

Introduction to Computer Science using JavaScript

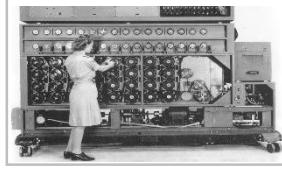
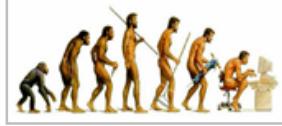
Ritwik Dutta

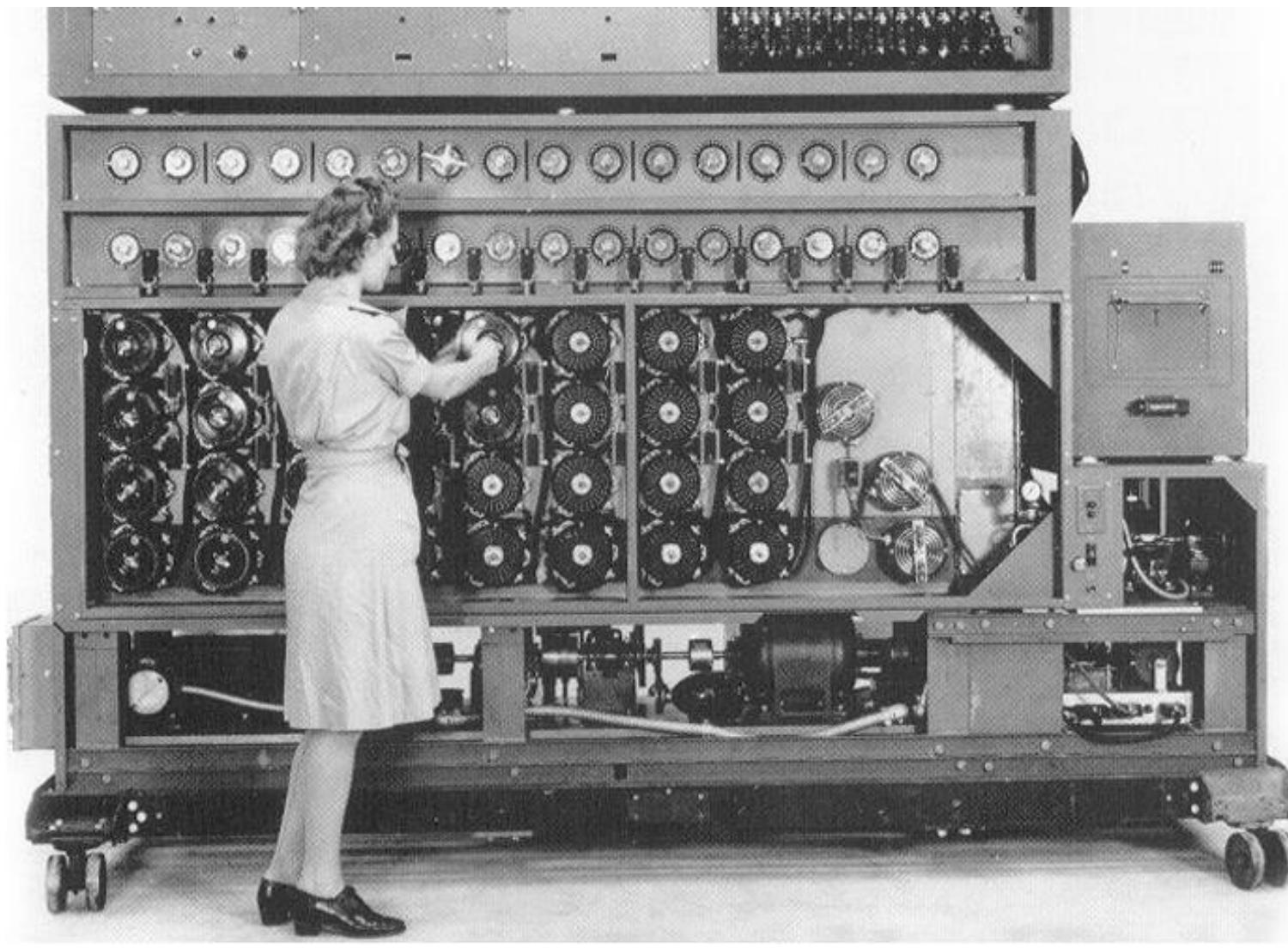
Archbishop Mitty High School

<http://ritwikd.com>

Table of content



-  1. Computers and programming
-  2. The evolution
-  3. Inside programming languages
-  4. Why JavaScript?
-  5. JavaScript programming basics
-  6. Basic data structures
-  7. Introductory algorithms & problems



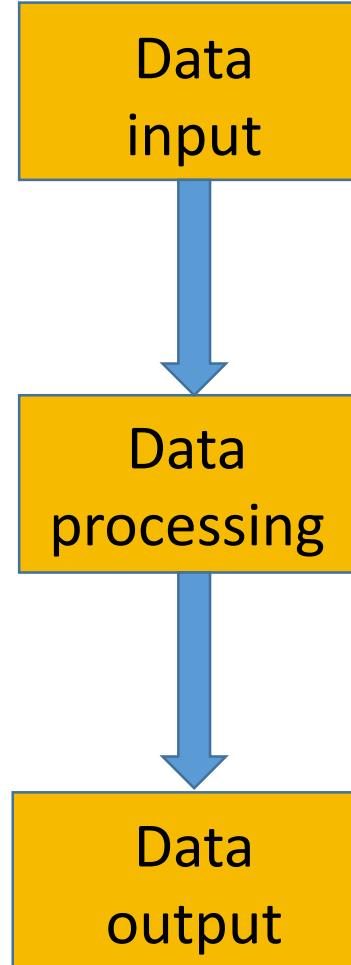
Computers & programming

Computer definition



- Machine that executes **tasks** according to **instructions**
- Tasks are basic operations
- Instructions are commands in a machine-readable format
- Tasks operate on input data and generate output data

Structure and flow



Input devices: Mouse, keyboard, mic, camera, scanner

Processing devices: Central Processing Unit (CPU),
Arithmetic Logic Unit (ALU)

Output devices: monitor, printer, speaker, modem

Structure of a modern computer

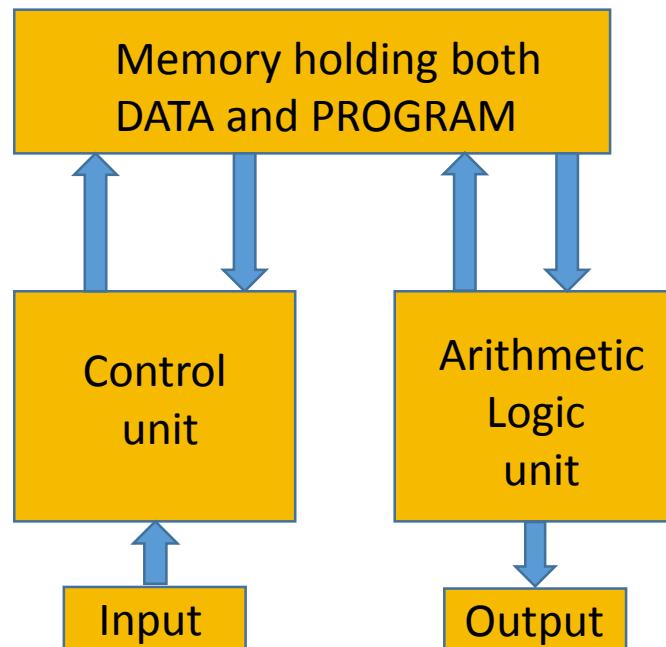


- Basic components
 - **Sequencer** determines order of instructions to be fetched
 - **Fetch unit** gets data and instructions stored in memory
 - **Control unit** determines type of operations to be performed
 - **CPU** executes different arithmetic and logic operations on different data according to instructions in the **Arithmetic Logic Unit (ALU)**
 - **Store unit** stores processed data back to memory
 - **Peripherals** allow information to be stored or retrieved from an external source (e.g., CDROM drives, USB flash drivers, etc.)
- Modern digital computers based on integrated circuits are many times smaller and more capable than their mechanical and analog predecessors

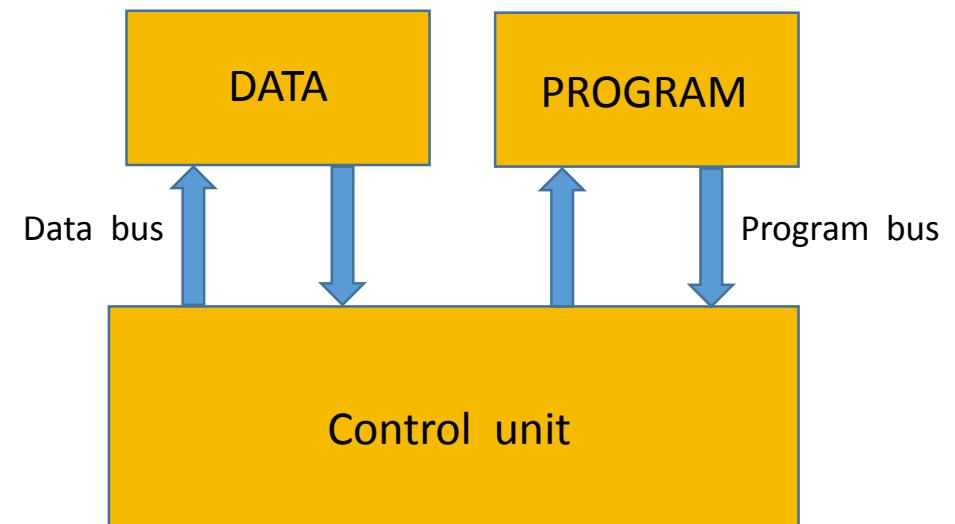
Where are instructions and data stored?



- Instructions and data are stored in and fetched from memory
- Both can be in the same memory - Von Neumann architecture
- Both can be in different memories - Harvard architecture



Von Neumann architecture



Harvard architecture

Introducing registers



- Regular memory in a computer is often big and slow
- To speed up operations, there is a small amount of very fast storage using registers often implemented inside the CPU
- Registers are used to quickly accept and store data that are being used immediately by the CPU
- Registers are usually denoted by the alphabet letters A, B, C, ... *etc.*
- There may be multiple registers of each type designated by A0, A1, A2, ... , *etc.*

Computer hardware



- Hardware refers to physical components that make up a computer system
- These components are physically connected to the computer and can be physically touched
- There are many different kinds of hardware that can be installed inside or connected to the outside of a computer
 - Examples of internal hardware are the CPU, the computer memory, the CDROM drive, *etc.*
 - Typical examples of external hardware are keyboard, mouse, computer monitor, printer, *etc.*

Computer algorithm



- An algorithm in general is a step by step procedure or formula to do a task or operation
- In order to tell a computer what calculation(s) to do, one needs to also specify how exactly to do it
- This “how-to-do-it” is nothing but a computer algorithm
- A computer algorithm is specified by a sequence of well-defined machine-readable instructions
- Algorithms are used for calculation, data processing, and sometimes reasoning
- An efficient algorithm is one that executes (produces the results) quickly and/or uses fewer resources ... we refer to these metrics as the **time** and **space complexity** of an algorithm respectively

Practical algorithm



Different algorithms to get to Tom's house from the airport

ALGORITHM 1

1. Collect luggage
2. Go to the taxi stand
3. Get into a cab
4. Give the driver Tom's address
5. Arrive at Tom's house

Algorithm 2 takes more time (buy ticket, slow bus)

Algorithm 2 uses more resources (bus, Tom's car)

ALGORITHM 2

1. Collect luggage
2. Go to the bus stand
3. Get on to bus no. 8
4. Buy ticket for zone 2
5. Get down at 6th street
6. Call Tom's cell phone
7. Tom picks you up in his car
8. Arrive at Tom's house

Computer program



- A program is a sequence of instructions written to perform a certain task
- The program implements a formula or an algorithm to do a certain task, but is itself not the formula or the algorithm
- A programmer decides on a suitable algorithm and writes the computer program, often in a human readable form, called the **source program**
- The source program is first loaded or stored in the computer memory
- Next, the program is converted to a machine readable form
- The program instructions are then fetched from memory and executed by the CPU inside the computer
- It is highly desired that the instructions implement an efficient algorithm

Computer software



- Software (SW) is a generic term used to distinguish it from hardware (HW)
- SW comprises a set of instructions and their associated data stored on a media such as flash, disk, or tape drives
- These instructions constitute a computer program
- Can think of software as a computer program stored on a media
- A computer program directs a computer's processor to perform specific operations
- Software programs are written in specific languages
- There are two classes of languages: **low-level languages** and **high-level languages**

Low-level programming (1)



- A low-level programming language is what a computer or a machine understands
- This low level programming language is called the **machine language**
- Machine language is the native binary language in terms of 1's and 0's that a machine understands
- Takes specialized knowledge to program in machine code

Machine instruction	Machine operation
0000 0000	Restart
0000 0010	Stop
0001 0000	Skip next instruction

Low-level programming (2)



- A machine language instruction explicitly tells the CPU what operation it is supposed to do, where in memory to get the operands from, how exactly to do the computation with those operands, and where in computer memory to store the results of the calculation
- Each type of CPU has its own unique machine language
- Humans are not able to think in terms of or interpret a machine language easily
- Programmers therefore commonly use a more English-like high-level language (for programming) to develop their source code that ultimately gets translated into the machine language

High-level programming



- High-level languages (such as Basic, C, JavaScript, etc.) are very English-like, hence easier to conceive of and interpret
- Programmers therefore prefer to program in high-level languages
- A computer does not understand any high-level language
- The computer needs all its instructions to be given in terms of a low-level machine language that it can understand
- Programs written by a programmer need to be translated from a high-level language into a machine language understood by the target computer



Compiler(1)

- One level of abstraction up from machine code, a **compiler** is a software program that translates a source code written in a high-level programming language to a lower level **assembly language** for a target machine
- The most common reason for this translation is to create an **executable program** that the computer understands and can execute
- In the process of translation, a good compiler also gives warnings and flags errors in programming

Assembly instruction	Machine operation
MOVE A0, 2000	Copy contents of register A0 to memory location 2000
MOVE (B0), D2	Copy contents of location pointed to by register B0 to register D2
ADD #15, C3	Add 15 to register C3 and put sum in C3

Compiler(2)



- Compilers can also optimize and generate more efficient target code
- Example

```
for i = 1 to 1000 do  
    A = B/C;
```

inefficient



```
A = B/C;  
for i = 1 to 1000 do  
    skip; // do nothing
```

optimized

Assembler, linker, loader



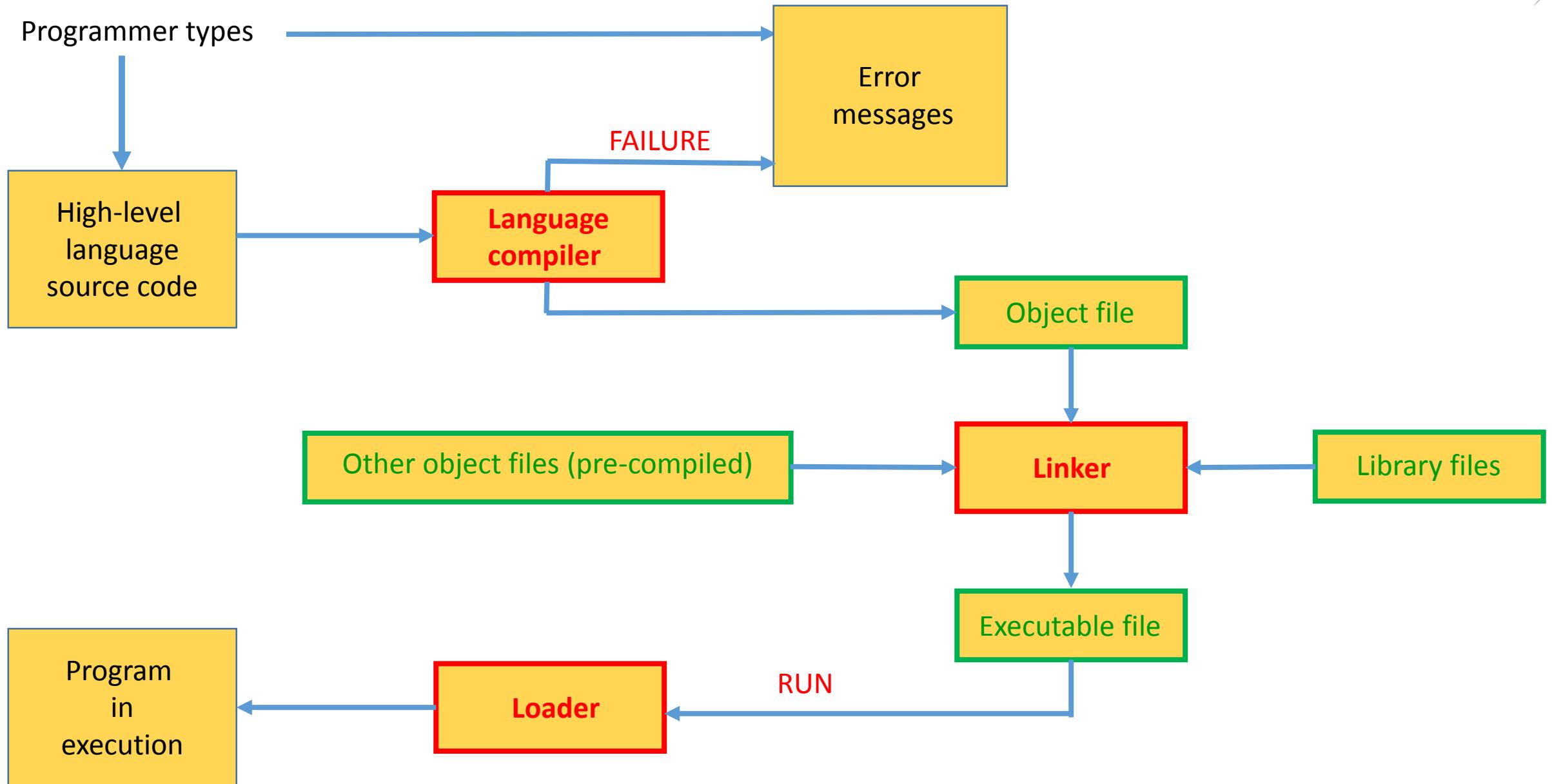
- Assembly language is less sophisticated than high-level language but more sophisticated than low-level machine language
- Assembly language code is translated into machine language by an **assembler**
- The assembler converts assembly language programs into **object files**
- Object files contain machine instructions, data, and information needed to place instructions properly in the computer memory
- A **linker** merges different object files produced by an assembler, and creates common **executable file**
- The executable file is loaded into the computer's memory by a **loader**
- Executable instructions are then fetched from memory and executed by the CPU

Compiled program simple flow



- A high level language program is written by a programmer
- The program is compiled by a compiler into the assembly language of the target CPU
- The assembly program is assembled by the assembler into machine language
- The assembled object files are linked by the linker into a single executable object file
- The object file is loaded by the loader into the computer memory for execution
- The instructions are fetched from the memory by the fetch unit
- The fetched instructions and data are executed by the CPU
- The results are stored in the memory by the store unit

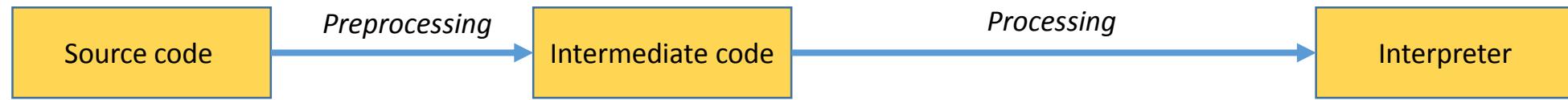
Putting it in a picture



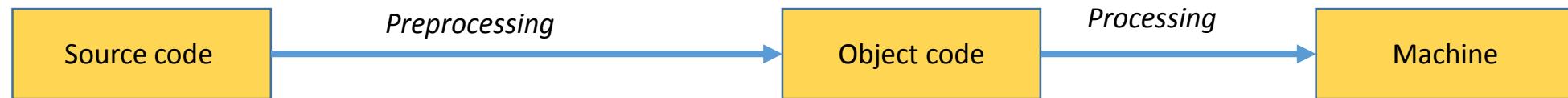
Interpreters versus compilers



- Both read input source code and analyze it
- Interpreters
 - Execute single instructions without object code generation, flag errors if any
 - Good for debugging & interaction



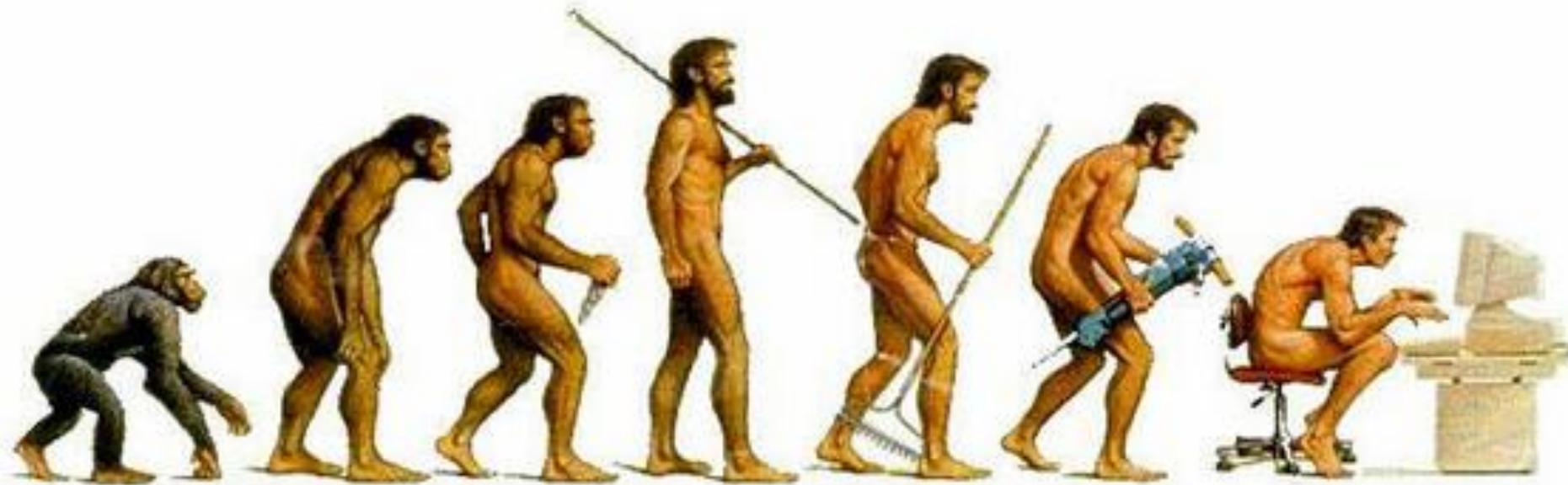
- Compilers
 - Read and analyze entire program, flag errors if any
 - “Improve” the program by making it more efficient
 - Generate object code executed on machine



Typical implementations



- Compilers
 - FORTRAN, C, C++, Java, COBOL, etc.
 - Strong need for optimization in many cases
- Interpreters
 - Python, PERL, Ruby, Java VM
 - Particularly effective if interpreter overhead is low relative to execution cost of individual statements
- Hybrid
 - Compile Java source to byte codes – Java Virtual Machine language
 - Interpret byte codes directly, or compile some or all byte codes to native code

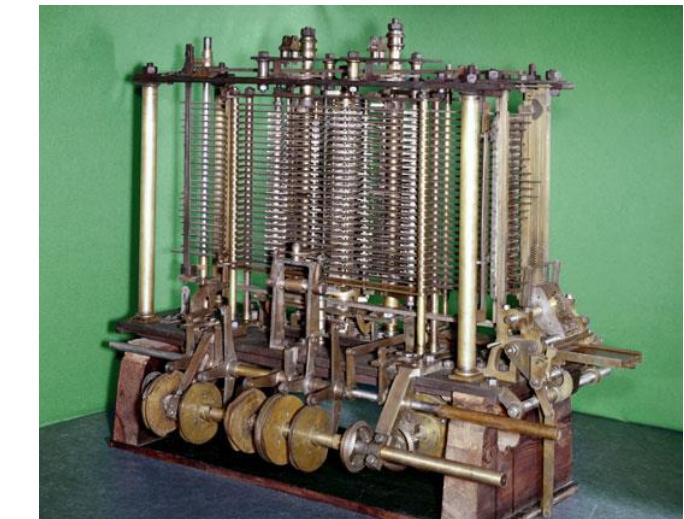
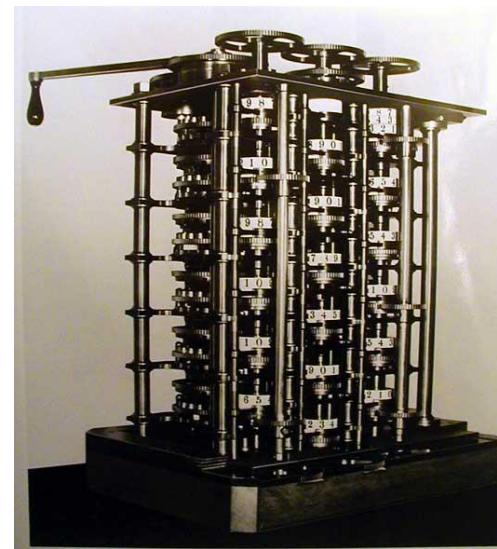
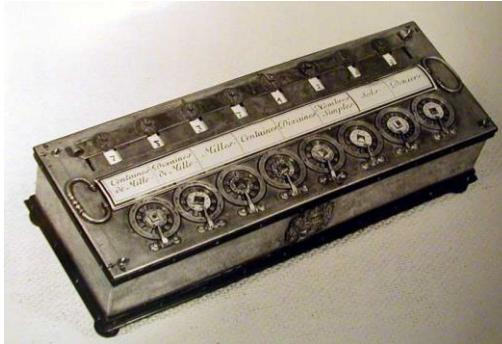


The evolution

Mechanical devices



- Pascal calculator (1642)
 - Performed addition using gears
- Leibniz stepped reckoner (1672)
 - Addition, multiplication, division, square roots using cylindrical wheels with movable carriage
- Babbage difference engine (1822)
 - Produced table of numbers
- Babbage analytical Engine (1837)
 - Performed calculations based on punched card instructions



Electro-mechanical devices



- Hollerith's tabulating machine (1889)

- Holes punched in cards represented information to be tabulated ... used for tabulation in US census



- Mark I (1944)

- Sophisticated calculator from Harvard & IBM that used mechanical telephone replay switches to store information and accepted data on punch cards

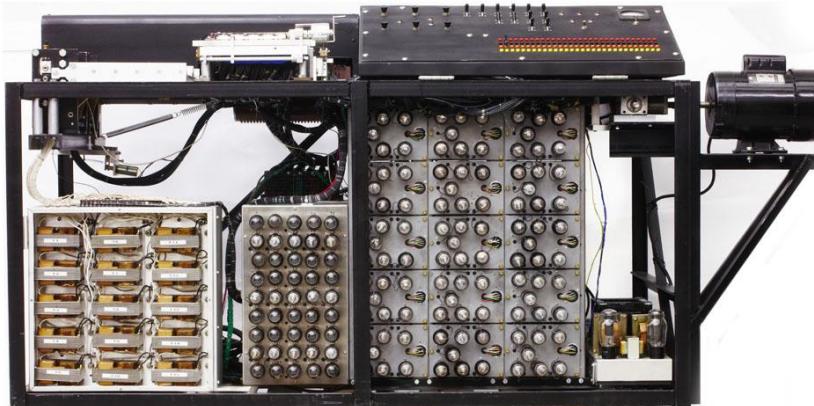


First generation computers



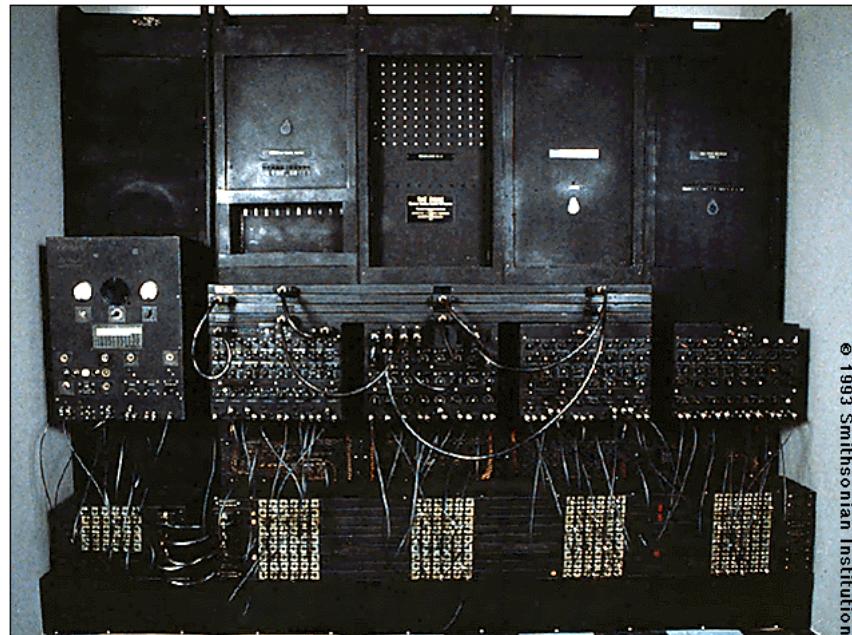
- Atanasoff-Berry computer (1939)

- Used vacuum tubes doing binary arithmetic
- Stored info by electronically burning holes in sheets of paper



- ENIAC (1943)

- Eckert and Mauchly
- Electronic Numerical Integration and Calculator
- 30 tons, 1500 sq ft., 17,000+ vacuum tubes
- Solved a problem in 20 min that otherwise took three days to solve



© 1993 Smithsonian Institution

Stored program computer(1)



- Alan Turing

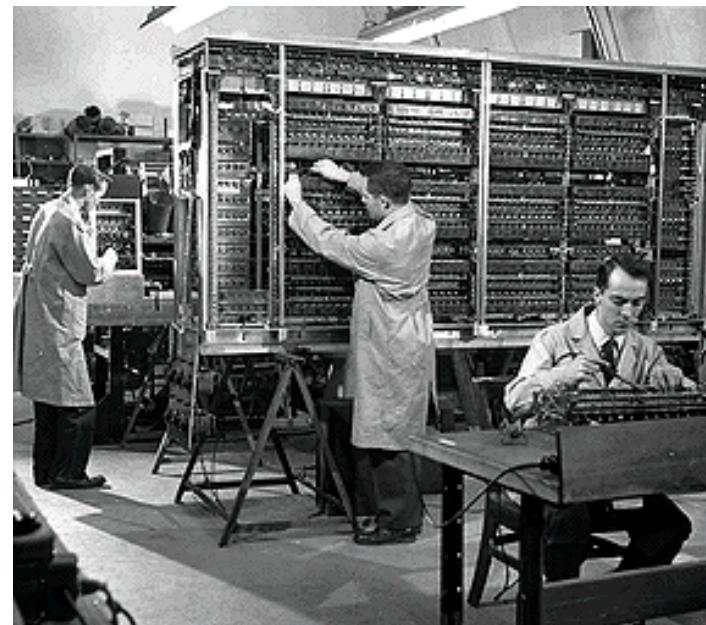
- Developed idea of “universal machine”
- Perform many different tasks by changing a program (list of instructions)

- Von Neumann

- Presented idea of stored program concept
- The stored program computer would store computer instructions in a memory and execute on a CPU

- EDVAC and EDSAC (1944)

- Von Neumann, Mauchly and Eckert
- Binary serial computers that recognized machine language
- Solved many problems by simply entering new instructions stored on paper tape



Stored program computer(2)



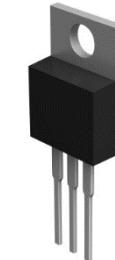
- UNIVAC (1951)
 - Mauchly and Eckert
 - First computer language C-10 developed by Betty Holberton who also developed first keyboard and numeric keypad
 - The UNIVAC I delivered to the U.S. Census Bureau was the first general purpose computer for commercial use



Second generation computers



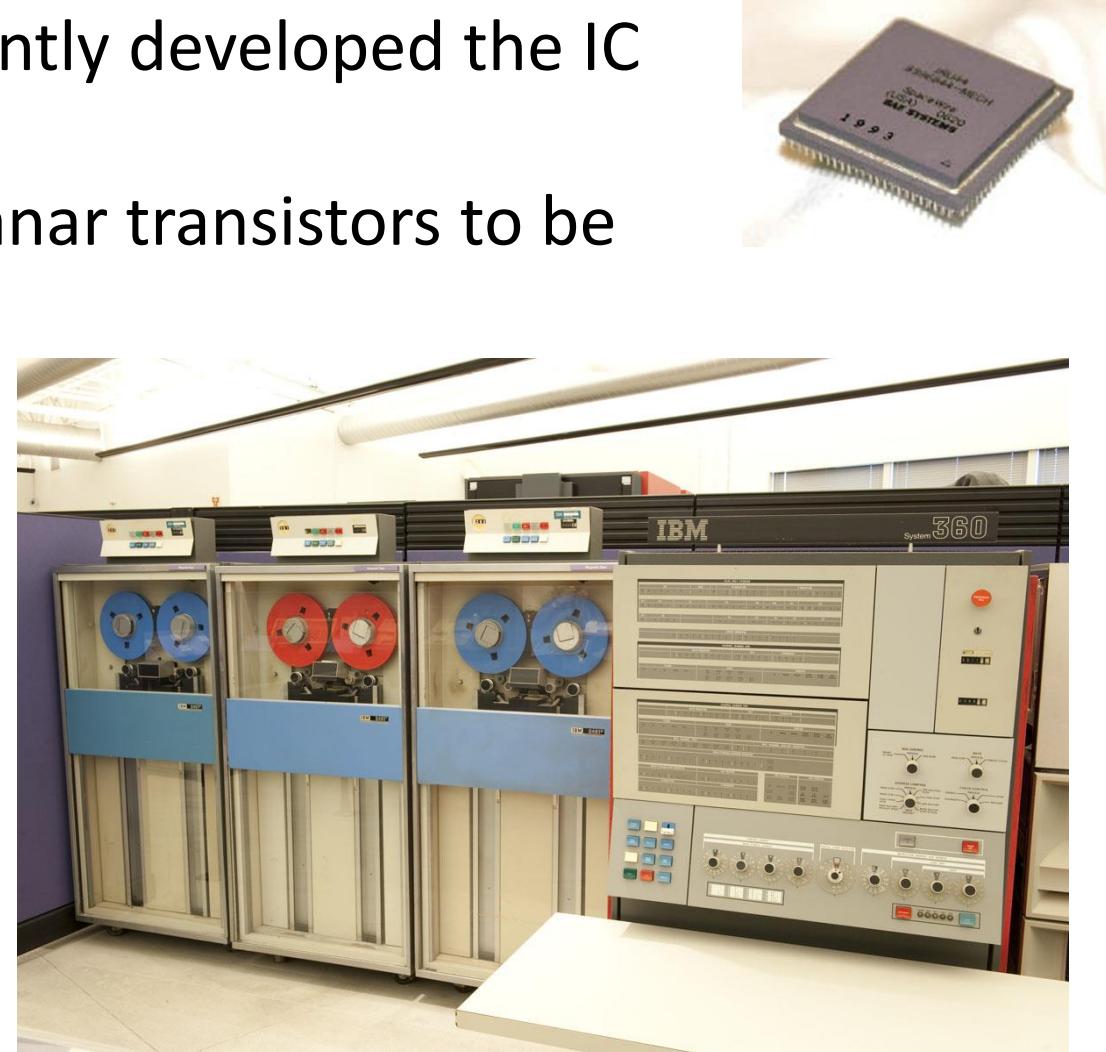
- Bell Labs transistor (1947)
 - Shockley, Bardeen, Brittain invented transistor that replaced vacuum tubes since less expensive and increased calculating speeds
- IBM model 650 (1960)
 - Medium sized computer
 - Magnetic tape and high speed reel-to-reel tape machines replaced punched cards
 - Gave computers ability to read (access) and write(store) data quickly and reliably



Third generation computers



- Integrated circuits (ICs)
 - Kilby and Noyce – working independently developed the IC (chip)
 - Allowed intricate circuits based on planar transistors to be printed and etched on silicon surface
- IBM 360 (1964)
 - One of the first mainframe computers using ICs
 - Communication with the mainframe via terminals
 - Covered range of applications, from small to large, both commercial and scientific

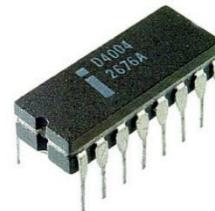


Fourth generation computers



- Microprocessor (1970)

- Intel's invention of entire CPU on a chip made it possible to build the microcomputer (PC)
- Altair 8800 was one of first PCs in 1975
- Wozniak and Jobs designed and built first Apple Computer in 1976
- IBM introduced IBM-PC in 1981

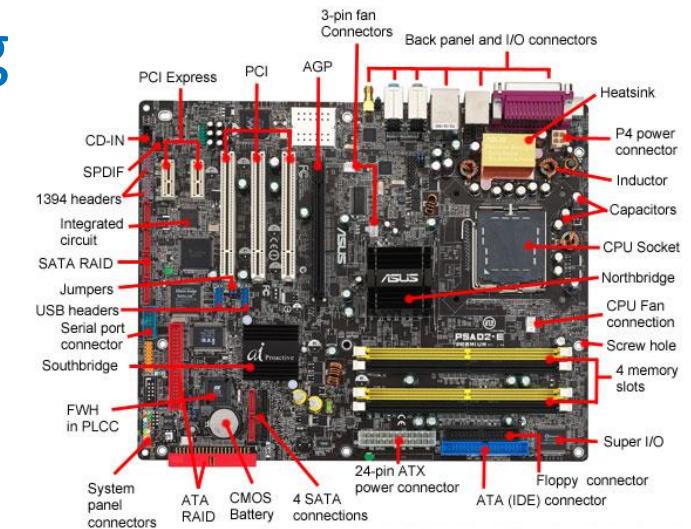


Today's personal computer (PC)



- Components plugged into a motherboard featuring

- BUS:** interconnection network that connects all components and allow data and control signals to travel from one to the other
- RAM slot:** where system memory chips are plugged into for the computer load the OS and drivers, thus making instruction processing faster
- CHIPSET:** *northbridge* IC responsible for communications between CPU and memory, the *southbridge* IC is responsible for the hard drive controller, I/O controller and integrated hardware such as sound card, video card, USB, PCI, ISA, IDE, BIOS, and Ethernet
- CPU socket(s):** socket(s) where CPUs are connected; CPU contains the Arithmetic Logic Unit (ALU) that processes data and controls the flow of data between various units



Programming language evolution



- The first computer program was written by Ada Lovelace- in 1842-43
- The first “modern” language Plankalkül, designed by Konrad Zuse, was described in 1943, but implemented in 1998
- Languages invented in 1950-1960 are still used today
 - Fortran, LISP, COBOL, BASIC
- Most languages used today were invented in 1960-1970
 - C, Pascal, C, ML, SQL
- Many internet languages were developed in early/mid 1990
 - Python, Java, JavaScript, PHP



Inside programming languages

Philosophy of a programming language



- “... a language intended for use by a person to express a process by which a computer can solve a problem” - Hope and Jipping
- “... a set of conventions for communicating an algorithm” - E. Horowitz
- “... the art of programming is the art of organizing complexity” - Djikstra

Reality of a programming language



- Comprises vocabulary and rules designed to communicate instructions to a computer
- Contains abstractions for defining and manipulating data structures and controlling execution flow
- Usually split into two components: syntax (form) and semantics (meaning)
- High-level programming languages are portable across machine architectures and get compiled into low-level assembly language by compiler
- Assembly language gets assembled into executable machine code by assembler

Desires of a programming language



- **Readable and maintainable**
 - Comments, names, syntax
- **Simple to learn and use**
 - Small number of concepts that can be combined to do complex stuff
- **Portable**
 - Language standardization
- **Offers abstraction**
 - Control and data structures that hide detail, easy to express idea
- **Compact**
 - Express ideas concisely
- **Efficient**
 - Efficiently translated into machine code, efficiently executed, acquire as little space in the memory as possible

Example languages



- AI symbolic computation (Haskell, Lisp, Prolog)
- Scientific computing (Fortran)
- Business report generation (COBOL)
- Systems programming low level (C)
- Scripting (Perl, Python, TCL)
- Distributed systems mobile computation (Java)
- Web (PHP, JavaScript)

Why so many languages?



- Allows choosing the right language for a given problem or goal
- If all you have is a hammer, every problem looks like a nail



Object-oriented programming



- A programming paradigm
- Main feature: communication between abstract objects
- Characteristics
 - “Objects” collect both data and operations
 - “Objects” provide data abstraction
- Key operations: Message Passing or Method Invocation

Object-oriented example



JAVASCRIPT

```
function Car(make, model, year) {  
    this.make = make;  
    this.model = model;  
    this.year = year;  
}  
  
var mycar = new Car("Eagle", "Talon TSi", 1993);  
var hiscar = new Car("Nissan", "300ZX", 1992);
```

Event-driven programming



- Browsers are interactive: alternate input and output
- GUI events: Windows/Mac/Xwindows
- Features: Icons, menus, dialog boxes, windows, buttons, scrollers, check boxes
- Common feature: User generates *events* via clicks, drags, keystrokes, timeouts
- One kind of interaction in browser: Hyperlinks
- Another type of interaction: Embedding of event-based JavaScript programs in HTML files

Structured programming(1)



- Recent art of computer programming, also knows as **modular programming**
- Essentially a logical programming technique aimed at improving clarity, quality, and development time of a computer program
- Makes use of regular arithmetic operators just like any other programming technique, but the program flow in this case follows a simple hierarchical model that employs looping constructs such as "for," "do", "repeat," "until", and "while"

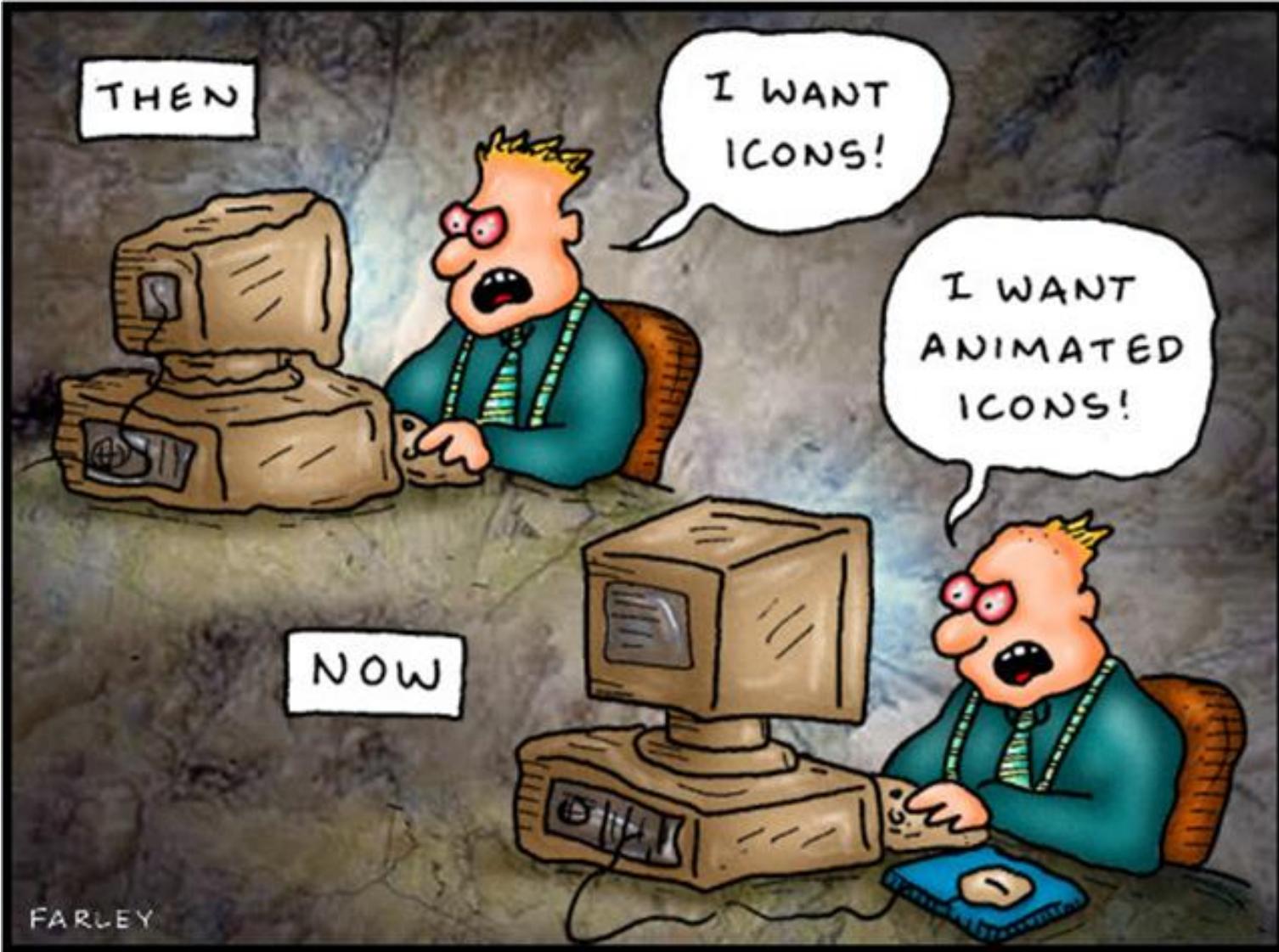
Note:

Not all programming languages feature all of the looping constructs)

Structured programming(2)



- One important goal is to avoid unmanageable and difficult-to-understand code caused by the usage of GOTO statements
- GO TO statements are often found in BASIC and FORTRAN programs
- Structured programming as promoted by high-level languages like C and Pascal discourage use of GO-TO statements, and encourage an overall program structure that reflects what the program is supposed to do, beginning with the first task and proceeding logically in an intuitive step-by-step manner
- A structured high-level program features loops, branch control structures and subroutines rather than GOTO statements; moreover, indentations and comments are used liberally to make the program and its intended logic clear to whoever is writing or reading the program



Why JavaScript?

The trend



- JavaScript is starting to play a central role in the implementation of cloud based mobile solutions
- Node.js, a software platform used to build scalable network applications based on Google's V8 JavaScript engine, is being adopted by millions of developers and enterprises for a wide range of use-cases
- All this is because JavaScript allows rapid development and maximizing user experience ... is able to deliver rich, dynamic web content ... is relatively lightweight has high ease of use

- In September 2012, industry analyst firm RedMonk, showed JavaScript as the top language of the enterprise
- JavaScript is a great language for teaching computer science basics --- programming, data structures, and algorithms

What is JavaScript (JS)?



- Object oriented, structured language
- Runs in a host environment
- Syntax comes from the Java and C
- No input/output commands ... used as a scripting language
- Host environment provides mechanisms for communicating with the outside world
- Most common host environment is the browser
- Code can be inserted into HTML pages and executed by browser
- Code can also be executed in a shell or console

JavaScript's web context



- HTML - used to store the content and formatting of a web page
- CSS - encodes the style of how the formatted content is graphically displayed
- JavaScript - used to add interactivity to a web page or create rich web applications

Embedding JavaScript in HTML



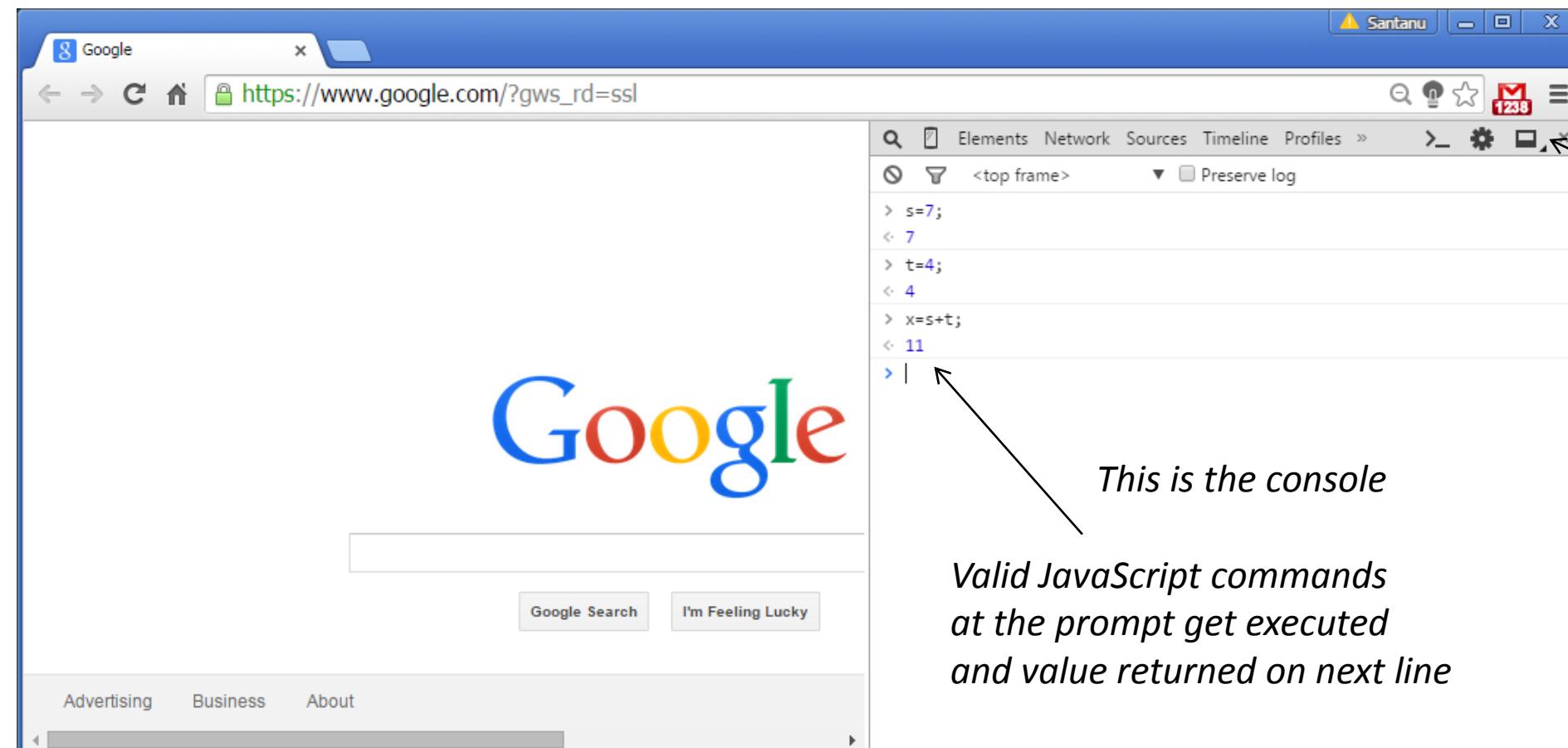
- Insert into an HTML page between `<script>` and `</script>` tags
- Can be put in `<body>` or `<head>` section
- The JavaScript code executes when the HTM page opens in a browser

```
<!DOCTYPE html>
<html>
<body>
.
.
<script>
JavaScript code goes here
(can be a file name with code)
</script>
.
.
</body>
</html>
```

Executing JavaScript in a console



- Control-Shift-J in Windows on Chrome browser brings up the console on the side or below



The first time you may need to long press the undock button for the  icon to show

Clicking thereafter will dock console to or undock console from main browser window

Writing multi-lines



- Use <Shift-Enter> to continue multiline without evaluating
- The final <Enter> will evaluate

```
> t=4;  
< 4  
> s=5  
< 5
```

The screenshot shows the developer tools console with two lines of code. The first line is 't=4;' followed by a cursor. An arrow points from the word 'Enter' to the cursor. The second line is 's=5' followed by a cursor. Another arrow points from the word 'Enter' to the cursor.

```
> t=4;  
< 4  
> s=5;  
< 5  
>
```

The screenshot shows the developer tools console with three lines of code. The first line is 't=4;' followed by a cursor. An arrow points from the word 'Shift-Enter' to the cursor. The second line is 's=5' followed by a cursor. Another arrow points from the word 'Enter' to the cursor. The third line is an empty line starting with '>' followed by a cursor.

<code>new String("foo")</code>	<code>String</code>	<code>object</code>
<code>1.2</code>	<code>Number</code>	<code>number</code>
<code>new Number(1.2)</code>	<code>Number</code>	<code>object</code>
<code>true</code>	<code>Boolean</code>	<code>boolean</code>
<code>new Boolean(true)</code>	<code>Boolean</code>	<code>object</code>
<code>new Date()</code>	<code>Date</code>	<code>object</code>
<code>new Error()</code>	<code>Error</code>	<code>object</code>
<code>[1,2,3]</code>	<code>Array</code>	<code>object</code>
<code>new Array(1, 2, 3)</code>	<code>Array</code>	<code>object</code>
<code>new Function("")</code>	<code>Function</code>	<code>function</code>
<code>/abc/g</code>	<code>RegExp</code>	<code>object (function in Nitro)</code>
<code>new RegExp("menu")</code>	<code>RegExp</code>	<code>object (function in Nitro)</code>

JavaScript programming basics

Variables



- Variables are symbolic names for values
- Names of variables are called **identifiers**
- No spaces or “!” in a name
- No digits in front of a name
- Declared in 2 main ways
 - With keyword *var*
 - Declares a local or global variable
 - Without keyword *var*
 - Declares a global variable

```
Developer Tools - https://www.google.com/
Elements Network Sources > x4 X Gears
<top frame>
> x = 10; // a variable
10
> y32 = 10; // another variable
10
> $4_B$$ = 5; // a third badly named variable
5
> z = x + y32 + $4_B$$ // add variables
25
> x2 // undefined variable
✖ ReferenceError: x2 is not defined
> z = z + 3;
28
> z = z + x3;
✖ ReferenceError: x3 is not defined
> w!2 = 4; // ! not allowed
✖ SyntaxError: Unexpected token !
> y 33 = 5; // space not allowed in var name
✖ SyntaxError: Unexpected number
>
```

Note: A variable declared using the *var* statement with no initial value specified (e.g., *var x;*) has the value *undefined*. An attempt to access an undeclared variable results in a *ReferenceError*

Data types



- Number
- String
- Boolean
- Array
- Object
- Null
- Undefined

Dynamically typed
 $x = 5; x = "America"$

```
Developer Tools - https://www.google.com/
Elements Network Sources Timeline Profiles Resources > X
<top frame>
> x = 5e-6; // NUMBER
0.000005
> student = "John"; // STRING
"John"
> x = true; y = false // BOOLEAN
false
> names=new Array("A", "B", "C"); // ARRAY
["A", "B", "C"]
> person = {name:"John", build:"big", id:3}; // OBJECT
Object {name: "John", build: "big", id: 3}
> x
true
> x = null;
null
> x
null
> y
false
> z
28
>
```

Strings



- Variables can store strings
- String objects are created with **new String()** or by assignment
- A string's properties can be obtained using methods
- Each character in a string has an index starting with 0
- There are many string object methods built into the language
 - `length();` `charAt();` `concat();` `indexOf();` `match();` `split();` `substr();` `search();` `toLowerCase();` `toString();` `trim();` `valueOf();` `replace();` `slice();`

The screenshot shows the Google Chrome Developer Tools console window. The title bar reads "Developer Tools - https://www.google.co...". The console tab is selected, showing the following code and output:

```
> text=new String("foo");
String {0: "f", 1: "o", 2: "o", length: 3}
> myname = "Lady Jane";
"Lady Jane"
> x = myname.length;
9
> y = myname.charAt(2);
"d"
> z = text[0];
"f"
>
```

Arithmetic operators



- = assign values
- Addition
- Subtraction
- Multiplication
- Division
- Remainder
- Increment
- Decrement
- Negation

```
Dev... -> X
> x=5+2 // ADD
7
> x = 4-2 // SUB
2
> x=2*3 // MULT
6
> x=4/2 // DIV
2
> x=5%2 // REM
1
> x++ //INC
1
> x
2
> x-- // DEC
2
> x
1
> -x // NEG
-1
>
```

```
< Pre-inc/dec >
> x=4
4
> ++x
5
> --x
4
>
```

+ can operate on strings!

```
Developer Tools - ...
Elements -> X
> txt1 = "Nice"
"Nice"
> txt2 = "day"
"day"
> text3=txt1+" "+txt2+"!"
"Nice day!"
```

Numbers can be added to strings!

```
Developer Tools - https://www.google.com/
Elements Network Sources Timeline > X
> 5+5
10
> y="5"+5
"55"
> z="Hello"+5
"Hello5"
```

Comparison operators



- `==` : equal to
- `===` : equal in value and type
- `!=` : not equal
- `!==` : different value and type
- `>` : greater than
- `<` : less than
- `>=` : greater than or equal
- `<=` : less than or equal

The screenshot shows a browser developer tools console window titled "Developer Tools - https://www.g...". The console lists the following JavaScript code and its execution results:

```
x=10 // number
10
y="10" // string
"10"
x == y // value check
true
x === y // check value and type
false
x !== y // different value and type
true
z = 50
50
x > y // greater than
false
x < y // less than
false
x < z
true
x < 10
false
x <= 10 // less than or equal to
true
z > 50
ReferenceError: z is not defined
z > 50
false
z >= 50 // greater than or equal to
true
> |
```

Logical operators



- `&&` : and
- `||` : or
- `!` : not

```
Developer Tools - h...
Elements >≡ ⚙️ 📁
✖️ 🔍
> x = 5
5
> y = 2
2
> (x < 9) && (y > 0) // AND
true
> (x == 5) || (y > 5) // OR
true
> !(x == y) // NOT
true
>
```



Bit operations

- Operands are converted to 32-bit integers and expressed in binary by a series of bits (zeros and ones)
- Each bit in the first operand is paired with the corresponding bit in the second operand: first bit to first bit, second bit to second bit, and so on
- The operator is applied to each pair of bits, and the result is constructed bitwise
- Example for bitwise AND ($0.0=0$, $0.1=0$, $1.0=0$, $1.1=1$)

• 9 _(base 10)	= 00000000000000000000000000001001	(base 2)
• 14 _(base 10)	= 00000000000000000000000000001110	(base 2)
• 14 & 9 _(base 10)	= 00000000000000000000000000001000	(base 2)

= 8_(base 10)

AND	IN1	
IN2	0	1
	0	0
	1	0

Truth tables and logical operations



INV	IN	
	0	1
OUT	1	0

AND	IN1	
	0	1
IN2	0	0 0
	1	0 1

OR	IN1	
	0	1
IN2	0	0 1
	1	1 1

XOR	IN1	
	0	1
IN2	0	0 1
	1	1 0

- `~` : not
- `&` : and
- `|` : or
- `^` : xor
- `<<` : left shift
- `>>` : sign propagate right shift
- `>>>` : zero propagate right shift

Note: $\sim x = -(x+1)$

```
Developer Tools - https://mail.go... Elements Network Developer Tools - https://mail.go...
<top frame>
> ~9 // NOT
-10
> 9 & 14 // AND
8
> 9 | 14 // OR
15
> 9 ^ 14 // XOR
7
> 9 << 2 // LSHIFT
36
> 9 >> 2 // SIGN RSHIFT
2
> -9 >> 2 // SIGN RSHIFT
-3
> 9 >>> 2 // ZERO FILL RSHIFT
2
> -9 >>> 2 // ZERO FILL DOES NOT WORK
1073741821
>
```

Conditional control: if-else



- A conditional statement is a set of commands that executes if a specified condition is true.

```
if (condition)
{
    code to be executed if condition is
    true
}
else
{
    code to be executed if condition is
    not true
}
```

The screenshot shows a browser developer tools console window titled "Developer ...". It displays a JavaScript session with the following interactions:

```
> score = 60;
60
> if (score > 50) {
  x = "Good"
}
else {
  x = "Bad"
}

"Good"
> x
"Good"
> score = 40
40
> if (score > 50) {
  x = "Good"
}
else {
  x = "Bad"
}

"Bad"
> x
"Bad"
>
```

Conditional control: switch



- Evaluates an expression, matching the expression's value to a case label, and executes statements associated with that case

```
switch(n)
{
    case 1:
        execute code block 1
        break;
    case 2:
        execute code block 2
        break;
    default:
        code to be executed if n is different
        from case 1 and 2
}
```

```
Developer Tools - http://...
Elements > <top frame> ▾
> day=new Date().getDay();
2
> switch (day) {
  case 6:
    x = "Saturday";
    break;
  case 0:
    x = "Sunday";
    break;
  default:
    x = "Weekday";
}
"Weekday"
> x
"Weekday"
>
```

Loops: for



- A **for** loop repeats until a specified condition evaluates to false

```
for (init counter; test counter; increment counter)
{
  code to be executed;
}
```

Parameters:

init counter: Initialize the loop counter value

test counter: Evaluate for each loop iteration. If it evaluates to TRUE, the loop continues. If it evaluates to FALSE, the loop ends

increment counter: Increases the loop counter value

The screenshot shows the Developer Tools console tab with the title "Developer Tools - http://...". The console output displays the execution of a for loop:

```
> for (var i=0; i<3; i++) {
    console.log(i);
}
0
1
2
< undefined
>
```

The numbers 0, 1, and 2 are colored blue, while the other text is black. The file path "VM3462:3" is shown next to each number.

Loops: do-while



- The **do...while** statement repeats until a specified condition evaluates to false

```
do {  
    code block to be executed  
} while (condition);
```

```
i=1  
1  
do {  
    i+=1;  
    console.log(i);  
} while (i < 5);  
2 VM3530:4  
3 VM3530:4  
4 VM3530:4  
5 VM3530:4  
undefined
```

Loops: while



- A **while** statement executes its statements as long as a specified condition evaluates to true

```
while (condition) {  
    code block to be executed  
}
```

A screenshot of a browser developer tools console window titled "Develop...". The console shows the following interaction:

```
> n=0; x=0;  
0  
> while (n < 3) {  
    n++;  
    x+=n;  
}  
6  
>
```

Break & Continue



- The **break** statement "jumps out" of a loop
- The **continue** statement "jumps over" one iteration in the loop

Breaks out of the loop when i=3

```
x=34;
34
> for (var i=0; i<5; i++) {
    if (i==3)
        break;
    x=x+i;
}
37
> X=34+1+2
```

The developer tools console shows the variable `x` being set to 34, then 34 again, followed by the execution of a `for` loop. Inside the loop, when `i` reaches 3, the `break` statement is executed, causing the loop to exit. The final value of `x` is 37.

Skips iteration loop when i=3

```
x=34;
34
> for (var i=0; i<5; i++) {
    if (i==3)
        continue;
    x=x+i;
}
41
> X=34+1+2+4
```

The developer tools console shows the variable `x` being set to 34, then 34 again, followed by the execution of a `for` loop. Inside the loop, when `i` reaches 3, the `continue` statement is executed, skipping the rest of the loop body and moving to the next iteration. The final value of `x` is 41.

Arrays



- An array is a data structure
- Each cell in an array has an automatic index starting with 0
- **pop()** function deletes the last item and gets its value
- **push()** function adds a last item
- Array of arrays are also possible

The screenshot shows a browser developer tools console window titled "Developer Tools - https://www....". The console displays a series of JavaScript commands and their results, illustrating array manipulation:

```
> numarray = [11, 21, 53, 42, 56]
[11, 21, 53, 42, 56]
> x = numarray[2]
53
> y = numarray.length
5
> numarray.pop()
56
> numarray
[11, 21, 53, 42]
> numarray.push(23)
5
> numarray
[11, 21, 53, 42, 23]
> numarray[0]
11
> numarray[4]
23
> numarray[5]
undefined
>
```

Functions



- Block of code that gets executed when “called”
- Arguments can be passed to a function
- `function foo (var1) {
 code to do something
}`

A screenshot of the Google Chrome Developer Tools. The 'Elements' tab is selected. In the main pane, there is a tree view showing a function definition:

```
> function foo (name) {
    console.log(name);
}
undefined
```

Below the tree, a command line interface shows the function being called with the argument "Jason":

```
> foo ("Jason")
Jason
VM3743:3
<
>
```

```
<!DOCTYPE html>
<html>
<head>
<title>Trying a JavaScript function </title>
<script>
function foo (name)
{
  alert("Hello " + name + "!");
}
</script>
</head>

<h1> This is an example </h1>
<p> Click button to call function with an argument</p>
<body>
<button onclick="foo('Professor')">Click me</button>
</body>
</html>
```

Objects



- Objects are just data, with properties and methods
- Dates, Arrays, Strings and Functions can be objects
- Can create own object , e.g., person, with properties
- Properties are accessed easily
- Methods are actions that objects can perform
 - Syntax: *objectName.methodName()*

String example

```
Developer Tools - https://www.... □ X
Elements Network >≡ ⚙
<top frame>
> message="Hello World!"
"Hello World!"
> x=message.length
12
> y=message.toUpperCase();
"HELLO WORLD!"
> y
"HELLO WORLD!"
```

Use "new"

```
Developer Tools - https://www.... □ X
Elements Network >≡ ⚙
<top frame>
> person=new Object();
> Object {}
> person.name="John";
"John"
> person.age=35;
35
> person.build="medium"
"medium"
> person.name
"John"
>
```

JavaScript object - Style 1



- Define a class using constructor **function()**
- Create object properties with **this** keyword
- Use **methods** to define actions (e.g., *getCarInfo*)
- Create objects with **new** keyword that calls the constructor
- Instantiate object calling **constructor** function
- Set properties and call methods

The screenshot shows the Developer Tools console in Google Chrome. The URL is http://www.phpied.com/3-ways-to-define-a-javascript-object/. The console output demonstrates the creation of a 'Car' constructor function, its properties, and methods, and the instantiation of a car object.

```
> function Car (type) {
  this.type = type;
  this.color = "blue";
  this.getInfo = getCarInfo;
}
undefined
> function getCarInfo() {
  return this.color + ' ' + this.type + ' car';
}
undefined
> car = new Car('Toyota');
▶ Car {type: "Toyota", color: "blue", getInfo: function}
> car.color
"blue"
> car.color = "bluish-grey";
"bluish-grey"
> console.log(car.getInfo());
bluish-greyToyota car
< undefined
>
```

VM4198:2

JavaScript object - Style 2



- Methods can be defined within constructor function
- Helps prevent pollution of global namespace, but method gets recreated every time object created

```
Developer Tools - https://www.google.com/_/chrome/newtab?rlz=1C1... □ X
Elements Network Sources Timeline Profiles »
<top frame>
function Car (type) {
  this.type = type;
  this.color = "blue";
  this.getInfo = function() {
    return this.color + ' ' + this.type + ' car';
  }
}
car = new Car('Toyota');
Car {type: "Toyota", color: "blue", getInfo: function}
car.color
"blue"
car.color="red";
"red"
console.log(car.getInfo());
red Toyota car
VM545:2
```

JavaScript object - Style 3



- Prevent recreation of method *getInfo()* every time new object created
- Add to *prototype* of constructor function

```
Developer Tools - https://www.google.com/_/chrome/newtab?rlz... Elements Network Sources Timeline Profiles >
<top frame>
> function Car (type) {
  this.type = type;
  this.color = "blue";
}
undefined
> Car.prototype.getInfo = function() {
  return this.color + ' ' + this.type + ' car';
}
function () {
  return this.color + ' ' + this.type + ' car';
}
> car = new Car('Toyota');
▶ Car {type: "Toyota", color: "blue", getInfo: function}
> car.color
"blue"
> car.color="green";
"green"
> console.log(car.getInfo());
green Toyota car
VM1005:2
< undefined
>
```

JavaScript object - Style 4



- Object *literals* are shorter way to define objects and arrays
- No need to (and cannot) create instance, it already exists
- Start using

```
Developer Tools - https://www.google.com/_/chrome/newt... X
Elements Network Sources Timeline Profiles >≡ ⚙️ 📁
🔍 
> var car = {
    type: "Toyota",
    color: "blue",
    getInfo: function() {
        return this.color + ' ' + this.type + ' car';
    }
}
undefined
> car.color
"blue"
> car.color='black';
"black"
> console.log(car.getInfo());
black Toyota car
< undefined
>
```

JavaScript object - Style 5



- Use singleton for a constructor function with no name that will be used few times
- `new function(){...}` defines an anonymous constructor function and invokes it with `new`

The screenshot shows the Developer Tools console in Google Chrome. The code demonstrates the creation of a singleton object:

```
Developer Tools - https://www.google.com/_/chrome/newt...
Elements Network Sources Timeline Profiles > X
<top frame>
> var car = new function() {
    this.type = "Toyota";
    this.color = "blue";
    this.getInfo = function() {
        return this.color + ' ' + this.type + ' car';
    }
}
undefined
> car.color
"blue"
> car.color = 'white';
"white"
> console.log(car.getInfo());
white Toyota car
VM1237:2
< undefined
> |
```

The code creates a new function object, sets its properties `this.type` to "Toyota" and `this.color` to "blue", and defines a method `this.getInfo` that returns a string combining the color and type. When `car.color` is checked, it returns "blue". Then, `car.color` is changed to "white", and `car.getInfo()` is called, outputting "white Toyota car". The VM identifier `VM1237:2` is shown at the bottom right.

The way we will use multiple methods



Developer Tools - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects

```
> function Car (name, color) { ← Constructor
  this.name=name;
  this.color=color;
}
< undefined
> Car.prototype = { ← Prototype

  add: function(name, color) {
    this.name=name;
    this.color=color;
  },
  getcol: function() { ← Methods
    return this.color;
  },
  getname: function() {
    return this.name;
  },
};

< ► Object {}
```

Developer Tools - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects

```
> var mycar = new Car;
< undefined
> mycar.add("Toyota", "Blue");
< undefined
> mycar.getcol();
< "Blue"
> mycar.getname();
< "Toyota"
>
```

Functions with no arguments



- Functions do not have to have arguments
- For such a function, even if arguments are passed, it does the same thing

```
Developer Tools - http://eloquentjavascript.n... Elements <top frame> Pres...  
> function example() {  
    console.log("Going is great!")  
}  
< undefined  
> example()  
Going is great! VM84:3  
< undefined  
> example(1)  
Going is great! VM84:3  
< undefined  
> example(1, 2, 3)  
Going is great! VM84:3  
< undefined  
> |
```

```
Developer Tools - http://eloquentjavascript.n... Elements Network Sources > Pres...  
< top frame> Preserve log  
> function example(arg) {  
    console.log("May or may not have args");  
    console.log(arg);  
}  
< undefined  
> example(10)  
May or may not have args VM427:3  
10 VM427:4  
< undefined  
> example(1, 2, 3)  
May or may not have args VM427:3  
1 VM427:4  
< undefined  
> example()  
May or may not have args VM427:3  
undefined VM427:4  
< undefined  
> |
```

Accessing arguments of a function (1)



- A special pseudo-array inside each function called “arguments”
- Contains all parameters by their number: `arguments[0]`, `arguments[1]`, ... etc.

```
Developer Tools - http://eloquentjavascript.net/06_objects.html
Elements Network > Preset log
< top frame >
function foo() {
  for(var i=0; i<arguments.length; i++) {
    console.log("Hi, " + arguments[i])
  }
}

< undefined
> foo
< function foo()
> foo()
< undefined
> foo(1)
Hi, 1
VM562:4
< undefined
> foo(1, 2)
Hi, 1
Hi, 2
VM562:4
< undefined
>
```

Accessing arguments of a function (2)



- Better programming technique is to test whether there are arguments passed before trying to access them

```
Developer Tools - https://www.google.com/?gws_rd=ssl
Elements Network Sources > Preset log
<top frame> Preserve log

> function foo() {
  if (arguments.length == 0) {
    console.log("No arguments passed")
  }
  else {
    for(var i=0; i<arguments.length; i++) {
      console.log ("Hi, " + arguments[i])
    }
  }
< undefined
> foo()
No arguments passed VM115:4
< undefined
> foo (1)
Hi, 1 VM115:8
< undefined
> foo (1, 2)
Hi, 1 VM115:8
Hi, 2 VM115:8
< undefined
>
```

Functions may or may not return a value



- `foo()` does not return a value

```
Developer Tools... <top frame>
function foo () {
  2;
  x=5;
}
undefined
foo()
undefined
y=foo()
undefined
y
undefined
> |
```

A screenshot of the Google Chrome Developer Tools console. It shows the definition of a function `foo` that contains two statements: `2;` and `x=5;`. When `foo()` is called, it returns `undefined`. Then, when `y=foo()` is executed, it also returns `undefined`, indicating that `foo()` did not return a value.

Good habit to
not omit the ","
at the end of
the statements

```
Developer Tools - h...
<top frame>
function goo () {
  x=5*2;
  return x;
}
undefined
goo()
10
y=goo();
10
y
10
>
```

A screenshot of the Google Chrome Developer Tools console. It shows the definition of a function `goo` that contains two statements: `x=5*2;` and `return x;`. When `goo()` is called, it returns the value `10`. Then, when `y=goo();` is executed, it also returns `10`, indicating that `goo()` returned a value.

- `goo()` returns a value

Objective look at functions



- JavaScript functions are *function objects* created with the *Function constructor*
- A Function object contains a string which contains the Javascript code of the function!
- Can pass code to another function in the same way you would pass a regular variable or object

The screenshot shows a browser's developer tools console window. The title bar reads "Developer Tools - http://recurial.com/programming/understanding-callback-functions-in-javascript...". The console tab is selected. The output area shows the following code execution:

```
<top frame>
> var func_multiply = new Function("arg1", "arg2", "return arg1 * arg2;");
undefined
> func_multiply(5,10);
50
>
```

Many ways of declaring *functions*



- `function A(){}` // function declaration
- `var B = function(){}` // function expression
- `var C = (function(){}
)` // function expression with grouping operators
- `var D = function foo(){}` // named function expression
- `var E = (function(){
 return function(){}
})()` // (IIFE) that returns a function
- `var F = new Function(){}` // Function constructor
- `var G = new function(){}` // special case: object constructor

Type name conventions



Standard function statement:

```
function getarea(w,h){  
    var area=w*h  
    return area  
}
```

Conditional function statement:

```
if (calculateit == true){  
    function getarea(w,h){ //standard function  
        var area=w*h  
        return area  
    }  
}
```

Function Literal (an anonymous function assigned to a variable):

```
var getarea=function(w,h) {  
    var area=w*h  
    return area  
}
```

Function Constructor ... creates a function on the fly, which is slower and generally discouraged:

```
var getarea=new Function("w", "h", "var area=w*h; return area")
```

Function subtleties



```
function myFirstFunc(param) {  
    //Do something  
};
```

Defines the function as soon as the enclosing scope is entered

```
var myFirstFunc = function(param) {  
    //Do something  
};
```

Only creates the function once execution reaches that line

Iterative programs



- Repeats a process to compute a result
- Applies a function repeatedly using the output from one iteration as the input to the next
- $5!=5\times4\times3\times2\times1=120$ can be calculated as:
 - Initialize variable result to 1 and iterate 5 times
 - $i=5$: $\text{result} = \text{result} * i = 1 \times 5 = 5$
 - $i=4$: $\text{result} = \text{result} * i = 5 \times 4 = 20$
 - $i=3$: $\text{result} = \text{result} * i = 20 \times 3 = 60$
 - $i=2$: $\text{result} = \text{result} * i = 60 \times 2 = 120$
 - $i=1$: $\text{result} = \text{result} * i = 120 \times 1 = 120$

The screenshot shows the Google Chrome Developer Tools Console window. The title bar says "Developer Tools - http...". The toolbar includes icons for search, refresh, and settings. The console tab is selected, showing the following interaction:

```
> function fact_iter (n) {  
    result=1;  
    for (var i=n; i>0; i--){  
        result=result*i;  
    }  
    return(result);  
}  
< undefined  
> fact_iter (5)  
< 120  
>
```

The storage is the same at every step

Recursive programs

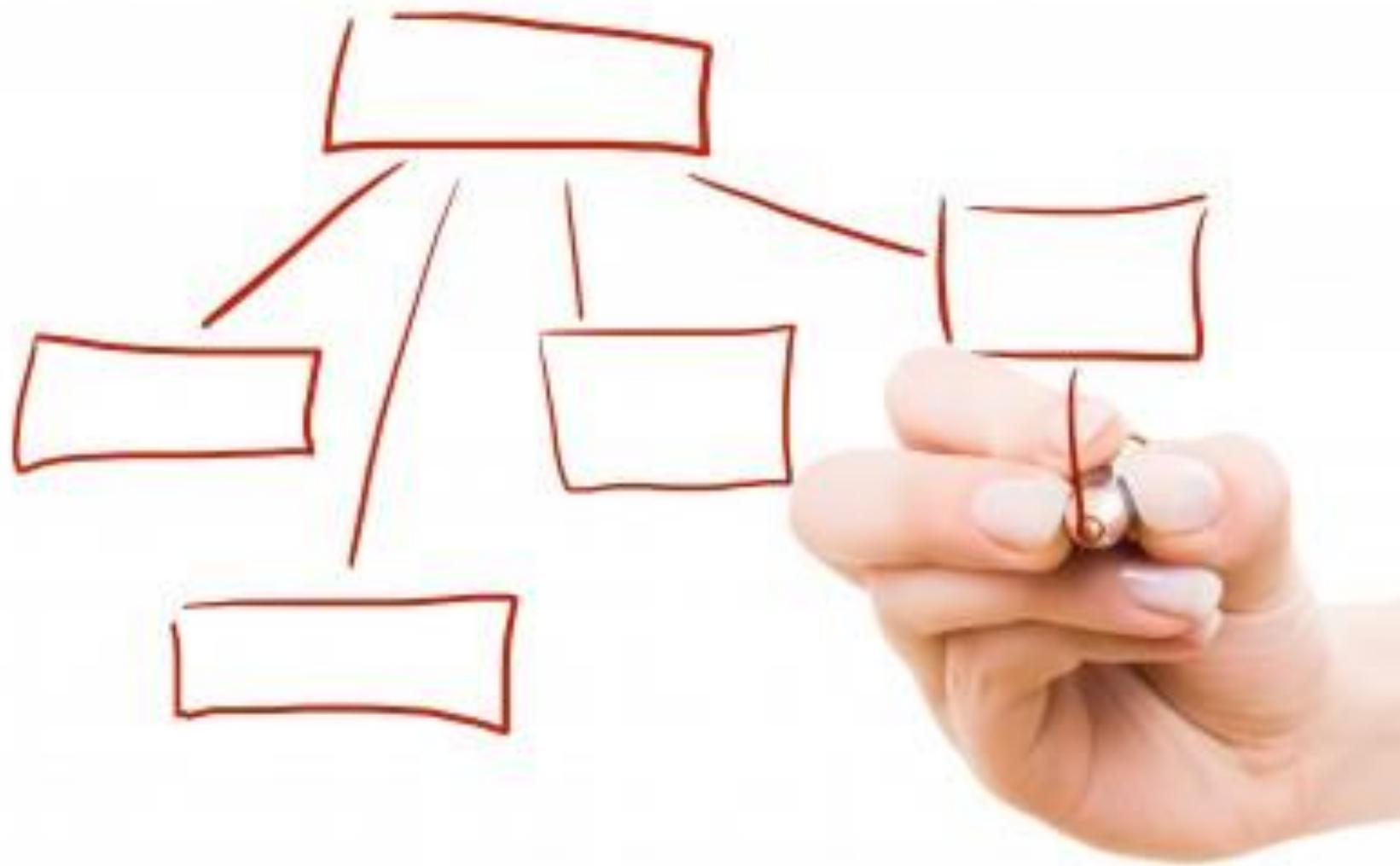


- Defines a function in terms of itself where final the solution is obtained from solution of a smaller instance of the same problem
- GNU=GNU is Not Unix is a recursion in a name
- $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$ can be calculated as:
 - $5! = 5 \times 4 \times 3 \times 2 \times 1 = 5 \times 4!$
 - $4! = 4 \times 3 \times 2 \times 1 = 4 \times 3!$
 - $3! = 3 \times 2 \times 1 = 3 \times 2!$
 - $2! = 2 \times 1 = 2 \times 1!$
 - $1! = 1$ (terminating condition)
 - Now, backtrack to calculate result of every step

The screenshot shows a browser's developer tools console window titled "Developer Tools - https...". The console tab is active, showing the following interaction:

```
> function fact_recur(n){  
  if (n==1)  
    return(1);  
  else  
    return(n*fact_recur(n-1))  
}  
< undefined  
> fact_recur(5)  
< 120  
>
```

The storage grows at every step till terminating condition is reached --- possible stack overflow!



Basic data structures

What are data structures?

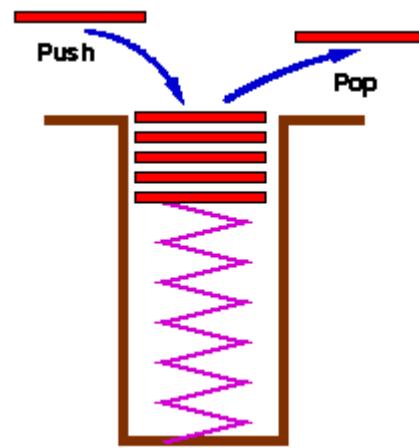


- Particular way of storing and organizing data in so that it can be used efficiently ... key to designing efficient algorithms
- Different kinds (*e.g.* arrays, lists, *etc.*) are suited to different kinds of applications
- Implementation usually requires writing a set of procedures that create and manipulate instances of that structure
- A programming language inherently supports some data structures using which desired ones are implemented
- Most basic operations: search, insert, delete, enumerate
- JavaScript features inbuilt support for **Array**, **String**, and **Object** that can be used to implement various useful data structures such as stacks, linked lists, *etc.*

Stack data structure



- A collection of elements in which the operations allowed on the collection are the addition of an element, known as *push*, and removal of an element, known as *pop*
- The relation between the push and pop operations is such that the stack is a Last-In-First-Out (LIFO) structure --- the last element added to the structure must be the first one to be removed
- A stack can be implemented using an array or a linked list
- A common model of a stack is an element stacker, where elements are "pushed" onto to the top and "popped" off the top



Stack implementation using array



Developer Tools - https://www.google.com/webhp?sourceid=chrom... □ X

Elements Network Sources Timeline Console ▶

✖ <top frame> ▼ Preserve log

```
> function Stack () {
    this.stackarray = new Array();
}
< undefined
> Stack.prototype = {
    pop: function() {
        uval = this.stackarray.pop();
        if (uval == undefined)
            console.log("Trying to pop an empty stack");
        return uval;
    },
    push: function(item) {
        this.stackarray.push(item);
    },
    empty: function() {
        value = (this.stackarray.length == 0);
        return(value);
    },
}
< ► Object {}
```

Developer Tools - http://www.i-program... □ X

Elements Network Sources ▶

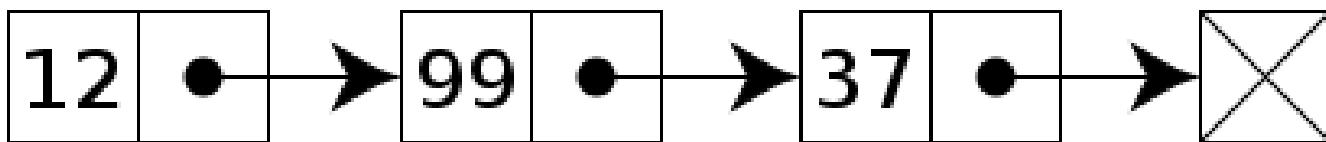
✖ <top frame> ▼

```
> var stack=new Stack();
undefined
> stack.push("A");
stack.push("B");
stack.push("C");
undefined
> stack.pop();
"C"
> stack.pop();
"B"
> stack.pop();
"A"
> stack.pop();
Trying to pop an empty stack VM10481:7
< undefined
> |
```

Linked list data structure



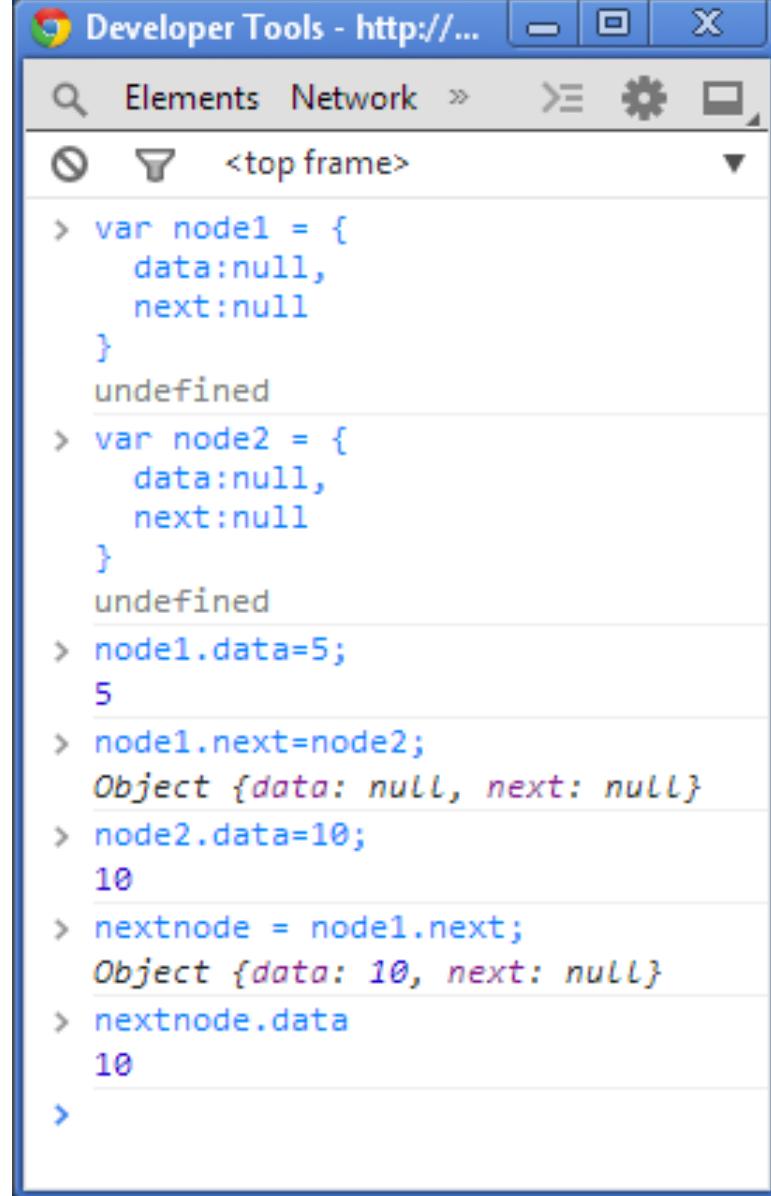
- A data structure consisting of a group of nodes which together represent a sequence
- In a simple form, each node comprises a *value* and a *link* to the next node in the sequence ... more complex variants add additional links.
- This structure allows for efficient insertion or removal of elements from any position in the sequence (as opposed to an array)



Node



- A node is a data structure that can be linked to another data structure
- Has two fields: *data* and *next*
 - Data stores the data held in the node and next is a reference to the next node in the list
- JavaScript is dynamically typed ...
data can store numbers, strings, objects, even another list
- This ability for a list to have another list stored as one of its nodes allows building more complex structures such as trees and acyclic graphs



```
Developer Tools - http://...
Elements Network > X
<top frame>
> var node1 = {
  data:null,
  next:null
}
undefined
> var node2 = {
  data:null,
  next:null
}
undefined
> node1.data=5;
5
> node1.next=node2;
Object {data: null, next: null}
> node2.data=10;
10
> nextnode = node1.next;
Object {data: 10, next: null}
> nextnode.data
10
>
```

Linked list implementation (1)



- List is essentially an object with a “head” node
- The head is a pointer to the first element or node of the list
- Use constructor function to create list object with private properties “length” and “head” ... Length is not necessary, just done here as an example

A screenshot of the Google Chrome Developer Tools. The title bar says "Developer Tools - https://d...". The main pane shows the following JavaScript code:

```
> function MyList () {
  this.length=0;
  this.head=null;
}
```

Linked list implementation (2)



- Prototype method to add or insert nodes to the list
 - Special case is when list is empty
 - Traverse list to add new node at the end of the list

```
Developer Tools - https://developer.mozilla.org/en-US/docs/Web/JavaScript/... □ X
Elements Network Sources Timeline > _ ⚙ □
<top frame> ▾ Preserve log
MyList.prototype = {
  addnode: function (value){
    //create a new node, place data in
    var node = {
      data: value,
      next: null
    },
    //variable to traverse the structure
    current;
    //special case: no items in the list yet
    if (this.head === null){
      this.head = node;
    } else {
      current = this.head;

      while(current.next){
        current = current.next;
      }

      current.next = node;
    }
    // update count
    this.length++;
  },
};

< ► Object {}
```

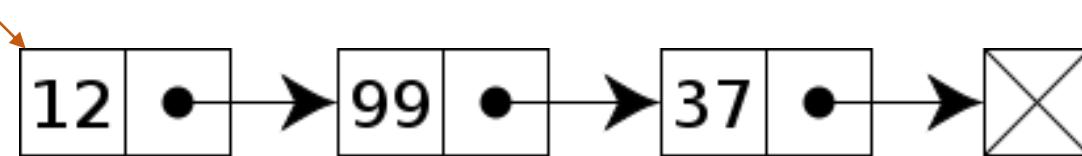
Linked list implementation (3)



- Create a list using the constructor defined earlier
- Use method to sequentially add nodes with values specified in the function argument

Example:

head



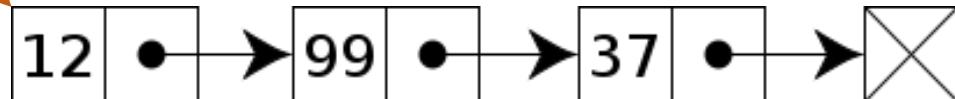
```
Developer Tools - https://www.google.com/?gws_rd=ssl
Elements Network > Preset log
<top frame>
function add_in_list () {
  list=new MyList();
  arglength=arguments.length;
  if (arglength == 0)
    console.log ("No arguments given");
  else
    for(var i=0; i<arglength; i++) {
      list.addnode(arguments[i]);
    }
  return(list);
}
< undefined
nlist=add_in_list(12, 99, 37);
< ► MyList {length: 3, head: Object}
nlist.head
< ► Object {data: 12, next: Object}
nlist.head.data
< 12
nlist.head.next.data
< 99
nlist.head.next.next.data
< 37
>
```

Traversing a list (1)



- Traverse the list from head and print all data values
- Allows you to search for a certain value

head



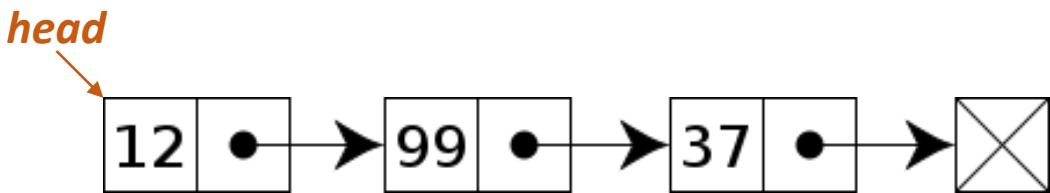
```
Developer Tools - https://www.google.com/?gws_rd=ssl
Elements Network >
< top frame > Preserve log
function traverse_list (list) {
  arglength=arguments.length;
  if (arglength == 0)
    console.log ("No list given");
  else {
    listlen=list.length;
    head=list.head;
    for(var i=0; i<listlen; i++) {
      console.log(head.data);
      head=head.next;
    }
  }
}
< undefined
> traverse_list (nlist)
12
99
37
< undefined
> |
```

The developer tools screenshot shows the execution of a JavaScript function named `traverse_list`. The function takes a list as an argument and iterates through its nodes, printing the data value of each node to the console. The list has three nodes with data values 12, 99, and 37. The variable `head` is set to the first node. The output in the console shows the values 12, 99, and 37, each followed by the file path `VM1733:10`.

Traversing a list (2)



- More efficient traversal based on null value of next pointer instead of using length of list



```
Developer Tools - https://www.google.com/?gws_rd=ssl
```

```
< top frame>
```

```
<function traverse_list (list) {  
    arglength=arguments.length;  
    if (arglength == 0)  
        console.log ("No list given");  
    else {  
        head=list.head;  
        while(head) {  
            console.log(head.data);  
            head=head.next;  
        }  
    }  
}< undefined  
> traverse_list (nlist)  
12 VM1751:9  
99 VM1751:9  
37 VM1751:9  
< undefined  
>
```

The screenshot shows the Google Chrome Developer Tools console. The code in the editor pane is:

```
> function traverse_list (list) {  
    arglength=arguments.length;  
    if (arglength == 0)  
        console.log ("No list given");  
    else {  
        head=list.head;  
        while(head) {  
            console.log(head.data);  
            head=head.next;  
        }  
    }  
}< undefined
```

The output pane shows the results of running the function:

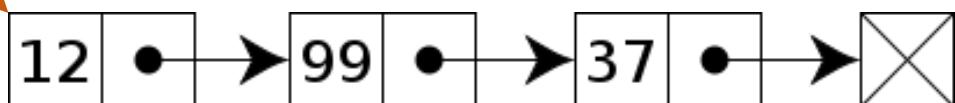
```
> traverse_list (nlist)  
12 VM1751:9  
99 VM1751:9  
37 VM1751:9  
< undefined  
>
```

Traversing a list (3)



- Could have written a method to traverse list as well
- Will keep to functions mostly

head

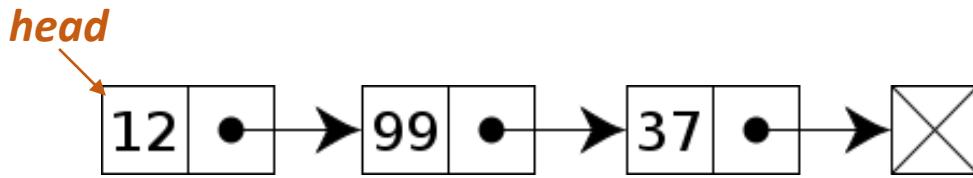


```
Developer Tools - https://mail.google.com/elements</top frame> Presen> MyList.prototype = {  
    // other methods  
  
    printlist: function (){  
        listlen=this.length;  
        head=this.head;  
        for(var i=0; i<listlen; i++) {  
            console.log(head.data);  
            head=head.next;  
        }  
    },  
};< Object {}>  
> mlist.printlist()  
12 VM2184:31  
99 VM2184:31  
37 VM2184:31  
< undefined  
>
```

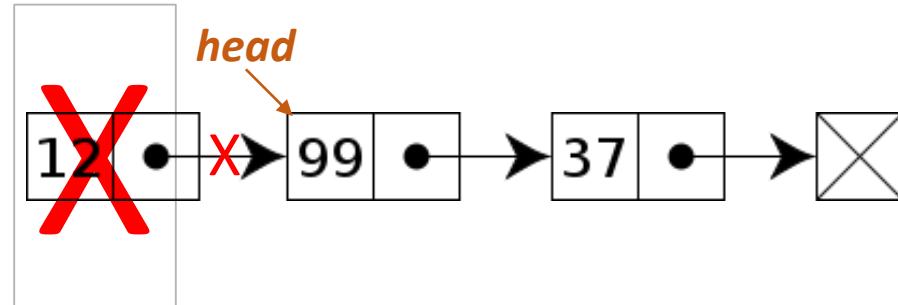
Removing an item from a list - idea



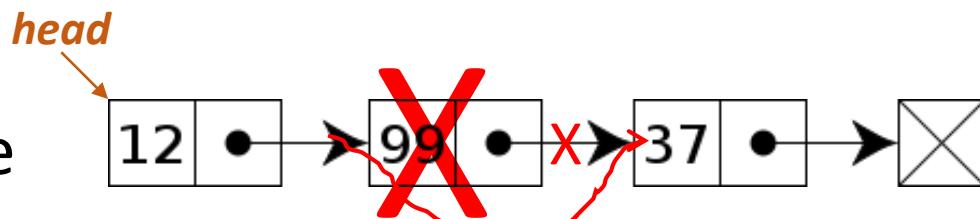
Given head pointer to a list



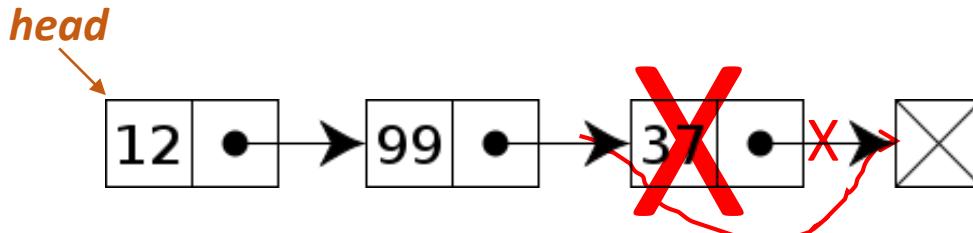
Remove 12: Delete head node



Remove 99: Delete intermediate node



Remove 37: Delete end node



Removing an item from a list - program



Developer Tools - https://mail.google.com/mail/u/0/?tab=wm

Elements Network Sources Timeline Console ▶ 🔍 ⚙️ X

<top frame> ▾ Preserve log

```
> function delete_node_from_list (list, nodeval) {
    // Destructively modifies passed list object
    arglength=arguments.length;
    if (arglength < 2)
        console.log ("Insufficient arguments");
    else {
        deleted=0;
        current=list.head;
        if (current.data == nodeval) { // if delete first item
            list.head=current.next;
            deleted=1;
        }
        else
            while(current) {
                prev=current;
                current=current.next;
                if (current && (current.data == nodeval)) {
                    prev.next=current.next;
                    deleted=1;
                    break;
                }
            }
        if (deleted==0)
            console.log ("Value not in nodelist");
        else
            list.length--; // fix length of list on delete
    }
}
< undefined
```

Developer Tools - https://mail.goo...

Elements Console ▶ 🔍 ⚙️ X

<top frame> ▾ Preserve log

```
> nlist=add_in_list(12, 99, 37);
< ►MyList {length: 3, head: Object}
> delete_node_from_list (nlist, 42)
    Value not in nodelist VM10159:26
< undefined
> delete_node_from_list (nlist, 99)
< undefined
> traverse_list (nlist);
    12 VM9336:10
    37 VM9336:10
< undefined
> delete_node_from_list (nlist, 12)
< undefined
> traverse_list (nlist);
    37 VM9336:10
< undefined
> |
```

Complexity of list operations



- For insert, delete and search, the time complexity is $O(n)$, where n is the number of elements in the list
 - There is no random access
 - You start searching from the head and traverse till you reach the "right" point
 - May have to go to the end of the list after traversing n elements



Hash table data structure (1)

- Also known as a hash map, it is used to implement an associative array, a structure that can map keys to values
- Uses a hash function to compute an index into an array of buckets or slots, from which the correct value can be found
- A practical example is to have shirts hung on rails keyed by color so that one can decide on a general color, say blue, and look at only the blue rail to decide on a shade, say sky blue, to wear instead of looking at a huge collection of shirts to choose from



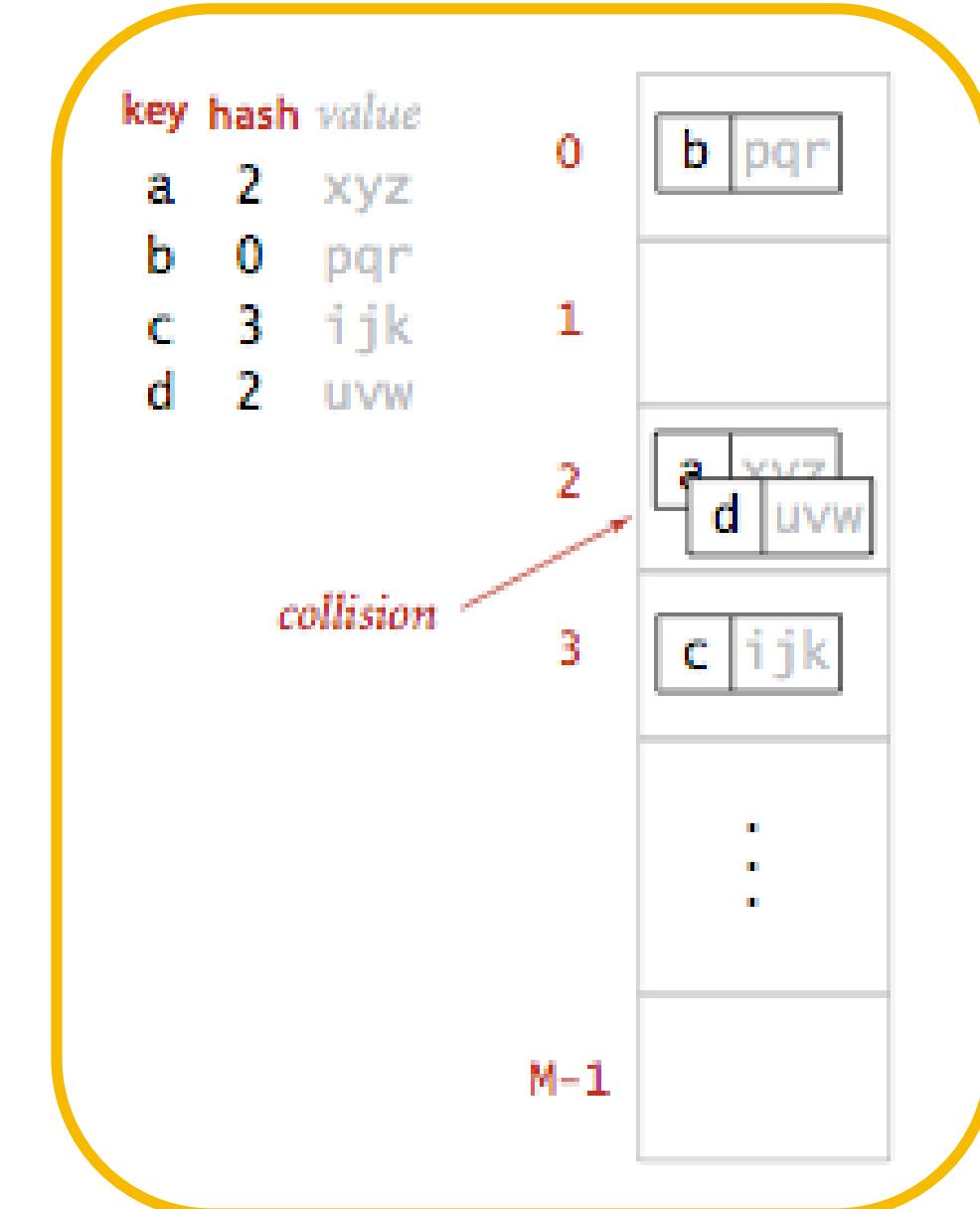
Hash table data structure (2)

- The idea of hashing is to distribute the entries (key/value pairs) across an array of buckets. Given a key, the hashing algorithm computes an index that suggests where the entry can be found
- If keys are small integers, we can use an array to implement a symbol table, by interpreting the key as an array index so that we can store the value associated with key i in array position i
- But, we still run the risk of “collisions” – where two or more items have the same hash value

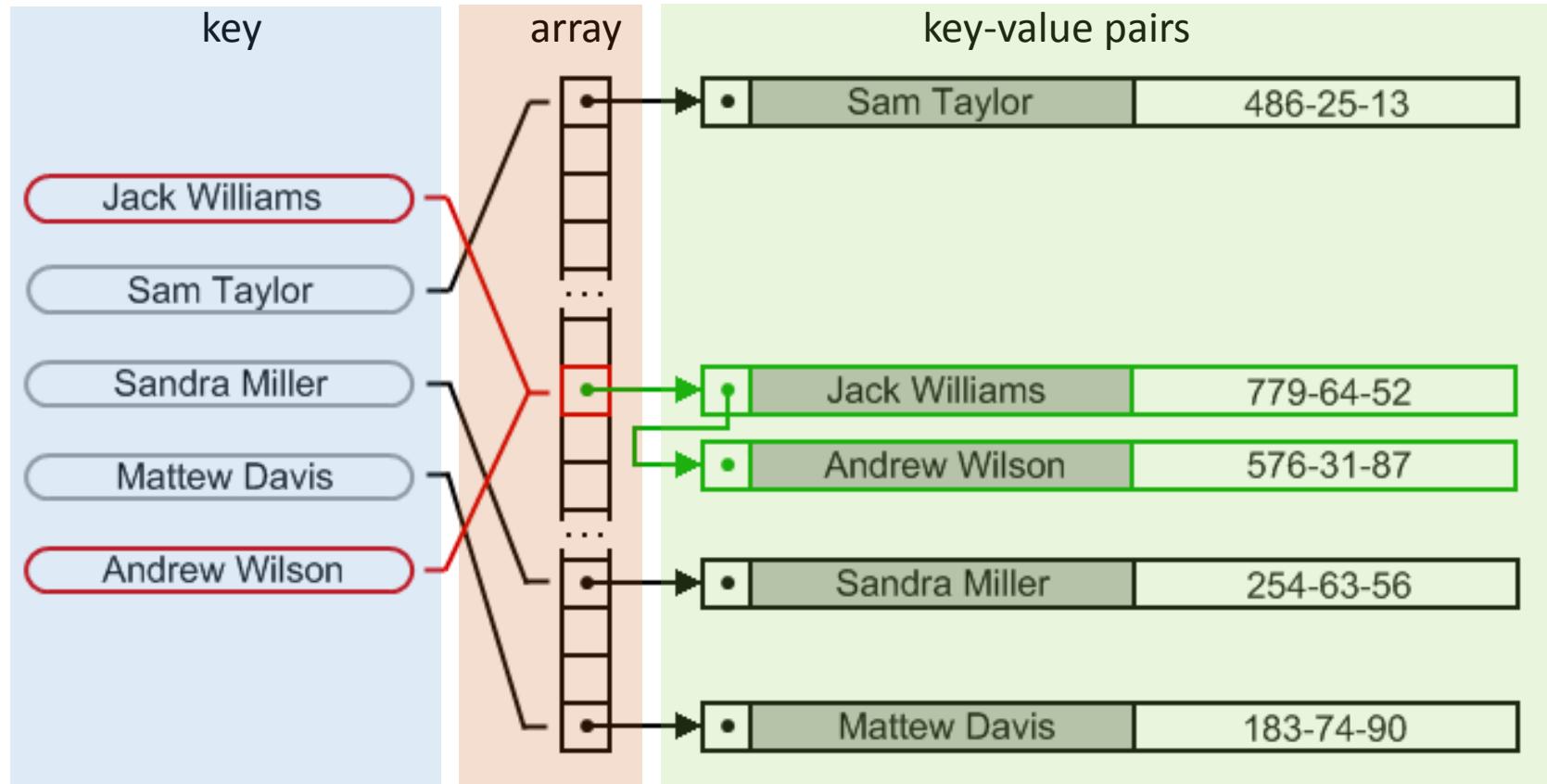
Hashing collision



- Search using hashing consists of two separate parts
 - Compute a hash function that transforms the search key into an array index. Ideally, different keys would map to different indices. But, practically, two or more different keys may hash to the same array index
 - The second part is a collision-resolution process, e.g., chaining



Chaining



Chaining helps resolve collisions. Each slot of the array contains a link to a singly-linked list containing key-value pairs with the same hash. New key-value pairs are added to the end of the list. Lookup algorithm searches through the list to find matching key. Chaining reduces the search space to fewer items in the linked list.

Associative arrays (1)



- Every object in Javascript is an associative array

```
Developer Tools - https://developer.mozilla.org/en-US/d...
Elements Network >_ < Preset log
<top frame> Preserve log
> var myArray=new Object();
myArray[ "A" ]=0;
myArray[ "B" ]=1;
myArray[ "C" ]=2;
< 2
> console.log(myArray[ "B" ]);
1 VM560:2
< undefined
> console.log(myArray.B);
1 VM561:2
< undefined
```

```
Developer Tools - https://developer.mozilla.org/en-US/d...
Elements Network >_ < Preset log
<top frame> Preserve log
> var MyArray={A:0,B:1,C:2};
< undefined
> console.log(myArray[ "B" ]);
1 VM563:2
< undefined
> console.log(myArray.B);
1 VM564:2
< undefined
>
```

- Can be described in many ways

Associative arrays (2)



- Can enumerate keys of an associative array easily

Developer Tools - https://www.google.com/webhp?sourcei...

Elements Network Sources Timeline Console ▶ ⚙️ ✎

<top frame> ▾ Preserve log

```
> var myObject = {};
< undefined
> myObject.city = "San Francisco";
myObject.state = "California";
myObject.country = "United States";
< "United States"
> for(property in myObject) {
    console.log(property + " = " + myObject[property]);
}
city = San Francisco
VM966:3
state = California
VM966:3
country = United States
VM966:3
< undefined
> |
```

Developer Tools - http://www.pr...

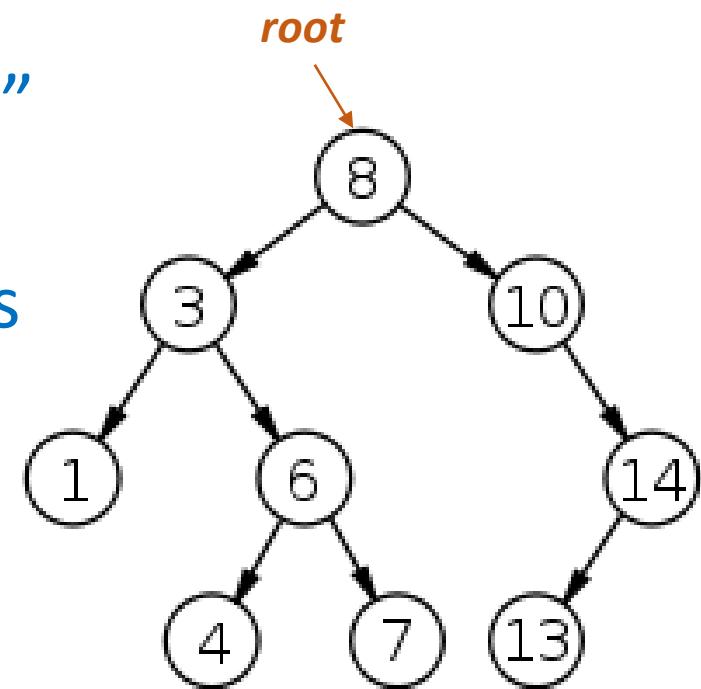
Elements Console ▶ ⚙️ ✎

<top frame> ▾ Pre

```
> var myArray=new Object();
< undefined
> myArray.city = "San Francisco";
myArray.state = "California";
myArray.country = "United States";
< "United States"
> Object.keys(myArray);
< ["city", "state", "country"]
> Object.keys(myArray).length
< 3
> |
```

Binary search tree (BST) data structure

- Particular type of container that store numbers, names, etc. in memory and allows fast lookup because they remain sorted
- Has a root node at the top
- Every internal node stores a value and has two sub-trees (children), commonly denoted “left” and “right”
- Value in each node must be greater than all values stored in the left sub-tree, and smaller than all values in the right sub-tree
- A node can have one child or both children as “null”
- Leaves are nodes that do not have any children



BST implementation (1)



- BST is essentially an object with a “root” node
- Use constructor function to create tree object with private property “root”

A screenshot of a browser's developer tools console, specifically the Chrome DevTools. The title bar says "Developer ...". The console interface includes a toolbar with icons for search, refresh, and settings. Below the toolbar, there are two tabs: one with a magnifying glass icon and another with a funnel icon, both set to "<top frame>". The main area of the console shows the following code:

```
> function MyTree() {
    this.root=null;
}
< undefined
> |
```

BST implementation (2)



- Instantiate tree, create node with value, insert in tree

```
Developer Tools - h...
Console >_ < top frame >
> function makenode (value) {
    var node = {
        data: value,
        left: null,
        right: null
    };
    return (node);
}
< undefined
>
```

```
Developer Tools - http://www.w3schools.com/js/js_object_prototyp... Elements Network Console > _ Preset log  
No Cache <top frame>  
> MyTree.prototype.add2tree = function (value) {  
    var node = makenode(value)  
    if (this.root == null)// initial case  
        this.root = node;  
    else {  
        current = this.root;  
        while (true) {  
            if (value <= current.data) {  
                if (current.left == null) {  
                    current.left = node;  
                    break;  
                }  
                else  
                    current = current.left;  
            }  
            else {  
                if (value > current.data)  
                    if (current.right == null) {  
                        current.right = node;  
                        break;  
                    }  
                else  
                    current = current.right;  
            }  
        }  
    }  
}  
< function MyTree.add2tree(value)  
>
```

```
Developer Tools - https://www.google.com/webhp?sourceid=chrome-instant&q=&sa=X&hl=en
```

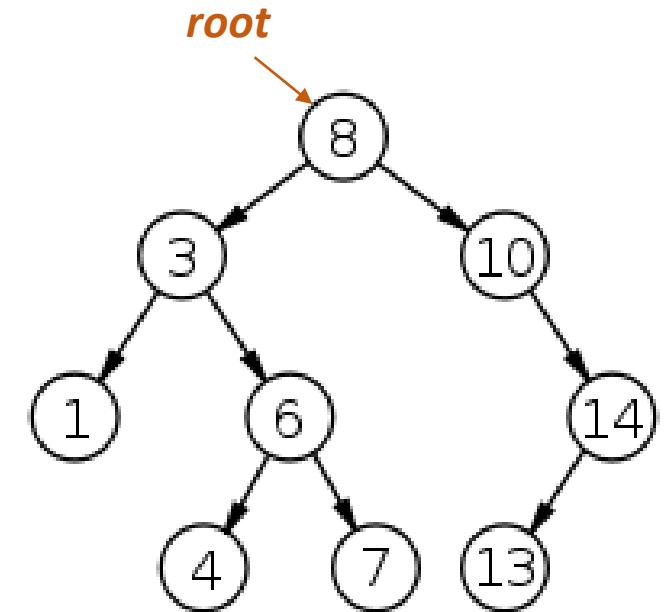
Elements Network Console > Preset log

```
> function BuildMyTree (valuelist){  
    // Build tree with values in arg list  
    tree=new MyTree(); //instantiate  
    for (i=0; i<valuelist.length; i++){  
        value=valuelist[i];  
        tree.add2tree(value);  
    }  
    return(tree);  
}  
< undefined  
> datalist=[8, 3, 1, 6, 4, 7, 10, 14, 13];  
NewTree=BuildMyTree(datalist);  
< ► MyTree {root: Object}  
> |
```

Traversing a BST



- Can traverse BST different ways starting at the root
 - Depth first
 - Pre-order (root, left, right): 8, 3, 1, 6, 4, 7, 10, 14, 13
 - In-order or symmetric (left, root, right): 1, 3, 4, 6, 7, 8, 10, 13, 14
 - Post-order (left, right, root): 1, 4, 7, 6, 3, 13, 14, 10, 8
 - Breadth first or level order: 8, 3, 10, 1, 6, 14, 4, 7, 13
- Recursive routines come to the rescue
 - For breadth first, will have to first collect nodes at the same level together (using hash map)
- Three main steps at any point: action on the current node, traversing to the left child node, and traversing to the right child node



Depth First Search (DFS)



```
Developer Tools - https://mail.goog... Elements Console <top frame> Preser
< top frame>
function TreePrintDFS_preorder (root){
    // print root, left tree, right tree
    console.log(root.data);
    if (root.left) {
        TreePrintDFS_preorder (root.left);
    }
    if (root.right) {
        TreePrintDFS_preorder (root.right);
    }
}
< undefined
TreePrintDFS_preorder(NewTree.root)
8 VM15218:4
3 VM15218:4
1 VM15218:4
6 VM15218:4
4 VM15218:4
7 VM15218:4
10 VM15218:4
14 VM15218:4
13 VM15218:4
< undefined
>
```

```
Developer Tools - https://mail.goog... Elements Console <top frame> Preser
< top frame>
function TreePrintDFS_inorder (root){
    // print left tree, root, right tree
    if (root.left) {
        TreePrintDFS_inorder (root.left);
    }
    console.log(root.data);
    if (root.right) {
        TreePrintDFS_inorder (root.right);
    }
}
< undefined
TreePrintDFS_inorder(NewTree.root)
1 VM15284:7
3 VM15284:7
4 VM15284:7
6 VM15284:7
7 VM15284:7
8 VM15284:7
10 VM15284:7
13 VM15284:7
14 VM15284:7
< undefined
> |
```

```
Developer Tools - https://mail.goog... Elements Console <top frame> Preser
< top frame>
function TreePrintDFS_postorder (root){
    // print left tree, right tree, root
    if (root.left) {
        TreePrintDFS_postorder (root.left);
    }
    if (root.right) {
        TreePrintDFS_postorder (root.right);
    }
    console.log(root.data);
}
< undefined
TreePrintDFS_postorder(NewTree.root)
1 VM15422:10
4 VM15422:10
7 VM15422:10
6 VM15422:10
3 VM15422:10
13 VM15422:10
14 VM15422:10
10 VM15422:10
8 VM15422:10
< undefined
>
```

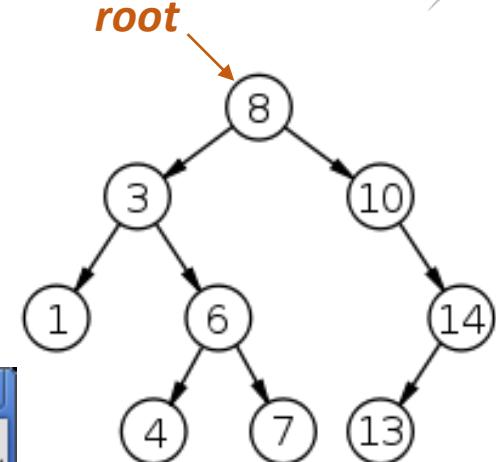
Order of operations changed in the highlighted region

Breadth First Search (DFS)



```
Developer Tools - https://mail.google.com/mail/u/0/?tab=wm
Elements Network Sources Timeline Profiles Resources Audits Console > ⚙️ ✎
<top frame> ▾ Preserve log
> function DepthMapCrawler(root, depthMap, level) {
  // collect nodes at same level
  // in an array keyed by level
  if (Object.keys(depthMap).indexOf(level.toString()) < 0) { // level not there
    depthMap[level] = new Array(); // create level specific array
  }
  depthMap[level].push(root);
  if (root.left) {
    DepthMapCrawler (root.left, depthMap, level + 1);
  }
  if (root.right) {
    DepthMapCrawler (root.right, depthMap, level + 1);
  }
}
< undefined
>
```

```
Developer Tools - https://mail.google.com/mail/u/0/?tab=wm
Elements Network Sources Timeline Profiles Console > ⚙️ ✎
<top frame> ▾ Preserve log
> function TreePrintBFS(root) {
  var depthMap = {}; // associative array for levels
  var startLevel = 1;
  DepthMapCrawler(root, depthMap, startLevel); // collect nodes
  depthMapKeys = Object.keys(depthMap);
  for (var i = 0; i < depthMapKeys.length; i++) {
    var key = depthMapKeys[i];
    nodesAtLevel = depthMap[key];
    outputMsg = "Level " + key.toString() + ": ";
    for (var j = 0; j < nodesAtLevel.length; j++) {
      outputMsg += nodesAtLevel[j].data.toString() + "\t";
    }
    console.log(outputMsg);
  }
}
< undefined
>
```

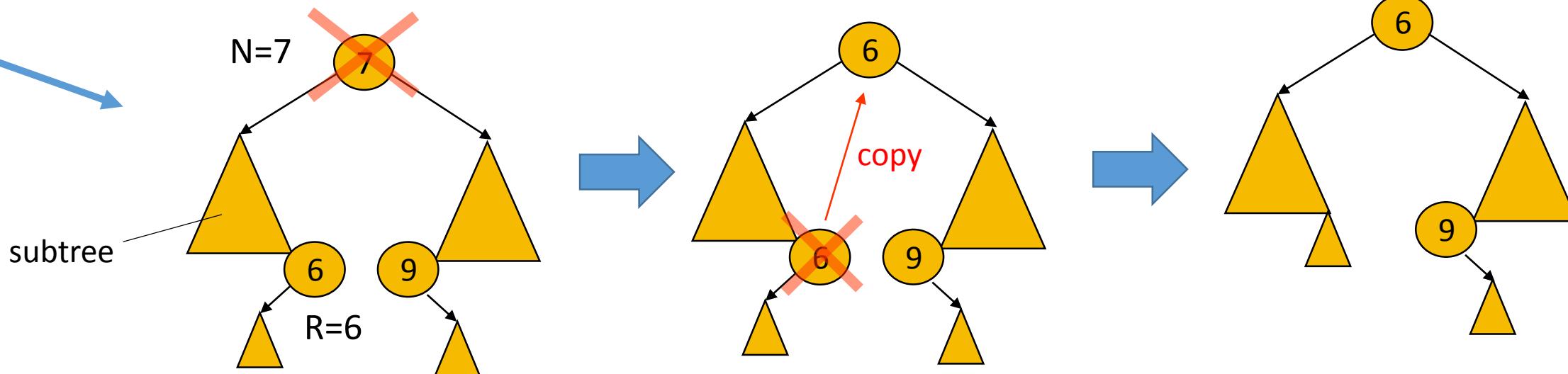


```
Developer Tools - https://...
Console > ⚙️ ✎
<top frame> ▾
> TreePrintBFS(NewTree.root)
Level 1: 8 VM15883:14
Level 2: 3 10 VM15883:14
Level 3: 1 6 14 VM15883:14
Level 4: 4 7 13 VM15883:14
< undefined
>
```

Deleting a node in a BST - idea



- No children: simply remove the node from the tree
- One child: remove the node and replace it with its child
- Two children: call the node to be deleted N. Do not delete N. Choose the rightmost (leftmost) node of left (right) sub-tree. Call it R. Copy value of R to N, make R's left (right) child replace R.



Deleting a node in a BST – program (1)



Developer Tools - http://www.w3schools.com/js/js_object_prototypes.asp

Elements Network Sources Timeline Profiles Resources Audits Console Preset log

```
> MyTree.prototype.DeleteNode = function (value) {
    if (this.root == null) // empty tree special case
        console.node ("Tree is empty");
    else {
        var node2delete = null,
            current = this.root,
            parent = null;
        while (current && !node2delete) { // search till found
            if (value < current.data) { // search left sub tree
                parent = current;
                current = current.left;
            }
            else { // search right sub tree
                if (value > current.data) {
                    parent = current;
                    current = current.right;
                }
                else { // found node, stop looking
                    node2delete = current;
                }
            }
        }
        if (node2delete) {
            if ((NoChildren (node2delete)) && (!parent)) { // one node in tree
                this.root=null; // special case where tree is eliminated
            }
            else{
                DeleteThisNode (node2delete, parent)
            }
        }
        else{
            console.log("No match to delete")
        }
    }
}

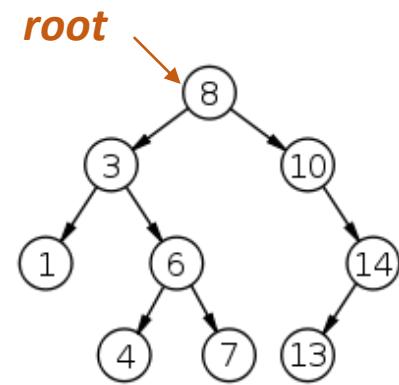
< function MyTree.DeleteNode(value)
> |
```

Deleting a node in a BST – program (2)



```
Developer Tools - http://www.w3schools.com/js/js_object_prototypes.asp
Elements Network Sources Timeline Profiles Resources Audits Console ▶ 🔍 X
Preserve log <top frame>
function DeleteThisNode (node, parent) {
    var NumChildren = (node.left !== null ? 1 : 0) + (node.right !== null ? 1 : 0);
    switch(NumChildren){
        case 0: // no children, just remove mode
            if (node.data < parent.data) {
                parent.left = null;
            }
            else {
                parent.right=null;
            }
            break;
        case 1: // one child, replace node by child
            if (node.data < parent.data) {
                parent.left=(node.left !== null ? node.left : node.right)
            }
            else {
                parent.right=(node.left !== null ? node.left : node.right)
            }
            break;
        case 2: // copy value of rightmost node on left subtree to node to be deleted
            // delete rightmost node
            var tmpparent=node,
                tmpchild=node.left,
                direction=0; // gone left, plan to go right next
            while (tmpchild.right) {
                direction=1; // going right was possible
                tmpparent = tmpchild;
                tmpchild = tmpchild.right;
            }
            node.data = tmpchild.data;
            (direction == 0) ? tmpparent.left = null : tmpparent.right = null;
    }
}
undefined
```

```
Developer Tools - http://www.w3schools.com/js/js_object_prototypes.asp
Elements Network Console ▶ 🔍 X
Preserve log <top frame>
datalist=[8, 3, 1, 6, 4, 7, 10, 14, 13];
NewTree=BuildMyTree(datalist);
NewTree.DeleteNode (81);
No match to delete VM44863:33
undefined
datalist=[8, 3, 1, 6, 4, 7, 10, 14, 13];
NewTree=BuildMyTree(datalist);
NewTree.DeleteNode (8);
TreePrintBFS (NewTree.root)
Level 1: 7 VM23045:26
Level 2: 3 10 VM23045:26
Level 3: 1 6 14 VM23045:26
Level 4: 4 13 VM23045:26
undefined
datalist=[8, 3, 1, 6, 4, 7, 10, 14, 13];
NewTree=BuildMyTree(datalist);
NewTree.DeleteNode (3);
TreePrintBFS (NewTree.root)
Level 1: 8 VM23045:26
Level 2: 1 10 VM23045:26
Level 3: 6 14 VM23045:26
Level 4: 4 7 13 VM23045:26
undefined
```

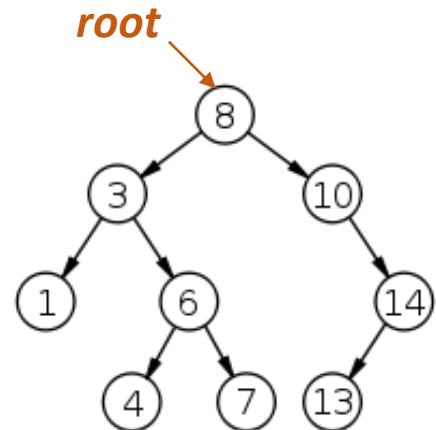


```
Developer Tools - http://www.w3schools.com/js/js_object_prototypes.asp
Elements Network Console ▶ 🔍 X
Preserve log <top frame>
datalist=[8, 3, 1, 6, 4, 7, 10, 14, 13];
NewTree=BuildMyTree(datalist);
NewTree.DeleteNode (6);
TreePrintBFS (NewTree.root)
Level 1: 8 VM23045:26
Level 2: 3 10 VM23045:26
Level 3: 1 4 14 VM23045:26
Level 4: 7 13 VM23045:26
undefined
```

Height of a BST



- Recurse from root to each leaf and find max height
 - Termination condition is a null node of height of 0



```
> function tree_height (root) {
    if (root == null)
        return (0);
    else {
        left_height = tree_height (root.left);
        right_height = tree_height (root.right);
        return (1 + (Math.max (left_height, right_height)));
    }
}
< undefined
```

A screenshot of the Chrome Developer Tools Elements tab. The title bar says "Developer Tools - http://www.al...". Below it is a toolbar with icons for search, element selection, and other tools. The main area shows the DOM tree with the following path:
<top frame>
> tree_height(NewTree.root)
< 4
> |

Complexity of BST operations

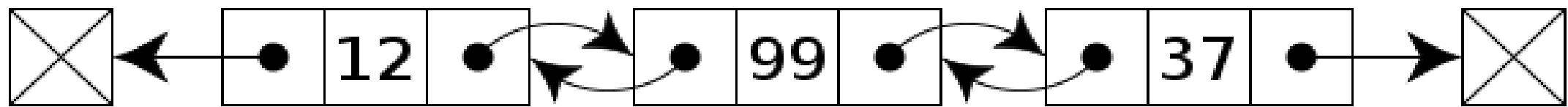


- BSTs keep the values in sorted order
- Operations can use the principle of binary search
 - Go to left or right sub-tree at any point depending on value comparison between stores and searched values
- On average, each comparison allows the operations to skip about half of the tree
- Lookup, insertion, and deletion are $O(\log n)$, where n is the number of nodes in the tree
- This is much better than the linear time $O(n)$ required to find values in an unsorted list or array

Doubly linked list data structure



- Each node has both **previous** and **next** pointers instead of only next as in a singly linked list
- Operations (traversal, lookup, insert, and delete) are really easy at the cost of extra storage of an additional pointer (**previous**) per node
- Can traverse in either direction



Heap data structure



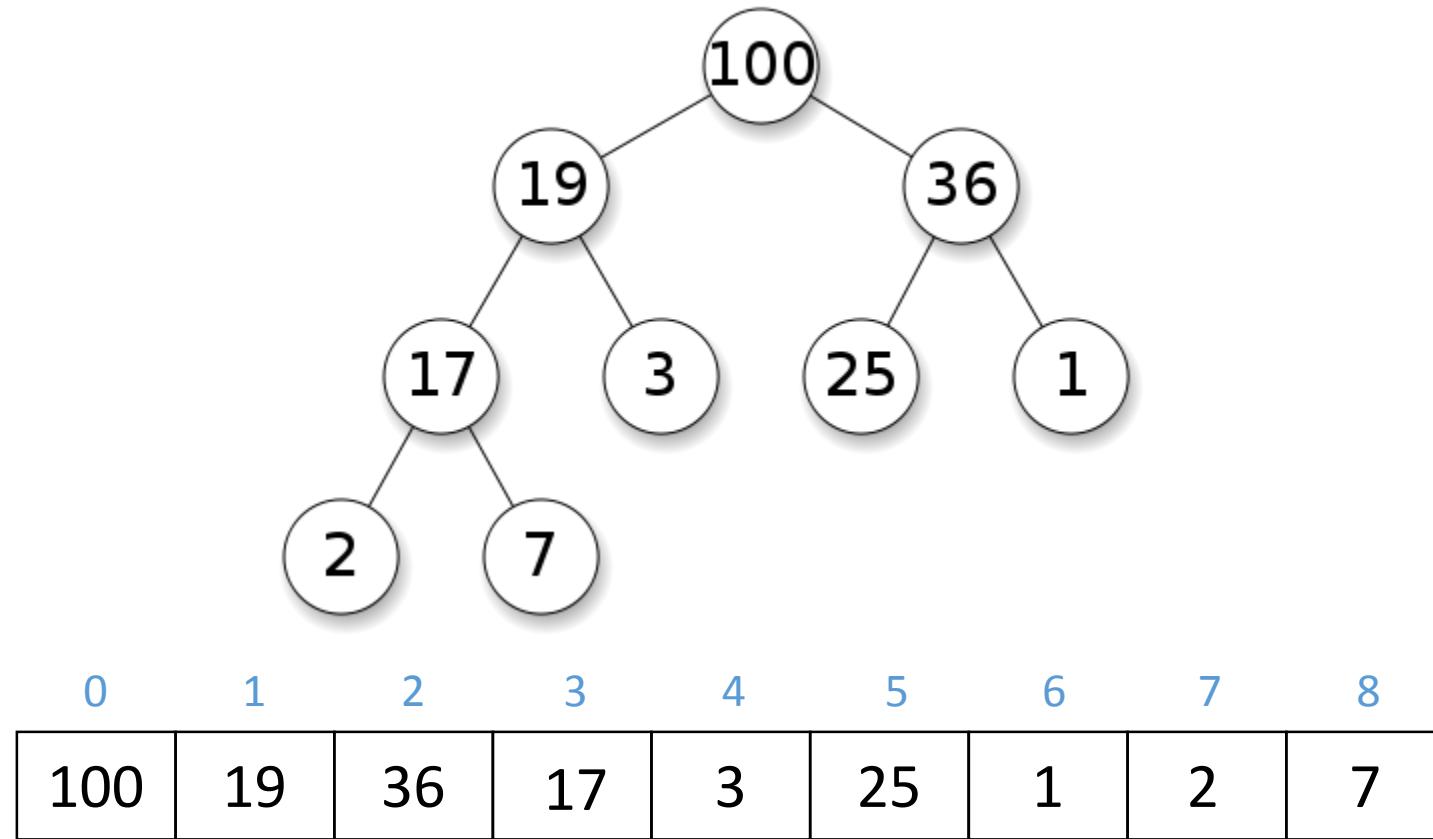
- Satisfies “heap” property with respect to some keys
 - Max heap: Value of parent key greater than those of children
 - Min heap: Value of parent key smaller than those of children
- Highest or lowest element (aka key) is at the root
- A complete binary heap with n elements has $\log(n)$ height
- A deletion or insertion violates the heap property and we need to “heapify” again (max_heapify to get a max heap)
- Can be implemented efficiently using an array that also allows efficient heapify operation without pointers as in binary trees
- Helps implement a priority queue that comes handy in executing jobs with highest priority

Representation using an array (1)



- The first element contains the root
- Next two elements of the array contain its children
- Next four contain the four children of the two child nodes, etc.
- Children of the node at position n would be at positions $2n+1$ and $2n+2$ in a zero-based array ($2n$ and $2n+1$ in a one-based array)
- Allows moving up or down the tree by doing simple index computations
- Balancing is done by swapping elements which are out of order
- Heapsort can be used to sort an array in-place

Representation using an array (2)



Children of 36 @ index n=2 are 25 @index $2n+1=5$ and 1 @index $2n+2=6$

Building a heap - idea



- Get n elements to be put on heap
- Arbitrarily put them on a binary tree using an array
- Array elements indexed by floor ($n/2$) to ($n-1$) are all leaves
- Leaves are conceptually one-element heaps
- Use a recursive function to max heapify in a bottom up manner starting with the leaves

Building a heap – program



Developer Tools - https://en.wikipedia.org/wiki/Binary_heap

Elements Network Sources Timeline Profiles Resources Audits Console

<top frame> Preserve log

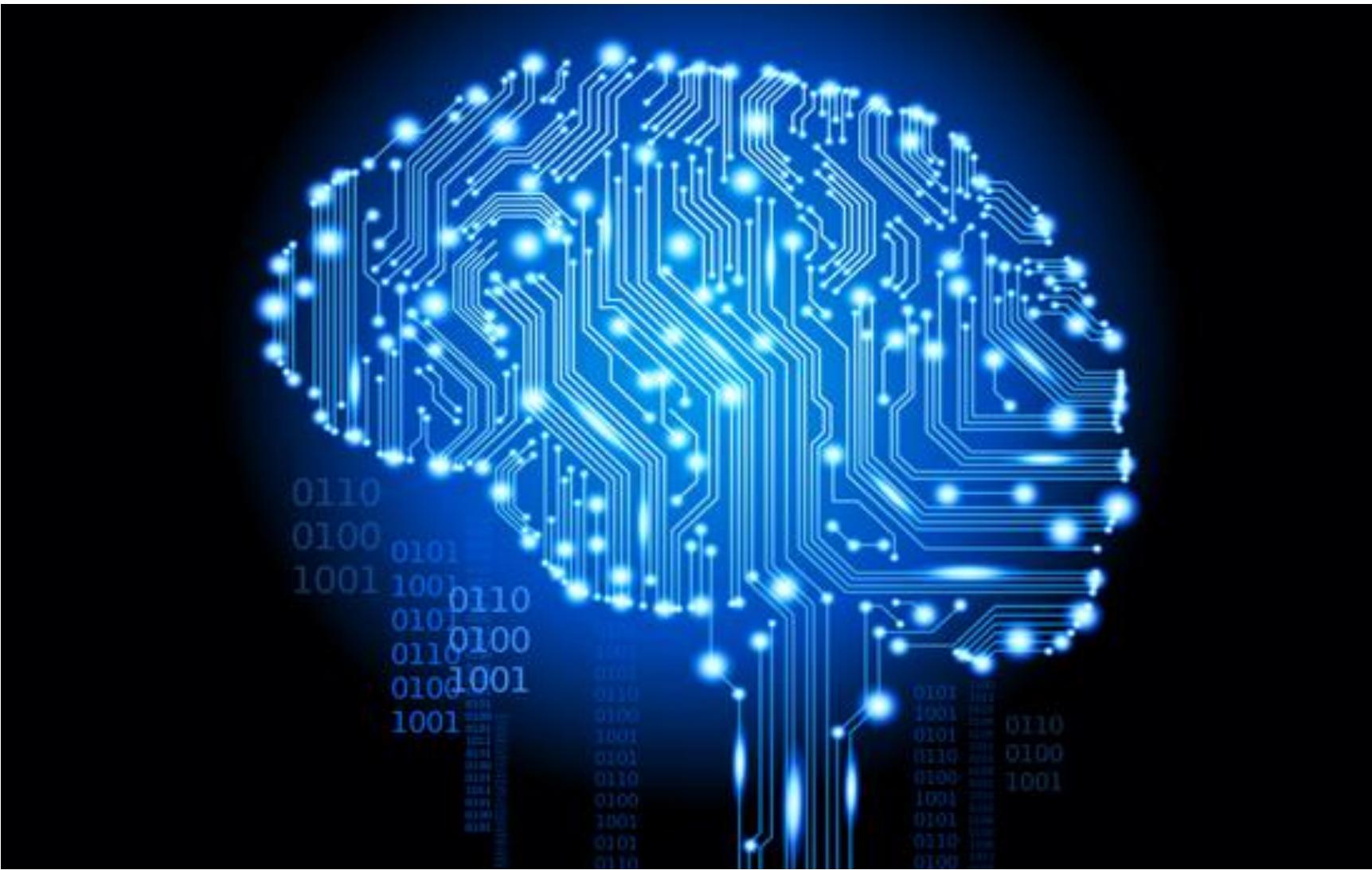
```
> function BuildMaxHeap (nodearray){  
    arraylength=nodearray.length;  
    last_notaleaf_index=Math.floor(arraylength/2)-1;  
    for (var index=last_notaleaf_index; index>=0; index--){  
        // Bottom up heapify  
        MaxHeapify(nodearray, index);  
    }  
  
    function MaxHeapify (nodearray, index){  
        var arraylength=nodearray.length,  
            leftchildindex=2*index+1,  
            rightchildindex=2*index + 2,  
            largestindex = index,  
            tmp=null;  
        if ((leftchildindex <= arraylength) && (nodearray[leftchildindex] > nodearray[largestindex])) {  
            largestindex = leftchildindex;  
        }  
        if ((rightchildindex <= arraylength) && (nodearray[rightchildindex] > nodearray[largestindex])){  
            largestindex=rightchildindex;  
        }  
        if (largestindex != index){  
            //swap to have largest at root of subtree  
            tmp = nodearray[index];  
            nodearray[index]=nodearray[largestindex];  
            nodearray[largestindex] = tmp;  
            // recursive call to propagate effects of above change  
            MaxHeapify(nodearray, largestindex);  
        }  
    }  
< undefined  
>
```

Developer Tools - https://en.wikipedia.org/w...

Elements Network Console

<top frame> Preserve log

```
> datalist=[0, 1, 2, 3, 4, 5, 6, 7, 8];  
BuildMaxHeap(datalist);  
datalist  
< [8, 7, 6, 3, 4, 5, 2, 1, 0]  
> datalist=[2, 7, 25, 1, 17, 3, 36, 100, 19];  
BuildMaxHeap(datalist);  
datalist  
< [100, 19, 36, 7, 17, 3, 25, 1, 2]  
>
```



Introductory algorithms & problems

Complexity



- A good algorithm should be efficient
- Efficiency is expressed as computational complexity
 - Time complexity – how much time it takes (common measure)
 - Space complexity – how much memory it takes
- Time complexity measured by number of required computational steps as a function of the input size
- Rate of growth or order of growth of the running time is used as a measure of complexity
- $O(n)$ is an asymptotic notation symbol used to express the rate of growth

O(n) asymptotic notation



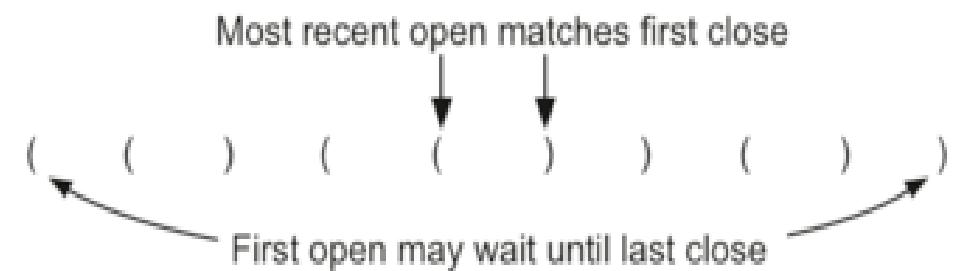
- $O(n)$ = upper bound of complexity specified in terms of input size n
- $O(1)$ is constant time, i.e. running time independent of the input size
- $O(n^2)$ or $O(n^3)$ is okay for small n , but not so okay for large n
- Algorithms with exponential or factorial time complexity are infeasible
- If time complexity can be expressed with a polynomial function, it has **polynomial time complexity**, else it has **exponential time complexity**

	Also called	$n = 100$	$n = 10,000$	$n = 1,000,000$
$O(1)$	Constant time	0.000001 sec.	0.000001 sec.	0.000001 sec.
$O(\lg n)$	Logarithmic time	0.000007 sec.	0.000013 sec.	0.00002 sec.
$O(n)$	Linear time	0.0001 sec.	0.01 sec.	1 sec.
$O(n \lg n)$		0.00066 sec.	0.13 sec.	20 sec.
$O(n^2)$	Quadratic time	0.01 sec.	100 sec.	278 hours
$O(n^3)$	Cubic time	1 sec.	278 hours	317 centuries
$O(2^n)$	Exponential time	10^{14} centuries	10^{2995} centuries	10^{30087} centuries
$O(n!)$	Factorial time	10^{143} centuries	10^{35645} centuries	N/A



Parentheses matching – idea (1)

- Method to read a string of parentheses and decide whether the symbols are balanced is important for programming language structures that use parentheses or curly brackets
 - As we process symbols from left to right, the most recent opening parenthesis must match the next closing symbol
 - The first opening symbol processed may have to wait until the very last symbol for its match
 - Closing symbols match opening symbols in the reverse order of their appearance; they match from the inside out
- Use a stack to solve the problem



Parentheses matching – idea (2)



- Starting with an empty stack, process the parenthesis strings from left to right on symbol at a time
- If a symbol is
 - an opening parenthesis: *push* it on the stack as a signal that a corresponding closing symbol needs to appear later for a match
 - a closing parenthesis: *pop* the stack
- As long as it is possible to pop the stack to match every closing symbol, the parentheses remain balanced
- If at any time there is no opening symbol on the stack to match a closing symbol, the string is not balanced properly
- At the end of the string, when all symbols have been processed, the stack should be empty

Parentheses matching – program



```
Developer Tools - https://www.google.com/webhp?sourceid=chrom... □ X
Elements Network Sources Timeline Console ▶ ⚙ □
✖ 🔍 <top frame> ▾ □ Preserve log

> function Stack () {
  this.stackarray = new Array();
}
< undefined

> Stack.prototype = {
  pop: function() {
    uval = this.stackarray.pop();
    if (uval == undefined)
      console.log("Trying to pop an empty stack");
    return uval;
  },
  push: function(item) {
    this.stackarray.push(item);
  },
  empty: function() {
    value = (this.stackarray.length == 0);
    return(value);
  },
}
< ► Object {}

>
```

```
Developer Tools - https://www.google.com/webhp?sourceid=chrom... □ X
Elements Network Sources Timeline Console ▶ ⚙ □
✖ 🔍 <top frame> ▾ □ Preserve log

> //Check function
function CheckParen (symbol_string) {
  var mystack = new Stack();
  var balanced = true;
  var index = 0;
  while ((index < symbol_string.length) && balanced) {
    symbol = symbol_string[index];
    if (symbol == "(")
      mystack.push(symbol);
    else
      if (mystack.empty())
        balanced = false;
      else
        mystack.pop();
    index = index + 1;
  }
  if (balanced && mystack.empty())
    return "Matched";
  else
    return "Not matched";
}
< undefined

>
```

```
Developer Tools... □ X
Elements ▶ ⚙ □
✖ 🔍
> CheckPare( "()()()" )
"Matched"
> CheckPare( "()()()(" )
"Not matched"
> CheckPare( ")()()" )
"Not matched"
>
```

Sorting



- Ordering items in a list based upon a key
- A sorted sequence makes look up more efficient
- Most common use is to sort numbers from lowest to highest or vice versa
- A sorting algorithm efficiently reorders elements in a list
- There are many sorting algorithms that vary in their time, space, and implementation complexity
- Bubble sort, insertion sort, quick sort, merge sort, heap sort are some examples of sorting algorithms
- Sorting examples in the next slides assume lowest to highest sorting order

Bubble sort - idea



- Consider a list of unsorted numbers
- Compare the first item to the second item and swap if first item is larger
- Compare the second item to the third and swap if necessary, and so on till the end of the list
- At the end of first pass, the smallest item is at the beginning of the list
- Repeat now starting with the second item and do a full pass
- Now the second smallest item is in the second position
- Repeat till the whole list is sorted
- Inefficient algorithm

Bubble sort - program



```
Developer Tools - http://www.nczonline...
Elements Network Console > ⚙️ 🖥️
🔗 <top frame> ▾ 🔍 Preserve log
> function BubbleSort(datalist) {
  listlength=datalist.length;
  for (i=0; i < listlength; i++) {
    for (j=i; j< listlength; j++) {
      if (datalist[i] > datalist[j]) {
        swap(i, j, datalist);
      }
    }
  }
  return(datalist);
}

function swap(i, j, datalist){
  tmp=datalist[i];
  datalist[i]=datalist[j];
  datalist[j]=tmp;
}
< undefined
> |
```

```
Developer Tools - http://www.nczonline...
Elements Network Console > ⚙️ 🖥️
🔗 <top frame> ▾ 🔍 Preserve log
> datalist=[2, 7, 25, 1, 17, 3, 36, 100, 19];
  BubbleSort (datalist)
< [1, 2, 3, 7, 17, 19, 25, 36, 100]
> datalist=[2, 7, 25, 1, 2, 3, 36, 100, 19];
  BubbleSort (datalist)
< [1, 2, 2, 3, 7, 19, 25, 36, 100]
> |
```

Insertion sort - idea



- Consider a list of unsorted numbers
- Consider a sorted section (which is initially the leftmost item) and an unsorted section (which is the remaining list items)
- For each item in the unsorted section, move it to the sorted section and place it at the right location
- Moving of a value from the unsorted to the sorted section may require the so-far sorted items to be shifted to make space for the new value coming in (to place at the right location)

Insertion sort - program



Developer Tools - http://www.nczonline.net/blog/2012/09/17/c...

Elements Network Sources Timeline Console ▶ 🔍 ⚙️

✖️ Preset log

```
> function InsertionSort(datalist) {
    for (j = 1; j <= (datalist.length-1); j++) { //unsorted part
        for (i = 0; i<j; i++) { // sorted part
            if (datalist[i] > datalist[j]) {
                //shift one over to make space
                circular_shift (i, j, datalist);
            }
        }
    }
    return(datalist);
}

function circular_shift (from, to, datalist){
    tmp=datalist[to];
    for (pos=to; pos>from; pos--){
        datalist[pos]=datalist[pos-1];
    }
    datalist[from]=tmp;
}
< undefined
```

|

Developer Tools - http://www.nczonline.net...

Elements Network Console ▶ 🔍 ⚙️

✖️ Preset log

```
> datalist=[2, 7, 25, 1, 17, 3, 36, 100, 19];
  InsertionSort (datalist)
< [1, 2, 3, 7, 17, 19, 25, 36, 100]
> datalist=[2, 7, 25, 1, 2, 3, 36, 100, 19];
  InsertionSort (datalist)
< [1, 2, 2, 3, 7, 19, 25, 36, 100]
>
```

Quick sort - idea



- Consider a list of unsorted numbers
- Find a pivot item in the list (say, the an item near the middle position)
- Compare items and shift with respect to the pivot so that all items to the left of the pivot are smaller than the pivot and all elements to the right of the pivot are greater than the pivot
- Recursively perform the same operations separately on the sub-list elements on the left of the pivot and the sub-list elements on the right of the pivot

Quick sort - program



Developer Tools - https://www.google.com/?gws_rd=ssl

Elements Network Sources Timeline Console »

Preserve log

```
> function QuickSort(datalist) {
  if (datalist.length === 0) {
    return [];
  }
  var pivotindex = Math.floor(datalist.length/2);
  var pivotvalue = datalist[pivotindex]
  var left = [];
  var right = [];
  for (i=0; i<datalist.length; i++){
    if ( i != pivotindex) {
      if (datalist[i] < pivotvalue ){
        left.push(datalist[i]);
      }
      else {
        right.push(datalist[i]);
      }
    }
  }
  return QuickSort(left).concat(pivotvalue, QuickSort(right));
}
< undefined
```

Developer Tools - https://www.google.com/?gws_rd=ssl

Elements Network Console »

Preserve log

```
> datalist=[2, 7, 25, 1, 17, 3, 36, 100, 19];
  QuickSort (datalist)
< [1, 2, 3, 7, 17, 19, 25, 36, 100]
> datalist=[2, 7, 25, 1, 2, 3, 36, 100, 19];
  QuickSort (datalist)
< [1, 2, 2, 3, 7, 19, 25, 36, 100]
>
```

Merge sort - idea



- Consider a list of n unsorted numbers
- Divide the unsorted list recursively into n sublists each containing 1 element
 - A list of 1 element is always sorted by definition
- Repeatedly merge sorted sublists into one final list
- This final list is the sorted list

Merge sort - program



Developer Tools - https://en.wikipedia.org/wiki/Merge_sort

```
<top frame> Preset log

> function SortedMerge (array1, array2){
  // Merge sorted arrays 1 and 2
  var sorted = [];
  if (array1.length == 0){
    return array2;
  }
  else{
    if (array2.length == 0){
      return array1;
    }
    if (array1[0] <= array2[0]){
      sorted.push(array1.shift());
      return sorted.concat(SortedMerge (array1, array2));
    }
    else{
      sorted.push(array2.shift());
      return sorted.concat(SortedMerge (array1, array2));
    }
  }
< undefined
:
```

Developer Tools - https://en.wikipedia.org/wiki/Merge_sort

```
<top frame> Preset log

> function MergeSort (datalist){
  if (datalist.length <= 1) {
    return datalist;
  }
  // Break into two subarrays
  var leftarray = [];
  var rightarray = [];
  var numelement=datalist.length;
  var breakpoint=Math.ceil(numelement/2);
  for (var i=1; i<=breakpoint; i++){
    leftarray.push(datalist.shift());
  }
  while (datalist.length!=0){
    rightarray.push(datalist.shift());
  }
  // Recurse on the subarrays
  leftarray = MergeSort(leftarray);
  rightarray = MergeSort(rightarray);
  // Start merging sorted arrays
  return SortedMerge(leftarray, rightarray);
}
< undefined
:
```

Developer Tools - https://en.wikipedia.org/wiki/Merge_sort

```
<top frame>

> var datalist = [11, 1,
  24, 5, 7, 81, 9, 10,
  2, 6, 8, 2]
MergeSort (datalist)
< [1, 2, 2, 5, 6, 7, 8,
  9, 10, 11, 24, 81]
:

```

Heap sort - idea



- Consider a list of unsorted numbers in an unsorted array
- Heapify the unsorted array
- The largest element is now at the root
- Remove the largest element and put in a new sorted array
- Heapify again and remove the next largest element from the root
- Continue till unsorted list is empty
- The sorted array now contains the sorted elements
- Make use of the BuildMaxHeap function implemented earlier

Heap sort - program



Developer Tools - http://www.w3schools.com/jsref/jsref_unsh... □ X

Elements Network Console >

✖ <top frame> ▼ Preserve log

```
> function HeapSort(datalist) {
    var sortedarray = new Array();
    while (datalist.length>0) {
        // Build max heap
        datalist=BuildMaxHeap(datalist);
        // Remove root and move to sorted array
        sortedarray.unshift(datalist.shift());
    }
    return sortedarray;
}
< undefined
>
```

Developer Tools - http://www.w3schools.com/jsref/jsref_unshift.asp □ X

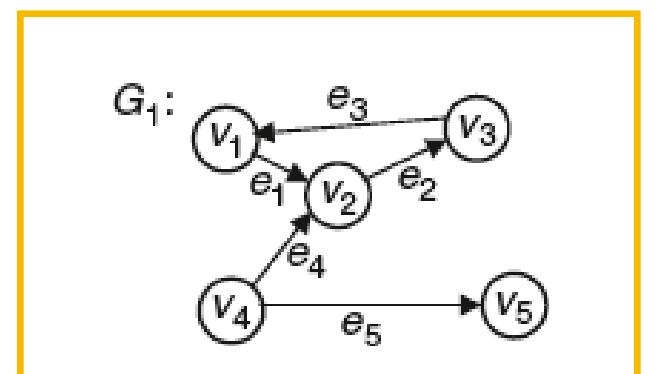
Elements Network Console >

✖ <top frame> ▼ Preserve log

```
> datalist=[2, 7, 25, 2, 1, 17, 3, 36, 100, 19];
  HeapSort (datalist)
< [1, 2, 2, 3, 7, 17, 19, 25, 36, 100]
> |
```

Graphs

- Models pairwise relationships among items
- Abstraction simplifies analysis and solution of specific problems
- Graphs have **vertices** (v) or node and **edges** (e) or arcs or branches
- Two vertices are **adjacent** if there is an edge between them
- **Degree** of a vertex is number of edges connected to it
- **Order** of a graph is number of vertices in it
- **Size** of a graph is number of edges in it
- A **path** is a sequence of alternating connected vertices and edges
 - **Length** of a path is the number of vertices on it
 - **Cycle** is a path with the same first and last vertex
- Directed edges create a **directed graph**

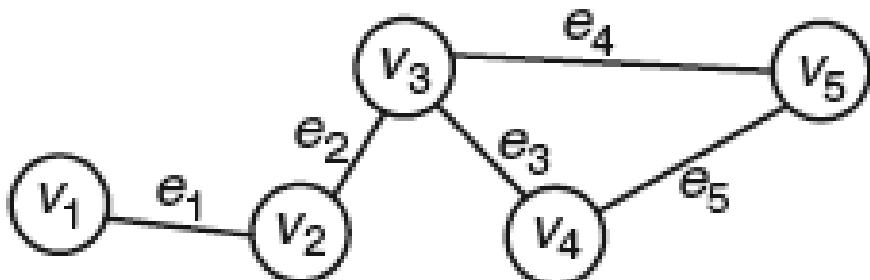


Representing graphs (1)



- Represented by an **adjacency matrix** (A)
- $A(i,j) = 1$ if vertices v_i and v_j are connected
- $A(i,j) = 0$ if vertices v_i and v_j are not connected

Graph



$V = (v_1, v_2, v_3, v_4, v_5)$ = set of vertices

$E = (e_1, e_2, e_3, e_4, e_5)$ = set of edges

$e_1 = (v_1, v_2), \quad e_2 = (v_2, v_3), \dots, etc.$

Adjacency matrix

$A =$

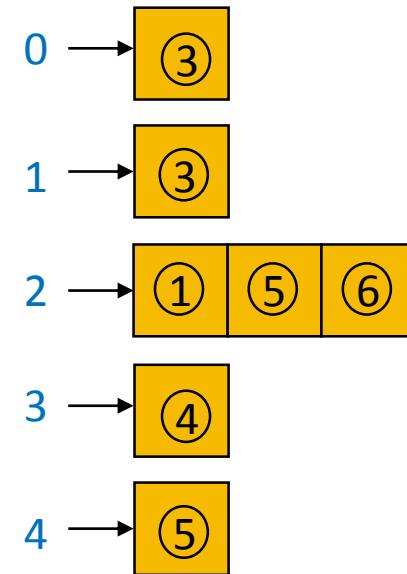
	1	2	3	4	5
1	0	1	0	0	0
3	0	1	0	1	1
4	0	0	1	0	1
5	0	0	1	1	0



Representing graphs (2)

- Alternate representation is by **adjacency list**
- Edges are stored as vertex indexed arrays of vertices adjacent to a vertex
- Allows efficient access of adjacent vertices of a vertex in a program
- If vertex 2 is connected to vertices 1, 5, 6, and is stored in array position 2, accessing this element gives us access to an array stored in position 2 that consists of vertices 1, 5, and 6

Adjacency list



Solving problems using graphs



- Can have node and edge objects with fields depending on whether graph is directed or not (*e.g., from, to, head, tail, connects_to, etc.*)
- Most of the time problems involve traversing graphs in various ways
- Need to develop efficient traversal helper functions using the adjacency representation and vertex and edge objects
- To deal with edge weights (*e.g.,* denoting distance between cities represented as nodes), the 1 in the adjacency matrix can be replaced by the weight value
 - Disadvantage is that we cannot represent a zero weight because it is then indistinguishable from no connection