

Course: Data Structures and Algorithms

Instructor: Homeyra Pourmohammadali

Introduction to Data Structures and Algorithms

Motivation

Messy Room: You are unable to find items you are looking for.

Organized Room: You arrange and keep everything in such a structure that you can easily search and find items.

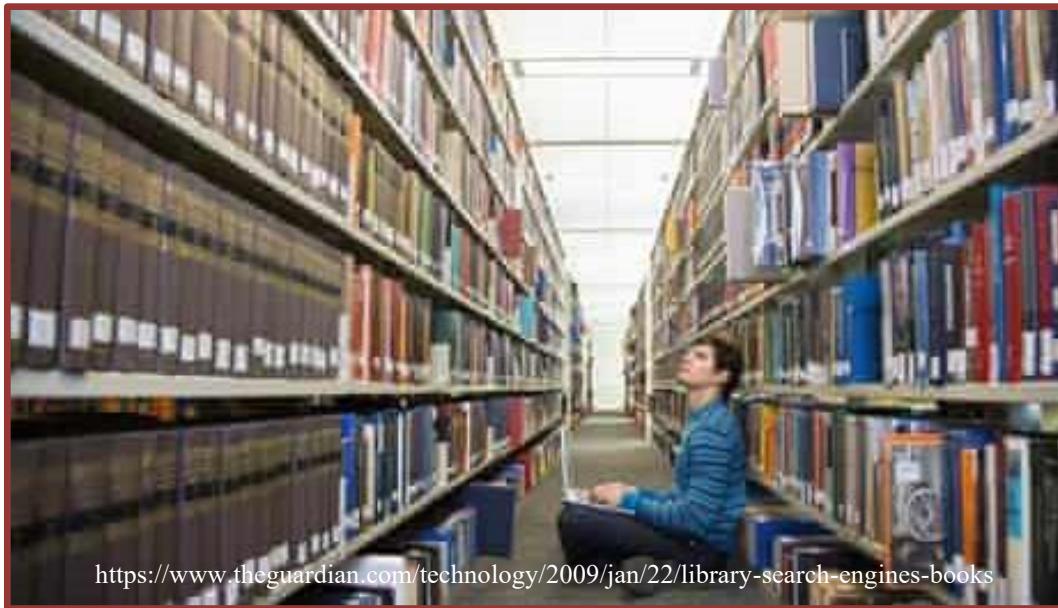


<https://www.pinterest.ca/pin/690106342880417383/>

Think about the importance of organizing room to access and use items in the room.

Motivation

In library: It will be frustrating to search and find a specific book, if they are not organized in a way that make any sense.



Library Science involves collection management, classification, cataloging, search methods, referencing, etc.

Motivation

- Similar to items in the room and books in the library, data also need to be structured and organized.
- In data structures (DS), we are interested to study the ways that information on our computers can be organized.
- Computer scientists process data using different type of data structures for the best data organization methods.



Learning Outcomes

By the end of this lecture you will be able to:

1. define data structure and list its different types
2. understand the components and attributes of an algorithm and see some of its real-life examples
3. find out the need for abstraction and define the abstract data type (ADT)
4. identify modules in software design

Introductions

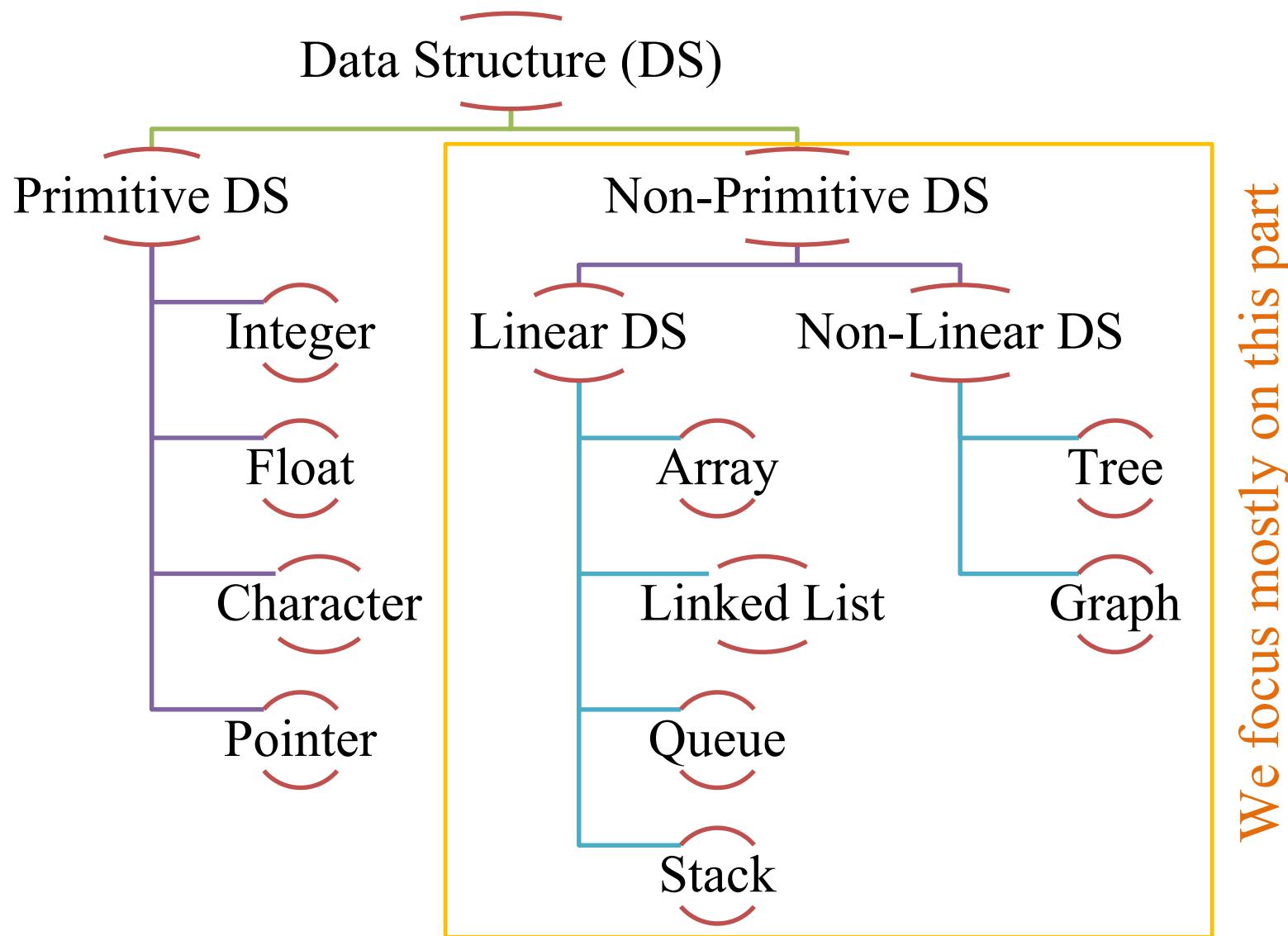
- The applications become more complex, with increasing data.
- Some hurdles that application may face are:



What is Data Structure?

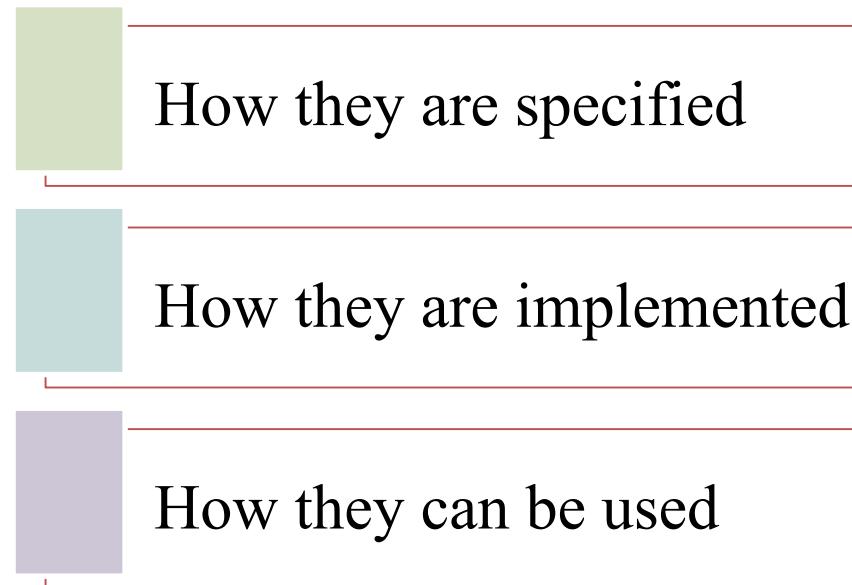
- Data structure (DS) is a coherent organization of related data items for systematic storage and manipulation, that can be accessed, queried and updated quickly and easily.
- Data structures are main components for creation of quick and powerful algorithms, and make codes more readable and understandable.
- Features of DS:
 1. They can be decomposed into their component elements.
 2. Elements arrangement affects how they are accessed.
 3. Both the elements arrangement and the access method can be encapsulated.

Types of Data Structures



Data Structures

- Data structure (DS) are usually discussed from different points of view:



- The object-oriented view of data objects will be shown.
- C++ language will be used for OOP and implementation.

What is an Algorithm?

Simple Definition: a set of steps to accomplish a task

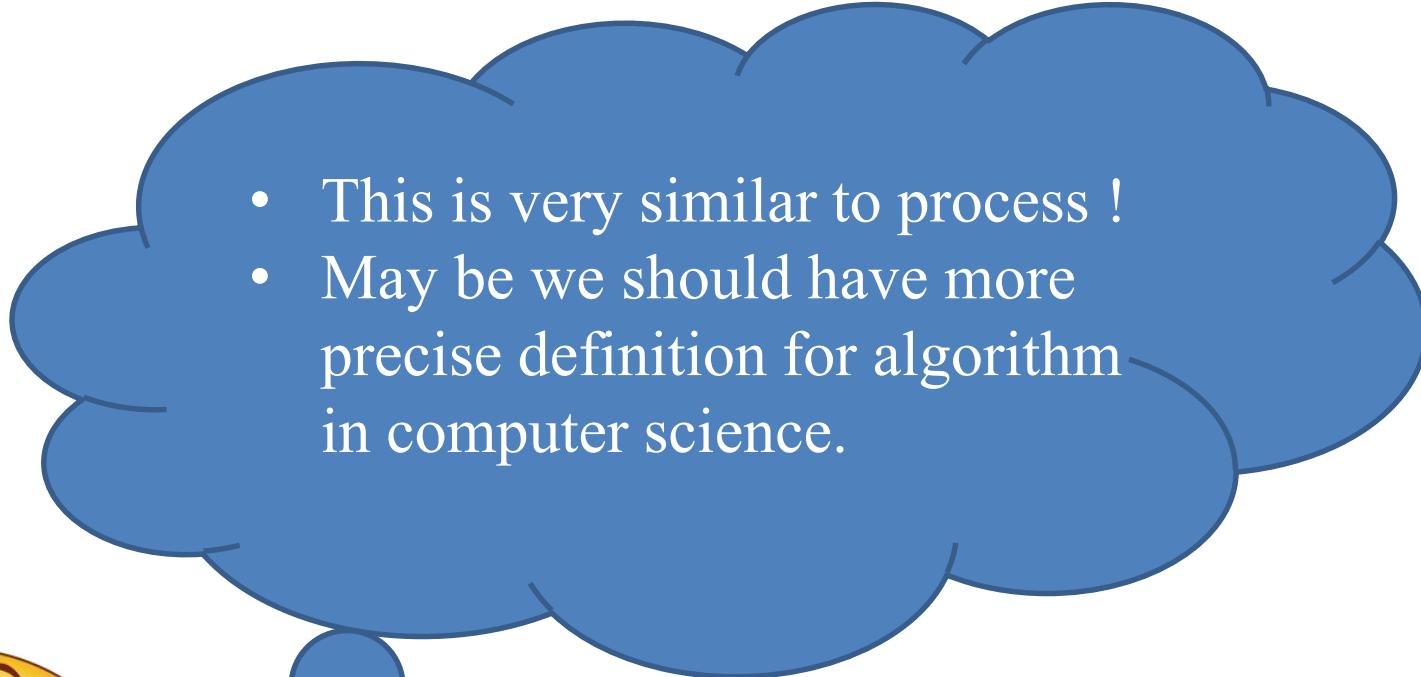
Then, am I using an algorithm while:

- making a sandwich?
- baking a cake?
- riding a bike?

YES !



What is an Algorithm?

- 
- 
- This is very similar to process !
 - May be we should have more precise definition for algorithm in computer science.

What is an Algorithm?

In computer science, an **algorithm** is:

- a finite sequence of unambiguous instructions performed to achieve a goal.
- not a solution, but instead, a precisely defined procedure for deriving solutions.

Process is a sequence of operations performed to achieve a goal and does not have to terminate.

- Examples: Living and breathing



Algorithms

Example. What are the important steps of an algorithm that can convert Canadian Dollar (CAD) to Japanese Yen (JPY)?

Algorithm: Convert Canadian Dollars to Yen

- **Inputs:** A value in Canadian Dollars
- **Outputs:** A corresponding value in Yen
- **Step 1.** Obtain CAD-to-JPY conversion rate from the internal currency conversion table.
- **Step 2.** Multiply inputted value in Canadian Dollars with the CAD-to-JPY conversion rate.
- **Step 3.** Output the multiplication result in Yen, and terminate.

In the next slide you will see the first

“You Try” Activity

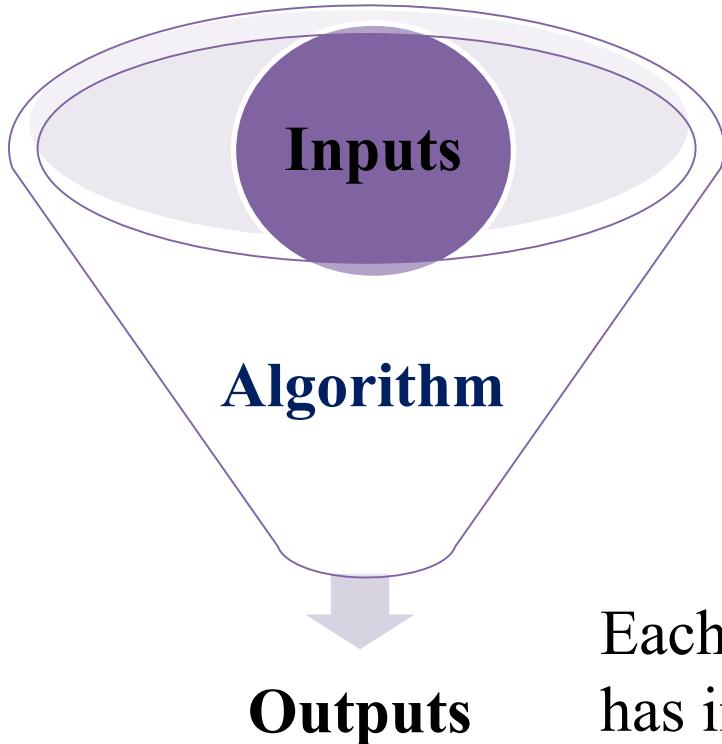
You may have some of these in each lecture. Whenever, you see them, please pause the video, read the question and think about it, and write your own thinking or answer in your notes. The “You Try” solutions/answers will be provided later.

You Try 1. What are the important steps of an algorithm that can translate a French word to an English word?

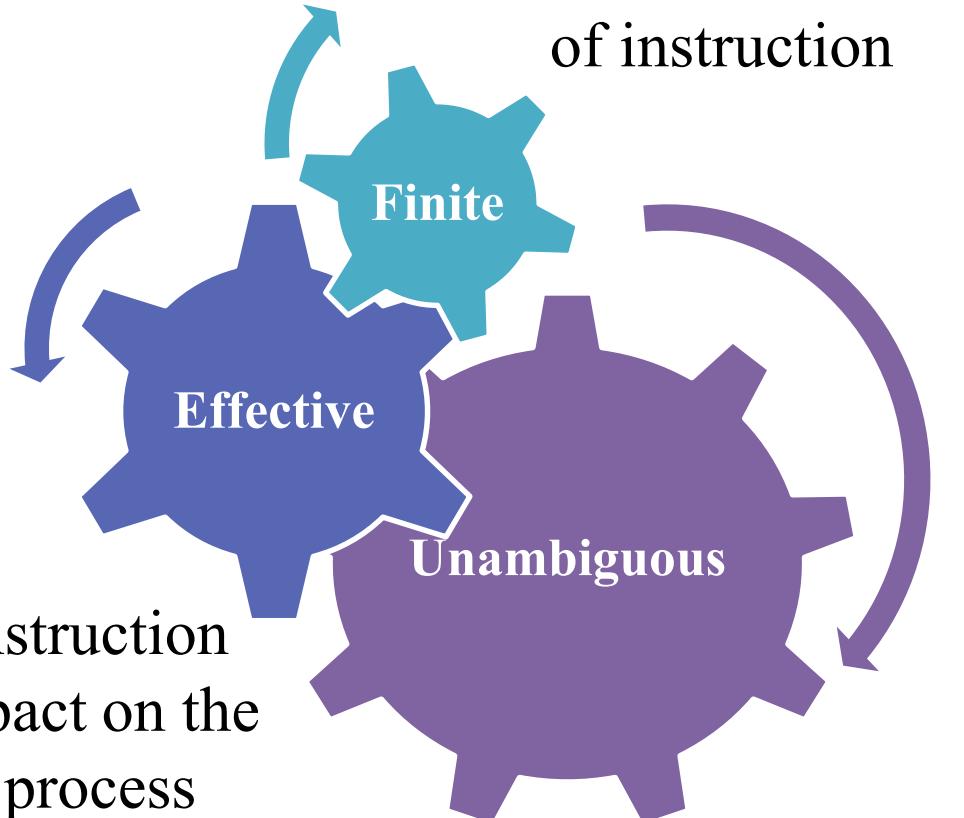
Algorithm: Translate a French word to an English word

- **Inputs:**
- **Outputs:**
- **Steps:**

Main Attributes of Algorithms

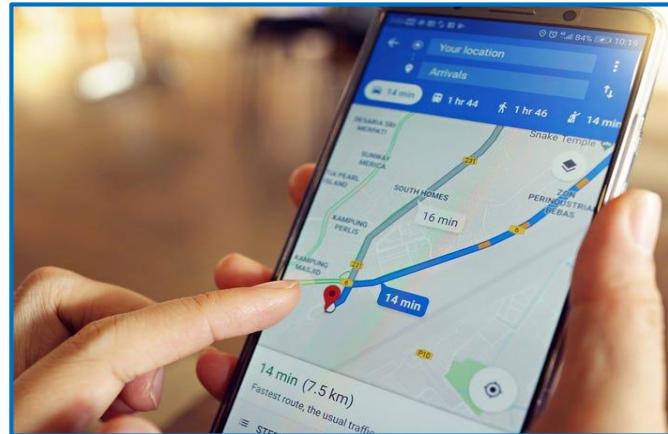
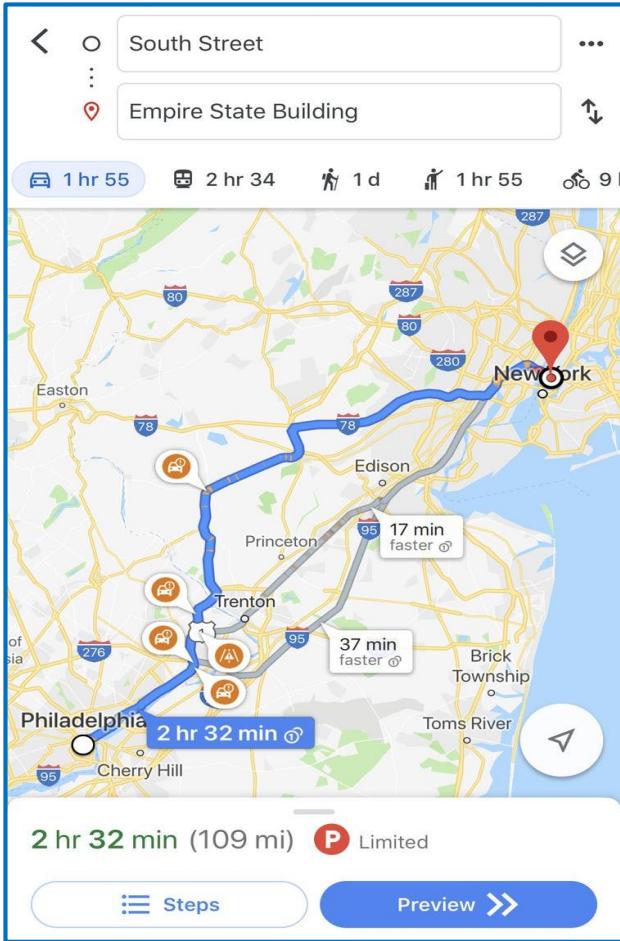


Each instruction
has impact on the
overall process



Simple and clear
instructions

Algorithms Tied to Our Real-Life



Google Map figures out how to get from one location to another using the “**Rout-finding algorithms.**”

<https://www.businessinsider.com/how-to-change-route-on-google-maps>

Algorithms Tied to Our Real-Life

- Nasa arranges and rearranges the solar panels on the international space station using the “optimization and scheduling algorithms.”



https://www.nasa.gov/mission_pages/station/structure/elements/solar_arrays-about.html

Algorithms Tied to Our Real-Life



<http://graphics.pixar.com/library/soulVolumetricChars/>

Pixar color makes a 3D model of a character based on a lighting in a virtual room using the “rendering algorithms.”



Algorithms Tied to Our Real-Life

- Live videos can quickly be transmitted across the internet using the “audio/video compression algorithms.”



<https://www.epiphan.com/blog/av-over-ip-remote-video-production/>

How to Resolve the Complexity Issues?

- Implementation of the algorithms can become quite complex depending on the problem.



When we start a car, we don't know about all its complex mechanical parts.

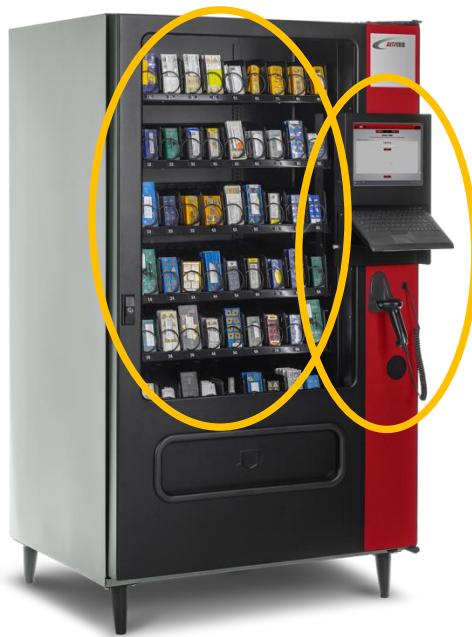


When we pay a vending machine, we don't know how correct change is calculated.

The inner working details are kept hidden from the user.

Meaning of Abstraction

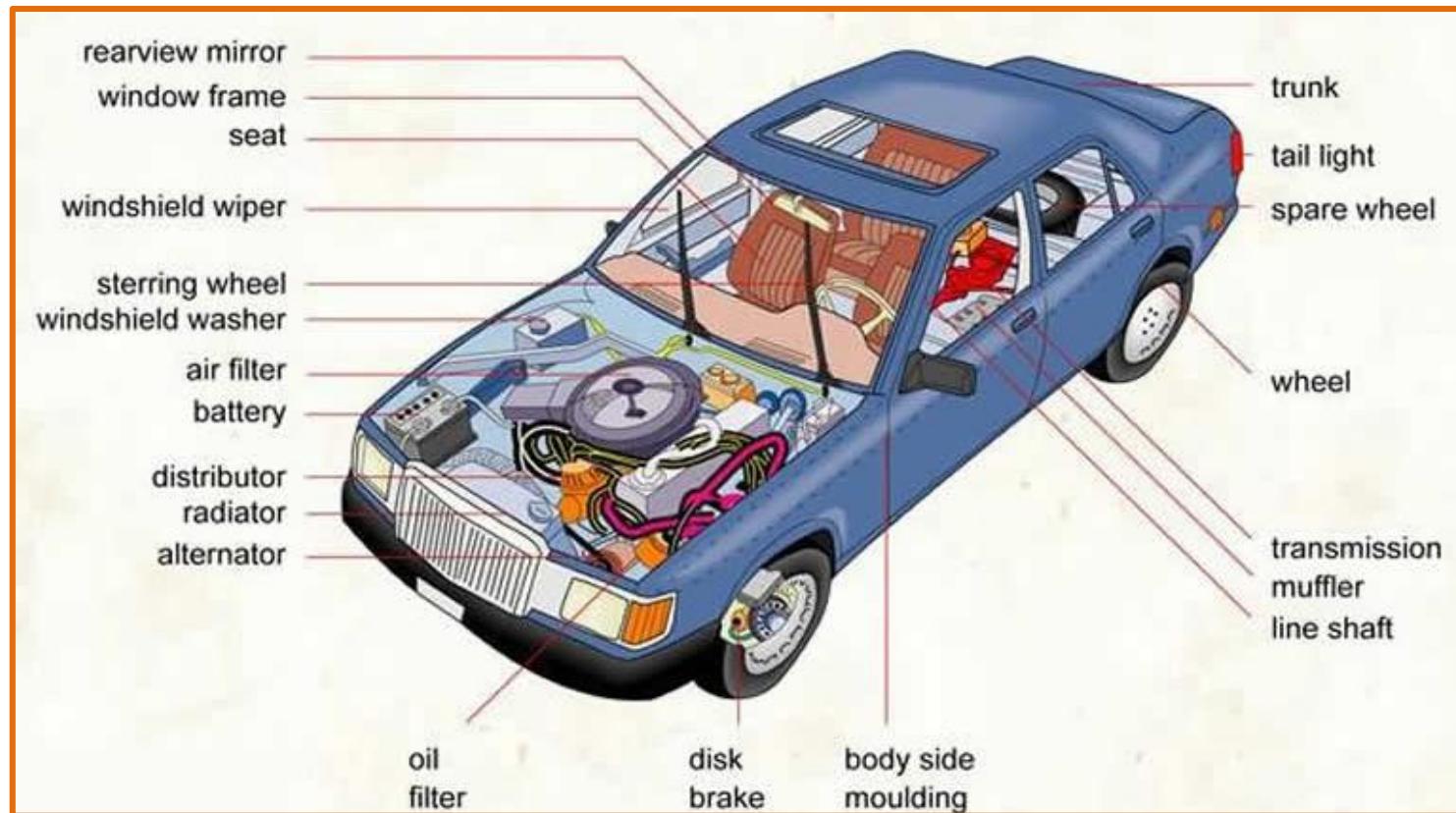
1. Break problem into logical part
2. Each part should have clear and simple interface.
3. Pick general abstraction (if possible) to make part easier to extend and reuse.



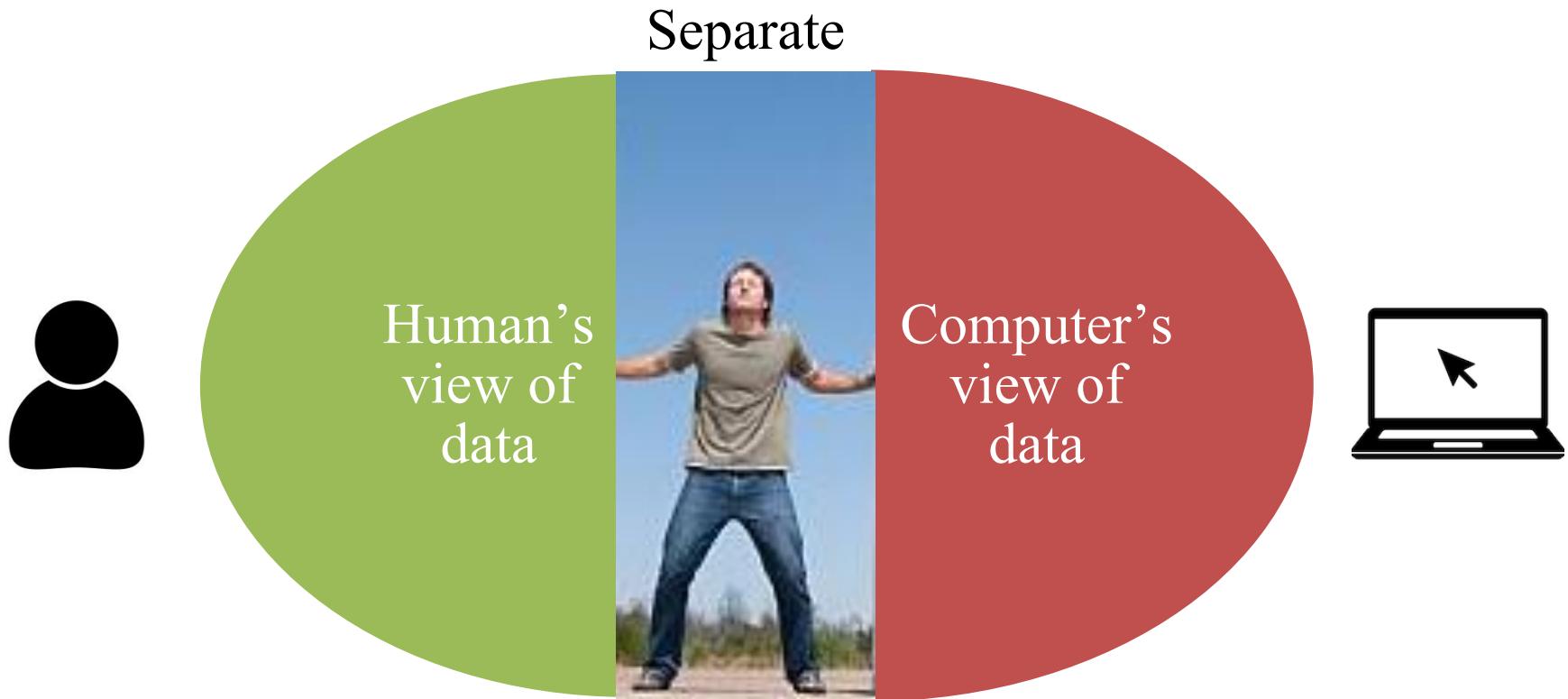
Vending
Machine

1. Change counting mechanism is separate from delivery mechanism.
2. Coin insertion, item selection and retrieval is only important for the users.
3. If accepting bill feature is desired, should not have to change the delivery mechanism to allow this feature.

You Try 2. In the car example try to list the *interface* items that the driver is exposed to, and parts that relates to *implementation* or *operation* part of the car (hidden info).



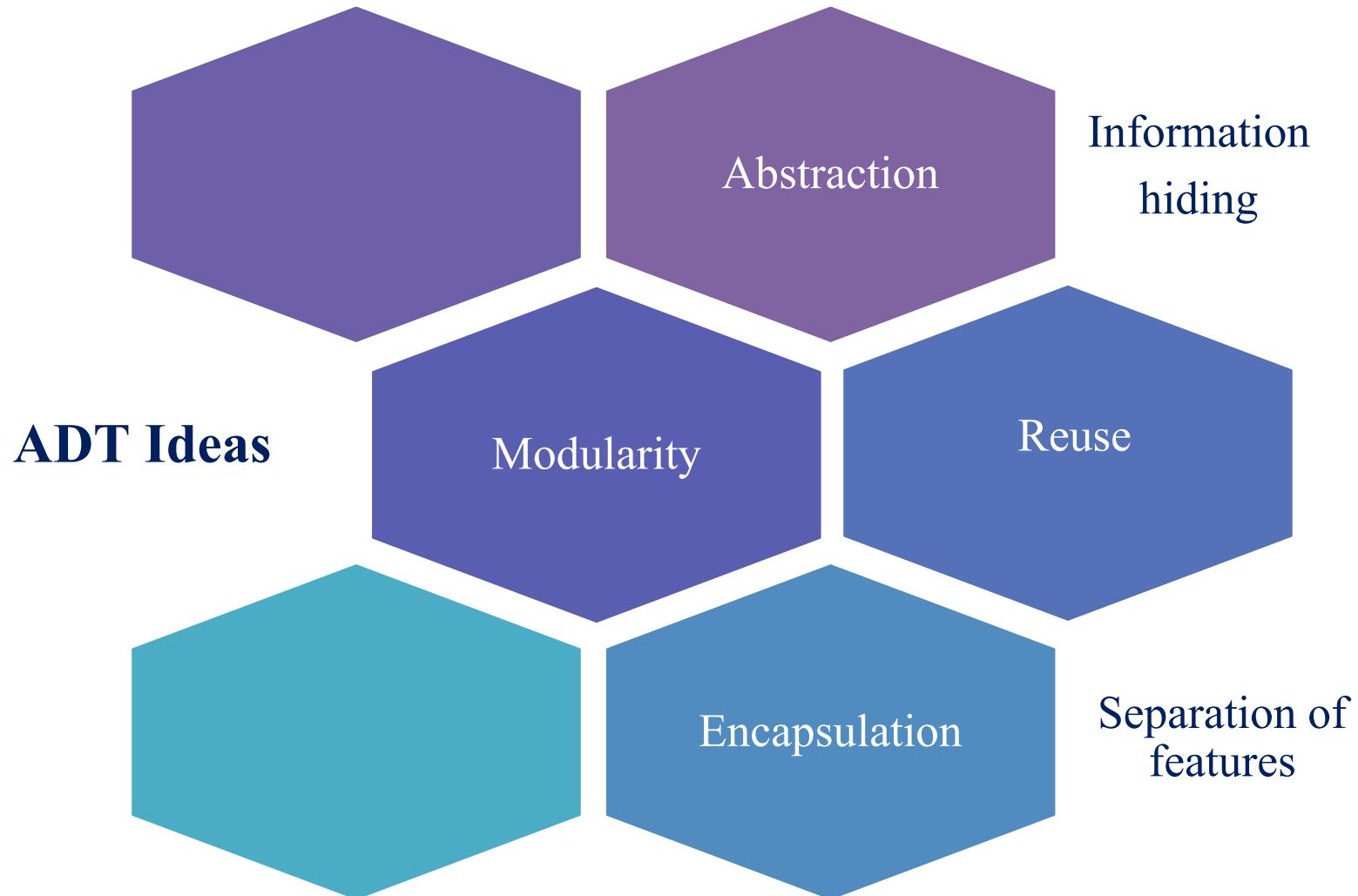
Meaning of Data Abstraction



Information in terms of units such as numbers, lists, etc.

A collection of bits that can be turned on or off.

We can use the ideas of abstraction to design programs using
“Abstract Data Types (ADT).”



Abstract Data Type (ADT)

- ADT is a collection of data items given a name, purpose, and a set of functions (*interface*) that operate on the data items.
- With the ADT, only the interface is exposed externally, and data organization is hidden.
- ADT can be described from different perspectives:



Example. Library

Application

Focus on **entities**.

e.g.

U Waterloo library
Or U of T library



<https://uwaterloo.ca/future-students/photospheres>

Logical

Focus on **what** questions.

What is library?
What services
(operations) can library
perform?

e.g.

- check out a book
- check in a book
- reserve a book
- pay a fine for an overdue book
- pay for a lost book

Implementation

Deal with **how** questions.

- How are books cataloged & organized on the shelf?
- How does the librarian process a book when it is checked in?

e.g. catalogued according to the Dewey decimal system , arranged in 4 levels of stacks, with 14 rows of shelves on each level.

Abstract Data Type (ADT)

- Some examples of ADTs are linked list, stack, queue, tree, and graphs.
- ADTs are often independent of the programming language.
- We will implement ADTs as *classes* in C++.

Next Lecture

We will focus on:

- a few examples of C++ implementation for a simple algorithm, a data structure and an ADT.
- dynamic memory allocation and its use.
- pointers and what they do, and how they work in code.

Readings/ Study Reference

Course Textbook: “*Data Structure and Algorithms in a Nutshell*” A. Wong et al.

Chapter 1. Introduction

Section 1.1. Data Structures

Section 1.2. Algorithm

Section 1.3. Abstract Data Type

Section 1.4. Challenge Questions

The End
of Lecture

Course: Data Structures and Algorithms

Instructor: Homeyra Pourmohammadali

**Examples of C++ Implementation of
A Data Structure and An ADT
& Challenge Questions**

Learning Outcomes

By the end of this lecture you will be able to:

1. see some examples of C++ implementation of data structure and abstract data type (ADT)
2. discuss the possible answers of challenge questions

Class and Object in OOP

In object-oriented programming,

- a **class** represents a **kind** of objects; it is a **template** for creating instances of specific objects; it is an **abstraction** of the things that are common to the kind of objects.
- An **object** represents an individual and specific item; it is an **instance** of the general class.

OOP (C++)

- Allows a data structure's members to be private.
- Methods (functions) that operate on these data items are included within the data structure.
- Users of the data structure use the public methods to achieve their goals, without accidentally affecting the private data items in any bad ways. This is called **encapsulation**.

Data Structure (DS) Example

Example 1. DS. Dealer Inventory

If you are a car dealer, how do you manage your inventory using C++?



...

A *data structure* (DS) is a group of *data elements* grouped together under one name. Each car acts as a data element, which is also known as *member* of DS, with different types and different lengths. DS can be declared in C++ using **struct**

Data items: [year], [type], [price], [model], [ID], ...

Methods/functions: print all, sort,

Example 1. Solution. DS. Dealer Inventory

```
#include <iostream> // include header file library
#include <string> // include the string library
using namespace std; //use names of objects and variables
//from the standard (std) library.
```

```
struct Car {
    int year;
    string brand;
    double price;
};
```

```
int main() {
```

Continue next page →

Example 1. Solution. DS. Dealer Inventory

```
Car carInventory[100];
```

```
//add car 1
```

```
carInventory[0].year = 2002;  
carInventory[0].brand = "Honda";  
carInventory[0].price = 1000.00;
```

```
//add car 2
```

```
carInventory[1].year = 1984;  
carInventory[1].brand = "Porsche";  
carInventory[1].price = 40000.00;
```

Continue next page →

Example 1. Solution. DS. Dealer Inventory

```
//print all  
  
cout<<"The first car is: " << carInventory[0].year << " " <<  
carInventory[0].brand << ", price: " << carInventory[0].price<< endl;  
  
cout<<"The second car is: " << carInventory[1].year << " " <<  
carInventory[1].brand << ", price: " << carInventory[1].price<< endl;  
  
return 0;  
}
```

End

Abstract Data Type (ADT)

Example

// ADT Purpose: Models a non-negative value in Canadian (CDN) dollars

```
class CurrencyAmountCDN {
```

// Private attributes

```
int dollars, cents;
```

// Helper Functions

```
void increment(int new_dollars, int new_cents) {
    int new_cents_value = (dollars + new_dollars) * 100
                        + (cents + new_cents);
    if (new_cents_value <= 0) {
        dollars = 0, cents = 0;
    } else {
        dollars = new_cents_value / 100;
        cents = new_cents_value % 100;
    }
}
```

CurrencyAmountCDN ADT
Implementation in C++

Continue next page →

public:

// Constructors

CurrencyAmountCDN() : dollars(0), cents(0) {} // default constructs

CurrencyAmountCDN(double value) {

if (value <= 0) {

 dollars = 0, cents = 0;

} else {

 dollars = value; // automatically casts value to integer

 cents = (value - dollars + 0.005) * 100;

}

}

CurrencyAmountCDN ADT
Implementation in C++

Continue next page →

// Service Functions

```
void add(CurrencyAmountCDN add_amount) {  
    increment(add_amount.dollars, add_amount.cents);  
}  
  
void sub(CurrencyAmountCDN sub_amount) {  
    increment(-sub_amount.dollars, -sub_amount.cents);  
}  
}
```

End

Challenge Questions

Challenge Questions

1. Label each either as (T) true or (F) false.

A data structure in the object-oriented context:

- (a) can be used to organize a collection of objects (T)
- (b) must be chosen carefully for specific case (T)
- (c) can be useful even if it does not have any data (depends)
- (d) must have at least one public method (T)

Challenge Questions

2. Is this statement (T) true or (F) false or it depends?
An algorithm, which is defined as a step-by-step procedure for solving a problem, can be used independently of the underlying data structure. (depends)

3. You can create as many instances of an abstract data type as you need (T)

Challenge Questions

4. What are the benefits of abstract data type (ADT)?
 - No need for user to understand the inner working operation
 - User only know methods to interact with data type
 - Provides more flexibility while programming
 - To users, ADT would be the same even if the implementation behind it changed as long as the method signatures stays the same

Readings/ Study Reference

Course Textbook: “*Data Structure and Algorithms in a Nutshell*” A. Wong et al.

Chapter 1. Introduction

Section 1.1. Data Structures

Section 1.2. Algorithm

Section 1.3. Abstract Data Type

Section 1.4. Challenge Questions

The End
of Lecture

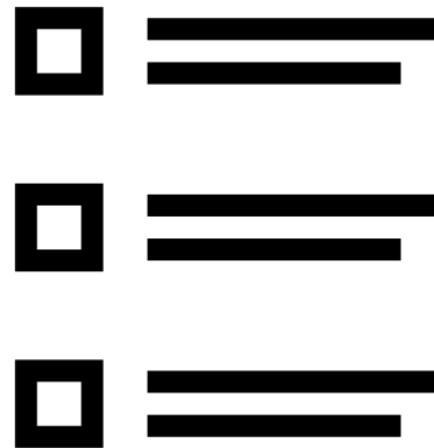
Course: Data Structures and Algorithms

Instructor: Homeyra Pourmohammadali

Dynamic Memory Allocation & Pointers

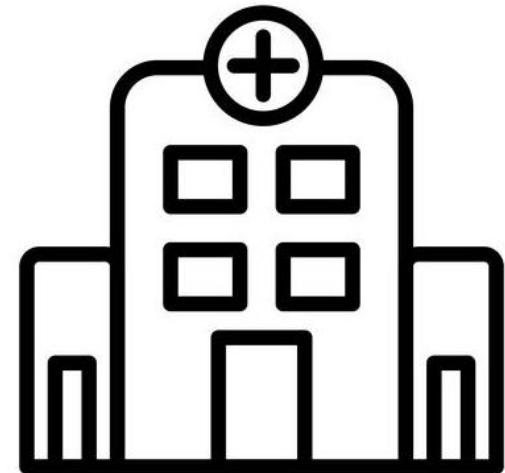
Motivation

- How many websites should **Google index** to provide optimal search results?
- How the underlying algorithms adapt to new data as the size of database grows?



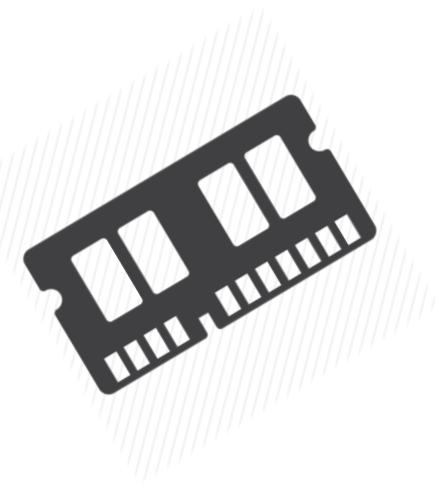
Motivation

- How many health records should a **hospital database** hold?
- How well applications can work as the size of database grows?



Motivation

- How computers manage their memory usage at runtime to allow for development of programs that adapt to the problem as it grows?
- How to write programs for applications in such a way that can scale to large datasets?

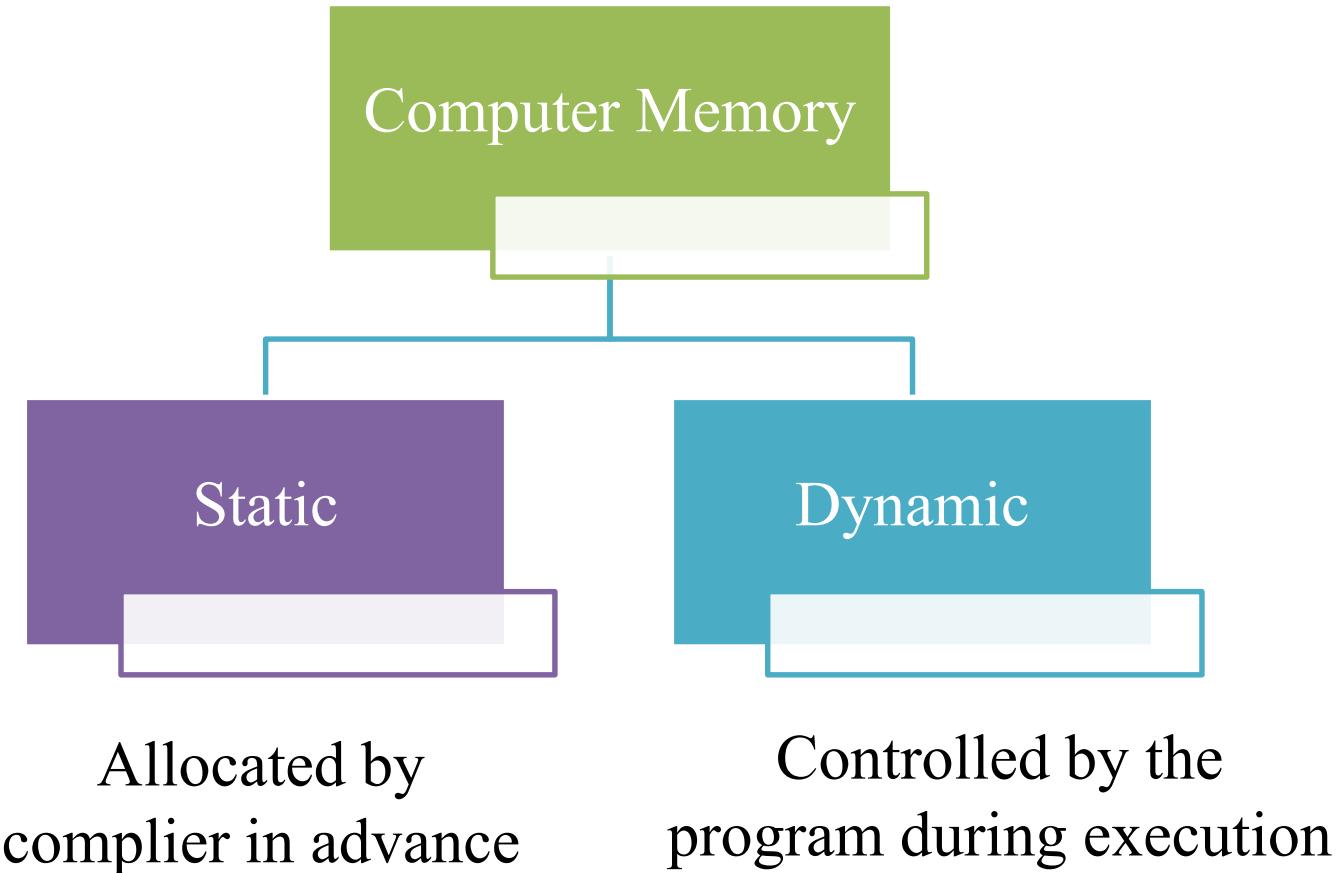


Learning Outcomes

By the end of this lecture you will be able to find out:

1. why and how dynamic memory is allocated.
2. how pointers are used for dynamic memory allocation and they are declared in C++.

Introduction



When Dynamic Memory Allocation is Needed?

DMA

When it is not known how much memory is required for the program beforehand.

When efficient use of memory space is intended.

When data structures with no upper bound of memory space is desired.

When the concept of structures and linked list in programming are used.

Dynamic Memory Allocation

- Dynamic memory allocation is a mechanism by which storage/memory can be allocated to variables **during the runtime**.
- **Example:** when the memory needed depends on user input. On these cases, programs need to dynamically allocate memory.
- When programs work with large data they do it in Random Access Memory (RAM) of computer.
- Programs access memory using an address.

Pointers

- A pointer is a data value that addresses a storage unit in memory (like the street address of a house).



<https://www.wideninghorizons.com/Numerology-for-Your-Address-Is-Your-Home-Happy.aspx>

- It is a variable that stores the address of another variable. Pointers "point to" the variable whose address they store.

Pointers Declaration

- A pointer can be used to address any type of data (e.g can be *int* or *long*, etc.)
- In C++, a pointer is declared with an asterisk ***** preceding its variable name.

For instance: $super = *star$

variable *super* equal to value pointed to by *star*

* is called “*dereference operator*”

Example. Pointer Declaration in C++

```
int A=7;  
int B=20;
```

```
int* pointerToA= &A //an ampersand sign (&), known as address-of operator.
```

```
int* pointerToB= &B // assigns the address of variable B to int* pointerToB
```

```
cout<<"Pointer A points to memory address " + pointerToA+  
      ", containing an int with value " + *pointerToA+ ":";
```

```
cout<<"Pointer B points to memory address " + pointerToB+  
      ", containing an int with value " + *pointerToB+ ":";
```

Outputs after running program, print these:

Pointer A points to memory address 0x660170, containing an int with value 7.
Pointer B points to memory address 0x660180, containing an int with value 20.

Next Lecture

We focus on:

- New dynamic memory allocation
- Memory leak and how it can be prevented
- Nodes and linked list

Readings/ Study Reference

Course Textbook: “*Data Structure and Algorithms in a Nutshell*” A. Wong et al.

Chapter 2. Linked Data Representation

Section 2.1. Dynamic Memory Allocation

Section 2.2. Pointers (up to **Subsection 2.2.1**)

The End of Lecture

Course: Data Structures and Algorithms

Instructor: Homeyra Pourmohammadali

**Pointers:
Memory Allocation & Garbage Collection**

Learning Outcomes

By the end of this lecture you will be able to:

1. extend your vision about memory structure and memory allocation
2. know the main operators for allocating and de-allocating memory at runtime
3. find about the memory leakage and dangling pointer

Memory Allocation

At the start of a program,

- the operating system (e.g., Windows) assigns the program a memory pool (free space) to use. This memory pool is large, but not infinite.

At the end of the program,

- this memory pool will be recollected/de-allocated by the operating system for other programs to use.

Static vs. Dynamic Memory Allocation

- During the program's lifetime, there are two ways to allocate memory space to each variable used in the program.

Static (Automatic) Memory Allocation



- Variables defined **without** using the `new` operator
- Has lifetime within a scope (e.g., of a function)
- At the end of the scope, the variable is **automatically** destroyed, and memory is **automatically** deallocated for other things in the program to use.

Dynamic (Manual) Memory Allocation



- Variables defined using the `new` operator
- Has lifetime **with the entire program**, or until user uses `delete` to **manually** de-allocate the memory space.

Program

- A passive sequence of instructions that can be interpreted and executed by a computer
- **Example:** Your C/C++ code compiled and stored onto secondary memory such as hard drive

Process

- An active representation of a program; It is a program interpreted and loaded by the system software
- **Example:** OS loads your C/C++ program, and then assigns it memory & execution context to run

Virtual Address Space

- A set of addresses available to each process
- Virtual addresses are mapped to main memory by OS (e.g., mapped to RAM)

Virtual Memory

Virtual addresses and their data represent Virtual Memory (VM).

The memory is called virtual as it is used to give an illusion of larger memory allocation.

Each process gets a virtual address space to hold its frequently used code and data.

Virtual Memory

- ❑ Address space can be defined by number of bits used to specify each address
 - For example, 32-bit or 64-bit address space
 - 0x155FFFFF is an address in 32-bit space
 - 0x155FFFFF can also represent an address in 64-bit memory space with extra 0s in front
 - In 32-bit address space, there are 32 bits available to store each address, so there are 2^{32} possible locations
 - In 64-bit address space, there are 64 bits available to store each address, so there are 2^{64} possible locations

Virtual Memory

□ Example: 32-bit Virtual Memory

- If each 32-bit memory location were to store a single byte (8 bits), then there are 4 gigabytes of available virtual memory space (2^{32} bytes)
- Each user process gets its own user space (e.g., 2 GB) while the system software runs in its own kernel space (e.g., another 2GB)

□ Within each user space, there are blocks for:

- Instructions (CODE),
- Global and static variables,
- Locally-allocated variables (STACK), and
- Dynamically-allocated variables (HEAP)

(see the visualization on the next slide)

Architecture of Memory

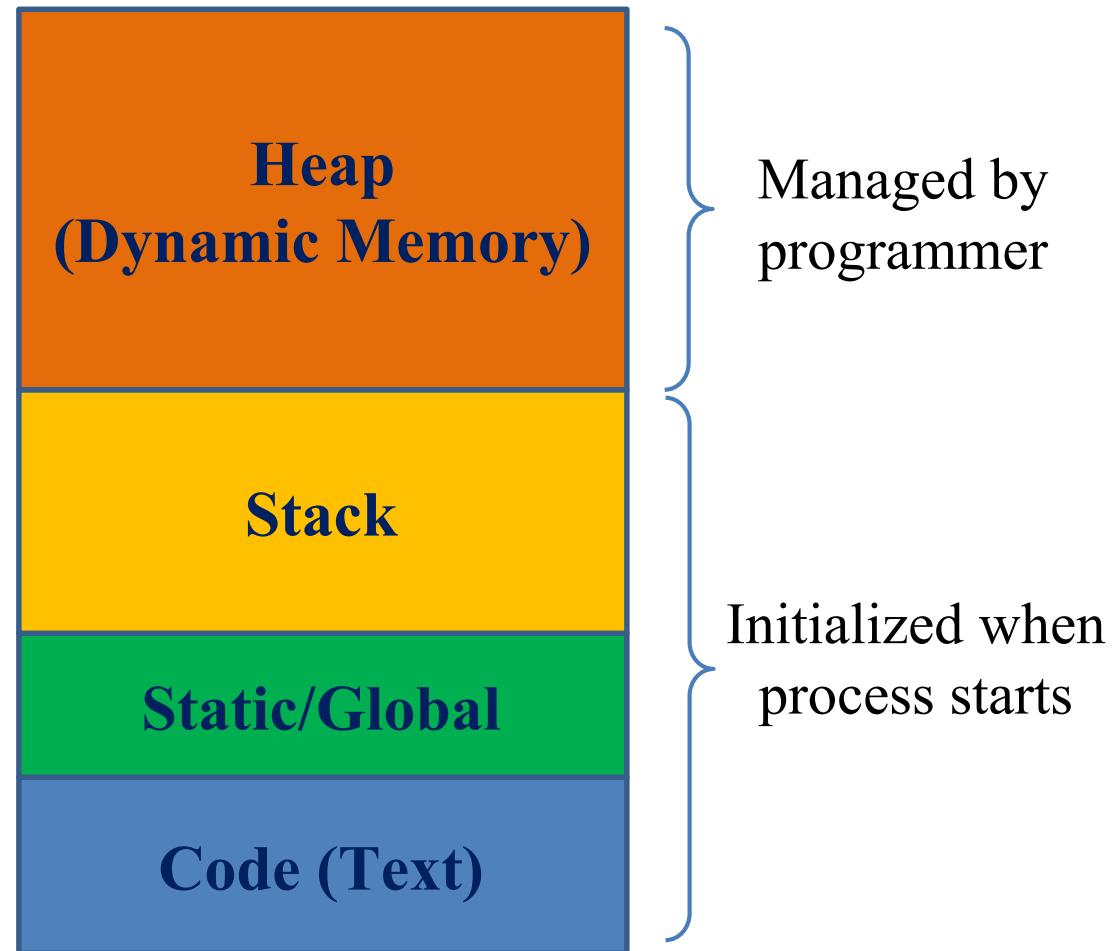
How does OS manages memory and how it is accessible to programmers?

Free store →

Function calls (to
call variables) →

For global variables →

For instructions →



Local and Dynamic Variables

Local Variables

- Declared within function definition (e.g., int i = 3, double j = 4.0)
- Stored on the Memory Stack
- Created when function is called
- Destroyed when function call completes
- Typically faster to use than dynamic variables

Dynamic Variables

- Created with the `new` operator (e.g., `new int(2)`)
- User responsible for calling the `delete` operator
- Created and destroyed while program runs
- Typically slower to use than local variables

Dynamic Memory and a Pointer

Dynamic Memory or the Memory Heap

- Reserved for dynamic variables
- Significantly larger in space availability than the Memory Stack

Pointer

- Memory address of a variable (e.g., 0x55542)
- Pointer represents a memory address and that address is an integer (e.g., represented as hex)
- **Stack Pointer:** Memory address is on the Memory Stack
- **Heap Pointer:** Memory address is on the Memory Heap
- Pointers are typed and stored in a (typically local) variable; does not store actual data value; it stores a pointer to *int* or *double* or another data type

Pointer Example and Address Operator

```
int *ip = NULL;
```

- ip is declared as a pointer to int variable; its value set to NULL/0
- ip is capable of holding a pointer to (address of) a variable of type int

```
int iv1 = 5;
```

```
ip = &iv1;
```

- Sets pointer ip to point to integer iv1; ip is now a stack pointer

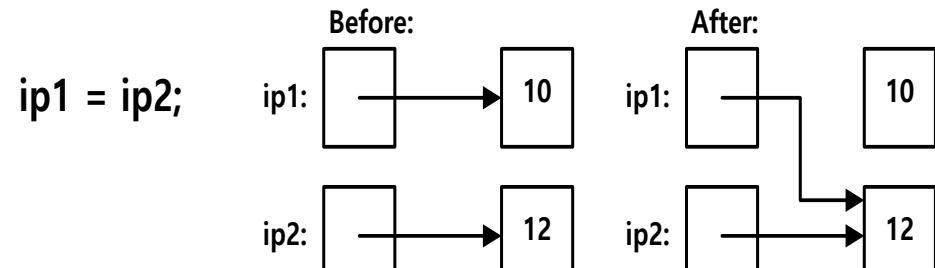
Address Operator (&):

- Used to determine the address of a variable
- Example: ip = &iv1; ⇒ “ip equals address of iv1” or “ip points to iv1”

Dereference and Address Operator

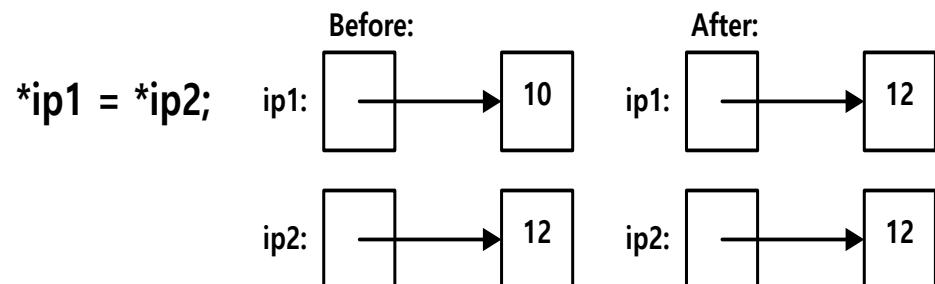
Dereference Operator (*): Used to follow the pointer to its target
Example:

- **cout << iv1;** ⇒ refer to **iv1** directly and output **5**
- **cout << *ip;** ⇒ refer to **iv1** indirectly and output **5**



Pointer variables can
be copied by address
or by value

- Examples:



The new and delete Operators

The new Operator

- Dynamically allocates variables for pointers
- Operator `new` creates variables
- No identifiers available to refer to each variable; instead, need to use pointers

The delete Operator

- Deallocates dynamic memory and returns it to free memory

The new and delete Operator Examples

Example of the new operator:

- `int* ip3 = new int(2);`
 - Creates new anonymous variable with **2** as its value, and assigns **ip3** to point to it
 - Can access the variable with `*ip3`, and use it just like any other variable (e.g., `cout << *ip3;`)
 - Eventually responsible for freeing up memory

Example of delete operator:

- `delete ip3;`
 - Deallocates dynamic memory and returns it to free memory

Memory Leak

- It occurs when a piece (or pieces) of memory that was previously allocated by a programmer (using **new**) is not properly de-allocated (using **delete**). Even though that memory is no longer in use by the program, it is still “reserved.”
- Without **delete**, no **new** operator can allocate the same address allocated by a previous **new**.
- If a program has a lot of memory that hasn’t been de-allocated, it could slow down the performance.
- If there’s no memory left in the program, it could cause the program to crash.

Dangling Pointers

- Dangling pointer is a non-null pointer that points to unallocated memory.
- Dereferencing a dangling pointer may cause the program to crash.
- For instance `delete ip3;` frees up dynamic memory
- But `ip3` still points to that location.
- This is referred to as a “dangling pointer”; no way to check later if it is a valid pointer.
- If `ip3` is dereferenced using `*ip3`, this will lead to unpredictable results.

Avoid Dangling Pointers

- Must avoid dangling pointers if pointers can still be accessed outside of their scope
 - For example, a member pointer may still be accessible as a class member via existing object reference
 - In that case, assign the pointer to **NULL** after deleting it to explicitly denote that it has been freed
 - `delete ip3;`
`ip3 = NULL;`
...
- (if someone tries to check `ip3` validity later, `if(ip3)` will be false)

C++ Implementation Example

You can use online C/C++ compiler if needed to run this code (e.g., cpp.sh)

```
int i = 5;  
int* i_ptr = &i;      // stack pointer example  
//delete i_ptr;      // deleting stack pointer is not allowed and also not needed
```

```
cout << "Initializing i_ptr2 \n";  
int* i_ptr2 = new int(10);    // heap pointer example  
cout << "*i_ptr2 value is:" << *i_ptr2 << " \n";  
cout << "i_ptr2 value is:" << i_ptr2 << " \n\n";
```

```
cout << "Deleting i_ptr2 \n";  
delete i_ptr2;            // deleting heap pointer  
cout << "*i_ptr2 value is:" << *i_ptr2 << " \n";  
cout << "i_ptr2 value is:" << i_ptr2 << " \n\n";
```

```
cout << "Setting i_ptr2 to NULL \n";  
i_ptr2 = NULL;           // ensuring that there is no dangling pointer  
cout << "i_ptr2 value is:" << i_ptr2 << " \n";
```

Pointers Applied:
Detailed Code Example

Next Lecture

We focus on:

- Nodes and how they can be inserted and deleted
- Linked Lists and how they are structured

Readings/ Study Reference

Course Textbook: “*Data Structure and Algorithms in a Nutshell*” A. Wong et al.

Chapter 2. Linked Data Representation

Section 2.2. Pointers

Subsection 2.2.1 Allocating Memory

Subsection 2.2.2 Garbage Collection

The End of Lecture

Course: Data Structures and Algorithms

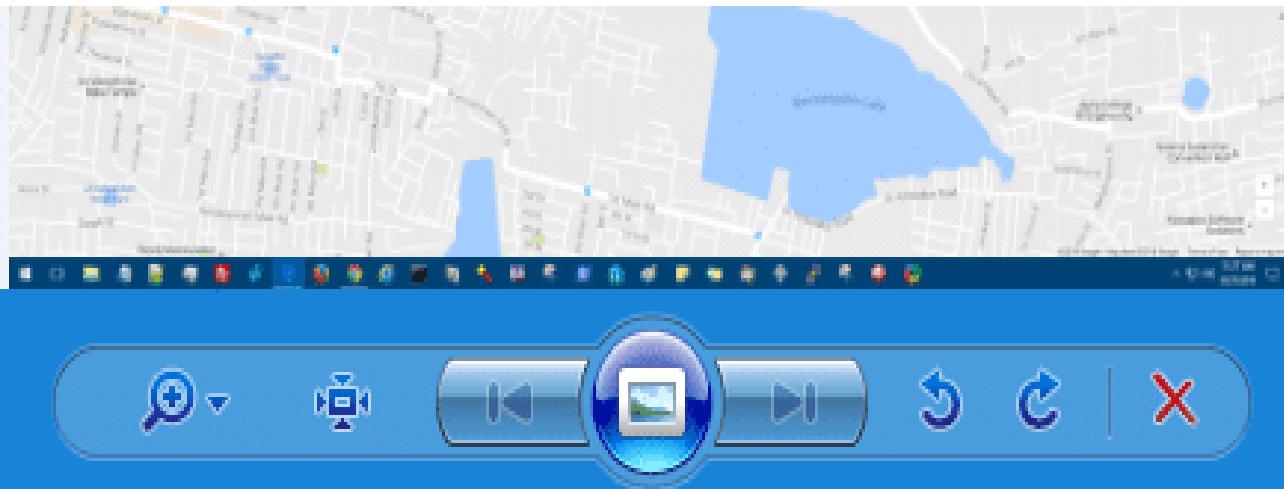
Instructor: Homeyra Pourmohammadali

Nodes and Linked Lists

Motivation

How previous and next images in the image viewer are linked?

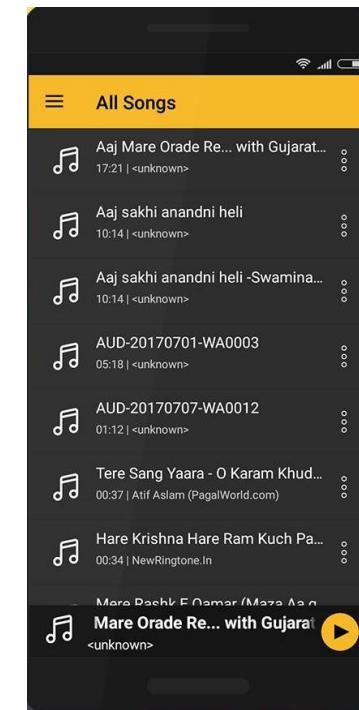
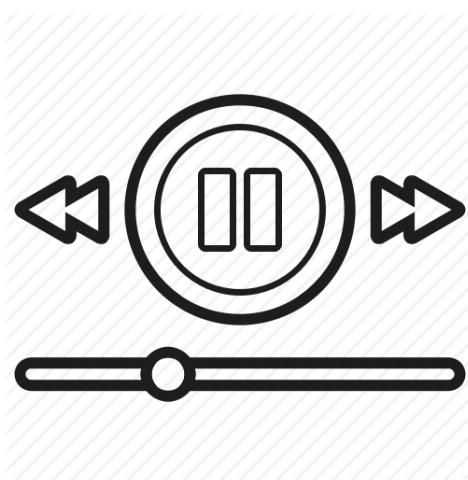
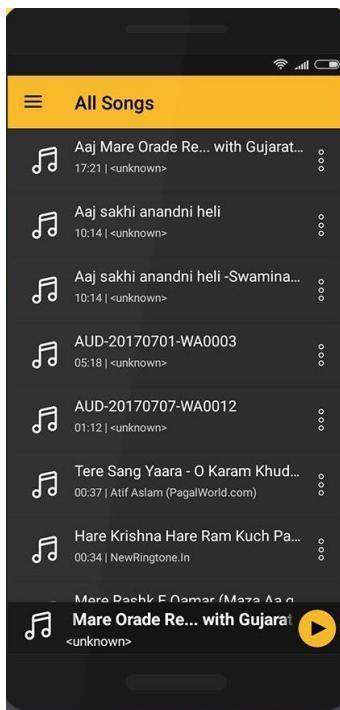
You can access images by next and previous button.



<https://www.winhelponline.com/blog/windows-photo-viewer-previous-next-buttons-disabled-arrow-keys-not-work-fix/>

Motivation

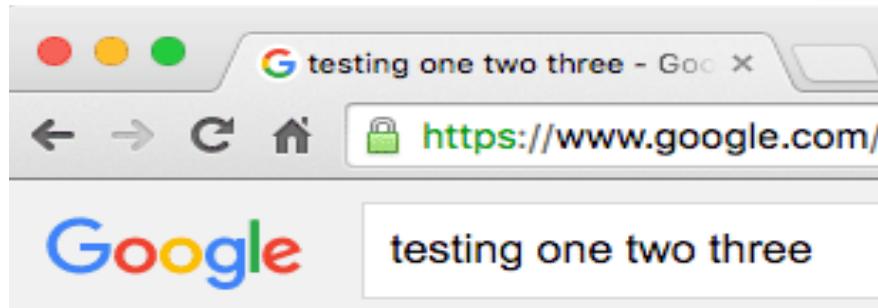
How songs in music player are linked to previous and next song?



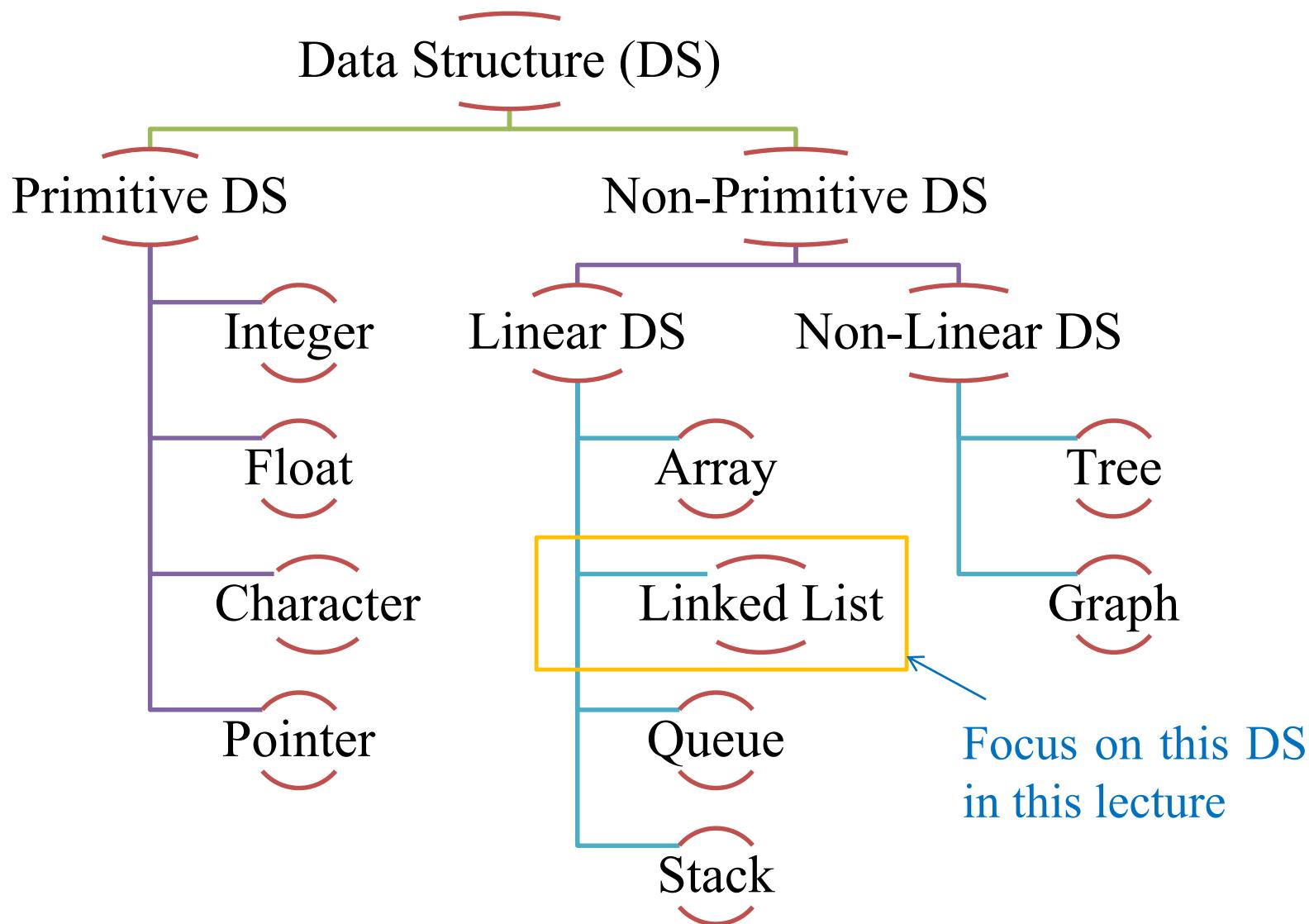
Motivation

How can you access previous and next url searched in web browser?

How back and next button are linked to each other?



Recall: Types of Data Structures



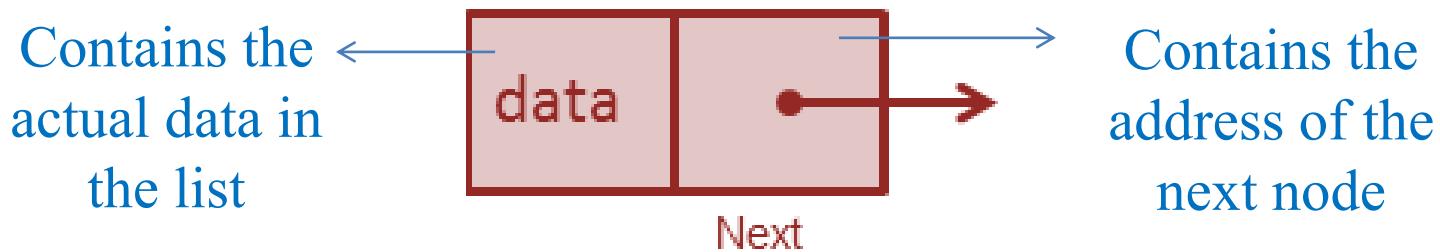
Learning Outcomes

By the end of this lecture you will be able to find out:

1. how pointers are used to store and manipulate data in a program and what is a node.
2. what is the linked list and its main operations.
3. how the nodes and linked lists are implemented in C++.
4. what are the ways of structuring linked lists.

Node

- A data structure that contains both data items and pointers to other nodes in the sequence.

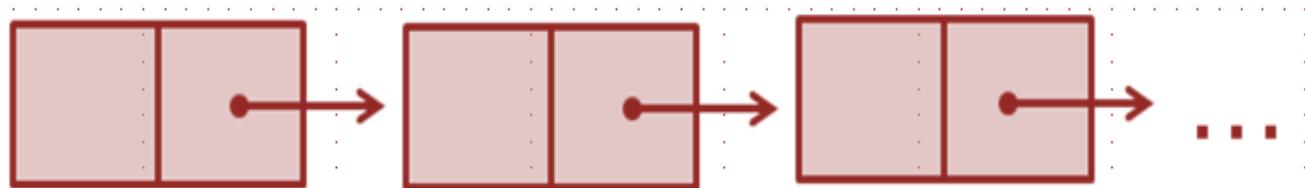


- Linked lists consists of nodes. We need to declare a structure which defines a single node. A **node** in C++ code:

```
struct Node{  
    DataType data;  
    Node* next;  
};
```

Linked Data Structure

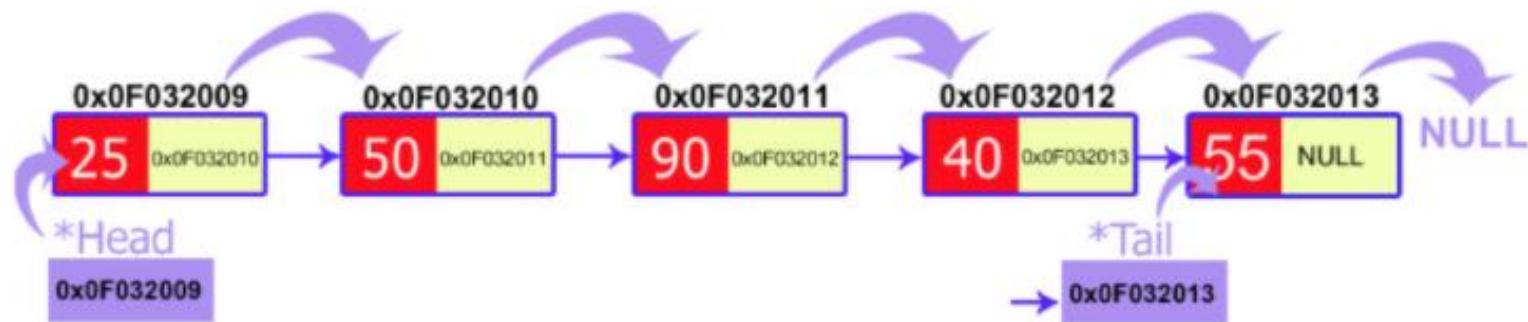
- Pointers can be used to address data structures and objects
- Can create objects that store relevant information, and then chain them together into an appropriate structure
- **Linked List:** An ordered container for sequence of nodes, where a pointer inside each node points to the next node in the list.



Visualization of a singly linked list

Linked List

Singly Linked List:

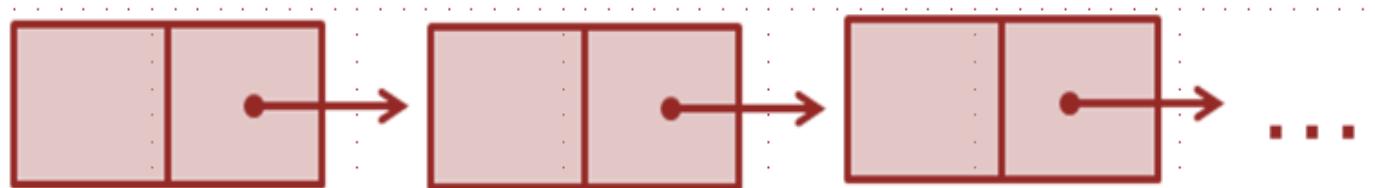


https://www.codementor.io/@codementorteam/a-comprehensive-guide-to-implementation-of-singly-linked-list-using-c_plus_plus-ondlm5azr

- Linked List contains a link element called **first (head)**.
- Each link is linked with its next link using its next link.
- Last link carries a link as **null** to mark the end (**tail**) of the list.
- A node's **successor** is the next node in the sequence.
- A node's **predecessor** is the previous node in the sequence.
- A list's **length** is the number of elements in the list.

You Try 1. Circle the correct answer.

1. The first node has nor successor / predecessor.
2. The last node has no successor/predecessor.
3. A list can/ cannot be empty (contain no element).



```

#include <iostream>
using namespace std;
struct node
{
    int data;
    node *next;
};

class linked_list
{
private:
    node *head, *tail;
public:
    linked_list()
    {
        head = NULL; // The constructor of the linked list is making both 'head' and
        tail = NULL; // 'tail' NULL because we have not yet added any element to
    } //our linked list and thus both are NULL

};

int main()
{
    linked_list a;
    return 0;
}

```

//Create a node (structure)

//Create a linked list

//Made two nodes – head and tail.

Linked List Manipulation

- By adding or removing links from the data sequence, the list structure can accommodate the needs of an algorithm.
- Nodes can be added to the chain to expand storage space of the list.
- Nodes can also be removed when not needed.
- Contiguous memory does not have to be allocated for the list.
- Contents of the list can be searched and printed.

Insert A New Node

A new node can be inserted:

in an empty list

at the front of the linked list

after a given node

at the end of the linked list

Insert A New Node

C++ Declaration (insertion in an empty list):

```
LinkedList list;  
Node* newNode = new Node();  
List.head = newNode;
```

```
#include <iostream>
using namespace std;
struct node
{
    int data;
    node *next;
};
```

```
class linked_list
{
private:
    node *head,*tail;
public:
    linked_list()
    {
        head = NULL;
        tail = NULL;
    }
```

```
void add_node(int n)
{
    node *tmp = new node;
    tmp->data = n;
    tmp->next = NULL;

    if(head == NULL)
    {
        head = tmp;
        tail = tmp;
    }
    else
    {
        tail->next = tmp;
        tail = tail->next;
    }
}
```

```
int main()
{
    linked_list a;
    a.add_node(1);
    a.add_node(2);
    return 0;
}
```

Created a function
of adding a node
to the linked list.

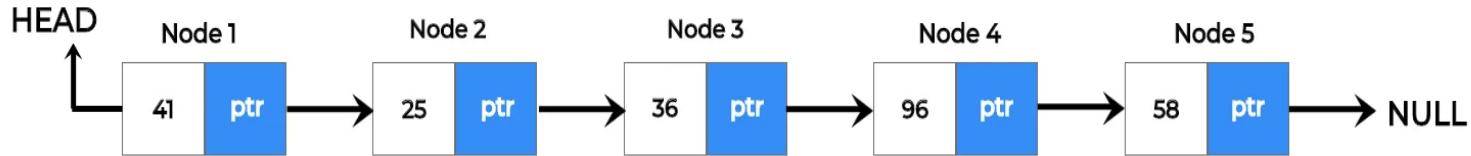
```
void add_node(int n)
{
    node *tmp = new node;
    tmp->data = n;
    tmp->next = NULL;

    if(head == NULL)
    {
        head = tmp;
        tail = tmp;
    }
    else
    {
        tail->next = tmp;
        tail = tail->next;
    }
}
```

Search an Element In the List

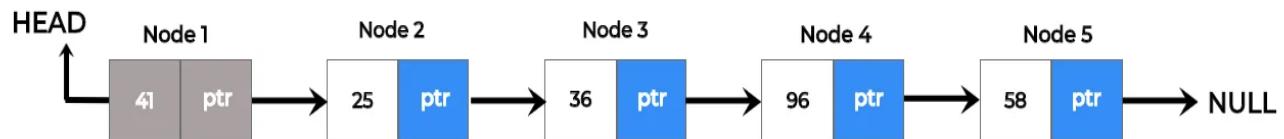
- Need to **walk down the list**, starting from the head
- Look at each node's data and use its pointer to the next node to find the next node in the chain (the process can be repeated).
- Once the node with NULL pointer is found, the function terminates.

Example. Search an element in linked list

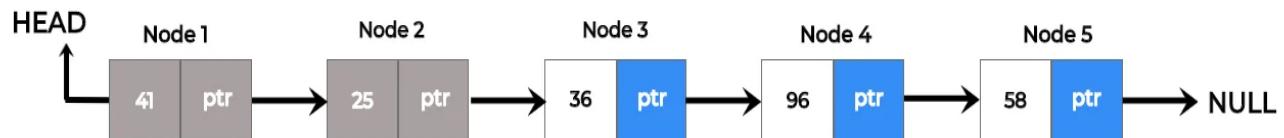


Element to be searched **36**

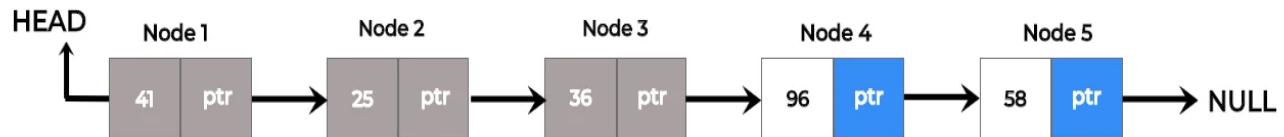
Is node 1->data == 36



Is node 2 ->data == 36



Is node 3 ->data == 36



Element to be searched is present at location **3**

You Try 2.

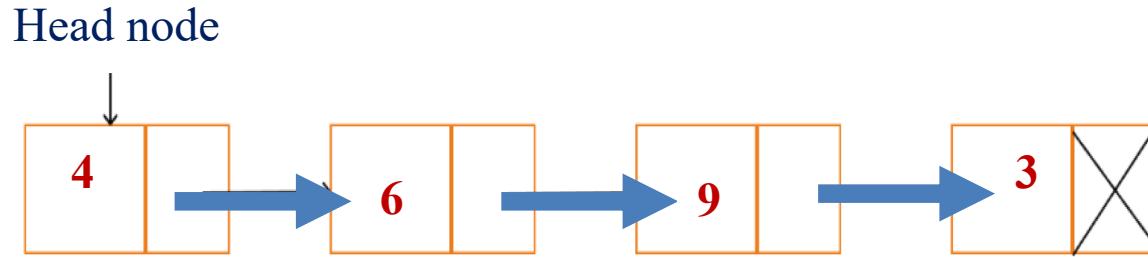
What would be the process if you just want to check the list contains a value or not?

Delete the Last Node From the List

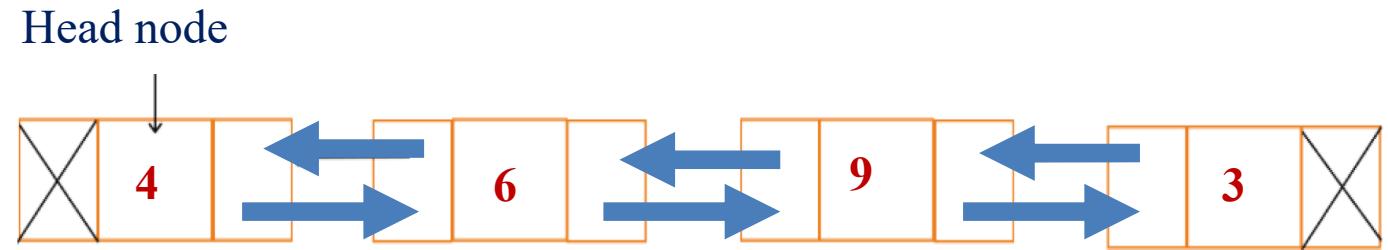
- Need to iterate through the list to find the end.
- Check if the item that you are iterating through is at the end of the list.
- If it is, delete it.

Types of Linked Lists

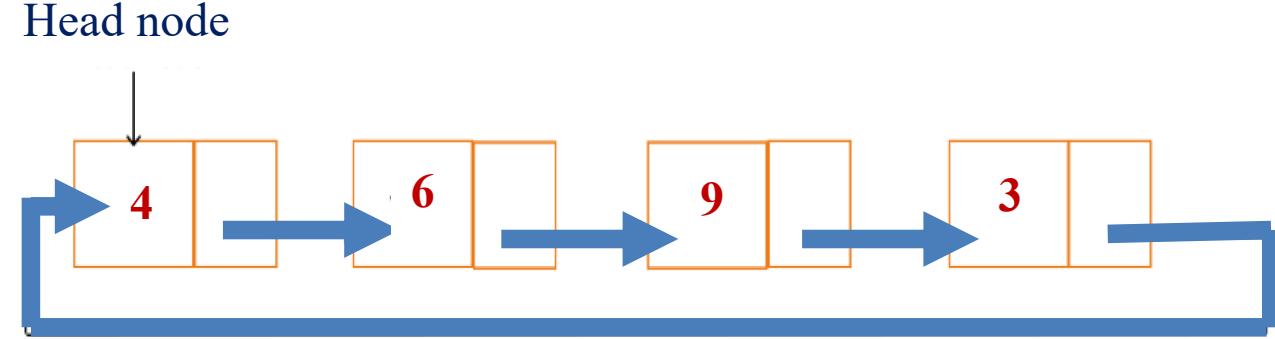
**Singly
Linked List**



**Doubly
Linked List**



**Circularly
Linked List**



Types of Linked Lists

Singly

- Starts with a pointer to the first node
- Ends with a null pointer
- Only traversed in one direction

Doubly

- Two *start pointers*: first element and last element
- Allows traversals both forwards and backwards
- Each node has a forward pointer and backward pointer

Circularly

- Pointer in the last node points back to the first node

Next Lecture

We focus on:

- Sequential list implementation
- Linked List implementation
- Comparing implementations

Readings/ Study Reference

Course Textbook: “*Data Structure and Algorithms in a Nutshell*” A. Wong et al.

Chapter 2. Linked Data Representation

Section 2.3. Nodes and Linked Lists

Section 2.4. Using Linked Data Representation in Practice

The End of Lecture

Course: Data Structures and Algorithms

Instructor: Homeyra Pourmohammadali

Sequential List and Linked List Implementation

Motivation

Sometimes we need to store a list of elements, add some elements to the list or remove them from the list (such as list of students enrolled in a class)



<https://www.theguardian.com/education/2020/feb>

Motivation

- Lists are one of the most useful and simple data structures.
- But how the List Abstract Data Type (ADT) can be created?
- How List ADT allow us to manipulate data efficiently?
- How List ADT can be implemented ?
- What kind of operations it can support?

Learning Outcomes

By the end of this lecture you will be able to:

1. define List ADT
2. distinguish the types of implementations of List ADT:
Sequential List versus Linked List
3. understand the main operations of these Lists and
their C++ implementations
4. compare these two implementations

Recall: Abstract Data Type (ADT)

- ADT is a collection of data items given a name, purpose, and a set of functions (*interface*) that operate on the data items.
- With the ADT, only the interface is exposed externally, and data organization is hidden.
- In C++, an ADT is usually implemented using class. A collection of values is associated to private data members and a set of basic operations is associated to public member functions.

List ADT Introduction

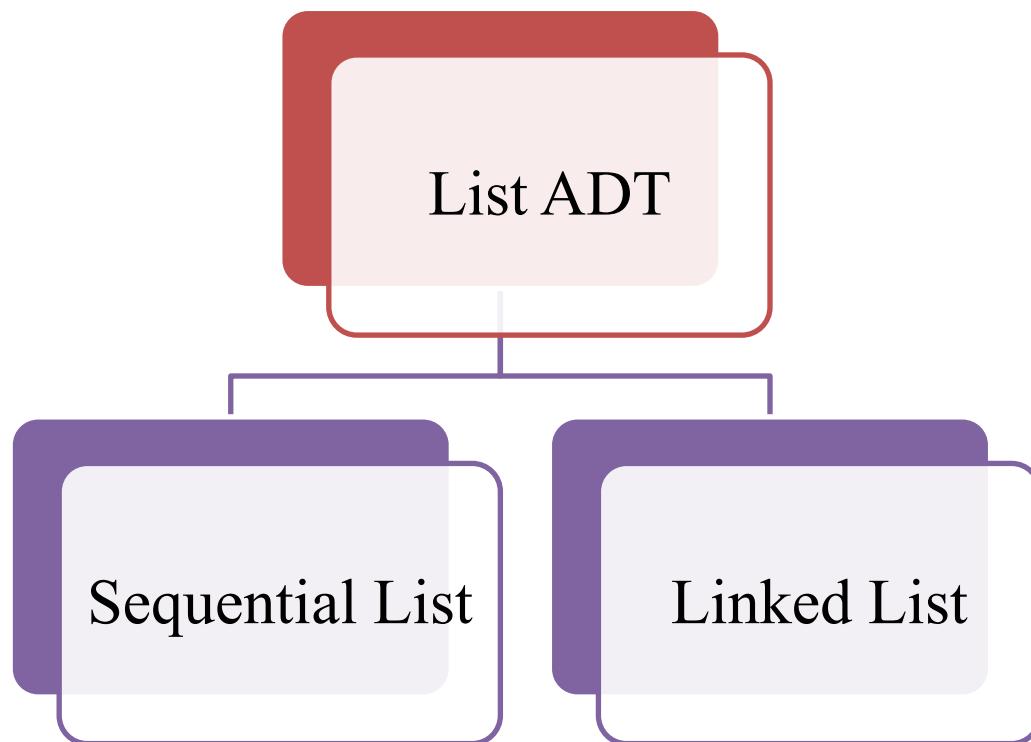
List ADT: A container structure for storing a connected sequence of data items

Selected List ADT operations:

- **Insert:** Inserts a value into the list at a specified position
- **Delete:** Removes data from the list at a given position
- **Select:** Returns the value stored at a given position
- **Replace:** Replaces the value stored at a given position
- **Size:** Returns the number of elements in the list

List ADT Introduction

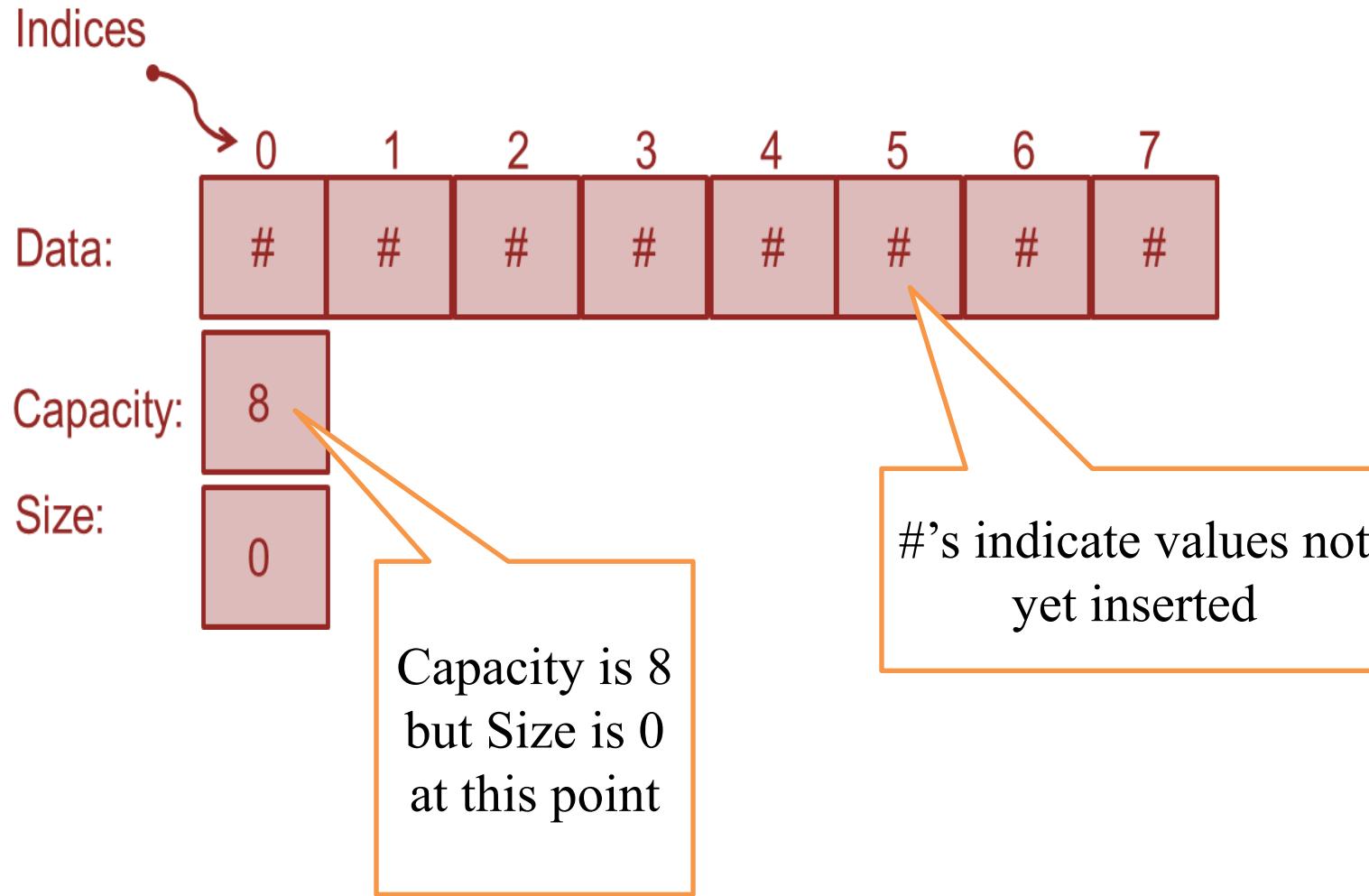
- We look at two different implementations of the List ADT



Sequential List Implementation

- Allocate a contiguous block of memory
- In C++ STL, a version of sequential list is provided as the `<vector>` library
- The implementation of List ADT needs three variables:
 - **data** – a contiguous memory location
 - **capacity** – the maximum number of elements that **data** can hold
 - **size** – the number of inserted elements in the list

Sequential List: Initialization



Sequential List: Insert

void insert(DataType value, int position)

- Inserts **value** into the list at the given **position** and increases **size** by 1
- After inserting the **value**, **data** at **position** and after is shifted towards the list end by one
- The list remains contiguous after each insertion
- If the **size** reaches **capacity**, the list **capacity** may be increased to accommodate new elements
- In the **<vector>** library, the capacity is increased on **push_back()** if out of space

Example: Sequential List Implementation. Insert

Initialize a list with capacity of 8 variables:

insert('c', 0) =>

0	1	2	3	4	5	6	7
c	#	#	#	#	#	#	#

size is 1

insert('b', 0) =>

0	1	2	3	4	5	6	7
b	c	#	#	#	#	#	#

size is 2

insert('a', 0) =>

0	1	2	3	4	5	6	7
a	b	c	#	#	#	#	#

size is 3

insert('k', 1) =>

0	1	2	3	4	5	6	7
a	k	b	c	#	#	#	#

size is 4

insert('z', 0) =>

0	1	2	3	4	5	6	7
z	a	k	b	c	#	#	#

size is 5

insert('y', 5) =>

0	1	2	3	4	5	6	7
z	a	k	b	c	y	#	#

size is 6

insert('m',7) [rejected; why?]

0	1	2	3	4	5	6	7
z	a	k	b	c	y	#	m

Invalid

Sequential List: Delete

void delete(int position)

- Removes a data item from the list at the given **position** and decreases **size** by one
- On deletion, **data** after **position** is moved towards the list front by one
- The list remains contiguous after each deletion
- If the **size** is significantly smaller than **capacity**, the list **capacity** can be decreased
- In the <**vector**> library, **shrink_to_fit()** can be called to try and reduce the capacity to size

Example: Sequential List Implementation. Delete

0	1	2	3	4	5	6	7
z	a	k	b	c	y	#	#

delete(3) =>

z	a	k	c	y	#y	#	#
---	---	---	---	---	----	---	---

Invalid

delete(0) =>

a	k	c	y	#y	#y	#	#
---	---	---	---	----	----	---	---

delete_front () is the same as delete(0)

delete_end () is the same as delete(size-1)

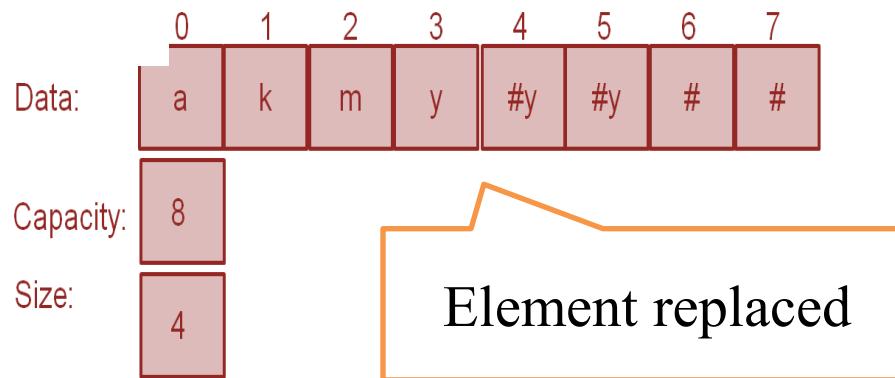
Sequential List: Select, Replace and Size

`DataType select(int position)`

- Implemented as array lookup; for example, as `data[index]`

`void replace(int position, DataType value)`

- Implemented as array replacement; for example `replace(2 , 'm')`



`int size()`

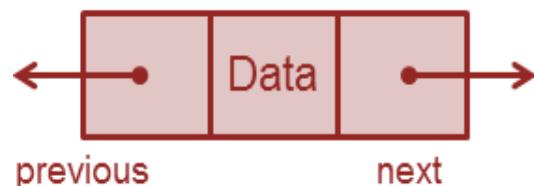
- Implemented by simply returning the size value

Linked List

- Linked lists are made of nodes which are data structures that contain both data items and pointers to other nodes in the sequence.

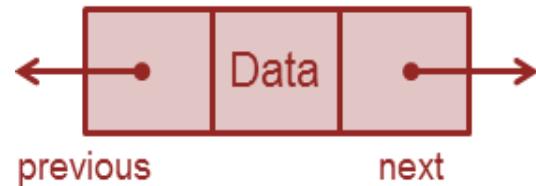


- Previously, the singly linked list implementation was shown, and in this lecture doubly linked list is implemented:



Doubly Linked List

- An implementation of Linked List that augments each node with a **previous (prev)** pointer

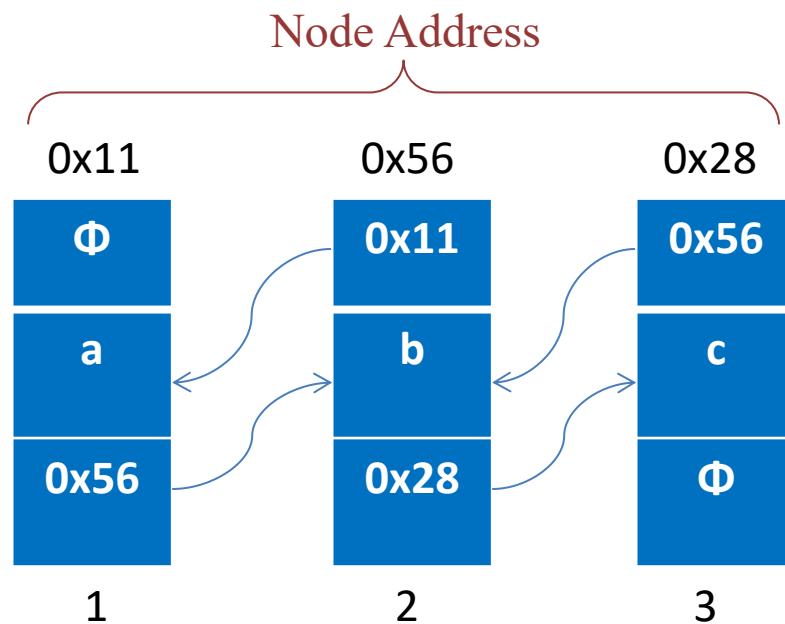
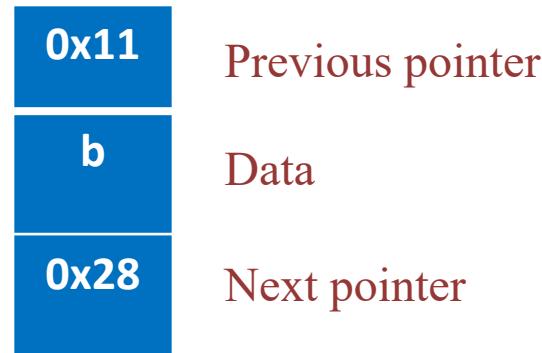


This implementation will also use the following variables:

- **first** – pointer to the start of the list; set to **NULL** at start
- **last** – pointer to the end of the list; set to **NULL** at start
- **size** – the number of inserted elements in the list; set to 0 at start

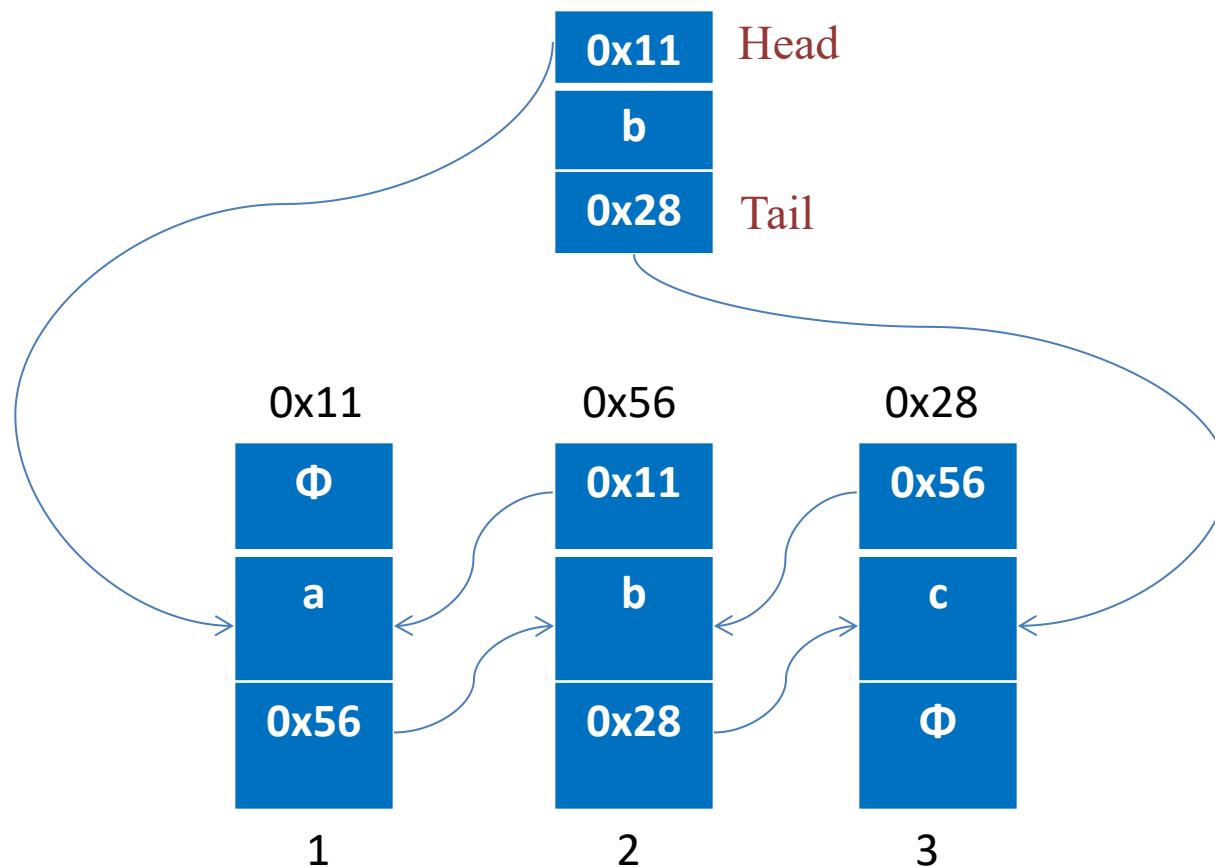
Doubly Linked List

A node in doubly linked list →



← A doubly linked list of three nodes

Doubly Linked List



Doubly Linked List

void insert(DataType value, int position)

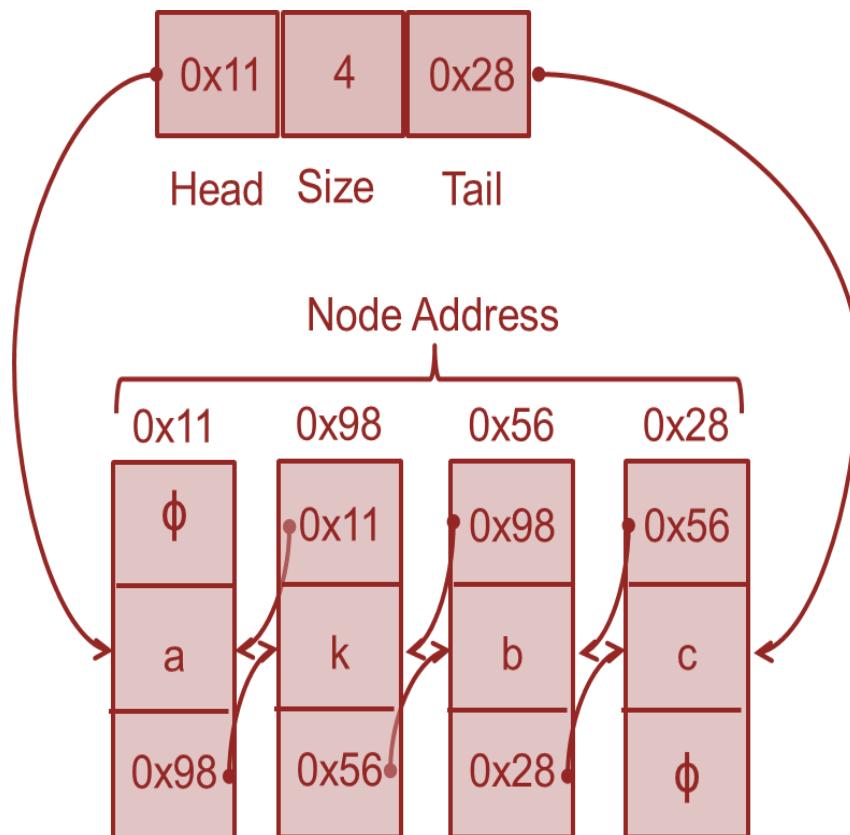
Example:

insert('c',0) =>

insert('b',0) =>

insert('a',0) =>

insert('k',1)



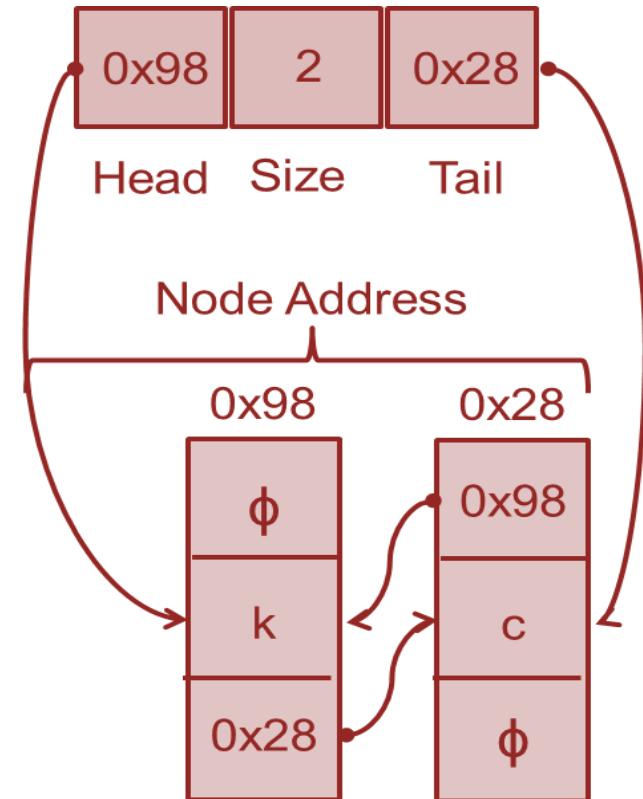
Doubly Linked List

void delete(int position)

- Implemented by iterating through the list until position and removing that node
- Example: delete(2) => delete(0)

DataType select(int position)

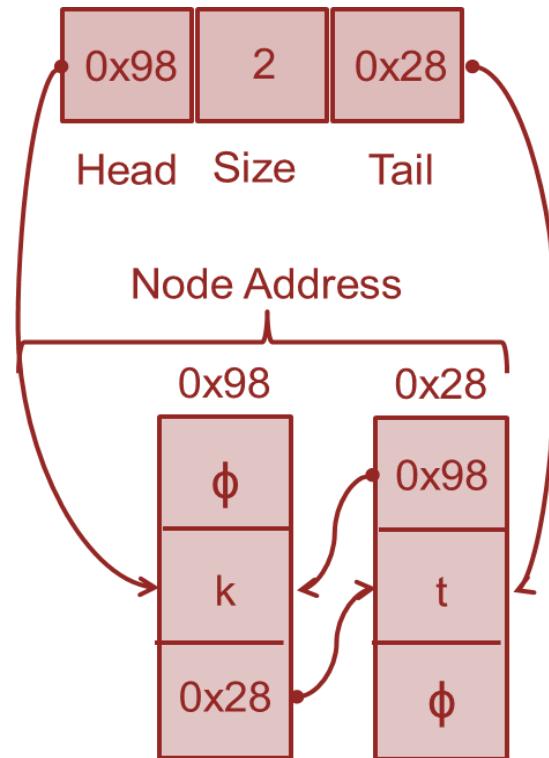
- Implemented by iterating through the list until position



Doubly Linked List

void replace(int position, DataType value)

- Implemented by iterating through the list until position and replacing its value
- Example: replace(1, 't')



int size()

- Implemented by returning size

Comparison of Implementations

Array Implementation	Linked List Implementation
Allows for insert operations to be very fast , but has a capacity that is limited by the size of its underlying array	Have slower insert since each insert requires iterating through the list, but can adjust its size since pointers need to be manipulated
Cost (in terms of time) of accessing elements does not change with the number of data elements in the list	Cost (in terms of time) of accessing elements linearly changes with the number of data elements in the list
Cost (in terms of time) of inserting/removing from beginning linearly changes with the number of data elements in the list	Cost (in terms of time) of inserting/removing from beginning linearly changes with the number of data elements in the list

Comparison of Implementations

Array Implementation	Linked List Implementation
Cost (in terms of time) of inserting/removing from end does not change with the number of data elements in the list	Cost (in terms of time) of inserting/removing from end linearly changes with the number of data elements in the list
Cost (in terms of time) of inserting/removing from the middle linearly changes with the number of data elements in the list	Cost (in terms of time) of inserting/removing from the middle linearly changes with the number of data elements in the list

Next Lecture

We focus on:

- more implementation examples
- review some challenge questions about nodes, pointers and lists

Readings/ Study Reference

Course Textbook: “*Data Structure and Algorithms in a Nutshell*” A. Wong et al.

Chapter 3. List

Section 3.1. Sequential List Implementation

Section 3.2. Linked List Implementation

Section 3.3. Comparing Implementations

The End of Lecture

Course: Data Structures and Algorithms

Instructor: Homeyra Pourmohammadali

Review of Selected Challenge Questions

Learning Outcomes

By the end of this lecture you will be able to:

1. review and discuss challenge questions about pointers and linked data representation
2. review and discuss challenge questions about implementations of sequential lists and linked lists

Challenge Question. Nodes and Pointers

Note: From the challenge questions of Sec 2.5 of your course textbook.

1. Draw the node structures of variables **a**, **b**, **c** with their value and memory address (you assign memory location any as long as it's consistent) as they change. Provide the output resulting from the “cout” instruction (See the code in the next slide).

Challenge Question. Nodes and Pointers

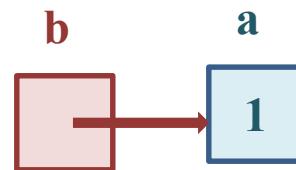
1. int a = 1;
2. int *b = &a;
3. int **c = &b;
4. int *d = b;
5. cout<< "a=" << a << endl;
6. cout<< "(*c)=" << (*c) << endl;
7. c = &d;
8. (**c) = 3;
9. cout<< "(*b)=" << (*b) << endl;
10. cout<< "(*c)=" << (*c) << endl;
11. cout<< "(**c)=" << (**c) << endl;

Challenge Question. Nodes and Pointers

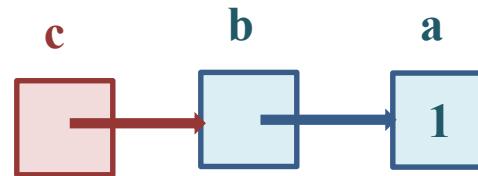
1. `int a = 1;`



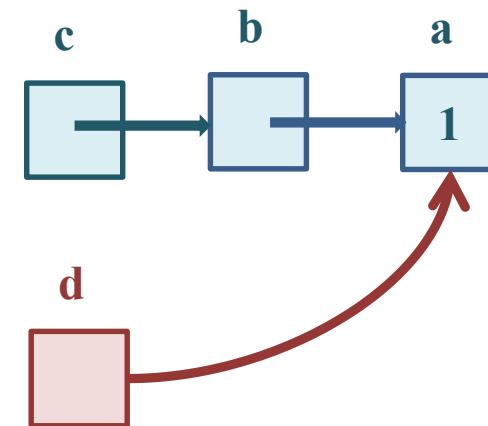
2. `int *b = &a;`



3. `int **c = &b;`



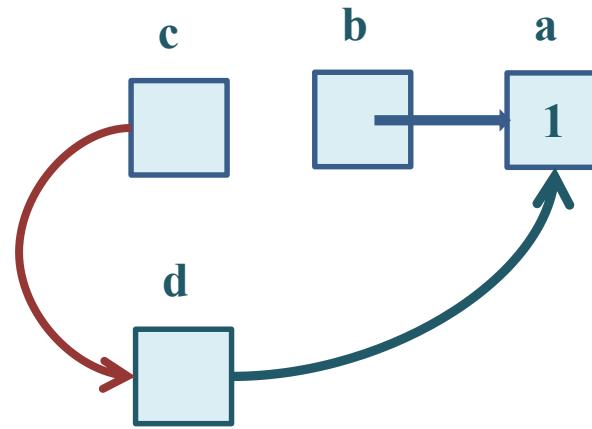
4. `int *d = b;`



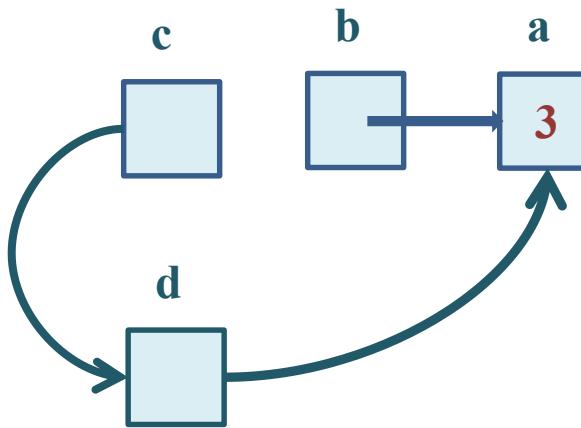
Challenge Question. Nodes and Pointers

5. `cout<< "a=" << a << endl;` a = 1
6. `cout<< "(*c)=" << (*c) << endl;` (*c) = 0x#####

7. `c = &d;`



8. `(**c) = 3;`



Challenge Question. Nodes and Pointers

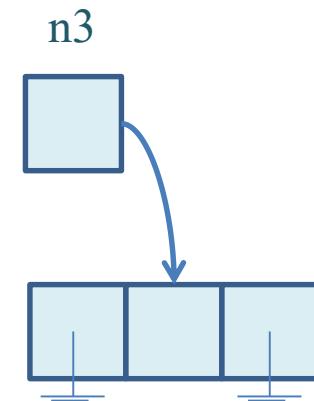
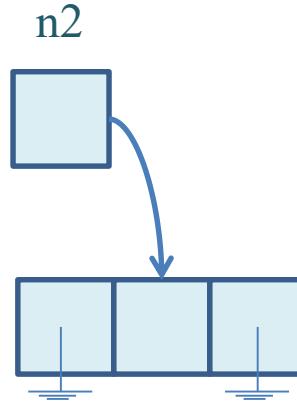
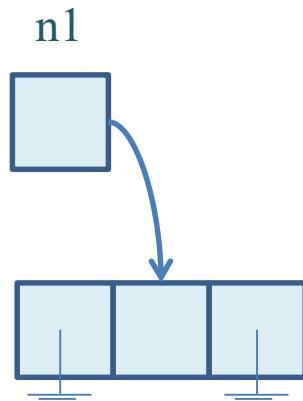
Note: From the challenge questions of Sec 2.5 of your course textbook.

2. Draw the node structure resulting from the following code:

1. Node *n1 = new Node();
2. Node *n2 = new Node();
3. Node *n3 = new Node();
4. n1->next = n3; n3->prev = n1;
5. n3->next = n2; n2->prev = n3;
6. n2->next = n1->next->next->prev;

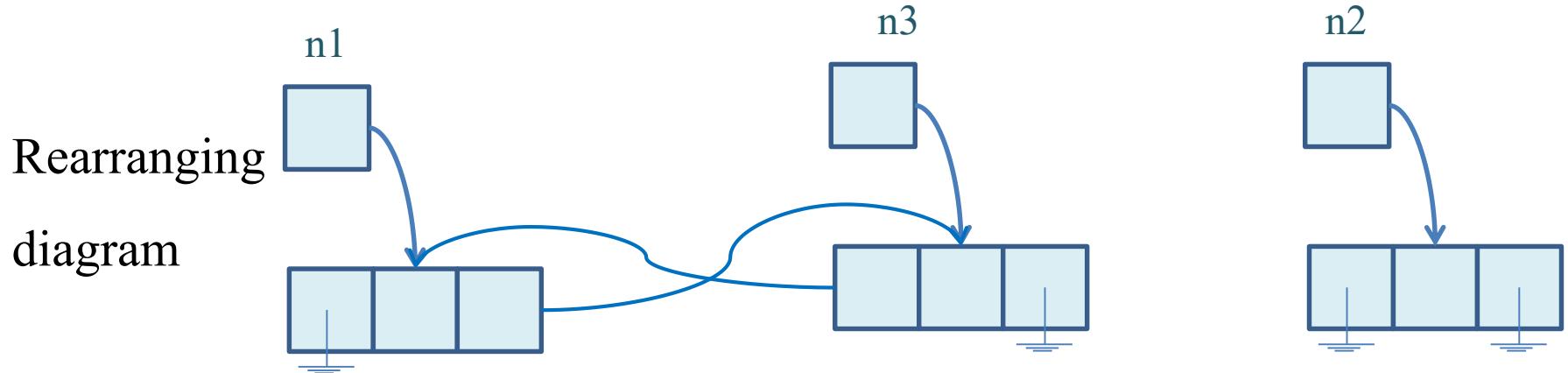
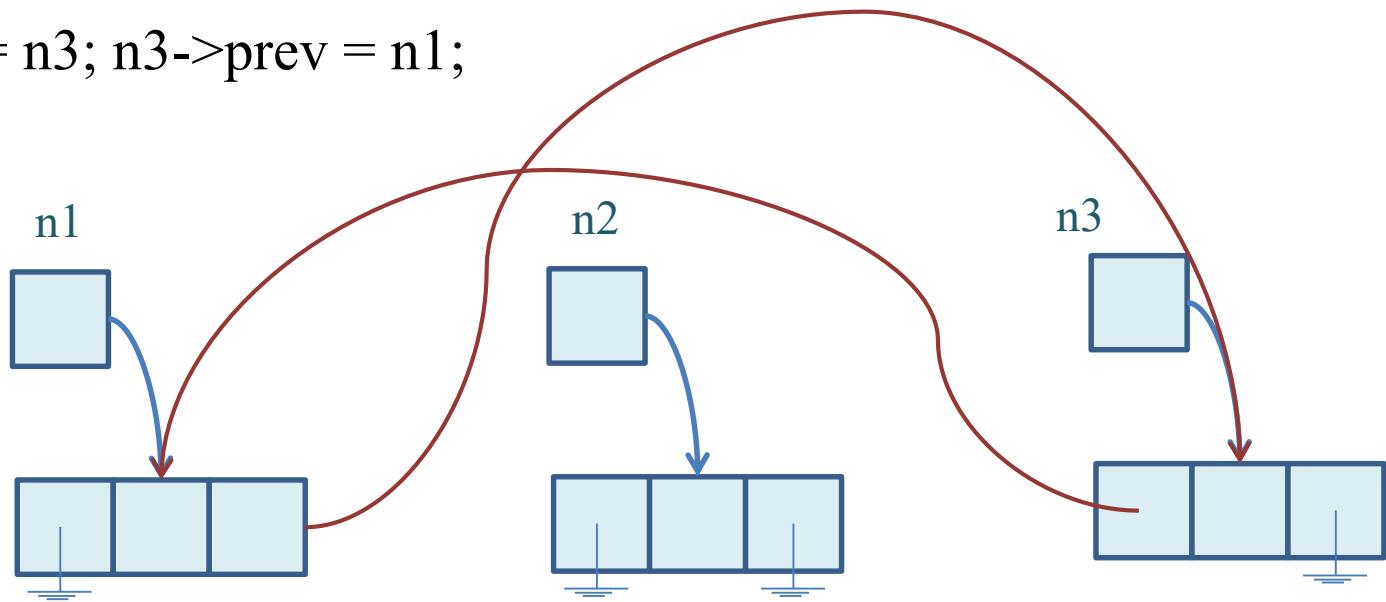
Challenge Questions

1. `Node *n1 = new Node();`
2. `Node *n2 = new Node();`
3. `Node *n3 = new Node();`



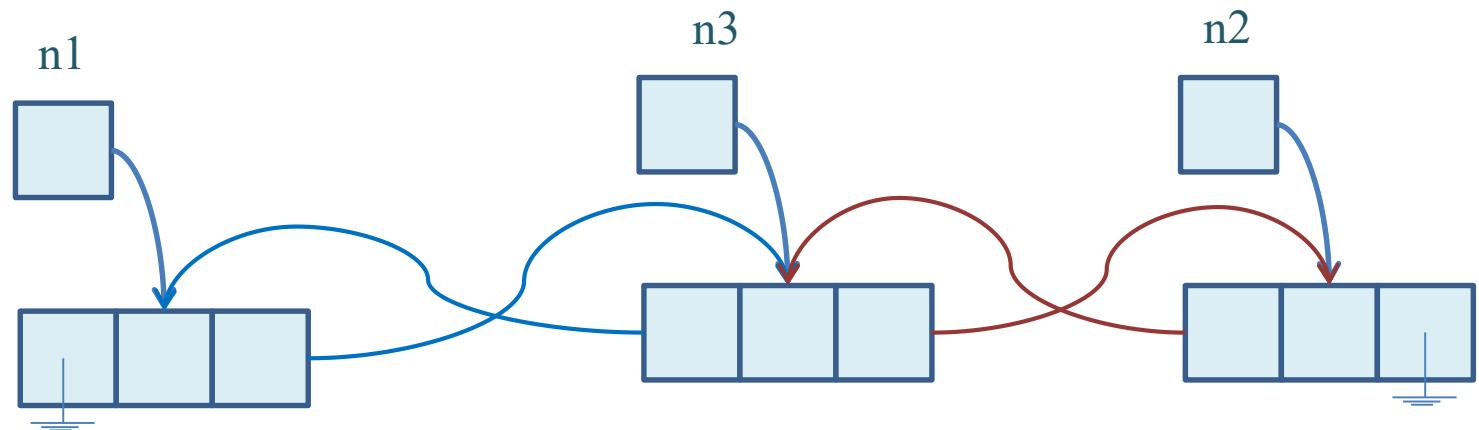
Challenge Questions

4. $n1 \rightarrow \text{next} = n3; n3 \rightarrow \text{prev} = n1;$



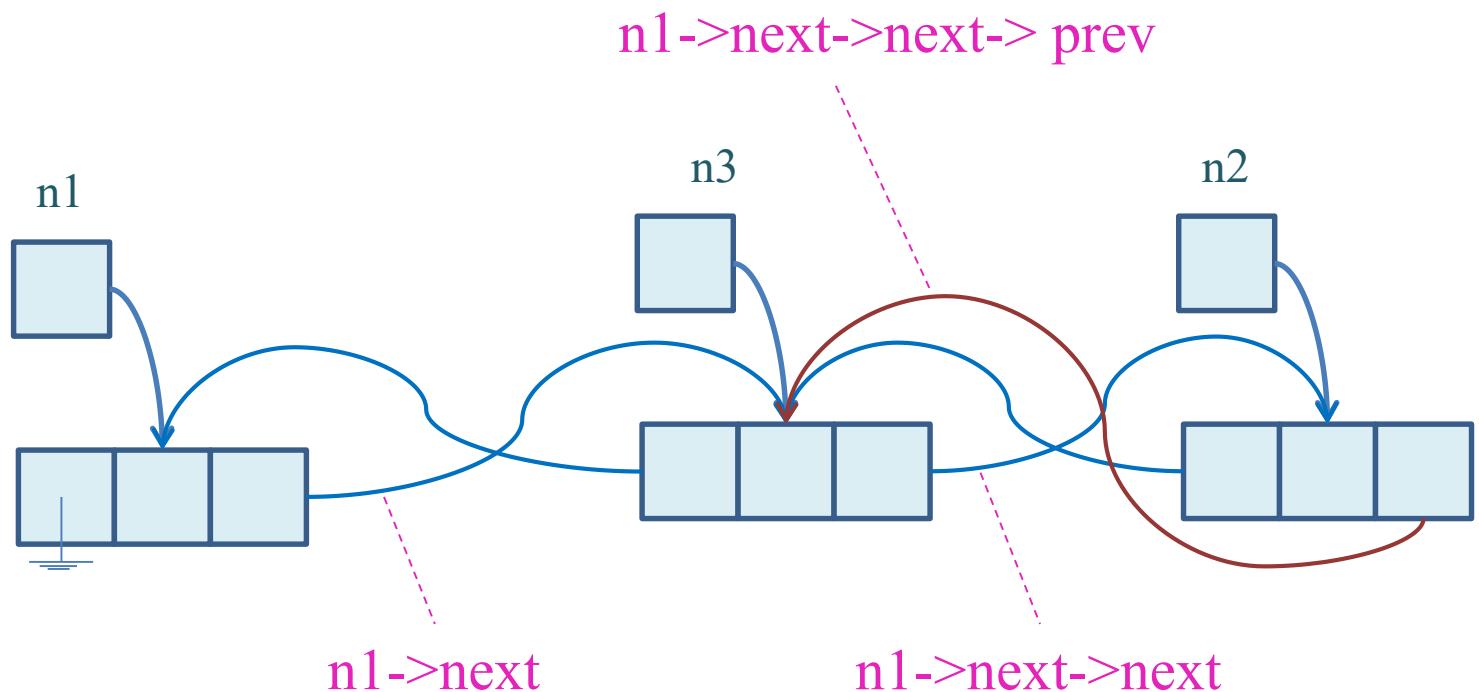
Challenge Questions

5. $n3->next = n2$; $n2->prev = n3$;



Challenge Questions

6. $n2->next = n1->next->next->prev;$



Challenge Question. Sequential versus Linked Lists

Note: From the challenge questions of Sec 3.4 of your course textbook.

1. Sequential lists and linked lists share the same abstract data type (ADT), and thus provide the same functionality. However, their internal implementations of this “signature” are different. Name a few instances when it would be better to use a sequential list rather than a linked list and vice versa.

Challenge Question. Sequential versus Linked Lists

Solution:

In terms of size of list:

- **Sequential list:** upper limit on the number of elements to be stored (it is based on a pre- allocated array); difficult to predict this limit ahead of the time leading to either not allocating enough space or wasting memory.
- **Linked list:** number of elements can grow indefinitely.

Challenge Question. Sequential versus Linked Lists

Solution:

In terms of insertion operation:

- **Sequential list:** It can be expensive. All the elements in the array with index higher than this location (i.e., to the right of the insertion location) may need to be shifted to the right to create a free space to add the element.
- **Linked list:** the insertion of an element in a linked list is easy and can be implemented by changing pointers.

Challenge Question. Sequential versus Linked Lists

Solution:

In terms of deletion operation:

- **Sequential list:** Deleting an element from an arbitrary location in a sequential list can be more expensive compared to deleting an element from a linked list.
- **Linked list:** less expensive

Challenge Question. Sequential versus Linked Lists

Solution:

Drawbacks of the linked list:

- Elements need to be accessed sequentially starting from the first node in a linked list. In case of sequential list, elements can be accessed directly (random access).
- Linked Lists need more memory space compared to sequential lists since not only the data values are stored, but also pointers to the previous and the next nodes.

Challenge Question. Sequential versus Linked Lists

Solution: In Terms of Applications:

Instances when it is better to use a **sequential** list:

1. A signal acquired from a sensor
 - A quick access to the elements of the signal may be preferable in order to implement different signal processing algorithms
 - No need for any insertions or deletions
 - Normally, the signal is acquired in frames of fixed length
2. List of branches in a retail system (e.g., software you find in grocery stores)

Challenge Question. Sequential versus Linked Lists

Solution: In Terms of Applications:

Instances when it is better to use a [linked list](#):

1. A log of events (such as failure, warnings, etc) in a machine or a device
 - It is hard to predict the number of events ahead of time
 - Elements are normally accessed in a sequential order
2. Alphabetically ordered list of customers in a customer relation management software
 - A lot of insertion and deletions
 - It is hard to predict the number of customers ahead of time
 - Access time is not critical.

Challenge Question. Insertion/Deletion

Note: From the challenge questions of Sec 3.4 of your course textbook.

2. Draw the sequential and linked list representations of the following code, instruction by instruction. Assume the sequential list has a capacity of 3.

```
list.insert('a', 0);
list.insert('b', 1);
list.insert('c', 0);
list.insert_back('d');
list.delete_front();
list.delete_back();
list.insert('d', 1);
```

Challenge Question. Insertion/Deletion

Solution

	sequential	Linked list	
list.insert('a',0);	Data: Size: 1	 Head size Tail	
list.insert('b',1);	Data: Size: 2	 Head size Tail	
list.insert('c',0);	Data: Size: 3	 Head size Tail	
list.insert_back('d');	Size = Capacity Error!		

Challenge Question. Insertion/Deletion

Solution

list.delete_front();	Data Size Capacity	a b # 1 3 0x11 0x28 0x96	0x11 0x28 0x96	0x11 3 0x96 Head size Tail
list.delete_back();	Data Size Capacity	a # # 1 3 0x11 0x28	0x11 0x28	0x11 2 0x28 Head size Tail
list.insert('d',1);	Data Size Capacity	a d # 1 3 0x11 0x64 0x28	0x11 0x64 0x28	0x11 3 0x28 Head size Tail

Challenge Question. Resizing Sequential List

Note: From the challenge questions of Sec 3.4 of your course textbook.

3. Sequential list implementations do not resize themselves on demand. Modify the given sequential list examples, to include the following special features:
 - It has a minimum capacity
 - If the number of inserted elements goes beyond the capacity of the list, the list should resize itself to twice its current capacity.
 - If the number of inserted elements goes below one third of its current capacity, the list should resize itself to half of its current capacity.

Challenge Question. Resizing Sequential List

Solutions: Modifications to the “insert” method to allow reallocation to a bigger array are highlighted in yellow.
Check the next slide for the C++ implementation.

```
bool SequentialList::Insert(DataType Val, unsigned int Index )  
{  
    // Insertion will create gaps in the array  
    if (Index > Size_) return false;  
    // The array is already Full  
    if (Capacity_ == Size_)  
    {  
        // Step1 : Doubling the capacity  
        Capacity_ = Capacity_*2;  
        // Step3: Allocating the new array  
        DataType* Temp = new DataType[Capacity_];  
        // Step3: Copying the data from the old array  
        for (int i = 0; i< Size_; i++)  
        {  
            Temp[i] = Data_[i];  
        }  
        // Step4: Deleting the old array  
        delete [] Data_;  
        // Step5: Assigning Data_ to the newly created Array  
        Data_ = Temp;  
    }  
}
```

Continue next
page →

```
// Shifting items with higher index to left

for (int i = Size_; i > Index; i--)
{
    Data_[i] = Data_[i-1];
}

// Inserting the new element

Data_[Index] = Val;

// Updating the size

Size_ = Size_ + 1; return true;
}
```

Next Lecture

We focus on:

- Mathematical Concept of Recursion
- Recursive Behavior of Computing Problems
- Understanding Call Trees and Call Traces

Readings/ Study Reference

Course Textbook: “*Data Structure and Algorithms in a Nutshell*” A. Wong et al.

Chapter 2. Linked Data Representation

Section 2.5. Challenge Questions

Chapter 3. List

Section 3.4. Challenge Questions

The End of Lecture

Course: Data Structures and Algorithms

Instructor: Homeyra Pourmohammadali

**Introduction to Recursion,
Call Trees and Call Traces**

Motivation

- Consider a sequence of similar Russian nesting dolls
- Each time you open a doll , you see a smaller version of it inside, until you reach the innermost doll that cannot be opened anymore.



<https://www.therussianstore.com/blog/the-history-of-nesting-dolls/>

Motivation

Consider the Factorial function, $n!$

- To compute $4!$, compute $4 \partial 3 \partial 2 \partial 1$
- However, the same function can be viewed as $4 \partial 3!$
- That is, we can represent the same problem using a smaller version of itself ($3!$), which is then combined (4∂) to obtain a solution to the original problem ($4!$)
- $n! = n. (n-1)!$

Motivation

Mathematical Example. Recursive Sequence

Recursive (in terms of previous terms):

$$a_1 = 1, \quad a_{n+1} = \sqrt{1 + a_n} \quad (n \geq 1)$$

$$a_1 = 1, \quad a_2 = \sqrt{1 + 1}, \quad a_3 = \sqrt{1 + \sqrt{1 + 1}}, \quad etc.$$

$$a_4 = \sqrt{1 + \sqrt{1 + \sqrt{1 + 1}}}$$

$$a_5 = \sqrt{1 + \sqrt{1 + \sqrt{1 + \sqrt{1 + 1}}}}$$

Motivation

Mathematical Example. The Fibonacci (Recursive) Sequence

$$F_0=0, F_1=1, \quad F_n=F_{n-1}+F_{n-2} \quad (n \geq 1)$$

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

$F_0 \quad F_1 \quad F_2 \quad F_3 \quad F_4 \quad F_5 \quad F_6 \quad F_7 \quad F_8 \quad F_9 \quad F_{10}$

$$F_5 = 5 = F_4 + F_3 = 2 + 3 = 5$$

$$F_{10} = 55 = F_9 + F_8 = 34 + 21 = 55$$

Learning Outcomes

By the end of this lecture you will be able to find out:

1. what the recursion technique is in computation,
2. what the base case and general (recursive) case are,
3. what the call trees are,
4. how to trace the calls within the tree.

Recursive Thinking

As a Problem Solving Strategy

- A computing problem can be solved by being represented as smaller versions of itself

Recursion

- The approach of representing and solving a computing problem using smaller, recurrent versions of itself.
- It is the process of repeating item in a self-similar way.

Recursion and Computing Problems

- Recursion is not necessary but may be a useful technique to solve a number of computing problems in less complex manner.
- Theoretically, a recursive algorithm can be represented non-recursively.
- However, a recursive solution may be shorter and simpler to write, especially for problems that exhibit recursion.

Recursive Function in C/C++

- A function that in its definition contains a call back to itself
- Require more space compared to an iterative program.
- It can be more difficult to debug compared to an equivalent iterative program.
- It uses more processor time.

Base Case and General Case

Base Case: The case for which the solution can be stated nonrecursively. (e.g. $0!=1$ in factorial example)

General (recursive) Case: The case for which the solution is expressed in terms of a smaller version of itself. (e.g. $n!$ can be represented as $n \cdot (n-1)!$)

Recursive Algorithm:

A solution that is expressed in terms of

- (1) smaller instances of itself and
- (2) a base case

Example: The Fibonacci Series. Base & General Cases

- $F_n = F_{n-1} + F_{n-2}$, where $n > 1$ [Recursive (General) Case]
- $F_0 = 0$; $F_1 = 1$ [Base Cases 1 and 2]
- To compute F_2 , use: $F_2 = F_1 + F_0 = 1 + 0 = 1$
- The recursion application needs to terminate with a base case of the problem

Example. The Fibonacci Series. Implementation

```
int fibonacci (int n)
{
    if (n <= 0) {          // base case 1
        return 0;
    } else if (n == 1) {    // base case 2
        return 1;
    } else if (n > 1) {    // recursive (general) case
        return fibonacci(n-1) + fibonacci(n-2);
    }
}
```

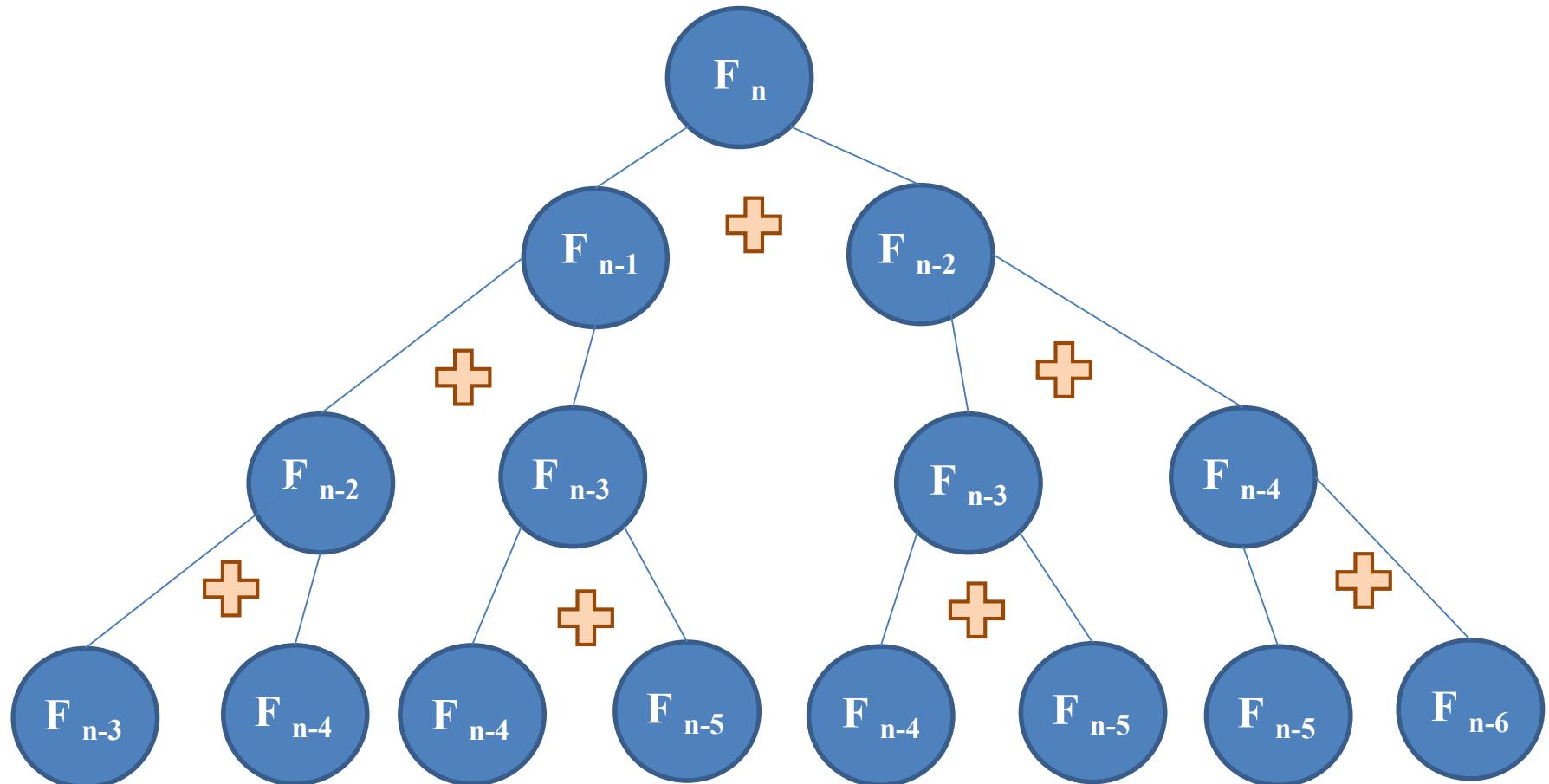
You Try. Coding Factorial Function

Write a few lines to implement factorial $n!$ in C++.
Specify base case and general (recursive) case.

Call Trees and Call Traces

- Allow visualization of function calls and recursion, where each call spawns smaller representations of itself
- This is represented as tree branches that are branching out from the tree root
- To trace the calls, we start off from the top of node (root) of call tree and go all the way down to the bottom of tree, which signifies the end of recursive calls.
- Values at base case are defined, which are used to find the value of upper level of tree. Then the process goes upward until we get to the root of the call tree.

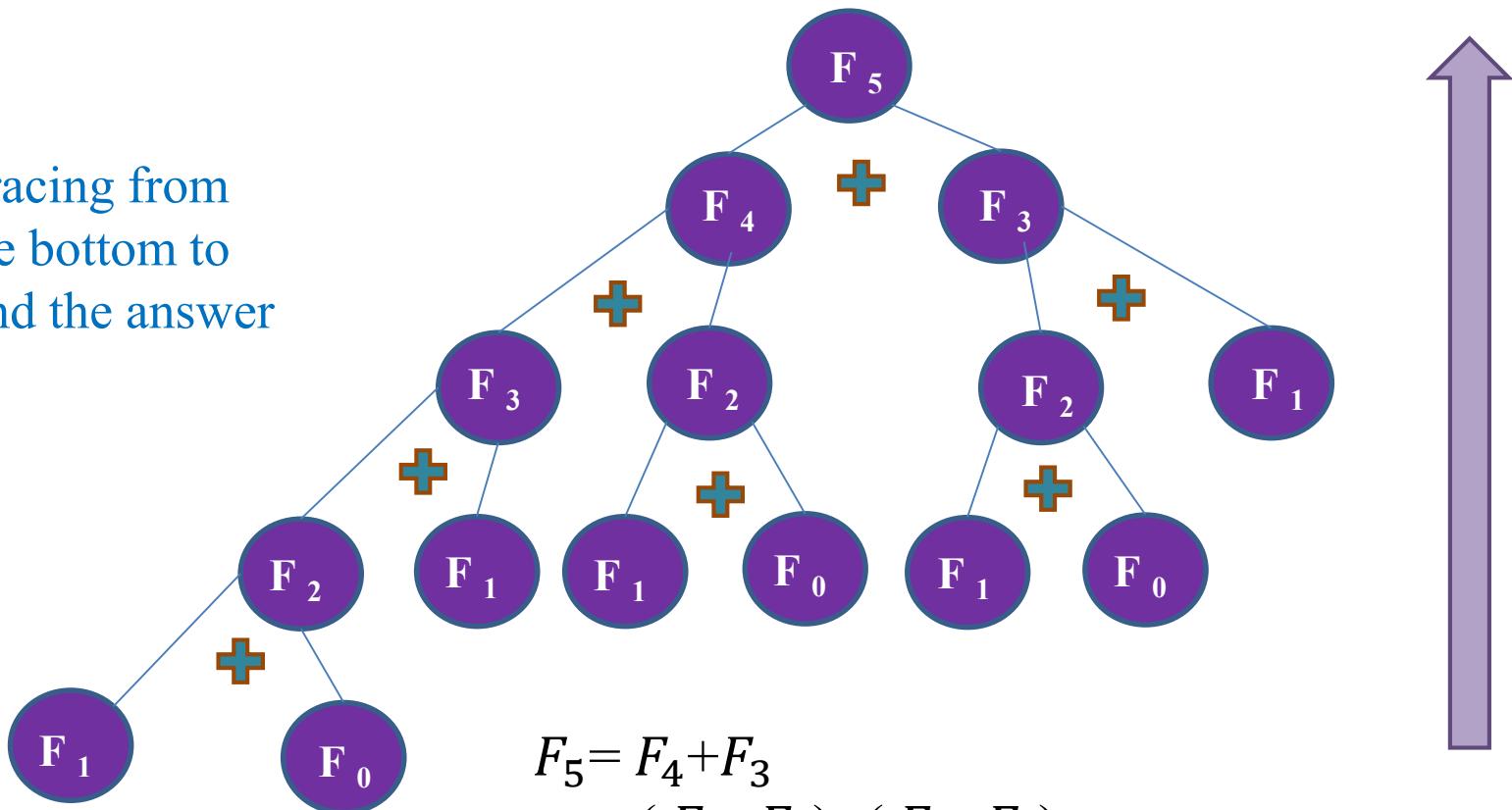
The Fibonacci Series: Call Tree to find nth number



$$F_0=0, F_1=1, \quad F_n = F_{n-1} + F_{n-2} \quad (n \geq 1)$$

Example. Call tree to find 6th number in the Fibonacci Series

Tracing from
the bottom to
find the answer



$$\begin{aligned}F_5 &= F_4 + F_3 \\&= (F_3 + F_2) + (F_2 + F_1) \\&= (F_2 + F_1) + (F_1 + F_0) + (F_1 + F_0) + 1 \\&= (F_1 + F_0) + 1 + 1 + 0 + 1 + 0 + 1 \\&= 1 + 0 + 1 + 1 + 0 + 1 + 0 + 1 \\&= 5\end{aligned}$$

Designing Recursive Algorithms

When designing recursive algorithms, answer the following two questions:

- **Q1:** How to represent the given problem using smaller versions of itself?
(i.e., how to design recursive case(s))
- **Q2:** When does the recursion reach its end point?
(i.e., how to design base case(s))

Designing Recursive Algorithms

For **Q1**, define what smaller means:

- Smaller could mean smaller input
- More precisely, smaller means closer to the end point of the problem

For **Q2**, define what is the end point:

- A problem reaches its end point when there is a trivial answer to the problem
- To refine it further would not be needed or would lead to undefined answers

Next Lecture

We focus on:

- Designing Recursive Algorithms
 - Infinite Regression
 - Bottom-up and Top-down Approaches
 - Divide and Conquer Approach

Readings/ Study Reference

Course Textbook: “*Data Structure and Algorithms in a Nutshell*” A. Wong et al.

Chapter 4. Recursion

Section 4.1. Why Recursion?

Section 4.2. Call Trees

Section 4.3. Call Traces

The End of Lecture

Course: Data Structures and Algorithms

Instructor: Homeyra Pourmohammadali

**You Try Question and Solution
Introduction to Recursion,
Call Trees and Call Traces**

You Try. Coding Factorial Function

Write a few lines to implement factorial $n!$ in C++.
Specify base case and general (recursive) case.

You Try Solution. Coding Factorial Function

Write a few lines to implement factorial $n!$ in C++.
Specify base case and general (recursive) case.

```
int Factorial(int number)
// Pre: number is nonnegative.
{
    if (number == 0)
        return 1;
    else
        return number * Factorial(number-1);
} // Factorial involves a recursive call to the
  // function, with the parameter number - 1.
```

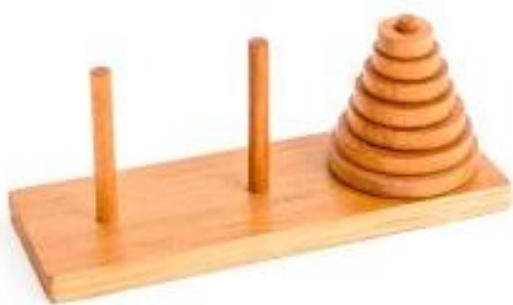
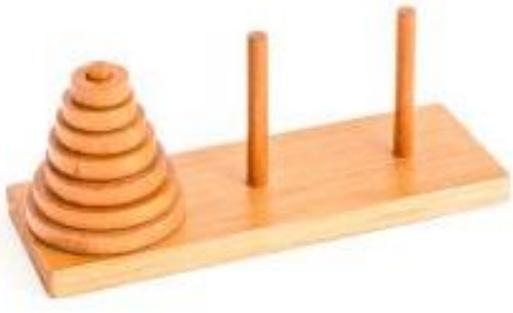
The End of Lecture

Course: Data Structures and Algorithms

Instructor: Homeyra Pourmohammadali

Designing Recursive Algorithms

Motivation



- Towers of Hanoi game or puzzle.
- 3 pegs and n disks of different sizes (usually 3 to 8).
- Strategy: move only one disk at a time, no larger disk can be placed on top of a smaller disk.
- Can we design an algorithm that can generate the solution, if we later change parameters, such as number of disks?

Learning Outcomes

By the end of this lecture you will be able to find out the answers to the following questions:

- What are the possible approaches of recursion?
- How to avoid the infinite recursion?
- How top-down is different from bottom-up approach?
- What is divide and conquer approach?
- Review “Tower of Hanoi” game as a classical recursive problem.

Recall: Design of A Recursive Algorithm

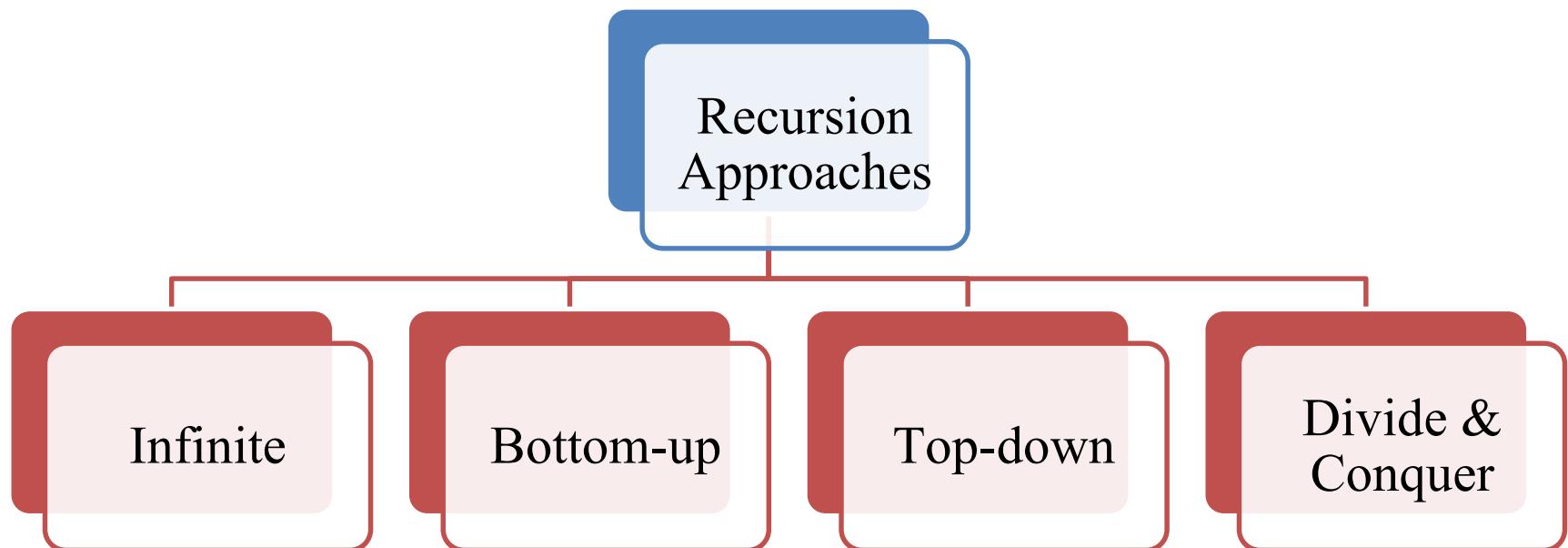
Recursive Algorithm: A solution that is expressed in terms of

- (1) smaller instances of itself and
- (2) a base case

We should answer two questions:

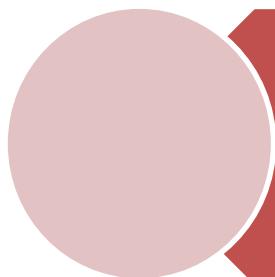
- **Q1:** How to represent the given problem using smaller versions of itself? (i.e., how to design recursive case(s))
- **Q2:** When does the recursion reach its end point? (i.e., how to design base case(s))

Possible Cases

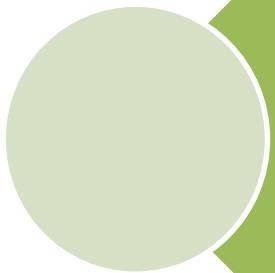


Common Mistakes in Design

Make sure the recursion stops at a certain point



The algorithm is missing an ending case



The recursive call does not lead to a smaller version of itself

Infinite Recursion

It is a recursive algorithm that never terminates.

Example. Infinite Recursion. No ending case

```
int function1(int p)
{
    return p*function1(p-2);
}
// function1 never stops calling another version of itself.
```

Infinite Recursion

Example. Infinite Recursion with ending and recursive cases

```
int function2(int p)
{
    if (p==0) return 0;
    else return p*function2(p-2);
}
```

// function2 only stops when the input p is positive even
// number. if p is 1 for instance, the input of the first
// recursive call is negative and function never stops calling
// itself.

Infinite Recursion

Example. Infinite Recursion. Having bigger version of itself

```
int function3(int p)
{
    if (p<0) return 0;
    else return p*function3(p+2);
}
```

// the recursive algorithm has a bigger version of itself (input
// value of function3 is increasing)

Infinite Recursion: Lessons Learned

- Ensure that a recursive function eventually terminates
- Ensure the ending case is properly set
- Ensure recursive call goes in the right direction
- Each recursive function use a chuck of memory space
- Infinite recursion takes up all memory and causes the program to freeze

Bottom-Up Problem Solving

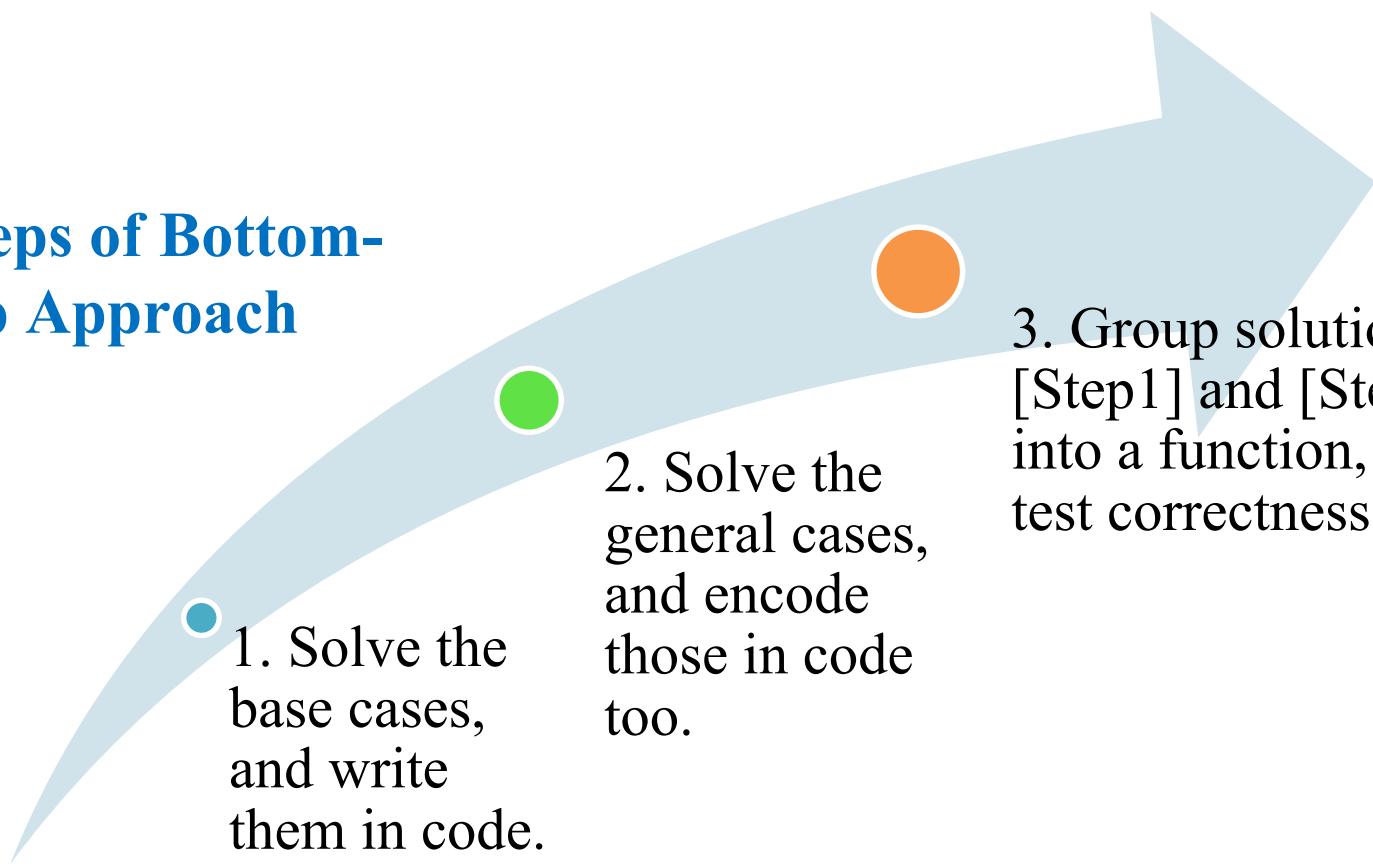
Exercise: Design a recursive algorithm that computes n factorial ($n!$)

Steps of Bottom-Up Approach

1. Solve the base cases, and write them in code.

2. Solve the general cases, and encode those in code too.

3. Group solutions for [Step1] and [Step2] into a function, and test correctness.



Bottom-Up Problem Solving

// [Step3] group into a function and test correctness

```
int factorial(int n) {
```

// [Step1] base case

```
    if (n < 1) return 1;
```

// [Step2] recursive case

```
    else return n * factorial(n-1);
```

```
}
```

Top-Down Problem Solving

Steps

1. Solve the general (recursive) cases, and encode them in code

2. Keep dividing the problem into smaller versions until base cases are reached; once the base cases are reached and solved, encode them in code

3. Group solutions for Steps 1 & 2 into a function, develop test cases to check correctness, and refine the code until it passes all required tests

Example. Top-Down Recursion Approach. Design a recursive function that finds the sum of squares between two positive integers a and b , where $a \leq b$, using the top-down approach.

Approach:

1. Begin with a recursive call of a smaller version
2. Add this value to the value that you can calculate

Example Solution. Top-Down Recursion Approach.

$$SS(a, b) = SS(a, b - 1) + b^2$$

$$SS(a, b) = SS(a, b - 2) + SS(a, b - 1) + b^2$$

$$SS(a, b) = \dots .$$

$$SS(a, b) = SS(a, a) + \dots + (b - 2)^2 + (b - 1)^2 + b^2$$

$$SS(a, b) = a^2 + \dots + (b - 2)^2 + (b - 1)^2 + b^2$$

Example Solution. Top-Down Recursion Approach.

```
int SS_TD(int a, int b)
{
    if (a==b) {
        return b*b;
    }
    else {
        return SS_TD(a, b-1)+(b*b);
    }
}
```

You Try 1. Bottom-up Recursion Approach.

How the recursive design of previous example (sum of squares) would be different if you use the bottom-up approach?

Approach:

1. Evaluate the value you can calculate (for instance squaring the starting value of the call)
2. Add this value to a recursive call of a smaller version

Divide and Conquer Approach



Break problem into smaller subproblems

Call recursively until subproblems are solved

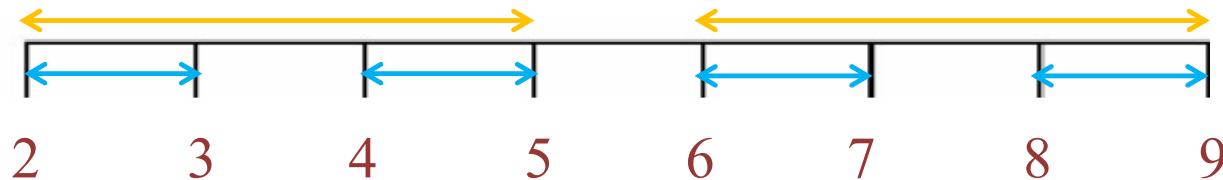
Combine solutions of subproblems into solution for the original problem

Divide and Conquer Approach

- In this approach the problem is broken down to at least two subproblems; therefore multiple recursive calls need to be made.
- Many efficient algorithms are based on this approach, such as:
 - quick sort and merge sort algorithms
 - algorithm for finding the closest pair of points

Example. Divide and Conquer Approach

Split the problem in the middle; for instance for $SS(2, 9)$:



$$SS(2,9) = SS(2,5) + SS(6,9)$$

$$SS(2,9) = SS(2,3) + SS(4,5) + SS(6,7) + SS(8,9)$$

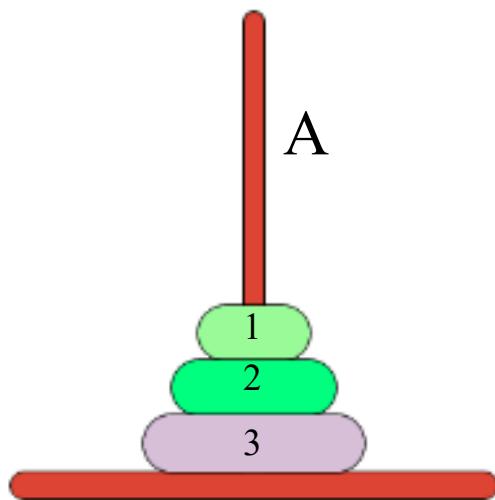
$$SS(2,9) = SS(2,2) + SS(3,3) + SS(4,4) + \dots + SS(9,9)$$

$$SS(2,9) = 2^2 + 3^2 + \dots + 9^2$$

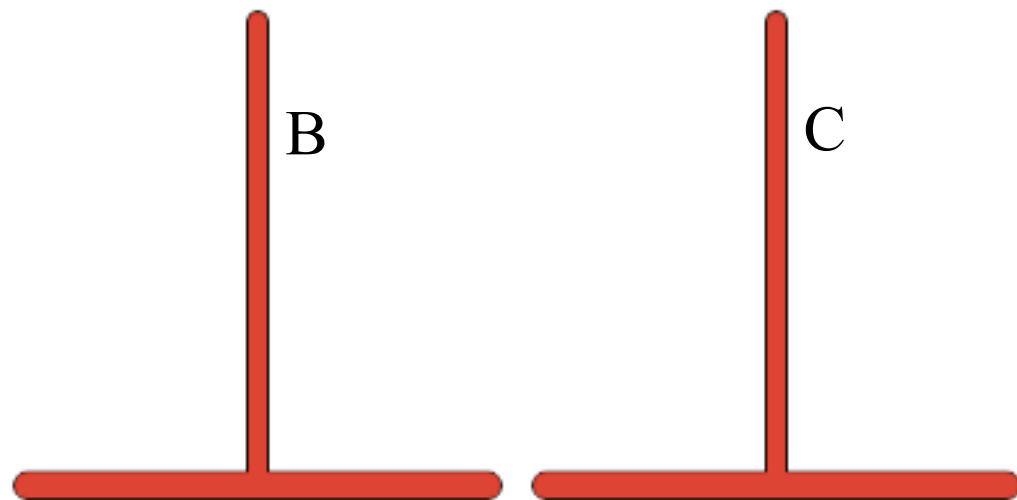
$$SS(2,9) = 284$$

Towers of Hanoi

Starting tower

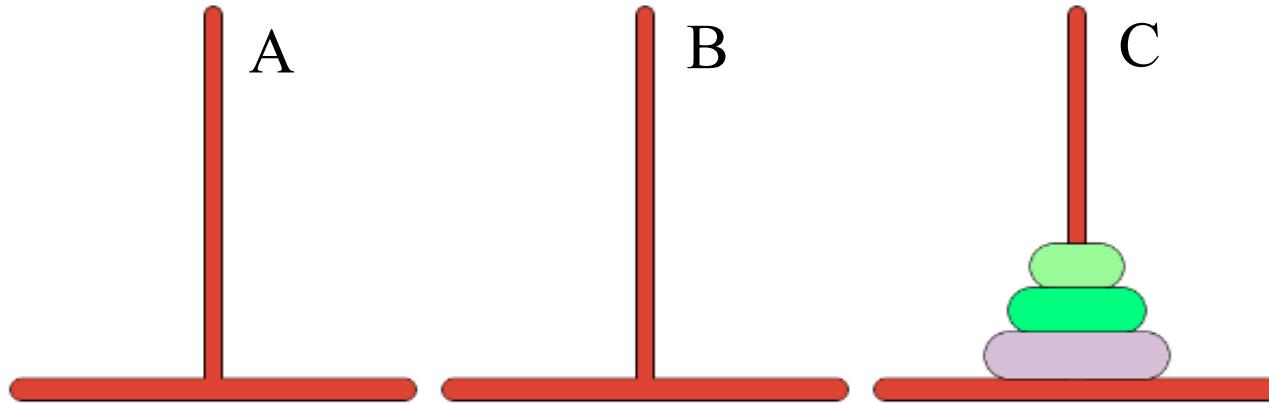


Destination tower



- Consider 3 pegs (tower) and 3 disks of different sizes
- move only one disk at a time, no larger disk can be placed on top of a smaller disk.
- What is the minimum number of moves to transfer all disks to destination tower?

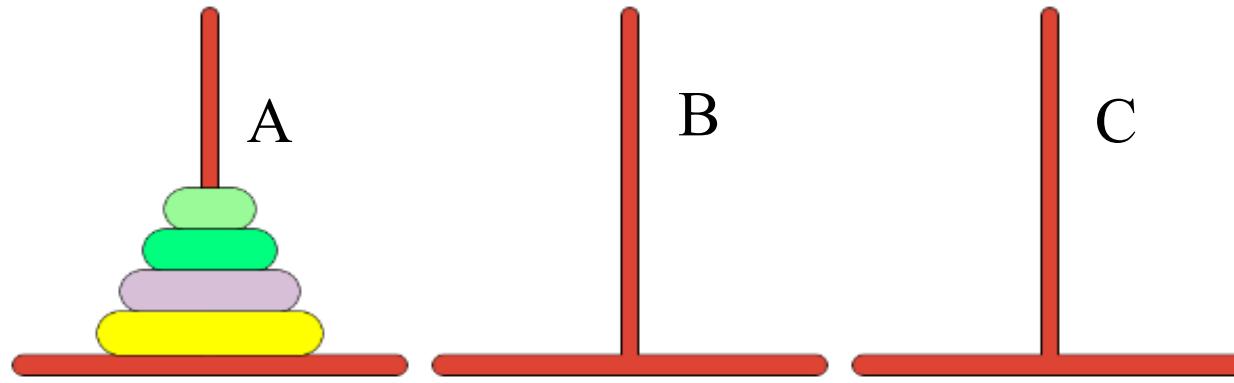
Tower of Hanoi



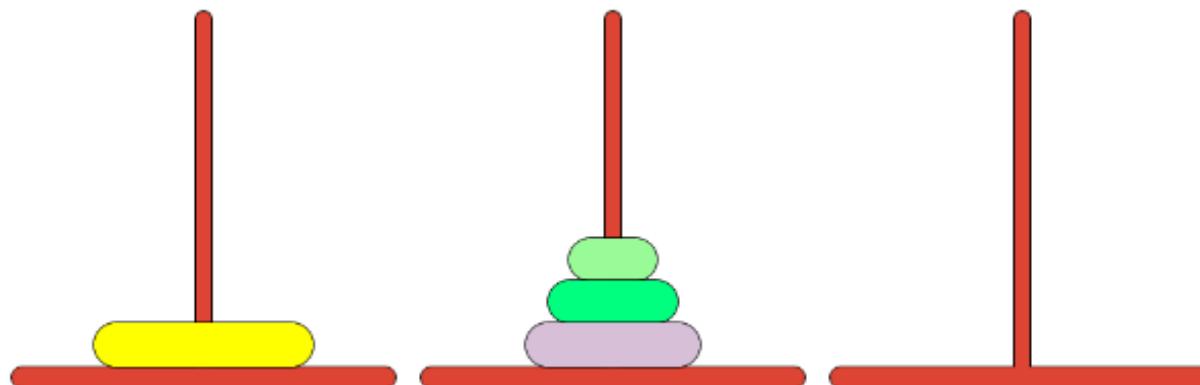
<https://www.mathsisfun.com/games/towerofhanoi.html>

- **Ending case:** when there is only one disk
- **Recursive case:** less disks need to be moved from a peg to another (which can be the smaller version of itself)

You Try 2. Tower of Hanoi. Minimum 15 moves is need to move four disks to C. Try and see in how many moves you can do it?

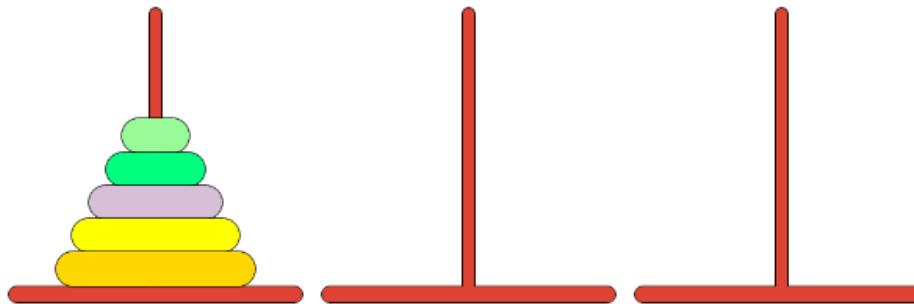


Did you reach to this configuration? What does this mean?



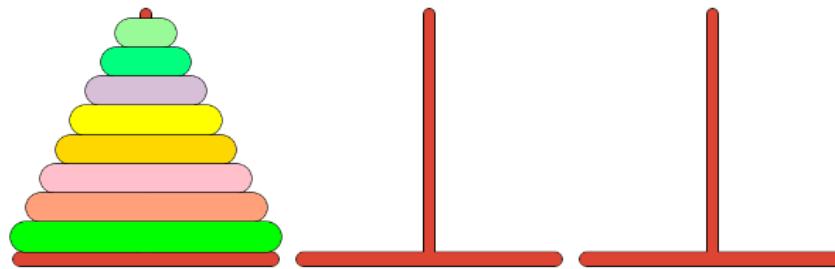
Tower of Hanoi

5 disks



31 moves

8 disks



255 moves

Next Lecture

We focus on:

- Algorithmic analysis and why it is needed
- Quantifying efficiency
- Concept of Big-O Notation

Readings/ Study Reference

Course Textbook: “*Data Structure and Algorithms in a Nutshell*” A. Wong et al.

Chapter 4. Recursion

Section 4.4. Designing Recursive Algorithms

Section 4.5. Conclusion

The End of Lecture

Course: Data Structures and Algorithms

Instructor: Homeyra Pourmohammadali

You Try Questions and Solutions
Designing Recursive Algorithms

You Try 1. Bottom-up Recursion Approach.

How the recursive design of previous example (sum of squares) would be different if you use the bottom-up approach?

Approach:

1. Evaluate the value you can calculate (for instance squaring the starting value of the call)
2. Add this value to a recursive call of a smaller version

You Try 1 Solution. Bottom-up Recursion Approach.

$$SS(a, b) = a^2 + SS(a + 1, b)$$

$$SS(a, b) = a^2 + SS(a + 1, b) + SS(a + 2, b)$$

$$SS(a, b) = \dots .$$

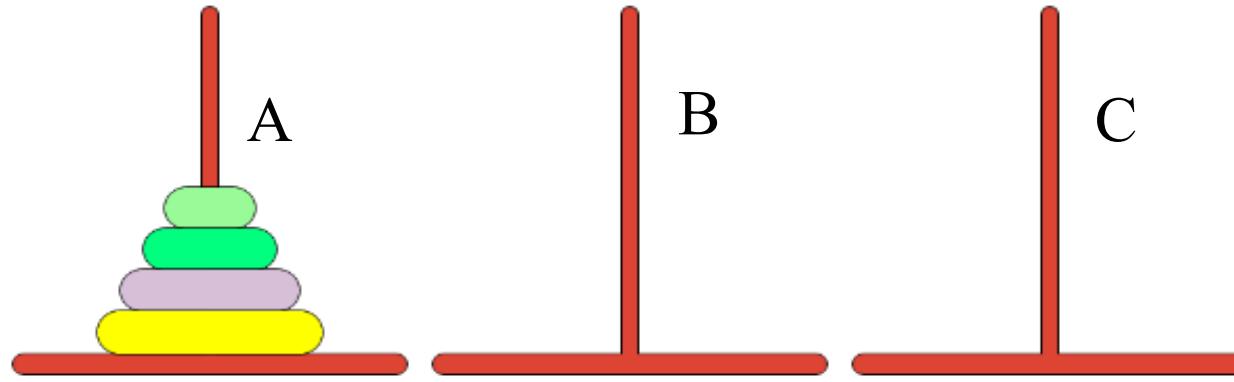
$$SS(a, b) = a^2 + SS(a + 1, b) + SS(a + 2, b) + \dots + SS(b, b)$$

$$SS(a, b) = a^2 + (a + 1)^2 + (a + 1)^3 + \dots + b^2$$

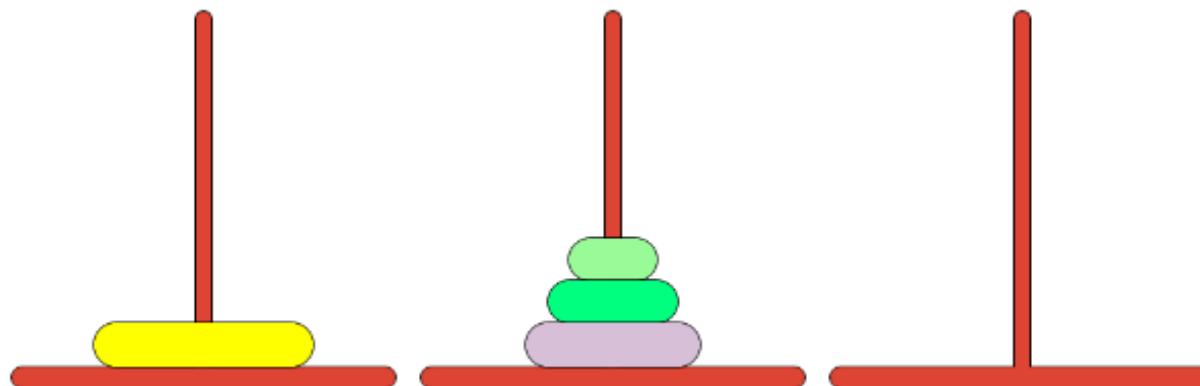
You Try 1 Solution. Bottom-up Recursion Approach.

```
int SS_BU(int a, int b)
{
    if (a==b) {
        return a*a;
    }
    else {
        return (a*a)+ SS_BU(a+1, b-1);
    }
}
```

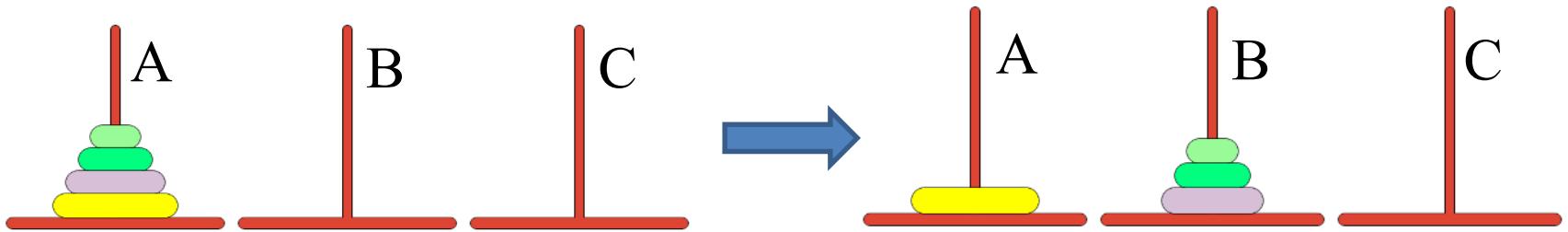
You Try 2. Tower of Hanoi. Minimum 15 moves is need to move four disks to C. Try and see in how many moves you can do it?



Did you reach to this configuration? What does this mean?



You Try 2 Solution. Tower of Hanoi.



The configuration in the right means that this problem (with $n=4$) contains a smaller version of itself (with $n=3$), but starting peg will change and move to the interim peg. Therefore, we need an extra input (e.g. interim) for the recursion function to address this shift. When the function is called the order of these pegs will change.

The End of You Try Activities

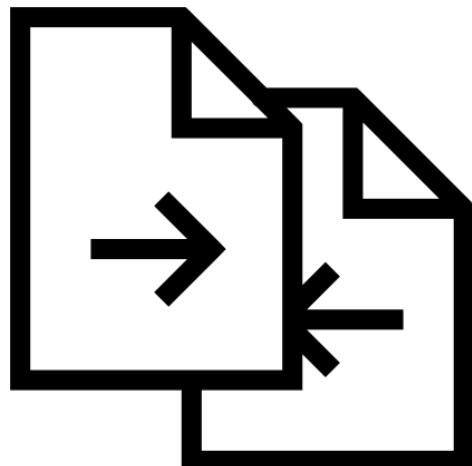
Course: Data Structures and Algorithms

Instructor: Homeyra Pourmohammadali

**Algorithmic Analysis, Quantifying Efficiency
and Big-O Notation**

Motivation

- Consider two algorithms used for sorting a list of items from smallest to largest
- How can we formally compare two sorting algorithms regardless of the size of the array, and what is the basis of comparison



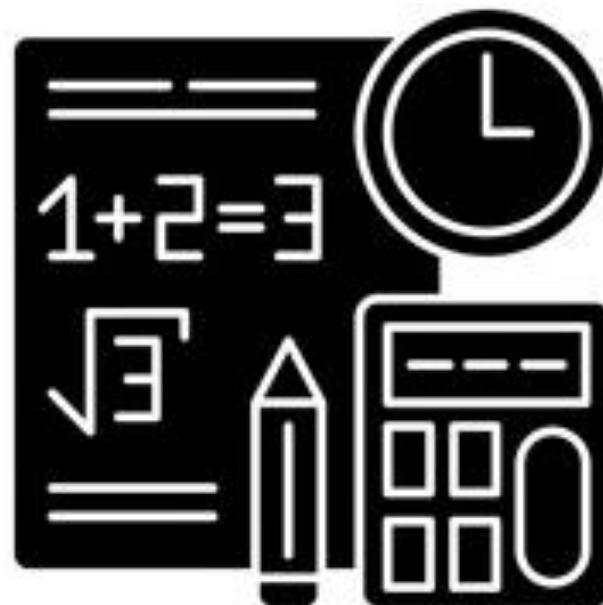
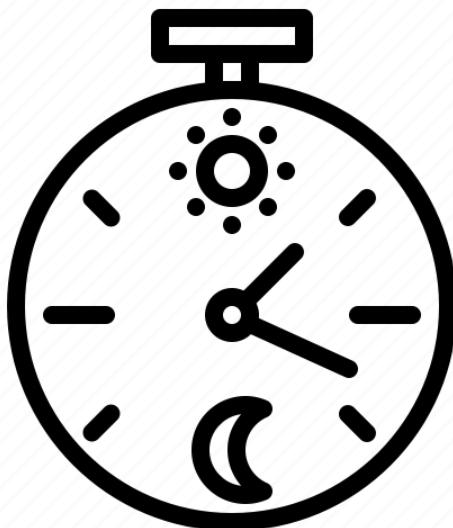
Motivation

- Can we calculate the efficiency of these algorithms before starting to code
- How can we quantify the efficiency of these algorithms and what methods are used for measuring it.



Motivation

- In algorithmic analysis we are concerned about the time that it takes for the program to end and space needed for the computational analysis.



Learning Outcomes

By the end of this lecture you will be able to find out about:

1. Algorithmic analysis and why it is needed
2. Quantifying efficiency
3. Algorithm growth rate and complexity classes
4. Concept of Big-O Notation

The Costs of a Computer Program

- Computing time and memory space
- Difficulties encountered by those who use the program
- Consequences of a program that does not work correctly
- Efforts needed to develop the solution
- Value of the time of people who work on the program
- Cost of maintenance, modification and expansion

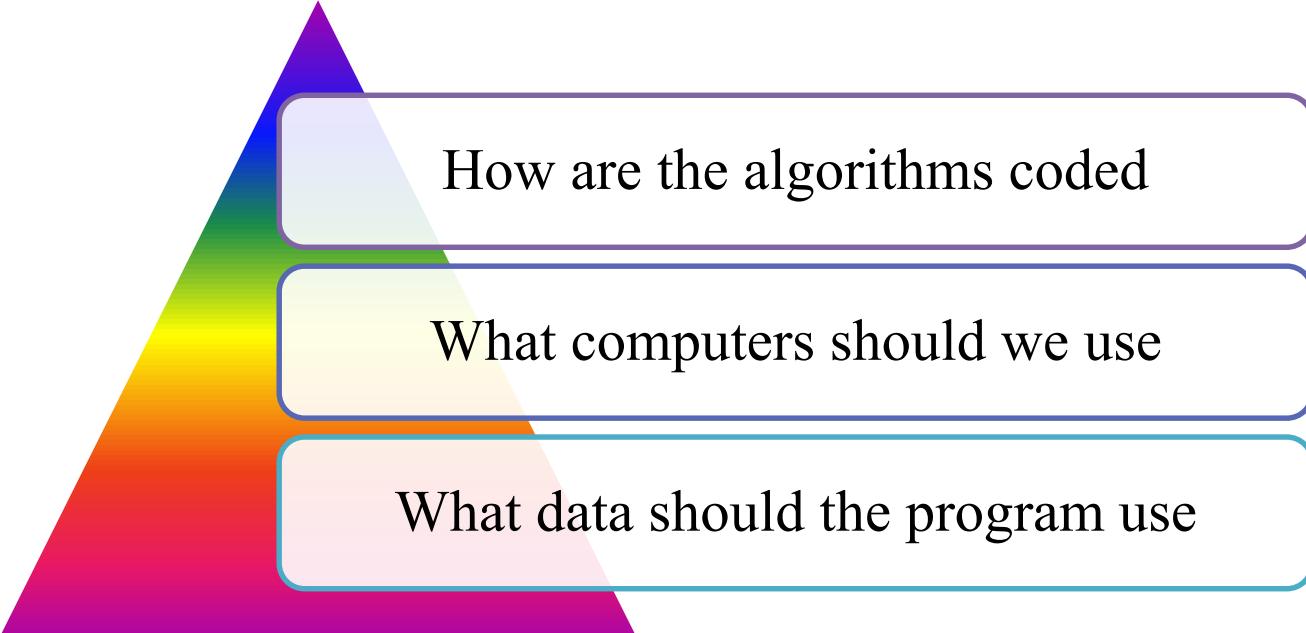
A computer program is good if the total cost it incurs over all phases of its life is minimal.

Quantifying Efficiency

- Efficiency criterion should be considered when selecting an algorithm and its implementation
- A comparison of algorithms should focus on significant differences in efficiency
- Algorithmic analysis provide tools for contrasting the efficiency of different algorithms
- We focus mainly on time efficiency

Comparing Programs instead of Algorithms

- How to compare time efficiency of two algorithms that solve the same problem
- Three difficulties comparing programs instead of algorithms:



How are the algorithms coded

What computers should we use

What data should the program use

Algorithmic Analysis

- This analysis should be independent of specific:
- Avoid analyzing an algorithm solely by studying the running times of a specific implementation.
- Running times are influenced by such factors as programming style, the particular computer, and the data on which the program is run.

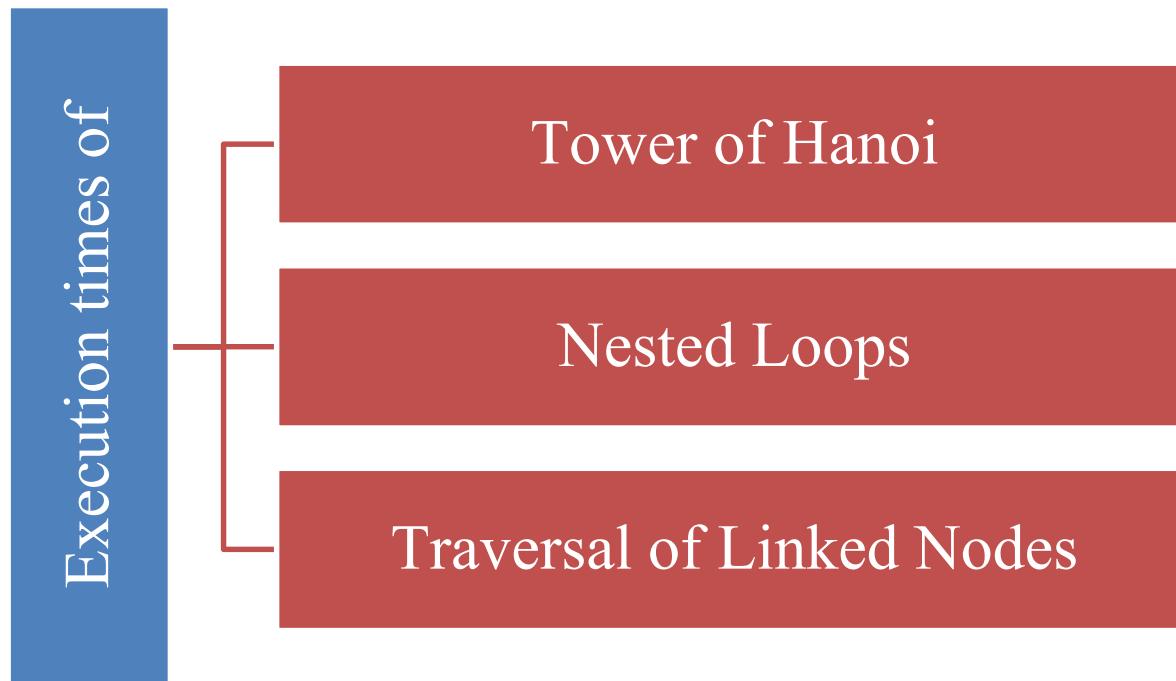
Implementations

Computers

Data

Execution Time of Algorithms

Look at three examples:



Execution Time of Algorithms

Tower of Hanoi: n disks need $2^n - 1$ moves

If each moves requires the same time m , the solution requires $(2^n - 1) * m$ time unit.

This time increases quickly as the number of disks increases.

If each move takes 0.5 sec for instance:

8 disks: needs 255 moves; takes 127.5 sec

100 disks: needs $1.2676506e+30$ moves; takes $6.338253e+29$ sec

Execution Time of Algorithms

Nested Loops: Consider this example:

```
for (i=1 through n)
    for (j=1 through i)
        for (k=1 through 5)
            Task T
```

If task *T* requires *t* time units, the innermost loop on *k* requires $5 * t$ time units.

The loop on *j* requires $5 * t * i$ time units, and the outermost loop on *i* requires $\sum_{i=1}^n 5 * t * i$ time units.

$$\sum_{t=1}^n 5 * t * i = 5 * t * (1 + 2 + 3 + \dots + n)$$

Execution Time of Algorithms

Traversal of Linked Nodes: Consider this example:

```
Node <ItemType>* curPtr=headPtr;           // 1 assignment
while (curPtr!=nullptr)                      // n+1 comparisons
{
    cout << curPtr->getItem()<endl;          // n writes
    curPtr=curPtr->getNext();                  // n assignments
}
```

- n nodes, statements require $n + 1$ assignments, $n + 1$ comparisons, and n write operations.

Execution Time of Algorithms

Traversal of Linked Nodes:

If each assignment, comparison, and write operation requires, respectively, a , c , and w time units, the statements require $(n + 1) * (a + c) + n * w$ time units.

Time required to write n nodes is proportional to n .

Algorithm Growth Rates

- Measure time requirement of an algorithm as a function of the problem size.
- Compare algorithm efficiencies for large problems
- When comparing the efficiency of various solutions, look only at significant differences.
- **Growth Rate:** proportional time requirement of an algorithm.

Big-O Notation

If an algorithm requires time proportional to $f(n)$, the algorithm is said to be **order $f(n)$** which is denoted by **$O(f(n))$** .

The function $f(n)$ is **growth-rate function** of an algorithm.

It is called **Big-O notation**, because the notation uses the capital letter O to denote **order**.

Definition of the Order of an Algorithm

- An algorithm is order $f(n)$ -denoted $O(f(n))$ - if constants k and n_0 exist such that the algorithm requires no more than $k*f(n)$ time units to solve a problem of size $n \geq n_0$.
- The requirement $n \geq n_0$ in the definition of $O(f(n))$ formalizes the notion of sufficiently large problems. In general, many values of k and n can satisfy the definition.

Order of an Algorithm

Example: Suppose an algorithm need $n^2 - 3n + 10$ sec to solve problem of size n.

If constant k and n_0 exist such that

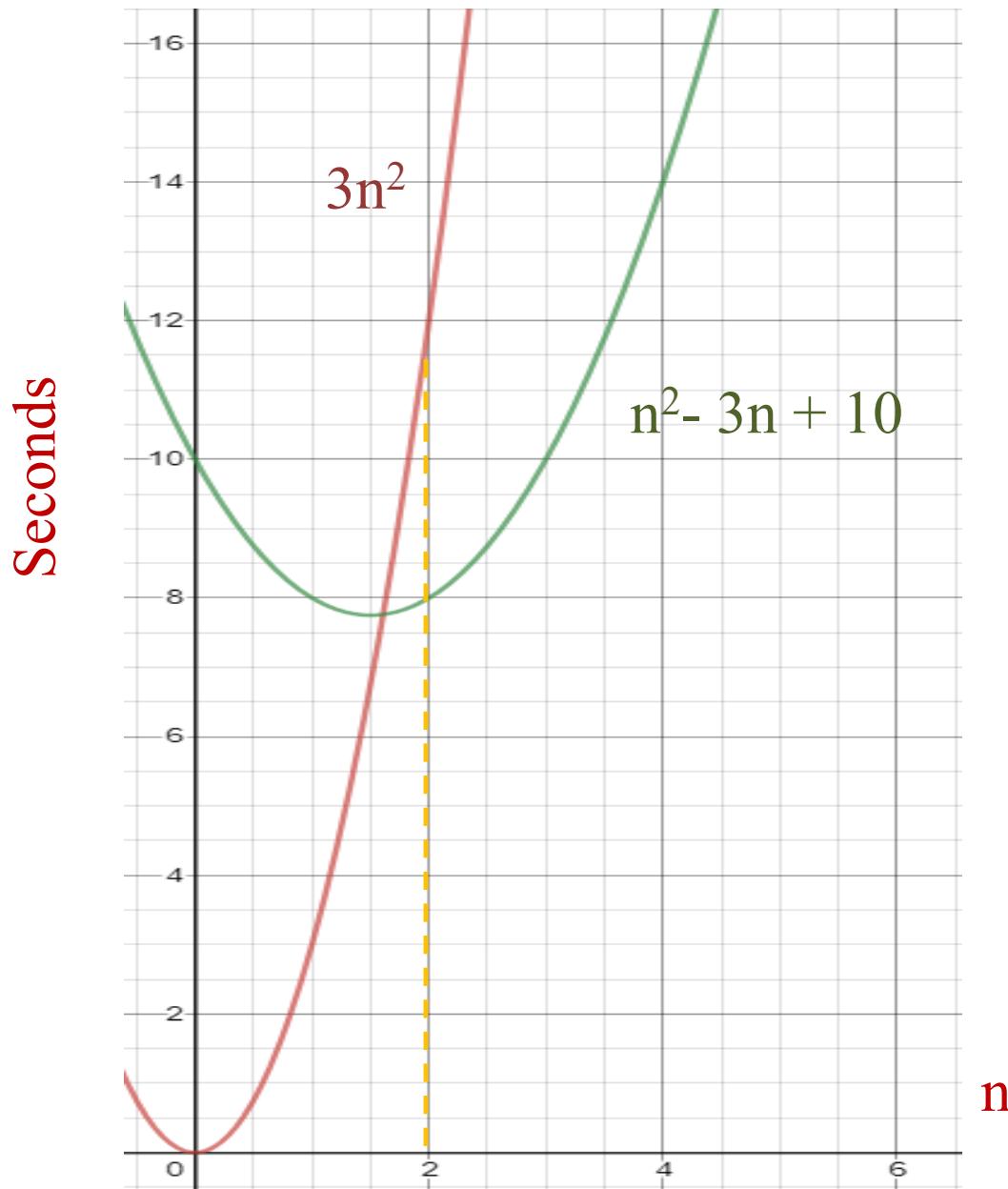
$$k * n^2 > n^2 - 3n + 10 \text{ for all } n \geq n_0$$

The algorithm is $O(n^2)$.

If $k = 3$ and $n_0 = 2$ then $3 * n^2 > n^2 - 3n + 10$ for all $n \geq 2$

Thus, algorithm required no more than $k * n^2$ time units for $n \geq n_0$ and so is $O(n^2)$.

Order of an Algorithm: Graphical



Complexity Classes

- We can classify different algorithms by considering how run time grows with respect to the size of the input.
- We can group algorithms with similar complexity together or contrast algorithms that fall into different complexity classes.

Complexity Classes



Complexity Class	Runtime Growth
$O(1)$	Constant time algorithms
$O(\log(n))$	Logarithmic time algorithms
$O(n)$	Linear time algorithms
$O(n * \log(n))$	$n \log n$ time algorithms
$O(n^2)$	Quadratic time algorithms
$O(n^3)$	Cubic time algorithms
$O(a^n)$	Exponential time algorithms

Time Complexity of Some Algorithms

Complexity Class	Examples of algorithms
$O(1)$	accessing an array element
$O(\log(n))$	Binary search
$O(n)$	Linear search
$O(n * \log(n))$	Quick sort, Merge sort
$O(n^2)$	Selection sort, Bubble sort
$O(n^3)$	Matrix multiplication
$O(2^n)$	Towers of Hanoi

Complexity & Efficiency

- Consider algorithm A requires time that is proportional to function f and algorithm B requires time that is proportional to a slower-growing function g .
- B is always significantly more efficient than A for large enough problems.
- For large problems, the proportional growth rate dominates all other factors in determining an algorithm's efficiency.

Next Lecture

We focus on:

- More on Big-O Notation & Examples

Readings/ Study Reference

Course Textbook: “*Data Structure and Algorithms in a Nutshell*” A. Wong et al.

Chapter 5. Algorithmic Analysis

Section 5.1. Why Do Algorithmic Analysis?

Section 5.2. Quantifying Efficiency

Section 5.3. Big-O Notation (up to 5.3.1)

The End of Lecture

Course: Data Structures and Algorithms

Instructor: Homeyra Pourmohammadali

**Algorithmic Analysis: Big-O Notation Examples
and Algorithm Runtime**

Learning Outcomes

By the end of this lecture you will be able to find out:

- what the properties of growth rate functions are
- how the Big-O notation definition can be used in different examples
- what the worst-, average- and best-case scenarios are
- how the runtime can be derived from the code using the knowledge about Big-O notation



Recall: Performance Analysis

- The performance measurement can be challenging and may be affected by the algorithm, compiler, programming language, and computer architecture, all of which impact program execution.
- Instead of analyzing the entire execution process, only the algorithm and the number of operations performed are classified.
- The efficiency of an algorithm before writing any code can be calculated to save time and cost.

Recall: Algorithmic Analysis

- Can formally analyze and compare two algorithms regardless of the input size
- Can express the **number of steps** an algorithm takes to run itself to completion (i.e., the number of steps taken)
- Can express this number **as a function $f(n)$** ; in $f(n)$, n is the input size
- Can compute $f(n)$ for each function/algorithm, and then compare that measurement against equivalent measurements for other functions/algorithms

Properties of Growth-rate Functions

Note: $O(f(n))$ means “is of order $f(n)$ ” or
“has order $f(n)$.” O is not a function.

You can ignore low-order terms in an algorithm’s growth-rate function

You can ignore a multiplicative constant in the high-order term of an algorithm’s growth-rate function.

$$O(f(n)) + O(g(n)) = O(f(n) + g(n))$$

1. Ignore Low-order Terms in Growth-rate Function

$f(n)$	10	1,000	100,000	1,000,000
1	1	1	1	1
$\log(n)$	3	9	16	19
n	10	10^3	10^5	10^6
$n \partial \log(n)$	30	9,965	10^6	10^7
n^2	10^2	10^6	10^{10}	10^{12}
n^3	10^3	10^9	10^{15}	10^{18}
2^n	10^3	10^{301}	$10^{30,103}$	$10^{301,030}$

Example: $O(n^3 + 4 \partial n^2 + 3 \partial n)$ would be $O(n^3)$.

2. Ignore Multiplicative Constants in High-order Term

Example: $O(5 \partial n^3) = O(n^3)$.

This observation follows from the definition of $O(f(n))$, if you let $k = 5$.

3. Can Combine Growth-rate Functions

$$O(f(n)) + O(g(n)) = O(f(n) + g(n))$$

Example: $O(n^2) + O(n) = O(n^2 + n) \rightarrow O(n^2)$
(by applying property 1).

Analogous rules hold for multiplication.

Simplification Rules for Big-O Notation

Logarithmic Functions

- Regardless of their base belong to the same $O(\log(n))$ logarithmic class of algorithms

Polynomial Functions

- (e.g., $ax^2 + bx + c$) where k is the largest degree belong to the same $O(n^k)$ class

Exponential Functions

- Belong to different classes depending on base (e.g., $O(2^n)$ & $O(3^n)$ are distinct).

Example. Big-O Notation. Simplification Rules

$$f(n) = n^3 + 2n^2 + \log(n)$$

$$f(n) = O(n^3)$$

$$f(n) = n^5 + 2^n + \log(\log(n))$$

$$f(n) = O(2^n)$$

Order of growth

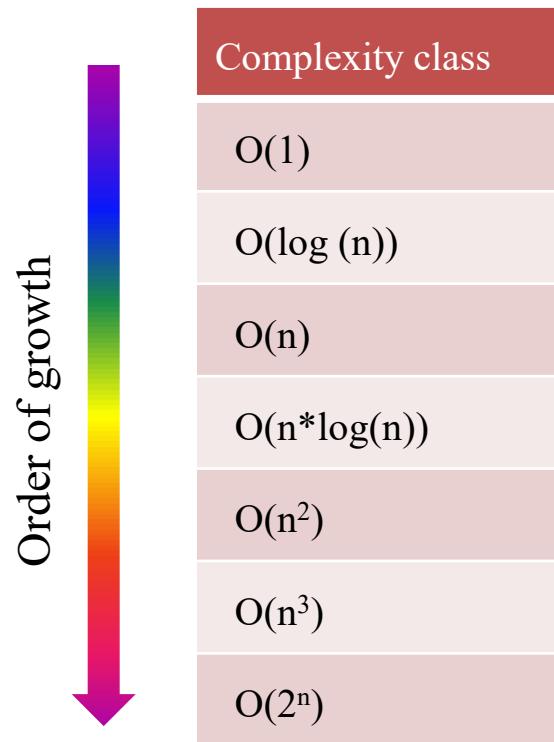
Complexity class
$O(1)$
$O(\log(n))$
$O(n)$
$O(n * \log(n))$
$O(n^2)$
$O(n^3)$
$O(2^n)$

You Try. Properties of Growth-rate Function



Find the order of the algorithms that have the following growth-rate functions:

- $O(8 \partial n^3 - 6 \partial n)$
- $O(3\partial \log_2 n + 30)$
- $O(7 \partial \log_2 n + n)$



Big-O Notation: Searching Algorithms

- For both `SequentialSearch` and `BinarySearch` functions (next slide), the size of the input n is the size of the array; K is the key value for which we are searching.
- For the two search algorithms, `SequentialSearch` takes roughly n operations to complete, if K is not found; hence, we classify it as linear time $O(n)$ algorithm.
- At the same time, `BinarySearch` takes less than n operations, and more specifically around $\log(n)$ operations, to complete if K is not found; hence, we classify it as logarithmic time $O(\log(n))$ algorithm.

```
int SequentialSearch(int A[], int size, int K) {  
    for (int i = 0; i < size; i++) {  
        if (A[i] == K)  
            return i;  
    }  
    return -1;  
}
```

Big-O Notation: Searching Algorithms

VS.

```
int BinarySearch(int A[], int L, int R, int K) {  
    // A must be already sorted for this to work  
    int mid = (L + R) / 2;  
    if (R < L) return -1;  
    else if (A[mid] == K)  
        return mid;  
    else if (K > A[mid])  
        return BinarySearch(A, mid + 1, R, K);  
    else  
        return BinarySearch(A, L, mid - 1, K);  
}
```

Big-O Notation Example

Which of the two search functions is faster given a sorted array as input?

- For small inputs, both search algorithms may finish in negligible amount of time
- As the input size grows, so does the time it takes for each algorithm to complete
- The growth rate of an algorithm will determine which algorithm performs faster for large inputs

Big-O Notation Example

To compare growth of two algorithms, **worst-case** scenario for each is considered

- Performance of an algorithm is expressed by assigning it to its own category
- Big-O notation is used to express performance information about an algorithm
- Using the Big-O notation, SequentialSearch is **O(n)** and BinarySearch is **O(log(n))**:

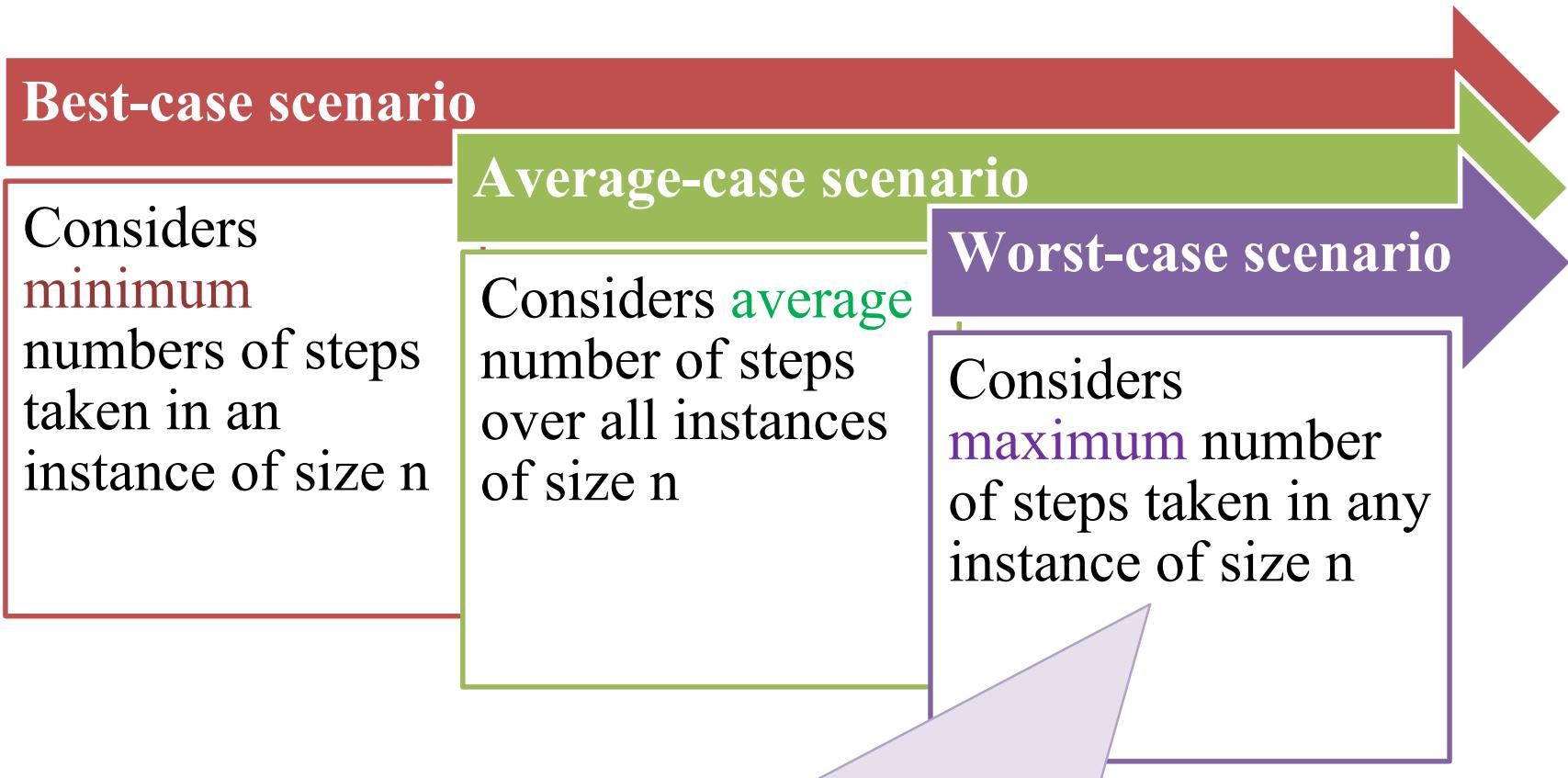
$$O(\log(n)) < O(n)$$

Why do we Care about Algorithmic Complexity?

- See the sample comparison chart below; the exact run-times depend on the hardware platform
- Even the quadratic time algorithms may become impractical when the input size is in multiple millions

n	$f(n)$	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10		0.003 μ s	0.01 μ s	0.033 μ s	0.1 μ s	1 μ s	3.63 ms
20		0.004 μ s	0.02 μ s	0.086 μ s	0.4 μ s	1 ms	77.1 years
30		0.005 μ s	0.03 μ s	0.147 μ s	0.9 μ s	1 sec	8.4×10^{15} yrs
40		0.005 μ s	0.04 μ s	0.213 μ s	1.6 μ s	18.3 min	
50		0.006 μ s	0.05 μ s	0.282 μ s	2.5 μ s	13 days	
100		0.007 μ s	0.1 μ s	0.644 μ s	10 μ s	4×10^{13} yrs	
1,000		0.010 μ s	1.00 μ s	9.966 μ s	1 ms		
10,000		0.013 μ s	10 μ s	130 μ s	100 ms		
100,000		0.017 μ s	0.10 ms	1.67 ms	10 sec		
1,000,000		0.020 μ s	1 ms	19.93 ms	16.7 min		
10,000,000		0.023 μ s	0.01 sec	0.23 sec	1.16 days		
100,000,000		0.027 μ s	0.10 sec	2.66 sec	115.7 days		
1,000,000,000		0.030 μ s	1 sec	29.90 sec	31.7 years		

Best-, Average-, and Worst-case Scenarios



Big-O runtime of an algorithm, is a way of describing its **worst-case** runtime

Big-Omega and Big-Theta Notations

Big-Omega (Ω) Notation

- describes the **lower-bound** runtime of a given algorithm
 - A function $f(n)$ is classified as $\Omega(g(n))$ if there exist two positive constants K and n_0 such that:
 - $|f(n)| \geq K|g(n)|$ for all $n \geq n_0$

Big-Theta (Θ) Notation

- describes the **tight-bound** runtime of a given algorithm
 - A function $f(n)$ is classified as $\Theta(g(n))$ if there exist positive constants K_1 , K_2 , and n_0 such that:
 - $|f(n)| \leq K_1|g(n)|$ and $|f(n)| \geq K_2|g(n)|$ for all $n \geq n_0$

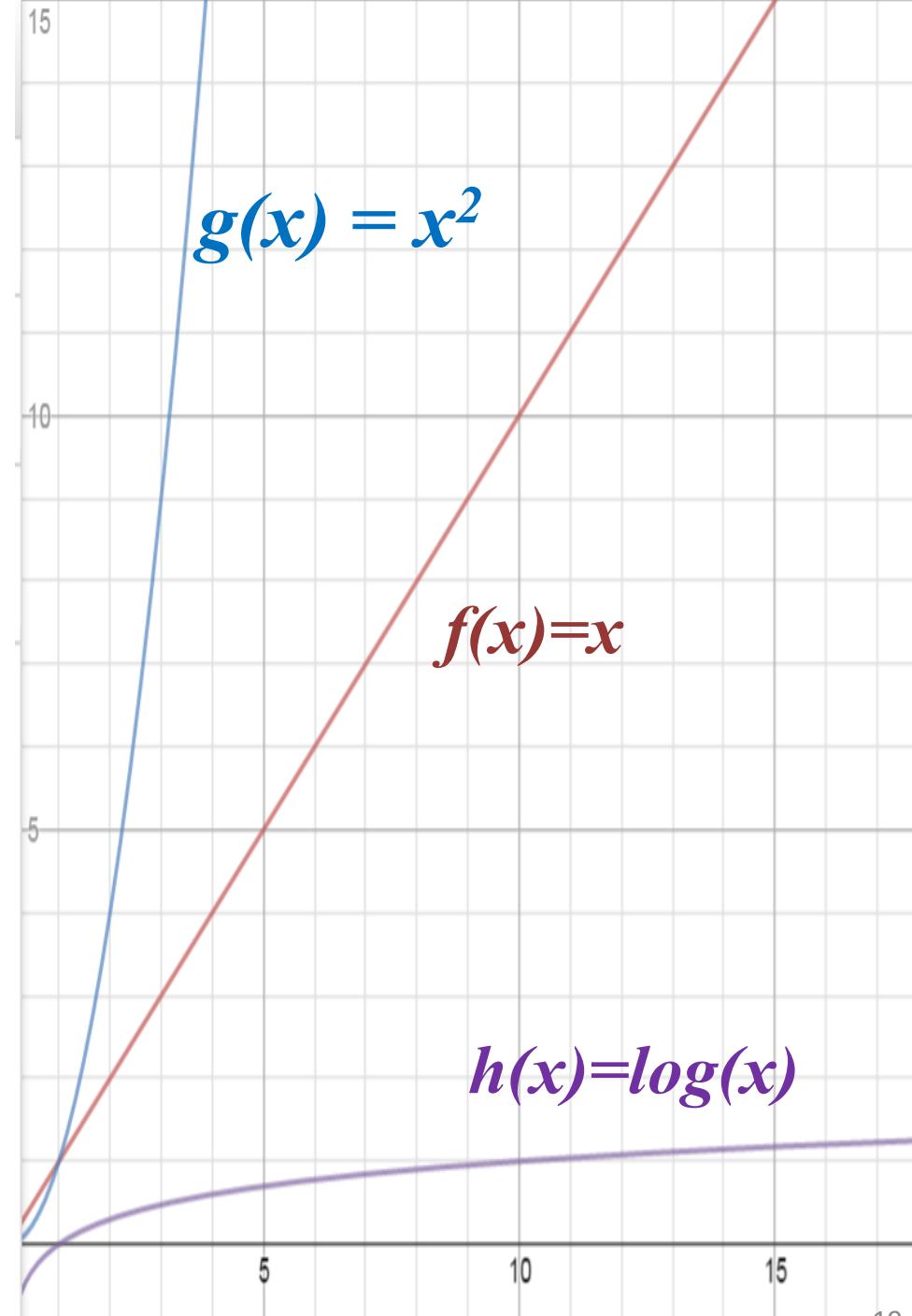
Big-O Notation

Function $f(n)$ represents the number of operations for an algorithm (growth-rate function) and n is the size of input

- A function $f(n)$ is classified as $O(g(n))$ if there exist two positive constants K and n_0 such that $|f(n)| \leq K|g(n)|$ for all $n \geq n_0$
- Visually, there exists a positive constant K for which $K|g(n)|$ is above $f(n)$ for all $n \geq n_0$ (see next slide)

Visual Demonstration of Big-O Notation

- Let $f(x) = x$, $g(x) = x^2$,
and $h(x) = \log(x)$
- Then, $f(x) = O(g(x))$
since $g(x)$ is above $f(x)$
- Similarly, $h(x) = O(g(x))$
and $h(x) = O(f(x))$



Example 1. Big-O Notation: Find constant K and n_0

Let $f(n) = 100n^2$ and $g(n) = n^2$, Show that $f(n) = O(g(n))$

Solution: This needs to be satisfied: $f(n) \leq K\partial g(n)$ for all $n \geq n_0$

Steps:

1) Select $n_0 = 1$

$$\text{for } n \geq n_0 \rightarrow 100 n^2 \leq K \partial n^2 \rightarrow 100 \leq K$$

2) Hence, for $K \geq 100$ and $n \geq 1 \rightarrow f(n) = O(g(n))$

Example 2. Big-O Notation: Find constant K and n_0

Let $f(n) = n^2$ and $g(n) = n^3$, show that $f(n) = O(g(n))$

Solution:

Steps:

$$1) \text{ for } n \geq n_0 \rightarrow n^2 \leq K \partial n^3 \rightarrow 1 \leq K \partial n \rightarrow \frac{1}{n} \leq K$$

$$2) \text{ Given } n_0 \geq 1$$

$$\text{for } K \geq 1, \frac{1}{n} \leq K \text{ holds}$$

$$3) \text{ Hence, for } K \geq 1 \text{ and } n \geq 1, f(n) = O(g(n))$$

You Try. Big-O Notation: Find constant K and n_0

Let $f(n) = n^2$ and $g(n) = 2^n$ show that $f(n) = O(g(n))$



Ulternative Approach For Comparing Function Growth

Another approach to comparing function growth is to

compute $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$

- If the limit is 0, then $f(n)$ grows slower than $g(n)$; that is, $f(n) = O(g(n))$ (e.g., n / n^2)
- If the limit is a constant c , then $f(n)$ grows as fast as $g(n)$; that is, $f(n) = O(g(n))$ (e.g., $2n / n$)
- If the limit is infinity ∞ , then $f(n)$ grows faster than $g(n)$; that is, $g(n) = O(f(n))$ (e.g., n^2 / n)

Example: Let $f(n) = n^2$ and $g(n) = 2^n$

Show that $f(n) = O(g(n))$

- Steps:

$$\begin{aligned} \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} &= \lim_{n \rightarrow \infty} \frac{n^2}{2^n} \\ &= \lim_{n \rightarrow \infty} \frac{2n}{\ln(2) 2^n} \text{ (using L'Hospital's Rule)} \\ &= \lim_{n \rightarrow \infty} \frac{2}{\ln(2)^2 2^n} \text{ (using L'Hospital's Rule)} \\ &= \frac{2}{\ln(2)^2} \lim_{n \rightarrow \infty} \frac{1}{2^n} = 0 \end{aligned}$$

- Hence, $f(n) = O(g(n))$

Summary

- Only significant differences in growth rate functions are meaningful.
- Worst-case analysis considers the maximum amount of work an algorithm requires on a problem of a given size, while average-case analysis considers the expected amount of work that it requires.
- Analyzing algorithm's time requirement is helpful in choosing an ADT implementation. If an application frequently uses particular ADT operations, the implementation should be efficient for at least those operations.

Next Lecture

We focus on:

- Analysis of recursive algorithms

Readings/ Study Reference

Course Textbook: “*Data Structure and Algorithms in a Nutshell*” A. Wong et al.

Chapter 5. Algorithmic Analysis

Section 5.3 Big-O Notation

Section 5.4 Algorithm Runtime

The End of Lecture

Course: Data Structures and Algorithms

Instructor: Homeyra Pourmohammadali

You Try Questions and Solutions

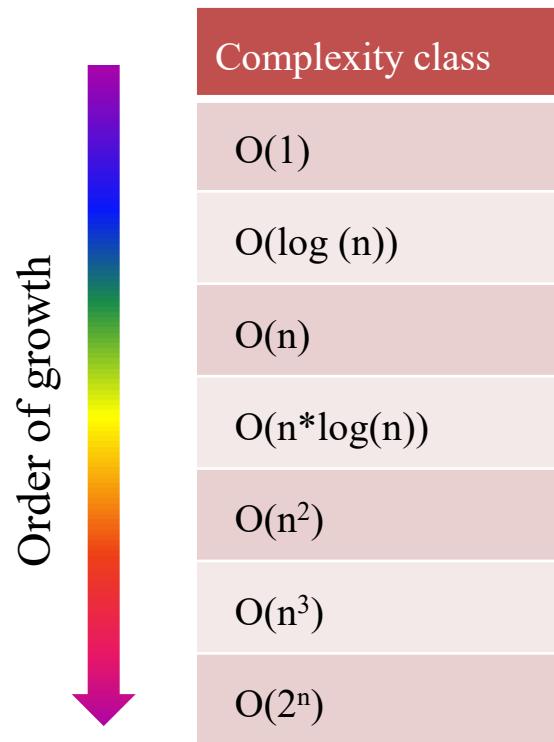
**Algorithmic Analysis: Big-O Notation Examples
and Algorithm Runtime**

You Try. Properties of Growth-rate Function



Find the order of the algorithms that have the following growth-rate functions:

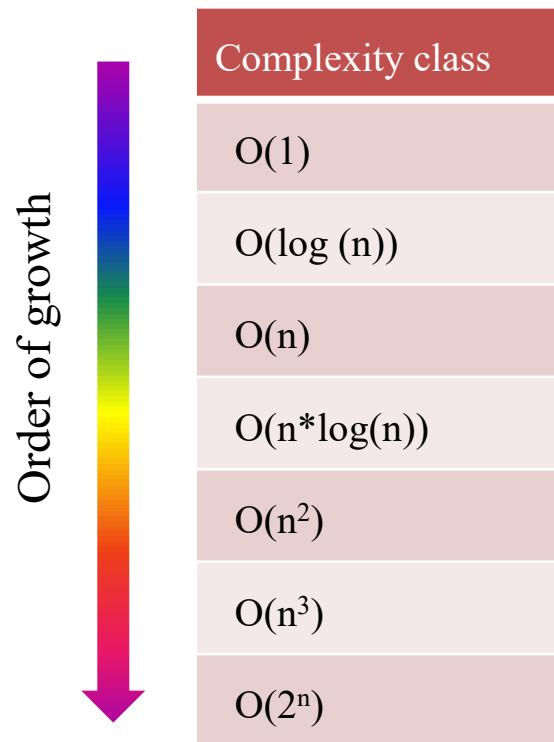
- $O(8 * n^3 - 6 * n)$
- $O(3 * \log_2 n + 30)$
- $O(7 * \log_2 n + n)$



You Try Solution. Properties of Growth-rate Function

Find the order of the algorithms that have the following growth-rate functions:

- $O(8 * n^3 - 6 * n) = O(n^3)$
- $O(3 * \log_2 n + 30) = O(\log_2 n)$
- $O(7 * \log_2 n + n) = O(n)$



You Try. Big-O Notation: Find constant K and n_0

Let $f(n) = n^2$ and $g(n) = 2^n$, show that $f(n) = O(g(n))$



You Try Solution. Big-O Notation: Find constant K and n_0

Let $f(n) = n^2$ and $g(n) = 2^n$ show that $f(n) = O(g(n))$

Steps:

1) For $n \geq n_0 \rightarrow n^2 \leq K * 2^n \rightarrow \frac{n^2}{2^n} \leq K \rightarrow \frac{n^2}{2^n} \leq K$

2) Set $K=2$ and find n_0 that satisfy Big-O requirements

$$n^2 \leq 2 * 2^n, \text{ then } n^2 \leq 2^{n+1}$$

3) Base cases: 1,2,3 are true, prove $(n+1)^2 \leq 2^{(n+1)+1}$ is true

$$n^2 + 2n + 1 \leq 2 * 2^{(n+1)}, n^2 \text{ grows less quickly than } 2^n$$

$$n^2 + 2n + 1 \leq 2 * n^2 \quad \text{then: } 2n + 1 \leq n^2 \text{ true for } n \geq 3$$

The End of You Try Activity

Course: Data Structures and Algorithms

Instructor: Homeyra Pourmohammadali

Analysis of Recursive Algorithms

Learning Outcomes

By the end of this lecture you will be able to find out:

- what the difference is between the Big-O computation of iterative and recursive algorithms
- how this computation can be addressed through examples

Introduction

A typical computer program needs to be:

Complete

- It should “do everything” needed.

Correct

- It should “do it right.”

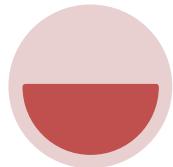
Usable

- Its user interface should be “easy to work with.”

Efficient

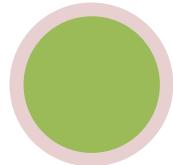
- It should finish in a “reasonable amount of time” considering the complexity and size of the task.

Iterative versus Recursive



Iterative

There is a loop
A block of code is repeated
Larger size of code



Recursive

An entity calls itself
The same function is called again
Smaller size of code

How about the time complexity of these two algorithms?

Big-O Computation

- Consider number of executions that the code will perform in the worst-case scenario.
- The strategy for Big-O computation depends on whether or not your program is recursive.
- For the iterative solutions, we try and count the number of executions that are performed.
- For the recursive solutions, we first try and compute the number of recursive calls that are performed.

How to Compute Big-O From Source Code for Iterative Algorithms?

- Express each loop as a summation/sigma, (Σ)
- For segments of code that are repeated on each iteration, use a constant (e.g., a)
- For nested loops, use nested summations (e.g. $\Sigma \Sigma$)
- Finally, use summation formulas for simplification

Example 1. Calculate Big-O of an Iterative Algorithm

Code Example	Time requirements	Big-O
<pre>for (int x = n; x >= 0; x--) { cout << x << endl; }</pre>	The loop executes n times	$O(n)$
<pre>for (int i = 0; i < 5; i++) { for (int j = 0; j < 10; j++) { cout << j << endl; } }</pre>	None of the <code>for</code> loops depends on the value n	$O(1)$
<pre>for (int i = 0; i < n; i++) { for (int j = 0; j < n; j++) { cout << j << endl; } }</pre>	The outer loop executes n times and each iteration, the inner loop executes n times, totaling in n^2 loops	$O(n^2)$

Example 1 Continued. Calculate Big-O of an Iterative Algorithm

Code Example	Time Requirements	Big-O
<pre>for (int i = 0; i < 5n; i++) { cout << "hello!" << endl; }</pre>	The loop executes $5n$ times and the constant 5 is insignificant as n grows	$O(n)$
<pre>for (int i = 0; i < n; i++) { for (int j = 0; j < n*n; j++) { cout << "tricky!" << endl; } }</pre>	The outer loop executes n times and each iteration, the inner loop executes n^2 times, totaling in n^3 executions	$O(n^3)$

Summations

When you have a code block which executes **1** time, then **2** times, then **3** times until **n** times.

Sum of an arithmetic series:

$$1 + 2 + 3 + \dots + (n - 2) + (n - 1) + n = \frac{n(n+1)}{2} = \frac{1}{2}n^2 + \frac{1}{2}n$$

Then Big-O notation is **O(n²)**.

Example 2. Calculate Big-O of an Iterative Algorithm

Three nested loops:

for $i = 0$ to $n - 2$ do {

for $j = i + 1$ to $n - 1$ do {

for $k = i$ to n do {

 // constant steps

}

}

}

$$f(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} \sum_{k=i}^n a =$$

$$a \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} (n - i + 1) = a \sum_{i=0}^{n-2} (n - i - 1)(n - i + 1) =$$

$$a((n + 1)(n - 1) + n(n - 2) + \dots + 3 * 1) =$$

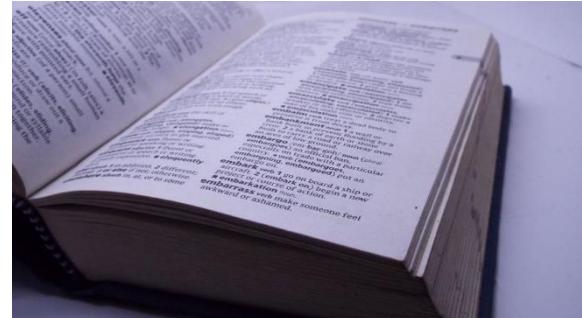
$$a \sum_{j=1}^{n-1} (j + 2)j = a \sum_{j=1}^{n-1} j^2 + a \sum_{j=1}^{n-1} 2j =$$

$$a \frac{(n - 1)n(2n - 1)}{6} + 2a \frac{(n - 1)n}{2} =$$

$$a \frac{n(n - 1)(2n + 5)}{6} = a \left(\frac{1}{3}n^3 + \frac{1}{2}n^2 - \frac{5}{6}n \right) = O(n^3)$$

Recursive Algorithms

- Recursion breaks a problem into smaller identical problems
- Consider a **binary search** algorithm and think about searching a word in a dictionary



- You can first **divide** the dictionary into **two halves** and then **conquer** the appropriate half.
- The dividing continues until you reach the **base case** (when only one page left).

How to Compute Big-O From Source Code for Recursive Algorithms?

- Define the **base cases** and the **recursive case**
- Use **backwards substitution** to go from the recursive case down to the base cases
- Use **summation** and other formulas for **simplification**

Example 3. Calculate Big-O of Recursive Algorithm

```
int BinarySearch(int A[], int L, int R, int K) {  
    // A must be already sorted for this to work  
  
    int mid = (L + R) / 2;  
    if (R < L)  
        return -1;  
  
    else if (A[mid] == K)  
        return mid;  
    else if (K > A[mid])  
        return BinarySearch(A, mid + 1, R, K);  
    else  
        return BinarySearch(A, L, mid - 1, K);  
}
```

Example 3 Continued. Calculate Big-O of Recursive Algorithm

Let $n = R - L + 1$, then

$$T(1) = a$$

$$T(n) = b + T(n/2)$$

$$T(n) = b + b + T(n/4)$$

$$T(n) = b + b + b + T(n/8)$$

$$T(n) = b + b + b + b + T(n/16)$$

$$T(n) = \dots$$

$$T(n) = ib + T(n/2^i)$$

Example 3 Continued. Calculate Big-O of Recursive Algorithm

When $(n / 2^i) = 1$, let $i = c$

It follows that $(n / 2^c) = 1$, and

$n = 2^c$, so $c = \log_2(n)$

$T(n) = cb + T(n/2^c)$, replace: $(n / 2^c) = 1$

$T(n) = cb + T(1)$, replace: $T(1) = a$, $c = \log_2(n)$

$T(n) = b \log_2(n) + a$

$T(n) = O(\log_2(n))$

You Try. Compute Big-O of Fibonacci Sequence



$F_n = 0$ for $n=0$ and $F_n = 1$ for $n=1$

$F_n = F_{n-1} + F_{n-2}$ for $n > 1$

Consider $T(n)$ for time requirement of $F(n)$. What is $T(0)$, $T(1)$, and $T(n)$?

Hint: you can make an assumption, simplify, and then use backward substitution.

Optionally, you can use call tree of this sequence and visually understand how time requirement is increasing with n .

Next Lecture

We focus on:

- Stack Overall Structure
- Stack Applications

Readings/ Study Reference

Course Textbook: “*Data Structure and Algorithms in a Nutshell*” A. Wong et al.

Chapter 5. Algorithmic Analysis

Section 5.4 Algorithm Runtime

Section 5.5 Analysis of Recursive Algorithms

The End of Lecture

Course: Data Structures and Algorithms

Instructor: Homeyra Pourmohammadali

**Stacks and Queues ADT
Structures, Operations and Applications**

Motivation

How your **web browser** handles previously visited webpages?

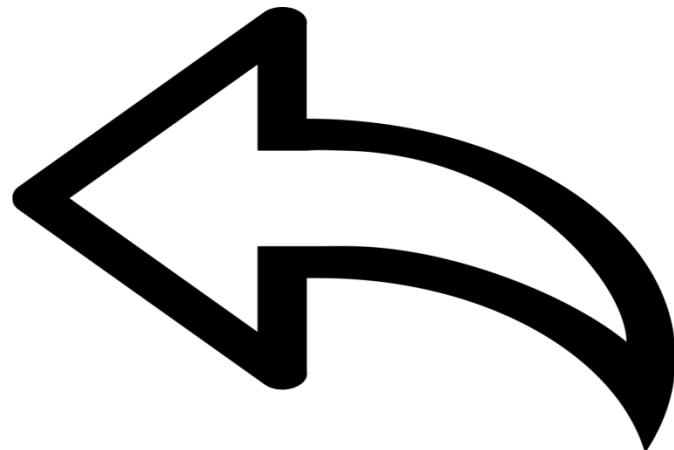
It makes sense to have a **stack** containing your visited web pages, that when you want to go to a new web page, allows the web address of the previous web page to be **pushed** into it.



Motivation

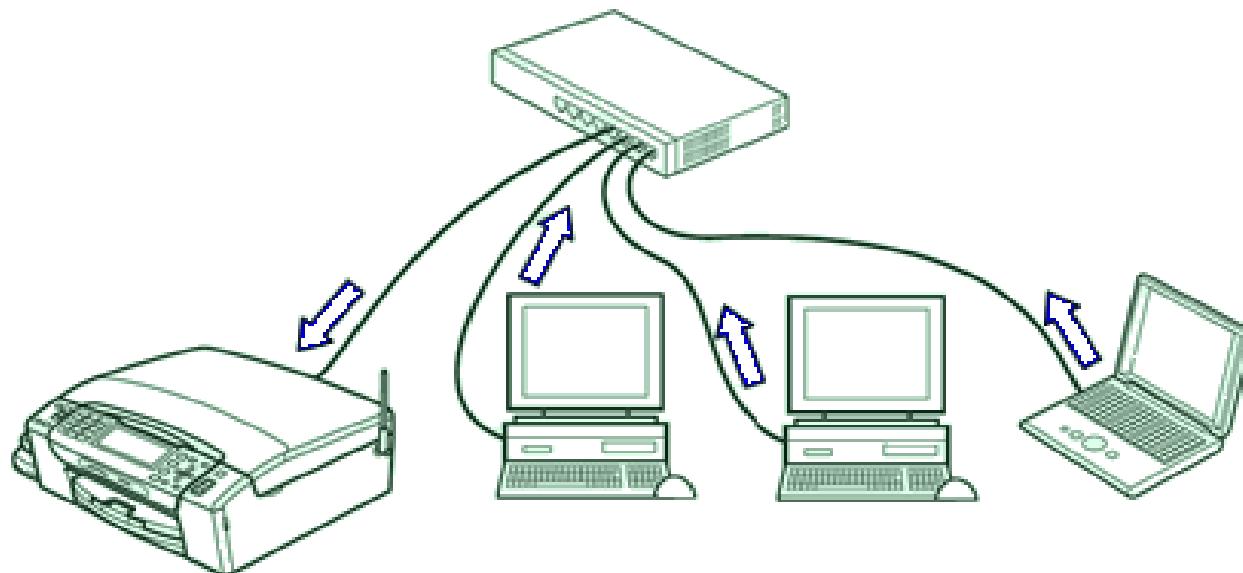
What is really involved when you press **back button**?

It makes sense that the data structure allows the previous web address to **pop**, so you can navigate through the most-recently visited web-page.



Motivation

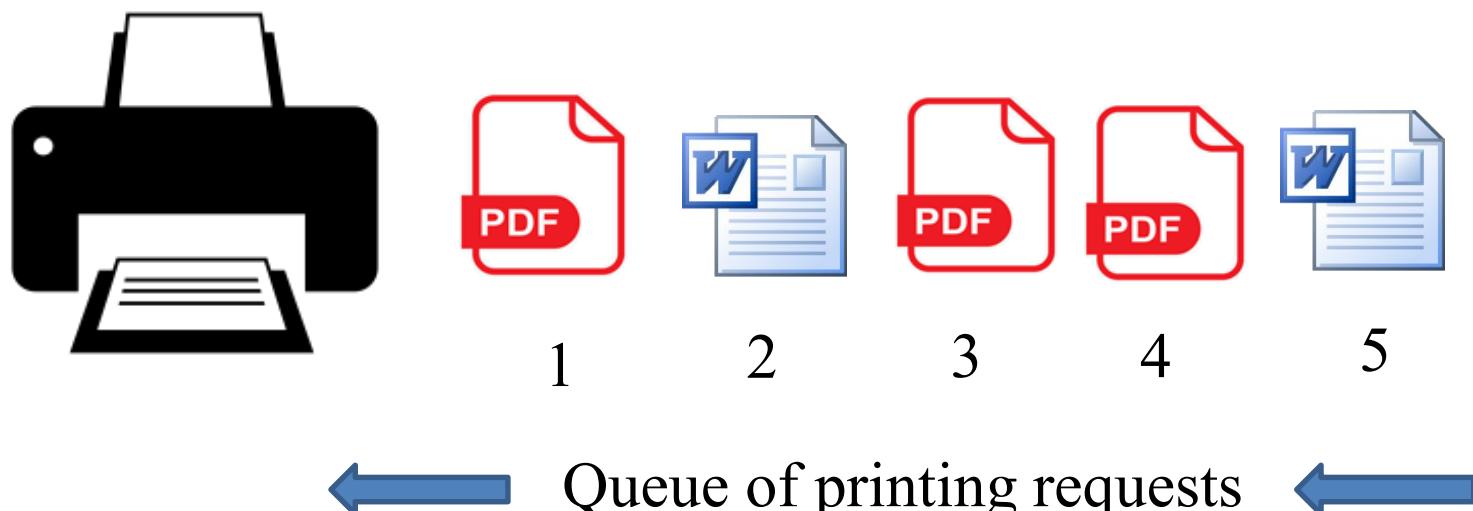
- How your **printer** handles multiple print requests?
- What is the mechanism that figures out which documents to print first and handles the rest of documents that need to be printed?



https://support.brother.com/g/b/faqend.aspx?c=us&lang=en&prod=fax5750e_us&faqid=faq00002756_000

Motivation

- The document is **enqueued** into a **queue** containing documents to print, when a new request is received by printer.
- Printer **dequeues** a document from the printing queue, when it is ready to print a new document.



Learning Outcomes

By the end of this lecture you will be able to:

- describe two most commonly used and fundamental ADTs called stacks and queues
- demonstrate their overall structure, main operations and interfaces
- list their common applications

Real-life Examples

Stacks



Collection of dinner plates
(vertical collections)

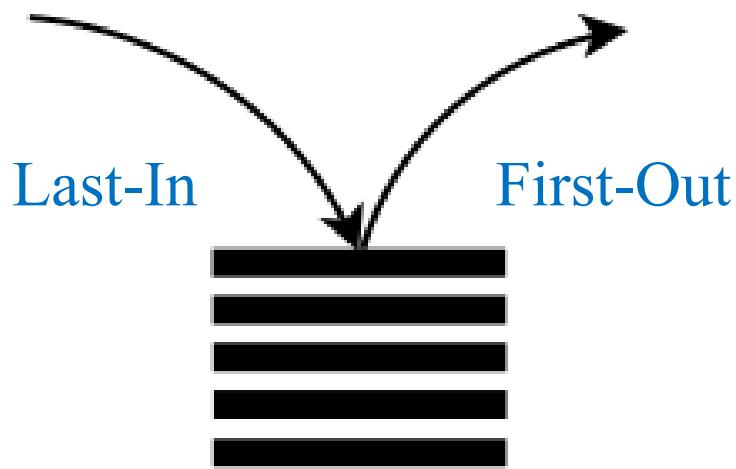


People standing in a line for
ticket (horizontal collection)

Working Principles

Stacks

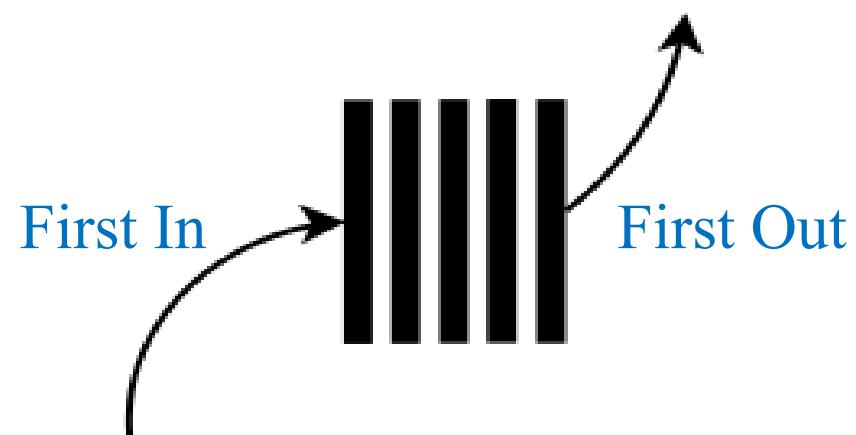
Last-In/First-Out (LIFO)
data structure.



Objects are inserted and removed at the same end.

Queues

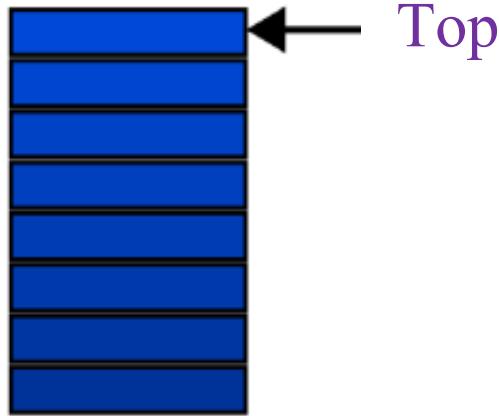
First In/ First Out (FIFO)
data structure.



Objects are inserted and removed from different ends

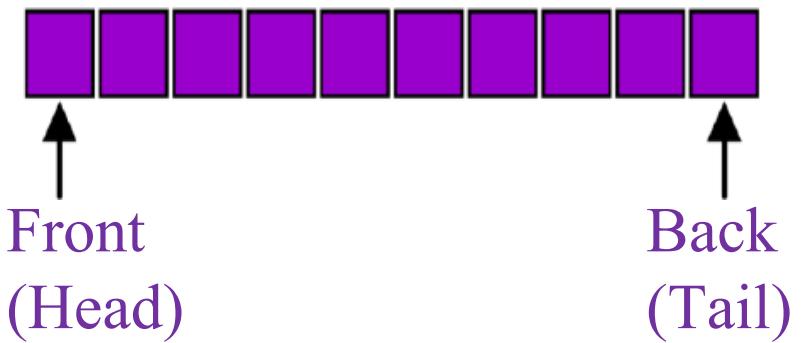
Index

Stacks



In stacks, one end is open and only the index of the top element is needed.

Queues



In queues, both ends are open and the indexes of both front and back ends are needed.

Main Operations

Stacks

Push is the insert operation.

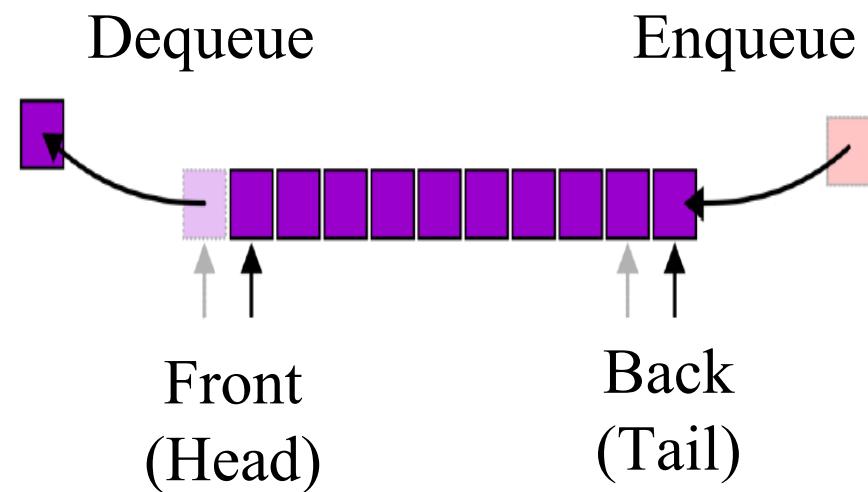
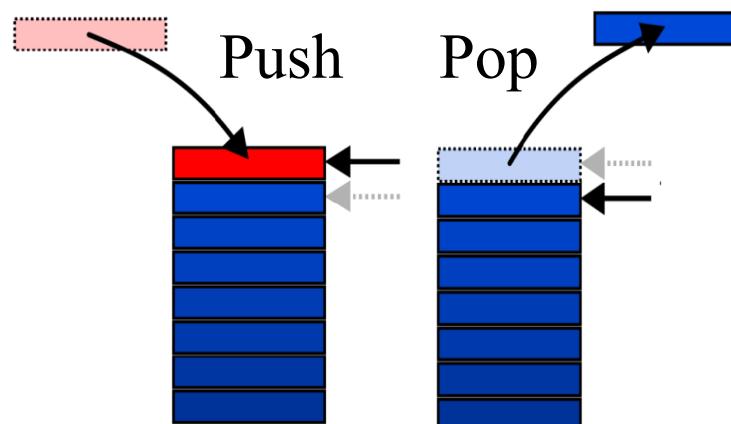
Pop is the delete operation.

Queues

Enqueue is the insert operation.

Dequeue is the delete operation.

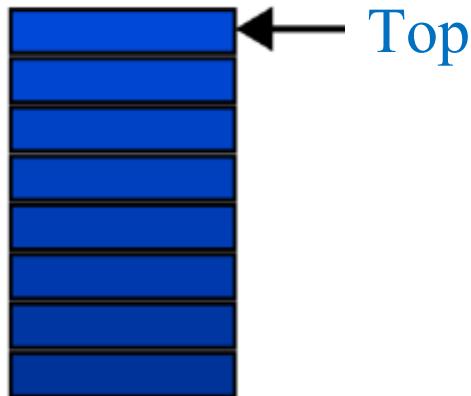
Time complexity of all: $O(1)$



Another Operation

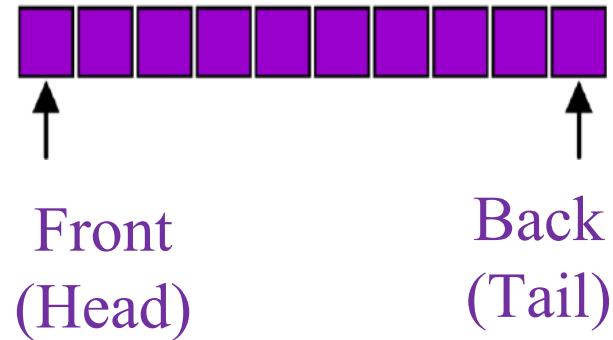
Stacks

Peek: peek the top item without removing it from stack.



Queues

Peek: peek the front item without removing it from the queue.



Undefined Operations

Stacks

1. Push on a full stack
(overflow)
2. Pop or peek an empty
stack (underflow)

Queues

1. Enqueue a full queue
(overflow)
2. Dequeue or peek an
empty queue (underflow)

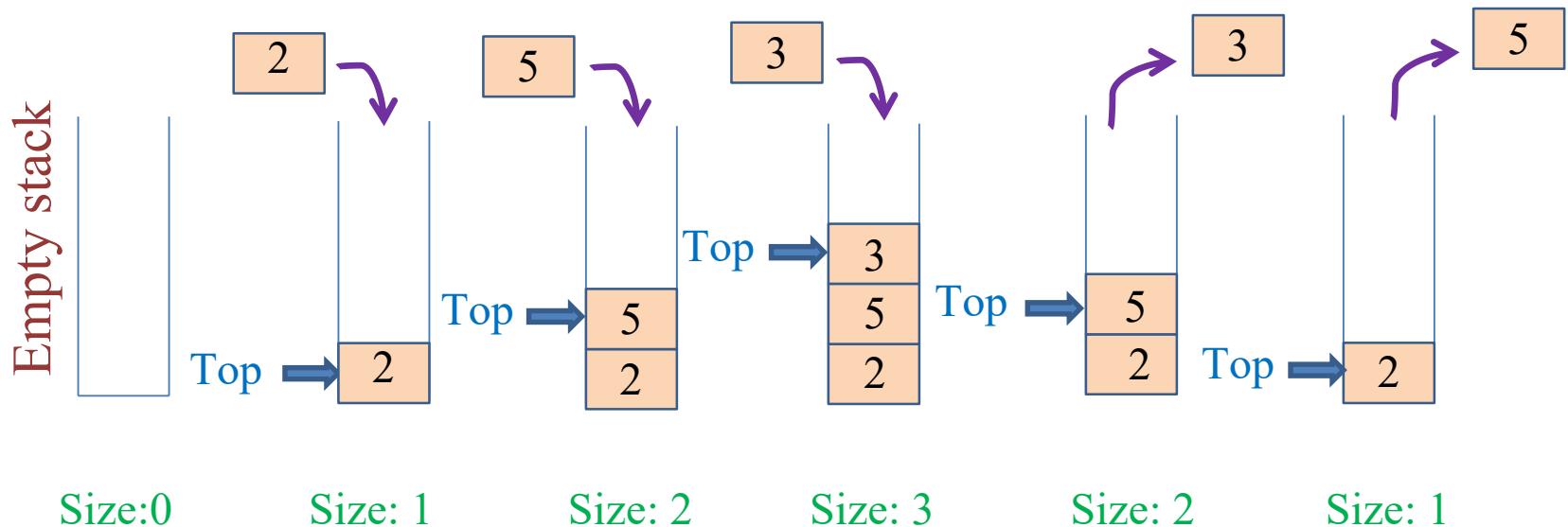
Operations Inputs and Outputs

Tasks	Stacks	Queues
Insert element	push(newEntry) boolean	enqueue(newEntry) boolean
Remove element	pop() boolean	dequeue() boolean
Peek top or front element	peek() Item Type	peek() Item Type
Check if it is empty	isEmpty() boolean	isEmpty() boolean

Example 1. Main Operations in Stack

Assume a stack is empty, show the result of following operations:

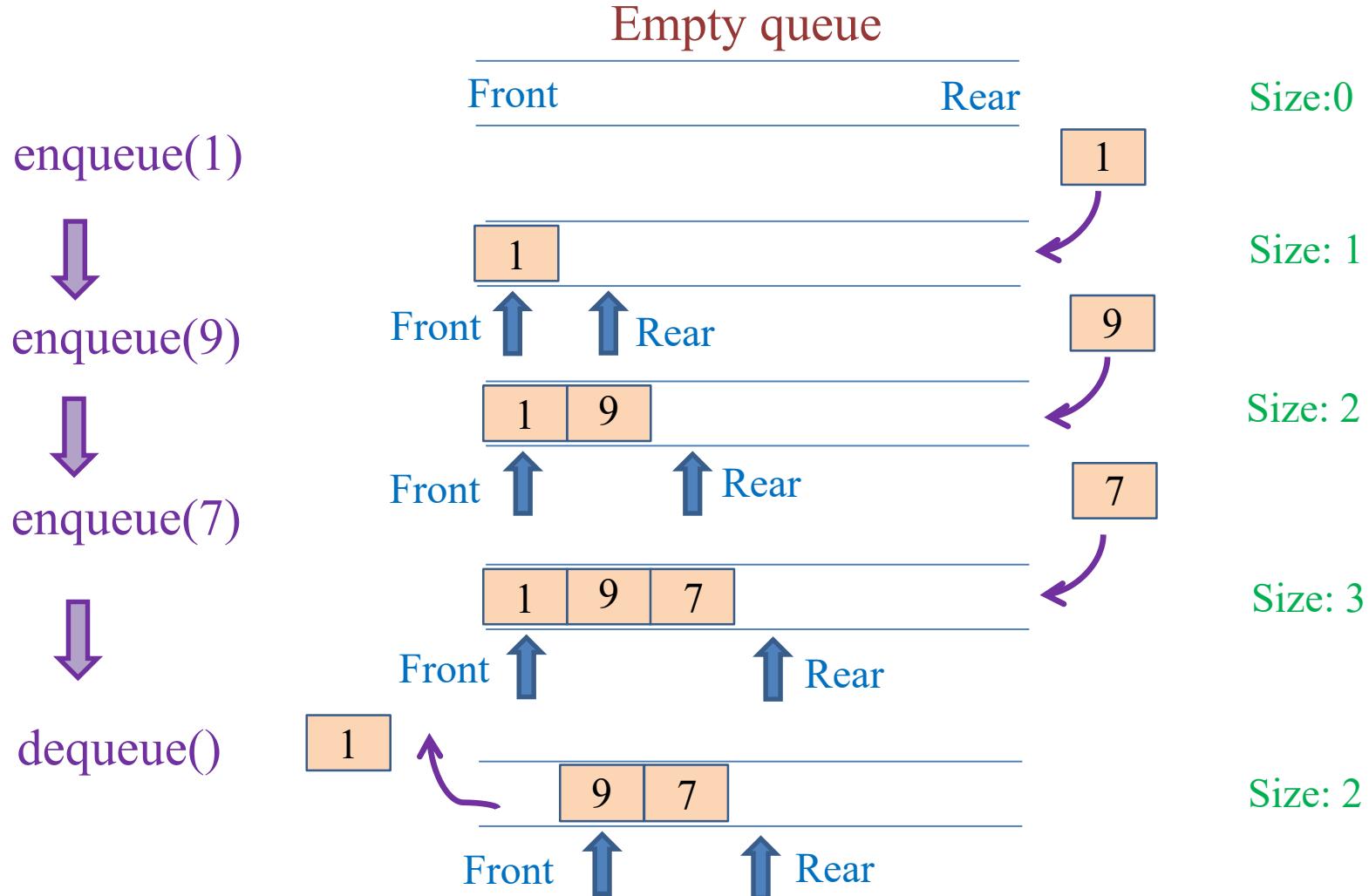
push(2) \Rightarrow push(5) \Rightarrow push(3) \Rightarrow pop() \Rightarrow pop()



Note that one element can be inserted or removed at a time.

Example 2. Main Operations in Queue.

Assume a queue is empty, show the result of following operations:



You Try 1. Main Operations in Stack



Show the result of operations on the given full stack.

push(W) \Rightarrow pop() \Rightarrow pop() \Rightarrow push(P) \Rightarrow pop() \Rightarrow pop()

D
R
C
E
A



Size:

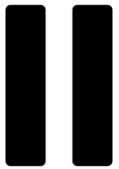
Size:

Size:

Size:

Size:

Size:



You Try 2. Main Operations in Queue.

Show the result of operations on the given full queue.

enqueue(U)



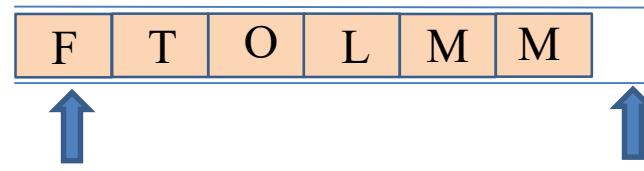
dequeue()



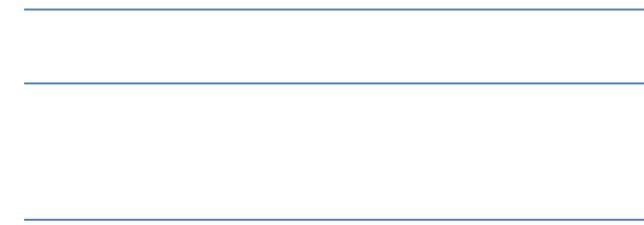
dequeue()



enqueue(U)



Size:



Size:



Size:



Size:

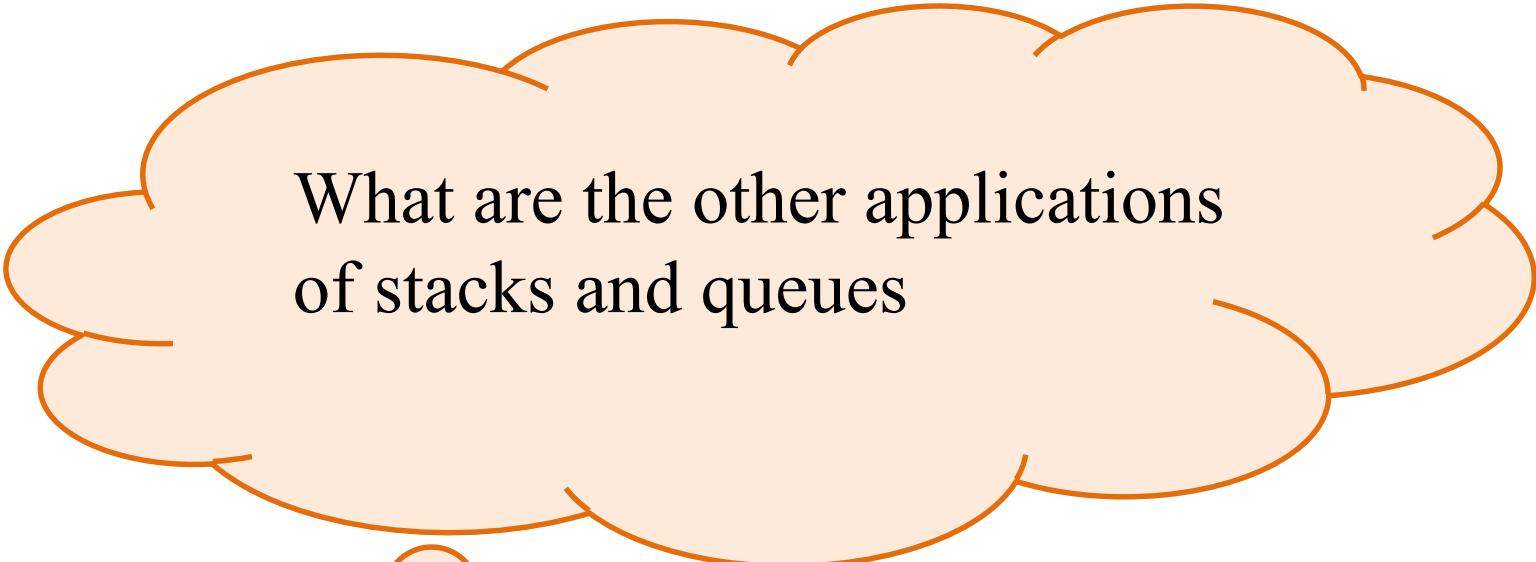
Some Applications

Stacks

1. Tracking undo and redo operations in applications (going forward and back in a web browser)
2. Parsing code, including HTML, matching parentheses in C++

Queues

1. Print queue: sending jobs to the same printer
2. Web servers, databases, mail servers



What are the other applications
of stacks and queues



Stack ADT Interface

```
class Stack {          // public interface for Stack ADT
    ...
public:
    typedef int StackItem; // integer stack item
    Stack();   // default constructor
    ~Stack();  // default destructor

    bool push(StackItem sItem); // push item into stack

    StackItem pop();          // pop item from stack and return
                              // reference to top item of stack
    StackItem peek(); // return reference to top item of stack
};
```

Queue ADT Interface

```
class Queue {          // public interface for Queue ADT
    ...
public:
    typedef int QueueItem;      // integer queue item
    Queue();                  // default constructor
    ~Queue();                 // default destructor

    bool enqueue(QueueItem qItem); //enqueue item into queue

    QueueItem dequeue(); //dequeue item from queue and return
                         // a reference to front item of queue

    QueueItem peek(); //return a reference to front item of queue
};
```

Implementation Methods

Stacks

Based on the **LIFO** principle.

Linked
Representation

- (e.g. linked lists)

Sequential
Representation

- (e.g. arrays)

Queues

Based on **FIFO** principle.

Linked
Representation

- (e.g. linked lists)

Sequential
Representation

- (e.g. arrays)

Next Lecture

We focus on:

- Implementation of stacks using sequential lists
- Implementation of stacks using linked lists

Readings/ Study Reference

Course Textbook: “*Data Structure and Algorithms in a Nutshell*” A. Wong et al.

Chapter 6. Stacks and Queue

Section 6.1 Stacks (Subsection 6.1.1)

Section 6.2 Queues (Subsection 6.2.1)

The End of Lecture

Course: Data Structures and Algorithms

Instructor: Homeyra Pourmohammadali

**Stacks ADT Application:
Check for Balanced Braces**

Learning Outcomes

By the end of this part of lecture you will be able to:

- use push and pop functions
- use stack ADT for the application of “brace balance check”

Example 1. Push and Pop

Syntax: stackname.push(newEntry)

Input: mystack // a newly created stack is empty
mystack.push(9); // insert 9 in mystack
mystack.push(7); // insert 7 in mystack (put it on top of 9)

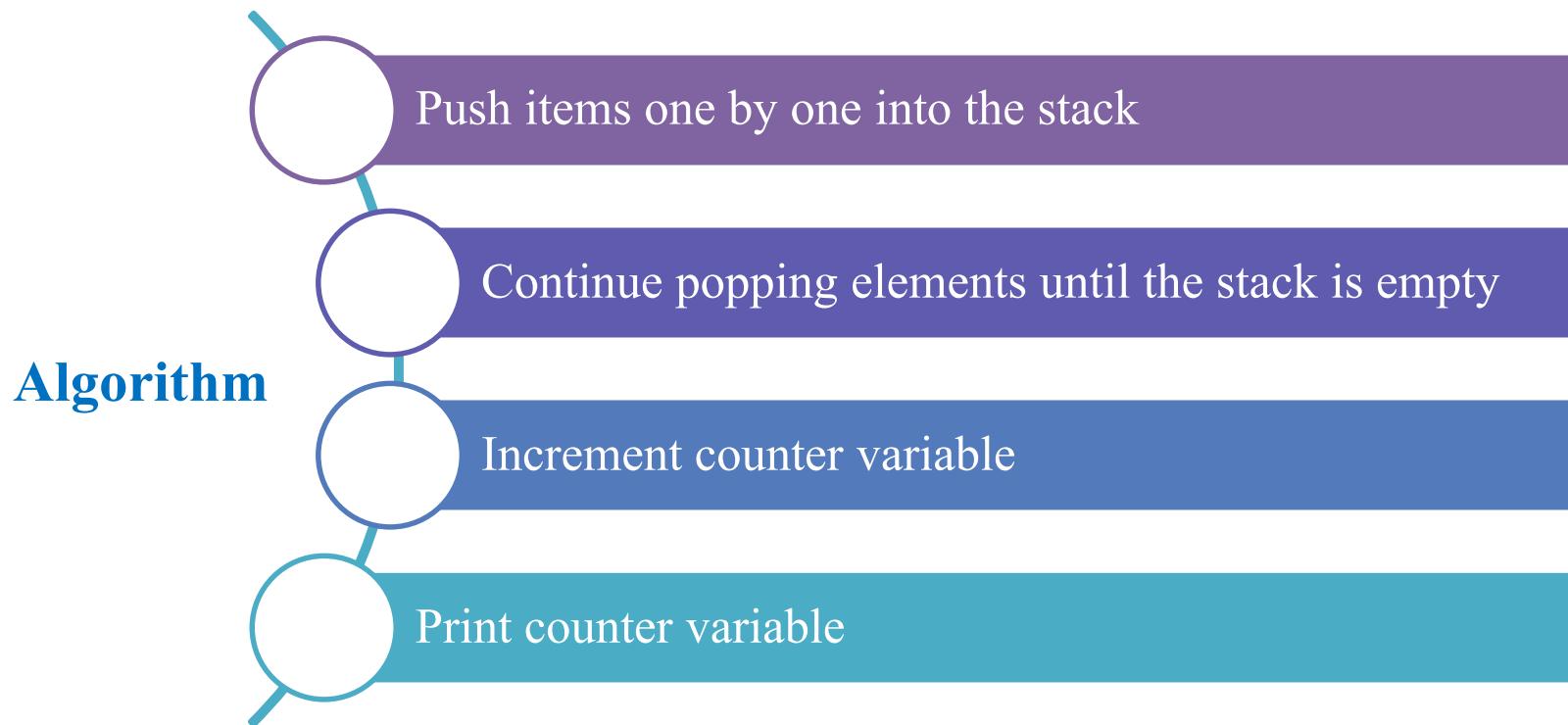
Output: 9, 7

Syntax: stackname.pop()

Input: mystack= 8,5,1,3,6 // mystack has 5 element (last one is on top)
mystack.pop(); // remove top element,6, from the stack
mystack.pop(); // remove new top element, 3

Output: 8, 5, 1

Example 2. Insert five integers into the stack and find the size of the stack without using “size” function.



```
#include <iostream>
#include <stack>
using namespace std;

int main()
{
    int b = 0;
    stack<int> mystack; // Create an empty stack
    mystack.push(11);
    mystack.push(5);
    mystack.push(9);
    mystack.push(9);
    mystack.push(1); // Stack contains these element 11, 5, 9, 9, 1
    // Counting number of elements
    while (!mystack.empty()) { // Counting number of elements
        mystack.pop();
        b++;
    }
    cout << b;
}
```

Example 2. Solution

Application of Stack ADT: Check for Balanced Braces

Problem Statement

Given an expression in form of a string, containing braces, check if the opening braces “{“ are balanced with the closing braces”}”.

Braces: a b c { d e f g { i j k } { l { m n } } o p }q r

Note: other versions of problem include parenthesis or square brackets instead of braces, or combination of all.

Parenthesis: a b c (d) e f g j k) (l (m (n) o p (q r

Brackets: a b c [d [e f g j k]] l] m n [o p q r

Combination: a b c [d (e f g j k) l { m n { o p q r]

Requirements for Balanced Braces

The braces are balanced if:

- 1 {
 - Each time you encounter a closing brace, “}”, it matches an already encountered opening brace “{”.
- 2 {
 - When you reach the end of the string, you have matched each “{”.

You need to keep track of each unmatched “{” and discard one each time you encounter a “}”.

Simple Pseudocode

One way to perform this task is to **push** each “{” encountered onto a **stack** and **pop** one off each time you encounter a “}”.

Simple pseudocode solution:

```
for ( each character in the string )  
{  
    if ( the character is a '{')  
        aStack.push('{')           // aStack is the name of our stack  
  
    else if ( the character is a '}')  
        aStack.pop()  
}
```

Detailed Pseudocode

```
// Checks the string aString to verify that braces match.  
// Returns true if aString contains matching braces, false otherwise.
```

```
checkBraces(aString: string): boolean
```

```
aStack = a new empty stack
```

```
balancedSoFar = true
```

```
i = 0
```

```
while (balancedSoFar and i < length of aString)
```

```
{
```

```
ch = character at position i in aString
```

```
i++
```

Continued next slide

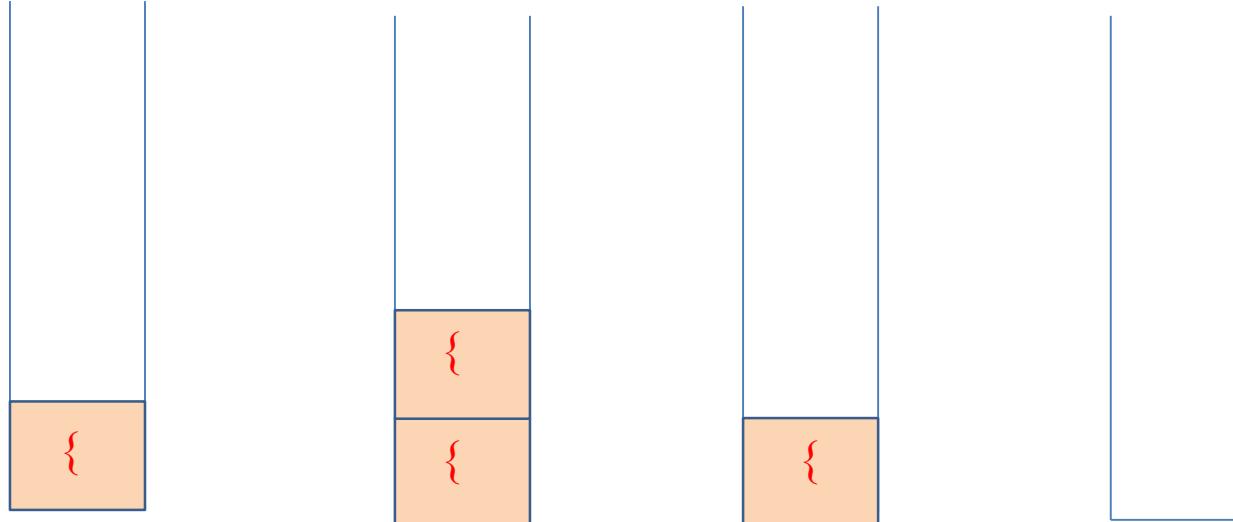
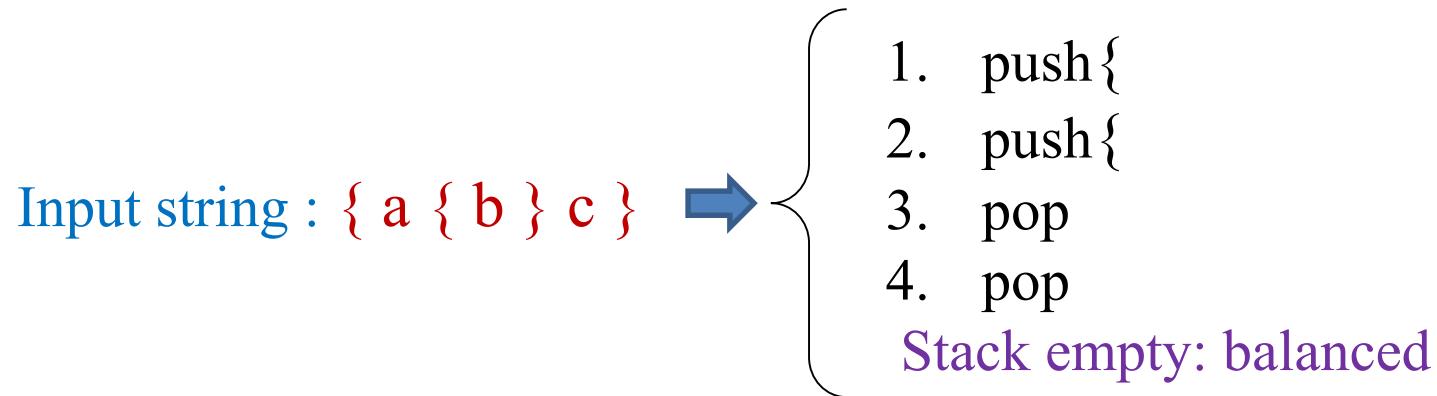
```

if (ch is a '{')
    aStack.push('{')      // Push an opening brace '{'
else if (ch is a '}'')
    aStack.pop()          // Pop a matching open brace
else                      // No matching open brace
    balancedSoFar = false
}
// Ignore all characters other than braces
}  // End of while loop
if (balancedSoFar and aStack.isEmpty())
    aString has balanced braces
else
    aString does not have balanced braces

```

End

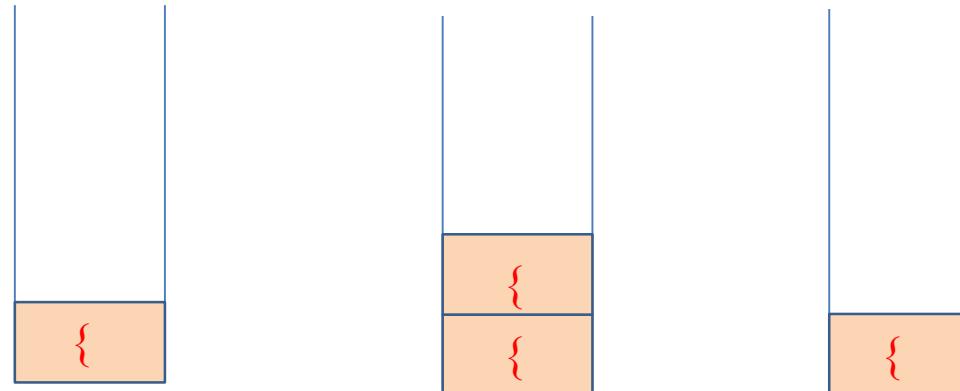
Traces of the algorithm that checks for balanced braces



Stack as algorithm executes

Traces of the algorithm that checks for balanced braces

Input string : { a { b c }  {
1. push {
2. push {
3. pop
Stack not empty: not balanced



If your string is in this form: { a b } c } stack would be empty when next } encountered, then it would not be balanced.



You Try 1. Check for Balanced Braces

For each of the following strings, trace the execution of the balanced-braces algorithm and show the contents of the stack at each step.

- a. x { { y z } } }
- b. { x { y { { z } } } }

The End of Lecture

Course: Data Structures and Algorithms

Instructor: Homeyra Pourmohammadali

**Stacks ADT Application:
Postfix Expression Evaluation**

Learning Outcomes

By the end of this part of lecture you will be able to:

- Use stack ADT to evaluate postfix expressions

Introduction: Operators and Operands

Examples of expressions: $1 - 6$, $A + B$, $4 * F$

Format: < Operand > < Operator > < Operand >

In $A + B$, A and B are operands and + is an operation

In $(4 * F)$, 4 and F are operands and * is an operation

One operator: $3+4=7$

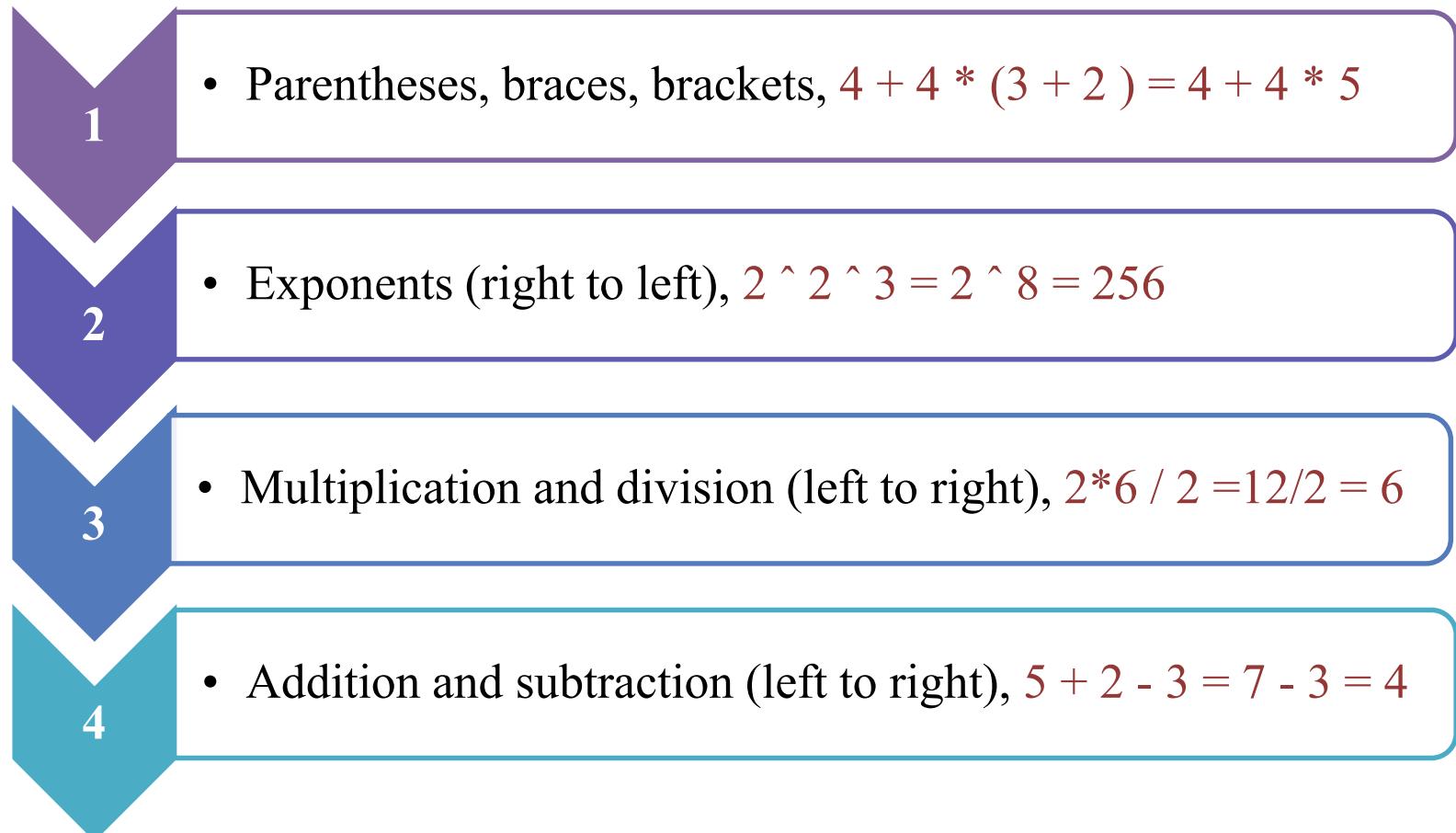
Two operator no parentheses: $5+2*3$

$$\left. \begin{array}{l} =5 + 6 = 11 \\ =7 * 3 = 21 \end{array} \right\}$$

Which one is correct?

Introduction: Order of Operations

Operator Precedence



Introduction: Infix, Prefix and Postfix

- **Infix:** operands separated by operator
 $<\text{Operand}><\text{Operator}><\text{Operand}>$
- **Prefix:** operands preceded by operator
 $<\text{Operator}><\text{Operand}><\text{Operand}>$
- **Postfix:** operands followed by operator
 $<\text{Operand}><\text{Operand}><\text{Operator}>$

Infix	Prefix	Postfix
$4 + 5$	$+ 4 \ 5$	$4 \ 5 +$
$A + B$	$+ A \ B$	$A \ B +$
$A + B * C$	$+ A * \ B \ C$	$A \ B \ C * +$

- Postfix expression can be parsed without ambiguity



You Try 1. Operator Precedence

If you assume the operators $+$, $-$, \times are left associative and \wedge is right associative, the order of precedence (from highest to lowest) would be \wedge , \times , $+$, $-$.

What is the postfix expression corresponding to the infix expression $A + B \times C - D \wedge E \wedge F$?

1. $ABC \times + DE \wedge F \wedge -$
2. $AB + C \times D - E \wedge F \wedge$
3. $- + A \times BC \wedge \wedge DEF$
4. $ABC \times + DEF \wedge \wedge -$

Problem Statement

Use stack ADT to parse and evaluate postfix expressions.

Simplifying Assumptions

The postfix expression is entered as a string of characters

1. The string is a syntactically correct postfix expression
2. No unary operators are present
3. No exponentiation operators are present
4. Operands are single lowercase letters that represent integer values

Algorithm Pseudocode

```
for ( each character ch in the string )
{
    if (ch is an operand)
        Push the value of the operand ch onto the stack
    else      // ch is an operator named op
    {
        // Evaluate and push the result
        operand2 = top of stack
        Pop the stack
        operand1 = top of stack
        Pop the stack
        result = operand1 op operand2
        Push result onto the stack
    }
}
```

Calculations

For instance when evaluating the expression:

$$2 * (3 + 4)$$

The effect of a postfix calculator on a stack can be visualized in a table (as shown in the next slide)

2 * (3 + 4)

Key entered	Calculator Action	Stack (bottom to top)
2	push 2	2
3	push 3	2 3
4	push 4	2 3 4
+	operand2 = peek (4)	2 3 4
	pop	2 3
	operand1 = peek (3)	2 3
	pop	2
	result = operand1 + operand2 (7)	
	push result	2 7
*	operand2 = peek (7)	2 7
	pop	2
	operand1 = peek (2)	2
	pop	
	result = operand1 * operand2 (14)	
	push result	14

You Try 2. Evaluate Postfix Expression



Evaluate the expression $4 * (5 - 3)$

Show the status of the stack after each step.

(use a table similar to the one shown in previous slide)

Next Lecture

We focus on:

- Implementations of stacks using arrays
- Implementations of stacks using linked lists

Readings/ Study Reference

Course Textbook: “*Data Structure and Algorithms in a Nutshell*” A. Wong et al.

Chapter 6. Stacks and Queue

Section 6.1 Stacks (subsection 6.1.3)

The End of Lecture

Course: Data Structures and Algorithms

Instructor: Homeyra Pourmohammadali

You Try Questions and Solutions

Stacks ADT Application:

Postfix Expression Evaluation

You Try 1. Operator Precedence

If you assume the operators $+$, $-$, \times are left associative and \wedge is right associative, the order of precedence (from highest to lowest) would be \wedge , \times , $+$, $-$.

What is the postfix expression corresponding to the infix expression $A + B \times C - D \wedge E \wedge F$?

1. $ABC \times + DE \wedge F \wedge -$
2. $AB + C \times D - E \wedge F \wedge$
3. $- + A \times BC \wedge \wedge DEF$
4. $ABC \times + DEF \wedge \wedge -$

← Answer

You Try 2. Evaluate Postfix Expression



Evaluate the expression $4 * (5 - 3)$

Show the status of the stack after each step.

(use a table similar to the one shown in previous slide)

$4 * (5 - 3)$

Key entered	Calculator Action	Stack (bottom to top)
4	push 4	4
5	push 5	4 5
3	push 3	4 5 3
-	operand2 = peek (3)	4 5 3
	pop	4 5
	operand1 = peek (5)	4 5
	pop	4
	result = operand1 - operand2 (2)	
	push result	4 2
*	operand2 = peek (2)	4 2
	pop	4
	operand1 = peek (4)	4
	pop	
	result = operand1 * operand2 (8)	
	push result	8

The End of You Try Activities

Course: Data Structures and Algorithms

Instructor: Homeyra Pourmohammadali

**Stack Implementation:
Sequential and Linked Representations**

Learning Outcomes

By the end of this part of lecture you will be able to:

- implement a stack using an array
- implement a stack using a linked list
- compare these implementations

Recall: Stack Data Structure

- Stack is a **LIFO** data structure that is **last in first out** method.
- The data or the element stored last in the stack (the top element) will be accessed first.
- In a stack both insertion and deletion takes place at the top of the stack.

Stack Implementation Methods

Array

Linked List

Easy to implement; saves memory (pointers are not used).

Static array: it doesn't grow/shrink at runtime.

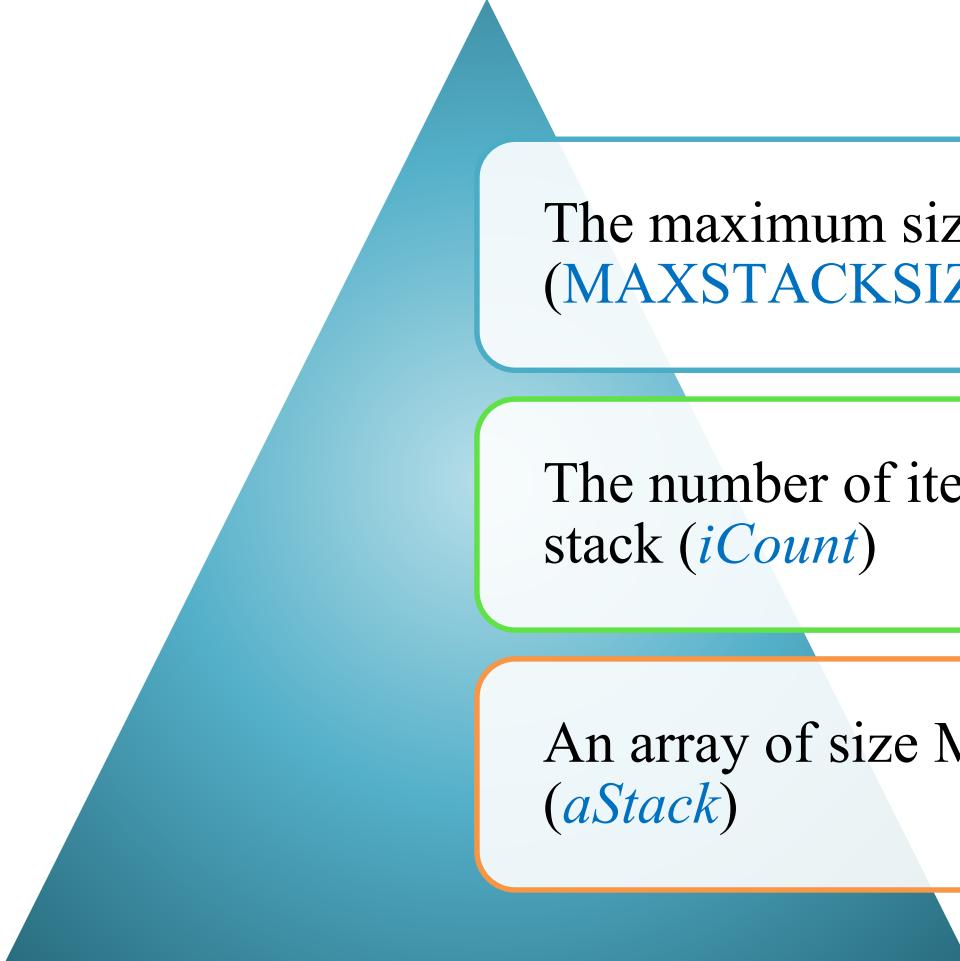
Dynamic array: it can grow/shrink at runtime.

Requires extra memory due to pointers.

Can grow and shrink at runtime.

Stack Implementation using Array

Array-based Implementation: Main Components



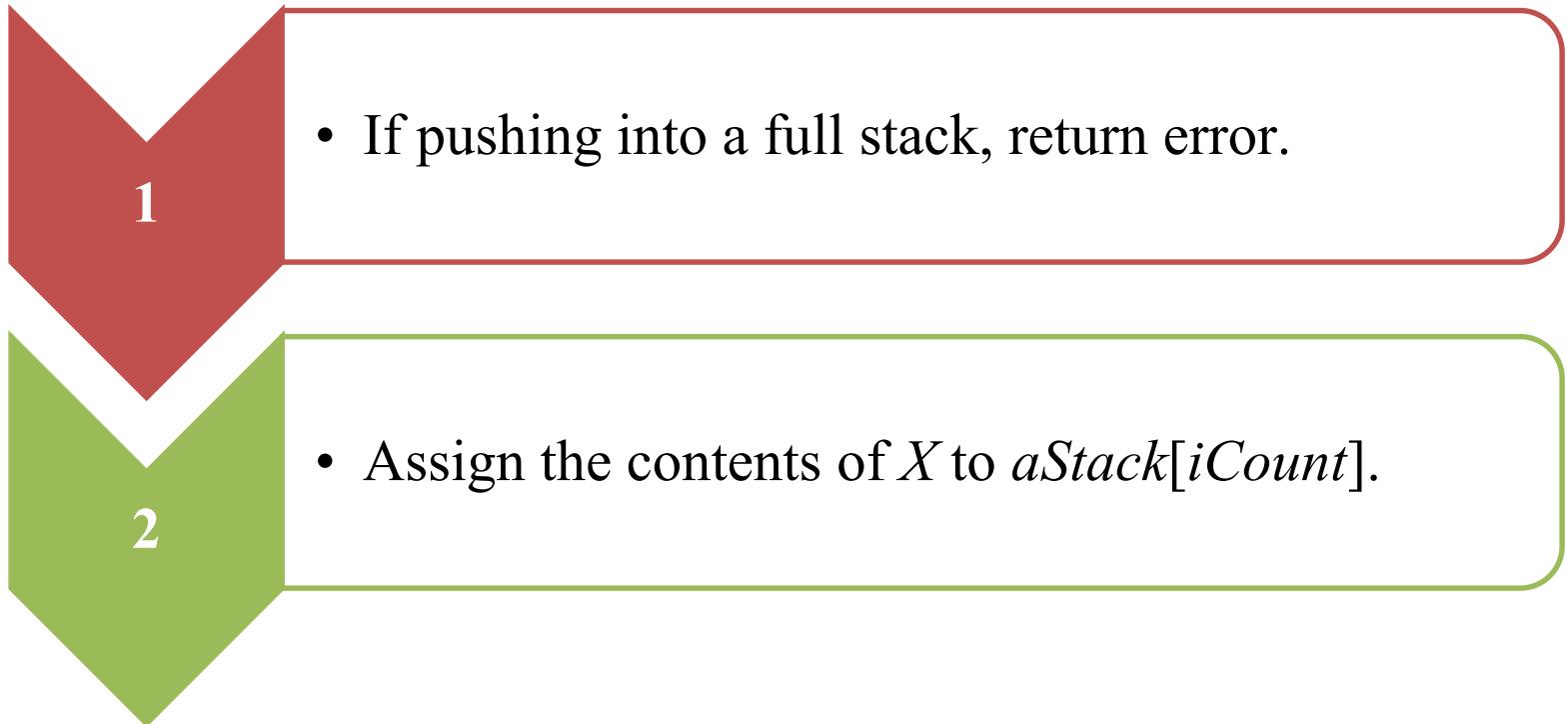
The maximum size of the stack
(MAXSTACKSIZE)

The number of items currently in the stack (*iCount*)

An array of size MAXSTACKSIZE
(*aStack*)

Array-based Implementation: Push Operation

Given an item X , we can push this item into the stack:

- 
- 1
 - If pushing into a full stack, return error.
 - 2
 - Assign the contents of X to $aStack[iCount]$.

Array-based Implementation: Pop Operation

1

- If popping from an empty stack, return error.

2

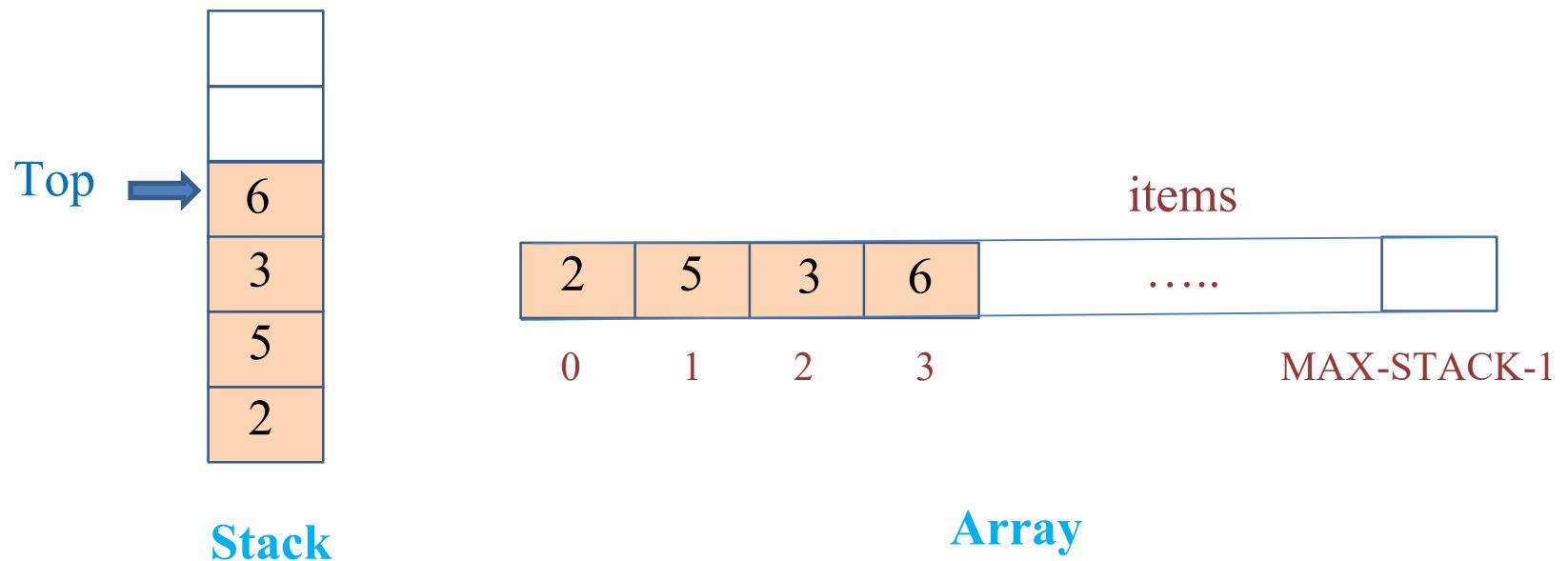
- Decrement $iCount$ by 1.

3

- Return a reference to $aStack[iCount]$

Array-Based Implementation

We can avoid shifting the current entries if we anchor the bottom of the stack at **index 0** and track the location of the stack's top using an index **top** .



Example 1. Array-based stack implementation. header file

```
//ADT stack: Array-based implementation @file ArrayStack.h
```

```
#ifndef _ARRAY_STACK
#define _ARRAY_STACK
#include "StackInterface.h"
```

```
const int MAX_STACK = maximum-size-of-stack;
template < class ItemType >
class ArrayStack : public StackInterface<ItemType>
{
private:
    ItemType items[MAX_STACK]; // Array of stack items
    int top; // Index to top of stack
```

Continue next slide →

Example 1. Array-based stack implementation. header file

public:

```
ArrayStack();           // Default constructor
bool isEmpty() const ;
bool push( const ItemType& newEntry);
bool pop();
ItemType peek() const ;
};
```

// end ArrayStack

```
#include "ArrayStack.cpp"
#endif
```

// Default constructor

Note: If you store a stack's entries in statically allocated memory, the compiler generated destructor and copy constructor will be sufficient. If, you use a dynamically allocated array, you have to define a destructor and a copy constructor.

Example 2. Array-based stack implementation. cpp file

```
// C++ program to implement basic stack operations
#include <bits/stdc++.h>
using namespace std;
#define MAX 1000 // Maximum size of Stack
class Stack {
    int top;
public:
    int a[MAX]; // Maximum size of Stack

    Stack() { top = -1; }
    bool push(int x);
    int pop();
    int peek();
    bool isEmpty();
};
```

Continue next slide →

Example 2. Array-based stack implementation. cpp file

```
bool Stack::push(int x)
{
    if (top >= (MAX - 1)) {
        cout << "Stack Overflow";
        return false;
    }

    else {
        a[++top] = x;
        cout << x << " pushed into stack\n";
        return true;
    }
}
```

Continue next slide →

Example 2. Array-based stack implementation. cpp file

```
int Stack::pop()
{
    if (top < 0) {
        cout << "Stack Underflow";
        return 0;
    }

    else {
        int x = a[top--];
        return x;
    }
}
```

Continue next slide →

Example 2. Array-based stack implementation. cpp file

```
int Stack::peek()
{
    if (top < 0) {
        cout << "Stack is Empty";
        return 0;
    }
    else {
        int x = a[top];
        return x;
    }
}
bool Stack::isEmpty()
{
    return (top < 0);
}
```

Continue next slide →

Example 2. Array-based stack implementation. cpp file

```
// to test above functions  
int main()  
{  
    class Stack s;  
    s.push(1);  
    s.push(3);  
    cout << s.pop() << " Popped from stack\n";
```

```
return 0;
```

```
}
```

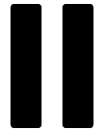
Output:

1 pushed into stack
3 pushed into stack
3 popped from stack

End of example 2

Alternate Array-based Implementation

- You can also allocate the storage at run time to implement stacks using arrays (**dynamically allocated arrays**).
- You can let the maximum number of items be a parameter to a class constructor.
- To allocate an array, you can attach to the data type name the array size in the bracket (e.g. **new ItemType[size]**).
- The **new** operator, returns the base address of the newly allocated array.



You Try 1. Array-based implementation. Dynamic Stack

True or False?

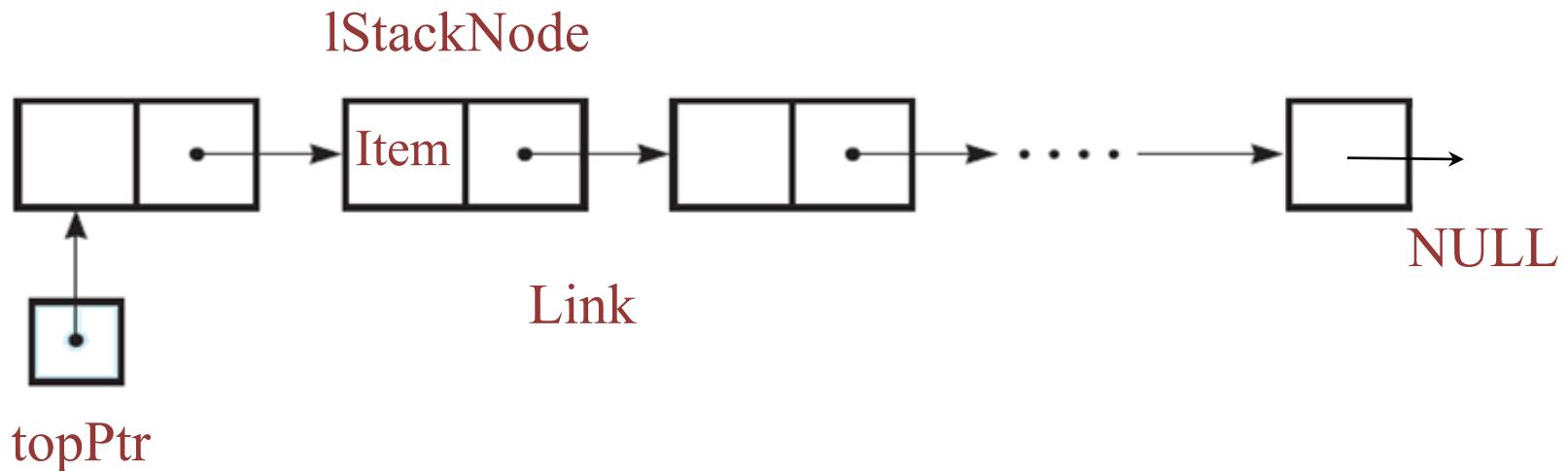
You need a class destructor and delete operation for stack implementation using dynamic array.

Reading: After watching the full lecture read the header file in section 6.4.1 of your textbook for DynamicStack and notice how this implementation (using dynamically resizable array) is different from stack implementation using static array.

Stack Implementation using Linked List

Linked-based Implementation

A linked stack representation, consists of a linked list (*lStack*), where each node in the linked list (*lStackNode*) holds an item (*Item*) and a pointer to the next node (*Link*)



topPtr is a pointer to the head of the linked nodes.

Linked-based Implementation

- For array-based implementation, the next element is put at the end of the list. Then, push function, insert new element at the end of array.
- For linked-based implementation, the next element is put at the first of the list. This gives immediate access to the insertion point (last item inserted). Then push function insert new element at the beginning of linked list.

Linked-based Implementation: Push Operation

1

- Create a temporary stack node *tempStackNode*.

2

- Assign the Contents of *X* to *tempStackNode.Item*

3

- Assign *tempStackNode.Link* to point to the 1st node of *lStack*.

4

- Assign *lStack* to point to *tempStackNode*.

Linked-based Implementation: Pop Operation

- 1 • If popping from an empty stack, return error.
- 2 • Create a temporary stack node pointer tmp .
- 3 • Point tmp to the first item of $lStack$.
- 4 • Create a stack item X .
- 5 • Assign the contents from $temp.Item$ to X .
- 6 • Assign $lStack$ to point to the 2nd node of $lStack$ (if one exists).
- 7 • Free tmp .
- 8 • Return X .

Example 3. Linked-based stack implementation. header file

```
/** ADT stack: Link-based implementation.  
@file LinkedStack.h */  
  
#ifndef _LINKED_STACK  
#define _LINKED_STACK  
#include "StackInterface.h"  
#include "Node.h"  
template < class ItemType>  
class LinkedStack : public StackInterface<ItemType>  
{  
private:  
    Node<ItemType>* topPtr; // Pointer to first node in the chain;  
    // this node contains the stack's top
```

Example 3. Linked-based stack implementation. header file

public:

```
// Constructors and destructor:  
    LinkedStack(); // Default constructor  
// Copy constructor:  
    LinkedStack(const LinkedStack<ItemType>& aStack);  
    virtual ~LinkedStack(); // Destructor  
// Stack operations:  
    bool isEmpty() const;  
    bool push(const ItemType& newItem);  
    bool pop();  
    ItemType peek() const;  
}; // end LinkedStack  
#include "LinkedStack.cpp"  
#endif
```

End of header file

Example 4. Linked-based stack implementation. cpp file

```
void StackType::Push(ItemType newItem)
// Adds newItem to the top of the stack. Stack is bounded by size of memory.
// Pre: Stack has been initialized. Post: If stack is full, FullStack exception
// is thrown else newItem is at the top of the stack
{
    if (IsFull())
        throw FullStack(); // throws an exception when full stack problem shows up
    else
    {
        NodeType* location; // declare a pointer to a node
        location = new NodeType; // get a new node
        location->info = newItem; // store the item in the node
        location->next = topPtr; // store address of first node in the next field of new node
        topPtr = location; //store address of new node
    }
}
```

Example 4. Linked-based stack implementation. cpp file

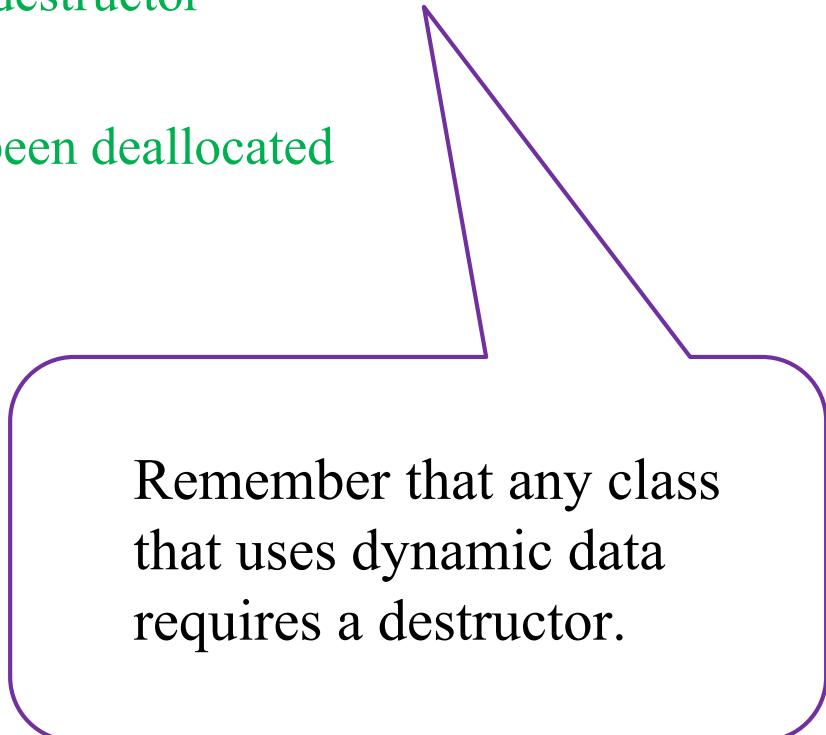
```
void StackType::Pop()
// Removes top item from Stack and returns it in item.
// Pre: Stack has been initialized.
// Post: If stack is empty, EmptyStack exception is thrown;
//       else top element has been removed.
{
    if (IsEmpty())
        throw EmptyStack();
    else
    {
        NodeType* tempPtr;
        tempPtr = topPtr;
        topPtr = topPtr->next;
        delete tempPtr;
    }
}
```

Example 4. Linked-based stack implementation. cpp file

```
bool StackType::IsEmpty() const  
  
// Returns true if there are no elements on the stack; false otherwise  
  
{  
  
    return (topPtr == NULL);  
  
}
```

Example 4. Linked-based stack implementation. cpp file

```
StackType::~StackType()      // Class destructor  
  
// Post: stack is empty; all items have been deallocated  
{  
    NodeType* tempPtr;  
    while (topPtr != NULL)  
    {  
        tempPtr = topPtr;  
        topPtr = topPtr->next;  
        delete tempPtr;  
    }  
}
```



Remember that any class that uses dynamic data requires a destructor.

Comparing Stack Implementation Methods

Linked-based Implementation

- It uses dynamically allocated storage.
- It only require space for the number of elements actually on the stack at run time.
- The elements are larger, because both the link and user's data need to be stored.

Array-based Implementation

- An array variable of maximum stack size takes the same amount of memory, no matter how many array slots are actually used.
- You need to reserve space for the maximum possible.

Comparing Stack Implementation Methods

Both Implementations

- In terms of Big-O comparison of stack operations, all types of stack implementations using static array, dynamically allocated array and linked list implementations, for the main operations such as `push`, `pop`, `IsFull` and `IsEmpty`, have the same constant complexity class of $O(1)$.

Comparing Stack Implementation Methods

Linked-based Implementation

- It gives more flexibility.
- In situations where the stack size is totally unpredictable, this implementation is preferable, because size is largely irrelevant.

Array-based Implementation

- It is simple and efficient
- If pushing and popping occur frequently, the array-based implementation executes faster because it does not incur the run time overhead of *new* and *delete* operations.

You Try 2. Time complexity. Stack implementation-linked list

A stack is implemented with a linked list. What is the time complexity of the push and pop operations of the stack implemented using linked list (if implemented efficiently)?

1. It is $O(1)$ for insertion and $O(n)$ for deletion
2. It is $O(n)$ for insertion and $O(1)$ for deletion
3. It is $O(1)$ for insertion and $O(1)$ for deletion
4. It is $O(n)$ for insertion and $O(n)$ for deletion

You Try 3. Stack Implementation using Linked List

Which statement is true?

1. In push operation, if new nodes are inserted at the beginning of linked list, then in pop operation, nodes must be removed from end.
2. In push operation, if new nodes are inserted at the end, then in pop operation, nodes must be removed from the beginning.
3. Both of the above
4. None of the above

Next Lecture

We focus on:

- Queues ADT Applications
- Queues ADT Implementation

Readings/ Study Reference

Course Textbook: “*Data Structure and Algorithms in a Nutshell*” A. Wong et al.

Chapter 6. Stacks and Queue

Section 6.1 Stacks (subsection 6.1.4, 6.1.5)

The End of Lecture

Course: Data Structures and Algorithms

Instructor: Homeyra Pourmohammadali

You Try Questions and Solutions

Stack Implementation:

Sequential and Linked Representations



You Try 1. Array-based implementation. Dynamic Stack

True or False?

You need a class destructor and delete operation for stack implementation using dynamic array.

Answer: True

Reading: After watching the full lecture read the header file in section 6.4.1 of your textbook for DynamicStack and notice how this implementation (using dynamically resizable array) is different from stack implementation using static array.

You Try 2. Time complexity. Stack implementation-linked list

A stack is implemented with a linked list. What is the time complexity of the push and pop operations of the stack implemented using linked list (if implemented efficiently)?

1. It is $O(1)$ for insertion and $O(n)$ for deletion
2. It is $O(n)$ for insertion and $O(1)$ for deletion
3. It is $O(1)$ for insertion and $O(1)$ for deletion
4. It is $O(n)$ for insertion and $O(n)$ for deletion

You Try 3. Stack Implementation using Linked List

Which statement is true?

1. In push operation, if new nodes are inserted at the beginning of linked list, then in pop operation, nodes must be removed from end.
2. In push operation, if new nodes are inserted at the end, then in pop operation, nodes must be removed from the beginning.
3. Both of the above
4. None of the above

You Try 3 Solution. Stack Implementation using Linked List

Discussion: Stacks are based on **Last In First Out** working principle. Two methods can be applied to implement a stack using linked list:

1. In push operation, if new nodes are inserted at the beginning of linked list, then in pop operation, nodes must be removed from beginning.
2. In push operation, if new nodes are inserted at the end of linked list, then in pop operation, nodes must be removed from end.

The End of You Try Activities

Course: Data Structures and Algorithms

Instructor: Homeyra Pourmohammadali

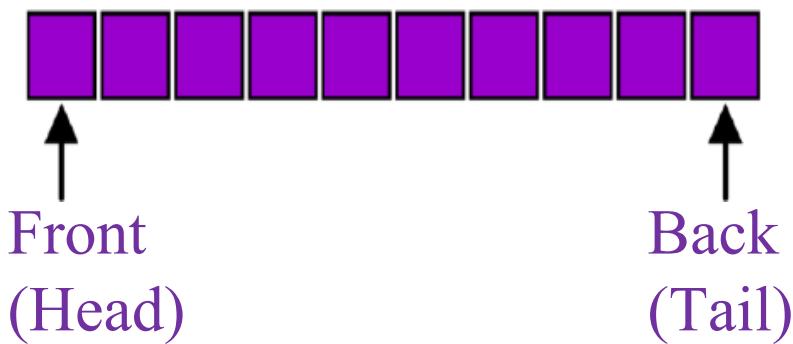
**Queue ADT Implementation:
Linked and Sequential Representations**

Learning Outcomes

By the end of this lecture you will be able to:

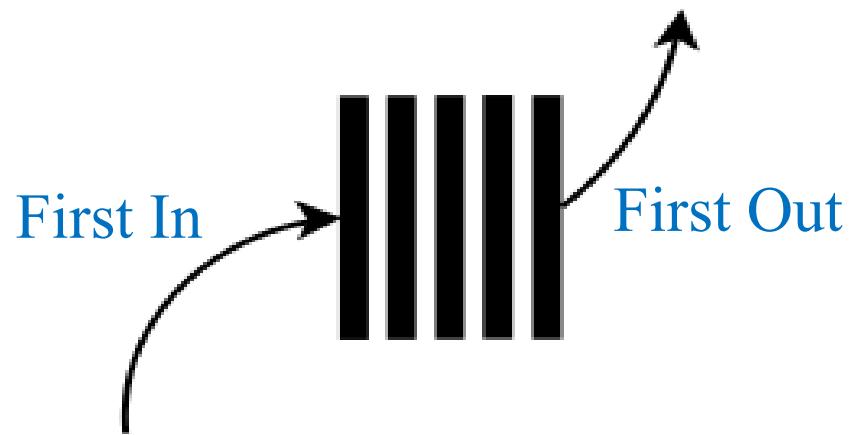
- use enqueue and dequeue functions.
- implement a queue using a linked list.
- implement a linear queue using a dynamic array
- find out why circular queues are used instead of linear queues.

Recall: Queue Data Structure



In queues, both ends are open and the indices of both front and back ends are tracked.

First In/ First Out (FIFO)
data structure.



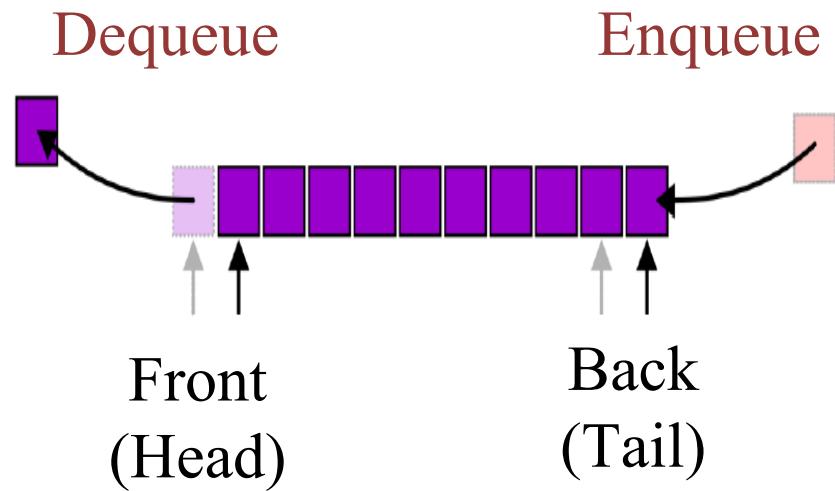
Objects are inserted and removed from different ends

Recall: Main Operations of Queues

- **Enqueue:** insert operation.
- **Dequeue:** delete operation.
- **Peek:** look at the front item (without changing queue).
- **isEmpty** check if the queue is empty

Undefined operations:

- Enqueue a full queue (**overflow**)
- Dequeue or peek an empty queue (**underflow**)



Operations Inputs and Outputs

Tasks	Queues
Adds <code>newEntry</code> at the back of queue. Output: True if the operation is successful; otherwise false.	<code>enqueue(newEntry)</code> <code>boolean</code>
Remove the front of queue (that was added earliest). Output: True if operation is successful; otherwise false.	<code>dequeue()</code> <code>boolean</code>
Peek front of queue Output: The front of the queue.	<code>peekFront()</code> <code>ItemType</code>
Check if queue is empty Output: True, if the queue is empty; otherwise false.	<code>isEmpty()</code> <code>boolean</code>

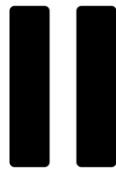
Example 1. enqueue and dequeue operations

See the effect of insertion and deletion operations on a queue of integers.

Operation	Queue after operation
aQueue = <i>an empty queue</i>	
aQueue.enqueue(9)	9
aQueue.enqueue(2)	9 2
aQueue.enqueue(8)	9 2 8
aQueue.peekFront()	9 2 8 (Returns 9)
aQueue.dequeue()	2 8
aQueue.dequeue()	8

Note: `enqueue` adds an item at the back of the queue and that `peekFront` looks at the item at the front of the queue, whereas `dequeue` removes the item at the front of the queue.

You Try 1. enqueue and dequeue operations



What do the initially empty queues queue1 and queue2 “look like” after the following sequence of operations?

Operations	Queue 1	Queue 2
queue1.enqueue(1)		
queue1.enqueue(2)		
queue2.enqueue(3)		
queue2.enqueue(4)		
queue1.dequeue()		
queueFront = queue2.peekFront()		
queue1.enqueue(queueFront)		
queue1.enqueue(5)		
queue2.dequeue()		
queue2.enqueue(6)		

You Try 2. enqueue and dequeue operations



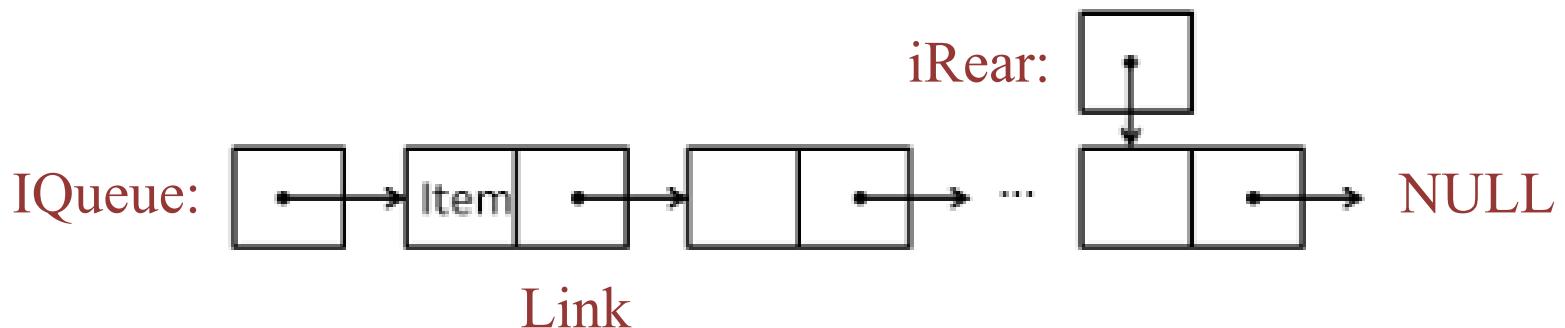
If you add the letters **A**, **B**, **C**, and **D** in sequence to a queue of characters and then remove them, in what order will they leave the queue? Try to show it in this table.

Operations	Queue

Queue Implementation using Linked List

Linked-based Implementation

- A linked queue representation, consists of a linked list (*lQueue*).
- Each node in the linked list (*lQueueNode*) holds an item (*Item*) and a pointer to the next node (*Link*) and a pointer (*iRear*) to the last element of *lQueue*.



Linked-based Implementation: Enqueue Operation

1

- Create a temporary queue node $tempQueueNode$.

2

- Assign the contents of X to $tempQueueNode.Item$.

3

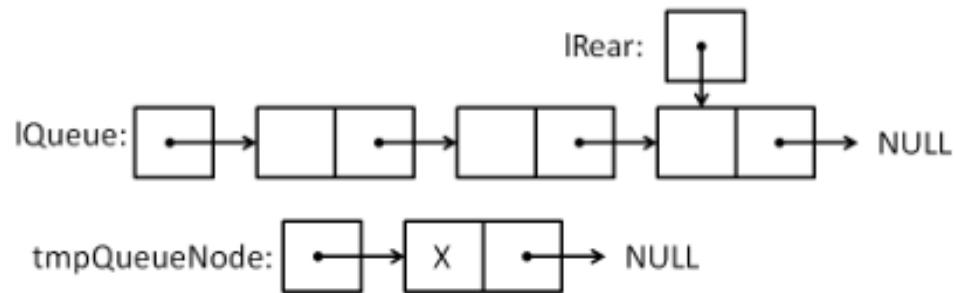
- Assign $lRear.Link$ to point to $tempQueueNode$.

4

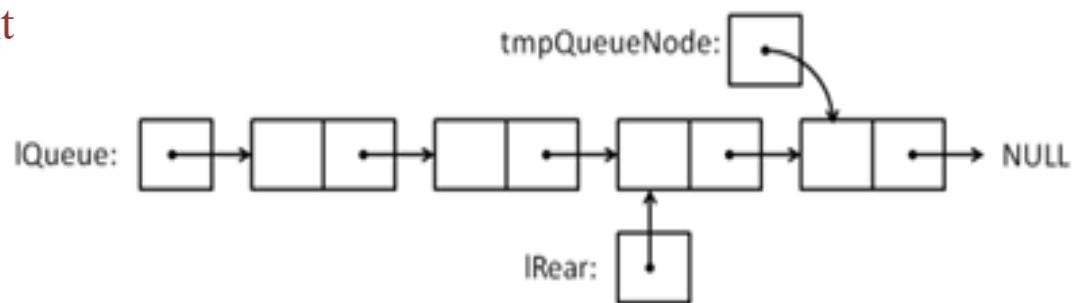
- Assign $lRear$ to point to $tempQueueNode$.

Linked-based Implementation: Enqueue Operation

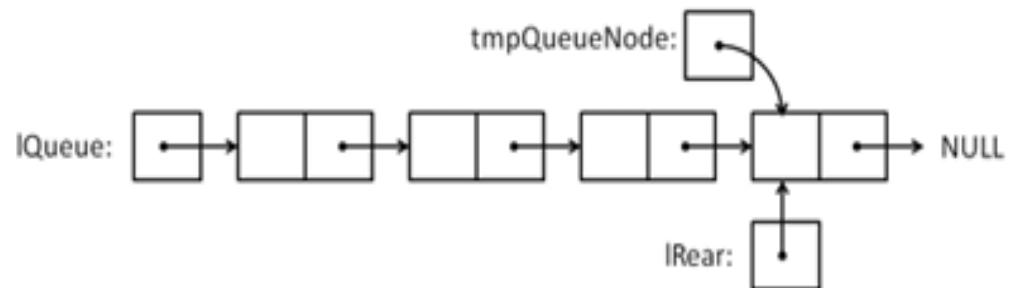
Assign the contents of X to $\text{tmpQueueNode}.Item$.



Assign $lRear.Link$ to point to tmpQueueNode .



Assign $lRear$ to point to tmpQueueNode .

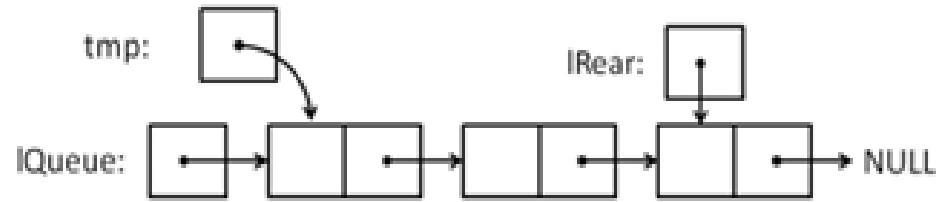


Linked-based Implementation: Dequeue Operation

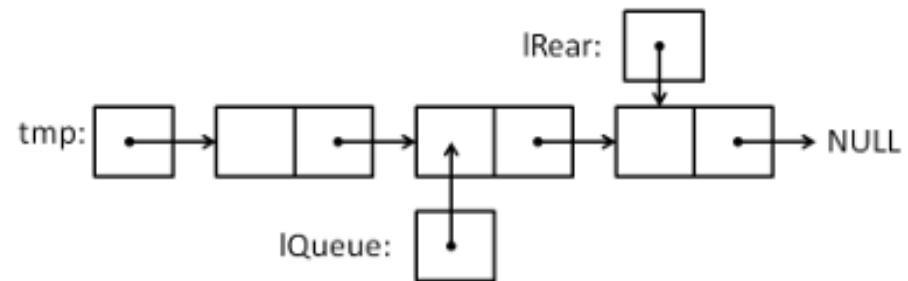
- 1 • If dequeuing from an empty queue, return error.
- 2 • Create a temporary queue node pointer tmp .
- 3 • Point tmp to the first item of $lQueue$.
- 4 • Assign $lQueue$ to point to the 2nd node of $lQueue$ (if one exists).
- 5 • Create a queue item X .
- 6 • Assigns the contents from $temp.Item$ to X .
- 7 • Free tmp .
- 8 • Return X .

Linked-based Implementation: Enqueue Operation

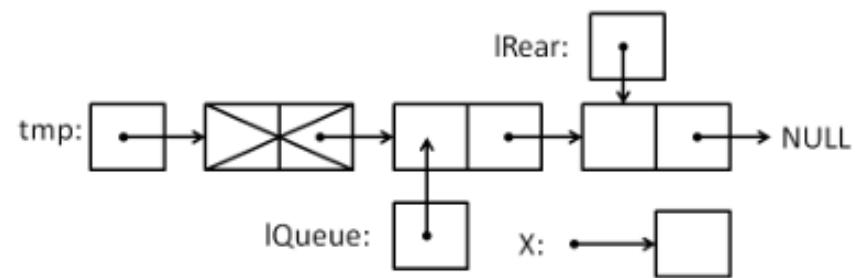
Point *tmp* to the first item of *lQueue*.



Assign *lQueue* to point to the 2nd node of *lQueue*.



Free *tmp*.



Linked-based Implementation: Peek Operation

If peeking from an empty queue, return error.

Create a temporary queue node pointer tmp .

Point tmp to the first item of $lQueue$.

Create a queue item X .

Assign the contents from $tmp.Item$ to X .
Then return Item to X.

Queue Implementation using Array

Array-based Implementation: Main Components

The maximum size of the queue
(MAXQUEUESIZE)

An index pointing to the front of the
queue (*iFront*)

An index pointing to location
immediately after the rear of queue
(*iRear*)

An array of size MAXQUEUESIZE
(*aQueue*)

Enqueue Operation

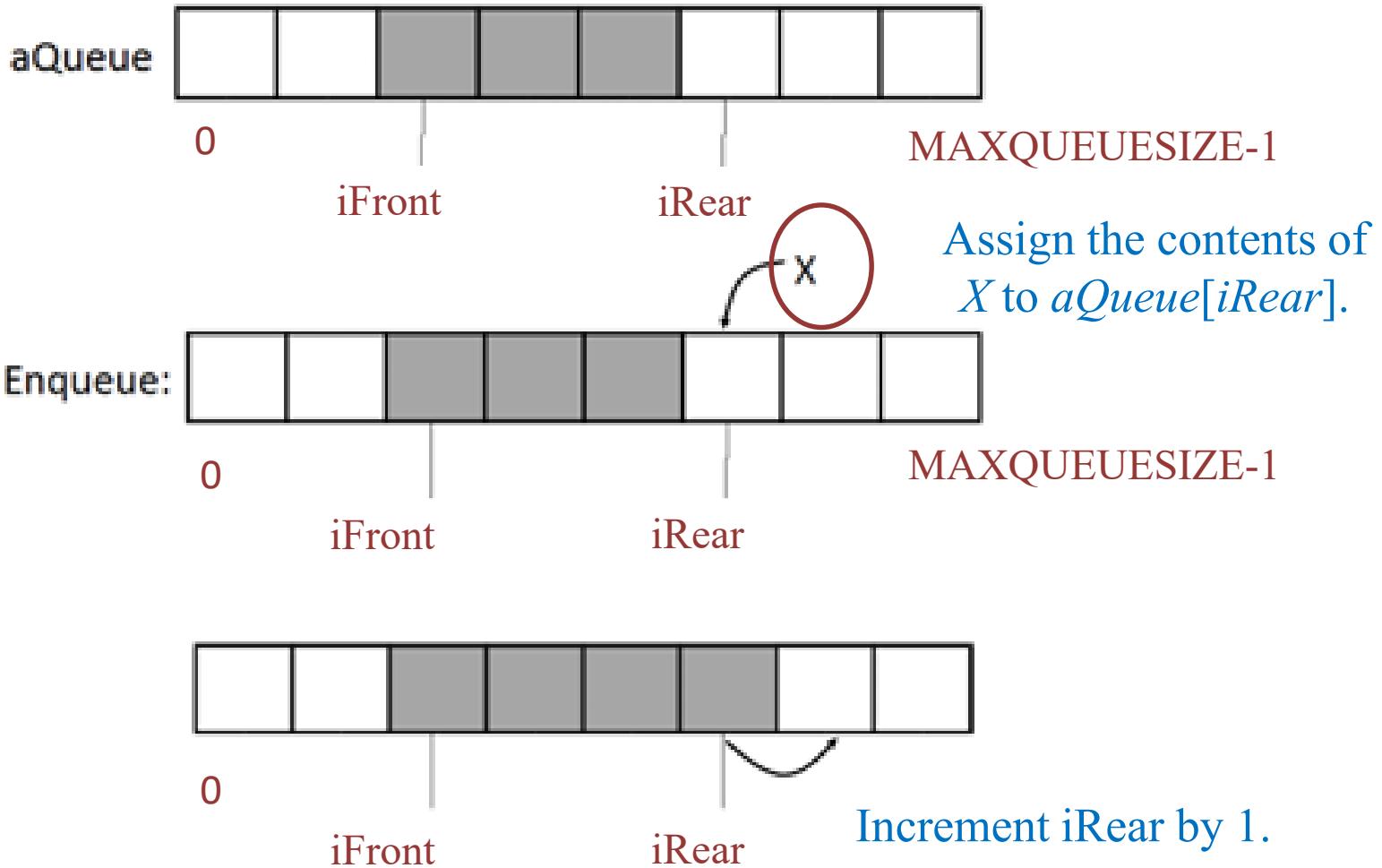
Given an item X , we can queue this item into the queue by:



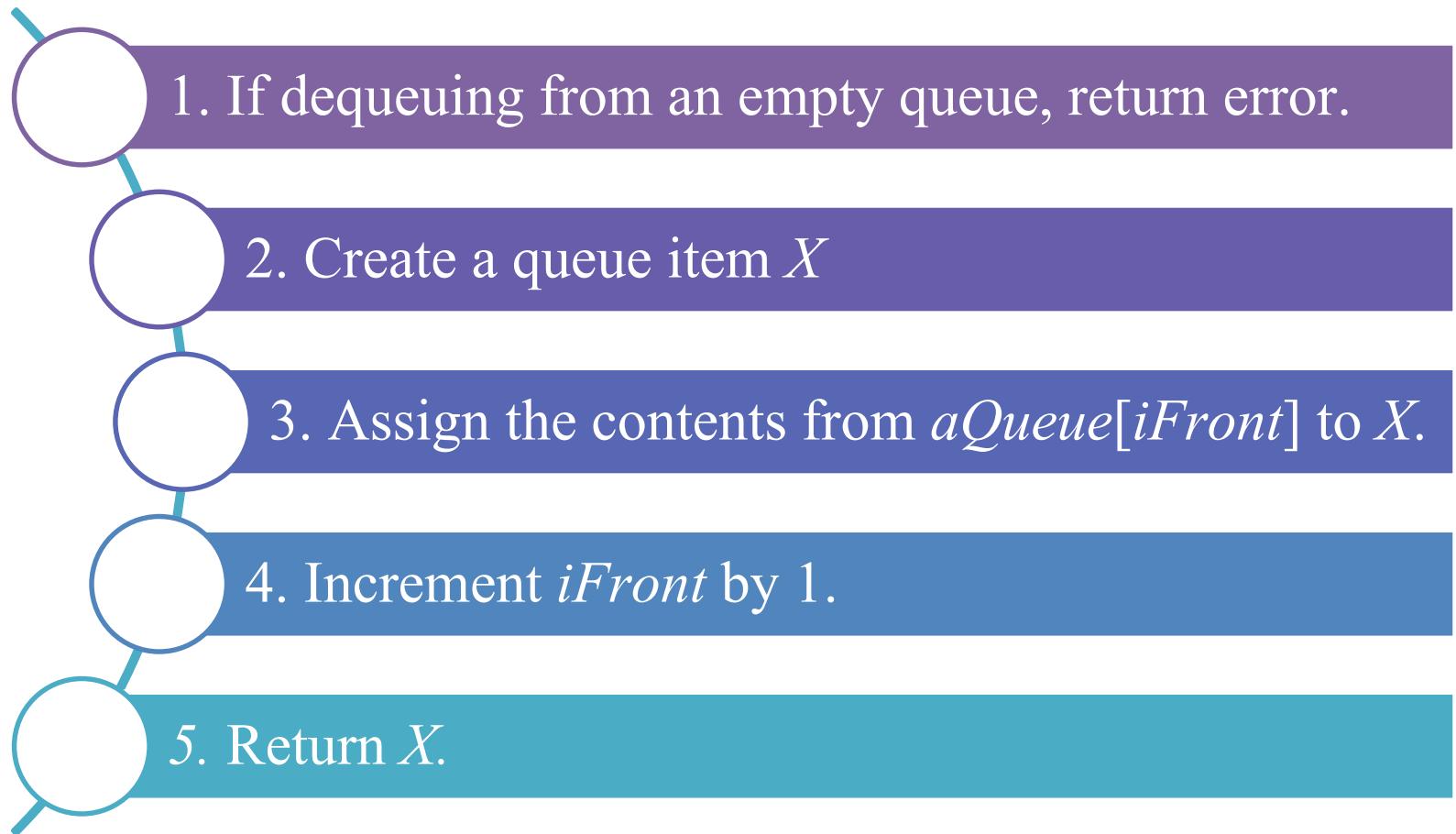
- If enqueueing a full queue, return error.
- Assign the contents of X to $aQueue[iRear]$.
- Increment $iRear$ by 1.

Enqueue Operation

An array of size MAXQUEUESIZE (*aQueue*)

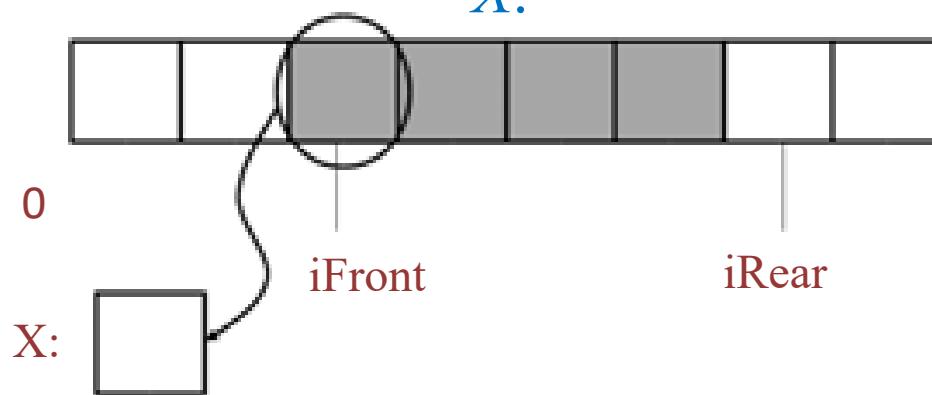


Dequeue Operation

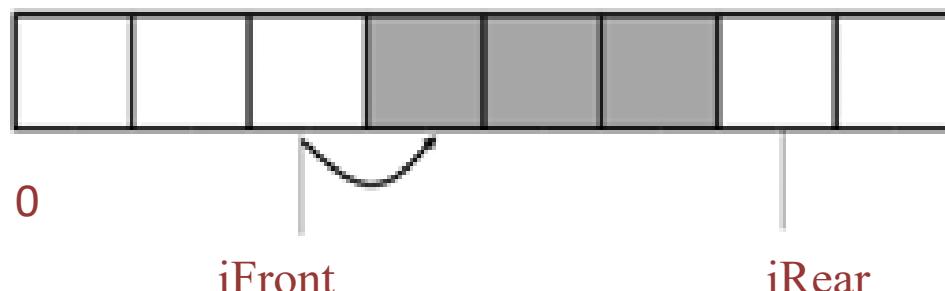


Dequeue Operation

Assign the contents from $aQueue[iFront]$ to X .

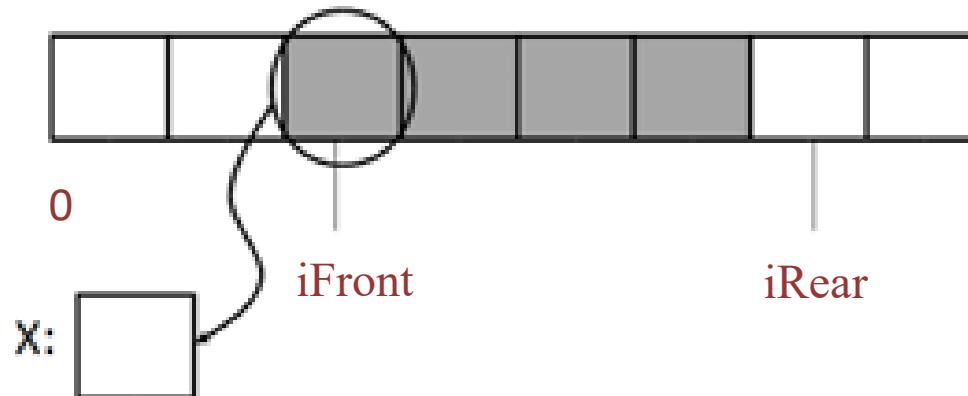


Increment $iFront$ by 1.



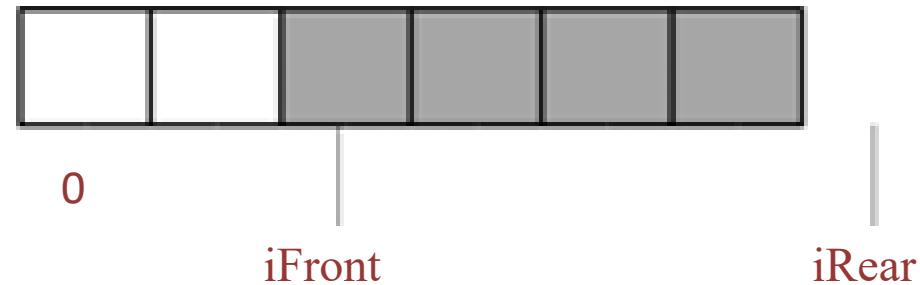
Peek Operation

1.
 - If peeking from an empty queue, return error.
2.
 - Return a reference to $aQueue[iFront]$.



Array-based Implementation Problems

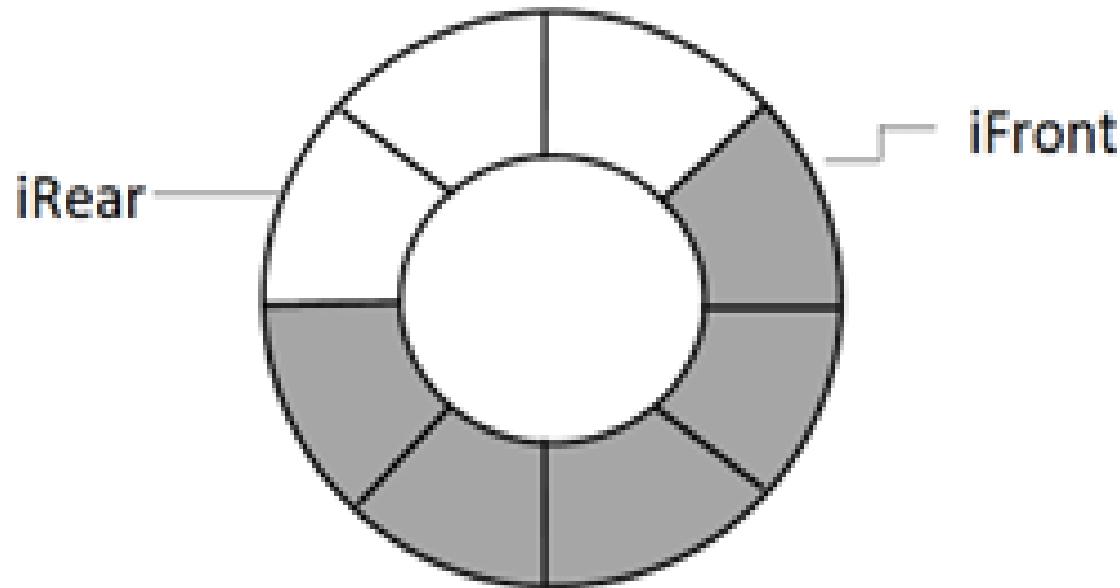
Question: What happens when an enqueue is called on this queue?



1. Cannot put an item into the location pointed by $aQueue[iRear]$ ($iRear$ pointing beyond the index limit of the array, causing an invalid memory write)
2. Cannot return an error (array is not full)

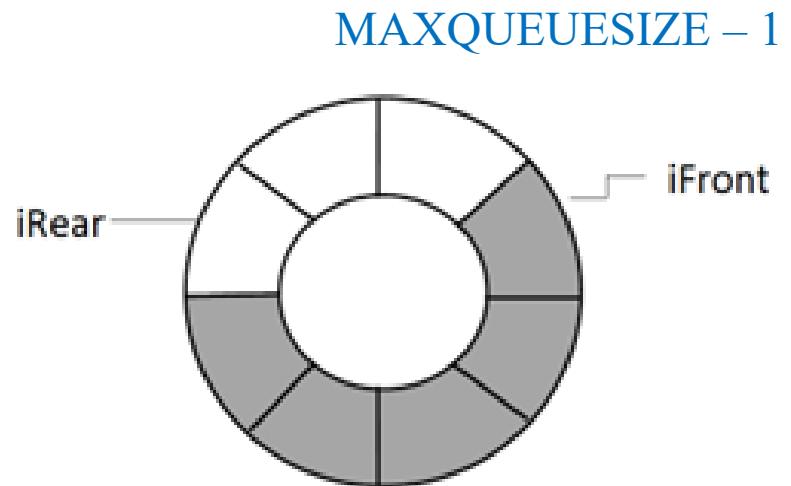
Circular Queue

A solution is possible by viewing the array as circular. To remove an item, you increment the queue index front , and to insert an item, you increment back . Notice that front and back “advance” clockwise around the array.



Circular Queue

- When either front or rear advances past $\text{MAXQUEUESIZE} - 1$, it should wrap around to 0.
- This wraparound eliminates the problem of rightward drift, because here the circular array has no end.
- You obtain the wraparound effect of a circular queue by using **modulo arithmetic (%) operator in C++** when incrementing front and rear .



Example 2. Circular Queue



If modulo arithmetic (%) is not used:

$$iRear = (iRear + 1) = (4+1) = 5$$



Index is beyond array limit



If modulo arithmetic is used:

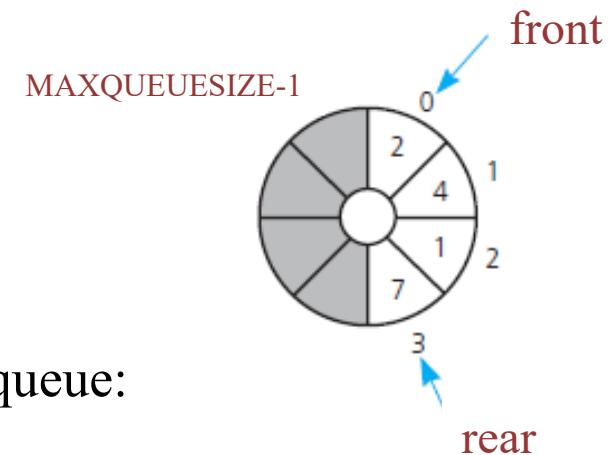
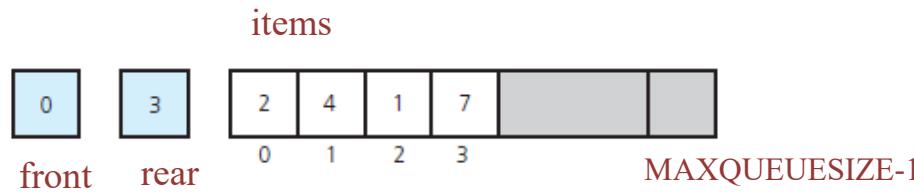
$$iRear = (iRear + 1)\%MAXQUEUESIZE = (4+1)\%5 = 0$$

Note: Operation $m\%n$ returns the remainder of m/n

With this implementation, an item can always be added at the end of queue as long as the queue is not full.

Example 3. Circular Queue

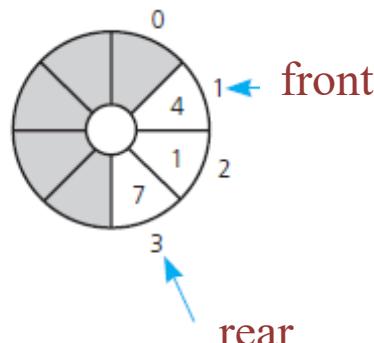
An array-based implementation: linear queue



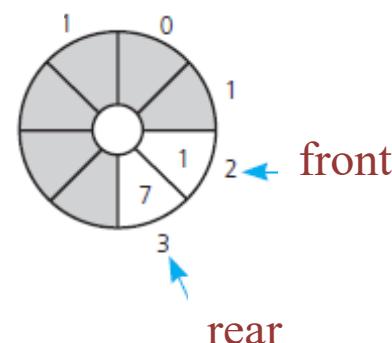
An array-based implementation: circular queue

The effect of three consecutive operations on the queue:

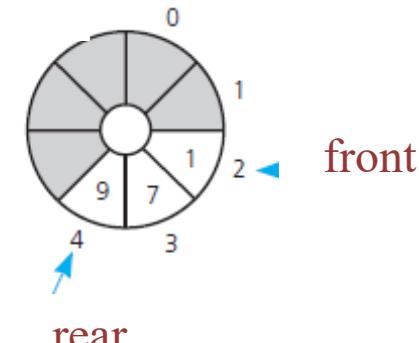
dequeue();



dequeue();



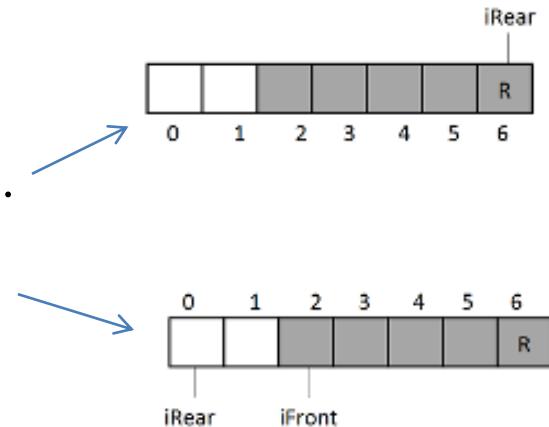
enqueue(9);



Circular Queue Implementation

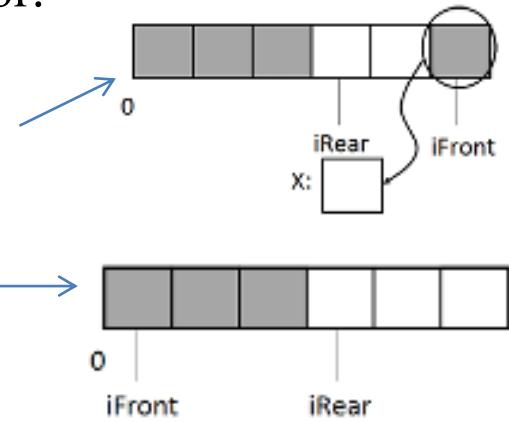
Enqueue Implementation:

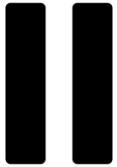
1. If enqueueing into a full queue, return error
2. Assign the contents of X to $aQueue[iRear]$.
3. $iRear = (iRear + 1) \% \text{MAXQUEUESIZE}$.



Dequeue Implementation:

1. If dequeuing into an empty queue, return error.
2. Create a queue item X .
3. Assign the contents of X to $aQueue[iFront]$.
4. $iFront = (iFront + 1) \% \text{MAXQUEUESIZE}$ →
5. Return X .





You Try 3. Comparing Implementation Methods

Think about two array-based and linked implementations, in terms of the amount of memory required to store the structure and the amount of “work” required by the solution, as expressed in Big-O notation.

Try to compare these implementations.

Next Lecture

We focus on:

- Trees and their terminologies
- Introducing the binary trees

Readings/ Study Reference

Course Textbook: “*Data Structure and Algorithms in a Nutshell*” A. Wong et al.

Chapter 6. Stacks and Queue

Section 6.2 Queues

Subsection 6.2.2. Linked Representation

Subsection 6.2.3. Sequential Representation

The End of Lecture

Course: Data Structures and Algorithms

Instructor: Homeyra Pourmohammadali

You Try Questions and Solutions
Queue ADT Implementation:
Linked and Sequential Representations

You Try 1. enqueue and dequeue operations

What do the initially empty queues queue1 and queue2 “look like” after the following sequence of operations?

Operations	Queue 1	Queue 2
queue1.enqueue(1)		
queue1.enqueue(2)		
queue2.enqueue(3)		
queue2.enqueue(4)		
queue1.dequeue()		
queueFront = queue2.peekFront()		
queue1.enqueue(queueFront)		
queue1.enqueue(5)		
queue2.dequeue()		
queue2.enqueue(6)		

You Try 1 Solution. enqueue and dequeue operations

Both queue1 and queue2 are initially empty.

Operations	Queue 1	Queue 2
queue1.enqueue(1)	1	
queue1.enqueue(2)	1 2	
queue2.enqueue(3)	1 2	3
queue2.enqueue(4)	1 2	3 4
queue1.dequeue()	2	3 4
queueFront = queue2.peekFront()	2	3 4 (Return 3)
queue1.enqueue(queueFront)	2 3	3 4
queue1.enqueue(5)	2 3 5	3 4
queue2.dequeue()	2 3 5	4
queue2.enqueue(6)	2 3 5	4 6

`enqueue` adds an item at the back of the queue; `peekFront` returns the item at the front of the queue; `dequeue` removes the item at the front of the queue.

You Try 2. enqueue and dequeue operations

If you add the letters **A**, **B**, **C**, and **D** in sequence to a queue of characters and then remove them, in what order will they leave the queue? Try to show it in this table.

Operations	Queue

You Try 2 Solution. enqueue and dequeue operations

If you add the letters **A**, **B**, **C**, and **D** in sequence to a queue of characters and then remove them:

Operations	Queue
queue.enqueue(A)	A
queue.enqueue(B)	A B
queue.enqueue(C)	A B C
queue.enqueue(D)	A B C D
queue.dequeue()	B C D
queue.dequeue()	C D
queue.dequeue()	D
queue.dequeue()	

They leave the queue at the same order that they entered the queue.

You Try 3. Comparing Implementation Methods

Think about two array-based and linked implementations, in terms of the amount of memory required to store the structure and the amount of “work” required by the solution, as expressed in Big-O notation.

Try to compare these implementations.

You Try 3 Solution. Comparing Implementation Methods

Size and Memory Requirements

An array variable of the maximum queue size takes the same amount of memory, no matter how many array slots are actually used; we need to reserve space for the maximum possible number of elements.

The linked implementation using dynamically allocated storage space requires space only for the number of elements actually in the queue at run time. Note, however, that the node elements are larger, because we must store the link (the next member) as well as the user's data.

You Try 3 Solution. Comparing Implementation Methods

Efficiency and Big-O

The class constructors, and IsEmpty operations are **O(1)**; they always take the same amount of work regardless of how many items are on the queue.

For enqueue and dequeue operations: in both implementations, we can directly access the front and rear of the queue. The amount of work done by these operations is independent of the queue size, so these operations also have **O(1)** complexity.

The class destructor in the dynamically allocated linked structure contains a loop that executes as many times as there are items on the queue. Thus the dynamically linked version has **O(N)** complexity.

The End of You Try Activities

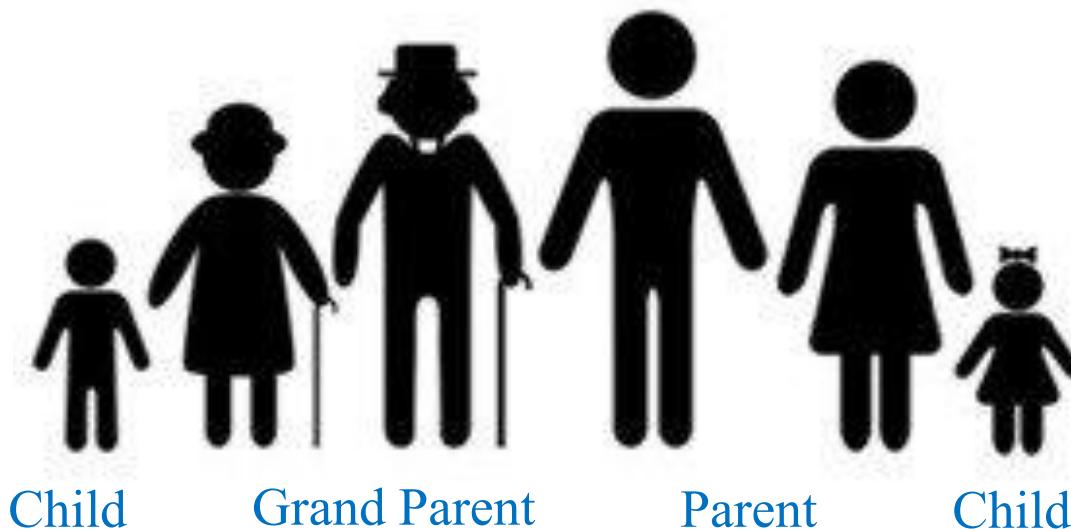
Course: Data Structures and Algorithms

Instructor: Homeyra Pourmohammadali

Trees: Basic Terminologies and Examples

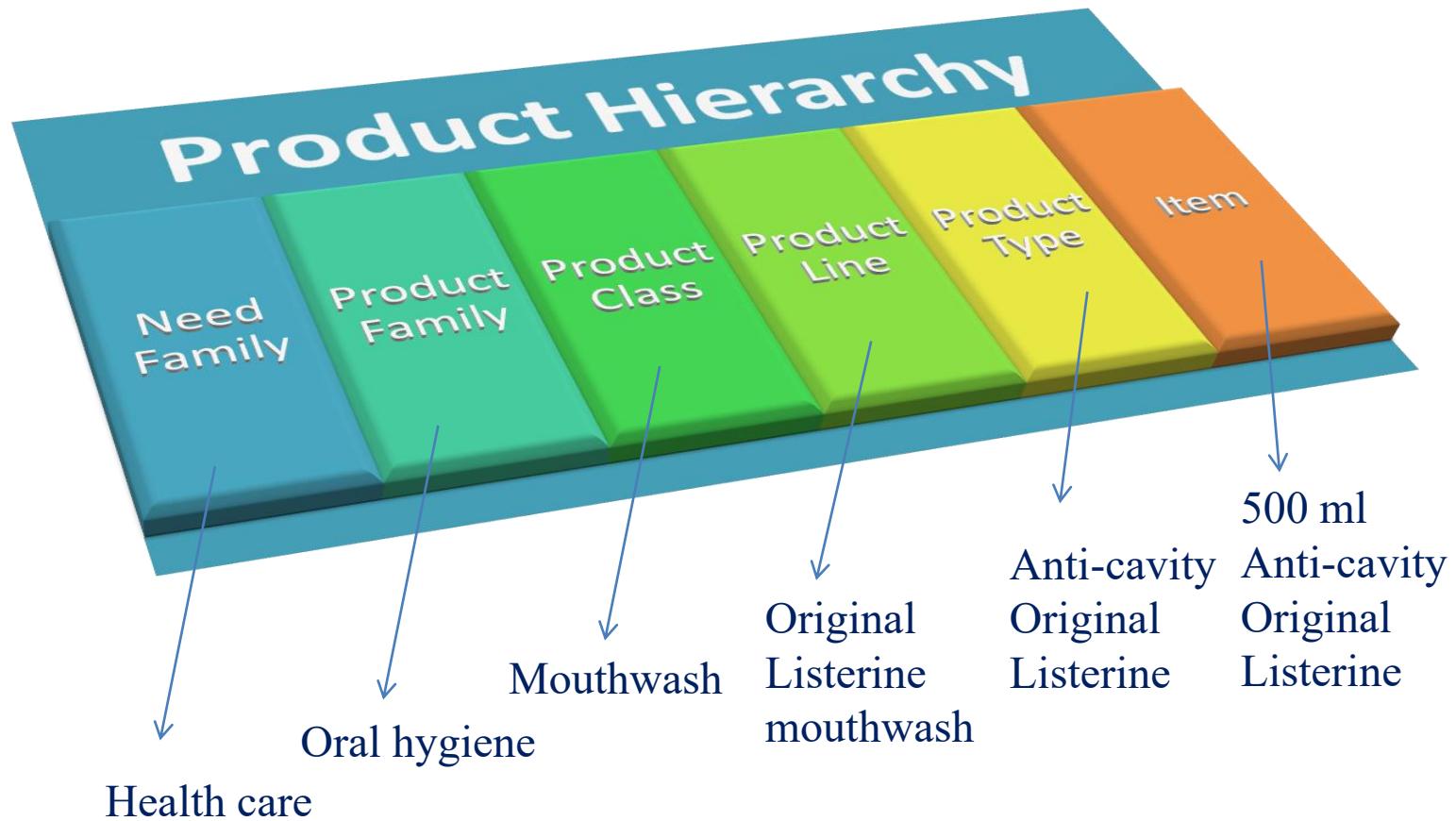
Motivation

- How do you organize or do some operations on data that are naturally **hierarchical**?
- **Example:** Ancestry



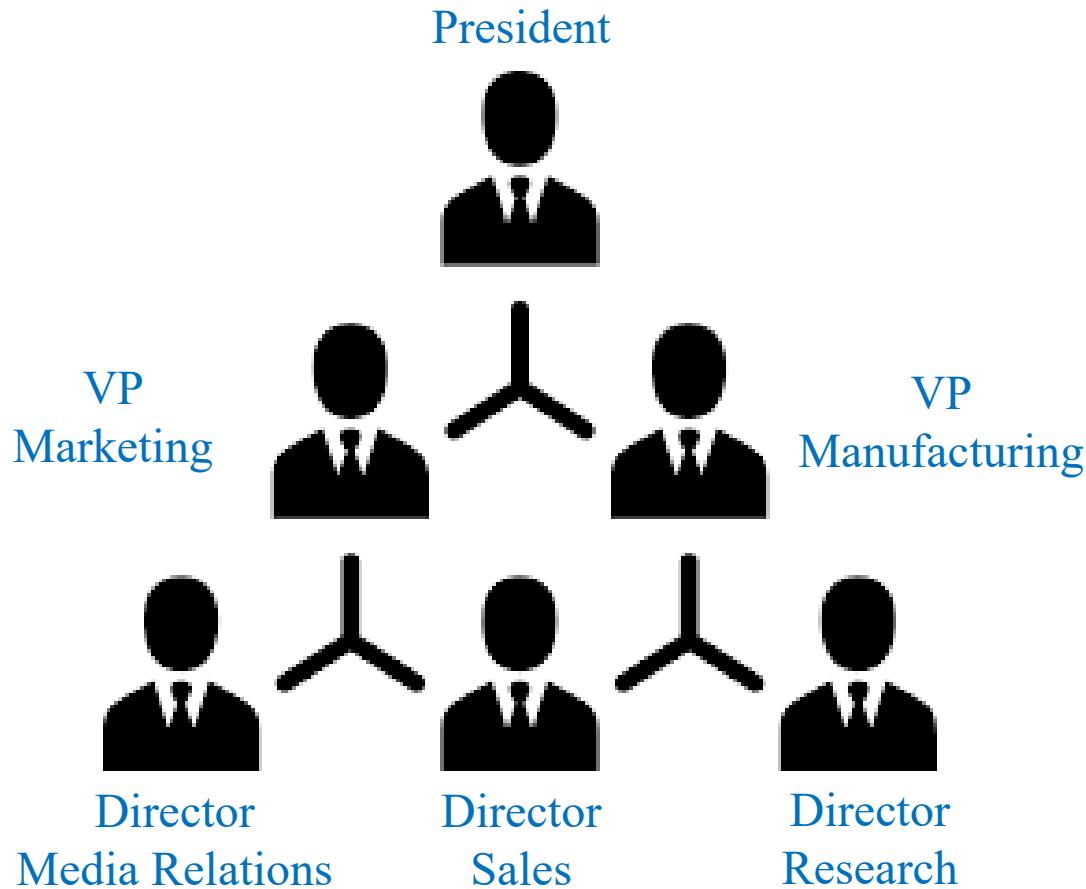
Motivation

- Example: Product hierarchy



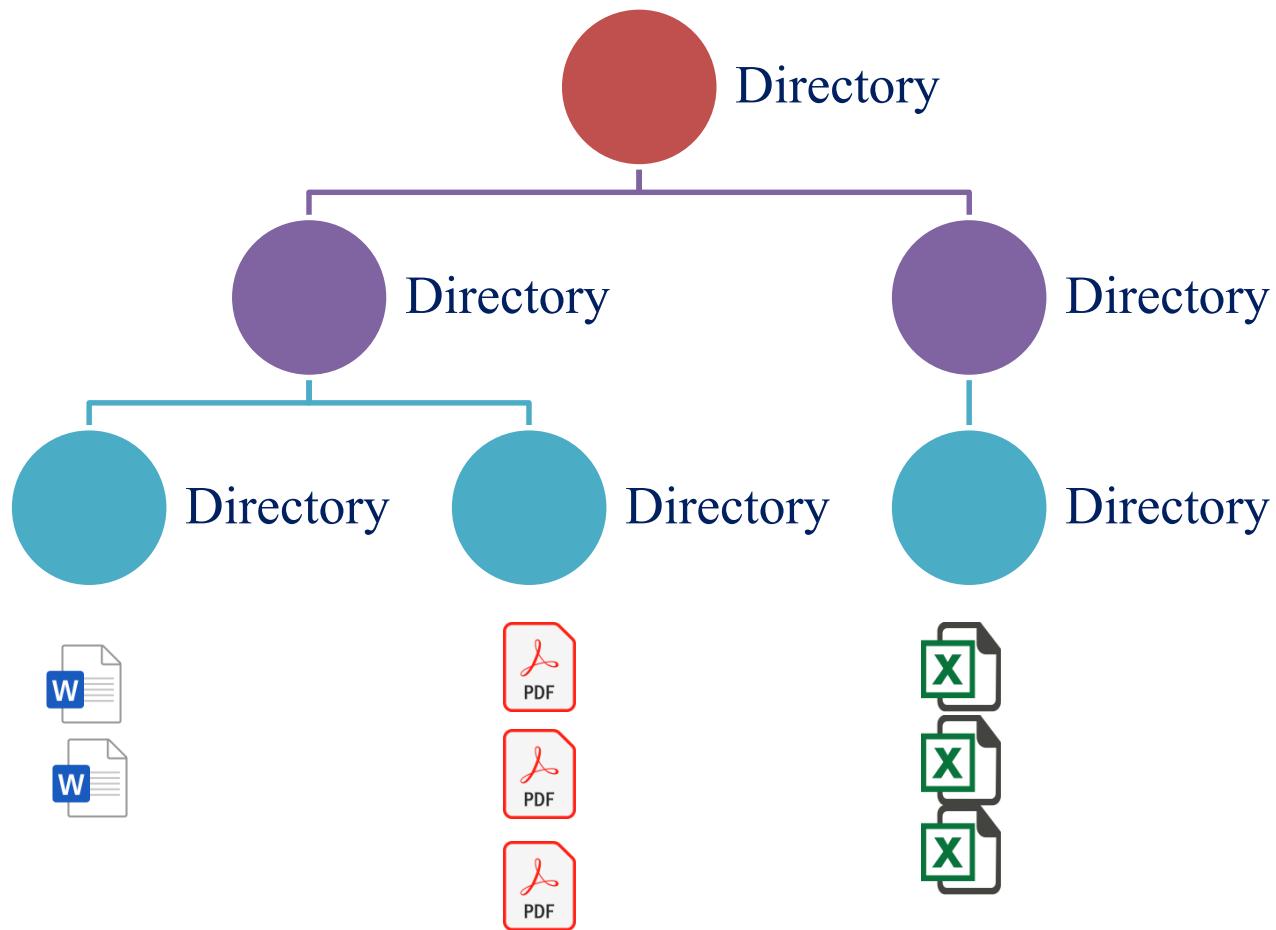
Motivation

Example: Organization hierarchy



Motivation

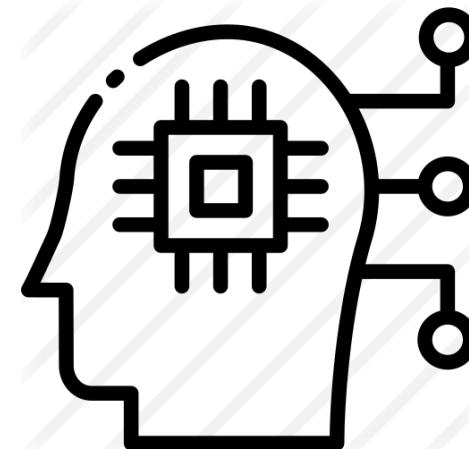
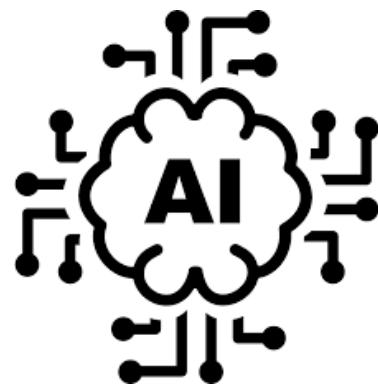
Example: Hierarchical file system structure



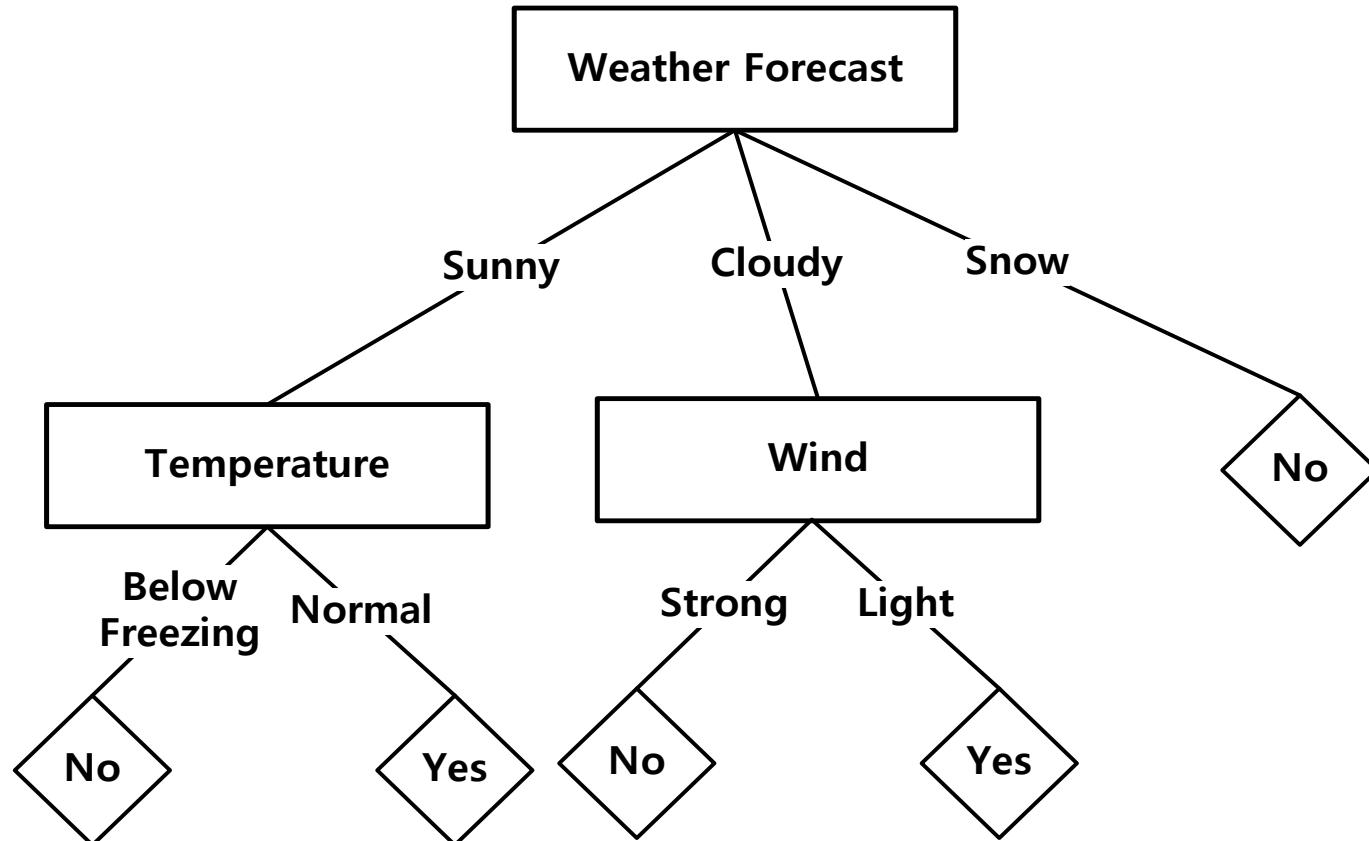
Motivation

How a **machine learning** algorithm uses data for decision tree-based learning?

In **artificial intelligence** (AI), decision trees are used to encode data regarding decision making.



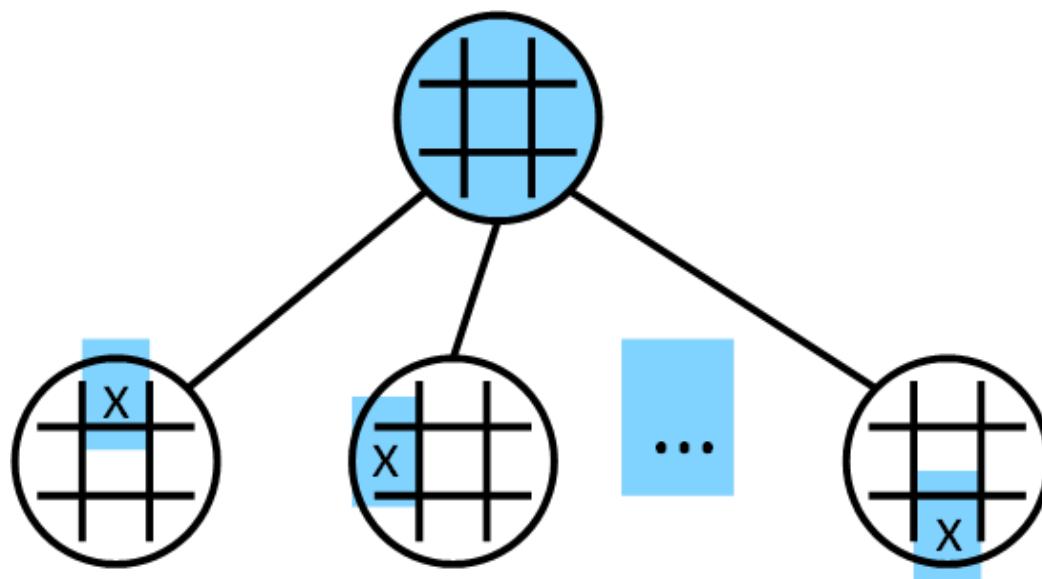
Motivation



Decision Tree: Play Badminton Outside

Motivation

- In a **game decision tree**, the edges represent possible moves or decisions while the nodes represent the possible outcomes of making a move or decision.



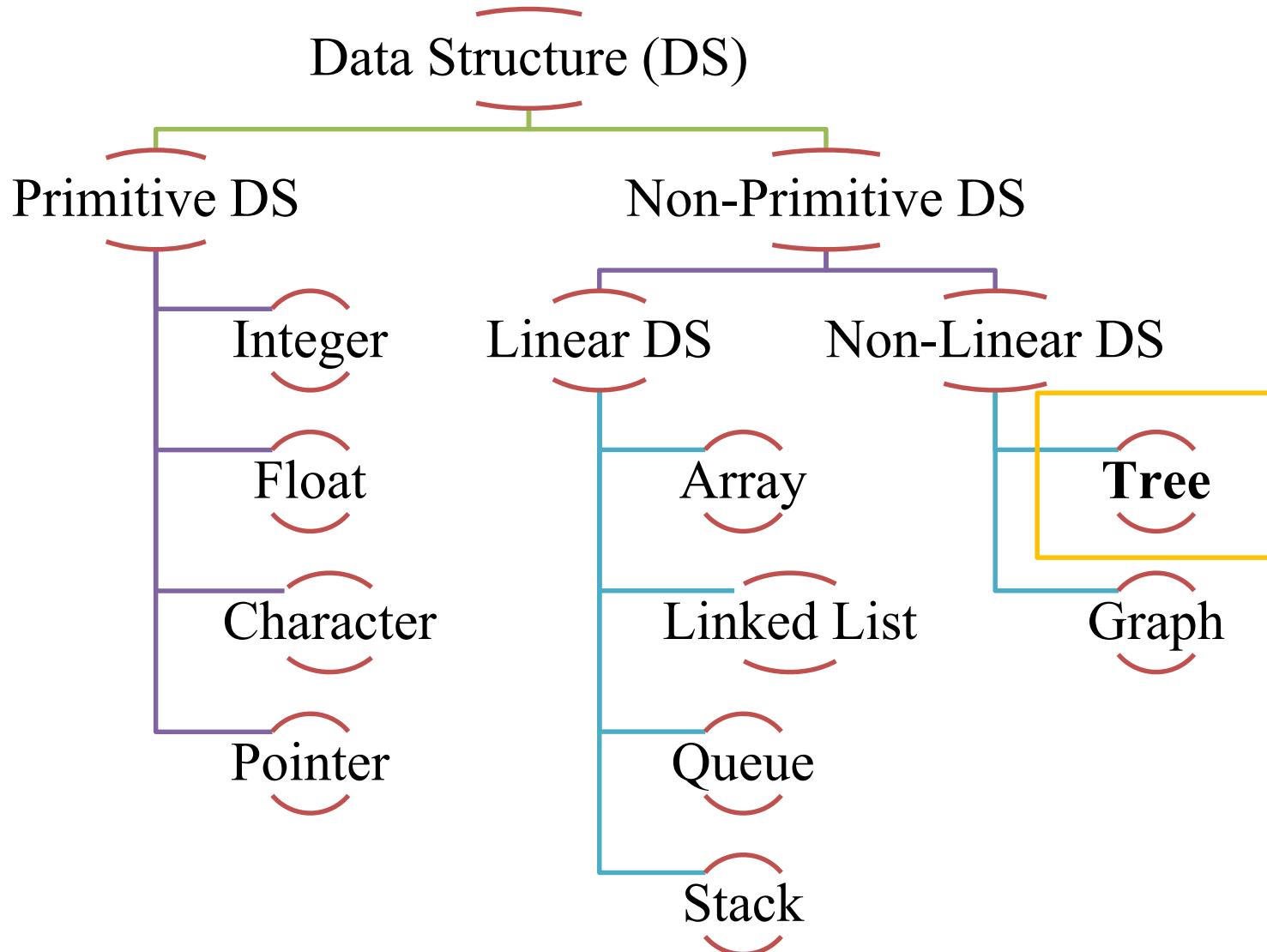
Tic-Tac-Toe Game Tree

Learning Outcomes

By the end of this lecture you will be able to:

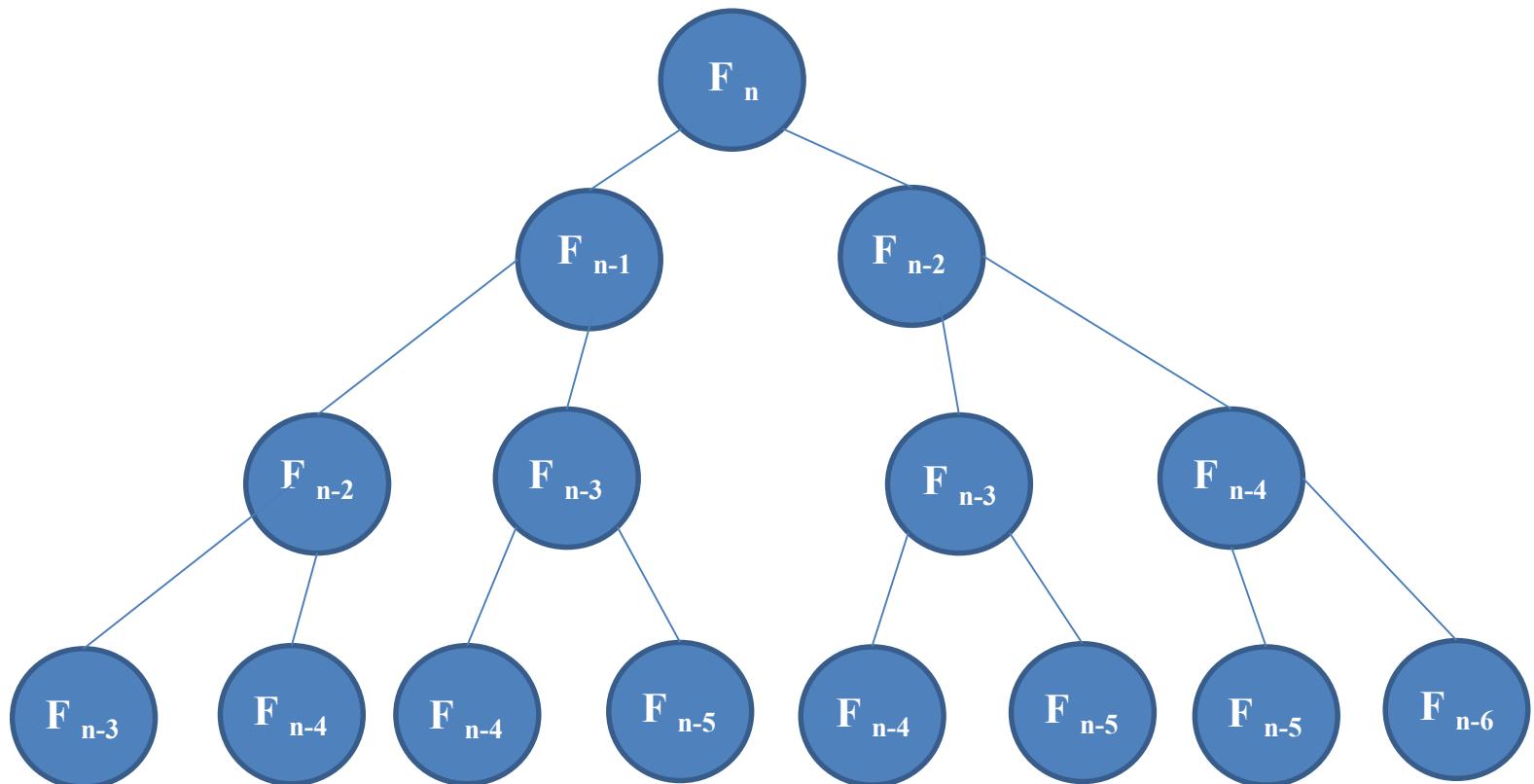
- describe a tree data structure and its components
- explain the relationships of nodes of a tree
- recognize different kinds of binary trees

Recall: Types of Data Structures



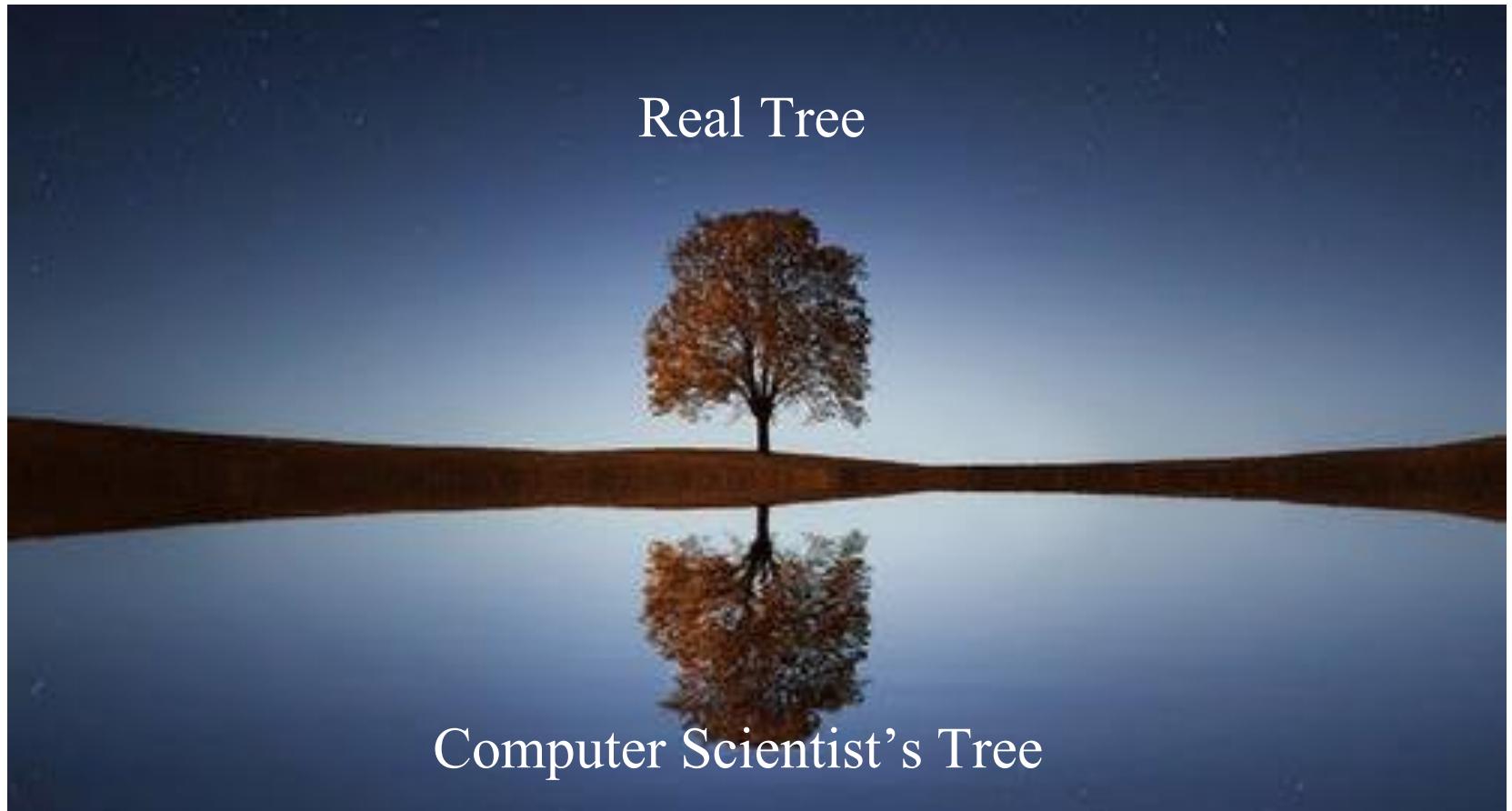
Recall: The Fibonacci Series Call Tree

We informally used call tree diagrams to represent the relationships between the calls of a recursive algorithm



$$F_0=0, F_1=1, \quad F_n = F_{n-1} + F_{n-2} \quad (n \geq 1)$$

Introduction



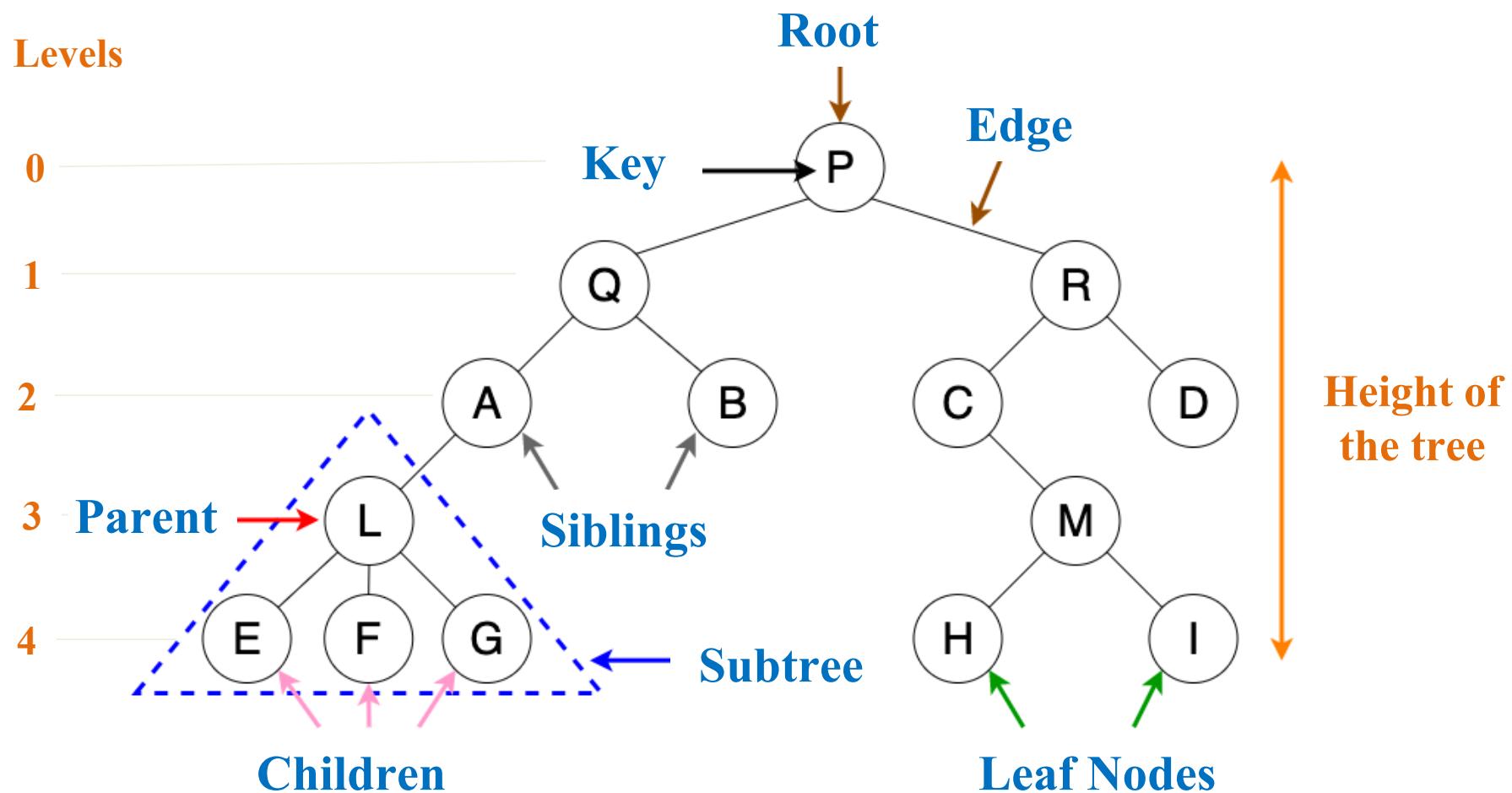
Computer Scientist's Tree

What do you see in this picture?

Basics of Tree ADT

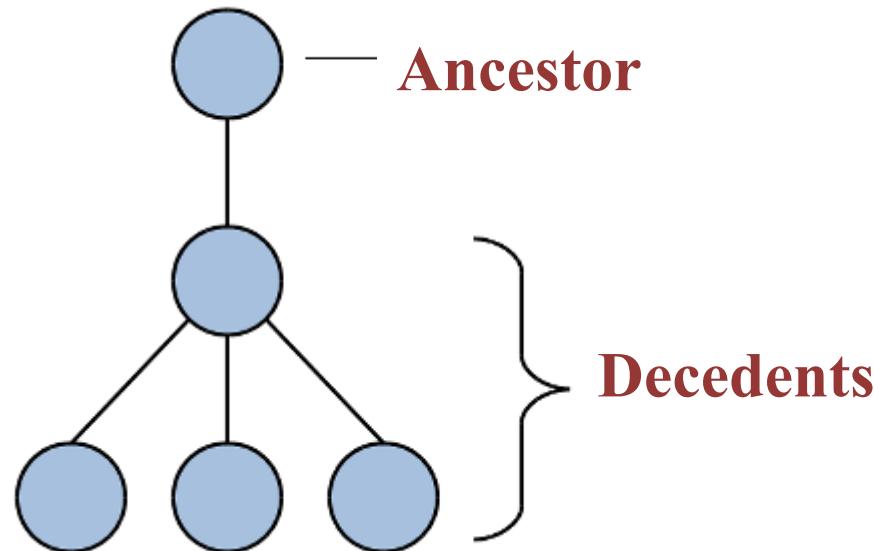
- A **hierarchical** nonlinear data structure composed of linked nodes connected by edges
- In **nonlinear** structure, the components do not form a simple sequence of first entry, second entry, third entry and so on, there is more complex linking between the components.
- There is one **root** node for the entire structure.
- Each node has zero or more nodes as its **children**.
- Each node has at most one **parent** node.

Basic Terminologies of Tree ADT



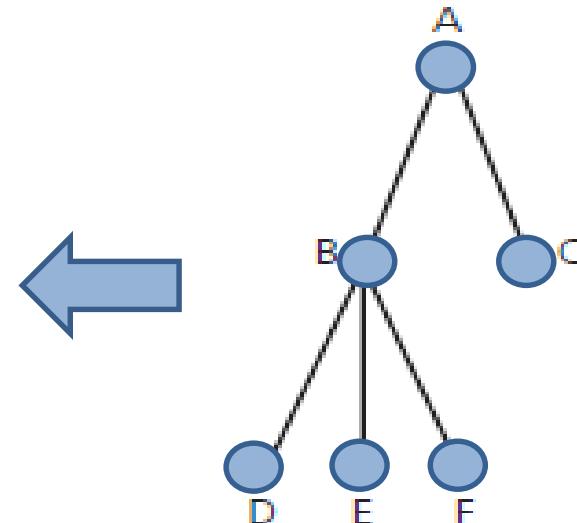
Ancestors and Decedents

- **Descendants:** All the nodes reachable from the current downwards to the bottom of the tree
- **Ancestors:** All the nodes reachable from the current upwards to the tree root
- The root of any tree is an ancestor of every node in that tree.

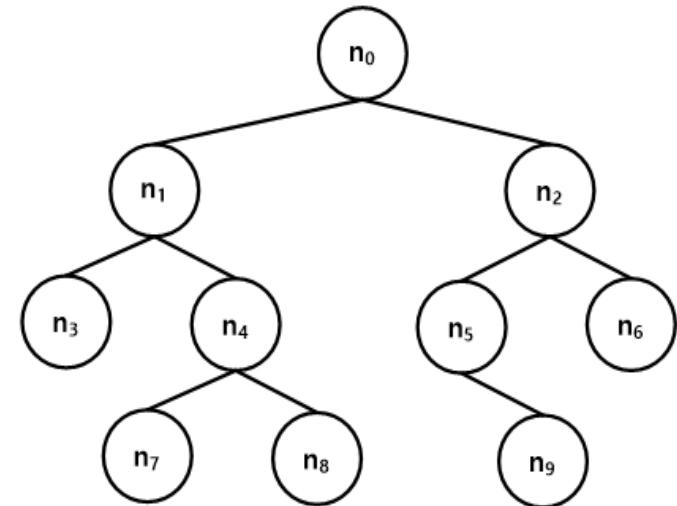
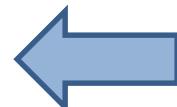


Example 1. Ancestors and Decedents

- A is an ancestor of D , and thus D is a descendant of A . Not all nodes are related by the ancestor or descendant relationship: B and C , for instance, are not so related.

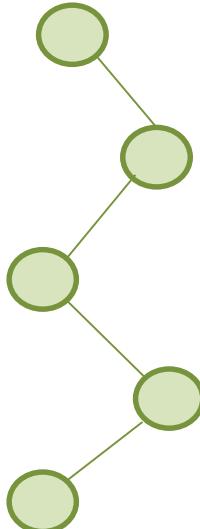


- n_1 's descendants:
 $\{n_3, n_4, n_7, n_8\}$
- n_5 's ancestors:
 $\{n_0, n_2\}$

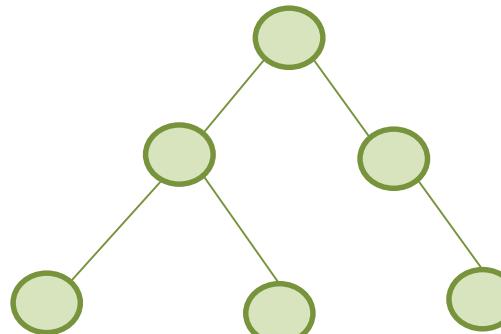


Unidirectional Relationship

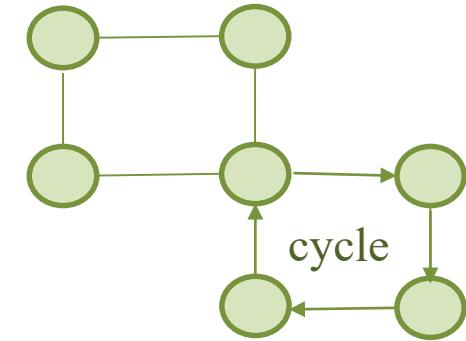
- Typically, there is unidirectional downwards relationship between node and its descendants (with no cycle).
- A node can access its descendants but descendants cannot access their ancestors.



Tree
5 nodes
4 edges

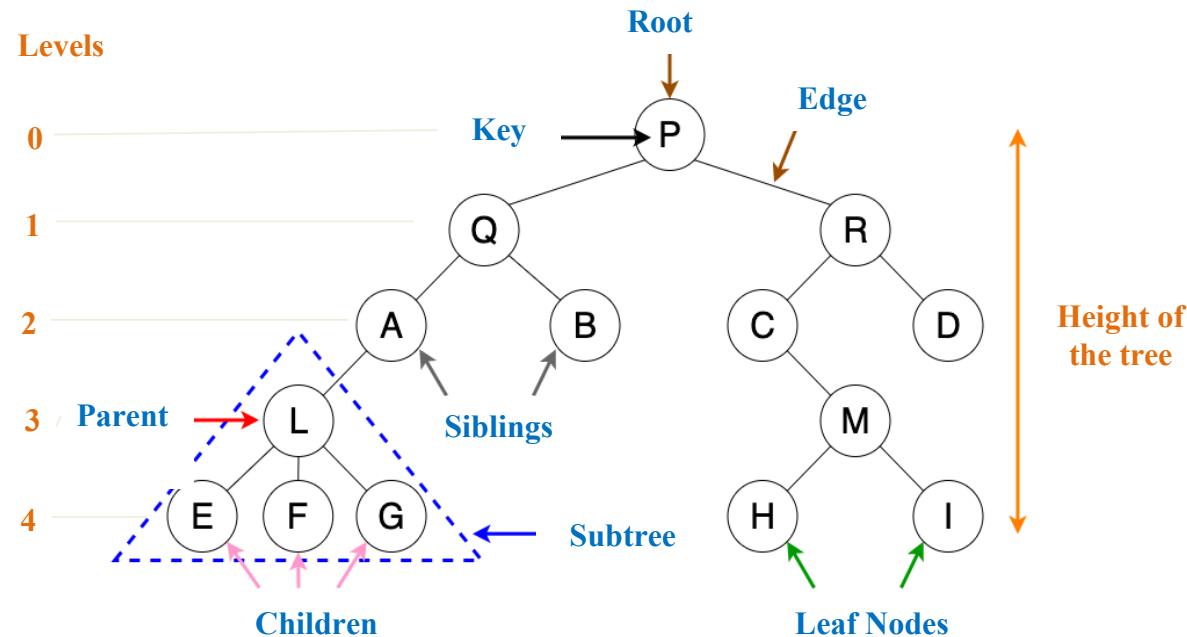


Tree
6 nodes
5 edges



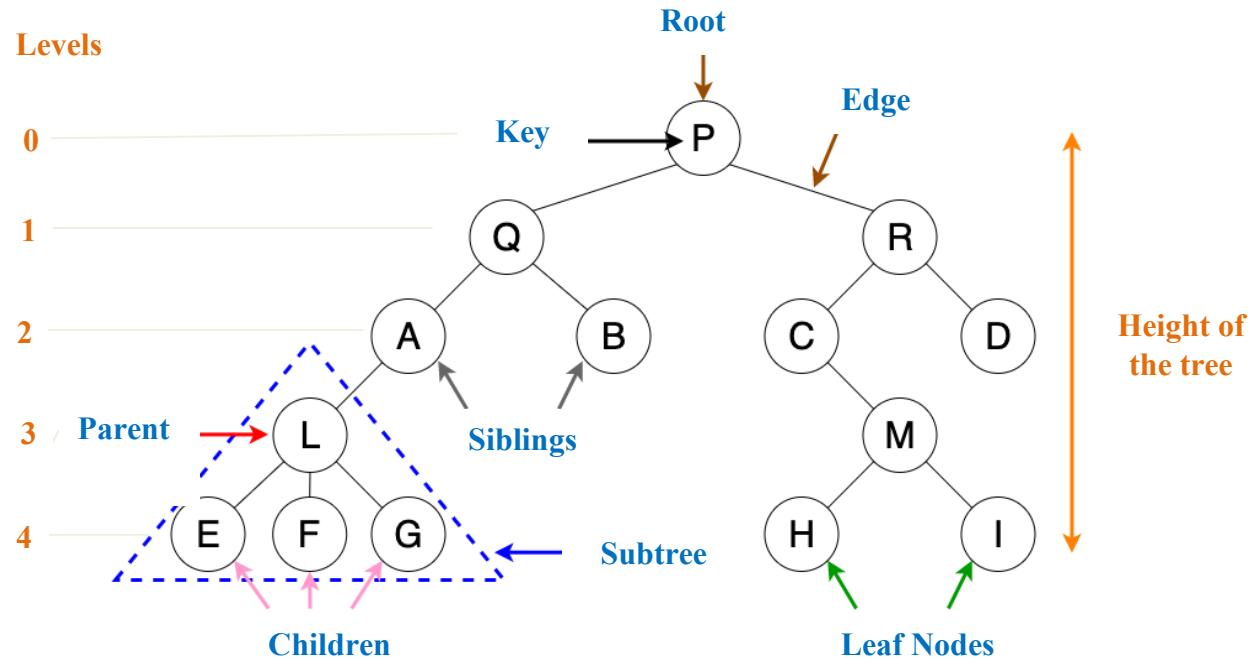
Not a Tree
7 nodes
8 edges

Subtree and Subtree of node n



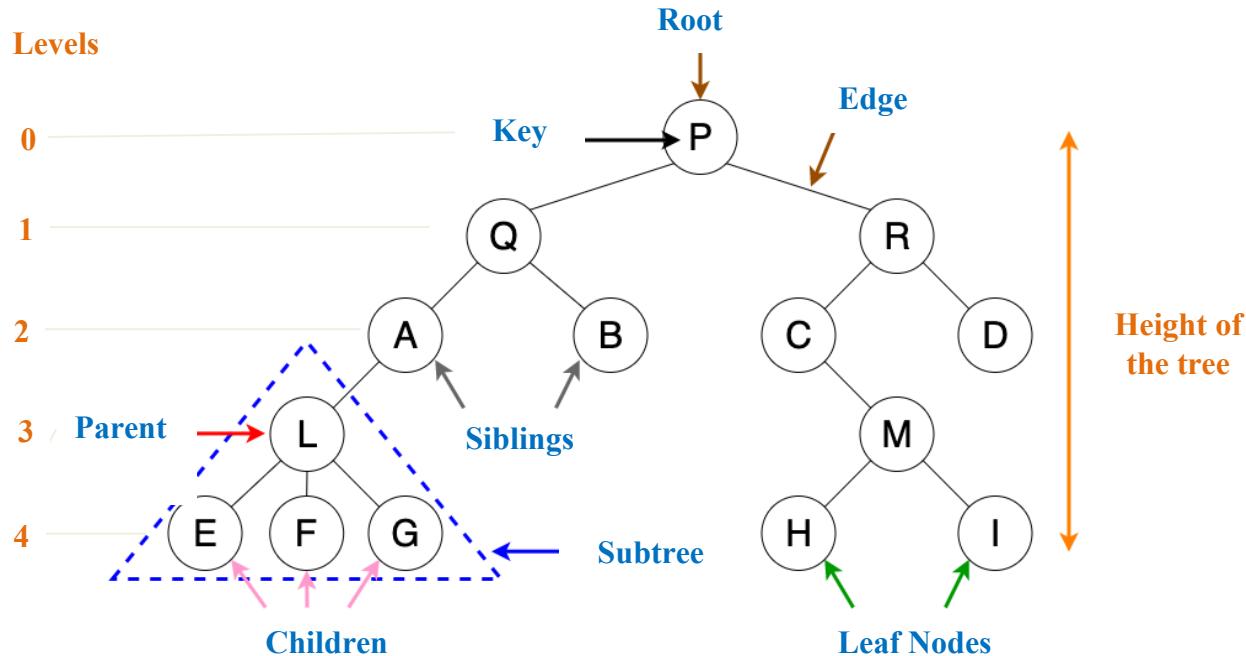
- A **subtree** in a tree is any node in the tree together with all of its descendants.
- A **subtree of a node n** is a subtree rooted at a child of n . For example, the subtree in the picture has L as its root and is a subtree of the node A .

Parent, Children and Leaf Nodes of a Tree



If an edge is between node 1 and node 2, and node 1 is above node 2 in the tree, then 1 is the **parent** of 2 , and 2 is a **child** of 1. In the above tree in nodes E, F, G are children of node L; nodes A and B are children of node Q; C and D are children of node R. A node that has no children is called a **leaf** of the tree. The leaves of the above tree are H, I, D and B.

Siblings and First Child of a Tree



Children of the same parent—for example, *A* and *B*—are called **siblings**. The **leftmost child** *E* is called the **oldest child**, or **first child**, of *L*.

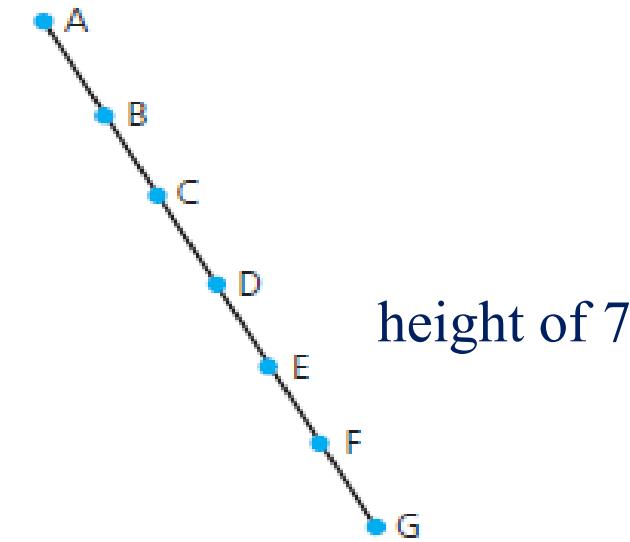
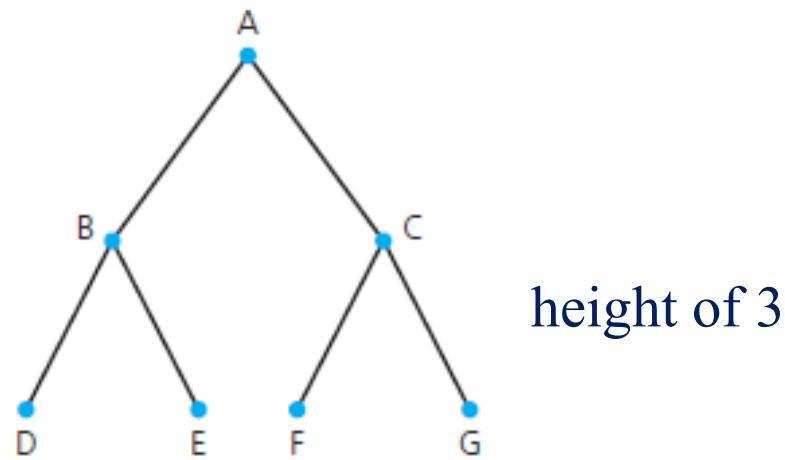
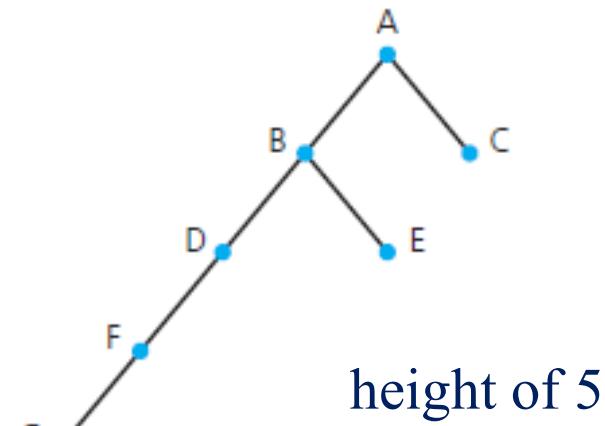
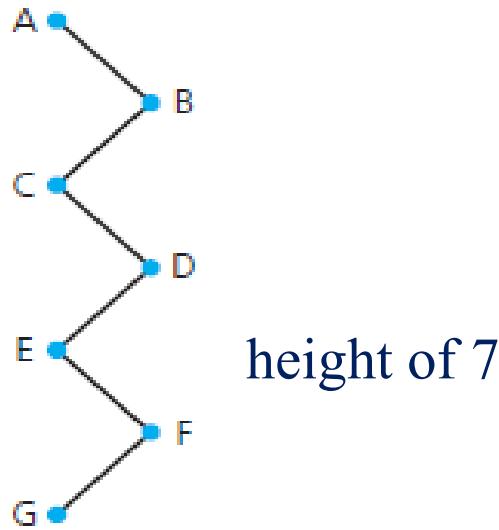
Height of a Tree and Level of a Node

Height of a tree: It is the number of nodes on the longest path from the root to a leaf.

Level of a node n :

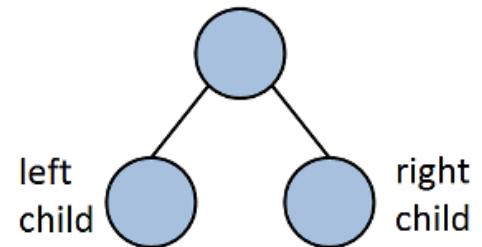
- If n is the root of the tree, it is at level 0.
- If n is not the root of the tree, its level is 1 greater than the level of its parent.

Example 2. Height of a Tree



Binary Tree: Full and Complete

- A **binary tree** is a set of nodes that is either empty or partitioned into a root node and one or two subsets that are binary subtrees of the root.
- Each node has at most two children, the left child and the right child.
- In a **full binary tree** of height h , all nodes that are at a level less than h have two children each.
- A **complete binary tree** of height h is a binary tree that is full down to level $h - 1$, with level h filled in from left to right.



Full and Complete Binary Trees

Full:

- Each node in a full binary tree has left and right subtrees of the same height.
- Among binary trees of height h , a full binary tree has as many leaves as possible, and they all are at level h .

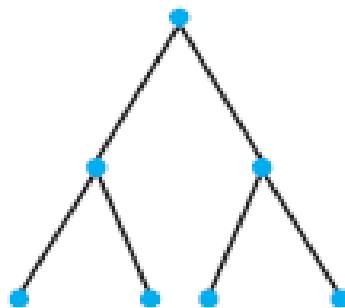
Complete:

- All nodes at level $h - 2$ and above have two children each
- When a node at level $h - 1$ has children, all nodes to its left at the same level have two children each,
- When a node at level $h - 1$ has one child, it is a left child

A full binary tree is complete.

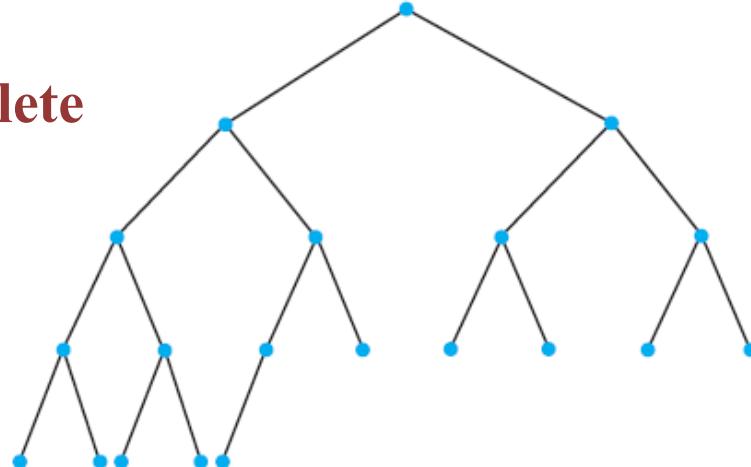
Example 3. Complete versus full binary tree

Full



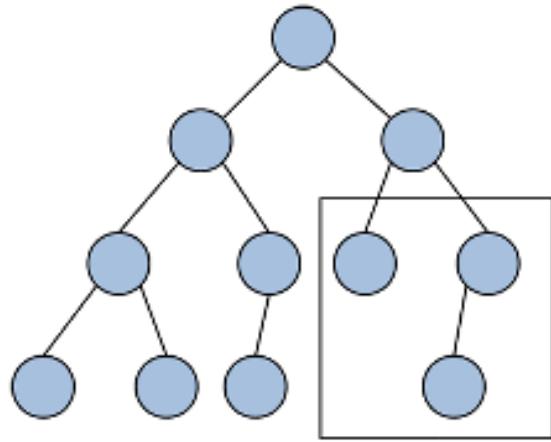
- all nodes that are at a level less than h have two children each ($h=3$).

Complete

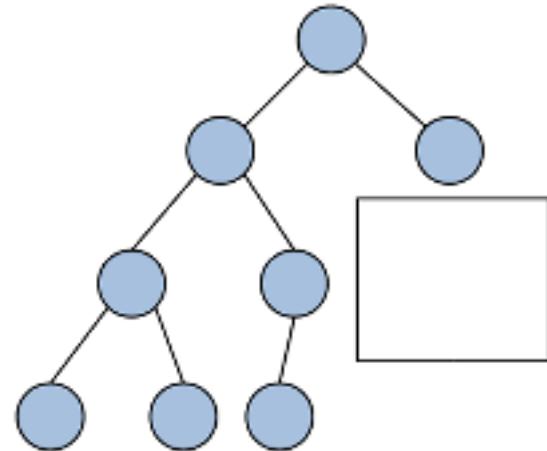


- full down to level $h - 1$, with level h filled in from left to right ($h=5$).
- completely filled at all levels
- all leaf nodes are as far left as possible

Example 4. Incomplete binary trees



The leaf nodes on the bottom-most level are not as far left as possible.



Not all levels above the bottom-most level are filled.

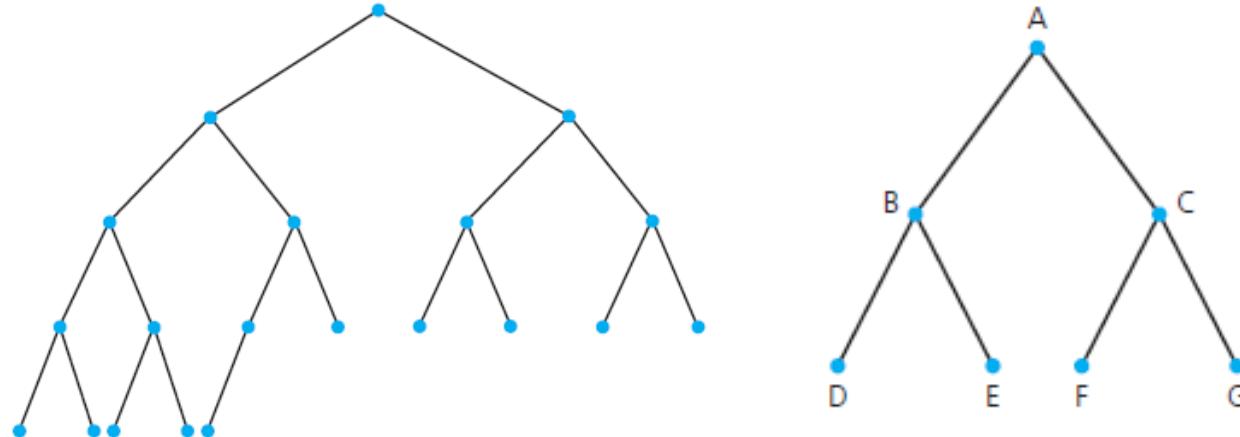
Balanced Binary Tree

- A binary tree is **balanced**, if the height of any node's right subtree differs from the height of the node's left subtree by no more than 1.
- A complete binary tree is balanced.

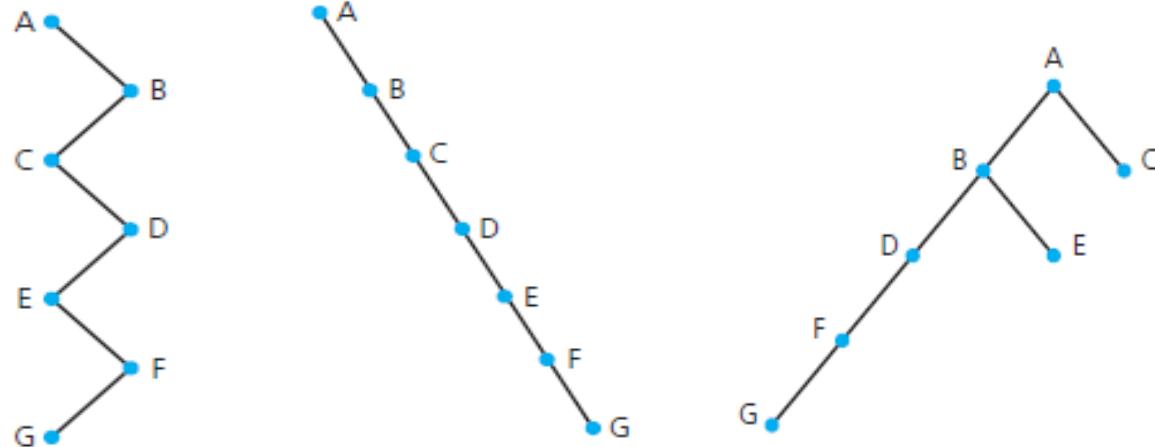
Example 4. Balanced/unbalanced binary trees

- **Balanced:** the height of any node's right subtree differs from the height of the node's left subtree by no more than 1.

Balanced:



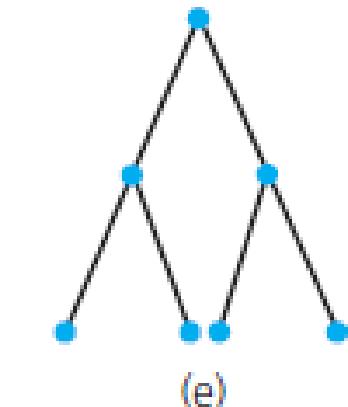
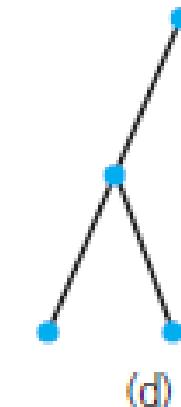
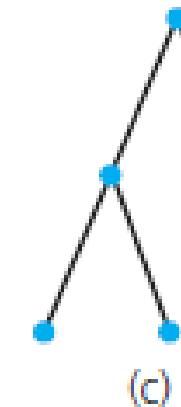
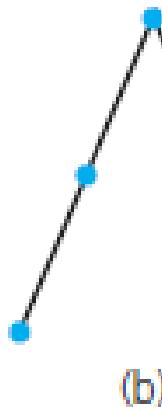
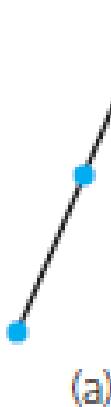
Unbalanced:



You Try 1.Type of Binary Tree



All the following binary trees have height of $h=3$. Specify the number of nodes, edges, and identify which ones are complete, which ones are full, and which ones are balanced?



You Try 2. Tree terminology



Match the description with appropriate option.

- | | |
|--|-----------|
| 1. The distance of a node from the root | Leaf node |
| 2. The top node of a tree structure; a node with no parent | Height |
| 3. The maximum level | Level |
| 4. A tree node that has no children | Root |

You Try 3. Type of Tree



“A structure with a unique starting node, in which each node is capable of having two child nodes, and in which a unique path exists from the root to every other node.”

The above statement, is the most appropriate for which of the following trees?

- 1) Complete tree
- 2) Binary tree
- 3) Full tree
- 4) Balanced tree

Next Lecture

We focus on:

- the maximum and minimum height of a binary tree
- binary tree ADT
- traversals of a binary tree: pre-order, in-order, post-order
- binary tree operations

Readings/ Study Reference

Course Textbook: “*Data Structure and Algorithms in a Nutshell*” A. Wong et al.

Chapter 7. Trees

Section 7.1 Tree Terminology

Section 7.2. Binary Trees (up to 7.2.2)

The End of Lecture

Course: Data Structures and Algorithms

Instructor: Homeyra Pourmohammadali

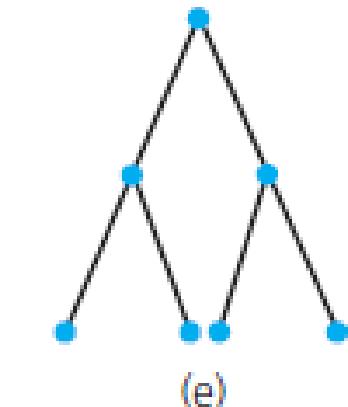
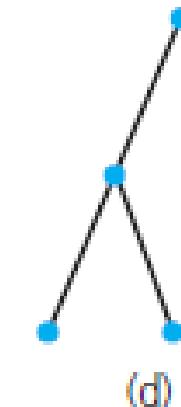
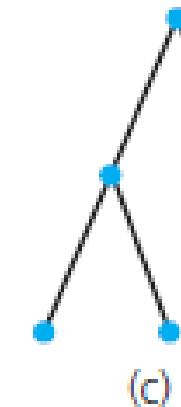
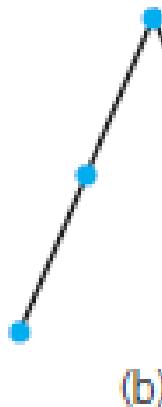
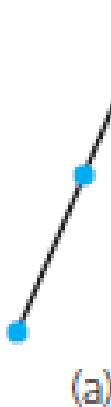
You Try Questions and Solutions

Trees: Basic Terminologies and Examples

You Try 1.Type of Binary Tree



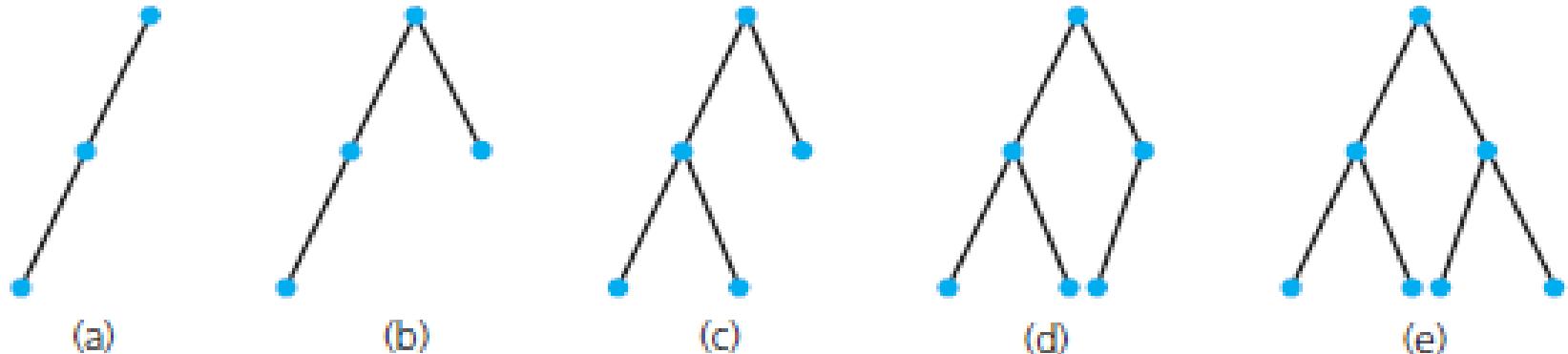
All the following binary trees have height of $h=3$. Specify the number of nodes, edges, and identify which ones are complete, which ones are full, and which ones are balanced?



You Try 1 Solution. Type of a Binary Tree

Full tree: all nodes except leaf nodes should have two children, every node has 0 or 2 children.

Complete tree: all the levels are completely filled except possibly the last level and the last level has all keys as left as possible.



incomplete

not full

unbalanced

complete

not full

balanced

complete

full

balanced

complete

not full

balanced

complete

full

balanced

You Try 2. Tree terminology



Match the description with appropriate option.

- | | |
|--|-----------|
| 1. The distance of a node from the root | Leaf node |
| 2. The top node of a tree structure; a node with no parent | Height |
| 3. The maximum level | Level |
| 4. A tree node that has no children | Root |

You Try 2 Solution. Tree terminology

Match the description with appropriate option.

1. The distance of a node from the root

2. The top node of a tree structure; a node with no parent

3. The maximum level

4. A tree node that has no children

Leaf node

Height

Level

Root

You Try 3. Type of Tree



“A structure with a unique starting node, in which each node is capable of having two child nodes, and in which a unique path exists from the root to every other node.”

The above statement, is the most appropriate for which of the following trees?

- 1) Complete tree
- 2) Binary tree
- 3) Full tree
- 4) Balanced tree

You Try 3 Solution. Type of Tree



“A structure with a unique starting node, in which each node is capable of having two child nodes, and in which a unique path exists from the root to every other node.”

The above statement, is the most appropriate for which of the following trees?

- 1) Complete tree
- 2) Binary tree
- 3) Full tree
- 4) Balanced tree

The End of You Try Activities

Course: Data Structures and Algorithms

Instructor: Homeyra Pourmohammadali

Binary Tree Traversal

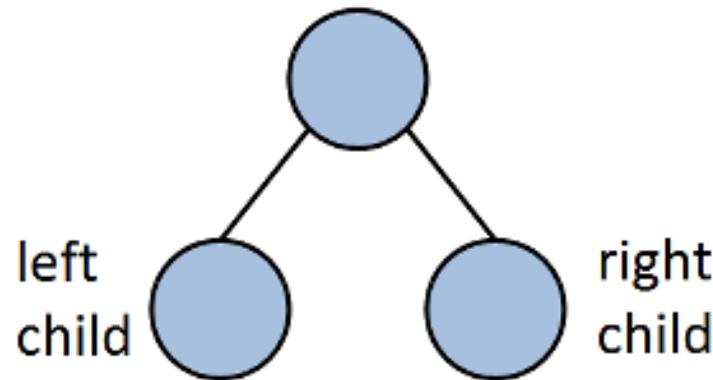
Learning Outcomes

By the end of this lecture you will be able to:

- find out how a complete binary tree can be indexed.
- see the interface for binary tree abstract data type.
- find out how traversal is done in binary tree, and what are different forms of traversal of a binary tree

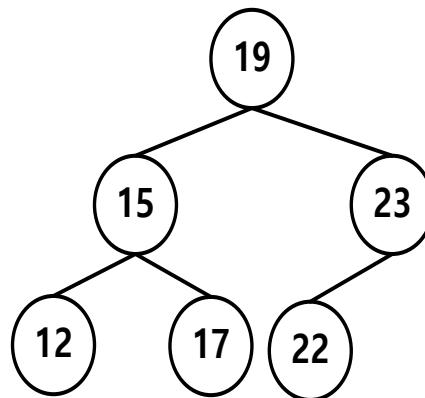
Recall: Binary Tree

- A **binary tree** is a set of nodes that is either empty or partitioned into a root node and one or two subsets that are binary subtrees of the root.
- Each node has at most two children, the left child and the right child.

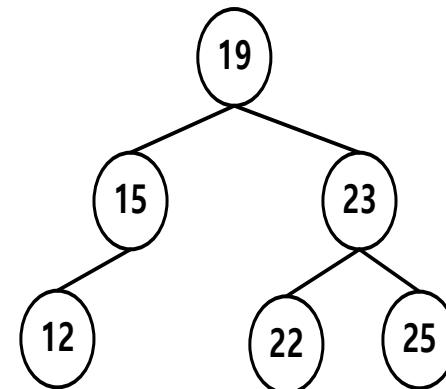


Recall: Complete Binary Tree

- A **complete binary tree** of height h is a binary tree that is full down to level $h - 1$, with level h filled in from left to right.
- A binary tree that is completely filled at all levels with the exception of the lowest level. At the lowest level, all leaf nodes are as far left as possible.



Complete Binary Tree



Not a Complete Tree

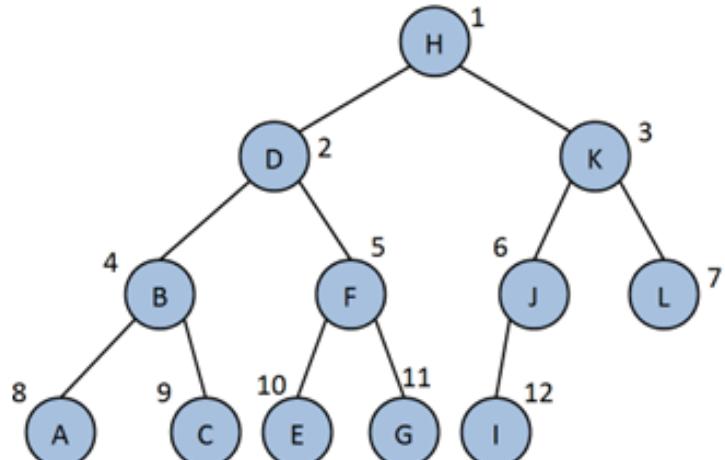
Complete Binary Tree

What is so special about complete binary trees compared to any other arbitrary tree?



Complete Binary Tree Indexing

- The nodes of complete tree are contiguous if we scan the tree from left to right.
- This allows for efficient sequential representation.
- Root, leaf, parent and children of any nodes can be accessed using simple mathematical indexing.



	H	D	K	B	F	J	L	A	C	E	G	I
0	1	2	3	4	5	6	7	8	9	10	11	12

Complete Binary Tree: Access Formulas

- **Root node:** array index 1
- **Parent of node i:** floor($i/2$)
- **Left child of node i:** $2i$
- **Right child of node i:** $2i + 1$
- **Is node i a leaf:** check if $n < 2i$

Example 1. Complete Binary Tree Indexing

Parent of node C ($i = 9$):

$$[i/2] = [9/2] = [4.5] = 4 \rightarrow B$$

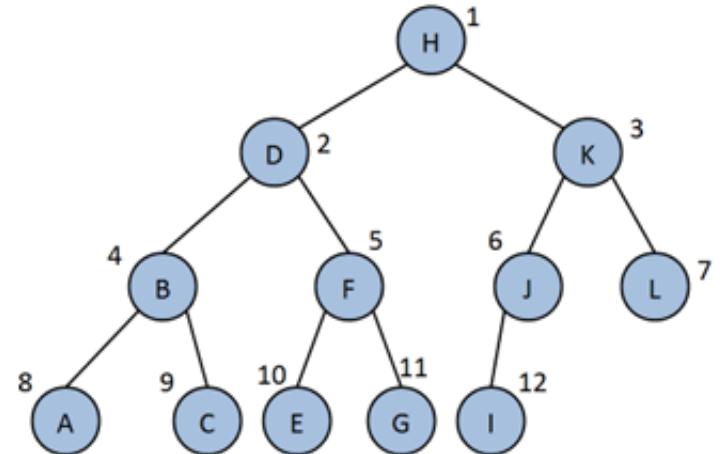
Right child of node F ($i = 5$):

$$[2i + 1] = [2(5) + 1] = [11] = 11 \rightarrow G$$

Leaf nodes of the tree ($n = 12$ nodes):

$$2i > n \rightarrow i > \left\lceil \frac{12}{2} \right\rceil = i > 6$$

$\rightarrow L, A, C, E, G, I$

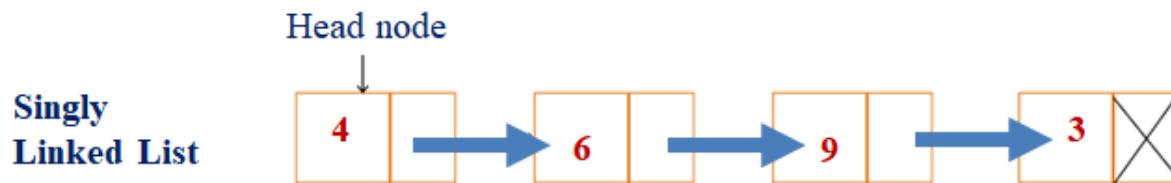


Complete Binary Tree: Access Formulas

- **Root node:** array index 1
- **Parent of node i :** floor($i/2$)
- **Left child of node i :** $2i$
- **Right child of node i :** $2i + 1$
- **Is node i a leaf:** check if $n < 2i$

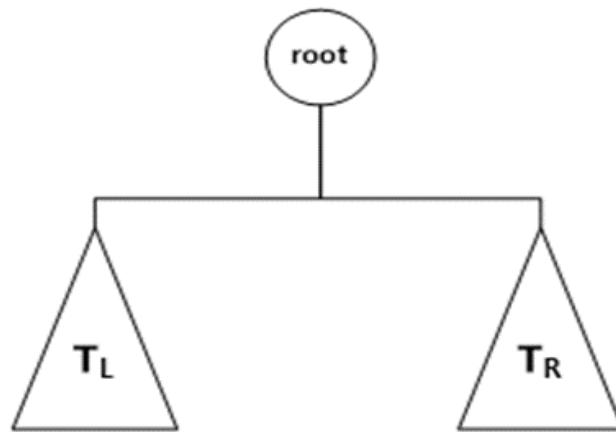
The Binary Tree ADT

- The binary tree operations: add and remove nodes, set or retrieve the data in the root of the tree, and test whether the tree is empty, and traversal (visiting every node in a binary tree).
- Visiting a node means “doing something with or to” the node.
- Recall: in traversing a linked list, each node is visited sequentially till the end of linked list, but in trees it is different.



Design a Binary Tree

Design: A finite set of nodes that is either empty or that consists of a root and two disjoint binary trees, left subtree (T_L) and right subtree (T_R)



Node Depth: The length of the path from the root to the current node; this path is quantified by counting the edges

Binary Tree Node ADT Interface

```
class BinaryTreeNode // Class for binary tree node
{
public:
    BinaryTreeNode(); // default constructor
    ~BinaryTreeNode(); // destructor

    BinaryTreeNode left(); // returns left child
    BinaryTreeNode right(); // returns right child

private:
    int iData; // holds the data value at this tree node
    BinaryTreeNode* leftChild; // points to left child
    BinaryTreeNode* rightChild; // points to right child
};
```

Pseudocode of Binary Tree (T) Traversal

The general form of a recursive traversal algorithm:

if (T is not empty)

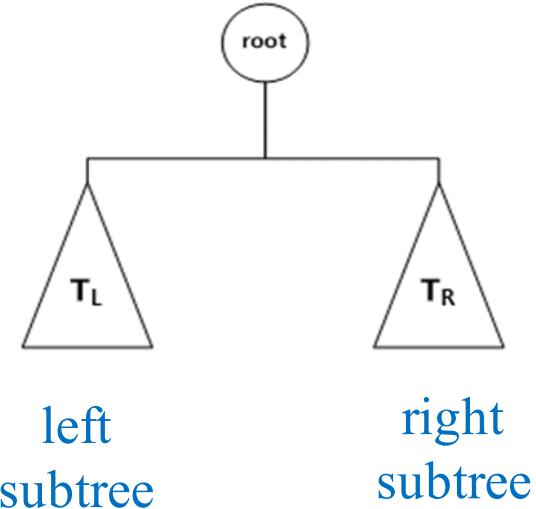
{

Display the data in T 's root

Traverse T 's left subtree

Traverse T 's right subtree

}

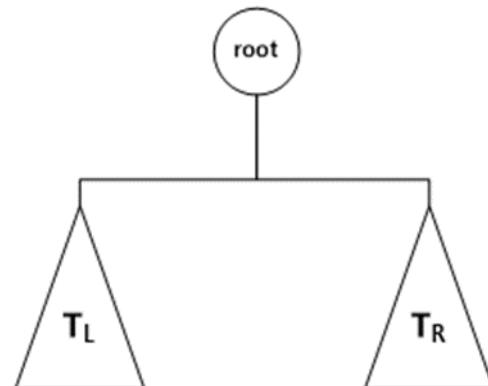


No action, if T is empty (base case).

Binary Tree (T) Traversal

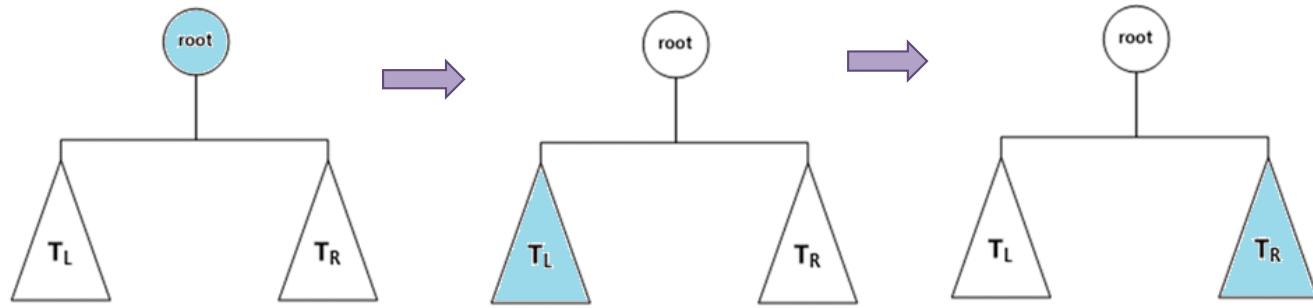
Three choices when visiting r :

- It can visit r before it traverses both of r 's subtrees
- It can visit r after it has traversed r 's left subtree T_L but before it traverses r 's right subtree T_R
- It can visit r after it has traversed both of r 's subtrees
- These choices result in **pre-order**, **in-order**, and **post-order** traversals, respectively.

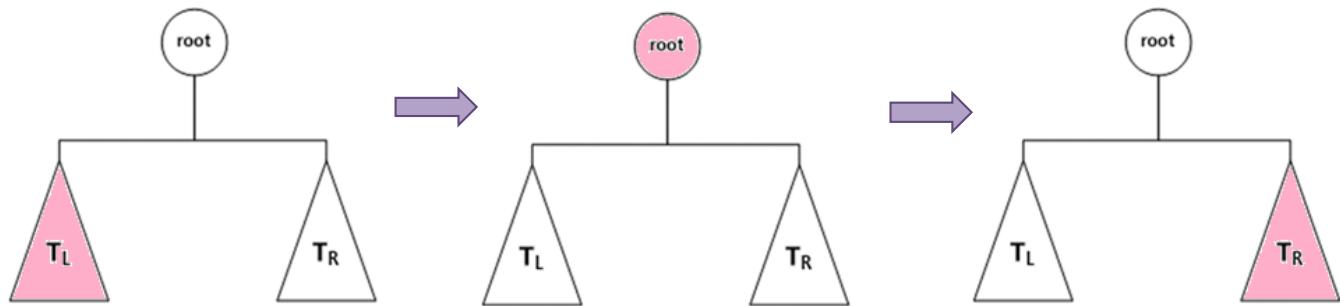


Traversing Binary Trees

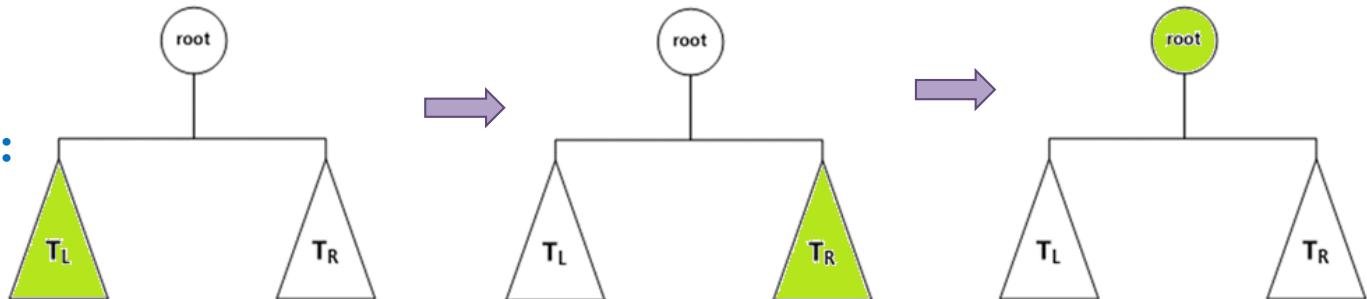
Pre-Order:



In-Order:



Post-Order:



Example 2. Pre-Order, In-Order, and Post-Order Traversal

Pre-Order:

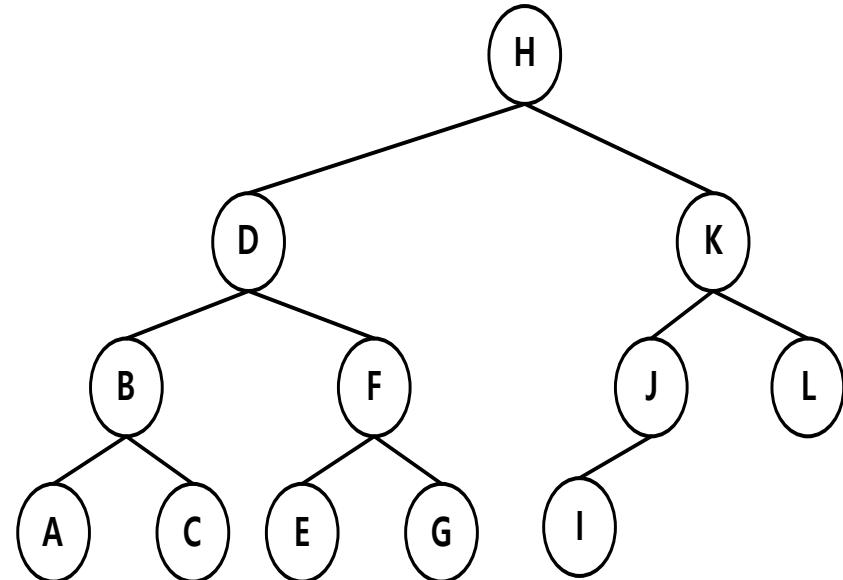
root → left subtree → right subtree

H D B A C F E G K J I L

In-Order:

left subtree → root → right subtree

A B C D E F G H I J K L

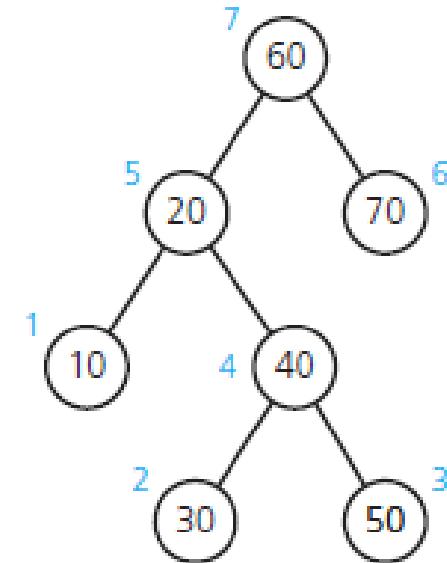
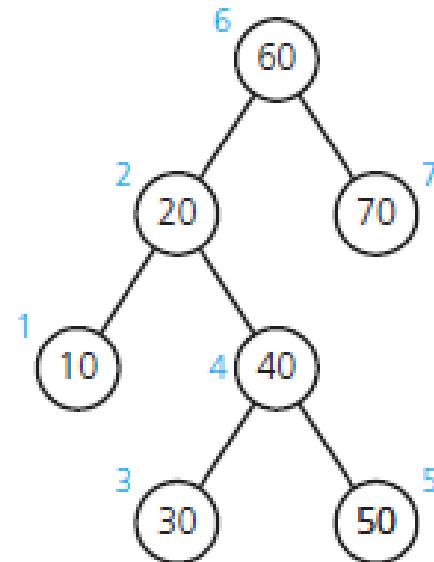
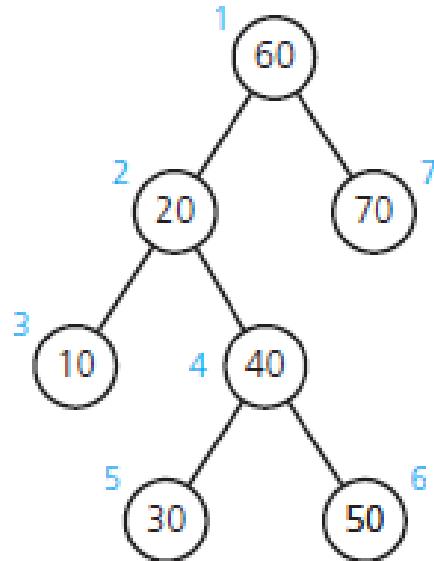


Post-Order:

left subtree → right subtree → root

A C B E G F D I J L K H

Example 3. Pre-Order, In-Order, and Post-Order Traversals



Pre-Order

$r \rightarrow T_L \rightarrow T_R$

60, 20, 10, 40, 30, 50, 70

In-Order

$T_L \rightarrow r \rightarrow T_R$

10, 20, 30, 40, 50, 60, 70

Post-Order

$T_L \rightarrow T_R \rightarrow r$

10, 30, 50, 40, 20, 70, 60

You Try 1. Post-Order Traversal



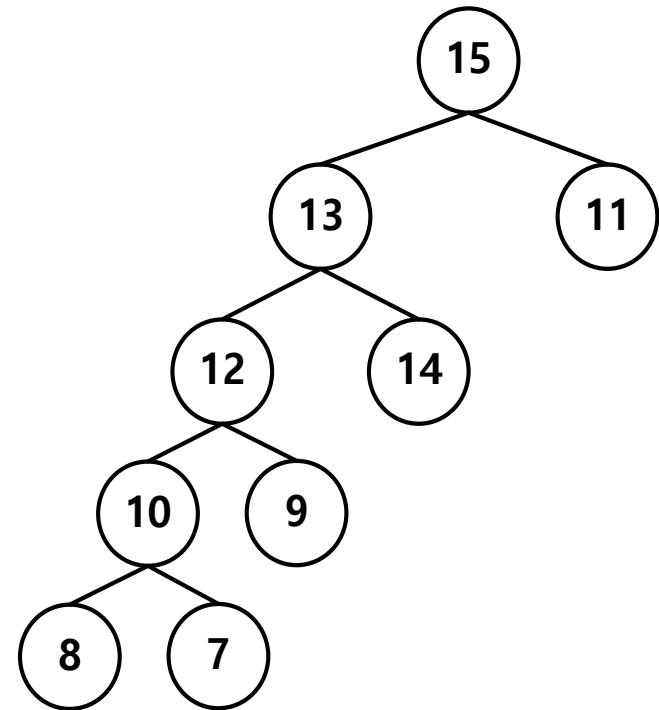
Choose the correct post-order of this binary tree.

(left subtree → right subtree → root)

a) 15 13 11 12 14 10 9 8 7

b) 8 10 7 12 9 13 14 11 15

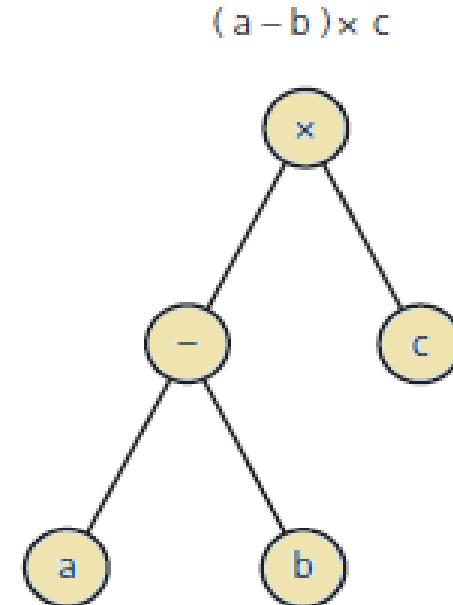
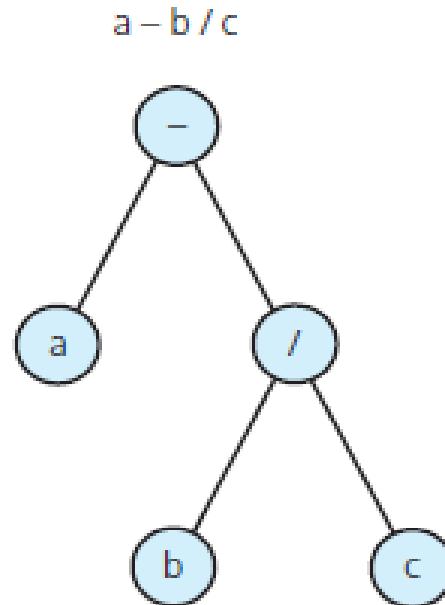
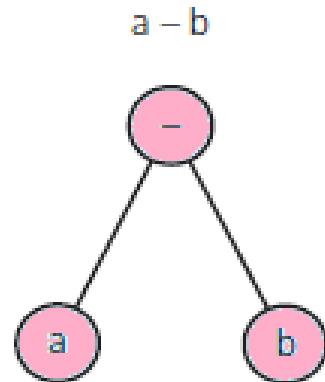
c) 8 7 10 9 12 14 13 11 15



Find pre-order and in-order.

Example 4. Binary Trees of Algebraic Expressions

Apply **post-order** traversal ($T_L \rightarrow T_R \rightarrow r$) to the following binary trees and display the nodes as you visit them. Show that the post-fix form of the expression is obtained.



a b -

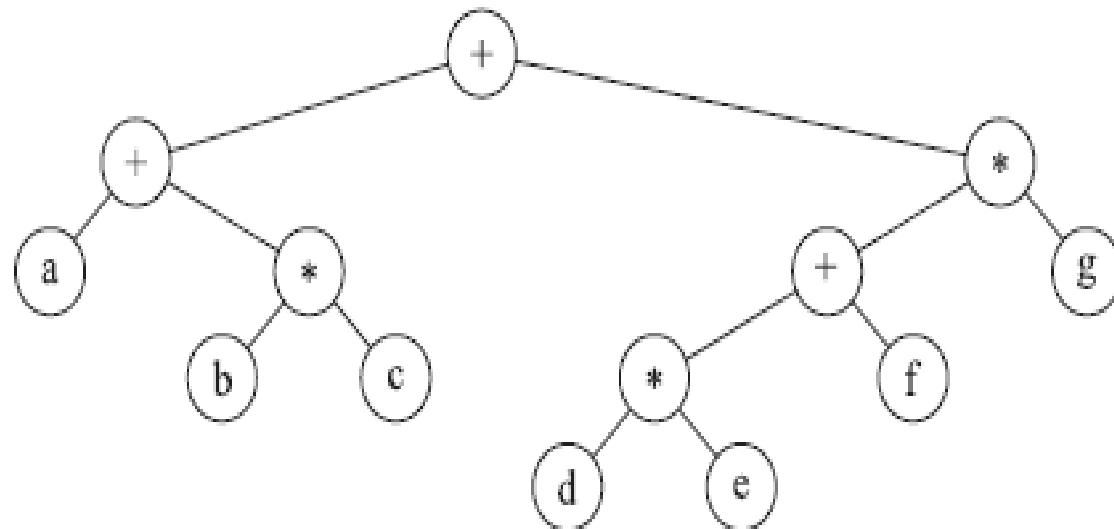
a b c / -

a b - c ×



You Try 2. Binary Trees of Algebraic Expressions

Apply pre-order ($r \rightarrow T_L \rightarrow T_R$) and post-order ($T_L \rightarrow T_R \rightarrow r$) traversals to the following binary tree and obtain the pre-fix and post-fix forms of the mathematical expression.



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

Big-O of Traversal

- Each of these traversals visits every node in a binary tree exactly once; thus n visits occur for a tree of n nodes.
- Each **visit** performs the same operations on each node, independently of n , so it must be $O(1)$.
- Thus, each **traversal** is $O(n)$.

Algorithm: Pre-Order

- Input: `BinaryTreeNode* T`
 - Output: Pre-order traversal of tree node data
 - Steps:
-

```
void pre_order(BinaryTreeNode* T) {  
    if (T == NULL) return;  
    visit(T->data); // print or other processing  
    pre_order(T->left);  
    pre_order(T->right);  
}
```

Algorithm: In-Order

- Input: `BinaryTreeNode* T`
 - Output: In-order traversal of tree node data
 - Steps:
-

```
void in_order(BinaryTreeNode* T) {  
    if (T == NULL) return;  
  
    in_order(T->left);  
  
    visit(T->data); // print or other processing  
  
    in_order(T->right);  
}
```

Algorithm: Post-Order

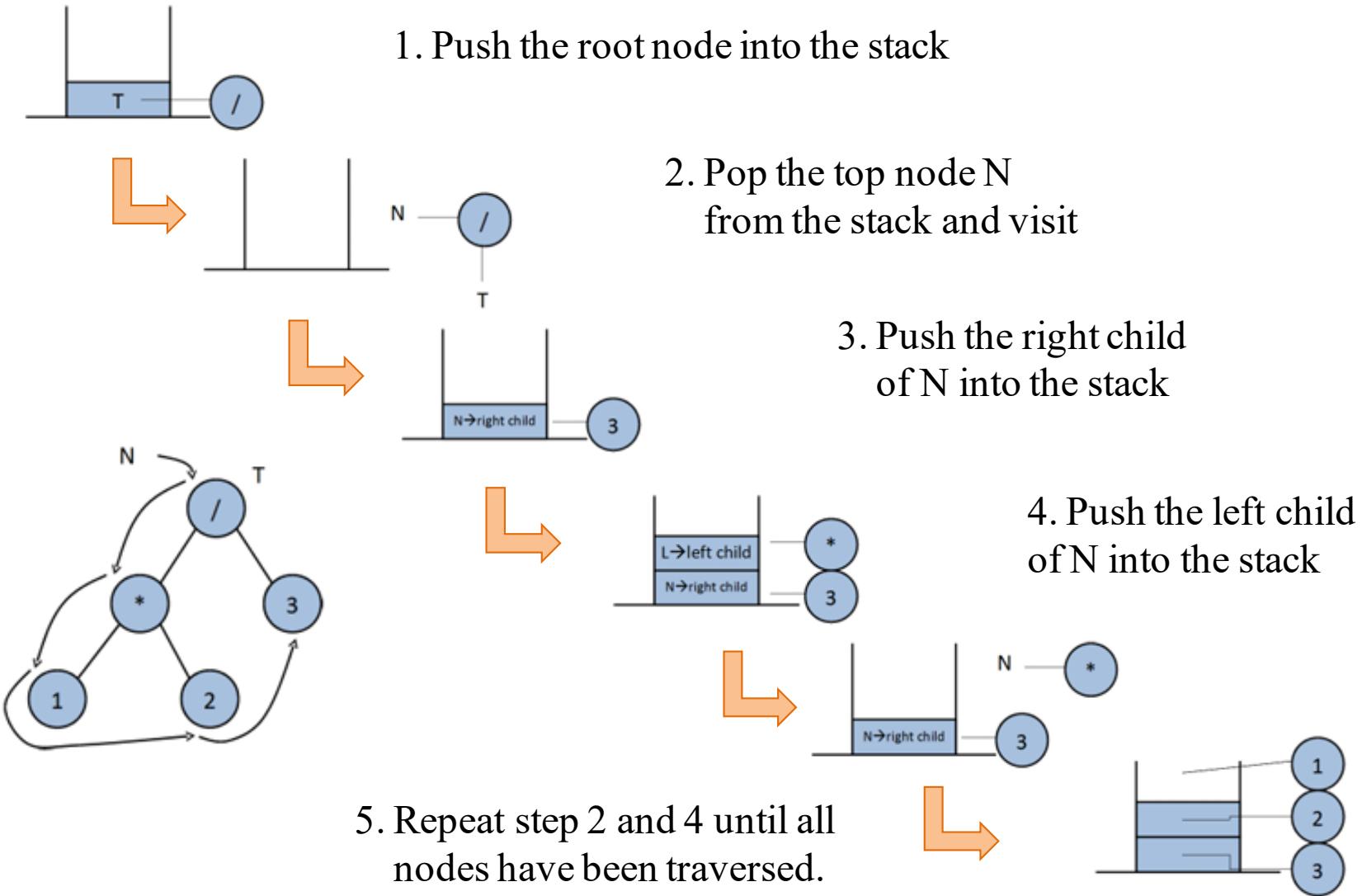
- Input: `BinaryTreeNode* T`
 - Output: Post-order traversal of tree node data
 - Steps:
-

```
void post_order(BinaryTreeNode* T) {  
    if (T == NULL) return;  
  
    post_order(T->left);  
  
    post_order(T->right);  
  
    visit(T->data); // print or other processing  
}
```

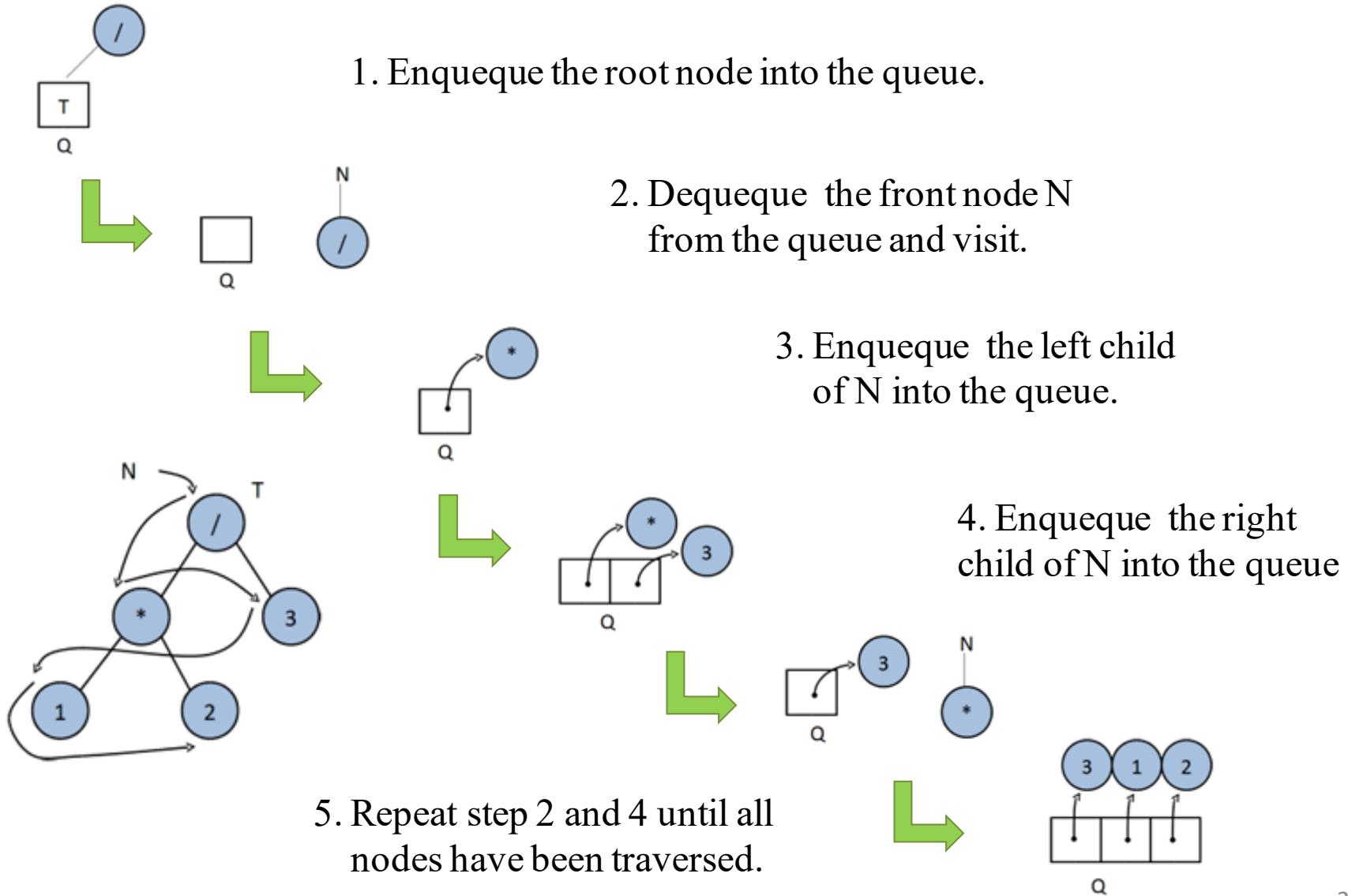
Using Stacks and Queues in Traversal of BT

- Use of a **stack** as a container for managing unvisited nodes is an efficient way for performing **pre-order traversal** on a binary tree.
- Use of a **queue** as a container for managing unvisited nodes is an efficient way for performing **level-order traversal** on a binary tree.

Pre-Order Traversal Using Stacks



Level-Order Traversal Using Queues



Next Lecture

We focus on:

- binary search trees (BST)
- main operations of BST such as search, insert and delete

Readings/ Study Reference

Course Textbook: “*Data Structure and Algorithms in a Nutshell*” A. Wong et al.

Chapter 7. Trees

Section 7.2. Binary Trees

The End of Lecture

Course: Data Structures and Algorithms

Instructor: Homeyra Pourmohammadali

You Try Questions and Solutions
Binary Tree Traversal

You Try 1. Post-Order Traversal



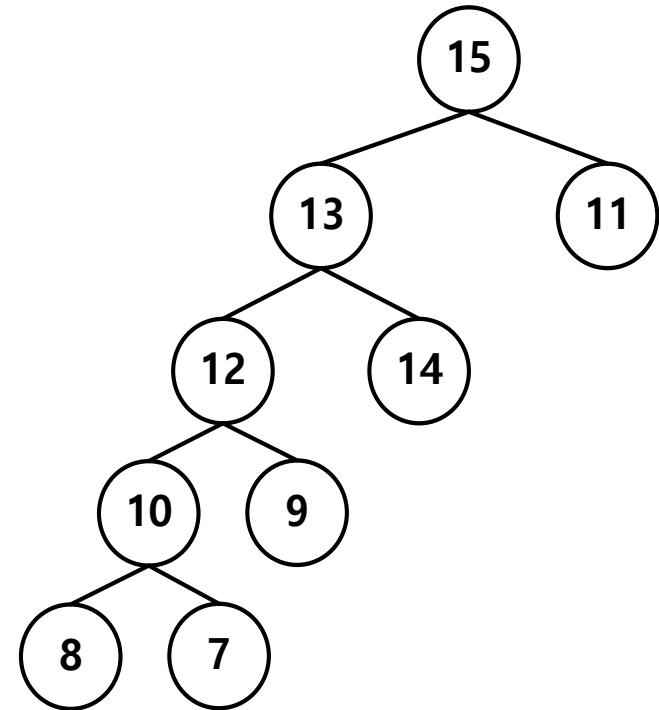
Choose the correct post-order of this binary tree.

(left subtree → right subtree → root)

a) 15 13 11 12 14 10 9 8 7

b) 8 10 7 12 9 13 14 11 15

c) 8 7 10 9 12 14 13 11 15



Find pre-order and in-order.

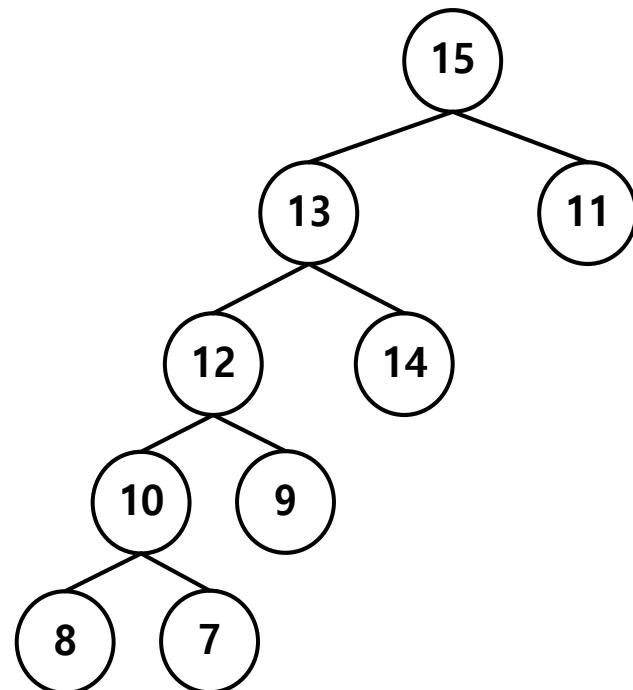
You Try 1 Solution. Post-Order Traversal

Post-order: $T_L \rightarrow T_R \rightarrow r$

a) 15 13 11 12 14 10 9 8 7

b) 8 10 7 12 9 13 14 11 15

c) 8 7 10 9 12 14 13 11 15



Pre-order ($r \rightarrow T_L \rightarrow T_R$)

15 13 12 10 8 7 9 14 11

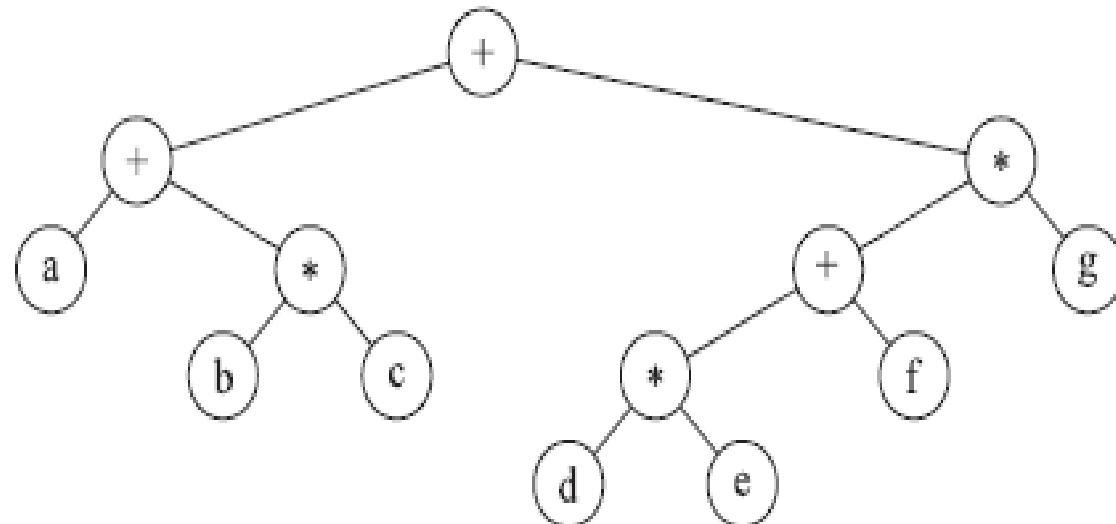
In-order ($T_L \rightarrow r \rightarrow T_R$)

8 10 7 13 9 13 14 15 11

You Try 2. Binary Trees of Algebraic Expressions



Apply **pre-order** ($r \rightarrow T_L \rightarrow T_R$) and **post-order** ($T_L \rightarrow T_R \rightarrow r$) traversals to the following binary tree and obtain the pre-fix and post-fix forms of the mathematical expression.



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

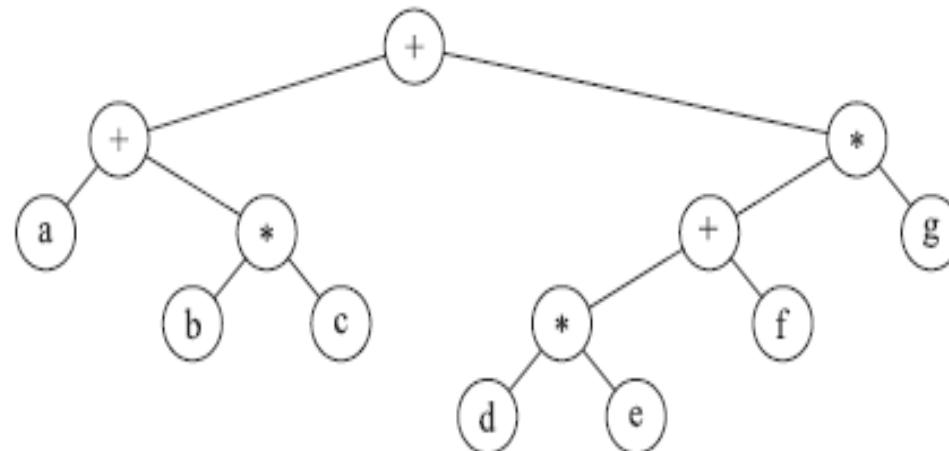
You Try 2 Solution. Binary Trees of Algebraic Expressions

Post-order traversal: recursively print out the left subtree, the right subtree, and then the operator (root).

The resulting output is **a b c * + d e * f + g * +**

Pre-order traversal: print out the operator (root) first and then recursively print out the left and right subtrees.

The resulting expression, **+ + a * b c * + * d e f g**



The End of You Try Activities

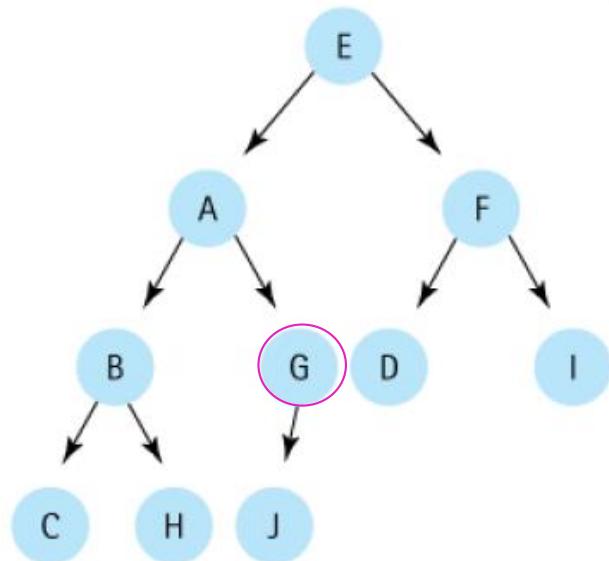
Course: Data Structures and Algorithms

Instructor: Homeyra Pourmohammadali

Binary Search Trees

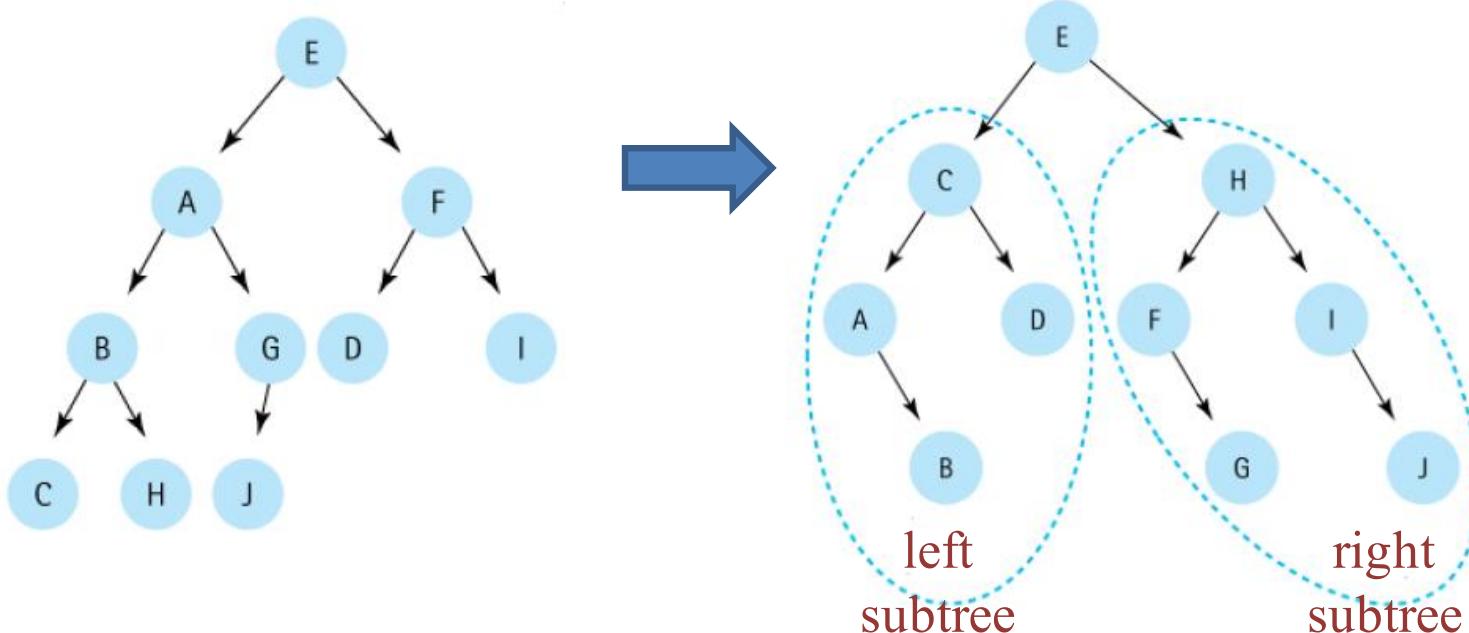
Motivation

- When searching a value in a binary tree (for instance G), we begin at the root. If the value is not the same as root, we need to keep searching.
- But which of its children should we examine next, the right or the left child?
- The nodes are not organized in any special order, so we have to check both subtrees. The search operation is $O(N)$, which is not better than linked list!



Motivation

- We want to add a special property to binary tree to support $O(\log N)$ searching.
- Put all the nodes with values smaller than the value in the root in its left subtree, and all the nodes with values larger than the value in the root in its right subtree.



Motivation

Some **binary search trees** are used to maintain sorted set of data. For instance, when you are placing online orders and you want to maintain the live data in sorted order of prices; or you want to know number of items purchased at cost below or above a given cost at any moment.



https://www.freepik.com/free-vector/online-food-order-isometric-e-commerce-poster_4016598.htm

Learning Outcomes

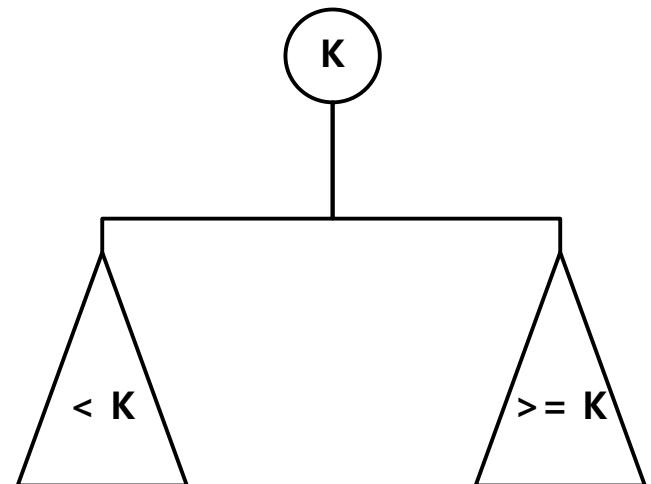
By the end of this lecture you will be able to:

- define the binary search tree (BST) at the logical level
- apply main operations such as search, insertion and deletion.
- see the interface for a BST abstract data type.
- discuss Big-O efficiency of a given BST operation.

Binary Search Trees (BST) Properties

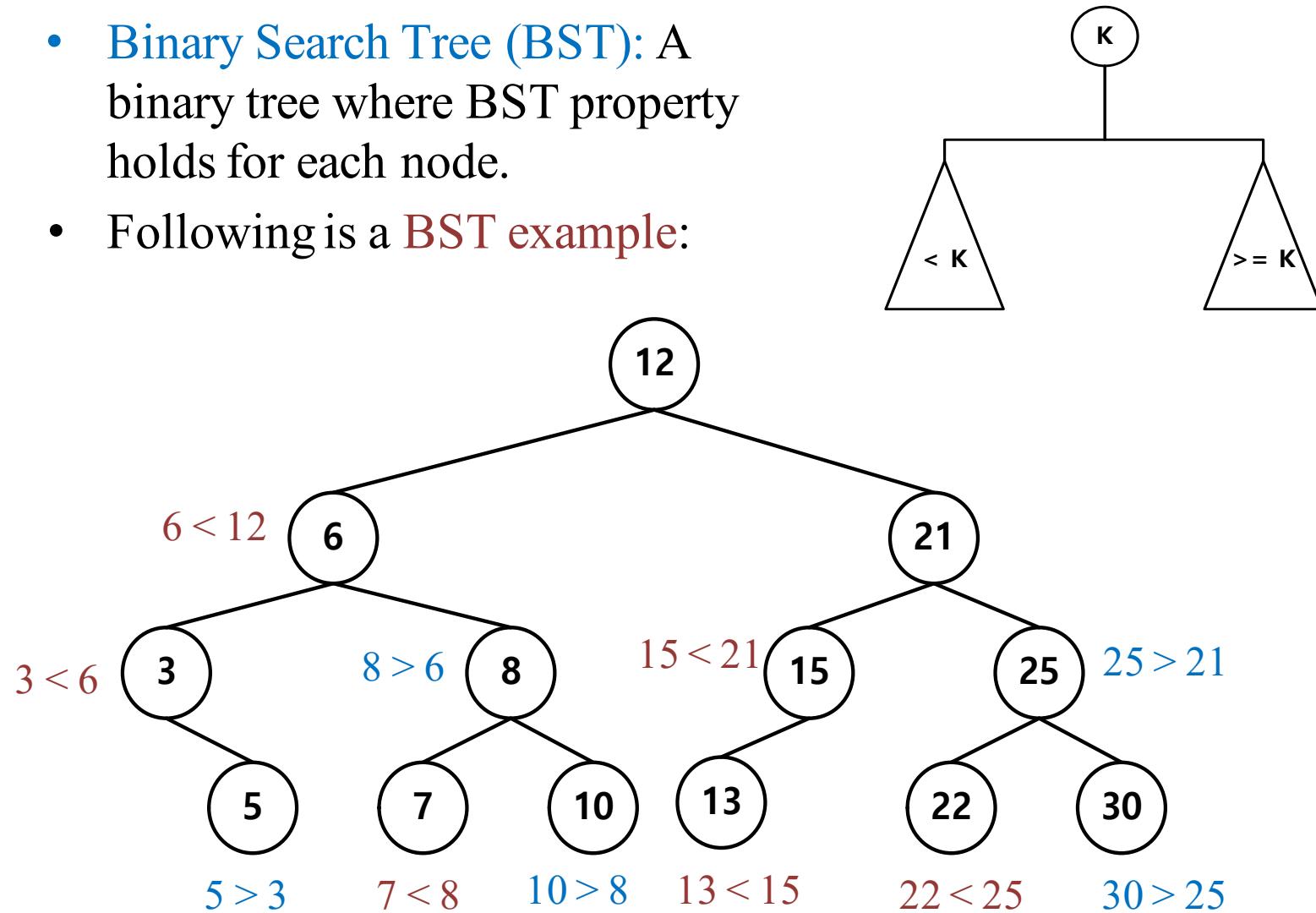
- Searching for a particular item is one operation for which the binary tree ADT is ill suited. The binary search tree corrects this deficiency by organizing its data by value.

- Each node has a key value K
- All keys in the left subtree are $< K$
- All keys in the right subtree are $\geq K$



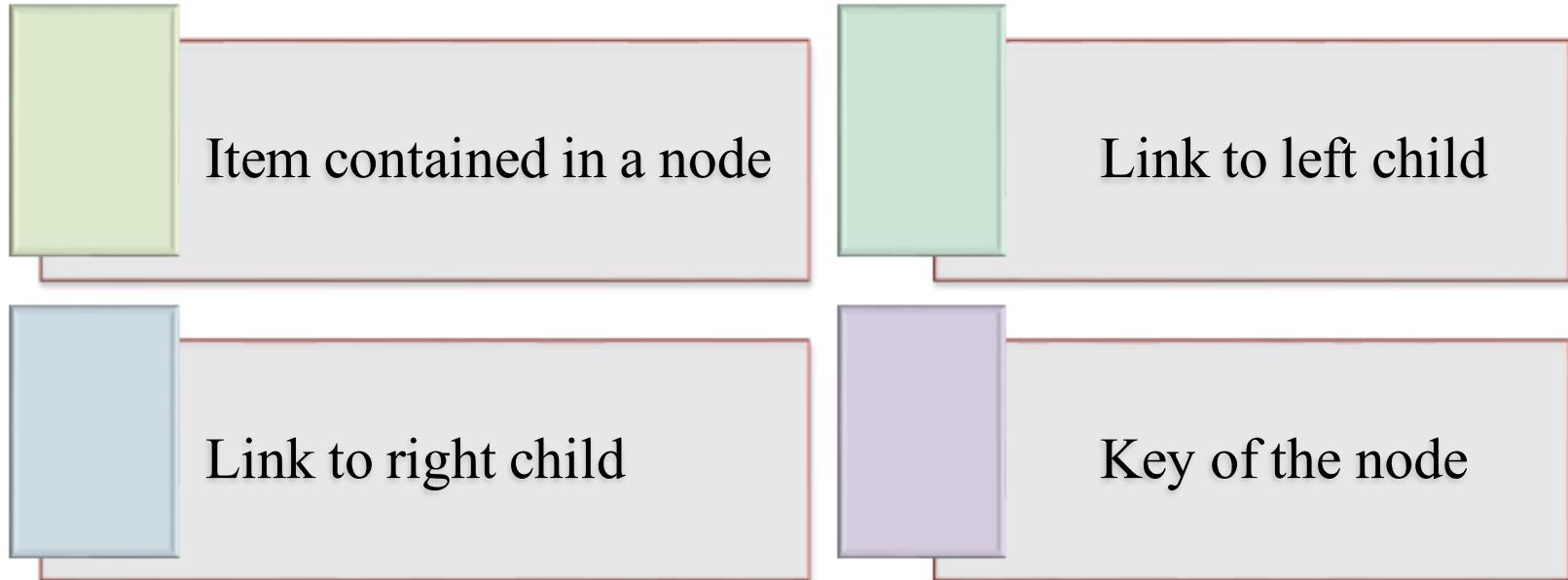
Binary Search Trees (BST) Properties

- **Binary Search Tree (BST):** A binary tree where BST property holds for each node.
- Following is a **BST example:**



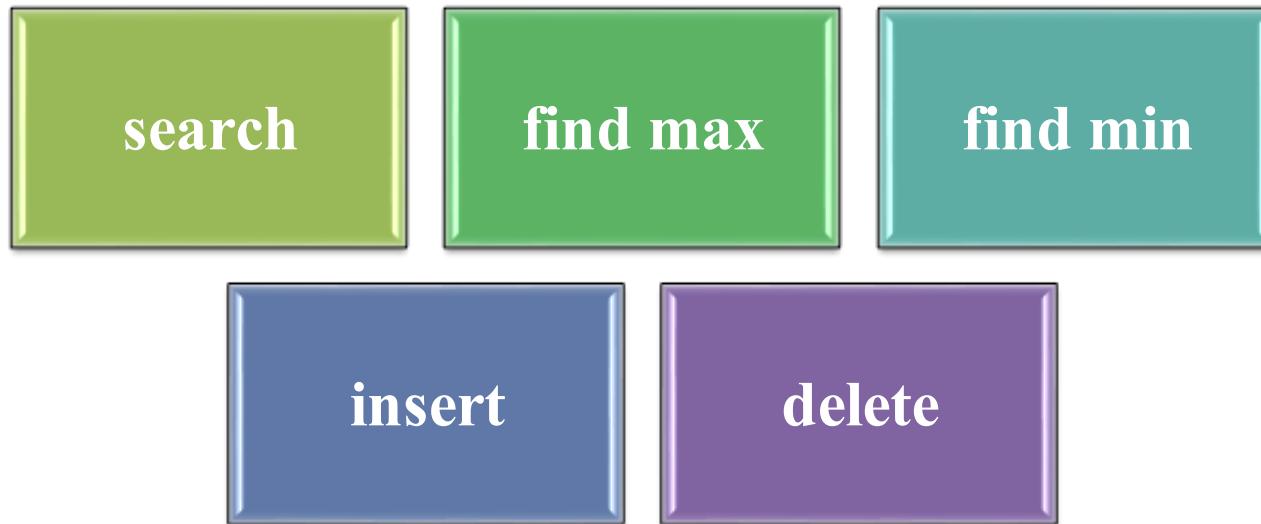
Binary Search Trees (BST) Structure

- In a generic binary tree, each node in the BST consists of:



Binary Search Trees (BST) Operations

- The overall structure of a simple BST ADT consists of these operations:



- Given the data members and operations, a simple interface for BST node ADT that can hold integers are defined.

BST Node Class Declaration

```
class BSTNode{  
    int key;      //holds the key value at this tree node  
    int data;     //holds the data value at this tree node  
  
    BSTNode* left;    //points to left child  
    BSTNode* right;   //points to right child  
  
public:  
    BSTNode();      //default constructor  
    ~BSTNode();     //destructor  
  
    BSTNode get_left();    //returns left child  
    BSTNode get_right();   //returns right child  
  
    BSTNode search(int key_val); //returns found node  
    BSTNode find_min();    //returns node with min key  
    BSTNode find_max();   //returns node with max key  
  
    bool insert(BSTNode &bNode); //inserts node  
    bool delete(BSTNode &bNode); //deletes node  
};
```

Search Operation

search, `find_max` and `find_min` involve
traversing and searching through the BST

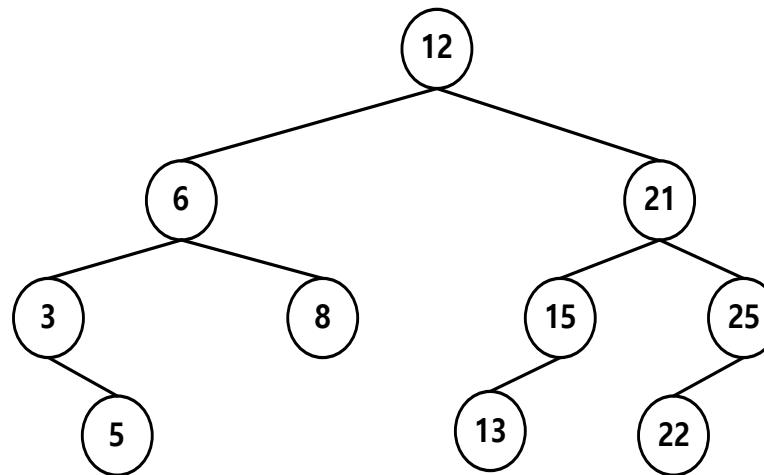
Binary Search Tree Traversal

- The traversals of a binary search tree (**BST**) are the same as the traversals of a binary tree (**BT**).
- The **in-order** traversal of a binary search tree, however, is of special note.

Example 1. Traversal of a Binary Search Tree (BST)

What is the speciality about the **in-order** traversal ($T_L \rightarrow r \rightarrow T_R$) of a binary search tree? (e.g. the BST shown below)

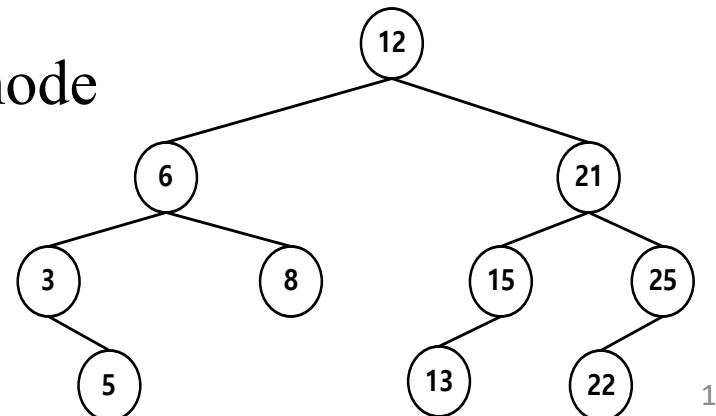
- a) It traverses in a non increasing order.
- b) It traverses in an increasing order.
- c) It traverses in a random fashion.
- d) It traverses based on priority of the node.



Example 1 Solution. BST Traversal

A BST consists of elements lesser than the node to the left and the ones greater than the node to the right.
traversal will give the elements in an increasing order.

- a) It traverses in a non increasing order
- b) It traverses in an increasing order
- c) It traverses in a random fashion
- d) It traverses based on priority of the node



In-Order: 3, 5, 6, 8, 12, 13, 15, 21, 22, 25

You Try 1. BST Traversal

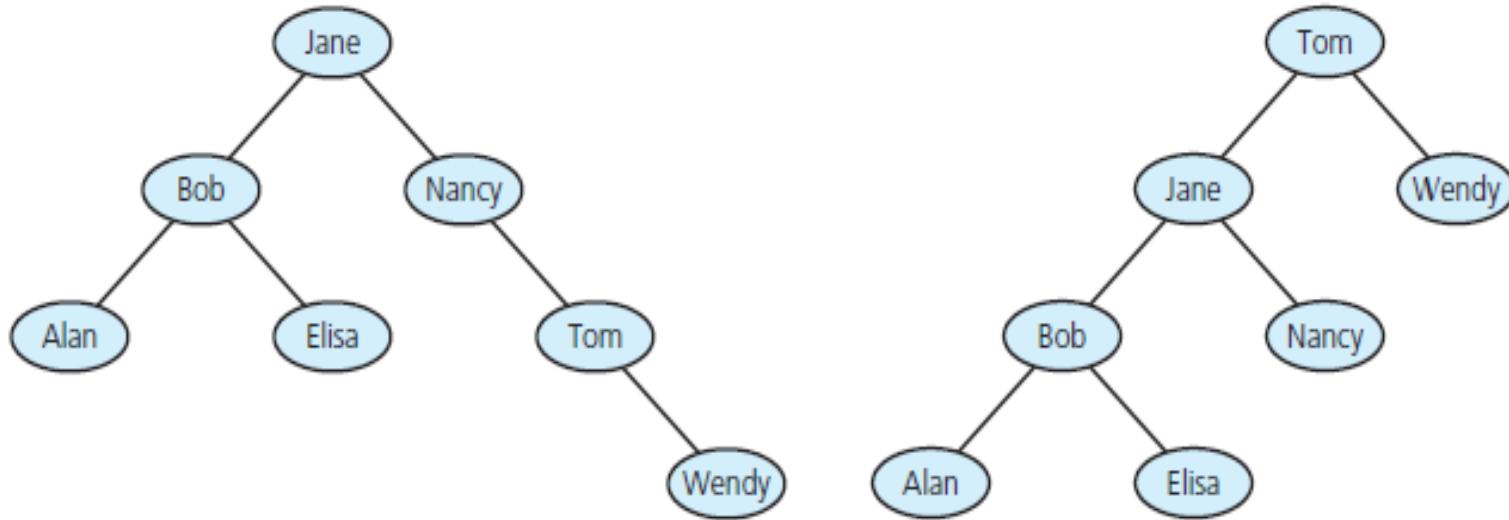


Which of the following is false about a binary search tree?

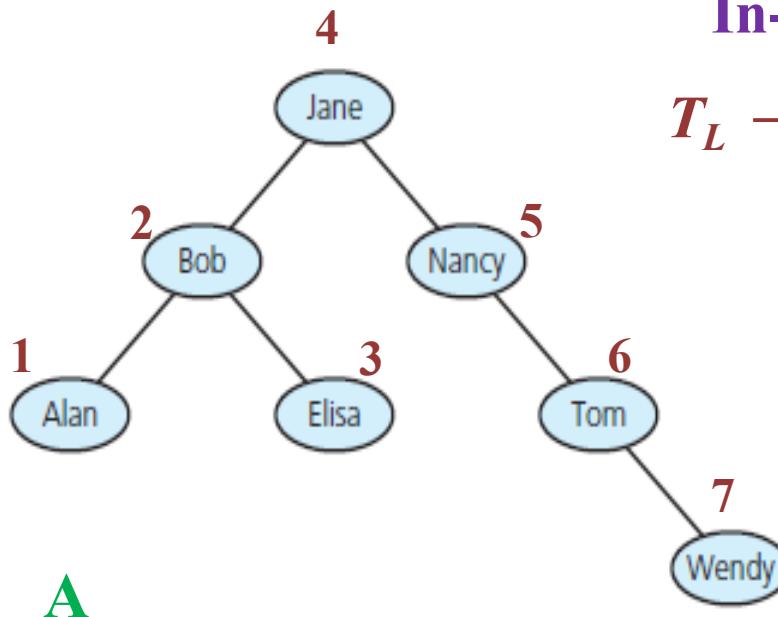
- a) The left child is always lesser than its parent.
- b) The right child is always greater than its parent.
- c) The left and right sub-trees should also be binary search trees.
- d) In order sequence gives decreasing order of elements.

Example 2. BST Traversal

The nodes of a binary search tree contain people's names. These objects are Alan, Bob, Elisa, Jane, Nancy, Tom, and Wendy. Find the **in-order** traversals ($T_L \rightarrow r \rightarrow T_R$) of the following BSTs. Are they similar? How about their pre-order and post-order traversals?

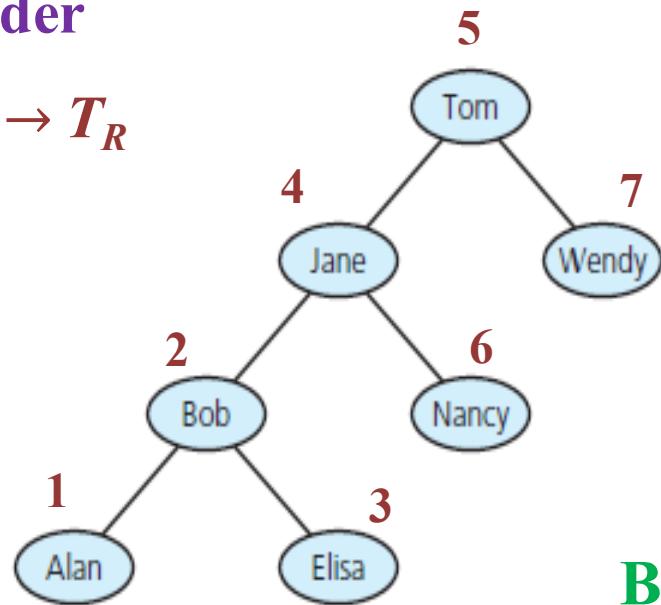


Example 2 Solution. BST Traversal



Alan, Bob, Elisa, Jane, Nancy, Tom, Wendy

In-Order
 $T_L \rightarrow r \rightarrow T_R$

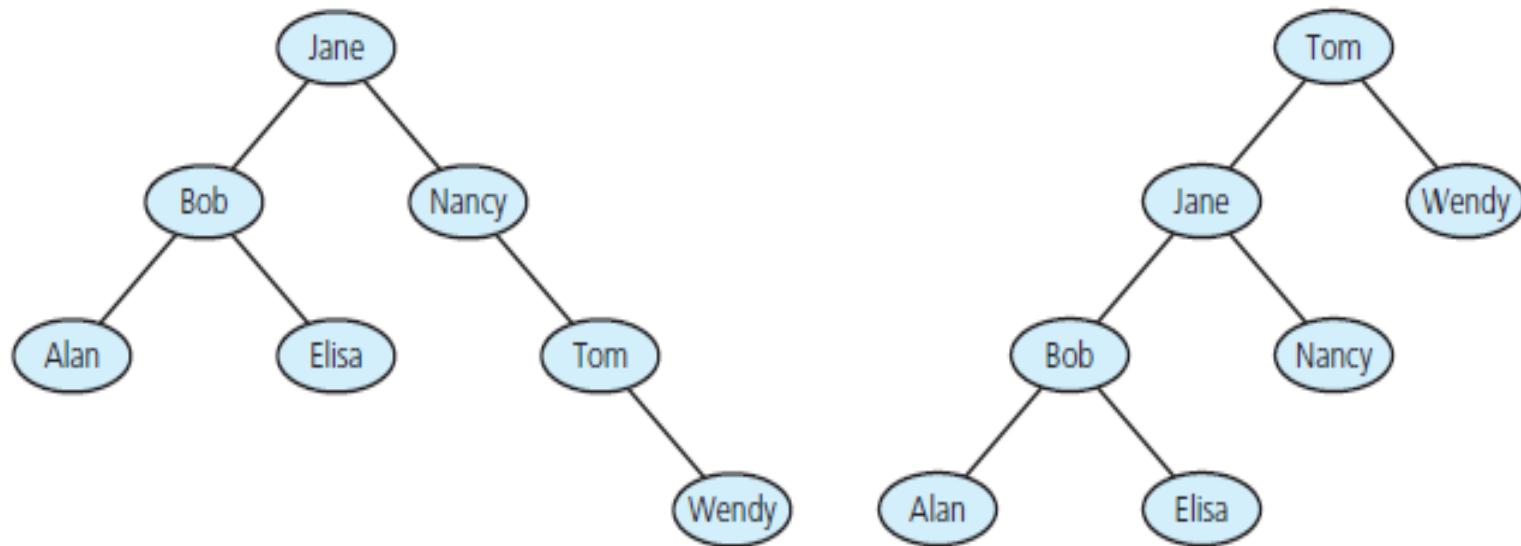


Alan, Bob, Elisa, Jane, Nancy, Tom, Wendy

- The result of the in-order traversals of both BSTs are the same.
- The in-order traversal of a binary search tree visits the tree's nodes in **sorted search-key order**.

Example 2 Solution. BST Traversal

The pre-order ($r \rightarrow T_L \rightarrow T_R$) and post-order ($T_L \rightarrow T_R \rightarrow r$) traversals of these BSTs are not necessarily the same and will not give the alphabetically ordered list of names.

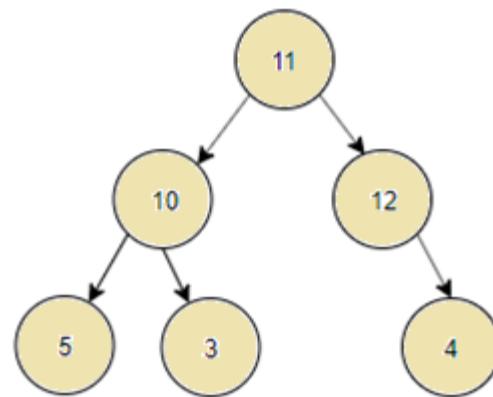


You Try 2. BST Traversal

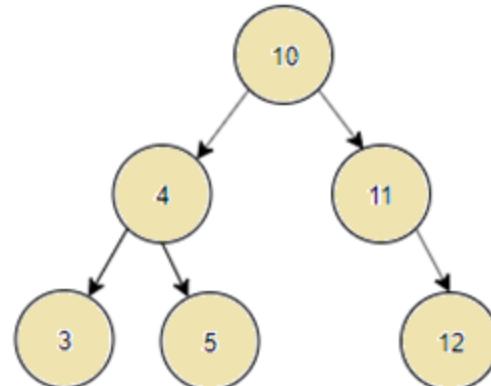


The pre-order traversal ($r \rightarrow T_L \rightarrow T_R$) of which of the following BSTs is 10, 4, 3, 5, 11, 12?

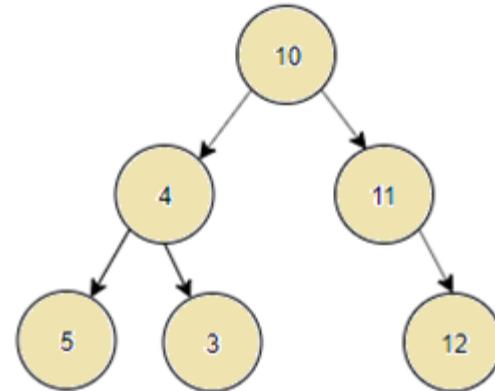
A)



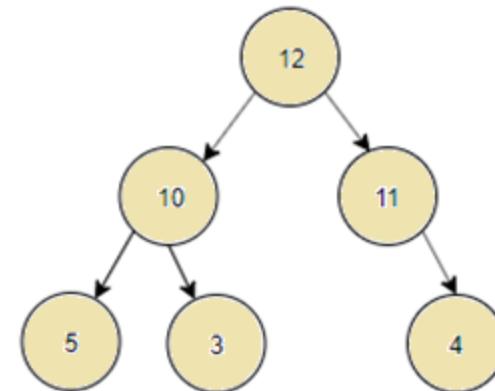
C)



B)



D)



Search Operation

find_min

1. Traverse to the **left** child of the root node, if a left child exists.
2. At the current node,
 - if a **left** child exist, travel to the left child.
 - if a **left** child does not exist, then current node is the node with the **lowest key** value.

Search Operation

find_max

1. Traverse to the **right** child of the root node, if a right child exists.
2. At the current node,
 - if a **right** child exist, travel to the right child.
 - if a **right** child does not exist, then current node is the node with the **highest key** value.

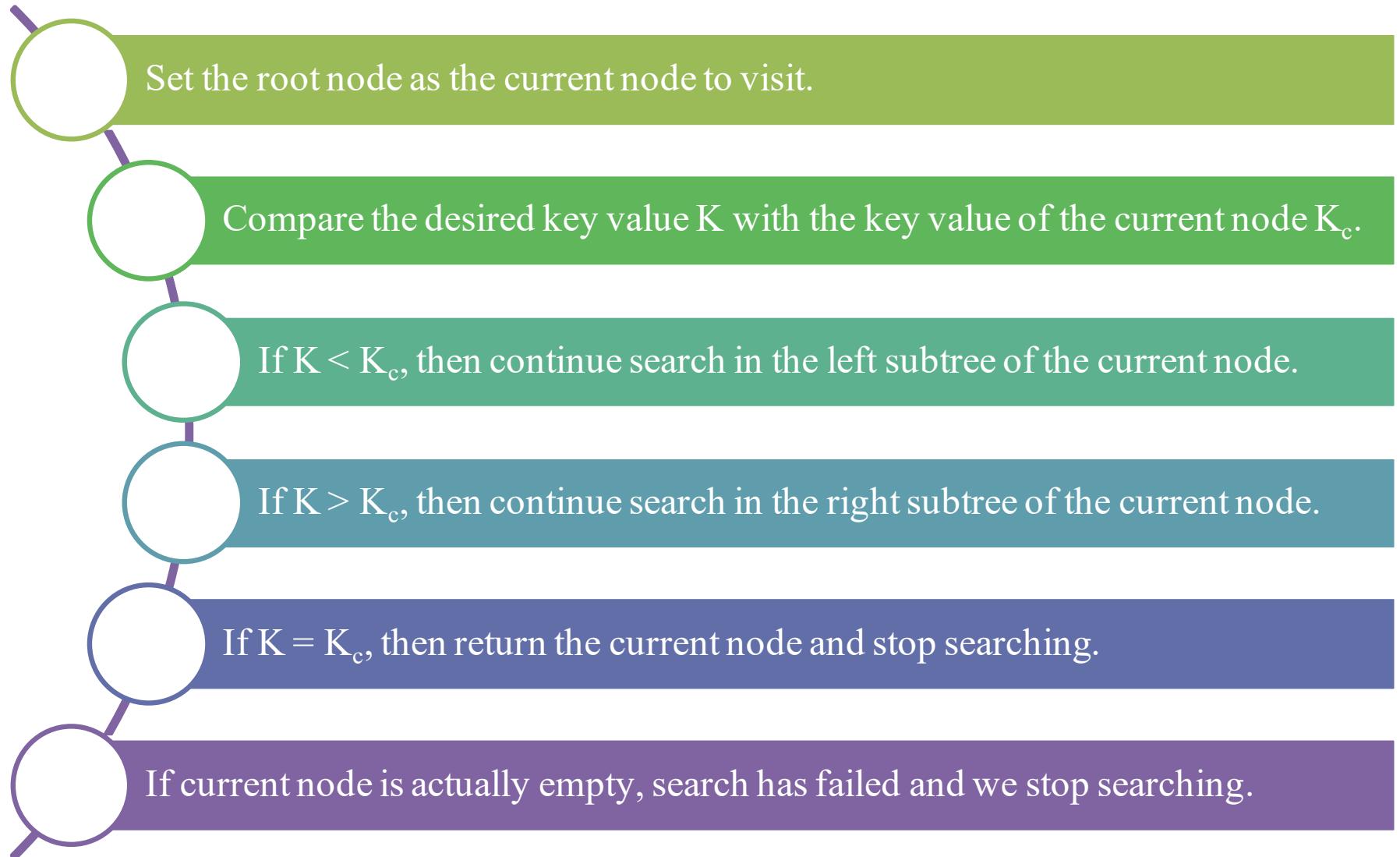
Search Operation

search

Traverse the BST until

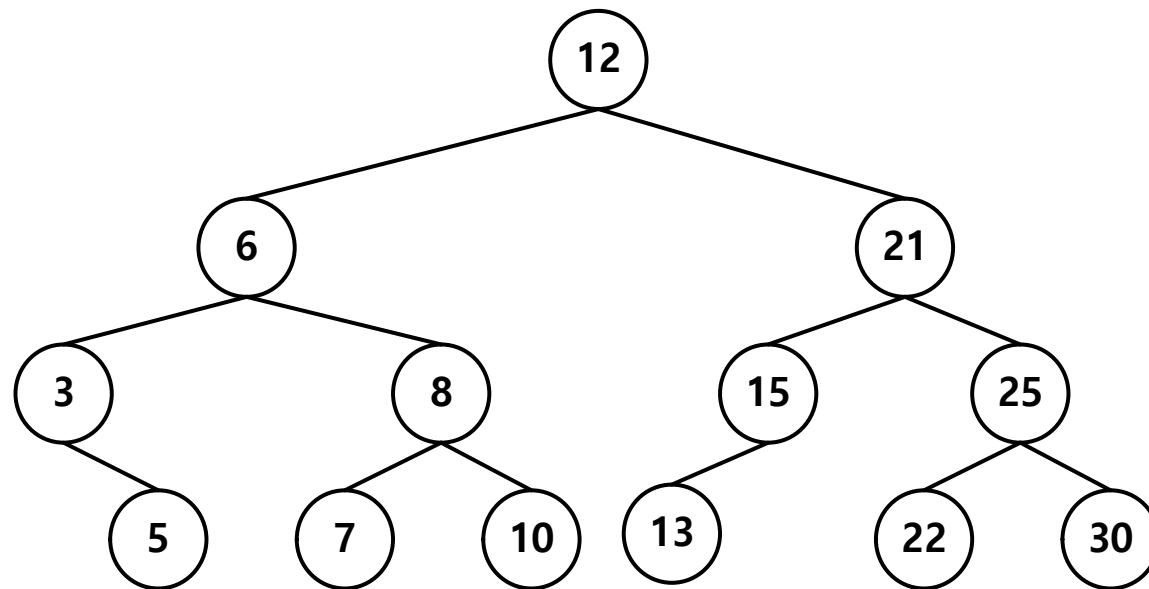
- finding the node with the desired key value
- reaching a leaf node in the BST

Algorithm for Searching BST

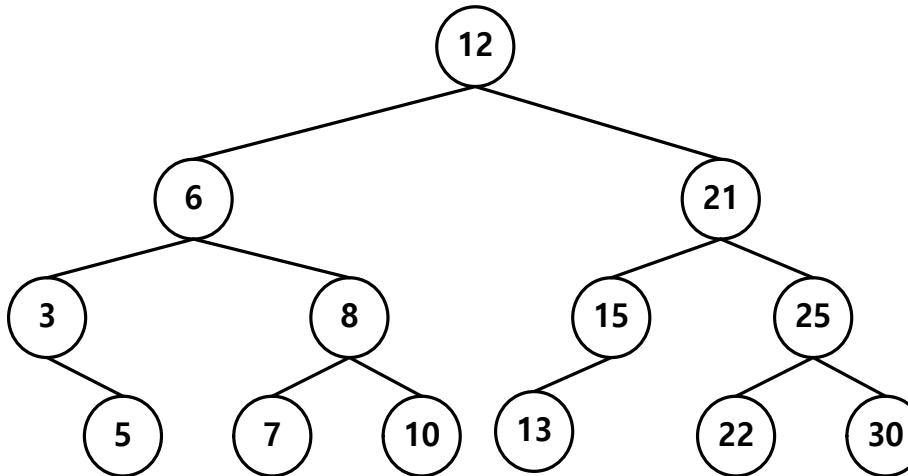


Example 3. Search in a BST

Show the steps of algorithm for searching the value of **8** in the following BST:



Example 3 Solution. Search in a BST

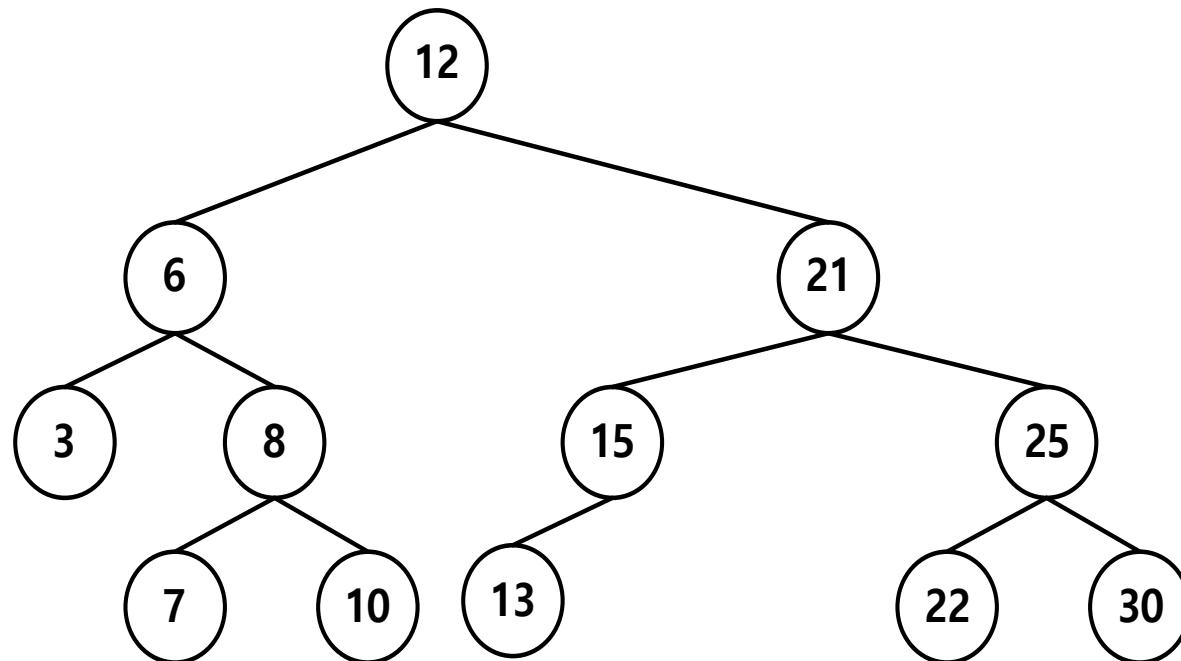


1. Set the root node as the current node to visit.
2. Compare 8 with current node 12.
3. $8 < 12$, then continue search in the left subtree of 12.
4. Compare 8 with the current node 6.
5. $8 > 6$, then continue search in the right subtree of 6.
6. Compare 8 with the current node 8.
7. $8=8$, then return 8 and stop searching.

You Try 3. Search in BST



Show the steps of algorithm for searching the value of **18** in the following BST:



Algorithm: Search(T, K) (recursive)

- **Input:** BST node T, key value K
 - **Output:** if found, return a node with key value K; otherwise, return empty node
 - **Steps:**
-

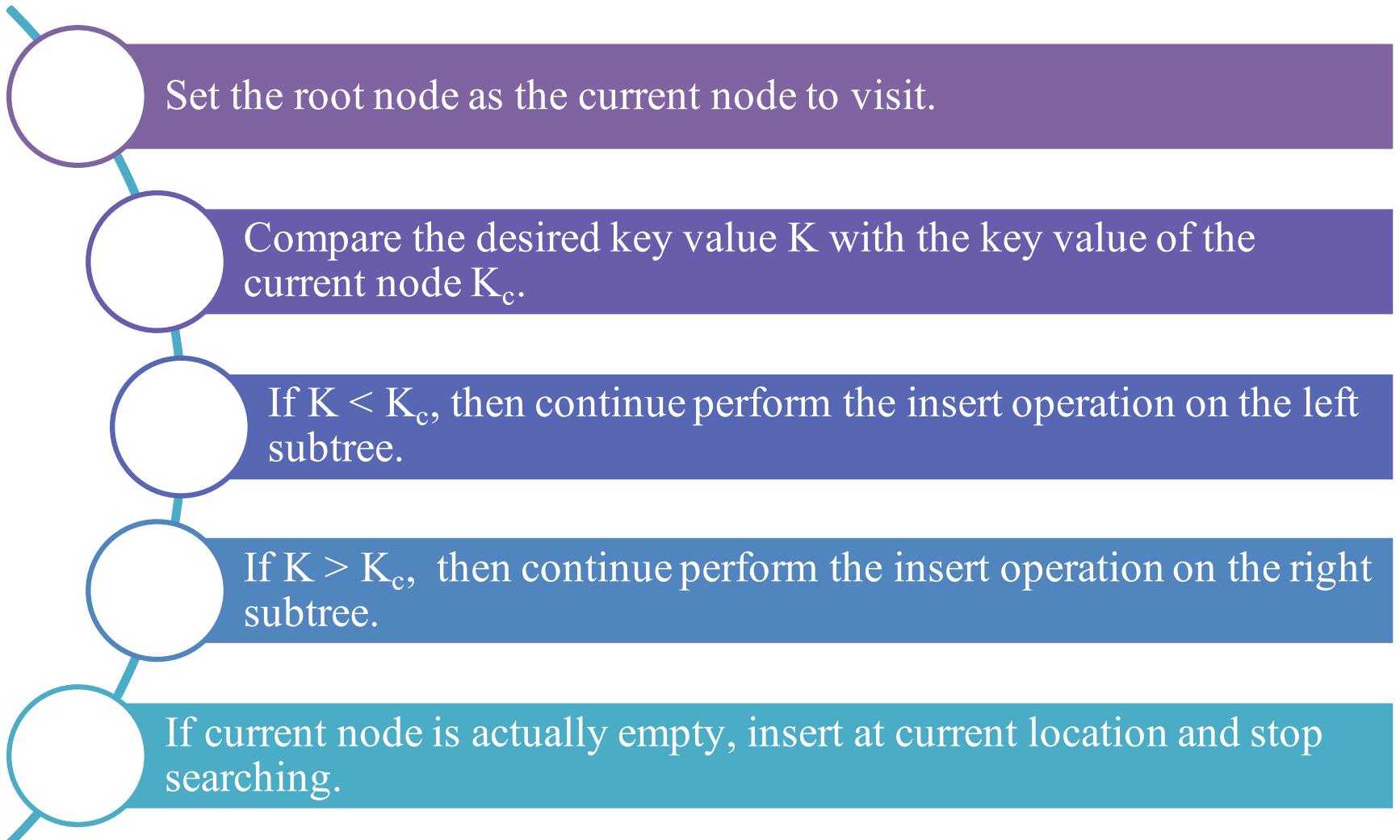
```
BSTNode Search(T, K) {  
    if (T == NULL) return BSTNode0;  
    if (T.key == K)  
        return T;  
    else if (K < T.key)  
        Search(T.leftChild, K);  
    else if (K > T.key)  
        Search(T.rightChild, K);  
}
```

Insert Operation

Insert Operation

- To create and maintain the information stored in a binary search tree, we need an operation that inserts new nodes into the tree.
- To find the right location to insert the node, assuming that a node with the same key value is not the tree, we need to traverse the BST.
- A new node is always inserted into its appropriate position in the tree **as a leaf**.

Algorithm for Inserting in BST

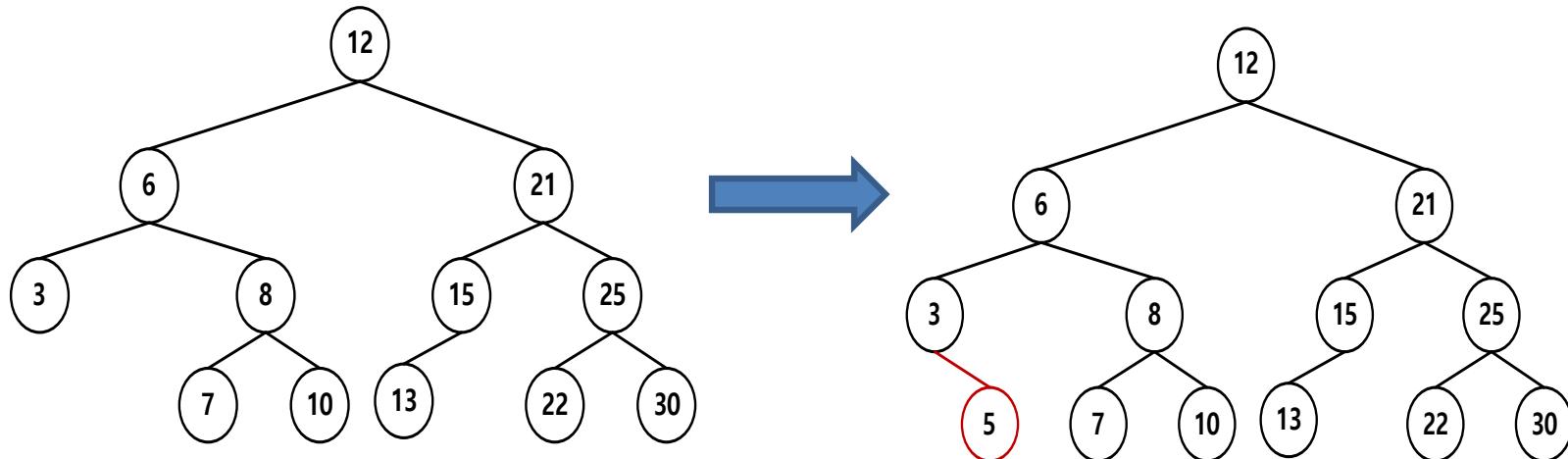
- 
- Set the root node as the current node to visit.
 - Compare the desired key value K with the key value of the current node K_c .
 - If $K < K_c$, then continue perform the insert operation on the left subtree.
 - If $K > K_c$, then continue perform the insert operation on the right subtree.
 - If current node is actually empty, insert at current location and stop searching.

Example 4. Insert in a BST

Insert the value of **5** in the following BST:

Solution:

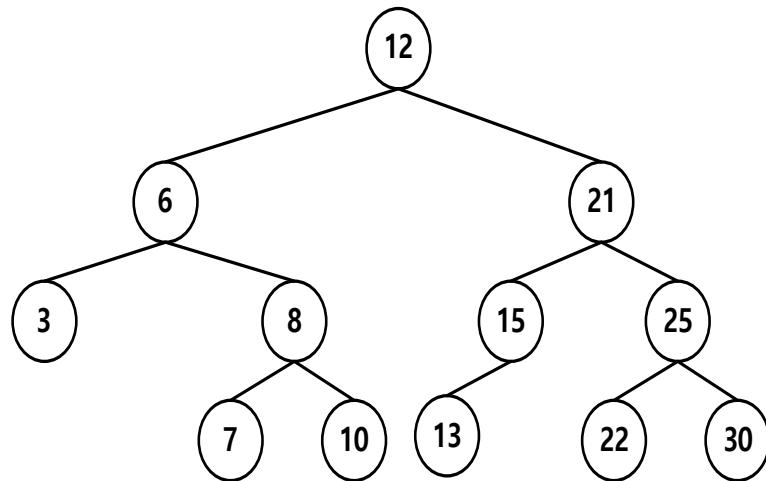
- 1) $5 < 12$, traverse to left subtree.
- 2) $5 < 6$, traverse to left subtree.
- 3) $5 > 3$, traverse to right subtree.
- 4) right subtree is empty, insert 5 as the right child of 3.



You Try 4 . Insert in a BST

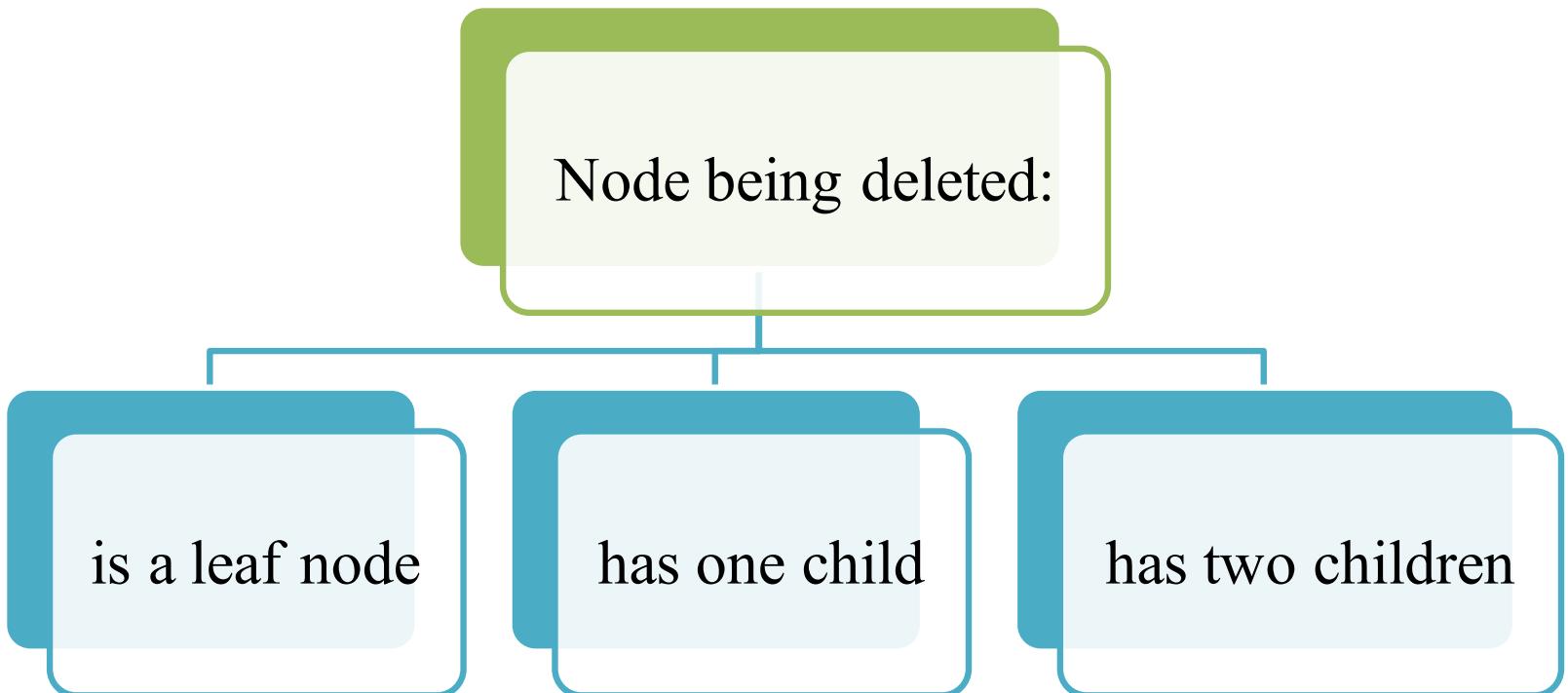


Insert the value of **17** in the following BST and write the steps:



Delete Operation

Delete Operation



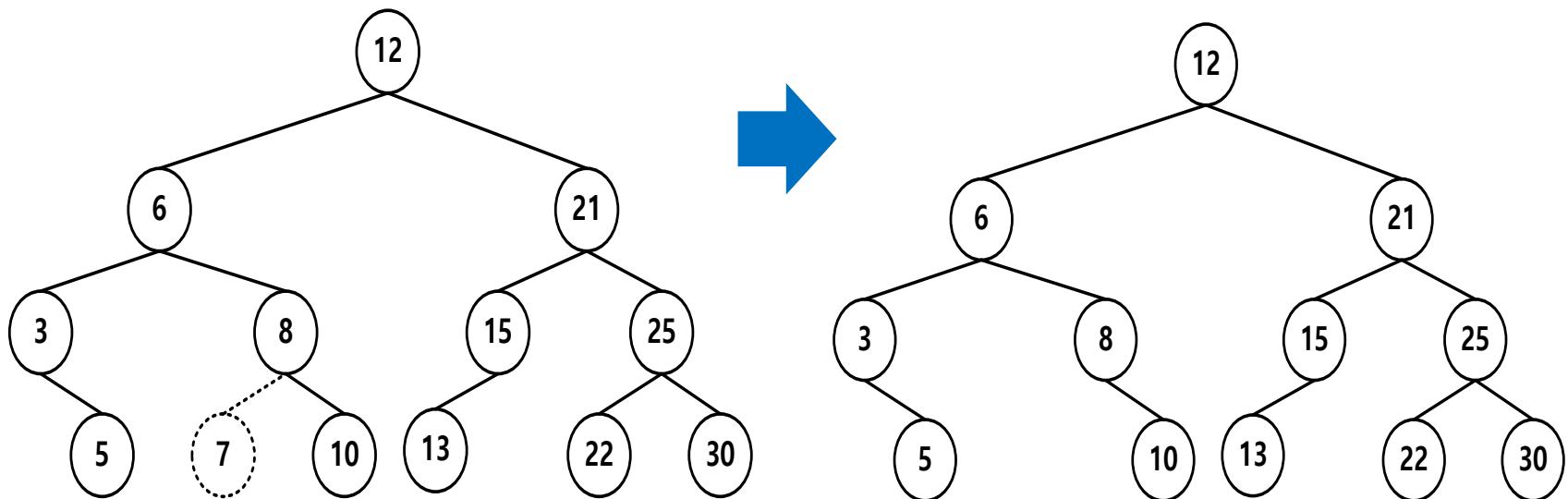
Delete Operation

1. If D cannot be found in T, return FALSE.
2. If D **is a leaf node**, remove it, then return TRUE and terminate.
3. If D **has one child node**, swap with the child node, delete the child node, then return TRUE and terminate.
4. If D **has two child nodes**, swap the values with successor or predecessor, delete the successor or predecessor respectively, and then return TRUE and terminate.
 - **Predecessor** is the maximum value in the left subtree of BST.
 - **Successor** is the smallest value in the right subtree of BST.

Deleting Nodes from BST

1. If D is a leaf node in T ([has no child](#)), remove it, then return TRUE.

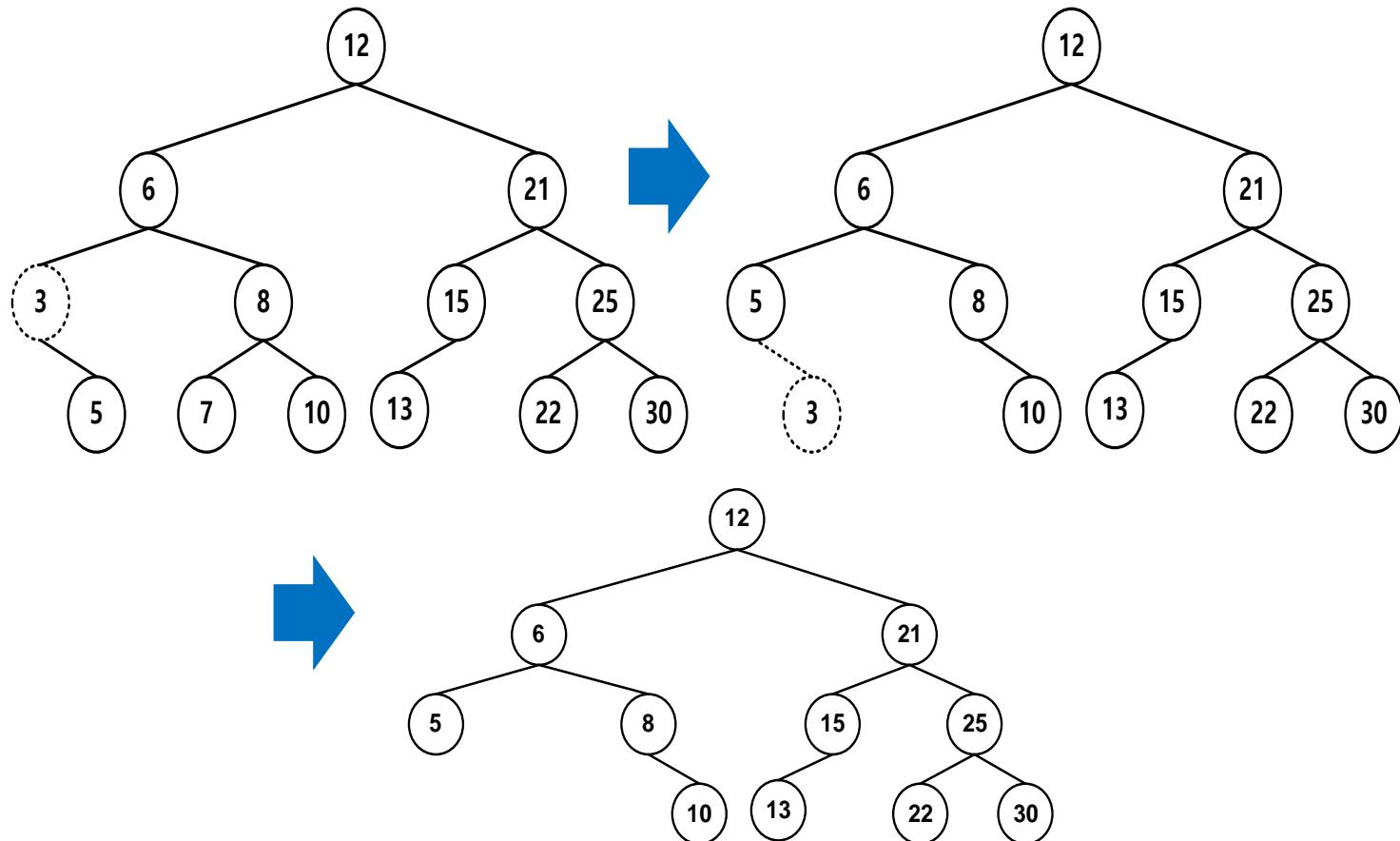
Example: Delete 7



Deleting Nodes from BST

2. If D has one child node, swap with the child node, delete the child node, then return TRUE.

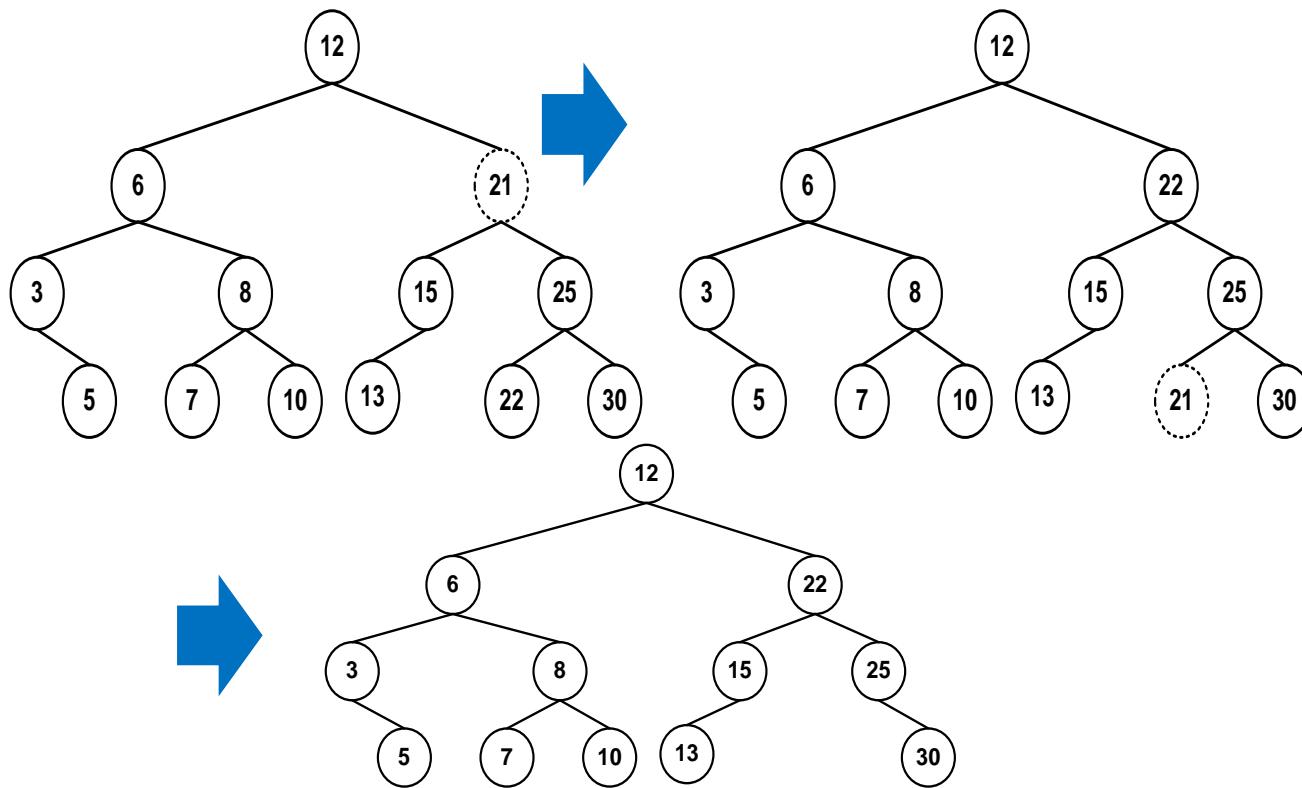
Example: Delete 3



Deleting Nodes from BST

3. If D has **two child nodes**, swap the values with successor or predecessor, delete the successor or predecessor respectively, and then return TRUE.

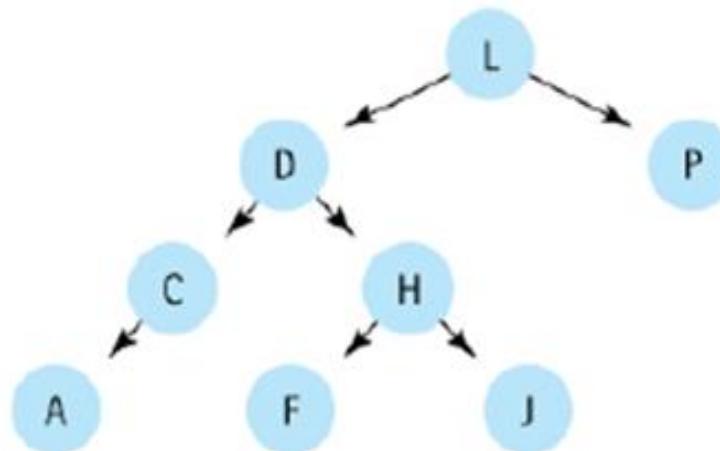
Example: Delete 21 (swap with the successor)



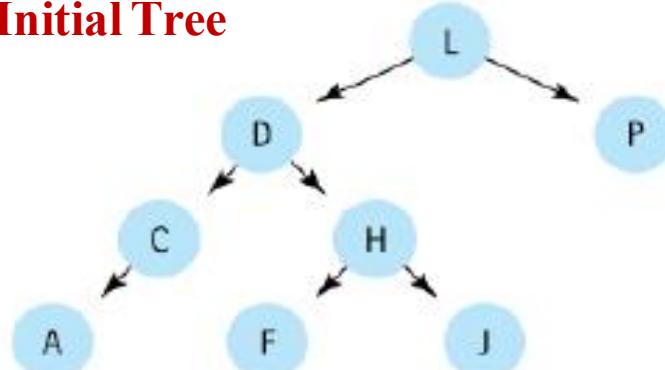
Example. Deleting nodes from a BST

Do the sequence of deletion on the following BST, and redraw the BST after each operation. Delete the nodes containing:

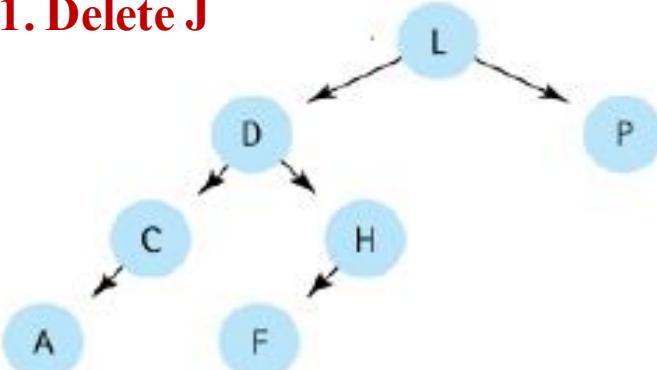
1. J
2. C
3. L
4. D
5. A



Initial Tree



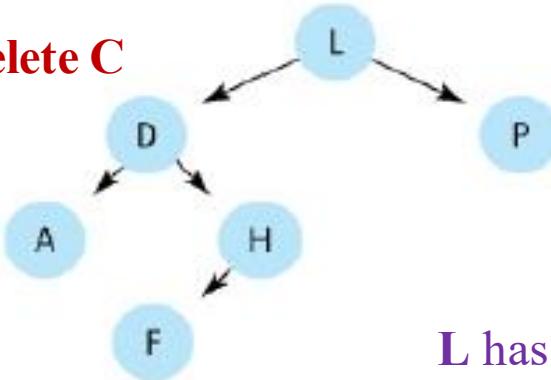
1. Delete J



J is a leaf node

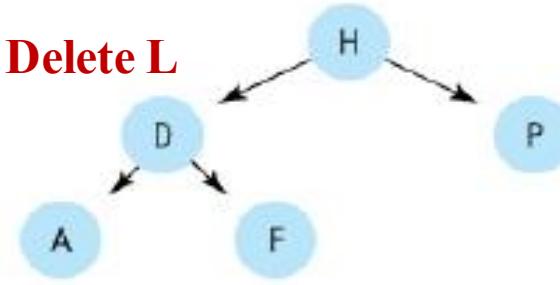
C has 1 child

2. Delete C



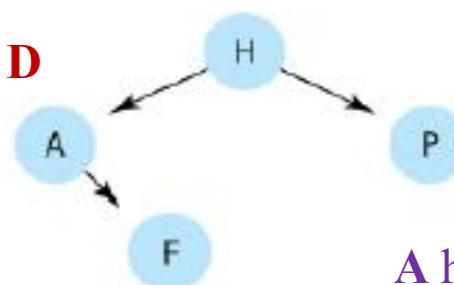
L has 2 children

3. Delete L



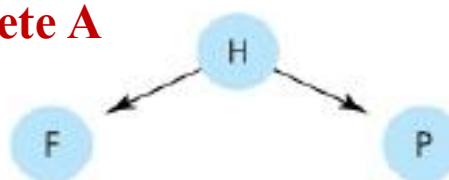
D has 2 children

4. Delete D



A has 1 child

5. Delete A

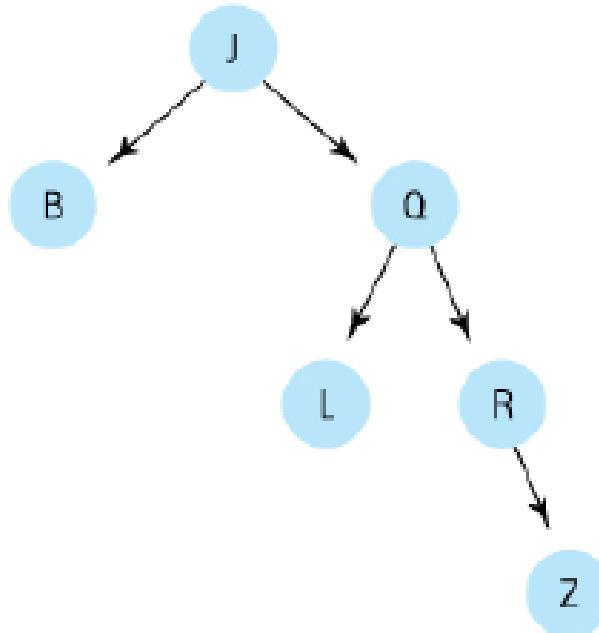


You Try 5. Deleting nodes from a BST



Do the sequence of deletion on the following BST, and redraw the BST after each operation. Delete the nodes containing:

1. Z
2. R
3. Q



Big-O of Different Operations of BST

Big-O of Main Operations of BST

- Each insertion, or removal operation requires a number of comparisons equal to the number of nodes along the path.
- The maximum number of comparisons for an insertion, or removal is the height of the tree.
- The height of an n -node binary tree ranges from $\log_2(n + 1)$ to n .
- If the height of the binary search tree is $\log_2(n + 1)$, the efficiency of its operations is $O(\log n)$.

Big-O of Different Operations of BST

Operation	Average Case	Worst Case
Insertion	$O(\log n)$	$O(n)$
Removal	$O(\log n)$	$O(n)$
Traversal	$O(n)$	$O(n)$

Conclusion

- The BST allows you to use a binary search-like algorithm to search for an item with a specified value.
- BSTs come in many shapes. The height of a BST with n nodes can range from a minimum of $O(\log_2(n + 1))$ to a maximum of $O(n)$.
- The shape of a BST determines the efficiency of its operations. The closer a BST is to a balanced tree, the closer the behavior of the search algorithm will be to a binary search.
- An in-order traversal of a BST visits the tree's nodes in sorted search-key order.

Next Lecture

We focus on:

- AVL Trees

Readings/ Study Reference

Course Textbook: “*Data Structure and Algorithms in a Nutshell*” A. Wong et al.

Chapter 7. Trees

Section 7.3. Binary Search Trees

The End of Lecture

Course: Data Structures and Algorithms

Instructor: Homeyra Pourmohammadali

You Try Questions and Solutions
Binary Search Trees

You Try 1. BST Traversal



Which of the following is false about a binary search tree?

- a) The left child is always lesser than its parent.
- b) The right child is always greater than its parent.
- c) The left and right sub-trees should also be binary search trees.
- d) In order sequence gives decreasing order of elements.

You Try 1 Solution. BST Traversal

Which of the following is false about a BST?

- a) The left child is always lesser than its parent.
- b) The right child is always greater than its parent.
- c) The left and right sub-trees should also be binary search trees.
- d) In-order sequence gives decreasing order of elements.

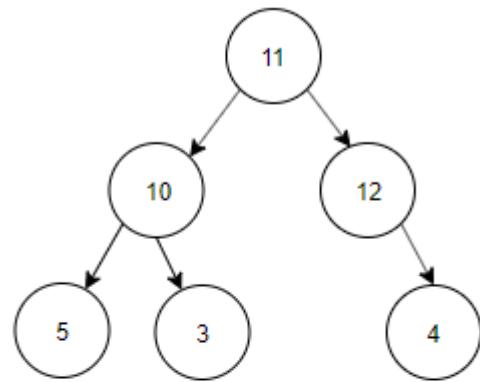
In order sequence of BSTs will always give **ascending order** of elements. Remaining all are true regarding BST.

You Try 2. BST Traversal

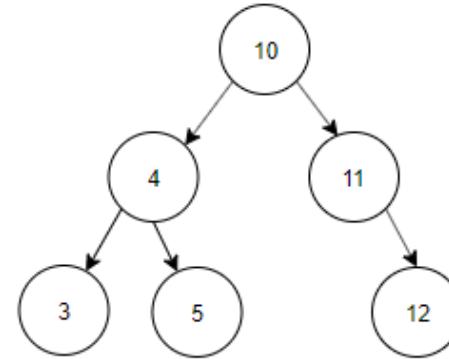


The pre-order traversal ($r \rightarrow T_L \rightarrow T_R$) of which of the following BSTs is 10, 4, 3, 5, 11, 12?

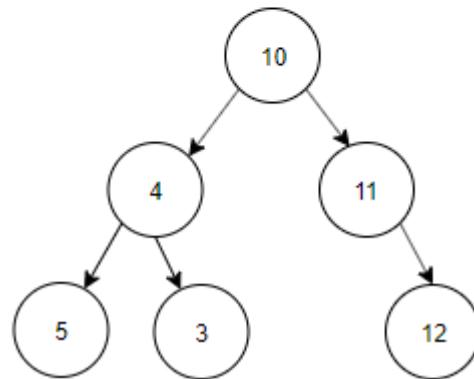
A)



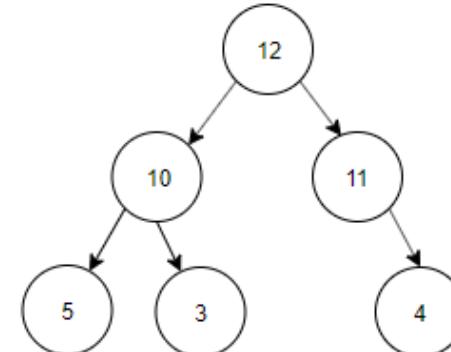
C)



B)



D)

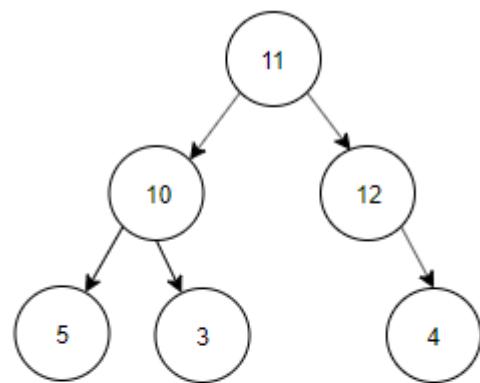


You Try 2 Solution. BST Traversal

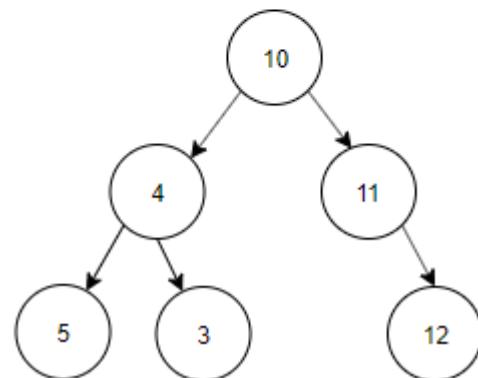


BSTs with pre-order traversal of 10, 4, 3, 5, 11, 12 is:

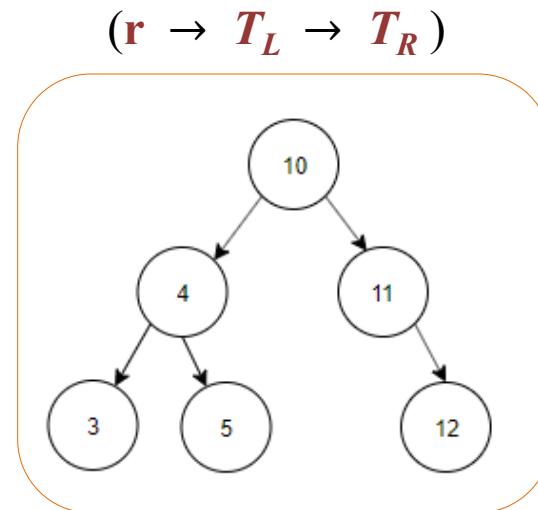
A)



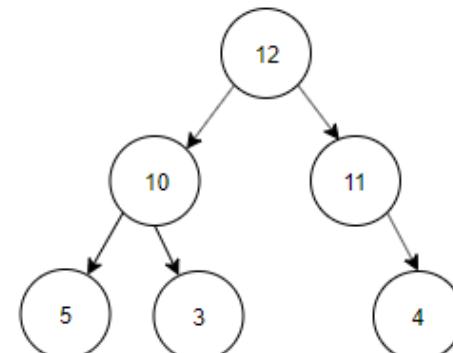
B)



C)

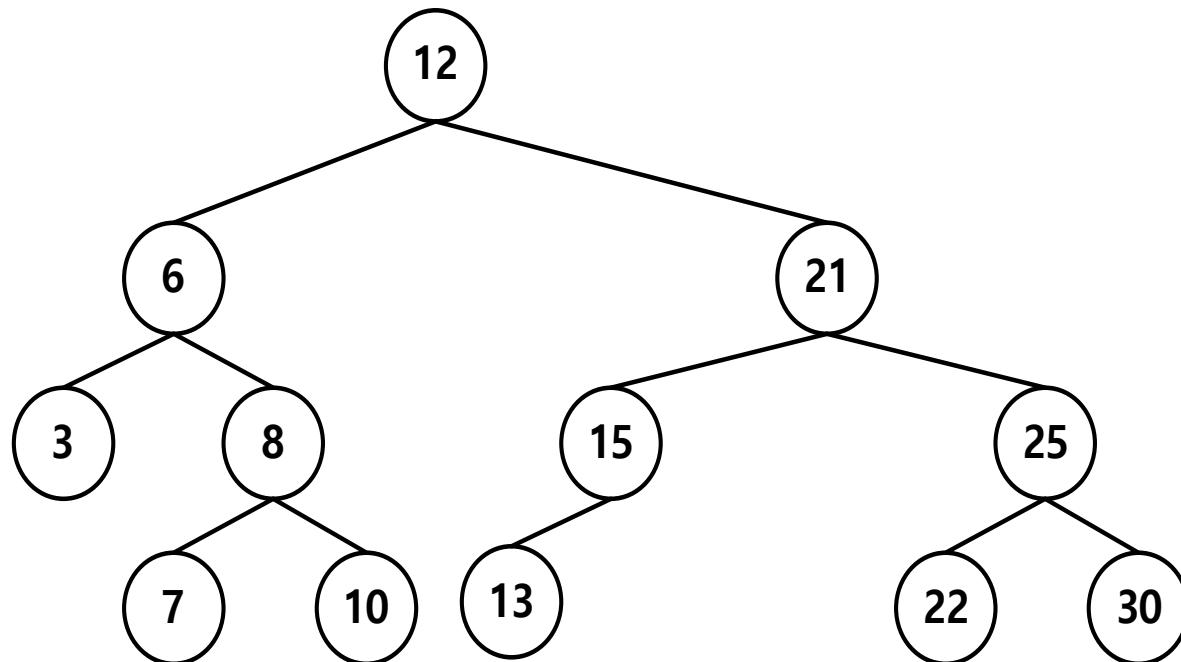


D)



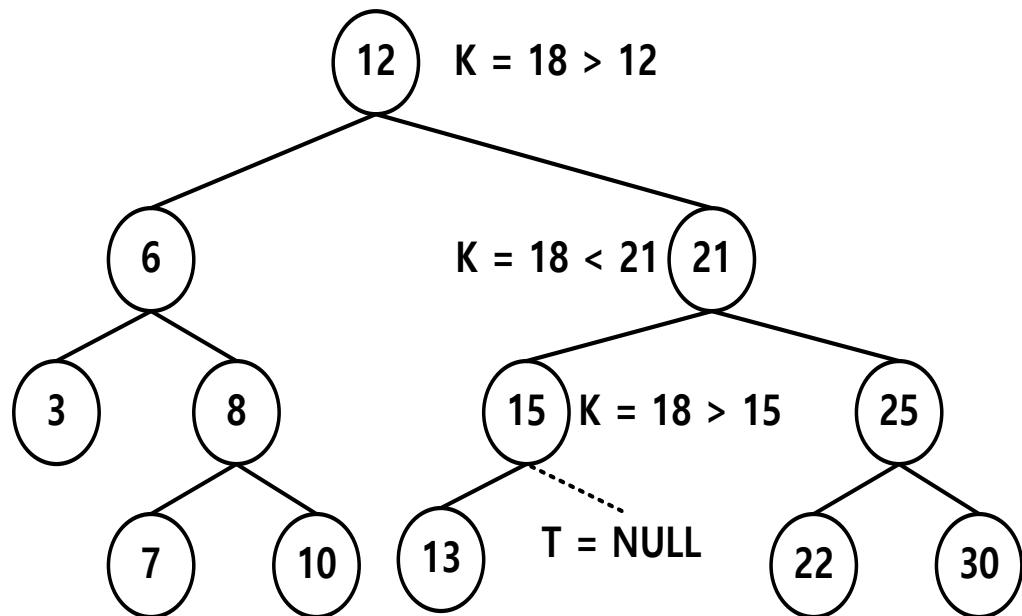
You Try 3. Search in BST

Show the steps of algorithm for searching the value of **18** in the following BST



You Try 3 Solution. Search in BST

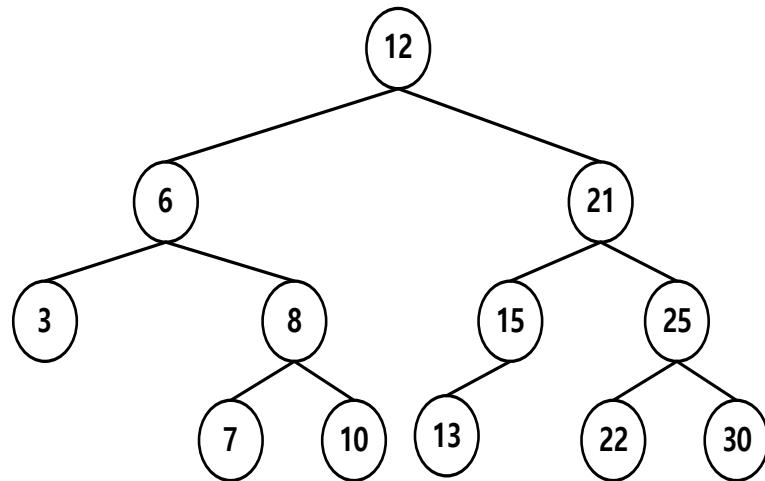
- Step 1. $K = 18 > 12$,
so traverse right.
- Step 2. $K = 18 < 21$,
so traverse left.
- Step 3. $K = 18 > 15$,
so traverse right.
- Step 4. $T = \text{NULL}$,
so return empty and terminate



You Try 4 . Insert in a BST



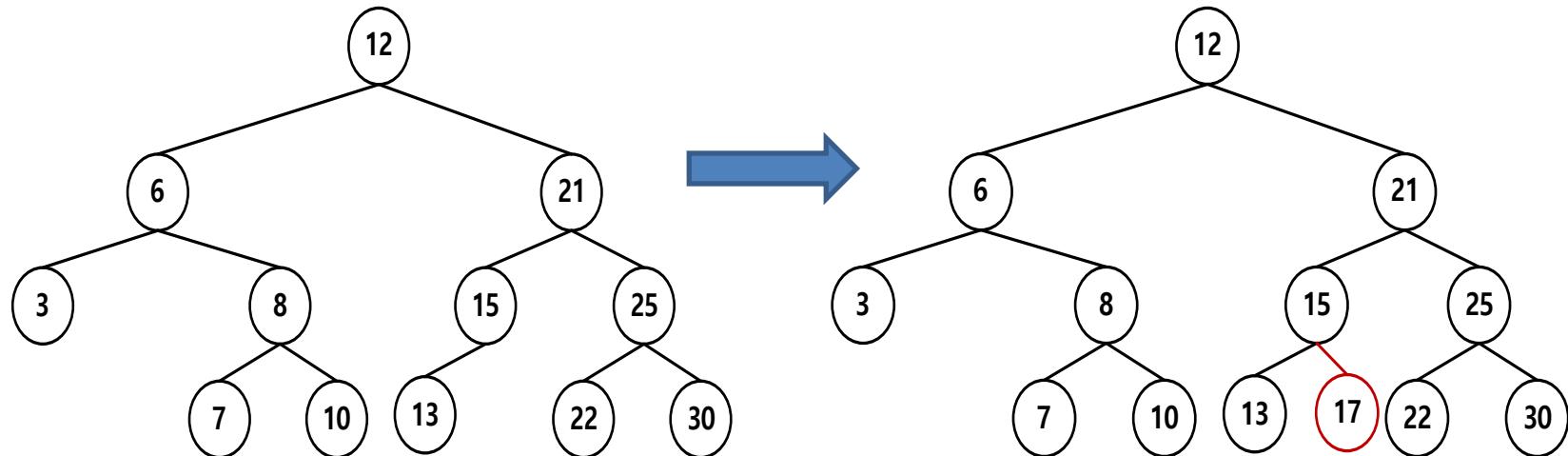
Insert the value of **17** in the following BST and write the steps:



You Try 4 Solution . Insert in a BST

Insert the value of 17 in the following BST and write the steps:

- 1) $17 > 12$, traverse to right subtree.
 - 2) $17 < 21$, traverse to left subtree.
 - 3) $17 > 15$, traverse to right subtree.
 - 4) right subtree is empty, insert 17 as the right child of 15.

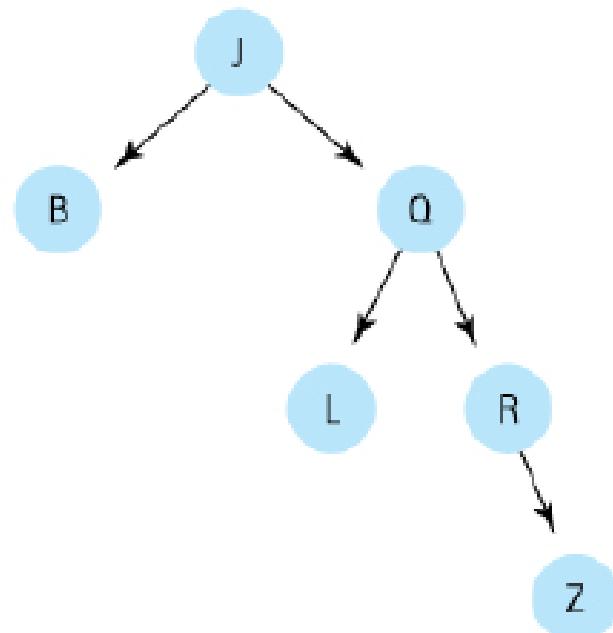


You Try 5. Deleting nodes from a BST



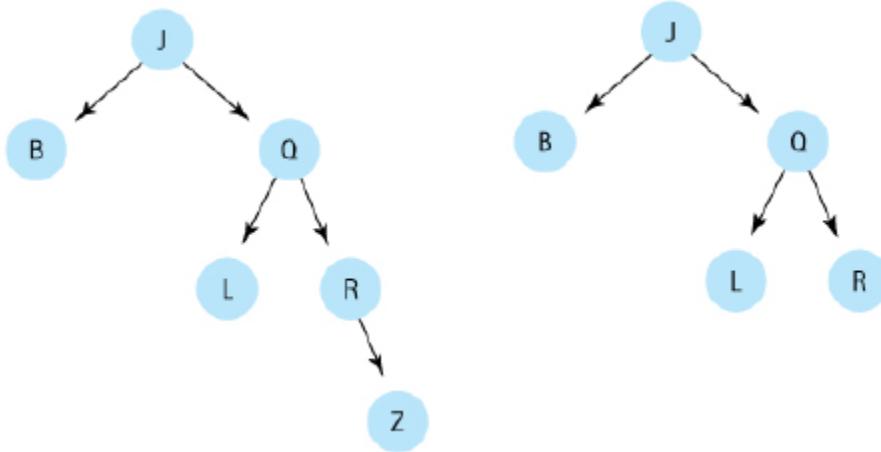
Do the sequence of deletion on the following BST, and redraw the BST after each operation. Delete the nodes containing:

1. Z
2. R
3. Q

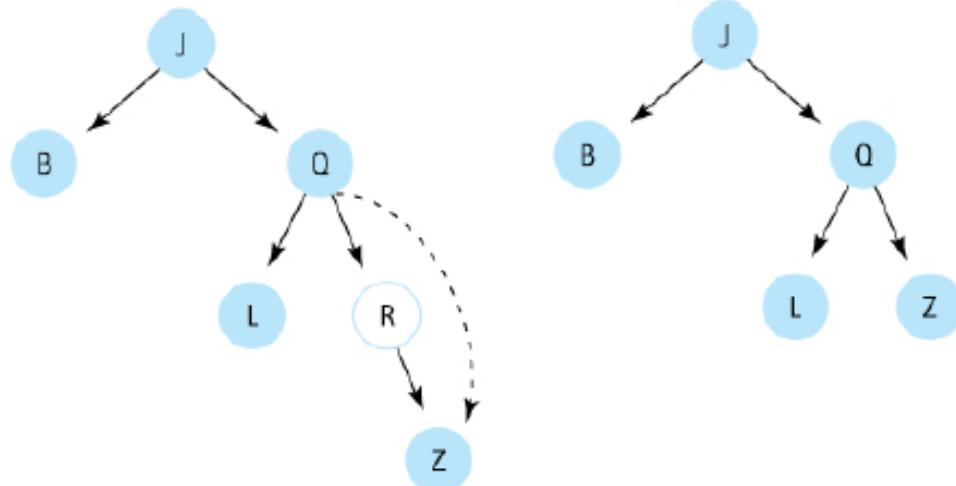


You Try 5 Solution. Deleting nodes from a BST

Deleting **Z**: a leaf node

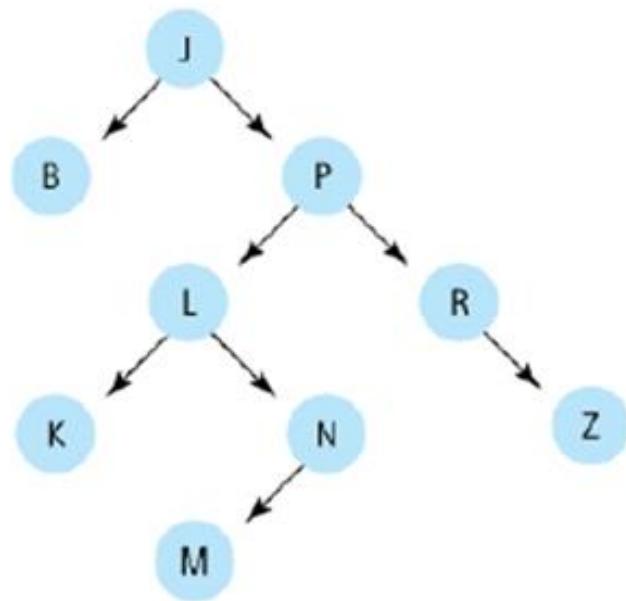
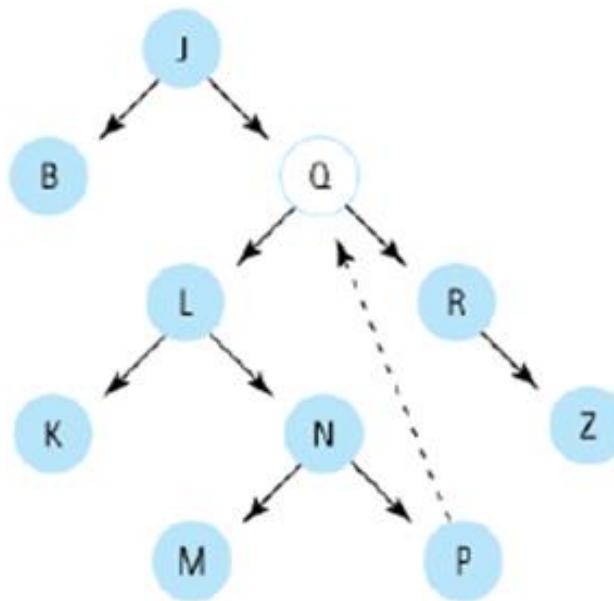


Deleting **R**: a node with one child



You Try 5 Solution. Deleting nodes from a BST

Deleting **Q**: a node with two children



The End of You Try Activities

Course: Data Structures and Algorithms

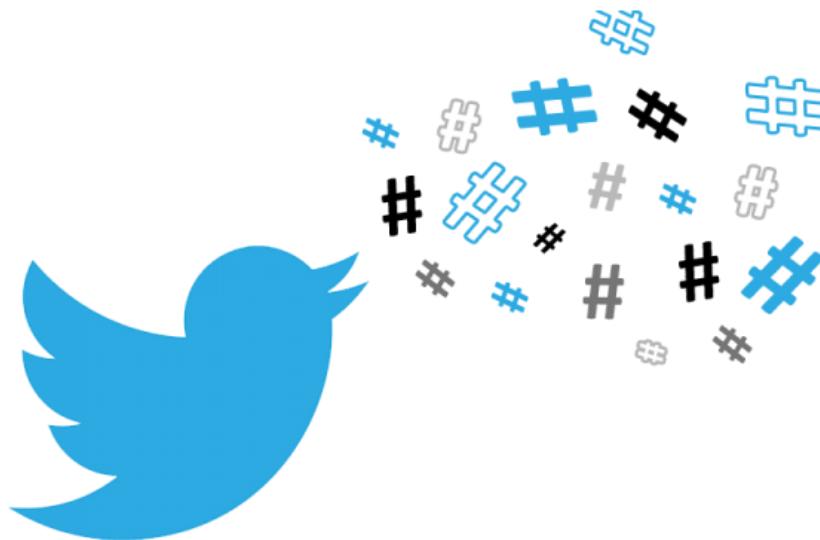
Instructor: Homeyra Pourmohammadali

AVL Trees

Motivation

Twitter hashtags are constantly changing; new ones are introduced, old ones become obsolete, and the searching of hashtags is a frequent action performed by users.

An AVL tree would be a good choice for this application to ensure efficient searching.



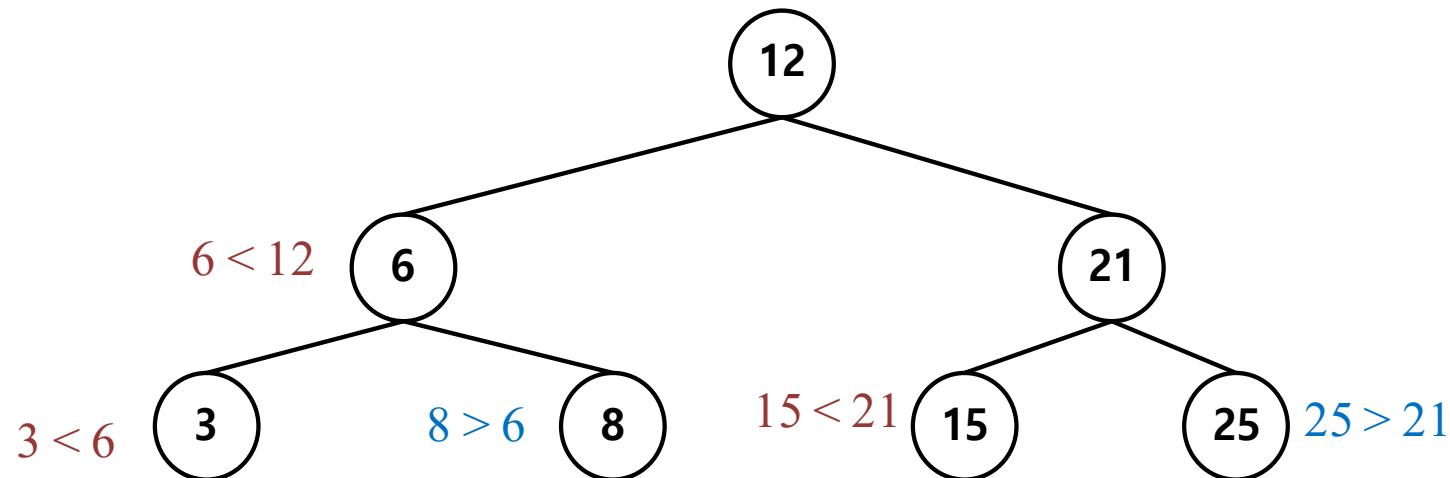
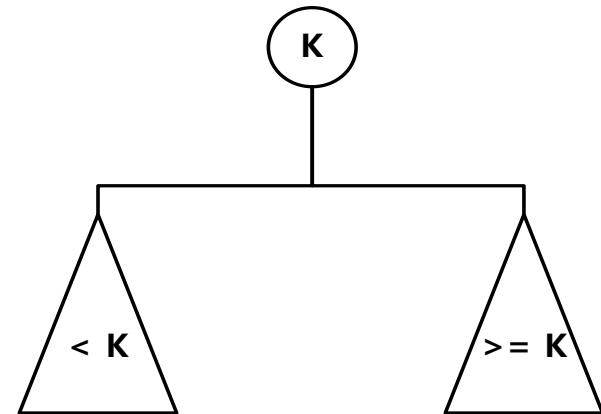
Learning Outcomes

By the end of this lecture you will be able to:

- find out how to check if a binary tree is balanced.
- define an AVL tree and its properties.
- define the four rotation operations to maintain the properties of AVL trees: LL, RR, LR and RL rotations.
- apply correct AVL rotation operations to unbalanced tree to regain balance.

Recall: Binary Search Trees

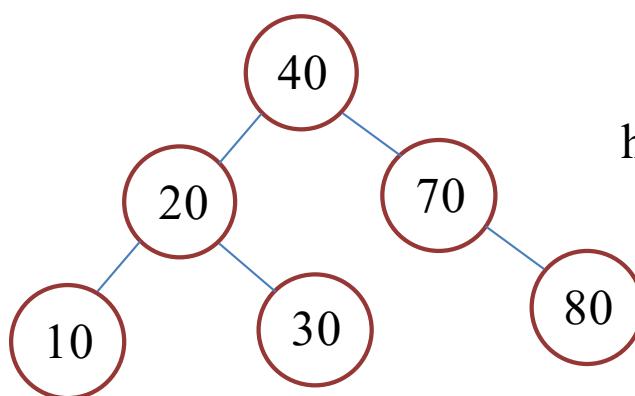
- Each node has a key value K
- All keys in the left subtree are $< K$
- All keys in the right subtree are $\geq K$



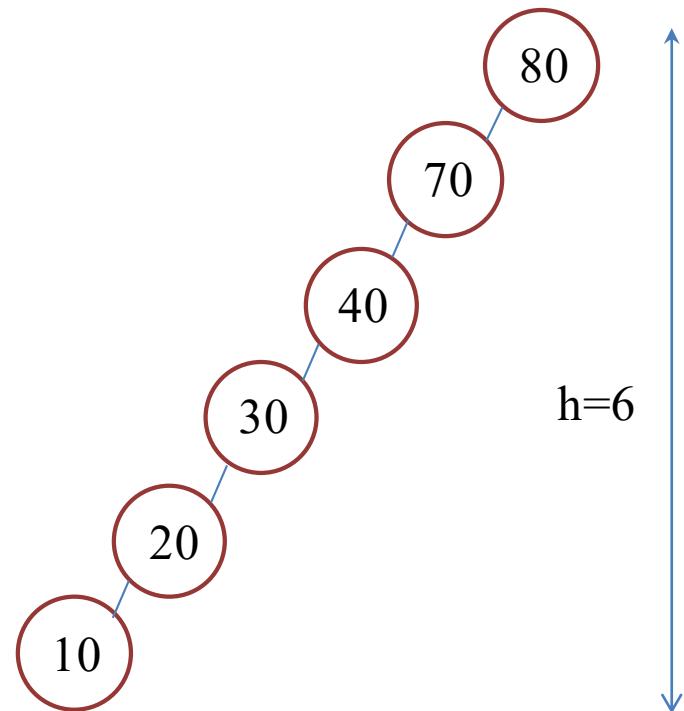
BSTs and Order of Keys Insertion

For instance, look at the following two BSTs of the same sets of keys.

Keys: 40, 20, 10, 70, 30, 80



Keys: 80, 70, 40, 30, 20, 10



For same set of keys different BSTs can be formed, depending on the order of the keys insertion.

Example 1. BSTs of similar sets of keys

Consider a set containing three integers: 4, 7, and 9. Different orders ($3! = 6$) are possible. For instance:

4, 7, 9

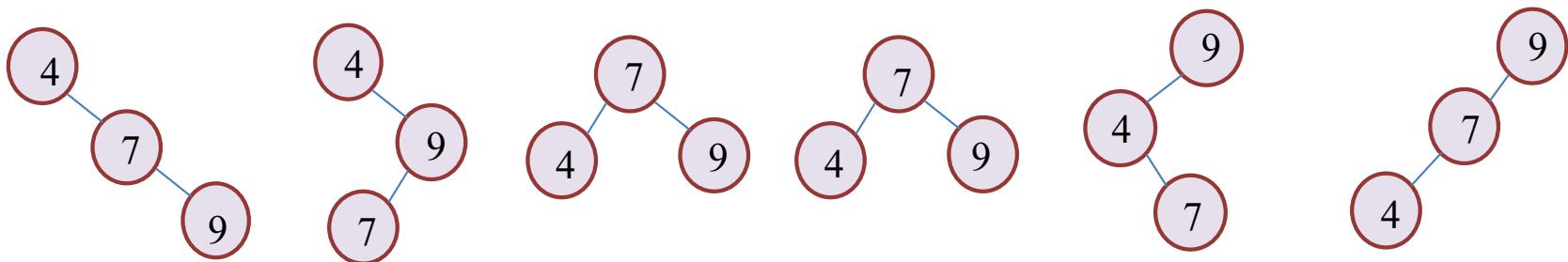
4, 9, 7

7, 9, 4

7, 4, 9

9, 4, 7

9, 7, 4



- Note that forms of BSTs are different.
- We desire a BST with minimum height, and lower time complexity.

BST Height and Time Complexity

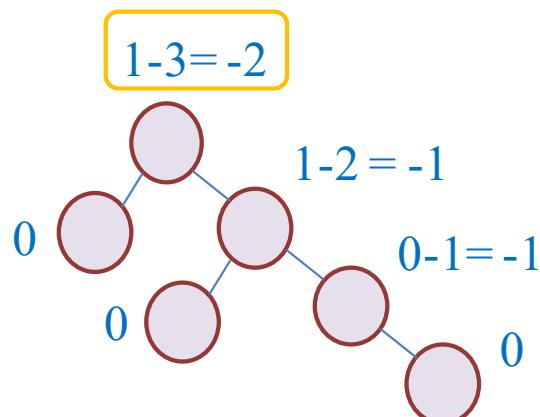
- The height of BST can change depending on the order that items are inserted (not known)
- The time complexity of searching an element can change between minimum $O(\log n)$ to maximum $O(n)$.
- Then one drawback of BST is that height is not under control and it depends on in which order the keys are inserted.
- Can BSTs be improved to resolve this issue? Can BSTs be converted to BSTs with minimum height?

Balanced Binary Tree (BT)

- A binary tree is said to be balanced if, the difference between the heights of left and right subtrees of every node in the tree is either -1, 0 or +1 (if the height of left and right children of every node differ by either -1, 0 or +1).
- Balance Factor = Height of T_L – Height of T_R
$$BF = H_L - H_R = \{ -1, 0, 1 \} \quad , \quad |BF| = |H_L - H_R| \leq 1$$
- If the balance factor of a node in BT is either 1, 0 or -1, then the node is balanced, otherwise it is unbalanced.

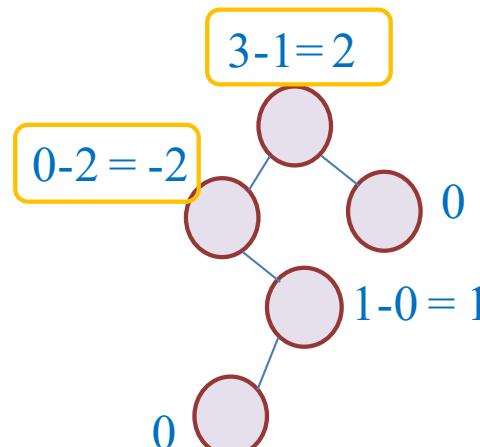
Example 2. Check for Balanced BT

$$BF = H_L - H_R = \{ -1, 0, 1 \} \quad , \quad |BF| = |H_L - H_R| \leq 1$$



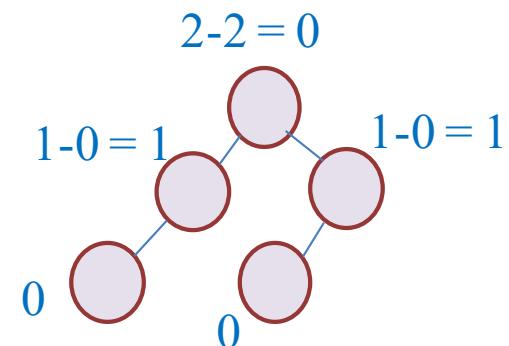
A

Unbalanced



B

Unbalanced



C

Balanced

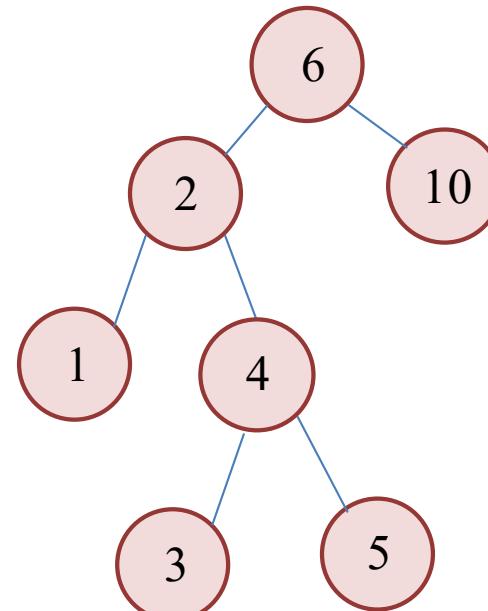
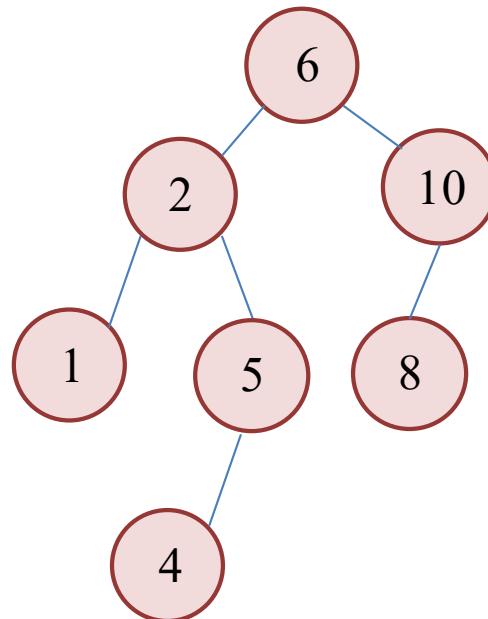
Note that each of the trees in Figure A and B has one imbalanced node.

AVL Tree

- An AVL tree is a balanced BST.
- Because the heights of the left and right subtrees of any node in a balanced BT differ by no more than 1, you can search an AVL tree almost as efficiently as a minimum-height BST.
- It is possible to rearrange any BST of n nodes to obtain a BST with the minimum possible height $\log_2(n + 1)$.
- An AVL tree maintains a height close to the minimum.
- Rotation can restore the balance in BST.

Example 3. AVL Tree-Property Checking

Which of the following BSTs is an AVL tree?



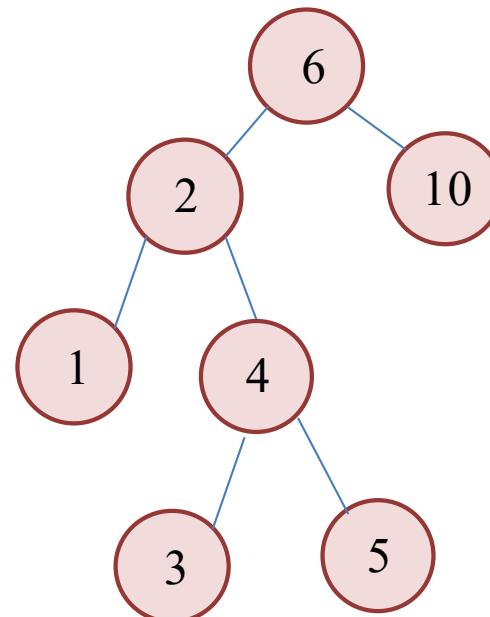
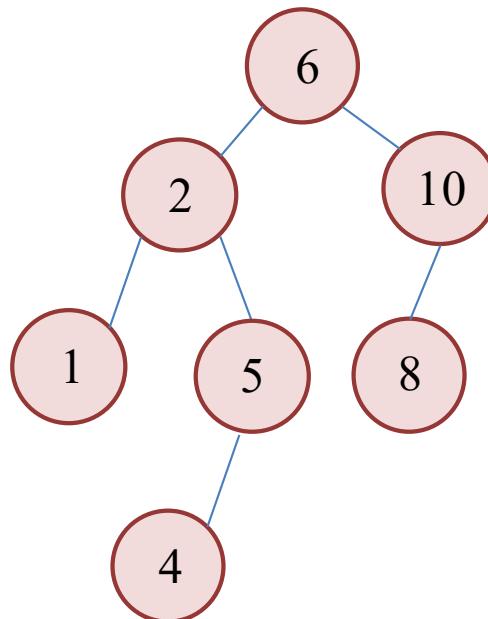
AVL Tree. The heights of two subtrees of any node differ by at most 1.

Not AVL Tree. The height of the left subtree of node with value of 6 is 3 and its right subtree is 1.

You Try 1. Balance Factor



Find the balance factor of each node of the following trees.



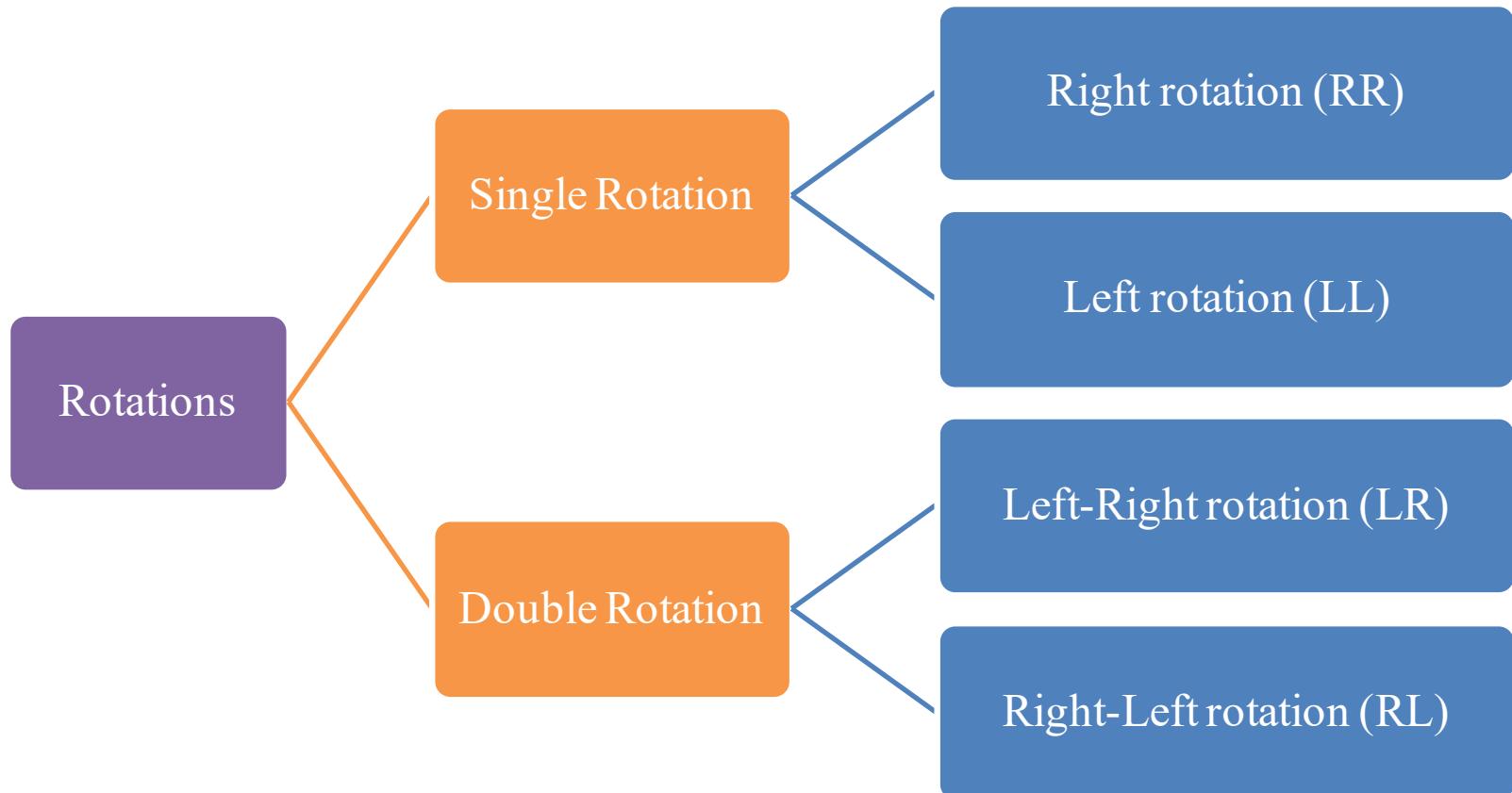
$$\text{Balance Factor} = \text{Height of } T_L - \text{Height of } T_R$$

AVL Tree Algorithm Strategy

- The basic strategy of the AVL algorithm is to monitor the shape of the BST.
- You insert or delete nodes just as you would for any BST, but after each insertion or deletion, you check that the tree is still an AVL tree.
- You determine whether any node in the tree has left and right subtrees whose heights differ by more than 1.

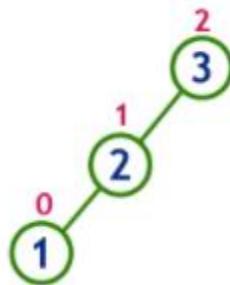
Maintaining AVL Tree Properties

- Transformations commonly used for restoring AVL properties:

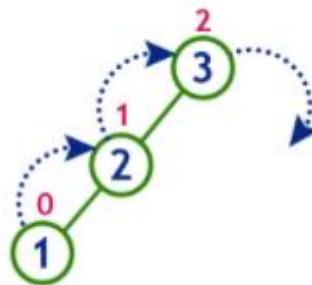


Single Right (RR) Rotation

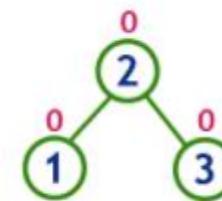
- Every node moves one position to **right** from the current position.
- **Example:** Insert 3, 2, 1



Unbalanced



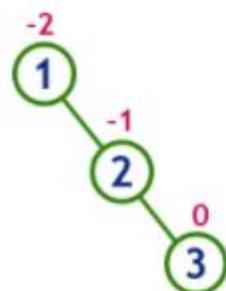
RR rotation



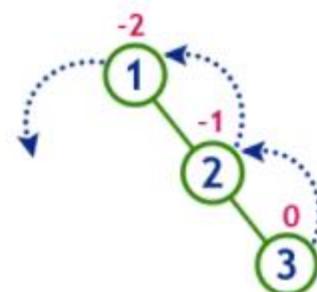
Balanced

Single Left (LL) Rotation

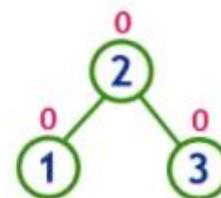
- Every node moves one position to **left** from the current position.
- **Example:** Insert 1, 2, 3



Unbalanced



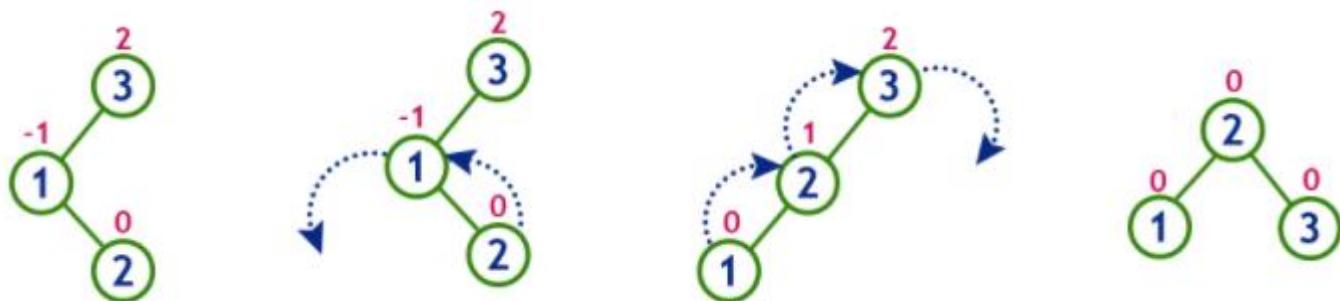
LL rotation



Balanced

Left-Right (LR) Rotation

- It is sequence of single left rotation followed by single right rotation. At first every node moves one position to left and one position to right from the current position.
- **Example:** Insert 3, 1, 2



Unbalanced

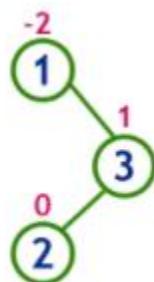
LL rotation

RR rotation

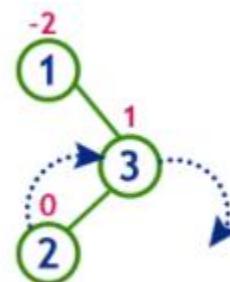
Balanced

Right-Left (RL) Rotation

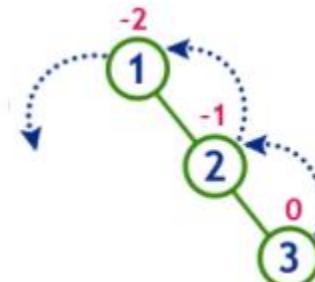
- It is sequence of single right rotation followed by single left rotation. At first every node moves one position to right and one position to left from the current position.
- **Example:** Insert 1, 3, 2



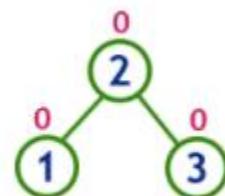
Unbalanced



RR rotation



LL rotation



Balanced

You Try 2. Rotation cases

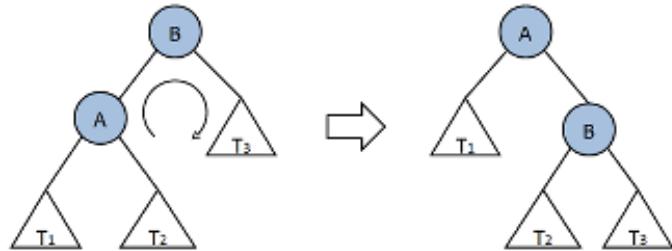


Make a BST with the following sequence of keys that are inserted in (a) and (b). Specify if the BST is balanced after each insertion and identify the type of rotation that is needed. Apply the appropriate rotation and make an AVL tree.

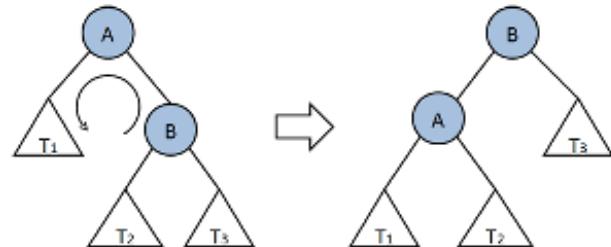
a) 6, 4, 2

b) 2, 4, 6

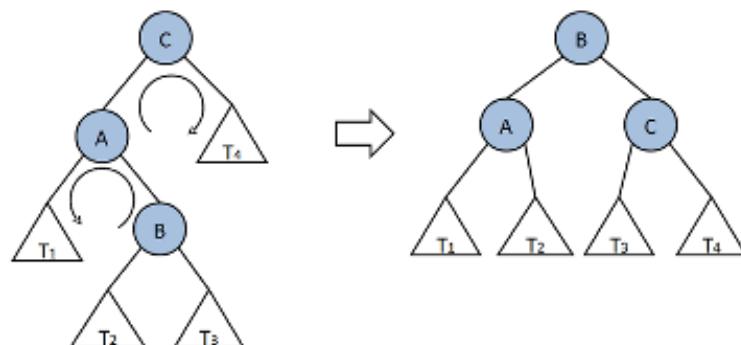
All Rotation Cases of BST Trees



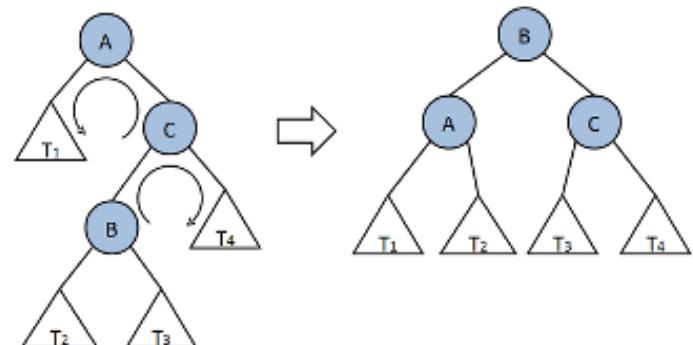
RR Rotation



LL Rotation



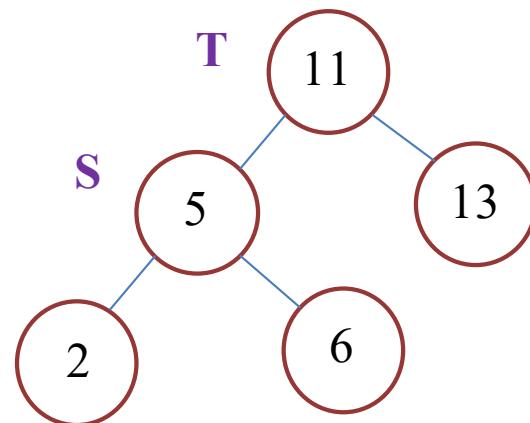
LR Rotation



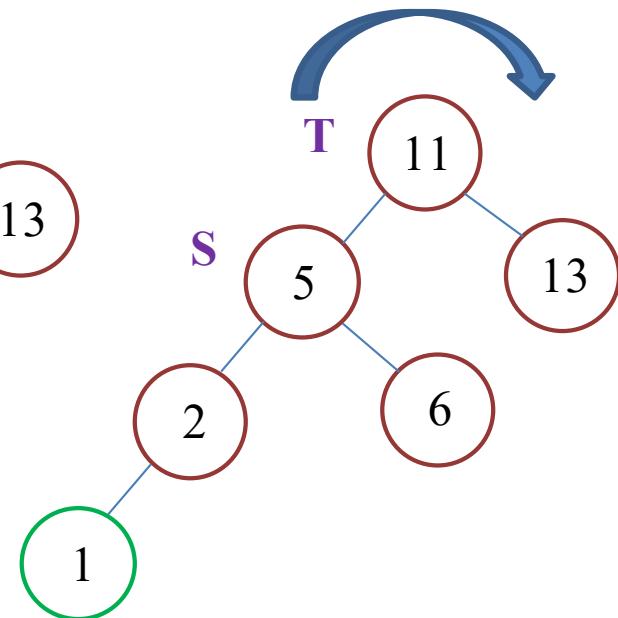
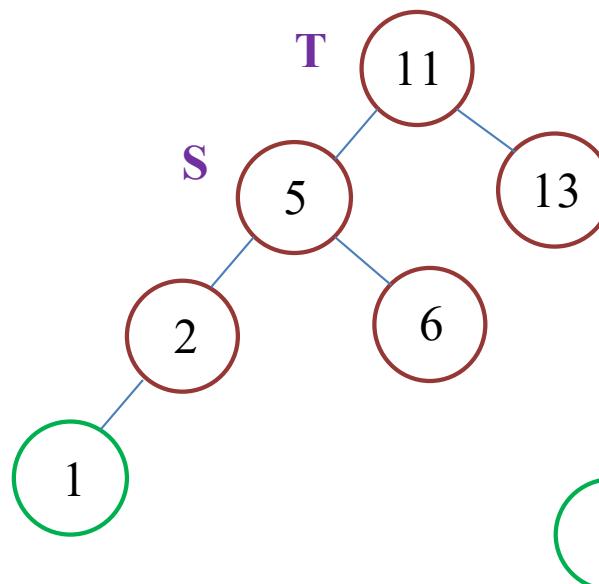
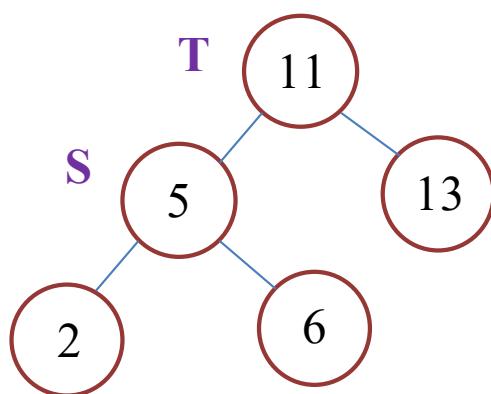
RL Rotation

Example 4. Single Rotation Case

Show what happens if 1 is inserted in the following AVL tree, and how the property of this tree is maintained.



Example 4 solution. Single Rotation Case

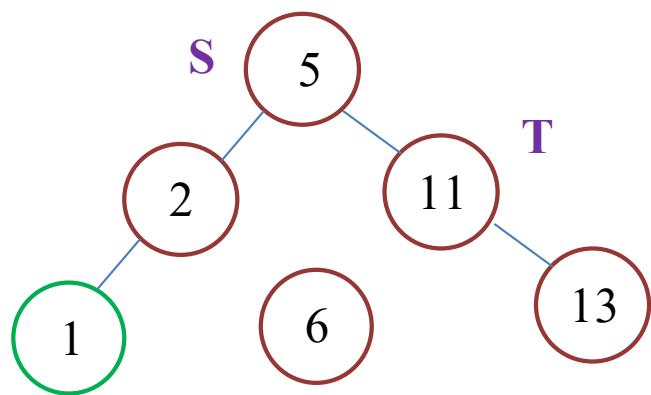


Balanced

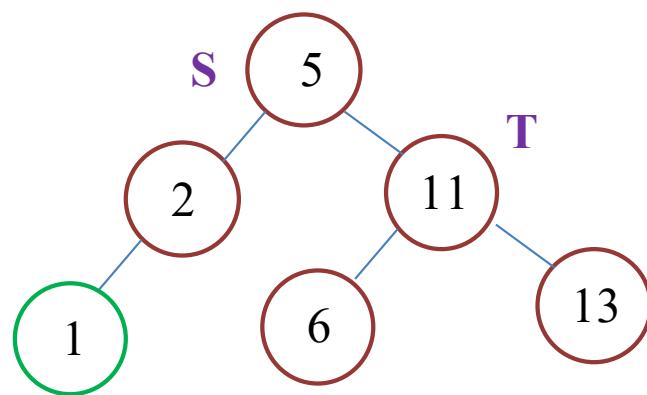
Insert 1-Unbalanced

Right Rotation

Example 4 Solution. Single Rotation Case



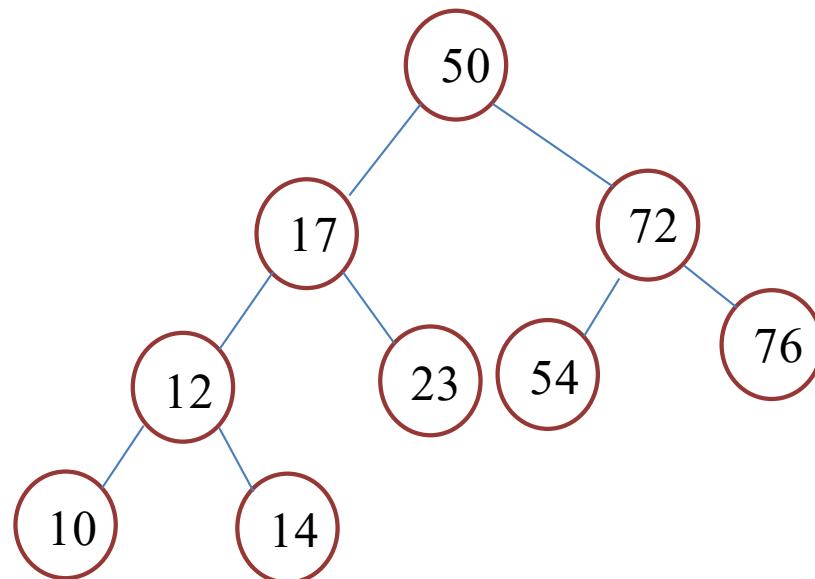
After Right Rotation



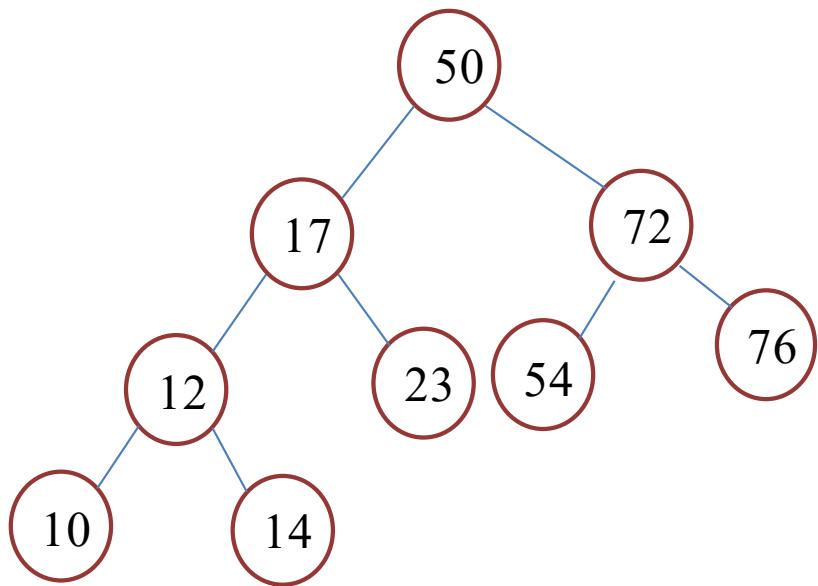
Balanced

Example 5. Right-Left Rotation Case

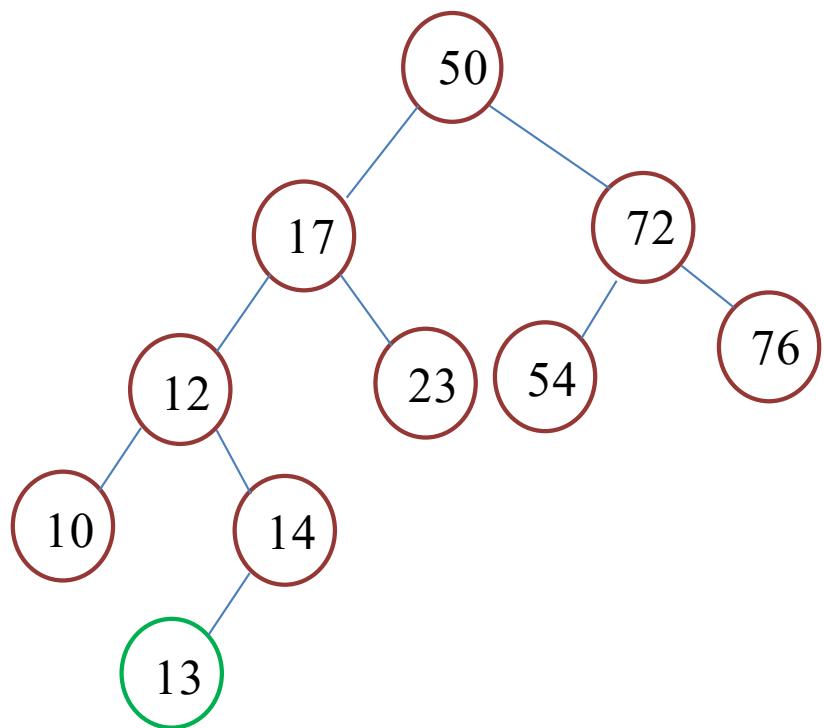
Show what happens if 13 is inserted in the following AVL tree, and how the property of this tree is maintained.



Example 5 Solution. Right-Left Rotation Case

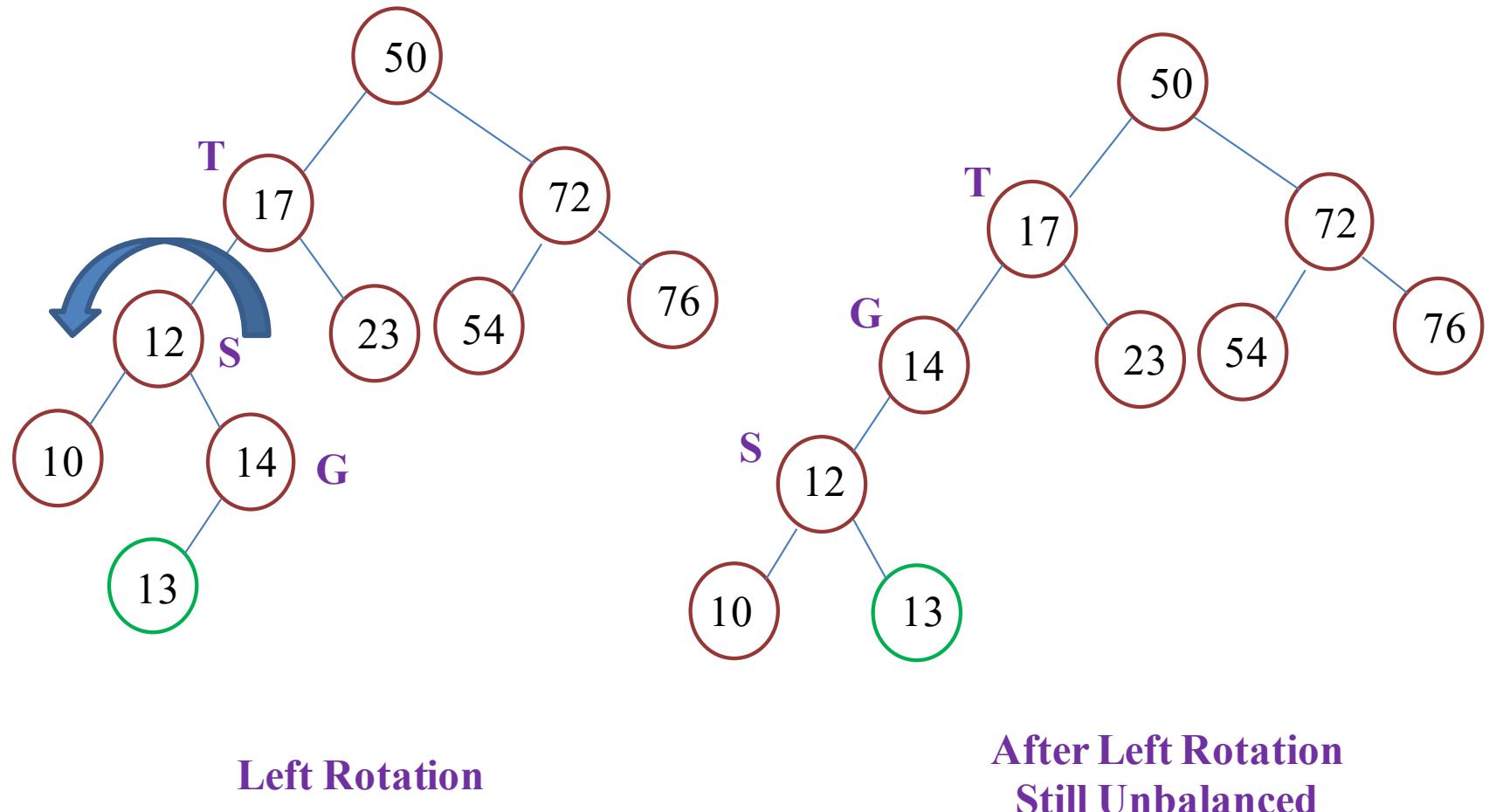


Balanced

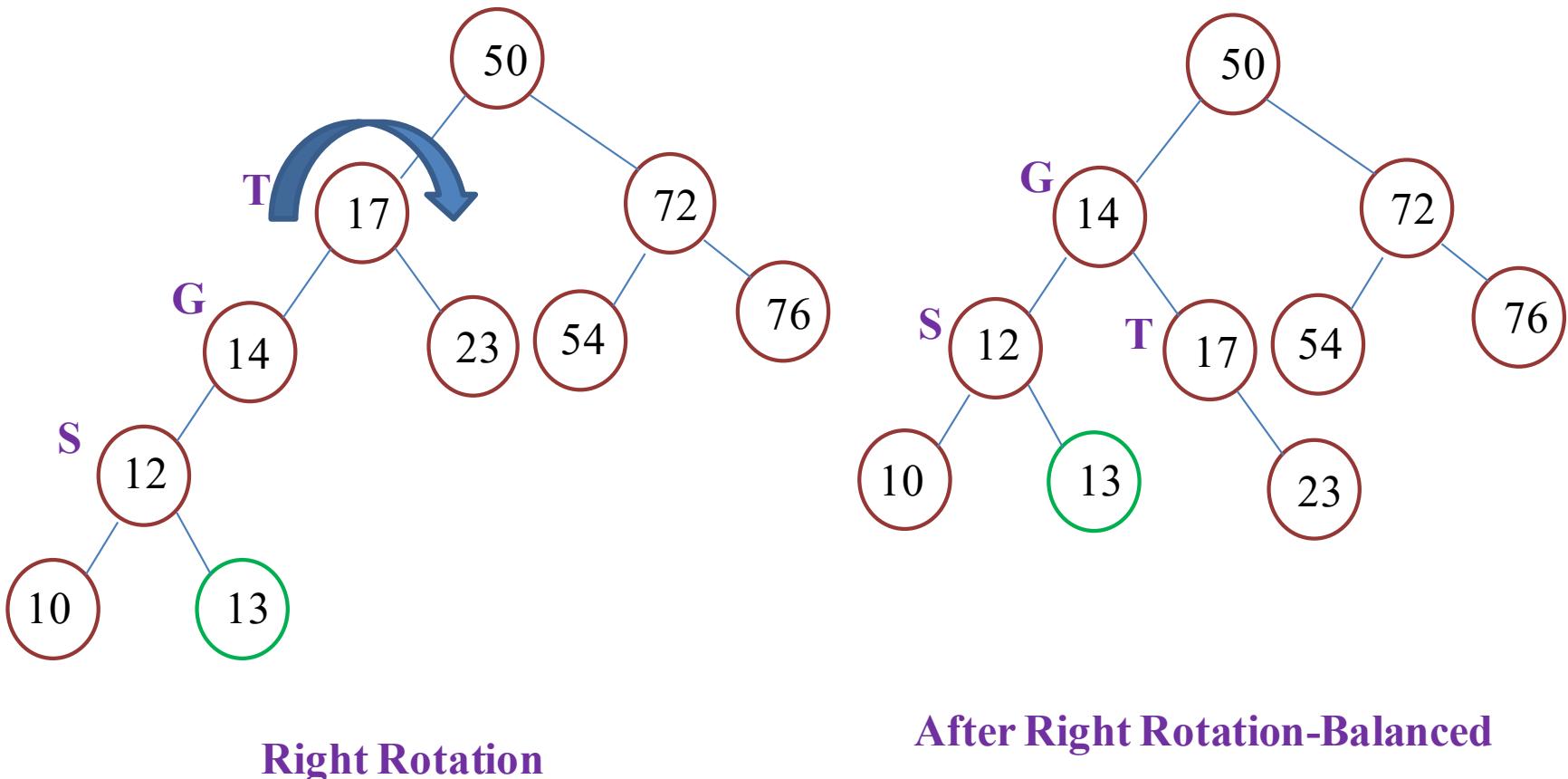


Insert 13-Unbalanced

Example 5 Solution. Right-Left Rotation Case



Example 5 Solution. Right-Left Rotation Case

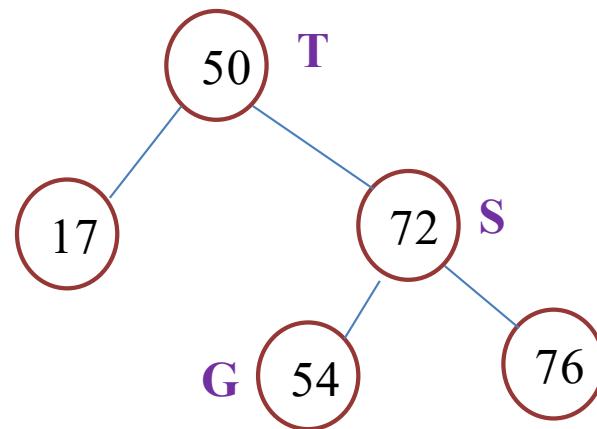


Note: The rotation operations were not applied to a direct child of the unbalanced node, but to the great-grandchild. Two rotation operations required to restore balance.

You Try 3. Right-Left Rotation Case



Show what happens if 60 is inserted in the following AVL tree, and how the property of this tree is maintained.



Delete Operation

- Deleting nodes from an AVL tree may also causes imbalance.
- The procedure for the imbalance detection and reshaping the tree to restore balance is the same as the procedure for insertion.
- If removing a node causes an imbalance in an AVL tree, identify the unbalanced node T that is nearest to the deleted node and perform one of the afore mentioned operations, depending on where the node was deleted with respect to T.

AVL Tree Implementation

Single Right Rotation Implementation

Pseudocode:

```
Node RotateRight(Node T)
```

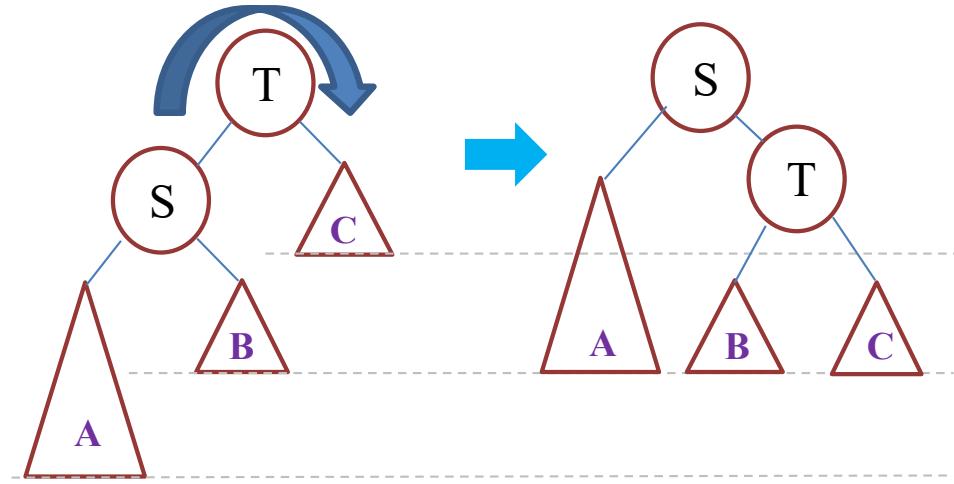
Node S = left child of T

Node B = right child of S

Right child of S = T

Left child of T = B

return S



```
TreeNode* RotateRight(TreeNode* T) // C++ example for rotate right  
// Returns the tree node resulting from a right rotation.
```

```
{  
    TreeNode* S = T->left;  
    TreeNode* B = S->right;  
    S->right = T;  
    T->left = B;  
    return S;  
}
```

Single Left Rotation Implementation

Pseudocode:

```
Node RotateLeft(Node T)
```

```
    Node S = right child of T
```

```
    Node B = left child of S
```

```
    Left child of S = T
```

```
    Right child of T = B
```

```
    return S
```

```
TreeNode* RotateLeft(TreeNode* T) // C++ example for rotate left
```

```
// Returns the tree node resulting from a left rotation.
```

```
{
```

```
    TreeNode* S = T->right;
```

```
    TreeNode* B = S->left;
```

```
    S->left = T;
```

```
    T->right = B
```

```
    return S;
```

```
}
```

Right-Left Rotation Implementation

Pseudocode:

```
Node RotateRightLeft(Node T)
```

Node S = right child of T

Right child of T = RotateRight(S)

return RotateLeft(T)

```
TreeNode* RotateRightLeft(TreeNode* T) // C++ example for rotate right-left
// Returns the tree node resulting from a right-left rotation
{
    TreeNode* S = T->right;
    T->right = RotateRight(S);
    return RotateLeft(T);
}
```

Left-Right Rotation Implementation

Pseudocode:

```
Node RotateLeRight(Node T)
```

```
Node S = left child of T
```

```
Left child of T = RotateLeft(S)
```

```
return RotateRight(T)
```

```
TreeNode* RotateLeftRight(TreeNode* T) // C++ example for rotate left-right
// Returns the tree node resulting from a left-right rotation
{
    TreeNode* S = T->left;
    T->left = RotateLeft(S);
    return RotateRight(T);
}
```

Finding Heights Difference

```
int Difference(TreeNode* T) const // C++ example for checking imbalance
// Returns the difference between the heights of the left and right subtrees of T.
// Assumes the given TreeNode* T is not null.
{
    return Height(T->left) - Height(T->right);
}
int Height(TreeNode* T) const
// Returns the height of a tree T.
{
    if (T == null)
        return 0;
    else {
        int heightLeft = Height(T->left);
        int heightRight = Height(T->right);
        if (heightLeft > heightRight)
            return heightLeft + 1;
        else
            return heightRight + 1;
    }
}
```

Check and Balance Subtree

Pseudocode Algorithm

Node Balance(Node T)

If Difference(T) is greater than 1:

If Difference(T's left subtree) is greater than 1:

 return RotateRight(T)

Else

 return RotateLeftRight(T)

Else if Difference(T) is less than -1:

If Difference(T's right subtree) is less than 0:

 return RotateLeft(T)

Else

 return RotateRightLeft(T)

Else

 return T

Check and Balance Subtree

```
TreeNode* Balance(TreeNode* T) // C++ example
// Checks and balances the subtree T.
{
    int balanceFactor = Difference(T); // BF of a balanced node must be either -1, 0 or 1.
    if (balanceFactor > 1) {
        if (Difference(T->left) > 1)
            return RotateRight(T);
        else
            return RotateLeftRight(T);
    }
    else if (balanceFactor < -1) {
        if (Difference(T->right) < 0)
            return RotateLeft(T);
        else
            return RotateRightLeft(T);
    }
    else
        return T;
}
```

Upon returning from each recursive call to **Insert**, we invoke **Balance** to determine if our insertion caused an imbalance and to perform the proper rotation operation, depending on which subtree the new node was inserted into.

Insert Implementation

```
void Insert(TreeNode*& tree, ItemType item)
// Inserts item into tree.
// Post: item is in tree; search property is maintained.
{
    if (tree == NULL)
        { // Insertion place found.
            tree = new TreeNode;
            tree->right = NULL;
            tree->left = NULL;
            tree->info = item;
        }
    else if (item < tree->info) {
        Insert(tree->left, item); // Insert in left subtree.
        tree->left = Balance(tree->left);
    }
    else {
        Insert(tree->right, item); // Insert in right subtree.
        Tree->right = Balance(tree->right); // Calls to Balance
    }
}
```

Insert New Item in AVL Tree

```
void TreeType::PutItem(ItemType item)
// Calls the recursive function Insert to insert item into tree.
{
    Insert(root, item);
    root = Balance(root); // calls to Balance on the root node
}
```

Next Lecture

We focus on:

- Breadth-first and depth-first strategies
- Heaps

Readings/ Study Reference

Course Textbook: “*Data Structure and Algorithms in a Nutshell*” A. Wong et al.

Chapter 7. Trees

Section 7.4. AVL Trees

The End of Lecture

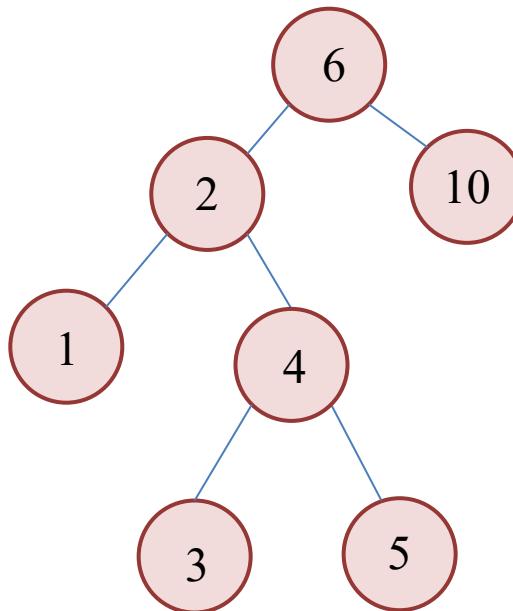
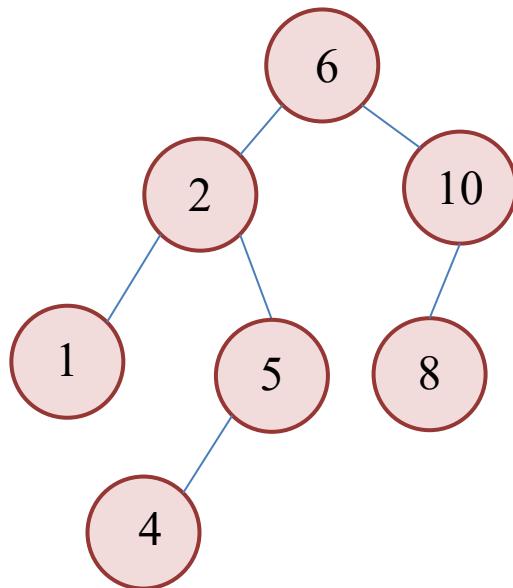
Course: Data Structures and Algorithms

Instructor: Homeyra Pourmohammadali

**You Try Questions and Solutions
AVL Trees**

You Try 1. Balance Factor

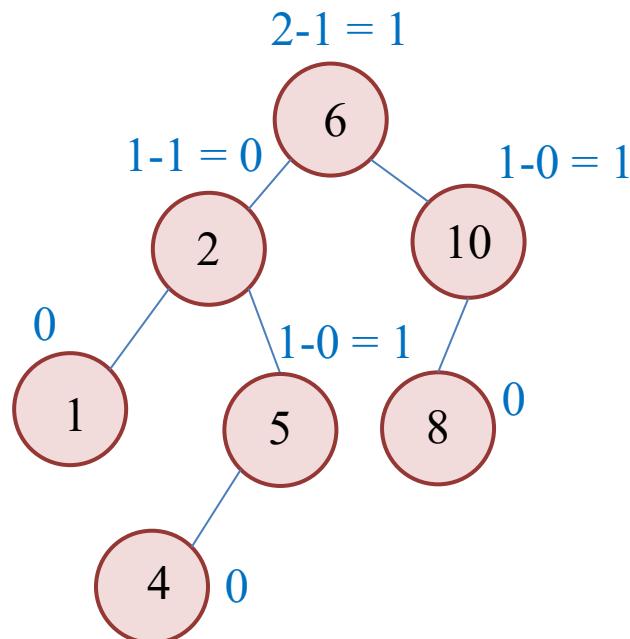
Find the balance factor of each node of the following trees.



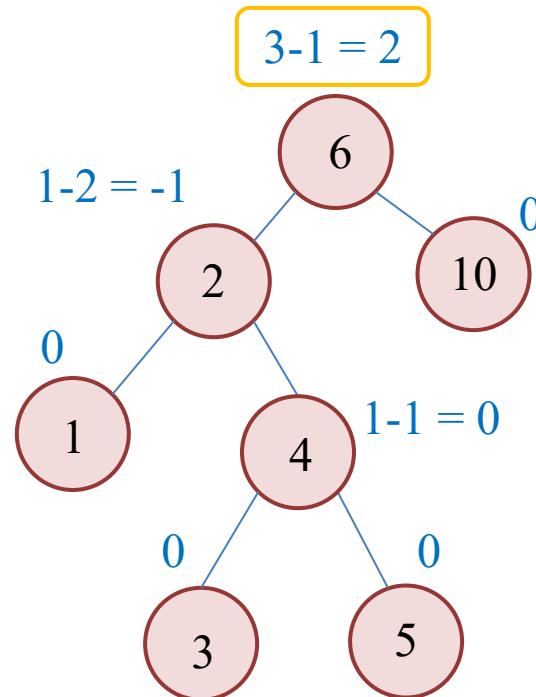
$$\text{Balance Factor} = \text{Height of } T_L - \text{Height of } T_R$$

You Try 1 Solution. Balance Factor

Find the balance factor of each node of the following trees.



Balanced



Unbalanced

$$\text{Balance Factor} = \text{Height of } T_L - \text{Height of } T_R$$

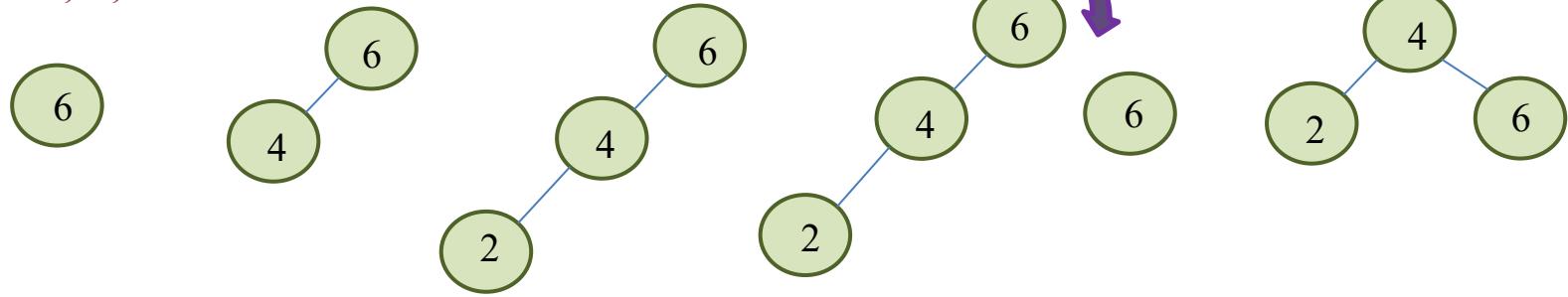
You Try 2. Rotation Cases

Make a BST with the following sequence of keys that are inserted in (a) and (b). Specify if the BST is balanced after each insertion and identify the type of rotation that is needed. Apply the appropriate rotation and make an AVL tree.

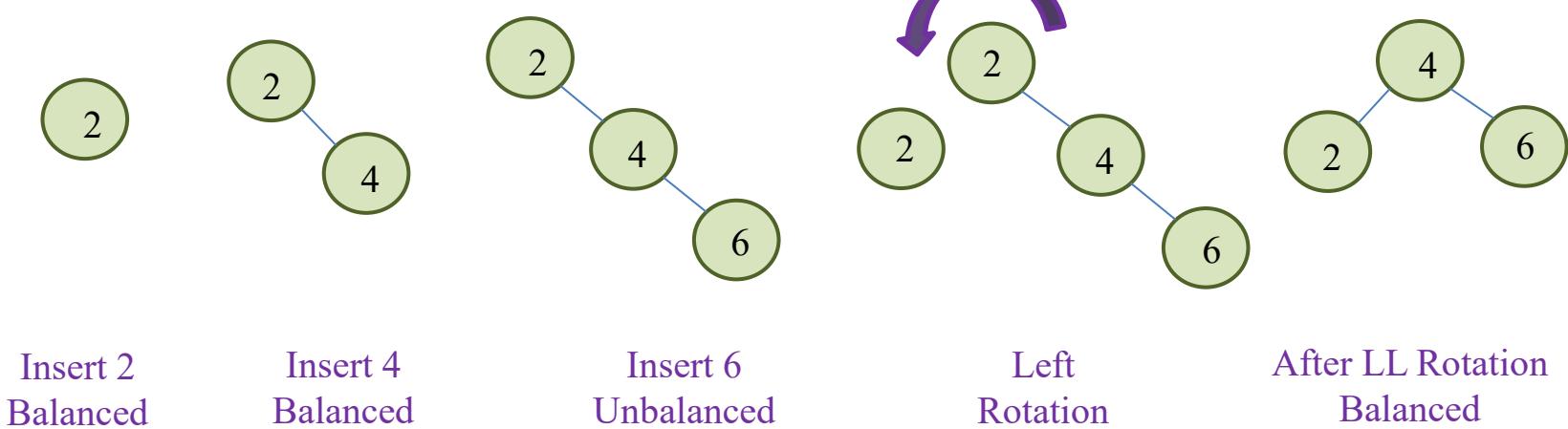
- a) 6, 4, 2
- b) 2, 4, 6

You Try 2 Solution. Rotation Cases

a) 6, 4, 2

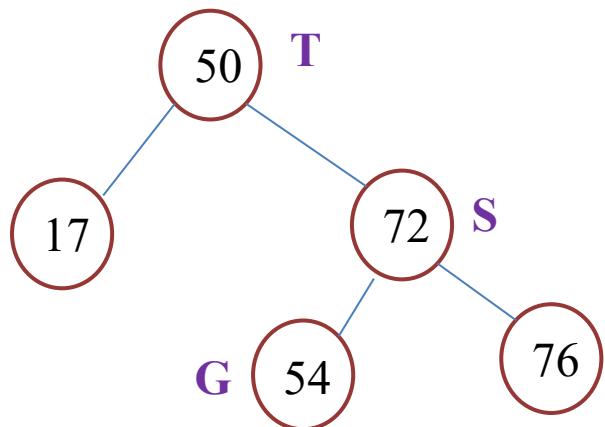


b) 2, 4, 6

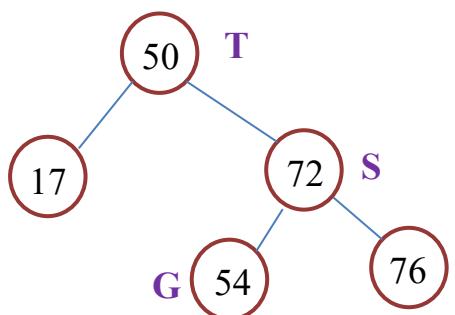


You Try 3. Right-Left Rotation Case

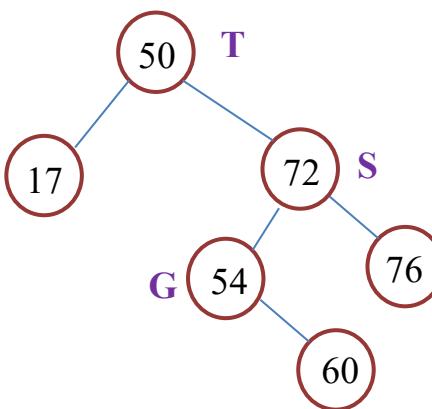
Show what happens if 60 is inserted in the following AVL tree, and how the property of this tree is maintained.



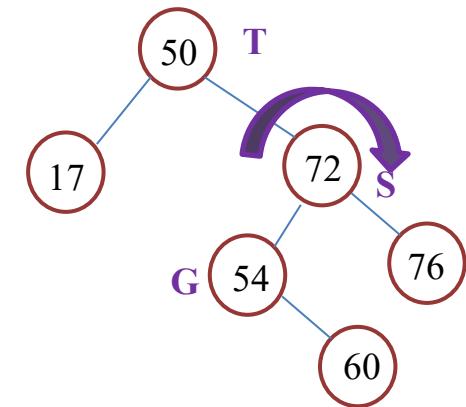
You Try 3 Solution. Right-Left Rotation Case



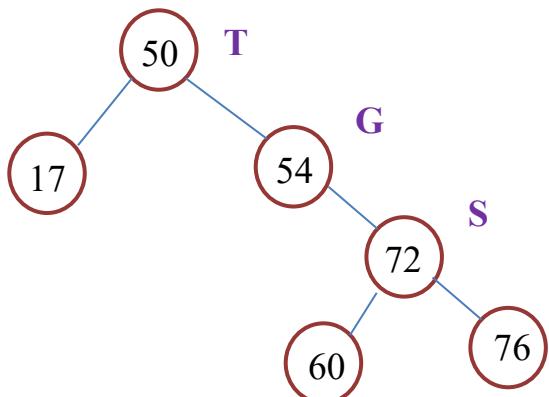
Balanced



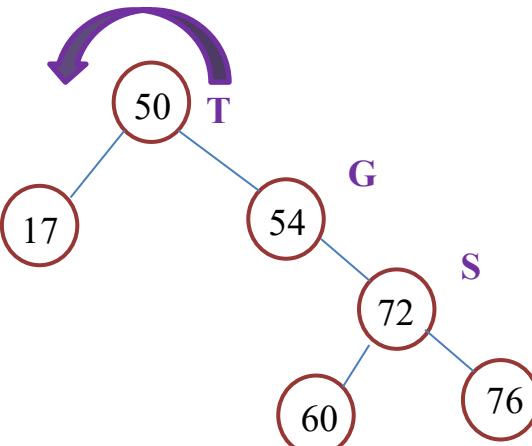
Insert 60-Unbalanced



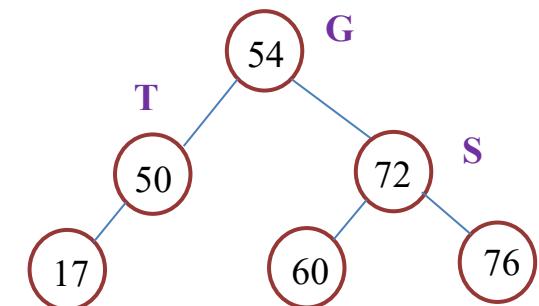
Right Rotation



Still Unbalanced



Left Rotation



After RL Rotation- Balanced

The End of You Try Activities

Course: Data Structures and Algorithms

Instructor: Homeyra Pourmohammadali

Heaps and Priority Queues

Motivation

What ADT should the ER staff use for their patients?

A queue would enable treatment of patients in the order of arrival.

How do the ER staff should assign some measure of urgency, or priority, to the patients waiting for treatment?



<https://cmajnews.com/2018/06/26/call-for-family-doctors-to-have-more-emergency-medicine-training-sparks-debate-cmaj-109-5627/>

Motivation

How does a secretary decide which jobs should be done first?

For instance, the jobs are processed in order of the employee's importance in the company.

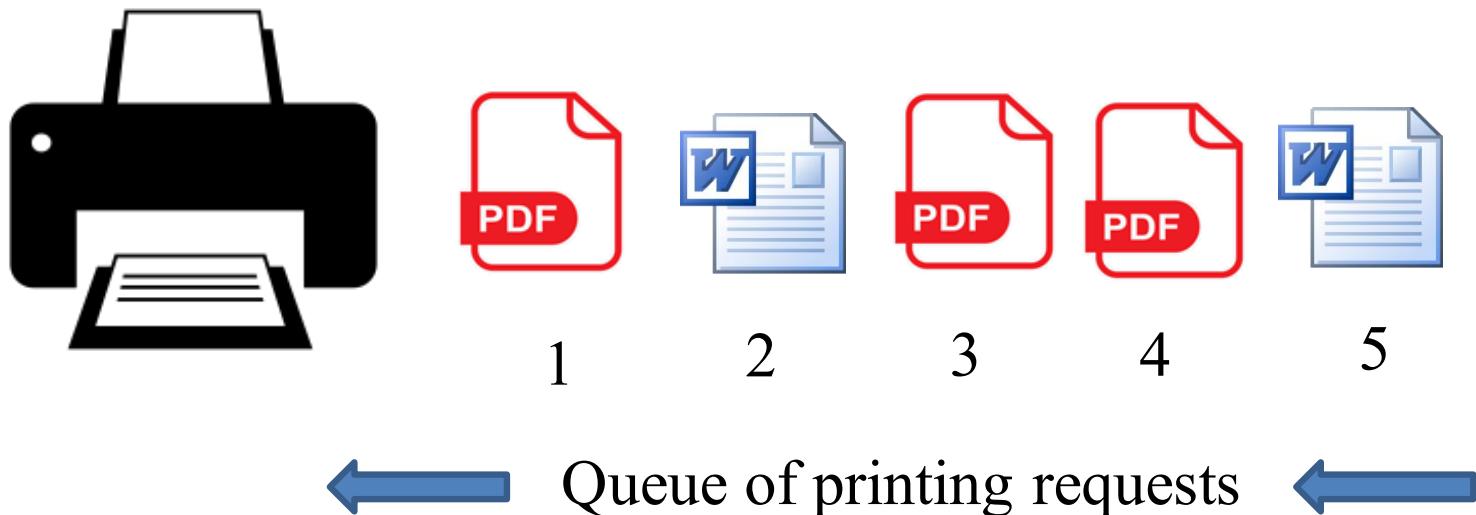
The secretary completes the president's work before starting the VP's job.



<https://www.istockphoto.com/search/2/image?phrase=restrained+secretary>

Motivation

- Some documents need to be prioritized to be printed.
- Consider a 1-page job versus a 200-page job in the queue of printing tasks. Which one is more reasonable to be done first?



Learning Outcomes

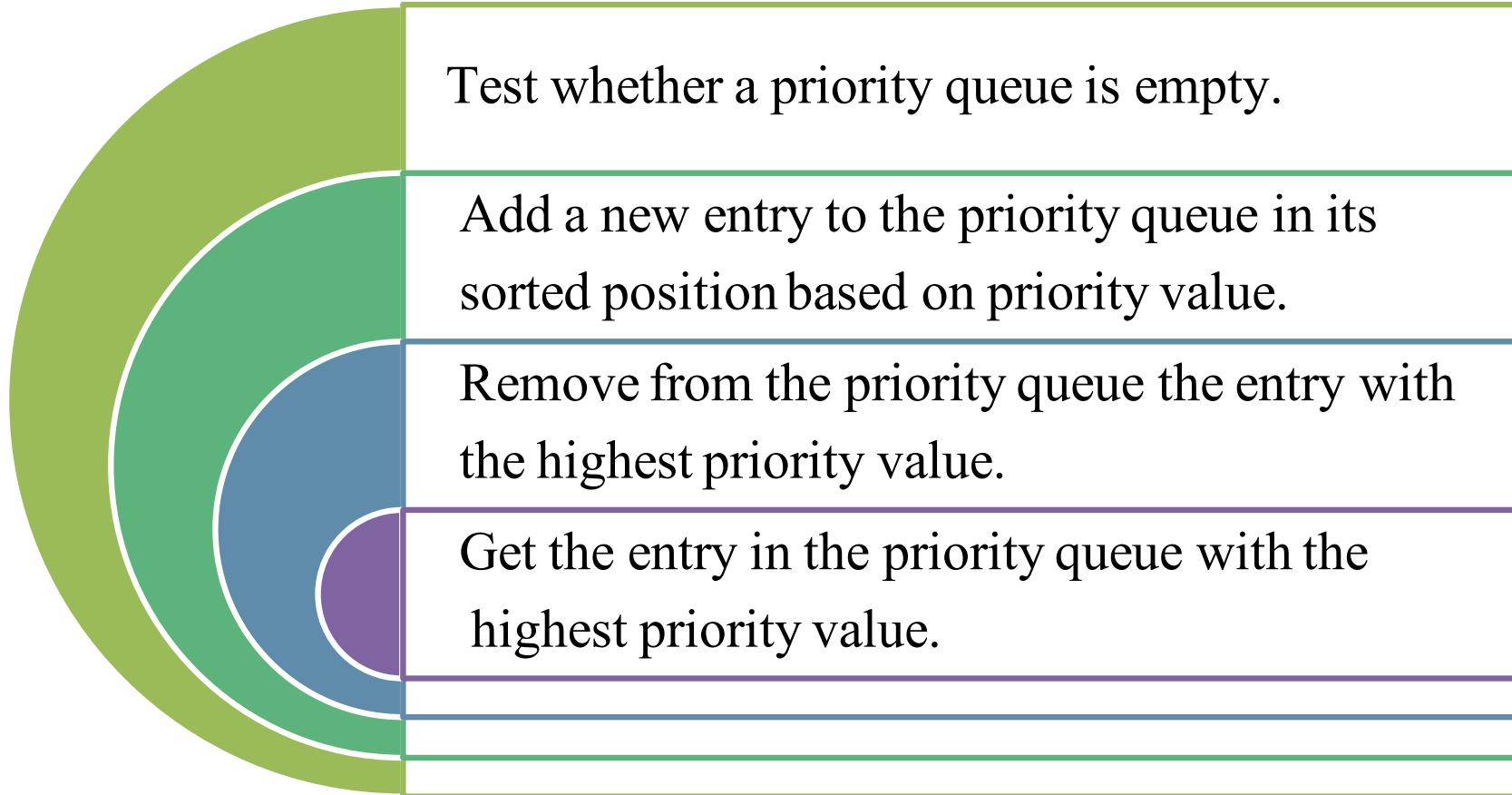
By the end of this lecture you will be able to:

- describe a priority queue.
- describe the shape and order properties of a heap.
- understand the main operations of heap.

Priority Queue

- You can organize data by priority.
- A **priority value** indicates, for example, a patient's priority for treatment or a task's priority for completion. The priority value becomes a part of the item that you insert into an ADT. (e.g. 1 to 10).
- You ask the ADT for the item that has the highest priority. Such an ADT is known as a **priority queue**.

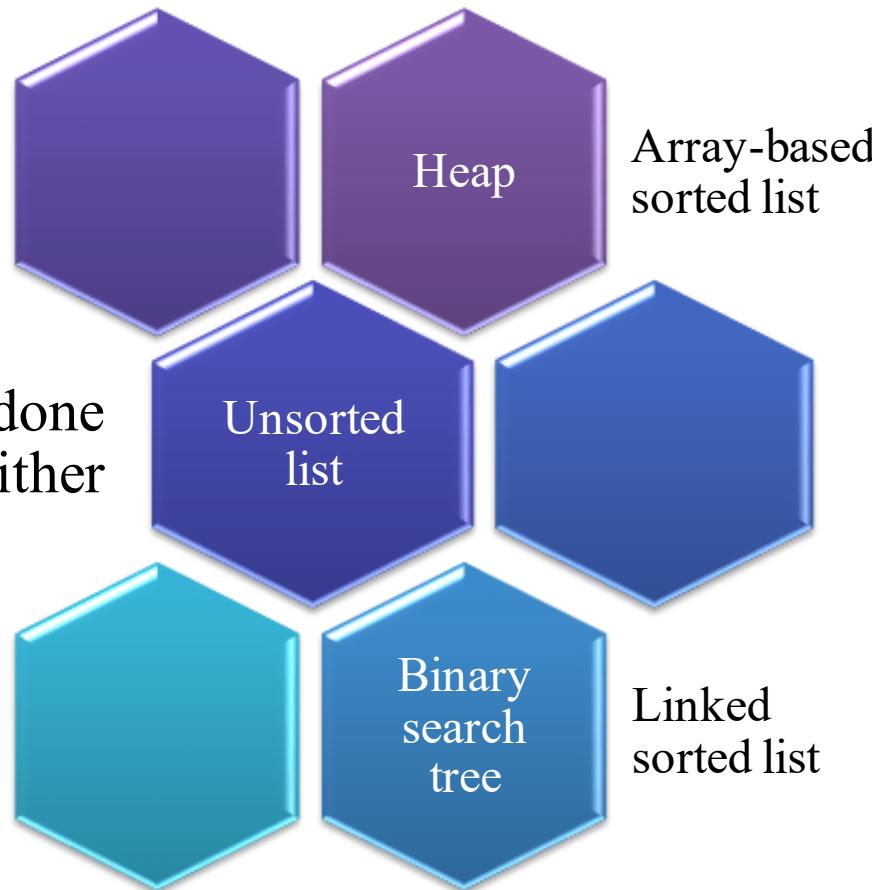
Priority Queue ADT Operations



Priority Queue Implementation

There are many ways to implement a priority queue.

It can be done
using either



Implement Priority Queue using Unsorted List

- **Enqueuing** an item would be very easy with an unsorted list—simply insert it at the end of the list.
- **Dequeuing** would require searching through the entire list to find the item with the highest priority.

Implement Priority Queue using Array-Based Sorted List

Dequeuing is very easy with array-based sorted lists—return the item with the highest priority, which would be in the $\text{length} - 1$ position. Thus, dequeuing is an $O(1)$ operation.

Enqueuing, is more expensive. We have to find the place to enqueue the item— $O(\log n)$ if we use a binary search—and rearrange the elements of the list after inserting the new item— $O(n)$.

Implement Priority Queue using Linked Sorted List

- Assume that the linked list is kept sorted from largest to smallest.
- **Dequeueing** simply requires removing and returning the first list element, an operation that requires only a few steps.
- **Enqueueing** is $O(n)$ because we must search the list one element at a time to find the insertion location.

Implement Priority Queue using Binary Search Tree

- **Enqueue** operation is like BST Insert operation, which requires $O(\log n)$ steps on average.
- **Dequeue** operation can be done $O(\log n)$ operation on average. The actual time depends on the number of levels in the tree, which we assume is approximately $\log n$. If BST is very narrow structure with many more levels, dequeue operation is $O(n)$.

Heap

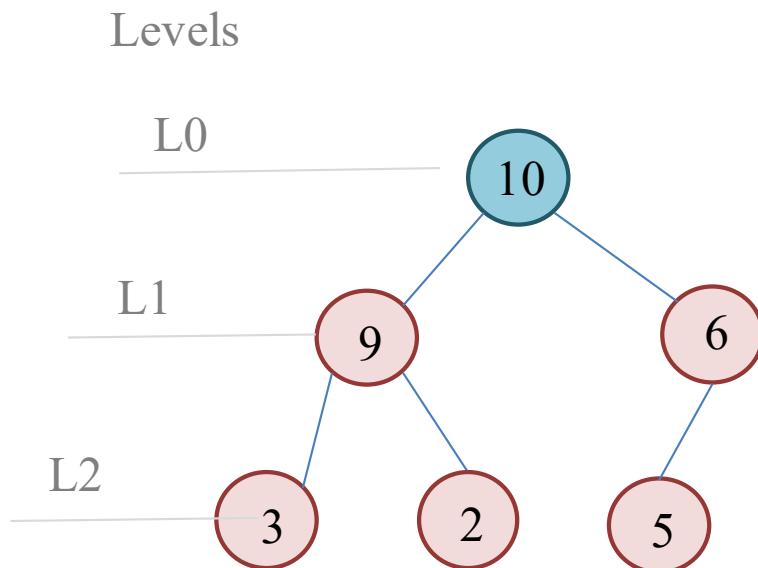
- A heap is a complete binary tree with values stored in its nodes such that no child has a value bigger than its that of its parents.
- The special feature of heaps is that we always know the location of the maximum value: It is in the root node.
- In heap definition, the root contains the item with the largest value. Such a heap is also known as a max-heap.
- A min-heap, on the other hand, places the item with the smallest value in its root.

Clarification Note

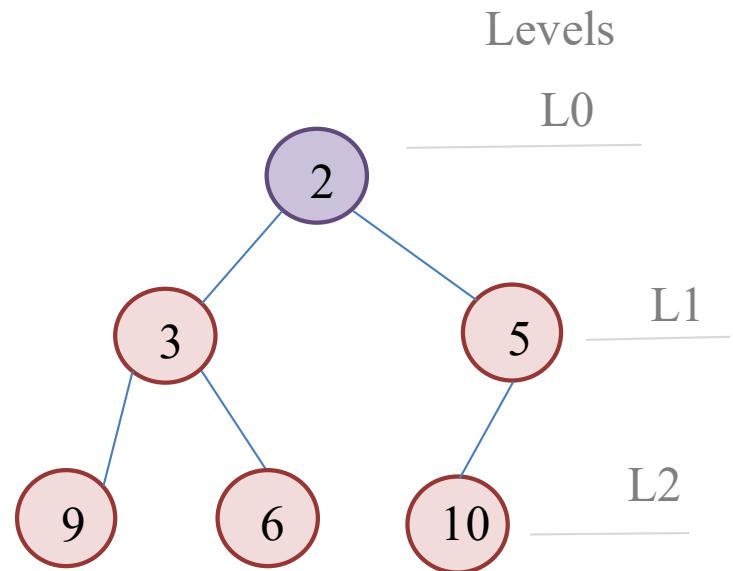
- Do not confuse the heap abstract data type with the collection of memory cells known as a heap.
- The heap memory is available for allocation to your program when you use the new operator.
- The heap that contains this available memory is not an instance of the ADT heap.

Example 2. max-heap and min-heap

The largest value (10)
in the **max-heap**



The smallest value (2)
in the **min-heap**



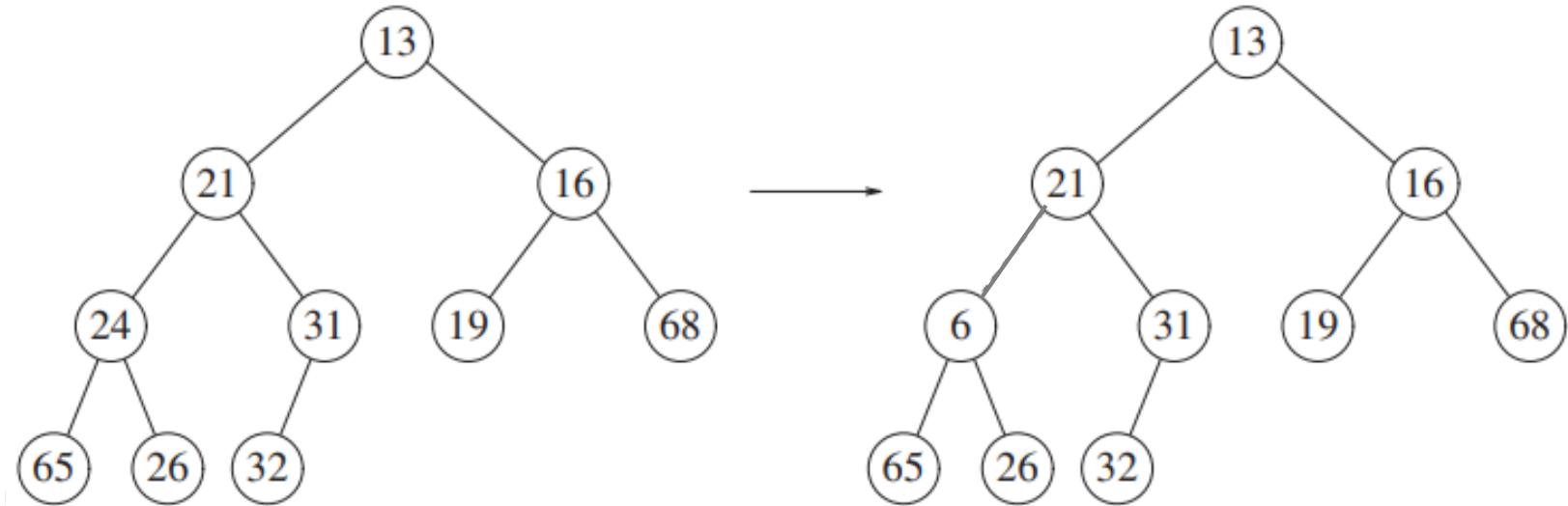
L0 values > L1 values > L2 values

L0 values < L1 values < L2 values

You Try 1. Heap

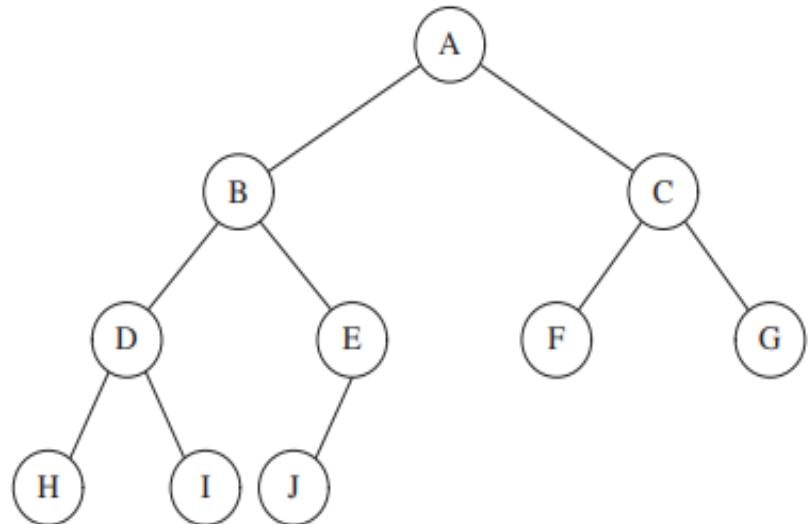


Which of the following trees is a heap?



Heap

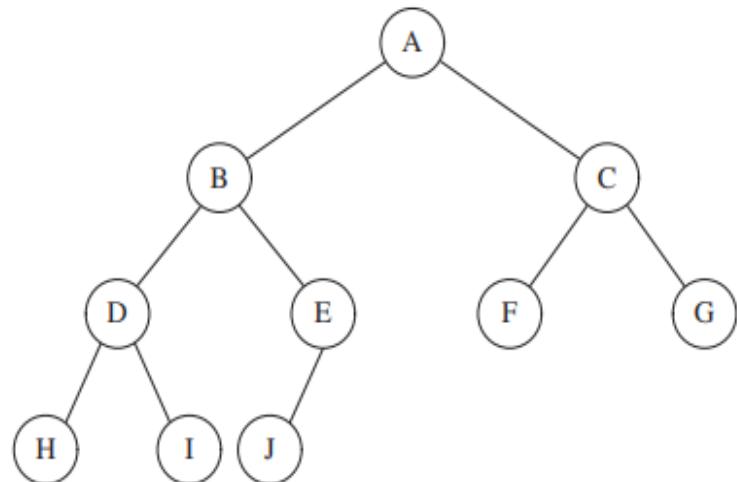
- A heap is a complete BT (filled from left to right).
- A complete BT of height h has between 2^h and $2^{h+1} - 1$ nodes.
- The height of a complete BT is $O(\log n)$.
- If the BT is complete and remains complete, you can use a memory-efficient array-based implementation



A complete BT

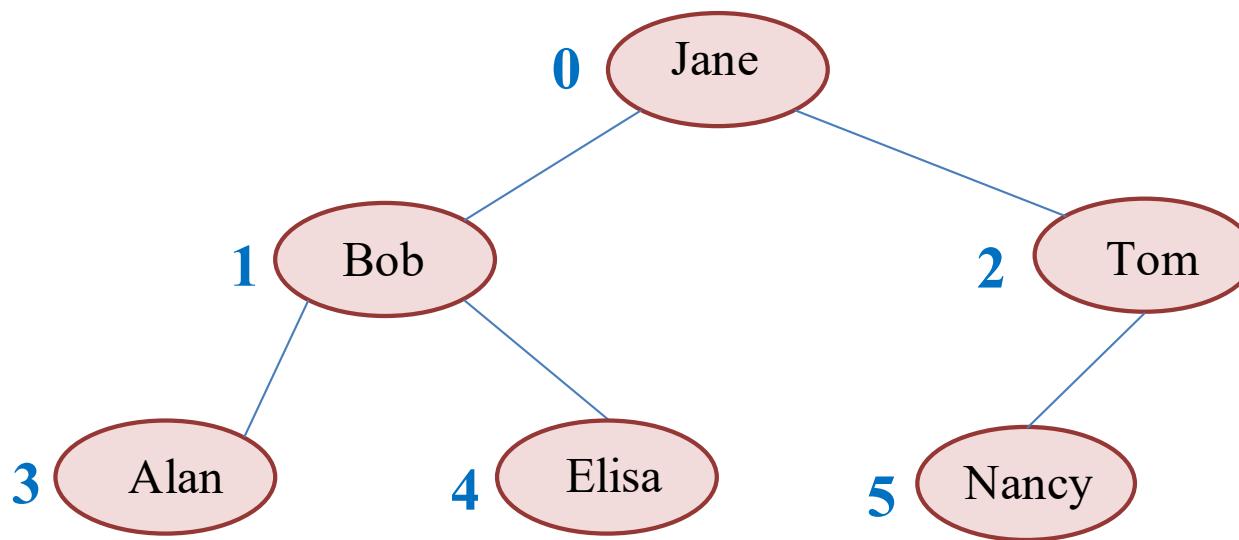
Heap Indexing

- The root is numbered 0, and the children of the root—which are at the next level of the tree—are numbered, left to right.
- You place these nodes into the array items in numeric order. That is, $\text{items}[i]$ contains the node numbered i .
- Remember that only the root in $\text{items}[0]$ does not have a parent.
- This array-based representation requires a complete binary tree.



Example 3. Complete BT Index and Array Representation

Show the indices of nodes of complete binary tree.



Jane	Bob	Tom	Alan	Elisa	Nancy		
0	1	2	3	4	5		

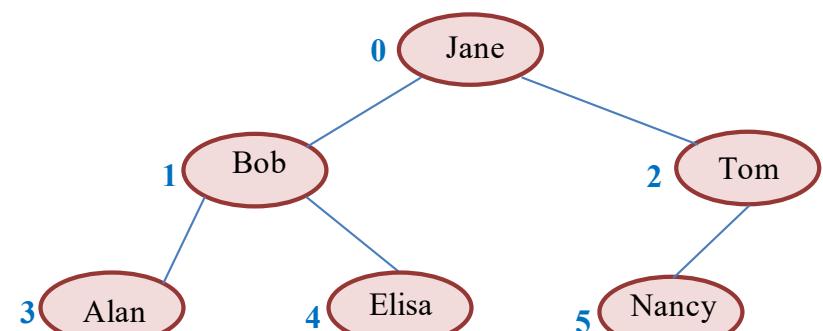
Example 3. Complete BT Index and Array Representation

Given any node $\text{items}[i]$, you can easily locate both of its children and its parent:

- Its left child, if it exists, is items $[2 * i + 1]$
- Its right child, if it exists, is items $[2 * i + 2]$
- Its parent, if it exists, is items $[(i - 1) / 2]$

For instance:

- **Left child** of node with index 2, has index $2*2+1=5$
- **Right child** of node with index 1, has index $2*1+2=4$
- **Parent** of node with index 3, is a node with index $(3-1)/2=1$

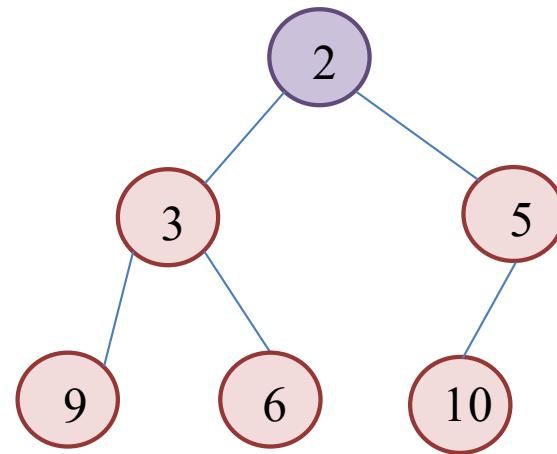
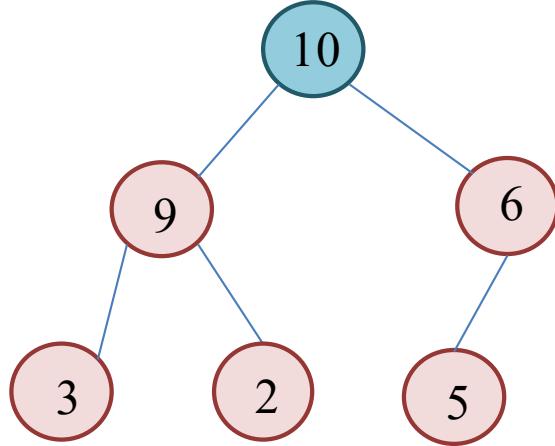


Jane	Bob	Tom	Alan	Elisa	Nancy	
0	1	2	3	4	5	

You Try 2. Array Representations of heaps



Show the arrays that represent the following max-heap and min-heap.



Heap Main Operations

Insert

- Insert a node into the heap

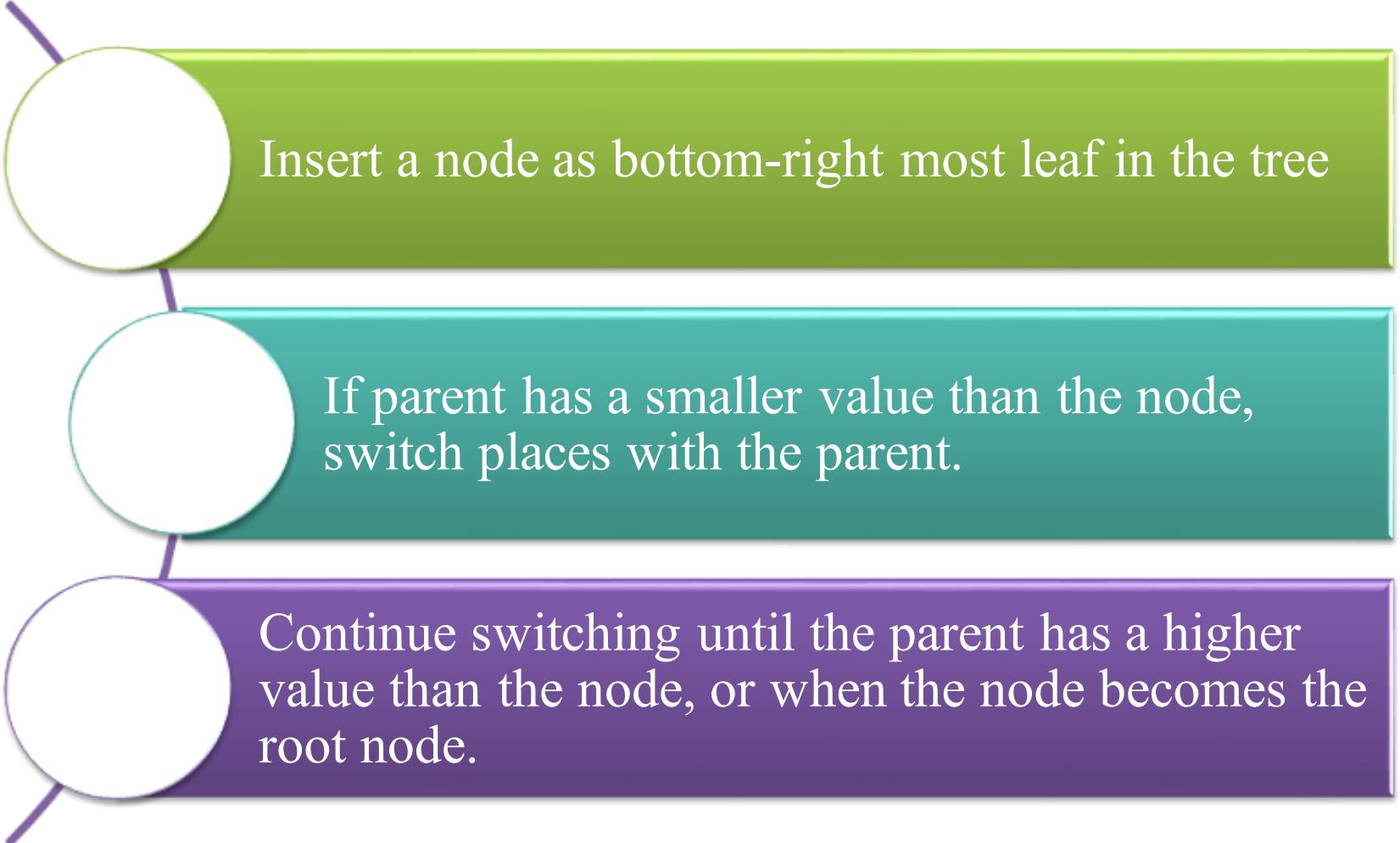
Remove

- Delete a node from the heap

Heapify

- Turn a binary tree into a heap

Heap Insert Operation

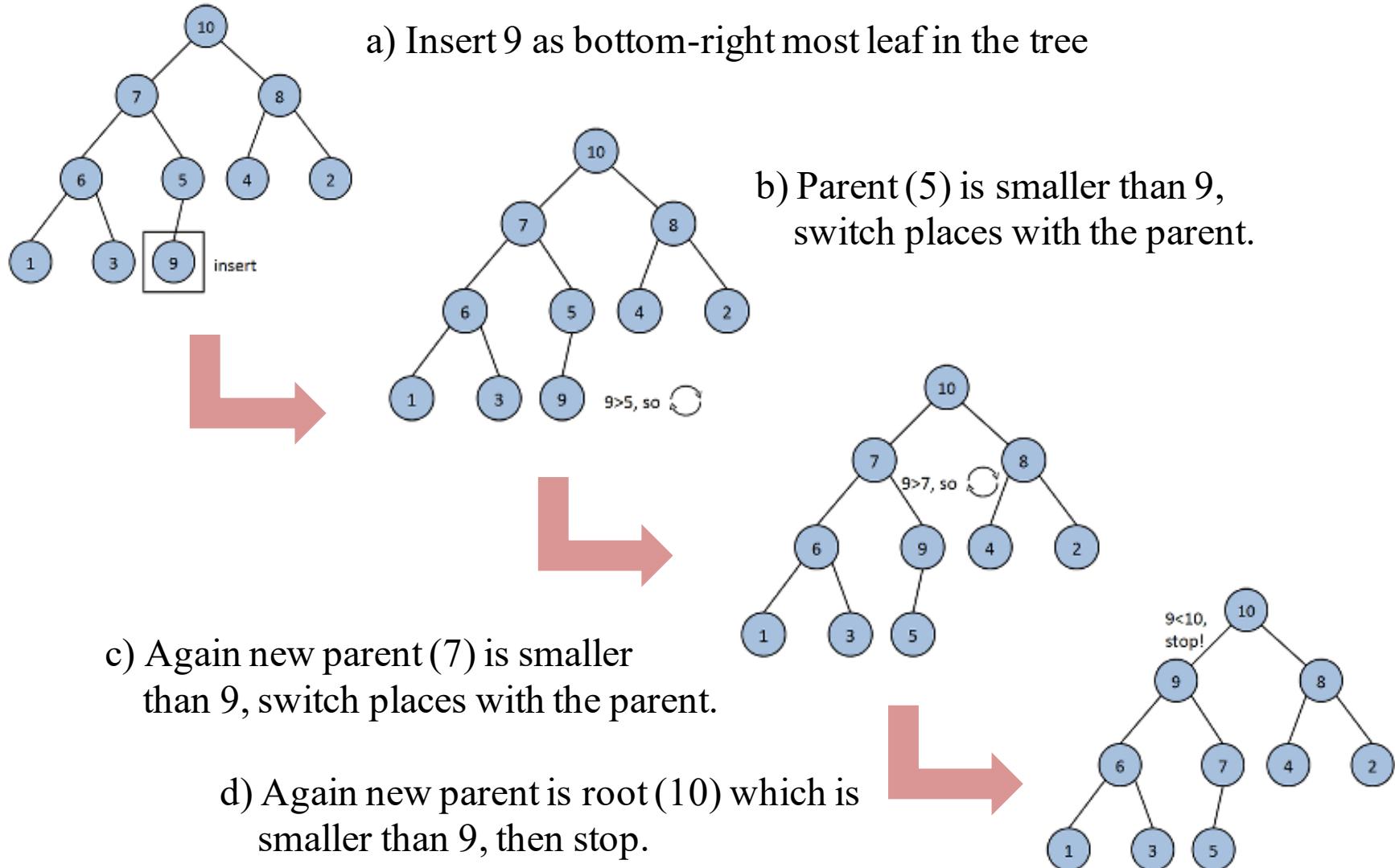


Insert a node as bottom-right most leaf in the tree

If parent has a smaller value than the node,
switch places with the parent.

Continue switching until the parent has a higher
value than the node, or when the node becomes the
root node.

Example 4. Heap Insert Operation



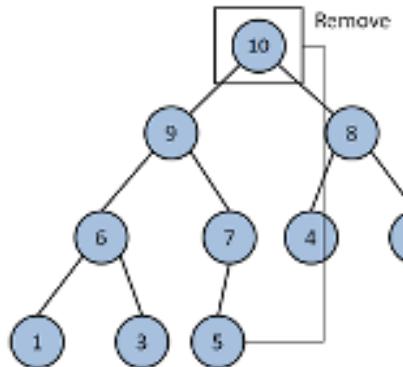
Heap Remove Operation

Replace root node with the bottom-right most leaf node in the tree.

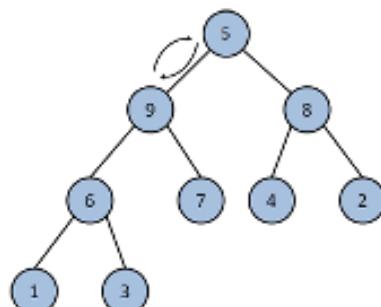
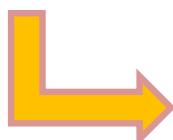
Switch node with the highest valued child.

Continue switching until the bottom of the tree is reached.

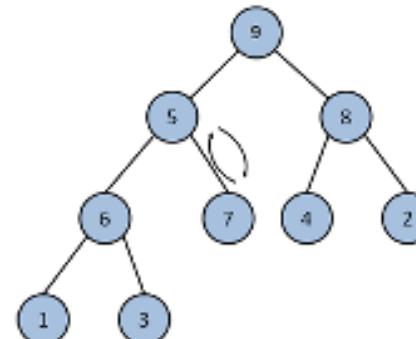
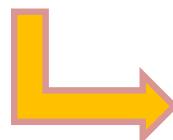
Example 5. Heap Remove Operation



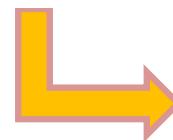
a) Replace root node with the bottom-right most leaf node in the tree.



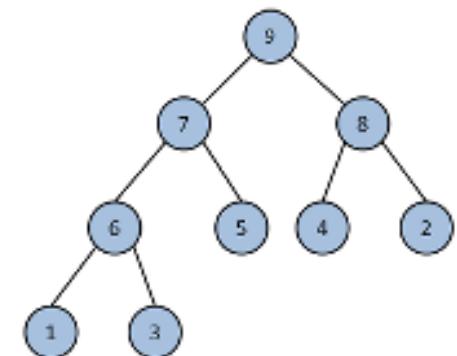
b) Switch node (5) with the highest valued child (9).



c) Continue switching (5 and 7)



d) Stoppe switching, the bottom of the tree is reached.



Heapify Operation

1. Enumerate the internal nodes of the tree in reverse level order.



2. In the order of enumerated list, percolate each internal node down using the top-down approach until it has no more children.

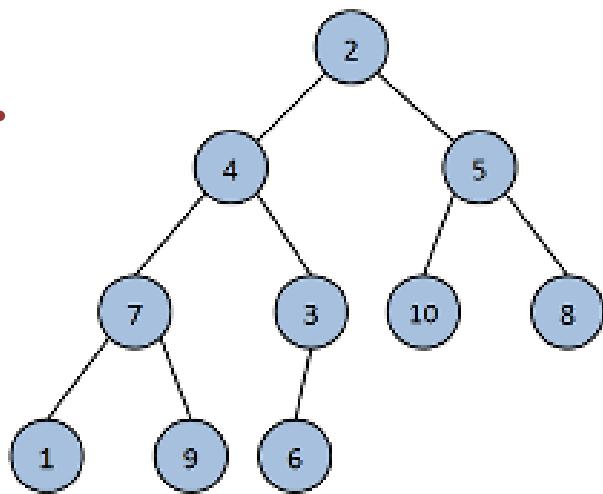


3. Percolate root node down using the top-down approach until it has no more children.

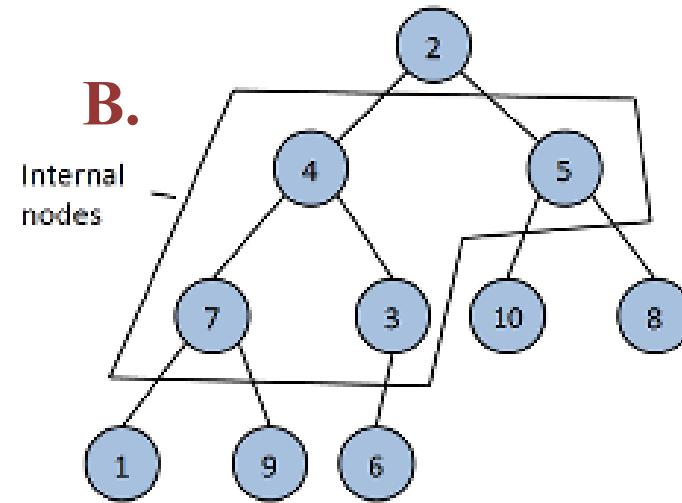


Example 6. Heapify Operation

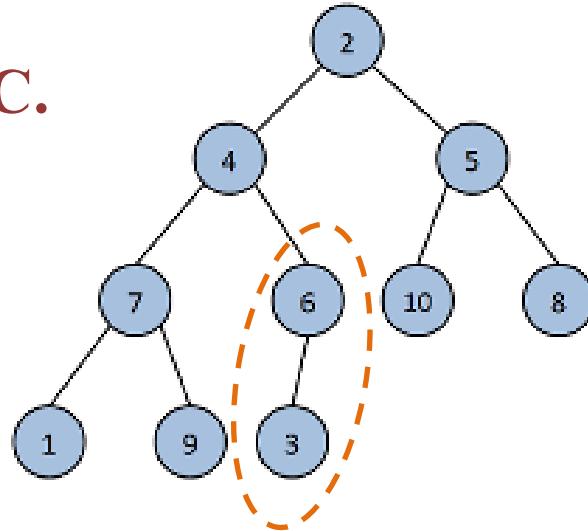
A.



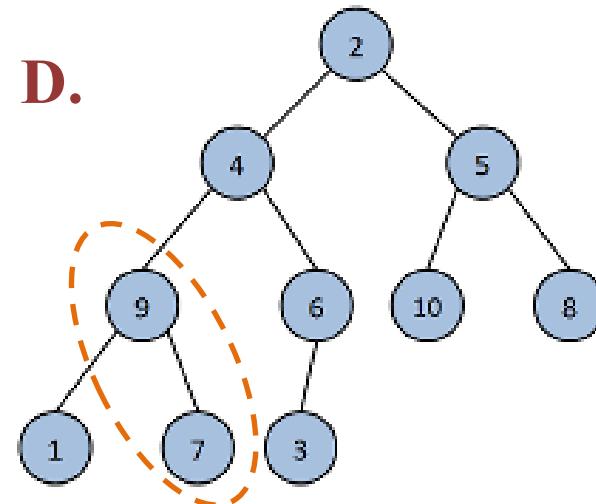
B.



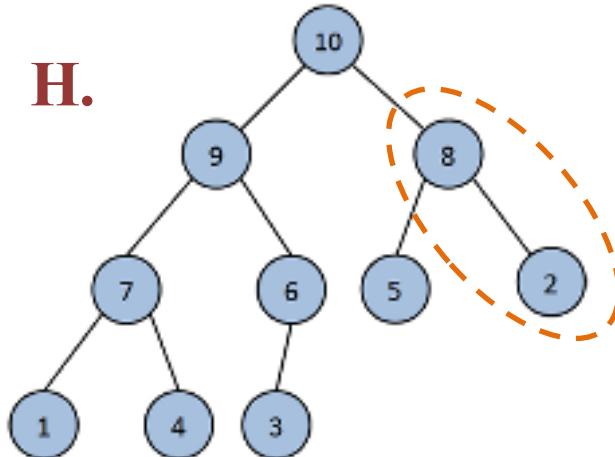
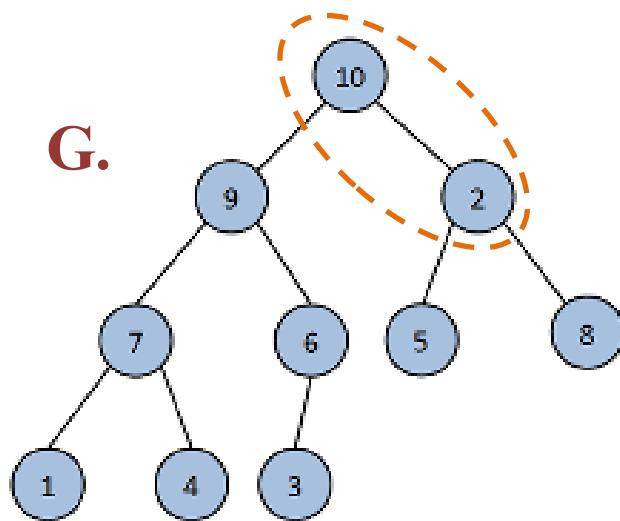
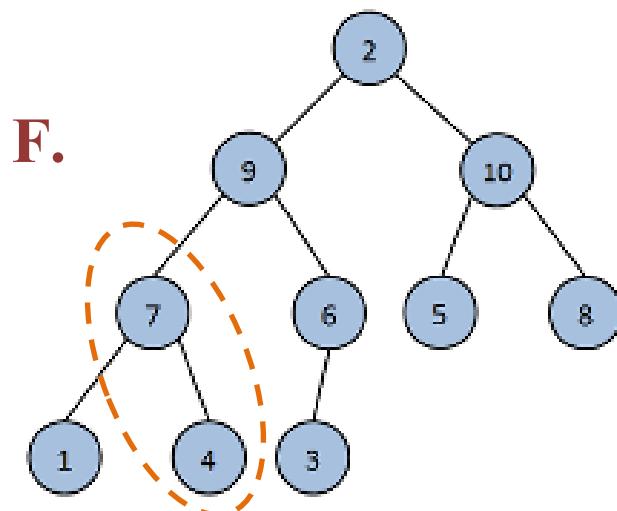
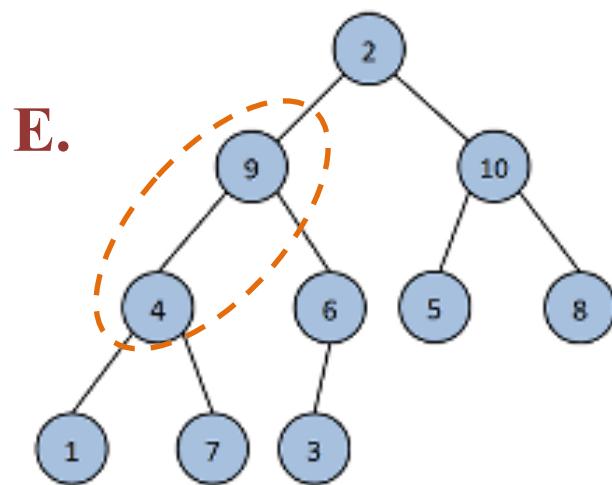
C.



D.



Example 3.



Next Lecture

We focus on:

- Heap Implementation
- Introduction to Graphs

Readings/ Study Reference

Course Textbook: “*Data Structure and Algorithms in a Nutshell*” A. Wong et al.

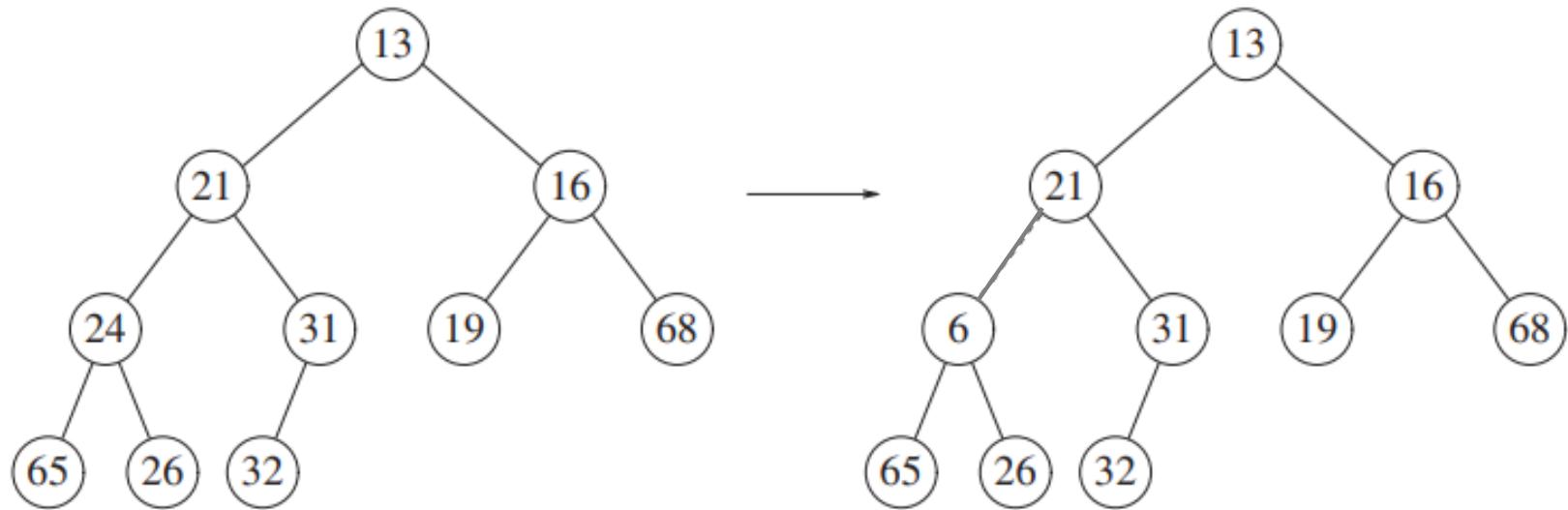
Chapter 7. Trees

Section 7.5. Heaps

The End of Lecture

You Try 1. Heap

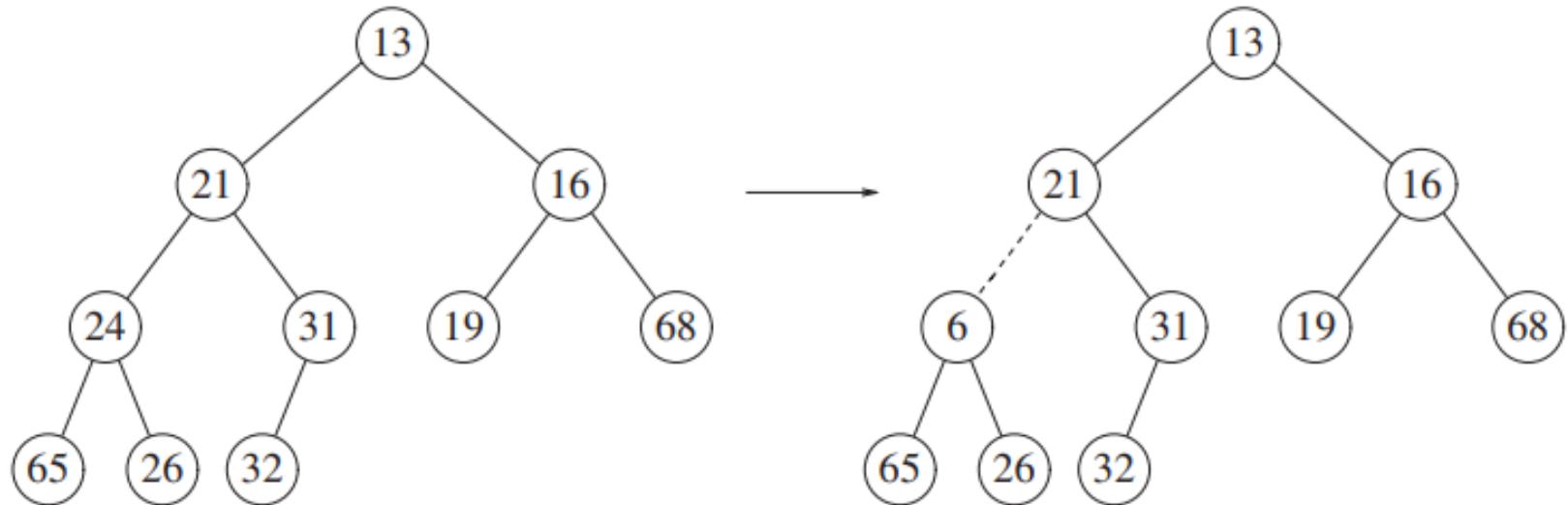
Which of the following trees is a heap?



You Try 1. Heap

Which of the following trees is a heap?

Is it max-heap or min-heap?



Two complete trees , only the left tree is a heap. It is a min-heap.
Because root node has the smallest value.

Example 1. Priority Queue-Tracking Assignment

- Professors and bosses like to assign tasks for us to assign tasks for us to do by certain dates.
- Using a priority queue, we can organize these assignments in the order in which we should complete them (e.g. by due dates).
- We can define a class **Assignment** of tasks that includes a data field date representing a task's due date.
- You can write a pseudocode to show how you could use a priority queue to organize your assignments and other responsibilities so that you know which one to complete first

Example 1. Priority Queue-Tracking Assignment

`assignmentLog = a new priority queue using due date as the priority value`

`project = a new instance of Assignment`

`essay = a new instance of Assignment`

`task = a new instance of Assignment`

`errand = a new instance of Assignment`

`assignmentLog.add(project)`

`assignmentLog.add(essay)`

`assignmentLog.add(task)`

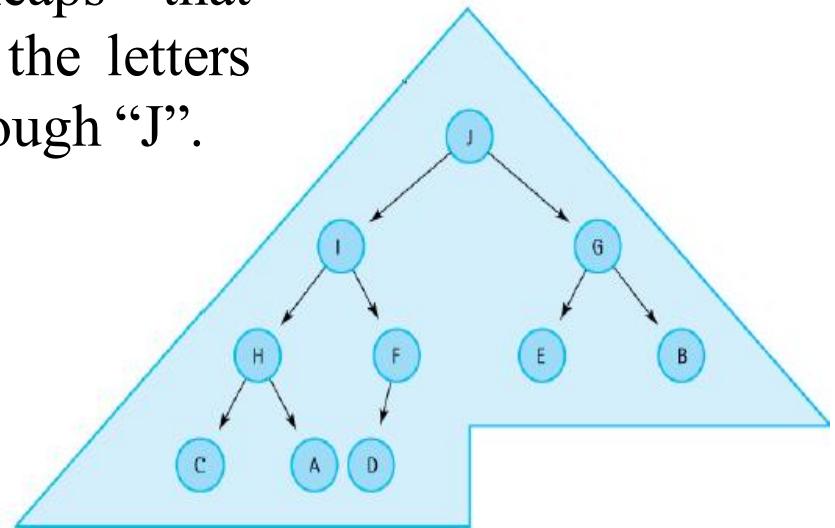
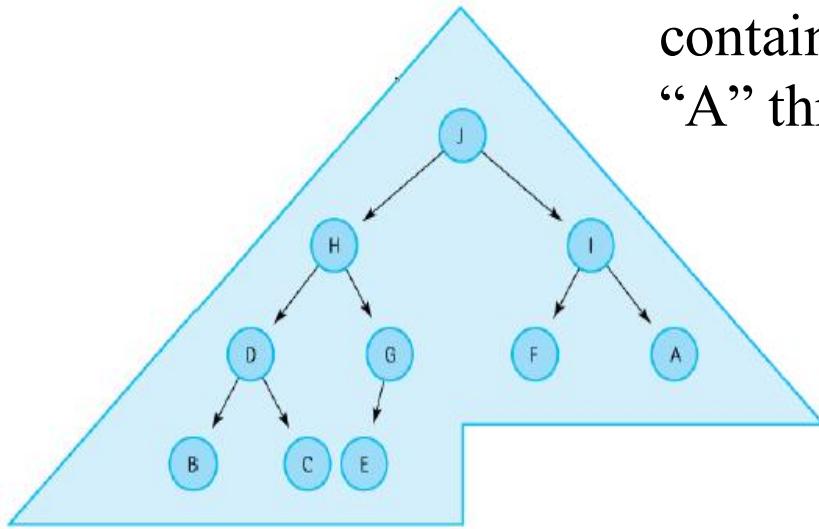
`assignmentLog.add(errand)`

`cout << "I should do the following first: "`

`cout << assignmentLog.peek()`

Heap

Two heaps that contain the letters “A” through “J”.



- Note: Different placement of the values, but similar shapes.
- Note: Both heaps have the same root node. A group of values can be stored in a binary tree in many ways and still satisfy the order property of heaps.

Course: Data Structures and Algorithms

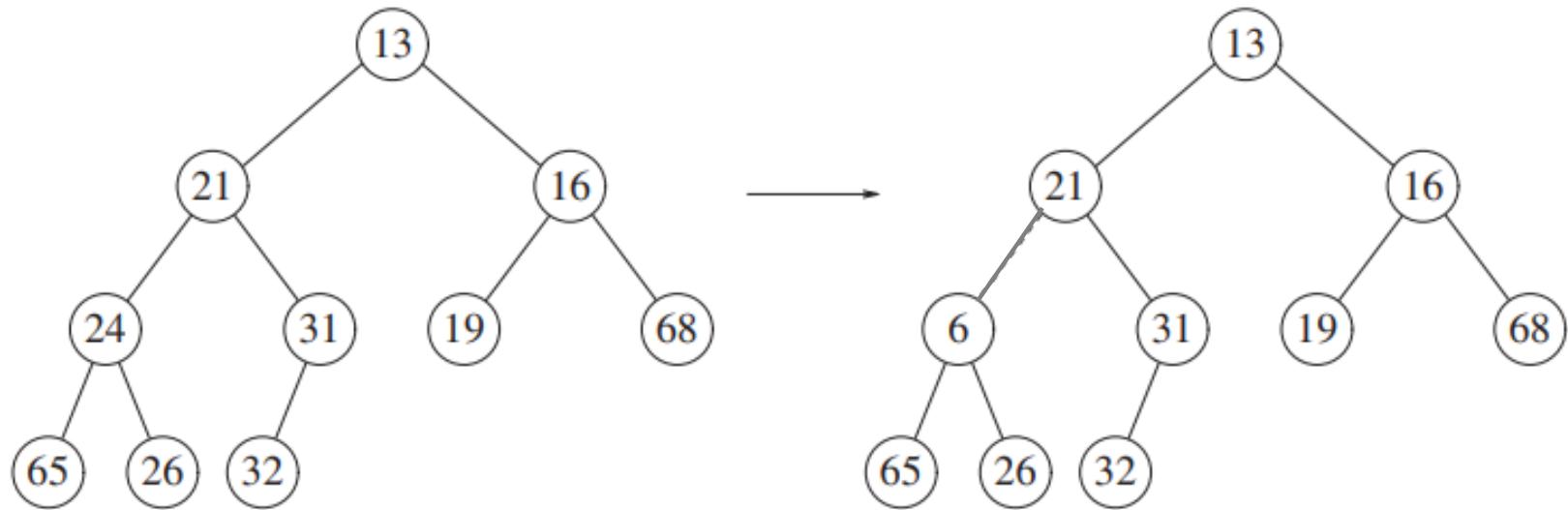
Instructor: Homeyra Pourmohammadali

You Try Questions and Solutions
Heaps and Priority Queues

You Try 1. Heap



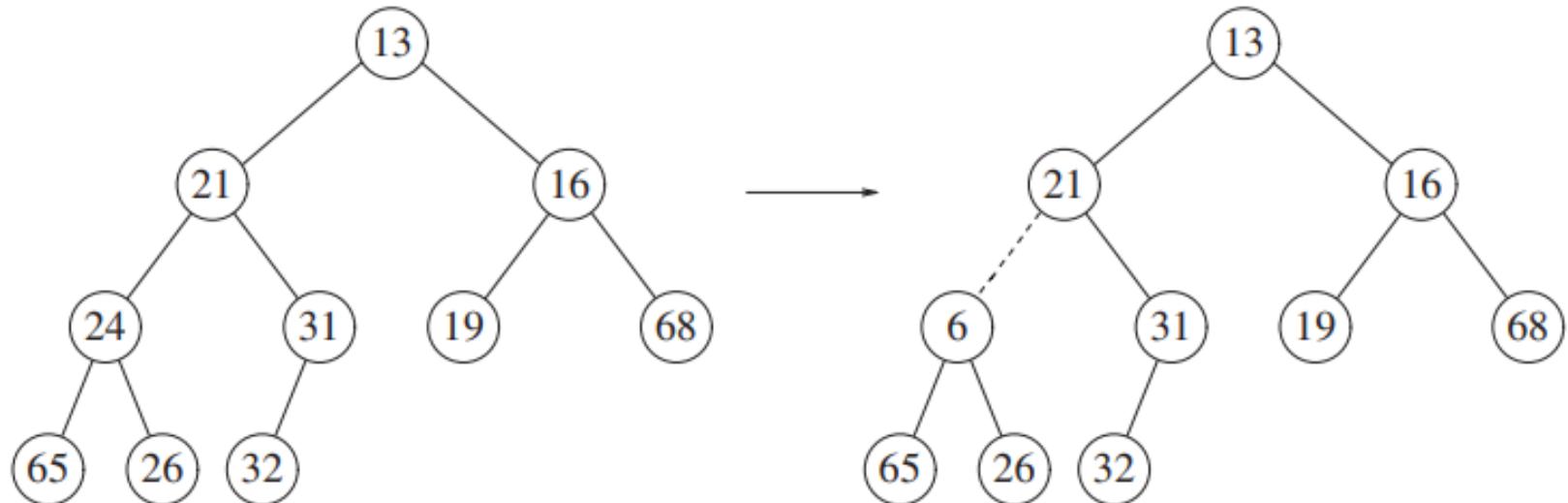
Which of the following trees is a heap?



You Try 1 Solution. Heap

Which of the following trees is a heap?

Is it max-heap or min-heap?

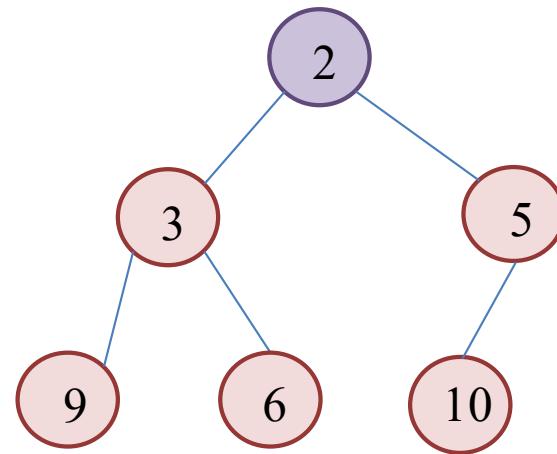
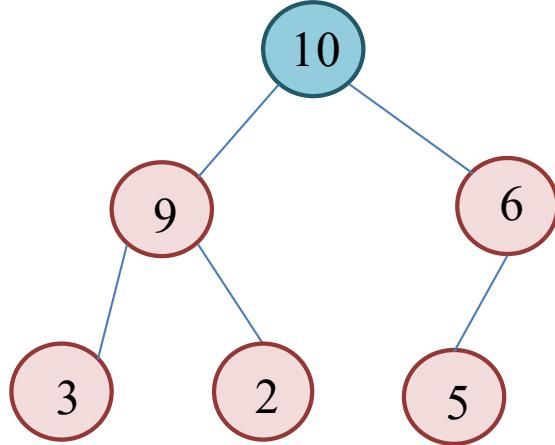


Two complete trees , only the left tree is a heap. It is a min-heap.
Because root node has the smallest value.

You Try 2. Array Representations of Heaps

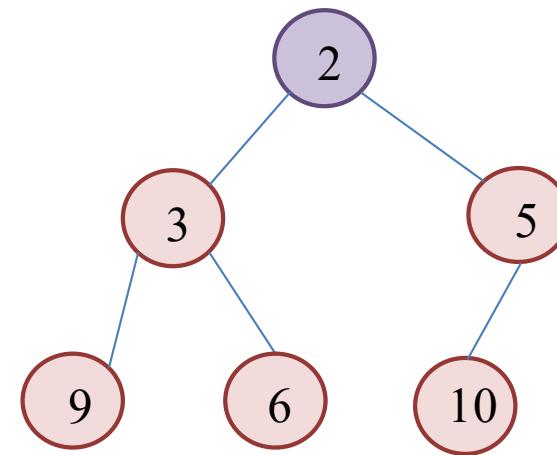
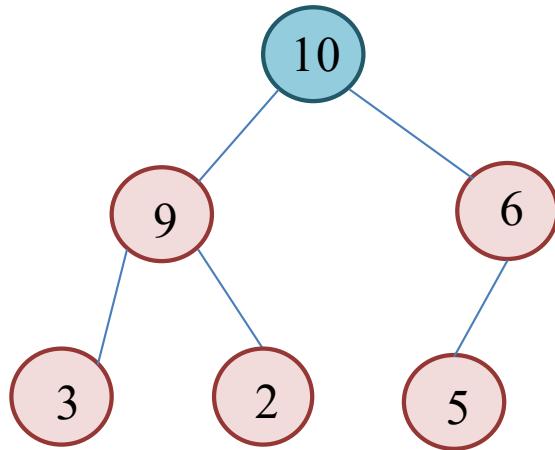


Show the arrays that represent the following max-heap and min-heap.



You Try 2 Solution. Array Representations of Heaps

Show the arrays that represent the following max-heap and min-heap.



10	9	6	3	2	5		
0	1	2	3	4	5		

2	3	5	9	6	10		
0	1	2	3	4	5		

The End of Lecture

Course: Data Structures and Algorithms

Instructor: Homeyra Pourmohammadali

Graph Structure, ADT and Search Traversals

Motivation

Which of the data structure you have learned so far would you use to represent the connection of people in a social network?



<https://www.dreamstime.com/illustration/searching-friends-social-network.html>

Motivation

How are the flight connections between airports around the world modelled? How are the optimal flight routes planned for travelling between different destinations?



<https://geographica.com/en/blog/flight-routes/>

Motivation

Graphs are used in diverse fields:



GPS Systems



Google Maps



Google Search



Social Networks



Operations Research



Learning Outcomes

By the end of this lecture you will be able to:

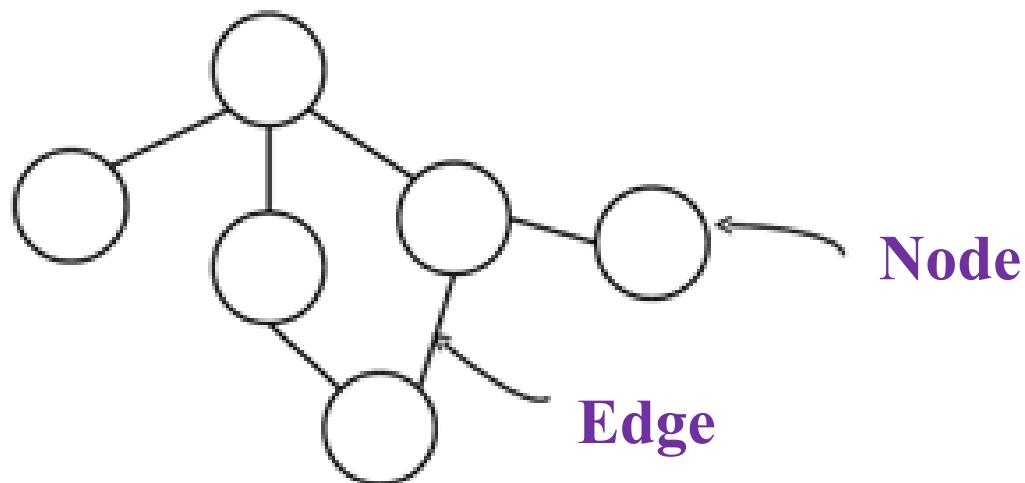
- define common terminologies related to graphs
- relate the concepts or application of each to a real-life example.
- list the main operations of graph ADT
- describe the graph search traversal strategies: depth-first and breadth-first strategies.

Graph Terminologies

Graph: A data structure that consists of a set of nodes and a set of edges that relate the nodes to one another.

Edge: A pair of vertices representing a connection between two nodes in a graph

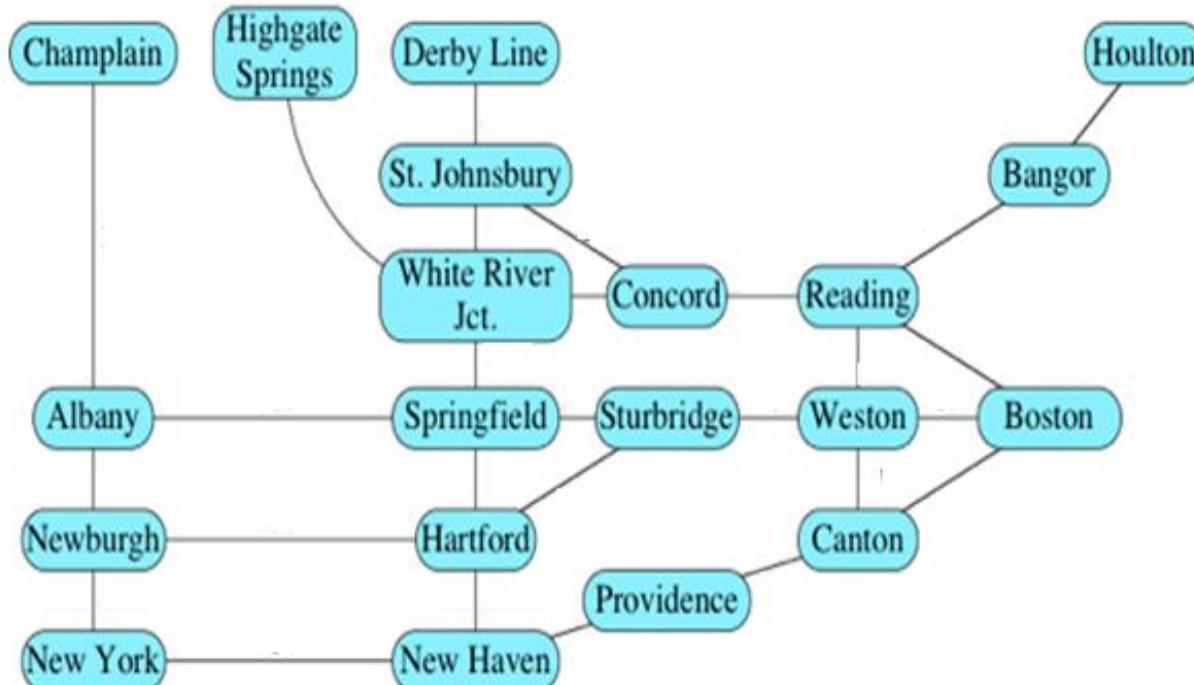
Vertex: A node in a graph



Example 1. Graphs in Real-life

What is the shortest route from New York to Boston?

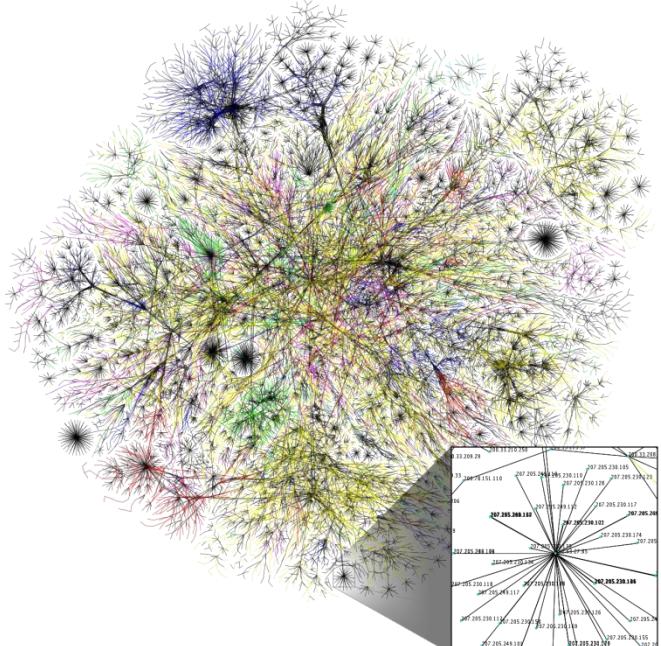
- **Node:** city
- **Edge:** path between cities



Example 2. Graphs in Real-life

Internet

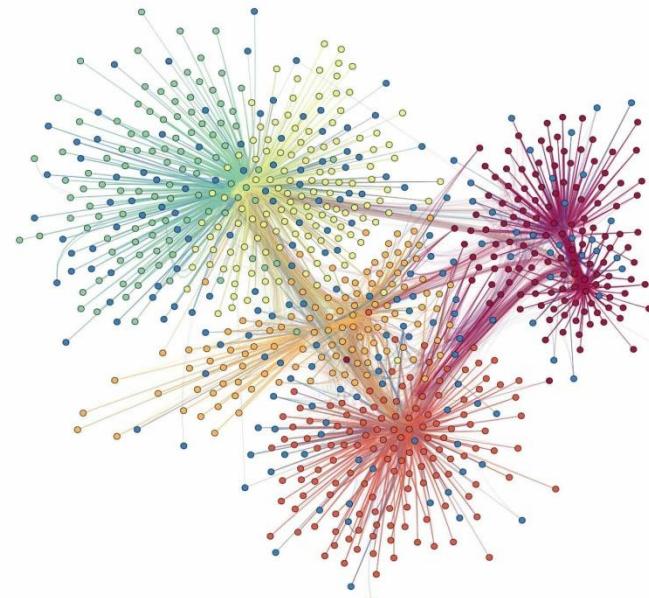
- **Node:** an IP address
- **Edge:** time delay between two IPs



<https://en.wikipedia.org/wiki/Internet>

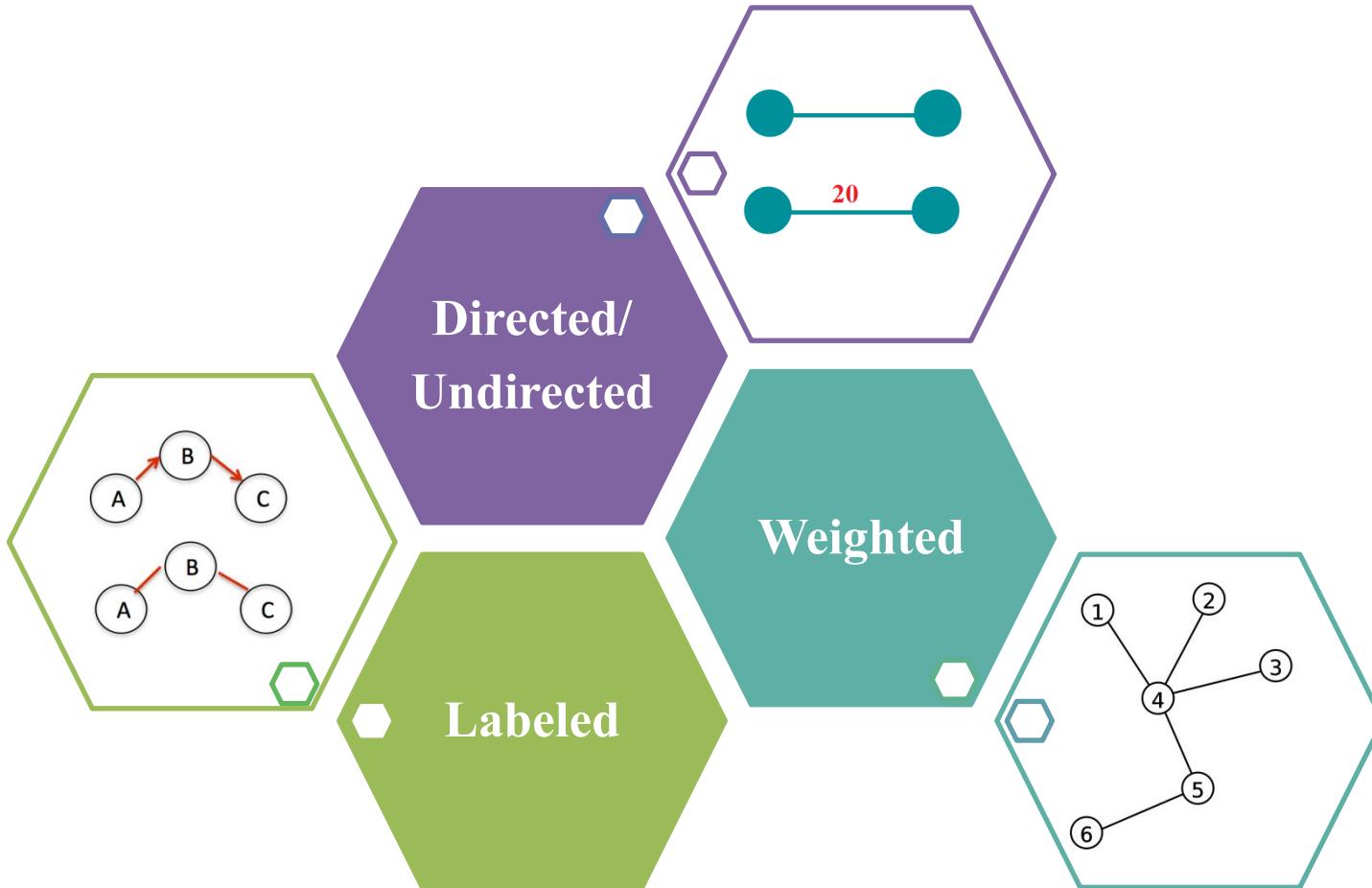
Social network

- **Node:** a person
- **Edge:** relationship, e.g., friendship



<https://towardsdatascience.com/influential-communities-in-social-network-simplified-fe5050dbe5a4>

Types of Graphs



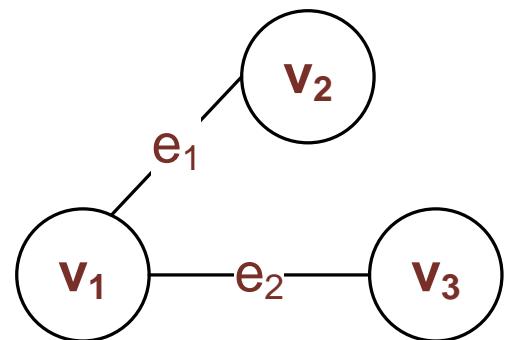
Formal Definition of a Graph

- Formally, a graph G is defined as: $G = (V, E)$ where V is a finite, nonempty set of vertices, and E is a set of edges. An edge $e(v_i, v_j)$ represents an edge going from v_i to v_j , where $v \in V$ and $e \in E$.
- To specify the set of vertices, list them in set notation, within $\{ \}$ braces.

For example:

$$V = \{v_1, v_2, v_3\}$$

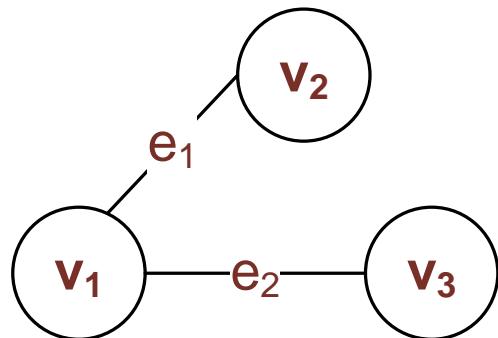
$$E = \{e_1, e_2\} \text{ or } E = \{(v_1, v_2), (v_1, v_3)\}$$



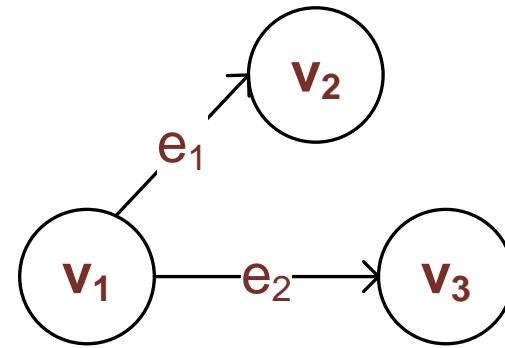
Directed and Undirected Graphs

Undirected Graph: A graph in which the edges have no direction. Then: $e(v_i, v_j) = e(v_j, v_i)$

Directed Graph: A graph in which each edge is directed from one vertex to another (or the same) vertex. Then:
 $e(v_i, v_j) \neq e(v_j, v_i)$



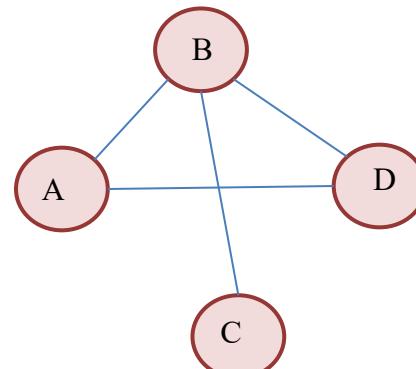
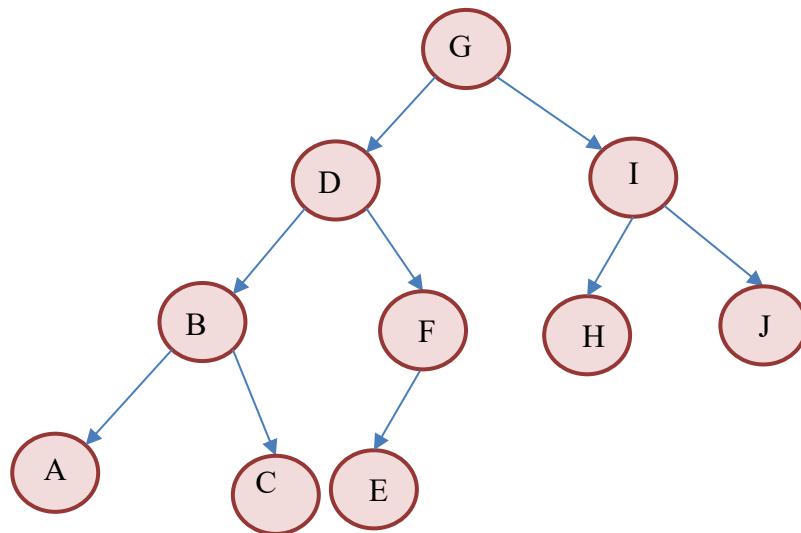
Undirected Graph



Directed Graph

Example 3. Representation of a Graph Nodes and Edges

Write the sets of nodes, V , and sets of edges, E , for these graphs:



$$V = \{A, B, C, D\}$$

$$E = \{ (A, B), (A, D), (B, C), (B, D) \}$$

$$V = \{A, B, C, D, E, F, G, H, I, J\}$$

$$E = \{ (G, D), (G, I), (D, B), (D, F), (I, H), (I, J), (B, A), (B, C), (F, E) \}$$

Example 4. Directed/Undirected Graph

- Google+: One-way following (directed edge)
- Facebook: Two-way reciprocal (undirected edge)

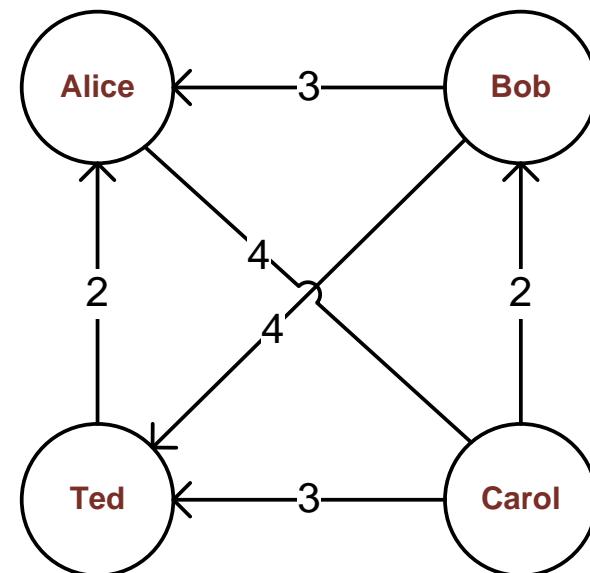


- One-way road (directed edge)
- Two-way road (undirected edge)



Weighted Graphs

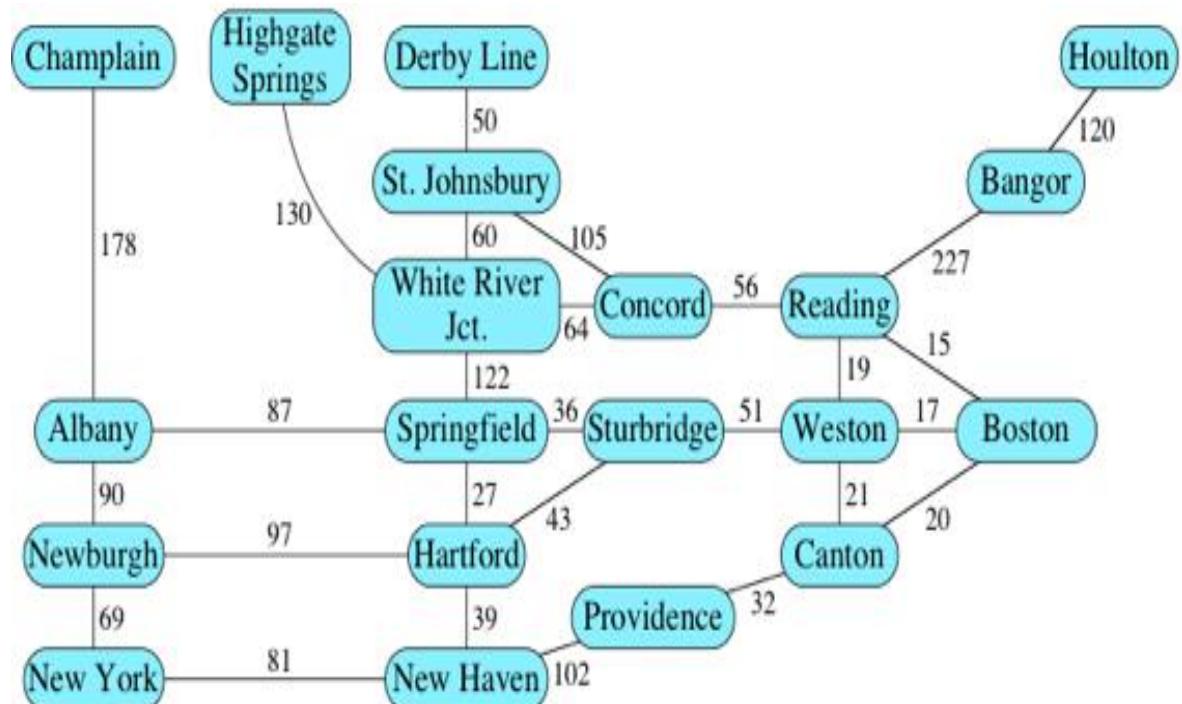
- It is a graph where each edge has a numerical value associated with it.
- The weight of edge $e(i, j)$, going from node i to j is $w(i, j)$.
- Used in modelling situations where there is associated cost or weight with each connection and the cost needs to be optimized.



Example 5. Weighted Graph

To find the **shortest route** between two locations, you can look for a path between two vertices with the **minimum sum of edge weights** over all paths between the two vertices.

The shortest path from New York to Concord goes from New York to New Haven to Hartford to Sturbridge to Weston to Reading to Concord, totaling **289** miles.

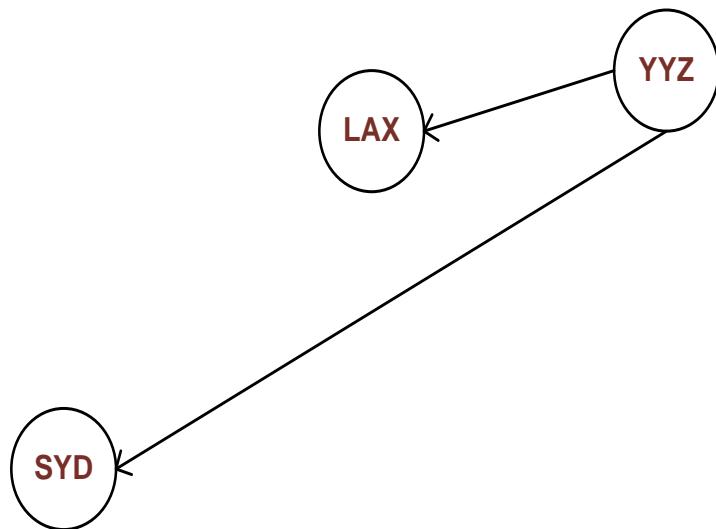


Labeled Graph

Labeled graph is a graph where each node has a unique symbolic label associated with it.

Example 6. Labeled Graph

- In modelling of flight connections between the airports around the world, each node is labeled with its associate airport name. (LAX, YYZ, etc.)

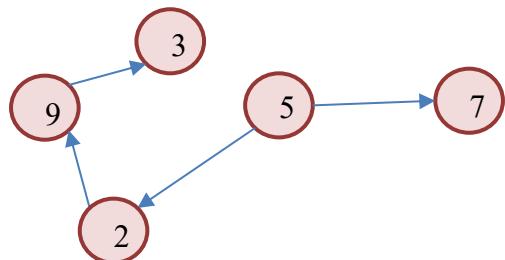
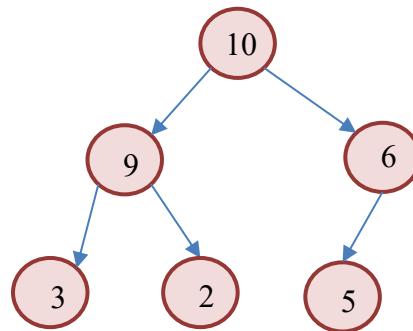


Path

Path is sequence of vertices that connects two nodes in a graph. For a path to exist, an uninterrupted sequence of edges must go from the first vertex, through any number of vertices, to the second vertex.

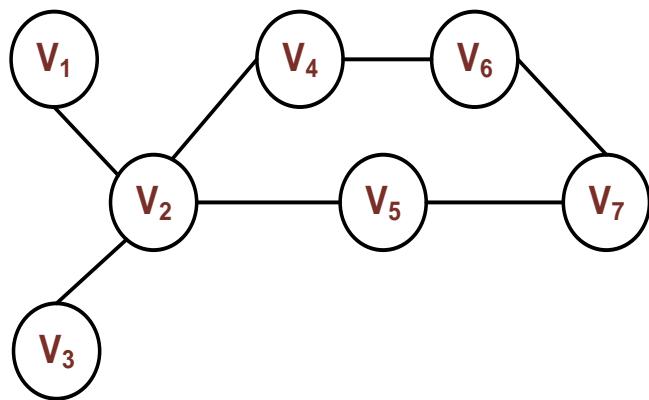
Example 7. Path of a Graph

- A unique path exists from the root to every other node in the tree.
- A path goes from vertex 5 to vertex 3, but not from 3 to 5.

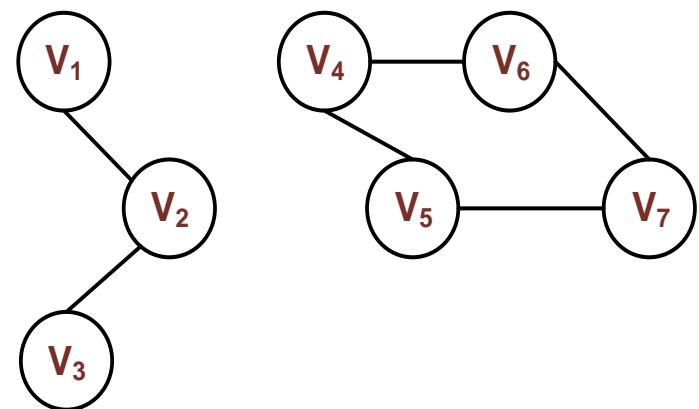


Graph Connectivity

- A **path** is a sequence of edges that connects a sequence of nodes



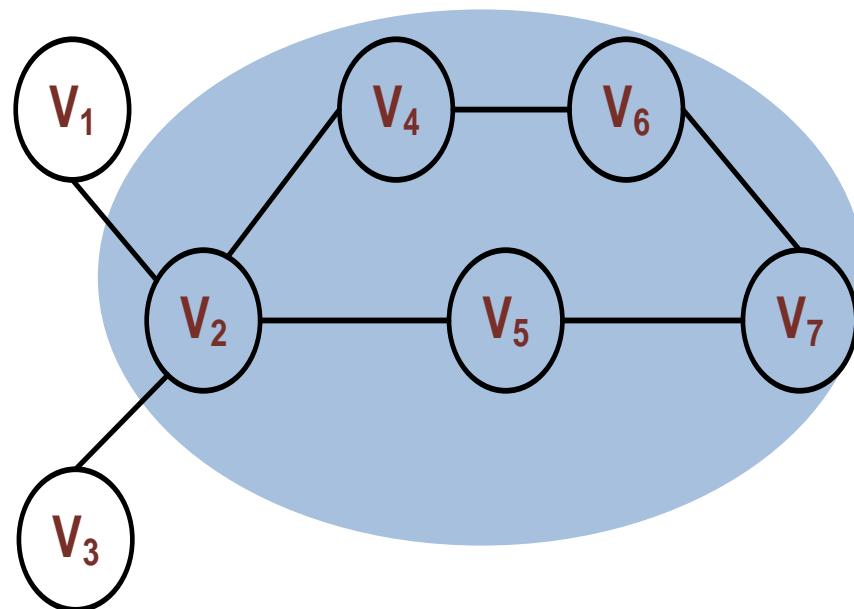
Connected Graph



Not Connected Graph

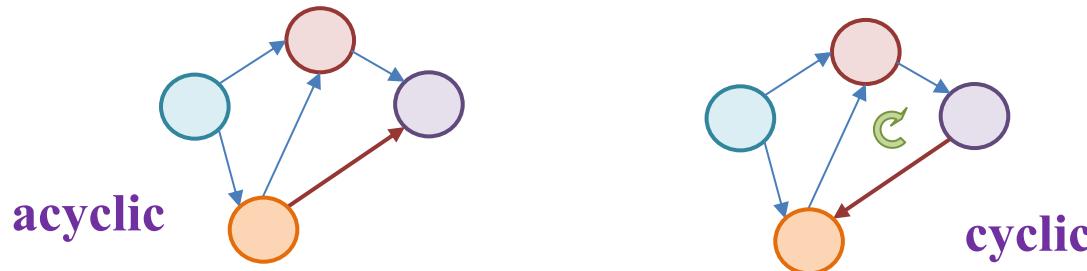
Graph Cycle

- A cycle is a set of edges that form a path such that the first node of the path corresponds to the last.



Directed Acyclic Graph

A directed acyclic graph (DAG) is a directed graph that has no cycles.

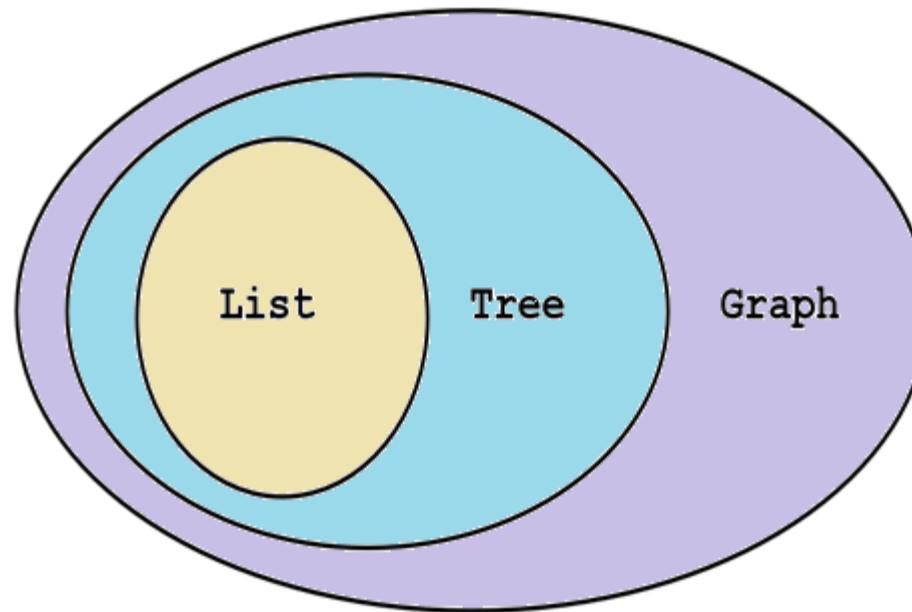


Example 8. Directed Acyclic Graph

In representation of a course calendar, one course can be prerequisite of another course but not the other way around. Then, taking one course let you register in a subsequent related course, but you are not allowed to take the courses in the reverse sequence.

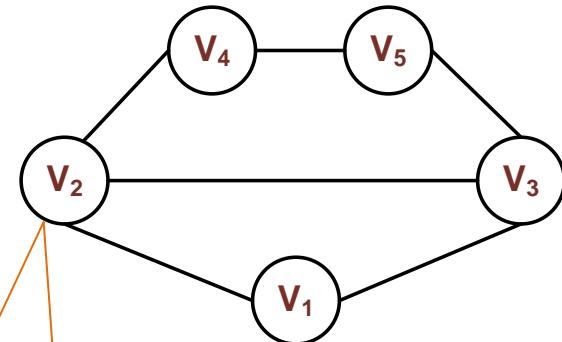
Relationship Between Graph, Tree & List

- A tree is a special case of graph that is connected and has no cycle.
- A list is a special case of tree in which each node has at most one child.



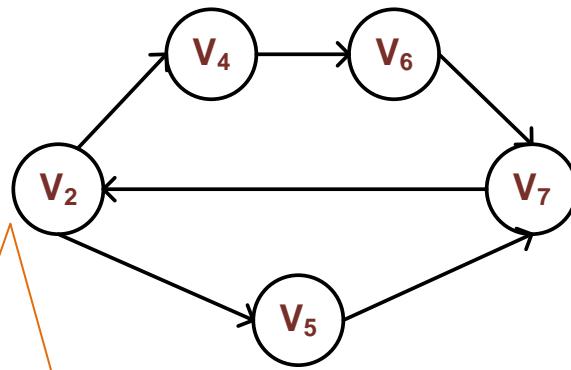
Degree of a Node

- For **undirected** graphs: **degree** of a node is the number of distinct edges where the node is an end point.
- For **directed** graphs: **in-degree** of a node is the number of distinct edges where the node is the end point.
- For **directed** graphs: **out-degree** of a node is the number of distinct edges where the node is the starting point.



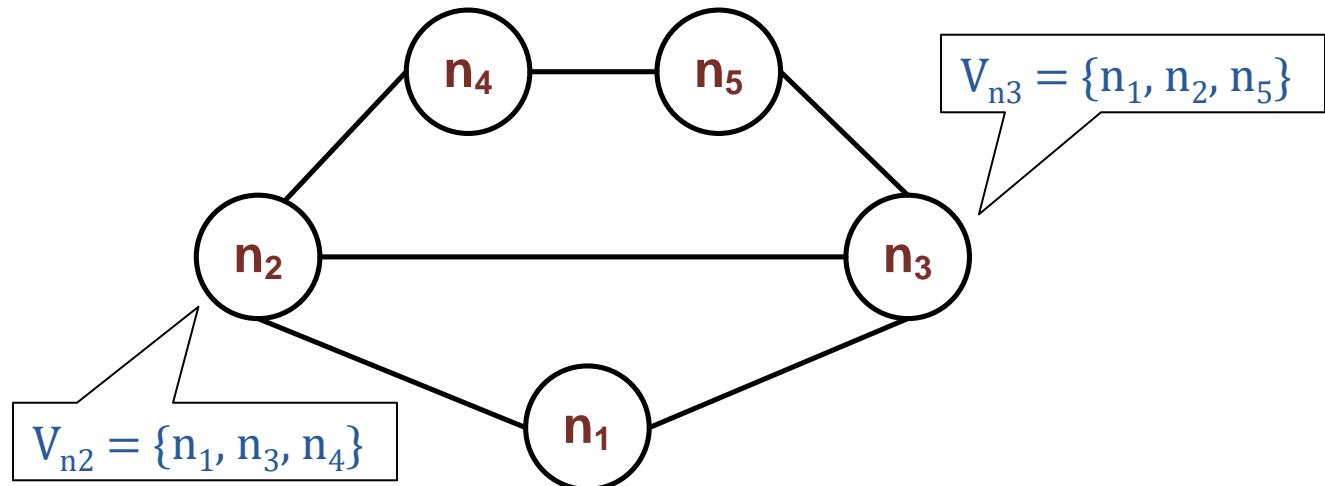
degree(v₂) = 3

in-degree(v₂) = 1
out-degree(v₂) = 2



Adjacency of Nodes

- Two nodes are adjacent if there exists an edge that connects them.
- The adjacency set of a node x , which is denoted as V_x , is the set of all nodes that are adjacent to x .



Graph ADT

- Insertion and removal operations are somewhat different for graphs than for other ADTs that you have studied, in that they apply to either vertices or edges.
- You can define the ADT graph so that its vertices either do or do not contain values.
- A graph whose vertices do not contain values represents only the relationships among vertices. Such graphs are not unusual, because many problems have no need for vertex values.

Graph ADT Operations

- Test whether a graph is empty.
- Get the number of vertices in a graph.
- Get the number of edges in a graph.
- See whether an edge exists between two given vertices.
- Insert a vertex in a graph whose vertices have distinct values that differ from the new vertex's value.
- Insert an edge between two given vertices in a graph.
- Remove a particular vertex from a graph and any edges between the vertex and other vertices.
- Remove the edge between two given vertices in a graph.
- Retrieve from a graph the vertex that contains a given value.

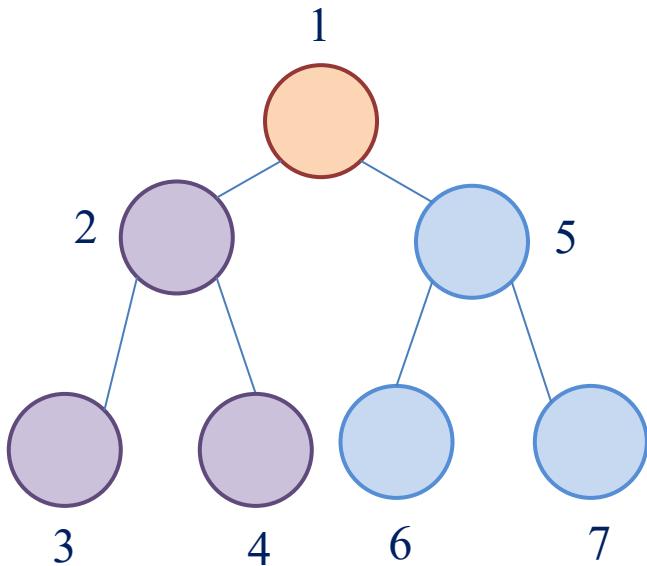
Graph Traversal

- This graph ADT specification includes only the most basic operations. It provides no traversal operations.
- Graph traversal can be done in many different orders. You can consider traversal as a graph application rather than an operation.
- The basic operations given in our specification allow us to implement different traversals independently of how the graph itself is implemented.

Depth-First and Breadth-First Traversal

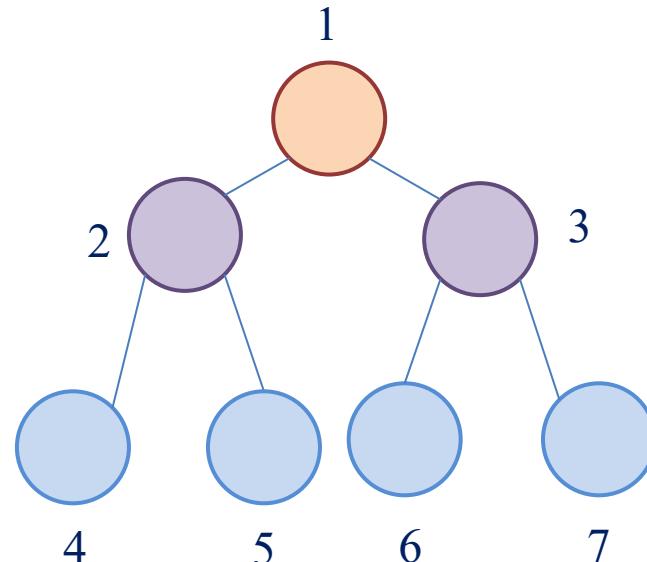
- Previously, we discussed the post-order tree traversal, which goes to the deepest level of the tree and works up.
- This strategy of going down a branch to its deepest point and then moving up is called a **depth-first** strategy.
- Another systematic way to visit each vertex in a tree is to visit each vertex on Level 0 (the root), then each vertex on Level 1, then each vertex on Level 2, and so on.
- Visiting each vertex by level in this way is called a **breadth-first** strategy. With graphs, both depth-first and breadth-first strategies are useful.

Depth-First and Breadth-First Tree Traversal



Depth-First Traversal

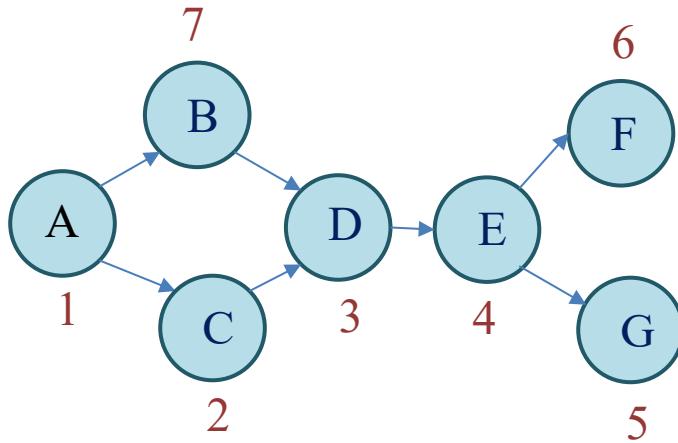
Goes down a branch to its deepest point and then moves up. Checks left subtree first then the right one.



Breadth-First Traversal

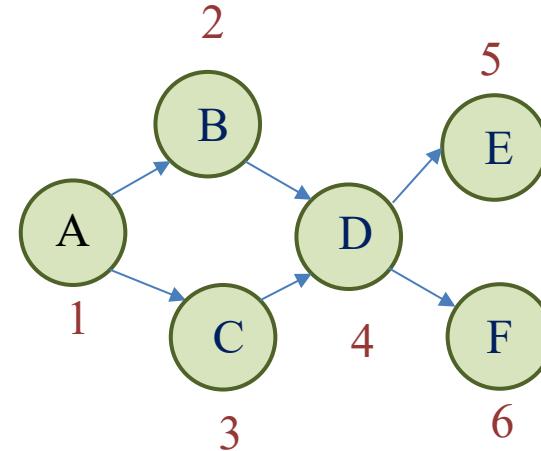
Visits each vertex by level.
Checks level 1, then level 2, etc.
From left to right.

Depth-First and Breadth-First Graph Traversal



Depth-First Traversal

Traverse as deep as possible into the graph before returning to nodes of lower depth. Access nodes that are deeper in graph before nodes that are earlier in the graph.



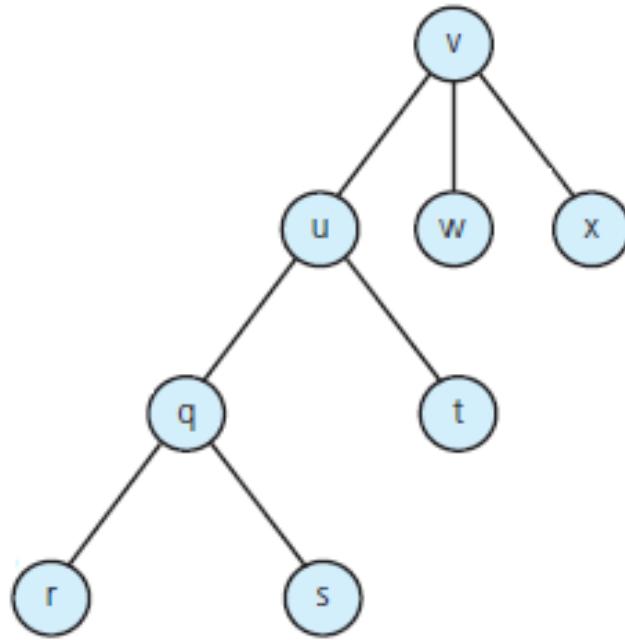
Breadth-First Traversal

Traverse as broadly as possible into the graph before returning to nodes of higher depth. Access nodes that are less deep in the graph before nodes that are deeper in the graph.

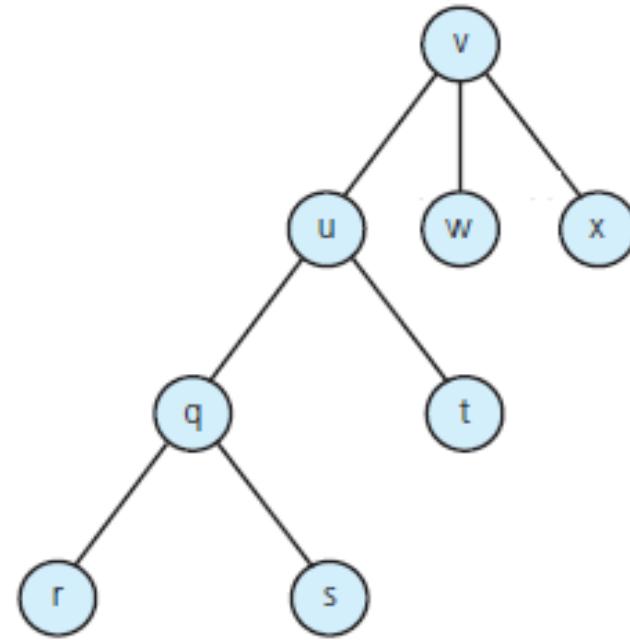
You Try 1. DF and BF Traversal



Specify the order that nodes are visited.



DF Traversal



BF Traversal

Graph Search

- Traverse through a graph based on its edges and visit all nodes in the graph in a particular order.

General Search Strategy

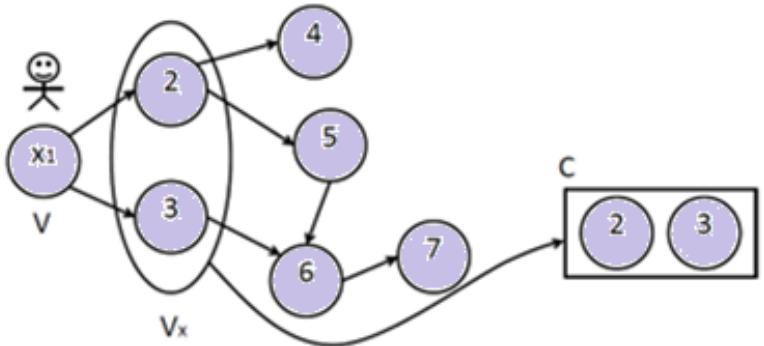
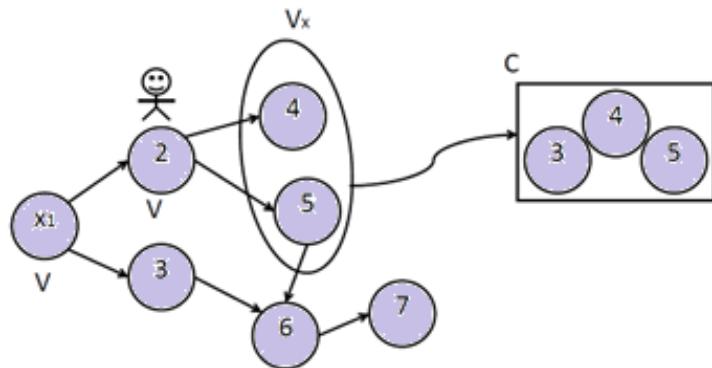
At node v_x , access its adjacency set V_x and insert all unvisited nodes in V_x into some container C.

Remove a node v_y from C, make it as visited and insert all unvisited nodes in its corresponding adjacency set V_y into C.

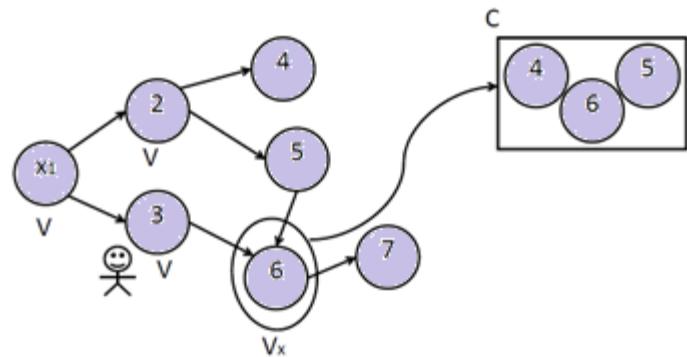
Repeat until the container is empty.

Graph Search

1. At node v_x , access its adjacency set V_x and insert all unvisited nodes in V_x into some container C.

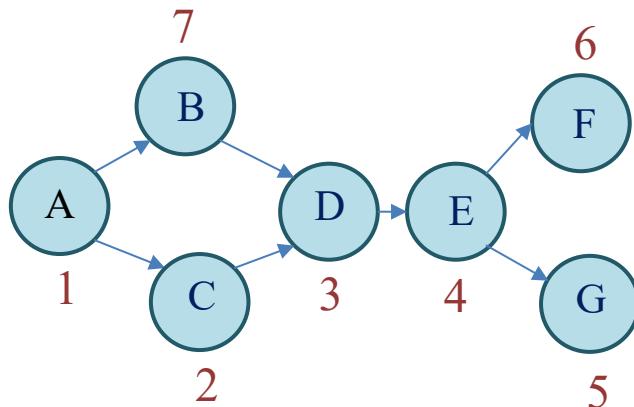


2. Remove a node v_y from C, make it as visited and insert all unvisited nodes in its corresponding adjacency set V_y into C.

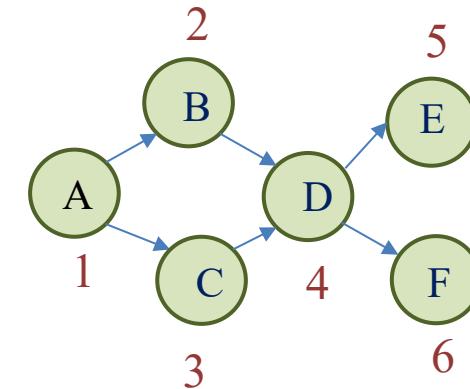
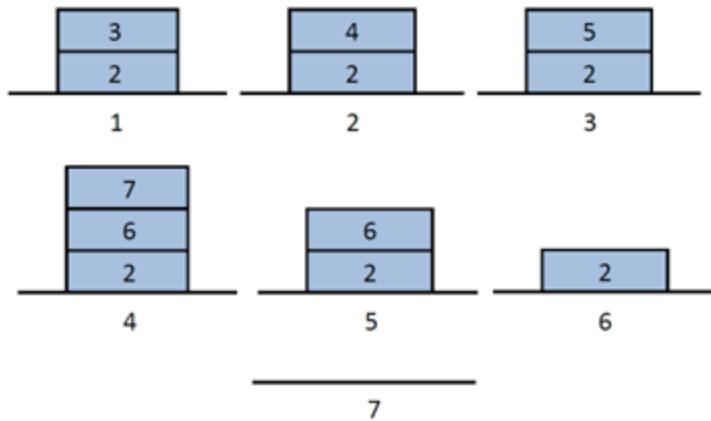


3. Repeat until the container is empty

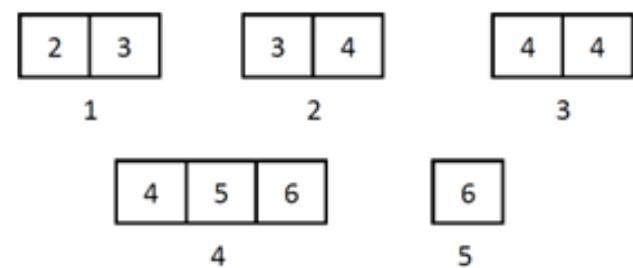
Depth-First and Breadth-First Graph Traversal



The logical order for a container in **depth-first** is a stack.



The logical order for a container in **breadth-first** is a queue.



Recursive Depth-First Search (DFS) Traversal

Pseudocode

// Traverses a graph beginning at vertex v by using a
// depth-first search: Recursive version.

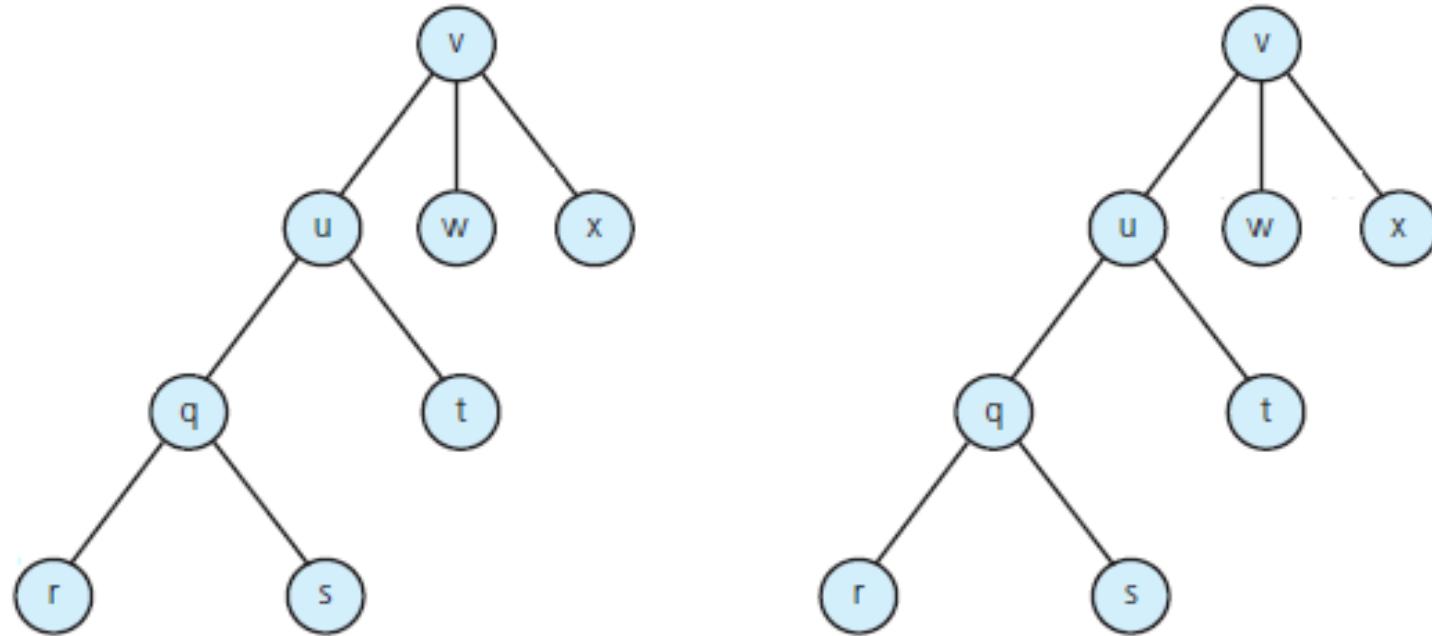
dfs(v: Vertex)

Mark v as visited

for (*each unvisited vertex u adjacent to v*)

 dfs(u)

Example 9. DFS Traversal Algorithm



The DFS traversal algorithm marks and then visits each of the vertices v , u , q , and r . When the traversal reaches a vertex—such as r —that has no unvisited adjacent vertices, it backs up and visits, if possible, an unvisited adjacent vertex. Thus, the traversal backs up to q and then visits s .

Iterative Depth-First Search (DFS) Traversal

// Traverses a graph beginning at vertex v by using a DFS Iterative version.

dfs(v: Vertex)

s= a new empty stack

s.push(v) // Push v onto the stack and mark it

Mark v as visited

// Loop invariant: there is a path from vertex v at the

// bottom of the stack s to the vertex at the top of s

while (!s.isEmpty())

{

if (no unvisited vertices are adjacent to the vertex on the top of the stack)

s.pop() // Backtrack

else

{

Select an unvisited vertex u adjacent to the vertex on top of the stack

s.push(u)

Mark u as visited

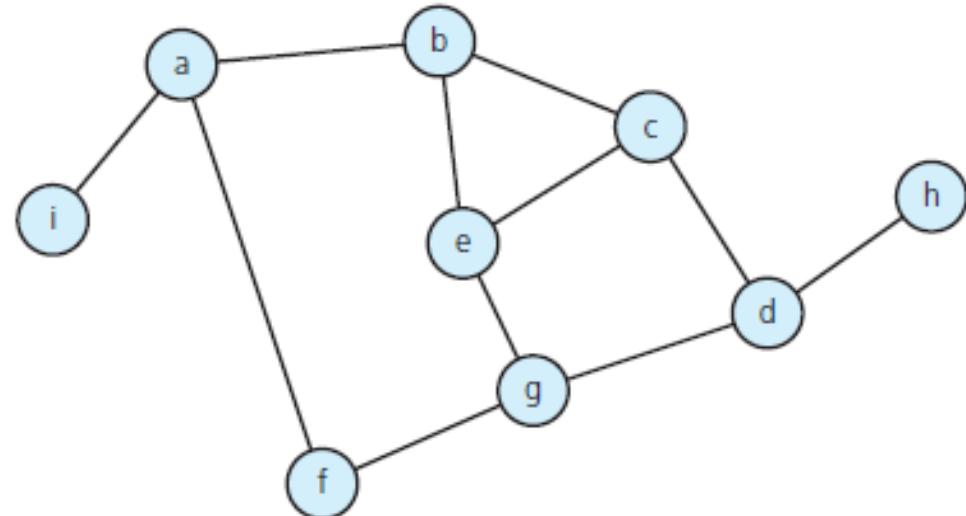
}

}

Example 10. DFS Traversal Algorithm using a stack

<u>Node visited</u>	<u>Stack (bottom to top)</u>
a	a
b	a b
c	a b c
d	a b c d
g	a b c d g
e	a b c d g e
(backtrack)	a b c d g
f	a b c d g f
(backtrack)	a b c d g
(backtrack)	a b c d
h	a b c d h
(backtrack)	a b c d
(backtrack)	a b c
(backtrack)	a b
(backtrack)	a
i	a i
(backtrack)	a
(backtrack)	(empty)

start at vertex a



The graph is connected, a **DFS** traversal will visit every vertex in this order:
a, b, c, d, g, e, f, h, i.
last visited, first explored strategy (stack)

Iterative Breadth-First Search (BFS) Traversal

```
// Traverses a graph beginning at vertex v by using a
// breadth-first search: Iterative version.

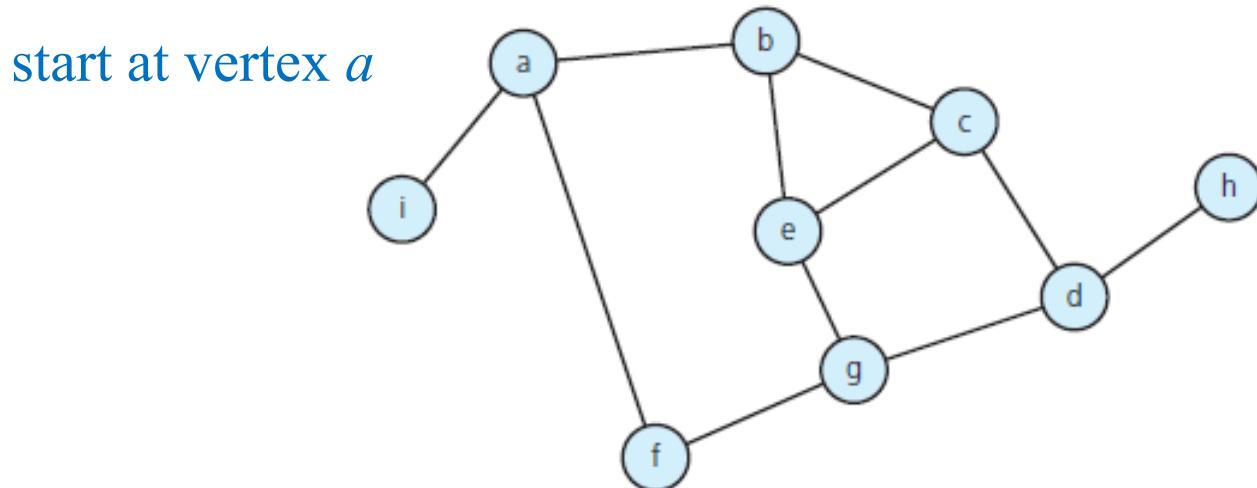
bfs(v: Vertex)
    q = a new empty queue
    // Add v to queue and mark it
    q.enqueue(v)
    Mark v as visited
    while (!q.isEmpty())
    {
        q.dequeue(w)
        // Loop invariant: there is a path from vertex w to every vertex in the
        // queue q
        for (each unvisited vertex u adjacent to w)
        {
            Mark u as visited
            q.enqueue(u)
        }
    }
```

You Try 2. BFS Traversal



The **BFS** traversal visits all of the vertices in this order:

1. $a, b, c, d, g, e, f, h, i.$
2. $a, f, g, e, b, c, d, h, i.$
3. $a, b, f, i, c, e, g, d, h.$
4. $a, i, g, e, b, c, d, h, f.$



Next Lecture

We focus on:

- Graph Matrix Representation
- Graph Linked Representation

Readings/ Study Reference

Course Textbook: “*Data Structure and Algorithms in a Nutshell*” A. Wong et al.

Chapter 8. Graphs

Section 8.1. Overall structure

Section 8.2. Cycle

Section 8.3. Degree of a node

Section 8.5. Search

The End of Lecture

Course: Data Structures and Algorithms

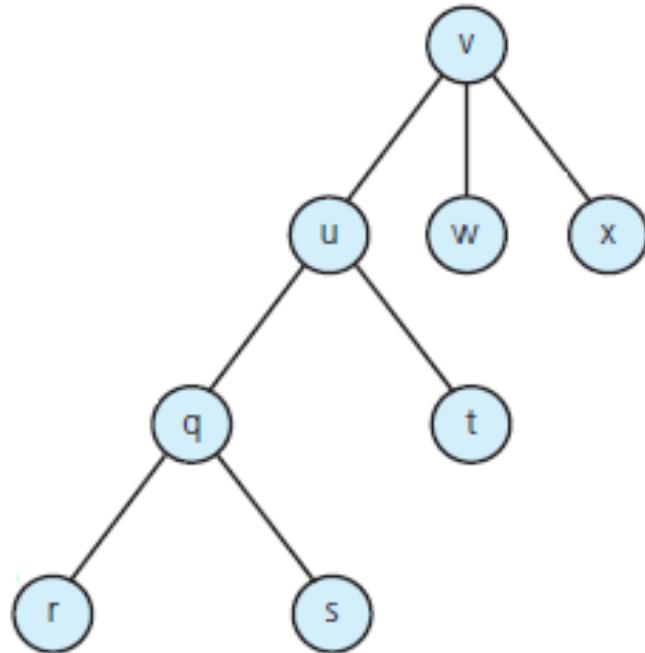
Instructor: Homeyra Pourmohammadali

You Try Questions and Solutions
Graph Structure, ADT and Search Traversals

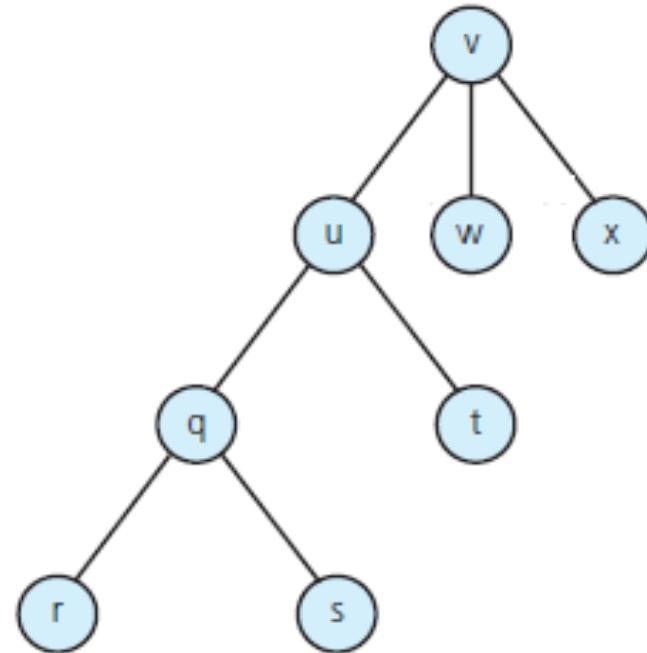
You Try 1. DF and BF Traversal



Specify the order that nodes are visited.

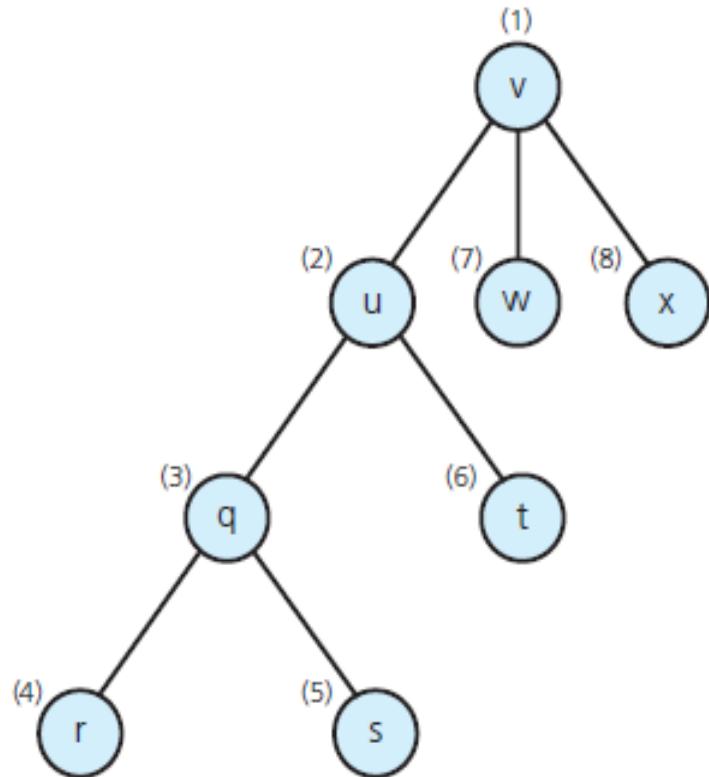


DF Traversal

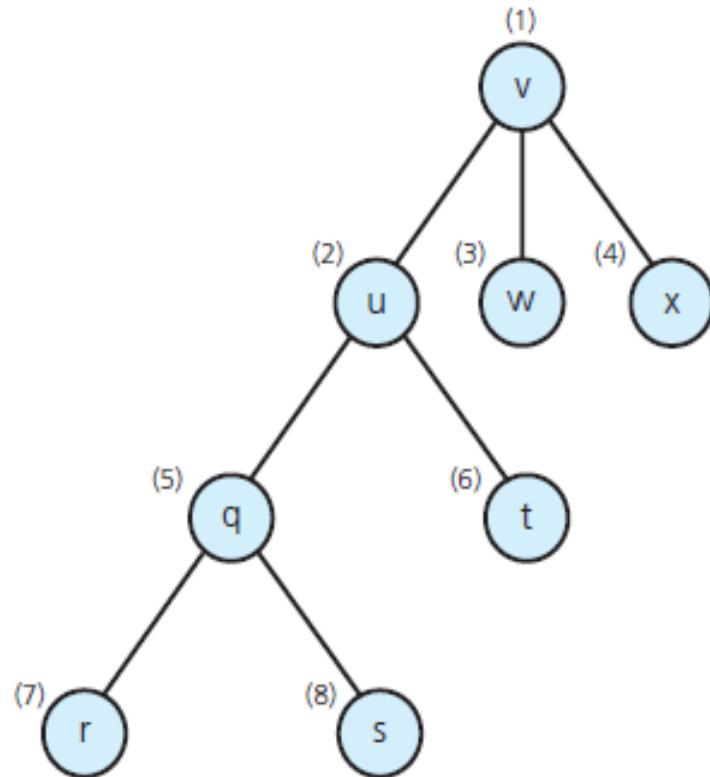


BF Traversal

You Try 1 Solution. DF and BF Traversal



DF Traversal



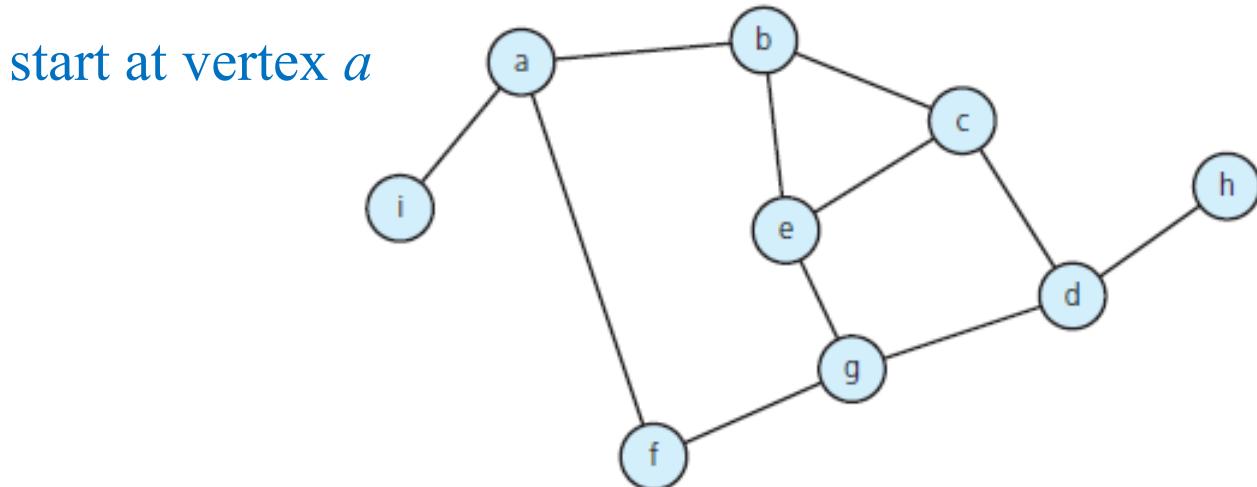
BF Traversal

You Try 2. BFS Traversal



The **BFS** traversal visits all of the vertices in this order:

1. $a, b, c, d, g, e, f, h, i.$
2. $a, f, g, e, b, c, d, h, i.$
3. $a, b, f, i, c, e, g, d, h.$
4. $a, i, g, e, b, c, d, h, f.$

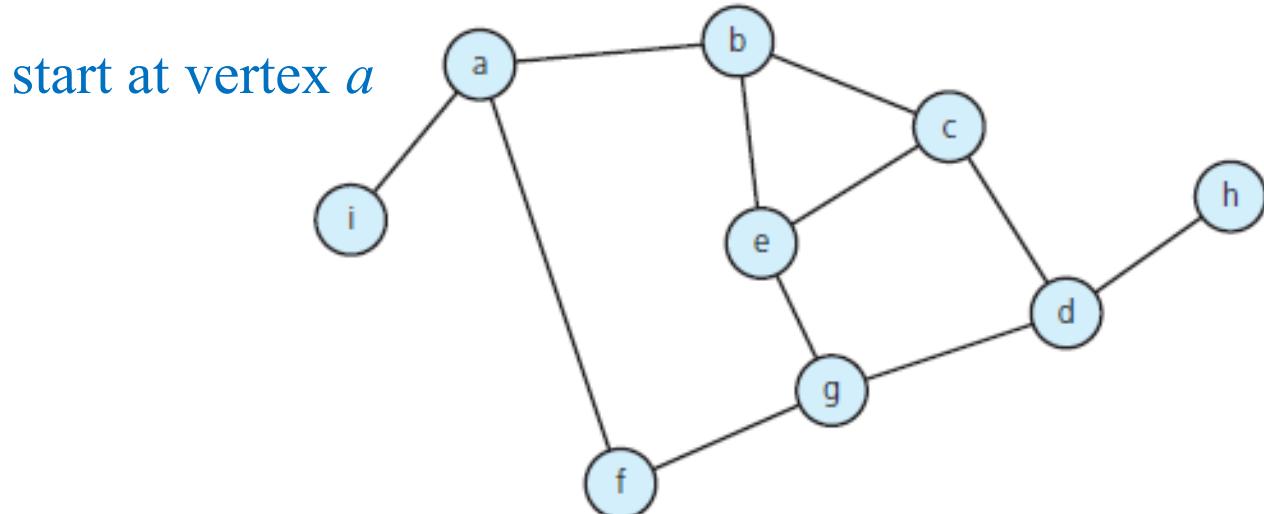


You Try 2 Solution. BFS Traversal

The **BFS** traversal visits all of the vertices in this order:

1. $a, b, c, d, g, e, f, h, i.$
2. $a, f, g, e, b, c, d, h.$
3. $a, b, f, i, c, e, g, d, h.$
4. a, i, g, e, b, c, d, h

A breadth-first search will visit the same vertices as a depth-first search, but in a different order.



The End of You Try Activities

Course: Data Structures and Algorithms

Instructor: Homeyra Pourmohammadali

Matrix and Linked Representations of Graphs

Learning Outcomes

By the end of this lecture you will be able to:

- find out how a graph can be represented using adjacency matrix and an adjacency list
- practice on both types representations
- compare memory efficiency of two representations

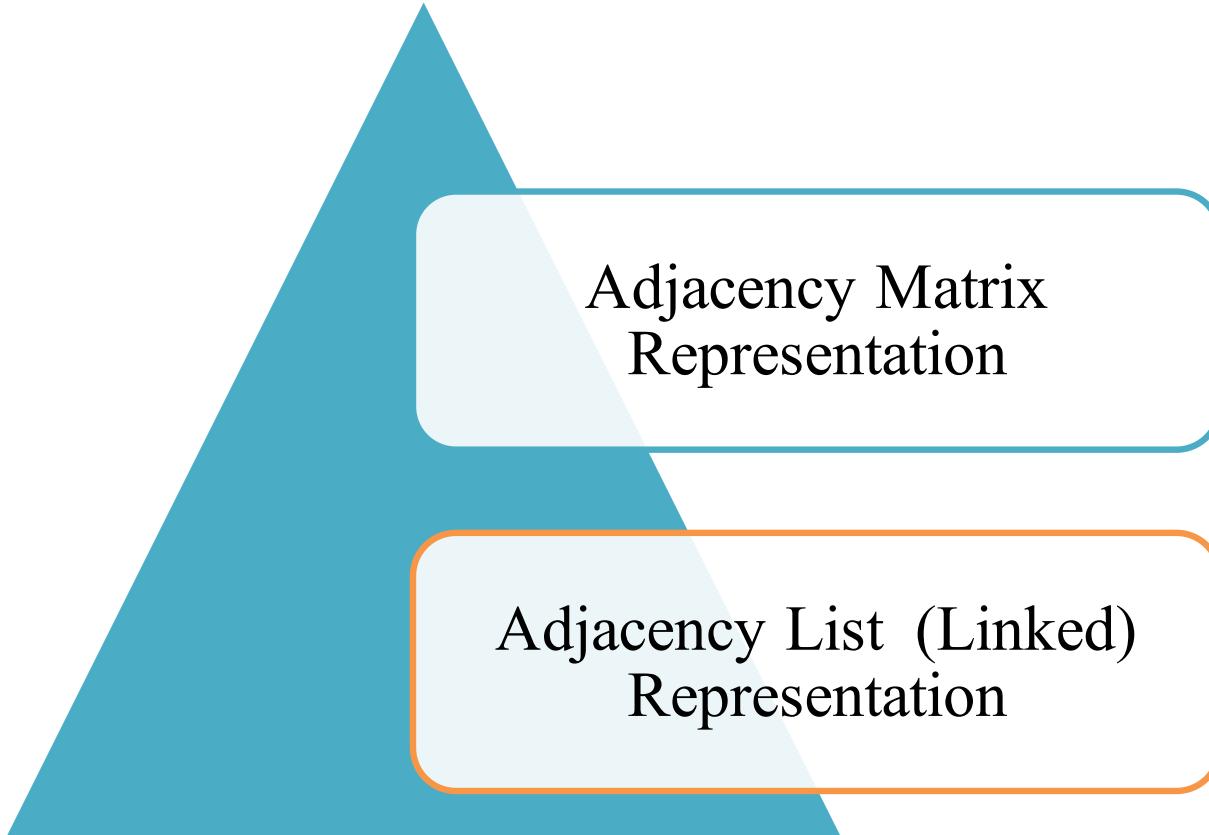
Recall

- Graphs can relax the shape property of a data structure to allow us to represent arbitrary networks of information.
- Graphs are made up of a set of nodes called vertices that are connected by links called edges.
- The vertices of a graph usually represent objects, and the edges represent relationships between the objects.
- Edges can be undirected or directed, and they can have associated weights.

Recall

- Breadth- first and depth-first are two common approaches to traverse a tree or a graph.
- There are many other approaches and goals for graph traversals. For instance, one goal of a traversal may be to establish a minimal cost path between two vertices.

Graph Representations



Adjacency Matrix
Representation

Adjacency List (Linked)
Representation

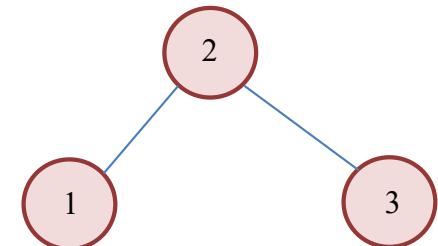
Adjacency Matrix Representation

- The graph is defined by a set of all nodes in the graph $V = \{v_1, v_2, \dots, v_n\}$ and a matrix M that specifies all edges between each node.
- M for a graph with n nodes (vertices) is an $n \times n$ matrix.
- For an unweighted graph, elements $M(i, j)$ is set to:
 - 1 if there exists an edge going from node i to node j .
(means that i and j are adjacent)
 - 0 if there is no edge between node i and j .
(means that i and j are not adjacent)

Example 1. Adjacency Matrix of an Unweighted Graph

- In this unweighted graph node 1 has no loop or edge with itself then matrix element: $M(1,1)=0$
- This is the same for nodes 2 and 3.
 $M(2, 2)=0, M (3,3)=0$ (all diagonal elements=0)
- Node 1 and 2 are connected by an edge, then $M (1, 2)=1$
- Node 2 and 1 are also connected by the same edge $M(2,1)=1$
- There is no edge between nodes 1 and 3: $M(1,3)=M(3,1)=0$
There is an edge between nodes 2 and 3: $M(2,3)=M(3,2)=1$
- Then M for this graph is:

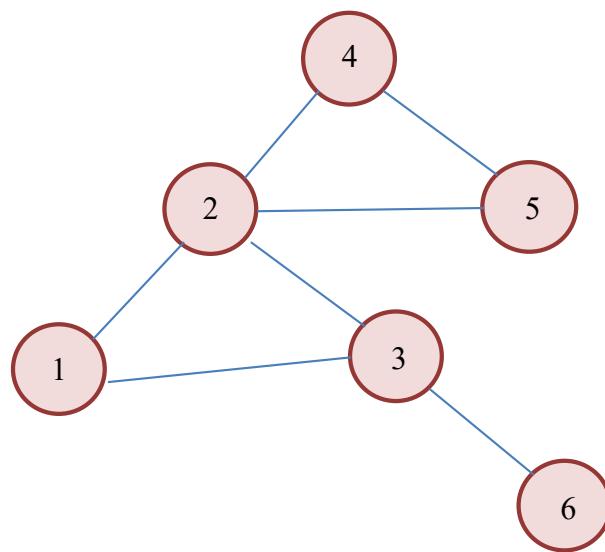
$$\begin{array}{c} \textcolor{violet}{j} \\ \textcolor{violet}{i} \\ \textcolor{blue}{1} \\ \textcolor{blue}{2} \\ \textcolor{blue}{3} \end{array} \begin{array}{cccc} & \textcolor{blue}{1} & \textcolor{blue}{2} & \textcolor{blue}{3} \\ \textcolor{blue}{1} & \left[\begin{array}{ccc} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{array} \right] \end{array}$$



Example 2. Adjacency Matrix of an Unweighted Graph

- In this unweighted graph, with no loop
(all diagonal elements=0)

	1	2	3	4	5	6
1	0					
2		0				
3			0			
4				0		
5					0	
6						0



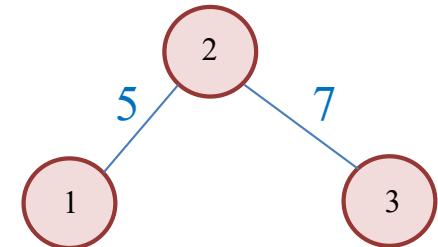
Adjacency Matrix Representation

- For undirected graphs, $M[i][j]$ is always same as $M[j][i]$ in the 2D array. In other words, an unweighted graph can be represented by a symmetrical matrix.
- For a weighted graph, elements $M(i, j)$ is set to:
 - each weight $w(i , j)$ if there exists an edge going from node i to node j .
 - ∞ if there is no edge between node i and j .

Example 3. Adjacency Matrix of an Weighted Graph

- In this weighted graph node 1 has no edge with itself then matrix element: $M(1,1)=\infty$
- This is the same for nodes 2 and 3.
- $M(2, 2)=\infty, M (3,3)=\infty$ (all diagonal elements=0)
- Node 1 and 2 are connected by an edge, then $M (1, 2)=M (2, 1)=5$
- There is no edge between nodes 1 and 3: $M(1,3)=M(3,1)=\infty$
- Then M for this graph is:

$$\begin{matrix} & \text{j} & 1 & 2 & 3 \\ i & & \begin{bmatrix} \infty & 5 & \infty \\ 5 & \infty & 7 \\ \infty & 7 & \infty \end{bmatrix} \\ 1 & & & & \\ 2 & & & & \\ 3 & & & & \end{matrix}$$



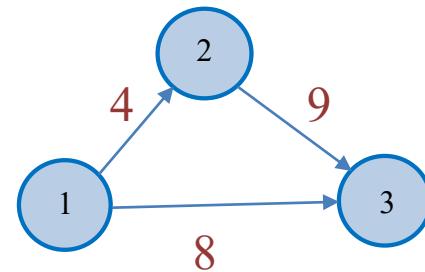


You Try 1.

Adjacency Matrix of a Weighted Directed Graph

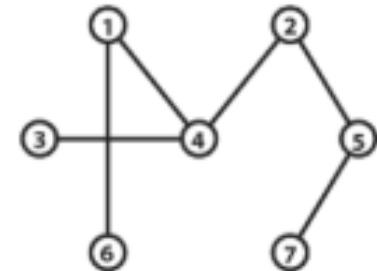
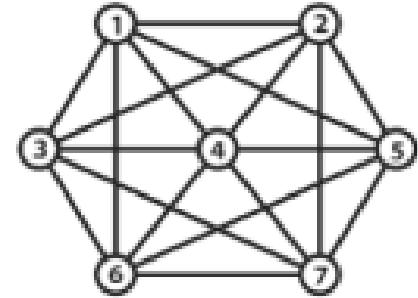
Represent this graph using adjacency matrix.

Is the matrix still symmetrical?



Space Efficiency of Matrix Representation

- The matrix is more appropriate for representation of **dense graphs** where each node is connected to majority of other nodes in the graph.
- The matrix representation for **sparse graphs**, where each node is connected to only a few nodes, would be **inefficient** in terms of memory usage.
- Memory requirements of matrix representation approach is $O(n^2)$, regardless of the number of edges.



<http://www.mcihanazer.com/tips/artificial-intelligence/graphs/graph-basics/>

You Try 2. Matrix Representation



1. Represent this unweighted graph using adjacency matrix.
2. Assume the following weights are applied to connecting edges, and re-represent the adjacency matrix:

$$w(A, B) = 2$$

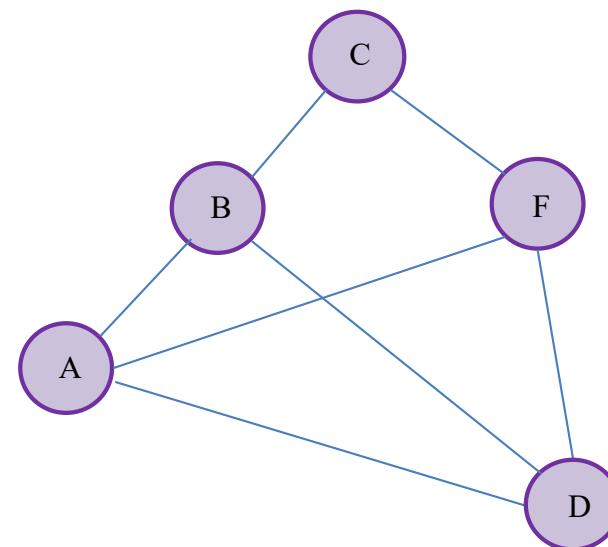
$$w(B, C) = 5$$

$$w(C, F) = 7$$

$$w(D, F) = 3$$

$$w(A, F) = 2$$

$$w(B, D) = 4$$

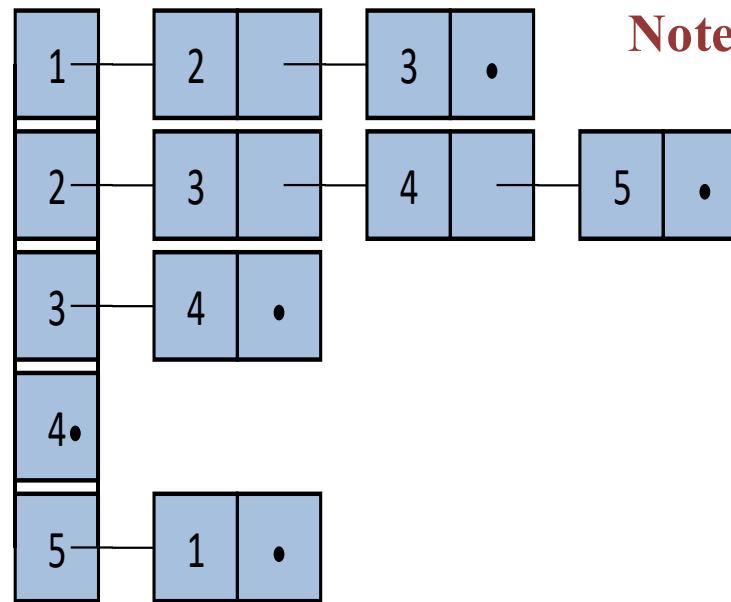
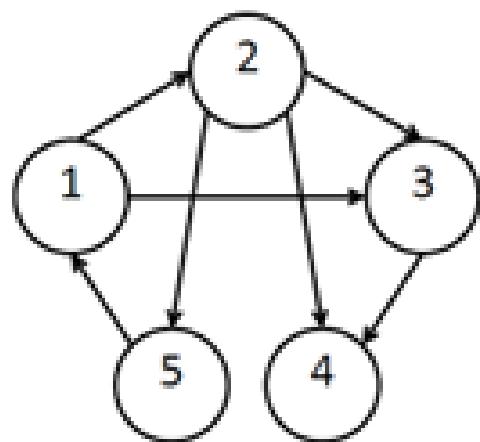


Adjacency List

- The graph G is defined by the set of all nodes in the graph $V = \{v_1, v_2 \dots, v_n\}$ and a list of linked lists L that specifies all edges that originates at each node.
- The list element $L(i)$ points to the i^{th} linked list where each node j in the list represents an edge going from node i to node j .
- Each pointer is the head of an adjacency list (linked list)

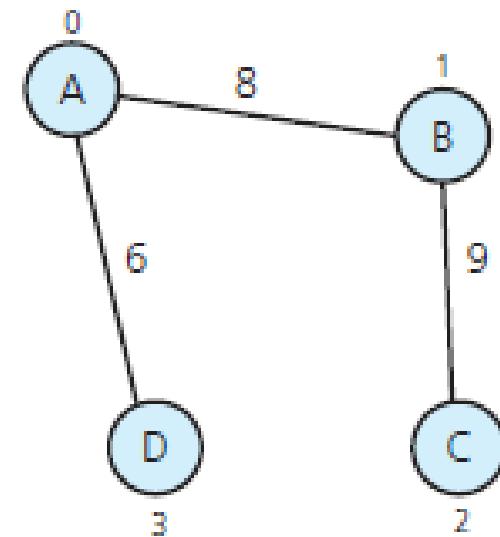
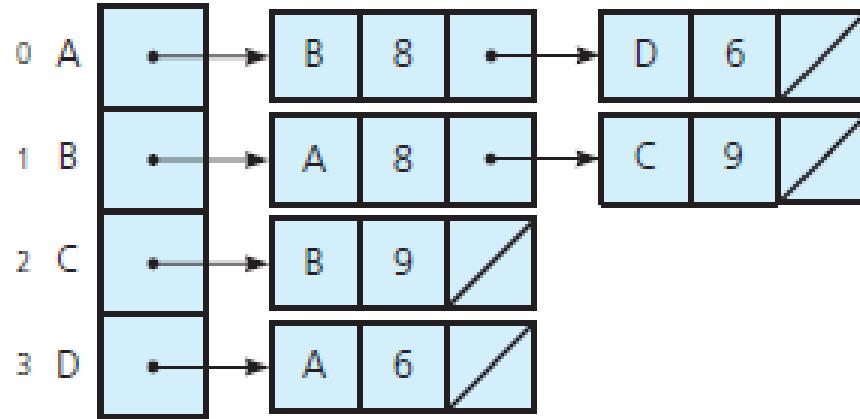
Example 4. Adjacency List Representation of a Graph

- The graph is directed, and unweighted.
- 1D array of node pointers
- Each pointer is the head of an adjacency list (linked list)



Example 5. Adjacency List Representation of a Graph

The graph is undirected and weighted.



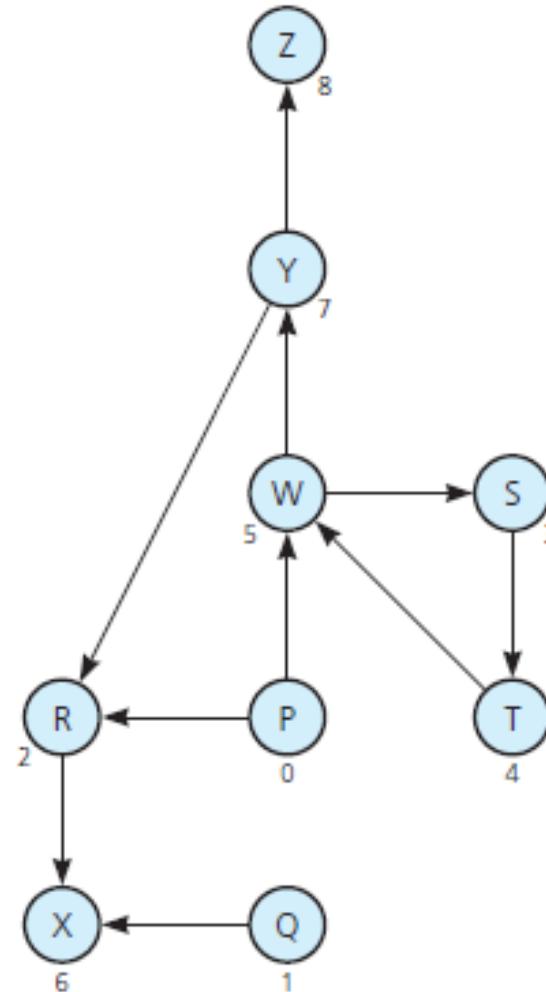
The adjacency list for an undirected graph treats each edge as if it were two directed edges in opposite directions.

You Try 3. Adjacency List Representation of a Graph



Show the adjacency list of this graph.

Starting with node P.



Space Efficiency of Linked Representation

- Memory requirements of matrix representation approach is $O(n)$.
- **Advantage:** it allows for significant memory overhead reductions compared to matrix representation.
- **Drawback:** it is necessary to traverse the linked list to gain access to the individual elements of the linked list, then access operation is $O(n)$.

Summary

- We saw how graphs can be represented easily by an adjacency matrix.
- However, when a graph has many vertices and few edges, the adjacency matrix can be mostly filled with null values, and thus is very inefficient in terms of storage space.
- A linked representation can be more space efficient, but it also involves more algorithmic complexity in managing the addition and removal of vertices in adjacency lists.

Next Lecture

We focus on:

- Graph applications
- Finding shortest path

Readings/ Study Reference

Course Textbook: “*Data Structure and Algorithms in a Nutshell*” A. Wong et al.

Chapter 8. Graphs

Section 8.4. Graph Representations

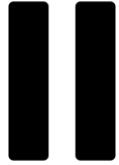
The End of Lecture

Course: Data Structures and Algorithms

Instructor: Homeyra Pourmohammadali

You Try Questions and Solutions

Matrix and Linked Representations of Graphs

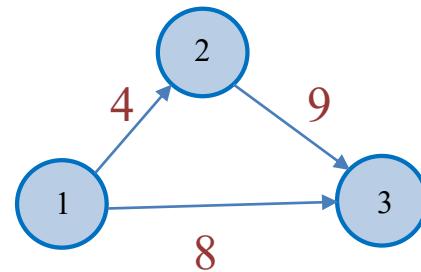


You Try 1.

Adjacency Matrix of a Weighted Directed Graph

Represent this graph using adjacency matrix.

Is the matrix still symmetrical?



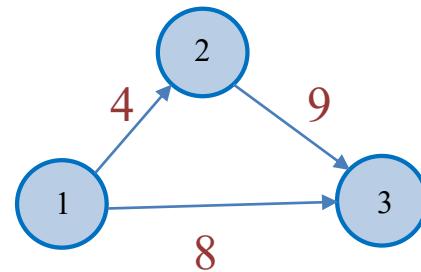
You Try 1 Solution.

Adjacency Matrix of a Weighted Directed Graph

Represent this graph using adjacency matrix.

Is the matrix still symmetrical?

$$\begin{matrix} & \text{j} & 1 & 2 & 3 \\ \text{i} & & \left[\begin{matrix} \infty & 4 & 8 \\ \infty & \infty & 9 \\ \infty & \infty & \infty \end{matrix} \right] \end{matrix}$$



The matrix is not symmetrical.

You Try 2. Matrix Representation



1. Represent this unweighted graph using adjacency matrix.
2. Assume the following weights are applied to connecting edges, and re-represent the adjacency matrix:

$$w(A, B) = 2$$

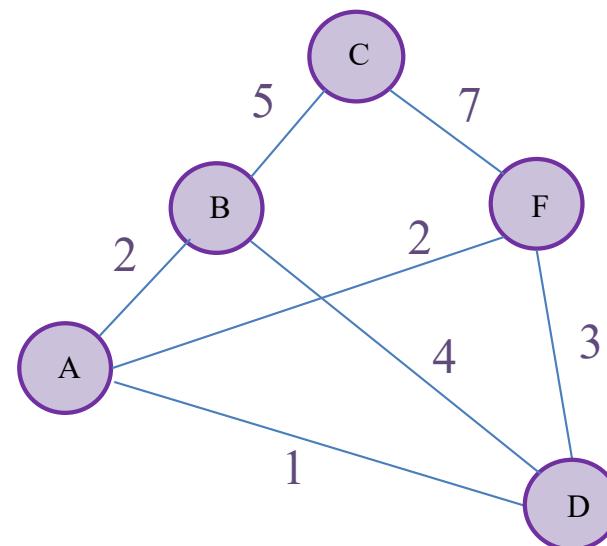
$$w(B, C) = 5$$

$$w(C, F) = 7$$

$$w(D, F) = 3$$

$$w(A, F) = 2$$

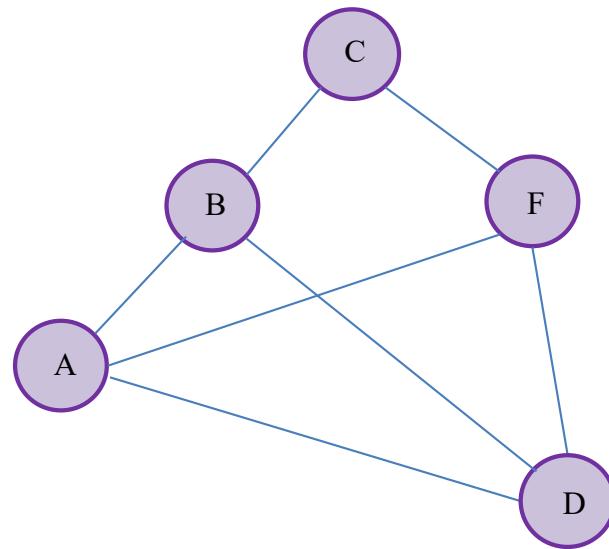
$$w(B, D) = 4$$



You Try 2 Solution.

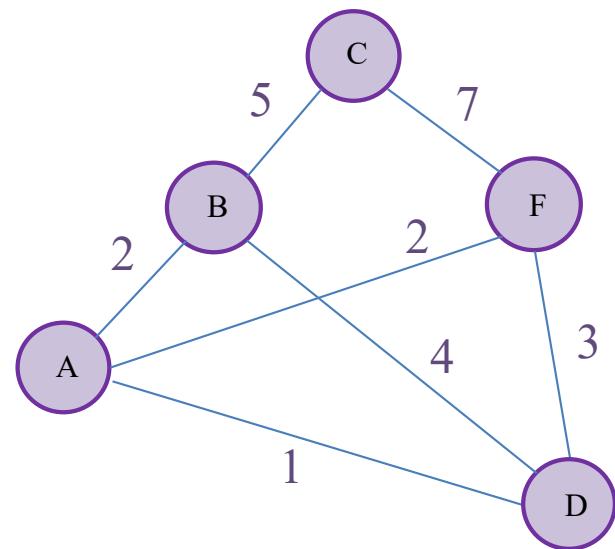
Matrix Representation of Unweighted Graph

	A	B	C	D	F
A	0	1	0	1	1
B	1	0	1	1	0
C	0	1	0	0	1
D	1	1	0	0	1
F	1	0	1	1	0



You Try 2 Solution. Matrix Representation of Weighted Graph

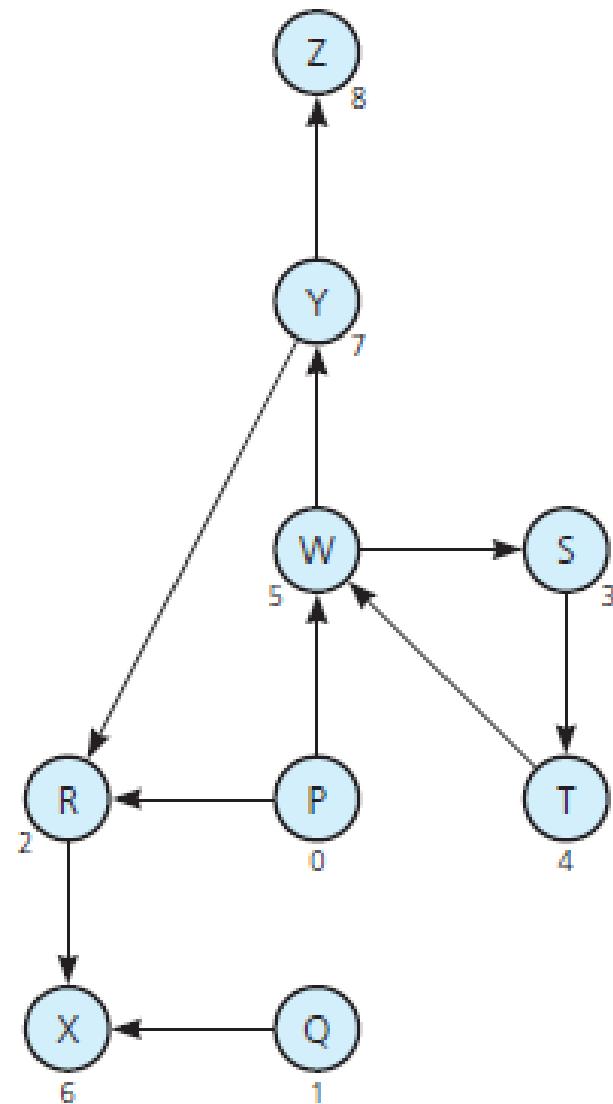
	A	B	C	D	F
A	0	2	0	1	2
B	2	0	5	4	0
C	0	5	0	0	7
D	1	4	0	0	3
F	2	0	7	3	0



You Try 3. Adjacency List Representation of a Graph

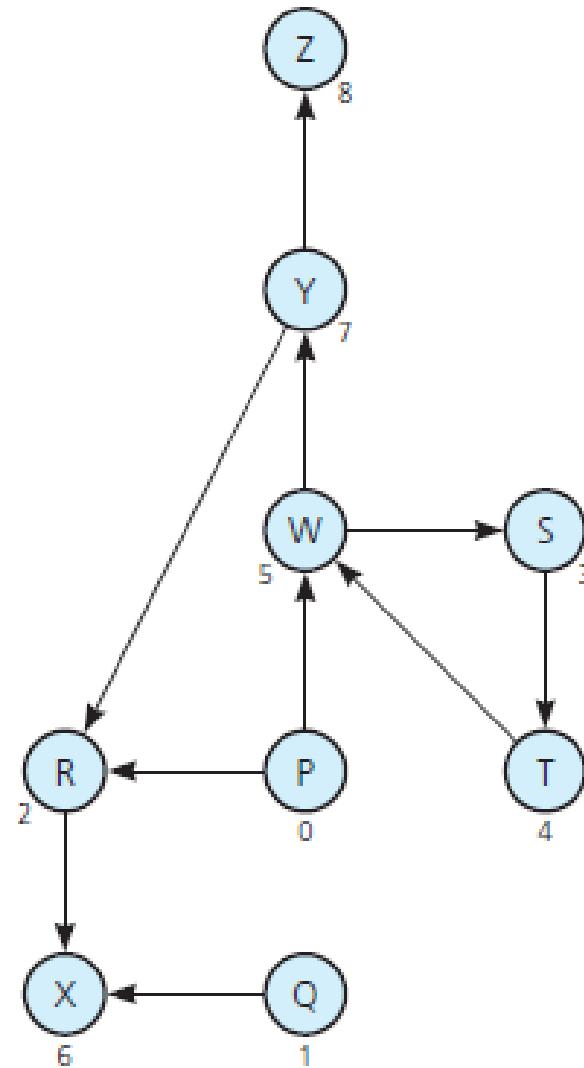
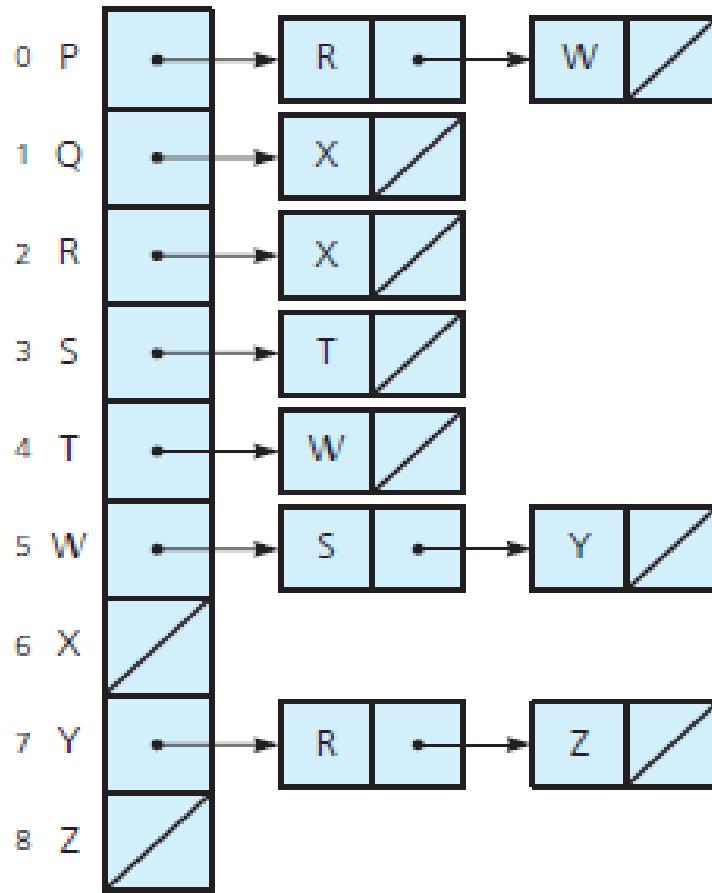
Show the adjacency list of this graph.

Starting with node P.



You Try 3 Solution. Adjacency List of a Graph

Graph is directed and unweighted.



The End of You Try Activities

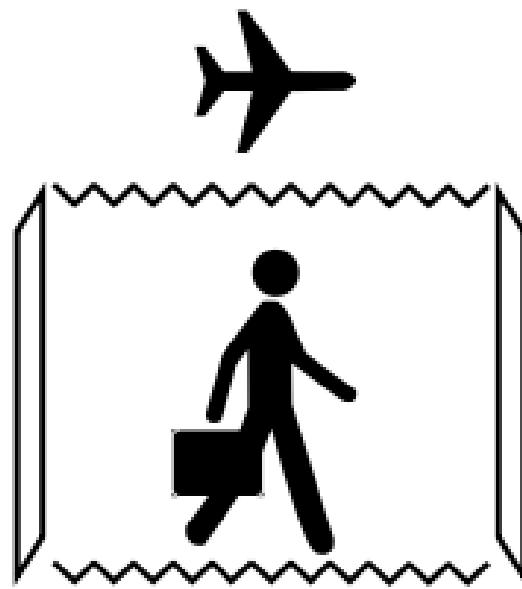
Course: Data Structures and Algorithms

Instructor: Homeyra Pourmohammadali

Graph Applications:
Topological Sorting & Finding Shortest Path

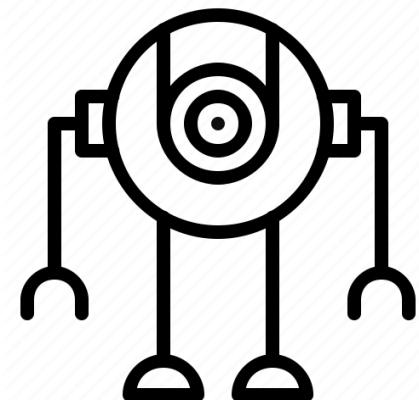
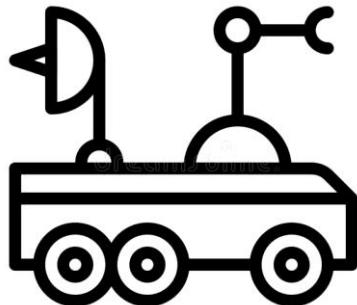
Motivation

- Modelling flight connections between the airports around the world and planning **optimal flight routes** for travelling between different destinations is one of the graph applications.



Motivation

- Modelling area maps within a topological framework, which is crucial for **path-finding for robot navigation**, is another application of graphs.



Motivation

- Consider the problem of scheduling tasks which are inter-dependent ie. let's say task A can only be done after task B and C have been completed. We can model such dependencies using a directed acyclic graph which would contain an edge B->A and C->A.
- A **topological sort** of such a graph would give us an order in which these tasks could be completed.



Learning Outcomes

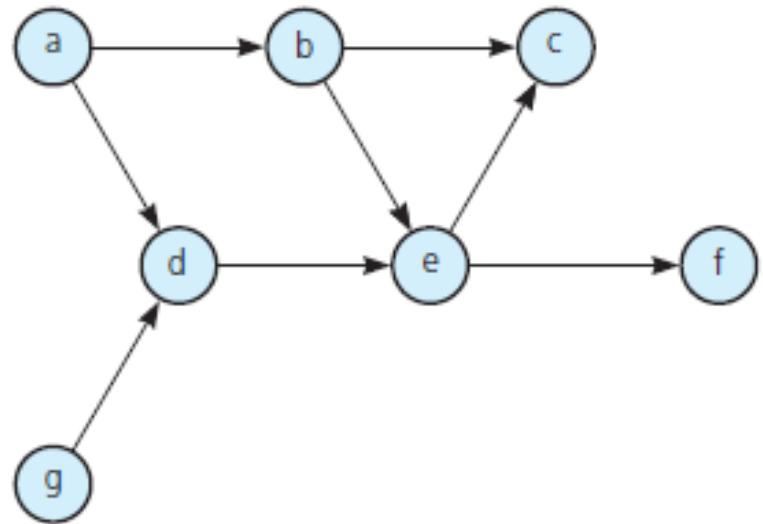
By the end of this lecture you will be able to:

- use graphs in topological sorting
- use graphs for finding shortest path using Dijkstra's algorithm

Application of Graphs: Topological Sorting

Topological Sorting

- A directed graph without cycles (acyclic directed graph), has a natural order (e.g. vertex a precedes b , which precedes c .)
- If the vertices represent academic courses, the graph represents the prerequisite structure for the courses.
- For example, course a is a prerequisite to course b , which is a prerequisite to both courses c and e .

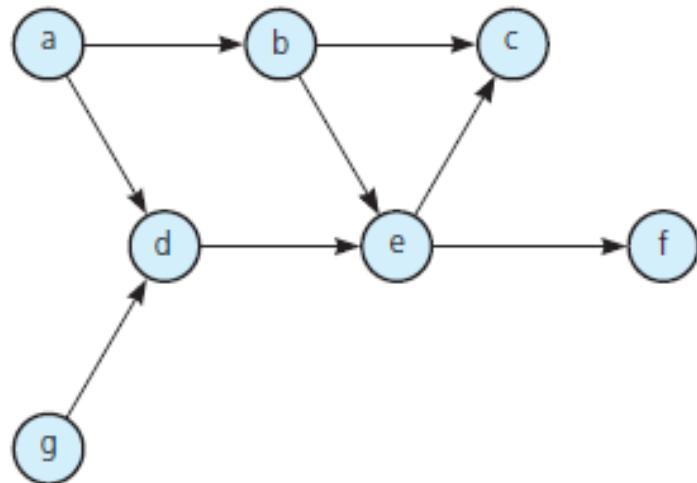


Directed graph

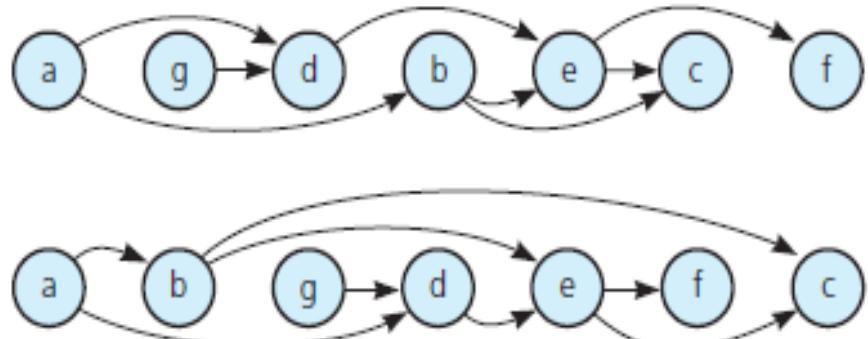
Topological Sorting

The vertices in a given graph may have several topological orders.

For example, consider two topological orders:



Directed graph



The graph arranged according
to the topological orders:

Top: *a, g, d, b, e, c, f*

Bottom: *a, b, g, d, e, f, c*

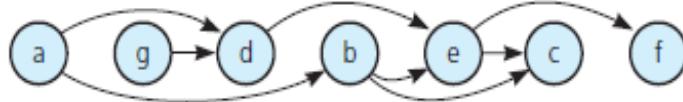
A Simple Topological Sorting Algorithm

// Arranges the vertices in graph theGraph into a
// topological order and places them in list aList.

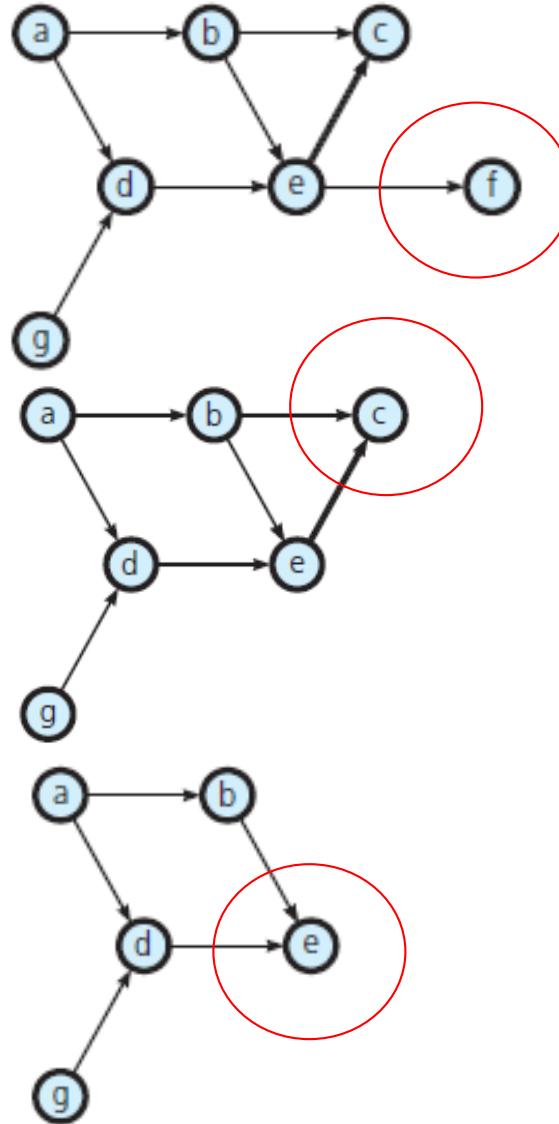
```
topSort1(theGraph: Graph, aList: List)
    n = number of vertices in theGraph
    for (step = 1 through n)
    {
        Select a vertex v that has no successors
        aList.insert(1, v)
        Remove from theGraph vertex v and its edges
    }
```

When the traversal ends, the list aList of vertices will be in topological order.

A Trace of Algorithm



Remove **f** from **theGraph**;
add it to **aList**.



Remove **c** from **theGraph**;
add it to **aList**.

List **aList**

f

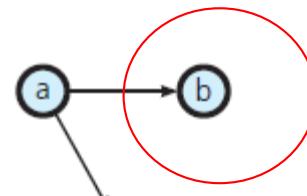
c f

Remove **e** from **theGraph**;
add it to **aList**.

e c f

A Trace of Algorithm

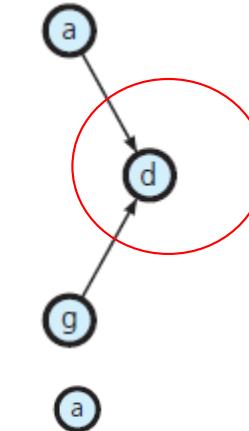
Remove **b** from **theGraph**;
add it to **aList**.



List **aList**

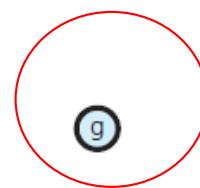
b e c f

Remove **d** from **theGraph**;
add it to **aList**.



d b e c f

Remove **g** from **theGraph**;
add it to **aList**.



g d b e c f

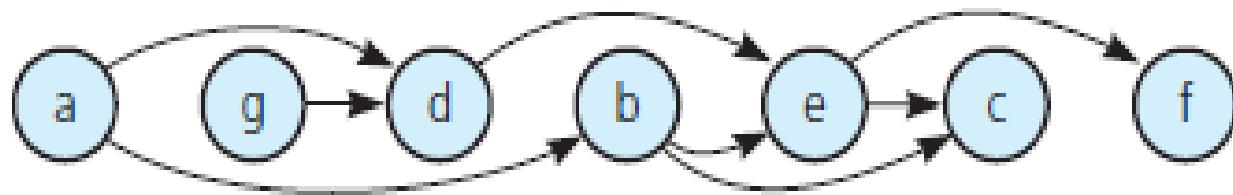
A Trace of Algorithm

Remove **a** from **theGraph**;
add it to **aList**.

List **aList**



a g d b e c f



Topological Sorting Algorithm

- Another algorithm is a simple modification of the iterative depth-first search algorithm.
- First you push all vertices that have no predecessor onto a stack.
- Each time you pop a vertex from the stack, you add it to the beginning of a list of vertices.

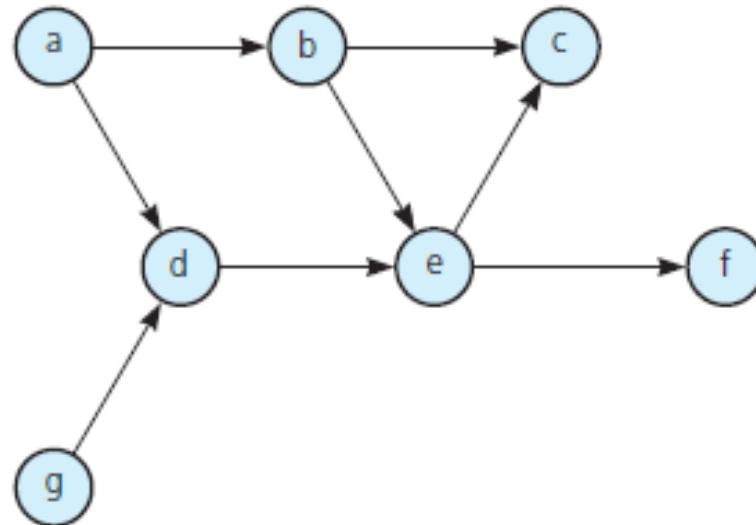
Another Topological Sorting Algorithm

```
// Arranges the vertices in graph theGraph into a topological order and places them in list aList.
topSort2(theGraph: Graph, aList: List)
    s = a new empty stack
    for (all vertices v in the graph)
        if (v has no predecessors)
    {
        s.push(v)
        Mark v as visited
    }
    while (!s.isEmpty())
    {
        if (all vertices adjacent to the vertex on the top of the stack have been visited)
        {
            s.pop(v)
            aList.insert(1, v)
        }
        else
        {
            Select an unvisited vertex u adjacent to the vertex on the top of the stack
            s.push(u)
            Mark u as visited
        }
    }
```

You Try 1. Topological Sorting Algorithm



In a table show a trace of topSort2 (in previous slide) for this graph. Show the content of stack **S** (bottom to top) and list **aList** (beginning to end) after each push or pop operation.



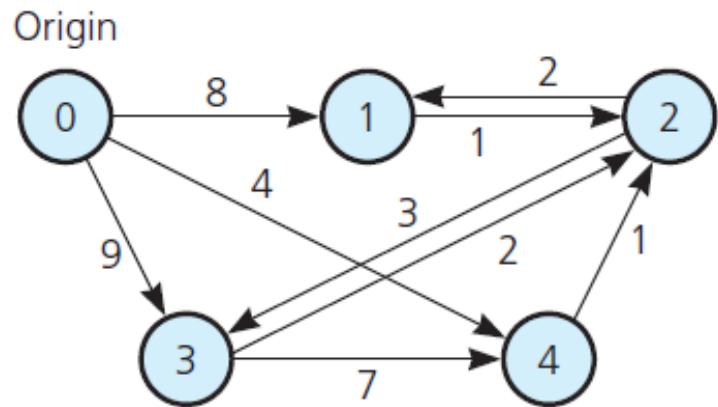
Application of Graphs: Finding Shortest Path

Shortest Path

- A weighted directed graph can represent the map of air flight routs, or any other transportation system.
- The **shortest path** between two given vertices in a weighted graph is the path that has the smallest sum of its edge weights.
- Although the term “shortest” is used, realize that the weights could be a measure other than distance, such as the cost of each flight in dollars or the duration of each flight in hours.
- The sum of the weights of the edges of a path is called the path’s **length** or **weight** or **cost** .

Example 1. Shortest Path

- Consider this graph:
- The shortest path from vertex 0 to vertex 1 is not the edge between 0 and 1—its cost is 8—but rather the path from 0 to 4 to 2 to 1, with a cost of 7.
- The graph's adjacency matrix can be represented as:



	0	1	2	3	4
0	∞	8	∞	9	4
1	∞	∞	1	∞	∞
2	∞	2	∞	3	∞
3	∞	∞	2	∞	7
4	∞	∞	1	∞	∞

Algorithm For Finding Shortest Path

- The algorithm determines the shortest paths between a given origin and *all* other vertices.
- The algorithm is called [Dijkstra's algorithm](#), developed by computer scientist Edsger W. Dijkstra (/ˈdaɪkstrə/) in 1956.



Edsger Wybe Dijkstra

Dijkstra's Algorithm

Construct the shortest paths from the starting node to its closest nodes.

Keep track of the visited nodes and update the shortest path information to each unvisited node.

Extend the paths until the destination node is reached.

Dijkstra's Algorithm

- It is a greedy algorithm.
- In greedy algorithm, you choose a solution that appears to be optimal solution at the time. You worry about possible new solutions later.
- It require a lot of bookkeeping to keep track of:
 - the shortest paths at each iteration
 - the sum of weights for these shortest paths
 - the visited and unvisited nodes

Dijkstra's Algorithm

Properties

- Weight of edge should be positive, cannot be negative (limitation).
- Distance to a node itself is defined as 0.

Inputs

- A graph (e.g., adjacency matrix).
- The start Node (e.g., 0)
- The end Node

Outputs

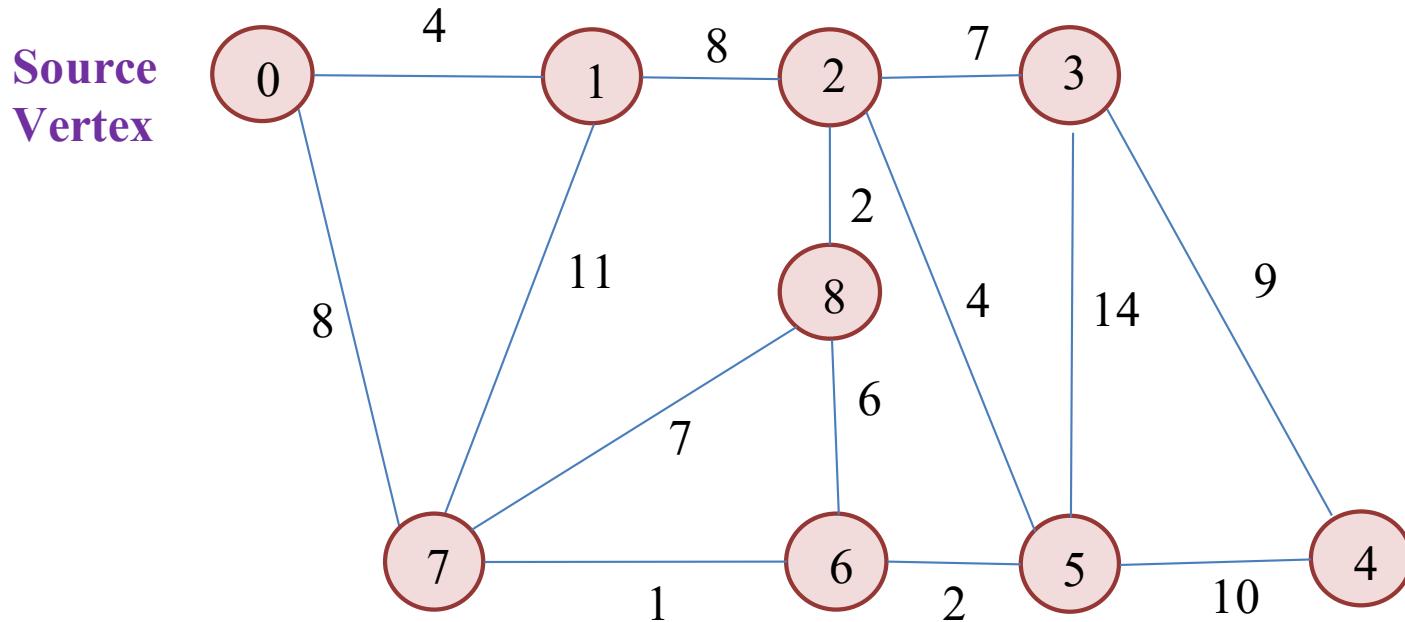
- The shortest path from start node to end node (e.g., print the path and the shortest distance)

Dijkstra's Algorithm Steps

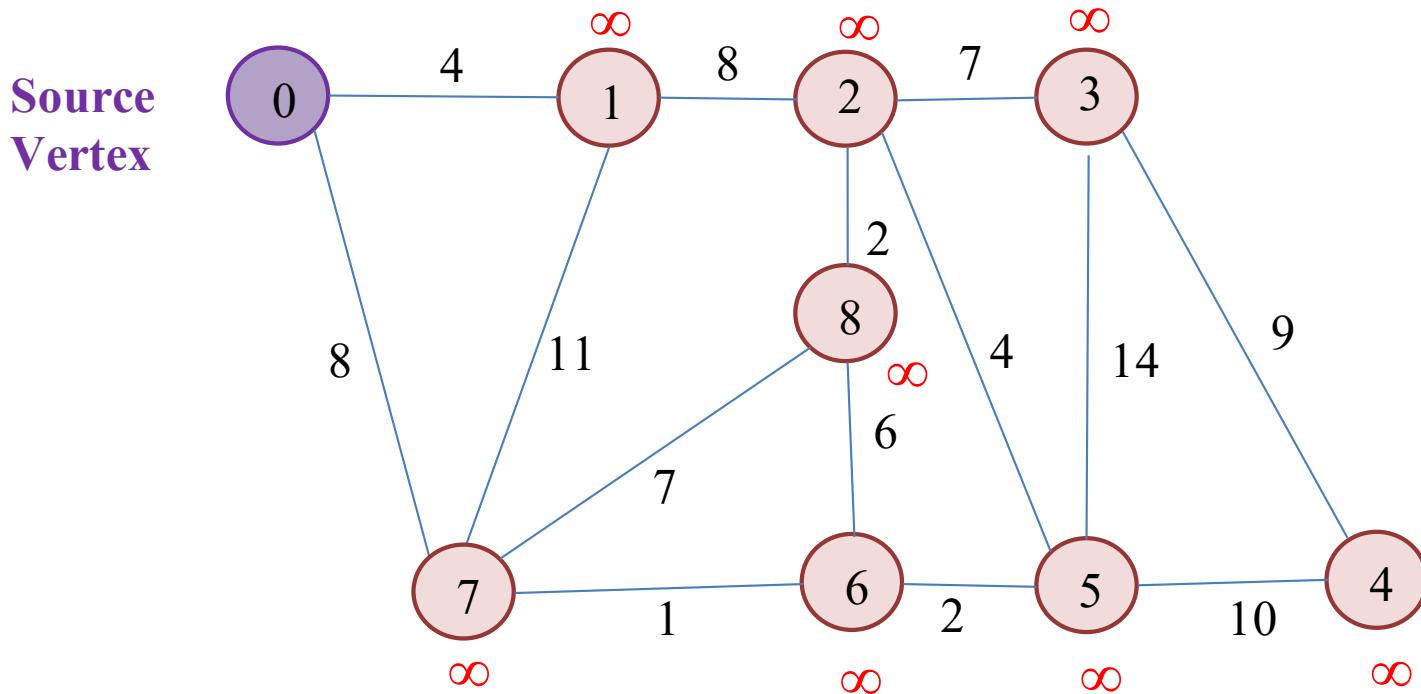
1. **Setup:** Represent the graph as set of nodes using matrix.
2. **Initialization:** make a framework for keeping track of the shortest paths at each iteration, the sum of weights for these shortest paths, visited and unvisited nodes, and initialize the algorithm based on the starting node.
3. **Update:** update the shortest path tracking information based on the relationship between the current visited node and the starting node.
4. **Repeat:** repeat step 3 until all nodes have been visited

Example 1. Dijkstra's Algorithm

Find the shortest paths from source to all vertices in the given undirected graph.

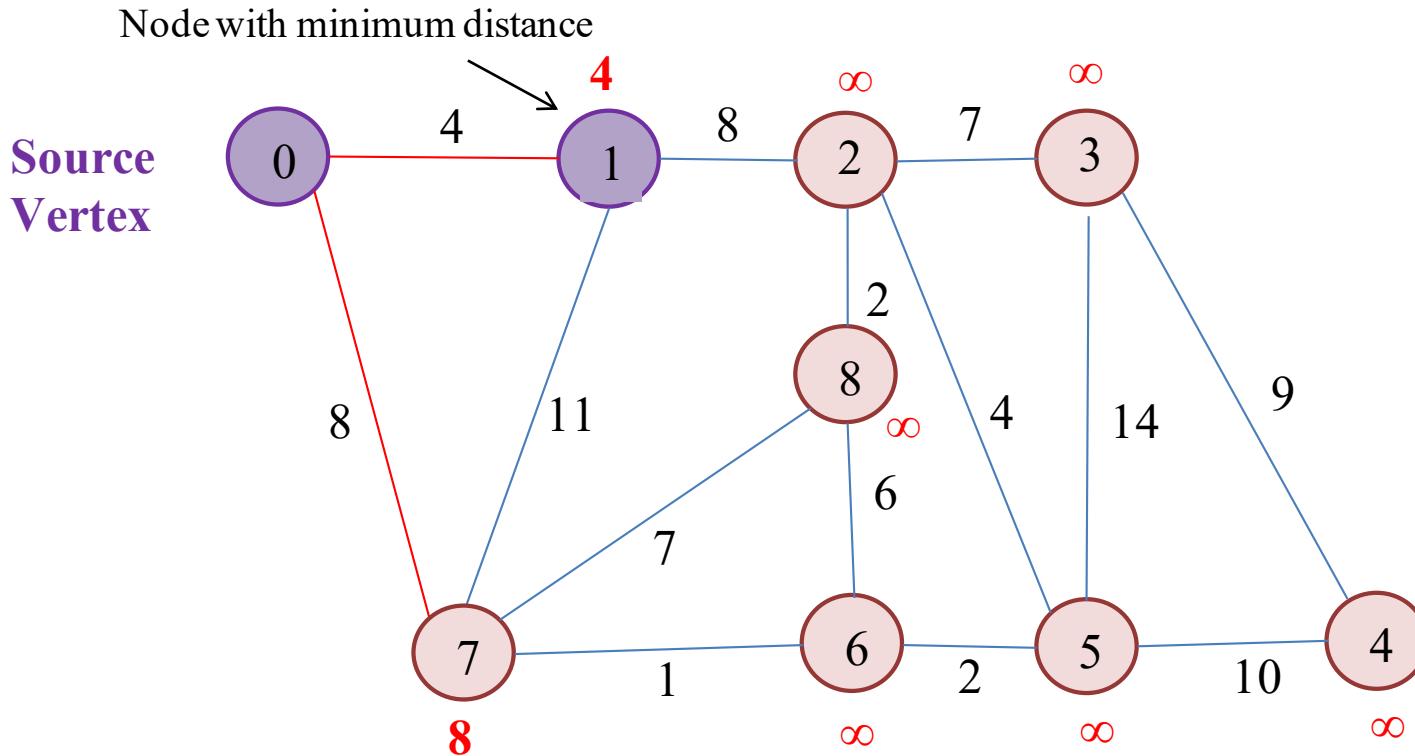


Example 1 Solution. Dijkstra's Algorithm



Create a set that can keep track of vertices whose minimum distance from source is calculated. The set is initially empty and the distances assigned to vertices { 0, INF, INF, INF, INF, INF, INF, INF } where 0 is for the source and infinity for the rest of nodes.

Example 1 Solution. Dijkstra's Algorithm



From all nodes except 0, pick the node with smallest distance from source which has not already been picked (node 1 with distance of 4). Now the set is {0, 1}. (purple nodes). Update the distance according to a formula between two vertex u and v:

If $(\text{distance of vertex } u + \text{weight of vertex } u \text{ to } v) < \text{distance of vertex } v$

Then $\text{distance of vertex } v = \text{distance of vertex } u + \text{weight of vertex } u \text{ to vertex } v$

Example 1 Solution. Dijkstra's Algorithm

If this true: $d(u) + w(u,v) < d(v)$
Then update: $d(v) = d(u) + w(u,v)$

Vertex 1 is connected to 2 and 7. Check these:

$$d(v1) + w(v1, v7) < d(v7)$$

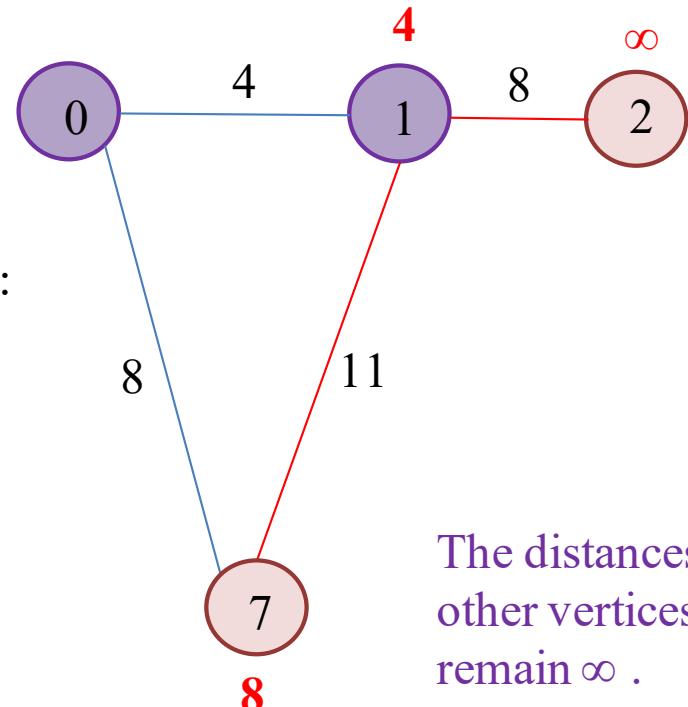
$$4 + 11 < 8$$

False, distance of vertex 7 is not updated.

$$d(v1) + w(v1, v2) < d(v2)$$

$$4 + 8 < \infty$$

True, distance of vertex 2 is updated to 12.



The distances of other vertices remain ∞ .

From all vertices except **0, 1** (which were already picked), pick the node with smallest distance from source (node 7 with distance of 8). Now the set is **{0, 1, 7}**. Update the distance according to a formula (next slide).

Example 1 Solution. Dijkstra's Algorithm

Vertex 7 is connected to 1, 8 and 6 (no need to check connection to source).

Check these:

$$d(v7) + w(v7, v1) < d(v1)$$

$$8 + 11 < 4$$

False, distance of vertex 1 is not updated.

$$d(v7) + w(v7, v8) < d(v8)$$

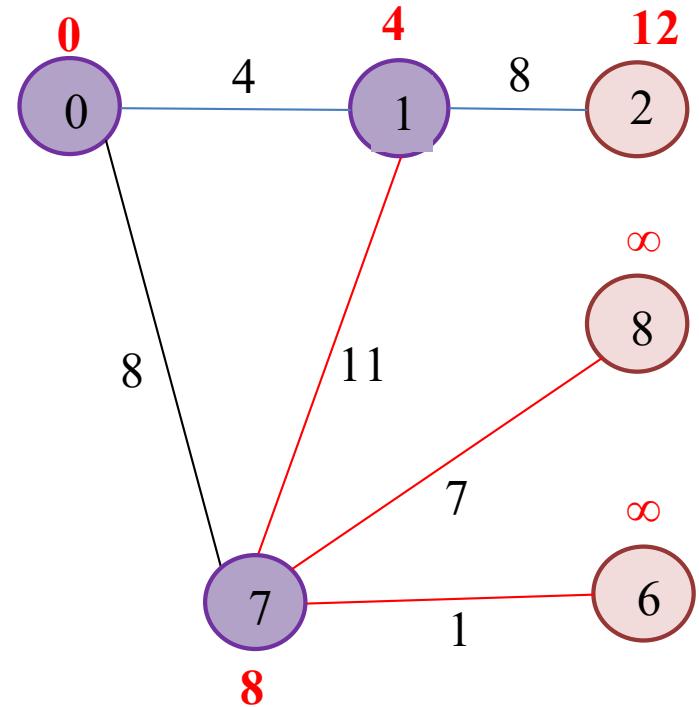
$$8 + 7 < \infty$$

True, distance of vertex 8 is updated 15.

$$d(v7) + w(v7, v6) < d(v6)$$

$$8 + 1 < \infty$$

True, distance of vertex 6 is updated to 9.



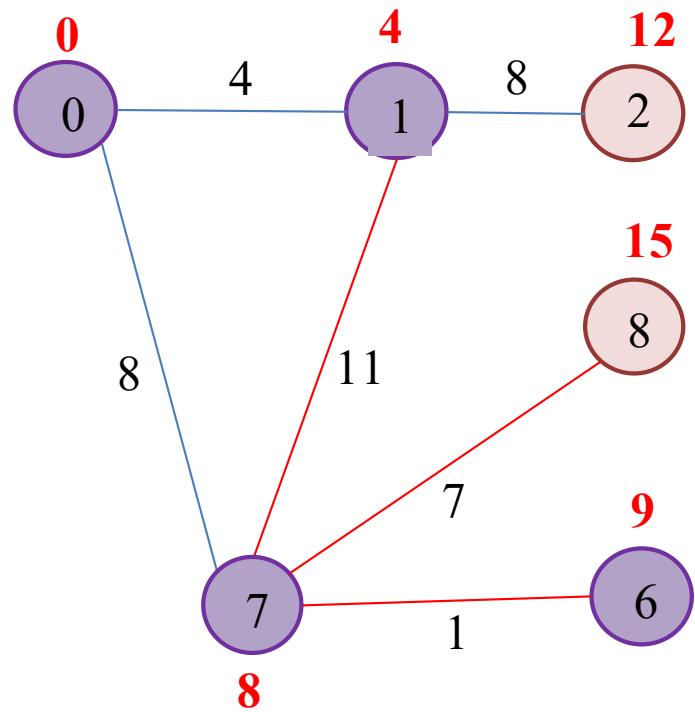
Then expect nodes 0, 1, 7, pick the node with minimum distance, which would be node 6 with distance of 9. Then set will change to: {0, 1, 7, 6}.

Example 1 Solution. Dijkstra's Algorithm

Then set has changed to: {**0, 1, 7, 6**}

The distance of nodes are updated.

Now we check the connected nodes to node 6.



Example 1 Solution. Dijkstra's Algorithm

Vertex 6 is connected to vertices 7, 8 and 5. Check these:

$$d(v_6) + w(v_6, v_7) < d(v_7)$$

$$9 + 1 < 8$$

False, distance of vertex 7 is not updated.

$$d(v_6) + w(v_6, v_8) < d(v_8)$$

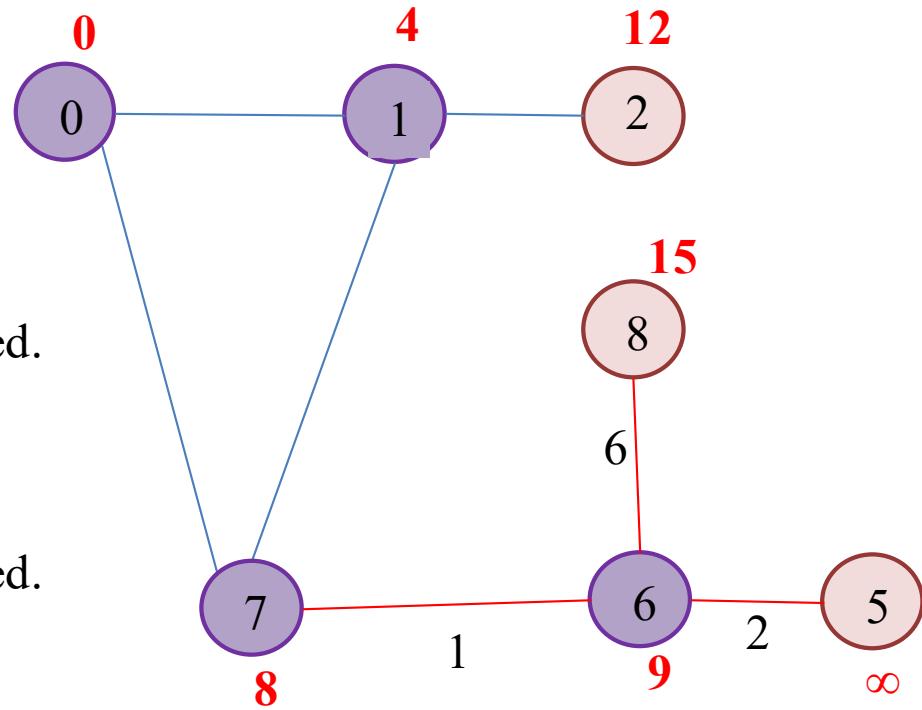
$$9 + 6 < 15$$

False, distance of vertex 8 is not updated.

$$d(v_6) + w(v_6, v_5) < d(v_5)$$

$$9 + 2 < \infty$$

True, distance of vertex 5 is updated to 11.



Then expect nodes 0, 1, 7, 6 pick the node with minimum distance, which would be node 5 with distance of 11. Then set will change to: {0, 1, 7, 6, 5}

Now we check the connected nodes to node 5.

Example 1 Solution. Dijkstra's Algorithm

Vertex 5 is connected to vertices 2,3,4,6 and .Check these:

$$d(v5) + w(v5, v2) < d(v2)$$

$$11 + 4 < 12$$

False, distance of vertex 2 is not updated.

$$d(v5) + w(v5, v3) < d(v3)$$

$$11 + 14 < \infty$$

True, distance of vertex 3 is updated to 25.

$$d(v5) + w(v5, v4) < d(v4)$$

$$11 + 10 < \infty$$

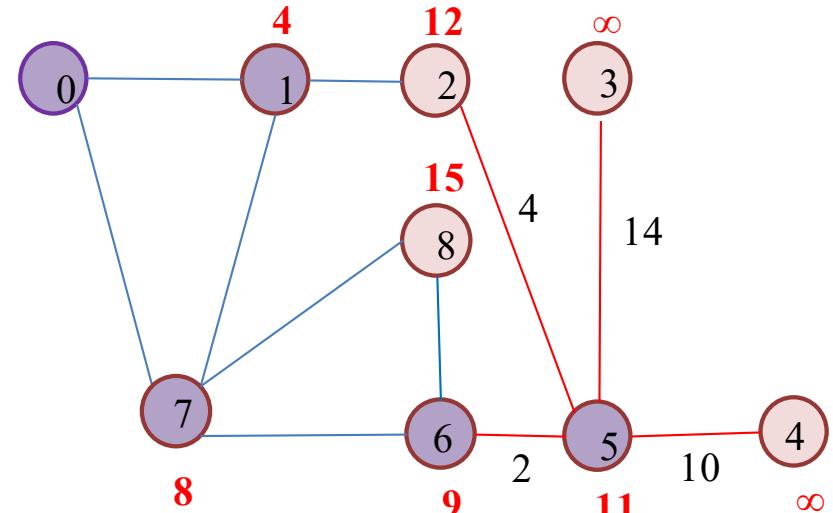
True, distance of vertex 4 is updated to 21.

$$d(v5) + w(v5, v6) < d(v6)$$

$$11 + 2 < 9$$

False, distance of vertex 6 is not updated.

Then expect nodes 0, 1, 7, 6, 5 pick the node with minimum distance, which would be node 2 with distance of 12. Then set will change to: {0, 1, 7, 6, 5, 2}



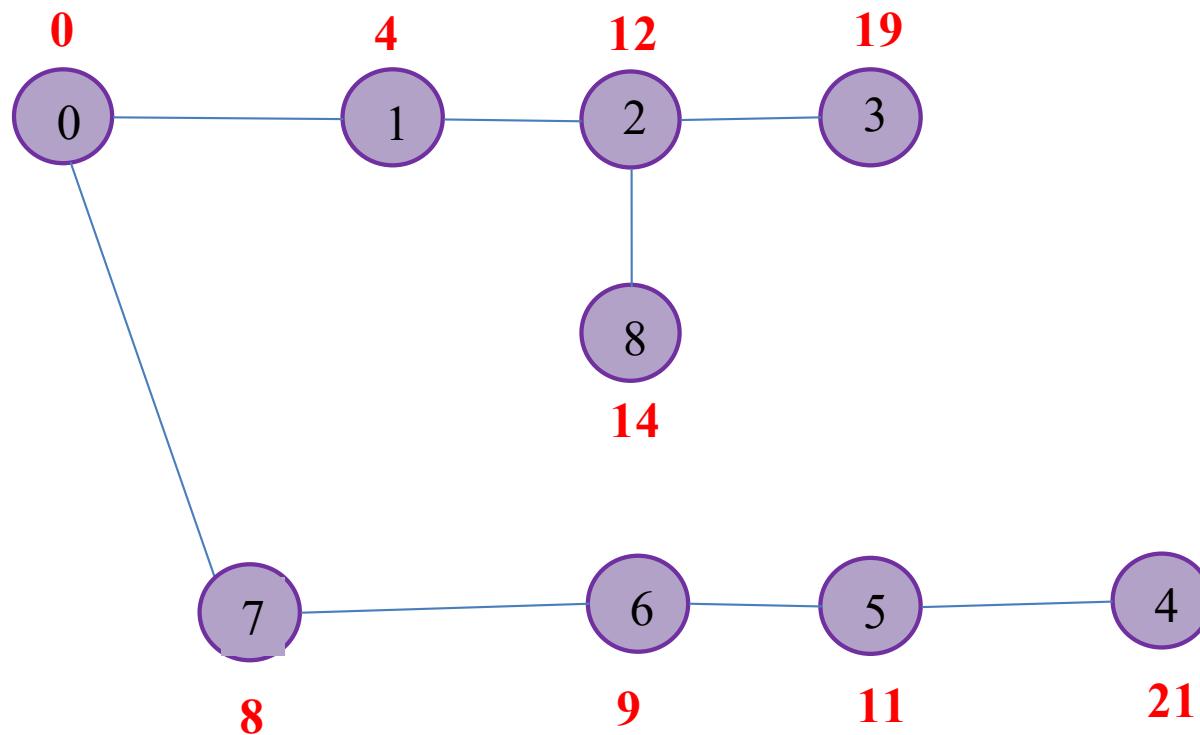
Example 1 Solution. Dijkstra's Algorithm

Let's see what happened to distance value of each vertex in every iteration:

V	1	2	3	4	5	6	7	8
Initial values	∞							
Iteration 1	1	4	∞	∞	∞	∞	∞	8
Iteration 2	7	4	12	∞	∞	∞	9	8
Iteration 3	6	4	12	∞	∞	11	9	8
Iteration 4	2	4	12	∞	∞	11	9	8

Example 1 Solution. Dijkstra's Algorithm

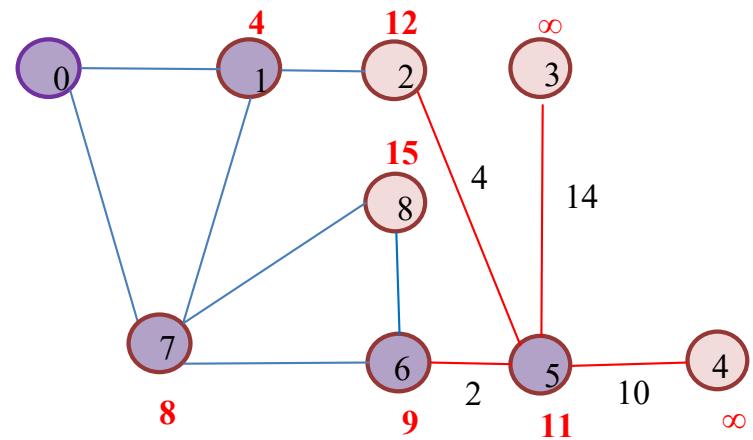
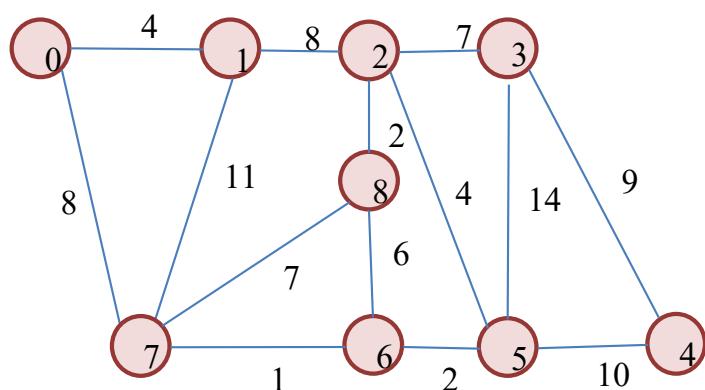
Repeat the steps until the set does include all vertices of the graph.
Finally the following will be the shortest path



You Try 2. Dijkstra's Algorithm



Show how the distance values of vertices will be updated in 5th iteration for the graph shown in previous examples.



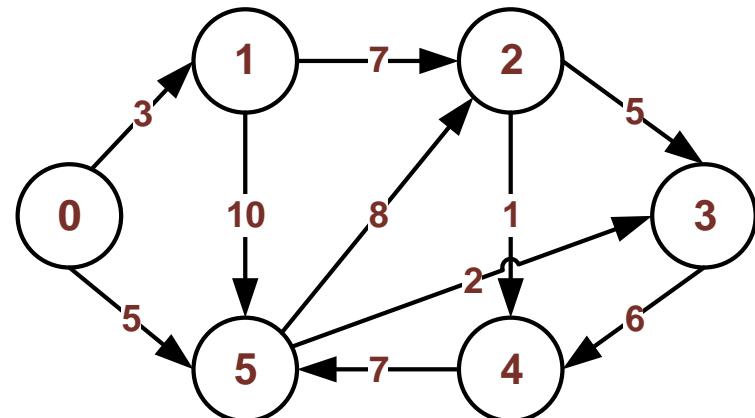
You Try 3. Dijkstra's Algorithm



The procedures of Dijkstra's algorithm for **directed graph** is similar to the steps for the undirected graph.

Just note that for instance node 2 connect to node 3 with weight of 5, but node 2 is not reachable from node 3.

Apply the procedures of Dijkstra's algorithm for the graph shown in the picture.



	0	1	2	3	4	5
0	0	3	∞	∞	∞	5
1	∞	0	7	∞	∞	10
2	∞	∞	0	5	1	∞
3	∞	∞	∞	0	6	∞
4	∞	∞	∞	∞	0	7
5	∞	∞	8	2	∞	0

Drawbacks of Dijkstra's Algorithm

- 1) It does blind search so wastes lot of time while processing.
- 2) It cannot handle negative edges.
- 3) This leads to acyclic graphs and most often cannot obtain the right shortest path.

Next Lecture

We focus on:

- Sorting: Bubble Sort and Selection Sort

Readings/ Study Reference

Course Textbook: “*Data Structure and Algorithms in a Nutshell*” A. Wong et al.

Chapter 8. Graphs

Section 8.6. Finding Shortest Path

The End of Lecture

Course: Data Structures and Algorithms

Instructor: Homeyra Pourmohammadali

You Try Questions and Solutions

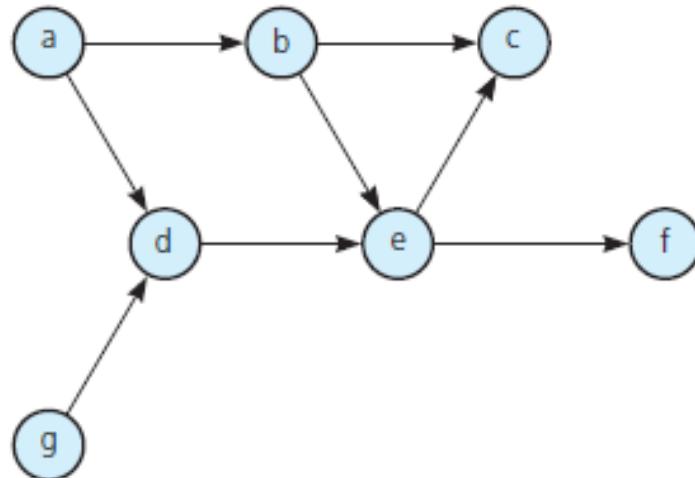
Graph Applications:

Topological Sorting & Finding Shortest Path

You Try 1. Topological Sorting Algorithm



In a table show a trace of topSort2 (in previous slide) for this graph. Show the content of stack **S** (bottom to top) and **aList** (beginning to end) after each push or pop operation.



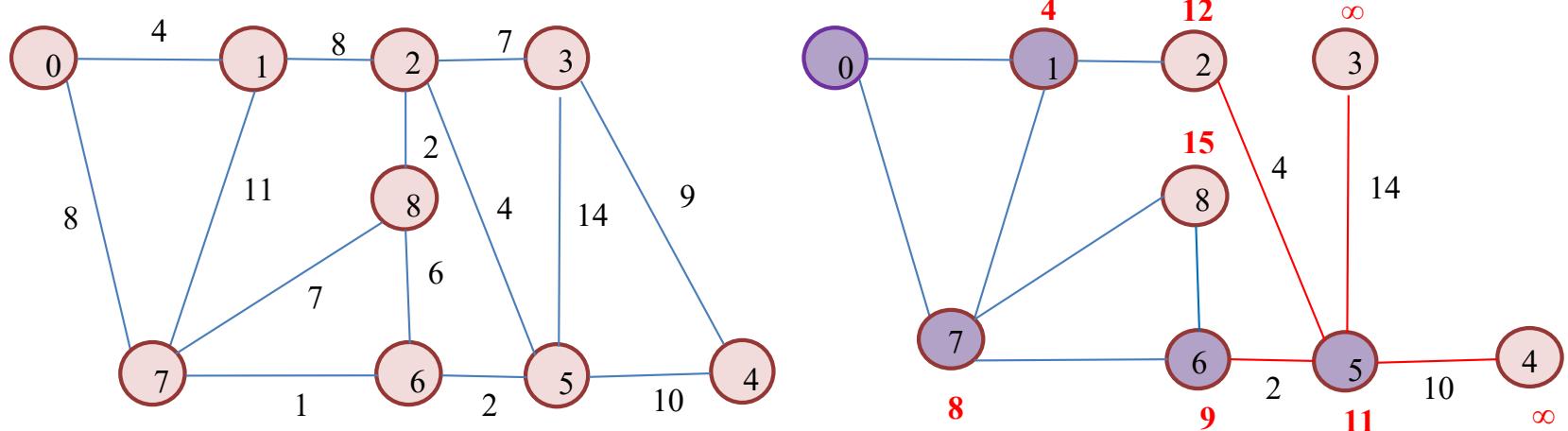
You Try 1 Solution. Topological Sorting Algorithm

Action	Stack s (bottom to top)	List aList (beginning to end)
Push a	a	
Push g	a g	
Push d	a g d	
Push e	a g d e	c
Push c	a g d e c	c
Pop c, add c to aList	a g d e	f c
Push f	a g d e f	e f c
Pop f, add f to aList	a g d e	d e f c
Pop e, add e to aList	a g d	g d e f c
Pop d, add d to aList	a g	g d e f c
Pop g, add g to aList	a	b g d e f c
Push b	a b	a b g d e f c
Pop b, add b to aList	a	
Pop a, add a to aList	(empty)	

You Try 2. Dijkstra's Algorithm



Show how the distance values of vertices will be updated in 5th iteration for the graph shown in previous example.



You Try 2 Solution. Dijkstra's Algorithm

Vertex 2 is connected to vertices 1,3,5, and 8 .Check these:

$$d(v2) + w(v2, v1) < d(v1)$$

$$12 + 8 < 4$$

False, distance of vertex 1 is not updated.

$$d(v2) + w(v2, v3) < d(v3)$$

$$12 + 7 < 25$$

True, distance of vertex 3 is updated to 19.

$$d(v2) + w(v2, v5) < d(v5)$$

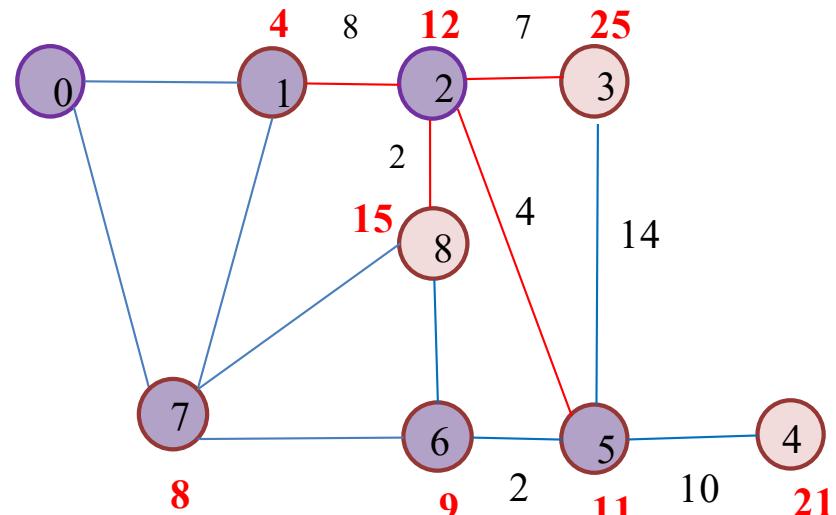
$$12 + 4 < 11$$

False, distance of vertex 5 is not updated.

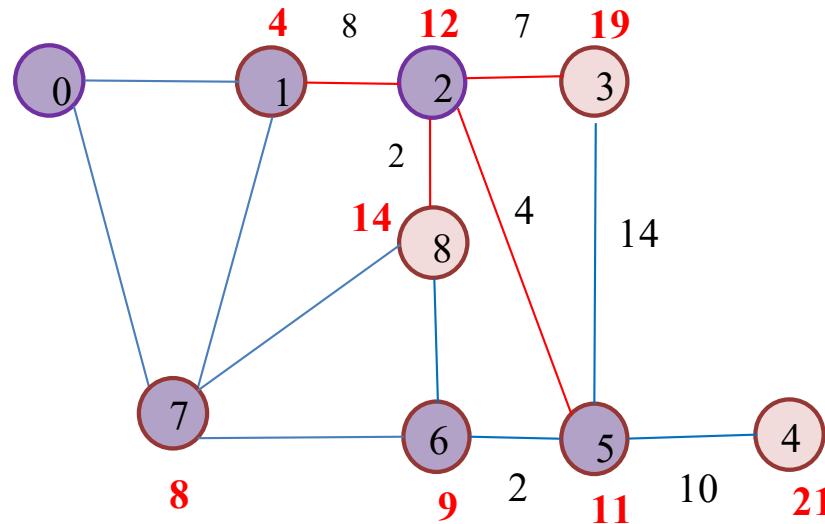
$$d(v2) + w(v2, v8) < d(v8)$$

$$12 + 2 < 15$$

True, distance of vertex 8 is updated to 14.



You Try 2 Solution. Dijkstra's Algorithm



Then expect nodes 0, 1, 7, 6, 5, 2 pick the node with minimum distance, which would be node 8 with distance of 14.

Then, for the next iteration, the set will change to:

{**0, 1, 7, 6, 5, 2, 8**}

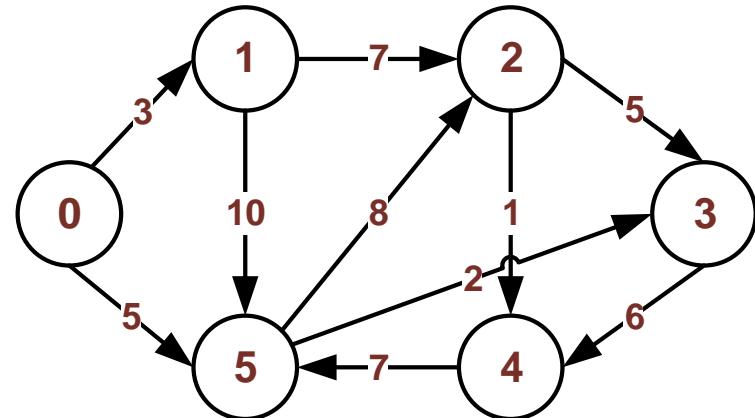
You Try 3. Dijkstra's Algorithm



The procedures of Dijkstra's algorithm for **directed graph** is similar to the steps for the undirected graph.

Just note that for instance node 2 connect to node 3 with weight of 5, but node 2 is not reachable from node 3.

Apply the procedures of Dijkstra's algorithm for the graph shown in the picture.



	0	1	2	3	4	5
0	0	3	∞	∞	∞	5
1	∞	0	7	∞	∞	10
2	∞	∞	0	5	1	∞
3	∞	∞	∞	0	6	∞
4	∞	∞	∞	∞	0	7
5	∞	∞	8	2	∞	0

You Try 3 Solution. Dijkstra's Algorithm

Full solution is available in section 8.6.1 to 8.6.5 of the course textbook.

Note that the source vertex in textbook is 1 but in this example is vertex zero, but directions and weights of edges are the same.

The End of You Try Activities

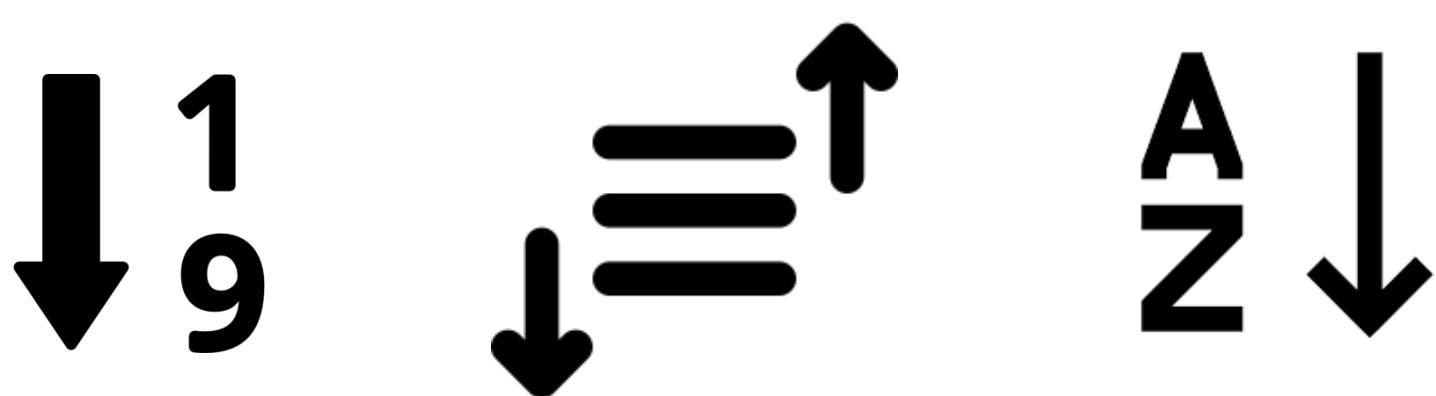
Course: Data Structures and Algorithms

Instructor: Homeyra Pourmohammadali

Sorting: Bubble Sort and Selection Sort

Motivation

- How do you write an algorithm that keep lists of elements in **sorted order**?
- What is the common goal of keeping sorted lists?
- How costly is the sorting operation?



Learning Outcomes

By the end of this lecture you will be able to:

- design and implement the Bubble Sort and Selection Sort algorithms.
- calculate the efficiency of these algorithms in terms of Big-O complexity.

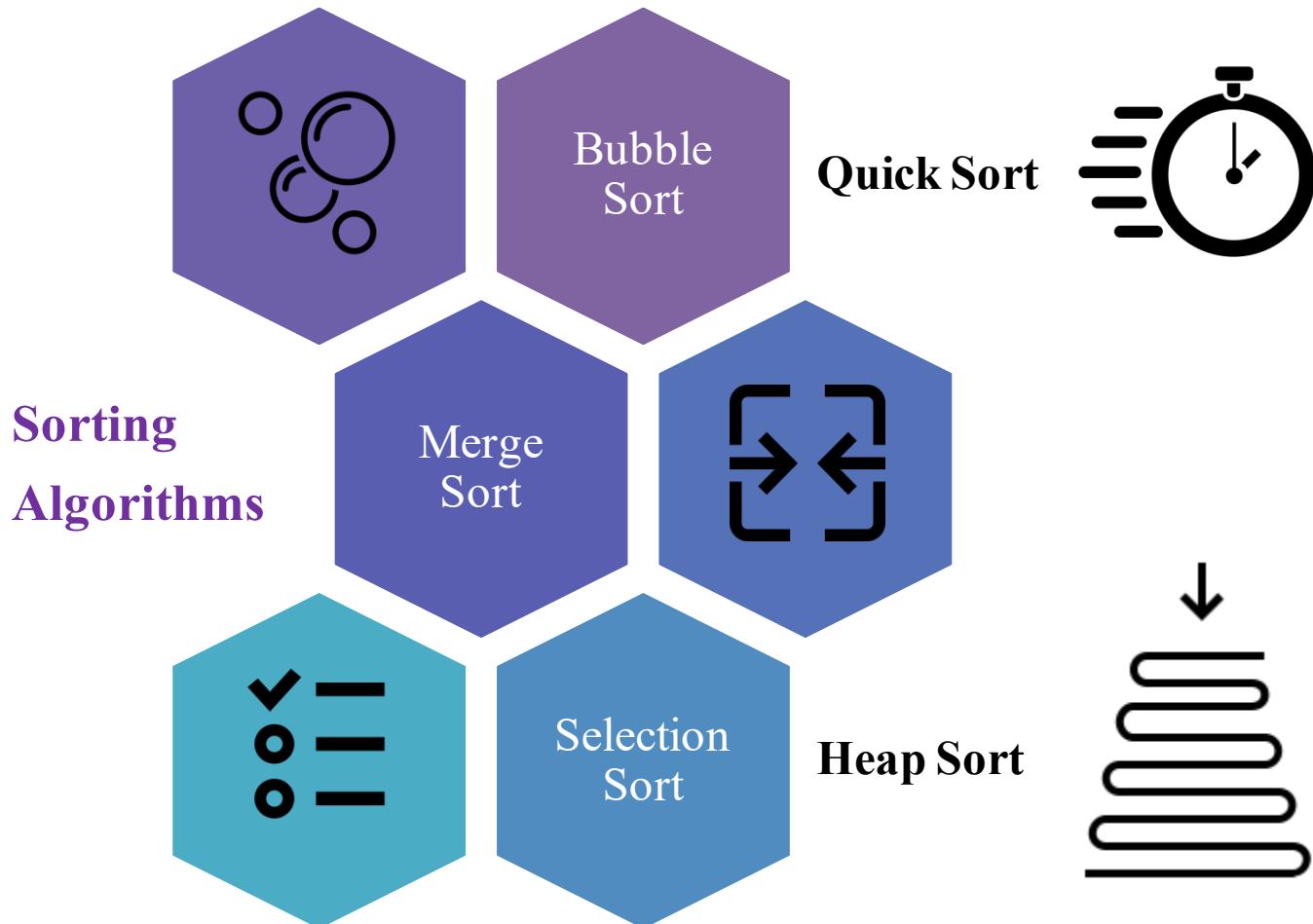
Introduction to Sorting

- Sorting is a process that organizes a collection of data into either ascending or descending order.
- The need for sorting arises in many situations.
- You may simply want to sort a collection of data before including it in a report.
- You must perform a sort as an initialization step for certain algorithms.

Example 1. Application of Sorting

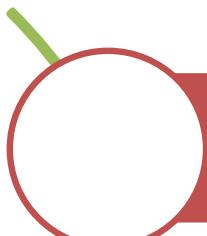
- Searching for data is one of the most common tasks performed by computers.
- When the collection of data to be searched is large, an efficient technique for searching—such as the binary search algorithm—is desirable.
- The binary search algorithm requires that the data be sorted. Thus, sorting the data is a step that must precede a binary search on a collection of data that is not already sorted.
- Good sorting algorithms, therefore, are quite valuable.

Different Sorting Algorithms

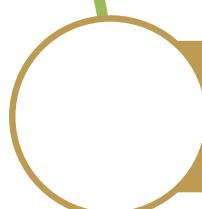


Bubble Sort

The bubble sort compares adjacent items and exchanges them if they are out of order. This sort usually requires several passes over the data.



During the first pass, compare the first two items in the array. Exchange them, if they are out of order.



Compare the items in the next pair—that is, in positions 2 and 3 of the array. Exchange them, if they are out of order.



Proceed by comparing and exchanging items two at a time, until you reach the end of the array.

Example 2. Bubble Sort. Ascending Order

Trace the bubble sort as it sorts the following array into ascending order, for two passes.

29	10	14	37	13
----	----	----	----	----

Example 2 Solution. Bubble Sort. Ascending Order

29	10	14	37	13
----	----	----	----	----

10	29	14	37	13
----	----	----	----	----

10	14	29	37	13
----	----	----	----	----

10	14	29	37	13
----	----	----	----	----

10	14	29	13	37
----	----	----	----	----

The first pass of a bubble sort of an array

You compare the items in the first pair (29 & 10) and exchange them because they are out of order.

Next you consider the second pair (29 & 14) and exchange these items because they are out of order.

The items in the third pair (29 & 37) are in order, and so you do not exchange them.

Finally, you exchange the items in the last pair (37 & 13).

Although the array is not sorted after the first pass, the largest item has “bubbled” to its proper position at the end of the array.

Example 2 Solution. Bubble Sort. Ascending Order

During the second pass of the bubble sort, you return to the beginning of the array and consider pairs of items in exactly the same manner as the first pass.

				
10	14	29	13	37

				
10	14	29	13	37

				
10	14	29	13	37

10	14	13	29	37
----	----	----	----	----

The second pass of a bubble sort of an array

The second pass considers the first $n - 1$ items of the array.

After the second pass, the second-largest item in the array will be in its proper place in the next-to-last position of the array.

Now, ignoring the last two items, which are in order, you continue with subsequent passes until the array is sorted.

Fewer passes might be possible to sort a particular array. You could terminate the process if no exchanges occur during any pass.

Example 2. Bubble Sort. Descending Order

Trace the bubble sort as it sorts the following array into descending order, for three passes.

29	10	14	37	13
----	----	----	----	----

Example 3. Bubble Sort. Descending Order

The procedure is the same, just sort from biggest to smallest.

The first pass

29	10	14	37	13

The second pass

29	14	37	13	10

The third pass

29	37	14	13	10

29	10	14	37	13

29	14	37	13	10

37	29	14	13	10

29	14	10	37	13

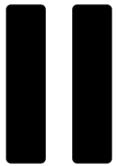
29	37	14	13	10

37	29	14	13	10

29	14	37	10	13

29	37	14	13	10

29	14	37	13	10



You Try 1. Bubble Sort

A) Trace the bubble sort as it sorts this array into ascending order.

25	30	20	80	40	60
----	----	----	----	----	----

B) Repeat the bubble sort as it sorts the same array into descending order.

C) Show three passes for both cases.

Bubble Sort Efficiency Analysis

- The bubble sort requires at most $n - 1$ passes through the array.
- Pass 1 requires $n - 1$ comparisons and at most $n - 1$ exchanges, pass 2 requires $n - 2$ comparisons and at most $n - 2$ exchanges. In general, pass i requires $n - i$ comparisons and at most $n - i$ exchanges.
- In the worst case, a bubble sort will require a total of

$$(n - 1) + (n - 2) + \dots + 1 = n \partial (n - 1) / 2$$

comparisons and the same number of exchanges.

Bubble Sort Efficiency Analysis

- Altogether, there are:

$$2 \partial n \partial (n - 1) = 2n^2 - 2n$$

- Therefore, the bubble sort algorithm is $O(n^2)$ in the worst case.
- The best case occurs when the original data is already sorted: Bubble Sort uses one pass, during which $n - 1$ comparisons and no exchanges occur.
- Thus, the bubble sort is $O(n)$ in the best case.

Implementation of The Bubble Sort

- At the implementation level, sorting algorithms may use various internal data structures, including an array, a linked list, a binary tree, and so on.
- For simplicity, we assume that we have an array and are reorganizing its values so they are in order by key.
- The number of values to be sorted and the array in which they are stored are parameters to our sorting algorithms.

Example 4: An Implementation of The Bubble Sort

```
//Sorts the items in an array into “ascending order”.
void bubbleSort(ItemType theArray[], int n)
{
    bool sorted = false; // False when swaps occur
    int pass = 1;
    while (!sorted && (pass < n))
    {
        // At this point, theArray[n+1-pass..n-1] is sorted
        // and all of its entries are > the entries in theArray[0..n-pass]
        sorted = true; // Assume sorted
        for ( int index = 0; index < n - pass; index++)
        {
            // At this point, all entries in theArray[0..index-1] are <= theArray[index]
            int nextIndex = index + 1;
            if (theArray[index] > theArray[nextIndex])
            {
                std::swap(theArray[index], theArray[nextIndex]); // Exchange entries
                sorted = false; // Signal exchange
            } // end if
        } // end for
        pass++; // Assertion: theArray[0..n-pass-1] < theArray[n-pass]
    } // end while
} // end bubbleSort
```

Selection Sort

Imagine some data that you can examine all at once. To sort it, you could select the largest item and put it in its place, select the next largest and put it in its place, and so on.

For a card player, this process is analogous to looking at an entire hand of cards and ordering it by selecting cards one at a time in their proper order.



<https://www.youtube.com/watch?v=voZuZ06Aet8>

Selection Sort

- To sort an array into **ascending order**, using selection sort, you first **search** it for the **largest item**.
- Because you want the **largest item** to be in the **last position** of the array, swap the last item with the largest item, even if these items happen to be identical.
- Now, ignoring the last (and largest) item of the array, search the rest of the array for its largest item and swap it with its last item, which is the next-to-last item in the original array.
- Continue until you have selected and swapped $n - 1$ of the n items in the array. The remaining item, is in its proper order, so it is not considered further.

Selection Sort: Alternative Approach

- To sort an array into **ascending order**, using selection sort, you can also first **search** it for the **smallest item**.
- Because you want the **smallest** item to be in the **first position** of the array, swap the **first** item with the **smallest** item, even if these items happen to be identical.
- Now, ignoring the first item of the array, search the rest of the array for its smallest item and swap it with its last item, which is the next-to-last item in the original array.
- Continue until you have selected and swapped $n - 1$ of the n items in the array. The remaining item, is in its proper order, so it is not considered further.

Example 5. Selection Sort. Ascending Order

Trace the selection sort as it sorts the following array into ascending order.

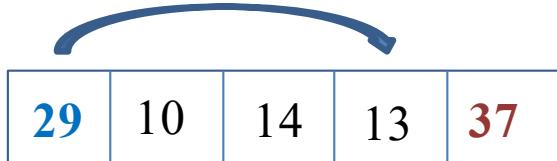
29	10	14	37	13
----	----	----	----	----

Example 5 Solution. Selection Sort. Ascending Order



29	10	14	37	13
----	----	----	----	----

Initial array (select the largest element (37))



29	10	14	13	37
----	----	----	----	----

After first swap (37 is in the last position, and second largest element is selected (29))



13	10	14	29	37
----	----	----	----	----

After second swap (29 is positioned just before 37, and third largest element is selected (14)).



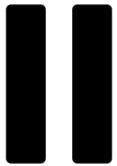
13	10	14	29	37
----	----	----	----	----

After third swap (14 is positioned just before 29, and fourth largest element is selected (13)).



10	13	14	29	37
----	----	----	----	----

After fourth swap (10 is positioned just before 14, and fifth and last largest element is selected (10)).



You Try 2. Selection Sort

- A) Trace the selection sort as it sorts this array into ascending order.

25	30	20	80	40	60
----	----	----	----	----	----

- B) Repeat the selection sort as it sorts the same array into descending order.

Example 6: Implementation of The Selection Sort

```
// Finds the largest item in an array.  
int findIndexofLargest(const ItemType theArray[], int size);  
  
// Sorts the items in an array into ascending order.  
void selectionSort(ItemType theArray[], int n)  
{  
    // last = index of the last item in the subarray of items yet to be sorted;  
    // largest = index of the largest item found  
    for (int last = n - 1; last >= 1; last--)  
    {  
        // At this point, theArray[last+1..n-1] is sorted, and its  
        // entries are greater than those in theArray[0..last].  
        // Select the largest entry in theArray[0..last]  
  
        int largest = findIndexofLargest(theArray, last+1);  
        // Swap the largest entry, theArray[largest], with  
        // theArray[last]  
  
        std::swap(theArray[largest], theArray[last]);  
    } // end for  
} // end selectionSort
```

Example 6: Implementation of The Selection Sort

```
int findIndexofLargest(const ItemType theArray[], int size)
{
    int indexSoFar = 0; // Index of largest entry found so far

    for ( int currentIndex = 1; currentIndex < size; currentIndex++)
    {
        // At this point, theArray[indexSoFar] >= all entries in
        // theArray[0..currentIndex - 1]

        if (theArray[currentIndex] > theArray[indexSoFar])
            indexSoFar = currentIndex;
    } // end for

    return indexSoFar; // Index of largest entry
} // end findIndexofLargest
```

Selection Sort Efficiency Analysis

- The `for` loop in the function `selectionSort` executes $n - 1$ times.
- The `selectionSort` calls each of the functions `findIndexofLargest` and `swap` $n - 1$ times.
- Each call to `findIndexofLargest` causes its loop to execute `last` times (that is, `size - 1` times when `size` is `last + 1`).
- The $n - 1$ calls to `findIndexofLargest`, for values of `last` that range from $n - 1$ down to 1, cause the loop in `findIndexofLargest` to execute a total of $(n - 1) + (n - 2) + \dots + 1 = n \partial (n - 1) / 2$ times.
- Because each execution of `findIndexofLargest`'s loop performs one comparison, the calls to `findIndexofLargest` require $n \partial (n - 1) / 2$ comparisons.

Selection Sort Efficiency Analysis

- The $n - 1$ calls to swap result in $n - 1$ exchanges. Each exchange requires three assignments, or data moves. Thus, the calls to swap require $3 \partial (n - 1)$ moves.
- Together, a selection sort of n items requires
$$n \partial (n - 1)/2 + 3 \partial (n - 1) = n^2/2 + 5 \partial n /2 - 3$$
- Ignore low-order terms and the multiplier. Thus, the selection sort is $O(n^2)$.

Comparing Efficiencies

- Both of the bubble sort and selection sort algorithm have the same runtime complexity of $O(n^2)$.
- In general, selection sort algorithm is more efficient than bubble sort, as it requires only one swap per iteration.
- Among the sorting algorithms, the bubble sort is the most inefficient algorithms available.

Next Lecture

We focus on:

- Merge Sort
- Quick Sort

Readings/ Study Reference

Course Textbook: “*Data Structure and Algorithms in a Nutshell*” A. Wong et al.

Chapter 9. Sorting

Section 9.1. Bubble Sort

Section 9.2. Selection Sort

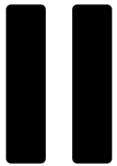
The End of Lecture

Course: Data Structures and Algorithms

Instructor: Homeyra Pourmohammadali

You Try Question and Solution

Sorting: Bubble Sort and Selection Sort



You Try 1. Bubble Sort

A) Trace the bubble sort as it sorts this array into ascending order.

25	30	20	80	40	60
----	----	----	----	----	----

B) Repeat the bubble sort as it sorts the same array into descending order.

C) Show three passes for both cases.

You Try 1 Solution. Bubble Sort. Ascending Order

The first pass

25	30	20	80	40	60
----	----	----	----	----	----

25	20	30	40	60	80
----	----	----	----	----	----

The third pass

20	25	30	40	60	80
----	----	----	----	----	----

25	30	20	80	40	60
----	----	----	----	----	----

20	25	30	40	60	80
----	----	----	----	----	----

20	25	30	40	60	80
----	----	----	----	----	----

25	20	30	80	40	60
----	----	----	----	----	----

20	25	30	40	60	80
----	----	----	----	----	----

20	25	30	40	60	80
----	----	----	----	----	----

25	20	30	80	40	60
----	----	----	----	----	----

20	25	30	40	60	80
----	----	----	----	----	----

20	25	30	40	60	80
----	----	----	----	----	----

25	20	30	40	80	60
----	----	----	----	----	----

20	25	30	40	60	80
----	----	----	----	----	----

25	20	30	40	60	80
----	----	----	----	----	----

You Try 1 Solution. Bubble Sort. Descending Order

The first pass

25	30	20	80	40	60
----	----	----	----	----	----

30	25	20	80	40	60
----	----	----	----	----	----

30	25	20	80	40	60
----	----	----	----	----	----

30	25	80	20	40	60
----	----	----	----	----	----

30	25	80	40	20	60
----	----	----	----	----	----

30	25	80	40	60	20
----	----	----	----	----	----

The second pass

30	25	80	40	60	20
----	----	----	----	----	----

30	25	80	40	60	20
----	----	----	----	----	----

30	80	25	40	60	20
----	----	----	----	----	----

30	80	40	25	60	20
----	----	----	----	----	----

30	80	40	60	25	20
----	----	----	----	----	----

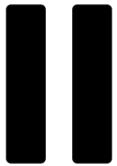
The third pass

30	80	40	60	25	20
----	----	----	----	----	----

80	30	40	60	25	20
----	----	----	----	----	----

80	40	30	60	25	20
----	----	----	----	----	----

80	40	60	30	25	20
----	----	----	----	----	----



You Try 2. Selection Sort

- A) Trace the selection sort as it sorts this array into ascending order.

25	30	20	80	40	60
----	----	----	----	----	----

- B) Repeat the selection sort as it sorts the same array into descending order.

You Try 2 Solution. Selection Sort

Ascending order

25	30	20	80	40	60
----	----	----	----	----	----



25	30	20	60	40	80
----	----	----	----	----	----



25	30	20	40	60	80
----	----	----	----	----	----



25	30	20	40	60	80
----	----	----	----	----	----

25	20	30	40	60	80
----	----	----	----	----	----

20	25	30	40	60	80
----	----	----	----	----	----

Descending order

25	30	20	80	40	60
----	----	----	----	----	----



25	30	60	80	40	20
----	----	----	----	----	----



40	30	60	80	25	20
----	----	----	----	----	----



40	80	60	30	25	20
----	----	----	----	----	----



60	80	40	30	25	20
----	----	----	----	----	----



80	60	40	30	25	20
----	----	----	----	----	----

The End of You Try Activities

Course: Data Structures and Algorithms

Instructor: Homeyra Pourmohammadali

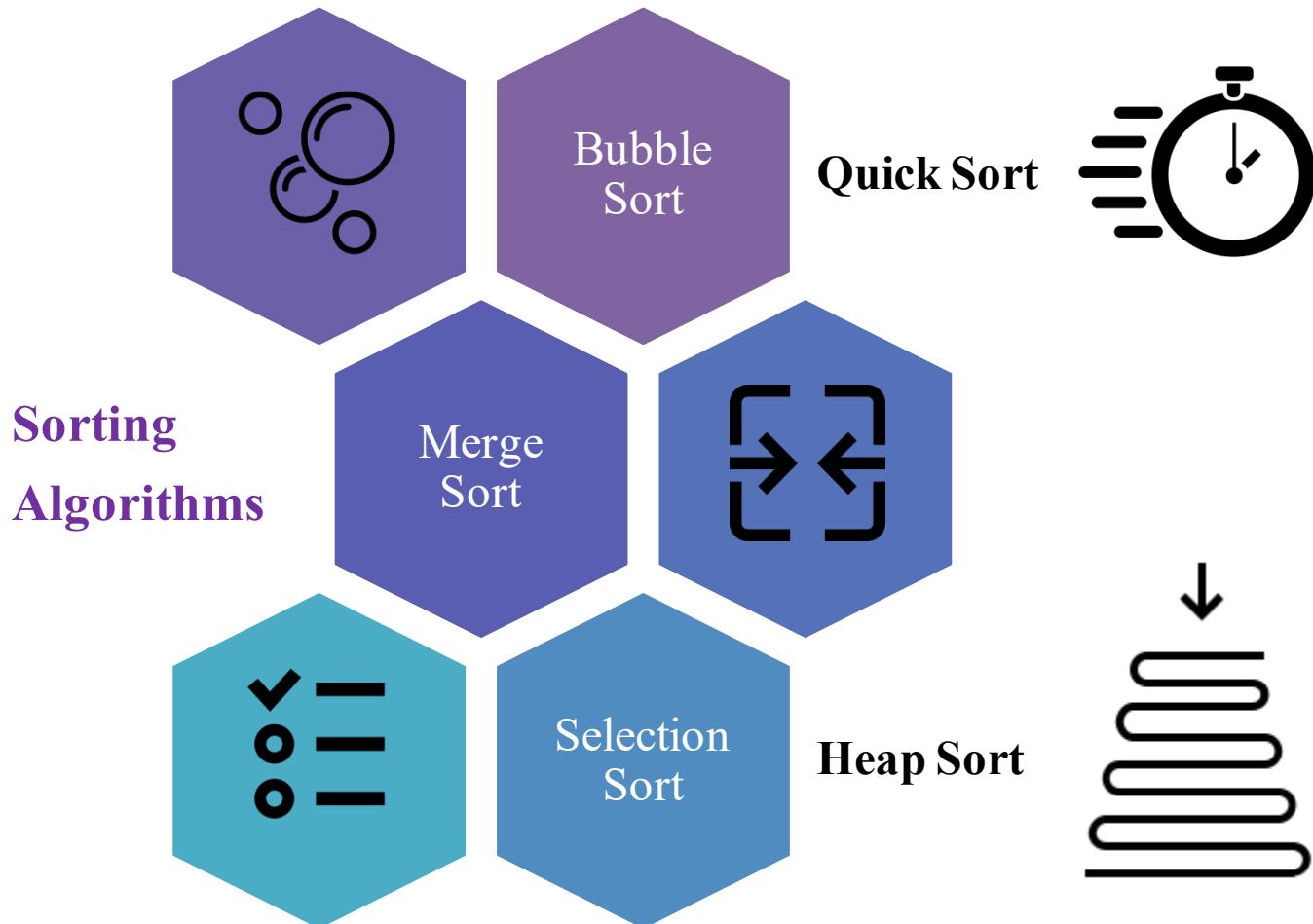
Sorting: Merge Sort and Quick Sort

Learning Outcomes

By the end of this lecture you will be able to:

- design and implement the Merge Sort and Quick Sort algorithms.
- calculate the efficiency of these algorithms in terms of Big-O complexity.

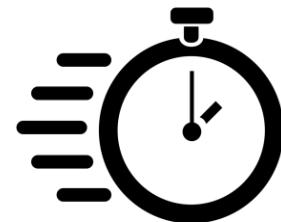
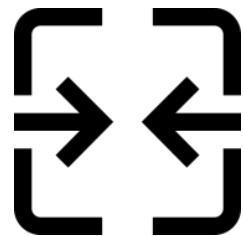
Different Sorting Algorithms



Merge Sort

Merge Sort

- Two important divide-and-conquer sorting algorithms, **merge sort** and **quick sort**, have elegant recursive formulations and are highly efficient.
- The presentations here are in the context of sorting arrays.
- The merge sort is a recursive sorting algorithm that always gives the same performance, regardless of the initial order of the array items.



Merge Sort Procedure

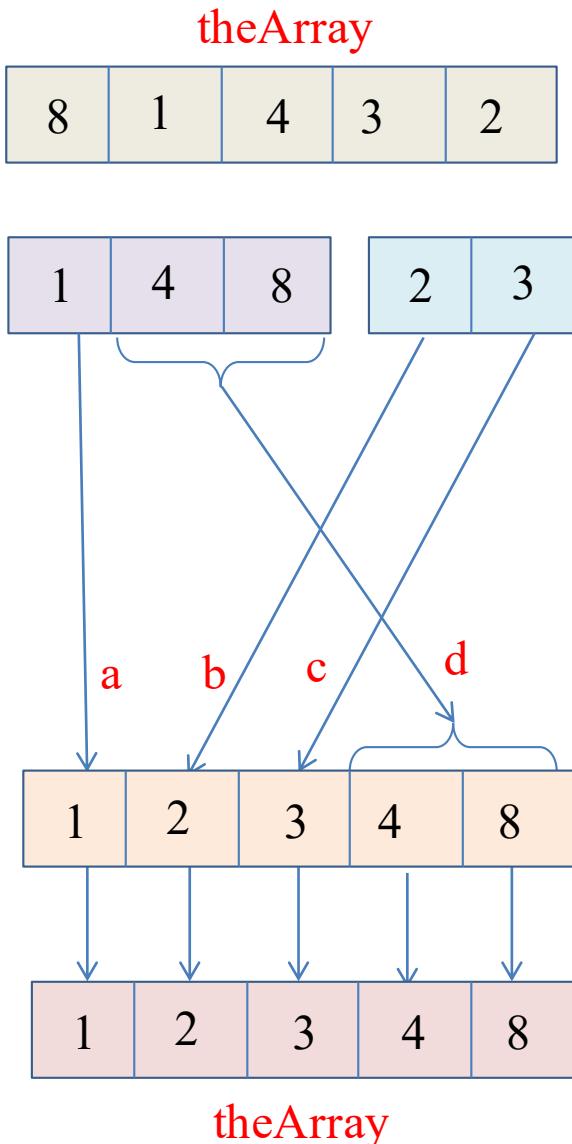
1. Divide the array into halves.
2. Sort each half.
3. Merge the sorted halves into one sorted array. This merge step compares an item in one half of the array with an item in the other half and moves the smaller item to a temporary array. This process continues until there are no more items to consider in one half. At that time, you simply move the remaining items to the temporary array.
4. Finally, you copy the temporary array back into the original array.

Example 1. Merge Sort. Ascending Order

Sort the following array using Merge sort algorithm.

8	1	4	3	2
---	---	---	---	---

Example 1. Merge Sort. Ascending Order



1. Divide the array in half

2. Sort each half

Note: We need to use an auxiliary temporary array

3. Merge the halves:

- 1 < 2, move 1 from left half to tempArray
- 4 > 2, move 2 from right half to tempArray
- 4 > 3, move 3 from right half to tempArray
- Right half is finished, so move rest of left half to tempArray

4. Copy temporary array back into original array

Merge Sort Algorithm

- Halve the array, recursively sort its halves, and then merge the halves.
- The merge sort requires a second array as large as the original array
- Although the merge step of the merge sort produces a sorted array, how do you sort the array halves prior to the merge step?
- The merge sort sorts the array halves by using a merge sort—that is, by calling itself recursively.

Merge Sort Pseudocode

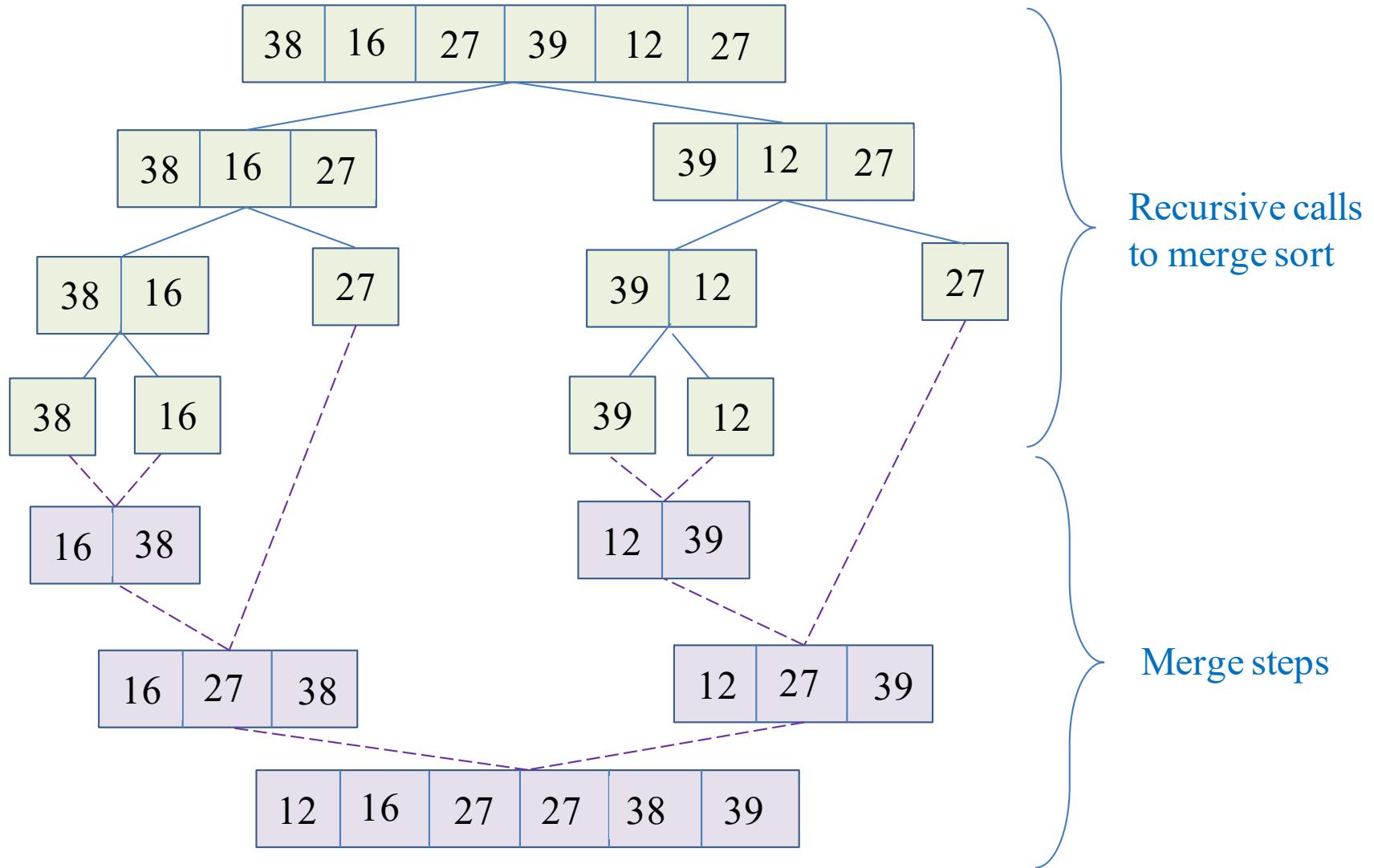
```
// Sorts theArray[first..last] by
// 1. Sorting the first half of the array
// 2. Sorting the second half of the array
// 3. Merging the two sorted halves
mergeSort(theArray: ItemArray, first: integer, last: integer)
  if (first < last)
  {
    mid = (first + last) / 2 // Get midpoint
    // Sort theArray[first..mid]
    mergeSort(theArray, first, mid)
    // Sort theArray[mid+1..last]
    mergeSort(theArray, mid + 1, last)
    // Merge sorted halves theArray[first..mid] and theArray[mid+1..last]
    merge(theArray, first, mid, last)
  }
  // If first >= last, there is nothing to do
```

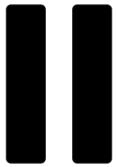
Example 2. Merge Sort. Trace of Algorithm

Show trace of Merge Sort algorithm for the following array.

38	16	27	39	12	27
----	----	----	----	----	----

Example 2. Merge Sort. Trace of Algorithm





You Try 1. Merge Sort .Trace of Algorithm

Show trace of Merge Sort algorithm
for the following array.

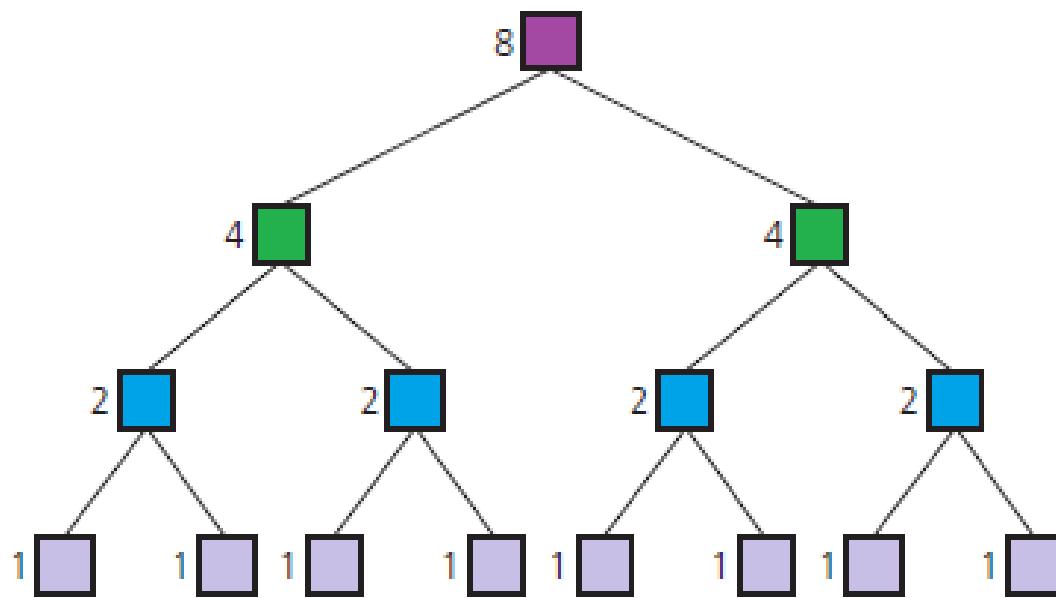
25	30	20	80	40	60
----	----	----	----	----	----

Merge Sort Efficiency Analysis

- If the total number of items in the two array segments to be merged is n , then merging the segments requires at most $n - 1$ comparisons.
- There are n moves from the original array to the temporary array, and n moves from the temporary array back to the original array.
- Thus, each merge step requires $3n - 1$ major operations.
- Each call to `mergeSort` recursively calls itself twice.

Merge Sort Efficiency Analysis

For example, levels of recursive calls to `mergeSort`, given an array of eight items are:



Level 0: mergesort 8 items

Level 1: 2 calls to mergesort with 4 items each

Level 2: 4 calls to mergesort with 2 items each

Level 3: 8 calls to mergesort with 1 item each

Merge Sort Efficiency Analysis

- The recursive calls continue until the array pieces each contain one item—that is, until there are n pieces, where n is the number of items in the original array.
- If n is a power of 2 ($n = 2^k$), the recursion goes $k = \log_2 n$ levels deep.
- For example, there are three levels of recursive calls to mergeSort because the original array contains eight items and $8 = 2^3$.
- If n is not a power of 2, there are $1 + \log_2 n$ (rounded down) levels of recursive calls to mergeSort.

Merge Sort Efficiency Analysis

Because there are either $\log_2 n$ or $1 + \log_2 n$ levels, the merge sort is $O(n \partial \log n)$ in both the worst and average cases.

Advantage of Merge Sort: extremely efficient algorithm with respect to time.

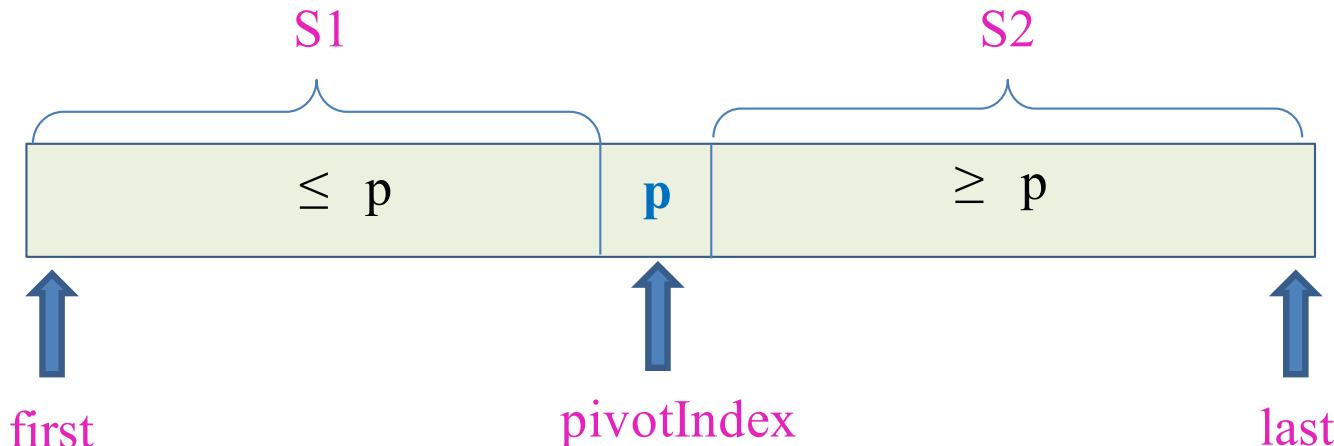
Drawback of Merge Sort: The merge step requires an auxiliary array that need extra storage and copying of entries.

Quick Sort

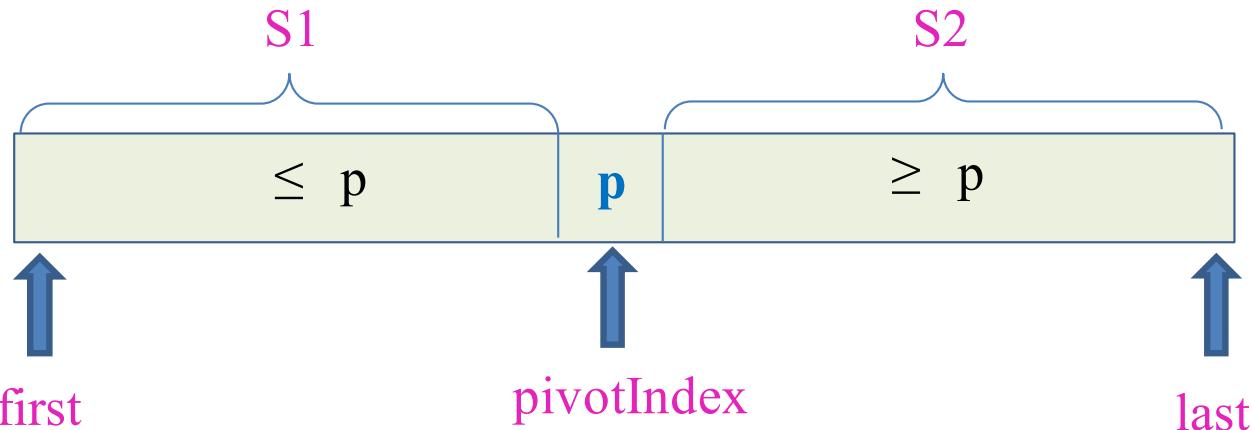
Quick Sort

Quick Sort is another **divide-and conquer** algorithm.

- Divide the array (list) into two parts based on a pivot.
- Put all entries less than the pivot (p) to one side ($S1$) and all of the entries larger than the pivot on the other ($S2$).



Quick Sort

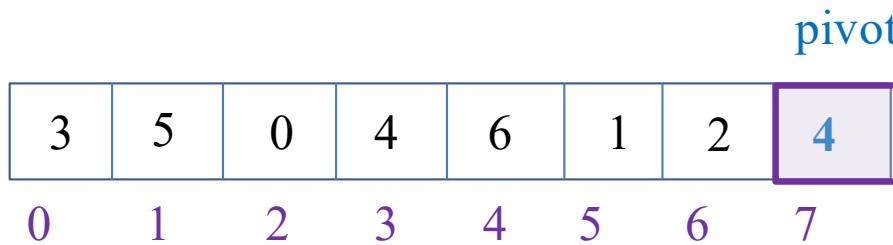


- The partition induces relationships among the array items that are the ingredients of a **recursive solution**.
- The left and right sorting problems are indeed **smaller problems** and are each closer than the original sorting problem to the **base case**—which is an array containing one item—because the pivot is not part of either $S1$ or $S2$.

Quick Sort Simple Pseudocode

```
// Sorts theArray[first..last].  
quickSort(theArray: ItemArray, first: integer, last: integer): void  
  if (first < last)  
  {  
    Choose a pivot item p from theArray[first..last]  
    Partition the items of theArray[first..last] about p  
  
    // The partition is theArray[first..pivotIndex..last]  
  
    quickSort(theArray, first, pivotIndex - 1)  // Sort S1  
    quickSort(theArray, pivotIndex + 1, last)   // Sort S2  
  }  
  // If first >= last, there is nothing to do
```

Example 3. Quick Sort. Partitioning



- Choose a pivot (4), swap it with the last element of array.

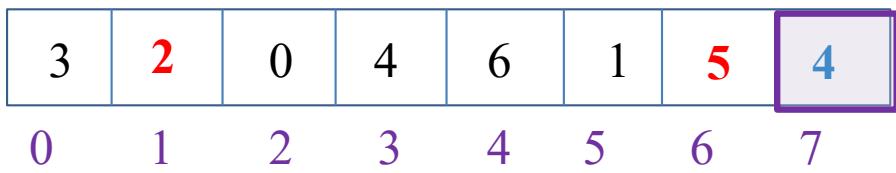
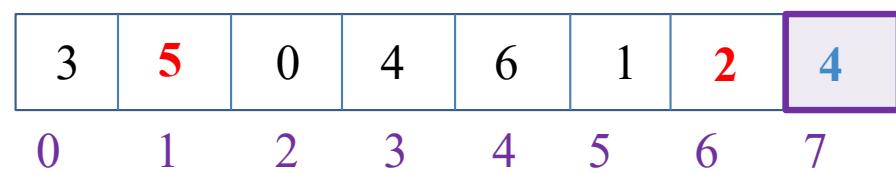
- Start at the beginning, move from **left to right**, look for the first entry that is greater than or equal to the pivot

(5, at indexFromLeft=1)

- Start from next-to-last entry and move from **right to left**, look for the first entry that is less than or equal to the pivot

(2, at indexFromRight=6)

- If **indexFromLeft < indexFromRight** ($1 < 6$), then **swap** the two entries (5 and 2) at those indices.



Example 3. Quick Sort . Partitioning

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

pivot

- Continue the searches from the left and from the right.
- The search from the left stops at 4 and from the right stops at 1.
- `indexFromLeft < indexFromRight` ($3 < 5$), `swap` 4 and 1.
- Continue the searches again. The search from the left stops at 6, while from the right goes beyond the 6 and stops at 1.
- `indexFromLeft > indexFromRight` ($4 > 3$), `no swap` is necessary and the search ended.
- Place the pivot between the subarrays S_1 and S_2 by swapping.

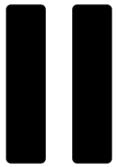
3	2	0	1	6	4	5	4
0	1	2	3	4	5	6	7

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

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

3	2	0	1	4	4	5	6
S1				pivot	S2		

You Try 2. Quick Sort. Partitioning



Could you swap pivot with the first entry of the array instead of the last entry?

What if the Entries Are Equal to Pivot?

- Both of the subarrays S_1 and S_2 can contain entries equal to the pivot. You can place any entries that equal the pivot into the same subarray.
- This makes one subarray larger than the other. To enhance the quick sort's performance, we want the subarrays to be as nearly equal in size as possible.
- Both the search from the left and right stop when they encounter an entry that equals the pivot.
- This means that rather than leaving such entries in place, they are swapped. Such an entry has a chance of landing in each of the subarrays.

How to Select the Pivot Element?

- Ideally, the pivot should be the **median** value in the array, so that S_1 and S_2 each have almost the same number of entries.
- Instead of getting the best pivot by finding the median of all values in the array, we will at least try to avoid a bad pivot.
- We will take as our pivot the **median of three entries** in the array: the **first** entry, the **middle** entry, and the **last** entry. We sort only those three entries and use the middle entry of the three as the pivot.
- The choice of a pivot element will not impact the algorithm's ability to sort an array, but might influence the worst case runtime.

Pseudocode To Sort First, Middle, and Last Entries

For an array of at least four entries.

// Arranges the first, middle, and last entries in an array into ascending order.

sortFirstMiddleLast(theArray: ItemArray, first: integer, mid: integer, last: integer): void

if (theArray[first] > theArray[mid])
Interchange theArray[first] and theArray[mid]

if (theArray[mid] > theArray[last])
Interchange theArray[mid] and theArray[last]

if (theArray[first] > theArray[mid])
Interchange theArray[first] and theArray[mid]

Partitioning Algorithm

- The quick sort rearranges the entries in an array during the partitioning process.
- Each partition places one entry—the pivot—in its correct sorted position.
- The entries in each of the two subarrays that are before and after the pivot will remain in their respective subarrays.

Pseudocode of Partitioning Algorithm

// Partitions theArray[first..last].

partition(theArray: ItemArray, first: integer, last: integer): integer

// Choose pivot and reposition it

mid = first + (last - first) / 2

sortFirstMiddleLast(theArray, first, mid, last)

Interchange theArray[mid] and theArray[last – 1]

pivotIndex = last - 1

pivot = theArray[pivotIndex]

// Determine the regions S1 and S2

indexFromLeft = first + 1

indexFromRight = last - 2

Next page →

Pseudocode of Partitioning Algorithm

```
done = false
while (not done)
{
    // Locate first entry on left that is  $\geq$  pivot
    while (theArray[indexFromLeft] < pivot)
        indexFromLeft = indexFromLeft + 1
    // Locate first entry on right that is  $\leq$  pivot
    while (theArray[indexFromRight] > pivot)
        indexFromRight = indexFromRight - 1
    if (indexFromLeft < indexFromRight)
    {
        Move theArray[firstUnknown] into S1
        Interchange theArray[indexFromLeft] and theArray[indexFromRight]
        indexFromLeft = indexFromLeft + 1
        indexFromRight = indexFromRight - 1
    }
    else
        done = true
}
```

Next page →

Pseudocode of Partitioning Algorithm

// Place pivot in proper position between S 1 and S 2 , and mark its new location

Interchange theArray[pivotIndex] and theArray[indexFromLeft]

pivotIndex = indexFromLeft

return pivotIndex

Quick Sort Efficiency Analysis

- The major effort occurs during the partitioning step which required no more than n comparisons and it will be an $O(n)$ task.
- There are either $\log_2 n$ or $1 + \log_2 n$ levels of recursive calls to quickSort function.
- Each call to quickSort function involves m comparisons and at most m exchanges, where m is the number of items in the subarray to be sorted ($m \leq n - 1$).
- Thus, the average-case of a quick sort would be $O(n \log n)$.

Quick Sort Efficiency Analysis

- It is possible that each partition has one empty subarray. Although one recursive call will have nothing to do, the other call must sort $n - 1$ entries instead of $n / 2$.
- This occurrence is the worst case because the nonempty subarray decreases in size by only 1 at each recursive call to quickSort, and so the maximum number of recursive calls to quickSort will occur.
- The result is n levels of recursive calls instead of $\log n$. Thus, in the worst case, quick sort is $O(n^2)$.

Summary

- The selection sort, bubble sort are $O(n^2)$ algorithms; they are slow for large problems.
- The quick sort and merge sort are two very efficient recursive sorting algorithms.
- In the average case, the quick sort is among the fastest known sorting algorithms ($O(n \log n)$), but its worst-case behavior is significantly slower ($O(n^2)$) than the merge sort's (rarely occurs in practice).
- The merge sort requires extra storage equal to the size of the array to be sorted.

Next Lecture

We focus on:

- Heap Sort

Readings/ Study Reference

Course Textbook: “*Data Structure and Algorithms in a Nutshell*” A. Wong et al.

Chapter 9. Sorting

Section 9.3. Merge Sort

Section 9.4. Quick Sort

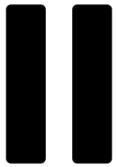
The End of Lecture

Course: Data Structures and Algorithms

Instructor: Homeyra Pourmohammadali

You Try Questions and Solutions

Sorting: Merge Sort and Quick Sort

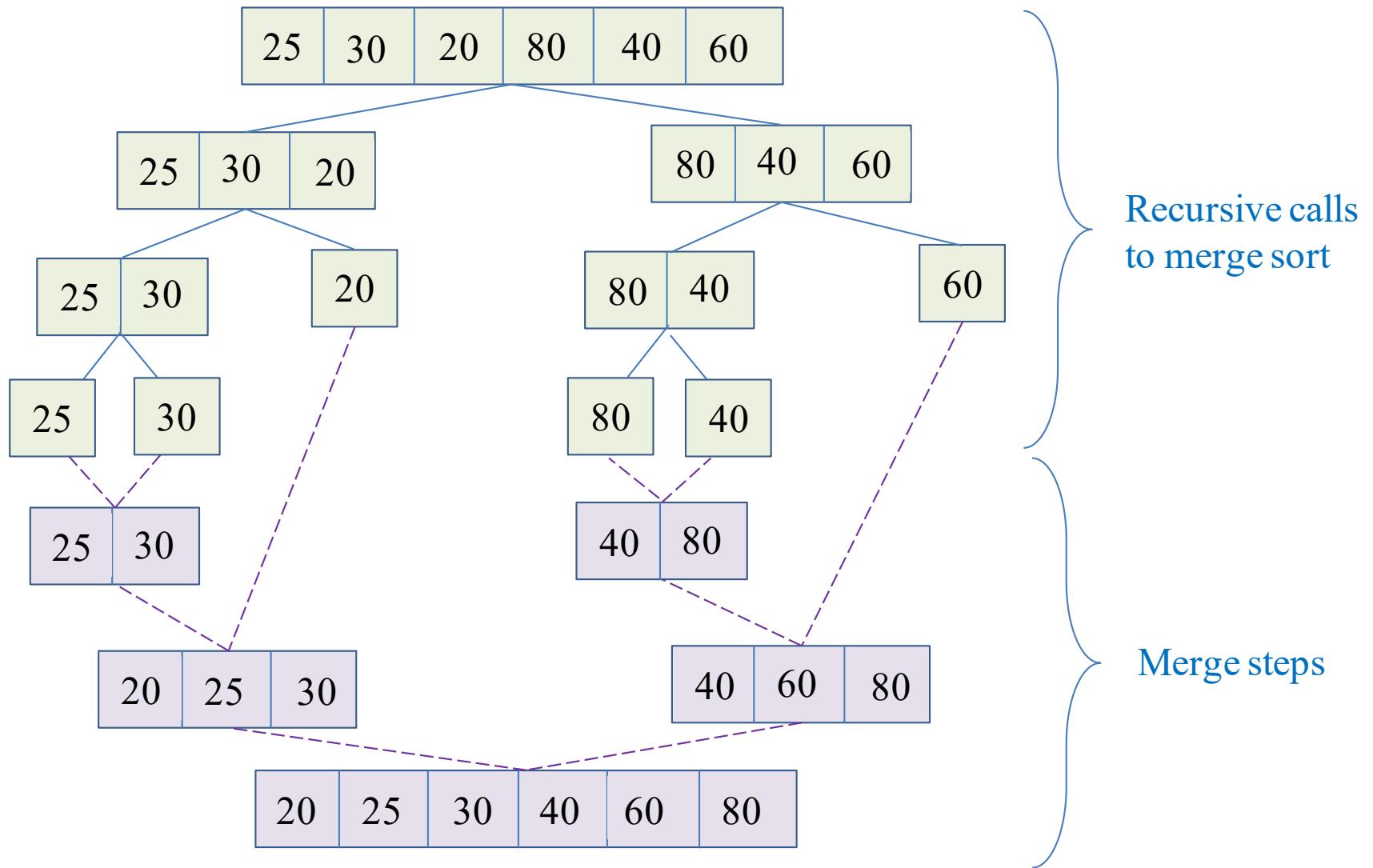


You Try 1. Merge Sort .Trace of Algorithm

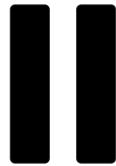
Show trace of Merge Sort algorithm
for the following array.

25	30	20	80	40	60
----	----	----	----	----	----

You Try 1 Solution. Merge Sort .Trace of Algorithm



You Try 2. Quick Sort. Partitioning



Could you swap pivot with the first entry of the array instead of the last entry?

Solution:

Yes, you can see the alternative example in your course textbook (Section 9.4, page 192-194).

The End of Lecture

Course: Data Structures and Algorithms

Instructor: Homeyra Pourmohammadali

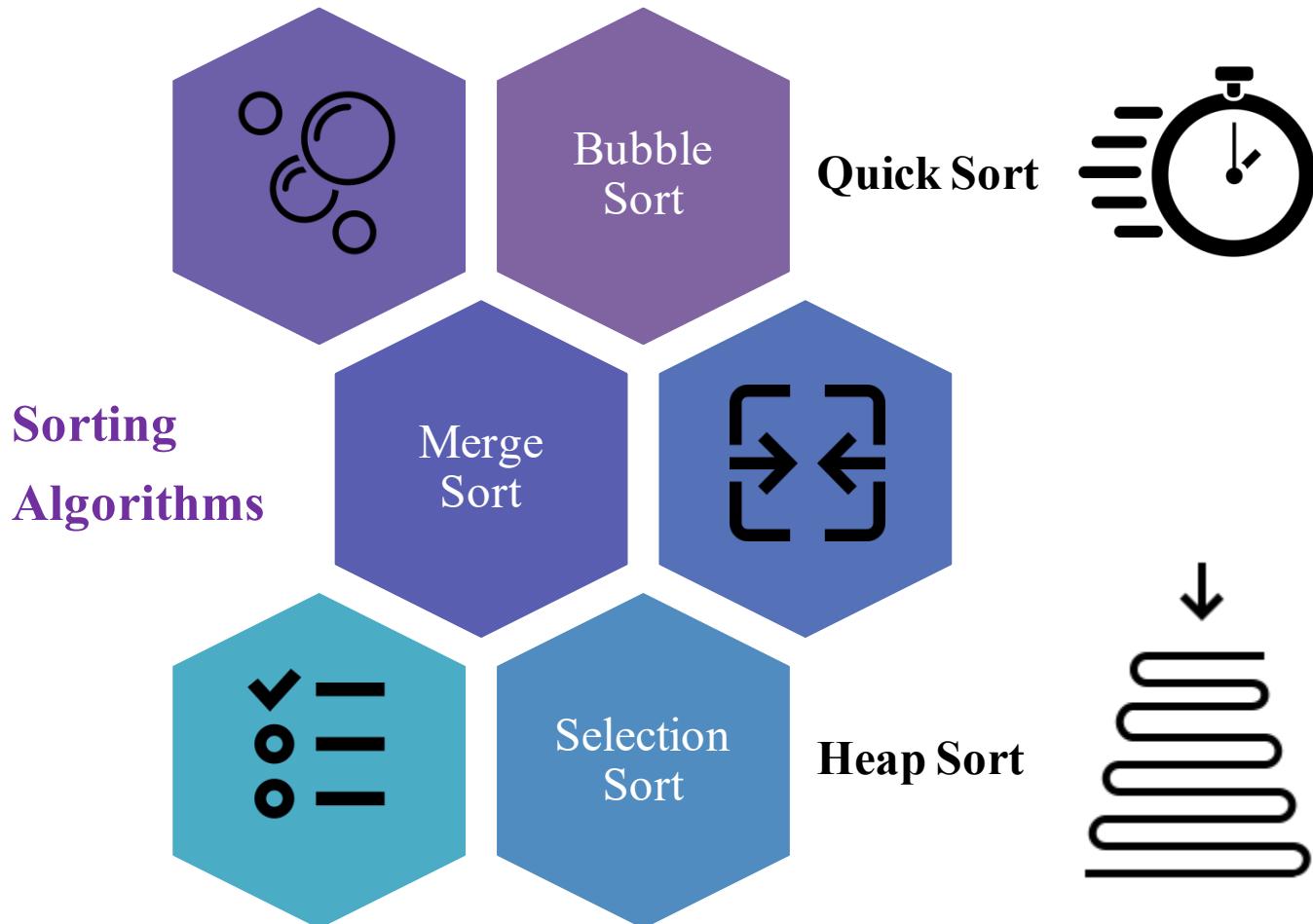
Sorting: Heap Sort

Learning Outcomes

By the end of this lecture you will be able to:

- design and implement the Heap Sort algorithm.
- compare the efficiency of this algorithm with other sorting algorithms.

Different Sorting Algorithms

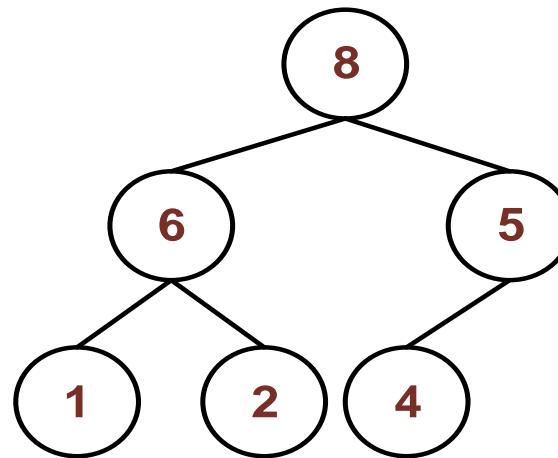


Heap

- Heap is a complete binary tree with largest value at the root.
- The root of a max heap contains the heap's largest value.
- Then, because of the order property of heaps, we always know where to find its greatest element and we can take advantage of this situation by using a heap to help us sort.
- The nodes of complete tree are contiguous if we scan the tree from left to right. This allows for efficient sequential representation (in an array).

Heap

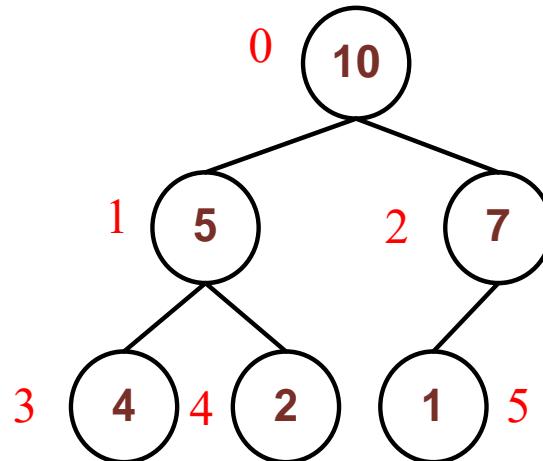
- A binary tree with keys stored in its nodes
- Parent key value \geq children key value
- Therefore the root node has the max value.
- A heap needs to be a complete binary tree.



Heap Example

Heap Array Representation

- Root node: index 0
- Parent of node i : $\text{floor}((i - 1)/2)$
- Left child of node i : $2i + 1$
- Right child of node i : $2i + 2$
- Is node i a leaf: check if $2i + 2 > n$
- When node i has only one child $2i + 2 = n$



0	1	2	3	4	5
10	5	7	4	2	1

Algorithms for Array-Based Heap Operations

Assume our class of heaps has the following private data members:

items: an array of heap items

itemCount: an integer equal to the number of items in the heap

maxItems: an integer equal to the maximum capacity of the heap

Removing An Item From a Heap

1. Remove the root (actually remove the last node of the tree and place its item in the root.) The result is *not* necessarily a heap (called semiheap).
2. If it is not a heap, transform the tree into a heap: The item in the root percolates down the tree until it reaches a node in which it is not out of place. Compare the root with its children. If the root is smaller than the larger of the items in its children, swap it with that larger item.

```
// Copy the item from the last node and place it into the root
items[0] = items[itemCount - 1]
// Remove the last node
itemCount --
```

Algorithm for Transformation into Heap

// Converts a semiheap rooted at index root into a heap.

heapRebuild(root: integer, items: ArrayType, itemCount: integer)

// Recursively percolate the item at index root down to its proper position
// by swapping it with its larger child, if the child is larger than the item.
// If the item is at a leaf, nothing needs to be done.

if (the root is not a leaf)

{

// The root must have a left child; assume it is the larger child

largerChildIndex = 2 * rootIndex + 1 // Left child index

if (the root has a right child)

{

rightChildIndex = largerChildIndex + 1 // Right child index

if (items[rightChildIndex] > items[largerChildIndex])

 largerChildIndex = rightChildIndex // Larger child index

}

Algorithm for Transformation into Heap

```
// If the item in the root is smaller than the item in the
// larger child, swap items

if (items[rootIndex] < items[largerChildIndex])
{
    Swap items[rootIndex] and items[largerChildIndex]

    // Transform the semiheap rooted at largerChildIndex into a heap
    heapRebuild(largerChildIndex, items, itemCount)
}

// Else root is a leaf, so you are done
```

Removing An Item From a Heap

Now the heap's remove operation uses `heapRebuild` as follows:

```
// Copy the item from the last node into the root
items[0] = items[itemCount - 1]
// Remove the last node
itemCount--
// Transform the semiheap back into a heap
heapRebuild(0, items, itemCount)
```

Adding An Item to a Heap

- A new item is inserted at the bottom of the tree, and it percolate up to its proper place.
- It is easy to percolate up a node, because the parent of the node in `items[i]` is always stored in `items[(i - 1) / 2]` unless, of course, the node is the root.

Adding an Item to a Heap: Pseudocode

```
// Insert newData into the bottom of the tree
items[itemCount] = newData

// Percolate new item up to the appropriate spot in the tree
newDataIndex = itemCount
inPlace = false

while ( (newDataIndex >= 0) and !inPlace)
{
    parentIndex = (newDataIndex - 1) / 2
    if (items[newDataIndex])
        inPlace = true
    else
    {
        Swap items[newDataIndex] and items[parentIndex]
        newDataIndex = parentIndex
    }
}
itemCount++
```

Heap Sort

- The **heap sort algorithm** uses a heap to sort an array of items that are in no particular order.
- Heap sort converts an array into a heap to locate the array's largest item. This step enables the heap sort to sort an array in an efficient manner.
- Heap sort, like merge sort, has good worst-case and average-case behaviors, but neither algorithm is as good in the average case as quick sort. Heap sort has an advantage over merge sort in that it does not require a second array.

Heap Sort Procedure

-
- 1 • Take the root (maximum) element off the heap, and put it in its place.
 - 2 • Reheap the remaining elements. (This action puts the next-largest element into the root position.)
 - 3 • Repeat until no more elements are left.

Example 1. Heap Sort. Ascending Order

Sort the following array in ascending order using heap sort.

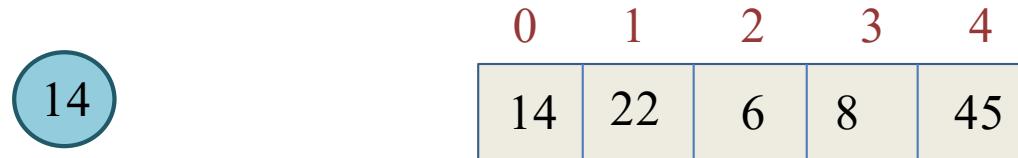
0	1	2	3	4
14	22	6	8	45

Show trace of heap sort procedures.

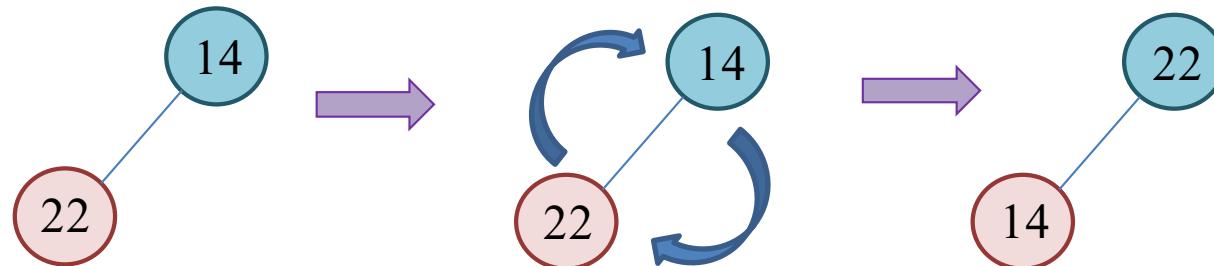
Example 1 Solution. Heap Sort. Ascending Order

A. Build a max heap, based on the given array

1. Insert first element, 14 (this will be the root of heap).



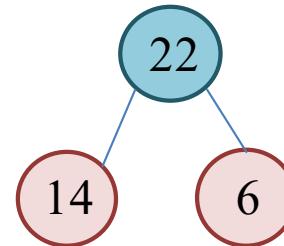
2. Insert the second element 22; since heap should be a complete binary tree, we should add 22 to the left most element in the tree in the next level. But $22 > 14$, and this is not a heap; 22 should percolate up (Swap 14 and 22):



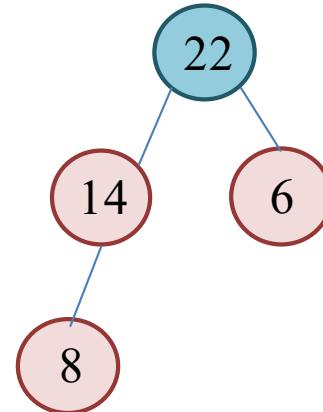
Example 1 Solution. Heap Sort. Ascending Order

- Insert the third element 6; since heap should be a complete binary tree, we should add 6 as the next child of 22. (all levels except last level should be filled). You see $6 < 22$, and this is still a heap; no swap is needed.

0	1	2	3	4
14	22	6	8	45

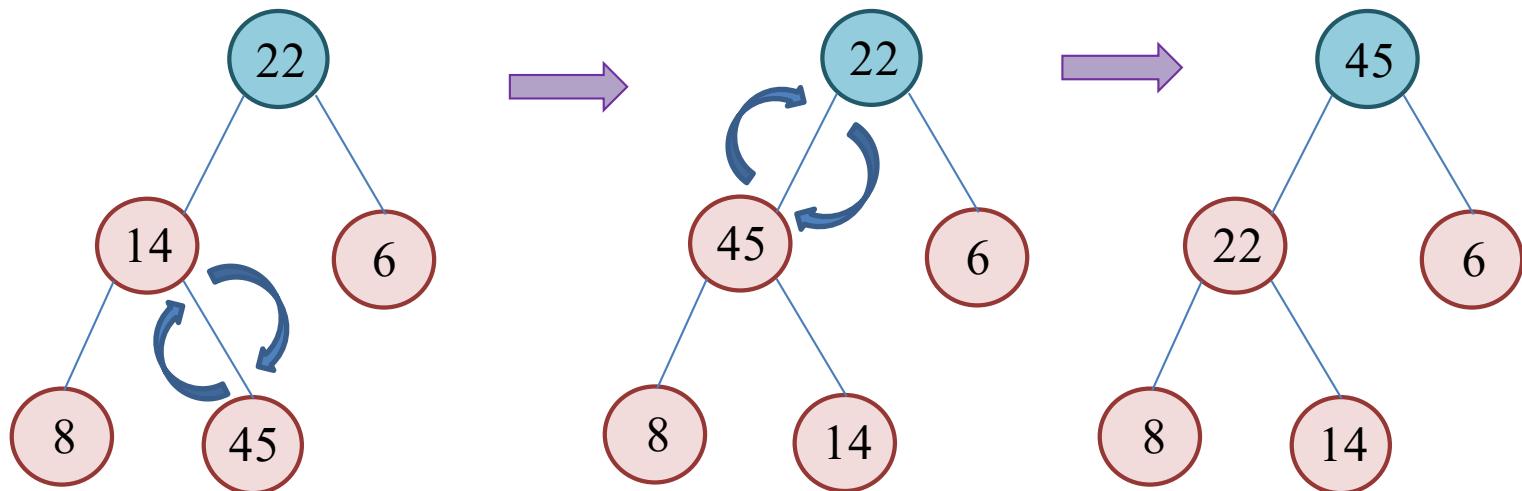


- Insert the fourth element 8; we should add 8 as the left child of 14. You see $8 < 14$, no swap is needed.



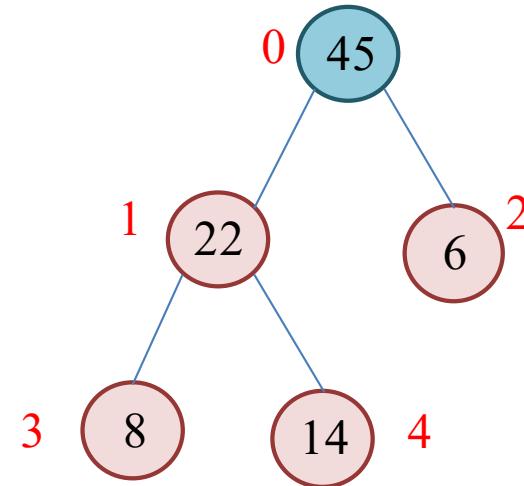
Example 1 Solution. Heap Sort. Ascending Order

5. Insert fifth element 45; we add 45 as the next child of 14. You see $45 > 14$ (child value is more than parent value). This is not a heap and we need to swap 14 and 45. Then we compare 45 with its new parent, 22. We see that $45 > 22$, then we need to do another swapping to satisfy the max heap condition. Then stop, because 45 is at the root now.



Example 1 Solution. Heap Sort. Ascending Order

Now all elements were inserted in a max heap.



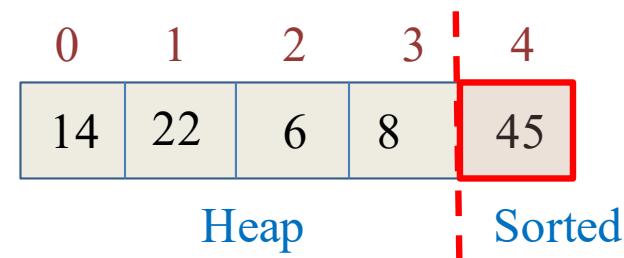
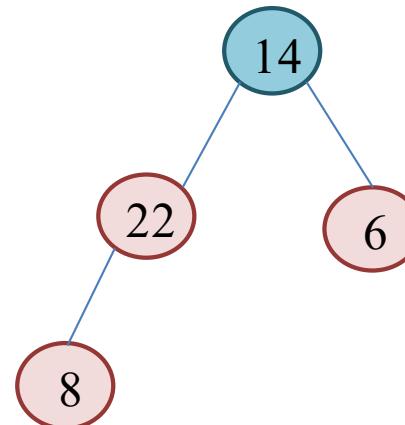
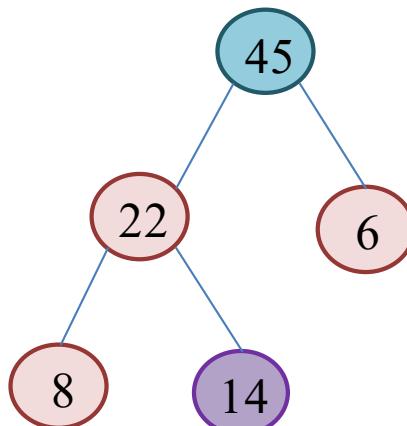
The array representation of this max heap would change to:

0	1	2	3	4
45	22	6	8	14

Example 1 Solution. Heap Sort. Ascending Order

B. Delete elements one by one from the max heap.

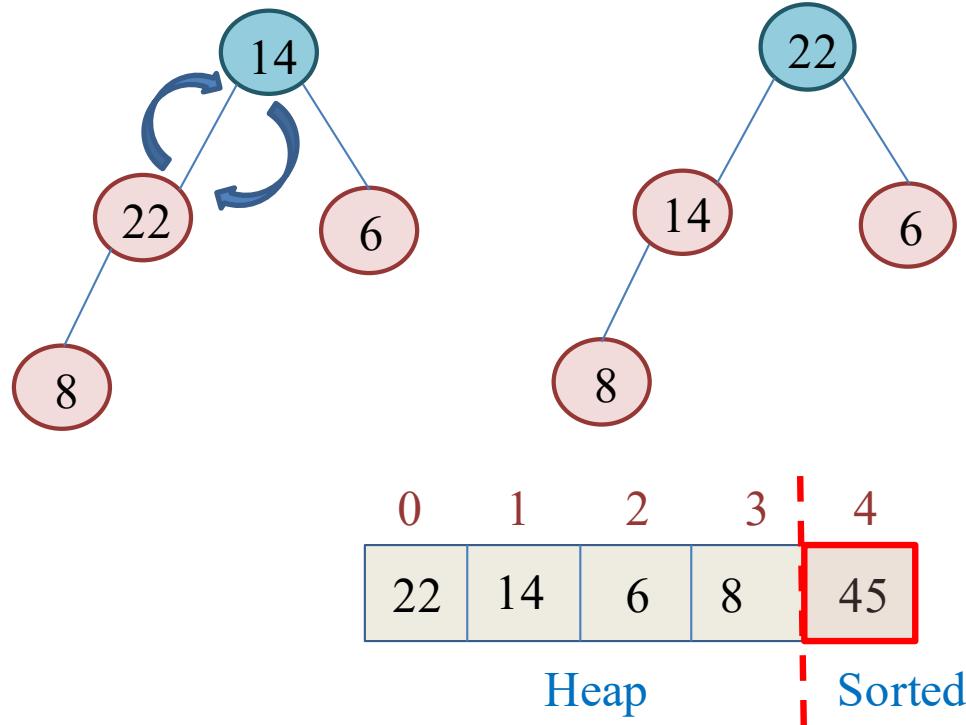
1. Replace root node with the bottom-right most leaf node in the heap and update the array. Place the removed root element in the last position of array (or swap 14 and 45 positions in the array)



Note: After each insertion we should check if the max heap condition is satisfied.

Example 1 Solution. Heap Sort. Ascending Order

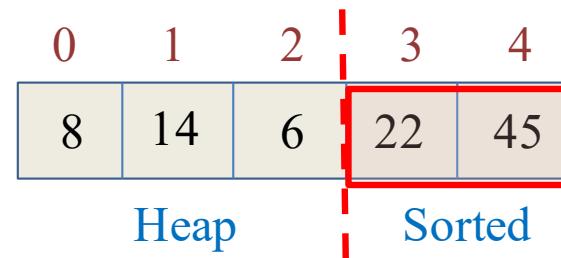
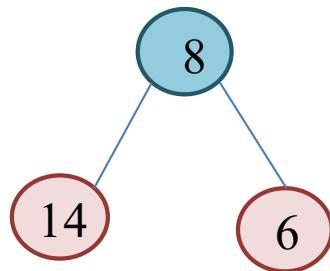
Swap node with the highest valued child and update the array.



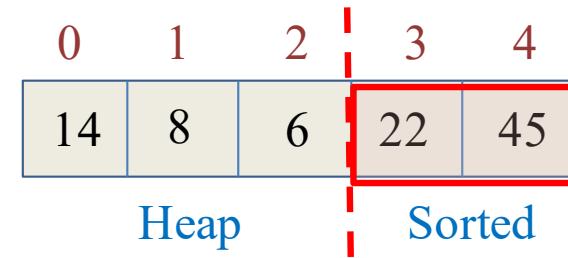
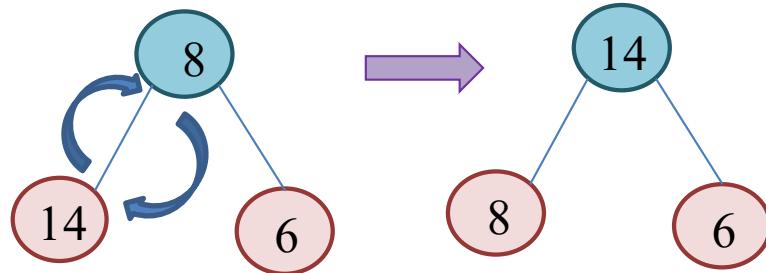
2. Remove next element 22; replace bottom-right—most leaf node (8) with the root (22) and update the array.

Example 1 Solution. Heap Sort. Ascending Order

After removing 22:

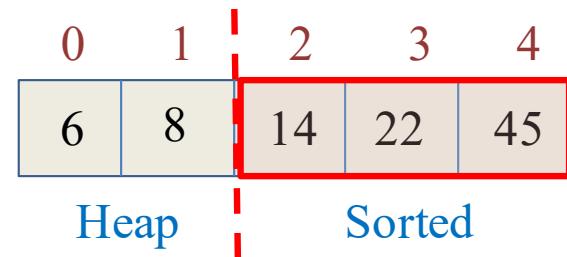
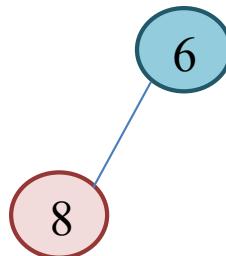


This is not a heap, then 8 and 14 should be swapped.

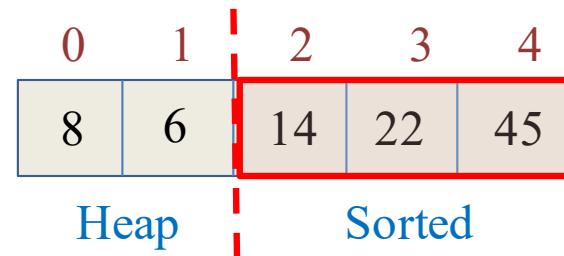
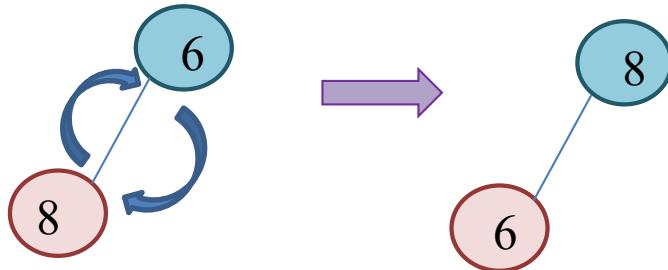


Example 1 Solution. Heap Sort. Ascending Order

3. Remove next element 14; replace bottom-right—most leaf node (6) (or last element of array) with the root 14 and update the array.



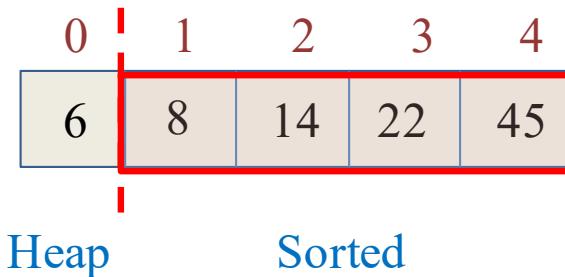
This is not a heap, then 8 and 6 should be swapped.



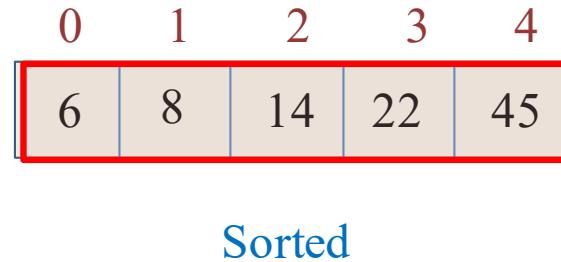
Example 1 Solution. Heap Sort. Ascending Order

4. Remove next element 8; replace 6 with the root 8 and update the array.

6

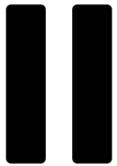


6 is the only element left, then the array has been sorted completely.



```
// Sorts anArray[0...n-1].  
heapSort(anArray: ArrayType, n: integer)  
    // Build initial heap  
    for (index = n / 2 down to 0)  
    {  
        // Assertion: The tree rooted at index is a semiheap  
        heapRebuild(index, anArray, n)  
        // Assertion: The tree rooted at index is a heap  
    }  
    // Assertion: anArray[0] is the largest item in heap anArray[0...n-1]  
    // Move the largest item in the Heap region—the root anArray[0]—to the beginning  
    // of the Sorted region by swapping items and then adjusting the size of the regions  
    Swap anArray[0] and anArray[n - 1]  
    heapSize = n - 1 // Decrease the size of the Heap region, expand the Sorted region  
    while (heapSize > 1)  
    {  
        // Make the Heap region a heap again  
        heapRebuild(0, anArray, heapSize)  
        // Move the largest item in the Heap region—the root anArray[0]—to the beginning  
        // of the Sorted region by swapping items and then adjusting the size of the regions  
        Swap anArray[0] and anArray[heapSize - 1]  
        heapSize-- // Decrease the size of the Heap region, expand the Sorted region  
    }
```

heapSort



You Try 1. Heap Sort .Trace of Algorithm

Show a trace of Heap Sort algorithm (in previous slide) for the following array (Sort it in ascending order).

10	9	6	3	2	5
----	---	---	---	---	---

Show how the heap is changing in each step and how its array representation is updated.

Heap Sort Algorithm Efficiency

- The analysis of the efficiency of heap sort is similar to that of merge sort.
- Both algorithms are $O(n \log n)$ in both the worst and average cases.
- Heap sort has an advantage over merge sort in that it does not require a second array.
- Quick sort is also $O(n \log n)$ in the average case but is $O(n^2)$ in the worst case.
- Even though quick sort has poor worst-case efficiency, it is generally the preferred sorting algorithm.

Next Lecture

We focus on:

- Hashing

Readings/ Study Reference

Course Textbook: “*Data Structure and Algorithms in a Nutshell*” A. Wong et al.

Chapter 9. Sorting

Section 9.5. Heap Sort

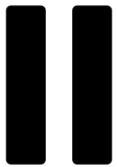
The End of Lecture

Course: Data Structures and Algorithms

Instructor: Homeyra Pourmohammadali

You Try Question and Solution

Sorting: Heap Sort



You Try 1. Heap Sort .Trace of Algorithm

Show a trace of Heap Sort algorithm (in previous slide) for the following array (Sort it in ascending order).

10	9	6	3	2	5
----	---	---	---	---	---

Show how the heap is changing in each step and how its array representation is updated.

You Try 1 Solution. Heap Sort .Trace of Algorithm

Array **anArray**

After making **anArray** a heap

Heap					
10	9	6	3	2	5
0	1	2	3	4	5

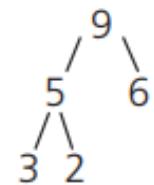
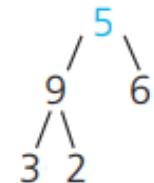
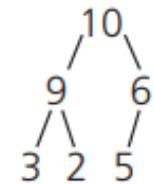
After swapping **anArray[0]** with **anArray[5]** and decreasing the size of the Heap region

Heap					
5	9	6	3	2	10
0	1	2	3	4	5

After **heapRebuild(0, anArray, 4)**

Heap					
9	5	6	3	2	10
0	1	2	3	4	5

Tree representation
of **Heap** region



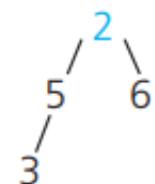
You Try 1 Solution. Heap Sort .Trace of Algorithm

Array `anArray`

Tree representation
of `Heap` region

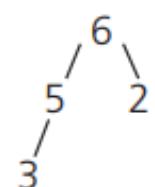
After swapping `anArray[0]` with `anArray[4]` and decreasing the size of the `Heap` region

Heap				Sorted	
2	5	6	3	9	10
0	1	2	3	4	5



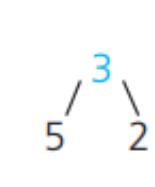
After `heapRebuild(0, anArray, 3)`

Heap				Sorted	
6	5	2	3	9	10
0	1	2	3	4	5



After swapping `anArray[0]` with `anArray[3]` and decreasing the size of the `Heap` region

Heap				Sorted	
3	5	2	6	9	10
0	1	2	3	4	5



You Try 1 Solution. Heap Sort .Trace of Algorithm

After `rebuildHeap(0, anArray, 2)`

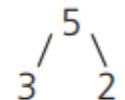
After swapping `anArray[0]` with `anArray[2]` and decreasing the size of the Heap region

After `heapRebuild(0, anArray, 1)`

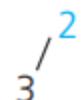
Array `anArray`

Heap			Sorted		
5	3	2	6	9	10
0	1	2	3	4	5

Tree representation of `Heap` region



Heap			Sorted		
2	3	5	6	9	10
0	1	2	3	4	5



Heap			Sorted		
3	2	5	6	9	10
0	1	2	3	4	5

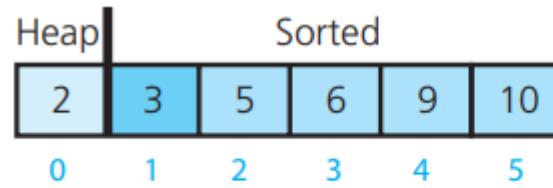


You Try 1 Solution. Heap Sort .Trace of Algorithm

Array **anArray**

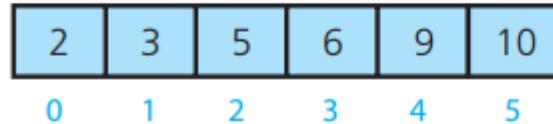
Tree representation
of **Heap region**

After swapping **anArray[0]**
with **anArray[1]** and decreasing
the size of the Heap region



2

Array is sorted



6

```
// Sorts anArray[0...n-1].  
heapSort(anArray: ArrayType, n: integer)  
    // Build initial heap  
    for (index = n / 2 down to 0)  
    {  
        // Assertion: The tree rooted at index is a semiheap  
        heapRebuild(index, anArray, n)  
        // Assertion: The tree rooted at index is a heap  
    }  
    // Assertion: anArray[0] is the largest item in heap anArray[0...n-1]  
    // Move the largest item in the Heap region—the root anArray[0]—to the beginning  
    // of the Sorted region by swapping items and then adjusting the size of the regions  
    Swap anArray[0] and anArray[n - 1]  
    heapSize = n - 1 // Decrease the size of the Heap region, expand the Sorted region  
    while (heapSize > 1)  
    {  
        // Make the Heap region a heap again  
        heapRebuild(0, anArray, heapSize)  
        // Move the largest item in the Heap region—the root anArray[0]—to the beginning  
        // of the Sorted region by swapping items and then adjusting the size of the regions  
        Swap anArray[0] and anArray[heapSize - 1]  
        heapSize-- // Decrease the size of the Heap region, expand the Sorted region  
    }
```

heapSort

The End of You Try Activity

Course: Data Structures and Algorithms

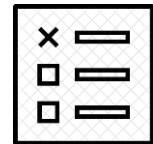
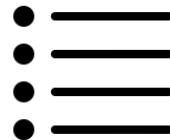
Instructor: Homeyra Pourmohammadali

Hashing: Hash Table and Hash Function

Motivation

Hash tables are used when crafting software components that provide:

- information security
- fast access to unordered data
- association of shorter strings with larger data records



Motivation

Information Security: Password Verification

When you use some web service and enter your credentials to log in, is your password sent in plain-text through the network to the server for verification?

Special cryptographic [hash functions](#) are used for this purpose.



Learning Outcomes

By the end of this lecture you will be able to find out:

- what is hash table data structure
- what is hash function and its applications
- what is hashing

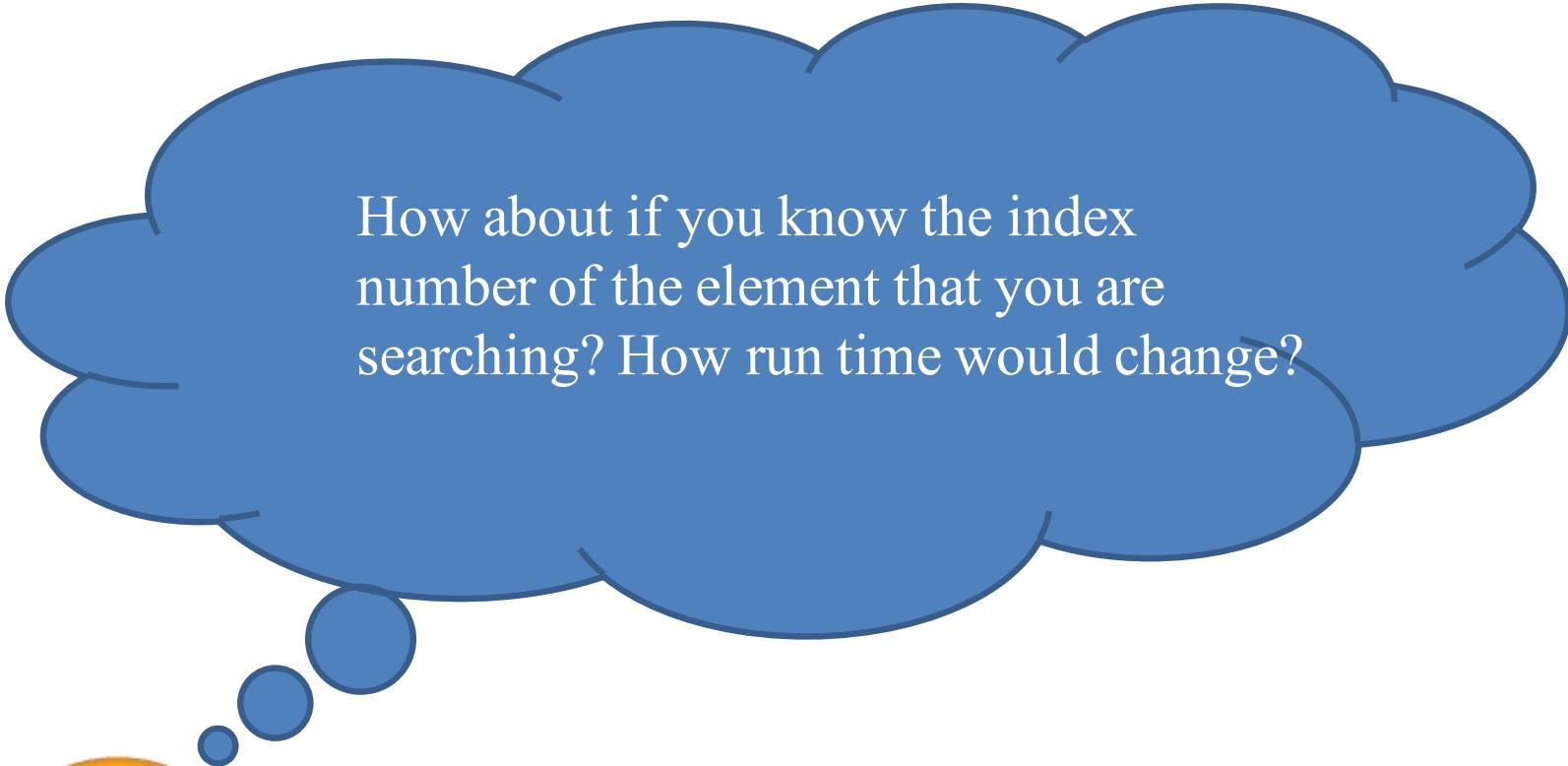
Introduction

We learned about the array and linked list data structures and their run time efficiency.

Data structure	Lookup
Array	$O(n)$
Linked list	$O(n)$

Consider an array variable; to find an item in the list you can do linear searching. If you don't know the index of the element, this requires checking each item. For large number of elements this would take a very long time ($O(n)$).

Introduction



How about if you know the index number of the element that you are searching? How run time would change?



In this case, the run time will be independent of the size of array and position of the element in the array, then your search will be faster, and ($O(1)$).

Introduction

Which elements of the array contains the value that you are searching?

Each index can be calculated using the element value itself. This way you relate the index number to the data.



Example 1. Hash Table

Consider an array used for storing the following names:
Jan, Tim, Mia, Sam, Leo, Ted, Be, Lou, Ada, Max and Zoe.

0	1	2	3	4	5	6	7	8	9	10
Jan	Tim	Mia	Sam	Leo	Ted	Bea	Lou	Ada	Max	Zoe

For instance, if you search for **Max** in this array, you need to start your linear search from the item at index 0 and check each item one by one till **index 9** to find Max.

0	1	2	3	4	5	6	7	8	9	10
Jan	Tim	Mia	Sam	Leo	Ted	Bea	Lou	Ada	Max	Zoe

But, if you look for item in index 5, you don't need such linear search.

Example 1. Hash Table

You can calculate each index using the element value itself.

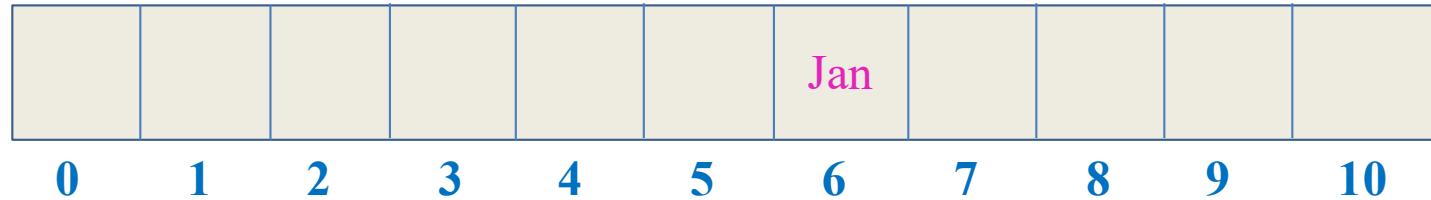
For instance, you can take each letter of the word **Jan** and get its ASCII code:

J	74	a	97	n	110
---	----	---	----	---	-----

Add the ASCII codes together: $74 + 97 + 110 = 281$

Divide 281 by the number of array elements (11 in here): $281 / 11 = 25.6$

Take the remainder of that calculation as the index (use modulo operator): **Index 6** will be where you place **Jan**.



Example 1. Hash Table

You can do the same for the other names, and calculate their index #:

							Sum	Modulo
Jan	J	74	a	97	n	110	281	6
Tim	T	84	i	105	m	109	298	1
Mia	M	77	i	105	a	97	279	4
Sam	S	83	a	97	m	109	289	3
Leo	L	76	e	101	o	111	288	2
Ted	T	84	e	101	d	100	285	10
Bea	B	66	e	101	a	97	264	0
Lou	L	76	o	111	u	117	304	7
Ada	A	65	d	100	a	97	262	9
Max	M	77	a	97	x	120	294	8
Zoe	Z	90	o	111	e	101	302	5

Example 1. Hash Table

The array will be:

0	1	2	3	4	5	6	7	8	9	10
Bea	Tim	Leo	Sam	Mia	Zoe	Jan	Lou	Max	Ada	Ted

Then, if you want to find **Sam** for instance, the index # (3) can be quickly calculated. This is an example of a **hash table**.

You can use it to store key value pairs, for example Tim's name is the **key** (used for finding its index) and his student number or his date of birth can be the corresponding **value**. The hash tables of key, value pairs sometimes is called **hash map**. In OOP, each person can be an instance of a class. You can store as much related data as you like for each key, such as last name, address, etc.

Jan	6
Tim	1
Mia	4
Sam	3
Leo	2
Ted	10
Bea	0
Lou	7
Ada	9
Max	8
Zoe	5

Example 1. Hash Table

The array will be:

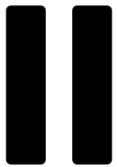
0	1	2	3	4	5	6	7	8	9	10
Bea	Tim	Leo	Sam	Mia	Zoe	Jan	Lou	Max	Ada	Ted

You notice that in this case, none of two names have the same index number, which won't happen often.

But, what if two or more names have the same index number and need to be stored in the same position?

This situation is known as **collision**: The condition resulting when two or more keys produce the same hash location.

Jan	6
Tim	1
Mia	4
Sam	3
Leo	2
Ted	10
Bea	0
Lou	7
Ada	9
Max	8
Zoe	5



You Try 1. Hash Table

Generate the index number for the following set of names, using the same hash function in previous example.

Mia, Tim, Bea, Zoe, Sue, Len, Moe, Lou, Rae, Max and Tod.

You can find the ASCII code of each letter using the table given in the next slide or use this link:

<http://sticksandstones.kstrom.com/appen.html>

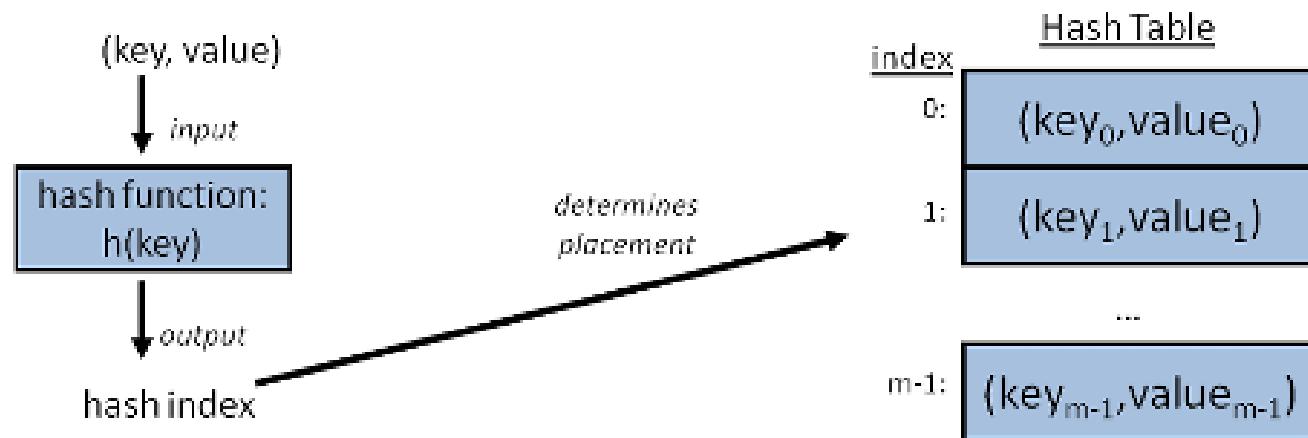
What do you notice? Which names generate the same index number? We discuss how to resolve collision later.

0	1	2	3	4	5	6	7	8	9	10

Letter	ASCII Code	Letter	ASCII Code
a	097	A	065
b	098	B	066
c	099	C	067
d	100	D	068
e	101	E	069
f	102	F	070
g	103	G	071
h	104	H	072
i	105	I	073
j	106	J	074
k	107	K	075
l	108	L	076
m	109	M	077
n	110	N	078
o	111	O	079
p	112	P	080
q	113	Q	081
r	114	R	082
s	115	S	083
t	116	T	084
u	117	U	085
v	118	V	086
w	119	W	087
x	120	X	088
y	121	Y	089
z	122	Z	090

Hash Table (Hash Map)

- A **hash table** is a data structure that is used for fast retrieval of data no matter how large is the data set.
- A hash table (hash map) ADT is a collection of $(key, value)$ associations, where each key is typically unique, and keys need to be comparable objects, such as strings.



Hash Function

- A hash function, $h(key)$, is used to map each key to specific location inside the data structure.
- A hash function, ideally should be simple to compute and should ensure that any two distinct keys get different cells.
- In this example, *Mia* hashes to 4, *Lou* hashes to 7, *Ted* hashes to 10.

0	1	2	3	4	5	6	7	8	9	10
Bea	Tim	Leo	Sam	Mia	Zoe	Jan	Lou	Max	Ada	Ted

Hash Function

Hash Function

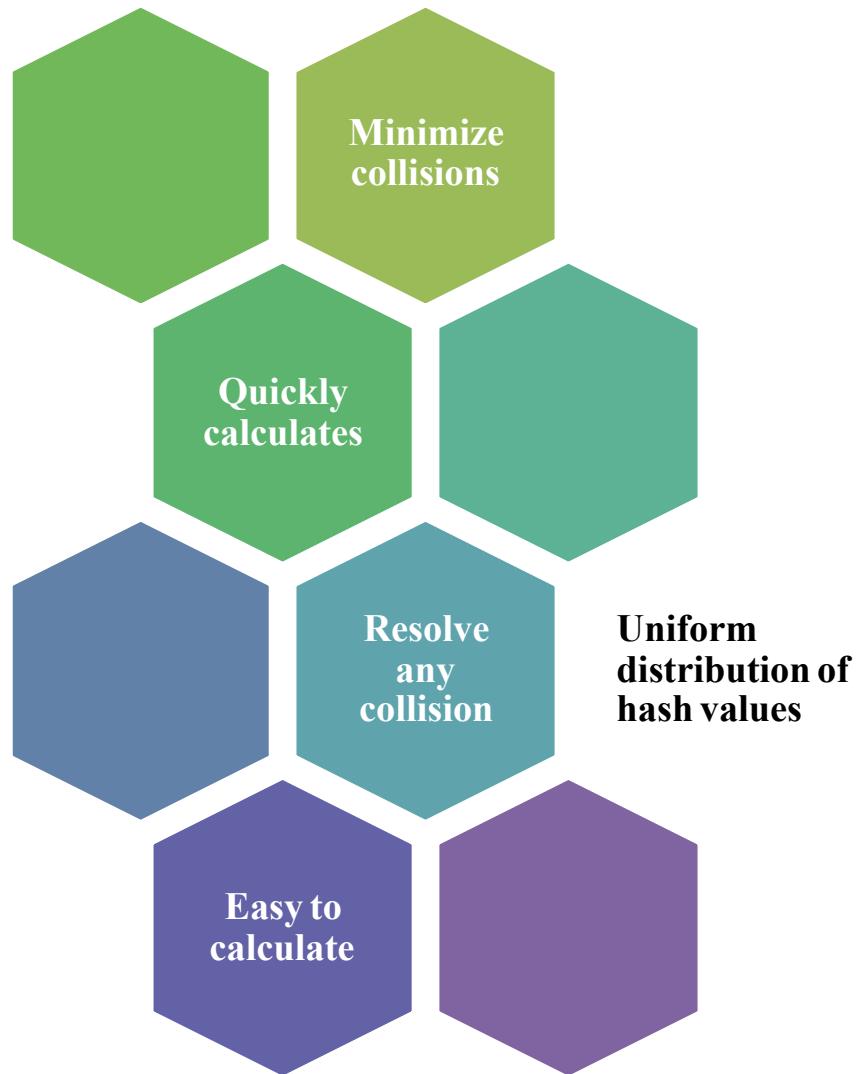
The calculation applied to a key which may be a very large number or very long string to transform it into small indexed number that corresponds to a position in the hash table.

For alphanumeric keys, you can divide the sum of ASCII codes in a key by the number of available addresses, n , and take the remainder.

For numeric keys, you can divide the key by number of available addresses, n , and take the remainder.

There are many hash functions available; this depends on your data and your selection.

Objectives of Hash Functions



Properties of Good Hash Functions

Hash functions should be designed with these properties:

One-way

- After the hash value generation, it must be impossible to convert it back into the original data.

Super fast

- If hash function computes hash values slowly, the procedure is useless. Hash value is the output string generated by a hash function.

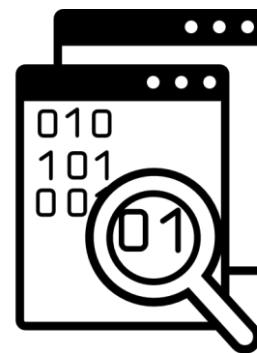
Collision-free

- Every input string must generate a unique output string (called cryptographic hash function).

Applications of Hash Functions

Searching through large quantities of data:

- The search term is converted to a hash value.
- This limits the number of letters, digits and symbols that have to be compared.
- This is much more efficient than searching every field that exists in the data table.



Applications of Hash Functions

Digital Signature

- Keys generated using hash functions are used to add a digital signature to messages
- This proves the integrity of a message but does not actually encrypt it.



Applications of Hash Functions

Protecting sensitive data

- Cyber attacks target the login details of online accounts.
- They either disrupt operation of a website or access information about payment method.
- Passwords need to be encrypted before they are stored, using hash functions.



Hashing

- The use of hash tables and hash functions is commonly referred to as **hashing**.
- For instance, when users enter their password and username into client login screen, a cryptographic hash function is used to map their password into an encoded (hashed) string.
- Then username and encoded password are sent across the Internet to the intended server authentication.

Main Operations of Hash Table ADT

In addition to storing key-value association, a hash table ADT includes the following three operations:

search

Returns the value that matches a particular key

insert

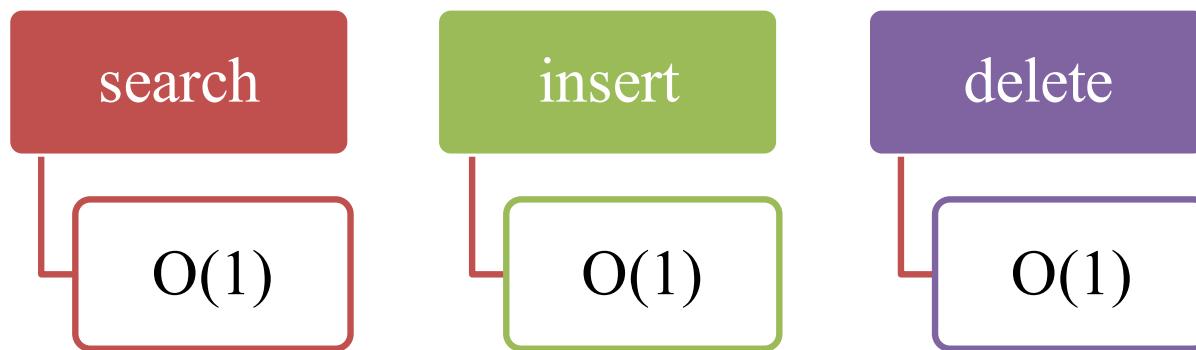
Inserts a key-value association into the hash table

delete

Deletes a key-value association from the hash table

Run Time Efficiency of Operations

All three operations are expected to have similar run-time when there is **no collision**.



Suitability of the hash function and collision can significantly affect run-time efficiency of these operations.

Next Lecture

We focus on:

- Collision and how to resolve it
- Hash table implementation using:
separate chaining and open addressing

Readings/ Study Reference

Course Textbook: “*Data Structure and Algorithms in a Nutshell*” A. Wong et al.

Chapter 10. Hashing

Section 10.1. Overall Structure

Section 10.2. Hashing

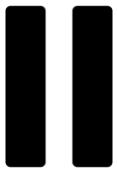
Section 10.4. Hash Functions

The End of Lecture

Course: Data Structures and Algorithms

Instructor: Homeyra Pourmohammadali

You Try Questions and Solutions
Hash Tables and Hashing



You Try 1. Hash Table

Generate the index number for the following set of names, using the same hash function in previous example.

Mia, Tim, Bea, Zoe, Sue, Len, Moe, Lou, Rae, Max and Tod.

You can find the ASCII code of each letter using the table given in the next slide or use this link:

<http://sticksandstones.kstrom.com/appen.html>

What do you notice? Which names generate the same index number? We discuss how to resolve collision later.

0	1	2	3	4	5	6	7	8	9	10

Letter	ASCII Code	Letter	ASCII Code
a	097	A	065
b	098	B	066
c	099	C	067
d	100	D	068
e	101	E	069
f	102	F	070
g	103	G	071
h	104	H	072
i	105	I	073
j	106	J	074
k	107	K	075
l	108	L	076
m	109	M	077
n	110	N	078
o	111	O	079
p	112	P	080
q	113	Q	081
r	114	R	082
s	115	S	083
t	116	T	084
u	117	U	085
v	118	V	086
w	119	W	087
x	120	X	088
y	121	Y	089
z	122	Z	090

You Try 1 Solution. Hash Table

							Sum	Modulo
Mia	M	77	i	105	a	97	279	4
Tim	T	84	i	105	m	109	298	1
Bea	B	66	e	101	a	97	264	0
Zoe	Z	90	o	111	e	101	302	5
Sue	S	83	u	117	e	101	301	4
Len	L	76	e	101	n	110	287	1
Moe	M	77	o	111	e	101	289	3
Lou	L	76	o	111	u	117	304	7
Rae	R	82	a	97	e	101	280	5
Max	M	77	a	97	x	120	294	8
Tod	T	84	o	111	d	100	285	9

collision

collision

collision

The End of You Try Activity

Course: Data Structures and Algorithms

Instructor: Homeyra Pourmohammadali

Hashing: Collisions

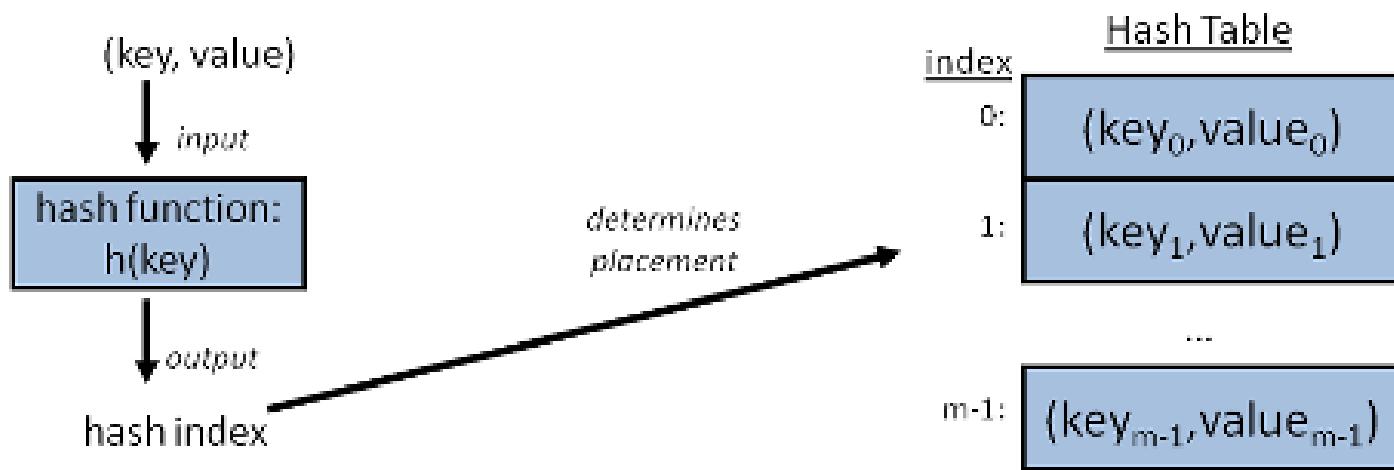
Learning Outcomes

By the end of this lecture you will be able to find out:

- what are the approaches to resolve collision
- what are separate chaining and open addressing methods

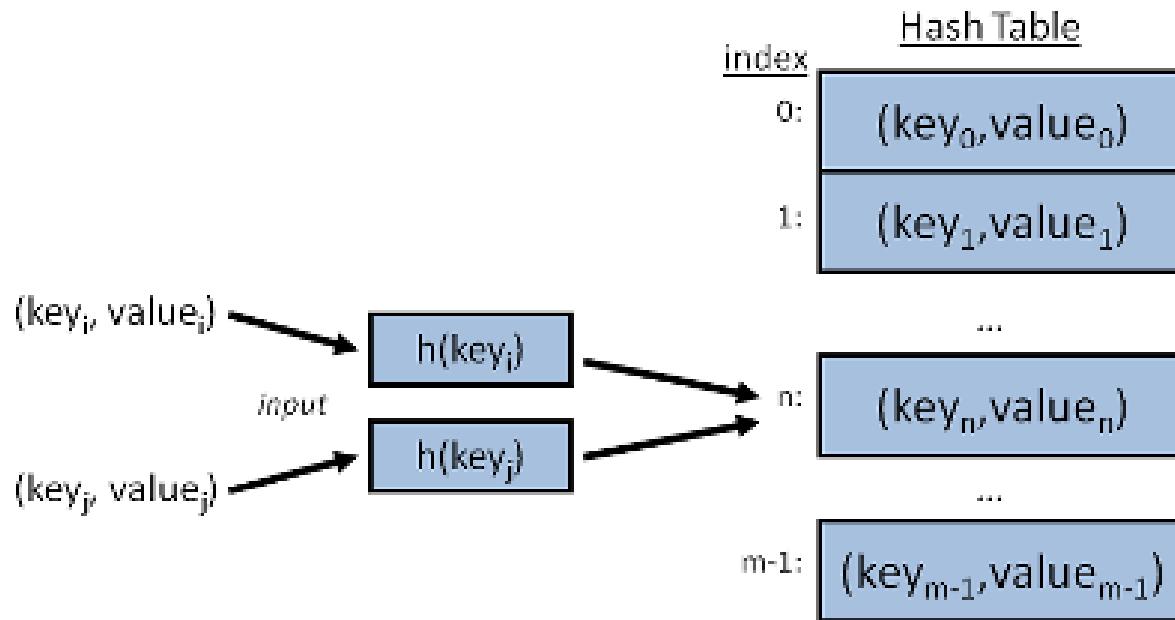
Recall: Hash Table (Hash Map)

- A hash table (has map) ADT is a collection of (*key*, *value*) associations, where each key is typically unique, and keys need to be comparable objects, such as strings.
- A hash function, $h(\text{key})$, is used to map each key to specific location inside the data structure.



Hash Table Collision

The condition resulting when two or more keys produce the same hash location.



In diagram, two distinct key-value pairs ($\text{key}_i, \text{value}_i$) and ($\text{key}_j, \text{value}_j$) are mapped to the same hash table index, n .

Hash Table Collision

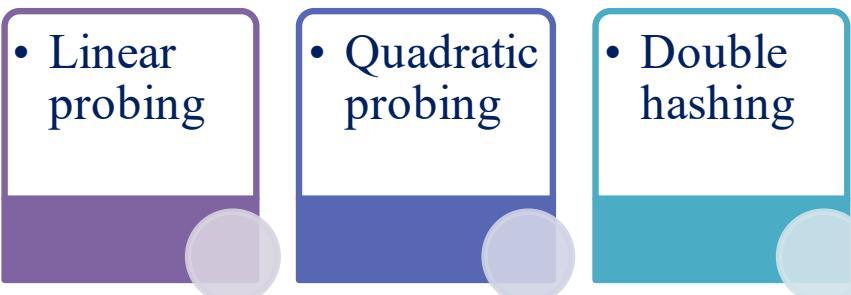
Two general approaches to collision resolution are common.

Open Addressing: This approach places the new item into another location within the hash table.

Separate Chaining: This approach changes the structure of the hash table so that each location $\text{table}[i]$ can accommodate more than one item.

Open Addressing

- If the hash function indicates a location in the hash table that is already occupied, you look—or **probe**—for some other empty, or open, location in which to place the item. Such schemes are said to use **open addressing**.
- There are various open-addressing schemes to probe for an empty location.
- Some open addressing techniques are:



Linear Probing

- In this simple scheme to resolve a collision, you search the hash table sequentially, starting from the original hash location.
- If `table[h(searchKey)]` is occupied, you check the locations `table[h(searchKey)+1]`, `table[h(searchKey)+2]`, and so on until you find an available location.
- Typically, you *wrap around* from the last array location to the first array location if necessary.
- One of the problems with linear-probing scheme is that items tend to cluster together in the hash table. One part of the hash table might be quite densely populated, even though another part has relatively few items.

Example 1. Hash Table. Open addressing. Linear Probing

Use ASCII codes and division method to find index number of each of the following names and create a hash table, using linear probing technique: Mia, Tim, Bea, Zoe, Sue, Len, Moe, Lou, Rae, Max, Tod.

Letter	ASCII Code
a	097
b	098
c	099
d	100
e	101
f	102
g	103
h	104
i	105
j	106
k	107
l	108
m	109
n	110
o	111
p	112
q	113
r	114
s	115
t	116
u	117
v	118
w	119
x	120
y	121
z	122

Letter	ASCII Code
A	065
B	066
C	067
D	068
E	069
F	070
G	071
H	072
I	073
J	074
K	075
L	076
M	077
N	078
O	079
P	080
Q	081
R	082
S	083
T	084
U	085
V	086
W	087
X	088
Y	089
Z	090

Example 1 Solution. Hash Table. Collision. Linear Probing

							Sum	Modulo
Mia	M	77	i	105	a	97	279	4
Tim	T	84	i	105	m	109	298	1
Bea	B	66	e	101	a	97	264	0
Zoe	Z	90	o	111	e	101	302	5
Sue	S	83	u	117	e	101	301	4

There is no collision while storing the first four elements.
 Sue generates number 4, but position 4 stores Mia → Collision
 We check next position (5), position 5 stores Zoe → Check the
 next position (position 6), position 6 is empty. Store Sue in 6.

0	1	2	3	4	5	6	7	8	9	10
Bea	Tim			Mia	Zoe	Sue				

Example 1 Solution. Hash Table. Collision. Linear Probing

							Sum	Modulo
Mia	M	77	i	105	a	97	279	4
Tim	T	84	i	105	m	109	298	1
Bea	B	66	e	101	a	97	264	0
Zoe	Z	90	o	111	e	101	302	5
Sue	S	83	u	117	e	101	301	4
Len	L	76	e	101	n	110	287	1

There is no collision while storing the first four elements.

Len generates 1, but position 1 stores Tim → there is collision

We check next position (2), position 2 is empty → store Len in position 2.

0	1	2	3	4	5	6	7	8	9	10
Bea	Tim	Len		Mia	Zoe	Sue				

Example 1 Solution. Hash Table. Collision. Linear Probing

							Sum	Modulo
Mia	M	77	i	105	a	97	279	4
Tim	T	84	i	105	m	109	298	1
Bea	B	66	e	101	a	97	264	0
Zoe	Z	90	o	111	e	101	302	5
Sue	S	83	u	117	e	101	301	4
Len	L	76	e	101	n	110	287	1
Moe	M	77	o	111	e	101	289	3
Lou	L	76	o	111	u	117	304	7
Rae	R	82	a	97	e	101	280	5

Position 5 is occupied. Next available space is at position 8.

0	1	2	3	4	5	6	7	8	9	10
Bea	Tim	Len	Moe	Mia	Zoe	Sue	Lou	Rae		

Example 1 Solution. Hash Table. Collision. Linear Probing

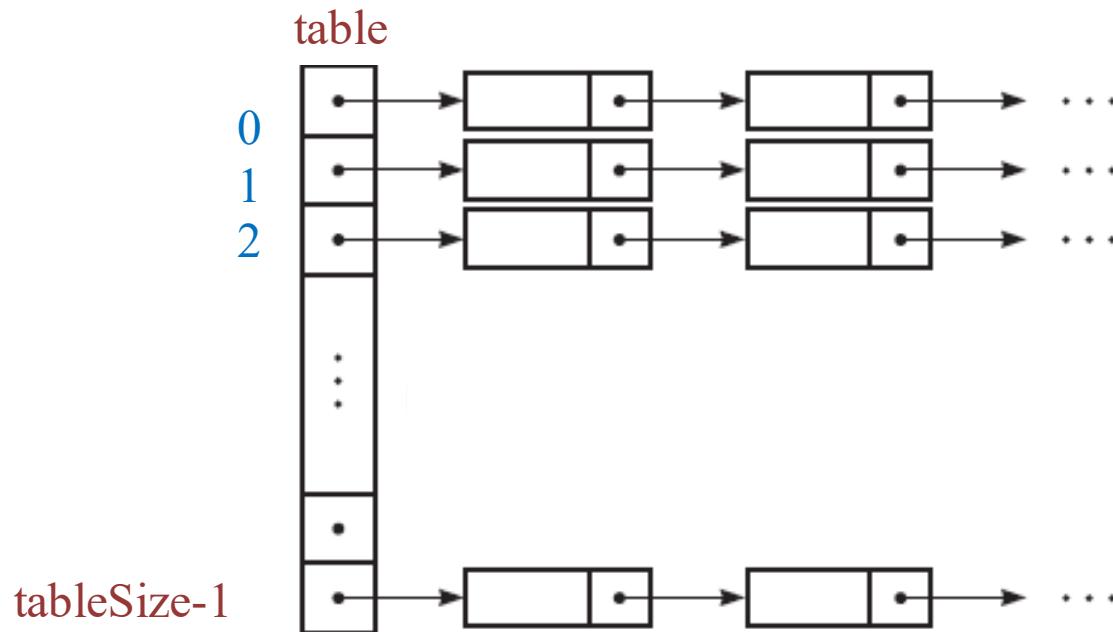
								Sum	Modulo
Mia	M	77	i	105	a	97	279	4	
Tim	T	84	i	105	m	109	298	1	
Bea	B	66	e	101	a	97	264	0	
Zoe	Z	90	o	111	e	101	302	5	
Sue	S	83	u	117	e	101	301	4	
Len	L	76	e	101	n	110	287	1	
Moe	M	77	o	111	e	101	289	3	
Lou	L	76	o	111	u	117	304	7	
Rae	R	82	a	97	e	101	280	5	
Max	M	77	a	97	x	120	294	8	
Tod	T	84	o	111	d	100	285	9	

0 1 2 3 4 5 6 7 8 9 10

Bea	Tim	Len	Moe	Mia	Zoe	Sue	Lou	Rae	Max	Tod
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Separate Chaining

- Another way to resolve collisions is to change the structure of the hash table—so that it can accommodate more than one item in the same location, using separate chaining.
- Each entry `table[i]` is a pointer to a chain of linked nodes containing the items that the hash function has mapped into location i .



Separate Chaining

- With separate chaining, the size is dynamic and can exceed the size of the hash table, because each linked chain can be as long as necessary.
- The length of these chains affects the efficiency of retrievals and removals.
- Separate chaining is the most time-efficient collision-resolution scheme.

Example 1. Hash Table. Separate Chaining

Use division method and hash function of $h(k)=2k+3$ to make a hash table of size $m=10$ using separate chaining technique for the given array: 3, 2, 9, 6, 11, 13, 7, 12

Note: Position of each element is found by:

$$h(k_i) = (2k_i + 3) \bmod m = (2k_i + 3) \% m$$

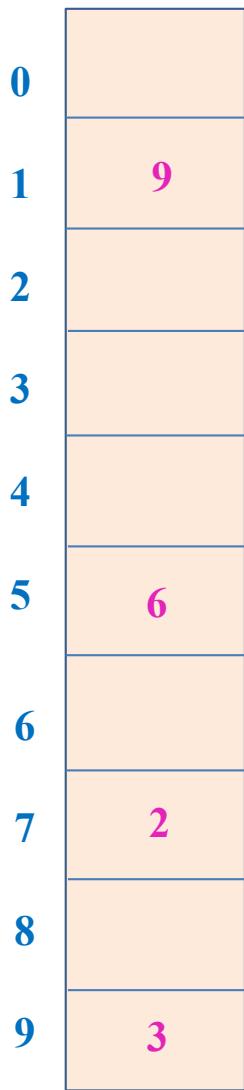
Example 2 Solution. Hash Table. Separate Chaining

Make a table to show keys and modulo

$$h(k_i) = (2k_i + 3) \bmod m = (2k_i + 3) \% m$$

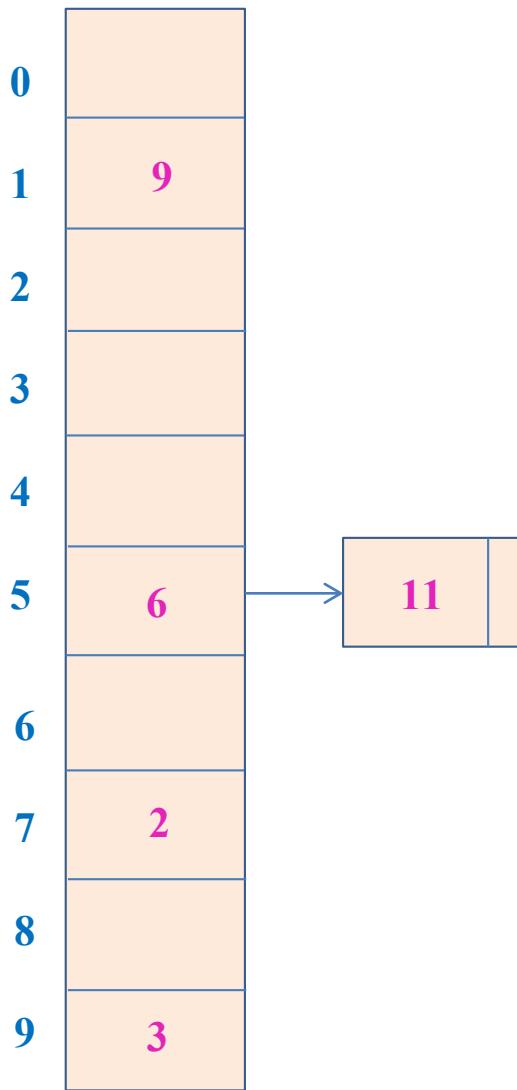
Key	$(2k_i + 3) \% m$	index	
3	$[(2*3) + 3] \% 10 = 9 \% 10$	9	
2	$[(2*2) + 3] \% 10 = 7 \% 10$	7	
9	$[(2*9) + 3] \% 10 = 21 \% 10$	1	
6	$[(2*6) + 3] \% 10 = 15 \% 10$	5	
11	$[(2*11) + 3] \% 10 = 25 \% 10$	5	Collision
13	$[(2*13) + 3] \% 10 = 29 \% 10$	9	Collision
7	$[(2*7) + 3] \% 10 = 17 \% 10$	7	Collision
12	$[(2*12) + 3] \% 10 = 27 \% 10$	7	Collision

Example 2 Solution. Hash Table. Separate Chaining



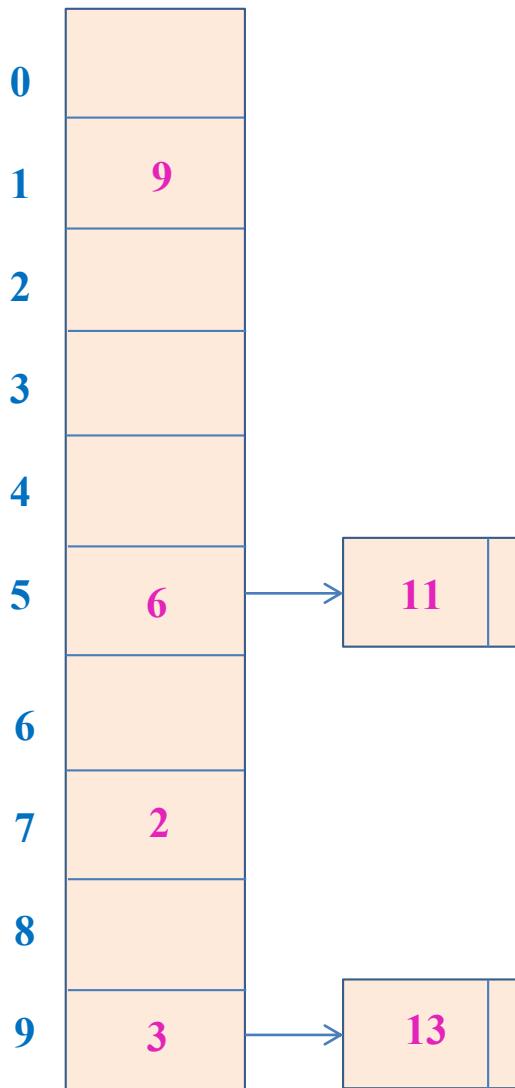
Key	index
3	9
2	7
9	1
6	5
11	5
13	9
7	7
12	7

Example 2 Solution. Hash Table. Separate Chaining



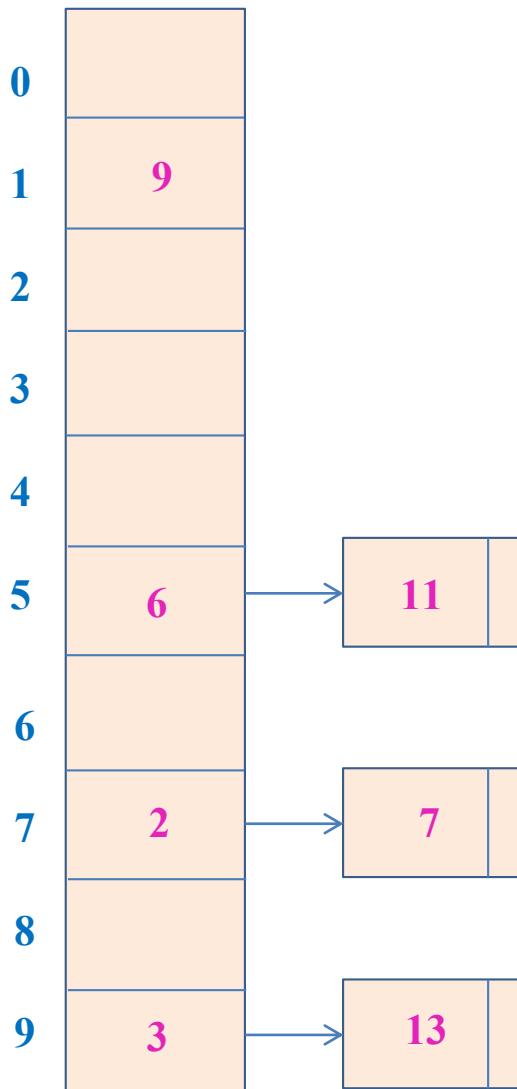
Key	index
3	9
2	7
9	1
6	5
11	5
13	9
7	7
12	7

Example 2 Solution. Hash Table. Separate Chaining



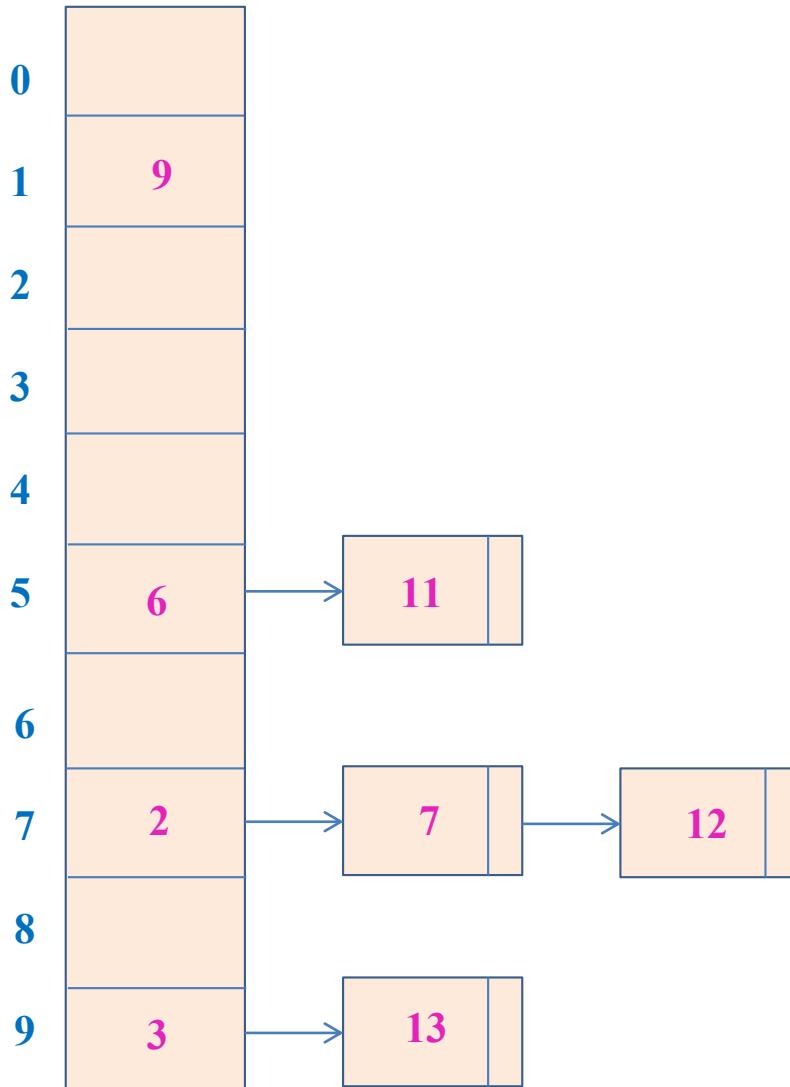
Key	index
3	9
2	7
9	1
6	5
11	5
13	9
7	7
12	7

Example 2 Solution. Hash Table. Separate Chaining



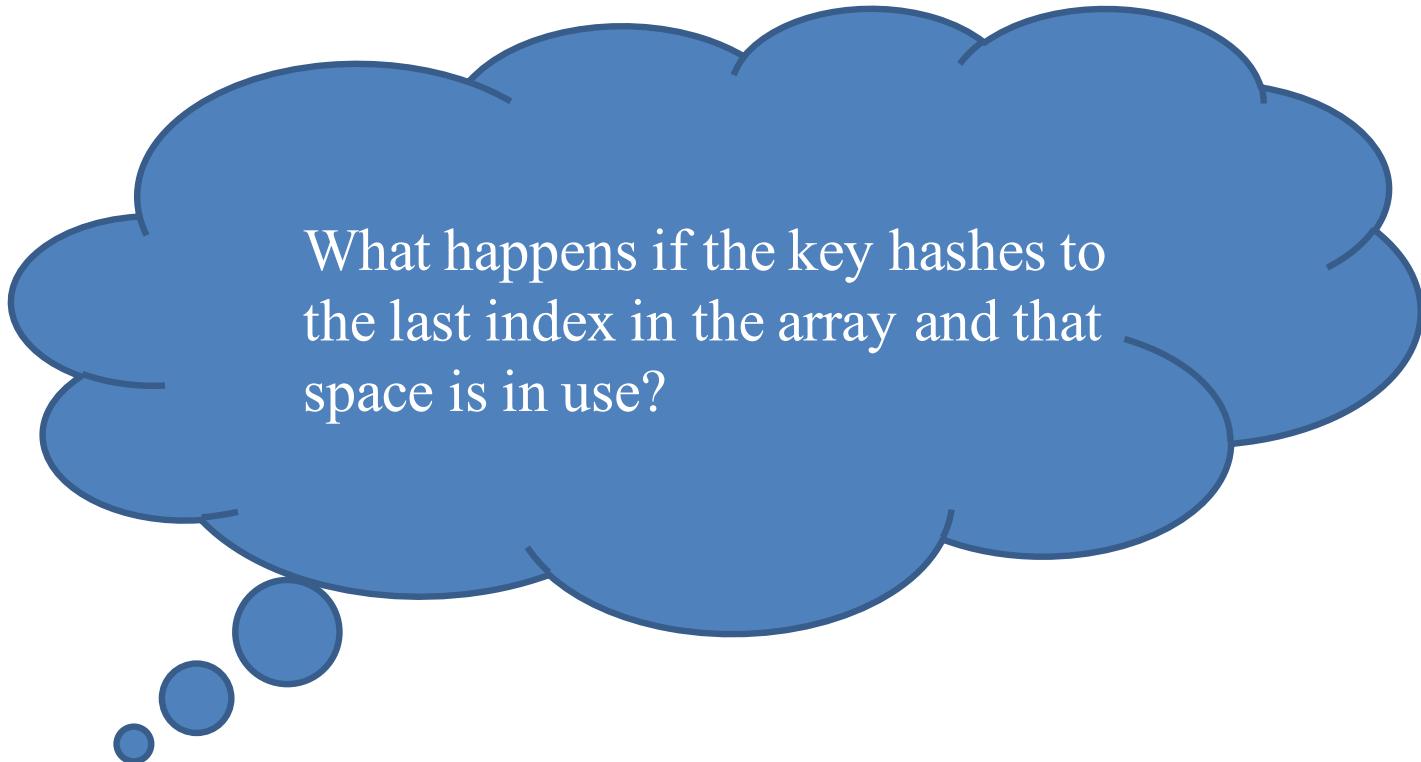
Key	index
3	9
2	7
9	1
6	5
11	5
13	9
7	7
12	7

Example 2 Solution. Hash Table. Separate Chaining



Key	index
3	9
2	7
9	1
6	5
11	5
13	9
7	7
12	7

Question



What happens if the key hashes to the last index in the array and that space is in use?



Answer: We can consider the array to be a circular structure and continue looking for an empty slot at the beginning of the array.



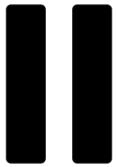
You Try 1. Hash Table. Separate Chaining

Use division method and hash function of $h(k)=3k + 5$ to make a hash table of size $m=10$ using separate chaining technique for the given array: 2, 3, 5, 6, 10, 11, 8, 7, 4

Note: Position of each element is found by:

$$h(k_i) = (3k_i+5) \bmod m = (3k_i+5) \% m$$

You Try 2. Hash Table. Linear Probing



Use the index numbers you found in “You Try 1”,
and apply linear probing technique.

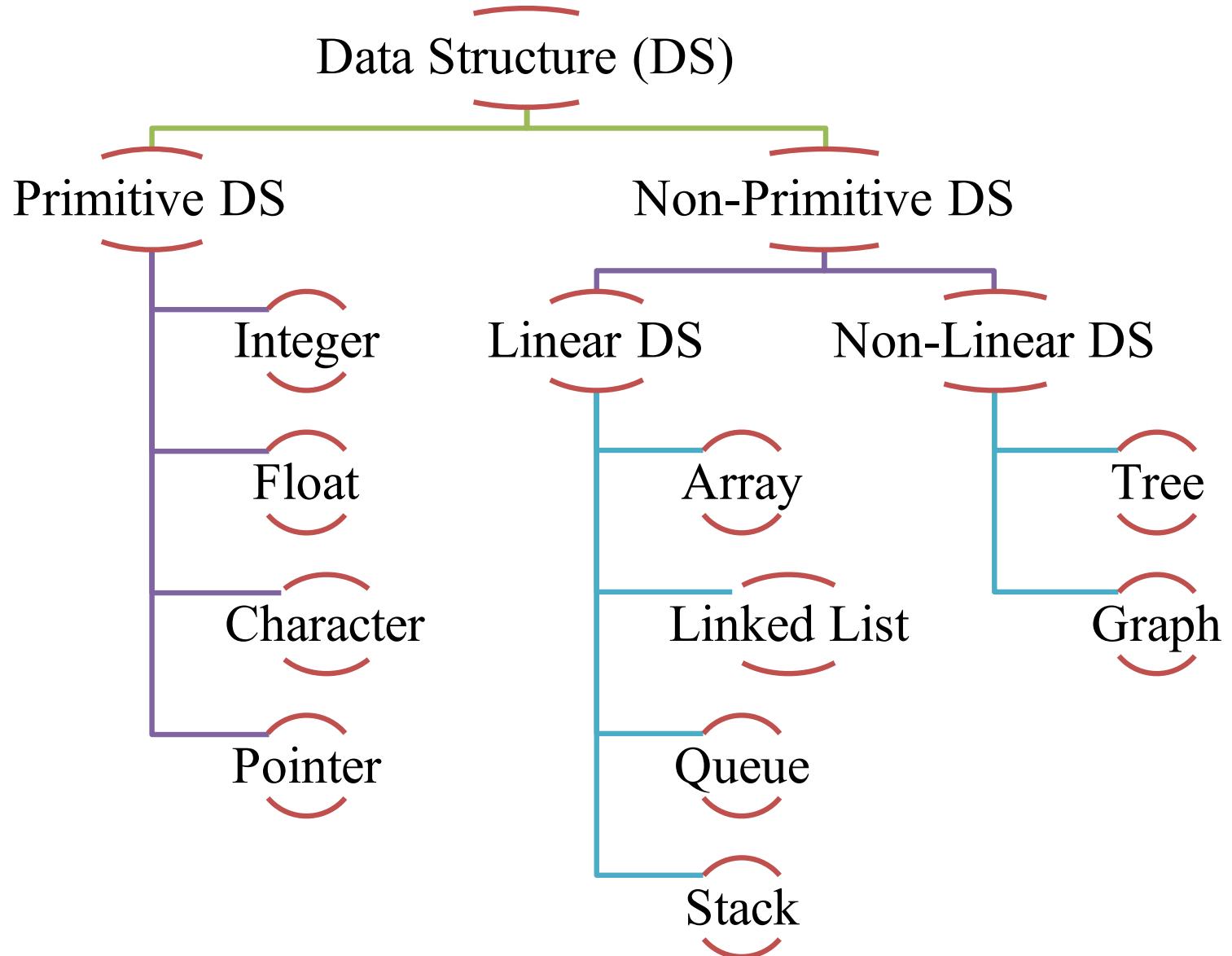
Hash Table: Time Complexity

	Worst		
	Search	Insertion	Deletion
Array	$O(n)$	$O(n)$	$O(n)$
Singly linked List	$O(n)$	$O(n)$	$O(n)$
Doubly lined List	$O(n)$	$O(n)$	$O(n)$
Binary Search Tree	$O(n)$	$O(n)$	$O(n)$
AVL Tree	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$
Hash Table	$O(n)$	$O(n)$	$O(n)$
	$O(1)$	$O(1)$	$O(1)$

When no collision, not adjusting capacity

AVL Tree vs. Hash Table for Search

- If need to traverse all items in an order (i.e., ID order), use [AVL Tree](#).
- If need to search for a range of items (e.g., all items with id values between 100 and 300), use [AVL Tree](#).
- If need to guarantee worst case time complexity $O(\log n)$, use [AVL Tree](#).
- If need fast average search performance (one item to search at a time), use [Hash Table](#).



Lectures Done!



Readings/ Study Reference

Course Textbook: “*Data Structure and Algorithms in a Nutshell*” A. Wong et al.

Chapter 10. Hashing

Section 10.3. Collisions

The End of Lecture

Course: Data Structures and Algorithms

Instructor: Homeyra Pourmohammadali

You Try Questions and Solutions

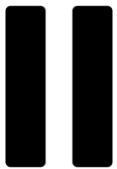
Hashing: Collisions

Example 1. Hash Table. Separate Chaining

Use division method and hash function of $h(k)=2k + 3$ to make a hash table of size $m=10$ using separate chaining technique for the given array: 3, 2, 9, 6, 11, 13, 7, 12

Note: Position of each element is found by:

$$h(k_i) = (2k_i+3) \bmod m = (2k_i+3) \% m$$



You Try 1. Hash Table. Separate Chaining

Use division method and hash function of $h(k)=3k + 5$ to make a hash table of size $m=10$ using separate chaining technique for the given array: 2, 3, 5, 6, 10, 11, 8, 7, 4

Note: Position of each element is found by:

$$h(k_i) = (3k_i+5) \bmod m = (3k_i+5) \% m$$

Example 2 Solution. Hash Table. Separate Chaining

Make a table to show keys and modulo

$$h(k_i) = (3k_i + 5) \bmod m = (3k_i + 5) \% m$$

Key	$(2k_i + 3) \% m$	index
2	$[(3*2) + 5] \% 10 = 11 \% 10$	1
3	$[(3*3) + 5] \% 10 = 14 \% 10$	4
5	$[(3*5) + 5] \% 10 = 20 \% 10$	0
6	$[(3*6) + 5] \% 10 = 23 \% 10$	3
10	$[(3*10) + 5] \% 10 = 35 \% 10$	5
8	$[(3*8) + 5] \% 10 = 29 \% 10$	9
7	$[(3*7) + 5] \% 10 = 26 \% 10$	6
4	$[(3*4) + 5] \% 10 = 17 \% 10$	7

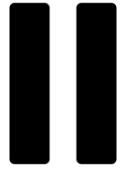
You Try 1 Solutions. Hash Table. Separate Chaining

0	5
1	2
2	
3	6
4	3
5	10
6	7
7	4
8	
9	8

Key	index
2	1
3	4
5	0
6	3
10	5
8	9
7	6
4	7

There is no collision

You Try 2. Hash Table. Linear Probing



Use the index numbers you found in “You Try 1”,
and apply linear probing technique.

You Try 2 Solution. Hash Table. Linear Probing

The result is the same as there is no collision.

0	5
1	2
2	
3	6
4	3
5	10
6	7
7	4
8	
9	8

The End of You Try Activity