INSTITUTO DE INFORMÁTICA
DEPARTAMENTO DE INFORMÁTICA APLICADA

Gabriel Fernandes e Henrique Peixoto

Universidade Federal do Rio Grande do Sul - UFRGS

Trabalho Final - INF01124 - Classificação e Pesquisa de Dados

Porto Alegre
2021

Sumário

Visão Geral

Apresentação

Este trabalho destina-se a demonstrar a aplicação, de organizada e prática, dos algoritmos aprendidos durante a disciplina de Classificação e Pesquisa de Dados. Para tanto, foram utilizadas estruturas que são condizentes com sistemas de busca de dados em grande escala, tais como árvores, tabelas hash e arquivos invertidos.

Este trabalho foi desenvolvido utilizando a linguagem de programação Python. Essa escolha foi feita por conta desta linguagem ser bastante versátil e oferecer uma grande facilidade para a implementação e utilização de estruturas de dados mais complexas. Tal feito seria mais complexo se tivéssemos utilizado a linguagem C, por exemplo.

Além disso, vale ressaltar que foram utilizados Git juntamente com GitHub para que fosse mantida a organização do código e para que se pudesse acompanhar o desenvolvimento do mesmo, de modo a preservar e deixar explícito as contribuições de cada membro da equipe envolvida no projeto.

Explicação

O conjunto de dados utilizado refere-se a mais de 80.000 jogos presentes na plataforma online de venda de jogos Steam. Confira o conjunto de dados através deste link. As informações de cada jogo contém dados tais como nome, preço, categorias, produtora(s), desenvolvedora(s), data de lançamento e descrição. Esses dados são um compilado de informações verdadeiras que podem ser encontradas na Steam, apesar de que alguns dados estejam defasados por conta de mudanças de preço nos jogos assim como a possível retirada de alguns títulos da plataforma.

O aplicativo desenvolvido tem por objetivo organizar esses dados nas estruturas de dados que foram selecionadas, pois chegou-se à conclusão de que essas estruturas seriam as mais adequadas para o tratamentos desses dados, assim como para sua busca, atualização, deleção e ordenação.

Justificativa

Somente ter acesso a todos esses dados não implica, necessariamente, que sua utilização seja simples e cômoda, haja vista que muitas vezes esses dados estão desorganizados. E por conta disso foi desenvolvida esta aplicação: para que esses dados sejam organizados, através dos algoritmos utilizados, e para que esses dados possam ser utilizados de forma útil.

Importância do Problema

Em consonância com o que foi exposto acima, organizar dados de forma coerente e utilizável vem tornando-se cada vez mais importante, note, por exemplo, o conceito de Big Data: *“Big data são dados com maior variedade que chegam em volumes crescentes e com velocidade cada vez maior”* (Gardner, 2001).

Além disso, estamos rodeados por informações e dados de toda e qualquer natureza que nos são úteis, tais como datas e horários, previsão do tempo, GPS, notícias, dentre outros, e todos esses dados chegam até nós de forma organizada de maneira que seja fácil compreender o que está sendo exibido. O que este trabalho faz é entender a importância dos dados e a maneira como eles devem ser tratados e apresentados.

Implementação

Arquitetura do Código

O código foi organizado através da utilização do padrão MVC (Models, Views, Controllers), é um padrão de arquitetura de software focado no reuso de código e a separação de conceitos em três camadas interconectadas, onde a apresentação dos dados e interação dos usuários (front-end) são separados dos métodos que interagem com o banco de dados (back-end).

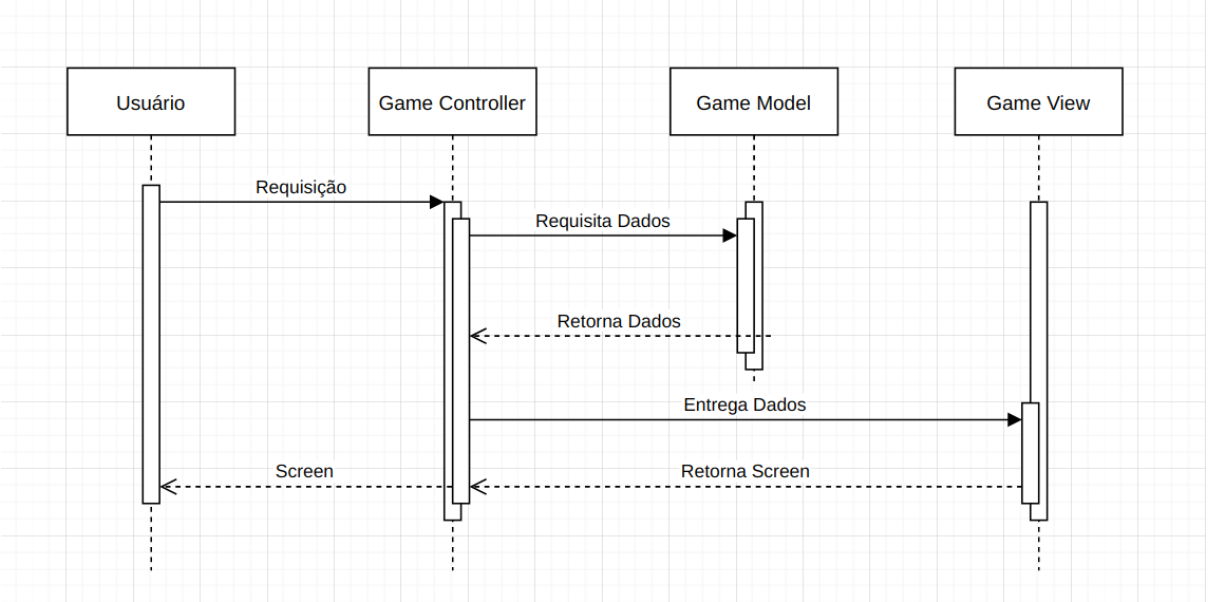


Figura 1: O padrão MVC.

Models, Views e Controllers são tipos de classes com responsabilidades definidas:

- As Classes do tipo Model são responsáveis pela interação com os dados, nestes projeto elas são as estruturas de dados criadas com os dados da Steam;
- As Classes do tipo Views são responsáveis por gerar uma screen dado um conjunto de dados pré-definido. Essa screen pode ser qualquer forma de interação fornecida para o usuário, seja ela gráfica, textual, sonora,...
- As Classes do tipo Controllers são responsáveis pelas ações que o usuário pode executar. São os métodos dos Controllers que, quando executados, usam as Models para pegar os dados, utilizam as Models para pegar os dados, entregam esses dados para uma View, e retorna a screen gerada para o usuário.

Na arquitetura MVC cada entidade da aplicação ganha o seu MVC, e dentro do controller da entidade são implementadas as ações de criação, leitura, atualização e deleção (CRUD) dos dados relativos àquela entidade, cada ação no seu próprio método.

O repositório do GitHub referente a aplicação pode ser encontrado através deste link.

Estruturas de Dados

As seguintes estruturas de dados foram utilizadas: árvore TRIE, arquivo invertido e conjuntos. O arquivo invertido é utilizado para fazer a consistência dos dados, já que uma vez extraídos devem ser formatados da maneira como se deseja e armazenados em outro arquivo. A árvore TRIE foi utilizada para armazenar atributos como o nome dos jogos, desenvolvedoras, categorias ou tags e colocar nas folhas, referentes a cada jogo, os ID dos registros onde esses nomes aparecem. Já os conjuntos foram utilizados como uma estrutura auxiliar para arquivo invertido. Esses detalhes de implementação serão melhor detalhados na subseção “Algoritmos Desenvolvidos”.

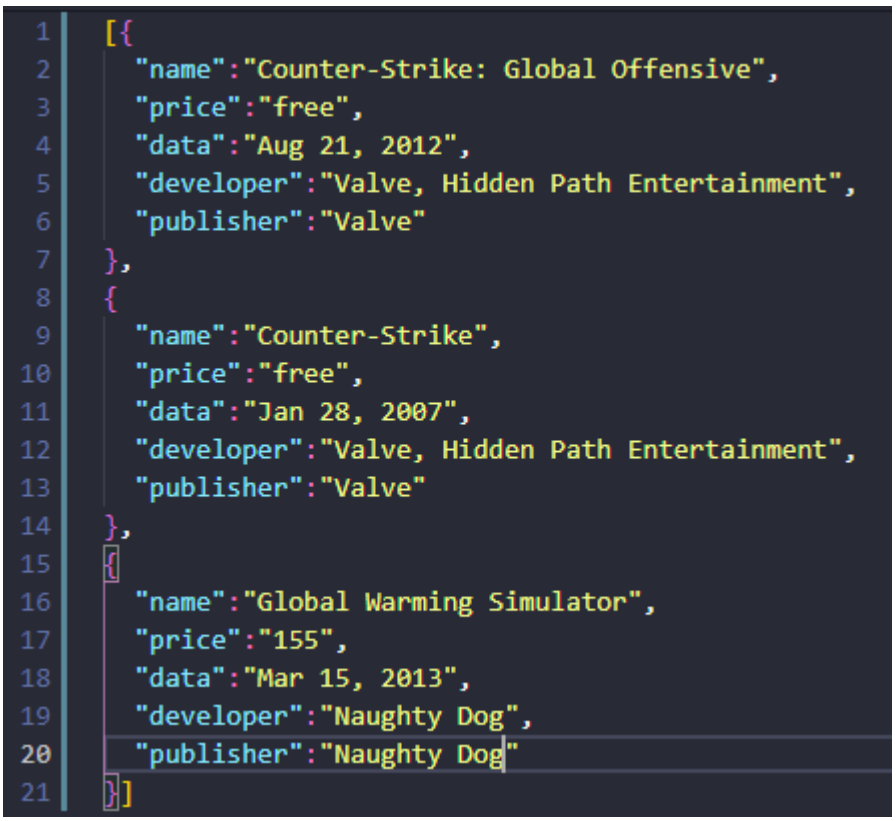
Arquivos utilizados

Foram usados um arquivo, aquele do qual são extraídos os dados sobre os jogos, esse link dá acesso ao arquivo em questão e que também é utilizado para persistir os dados.

Algoritmos desenvolvidos

Mostraremos, através de um exemplo, como é feita a extração e preparação dos dados, isso abrange o preenchimento dos conjuntos e da árvore TRIE bem como o salvamento dos dados. Os dados que serão usados neste exemplo estão presentes no conjunto de dados utilizado, por praticidade serão utilizados somente alguns registros do conjunto de dados e de maneira simplificada.

A primeira coisa a ser feita é a abertura do arquivo que contém os dados dos jogos. Esse arquivo não está organizado de modo legível, por isso é feito o parsing do mesmo no formato JSON (JavaScript Object Notation). Esse processo gera um array onde cada jogo é um dicionário (de acordo com a nomenclatura utilizada pela linguagem Python). Originalmente, esses dicionários não tem um campo “id”, então temos que percorrer este array e, para cada jogo, adicionar um campo “id”, que é determinado pela classe Entity. Suponha que o arquivo com os jogos tenha os seguintes dados:



```
[{
  "name":"Counter-Strike: Global Offensive",
  "price":"free",
  "data":"Aug 21, 2012",
  "developer":"Valve, Hidden Path Entertainment",
  "publisher":"Valve"
},
{
  "name":"Counter-Strike",
  "price":"free",
  "data":"Jan 28, 2007",
  "developer":"Valve, Hidden Path Entertainment",
  "publisher":"Valve"
},
{
  "name":"Global Warming Simulator",
  "price":"155",
  "data":"Mar 15, 2013",
  "developer":"Naughty Dog",
  "publisher":"Naughty Dog"
}]
```

a) Os dados sobre os jogos.

```
def create(self, item):
    item = super().create(item)

    for word in item["name"].strip().split():
        word = word.lower()
        if word not in self._names_table:
            # cria conjunto (set) de itemns
            self._names_table[word] = {item["id"]}
        else:
            # add o item em conjunto exitente
            self._names_table[word].add(item["id"])
    return item
```

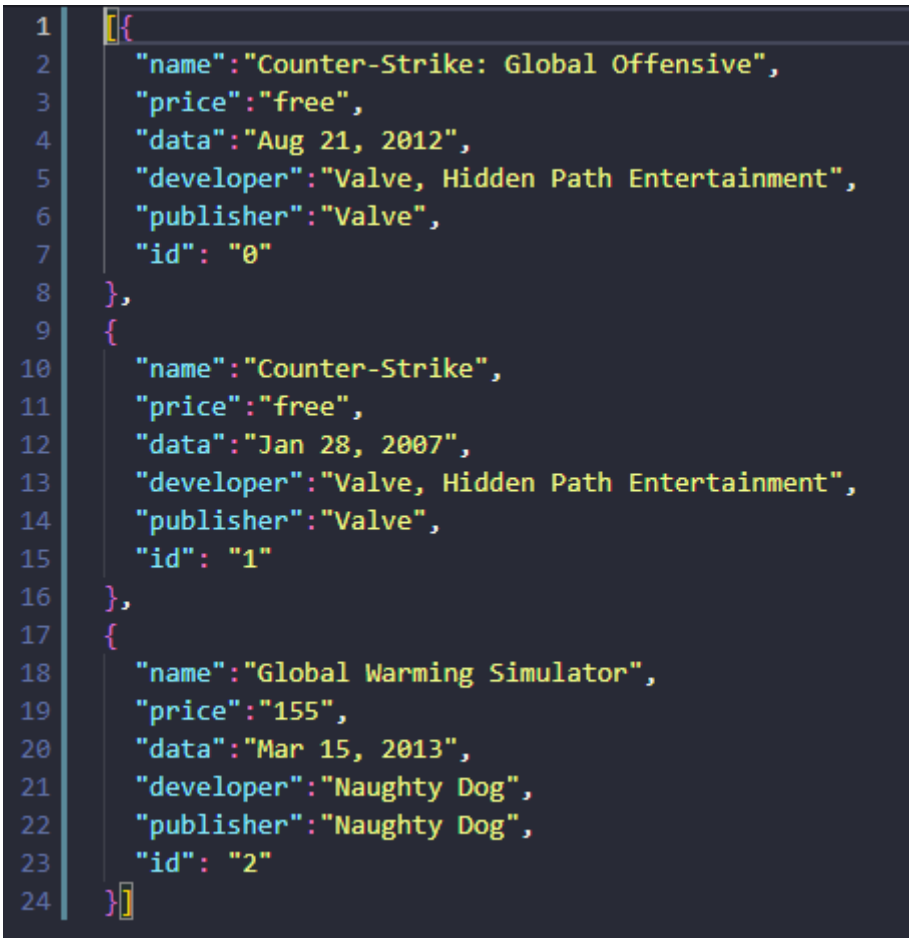
b) Algoritmo de criação dos conjuntos (game.py).

Figura 2: Preparação dos dados sobre os jogos.

Então, é selecionado o nome de cada jogo, executa-se o parsing do nome, dividindo-o através de seus espaços em branco, por exemplo, o jogo “Metal Gear Solid: Phantom Pain” após o processo de parsing torna-se o array [“metal”, “gear”, “solid”, “phantom”, “pain”], as palavras também são transformadas para minúsculas, para que se mantenha um padrão, assim como quebras de linha e “dois pontos” também são retirados.

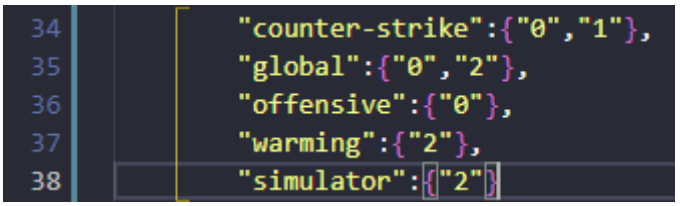
Dessa maneira vamos preenchendo os conjuntos, de modo que cada palavra, palavra essa que pertence ao nome de um jogo, é uma chave e o seu valor associado é um conjunto contendo todos os *IDs* dos jogos onde essa palavra aparece.

No momento em que inserimos um registro também o inserimos na TRIE, onde a folha, referente ao ID inserido, irá guardar os dados sobre o jogo associado ao ID em questão.



```
[{
  "name":"Counter-Strike: Global Offensive",
  "price":"free",
  "data":"Aug 21, 2012",
  "developer":"Valve, Hidden Path Entertainment",
  "publisher":"Valve",
  "id": "0"
},
{
  "name":"Counter-Strike",
  "price":"free",
  "data":"Jan 28, 2007",
  "developer":"Valve, Hidden Path Entertainment",
  "publisher":"Valve",
  "id": "1"
},
{
  "name":"Global Warming Simulator",
  "price":"155",
  "data":"Mar 15, 2013",
  "developer":"Naughty Dog",
  "publisher":"Naughty Dog",
  "id": "2"
}]
```

a) Dados sobre os jogos depois da adição dos IDs.



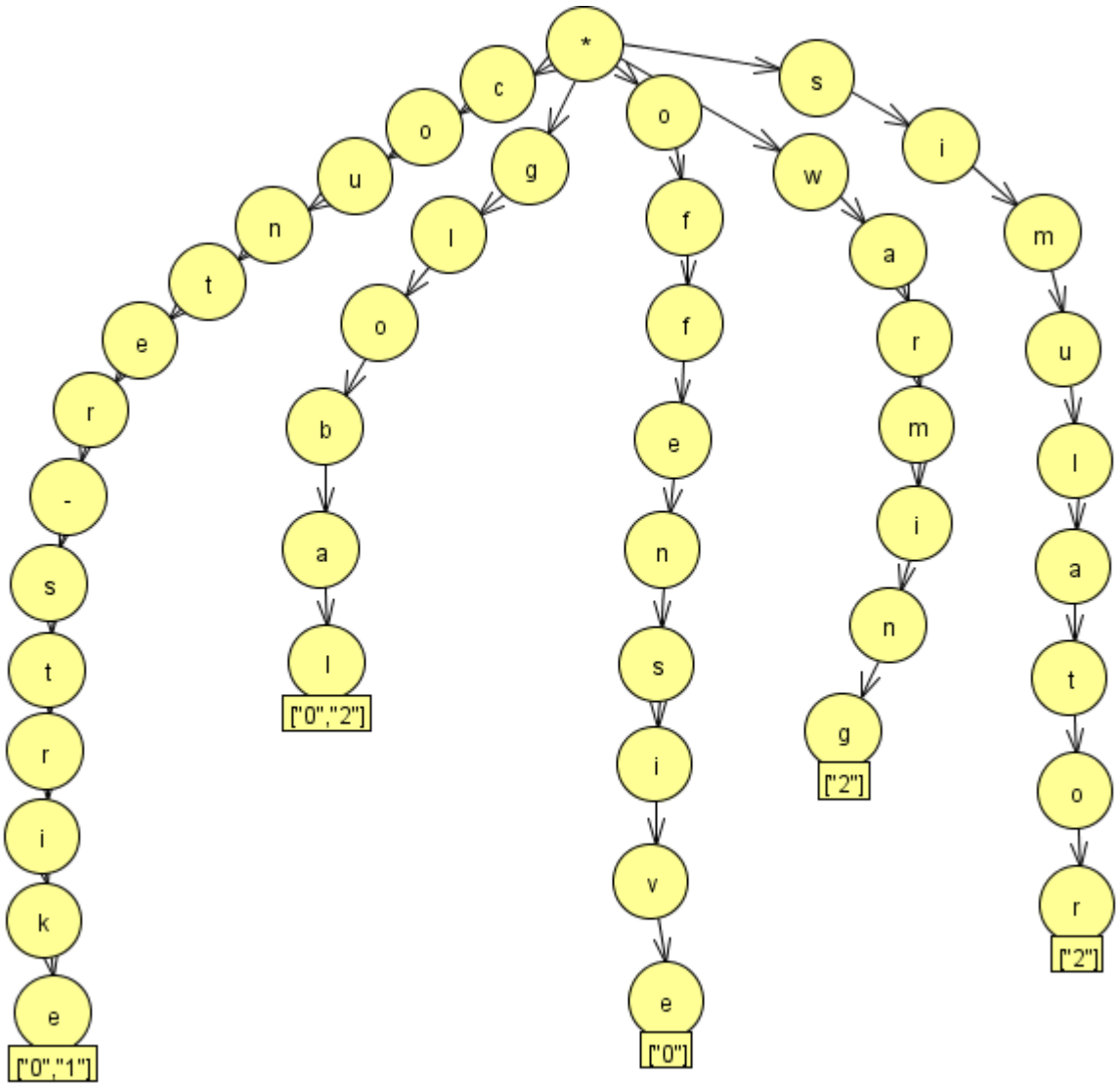
```
"counter-strike":{"0","1"},
"global":{"0","2"},
"offensive":{"0"},
"warming":{"2"},
"simulator":{"2"}
```

b) Criação dos conjuntos a partir dos nomes dos jogos.

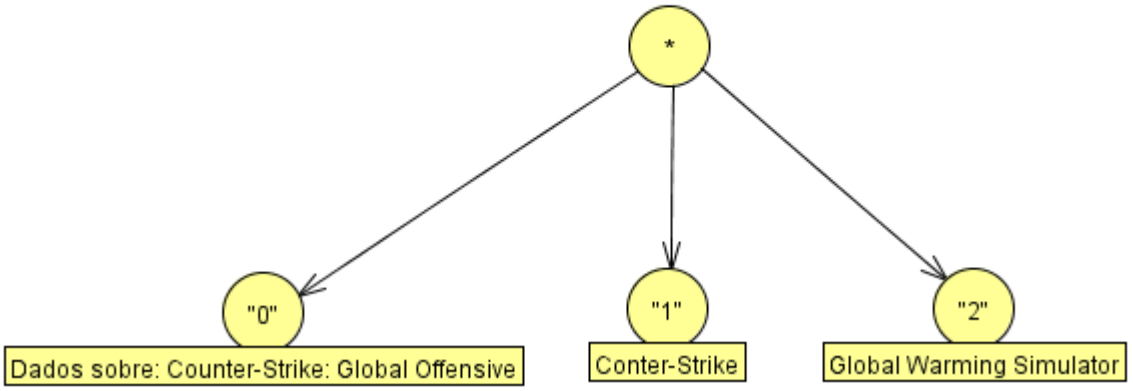
Figura 3: Criação dos conjuntos.

```python
    def __setitem__(self, word: str, id):
        """
            Método de Trie: adiciona uma palavra na TRIE.

            Param: add(word, id)
                word:str - palavra que será inserida
                id:int - id da palavra
        """
        word = self._standardize_string(word)
        node = self.root
        for char in word:
            found_in_child = False
            # Procura pelo caractere nos filhos do nodo atual
            for child in node.children:
                if child.char == char:
                    # Encontramos o caractere, incrementamos o contador pois
                    # agora temos mais uma palavra que tem o mesmo prefixo
                    child.counter += 1
                    # E vamos para o nodo filho onde econtramos o caractere de 'word'
                    node = child
                    found_in_child = True
                    break
            # Se não achamos o caractere de 'word' temos que adicioná-lo
            if not found_in_child:
                new_node = TrieNode(char,id)
                node.children.append(new_node)
                node = new_node

        # Quando terminarmos a palavra, marcamos o seu final
        node.word_finished = True
```

a) Algoritmo que implementa a TRIE (trie.py).



b) TRIE criada com base nos nomes dos jogos.



c) TRIE criada com base nos IDs dos jogos.

Figura 4: Criação da TRIE a partir dos conjuntos e algoritmo que implementa a TRIE.

Pela TRIE construída pode-se levar a pensar que não poderiam ter prefixos no meio de palavras ou que não ocorreriam ramificações na árvore. Abaixo são exibidas duas situações que a implementação dessa TRIE é capaz de suportar. Estes casos também podem ocorrer para outros elementos inseridos na TRIE (IDs, desenvolvedores, preços, dentre outros).

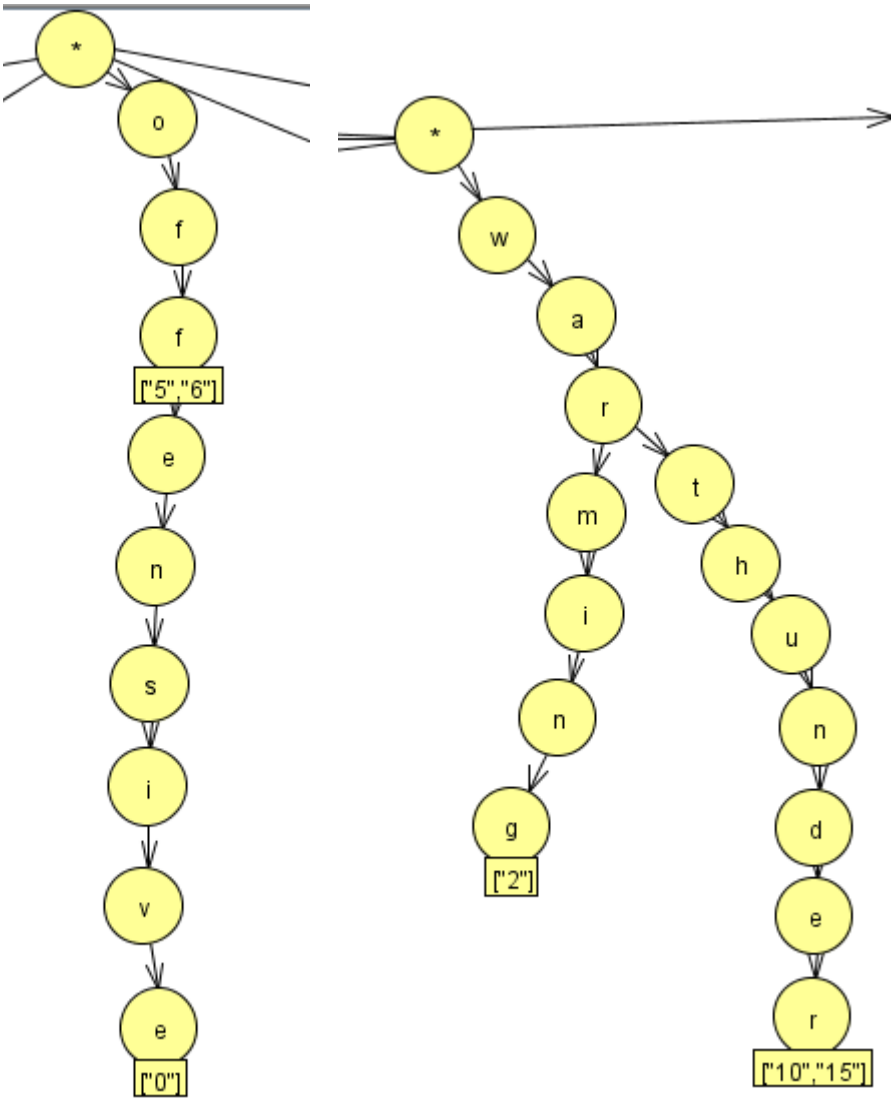


Figura 5: Outras possibilidades para a TRIE.

Guia de Uso

Características

Exemplos de Uso

- Inicialização das estruturas: precisamos importar as classes Game e Entity. Após isso, abrimos o arquivo que contém os jogos e os colocamos nas estruturas utilizadas.

```
from steam.models.game import Game
from steam.models.entity import Entity
```

a) Importando as classes Game e Entity.

```
Games = Game()
```

b) Instanciamos um objeto do tipo Game.

```
file_content = open("archive/steam_data.json")
file = file_content.read()
file = json.loads(file)
file_content.close()

file_content = open("archive/steam_data.json", "w")

for game in file:
    game = Games.create(game)

json.dump(file, file_content)
file_content.close()
```

c) Abrimos o arquivo onde estão os jogos e os inserimos nas estruturas, após isso, salvamos as alterações.

Figura 6: A inicialização das estruturas com os dados dos arquivos.

- Inserção de dados: para inserir um jogo precisamos usar o método *create* da classe Game. Primeiro, precisamos importar a classe Game e a classe Entity, de onde Game herda alguns métodos..

```
from steam.models.game import Game
from steam.models.entity import Entity
```

a) A importação da classe Game e da classe Entity.

Depois, precisamos criar um jogo; para isso devemos escrever pelo menos um nome para o jogo, o ID será escolhido pelo método *create.* Você pode adicionar outros atributos se preferir, tais como preço, desenvolvedora, produtora, descrição do jogo, tags, categorias e data de publicação. Caso algum desses atributos não seja preenchido, eles serão preenchidos pelo método *create*.

```
Games = Game()
```

b) Instanciamos um objeto do tipo Game.

```
28      a_game = {
29          "name":"War simulator",
30          "price":"10",
31          "publisher":"Valve",
32          "developer":"Sony",
33          "full_desc":"Who cares? Just gimme a good grade!"
34      }
```

c) O jogo a ser inserido.

```
a_game = Games.create(a_game)
```

d) Adicionando um jogo, o retorno da função é o jogo criado já com o ID.

```
file_content = open("archive/steam_data.json")
file = file_content.read()
file = json.loads(file)
file_content.close()

for game in file:
    game = Games.create(game)

game1 = Game.create(game1)
game2 = Game.create(game2)
...
gamen = Game.create(gamen)

file_content = open("archive/steam_data.json", "w")

json.dump(file, file_content)
file_content.close()
```

e) Salvando o jogo criado no arquivo.

Figura 7: Adição de um jogo.

Note que *file* também contém os dados dos jogos que foram utilizados para inicializar as estruturas, portanto, o que estamos fazendo é sobrescrever os dados sobre os jogos. Podemos inserir tantos jogos quanto forem necessários.

- Ordenamento de dados: após inserir os jogos, podemos ordenar esses dados. No primeiro exemplo, iremos ordenar os jogos através de seus nomes em ordem decrescente.

```
Games.sort_by("name")
games_sorted = Games.paginate(1,80893)

for game in games_sorted:
    print(game["name"])
```

a) Ordenando todos os jogos pelo nome.

Figura 8: Ordenando os jogos pelo nome em ordem crescente.

Agora iremos ordenar os jogos pelo preço em ordem crescente.

```
Games.sort_by("price", True)
games_sorted = Games.paginate(1,80893)

for game in games_sorted:
    print(game["price"])
```

a) Ordenando os jogos pelo preço.

Figura 9: Ordenando todos os jogos pelo preço.

Quando o método *sort_by não* recebe o valor booleano, ele ordena os dados em ordem crescente, mas quando o valor booleano passado é *False*, os dados são ordenados em ordem crescente.

Elementos de Terceiros

O algoritmo de ordenamento utilizado pelo Python é o Timsort. Vejamos a sua definição: "*Timsort is a hybrid stable sorting algorithm, derived from merge sort and insertion sort, designed to perform well on many kinds of real-world data. It was implemented by Tim Peters in 2002 for use in the Python programming language. The algorithm finds subsequences of the data that are already ordered (runs) and uses them to sort the remainder more efficiently. This is done by merging runs until certain criteria are fulfilled*"[1].

Considerações Finais

A realização deste trabalho é a culminação de um processo de aprendizado que ocorreu ao longo de um semestre e que teve o objetivo de explanar algoritmos muito importantes para a computação como um todo. É necessário que se tenha conhecimento desses algoritmos para que se possa aplicá-los nas situações corretas. Não só isso, mas também é preciso que se saiba analisar um processo, no caso deste trabalho, classificar e ordenar dados, e utilizar as estruturas de dados e os algoritmos mais adequados possíveis para a resolução deste processo, seja esta resolução visando um melhor desempenho da aplicação em detrimento da quantidade de memória consumida ou o caso contrário.

Este trabalho também revelou aos seus participantes as maiores e mais prestigiadas habilidades que concernem o pensamento computacional, a saber: abstração, reconhecimento de padrões, decomposição de problemas e a criação de algoritmos.

Além disso, é muito satisfatório entender como as estruturas usadas neste trabalho, árvore TRIE, conjuntos e arquivos, trabalham juntas para compor a aplicação.

A maior dificuldade foi trabalhar com os arquivos, por isso escolhemos um formato um tanto flexível, JSON, que facilitou bastante a manipulação dos dados.

Referências

[1] Timsort, Wikipedia. https://en.wikipedia.org/wiki/Timsort