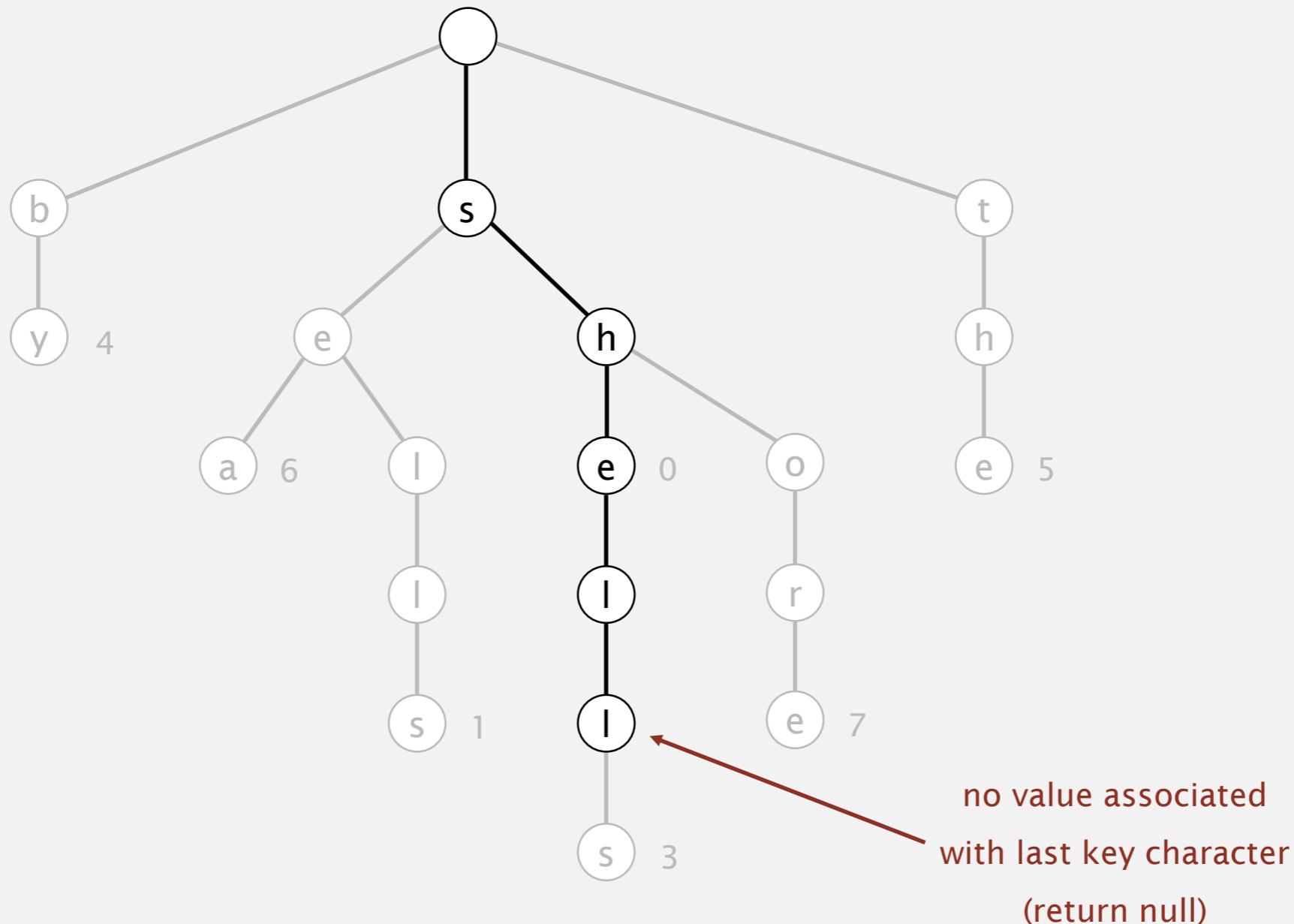


# Search in a trie

Follow links corresponding to each character in the key.

- Search hit: node where search ends has a non-null value.
- Search miss: reach a null link or node where search ends has null value.

`get("shell")`

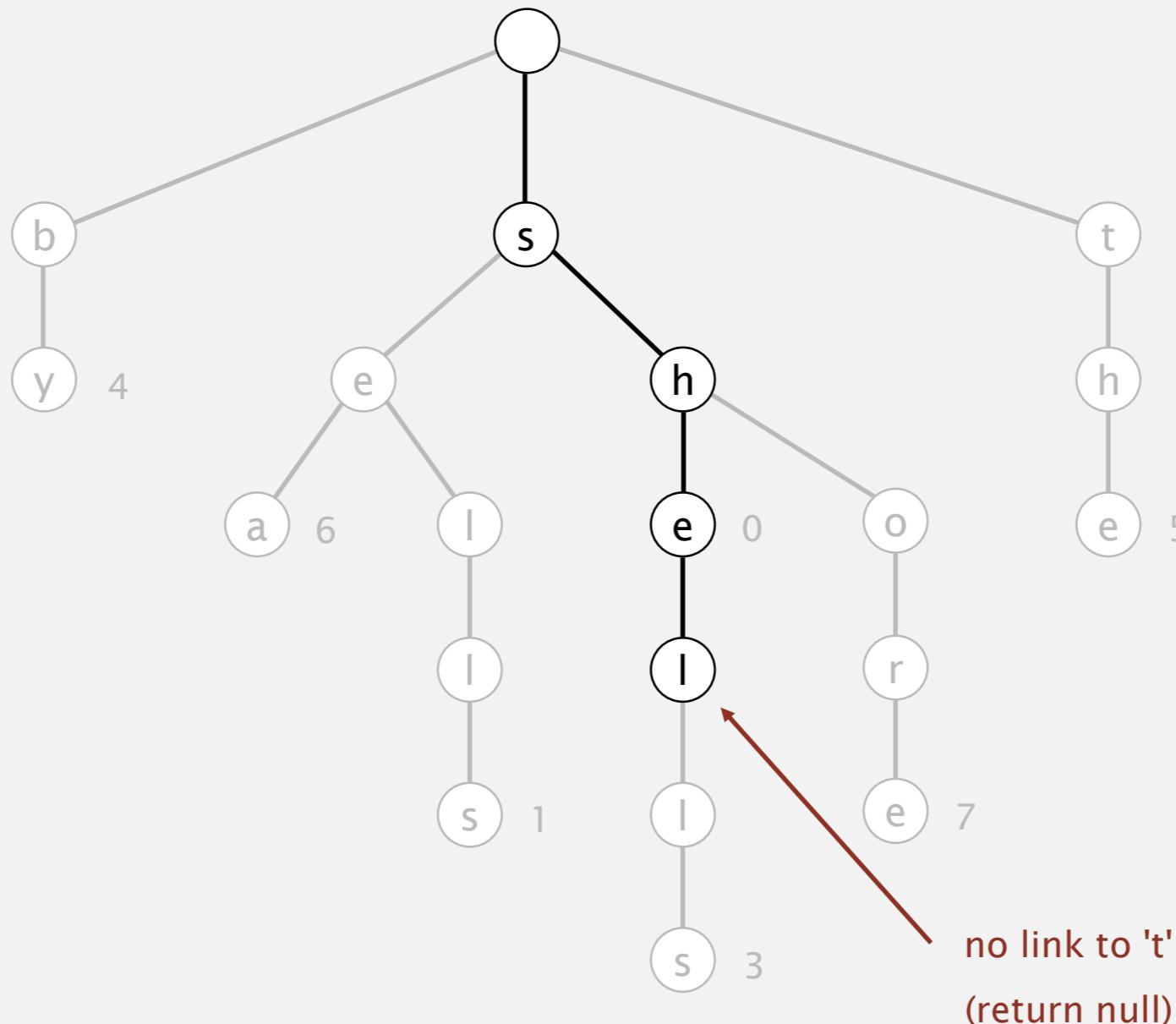


# Search in a trie

Follow links corresponding to each character in the key.

- Search hit: node where search ends has a non-null value.
- Search miss: reach a null link or node where search ends has null value.

get("shelter")

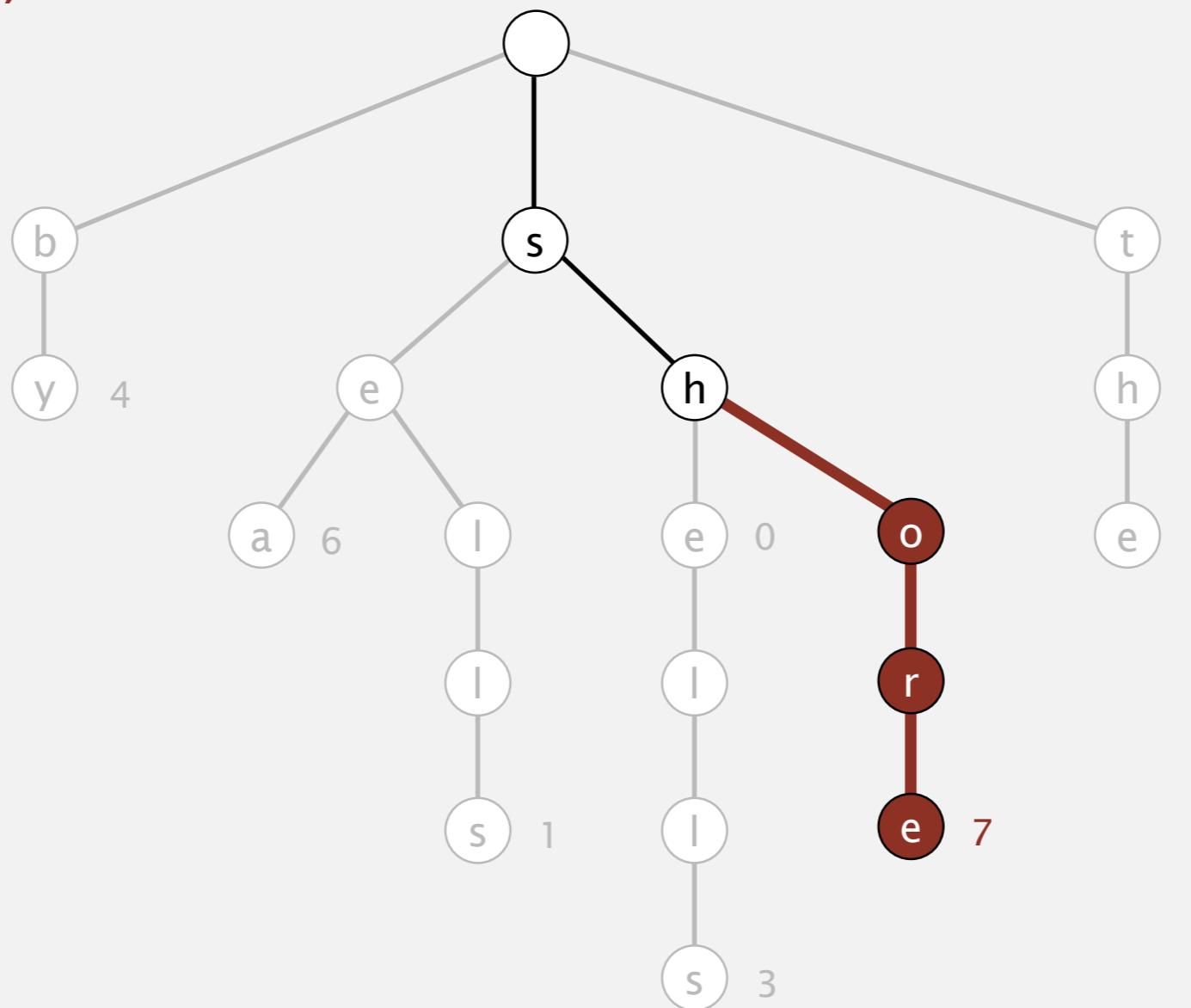


# Insertion into a trie

Follow links corresponding to each character in the key.

- Encounter a null link: create new node.
- Encounter the last character of the key: set value in that node.

`put("shore", 7)`



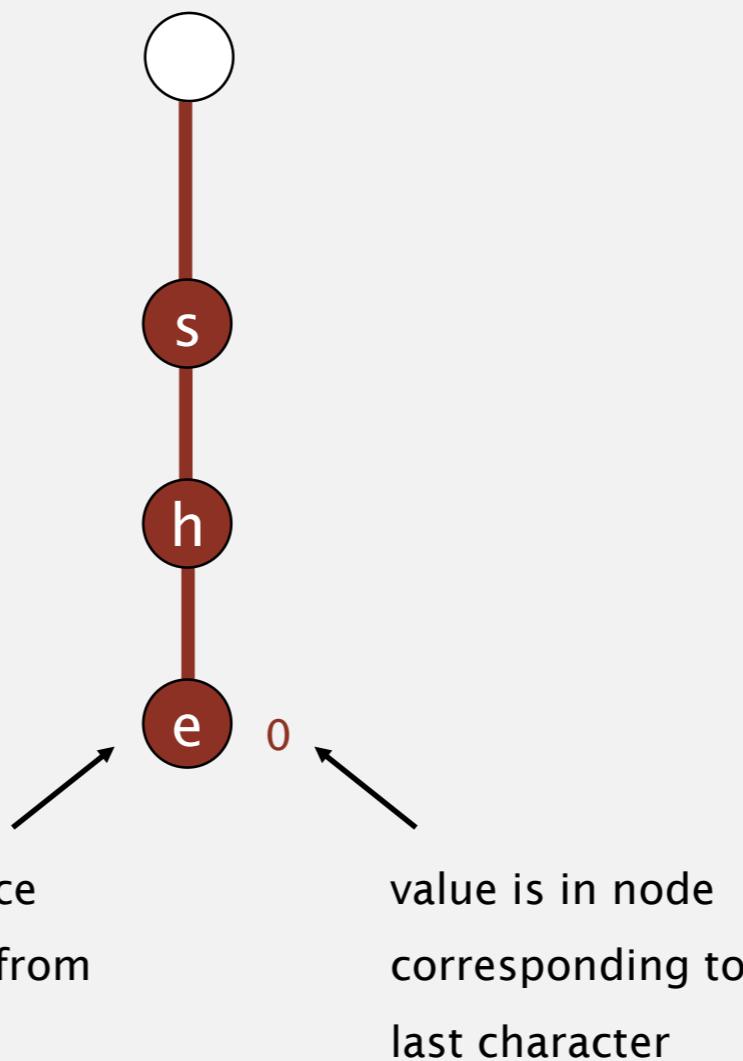
# Trie construction demo

trie



# Trie construction demo

`put("she", 0)`



# Trie construction demo

she

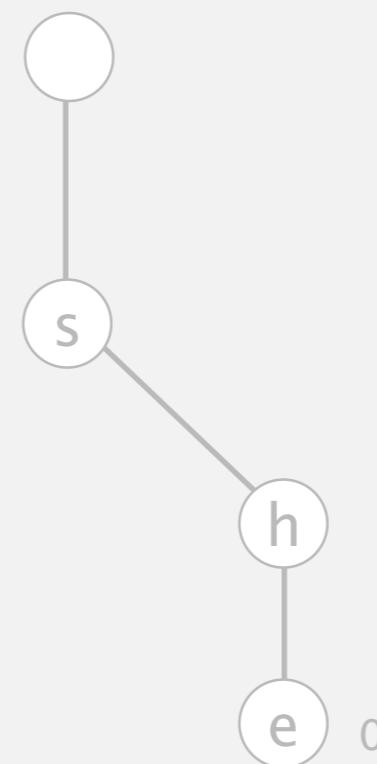
trie



# Trie construction demo

she

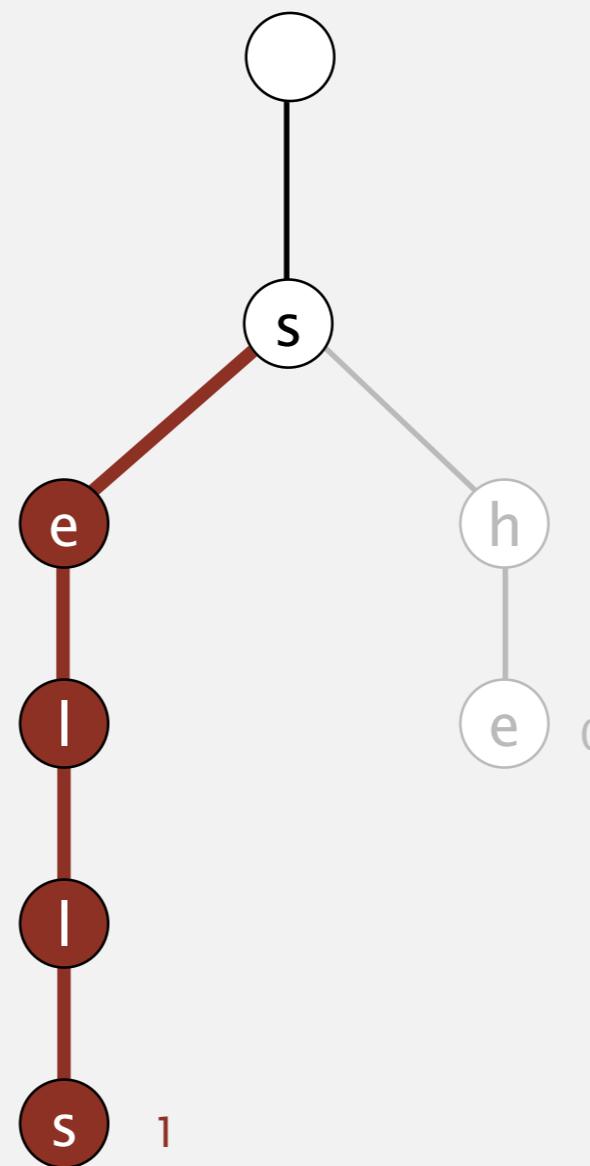
trie



# Trie construction demo

she

**put("sells", 1)**

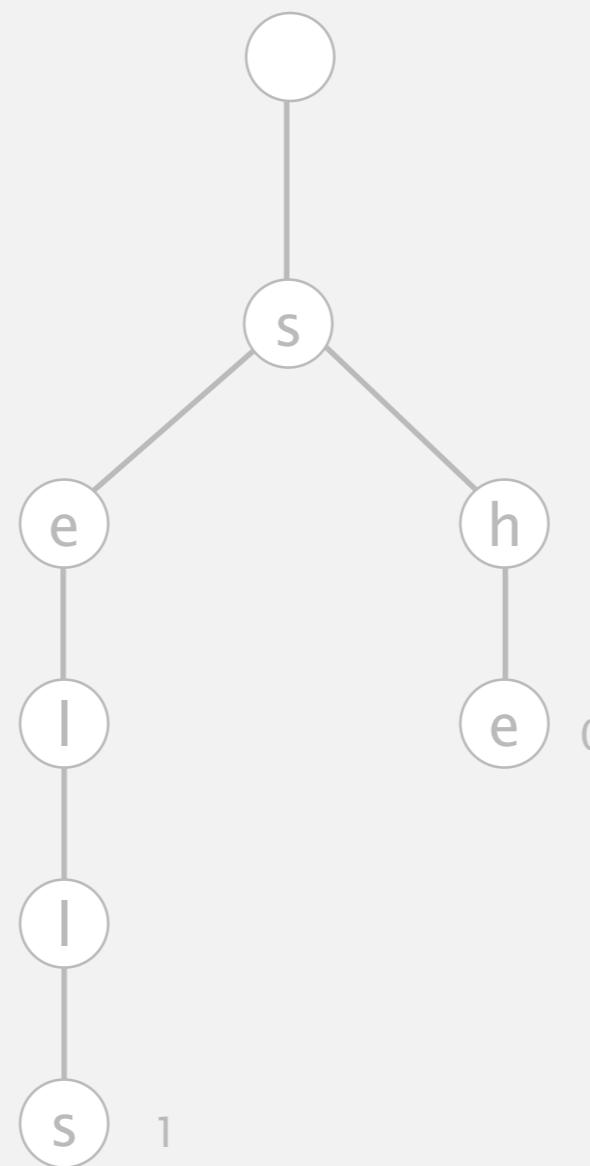


# Trie construction demo

she

sells

**trie**

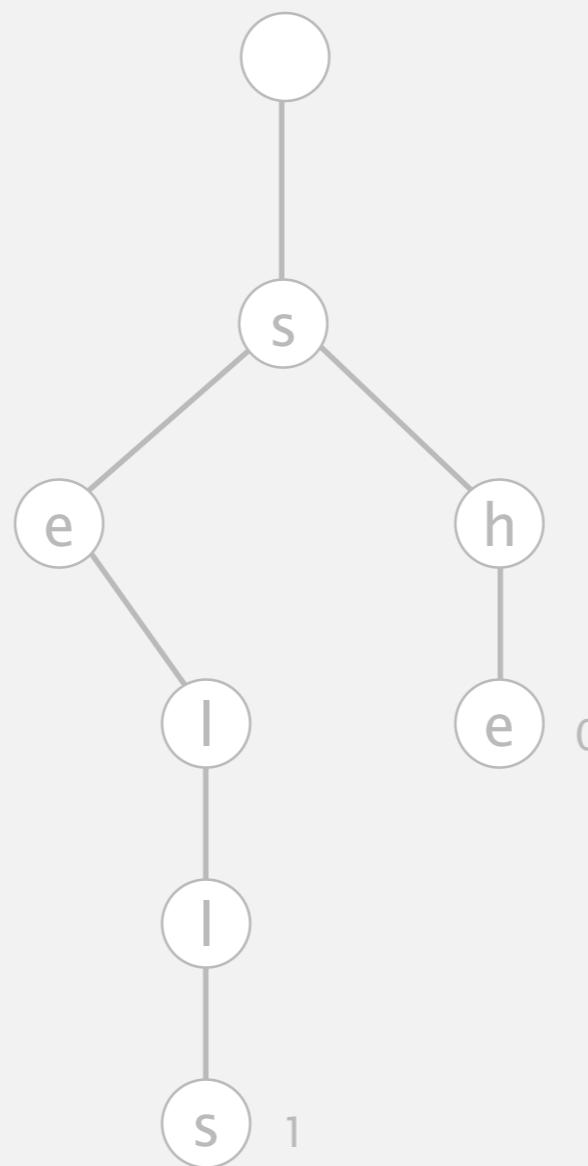


# Trie construction demo

she

sells

**trie**

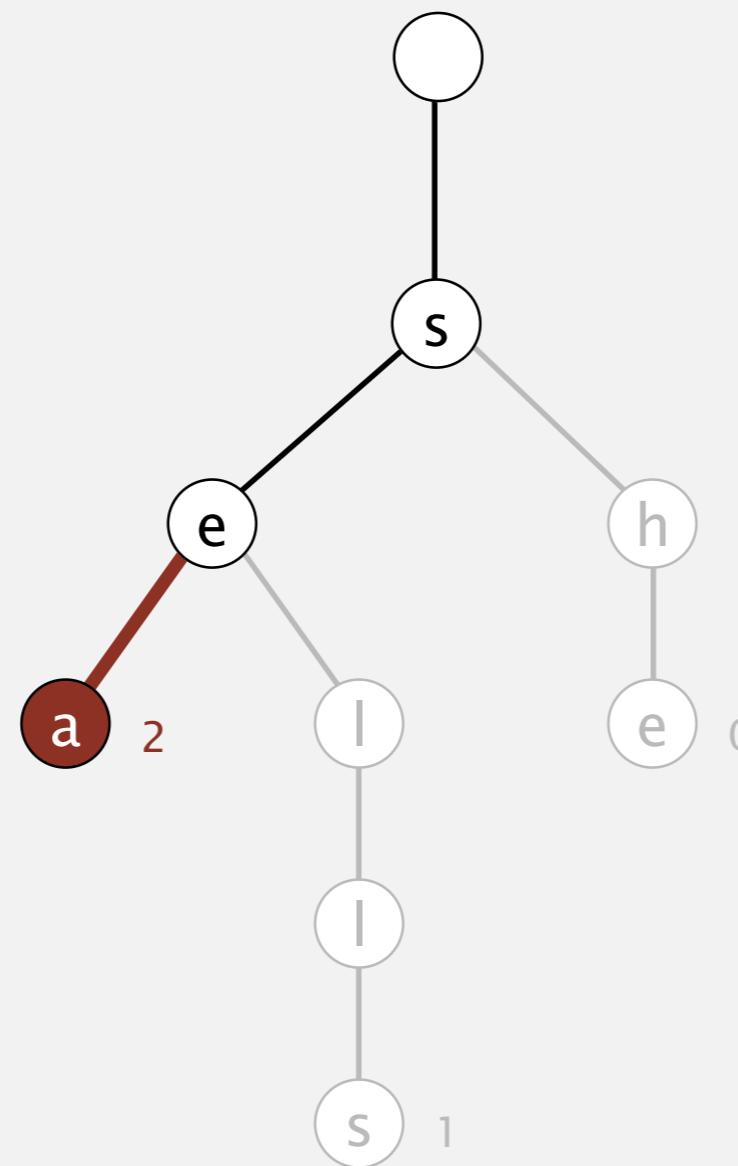


# Trie construction demo

she

sells

**put("sea", 2)**



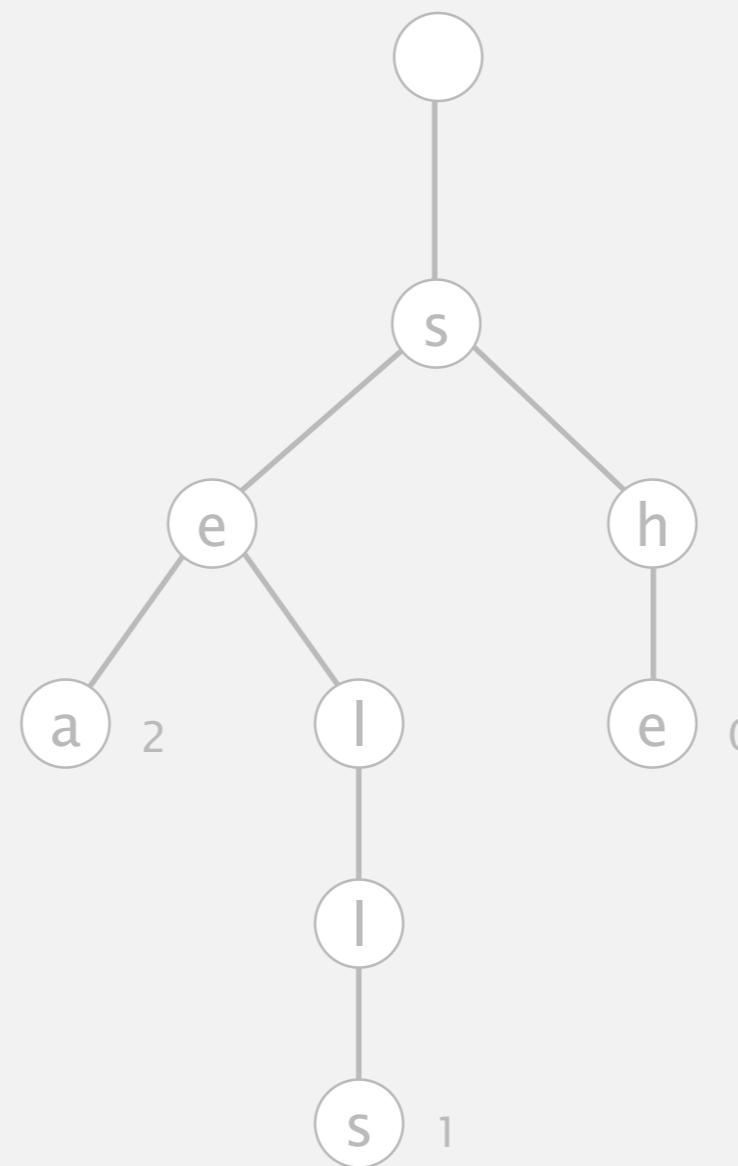
# Trie construction demo

she

sells

sea

trie



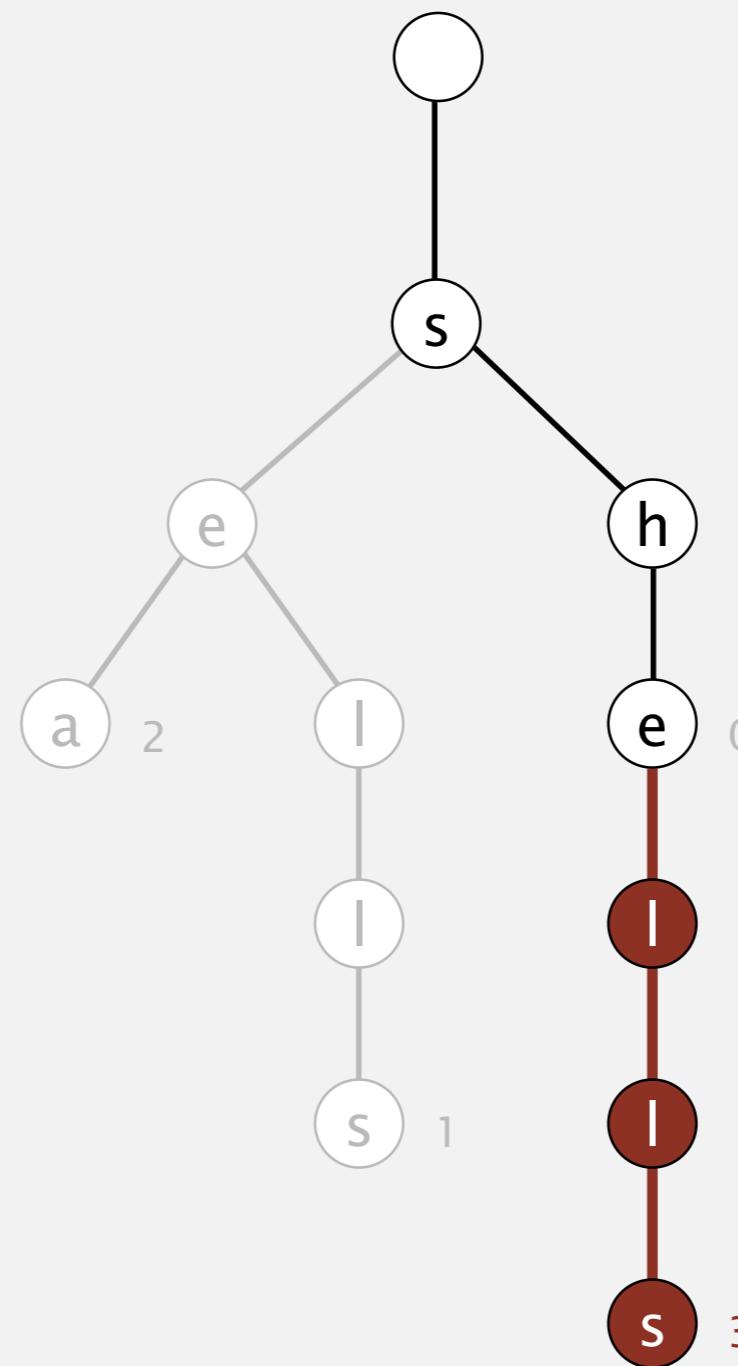
# Trie construction demo

she

sells

sea

**put("shells", 3)**



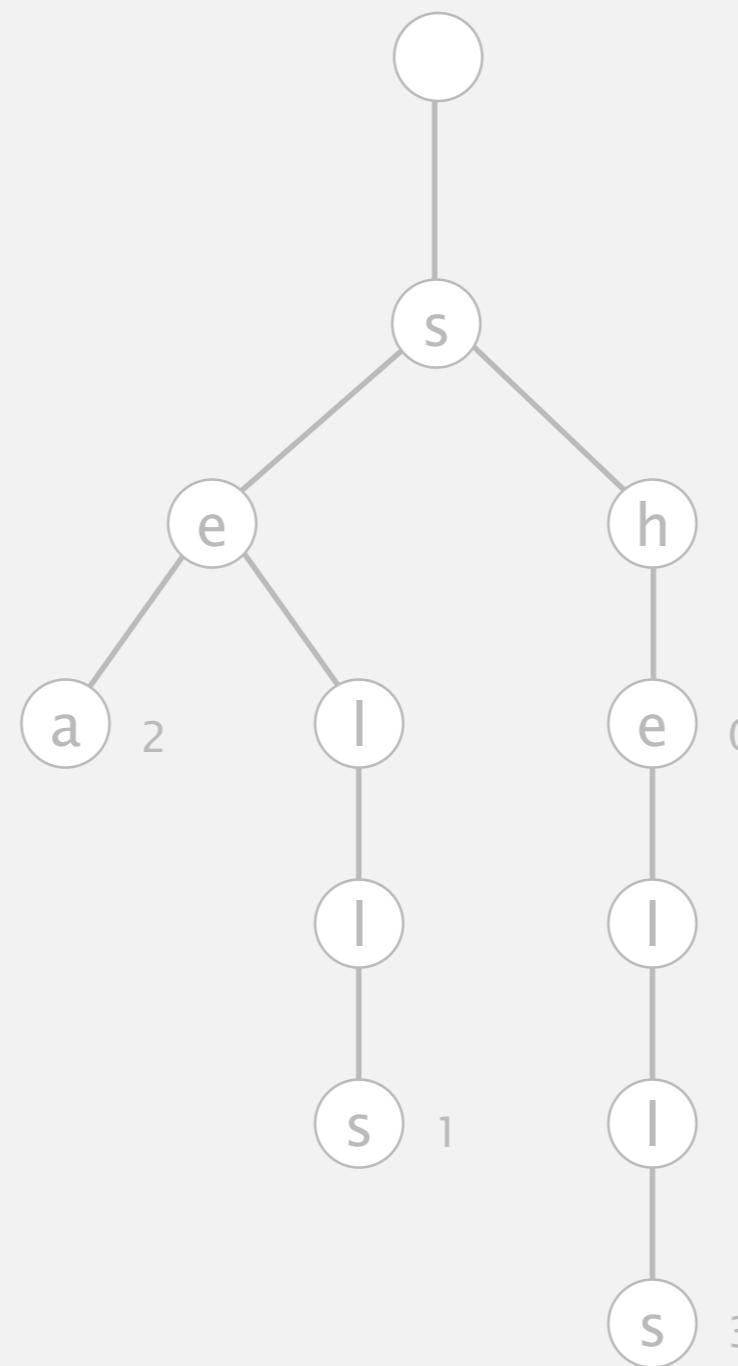
# Trie construction demo

she

sells

sea

trie



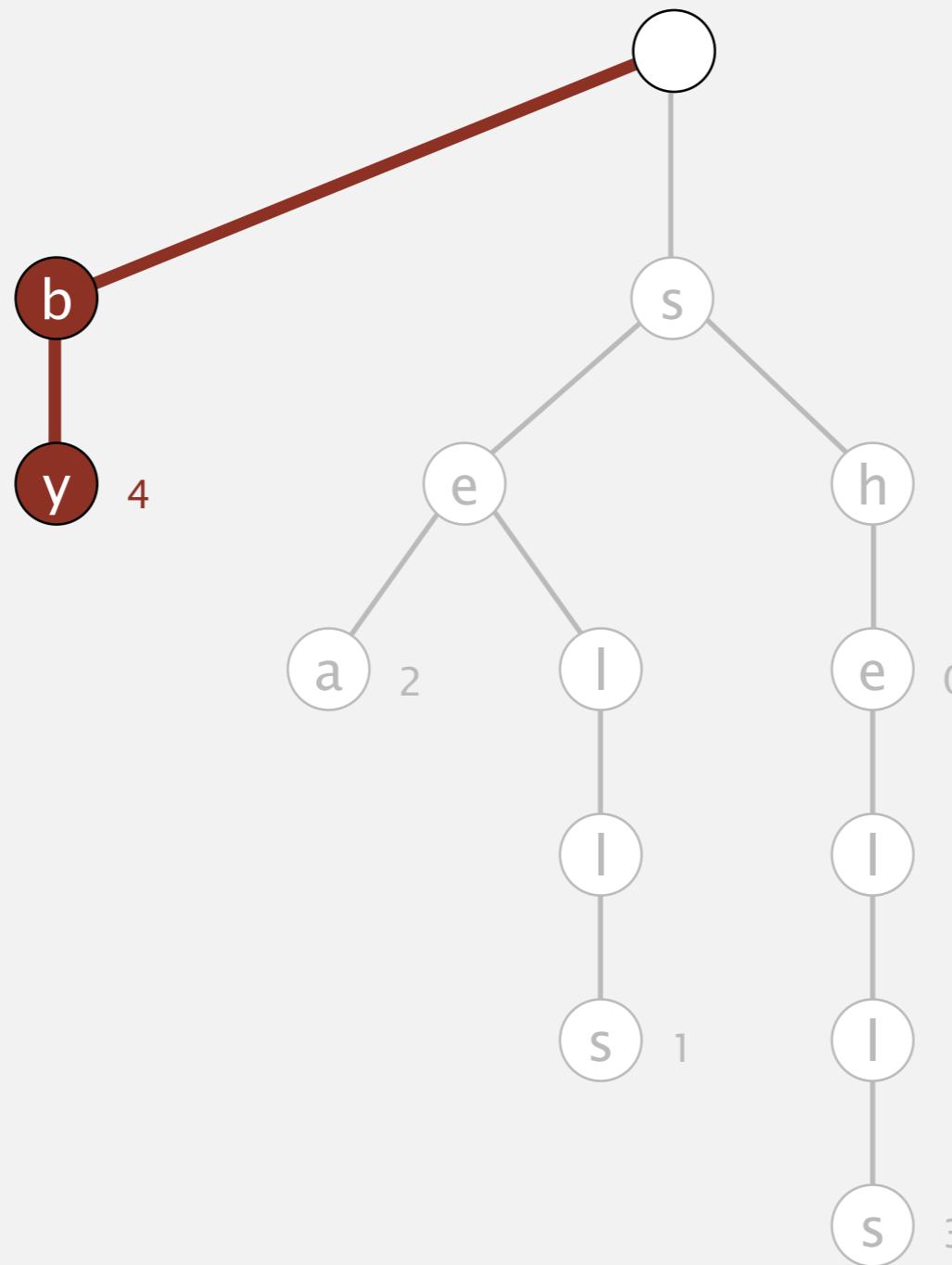
# Trie construction demo

she

sells

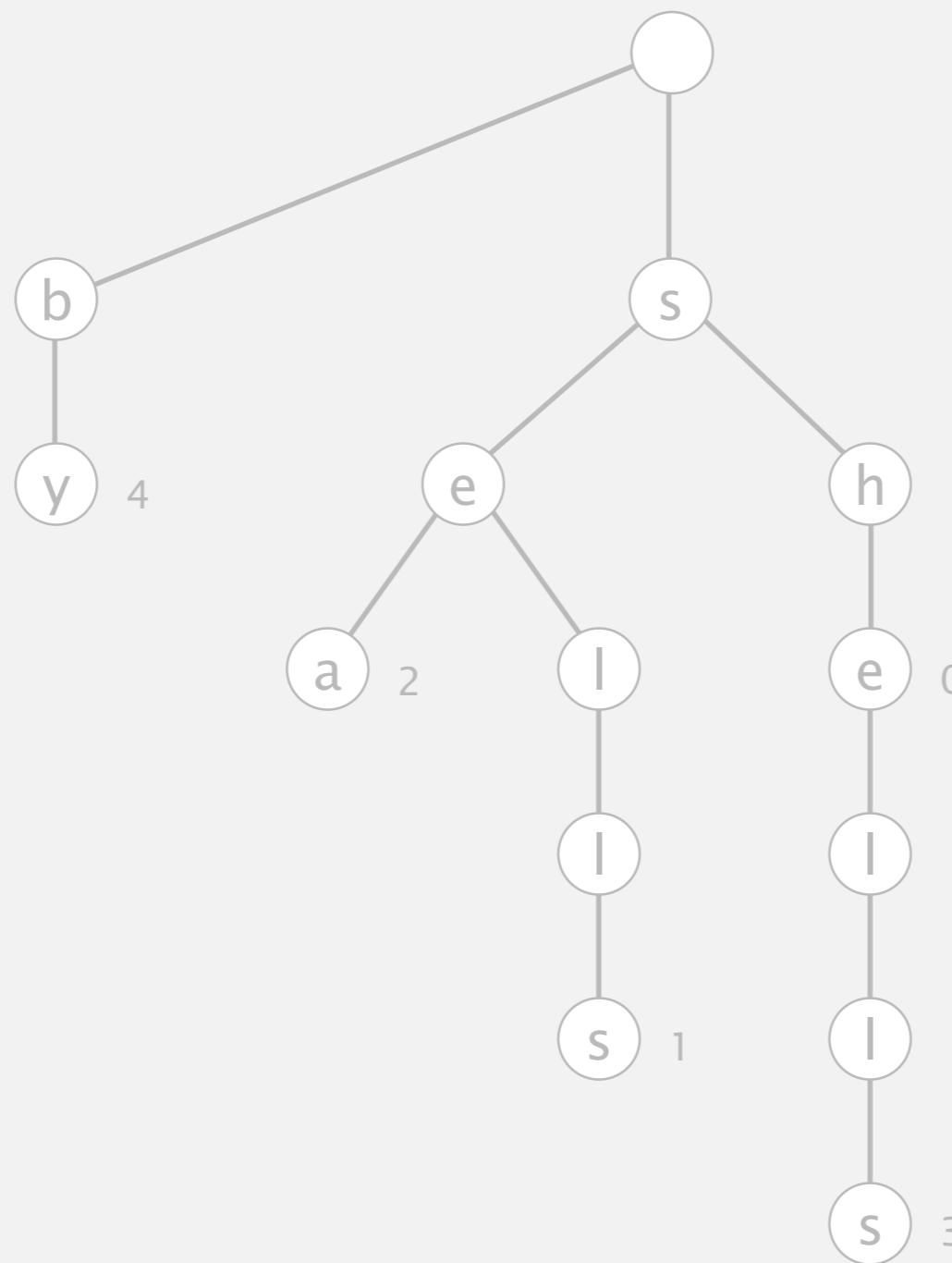
sea

**put("by", 4)**



# Trie construction demo

she  
sells  
sea  
by  
trie



# Trie construction demo

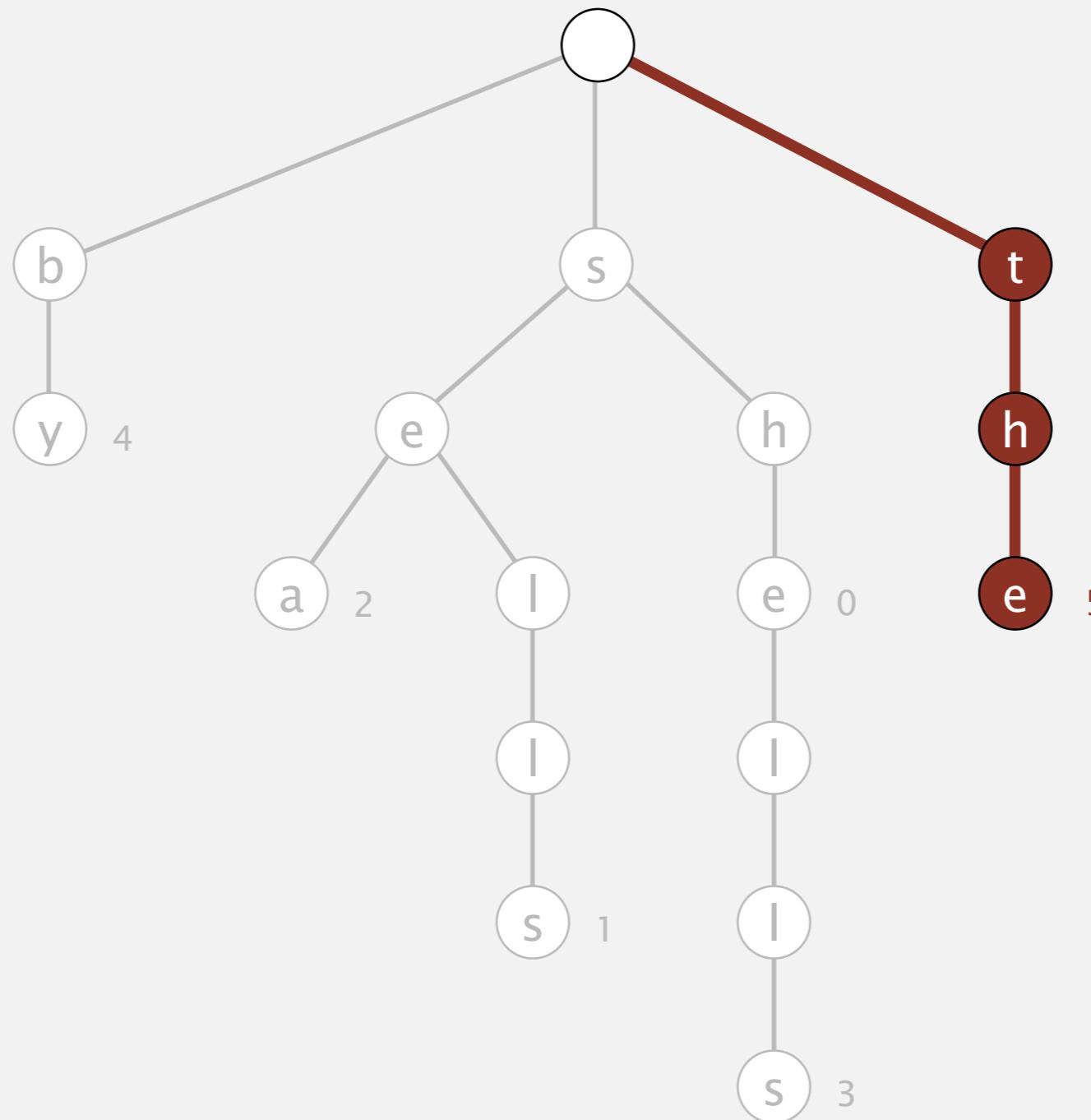
she

sells

sea

by

**put("the", 5)**



# Trie construction demo

she

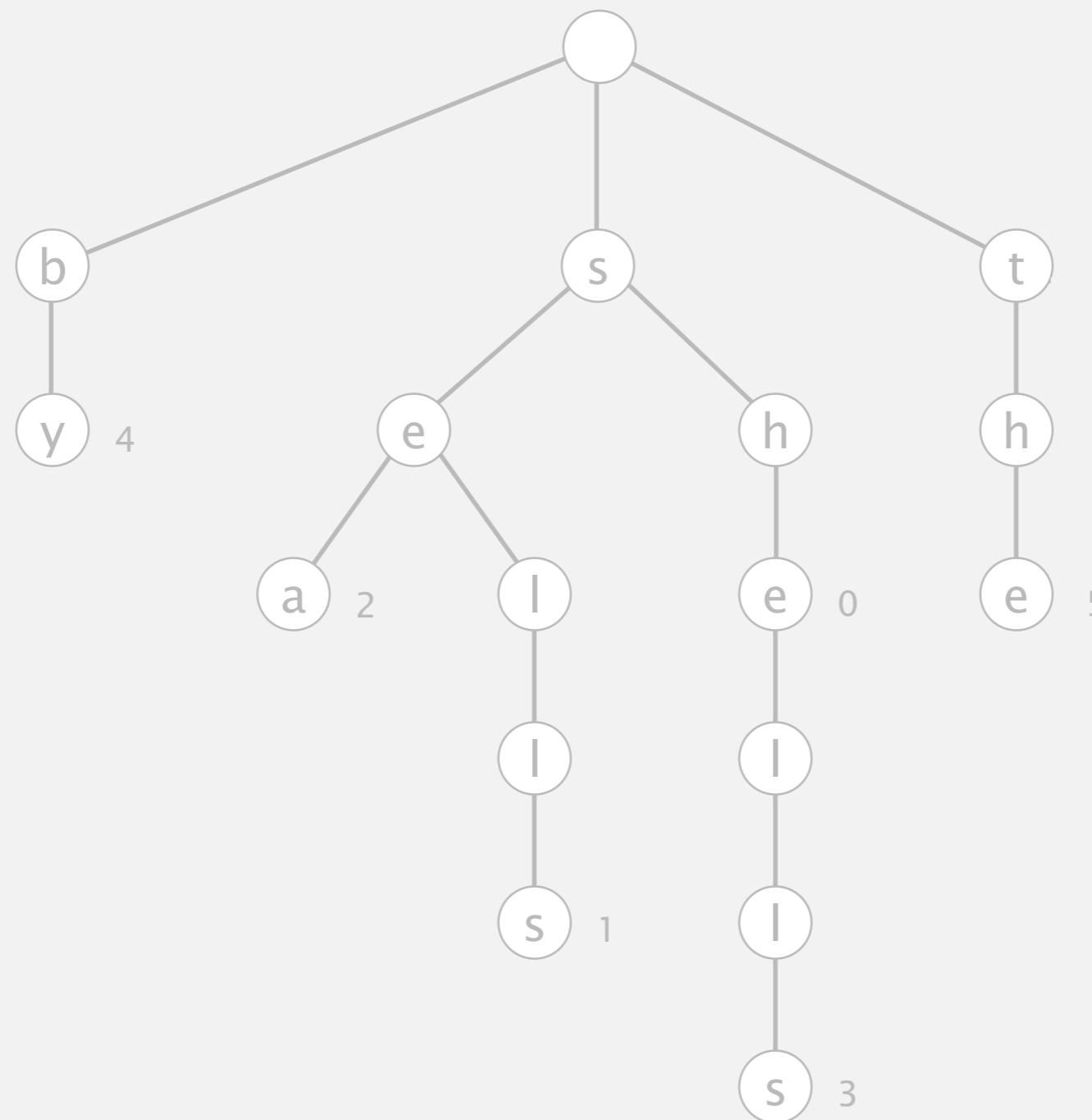
sells

sea

by

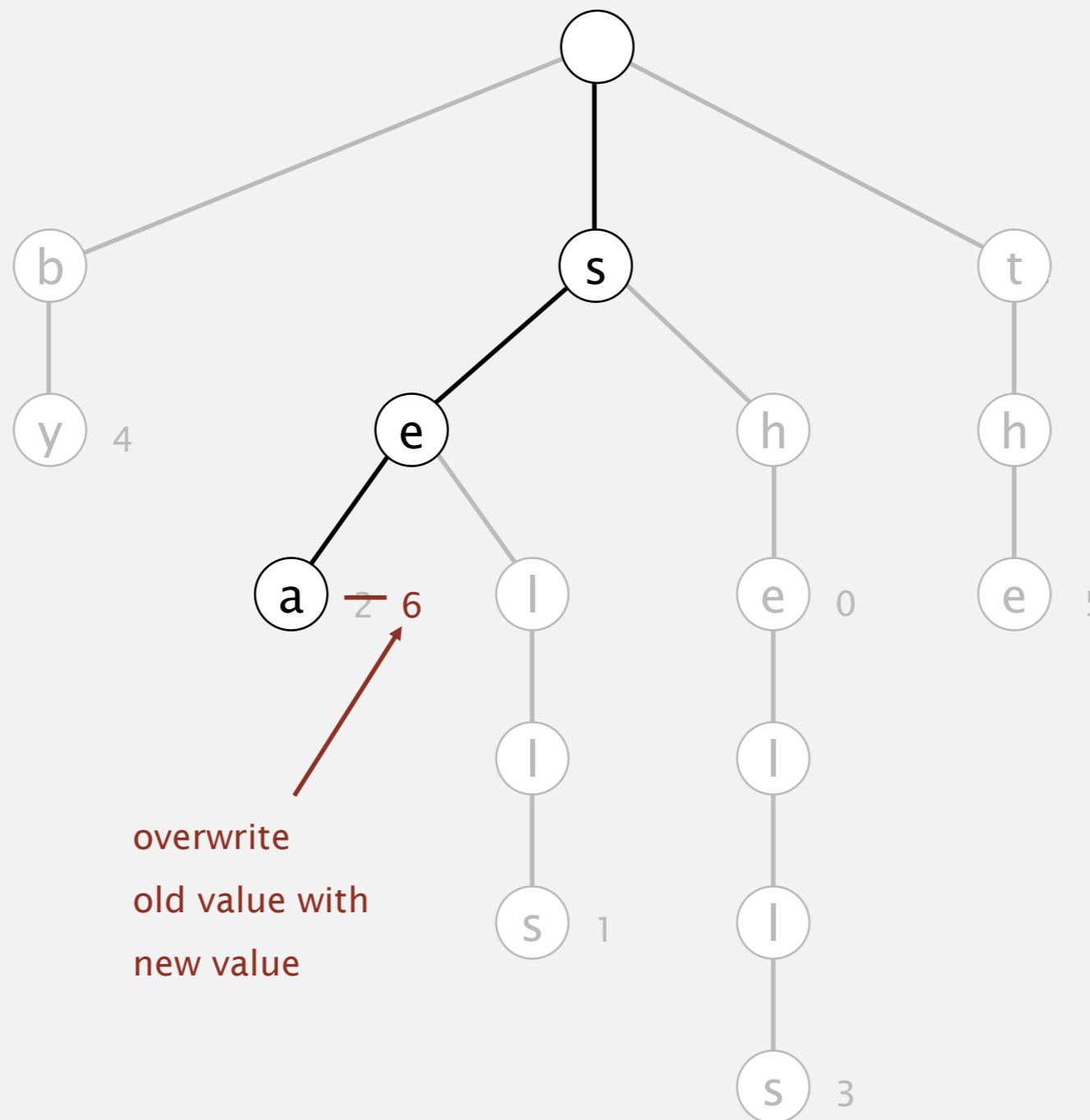
the

trie



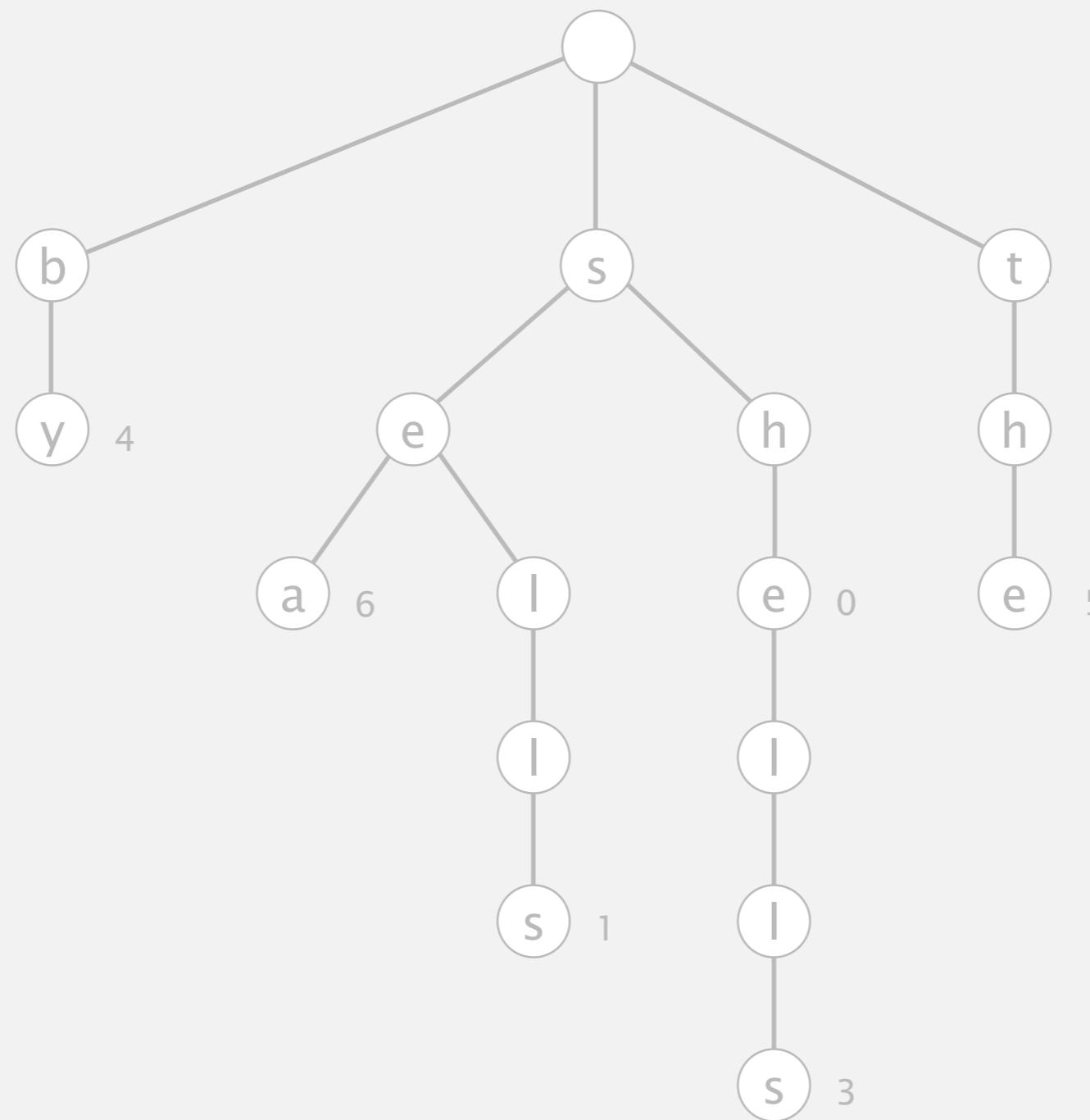
# Trie construction demo

`put("sea", 6)`



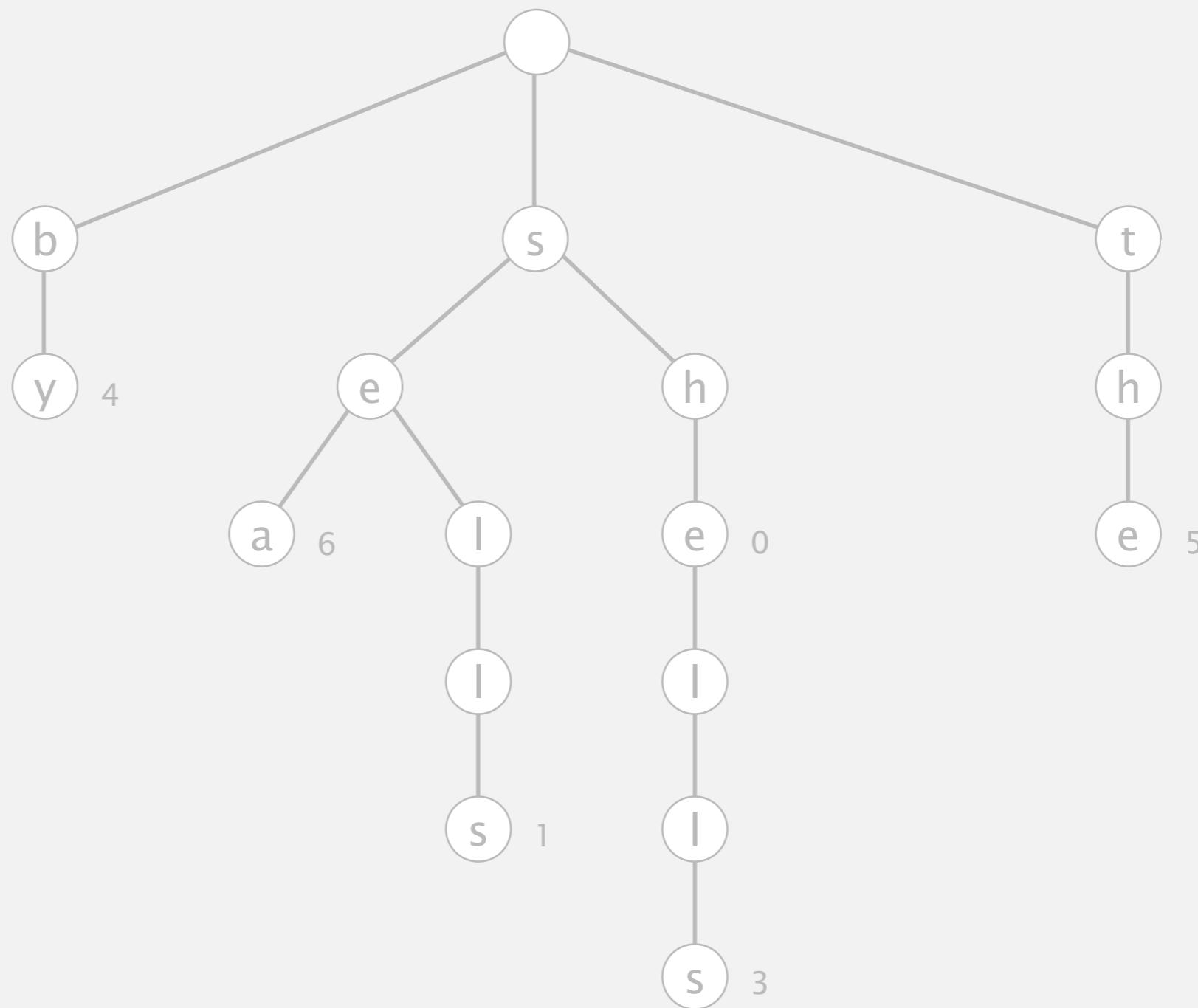
# Trie construction demo

trie



# Trie construction demo

trie



# Trie construction demo

she

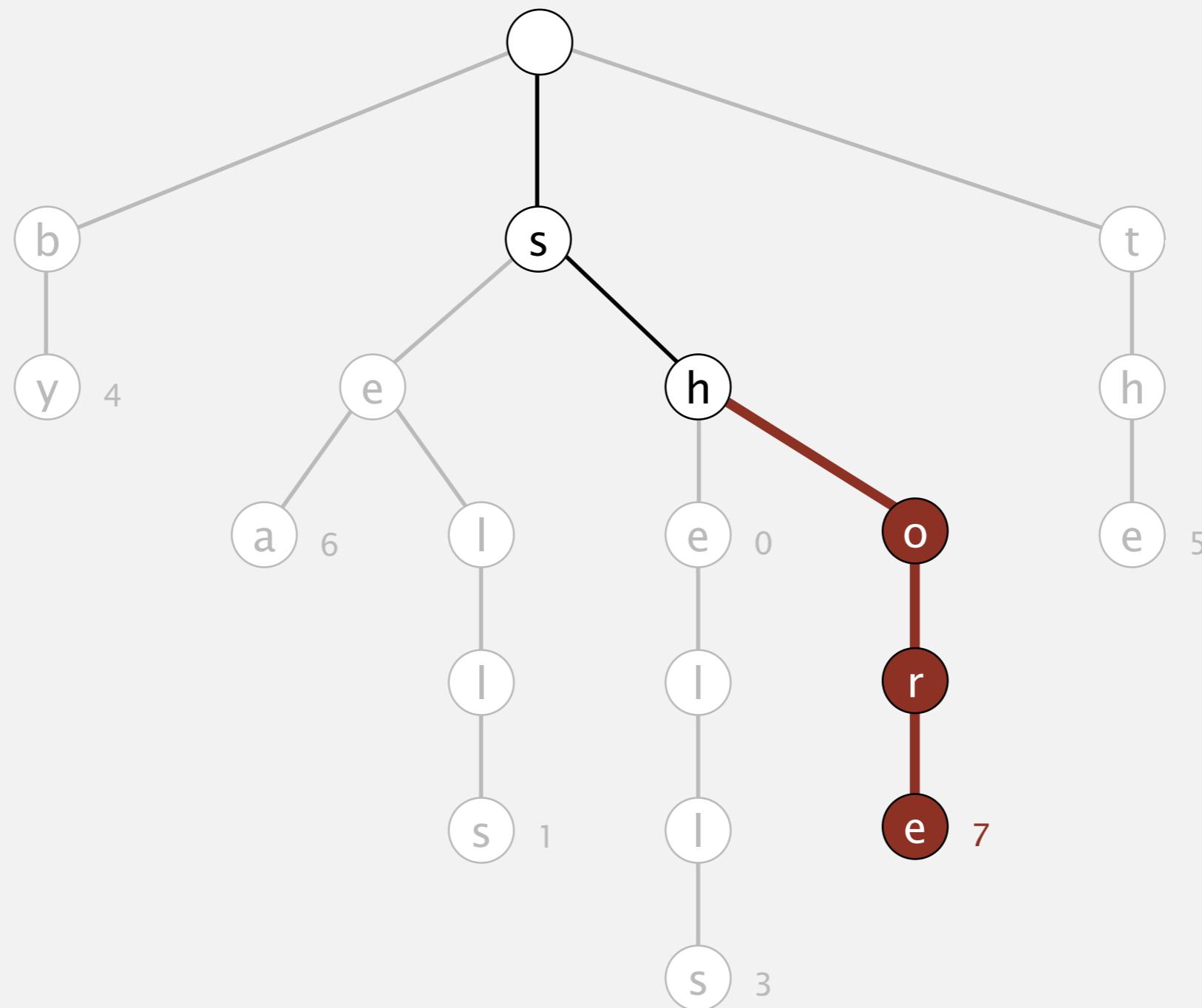
sells

sea

by

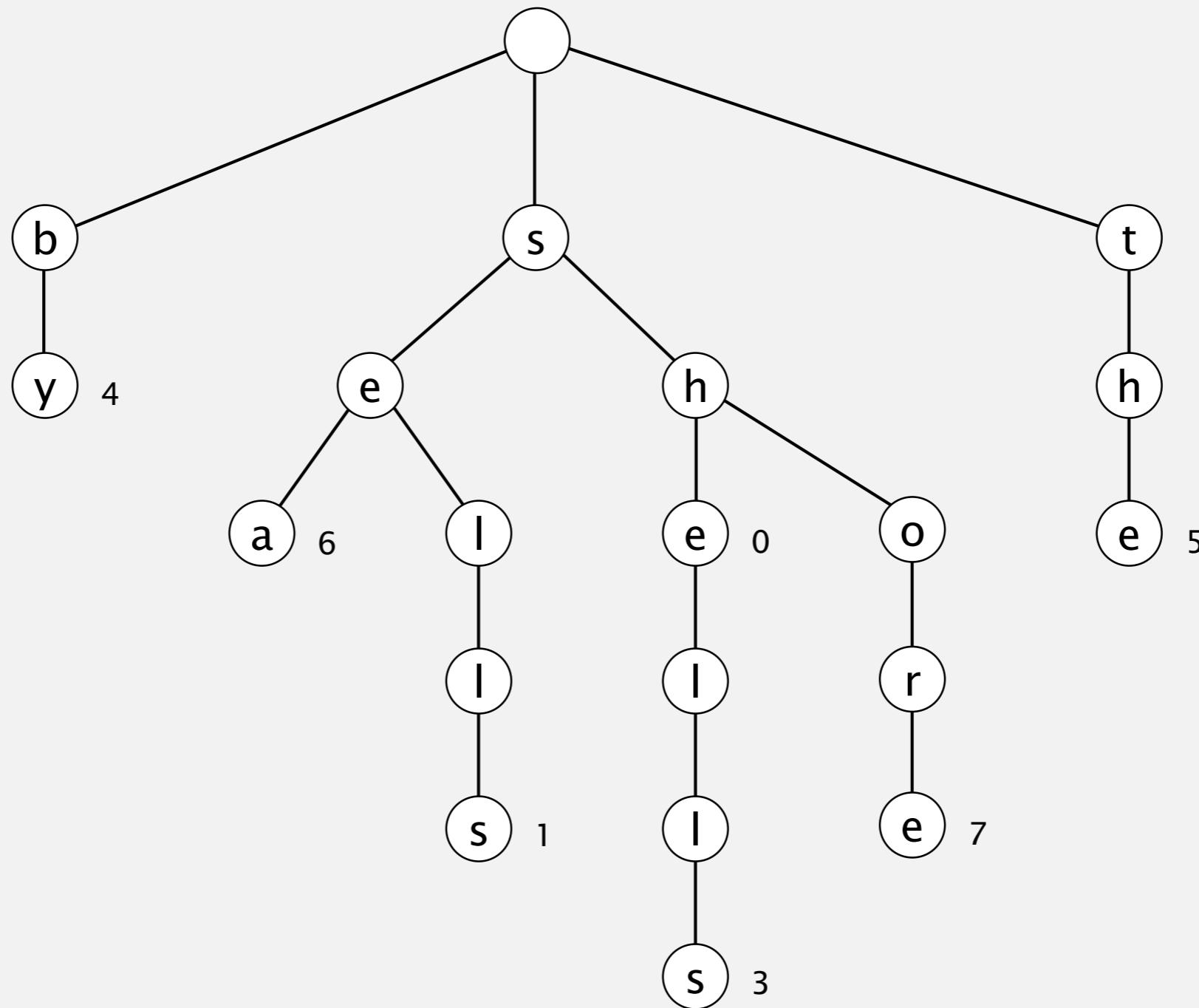
the

**put("shore", 7)**



# Trie construction demo

she  
sells  
sea  
by  
the  
shore  
**trie**

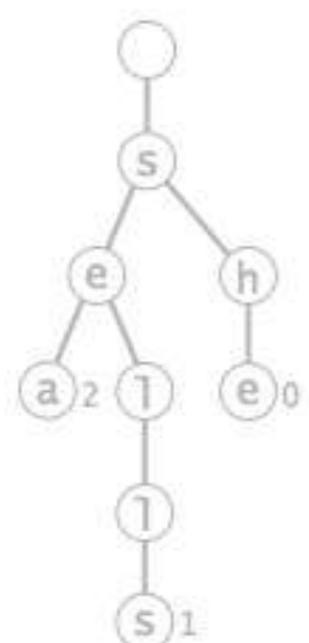


# Trie representation: implementation

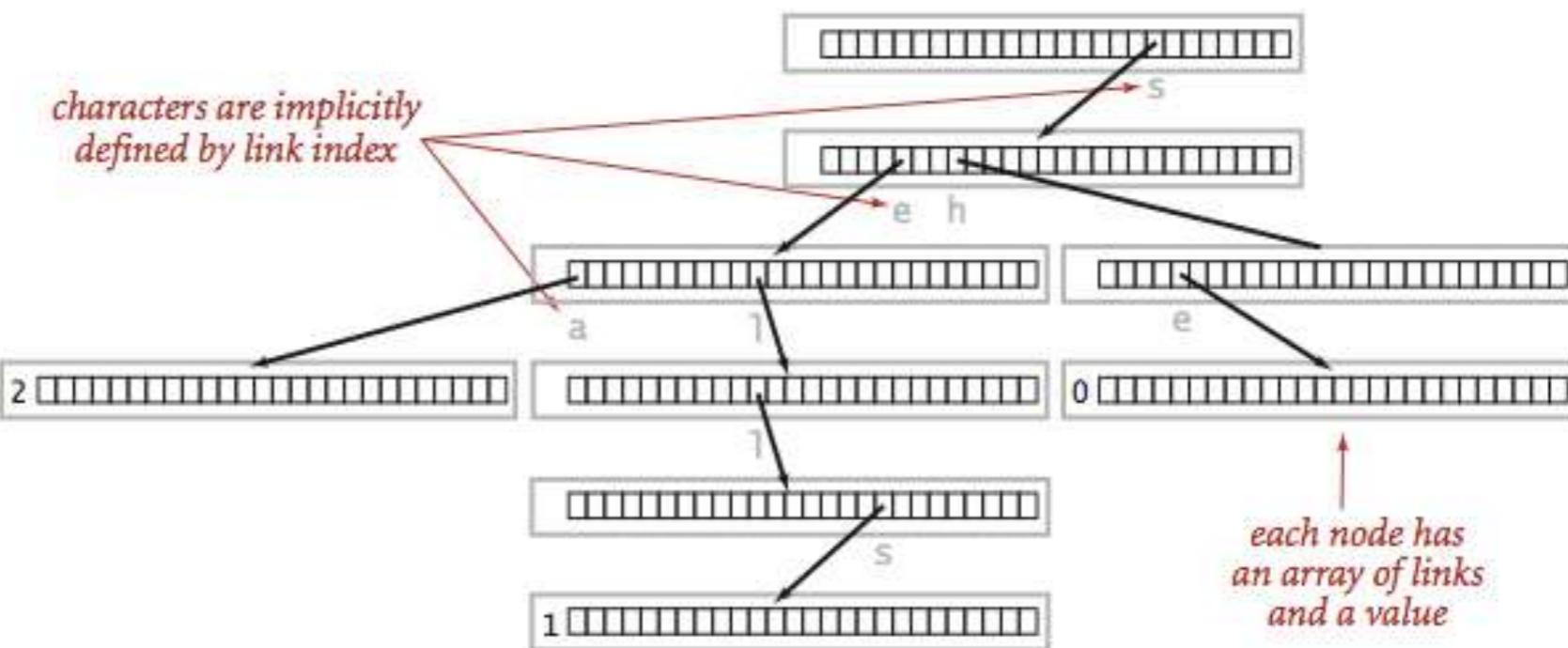
Node. A value, plus references to  $R$  nodes.

```
struct Node
{
    int value;
    Node * next[R];
}
```

A child node for each character in Alphabet.  
No need to search for character, but a  
pointer reserved for each character in  
memory



*characters are implicitly defined by link index*



Trie representation ( $R = 26$ )

# R-way trie: implementation

```
#define R 256 ← extended ASCII

Node * root;

put(&root, key, val, 0);

void put(Node ** x, char *key, int val, int d)
{
    if (*x == null) *x = getNode();
    if (d == strlen(key)) { *x->value = val; return; }
    char c = key[d];
    put(&(x->next[c]), key, val, d+1);
}

:::
```

## R-way trie: implementation (continued)

```
Node * getNode() {  
    Node * pNode = NULL;  
    pNode = (Node *)malloc(sizeof(Node));  
    if (pNode) {  
        for (int i = 0; i < R; i++)  
            pNode->next[i] = NULL;  
    }  
    return pNode;  
}
```

## R-way trie: implementation (continued)

```
int get(Node * x, char * key, int d)
{
    if (x == null) return -1; // -1 refers no match
    if (d == strlen(key)) return x->value;
    char c = key[d];
    return get(x->next[c], key, d+1);
}

}
```

# Trie performance

**Search hit.** Need to examine all  $L$  characters for equality.

**Search miss.**

- Could have mismatch on first character.
- Typical case: examine only a few characters (sublinear).

**Space.**  $R$  null links at each leaf.

(but sublinear space possible if many short strings share common prefixes)

**Bottom line.** Fast search hit and even faster search miss, but wastes space.

# String symbol table implementations cost summary

|                                | character accesses (typical case) |             |        |                             |
|--------------------------------|-----------------------------------|-------------|--------|-----------------------------|
| implementation                 | search <sub>hit</sub>             | Search miss | insert | space <sub>references</sub> |
| hashing<br>(separate chaining) | N                                 | N           | 1      | N                           |
| R-way trie                     | L                                 | $\log_R N$  | L      | $RNw$                       |

$N$  = number of entries,  $L$  = key length,  
 $R$  = alphabet size,  $w$  = average key length

## R-way trie.

- Method of choice for small  $R$ .
- Too much memory for large  $R$ .

**Challenge.** Use less memory, e.g., 65,536-way trie for Unicode!

# BBM 201 – DATA STRUCTURES



HACETTEPE UNIVERSITY

**DEPT. OF COMPUTER ENGINEERING**

**UNDIRECTED GRAPHS**

**Acknowledgement:** The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.

# TODAY

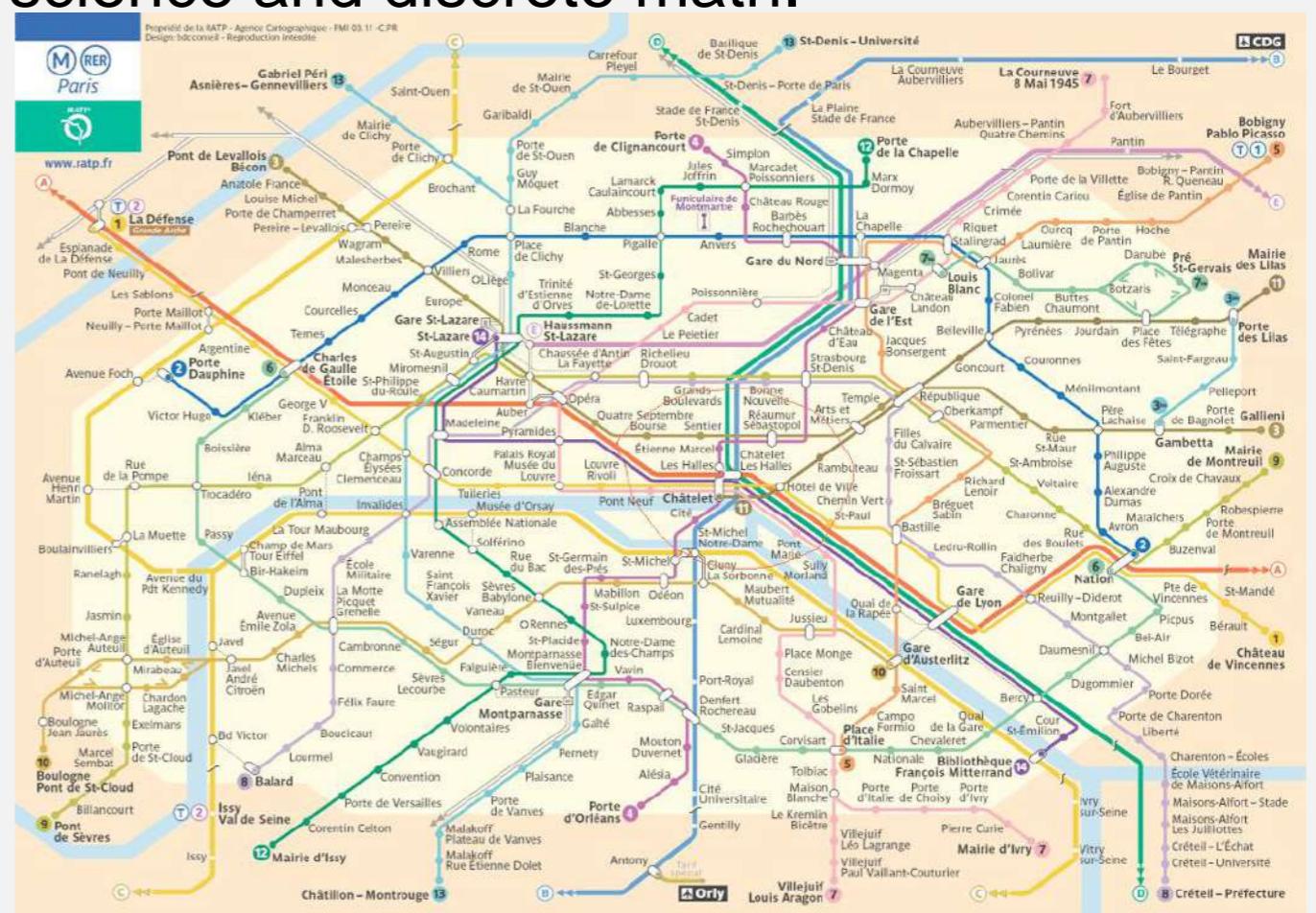
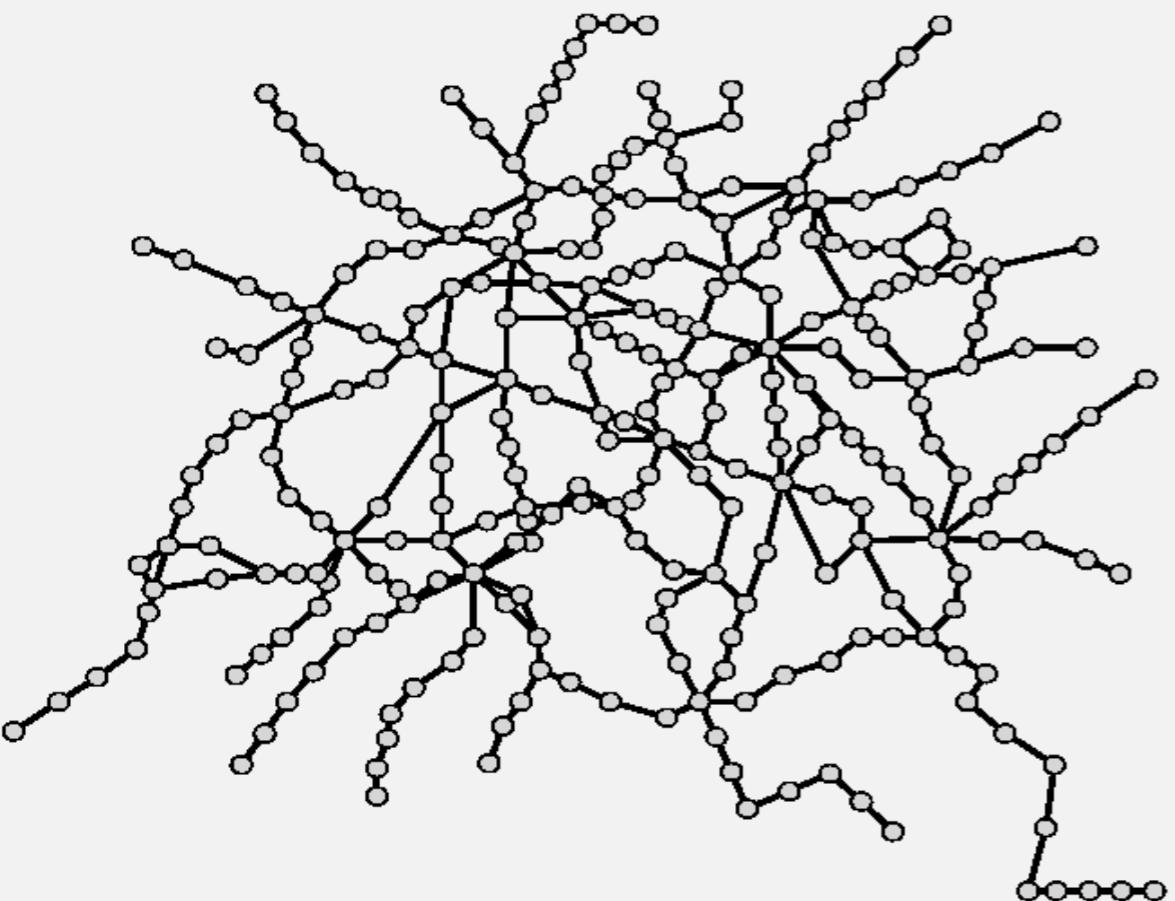
- ▶ **Undirected Graphs**
- ▶ **Graph API**
- ▶ **Depth-first search**
- ▶ **Breadth-first search**

# Undirected graphs

Graph. Set of vertices connected pairwise by edges.

## Why study graph algorithms?

- Thousands of practical applications.
- Hundreds of graph algorithms known.
- Interesting and broadly useful abstraction.
- Challenging branch of computer science and discrete math.

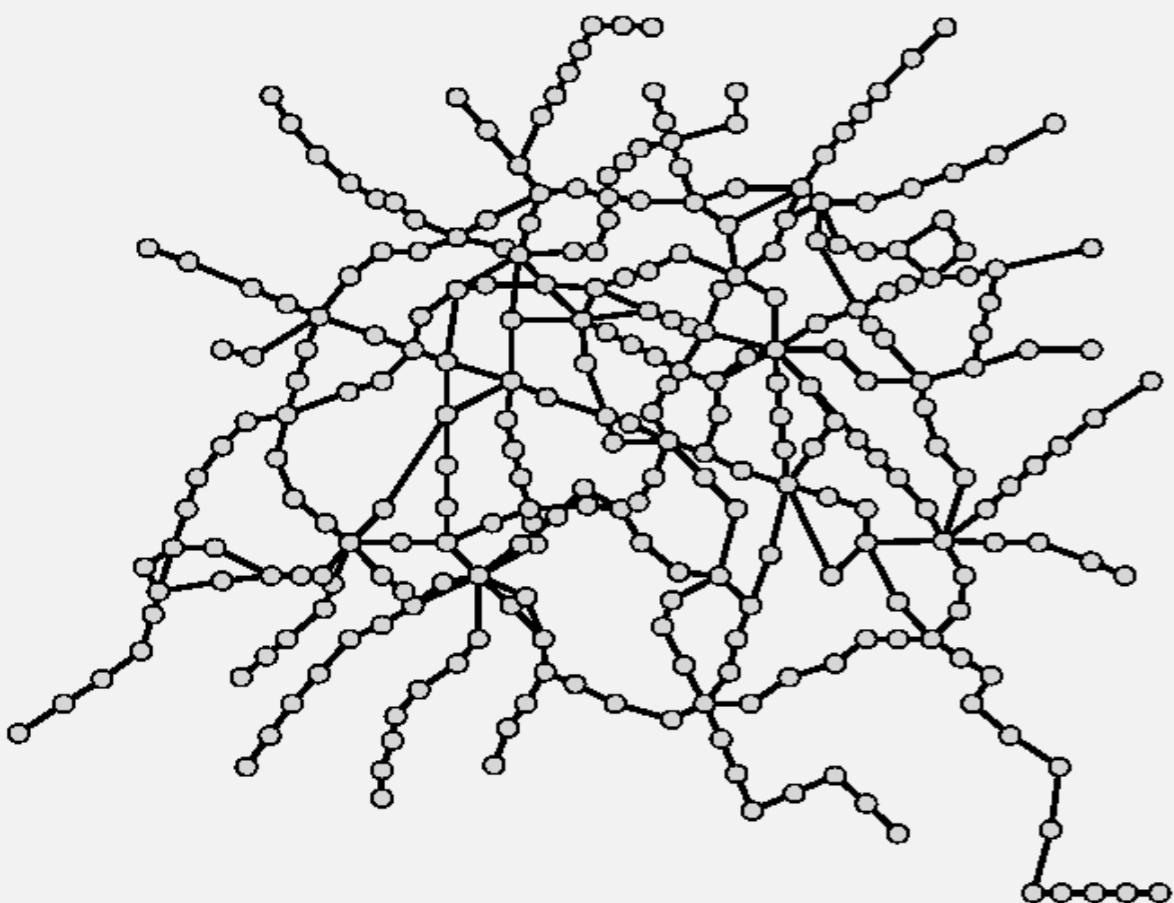


# Undirected graphs

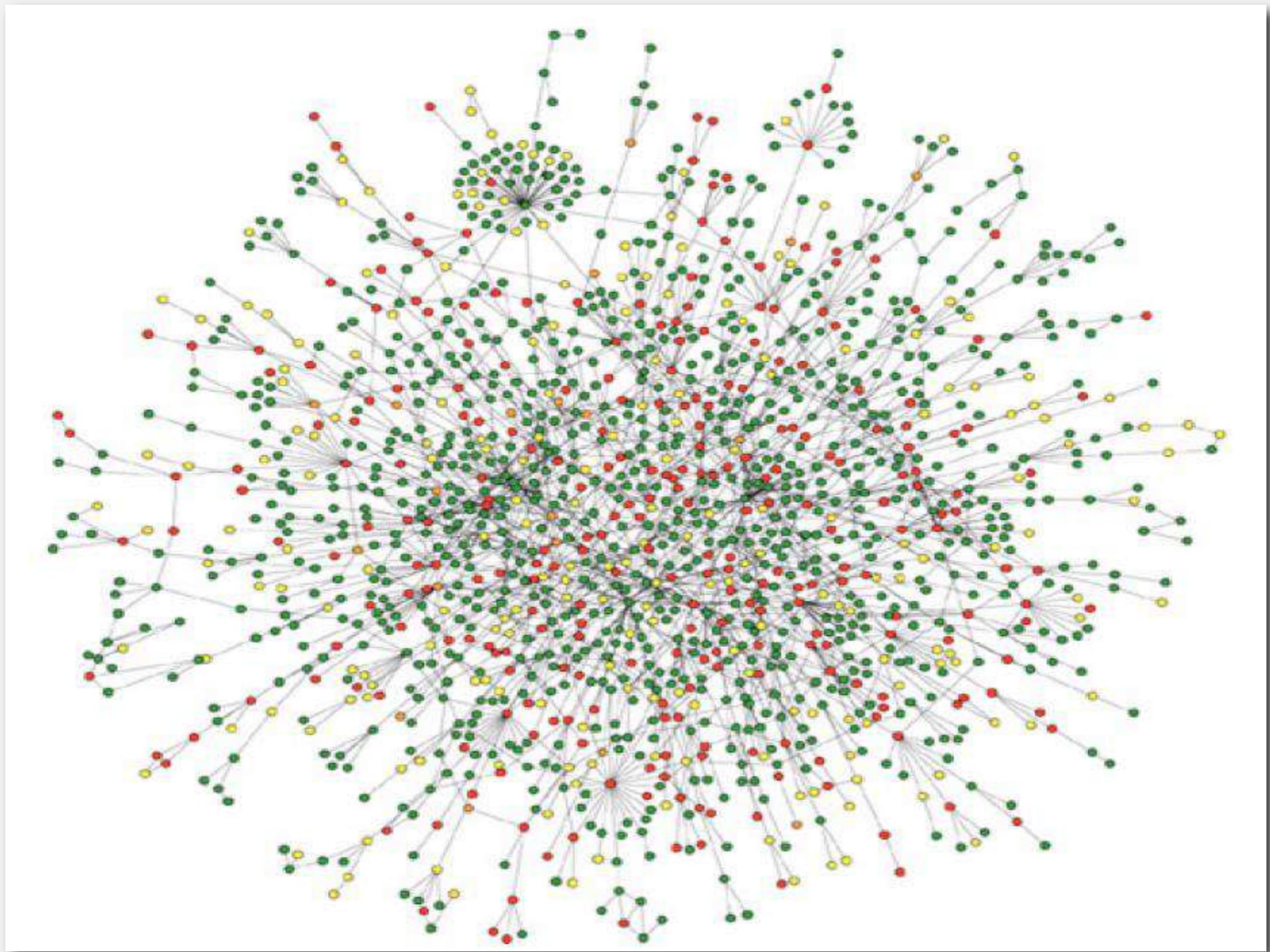
Graph. Set of **vertices** connected pairwise by **edges**.

## Why study graph algorithms?

- Thousands of practical applications.
- Hundreds of graph algorithms known.
- Interesting and broadly useful abstraction.
- Challenging branch of computer science and discrete math.

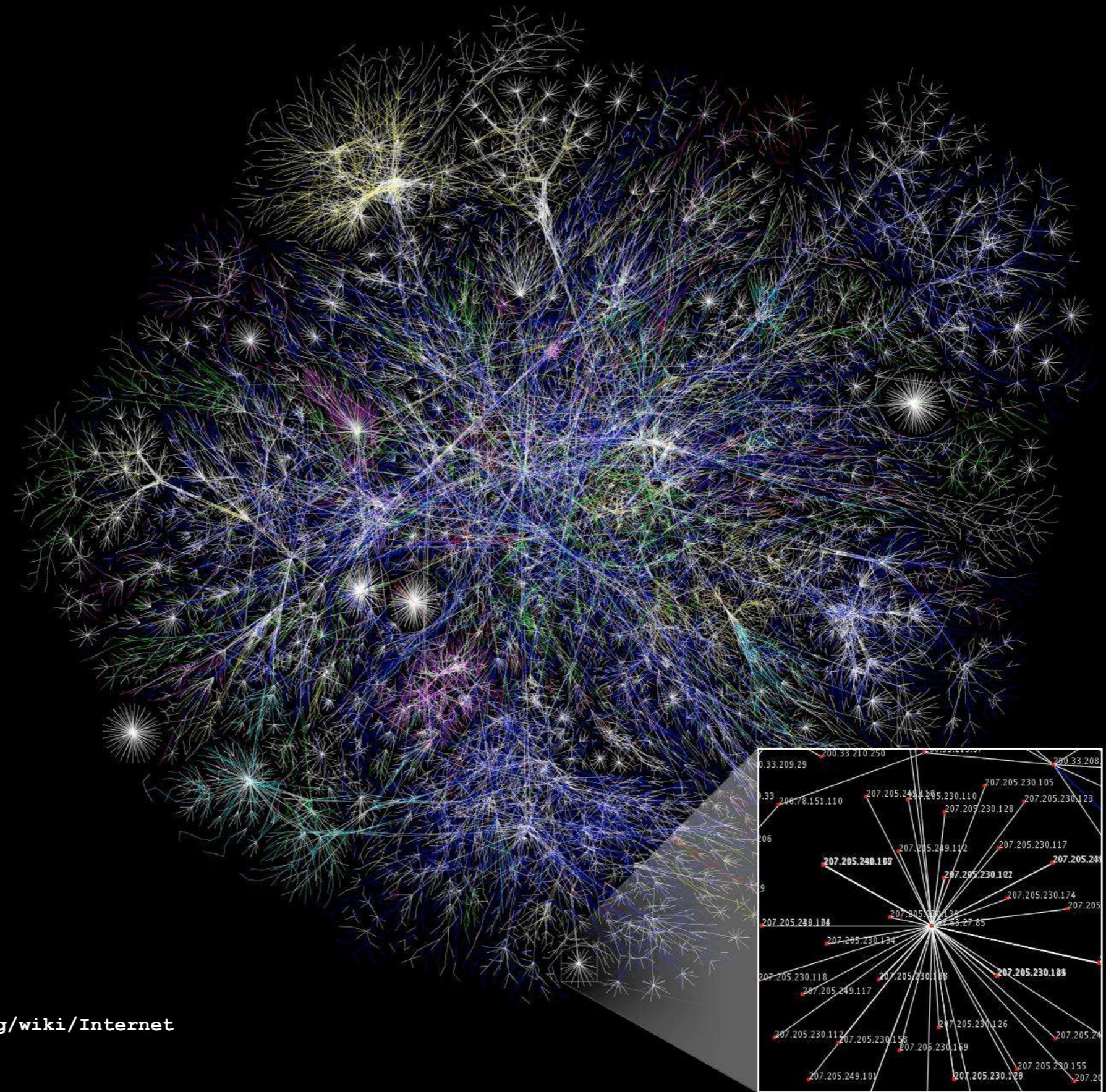


# Protein-protein interaction network

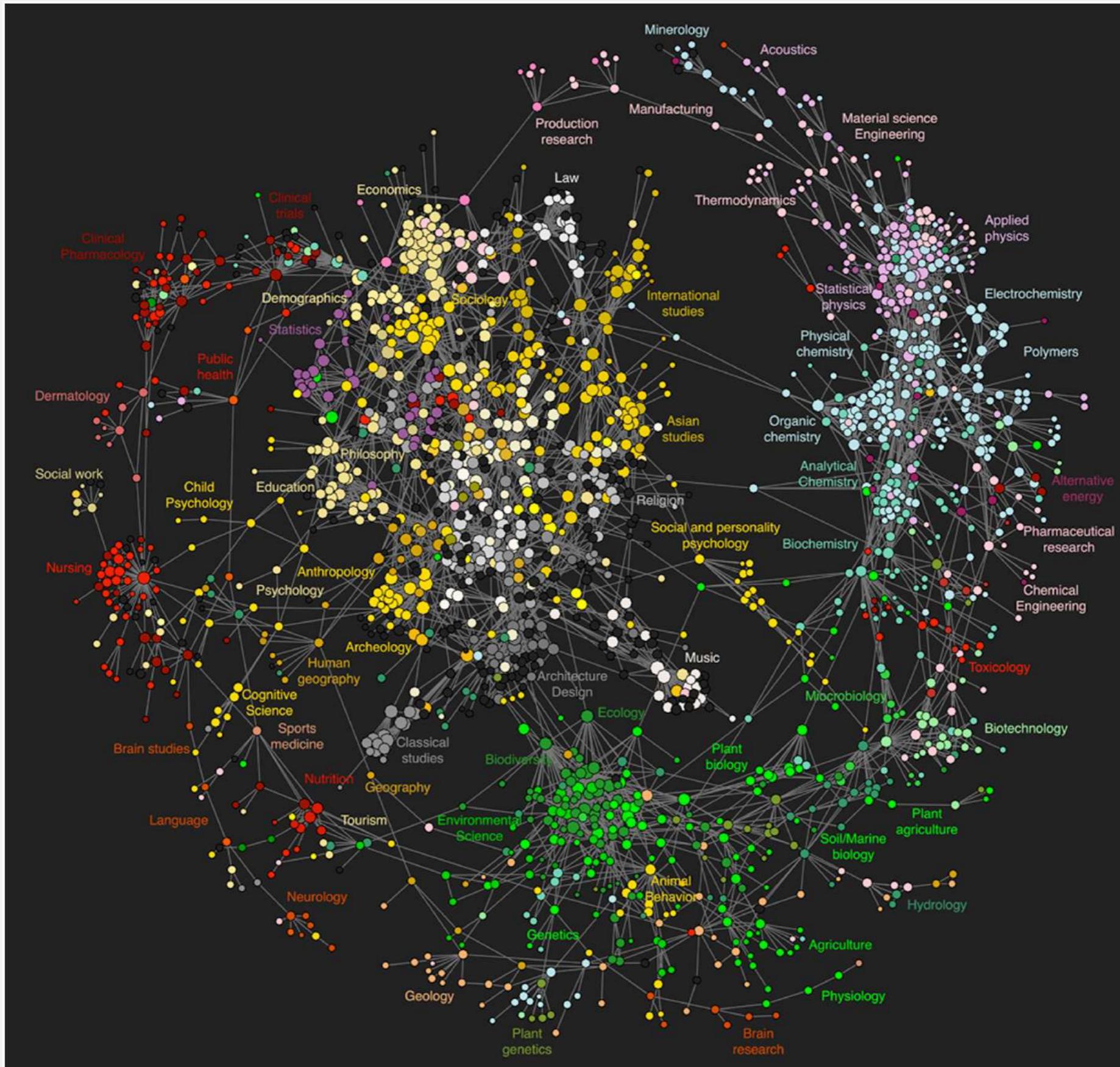


Reference: Jeong et al, Nature Review | Genetics

# The Internet as mapped by the Opte Project



# Map of science clickstreams



# 10 million Facebook friends



"Visualizing Friendships" by Paul Butler

# Graph applications

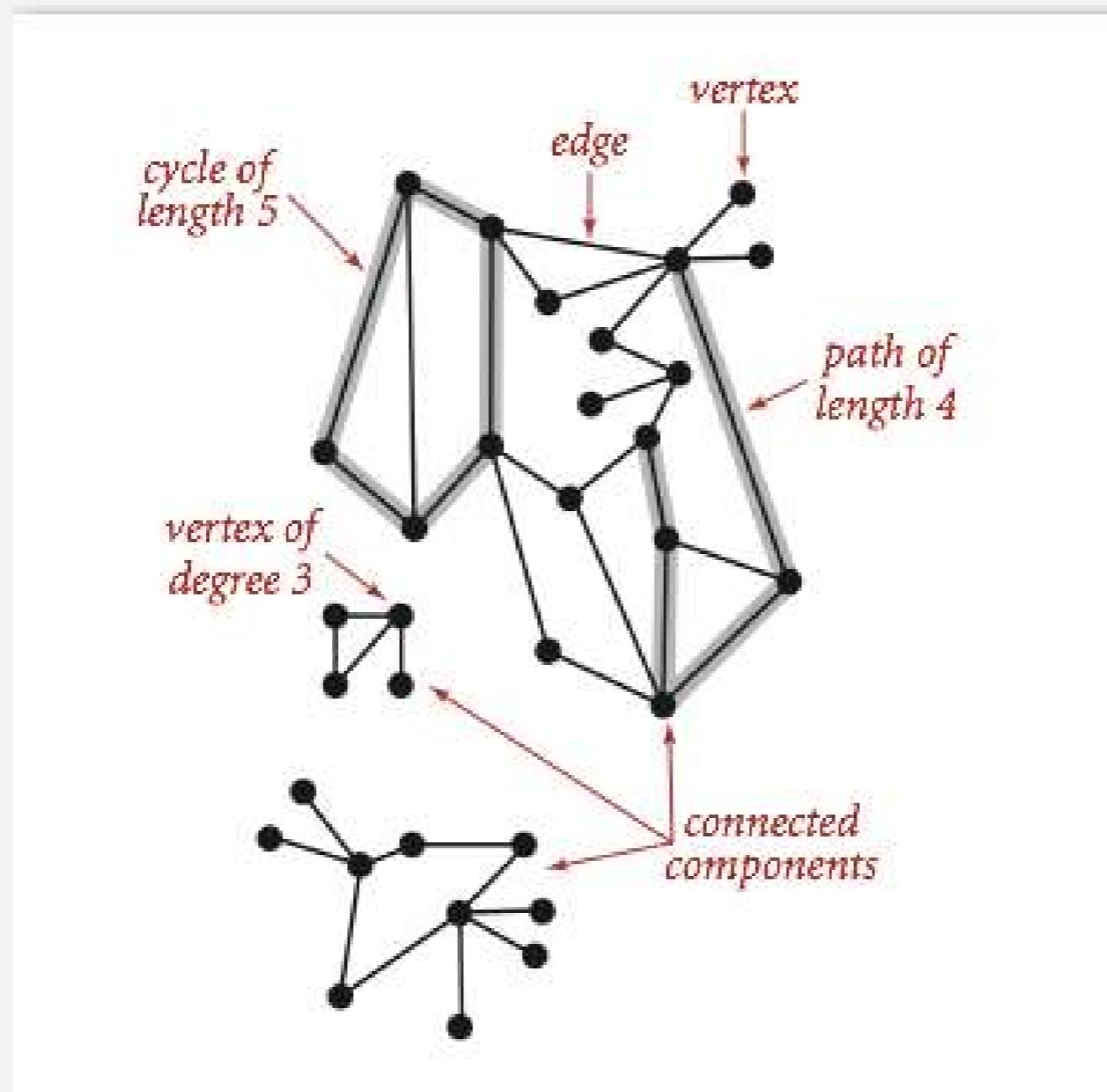
| graph               | vertex                       | edge                        |
|---------------------|------------------------------|-----------------------------|
| communication       | telephone, computer          | fiber optic cable           |
| circuit             | gate, register, processor    | wire                        |
| mechanical          | joint                        | rod, beam, spring           |
| financial           | stock, currency              | transactions                |
| transportation      | street intersection, airport | highway, airway route       |
| internet            | class C network              | connection                  |
| game                | board position               | legal move                  |
| social relationship | person, actor                | friendship, movie cast      |
| neural network      | neuron                       | synapse                     |
| protein network     | protein                      | protein-protein interaction |
| chemical compound   | molecule                     | bond                        |

# Graph terminology

**Path.** Sequence of vertices connected by edges.

**Cycle.** Path whose first and last vertices are the same.

Two vertices are **connected** if there is a path between them.



# Some graph-processing problems

**Path.** Is there a path between  $s$  and  $t$ ?

**Shortest path.** What is the shortest path between  $s$  and  $t$ ?

**Cycle.** Is there a cycle in the graph?

**Euler tour.** Is there a cycle that uses each edge exactly once?

**Hamilton tour.** Is there a cycle that uses each vertex exactly once?

**Connectivity.** Is there a way to connect all of the vertices?

**MST.** What is the best way to connect all of the vertices?

**Biconnectivity.** Is there a vertex whose removal disconnects the graph?

**Planarity.** Can you draw the graph in the plane with no crossing edges?

**Graph isomorphism.** Do two adjacency lists represent the same graph?

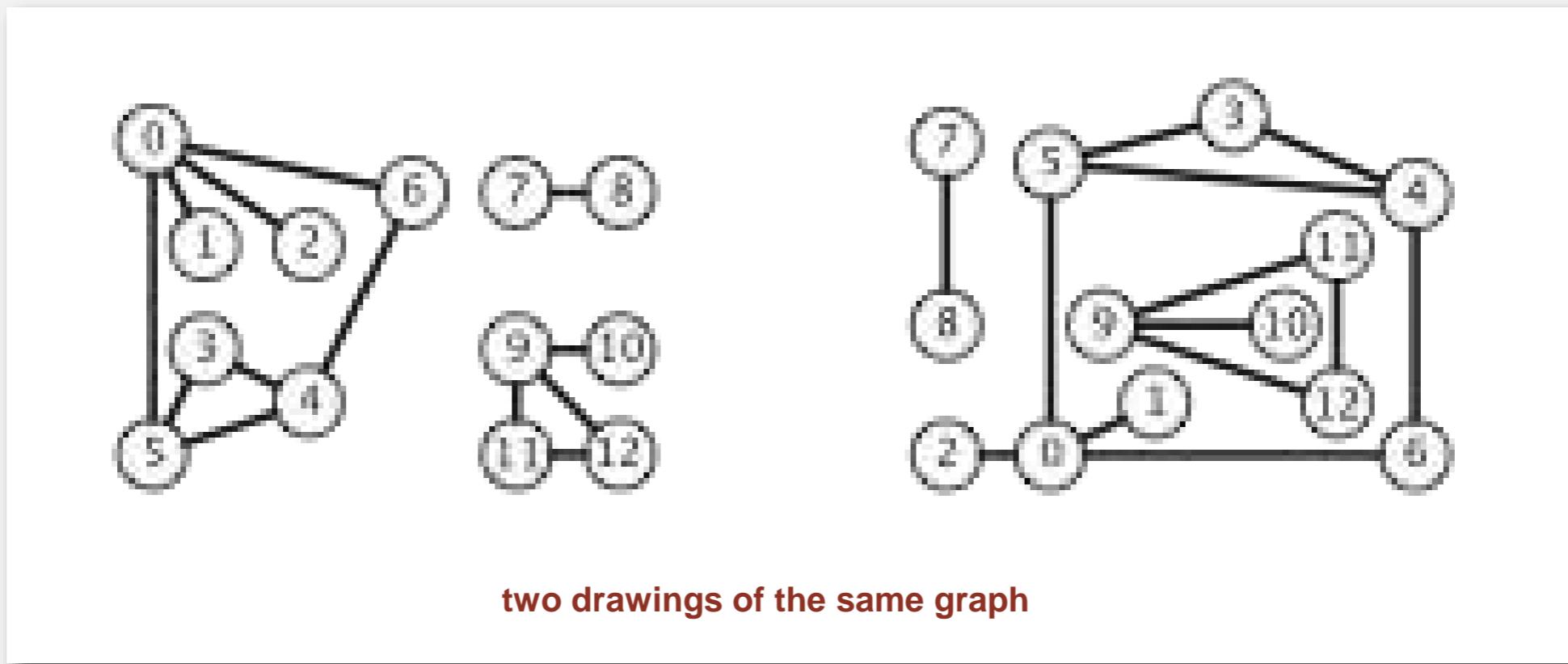
**Challenge.** Which of these problems are easy? difficult? intractable?

# UNDIRECTED GRAPHS

- ▶ **Graph API**
- ▶ **Depth-first search**
- ▶ **Breadth-first search**

# Graph representation

Graph drawing. Provides intuition about the structure of the graph.

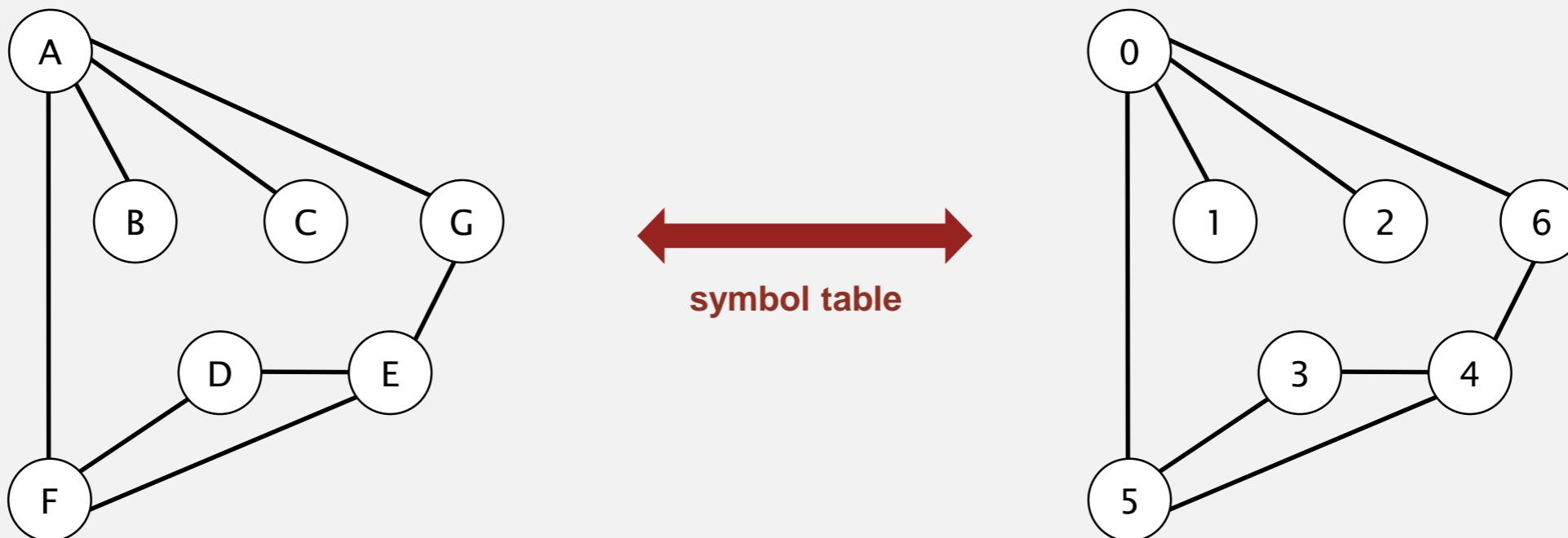


Caveat. Intuition can be misleading.

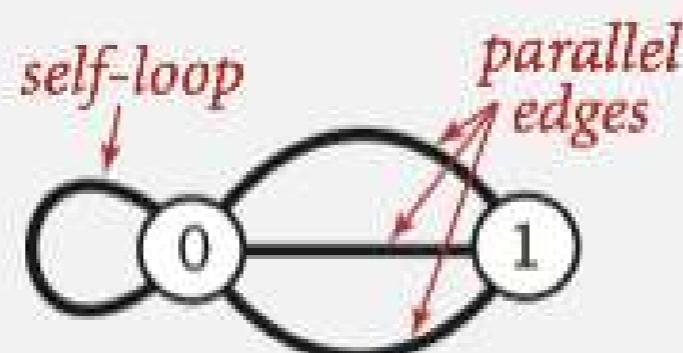
# Graph representation

## Vertex representation.

- This lecture: use integers between 0 and  $V - 1$ .
- Applications: convert between names and integers with symbol table.



## Anomalies.



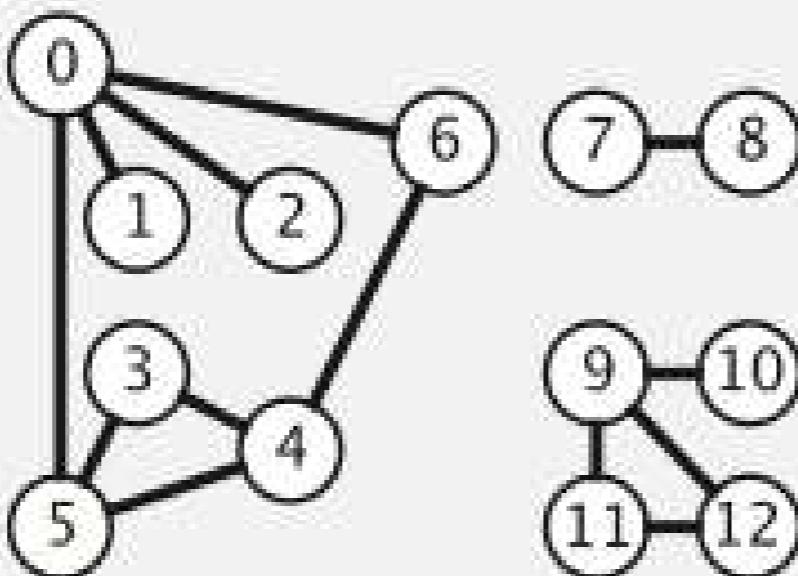
# Graph API: sample client

Graph input format.

**tinyG.txt**

V → 13  
E → 13

0 5  
4 3  
0 1  
9 12  
6 4  
5 4  
0 2  
11 12  
9 10  
0 6  
7 8  
9 11  
5 3

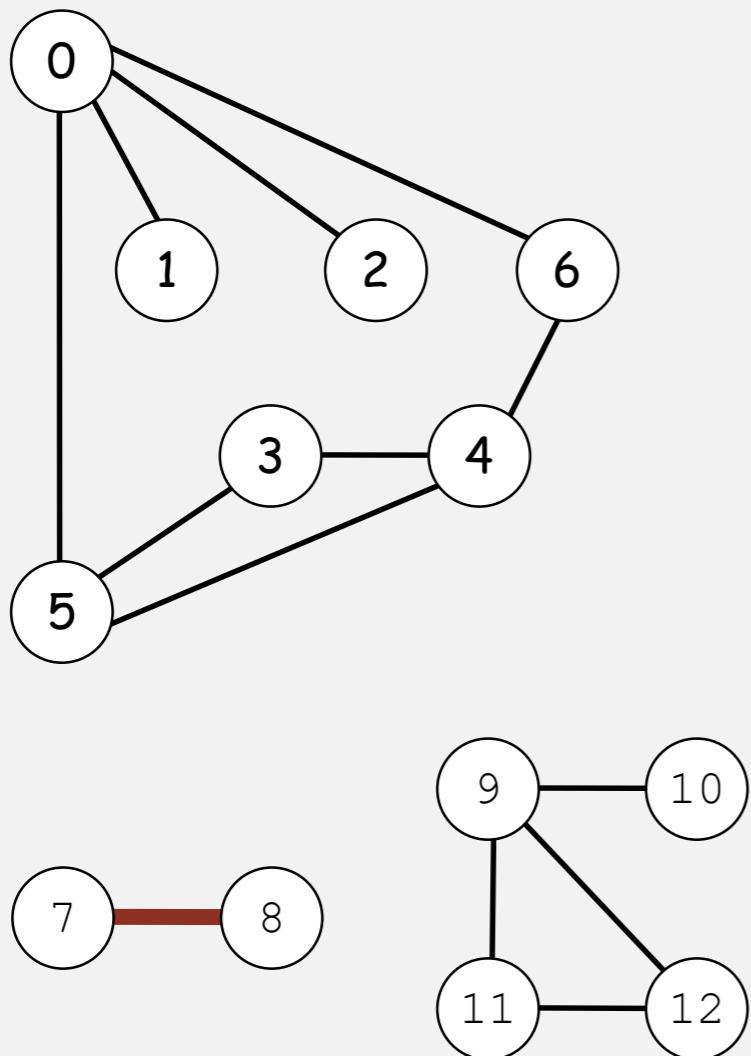


**tinyG.txt**

0-6  
0-2  
0-1  
0-5  
1-0  
2-0  
3-5  
3-4  
...  
12-11  
12-9

# Set-of-edges graph representation

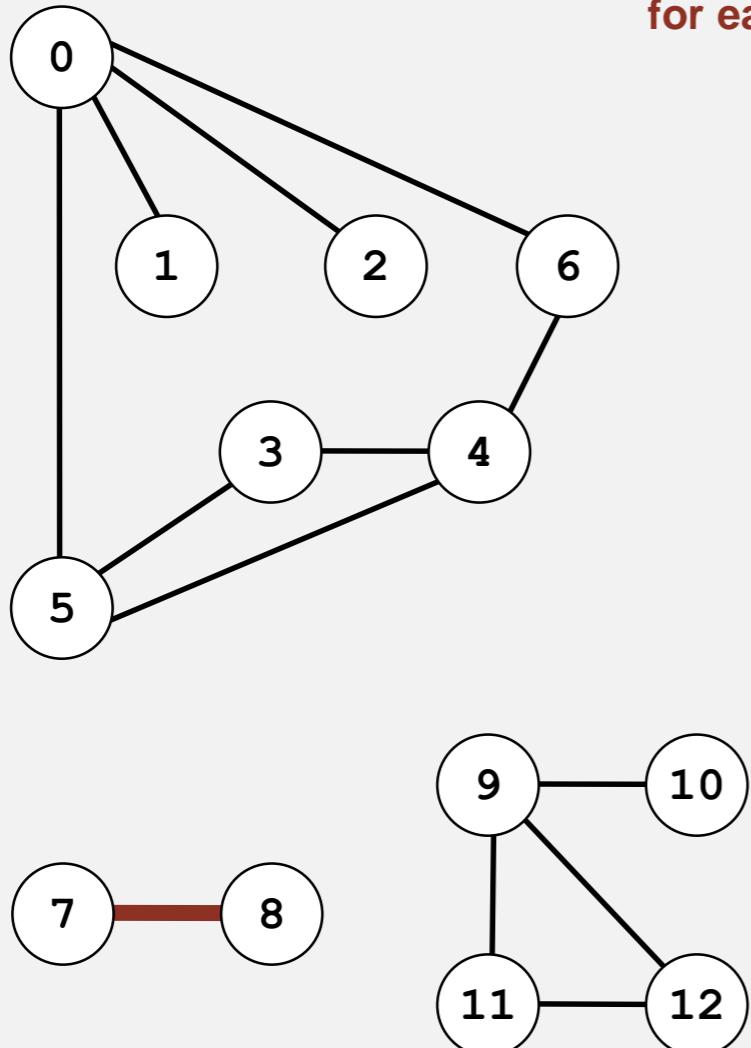
Maintain a list of the edges (linked list or array).



|    |    |
|----|----|
| 0  | 1  |
| 0  | 2  |
| 0  | 5  |
| 0  | 6  |
| 3  | 4  |
| 3  | 5  |
| 4  | 5  |
| 4  | 6  |
| 7  | 8  |
| 9  | 10 |
| 9  | 11 |
| 9  | 12 |
| 11 | 12 |

# Adjacency-matrix graph representation

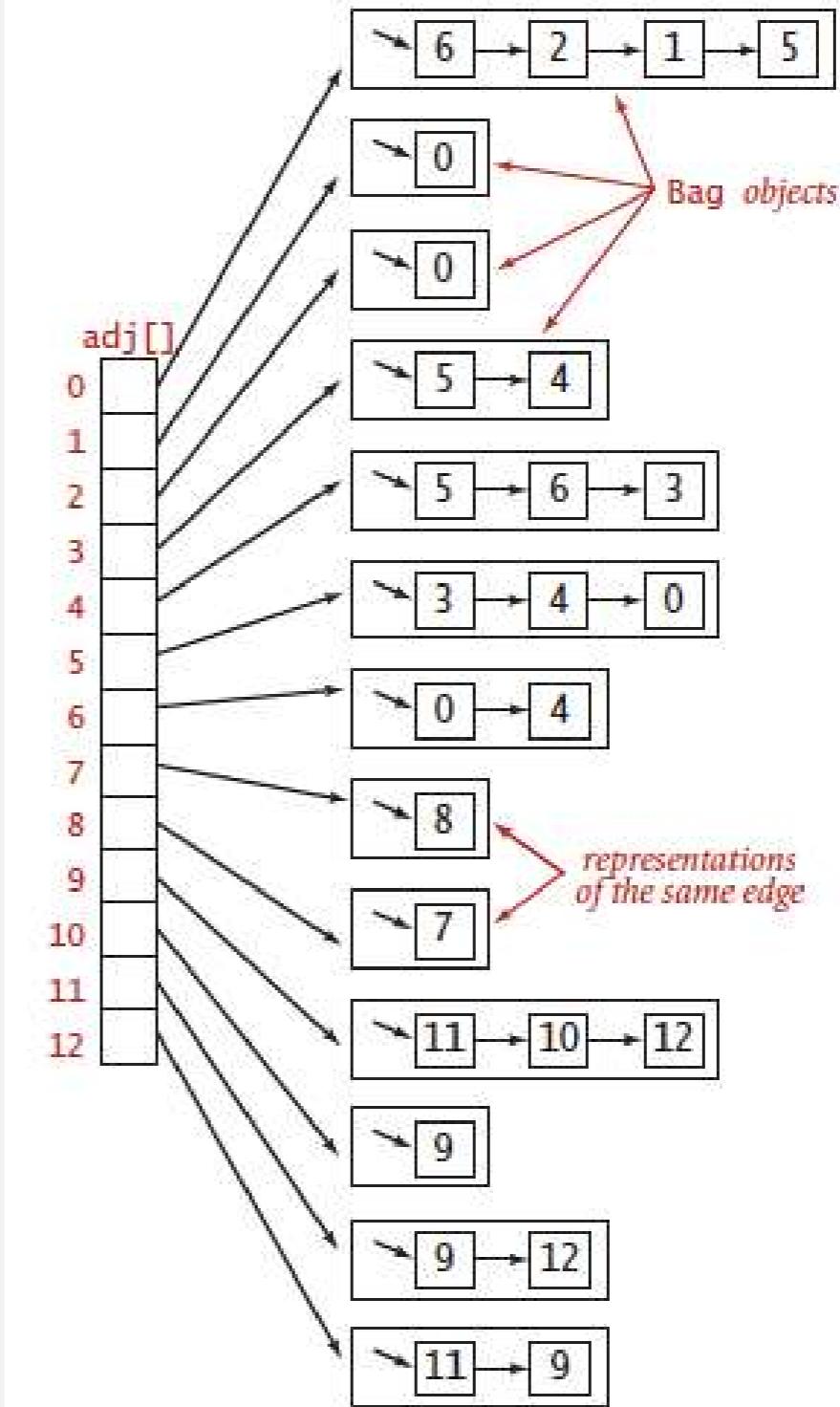
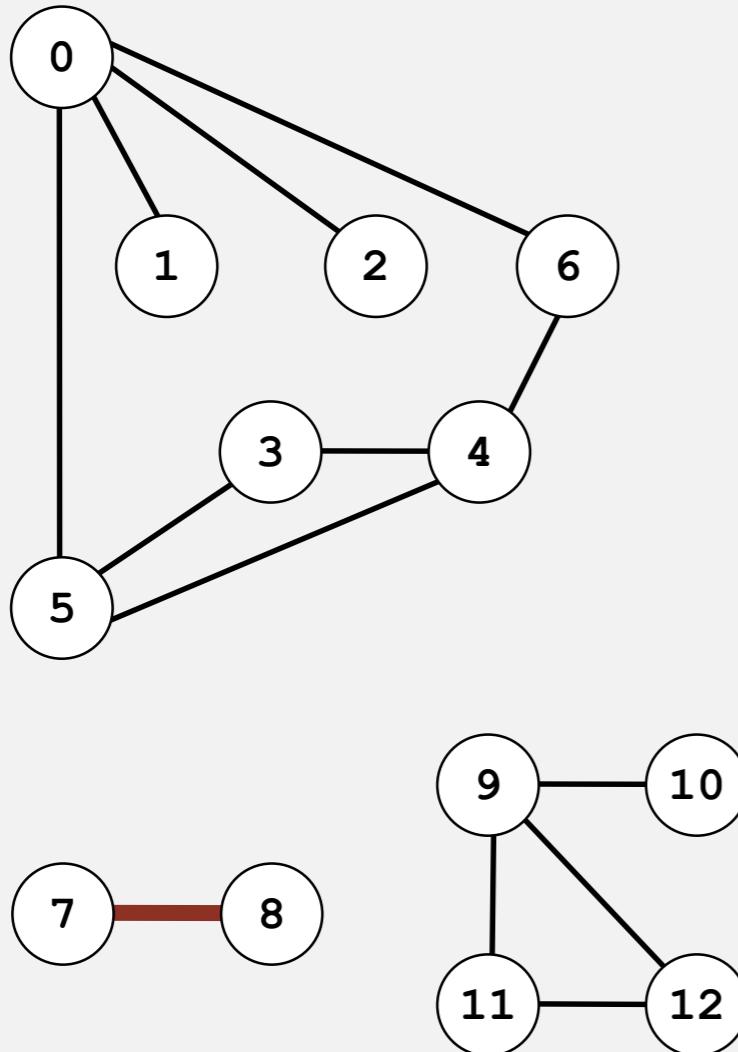
Maintain a two-dimensional  $V$ -by- $V$  boolean array;  
for each edge  $v-w$  in graph:  $\text{adj}[v][w] = \text{adj}[w][v] = \text{true}$ .



| 0  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|----|---|---|---|---|---|---|---|---|---|----|----|----|
| 0  | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0  | 0  | 0  |
| 1  | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0  | 0  | 0  |
| 2  | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0  | 0  | 0  |
| 3  | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0  | 0  | 0  |
| 4  | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0  | 0  | 0  |
| 5  | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0  | 0  | 0  |
| 6  | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0  | 0  | 0  |
| 7  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0  | 0  | 0  |
| 8  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0  | 0  | 0  |
| 9  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0  | 1  | 1  |
| 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0  | 1  | 1  |
| 11 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1  | 0  | 0  |
| 12 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1  | 0  | 0  |
| 12 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1  | 0  | 0  |

# Adjacency-list graph representation

Maintain vertex-indexed array of lists



Adjacency-lists representation (undirected graph)

# Adjacency-list graph representation: Java implementation

```
//for each edge call addEdge twice  
addEdge(v,w);  
addEdge(w,v);  
  
public void addEdge(int v, int w){  
    node *q;  
    //acquire memory for the new node  
    q=(node*)malloc(sizeof(node));  
    q->vertex=w;  
    q->next=NULL;  
    //insert the node to beginning of the  
    linked list  
    q->next=adj[v];  
    adj[v]= q;  
}
```

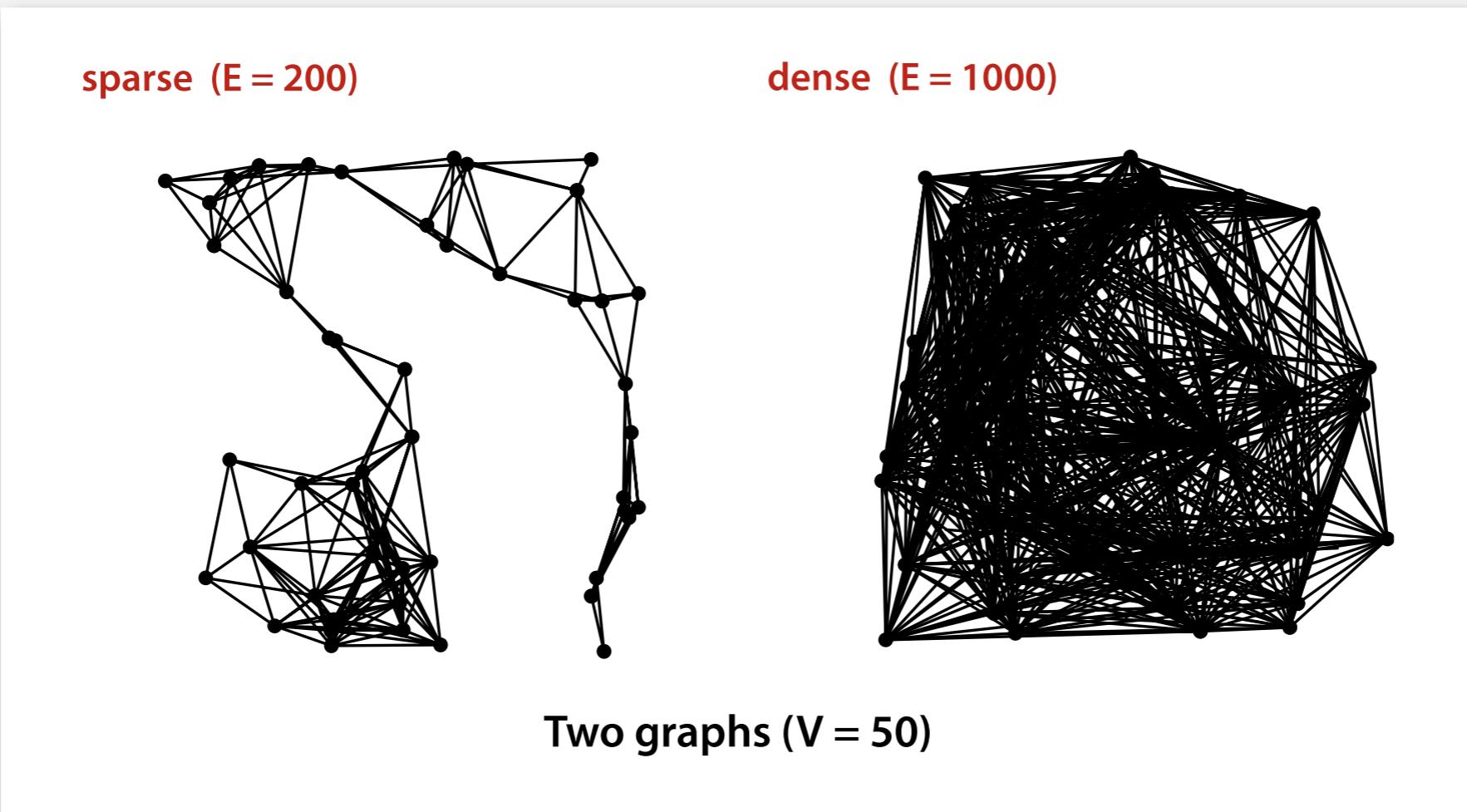
```
typedef struct node  
{  
    struct node *next;  
    int vertex;  
}node;  
  
node * adj [13];
```

# Graph representations

In practice. Use adjacency-lists representation.

- Algorithms based on iterating over vertices adjacent to  $v$ .
- Real-world graphs tend to be **sparse**.

huge number of vertices,  
small average vertex degree



# Graph representations

In practice. Use adjacency-lists representation.

- Algorithms based on iterating over vertices adjacent to  $v$ .
- Real-world graphs tend to be **sparse**.

huge number of vertices,  
small average vertex degree

| representation   | space   | add edge | edge between<br>$v$ and $w$ ? | iterate over vertices<br>adjacent to $v$ ? |
|------------------|---------|----------|-------------------------------|--|
| list of edges    | $E$     | 1        | $E$                           | $E$  |
| adjacency matrix | $V^2$   | 1        | 1                             | $V$  |
| adjacency lists  | $E + V$ | 1        | $\text{degree}(v)$            | $\text{degree}(v)$                         |

# UNDIRECTED GRAPHS

- ▶ **Graph API**
- ▶ **Depth-first search**
- ▶ **Breadth-first search**

# Depth-first search

**Goal.** Systematically search through a graph.

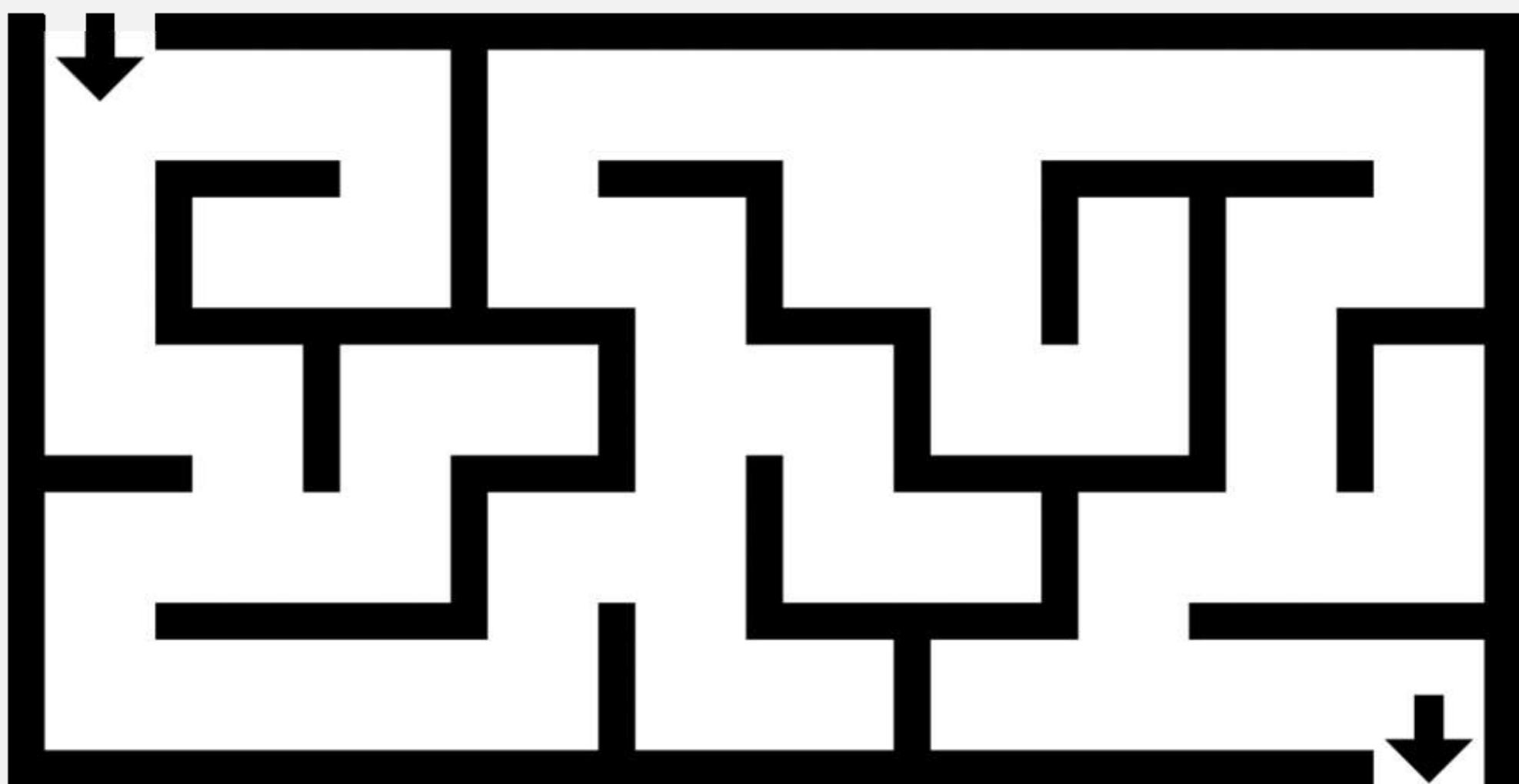
**DFS (to visit a vertex v)**

**Mark v as visited.**

**Recursively visit all unmarked  
vertices w adjacent to v.**

## Typical applications.

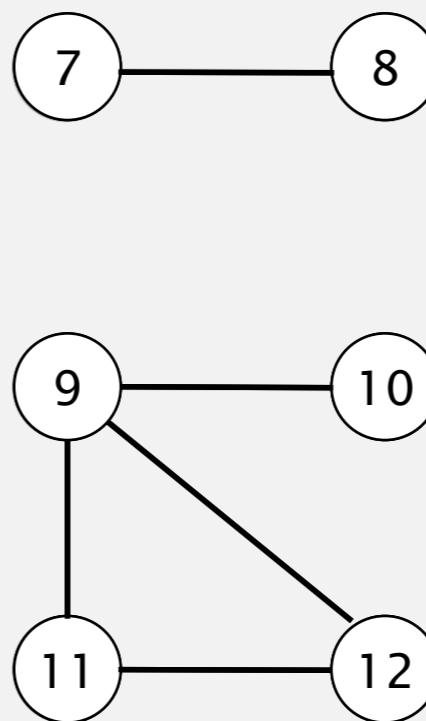
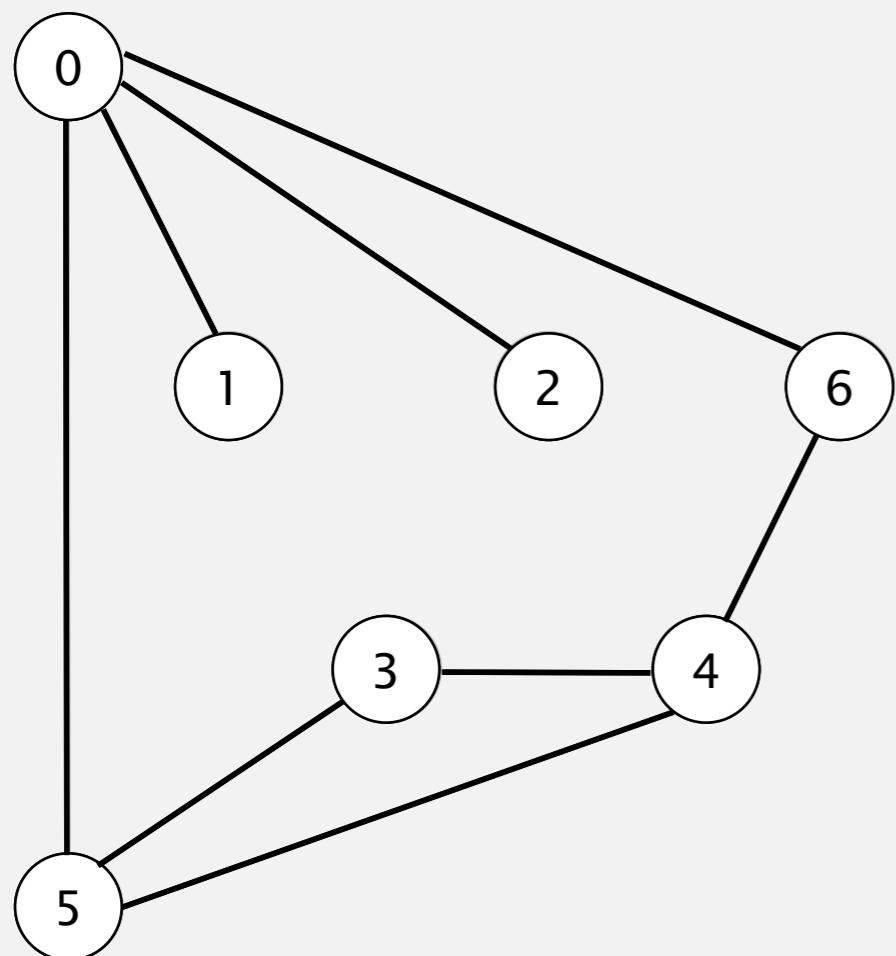
- Find all vertices connected to a given source vertex.
- Find a path between two vertices.



# Depth-first search

To visit a vertex  $v$ :

- Mark vertex  $v$  as visited.
- Recursively visit all unmarked vertices adjacent to  $v$ .



**tinyG.txt**

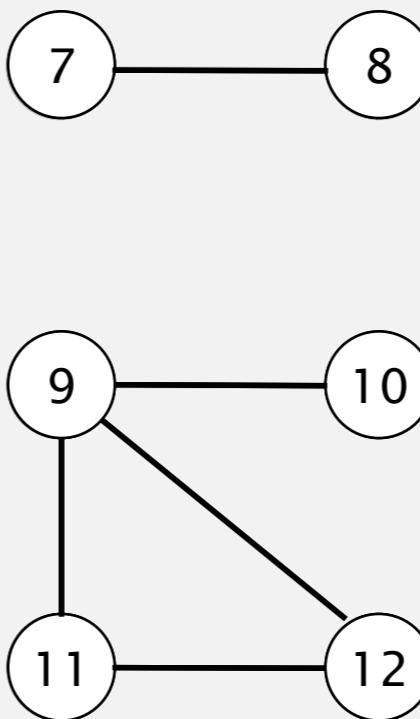
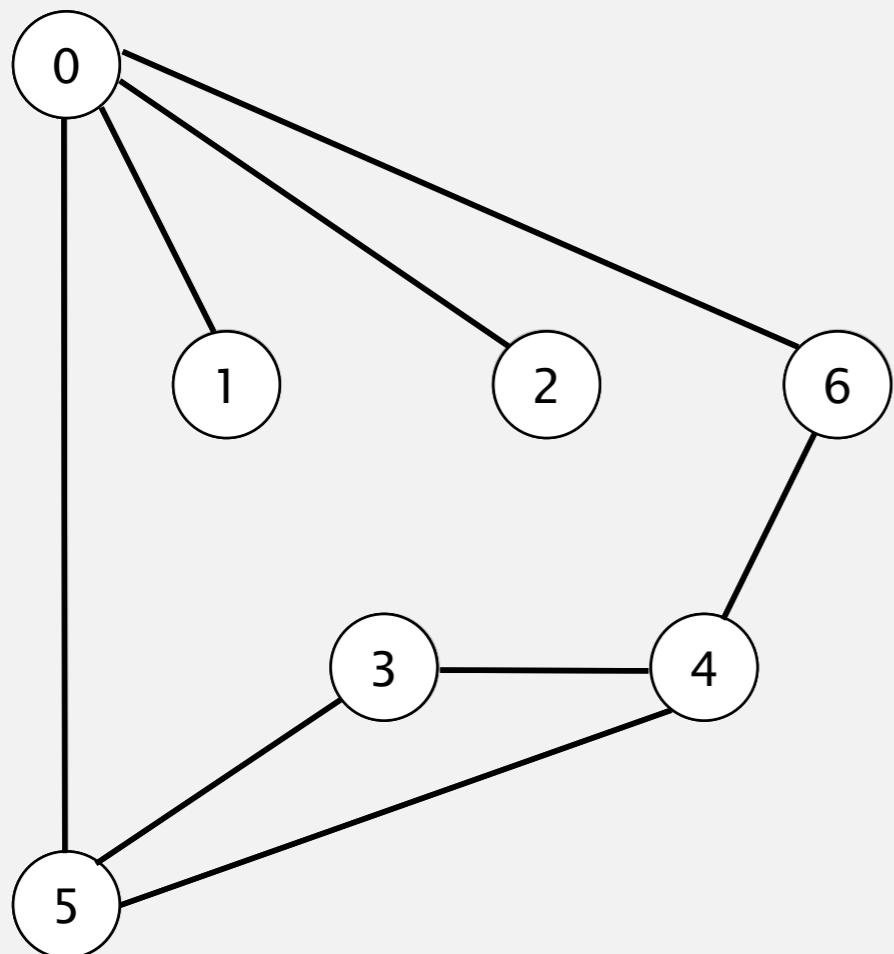
V → 13  
E ← 13

|    |    |
|----|----|
| 0  | 5  |
| 4  | 3  |
| 0  | 1  |
| 9  | 12 |
| 6  | 4  |
| 5  | 4  |
| 0  | 2  |
| 11 | 12 |
| 9  | 10 |
| 0  | 6  |
| 7  | 8  |
| 9  | 11 |
| 5  | 3  |

# Depth-first search

To visit a vertex  $v$ :

- Mark vertex  $v$  as visited.
- Recursively visit all unmarked vertices adjacent to  $v$ .



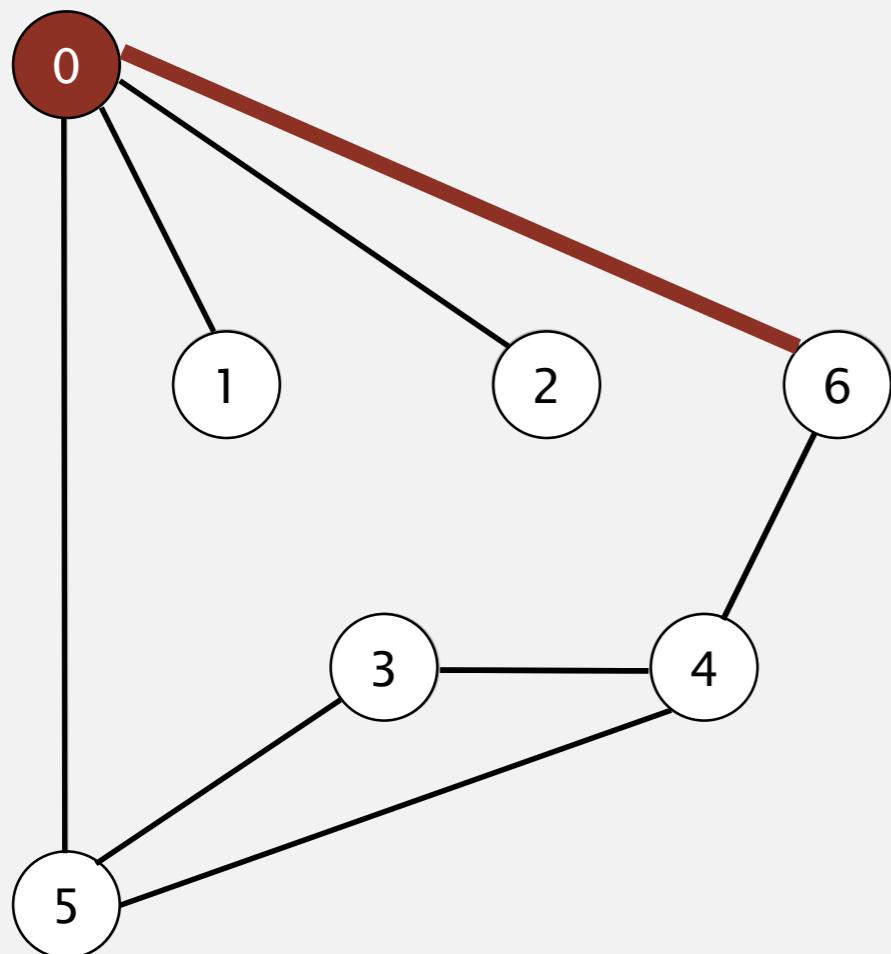
| <b>v</b> | <b>marked[]</b> | <b>edgeTo[v]</b> |
|----------|-----------------|------------------|
| 0        | F               | -                |
| 1        | F               | -                |
| 2        | F               | -                |
| 3        | F               | -                |
| 4        | F               | -                |
| 5        | F               | -                |
| 6        | F               | -                |
| 7        | F               | -                |
| 8        | F               | -                |
| 9        | F               | -                |
| 10       | F               | -                |
| 11       | F               | -                |
| 12       | F               | -                |

**graph G**

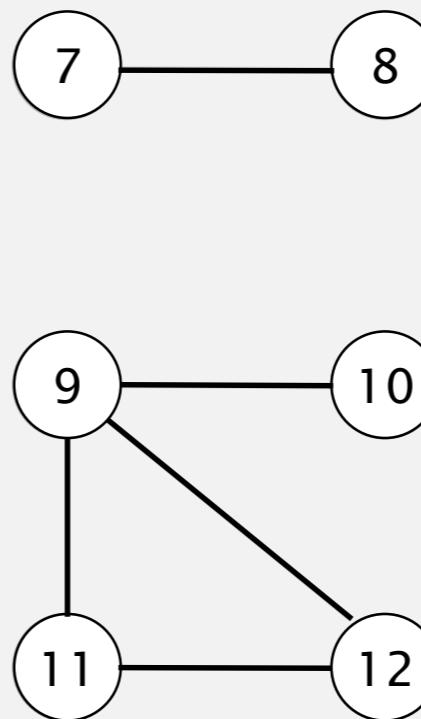
# Depth-first search

To visit a vertex  $v$ :

- Mark vertex  $v$  as visited.
- Recursively visit all unmarked vertices adjacent to  $v$ .



visit 0

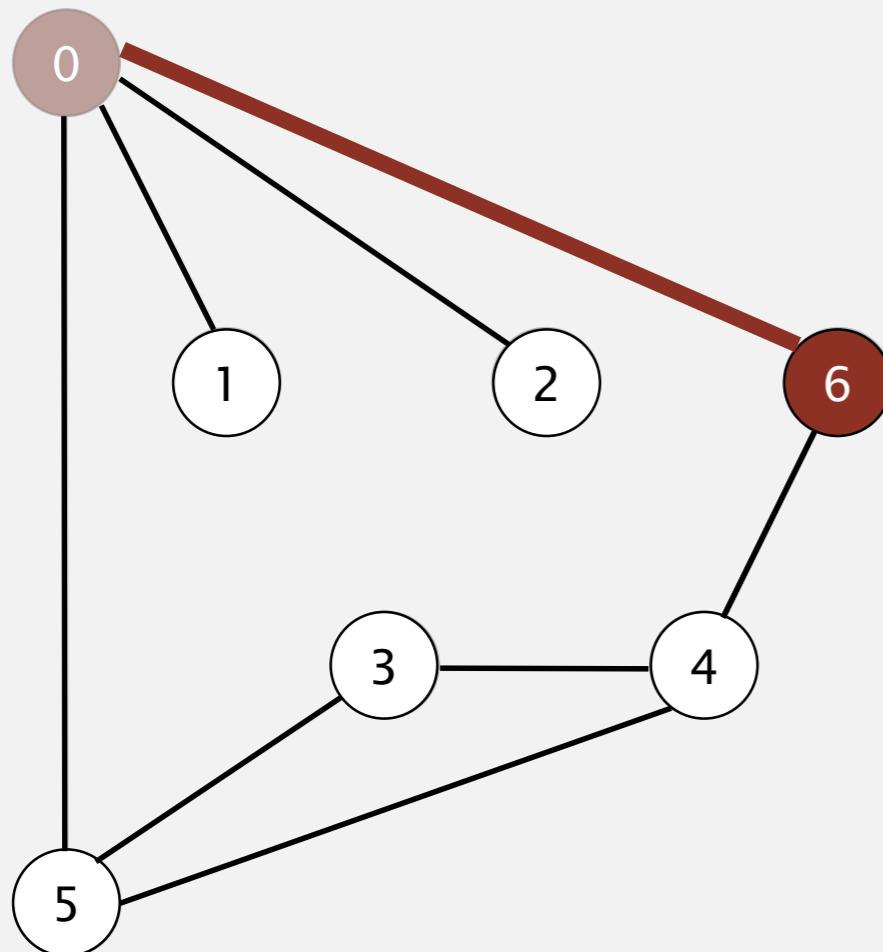


| $v$ | marked[] | edgeTo[ $v$ ] |
|-----|----------|---------------|
| 0   | T        | -             |
| 1   | F        | -             |
| 2   | F        | -             |
| 3   | F        | -             |
| 4   | F        | -             |
| 5   | F        | -             |
| 6   | F        | -             |
| 7   | F        | -             |
| 8   | F        | -             |
| 9   | F        | -             |
| 10  | F        | -             |
| 11  | F        | -             |
| 12  | F        | -             |

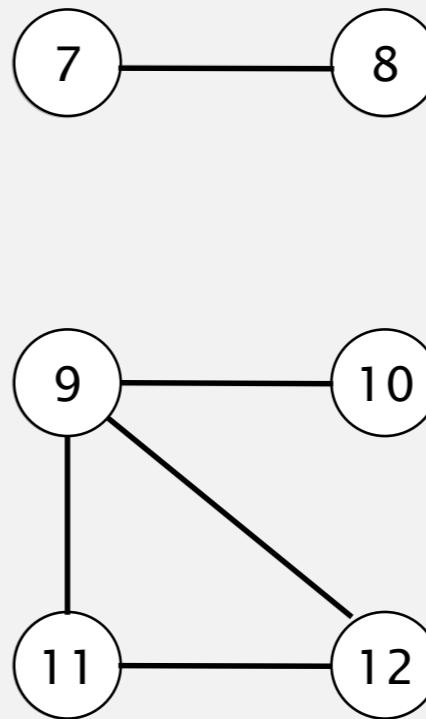
# Depth-first search

To visit a vertex  $v$ :

- Mark vertex  $v$  as visited.
- Recursively visit all unmarked vertices adjacent to  $v$ .



visit 6

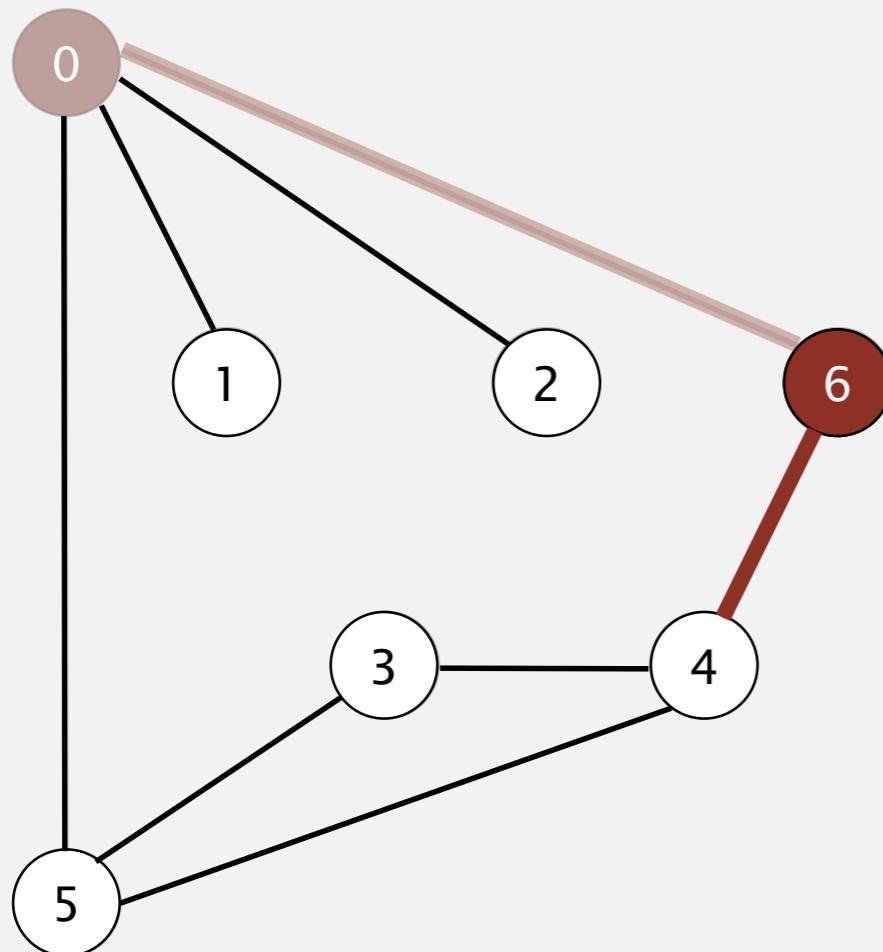


| $v$ | marked[] | edgeTo[v] |
|-----|----------|-----------|
| 0   | T        | -         |
| 1   | F        | -         |
| 2   | F        | -         |
| 3   | F        | -         |
| 4   | F        | -         |
| 5   | F        | -         |
| 6   | T        | 0         |
| 7   | F        | -         |
| 8   | F        | -         |
| 9   | F        | -         |
| 10  | F        | -         |
| 11  | F        | -         |
| 12  | F        | -         |

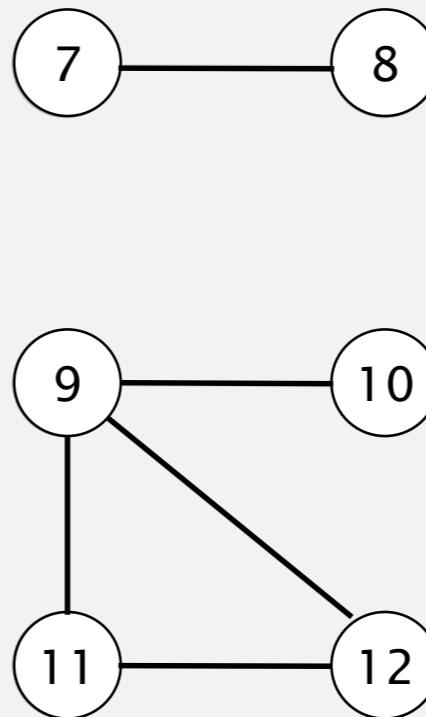
# Depth-first search

To visit a vertex  $v$ :

- Mark vertex  $v$  as visited.
- Recursively visit all unmarked vertices adjacent to  $v$ .



visit 6

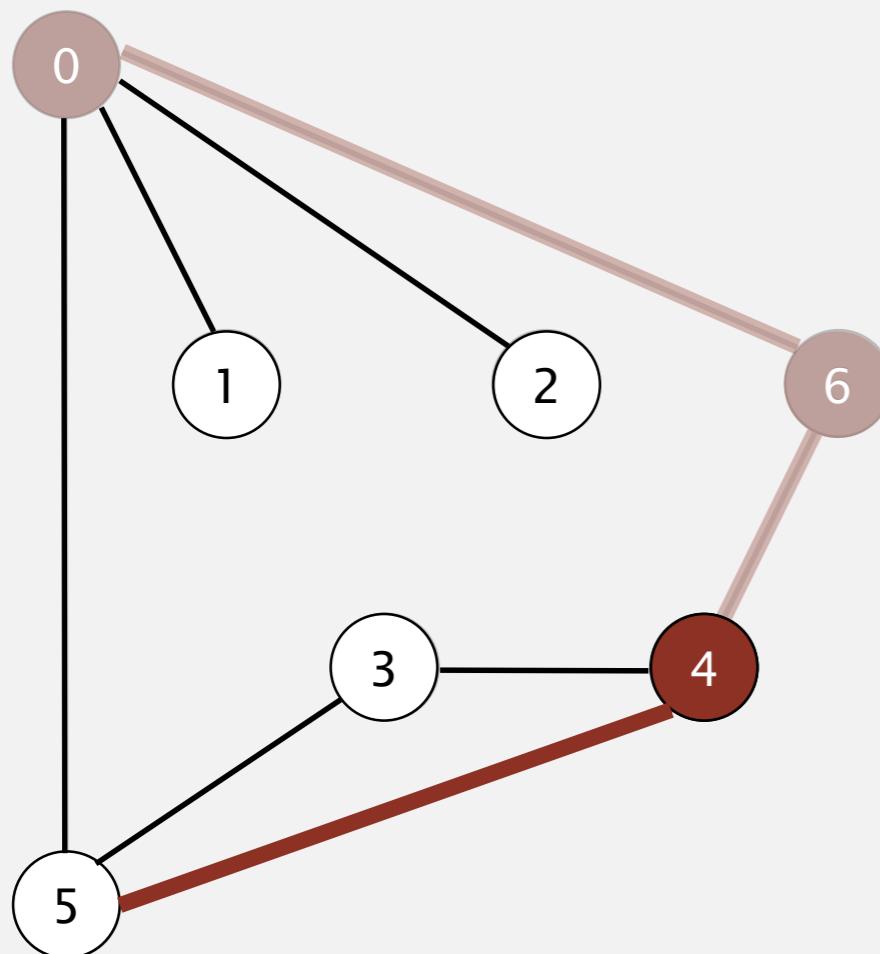


| $v$ | marked[] | edgeTo[v] |
|-----|----------|-----------|
| 0   | T        | -         |
| 1   | F        | -         |
| 2   | F        | -         |
| 3   | F        | -         |
| 4   | F        | -         |
| 5   | F        | -         |
| 6   | T        | 0         |
| 7   | F        | -         |
| 8   | F        | -         |
| 9   | F        | -         |
| 10  | F        | -         |
| 11  | F        | -         |
| 12  | F        | -         |

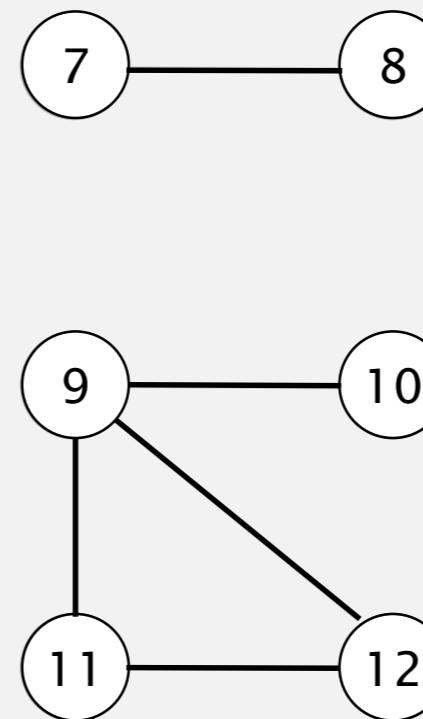
# Depth-first search

To visit a vertex  $v$ :

- Mark vertex  $v$  as visited.
- Recursively visit all unmarked vertices adjacent to  $v$ .



visit 4

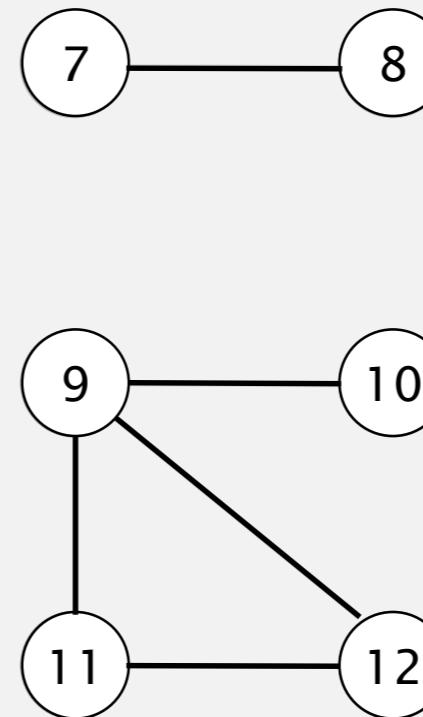
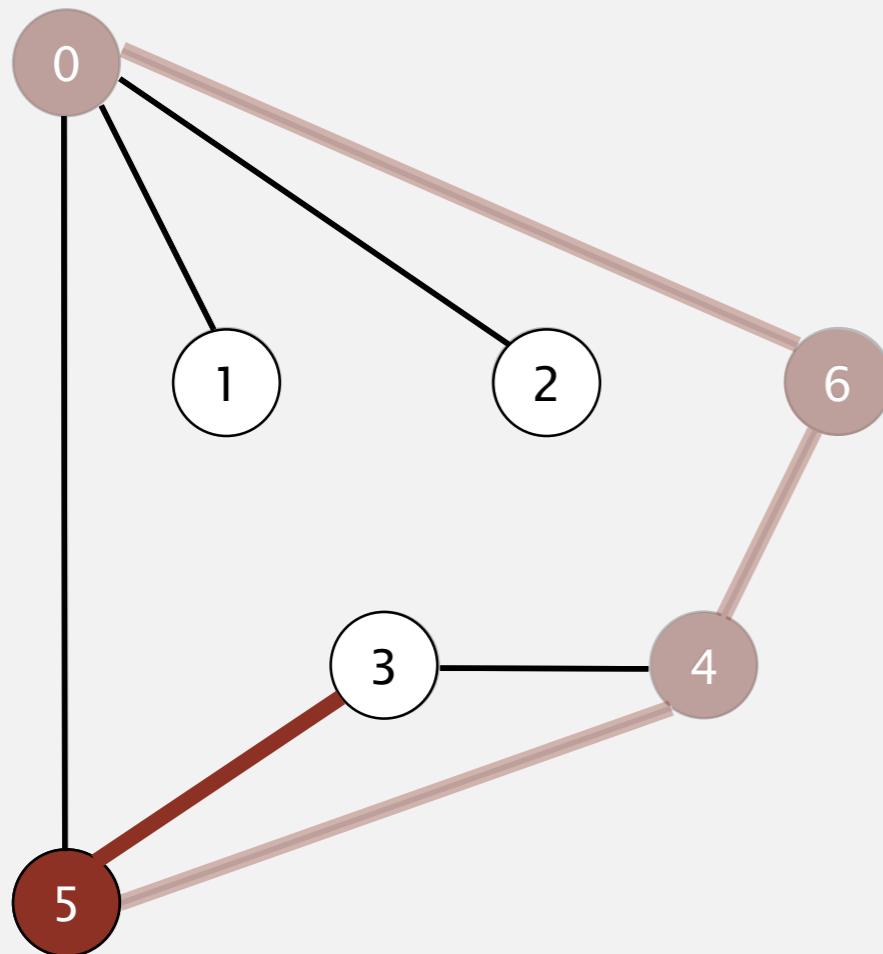


| $v$ | marked[] | edgeTo[v] |
|-----|----------|-----------|
| 0   | T        | -         |
| 1   | F        | -         |
| 2   | F        | -         |
| 3   | F        | -         |
| 4   | T        | 6         |
| 5   | F        | -         |
| 6   | T        | 0         |
| 7   | F        | -         |
| 8   | F        | -         |
| 9   | F        | -         |
| 10  | F        | -         |
| 11  | F        | -         |
| 12  | F        | -         |

# Depth-first search

To visit a vertex  $v$ :

- Mark vertex  $v$  as visited.
- Recursively visit all unmarked vertices adjacent to  $v$ .

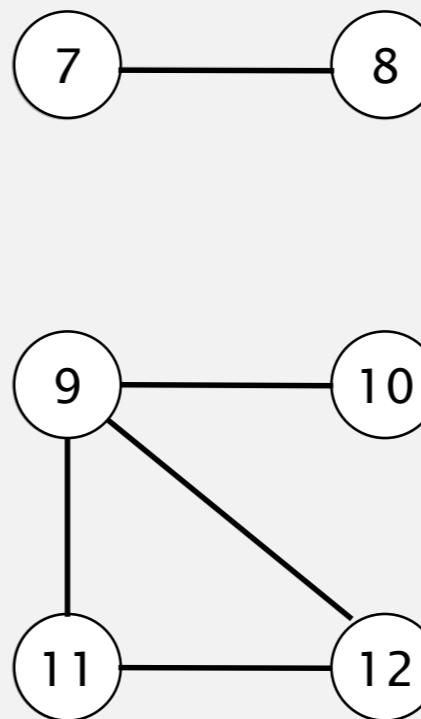
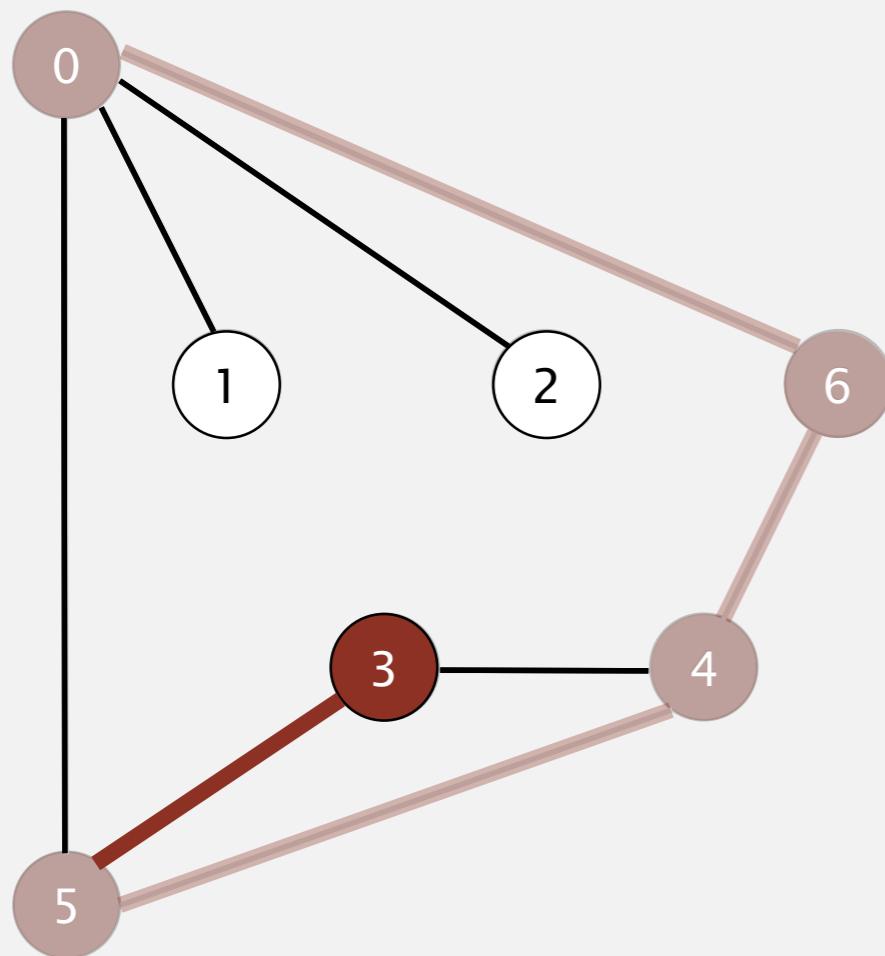


| $v$ | marked[] | edgeTo[v] |
|-----|----------|-----------|
| 0   | T        | -         |
| 1   | F        | -         |
| 2   | F        | -         |
| 3   | F        | -         |
| 4   | T        | 6         |
| 5   | T        | 4         |
| 6   | T        | 0         |
| 7   | F        | -         |
| 8   | F        | -         |
| 9   | F        | -         |
| 10  | F        | -         |
| 11  | F        | -         |
| 12  | F        | -         |

# Depth-first search

To visit a vertex  $v$ :

- Mark vertex  $v$  as visited.
- Recursively visit all unmarked vertices adjacent to  $v$ .

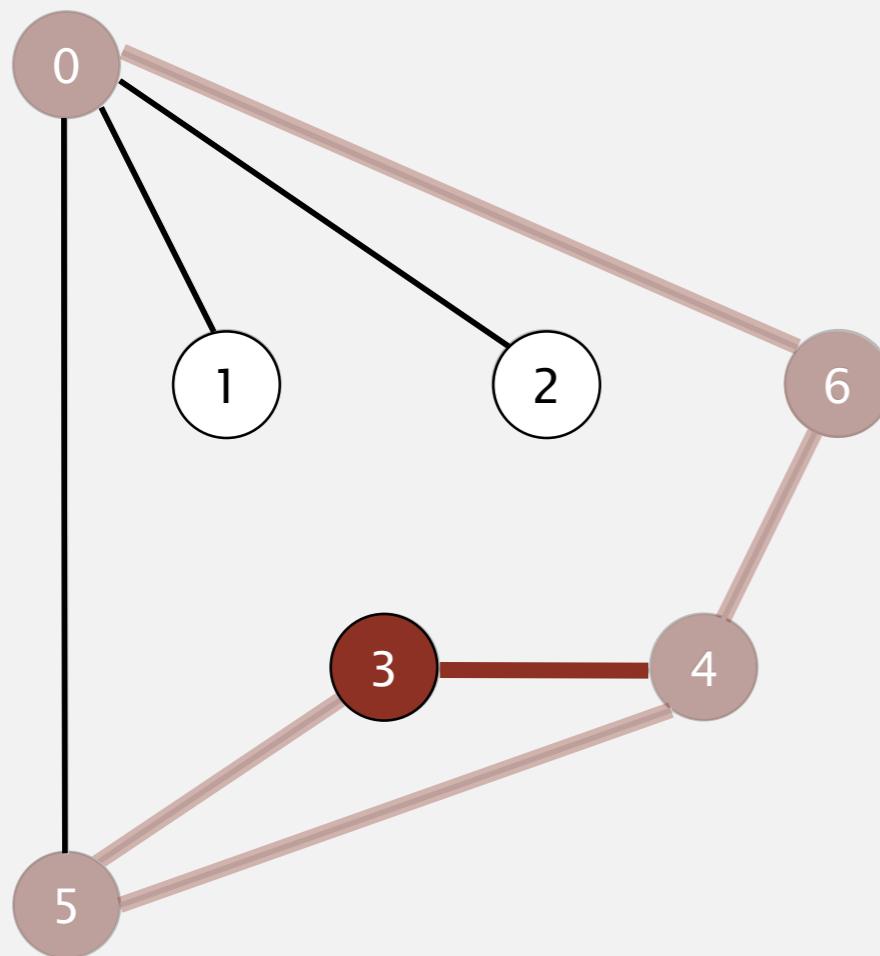


| $v$ | marked[] | edgeTo[v] |
|-----|----------|-----------|
| 0   | T        | -         |
| 1   | F        | -         |
| 2   | F        | -         |
| 3   | T        | 5         |
| 4   | T        | 6         |
| 5   | T        | 4         |
| 6   | T        | 0         |
| 7   | F        | -         |
| 8   | F        | -         |
| 9   | F        | -         |
| 10  | F        | -         |
| 11  | F        | -         |
| 12  | F        | -         |

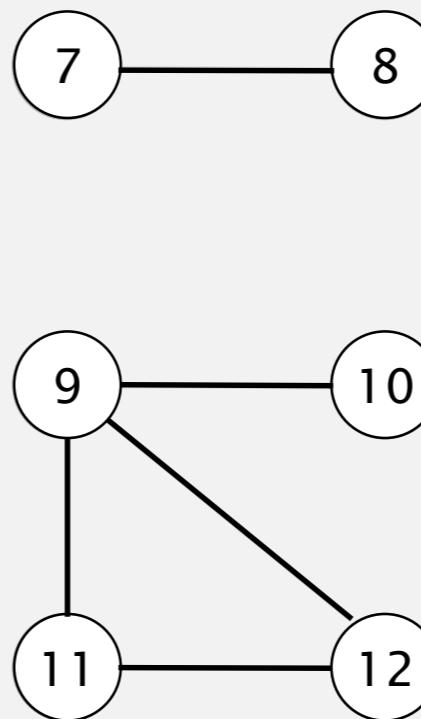
# Depth-first search

To visit a vertex  $v$ :

- Mark vertex  $v$  as visited.
- Recursively visit all unmarked vertices adjacent to  $v$ .



visit 3

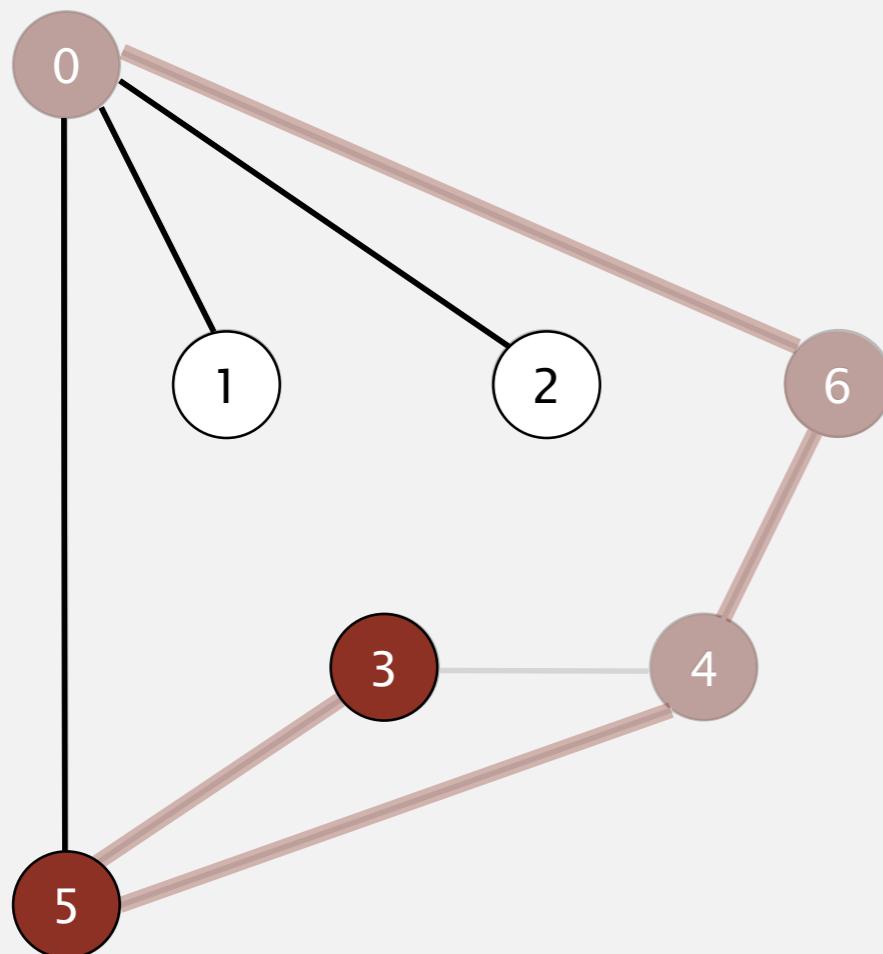


| $v$ | marked[] | edgeTo[v] |
|-----|----------|-----------|
| 0   | T        | -         |
| 1   | F        | -         |
| 2   | F        | -         |
| 3   | T        | 5         |
| 4   | T        | 6         |
| 5   | T        | 4         |
| 6   | T        | 0         |
| 7   | F        | -         |
| 8   | F        | -         |
| 9   | F        | -         |
| 10  | F        | -         |
| 11  | F        | -         |
| 12  | F        | -         |

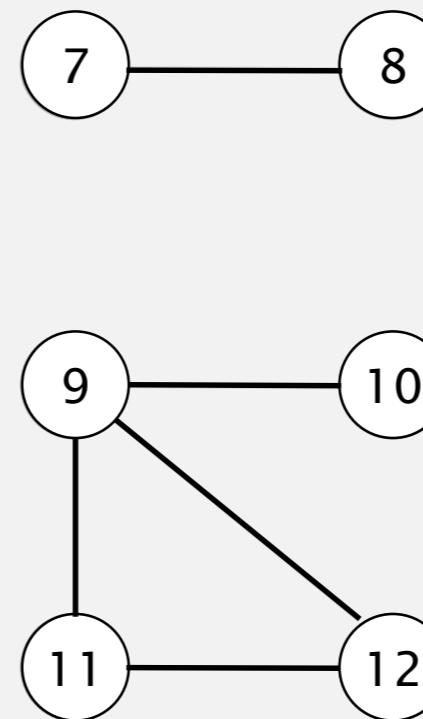
# Depth-first search

To visit a vertex  $v$ :

- Mark vertex  $v$  as visited.
- Recursively visit all unmarked vertices adjacent to  $v$ .



3 done

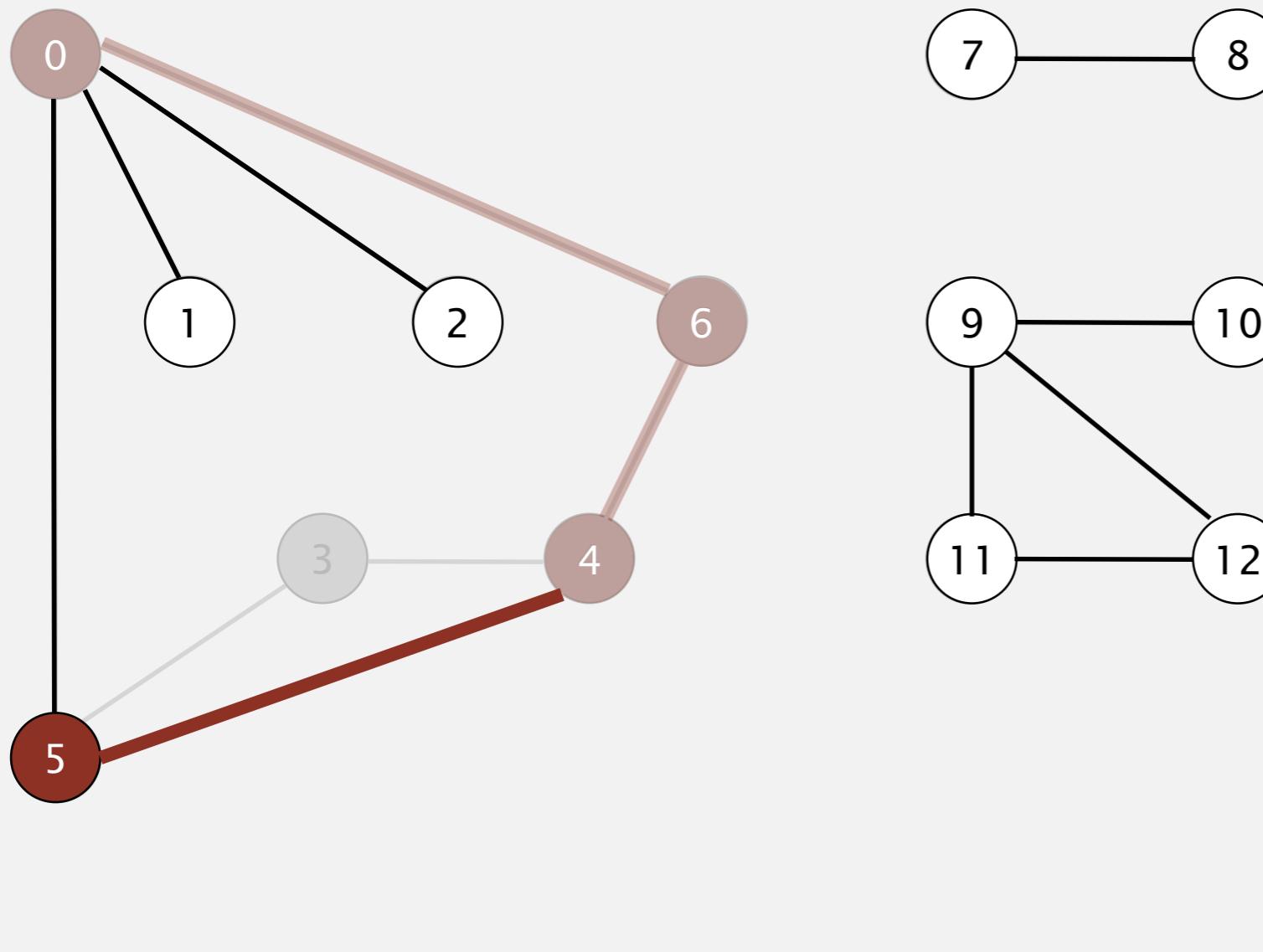


| $v$ | marked[] | edgeTo[ $v$ ] |
|-----|----------|---------------|
| 0   | T        | -             |
| 1   | F        | -             |
| 2   | F        | -             |
| 3   | T        | 5             |
| 4   | T        | 6             |
| 5   | T        | 4             |
| 6   | T        | 0             |
| 7   | F        | -             |
| 8   | F        | -             |
| 9   | F        | -             |
| 10  | F        | -             |
| 11  | F        | -             |
| 12  | F        | -             |

# Depth-first search

To visit a vertex  $v$ :

- Mark vertex  $v$  as visited.
- Recursively visit all unmarked vertices adjacent to  $v$ .

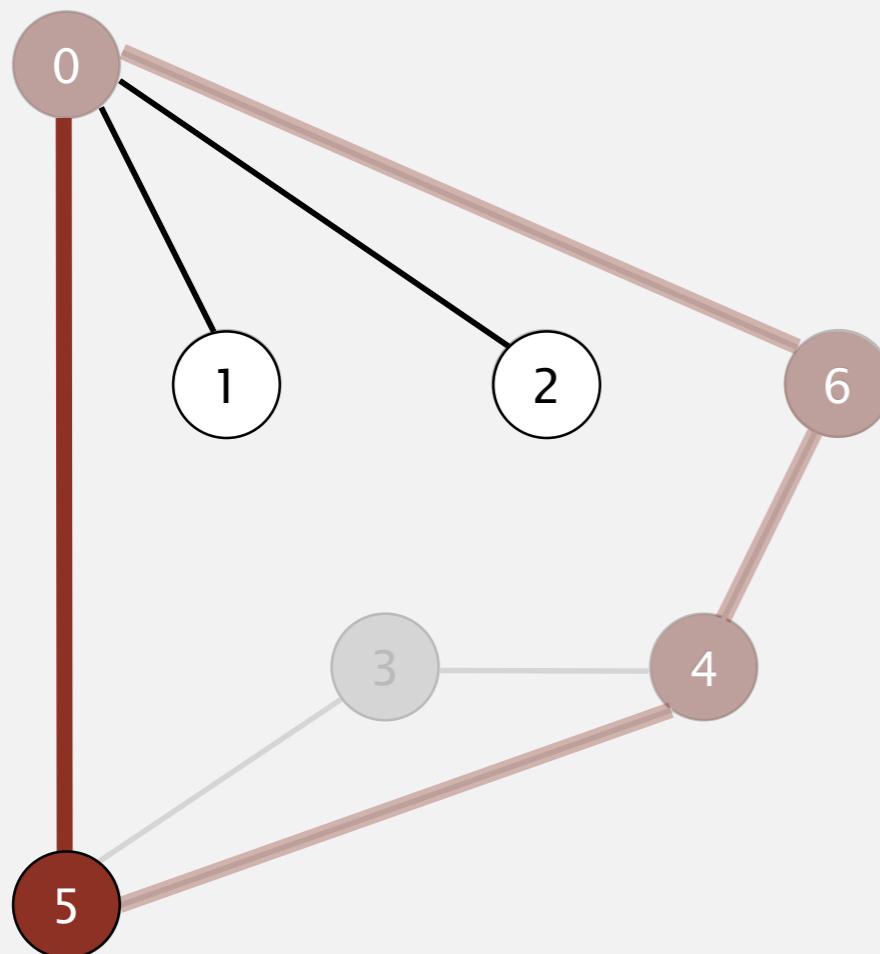


| $v$ | marked[] | edgeTo[v] |
|-----|----------|-----------|
| 0   | T        | -         |
| 1   | F        | -         |
| 2   | F        | -         |
| 3   | T        | 5         |
| 4   | T        | 6         |
| 5   | T        | 4         |
| 6   | T        | 0         |
| 7   | F        | -         |
| 8   | F        | -         |
| 9   | F        | -         |
| 10  | F        | -         |
| 11  | F        | -         |
| 12  | F        | -         |

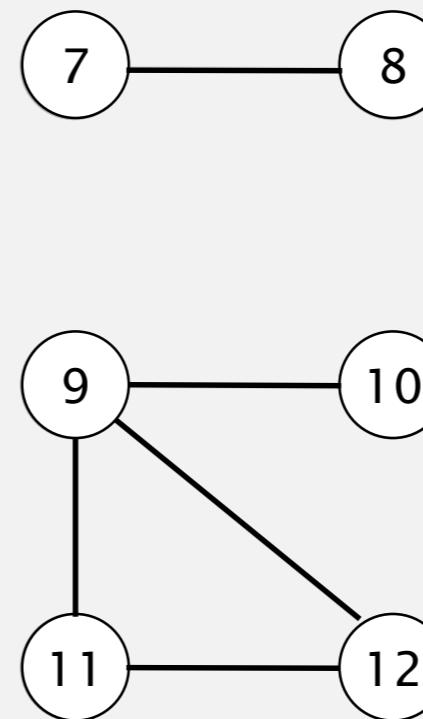
# Depth-first search

To visit a vertex  $v$ :

- Mark vertex  $v$  as visited.
- Recursively visit all unmarked vertices adjacent to  $v$ .



visit 5

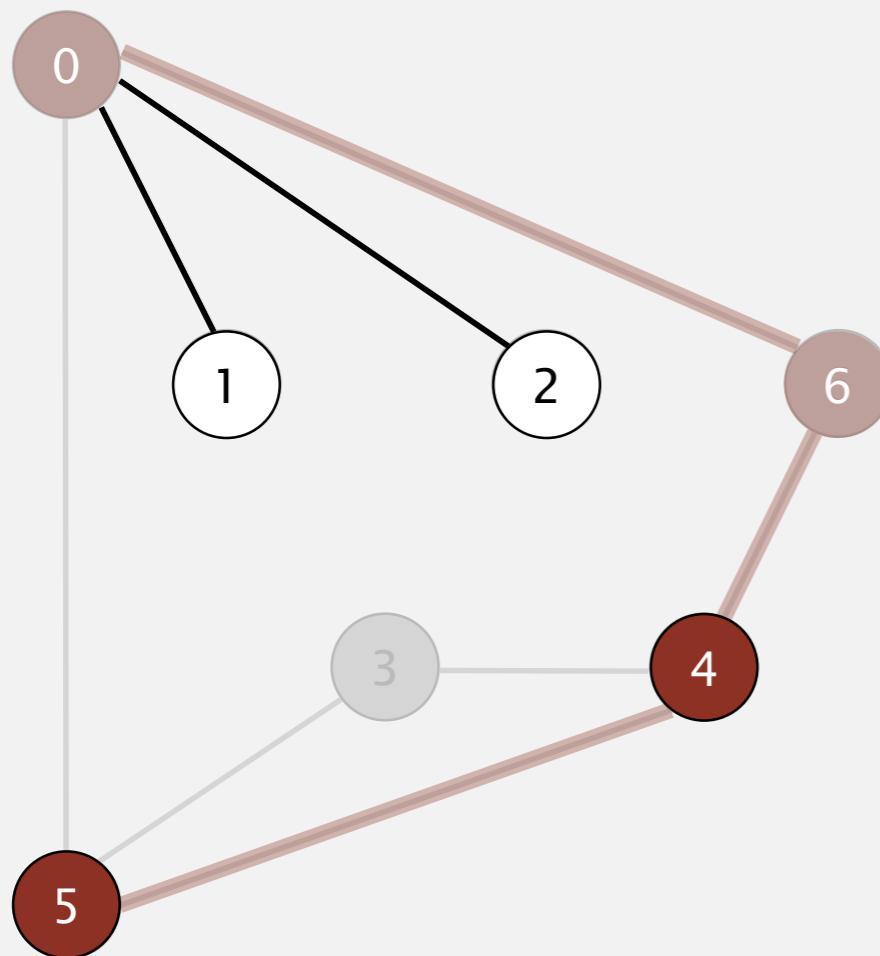


| $v$ | marked[] | edgeTo[v] |
|-----|----------|-----------|
| 0   | T        | -         |
| 1   | F        | -         |
| 2   | F        | -         |
| 3   | T        | 5         |
| 4   | T        | 6         |
| 5   | T        | 4         |
| 6   | T        | 0         |
| 7   | F        | -         |
| 8   | F        | -         |
| 9   | F        | -         |
| 10  | F        | -         |
| 11  | F        | -         |
| 12  | F        | -         |

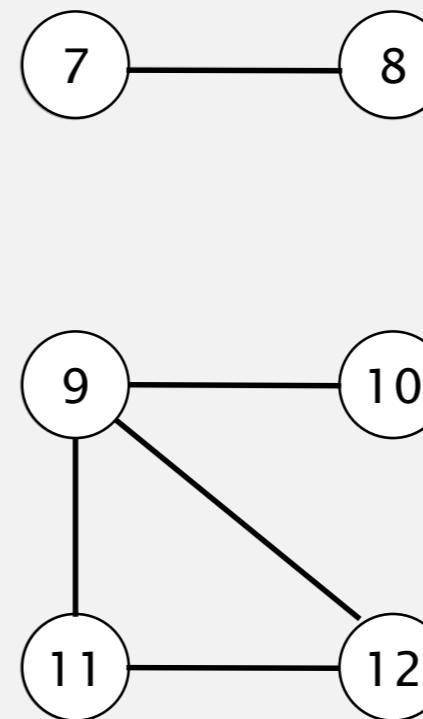
# Depth-first search

To visit a vertex  $v$ :

- Mark vertex  $v$  as visited.
- Recursively visit all unmarked vertices adjacent to  $v$ .



5 done

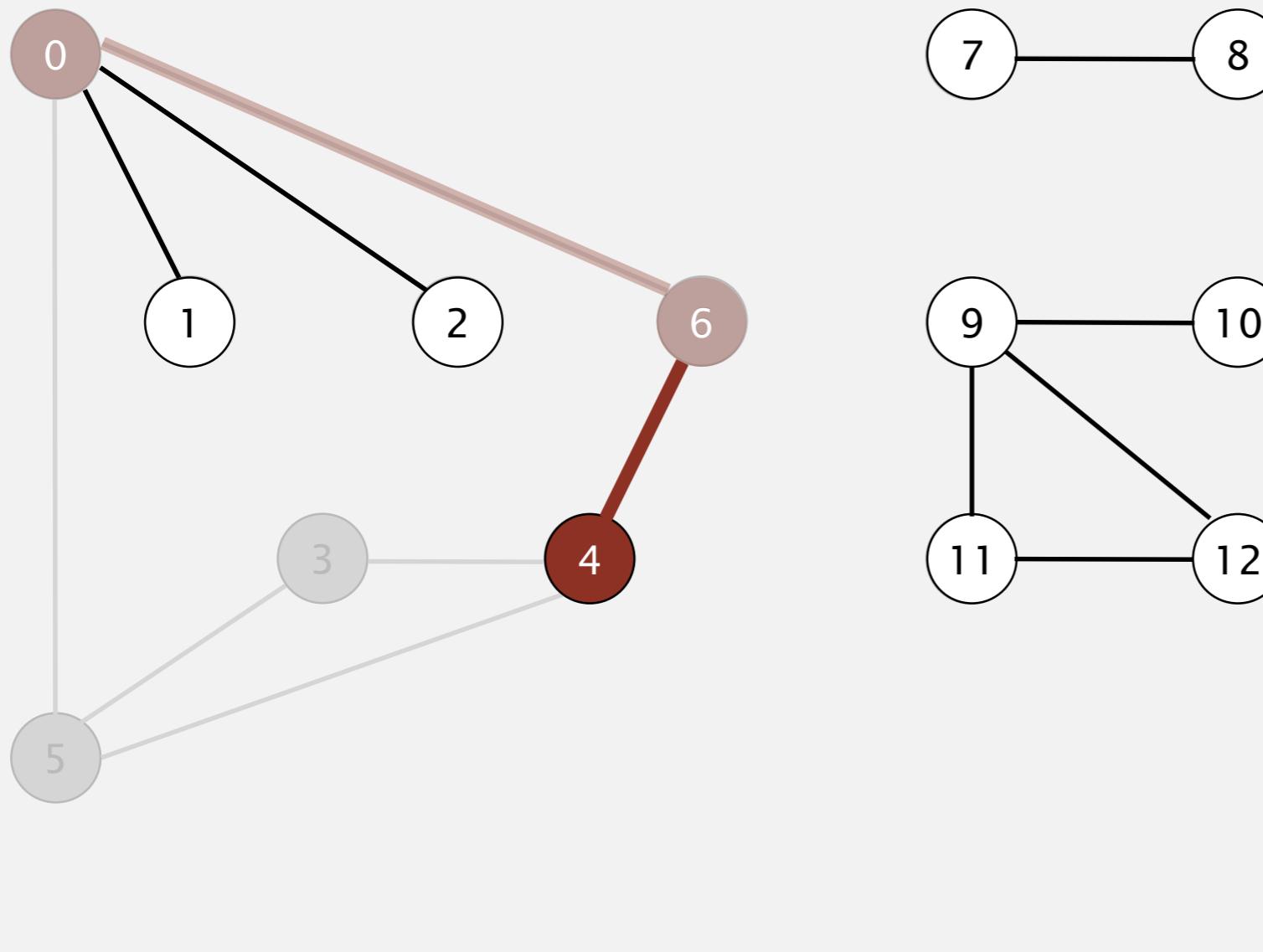


| $v$ | marked[] | edgeTo[ $v$ ] |
|-----|----------|---------------|
| 0   | T        | -             |
| 1   | F        | -             |
| 2   | F        | -             |
| 3   | T        | 5             |
| 4   | T        | 6             |
| 5   | T        | 4             |
| 6   | T        | 0             |
| 7   | F        | -             |
| 8   | F        | -             |
| 9   | F        | -             |
| 10  | F        | -             |
| 11  | F        | -             |
| 12  | F        | -             |

# Depth-first search

To visit a vertex  $v$ :

- Mark vertex  $v$  as visited.
- Recursively visit all unmarked vertices adjacent to  $v$ .

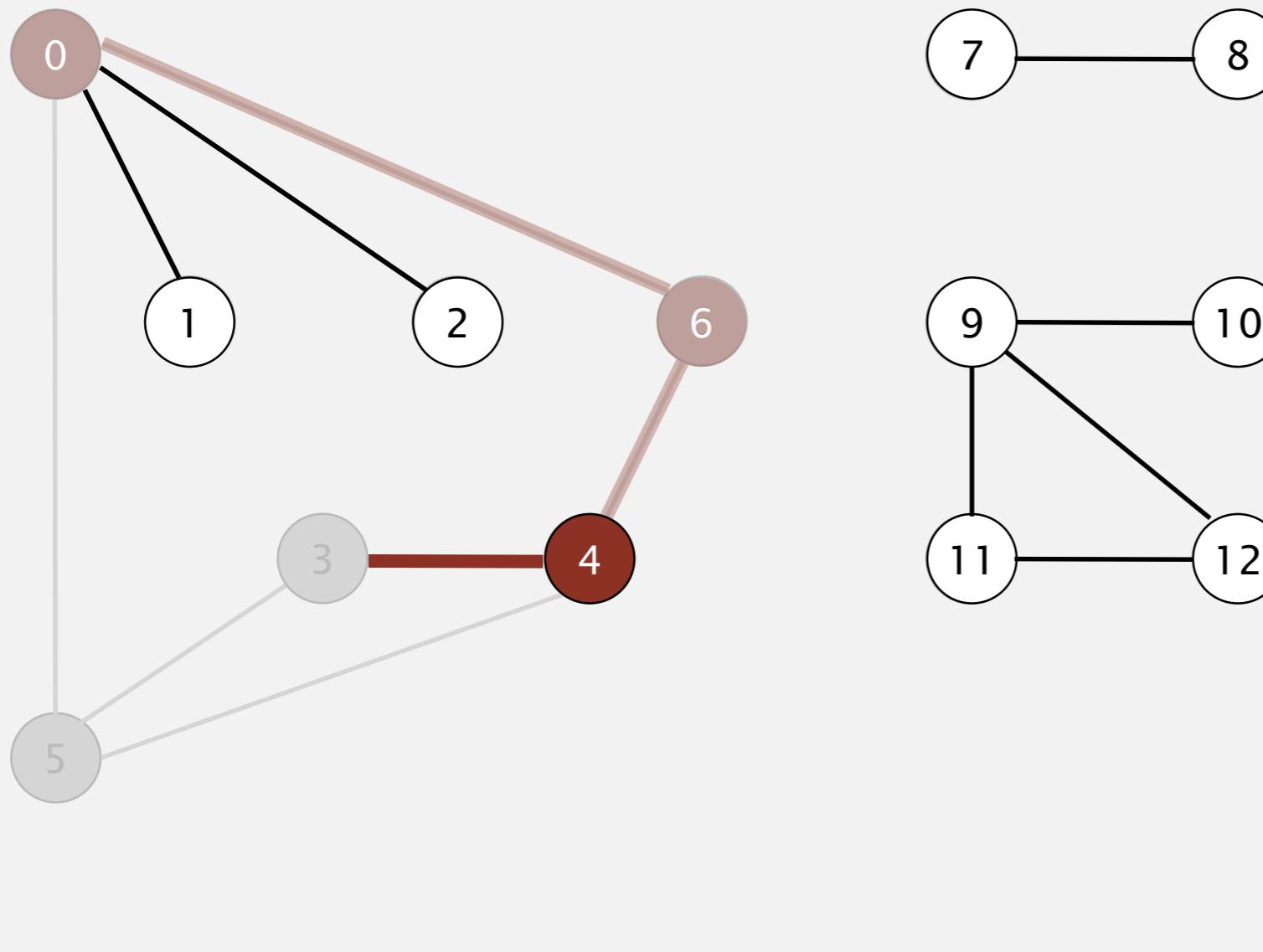


| $v$ | marked[] | edgeTo[v] |
|-----|----------|-----------|
| 0   | T        | -         |
| 1   | F        | -         |
| 2   | F        | -         |
| 3   | T        | 5         |
| 4   | T        | 6         |
| 5   | T        | 4         |
| 6   | T        | 0         |
| 7   | F        | -         |
| 8   | F        | -         |
| 9   | F        | -         |
| 10  | F        | -         |
| 11  | F        | -         |
| 12  | F        | -         |

# Depth-first search

To visit a vertex  $v$ :

- Mark vertex  $v$  as visited.
- Recursively visit all unmarked vertices adjacent to  $v$ .

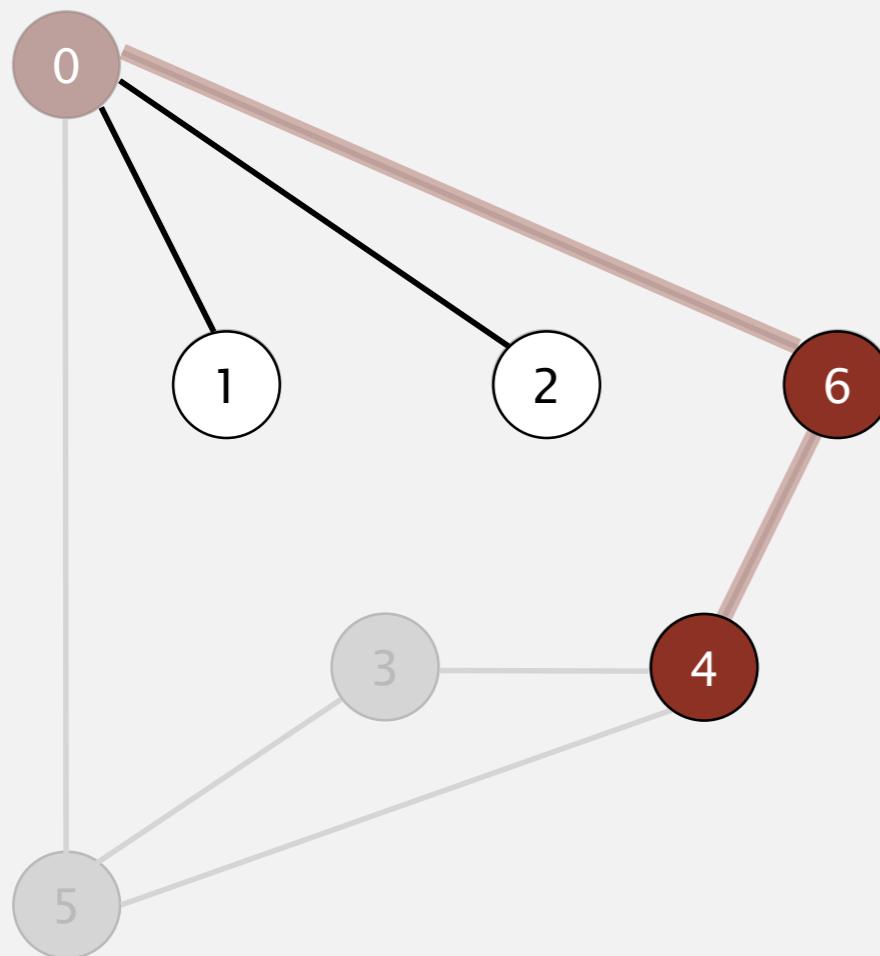


| $v$ | marked[] | edgeTo[v] |
|-----|----------|-----------|
| 0   | T        | -         |
| 1   | F        | -         |
| 2   | F        | -         |
| 3   | T        | 5         |
| 4   | T        | 6         |
| 5   | T        | 4         |
| 6   | T        | 0         |
| 7   | F        | -         |
| 8   | F        | -         |
| 9   | F        | -         |
| 10  | F        | -         |
| 11  | F        | -         |
| 12  | F        | -         |

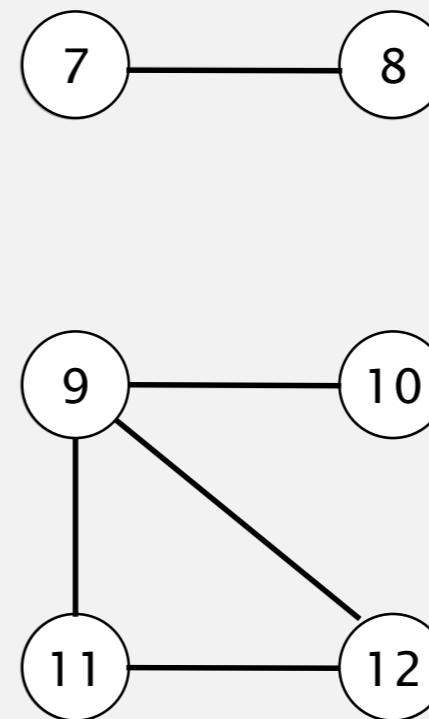
# Depth-first search

To visit a vertex  $v$ :

- Mark vertex  $v$  as visited.
- Recursively visit all unmarked vertices adjacent to  $v$ .



4 done

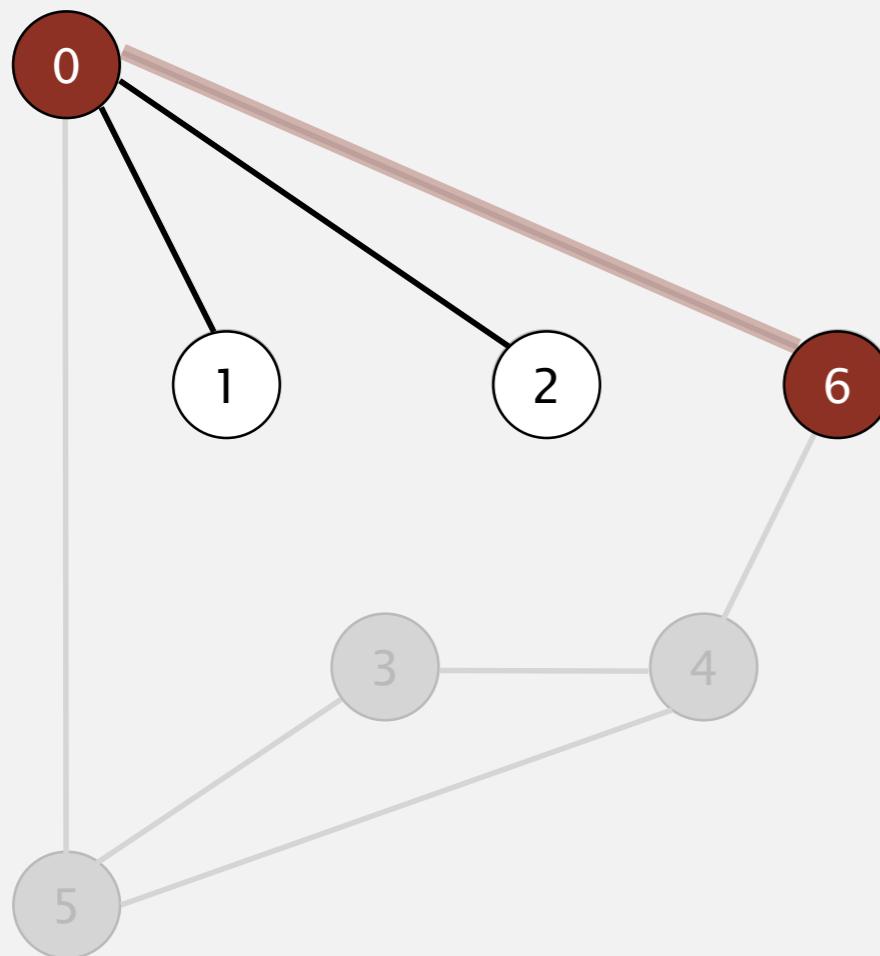


| $v$ | marked[] | edgeTo[v] |
|-----|----------|-----------|
| 0   | T        | -         |
| 1   | F        | -         |
| 2   | F        | -         |
| 3   | T        | 5         |
| 4   | T        | 6         |
| 5   | T        | 4         |
| 6   | T        | 0         |
| 7   | F        | -         |
| 8   | F        | -         |
| 9   | F        | -         |
| 10  | F        | -         |
| 11  | F        | -         |
| 12  | F        | -         |

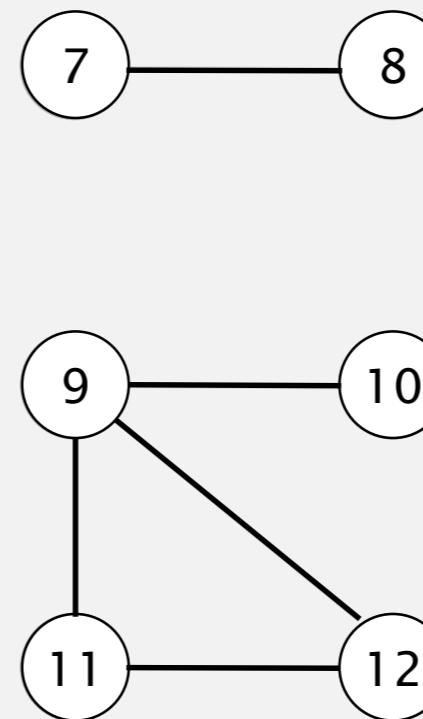
# Depth-first search

To visit a vertex  $v$ :

- Mark vertex  $v$  as visited.
- Recursively visit all unmarked vertices adjacent to  $v$ .



6 done

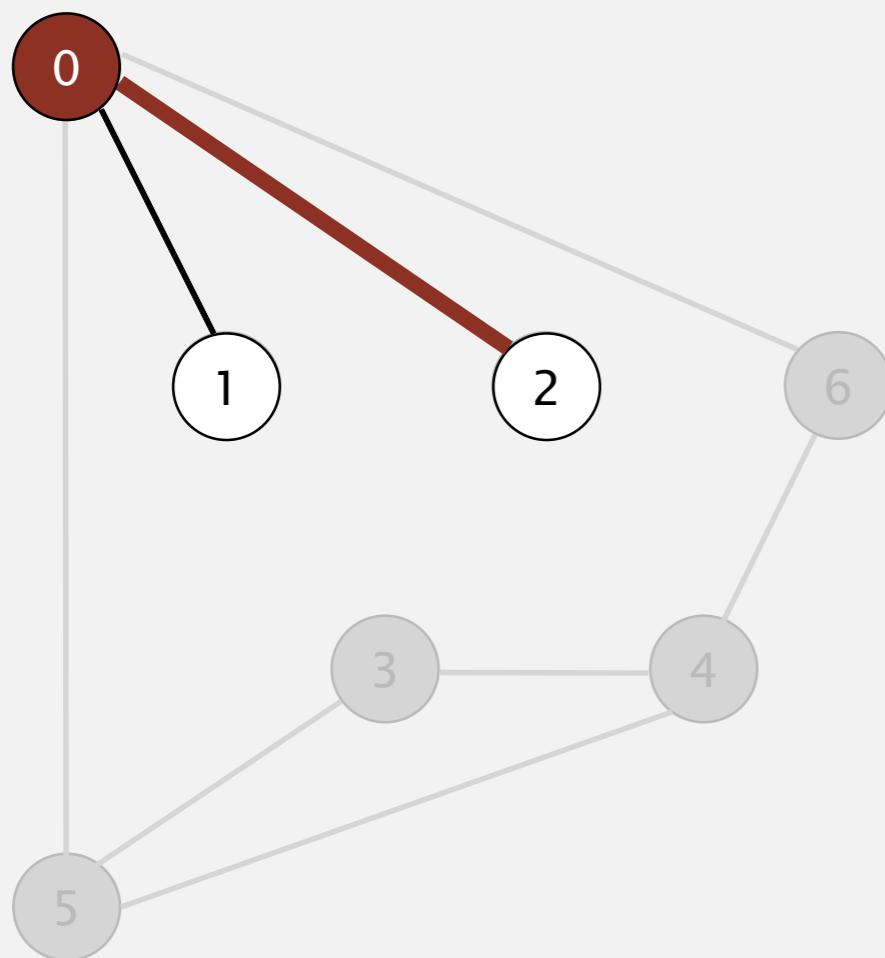


| $v$ | marked[] | edgeTo[v] |
|-----|----------|-----------|
| 0   | T        | -         |
| 1   | F        | -         |
| 2   | F        | -         |
| 3   | T        | 5         |
| 4   | T        | 6         |
| 5   | T        | 4         |
| 6   | T        | 0         |
| 7   | F        | -         |
| 8   | F        | -         |
| 9   | F        | -         |
| 10  | F        | -         |
| 11  | F        | -         |
| 12  | F        | -         |

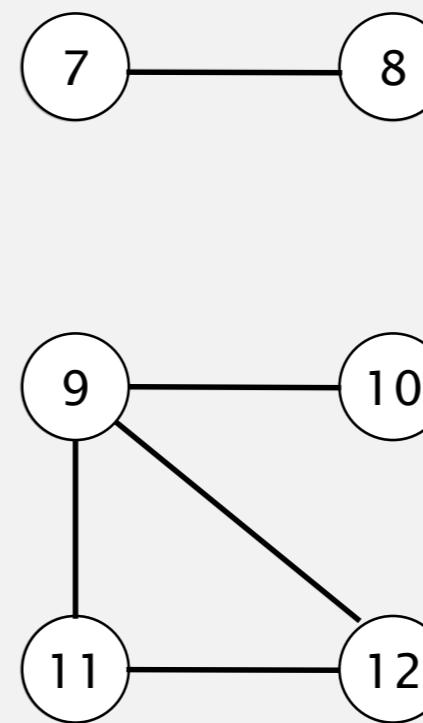
# Depth-first search

To visit a vertex  $v$ :

- Mark vertex  $v$  as visited.
- Recursively visit all unmarked vertices adjacent to  $v$ .



visit 0

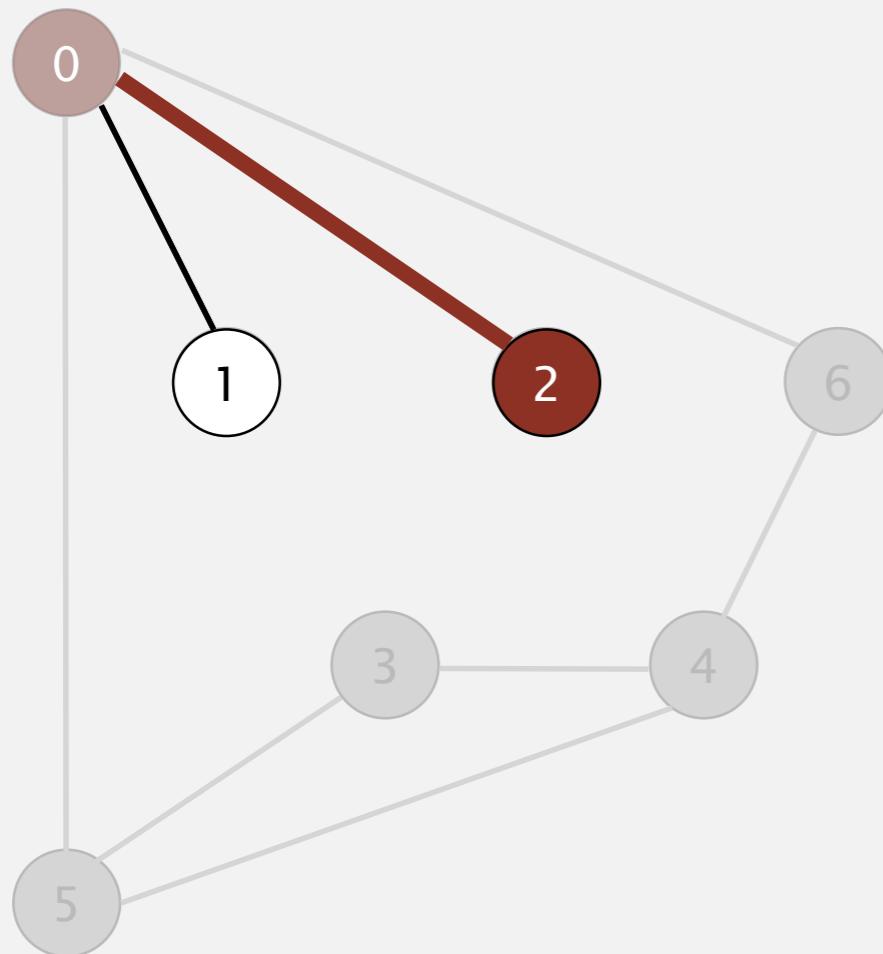


| $v$ | marked[] | edgeTo[v] |
|-----|----------|-----------|
| 0   | T        | -         |
| 1   | F        | -         |
| 2   | F        | -         |
| 3   | T        | 5         |
| 4   | T        | 6         |
| 5   | T        | 4         |
| 6   | T        | 0         |
| 7   | F        | -         |
| 8   | F        | -         |
| 9   | F        | -         |
| 10  | F        | -         |
| 11  | F        | -         |
| 12  | F        | -         |

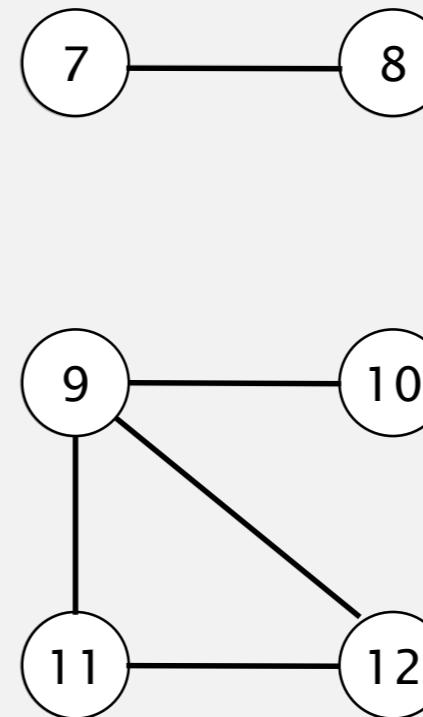
# Depth-first search

To visit a vertex  $v$ :

- Mark vertex  $v$  as visited.
- Recursively visit all unmarked vertices adjacent to  $v$ .



visit 2

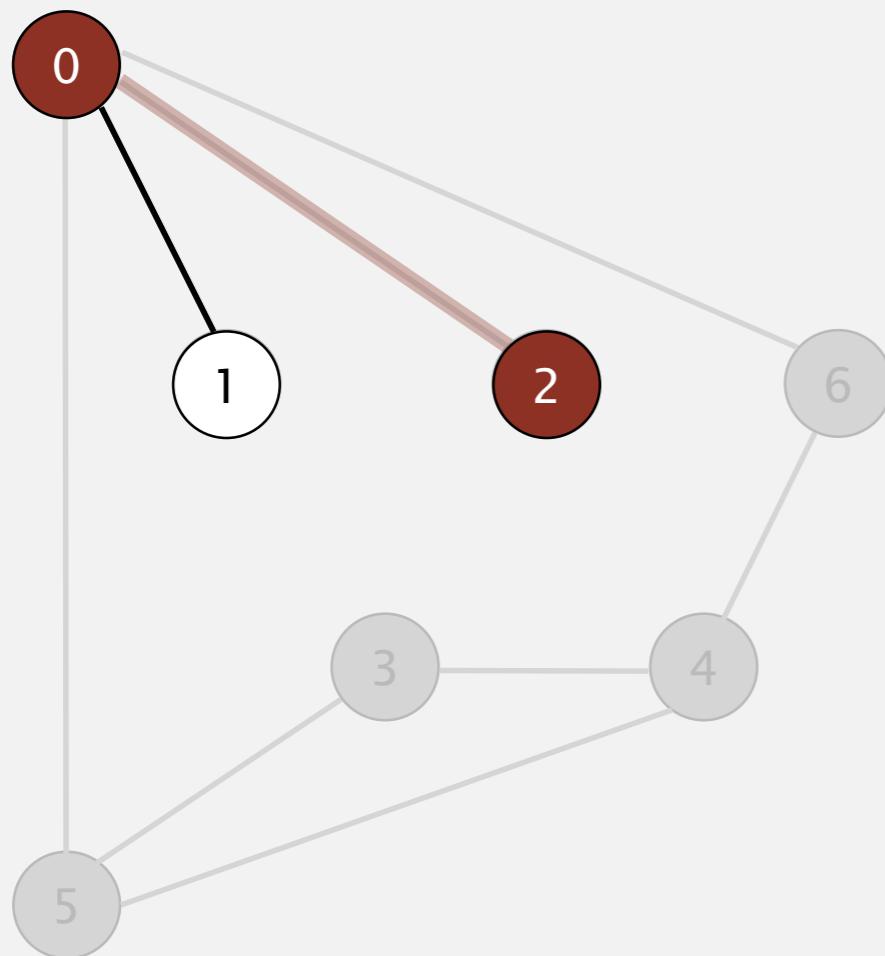


| <code>v</code> | <code>marked[]</code> | <code>edgeTo[v]</code> |
|----------------|-----------------------|------------------------|
| 0              | T                     | -                      |
| 1              | F                     | -                      |
| 2              | T                     | 0                      |
| 3              | T                     | 5                      |
| 4              | T                     | 6                      |
| 5              | T                     | 4                      |
| 6              | T                     | 0                      |
| 7              | F                     | -                      |
| 8              | F                     | -                      |
| 9              | F                     | -                      |
| 10             | F                     | -                      |
| 11             | F                     | -                      |
| 12             | F                     | -                      |

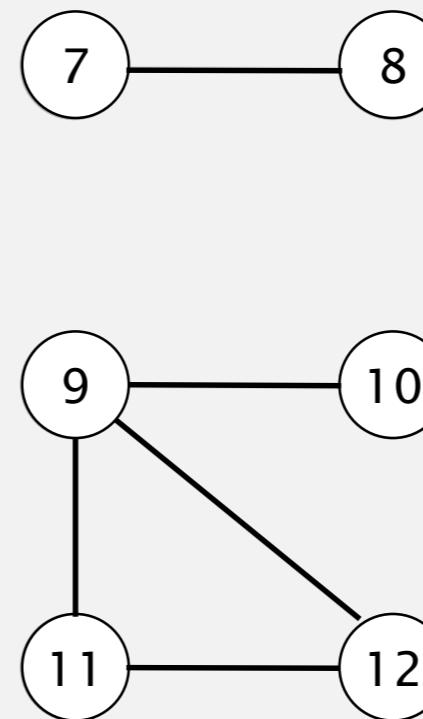
# Depth-first search

To visit a vertex  $v$ :

- Mark vertex  $v$  as visited.
- Recursively visit all unmarked vertices adjacent to  $v$ .



2 done

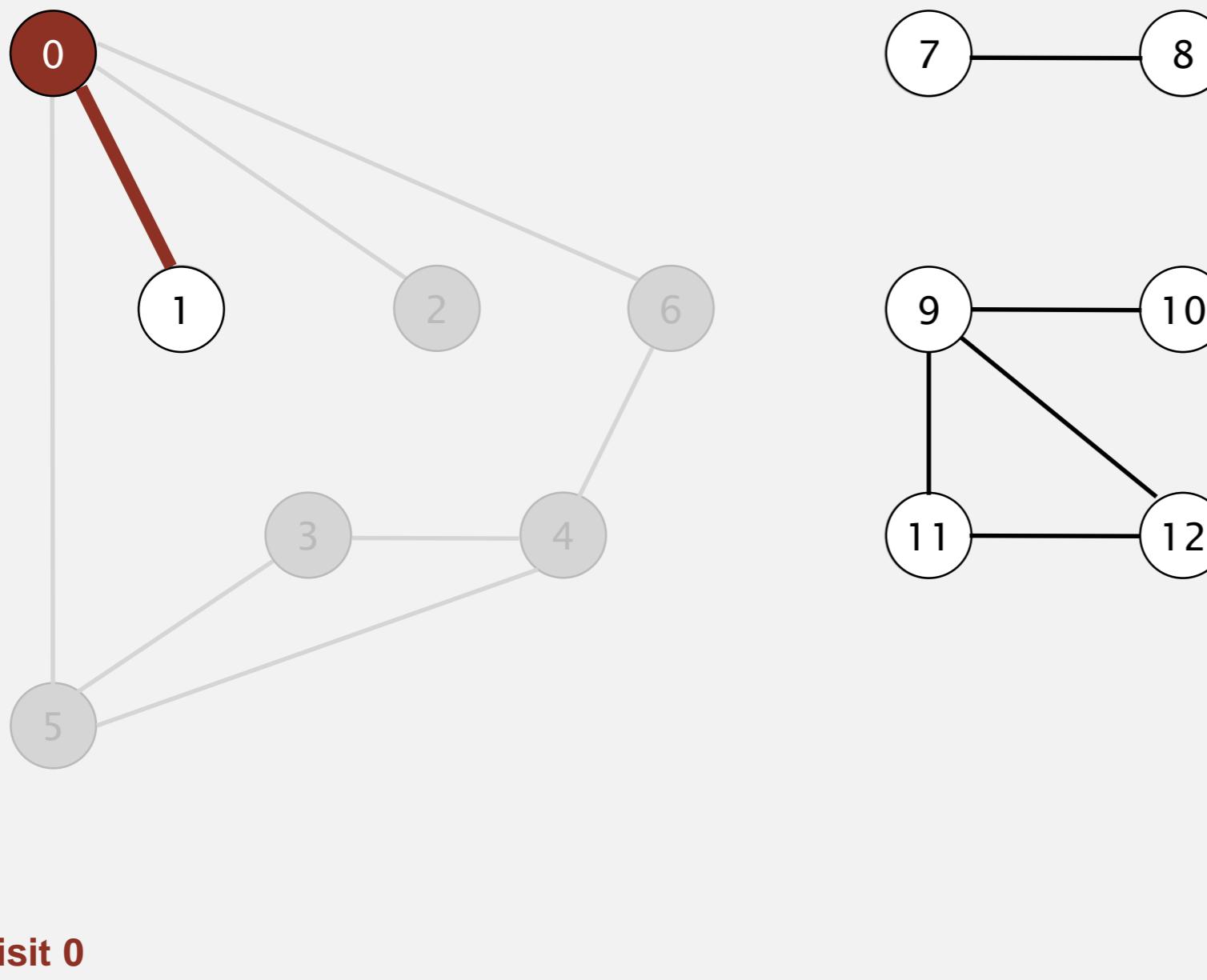


| $v$ | marked[] | edgeTo[ $v$ ] |
|-----|----------|---------------|
| 0   | T        | -             |
| 1   | F        | -             |
| 2   | T        | 0             |
| 3   | T        | 5             |
| 4   | T        | 6             |
| 5   | T        | 4             |
| 6   | T        | 0             |
| 7   | F        | -             |
| 8   | F        | -             |
| 9   | F        | -             |
| 10  | F        | -             |
| 11  | F        | -             |
| 12  | F        | -             |

# Depth-first search

To visit a vertex  $v$ :

- Mark vertex  $v$  as visited.
- Recursively visit all unmarked vertices adjacent to  $v$ .

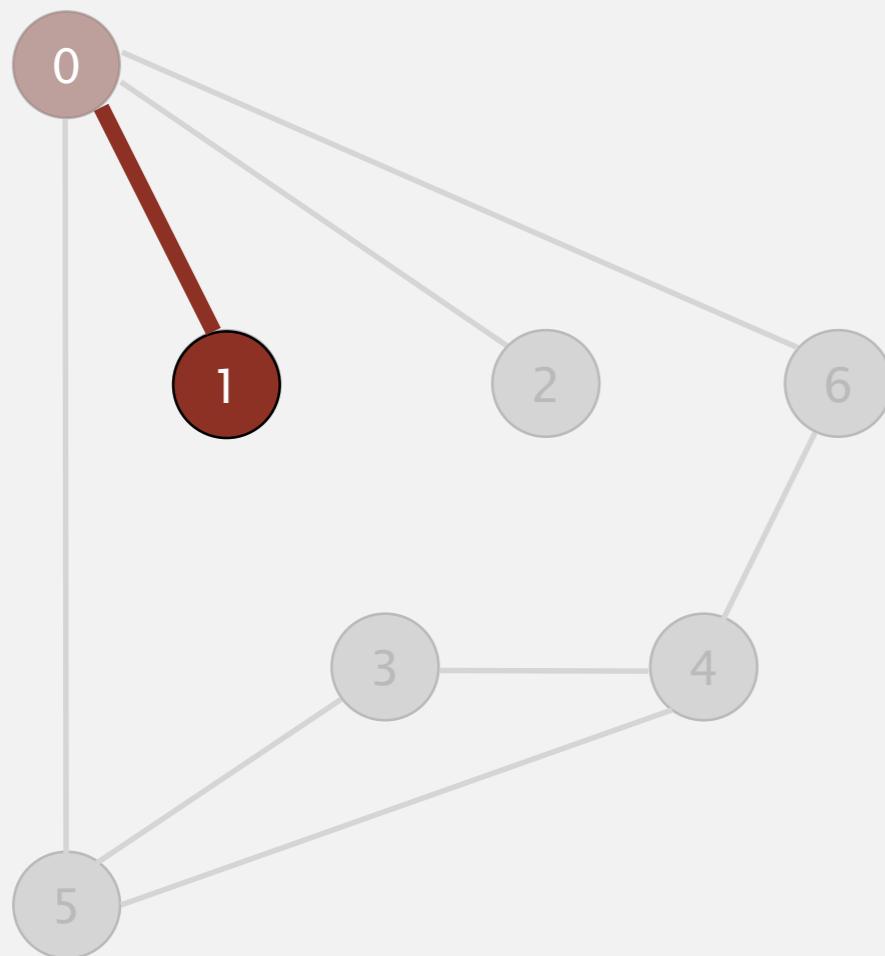


| $v$ | marked[] | edgeTo[v] |
|-----|----------|-----------|
| 0   | T        | -         |
| 1   | F        | -         |
| 2   | T        | 0         |
| 3   | T        | 5         |
| 4   | T        | 6         |
| 5   | T        | 4         |
| 6   | T        | 0         |
| 7   | F        | -         |
| 8   | F        | -         |
| 9   | F        | -         |
| 10  | F        | -         |
| 11  | F        | -         |
| 12  | F        | -         |

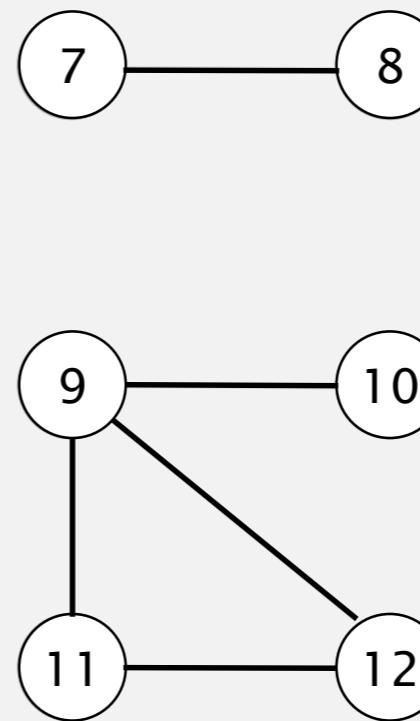
# Depth-first search

To visit a vertex  $v$ :

- Mark vertex  $v$  as visited.
- Recursively visit all unmarked vertices adjacent to  $v$ .



visit 1

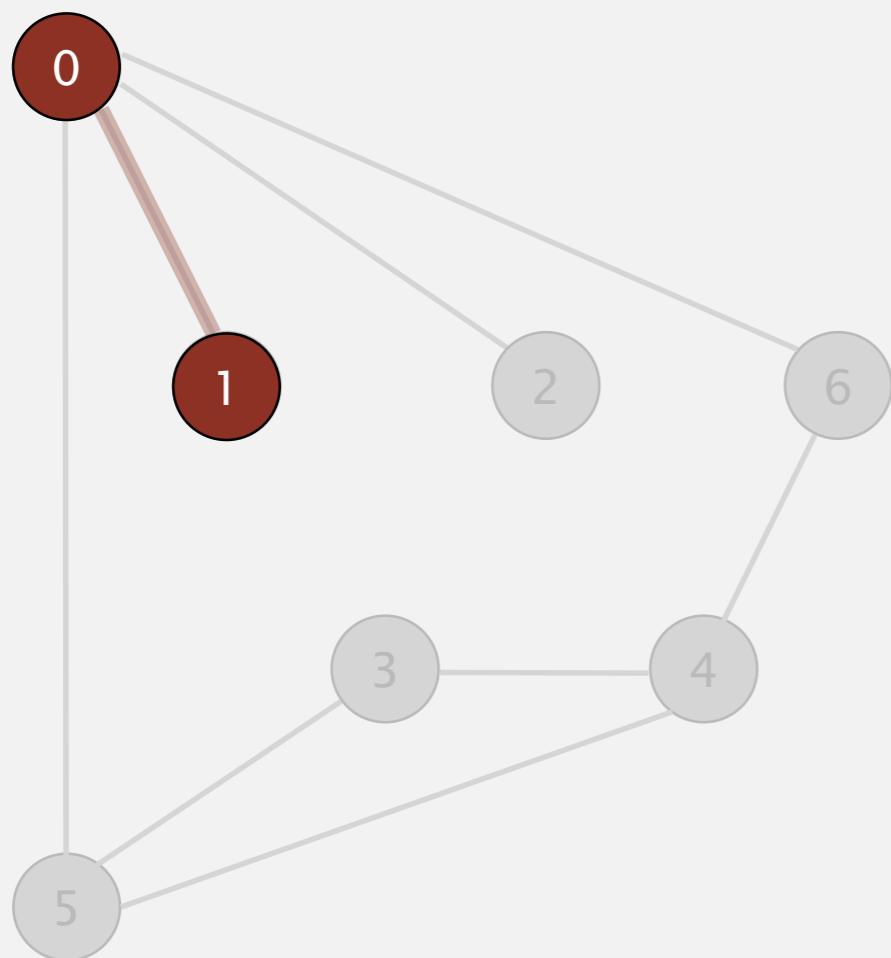


| $v$ | marked[] | edgeTo[v] |
|-----|----------|-----------|
| 0   | T        | -         |
| 1   | T        | 0         |
| 2   | T        | 0         |
| 3   | T        | 5         |
| 4   | T        | 6         |
| 5   | T        | 4         |
| 6   | T        | 0         |
| 7   | F        | -         |
| 8   | F        | -         |
| 9   | F        | -         |
| 10  | F        | -         |
| 11  | F        | -         |
| 12  | F        | -         |

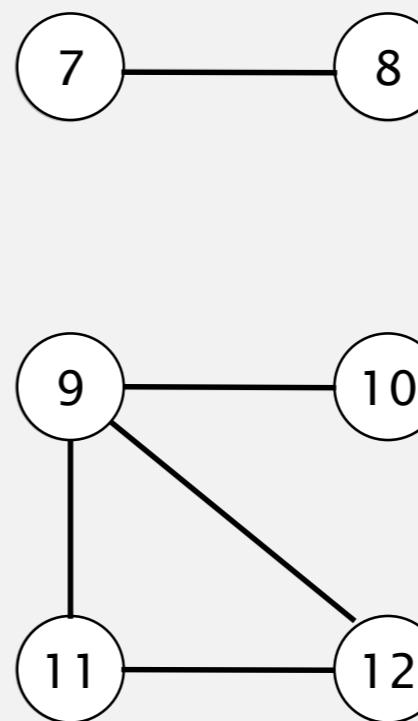
# Depth-first search

To visit a vertex  $v$ :

- Mark vertex  $v$  as visited.
- Recursively visit all unmarked vertices adjacent to  $v$ .



1 done

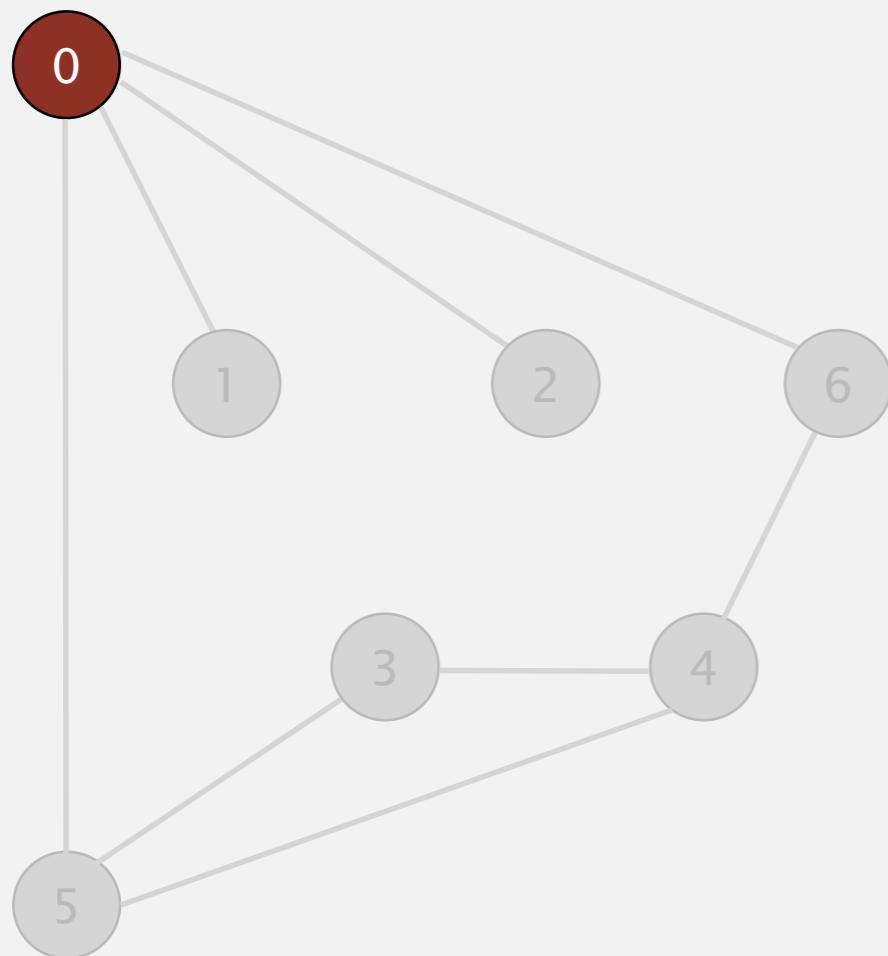


| $v$ | marked[] | edgeTo[v] |
|-----|----------|-----------|
| 0   | T        | -         |
| 1   | T        | 0         |
| 2   | T        | 0         |
| 3   | T        | 5         |
| 4   | T        | 6         |
| 5   | T        | 4         |
| 6   | T        | 0         |
| 7   | F        | -         |
| 8   | F        | -         |
| 9   | F        | -         |
| 10  | F        | -         |
| 11  | F        | -         |
| 12  | F        | -         |

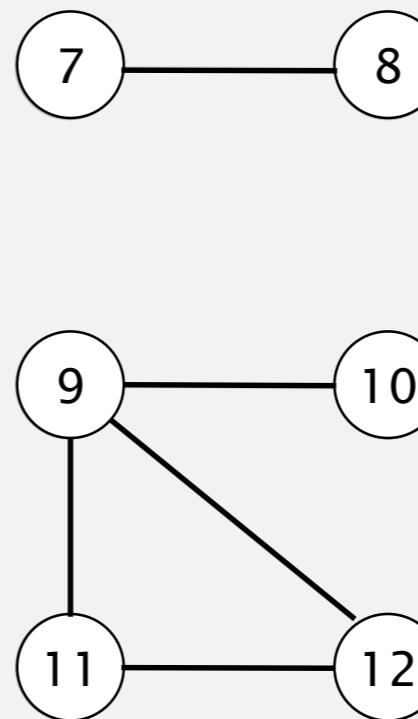
# Depth-first search

To visit a vertex  $v$ :

- Mark vertex  $v$  as visited.
- Recursively visit all unmarked vertices adjacent to  $v$ .



0 done

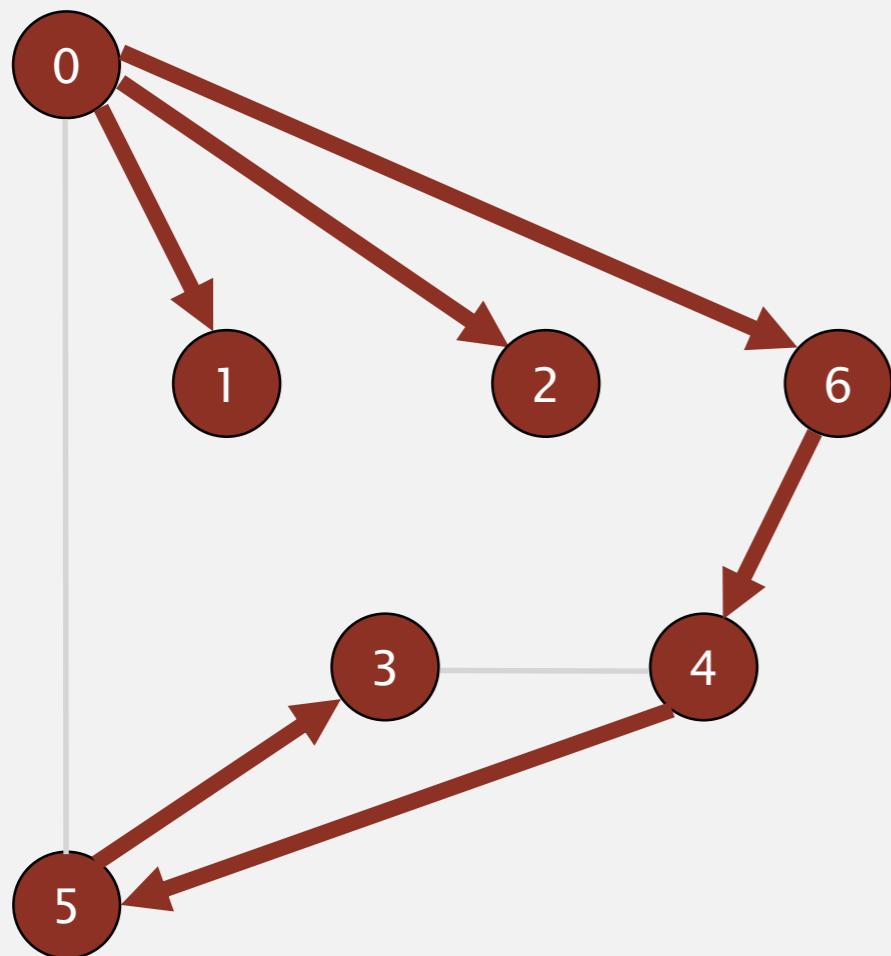


| $v$ | marked[] | edgeTo[ $v$ ] |
|-----|----------|---------------|
| 0   | T        | -             |
| 1   | T        | 0             |
| 2   | T        | 0             |
| 3   | T        | 5             |
| 4   | T        | 6             |
| 5   | T        | 4             |
| 6   | T        | 0             |
| 7   | F        | -             |
| 8   | F        | -             |
| 9   | F        | -             |
| 10  | F        | -             |
| 11  | F        | -             |
| 12  | F        | -             |

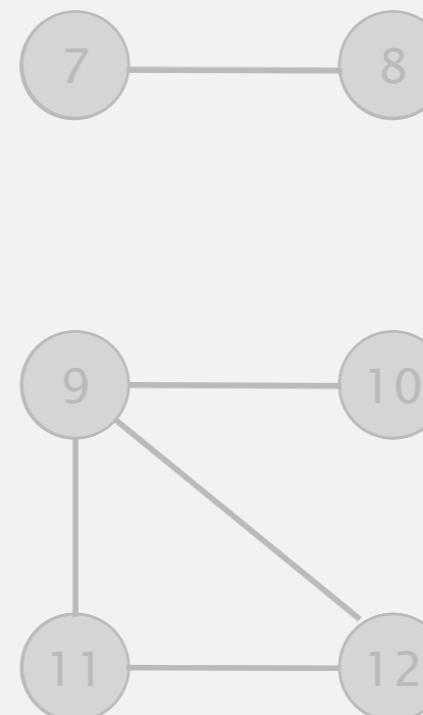
# Depth-first search

To visit a vertex  $v$ :

- Mark vertex  $v$  as visited.
- Recursively visit all unmarked vertices adjacent to  $v$ .



vertices reachable from 0



| $v$ | marked[] | edgeTo[v] |
|-----|----------|-----------|
| 0   | T        | -         |
| 1   | T        | 0         |
| 2   | T        | 0         |
| 3   | T        | 5         |
| 4   | T        | 6         |
| 5   | T        | 4         |
| 6   | T        | 0         |
| 7   | F        | -         |
| 8   | F        | -         |
| 9   | F        | -         |
| 10  | F        | -         |
| 11  | F        | -         |
| 12  | F        | -         |

# Depth-first search

**Goal.** Find all vertices connected to  $s$  (and a path).

## Algorithm.

- Use recursion (ball of string).
- Mark each visited vertex (and keep track of edge taken to visit it).
- Return (retrace steps) when no unvisited options.

## Data structures.

- `boolean[] marked` to mark visited vertices.
- `int[] edgeTo` to keep tree of paths.  
 $(\text{edgeTo}[w] == v)$  means that edge  $v-w$  taken to visit  $w$  for first time

# Depth-first search

```
void DFS(int i)
{
    node *p = adj[i];
    visited[i]=1;
    while(p != NULL)
    {
        i = p->vertex;
        if(!visited[i])
            DFS(i);
        p = p->next;
    }
}
```

```
typedef struct node
{
    struct node *next;
    int vertex;
}node;

//GLOBAL PARAMETERS
node * adj [13];
int visited[13];
```

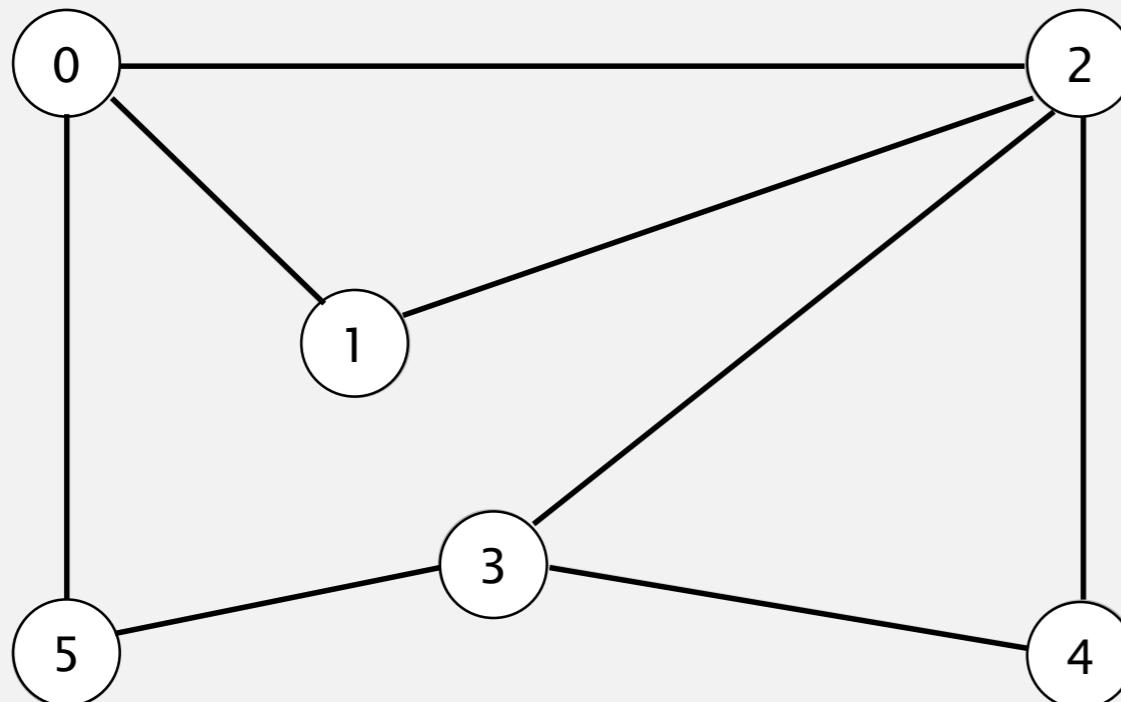
# UNDIRECTED GRAPHS

- ▶ **Graph API**
- ▶ **Depth-first search**
- ▶ **Breadth-first search**

# Breadth-first search

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices adjacent to  $v$  and mark them.



**tinyCG.txt**

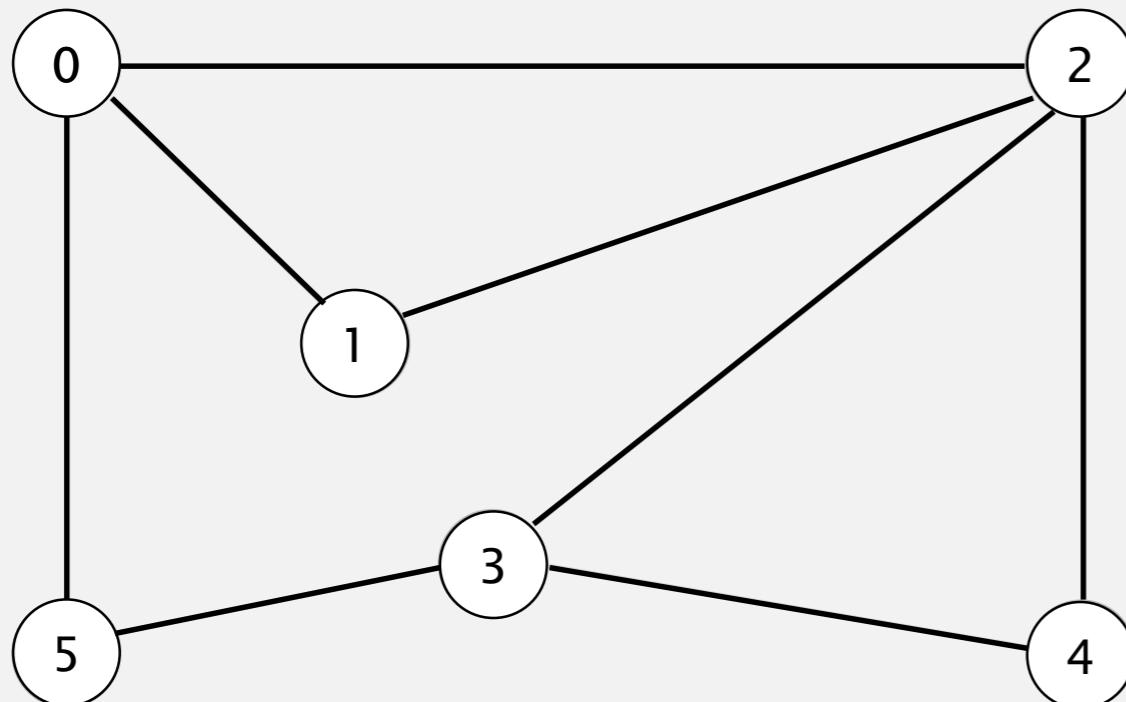
|   |   |   |
|---|---|---|
| V | → | 6 |
| E | → | 8 |
| 0 | 5 |   |
| 2 | 4 |   |
| 2 | 3 |   |
| 1 | 2 |   |
| 0 | 1 |   |
| 3 | 4 |   |
| 3 | 5 |   |
| 0 | 2 |   |

**graph G**

# Breadth-first search

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices adjacent to  $v$  and mark them.



queue

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

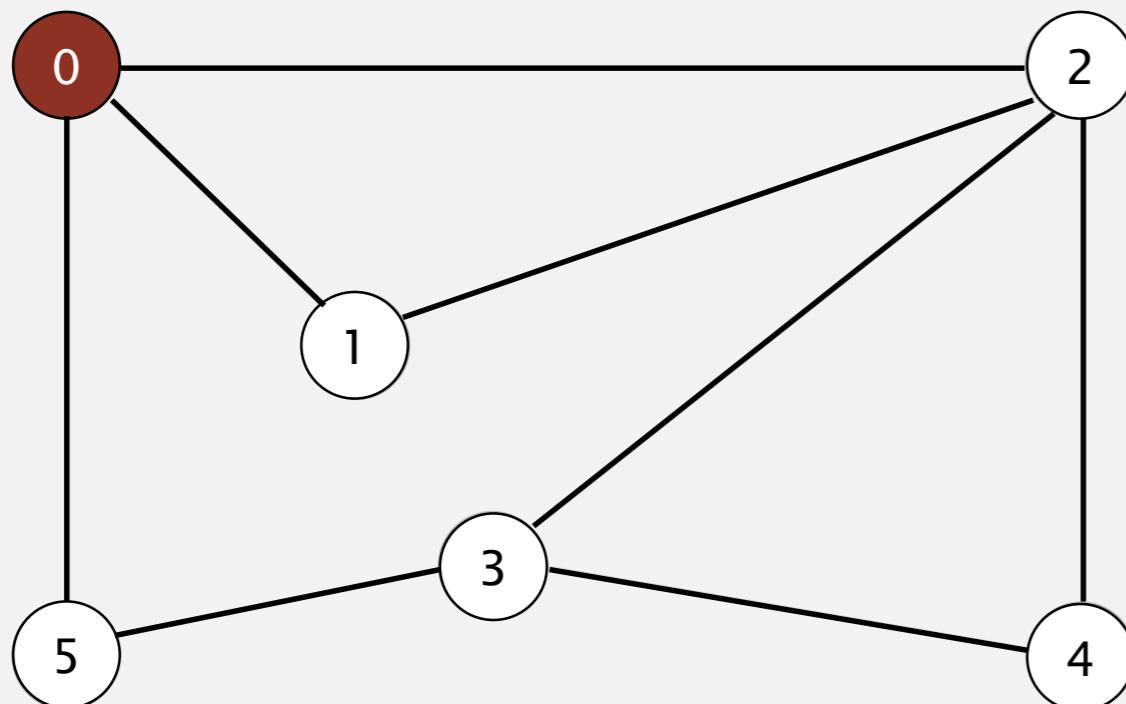
| v | edgeTo[v] |
|---|-----------|
| 0 | -         |
| 1 | -         |
| 2 | -         |
| 3 | -         |
| 4 | -         |
| 5 | -         |

**add 0 to queue**

# Breadth-first search

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices adjacent to  $v$  and mark them.



queue

| v |
|---|
| 0 |
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |

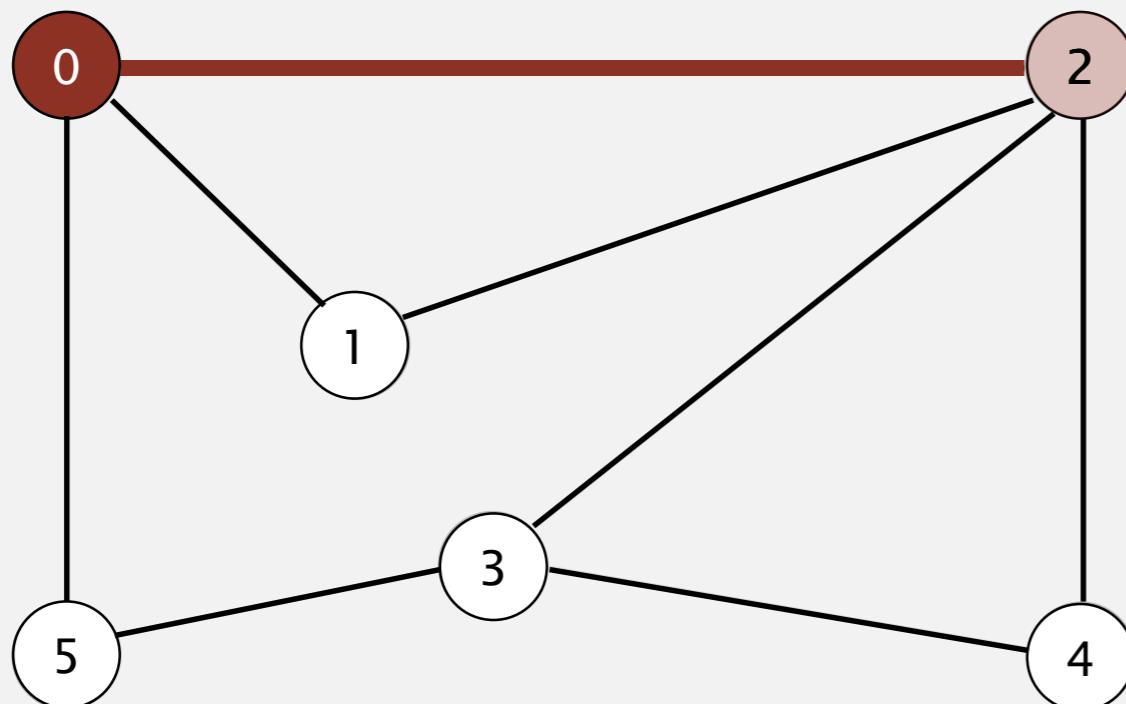
| v | edgeTo[v] |
|---|-----------|
| 0 | -         |
| 1 | -         |
| 2 | -         |
| 3 | -         |
| 4 | -         |
| 5 | -         |

dequeue 0

# Breadth-first search

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices adjacent to  $v$  and mark them.



queue

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

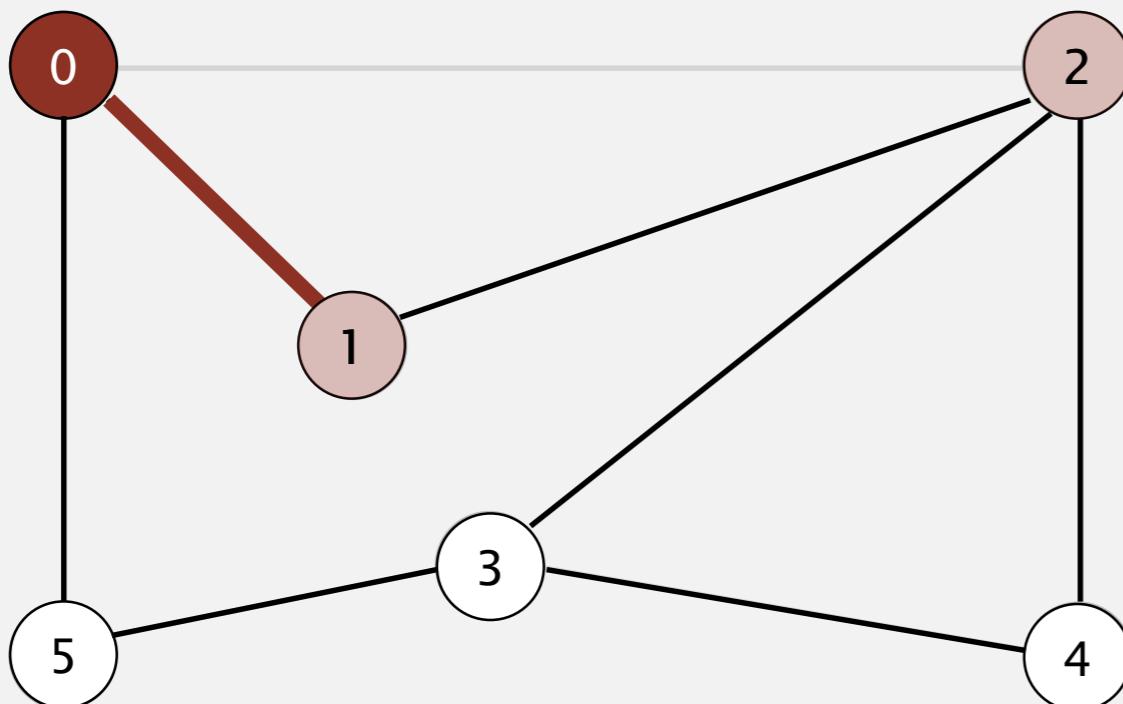
| v | edgeTo[v] |
|---|-----------|
| 0 | -         |
| 1 | -         |
| 2 | 0         |
| 3 | -         |
| 4 | -         |
| 5 | -         |

dequeue 0

# Breadth-first search

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices adjacent to  $v$  and mark them.



queue

| v |
|---|
| 0 |
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |

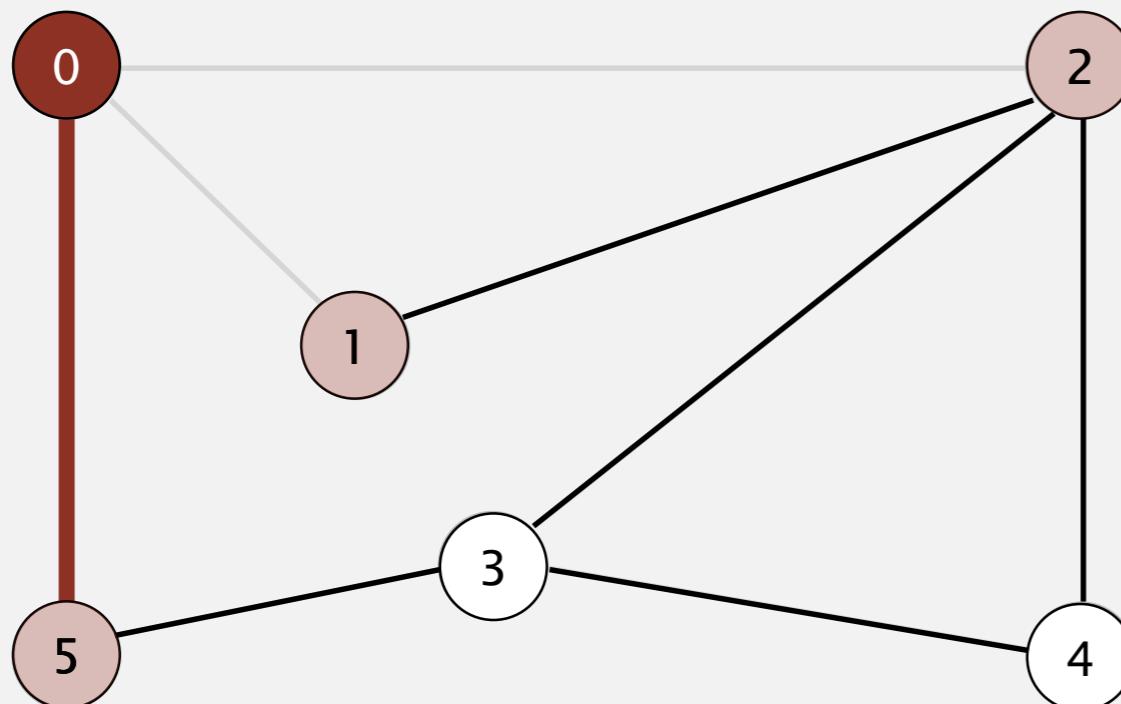
| v | edgeTo[v] |
|---|-----------|
| 0 | -         |
| 1 | 0         |
| 2 | 0         |
| 3 | -         |
| 4 | -         |
| 5 | -         |

dequeue 0

# Breadth-first search

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices adjacent to  $v$  and mark them.



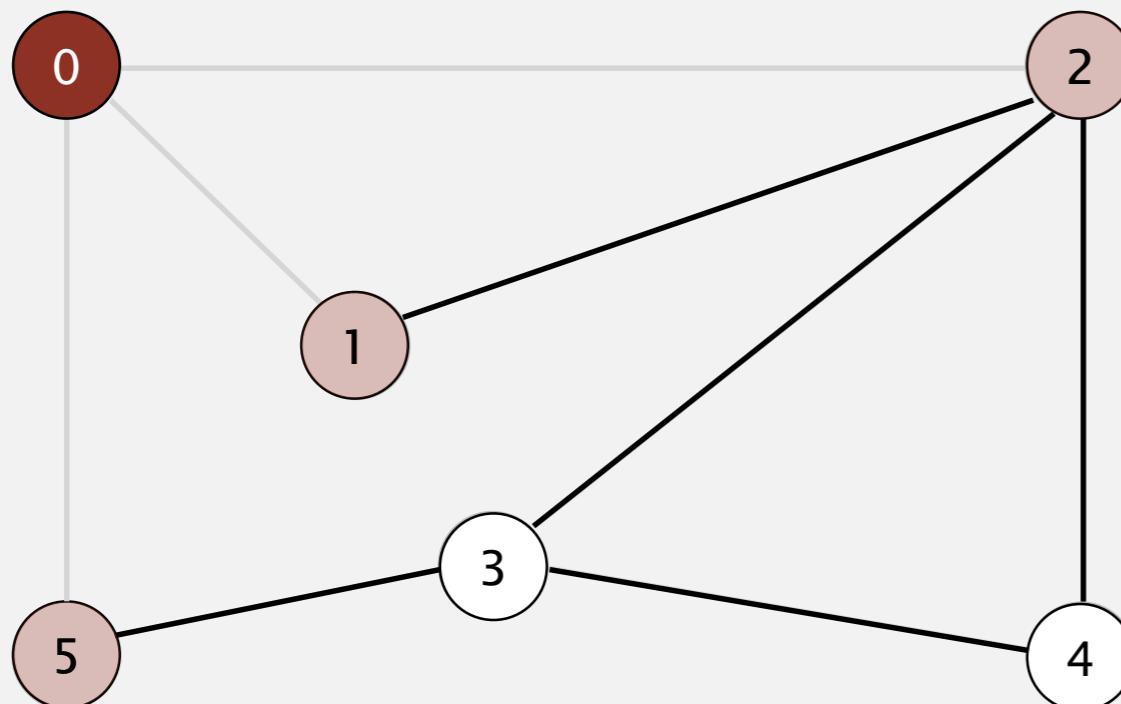
**dequeue 0**

| queue | v | edgeTo[v] |
|-------|---|-----------|
|       | 0 | -         |
|       | 1 | 0         |
|       | 2 | 0         |
|       | 3 | -         |
|       | 4 | -         |
| 1     | 5 | 0         |
| 2     |   |           |

# Breadth-first search

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices adjacent to  $v$  and mark them.



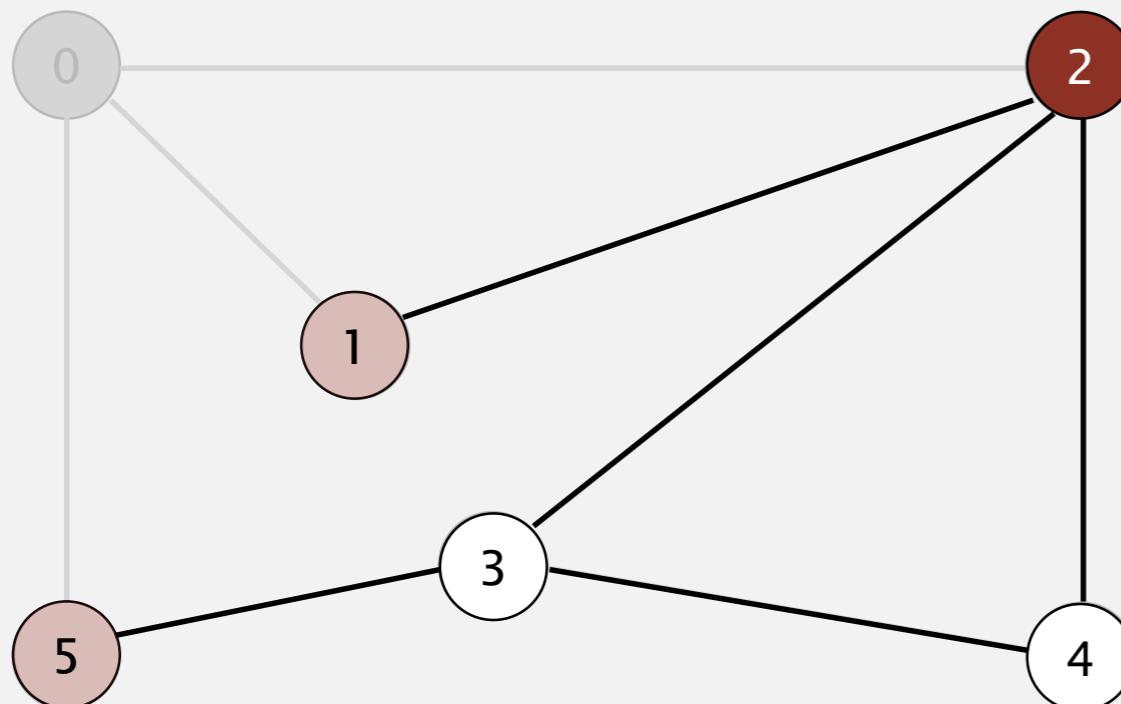
| queue | v | edgeTo[v] |
|-------|---|-----------|
|       | 0 | -         |
|       | 1 | 0         |
|       | 2 | 0         |
| 5     | 3 | -         |
| 1     | 4 | -         |
| 2     | 5 | 0         |

0 done

# Breadth-first search

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices adjacent to  $v$  and mark them.



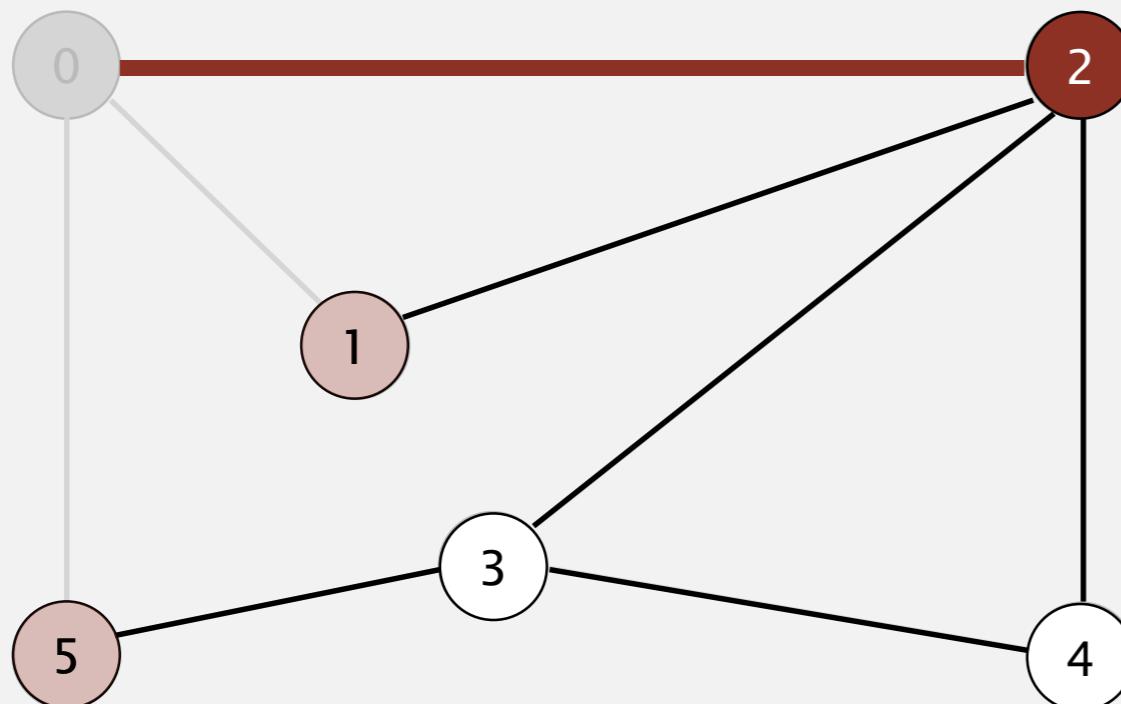
| queue | v | edgeTo[v] |
|-------|---|-----------|
|       | 0 | -         |
|       | 1 | 0         |
|       | 2 | 0         |
| 5     | 3 | -         |
| 1     | 4 | -         |
| 5     | 0 | 5         |
| 2     | - | -         |

**dequeue 2**

# Breadth-first search

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices adjacent to  $v$  and mark them.



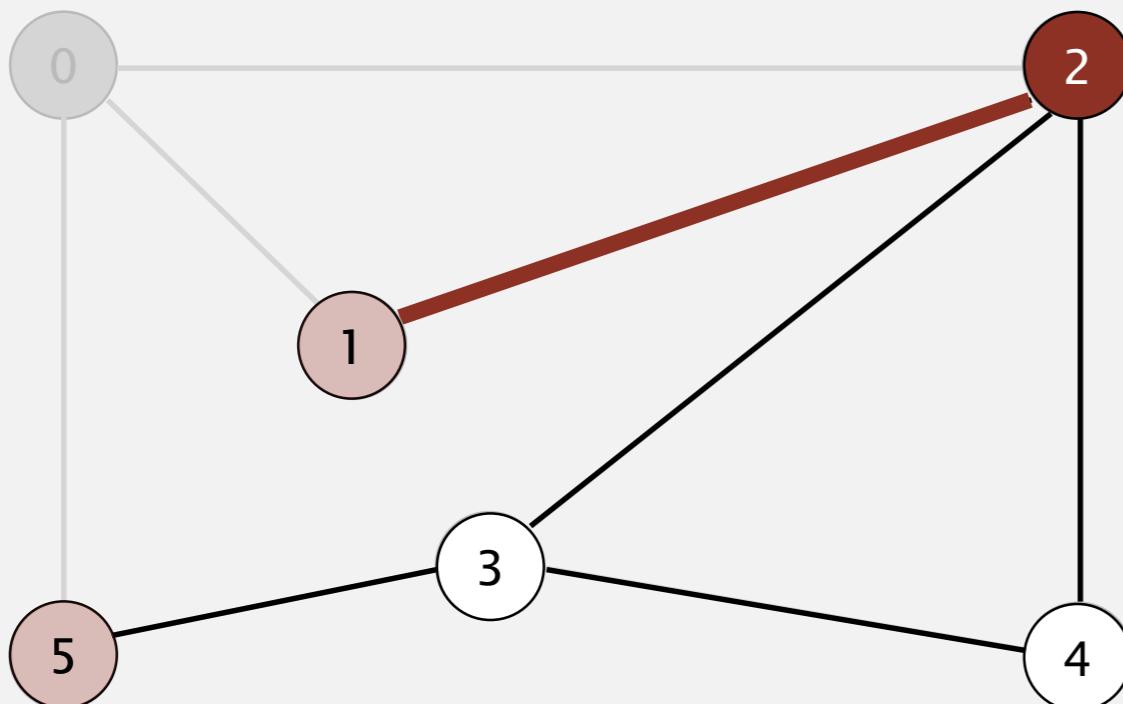
| queue | v | edgeTo[v] |
|-------|---|-----------|
|       | 0 | -         |
|       | 1 | 0         |
|       | 2 | 0         |
|       | 3 | -         |
|       | 4 | -         |
| 5     | 5 | 0         |
| 1     | 1 |           |

**dequeue 2**

# Breadth-first search

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices adjacent to  $v$  and mark them.



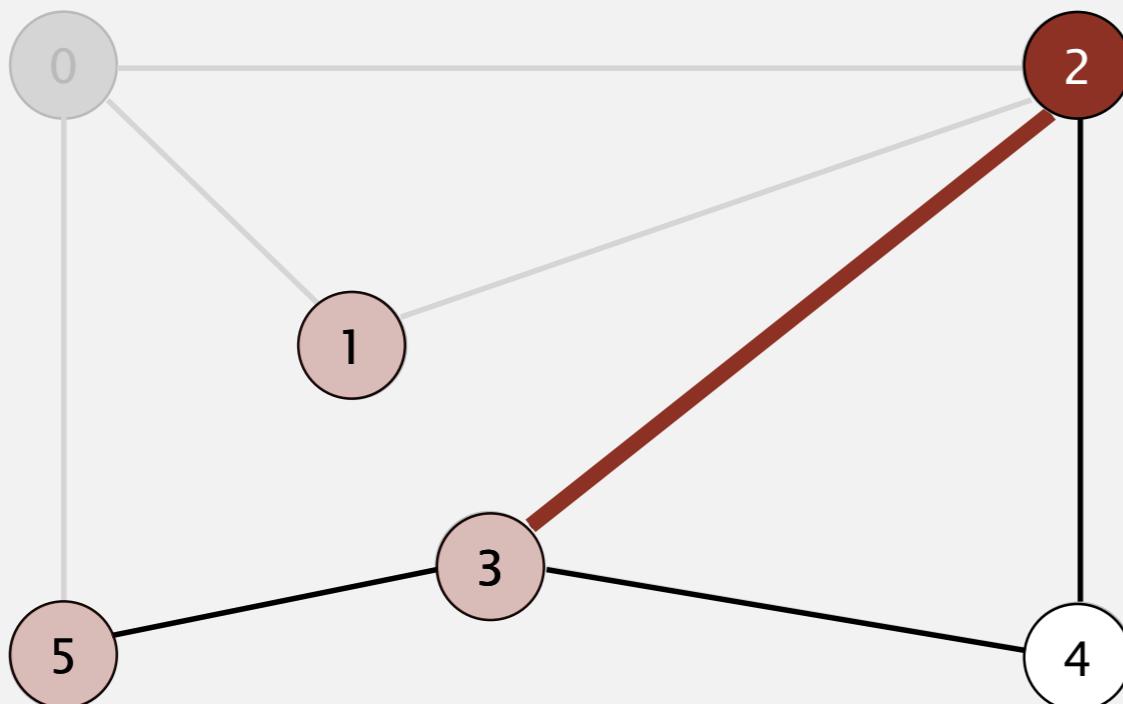
| queue | v | edgeTo[v] |
|-------|---|-----------|
|       | 0 | -         |
|       | 1 | 0         |
|       | 2 | 0         |
|       | 3 | -         |
|       | 4 | -         |
| 5     | 5 | 0         |
| 1     | 1 |           |

**dequeue 2**

# Breadth-first search

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices adjacent to  $v$  and mark them.



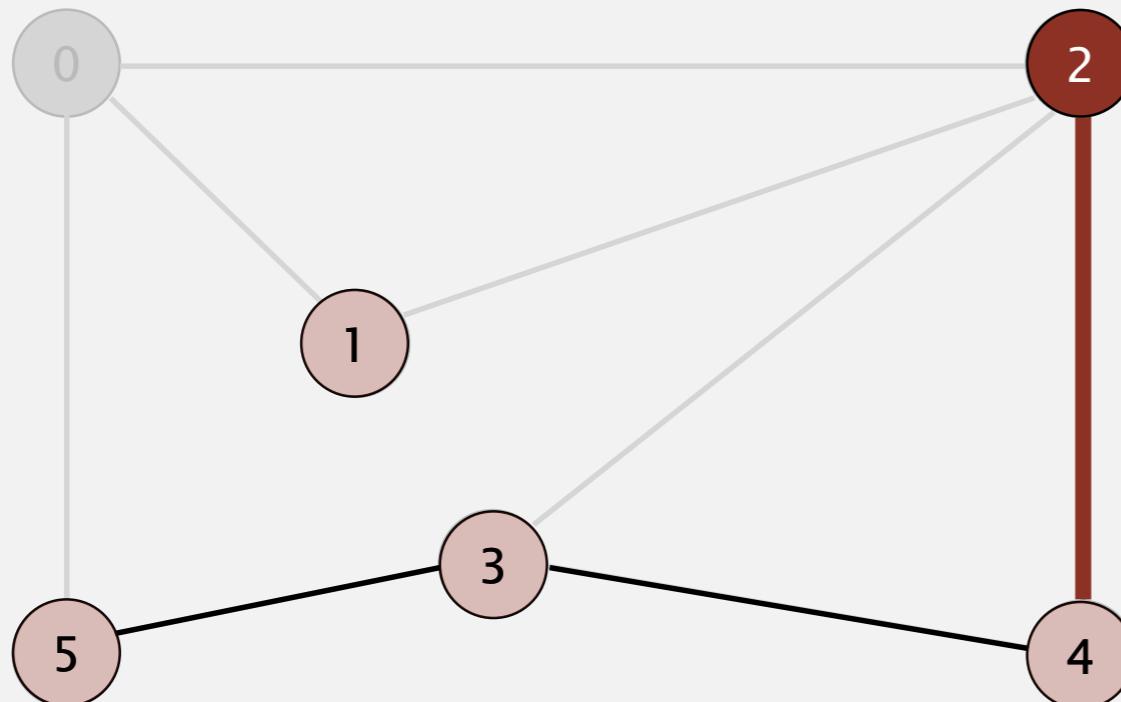
| queue | v | edgeTo[v] |
|-------|---|-----------|
|       | 0 | -         |
|       | 1 | 0         |
|       | 2 | 0         |
|       | 3 | 2         |
|       | 4 | -         |
| 5     | 5 | 0         |
| 1     |   |           |

**dequeue 2**

# Breadth-first search

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices adjacent to  $v$  and mark them.



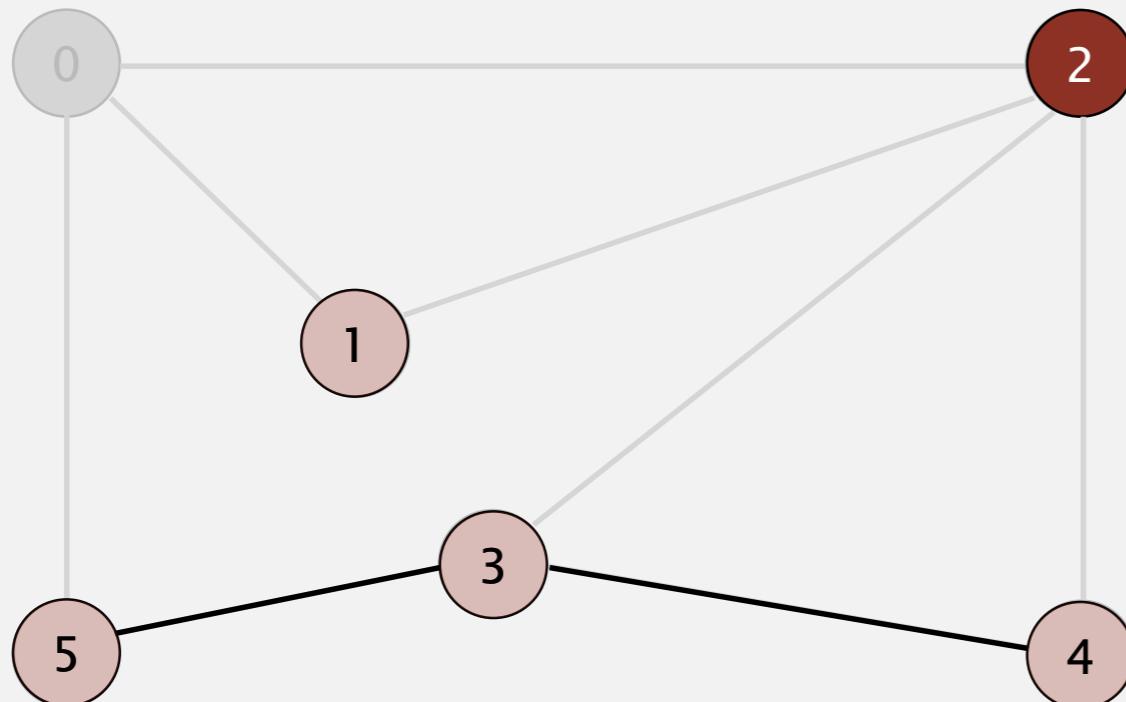
| queue | v | edgeTo[v] |
|-------|---|-----------|
|       | 0 | -         |
|       | 1 | 0         |
|       | 2 | 0         |
| 3     | 3 | 2         |
| 4     | 4 | 2         |
| 5     | 5 | 0         |
| 1     | 1 |           |

**dequeue 2**

# Breadth-first search

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices adjacent to  $v$  and mark them.



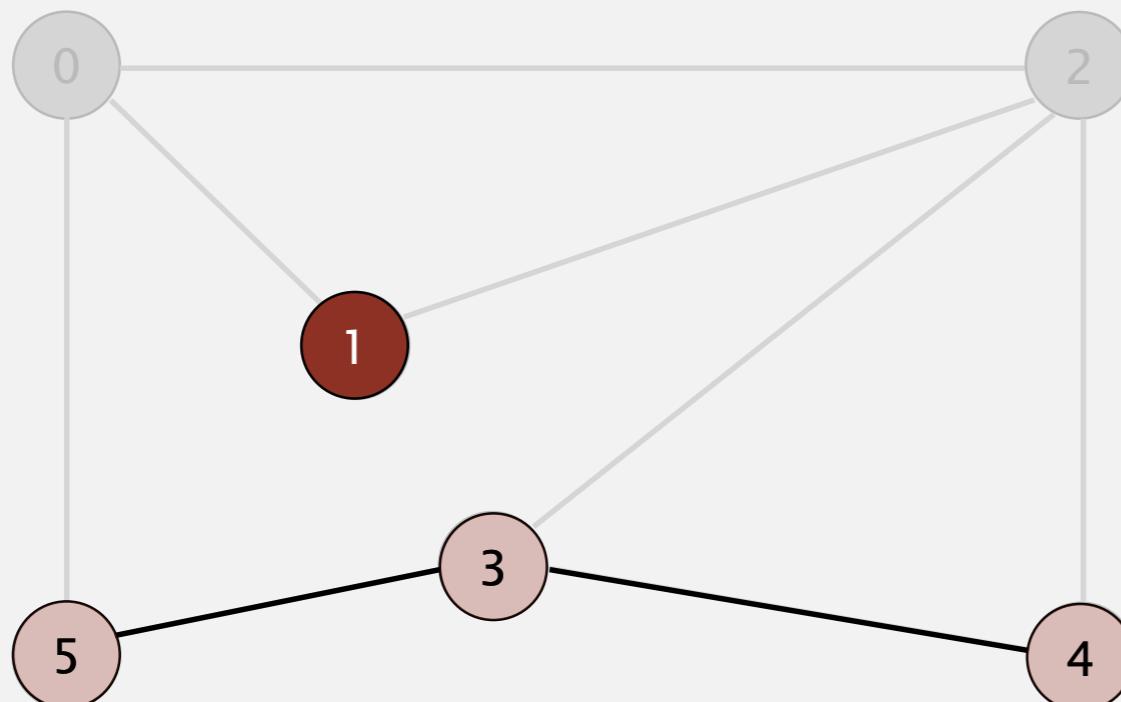
| queue | v | edgeTo[v] |
|-------|---|-----------|
|       | 0 | -         |
|       | 1 | 0         |
| 4     | 2 | 0         |
|       | 3 | 2         |
|       | 4 | 2         |
| 5     | 5 | 0         |
|       | 1 |           |

2 done

# Breadth-first search

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices adjacent to  $v$  and mark them.



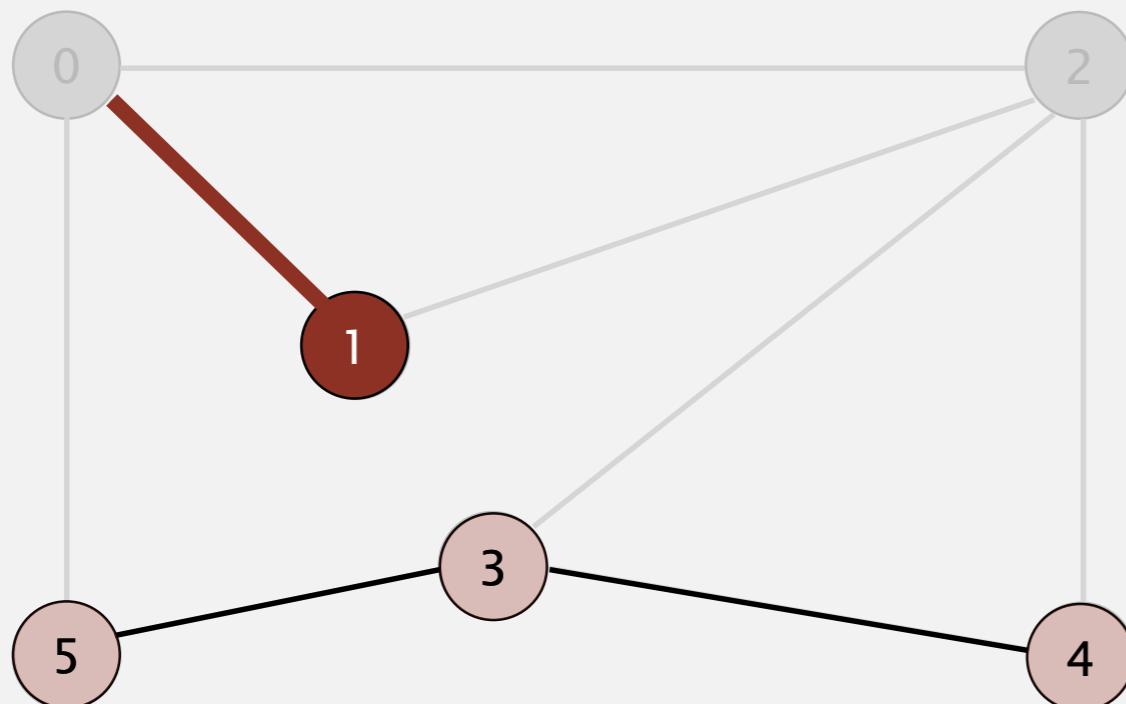
**dequeue 1**

| queue | v | edgeTo[v] |
|-------|---|-----------|
|       | 0 | -         |
|       | 1 | 0         |
|       | 2 | 0         |
|       | 3 | 2         |
|       | 4 | 2         |
|       | 5 | 0         |
|       | 1 |           |

# Breadth-first search

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices adjacent to  $v$  and mark them.



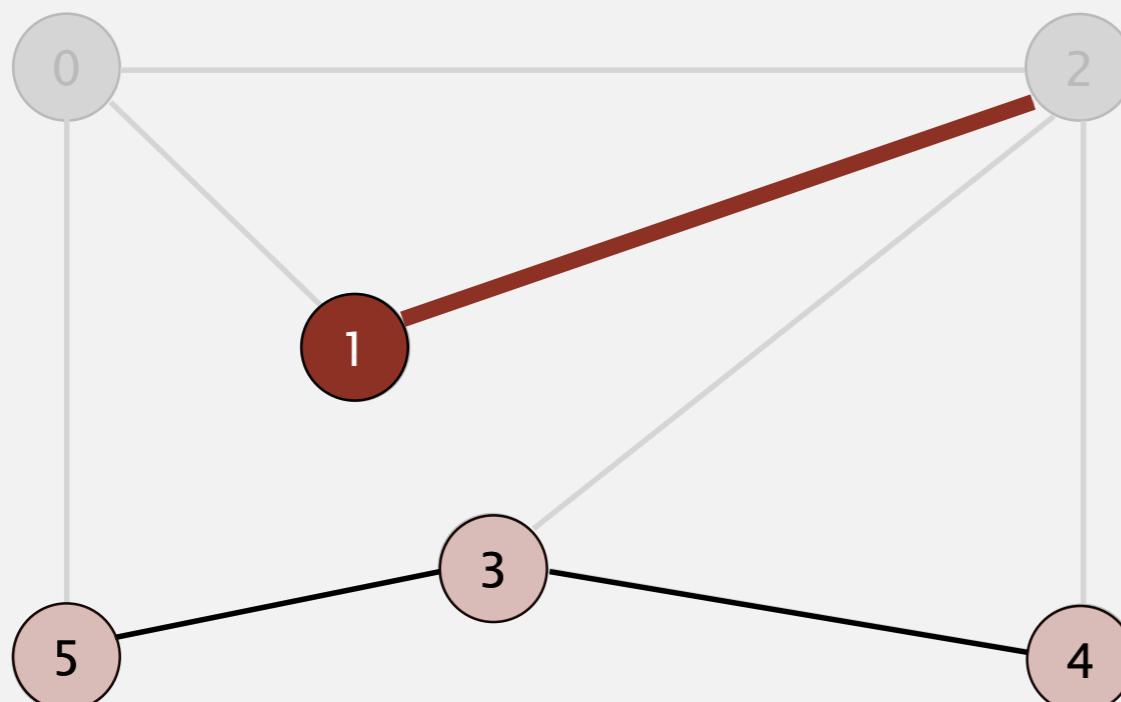
**dequeue 1**

| queue | v | edgeTo[v] |
|-------|---|-----------|
|       | 0 | -         |
|       | 1 | 0         |
|       | 2 | 0         |
| 4     | 3 | 2         |
| 3     | 4 | 2         |
| 5     | 5 | 0         |
| 5     |   |           |

# Breadth-first search

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices adjacent to  $v$  and mark them.



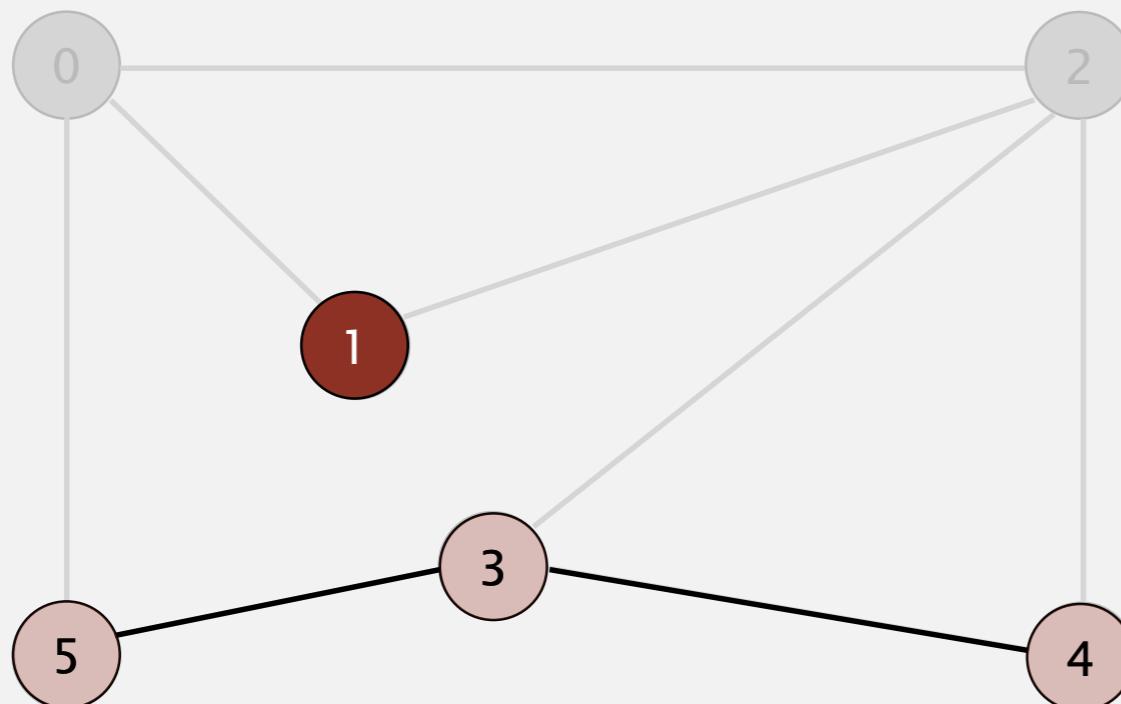
**dequeue 1**

| queue | v | edgeTo[v] |
|-------|---|-----------|
|       | 0 | -         |
|       | 1 | 0         |
|       | 2 | 0         |
| 4     | 3 | 2         |
| 3     | 4 | 2         |
| 5     | 5 | 0         |
| 5     |   |           |

# Breadth-first search

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices adjacent to  $v$  and mark them.



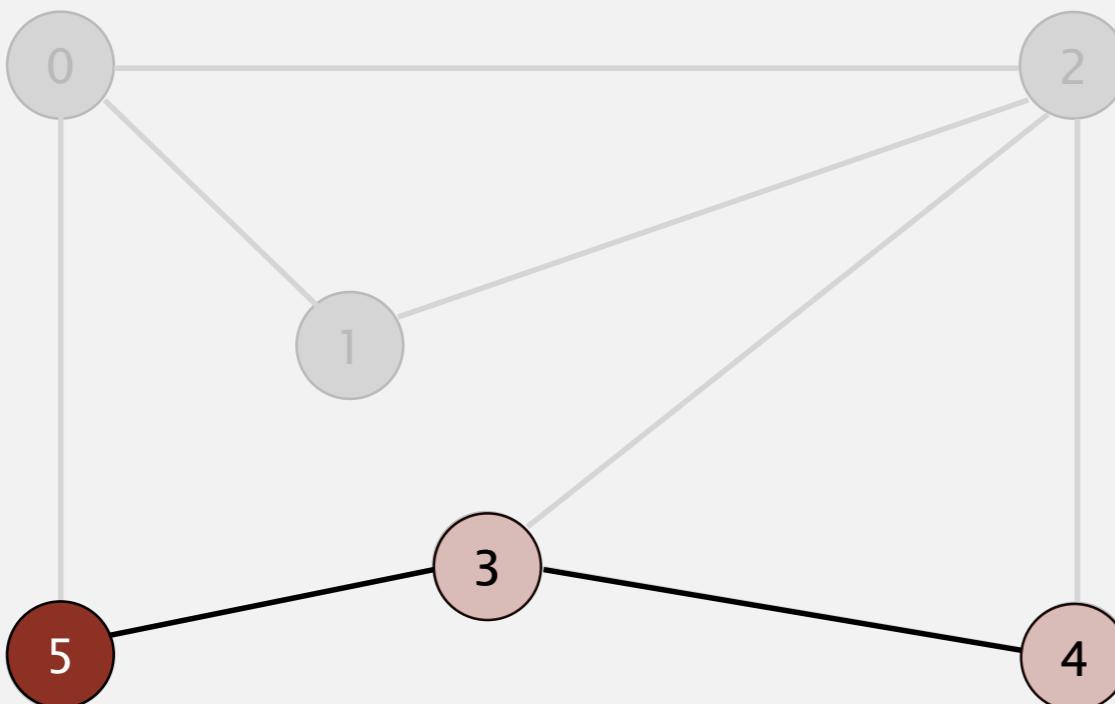
| queue | v | edgeTo[v] |
|-------|---|-----------|
|       | 0 | -         |
|       | 1 | 0         |
|       | 2 | 0         |
| 4     | 3 | 2         |
| 3     | 4 | 2         |
| 5     | 5 | 0         |
| 5     |   |           |

1 done

# Breadth-first search

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices adjacent to  $v$  and mark them.



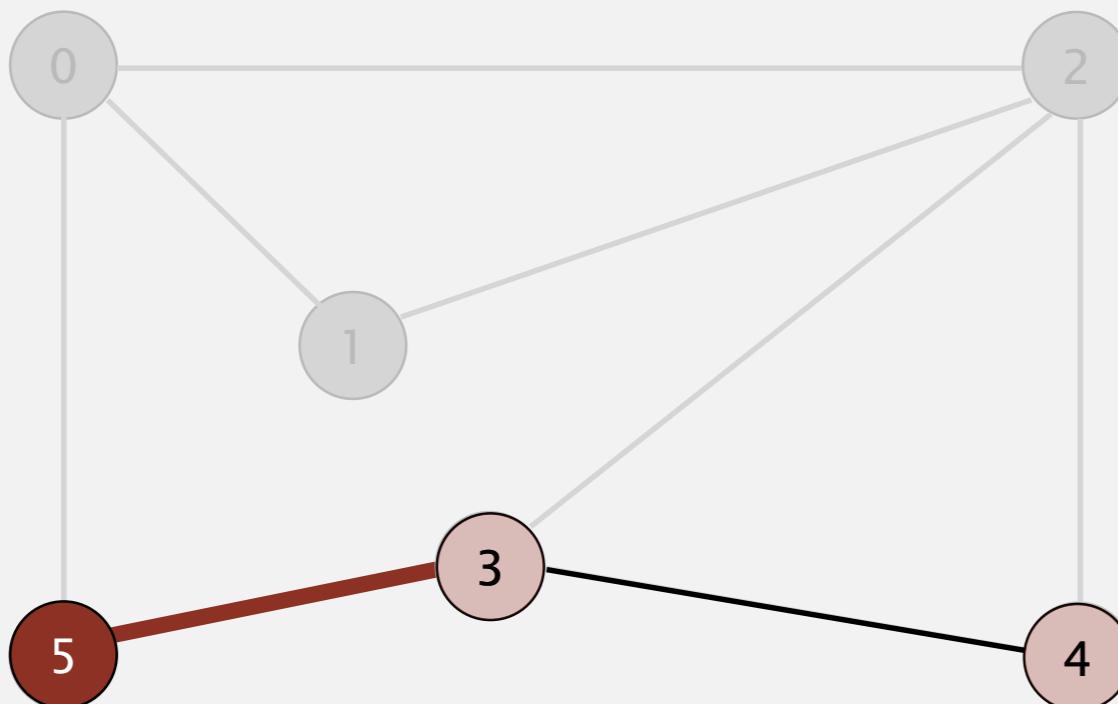
**dequeue 5**

| queue | v | edgeTo[v] |
|-------|---|-----------|
|       | 0 | -         |
|       | 1 | 0         |
|       | 2 | 0         |
| 4     | 3 | 2         |
| 3     | 4 | 2         |
| 5     | 5 | 0         |

# Breadth-first search

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices adjacent to  $v$  and mark them.



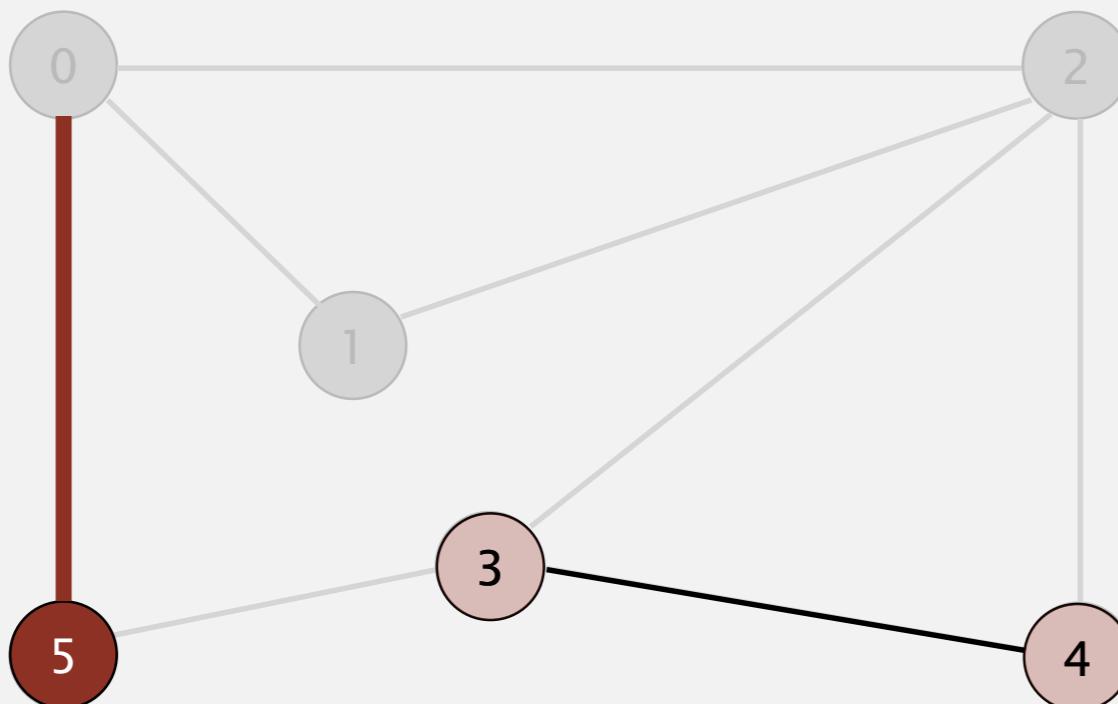
| queue | v | edgeTo[v] |
|-------|---|-----------|
|       | 0 | -         |
|       | 1 | 0         |
|       | 2 | 0         |
|       | 3 | 2         |
|       | 4 | 2         |
| 4     | 5 | 0         |
| 3     |   |           |

**dequeue 5**

# Breadth-first search

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices adjacent to  $v$  and mark them.



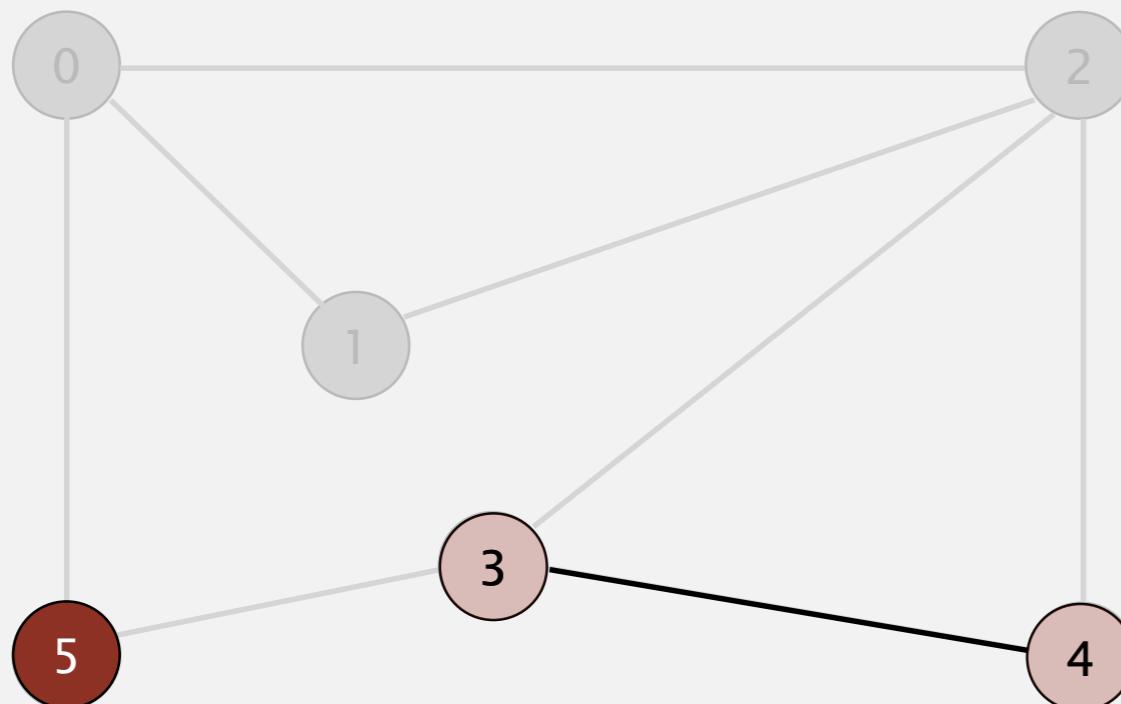
| queue | v | edgeTo[v] |
|-------|---|-----------|
|       | 0 | -         |
|       | 1 | 0         |
|       | 2 | 0         |
|       | 3 | 2         |
|       | 4 | 2         |
| 4     | 5 | 0         |
| 3     |   |           |

**dequeue 5**

# Breadth-first search

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices adjacent to  $v$  and mark them.



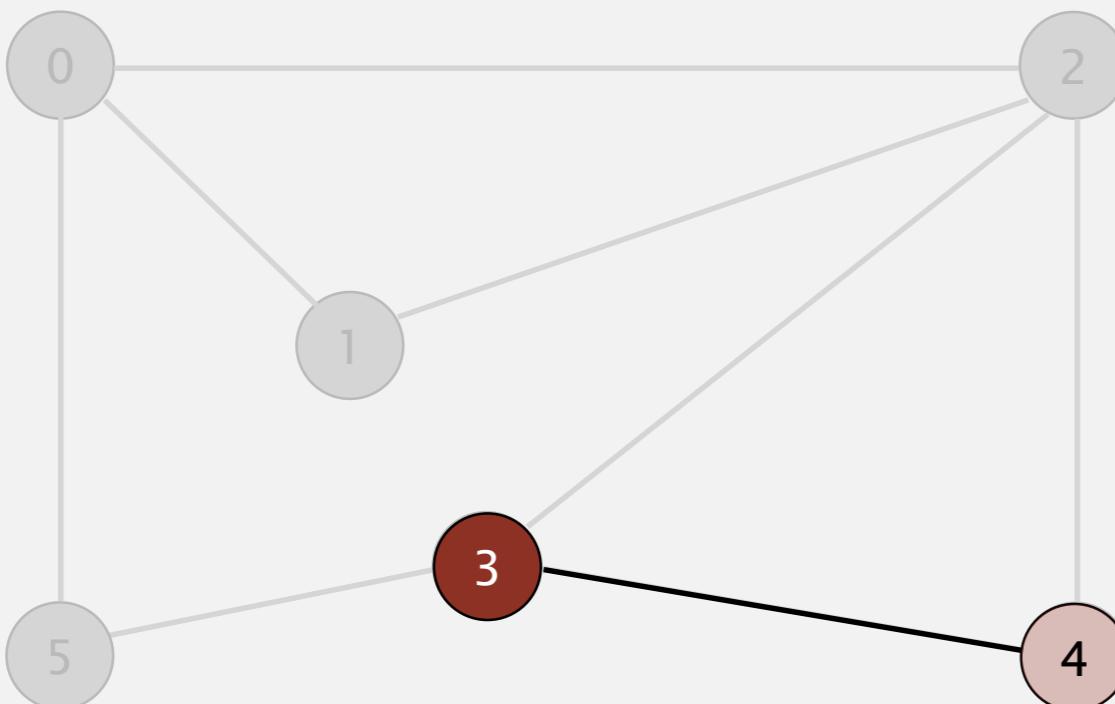
| queue | v | edgeTo[v] |
|-------|---|-----------|
|       | 0 | -         |
|       | 1 | 0         |
|       | 2 | 0         |
|       | 3 | 2         |
| 4     | 4 | 2         |
| 4     | 5 | 0         |
| 3     |   |           |

5 done

# Breadth-first search

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices adjacent to  $v$  and mark them.



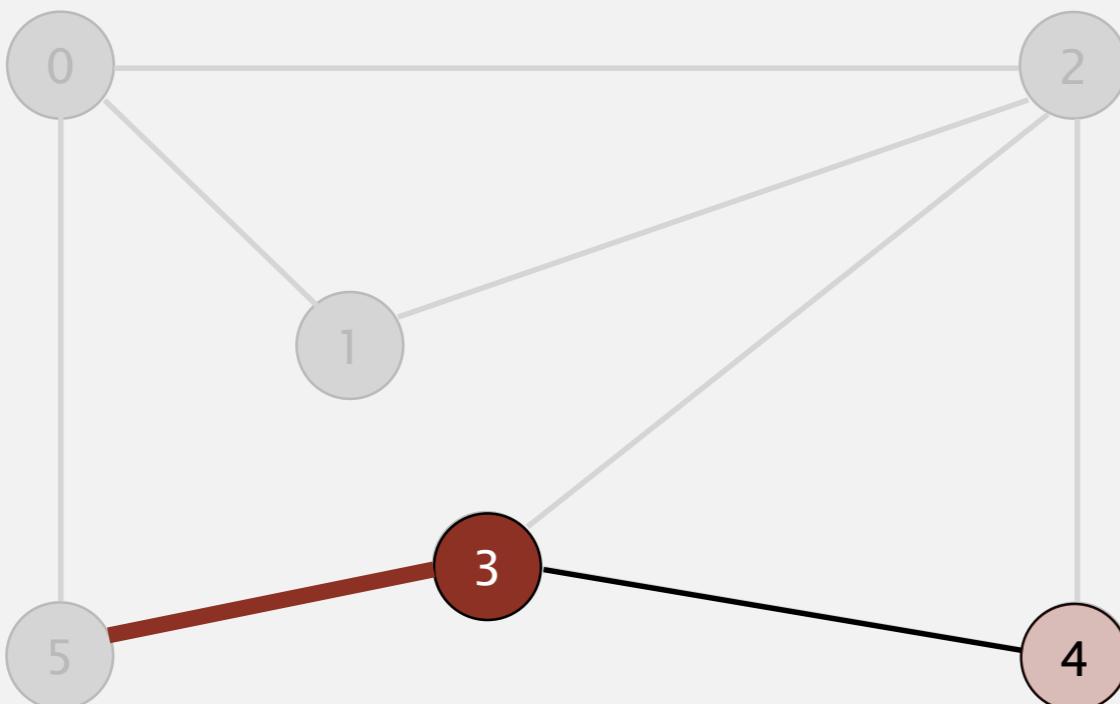
| queue | v | edgeTo[v] |
|-------|---|-----------|
|       | 0 | -         |
|       | 1 | 0         |
|       | 2 | 0         |
|       | 3 | 2         |
| 4     | 4 | 2         |
| 3     | 5 | 0         |

dequeue 3

# Breadth-first search

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices adjacent to  $v$  and mark them.



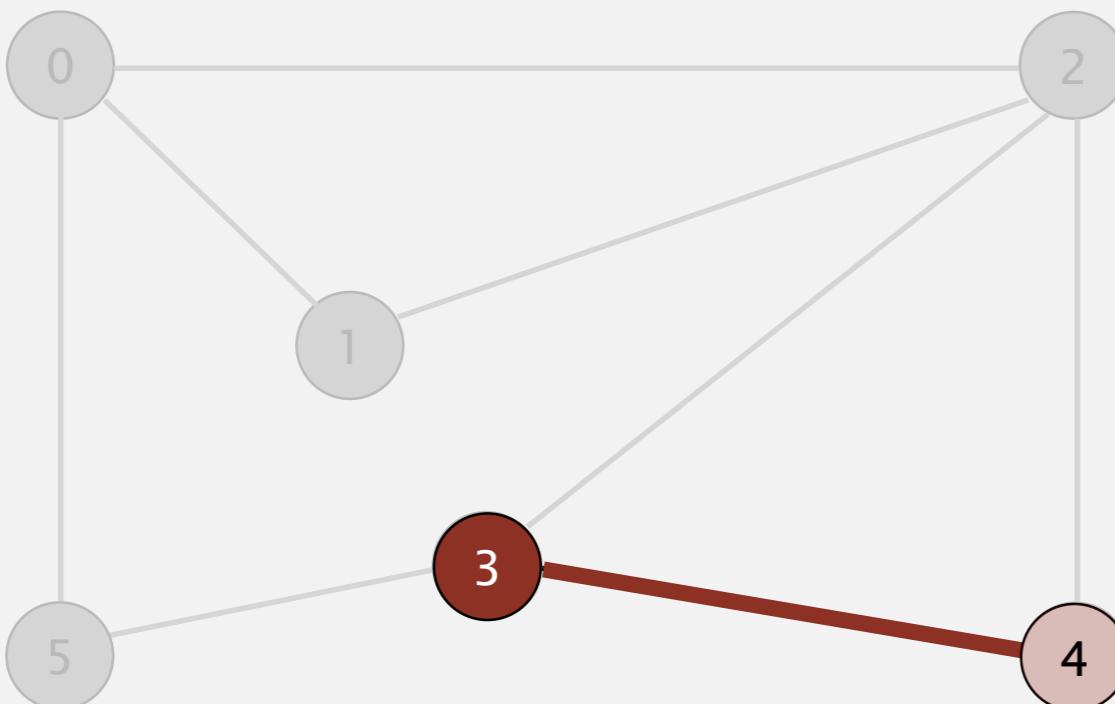
| queue | v | edgeTo[v] |
|-------|---|-----------|
|       | 0 | -         |
|       | 1 | 0         |
|       | 2 | 0         |
|       | 3 | 2         |
|       | 4 | 2         |
| 4     | 5 | 0         |

**dequeue 3**

# Breadth-first search

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices adjacent to  $v$  and mark them.



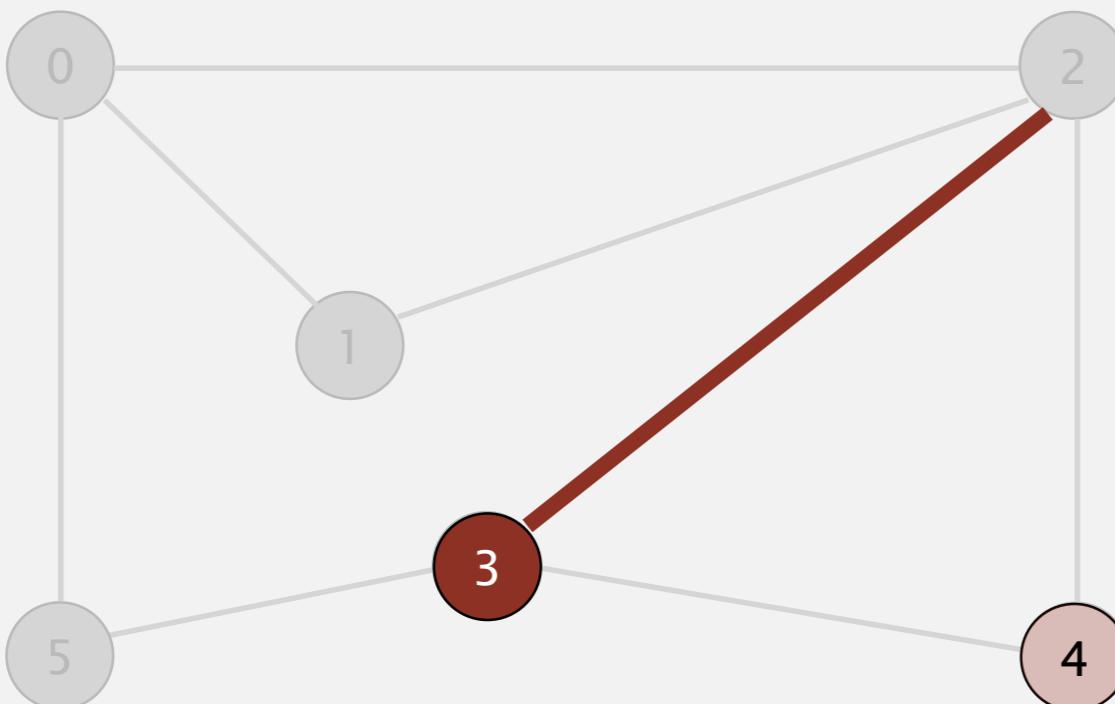
| queue | v | edgeTo[v] |
|-------|---|-----------|
|       | 0 | -         |
|       | 1 | 0         |
|       | 2 | 0         |
|       | 3 | 2         |
|       | 4 | 2         |
| 4     | 5 | 0         |

**dequeue 3**

# Breadth-first search

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices adjacent to  $v$  and mark them.



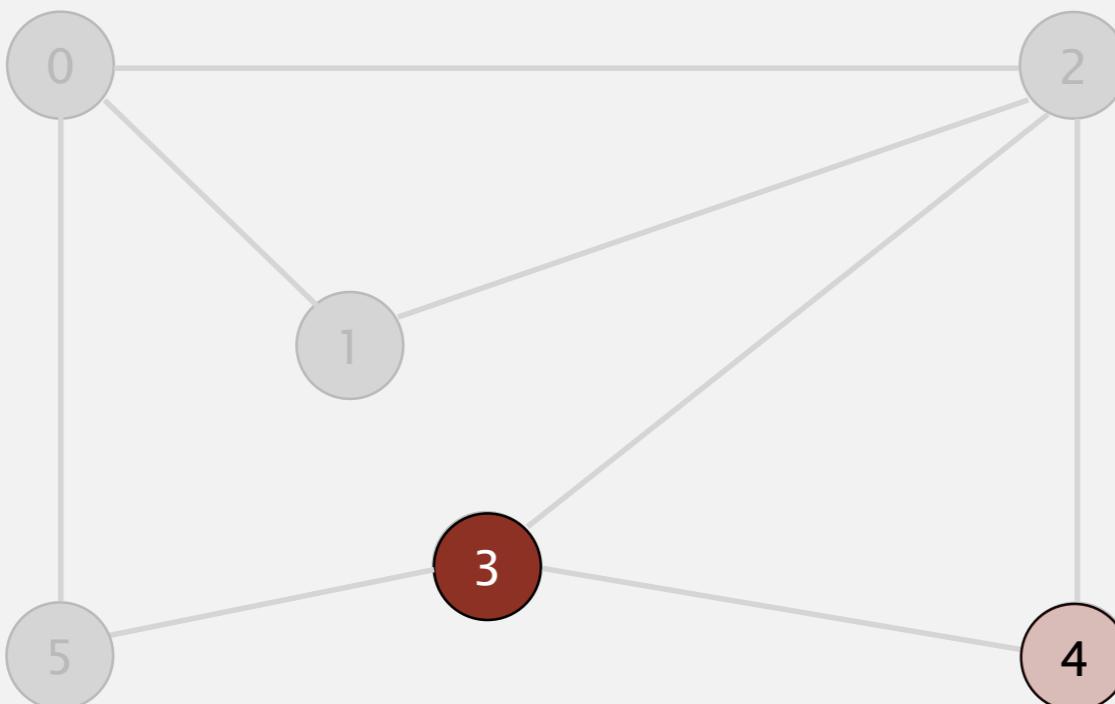
dequeue 3

| queue | v | edgeTo[v] |
|-------|---|-----------|
|       | 0 | -         |
|       | 1 | 0         |
|       | 2 | 0         |
|       | 3 | 2         |
|       | 4 | 2         |
| 4     | 5 | 0         |

# Breadth-first search

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices adjacent to  $v$  and mark them.



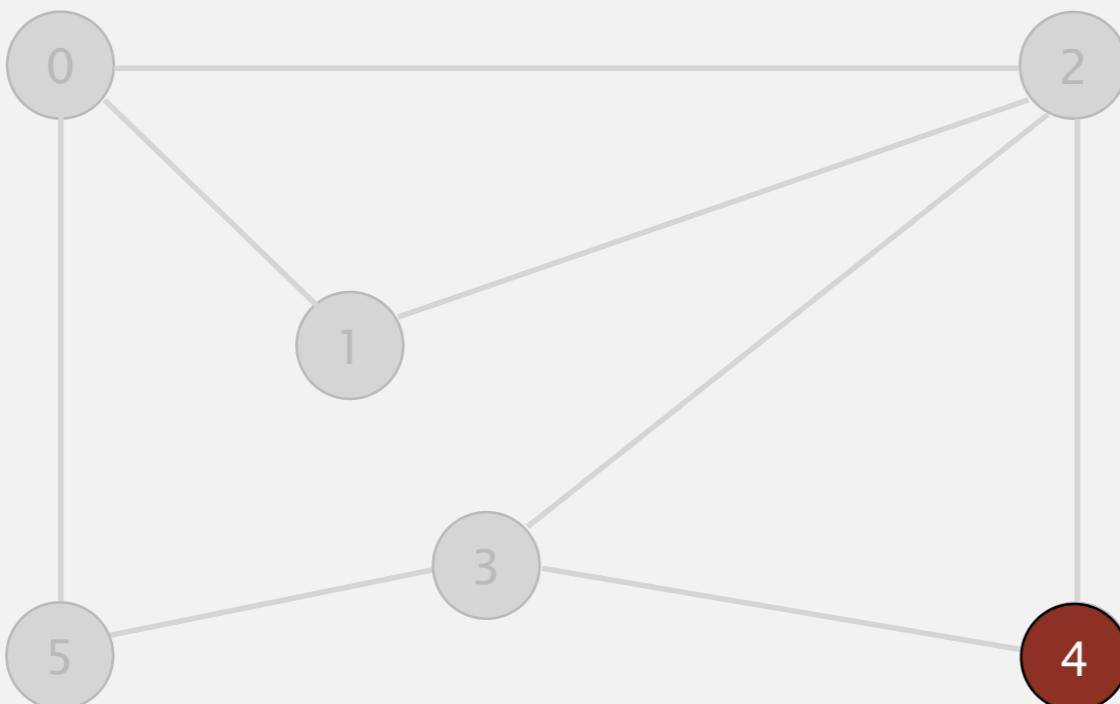
| queue | v | edgeTo[v] |
|-------|---|-----------|
|       | 0 | -         |
|       | 1 | 0         |
|       | 2 | 0         |
|       | 3 | 2         |
|       | 4 | 2         |
| 4     | 5 | 0         |

3 done

# Breadth-first search

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices adjacent to  $v$  and mark them.



queue

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

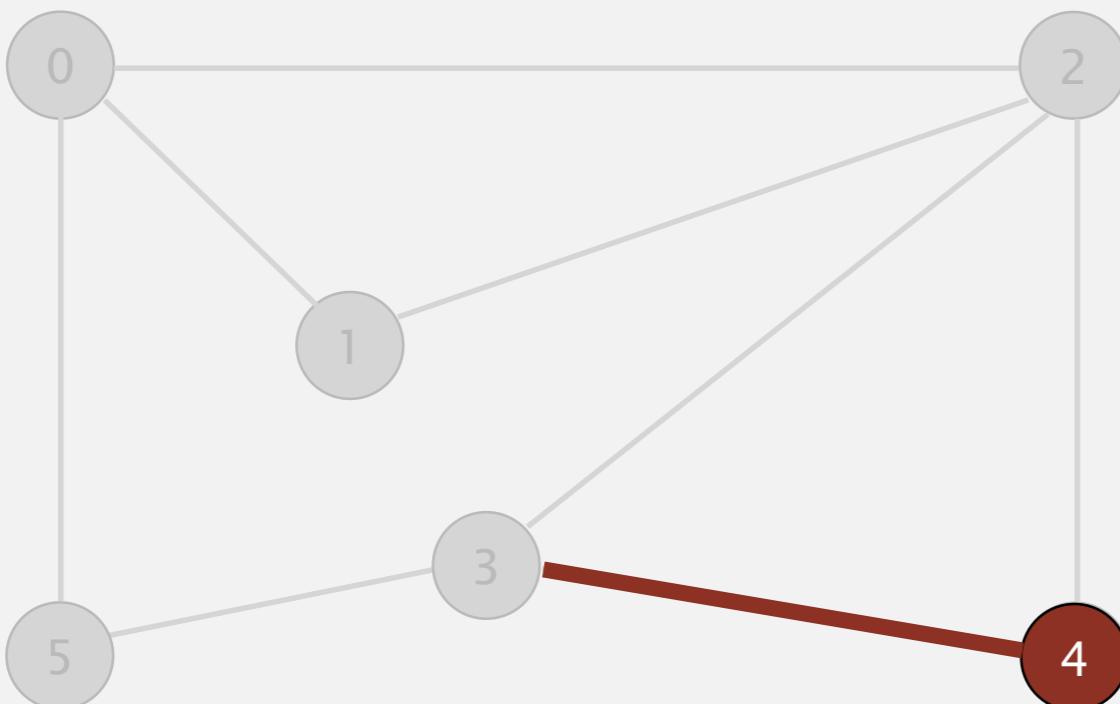
4

dequeue 4

# Breadth-first search

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices adjacent to  $v$  and mark them.



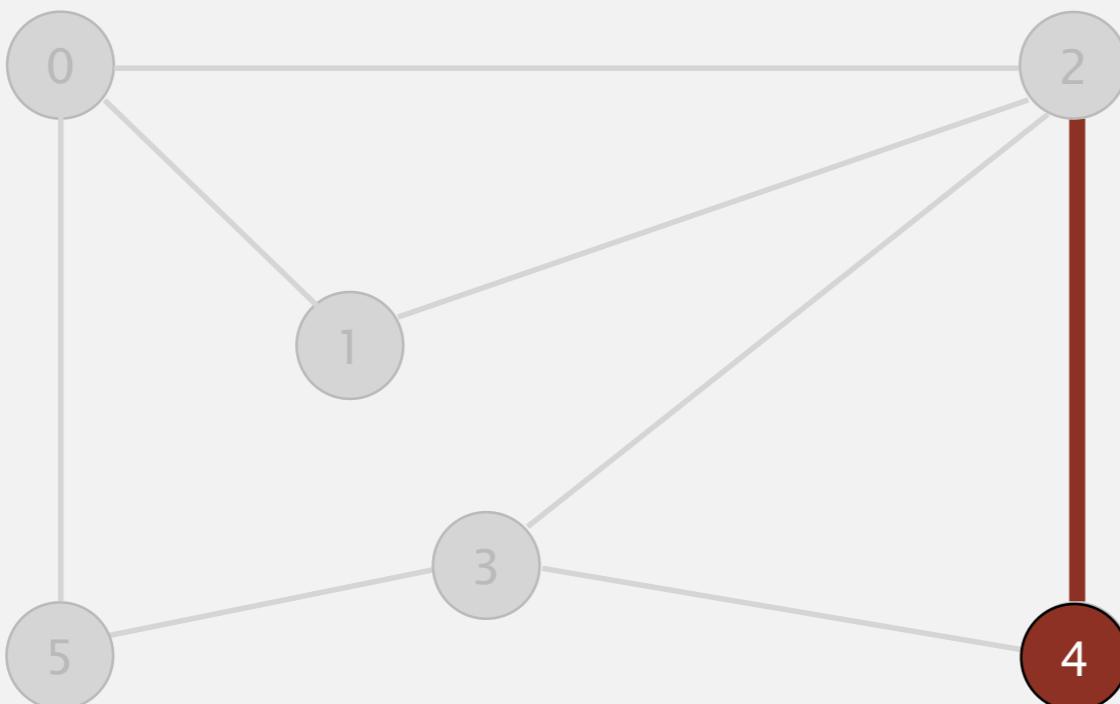
| queue | v | edgeTo[v] |
|-------|---|-----------|
|       | 0 | -         |
|       | 1 | 0         |
|       | 2 | 0         |
|       | 3 | 2         |
|       | 4 | 2         |
|       | 5 | 0         |

**dequeue 4**

# Breadth-first search

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices adjacent to  $v$  and mark them.



queue

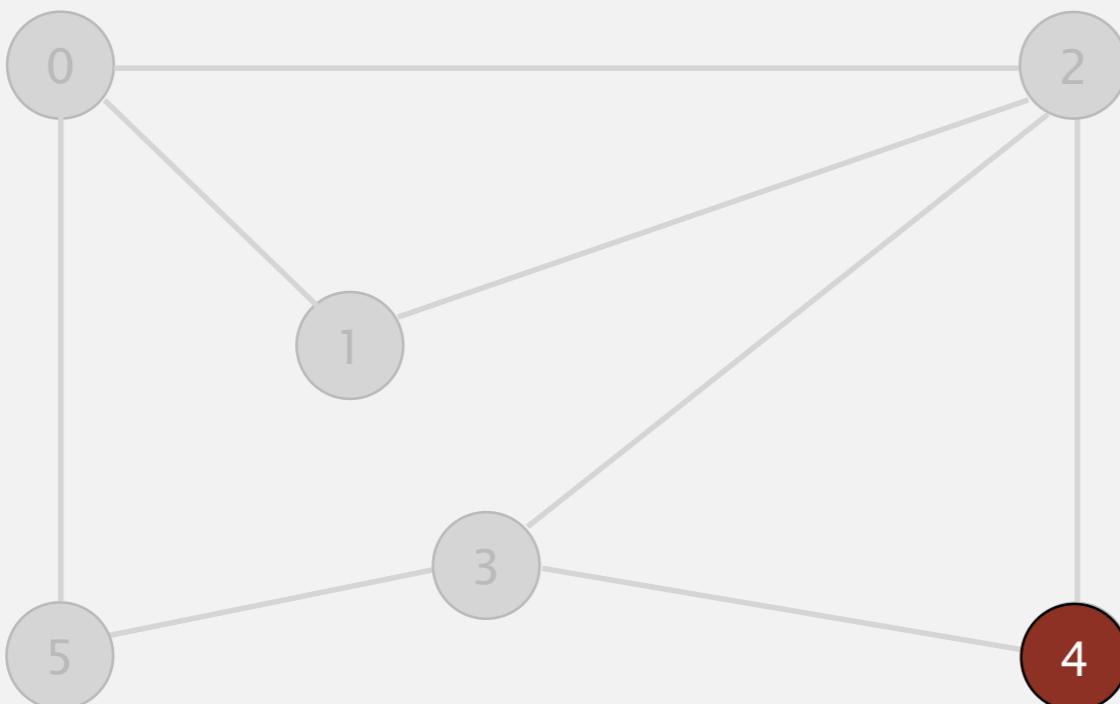
| v | edgeTo[v] |
|---|-----------|
| 0 | -         |
| 1 | 0         |
| 2 | 0         |
| 3 | 2         |
| 4 | 2         |
| 5 | 0         |

dequeue 4

# Breadth-first search

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices adjacent to  $v$  and mark them.



queue

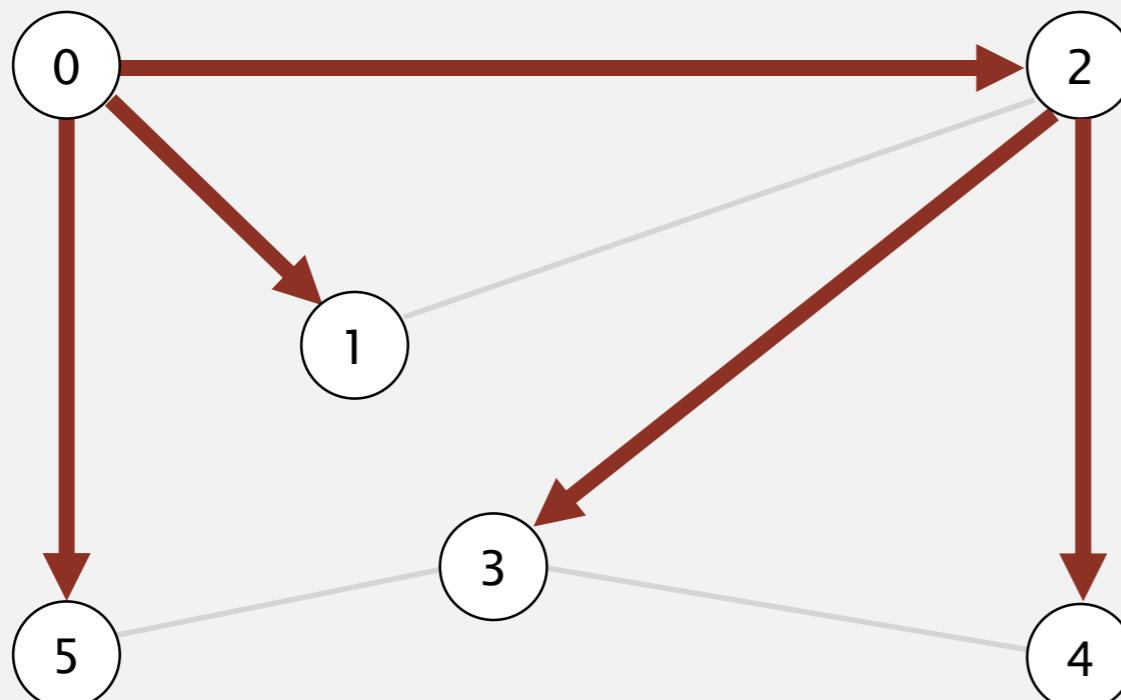
| v | edgeTo[v] |
|---|-----------|
| 0 | -         |
| 1 | 0         |
| 2 | 0         |
| 3 | 2         |
| 4 | 2         |
| 5 | 0         |

4 done

# Breadth-first search

Repeat until queue is empty:

- Remove vertex  $v$  from queue.
- Add to queue all unmarked vertices adjacent to  $v$  and mark them.



| $v$ | edgeTo[ $v$ ] |
|-----|---------------|
| 0   | -             |
| 1   | 0             |
| 2   | 0             |
| 3   | 2             |
| 4   | 2             |
| 5   | 0             |

done

# Breadth-first search

Depth-first search. Put unvisited vertices on a **stack**.

Breadth-first search. Put unvisited vertices on a **queue**.

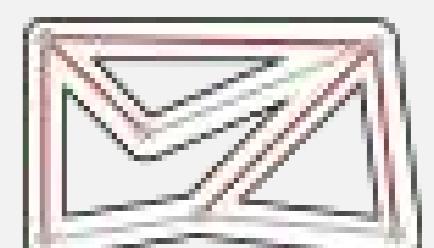
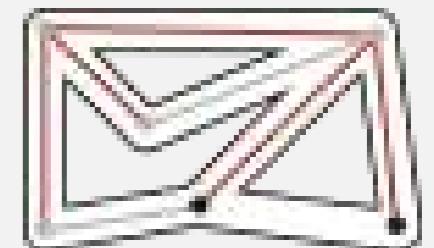
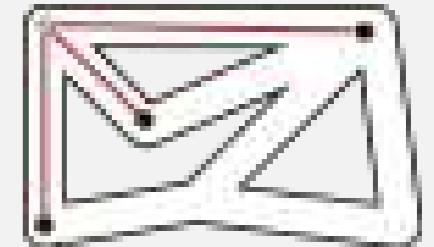
Shortest path. Find path from  $s$  to  $t$  that uses **fewest number of edges**.

## **BFS (from source vertex $s$ )**

Put  $s$  onto a **FIFO queue**, and mark  $s$  as visited.

Repeat until the queue is empty:

- remove the least recently added vertex  $v$
- add each of  $v$ 's unvisited neighbors to the queue,  
and mark them as visited.



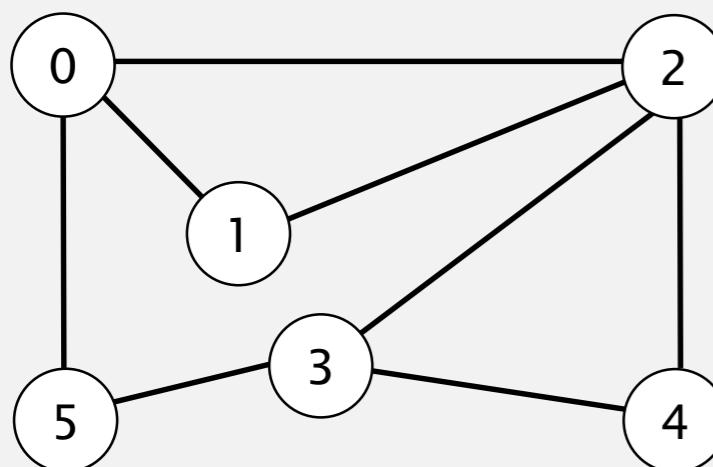
Intuition. BFS examines vertices in increasing distance from  $s$ .

# Breadth-first search properties

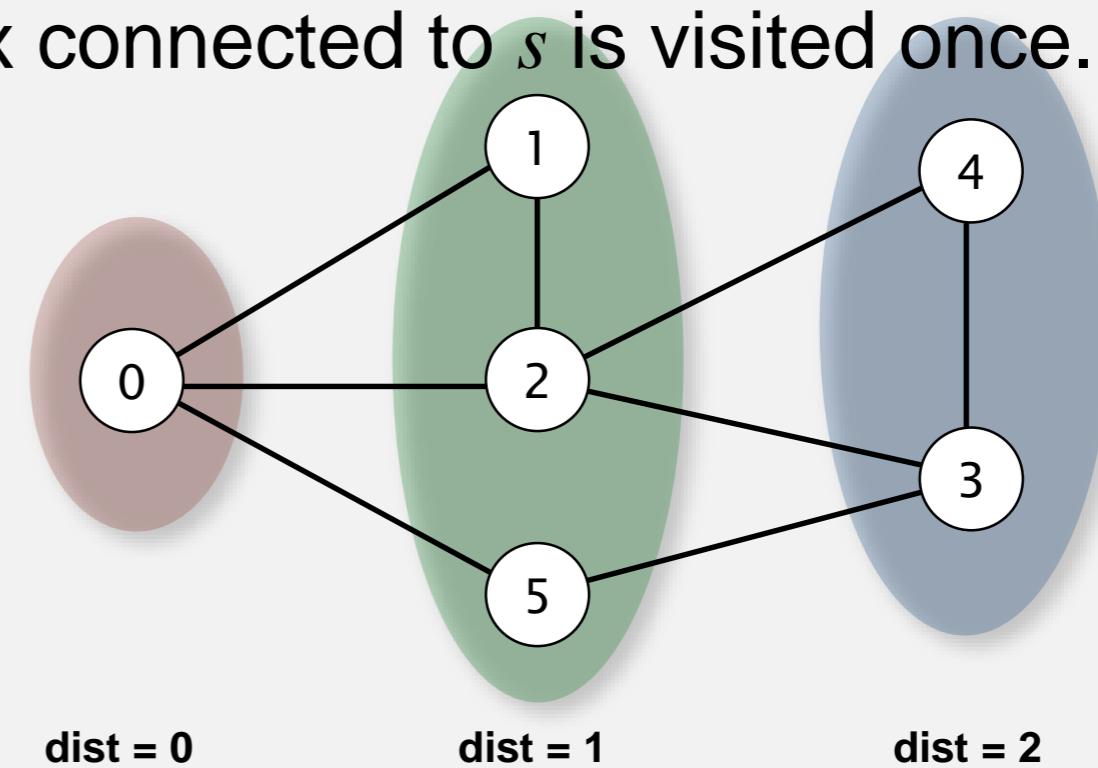
**Proposition.** BFS computes shortest path (number of edges) from  $s$  in a connected graph in time proportional to  $E + V$ .

**Pf. [correctness]** Queue always consists of zero or more vertices of distance  $k$  from  $s$ , followed by zero or more vertices of distance  $k + 1$ .

**Pf. [running time]** Each vertex connected to  $s$  is visited once.



standard drawing



# **BBM 201**

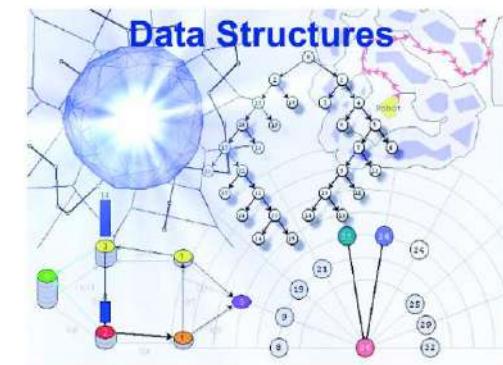
# **DATA STRUCTURES**

---

Linked Lists Examples



2018-2019 Fall



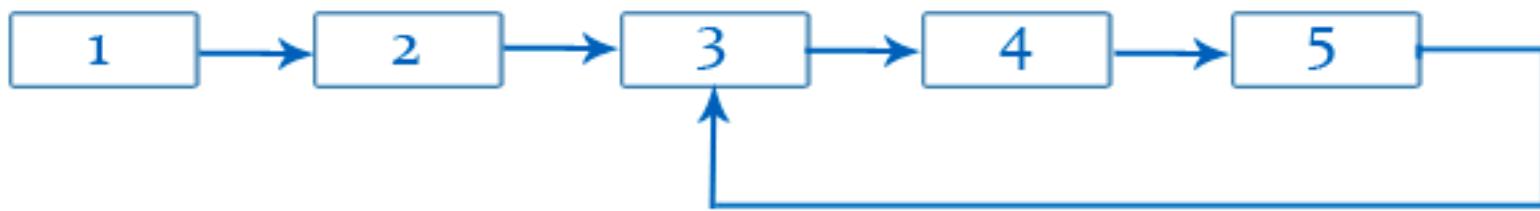
## **Q1 : Delete alternate nodes of a Linked List**

Given a Singly Linked List, starting from the second node delete all alternate nodes of it. For example, if the given linked list is 1->2->3->4->5 then your function should convert it to 1->3->5, and if the given linked list is 1->2->3->4 then convert it to 1->3.

# Q1 : Delete alternate nodes of a Linked List

```
struct node *deleteAlternative(struct node *head)
{
    if(head==NULL)
        return NULL;
    struct node *p = head;
    struct node *q = p->link;
    while( p != NULL && q != NULL)
    {
        p->link = q->link;
        free(q);
        p = p->link;
        if( p != NULL)
            q = p->link;
    }
    return head;
}
```

**Q2 : Write a program to detect loop in a Linked List.**



## **Q2 : Write a program to detect loop in a Linked List.**

```
void detectLoop(struct node *head)
{
    struct node *slowPtr,*fastPtr;
    slowPtr = head;
    fastPtr = head;

    while(slowPtr != NULL && fastPtr != NULL && fastPtr->next != NULL)
    {
        slowPtr = slowPtr->next;
        fastPtr = fastPtr->next->next;
        if (slowPtr == fastPtr)
        {
            printf("%s","Loop Detected");
            exit(0);
        }
    }
}
```

**Time Complexity : O(n)**

**Space Complexity : O(1)**

**Q3 : Write a C program to return the nth node from the end of a linked list.**

### **Q3 : Write a C program to return the nth node from the end of a linked list.**

**Brute Force Approach:** Let's assume length of linked list be 'L' , nth node from end =  $L - n + 1$  node from start.

Step 1: Iterate over the linked list and determine its length.

Step 2: Now using the above mentioned formula , we can determine the position of the required node.

Step 3: So we iterate the linked list again till we reach the required node.

Step 4: Return the required node.

**Q3 : Write a C program to return the nth node from the end of a linked list.**

**Solution 2 :** Suppose one needs to get to the 6th node from the end in this LL. First, just keep on incrementing the first pointer (ptr1) till the number of increments cross n (which is 6 in this case)

Step 1 : 1(ptr1,ptr2) -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10

Step 2 : 1(ptr2) -> 2 -> 3 -> 4 -> 5 -> 6(ptr1) -> 7 -> 8 -> 9 -> 10

Now, start the second pointer (ptr2) and keep on incrementing it till the first pointer (ptr1) reaches the end of the LL.

Step 3 : 1 -> 2 -> 3 -> 4(ptr2) -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 (ptr1)

So here you have!, the 6th node from the end pointed to by ptr2!

### Q3 : Write a C program to return the nth node from the end of a linked list.

```
mynode * nthNode(mynode *head, int n)
{
    mynode *ptr1,*ptr2; int count;
    if(!head)  return(NULL);
    ptr1 = ptr2 = head;
    count = 0;
    while(count < n) {
        count++;
        if((ptr1->next) != NULL)
            ptr1 =ptr1->next;
        else
            return NULL;
    }
    while((ptr1->next) != NULL)
    {
        ptr1 = ptr1->next;
        ptr2=ptr2->next;
    }
    return(ptr2);
}
```

**Time Complexity :** O(n)

**Q4: Write a function to get the intersection point of two Linked lists (Y Shape).**

**Q4: Write a function to get the intersection point of two Linked lists (Y Shape).**

**Brute Force Approach :** Using 2 for loops.

Outer loop will be for each node of the 1st list

Inner loop will be for 2nd list.

**Q4: Write a function to get the intersection point of two Linked lists (Y Shape).**

**Solution 2:**

Find the length of both the linked list

Find the difference in length (d)

For the longer linked list, traverse d node and check the addresses with another linked list.

## **Q4: Write a function to get the intersection point of two Linked lists (Y Shape).**

```
int getIntesectionNode(struct node* head1, struct node* head2)
{
    int c1 = getCount(head1);
    int c2 = getCount(head2);
    int d;
    d = abs( c1 - c2);
    if(c1 > c2)
    {
        return getIntesectionNode(d, head1, head2);
    }
    else
    {
        return getIntesectionNode(d, head2, head1);
    }
}
```

#### **Q4: Write a function to get the intersection point of two Linked lists (Y Shape).**

```
int getIntesectionNode(int d, struct node* head1, struct node* head2){  
    int i;  
    struct node* current1 = head1;  
    struct node* current2 = head2;  
    for(i = 0; i < d; i++)  
    {  
        if(current1 == NULL)  
            return -1;  
        current1 = current1->next;  
    }  
    while(current1 != NULL && current2 != NULL)  
    { // we are comparing the addresses not data  
        if(current1 == current2)  
            return current1->data;  
        current1= current1->next;  
        current2= current2->next;  
    }  
    return -1;  
}
```

## **Q4: Write a function to get the intersection point of two Linked lists (Y Shape).**

```
int getCount(struct node* head)
{
    struct node* current = head;
    int count = 0;
    while (current != NULL)
    {
        count++;
        current = current->next;
    }
    return count;
}
```

**Time Complexity :**  $O(m+n)$

**Space Complexity :**  $O(1)$



Hacettepe University  
Computer Engineering Department

# BBM203 Software Laboratory I

Week 14  
Recitation VII

Fall 2018

# Example 1: Triangular Matrix – to recap

- Suppose we have an upper triangular B matrix where non-zero elements are confined at diagonal and its above and also suppose we replace every element into an array ARA with column-major order.

$$B = \begin{pmatrix} b_{00} & b_{01} & b_{02} & b_{03} \\ \textcolor{red}{0_{10}} & b_{11} & b_{12} & b_{13} \\ \textcolor{red}{0_{20}} & \textcolor{red}{0_{21}} & b_{22} & b_{23} \\ \textcolor{red}{0_{30}} & \textcolor{red}{0_{31}} & \textcolor{red}{0_{32}} & b_{33} \end{pmatrix}$$

$$\begin{array}{ccccccc} 0 & 1 & 2 & 3 & \dots \\ ARA = [b_{00} & b_{01} & b_{11} & b_{02} & \dots & b_{33}] \end{array}$$

- Write a method that accepts row (i), column (j) indexes and size (n) then returns (with O(1) complexity)
  - 2; if they point somewhere out of the matrix,
  - 1; if they point somewhere in lower triangular matrix,
  - The corresponding index on array ARA; otherwise (e.g., 9 : when  $i, j = 3$ )

# Sol. 1: Triangular Matrix – to recap

$$B = \begin{pmatrix} b_{00} & b_{01} & b_{02} & b_{03} \\ 0_{10} & b_{11} & b_{12} & b_{13} \\ 0_{20} & 0_{21} & b_{22} & b_{23} \\ 0_{30} & 0_{31} & 0_{32} & b_{33} \end{pmatrix}$$

$$ARA = [b_{00} \quad b_{01} \quad b_{11} \quad b_{02} \quad \cdots \quad b_{33}]$$

```
int upper_triangular_access (int i, int j, int n) {
    if (i < 0 || i >= n || j < 0 || j >= n)
        return -2;
    else if (j >= i)
        return (j * (j+1) / 2) + i;
    else
        return -1;
}
```

# Example 2: Sparse Matrix – to recap

- Suppose  $a$  and  $b$  sparse matrices are stored with triple structure in the memory. Write a method named **seyrek\_matris\_cikar** that subtracts  $b$  from  $a$  and transfers the outcome into matrix  $c$  with a less complexity than  $O(a[0].val+b[0].val)$ .
- Note: Your implementation should use a macro **KARSILASTIR** and a procedure **cyetasi**.

```
#define MAX_TERIM_SAY 101
#define KARSILASTIR(x,y) (((x)<(y)) ? -1 : ((x) == (y)) ? 0 : 1)

void cyetasi(int row, int col, int val, int *pc, terim c[]) {
    if (*pc => MAX_TERIM_SAY) {
        printf('\n not feasible');
        exit(-1);
    }
    c[*pc].row = row;
    c[*pc].col = col;
    c[(*pc)++].val = val;
}
```

```
typedef struct {
    int row;
    int col;
    int val; } terim;
```

example:

| <b>a</b>    | <b>row</b> | <b>col</b> | <b>val</b> |
|-------------|------------|------------|------------|
| <b>a[0]</b> | 5          | 5          | 4          |
| <b>a[1]</b> | 0          | 2          | 9          |
| <b>a[2]</b> | 1          | 3          | 3          |
| <b>a[3]</b> | 2          | 0          | 1          |
| <b>a[4]</b> | 4          | 4          | 6          |

-

| <b>b</b>    | <b>row</b> | <b>col</b> | <b>val</b> |
|-------------|------------|------------|------------|
| <b>b[0]</b> | 5          | 5          | 5          |
| <b>b[1]</b> | 0          | 1          | -6         |
| <b>b[2]</b> | 0          | 2          | 1          |
| <b>b[3]</b> | 1          | 2          | 4          |
| <b>b[4]</b> | 2          | 0          | 1          |
| <b>b[5]</b> | 3          | 2          | -7         |

=

| <b>c</b>    | <b>row</b> | <b>col</b> | <b>val</b> |
|-------------|------------|------------|------------|
| <b>c[0]</b> | 5          | 5          | 6          |
| <b>c[1]</b> | 0          | 1          | 6          |
| <b>c[2]</b> | 0          | 2          | 8          |
| <b>c[3]</b> | 1          | 2          | -4         |
| <b>c[4]</b> | 1          | 3          | 3          |
| <b>c[5]</b> | 3          | 2          | 7          |
| <b>c[6]</b> | 4          | 4          | 6          |

# Sol. 2: Sparse Matrix – to recap

```

void seyrek_matris_cikar (terim a[], terim b[], terim c[]) {
    int pa = 1, pb = 1, pc = 1, diff;
    if ( a[0].row != b[0].row || a[0].col != b[0].col ) {
        printf('\n Incompatible array size!');
        exit(0);
    }
    while (pa <= a[0].val && pb <= b[0].val)
        switch ( KARSILASTIR(a[pa].row, b[pb].row) ) {
            case -1: // a[pa].row < b[pb].row
                cyetasi(a[pa].row, a[pa].col, a[pa].val, &pc, c);
                pa++; break;
            case 1: // a[pa].row > b[pb].row
                cyetasi(b[pb].row, b[pb].col, -b[pb].val, &pc, c);
                pb++; break;
            case 0: // a[pa].row = b[pb].row
                switch ( KARSILASTIR(a[pa].col, b[pb].col) ) {
                    case -1: // a[pa].col < b[pb].col
                        cyetasi(a[pa].row, a[pa].col, a[pa].val, &pc, c);
                        pa++; break;
                    case 1: // a[pa].col > b[pb].col
                        cyetasi(b[pb].row, b[pb].col, -b[pb].val, &pc, c);
                        pb++; break;
                    case 0: // a[pa].col = b[pb].col
                        fark = a[pa].val - b[pb].val;
                        if (fark)
                            cyetasi(b[pb].row, b[pb].col, fark, &pc, c);
                        pa++; pb++; break;
                }
            }
        for (;pa <= a[0].val; pa++)
            cyetasi(a[pa].row, a[pa].col, a[pa].val, &pc, c);
        for (;pb <= b[0].val; pb++)
            cyetasi(b[pb].row, b[pb].col, -b[pb].val, &pc, c);
        c[0].row = a[0].row; c[0].col = a[0].col; c[0].val = pc - 1;
    }
}

```

| a    | row | col | val |
|------|-----|-----|-----|
| a[0] | 5   | 5   | 4   |
| a[1] | 0   | 2   | 9   |
| a[2] | 1   | 3   | 3   |
| a[3] | 2   | 0   | 1   |
| a[4] | 4   | 4   | 6   |

| b    | row | col | val |
|------|-----|-----|-----|
| b[0] | 5   | 5   | 5   |
| b[1] | 0   | 1   | -6  |
| b[2] | 0   | 2   | 1   |
| b[3] | 1   | 2   | 4   |
| b[4] | 2   | 0   | 1   |
| b[5] | 3   | 2   | -7  |

| c    | row | col | val |
|------|-----|-----|-----|
| c[0] | 5   | 5   | 6   |
| c[1] | 0   | 1   | 6   |
| c[2] | 0   | 2   | 8   |
| c[3] | 1   | 2   | -4  |
| c[4] | 1   | 3   | 3   |
| c[5] | 3   | 2   | 7   |
| c[6] | 4   | 4   | 6   |

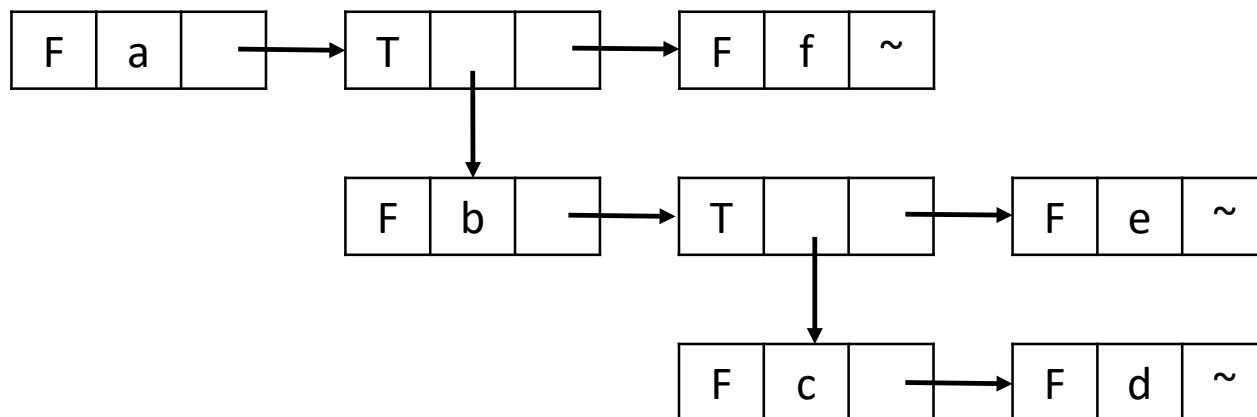
# Example 3: Generalized lists

- Write a procedure that constructs a generalized list depending on a provided input.

Example:

*input* -> (a,(b,(c,d),e),f)

*output* ->



# Sol. 3.1: Generalized lists (iterative approach)

```
#define compare(letter) (((letter) == '(') ? 0 : ((letter) == ')') ? 1 : ((letter) == ',') ? 2 : 3);

typedef enum {T, F} boolean;

struct node {
    boolean flag;
    union{
        char key;
        struct node* down_address;
    };
    struct node* next_address;
};

struct stack {
    struct node* address;
};

struct stack* _stack[MAX_ITEM_SIZE];
int stack_pointer = -1;

struct node* pop() {
    return _stack[stack_pointer--];
}

void push (struct node* address) {
    _stack[++stack_pointer] = address;
}

struct node* construct_gen_list(char* inp) {
    int char_pos = 0, switch_key;
    struct node* n = (struct node*)malloc(sizeof(struct node));
    char letter;
    while (char_pos < strlen(inp)) {
        letter = inp[char_pos];
        switch_key = compare(letter);
        switch (switch_key) {
            case 0:
                n->flag = T;
                n->next_address = NULL;
                if (inp[char_pos + 1] != ')') {
                    n->down_address = (struct node*)malloc(sizeof(struct node));
                    push(n);
                    n = n->down_address;
                }
                else {
                    char_pos++;
                    n->down_address = NULL;
                }
                break;
            case 1:
                n = pop();
                break;
            case 2:
                n->next_address = (struct node*)malloc(sizeof(struct node));
                n = n->next_address;
                break;
            case 3:
                n->key = letter;
                n->flag = F;
                n->next_address = NULL;
                break;
        }
        char_pos++;
    }
    return n;
}
```

# Sol. 3.2: Generalized lists (recursive approach)

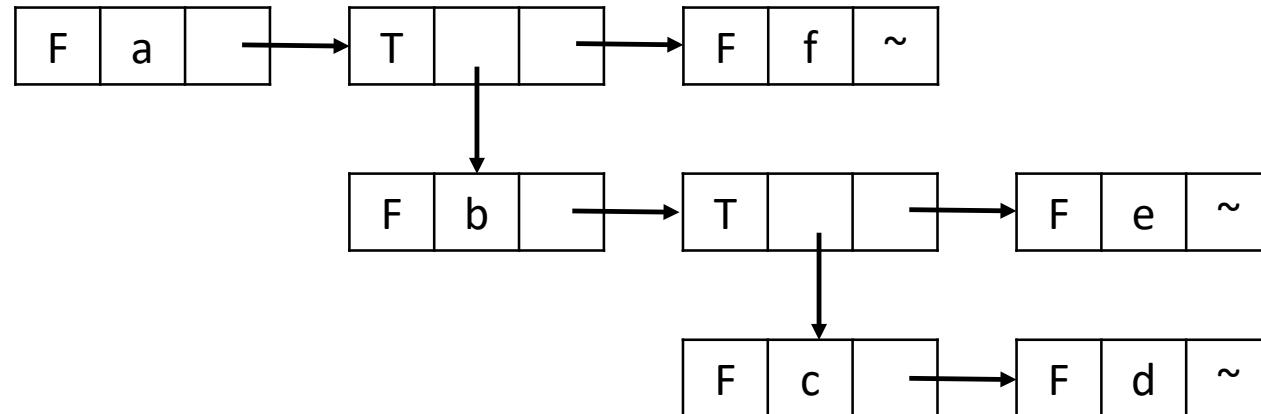
```
struct node* construct_gen_list(struct node* n, char* inp, int* ind) {  
  
    if (inp[*ind] == '(') {  
        (*ind)++;  
        n->flag = T;  
        n->next_address = NULL;  
        n->down_address = inp[*ind] == ')' ? NULL : construct_gen_list((struct node*)malloc(sizeof(struct node)), inp, ind);  
    }  
  
    if (inp[*ind] == ')') {  
        (*ind)++;  
    }  
  
    if (inp[*ind] >= 'a' && inp[*ind] <= 'z') {  
        n->flag = F;  
        n->key = inp[*ind];  
        (*ind)++;  
        n->next_address = NULL;  
    }  
  
    if (inp[*ind] == ',') {  
        (*ind)++;  
        n->next_address = construct_gen_list((struct node*)malloc(sizeof(struct node)), inp, ind);  
    }  
  
    return n;  
}
```

# Example 4: Generalized lists

- Write a recursion that prints out the corresponding input by traversing a given generalized list.

Example:

*input ->*



*output -> (a,(b,(c,d),e),f)*

# Sol. 4: Generalized lists

```
void print_generalized_list(struct node * n) {  
  
    if (n->flag == T) {  
        printf("(");  
        if (n->down_address != NULL)  
            print_generalized_list(n->down_address);  
        printf(")");  
    } else  
        printf("%c",n->key);  
  
    if (n->next_address != NULL) {  
        printf(",");  
        print_generalized_list(n->next_address);  
    }  
}
```