Tusk 1

CRM vs ERP

CRM and ERP are two distinct types of software systems that are used by businesses to manage their operations.

CRM stands for Customer Relationship Management. It is a software system that is designed to help businesses manage their customer interactions, sales activities, and marketing campaigns. CRM software typically includes features such as contact management, lead tracking, sales forecasting, and customer analytics.

ERP, on the other hand, stands for Enterprise Resource Planning. It is a software system that is designed to help businesses manage their entire organization's resources, including finance, manufacturing, inventory, and human resources. ERP software typically includes modules for accounting, procurement, supply chain management, production planning, and HR management.

While there is some overlap between the two software systems, they are designed for different purposes. CRM is primarily focused on managing customer interactions and sales, while ERP is focused on managing the broader organizational resources and operations.

In summary, CRM and ERP are two different types of software systems that serve different purposes. CRM is focused on managing customer relationships and sales activities, while ERP is focused on managing the broader resources and operations of an organization.

Tusk 2

What is Software on the shelf (SOTS) ?

"Software on the shelf" is a term used to describe pre-packaged software that is readily available for purchase and use without requiring any significant modifications or customization. This type of software is typically designed to be applicable to a wide range of businesses or organizations and may include features that are commonly used by many different types of users.

The term "on the shelf" suggests that the software is physically available and ready to be used immediately after purchase, much like a book or any other tangible product that can be purchased off a shelf. This is in contrast to custom software, which is developed specifically for

a particular organization or user and is tailored to their specific needs and requirements.

Examples of software on the shelf include popular productivity software like Microsoft Office or Adobe Creative Suite, as well as customer relationship management (CRM) systems like Salesforce or Hubspot.

Tusk 3

Clean Code vs Spaghetti Code

Clean code and spaghetti code are two contrasting concepts in software development.

Clean code refers to code that is easy to read, understand, and maintain. It follows best practices and principles of software development such as SOLID principles, DRY (Don't Repeat Yourself) principle, and KISS (Keep It Simple Stupid) principle. Clean code is often well-structured, well-documented, and has a clear and concise purpose.

On the other hand, spaghetti code refers to code that is difficult to read, understand, and maintain. It is often poorly structured, hard to debug, and prone to errors. Spaghetti code may also violate best practices and principles of software development, making it hard to modify or extend.

The term "spaghetti code" comes from the idea that the code looks like a plate of spaghetti, with lines of code crisscrossing each other in an unorganized and confusing manner.

In summary, clean code is desirable because it makes it easier to develop, maintain, and scale software applications, while spaghetti code can lead to technical debt and can slow down the development process.

Tusk 4

Paging & Fragmentation

Paging is a memory management scheme used by modern operating systems to manage memory allocation and access. In this scheme, physical memory is divided into fixed-size blocks called "pages", and logical memory used by a program is divided into the same-sized "page frames".

When a program requests memory, the operating system allocates one or more page frames

and maps them to the logical memory addresses requested by the program. As the program executes, it accesses the logical memory addresses, and the operating system translates those addresses to physical memory addresses using the page table.

If the program requests more memory than is available in physical memory, the operating system uses a technique called "page swapping" to move less frequently used pages out of physical memory to disk, freeing up space for more frequently used pages. When the program accesses a swapped-out page, the operating system brings it back into physical memory by swapping out another page, if necessary.

Paging allows the operating system to manage memory more efficiently and enables programs to access more memory than is physically available. However, it can also introduce overhead due to the need for frequent page table lookups and page swapping, which can impact performance.

Fragmentation is a term used in computing to describe a situation where a storage device, such as a hard disk or memory, becomes fragmented or broken into small, non-contiguous pieces.

In the context of memory management, fragmentation occurs when the free memory available for a program to use is divided into small, non-contiguous blocks that are scattered throughout the memory space. This can occur over time as a result of memory allocation and deallocation, where memory blocks are allocated and freed in a way that leaves gaps between them.

Fragmentation can be classified into two types: internal fragmentation and external fragmentation. Internal fragmentation happens when a memory block is allocated to a program that is larger than what it actually needs, resulting in wasted memory. External fragmentation occurs when the memory space available for a program to use is broken up into small, non-contiguous blocks that are too small to satisfy the memory allocation request of the program.

Fragmentation can lead to performance issues as it can slow down memory access times, increase the overhead of memory management, and reduce the amount of memory available for use by programs. To address fragmentation, techniques such as memory compaction and garbage collection are used to consolidate small memory blocks into larger contiguous blocks, which can be more efficiently utilized by programs.

Tusk 5

what time that we need code to be slower ??

While in general we want our code to run as fast as possible, there are some cases where we may actually want our code to be slower. Here are a few examples:

Animation and video games: In animation and video games, it can be desirable to slow down the code in order to make the animation or game look smoother and more natural. This can be achieved through techniques like frame interpolation and motion blur, which rely on slowing down the code to achieve a desired visual effect.

Debugging and testing: When debugging or testing code, it can be useful to slow down the code in order to more easily observe what is happening in the program. By slowing down the code, we can see the state of the program at different stages of execution and more easily identify and diagnose issues.

Human interaction: In some cases, we may want our code to run slower in order to enable interaction with humans. For example, in a game where a player needs to make a decision, we may want to slow down the code to allow the player enough time to make a decision.

These are just a few examples where we may want our code to be slower. However, in general, we strive to make our code as efficient and fast as possible, while still achieving the desired functionality and performance requirements.

Tusk 6

how many times RAM is faster than VM?

RAM (Random Access Memory) is the physical memory in a computer that holds data and instructions that the CPU (Central Processing Unit) can access quickly. Virtual memory is a technique used by the operating system to increase the amount of usable memory beyond the physical RAM available in a computer. It uses the hard drive as a temporary storage space to swap out data from RAM when it is not being used, allowing more programs to run simultaneously.

RAM is much faster than virtual memory because it is directly accessible by the CPU and does not require any data transfer. Virtual memory, on the other hand, is significantly slower than RAM because data has to be swapped in and out of the hard drive, which takes more time and reduces performance.

The speed of RAM and virtual memory depends on various factors, such as the type, size, and

configuration of the RAM and the hard drive, the number and size of programs running on the computer, and the operating system settings. However, some general estimates are:

- The speed of RAM can range from about 2 GB/s to 25 GB/s or more, depending on the type of RAM (such as DDR3 or DDR4) and its frequency (such as 1333 MHz or 3200 MHz).

- The speed of virtual memory can range from about 50 MB/s to 200 MB/s or more, depending on the type of hard drive (such as HDD or SSD) and its interface (such as SATA or NVMe).

Therefore, RAM can be about 10 to 500 times faster than virtual memory, depending on the specific hardware and software conditions. However, this does not mean that virtual memory is useless. It is still a useful technique to extend the available memory for running more programs and avoiding out-of-memory errors. The best way to improve the performance of a computer is to have enough RAM for its needs and use a fast hard drive for virtual memory.

Tusk 7

How to inverse periority queue in python

A priority queue is a data structure that stores items with a priority value and allows retrieving the item with the highest priority first. In Python, you can use the queue.PriorityQueue class to implement a priority queue.

To inverse a priority queue in Python, you have a few options:

- You can multiply the priority value by -1 when putting items into the queue, and then multiply it by -1 again when getting items from the queue. This will reverse the order of the items based on their priority values. For example:

```python

import queue

q = queue.PriorityQueue()

q.put((-1 * 9, 9)) # put 9 with priority -9

q.put((-1 * 1, 1)) # put 1 with priority -1

q.put((-1 * 4, 4)) # put 4 with priority -4
```

```
q.put((-1 * 5, 5)) # put 5 with priority -5

while not q.empty():

    item = q.get()

    print(-1 * item[0], item[1]) # print priority and value
```

Output:

```text
9 9

5 5

4 4

1 1
```

- You can create a wrapper class that implements the __lt__ method to compare items based on their priority values in reverse order. This will allow you to use the original priority values without multiplying them by -1. For example:

```python
import queue

class Wrapper:

    def __init__(self, value):

        self.value = value

    def __lt__(self, next):

        return self.value[0] > next.value[0] # compare priority values in reverse order

q = queue.PriorityQueue()

q.put(Wrapper((9, 9))) # put 9 with priority 9

q.put(Wrapper((1, 1))) # put 1 with priority 1
```

```
q.put(Wrapper((4, 4))) # put 4 with priority 4

q.put(Wrapper((5, 5))) # put 5 with priority 5

while not q.empty():

   item = q.get()

   print(item.value[0], item.value[1]) # print priority and value
```

Output:

```text
9 9

5 5

4 4

1 1
```

- You can use a heap data structure instead of a queue.PriorityQueue class. A heap is a binary tree that maintains the heap property: the parent node is always smaller (or larger) than its children nodes. In Python, you can use the heapq module to implement a heap. To inverse a heap, you can use the same methods as for a priority queue: multiply the priority values by -1 or use a wrapper class. For example:

```python
import heapq

heap = []

heapq.heappush(heap, (-1 * 9, 9)) # push 9 with priority -9

heapq.heappush(heap, (-1 * 1, 1)) # push 1 with priority -1

heapq.heappush(heap, (-1 * 4, 4)) # push 4 with priority -4

heapq.heappush(heap, (-1 * 5, 5)) # push 5 with priority -5

while heap:
```

```
    item = heapq.heappop(heap)
    print(-1 * item[0], item[1]) # print priority and value
```

Output:

```text
9 9
5 5
4 4
1 1
```

Tusk 8

How can I implement a custom comparator function for a priority queue?

A custom comparator function is a function that defines how to compare two items based on their priority values. A priority queue in Python can use a custom comparator function in different ways:

- You can use a wrapper class that inherits from the queue.PriorityQueue class and overrides the put and get methods to use the custom comparator function internally. For example:

```python
import queue
class CustomPriorityQueue(queue.PriorityQueue):
    def __init__(self, comparator):
        super().__init__()
```

```
        self.comparator = comparator # store the comparator function
    def put(self, item):
        # wrap the item in a tuple with the priority value from the comparator function
        priority = self.comparator(item)
        super().put((priority, item))
    def get(self):
        # unwrap the item from the tuple and return it
        priority, item = super().get()
        return item
```

- You can use a wrapper class that implements the __lt__ method to compare two items based on their priority values using the custom comparator function. For example:

```python
import queue
class Wrapper:
    def __init__(self, item, comparator):
        self.item = item
        self.comparator = comparator # store the comparator function
    def __lt__(self, other):
        # compare two items using the comparator function
        return self.comparator(self.item, other.item)
q = queue.PriorityQueue()
# define a custom comparator function that compares items by their length
def compare_by_length(a, b):
    return len(a) < len(b)
```

```
# wrap each item in a Wrapper object with the comparator function
q.put(Wrapper("hello", compare_by_length))
q.put(Wrapper("world", compare_by_length))
q.put(Wrapper("hi", compare_by_length))
# unwrap each item from the Wrapper object and print it
while not q.empty():
    print(q.get().item)
```

- You can use a heap data structure instead of a queue.PriorityQueue class and use the heapq module to implement a priority queue. To use a custom comparator function with a heap, you can use the functools.cmp_to_key helper function to convert it to a key function. For example:

```python
import heapq
import functools
heap = []
# define a custom comparator function that compares items by their value
def compare_by_value(a, b):
    return a.value - b.value
# convert the comparator function to a key function using cmp_to_key
key_func = functools.cmp_to_key(compare_by_value)
# push each item to the heap with its key value from the key function
heapq.heappush(heap, (key_func(a), a))
heapq.heappush(heap, (key_func(b), b))
heapq.heappush(heap, (key_func(c), c))
# pop each item from the heap and print it
```

```
while heap:
    print(heapq.heappop(heap)[1])
```

Tusk 9

How to create a multiset and unorderedset with python

A multiset is a collection that allows multiple occurrences of the same element, while an unordered set is a collection that does not preserve the order of the elements. In Python, there are different ways to create a multiset and an unordered set:

- To create a multiset, you can use the multiset module that provides a Multiset class that is similar to the built-in set class but allows repeated elements. You can install this module using pip install multiset and then import it using from multiset import *. You can create a multiset object by passing an iterable or a mapping of elements and their multiplicities to the Multiset constructor. For example:

```python
from multiset import *
ms = Multiset("aab") # create a multiset from a string
print(ms) # {a, a, b}
ms = Multiset({"a": 2, "b": 1}) # create a multiset from a dictionary
print(ms) # {a, a, b}
```

- To create an unordered set, you can use the built-in set class that provides a collection of unique and hashable elements. You can create a set object by passing an iterable to the set constructor or using curly braces {}. For example:

```python
```

```
s = set("abc") # create a set from a string

print(s) # {'a', 'b', 'c'}

s = {"a", "b", "c"} # create a set using curly braces

print(s) # {'a', 'b', 'c'}
```

Tusk 10

hash table

A hash table is a data structure that is used to store and retrieve key-value pairs, where the key is used to calculate the index into an array, called a "bucket", that stores the associated value. The index is calculated using a hash function, which takes the key as input and outputs an integer value that is used to index into the bucket.

The hash function is designed to distribute keys uniformly across the array, so that each key maps to a unique index. However, collisions can occur when two or more keys map to the same index. To handle collisions, various techniques are used, such as open addressing or chaining.

In open addressing, when a collision occurs, the algorithm searches for the next available bucket in the array, starting from the index calculated by the hash function. In chaining, each bucket contains a linked list of key-value pairs that have mapped to the same index.

Hash tables have several advantages over other data structures, such as arrays or linked lists, including constant-time average-case complexity for search, insertion, and deletion operations. This makes them useful for implementing many algorithms and data structures, such as sets, maps, and caches.

However, hash tables also have some drawbacks, such as the potential for collisions, which can lead to performance degradation and the need for more complex collision resolution techniques. Additionally, hash tables can have high memory overhead due to the need for allocating the array and managing collisions.

hash map

A hash map is a specific implementation of a hash table, which is a data structure used to store and retrieve key-value pairs. The hash map uses a hash function to map keys to their associated values, and stores them in an array of buckets.

When a key-value pair is inserted into the hash map, the hash function is used to calculate the index of the bucket where the pair should be stored. If another pair already exists at that index, then a collision has occurred. In this case, the hash map may use different strategies to resolve the collision, such as chaining or open addressing.

To retrieve a value associated with a key from a hash map, the hash function is used to calculate the index of the bucket where the value may be stored. If a value is found at that index, then it is returned. If not, then the hash map assumes that the key is not present in the map.

Hash maps have several advantages over other data structures, such as arrays or linked lists, including constant-time average-case complexity for search, insertion, and deletion operations. This makes them useful for implementing many algorithms and data structures, such as sets, maps, and caches.

However, hash maps also have some drawbacks, such as the potential for collisions, which can lead to performance degradation and the need for more complex collision resolution techniques. Additionally, hash maps can have high memory overhead due to the need for allocating the array and managing collisions.