

Introduction to Algorithms and Data Structures ADS

Lars Bech Sørensen

ADS course description

Main purpose

- Design, implement and analyze different algorithms
- Become acquainted with different advanced data structures

ADS course description

Main purpose

- Design, implement and analyze different algorithms
- Become acquainted with different advanced data structures

Skills

- Be able to analyze algorithms using the Big-O notation
- Be able to design and implement algorithms and data structures in an object oriented language

ADS course description

Knowledge

- Know different linear data structures (sets, maps, queues and stacks)
- Know different non-linear data types (Trees, Heaps and Graphs)
- Know different sorting and searching algorithms
- Know different algorithm types and templates
- Know the concept of Abstract Data Types

Why study algorithms and data types?

- Speed and memory usage:
 - We want fast/efficient programs that doesn't use much memory

Why study algorithms and data types?

- Speed and memory usage:
 - We want fast/efficient programs that doesn't use much memory
- What does efficiency means?
 - Gordon E. Moore's law about components

Why study algorithms and data types?

- Speed and memory usage:
 - We want fast/efficient programs that doesn't use much memory
- What does efficiency means?
 - Gordon E. Moore's law about components
 - But the code matter just as much ...

Why study algorithms and data types?

- Speed and memory usage:
 - We want fast/efficient programs that doesn't use much memory
- What does efficiency means?
 - Gordon E. Moore's law about components
 - But the code matter just as much ...
- We need tools to analyze and document our programs regarding speed and memory usage.

**The goal:
To ensure software quality**

Program Efficiency and Correctness

Assertions and Loop Invariants

Unit testing

**The goal:
To ensure software quality**

Program Efficiency and Correctness

Assertions and Loop Invariants

Unit testing

Big-O notation

**The goal:
To ensure software quality**

Program Efficiency and Correctness

Assertions and Loop Invariants

Unit testing

Big-O notation

Software Design

Robustness, Usability, Reliability,
Maintainability, Reusability, Portability

The challenge: how to manage complexity

- Using top-down design and object-oriented design
- Managing complexity:
 - Data abstraction
 - Procedural abstraction
 - Information hiding
- Class diagrams document interactions between classes

Abstract Data Types

Using Abstraction to Manage Complexity

Abstract Data Types

Using Abstraction to Manage Complexity

- An **abstraction** is a model of a physical entity or activity
 - Models include relevant facts and details
 - Models exclude matters irrelevant to system/task

Abstract Data Types

Using Abstraction to Manage Complexity

- An **abstraction** is a model of a physical entity or activity
 - Models include relevant facts and details
 - Models exclude matters irrelevant to system/task
- **Abstraction** helps programmers:
 - Complex issues handled in manageable pieces

Abstract Data Types

Using Abstraction to Manage Complexity

- An **abstraction** is a model of a physical entity or activity
 - Models include relevant facts and details
 - Models exclude matters irrelevant to system/task
- **Abstraction** helps programmers:
 - Complex issues handled in manageable pieces
- **Procedural abstraction:** distinguishes ...
 - What to achieve (by a procedure) ...
 - From how to achieve it (implementation)

Abstract Data Types

Using Abstraction to Manage Complexity

- An **abstraction** is a model of a physical entity or activity
 - Models include relevant facts and details
 - Models exclude matters irrelevant to system/task
- **Abstraction** helps programmers:
 - Complex issues handled in manageable pieces
- **Procedural abstraction:** distinguishes ...
 - What to achieve (by a procedure) ...
 - From how to achieve it (implementation)
- **Data abstraction:** distinguishes ...
 - Data objects for a problem and their operations ...
 - From their representation in memory

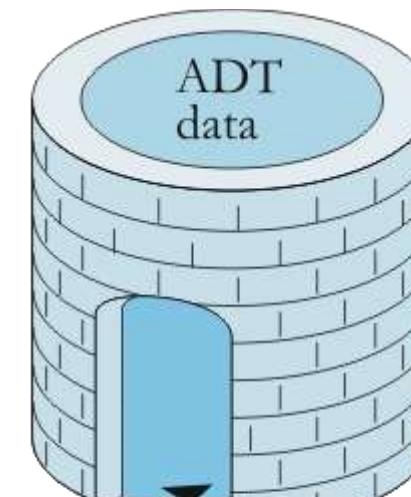
Using Abstraction to Manage Complexity (2)

- If another class uses an object only through its methods, the other class will not be affected if the data representation changes
- **Information hiding:** Concealing the details of a class implementation from users of the class
 - Enforces the discipline of data abstraction

Abstract Data Types, and Pre- and Post-conditions

- A major goal of software engineering: write reusable code
- **Abstract data type** (ADT): data + methods
 - Names, parameters, return types of methods
 - No indication of how achieved (procedural abstraction)
 - No representation (data abstraction)
- A class may **implement** an ADT
 - Must provide bodies for all methods of the ADT

Abstract Data Types.



ADT
operations

Abstract Data Types, and Pre- and Postconditions (continued)

- An ADT is a contract between
 - The interface designer and ...
 - The coder of a class that implements the interface
- **Precondition:** any assumption/constraint on the method data before the method begins execution
- **Postcondition:** describes result of executing the method

JAVA API

- JAVA has an extensive library of collections:
- **The Collections Framework** (try google it)
- Why learn how to implement ADT's ?
 - General understanding (makes you better programmers)
 - Special cases (not found in Java)
 - Choice of ADT (what is the best ADT for a given task)

Software patterns

- Software patterns are usually considered to be on a higher level than ADT's and algorithms.
- ADT and algorithms are the building blocks of many software patterns.
- The goal of software quality is achieved using both ADT's and software patterns.
- We will encounter examples of that in this course (but this is not the main focus of ADS).

Methodology

- We will be using Unit testing and Test Driven Development (TDD)
- No difference regarding analysis and design
- Regardless of approach unit tests are the foundation (and they are part of the documentation)
- More about TDD later in the course

Algorithms and Pseudo-Code

- An *ordered sequence* of *unambiguous and well-defined instructions* that *performs some task* and *halts in finite time*.
1. an *ordered sequence* means that you can number the steps
 2. *unambiguous and well-defined instructions* means that each instruction is clear, do-able, and can be done without difficulty
 3. *performs some task*
 4. *halts in finite time* (algorithms terminate!)

Algorithms and Pseudo-Code

Algorithmic Operations:

1. sequential operations - instructions are executed in order
2. conditional ("question asking") operations - a control structure that asks a true/false question and then selects the next instruction based on the answer
3. iterative operations (loops) - a control structure that repeats the execution of a block of instructions

How to represent algorithms?

1. Use natural languages:
 - too verbose
 - too "context-sensitive"- relies on experience/history/culture of the reader
2. Use formal programming languages:
 - too low level
 - requires us to deal with complicated syntax of programming language
3. Pseudo-Code:

natural language constructs modelled to look like statements available in many programming languages

Example

Algorithm: arrayMax(A,n)

Input array A of n integers

Output maximum element of A

currentMax =A[0]

for i=1 to i=n-1

 if A[i]>currentMax

 then currentMax=A[i]

return currentMax

Exercise

Make pseudocode for

- `arraySum`

 Input: array A of n integer elements

 Output: sum of all elements

- `isElement`

 Input: array A of n integer elements and integer x

 Output: 1 if there exist $A[i] = x$, 0 if not

- `sortArray`

 Input: array A of n integer elements in random order

 Output: array A sorted

arraySum

Algorithm: arraySum(A,n)

Input array A of n integers

Output: sum of all elements

sum =0

for i=0 to i=n-1

 sum = sum + A[i]

return sum

isElement

Algorithm: isElement(A,n,x)

Input array A of n integers and
integer x

Output: 1 if there exist $A[i] = x$, 0 if not

isElm =0

for i=0 to i=n-1

 if $A[i] == x$

 then isElm = 1

return isElm

sortArray

Algorithm: sortArray

Input: array A of n integer elements in
random order

Output: array A sorted

i = 0

while (i < n-1)

 while (A[i] > A[i+1])

 temp = A[i]

 A[i] = A[i+1]

 A[i+1] = temp

 if i > 0

 i --

 i++

return A

Program Efficiency and Correctness

Assertions and Loop Invariants
Big-O notation

Reasoning about Programs: Assertions and Loop Invariants

- **Assertions:**
 - Logical statements about program state
 - Claimed to be true
 - At a particular point in the program
 - Written as a comment, OR use **assert** statement
- Preconditions and postconditions are assertions
- Loop invariants are also assertions

Reasoning about Programs: Loop Invariants

A **loop invariant**:

- Helps prove that a loop meets its specification
- Is true before loop begins
- Is true at the beginning of each iteration
- Is true just after loop exit

Example: Sorting an array of n elements

Sorted(i): Array elements j , for $0 \leq j < i$, are sorted

Beginning: Sorted(0) is (trivially) true

Middle: We insure initial portion sorted as we increase i

End: Sorted(n): All elements $0 \leq j < n$ are sorted

Efficiency of Algorithms

Question: How can we characterize the performance of an algorithm ...

- Without regard to a *specific computer*?
- Without regard to a *specific language*?
- Over a wide *range of inputs*?

Desire: Function that describes execution time in terms of input size

- Other measures might be memory needed, etc.

Efficiency of Algorithms

How many operations?

a = a + 3;

Efficiency of Algorithms

How many operations?

```
int a = 0;  
  
for (int i=0; i<n; i++) {  
    a = a + i  
}
```

The “Order” of Performance: (Big) O

- Basic idea:
 1. Ignore constant factor: computer and language implementation details affect that: go for fundamental rate of increase with problem size.
 2. Consider fastest growing term: Eventually, for large problems, it will dominate: $x^2 - 100*x$
- Value: Compares fundamental performance difference of algorithms
- Caveat: For smaller problems, big-O worse performer may actually do better

$$T(n) = O(f(n))$$

- $T(n)$ = time for algorithm on input size n
- $f(n)$ = a simpler function that grows at about the same rate
- Example: $T(n) = 3n^2+5n+17 = O(n^2)$
 - $f(n)= n^2$ has faster growing term
 - no extra leading constant in $f(n)$

$T(n) = O(f(n))$ Defined

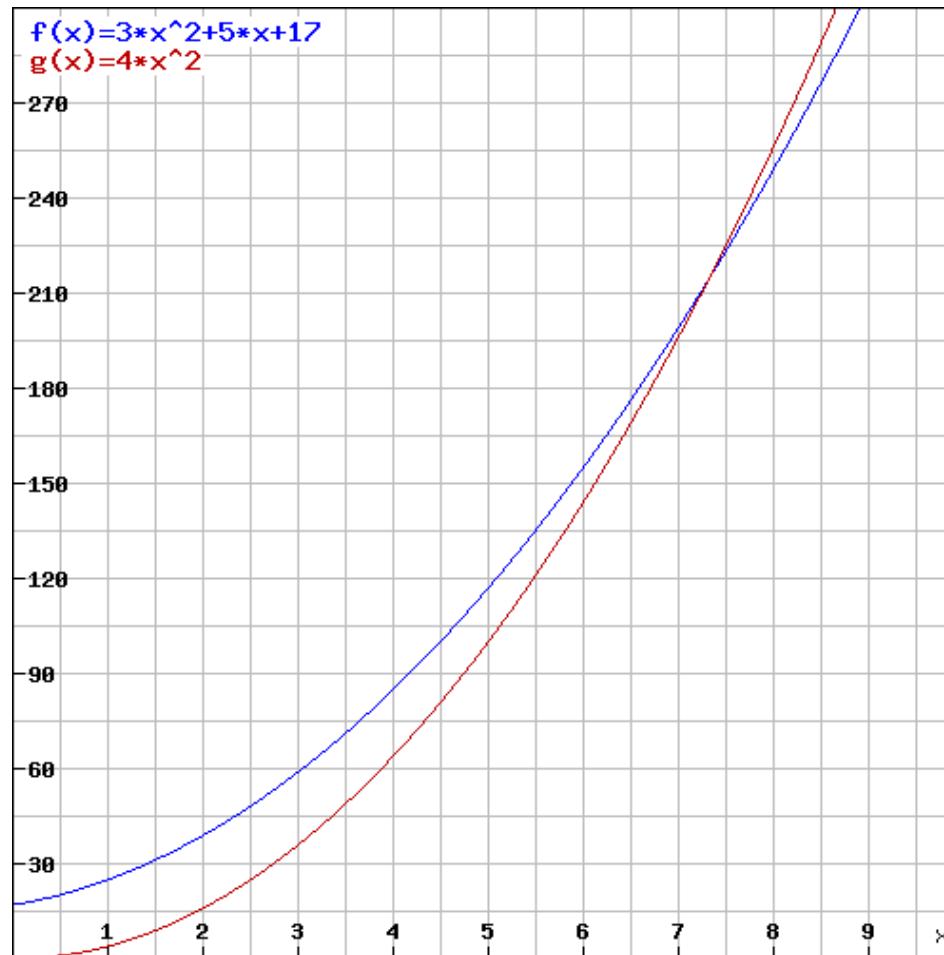
1. $\exists n_0$ and
2. $\exists c$ such that

If $n > n_0$ then $c \cdot f(n) \geq T(n)$

Example: $T(n) = 3n^2 + 5n + 17$

Pick $c = 4$, say; need $4n_0^2 > 3n_0^2 + 5n_0 + 17$
 $n_0^2 > 5n_0 + 17$, for which $n_0 = 8$ will do.

C=4



Efficiency of Algorithms (continued)

Symbols used in Quantifying Software Performance

$T(n)$	The time that a function takes as a function of the number of inputs, n . We may not be able to measure or determine this exactly.
$f(n)$	Any function of n . Generally $f(n)$ will represent a simpler function than $T(n)$, for example n^2 rather than $1.5n^2 - 1.5n$.
$\mathbf{O}(f(n))$	Order of magnitude. $\mathbf{O}(f(n))$ is the set of functions that grow no faster than $f(n)$. We say that $T(n) = \mathbf{O}(f(n))$ to indicate that the growth of $T(n)$ is bounded by the growth of $f(n)$.

Big (O) is the **worst** case scenario

Efficiency of Algorithms (continued)

Other variations:

- Omega: the growth rate of $T(N)$ is greater than or equal to $\Omega(g(N))$
- Theta: the growth rate of $T(N)$ is equal to $\Theta(h(N))$
- Little o: the growth rate of $T(N)$ is sharp less than $o(p(N))$ – big Oh allows $T(N)$ to be equal to $O(f(N))$

Big O is by far most used (we usually care about worst case – and average case is often much more complicated to find).

Efficiency of Algorithms (continued)

Common Growth Rates

Big-O	Name
$O(1)$	Constant
$O(\log n)$	Logarithmic
$O(n)$	Linear
$O(n \log n)$	Log-Linear
$O(n^2)$	Quadric
$O(n^3)$	Cubic
$O(2^n)$	Exponential
$O(n!)$	Factorial

Logarithmic the opposite of exponential

Definition: $c = 2^r \iff \log_2(c) = r$

Binary logarithm (base 2) often written \log_2 or \ln_2 .

Natural logarithm (\ln) base e

Decimal logarithm (\log) base 10

$$\text{Log}_2(n) = \ln(n)/\ln(2)$$

$$r=1: c = 2^1 = 2$$

$$\log_2(2) = 1$$

$$r=2: c = 2^2 = 4$$

$$\log_2(4) = 2$$

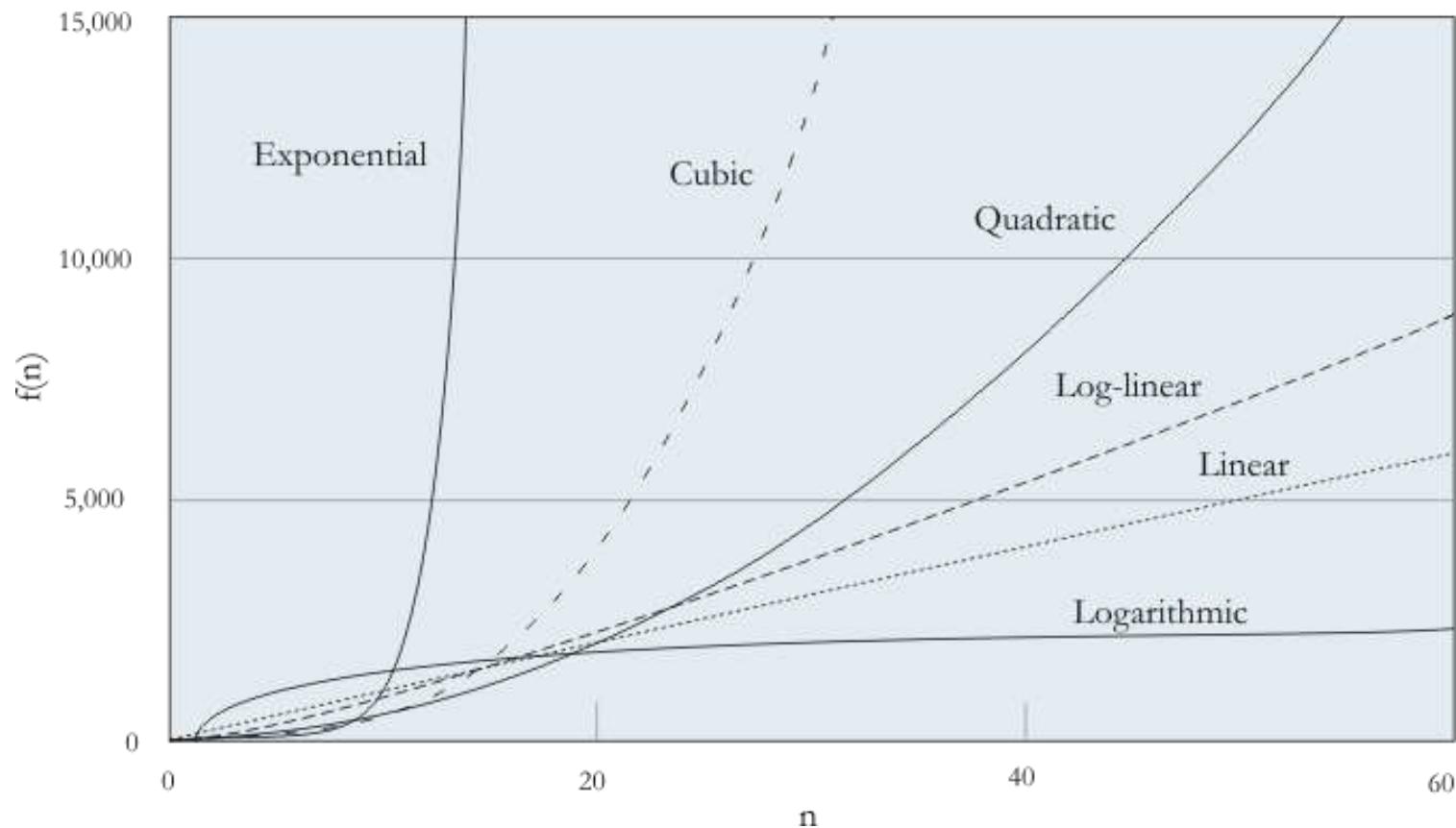
$$r=3: c = 2^3 = 8$$

$$\log_2(8) = 3$$

$$r=4: c = 2^4 = 16$$

$$\log_2(16) = 4$$

Efficiency of Algorithms (continued)



Efficiency Examples

One loop

```
int find (int x[], int val) {  
    for (int i = 0; i < x_LENGTH; i++) {  
        if (x[i] == val)  
            return i;  
    }  
    return -1; // not found  
}
```

Efficiency Examples

One loop

```
int find (int x[], int val) {  
    for (int i = 0; i < x_LENGTH; i++) {  
        if (x[i] == val)  
            return i;  
    }  
    return -1; // not found  
}
```

Letting n be `x.length`:

Average iterations if $found = n/2 = 0.5*n = O(n)$

Iterations if $not\ found = n = O(n)$

Hence this is called *linear search*.

Efficiency Examples

Using functions from a loop

```
bool all_different (int x[], int y[]) {  
    for (int i = 0; i < X_LENGTH; i++) {  
        if (find(y, x[i]) != -1)  
            return false;  
    }  
    return true; // no x element found in y  
}
```

Efficiency Examples

Using functions from a loop

```
bool all_different (int x[], int y[]) {  
    for (int i = 0; i < x_LENGTH; i++) {  
        if (find(y, x[i]) != -1)  
            return false;  
    }  
    return true; // no x element found in y  
}
```

Letting m be x_LENGTH and n be y_LENGTH m:

Time if all different =

$$m \cdot \text{cost of search}(n) = O(m) \cdot O(n) = O(m \cdot n)$$

Efficiency Examples

Nested loops

```
bool unique (int x[]) {  
    for (int i = 0; i < x_LENGTH; i++) {  
        for (int j = 0; j < x_LENGTH; j++) {  
            if (i != j && x[i] == x[j])  
                return false;  
        }  
    }  
    return true; // no duplicates in x  
}
```

Efficiency Examples

Nested loops

```
bool unique (int x[]) {  
    for (int i = 0; i < x_LENGTH; i++) {  
        for (int j = 0; j < x_LENGTH; j++) {  
            if (i != j && x[i] == x[j])  
                return false;  
        }  
    }  
    return true; // no duplicates in x  
}
```

Letting n be x_LENGTH:

Time if unique: Outer and inner loop runs n times so
 $O(n) * O(n)$ or n^2 iterations = $O(n^2)$

Efficiency Examples

Optimization

```
bool unique (int x[]) {  
    for (int i = 0; i < x_LENGTH; i++) {  
        for (int j = i+1; j < x_LENGTH; j++) {  
            if (x[i] == x[j])  
                return false;  
        }  
    }  
    return true; // no duplicates in x  
}
```

Efficiency Examples

Optimization

```
bool unique (int x[]) {  
    for (int i = 0; i < x_LENGTH; i++) {  
        for (int j = i+1; j < x_LENGTH; j++) {  
            if (x[i] == x[j])  
                return false;  
        }  
    }  
    return true; // no duplicates in x  
}
```

Letting n be x_LENGTH:

Time if unique = $(n-1) + (n-2) + \dots + 2 + 1$ iterations =

$$n(n-1)/2 \text{ iterations} = 0.5n^2 - 0.5n = O(n^2)$$

a factor of 2 better but *still* ... the growth rate remains polynomial

Efficiency Examples

Logarithms

```
for (int i = 1; i < n; i *= 2) {  
    do something with x[i]  
}
```

Efficiency Examples

Logarithms

```
for (int i = 1; i < n; i *= 2) {  
    do something with x[i]  
}
```

Sequence is 1, 2, 4, 8, ..., $\approx n$.

Number of iterations = $\log_2 n = \log n$.

Computer scientists generally use base 2 for log, since that matches with number of *bits*, etc.

Strictly speaking we should designate base (by writing \log_2 or \log_{10}) but base 2 is so common in computer science that we often omit it.

Chessboard Puzzle

Payment scheme #1: \$1 on first square, \$2 on second, \$3 on third, ..., \$64 on 64th.

Payment scheme #2: 1¢ on first square, 2¢ on second, 4¢ on third, 8¢ on fourth, etc.

Which is best?

Chessboard Puzzle Analyzed

Payment scheme #1: Total = \$1+\$2+\$3+...+\$64 =
 $\$64 \times 65 / 2 = \2080

Payment scheme #2: $1\text{¢} + 2\text{¢} + 4\text{¢} + \dots + 2^{63}\text{¢} = 2^{64} - 1\text{¢} = \$184.467440737 \text{ trillion}$

Many cryptographic schemes require $O(2^n)$ work to break a key of length n bits. A key of length $n=128$ is perhaps breakable, but one with $n=256$ is not.

Topics

- To learn about the List ADT
- To learn about the Stack ADT
- To learn how to implement a stack using an underlying array or linked list
- To learn how to represent a waiting line (queue) and learn how to use the methods in the queue interface
- To understand how to implement the Queue interface using a single-linked list, a circular array, and a double-linked list.

The Lists ADT

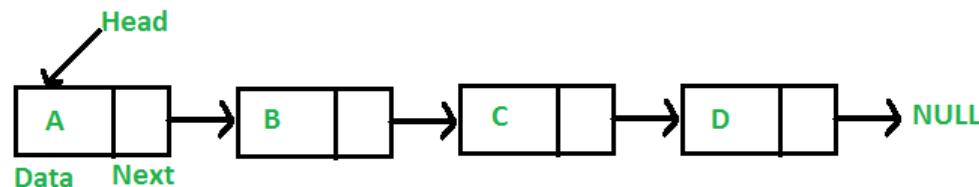
- A linear ordering of elements
- The start is often called the head
- The end is often called the tail
- All elements (except head and tail) have an element before and after.

Operations on a list:

- a constructor for creating an empty list
- an operation for testing whether or not a list is empty
- an operation for getting the size of the list
- an operation for inserting an entity into a list
 - at a specific position
 - at the tail or the head
- an operation for deleting an entity from a list
 - at a specific position
 - at the tail or the head
- an operation for finding an entity in a list

What is a linked list?
Single, double or circular.

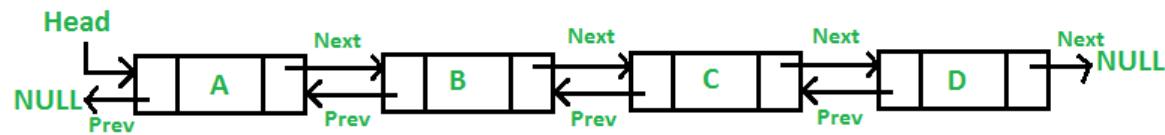
Nodes: pointers (references) and data.



Head and tail are often special sentinel nodes without data.
Helps avoid a lot special cases.

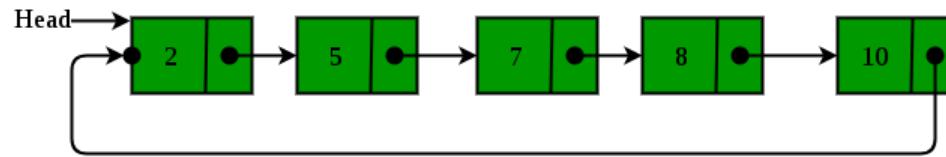
Doubly linked list

Some algorithms are more efficient if we can go forward or backward



Circular lists

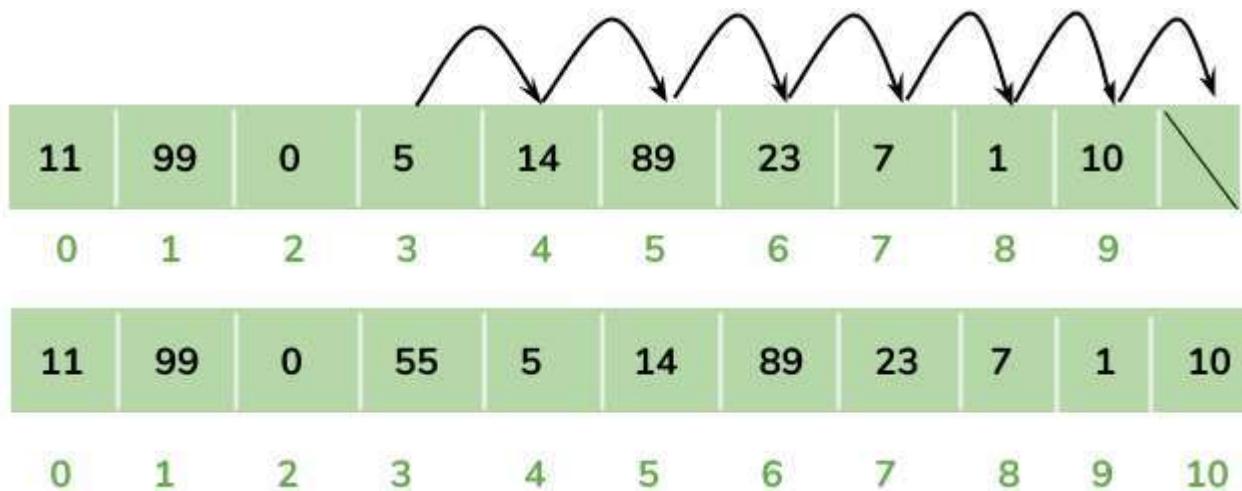
Especially good for implementing queues when used in games



ArrayList implementation of a list

- Same operations but now in an array
- Adding or deleting requires that elements are moved in order to avoid gaps or to make room for new elements (except when an element is added after the last element) :

Adding 55 at index 3 requires that all elements from index 3 and up are right shifted:



Exercise

- Find the time complexity of the list operations when implemented with linked lists and with arrays.

Stack Abstract Data Type

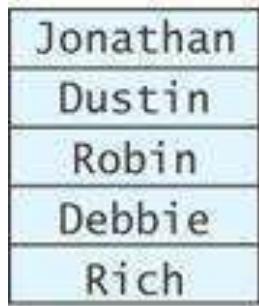
- A stack is one of the most commonly used data structures in computer science
- A stack can be compared to a Pez dispenser
 - Only the top item can be accessed
 - You can extract only one item at a time
- The top element in the stack is the last added to the stack (most recently)
- The stack's storage policy is *Last-In, First-Out*, or *LIFO*



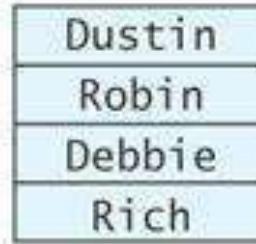
Specification of the Stack Abstract Data Type

- Only the top element of a stack is visible; therefore the number of operations performed by a stack are few
- We need the ability to
 - ❑ test for an empty stack (empty)
 - ❑ inspect the top element (peek)
 - ❑ retrieve the top element (pop)
 - ❑ put a new element on the stack (push)

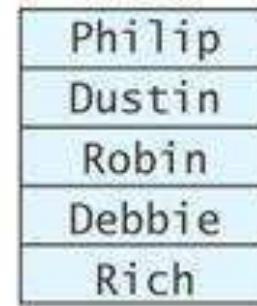
A Stack of Strings



(a)



(b)



(c)

- “Rich” is the oldest element on the stack and “Jonathan” is the youngest (Figure a)
- `String last = names.peek();` stores a reference to “Jonathan” in `last`
- `String temp = names.pop();` removes “Jonathan” and stores a reference to it in `temp` (Figure b)
- `names.push(“Philip”);` pushes “Philip” onto the stack (Figure c)

Stack Applications

Finding Palindromes

- Palindrome: a string that reads identically in either direction, letter by letter (ignoring case)
 - kayak
 - "I saw I was I"
 - “Able was I ere I saw Elba”
 - "Level madam level"
- Problem: Write a program that reads a string and determines whether it is a palindrome

Testing

- To test this class using the following inputs:
 - a single character (always a palindrome)
 - multiple characters in a word
 - multiple words
 - different cases
 - even-length strings
 - odd-length strings
 - the empty string (considered a palindrome)

Balanced Parentheses

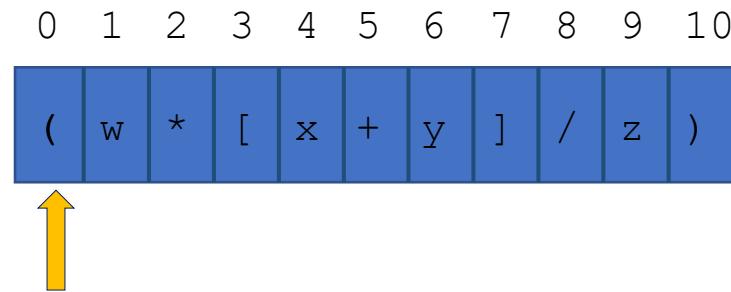
- When analyzing arithmetic expressions, it is important to determine whether an expression is balanced with respect to parentheses

$$(\ a \ + \ b \ * \ (\ c \ / \ (\ d \ - \ e \) \) \) \ + \ (\ d \ / \ e \)$$

- The problem is further complicated if braces or brackets are used in conjunction with parentheses
- The solution is to use stacks!

Balanced Parentheses (cont.)

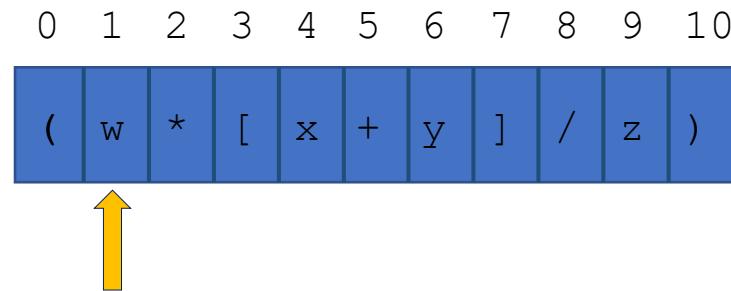
Expression: (w * [x + y] / z)



balanced : **true**
index : 0

Balanced Parentheses (cont.)

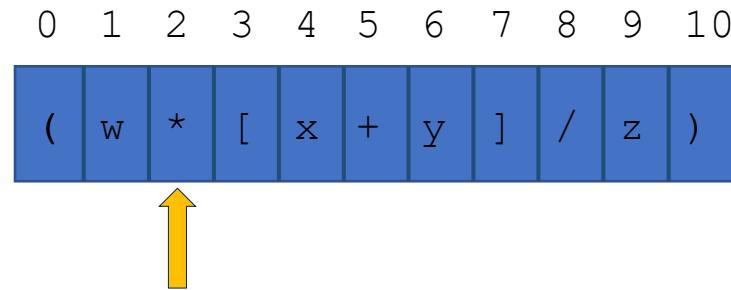
Expression: (w * [x + y] / z)



balanced : **true**
index : 1

Balanced Parentheses (cont.)

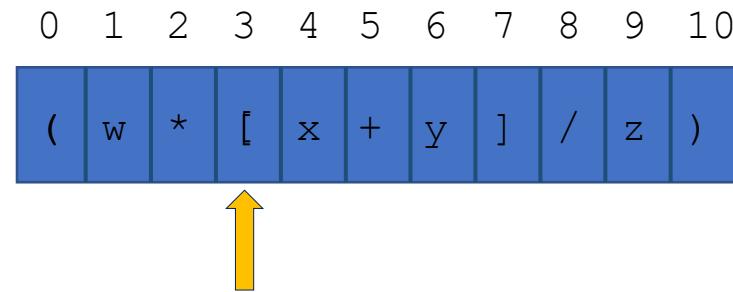
Expression: (w * [x + y] / z)



balanced : **true**
index : 2

Balanced Parentheses (cont.)

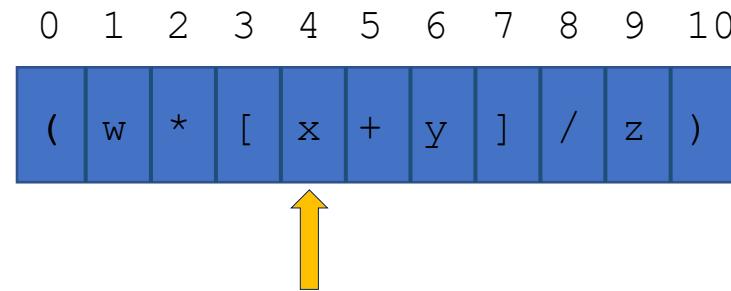
Expression: (w * [x + y] / z)



```
balanced : true  
index      : 3
```

Balanced Parentheses (cont.)

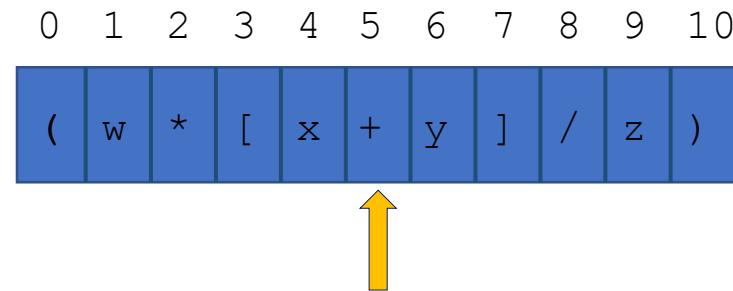
Expression: (w * [x + y] / z)



```
balanced : true  
index      : 4
```

Balanced Parentheses (cont.)

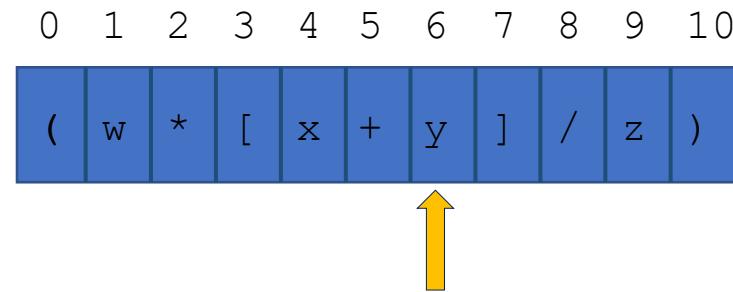
Expression: (w * [x + y] / z)



balanced : **true**
index : 5

Balanced Parentheses (cont.)

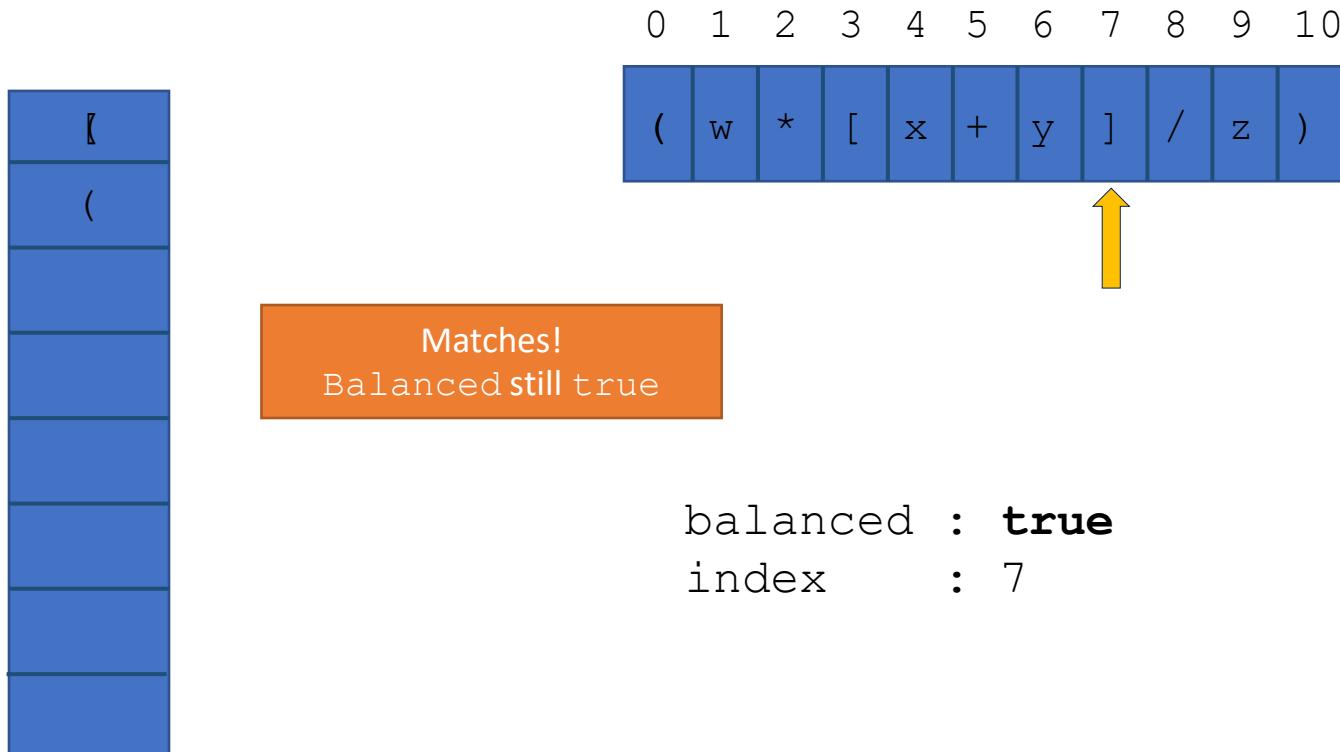
Expression: (w * [x + y] / z)



balanced : **true**
index : 6

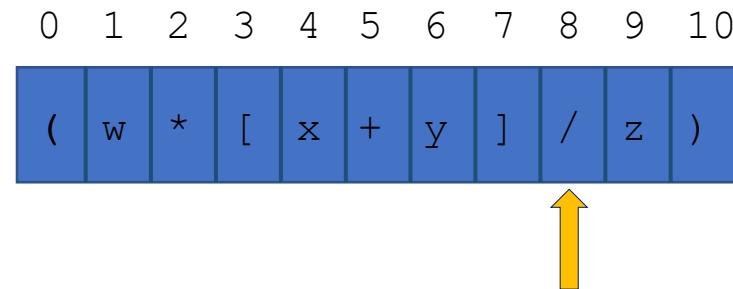
Balanced Parentheses (cont.)

Expression: (w * [x + y] / z)



Balanced Parentheses (cont.)

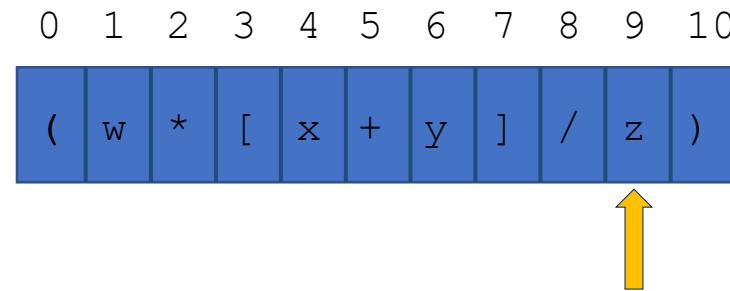
Expression: (w * [x + y] / z)



balanced : **true**
index : 8

Balanced Parentheses (cont.)

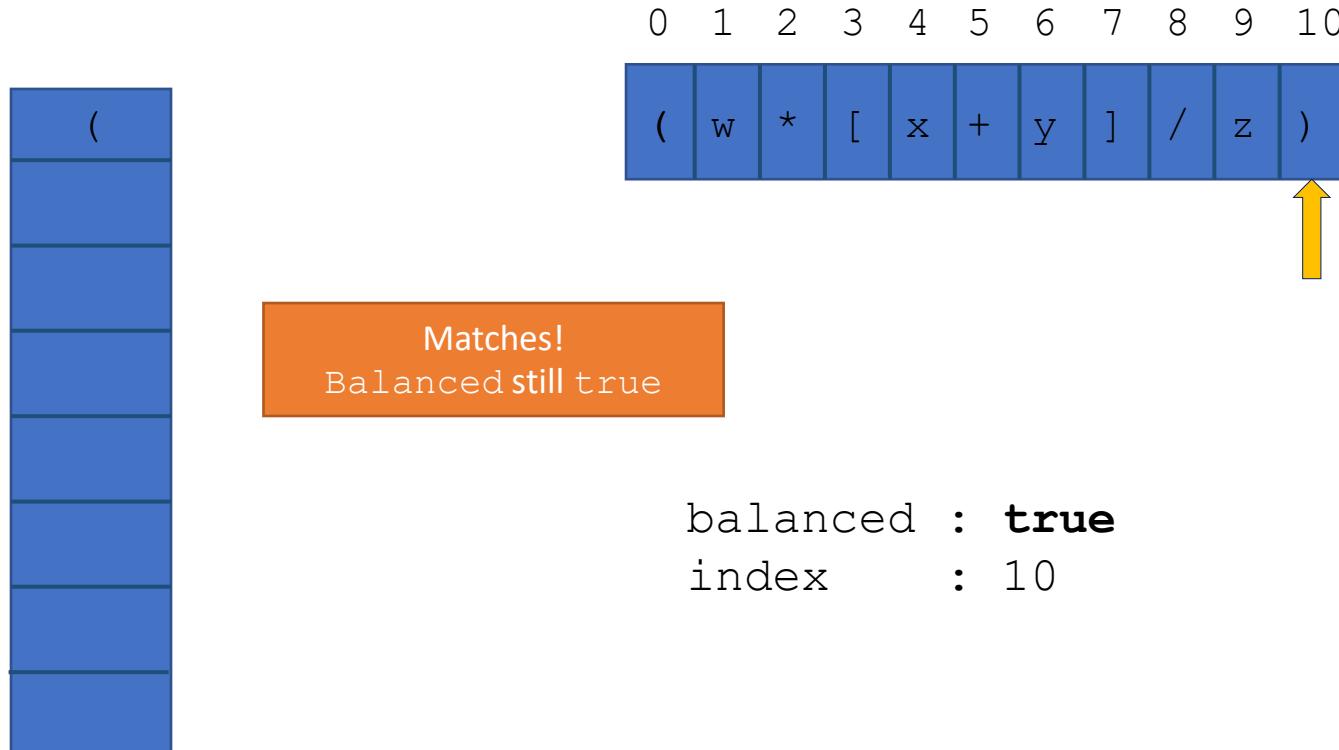
Expression: (w * [x + y] / z)



balanced : **true**
index : 9

Balanced Parentheses (cont.)

Expression: (w * [x + y] / z)



Testing

- Provide a variety of input expressions displaying the result true or false
- Try several levels of nested parentheses
- Try nested parentheses where corresponding parentheses are not of the same type
- Try unbalanced parentheses
- No parentheses at all!

- PITFALL: attempting to pop an empty stack will throw an `EmptyStackException`. You can guard against this by either testing for an empty stack or catching the exception

Implementing a Stack

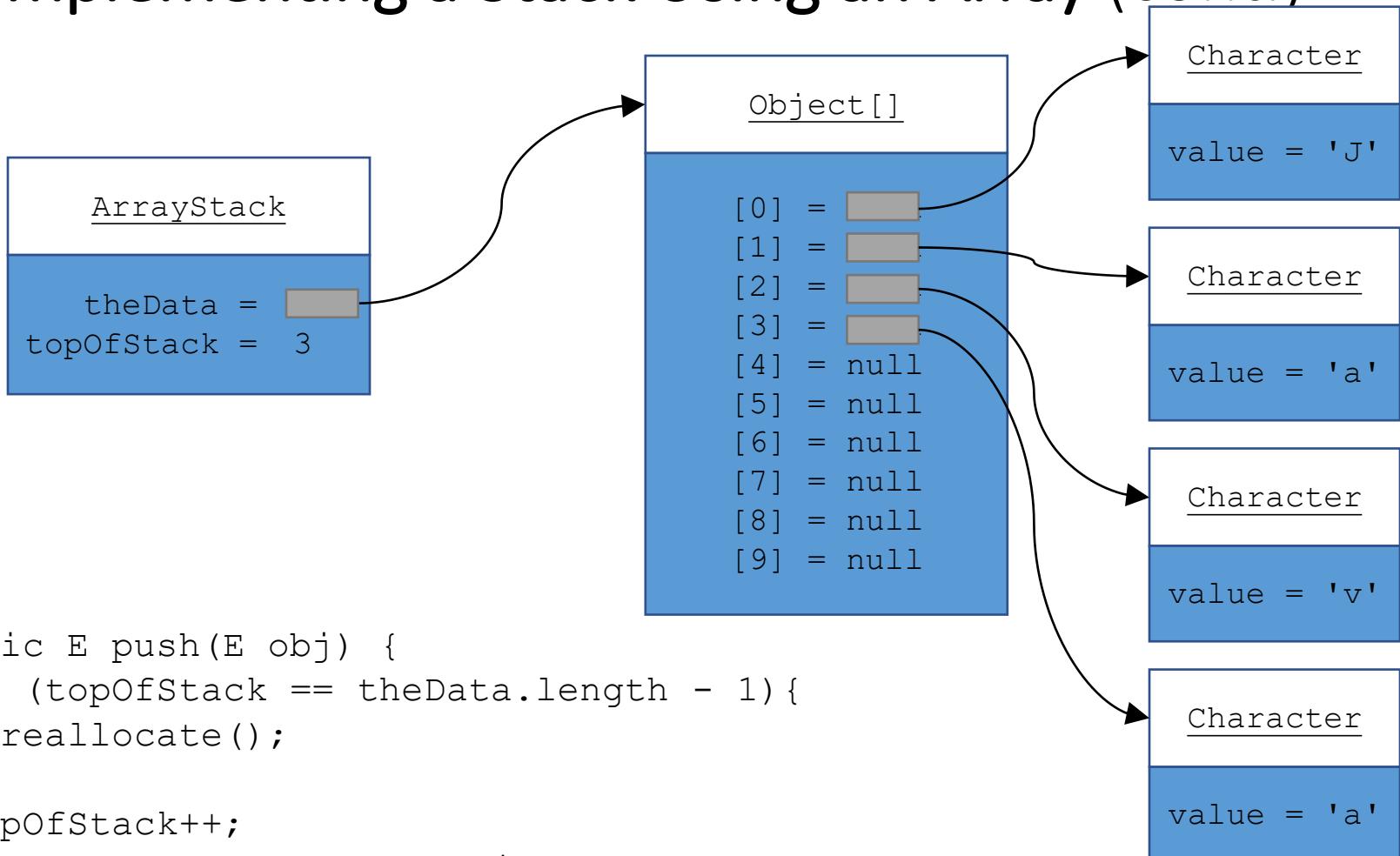
Implementing a Stack with a List Component

- We can use either the `ArrayList` or the `LinkedList` classes, as both implement the `List` interface. The `push` method, for example, can be coded as

```
public E push(E obj) {  
    theData.add(obj);  
    return obj;  
}
```

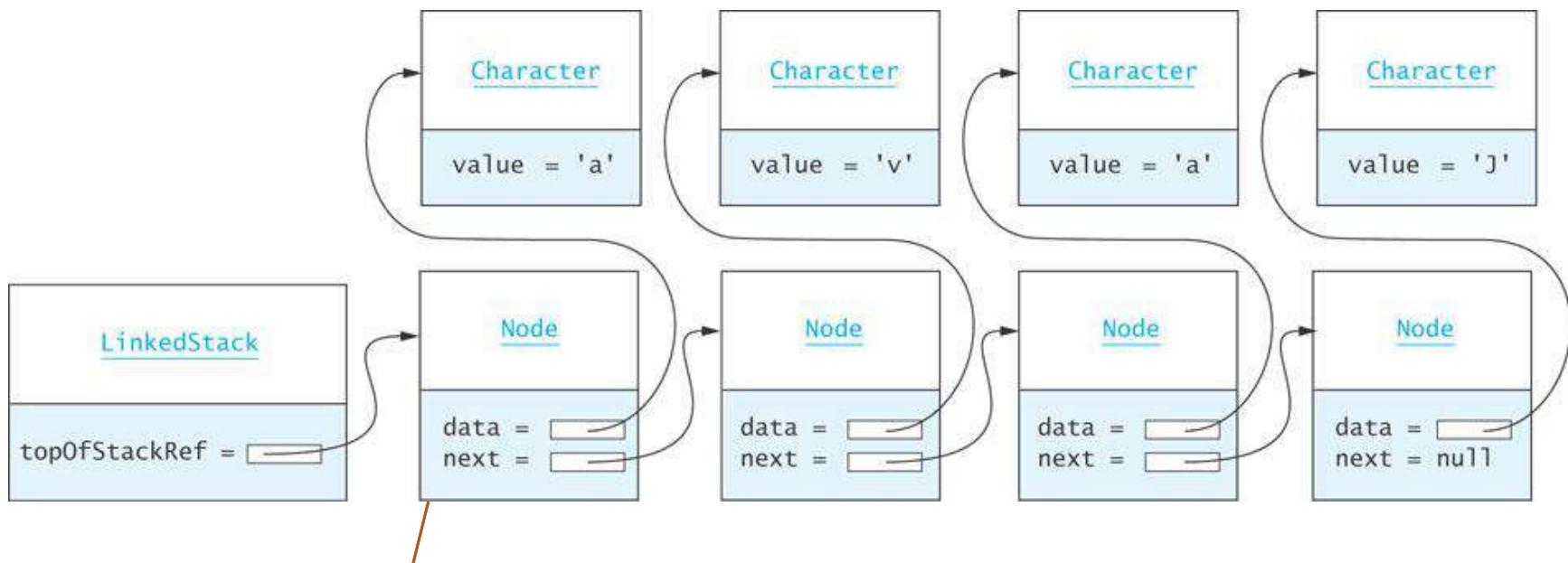
- A class which adapts methods of another class by giving different names to essentially the same methods (`push` instead of `add`) is called an *adapter class*
- Writing methods in this way is called *method delegation*

Implementing a Stack Using an Array (cont.)



Implementing a Stack as a Linked Data Structure

- We can also implement a stack using a linked list of nodes



when the list is empty, pop
returns null

Additional Stack Applications

- Postfix and infix notation
 - Expressions normally are written in infix form, but
 - it easier to evaluate an expression in postfix form since there is no need to group sub-expressions in parentheses or worry about operator precedence

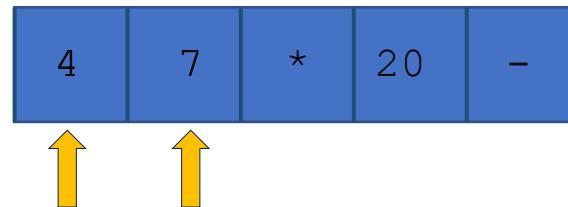
Postfix Expression	Infix Expression	Value
<u>4</u> <u>7</u> <u>*</u>	<u>4</u> * <u>7</u>	28
<u>4</u> <u>7</u> <u>2</u> <u>+</u> <u>*</u>	<u>4</u> * (<u>7</u> + <u>2</u>)	36
<u>4</u> <u>7</u> <u>*</u> <u>20</u> <u>-</u>	(<u>4</u> * <u>7</u>) - <u>20</u>	8
<u>3</u> <u>4</u> <u>7</u> <u>*</u> <u>2</u> <u>/</u> <u>+</u>	<u>3</u> + ((<u>4</u> * <u>7</u>) / <u>2</u>)	17

Evaluating Postfix Expressions (cont.)



- ➡ 1. create an empty stack of integers
- ➡ 2. while there are more tokens
- ➡ 3. get the next token
- ➡ 4. if the first character of the token is a digit
- ➡ 5. push the token on the stack
- 6. else if the token is an operator
- 7. pop the right operand off the stack
- 8. pop the left operand off the stack
- 9. evaluate the operation
- 10. push the result onto the stack
- 11. pop the stack and return the result

Evaluating Postfix Expressions (cont.)

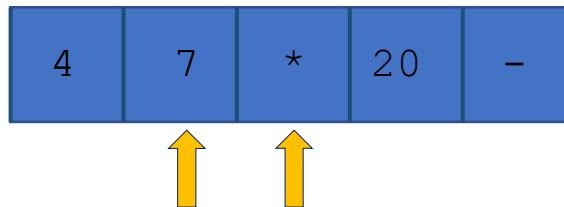


1. create an empty stack of integers
2. while there are more tokens
 3. get the next token
 4. if the first character of the token is a digit
 5. push the token on the stack
 6. else if the token is an operator
 7. pop the right operand off the stack
 8. pop the left operand off the stack
 9. evaluate the operation
 10. push the result onto the stack
 11. pop the stack and return the result

Evaluating Postfix Expressions (cont.)



4 * 7

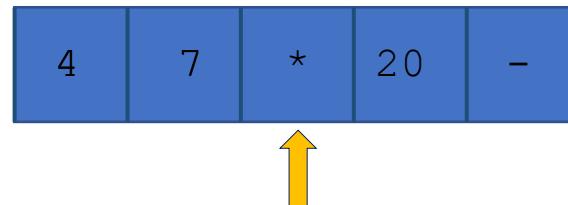


1. create an empty stack of integers
2. while there are more tokens
 3. get the next token
 4. if the first character of the token is a digit
 5. push the token on the stack
 6. else if the token is an operator
 7. pop the right operand off the stack
 8. pop the left operand off the stack
 9. evaluate the operation
 10. push the result onto the stack
 11. pop the stack and return the result

Evaluating Postfix Expressions (cont.)

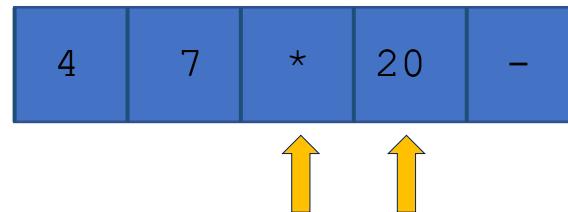
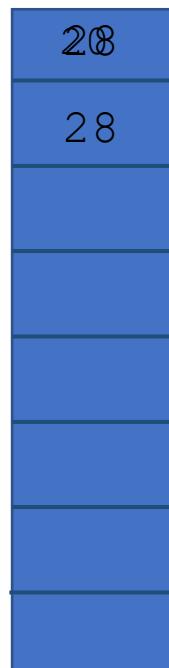


28



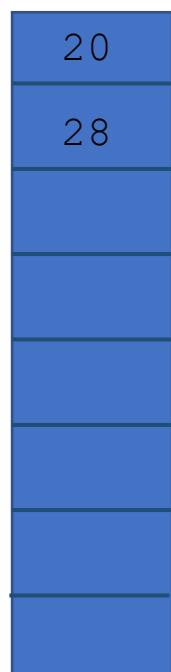
1. create an empty stack of integers
2. while there are more tokens
3. get the next token
4. if the first character of the token is a digit
5. push the token on the stack
6. else if the token is an operator
7. pop the right operand off the stack
8. pop the left operand off the stack
9. evaluate the operation
 ➡
10. push the result onto the stack
 ➡
11. pop the stack and return the result

Evaluating Postfix Expressions (cont.)



1. create an empty stack of integers
2. while there are more tokens
 3. get the next token
 4. if the first character of the token is a digit
 5. push the token on the stack
 6. else if the token is an operator
 7. pop the right operand off the stack
 8. pop the left operand off the stack
 9. evaluate the operation
 10. push the result onto the stack
 11. pop the stack and return the result

Evaluating Postfix Expressions (cont.)



28 - 20



1. create an empty stack of integers
2. while there are more tokens
 3. get the next token
 4. if the first character of the token is a digit
 5. push the token on the stack
 6. else if the token is an operator
 7. pop the right operand off the stack
 8. pop the left operand off the stack
 9. evaluate the operation
 10. push the result onto the stack
 11. pop the stack and return the result

Evaluating Postfix Expressions (cont.)



8



1. create an empty stack of integers
2. while there are more tokens
3. get the next token
4. if the first character of the token is a digit
5. push the token on the stack
6. else if the token is an operator
7. pop the right operand off the stack
8. pop the left operand off the stack
9. evaluate the operation
 ➡
10. push the result onto the stack
 ➡
11. pop the stack and return the result

Evaluating Postfix Expressions (cont.)



1. create an empty stack of integers
2. while there are more tokens
 3. get the next token
 4. if the first character of the token is a digit
 5. push the token on the stack
 6. else if the token is an operator
 7. pop the right operand off the stack
 8. pop the left operand off the stack
 9. evaluate the operation
 10. push the result onto the stack
11. pop the stack and return the result

Queue ADT

- The queue, like the stack, is a widely used data structure
- A queue differs from a stack in one important way
 - A stack is LIFO list – *Last-In, First-Out*
 - while a queue is FIFO list, *First-In, First-Out*

Queue Abstract Data Type

- A queue can be visualized as a line of customers waiting for service
- The next person to be served is the one who has waited the longest
- New elements are placed at the end of the line



Print Queue

- Operating systems use queues to
 - keep track of tasks waiting for a scarce resource
 - ensure that the tasks are carried out in the order they were generated
- Print queue: printing is much slower than the process of selecting pages to print, so a queue is used

The screenshot shows a Windows-style application window titled "HP LaserJet 4050 Series PS - Use Printer Offline". The menu bar includes "Printer", "Document", "View", and "Help". The main area is a table displaying the following data:

Document Name	Status	Owner	Pages	Size	Submitted	P
Microsoft Word - Queues_Paul_1007.doc	Paul Wolfgang	52	9.75 MB	1:53:18 PM	10/7/2003	
Microsoft Word - Stacks.doc	Paul Wolfgang	46	9.05 MB	1:53:57 PM	10/7/2003	
Microsoft Word - Trees2.doc	Paul Wolfgang	54	38.4 MB	1:54:41 PM	10/7/2003	

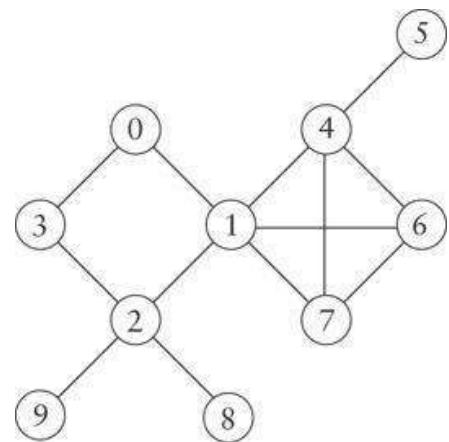
At the bottom, a status bar indicates "3 document(s) in queue".

Unsuitability of a Print Stack

- Stacks are Last-In, First-Out (LIFO)
- The most recently selected document would be the next to print
- Unless the printer stack is empty, your print job may never be executed if others are issuing print jobs

Using a Queue for Traversing a Multi-Branch Data Structure

- A graph models a network of nodes, with links connecting nodes to other nodes in the network
- A node in a graph may have several neighbors
- Programmers doing a *breadth-first traversal* often use a queue to ensure that nodes closer to the starting point are visited before nodes that are farther away
- You will learn more about graph traversal later



Class LinkedList Implements the Queue Interface

- The `LinkedList` class provides methods for inserting and removing elements at either end of a double-linked list, which means all `Queue` methods can be implemented easily
- The Java 5.0 `LinkedList` class implements the `Queue` interface

```
Queue<String> names = new LinkedList<String>();
```

- creates a new `Queue` reference, `names`, that stores references to `String` objects
- The actual object referenced by `names` is of type `LinkedList<String>`, but because `names` is a type `Queue<String>` reference, you can apply only the `Queue` methods to it

Maintaining a Queue of Customers

- Write a menu-driven program that maintains a list of customers
- The user should be able to:
 - insert a new customer in line
 - display the customer who is next in line
 - remove the customer who is next in line
 - display the length of the line
 - determine how many people are ahead of a specified person

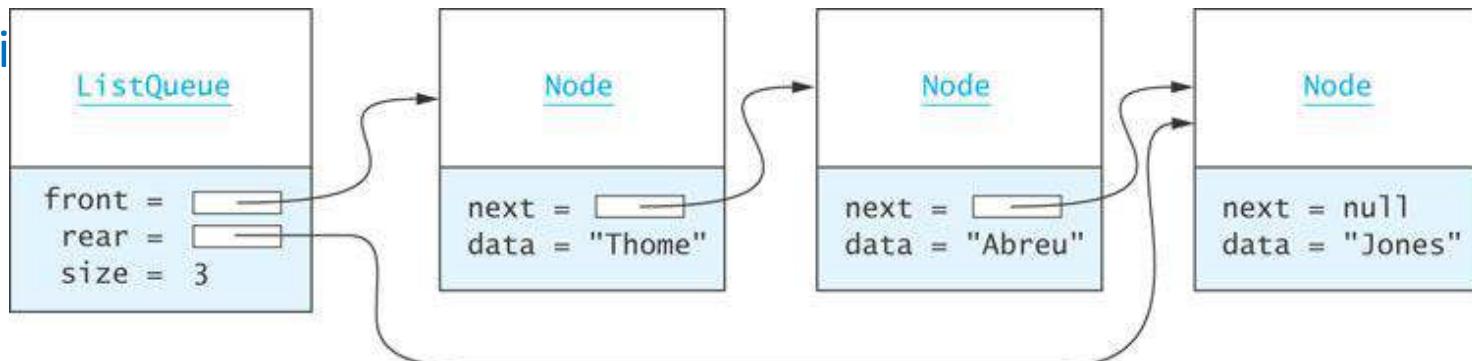
Using a Double-Linked List to Implement the Queue Interface

- Insertion and removal from either end of a double-linked list is $O(1)$ so either end can be the front (or rear) of the queue
- Java designers decided to make the head of the linked list the front of the queue and the tail the rear of the queue
- Problem: If a `LinkedList` object is used as a queue, it will be possible to apply other `LinkedList` methods in addition to the ones required and permitted by the `Queue` interface
- Solution: Create a new class with a `LinkedList` component and then code (by delegation to the `LinkedList` class) only the public methods required by the `Queue` interface

Using a Single-Linked List to Implement a Queue

- Insertions are at the rear of a queue and removals are from the front
- We need a reference to the last list node so that insertions can be performed at O(1)
- The number of elements in the queue is changed by methods insert and remove

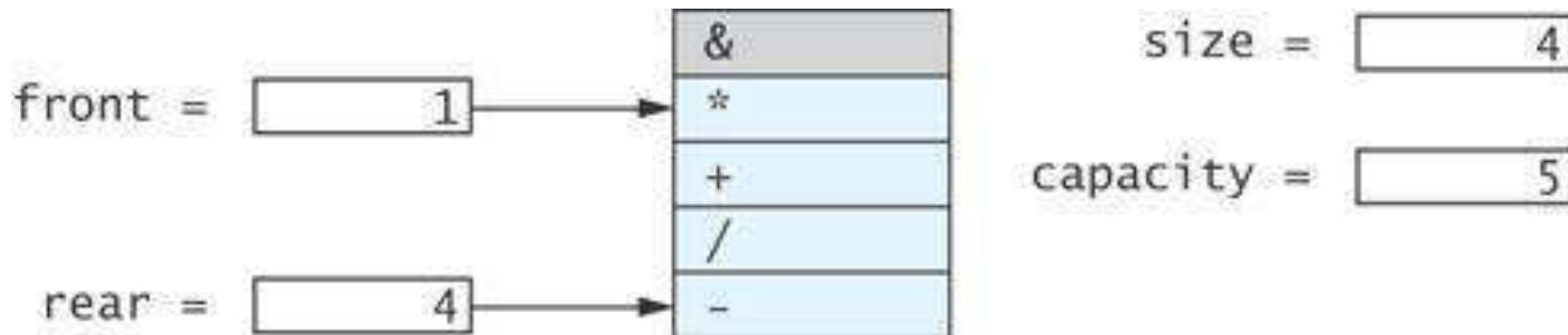
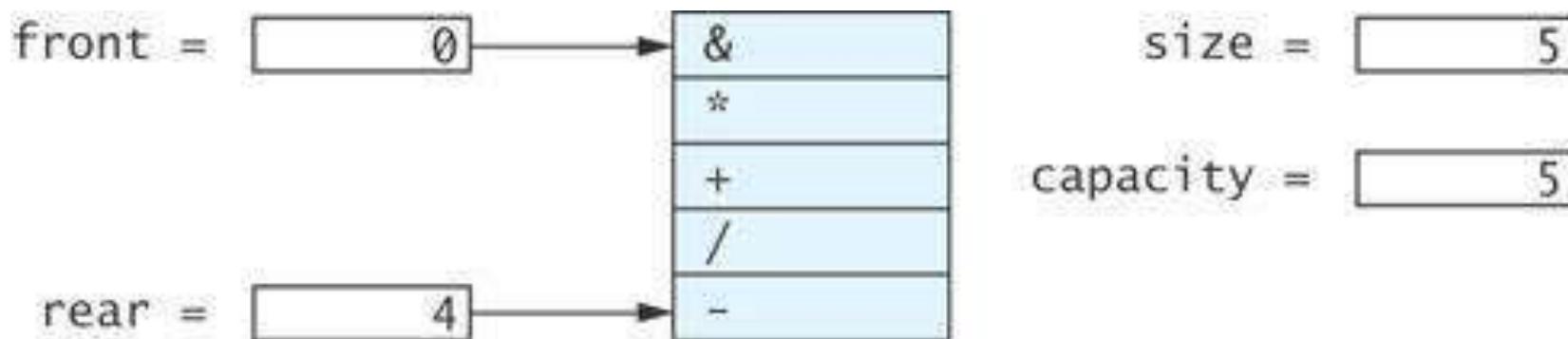
- Listi



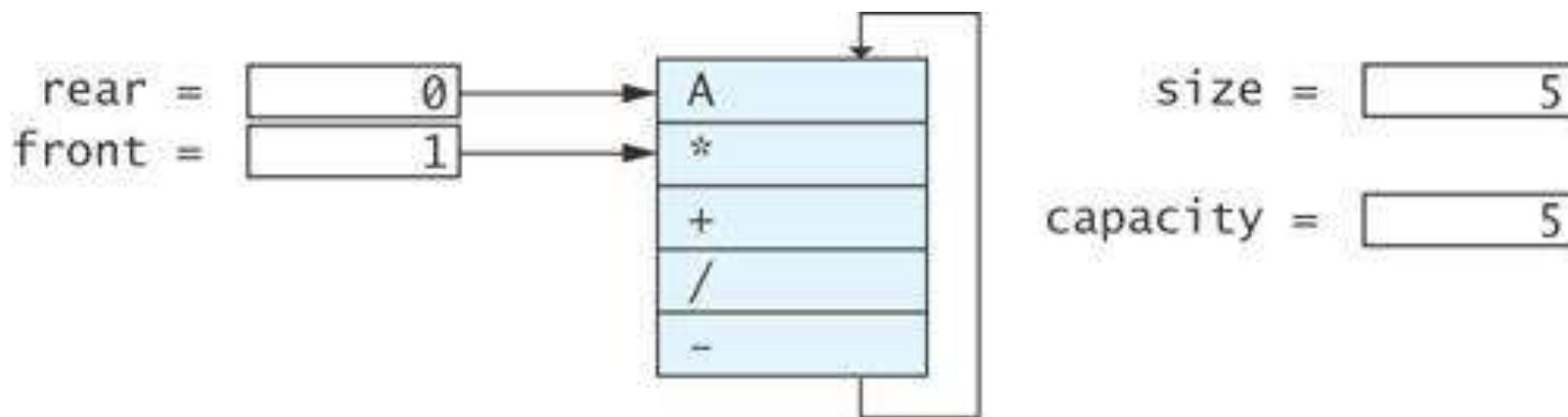
Implementing a Queue Using a Circular Array

- The time efficiency of using a single- or double-linked list to implement a queue is acceptable
- However, there are some space inefficiencies
- Storage space is increased when using a linked list due to references stored in the nodes
- Array Implementation
 - Insertion at rear of array is constant time $O(1)$
 - Removal from the front is linear time $O(n)$
 - Removal from rear of array is constant time $O(1)$
 - Insertion at the front is linear time $O(n)$
- We now discuss how to avoid these inefficiencies in an array

Implementing a Queue Using a Circular Array (cont.)



Implementing a Queue Using a Circular Array (cont.)



Comparing the Three Implementations

- Linked list
- Doubly linked list
- Circular array
(we do not consider a normal array since this is inefficient)
- Computation time
 - All three implementations are comparable in terms of computation time
 - All operations are $O(1)$ regardless of implementation

Comparing the Three Implementations (cont.)

- Storage
 - Linked-list implementations require more storage due to the extra space required for the links
 - Each node for a single-linked list stores two references (one for the data, one for the link)
 - Each node for a double-linked list stores three references (one for the data, two for the links)
 - A double-linked list requires 1.5 times the storage of a single-linked list
 - A circular array that is filled to capacity requires half the storage of a single-linked list to store the same number of elements,
 - but a recently reallocated circular array is half empty, and requires the same storage as a single-linked list

Simulating Waiting Lines Using Queues

- *Simulation* is used to study the performance of a physical system by using a physical, mathematical, or computer model of the system
- Simulation allows designers of a new system to estimate the expected performance before building it
- Simulation can lead to changes in the design that will improve the expected performance of the new system
- Simulation is useful when the real system would be too expensive to build or too dangerous to experiment with after its construction

Simulating Waiting Lines Using Queues

(cont.)

- System designers often use computer models to simulate physical systems
 - Example: an airline check-in counter
- A branch of mathematics called *queuing theory* studies such problems

ADS

Introduction to Trees

Agenda

- Trees as data structures
- Tree terminology
- Tree implementations
- Analyzing tree efficiency
- Tree traversals
- Expression trees

Trees

- A *tree* is a non-linear structure in which elements are organized into a hierarchy
- A tree is comprised of a set of *nodes* in which elements are stored and *edges* connect one node to another
- Each node is located on a particular *level*
- There is only one *root* node in the tree

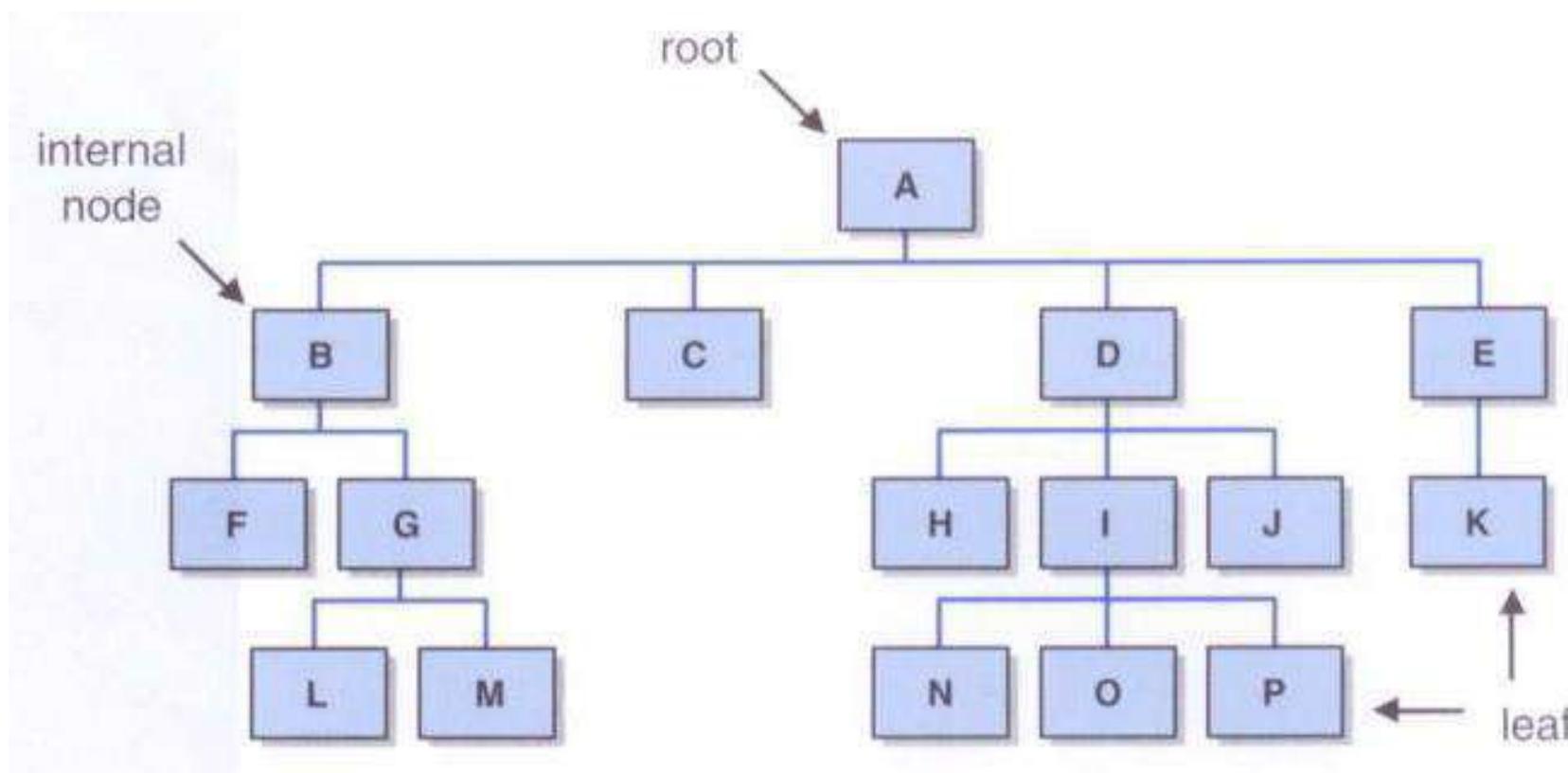
Trees

- Nodes at the lower level of a tree are the *children* of nodes at the previous level
- A node can have only one *parent*, but may have multiple children
- Nodes that have the same parent are *siblings*
- The root is the only node which has no parent

Trees

- A node that has no children is a *leaf* node
- A note that is not the root and has at least one child is an *internal node*
- A *subtree* is a tree structure that makes up part of another tree
- We can follow a *path* through a tree from parent to child, starting at the root
- A node is an *ancestor* of another node if it is above it on the path from the root.

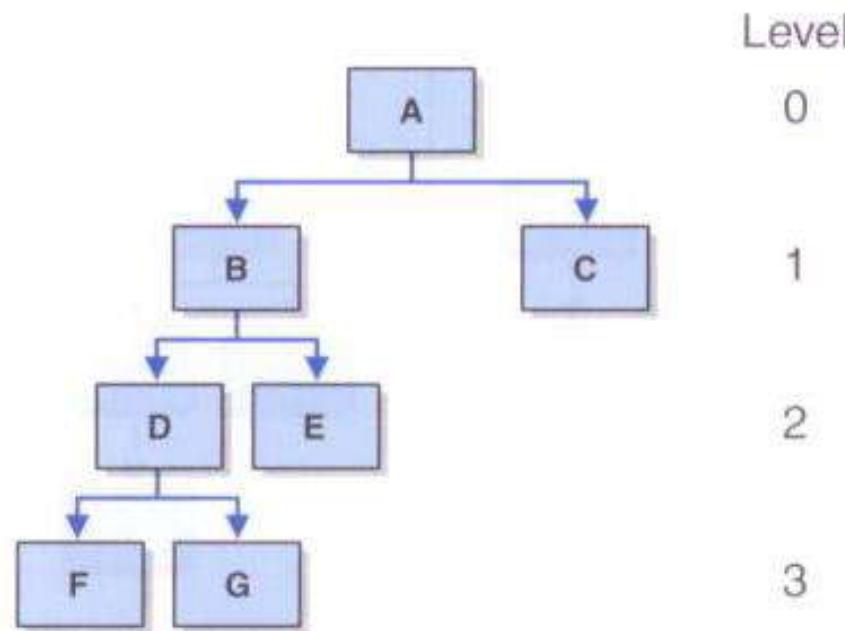
Trees



Trees

- Nodes that can be reached by following a path from a particular node are the *descendants* of that node
- The *level* of a node is the length of the path from the root to the node
- The *path length* is the number of edges followed to get from the root to the node
- The *height* of a tree is the length of the longest path from the root to a leaf

Trees

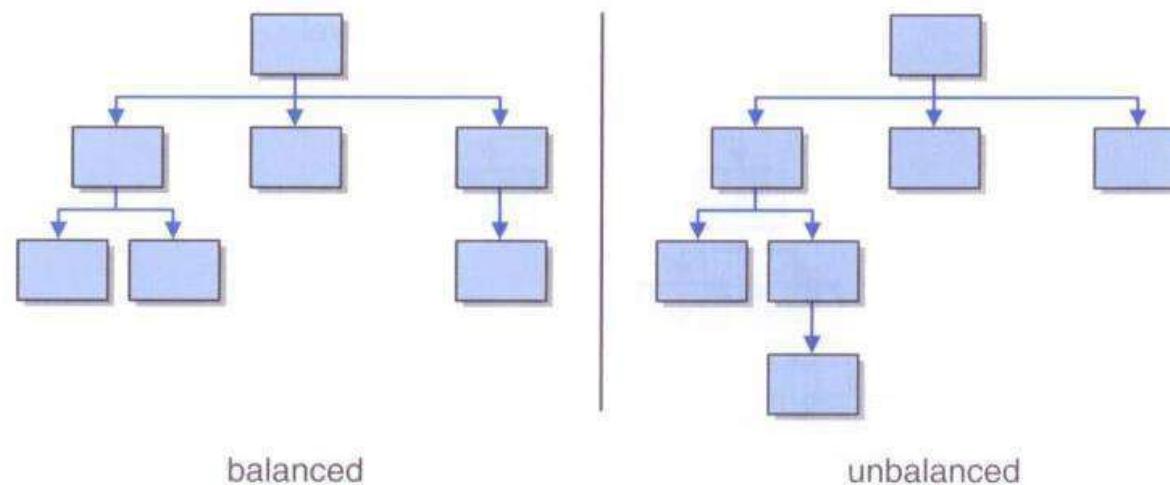


Classifying Trees

- Trees can be classified in many ways
- One important criterion is the maximum number of children any node in the tree may have
- This may be referred to as the *order of the tree*
- *General trees* have no limit to the number of children a node may have
- A tree that limits each node to no more than n children is referred to as an *n-ary tree*

Balanced Trees

- Trees in which nodes may have at most two children are called *binary trees*
- A tree is *balanced* if all of the leaves of the tree are on the same level or within one level of each other

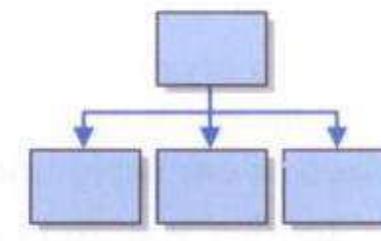
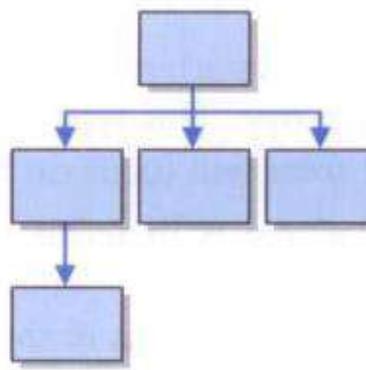
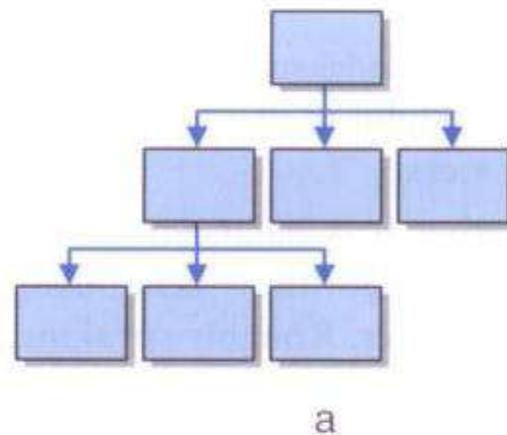


Full and Complete Trees

- A balanced n-ary tree with m elements will have a height of $\log_n m$
- A balanced binary tree with n nodes has a height of $\log_2 n$
- An n-ary tree is *full* if all leaves of the tree are at the same height and every non-leaf node has exactly n children
- A tree is *complete* if it is full, or full to the next-to-last level with all leaves at the bottom level on the left side of the tree

Full and Complete Trees

- Three complete trees:



- Only tree c is full

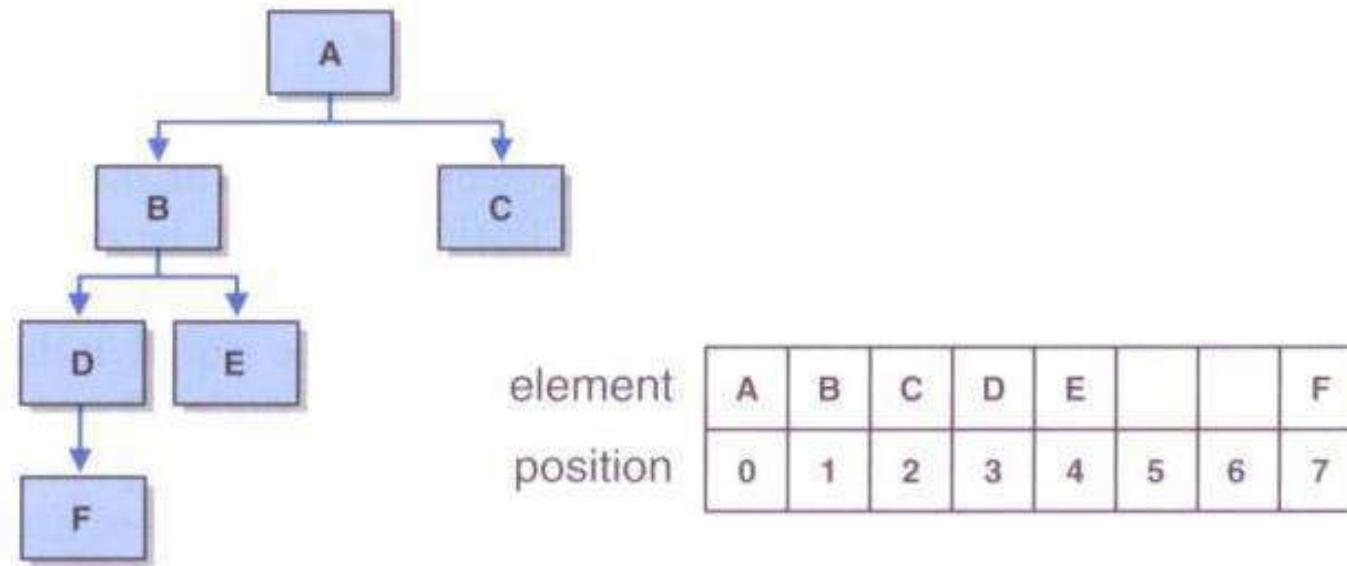
Implementing Trees: linked structure or array

- An obvious choice for implementing trees is a linked structure
- Exactly like a linked list except there are n next pointers
- Array-based implementations are the less obvious choice, but are sometimes useful

Computed Child Links

- For full or complete binary trees, we can use an array to represent a tree
- For any element stored in position n ,
 - the element's left child is stored in array position $(2n+1)$
 - the element's right child is stored in array position $(2*(n+1))$
- If the represented tree is not complete or relatively complete, this approach can waste large amounts of array space

Computed Child Links



Implementing Binary Trees ADT with Links

- A `LinkedBinaryTree` class holds a reference to the root node.
- All operations (public methods) are implemented in `LinkedBinaryTree`.
- A `BinaryTreeNode` class represents each node, with links to a possible left and/or right child.

Tree Traversals

- For linear structures, the process of iterating through the elements is fairly obvious (forwards or backwards)
- For non-linear structures like a tree, the possibilities are more interesting
- Let's look at four classic ways of *traversing* the nodes of a tree
- All traversals start at the root of the tree
- Each node can be thought of as the root of a subtree

Tree Traversals

- *Preorder*: visit the root, then traverse the subtrees from left to right
- *Inorder*: traverse the left subtree, then visit the root, then traverse the right subtree
- *Postorder*: traverse the subtrees from left to right, then visit the root
- *Level-order*: visit each node at each level of the tree from top (root) to bottom and left to right

Tree Traversals

Preorder:

A B D E C

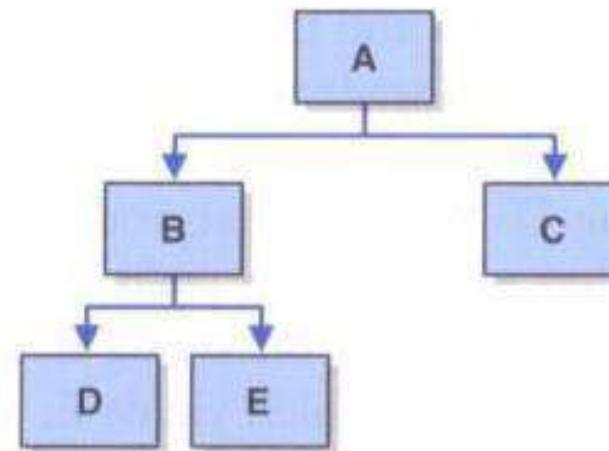
Inorder:

D B E A C

Postorder:

D E B C A

Level-Order: A B C D E



Tree Traversals

- Recursion simplifies the implementation of tree traversals
- Preorder (pseudocode):

```
    Visit node  
    Traverse (left child)  
    Traverse (right child)
```

- Inorder:

```
    Traverse (left child)  
    Visit node  
    Traverse (right child)
```

Tree Traversals

- Postorder:
 - Traverse (left child)
 - Traverse (right child)
 - Visit node
- A level-order traversal is more complicated
- It requires the use of extra structures (such as queues and/or lists) to create the necessary order

A Binary Tree ADT

Operation	Description
getRoot	Returns a reference to the root
isEmpty	Determines whether the tree is empty
size	Returns the number of elements in the tree
contains	Determines if an element is present in the tree
inOrder	Returns an inOrder representation of the tree
preOrder	Returns an preOrder representation of the tree
postOrder	Returns an postOrder representation of the tree
levelOrder	Returns an level Order representation of the tree
height	Returns the height of the tree

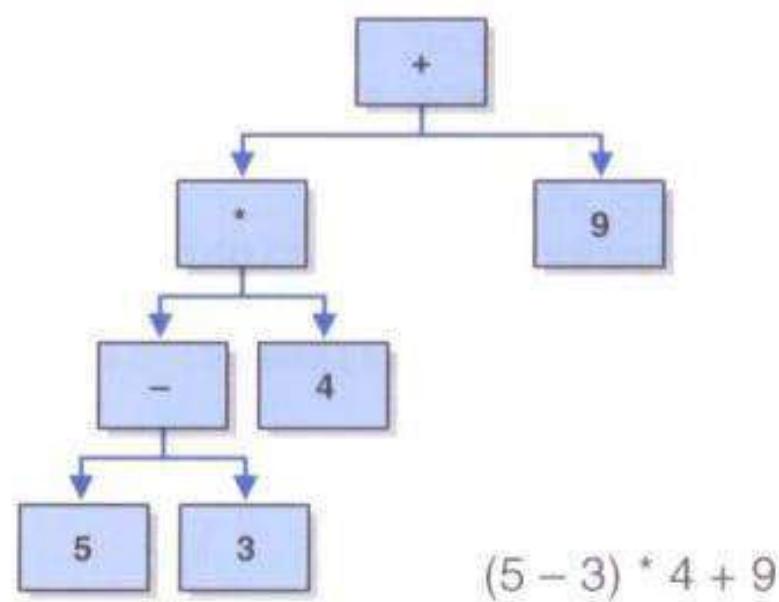
Nb. Notice that there is no add or delete operation.

A Binary Tree Node ADT

Operation	Description
setElement	Store the element in the Node
getElement	Returns the element from the Node
addLeftChild	Add a left child to the Node
addRightChild	Add a right child to the Node
getLeftChild	Returns a reference to the left child
getRightChild	Returns a reference to the right child

Expression Trees

- An *expression tree* is a tree that shows the relationships among operators and operands in an expression
- An expression tree is evaluated from the bottom up



Binary search trees

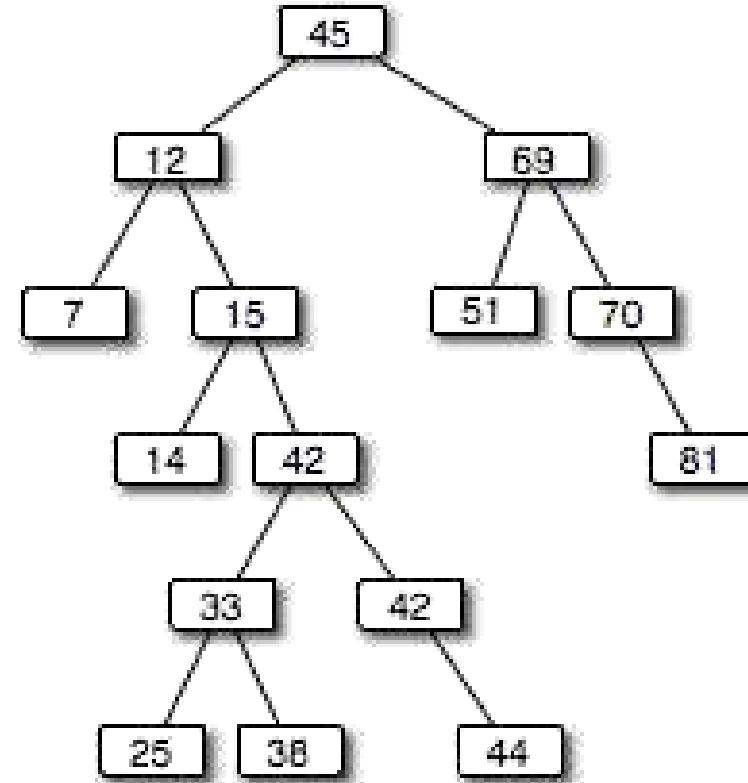
Binary search trees.

- Binary search tree processing
- Using BSTs to solve problems
- BST implementations

Binary Search Trees

- A *search tree* is a tree whose elements are organized to facilitate finding a particular element when needed
- A *binary search tree* is a binary tree that, for each node n
 - the left subtree of n contains elements less than the element stored in n
 - the right subtree of n contains elements greater than or equal to the element stored in n

Binary Search Trees



Binary Search Trees

- To determine if a particular value exists in a tree
 - start at the root
 - compare target to element at current node
 - move left from current node if target is less than element in the current node
 - move right from current node if target is greater than element in the current node
- We eventually find the target or encounter the end of a path (target is not found)

Binary Search Trees

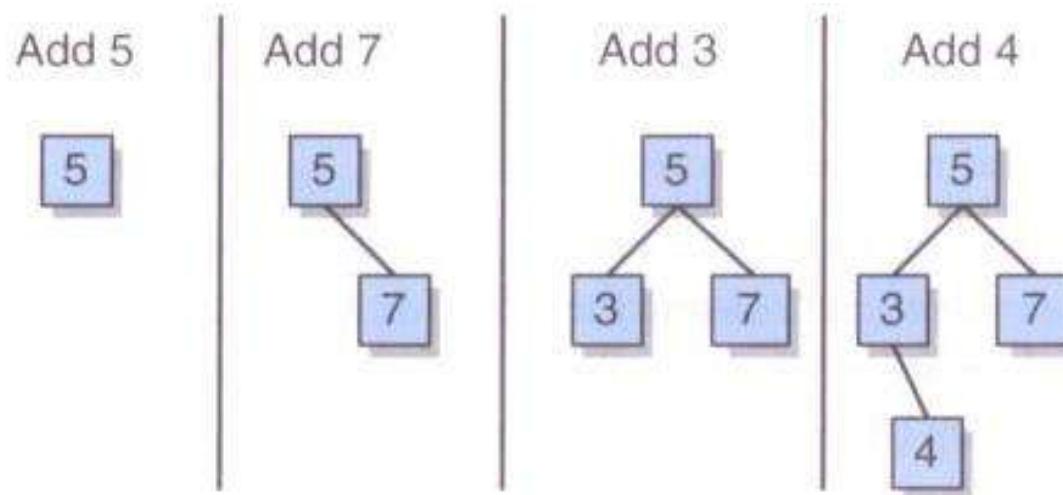
- The particular shape of a binary search tree depends on the order in which the elements are added
- The shape may also be dependant on any additional processing performed on the tree to reshape it
- Binary search trees can hold any type of data, so long as we have a way to determine relative ordering
- Objects implementing the Comparable interface provide such capability

Binary Search Trees

- Process of adding an element is similar to finding an element
- New elements are added as leaf nodes
- Start at the root, follow path dictated by existing elements until you find no child in the desired direction
- Then add the new element

Binary Search Trees

- Adding elements to a BST:

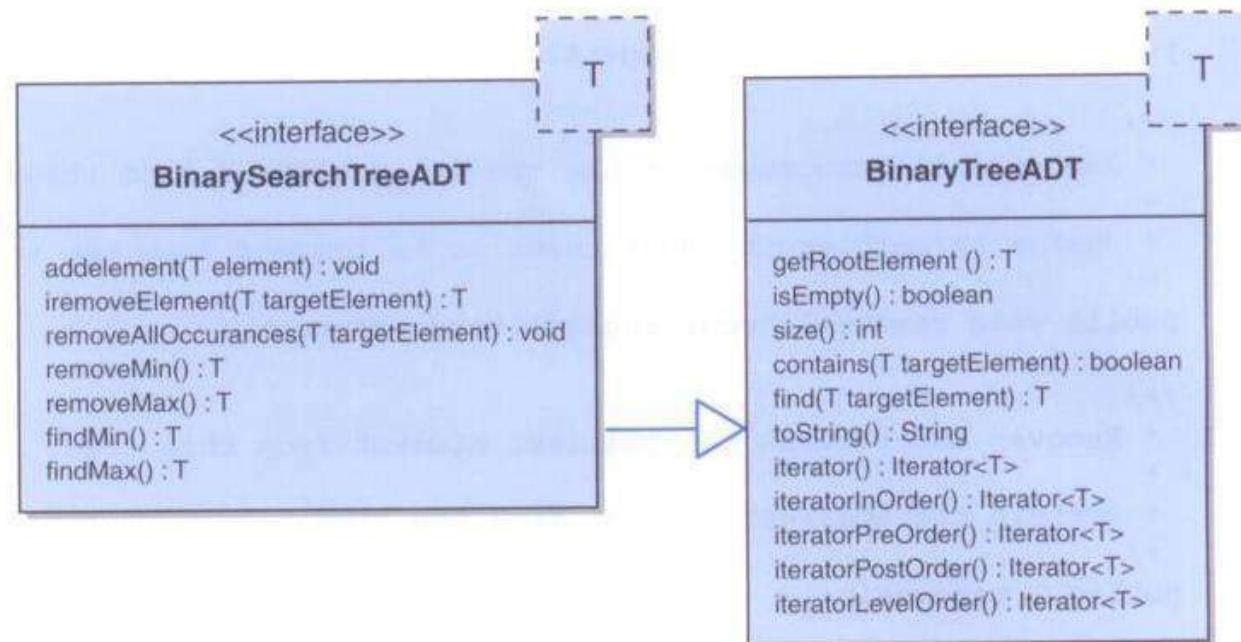


Binary Search Trees

- BST operations:

Operation	Description
addElement	Add an element to the tree.
removeElement	Remove an element from the tree.
removeAllOccurrences	Remove all occurrences of element from the tree.
removeMin	Remove the minimum element in the tree.
removeMax	Remove the maximum element in the tree.
findMin	Returns a reference to the minimum element in the tree.
findMax	Returns a reference to the maximum element in the tree.

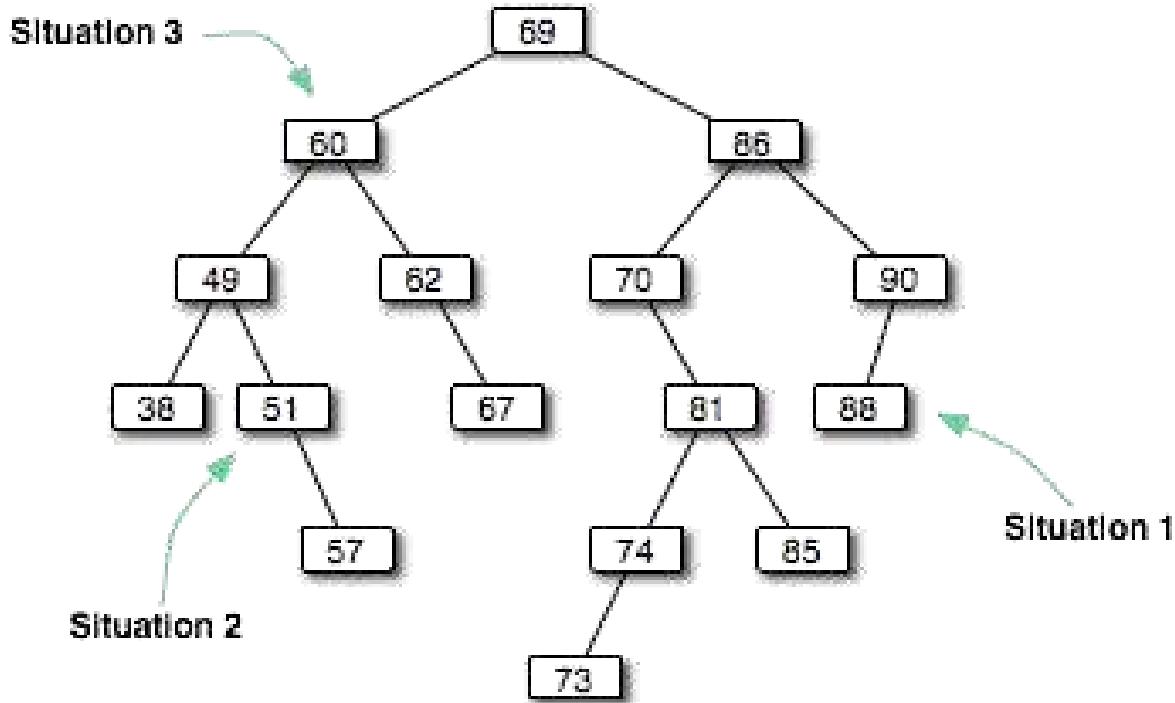
Binary Search Trees



BST Element Removal

- Removing a target in a BST is not as simple as that for linear data structures
- After removing the element, the resulting tree must still be valid
- Three distinct situations must be considered when removing an element
 - The node to remove is a leaf
 - The node to remove has one child
 - The node to remove has two children

BST Element Removal

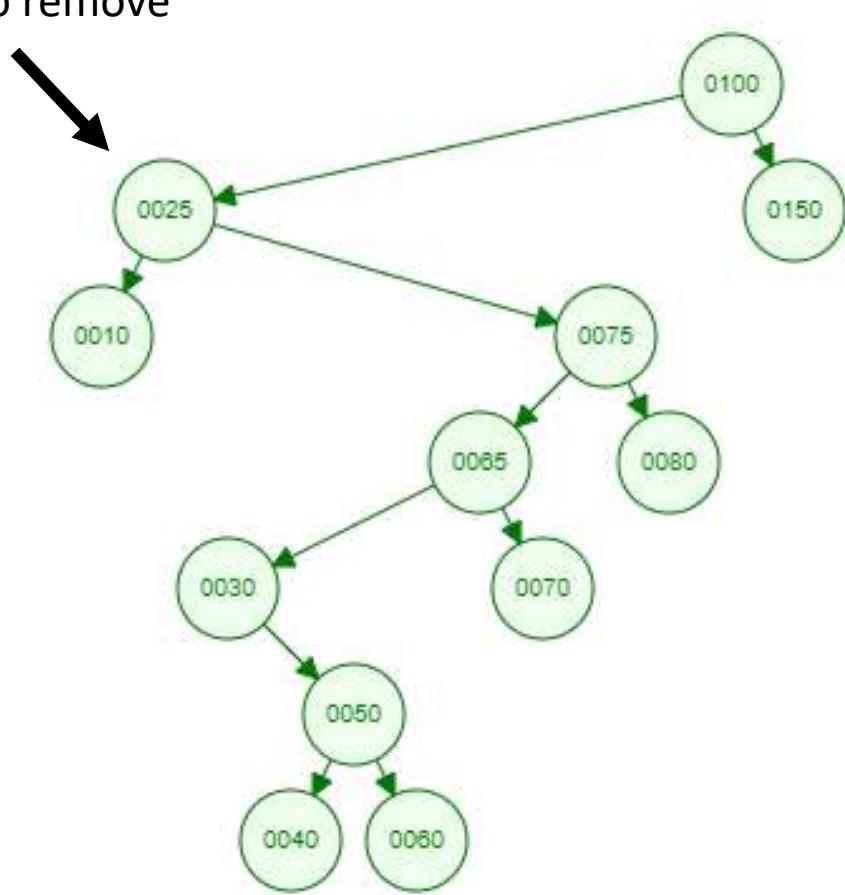


BST Element Removal

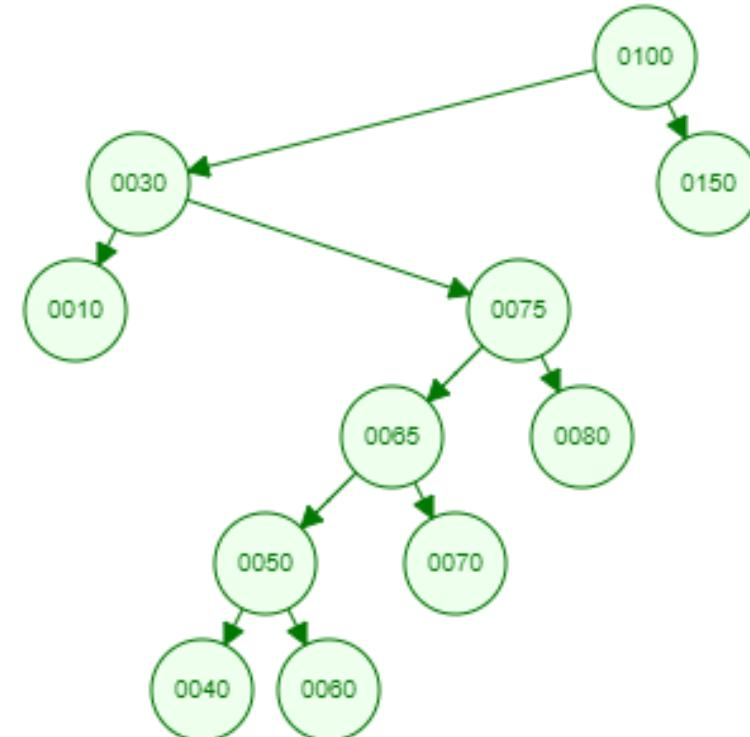
- Dealing with the situations
 - Node is a leaf: it can simply be deleted
 - Node has one child: the deleted node is replaced by the child
 - Node has two children: an appropriate node is found lower in the tree and used to replace the node
 - Good choice: *inorder successor* (node that would follow the removed node in an inorder traversal)
 - The inorder successor is guaranteed not to have a left child
 - Thus, removing the inorder successor to replace the deleted node will result in one of the first two situations (it's a leaf or has one child)

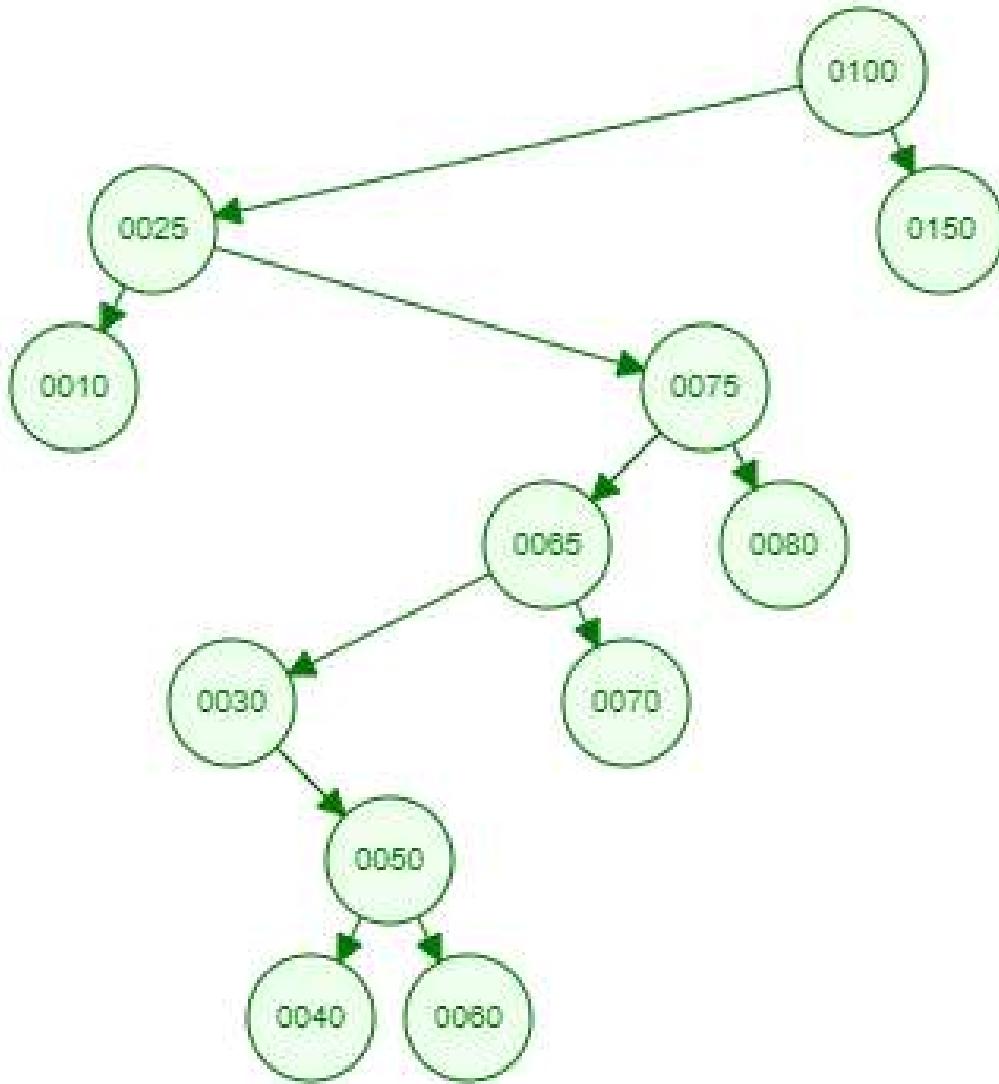
Removing an internal node in a BST:

Node to remove



Tree after removal



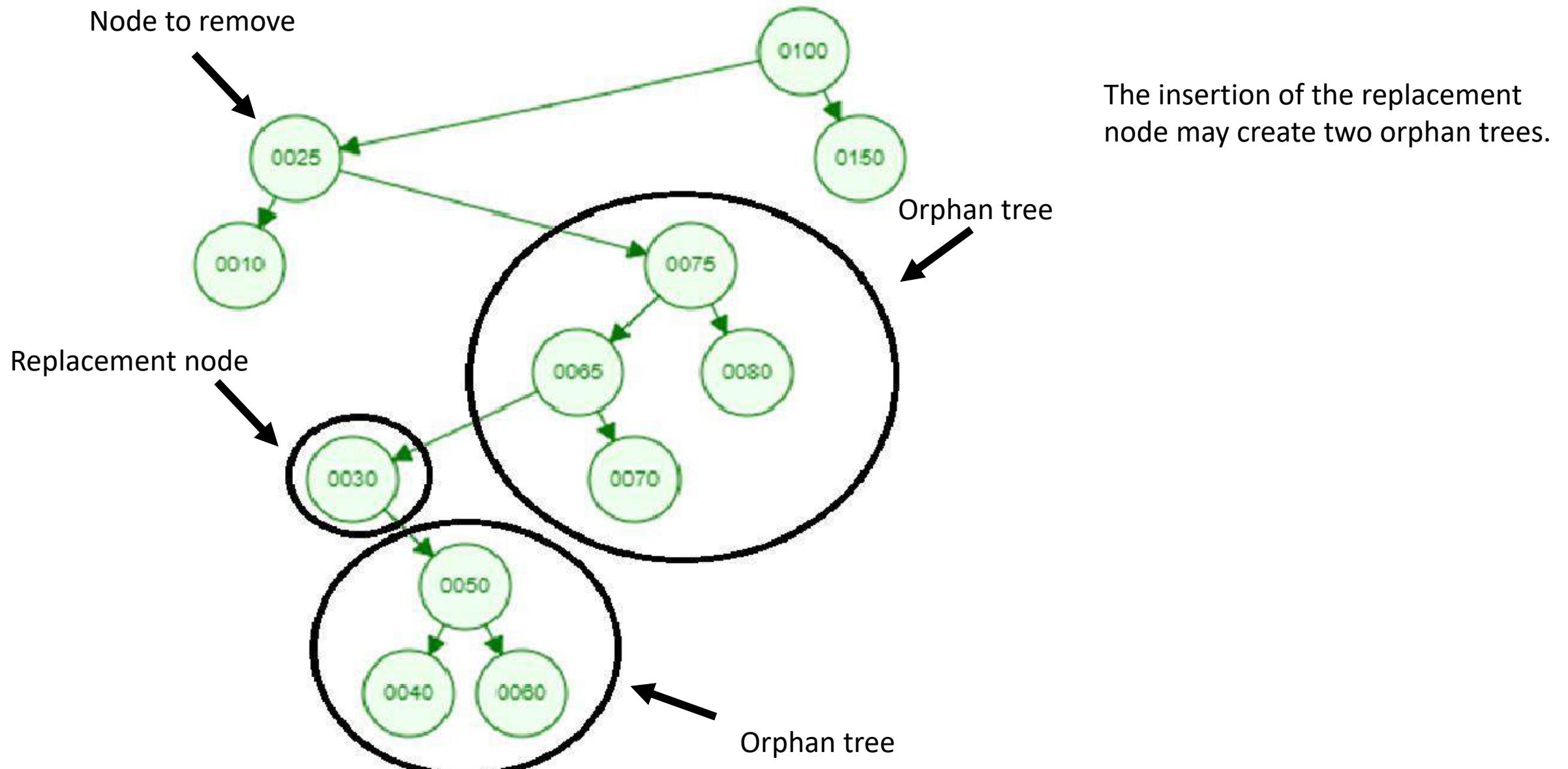


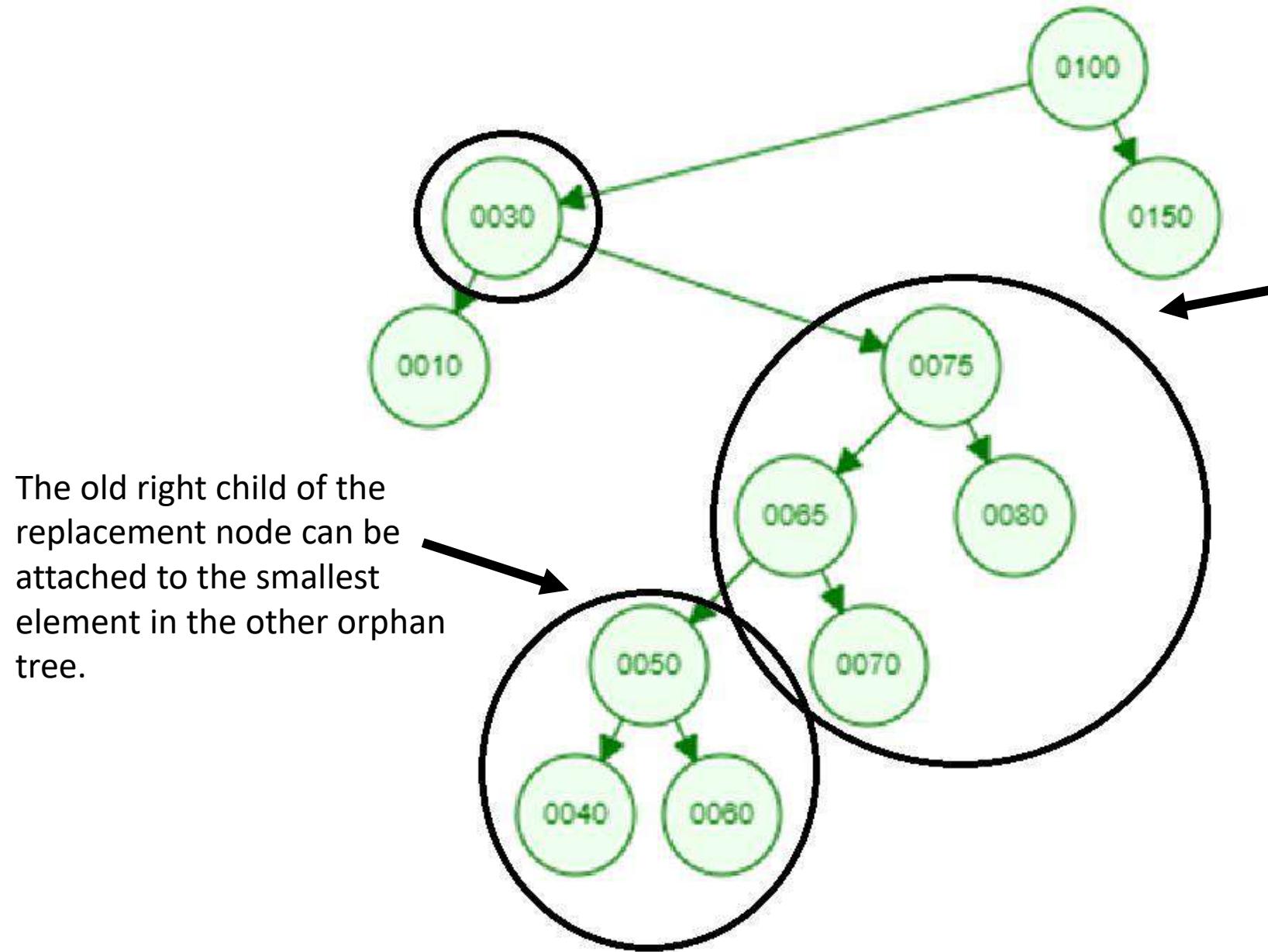
To remove a node in a BST find the inorder successor which is the smallest node in the right subtree. This is equivalent to call **findMin()** on the right child of the node that is to be deleted.

If 25 is to be removed the inorder successor is 30. It can be found by **findMin(75)**

The node to replace the deleted node is called the replacement node (in this case where we want to delete 25 the replacement node is 30).

Alternatively this algorithm can be mirrored so instead the replacement node is the inorder predecessor (the largest node in the left subtree).





The old right child of the replacement node can be attached to the smallest element in the other orphan tree.

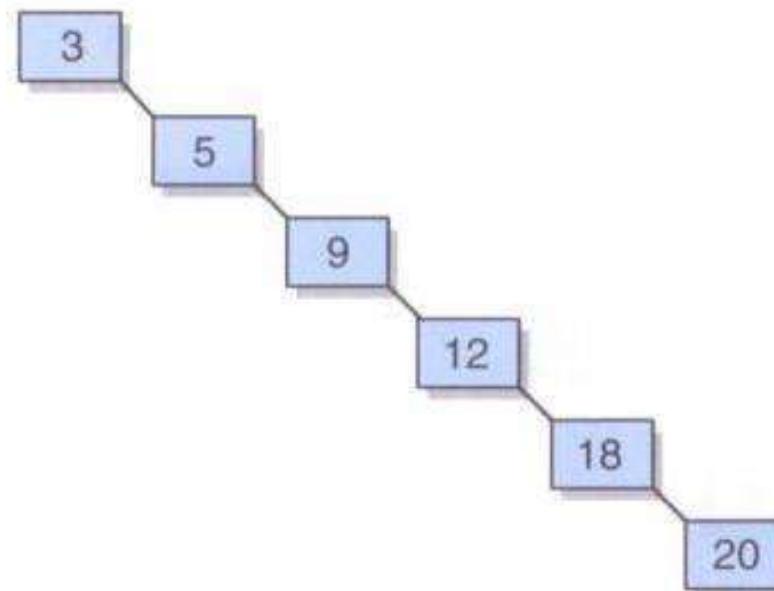
The old right child of the deleted node can be attached to the replacement node as right child.

Balanced binary search trees

AVL and Red/Black trees

Balancing BSTs

- As operations are performed on a BST, it could become highly unbalanced (*a degenerate tree*)



Balancing BSTs

- Our implementation does not ensure the BST stays balanced
- Other approaches do, such as AVL trees, red/black trees and 2-4 trees
- We will explore *rotations* – operations on binary search trees to assist in the process of keeping a tree balanced

Ordered List Analysis

- Comparing operations for both implementations of ordered lists:

Operation	LinkedList	BinarySearchTreeList
removeFirst	O(1)	O(log n)
removeLast	O(n)	O(log n)
remove	O(n)	O(log n)*
first	O(1)	O(log n)
last	O(n)	O(log n)
contains	O(n)	O(log n)
isEmpty	O(1)	O(1)
size	O(1)	O(1)
add	O(n)	O(log n)*

*both the add and remove operations may cause the tree to become unbalanced

Self balancing trees: AVL Trees

- An AVL tree (named after the creators) ensures a BST stays balanced
- For each node in the tree, there is a numeric *balance factor* – the difference between the heights of its subtrees
- After each add or removal, the balance factors are checked, and rotations performed as needed

AVL Tree

- Insert/erase: update balance of each subtree from point of change to the root
- Rotation brings unbalanced tree back into balance
- The height of a tree is the number of nodes in the longest path from the root to a leaf node
 - Height of empty tree is 0:
$$\text{ht(empty)} = 0$$
 - Height of others:
$$\text{ht}(n) = 1 + \max(\text{ht}(n.\text{left}), \text{ht}(n.\text{right}))$$
- Balance(n) = $\text{ht}(n.\text{right}) - \text{ht}(n.\text{left})$

AVL Tree

- The balance of node n = $ht(n.right) - ht(n.left)$
- In an AVL tree, restrict balance to -1, 0, or +1
 - That is, keep nearly balanced at each node

AVL Tree Insertion

- We consider cases where new node is inserted into the *left* subtree of a node n
 - Insertion into right subtree is symmetrical
- **Case 1:** The left subtree height does not increase
 - No action necessary at n
- **Case 2:** Subtree height increases, and $\text{balance}(n) = +1$ or 0
 - Decrement $\text{balance}(n)$ to 0 or -1
- **Case 3:** Subtree height increases, and $\text{balance}(n) = -1$
 - Need more work to obtain balance
(because balance is now -2)

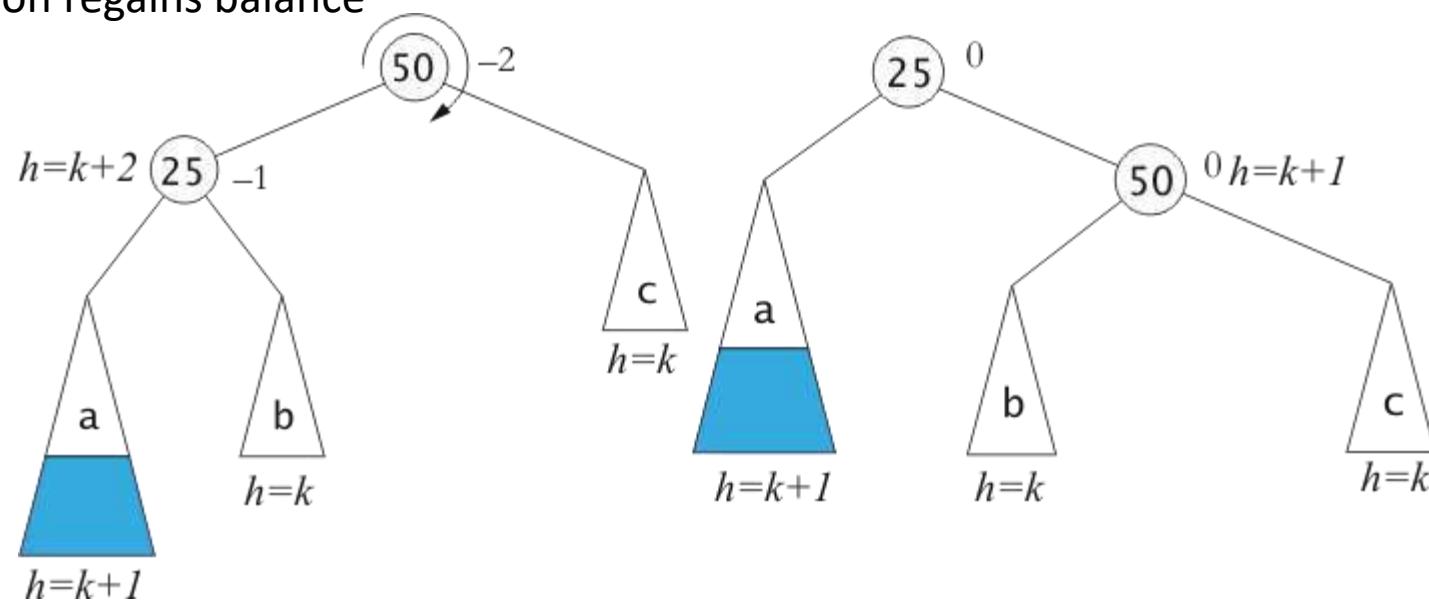
AVL Tree Insertion: Rebalancing

These are the cases:

- **Case 3a:** Left subtree of left child grew:
Left-left heavy tree
- **Case 3b:** Right subtree of left child grew:
Left-right heavy tree

Rebalancing a Left-Left Tree

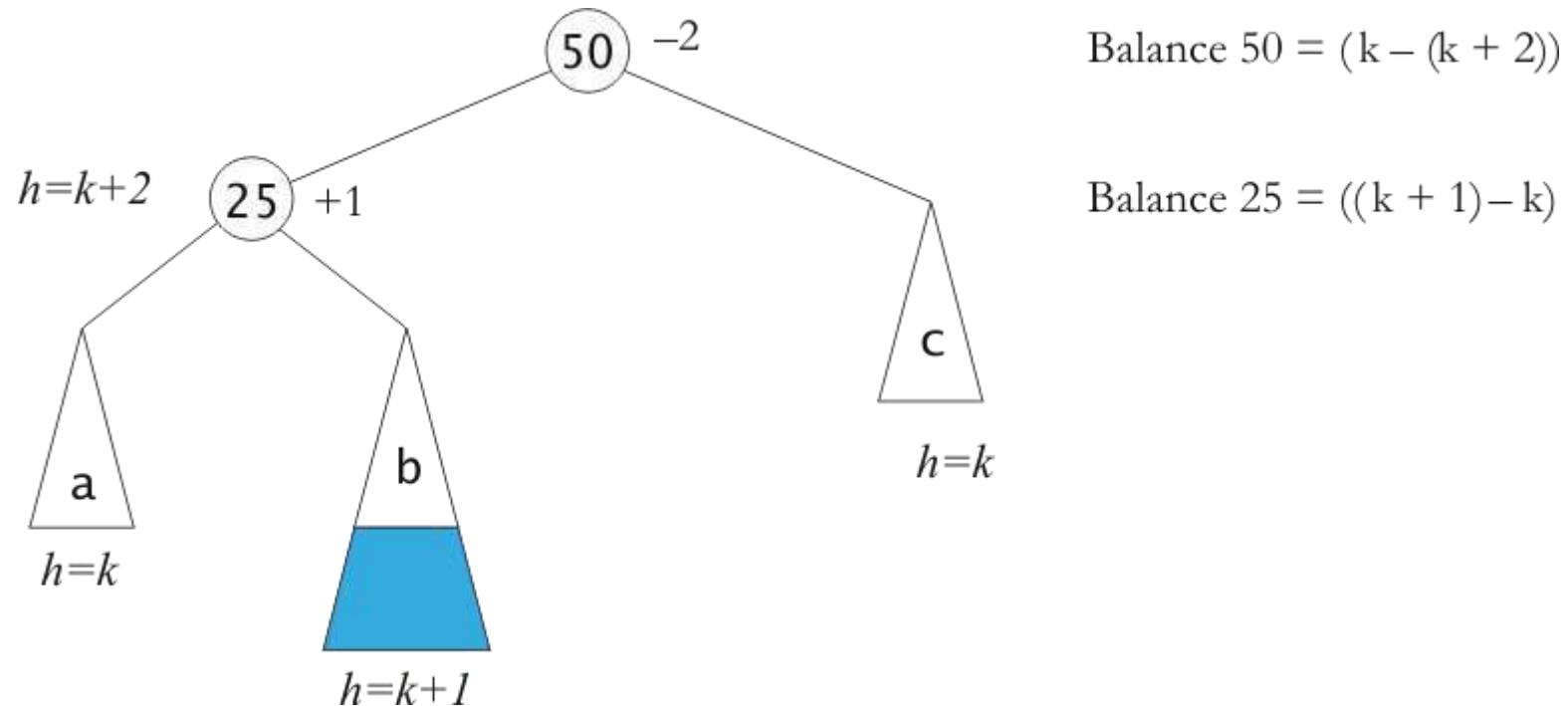
- Actual heights of subtrees are unimportant
 - Only difference in height matters when balancing
- In left-left tree, root and left subtree are left-heavy
- One right rotation regains balance



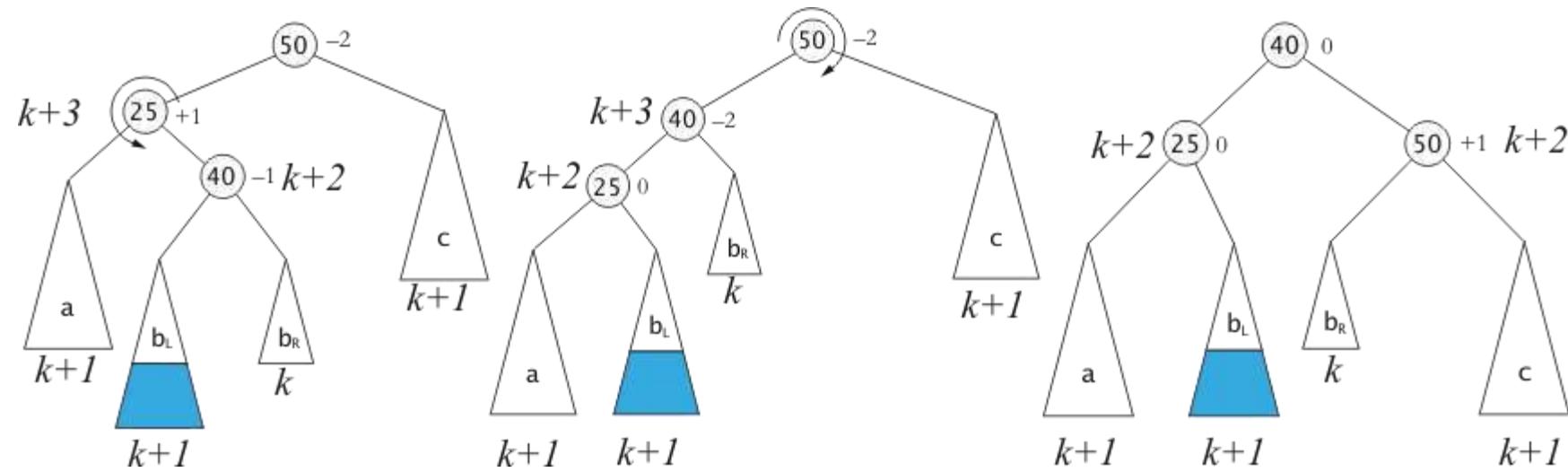
Rebalancing a Left-Right Tree

- Root is left-heavy, left subtree is right-heavy
- A simple right rotation cannot fix this
- Need:
 - Left rotation around child, then
 - Right rotation around root

Rebalancing Left-Right Tree (2)



Rebalancing Left-Right Tree (3)



4 Critically Unbalanced Trees

- Left-Left (parent balance is -2, left child balance is -1)
 - Rotate right around parent
- Left-Right (parent balance -2, left child balance +1)
 - Rotate left around child
 - Rotate right around parent
- Right-Right (parent balance +2, right child balance +1)
 - Rotate left around parent
- Right-Left (parent balance +2, right child balance -1)
 - Rotate right around child
 - Rotate left around parent

Removing in an AVL tree

Leaf and single parent: check from removing point and up

No balance is changed below the inserting point but could be changed upwards (at least the parent of the deleted node will always change balance).

Follow the ancestors:

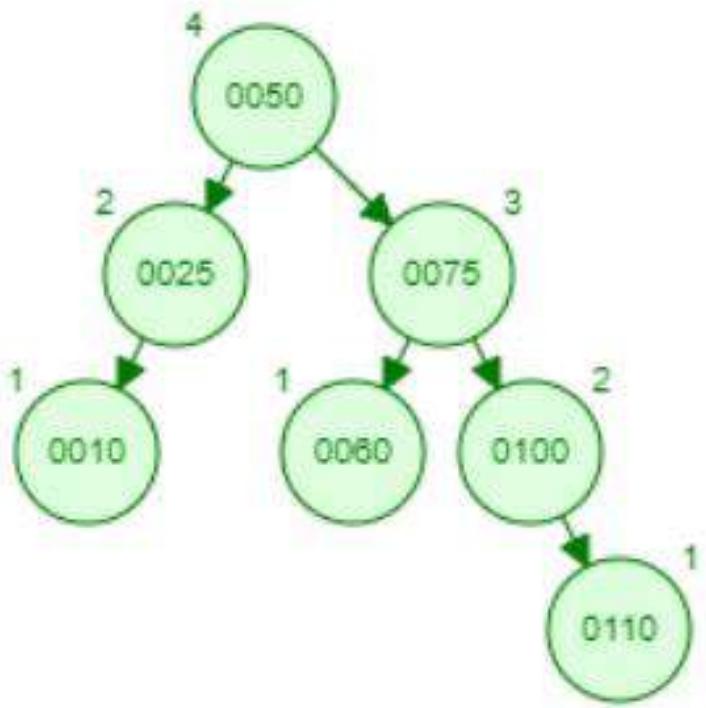
If the balance does not change: stop

If the balance changes:

and is -1, 0 or 1: continue to parent (if root stop)

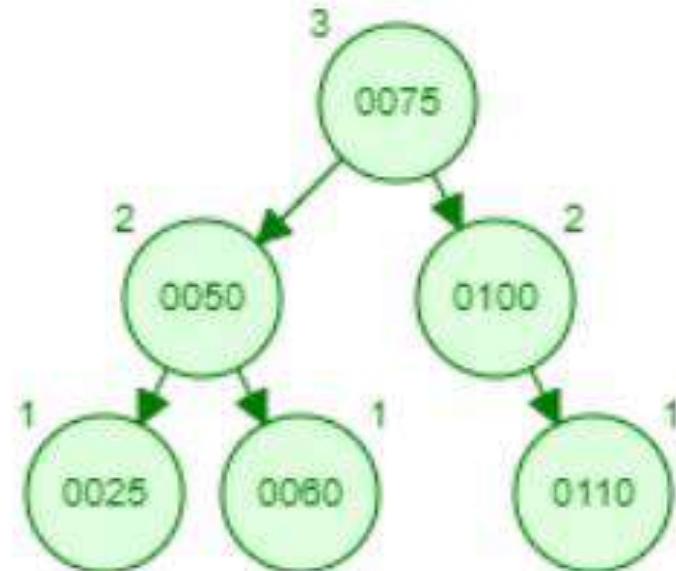
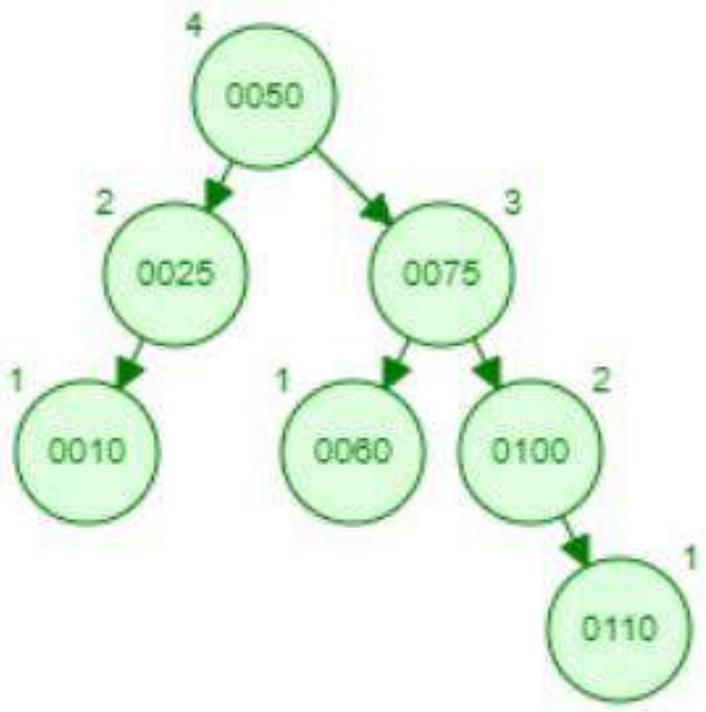
and is -2 or 2: rotate and stop

Remove a leaf or single parent is simple:
check up from the point of deletion.



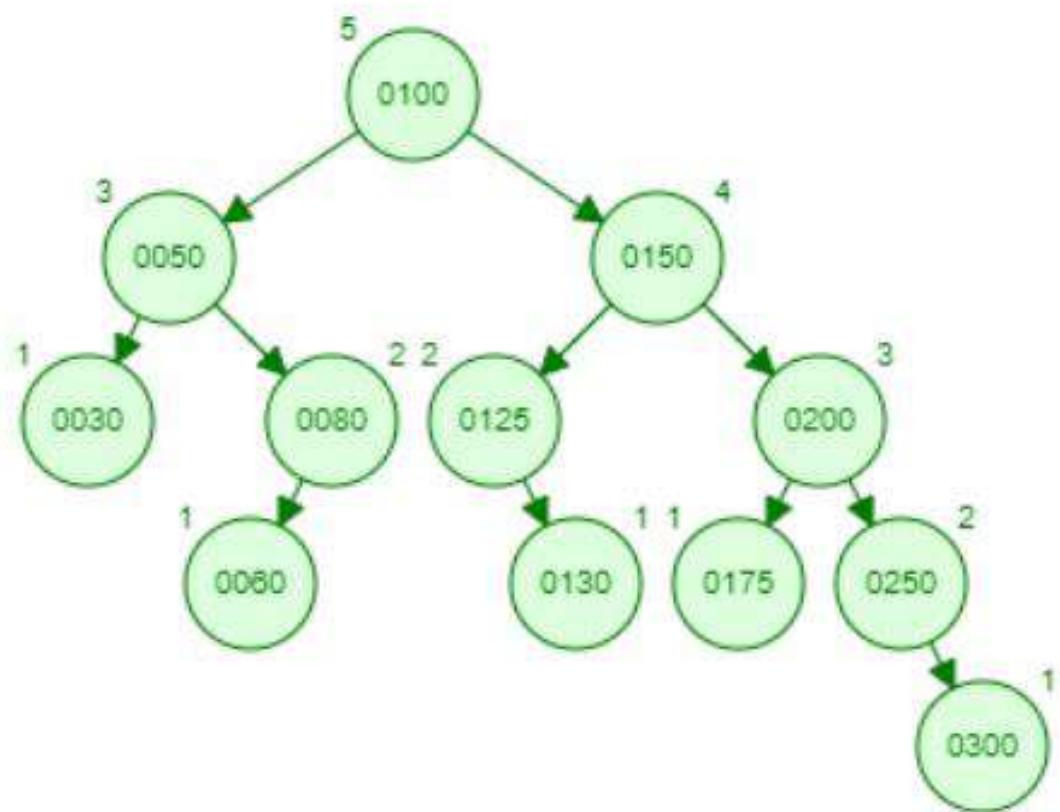
Removing 10 will cause
the tree to be right heavy

Remove a leaf or single parent is simple:
check up from the point of deletion.

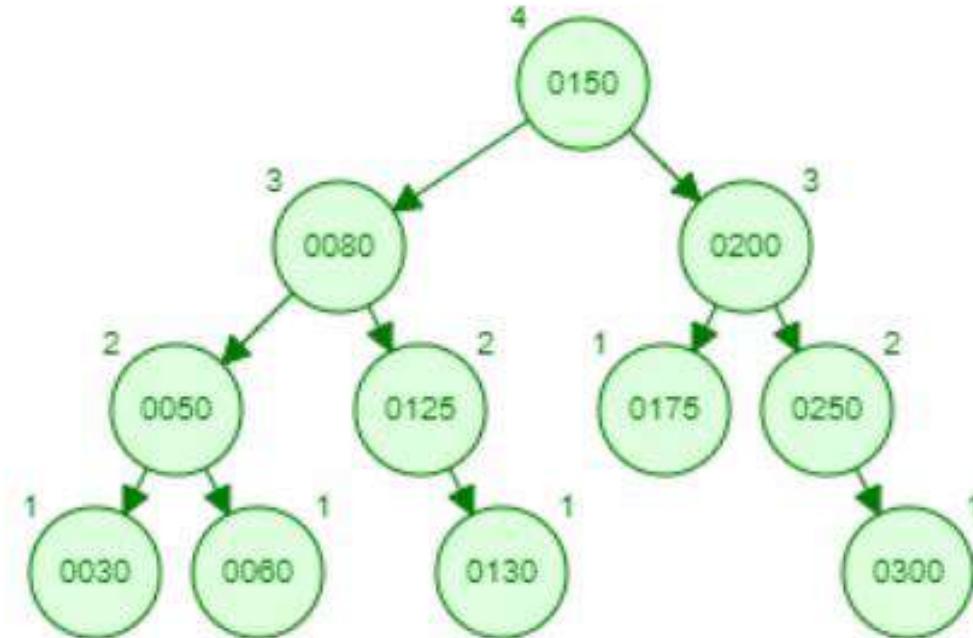
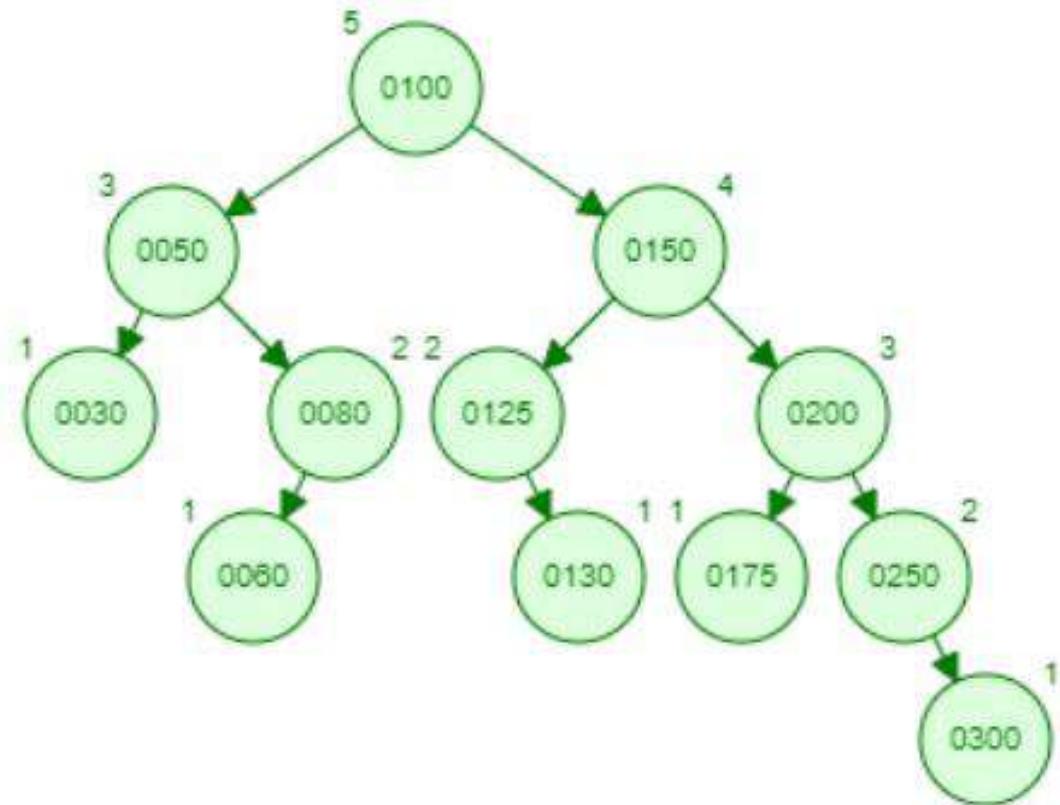


Check the balance of 25 (will change from -1 to zero)
Since it was changed check the balance of 50 (will change from 1 to 2)

Deleting an internal note: check from the replacement note and up.



Deleting an internal note: check from the replacement note and up.



Deleting 100: replacement node is 80. Check from 60 and up.
50 will change to zero. 80 will change to 2. Do a left rotation around 80 and stop.

Performance of AVL Trees

- Worst case height: $1.44 \lceil \log n \rceil$
- Thus, lookup, insert, remove all $O(\log n)$
- Empirical cost is $0.25 + \log n$ comparisons to insert

Red/Black Trees

- Another balanced BST approach is a red/black tree
- Each node has a color, usually implemented as a boolean value
- The following rules govern the color of a node:
 - the root is black
 - all children of red nodes are black
 - every path from the root to a leaf contains the same number of black nodes

Insertion

New nodes are coloured red and inserted as a standard BST.

There are 4 scenarios:

1. **N** is the root node, i.e., first node of red–black tree
2. **N**'s parent (**P**) is black
3. **P** is red (so it can't be the root of the tree) and **N**'s uncle (**U**) is red
4. **P** is red and **U** is black

Insertion

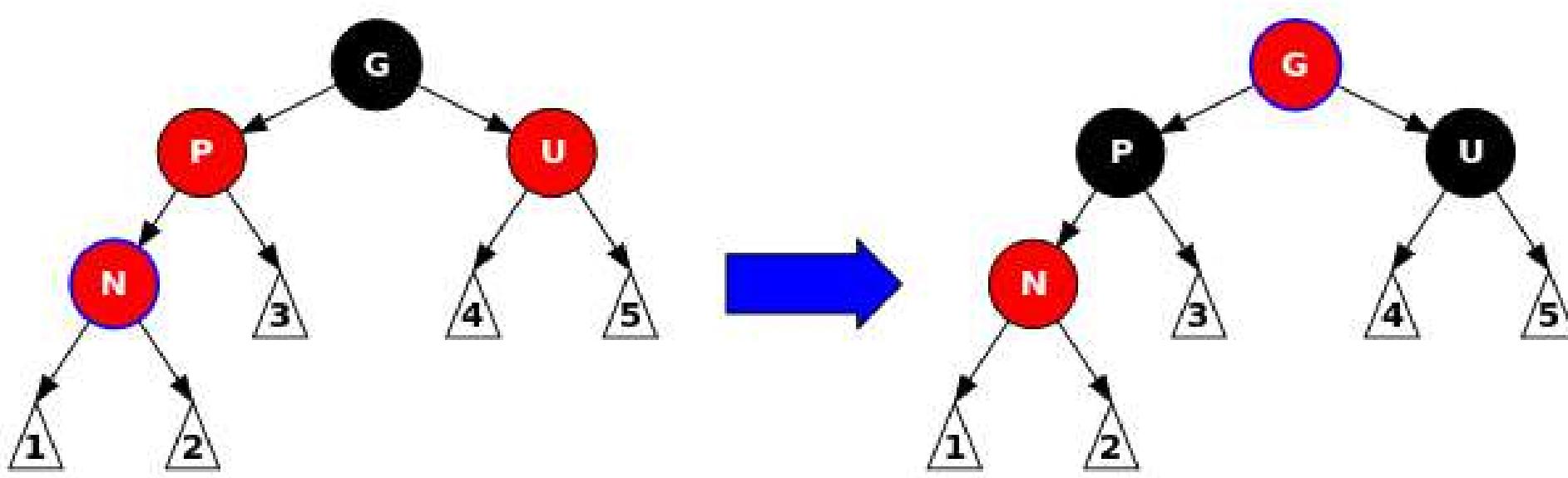
1. **N** is the root node, i.e., first node of red–black tree:

Just change the colour to black

2. **N**'s parent (**P**) is black

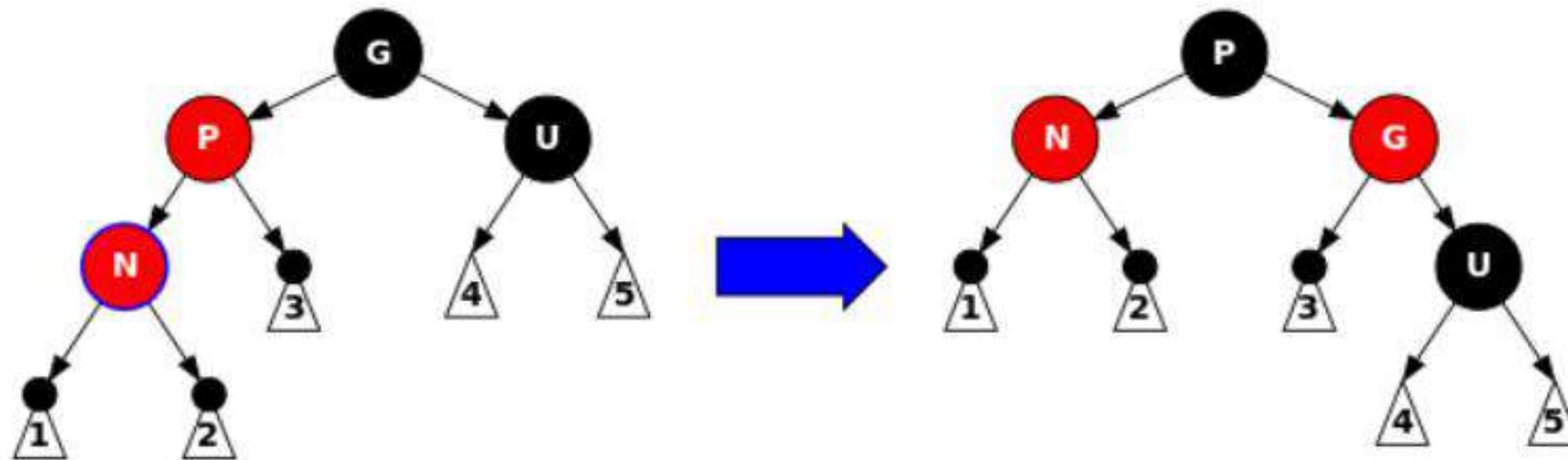
Everything is fine: no invariant is violated

3. P is red and N's uncle (U) is red.



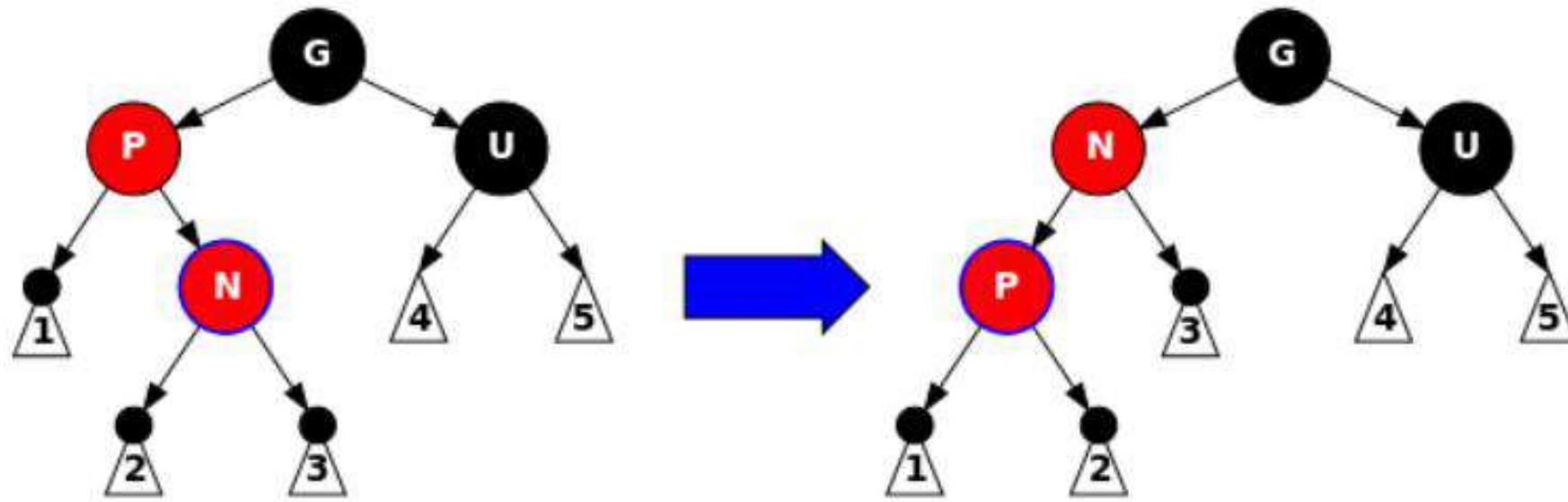
Change the colour of the parent and uncle
(possibly also grandparent if it is the root)

4. P is red and U is black



Rotate around the grandparent and then change
the colour of parent and the grandparent

4. P is red and U is black



If the tree is left-right heavy do a left rotation of the parent first.
(and mirror it if it is right-left heavy)

Sets, maps and hash tables

Agenda

- The **Map** and **Set** containers and how to use them with hash functions
- *Hash functions* and its use in efficient search & retrieval
- Two forms of hash tables:
 - *Open addressing and probing*
 - *Separate chaining*
 - Their relative benefits and performance tradeoffs
- *Implementing* both hash table forms
- Introduction to implementation of Maps and Sets

The Set Abstraction 1

- A set is a collection that contains no duplicate elements and at most one null element.
- Adding "apples" to the set {"apples", "oranges", "pineapples"} results in the same set (no change).

The Set Abstraction 2

- Operations on sets include:
 - Testing for membership
 - Adding elements
 - Removing elements
 - Union $A \cup B$
 - Intersection $A \cap B$
 - Difference $A - B$
 - Subset $A \subset B$

The Set Abstraction 3

- The union of two sets A, B is a set whose elements belong either to A or B or to both A and B.

Example: $\{1, 3, 5, 7\} \cup \{2, 3, 4, 5\}$ is $\{1, 2, 3, 4, 5, 7\}$

- The intersection of sets A, B is the set whose elements belong to both A and B.

Example: $\{1, 3, 5, 7\} \cap \{2, 3, 4, 5\}$ is $\{3, 5\}$

The Set Abstraction 4

- The difference of sets A, B is the set whose elements belong to A but not to B.

Examples: $\{1, 3, 5, 7\} - \{2, 3, 4, 5\}$ is $\{1, 7\}$;

$\{2, 3, 4, 5\} - \{1, 3, 5, 7\}$ is $\{2, 4\}$

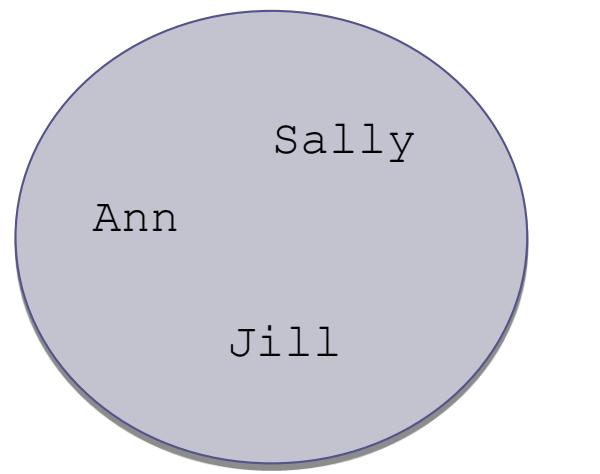
- Set A is a subset of set B if every element of set A is also an element of set B.

Example: $\{1, 3, 5, 7\} \subset \{1, 2, 3, 4, 5, 7\}$ is true

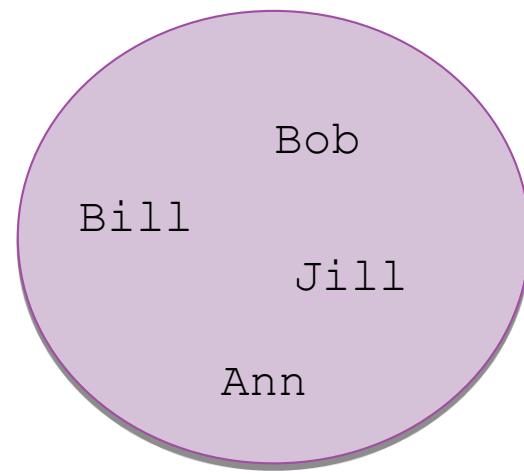
List vs. set

- Sets allow no duplicates
- Sets do not have *positions* - *in opposition to lists*
- Sets can give constant operation time for lookup, insert and delete

The Set Interface and Methods(cont.)

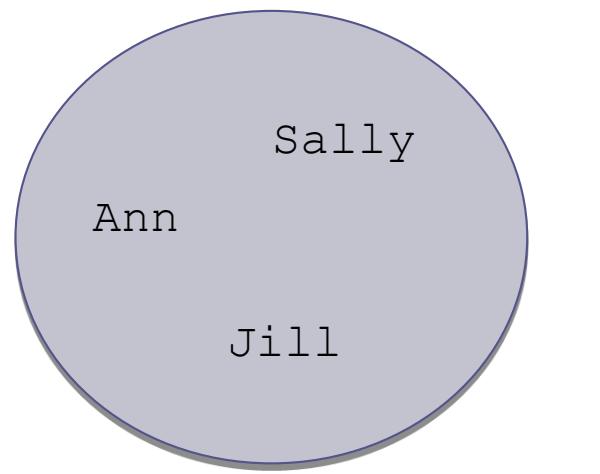


setA

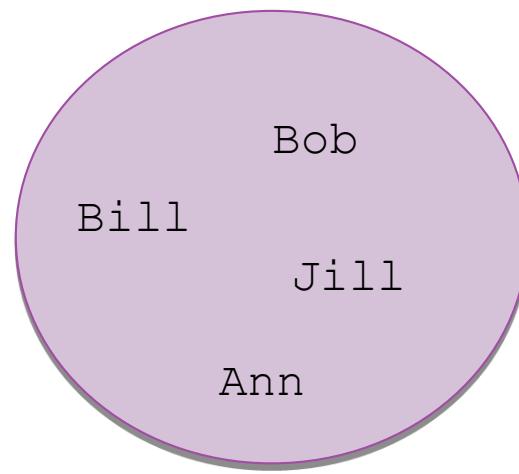


setB

The Set Interface and Methods(cont.)



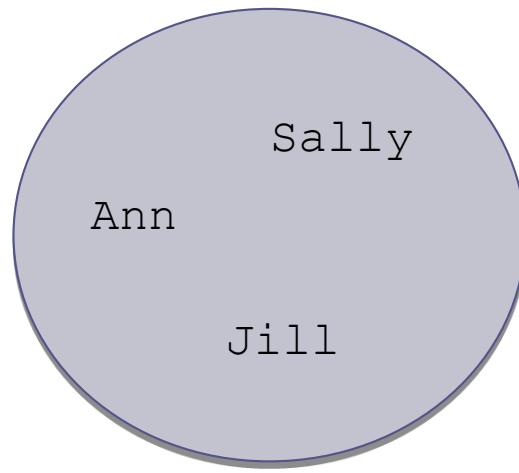
setA



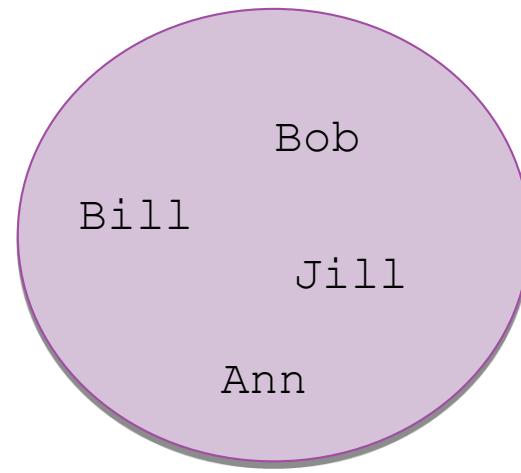
setB

```
setA.addAll(setB);
```

The Set Interface and Methods(cont.)



setA



setB

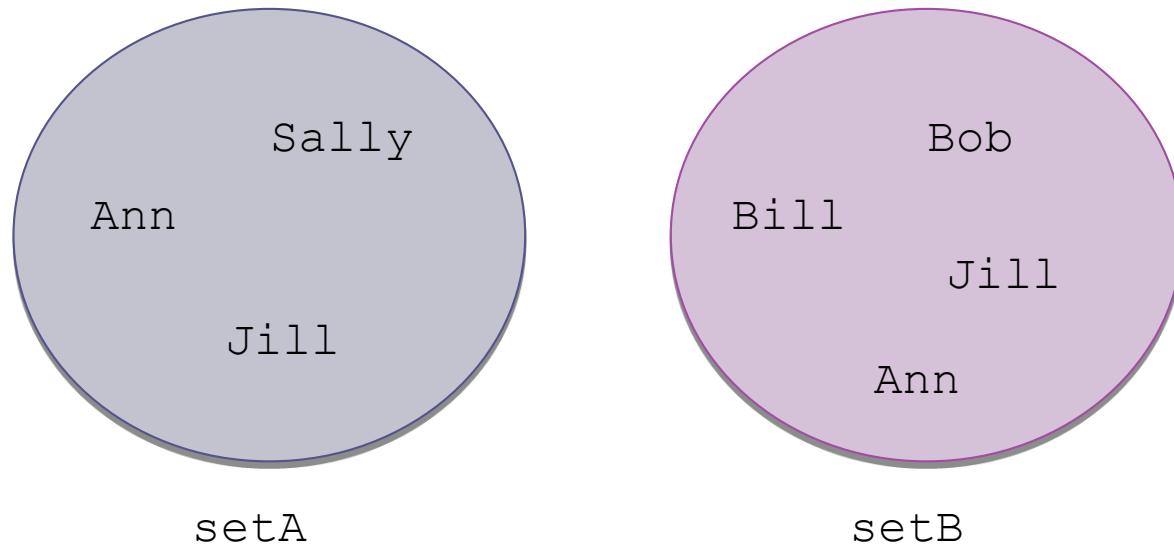
```
setA.addAll(setB);
```

```
System.out.println(setA);
```

Outputs:

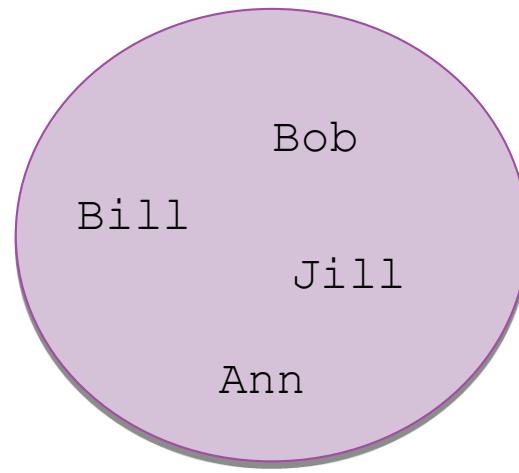
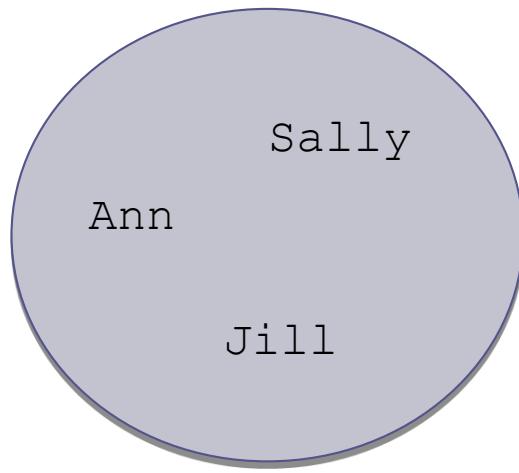
```
[Bill, Jill, Ann, Sally, Bob]
```

The Set Interface and Methods(cont.)



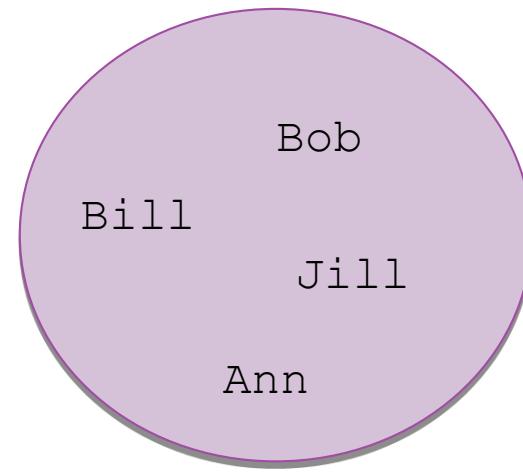
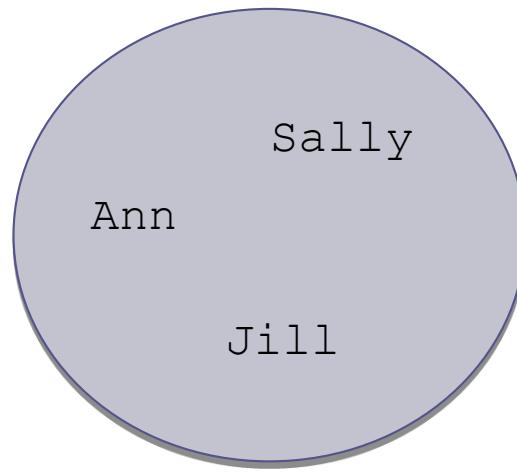
If a copy of original `setA` is in `setACopy`, then . . .

The Set Interface and Methods(cont.)



```
setACopy.retainAll(setB);
```

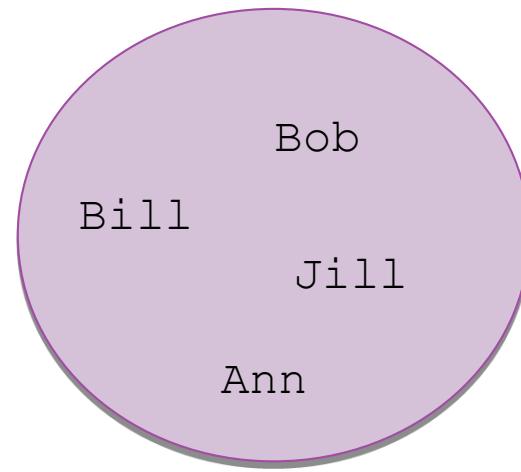
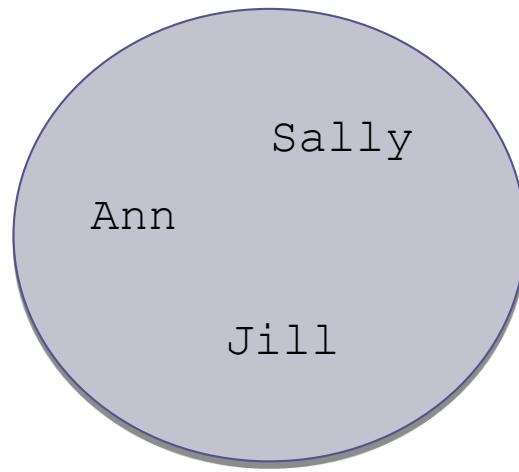
The Set Interface and Methods(cont.)



```
setACopy.retainAll(setB);  
  
System.out.println(setACopy);
```

Outputs:
[Jill, Ann]

The Set Interface and Methods(cont.)



```
setACopy.removeAll(setB);  
  
System.out.println(setACopy);
```

Outputs:
[Sally]

Maps and the Map Interface

- Map is related to Set: it is a set of ordered pairs
- Ordered pair: (*key*, *value*)
 - In a given `map`, there are no duplicate *keys*
 - Values may appear more than once
- Can think of key as “mapping to” a particular value
- Maps support efficient organization of information in tables
- Mathematically, these maps are:
 - Many-to-one (not necessarily one-to-one)
 - Onto (every value in the map has a key)

The map functions

- Template parameters
 - Key_Type: The type of the keys
 - Value_Type: The type of the values
 - Compare: The function class that compares the keys.
- All functions defined for the set are defined for the map, taking a pair`<Key_Type, Value_Type>`.

The map functions

- The Map interface should have methods of the form

v.get (Object key)

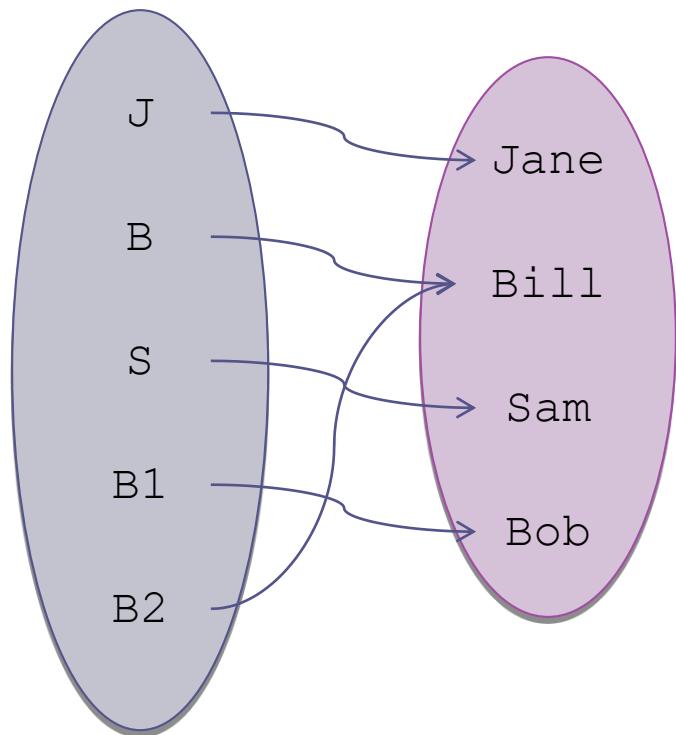
v.put (K key, v value)

- When information about an item is stored in a table, the information should have a unique ID
 - A unique ID may or may not be a number
 - This unique ID is equivalent to a key

Map Interface (cont.)

The following statements build a Map object:

```
Map<String, String> aMap =  
    new HashMap<String,  
    String>();  
  
aMap.put("J", "Jane");  
aMap.put("B", "Bill");  
aMap.put("S", "Sam");  
aMap.put("B1", "Bob");  
aMap.put("B2", "Bill");
```



Hash Tables

- **Goal:** access item given its *key* (not its *position*)
- Therefore, want to locate it directly from the key
- In other words, we wish to avoid much searching
- Hash tables provide this capability
 - Constant time in the average case! $O(1)$
 - Linear time in the worst case $O(n)$
- Searching an array: $O(n)$
- Searching BBST: $O(\log n)$

Hash Codes

- Suppose we have a table of size N
- A hash code is:
 - A number in the range 0 to N-1
 - We compute the hash code from the key
 - You can think of this as a “default position” when inserting, or a “position hint” when looking up
- A hash function is a way of computing a hash code
- **Desire:** The set of keys should spread evenly over the N values
- When two keys have the same hash code: collision

A Simple Hash Function

- Want to count occurrences of each Unicode character in a file
- There are 2^{16} possible characters, but ...
 - Maybe only 100 or so occur in a given file
- Approach: hash character to range 0-199
 - That is, use a hash table of size 200
- A possible hash function for this example:

```
int hash = uni_char % 200;
```

But d=100 so hash d=100%200=100
and ĩ=300 so hash ĩ=300%200=100

Collision !

Devising Hash Functions

- Simple functions often produce many collisions
 - ... but complex functions may not be good either!
- It is often an empirical process
 - Adding letter values in a string: same hash for strings with same letters in different order
 - e.g. sing = sign
 - Better approach:

```
size_t hash = 0;
for (size_t i = 0; i < s.size(); ++i)
    hash += 31^((s.size() - 1) - i) + s[i];
```
 - This is the hash function used by Java in its String class.

Devising Hash Functions (2)

- The **String** hash is good in that:
 - Every letter affects the value
 - The order of the letters affects the value
 - The values tend to be spread well over the integers
 - But long strings are expensive to compute (often only a sample of letters are used)

e.g. Tom =

‘T’ = 84 ‘o’=111 ‘m’ = 109

Hash Tom = $84 \cdot 31^2 + 111 \cdot 31^1 + 109 \cdot 31^0 =$
 $80724 + 3441 + 109 = 84274$

Devising Hash Functions (3)

- Guidelines for good hash functions:
 - Spread values evenly: as if “random”
 - Cheap to compute
- Generally, number of possible values are much greater than table size

Open Addressing and probing

- Will consider two ways to organize hash tables
 - Open addressing and probing
 - Separate chaining
- Open addressing and probing:
 - Hashed items are in a single array
 - Hash code gives position “hint”
 - Handle collisions by checking multiple positions
 - Each check is called a probe of the table

Linear Probing

- Probe by incrementing the index
- If “fall off end”, wrap around to the beginning
 - Take care not to cycle forever!
- 1. Compute **index** as **hash_fcn()** % **table.size()**
- 2. if **table[index]** == **NULL**, item is not in the table
- 3. if **table[index]** matches item, found item (done)
- 4. Increment index circularly and go to 2
- Why must we probe repeatedly?
 - **hashCode** may produce collisions

Search Termination

Ways to obtain proper termination

- Stop when you come back to your starting point
- Stop after probing N slots, where N is table size
- Stop when you reach the bottom the second time
- Ensure table never full
 - Reallocate when occupancy exceeds threshold

Hash Table Considerations

- Cannot traverse a hash table
 - Order of stored values is arbitrary
 - Can use an iterator to produce in arbitrary order
- When item is deleted, cannot just set its entry to null
 - Doing so would break probing
 - Must store a “dummy value” instead
 - Deleted items waste space and reduce efficiency
- Higher occupancy causes makes for collisions

Hash Table Example

- Table of strings, initial size 5
- Add “Tom”, hash 84274 → 4 Slot 4
- Add “Dick”, hash 2129869 → 4 Slot 0 (wraps)
- Add “Harry”, hash 69496448 → 3 Slot 3
- Add “Sam”, hash 82879 → 4 Slot 1 (wraps)
- Add “Pete”, hash 2484038 → 3 Slot 2 (wraps)
- Note: many lookups will probe a lot!
- Size 11 gives these slots: 3, 5, 10, 5→6, 7

Reducing Collisions By Growing: Rehashing

- Choose a new larger size, e.g., doubling
- (Re)insert non-deleted items into new array
- Install the new array and drop the old
- Similar to reallocating a vector
 - *But*, elements can move around in reinsertion
 - Rehashing distributes items at least as well

Quadratic Probing

- Linear probing
 - Tends to form long clusters of keys in the table
 - This causes longer search chains
- Quadratic probing can reduce the effect of clustering
 - Index increments form a quadratic series
 - Direct calculation involves multiply, add, remainder
 - Incremental calculation better
 - Probe sequence may not produce all table slots

Quadratic Probing (2)

- Generating the quadratic sequence

Want: $s, s+1^2, s+2^2, s+3^2, s+4^2$, etc. (all % length)

“Trick” to calculate incrementally:

Initially:

```
size_t index = ... 1st probe slot ...
int k = -1;
```

At each iteration:

```
k += 2;
index = (index + k) % table.size();
```

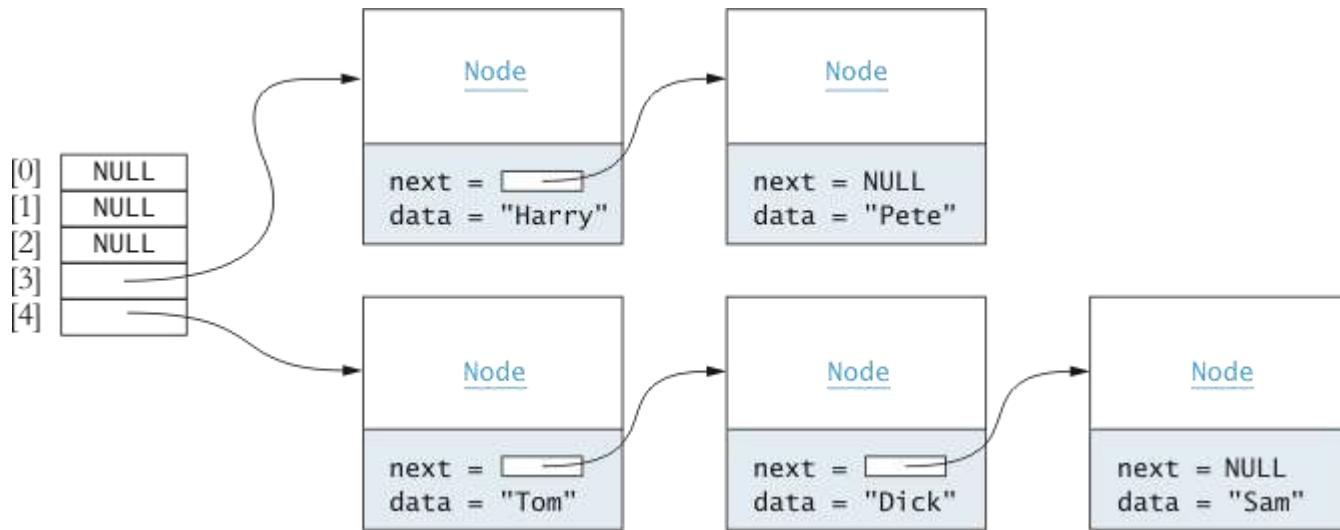
Double hashing

- Another way to deal with conflicts is double hashing.
- When a collision happens the next index is found by applying a second function on the key and add this value to the original hash code:
 $\text{hash_fcn}(i) + \text{hash}_2\text{_fcn}(i)$
- If this also produce a conflict 2 times the second function is added and if that also produce a conflict then 3 time etc.

Separate chaining

- Alternative to open addressing and probing
- Each table slot references a linked list
 - List contains all items that hash to that slot
 - The linked list is often called a bucket
 - So sometimes called bucket hashing
- Examines only items with same hash code
- Insertion about as complex
- Deletion is simpler
- Linked list can become long → rehash

Chaining Picture



Two items hashed to bucket 3

Three items hashed to bucket 4

Performance of Hash Tables

- Load factor = # filled cells / table size
 - Between 0 and 1 for open addressing
- Load factor has greatest effect on performance
- Lower load factor → better performance
 - Reduce collisions in sparsely populated tables
- Expected # probes p for open addressing, linear probing, load factor L : $p = \frac{1}{2}(1 + 1/(1-L))$
 - As L approaches 1, this zooms up
- For chaining, $p = 1 + (L/2)$
 - Note: Here L can be greater than 1!

Performance of Hash Tables (3)

- Hash table:
 - Insert: average $O(1)$
 - Search: average $O(1)$
- Sorted array:
 - Insert: average $O(n)$
 - Search: average $O(\log n)$
- Binary Search Tree:
 - Insert: average $O(\log n)$
 - Search: average $O(\log n)$
- A balanced tree can *guarantee* $O(\log n)$

Priority Queues

- A FIFO queue removes elements in the order in which they were added.
- A LIFO queue (stack) removes the last inserted first
- A *priority queue* removes elements in priority order, independent of the order in which they were added
- Priority queues are helpful in many scheduling situations
- A heap is a classic mechanism for implementing priority queues

Heaps

- Heaps, conceptually
- Using heaps to solve problems
- Heap implementations
- Using heaps to implement priority queues

Heaps

- A *heap* is a complete binary tree in which each element is less than or equal to both of its children
- Where the BST has a left/right ordering the heap has an up/down ordering
- A heap has both structural and ordering constraints
- As with binary search trees, there are many possible heap configurations for a given set of elements
- Our definition above is a *minheap*
- A similar definition could be made for a *maxheap*

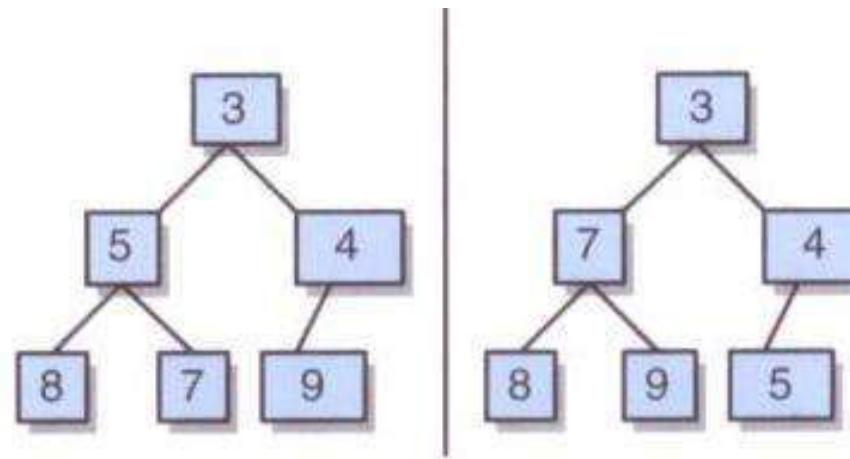
Heaps

- Operations on a heap:

Operation	Description
addElement	Adds the given element to the heap.
removeMin	Removes the minimum element in the heap.
findMin	Returns a reference to the minimum element in the heap.

Heaps

- Two minheaps containing the same data:

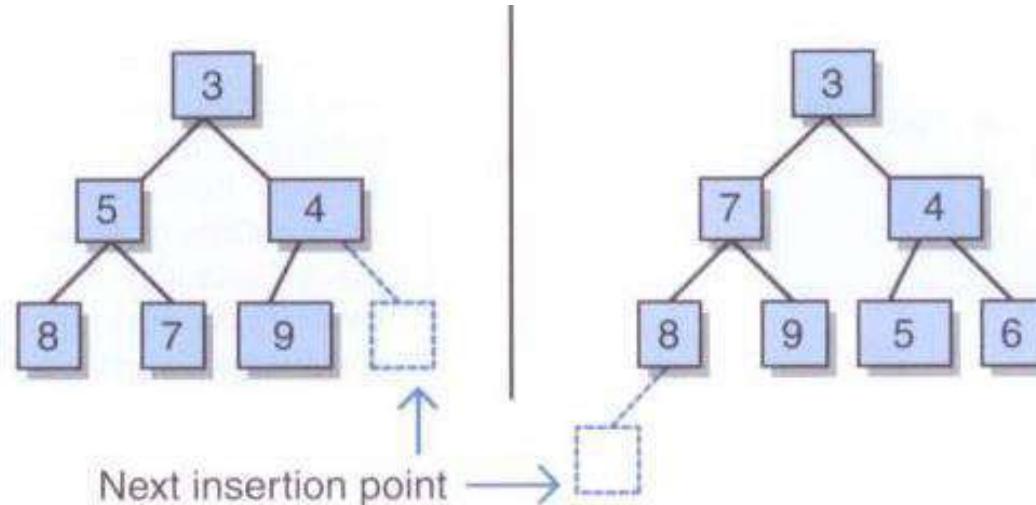


Adding a New Element

- To add an element to the heap, add the element as a leaf, keeping the tree complete
- Then, move the element up toward the root, exchanging positions with its parent, until the relationship among the elements is appropriate
- This will guarantee that the resulting tree will conform to the heap criteria

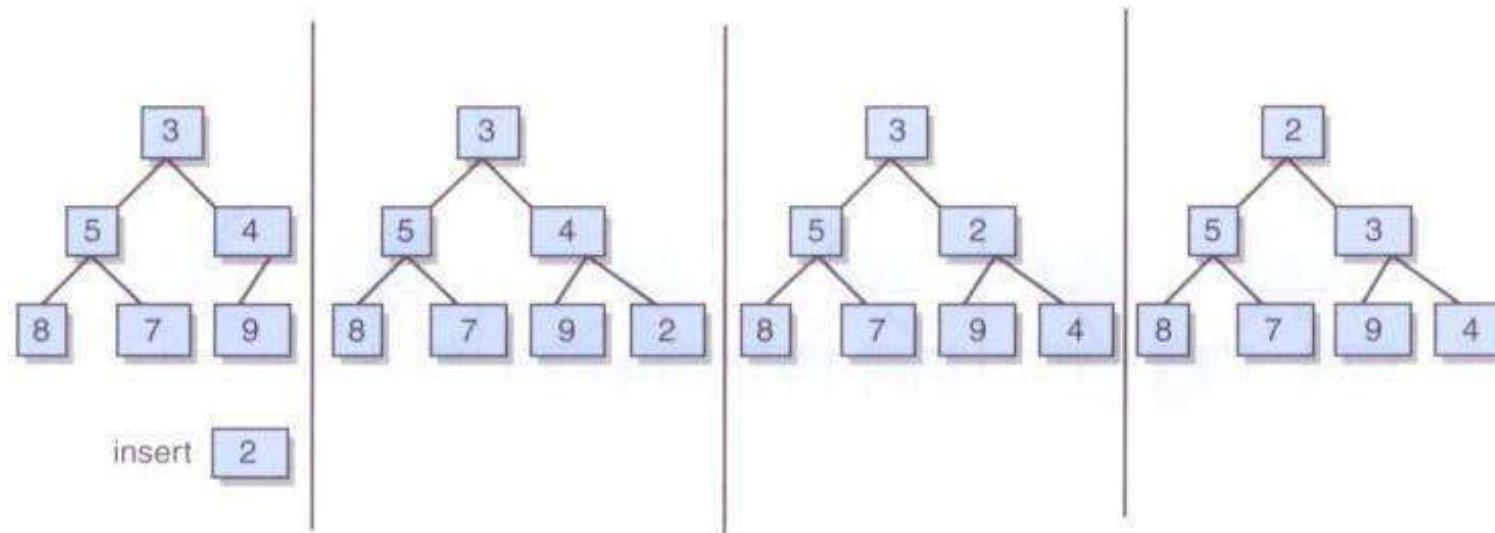
Adding a New Element

- The initial insertion point for a new element in a heap:



Adding a New Element

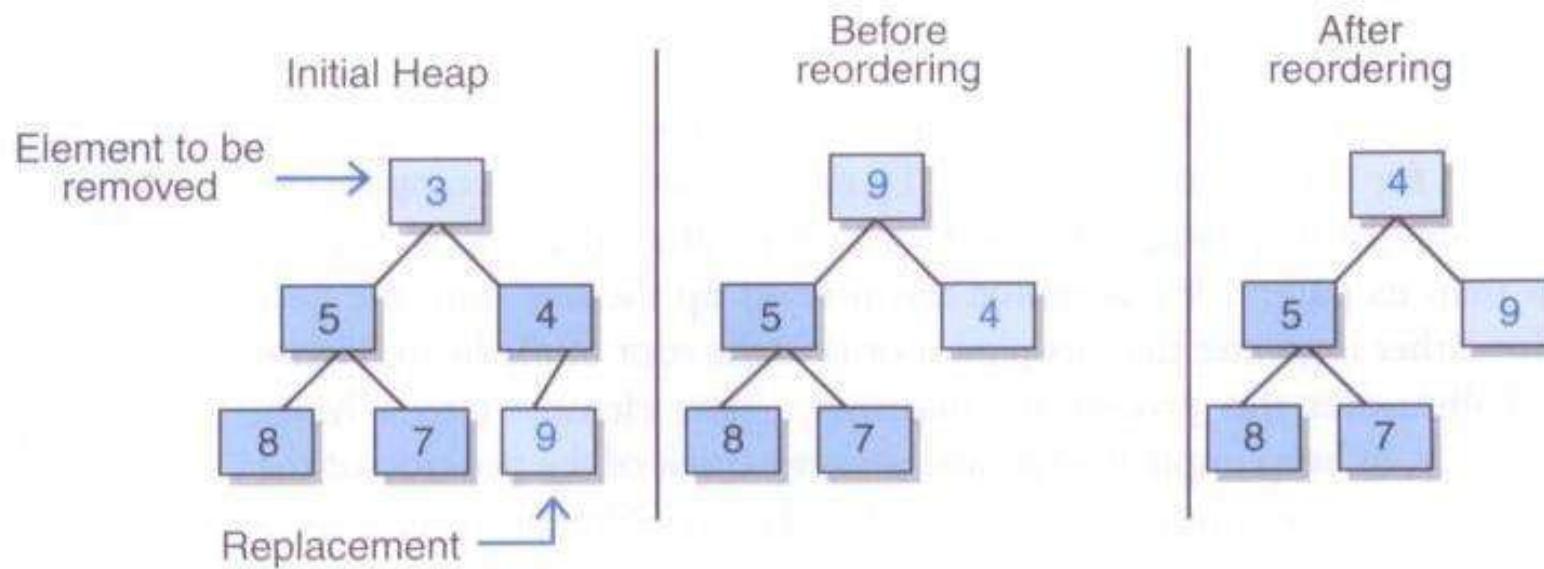
- Inserting an element and moving it up the tree as far as appropriate:



Removing the Min Element

- Remove the root (min) and reconstruct the heap
- First, move the last leaf of the tree to be the new root of the tree
- Then, move it down the tree as needed until the relationships among the elements is appropriate
- The algorithm is: compare the parent with its children and swap with the smallest if the child is less than the parent.

Removing the Min Element



Implementing Heaps with Links

- The operations on a heap require moving up the heap as well as down
- So we'll add a parent pointer to the `HeapNode` class, which is itself based on the node for a binary tree
- In the heap itself, we'll keep track of a pointer so that we always know where the last leaf is

Implementing Heaps with Arrays

- Since a heap is a complete tree, an array-based implementation is a good choice because it is very memory efficient
- As previously discussed, a parent element at index n will have children stored at index $2n+1$ and $2n+2$ of the array
- Conversely, for any node other than the root, the parent of the node is found at index $(n-1)/2$

Heap Sort

- Given the ordering property of a heap, it is natural to think of using a heap to sort a list of numbers
- A *heap sort* sorts a set of elements by adding each one to a heap, then removing them one at a time
- The smallest element comes off the heap first, so the sequence will be in ascending order
- The array can be sorted “in place” no extra memory is needed.

Heap sort

Sort the array:

31	4	9	33	22	11
----	---	---	----	----	----

First step is to create a heap out of the array

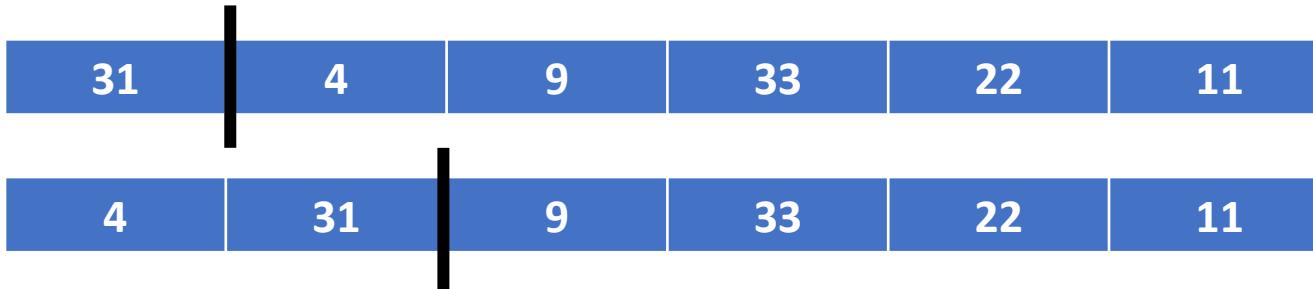
0	1	2	3	4	5
---	---	---	---	---	---

31	4	9	33	22	11
----	---	---	----	----	----

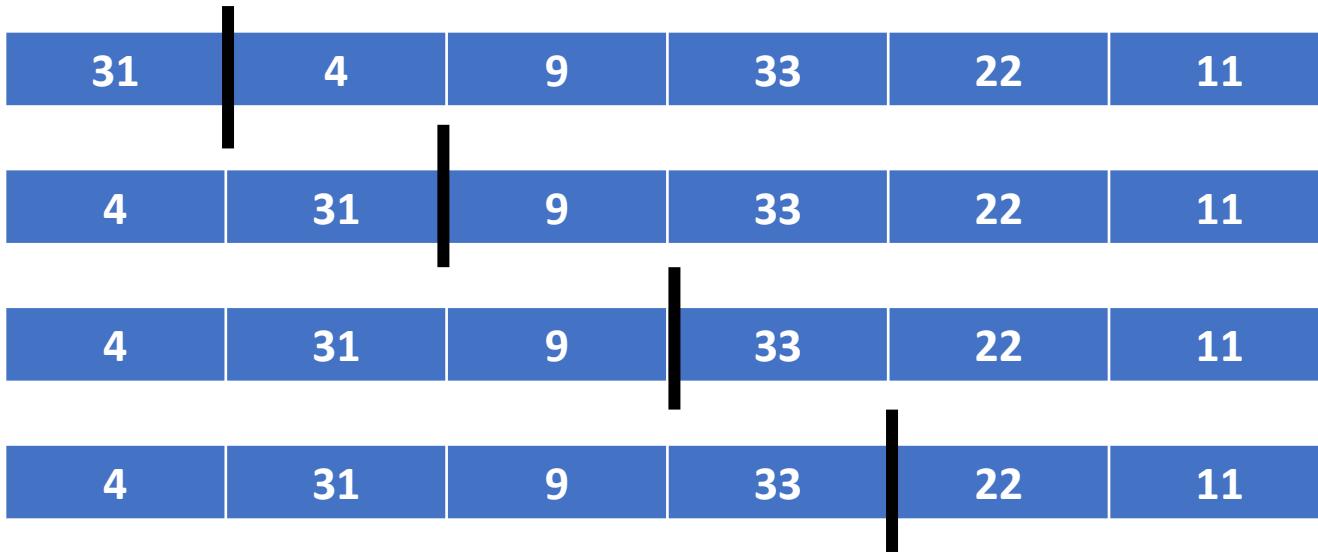
0	1	2	3	4	5
---	---	---	---	---	---

31	4	9	33	22	11
----	---	---	----	----	----

0	1	2	3	4	5
---	---	---	---	---	---



0	1	2	3	4	5
---	---	---	---	---	---



0	1	2	3	4	5
---	---	---	---	---	---

31	4	9	33	22	11
----	---	---	----	----	----

4	31	9	33	22	11
---	----	---	----	----	----

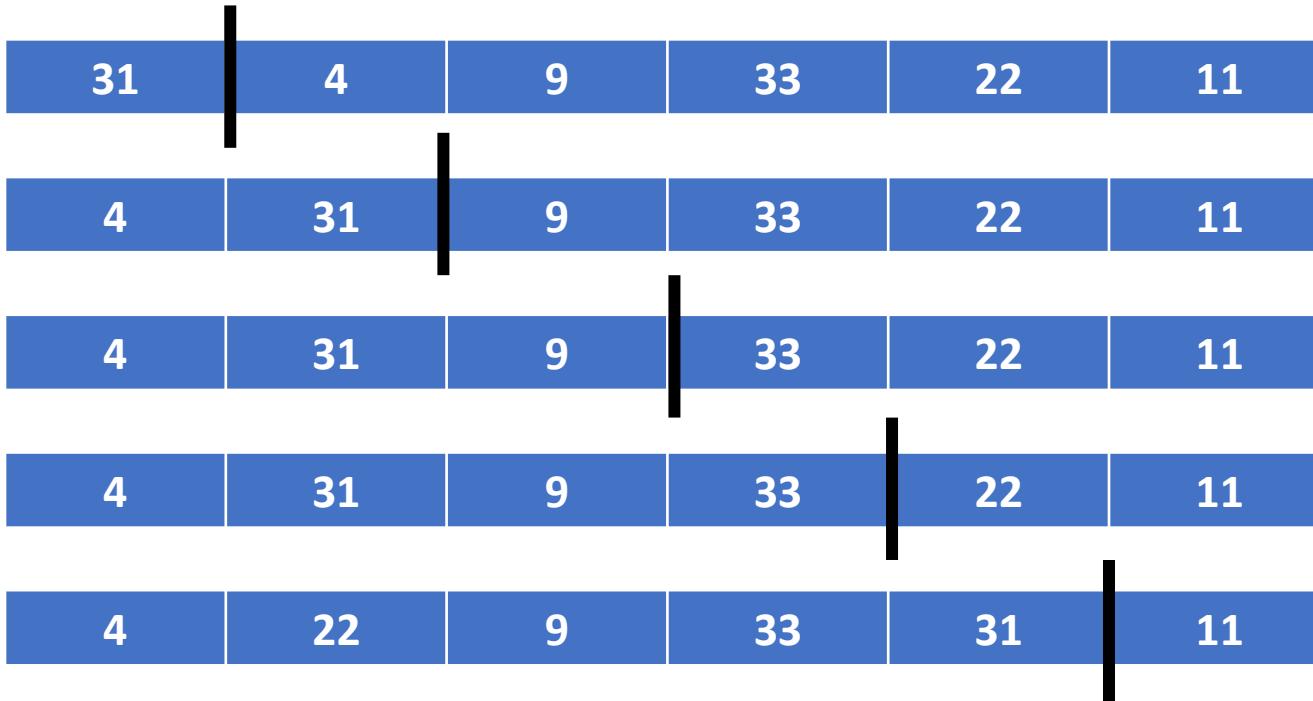
4	31	9	33	22	11
---	----	---	----	----	----

4	31	9	33	22	11
---	----	---	----	----	----

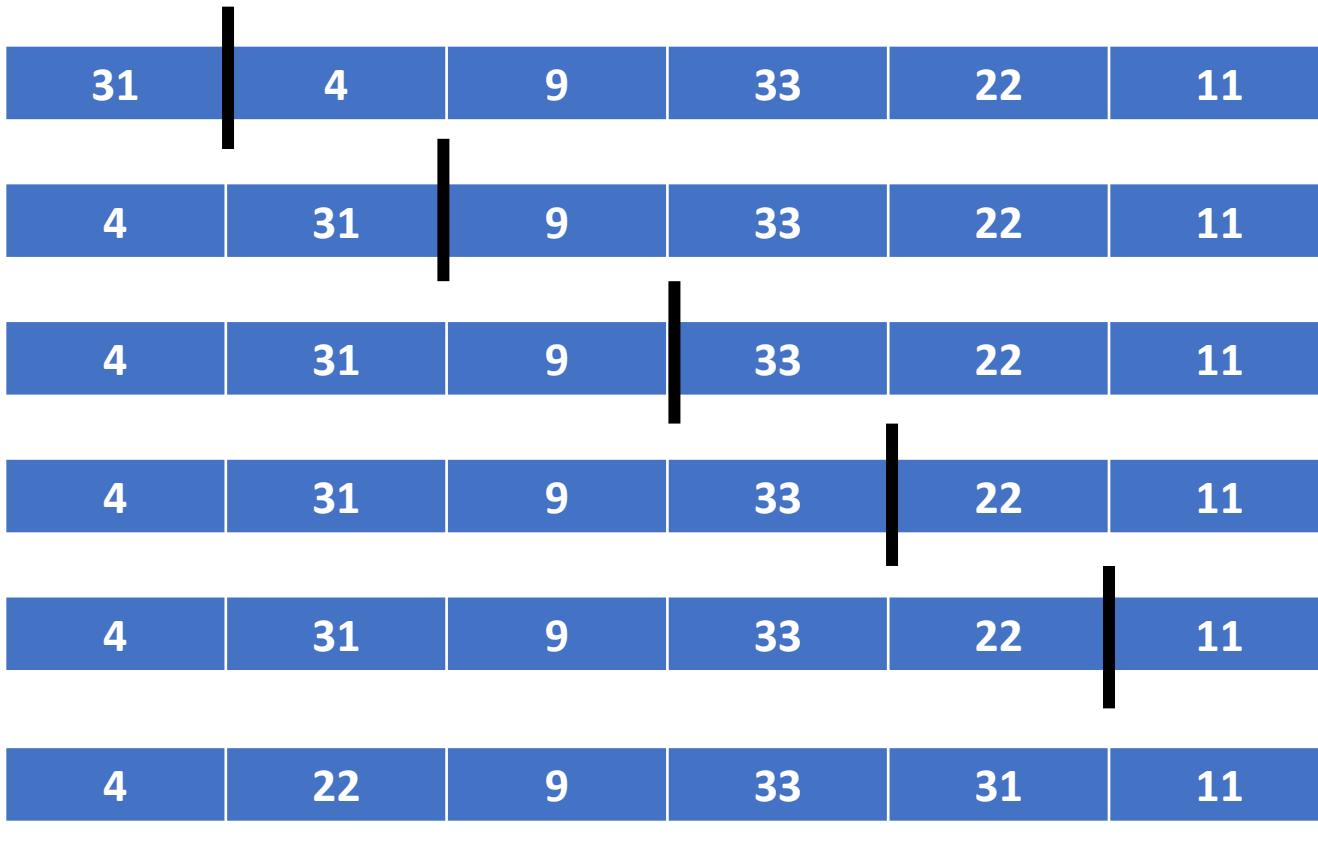
4	31	9	33	22	11
---	----	---	----	----	----

$$(n-1)/2 = p \quad (4-1)/2 = 1$$

0	1	2	3	4	5
---	---	---	---	---	---



0	1	2	3	4	5
---	---	---	---	---	---



$$(n-1)/2 = p \quad (5-1)/2 = 2$$

Heap sorte

Now the heap:  4 | 22 | 9 | 33 | 31 | 11

Is transformed into a sorted array.

0	1	2	3	4	5
---	---	---	---	---	---

4	22	9	33	31	11
---	----	---	----	----	----

	22	9	33	31	11	4
--	----	---	----	----	----	---

0	1	2	3	4	5
---	---	---	---	---	---

4	22	9	33	31	11
---	----	---	----	----	----

11	22	9	33	31	4	4
----	----	---	----	----	---	---

0	1	2	3	4	5
---	---	---	---	---	---

4	22	9	33	31	11
---	----	---	----	----	----

9	22	11	33	31	4	4
---	----	----	----	----	---	---

0	1	2	3	4	5
---	---	---	---	---	---

4	22	9	33	31	11
---	----	---	----	----	----

9	11	22	33	31	4	4
---	----	----	----	----	---	---

	11	22	33	31	4	9
--	----	----	----	----	---	---

0	1	2	3	4	5
---	---	---	---	---	---

4	22	9	33	31	11
---	----	---	----	----	----

9	11	22	33	31	4	4
---	----	----	----	----	---	---

31	11	22	33	9	4	9
----	----	----	----	---	---	---

0	1	2	3	4	5
---	---	---	---	---	---

4	22	9	33	31	11
---	----	---	----	----	----

9	11	22	33	31	4	4
---	----	----	----	----	---	---

11	31	22	33	9	4	9
----	----	----	----	---	---	---

0	1	2	3	4	5
---	---	---	---	---	---

4	22	9	33	31	11
---	----	---	----	----	----

9	11	22	33	31	4	4
---	----	----	----	----	---	---

11	31	22	33	9	4	9
----	----	----	----	---	---	---

	31	22	33	9	4	11
--	----	----	----	---	---	----

0	1	2	3	4	5
---	---	---	---	---	---

4	22	9	33	31	11
---	----	---	----	----	----

9	11	22	33	31	4	4
---	----	----	----	----	---	---

11	31	22	33	9	4	9
----	----	----	----	---	---	---

33	31	22	11	9	4	11
----	----	----	----	---	---	----

0	1	2	3	4	5
---	---	---	---	---	---

4	22	9	33	31	11
---	----	---	----	----	----

9	11	22	33	31	4	4
---	----	----	----	----	---	---

11	31	22	33	9	4	9
----	----	----	----	---	---	---

22	31	33	11	9	4	11
----	----	----	----	---	---	----

0	1	2	3	4	5
---	---	---	---	---	---

4	22	9	33	31	11
---	----	---	----	----	----

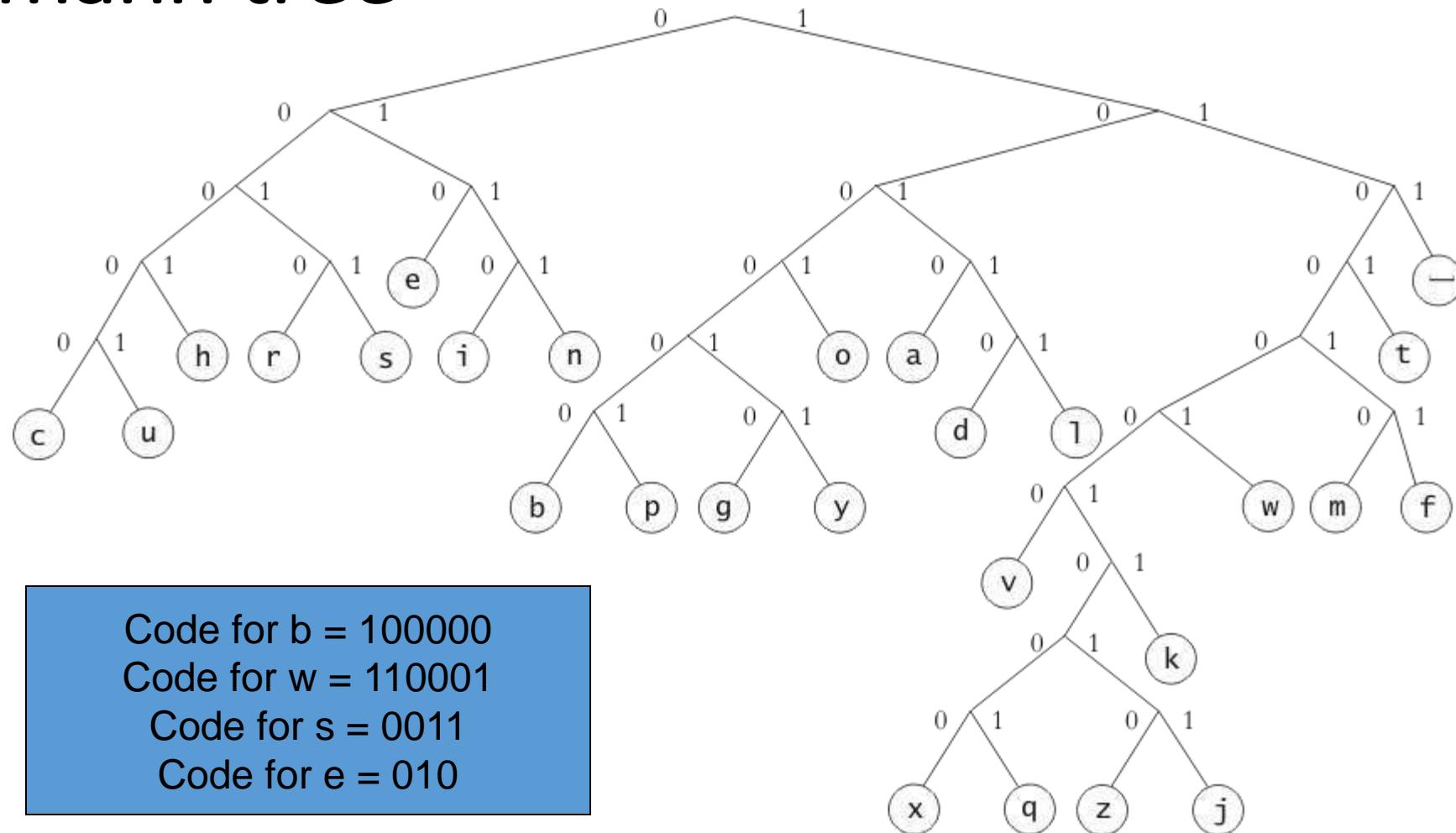
9	11	22	33	31	4	4
---	----	----	----	----	---	---

11	31	22	33	9	4	9
----	----	----	----	---	---	---

22	31	33	11	9	4	11
----	----	----	----	---	---	----

33	31	22	11	9	4
----	----	----	----	---	---

Hoffmann tree



How to use

- To encode: follow the path to the letter and write down the ciphers
- To decode: follow the path given by the ciphers and find the letter.

Exercises

Using the Huffman tree:

- a. Write the binary string for the message “scissors cuts paper”.

- b. Decode the following binary string:

110001000101000100101110110001111111000110101011101101
001

Sorting

Agenda

- How sorting algorithms works
- How to implement sorting algorithms
 - Selection sort
 - Bubble sort
 - Insertion sort
 - Shell sort
 - Merge sort
 - Heapsort
 - Quicksort

Running time and memory usage

- Understand the performance of these algorithms
 - What influence has the size of the input on the running time?
 - What influence has the ordering of the input (is it randomly distributed, is it nearly sorted, is it sorted the wrong way etc.) on the running time.
 - What is worst case, average case and best case?
- Is axillary memory needed?

Selection Sort

- A relatively easy to understand algorithm
- Sorts an array in passes
 - Each pass selects the next smallest element
 - At the end of the pass, places it where it belongs
- Efficiency is $O(n^2)$, hence called a quadratic sort
- Performs:
 - $O(n^2)$ comparisons
 - $O(n)$ exchanges (swaps)
 - In place sorting (no extra memory)

Selection Sort Algorithm

1. for **fill** = 0 to $n-2$ do // steps 2-6 form a pass
2. set **pos_min** to **fill**
3. for **next** = **fill**+1 to $n-1$ do
4. if item at **next** < item at **pos_min**
5. set **pos_min** to **next**
6. Exchange item at **pos_min** with one at **fill**

Selection Sort Example

35	65	30	60	20	scan 0-4, smallest 20
					swap 35 and 20
20	65	30	60	35	scan 1-4, smallest 30
					swap 65 and 30
20	30	65	60	35	scan 2-4, smallest 35
					swap 65 and 35
20	30	35	60	65	scan 3-4, smallest 60
					swap 60 and 60
20	30	35	60	65	done

Selection Sort

- Why quadratic ?
- First pass do $n-1$ comparisons, second do $n-2$ comparisons and so forth until there are two elements left (when there is one element left there can be no comparisons).
- Mathematical: $n-1 + n-2 + \dots + 2 + 1$
- According to Gauss then for $n + n-1 + n-2 + \dots + 1$

$$s = \frac{n(n+1)}{2} \quad \text{We have to subtract } n:$$

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

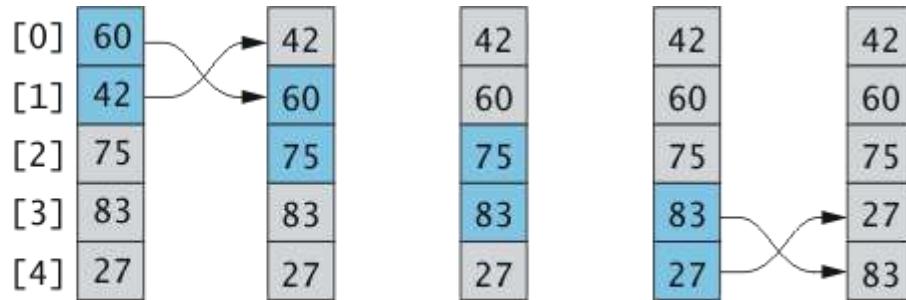
- Which is quadratic

Bubble Sort

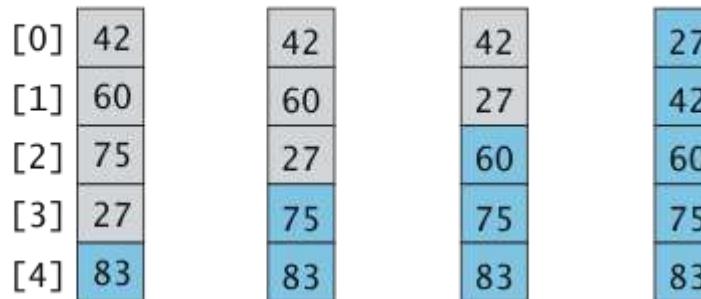
- Compares adjacent array elements
 - Exchanges their values if they are out of order
- Smaller values bubble up to the top of the array
 - Larger values sink to the bottom

Bubble Sort Example

One pass
of Bubble Sort



Array after
Completion
of Each Pass



Bubble Sort Algorithm

1. do
2. for each pair of adjacent array elements
3. if values are out of order
4. Exchange the values
5. while the array is not sorted

Bubble Sort Algorithm, Refined

1. do
2. Initialize **exchanges** to **false**
3. for each pair of adjacent array elements
4. if values are out of order
5. Exchange the values
6. Set **exchanges** to **true**
7. while **exchanges**

Analysis of Bubble Sort

- Excellent performance *in some cases*
 - But *very poor* performance in others!
- Works ***best*** when array is nearly sorted to begin with
- Worst case number of comparisons: $O(n^2)$
- Worst case number of exchanges: $O(n^2)$
- *Best case* occurs when the array is already sorted:
 - $O(n)$ comparisons
 - $O(1)$ exchanges (none actually)

Insertion Sort

- Based on technique of card players to arrange a hand
 - Player keeps cards picked up so far in sorted order
 - When the player picks up a new card
 - Makes room for the new card
 - Then inserts it in its proper place

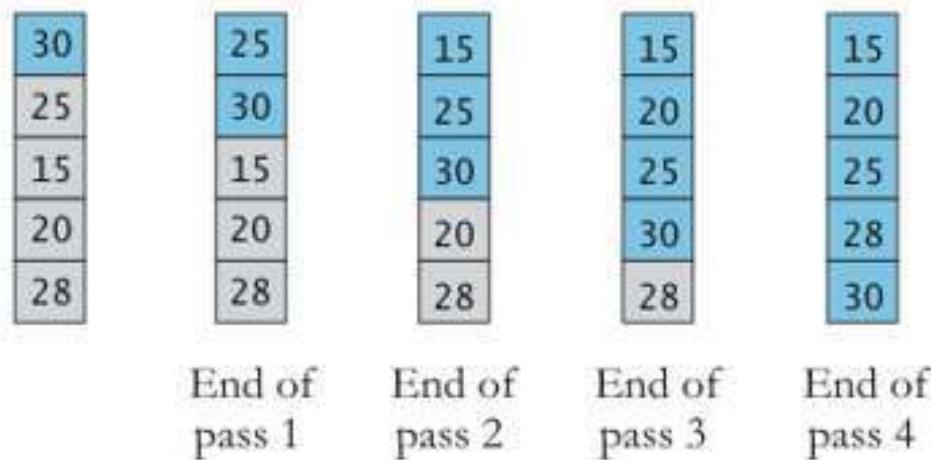


Insertion Sort Algorithm

- For each element from 2nd (`next_pos = 1`) to last:
 - Insert element at `next_pos` where it belongs
 - Increases sorted subarray size by 1
- To make room:
 - Hold `next_pos` value in a variable
 - Shuffle elements to the right until gap at right place

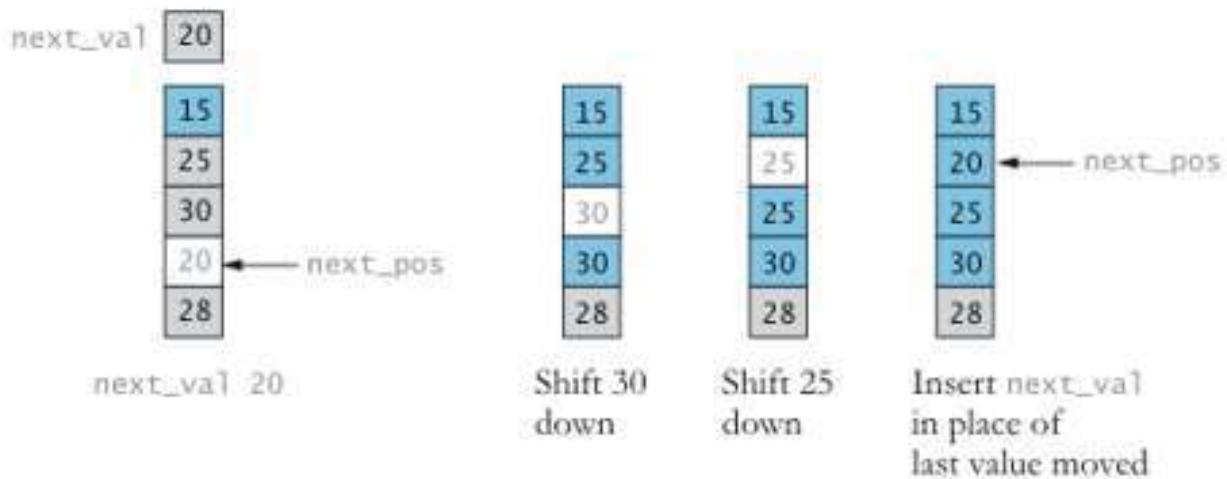
Insertion Sort Example

An Insertion
Sort



Insertion Sort Example

Inserting the
Fourth Array
Element



Analysis of Insertion Sort

- Maximum number of comparisons: $O(n^2)$
- In the best case, number of comparisons: $O(n)$
- Fast for small arrays and nearly sorted arrays (used by more complex sorting algorithms)
- In place sorting

Comparison of Quadratic Sorts

- None good for large arrays!

	Number of Comparisons		Number of Exchanges	
	Best	Worst	Best	Worst
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n)$	$O(n)$
Bubble Sort	$O(n)$	$O(n^2)$	$O(1)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n)$	$O(n^2)$

n	n^2	$n \log n$
8	64	24
16	256	64
32	1,024	160
64	4,096	384
128	16,384	896
256	65,536	2,048
512	262,144	4,608

Shell Sort: A Better Insertion Sort

- Shell sort is a variant of insertion sort
 - It is named after Donald Shell
 - Average performance: $O(n^{3/2})$ or better
- Divide and conquer approach to insertion sort
 - Sort many smaller subarrays using insertion sort
 - Sort progressively larger arrays
 - Finally sort the entire array
- These arrays are elements separated by a gap
 - Start with large gap
 - Decrease the gap on each “pass”

Illustration of Shell Sort

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
40	35	80	75	60	90	70	75	55	90	85	34	45	62	57	65

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
40	35	80	75	34	45	62	57	55	90	85	60	90	70	75	65

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
40	35	80	75	34	45	62	57	55	90	85	60	90	70	75	65

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
40	34	45	62	35	55	65	57	60	75	70	75	90	85	80	90

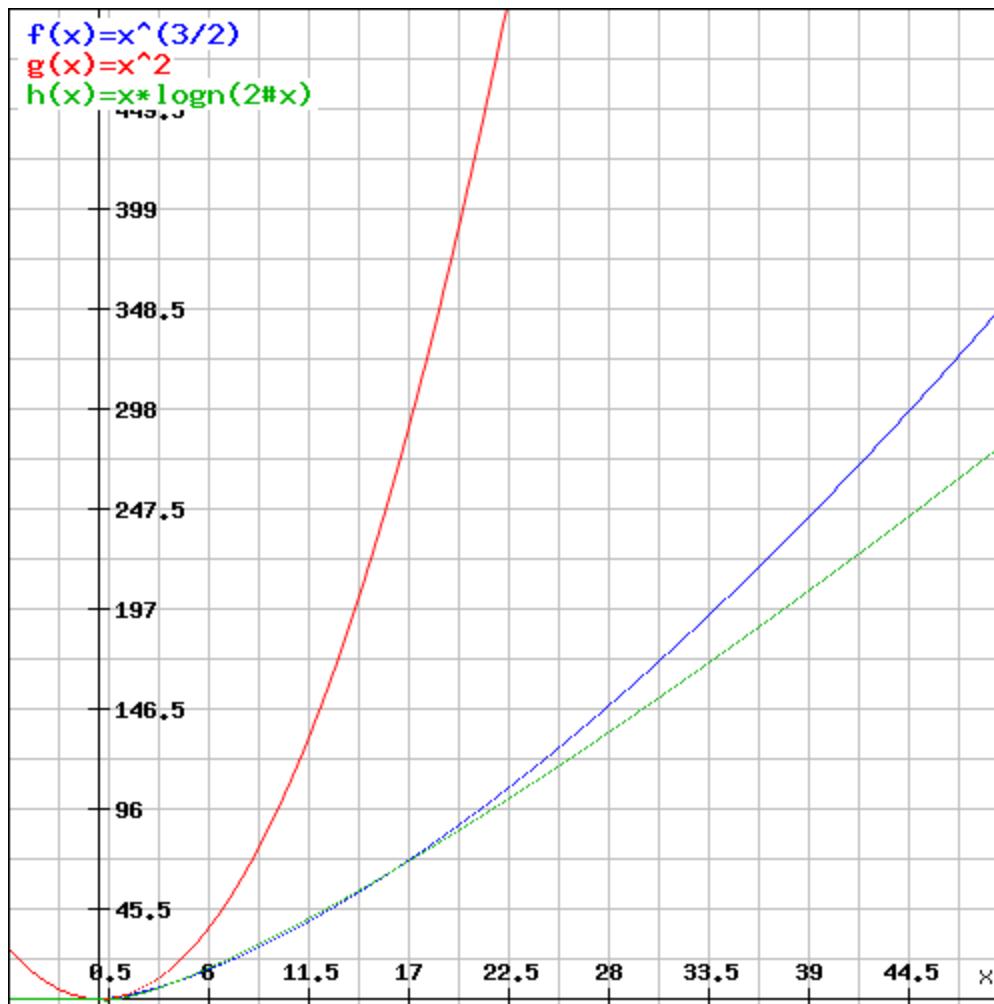
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
34	35	40	45	55	57	60	62	65	70	75	75	80	85	90	90

Shell Sort Why it works:

- We know that insertion sort is ok on small arrays.
- We know that insertion sort is ok on almost sorted arrays.
- Combining the two facts.
- But what about very large arrays?

Analysis of Shell Sort

- ***Intuition:***
 - Reduces work by moving elements farther earlier
 - Its general analysis is an open research problem
 - Performance depends on sequence of gap values
 - For sequence 2^k , performance is $O(n^2)$
 - Hibbard's sequence (2^k-1), performance is $O(n^{3/2})$
 - We start with $n/2$ and repeatedly divide by 2.2
 - Empirical results show this is $O(n^{5/4})$ or $O(n^{7/6})$
 - No theoretical basis (proof) that this holds



Merge Sort

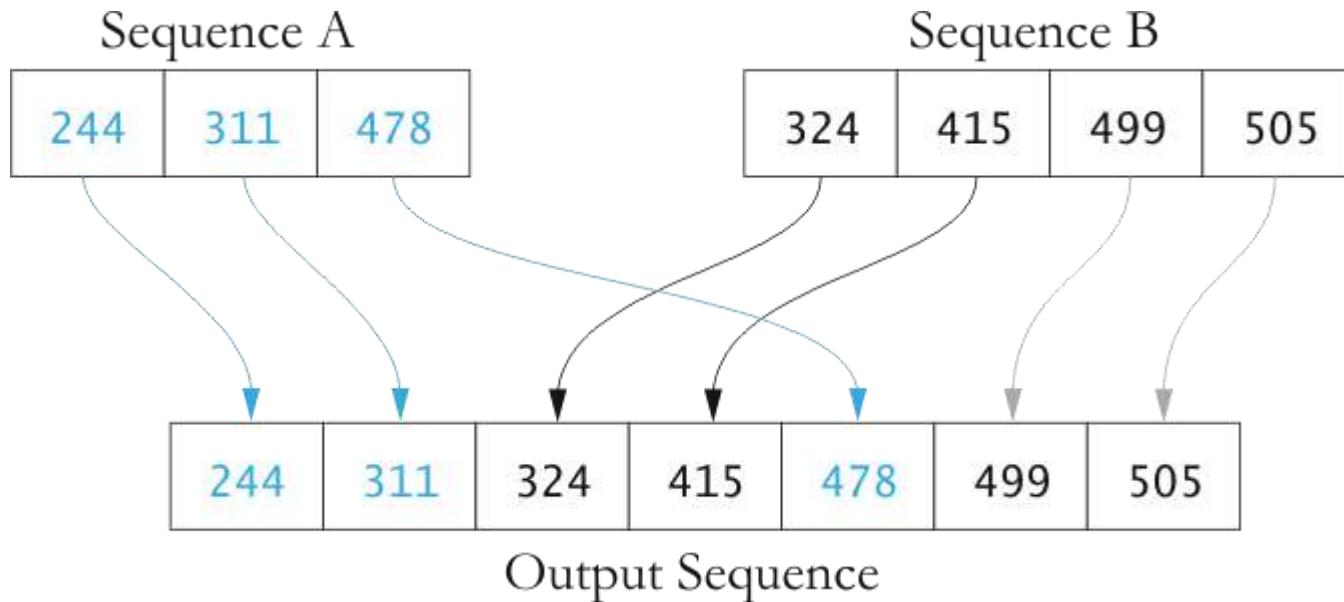
- A merge is a common data processing operation:
 - Performed on two sequences of data
 - Items in both sequences use same compare
 - Both sequences in ordered of this compare
 - **Goal:** Combine the two sorted sequences in one larger sorted sequence
- Merge sort merges longer and longer sequences

Merge Algorithm (Two Sequences)

Merging two sorted sequences:

1. Access the first item from both sequences
2. While neither sequence is finished
 1. Compare the current items of both
 2. Copy smaller current item to the output
 3. Access next item from that input sequence
3. Copy any remaining from first sequence to output
4. Copy any remaining from second to output

Picture of Merge



Analysis of Merge

- Two input sequences, total length n elements
 - Must move each element to the output
 - Merge time is $O(n)$
- Must store both input and output sequences
 - An array cannot be merged in place
 - Additional space needed: $O(n)$

Merge Sort Algorithm

Overview:

- Split array into two halves
- Sort the left half (recursively)
- Sort the right half (recursively)
- Merge the two sorted halves

Merge Sort Algorithm (2)

Detailed algorithm:

- if **tSize** ≤ 1 , return (no sorting required)
- set **hSize** to **tSize** / 2
- Allocate **LTab** of size **hSize**
- Allocate **RTab** of size **tSize** – **hSize**
- Copy elements 0 .. **hSize** – 1 to **LTab**
- Copy elements **hSize** .. **tSize** – 1 to **RTab**
- Sort **LTab** recursively
- Sort **RTab** recursively
- Merge **LTab** and **RTab** into **a**

Merge Sort Example

50	60	45	30	90	20	80	15
----	----	----	----	----	----	----	----

1. *Split array into 4-element arrays*

50	60	45	30
----	----	----	----

2. *Split left array into two 2-element arrays*

50	60
----	----

3. *Split left array (50, 60) into two 1-element arrays*

50	60
----	----

4. *Merge two 1-element arrays into a 2-element array*

45	30
----	----

5. *Split right array from Step 2 into two 2-element arrays*

30	45
----	----

6. *Merge tow 1-element arrays into a 2-element array*

30	45	50	60
----	----	----	----

7. *Merge two 2-element arrays into a 4-element array*

Merge Sort Analysis

- Splitting/copying n elements to subarrays: $O(n)$
- Merging back into original array: $O(n)$
- Recursive calls: 2, each of size $n/2$
 - Their total non-recursive work: $O(n)$
- Next level: 4 calls, each of size $n/4$
 - Non-recursive work again $O(n)$
- Size sequence: $n, n/2, n/4, \dots, 1$
 - Number of levels = $\log n$
 - Total work: $O(n \log n)$

Heapsort

- Merge sort time is $O(n \log n)$
 - **But** requires (temporarily) n extra storage items
- Heapsort
 - Works *in place*: no additional storage
 - Offers same $O(n \log n)$ performance
- Idea (not quite in-place):
 - Insert each element into a priority queue
 - Repeatedly remove from priority queue to array
 - Array slots go from 0 to $n-1$

Algorithm for In-Place Heapsort

- Build heap starting from unsorted array
- While the heap is not empty
 - Remove the first item from the heap:
 - Swap it with the last item
 - Restore the heap property

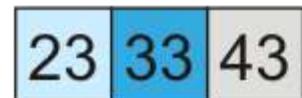
Heapsort Analysis

- Insertion cost is $\log i$ for heap of size i
 - Total insertion cost = $\log(n) + \log(n-1) + \dots + \log(1)$
 - This is $O(n \log n)$
- Removal cost is also $\log i$ for heap of size i
 - Total removal cost = $O(n \log n)$
- Total cost is $O(n \log n)$

Quicksort

- Developed in 1962 by C. A. R. Hoare
- Given a pivot value:
 - Rearranges array into two parts:
 - Left part \leq pivot value
 - Right part $>$ pivot value
- Average case for Quicksort is $O(n \log n)$
 - Worst case is $O(n^2)$

Quicksort Example



Algorithm for Quicksort

first and **last** are end points of region to sort

- if **first < last**
- Partition using **pivot**, which ends in **piv_index**
- Apply Quicksort recursively to left subarray
- Apply Quicksort recursively to right subarray

Performance: $O(n \log n)$ provide **piv_index** not always too close to the end

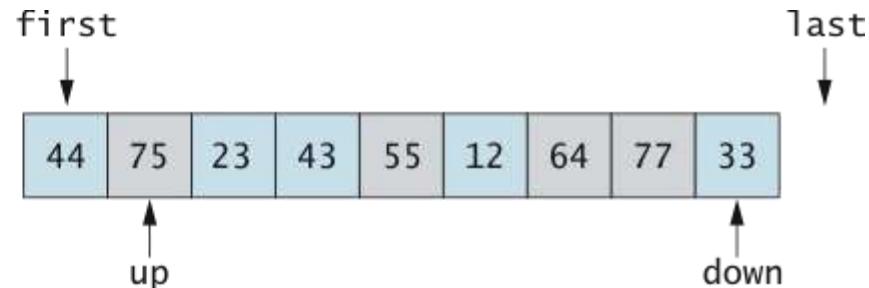
Performance $O(n^2)$ when **piv_index** always near end

Trace of Algorithm for Partitioning

Locating First Values to Exchange

`pivot_value`

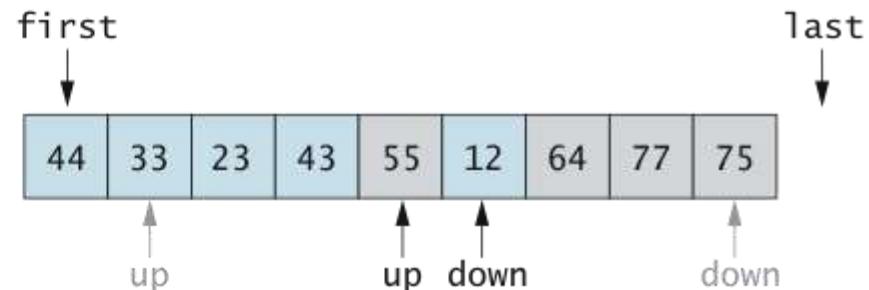
44



After the First Exchange

`pivot_value`

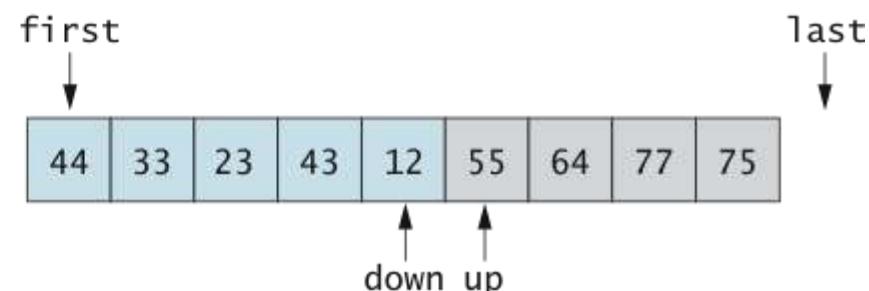
44



After the Second Exchange

`pivot_value`

44

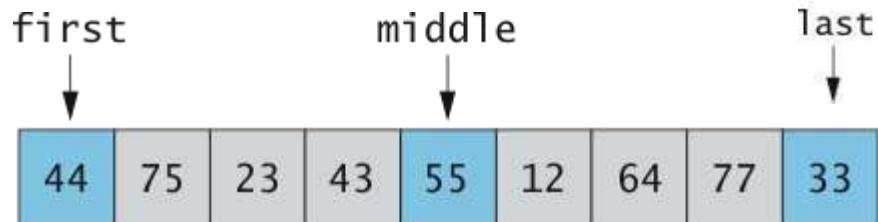


Algorithm for Partitioning

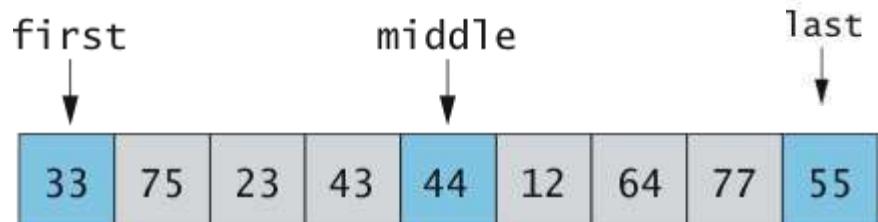
1. Set pivot value to $a[\text{fst}]$
2. Set **up** to fst and **down** to lst
3. do
4. Increment **up** until $a[\text{up}] > \text{pivot}$ or **up** = lst
5. Decrement **down** until $a[\text{down}] \leq \text{pivot}$ or
 down = fst
6. if **up** < **down**, swap $a[\text{up}]$ and $a[\text{down}]$
7. while **up** is to the left of **down**
8. swap $a[\text{fst}]$ and $a[\text{down}]$
9. return **down** as pivIndex

Revised Partitioning Algorithm

- Quicksort is $O(n^2)$ when each split gives 1 empty array
- This happens when the array is already sorted
- Solution approach: pick better pivot values
- Use three “marker” elements: first, middle, last
- Let pivot be one whose value is between the others



After sorting, median is in middle



Testing Sorting Algorithms

- Need to use a variety of test cases
 - Small and large arrays
 - Arrays in random order
 - Arrays that are already sorted (and reverse order)
 - Arrays with duplicate values
- Compare performance on each type of array

Summary of Performance

Number of Comparisons			
	Best	Average	Worst
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Shell Sort	$O(n^{7/6})$	$O(n^{5/4})$	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$

Algorithm Design

“There are only seven movie plots”

-- *unknown*

“There are only ten programs”

-- *unknown*

Algorithm types:

- Brute Force
- Greedy algorithms
 - Scheduling
 - Bin packing
- Divide and conquer
 - Merge sort
 - Closest-Points problem
- Dynamic Programming
 - Coin change
 - Fibonacci
- Randomized algorithms
- Backtracking
 - Maze solving
 - Permutations

Brute Force

- Generate all candidates for solutions and verify all of them individually.
- Brute Force Algorithms are almost always at least quadratic and often exponential and become unrealistic to use even for small size problems.

Greedy algorithms

- Always choose the locally best solution
- Often a bad idea, but does work sometimes
- Some of the most used graph algorithms are greedy algorithms.

Scheduling

Job	Time
j_1	15
j_2	8
j_3	3
j_4	10

Figure 10.1 Jobs and times

Scheduling

Job	Time
j_1	15
j_2	8
j_3	3
j_4	10

Figure 10.1 Jobs and times

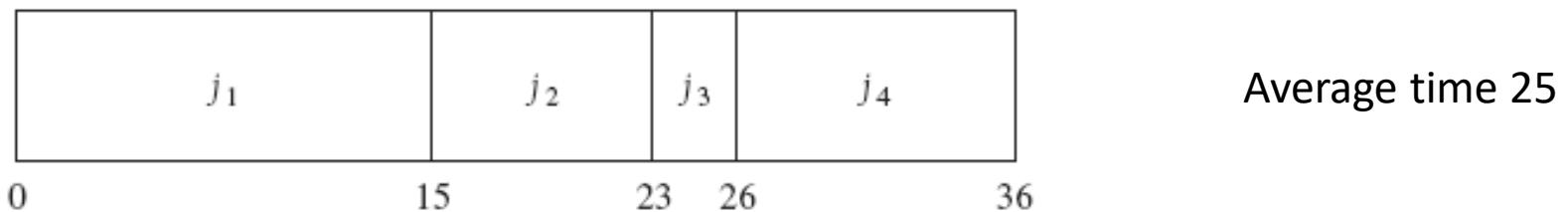
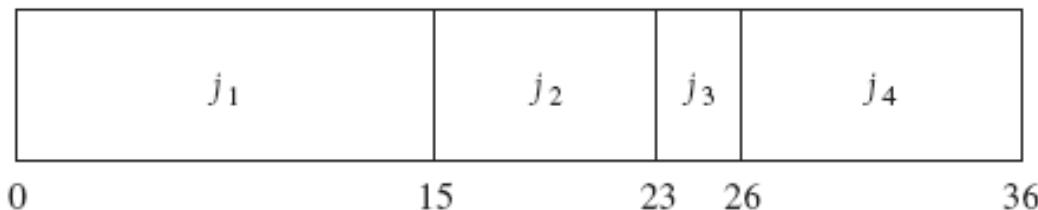


Figure 10.2 Schedule #1

Scheduling

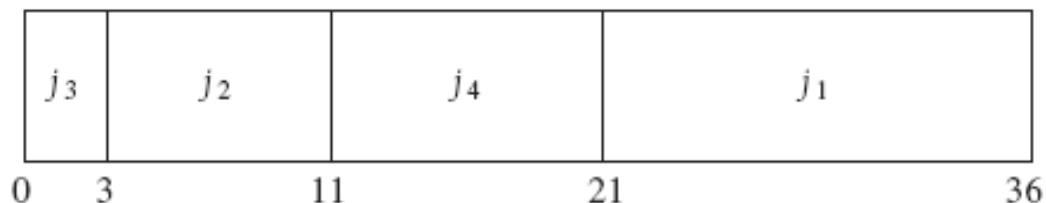
Job	Time
j_1	15
j_2	8
j_3	3
j_4	10

Figure 10.1 Jobs and times



Average time 25

Figure 10.2 Schedule #1



Average time 17,75

Figure 10.3 Schedule #2 (optimal)

Bin Packing

- On / Off line

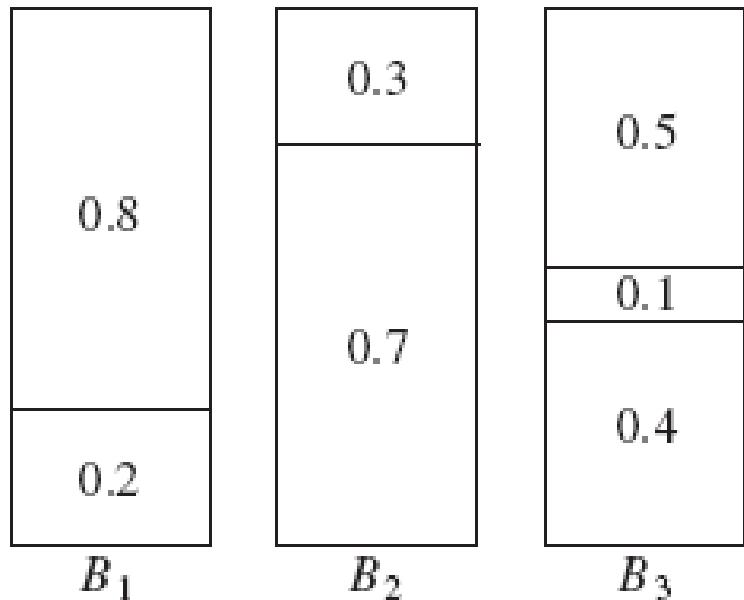


Figure 10.20 Optimal packing for 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8

Bin Packing

- Next fit

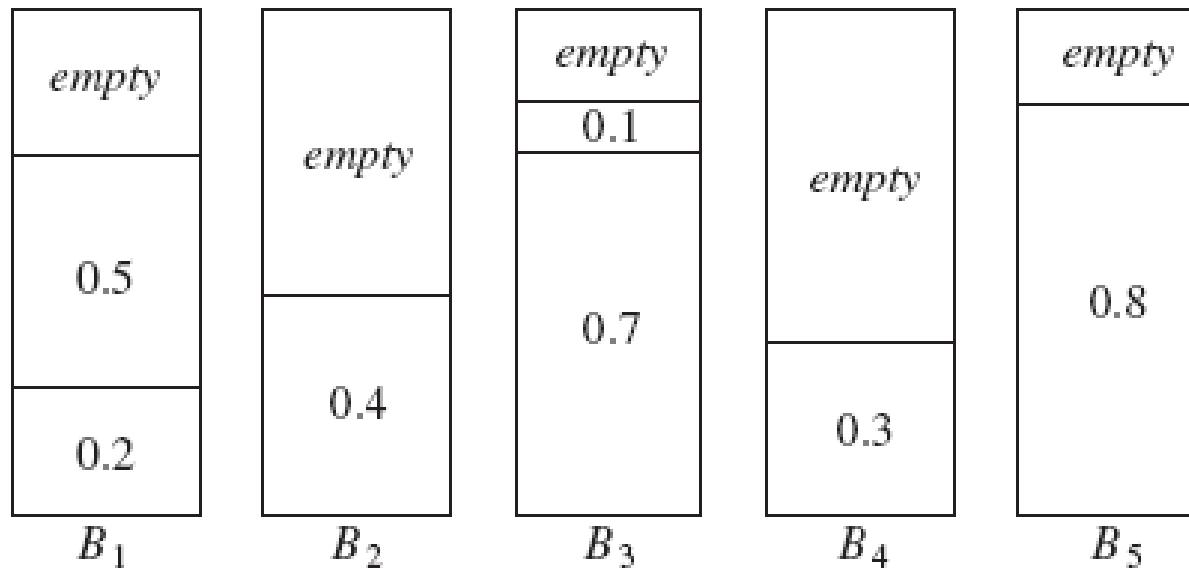


Figure 10.21 Next fit for 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8

Bin Packing

- First fit:

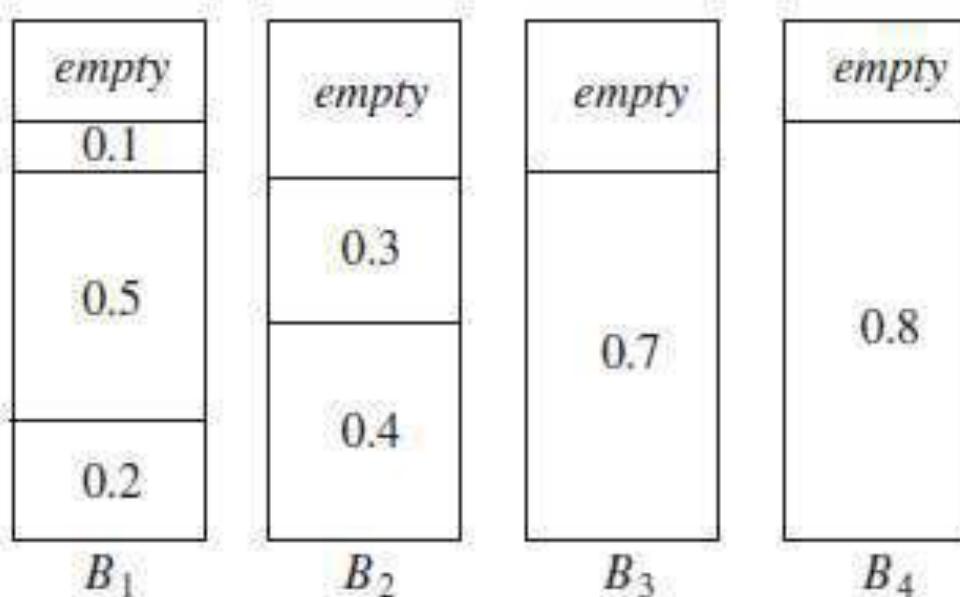


Figure 10.24 First fit for 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8

Bin Packing

- First fit decreasing (off-line)

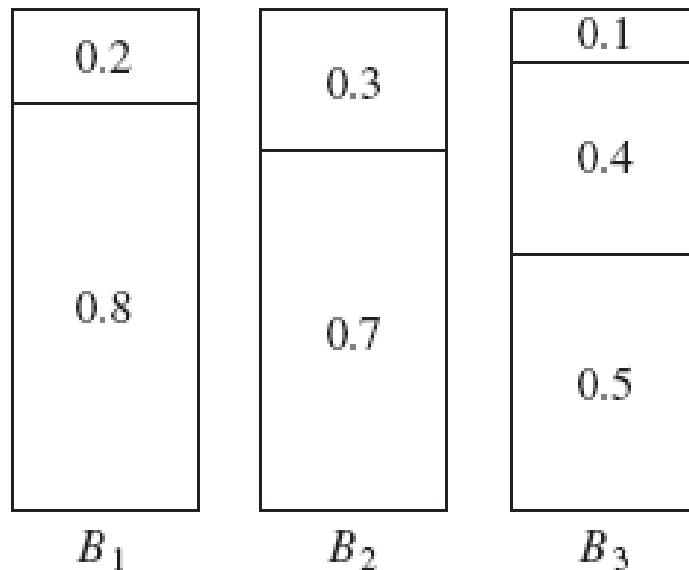


Figure 10.27 First fit for 0.8, 0.7, 0.5, 0.4, 0.3, 0.2, 0.1

Bin Packing

- Best fit

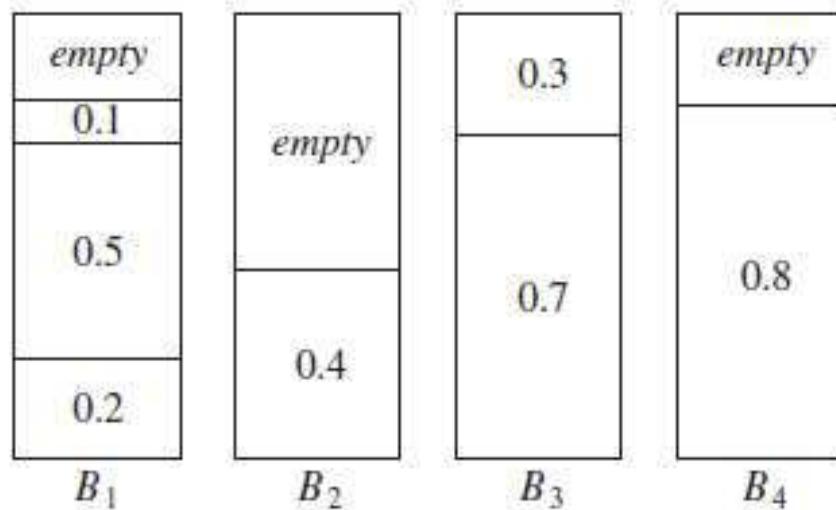


Figure 10.26 Best fit for 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8

Divide and conquer

- Closely related to recursion
- The idea is to split the problem in smaller problems, that (recursively) might be solved with the same algorithm as the main problem.
- Classical examples are the Towers of Hanoi, MergeSort and QuickSort.
- Divide and Conquer can utilize a parallel platform (multiprocessor, multi core systems etc.).

Divide and conquer: time complexity

- Not a trivial task (due to recursion)
- A simple example: Mergesort

Divide and conquer: time complexity

- Not a trivial task (due to recursion)
 - A simple example: Mergesort
-
- For each recursive step:

$$t(n) = 2t(n/2) + O(n)$$

Divide and conquer: time complexity

- Not a trivial task (due to recursion)
- A simple example: Mergesort
- For each recursive step:

$$t(n) = 2t(n/2) + O(n)$$

This can be generalized to

$$t(n) = at(n/b) + f(n)$$

The Master Theorem

If $f(n) = O(1)$

$$t(n) = at(n/b) + f(n)$$

- The master theorem recognizes two cases:

If $a = 1$ and $b > 1$, the solution becomes $t(n) = O(\log n)$

If $a > 1$ and $b > 1$, the solution becomes $t(n) = O(n^{\log b a})$

The Master Theorem

If $f(n) = O(n)$

$$t(n) = at(n/b) + f(n)$$

- The master theorem recognizes two cases:
 - if $a > b$, the solution becomes $t(n) = O(n^{\log b a})$
 - if $a = b$, the solution becomes $t(n) = O(n \log n)$
 - if $a < b$, the solution becomes $t(n) = O(n)$

Application of divide and conquer

- Closest-Points problem

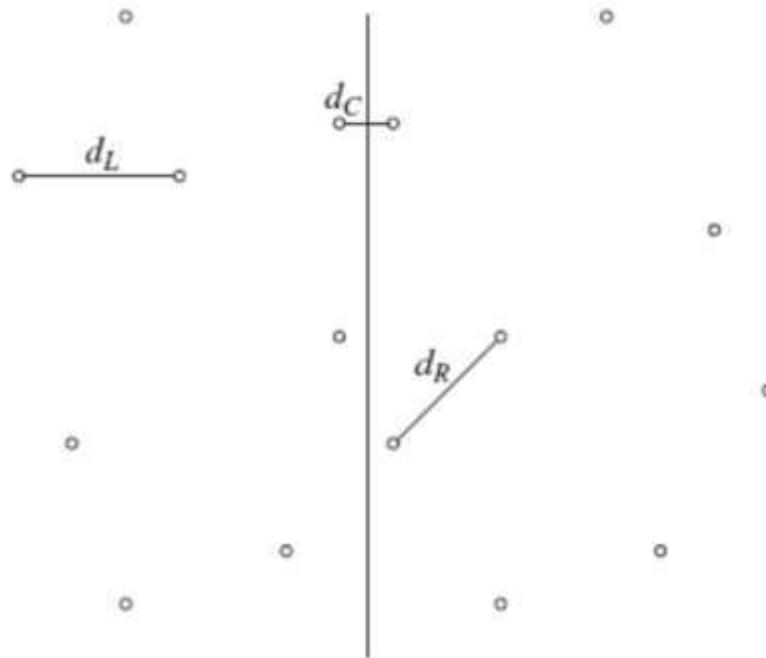


Figure 10.30 P partitioned into P_L and P_R ; shortest distances are shown

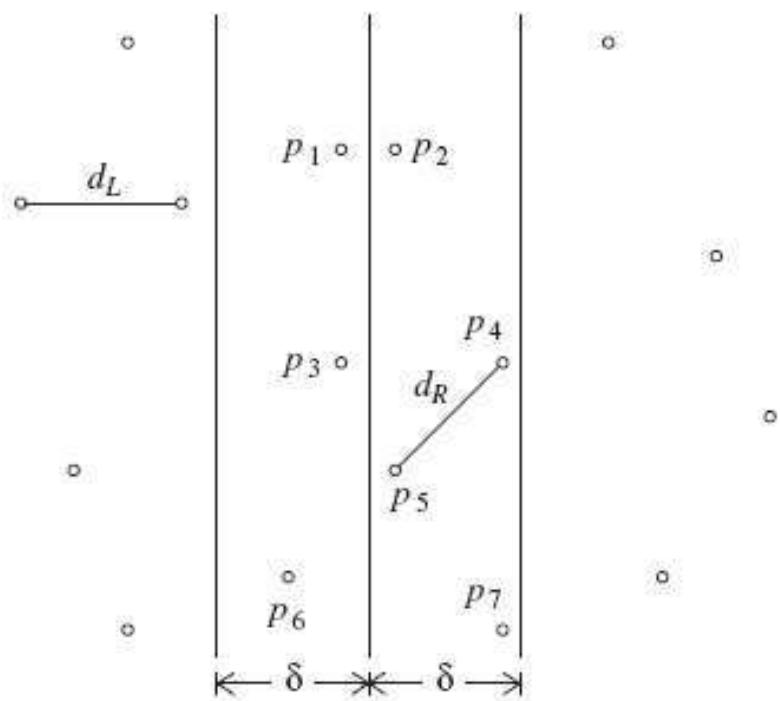


Figure 10.31 Two-lane strip, containing all points considered for d_C strip

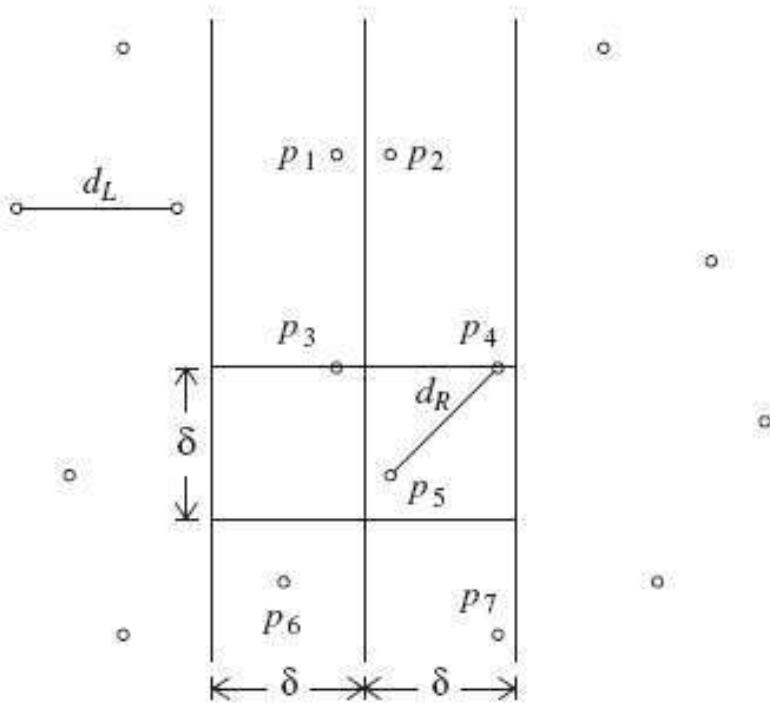


Figure 10.34 Only p_4 and p_5 are considered in the second for loop

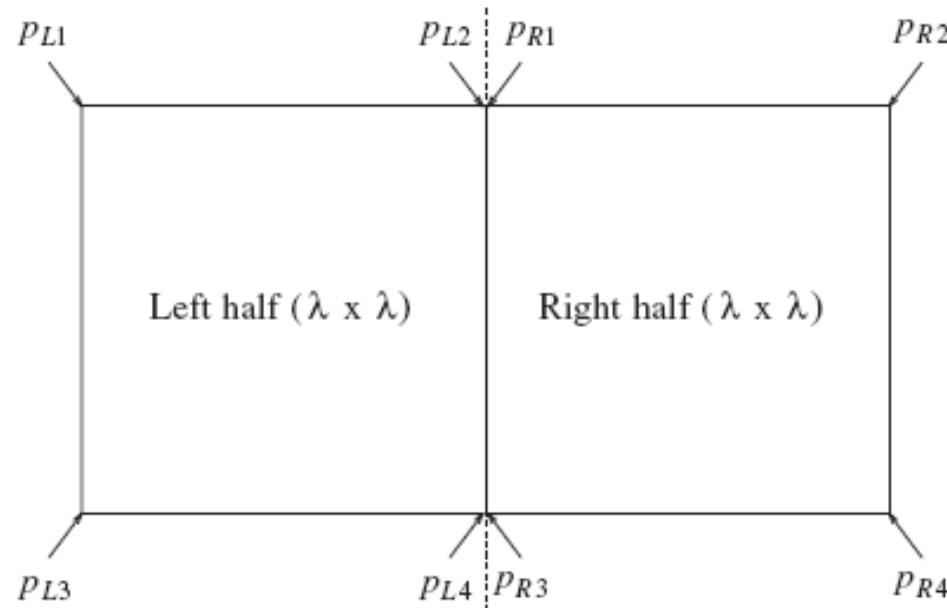


Figure 10.35 At most eight points fit in the rectangle; there are two coordinates shared by two points each

Randomized algorithms aka. Monte Carlo Algorithms

- A choice in a Monte Carlo algorithm is made randomly.
- They are surprisingly good for solving some optimization problems.
- Quick sort has elements of randomness.
- Can be used int primality testing.
- Randomness be used to counter “anti solver algorithm” eg. in sudoku solvers.

Dynamic Programming Algorithms

"Never do the same calculation twice".

In dynamic programming the result is stored in a table every time a calculation is done/a subproblem is solved.

If the same calcuation/problem solving is later needed it is just a matter of table lookup.

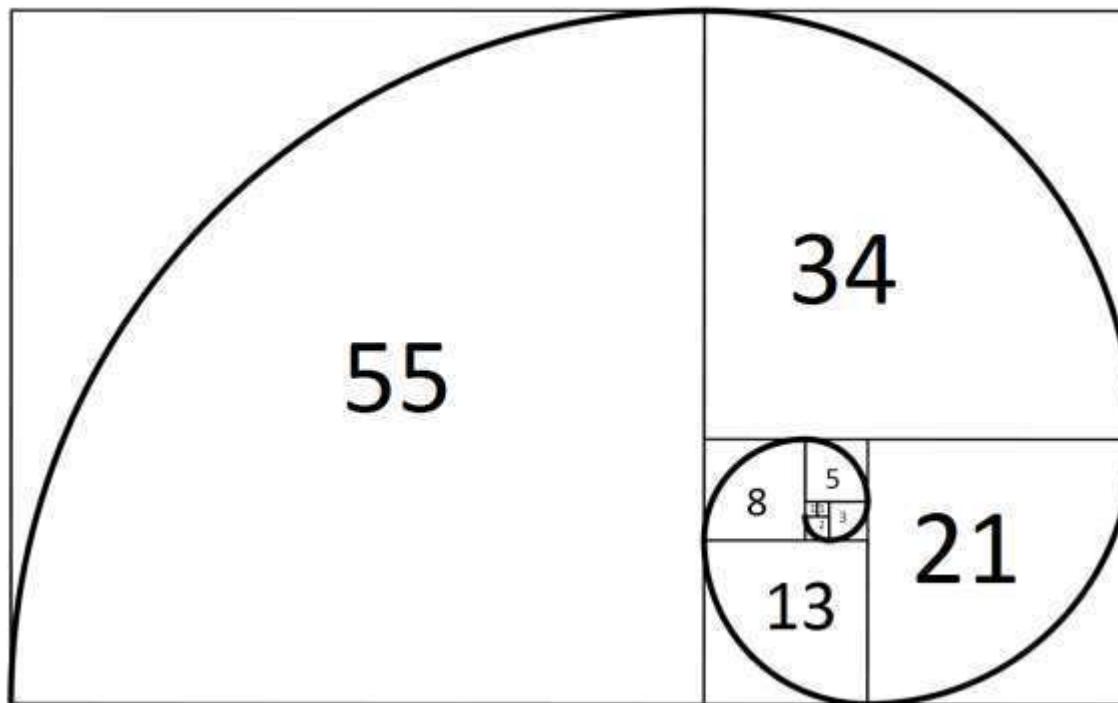
It can be used to repair the recursive version of the Fibonacci problem.

Fibonacci and Dynamic programming

- Dynamic programming is useful when sub-problems share sub-sub-problems:
 - Fib(8) has the sub-problems Fib(7) and Fib(6)
 - Fib(7) and Fib(6) both has the sub-sub-problem Fib(4)
- Dynamic programming solves each sub-problem once, and stores the answer somehow for later re-use.

Example

- Fibonacci sequence
- 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144

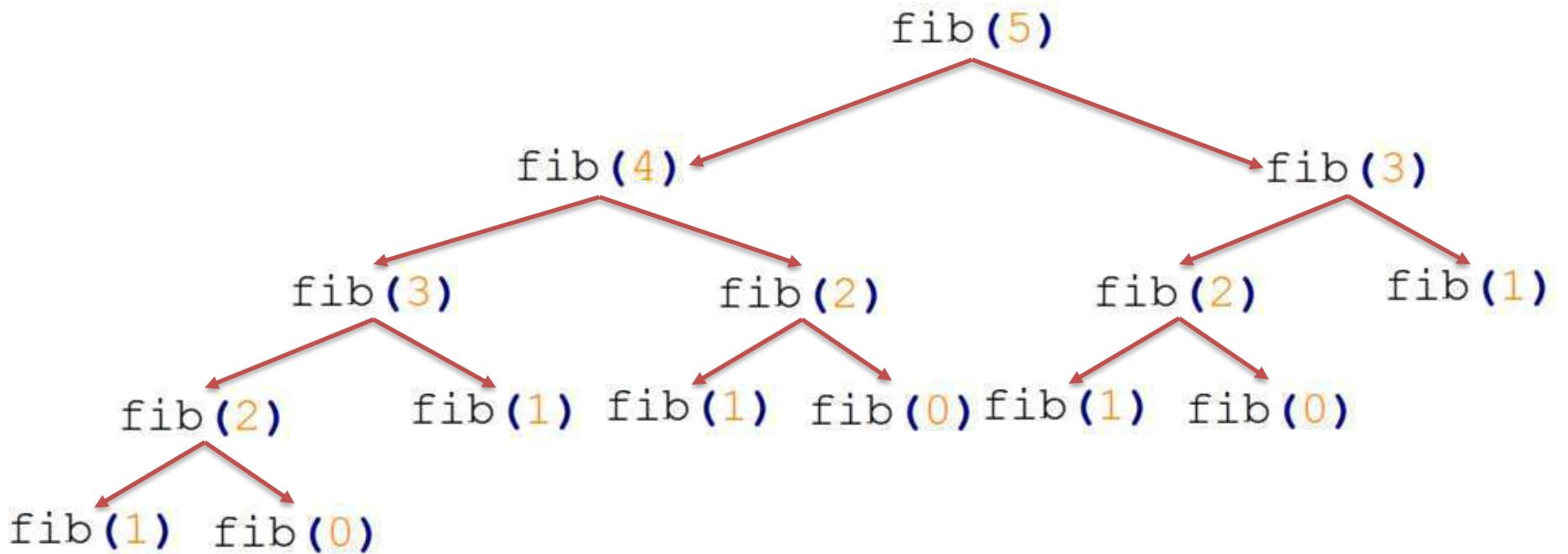


Implementation

- Good example of recursion

```
/** Recursive method to calculate Fibonacci numbers
 * (in RecursiveMethods.java).
 * pre: n >= 1
 * @param n The position of the Fibonacci number being calculated
 * @return The Fibonacci number
 */
public static int fibonacci(int n) {
    if (n <= 2)
        return 1;
    else
        return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Recursive call tree



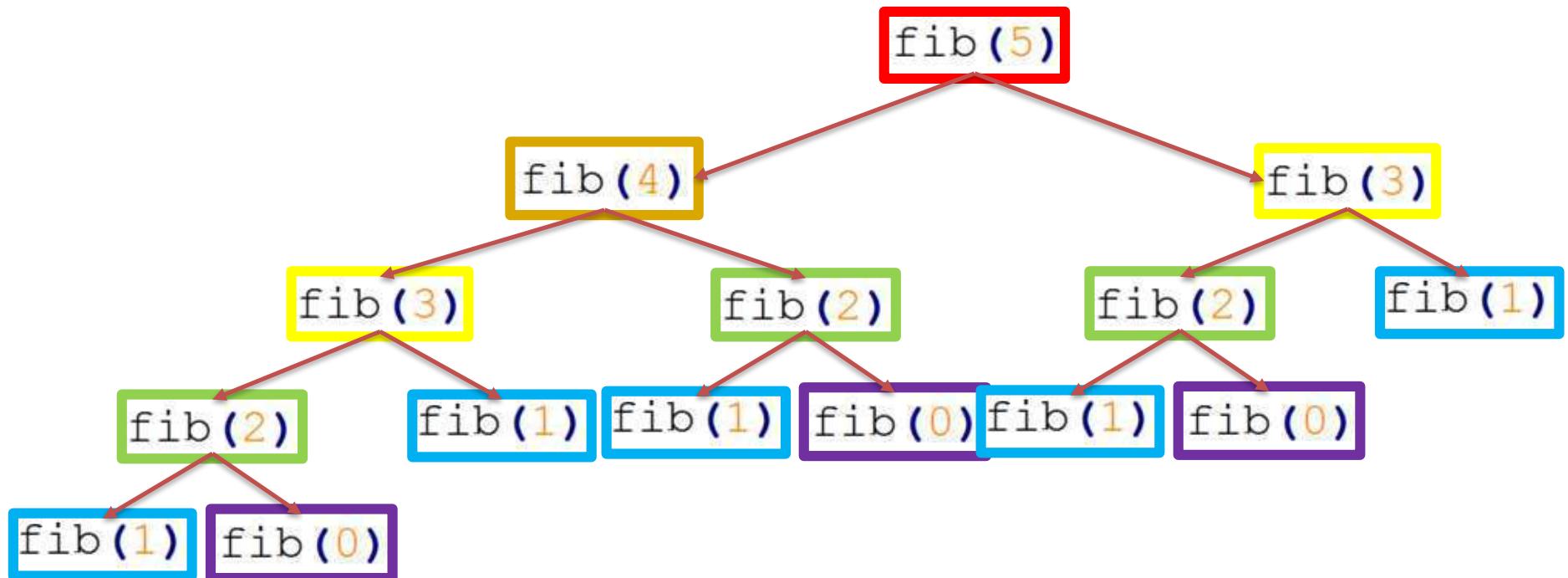
Execution time

Time to compute Fib (i)

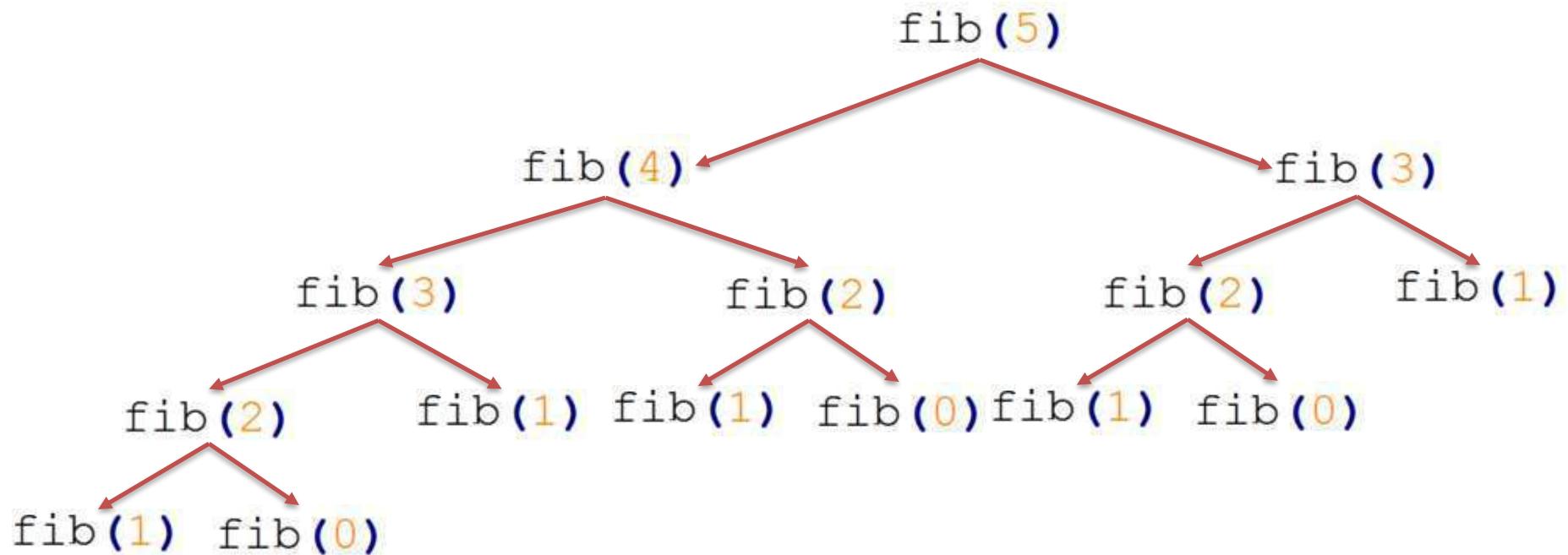


Recursive call tree

Notice a lot of “duplicate” calls/computations.



As we increase i in $\text{Fib}(i)$, the call tree grows exponentially
The call trees become huge!



What if we stored/reused calculations?

- Whenever I calculate $\text{Fib}(i)$, I store the value, and can reuse it next time I need $\text{Fib}(i)$:

```
public static HashMap<Integer, Long> map;  
  
public static long fib(int a) {  
    if(map.get(a) != null) {  
        return map.get(a);  
    }  
    if(a <= 0) return 0L;  
    if(a == 1) return 1L;  
  
    long res = fib(a-1) + fib(a-2);  
    map.put(a, res);  
    return res;  
}
```



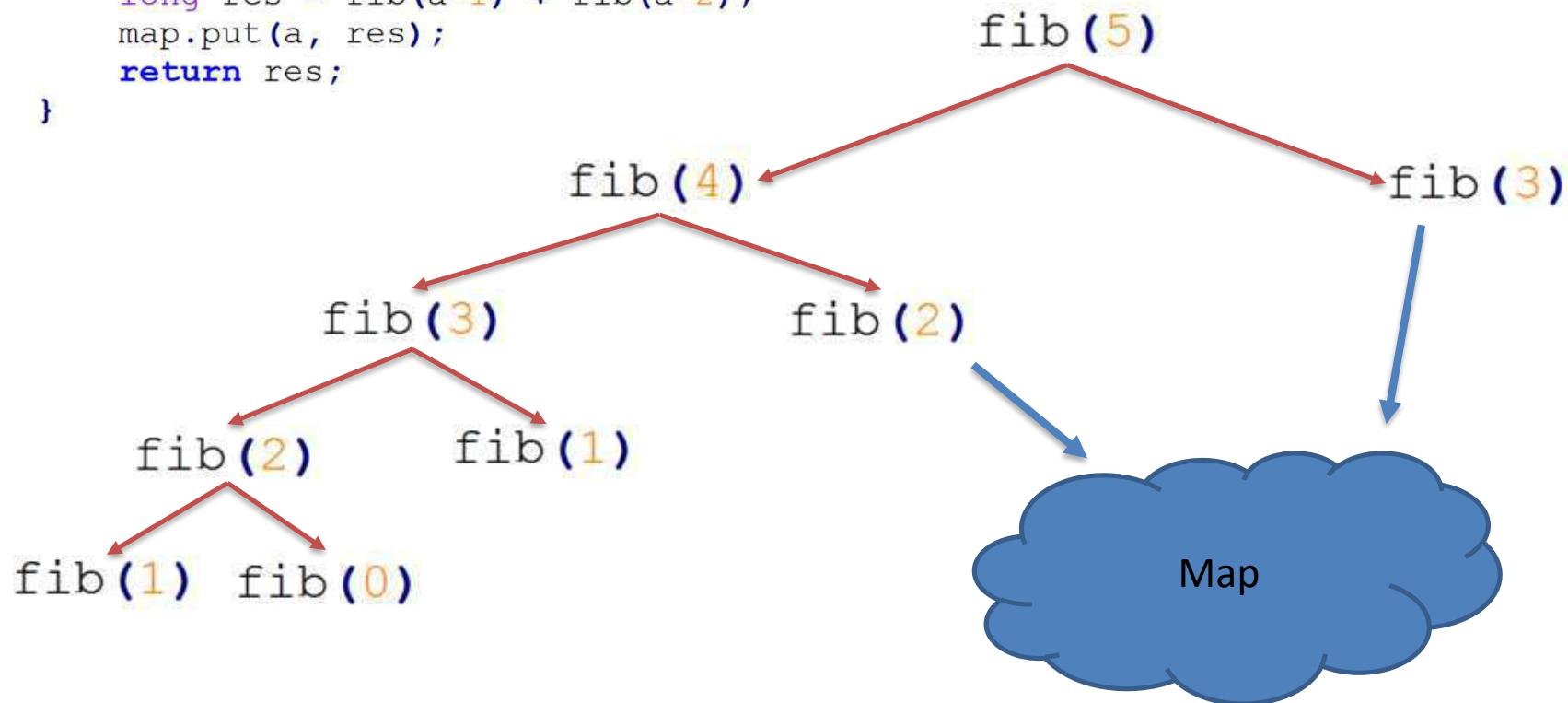
a	Fib(a)
0	0
1	1
2	1
3	2
4	3
5	5
6	8
7	13
8	21
9	34

What if we stored/reused calculations?

```
public static HashMap<Integer, Long> map;

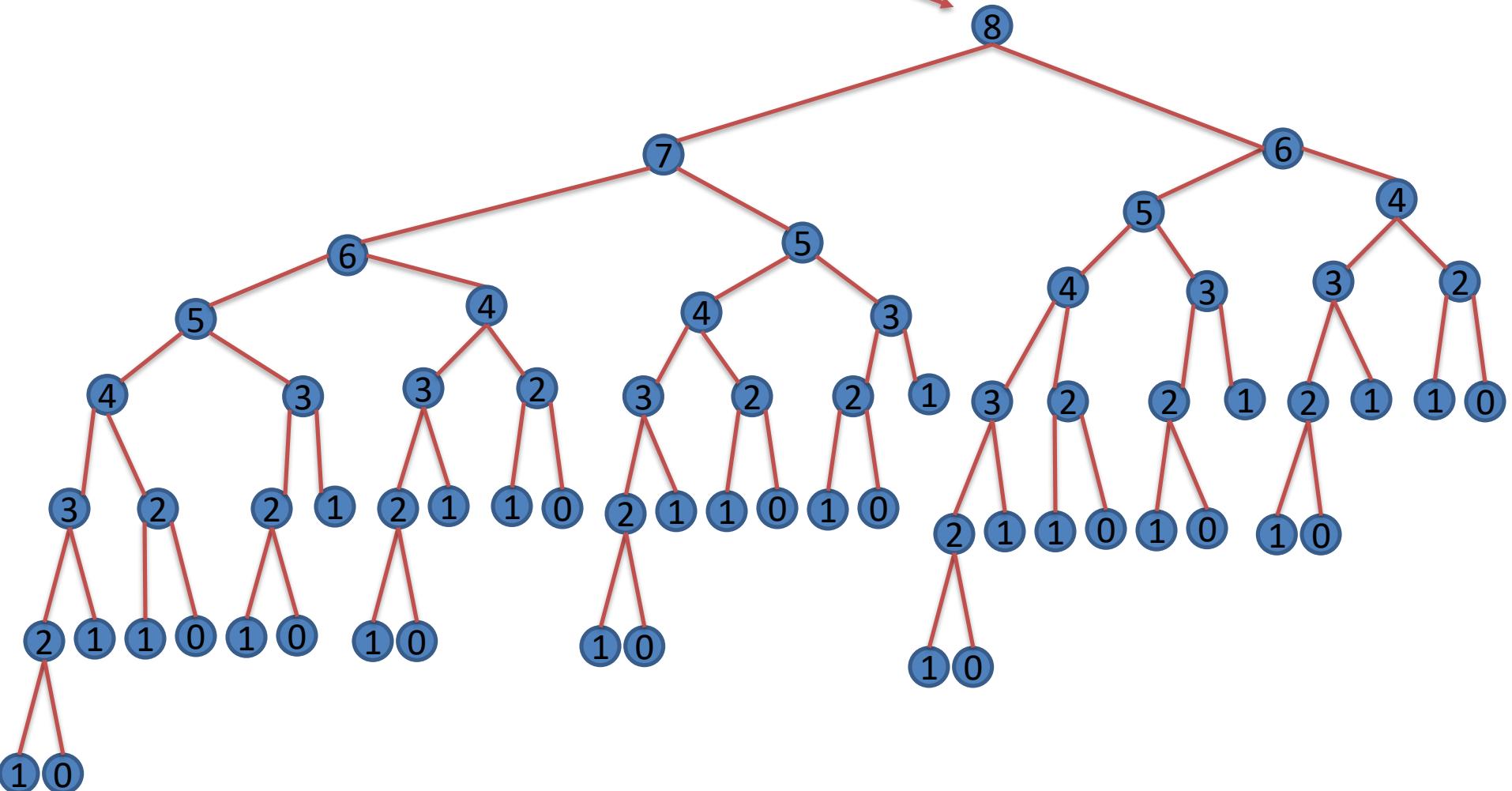
public static long fib(int a) {
    if(map.get(a) != null) {
        return map.get(a);
    }
    if(a <= 0) return 0L;
    if(a == 1) return 1L;

    long res = fib(a-1) + fib(a-2);
    map.put(a, res);
    return res;
}
```



Number of recursive calls

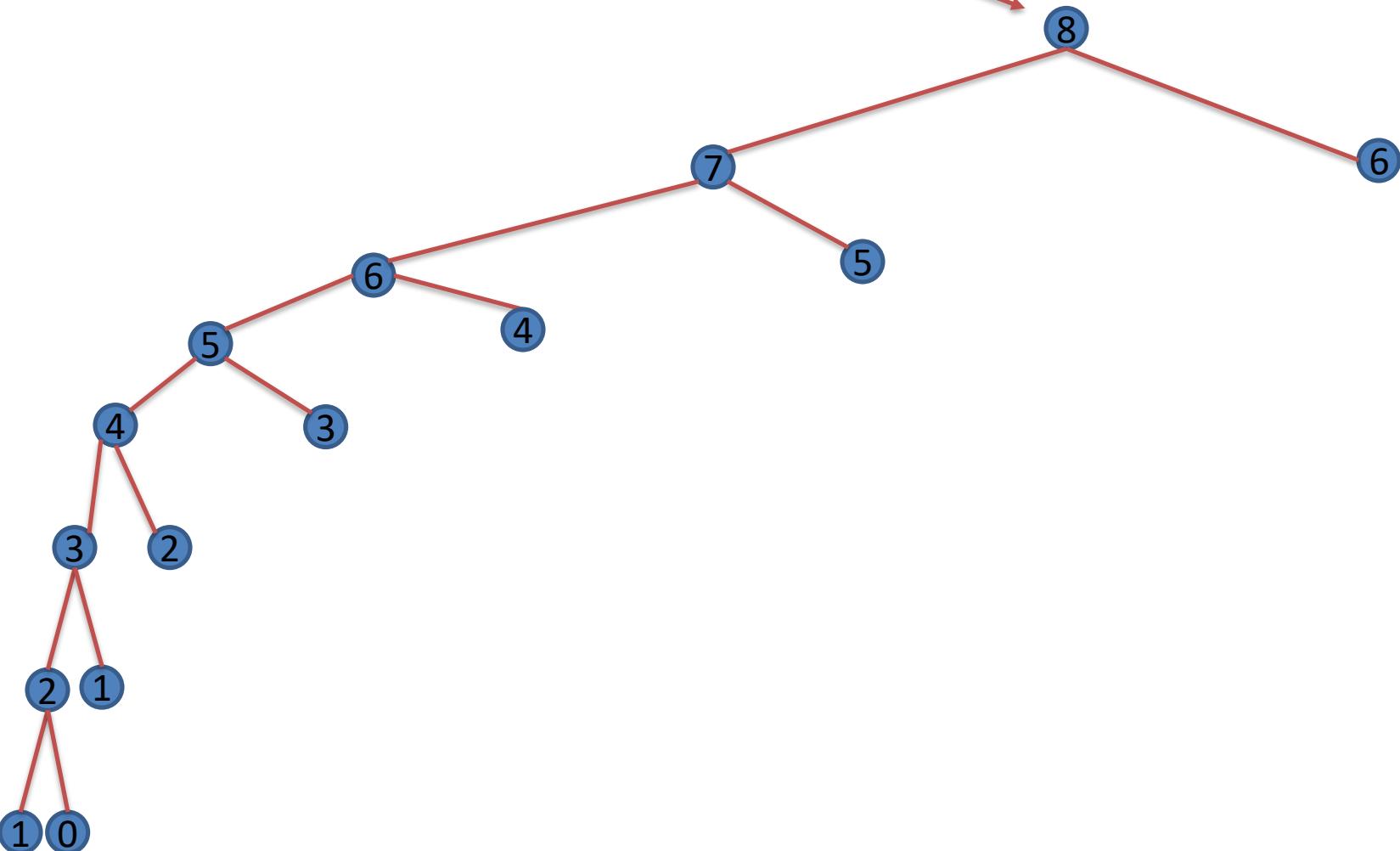
Fib (8)



67 calls

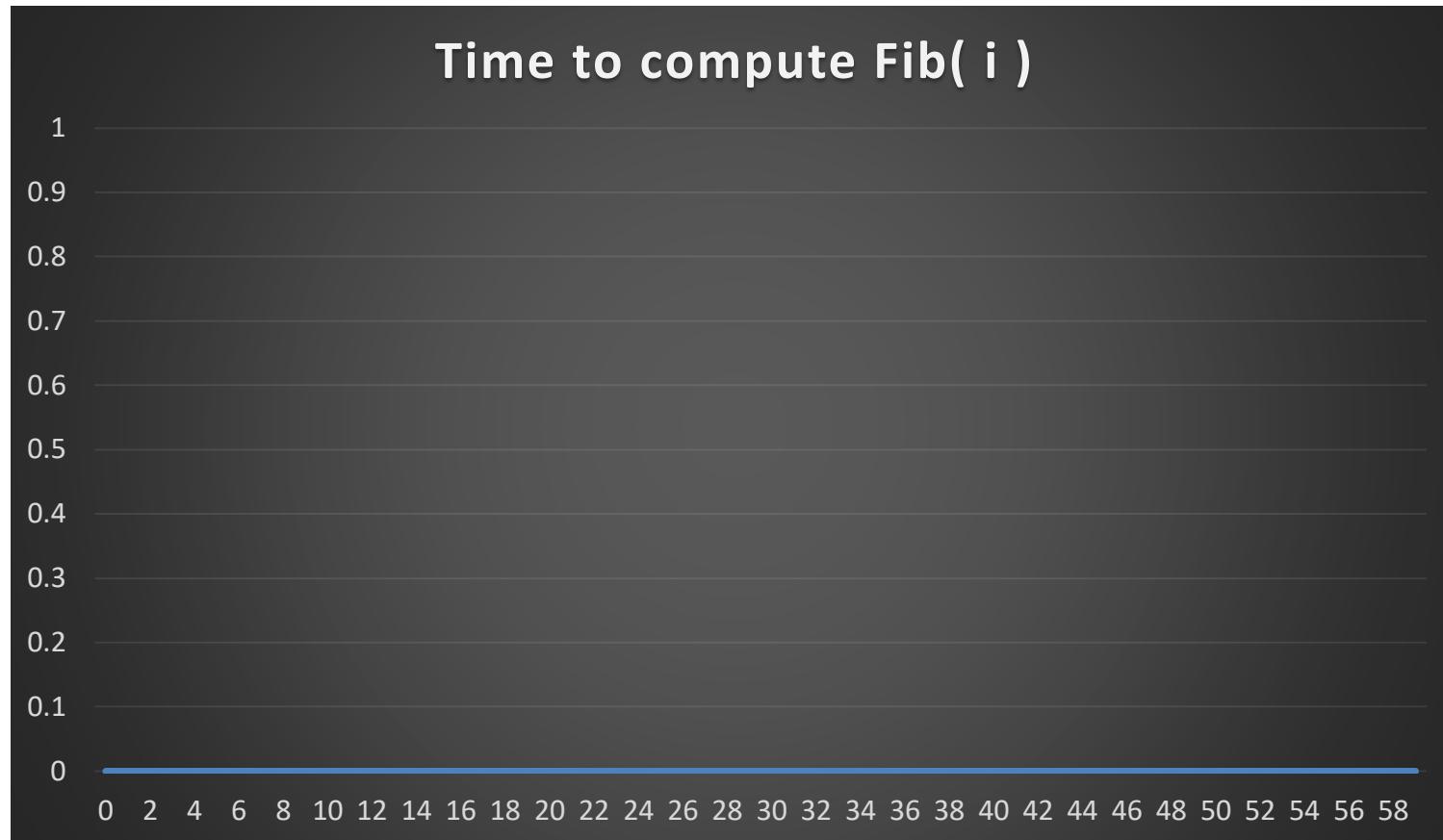
Number of recursive calls

Fib (8)



67 → 15 calls

Execution time



Fib(52) : 0 ms compared to 124,385 ms

Backtracking Algorithms

In backtracking recursion is used to remember earlier states of the problem solving, so it is possible to go back and restart the problem solving from this point.

In every step a choice has to be made before doing the next step.

A solving attempt ends as soon as it is clear, that it will not lead to a solution.

Finding a way through a maze

- Possibly the oldest backtracking algorithm

Problem:

- Generate all permutations of the three letters A, B, and C

Solution:

ABC

ACB

BAC

BCA

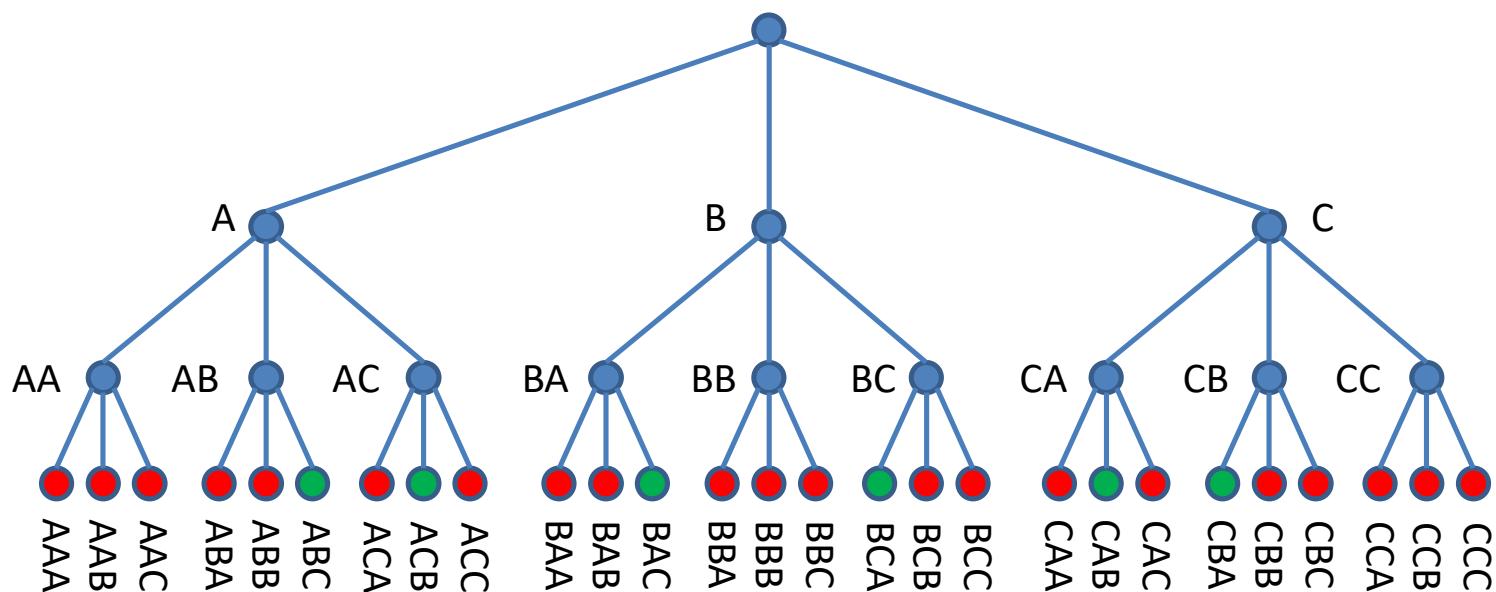
CAB

CBA

Brute force algorithm

```
char c[] = new char[3];
...
for( c[0] = 'a'; c[0] <= 'c'; ++c[0] )
    for( c[1] = 'a'; c[1] <= 'c'; ++c[1] )
        for( c[2] = 'a'; c[2] <= 'c'; ++c[2] )
            if( c[0] != c[1] && c[0] != c[2] && c[1] != c[2] )
                System.out.println( "" + c[0] + c[1] + c[2] );
```

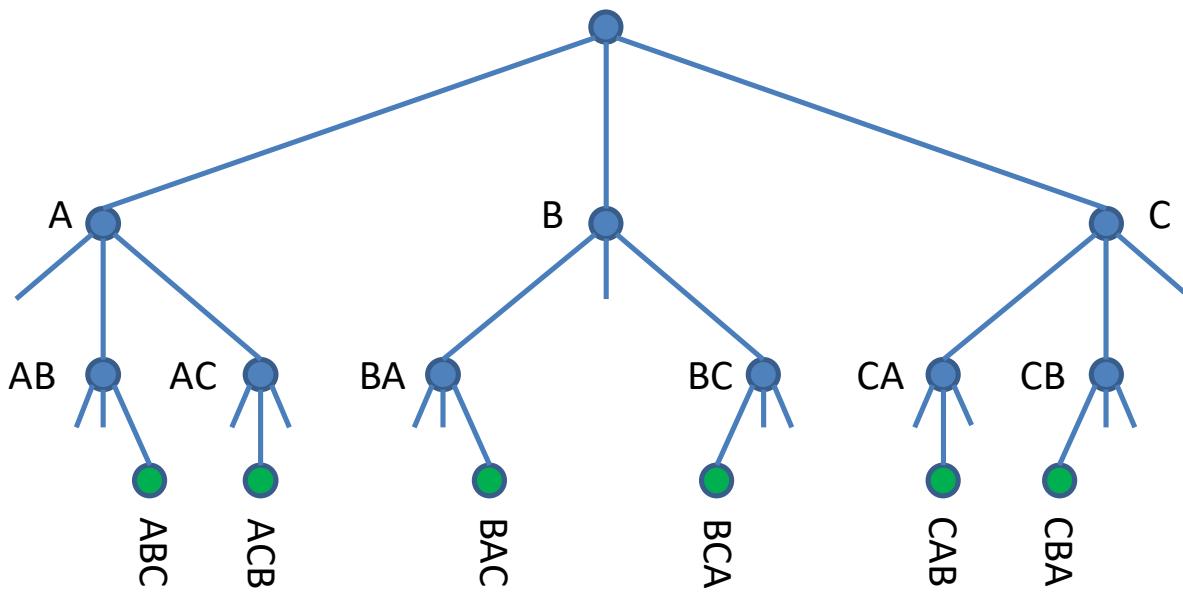
Brute force tree



Stopping useless attempts early

```
char c[] = new char[3];
...
for( c[0] = 'a'; c[0] <= 'c'; ++c[0] )
    if( OK( 0 ) )
        for( c[1] = 'a'; c[1] <= 'c'; ++c[1] )
            if( OK( 1 ) )
                for( c[2] = 'a'; c[2] <= 'c'; ++c[2] )
                    if( OK( 2 ) )
                        System.out.println( "" + c[0] + c[1] + c[2] );
...
boolean OK( int position )
{
    for( int i = 0; i < position; ++i )
        if( c[i] == c[position] )
            return false;
    return true;
}
```

Reduced tree



Backtracking version

```
char c[] = new char[3];
tryPosition( 0 );
...
void tryPosition( int position )
{
    if( position == 3 )
        System.out.println( "" + c[0] + c[1] + c[2] );
    else
        for( c[position] = 'a'; c[position] <= 'c'; ++c[position] )
            if( OK( position ) )
                tryPosition( position + 1 );
}
...
boolean OK( int position )
{
    for( int i = 0; i < position; ++i )
        if( c[i] == c[position] )
            return false;

    return true;
}
```

General backtracking template

```
tryStep( step )
{
    if( done( step ) )
        reportResult();
    else
        for all possible actions
            if( OK( action ) ) {
                doAction();
                tryStep( next( step ) );
                undoAction();
            }
}
```

Heuristic Algorithms

Many of the previous algorithm types might be improved using heuristics, which means that in each step "hints" will be used to make better choices.

The heuristics could come from guessing, analyzing, experience, statistics etc.

Heuristics are often used to prioritize the choices in backtracking.

Heuristics can help solve exponential problems in almost polynomial time.

Graphs

Topic Outline

- Graph Terminology
- The Graph ADT and Edge Class
- Implementing the Graph ADT
- Traversals of Graphs
- Application of Graph Traversals
- Algorithms Using Weighted Graphs

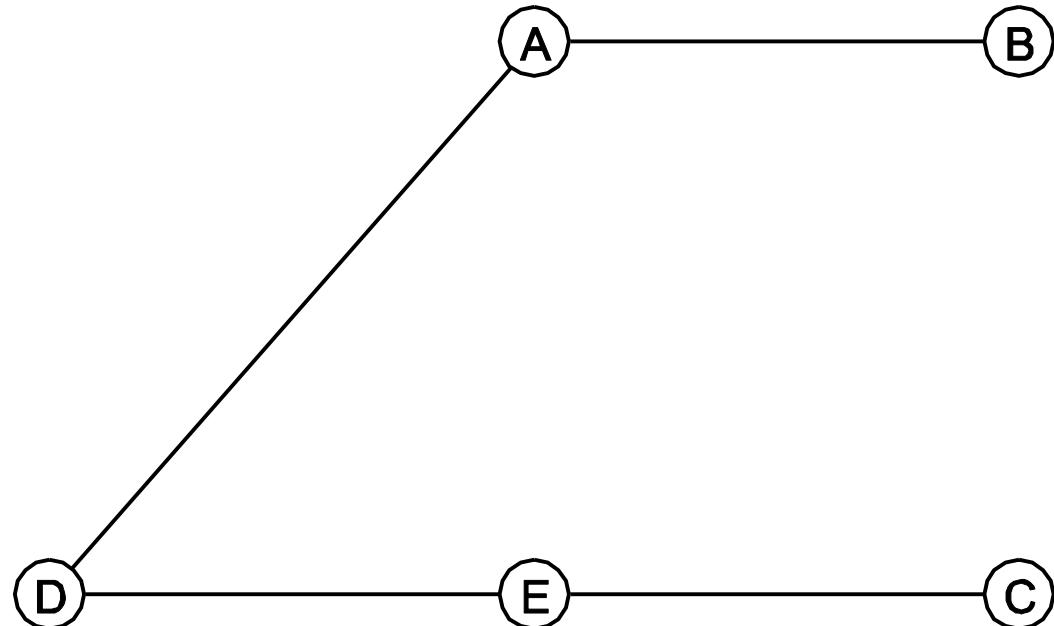
What is a Graph

- A graph is a set of vertices and a set of edges that connect pairs of distinct vertices.
 - Mathematically we say:
 $G = (V, E)$ where
 $E \subset V \times V$
- A graph may be directed or undirected.
 - In an undirected graph, $(u, v) \in E$ iff $(v, u) \in E$
 - This implies that there are two edges (u, v) and (v, u) in an undirected graph. This is not correct, instead the edges of an undirected graph are simply pairs, not ordered pairs.
- Visually we represent the vertices as points (or labeled circles) and the edges as lines joining the vertices.

Example: A Undirected Graph

$$V = \{A, B, C, D, E\}$$

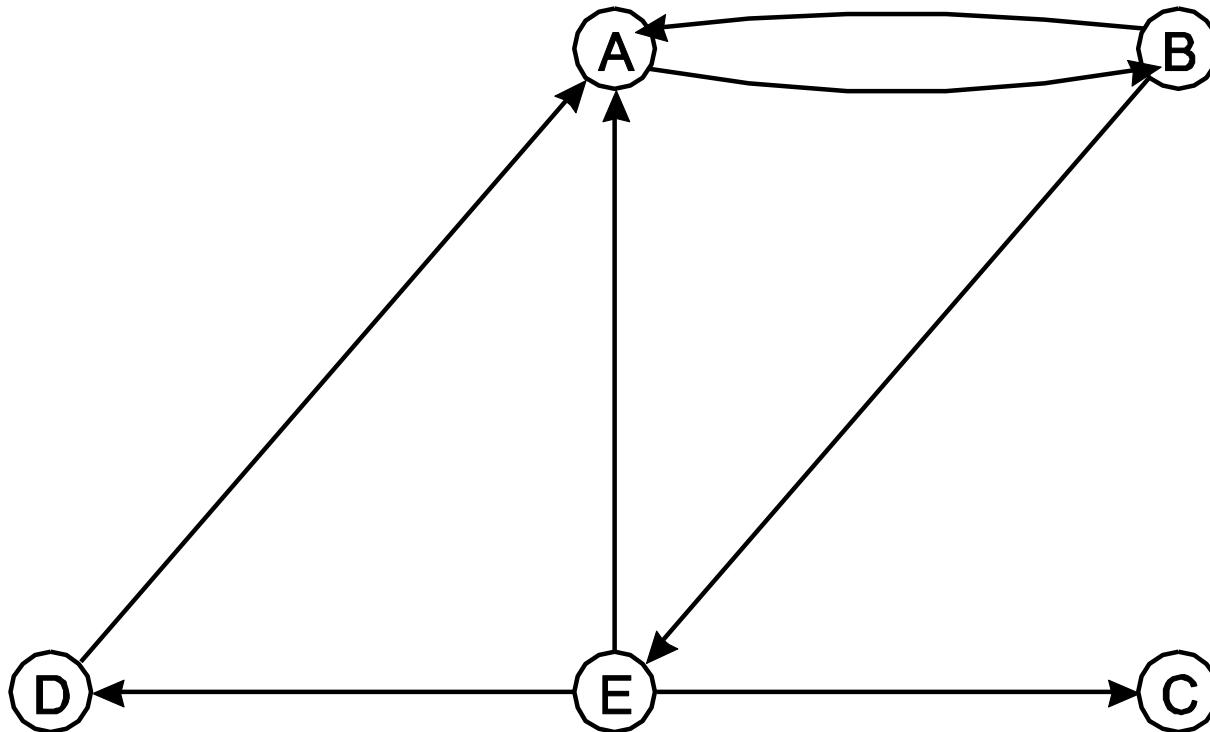
$$E = \{\{A, B\}, \{A, D\}, \{C, E\}, \{D, E\}\}$$



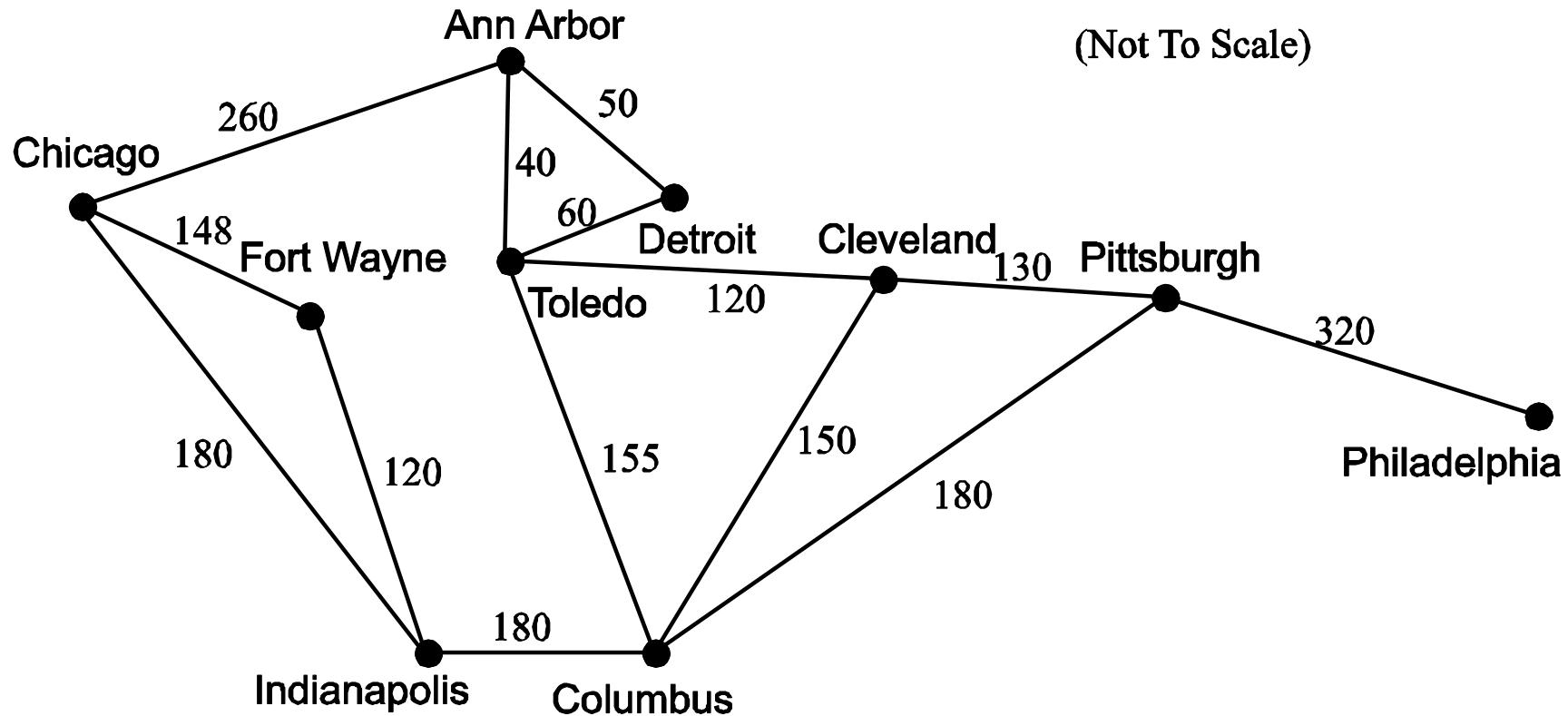
A Directed Graph

$$V = \{A, B, C, D, E\}$$

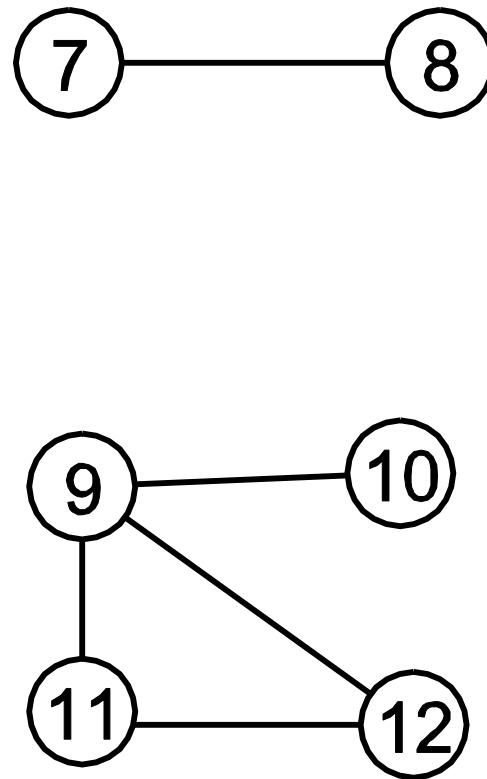
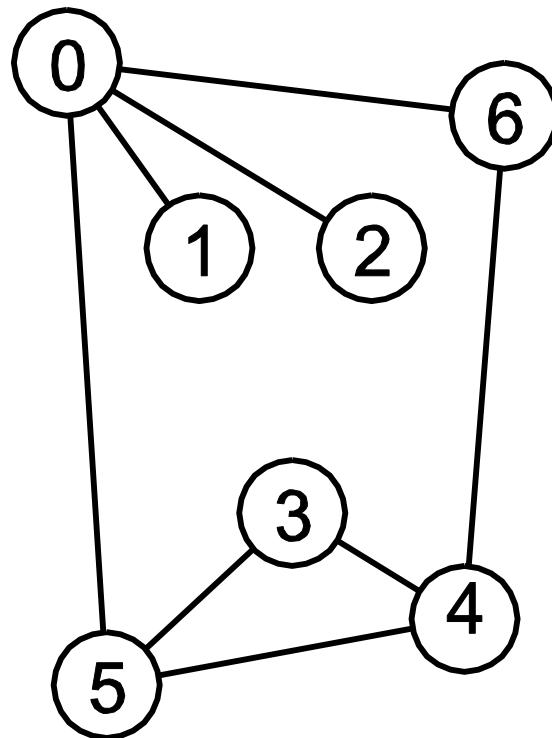
$$E = \{(A, B), (B, A), (B, E), (D, A), (E, A), (E, D), (E, C)\}$$



Weighted Graph



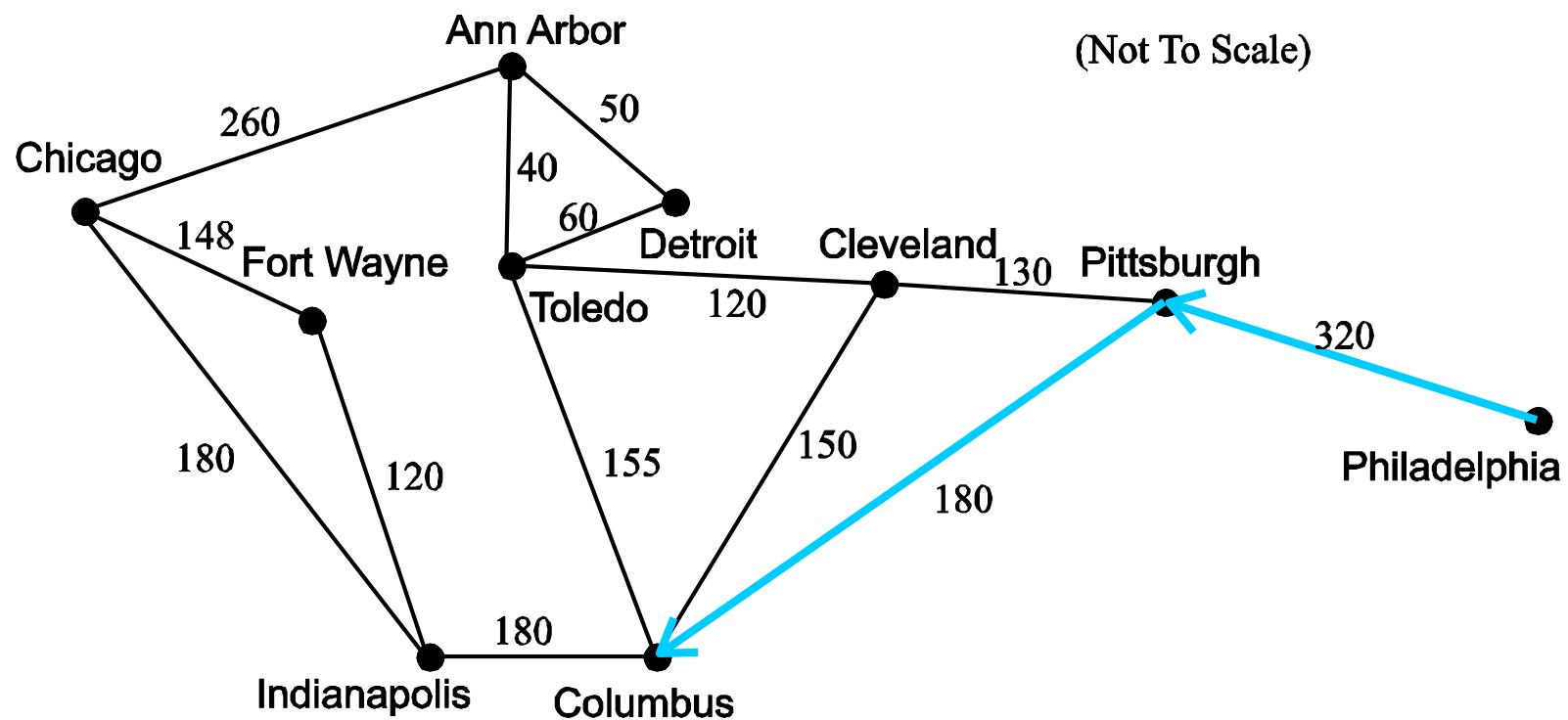
Disconnected Graph



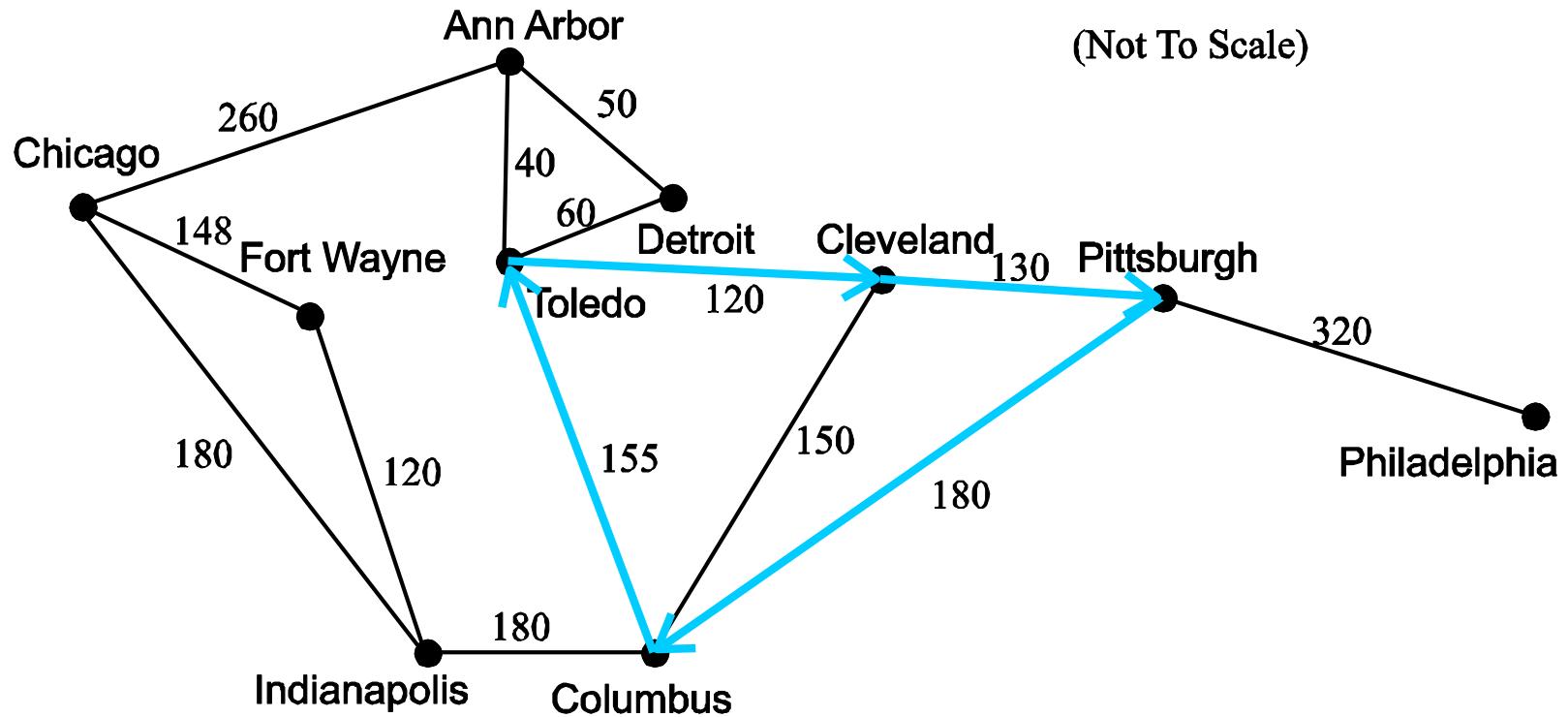
Graph Definitions

- Path
 - A sequence of vertices in which each successive vertex is adjacent to its predecessor.
 - A simple path the vertices and edges are distinct.
 - A cycle is a simple path in which the first and final vertices are the same.
- Connected graph
 - A graph if there is a path from every vertex to every other vertex.
 - A graph that is not connected, consists of connected components.

Example of a Path

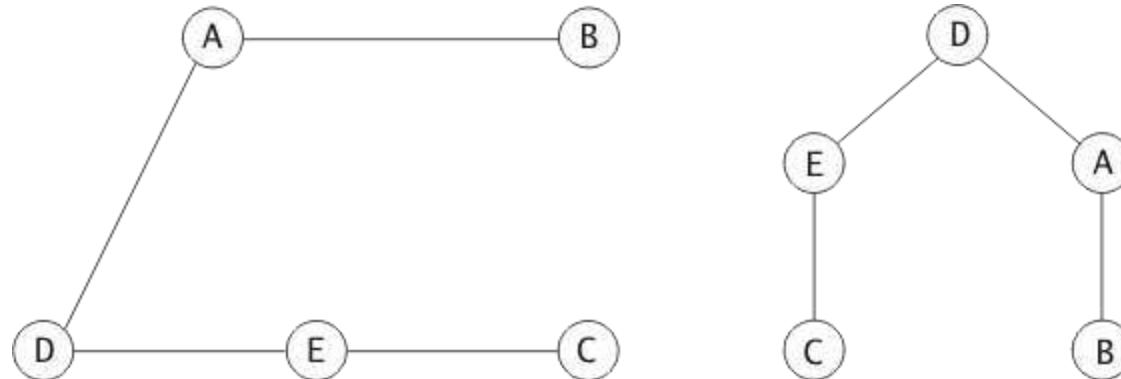


Example of a Cycle



Relationship Between Graphs and Trees

- A tree is a special case of a graph.
- Any connected graph which does not contain cycles can be considered a tree.
- Any node can be chosen to be the root.



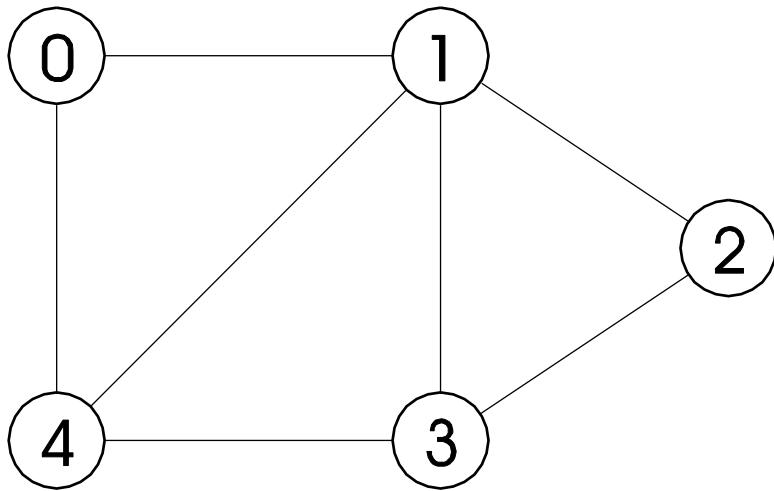
Abstract Representation of a Graph

- We will consider the vertices to be the integers from 0 to $|V|-1$.
- Operations on a graph:
 - Create a graph of n vertices.
 - Insert an edge
 - Remove an edge
 - Query the existence of an edge
 - Iterate over the vertices adjacent to a given vertex.

Concrete Representation of a Graph

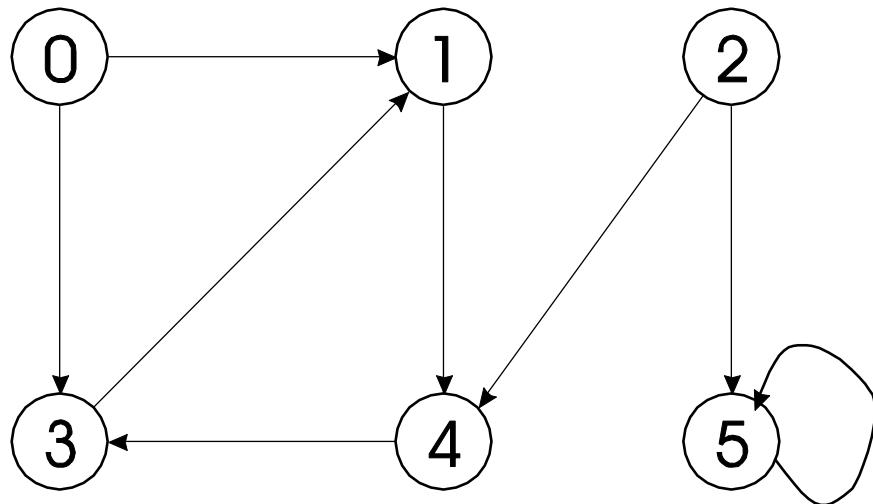
- Two methods of representing a graph are popular:
 - Adjacency Matrix
 - A $|V| \times |V|$ matrix with 1 (or the weight) (or true) for an edge and 0 (or infinity) (or false) for not-an-edge
 - Adjacency List
 - A vector (array) of lists, one for each vertex. Each list has the adjacent vertices.

Adjacency Matrix



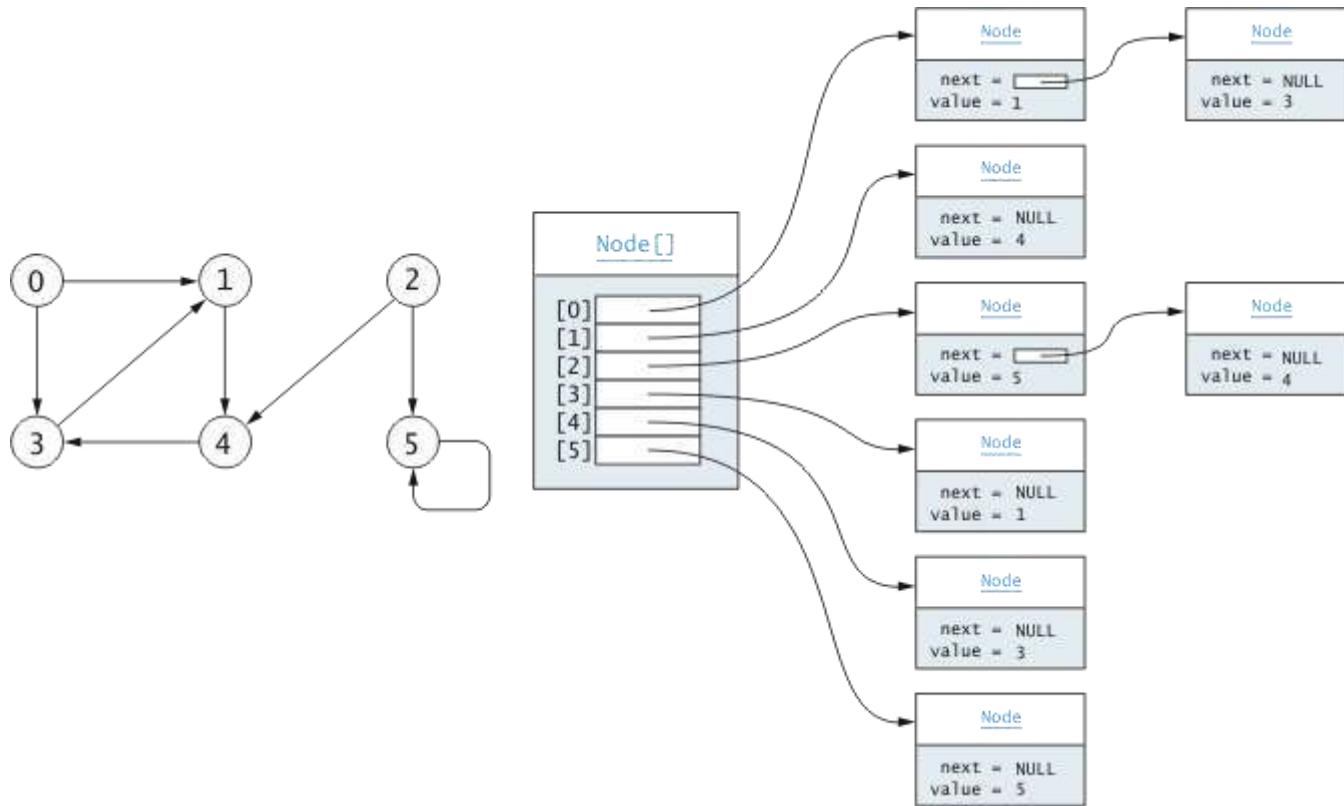
	0	1	2	3	4
0	1.0				1.0
1	1.0		1.0	1.0	1.0
2		1.0		1.0	
3		1.0	1.0		1.0
4	1.0	1.0		1.0	

Adjacency Matrix (directed graph)

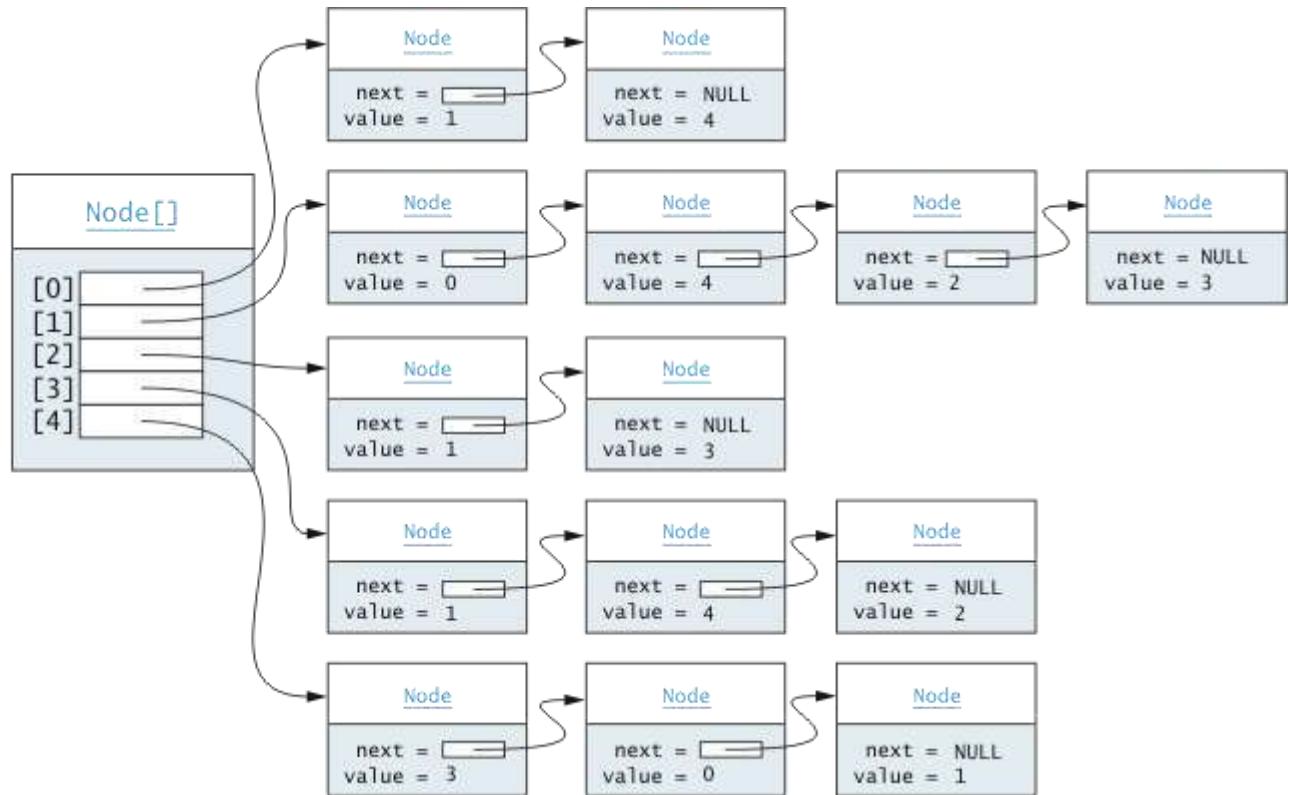
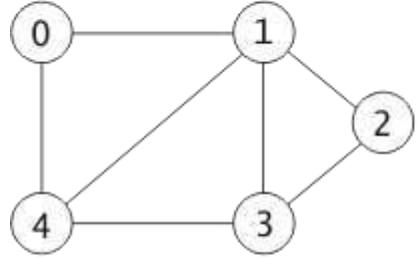


	0	1	2	3	4	5
0		1.0		1.0		
1					1.0	
2					1.0	1.0
3		1.0				
4				1.0		
5						1.0

Adjacency List (Directed Graph)



Adjacency List Representation



Observation About Graphs

(only connected graphs)

- Max number of $|E| = |V| * (|V|-1)$
- Min number of $|E| = |V|-1$

Observation About Adjacency List Representation

- The operation
`is_edge(int u, int v)`
is **$O(|E_v|)$**
- An iteration over all vertices adjacent to a given vertex V is **$O(|E_v|)$**
- Any algorithm that iterates over all of the edges by iterating over the vertices is **$O(|E|)$**

Observation About Adjacency Matrix Representation

- The operation
`is_edge(int u, int v)`
is **O(1)**
- An iteration over all vertices adjacent to a given vertex is **O(|V|)**
- An algorithm that iterates over all of the edges is **O(|V|^2)**

Comparison of Adjacency List and Adjacency Matrix Representation

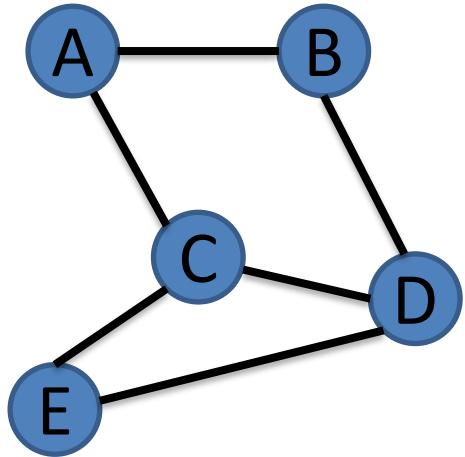
	Existents of an edge between two vertices	Iteration over all vertices adjacent to a given vertex	Iteration over all edges in a graph
Adjacency List	$O(E_v)$	$O(E_v)$	$O(E)$
Adjacency Matrix	$O(1)$	$O(V)$	$O(V ^2)$

Which to Use?

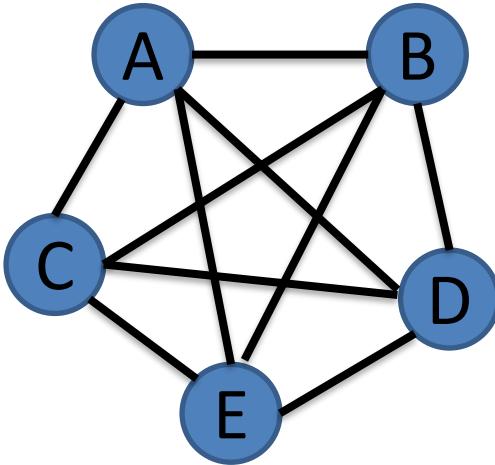
- Generally $|E| \ll |V|^2$
 - i.e. the adjacency matrix is sparse.
- Thus, for most applications, the adjacency list is preferred.
- But if $O(|E|) = O(|V|^2)$
- There are some algorithms, where the adjacency matrix is preferred.

Density, undirected

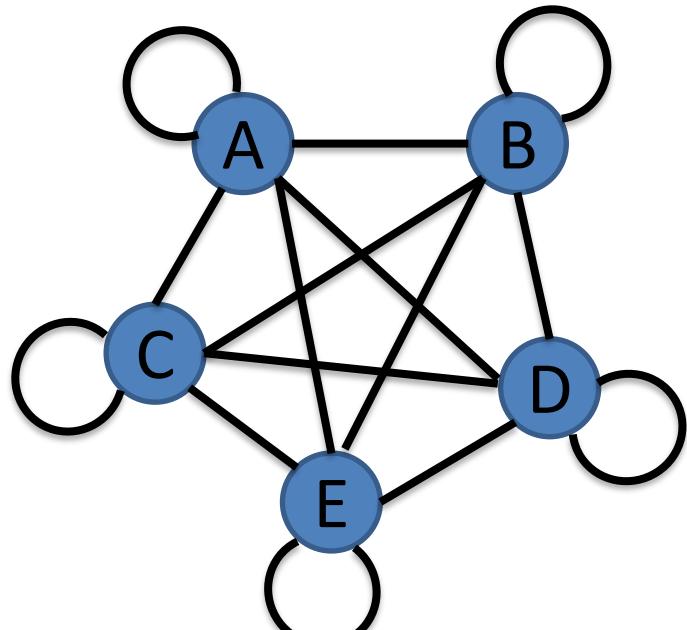
$|E|$ = number of edges
 $|V|$ = number of vertices



$|E| = 6$
 $|V| = 5$



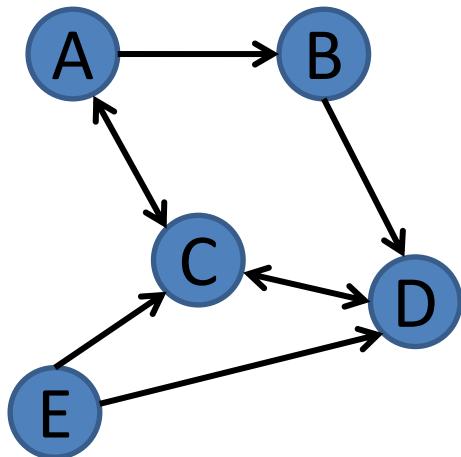
$|E| = 10$
 $|V| = 5$



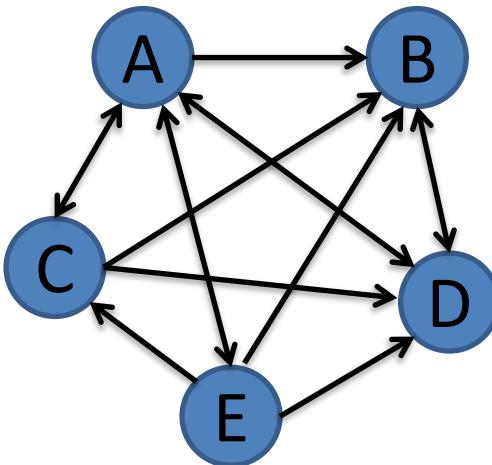
$|E| = 15$
 $|V| = 5$

Density, directed

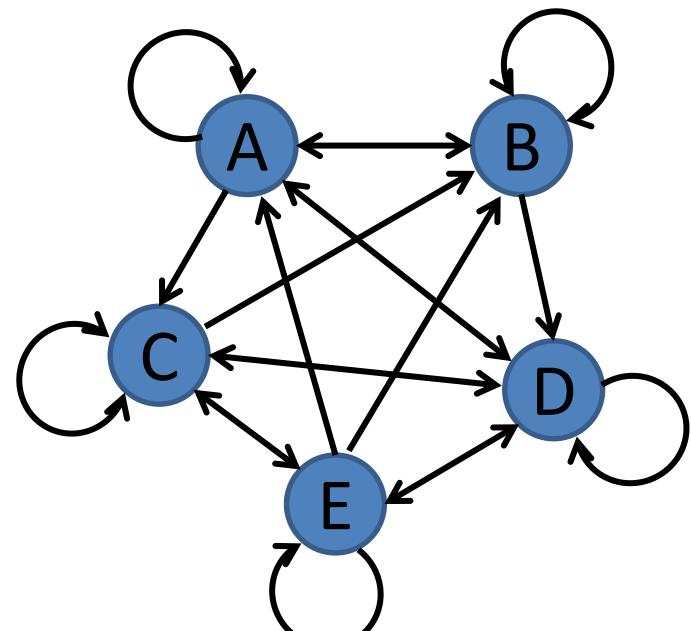
- $|E|$ = number of edges
- $|V|$ = number of vertices



$$\begin{aligned}|E| &= 8 \\ |V| &= 5\end{aligned}$$



$$\begin{aligned}|E| &= 14 \\ |V| &= 5\end{aligned}$$



$$\begin{aligned}|E| &= 24 \\ |V| &= 5\end{aligned}$$

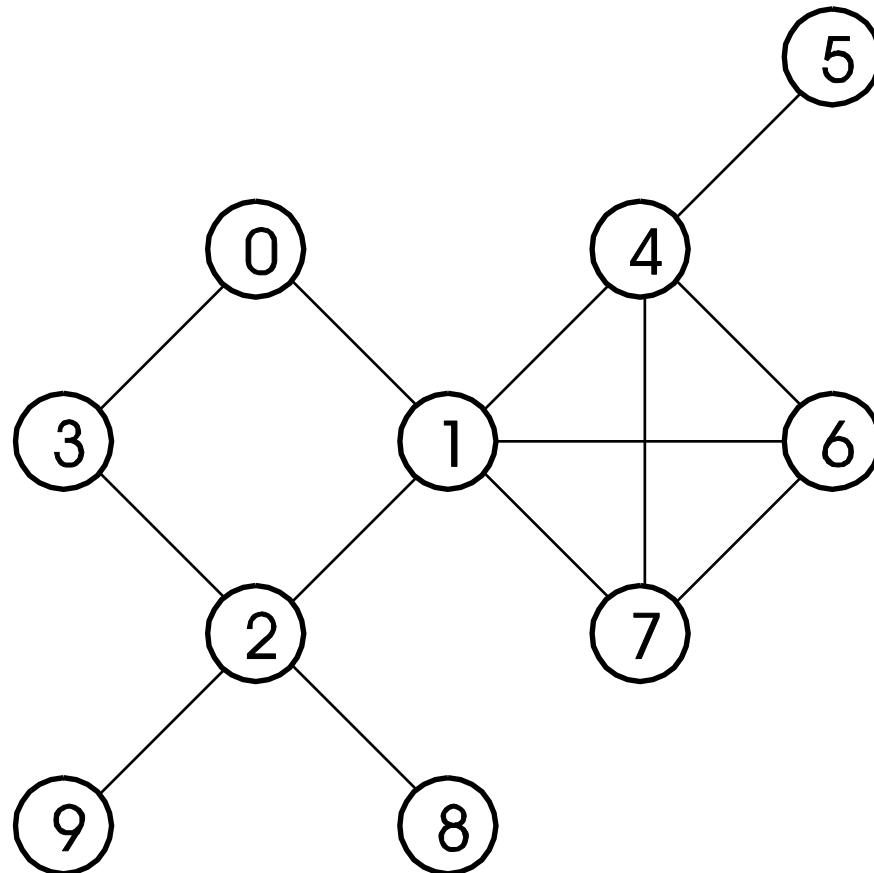
Breadth First Search

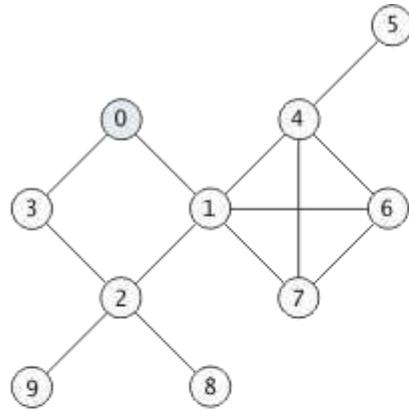
- Starting at a source vertex
- Systematically explore the edges to “discover” every vertex reachable from s .
- Produces a “breadth-first tree”
 - Root of s
 - Contains all vertices reachable from s
 - Path from s to v is the shortest path

Algorithm

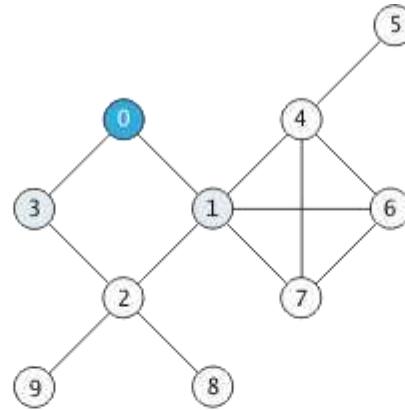
1. Take a start vertex, mark it identified (color it gray), and place it into a queue.
2. While the queue is not empty
 1. Take a vertex, u , out of the queue (Begin visiting u)
 2. For all vertices v , adjacent to u ,
 1. If v has not been identified or visited
 1. Mark it identified (color it gray)
 2. Place it into the queue
 3. Add edge u, v to the Breadth First Search Tree
 3. We are now done visiting u (color it black)

Example

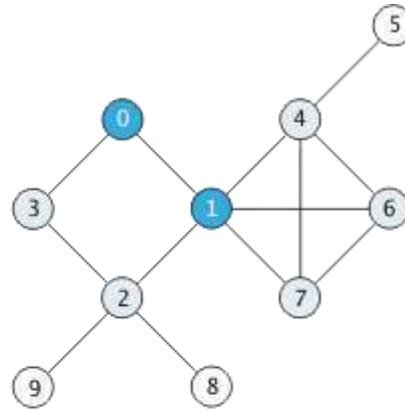




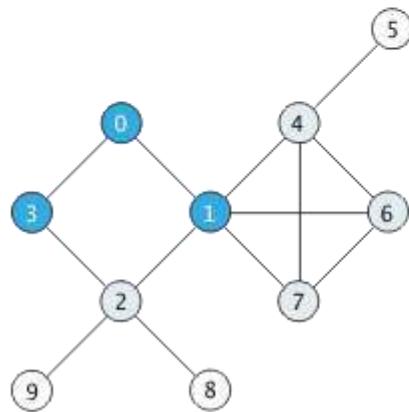
(a)



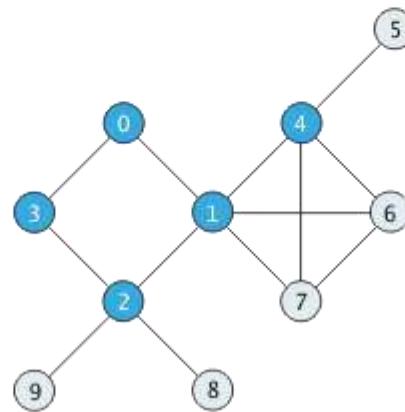
(b)



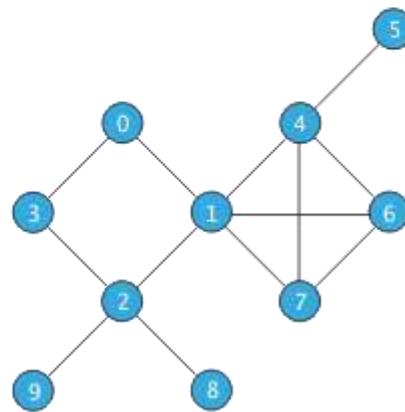
(c)



(d)



(e)

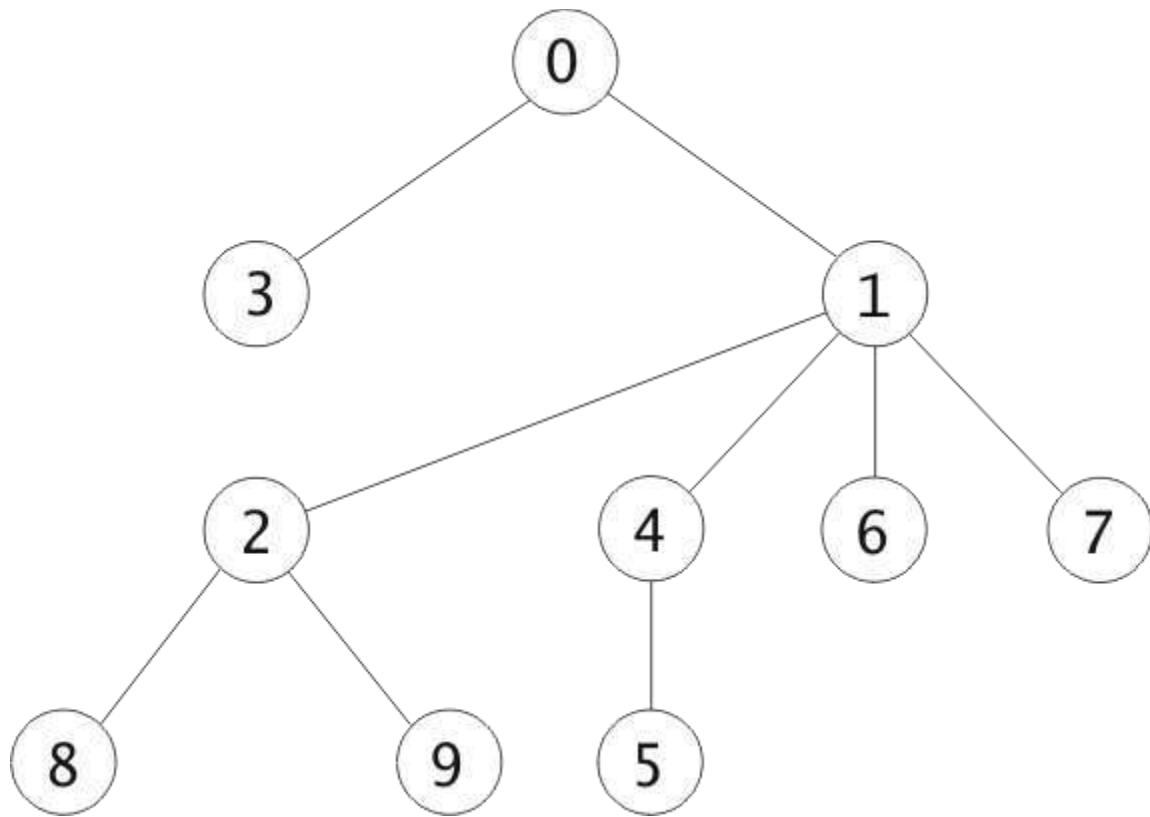


(f)

Trace of Breadth First Search

Vertex Being Visited	Queue Contents after Visit	Visit Sequence
0	1 3	0
1	3 2 4 6 7	0 1
3	2 4 6 7	0 1 3
2	4 6 7 8 9	0 1 3 2
4	6 7 8 9 5	0 1 3 2 4
6	7 8 9 5	0 1 3 2 4 6
7	8 9 5	0 1 3 2 4 6 7
8	9 5	0 1 3 2 4 6 7 8
9	5	0 1 3 2 4 6 7 8 9
5	empty	0 1 3 2 4 6 7 8 9 5

Breadth-First Search Tree



<u>vector<int> parent</u>
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
[9]

The table shows the parent pointers for each node in the BFS tree. The vector is indexed from 0 to 9. The values represent the parent node index for each child node. For example, node 0's parent is -1, and nodes 1, 2, 4, 6, and 7 have parent 1.

Application of BFS

- A Breadth First Search finds the shortest path from the start vertex to all other vertices based on number of edges.
- We can use a Breadth First Search to find the shortest path through a maze.
- This may be a more efficient solution than that found by a backtracking algorithm.

Depth First Search

- Start at some vertex
- Follow a simple path discovering new vertices until you cannot find a new vertex.
- Back-up until you can start finding new vertices.

Depth First Search using recursion.

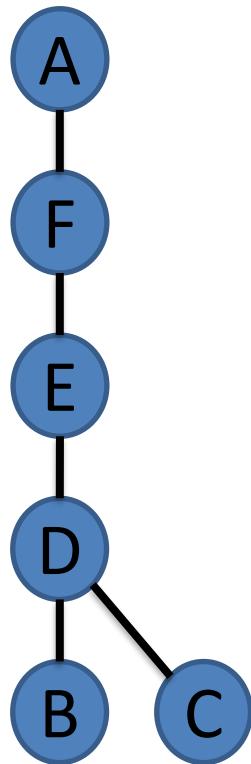
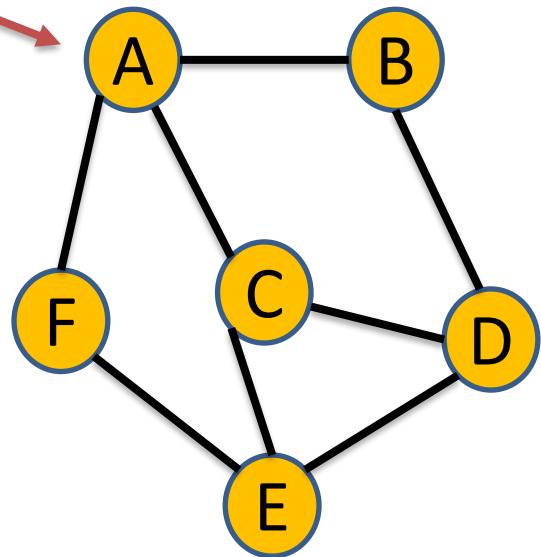
1. Start at vertex u . Mark it visited (color gray) and set its discovery time.
2. For each vertex v adjacent to u
 1. If v has not been visited
Insert u, v into DFS tree
 2. Recursively apply this algorithm starting at v
3. Mark u finished (color it black) and set its finish time.

Depth first using loops and a stack

Use a stack S and choose a start vertex u.

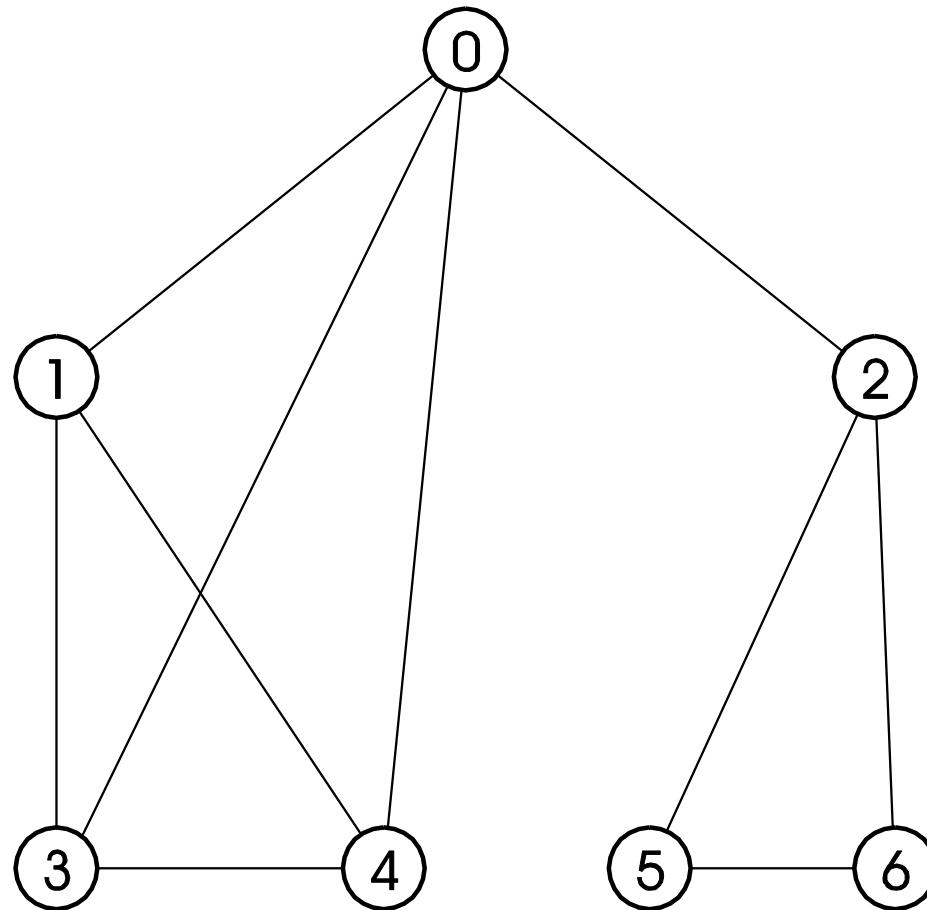
1. S.push(u). Add to tree as root (set u.parent to reference the depth first search tree)
2. while(S is not empty)
 1. v = S.pop
 2. if v is not visited
 - 1.visit v (color black).
 1. for each vertice u connected to v, in low-to-high order
 1. if u has not been visited
 2. set u.parent to v
 3. S.push(u)
 2. add v as child to v.parent in the DFS tree

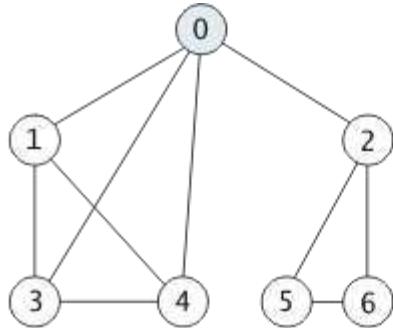
Depth first



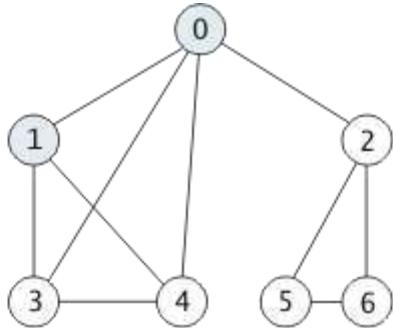
Visited	Stack after iteration	Visit sequence
A	BCF	A
F	BCE	AF
E	BCCD	AFE
D	BCCBC	AFED
C	BCCB	AFEDC
B	BCC	AFEDCB

DFS Example

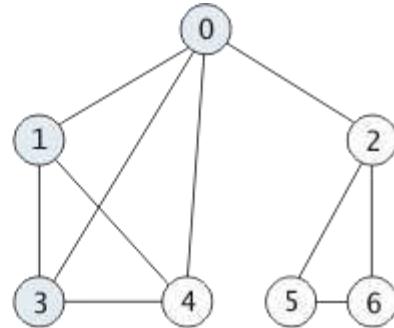




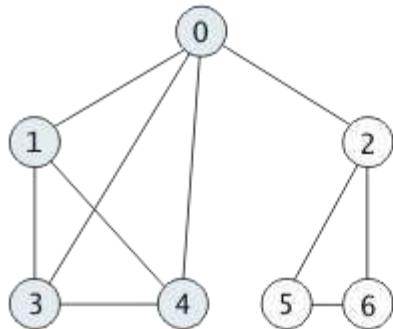
(a)



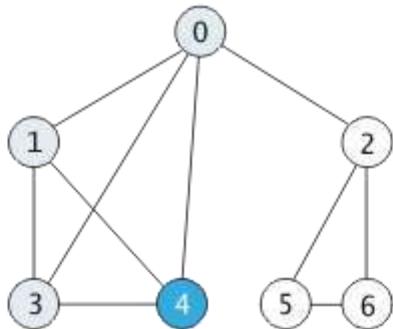
(b)



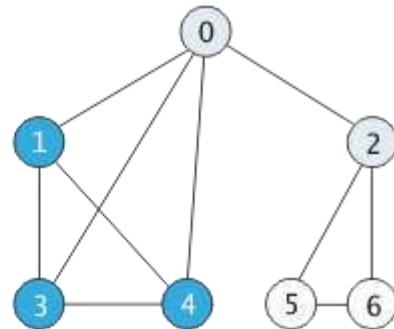
(c)



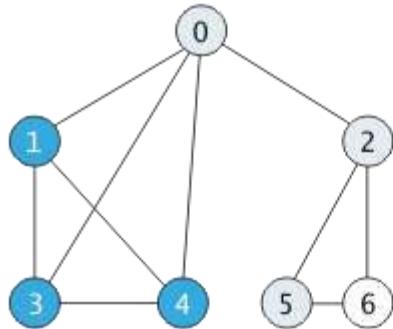
(d)



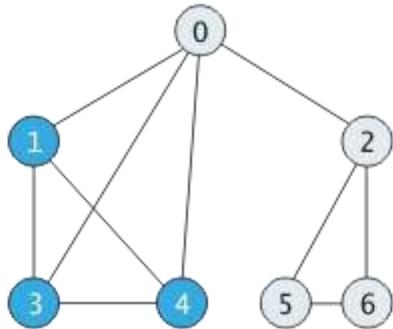
(e)



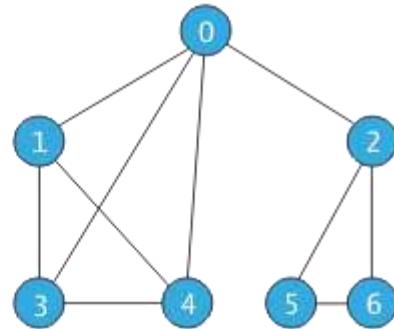
(f)



(g)



(h)

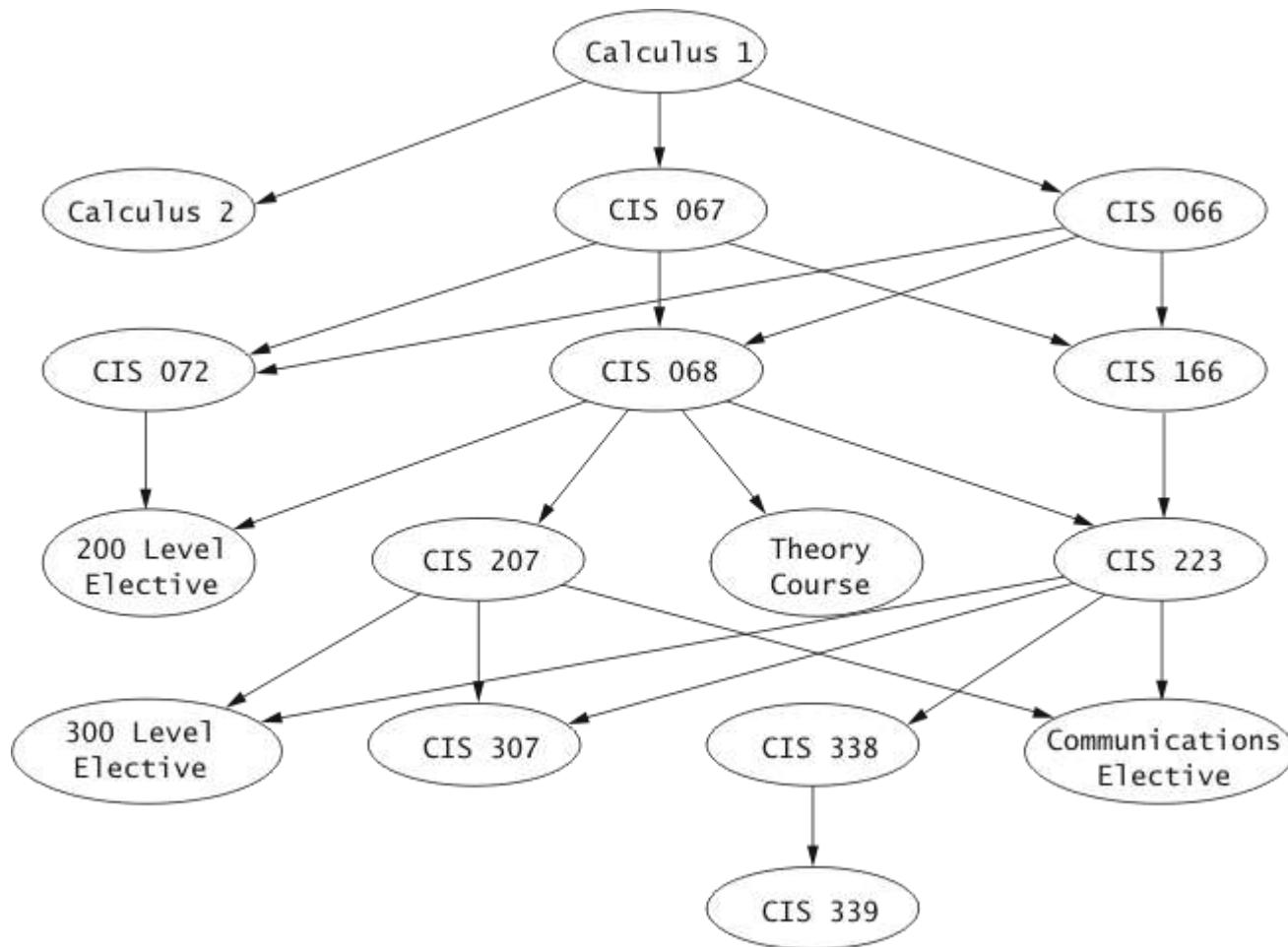


(i)

Application of DFS

- A topological sort is an ordering of the vertices such that if (u, v) is an edge in the graph, then v does not appear before u in the ordering.
- A Depth First Search can be used to determine a topological sort of a Directed Acyclic Graph (DAG)
- An application of this would be to determine an ordering of courses that is consistent with the prerequisite requirements.

CIS Prerequisites



Topological Sort Algorithm

- Observation: If there is an edge from u to v , then in a DFS u will have a finish time later than v .
- Perform a DFS and record the finish times.
- Order the vertices by decreasing finish time.

Application of Graphs

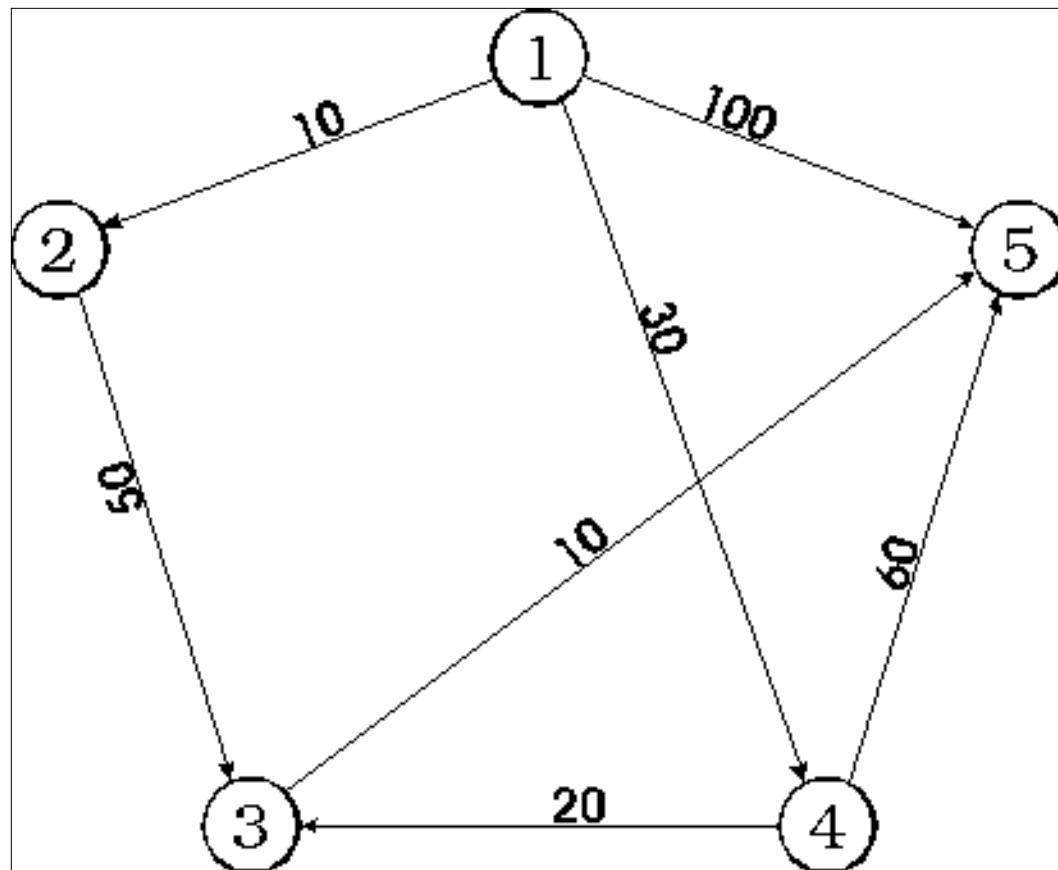
- Determine network connectivity.
- Schedule courses in accordance with prerequisite requirements.
- Finding the shortest (or least cost) path.
 - Network routing
 - Travel planning

Graphs part two

Weighted Graph

- A Weighted Graph is a Graph to which a value is associated with each of the edges.
- Weighted Graphs may be either directed or undirected.

Example of a Directed Weighted Graph

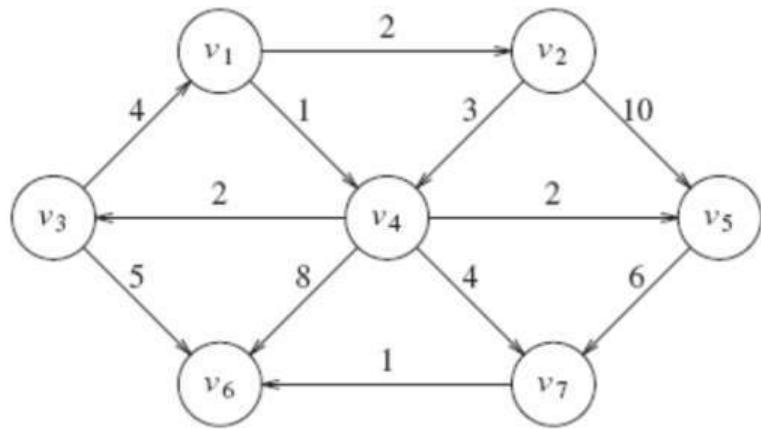


Finding the Shortest Path

- Given a graph we want to find the shortest path from a starting vertex to all other vertices.
- The famous computer scientist Edsger W. Dijkstra developed an algorithm to solve this problem.
- This algorithm is an example of a greedy algorithm.
 - A greedy algorithm is one that attempts to find the locally optimal solution at each step.

Dijkstra's Algorithm

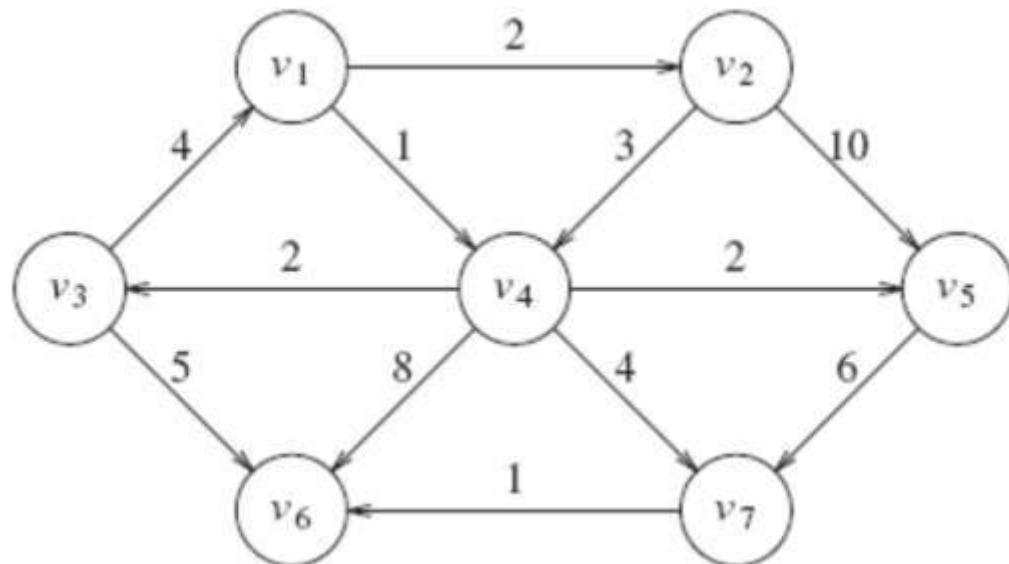
1. Initialize S with the start vertex, s , and $V - S$ with the remaining vertices.
2. for all v in $V - S$
 - 3. Set $p[v]$ to s
 - 4. if there is an edge (s, v)
 - 5. Set $d[v]$ to $w(s, v)$
 - 6. else
 - 6. Set $d[v]$ to ∞
 - 7. while $V - S$ is not empty
 - 8. for all u in $V - S$, find the smallest $d[u]$
 - 9. Remove u from $V - S$ and add it to S
 - 10. for all v adjacent to u in $V - S$
 - 11. if v is not known
 - 12. if $d[u] + w(u, v)$ is $< d[v]$
 - 13. Set $d[v]$ to $d[u] + w(u, v)$
 - 14. Set $p[v]$ to u .



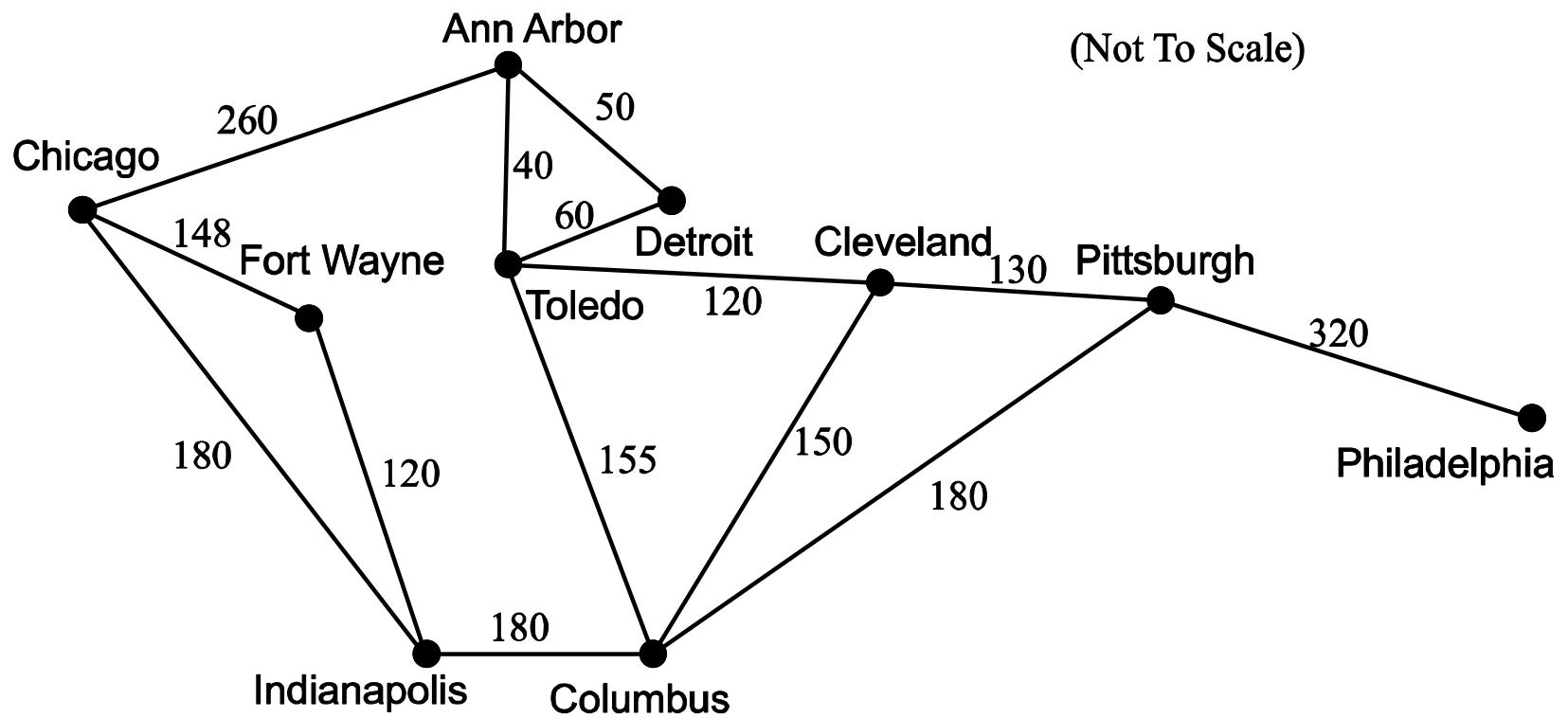
Vertex	Known	d_v	p_v
1			
2			
3			
4			
5			
6			
7			

Trace of Dijkstra

Vertex	Known	d_v	P_v
1	T	Na	Na
2	T	2	1
3	T	00, 3	1, 4
4	T	1	1
5	T	00, 3	1, 4
6	T	00, 9, 8, 6	1, 4, 3, 7
7	T	00, 5	1, 4



Another Weighted Graph



Trace of Dijkstra

s	d[v]	p[v]
Phil	Na	Na
Pitt	320	Phil,
Cle	∞ , 450	Phil, Pitt
Col	∞ , 500	Phil, Pitt
Indi	∞ , 680	Phil, Col
Tol	∞ , 570	Phil, Cle
Det	∞ , 630	Phil, Tol
Ann	∞ , 610	Phil, Tol
FW	∞ , 800	Phil, Indi
Chi	∞ , 860	Phil, Indi

Analysis of Dijkstra's Algorithm

- Step 2: For loop is $|V|$
- Step 7: While loop is $|V|$
 - Step 8: For loop is $|V|$
 - Step 10: The loop takes from 1 to $|V|$ time depending on the density of the graph.
- Regardless of the loop at step 10 the loop at step 8 always runs $|V|$
- The implementation is therefore $O(|V|^2)$

Dijkstra's Algorithm

1. Initialize S with the start vertex, s , and $V - S$ with the remaining vertices.
2. for all v in $V - S$
 - 3. Set $p[v]$ to s
 - 4. if there is an edge (s, v)
 - 5. Set $d[v]$ to $w(s, v)$
 - 6. else
 - 6. Set $d[v]$ to ∞
 - 7. while $V - S$ is not empty
 - 8. for all u in $V - S$, find the smallest $d[u]$
 - 9. Remove u from $V - S$ and add it to S
 - 10. for all v adjacent to u in $V - S$
 - 11. if v is not known
 - 12. if $d[u] + w(u, v)$ is $< d[v]$
 - 13. Set $d[v]$ to $d[u] + w(u, v)$
 - 14. Set $p[v]$ to u .

Dijkstra's Algorithm improved

PQ: a priority queue (Heap e.g.)

V: a graph

1. for all vertices u in V

 set $d[u] = \infty$

 start vertice u

$d[u] = 0$

2. $\text{PQ.add}(u, d[u])$

3. while(PQ !empty)

4. $u = \text{PQ.getMin}$

5. for all v adjecent to u

6. if v is not known

 if $d[u] + w(u, v) < d[v]$

 Set $d[v]$ to $d[u] + w(u, v)$

 Set $p[v]$ to u .

$\text{PQ.add}(v, d[v])$

7.

Analysis of Dijkstra's improved algorithm

- Step 1: For loop is $|V|$
- Step 2: To add a vertex to a PQ (Heap) takes $\log(|V|)$ time
- Step 3: All vertices will be added once to the PQ and every time the while loop runs one vertex is taken out so the while loop runs $|V|$ times
 - Step 4: To take a vertex from the PQ takes $\log(|V|)$ time
 - Step 5: The loop takes from 1 to $|V|$ time depending on the density of the graph.
- If Step 5 is 1 then we have $O(|V| \log |V|)$.
- If Step 5 is $|V|$ we have $O(|V|^2 \log |V|)$ or $O(|E| \log |V|)$,
- So it is only an improvement with sparse graphs.

Minimum spanning tree

- Let $G = (V, E)$ be an un-directed weighted graph.
- Wish to find an acyclic subset $T \subseteq E$ such that $w(T) = \sum_{(u,v) \in T} w(u,v)$ is the minimum.
- This is the minimum spanning tree.

Prim's Algorithm

1. Initialize S with the start vertex, s , and $V - S$ with the remaining vertices
2. for all v in $V - S$
 3. Set $p[v]$ to s
 4. if there is an edge (s, v)
 5. Set $d[v]$ to $w(s, v)$
 - else
 6. Set $d[v]$ to ∞
7. while $V - S$ is not empty
 8. for all u in $V - S$, find the smallest $d[u]$
 9. Remove u from $V - S$ and add it to S
 10. Insert the edge $(u, p[u])$ into the spanning tree.
 11. for all v adjacent to u in $V - S$
 12. if v is not known
 13. if $w(u, v) < d[v]$
 14. Set $d[v]$ to $w(u, v)$
 15. Set $p[v]$ to u .

Prim's Algorithm improved

PQ: a priority queue (Heap e.g.)

V: a graph

for all vertices v in V

 set $d[v] = \infty$

start vertex v

$d[v] = 0;$

PQ.add($v, d[v]$)

while(PQ ! empty)

$u = PQ.getMin$

Insert the edge $(u, p[u])$ into the spanning tree

 for all v adjacent to u

 if v is not known

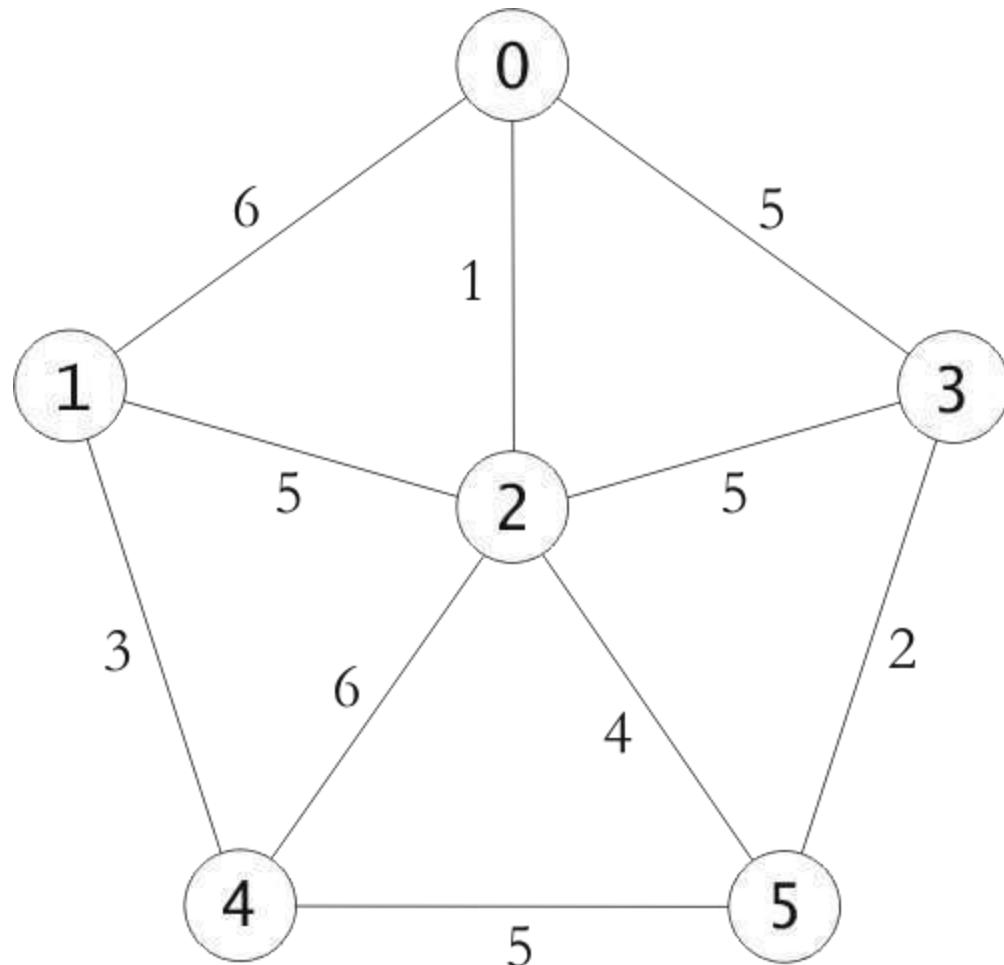
 if $w(u, v) < d[v]$

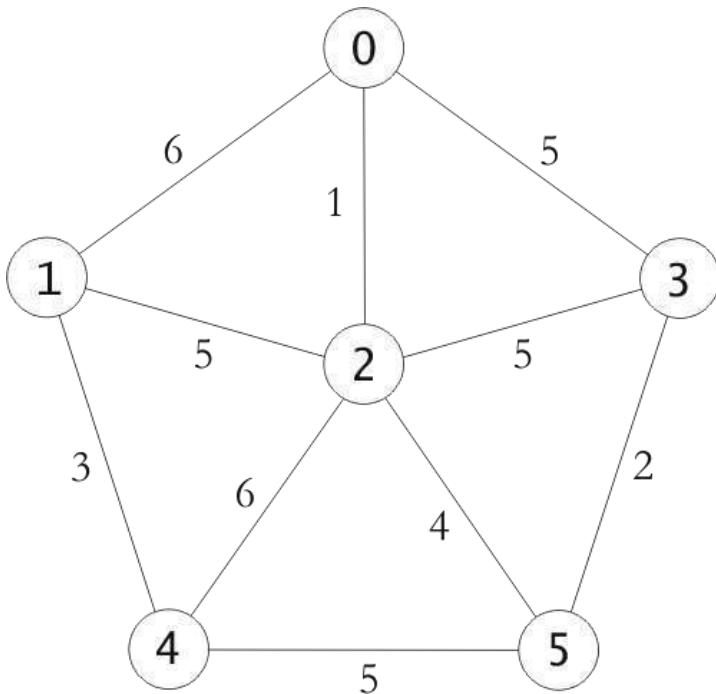
 Set $d[v]$ to $w(u, v)$

 Set $p[v]$ to u .

 PQ.add($v, d[v]$)

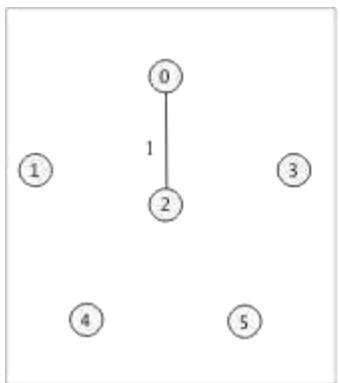
Example



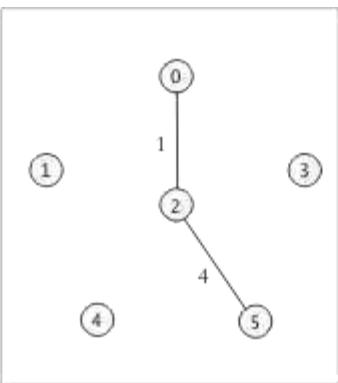


S	Known	D[v]	P[u]
0			
1			
2			
3			
4			
5			

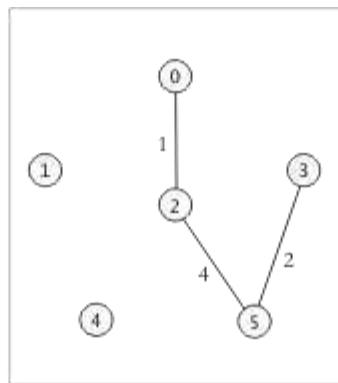
Building the MST



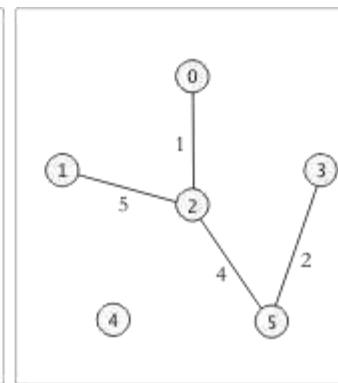
(a)



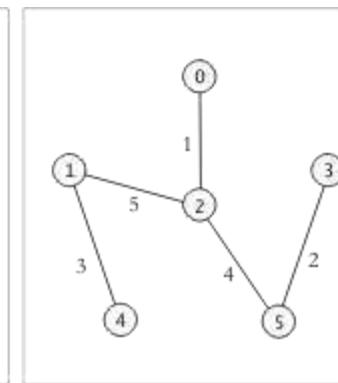
(b)



(c)



(d)



(e)

How to trace Prim's algorithm

S	D[v]	P[u]
0	na	na
1	6 - 5	0 - 2
2	1	0
3	5 - 2	0 - 5
4	∞ - 6 - 5 - 3	0 - 2 - 5 - 1
5	∞ - 4	0 - 2

S01 - Intro & Big-O

01 June 2022 17:59

Course:

Purpose:

Design, Implement, Analyze different algorithms
Get familiar with different advanced data structures

Skills:

Be able to analyze algorithms using big O-Notation
Be able to design and implement algorithms and data structures in object oriented programming language

Knowledge:

Know different linear data structures (Sets, Maps, Queues and Stacks)
Know different non linear data types (Trees, Heaps and Graphs)
Know different searching algorithms
Know different algorithm types and templates
Know the concept of abstract data types

Why study algorithms and data types:

Speed and memory usage:
We want fast/efficient programs that doesn't use much memory

What does efficiency means:

Gordon E. Moore's law about components (The laws says that the number of transistors doubles every two years, although the cost of computer is halved)
But the code matter just as much

We need tools to analyze and document our programs regarding speed and memory usage

Goal - to ensure software quality:

Program efficiency and correctness

- Unit-testing
- Big-O notation

Software design:

- Robustness,
- Usability
- Reliability
- Reusability
- Maintainability
- Portability

The challenge - how to manage complexity:

Using top-down approach and object-oriented design

Managing complexity:

- Data abstraction
- Procedural abstraction
- Information hiding

Class diagram documents interactions between classes

Abstract data types - using abstractions to manage complexity

An abstraction is a model of physical entity or activity

Models include relevant facts and details

Models exclude matters irrelevant to system/task

Abstraction helps programmers

Complex issues handled in manageable pieces

Procedural abstraction distinguishes

What to achieve (by a procedure)

From how to achieve it (implementation)

Data abstraction distinguishes

Data object for a problem and their operations

From their representation in memory

If another class uses an object only through its methods, the other class is not affected if the data representation changes

Information hiding - concealing the details of a class implementation from users of the class

Enforces the discipline of data abstraction

Major goal of software engineering - write reusable code

Abstract data type interface (ADT) data + methods

Contain only method signatures - name, parameters and return types

No indication of how achieved the result (procedural abstraction)

No representation (data abstraction)

Class implementing ADT must provide the method body with implementation

ADT is a contract between interface designer and coder that implements the interface

Precondition:

Any assumptions/constraint on the method input before execution

Postcondition:

Describes the result of executing the method

JAVA API

Java has an extensive library of collections - The Collections Framework

Why to learn ADT

General understanding

Special cases when collection is not found in Java

Choice of ADT - what is best for the given task

Software Patterns

Usually considered to be on higher level as ADT and algorithms

ADT and algorithms are building blocks for many software patterns

How to represent algorithms

Natural languages

Too verbose

Understanding depends on the reader experience

Programming language

Too low level

Requires following complicated syntax of programming languages

Can be written only for one programming language

Pseudo - code

Natural language modelled to look like statements

Can be implemented in different programming languages

Pseudo-code

- "An unordered sequence of" " - You can number the steps
- "unambiguous and well defined instructions" - each instruction is clean, do-able and can be done without difficulty
- "that performs some task" " - performs a task
- "and halts in finite time" " - the algorithm terminates

Assertions

Logical statements about program state

Claimed to be true at a particular point of program

Written as comment or as "assert" statement

Examples:

Preconditions

Postconditions

Loop invariants - The true condition before each loop cycle

Loop invariant

Helps prove that a loop meets its specification

Is true before loop begins

Is true at the beginning of each iteration

Is true just after loop exit

Example

Sorted(j): Array elements j, for $0 \leq j \leq i$, are sorted

Beginning: Sorted(0) is true

Middle: We insure initial portion sorted as we increase i

End: Sorted(n): All elements $0 \leq j < n$ are sorted

Efficiency of algorithms

How to characterize performance of an algorithm without regard to a specific:

Computer

Language

Over wide range of inputs?

Desire - function that describes execution time in terms of input size

Big-O notation

Idea

Ignore constant factor - implementation details - go for fundamental rate of increase

Consider fastest growing term - for large problems it will eventually dominate/

Value - Compares fundamental performance difference of algorithms

Caveat - For smaller problems, big-O worse performer may do better

Formulas

$T(n) = \text{time for algorithm on input size } n$

$f(n) = \text{a simpler function that grows at about the same rate}$

Example: $T(n) = 3n^2 + 5n + 17 = O(n^2)$

$f(n) = n^2$ has faster growing term

No extra leading constant in $f(n)$

Big-O is the worst case scenario

Variations for describing algorithms efficiency

Omega - growth rate of $T(N)$ is greater than or equal to $\Omega(g(N))$

Theta - growth rate of $T(N)$ is equal to $\Theta(h(N))$

Little o - growth rate of $T(N)$ is sharp less than $o(p(N))$

Big O - by far most used, we care about the worst case

Common growth rates

$O(1)$

$O(\log n)$

$O(n)$

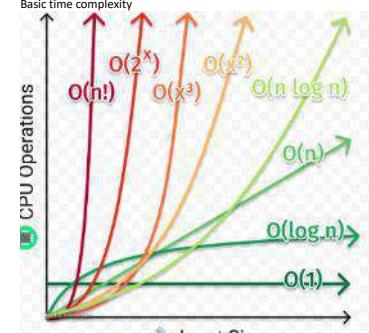
$O(n \log n)$

$O(n^2)$

$O(n^3)$

$O(2^n)$

$O(n!)$



S01 - Ex

05 June 2022 18:44

Exercise 1.1

Order the following by their $O(\cdot)$ ranking

- | | |
|-------------------------|----------------------|
| a) n^2 | |
| b) $\log n$ | $\log \log n$ |
| c) $n\sqrt{n}$ | $\log n$ |
| d) 2^n | n |
| e) $n \log n$ | $n \log n$ |
| f) $n!$ | $n\sqrt{n}$ |
| g) $\log \log n$ | $\frac{n^2}{\log n}$ |
| h) n | n^2 |
| i) n^n | 2^n |
| j) $\frac{n^2}{\log n}$ | $n!$ |
| | n^n |

Exercise 1.2

Which of the following are true?

- a) $4n^2 = O(n^2)$ ✓
b) $4n^2 + 18n \log n = O(n^2)$ ✓
c) $4n^2 + 18n \log n = O(n)$ ✗
d) $4n^2 + 18n \log n = O(n \log n)$ ✗
e) $4n + 18n \log n = O(n \log n)$ ✗
f) $4n + 18n \log n = O(n^2)$ ✓

Exercise 1.3

What is the complexity of the following algorithms?

a) m^n :

```
power(m, n):
    r = 1
    for i ∈ [1..n]:
        r = r * m
    return r
```

b)

m^n :

```
power(m, n):
    if n = 0 then
        return 1
    else if n is even then
        return power(m * m, n / 2)
    else
        return m * power(m * m, (n - 1) / 2)
```

c)

```
reverse( a[] )
    left = 0;
    right = a.length - 1;           O(n / 2)

    while( left < right )
        swap( a, left++, right-- );
```

```
swap( a[], left, right )
tmp = a[left]
a[left] = a[right]
a[r] = tmp
```

S02 - Lists, Queues, Stacks

01 June 2022 18:13

SO2 - Ex

05 June 2022 19:18

Order the following functions by growth rate: ~~\times~~ , ~~\sqrt{x}~~ , ~~$\log x$~~ , ~~N~~ , ~~N^2~~ , ~~$N \log N$~~ , ~~$N \log^2 N$~~ , ~~$N \log(N^2)$~~ , ~~N^3~~ , ~~$N^{1.5}$~~ , ~~$N^{1/2}$~~ , ~~$N^{\log N}$~~ , ~~$N \log^2 N$~~ , ~~$N \log(N^2)$~~ , ~~$N^2 \log N$~~ , ~~$N^{1.5}$~~ , ~~$N^{1/2}$~~ , ~~N^3~~ , ~~$2^{N/2}$~~ , ~~2^N~~ . Indicate which functions grow at the same rate.

37.
 \sqrt{N} ,
 $2/N$,
 $N \log \log N$,
 N ,
 $N \log N$,
 $N \log^2 N$,
 $N \log(N^2)$,
 $N^2 \log N$,
 $N^{1.5}$,
 $N^{1/2}$,
 N^3 ,
 $2^{N/2}$,
 2^N

2.7 For each of the following six program fragments:

- Give an analysis of the running time (Big-Oh will do).
- Implement the code in Java, and give the running time for several values of N .
- Compare your analysis with the actual running times.

```
(1) sum = 0;                                O(n)
    for( i = 0; i < n; i++ )
        sum++;

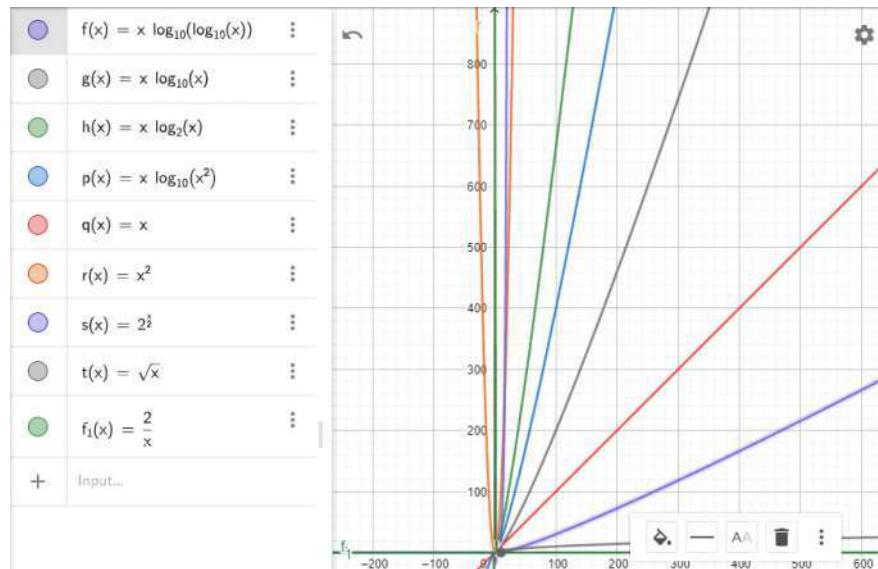
(2) sum = 0;
    for( i = 0; i < n; i++ )
        for( j = 0; j < n; j++ )            O(n^2)
            sum++;

(3) sum = 0;
    for( i = 0; i < n; i++ )
        for( j = 0; j < n * n; j++ )      n^3
            sum++;

(4) sum = 0;
    for( i = 0; i < n; i++ )
        for( j = 0; j < i; j++ )          O(n * log(n))
            sum++;

(5) sum = 0;
    for( i = 0; i < n; i++ )
        for( j = 0; j < i * i; j++ )      O(n^3) - log missing?
            for( k = 0; k < j; k++ )
                sum++;

(6) sum = 0;
    for( i = 1; i < n; i++ )
        for( j = 1; j < i * i; j++ )
            if( j % i == 0 )
                for( k = 0; k < j; k++ )
                    sum++;
```



- 2.6 In a recent court case, a judge cited a city for contempt and ordered a fine of \$2 for the first day. Each subsequent day, until the city followed the judge's order, the fine was squared (that is, the fine progressed as follows: \$2, \$4, \$16, \$256, \$65, 536, ...).
- What would be the fine on day N ?
 - How many days would it take the fine to reach D dollars? (A Big-Oh answer will do.)

S03 - Binary Trees

02 June 2022 13:15

Binary trees

- Non-linear structure where elements are organised into hierarchy
- Set of nodes where elements are stored and edges connect one node to another
- Each node is located on particular level
- Each tree has only one root

- Nodes at lower level of tree are children of nodes at previous level
- Node has only one parent but may have multiple children
- Sibling nodes- nodes with the same parents
- Root node - is the only node that has no parent
- Leaf node - Node with no children
- Internal node - Node that is not root and has at least one children
- Subtree - tree structure that makes up part of another tree
- Path - can be followed from parent to child, starting from root
- Ancestor node is a node that is above the node in its path from the root

Types

- Binary tree - where nodes has at most two children
- Balanced trees - is a tree where all leaves of the tree are on the same level or within one level of each other
- Full - all leaves are at the same height and every non-leaf children has exactly n children
- Complete - if its full to the pre-last level with all leaves at the bottom level on the left side of the tree

Computed child link

- Array can be used for full or complete binary trees for representation - otherwise it can waste large amount of array space
- The system
 - Element is stored in position n
 - Element's left child is stored in array in position (2n+1)
 - Element's right child is stored in array in position (2 * (n+1))

Operations

Adding

- Similar to finding
- New elements are added as leaf nodes
- Starting at root, follow path dictated by existing elements until there is no child in the desired direction and then add new element
- If the element is equal to the already existing element, we add it to the right child

Removing

- Node is a leaf - simply can be removed
- Node has one child - deleted node is replaced by the child
- Node has two children - appropriate node is found lower in the tree and used to replace the node
 - Successor looks for smallest node in right subtree

Traversals

- Preorder -
- Inorder -
- Postorder -
- Levelorder -

Preorder:

A B D E C

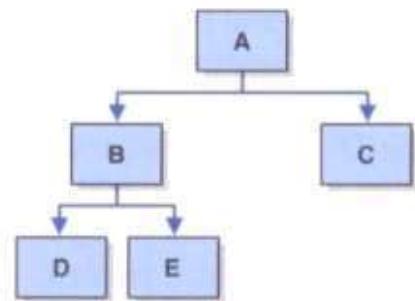
Inorder:

D B E A C

Postorder:

D E B C A

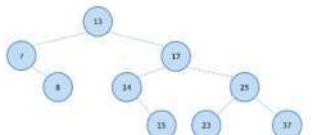
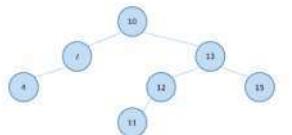
Level-Order: A B C D E



S03 - Ex

05 June 2022 22:37

1.
Draw the two trees as arrays, using computed child links.

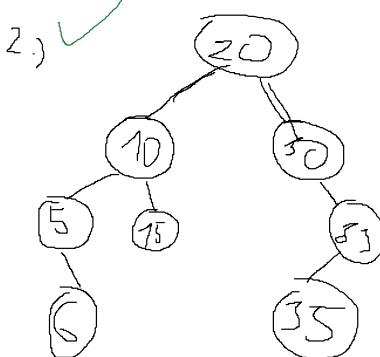
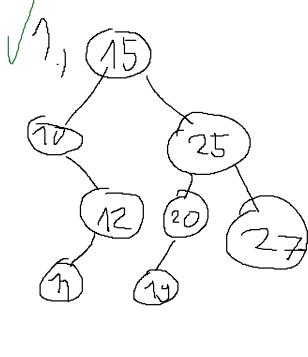


0	1	2	3	4	5	6	7	8	9	10	11	12	13
10	7	13	4		12	15				11			

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
13	7	17		8	14	25				15	23	37		

2.
The two arrays represent trees that are implemented with computed child links. Draw the trees.

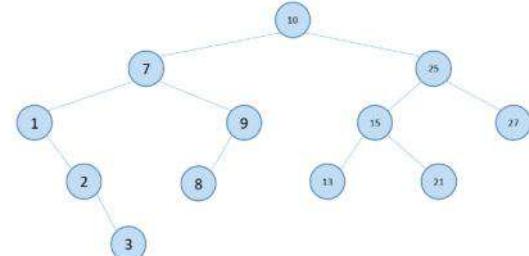
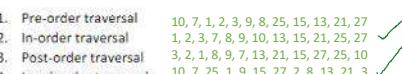
15	10	25	-	12	20	27	-	-	11	-	19	-	
20	10	30	5	15	-	33	-	6	-	-	-	35	-



3.

Write the sequences of numbers resulting from doing a:

1. Pre-order traversal 10, 7, 1, 2, 3, 9, 8, 25, 15, 13, 21, 27
2. In-order traversal 1, 2, 3, 7, 8, 9, 10, 13, 15, 21, 25, 27
3. Post-order traversal 3, 2, 1, 8, 9, 7, 13, 21, 15, 27, 25, 10
4. Level-order traversal 10, 7, 25, 1, 9, 15, 27, 2, 8, 13, 21, 3

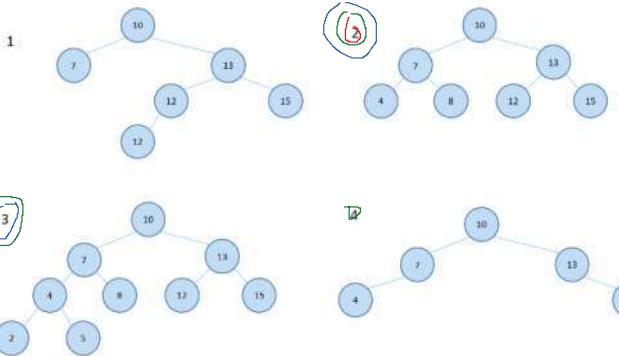


What is the time complexity of the traversals? $O(n)$

4.

For each of the trees determine if it is:

- Full —
- Complete —
- Balanced —



S04 - Balanced Binary Trees

02 June 2022 13:15

BST - Binary Search Trees

Can become highly unbalanced after operations

AVL tree

Named after creators

Ensures that the BST stays balanced

Each node has numeric "balance factor" - difference between heights of its subtrees

After each add or removal, balance factors are checked and rotations performed as needed

Height of empty tree is 0

Height of others is $- ht(n) = 1 + \max(ht(n.left), ht(n.right))$

Balance(n) = $ht(n.right) - ht(n.left)$

Critically unbalanced trees

Left - Left (Parent balance is -2, left child balance is -1)

 Rotate right around parent

Left - Right (Parent balance -2, left child balance +1)

 Rotate left around child

 Rotate right around parent

Right - Right (Parent balance +2, right child balance +1)

 Rotate left around parent

Right - Left (Parent balance +2, right child balance -1)

 Rotate right around child

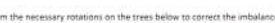
 Rotate left around parent

Comparing operations

Operation	LinkedList	BinarySearchTreeList
removeFirst	O(1)	O(log n)
removeLast	O(n)	O(log n)
remove	O(n)	O(log n)*
first	O(1)	O(log n)
last	O(n)	O(log n)
contains	O(n)	O(log n)
isEmpty	O(1)	O(1)
size	O(1)	O(1)
add	O(n)	O(log n)*

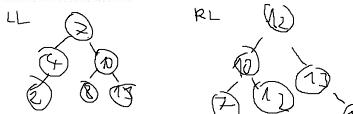
*both the add and remove operations may cause the tree to become unbalanced

1. Perform the necessary rotations on the trees below to correct the imbalance:

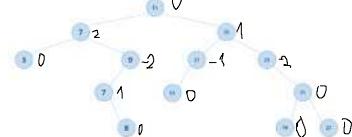


```
graph TD; subgraph LeftTree [Left Tree]; N1((1)) --- N2((2)); N2 --- N3((3)); N3 --- N4((4)); N4 --- N5((5)); N5 --- N6((6)); end; subgraph RightTree [Right Tree]; N7((7)) --- N8((8)); N8 --- N9((9)); N9 --- N10((10)); N10 --- N11((11)); N11 --- N12((12)); end;
```

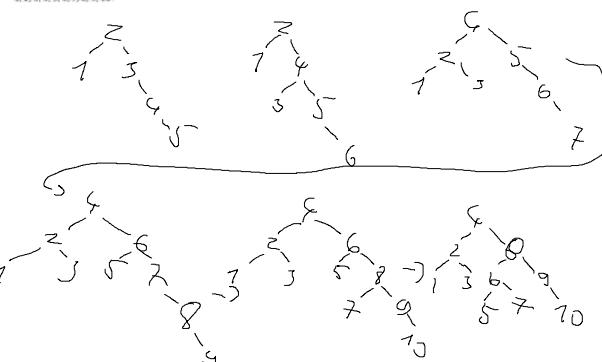
What type of rotation did you use?



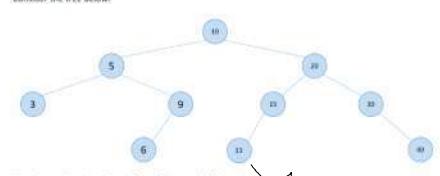
2. Write the balance factor next to each node in the tree below:



3. Draw the AVL-Tree, which results from adding the following sequence of numbers to an empty tree:
1, 2, 3, 4, 5, 6, 7, 8, 9, 10



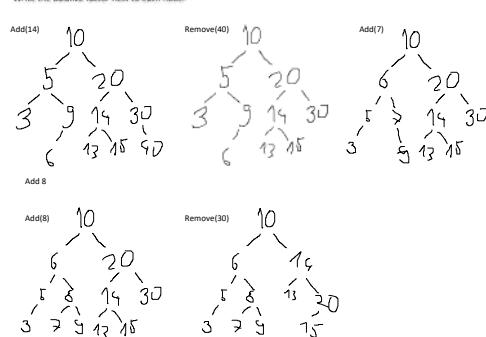
- Consider the tree below.



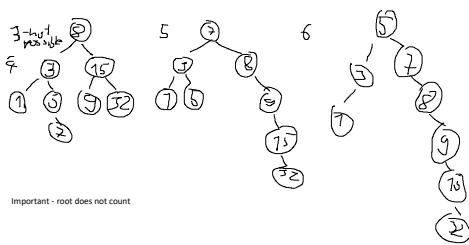
- Draw the resulting AVL-Tree after performing

 - add[14]
 - remove[40]
 - add[7]
 - add[8]
 - remove[30]

Write the balance factor next to each node.



For the following set of elements {1, 3, 5, 7, 8, 9, 15, 32} draw binary search trees of height 3, 4, 5, 6.

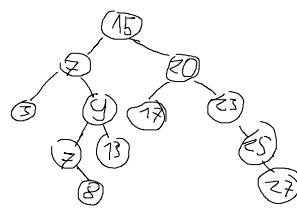


Important - root does not count

3.

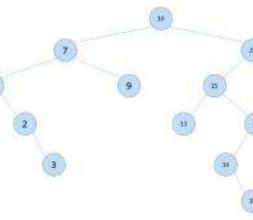
Draw the BST (non-balanced) after adding the following sequence of numbers to an empty tree:

15, 7, 9, 3, 7, 8, 20, 23, 17, 25, 13, 27

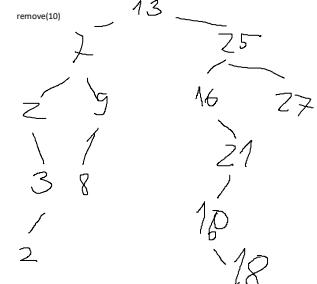


4.

Consider the following unbalanced binary search-tree:



remove(10)



Draw the resulting tree after the following operations are called on the tree:

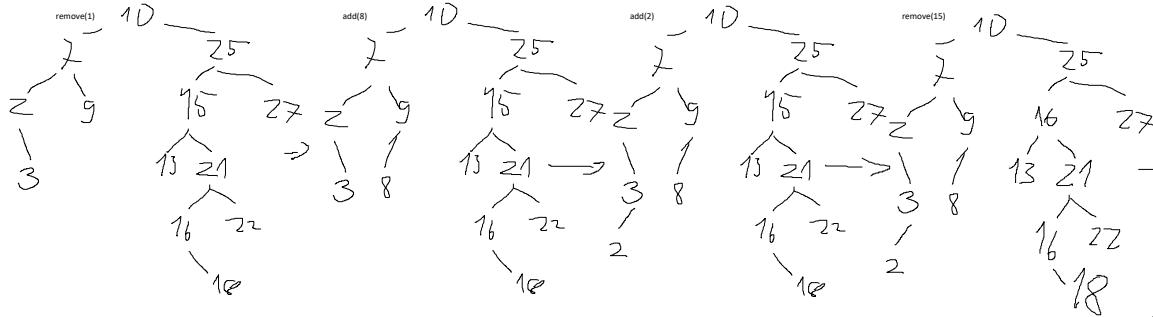
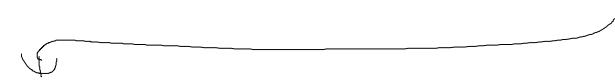
- remove(1)
- add(8)
- add(2)
- remove(15)
- remove(22)
- add(18)
- remove(10)

```

highToLow(Node node) {
    if(node == null) {
        return;
    }
    highToLow(node.right);
    System.out.println(node.value);
    highToLow(node.left);
}

//TODO
Stack<Node> stack = new Stack();
Node current = root;
while(true) {
    if(current.right == null) {
        } else {
            stack.push(current.right);
            sout(current)
        }
        current.push();
    try() {
        current = stack.pop();
    } catch(Exception e) {
        break;
    }
}

```



S05 - Sets, Maps and Hashing

02 June 2022 13:16

Set

Collection that contains no duplicate elements and at most one null element

Operations

Testing for membership

Adding elements

Removing elements

Union

- set whose elements belong to either A or B or to both A and B

Intersection

- set whose element belong to both A and B

Difference

- set whose elements belong to A but not to B

Subset

- set A is a subset of set B if every element of set A is also an element of set B

Map

Related to set - it is set of ordered pairs

Ordered pair - key, value

In a map there are no duplicate keys

Values may appear more than once

This is mapped to a particular value

Hash Map

Reducing collisions by growing - rehashing

Choose a new larger array size - for example doubling

Install the new array and drop the old

Probing

Linear probing

Tends to form long clusters of keys in the table

By incrementing the index

If "fall off end", wrap around to the beginning

Take care of not cycling forever!

Steps

Compute index as hash_fcn() % table.size()

If table[index] == NULL, item is not in the table

If table[index] matches the item, found item (done)

Increment index circularly and go to 2

Termination

Stop when you come back to your starting point

Stop after probing N slots, where N is table size

Stop when you reach bottom the second time

Ensure table never full

Considerations

Cannot traverse a hash table

When item is deleted, cannot just set its entry to null - would break probing

Quadrating probing

Index increments form a quadratic series

Direct calculations involves multiply, add remainder

Probe sequence may not produce all table slots

Sequence - s, s + 1^2, s + 2^2, s + 3^2, s + 4^2, s + 5^2, ...

Double hashing

When collision occurs, apply second function on the key and add this value

If this produce conflict 3 times, add third function

Separate chaining

Alternative to open addressing and probing

Each table slot references a linked list

List contains all items that hash to that slot

Linked list is often called bucket - Bucket hashing

Insertion about as complex

Deletion is simpler

Linked List can become long - rehash

Performance of hash tables

Hash table

Insert - average $O(1)$

Search - average $O(1)$

Sorted array

Insert - average $O(n)$

Search - Average $O(\log n)$

Binary Search Tree

Insert - average $O(\log n)$

Search - average $O(\log n)$

Balanced tree can guarantee $O(\log n)$

1. Collections implementing the *Set* interface must contain unique elements.

True

2. Sets contain no pair of elements *e1* and *e2* such that *e1.equals(e2)*, and at most one *null* element.

True

3. Although you cannot reference a specific element of a Set, you can iterate through all its elements using an Iterator object.

False - you cannot reference specific element of a set - only convert it to array, and you can iterate through it

4. Mathematically, a(n) _____ Hash map _____ is a set of ordered pairs whose elements are known as the key and the value.

5. The intersection of sets A, B is a set _____.

- A) whose elements belong to A but not to B
B) whose elements belong to both A and B
C) whose elements belong either to A or B or to both A and B
D) set where every element of A is also an element of B

6. With respect to the Map interface, the _____ method returns the current value associated with a given key.

- A) isEmpty
B) insert
C) put
D) get

7. With respect to the Map interface, the _____ method either inserts a new mapping or changes the value associated with an existing mapping.

- A) get
B) isEmpty
C) put
D) insert

// Map contains only unique keys

8. Which of the following is an example of a map?

- A) { (J, Jane), (B, Bill), (S, Sam), (B1, Bob), (B, Bill) } **•**
B) { (J, Jane), (B, Bill), (S, Sam), (B1, Bob), (B2, Bill) }
C) { (J, Jane), (B, Bill), (S, Sam), (B1, Bob), (J, Jane) } **•**
D) { (S, Sam), (B, Bill), (S, Sam), (B1, Bob), (B2, Bill) } **•**

9. The union of two sets A, B is a(n) _____.

- A) set whose elements belong to both A and B
B) set whose elements belong to A but not to B
C) set where every element of A is also an element of B
D) set whose elements belong either to A or B or to both A and B

Hash table exercise:

Given input {4371, 1323, 6173, 4199, 4344, 9679, 1989} and a hash function:

$h(x) = x \bmod 10$, show the tables for a hash table size 10:

- a. Separate chaining hash table.
b. Hash table using linear probing.
c. Hash table using quadratic probing.
d. Hash table with second hash function $h_2(x) = 7 - (x \bmod 7)$.
e. Show the result of rehashing the hash tables to new tables size 19

a.	0	1	2	3	4	5	6	7	8	9
	4371			1323	4344					4199
				6173						9679
										1989



b.	0	1	2	3	4	5	6	7	8	9
	9679	4371	1989	1323	6173	4344				4199



c.	0	1	2	3	4	5	6	7	8	9
	9679	4371	1323	6173	4344		1989			4199



d. - cannot insert value 1989 due to infinite cycle

d.	0	1	2	3	4	5	6	7	8	9
	4371		1323	6173	9679		4344			4199



S06 - Priority Queues (Heaps), Tree rebalancing

02 June 2022 13:17

Heaps

Complete Binary tree

Each element is less than or equal to both of its children

BST has left/right ordering, Heaps has up/down ordering

Has structural and ordering constraints

Is a complete binary tree

Adding

- Add the element as leaf, keeping the tree complete

Move element up towards root exchanging positions until the relationship among the elements is appropriate

Removing

- Remove root (min) and reconstruct the heap

Move the last leaf of the complete tree to be new root

Move it down the tree as needed until the relationship among elements is appropriate

The algorithm - Compare parent with its children and swap the smallest if the child is less than the parent

S06 - Ex - Sorting

07 June 2022 12:56

True

Which of the following statements are true?

False

1. The number of comparisons for a selection sort is represented by the series: $(n - 1) + (n - 2) + \dots + 2 + 1$
2. Insertion sort is considered a quadratic sort.
3. With respect to selection sort, the number of comparisons is $O(n)$.
4. The improvement of Shell sort over insertion sort is much more significant for small arrays.
5. The method `sort(int[] items)`, in class `java.util.Arrays`, sorts the array item in ascending order.

Fill out the blanks:

- ✓ 6. A class that implements the Comparable interface must define a(n) compareTo method that determines the natural ordering of its objects.
- ✓ 7. Selection sorts an array by making several passes through the array, selecting the next smallest item each time and placing it where it belongs in the array.
- ✓ 8. In the best case, selection sort makes $O(\underline{n^2})$ comparisons.

✗ 9. With respect to merge sort, additional space usage is $O(\underline{\hspace{2cm}})$.

- ✓ 10. The idea behind merge sort is to sort many smaller subarrays using insertion sort before sorting the entire array.
- ✓ 11. Shell sort has $O(n^{3/2})$ or better performance.
- ✗ 12. Insertion sort is an example of a(n) sorting algorithm.

- ✓ 13. You can think of the Shell sort as a divide-and-conquer approach to insertion sort.

- ✓ 14. The following represents the sort algorithm.

Set the initial value of *gap* to $n / 2$.
while *gap* > 0
 for each array element from position *gap* to the last element
 Insert this element where it belongs in its subarray.
 if *gap* is 2, set it to 1.
 else *gap* = *gap*/2.2.

- A) Shell
B) Heap
C) Insertion
D) Quick

- ✓ 15. The following is the algorithm.

Access the first item from both sequences.
while not finished with either sequence
 Compare the current items from the two sequences, copy the smaller current item to the output sequence, and
 access the next item from the input sequence whose item was copied
 Copy any remaining items from the first sequence to the output sequence.
 Copy any remaining items from the second sequence to the output sequence.

- A) Shell sort

Copy any remaining items from the first sequence to the output sequence.
Copy any remaining items from the second sequence to the output sequence.

- A) Shell sort
- B) Selection sort
- C) Merge sort
- D) Heapsort

16. In merge sort, the total effort to reconstruct the sorted array through merging is ____.

- A) $O(1)$
- B) $O(\log_2 n)$
- C) $O(n \log n)$
- D) $O(n^2)$

17. Which of the following sorts is not $O(n \lg(n))$?

- A) selection
- B) heap
- C) merge

18. The best sorting algorithms provide ____ average-case behavior and are considerably faster for large arrays.

- A) $O(1)$
- B) $O(n)$
- C) $O(n^2)$
- D) $O(n \log n)$

19. Which of the following generally gives the worst performance?

- A) Selection sort
- B) Bubble sort
- C) Insertion sort
- D) Shell sort

20. Shell sort is ____ if successive powers of 2 are used for gap.

- A) $O(\log n)$
- B) $O(1)$
- C) $O(\log_2 n)$
- D) $O(n^2)$

21. The following represents the algorithm for ____ sort.

for each array element from the second (`nextPos = 1`) to the last
 Insert the element at `nextPos` where it belongs in the array, increasing
 the length of the sorted subarray by 1 element.

- A) merge
- B) shell
- C) selection
- D) insertion

- ✓ 22. The following is the ____ algorithm.
- Build a heap by rearranging the elements in an unsorted array.
while the heap is not empty
 Remove the first item from the heap by swapping it with the last item in the heap
 and restoring the heap property.
- A) Quicksort
 - B) Bubble sort
 - C) Merge sort
 - D) In-Place Heapsort**

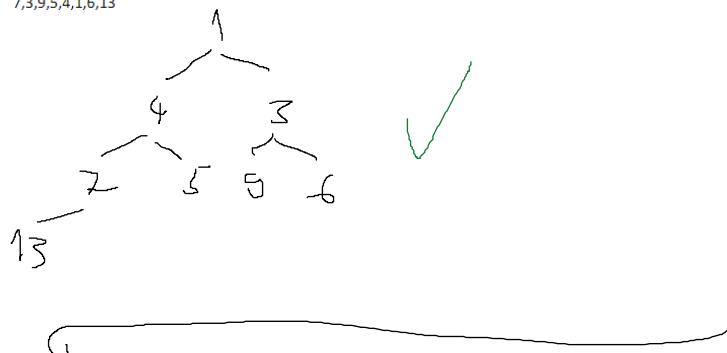
S06 - Ex - Heap Exercises

06 June 2022 19:40

1.

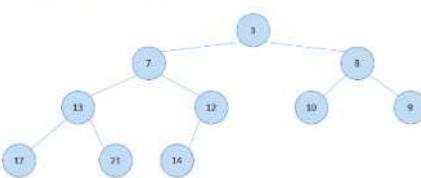
Draw the min-heap, which results from adding the following numbers to an empty min-heap, draw both linked and array representation:

7,3,9,5,4,1,6,13



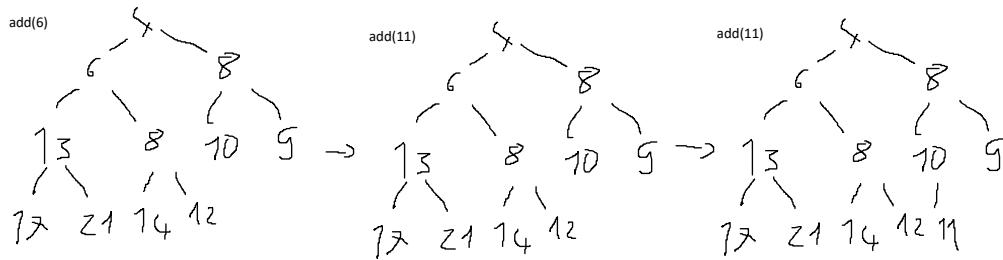
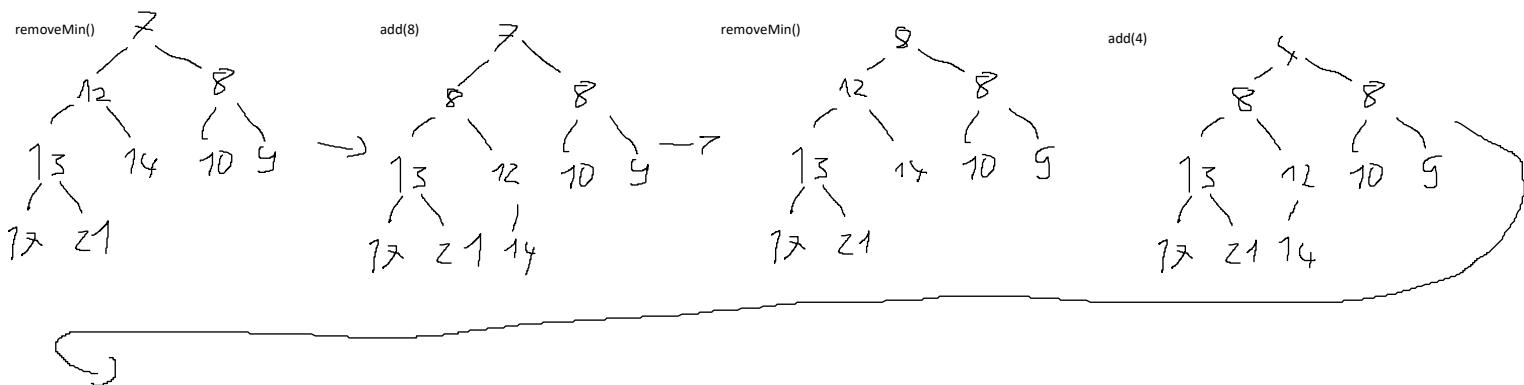
2.

Consider the following min-heap:



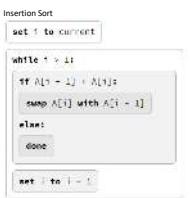
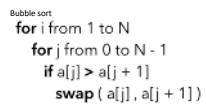
Draw the resulting min-heap after the following operations:

- removeMin()
- add(8)
- removeMin()
- add(4)
- add(6)
- add(11)



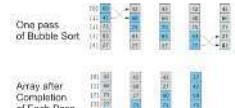
Sort Algorithm Comparison

	Number of Comparisons		
	Best	Average	Worst
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n)$	$O(n)$
Shell Sort	$O(n^{3/2})$	$O(n^2)$	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$



Shell sort

- Improved insertion algorithm
- Divide and conquer approach to insertion sort
- Steps:
 - Sort many subarrays using insertion sort
 - Sort progressively larger arrays
 - Finally sort the entire array
- Array elements are separated by gap
- Start with a large gap
- Decrease the gap on each "pass"
- Reduces work by moving elements farther earlier
- Performance depends on gap values



Array after Completion of Each Pass



```

procedure function mergeSort():
    WHILE length(left) > 0 AND length(right) > 0:
        IF first(left) < first(right):
            append first(left) to result:
            left = rest(left)
        ELSE:
            append first(right) to result:
            right = rest(right)
        IF length(left) == 0:
            append left to result:
        END IF
        IF length(right) == 0:
            append right to result:
        END IF
    END WHILE
    return result
END FUNCTION
Merge sort algorithm
  
```

Heap sort

- Requires no additional storage
- Max Heap - Sorting from lowest number to highest
- Min Heap - Sorting from highest number to lowest

Quick sort

	Merge sort	Bubble sort	Selection sort	Heap sort	Quick sort
A Guaranteed sorting time is $O(n \log n)$.	✗	□	□	✗	□
B Best case sorting time is $O(n)$.	□	✗	□	□	□
C Worst case sorting time is $O(n^2)$.	□	✗	✗	□	✗
D Sorting time is always $O(n^2)$.	□	□	✗	□	□
E Worst case sorting time is $O(n \log n)$.	✗	□	□	✗	□
F Worst case sorting time is $O(n^2)$ but average sorting time is $O(n \log n)$.	□	□	□	□	✗
G Except a few constants no auxiliary memory is required (the array can be sorted in place).	□	✗	✗	✗	(A)
H The distribution of numbers in the array affects search time; as an example if the array is sorted then it will have a different time complexity than if the numbers are randomly distributed.	□	✗	□	□	✗
I The algorithm is usually implemented with recursion.	✗	□	□	□	✗

S08 - Algorithm Design Part Two

02 June 2022 13:17

S09 - Graph Algorithms Part One

02 June 2022 13:18

Graph

Set of vertices and edges that connect pairs of distinct vertices
Representation of graphs

Adjacency Matrix
Adjacency List

Types

Directed
Undirected
Weighted
Disconnected

Tree searching algorithms
Breadth-first search
Uses Queue

Algorithm

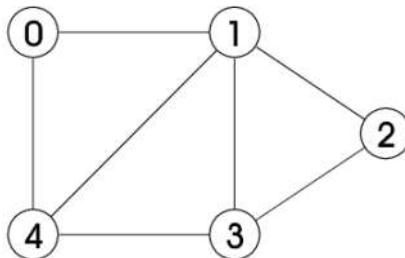
- 1) Take a vertex, u , out of queue and start visiting it
- 2) For all vertices v , adjacent to u
 - a) If v has not been identified or visited
 - i) Mark it identified
 - ii) Place it into queue
 - iii) Add u, v to the breadth first search tree
- 3) We are now done visiting u

Depth-first search

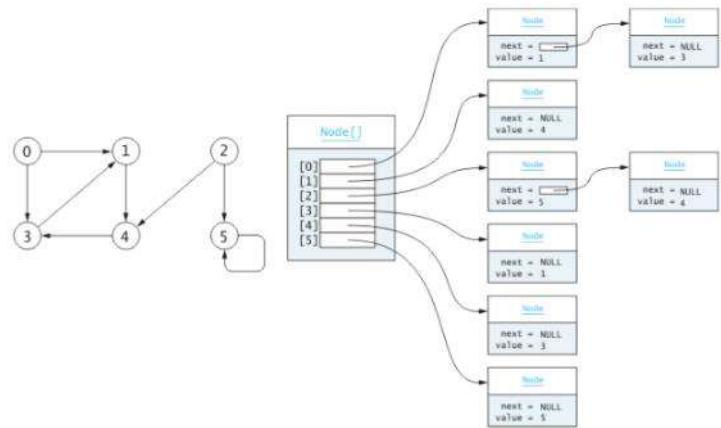
Uses Stack



Graph adjacency examples
Adjacency Matrix



Adjacency List

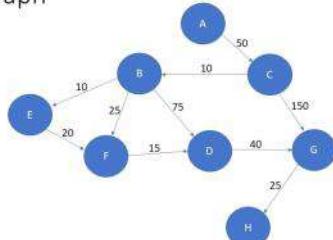


How to trace Prim's algorithm

S	D[v]	P[u]
0	na	na
1	6 - 5	0 - 2
2	1	0
3	5 - 2	0 - 5
4	∞ - 6 - 5 - 3	0 - 2 - 5 - 1
5	∞ - 4	0 - 2

1. Make an adjacency list representation of the graph and an adjacency matrix representation of the graph:

Graph



2. Find the density D of the graph:

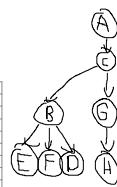
$$\text{Number of edges / number of vertices} \\ 8 / 10 = 0.8$$

3. Which of the representation from uses least memory for this specific graph?
(assuming weight, name and pointers all use one memory word). Adjacency List

4. Do a Breadth-First search of the graph starting from node A.
Show the trace of the algorithm and draw the Breadth-First search tree from your search.

5. Do a Depth-First search of the graph starting from node A.
Show the trace of the algorithm and draw the Depth-First search tree from your search.

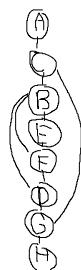
6. Find a topological ordering starting from node A.



Index	Name	Directed						
		From	To	>	>	>	>	>
0	A (0)	C (50)		A	-			
1	B (0)	F (20)	F (25)	D (75)	D (25)	B	-	
2	C (0)	B (10)	G (150)	C (10)	-			150
3	D (0)	G (40)		D	-			40
4	E (0)	F (20)		E	-			
5	F (0)	D (15)		F	-	15	-	
6	G (0)	H (0)		G	-		-	25
7	H (0)			H	-		-	-

Breadth First Search		
Vertex visited	Queue content	Visit sequence
A	C	A
C	GB	AC
G	BH	ACG
B	HEFD	ACGB
H	EFD	ACGBH
E	FD	ACGBHF
F	D	ACGBHED
D	-	ACGBHEDF

Depth First Search		
Vertex visited	Stack Content	Visit sequence
A	C	A
C	BG	AC
G	BH	ACG
B	HEFD	ACGB
H	EFD	ACGBH
E	FD	ACGBHF
F	D	ACGBHED
D	-	ACGBHEDF



S10 - Graph Algorithms Part Two

02 June 2022 13:18

Time Complexity 1 (right: 5%, wrong: -0.5%)

```
public void辗转相除法(int a, int b, int d) {
    int t = a;
    int r = b;
    while (t > 0) {
        if (t % 2 == 1) {
            d += r;
        }
        r = r * 2;
        t = t / 2;
    }
}
```

A. $O(n^2)$

B. $O(n \log n)$

C. $O(n^2 \log n)$

D. $O(n^2 \log n^2)$

E. $O(n \log_2 n)$

Time Complexity 2 (right: 5%, wrong: -1%)

$$T(n) = n\sqrt{\frac{n}{2}} + 3n(\frac{4}{3}\log(n) + 2n)$$

Time Complexity 3 (right: 5%)

What is the time complexity of the following algorithm?

```
for (int i = 0; i < n; i++) {
    for (int j = i+1; j < n; j++) {
        for (int k = j+1; k < n; k++) {
            cout << a[i] << a[j] << a[k];
        }
    }
}
```

A. $O(n^3)$

B. $O(n^2 \log n)$

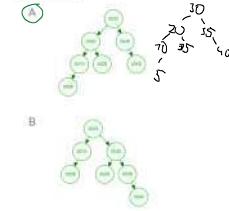
C. $O(n^2)$

D. $O(n^3 \log n)$

E. $O(n^3 \log_2 n)$

Match the left tree to the result of inserting the following numbers in the order listed.

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11

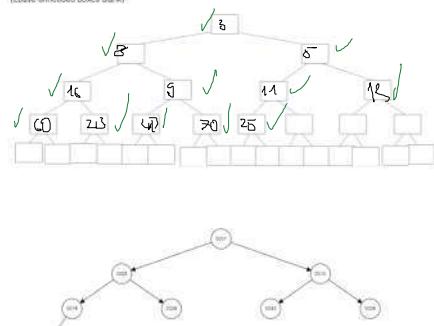


- A. 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 11
B. 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1
C. 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1
D. 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1
E. 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1

Insert the following numbers from left to right into the tree below following the rules of inserting into a min-heap.

23, 11, 5, 16, 42, 25, 13, 60, 8, 9, 70, 3

(Leave unneeded boxes blank)

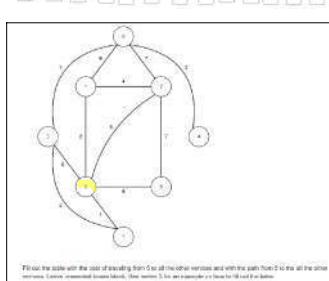
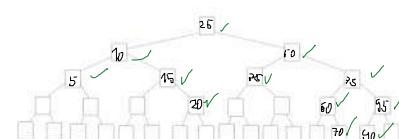
The following search algorithms are used to sort an array of size n . Mark all the correct answers.

	Merge sort	Bubble sort	Selection sort	Heap sort
A. Guaranteed sorting time is $O(n \log n)$.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
B. Best-case sorting time is $O(n)$.	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
C. Worst-case sorting time is $O(n^2)$.	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
D. Sorting time is always $O(n^2)$.	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
E. Worst-case sorting time is $O(n \log n)$.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
F. Worst-case running time is $O(n^2)$ but average sorting time is $O(n \log n)$.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
G. Except a few constants no algorithm necessarily requires that the array can be sorted in place.	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
H. The distribution of the numbers in the array affects search time; as an example if the array is already sorted then it will have different time complexity than if the numbers are randomly distributed.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
I. The algorithm is usually implemented with recursion.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>

Adding to binary search tree (right: 5%)

Below is an empty binary search tree. Insert the following sequence of numbers from left to right:

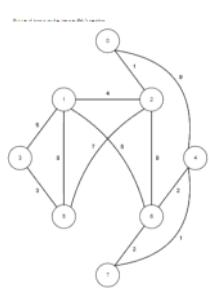
25, 10, 5, 50, 15, 75, 95, 80, 35, 20, 60, 70



Insert into AVL tree (right: 5%, wrong: -1%)

What kind of rebalancing operation, if any, is required to keep the AVL tree from the previous question balanced?

- A. A left rotation.
- B. A left-right rotation.
- C. A right-left rotation.
- D. A right rotation.
- E. No operation is required.



Row	1	2	3	4	5	6	7	8	9	10
1	1	5	2							
2	9	5	3							
3	7	4	5	1						
4	3									
5	9	2	5	0	6					
6	5									
7	1									
8	2	1								
9	5	6	4							
10	7	6	5	4	3					

Quick sort

x x

□

x

□

x x

x ✓

/ x x

x x

x x

x x

Hash table (right: 10%)

Set the following numbers in the hash table using quadratic probing:

2, 5, 18, 3, 13, 17 (leave unneeded boxes blank)

0	1	2	3	4	5	6	7	8	9
	# 17	# 13	# 23	# 3	# 5		# 7	# 18	

7 # 23 # 5 # 18 # 3 # 13 # 17

TODO

02 June 2022 16:42

- Learn AVL rebalancing
- Prim algorithm
- Check - what is big O of this function (S02-Ex2.7)57

Questions to ask at:

- Test exam exercise - Insert into AVL tree - what kind of operation?
- Test exam exercise - Hash table - quadratic probing
- Depth first search
- Breadth first search
- Do the last exercise for graphs

Hash table (right: 10%)

Insert the following numbers in the hash table using quadratic probing:

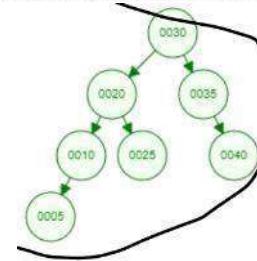
7, 23, 5, 18, 3, 13, 17 (leave unneeded boxes blank)

0	1	2	3	4	5	6	7	8	9
	13		23	3	5		7	18	17

Insert into AVL tree (right: 5%, wrong: -1%)

What kind of rebalancing operation, if any, is required to keep the AVL tree from the previous question balanced?

- A A left rotation
- B A left right rotation
- C A right left rotation
- D A right rotation



- E No operation is required

Time Complexity 1 (right: 5%, wrong: -0.5%)

What is the time complexity of this function:

```
public void mystic(int a, int b, int c) {  
    int i = 0;  
    int r=b;  
    while (i < a) {  
        if (b % 2 == 0) {  
            i++;  
        }  
        b++;  
        for (int j = 1; j <= c; j *= 2) {  
            for(int q=0; q<r; q++) {  
                System.out.println(q);  
            }  
        }  
    }  
}
```

- $O(a b^2 c^3)$
- $O(a b^2 c^2)$
- $O(a b^2 \log(c))$
- $O(a b^2 \sqrt{c})$
- $O(a \log(b) c^2)$
- $O(a \sqrt{b} \log(c))$
- $O(a \log(b) \sqrt{c})$
- $\cancel{O(a b \log(c))}$
- $O(a \log(b) \log(c))$
- $O(\log(a) b c^2)$

Time Complexity 2 (right: 5%, wrong: -1%)

Given the time function:

$$T(n) = n\sqrt{\frac{n}{2}} + 3n(4 \log(n) + 2n)$$

What is the big Oh function for T(n)?

A $O(n)$

B $O(n^2)$

C $O(n^3)$

D $O(n^2 \log(n))$

E $O(n \log(n))$

Section 3

Time Complexity 3 (right: 5%)

What is the time complexity of the following operations?

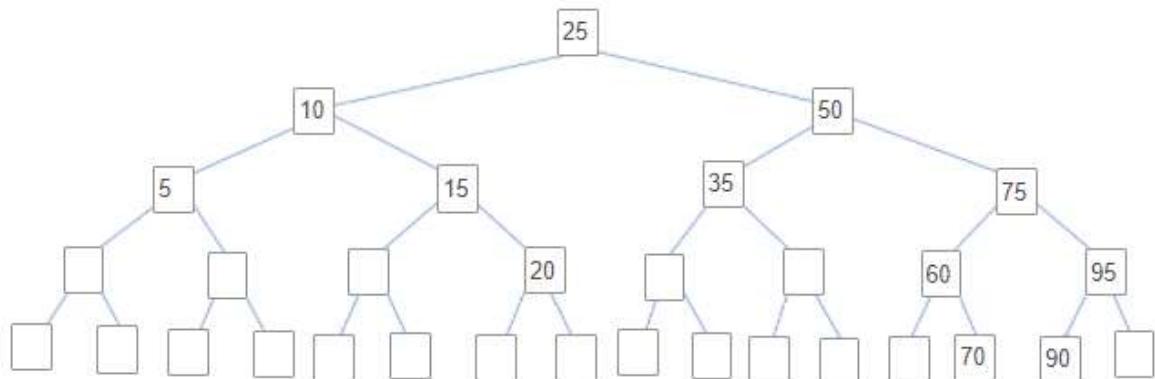
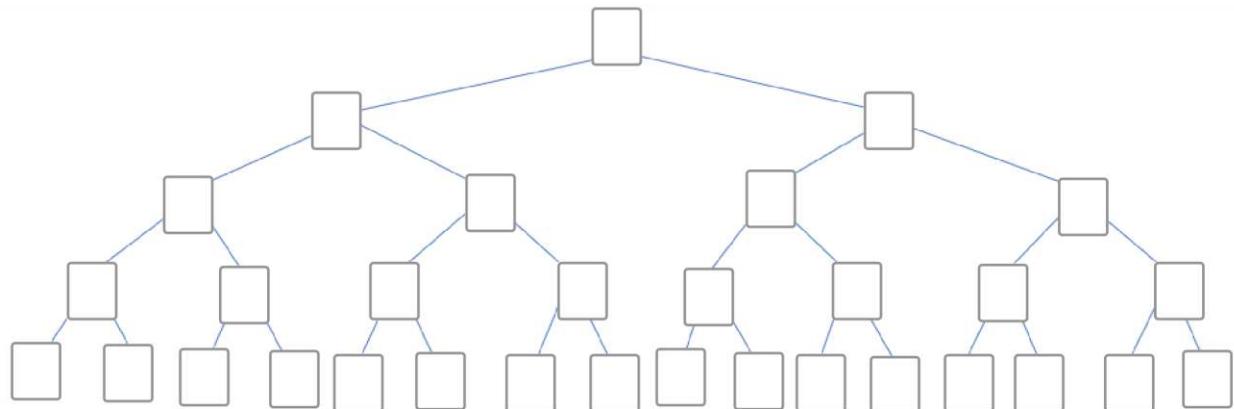
	O(1)	O(log n)	O(n)	O(n(log(n))
Finding the biggest element in an unordered list	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>
Sorting a list with heap sort	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
Removing from a binary search tree.	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>
Adding an element to a balanced binary search tree	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>
Adding an element to a priority queue implemented with a heap.	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>

Section 4

Adding to binary search tree (right: 5%)

Below is an empty binary search tree. Insert the following sequence of numbers from left to right:

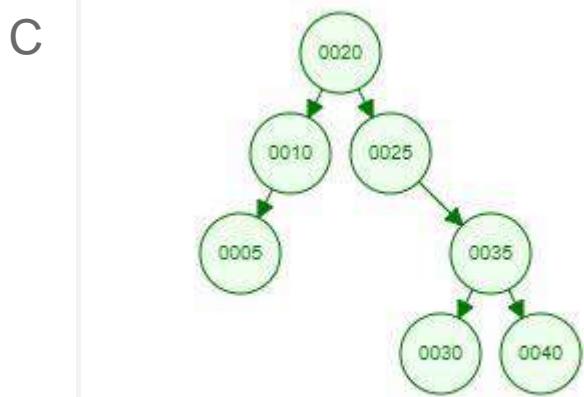
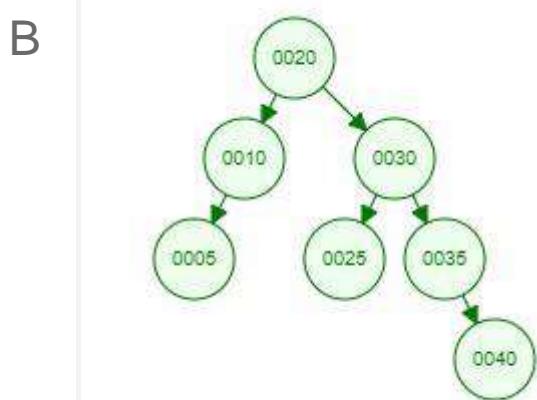
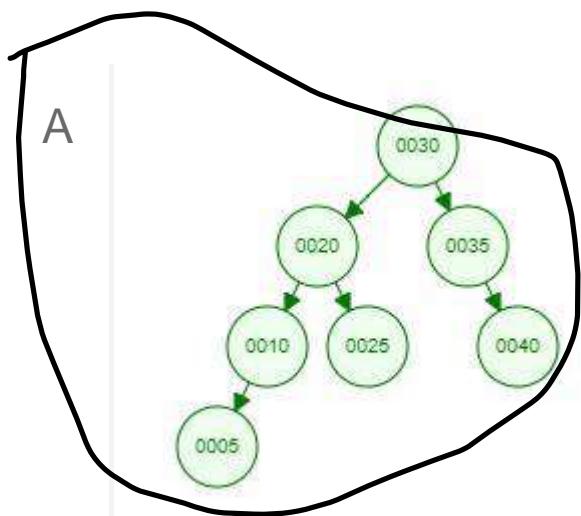
25, 10, 5, 50, 15, 75, 95, 90, 35, 20, 60, 70



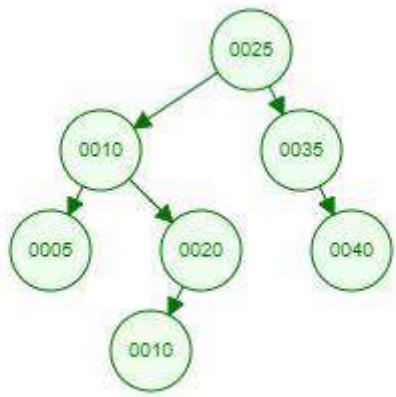
Section 5

Insert into AVL tree (right: 10%, wrong: -2%)

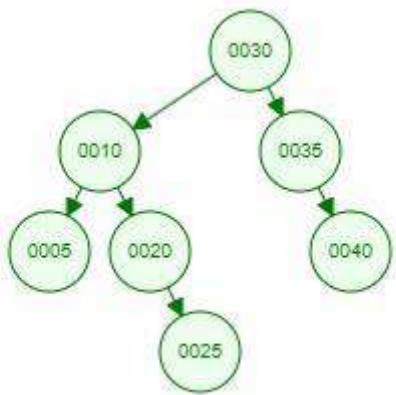
Mark the AVL tree that is the result of inserting the following number in the order listed:
10, 20, 30, 25, 35, 40, 5



D



E



Insert into AVL tree (right: 5%, wrong: -1%)

What kind of rebalancing operation, if any, is required to keep the AVL tree from the previous question balanced?

- A A left rotation

- B A left right rotation

- C A right left rotation

- D A right rotation

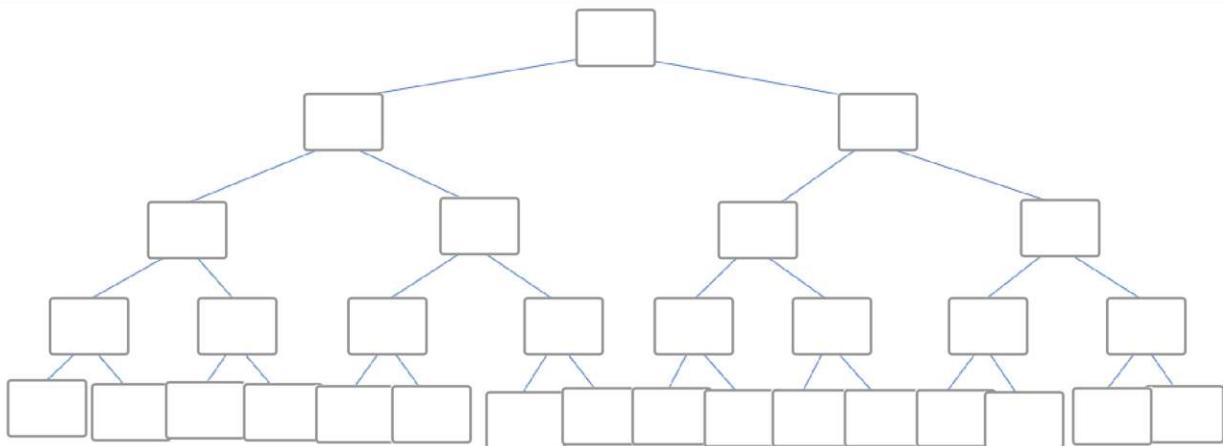
- E No operation is required

Heaps (right: 10%)

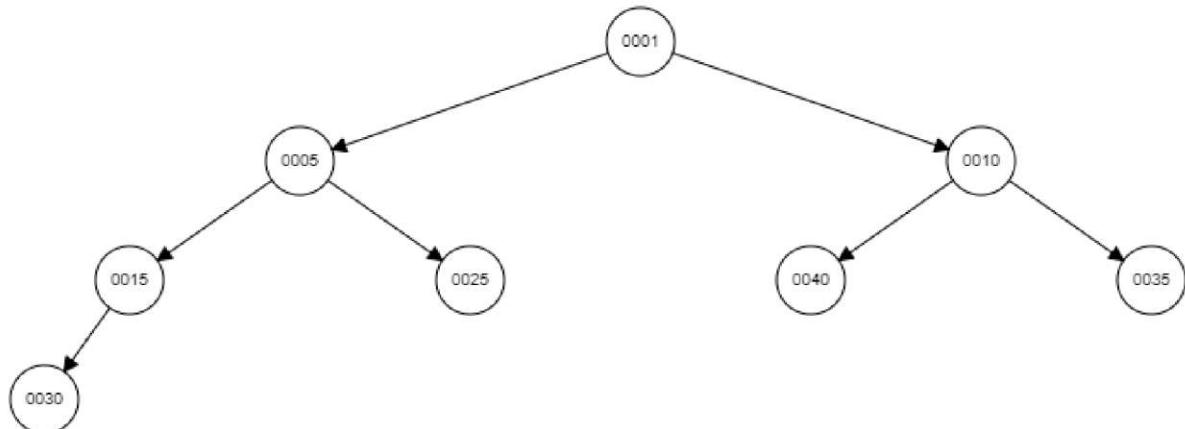
Insert the following numbers from left to right into the tree below following the rules of inserting into a min-heap.

23, 11, 5, 16, 42, 25, 13, 60, 8, 9, 70, 3

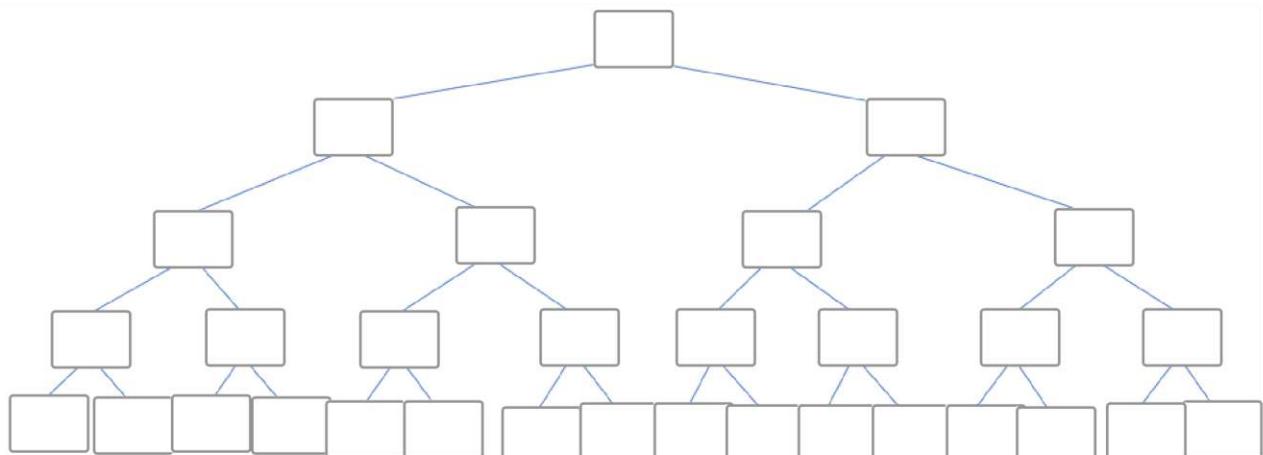
(Leave unneeded boxes blank)

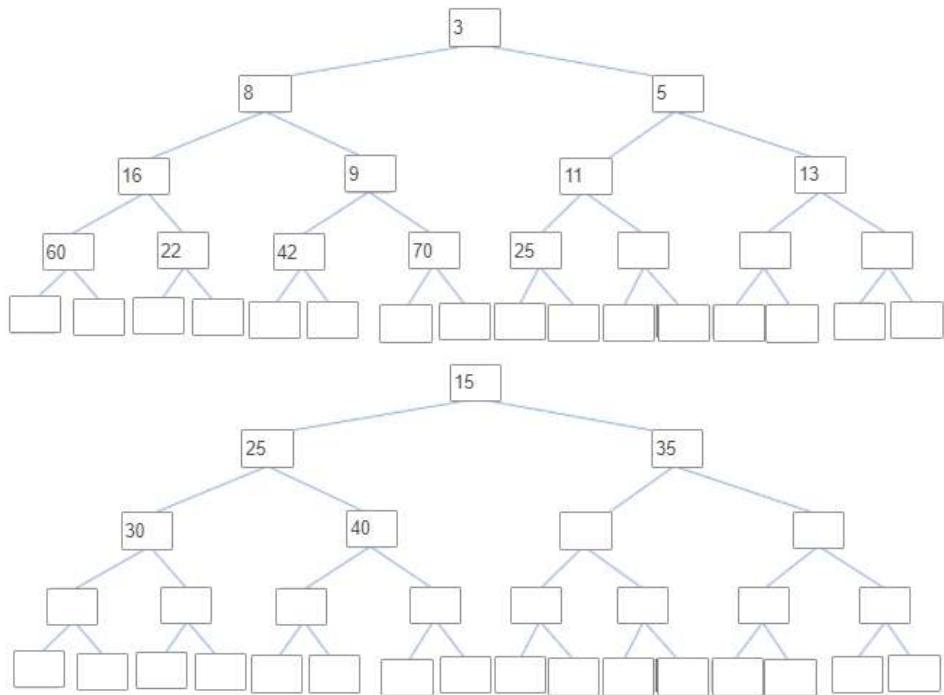


Part two:



RemoveMin() is called three times on the heap above. Insert into the heap below what it looks like after the three RemoveMin() calls. (Leave unneeded boxes blank)

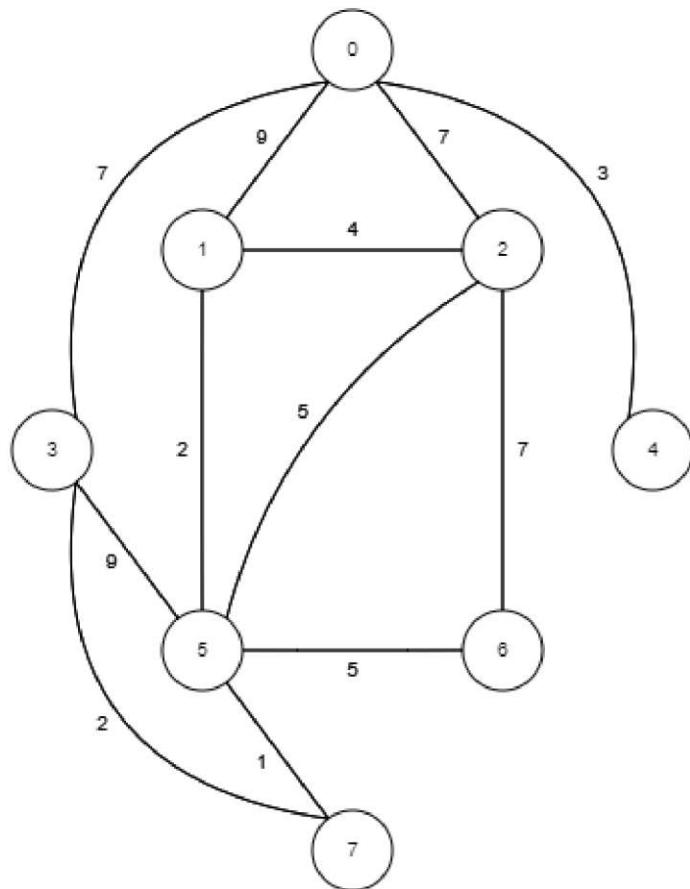




Section 8

Graphs (right: 15%)

Find the shortest path from 5 to all other nodes using Dijkstra's algorithm starting from 1:



Fill out the table with the cost of traveling from 5 to all the other vertices and with the path from 5 to the all the other vertices. Leave unneeded boxes blank. See vertex 3 for an example on how to fill out the table

Vertice	Cost	Path
0	[]	5->[]->[]->[]->[]
1	[]	5->[]->[]->[]->[]
2	[]	5->[]->[]->[]->[]
3	3	5-> 7 -> 3
4	[]	5->[]->[]->[]->[]
5	0	na
6	[]	5->[]->[]->[]->[]
7	[]	[]

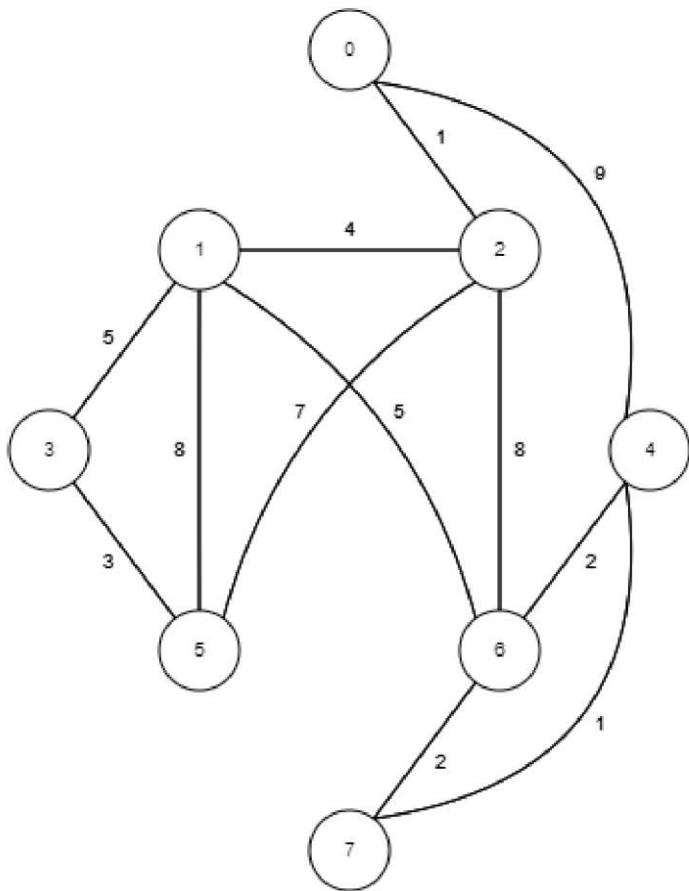


Vertex	Cost	Path
0	10	5-> 7 -> 3 -> 0 -> [] -> []
1	2	5-> 1 -> [] -> [] -> [] -> []
2	5	5-> 2 -> [] -> [] -> [] -> []
3	3	5-> 7 -> 3
4	13	5-> 7 -> 3 -> 0 -> 4 -> []
5	0	na
6	5	5-> 6 -> [] -> [] -> [] -> []
7	1	5-> 7 -> [] -> [] -> [] -> []

Section 9

Graphs (right: 15%)

Find the minimum spanning tree using Prim's algorithm:



Show the trace of using the algorithm (leave unneeded boxes blank).

	$d[v]$	$p[v]$
0	[] [] [] [] [] [] []	[] [] [] [] [] []
1	[] [] [] [] [] []	[] [] [] [] []
2	[] [] [] [] [] []	[] [] [] [] []
3	[] [] [] [] [] []	[] [] [] [] []
4	[] [] [] [] [] []	[] [] [] [] []
5	na	na
6	[] [] [] [] [] []	[] [] [] [] []

	$d[v]$	$p[v]$
7	[dashed box]	[dashed box]

■ 0 ■ 1 ■ 2 ■ 3 ■ 4 ■ 5 ■ 6 ■ 7 ■ 8 ■ 9
■ 10 ■ 11 ■ 12 ■ ∞

	$d[v]$	$p[v]$
0	∞ 1 [dashed box]	5 2 [dashed box]
1	8 5 [dashed box]	5 3 [dashed box]
2	7 4 [dashed box]	5 1 [dashed box]
3	3 [dashed box]	5 [dashed box]
4	∞ 9 2 [dashed box]	5 0 6 [dashed box]
5	na	na
6	∞ 5 [dashed box]	5 1 [dashed box]
7	∞ 2 1 [dashed box]	5 6 4 [dashed box]

Section 10

Sorting (right: 15%)

The following search algorithms are used to sort an array of size n . Mark all the correct statements for each algorithm. Note there can be more than one correct answer for each algorithm.

	Merge sort	Bobble sort	Selection sort	Heap sort	Quick sort
A Guaranteed sorting time is $O(n \log n)$.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
B Best case sorting time is $O(n)$	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
C Worst case sorting time is $O(n^2)$	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
D Sorting time is always $O(n^2)$	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
E Worst case sorting time is $O(n \log n)$	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
F Worst case sorting time is $O(n^2)$ but average sorting time is $O(n \log n)$.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
G Except a few constants no auxiliary memory is required (the array can be sorted in place).	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/> (✓)
H The distribution of the numbers in the array affects search time; as an example if the array is almost sorted then it will have a different time complexity than if the numbers are randomly distributed.	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
I The algorithm is usually implemented with recursion.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>

Section 11

Hash table (right: 10%)

Insert the following numbers in the hash table using quadratic probing:

7, 23, 5, 18, 3, 13, 17 (leave unneeded boxes blank)

0	1	2	3	4	5	6	7	8	9
<input type="text"/>									

<input type="text"/> 7	<input type="text"/> 23	<input type="text"/> 5	<input type="text"/> 18	<input type="text"/> 3	<input type="text"/> 13	<input type="text"/> 17
------------------------	-------------------------	------------------------	-------------------------	------------------------	-------------------------	-------------------------

0	1	2	3	4	5	6	7	
<input type="text"/>	<input type="text"/> 17	<input type="text"/> 13	<input type="text"/> 23	<input type="text"/> 3	<input type="text"/> 5	<input type="text"/>	<input type="text"/> 7	<input type="text"/>