

Introduction to Java

Data structure

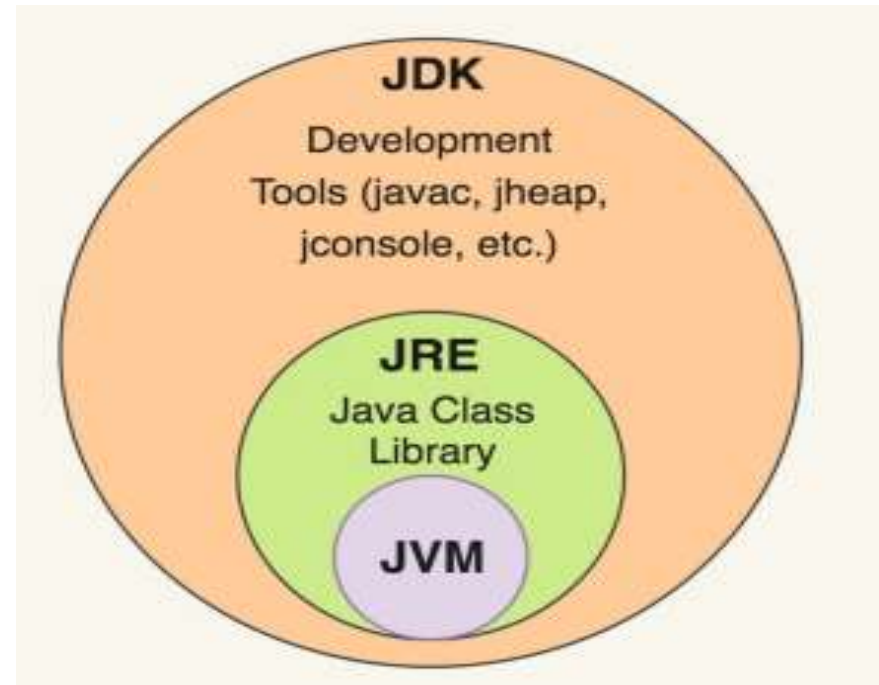
Fall 2023

Hashemite University

Dr. Mohammad Al-hammouri

Java Development Environment

- The Java Development Kit (JDK)
 - Is a cross-platformed software development environment that offers a collection of tools and libraries necessary for developing Java-based software applications
- JVM(Java Virtual Machine)
 - acts as a run-time engine to run Java applications. JVM is the one that actually calls the **main** method present in a Java code



Java Development Environment

- Java is a compiled language.
- Programs are compiled into byte-code executable files, which are executed through the Java virtual machine (JVM).
- The JVM reads each instruction and executes that instruction.
- A programmer defines a Java program in advance and saves that program in a text file known as source code.
- For Java, source code is conventionally stored in a file named with the **.java** suffix (e.g., demo.java) and the byte-code file is stored in a file named with a **.class** suffix, which is produced by the Java compiler.

Creating Java Application

Phase 1 (Creating a Program)

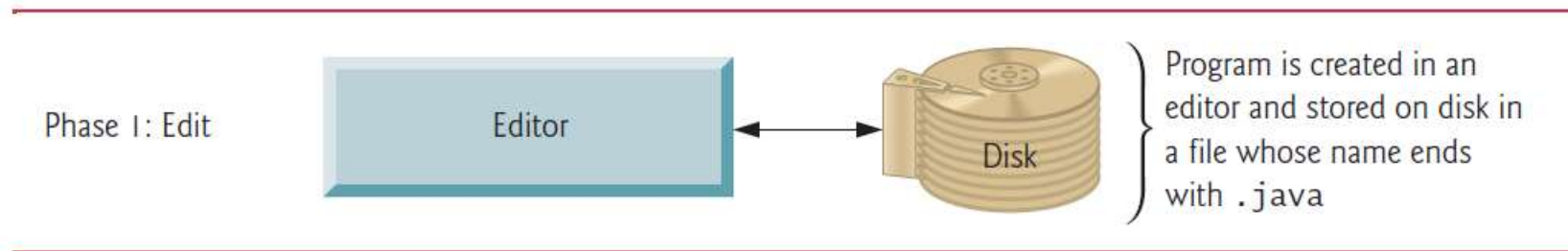


Fig. 1.6 | Typical Java development environment—editing phase.

Popular Java IDEs

Eclipse (www.eclipse.org)

NetBeans (www.netbeans.org)

IntelliJ IDEA (www.jetbrains.com)

Popular Editor

Vscode (Visual studio code)

Creating Java Application

Phase 2 (compile)

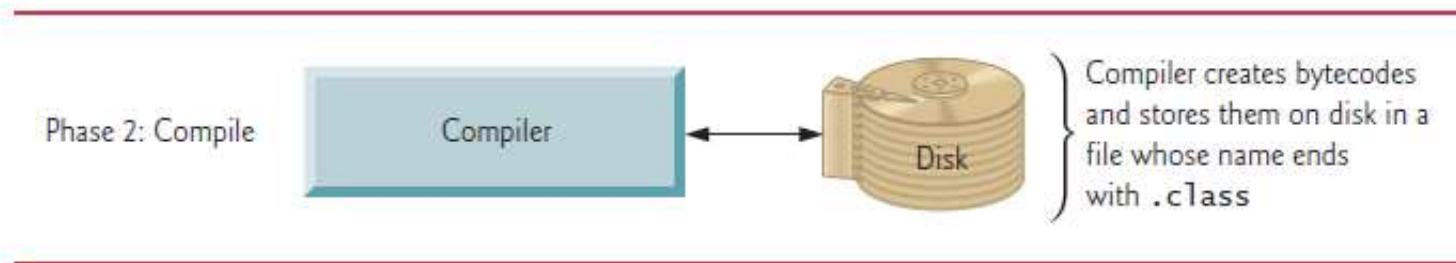


Fig. 1.7 | Typical Java development environment—compilation phase.

Compile command: **javac *.java**

Example: **javac Welcom.java** will generate **Welcom.class**

Creating Java Application

Phase 2 (compile)

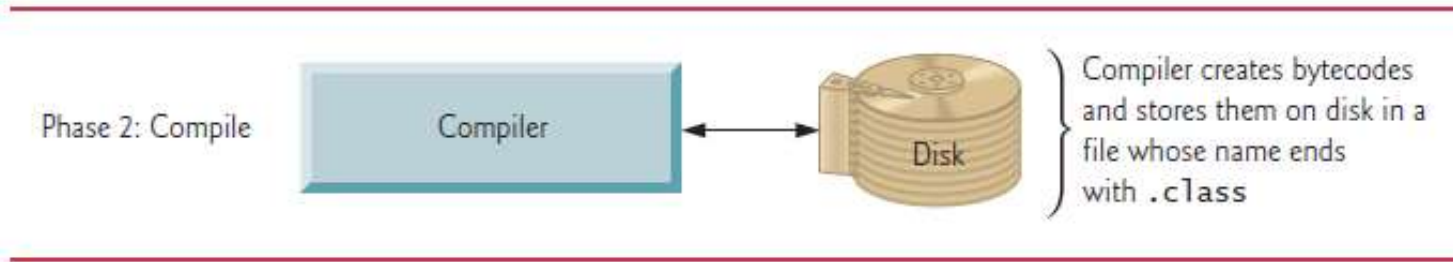


Fig. 1.7 | Typical Java development environment—compilation phase.

Unlike **machine-language** instructions, which are *platform dependent*

Java's bytecodes are **portable**—without recompiling the source code, the same bytecode instructions can execute on any platform containing a JVM

To execute a Java application, run: `java Welcom`

Creating Java Application

Phase 3 (Load)

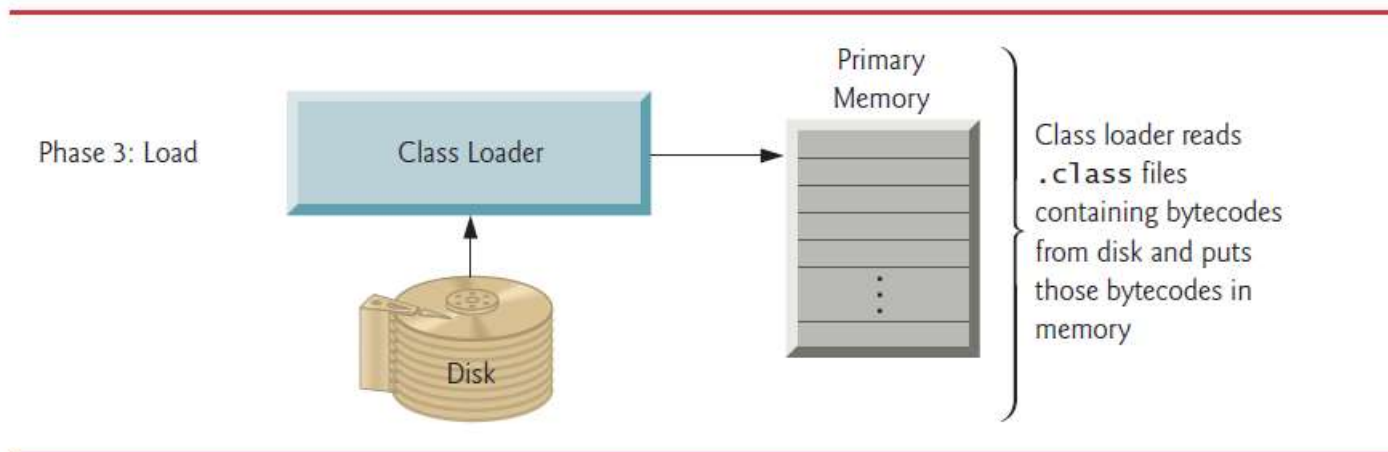


Fig. 1.8 | Typical Java development environment—loading phase.

Creating Java Application

Phase 4 (Verification)

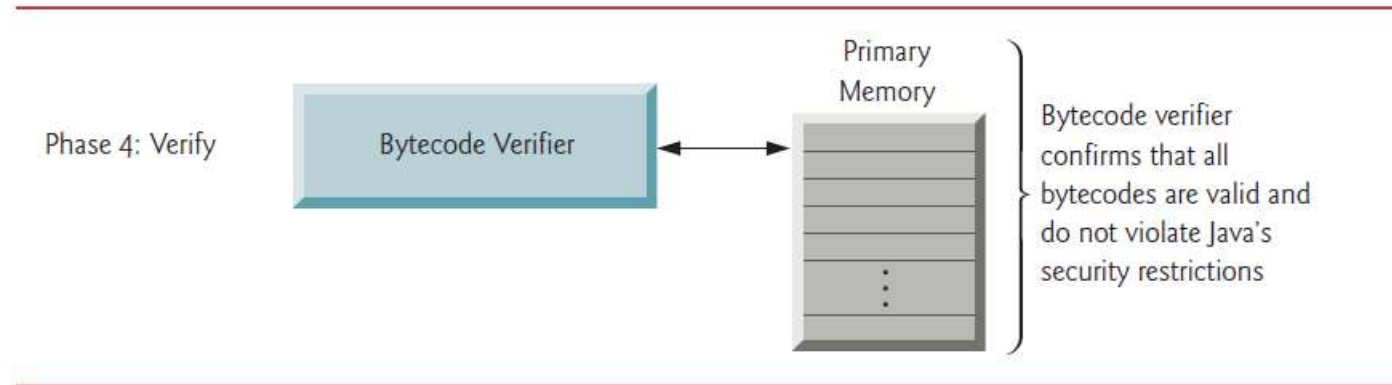


Fig. 1.9 | Typical Java development environment—verification phase.

The **bytecode verifier** examines their bytecodes to ensure that they're valid and do not violate Java's security restrictions

Creating Java Application

Phase 5 (Execution)

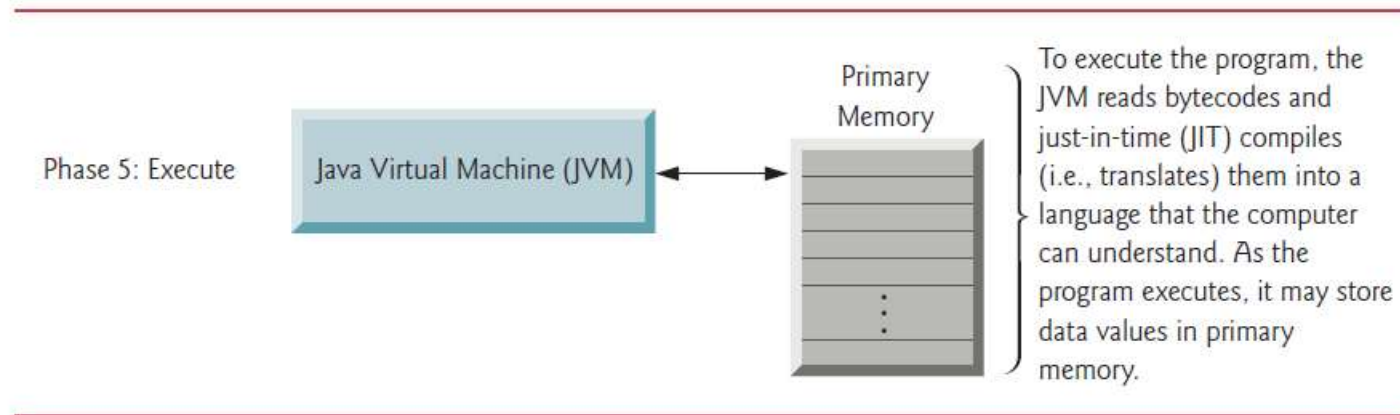
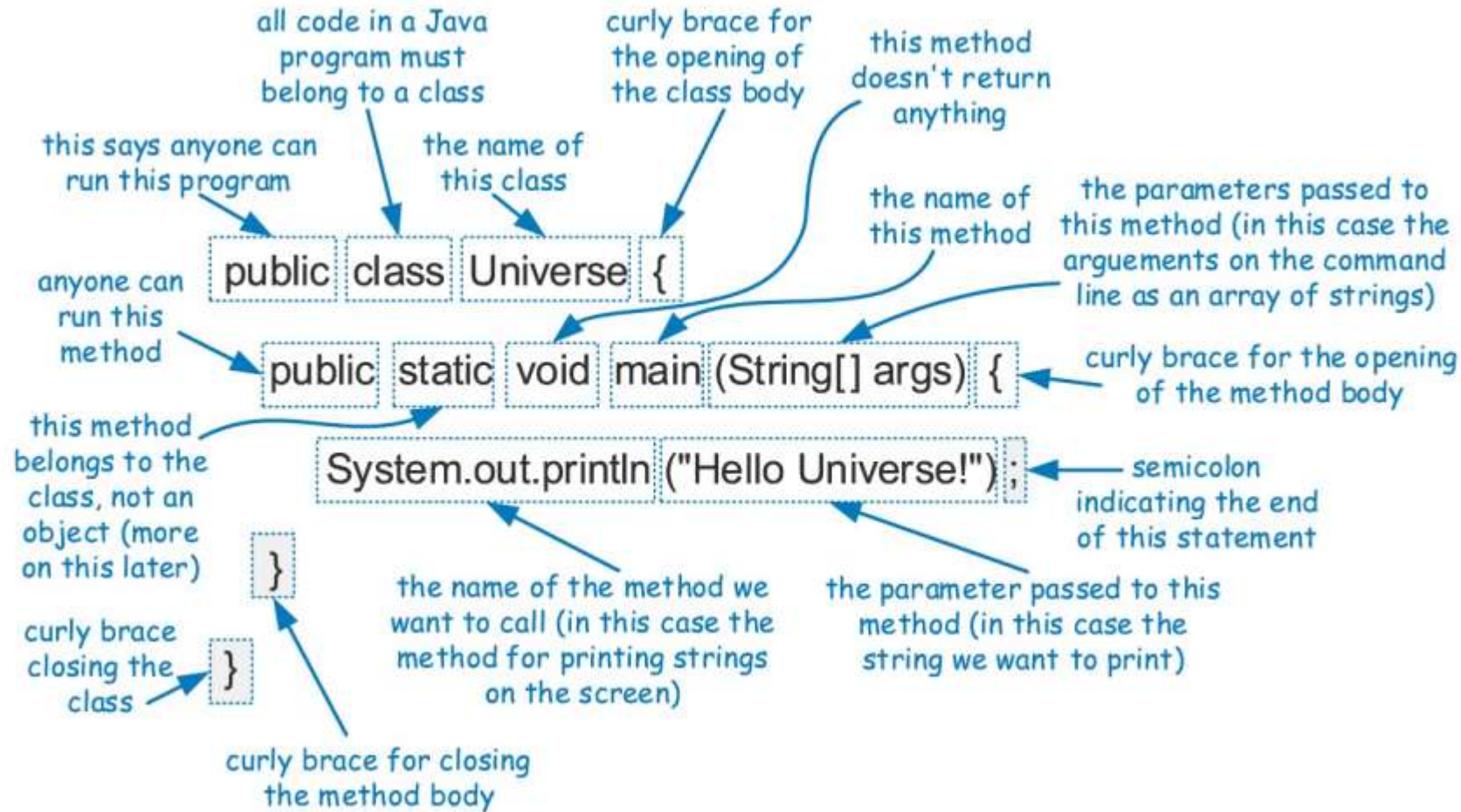


Fig. 1.10 | Typical Java development environment—execution phase.

JVM executes the program's bytecodes, thus performing the actions specified by the program

Simple Java Program

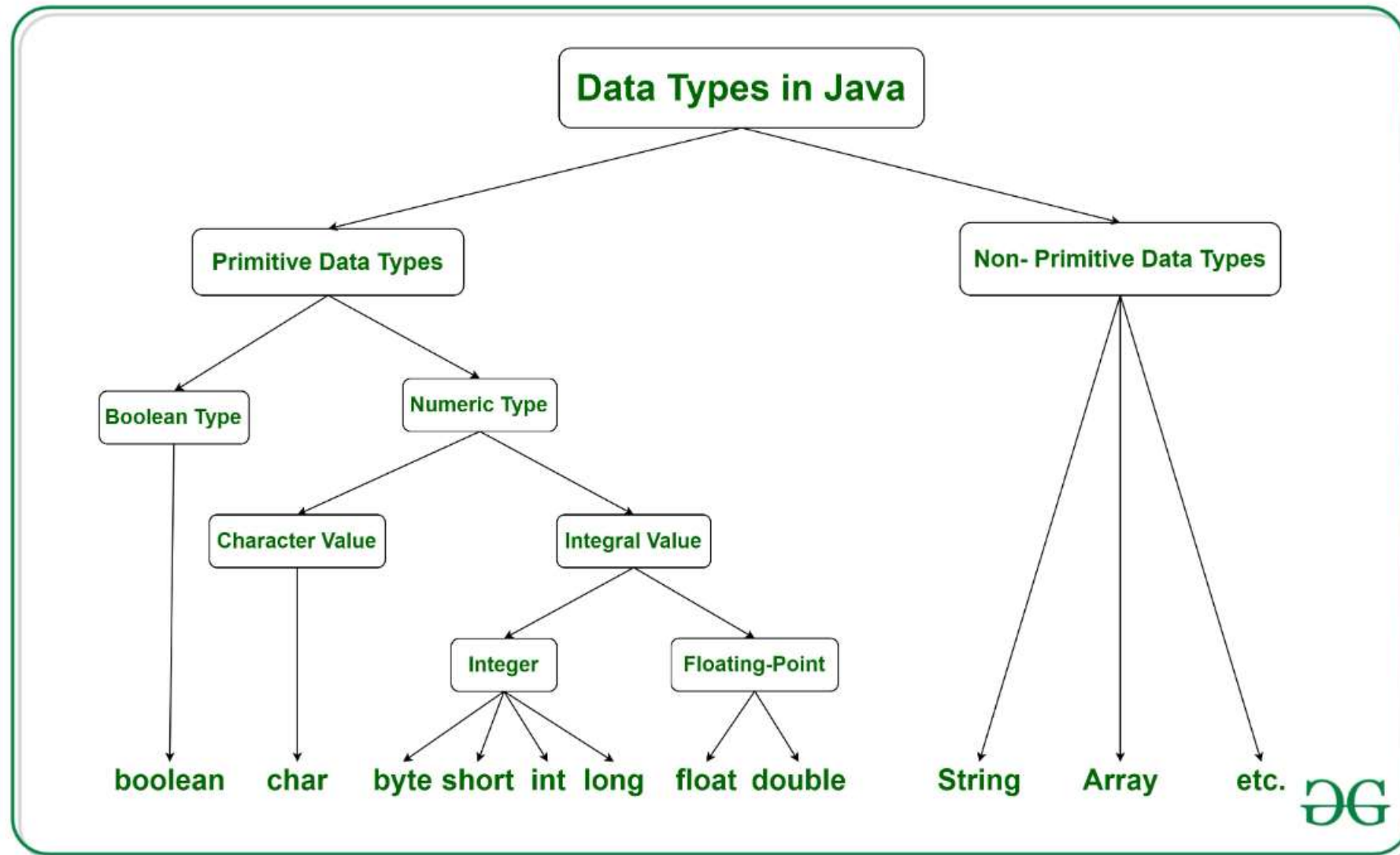


Identifiers

- The name of a class, method, or variable in Java is called an identifier, which can be any string of characters as long as it begins with a letter and consists of letters

Reserved Words				
abstract	default	goto	package	synchronized
assert	do	if	private	this
boolean	double	implements	protected	throw
break	else	import	public	throws
byte	enum	instanceof	return	transient
case	extends	int	short	true
catch	false	interface	static	try
char	final	long	strictfp	void
class	finally	native	super	volatile
const	float	new	switch	while
continue	for	null		

Data types in Java



Data types in Java: PRIMITIVE type

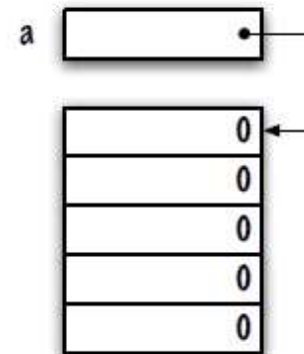
- A data item has a PRIMITIVE type if it represents a single value, which cannot be decomposed.
- Java has a number of pre-defined primitive data types. Examples are:
 - **int** (not Integer) represents integers (e.g. 3)
 - **double** represents “real” numbers (e.g. 3.0)
 - **char** represents single characters (e.g. 'A') 2 bytes instead of 1, use Unicode (over 64,000 characters) and not ASCII code
 - **boolean** represents Boolean values (e.g. true)

Data types in Java: Reference data types

- A reference data type in Java is a type of data that makes references to objects in memory rather than storing the values of those objects. Such as : strings, objects, arrays, etc.
- Array example:

```
int [] a;  
a = new array [ 5 ];
```

⇒

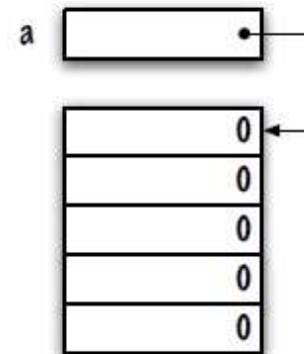


Data types in Java: Reference data types

- A reference data type in Java is a type of data that makes references to objects in memory rather than storing the values of those objects. Such as : strings, objects, arrays, etc.
- Array example:

```
int [] a;  
a = new array [ 5 ];
```

⇒



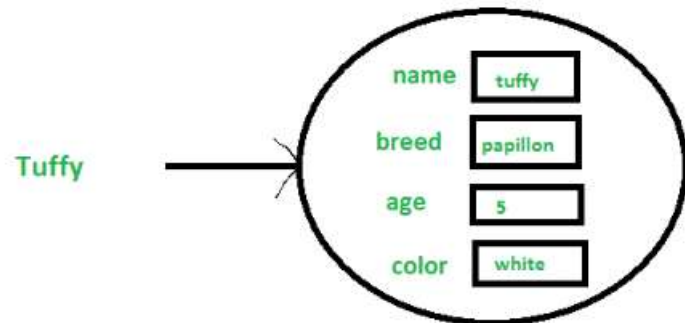
Data types in Java: Reference data types

- Strings are defined as an array of characters.

```
String s = "GeeksforGeeks";
```

- Classess and objects

```
Dog tuffy = new Dog("tuffy", "papillon", 5, "white");
```



Data types in Java: Reference data types

- **Classes** are known as reference types in Java, and a variable of that type is known as a reference variable.
- A **reference variable** is capable of storing the location (i.e., memory address) of an object from the declared class.
- In Java, a new object is created by using the **new** operator followed by a call to a constructor for the desired class.

For loop Example

- Compute the sum of an array of doubles:

```
public static double sum(double[ ] data) {  
    double total = 0;  
    for (int j=0; j < data.length; j++)    // note the use of length  
        total += data[j];  
    return total;  
}
```

For-Each Loop Example

- Compute the sum of an array of doubles:

```
public static double sum(double[ ] data) {  
    double total = 0;  
    for (double val : data)           // Java's for-each loop style  
        total += val;  
    return total;  
}
```

Simple Output

- Java provides a built-in static object, called `System.out`, that performs output to the “standard output” device, with the following methods:
- `System.out.print(String s)`: print string “s”
- `System.out.print(Object o)`: print object “o” using its `toString` method
- `System.out.println(String s)` followed by newline character
- `System.out.println(Object o)` followed by newline character

Simple Input

- There is also a special object, **System.in**, for performing input from the Java console window.
- A simple way of reading input with this object is to use it to create a **Scanner** object, using the expression

Scanner Input = new Scanner(System.in),

```
import java.util.Scanner; // Import the Scanner class

class Main {
    public static void main(String[] args) {
        Scanner myObj = new Scanner(System.in); // Create a Scanner object
        System.out.println("Enter username");

        String userName = myObj.nextLine(); // Read user input
        System.out.println("Username is: " + userName); // Output user input
    }
}
```

Wrapper Types

- Wrapper Types are classes that allow primitive data types to be accessed as objects.
- In Java, there are eight primitive data types: byte, short, int, long, float, double, char, and Boolean.
- Each of these primitive data types has a corresponding wrapper class, which allows them to be treated as objects
 - ex: **Integer**: Wraps **int**, **Double**: Wraps **double**, **Character**: Wraps **char**
- Wrapper classes are useful when you need to use objects instead of primitives
- Or when you want to take advantage of the object-oriented features of Java. you can use **Integer** to wrap an **int** and then call methods like `toString()` or `parseInt()` on it.

Wrapper Types

<i>Base Type</i>	<i>Class Name</i>	<i>Creation Example</i>	<i>Access Example</i>
boolean	Boolean	obj = new Boolean(true);	obj.booleanValue()
char	Character	obj = new Character('Z');	obj.charValue()
byte	Byte	obj = new Byte((byte) 34);	obj.byteValue()
short	Short	obj = new Short((short) 100);	obj.shortValue()
int	Integer	obj = new Integer(1045);	obj.intValue()
long	Long	obj = new Long(10849L);	obj.longValue()
float	Float	obj = new Float(3.934F);	obj.floatValue()
double	Double	obj = new Double(3.934);	obj.doubleValue()

```
int j = 8;
Integer a = new Integer(12);
int k = a;           // implicit call to a.intValue()
int m = j + a;       // a is automatically unboxed before the addition
a = 3 * m;           // result is automatically boxed before assignment
Integer b = new Integer("-135"); // constructor accepts a String
int n = Integer.parseInt("2013"); // using static method of Integer class
```


Object-oriented programming in Java

Data structure

Fall 2023

Hashemite University

Dr. Mohammad Al-hammouri

Overview

- **Object-Oriented Programming (OOP): attributes, instance variables and methods**

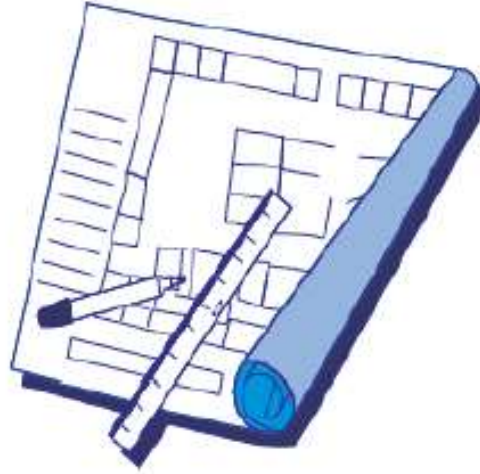
We analyze together a complex computer system, such as School system application to identify the main objects, their attributes and behaviors, as well as the associations between these objects. We discover together that the object-oriented programming makes concrete an abstract activity.

- **Abstractions** is a process of hiding the implementation details from the user and showing only the functionality to the user.
- We often compare an abstraction to a “black box” It can be used without worrying about its content.

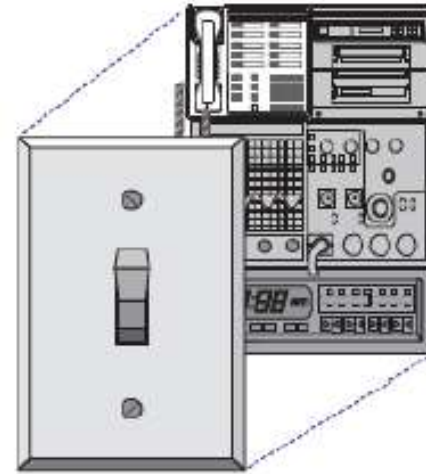
Object-Oriented Design Principles



Modularity



Abstraction



Encapsulation

Object-Oriented Design Principles

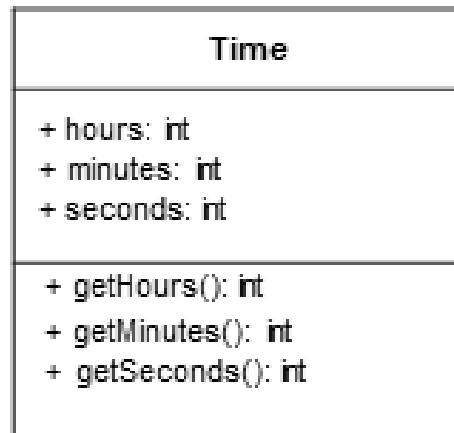
- **Abstractions** is to distill a system to its most fundamental parts.
(i.e., specifies the **type** of data stored, the **operations** supported on them, and the types of parameters of the operations)
 - **Abstractions** specifies what each operation does, but not how it does it.
- **Modularity**: concept of breaking down a software system into smaller, self-contained modules or components. Each module contains related functions, variables, classes.
- **Encapsulation**: bundling of data (attributes) and methods (functions) that operate on the data into a single unit known as a class
 - **Encapsulation** allows the internal representation of an object to be hidden from the outside world and restricts direct access to some of its components

Object-oriented programming

- The central concept of object-oriented programming is the **object**!
- An object has:
 - Properties (attributes) that define its state
 - Behaviours (methods)
- **Java** is object-oriented.
- The programmer defines new **data types** using classes and objects.
 - e.g. `class Account; Account myAccount`
- A class defines the characteristics (properties and behaviours) that are common to a set of objects.

Class

- Class is a data type (reference data type)
- A “class” can be used as a template to create objects with identical sets of attributes.
- A **class diagram** (from UML) is a box with three sections: the name of the class, the attributes, and the methods. The class **Time** is an example (class name should be singular and starts with capital letter)



Creating and Using Objects

- In Java, a new object is created by using the `new` operator followed by a call to a constructor for the desired class.
 - **Example:** `Time _time = new Time()`
- A `constructor` is a method that always shares the same name as its class. The new operator returns a reference to the newly created instance; the returned reference is typically assigned to a variable for further use.
- One of the primary uses of an object reference variable is to access the members of the class for this object.
- This access is performed with the dot “.” operator.
 - **Example:** `_time.hours , _time.getHours()`

Class Access Control

- The **public** class modifier indicates that all classes may access the defined member (attribute/method).
- The **private** class modifier indicates that access to a defined member of a class be granted only to code within that class.
- The **protected** class modifier indicates that access to the defined member is only granted to classes that are designated as subclasses of the given class through inheritance.

Unified Modeling Language (UML)

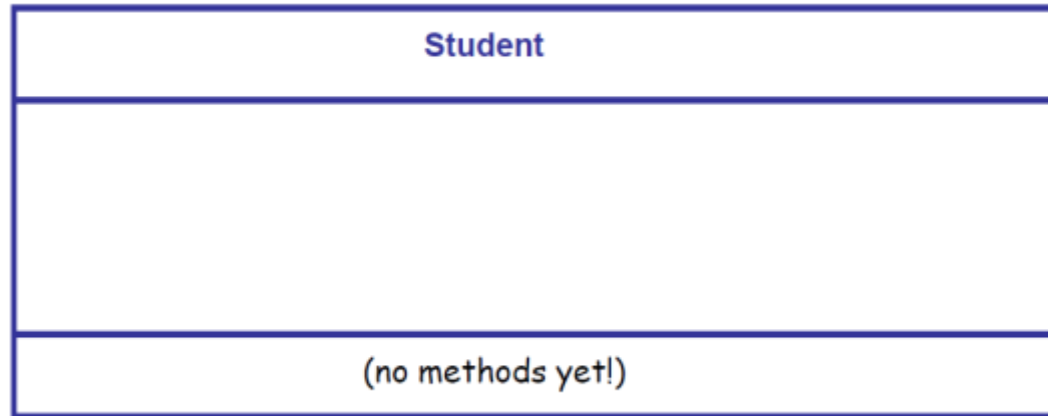
A **class diagram** has three portions.

1. The name of the class
2. The recommended instance variables
3. The recommended methods of the class.

class:	CreditCard	
fields:	<div>– customer : String</div> <div>– bank : String</div> <div>– account : String</div> <div>– limit : int</div> <div># balance : double</div>	
methods:	<div>+ getCustomer() : String</div> <div>+ getBank() : String</div> <div>+ charge(price : double) : boolean</div> <div>+ makePayment(amount : double)</div> <div>+ getAccount() : String</div> <div>+ getLimit() : int</div> <div>+ getBalance() : double</div>	

Example (Student class)

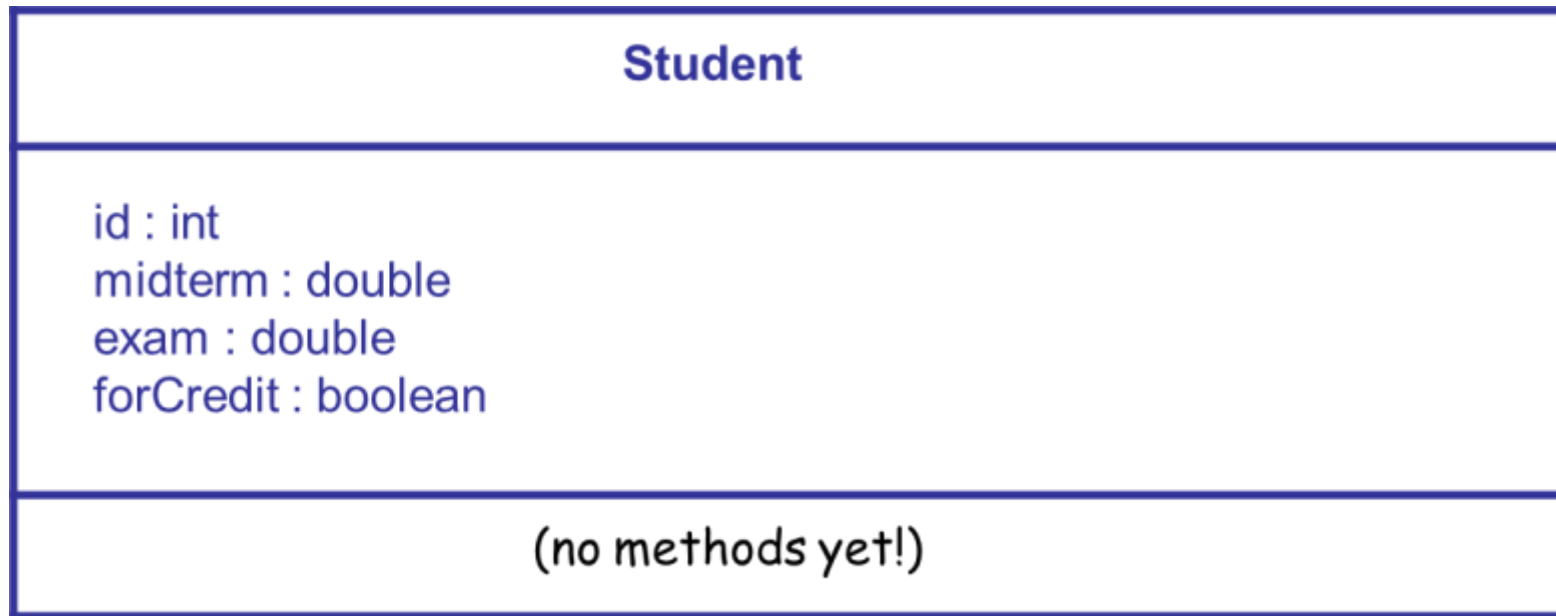
For each student, we want to store their ID number, their midterm score, their exam score, and whether or not the student is taking the course for credit. We shall deal with the final mark later.



Student class: Student Information

- How can we store all the information about each student in a course?
 - ID (student number) (integer)
 - midterm mark (real)
 - final exam mark (real)
 - is taking this course for credit or not (Boolean)
- **Solutions:**
 1. Each value is stored in a separate variable:
 - **Difficult to manipulate / exchange all information about a student.**
 2. Put all the values into an array:
 - **The variables are not of the same type.**
 3. ??

First version of a Student Class



Translation to Java

```
public class Student
{
    public int id;
    public double midterm;
    public double exam;
    public boolean forCredit;
    // methods
}
```

Translation to Java

```
public class Student
{
    public int id;
    public double midterm;
    public double exam;
    public boolean forCredit;
    public double finalMark
    // methods
    public void calculateFinalMark()
    {
        this.finalMark = 0.2 * this.midterm
        + 0.8 * this.exam;
    }

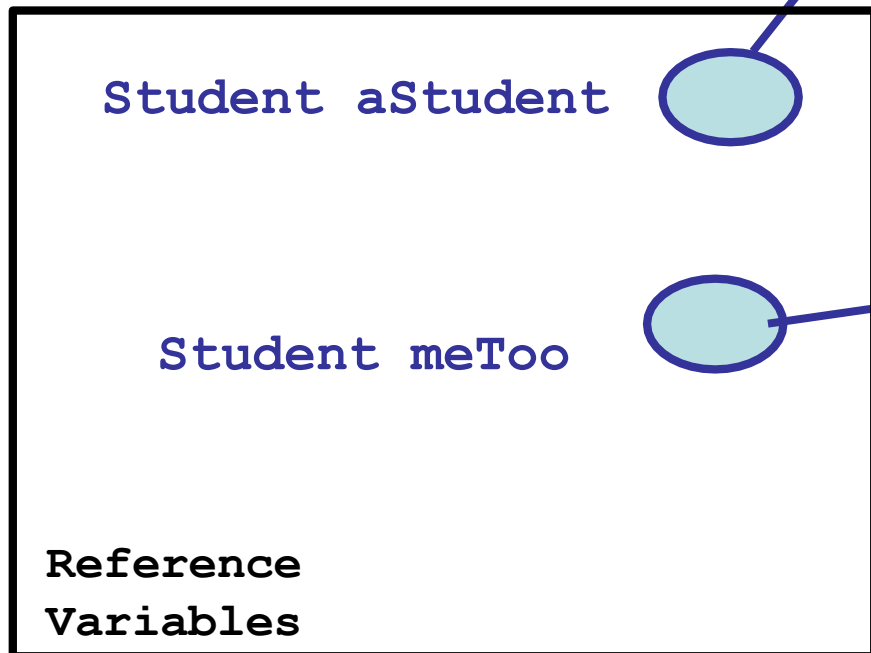
}

//In the main program
aStudent.calculateFinalMark();
```

How to use the class

format:
<class name>

(the underlining shows
that this is an **instance**
diagram)



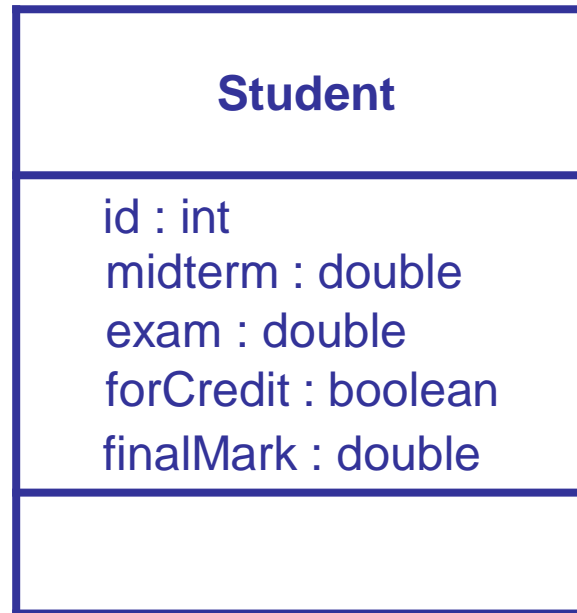
<u>Student.</u>	
id	1234567
midterm	60.0
exam	80.0
forCredit	true

<u>Student</u>	
id	1069665
midterm	73.0
exam	79.0
forCredit	false

Information hiding...Step 1

- Suppose we want to modify the `Student` class to keep the course final mark, which is 20% of the midterm mark plus 80% of the final mark.
 - We could add a field `finalMark` to our class.
- We want to make sure that
$$\text{finalMark} = (0.2 \times \text{midterm} + 0.8 \times \text{exam})$$
is always true for consistency.

Example: Student class- adding variables



Where did **this** come from?

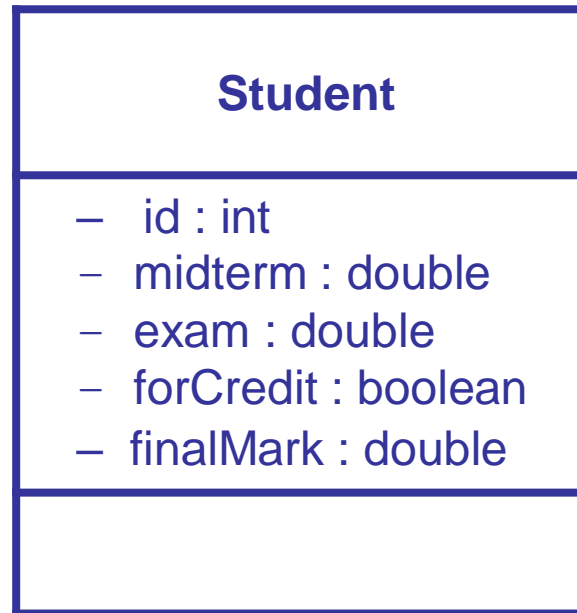
- **this** refers to the called object (the current object)
- During the call **aStudent.calculateFinalMark()**, **this** is a reference to **aStudent**
 - **this.exam** is **translated to aStudent.exam** which is a variable with 80.0 as its value
- During the call **meToo.calculateFinalMark()**, **this** is a reference to **meToo**
 - **this.exam** is **meToo.exam**, which is a variable with 79.0 as its value

Note: inside the class, we can access attributes directly without **this**

Information hiding.....

- Now suppose we want to make sure that only the finalMark to be changed only if the midterm and the final marks are changed? And we don't want to allow the main program to change it directly through the dot operator **aStudent.finalMark=100**

Example: Student class- hiding variables



- The “–” in front of the variable indicates that the attribute is **private**.
- By declaring a field to be private, **only methods declared inside the class** are allowed access to the field value (either for viewing the value, or changing the value).

Creating the Student Class

```
public class Student
{
    // were previously public
    private int id;
    private double midterm;
    private double exam;
    private boolean forCredit;
    private double finalMark;
    // methods
}
```

How do we use the class

- If we try the following:

```
Student aStudent = new Student();
```

```
aStudent.finaMark = 100 ; // error!
```

the compiler returns an error since access to **finaMark** is no longer allowed from outside the class.

Information Hiding

- This information hiding is also called data abstraction (hide implementation details) and also encapsulation).
- The private fields and methods cannot be accessed directly but only from methods in the class.
- You can define **a few public methods** to allow other parts of the program to access fields.
- The public methods represent the **interface** of the class relative to other parts of the program.

But how to provide access?

Now: No one can access id or any other variable, either to set it or to view it !!

- We can create additional access methods in the class

Student:

- “**accessor**” : requests to see the value of a private field (sometimes called getters).
- “**modifier**”: requests to modify the value of a private field (sometimes called setters).

Accessors and Modifiers

- An Accessor (a Get method)
 - A **public instance method** (called using a reference to an object such as: `double examMark=aStudent.getExam();`)
 - Returns the value of the field of the object;
 - Has no parameters;
 - Often named as follows: getFieldName (also called a getter method).
- A Modifier (a set method)
 - A public instance method (called using a reference to an object such as `aStudent.setExam(100);`)
 - Assigns a value to a field;
 - Accepts values in a parameter of the same type as the field;
 - Often called setFieldName (also called setter method).

Student class with Information Hiding

Student
<ul style="list-style-type: none">– id : int– midterm : double– exam : double– forCredit : boolean– finalMark : double
<ul style="list-style-type: none">+ getId() : int+ setId(newID : int)+ getMidterm() : double+ setMidterm(newMark: double)+ getExam() : double+ setExam(newMark: double)+ getForCredit() : boolean+ setForCredit(newValue : boolean)

Accessors and Modifiers

- Examples for the `forCredit` field in the class:

`boolean getForCredit()`

- method to return the value of `forCredit`
- the `+` indicates that the method has `public` visibility
- the return type is `boolean`, and in UML notation, appears at the end of the method.

`void setForCredit(newValue : boolean)`

- method to change the value of `forCredit`
- one parameter `newValue`, of type `boolean`
- `no` return value

Translation to Java

```
public class Student
{
    // Attributes

    private int id;
    private double midterm; private double
    exam; private boolean forCredit; private
    double finalMark;

    // Methods

    public int getId()
    {
        // insert code here
    }
    public void setId( int newId )
    {
        // insert code here
    }
    public double getMidterm()
    {
        // insert code here
    }
    public void setMidterm( double newMark )
    {
        // insert code here
    }
}
```

// continued from left side

```
    public double getExam()
    {
        // insert code here
    }
    public void setExam( double newMark )
    {
        // insert code here
    }
    public boolean getForCredit()
    {
        // insert code here
    }
    public void setForCredit( boolean newValue )
    {
        // insert code here
    }
    public double getFinalMark()
    {
        // insert code here
    }
    .....
} // end of class Student
```


Implementing Java accessors and modifiers

```
public class Student    // not all attributes/methods shown!
{
    // attribute
    boolean forCredit;
    // ... other attributes declared

    // accessor: return the requested value
    public boolean getForCredit()
    {
        return this.forCredit ;
    }

    // modifier: save the requested value in object's
    // attribute

    public void setForCredit( boolean newValue )
    {
        this.forCredit = newValue;
    }

    // ...other methods are similar
}
```

Calling Java Accessor and Modifier Methods

- Again, use the dot operator (.)
- The following code causes errors, why?

- **Student aStudent = new Student();**
 aStudent.id = 1234567;
 int myId = aStudent.id;
 System.out.println(myId);

// error here!
// error here!

- The compiler enforces the private access to **id**.
- Solution: Instead, use the modifier and accessor methods.

Student aStudent = new Student();
aStudent.setId(1234567);
int myId = aStudent.getId();
System.out.println(myId);

//ok!
//ok!

Back to Information Hiding

- To implement our strategy of hiding the `finalMark` field, we can do the following:
 1. Set the variable as private
 2. We will provide an accessor method for `finalMark`, but **NOT** a modifier method.
 3. We can provide a method `calculateFinalMark()` to recalculate the final mark if the midterm or exam marks are changed.
 4. The modifier methods `setMidterm()` and `setExam()` will call `calculateFinalMark()` so that they automatically update the final mark.
 5. We should also restrict access to `calculateFinalMark()` because it isn't meant for use outside the class.

Implementing Information Hiding

- The following implements our strategy where the final mark can only be changed by modifying the midterm or exam values.

```
public class Student
{
    .....
    // attributes and other methods would go here
    public void setMidterm(double newValue){
        this.midterm = newValue;
        this.calculateFinalMark();
    }

    public void setExam( double newValue){
        this.exam = newValue;
        this.calculateFinalMark();
    }

    public void calculateFinalMark(){
        this.finalMark = 0.2 * this.midterm + 0.8 * this.exam;
    }
}
```

Implementing Information Hiding

- The following implements our strategy where the final mark can only be changed by modifying the midterm or exam values.

```
public class Student
{
    // attributes and other methods would go here
    public void setMidterm( double newValue ){
        this.midterm = newValue; this.calculateFinalMark(
    );
    }

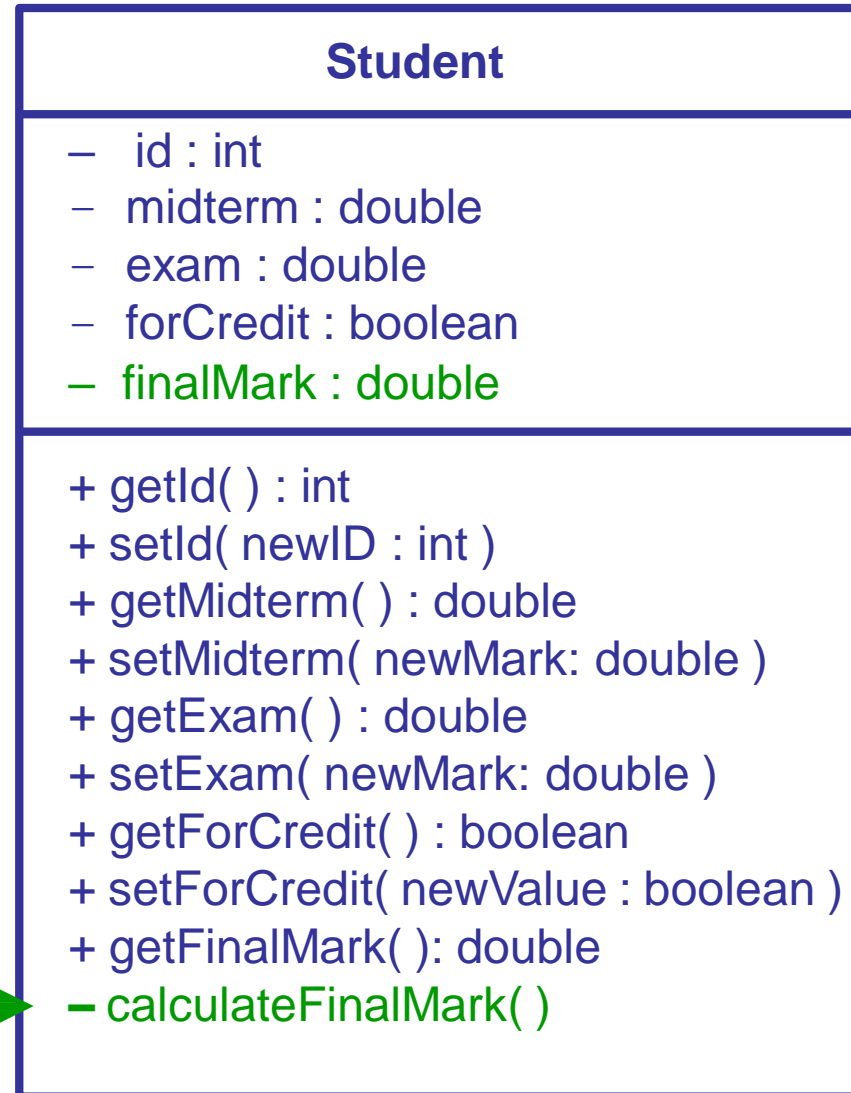
    public void setExam( double newValue ){
        this.exam = newValue;
        this.calculateFinalMark();
    }

    private void calculateFinalMark(){
        this.finalMark = 0.2 * this.midterm + 0.8 * this.exam;
    }
}
```

Student class with Information Hiding

no modifier
for this value

private
method



Translation to Java

```
public class Student
{
    // Attributes

    private int id;
    private double midterm;
    private double exam;
    private boolean forCredit;
    private double finalMark;

    // Methods

    public int getId()
    {
        // insert code here
    }
    public void setId( int newId )
    {
        // insert code here
    }
    public double getMidterm()
    {
        // insert code here
    }
    public void setMidterm( double newMark )
    {
        // insert code here
    }
    // continued at right
```

```
// continued from left side

    public double getExam()
    {
        // insert code here
    }
    public void setExam( double newMark )
    {
        // insert code here
    }
    public boolean getForCredit()
    {
        // insert code here
    }
    public void setForCredit( boolean newValue )
    {
        // insert code here
    }
    public double getFinalMark()
    {
        // insert code here
    }

    private void calculateFinalMark()
    {
        // insert code here
    }
} // end of class Student
```

Translation to Java

- The following implements our strategy where the final mark can only be changed by modifying the midterm or exam values.

```
public class Student
{
    // attributes and other methods would go here
    public void setMidterm( double newValue )
    {
        this.midterm = newValue;
        this.calculateFinalMark( );
    }

    public void setExam( double newValue )
    {
        this.exam = newValue;
        this.calculateFinalMark();
    }

    private void calculateFinalMark()
    {
        this.finalMark = 0.2 * this.midterm + 0.8 * this.exam;
    }
}
```


Object-orientation

- The approach we have taken with our student class is an “object oriented” approach:
 - We have a class that is a template for the creation of objects.
 - Student objects can be referred to as **INSTANCES** of the **CLASS** Student.
 - Object are instances of classes that have **instance methods** that use the field values for a specific object
 - e.g. `getFinalMark()` will have different results for different objects because `this.exam` is different for different objects
 - If you want the object to do something for you, you have to ask it by calling a method on that object.
 - That is, you can’t sneak inside an object from outside the class and change the private field values.
 - You also can’t call an instance method, without using an object reference:
`x = 3.0 + getFinalMark()` is meaningless. Whose final mark are we referring to?

Initialization of Objects

- When we create a new `Student`, we used “new”
`aStudent = new Student();`

The method `Student()` is called the default constructor of the class

- A constructor is a method that is executed when an object is created;
- A constructor has the same name as its class;
- A constructor can have arity 0, 1, 2, etc. `arities(formal parameters lists)`.
- , in the context
- A constructor has no return type.
- Since the constructor is called when the object is first created, a constructor generally serves to initialize the instance variables.

Constructors

- A constructor is a special method in a class used to create an object.
 - the same as the class;
 - no return type
 - can have 0, 1 or more formal parameters.
 - Can only be called once, when the object is created
- The formal parameters, if any, in a constructor are used to initialize the values of the object.
- Because there may be different ways to initialize an object, a class may have any number of constructors, distinguished from each other by **different arities (formal parameters lists)**.

Implementation in Java

- The following is a constructor that sets a value for all of the fields in the Student:

```
class Student
{
    // ... fields would be defined here ...
    public Student(int theId, double theMidterm, double theExam, boolean isForCredit)
    {
        this.id = theId;
        this.midterm = theMidterm;
        this.exam = theExam;
        this.forCredit = isForCredit;
    }
    // ... Other methods ...
}
```

- This constructor could be used as follows:

```
Student aStudent = new Student(1234567, 60.0, 80.0, true);
```

Constructors of class Student

- If provide the ID number and whether the student is taking the course we are doing course registrations, we may only want to for credit. (We don't know the student's marks yet!)
- We could **also** provide the following constructor:

```
public Student(int theID, boolean isForCredit )
{
    this.id = theID;
    this.midterm = 0.0;
    this.exam = 0.0;           // an initial(safe) value
    this.forCredit = isForCredit; // a initial (safe) value
}
```

- When there is more than one constructor, they must have parameter lists that can be distinguished by the **number**, **order**, and **type** of formal parameters.

Array Fields in Classes

```
public class Student
{
    private int id ;
    private double midterm ;
    private double exam ;
    private boolean forCredit;
    private double[] assignments;
    // methods
}
```

- The array reference variable **assignments** contains the value **null**.

Array field initialization

- Here is a constructor that creates and initializes an array in an object. The constructor has a parameter that is the number of assignments.

```
public Student( int numberOfAssignments )
{
    this.id = 0;
    this.midterm = 0.0; this.exam
    = 0.0 ; this.forCredit = false;
    this.assignments = new double[numberOfAssignments];

    // loop to initialize each item in array
    for ( index=0; index < numberOfAssignments; index = index+1 )
    {
        this.assignments[index] = 0.0;
    }
}
```

Example – Integer class

```
class Integer {  
    int value;  
  
    Integer(int v) {  
        value = v;  
    }  
  
    int getValue() {  
        return value;  
    }  
  
    Integer plus(Integer other){  
        int sum;  
        sum = value + other.value  
        return new Integer(sum);  
    }  
}
```


Example – Integer class

```
public class Test {  
    public static void main(String[] args) {  
        Integer a, b, c;  
  
        a = new Integer(10);  
  
        b = new Integer(5);  
  
        c = a.plus(b);  
  
        System.out.println(c.getValue());  
    }  
}
```

Example – COUNTER class

```
public class Counter {  
    private int count;           // a sim  
    public Counter() { }         // defau  
    public Counter(int initial) { count = initial; }  
    public int getCount() { return count; }  
    public void increment() { count++; }  
    public void increment(int delta) { count += delta; }  
    public void reset() { count = 0; }  
}
```

```
public class CounterDemo {  
    public static void main(String[] args) {  
        Counter c;               // declares a variable; no counter yet constructed  
        c = new Counter();        // constructs a counter; assigns its reference to c  
        c.increment();            // increases its value by one  
        c.increment(3);           // increases its value by three more  
        int temp = c.getCount();  // will be 4  
        c.reset();                // value becomes 0  
        Counter d = new Counter(5); // declares and constructs a counter having value 5  
        d.increment();            // value becomes 6  
        Counter e = d;            // assigns e to reference the same object as d  
        temp = e.getCount();       // will be 6 (as e and d reference the same counter)  
        e.increment(2);           // value of e (also known as d) becomes 8  
    }  
}
```

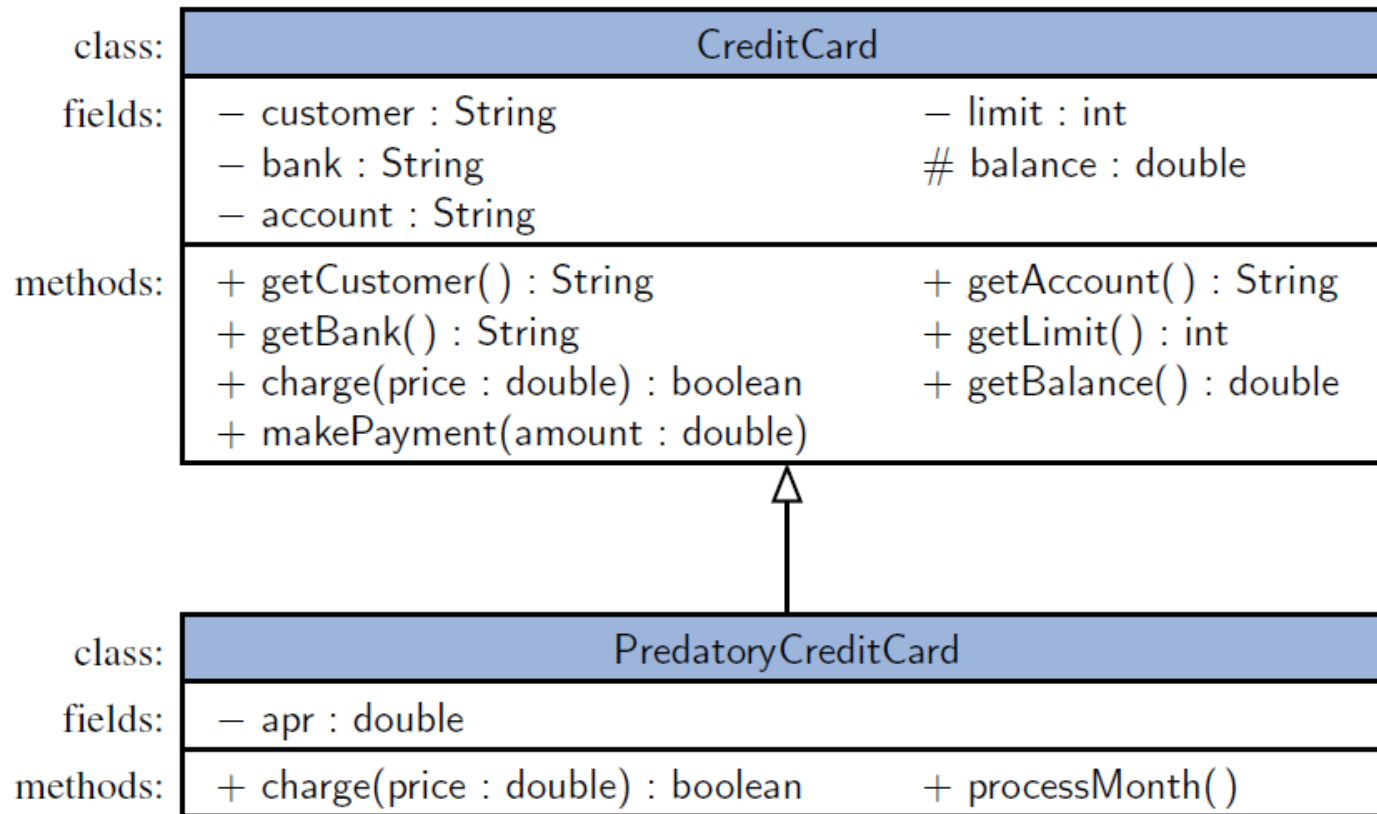
Inheritance

- A mechanism for a modular and hierarchical organization
- This allows a new class to be defined based upon an existing class as the starting point.
- The existing class is typically described as the **base class**, **parent class**, or **superclass**, while the newly defined class is known as the **subclass** or **child class**.
- There are two ways in which a subclass can differentiate itself from its superclass:
 - A subclass may specialize an existing behavior by providing a new implementation that overrides an existing method.
 - A subclass may also extend its superclass by providing brand new methods.

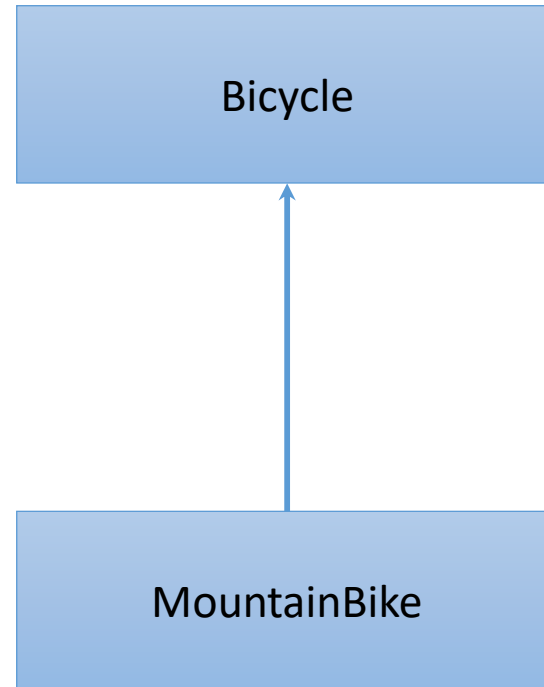
Inheritance

- A mechanism for a modular and hierarchical organization
- This allows a new class to be defined based upon an existing class as the starting point.
- The existing class is typically described as the **base class**, **parent class**, or **superclass**, while the newly defined class is known as the **subclass** or **child class**.
- There are two ways in which a subclass can differentiate itself from its superclass:
 - A subclass may specialize an existing behavior by providing a new implementation that overrides an existing method.
 - A subclass may also extend its superclass by providing brand new methods.

Inheritance Example



Inheritance Example



Inheritance Example- Bicycle class

```
// base class
class Bicycle {
    // the Bicycle class has two fields
    public int gear;
    public int speed;

    // the Bicycle class has one constructor
    public Bicycle(int gear, int speed)
    {
        this.gear = gear;
        this.speed = speed;
    }

    // the Bicycle class has three methods
    public void applyBrake(int decrement)
    {
        speed -= decrement;
    }
```

```
    public void speedUp(int increment)
    {
        speed += increment;
    }

    // toString() method to print info of Bicycle
    public String toString()
    {
        return ("No of gears are " + gear + "\n"
                + "speed of bicycle is " + speed);
    }
}
```

Inheritance Example- MountainBike class

```
// derived class
class MountainBike extends Bicycle {

    // the MountainBike subclass adds one more field
    public int seatHeight;

    // the MountainBike subclass has one constructor
    public MountainBike(int gear, int speed,
        int startHeight)
    {
        // invoking base-class(Bicycle) constructor
        super(gear, speed);
        seatHeight = startHeight;
    }
}
```

```
// the MountainBike subclass adds one more method
public void setHeight(int newValue)
{
    seatHeight = newValue;
}

// overriding toString() method
// of Bicycle to print more info
@Override public String toString()
{
    return (super.toString() + "\nseat height is "
        + seatHeight);
}
```


Inheritance Example- MountainBike class

```
// derived class
```

```
class MountainBike extends Bicycle {
```

```
    // the MountainBike subclass adds one more field
    public int seatHeight;
```

```
    // the MountainBike subclass has one constructor
```

```
    public MountainBike(int gear, int speed,
                        int startHeight)
```

```
    {
```

```
        // invoking base-class(Bicycle) constructor
```

```
        super(gear, speed);
```

```
        seatHeight = startHeight;
```

```
    }
```

```
    // the MountainBike subclass adds one more method
```

```
    public void setHeight(int newValue)
```

```
    {
```

```
        seatHeight = newValue;
```

```
    }
```

```
    // overriding toString() method
```

```
    // of Bicycle to print more info
```

```
    @Override public String toString()
```

```
    {
```

```
        return (super.toString() + "\nseat height is "
                + seatHeight);
```

```
    }
```

Inheritance Example- MountainBike class

Run | Debug

```
public static void main(String args[])
{
    MountainBike mb = new MountainBike(gear:3, speed:100, startHeight:25);
    //mb.speedUp(30);

    System.out.println(mb.toString());
}
```

Java Exceptions

- When executing Java code, different errors can occur: coding errors made by the programmer, errors due to wrong input, etc.
- When an error occurs, Java will normally stop and generate an error message.

```
public class Main {  
    public static void main(String[ ] args) {  
        int[] myNumbers = {1, 2, 3};  
        System.out.println(myNumbers[10]); // error!  
    }  
}
```

The output will be something like this:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10  
    at Main.main(Main.java:4)
```

Java Exceptions

- Java `try` and `catch`

```
public class Main {  
    public static void main(String[] args) {  
        try {  
            int[] myNumbers = {1, 2, 3};  
            System.out.println(myNumbers[10]);  
        } catch (Exception e) {  
            System.out.println("Something went wrong.");  
        }  
    }  
}
```

The output will be:

```
Something went wrong.
```

Java Exceptions

- The `throw` keyword

```
public class Main {  
    public static void main(String[] args) {  
        try {  
            int[] myNumbers = {1, 2, 3};  
            System.out.println(myNumbers[10]);  
        } catch (Exception e) {  
            System.out.println("Something went wrong.");  
        }  
    }  
}
```

The output will be:

```
Something went wrong.
```

Generics

- Java includes support for writing generic classes and methods that can operate on a variety of data types while often avoiding the need for explicit casts.
- Using Generics, it is possible to create classes that work with different data types

```
class Test<T> {  
    // An object of type T is declared  
    T obj;  
    Test(T obj) { this.obj = obj; }  
    // constructor  
    public T getObject() { return this.obj; }  
}
```

```
public static void main(String[] args)  
{  
    // instance of Integer type  
    Test<Integer> iObj = new Test<Integer>(15);  
    System.out.println(iObj.getObject());  
  
    // instance of String type  
    Test<String> sObj  
        = new Test<String>("GeeksForGeeks");  
    System.out.println(sObj.getObject());  
}
```

Generics

```
public class Pair<A,B> {  
    A first;  
    B second;  
    public Pair(A a, B b) {  
        first = a;  
        second = b;  
    }  
    public A getFirst() { return first; }  
    public B getSecond() { return second; }  
}
```

Object declaration

```
Pair<String,Double> bid;
```

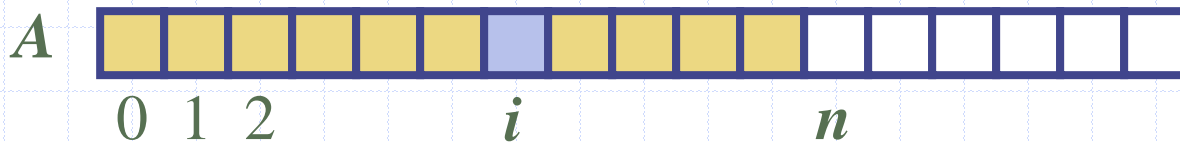
Presentation for use with the textbook **Data Structures and Algorithms in Java, 6th edition**, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Arrays



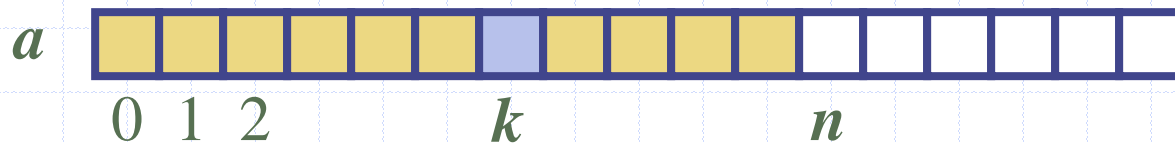
Array Definition

- An **array** is a sequenced collection of variables all of the same type. Each cell, in an array has an **index**, which uniquely refers to the value stored in that cell. The **cells** of an array, A , are numbered 0, 1, 2, and so on.
- Each value stored in an array is often called an **element** of that array.



Array Length and Capacity

- Since the length of an array determines the maximum number of things that can be stored in the array, we will sometimes refer to the length of an array as its *capacity*.
- In Java, the length of an array named a can be accessed using the syntax **$a.length$** .
- Thus, the cells of an array, a , are numbered 0, 1, 2, and so on,, and the cell with index k can be accessed with syntax $a[k]$.



Declaring Arrays (first way)

- The first way to create an array is to use an assignment to a literal form when initially declaring the array, using a syntax as:

elementType[] *arrayName* = {*initialValue*₀, *initialValue*₁, ..., *initialValue*_{N-1}};

- The ***elementType*** can be any Java base type or class name, and *arrayName* can be any valid Java identifier. The initial values must be of the same type as the array.

Declaring Arrays (second way)

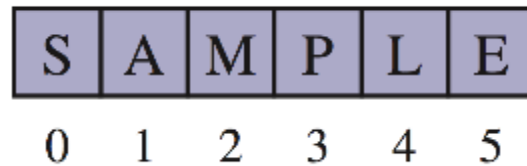
- The second way to create an array is to use the **new** operator.
 - However, because an array is not an instance of a class, we do not use a typical constructor. Instead we use the syntax:

elementType arrayName = **new** elementType[length]

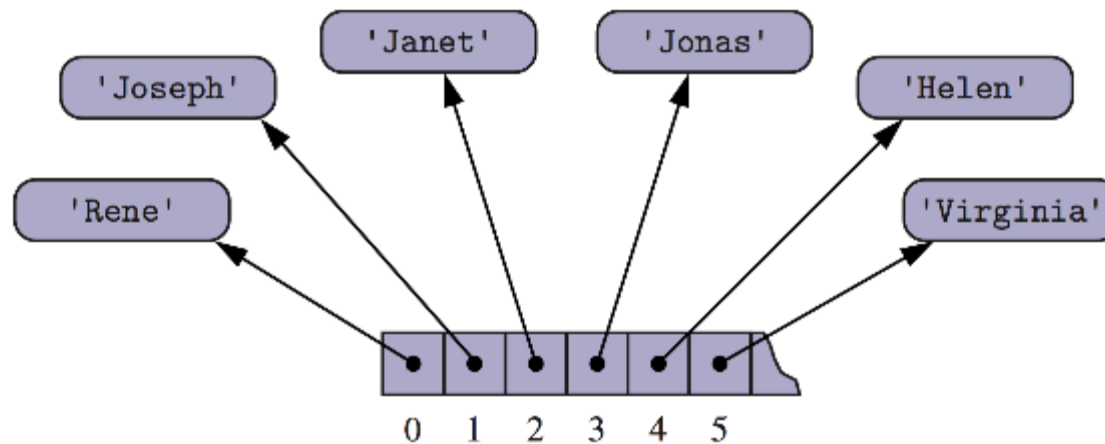
- *length* is a positive integer denoting the length of the new array.
- The **new** operator returns a reference to the new array, and typically this would be assigned to an array variable.

Arrays of Characters or Object References

- An array can store primitive elements, such as characters.



- An array can also store references to objects.



Java Example: Game Entries

- A game entry stores the name of a player and her best score so far in a game

```
1 public class GameEntry {
2     private String name;           // name of the person earning this score
3     private int score;             // the score value
4     /** Constructs a game entry with given parameters.. */
5     public GameEntry(String n, int s) {
6         name = n;
7         score = s;
8     }
9     /** Returns the name field. */
10    public String getName() { return name; }
11    /** Returns the score field. */
12    public int getScore() { return score; }
13    /** Returns a string representation of this entry. */
14    public String toString() {
15        return "(" + name + ", " + score + ")";
16    }
17 }
```

Java Example: Scoreboard

- Keep track of players and their best scores in an array, board
 - The elements of board are objects of class GameEntry
 - Array board is sorted by score

```
1  /** Class for storing high scores in an array in nondecreasing order. */
2  public class Scoreboard {
3      private int numEntries = 0;           // number of actual entries
4      private GameEntry[ ] board;          // array of game entries (names & scores)
5      /** Constructs an empty scoreboard with the given capacity for storing entries. */
6      public Scoreboard(int capacity) {
7          board = new GameEntry[capacity];
8      }
9      ... // more methods will go here
36 }
```

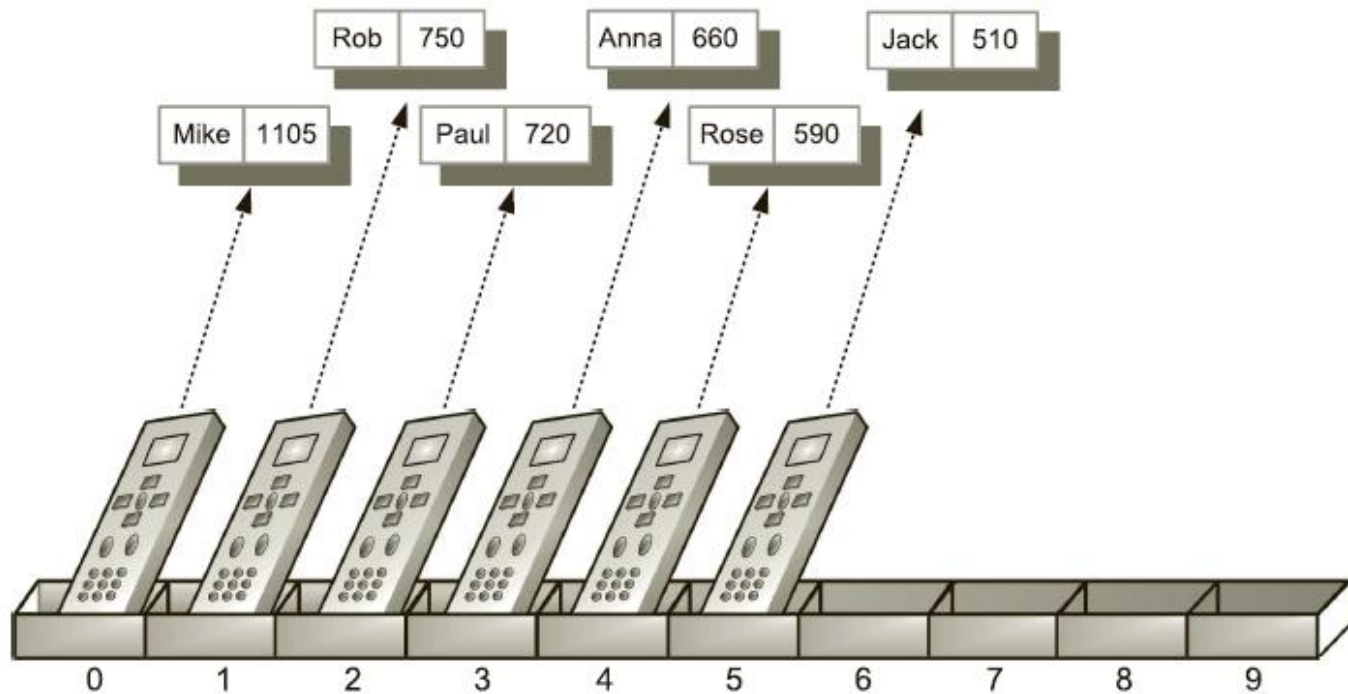
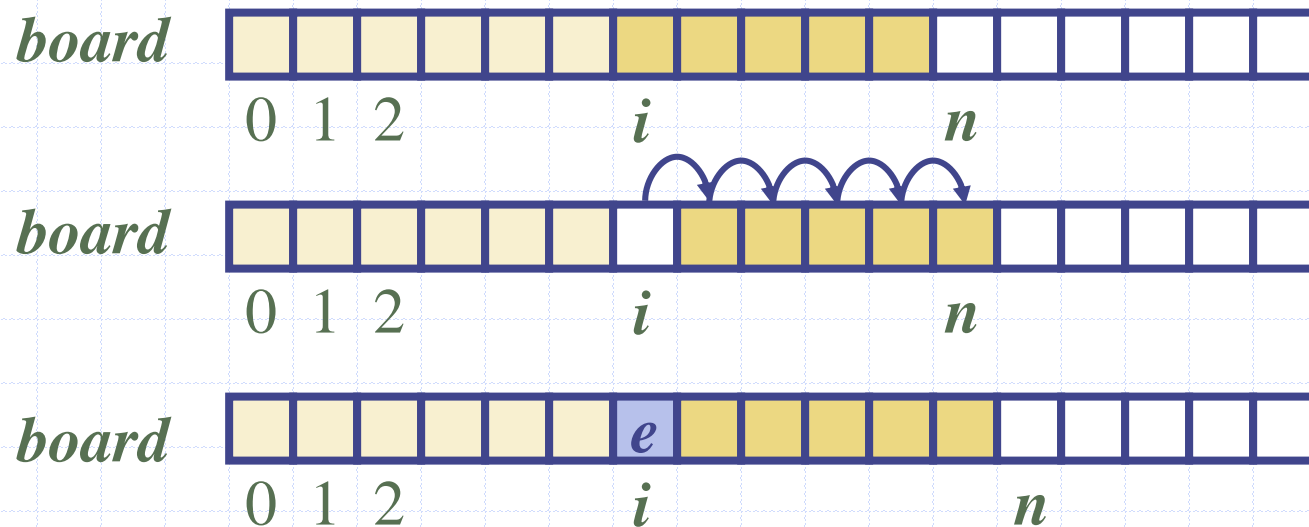


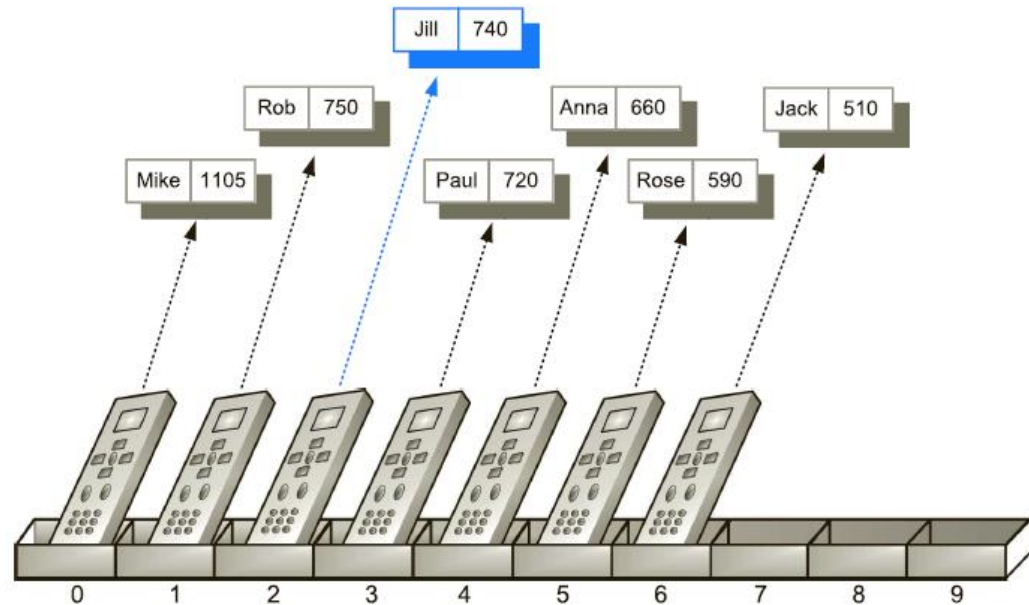
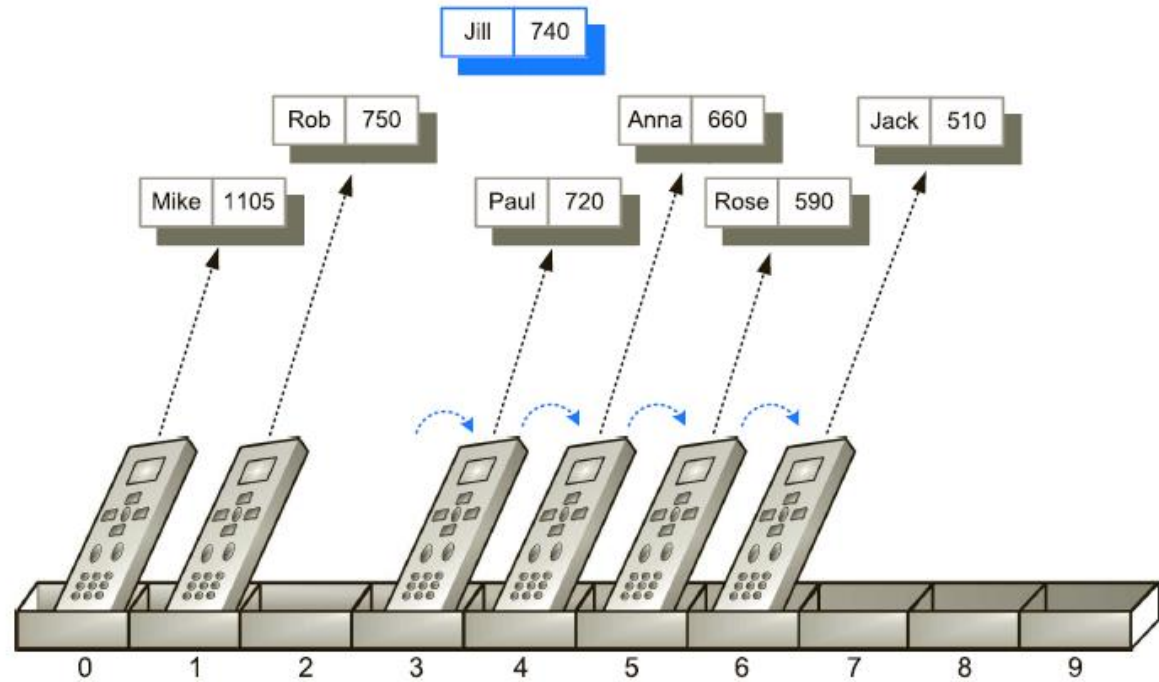
Figure 3.1: An illustration of an array of length ten storing references to six GameEntry objects in the cells with indices 0 to 5; the rest are **null** references.

Adding an Entry

- To add an entry e into array $board$ at index i



Add entry

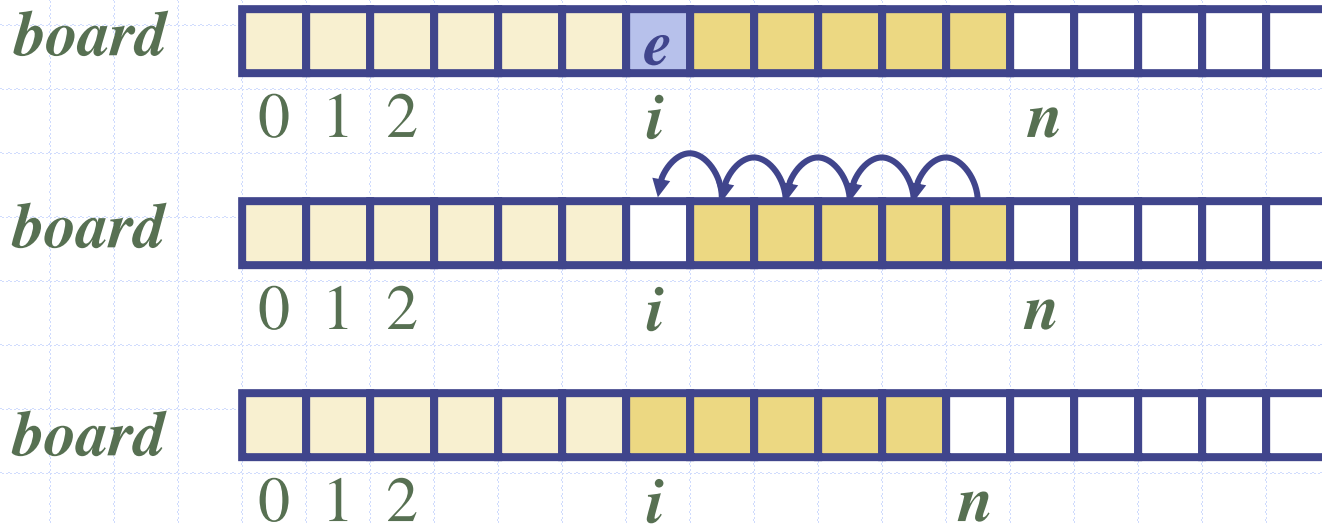


Java Example

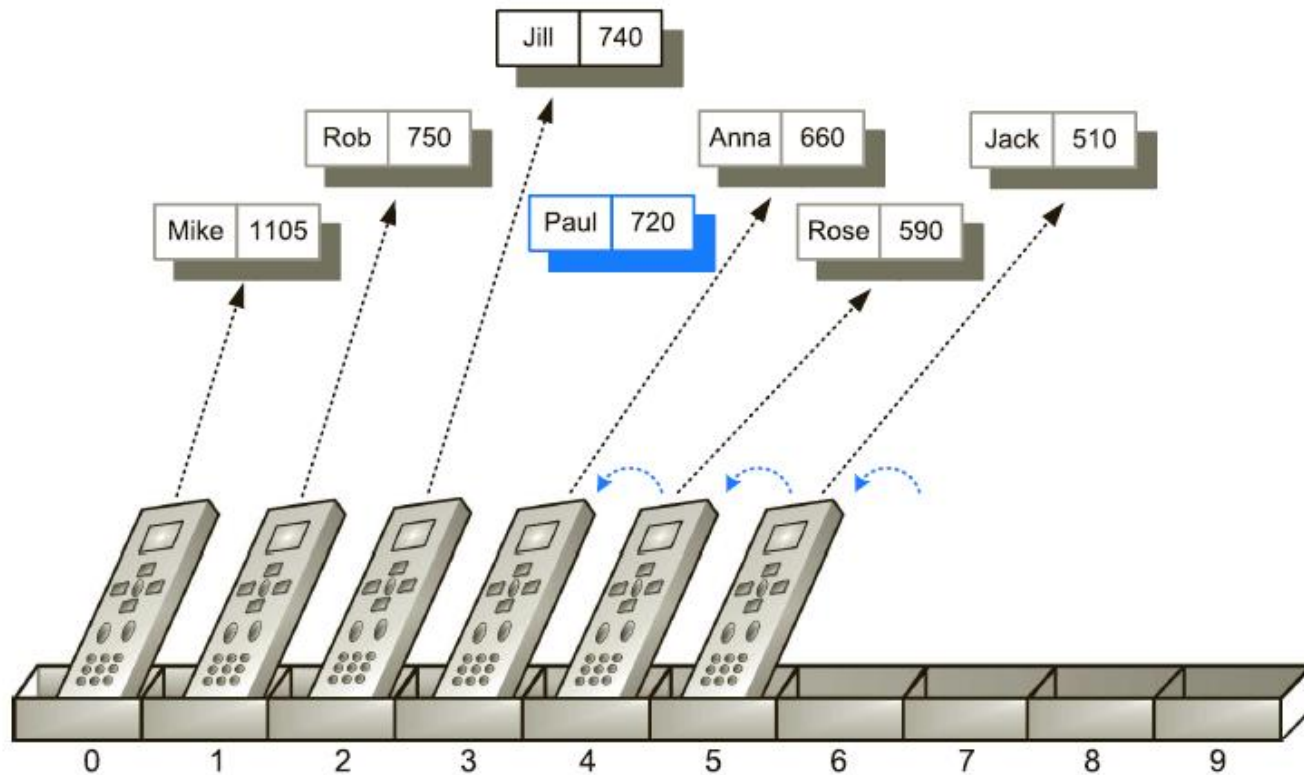
```
9  /** Attempt to add a new score to the collection (if it is high enough) */
10 public void add(GameEntry e) {
11     int newScore = e.getScore();
12     // is the new entry e really a high score?
13     if (numEntries < board.length || newScore > board[numEntries-1].getScore()) {
14         if (numEntries < board.length)           // no score drops from the board
15             numEntries++;                         // so overall number increases
16         // shift any lower scores rightward to make room for the new entry
17         int j = numEntries - 1;
18         while (j > 0 && board[j-1].getScore() < newScore) {
19             board[j] = board[j-1];               // shift entry from j-1 to j
20             j--;                                  // and decrement j
21         }
22         board[j] = e;                             // when done, add new entry
23     }
24 }
```

Removing an Entry

- To remove the entry e at index i , we need to fill the hole left by e by shifting backward the $n - i - 1$ elements $board[i + 1], \dots, board[n - 1]$



Remove entry

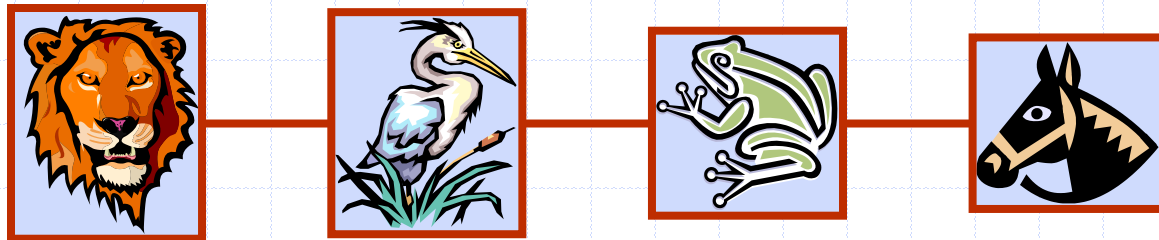


Java Example

```
25  /** Remove and return the high score at index i. */
26  public GameEntry remove(int i) throws IndexOutOfBoundsException {
27      if (i < 0 || i >= numEntries)
28          throw new IndexOutOfBoundsException("Invalid index: " + i);
29      GameEntry temp = board[i];           // save the object to be removed
30      for (int j = i; j < numEntries - 1; j++) // count up from i (not down)
31          board[j] = board[j+1];           // move one cell to the left
32      board[numEntries - 1] = null;         // null out the old last score
33      numEntries--;
34      return temp;                         // return the removed object
35  }
```

Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014
Edited by: Dr- Mohammad Al-hammouri

Singly Linked Lists

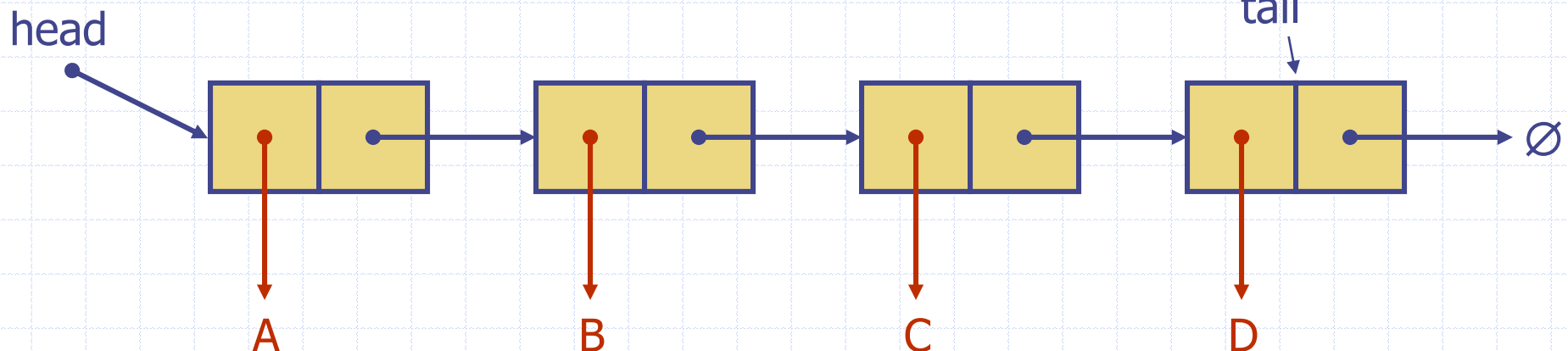
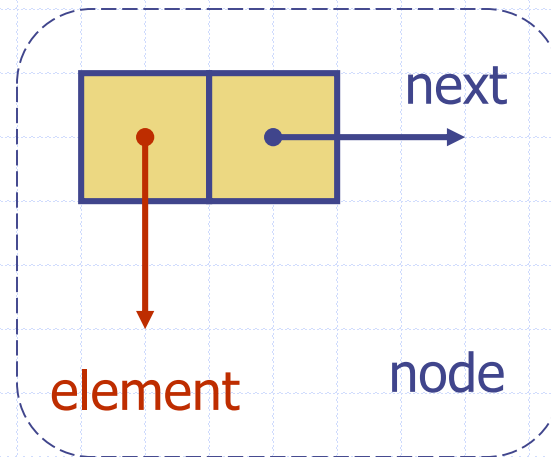


Array

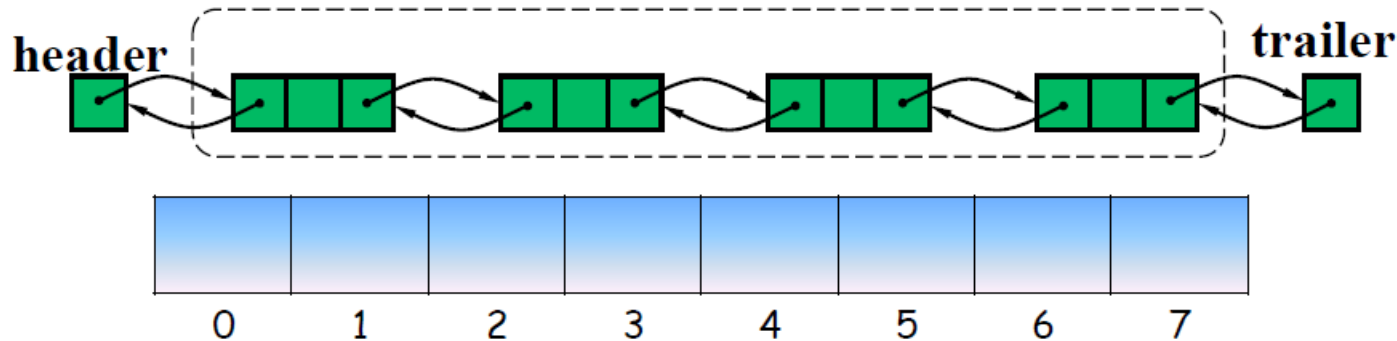
- Array are great for storing things in a certain order but it has some drawbacks
- The capacity of the array must be fixed when it is created
- The insertions and deletions at interior positions of an array can be time consuming if many elements must be shifted.
- We introduce a new data structure known as a linked list, which provides an alternative to an array-based structure

Singly Linked List

- A singly linked list is a concrete data structure consisting of a sequence of nodes, starting from a head pointer
- Each node stores
 - Element: refers to an object that is an element of the sequence
 - link to the next node



Linked List VS Array



Linked Structures	Arrays
<input checked="" type="checkbox"/> Dynamic Structure (never full)	<input checked="" type="checkbox"/> Static Structure
<input checked="" type="checkbox"/> No need to move elements for insertions and deletions	<input checked="" type="checkbox"/> Insertions and deletions require movements of objects
<input checked="" type="checkbox"/> No direct access to an element (necessary to traverse the structure)	<input checked="" type="checkbox"/> Allow direct access to an element

A Nested Node Class

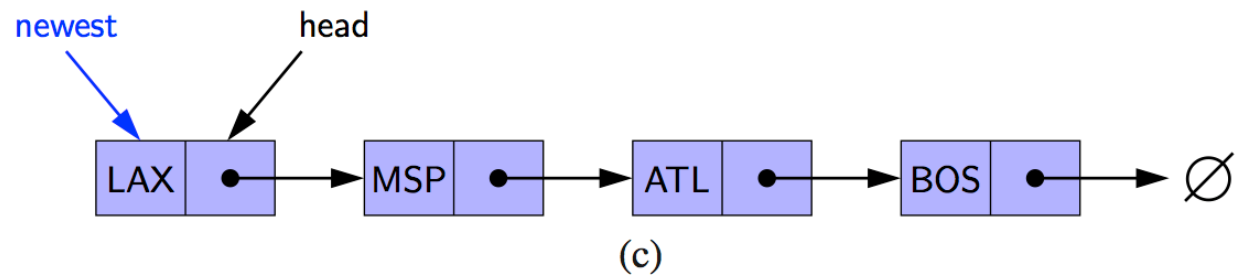
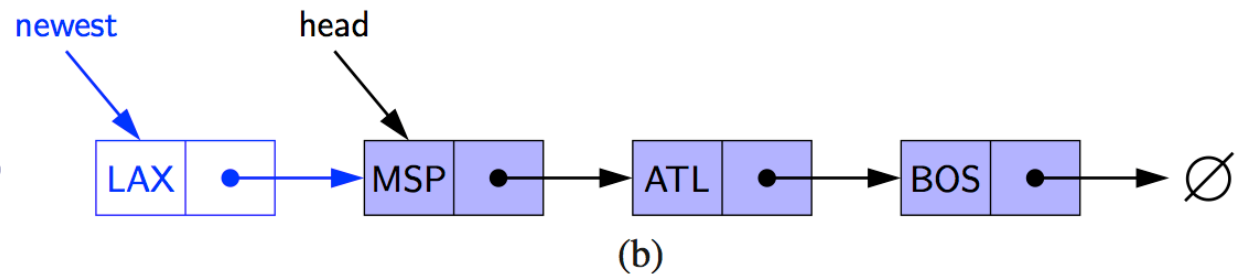
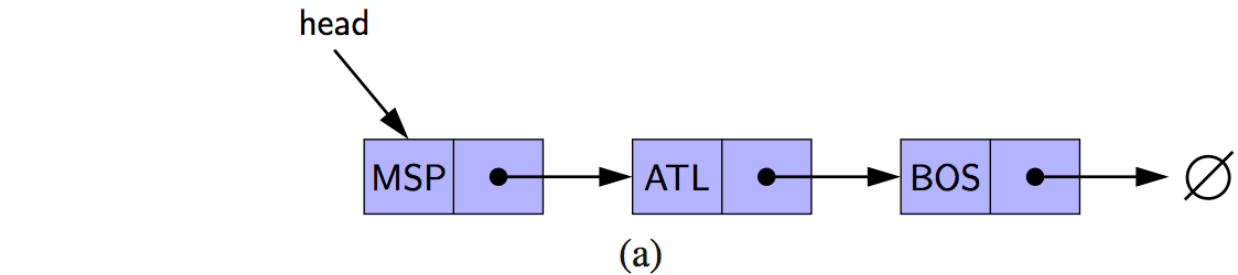
```
1  public class SinglyLinkedList<E> {
2      //----- nested Node class -----
3      private static class Node<E> {
4          private E element;           // reference to the element stored at this node
5          private Node<E> next;        // reference to the subsequent node in the list
6          public Node(E e, Node<E> n) {
7              element = e;
8              next = n;
9          }
10         public E getElement() { return element; }
11         public Node<E> getNext() { return next; }
12         public void setNext(Node<E> n) { next = n; }
13     } //----- end of nested Node class -----
    ... rest of SinglyLinkedList class will follow ...
```

Accessor Methods

```
1  public class SinglyLinkedList<E> {  
...  (nested Node class goes here)  
  
14  // instance variables of the SinglyLinkedList  
15  private Node<E> head = null;           // head node of the list (or null if empty)  
16  private Node<E> tail = null;          // last node of the list (or null if empty)  
17  private int size = 0;                  // number of nodes in the list  
18  public SinglyLinkedList() { }           // constructs an initially empty list  
19  // access methods  
20  public int size() { return size; }  
21  public boolean isEmpty() { return size == 0; }  
22  public E first() {                     // returns (but does not remove) the first element  
23      if (isEmpty()) return null;  
24      return head.getElement();  
25  }  
26  public E last() {                       // returns (but does not remove) the last element  
27      if (isEmpty()) return null;  
28      return tail.getElement();  
29  }
```

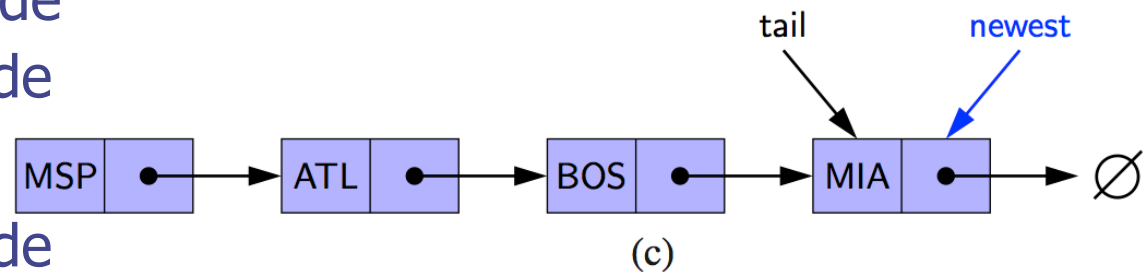
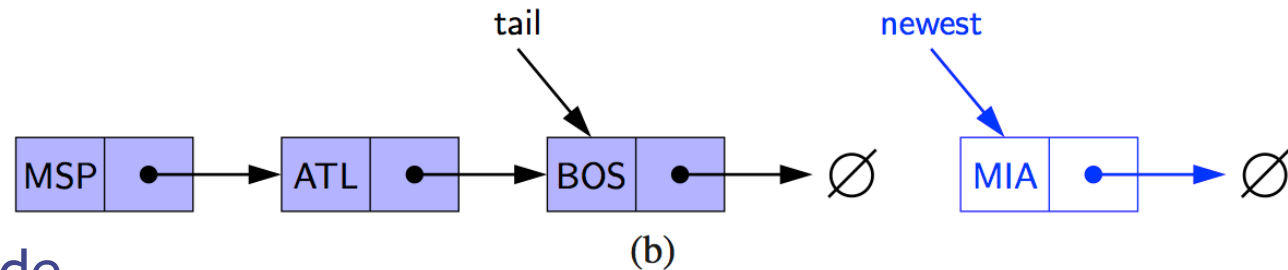
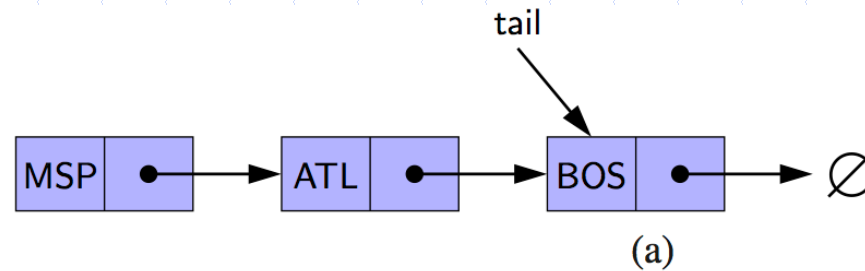
Inserting at the Head

- Allocate new node
- Insert new element
- Have new node point to old head
- Update head to point to new node



Inserting at the Tail

- Allocate a new node
- Insert new element
- Have new node point to null
- Have old last node point to new node
- Update tail to point to new node

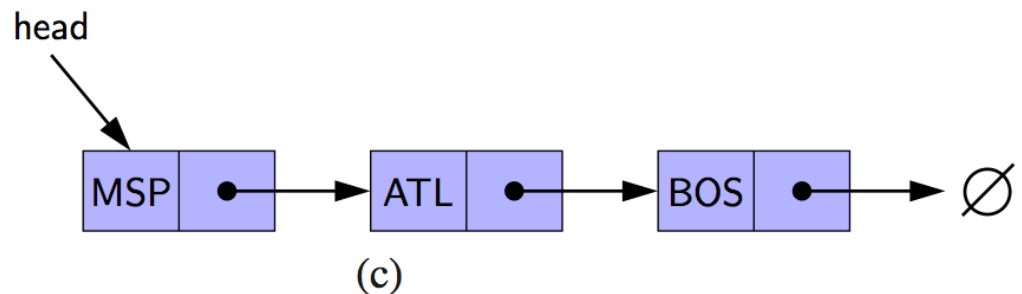
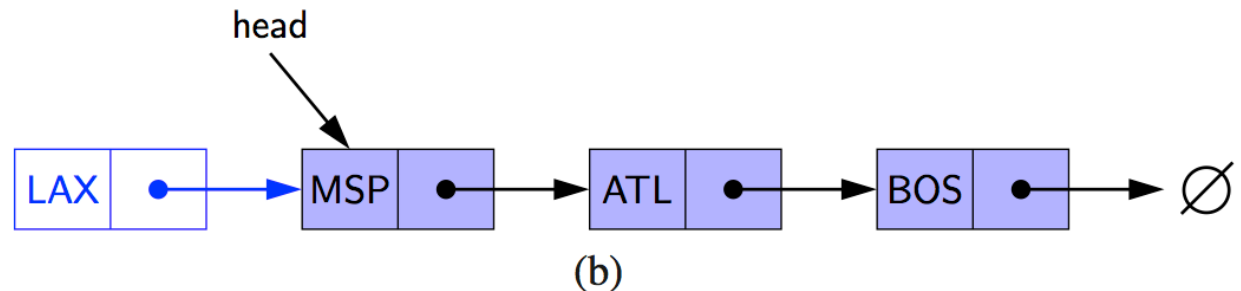
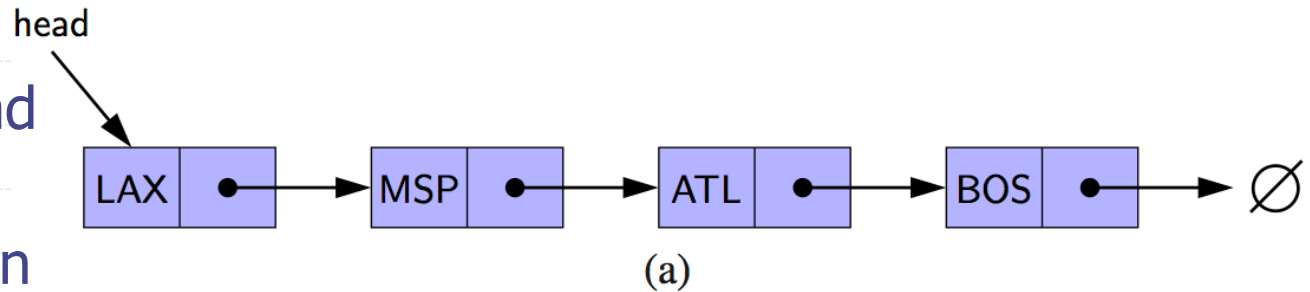


Java Methods

```
31  public void addFirst(E e) {           // adds element e to the front of the list
32      head = new Node<>(e, head);       // create and link a new node
33      if (size == 0)
34          tail = head;                 // special case: new node becomes tail also
35      size++;
36  }
37  public void addLast(E e) {            // adds element e to the end of the list
38      Node<E> newest = new Node<>(e, null); // node will eventually be the tail
39      if (isEmpty())
40          head = newest;                // special case: previously empty list
41      else
42          tail.setNext(newest);         // new node after existing tail
43      tail = newest;                    // new node becomes the tail
44      size++;
45  }
```

Removing at the Head

- Update head to point to next node in the list
- Allow garbage collector to reclaim the former first node

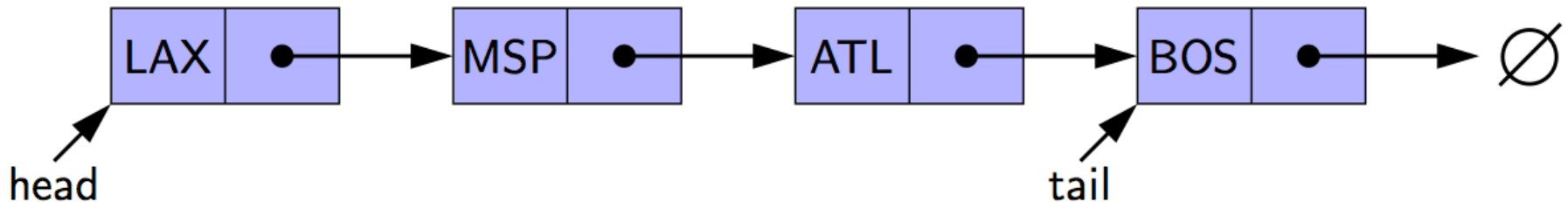


Java Method

```
46  public E removeFirst() {           // removes and returns the first element
47      if (isEmpty()) return null;    // nothing to remove
48      E answer = head.getElement();
49      head = head.getNext();         // will become null if list had only one node
50      size--;
51      if (size == 0)
52          tail = null;               // special case as list is now empty
53      return answer;
54  }
55 }
```

Removing at the Tail

- Removing at the tail of a singly linked list is not efficient!
- There is no constant-time way to update the tail to point to the previous node

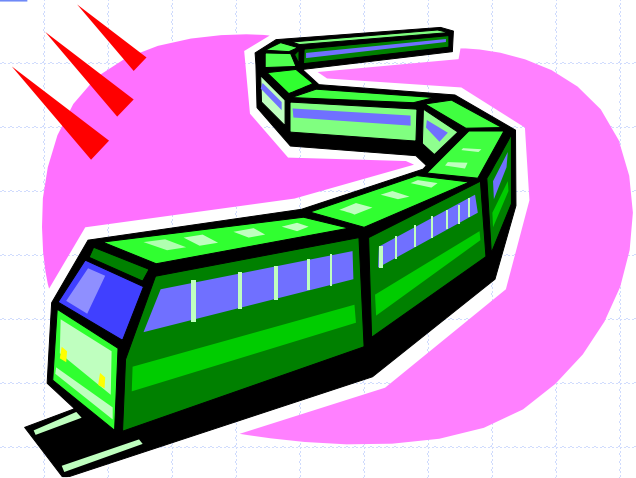


Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich,
R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Edited by: Dr. Mohammad Al-hammouri

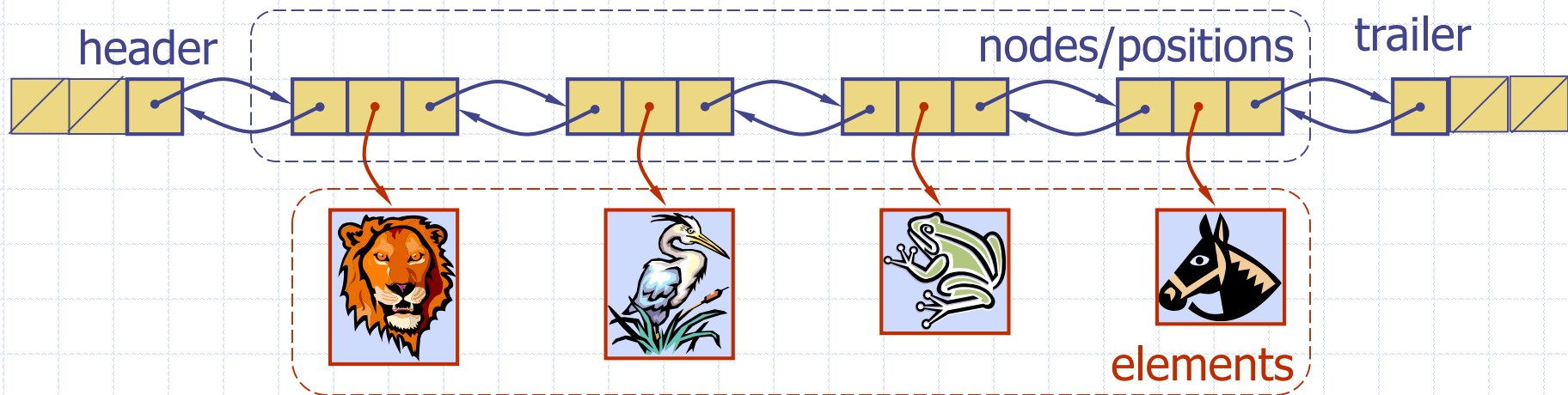
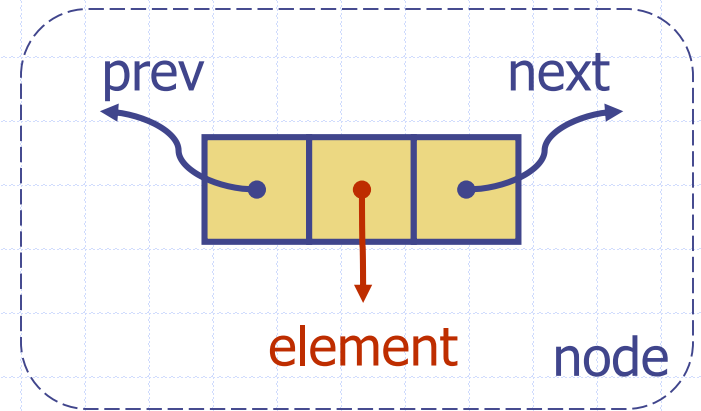
Hashemite University

Doubly Linked Lists



Doubly Linked List

- A doubly linked list can be traversed forward and backward
- Nodes store:
 - element
 - link to the previous node
 - link to the next node
- Special trailer and header nodes



Singly Linked Lists

□ Pros Singly Linked Lists:

- **Memory Efficiency:** Singly linked lists use less memory per node
- **Simplicity:** Easier to implement and use, especially for basic operations like insertion and deletion at the beginning.

□ Cons of Singly Linked Lists:

- **Backward Traversal:** Traversing in reverse order requires starting from the beginning of the list, making it less efficient.
- **Limited Functionality:** Certain operations (like deleting the previous node) require traversal from the beginning, impacting performance.

Double Linked Lists

□ Pros Doubly Linked list :

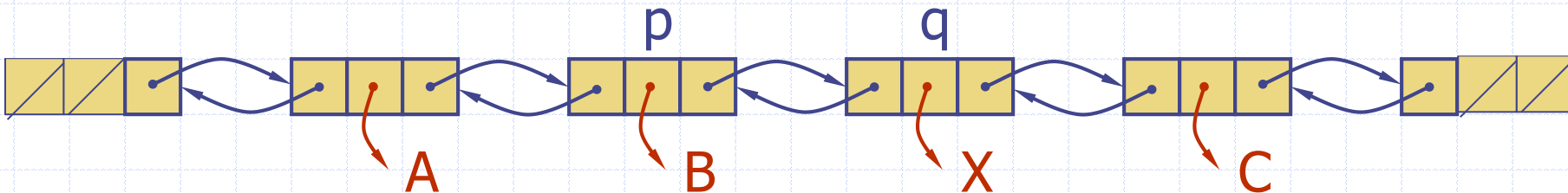
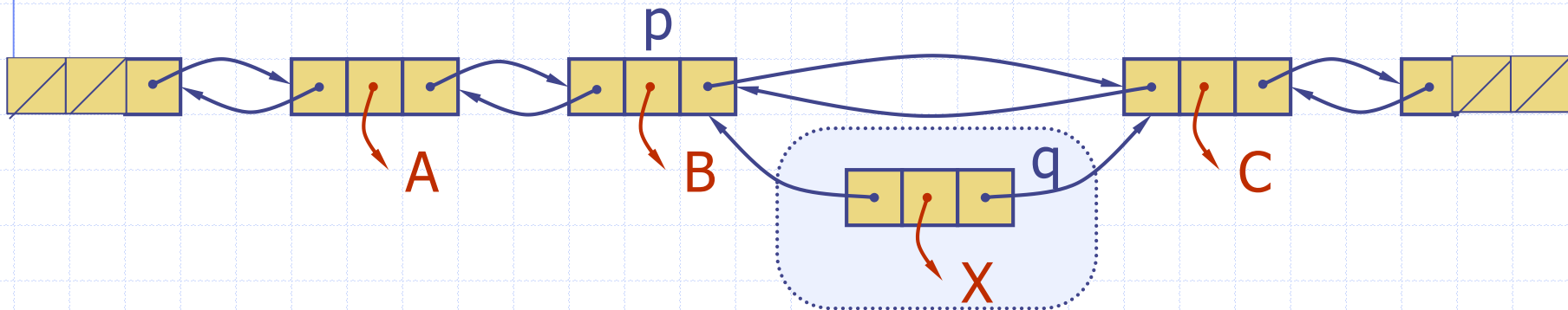
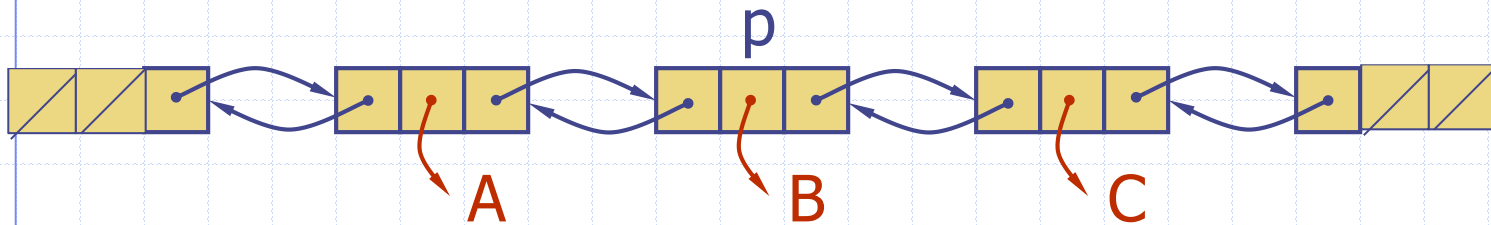
- Bidirectional Traversal: Allows easy traversal in both forward and backward directions.
- Insertion and Deletion: Operations involving adjacent nodes are more efficient because you don't need to traverse from the beginning

□ Cons of Doubly Linked Lists:

- Memory Usage: Doubly linked lists use more memory per node due to the additional previous node reference.
- Complexity: Slightly more complex to implement due to the handling of two pointers

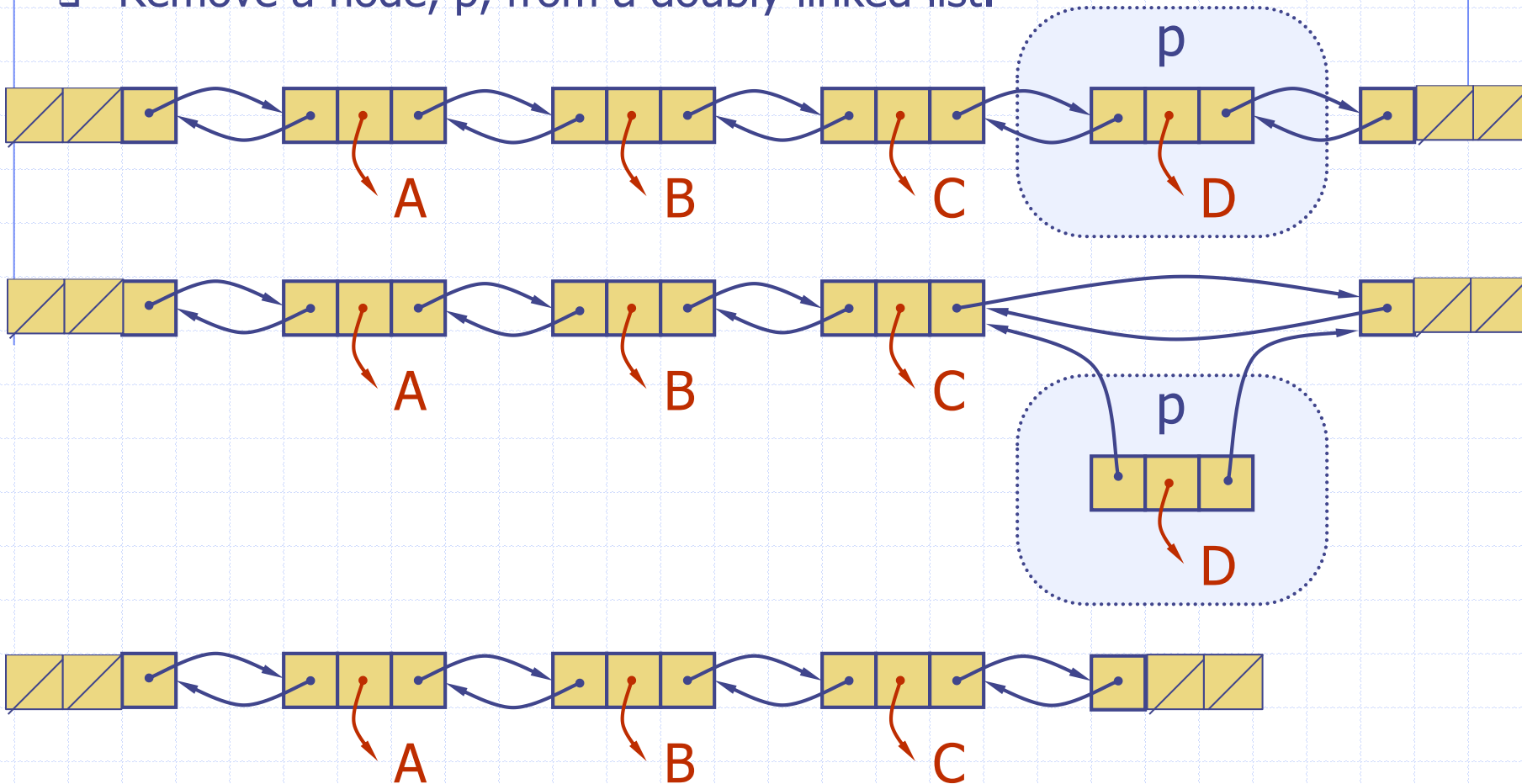
Insertion

- Insert a new node, q , between p and its successor.



Deletion

- Remove a node, p , from a doubly linked list.



Doubly-Linked List in Java

```
1  /** A basic doubly linked list implementation. */
2  public class DoublyLinkedList<E> {
3      //----- nested Node class -----
4      private static class Node<E> {
5          private E element;           // reference to the element stored at this node
6          private Node<E> prev;        // reference to the previous node in the list
7          private Node<E> next;        // reference to the subsequent node in the list
8          public Node(E e, Node<E> p, Node<E> n) {
9              element = e;
10             prev = p;
11             next = n;
12         }
13         public E getElement() { return element; }
14         public Node<E> getPrev() { return prev; }
15         public Node<E> getNext() { return next; }
16         public void setPrev(Node<E> p) { prev = p; }
17         public void setNext(Node<E> n) { next = n; }
18     } //----- end of nested Node class -----
19 }
```

Doubly-Linked List in Java, 2

```
21 private Node<E> header;           // header sentinel
22 private Node<E> trailer;          // trailer sentinel
23 private int size = 0;              // number of elements in the list
24 /** Constructs a new empty list. */
25 public DoublyLinkedList() {
26     header = new Node<>(null, null, null); // create header
27     trailer = new Node<>(null, header, null); // trailer is preceded by header
28     header.setNext(trailer);           // header is followed by trailer
29 }
30 /** Returns the number of elements in the linked list. */
31 public int size() { return size; }
32 /** Tests whether the linked list is empty. */
33 public boolean isEmpty() { return size == 0; }
34 /** Returns (but does not remove) the first element of the list. */
35 public E first() {
36     if (isEmpty()) return null;
37     return header.getNext().getElement(); // first element is beyond header
38 }
39 /** Returns (but does not remove) the last element of the list. */
40 public E last() {
41     if (isEmpty()) return null;
42     return trailer.getPrev().getElement(); // last element is before trailer
43 }
```

Doubly-Linked List in Java, 3

```
44 // public update methods
45 /** Adds element e to the front of the list. */
46 public void addFirst(E e) {
47     addBetween(e, header, header.getNext()); // place just after the header
48 }
49 /** Adds element e to the end of the list. */
50 public void addLast(E e) {
51     addBetween(e, trailer.getPrev(), trailer); // place just before the trailer
52 }
53 /** Removes and returns the first element of the list. */
54 public E removeFirst() {
55     if (isEmpty()) return null; // nothing to remove
56     return remove(header.getNext()); // first element is beyond header
57 }
58 /** Removes and returns the last element of the list. */
59 public E removeLast() {
60     if (isEmpty()) return null; // nothing to remove
61     return remove(trailer.getPrev()); // last element is before trailer
62 }
```

Doubly-Linked List in Java, 4

```
64 // private update methods
65 /** Adds element e to the linked list in between the given nodes. */
66 private void addBetween(E e, Node<E> predecessor, Node<E> successor) {
67     // create and link a new node
68     Node<E> newest = new Node<>(e, predecessor, successor);
69     predecessor.setNext(newest);
70     successor.setPrev(newest);
71     size++;
72 }
73 /** Removes the given node from the list and returns its element. */
74 private E remove(Node<E> node) {
75     Node<E> predecessor = node.getPrev();
76     Node<E> successor = node.getNext();
77     predecessor.setNext(successor);
78     successor.setPrev(predecessor);
79     size--;
80     return node.getElement();
81 }
82 } //----- end of DoublyLinkedList class -----
```

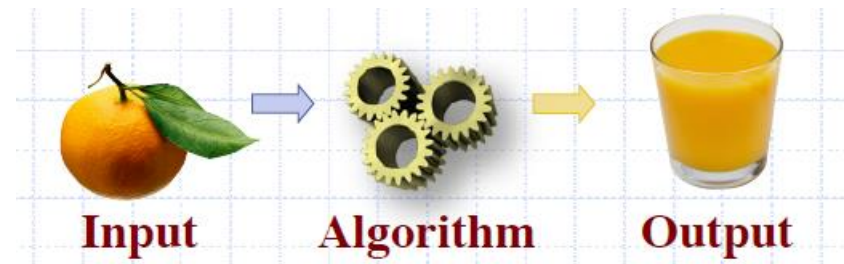
Analysis of Algorithms using Big O Notation

Data structure

Fall 2023

Hashemite University

Dr. Mohammad Al-hammouri



What is Big O Notation?

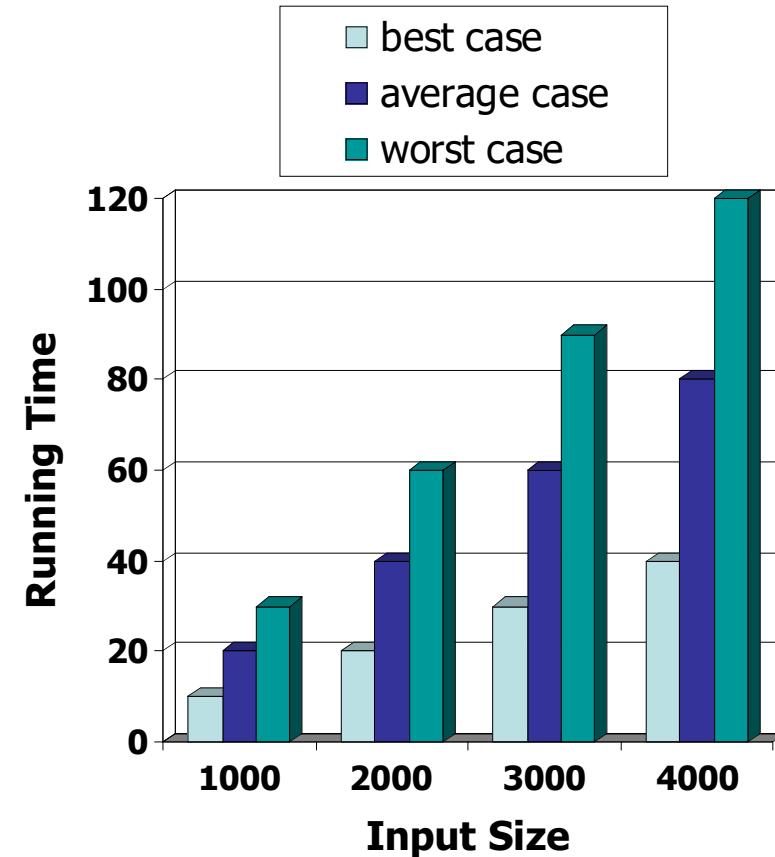
- **Big O notation** is a mathematical notation used in computer science
- Describe the performance or complexity of an algorithm in terms of time
- Represents the worst-case scenario (upper bound) for how an algorithm's time requirements increase as the size of the input increase
- Different time complexities: **$O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$** , etc.

Explanation of the "O" and "n" in Big O

- The "**O**" represents the order of the function describing the upper bound (worst-case scenario) of an algorithm's time complexity
- **n**: This variable represents the size of the input to the algorithm. It could be the number of elements in an array, the length of a string, the number of nodes in a linked list.
- **O(n)**, it means the algorithm's performance grows linearly with the size of the input. As the input size (n) increases, the time taken by the algorithm also increases linearly.

Running Time

- The running time of an algorithm typically grows with the input size.
- The running time of an algorithm typically grows with the input size.
- Average case time is often difficult to determine.
- We focus on the worst case running time.



$O(1)$ - Constant Time Complexity

- **Definition:** Algorithms with constant time complexity execute in the same amount of time regardless of the input size.

Regardless of the size of the array, the time complexity is constant

```
public int getElementAtIndex(int[] arr, int index) {  
    return arr[index];  
}
```

$O(n)$ - Linear Time Complexity

- **Definition:** Algorithms with linear time complexity have a runtime proportional to the input size.
- **Example:** Finding an element in an unsorted array

In the worst case, the function needs to iterate through all n elements in the array to find the target

```
public boolean linearSearch(int[] arr, int target) {  
    for (int num : arr) {  
        if (num == target) {  
            return true;  
        }  
    }  
    return false;  
}
```

$O(n^2)$ - Quadratic Time Complexity

- **Definition:** Algorithms with quadratic time complexity have nested loops, leading to performance issues for larger inputs.
- In the bubble sort algorithm, there are two nested loops that compare adjacent elements and swap them if they are in the wrong order.

```
int n = arr.length;
for (int i = 0; i < n - 1; i++) {
    for (int j = 0; j < n - i - 1; j++) {
        if (arr[j] > arr[j + 1]) {
            // Swap arr[j] and arr[j + 1]
            int temp = arr[j];
            arr[j] = arr[j + 1];
            arr[j + 1] = temp;
        }
    }
}
```

Logarithmic Time Complexity

- $O(\log n)$: Algorithms with logarithmic time complexity reduce the problem size in each step.
 - **Example:** Binary search in a sorted array
- $O(n \log n)$ - Linearithmic Time Complexity (Merge Sort).

Stack, Queue and D-Queue

Data structure

Fall 2023

Hashemite University

Dr. Mohammad Al-hammouri

Stacks, Queues, Deques

- Abstract Data Types
- Stacks
- Queues
- Deques

Abstract Data Types (ADT)

- An **Abstract Data Type** is an abstraction of a data structure.
- The **ADT** specifies:
 - what can be stored in the ADT
 - what operations can be done on/by the ADT.
 - It specifies **what** each operation does, **but NOT how** it does it
- In Java an **ADT** can be expressed by an **interface**.

Abstract Data Types (ADTs)

- Specify precisely the operations that can be performed
- The implementation is HIDDEN and can easily change

EXAMPLE

Phone Book ADT:

Objects of type: Phone Book Entry (name, phone, e-mail)

Operations: find, add, remove

Stacks, Queues, and Deques

ADT Stack

- Implementation with Arrays

- Implementation with Singly Linked List

ADT Queue

- Implementation with Arrays

- Implementation with Singly Linked List

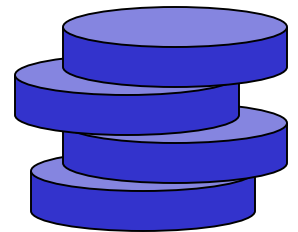
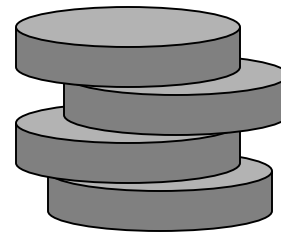
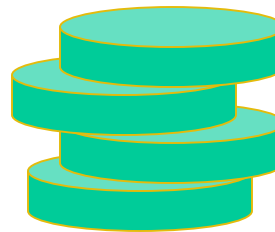
ADT Double Ended Queues

- Implementation with doubly Linked List

Stacks

PUSH

POP



The Stack Abstract Data Type

- Main methods:
 - `push(element)`: Inserts element onto top of stack
 - `object pop()`: Removes and returns the last element inserted (the one on the top); **if the stack is empty, returns null**
- Support methods:
 - `integer size()`: Returns the number of elements stored in stack
 - `boolean isEmpty()`: Returns a boolean indicating if stack is empty.
 - `object top()`: Returns the top element of the stack, without removing it; **if the stack is empty, returns null**

Stack Operation Example

Table shows a series of stack operations and their effects on an initially empty stack *S* of integers.

Method	Return Value	Stack Contents
push(5)	–	(5)
push(3)	–	(5, 3)
size()	2	(5, 3)
pop()	3	(5)
isEmpty()	false	(5)
pop()	5	()
isEmpty()	true	()
pop()	null	()
push(7)	–	(7)
push(9)	–	(7, 9)
top()	9	(7, 9)
push(4)	–	(7, 9, 4)
size()	3	(7, 9, 4)
pop()	4	(7, 9)
push(6)	–	(7, 9, 6)
push(8)	–	(7, 9, 6, 8)
pop()	8	(7, 9, 6)

Java Interface (1)

- Another way to achieve **abstraction** in Java, is with interfaces.
- An interface is a completely "abstract class" that is used to group related methods with empty bodies
- interfaces **cannot** be used to create objects

```
// Interface
```

```
interface Animal {  
    public void animalSound(); // interface method (does not have a body)  
    public void sleep(); // interface method (does not have a body)  
}
```

Java Interface (2)

- Dog "implements" the Animal interface

```
class Dog implements Animal {  
    public void animalSound() {  
        // The body of animalSound() is provided here  
        System.out.println("The Dog says: hho hho");  
    }  
    public void sleep() {  
        // The body of sleep() is provided here  
        System.out.println("Zzz");  
    }  
}  
  
public static void main(String[] args) {  
    Dog myDog = new Dog(); // Create a Dog object  
    myDog.animalSound();  
    myDog.sleep();  
}
```

ADT for Stack:

Our own Stack Interface in Java

```
public interface Stack<E> {  
    int size();  
    boolean isEmpty();  
    E top(); //Returns, but does not remove, the element at the  
top of the stack  
    void push(E element); //Inserts an element at the top of the  
stack  
    E pop(); //Removes and returns the top element from the  
stack.  
}
```

Simple Array-Based Stack Implementation(1)

```
public class ArrayStack<E> implements Stack<E> {
    public static final int CAPACITY=1000; // default array capacity
    private E[] data; // generic array used for storage
    private int t = -1; // index of the top element in stack
    public ArrayStack() { this(CAPACITY); } // constructs stack with default capacity
    public ArrayStack(int capacity) { // constructs stack with given capacity
        data = (E[]) new Object[capacity]; // safe cast; compiler may give warning
    }
    public int size() { return (t + 1); }
    public boolean isEmpty() { return (t == -1); }
    public void push(E e) throws IllegalStateException {
        if (size() == data.length) throw new IllegalStateException("Stack is full");
        data[++t] = e; // increment t before storing new item
    }
}
```


Simple Array-Based Stack Implementation (2)

```
public E top() {  
    if (isEmpty()) return null;  
    return data[t];  
}
```

```
public E pop() {  
    if (isEmpty()) return null;  
    E answer = data[t];  
    data[t] = null;  
    t--;  
    return answer;  
}
```

// dereference to help garbage collection

Generic....

- In Java, you cannot directly create an array of a generic type like `array = new E[size];`
- Create an array of Objects, then cast it to type E[]
`array = (E[]) new Object[size];`
- In Java, “**Object**” is a fundamental class that serves as the root of the class hierarchy. It is the superclass of all other classes in Java

```
Object obj1 = new String("Hello");  
Object obj2 = new Integer(10);  
Object obj3 = new MyClass();
```

Sample usage of our ArrayStack class.

Stack<Integer> S = new ArrayStack<>();	// contents: ()	
S.push(5);	// contents: (5)	
S.push(3);	// contents: (5, 3)	
System.out.println(S.size());	// contents: (5, 3)	outputs 2
System.out.println(S.pop());	// contents: (5)	outputs 3
System.out.println(S.isEmpty());	// contents: (5)	outputs false
System.out.println(S.pop());	// contents: ()	outputs 5
System.out.println(S.isEmpty());	// contents: ()	outputs true
System.out.println(S.pop());	// contents: ()	outputs null
S.push(7);	// contents: (7)	
S.push(9);	// contents: (7, 9)	
System.out.println(S.top());	// contents: (7, 9)	outputs 9
S.push(4);	// contents: (7, 9, 4)	
System.out.println(S.size());	// contents: (7, 9, 4)	outputs 3
System.out.println(S.pop());	// contents: (7, 9)	outputs 4
S.push(6);	// contents: (7, 9, 6)	
S.push(8);	// contents: (7, 9, 6, 8)	
System.out.println(S.pop());	// contents: (7, 9, 6)	outputs 8

Drawback of This Array-Based Stack Implementation

- The array implementation of a stack is simple and efficient.
- However it relies on a fixed-capacity array, which limits the ultimate size of the stack
- In cases where a user has a good estimate on the number of items needing to go in the stack, the array-based implementation is hard to beat.
- If capacity estimate is wrong, there can be grave consequences

Implementing a Stack with a Singly Linked List

- Linked-list approach has efficient memory usage
 - It is always proportional to the number of actual elements currently in the stack
- Decide if the top of the stack is at the front or back of the list.
- We can insert and delete elements in constant time only at the front

<i>Stack Method</i>	<i>Singly Linked List Method</i>
size()	list.size()
isEmpty()	list.isEmpty()
push(<i>e</i>)	list.addFirst(<i>e</i>)
pop()	list.removeFirst()
top()	list.first()

Implementation of a Stack using a SinglyLinkedList as storage

```
1 public class LinkedStack<E> implements Stack<E> {
2     private SinglyLinkedList<E> list = new SinglyLinkedList<>(); // an empty list
3     public LinkedStack() { } // new stack relies on the initially empty list
4     public int size() { return list.size(); }
5     public boolean isEmpty() { return list.isEmpty(); }
6     public void push(E element) { list.addFirst(element); }
7     public E top() { return list.first(); }
8     public E pop() { return list.removeFirst(); }
9 }
```

Reversing an Array Using a Stack

```
1  /** A generic method for reversing an array. */
2  public static <E> void reverse(E[ ] a) {
3      Stack<E> buffer = new ArrayStack<>(a.length);
4      for (int i=0; i < a.length; i++)
5          buffer.push(a[i]);
6      for (int i=0; i < a.length; i++)
7          a[i] = buffer.pop();
8  }
```

Reversing an Array Using a Stack

```
/** Tester routine for reversing arrays */  
public static void main(String args[ ]) {  
    Integer[ ] a = {4, 8, 15, 16, 23, 42};    // autoboxing allows this  
    String[ ] s = {"Jack", "Kate", "Hurley", "Jin", "Michael"};  
    System.out.println("a = " + Arrays.toString(a));  
    System.out.println("s = " + Arrays.toString(s));  
    System.out.println("Reversing...");  
    reverse(a);  
    reverse(s);  
    System.out.println("a = " + Arrays.toString(a));  
    System.out.println("s = " + Arrays.toString(s));  
}
```

The output from this method is the following:

```
a = [4, 8, 15, 16, 23, 42]  
s = [Jack, Kate, Hurley, Jin, Michael]  
Reversing...  
a = [42, 23, 16, 15, 8, 4]  
s = [Michael, Jin, Hurley, Kate, Jack]
```

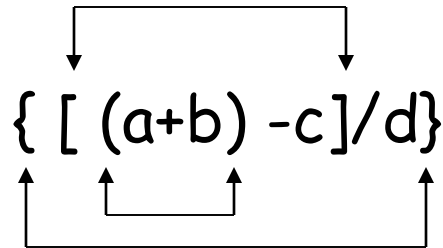

Parenthesis Matching

Each “(”, “{”, or “[” must be paired with a matching “)”, “}”, or “]”

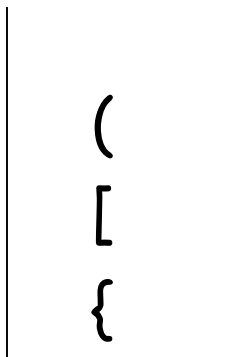
- correct: ()(()){([())}
- correct: (())(()){([())}
- incorrect:)(()){([())}
- incorrect: ({ []})
- incorrect: (

Parenthesis Matching

Checking for balanced parenthesis



$\{ [a+b) -c]/d \}$



$\{ [(a+b) -c]/d \}$

When we read an open parenthesis $input='('$
pop $ch='('$ from stack and check they have matching types

Matching delimiters in an arithmetic expression

```
public static boolean isMatched(String expression) {  
    final String opening    = "({[";           // opening delimiters  
    final String closing    = ")}]";           // respective closing delimiters  
    Stack<Character> buffer = new LinkedStack<>();  
    for (char c : expression.toCharArray()) {  
        if (opening.indexOf(c) != -1)           // this is a left delimiter  
            buffer.push(c);  
        else if (closing.indexOf(c) != -1) {     // this is a right delimiter  
            if (buffer.isEmpty())                // nothing to match with  
                return false;  
            if (closing.indexOf(c) != opening.indexOf(buffer.pop()))  
                return false;                   // mismatched delimiter  
        }  
    }  
    return buffer.isEmpty();                    // were all opening delimiters matched?  
}
```

HTML Tag Matching

- ❑ For fully-correct HTML, each `<name>` should pair with a matching `</name>`

```
<body>
<center>
<h1> The Little Boat </h1>
</center>
<p> The storm tossed the little
boat like a cheap sneaker in an
old washing machine. The three
drunken fishermen were used to
such treatment, of course, but
not the tree salesman, who even as
a stowaway now felt that he
had overpaid for the voyage. </p>
<ol>
<li> Will the salesman die? </li>
<li> What color is the boat? </li>
<li> And what about Naomi? </li>
</ol>
</body>
```

The Little Boat

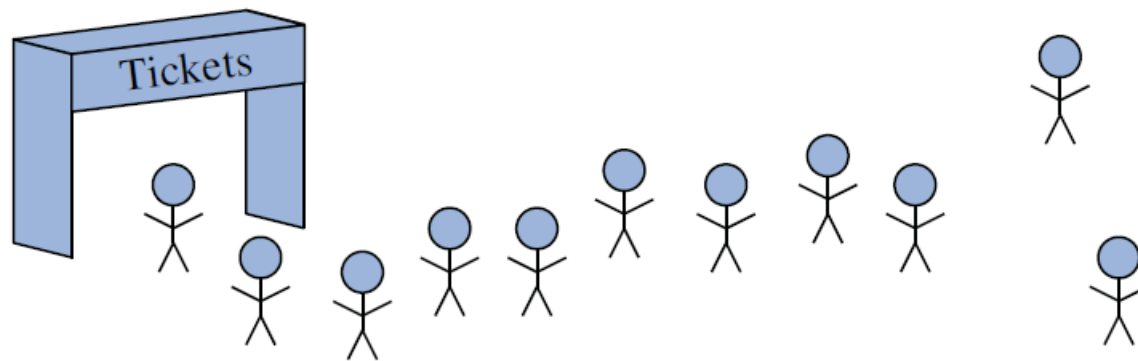
The storm tossed the little boat like a cheap sneaker in an old washing machine. The three drunken fishermen were used to such treatment, of course, but not the tree salesman, who even as a stowaway now felt that he had overpaid for the voyage.

1. Will the salesman die?
2. What color is the boat?
3. And what about Naomi?

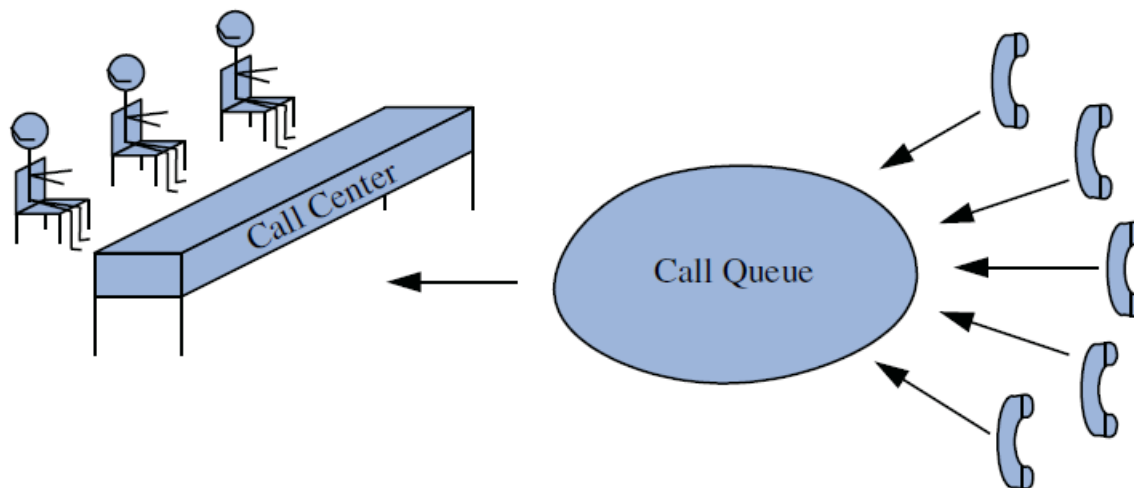
Stacks, Queues, Deques

- Abstract Data Types
- Stacks
- **Queues**
- Deques

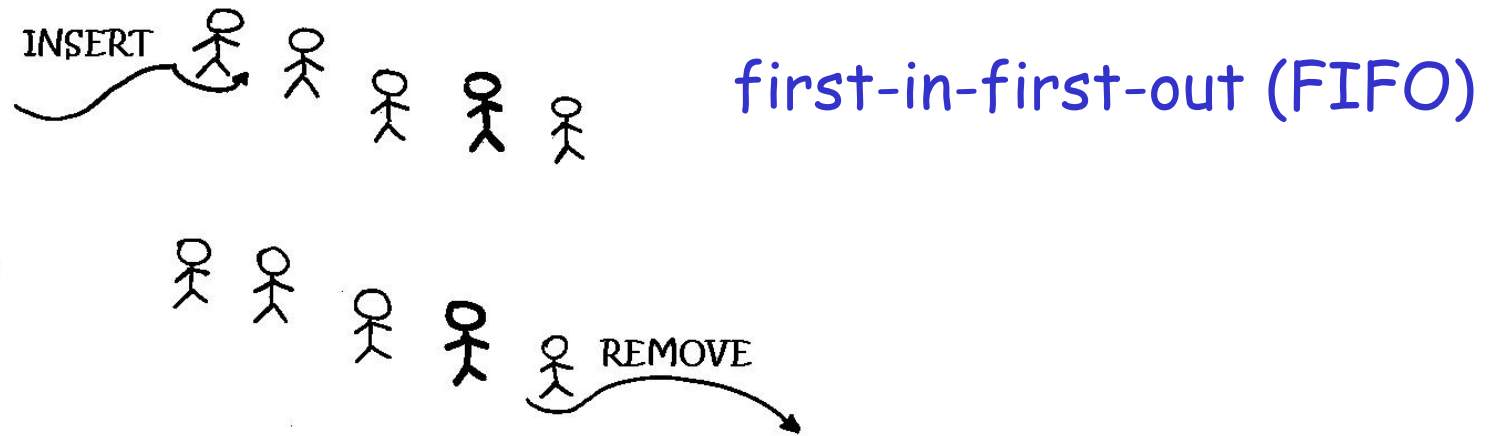
The Queue



(a)



The Queue



Elements are inserted at the **rear** (**enqueued**) and removed from the **front** (**dequeued**)

Applications of Queues

- Direct applications
 - Waiting lists
 - Access to shared resources (e.g., printer)
 - Multiprogramming
- Indirect applications
 - Auxiliary data structure for algorithms
 - Component of other data structures

The Queue Abstract Data Type

- Fundamental methods:

enqueue(o): Insert object o at the rear of the queue

dequeue(): Remove the object from the front of the queue and return it; **if the queue is empty return null.**

- Support methods:

size(): Return the number of objects in the queue

isEmpty(): Return a boolean value that indicates whether the queue is empty

first(): Return, but do not remove, the front object in the queue; **if the queue is empty return null.**

Example

Method	Return Value	first $\leftarrow Q \leftarrow$ last
enqueue(5)	–	(5)
enqueue(3)	–	(5, 3)
size()	2	(5, 3)
dequeue()	5	(3)
isEmpty()	false	(3)
dequeue()	3	()
isEmpty()	true	()
dequeue()	null	()
enqueue(7)	–	(7)
enqueue(9)	–	(7, 9)
first()	7	(7, 9)
enqueue(4)	–	(7, 9, 4)

Array-based Queue



Figure 6.5: Using an array to store elements of a queue, such that the first element inserted, “A”, is at cell 0, the second element inserted, “B”, at cell 1, and so on.

Dequeue operation: The element to be removed is stored at index 0 of the array. $O(n)$

Array-based Queue

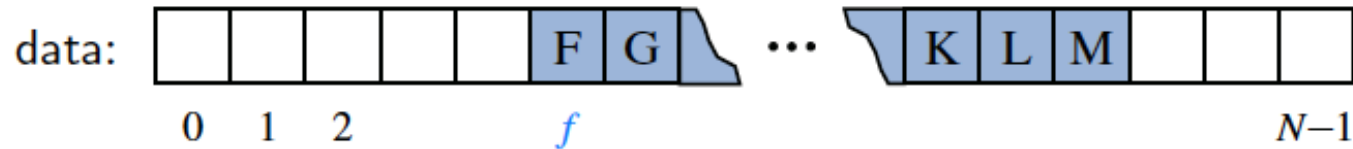


Figure 6.6: Allowing the front of the queue to drift away from index 0. In this representation, index f denotes the location of the front of the queue.

Dequeue operation: Such an algorithm for dequeue would run in $O(1)$ time

The back of the queue would reach the end of the underlying array even when there are fewer than N elements currently in the queue

Array-based Queue (circular Array)

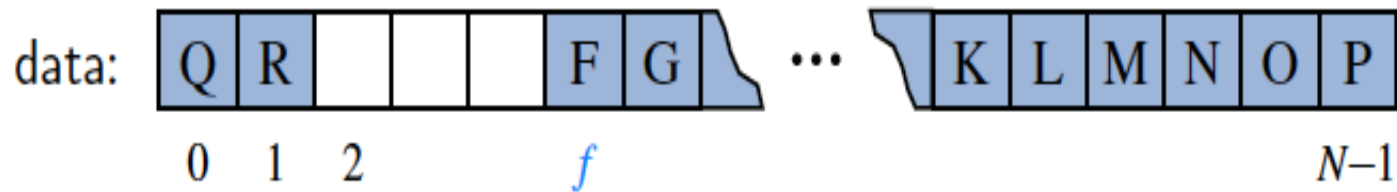
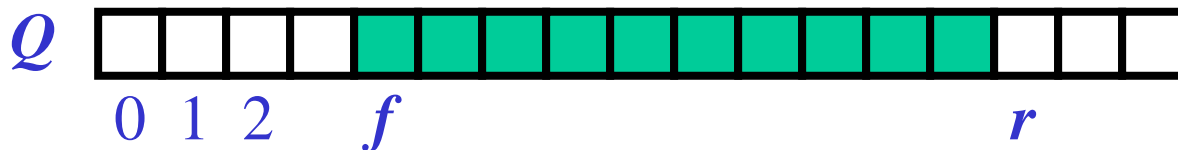


Figure 6.7: Modeling a queue with a circular array that wraps around the end.

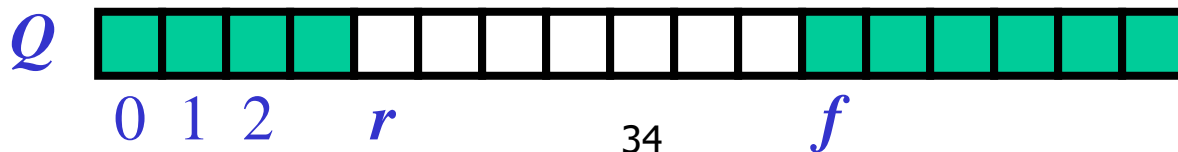
Array-based Queue

- Use an array of size N in a circular fashion
- Two variables keep track of the front and size
 f index of the front element
 sz number of stored elements
- When the queue has fewer than N elements, array location $r = (f + sz) \bmod N$ is the first empty slot past the rear of the queue

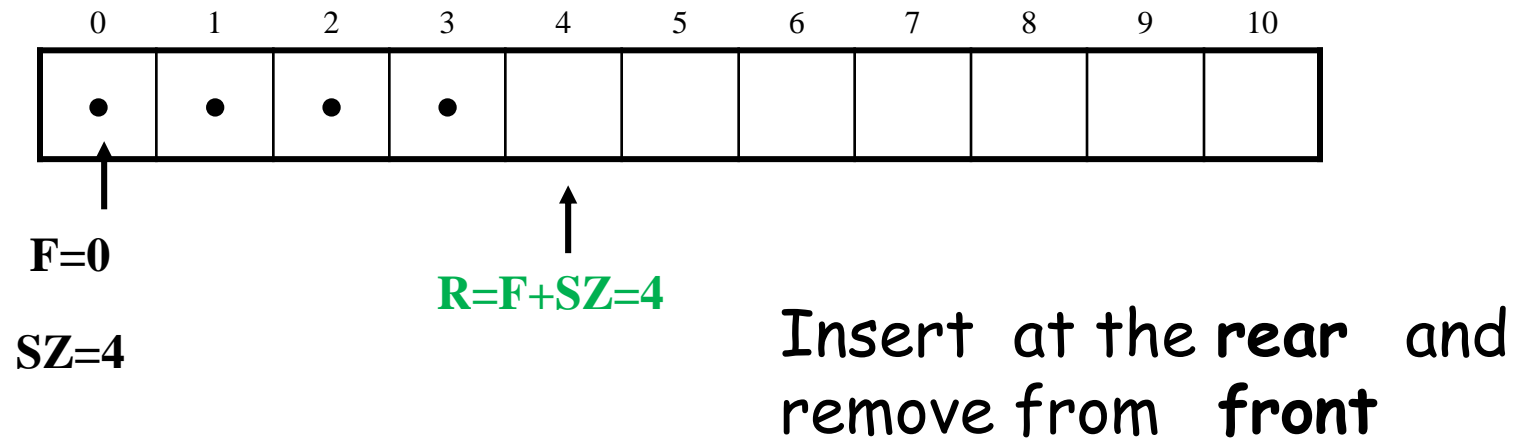
normal configuration



wrapped-around configuration

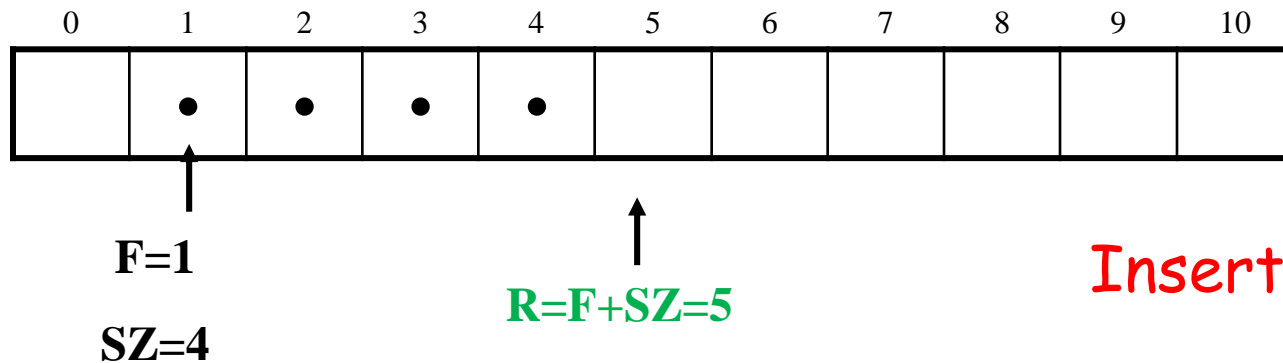
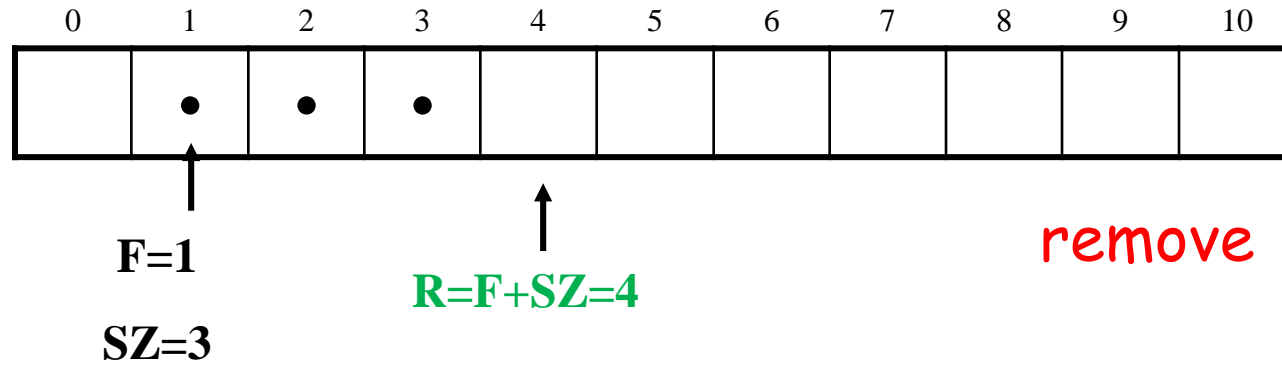
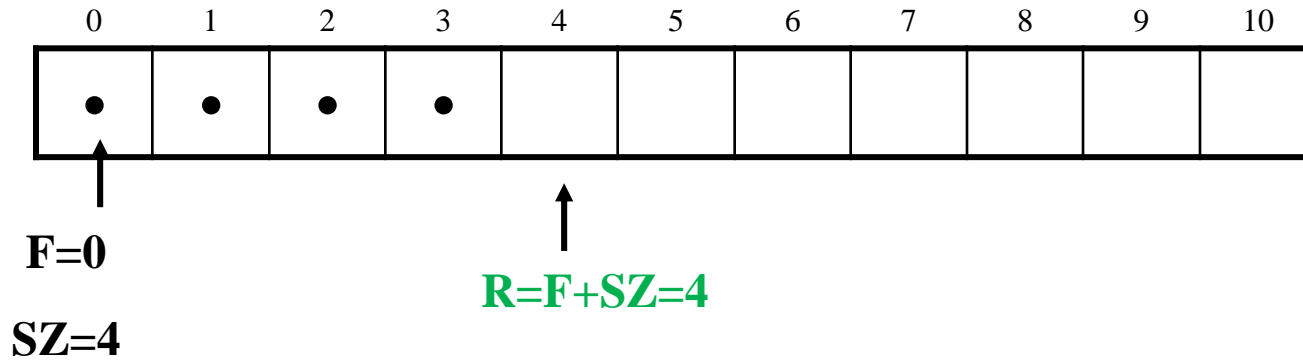


Implementing a Queue with an Array

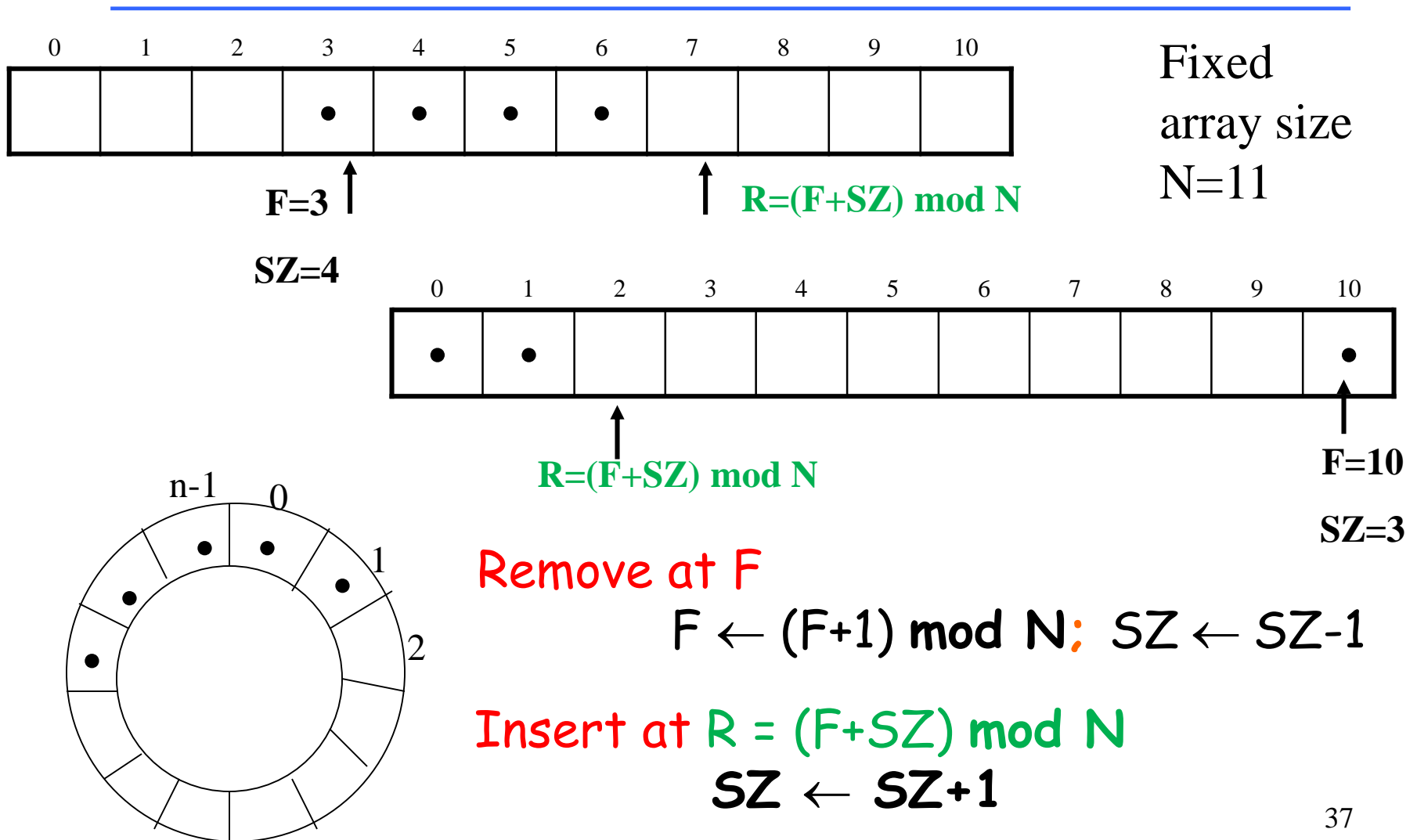


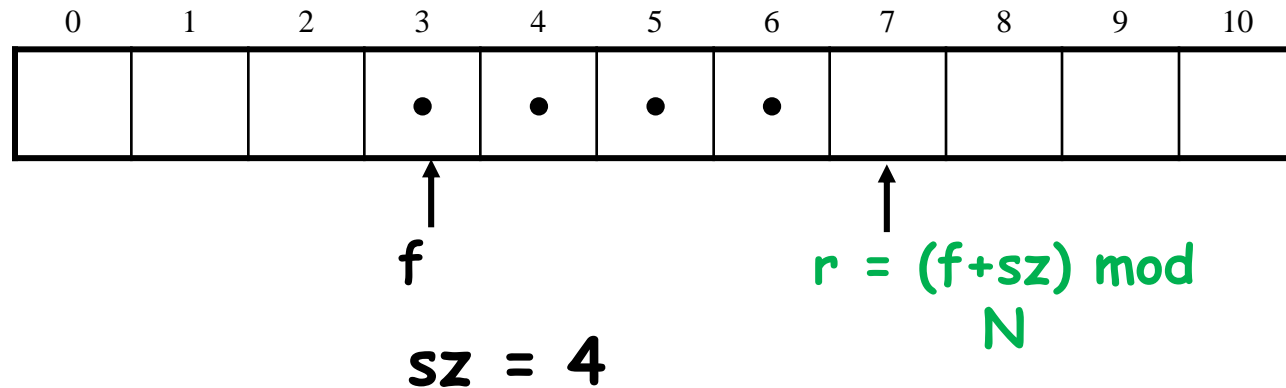
Remove: $F \leftarrow F+1$; $SZ \leftarrow SZ-1$

Insert: Insert at $F+SZ$ ($=R$);
 $SZ \leftarrow SZ+1$ (note that $R=R+1$)



Implementing a Queue with an Array





Questions:

What is the status of an empty queue?

Answer: $f=r$ and $sz=0$; f may be any index in $0..N-1$

Initialize data structure with:

$Q \leftarrow$ new array of size N

$f \leftarrow 0$; $sz \leftarrow 0$;

Algorithm **size()**:
 return sz

Algorithm **isEmpty()**:
 return $(sz == 0)$

Algorithm **first()**:
 if **isEmpty()** then
 return null
 return $Q[f]$

Algorithm **dequeue()**:
 if **isEmpty()** then
 return null
 temp $\leftarrow Q[f]$
 $Q[f] \leftarrow$ null
 $f \leftarrow (f + 1) \bmod N$
 $sz \leftarrow sz - 1$
 return temp

Algorithm **enqueue(o)**:
 if $sz = N - 1$ then
 throw *IllegalStateException*
 else
 $r \leftarrow (f + sz) \bmod N$
 $Q[r] \leftarrow o$
 $sz \leftarrow sz + 1$

Performance

Time:

size()	$O(1)$
isEmpty()	$O(1)$
first()	$O(1)$
enqueue(o)	$O(1)$
dequeue()	$O(1)$

Our Queue Interface in Java

- Java interface corresponding to our Queue ADT
- Assumes that **first()** and **dequeue()** return null if queue is empty

```
public interface Queue<E> {  
    int size();  
    boolean isEmpty();  
    E first();  
    void enqueue(E e);  
    E dequeue();  
}
```

Array-based Implementation

concrete data structure for Queue based on arrays

```
1  /** Implementation of the queue ADT using a fixed-length array. */
2  public class ArrayQueue<E> implements Queue<E> {
3      // instance variables
4      private E[] data;           // generic array used for storage
5      private int f = 0;          // index of the front element
6      private int sz = 0;         // current number of elements
7
8      // constructors
9      public ArrayQueue() {this(CAPACITY);} // constructs queue with default capacity
10     public ArrayQueue(int capacity) {      // constructs queue with given capacity
11         data = (E[]) new Object[capacity]; // safe cast; compiler may give warning
12     }
13
14     // methods
15     /** Returns the number of elements in the queue. */
16     public int size() { return sz; }
17
18     /** Tests whether the queue is empty. */
19     public boolean isEmpty() { return (sz == 0); }
20
```

Queue ADT

Array-based Implementation (2)

```
21  /** Inserts an element at the rear of the queue. */
22  public void enqueue(E e) throws IllegalStateException {
23      if (sz == data.length) throw new IllegalStateException("Queue is full");
24      int avail = (f + sz) % data.length;    // use modular arithmetic
25      data[avail] = e;
26      sz++;
27  }
28
29  /** Returns, but does not remove, the first element of the queue (null if empty). */
30  public E first() {
31      if (isEmpty()) return null;
32      return data[f];
33  }
34
35  /** Removes and returns the first element of the queue (null if empty). */
36  public E dequeue() {
37      if (isEmpty()) return null;
38      E answer = data[f];
39      data[f] = null;                // dereference to help garbage collection
40      f = (f + 1) % data.length;
41      sz--;
42      return answer;
43  }
```

Comparison to java.util.Queue

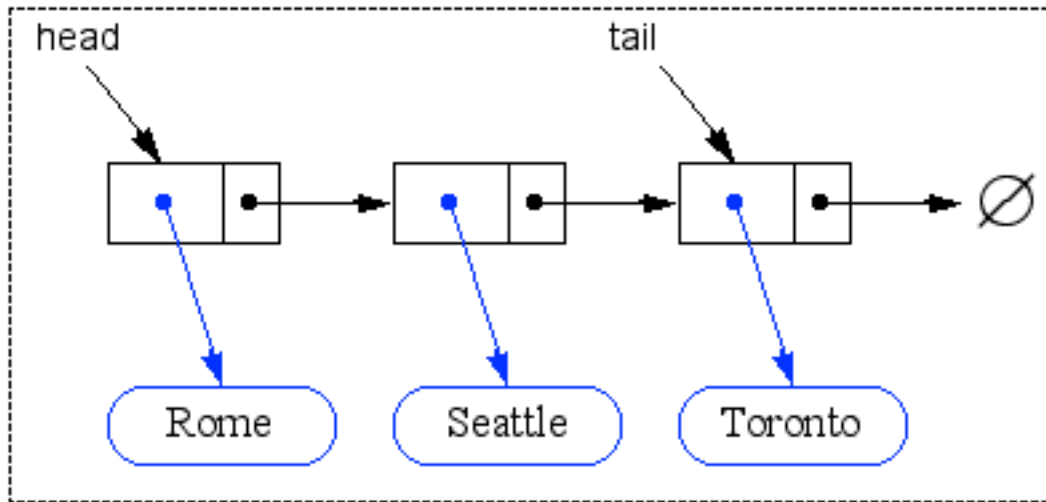
- Our Queue methods and corresponding methods of **java.util.Queue**:

Our Queue ADT	Interface java.util.Queue	
	throws exceptions	returns special value
enqueue(<i>e</i>)	add(<i>e</i>)	offer(<i>e</i>)
dequeue()	remove()	poll()
first()	element()	peek()
size()	size()	
isEmpty()	isEmpty()	

Implementing a Queue with a Singly Linked List

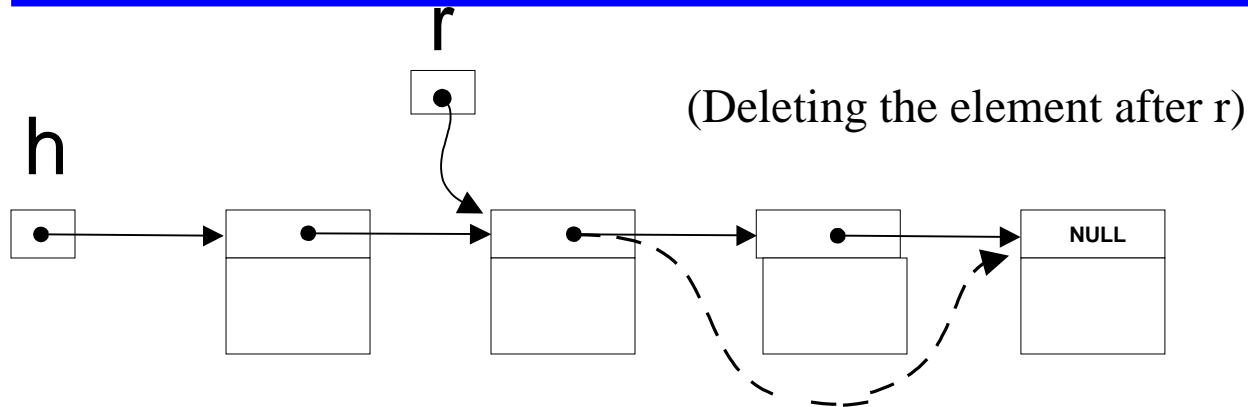
Nodes connected in singly linked list

We keep a pointer to the head and one to the tail



The head of the list is the front of the queue, the tail of the list is the rear of the queue. *Why not the opposite?*

Remember



First element (easy)

$h \leftarrow h.getNext()$

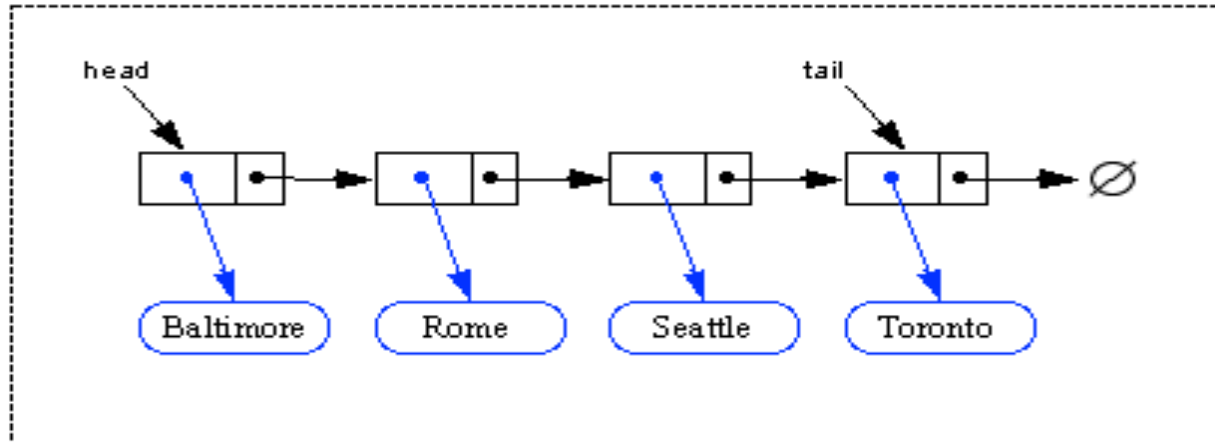
Element after r (easy)

$w \leftarrow r.getNext()$
 $r.setNext(w.getNext())$

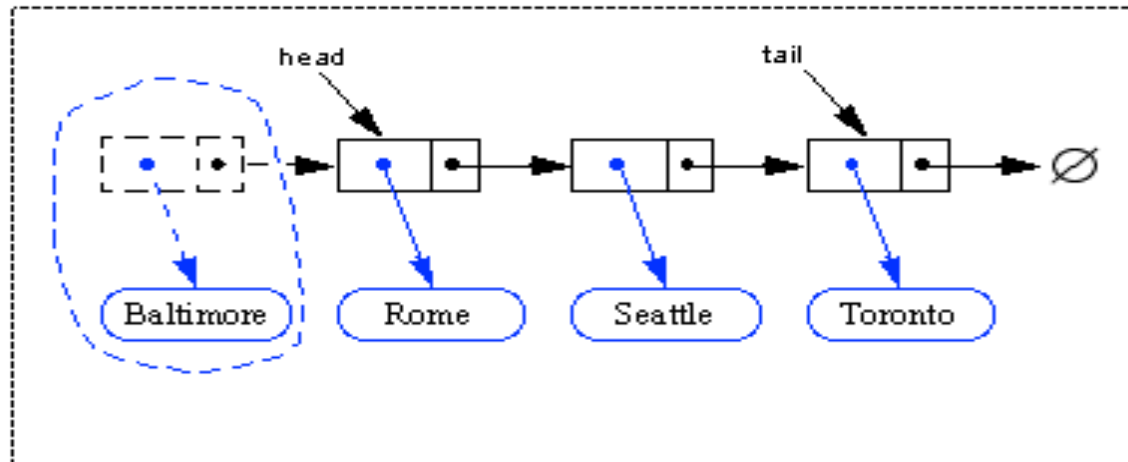
Element at r (difficult)

- Use a pointer to the preceding element, or
- Exchange the content of the element at r with the contents of the element following r , and delete the element after r . (this is impossible if r points to the last element)

Removing at the Head



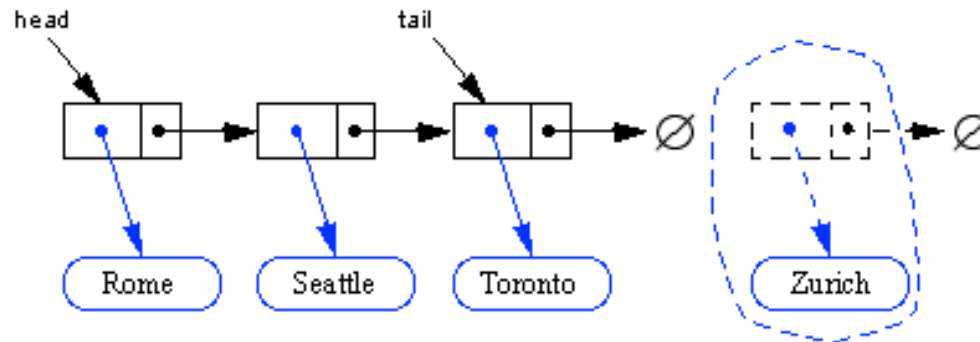
- advance head reference



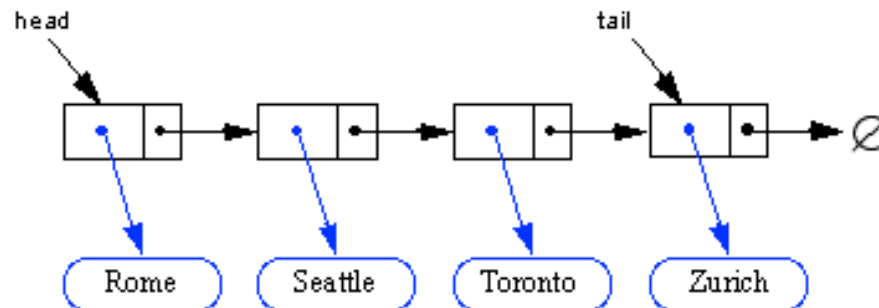
- inserting at the head is just as easy (but we won't do it !)

Inserting at the Tail

- create a new node



- chain it and move the tail reference



- how about removing at the tail?

This is very difficult; we won't do it as it would take $O(n)$.

Queue implementation using Singly Linked Lists

concrete data structure for Queue based on linked list

```
public class LinkedQueue<E> implements Queue<E> {  
    private SinglyLinkedList<E> list = new SinglyLinkedList<>(); // an empty list  
    public LinkedQueue() { } // new queue relies on the initially empty list  
    public void enqueue(E element) { list.addLast(element); }  
    public E dequeue() { return list.removeFirst(); }  
    public int size() { return list.size(); }  
    public E first() { return list.first(); }  
    public boolean isEmpty() { return list.isEmpty(); }  
}
```

Queue ADT

ADAPTING an existing class
SinglyLiskedList<E>
(See lecture on singly linked lists
where this class is defined)

Performance

Time:

size()	$O(1)$
isEmpty()	$O(1)$
front()	$O(1)$
enqueue(o)	$O(1)$
dequeue()	$O(1)$

Space: $O(n)$ when storing n elements

← To calculate, need to verify the big-Oh of each method you call. Looking at previous page, the corresponding operations used for singly linked list are all $O(1)$.
(Review lecture about singly linked lists)

Choosing Queue implementation

If we know in advance a reasonable upper bound for the number of elements in the queue, then use

⇒ ARRAYS (fixed size N , as seen here)

Otherwise

⇒ LISTS

⇒ Extendible ARRAYS (when full increases the size; as will see later)

Example: Palindromes

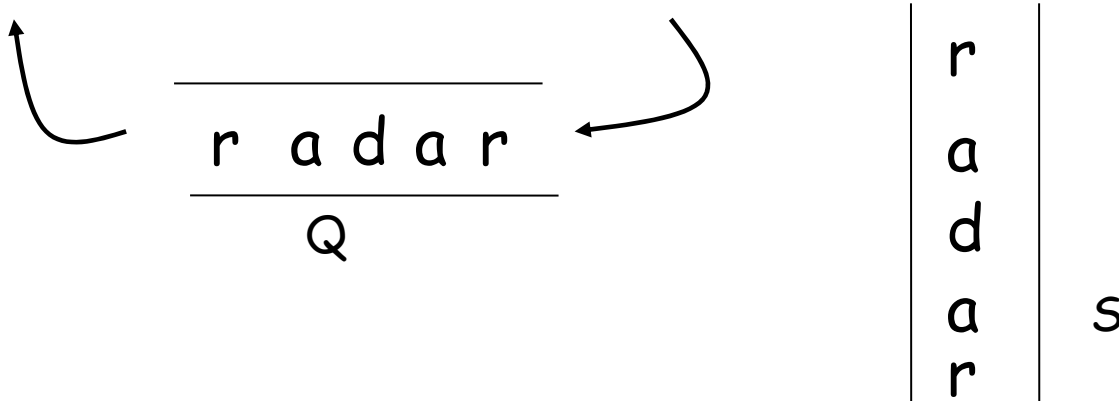
Problem: Decide if a string is a palindrome using one queue and one stack

“RADAR”

“WAS IT A CAR OR A CAT I SAW”

Read the line into a stack and into a queue

Compare the outputs of the queue and the stack



MADAM, I' M ADAM

LONELY TYLENOL

NEVER ODD OR EVEN

NO LEMON NO MELON

TOO BAD I HID A BOOT

O STONE BE NOT SO

RACE FAST SAFE CAR

DO GEESE SEE GOD

NO DEVIL LIVED ON

A TOYOTA' S A TOYOTA

A more general ADT: Double-Ended Queues (Deque)

A **double-ended** queue, or **deque**, supports insertion and deletion from the front and back.

Main methods:

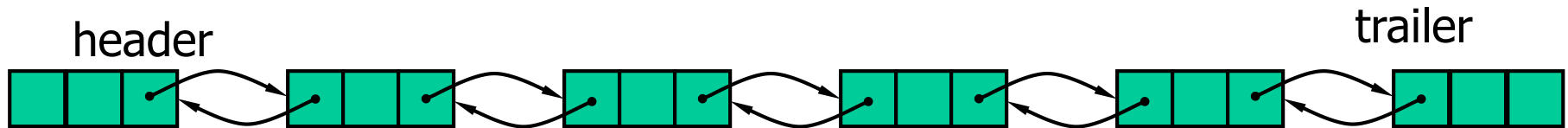
addFirst(e):	Insert e at the beginning of deque.
addLast(e):	Insert e at end of deque
removeFirst():	Removes and returns first element
removeLast():	Removes and returns last element

Support methods:

- first()**
- last()**
- size()**
- isEmpty()**

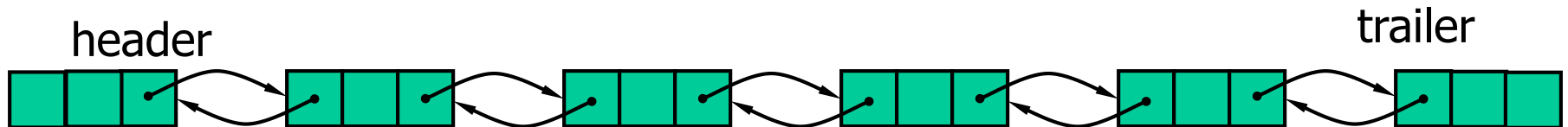
Implementing Deques with Doubly Linked Lists

Deletions at the tail of a singly linked list cannot be done efficiently



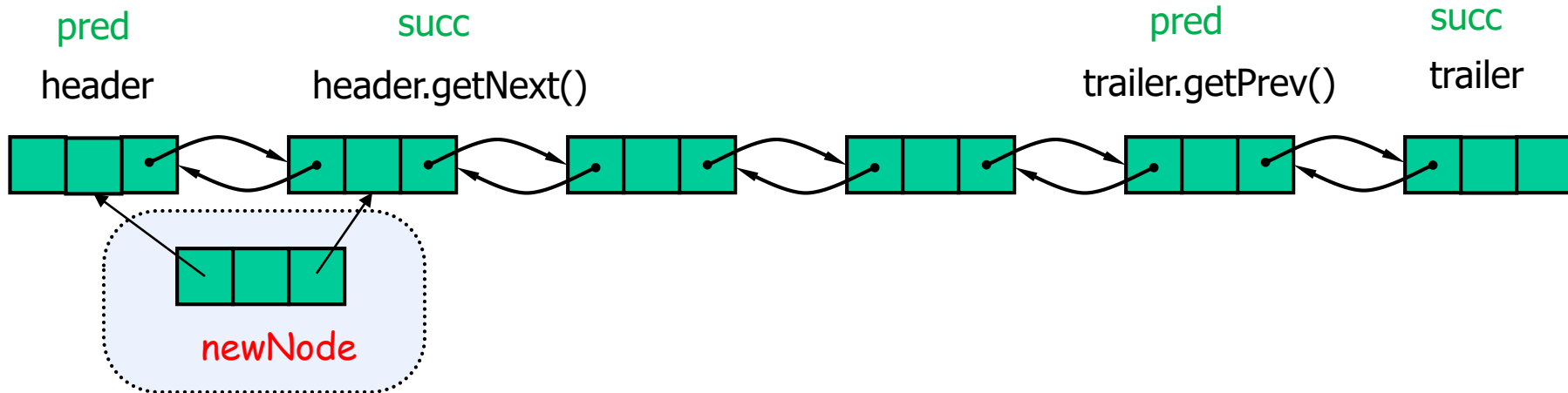
To implement a deque, we use a **doubly linked** list with special header and trailer nodes

-
- The **header** node goes before the first list element. It has a valid next link but a null prev link.
 - The **trailer** node goes after the last element. It has a valid prev reference but a null next reference.



NOTE: the header and trailer nodes are sentinel or “dummy” nodes because they do not store elements.

Insertion : `addFirst(o)` and `addLast(o)`

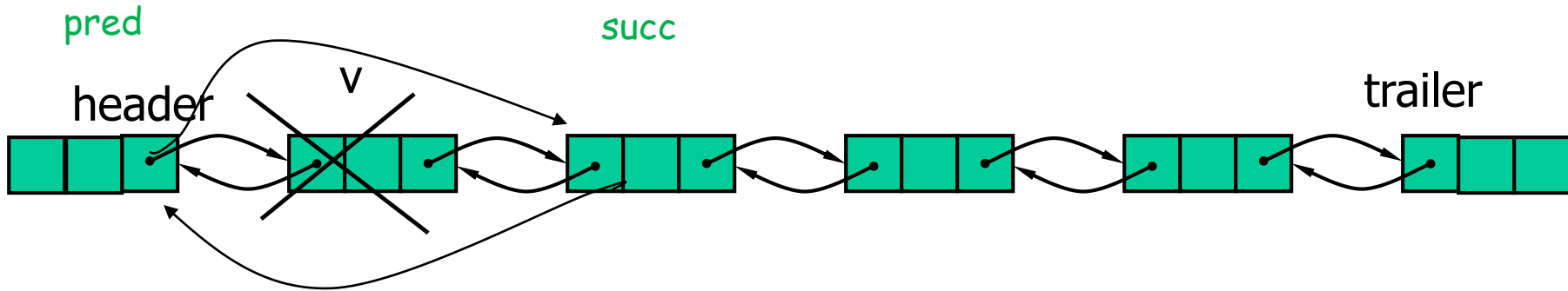


```
Algorithm addBetween(e, pred, succ):  
    newNode ← new Node;  
    newNode.setElement(e);  
    newNode.setPrev(pred);  
    newNode.setNext(succ);  
    pred.setNext(newNode);  
    succ.setPrev(newNode);  
    size ← size+1
```

```
Algorithm addFirst(e):  
    addBetween(e, header, header.getNext())
```

```
Algorithm addLast(e):  
    addBetween(e, trailer.getPrev(), trailer);
```

Deletion : `removeFirst()` and `removeLast()`

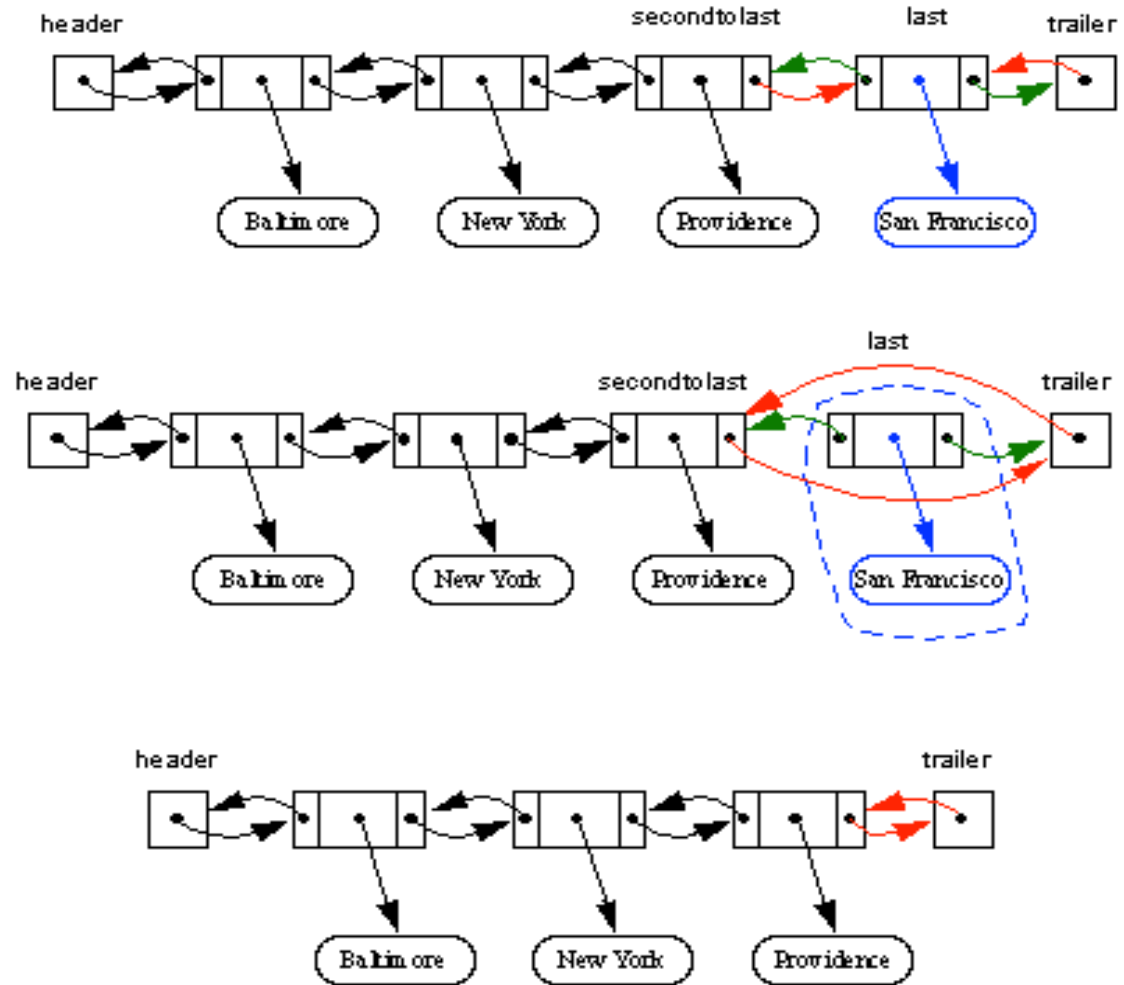


```
Algorithm remove(v):  
    pred ← v.getPrev();  
    succ ← v.getNext();  
    newNode ← new Node;  
    pred.setNext(succ);  
    succ.setPrev(pred);  
    size ← size-1  
    return v.getElement();
```

```
Algorithm removeFirst():  
    if (isEmpty()) return null;  
    return remove(header.getNext());
```

```
Algorithm removeLast():  
    if (isEmpty()) return null;  
    return remove(trailer.getPrev());
```

Here's a visualization of the code for `removeLast()`:



With this implementation, all methods have complexity $O(1)$

Performance:

Dequeues using Doubly Linked Lists

Operation	Time
<code>addFirst(e):</code>	$O(1)$
<code>addLast(e):</code>	$O(1)$
<code>removeFirst():</code>	$O(1)$
<code>removeLast():</code>	$O(1)$
<code>first()</code>	$O(1)$
<code>last()</code>	$O(1)$
<code>size()</code>	$O(1)$
<code>isEmpty()</code>	$O(1)$

Space: $O(n)$
for a deque with n items

Implementing Stacks and Queues with Deques

Stacks with Deques:

Stack Method	Deque Implementation
size()	size()
isEmpty()	isEmpty()
top()	last()
push(o)	addLast(o)
pop()	removeLast()

Queues with Deques:

Queue Method	Deque Implementation
size()	size()
isEmpty()	isEmpty()
front()	first()
enqueue(o)	addLast(o)
dequeue()	removeFirst()

Recursion

Data structure

Fall 2023

Hashemite University

Dr. Mohammad Al-hammouri



The Recursion Pattern

- **Recursion**: when a method calls itself
- Classic example – the factorial function:
$$n! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot (n-1) \cdot n$$
- Recursive definition:
$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot f(n-1) & \text{else} \end{cases}$$
- As a Java method:

```
1 public static int factorial(int n) throws IllegalArgumentException {  
2     if (n < 0)  
3         throw new IllegalArgumentException();    // argument must be nonnegative  
4     else if (n == 0)  
5         return 1;                               // base case  
6     else  
7         return n * factorial(n-1);               // recursive case  
8 }
```

Content of a Recursive Method

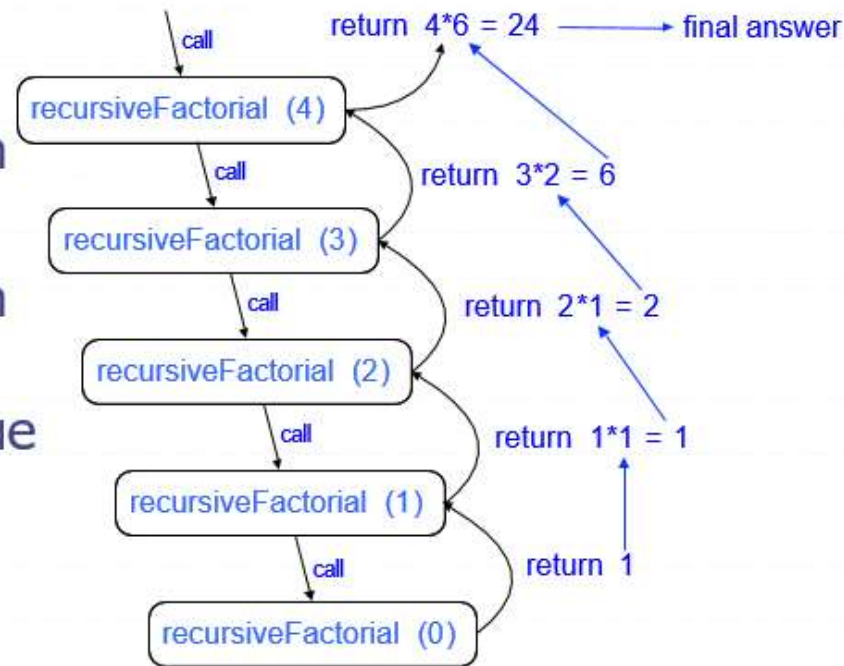
- **Base case(s)**
 - Values of the input variables for which we perform no recursive calls are called **base cases** (there should be at least one base case).
 - Every possible chain of recursive calls **must** eventually reach a base case.
- **Recursive calls**
 - Calls to the current method.
 - Each recursive call should be defined so that it makes progress towards a base case.

Visualizing Recursion

□ Recursion trace

- A box for each recursive call
- An arrow from each caller to callee
- An arrow from each callee to caller showing return value

□ Example

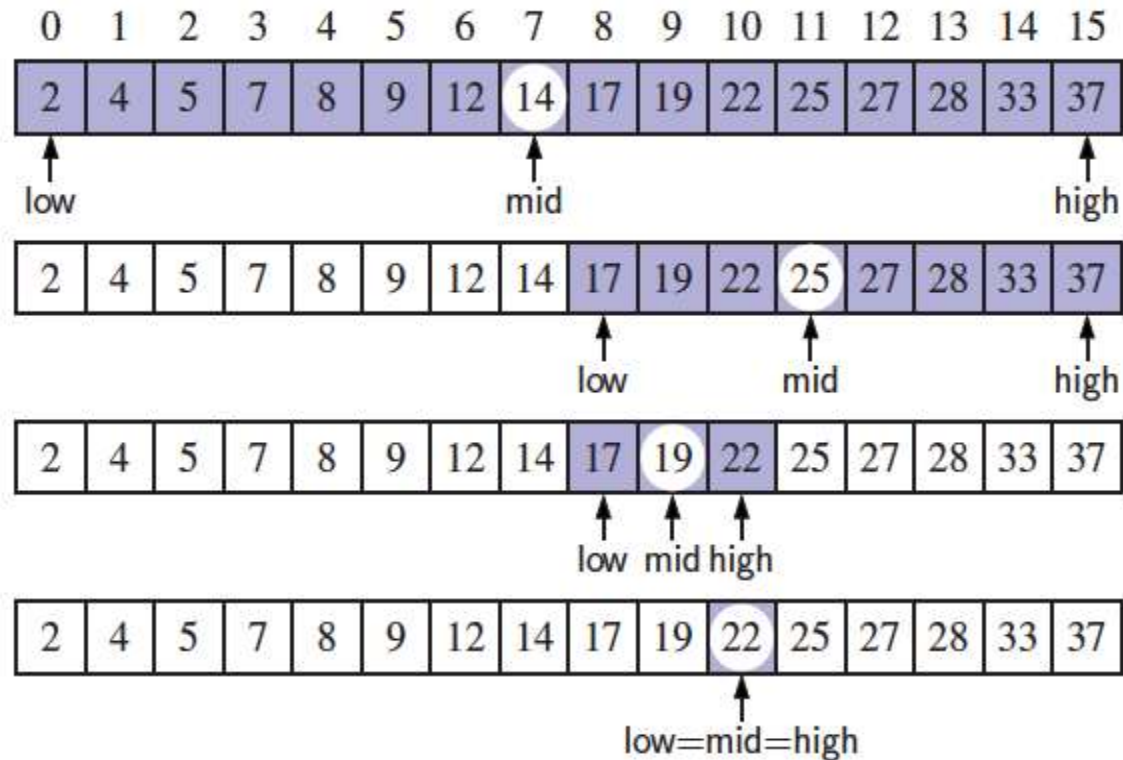


Example: Binary Search

- Search for an integer in an ordered list

```
5 public static boolean binarySearch(int[ ] data, int target, int low, int high) {  
6     if (low > high)  
7         return false; // interval empty; no match  
8     else {  
9         int mid = (low + high) / 2;  
10        if (target == data[mid])  
11            return true; // found a match  
12        else if (target < data[mid])  
13            return binarySearch(data, target, low, mid - 1); // recur left of the middle  
14        else  
15            return binarySearch(data, target, mid + 1, high); // recur right of the middle  
16    }  
17 }
```

Visualizing Binary Search



Reversing the n elements of an array

- In creating recursive methods, it is important to define the methods in ways that facilitate recursion.
- This sometimes requires we define additional parameters that are passed to the method.
- **reverseArray**(int[] data, int low, int high)

0	1	2	3	4	5	6	7
4	3	6	2	7	8	9	5
5	3	6	2	7	8	9	4
5	9	6	2	7	8	3	4
5	9	8	2	7	6	3	4
5	9	8	7	2	6	3	4

Reversing the n elements of an array

Eventually a base case is reached when the condition $\text{low} < \text{high}$ fails, either because **low == high** in the case that n is odd, or because **low == high + 1** in the case that n is even.

```
1  /** Reverses the contents of subarray data[low] through data[high] inclusive. */
2  public static void reverseArray(int[ ] data, int low, int high) {
3      if (low < high) {                // if at least two elements in subarray
4          int temp = data[low];        // swap data[low] and data[high]
5          data[low] = data[high];
6          data[high] = temp;
7          reverseArray(data, low + 1, high - 1);    // recur on the rest
8      }
9  }
```

Trees

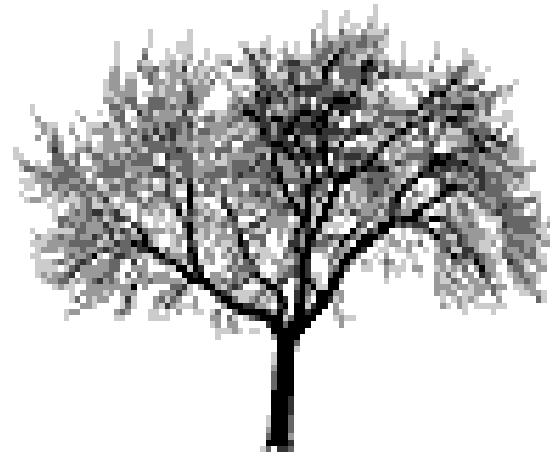
Data structure

Fall 2023

Hashemite University

Dr. Mohammad Al-hammouri

Trees

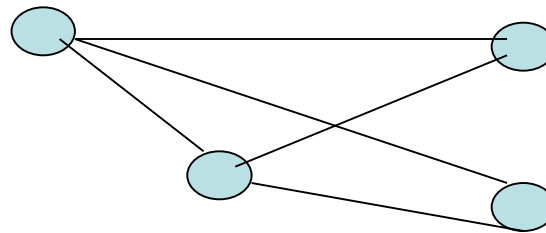


- Trees
- Binary Trees
- Properties of Binary Trees
- Traversals of Trees
- Data Structures for Trees

Trees

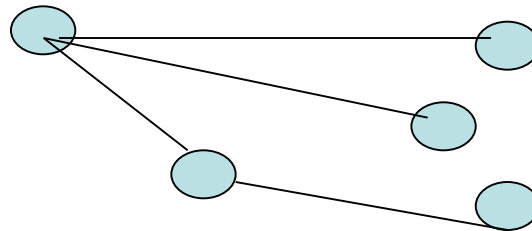
A **graph** $G = (V, E)$ consists of a set V of VERTICES

and a set E of edges, with $E = \{\{u, v\}: u, v \in V, u \neq v\}$

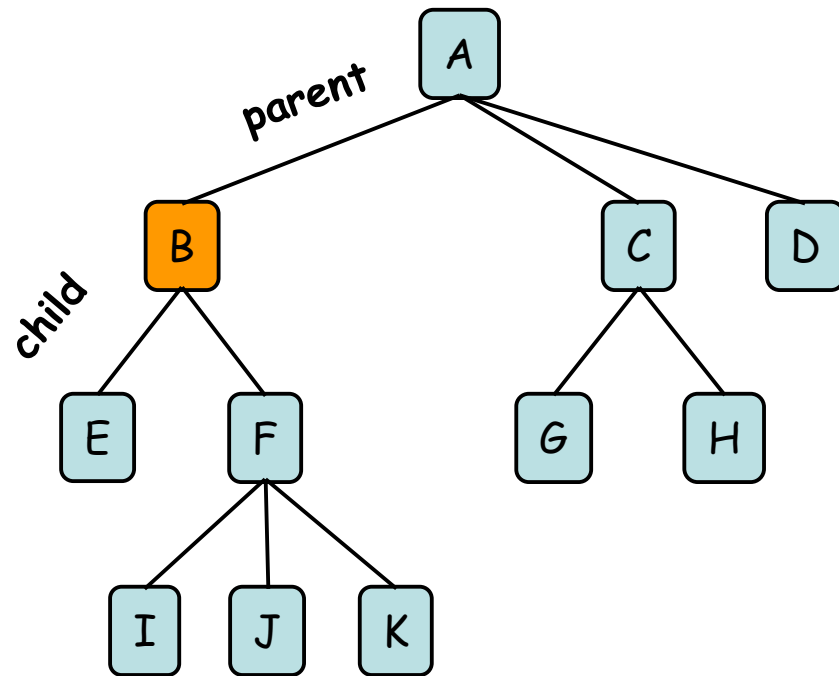


A **tree** is a connected graph with no cycles.

→ \exists a path between each pair of vertices.



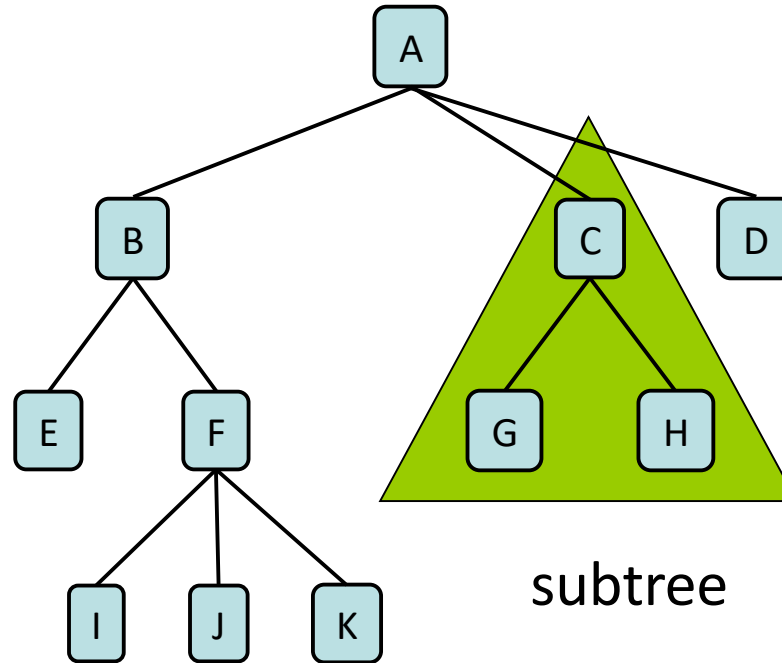
(Rooted) Trees



Tree Terminology

- **Root**: node without parent (A)
- **Internal node**: node with at least one child (A, B, C, F)
- **External node (leaf)**: node without children (E, I, J, K, G, H, D)
- **Ancestors** of a node: parent, grandparent, grand-grandparent, etc.
- **Descendant** of a node: child, grandchild, grand-grandchild, etc.

- **Subtree**: tree consisting of a node and its descendants

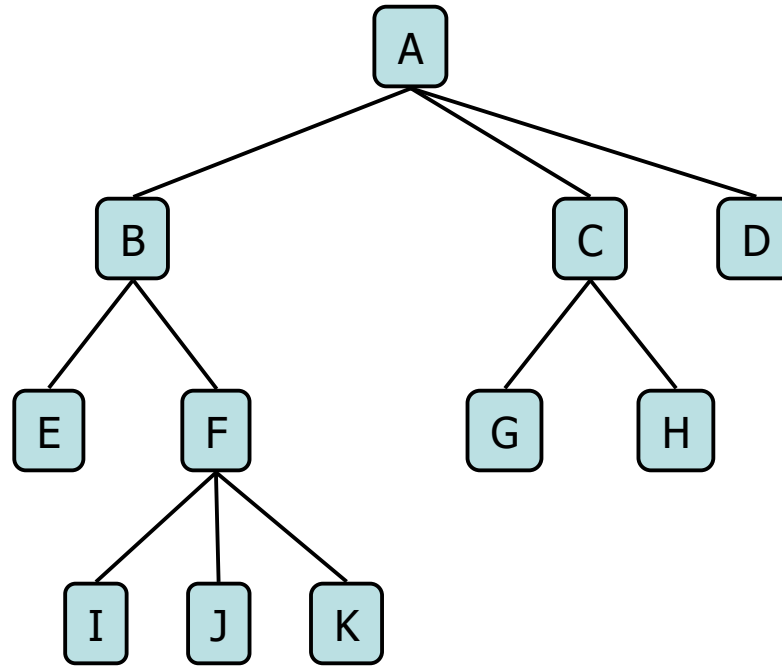


Tree Terminology

Distance between two nodes: number of “edges” between them

• **Depth** of a node: number of ancestors (= distance from the root)

• **Height** of a tree: maximum depth of any node



Example of Files System representation

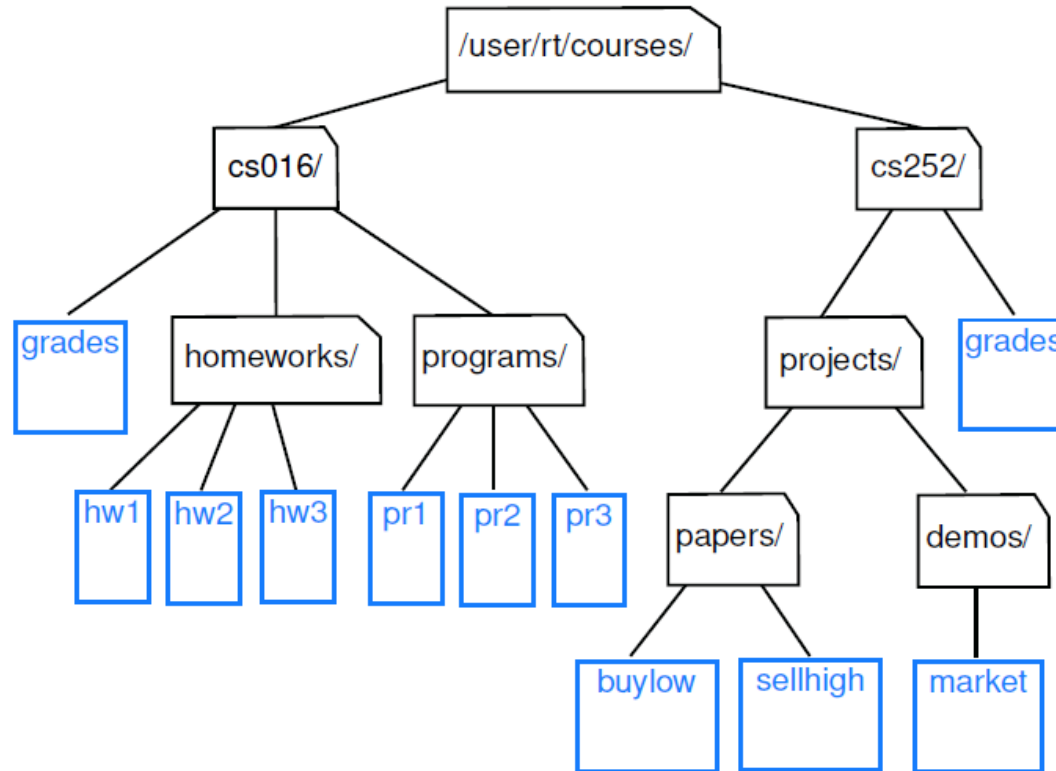
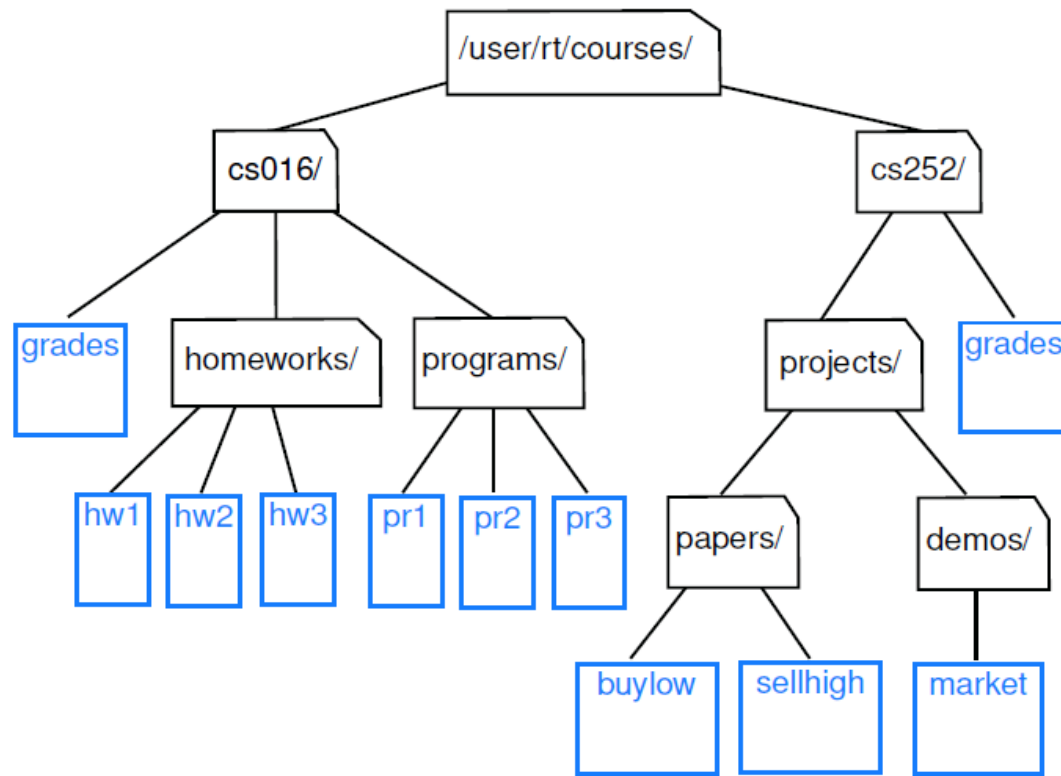


Figure 8.3: Tree representing a portion of a file system.

The internal nodes of the tree are associated with directories and the leaves are associated with regular files



cs252/ is an ancestor of papers/,

pr3 is a descendant of cs016/

Subtree rooted at cs016/ consists of the nodes cs016/, grades, homeworks/, programs/, hw1, hw2, hw3, pr1, pr2, and pr3.

Ordered Tree

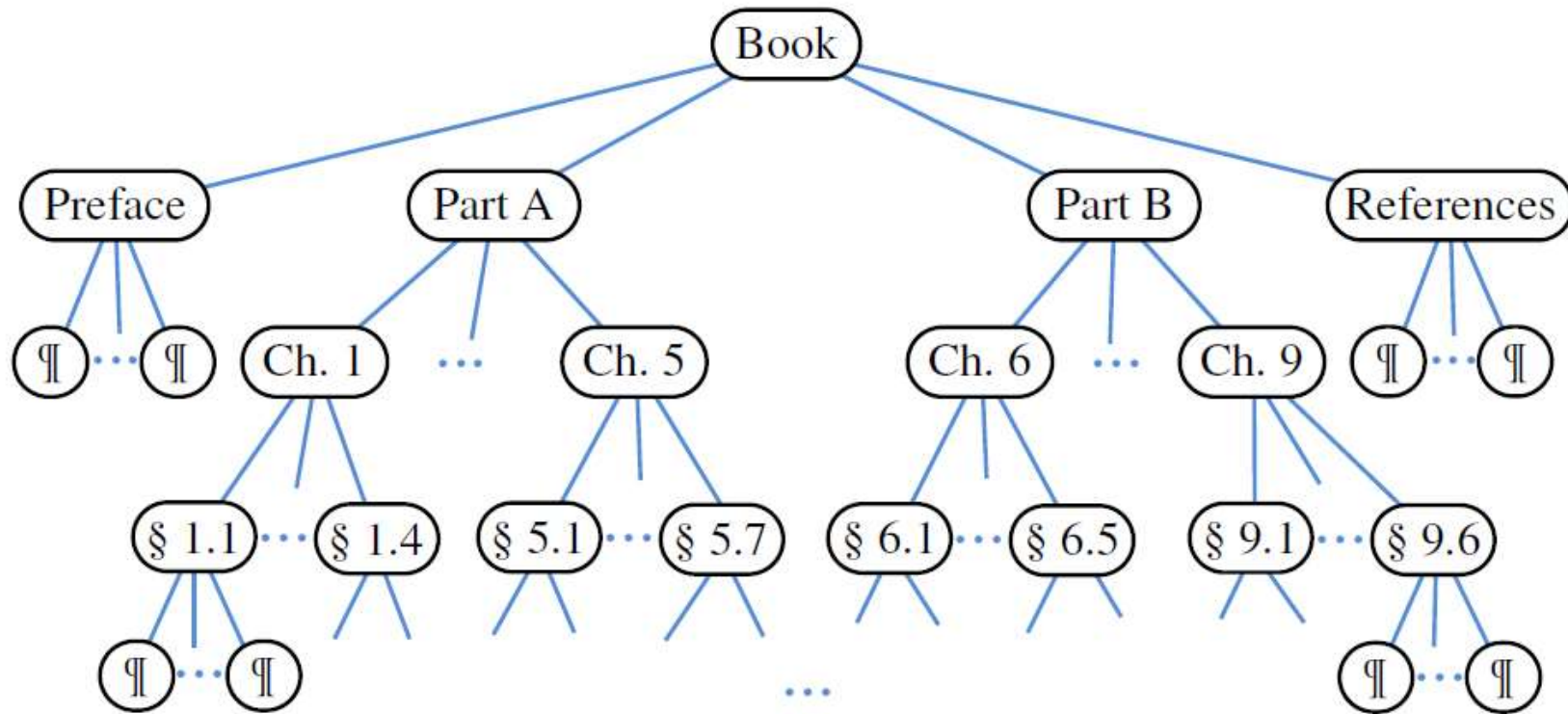


Figure 8.4: An ordered tree associated with a book.

Java Iterator (`java.util` package)

- An `Iterator` is an object that can be used to loop through collections, like `ArrayList`.
- It is called an "iterator" because "iterating" is the technical term for looping.

Java Iterator

```
import java.util.ArrayList;
import java.util.Iterator;

public class Main {
    public static void main(String[] args) {

        // Make a collection
        ArrayList<String> cars = new ArrayList<String>();
        cars.add("Volvo");
        cars.add("BMW");
        cars.add("Ford");
        cars.add("Mazda");

        // Get the iterator
        Iterator<String> it = cars.iterator();

        // Loop through a collection
        while(it.hasNext()) {
            System.out.println(it.next());
        }
    }
}
```

Position Interface

Provide a general abstraction for the location of an element within a data structure

```
1 public interface Position<E> {  
2     /**  
3      * Returns the element stored at this position.  
4      *  
5      * @return the stored element  
6      * @throws IllegalStateException if position no longer valid  
7      */  
8     E getElement() throws IllegalStateException;  
9 }
```

Code Fragment 7.7: The `Position` interface.

Position Interface

A **position** object for a tree supports the method:

`getElement()`: Returns the element stored at this position.

`root()`: Returns the position of the root of the tree
(or null if empty).

`parent(p)`: Returns the position of the parent of position p
(or null if p is the root).

`children(p)`: Returns an iterable collection containing the children of
position p (if any).

`numChildren(p)`: Returns the number of children of position p .

`isInternal(p)`: Returns true if position p has at least one child.

`isExternal(p)`: Returns true if position p does not have any children.

`isRoot(p)`: Returns true if position p is the root of the tree.

Tree Interface

```
1  /** An interface for a tree where nodes can have an arbitrary number of children. */
2  public interface Tree<E> extends Iterable<E> {
3      Position<E> root();
4      Position<E> parent(Position<E> p) throws IllegalArgumentException;
5      Iterable<Position<E>> children(Position<E> p)
6          throws IllegalArgumentException;
7      int numChildren(Position<E> p) throws IllegalArgumentException;
8      boolean isInternal(Position<E> p) throws IllegalArgumentException;
9      boolean isExternal(Position<E> p) throws IllegalArgumentException;
10     boolean isRoot(Position<E> p) throws IllegalArgumentException;
11     int size();
12     boolean isEmpty();
13     Iterator<E> iterator();
14     Iterable<Position<E>> positions();
15 }
```

Tree Abstract Class

```
/** An abstract base class providing some functionality of the Tree interface. */  
public abstract class AbstractTree<E> implements Tree<E> {  
    public boolean isInternal(Position<E> p) { return numChildren(p) > 0; }  
    public boolean isExternal(Position<E> p) { return numChildren(p) == 0; }  
    public boolean isRoot(Position<E> p) { return p == root(); }  
    public boolean isEmpty() { return size() == 0; }  
}
```

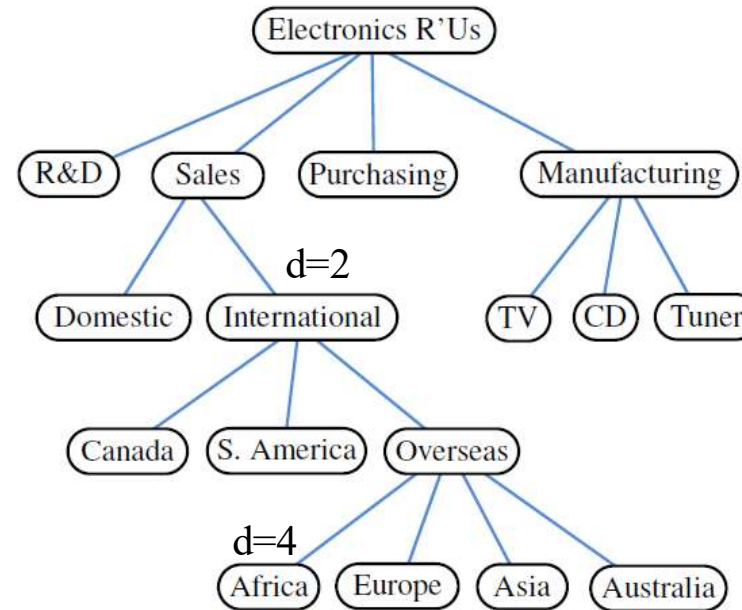

Computing Depth and Height

- The depth of \mathbf{p} is the number of ancestors of \mathbf{p} , other than \mathbf{p} itself.
- Note that this definition implies that the depth of the root of \mathbf{T} is 0. The depth of \mathbf{p} can also be recursively defined as follows:
 - If p is the root, then the depth of p is 0.
 - Otherwise, the depth of p is one plus the depth of the parent of p .

Computing Depth and Height

```
public int depth(Position<E> p) {  
    if (isRoot(p))  
        return 0;  
    else  
        return 1 + depth(parent(p));  
}
```

```
public int height(Position<E> p) {  
    int h = 0;  
    for (Position<E> c : children(p))  
        h = Math.max(h, 1 + height(c));  
    return h;  
}
```

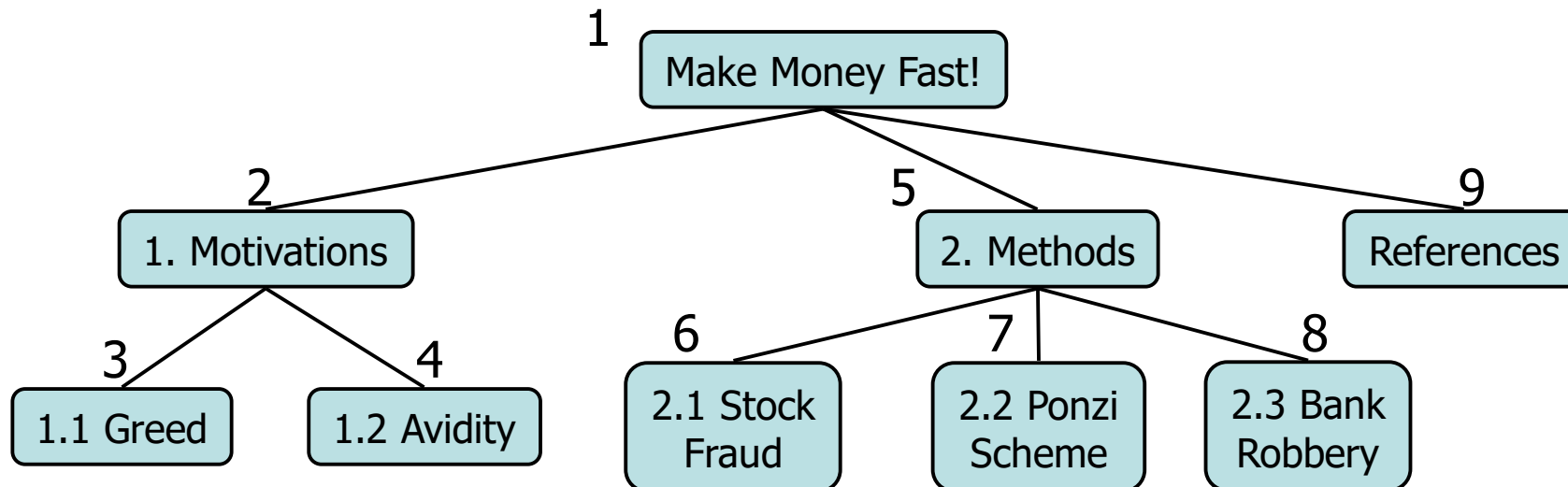


Traversing Trees

Preorder Traversal

- A traversal visits the nodes of a tree in a systematic manner
- In a preorder traversal, a **node is visited before its descendants**
- Application: print a structured document

Algorithm *preOrder*(*v*)
visit(*v*)
for each child *w* of *v*
 preOrder (*w*)

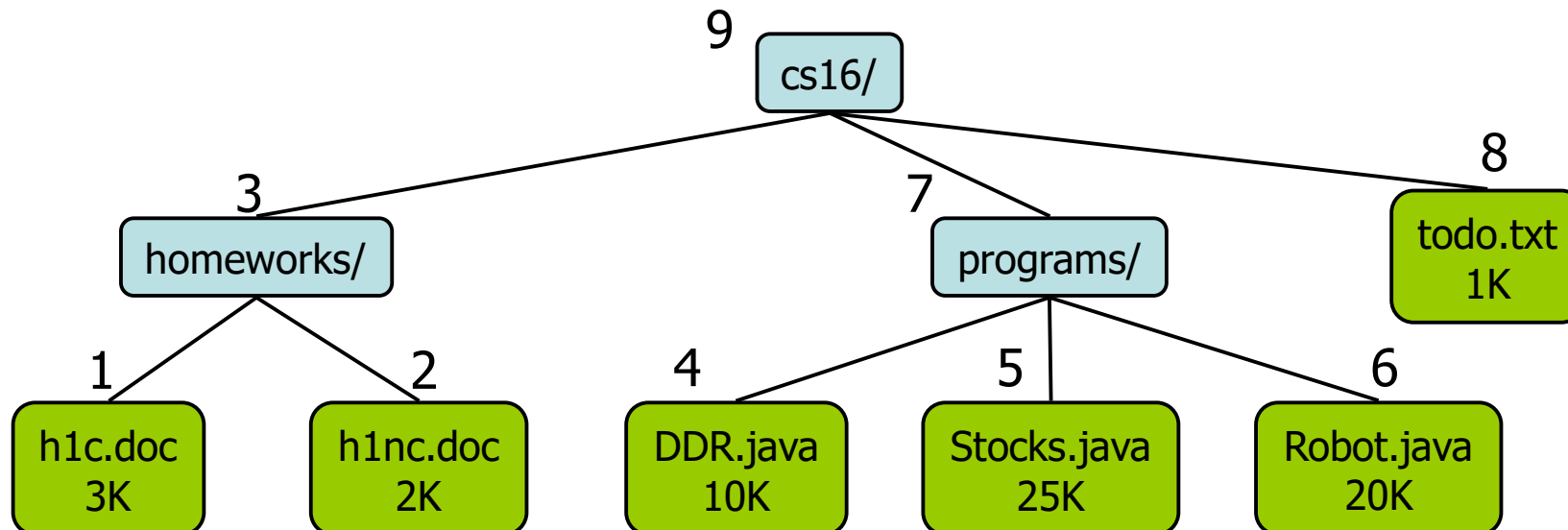


Traversing Trees

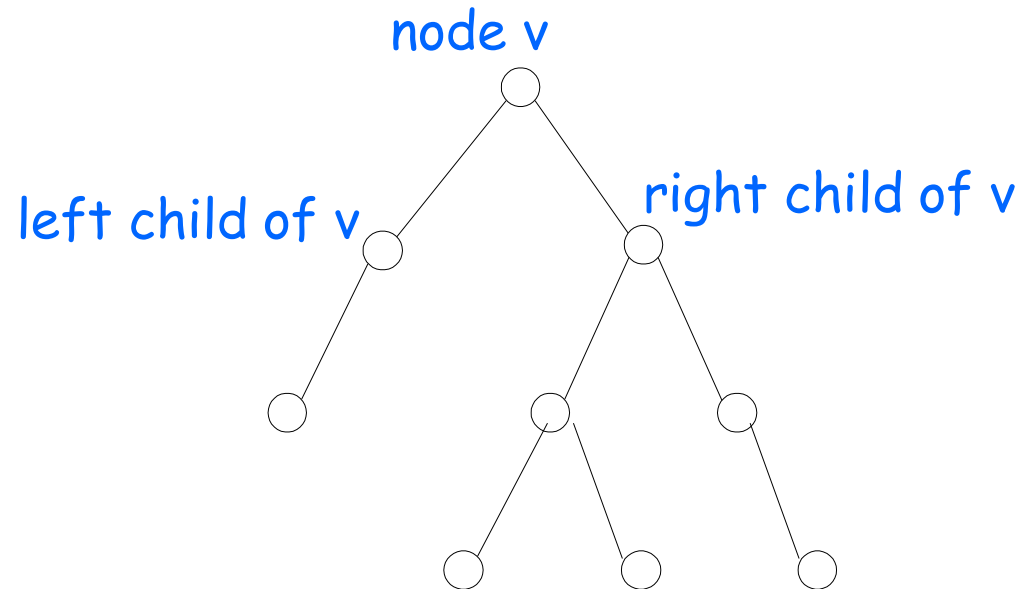
Postorder Traversal

- In a postorder traversal, a **node is visited after its descendants**
- Application: compute space used by files in a directory and its subdirectories

Algorithm *postOrder*(*v*)
 for each child *w* of *v*
 postOrder (*w*)
 visit(*v*)



Binary Trees



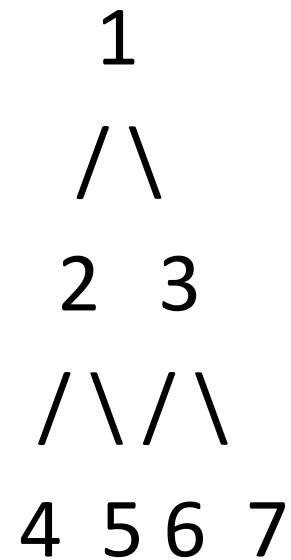
Each child node is labeled as being either a ***left child*** or a ***right child***.

Children are ordered (A left child precedes a right child in the order of children of a node.)

Each node has at most two children. (0, 1, or 2 children)

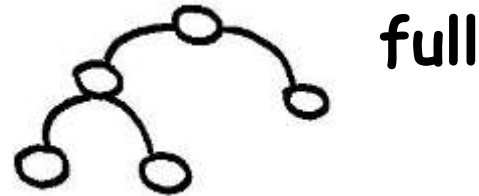
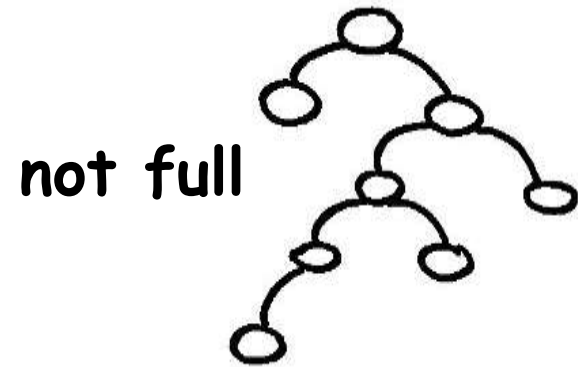
Binary Tree

- A full binary tree is a binary tree in which every node has either 0 or 2 children



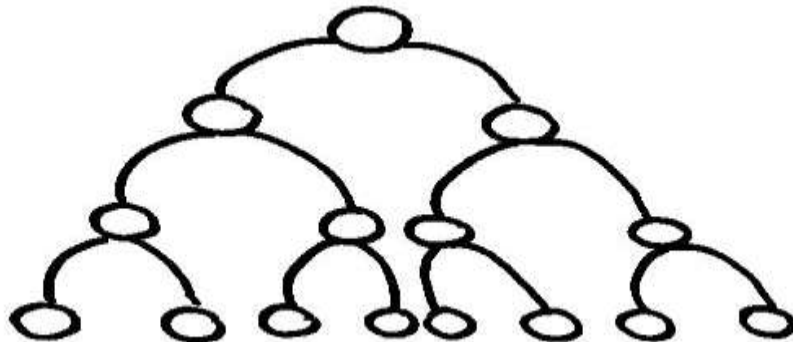
“Full” Binary Trees (or “Proper”)

Each node: $\left\{ \begin{array}{l} \text{is a leaf, or} \\ \text{has two children} \end{array} \right.$



Perfect Binary Trees

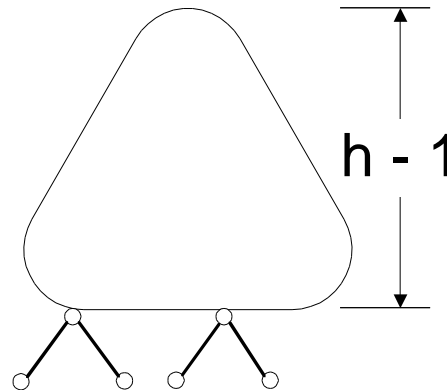
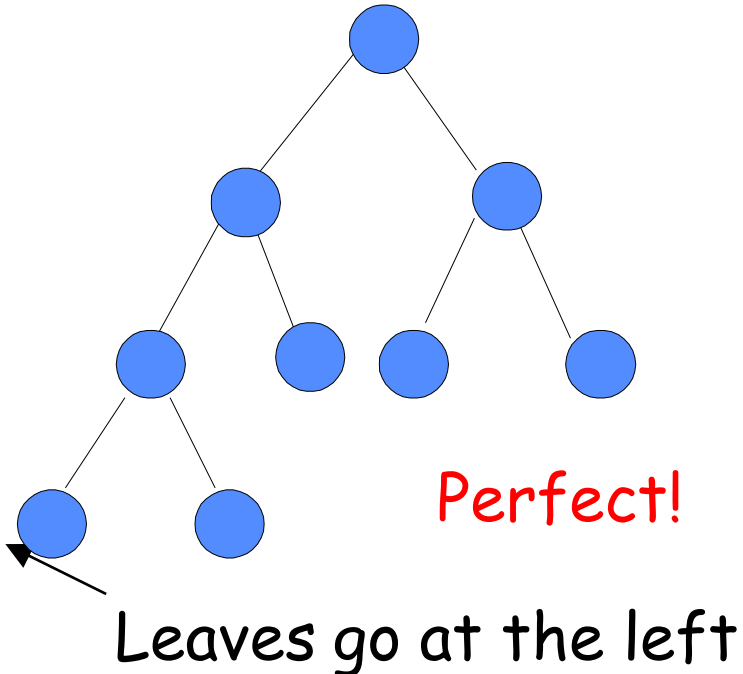
Full binary trees with all leaves at the same level:



Complete Binary Trees

Complete binary tree of depth $h =$

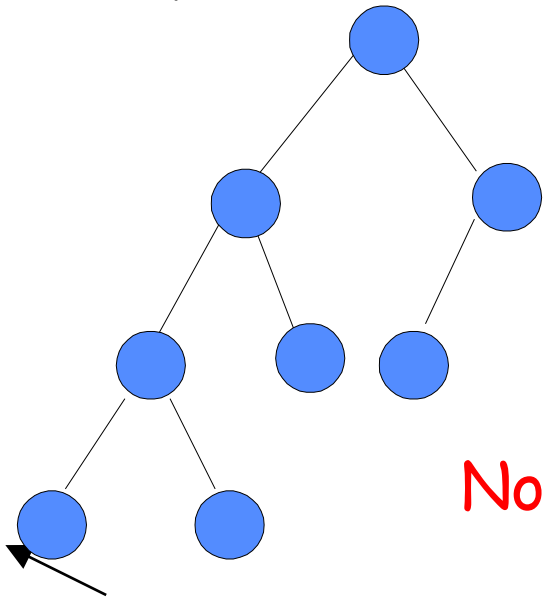
Perfect tree of depth $(h-1)$ with one or more leaves at level h which are placed as left as possible.



Complete Binary Trees

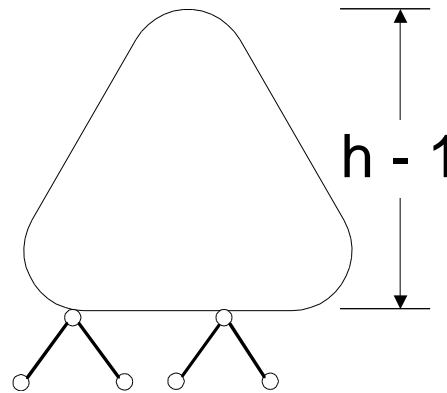
Complete binary tree of depth $h =$

Perfect tree of depth $(h-1)$ with one or more leaves at level h which are placed as left as possible.



Not Perfect!

Leaves go at the left



Binary Trees Example

internal nodes: operators
external nodes: operands

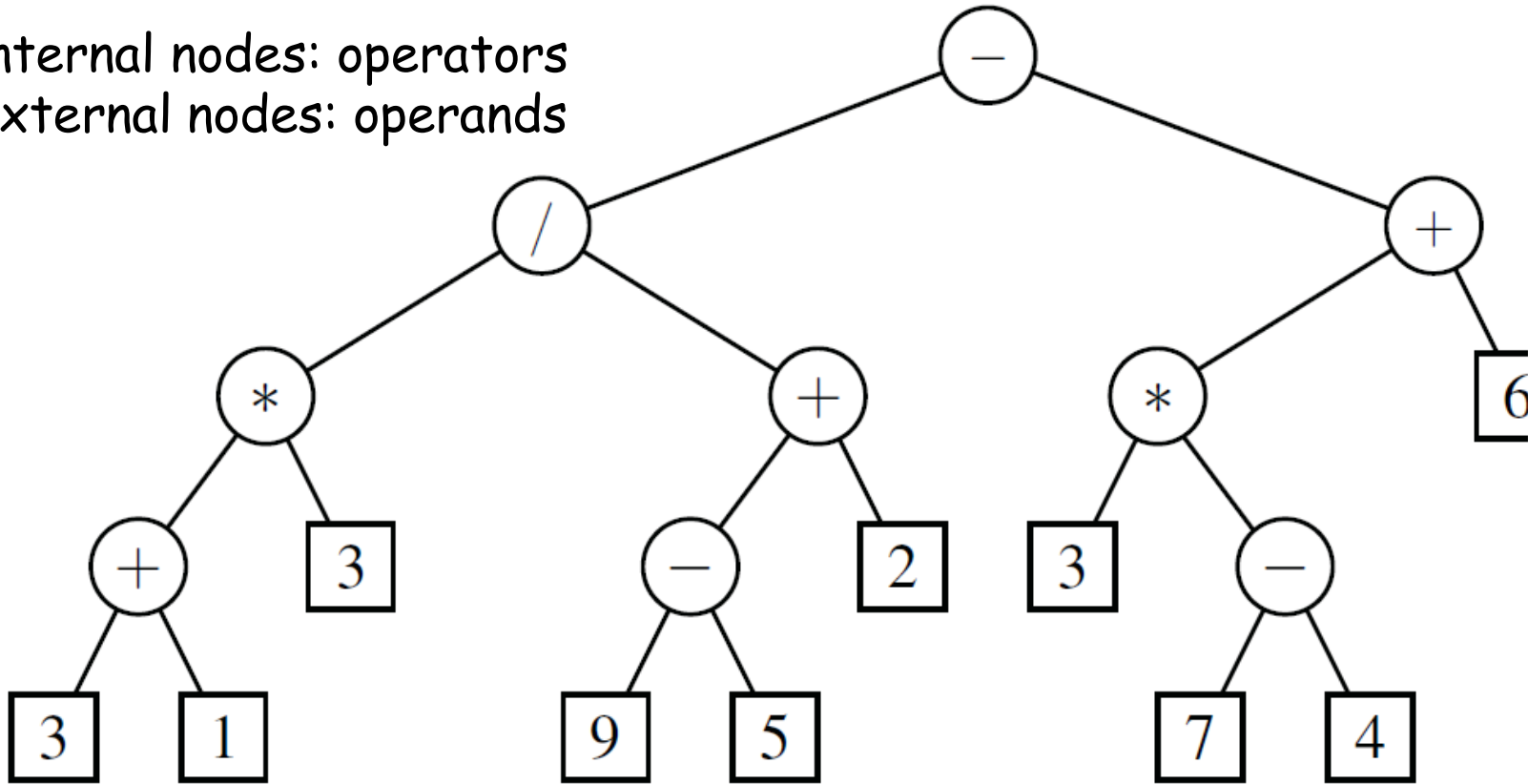
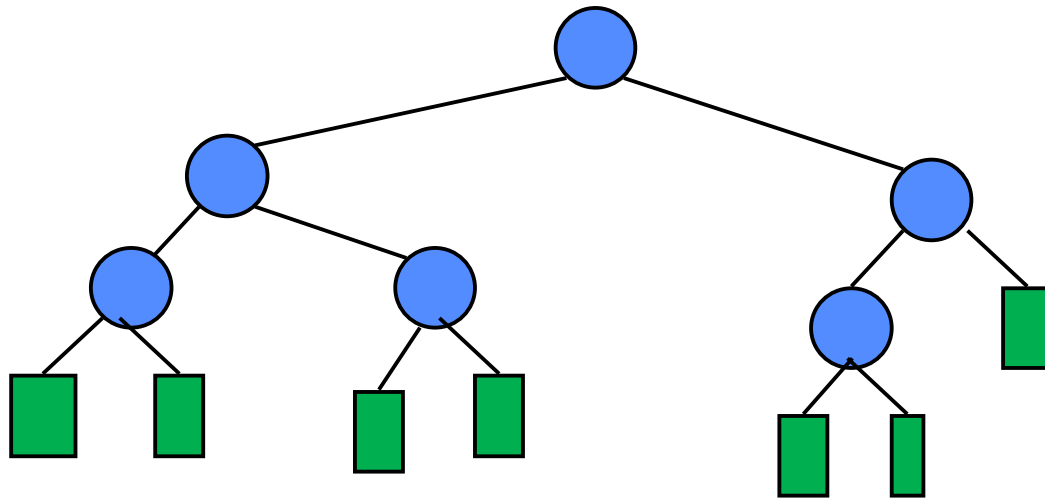


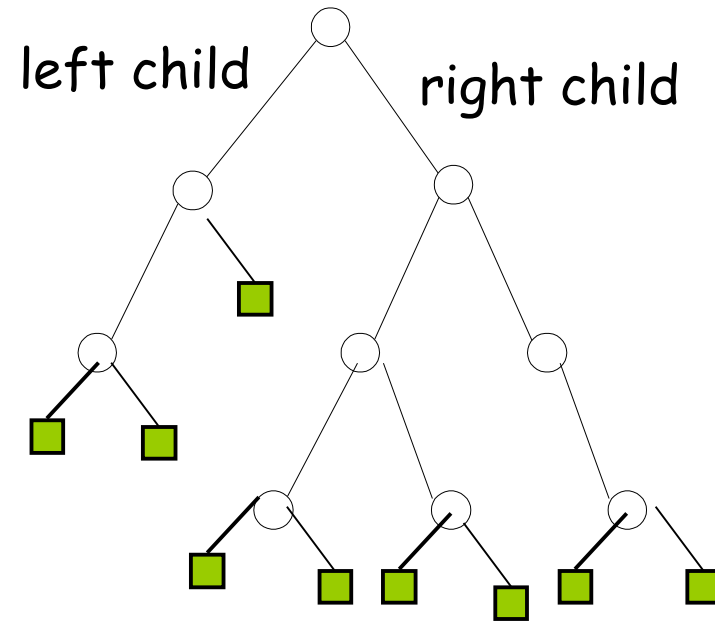
Figure 8.6: A binary tree representing an arithmetic expression. This tree represents the expression $((((3 + 1) * 3) / ((9 - 5) + 2)) - ((3 * (7 - 4)) + 6))$. The va

Binary Trees



In this book, children nodes are “completed” with “dummy” nodes and all trees are considered FULL

Binary Trees + dummy leaves



Each internal node has two children

Examples of Binary Trees

Decision Tree

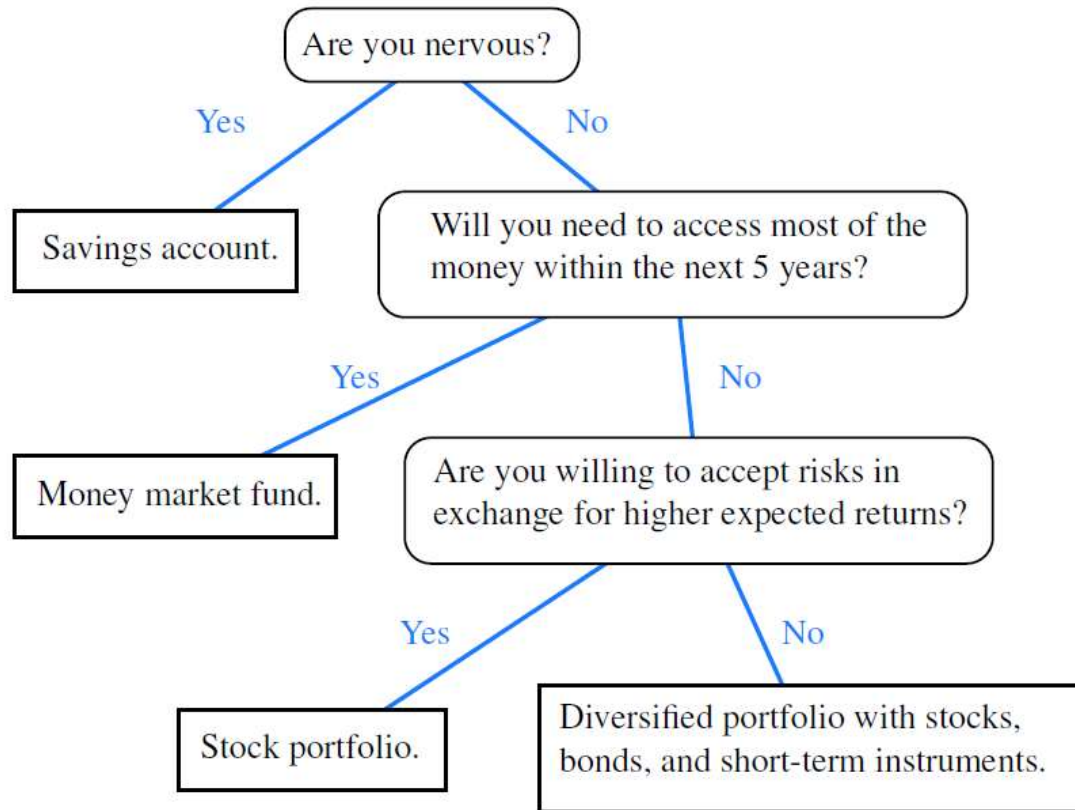


Figure 8.5: A decision tree providing investment advice.

BinaryTree interface

`left(p)`: Returns the position of the left child of *p*
(or null if *p* has no left child).

`right(p)`: Returns the position of the right child of *p*
(or null if *p* has no right child).

`sibling(p)`: Returns the position of the sibling of *p*
(or null if *p* has no sibling).

```
/** An interface for a binary tree, in which each node has at most two children. */  
public interface BinaryTree<E> extends Tree<E> {  
    /** Returns the Position of p's left child (or null if no child exists). */  
    Position<E> left(Position<E> p) throws IllegalArgumentException;  
    /** Returns the Position of p's right child (or null if no child exists). */  
    Position<E> right(Position<E> p) throws IllegalArgumentException;  
    /** Returns the Position of p's sibling (or null if no sibling exists). */  
    Position<E> sibling(Position<E> p) throws IllegalArgumentException;  
}
```

Code Fragment 8.6: A BinaryTree interface that extends the Tree interface from

Binary Trees implementation

/** Returns the Position of p's sibling (or null if no sibling exists). */

```
public Position<E> sibling(Position<E> p) {  
    Position<E> parent = parent(p);  
    if (parent == null) return null;           // p must be the root  
    if (p == left(parent))                    // p is a left child  
        return right(parent);                // (right child might be null)  
    else                                       // p is a right child  
        return left(parent);                 // (left child might be null)  
}
```

/** Returns an iterable collection of the Positions representing p's children. */

```
public Iterable<Position<E>> children(Position<E> p) {  
    List<Position<E>> snapshot = new ArrayList<>(2); // max capacity of 2  
    if (left(p) != null)  
        snapshot.add(left(p));  
    if (right(p) != null)  
        snapshot.add(right(p));  
    return snapshot;  
}
```

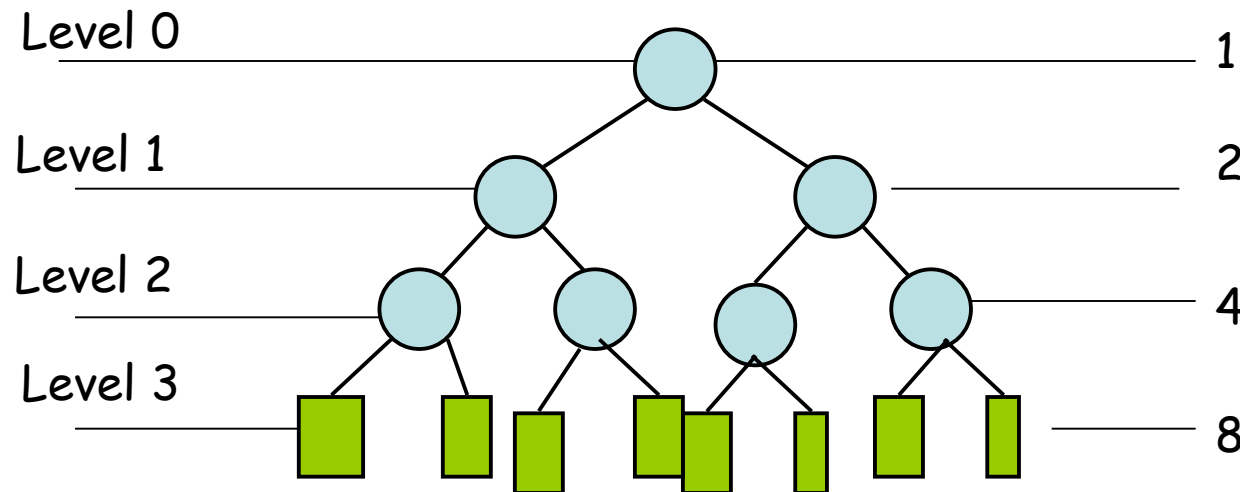
Properties of Binary Trees

- Notation

n # of nodes, e # of leaves

i # of internal nodes, h height

Maximum number of
nodes at each level ?



level i ----- 2^i

Properties of Full Binary Trees

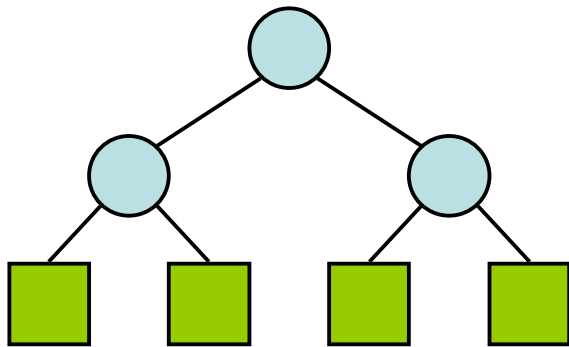
- Notation

n number of nodes

e number of leaves

i number of
internal nodes

h height



- Some Properties:

- $e = i + 1$

- $n = 2e - 1$

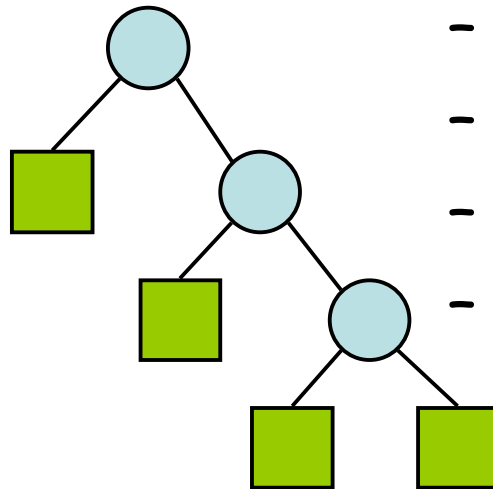
- $h \leq i$

- $h \leq (n - 1)/2$

- $e \leq 2^h$

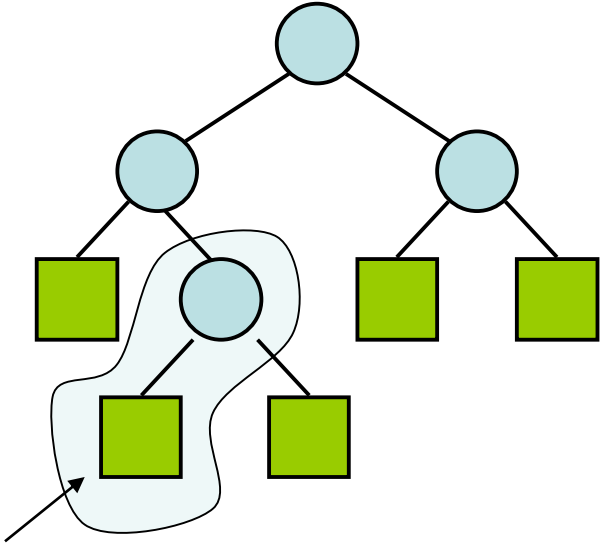
- $h \geq \log_2 e$

- $h \geq \log_2 (n + 1) - 1$

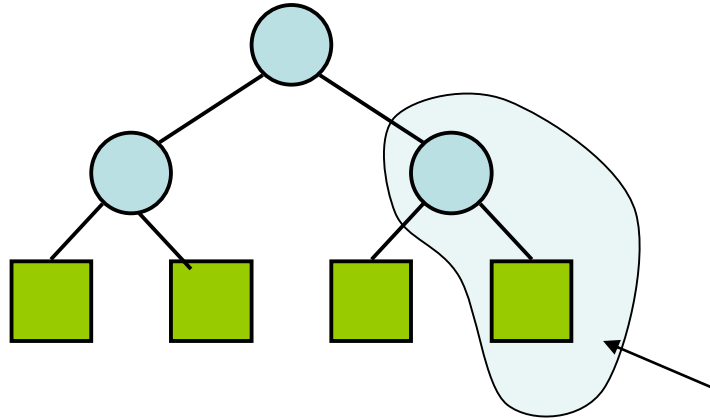
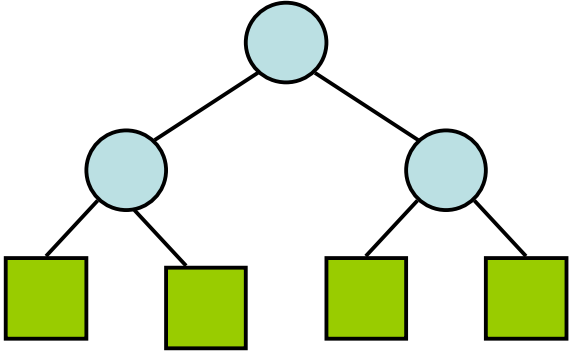


$$e = i + 1$$

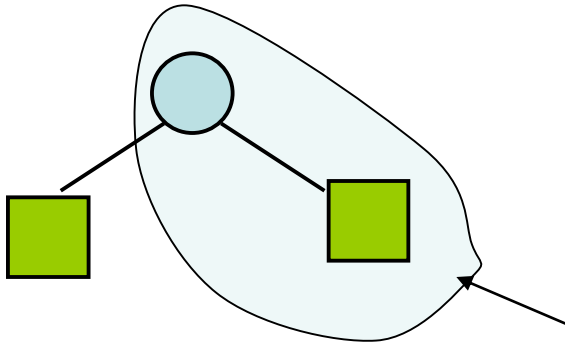
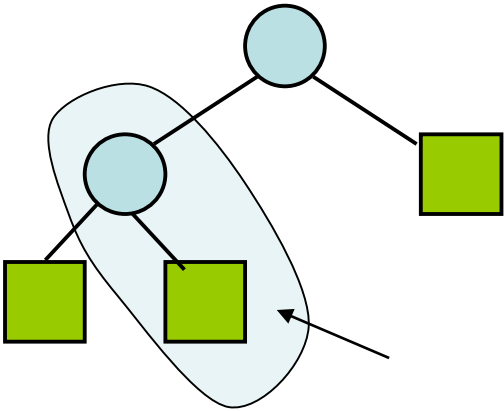
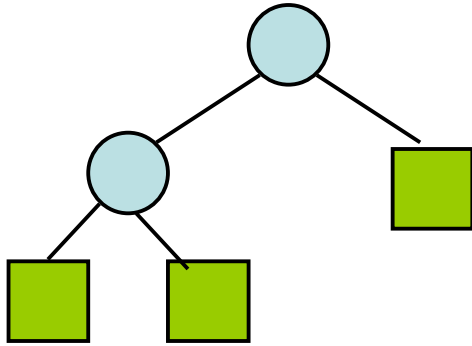
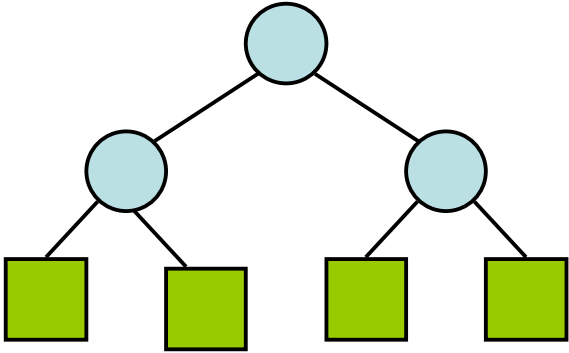
e number of leaves
 i number of internal nodes



$$e = i + 1$$



$$e = i + 1$$



$$n = 2e - 1$$

$$n = i + e$$

$$e = i + 1 \text{ (just proved)}$$

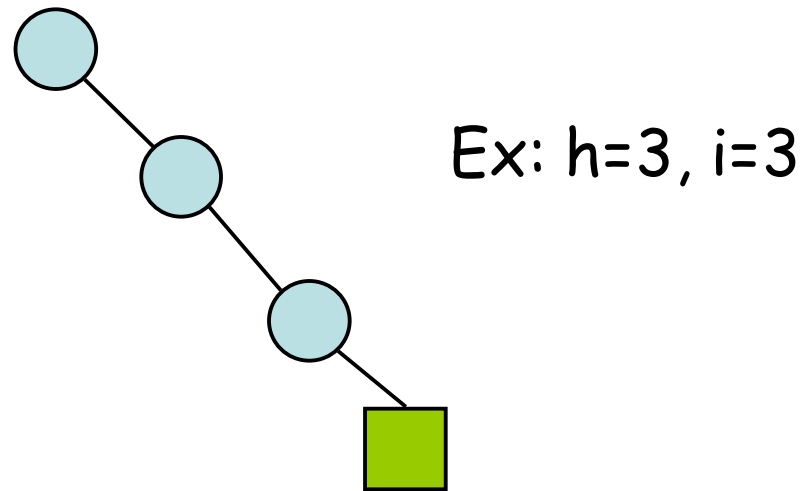
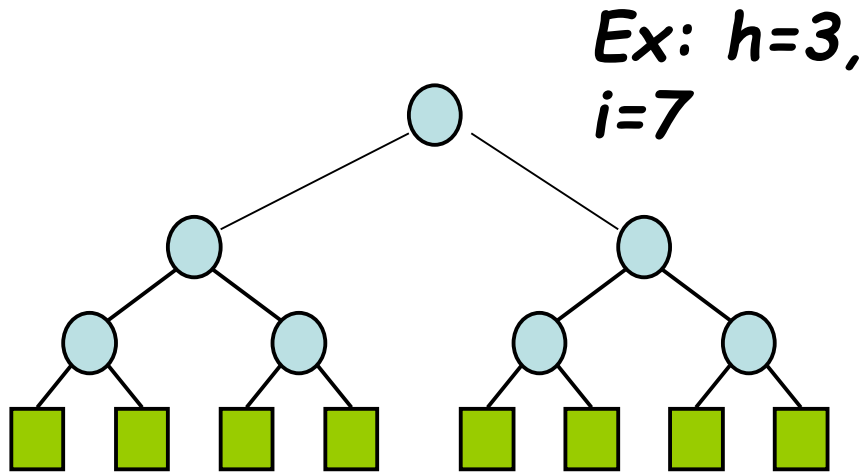
$$i = e - 1$$

$$n = e - 1 + e = 2e - 1$$

$$h \leq i$$

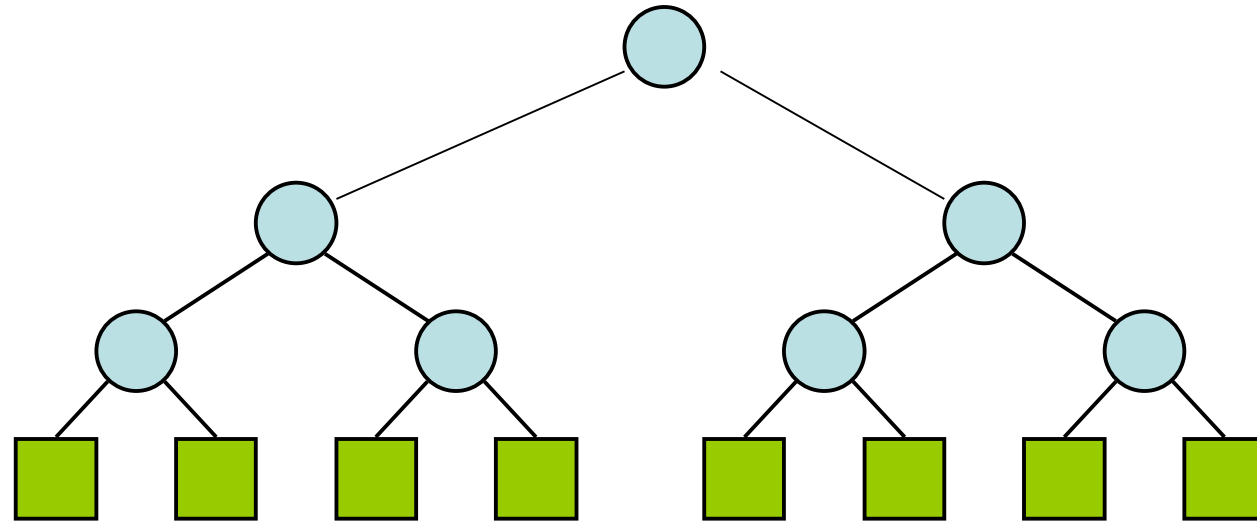
(h = max n. of ancestors)

There must be at least one internal node for each level
(except the last) !



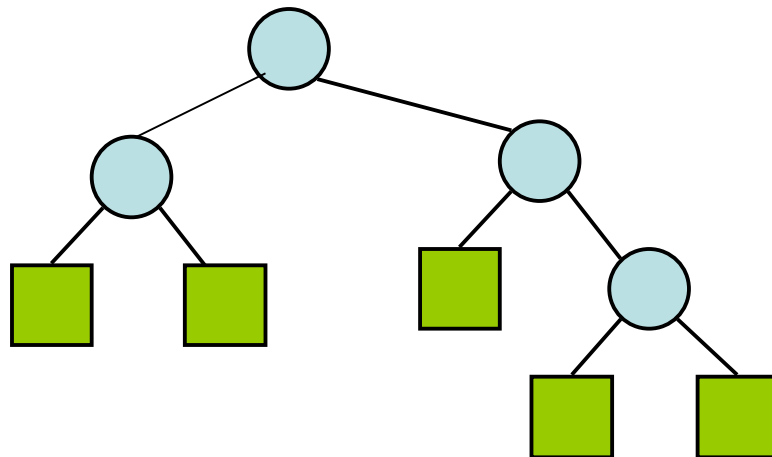
$$e \leq 2^h$$

level i ----- max n. of nodes is 2^i



$h = 3$

2^3 leaves
if all leaves
at last level h



otherwise less

Since $e \leq 2^h$

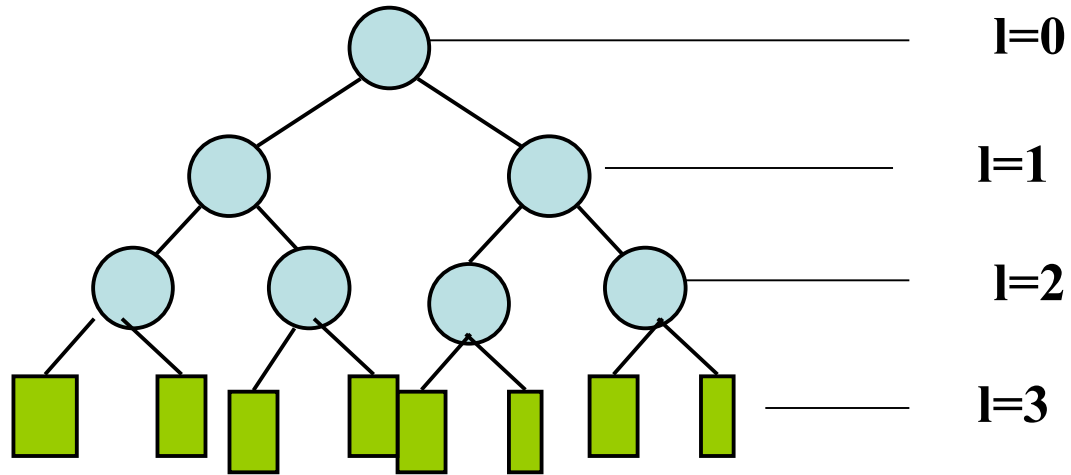
$$\log_2 e \leq \log_2 2^h$$

$$\log_2 e \leq h \quad \longrightarrow$$

$$h \geq \log_2 e$$

In Perfect Binary Trees

$$n = 2^{h+1} - 1$$



$l=0$

As a consequence:

In **Binary trees**:

obviously $n \leq 2^{h+1} - 1$

$$n \leq 2^{h+1} - 1$$

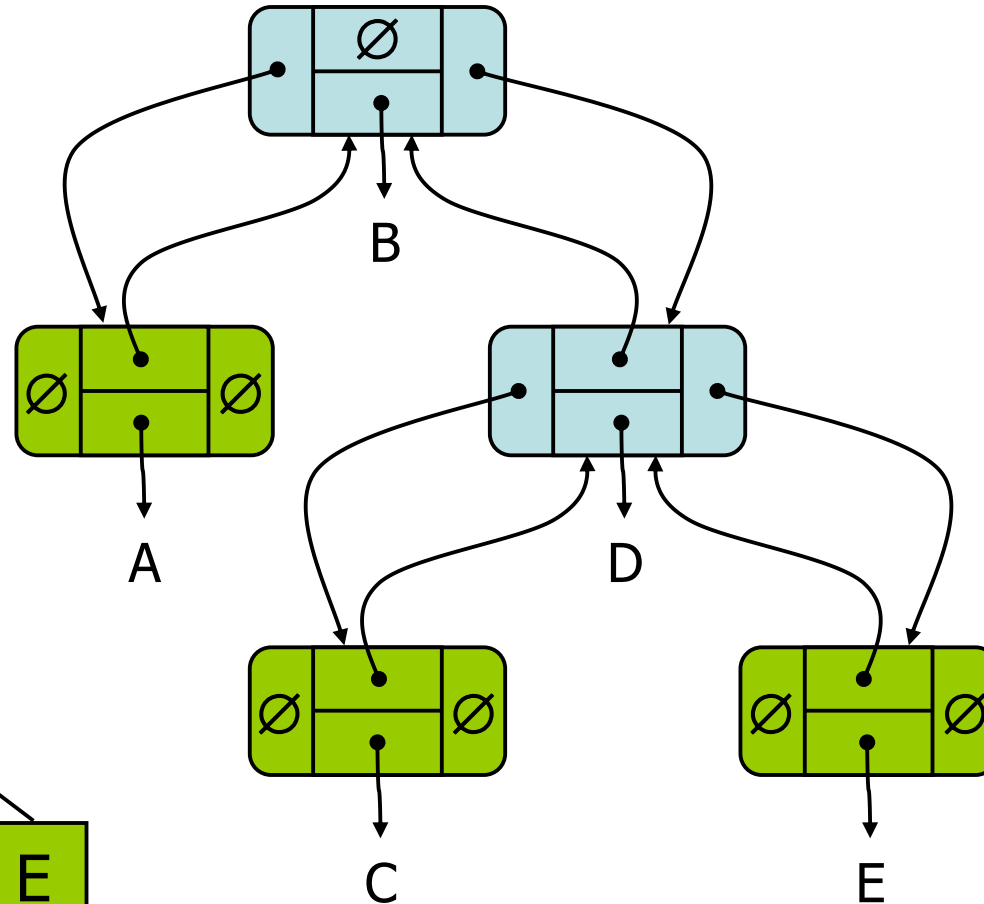
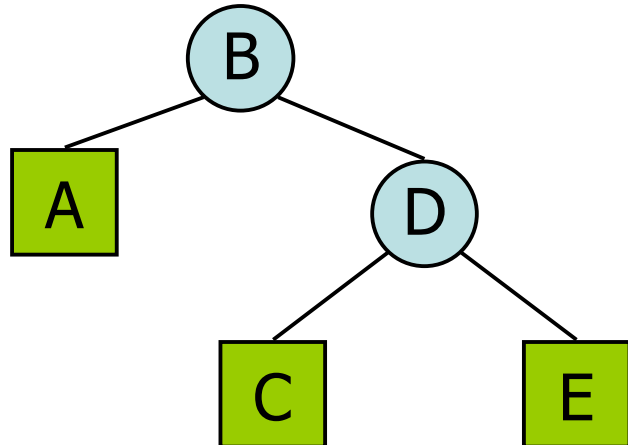
$$n+1 \leq 2^{h+1}$$

$$\log_2 (n+1) \leq h+1$$

$$h \geq \log_2 (n+1) - 1$$

Implementing Binary Trees with a Linked Structure

- A node is represented by an object storing
 - Element
 - Parent node
 - Left child node
 - Right child node
- Node objects implement the Position ADT



Implementing Binary Trees with a Linked Structure

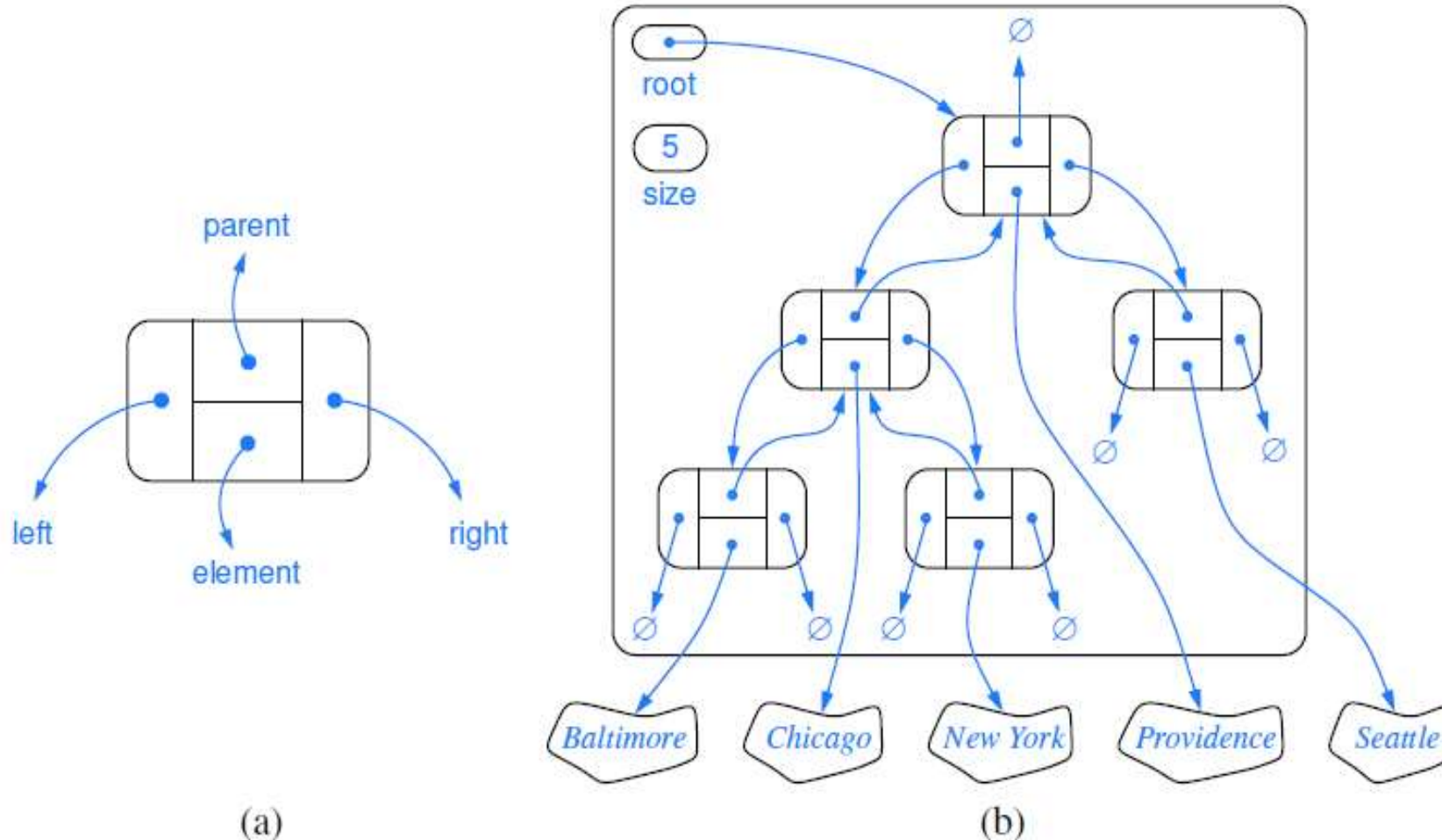


Figure 8.9: A linked structure for representing: (a) a single node; (b) a binary tree.

Operations for Updating a Linked Binary Tree

`addRoot(e)`: Creates a root for an empty tree, storing e as the element, and returns the position of that root; an error occurs if the tree is not empty.

`addLeft(p, e)`: Creates a left child of position p , storing element e , and returns the position of the new node; an error occurs if p already has a left child.

`addRight(p, e)`: Creates a right child of position p , storing element e , and returns the position of the new node; an error occurs if p already has a right child.

Operations for Updating a Linked Binary Tree

$\text{set}(p, e)$: Replaces the element stored at position p with element e , and returns the previously stored element.

$\text{attach}(p, T_1, T_2)$: Attaches the internal structure of trees T_1 and T_2 as the respective left and right subtrees of leaf position p and resets T_1 and T_2 to empty trees; an error condition occurs if p is not a leaf.

$\text{remove}(p)$: Removes the node at position p , replacing it with its child (if any), and returns the element that had been stored at p ; an error occurs if p has two children.

An implementation of the LinkedBinaryTree class

```
public class LinkedBinaryTree<E> extends AbstractBinaryTree<E> {  
  
    //----- nested Node class -----  
    protected static class Node<E> implements Position<E> {  
        private E element;           // an element stored at this node  
        private Node<E> parent;      // a reference to the parent node (if any)  
        private Node<E> left;        // a reference to the left child (if any)  
        private Node<E> right;       // a reference to the right child (if any)  
        /** Constructs a node with the given element and neighbors. */  
        public Node(E e, Node<E> above, Node<E> leftChild, Node<E> rightChild) {  
            element = e;  
            parent = above;  
            left = leftChild;  
            right = rightChild;  
        }  
    }  
}
```

An implementation of the `LinkedBinaryTree` class

```
protected Node<E> createNode(E e, Node<E> parent,  
                             Node<E> left, Node<E> right) {  
    return new Node<E>(e, parent, left, right);  
}  
  
// LinkedBinaryTree instance variables  
protected Node<E> root = null;           // root of the tree  
private int size = 0;                     // number of nodes in the tree
```


An implementation of the `LinkedBinaryTree` class

```
protected Node<E> validate(Position<E> p) throws IllegalArgumentException {
    if (!(p instanceof Node))
        throw new IllegalArgumentException("Not valid position type");
    Node<E> node = (Node<E>) p;           // safe cast
    if (node.getParent() == node)        // our convention for defunct node
        throw new IllegalArgumentException("p is no longer in the tree");
    return node;
}

public Position<E> parent(Position<E> p) throws IllegalArgumentException {
    Node<E> node = validate(p);
    return node.getParent();
}

/** Returns the Position of p's left child (or null if no child exists). */
public Position<E> left(Position<E> p) throws IllegalArgumentException {
    Node<E> node = validate(p);
    return node.getLeft();
}
```

An implementation of the `LinkedBinaryTree` class

```
public Position<E> addRoot(E e) throws IllegalStateException {
    if (!isEmpty()) throw new IllegalStateException("Tree is not empty");
    root = createNode(e, null, null, null);
    size = 1;
    return root;
}

/** Creates a new left child of Position p storing element e; returns its Position. */
public Position<E> addLeft(Position<E> p, E e)
    throws IllegalArgumentException {
    Node<E> parent = validate(p);
    if (parent.getLeft() != null)
        throw new IllegalArgumentException("p already has a left child");
    Node<E> child = createNode(e, parent, null, null);
    parent.setLeft(child);
    size++;
    return child;
}
```

An implementation of the LinkedBinaryTree class

```
public void attach(Position<E> p, LinkedBinaryTree<E> t1,
                    LinkedBinaryTree<E> t2) throws IllegalArgumentException {
    Node<E> node = validate(p);
    if (isInternal(p)) throw new IllegalArgumentException("p must be a leaf");
    size += t1.size() + t2.size();
    if (!t1.isEmpty()) {                               // attach t1 as left subtree of node
        t1.root.setParent(node);
        node.setLeft(t1.root);
        t1.root = null;
        t1.size = 0;
    }
    if (!t2.isEmpty()) {                               // attach t2 as right subtree of node
        t2.root.setParent(node);
        node.setRight(t2.root);
        t2.root = null;
        t2.size = 0;
    }
}
```

// Remove the node at Position p and replace it with its child, if any. //

An implementation of the LinkedBinaryTree class

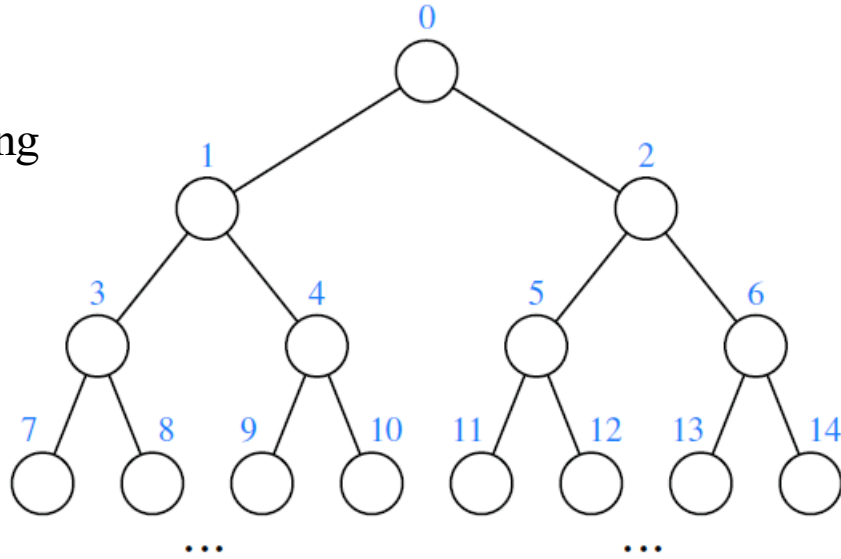
```
public E remove(Position<E> p) throws IllegalArgumentException {
    Node<E> node = validate(p);
    if (numChildren(p) == 2)
        throw new IllegalArgumentException("p has two children");
    Node<E> child = (node.getLeft() != null ? node.getLeft() : node.getRight());
    if (child != null)
        child.setParent(node.getParent()); // child's grandparent becomes its parent
    if (node == root)
        root = child; // child becomes root
    else {
        Node<E> parent = node.getParent();
        if (node == parent.getLeft())
            parent.setLeft(child);
        else
            parent.setRight(child);
    }
    size--;
    E temp = node.getElement();
    node.setElement(null); // help garbage collection
    node.setLeft(null);
    node.setRight(null);
    node.setParent(null); // our convention for defunct node
    return temp;
}
```


Array-Based Representation of a Binary Tree

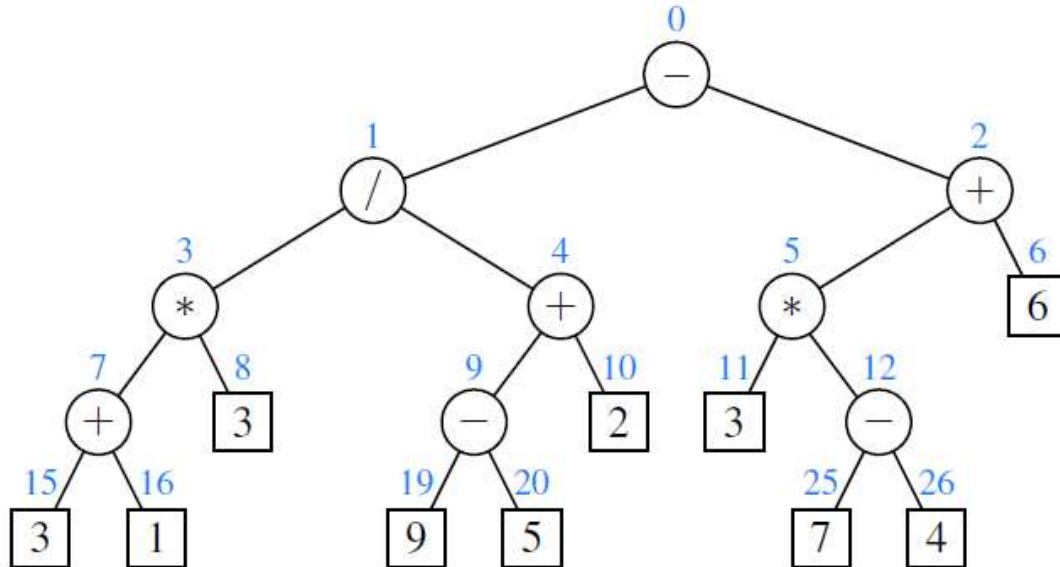
- An alternative representation of a binary tree T is based on a way of numbering the positions of T .
- For every position p of T , let $f(p)$ be the integer defined as follows
 - If p is the root of T , then $f(p) = 0$.
 - If p is the left child of position q , then $f(p) = 2f(q) + 1$.
 - If p is the right child of position q , then $f(p) = 2f(q) + 2$.

Array-Based Representation of a Binary Tree

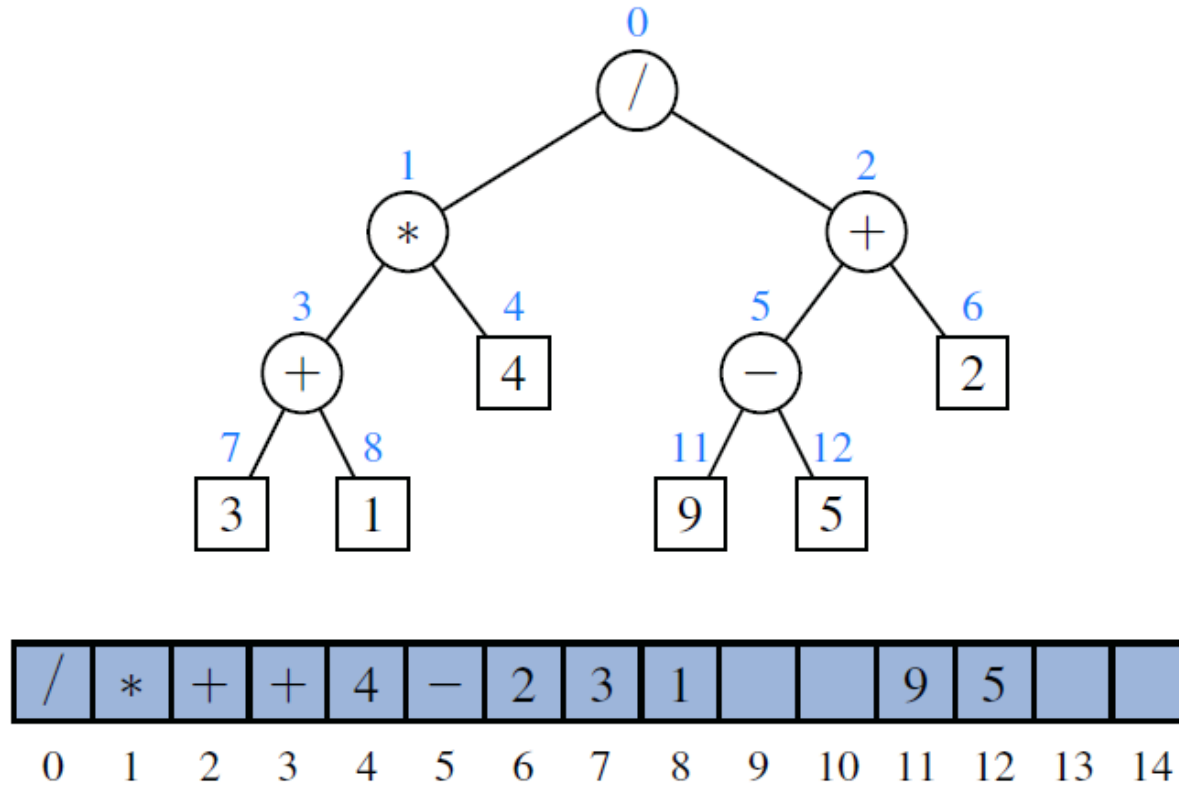
General numbering



Example



Array-Based Representation of a Binary Tree



Drawback of an array representation is that many update operations for trees cannot be efficiently supported. For example, removing a node and promoting its child takes $O(n)$

Traversing Binary Trees

Pre-, post-, in- (order)

- Refer to the place of the parent relative to the children
- **pre** is before: parent, child, child
- **post** is after: child, child, parent
- **in** is in between: child, parent, child

Traversing Binary Trees (Preorder)

Algorithm preOrder(T,v)

visit(v)

if v is internal:

preOrder (T,T.LeftChild(v))

preOrder (T,T.RightChild(v))

Algorithm preorder(p):

perform the “visit” action for position p { this happens before any recursion }

for each child c in children(p) **do**

 preorder(c) { recursively traverse the subtree rooted at c }

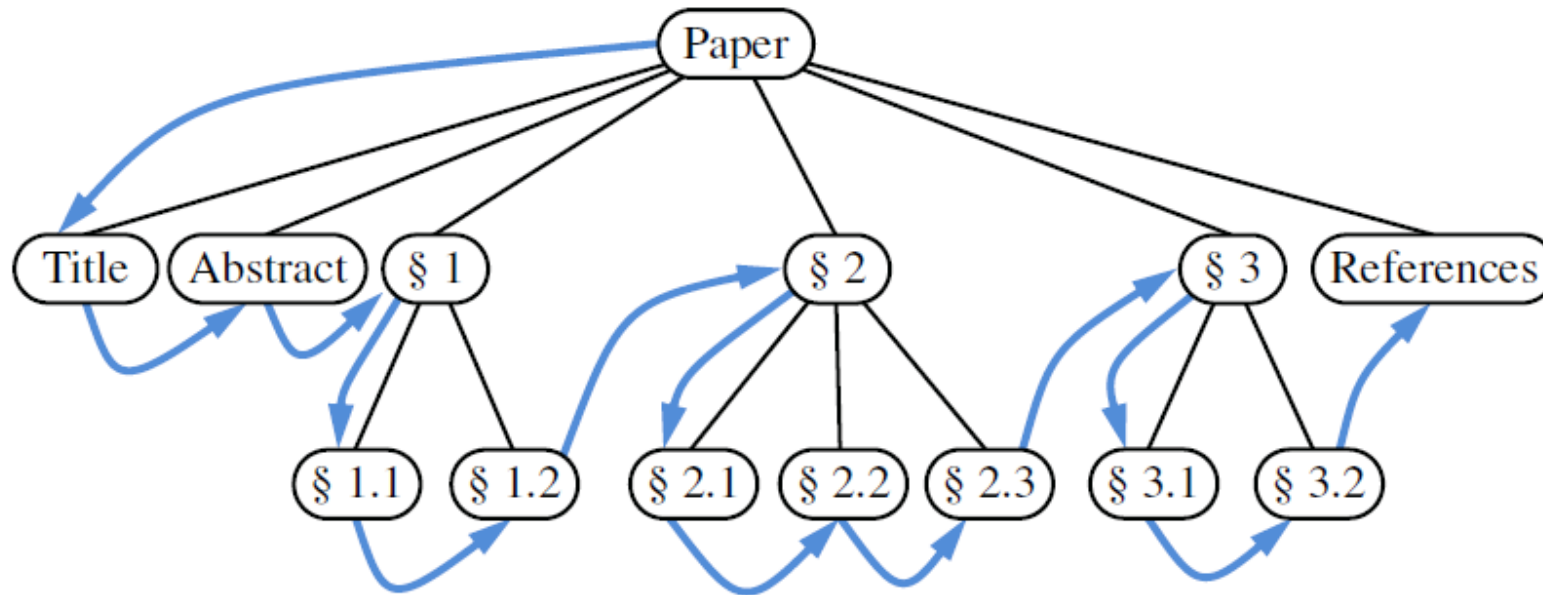
Traversing Binary Trees (Preorder)

Algorithm preorder(p):

perform the “visit” action for position p { this happens before any recursion }

for each child c in children(p) **do**

 preorder(c) { recursively traverse the subtree rooted at c }



Traversing Binary Trees (Post -order)

Algorithm postOrder(T, v)

if v is internal:

```
postOrder(T,T.LeftChild(v))
```

```
postOrder(T,T.RightChild(v))
```

visit(v)

Algorithm postorder(p):

for each child c in $\text{children}(p)$ **do**

postorder(c) { recursively traverse the subtree rooted at c }

perform the “visit” action for position p { this happens after any recursion }

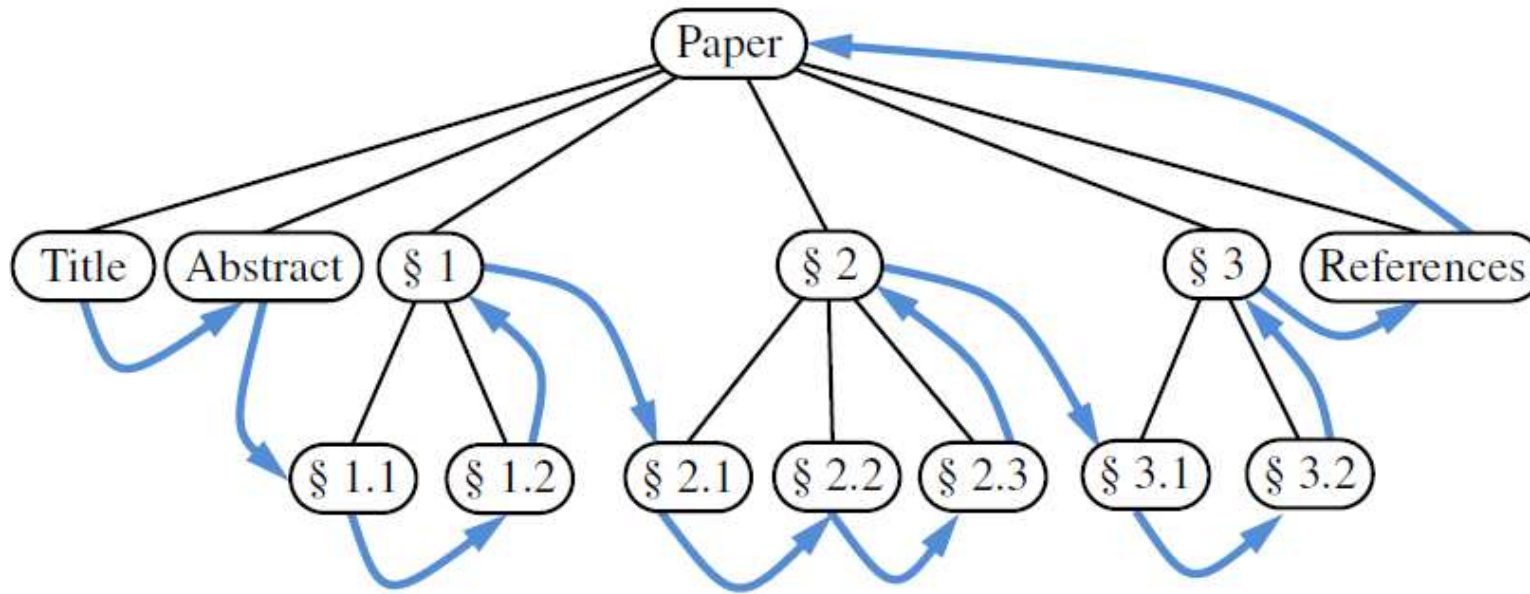
Traversing Binary Trees (Post -order)

Algorithm postorder(p):

for each child c in children(p) **do**

 postorder(c) { recursively traverse the subtree rooted at c }

 perform the “visit” action for position p { this happens after any recursion }



```

Algorithm inOrder(T,v)
    if v is internal:
        inOrder (T,T.LeftChild(v))
    visit(v)
    if v is internal:
        inOrder(T,T.RightChild(v))

```

if v is internal:

inOrder (T,T.LeftChild(v))

visit(v)

if v is internal:

inOrder(T,T.RightChild(v))

Algorithm inorder(p):

if p has a left child lc then

$$\text{inorder}(lc) \quad \{ \text{recursively traverse the left subtree of } p \}$$

perform the “visit” action for position p

if p has a right child rc **then**

inorder(rc) { recursively traverse the right subtree of p }

Traversing Binary Trees (inOrder)

Algorithm $\text{inorder}(p)$:

if p has a left child lc **then**

$\text{inorder}(lc)$

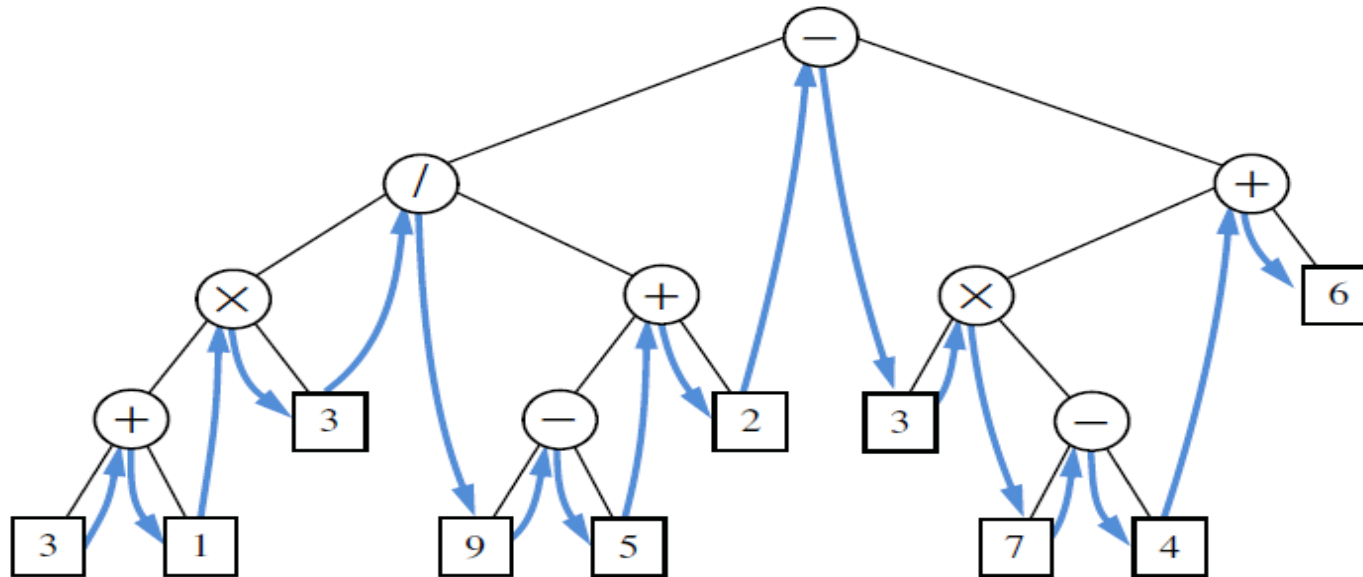
{ recursively traverse the left subtree of p }

perform the “visit” action for position p

if p has a right child rc **then**

$\text{inorder}(rc)$

{ recursively traverse the right subtree of p }



In-Order Traversing Binary Search Trees

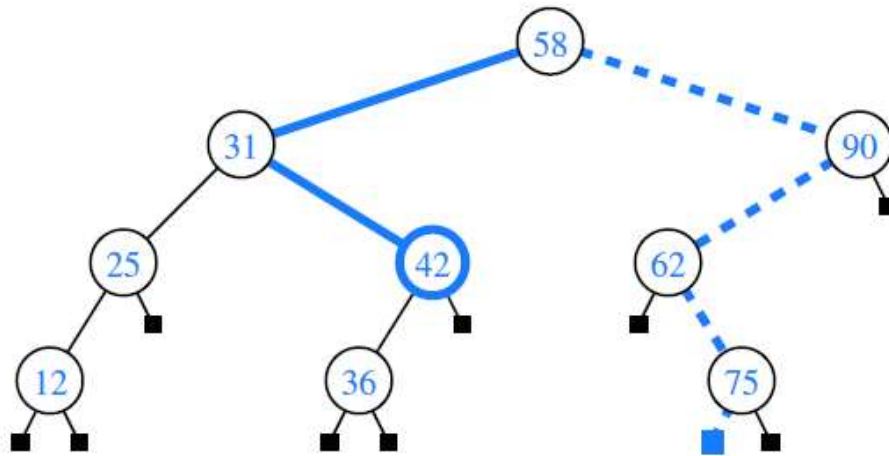
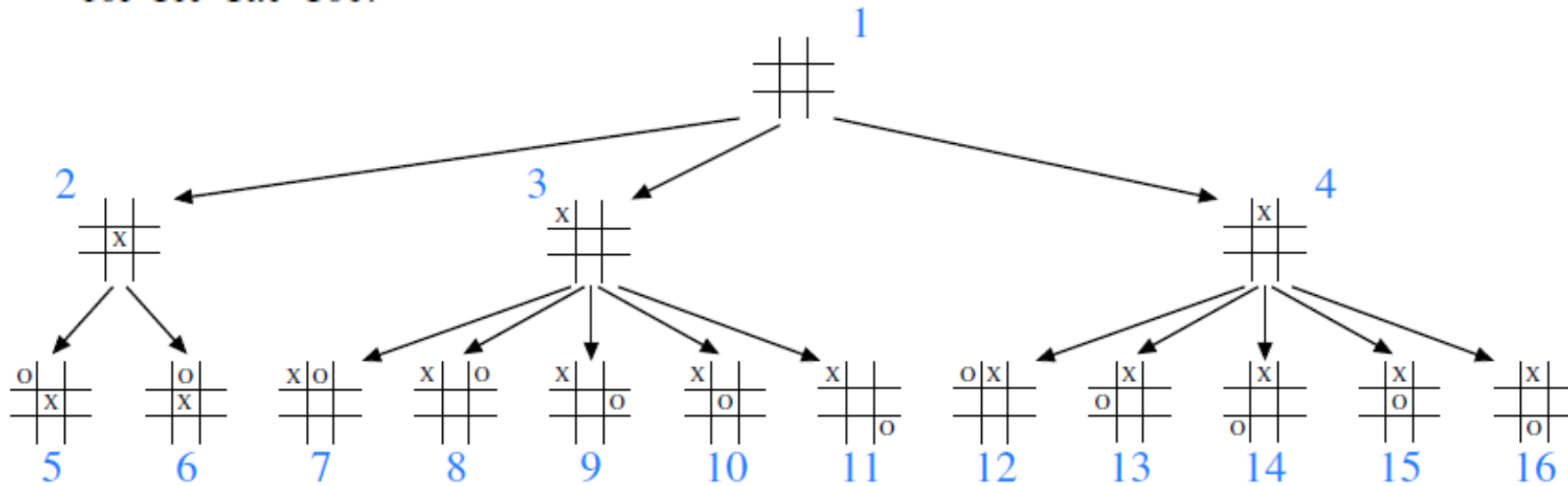


Figure 8.17: A binary search tree storing integers. The solid path is traversed when searching (successfully) for 42. The dashed path is traversed when searching (unsuccessfully) for 70.

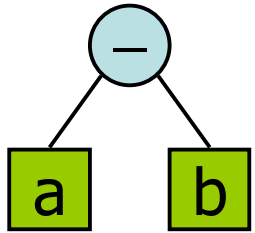
Traversing Binary Trees

(Breadth-First Tree Traversal)

visit all the positions at depth d before we visit the positions at depth $d+1$.



Arithmetic Expressions



Inorder: $a - b$

Postorder: $a b -$

Preorder: $- a b$

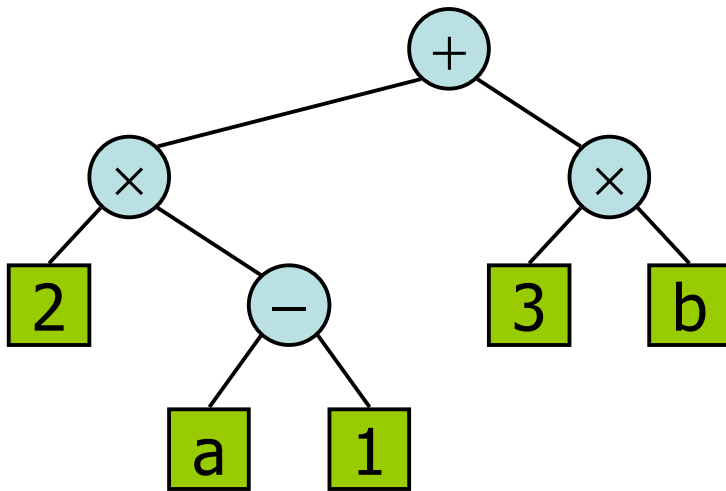
Inorder:

$2 \times a - 1 + 3 \times b$

$((2 \times (a - 1)) + (3 \times b))$

Postorder:

$2 a 1 - \times 3 b \times +$



Priority Queue & Heaps

Data structure

Fall 2023

Hashemite University

Dr. Mohammad Al-hammouri

Priority Queue ADT

- A priority queue stores a collection of entries.
- Each entry is a pair (key, value)
- Main methods of the Priority Queue ADT:
 - **insert(k, v)**: inserts an entry with key k and value v
 - **removeMin()** removes and returns the entry with smallest key, or null if the the priority queue is empty

Priority Queue ADT

Method	Return Value	Priority Queue Contents
insert(5,A)		{ (5,A) }
insert(9,C)		{ (5,A), (9,C) }
insert(3,B)		{ (3,B), (5,A), (9,C) }
min()	(3,B)	{ (3,B), (5,A), (9,C) }
removeMin()	(3,B)	{ (5,A), (9,C) }
insert(7,D)		{ (5,A), (7,D), (9,C) }
removeMin()	(5,A)	{ (7,D), (9,C) }
removeMin()	(7,D)	{ (9,C) }
removeMin()	(9,C)	{ }
removeMin()	null	{ }
isEmpty()	true	{ }

Unsorted List Implementation

Please check `UnsortedListPriorityQueue.java`

Method	Running Time
size	$O(1)$
isEmpty	$O(1)$
insert	$O(1)$
min	$O(n)$
removeMin	$O(n)$

Sorted List Implementation

Method	Unsorted List	Sorted List
size	$O(1)$	$O(1)$
isEmpty	$O(1)$	$O(1)$
insert	$O(1)$	$O(n)$
min	$O(n)$	$O(1)$
removeMin	$O(n)$	$O(1)$

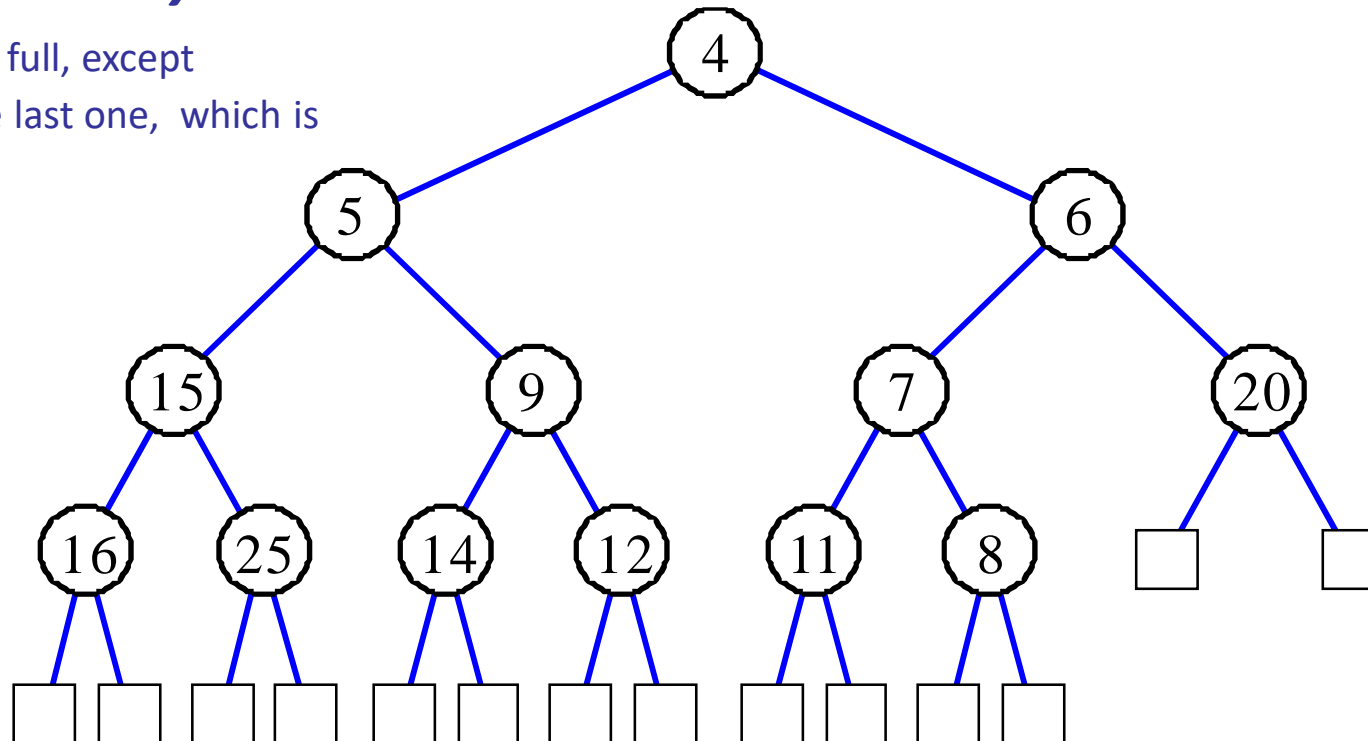
Heaps

Complete binary tree that stores a collection of keys (or key-element pairs)
at its internal nodes and that satisfies
the additional property: **key(parent) \leq key(child)**

REMEMBER:

complete binary tree

all levels are full, except
(possibly) the last one, which is
left-filled



Heaps

Complete Binary Tree Property: A heap T with height h is a *complete* binary tree if levels $0, 1, 2, \dots, h-1$ of T have the maximal number of nodes possible (namely, level i has 2^i nodes, for $0 \leq i \leq h-1$) and the remaining nodes at level h reside in the leftmost possible positions at that level.

Proposition 9.2: A heap T storing n entries has height $h = \lfloor \log n \rfloor$.

Heaps

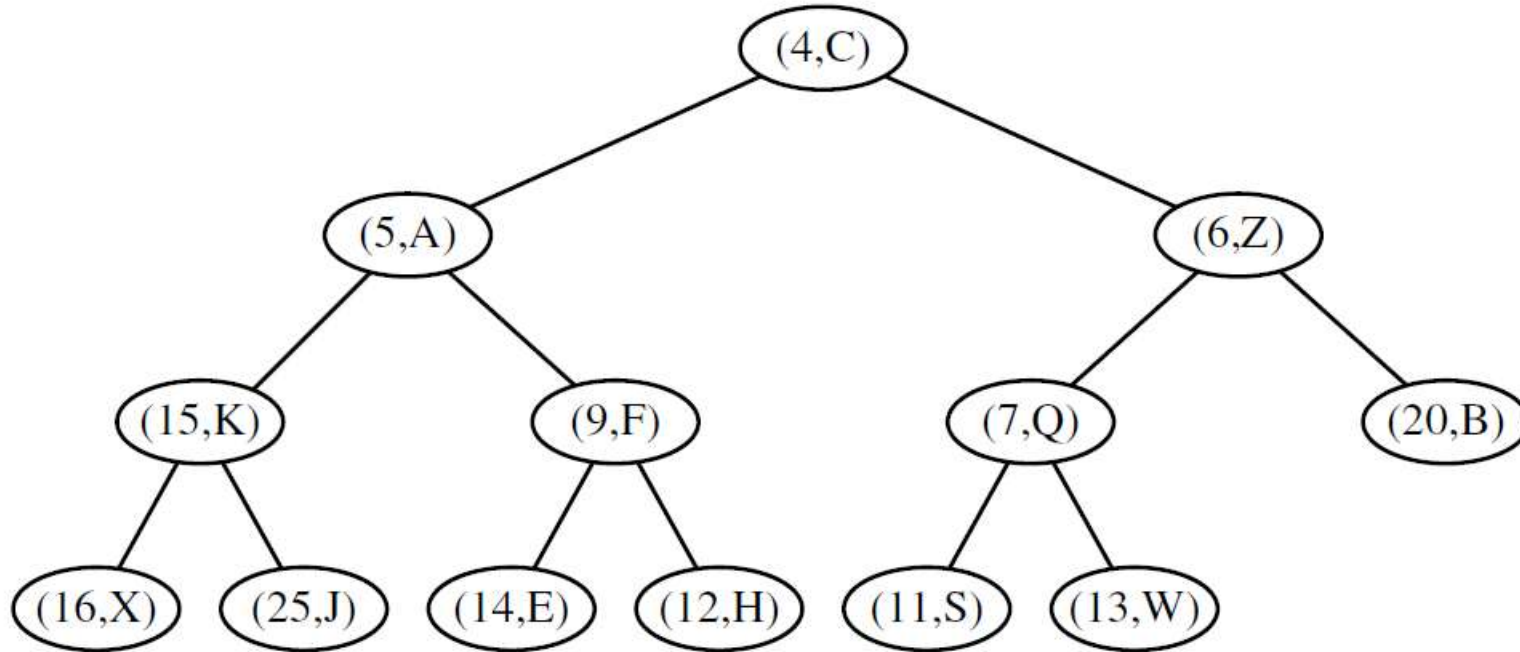


Figure 9.1: Example of a heap storing 13 entries with integer keys. The last position is the one storing entry $(13, W)$.

Heap to implement a priority queue

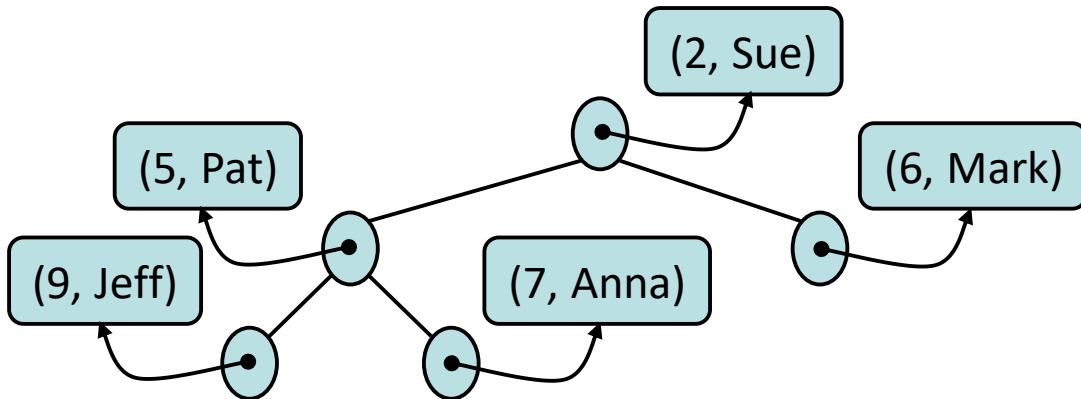
- We could use a heap to implement a priority queue
- We store a (key, element) item at each node

Insert.

removeMin().

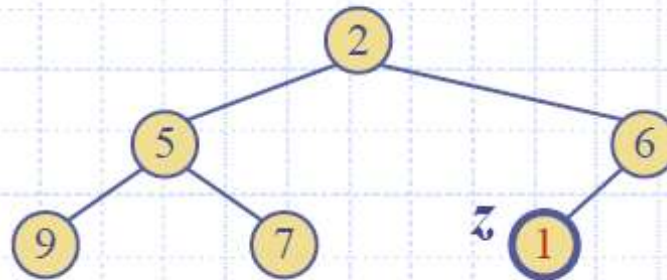
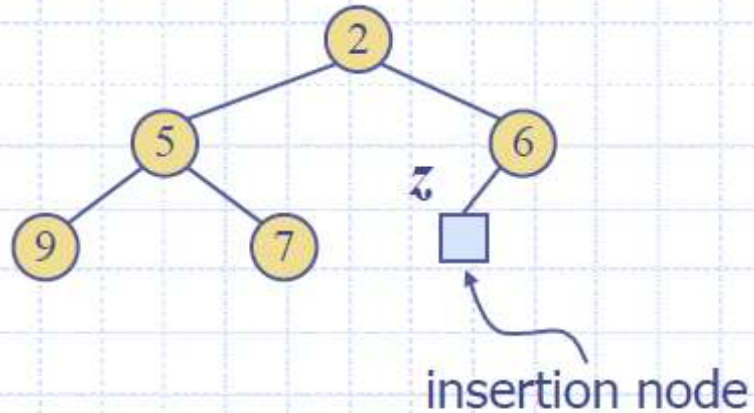
→ Remove the root

→ Re-arrange the heap!



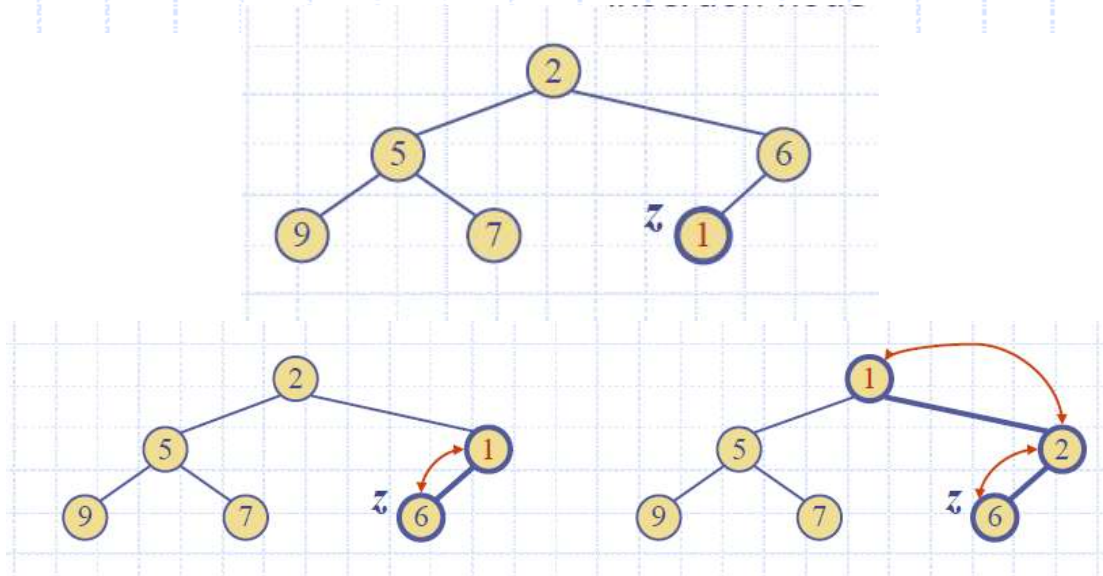
Heap Insertion

- ❑ Method `insertItem` of the priority queue ADT corresponds to the insertion of a key k to the heap
- ❑ The insertion algorithm consists of three steps
 - Find the insertion node z (the new last node)
 - Store k at z
 - Restore the heap-order property (discussed next)

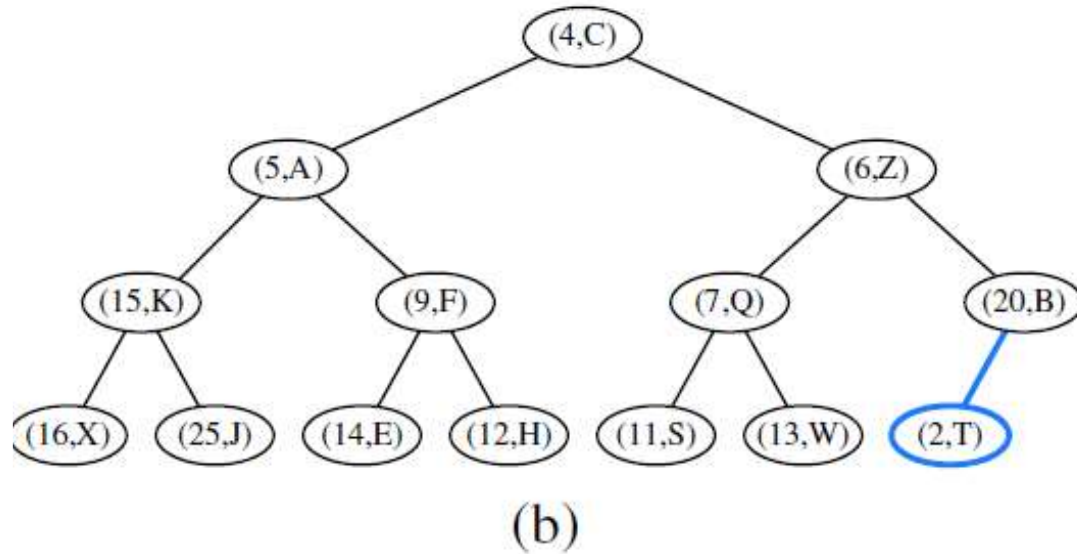
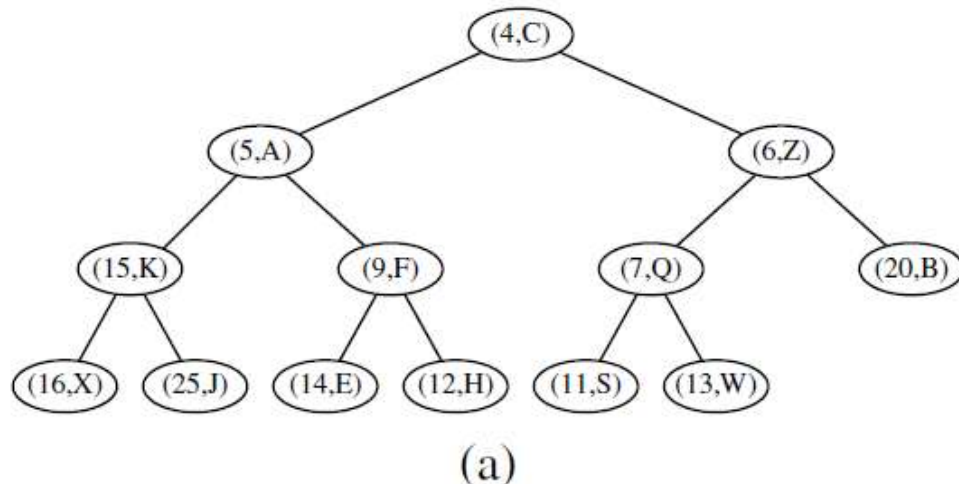


Up-Heap

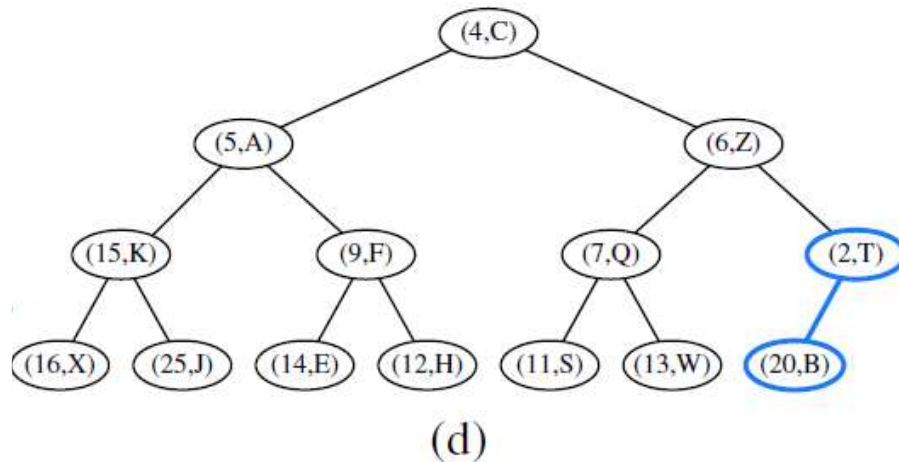
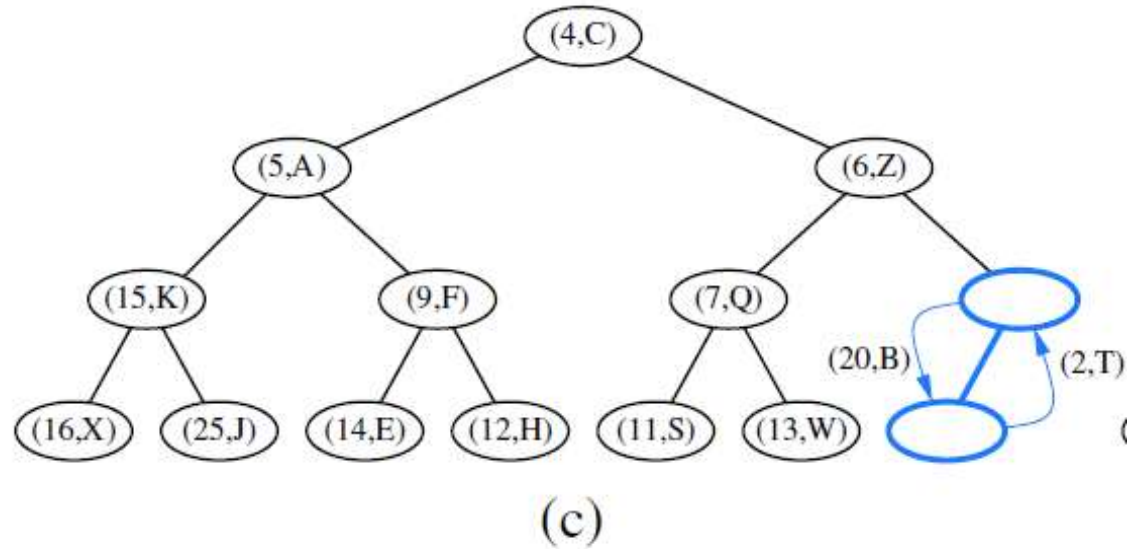
- ❑ After the insertion of a new key k , the heap-order property may be violated
- ❑ Algorithm upheap restores the heap-order property by swapping k along an upward path from the insertion node
- ❑ Upheap terminates when the key k reaches the root or a node whose parent has a key smaller than or equal to k
- ❑ Since a heap has height $O(\log n)$, upheap runs in $O(\log n)$ time



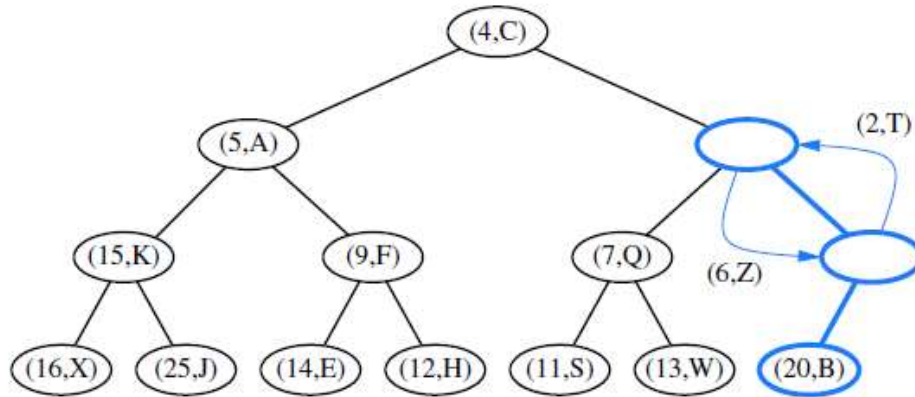
Heap Insertion 1 (up-heap bubbling)



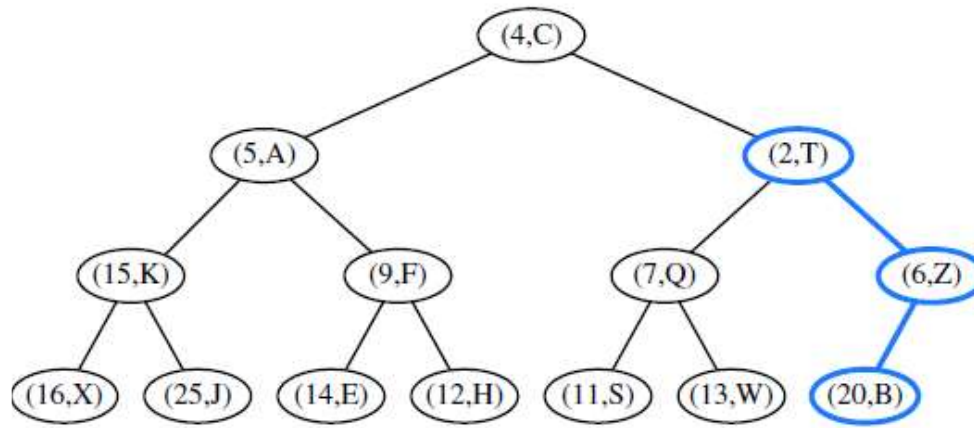
Heap Insertion 2 (up-heap bubbling)



Heap Insertion 2 (up-heap bubbling)

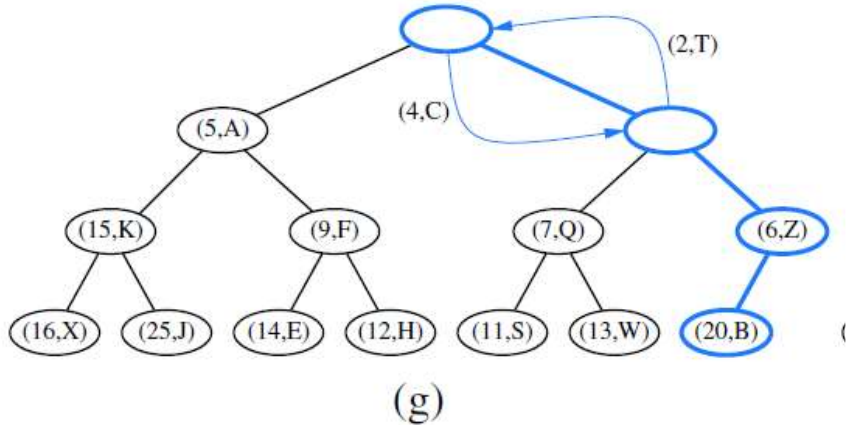


(e)

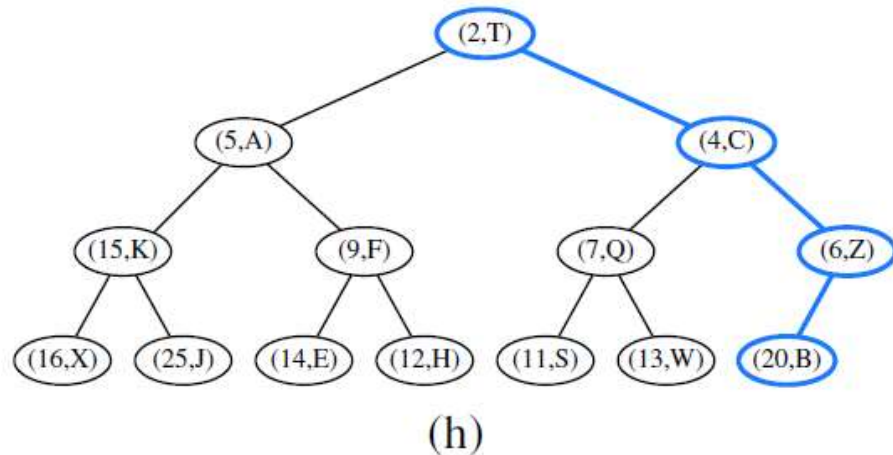


(f)

Heap Insertion 2 (up-heap bubbling)



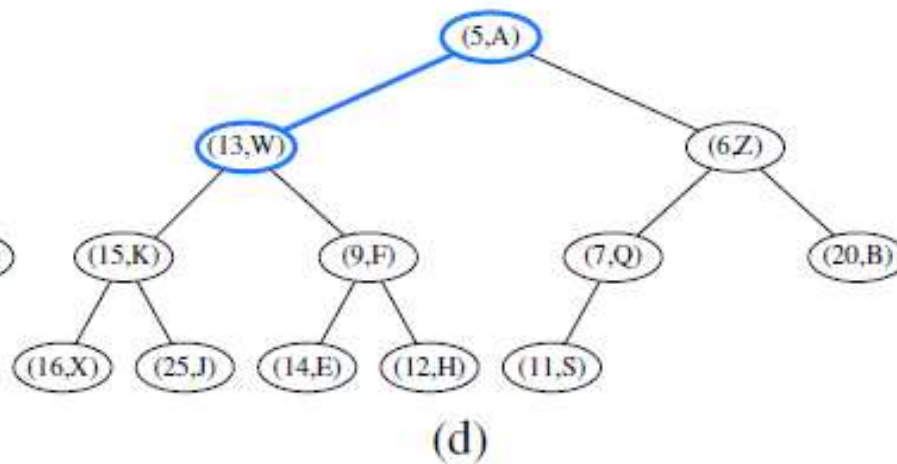
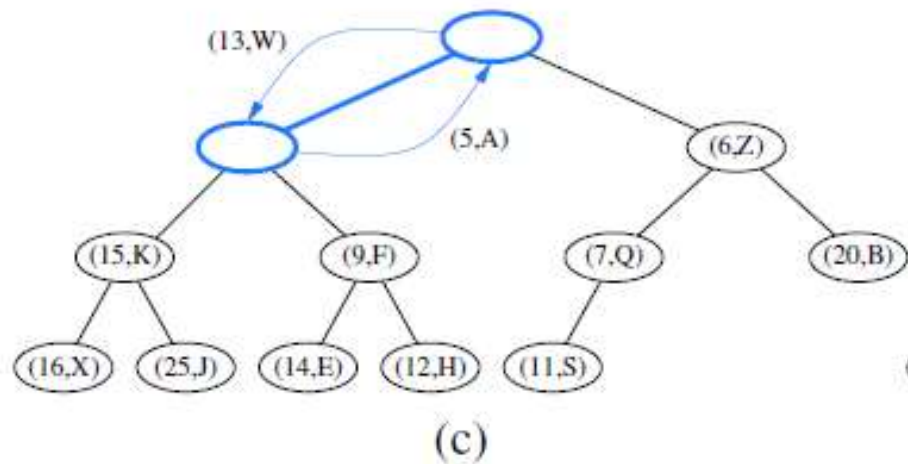
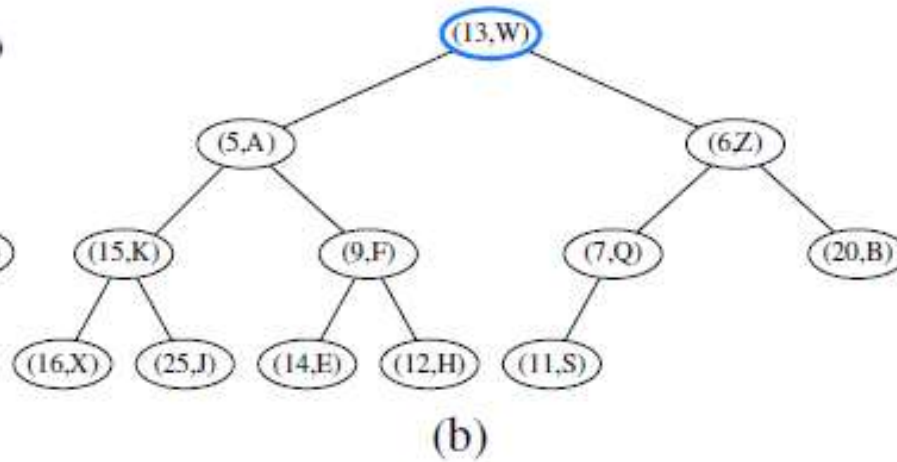
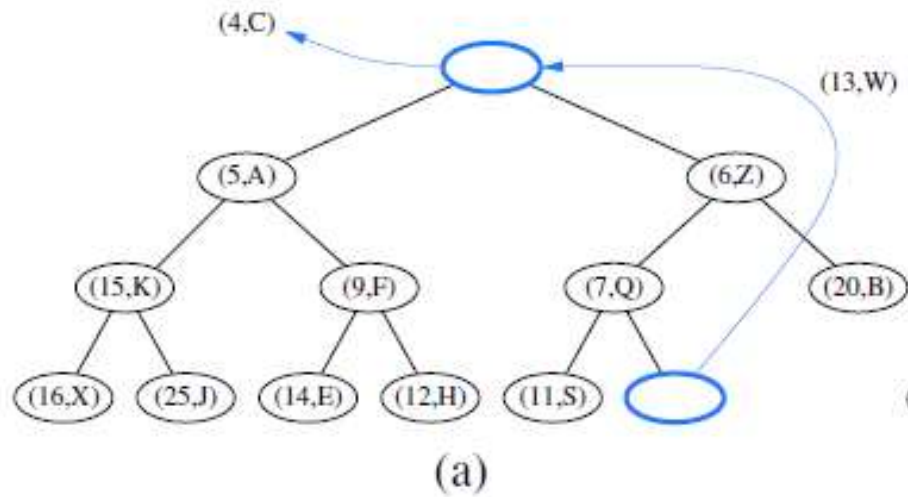
In the worst case, the number of swaps performed in insert is equal to the height of T : $\lfloor \log n \rfloor$



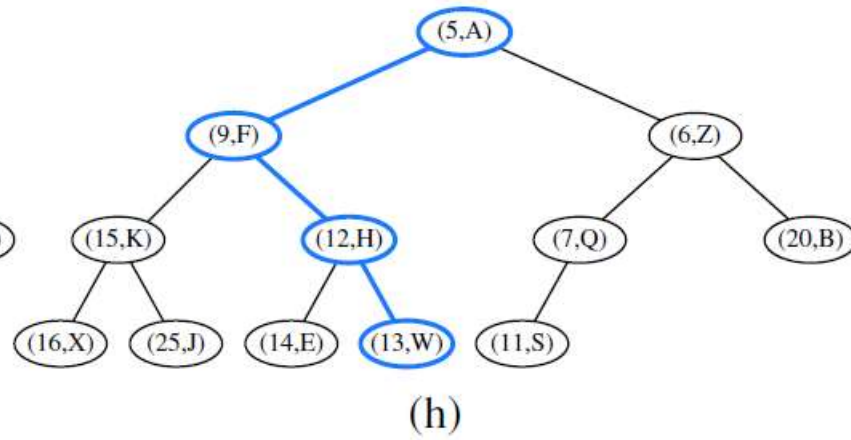
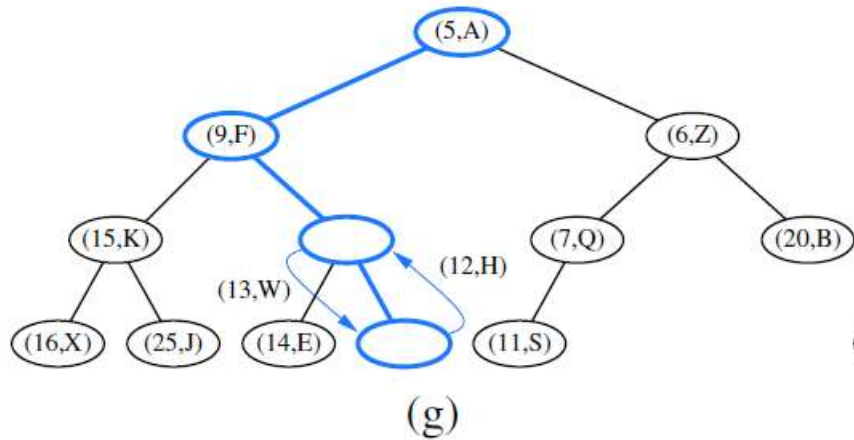
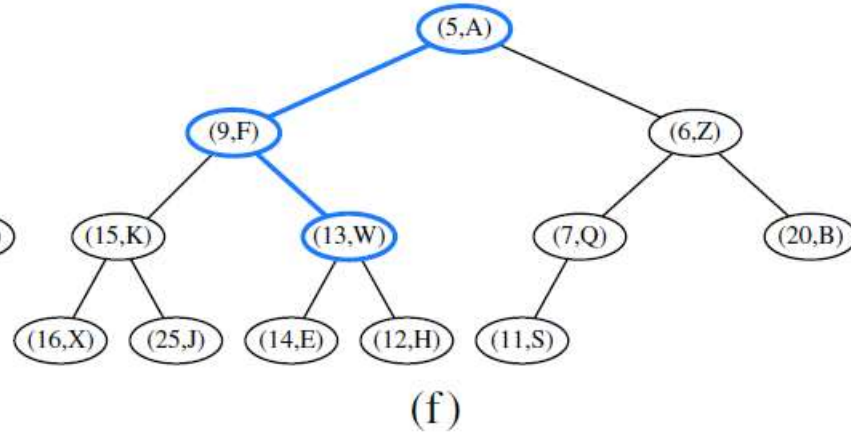
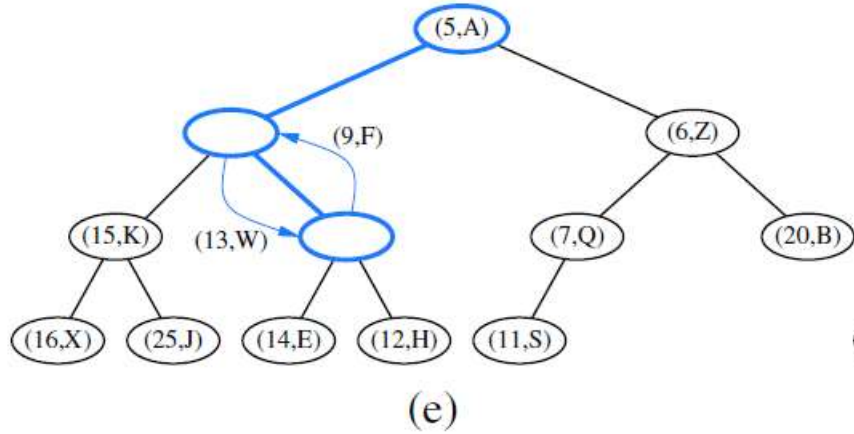
Heap removeMin (Down-heap bubbling)

- An entry with the smallest key is stored at the root r of T .
- We cannot simply delete node r , because this would leave two disconnected subtrees.
- The shape of the heap respects the complete binary tree property
- Deleting the leaf at the last position p of T , defined as the rightmost position at the bottommost level of the tree (Preserve complete binary tree property).
- Down-heap compares the parent with **the smallest child**.

Heap **removeMin** (Down-heap bubbling)

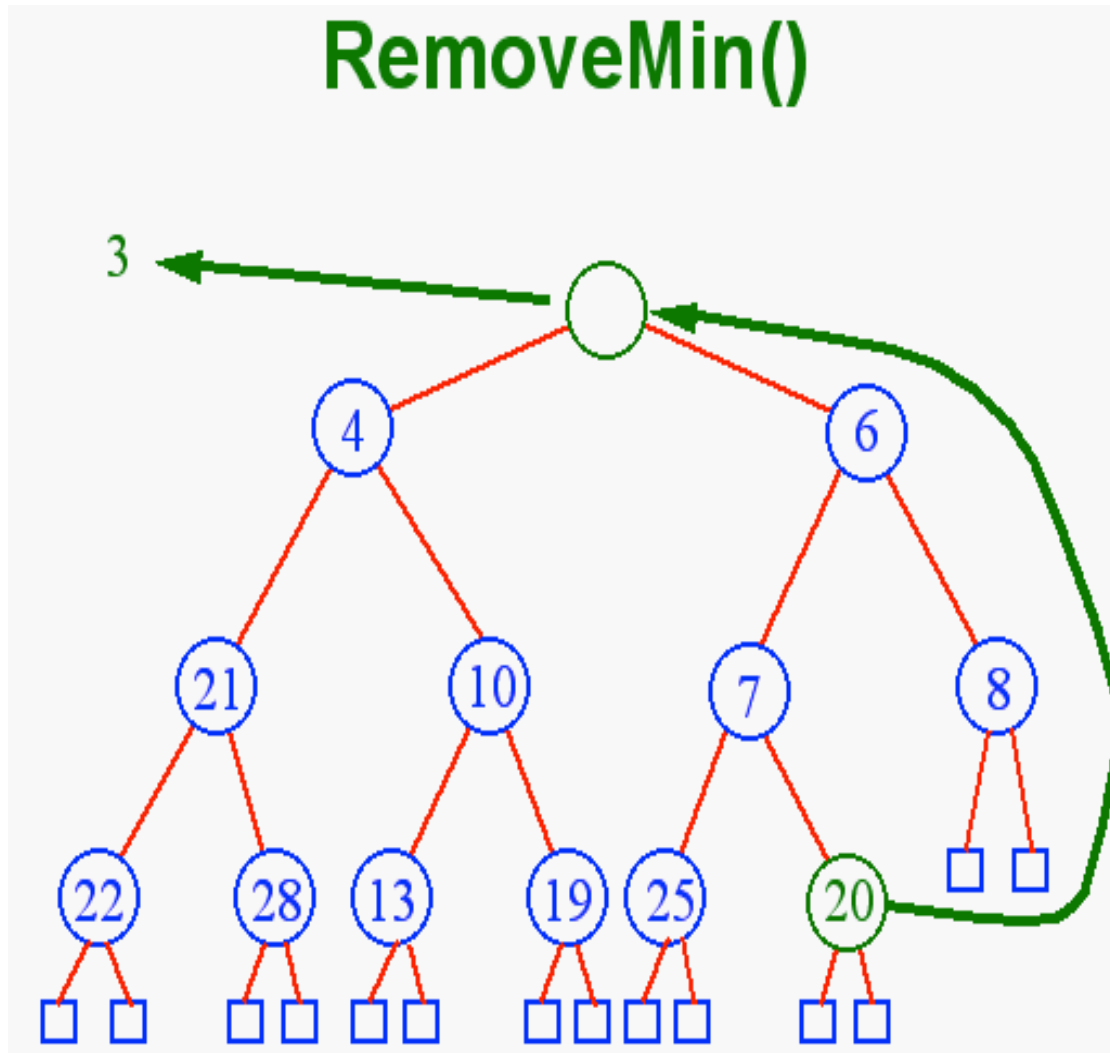


Heap removeMin (Down-heap bubbling)

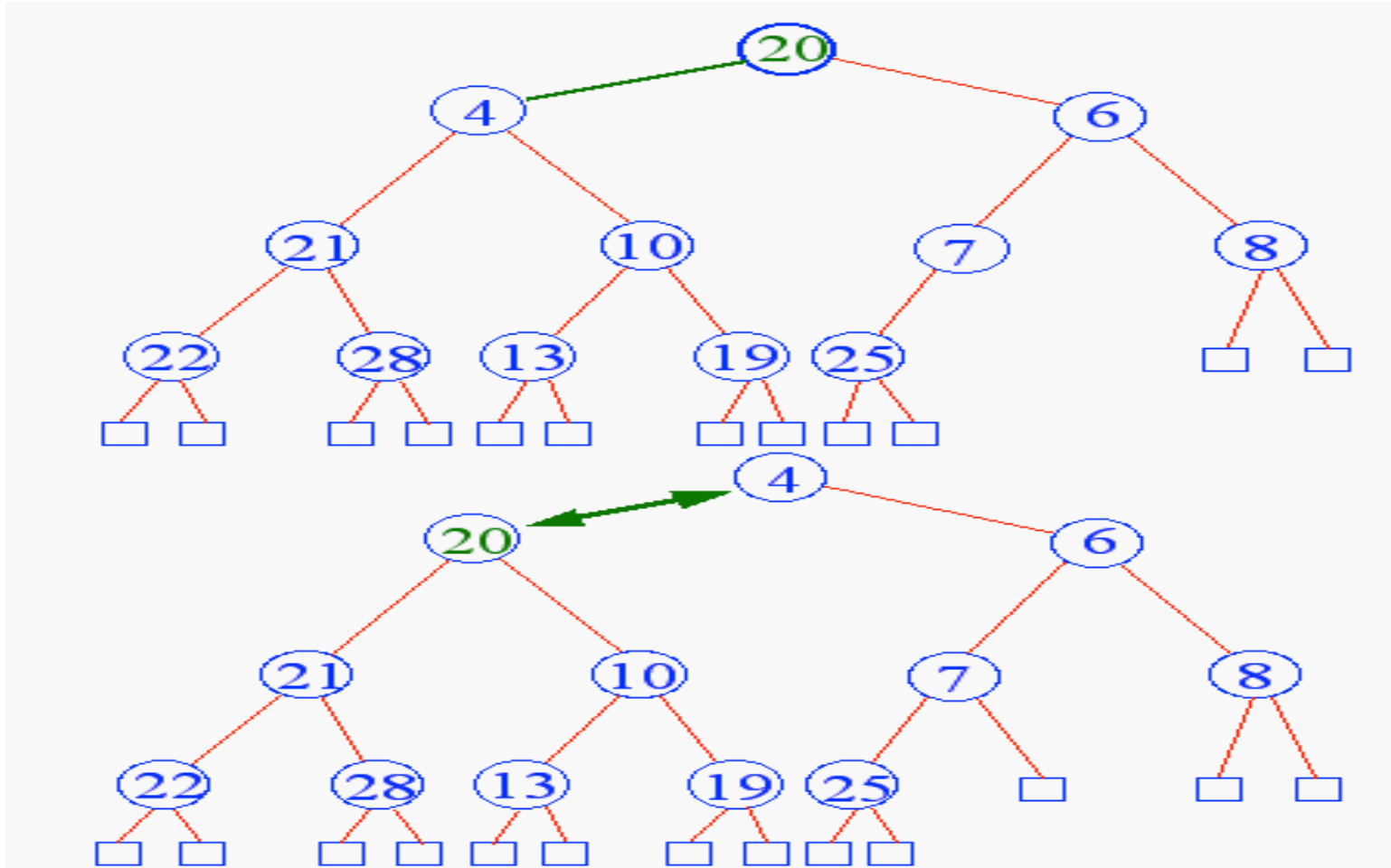


in the worst case, equal to the height of heap T , that is, it is $\lfloor \log n \rfloor$

Removal From a Heap

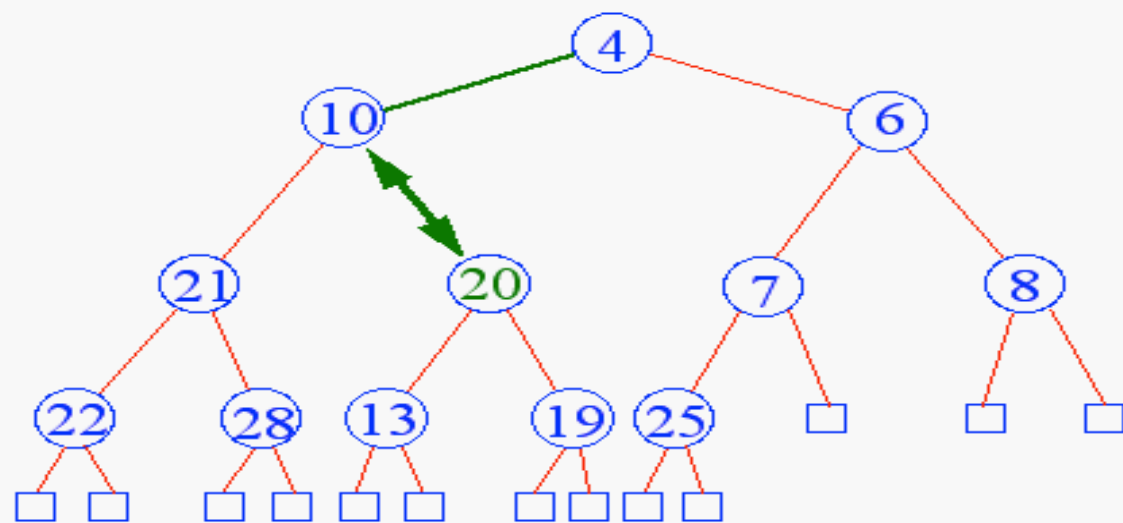
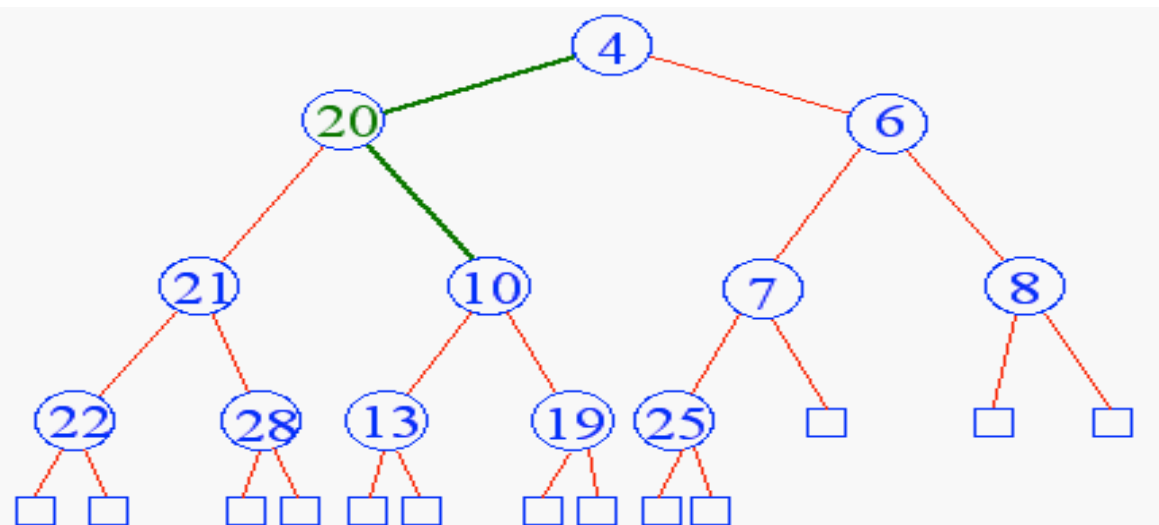


Downheap

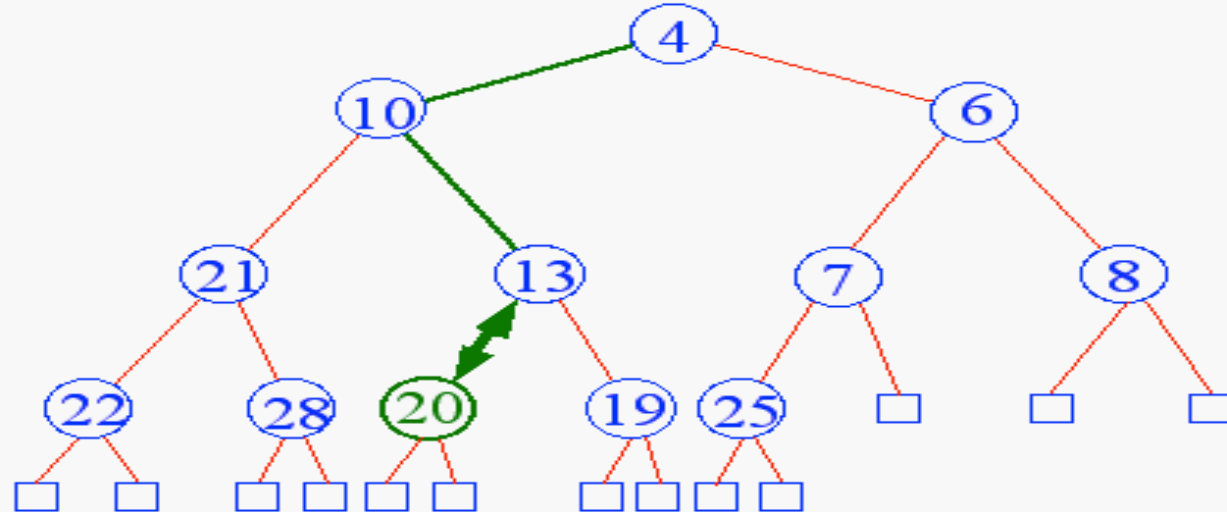
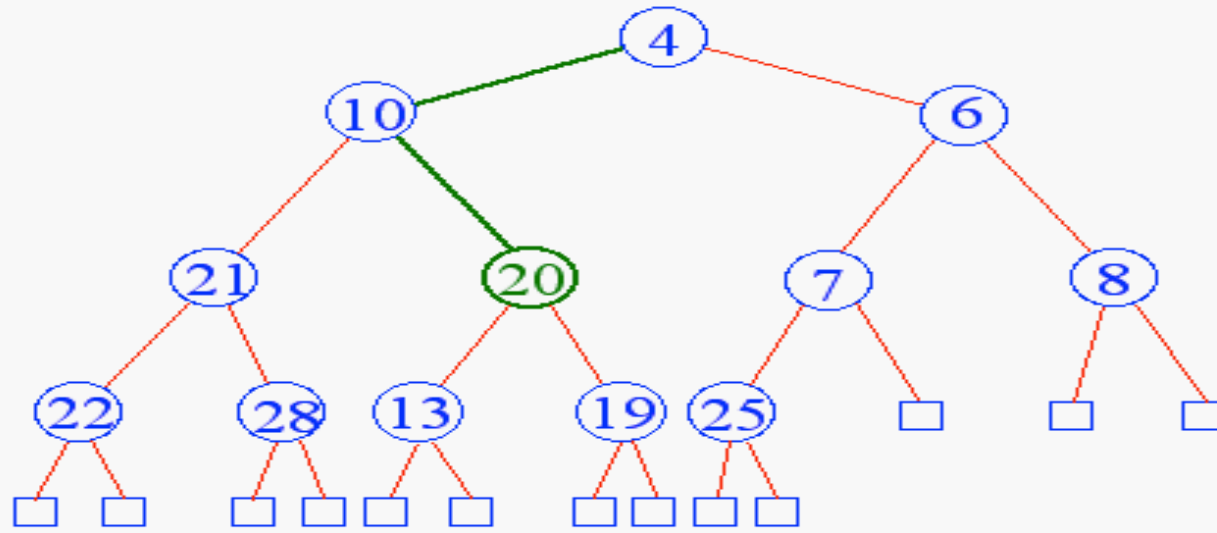


- Down-heap compares the parent with the smallest child. If the child is smaller, it switches the two.

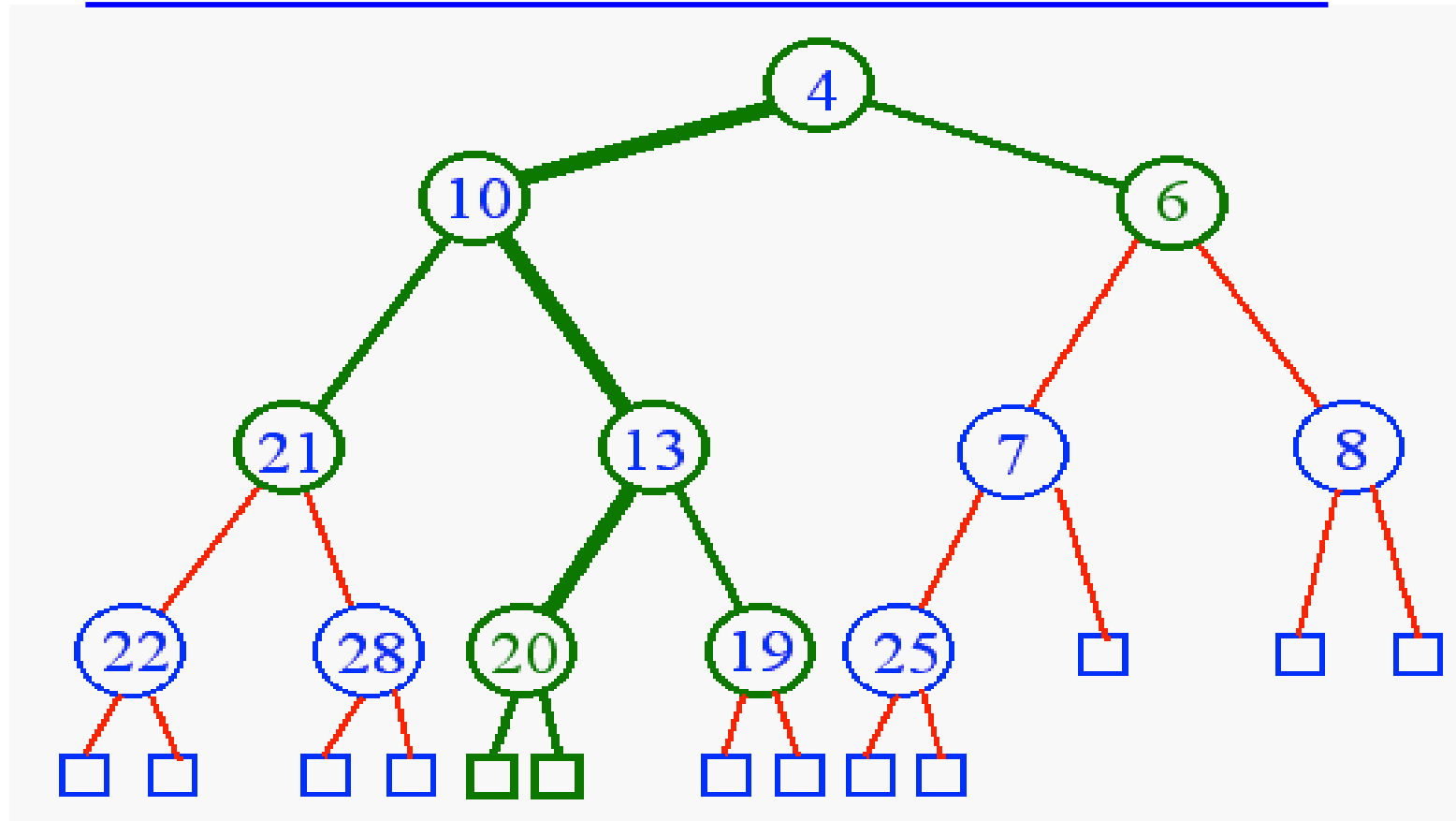
Downheap Continues



Downheap Continues



End of Down-heap

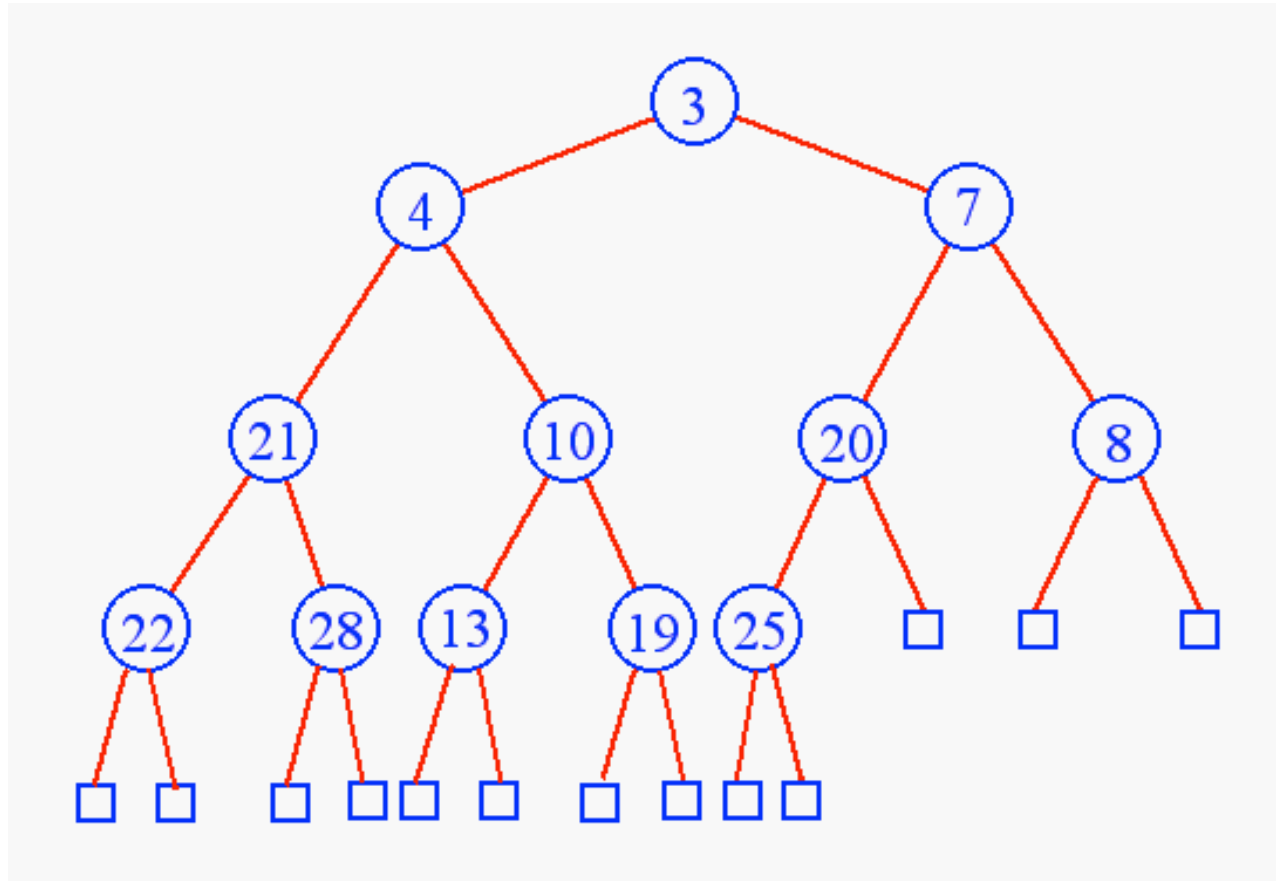


- Downheap terminates when the key is greater than the keys of both its children or the bottom of the heap is reached.

(total #swaps) $\leq (h - 1)$, which is **$O(\log n)$**

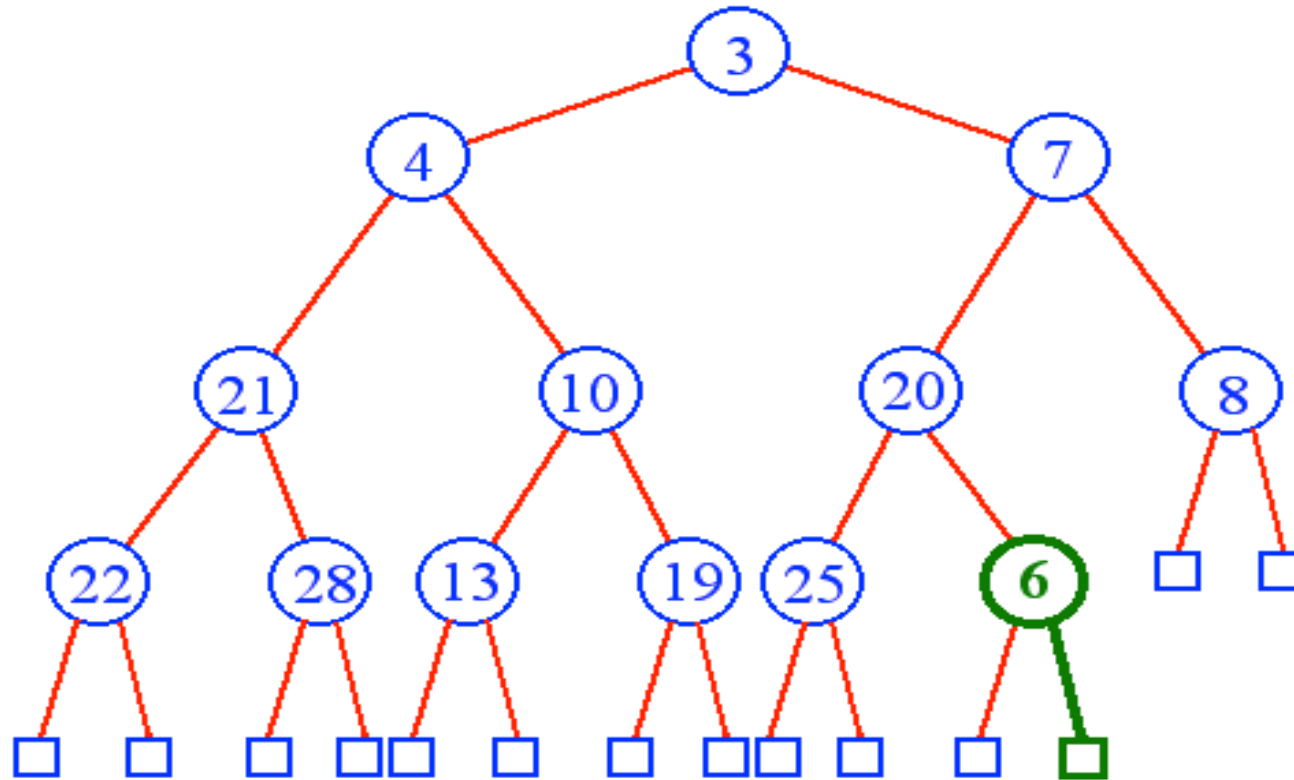
Heap Insertion

The key to insert is 6



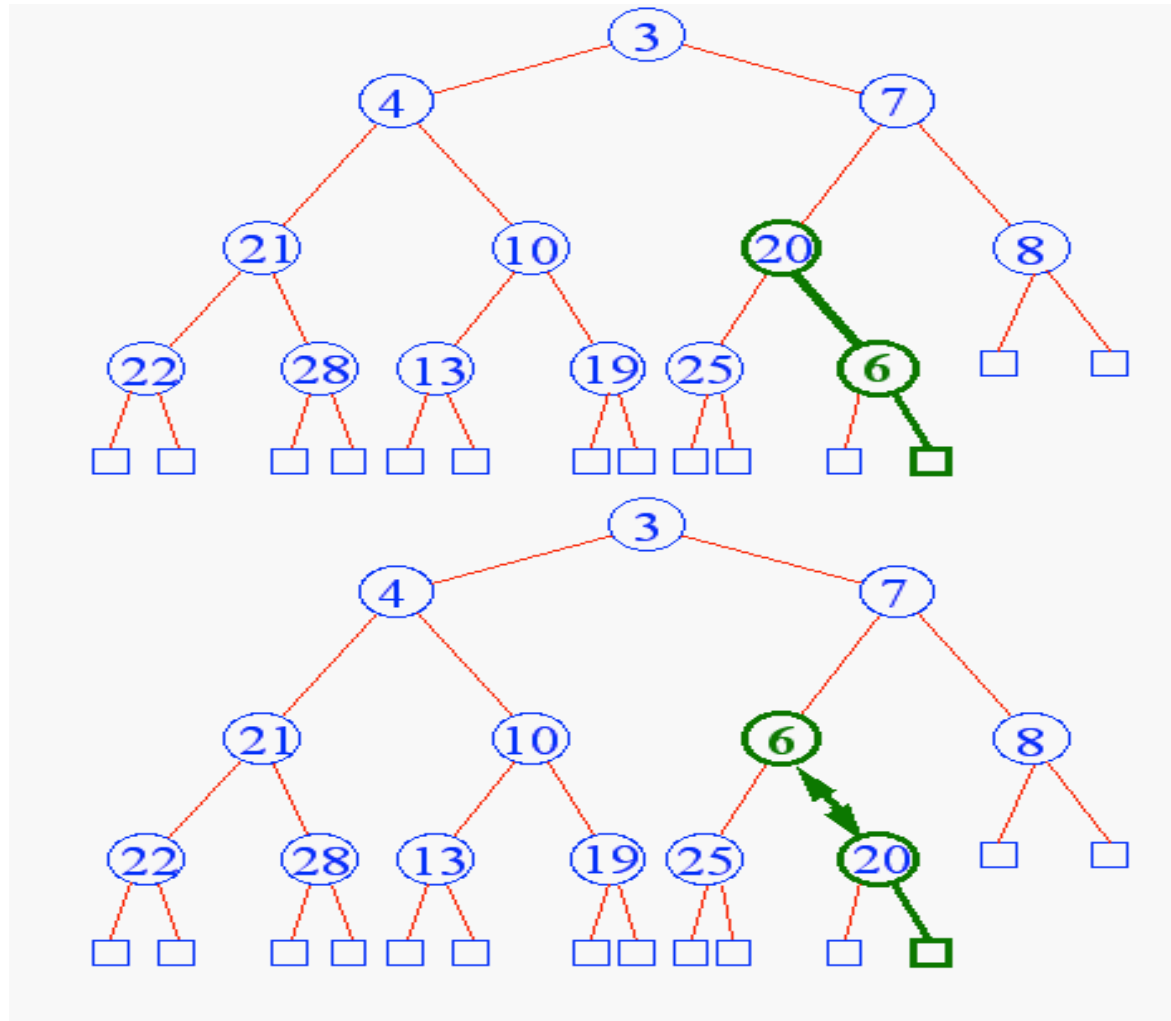
Heap Insertion

Add the key in the next available position

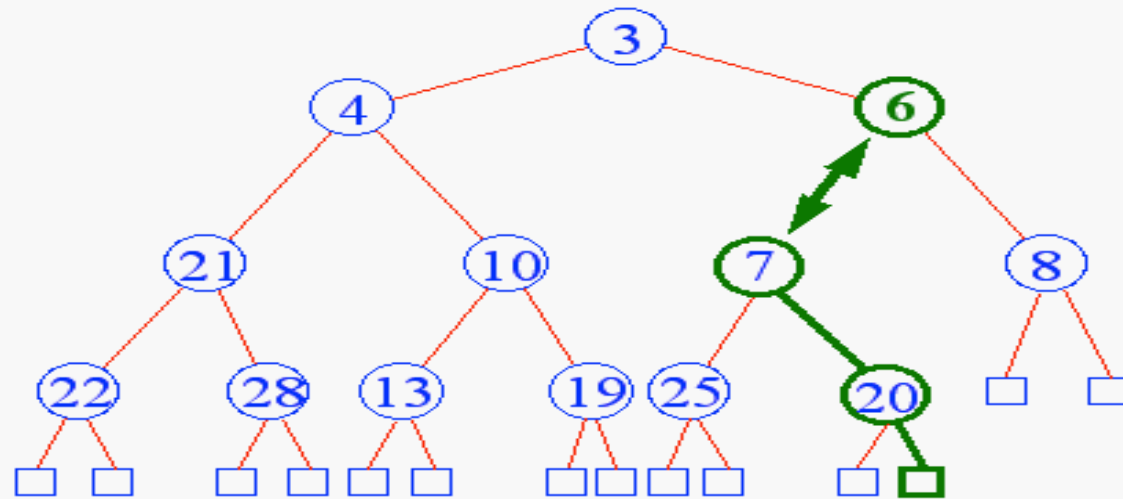
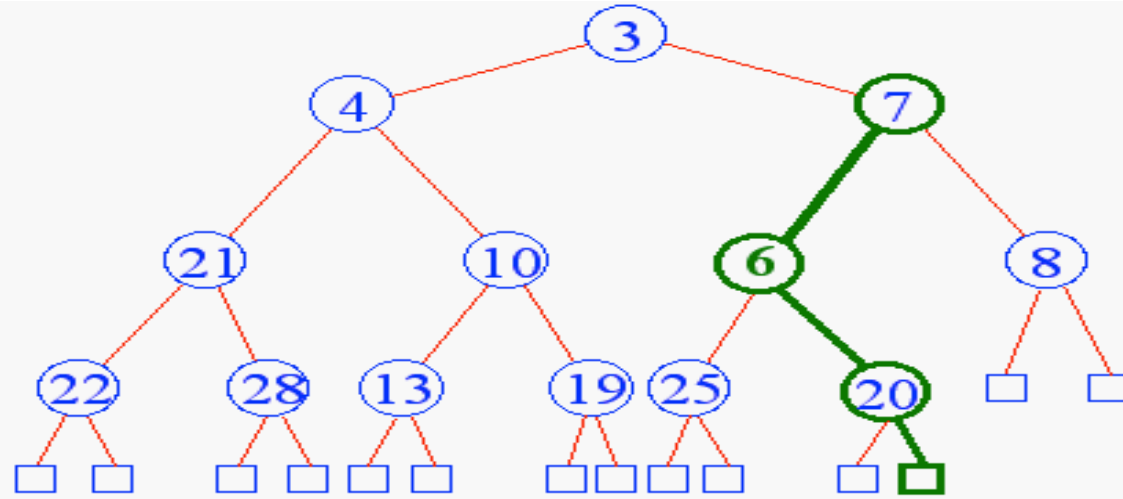


Upheap

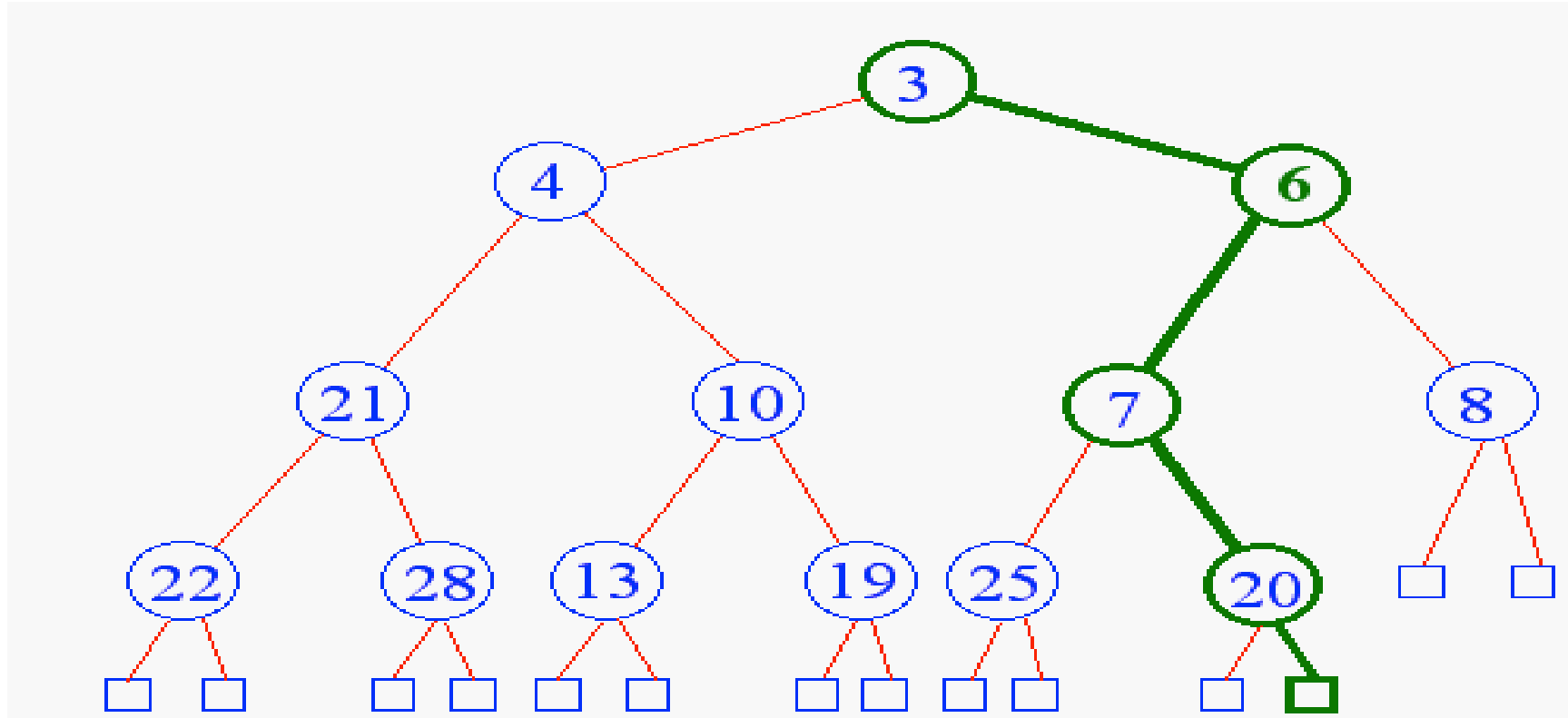
- Swap parent-child keys out of order



Upheap Continues



End of Upheap

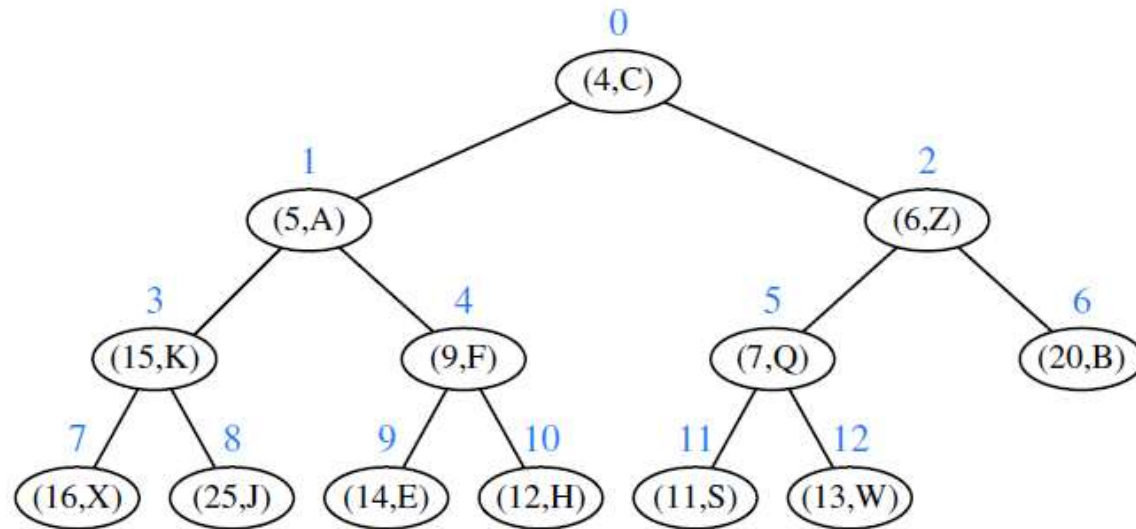


- Upheap terminates when new key is greater than the key of its parent or the top of the heap is reached

(total #swaps) $\leq (h - 1)$, which is $O(\log n)$

Array-Based Representation of a Complete Binary Tree

- If p is the root, then $f(p) = 0$.
- If p is the left child of position q , then $f(p) = 2f(q) + 1$.
- If p is the right child of position q , then $f(p) = 2f(q) + 2$.



(4,C)	(5,A)	(6,Z)	(15,K)	(9,F)	(7,Q)	(20,B)	(16,X)	(25,J)	(14,E)	(12,H)	(11,S)	(13,W)
0	1	2	3	4	5	6	7	8	9	10	11	12

Array-Based Representation of a Complete Binary Tree

- The array-based heap representation avoids some complexities of a linked tree structure.
- Methods insert and removeMin depend on locating the last position of a heap.
- With the array-based representation of a heap of size n , the last position is simply at index $n-1$.

Priority queue implemented with an array-based heap

```
/** Removes and returns an entry with minimal key (if any). */
public Entry<K,V> removeMin() {
    if (heap.isEmpty()) return null;
    Entry<K,V> answer = heap.get(0);
    swap(0, heap.size() - 1);           // put minimum item at the end
    heap.remove(heap.size() - 1);       // and remove it from the list;
    downheap(0);                         // then fix new root
    return answer;
}

protected void upheap(int j) {
    while (j > 0) {                      // continue until reaching root (or break statement)
        int p = parent(j);
        if (compare(heap.get(j), heap.get(p)) >= 0) break; // heap property verified
        swap(j, p);
        j = p;                           // continue from the parent's location
    }
}
```


Priority queue implemented with an array-based heap

```
30  /** Moves the entry at index j lower, if necessary, to restore the heap property. */
31  protected void downheap(int j) {
32      while (hasLeft(j)) {                // continue to bottom (or break statement)
33          int leftIndex = left(j);
34          int smallChildIndex = leftIndex; // although right may be smaller
35          if (hasRight(j)) {
36              int rightIndex = right(j);
37              if (compare(heap.get(leftIndex), heap.get(rightIndex)) > 0)
38                  smallChildIndex = rightIndex; // right child is smaller
39          }
40          if (compare(heap.get(smallChildIndex), heap.get(j)) >= 0)
41              break; // heap property has been restored
42          swap(j, smallChildIndex);
43          j = smallChildIndex; // continue at position of the child
44      }
45  }
```

Priority queue implemented with an array-based heap

```
/** Inserts a key-value pair and returns the entry created. */  
public Entry<K,V> insert(K key, V value) throws IllegalArgumentException {  
    checkKey(key);      // auxiliary key-checking method (could throw exception)  
    Entry<K,V> newest = new PQEntry<>(key, value);  
    heap.add(newest);    // add to the end of the list  
    upheap(heap.size() - 1); // upheap newly added entry  
    return newest;  
}
```

```
public Entry<K,V> min() {  
    if (heap.isEmpty()) return null;  
    return heap.get(0);  
}
```

Analysis of a Heap-Based Priority Queue

Method	Running Time
size, isEmpty	$O(1)$
min	$O(1)$
insert	$O(\log n)^*$
removeMin	$O(\log n)^*$

Locating the last position of a heap, as required for **insert** and **removeMin**, can be performed in $O(1)$ time for an array-based representation, or $O(\log n)$ time for a linked-tree representation

Map & Hash Tables

Data structure

Fall 2023

Hashemite University

Prepared by

Dr. Mohammad Al-hammouri

Lucia Mura (uottawa.ca)

Map ADT

A MAP is an ADT to efficiently store and retrieve values based on a uniquely identifying **search key**.

It stores key-value pairs (k,v) , which we call **entries**.

Keys are unique/no repeats (they uniquely identify the value); a key is mapped to a value.

The main operations of a MAP are **searching**, **inserting**, and **deleting** items.

Examples: student records, user accounts, etc

Typical keys are username, user ID, etc.

Map ADT

Maps are also known as **associative arrays**:

- The entry's key serves somewhat like an index into the map.
- However, unlike a standard array, a key of a map need not be numeric.
- does not directly designate a position within the structure.

Dictionary ADT is related, although it normally refers to a similar ADT that allows repeated keys.

The MAP ADT methods:

get(k): returns the value v associated to key k , if such entry exists; otherwise returns null.

put(k, v): if M does not have an entry with key k , then adds (k, v) and returns null; otherwise it replaces with v the value of the entry with key equal to k and returns the old value.

remove(k): removes from M the entry with key k and returns its value; if M has no such entry, then returns null.

size(): returns the number of entries in M .

isEmpty(): boolean indicating if M is empty.

The MAP ADT methods:

keySet(): Returns an iterable collection containing all the keys stored in M.

values(): Returns an iterable collection containing all the values of entries stored in M (with repetition if multiple keys map to the same value).

entrySet(): Returns an iterable collection containing all the key-value entries in M.

MAP ADT: examples

Applications/examples:

- **University information system:**

key= student id

value= student record (name, address, course grades)

- **A domain name system (DNS)** maps

a host name (key, e.g. www.wiley.com) to a

a IP address (value, e.g. 208.215.179.146)

- **A social media site** maps

a username which is the key (usually nonnumeric) to

the user info which is the value (typically tons of personal info)

MAP examples

<i>Method</i>	<i>Return Value</i>	<i>Map</i>
isEmpty()	true	{}
put(5,A)	null	{(5,A)}
put(7,B)	null	{(5,A), (7,B)}
put(2,C)	null	{(5,A), (7,B), (2,C)}
put(8,D)	null	{(5,A), (7,B), (2,C), (8,D)}
put(2,E)	C	{(5,A), (7,B), (2,E), (8,D)}
get(7)	B	{(5,A), (7,B), (2,E), (8,D)}
get(4)	null	{(5,A), (7,B), (2,E), (8,D)}
get(2)	E	{(5,A), (7,B), (2,E), (8,D)}
size()	4	{(5,A), (7,B), (2,E), (8,D)}
remove(5)	A	{(7,B), (2,E), (8,D)}
remove(2)	E	{(7,B), (8,D)}
get(2)	null	{(7,B), (8,D)}
remove(2)	null	{(7,B), (8,D)}
isEmpty()	false	{(7,B), (8,D)}
entrySet()	{(7,B), (8,D)}	{(7,B), (8,D)}
keySet()	{7, 8}	{(7,B), (8,D)}
values()	{B, D}	{(7,B), (8,D)}

Hash Tables

- A hash table, also known as a hash map, is a data structure that implements an associative array (Map) abstract data type that stores a collection of unique keys and their associated values.
- Each **key** is **unique** within the table.
- The key is hashed using a hash function to determine the index or **bucket** in which the corresponding value should be stored.
- **Hash Function:** A hash function is used to map keys to indices or buckets in the hash table. The goal is to distribute keys evenly across the array of buckets to minimize collisions.

Hash Tables

- Example: store the value associated with key k at index k of the table

0	1	2	3	4	5	6	7	8	9	10
	D		Z			C	Q			

Figure 10.3: A lookup table with length 11 for a map containing entries (1,D), (3,Z), (6,C), and (7,Q).

Hash Tables

- There may be two or more distinct keys that get mapped to the same index. As a result, we will conceptualize our table as a bucket array,

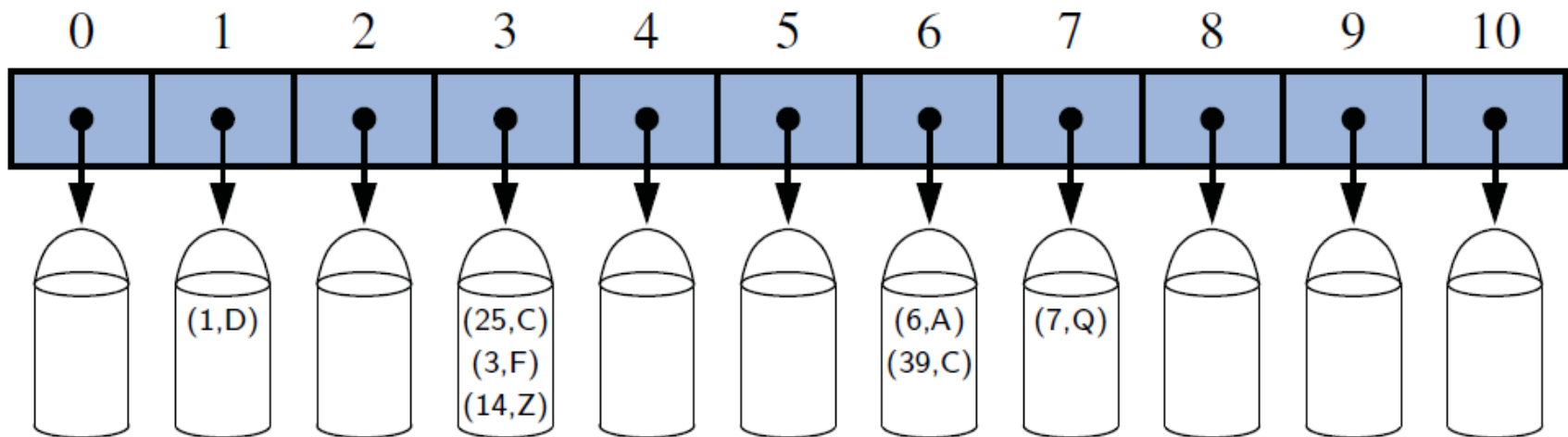
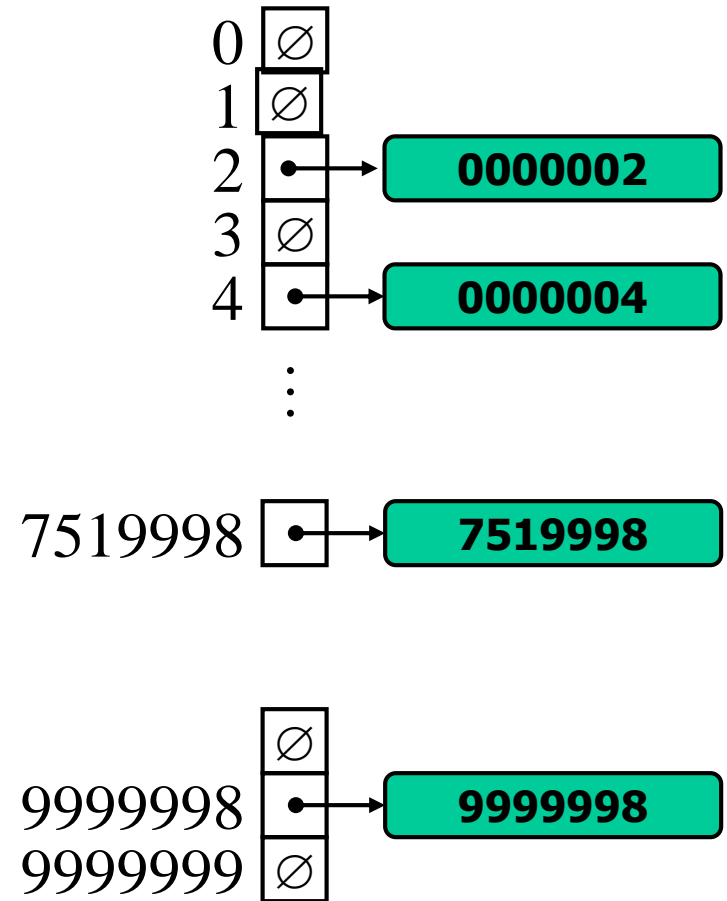


Figure 10.4: A bucket array of capacity 11 with entries (1,D), (25,C), (3,F), (14,Z), (6,A), (39,C), and (7,Q), using a simple hash function.

Example

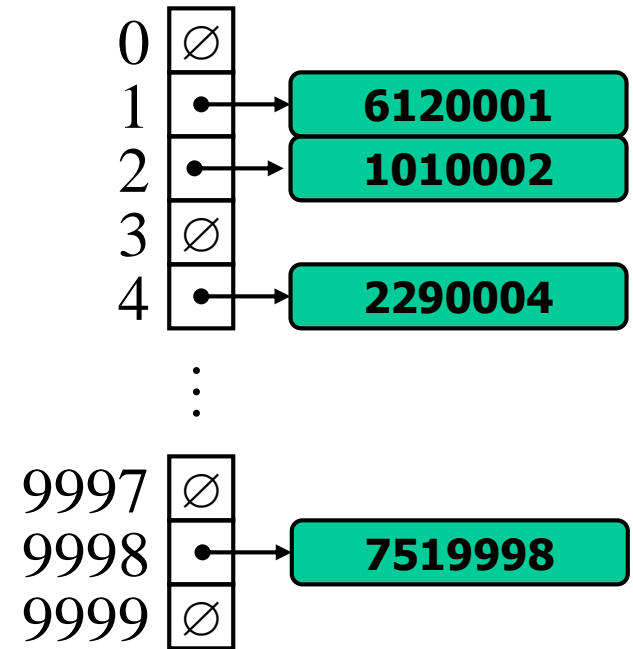
Student records are stored in an array using a 7 digit student i.d. the index.

If the i.d. were used unmodified, the array would have to have enough room for 10,000,000 student records.



Example

Instead, student i.d.'s are “hashed” to produce an integer between, say 1 and 10,000 which indexes into an array.



Problem

Since a possible 10,000,000 numbers are being compressed into just 10,000 how can we guarantee that no 2 i.d.'s end up stored in the same place?

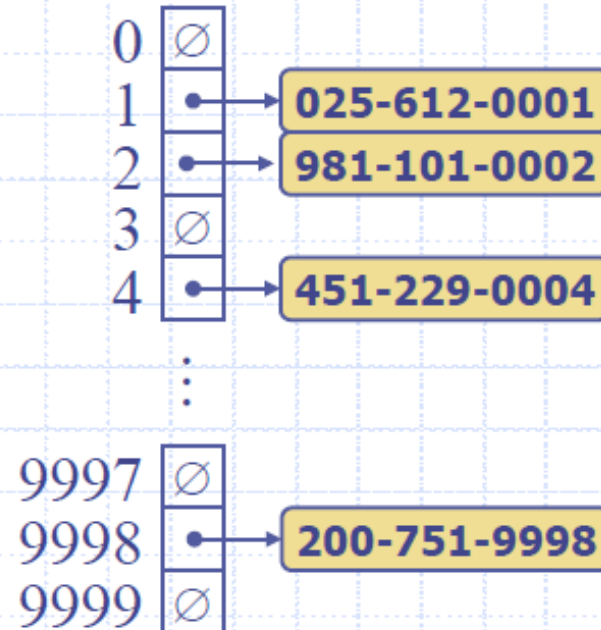
Hash Function

- A **hash function** h maps keys of a given type to integers in a fixed interval $[0, N - 1]$
- Example:
$$h(x) = x \bmod N$$

is a hash function for integer keys
- The integer $h(x)$ is called the **hash value** of key x
- A **hash table** for a given key type consists of
 - Hash function h
 - Array (called table) of size N
- When implementing a map with a hash table, the goal is to store item (k, o) at index $i = h(k)$

Hash Function

- We design a hash table for a map storing entries as (SSN, Name), where SSN (social security number) is a nine-digit positive integer
- Our hash table uses an array of size $N = 10,000$ and the hash function $h(x) = \text{last four digits of } x$



Hash Function

- A hash function is usually specified as the composition of two functions:

Hash code:

$h_1: \text{keys} \rightarrow \text{integers}$

Compression function:

$h_2: \text{integers} \rightarrow [0, N-1]$

- The hash code is applied first, and the compression function is applied next on the result, i.e.,

$$h(x) = h_2(h_1(x))$$

- The goal of the hash function is to “disperse” the keys in an apparently random way

—Example —

The keys all have different first letters.

CAT, ELEPHANT, FOX,
SKUNK, ZEBRA

$h(\text{CAT}) = 2$

$h(\text{ELEPHANT}) = 4$

$h(\text{FOX}) = 5$

⋮

	0
	1
CAT	2
	3
ELEPHANT	4
FOX	5
⋮	
SKUNK	
⋮	
ZEBRA	

Problem

If we want to insert a key that doesn't have a different first letter




COLLISIONS

	0
	1
CAT	2
	3
ELEPHANT	4
FOX	5
	6
	7
	8
⋮	⋮
⋮	⋮
SKUNK	
⋮	
⋮	
ZEBRA	

Problem

If we want to insert a key that doesn't have a different first letter


COLLISIONS

We want to insert:
CRICKET

$h(\text{CRICKET}) = 2$

index 2 is occupied

	0
	1
CAT	2
	3
ELEPHANT	4
FOX	5
	6
	7
	8
⋮	⋮
⋮	⋮
SKUNK	
⋮	
⋮	
ZEBRA	

Problem A

Address Generation

Construction of the function $h(K_i)$

- **Simple to calculate**
- **Uniformly distribute the elements in the table**

Problem B

Collision Resolution

What strategy to use if two keys map to same location $h(K_i)$

Definition:

load factor of an Hash Table

$$\alpha = \frac{n}{N} \begin{array}{l} \longrightarrow \\ \longrightarrow \end{array} \begin{array}{l} \# \text{ of elements} \\ \# \text{ of cells} \end{array}$$

Address Generation

Split problem into 2 sub-problems:

Hash code map:

h_1 : keys \rightarrow integers

$$h(x) = h_2(h_1(x))$$

Compression map:

h_2 : integers $\rightarrow [0, \text{TableSize} - 1]$

Hash Code Maps

Hash codes reinterpret the key as an integer. They need to:

1. Give the same result for the same key

and should:

2. Provide good “spread”

Examples:

- **Memory address:**

In Java, the default implementation of the hashCode() method for objects is based on the memory address of the object. For **numeric and string keys**, the default hashCode() behavior is based on the content of the object rather than the memory address

- **Integer cast:**

- We reinterpret the bits of the key as an integer

Compression Maps

Compression Maps:

- Take the output of the hash code and compress it into the desired range.
- If the result of the hash code was the same, the result of the compression map should be the same.
- Compression maps should maximize “spread” so as to minimize collisions.

Compression Maps

Compression Maps:

- Take the output of the hash code and compress it into the desired range.
- If the result of the hash code was the same, the result of the compression map should be the same.
- Compression maps should maximize “spread” so as to minimize collisions.

Compression Maps Examples

- Division:

- $h_2(\mathbf{y}) = \mathbf{y} \bmod \mathbf{N}$
- The size \mathbf{N} of the hash table is usually chosen to be a prime (number theory).

- Multiply, Add and Divide (MAD):

- $h_2(\mathbf{y}) = (\mathbf{a}\mathbf{y} + \mathbf{b}) \bmod \mathbf{N}$
- \mathbf{a} and \mathbf{b} are nonnegative integers such that $\mathbf{a} \bmod \mathbf{N} \neq 0$
- Otherwise, every integer would map to the same value \mathbf{b}

Address Generation

Some examples ...

Address Generation (a)

N = size of the table

$$r = \lceil \log N \rceil$$

Example: $N = 2^9$

$r=9$: number of bits to represent a cell

For a given key, the **Hash code** must return a sequence of bit

The **Compression Map** must return 9 bits representing a cell

000000000
000000001
000000011

.....

Address Generation (a)

Hash code

$h_1(x)$: gives a binary string

N = size of the table

$$r = \lceil \log N \rceil$$

Compression Map

$h_2(h_1(x))$: = subset (of r bits) of $h_1(x)$

a.1) the r least significant bits

a.2) the r most significant bits

a.3) the central r bits

→ Simple to calculate

→ Doesn't guarantee a random distribution

—Example —

Keys are words (animals)

Each word: 6 characters (just for this example)

h_1 transforms the key into a string of bits

CHAT--

000011 001000 000001 010010 100000 100000
└──┬──┬──┬──┬──┬──┘
C H A T - -

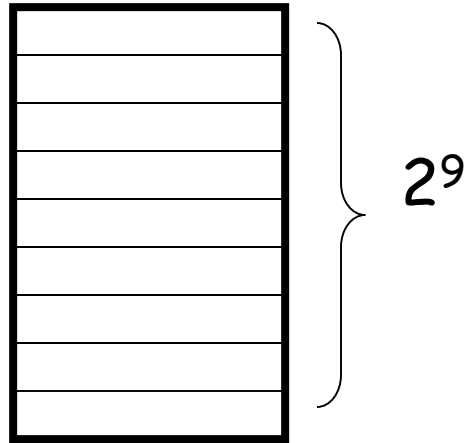
BAT---

000010 000001 010010 100000 100000

Coding of letters

A	000001
B	000010
C	000011
⋮	
H	001000
⋮	
T	010010
⋮	
-	100000

Size of the table: $N = 2^9$



$$N = 2^9$$
$$r = 9$$

CHAT--

$$h_2(\underbrace{000001}_{C}\underbrace{1001}_{H}\underbrace{00000000}_{A}\underbrace{101001}_{T}\underbrace{01000000}_{-}\underbrace{100000}_{-}) =$$

a1) Example of address Generation:
the r least significant bits

$$(r = 9)$$

$$h_2(000011001000000001010010100000100000) =$$

000100000

—

All the animals of 4 (or less) characters
hash to the same location.

a2) Example of address Generation:
the r most significant bits

$$(r = 9)$$

$$h_2(000011001000000001010010100000100000) =$$

000011001

All the animals that begin with the same first two letters hash to the same location.

Address Generation (b)

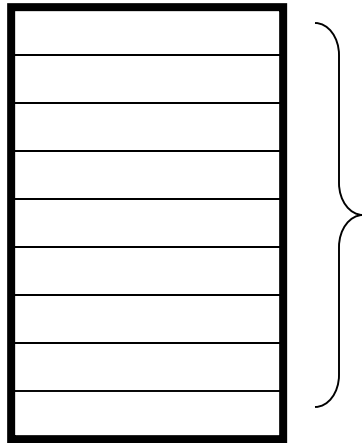
$h_1(x)$: gives a binary string

b) $h_2(h_1(x))$: sum of subset of bits of $h_1(x)$

→ Simple to calculate

→ More random than **a)**

—Example —



$$N = 2^9$$

$$r = 9$$

CHAT--

Coding of letters

A	000001
B	000010
C	000011
⋮	
H	001000
⋮	
T	010010
⋮	
⌊	100000

$$h_2(\underline{000011001}000000\underline{000101001}0100\underline{000100000}) =$$

000011001 most significant

000101001 central

000100000 least significant

XOR 000010000

Address Generation (c)

$h_1(x)$: gives a binary string

c) $h_2(h_1(x))$: subset (of r bits) of $h_1(x)^2$

→ Multiplication is involved

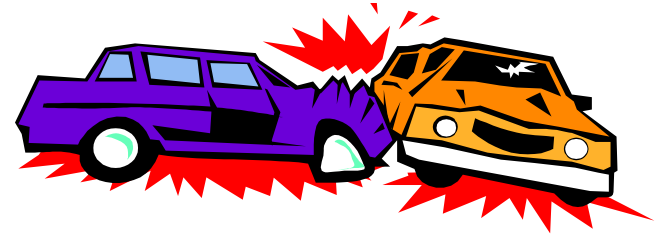
→ More random than a) and b)

Address Generation (d)

d) $h_2(h_1(x)) := h_1(x) \text{ MOD } N$

→ Division is involved!

→ Very random (if N is odd)



Collision Resolution

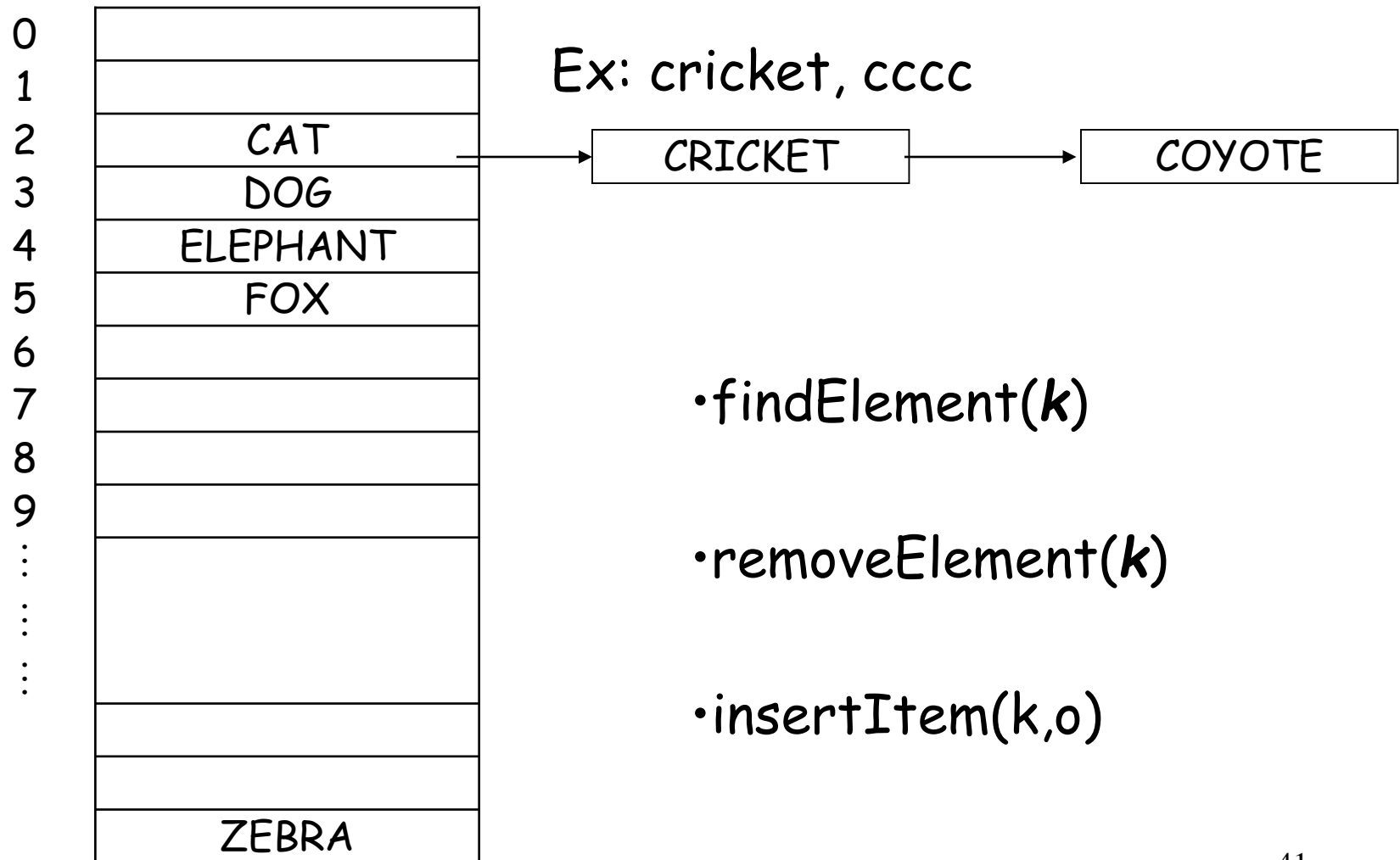
Collision Resolution

Separate Chaining

- Separate Chaining is a technique used for handling collisions in hash tables.
- When two keys hash to the same index (i.e., they produce the same hash code).
- Separate Chaining allows multiple key-value pairs to be stored at the same index in the form of a linked list or another data structure.

Collision Resolution

Separate Chaining



Complexity with separate chaining

Complexity of Separate chaining for collision resolution depends on various factors:

- Distribution of keys.
- Load factor.
- Efficiency of the hash function.

α is the load factor (n/m), and n is the number of elements in the table, m is the number of buckets (size of the array).

Complexity with separate chaining

- **Insertion (put):**

- Average Case: $O(1 + \alpha)$, where α is the load factor (n/m), and n is the number of elements in the table, m is the number of buckets (size of the array).
- In the average case, if the load factor is small, the chance of collisions is low, and insertions can be done in constant time.

- **Lookup (get):**

- Average Case: $O(1 + \alpha)$, similar to insertion.
- Again, in the average case, lookups have constant time complexity.
- In the average case, assuming a well-distributed set of keys, the length of the linked list at any index is expected to be small and roughly constant.

- **Deletion:**

- Similar to insertion and lookup, the average-case time complexity for deletion is $O(1 + \alpha)$.

Collision Resolution

(1) Linear Probing

1. Open Addressing

0	
1	
2	CAT
3	CRICKET
4	ELEPHANT
5	FOX
6	
7	
8	
9	
⋮	
⋮	
⋮	
	ZEBRA

→ COYOTE

- $h(\text{COYOTE}) = 2$ OCCUPIED
- We consider 3 OCCUPIED
- We consider 4 OCCUPIED
- “ 5 OCCUPIED
- “ 6 FREE!

Collision Resolution

Linear Probing

$h(K_i), h(K_i) + 1, h(K_i) + 2, h(K_i) + 3 \dots$

$h_0(K_i) \quad h_1(K_i) \quad h_2(K_i) \quad h_3(K_i)$

Let $h_0(K_i) = h(K_i)$

$$h_j(K_i) = [h(K_i) + j] \bmod N$$

This typically wraps around to the beginning of the array (hash table) if the end is reached

Search with Linear Probing

- Consider a hash table **A** that uses linear probing
findElement(*k*)
 - We start at cell ***h(k)***
 - We probe consecutive locations until one of the following occurs:
 - An item with key ***k*** is found, or
 - An empty cell is found, or
 - ***N*** cells have been unsuccessfully probed

Updates with Linear Probing

To handle insertions and deletions, we introduce a special object, called **AVAILABLE**, which replaces deleted elements:

- **removeElement(k)**
 - We search for an item with key k
 - If such an item (k, o) is found, we replace it with the special item **AVAILABLE** and we return element o
 - Else, we return **NO_SUCH_KEY**
- **insert Item(k, o)**
 - We throw an exception if the table is full
 - We start at cell $h(k)$
 - We probe consecutive cells until one of the following occurs
 - A cell i is found that is either empty or stores **AVAILABLE**, or
 - N cells have been unsuccessfully probed
 - We store item (k, o) in cell i

Performances of Linear Probing

Search: Average number of probes, $C(\alpha)$

Experimental results for a hash table
with load factor α

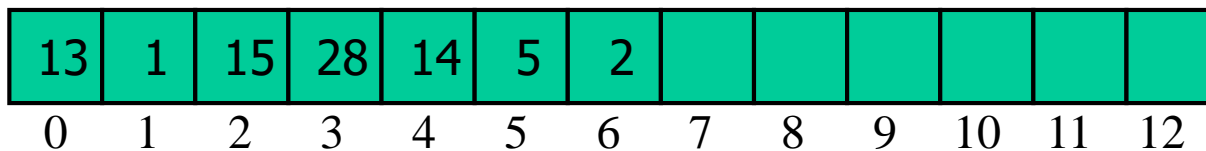
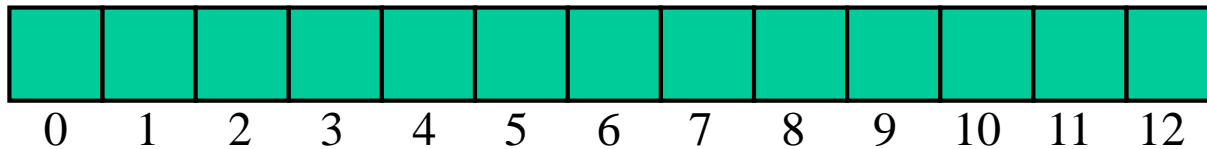
$\alpha=n/N$	$C(\alpha)$
0.1 (10%)	1.06
0.5 (50%)	1.50
0.75 (75%)	2.50
0.9 (90%)	5.50

Example of Linear probing

$$N = 13$$

$$h_j(k_i) = [h(k_i) + j] \bmod N$$

Insert keys 13,15,5,28,1,14,2 in this order



Problem with Linear Probing: PRIMARY CLUSTERING

2	CAT
3	CRICKET
4	ELEPHANT
5	FOX
6	CCC
7	



$$h(\text{COYOTE}) = 2$$

$$h_1(\text{COYOTE}) = 3$$

$$h_2 = 4$$

$$h_3 = 5$$

$$h_4 = 6$$

$$h_5 = 7 !$$



Here we are using as address generation the integer corresponding to the first letter

Idea:

Use a non-linear probe

Collision Resolution

(2) Quadrating Probing

$$\underbrace{h(k_i)}_{h_0(k_i)}, \underbrace{h(k_i)+1}_{h_1(k_i)}, h(k_i)+4, h(k_i)+9, \dots$$

$$h_j(k_i) = [h(k_i) + j^2] \bmod N$$

N: prime

→ *mod* is hard to calculate

→ Visits only half of the table

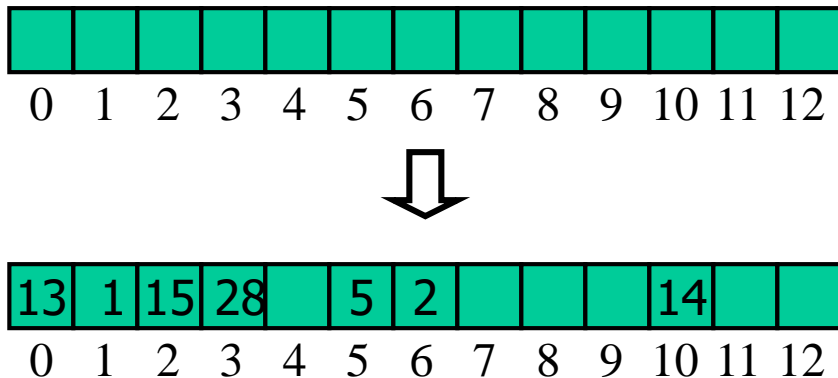
but...

Example of Quadratic probing

$$N = 13$$

$$h_j(k_i) = [h(k_i) + j^2] \bmod N$$

Insert keys 13,15,5,28,1,14,2 in this order



13	1	15	28	-	5	2	-	-	-	14	-	-
0	1	2	3	4	5	6	7	8	9	10	11	12

Find 14:

$$h_0(14) = 14 \bmod 13 = 1$$

Probe 1

Probe 2

Probe 5

Probe 10 --- FOUND

13	1	15	28	-	5	2	-	-	-	14	-	-
0	1	2	3	4	5	6	7	8	9	10	11	12

Delete 5:

Probe 5

13	1	15	28	-	avail	2	-	-	-	14	-	-
0	1	2	3	4	5	6	7	8	9	10	11	12

13	1	15	28	-	avail	2	-	-	-	14	-	-
0	1	2	3	4	5	6	7	8	9	10	11	12

Find 14:

Probe 1

Probe 2

Probe 5

???

Probe 10

Performances of Quadratic Probing

Experimental results for a hash table
with load factor α

Search

$\alpha = n/N$	$C(\alpha)$
0.1 (10%)	1.05
0.5 (50%)	1.44
0.75 (75%)	1.99
0.9 (90%)	2.79

Problem with non linear Probing

CLUSTERING

Two keys that hash to the same place follow the same collision path

Idea:

Double Hashing

Collision Resolution

Open Addressing: (3) Double Hashing



Choice of primary hashing function $h()$

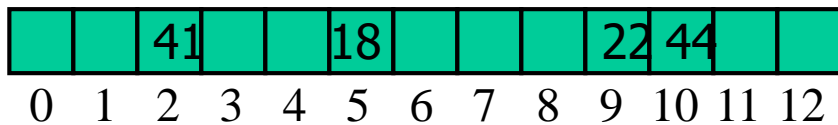
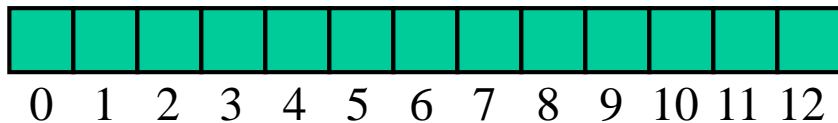
Choice of secondary hashing function $d()$

$$h_j(k_i) = [h(k_i) + j \cdot d(k_i)] \bmod N$$

Example of Double Hashing

- $N = 13$
- $h(k) = k \bmod 13$
- $d(k) = 7 - k \bmod 7$
- Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

k	$h(k)$	$d(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		



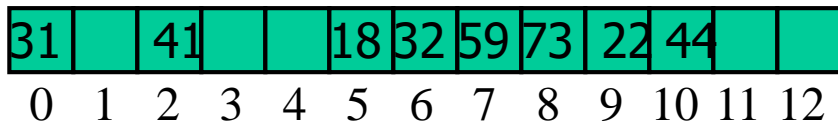
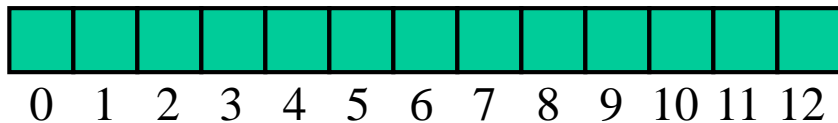
↑
occupied

$$h_j(k_i) = [h(k_i) + j \cdot d(k_i)] \bmod N$$

Example of Double Hashing

- $N = 13$
- $h(k) = k \bmod 13$
- $d(k) = 7 - k \bmod 7$
- Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

k	$h(k)$	$d(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		



31		41			18	32	59	73	22	44		
0	1	2	3	4	5	6	7	8	9	10	11	12

Remove 22 ($22 \bmod 13 = 9$)

$$h(k) = k \bmod 13$$

$$d(k) = 7 - k \bmod 7$$

31		41			18	32	59	73	AVA	44		
0	1	2	3	4	5	6	7	8	9	10	11	12

Search 31

Primary hash function: $31 \bmod 13 = 5$ **Occupied and different**

Secondary hash function: $7 - 31 \bmod 7 = 4$.

Probe cell $5+4=9$: **AVAILABLE**

Probe cell $(9+4) \bmod 13$: **FOUND**

$$h(k) = k \bmod 13$$

$$d(k) = 7 - k \bmod 7$$

Another Example of Double Hashing

$$h(k_i) = k_i \bmod N$$

$$h'(k_i) = k_i \operatorname{div} N$$

N prime!

Performances of Double Hashing

Experimental results for a hash table
with load factor α

Search

$\alpha = n/N$	$C(\alpha)$
0.1 (10%)	1.05
0.5 (50%)	1.38
0.75 (75%)	1.83
0.9 (90%)	2.55

Performance of Hashing: Summary

- In the worst case, searches, insertions and removals on a hash table take $O(n)$ time
- The worst case occurs when all the keys inserted into the dictionary collide
- The load factor $\alpha = n/N$ affects the performance of a hash table.
- The expected running time of all the dictionary ADT operations in a hash table is $O(1)$
- In practice, hashing is very fast provided the load factor is not close to 100%
- Applications of hash tables:
 - databases
 - compilers
 - browser caches

Binary and AVL Trees

Data structure

Fall 2023

Hashemite University

Dr. Mohammad Al-hammouri

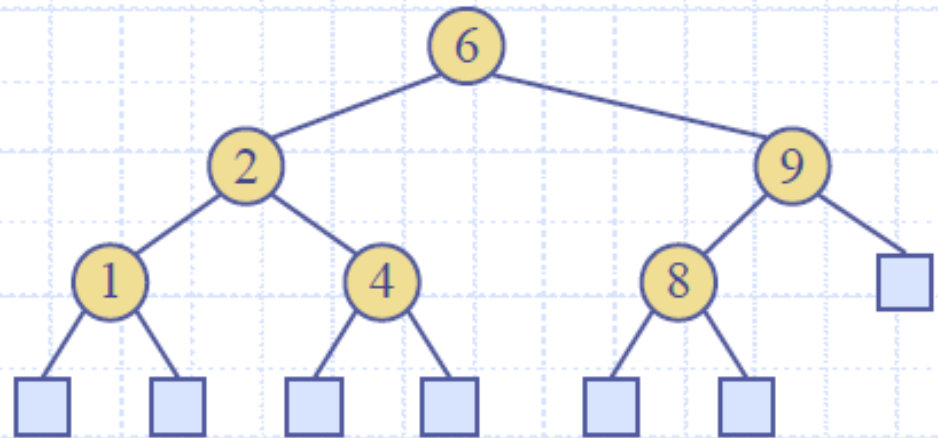
Binary Search Tree

◆ A binary search tree is a binary tree storing keys (or key-value entries) at its internal nodes and satisfying the following property:

- Let u , v , and w be three nodes such that u is in the left subtree of v and w is in the right subtree of v . We have
$$key(u) \leq key(v) \leq key(w)$$

◆ External nodes do not store items

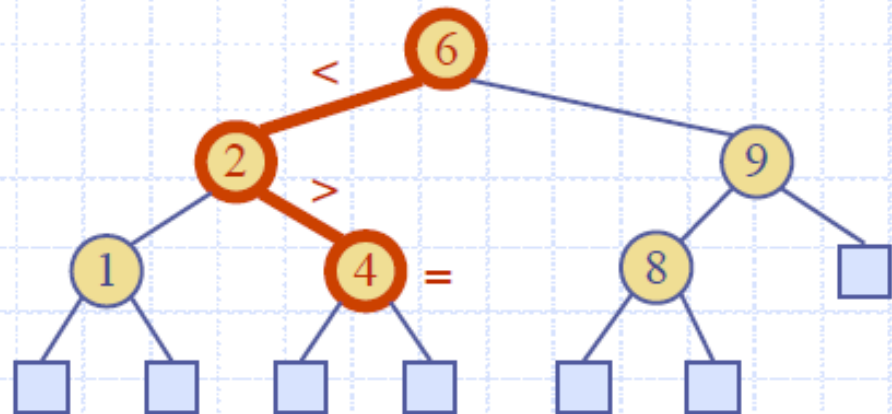
◆ An inorder traversal of a binary search tree visits the keys in increasing order



Binary Search Tree (search Function)

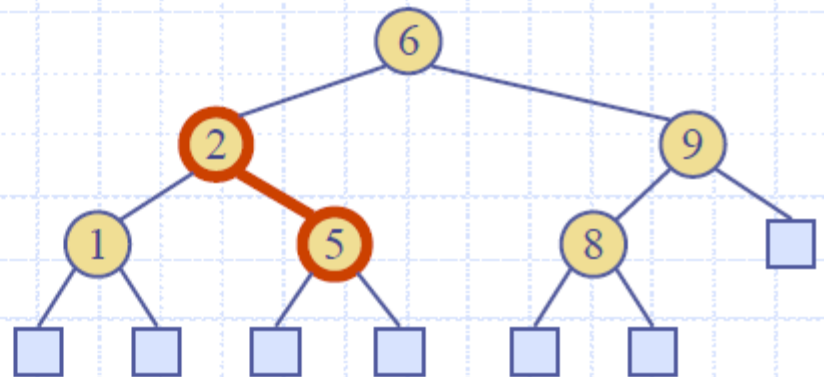
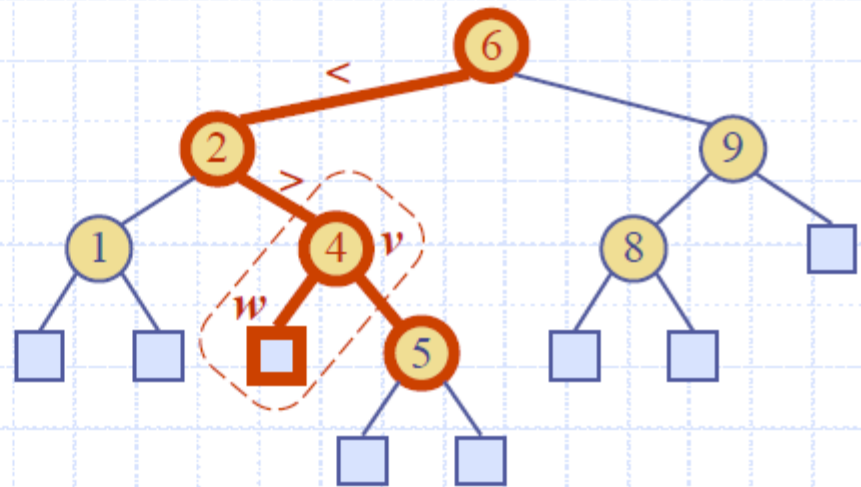
- ◆ To search for a key k , we trace a downward path starting at the root
- ◆ The next node visited depends on the comparison of k with the key of the current node
- ◆ If we reach a leaf, the key is not found
- ◆ Example: **get(4):**
 - Call `TreeSearch(4, root)`
- ◆ The algorithms for nearest neighbor queries are similar

```
Algorithm TreeSearch( $k, v$ )  
  if T.isExternal( $v$ )  
    return  $v$   
  if  $k < \text{key}(v)$   
    return TreeSearch( $k, \text{left}(v)$ )  
  else if  $k = \text{key}(v)$   
    return  $v$   
  else {  $k > \text{key}(v)$  }  
    return TreeSearch( $k, \text{right}(v)$ )
```



Binary Search Tree (Deletion)

- ◆ To perform operation **remove(k)**, we search for key k
- ◆ Assume key k is in the tree, and let v be the node storing k
- ◆ If node v has a leaf child w , we remove v and w from the tree with operation **removeExternal(w)**, which removes w and its parent
- ◆ Example: remove 4

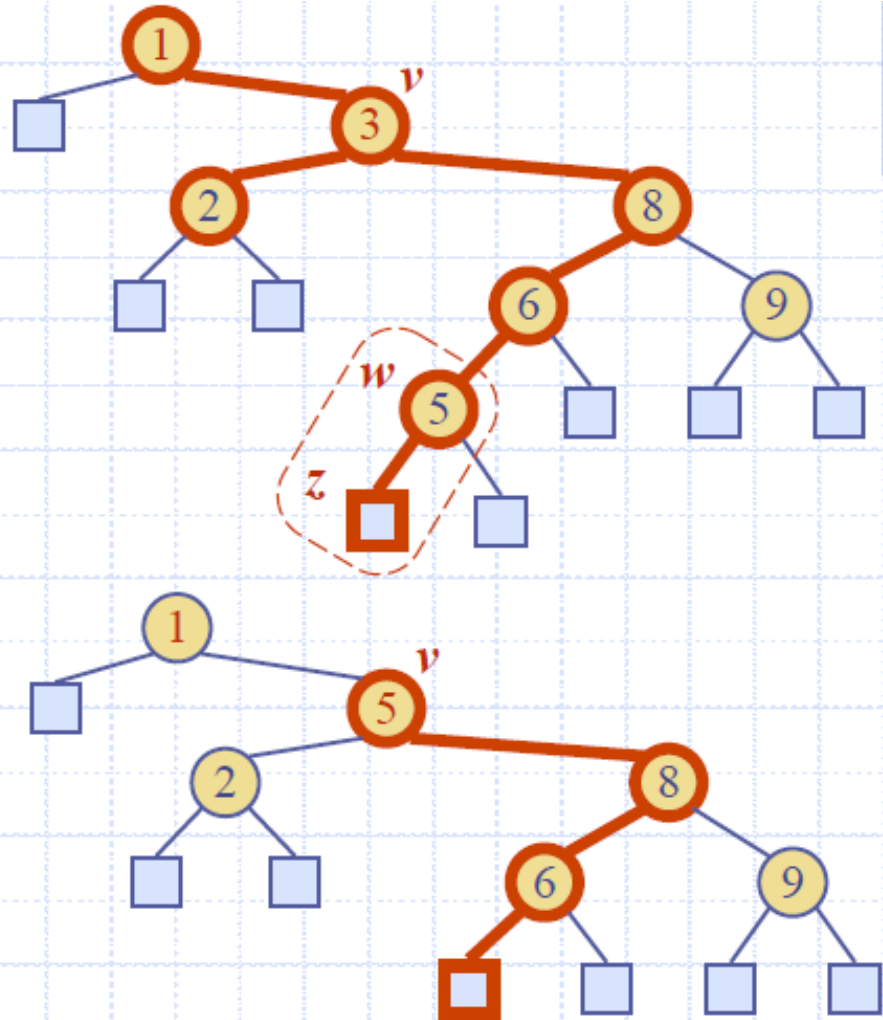


Binary Search Tree (Deletion cont.)

◆ We consider the case where the key k to be removed is stored at a node v whose children are both internal

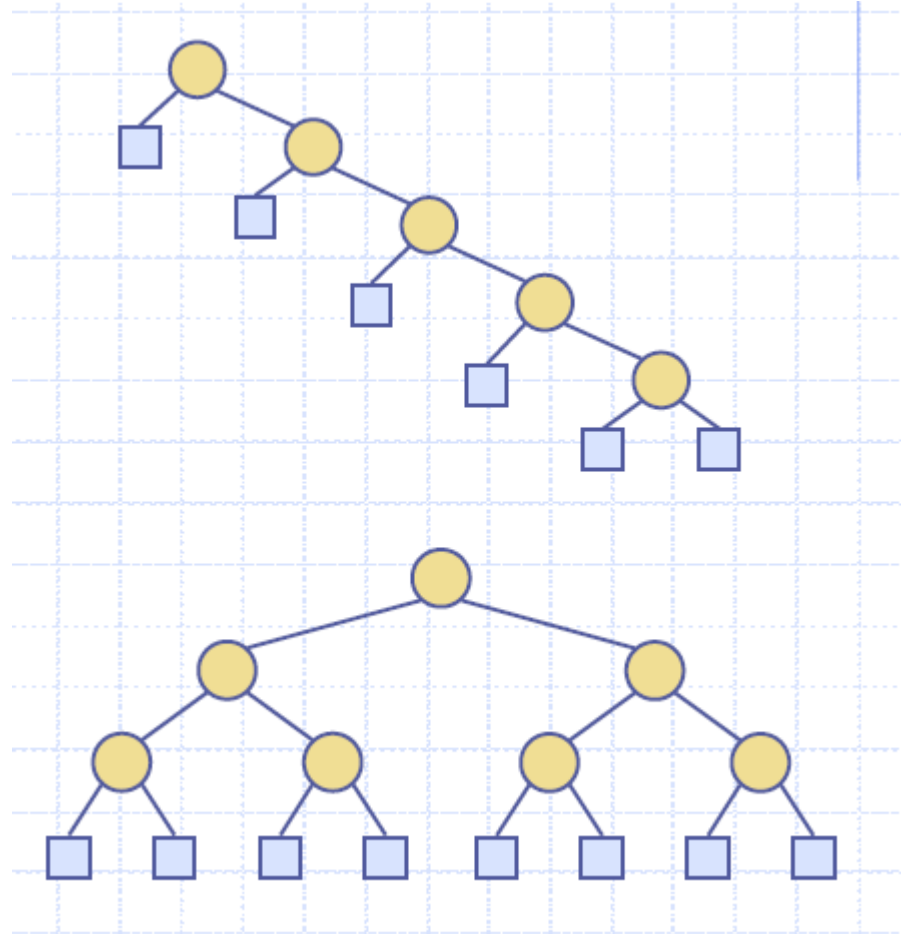
- we find the internal node w that follows v in an inorder traversal
- we copy $key(w)$ into node v
- we remove node w and its left child z (which must be a leaf) by means of operation `removeExternal(z)`

◆ Example: remove 3



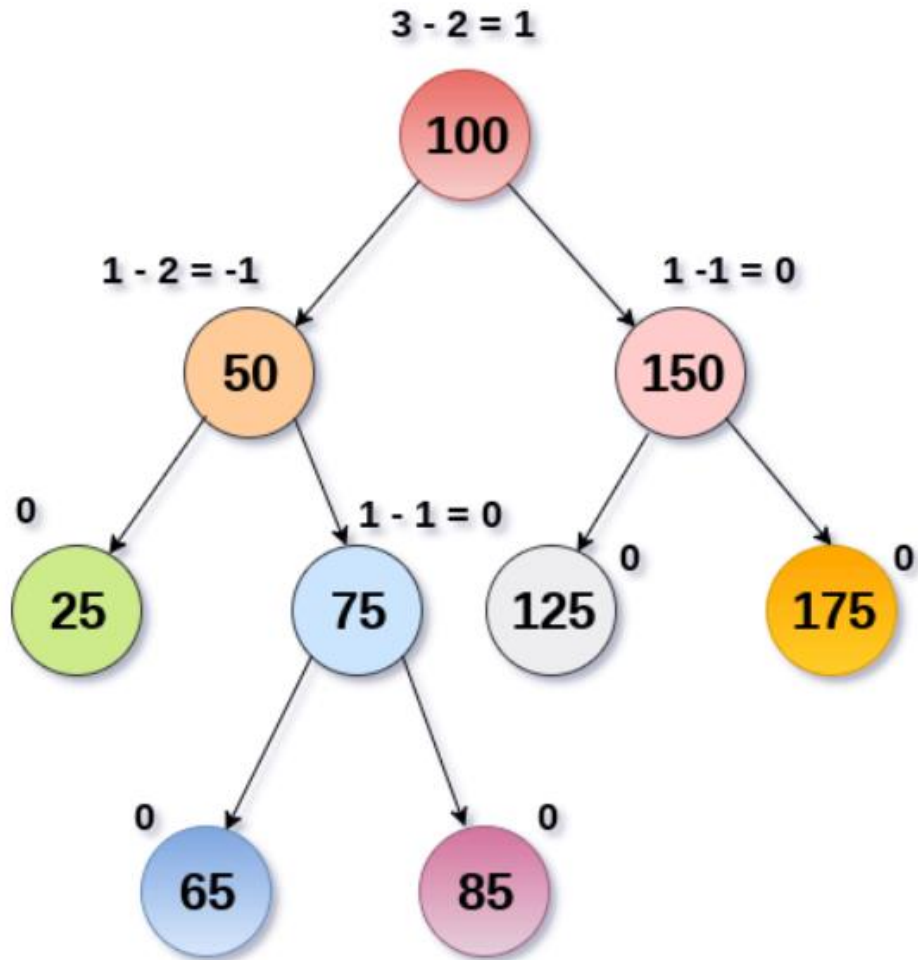
Binary Search Tree

- Consider a binary search tree of height h
- Complexity of search, insert, and remove is **$O(h)$**
- Complexity is **$O(n)$** in the worst case in **skewed** binary tree, also known as an unbalanced binary tree
- The best case is **$\log n$** in balance binary tree



AVL Trees

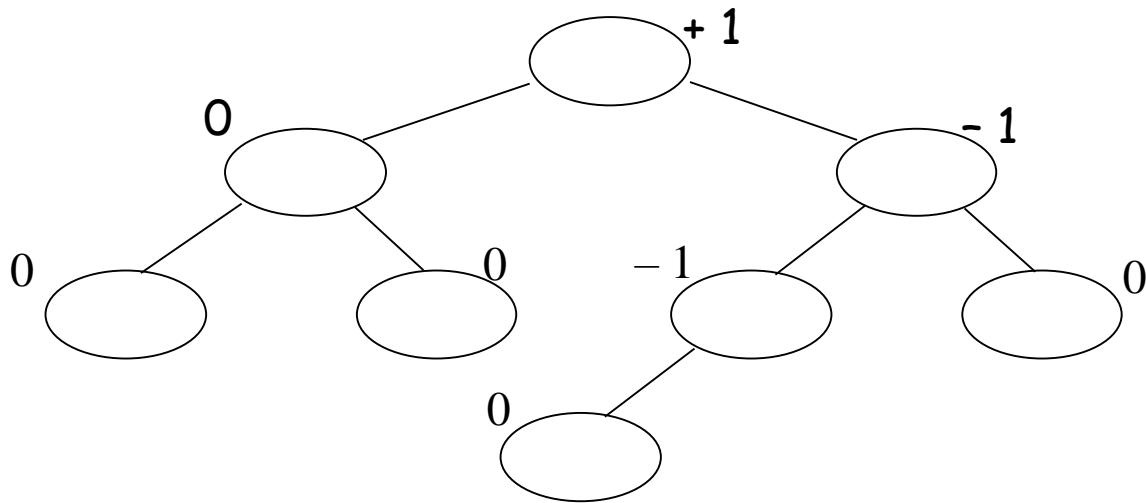
- **AVL** trees are Balanced
- An **AVL** Tree is a binary search tree such that for every internal node v of T , the heights of the children of v can differ by at most 1.
- The height of an **AVL** tree storing n keys is $O(\log n)$.



Balancing Factor

$\text{height}(\text{right subtree}) - \text{height}(\text{left subtree})$

$\in \{-1, 0, 1\}$ for AVL tree



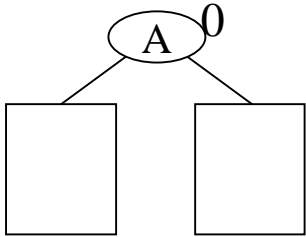
The **height** of an AVL tree T storing n keys is $O(\log n)$.

Insertion

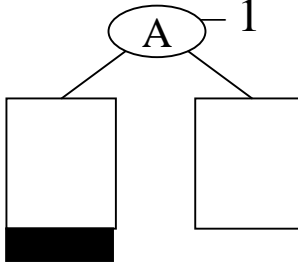
- A binary search tree T is called *balanced* if for every node v , the height of v 's children differ by at most one.
- Inserting a node into an AVL tree involves performing an `expandExternal(w)` on T , which changes the heights of some of the nodes in T .
- If an insertion causes T to become *unbalanced* we have to rebalance...

Insertion

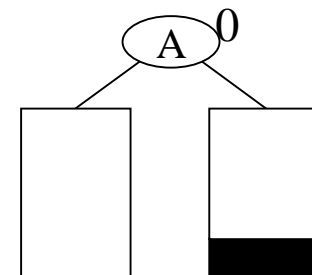
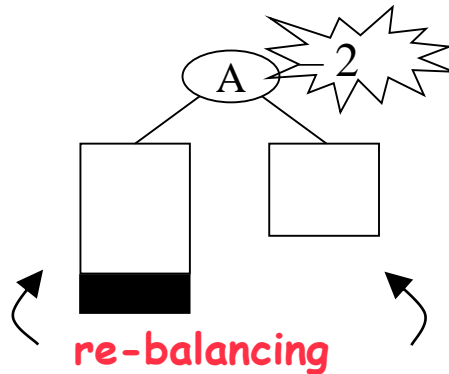
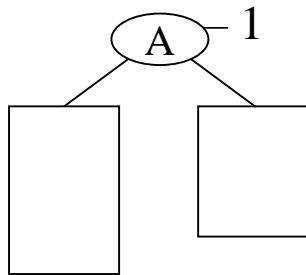
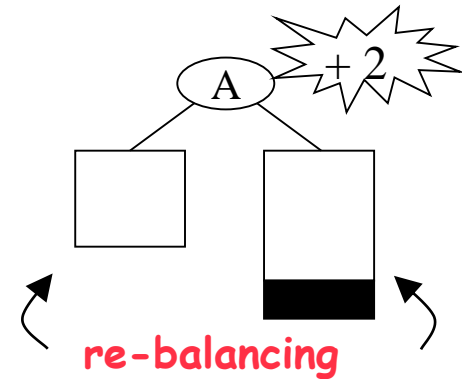
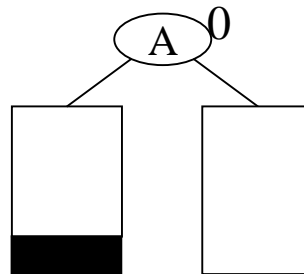
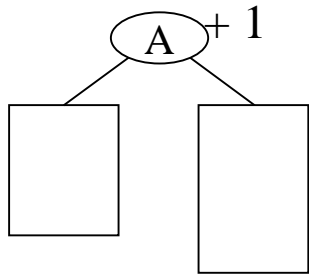
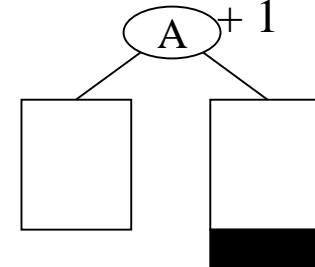
Before



After left
insertion



After right
insertion



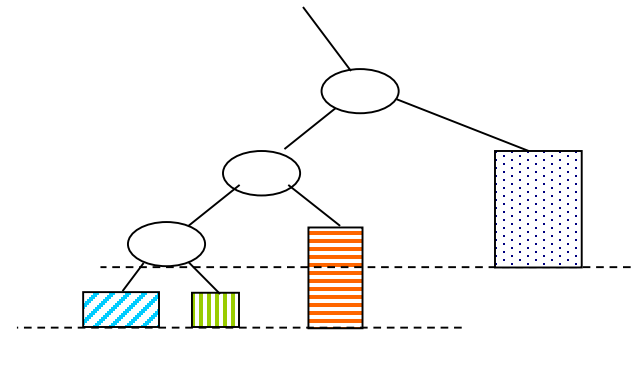
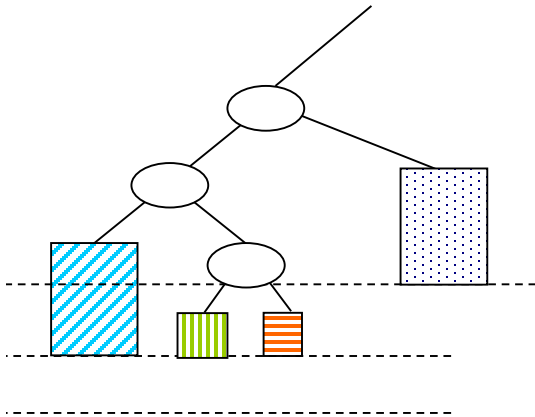
Rebalancing after insertion

We are going to identify 3 nodes which form a grandparent, parent, child triplet and the 4 subtrees attached to them. We will rearrange these elements to create a new balanced tree.

Rebalancing

Step 1: Trace the path back from the point of insertion to the first node whose **grandparent is unbalanced**. Label this node x , its parent y , and grandparent z .

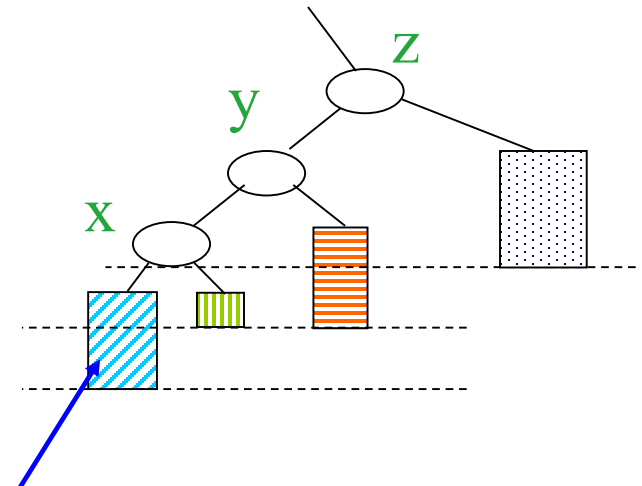
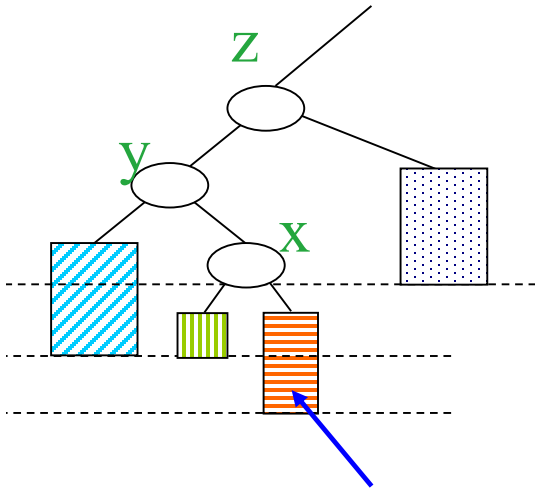
Examples



Rebalancing

Step 1: Trace the path back from the point of insertion to the first node whose **grandparent is unbalanced**. Label this node **x**, its parent **y**, and grandparent **z**.

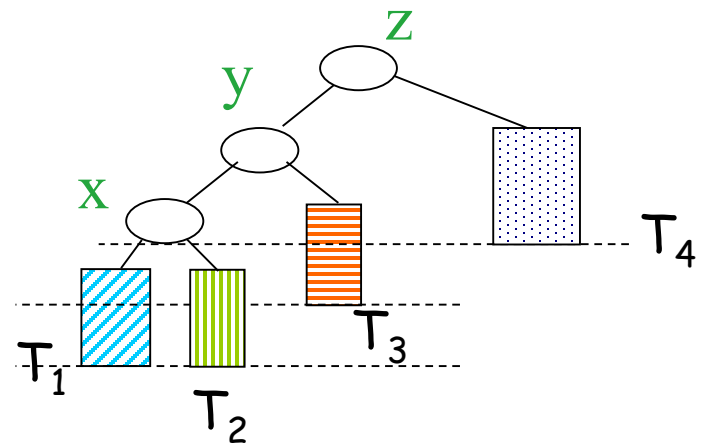
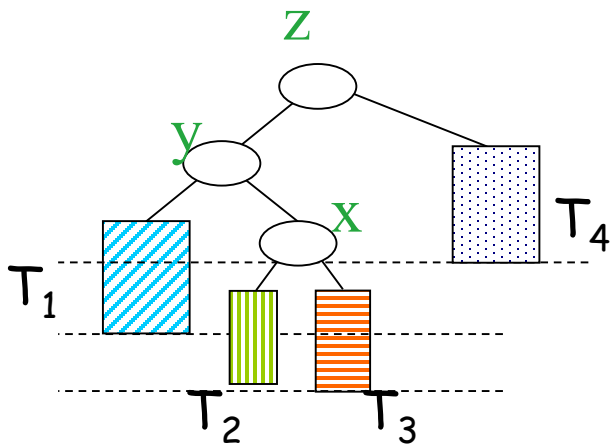
Examples



Rebalancing

Step 2: These nodes will have 4 subtrees connected to them. Label them T_1, T_2, T_3, T_4 from left to right.

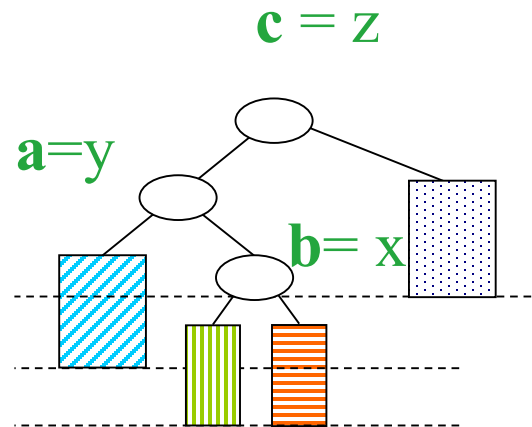
Examples



Rebalancing

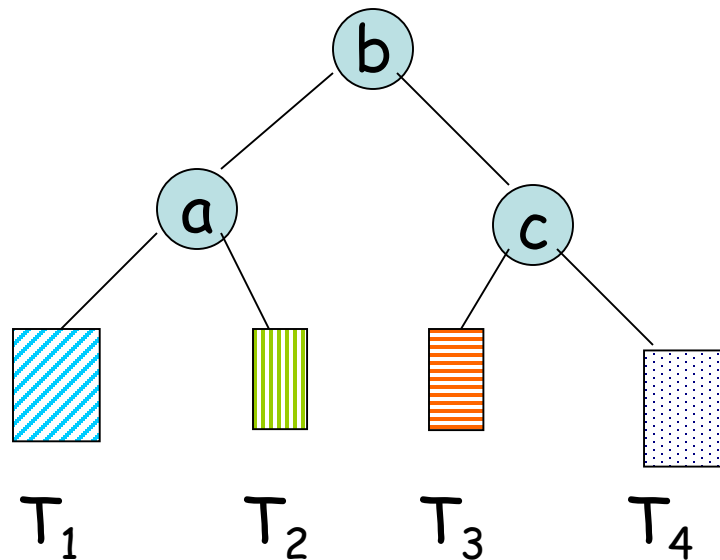
Step 3: Rename x, y, z to a, b, c according to their inorder traversal i.e. if y, x, z is the relative order of those nodes following the inorder traversal then label y 'a', x 'b' and z 'c'.

Example



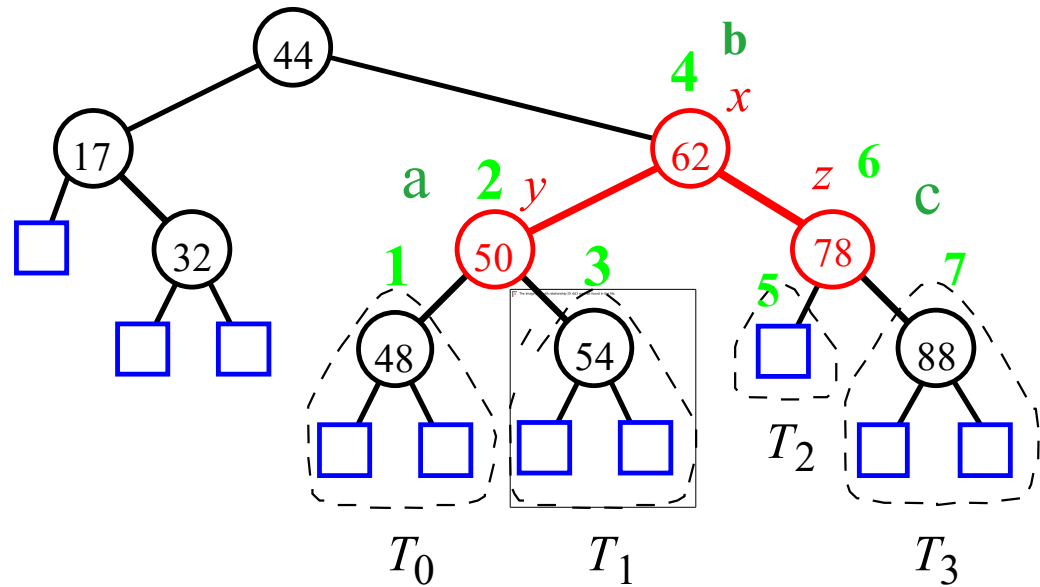
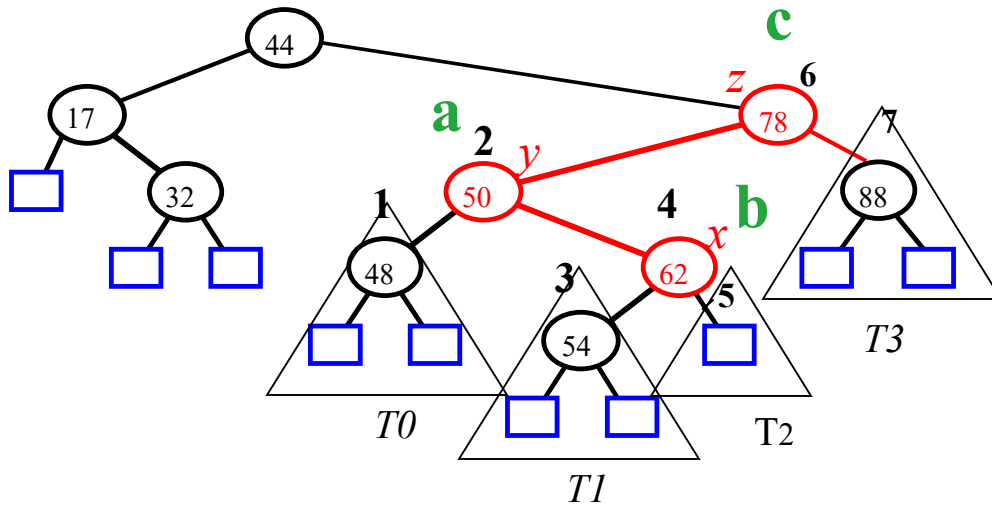
Rebalancing

Step 4: Replace the tree rooted at z with the following tree:



Rebalance done!

Example: after
inserting 54



Does this really work?

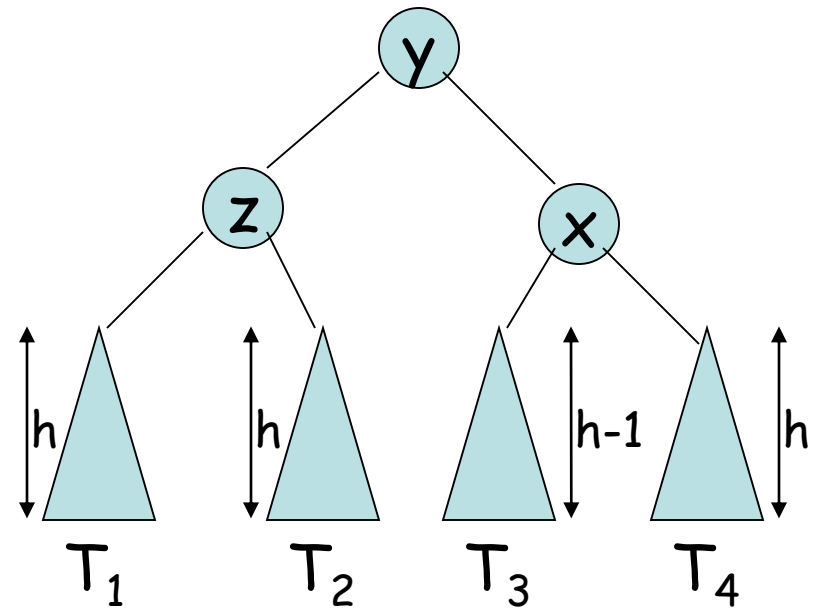
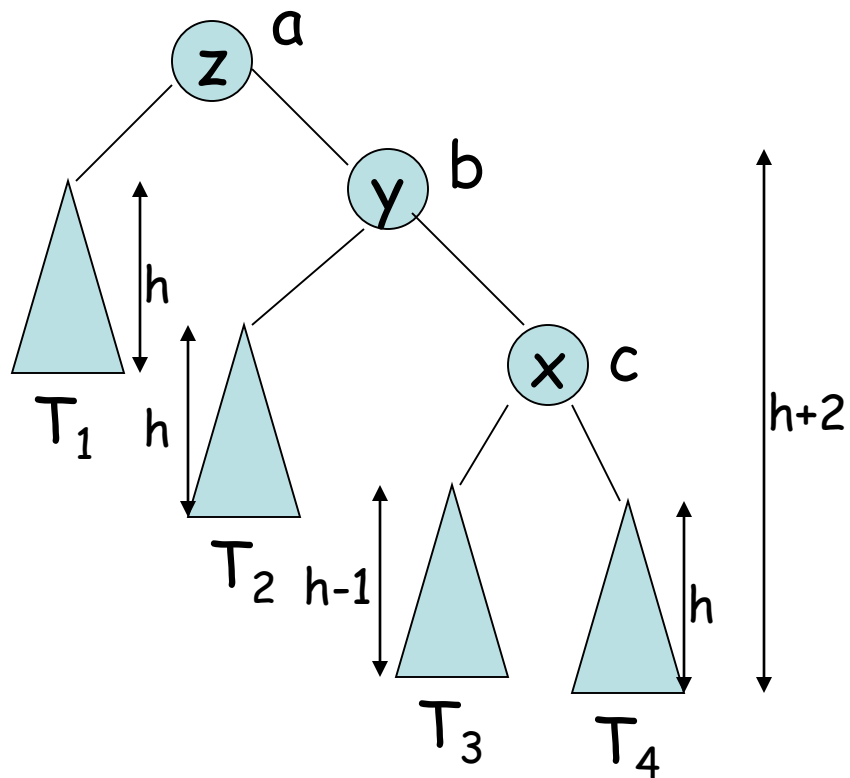
We need to see that the new tree is :

- a) A Binary search tree - the inorder traversal of our new tree should be the same as that of the old tree

Inorder traversal: by definition is T1 a T2 b T3 c T4

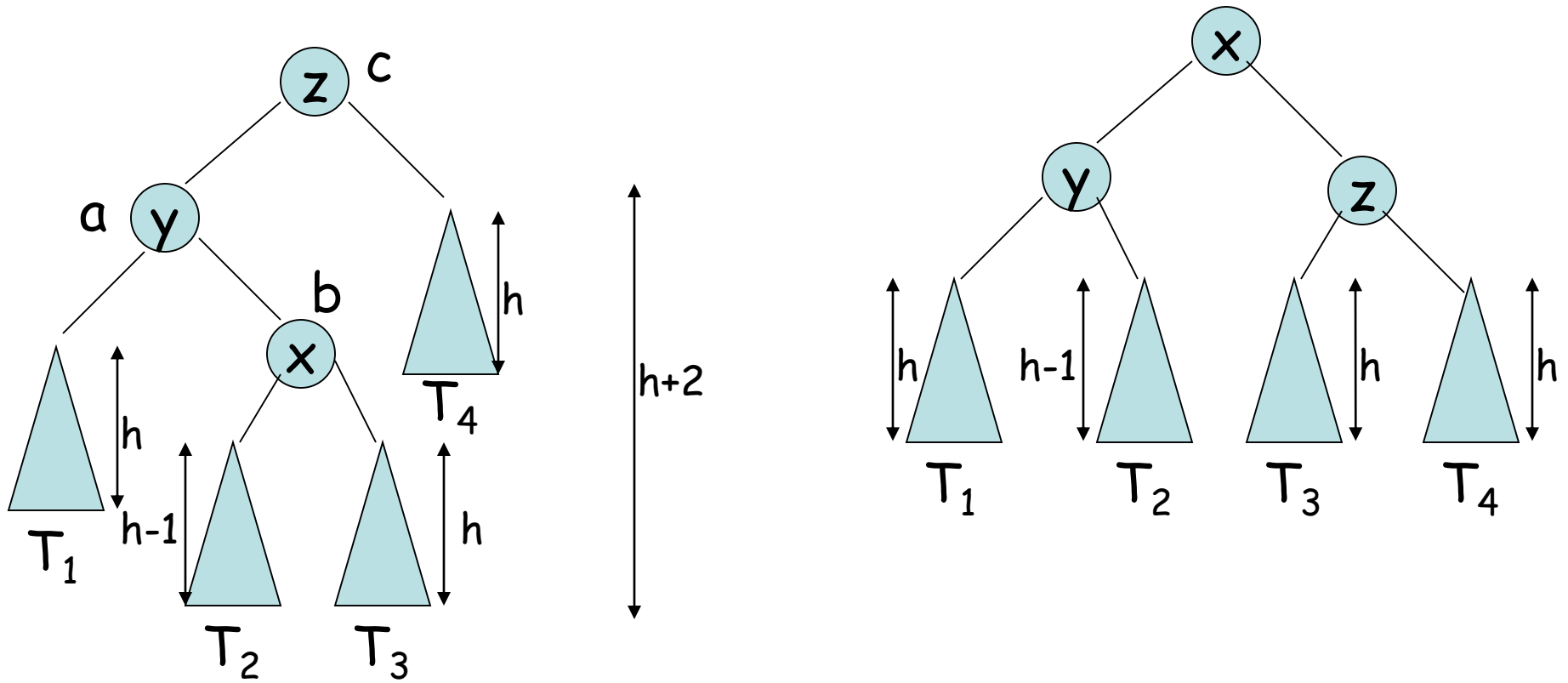
- b) Balanced: have we fixed the problem?

Example 1



Inorder: $T_1 z T_2 y T_3 x T_4$

Example 2



Inorder: $T_1 \ y \ T_2 \ x \ T_3 \ z \ T_4$