ESTRUTURA DE DADOS DO TIPO ÁRVORE
ÁRVORE TRIE

O QUE É ÁRVORE (TRIE):

- Conhecida como árvore de prefixos (*prefix tree*);
- É um tipo de árvore de busca;
- Desenvolvida por *René de la Briandais* em 1959, com o nome dado por *Edward Fredkin* em 1961;
- Origem do Nome: reTRIEaval (recuperação), pois geralmente é usada na recuperação de dados, normalmente cadeias de caracteres;
- Utilizada para armazenar um array associativo;

CARACTERÍSTICAS (TRIE):

- Estrutura de dados do tipo árvore ordenada;
- A Árvore não é binária, então vários ramos podem ser criados;
- Ela busca o prefixo (raíz da cadeia de caracteres), podendo fazer sujestões em uma busca, por exemplo.
- Árvore multidirecional;
- Chaves são cadeias de caracteres;
- Possui sua raíz vazia com referências para próximos nós, um para cada possível caractere do alfabeto.
- O **Tamanho** da trie é diretamente relacionado com a quantidade de caracteres possíveis. Cada nó contém 2 coisas, um valor que pode ser nulo e um vetor de referências para os nós filhos.
- O **grau** da árvore corresponde ao tamanho do alfabeto;
- Os nós da árvore devem indicar quando completar uma chave;

COMPLEXIDADE DA ÁRVORE (TRIE):

M → Tamanho da Palavra
Alfabeto → 26 letras itálicas

- **Pior Caso:** O(26*M) = O(M)
- **Médio Caso:** O(M)
- **Melhor Caso:** O(1)

ESTRUTURA DE DADOS DO TIPO ÁRVORE
ÁRVORE TRIE

APLICAÇÕES:

- Dicionários;
- Recuperação de dados;
- Auxiliar de busca em pesquisas (Recursos Auto-Completar);
  - Correção de chaves de busca (Google);
- Corretores automáticos (Corretor ortográfico do Word);
- Solução de COLISÃO HASH: Árvore trie e tabela hash são bem parecidas, mas com algumas pequenas diferenças. Ambas usam vetores. Hash usa vetores combinados com listas linkadas. Trie usa vetores combinado com vetores/referências. O problema da tabela hash é que quando for inserir palavras com mesmo valores no vetor teremos o problema de colisão, na trie não acontece pois mesmo que 2 palavras comecem com a mesma letra, algum nó dessas palavras terão valores únicos para elas.

PONTOS POSITIVOS DA (TRIE):

- O grande trabalho de se criar uma trie está no inicio onde com poucas palavras, a toda nova inserção deverá ser criado várias referências para os possíveis próximos nós, porém quando já se tem uma grande quantidade de palavras inseridas fica cada vez mais fácil inserir novas palavras, pois a maioria das referências necessárias já existem, portanto é muito mais fácil inserir os “nós intermediários”. Outro ponto positivo é o fator de busca quando realizar operações na trie, visto que considerando o alfabeto do inglês, teremos somente 26 possíveis índices para procurar e isto não irá mudar considerando apenas um alfabeto, portando um valor constante no total de buscas.

PONTOS NEGATIVOS DA (TRIE):

- Cada nó contém 26 referências, e provavelmente nem todas nunca serão usadas, portando tem um grande custo de espaço em memória primaria e até secundária dependendo do caso.

TRIE BINÁRIA:

- Considera o alfabeto 0 e 1. Funciona do mesmo jeito, cada digito do número binário alocado em um nó. Esquerda e Direita: 0 inserido a esquerda, 1 inserido a direita. É como considera as letras através de números binários, vide tabela ASCII. Portanto cada chave pode ser convertida para binário. (Quando entra no assunto da trie binária chega na parte da arvore patrícia).

ESTRUTURA DE DADOS DO TIPO ÁRVORE
ÁRVORE TRIE

OPERAÇÕES (INTERFACE):

- **inicia()** // inicializa uma trie vazia
  - Toda Trie tem sua raiz vazia com referencias para os próximos nós, um para cada possível caractere do alfabeto. O tamanho da trie é diretamente relacionado com a quantidade de caracteres possíveis (26). Cada nó contém 2 coisas, um valor que pode ser nulo e um vetor de referências para os nós filhos.

- **busca()** // busca uma string na trie

  - Se o primeiro caractere não pertence a árvore, a chave não pertence a TRIE; Ou seja, ao buscar o primeiro caractere da chave desejada, verifica-se no vetor de referências do primeiro nó (raíz) se existe aquele caracter.

  Se existir:

  Verificar o próximo nó se existe o segundo caractere da chave procurara até que todos os caracteres pertençam em sequencia a TRIE, então, a chave pertence a árvore. Então, retornamos todo o "caminho" percorrido a partir da raíz para formar a palavra.

  Se não existir:

  Caso algum caractere não exista, a chave não pertence a TRIE. Então, retornamos nulo (NULL), garantindo que a palavra não existe na árvore.

- **insere()** // insere uma string ou caractere na trie

  - Faz-se uma **busca()** pela palavra a ser inserida.

    Caso exista:

    - Nada será feito.

    Caso não exista:

    - Procura o último nó **n** da substring da palavra a ser inserida e insere o restante dos caracteres a partir do nó **n**.

    * Caso não encontre nenhuma substring, a palavra é inserida por completa.

  - Verifica-se o vetor de referências da raiz, caso retorne nulo é criado um novo nó sendo referenciado pelo índice correto do vetor, exemplo Pá, caso não exista a letra P na arvore cria-se este nó, no índice 15 visto que P é a décima quinta letra do alfabeto. Após isto verifica-se se este novo nó retorna nulo para o próximo nó, se retornar cria-se o novo nó no índice 5 do vetor de referencias do nó P, pois a letra E é a quinta letra do alfabeto. **Esquerda ou Direita:** O que determina se será iniciado um

nó filho a esquerda ou a direita é o a ordem das letras. Menor a esquerda, maior a direita.

```
     P
     I
  A  E O      Exemplo a pie, pia e pio.
```

- **remove()** // remove uma string da trie
  - Busca a chave a ser removida.
  - A partir da filha (buttom-up), são removidos todos os nós que tem apenas um filho.
  - Busca por toda a palavra na arvore e muda o valor do indicie referente a letra para nulo, depois verifica-se este nó final aponta para outro nó que contém outra letra que forma outra palavra que também depende das letras da palavra excluída, caso não exista pode remover os nós sem problemas, caso exista então os nós da palavra devem permanecer e agora a ultima letra da palavra excluída não representa mais o “fim” do “caminho” mas sim o próximo nó.

- **triesize()** // retornar tamanho, em numero de cadeias armazenadas;

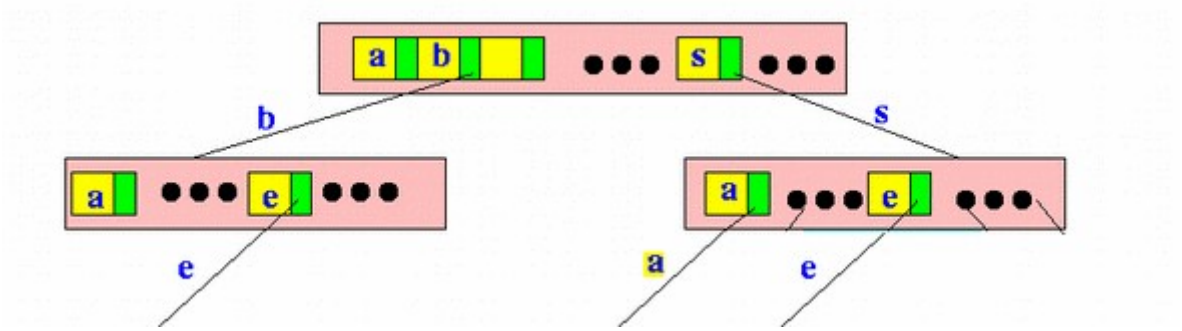
- **vazia()** // retorna se esta ou não vazia

- **exibeCadeias()** // exibir todas as cadeias armazenadas na trie;

IMPLEMENTAÇÃO (R-WAY):

- É a implementação mais símples;
- Cada nó contém todos os valores do alfabeto + 1 símbolo especial para determinar se é uma chave;

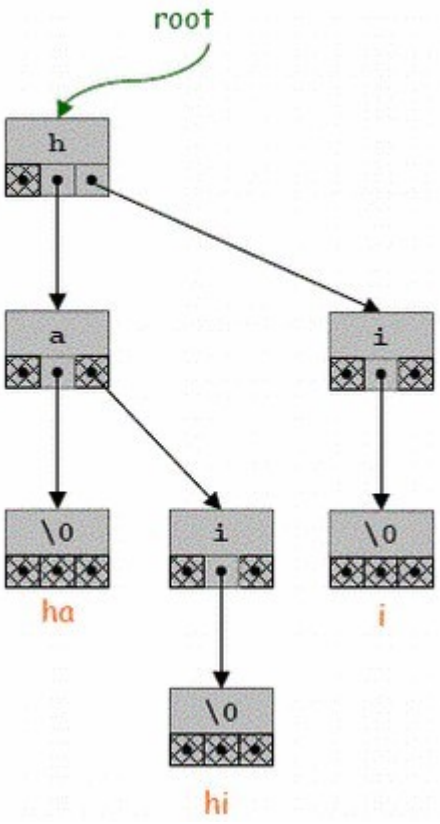
Ponto negativo: Desperdício de espaço.

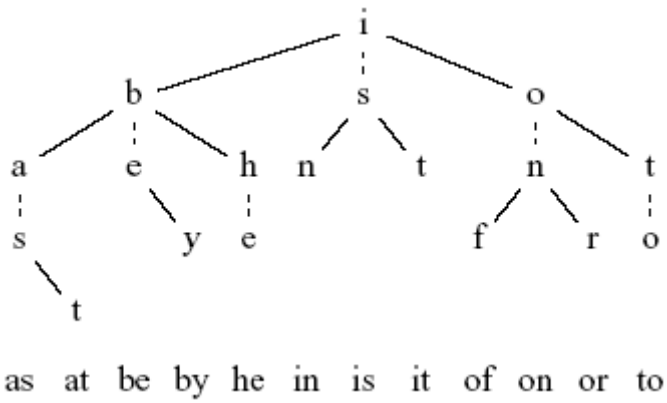


ESTRUTURA DE DADOS DO TIPO ÁRVORE
ÁRVORE TRIE

IMPLEMENTAÇÃO (TST):

- Ternary Search Tree (Árvore de busca composta por três partes);
- Soluciona o problema de desperdício de espaço;
- Cada nó aloca três ponteiros:
  Centro: caractere segunte.
  Filho a esquerda: caractere alternativo menor.
  Filho a direita: caractere alternativo maior.
- Incluindo root (raíz).
- Para a árvore ficar balanceada as chaves devem estar ordenadas.
- Insere as chaves a partir da chave central (semelhante a busca binária).





SITES (REFERÊNCIAS):

[1] https://medium.com/basecs/trying-to-understand-tries-3ec6bede0014
[2] https://pt.wikipedia.org/wiki/Trie
[3] https://en.wikipedia.org/wiki/Trie
[4] https://stackoverflow.com/questions/17891442/what-is-the-best-worst-average-case-big-o-runtime-of-a-trie-data-structu
[5] http://www.ufjf.br/jairo_souza/files/2009/12/6-Strings-Pesquisa-Digital.pdf
[6] https://people.ok.ubc.ca/ylucet/DS/Trie.html