Computer Science Fundamentals:
Intro to Algorithms, Systems, & Data Structures

Christian J. Rudder

October 2024

Contents

Contents **1**

1 Memory Management **6**
1.1 CPU Arichitecture . . . 6
1.2 Code Security . . . 11
1.3 Stack Data Structures . . . 12
1.4 Heap Data Structures . . . 16
1.5 Hashing & Collisions . . . 23
1.5.1 Open Addressing . . . 24
1.5.2 Searching: Insertion & Deletion . . . 27
1.5.3 Separate Chaining & Linked Lists . . . 28
1.5.4 Load Factor & Performance Metrics . . . 33
1.6 Virtual Memory . . . 34
1.6.1 Problem Space of Virtual Memory . . . 34
1.6.2 Virtual Memory Implementation (Page Tables) . . . 37
1.6.3 Page Faults & Translation Lookaside Buffer (TLB) . . . 40
1.6.4 Multi-level Page Tables . . . 42

2 Computational Algorithms **45**
2.1 Information Theory . . . 45
2.1.1 Defining Information . . . 45
2.1.2 Efficient Encodings . . . 50
2.1.3 Error Detection & Correction . . . 51

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 —

Memory Management

1.1 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 1.1: 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.

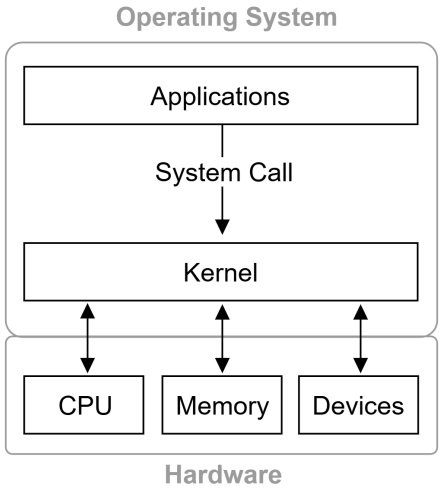


Figure 1.1: User-level applications make syscalls to the kernel to access hardware resources.

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. ■

1.2 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.

1.3 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**.

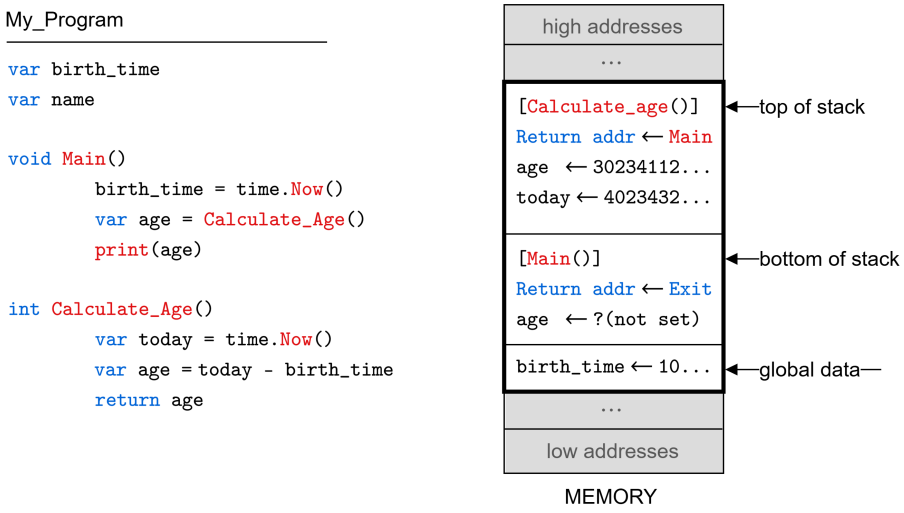


Figure 1.2: Here is a simplified look at how memory manages the stack. On the left is our program written in some abstract language, and on the right is the call stack in memory (simplified). The program has a global 'birth_time' variable, which is initialized in the `Main` function. The `Main` function 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 1.2: Typical x86-32 Stack-Frame Layout, where offsets are typically a multiple of 4 bytes.

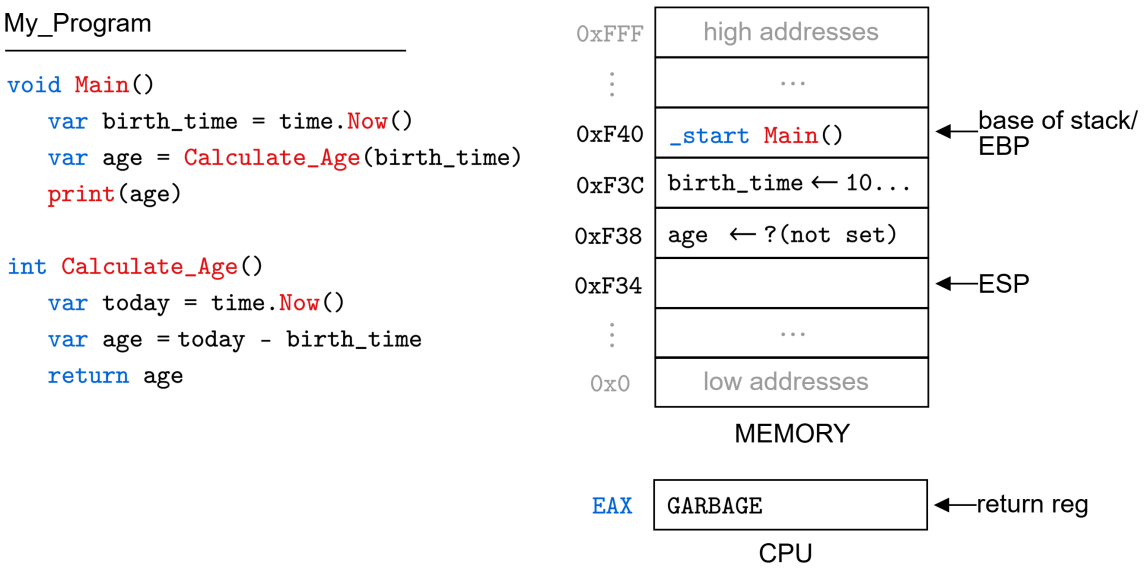


Figure 1.3: Revisiting Figure (1.2) with slight alterations to the code: This is a snapshot of the code executing right before `CalculateAge(birth_time)` is called. For simplicity sake, let's say the stack begins at address `0xF40` (Hexadecimal), growing downwards. Here the base of the stack and the 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
```

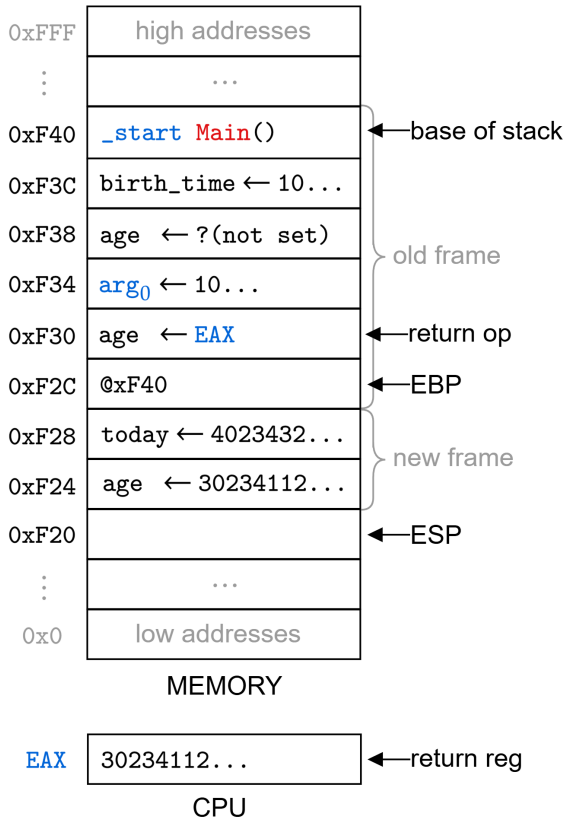


Figure 1.4: Revisiting Figure (1.3) at the moment the function `CalculateAge(birth_time)` has supplied its return value to the EAX register, and is about to return. We see that before calling `CalculateAge(birth_time)`: The old frame pushed it's arguments (`birth_time`) onto the stack, then the return 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.

1.4 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:

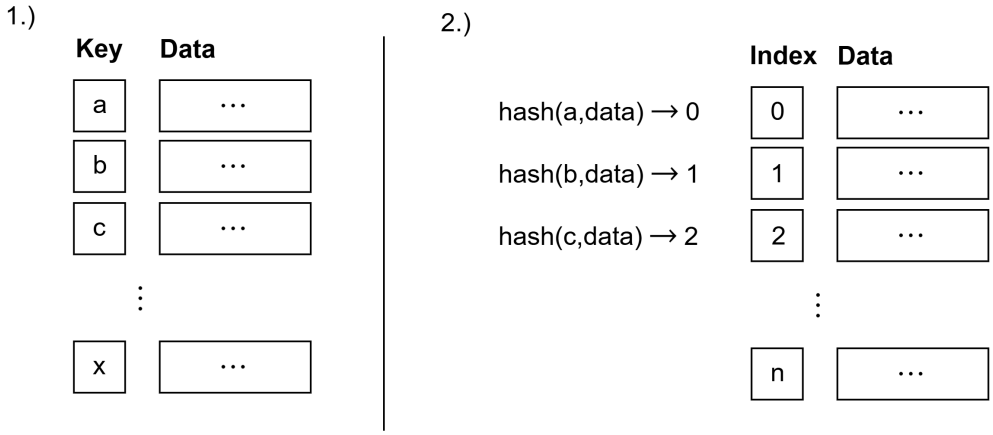


Figure 1.5: On the left (1) demonstrates a typical diagram one might find when learning about hash tables. Here a through x are the keys, which house some type of data. On the right (2) shows a slightly more detailed version, which emphasizes that keys a–x are hashed to indices 0–n in the hash table. 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.

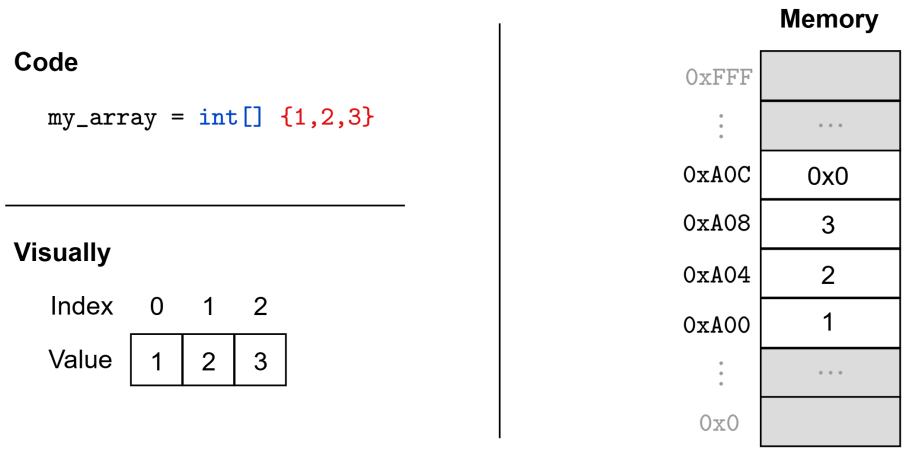


Figure 1.6: Here there are three sections breaking down how arrays look: In code, typical diagram depictions (Visually), and in memory. The code depiction illustrates a toy language creating an array of numbers ($[1, 2, 3]$). The visual depiction shows how indices relate to values. The memory depiction 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 (1.5), we illustrate heaps and arrays:

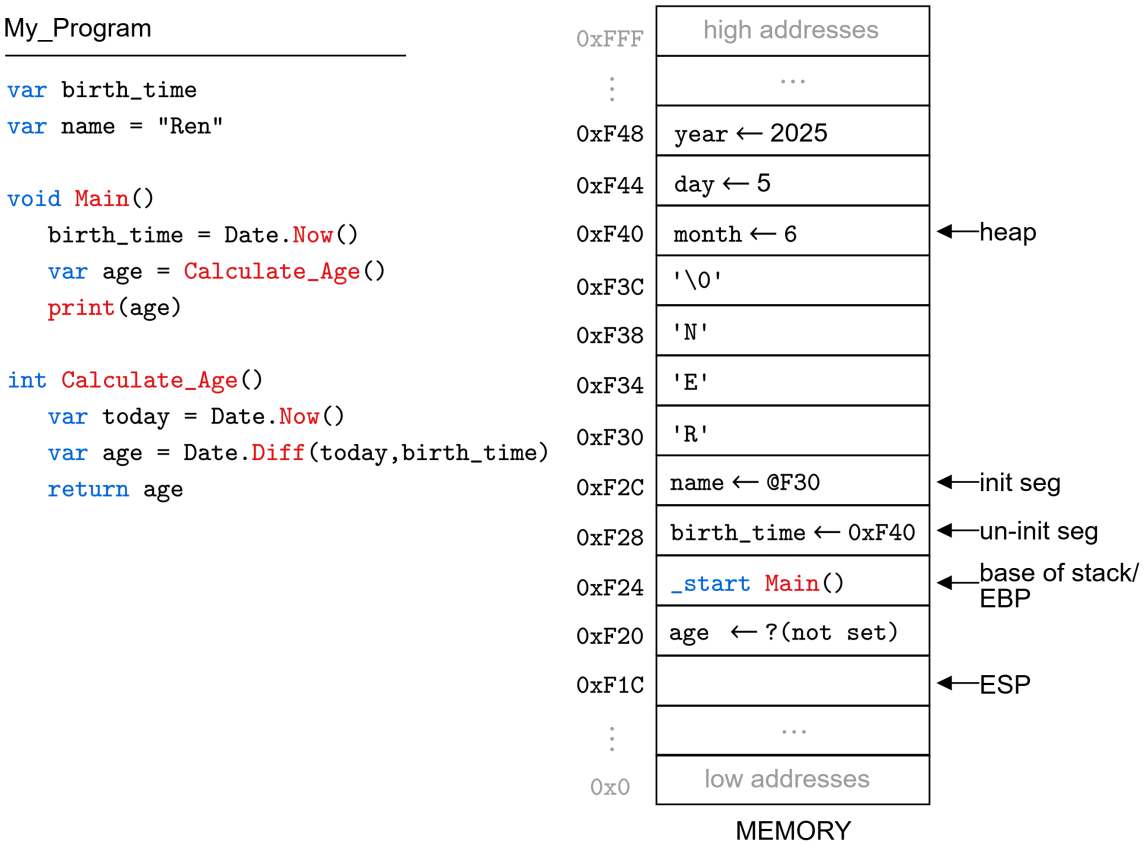


Figure 1.7: Here `birth_time` and `name` are global variables. Following the C convention, `birth_time` is placed in the uninitialized data segment, while `name` is placed in the initialized data segment. Since `name` is a string, it holds a reference to the first character in the string, which is stored 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 (1.7), 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:

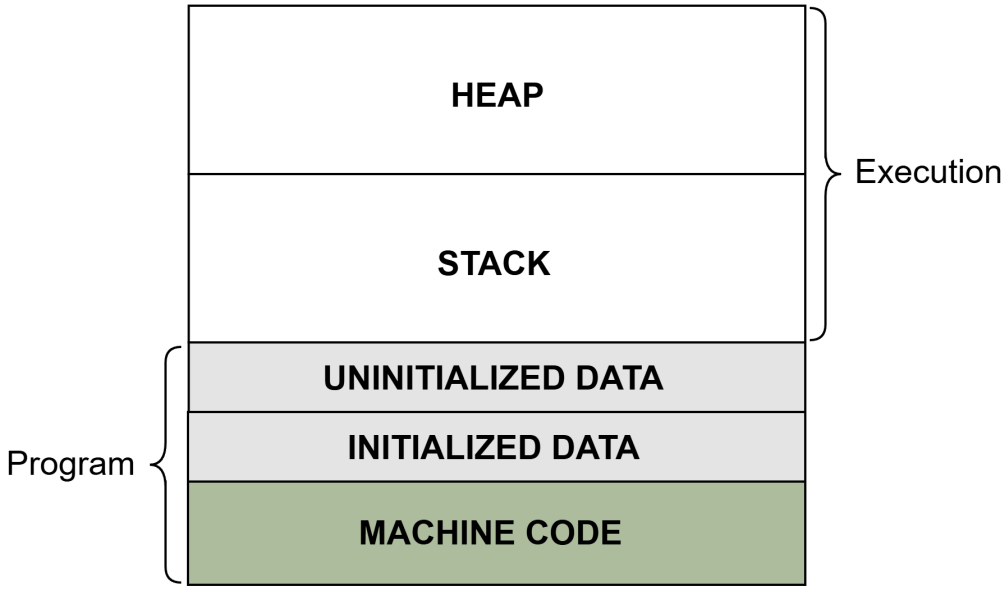


Figure 1.8: The above figure demonstrates the relationship from bottom-to-top the order at which data is loaded into memory. First the program compiles to machine code and loaded by the OS into memory. From there, provisions to the data segment (static memory: uninitialized and initialized) are made. 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.

1.5 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.

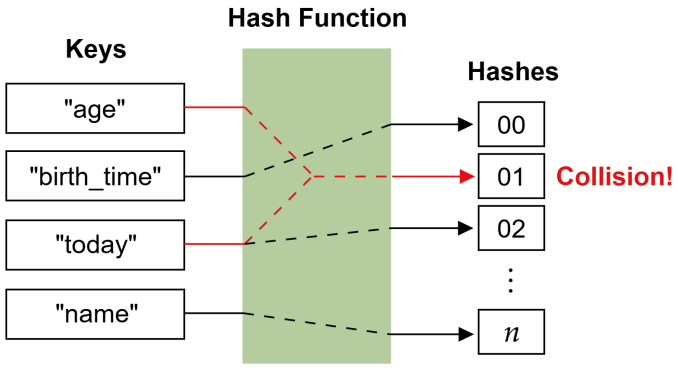


Figure 1.9: Four keys ('age', 'birth_time', 'today', 'name') go through a hash function to n possible indices. Keys, 'birth_time' and 'name', find a unique one-to-one mapping; However, 'age' and 'today' both hash to the same index, causing a collision.

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.

1.5.1 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).

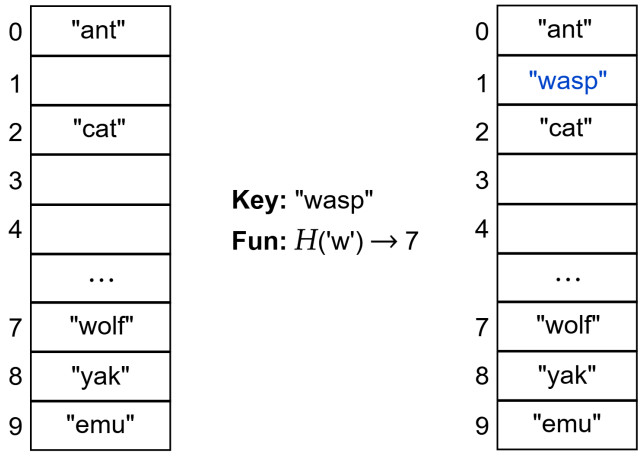


Figure 1.10: On the left is an existing hash table of 10 elements filled with various keys. The middle shows the insertion of a new key, 'wasp', which the function H hashes to index 7; However, index 7 already occupied. The algorithm walks through the table, wrapping around to the beginning, finding 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 1.10).

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.

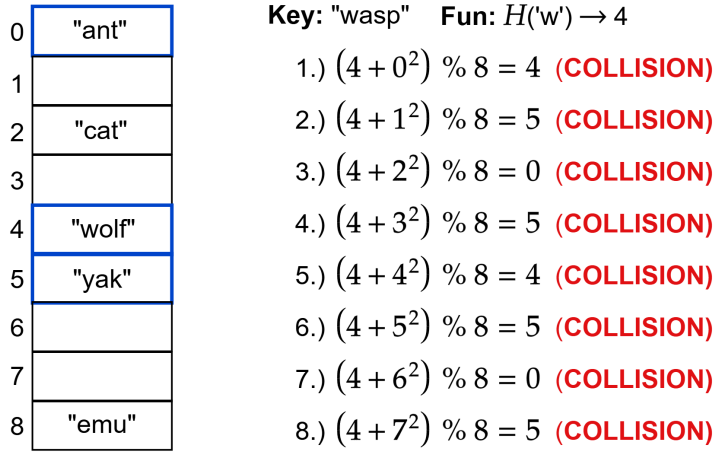


Figure 1.11: On the left is an existing hash table of 8 elements. We attempt to insert a new key, 'wasp', which hashes to index 4; Though, 4 in occupied. The algorithm continues with $(4+1^2) \ \% \ 8 = 5$, which is also occupied. After some probing, it appears only indices $4, 5, 0$ are appearing, from 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 $\{0, 3, 9, 6\}$.

■

1.5.2 Searching: Insertion & Deletion

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

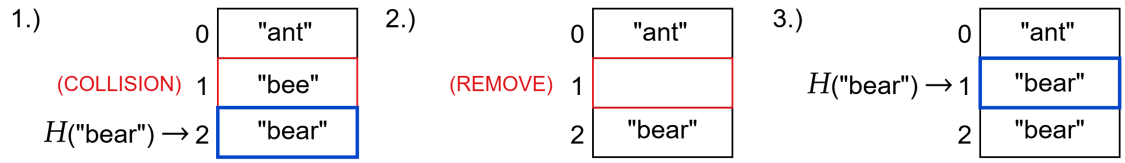


Figure 1.12: Demonstrates complications when inserting into a table blindly. 3.) niaively inserts "bear" at index 1 despite it already existing in the table. This is caused by a previous collision (1) and removal of the collider (2).

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.

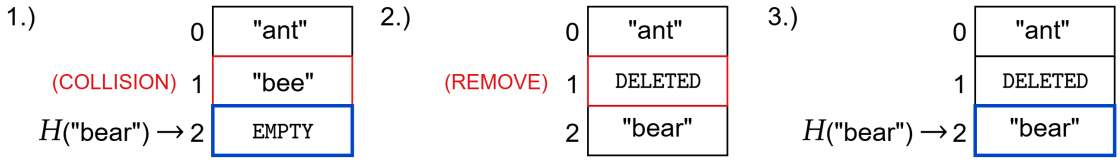


Figure 1.13: Revisiting Figure 1.12 at (3) we continue to probe after finding a deleted index. This leads us to find the already inserted key, "bear", at index 2.

1.5.3 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.

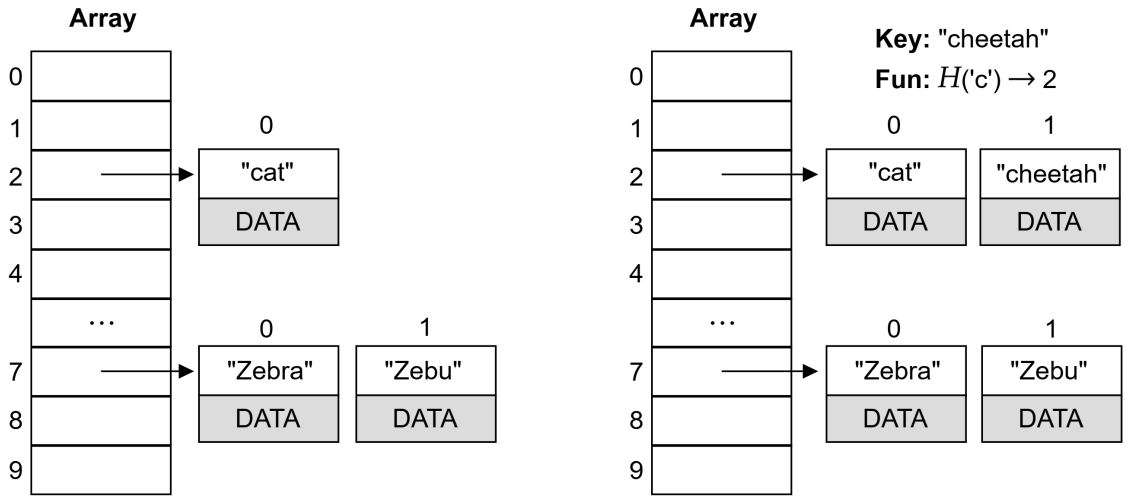


Figure 1.14: The left an array which points to another array (the bucket). The right shows the insertion of "cheetah" into the table, indexing at 2, adding to the end of the bucket. Here it's unclear how the data for each element is stored (perhaps a nested array). Additionally, the **same problem** 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.

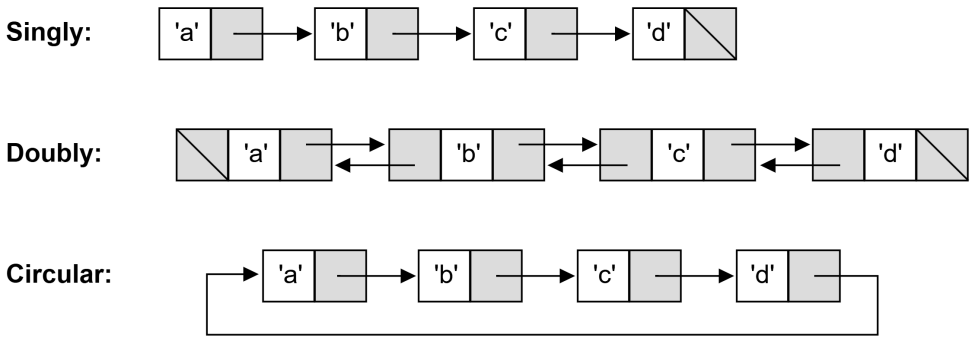


Figure 1.15: A visual representation of a singly, doubly, and circular linked lists.

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 1.14 with a linked list implementation:

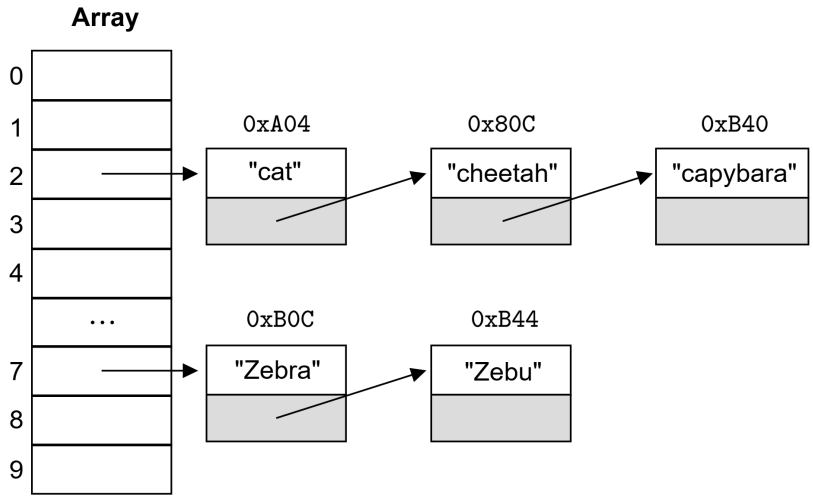


Figure 1.16: Here we see an array with two buckets, each bucket is a linked list. Each node points to the next node in the list. In particular, each node lives in at an ambiguous memory location.

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).

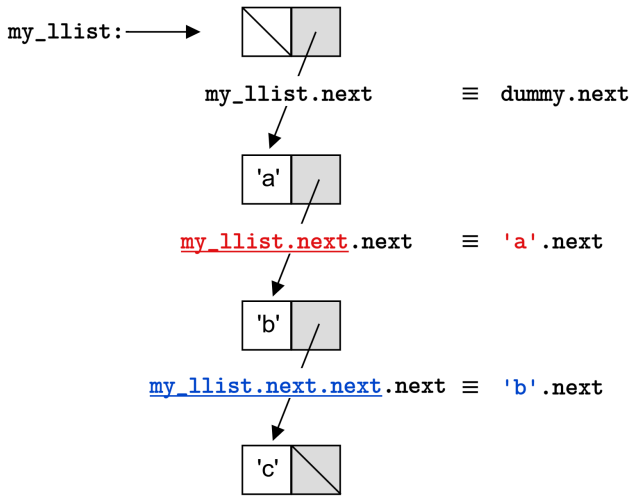
Visualizations on the next page.

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. ■

1.5.4 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 1.3: 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 1.4: 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.

1.6 Virtual Memory

1.6.1 Problem Space of Virtual Memory

Virtual memory solves three problems [1]:

- **Not enough memory, Memory fragmentation, and Security**

Definition 6.1: Not Enough Memory

Back then, computer memory was expensive, and many computers had very little memory (e.g., 4–1 GiB or even less). CPUs could only support up to 4 GiB of memory, as CPUs were 32-bit (2^{32} addresses $= 2^{32} bytes = 4GiB$). On the other hand, 64-bit CPUs can support up to 2^{64} addresses = 16 million TB of memory.

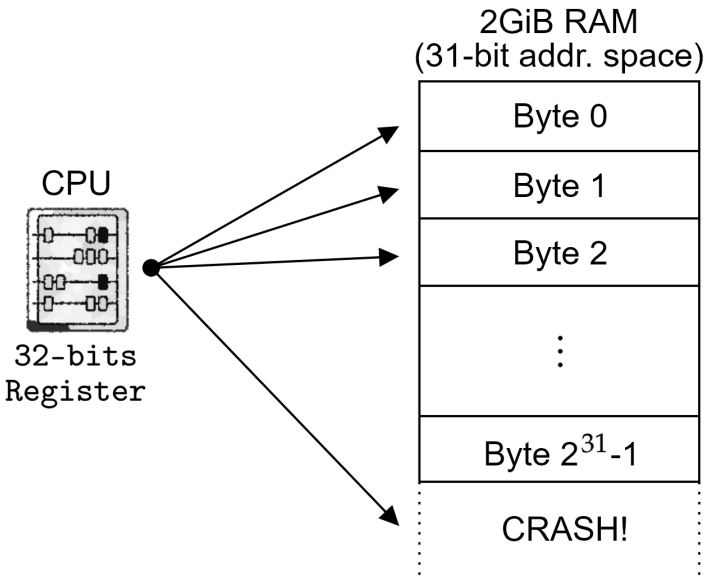


Figure 1.17: A 32-bit CPU accessing 2 GiBs of RAM, where a crash happens when trying to access beyond the 31-bit address space.

The next problem deals with multiple processes allocating and deallocating memory:

Definition 6.2: Memory Fragmentation

Memory can be thought of as a big array, where each cell is a resource a program can use. We want memory usage to be contiguous (i.e., no gaps or holes). So say we have an array

$$[O, O, O, O]$$

Where O represents free space in our array, each cell 1 GiB of space. If we have processes A and B take 1 and 2 GiBs respectively, we might have a memory layout of:

$$[A, B, B, O,]$$

If we then free process A, we might have a memory layout of:

$$[O, B, B, O]$$

Now, if another process C needs 2 GiBs of memory, it will not be able to find a contiguous space of 2 GiBs, even though we have 2 GiBs of free space. This is called **memory fragmentation**.

Now finally we have the problem of protecting memory from other processes:

Definition 6.3: Memory Security

In a multi-process system, processes may have collisions when trying to access the same memory space. For example, if process A is a weather service and process B is some finance service, we don't want the weather service to overwrite the same memory space where the finance service is storing critical data. This is called **memory security**.

So in theory we want to give each process it s own portion of memory, to solve overlapping access:

Definition 6.4: Virtual Memory

To solve process memory collisions, we give each process its own fictional view of memory, called **virtual memory**. Though for this to work, each virtual view is mapped to an actual place in the original memory we call **physical memory**.

Virtual and physical addresses are the cells spaces themselves.

Consider the figure below showing how virtual memory works in theory:

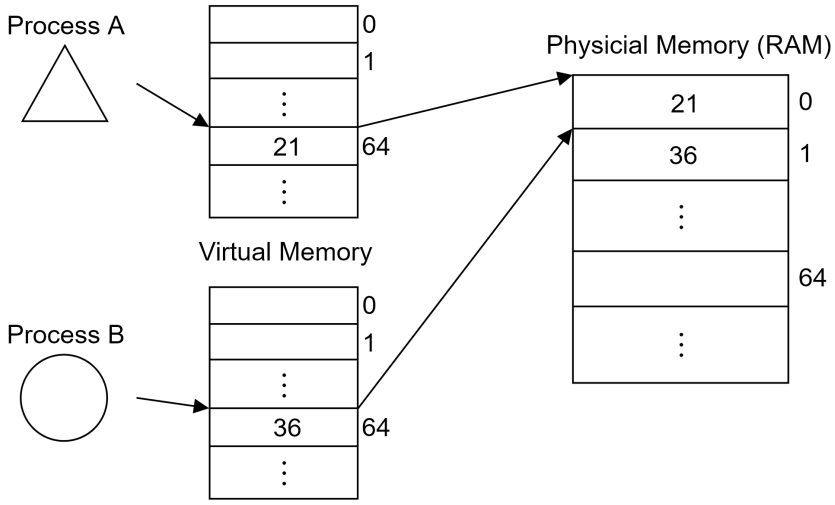


Figure 1.18: Processes A and B write to memory cell 64 in their view of memory, but in reality they map to different physical memory cells (0 and 1 respectively).

A quick aside:

Theorem 6.1: Physical Memory the CPU Accesses

In reality the CPU can access the physical memory of many other devices (e.g., hard drives, SSDs, etc.). In addition, the OS takes up some of the physical memory as well. The rest is left to programs to use. The program allocated space is the memory we refer to going forward as the **physical memory**.

Virtual memory solves the three problems we mentioned before:

Definition 6.5: Virtual Memory & Not Enough Memory (Swapping)

The physical memory can be much smaller than what a program thinks it has in virtual memory. When a program tries to access memory it does not have, the OS will **swap** physical memory to external storage to free up space. This means, while a program is not using a portion of memory at a given time, the OS can swap it in and out depending on system needs.

Memory that is swapped out is called **swap-memory**. Every time a we try to access such absent memory in our mappings, it's called a **page fault** (more on this later).

Definition 6.6: Virtual Memory & Fragmentation

Virtual memory allows programs to think they have contiguous memory. This simplifies their memory management, while in reality the OS manages the split memory mappings.

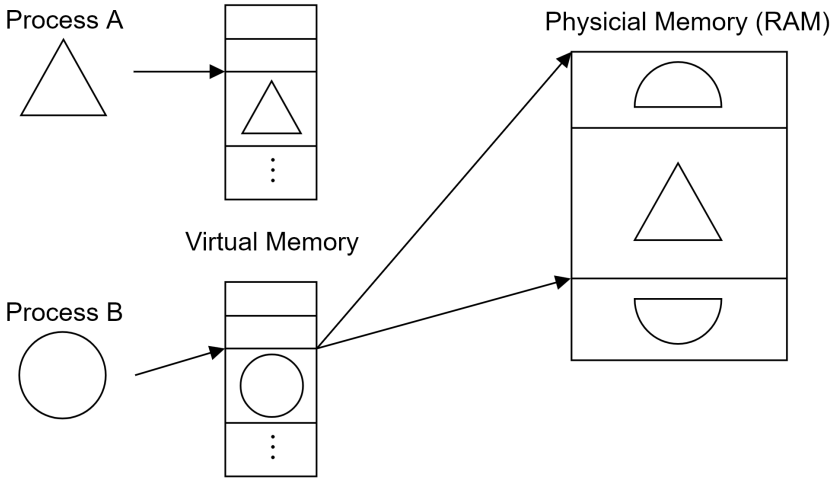


Figure 1.19: Here two processes A and B are using the same physical memory space. Both think they have contiguous memory, but in reality, B is split into two parts in the physical memory.

Definition 6.7: Virtual Memory & Security

Memory cells are collision safe as memory mappings are not shared in physical memory; However, this would be inefficient if say A and B depend on a secondary process C. In this case, A and B may share the same mapping to C's memory.

1.6.2 Virtual Memory Implementation (Page Tables)

We discuss the mapping mechanism further in linking virtual to physical memory.

Definition 6.8: Page Tables

The OS keeps a **page table**, where each entry is a mapping of a virtual memory cell to a physical one, called a **Page Table Entry (PTE)**. CPUs work with words (32 bits = 4 bytes), so the page table has one entry for every word (address) in the virtual memory space.

We make the following observation:

Theorem 6.2: Theoretical Page Table Space Complexity

If our CPU 32-bits, then there are 2^{32} addresses. CPUs speak in words, hence we'd need 2^{30} words, and thus $\approx$ 1 billion PTEs (4 GiB per table). This is too much memory to practically implement for each process.

We solve the above problem, via the following strategy:

Definition 6.9: Memory Chunking (Pages)

Instead of mapping each address to a PTE—which would be too large. We chunk the memory into **pages** reducing the number of PTEs needed. Typically, page sizes are some power of 2, most commonly 4 KiB (4096 bytes), meaning 1024 words per page. This reduces our page table requirement from 4 GiB to 4 MiB (1 Million PTEs). Despite moving 4 KiBs of data at a time, this works well in practice, as adjacent memory is often accessed together.

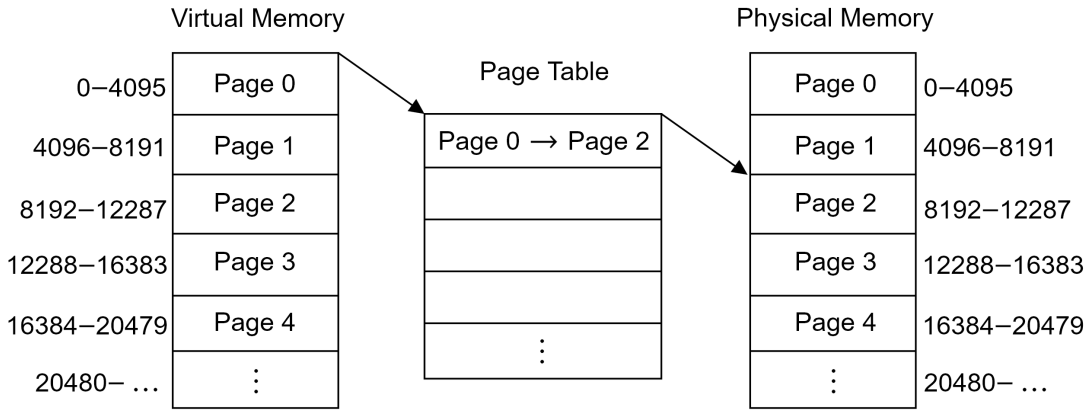


Figure 1.20: A 4 MiB page table, each PTE 4 KiB (4096 bytes, 1024 words), showing mappings.

One must ask themselves:

- What is the relationship of how the Page Numbers are separated.
- Can we determine the page number given a specific address?
- How might we convert a virtual to a physical address given some address?

We discuss the following on the next page.

Example 6.1: Converting Virtual to Physical Memory (intuition)

We may find conversions from virtual to physical memory via the following formula:

$$\text{PA} = \underbrace{(\text{VA} \ \% \ \text{Page Size})}_{\text{offset}} + \underbrace{(\text{Page Size} \cdot \text{Physical Page Number})}_{\text{Physical Page starting index}}$$

Where % is modulo and Page Number $= \left\lfloor \frac{\text{Addr.}}{\text{Page Size}} \right\rfloor$. **However,** this is not the most efficient way, and is strictly for educational/intuition building purposes.

Given the Figure (1.20), $104 \to 8296$ from $104 + 4096 * 2 = 8296$. ■

Theorem 6.3: Converting Virtual to Physical Memory

Addresses are binary numbers, typically represents as hexadecimal (base 16) numbers.

The first 12 bits of an address is the **offset**. The remaining higher-order bits yields the **page number**. To find the physical address, take the last 12 bits of the address, index the higher-order bits into the page table, and append the offset to the physical page number.

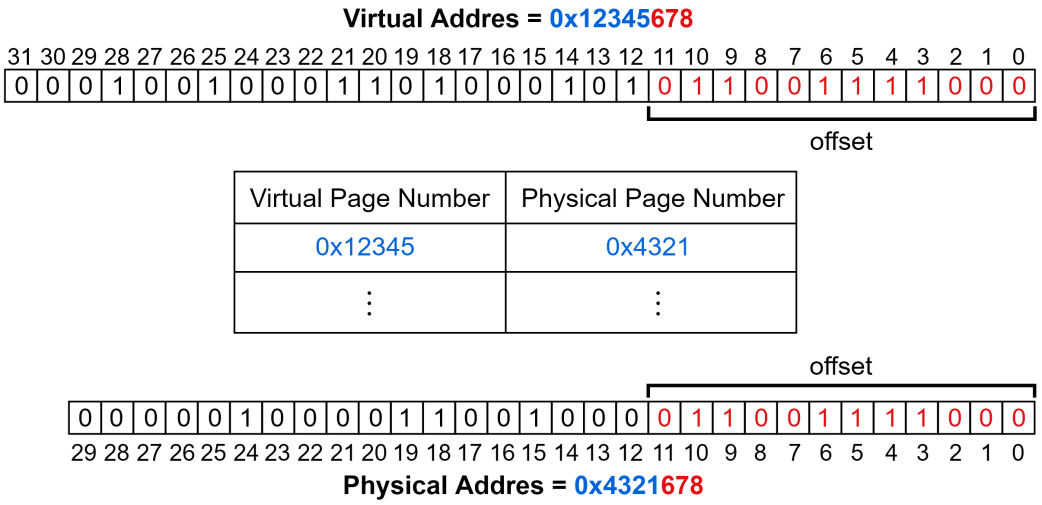


Figure 1.21: The conversion of 0x123456 (32-bit, 4 GiB) → 0x4321678 (30-bit, 1 GiB). Recall that if a accessing a page that is not in memory, causes a page fault, from which the OS with begin to swap memory with external storage devices.

1.6.3 Page Faults & Translation Lookaside Buffer (TLB)

So far we have discussed how the mapping system of virtual memory works, but now we pivot to how the OS handles swapping data in and out of memory on page faults.

Definition 6.10: Swapping on Page Faults

A **page fault** occurs when a program tries to access a page that does not have a mapping in the page table. The OS then picks the **least recently used (LRU)** page in the page table, and swaps it out to external storage. A page is **dirty** if it has been written to. In such case, we write back its contents to external storage before swapping. If the page is not dirty, we may discard it on swap.

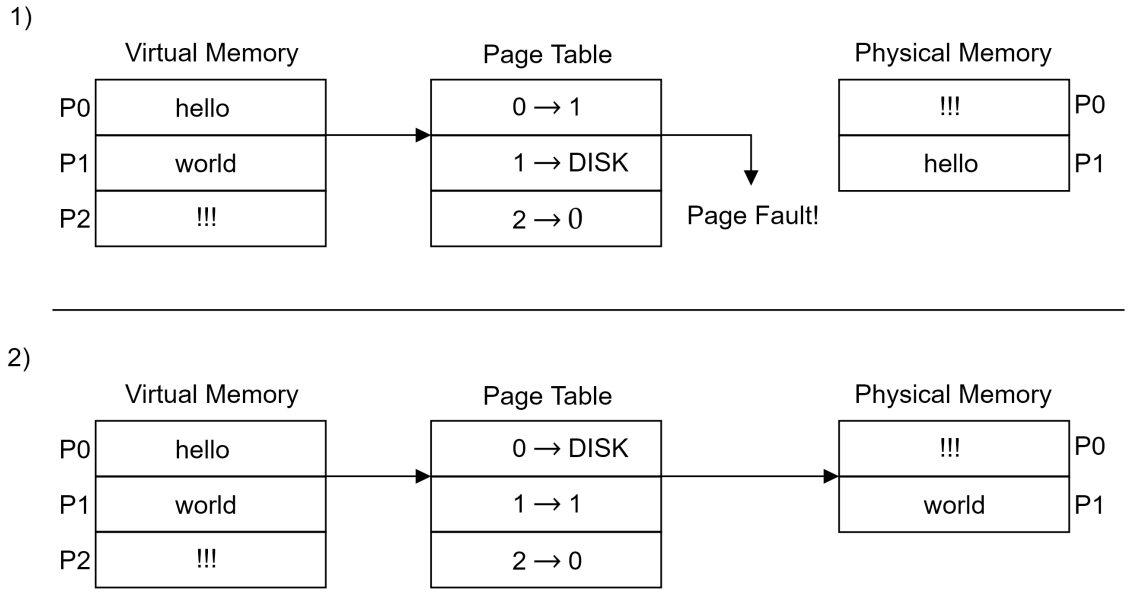


Figure 1.22: 1) A page fault occurs when trying to access page 1. 2) The OS has swapped out page 0 to external storage (DISK), and now page 1 can be accessed. Note: This diagram is for illustrative purposes, virtual memory does not contain data, only addresses binding to physical memory.

Definition 6.11: Direct Memory Access (DMA)

Page faults are expensive, as swapping data with I/O devices may take some time. Modern CPUs have a **Direct Memory Access (DMA)** module, which allows I/O devices to access memory directly, while the CPU completes other tasks.

Still our routine of mapping is expensive in it of itself.

Definition 6.12: Translation Lookaside Buffer (TLB)

The mapping process includes three steps:

1. Index the page table: Access Memory (RAM).
2. Convert Addresses: Computation.
3. Interact: Access Memory (RAM).

To speed this up, we create a small cache of the most recent Page Table Lookups, called the **Translation Lookaside Buffer (TLB)**. If a page in not in the TLB, we must preform the translation then load it into the TLB for future use.

Every successful TLB lookup is called a **hit**, while a failed lookup is called a **miss**. The **hit time** and **miss time** are the time it takes to access the TLB and page table respectively (measured in CPU clock cycles). The **hit rate** is the ratio of hits to the total number of lookups.

Modern CPU architectures usually have two TLBs, one for instructions (**ITLB**) and one for data (**DTLB**).

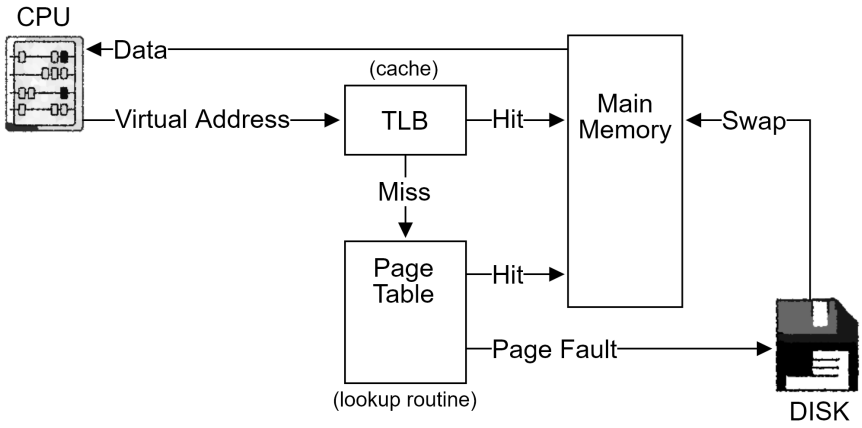


Figure 1.23: Demonstrating how a TLB lookup effects the memory access flow. CPU attempts to access memory providing the virtual address to the TLB. (Happy path) A hit occurs, memory is accessed and passed to the CPU. (Ok path) A TLB miss occurs, causing a page table lookup. A page table hit occurs, the mapping is cached and the memory is accessed. (Sad path) both a TLB and page table miss occurs, causing a page fault. The OS must decide which page to swap out. After swapping, the addressed is cached and the memory is accessed.

So far what we've been covering actually resides within the CPU architecture:

Definition 6.13: Memory Management Unit (MMU)

The **Memory Management Unit (MMU)** is a hardware component that manages the mapping of virtual to physical memory. It is responsible for translating virtual addresses to physical addresses, and it also handles page faults and TLB lookups.

1.6.4 Multi-level Page Tables

Let's define the problem space of multi-level page tables [**?**]:

Theorem 6.4: Single Level Page Table Cost

In a 32-bit system, each page table requires 4 MiB of memory. If we had 100 programs running, that's easily 400 MiB of memory.

Additionally, if we move page tables to disk (external storage), we lose any reference we had to them in an attempt to free up memory. So we need to keep some reference oracle in memory.

This highlights the need for a more efficient way

Definition 6.14: Multi-level Page Tables

Recall that in a 32-bit system, we have 4 GiB of memory. In single level page tables, we chunk our addresses into 4 KiBs (4096 bytes), and load all of them at once. That's

$$4 \text{ KiB} \cdot 1024 \text{ PTEs } = 4096 \text{ KiB} = 4 \text{ MiB}$$

We wish to offload data without losing references. To do this we allocate a single chunk of memory (4 KiB) to serve as an reference oracle, called the **page directory** or **1^{st} level page table**. Each entry in the page directory points to other chunks from which we are safe to load and unload out of memory. These referenced chunks are called the **2^{nd} level page tables**. Each entry in the 2^{nd} level page table points to a physical page.

Definition 6.15: Converting Virtual to Physical Memory (Multi-level)

In multi-level page tables, the first 12 bits are the **offset**, the next 10 bits are the index into the **second-level page table**, and the last 10 bits are the index into the **first-level page directory**. These tables are treated as arrays: indexing the first table yields the address of the second table, and indexing the second table yields the physical page number.

Recall, in single-level page tables the higher 20 bits are the page number and the index, which yields the physical page number. Multi-level breaks away from this, and instead as illustrated below:

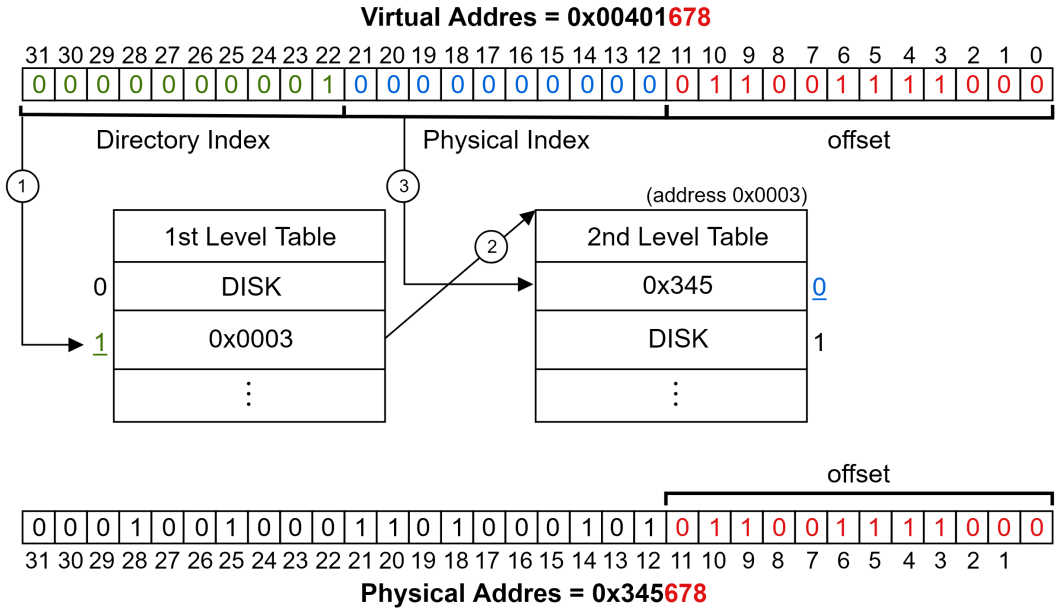


Figure 1.24: A multi-level page table, where the first level is a page directory, and the second level is a page table which points to physical pages.

Definition 6.16: Multi-level Page Scaling (64-bit)

Increasing the number of bits allows us to scale the number of levels in our page table. In 64-bit architectures using 4-level paging (e.g., x86-64), only the lower 48 bits are used. The virtual address is divided into five parts:

- 12 bits for the page offset
- 9 bits for the 4^{th}-level page table (PT)
- 9 bits for the 3^{rd}-level page directory (PD)
- 9 bits for the 2^{nd}-level page directory pointer table (PDPT)
- 9 bits for the 1^{st}-level page map level 4 (PML4)

Modern versions of Windows and Linux support 5 levels, which allows for a maximum of 129 PiB, while 4 gives us 256 TiB.

Finally we talk about the performance of our lookups:

Definition 6.17: Effective Memory Access Time (EMAT)

Effective Memory Access Time (EMAT) accounts for the time it takes to access memory in the presence of a Translation Lookaside Buffer (TLB), multi-level paging, and potential page faults. Given:

- t = TLB access time; m = Memory access time
- S = Page fault service time; n = Number of page levels
- h = TLB hit rate; p = Page hit rate (1 - page fault rate)

The EMAT is computed as:

$$\text{EMAT} = h(t+m) + (1-h)\,(t + p(n \cdot m) + (1-p)S)$$

Essentially [**?**],

```
EMAT=
TLB hit*(TLB access time + Memory access time)
+ TLB Miss*(TLB access time + PageHit*[n * memory access time]
+ PageMiss*PageFaultServiceTime)
```

Example 6.2: Three-Level Paging System

Suppose the system has:

- $n = 3$ page levels
- $t = 5$ ns (TLB lookup)
- $m = 100$ ns (memory access time)
- $\alpha = 0.80$ (TLB hit rate)

Then the EMAT is:

$$\begin{aligned}\text{EMAT} &= (5+100)\cdot 0.80 + (5 + 3\cdot 100 + 100)\cdot(1-0.80)\\ &= 105\cdot 0.80 + 405\cdot 0.20 = 84 + 81 = \boxed{165 \text{ ns}}\end{aligned}$$

For a better TLB hit rate $\alpha = 0.98$:

$$\text{EMAT} = 105\cdot 0.98 + 405\cdot 0.02 = 102.9 + 8.1 = \boxed{111 \text{ ns}} \qquad [?]$$

■

— 2 —

Computational Algorithms

2.1 Information Theory

2.1.1 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" [3, 2].

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.

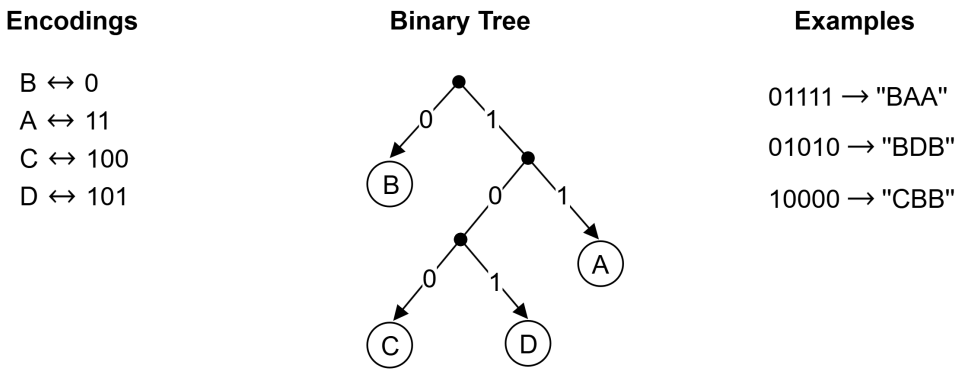


Figure 2.1: Encodings start at the root, each edge taken writes the next bit.

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:

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

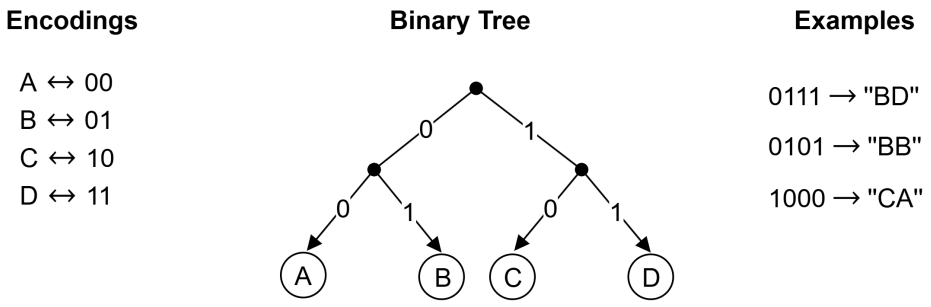


Figure 2.2: A fixed-length encoding for four symbols, represented as a binary tree.

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 (2.1), 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.

2.1.2 Efficient Encodings

Now that we are familiar with Binary Tree Encodings (Theorem 1.1), optimal encodings:

Definition 1.5: Huffman's Algorithm

Build a subtree with the two least probable symbols. Such subtree is now considered a single symbol with a probability equal to the sum of the previous two. Do so until all symbols are combined into a single tree, called a **Huffman tree**, an optimal variable-length encoding.

Time Complexity: $O(n \log n)$, where n is the number of symbols. The actual encoding is done in $O(n)$ time, the $\log n$ factor comes from the need to sort the symbols by probability.

Symbol	Probability
A	1/3
B	1/2
C	1/12
D	1/12

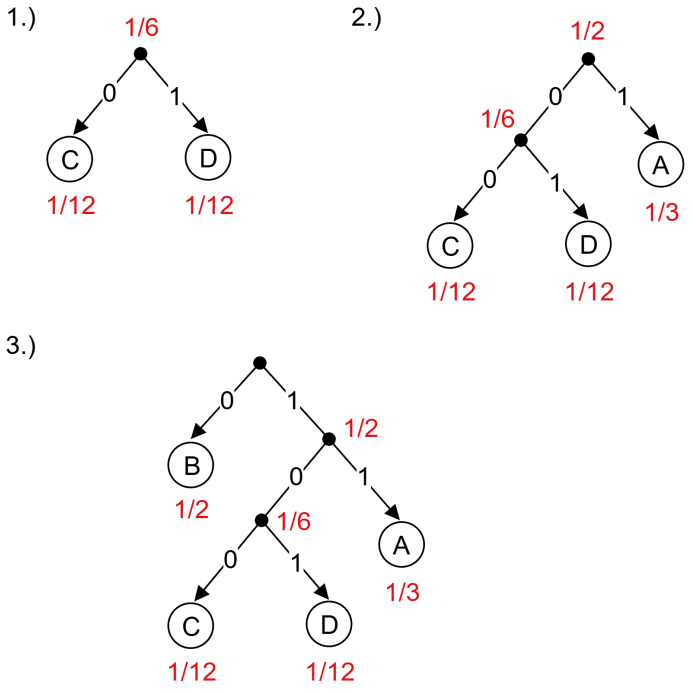


Figure 2.3: A Huffman tree for the symbols A, B, C, and D with their respective probabilities. (1) C and D are combined first having the highest probabilities, 1/12, which combine to a subtree with a probability of 1/6. (2) A is added, 1/3 combining with the subtree $C + D$ to form a new subtree with a probability of 1/2. (3) Finally, B is added, which has the highest probability of 1/2, resulting in the final Huffman tree.

This brings us to the field of compression:

Definition 1.6: Lossless Data Compression

Lossless data compression is a technique that reduces the size of a file without losing any information. This is often achieved by encoding chunks of redundancy into a single symbol.

2.1.3 Error Detection & Correction

Two people Alice and Bob were trying to communicate a game of heads or tails over a noisy channel:

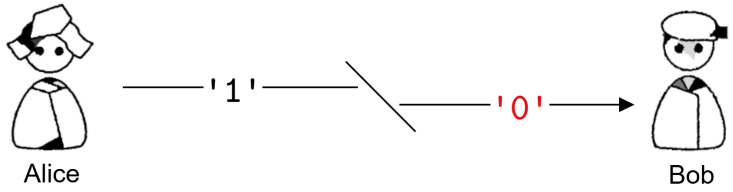


Figure 2.4: Alice and Bob trying to communicate '1' for heads, but the message is corrupted, and Bob receives a '0' instead.

As we see above, such a single bit error can be catastrophic. To mitigate this, our encodings must be more robust.

Definition 1.7: Hamming Distance

A **Hamming distance** is the number of bit positions in which two same length encodings differ. If there is a single bit error, the Hamming distance is 1. If there is none, then the two encodings are identical.

Example 1.5: Hamming Distance

The Hamming distance between the encodings,

$$0\ 1\ 1\ 0\ 1\ 1\ 1$$
$$0\ 1\ 0\ 0\ 1\ 1\ 0$$

is 2, as there are two bit positions that differ (marked in red). ■

Definition 1.8: Single-bit Error Detection

If two valid distinct encodings have a Hamming distance of 1, then a single-bit error may occur; Therefore if we increase the minimum Hamming distance to 2, we can detect single-bit errors. This may be done by adding a parity bit to the end of the encoding:

- **Even Parity:** Once added, the total number of 1's in the encoding is even.
- **Odd Parity:** Once added, the total number of 1's in the encoding is odd.

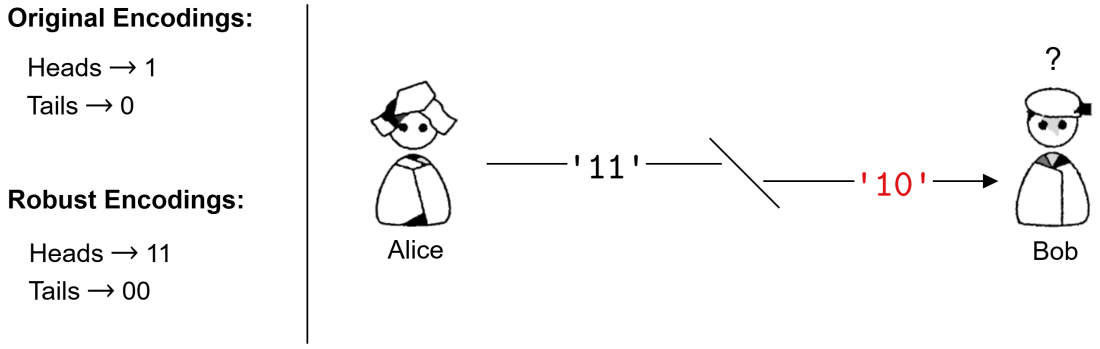


Figure 2.5: Alice and Bob using even parity to detect single-bit errors. Bob receives a single bit error and immediately detects it by checking the parity. **Note:** This does not help us if more than one bit is flipped.

Theorem 1.5: Detecting Multi-bit Errors

To detect E errors within an encoding, a minimum Hamming distance of $E + 1$ is required between each code word.

Theorem 1.6: Error Correction

Let there be code A and B, if the Hamming distance between them is $2E + 1$, then we can correct E errors. This is because the set of possible errors A_e and B_e will not overlap, allowing us to deduce the original encoding.

Bibliography

[1] Computerphile. But, what is virtual memory?, 2023. Accessed: 2025-04-18.

[2] 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.

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