AED2 - Aula 20 - Tries

Tries são árvores de busca digital em que toda chave está numa folha.

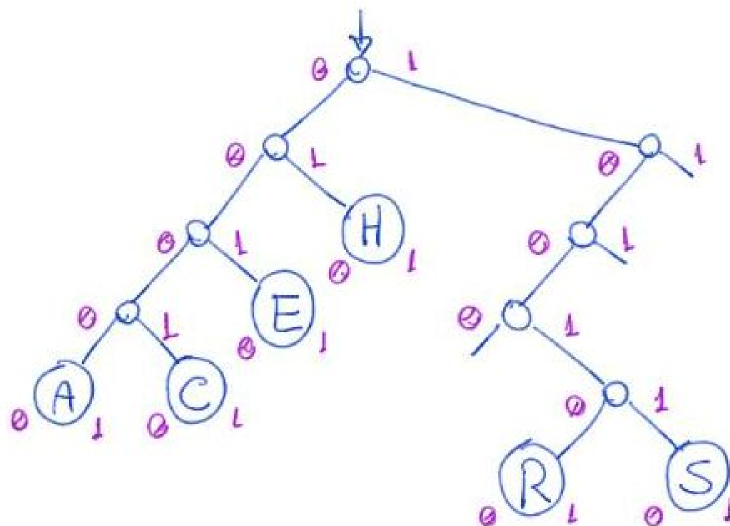
- Com isso, as chaves podem ser mantidas em ordem,
    - o que permite implementar de modo eficiente operações como
        - mínimo, máximo, predecessor, sucessor, percurso ordenado.
- Curiosamente, as operações máximo e mínimo, intimamente relacionadas
    - com ordem das chaves, podem ser implementadas eficientemente
        - nas árvores de busca digital básicas, ainda que
            - estas árvores não garantam a ordem das chaves.
                - Como? Por que?
- O nome trie vem de “information reTRIEval”,
    - mas pronunciamos “try” para diferenciar de “tree”.

Nos exemplos envolvendo tries,

- usaremos a seguinte representação binária de caracteres
- Os bits são numerados, a partir do 0, da esquerda para a direita.

	A 00001	B 00010	C 00011
D 00100	E 00101	F 00110	G 00111
H 01000	I 01001	J 01010	K 01011
L 01100	M 01101	N 01110	O 01111
P 10000	Q 10001	R 10010	S 10011
T 10100	U 10101	V 10110	W 10111
X 11000	Z 11001		

Exemplo de trie:



- uma propriedade central da trie é que todos os descendentes de um nó
    - tem prefixo comum com o daquele nó,
    - sendo que a raiz é associada com o prefixo vazio.
- uma característica única das tries entre as árvores de busca, é que
    - sua estrutura depende apenas das chaves que ela armazena,
        - e não da ordem em que elas foram inseridas.

Busca em trie:

- para buscar uma chave, basta percorrer o caminho na árvore
    - seguindo os bits da chave (0 desce à esquerda, 1 à direita).
- Se chegar numa folha, verificar se é a chave procurada.
    - Se for devolve o nó, caso contrário devolve falha da busca.
        - Exemplos na árvore anterior: buscar E (00101) ou D (00100).
- Se chegar num apontador vazio, devolve falha da busca.
    - Exemplo na árvore anterior: buscar T (10100).

Código da busca:

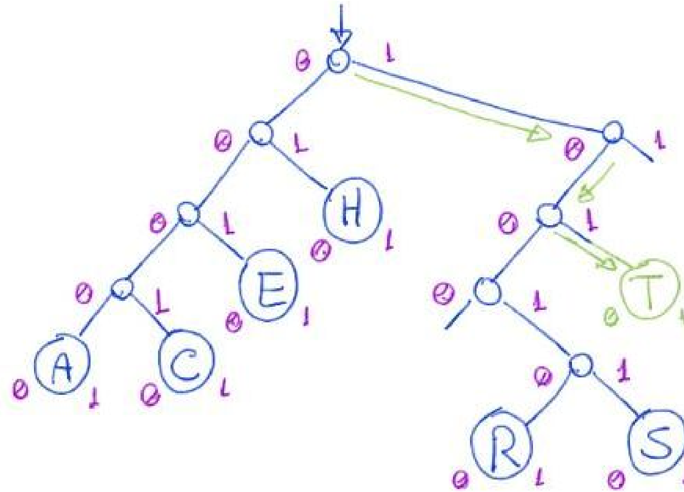
```
Noh *buscaR(Arvore r, Chave chave, int d, Noh **pai)
{
   if (r == NULL)
       return r;
   if (r->esq == NULL && r->dir == NULL) // eh uma folha
   {
       if (r->chave == chave)
           return r;
       return NULL;
   }
   if (digit(chave, d) == 0)
   {
       *pai = r;
       return buscaR(r->esq, chave, d + 1, pai);
   }
   // digit(chave, d) == 1
   *pai = r;
   return buscaR(r->dir, chave, d + 1, pai);
}
```

- Exemplo de uso

```
aux = buscaR(r, chaves[i], 0, &pai);
```

Exemplo de inserção do T (10100):

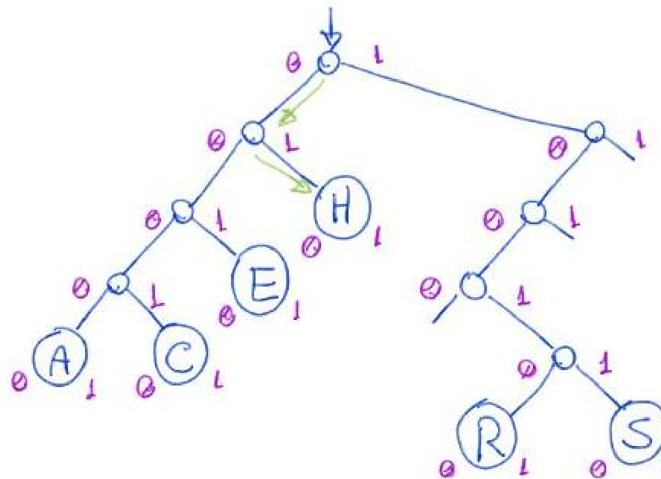
- Primeiro a chave T é buscada.



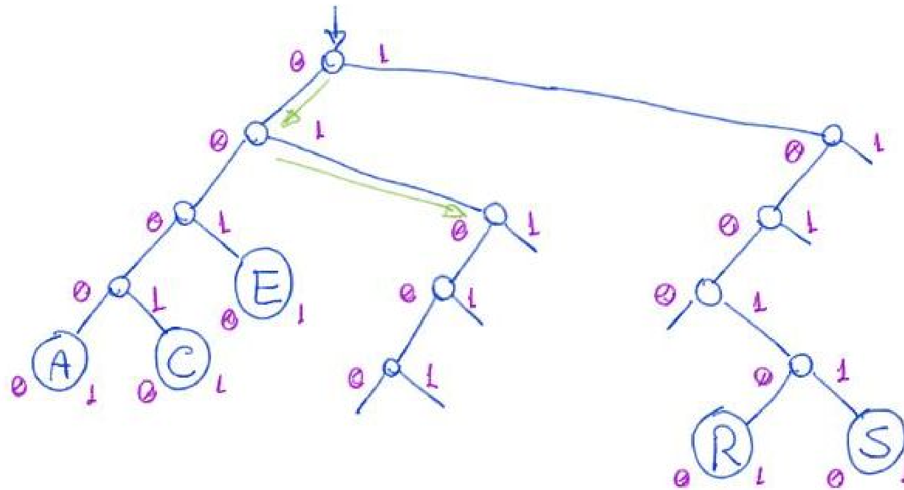
- Como a busca terminou em um apontador nulo de um nó interno,
    - basta substituir tal apontador pelo novo nó.

Exemplo de inserção do I (01001):

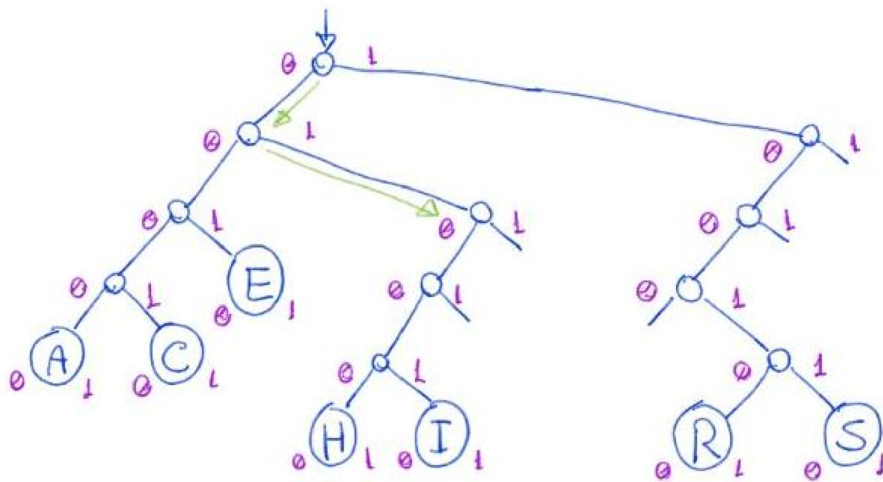
- Primeiro a chave I é buscada.



- Como a busca terminou em uma folha diferente de I
    - é necessário ramificar para separar as chaves.
- As chaves H (01000) e I (01001) coincidem nos 4 primeiros dígitos.
    - Por isso, é necessário criar nós internos até o nível 4.



- Então inserimos H e I de acordo com o valor de seu próximo bit,
    - que é o primeiro bit em que eles diferente (no caso, é o bit 5).



Códigos da inserção:

- Função que invoca a criação e um novo nó e manda inseri-lo na árvore

```
Arvore inserir(Arvore r, Chave chave, Item conteudo)
{
   Noh *novo = novoNoh(chave, conteudo);
   return insereR(r, novo, 0);
}
```

- Função que cria um novo nó

```
Noh *novoNoh(Chave chave, Item conteudo)
{
   Noh *novo;
   novo = (Noh *)malloc(sizeof(Noh));
   novo->chave = chave;
   novo->conteudo = conteudo;
   novo->esq = NULL;
   novo->dir = NULL;
   return novo;
```

```
}
```

- Função que insere recursivamente o novo nó na árvore

```
Arvore insereR(Arvore r, Noh *novo, int d)
{
   if (r == NULL)
   {
      return novo;
   }
   if (r->esq == NULL && r->dir == NULL)
   {
      return ramifique2(r, novo, d);
   }
   if (digit(novo->chave, d) == 0)
   {
      r->esq = insereR(r->esq, novo, d + 1);
   }
   else // digit(novo->chave, d) == 1
   {
      r->dir = insereR(r->dir, novo, d + 1);
   }
   return r;
}
```

- Função que faz a ramificação na árvore,
  - criando novos nós internos,
    - quando duas folhas p e q compartilham um prefixo.

```
Arvore ramifique(Noh *p, Noh *q, int d)
{
   Noh *inter;
   inter = (Noh *)malloc(sizeof(Noh));
   inter->chave = -1;                          // apenas para impressão
   if (digit(p->chave, d) == digit(q->chave, d)) // chaves não diferem no dígito d
   {
      if (digit(p->chave, d) == 0)
      {
         inter->dir = NULL;
         inter->esq = ramifique(p, q, d + 1);
      }
      else // digit(p->chave, d) == 1
      {
         inter->esq = NULL;
         inter->dir = ramifique(p, q, d + 1);
      }
   }
   else // chaves diferem no dígito d
   {
      if (digit(p->chave, d) == 0)
      {
         inter->esq = p;
```

```
            inter->dir = q;
        }
        else // digit(p->chave, d) == 1
        {
            inter->esq = q;
            inter->dir = p;
        }
    }
    return inter;
}
```

- Versão mais elegante da ramificação, que usa
    - manipulação de bits e um switch para decidir o que fazer.

```
Arvore ramifique2(Noh *p, Noh *q, int d)
{
    Noh *inter;
    inter = (Noh *)malloc(sizeof(Noh));
    inter->chave = -1; // apenas para impressão
    switch (digit(p->chave, d) * 2 + digit(q->chave, d))
    {
    // lembre qu em binário 0 = 00, 1 = 01, 2 = 10, 3 = 11
    case 0:
        inter->esq = ramifique2(p, q, d + 1);
        inter->dir = NULL;
        break;
    case 1:
        inter->esq = p;
        inter->dir = q;
        break;
    case 2:
        inter->esq = q;
        inter->dir = p;
        break;
    case 3:
        inter->dir = ramifique2(p, q, d + 1);
        inter->esq = NULL;
        break;
    }
    return inter;
}
```

Para a operação de remoção

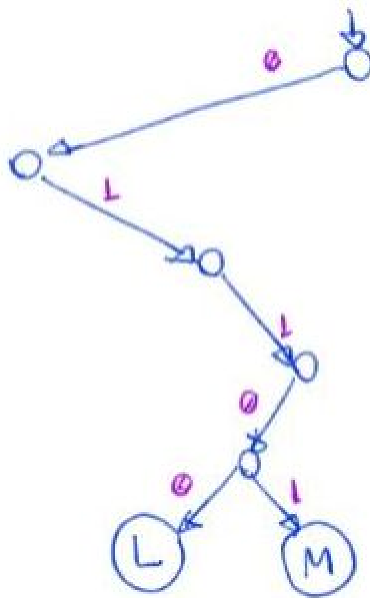
- podemos usar funções semelhantes àquelas da árvore digital básica,
    - atentando para o fato de que
        - essas não removem nós intermediários,
            - ainda que sejam removidas as folhas destes.
- Nesta situação, nós intermediários podem se tornar folhas,
    - o que precisa ser tratado.

Quanto à eficiência de tempo das operações, elas continuam sendo

- proporcionais à altura da árvore,
    - que no pior caso corresponde ao comprimento da chave,
        - i.e., ao número de dígitos da mesma.
- Já, se as chaves forem sequências aleatórias de bits,
    - o tempo esperado é proporcional a lg n,
        - sendo n o número de chaves.

Quanto à eficiência de espaço, note que

- uma trie pode precisar de muitos nós internos
    - para armazenar poucas folhas.



- De fato, desperdício de memória é um problema das tries.
    - Embora elas ocupem espaço proporcional ao número de itens,
        - se as chaves forem aleatórias.

Assim como fizemos com as árvores digitais básicas, podemos construir tries

- para tratar chaves que são strings ou que tem dígitos com mais de 1 bit.
- Neste caso o gasto de memória por nó cresce, pois cada nó terá
    - um vetor de filhos do tamanho do universo de valores que
        - os caracteres da string ou dígitos da chave podem assumir.

Uma versão mais sofisticada das tries, chamada Patricia Tries

- evita desperdiçar espaço, tem operações mais eficientes em tempo,
    - e pode ser usada para indexar chaves de tamanho variável.
- O termo PATRICIA é um acrônimo para
    - Practical Algorithm to Retrieve Information Coded in Alphanumeric.