Computer Science Fundamentals:

Intro to Algorithms, Systems, & Data Structures

Christian J. Rudder

October 2024

Contents

Contents **1**

1 Building a Computer **5**

2 Memory Management **6**
2.1 CPU Arichitecture . . . 6
2.2 Code Security . . . 11
2.3 Stack Data Structures . . . 12
2.4 Heap Data Structures . . . 16
2.5 Hashing & Collisions . . . 23
Open Addressing . . . 24
Searching: Insertion & Deletion . . . 27
Separate Chaining & Linked Lists . . . 28
Load Factor & Performance Metrics . . . 33

3 Computational Algorithms **34**
3.1 Information Theory . . . 34
Defining Information . . . 34
3.2 Number Base System Encodings . . . 39
3.3 Computing Large Numbers . . . 42
3.4 Computational Efficiency . . . 50

Bibliography **53**

This page is left intentionally blank.

Preface

Big thanks to **Christine Papadakis-Kanaris**
for teaching Intro. to Computer Science II,
Dora Erdos and **Adam Smith**
for teaching BU CS330: Introduction to Analysis of Algorithms
With contributions from:
S. Raskhodnikova, E. Demaine, C. Leiserson, A. Smith, and K. Wayne,
at Boston University

Please note: These are my personal notes, and while I strive for accuracy, there may be errors. I encourage you to refer to the original slides for precise information.
Comments and suggestions for improvement are always welcome.

Prerequisites

— 1 —

Building a Computer

— 2 —

Memory Management

2.1 CPU Arichitecture

This section provides a high-level overview of the CPU to provide context/motivation for the following algorithms and data structures.

Definition 1.1: Central Processing Unit (CPU)

The **CPU (Central Processing Unit)**, is a hardware component that *computes* instructions within a computer. Abstract models that define interfaces between hardware and software for a CPU are called **instruction set architectures (ISA)**.

Possible operations are detailed as **opcodes** (operation codes), which are numeric identifiers for each instruction. Moreover, the ISA defines supported data types, **registers (temporary storage locations)**, and addressing modes (ways to access memory).

ISA's are defines the instruction set, which allows for flexibility in hardware performance needs. This various categories:

- **CISC (Complex Instruction Set Computing):** Large number of complex instructions (multiple operations per instruction).
- **RISC (Reduced Instruction Set Computing):** Small set of simple/efficient instructions.
- **VLIW (Very Long Instruction Word):** Enables instruction parallelism (simultaneous execution).
- **EPIC (Explicitly Parallel Instruction Computing):** More explicit control over parallel execution.

Smaller more theoretical architectures exists such as **MISC (Minimal Instruction Set Computing)** and **OISC (One Instruction Set Computing)**, which are not used in practice. Popular CPU architectures include x86_64, and ARM64 (64-bit), originating from x86 and ARM (32-bit).

The implementation of a CPU on a circuit board is called a **microprocessor**. Multiple CPUs on a single circuit board are **multi-core processors**, where each *core* is a fully functional CPU.

Definition 1.2: CPU Anatomy: Von Neumann Architecture

Modern computers operate on the **Von Neumann architecture**, which consists of three primary components:

- **ALU (Arithmetic Logic Unit):** Performs arithmetic and logical operations (e.g., addition, subtraction, AND, OR).
- **Control Unit (CU):** Directs the operation of the CPU, fetching and decoding instructions, and controlling the flow of data.
- **Memory Unit (MU):** Manages data storage and retrieval, including registers and cache memory.

All these components have volatile memory, lost when the computer is turned off.

Definition 1.3: CPU Execution Flow

The **CPU execution flow** is the sequence of operations that the CPU performs to execute a program. It typically follows these steps:

1. **Fetch:** Fetches the next instruction from memory.
2. **Decode:** Decodes the fetched instruction to associated opcode and operands.
3. **Execute:** Perform decoded operation using the ALU or other components.
4. **Store:** Save results of the operation back into memory or registers.

This cycle is repeated until the program completes or an interrupt occurs.

Definition 1.4: Registers

Registers are small, high-speed storage locations within the CPU that hold data temporarily during execution. Common types of registers include:

- **General-Purpose Registers (GPR):** Hold general data storage and manipulation.
- **Special-Purpose Registers:** For specific functions, such as a reference to the current line of code.
- **Floating-Point Registers:** Floating-point arithmetic (e.g., decimal numbers).

Registers are faster than main memory (RAM) and are used to store frequently accessed data during program execution.

The following is an example of the primary registers in the x86-32 (IA-32) architecture, which is a CISC architecture.

Register	Size	Purpose
EAX	32-bit	Accumulator (arithmetic / return value)
EBX	32-bit	Base register (data pointer)
ECX	32-bit	Counter (loops, shifts)
EDX	32-bit	Data register (I/O, multiply/divide)
ESI	32-bit	Source index (string / memory ops)
EDI	32-bit	Destination index (string / memory ops)
EBP	32-bit	Base/frame pointer (stack-frame anchor)
ESP	32-bit	Stack pointer
EIP	32-bit	Instruction pointer (program counter)
EFLAGS	32-bit	Flags / status register (ZF, CF, OF...)

Table 2.1: Primary registers of the x86-32 (IA-32) architecture. **Note:** Registers are prefixed with 'E' for 32-bit, 'R' for 64-bit in x86-64.

Definition 1.5: Machine Code & Compilation

Code is separated into two main areas of memory management, the program itself, and the data in transit during execution. The program itself is broken up such as follows:

- **Text Segment:** The part of the program which contains the executable code.
- **Data Segment:** The part of the program which contains global and static variables.
- **Machine Code:** The compiled code of the program, which is executed by the CPU.

Once the code compiles, our data segment is further divided into two parts in memory:

- **Initialized Data:** Data given a value before the program starts (global variables).
- **Uninitialized Data:** Data yet to be assigned (local variables), which are zeroed at program start.

By memory we mean the **RAM (Random Access Memory)** hardware component, which stores temporary data, constantly communicating with the CPU or external storage (e.g., hard drive, SSD). Each memory cell is IDed by a unique monotonic **address**, often in hexadecimal format(e.g., 0xF00, 0xF01, etc).

Definition 1.6: Operating System (OS)

Implemented ISAs only provide an interface to the CPU; Programmers must design how their systems utilize the CPU (e.g., file and memory management), such software is called an **operating system (OS)**.

Tip: In an analogous sense, say we have a train riding service. The ISA would be the specifications of the trains, rails, routes, and stations needed. The physical implementation of trains, rails, and stations would be the CPU. The OS would be the train schedule system, managing external factors such as workers and other tasks effecting the train service.

Definition 1.7: The Kernel

The **kernel** is a **process** (a program) vital for OS operation, always running with the highest priority. It is the only program that can directly interact with the CPU and various hardware components.

Other processes running on the system are called **user processes**. This is where applications and other user-level programs run. If a user wishes to perform a task that requires hardware access (e.g., writing/reading files), they must request the kernel called a **system call (syscall)**. System calls provide an **Application Programming Interface (API)** for user processes to interact with the kernel.

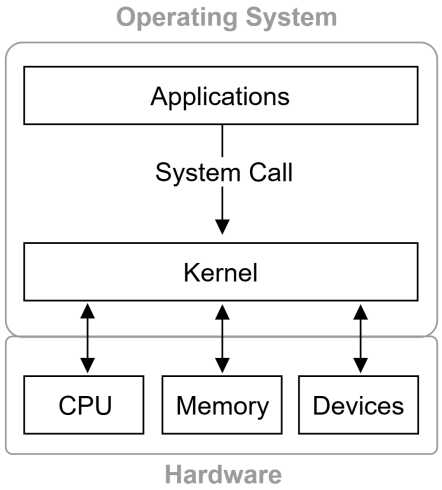


Figure 2.1: User-level applications make syscalls to the kernel to access hardware resources.

Definition 1.8: Bus

A **bus** is a collection of physical signal lines (wires or pins) and protocols that carry data, addresses, and control signals between components inside a computer (e.g. CPU, memory, I/O devices) or between multiple boards and peripherals. There are two main types of buses:

- **Serial Bus:** Transfers data one bit at a time over a single channel (e.g., USB).
- **Parallel Bus:** Transfers multiple bits simultaneously over multiple channels (e.g., PCI).

Definition 1.9: Device Drivers

The kernel exposes generic interfaces to various sub-systems (e.g., file system) that user processes can use to perform tasks; **Device drivers** implement such interfaces, translating generic system calls into hardware-specific operations for specific devices (e.g., disk drives, network cards, etc.). Drivers must be loaded into kernel space.

This text does not concern assembly code, so **do not** get caught in the specifics of this Example:

Example 1.1: Assembly Code

An assembly example demonstrating initialized (.data) and uninitialized (.bss) data sections:

```
section .data                   ; Initialized data section
    num1    dd  7               ; num1 is initialized to 7
    num2    dd  3               ; num2 is initialized to 3

section .bss                    ; Uninitialized data section
    temp    resd 1              ; temp is reserved (uninitialized)
    result  resd 1              ; result is reserved (uninitialized)

section .text                   ; Code section
    global _start

_start:
    mov eax, [num1]             ; Load num1 into eax
    mov [temp], eax             ; Store num1 in temp
    mov ebx, [num2]             ; Load num2 into ebx
    add eax, ebx                ; Add num2 to eax (eax = num1 + num2)
    mov [result], eax           ; Store the sum in result
; Exit syscall removed for simplicity
```

In this example, 'num1' and 'num2' are initialized before execution, while 'temp' and 'result' are uninitialized and only receive values during program execution. ■

2.2 Code Security

At a *very* high-level, vulnerabilities exploited by hackers stem from flaws that the programmer forgot to consider (i.e., bugs). To learn more on cybersecurity, consider our other text here.

Definition 2.1: Proper Encapsulation

Proper encapsulation is the practice of hiding implementation details and exposing only necessary interfaces to prevent unauthorized access or modification.

Example 2.1: Student Class

Consider a simple 'Student' class in an object-oriented programming language:

```
public class Student {
    private String name; // Private field, not accessible outside
        the class
    private int age;     // Private field, not accessible outside
        the class

    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() { // Public method to access name
        return name;
    }
    // Other methods...
}
```

Upon creating a new student instance `new Student("Alice", 20)`, the name and age are private, preventing direct access via **dot notation** (e.g., `student.name`). The only way to access the name is through the public method `getName()`. Here we do not have a method for accessing age. ■

Definition 2.2: Risks of Accessing Main Memory

Programs accesses main memory (RAM) to read and write data; **It's critical** that such references to RAM are abstracted to avoid malicious or accidental access of data.

For example, in Java when users print objects, instead of printing the object's memory address, it prints the `toString()` method, which **by default** prints the class name and hash code of the object.

In conclusion, there are significant risks when dealing with memory management.

2.3 Stack Data Structures

Let's talk about our first data structure, the stack:

Definition 3.1: Stack

A **stack data structure** is a collection of elements that follows a **Last In, First Out** (LIFO) principle. I.e., in a stack of plates, the last added plate is the first one to be removed, not the middle or bottom/first plate. Each *plate* in the stack is called a **stack frame**.

A **call stack** is a stack which keeps track of function calls in a program as well as any local variables within such functions.

This is why we say a variable is in **scope**, as when a function is taken off the stack, or a new stack frame is placed on top, the variables in the previous or discarded stack frame are **no longer accessible**.

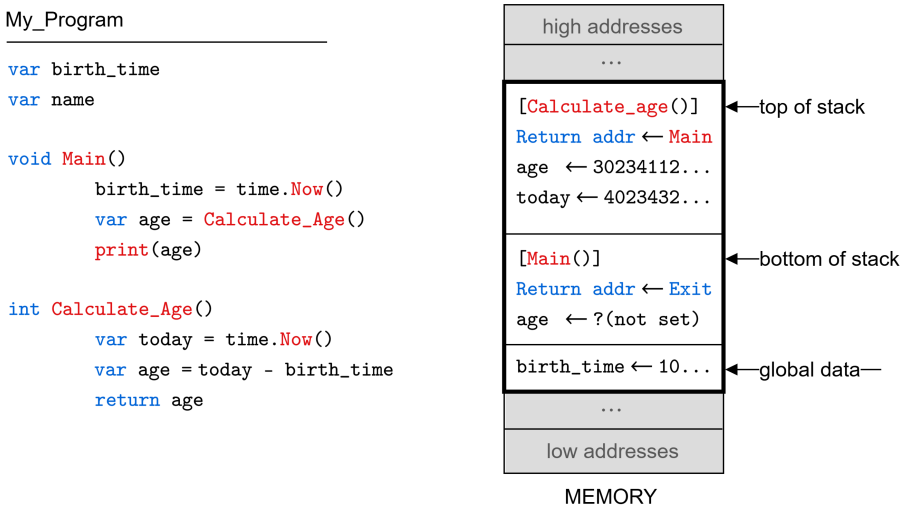


Figure 2.2: Here is a simplified look at how memory manages the stack. On the left is our program written in some abstract language, and on the right is the call stack in memory (simplified). The program has a global 'birth_time' variable, which is initialized in the `Main` function. The `Main` function then calls the `Calculate_Age` function which uses the 'birth_time' variable to calculate the 'age' via the difference of the current time and the 'birth_time.' Looking at the memory, we see at the bottom of our memory contains global variables accessible to any frame. Next, is the bottom of the stack, containing a return address to exit the program, while awaiting the result of the function call for 'age'. The top of our stack contains another frame that we will return the value a new 'age' (not the same as the one before) not accessible from the main function. This new frame also contains a new local variable 'today.' Once this function returns, `Main` will have the result of its local variable 'age.'. Concretely, the 'age' variable in both the `Main` and `Calculate_Age` function are completely separate despite sharing the same *name*.

Please Note: The above figure is a simplified version; This presentation derivatives from what actually happens for teaching sake. In the following pages we define the stack frame in more detail.

Tip: A lot of demonstrations (including this text) will show the stack growing **upwards**; This is strictly because it's easier to visualize and does not accurately portray what a stack really does or looks like. In the following pages we will clear this up, and show how the stack actually grows from top-to-bottom. Of course, there is always room for deviation if a developer wishes to implement a stack in some other arbitrary way. Nonetheless, the following is what one might typically expect in a stack implementation.

Definition 3.2: Stack Frame Anatomy

Under the x86-32 calling convention Two registers keep track our place in the stack:

- **Base Pointer (BP/EBP):** Points to the base (i.e. "bottom") of the current function's stack frame.
- **Stack Pointer (SP/ESP):** Points to the "top" of the current function's stack frame, i.e., the next free byte where a push would land.

When the program starts, the operating system *reserves* a contiguous region of memory for the stack. By convention, the *bottom* of that region lies at a higher address, and the stack "grows downward" toward lower addresses as data is pushed. If the stack pointer ever moves past the reserved limit—a **stack overflow** occurs.

A single **stack frame** itself is a contiguous block of memory in which the function stores:

- **Parameters:** The arguments passed in by the caller,
- **Return Address:** The address of the next instruction to execute after the function returns,
- **Old Base Pointer:** The caller's 'EBP', saved so that on return we can restore the previous frame,
- **Local Variables:** Space for any locals or temporaries that the function needs.

This is why variables in previous or new functions calls become "**out of scope**" (no longer accessible), as they belong to some other stack frame; When it comes to **Global Variables**, they live in a separate region of memory, defined by the **data segment** (1.5).

Moreover, a call to a new function invokes the **call** instruction, this automatically pushes the return address to the current frame onto the stack. Additionally, the CPU reserves the **EAX** register for the return value (number or address) of a function. When the function returns, it can place its result in 'EAX', and the caller can retrieve it from there. During constant use the 'EAX' register may contain **garbage** data from previous use, unless explicitly set to zero or some other value.

High Addresses		
Contents	**Offset**	**Notes**
(Parameters 3, 4, ...)	$EBP + 16, +20, \ldots$	Third-and-onward arguments, if any.
Parameter 2	$EBP + 12$	Second argument passed on stack.
Parameter 1	$EBP + 8$	First argument passed on stack.
Return Address	$EBP + 4$	Auto-pushed by the `call` instruction.
Old EBP (Saved BP)	$EBP + 0$	The caller's base pointer
Current Frame (locals/temporaries)		
Local Variable 1	$EBP - 4$	First 4-byte local (or smallest slot).
Local Variable 2	$EBP - 8$	Next 4-byte local or part of a larger object.
...	⋮	(additional locals at $EBP - 12, -16, \ldots$)
Low Addresses		

Table 2.2: Typical x86-32 Stack-Frame Layout, where offsets are typically a multiple of 4 bytes.

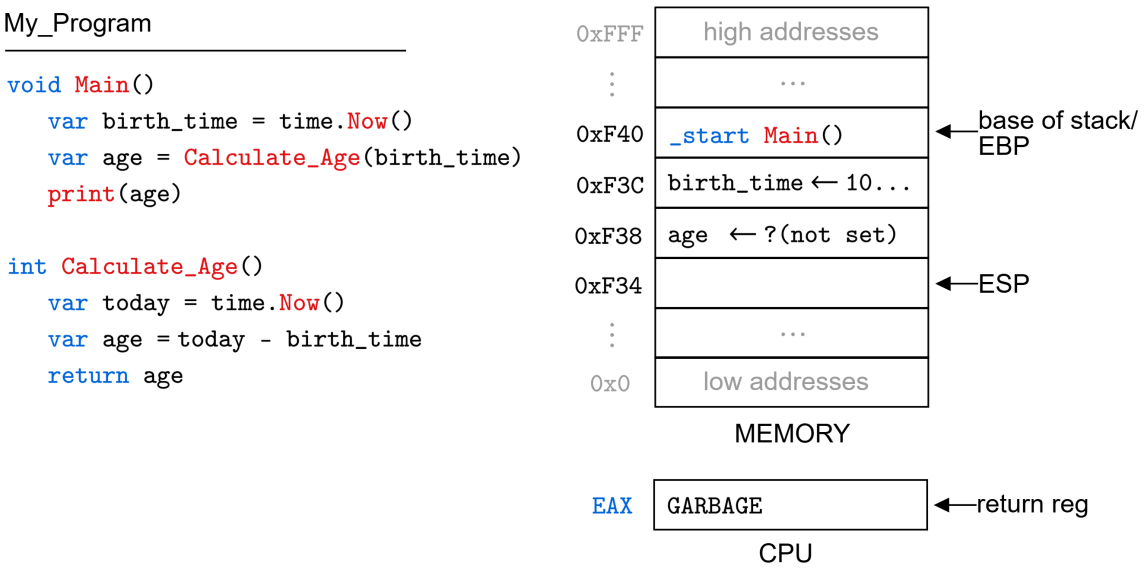


Figure 2.3: Revisiting Figure (2.2) with slight alterations to the code: This is a snapshot of the code executing right before `CalculateAge(birth_time)` is called. For simplicity sake, let's say the stack begins at address `0xF40` (Hexadecimal), growing downwards. Here the base of the stack and the EBP are one and the same. We include the CPU's EAX (return register), which contains garbage. Address `0xF38` is currently just reserved space for 'age'.

My_Program

```
void Main()
    var birth_time = time.Now()
    var age = Calculate_Age(birth_time)
    print(age)

int Calculate_Age()
    var today = time.Now()
    var age = today - birth_time
    return age
```

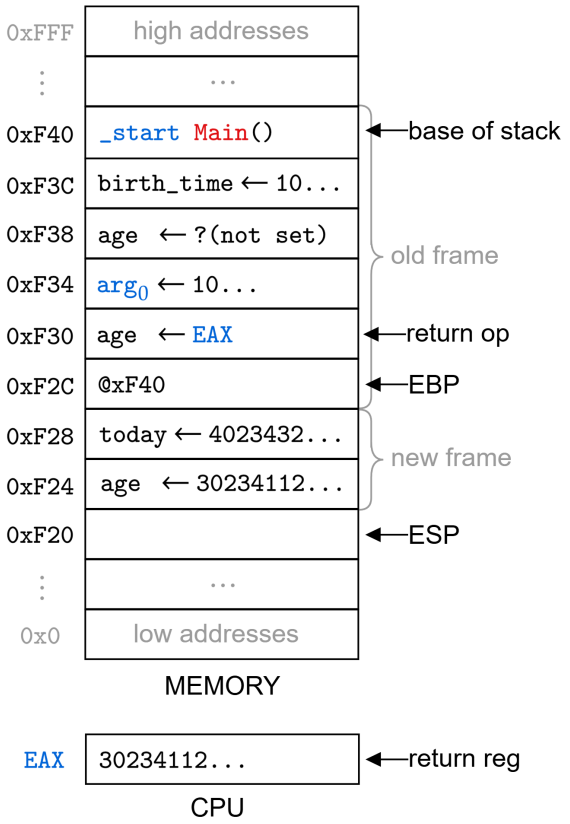


Figure 2.4: Revisiting Figure (2.3) at the moment the function `CalculateAge(birth_time)` has supplied its return value to the EAX register, and is about to return. We see that before calling `CalculateAge(birth_time)`: The old frame pushed it's arguments (`birth_time`) onto the stack, then the return address (IP/Next Instruction) onto the stack, and finally the old EBP (Base Pointer) onto the stack. The 'new frame' then sets the saved EBP address to the current EBP, concluding the old frame into the 'new frame'. Moreover, since the offset looks for local variables below `0xF40`, the above 'birth_time' and 'age' are **out of scope** for the 'new frame', vice-versa. **Note:** This is still a high-level abstraction of what actually happens sequentially with opcodes; Nonetheless, this is the fundamental idea of how a stack works.

This concludes our discussion on stack structures; We continue with the heap structure next.

2.4 Heap Data Structures

So far we have simply said global data is declared in the **data segment** of memory. There is a second segment of memory that builds ontop of this called the **heap**:

Definition 4.1: Heap – Dynamic vs. Static Memory

When a program runs there is a **static** (fixed) region reserved for the program's data segment (local/global variables). During execution, more objects may be created, needing additional memory; A new region of memory is reserved **dynamically**, building upwards from the top of the data segment, called the **heap**.

Language protocols either **manually** (e.g., Assembly, C) or **automatically** (e.g., Python, Java) manage this memory:

- **Manual Memory Management:** The programmer must explicitly allocate and deallocate memory using functions like 'malloc' and 'free' in C.
- **Automatic Memory Management:** The language runtime automatically allocates and deallocates memory, often using a **garbage collector** to reclaim unused memory (no variables pointing to it).

Unlike the stack, this allows values to be accessed from anywhere in the program, regardless of the function call or scope.

We discuss hash tables more in-depth in the following section, for now we provide a high-level idea:

Definition 4.2: Hash Table

A **hash table** uses a **hash function**, taking a **key** (e.g., number or string) and producing a fixed-sized **hash** value (index), creating a table of mappings (key→hash). At such indices lies data associated with the key, enabling fast data retrievals.

A **universal hash function** is a hash function that uniformly distributes hashes across the hash table.

Note: The input in many context (typically cryptographic), may be called 'data' or 'message'; The output: hash, checksum, fingerprint, or digest.

Definition 4.3: Arrays in Memory

Arrays list elements sequentially in memory. A reference to an array is a pointer to the first element. To terminate reading an array, we must either know the size of said array or have some **sentinel value** (e.g., 'null' or '0') to indicate the end of the array.

Consider the following examples:

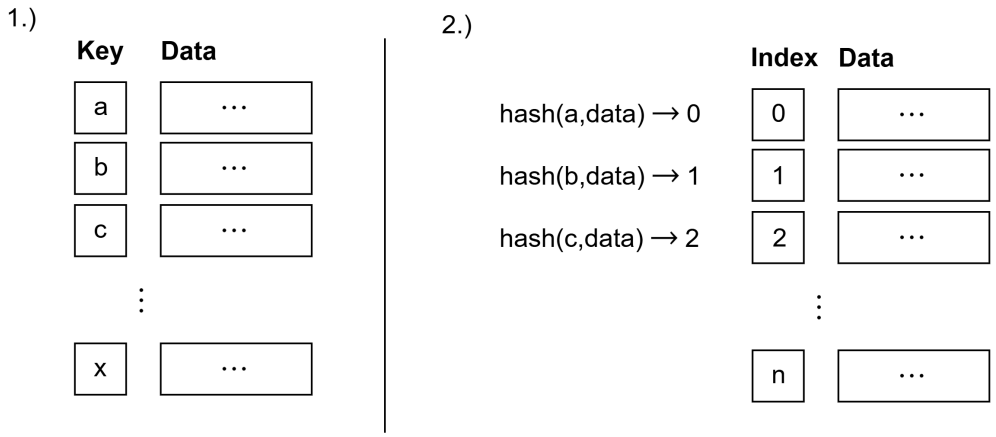


Figure 2.5: On the left (1) demonstrates a typical diagram one might find when learning about hash tables. Here a through x are the keys, which house some type of data. On the right (2) shows a slightly more detailed version, which emphasizes that keys a–x are hashed to indices 0–n in the hash table. The data could be any other value (e.g., number, string, or object). Moreover, hash tables under the hood are arrays, with each index pointing to whatever data is associated with the key.

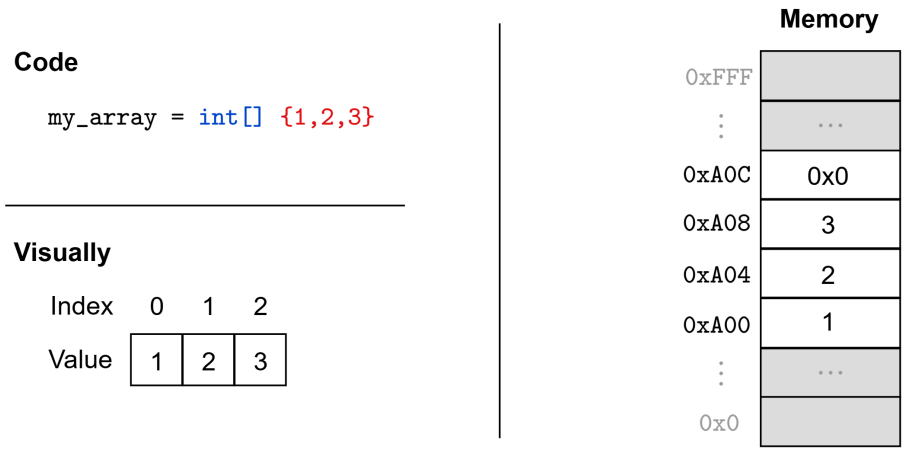


Figure 2.6: Here there are three sections breaking down how arrays look: In code, typical diagram depictions (Visually), and in memory. The code depiction illustrates a toy language creating an array of numbers ($[1, 2, 3]$). The visual depiction shows how indices relate to values. The memory depiction shows how the array is laid out in contiguous memory locations, where `0x0` is the sentinel value (a bit-pattern of all zeros). In code, `my_array` holds the address `0xA00`.

Objects behave very similarly to arrays, but with a few key differences:

Definition 4.4: Objects in Memory

An **Object** (or **struct**), is a collection of key-value pairs, where each key is called a **field** or **attribute** and each value can be any data type (e.g., number, string, or address).

Attributes are stored in array like fashion, where each element is a fixed-offset from the head (start) of the object. The object itself is a pointer to the first element. Accessing attributes works differently in compiled (e.g., C) vs. interpreted (e.g., Python) languages:

- **Compiled Languages:** There is no lookup, as the compiler has *hardcoded* the offsets of each attribute interaction (e.g., 'object.attribute' is translated to a direct memory access).
- **Interpreted Languages:** The interpreter looks up a hash table lookup for the attribute name.

Depending on the use case, objects may be stored in the heap or stack:

- **Static Objects:** An objects whose size is known at compile time can be allocated on the stack. I.e., no changes to the object are made after creation (e.g., Math and Time objects, which purely exist to compute).
- **Dynamic Objects:** Often just called **objects**, are allocated on the heap, allowing for dynamic resizing and modification (e.g., a student object with attributes like 'name', 'age', and 'grades' that can change over time).

Languages like Java push this even further by allowing both static (shared) and dynamic (personal) fields within a class.

Definition 4.5: Object-oriented – Classes, Interfaces, & Polymorphism

Object-oriented programming is a paradigm where objects are the main building blocks of the program. A **class** is a blueprint for defining how an object will behave once **instantiated** (created). In this paradigm, functions are called **methods**, as they are defined and used within the class (i.e., globally does not exist in independence).

Some languages (e.g., Java, C++) support **interfaces** (or protocols), which specify a set of methods that implementing classes must provide. Although the terminology varies (abstract classes, traits, protocols, etc.), they all ultimately describe capabilities an object must fulfill.

Inheritance is the main motivation behind classes and interfaces, enabling a **child** (sub) class to reuse or extend the functionality of a **parent** (super) class. To further reduce redundancy, **Polymorphism** allows objects to **override** (redefine) methods of the same signature (variable name) to accept different types of data or behave differently based on their context.

Example 4.1: Java – Classes, Interfaces, Abstracts, Inheritance, & Polymorphism

Consider the following Java code:

```java
public interface Animal { // Rough blueprint
    void eat(); void sleep(); void sound();
}

public abstract class Cat implements Animal { // Partial blueprint
    @Override
    public void eat() {
        System.out.println("Cat eats fish");
    }

    @Override
    public void sound() {
        System.out.println("Meow");
    }

    // Abstract method to demonstrate subclass-specific behavior
    public abstract void run();
}

public class Cheetah extends Cat { // Inheritance from parent Cat class
    @Override
    public void run() {
        System.out.println("Cheetah runs at 120 km/h");
    }

    @Override
    public void sound() {
        System.out.println("Chirp");
    }
}

public class Main {
    public static void main(String[] args) {
        // Polymorphism: reference is Animal, instance is Cheetah
        Animal anim = new Cheetah();
        anim.eat();       // calls Cat.eat()
        anim.sound();     // calls Cheetah.sound()
    }
}
```

Java polymorphism allows parent types to host children instances as seen with `Animal anim = new Cheetah();`, but `Cheetah chet = new Cheetah();` also works. This allows us to create arrays of different animal types:
E.g., `Animal[] zoo = {new Cheetah(), new Lion(), new Elephant()};`,
assuming they all implement the `Animal` interface. ■

Strings are not what they seem:

Definition 4.6: Strings & Characters in Memory

A **character** is represented by a numeric code unit:

- In C, a single `char` (1 byte) typically holds an ASCII code (0–127). Characters beyond U+FFFF use two `char` values, a **surrogate pair**.
- In Java, `char` is a 16-bit UTF-16 code unit (U+0000..U+FFFF). ASCII values (0–127) map directly to the same Unicode code points. We can take advantage of the encoding:

```
char c = 'A';
System.out.println((int)c);  // prints 65, since 'A' is U+0041
```

This allows us to do things like checking for valid characters:

```
if ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z')) {
    // c is in 'a'..'z' or 'A'..'Z'
}
```

We can also perform arithmetic on `char`:

```
char c = 'A';                // U+0041 (65)
char next = (char)(c + 1);   // 'B' (66)
```

Typically, a **string** is stored as a contiguous array of **characters**. In low-level languages (e.g. C), that array ends with a null terminator (`\0`) and literal strings reside in the data segment. In higher-level languages (e.g. Java, Python), strings are full objects with methods. For e.g.,

C:

- String literals (e.g. `"Hello"`) are placed in the (often read-only) data segment.
- Runtime-constructed strings (via `malloc`, `strcpy`, etc.) live on the heap.

Java:

- Compile-time literals are **interned** (stored as a single shared copy) into the **String Constant Pool** section (specially reserved on the heap).
- Any other `String` (e.g. via `new String(...)`, concatenation, or user input) also resides on the heap but outside the pool.
- Because Java strings are immutable, interning lets multiple references share the same character data.

With slight alterations to our code in Figure (2.5), we illustrate heaps and arrays:

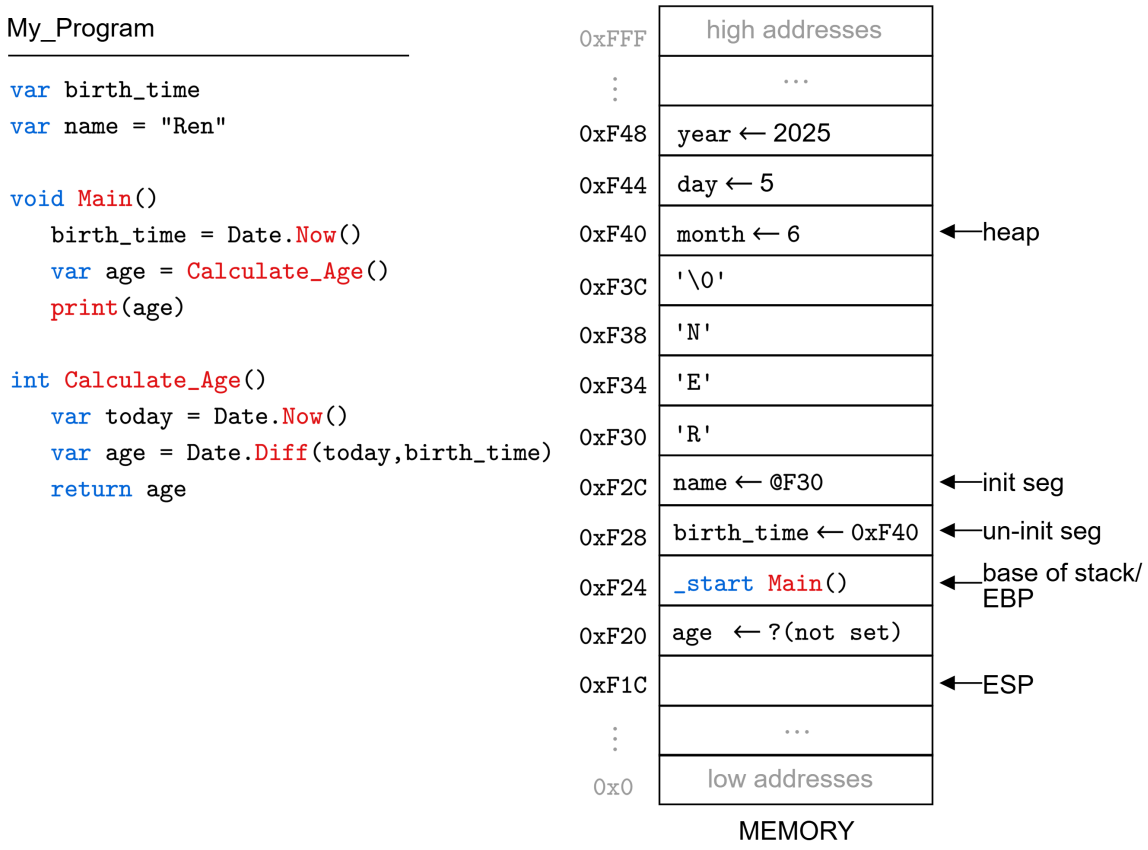


Figure 2.7: Here `birth_time` and `name` are global variables. Following the C convention, `birth_time` is placed in the uninitialized data segment, while `name` is placed in the initialized data segment. Since `name` is a string, it holds a reference to the first character in the string, which is stored contiguously in the initialized data segment ending with a null terminator (‘\0’). During execution, `Date.Now()` a method call from a `Date` object is called; This method returns a new object, which is placed on the heap with its attributes (`month`, `day`, `year`) stored contiguously in memory. **Note:** Methods such as `Date.Diff()` are code (not data), which do not live in the heap or stack.

Definition 4.7: Factory Method

A **factory method** is a function that creates and returns an object, often initializing it with default values or parameters. In Figure (2.7), the method `Date.Now()` is a factory method that creates a new `Date` object with the current date and time.

The below illustration summarizes the heap and stack in memory:

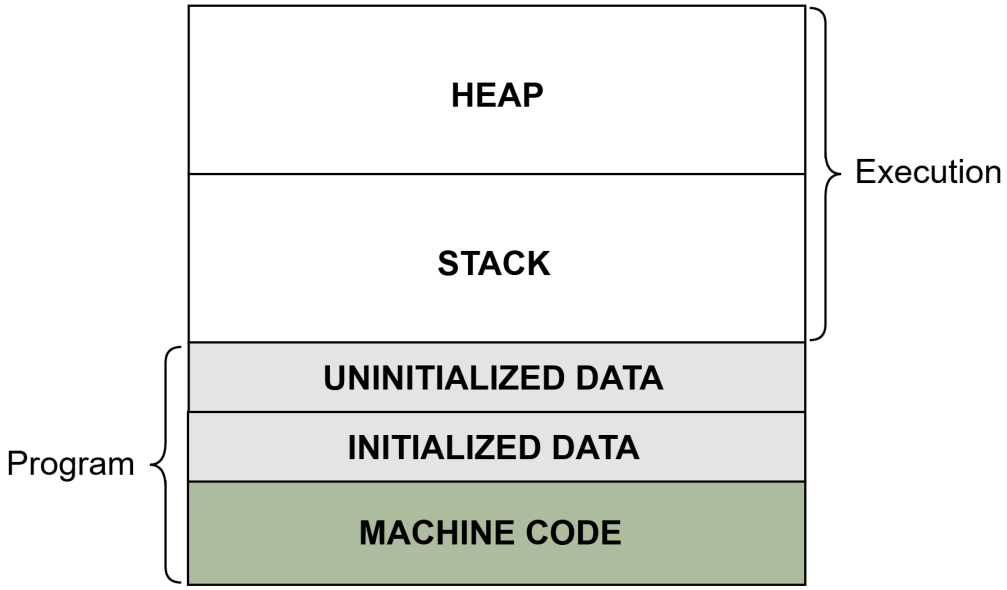


Figure 2.8: The above figure demonstrates the relationship from bottom-to-top the order at which data is loaded into memory. First the program compiles to machine code and loaded by the OS into memory. From there, provisions to the data segment (static memory: uninitialized and initialized) are made. Depending on the OS, some objects may have already been loaded into the heap, which are referenced by initialized data segment variables. Then as functions are called, the stack grows downwards within its allotted memory space. During execution of each stack frame, new objects may be placed on the heap, referenced by variables in the stack or data segment. Then depending on the language, a garbage collector periodically checks for objects with no references (i.e., no variables pointing to them) and deallocates them; Alternatively, the program explicitly deallocates memory using functions like 'free' in C.

2.5 Hashing & Collisions

In the previous section we lightly touched on the topic of hashing in Definition (4.2). This section will dive into more detail and difficulties collisions in hashing.

Definition 5.1: Collisions

A **collision** occurs when two different keys hash to the same index in a hash table. This is an unavoidable issue in hashing when keys begin to exceed the available indices.

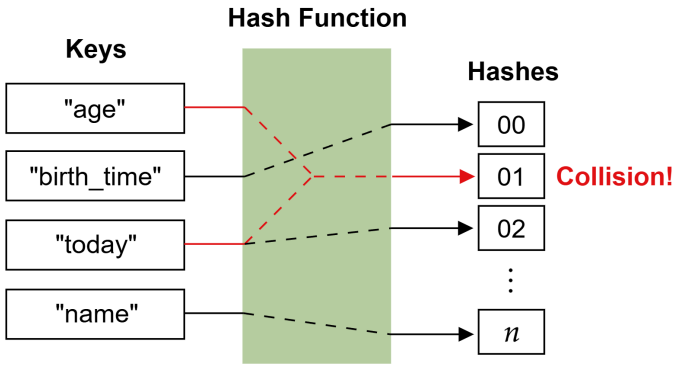


Figure 2.9: Four keys ('age', 'birth_time', 'today', 'name') go through a hash function to n possible indices. Keys, 'birth_time' and 'name', find a unique one-to-one mapping; However, 'age' and 'today' both hash to the same index, causing a collision.

Example 5.1: Simple Hashing Algorithm

Consider the hashing algorithm H, it takes the first ASCII value modulo the size of the table. Concretely, $H(k) := \text{ASCII}(k[0]) \ \% \ n$, where n is the size of the table.

Given the function H, we consider the following keys under a hash table of size 10:

- **Key:** 'apple' $\rightarrow$ ASCII value $= 97 \rightarrow H(\text{apple}) = 97 \ \% \ 10 = 7$
- **Key:** 'banana' $\rightarrow$ ASCII value $= 98 \rightarrow H(\text{banana}) = 98 \ \% \ 10 = 8$
- **Key:** 'bread' $\rightarrow$ ASCII value $= 98 \rightarrow H(\text{bread}) = 98 \ \% \ 10 = 8$

Here, we see that 'banana' and 'bread' both hash to index 8, causing a collision. ■

One could have a superb hashing algorithm, but when space is tight, collisions are inevitable. We'll look at two particular methods for dealing with this issue.

Open Addressing

Our first method:

Definition 5.2: Open Addressing

Open addressing is a collision resolution method where, upon a collision, the algorithm searches for the next available slot via a probing sequence.

Wrap Around: the algorithm uses a modulo operation (e.g., Given a table size of 10 and request for index 12, the algorithm would use $12 \% 10 = 2$).

Time Complexity: $O(n)$, where n is the number of elements in the hash table. For example, say the only free index is at 0 with all other indices occupied. If we hash to index 1, the algorithm will have to walk all n indices to find the free index at 0. Changing the probe method only switches order of indices checked, not the worst case.
Space Complexity: $O(n)$, where n is the size of the hash table (no additional space).

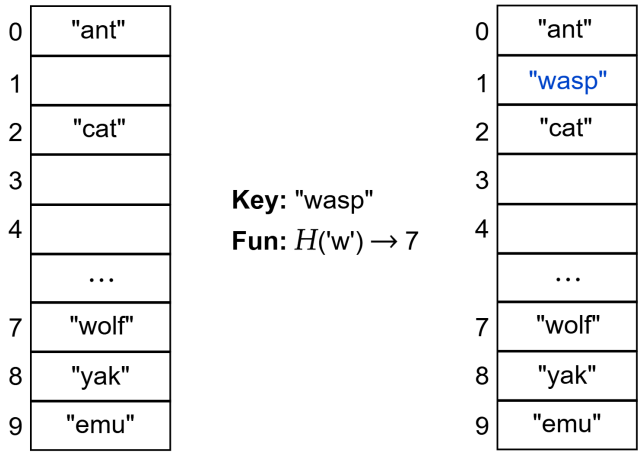


Figure 2.10: On the left is an existing hash table of 10 elements filled with various keys. The middle shows the insertion of a new key, 'wasp', which the function H hashes to index 7; However, index 7 already occupied. The algorithm walks through the table, wrapping around to the beginning, finding a free index at 1. The right shows the final state of the hash table with 'wasp' inserted at index 1.

Definition 5.3: Linear Probing

Linear Probing in open addressing refers to sequentially checking each index for an available slot (e.g., Figure 2.10).

Definition 5.4: Quadratic Probing

Given a universal hashing function $H(x)$, a **quadratic probing** resolves collisions by defining,

$$h(x,k) := (H(x) + k^2) \ \% \ n$$

Where k defines the number of collisions, and n hash table size. The algorithm may **never discover** particular cells due to its even probing style. We **terminate execution** once n indices have been checked, avoiding an infinite loop.

So why even use quadratic probing?

Definition 5.5: Clustering

Clustering is a phenomenon in open addressing where multiple keys form contiguous *runs* of occupied indices. This degrades linear probing performance; Such is the main motivation behind quadratic probing.

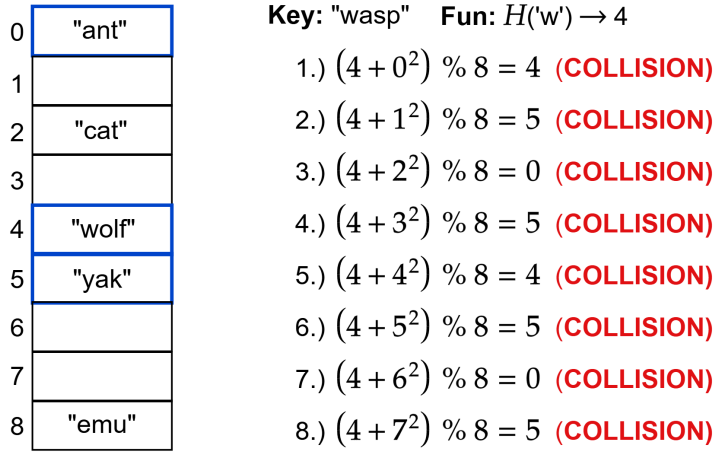


Figure 2.11: On the left is an existing hash table of 8 elements. We attempt to insert a new key, 'wasp', which hashes to index 4; Though, 4 in occupied. The algorithm continues with $(4+1^2) \ \% \ 8 = 5$, which is also occupied. After some probing, it appears only indices $4, 5, 0$ are appearing, from which are all occupied. The algorithm terminates at its n-th attempt with $(4 + 7^2) \ \% \ 8 = 5$. No spaces were found. One could imagine that if the table were larger or $0, 4, 5$ were free, the algorithm would have had better success.

Definition 5.6: Double Hashing

Double hashing resolves collisions by using two hash functions; Given, $H_1(x)$ and $H_2(x)$ uniform hashing functions, we define the probe sequence $h(x, k)$ as:

$$h(x, k) := \big(H_1(x) + k \cdot H_2(x)\big) \ \% \ n,$$

Where k is the collision count, and n is the hash table size. $H_2(x)$ is chosen such that:

- The hash satisfies $0 < H_2(x) < n$ (i.e., The result is likely taken by modulo n).
- It is pair-wise independent from $H_1(x)$ (i.e., not a transformation of/related to $H_1(x)$).
- Computationally inexpensive to evaluate.
- All outputs of $H_2(x)$ are relatively prime to n (does not share any common factors other than 1), ensuring all entries are probed.

We elaborate on the need for relatively prime numbers:

Theorem 5.1: Probing Period

A **period** defines the number of unique elements before the sequence begins to repeat. This cycle length is defined as the ratio:

$$\frac{n}{\gcd(n, H_2(x))}$$

Where n is the hash table size, and $H_2(x)$ is each hash output on an arbitrary x input. We ideally want $\gcd(n, H_2(x)) = 1$ for each x to achieve a full period of n. Hence, if n is a power of 2, $H_2(x)$ may uniformly provide odd numbers; Otherwise, for that particular key, it will only partially probe the table.

Without too much number theory, we attempt to intuitively understand the theorem:

Proof 5.1: Length of Probing Period

In terms of modulo, n defines a cycle of n elements, $0, 1, \ldots, n-1$; Each element is called a **residue class** (i.e., all possible remainders). E.g., 8 has residue classes 0–7 . Given a finite set of integers $\mathcal{H}$ (i.e., our hash function), all $\mathcal{H}_i$ need be co-prime to n to exhaust all residue classes. We exclude all $\mathcal{H}_i \geq n$, as n's cycle is definitively over (also by Definition 5.6). E.g., $1 \ \% \ 8 = 1$, $9 \ \% \ 8 = 1$. We pick a fixed-hash $h := \mathcal{H}_i$ such that $\gcd(n, h) = d > 1$. Recall that we calculate $(k \cdot h) \ \% \ n$ for each k collision. Hence if h and n factors intersect at d, then the $k = n/d_{th}$ collision will produce a multiple of n, terminating prematurely at 0.

■

Let's try an example to see how this works:

Example 5.2: Double Hashing without co-primes

Consider a $H_1(x)$ function which always causes collisions, forcing the use the $H_2(x)$ function:

$$H_1(x) := x^0 \quad H_2(x) := x^2 \ \% \ (n-1)$$

Giving us the full function:

$$h(x,k) := \big(x^0 + k \cdot \big(x^2 \ \% \ (n-1)\big)\big) \ \% \ n$$

Observe when $n = 12$, with key $x = 3$, while increasing k collisions:

$k \cdot (3^2 \ \% \ 11) \ \% \ 12$	$h(x,k)$
$0 \cdot 9 \ \% \ 12$	0
$1 \cdot 9 \ \% \ 12$	9
$2 \cdot 9 \ \% \ 12$	6
$3 \cdot 9 \ \% \ 12$	3
$4 \cdot 9 \ \% \ 12$	0
$5 \cdot 9 \ \% \ 12$	9
$6 \cdot 9 \ \% \ 12$	6
$7 \cdot 9 \ \% \ 12$	3
$8 \cdot 9 \ \% \ 12$	0
$9 \cdot 9 \ \% \ 12$	9
$10 \cdot 9 \ \% \ 12$	6

Since $\gcd(12, 9) = 3$, the probing period is $12/3 = 4$, only touching indices $\{0, 3, 9, 6\}$.

■

Searching: Insertion & Deletion

Things become trickier with a populated hash table with previous deletions:

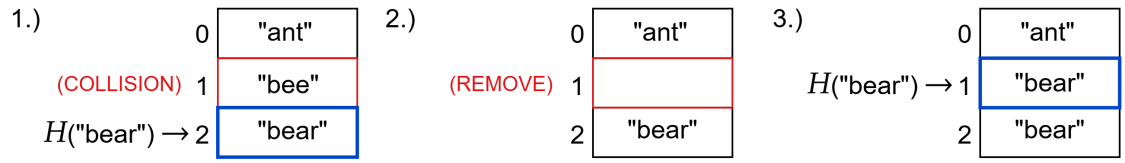


Figure 2.12: Demonstrates complications when inserting into a table blindly. 3.) niaively inserts "bear" at index 1 despite it already existing in the table. This is caused by a previous collision (1) and removal of the collider (2).

Theorem 5.2: Safe Insertion

To safely insert a key into a hash table, we create three distinctions for each cell:

- **Empty:** The index is empty, and a key can be inserted.
- **Occupied:** The index is occupied (blocked) by another key.
- **Deleted:** This index was previously occupied but free.

We may proceed as normal for events Empty and Occupied, but Deleted requires special handling; First, take note of the first **deleted index** then probe the table:

- **Empty:** If an empty index is found, insert at the first deleted index.
- **Occupied/Deleted:** Continue probing.
- **Duplicate** If the same key is found, insert any data, otherwise terminate.

Insertion at the first deleted index is safe, as if the key already existed in the table, it would have been inserted at any found empty index.

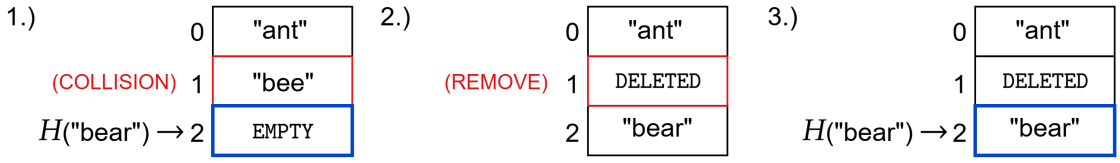


Figure 2.13: Revisiting Figure 2.12 at (3) we continue to probe after finding a deleted index. This leads us to find the already inserted key, "bear", at index 2.

Separate Chaining & Linked Lists

A problem we have with open addressing is that it requires a lot of array space; If we run out of space, we have to resize the table, **which is costly**:

Theorem 5.3: Resizing a Hash Table

Resizing a hash table requires rehashing all keys, which is $O(n)$, where n is the number of elements in the table, excluding the computation cost of hashing each key again.

This brings us to the motivation for our second method of collision resolution:

Definition 5.7: Separate Chaining

Separate chaining is a collision resolution method where each index contains a **bucket**, which contains all keys that hash to such index. This saves contiguous memory space, as the array can stay a fixed size while holding object references living in the heap.

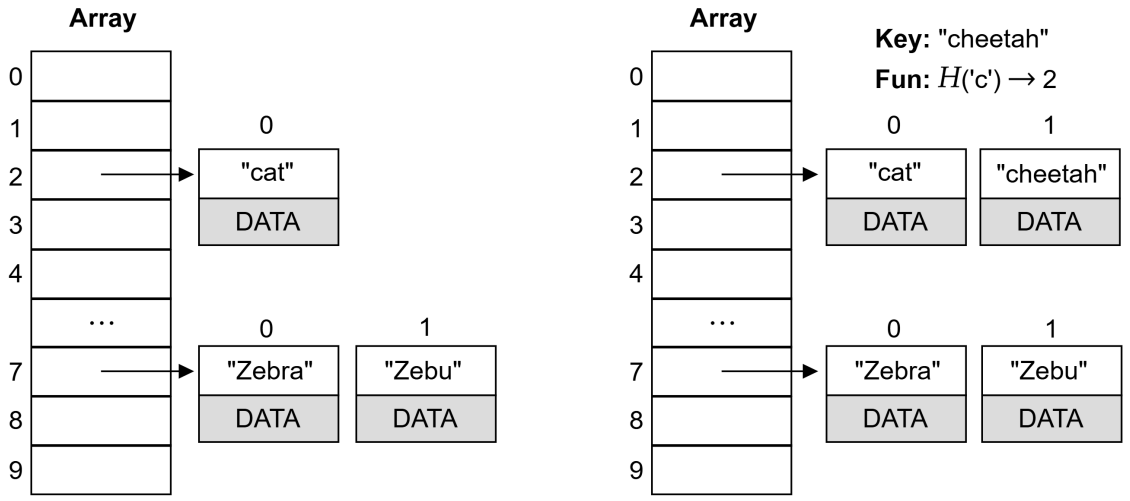


Figure 2.14: The left an array which points to another array (the bucket). The right shows the insertion of "cheetah" into the table, indexing at 2, adding to the end of the bucket. Here it's unclear how the data for each element is stored (perhaps a nested array). Additionally, the **same problem** of resizing occurs, as if a bucket is another array, we still have to resize it (there is no need to rehash the keys in buckets).

Definition 5.8: Resizing an Array

Many languages provide a method for resizing an array; However, this is costly as perhaps the next contiguous memory cell for the array is not available in memory. Hence, a new memory region is found and all elements are copied to a new array of larger size, typically $O(n)$, where n is the number of elements in the array.

Tip: In languages like Java or GO, resizing an array (ArrayList or Slice) is done by creating a new array of double the size and copying all elements over.

We introduce a new data structure to solve this problem:

Definition 5.9: Linked List

A **linked list** is a data structure consisting of single objects called **nodes**. Nodes are *linked* together via a reference to the next node in the list. This means there are no indices, and the next node can live anywhere in memory. The first element is called the **head**, and the last element is called the **tail**.

Each node at the very least contains a **value** and a **next** pointer to the next node in the list. Since a node is an object, an indefinite number of fields can be added to its class definition.

Definition 5.10: Common Types of Linked Lists

- **Singly Linked List:** Each node contains a reference to the next node.
- **Doubly Linked List:** Each node contains a reference to both the next and previous nodes.
- **Circular Linked List:** The last node points back to the first node, creating a cycle.

Time Complexity: Search is $O(n)$, as we may need to traverse the entire list.

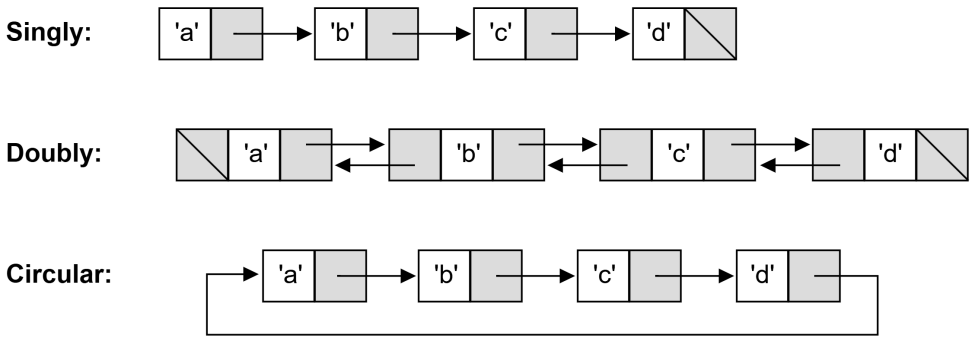


Figure 2.15: A visual representation of a singly, doubly, and circular linked lists.

Theorem 5.4: Open Addressing vs. Separate Chaining

If updates to the table are **rare** (rehashing on resize), choose open addressing. If updates are **frequent**, consider separate chaining.

It's worth noting that, separate chaining with linked lists requires **more memory** overhead for each node object.

We revisit Figure 2.14 with a linked list implementation:

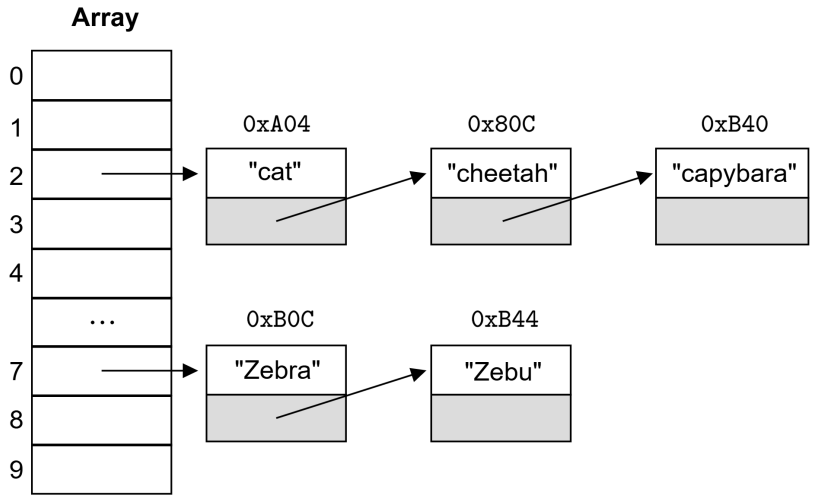


Figure 2.16: Here we see an array with two buckets, each bucket is a linked list. Each node points to the next node in the list. In particular, each node lives in at an ambiguous memory location.

Definition 5.11: Insertion & Deletion in Linked Lists

Inserting or deleting relies on shifting pointers around. To make sure references aren't lost to the linked list, we always point to a **dummy head** node, which holds no data and always points to the first actual node.

Say we have two nodes, A and B sequentially in a singly linked list referenced by `my_llist` (points to dummy head). it has one dot operator, `my_llist.next` (the next node in the list).

- **Inserting:** We attempt to add a new node C,
  - **At the head:** Set `my_llist.next = C` and `C.next = A`, $O(1)$.
  - **At the tail:** Traverse the list until via `my_llist.next` until a null `.next` is found, set the previous node's `.next` to C and `C.next = null`, $O(n)$.
  - **In the middle:** Traverse the list until node A is found, set `C.next = A.next`, then `A.next = C.next`, $O(n)$.
- **Deleting:** Now we have three sequential nodes, A, B, and C.
  - **The head:** Set `my_llist.next = A.next`, discards references to A, $O(1)$.
  - **The tail:** Traverse and set `B.next = null`, $O(n)$ (discards C).
  - **In the middle:** Traverse and set `A.next = B.next`, $O(n)$ (discards B).

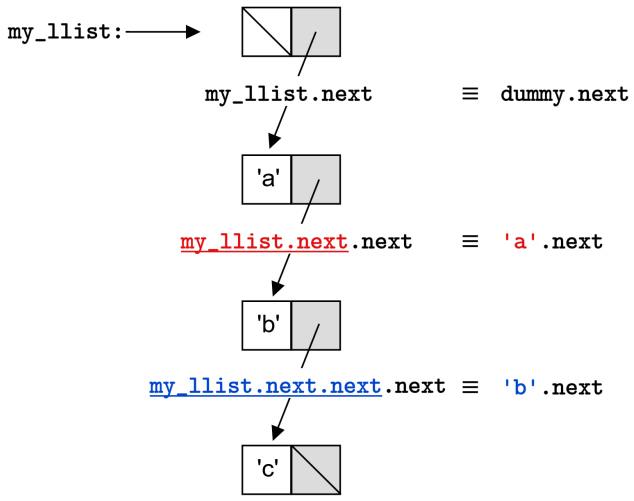
Visualizations on the next page.

Example 5.3: Insertion & Deletion in Linked Lists – Corollary

If we know the position of the node to be inserted or deleted, we can directly access it via a pointer. Revisiting Definition 5.11, we can insert or delete in constant time $O(1)$, instead of traversing the list. To demonstrate deletion, we have three sequential nodes, A, B, and C:

- **The head:** Set `my_llist.next = my_llist.next.next`, discards references to A.
- **The tail:** Set `my_llist.next.next.next = null`, discards C.
- **In the middle:** Set `my_llist.next.next = my_llist.next.next.next`, discards B.

This is a bit hard to read, we visualize it as follows:



In other words, `my_llist` is the dummy head node, `my_llist.next` returns the dummy node's `next` pointer (an address). If we *act* on such address, we access the address's fields. Hence, `my_llist.next.next` ≡ `A.next`. Again, `A.next` ≡ `B` (the address). So,

$$\begin{aligned} \texttt{my\_llist.next.next.next} &\equiv \texttt{A.next.next} \\ &\equiv \texttt{B.next} \\ &\equiv \texttt{C} \quad \text{(the address)} \end{aligned}$$

Still, it's not advisable to code like this, primarily because of readability. ■

Load Factor & Performance Metrics

We want to be pre-emptive at avoiding collisions and resizing the table. The following measurement tracks this:

Definition 5.12: Load Factor

The **load factor** α of a hash table is defined as the ratio of the number of elements n to the size of the table m:

$$\alpha := \frac{n}{m}$$

- **Open Addressing:** $\alpha < 0.5$.
- **Separate Chaining:** $\alpha < 1$.

Once α exceeds the optimal threshold, we should **consider resizing** the table. Upon good load conditions, we generally expect $\Theta(1)$ and $\Theta(1 + \alpha)$ time complexity for insertion and deletion, respectively with open addressing and separate chaining.

Method	Insertion	Deletion
Open Addressing	$\Theta(1)$ average, $O(n)$ worst	$\Theta(1)$ average, $O(n)$ worst
Separate Chaining	$\Theta(1 + \alpha)$ average, $O(n)$ worst	$\Theta(1 + \alpha)$ average, $O(n)$ worst

Table 2.3: Time-complexity comparison of insertion and deletion in open addressing vs. separate chaining (where α is the load factor).

Operation	**Dynamic Array**	**Singly Linked List**
Insert at head	$O(n)$ (shift all elements)	$O(1)$
Insert at tail	$O(1)$	$O(n)$
Insert in middle	$O(n)$	$O(n)$
Delete at head	$O(n)$	$O(1)$
Delete at tail	$O(1)$	$O(n)$
Delete in middle	$O(n)$	$O(n)$
Search for element	$O(n)$	$O(n)$
Random access	$O(1)$	$O(n)$ (traversal)

Table 2.4: Inserting or deleting elements in dynamic arrays (growing) versus singly linked lists. In particular, maintaining order within a dynamic array forces shifting of elements upon insertion or deletion.

— 3 —

Computational Algorithms

3.1 Information Theory

Defining Information

The following sections **heavily** reference Chris Terman's "Computation Structures" from the MIT OpenCourseWare, and Victor Shoup's "A Computational Introduction to Number Theory and Algebra" [2, 1].

Definition 1.1: Information

Information measures the amount of uncertainty about a given fact provided some data.

Example 1.1: Playing Deck of Cards

Given a 52-card deck, a card is drawn at random. One of the following data points is revealed:

a) The card is a heart (13 possibilities).

b) The card is not the Ace of Spades (51 possibilities).

c) The card is the "Suicide King," i.e., King of Hearts (1 possibility). ■

Definition 1.2: Quantifying Information

Given a discrete (finite) random variable X with n possible outcomes $(x_1, x_2, \ldots, x_n)$ and a probability $P(X) = p_i$ for each outcome x_i, the **information content** of X is defined as:

$$I(X_i) := \log_2 \left(\frac{1}{p_i} \right)$$

Where $1/p_i$ is the probability of x_i, while Log base 2 measures how many bits (0 or 1) are needed to represent the outcome.

Example 1.2: Generalizing Information Content

A heart drawn from a 52-card deck may be represented as follows:

$$I(\text{heart}) = \log_2\left(\frac{1}{13/52}\right) \approx 2 \text{ bits}$$

More generally, we may redefine the information content as follows:

$$I(\text{data}) = \log_2\left(\frac{1}{M \cdot (1/N)}\right) = \log_2\left(\frac{N}{M}\right)$$

Where N is the total number of possible outcomes (e.g., 52 cards in a deck), and M is the number of outcomes that match the data (e.g., 13 hearts in a deck). Hence, $M \cdot (1/N)$ is the amount of information received from the data. Consider two more examples:

- **Information in one coin flip:** $\log_2(2/1) = 1$ bit $(N := 2, M := 1)$.
- **Rolling 2 dice:** $\log_2(36/1) \approx 5.17$ or 6 bits $(N := 36, M := 1)$.

■

Definition 1.3: Entropy

The **entropy** of a discrete random variable X is the average amount of information contained in all possible outcomes of X. It is defined as:

$$H(X) := E(I(X)) = \sum_{i=1}^{N} p_i \cdot \log_2\left(\frac{1}{p_i}\right)$$

Where function E is the expected value (i.e., average) of the information content $I(X)$ across all outcomes of X. This conveys how many bits b are needed to represent the outcomes of X:

- $b < H(X)$: Information is lost (i.e., not all outcomes can be represented).
- $b = H(X)$: An optimal representation.
- $b > H(X)$: Redundancy (i.e., not an efficient use of resources.).

Tip: For refreshers on $\sum$ consider our other text: Concise Works: Discrete Math.

Example 1.3: The Entropy of Four Choices

Consider a discrete random variable and it's possible outcomes $X := \{A, B, C, D\}$:

choice_i	p_i	$\log_2(1/p_i)$
A	$1/3$	$1.58\,\text{bits}$
B	$1/2$	$1\,\text{bit}$
C	$1/12$	$3.58\,\text{bits}$
D	$1/12$	$3.58\,\text{bits}$

Hence, the entropy of X is:

$$\begin{aligned} H(X) := \sum_{i=1}^{4} p_i \cdot \log_2\left(\frac{1}{p_i}\right) = &\left(\frac{1}{3} \cdot 1.58\right) + \\ &\left(\frac{1}{2} \cdot 1\right) + \\ &\left(\frac{1}{12} \cdot 3.58\right) + \\ &\left(\frac{1}{12} \cdot 3.58\right) + \\ &\approx 1.626\,\text{bits} \end{aligned}$$

The entropy of X is approximately 1.626 bits, meaning that on average, we should be able to represent the outcomes of X using less than 2 bits per outcome.

■

Let's discuss how we might go about representing our outcomes:

Definition 1.4: Encoding

An **encoding** is an unambiguous mapping from a set of symbols to a set of bit strings:

- **Fixed-length encoding:** Uses a fixed number of bits to represent each symbol.
- **Variable-length encoding:** Uses a different number of bits for each symbol.

Example 1.4: Encoding Four Symbols

Consider the four symbols A, B, C, D and each possible encoding for them:

	Encoding for each symbol				Encoding for,
	A	B	C	D	"ABBA"
1.)	00	01	10	11	00 01 01 00
2.)	01	1	000	001	01 1 1 01
3.)	0	1	10	11	0 1 1 0

(1) Is a fixed-length encoding, (2) is a variable-length encoding, and (3) is also a variable-length encoding and uses fewer bits; **However**, it is ambiguous. Depending on how our program reads the string, it may group and misinterpret the bits.

E.g., (3) could be, "0 11 0" (A D A) or " 0 1 10" (A B C). Hence, an invalid encoding. ■

Theorem 1.1: Binary Tree Encoding

Binary trees may represent unambiguous encodings, where each symbol is a leaf node, and each edge represents the next bit. Since each path is unique, the encoding is unambiguous.

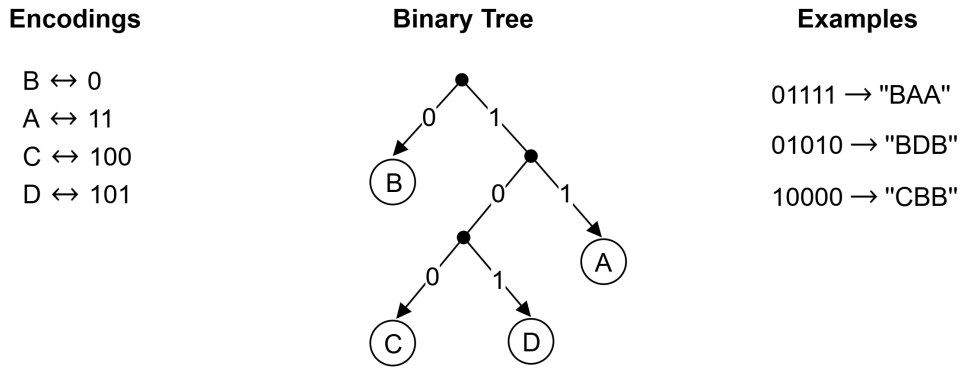


Figure 3.1: Encodings start at the root, each edge taken writes the next bit.

Theorem 1.2: Binary Tree Encoding – Fixed-Length

A fixed-length encoding is optimal when the number of symbols n bear an equal probability of occurrence. In a binary tree encoding, all leaves have the same depth.

I.e., for n symbols, each have a probability of $1/n$, hence an entropy of:

$$H(X) = \sum_{i=1}^{n} \left(\frac{1}{n} \cdot \log_2 \left(\frac{1}{1/n} \right) \right) = \log_2(n)$$

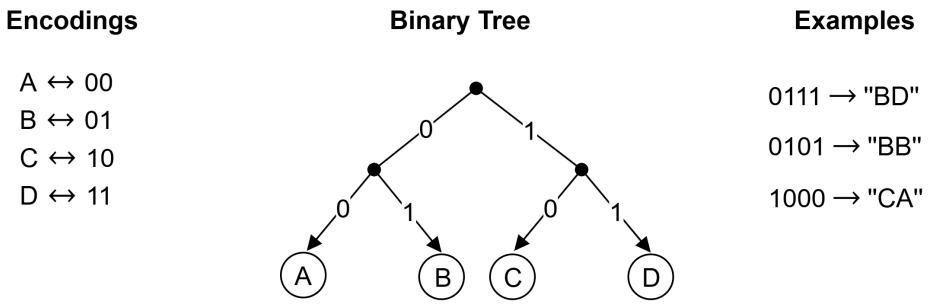


Figure 3.2: A fixed-length encoding for four symbols, represented as a binary tree.

Theorem 1.3: Choosing Variable-Length Encoding

If a symbol A has the high probability of occurrence, it should be represented with the shortest possible bit string. Vice versa, if symbol B has the low probability, then it should receive the longest bit string.

E.g., in Figure (3.1), the symbol B may be assumed to have the highest probability of occurrence, with C and D having the lowest.

Theorem 1.4: Variable vs. Fixed-Length Encoding

Though we would like to use a variable-length encoding in theory, a fixed-length encoding for complex data structures is provides simplicity and scalability.

Moving forward we focus on such fixed-length encodings.

3.2 Number Base System Encodings

We now explore how to represent our base two or any other base number in a fixed-length encoding.

Definition 2.1: Number Base Fixed-length Encoding

Each memory cell in the computers stack is an integer value represented in a fixed base, typically $B = 2$, meaning **binary**. Where each digit is less than the base B. We represent integers in memory as:

$$a = \sum_{i=0}^{k-1} a_i B^i$$

Where a_i represents the individual digits, and B is the base. For large integers, computations may require manipulating several memory cells to store the full number.

Example 2.1: Binary & Decimal Representations

Consider the integer, $a = 13$, in base, $B := 2$ **(binary)**, using Definition 2.1:

$$a = \underbrace{(1 \cdot 2^3) + (1 \cdot 2^2) + (0 \cdot 2^1) + (1 \cdot 2^0)}_{\text{binary: } 1101} = 8 + 4 + 0 + 1 = 13$$

Here, the coefficients $a_3 = 1$, $a_2 = 1$, $a_1 = 0$, and $a_0 = 1$ correspond to the binary digits of 13, where each power of 2 represents the binary place value. Similarly, if we want to represent, $a = 45$, in base, $B := 10$ **(decimal)**:

$$a = 4 \cdot 10^1 + 5 \cdot 10^0 = 40 + 5 = 45$$

In this case, the coefficients $a_1 = 4$ and $a_0 = 5$ correspond to the decimal digits of 45. ■

We continue to common bases that one may encounter in computing and mathematics:

Definition 2.2: Hexadecimal

Hexadecimal base $B = 16$, using digits 0-9 and the letters A-F, where:

$$A = 10,\ B = 11,\ C = 12,\ D = 13,\ E = 14, \text{ and } F = 15.$$

Hexadecimal is commonly used in computing due to its compact representation of binary data. E.g., A **byte** (8 bits) can be represented as two hexadecimal digits, simplifying the display of binary data.

Theorem 2.1: Base $2 \leftrightarrow 16$ Conversion

Let bases $B := 2$ (binary) and $H := 16$ (hexadecimal). At a high-level:

Binary to Hexadecimal:

1. Group B digits in sets of 4, right to left. **Pad** leftmost group with 0's if necessary for a full group.
2. Compute each group, replacing the result with their H digit.
3. Finally, combine each H group.

Hexadecimal to Binary:

1. Convert each H digit into a 4 bit B group.
2. Finally, combine all B groups.

Additionally, we may also trim any leading 0's.

Example 2.2: Base $2 \leftrightarrow 16$ Conversion

- Binary to Hexadecimal:

$$101101111010_2 \quad \Rightarrow \quad \text{Group as } (\underbrace{[1011]}_{11} \ \underbrace{[0111]}_{7} \ \underbrace{[1010]}_{10}) \quad \Rightarrow \quad B7A_{16}$$

- Hexadecimal to Binary:

$$3F5_{16} \quad \Rightarrow \quad [0011]\ [1111]\ [0101]_2 \Rightarrow \quad 1111110101_2$$

■

The following definition is for completeness: applications of such a base are currently seldom.

Definition 2.3: Unary

Unary, base $B = 1$. A system where each number is represented by a sequence of the same symbol. This system is more theoretical than practical given today's systems.

Given a toy unary system, we may represent numbers with a single symbol "I": $5 =$ `IIIII` or $2 =$ `II`. The absence of symbols may represent 0.

Definition 2.4: Most & Least Significant Bit

In a binary number, the **most significant bit (MSB)** is the leftmost bit. The **least significant bit (LSB)** is the rightmost bit.

E.g., In the byte (8 bits), $[1111\ 1110]_2$, the MSB = 1 and the LSB = 0.

Theorem 2.2: Adding Binary

We may use the add and carry method alike decimal addition:

- $0 + 0 = 0$
- $1 + 0 = 1$
- $0 + 1 = 1$
- $1 + 1 = 0$ (add 1 to the next digit (left))

We call the last step a **carry**, as we carry our overflow to the next digit.

Example 2.3: Binary Addition

Adding $0010\ 0011\ 0100_2$ and 0100_2:

$$\begin{array}{r} \overset{1}{} \\ 1\,1000\,0100 \\ +\,0\,0001\,0100 \\ \hline 1\,1001\,1000 \end{array}$$

Where $[1\ 1000\ 0100]_2 + [0\ 0001\ 0100]_2 = [1\ 1001\ 1000]_2$. ■

Definition 2.5: Signed Binary Numbers - Two's Complement

In a **two's complement system**, an n-bit signed (positive or negative) binary number can represent values in the range $[-2^{n-1}, 2^{n-1} - 1]$. Then by most significant bit (MSB):

- If MSB is 0, the number is positive;
- If MSB is 1, the number is negative.

Conversion to Two's Complement :

1. Take an unsigned binary number and invert all bits, turning 0's to 1's and 1's to 0's.
2. Finally add 1 to the least significant bit.

Example 2.4: Two's Complement Conversion

Converting -5 into a 4-bit two's complement:

$$\begin{aligned} 5 \to 0101 \quad & \text{(binary for 5)} \\ 1010 \quad & \text{(inverted)} \\ 1011 \quad & \text{(add 1)} \end{aligned}$$

Thus, -5 is represented as 1011 in 4-bits under two's complement. ■

3.3 Computing Large Numbers

Before moving forward, we must understand on paper how computation works at a mathematical level. Then we will re-visit encodings at the bit & byte level of abstraction.

Definition 3.1: Wordsize

Our machine has a fixed **wordsize**, which is how much each memory cell can hold. Systems like 32-bit or 64-bit can hold 2^{32} (≈ 4.3 billion) or 2^{64} (≈ 18.4 quintillion) bits respectively.

We say the ALU can performs arithmetic operations at $O(1)$ time, within wordsize. Operations beyond this size we deem **large numbers**.

The game we play in the following algorithms is to compute large integers without exceeding wordsize. Moving forward, we assume our machine is a typical 64-bit system.

Function 3.1: Length of digits - $\|a\|$

We will use the notation $\|a\|$ to denote the number of digits in the integer a. For example, $\|123\| = 3$ and $\|0\| = 1$.

Our first hurdle is long division as , which will set up long addition and subtraction for success. **Scenario - Grade School Long Division:** Goes as follows, take $\frac{a}{b}$. Find how times b fits into a evenly, q times. Then $a - bq$ is our remainder r.

Definition 3.2: Computer Integer Division

Our ALU only returns the quotient after division. We denote the quotient as $\lfloor a/b \rfloor : a, b \in \mathbb{Z}$.

Example 3.1: Long Division

let $a = \{12, 5, 17, 40, 89\}$, $b = \{4, 2, 3, 9, 10\}$ respectively, and base $B = 10$,

$$\begin{array}{lllll} (1.)\ \begin{array}{r} 3 \\ 4\overline{)12} \\ 12 \\ \hline 0 \end{array} & (2.)\ \begin{array}{r} 2 \\ 2\overline{)5} \\ 4 \\ \hline 1 \end{array} & (3.)\ \begin{array}{r} 5 \\ 3\overline{)17} \\ 15 \\ \hline 2 \end{array} & (4.)\ \begin{array}{r} 4 \\ 9\overline{)40} \\ 36 \\ \hline 4 \end{array} & (5.)\ \begin{array}{r} 8 \\ 10\overline{)89} \\ 80 \\ \hline 9 \end{array} \end{array}$$

Take (3.), $a = 17$, $b = 3$: 3 fits into 17 five times, which is 15. 17 take away 15 is 2, our remainder. We create an algorithm to compute this process.

Key Observation: Consider the following powers of 2 of form $x = 2^n + s$, where $x, n, s \in \mathbb{Z}$:

$$\begin{aligned} 3 = 2 + 1 &= 0000\ 0011_2 \quad (1) \\ 6 = 4 + 2 &= 0000\ 0110_2 \quad (2) \\ 12 = 8 + 4 &= 0000\ 1100_2 \quad (3) \\ 24 = 16 + 8 &= 0001\ 1000_2 \quad (4) \\ 48 = 32 + 16 &= 0011\ 0000_2 \quad (5) \\ 96 = 64 + 32 &= 0110\ 0000_2 \quad (6) \\ 192 = 128 + 64 &= 1100\ 0000_2 \quad (7) \end{aligned}$$

Notice that as we increase the power of 2, the number of bits shift left towards a higher-order bit. Now, instead of calculating powers of 2, we shift bits left or right, to yield instantaneous results. ■

Theorem 3.1: Binary Bit Shifting (Powers of 2)

Let x be a binary unsigned integer. Where "$\ll$" and "$\gg$" are left and right bit shifts:
Left Shift by k bits: $x \ll k := x \cdot 2^k$
Right Shift by k bits: $x \gg k := \lfloor x/2^k \rfloor$
Remainder: bits pushed out after right shift(s).

Example 3.2: Bit Shifting in Base 2

Observe, $16 = 10000$ (4 zeros), $8 = 1000$ (3 zeros), we shift by 4 and 3 respectively:

- Instead of $3 \cdot 16$ in base 10, we can $3 \ll 4 = 48$, as $3 \cdot 2^4 = 48$.
- Conversely, Instead of $48/16$ in base 10, $48 \gg 4 = 3$, as $\lfloor 48/2^4 \rfloor = 3$.
- Catching the remainder: say we have $37/8$ base 10, then,

$$37 = 100101_2 \quad \text{and} \quad 8 = 1000_2 \text{ then } 37 \gg 3 = 4 \text{ remainder } 5,$$

as $[100101] \gg 3 = [000100]101$, where 101_2 is our remainder 5_{10}.

■

Function 3.2: Division with Remainder in Binary (Outline) - *QuoRem()*

For binary integers, let dividend $a = (a_{k-1} \cdots a_0)_2$ and divisor $b = (b_{\ell-1} \cdots b_0)_2$ be unsigned, with $k \geq 1$, $\ell \geq 1$, ensuring $0 \leq b \leq a$, and $b_{\ell-1} \neq 0$, ensuring $b > 0$.

We compute q and r such that, $a = bq + r$ and $0 \leq r < b$. Assume $k \geq \ell$; otherwise, $a < b$. We set $q \leftarrow 0$ and $r \leftarrow a$. Then quotient $q = (q_{m-1} \cdots q_0)_2$ where $m := k - \ell + 1$.

Input: a, b (binary integers)
Output: q, r (quotient and remainder in binary)

1 **Function** *QuoRem*(a, b)**:**
2 $\quad r \leftarrow a$;
3 $\quad q \leftarrow \{0_{m-1} \cdots 0\}$;
4 $\quad$ **for** $i \leftarrow \|a\| - \|b\| - 1$ ***down to*** 0 **do**
5 $\quad\quad q_i \leftarrow \left\lfloor \dfrac{r}{b \ll i} \right\rfloor$;
6 $\quad\quad r \leftarrow r - (q_i \cdot (b \ll i))$;

Time Complexity: $O(\|a\|(\|a\| - \|b\|))$. In short, line 5 we preform division on $\|a\|$ bits of decreasing size. Though **not totally necessary**, For more detail visit https://shoup.net/ntb/ntb-v2.pdf on page 60. General n cases can be found in Theorem (3.2).

Note: The above function is laid out in a more general form in its original text. But the above suffices for our purposes. **Remember:** The ALU can handle basic arithmetic operations in $O(1)$ time, as long as the numbers are small enough to fit in a single word.

Example 3.3: QuoRem - Quotient and Remainder

Example Let $a = 47_{10} = 101111_2$ and $b = 5_{10} = 101_2$, we run *QuoRem(a, b)*. We summarize the above example as, "How many times does 101_2 fit into 101111_2?"

For q (quotient) and r (remainder):

1. Does $5 \ll 3$ fit into 101111_2?
   - **It fits!** As $[0000\ 0101]_2 \ll 3 = [0010\ 1000]_2$, is less than or equal to 101111_2.
   - **So:** $q_1 = 1000_2$ (3 zero shifts), $r_1 = (101111_2 - 101000_2) = 0111_2$.
2. Does $5 \ll 2$ fit into 0111_2?
   - **No fits!** As $[0000\ 0101]_2 \ll 2 = [0001\ 0100]_2$, is greater than 0111_2.
   - **So:** nothing is changed, $q_2 = 1000_2$, $r_2 = 0111_2$.
3. Does $5 \ll 1$ fit into 0111_2?
   - **No fits!** As $[0000\ 0101]_2 \ll 1 = [0000\ 1010]_2$, is greater than 0111_2,
   - **So:** nothing is changed, $q_2 = 1000_2$, $r_2 = 0111_2$.
4. Does $5 \ll 0$ fit into 0111_2?
   - **It fits!** As $[0000\ 0101]_2 \ll 0 = [0000\ 0101]_2$, is less than or equal to 0111_2.
   - **So:** $q_2 = 0001_2$ (0 zero shifts), then $q_1 + q_2 = 1001_2$, $r_2 = 0111_2 - 0101 = 0010_2$.
5. **Return:** $q = 1001_2 = 9_{10}$, $r = 0010 = 2_{10}$

We may verify this in decimal, $47 = 5 \cdot 9 + 2$ (dividend = divisor · quotient + remainder). ■

Note: The above example touches on the "Division Algorithm," which is not technically an algorithm, but rather a theorem. To learn more, consider: Concise Works: Number Theory

Scenario - Grade School Long Addition: We craft an algorithm for grade school long addition, which goes as follows:

$$\begin{array}{r} \overset{1}{2}5\,\overset{1}{3}08 \\ +\,39\,406 \\ \hline 64\,714 \end{array}$$

Where adding, $25,308 + 39,406 = 64,714$. We create an algorithm to compute this in the following function.

Function 3.3: Addition of Binary Integers - *Add()*

Let $a = (a_{k-1} \cdots a_0)_2$ and $b = (b_{\ell-1} \cdots b_0)_2$ be unsigned binary integers, where $k \geq \ell \geq 1$. We compute $c := a + b$ where the result $c = (c_k c_{k-1} \cdots c_0)_2$ is of length $k + 1$, assuming $k \geq \ell$. If $k < \ell$, swap a and b. This algorithm computes the binary representation of $a + b$.

Input: a, b (binary integers)
Output: $c = (c_k \cdots c_0)_2$ (sum of $a + b$)

```
1  Function Add(a, b):
2      carry ← 0;
3      for i ← 0 to ℓ − 1 do
4          tmp ← a_i + b_i + carry;
5          (carry, c_i) ← QuoRem(tmp, 2);
6      for i ← ℓ to k − 1 do
7          tmp ← a_i + carry;
8          (carry, c_i) ← QuoRem(tmp, 2);
9      c_k ← carry;
10     return c = (c_k ··· c_0)_2;
```

Note: $0 \leq carry \leq 1$ and $0 \leq tmp \leq 3$.
Time Complexity: $O(\max(\|a\|, \|b\|))$, as we iterate at most the length of the largest input.
Space Complexity: $O(\|a\| + \|b\|)$, though $c = k + 1$, constants are negligible as $k, \ell \to \infty$.

For subtracting, $5,308 - 3,406 = 1,904$, where we borrow 10 from the 5 to make 13:

$$\begin{array}{r} {\not{5}}^{4}\,{}^{10}3\,08 \\ -\,3\;406 \\ \hline 1\;904 \end{array}$$

Function 3.4: Subtraction of Binary Integers - *Subtract()*

Let $a = (a_{k-1} \cdots a_0)_2$ and $b = (b_{\ell-1} \cdots b_0)_2$ be unsigned binary integers, where $k \geq \ell \geq 1$ and $a \geq b$. We compute $c := a - b$ where the result $c = (c_{k-1} \cdots c_0)_2$ is of length k, assuming $a \geq b$. If $a < b$, swap a and b and set a negative flag to indicate the result is negative. This algorithm computes the binary representation of $a - b$.

Input: a, b (binary integers)
Output: $c = (c_{k-1} \cdots c_0)_2$ (difference of $a - b$)

```
1  Function Subtract(a, b):
2      borrow ← 0;
3      for i ← 0 to ℓ − 1 do
4          tmp ← a_i − b_i − borrow;
5          if tmp < 0 then
6              borrow ← 1;
7              c_i ← tmp + 2;
8          else
9              borrow ← 0;
10             c_i ← tmp;
11     for i ← ℓ to k − 1 do
12         tmp ← a_i − borrow;
13         if tmp < 0 then
14             borrow ← 1;
15             c_i ← tmp + 2;
16         else
17             borrow ← 0;
18             c_i ← tmp;
19     return c = (c_{k−1} ··· c_0)_2;
```

Note: $0 \leq borrow \leq 1$. Subtraction may produce a borrow when $a_i < b_i$.
Time Complexity: $O(\max(\|a\|, \|b\|))$, iterating at most the length of the largest input.
Space Complexity: $O(\|a\| + \|b\|)$, as the length of c is at most k, with constants negligible as $k, \ell \to \infty$.

For multiplication, $24 \cdot 16 = 384$:

$$\begin{array}{r} {\scriptstyle 2} \\ 24 \\ \times\ 16 \\ \hline 144 \\ +240 \\ \hline 384 \end{array}$$

Where $6 \cdot 4 = 24$, we write the 4 and carry the 2. Then $6 \cdot 2 = 12$ plus the carried 2 is 14. Then we multiply the next digit, 1, we add a 0 below our 144, and repeat the process. Every new 10s place we add a 0. Then we add our two products to get 384.

We create an algorithm to compute this process in the following function:

Function 3.5: Multiplication of Base-B Integers - *Mul()*

Let $a = (a_{k-1} \cdots a_0)_B$ and $b = (b_{\ell-1} \cdots b_0)_B$ be unsigned integers, where $k \geq 1$ and $\ell \geq 1$. The product $c := a \cdot b$ is of the form $(c_{k+\ell-1} \cdots c_0)_B$, and may be computed in time $O(k\ell)$ as follows:

Input: a, b (base-B integers)
Output: $c = (c_{k+\ell-1} \cdots c_0)_B$ (product of $a \cdot b$)

```
1  Function Mul(a, b):
2      for i ← 0 to k + ℓ − 1 do
3          c_i ← 0;
4      for i ← 0 to k − 1 do
5          carry ← 0;
6          for j ← 0 to ℓ − 1 do
7              tmp ← a_i · b_j + c_{i+j} + carry;
8              (carry, c_{i+j}) ← QuoRem(tmp, B);
9          c_{i+ℓ} ← carry;
10     return c = (c_{k+ℓ−1} ··· c_0)_B;
```

Note: At every step, the value of *carry* lies between 0 and $B - 1$, and the value of *tmp* lies between 0 and $B^2 - 1$.
Time Complexity: $O(\|a\| \cdot \|b\|)$, since the algorithm involves $k \cdot \ell$ multiplications.
Space Complexity: $O(\|a\| + \|b\|)$, since we store the digits of a, b, and c.

Function 3.6: Decimal to Binary Conversion - *DecToBin()*

This function converts a decimal number n into its binary equivalent by repeatedly dividing the decimal number by 2 and recording the remainders.

Input: n (a decimal number)
Output: b (binary representation of n)

```
1 Function DecToBin(n):
2     b ← empty string;
3     while n > 0 do
4         r ← n mod 2;
5         n ← ⌊n/2⌋;
6         b ← r + b;
7     return b;
```

Time Complexity: $O(\log n)$, as the number of iterations is proportional to the number of bits in n.
Space Complexity: $O(n)$, storing our input n.

Example: Converting 89 to binary given the above function:

$$\begin{aligned}
89_{10} \div 2 = 44 \quad & \text{rem } 1, \longleftarrow \textbf{ LSB} \\
44_{10} \div 2 = 22 \quad & \text{rem } 0, \\
22_{10} \div 2 = 11 \quad & \text{rem } 0, \\
11_{10} \div 2 = 5 \quad & \text{rem } 1, \\
5_{10} \div 2 = 2 \quad & \text{rem } 1, \\
2_{10} \div 2 = 1 \quad & \text{rem } 0, \\
1_{10} \div 2 = 0 \quad & \text{rem } 1. \longleftarrow \textbf{ MSB}
\end{aligned}$$

Thus, $89_{10} = 1011001_2$.

Theorem 3.2: Time Complexity of Basic Arithmetic Operations

We generalize the time complexity to a and b as n-bit integers.

(i) **Addition & Subtraction:** $a \pm b$ in time $O(n)$.

(ii) **Multiplication:** $a \cdot b$ in time $O(n^2)$.

(iii) **Quotient Remainder** quotient $q := \left\lfloor \frac{a}{b} \right\rfloor : b \neq 0, a > b$; and remainder $r := a \mod b$ has time $O(n^2)$.

3.4 Computational Efficiency

Theorem 4.1: Binary Length of a Number - $\|a\|$

The binary length of an integer a_{10} in binary representation, is given by:

$$\|a\| := \begin{cases} \lfloor \log_2 |a| \rfloor + 1 & \text{if } a \neq 0, \\ 1 & \text{if } a = 0, \end{cases}$$

as $\lfloor \log_2 |a| \rfloor + 1$ correlates to the highest power of 2 required to represent a.

Example: Think about base 10 first. Let there be a 9 digit number $d = 684,301,739$. To reach 9 digits takes 10^8; The exponent plus 1 yields $\|d\|$. Hence, $\lfloor \log_{10} d \rfloor + 1$ is $\|d\|$.

Now, let there be a 7 digit binary number $b = 1001000$, which expanded is:

$$(1 \cdot 2^6) + (0 \cdot 2^5) + (0 \cdot 2^4) + (1 \cdot 2^3) + (0 \cdot 2^2) + (0 \cdot 2^1) + (0 \cdot 2^0) = 72,$$

Taking 6 powers of 2 to reach 72, we add 1 to get $\|b\| = 7$. Hence, $\|b\| = \lfloor \log_2 b \rfloor + 1$.
Additionally, if $a = 0_2$ then $\|a\| = 1$. as $a^0 = 1$.

Theorem 4.2: Splitting Higher and Lower Bits

Let a be a binary number with n bits. We can split a into two numbers A_1 and A_0 with $n/2$ bits each, representing the first and second halves respectively. Where:

$$A_1 := \frac{a}{2^{\lceil n/2 \rceil}} \quad \text{and} \quad A_0 := a \bmod 2^{\lceil n/2 \rceil}$$

Example: Let's start with base 10. To achieve $A_1 = 7455$ and $A_0 = 62,010$, for $a = 745,562,010$. we take the length $\|a\| := \lfloor \log_{10}(745,562,010) \rfloor + 1 = 9$, as $10^8 \leq 745,562,010 < 10^9$. Then:

$$A_1 = \frac{745,562,010}{10^{\lceil 9/2 \rceil}} = 7455, \quad \text{and} \quad A_0 = 745,562,010 \bmod 10^{\lceil 9/2 \rceil} = 62,010$$

as $10^5 \leq 62,010 < 10^6$. Likewise to finding the remainder in base 2, we can use the same bit shifting technique for base 10 (3.1). We see,

$$[745,562,010]_{10} \text{ right shift by } 5, [000,007,455]_{10} \; 62,010.$$

Hence, $62,010$ is pushed out, and our remainder. Then, we can apply the same technique to base 2. Let $a = 1111\ 1111\ 1001\ 1001_2$. We have $\|a\| := 16$, then:

$$A_1 = \frac{1111\ 1111\ 1001\ 1001_2}{2^{\lceil 16/2 \rceil}} = 1111\ 1111_2, \text{ and } A_0 = 1111\ 1111\ 1001\ 1001_2 \bmod 2^{\lceil 16/2 \rceil} = 1001\ 1001_2$$

Scenario - *Divide and Conquer Multiplication*: We are to compute,

$$A_1 2^{\lceil n/2 \rceil} + A_0 =: a \quad \times \quad b := B_1 2^{\lceil n/2 \rceil} + B_0.$$

Then we have,

$$\begin{aligned} a \cdot b &= (A_1 2^{\lceil n/2 \rceil} + A_0)(B_1 2^{\lceil n/2 \rceil} + B_0) \\ &= (A_1 2^{\lceil n/2 \rceil})(B_1 2^{\lceil n/2 \rceil}) + (A_1 2^{\lceil n/2 \rceil})B_0 + (B_1 2^{\lceil n/2 \rceil})A_0 + A_0 B_0 \\ &= (A_1 B_1)2^n + (A_1 B_0 + B_1 A_0)2^{\lceil n/2 \rceil} + A_0 B_0. \end{aligned}$$

We need to compute 4 products, (A_1B_1), $(A_1B_0), (B_1A_0)$, and (A_0B_0). We now attempt to solve them independently:

Function 4.1: Multiplication of n-bit Integers - *Multiply()*

Let a and b be n-bit integers of base 2. This algorithm recursively computes the product of a and b using a straightforward divide-and-conquer approach, without using Karatsuba's optimization.

Input: n, a, b (where a and b are n-bit integers)
Output: The product $a \times b$

```
1  Function Multiply(n, a, b):
2      if n < 2 then
3          return the result of grade-school multiplication for a × b;
4      else
5          A_1 ← a ÷ 2^{n/2};  A_0 ← a mod 2^{n/2};
6          B_1 ← b ÷ 2^{n/2};  B_0 ← b mod 2^{n/2};
7          p_1 ← Multiply(n/2, A_1, B_1);
8          p_2 ← Multiply(n/2, A_1, B_0);
9          p_3 ← Multiply(n/2, A_0, B_1);
10         p_4 ← Multiply(n/2, A_0, B_0);
11         return p_1 · 2^n + (p_2 + p_3) · 2^{n/2} + p_4;
```

Time Complexity: $O(n^2)$, as in our master method $T(n) = 4T(n/2) + O(n)$, Theorem (**??**).
Space Complexity: $O(n)$, storing $n + n$ bits for a and b, while we track $O(\log_2 n)$ depth in the recursion stack.

We appear to make no improvement, however there's a small trick to reduce the number of multiplications. We continue on the next page.

Observe our full term, $c := (A_1B_1)2^n + (A_1B_0 + B_1A_0)2^{\lceil n/2 \rceil} + A_0B_0$. Say we computed another term,

$$z := (A_1 + A_0)(B_1 + B_0) = (A_1B_1) + (A_1B_0) + (B_1A_0) + (A_0B_0).$$

Notice how z also contains (A_1B_1) and (A_0B_0), which are also in c. Say $m = (A_1B_0) + (B_1A_0)$. Let $x := (A_1B_1)$ and $y := (A_0B_0)$ then $z - x - y = m$. This reduces the number of multiplications to 3, as we only compute (A_1B_1), (A_0B_0) once, and then z.

We employ the above strategy, which is **Karatsuba's multiplication algorithm**:

Function 4.2: Karatsuba's Multiplication Algorithm - *KMul()*

Let a and b be n-bit integers of base 2. This algorithm recursively computes the product of a and b using a divide-and-conquer approach.

Input: n, a, b (where a and b are n-bit integers)
Output: The product $a \times b$

```
1  Function Multiply(n, a, b):
2      if n < 2 then
3          return the result of grade-school multiplication for a × b;
4      else
5          A_1 ← a ÷ 2^{n/2};  A_0 ← a mod 2^{n/2};
6          B_1 ← b ÷ 2^{n/2};  B_0 ← b mod 2^{n/2};
7          x ← Multiply(n/2, A_1, B_1);
8          y ← Multiply(n/2, A_0, B_0);
9          z ← Multiply(n/2, A_1 + A_0, B_1 + B_0);
10         return x · 2^n + (z − x − y) · 2^{n/2} + y;
```

Time Complexity: $O(n^{\log_2 3}) \approx O(n^{1.585})$, as in our master method $T(n) = 3T(n/2) + O(n)$, Theorem (**??**).
Space Complexity: $O(n)$.

Bibliography

[1] Victor Shoup. *A Computational Introduction to Number Theory and Algebra*. Cambridge University Press, version 2 edition, 2008. Electronic version distributed under Creative Commons Attribution-NonCommercial-NoDerivs 3.0.

[2] Chris Terman. 6.004 computation structures, 2017. Undergraduate course, Spring 2017.