INSTRUCTIONS

- install dependencies

  `pip install -r requirements.txt`

- Crawling data

  `python3 scrawler.py`

  The time interval between each grab can be controlled by adjusting the `TimeGap` parameter.

- Build trees and analyze

  `python3 main.py`

TYPE HINTS

Since Python 3.6, Python supports static type checking, and with IDE, we can easily check possible type errors when writing code; in addition, we can also use `Protocol` for interface conventions (before the `abstractmethod` method is required).

In this project, we take full advantage of the capabilities of Type Annotation, here are some examples:

```
class Serializable(Protocol):
    """Protocol for serializable data types."""

    @abstractmethod
    def dump(self) -> Any:
        ...

    @abstractclassmethod
    def load(cls: Type['ST'], data: Any) -> 'ST':
```

```
        ...

class Comparable(Protocol):
    """Protocol for annotating comparable types."""

    @abstractmethod
    def __lt__(self: 'CT', other: 'CT', /) -> bool:
        ...

CT = TypeVar('CT', bound=Comparable)
ST = TypeVar('ST', bound=Serializable)
```

Through these interface classes, the chance of errors in our code is reduced.

DATA STRUCTURE

The basic B-Tree data structure is implemented in the `btree.py` file, which is often used to implement databases because of its good query performance, especially on traditional mechanical disks; this is because for B-Trees , each node is an ordered list, the data of this list will be stored on the disk in blocks, and for traditional mechanical disks, this will reduce the total seek time of the target data.

When querying the target data, it will first start from the root node to check whether the target data is in this node; when the target data is not queried, it will extend downward until it reaches the leaf node.

The time complexity of the query is:

- Iteration: $O(\log_m n)$
- Compare: $O(\log_2 m \cdot \log_m n)$

Here, because the data within each node is ordered, binary search optimization can be used.

When persistent storage is required, the `dump` function of the root node of the B-Tree can be called, which will recursively call the `dump` function of the child nodes and return a dictionary; due to the use of type hints, the data element class implements own `dump` function, so the dictionary can be saved as a file using the `json` library.

When the tree needs to be loaded, the recursive function is also used to build the leaf nodes from top to bottom.

MOVIE CLASS ABSTRACT

The abstraction of the movie class is implemented in the `movies.py` file, and redundant code is reduced by decorating the movie class with `dataclasses.dataclass`.

```
@dataclass
class Movie(Serializable):
    id: str
    name: str
    description: str
    rating: float
    votes: int
    year: int
    director: str
    stars: List[str]
    categories: List[str]
    duration: int
```

Meanwhile, the `Movie` class inherits the `Serializable` interface, which means that it can be indexed directly in the B-Tree by implementing the `dump` & `load` methods.

CRAWLER

In `scrawler.py`, with the help of `request` and `beautifulsoup` libraries, we implement a simple web crawler, which can directly obtain data from IMDb web pages. The reason for this is that most of IMDb's APIs need to be charged, and the price is not high. Philip.

The data captured here is: the top 1000 movie data in the IMDb database.

ANALYZE DATA

Using `matplotlib`, we performed a simple analysis of the captured data. The analysis items are:

- Top 5 movie actors - by total votes
- 5 most popular movies - by total votes
- Number of movies released each year
- Total audience votes per year
- Proportion of different movie categories