

Advanced Topics in Data Mining and Logic Programming

Tabulação em Programação Lógica

Ricardo Rocha
DCC-FCUP, University of Porto
ricroc@dcc.fc.up.pt

Resolução SLD

- Apesar do poder, da flexibilidade e dos bons resultados que o Prolog tem demonstrado desde o aparecimento da WAM, a estratégia de resolução SLD na qual o Prolog se baseia é limitadora do potencial inerente ao paradigma da Programação Lógica.
- A resolução SLD não consegue tratar devidamente:
 - ◆ **Ciclos positivos infinitos** (expressividade insuficiente)
 - ◆ **Ciclos negativos infinitos** (inconsistência)
 - ◆ **Computações redundantes** (ineficiência)

Resolução SLD: Ciclos Infinitos

```
path(X,Z) :- path(X,Y), edge(Y,Z).  
path(X,Z) :- edge(X,Z).
```

```
edge(1,2).  
edge(2,1).
```

```
?- path(1,Z)  
  |  
?- path(1,Y), edge(Y,Z)  
  |  
ciclo infinito
```

Resolução SLD: Ciclos Infinitos

```
path(X,Z) :- edge(X,Y), path(Y,Z).  
path(X,Z) :- edge(X,Z).
```

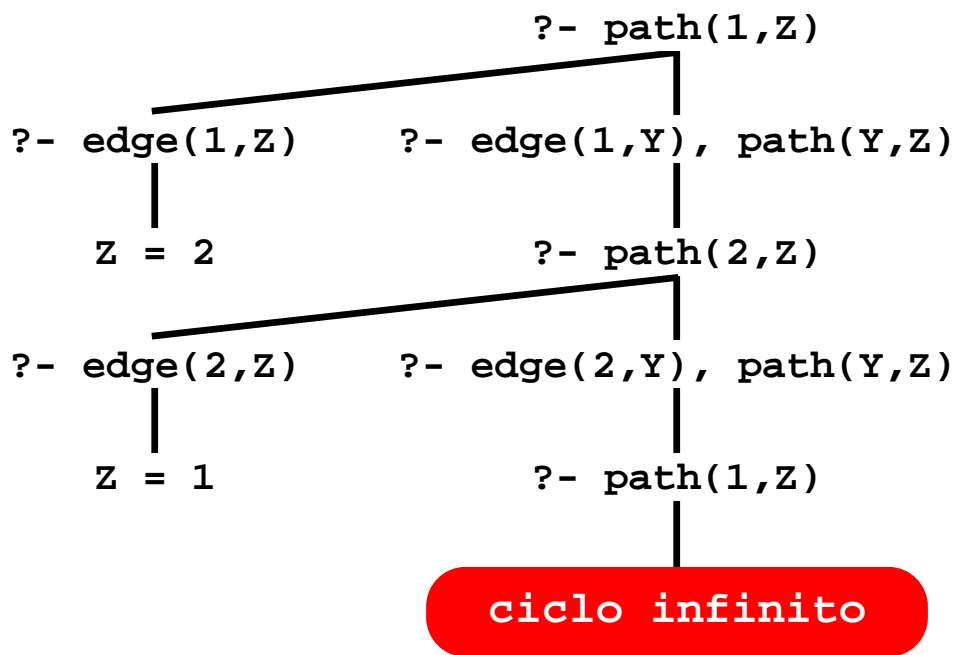
```
edge(1,2).  
edge(2,1).
```

```
?- path(1,Z)  
  |  
?- edge(1,Y), path(Y,Z)  
  |  
?- path(2,Z)  
  |  
?- edge(2,Y), path(Y,Z)  
  |  
?- path(1,Z)  
  |  
ciclo infinito
```

Resolução SLD: Ciclos Infinitos

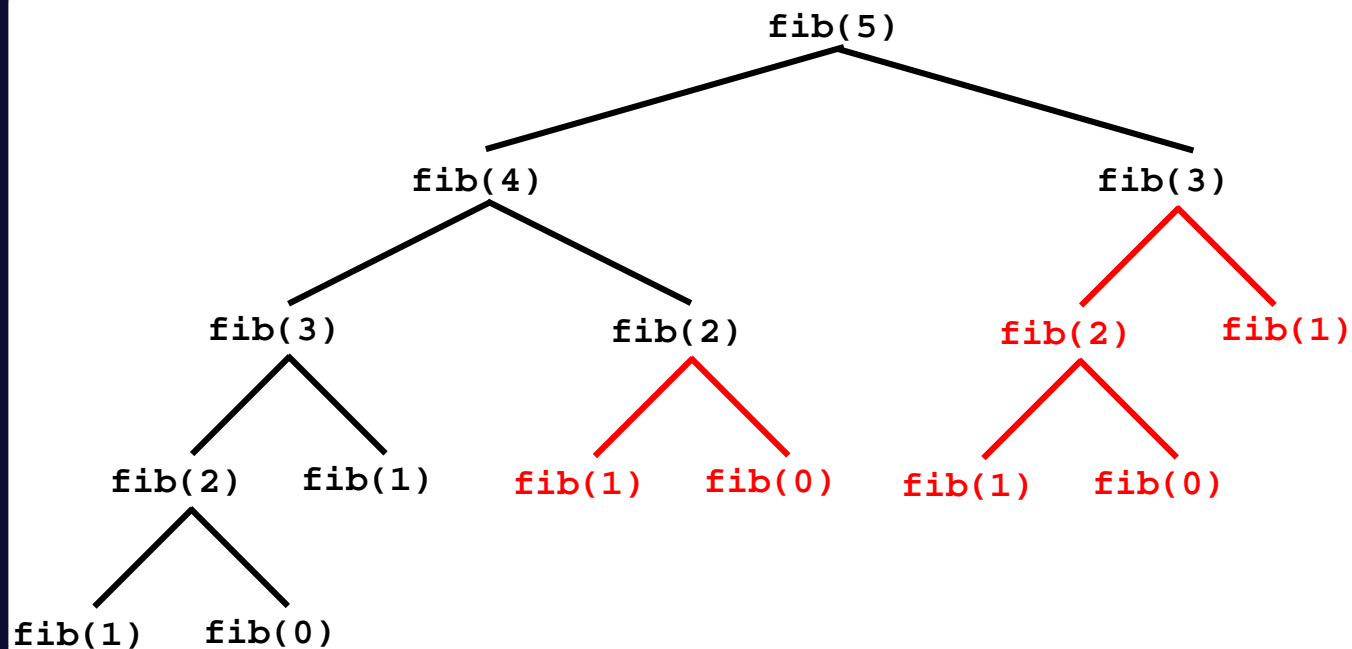
```
path(X,Z) :- edge(X,Z)
path(X,Z) :- edge(X,Y), path(Y,Z).

edge(1,2).
edge(2,1).
```



Resolução SLD: Computações Redundantes

```
fib(0,1).  
fib(1,1).  
fib(N,Z) :- P is N - 1,  
            Q is N - 2,  
            fib(P,X),  
            fib(Q,Y),  
            Z is X + Y.
```



Tabulação

É necessária uma estratégia de resolução diferente!

Tabulação

É necessária uma estratégia de resolução diferente!

- Uma das mais bem sucedidas técnicas para solucionar a incapacidade da resolução SLD no que respeita a ciclos infinitos e computações redundantes é a **Tabulação**.
- A ideia básica da tabulação consiste em ir guardando numa área auxiliar, a **tabela**, as soluções intermédias encontradas para determinados objectivos, para que estas possam ser reutilizadas assim que surgirem chamadas repetidas a esses mesmos objectivos.

Tabulação: Modelo Básico de Execução

- Predicados **não tabelados** são avaliados tal como em Prolog aplicando resolução SLD.

Tabulação: Modelo Básico de Execução

- Predicados **não tabelados** são avaliados tal como em Prolog aplicando resolução SLD.
- Sempre que um objectivo de um predicado **tabelado** é chamado pela **primeira vez**, adiciona-se à tabela uma nova entrada relativa a esse objectivo, e a execução prossegue aplicando resolução SLD.

Tabulação: Modelo Básico de Execução

- Predicados **não tabelados** são avaliados tal como em Prolog aplicando resolução SLD.
- Sempre que um objectivo de um predicado **tabelado** é chamado pela **primeira vez**, adiciona-se à tabela uma nova entrada relativa a esse objectivo, e a execução prossegue aplicando resolução SLD.
- Sempre que se obtém uma **nova solução** para um objectivo tabelado, esta é guardada na tabela.

Tabulação: Modelo Básico de Execução

- Predicados **não tabelados** são avaliados tal como em Prolog aplicando resolução SLD.
- Sempre que um objectivo de um predicado **tabelado** é chamado pela **primeira vez**, adiciona-se à tabela uma nova entrada relativa a esse objectivo, e a execução prossegue aplicando resolução SLD.
- Sempre que se obtém uma **nova solução** para um objectivo tabelado, esta é guardada na tabela.
- Chamadas repetidas de objectivos tabelados são resolvidas **consumindo** as soluções previamente encontradas e guardadas na tabela.

Tabulação: Modelo Básico de Execução

- Predicados **não tabelados** são avaliados tal como em Prolog aplicando resolução SLD.
- Sempre que um objectivo de um predicado **tabelado** é chamado pela **primeira vez**, adiciona-se à tabela uma nova entrada relativa a esse objectivo, e a execução prossegue aplicando resolução SLD.
- Sempre que se obtém uma **nova solução** para um objectivo tabelado, esta é guardada na tabela.
- Chamadas repetidas de objectivos tabelados são resolvidas **consumindo** as soluções previamente encontradas e guardadas na tabela.
- Assim que todas as soluções forem consumidas, o estado da computação é **suspenso** e a execução falha. Entretanto, se durante a exploração do objectivo em causa surgirem novas soluções, então a computação suspensa é **retomada** para consumir as novas soluções. Caso contrário, é considerada **completa**.

Tabulação: Modelo Básico de Execução

- Predicados **não tabelados** são avaliados tal como em Prolog aplicando resolução SLD.
- Sempre que um objectivo de um predicado **tabelado** é chamado pela **primeira vez**, adiciona-se à tabela uma nova entrada relativa a esse objectivo, e a execução prossegue aplicando resolução SLD.
- Sempre que se obtém uma **nova solução** para um objectivo tabelado, esta é guardada na tabela.
- Chamadas repetidas de objectivos tabelados são resolvidas **consumindo** as soluções previamente encontradas e guardadas na tabela.
- Assim que todas as soluções forem consumidas, o estado da computação é **suspenso** e a execução falha. Entretanto, se durante a exploração do objectivo em causa surgirem novas soluções, então a computação suspensa é **retomada** para consumir as novas soluções. Caso contrário, é considerada **completa**.
- A computação **termina** assim que todas as soluções tiverem sido consumidas e não for possível continuar a execução por aplicação da resolução SLD.

Tabulação: Ciclos Infinitos

0. path(1,Z)

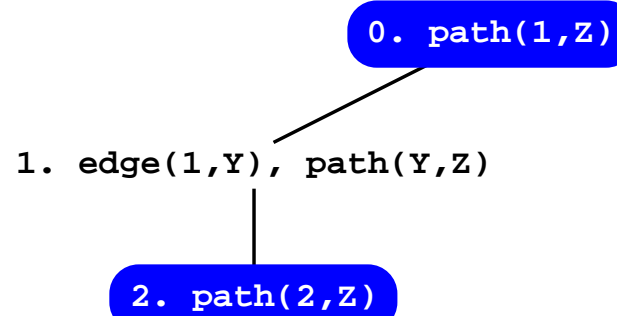
Programa

```
:- table path/2.  
  
path(X,Z):- edge(X,Y),  
             path(Y,Z).  
path(X,Z):- edge(X,Z).  
  
edge(1,2).  
edge(2,1).
```

Tabela

Objectivo	Soluções
0. path(1,Z)	

Tabulação: Ciclos Infinitos



Programa

```

:- table path/2.

path(X,Z):- edge(X,Y),
             path(Y,Z).
path(X,Z):- edge(X,Z).

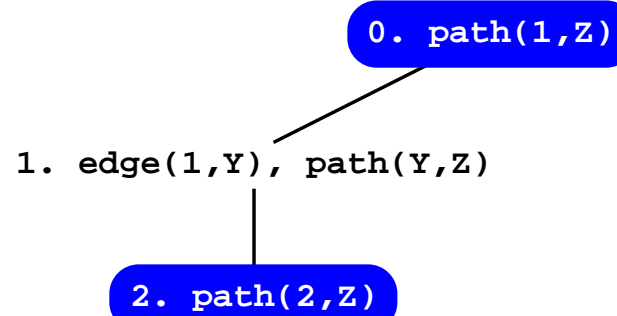
edge(1,2).
edge(2,1).
  
```

2. path(2,Z)

Tabela

Objectivo	Soluções
0. path(1,Z)	
2. path(b,Z)	

Tabulação: Ciclos Infinitos

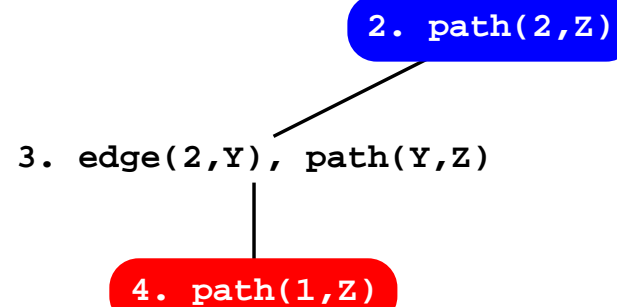


Programa

```
:- table path/2.

path(X,Z):- edge(X,Y),
             path(Y,Z).
path(X,Z):- edge(X,Z).

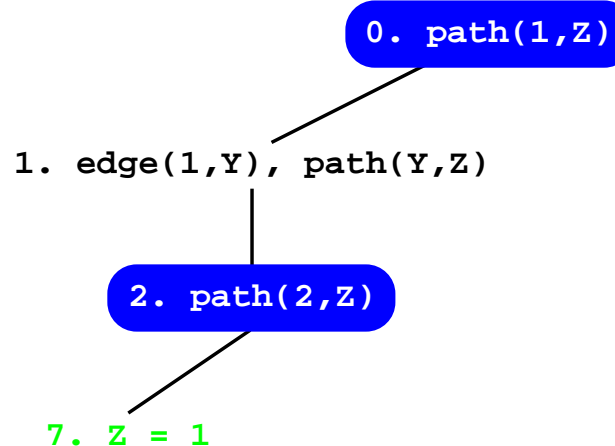
edge(1,2).
edge(2,1).
```



Tabela

Objectivo	Soluções
0. path(1,Z)	
2. path(b,Z)	

Tabulação: Ciclos Infinitos



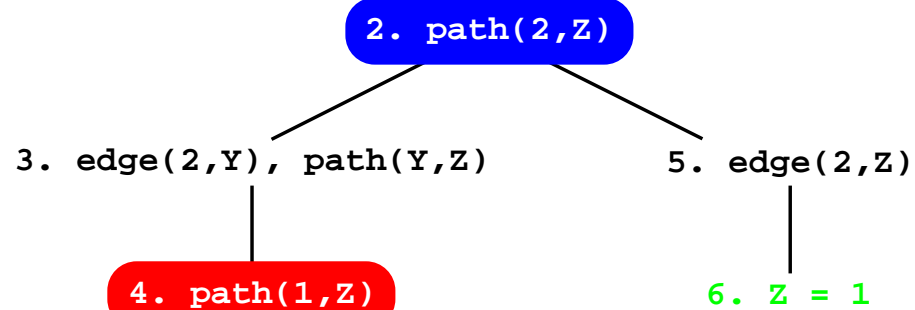
Programa

```

:- table path/2.

path(X,Z):- edge(X,Y),
             path(Y,Z).
path(X,Z):- edge(X,Z).

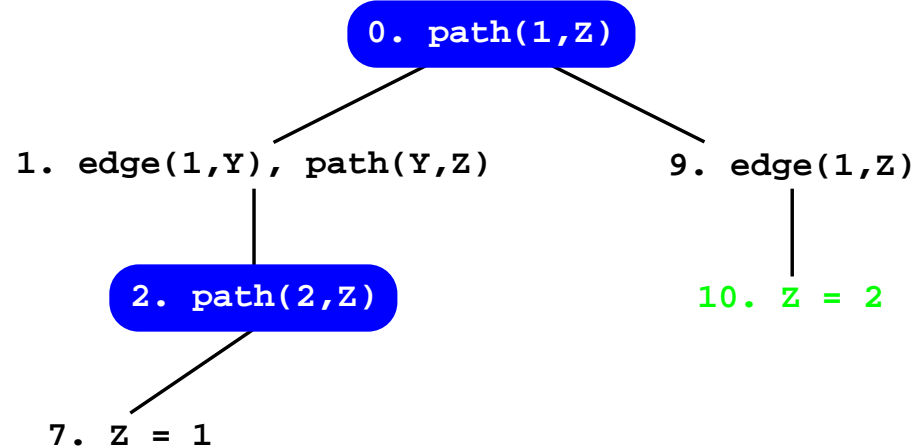
edge(1,2).
edge(2,1).
  
```



Tabela

Objectivo	Soluções
0. path(1,Z)	7. Z = 1
2. path(b,Z)	6. Z = 1

Tabulação: Ciclos Infinitos



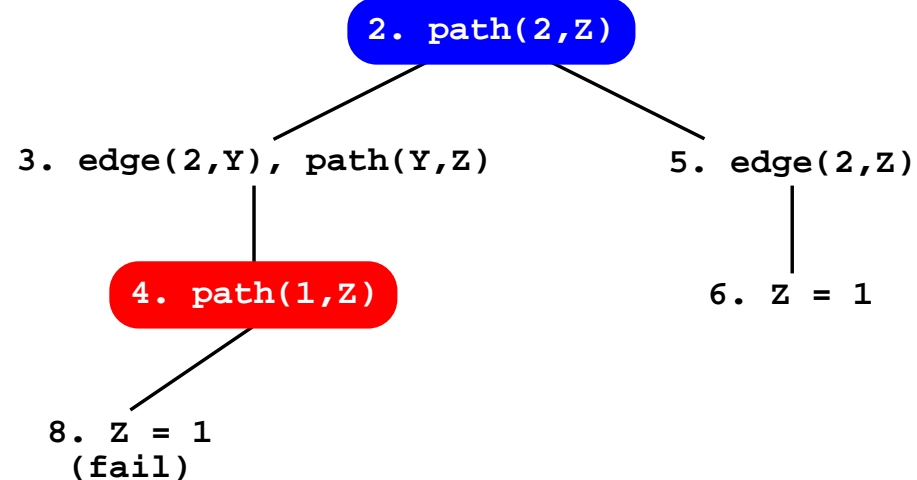
Programa

```

:- table path/2.

path(X,Z):- edge(X,Y),
             path(Y,Z).
path(X,Z):- edge(X,Z).

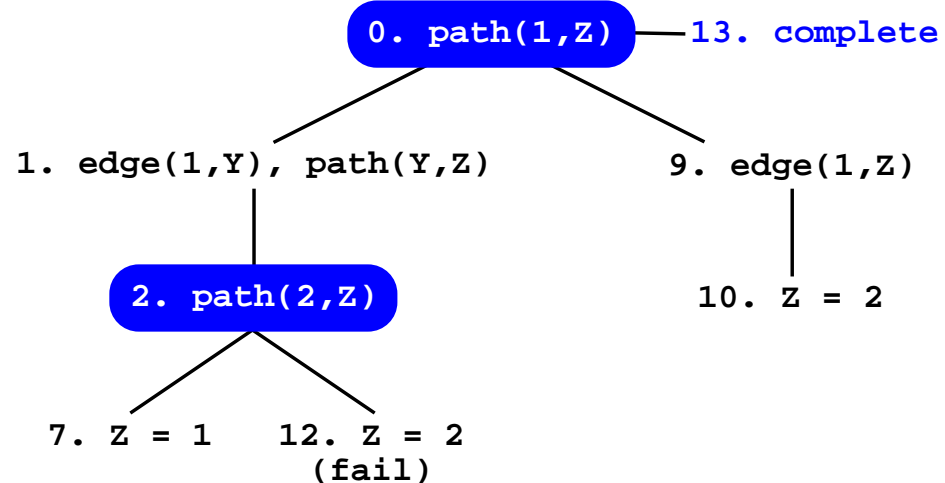
edge(1,2).
edge(2,1).
  
```



Tabela

Objectivo	Soluções
0. path(1,Z)	7. Z = 1 10. Z = 2
2. path(b,Z)	6. Z = 1

Tabulação: Ciclos Infinitos



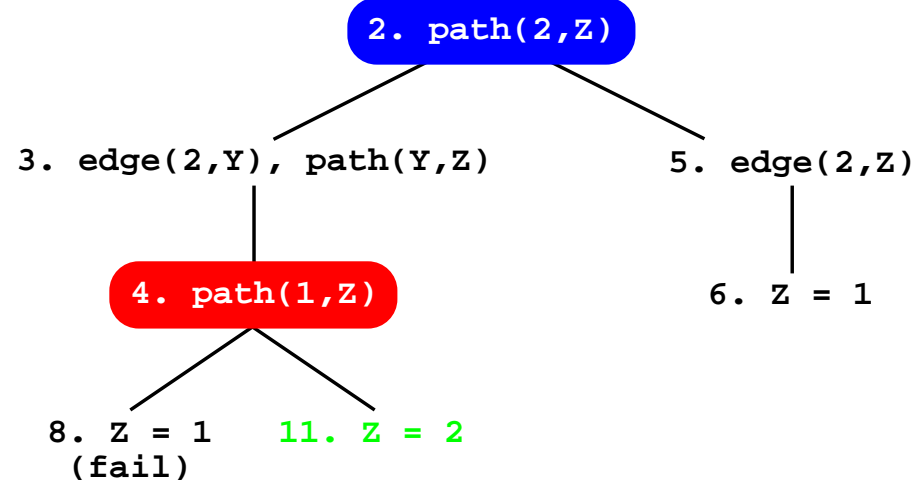
Programa

```

:- table path/2.

path(X,Z):- edge(X,Y),
             path(Y,Z).
path(X,Z):- edge(X,Z).

edge(1,2).
edge(2,1).
  
```

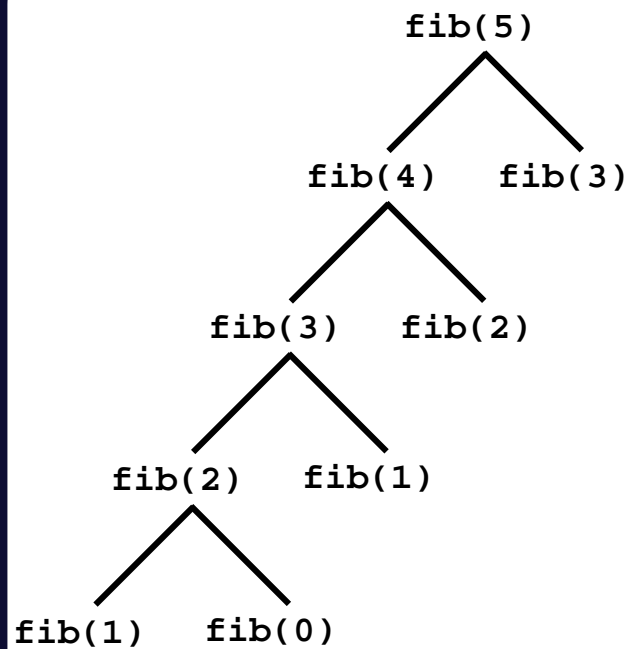


Tabela

Objectivo	Soluções
0. path(1,Z)	7. Z = 1 10. Z = 2 13. complete
2. path(b,Z)	6. Z = 1 11. Z = 2 13. complete

Tabulação: Computações Redundantes

```
fib(0,1).  
fib(1,1).  
fib(N,Z) :- P is N - 1,  
            Q is N - 2,  
            fib(P,X),  
            fib(Q,Y),  
            Z is X + Y.
```



Tabulação: Completude e Líderes

- Classificação dos nós da árvore de procura:
 - ◆ **Interiores:** representam os objectivos não tabelados.
 - ◆ **Geradores:** representam as primeiras chamadas a objectivos tabelados.
 - ◆ **Consumidores:** representam as chamadas repetidas a objectivos tabelados.

Tabulação: Completude e Líderes

- Classificação dos nós da árvore de procura:
 - ◆ **Interiores:** representam os objectivos não tabelados.
 - ◆ **Geradores:** representam as primeiras chamadas a objectivos tabelados.
 - ◆ **Consumidores:** representam as chamadas repetidas a objectivos tabelados.
- Um objectivo tabelado está **completamente avaliado** quando:
 - ◆ Não podem ser geradas mais soluções por resolução SLD.
 - ◆ Todas as chamadas ao objectivo consumiram todas as soluções encontradas.

Tabulação: Completude e Líderes

- Classificação dos nós da árvore de procura:
 - ◆ **Interiores:** representam os objectivos não tabelados.
 - ◆ **Geradores:** representam as primeiras chamadas a objectivos tabelados.
 - ◆ **Consumidores:** representam as chamadas repetidas a objectivos tabelados.
- Um objectivo tabelado está **completamente avaliado** quando:
 - ◆ Não podem ser geradas mais soluções por resolução SLD.
 - ◆ Todas as chamadas ao objectivo consumiram todas as soluções encontradas.
- Um conjunto de objectivos pode ser mutuamente dependente, **strongly connected component (SCC)**, e nesse caso só pode ser completo em simultâneo.
- Um SCC está completamente avaliado quando cada objectivo do SCC está completamente avaliado. A completude de um SCC deve ser realizada pelo **líder** do SCC, isto é, o gerador associado ao objectivo mais antigo do SCC.

Tabulação: Principais Modelos de Execução

➤ Tabulação por Suspensão da Computação

- ◆ A execução é vista como uma sequência de sub-computações que podem ser suspensas e mais tarde recuperadas até se atingir um ponto-fixo.
- ◆ A suspensão das sub-computações é conseguida por congelamento das pilhas de execução, por cópia das pilhas de execução para áreas auxiliares, ou por combinação de ambas as técnicas.

Tabulação: Principais Modelos de Execução

➤ Tabulação por Suspensão da Computação

- ◆ A execução é vista como uma sequência de sub-computações que podem ser suspensas e mais tarde recuperadas até se atingir um ponto-fixo.
- ◆ A suspensão das sub-computações é conseguida por congelamento das pilhas de execução, por cópia das pilhas de execução para áreas auxiliares, ou por combinação de ambas as técnicas.

➤ Tabulação Linear

- ◆ A execução é vista como uma única computação onde os objectivos tabelados são recursivamente reavaliados até se atingir um ponto-fixo sem que para isso seja necessário suspender/recuperar sub-computações.
- ◆ O ponto fraco destes modelos é a necessidade de utilizar re-computação para calcular pontos-fixos.

Tabulação: Sistemas Mais Conhecidos

- **Suspensão por Congelamento das Pilhas de Execução**
 - ◆ XSB Prolog (modelo SLG-WAM)
 - ◆ YapTab
- **Suspensão por Cópia das Pilhas de Execução**
 - ◆ XSB Prolog (modelo CAT)
 - ◆ Mercury
- **Suspensão por Congelamento e Cópia das Pilhas de Execução**
 - ◆ XSB Prolog (modelo CHAT)
- **Tabulação Linear**
 - ◆ ALS-Prolog (modelo DRA)
 - ◆ B-Prolog (modelo SLDT)

YapTab: Extensões de Suporte

- **Tabela:** espaço para guardar os objectivos tabelados e respectivas soluções.

YapTab: Extensões de Suporte

- **Tabela:** espaço para guardar os objectivos tabelados e respectivas soluções.
- **Suspensão/Recuperação:** mecanismos que permitam suspender e recuperar o estado da computação.

YapTab: Extensões de Suporte

- **Tabela:** espaço para guardar os objectivos tabelados e respectivas soluções.
- **Suspensão/Recuperação:** mecanismos que permitam suspender e recuperar o estado da computação.
- **Novas instruções:** instruções de suporte às 4 operações básicas.

YapTab: Extensões de Suporte

- **Tabela:** espaço para guardar os objectivos tabelados e respectivas soluções.
- **Suspensão/Recuperação:** mecanismos que permitam suspender e recuperar o estado da computação.
- **Novas instruções:** instruções de suporte às 4 operações básicas.
 - ◆ **Tabled Subgoal Call:** verifica se um objectivo já existe na tabela, e senão insere-o e cria um novo nó gerador. Caso contrário, cria um nó consumidor e começa a consumir as soluções disponíveis.

YapTab: Extensões de Suporte

- **Tabela:** espaço para guardar os objectivos tabelados e respectivas soluções.
- **Suspensão/Recuperação:** mecanismos que permitam suspender e recuperar o estado da computação.
- **Novas instruções:** instruções de suporte às 4 operações básicas.
 - ◆ **Tabled Subgoal Call:** verifica se um objectivo já existe na tabela, e senão insere-o e cria um novo nó gerador. Caso contrário, cria um nó consumidor e começa a consumir as soluções disponíveis.
 - ◆ **New Answer:** verifica se uma nova solução já existe na tabela, e senão insere-a.

YapTab: Extensões de Suporte

- **Tabela:** espaço para guardar os objectivos tabelados e respectivas soluções.
- **Suspensão/Recuperação:** mecanismos que permitam suspender e recuperar o estado da computação.
- **Novas instruções:** instruções de suporte às 4 operações básicas.
 - ◆ **Tabled Subgoal Call:** verifica se um objectivo já existe na tabela, e senão insere-o e cria um novo nó gerador. Caso contrário, cria um nó consumidor e começa a consumir as soluções disponíveis.
 - ◆ **New Answer:** verifica se uma nova solução já existe na tabela, e senão insere-a.
 - ◆ **Answer Resolution:** consome a próxima solução disponível, se alguma. Caso contrário, suspende a computação e calcula uma possível resolução para continuar a execução.

YapTab: Extensões de Suporte

- **Tabela:** espaço para guardar os objectivos tabelados e respectivas soluções.
- **Suspensão/Recuperação:** mecanismos que permitam suspender e recuperar o estado da computação.
- **Novas instruções:** instruções de suporte às 4 operações básicas.
 - ◆ **Tabled Subgoal Call:** verifica se um objectivo já existe na tabela, e senão insere-o e cria um novo nó gerador. Caso contrário, cria um nó consumidor e começa a consumir as soluções disponíveis.
 - ◆ **New Answer:** verifica se uma nova solução já existe na tabela, e senão insere-a.
 - ◆ **Answer Resolution:** consome a próxima solução disponível, se alguma. Caso contrário, suspende a computação e calcula uma possível resolução para continuar a execução.
 - ◆ **Completion:** determina se um objectivo tabelado está completamente avaliado, e senão calcula uma possível resolução para continuar a execução.

YapTab: Representação da Tabela

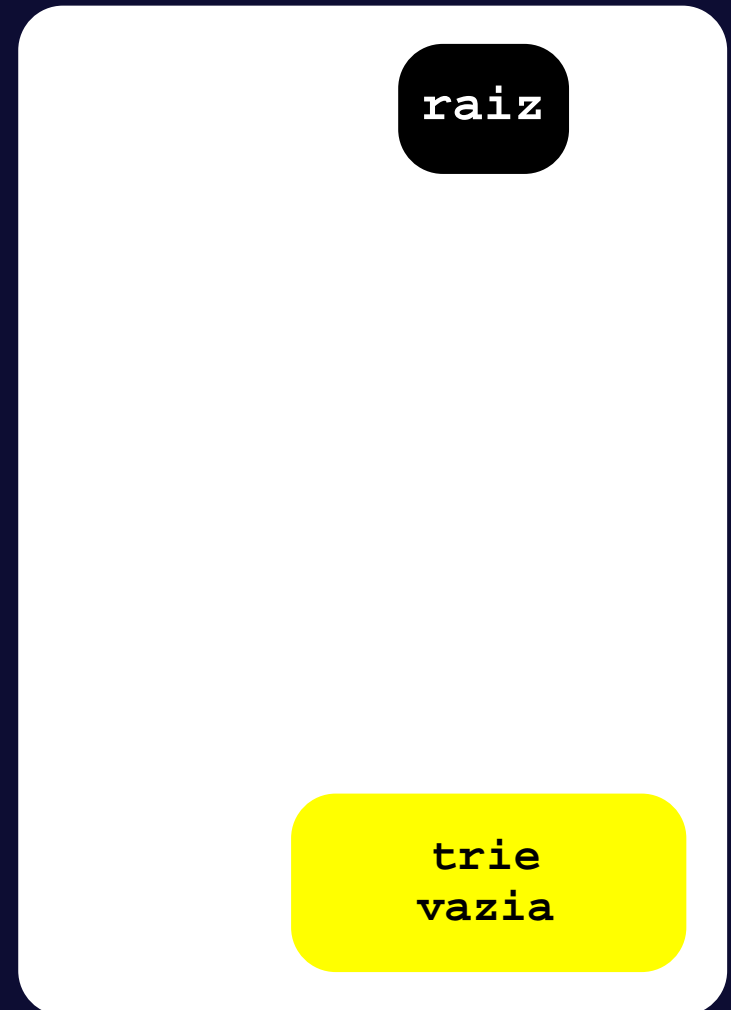
- Para conseguir um sistema de tabulação eficiente, a tabela necessita de ser representada por uma estrutura de dados que seja **compacta** e que permite **rápida procura e inserção** de termos.

YapTab: Representação da Tabela

- Para conseguir um sistema de tabulação eficiente, a tabela necessita de ser representada por uma estrutura de dados que seja **compacta** e que permite **rápida procura e inserção** de termos.
- **Tries** é uma estrutura de dados do tipo árvore em que prefixos comuns são representados apenas uma vez.
 - ◆ Cada caminho através dos nós da trie corresponde a um único termo.
 - ◆ Termos com prefixos comuns ramificam-se a partir do primeiro símbolo distinto.
 - ◆ O ponto de entrada é o nó raiz, nós interiores representam símbolos dos termos, e nós folha representam termos completos.

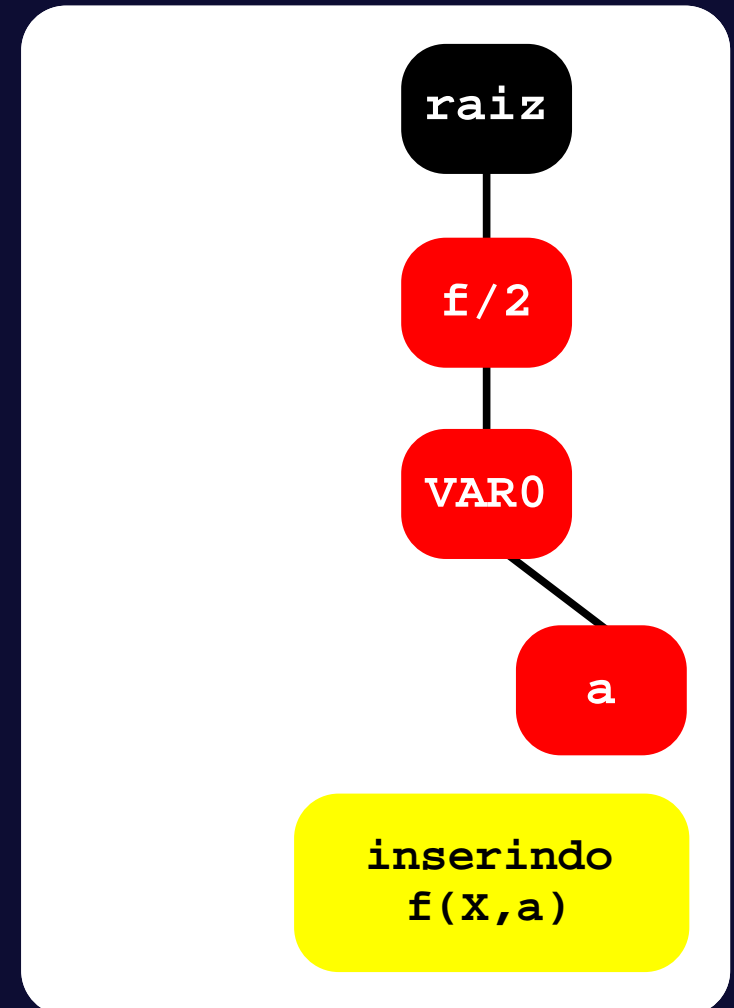
YapTab: Representação da Tabela

- Para conseguir um sistema de tabulação eficiente, a tabela necessita de ser representada por uma estrutura de dados que seja **compacta** e que permite **rápida procura e inserção** de termos.
- **Tries** é uma estrutura de dados do tipo árvore em que prefixos comuns são representados apenas uma vez.
 - ◆ Cada caminho através dos nós da trie corresponde a um único termo.
 - ◆ Termos com prefixos comuns ramificam-se a partir do primeiro símbolo distinto.
 - ◆ O ponto de entrada é o nó raiz, nós interiores representam símbolos dos termos, e nós folha representam termos completos.



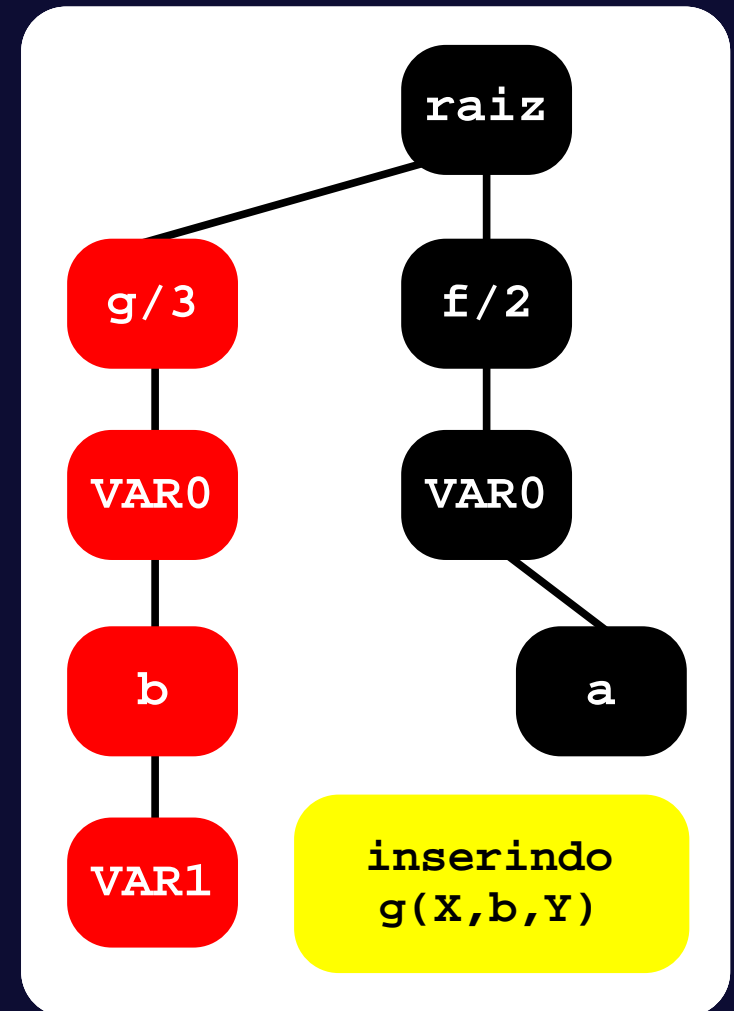
YapTab: Representação da Tabela

- Para conseguir um sistema de tabulação eficiente, a tabela necessita de ser representada por uma estrutura de dados que seja **compacta** e que permite **rápida procura e inserção** de termos.
- **Tries** é uma estrutura de dados do tipo árvore em que prefixos comuns são representados apenas uma vez.
 - ◆ Cada caminho através dos nós da trie corresponde a um único termo.
 - ◆ Termos com prefixos comuns ramificam-se a partir do primeiro símbolo distinto.
 - ◆ O ponto de entrada é o nó raiz, nós interiores representam símbolos dos termos, e nós folha representam termos completos.



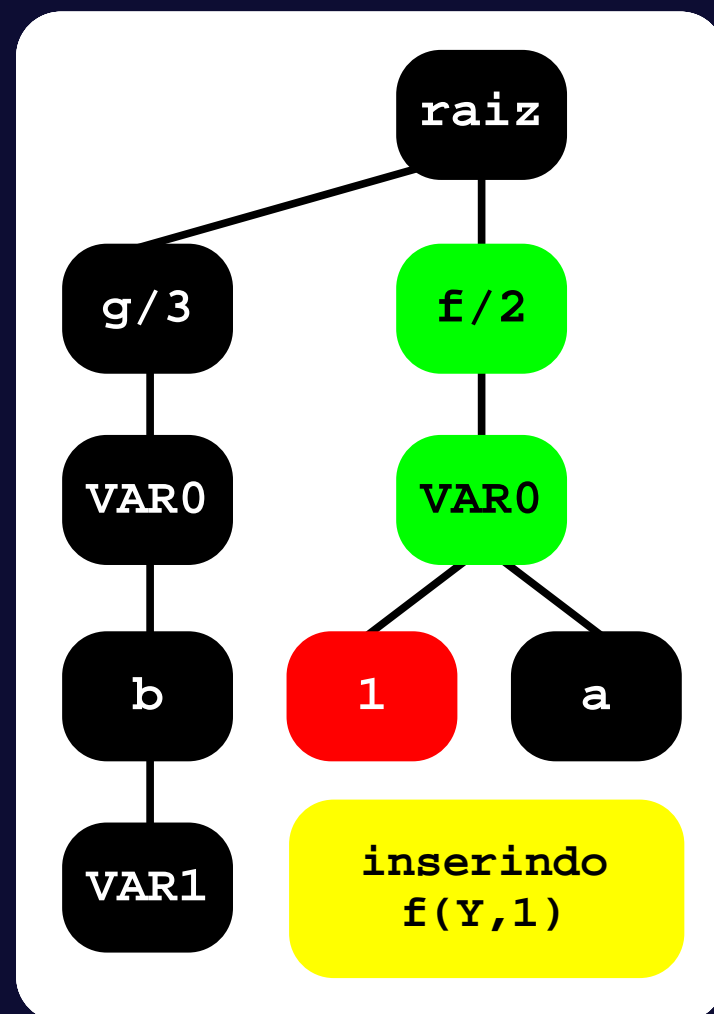
YapTab: Representação da Tabela

- Para conseguir um sistema de tabulação eficiente, a tabela necessita de ser representada por uma estrutura de dados que seja **compacta** e que permite **rápida procura e inserção** de termos.
- **Tries** é uma estrutura de dados do tipo árvore em que prefixos comuns são representados apenas uma vez.
 - ◆ Cada caminho através dos nós da trie corresponde a um único termo.
 - ◆ Termos com prefixos comuns ramificam-se a partir do primeiro símbolo distinto.
 - ◆ O ponto de entrada é o nó raiz, nós interiores representam símbolos dos termos, e nós folha representam termos completos.



YapTab: Representação da Tabela

- Para conseguir um sistema de tabulação eficiente, a tabela necessita de ser representada por uma estrutura de dados que seja **compacta** e que permite **rápida procura e inserção** de termos.
- **Tries** é uma estrutura de dados do tipo árvore em que prefixos comuns são representados apenas uma vez.
 - ◆ Cada caminho através dos nós da trie corresponde a um único termo.
 - ◆ Termos com prefixos comuns ramificam-se a partir do primeiro símbolo distinto.
 - ◆ O ponto de entrada é o nó raiz, nós interiores representam símbolos dos termos, e nós folha representam termos completos.



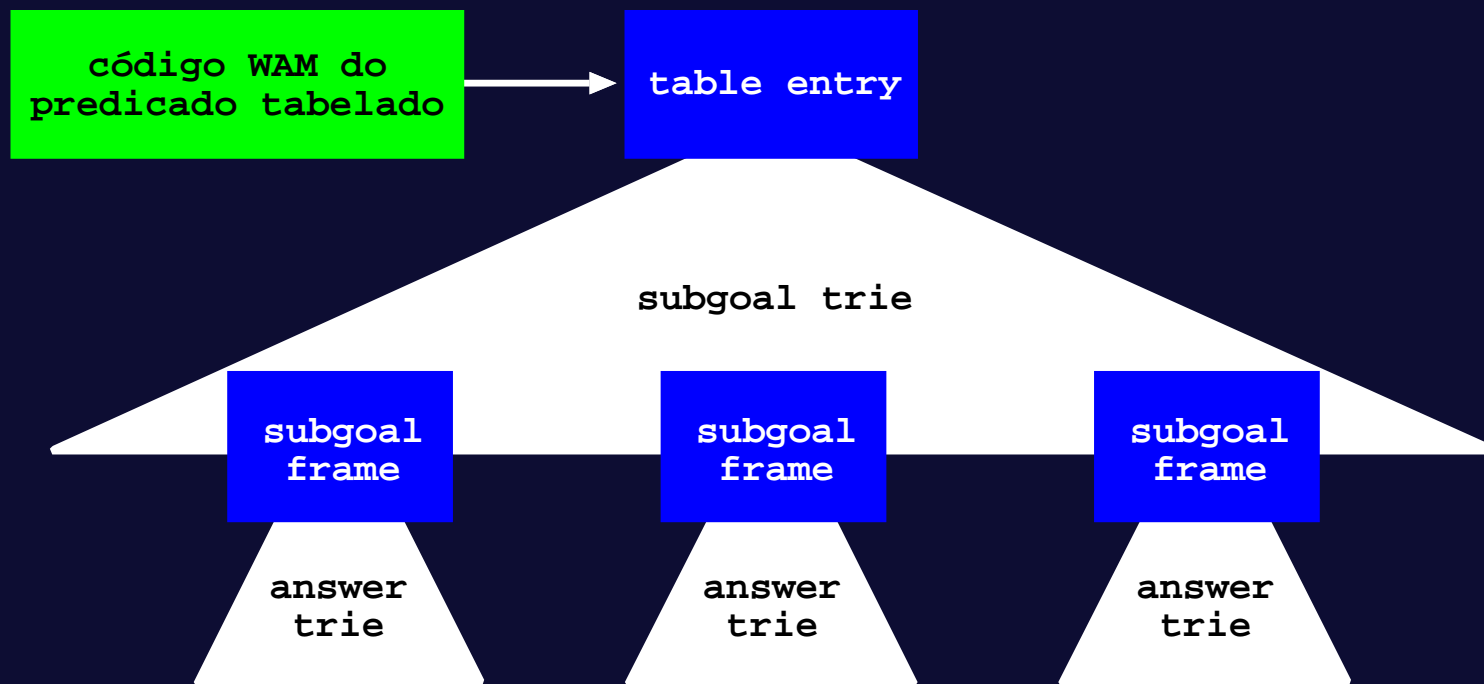
YapTab: Organização da Tabela

➤ Subgoal Trie

- ◆ Guarda os objectivos tabelados. Cada **table entry** representa o ponto de entrada dos objectivos de um predicado.

➤ Answer Trie

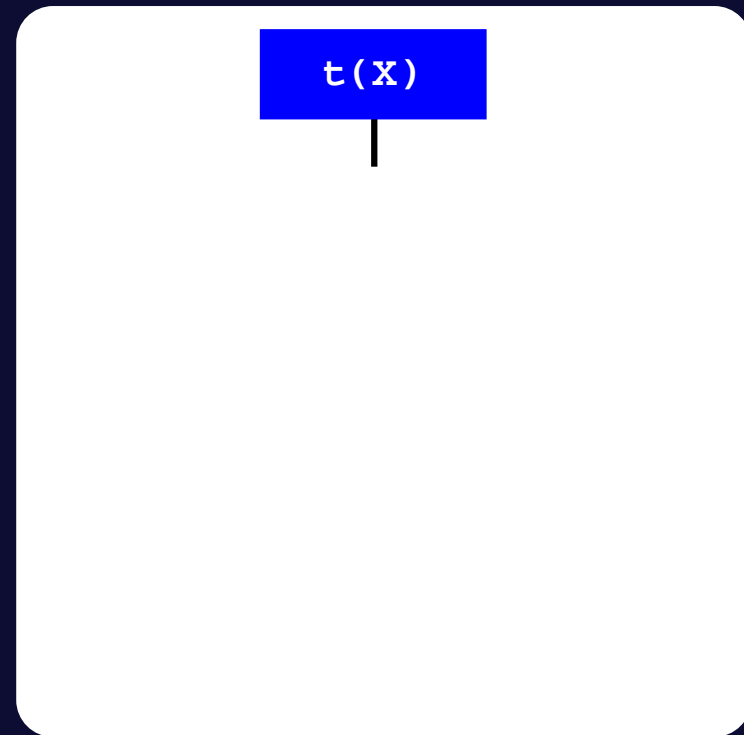
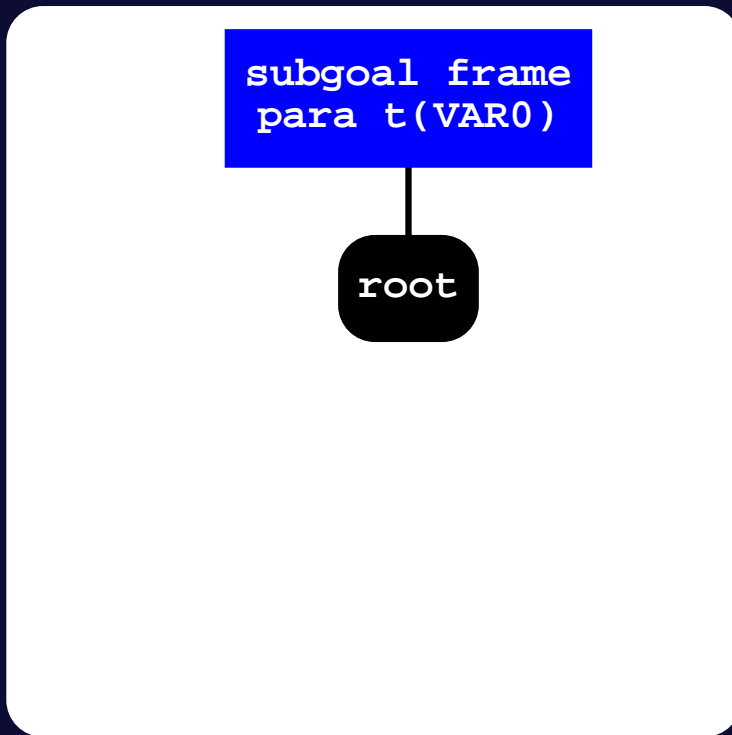
- ◆ Guarda as soluções dos objectivos tabelados. Cada **subgoal frame** representa o ponto de entrada das soluções de um objectivo.



YapTab: Organização da Tabela

➤ Answer Trie

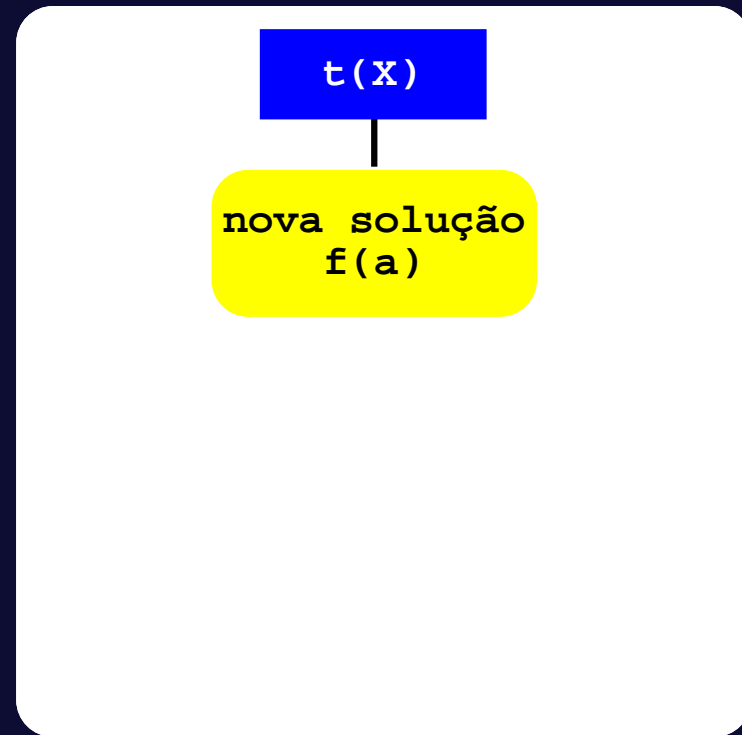
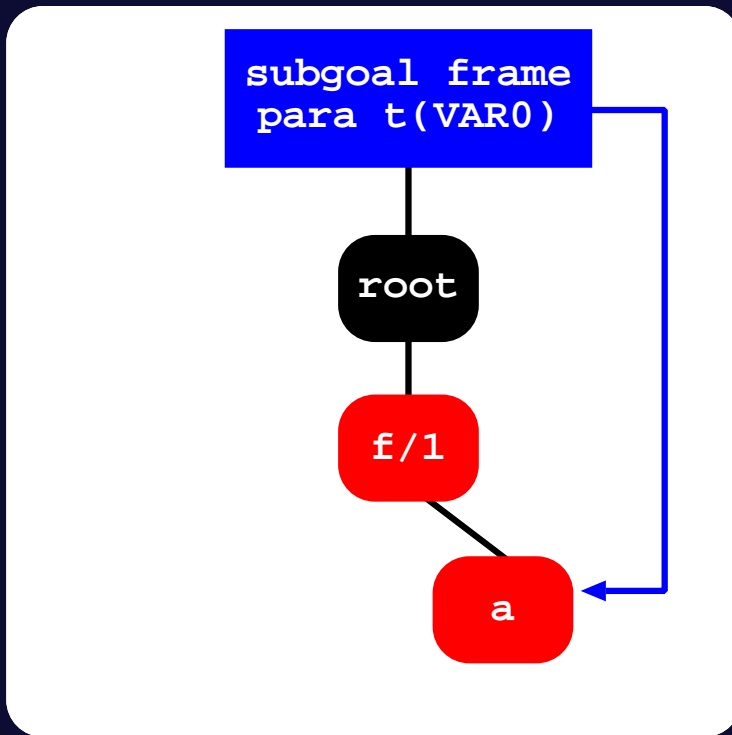
- ◆ As soluções são mantidas numa lista pela ordem que foram encontrados.
- ◆ Os nós consumidores guardam a referência da última solução consumida, e consomem mais soluções seguindo a lista.



YapTab: Organização da Tabela

➤ Answer Trie

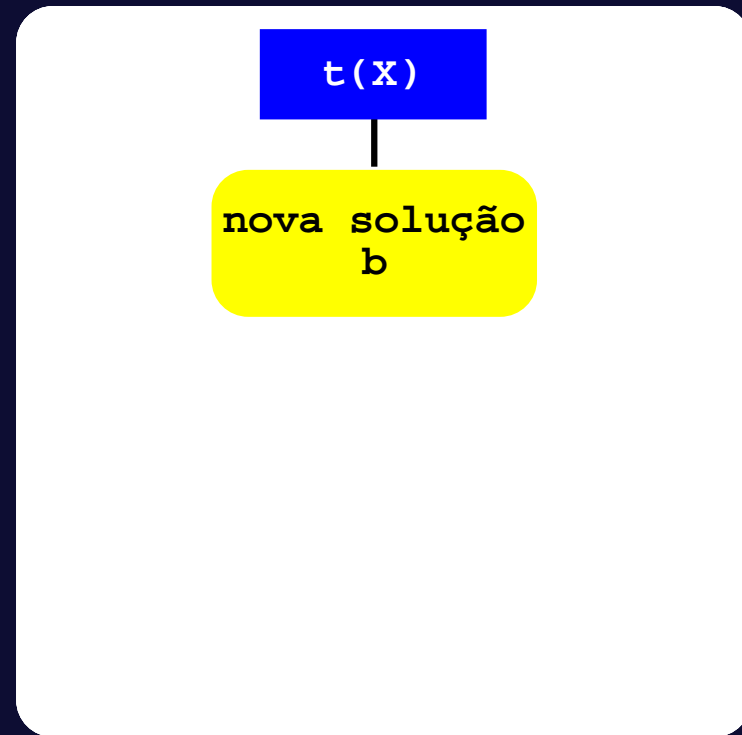
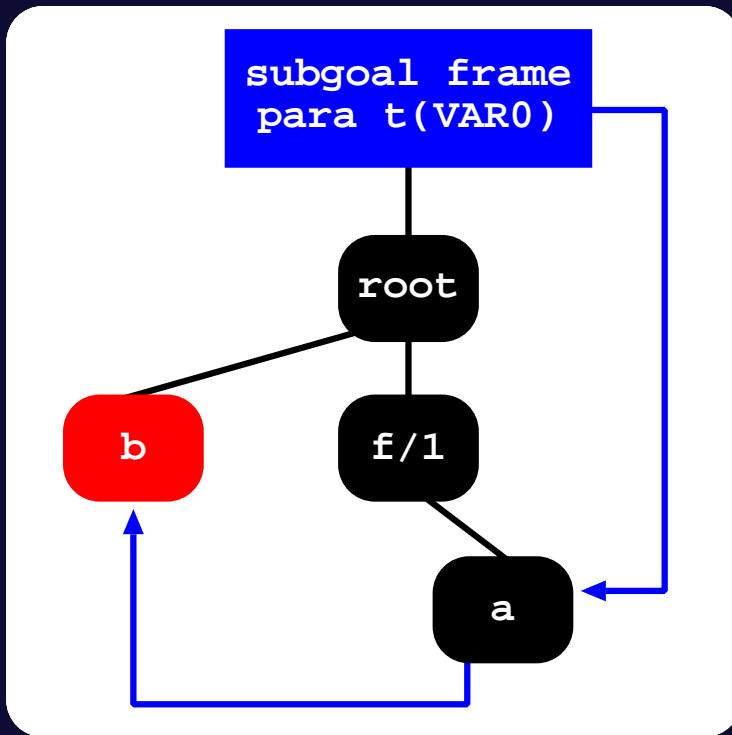
- ◆ As soluções são mantidas numa lista pela ordem que foram encontrados.
- ◆ Os nós consumidores guardam a referência da última solução consumida, e consomem mais soluções seguindo a lista.



YapTab: Organização da Tabela

➤ Answer Trie

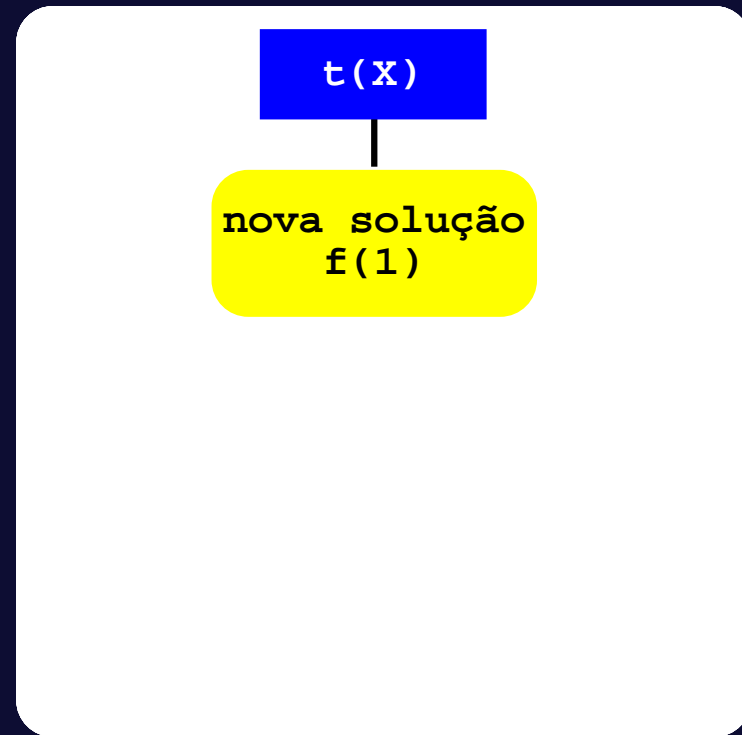
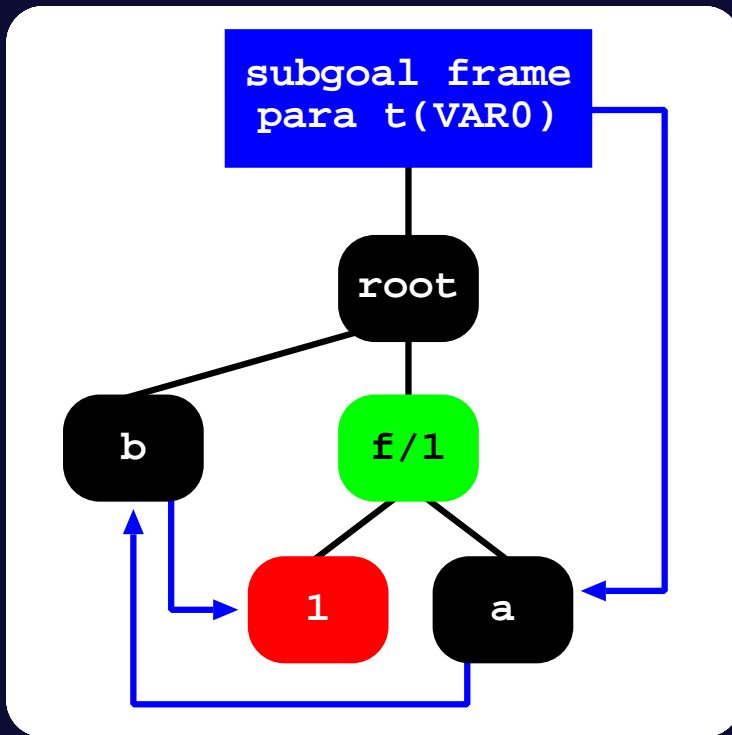
- ◆ As soluções são mantidas numa lista pela ordem que foram encontrados.
- ◆ Os nós consumidores guardam a referência da última solução consumida, e consomem mais soluções seguindo a lista.



YapTab: Organização da Tabela

➤ Answer Trie

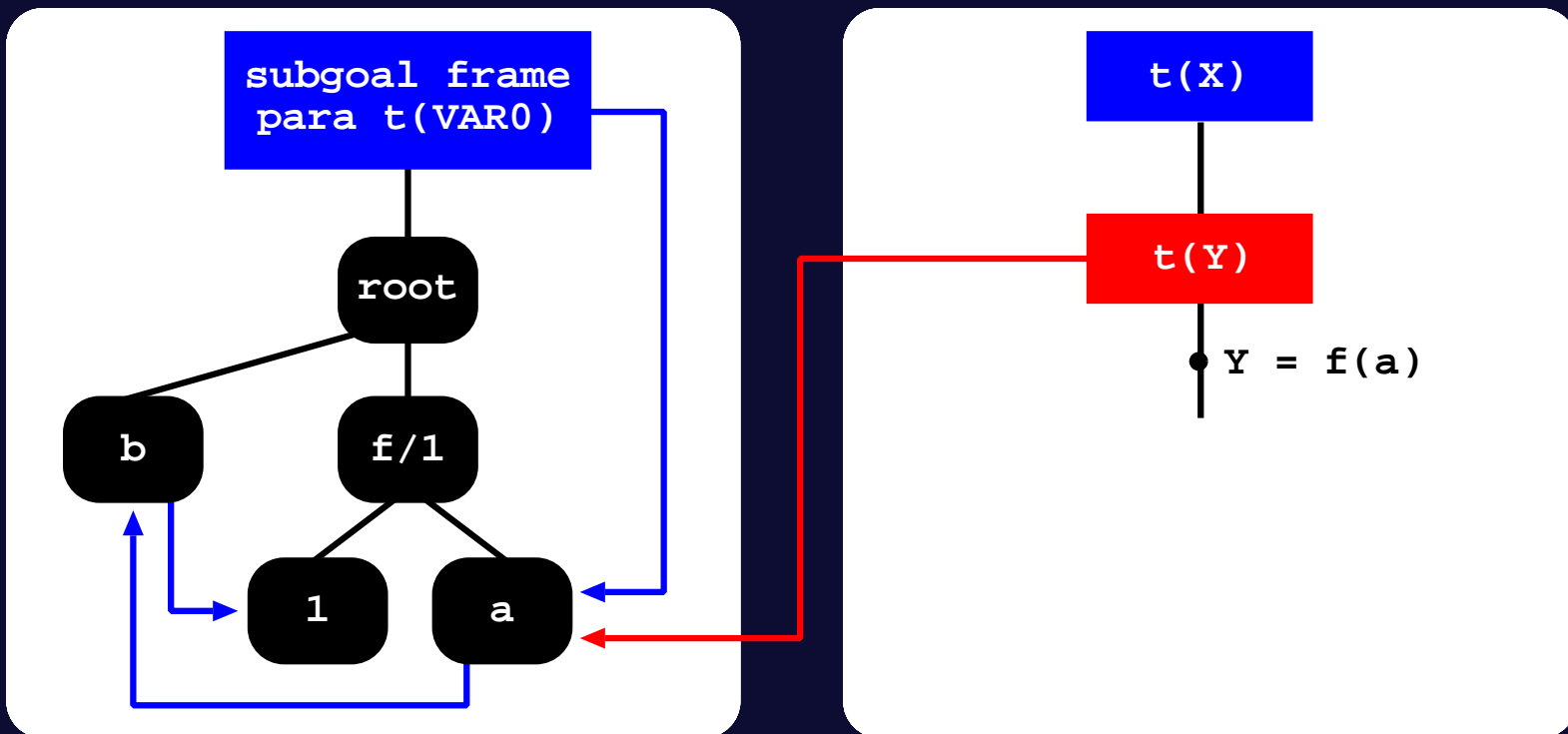
- ◆ As soluções são mantidas numa lista pela ordem que foram encontrados.
- ◆ Os nós consumidores guardam a referência da última solução consumida, e consomem mais soluções seguindo a lista.



YapTab: Organização da Tabela

➤ Answer Trie

- ◆ As soluções são mantidas numa lista pela ordem que foram encontrados.
- ◆ Os nós consumidores guardam a referência da última solução consumida, e consomem mais soluções seguindo a lista.

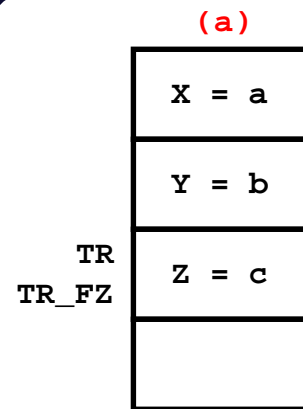
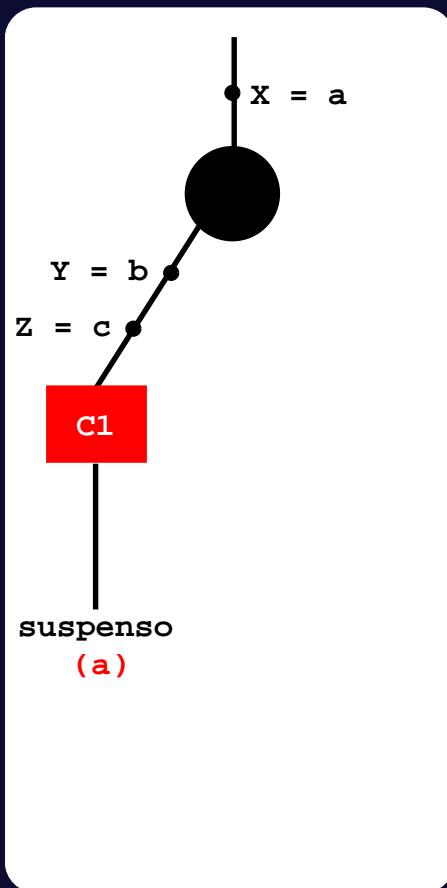


YapTab: Suspensão/Recuperação

- **Freeze Registers:** permitem congelar as pilhas de execução.
- **Forward Trail:** permite restaurar as atribuições condicionais.

YapTab: Suspensão/Recuperação

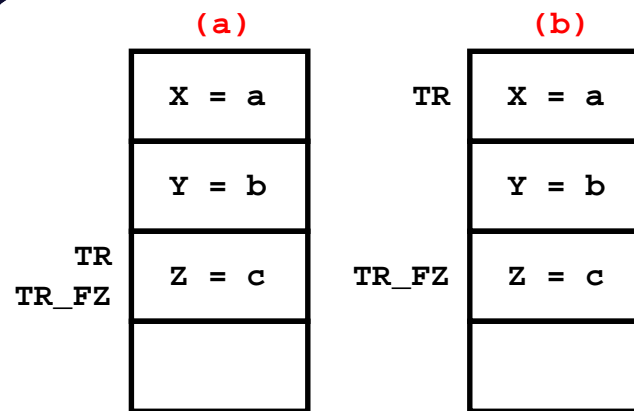
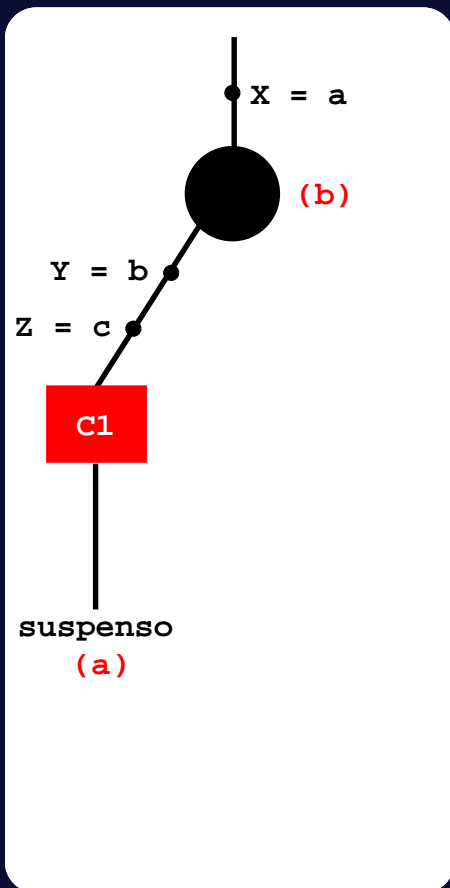
- **Freeze Registers:** permitem congelar as pilhas de execução.
- **Forward Trail:** permite restaurar as atribuições condicionais.



TR: registo da trilha
TR_FZ: registo da trilha congelada

YapTab: Suspensão/Recuperação

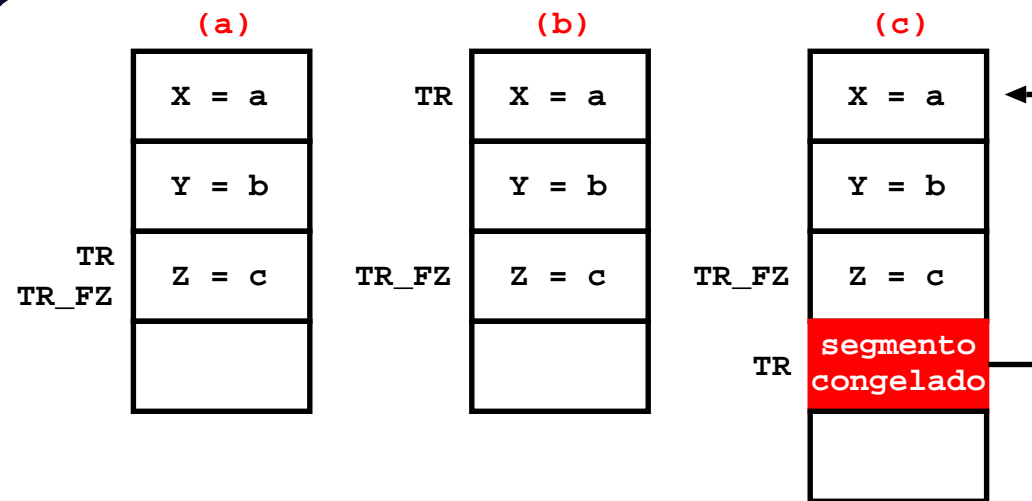
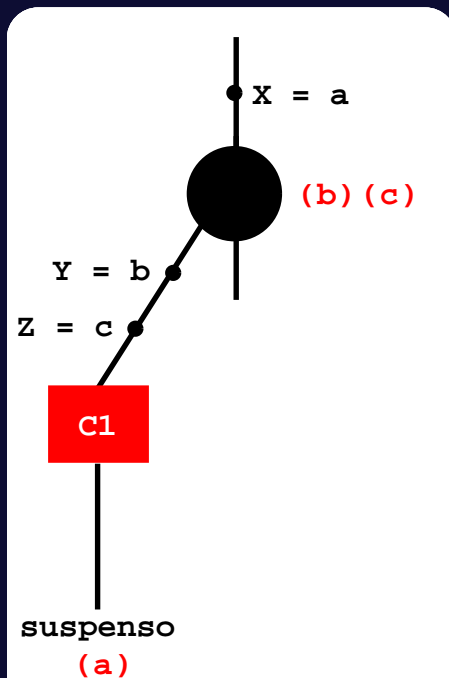
- **Freeze Registers:** permitem congelar as pilhas de execução.
- **Forward Trail:** permite restaurar as atribuições condicionais.



TR: registo da trilha
TR_FZ: registo da trilha congelada

YapTab: Suspensão/Recuperação

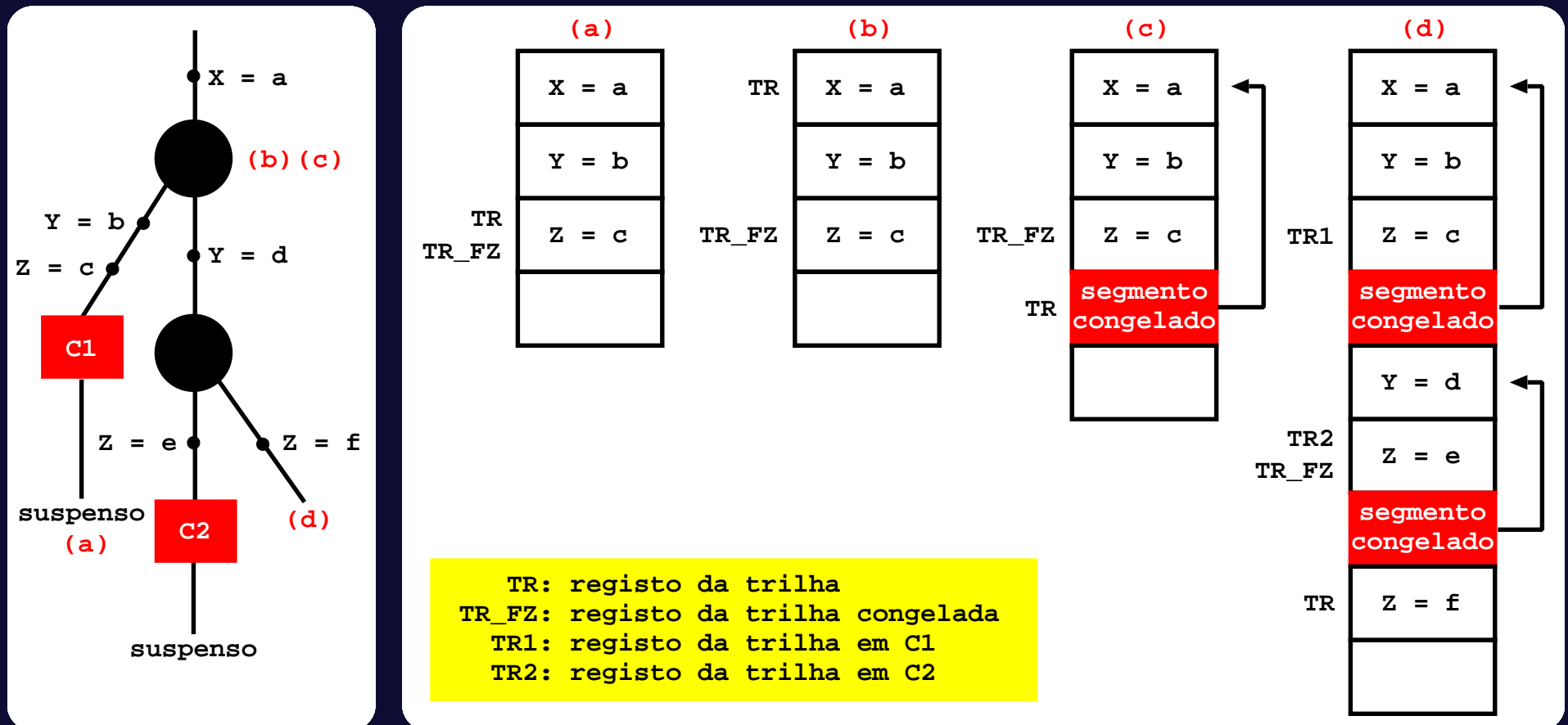
- **Freeze Registers:** permitem congelar as pilhas de execução.
- **Forward Trail:** permite restaurar as atribuições condicionais.



TR: registo da trilha
TR_FZ: registo da trilha congelada

YapTab: Suspensão/Recuperação

- **Freeze Registers:** permitem congelar as pilhas de execução.
- **Forward Trail:** permite restaurar as atribuições condicionais.



YapTab: Completion

- A operação de completion é executada sempre que a computação atinge um **nó gerador** e este já explorou todas as suas alternativas.

O nó gerador é líder?

- **Não** → Falha (backtracking).
- **Sim** → Existe algum nó consumidor CN mais novo com soluções por consumir?
 - ◆ **Sim** → Retoma a computação em CN.
 - ◆ **Não** → Completa o SCC corrente.

YapTab: Answer Resolution

- A operação de answer resolution é executada sempre que a computação atinge um **nó consumidor**.

O nó consumidor tem soluções por consumir?

- **Sim** → Carrega a próxima solução e prossegue a execução.
- **Não** → Retoma a computação no mais novo dos seguintes nós:
 - ◆ Nós consumidores mais antigos com soluções por consumir.
 - ◆ Líder onde foi executada a última operação de completion sem sucesso.

YapTab: Calcular o Líder

- **Ideia chave:** para cada novo consumidor C , calcular antecipadamente o líder do SCC que inclui C .
 - ◆ Calcular o gerador G associado ao consumidor C .
 - ◆ Para todos os consumidores entre C e G , calcular o líder A mais antigo.
 - ◆ O mais antigo entre G e A é o líder do SCC que inclui C .

YapTab: Calcular o Líder

- **Ideia chave:** para cada novo consumidor C , calcular antecipadamente o líder do SCC que inclui C .
 - ◆ Calcular o gerador G associado ao consumidor C .
 - ◆ Para todos os consumidores entre C e G , calcular o líder A mais antigo.
 - ◆ O mais antigo entre G e A é o líder do SCC que inclui C .

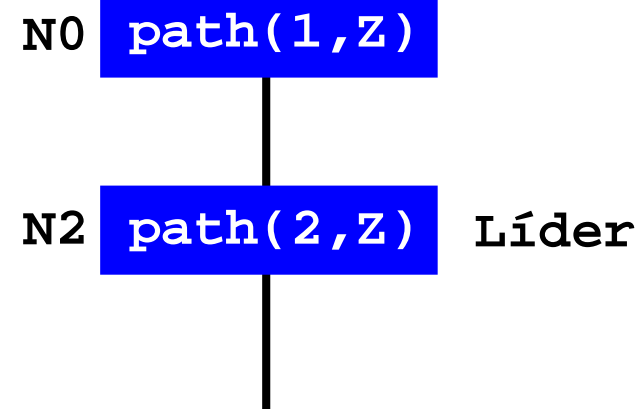
- Em qualquer momento da computação, o líder é sempre o mais novo de entre:
 - ◆ O líder calculado pelo consumidor mais novo.
 - ◆ O gerador mais novo.

N0 **path(1,Z)** Líder



YapTab: Calcular o Líder

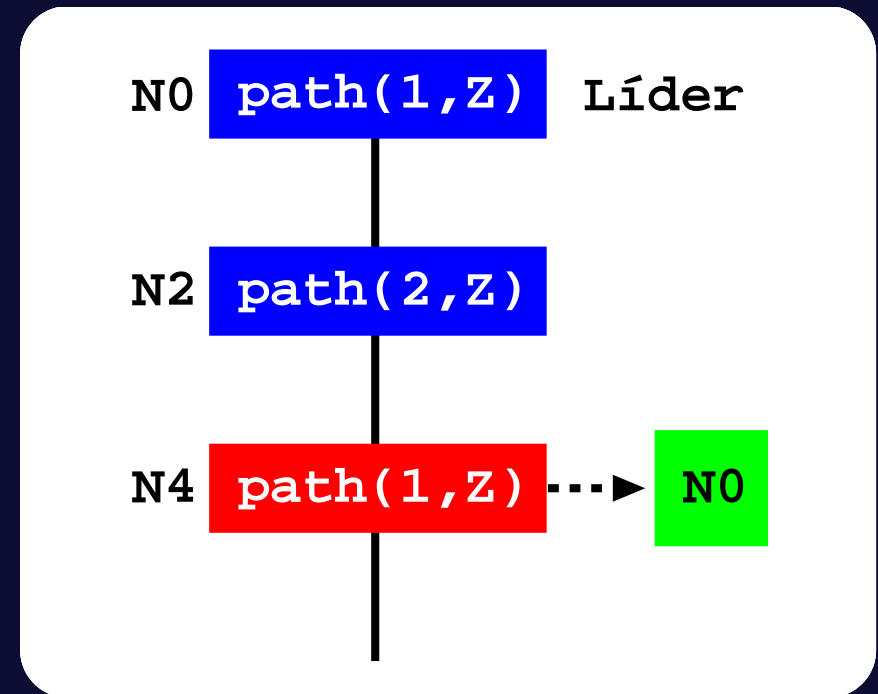
- **Ideia chave:** para cada novo consumidor C , calcular antecipadamente o líder do SCC que inclui C .
 - ◆ Calcular o gerador G associado ao consumidor C .
 - ◆ Para todos os consumidores entre C e G , calcular o líder A mais antigo.
 - ◆ O mais antigo entre G e A é o líder do SCC que inclui C .
- Em qualquer momento da computação, o líder é sempre o mais novo de entre:
 - ◆ O líder calculado pelo consumidor mais novo.
 - ◆ O gerador mais novo.



YapTab: Calcular o Líder

- **Ideia chave:** para cada novo consumidor C, calcular antecipadamente o líder do SCC que inclui C.
 - ◆ Calcular o gerador G associado ao consumidor C.
 - ◆ Para todos os consumidores entre C e G, calcular o líder A mais antigo.
 - ◆ O mais antigo entre G e A é o líder do SCC que inclui C.

- Em qualquer momento da computação, o líder é sempre o mais novo de entre:
 - ◆ O líder calculado pelo consumidor mais novo.
 - ◆ O gerador mais novo.



YapTab: Tabulação Incompleta

- Quando a utilização de um mecanismo de corte invalida a computação de um objectivo tabelado diz-se que a tabela desse objectivo fica **incompleta**.

YapTab: Tabulação Incompleta

- Quando a utilização de um mecanismo de corte invalida a computação de um objectivo tabelado diz-se que a tabela desse objectivo fica **incompleta**.
- ◆ Algumas aplicações para serem eficientes no processo de avaliação de cláusulas usam frequentemente o predicado de corte **once/1** para eliminar a procura por diferentes soluções. A sua definição é:

once(Goal) :- call(Goal), !.

YapTab: Tabulação Incompleta

- Quando a utilização de um mecanismo de corte invalida a computação de um objectivo tabelado diz-se que a tabela desse objectivo fica **incompleta**.
- ◆ Algumas aplicações para serem eficientes no processo de avaliação de cláusulas usam frequentemente o predicado de corte **once/1** para eliminar a procura por diferentes soluções. A sua definição é:

once(Goal) :- call(Goal), !.

- ◆ Por exemplo, se **p1/2** for um predicado tabelado, a execução de

once(p1(a,X), p2(X)).

leva a que **p1(a,X)** seja removido das pilhas de execução antes de completar. Logo, qualquer chamada posterior a **p1(a,X)** não pode ser resolvida consumindo apenas as eventuais soluções existentes na tabela, pois poder-se-ia perder parte da computação.

YapTab: Tabulação Incompleta

- A forma habitual de lidar com este problema é eliminar as tabelas incompletas e reavaliar o objectivo tabelado desde o início quando este é novamente chamado.

YapTab: Tabulação Incompleta

- A forma habitual de lidar com este problema é eliminar as tabelas incompletas e reavaliar o objectivo tabelado desde o início quando este é novamente chamado.
- O sistema YapTab implementa um mecanismo diferente que utiliza as soluções previamente calculadas para evitar re-computação em algumas situações.

YapTab: Tabulação Incompleta

- A forma habitual de lidar com este problema é eliminar as tabelas incompletas e reavaliar o objectivo tabelado desde o início quando este é novamente chamado.
- O sistema YapTab implementa um mecanismo diferente que utiliza as soluções previamente calculadas para evitar re-computação em algumas situações.
 - ◆ Por defeito, tabelas incompletas são mantidas intactas e não eliminadas.
 - ◆ Chamadas posteriores a objectivos incompletos são resolvidas começando por consumir as soluções existentes na tabela.
 - ◆ Só após todas as soluções serem consumidas é que o objectivo é reavaliado desde o início.
 - ◆ Entretanto, se novas soluções forem encontradas, estas são adicionadas à tabela como habitualmente. Se a computação for novamente invalidada por um mecanismo de corte, o processo repete-se até que eventualmente a tabela fique completa.

YapTab: Tabulação Incompleta

- Considere agora que após a execução de

once(p1(a,X), p2(X)).

p1/2 é chamado novamente durante a execução de

once(p1(a,X), p3(X)).

YapTab: Tabulação Incompleta

- Considere agora que após a execução de
`once(p1(a,X), p2(X)).`
`p1/2` é chamado novamente durante a execução de
`once(p1(a,X), p3(X)).`
- Se **`p3(X)`** suceder com alguma das soluções encontradas previamente para **`p1(a,X)`**, então existe um claro benefício do facto de termos mantido a tabela incompleta para **`p1(a,X)`**.

YapTab: Tabulação Incompleta

- Considere agora que após a execução de

`once(p1(a,X), p2(X)).`

`p1/2` é chamado novamente durante a execução de

`once(p1(a,X), p3(X)).`

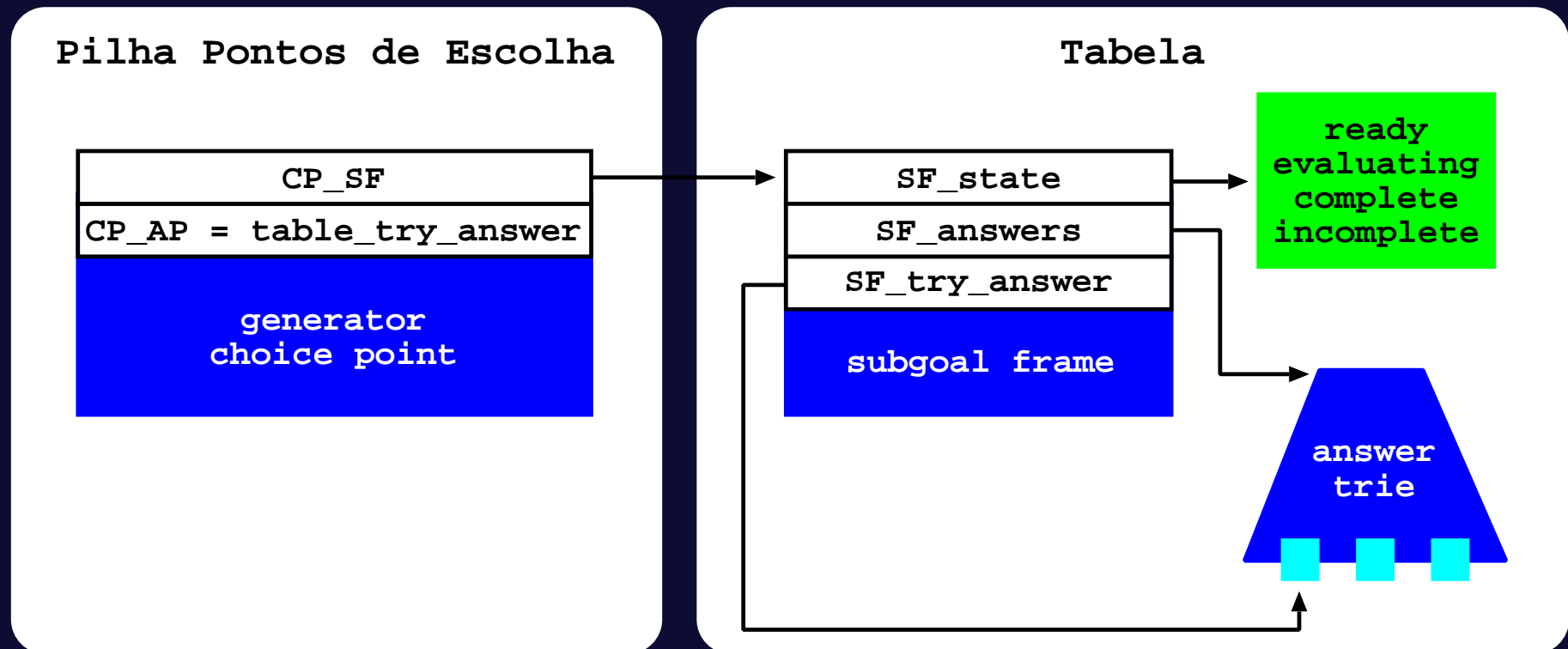
- Se **`p3(X)`** suceder com alguma das soluções encontradas previamente para **`p1(a,X)`**, então existe um claro benefício do facto de termos mantido a tabela incompleta para **`p1(a,X)`**.
- Caso contrário, **`p1(a,X)`** é reavaliado desde o início, o que significa que irá falhar enquanto não for encontrada uma solução diferente das já existentes.

Nestas situações, não é possível beneficiar do facto de termos mantido a tabela incompleta mas, por outro lado, também não existe qualquer custo já que o esforço computacional necessário para avaliar o objectivo, com ou sem tabelas incompletas, é o mesmo.

YapTab: Tabulação Incompleta

➤ Extensões de Suporte

- ◆ Nova pseudo-instrução: **table_try_answer**.
- ◆ Novo campo nas subgoal frames: **SF_try_answer**.
- ◆ Novo estado para as subgoal frames: **incomplete**.



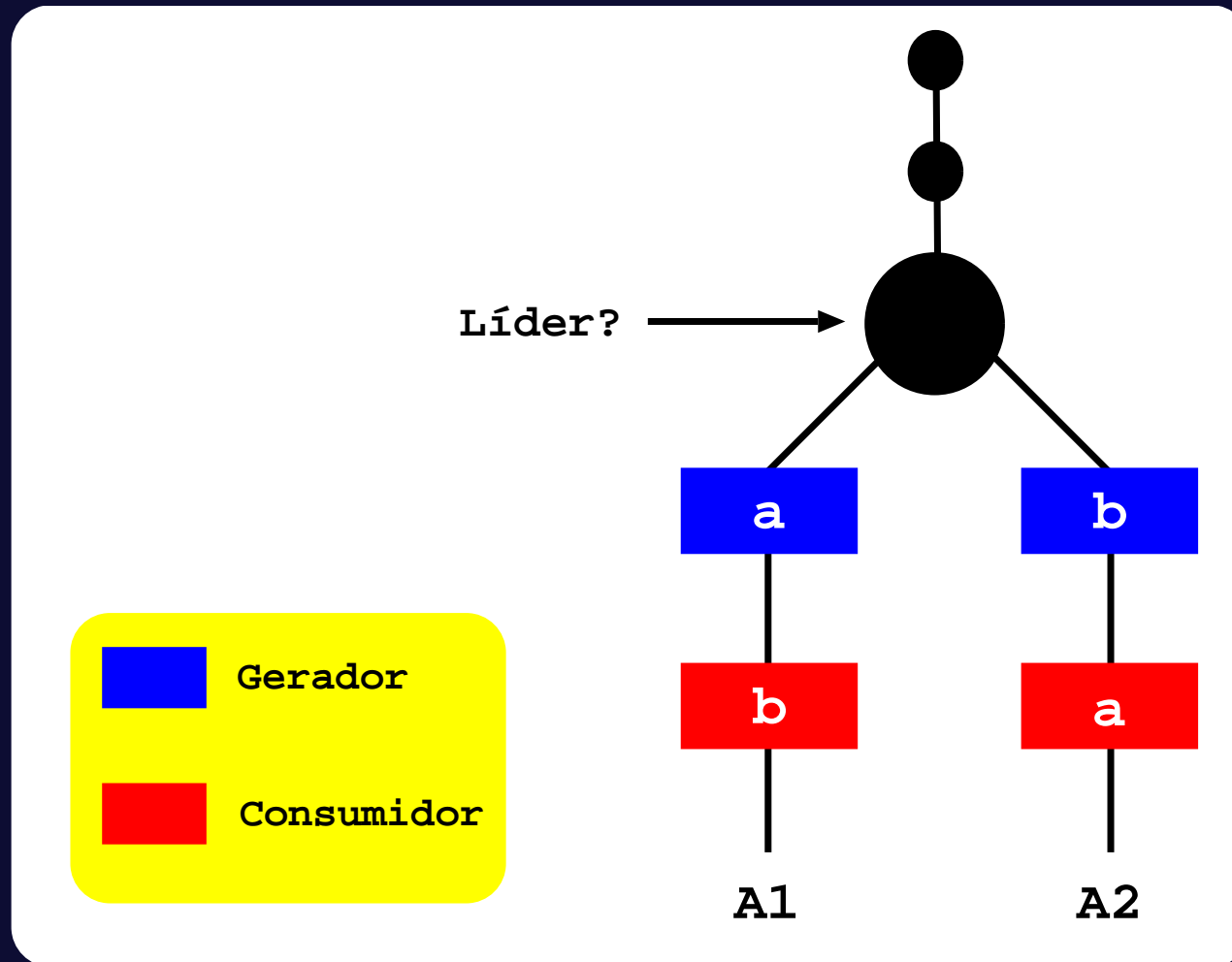
Tabulação e Paralelismo

- Será que o mecanismo de tabulação possui potencial para a execução paralela?
 - ◆ A tabulação baseia-se igualmente na exploração de um espaço de procura para encontrar soluções que satisfaçam os objectivos.
 - ◆ A ordem das soluções tabeladas não é relevante.

Tabulação e Paralelismo

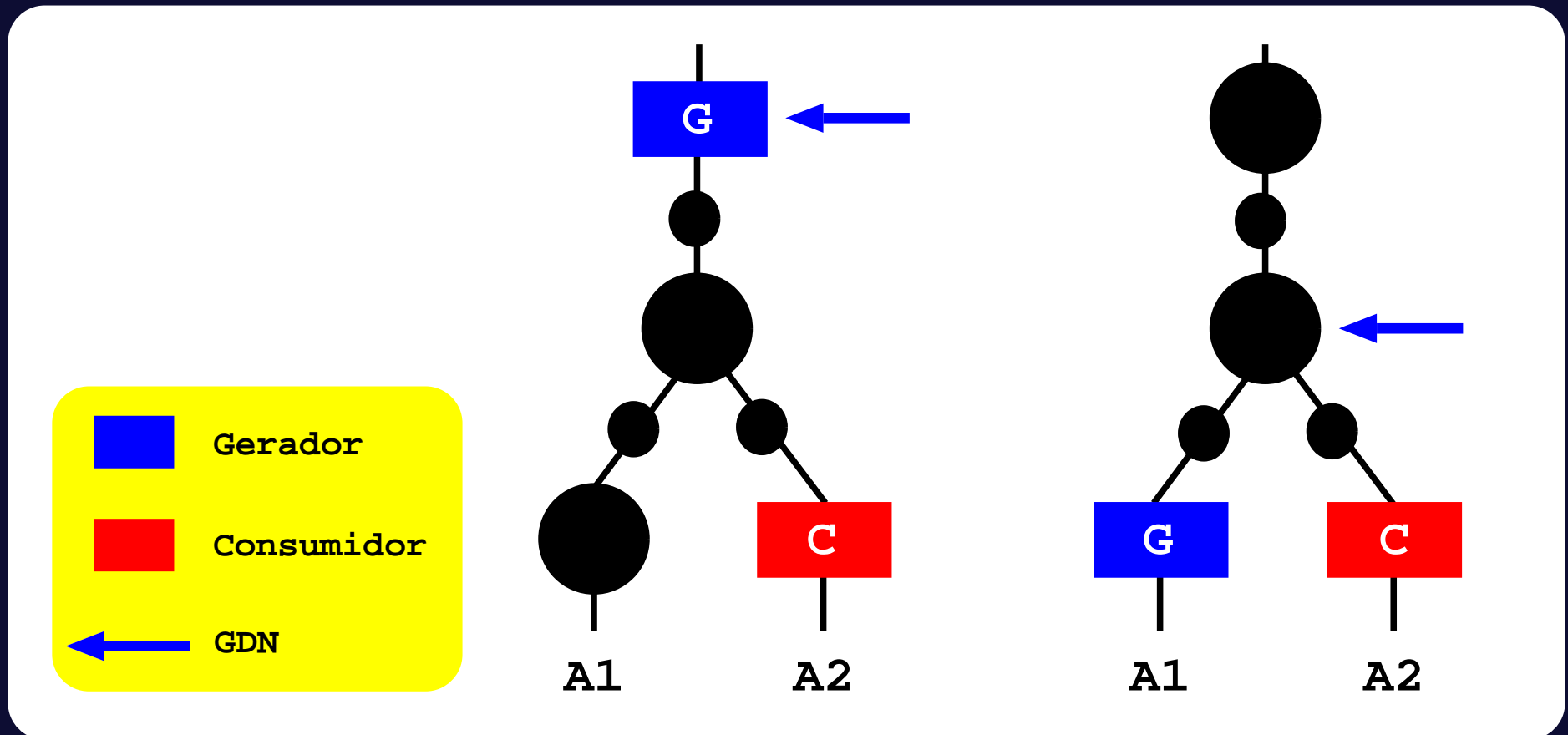
- Será que o mecanismo de tabulação possui potencial para a execução paralela?
 - ◆ A tabulação baseia-se igualmente na exploração de um espaço de procura para encontrar soluções que satisfaçam os objectivos.
 - ◆ A ordem das soluções tabeladas não é relevante.
- Principais desafios a resolver:
 - ◆ O modelo paralelo para tabulação é necessariamente bastante **mais complexo** do que os modelos tradicionais de paralelismo.
 - ◆ A execução paralela introduz **concorrência** no acesso à tabela.

Como Calcular o Líder no Ambiente Paralelo?



Generator Dependency Node (GDN)

- **Definição:** o GDN de um consumidor C, é o nó mais novo que é comum à ramificação corrente de C e à ramificação do gerador G associado a C.

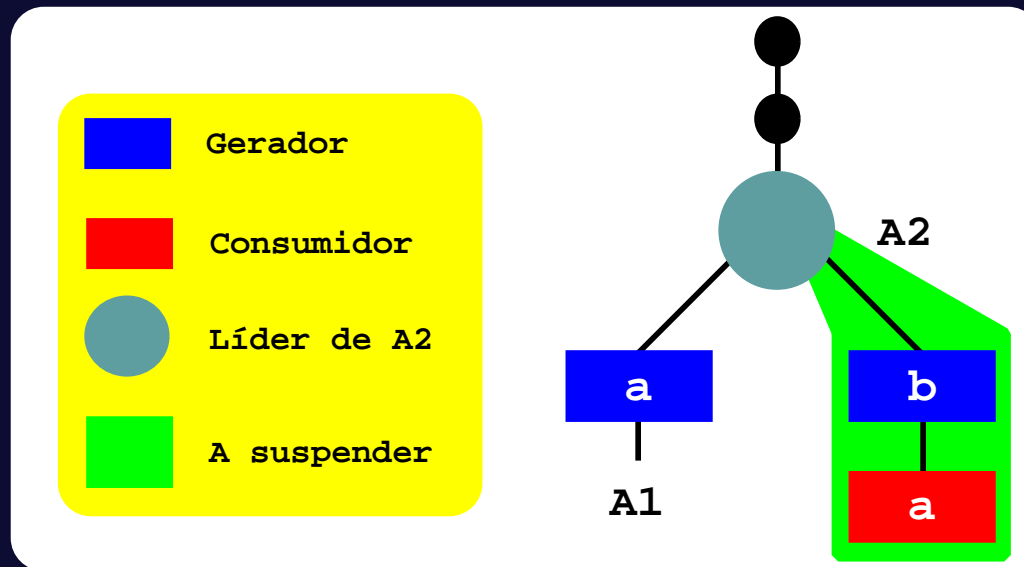


Como Calcular o Líder no Ambiente Paralelo?

- **Ideia chave:** substituir o cálculo do gerador pelo cálculo do GDN.
- Para cada novo consumidor C, calcular antecipadamente o líder do SCC que inclui C.
 - ◆ Calcular o GDN associado ao consumidor C.
 - ◆ Para todos os consumidores entre C e o GDN, calcular o líder A mais antigo.
 - ◆ O mais antigo entre o GDN e A é o líder do SCC que inclui C.
- No ambiente paralelo, o líder pode ser qualquer tipo de nó:
 - ◆ Gerador
 - ◆ Consumidor
 - ◆ Interior

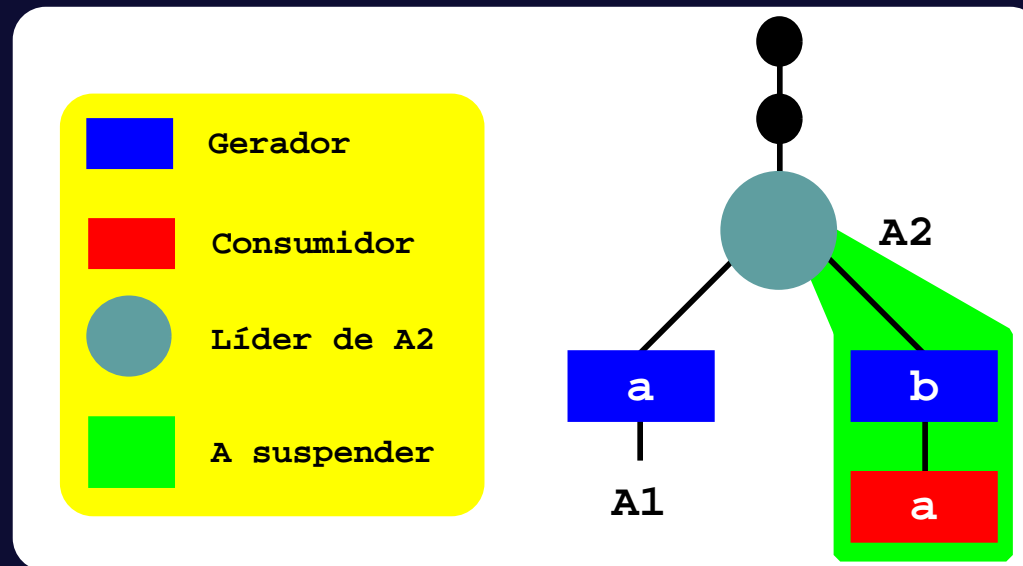
Completion em Nós Líderes Públicos

- **Problema:** um SCC não pode ser completo se existir a possibilidade dos restantes agentes poderem encontrar novas soluções para os nós consumidores do SCC.
- **Solução:** suspender o SCC a partir do nó líder.



Completion em Nós Líderes Públicos

- **Problema:** um SCC não pode ser completo se existir a possibilidade dos restantes agentes poderem encontrar novas soluções para os nós consumidores do SCC.
- **Solução:** suspender o SCC a partir do nó líder.



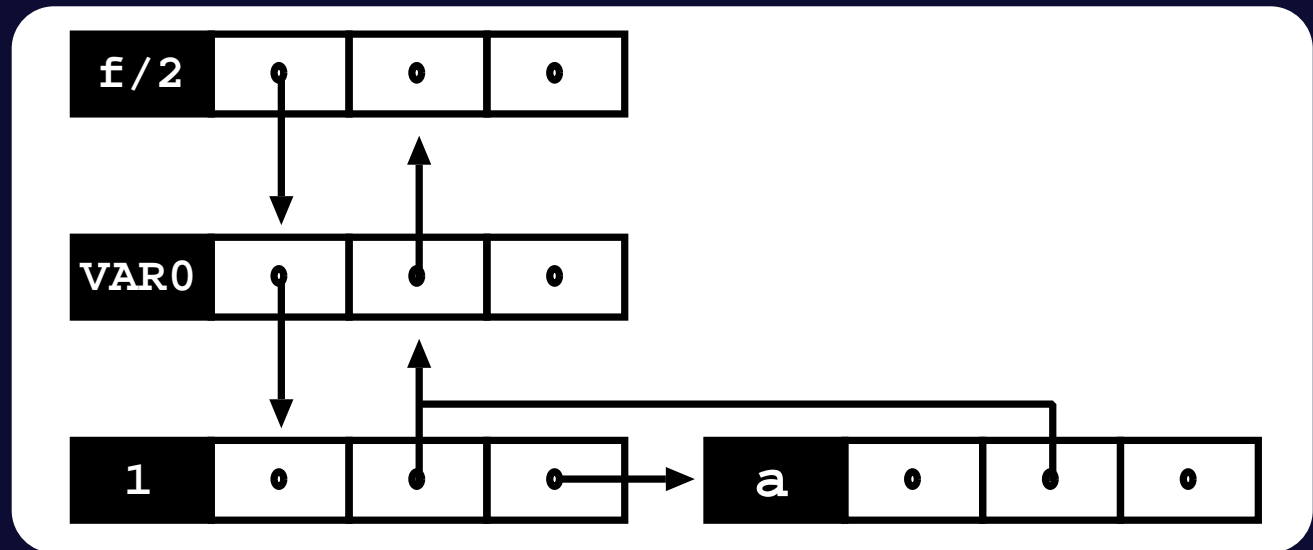
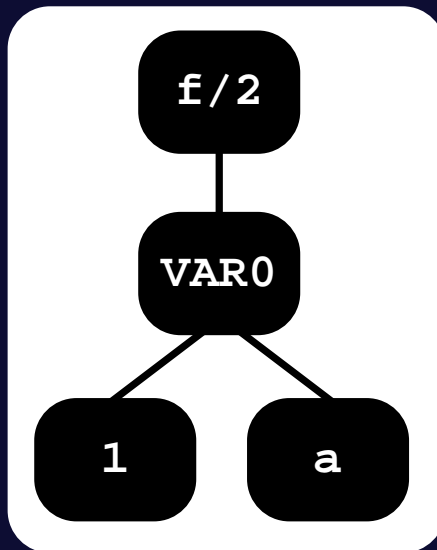
- Condições para efectuar completion num nó líder público:
 - ◆ O nó líder ser único (não pertencer a outro agente ou a um SCC suspenso).
 - ◆ Não existirem soluções por consumir no SCC e em todos os SCCs suspensos a partir de nós do SCC.

Acesso Concorrente à Tabela

- Por forma a garantir concorrência nas operações de escrita sobre a tabela são necessários mecanismos que garantam **exclusão mútua**.
- Existem três esquemas principais de **locking**:
 - ◆ **TLNL**: Table Lock at Node Level
 - ◆ **TLWL**: Table Lock at Write Level
 - ◆ **TLWL-ABC**: Table Lock at Write Level - Allocate Before Check
- A eficiência de cada esquema depende do:
 - ◆ Número de locks necessários para inserir um termo (**lock count**).
 - ◆ Quantidade média de tempo que um lock está activo (**lock duration**).

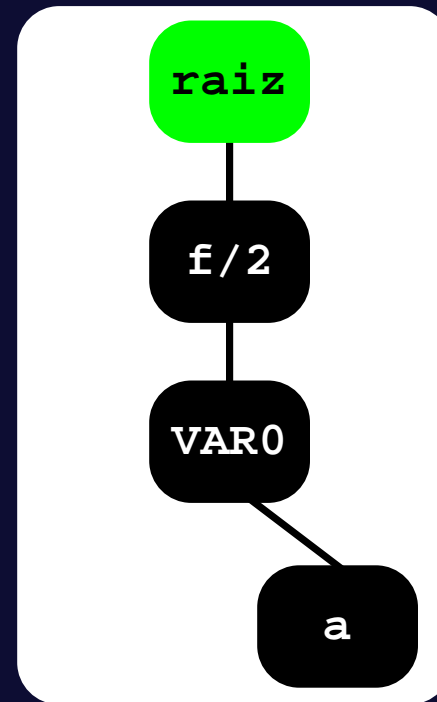
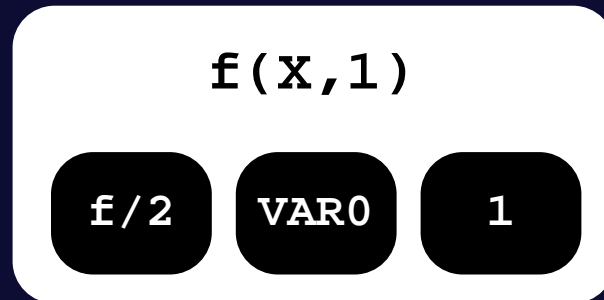
Estrutura dos Nós da Trie

- Cada nó é composto por quatro campos:
- ◆ **TrNode_symbol**: símbolo representado pelo nó.
 - ◆ **TrNode_child**: apontador para o primeiro nó filho.
 - ◆ **TrNode_parent**: apontador para o nó pai.
 - ◆ **TrNode_next**: apontador para o nó irmão seguinte.



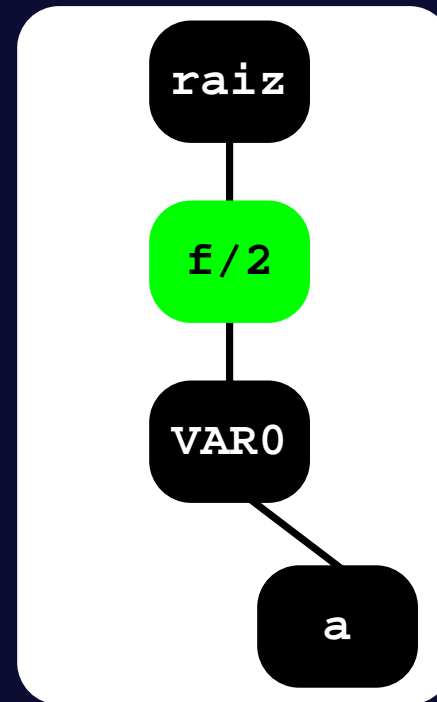
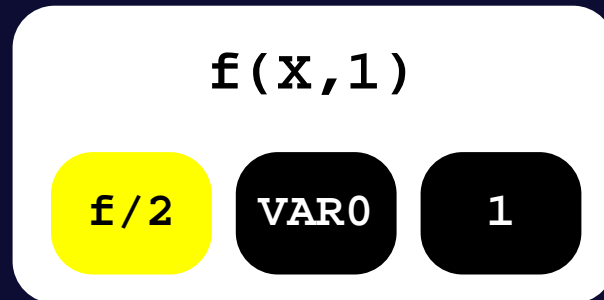
Inserindo um Termo na Tabela

```
...  
parent = raiz  
parent = trie_check_insert(f/2, parent)  
parent = trie_check_insert(VAR0, parent)  
parent = trie_check_insert(1, parent)  
...
```



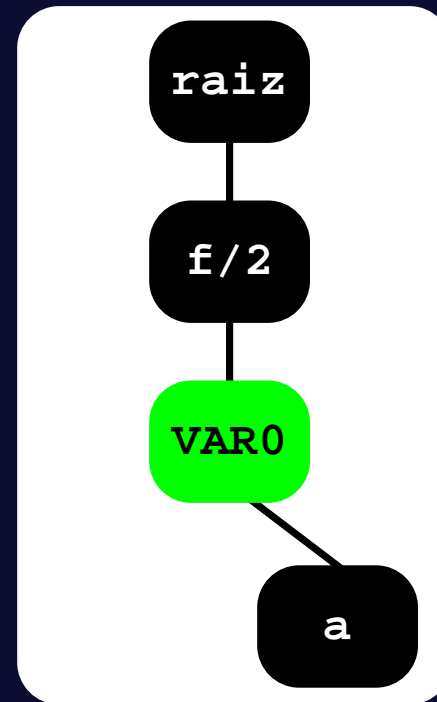
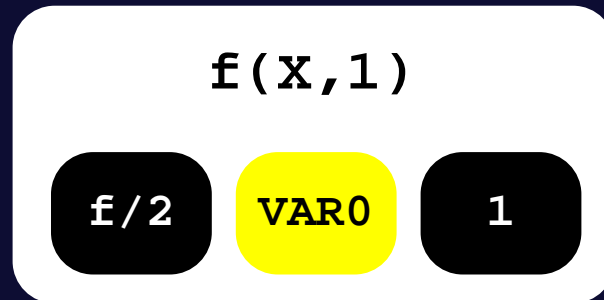
Inserindo um Termo na Tabela

```
...  
parent = raiz  
parent = trie_check_insert(f/2, parent)  
parent = trie_check_insert(VAR0, parent)  
parent = trie_check_insert(1, parent)  
...
```



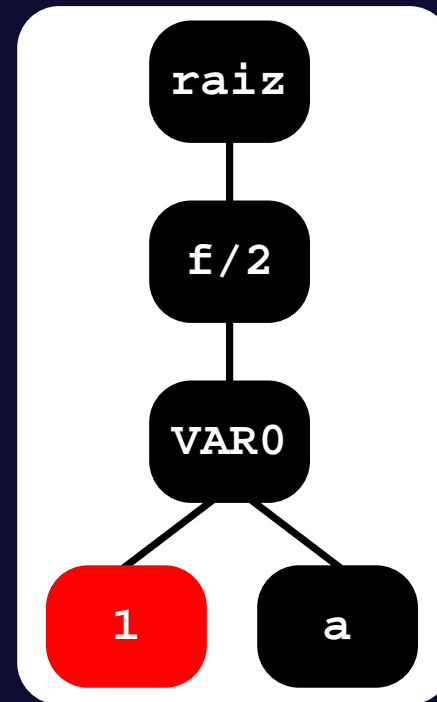
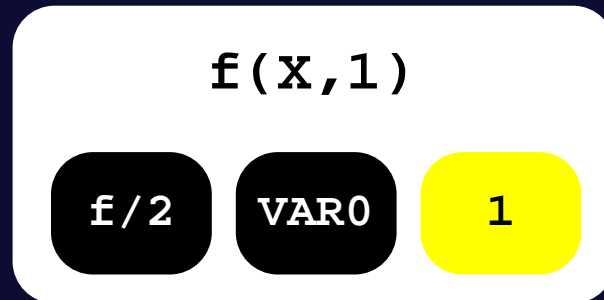
Inserindo um Termo na Tabela

```
...  
parent = raiz  
parent = trie_check_insert(f/2, parent)  
parent = trie_check_insert(VAR0, parent)  
parent = trie_check_insert(1, parent)  
...
```



Inserindo um Termo na Tabela

```
...  
parent = raiz  
parent = trie_check_insert(f/2, parent)  
parent = trie_check_insert(VAR0, parent)  
parent = trie_check_insert(1, parent)  
...
```



Inserindo um Termo na Tabela

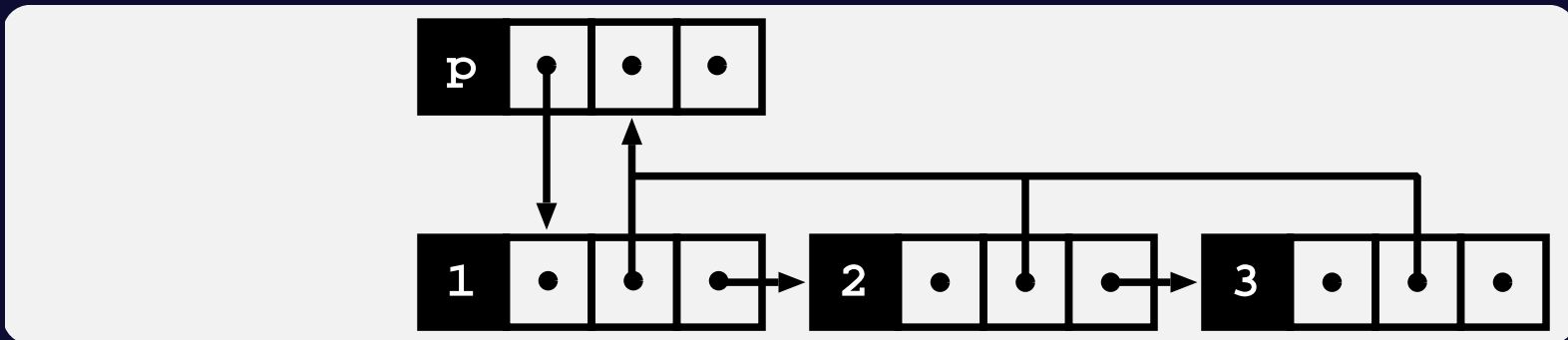
```
...  
parent = raiz  
parent = trie_check_insert(f/2, parent)  
parent = trie_check_insert(VAR0, parent)  
parent = trie_check_insert(1, parent)  
...
```

Inserindo um Termo na Tabela

```

trie_check_insert(symbol s, trie node parent) {
  child = TrNode_child(parent)
  while (child) {
    if (TrNode_symbol(child) == s) return child
    child = TrNode_next(child)
  }
  child = new_trie_node(s, NULL, parent, TrNode_child(parent))
  TrNode_child(parent) = child
  return child
}

```

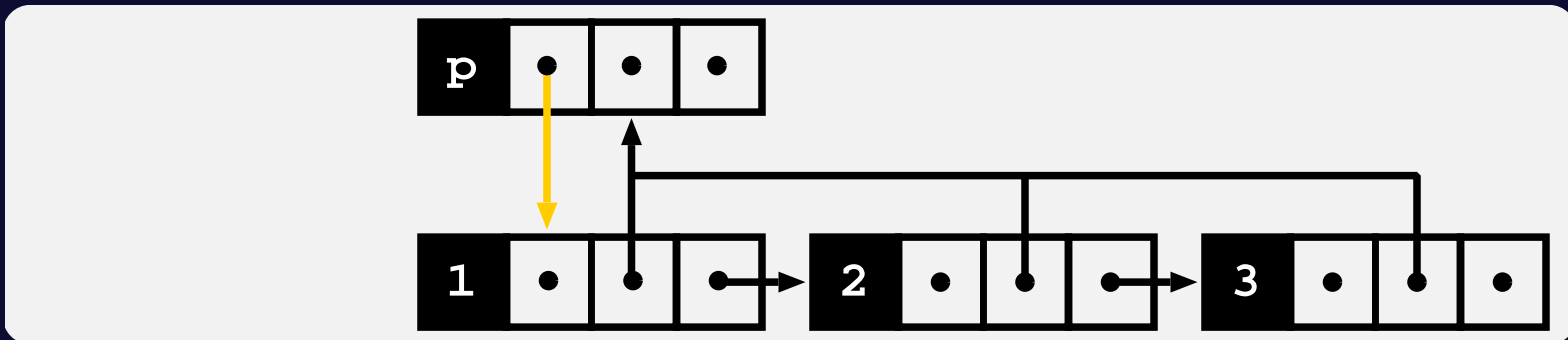


Inserindo um Termo na Tabela

```

trie_check_insert(symbol s, trie node parent) {
  child = TrNode_child(parent)
  while (child) {
    if (TrNode_symbol(child) == s) return child
    child = TrNode_next(child)
  }
  child = new_trie_node(s, NULL, parent, TrNode_child(parent))
  TrNode_child(parent) = child
  return child
}

```

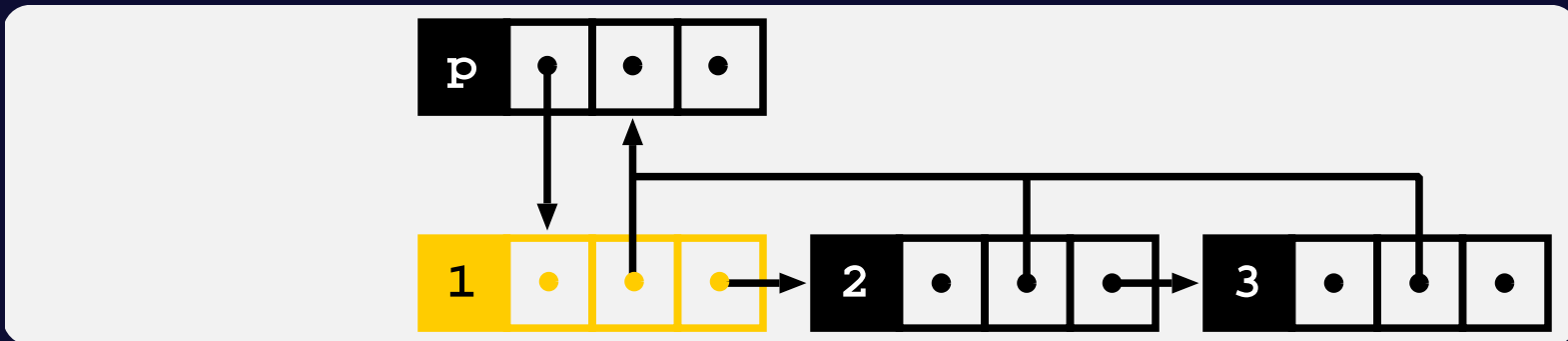


Inserindo um Termo na Tabela

```

trie_check_insert(symbol s, trie node parent) {
  child = TrNode_child(parent)
  while (child) {
    if (TrNode_symbol(child) == s) return child
    child = TrNode_next(child)
  }
  child = new_trie_node(s, NULL, parent, TrNode_child(parent))
  TrNode_child(parent) = child
  return child
}

```

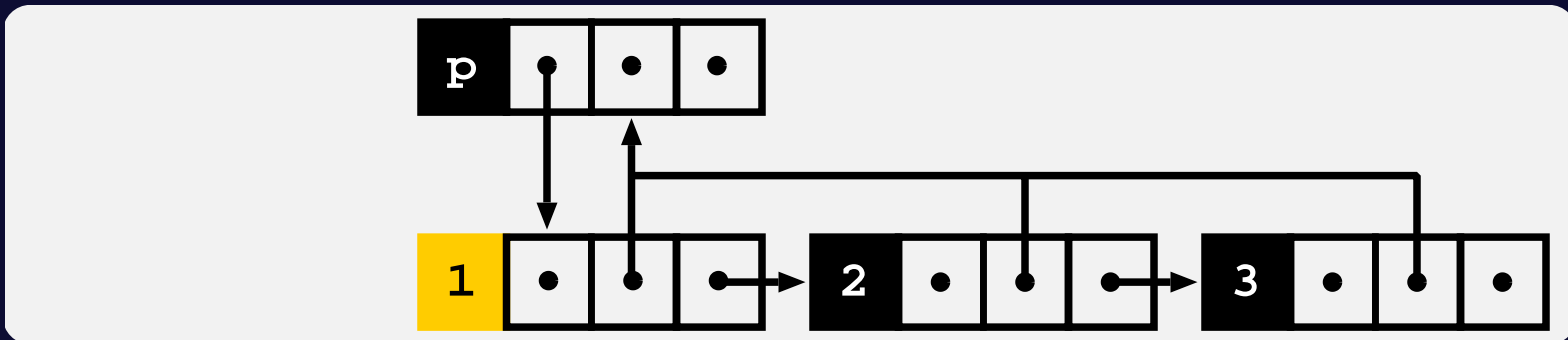


Inserindo um Termo na Tabela

```

trie_check_insert(symbol s, trie node parent) {
  child = TrNode_child(parent)
  while (child) {
    if (TrNode_symbol(child) == s) return child
    child = TrNode_next(child)
  }
  child = new_trie_node(s, NULL, parent, TrNode_child(parent))
  TrNode_child(parent) = child
  return child
}

```

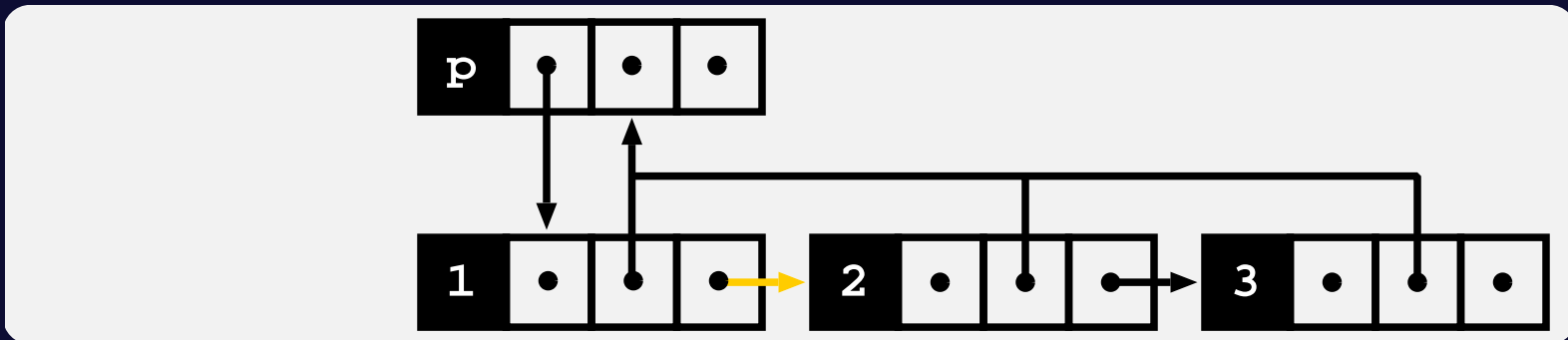


Inserindo um Termo na Tabela

```

trie_check_insert(symbol s, trie node parent) {
  child = TrNode_child(parent)
  while (child) {
    if (TrNode_symbol(child) == s) return child
    child = TrNode_next(child)
  }
  child = new_trie_node(s, NULL, parent, TrNode_child(parent))
  TrNode_child(parent) = child
  return child
}

```

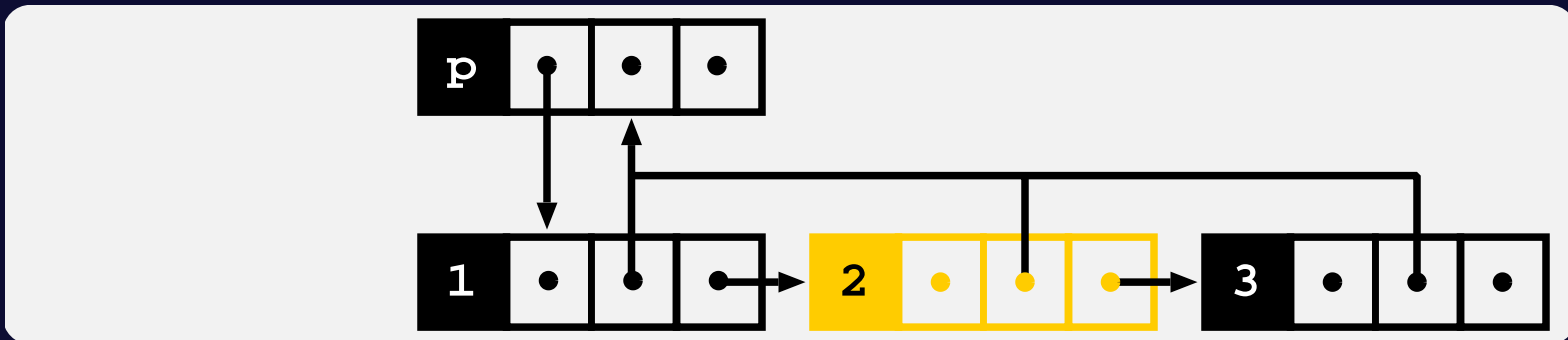


Inserindo um Termo na Tabela

```

trie_check_insert(symbol s, trie node parent) {
  child = TrNode_child(parent)
  while (child) {
    if (TrNode_symbol(child) == s) return child
    child = TrNode_next(child)
  }
  child = new_trie_node(s, NULL, parent, TrNode_child(parent))
  TrNode_child(parent) = child
  return child
}

```

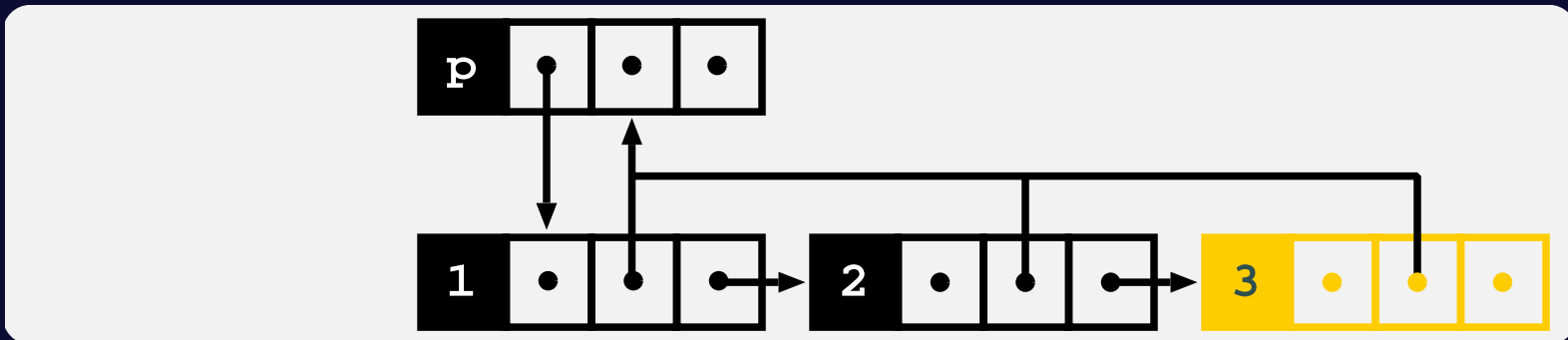


Inserindo um Termo na Tabela

```

trie_check_insert(symbol s, trie node parent) {
  child = TrNode_child(parent)
  while (child) {
    if (TrNode_symbol(child) == s) return child
    child = TrNode_next(child)
  }
  child = new_trie_node(s, NULL, parent, TrNode_child(parent))
  TrNode_child(parent) = child
  return child
}

```

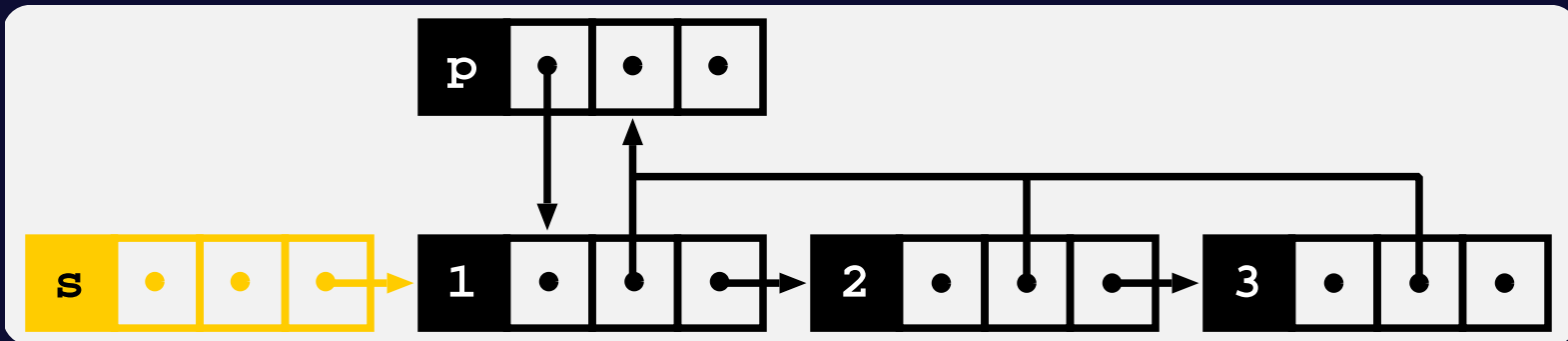


Inserindo um Termo na Tabela

```

trie_check_insert(symbol s, trie node parent) {
  child = TrNode_child(parent)
  while (child) {
    if (TrNode_symbol(child) == s) return child
    child = TrNode_next(child)
  }
  child = new_trie_node(s, NULL, parent, TrNode_child(parent))
  TrNode_child(parent) = child
  return child
}

```

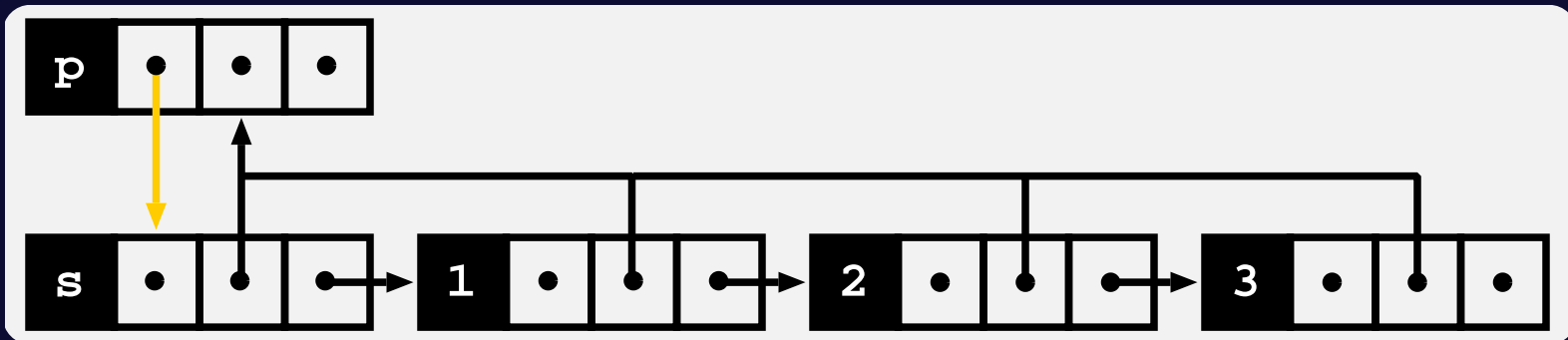


Inserindo um Termo na Tabela

```

trie_check_insert(symbol s, trie node parent) {
  child = TrNode_child(parent)
  while (child) {
    if (TrNode_symbol(child) == s) return child
    child = TrNode_next(child)
  }
  child = new_trie_node(s, NULL, parent, TrNode_child(parent))
  TrNode_child(parent) = child
  return child
}

```



Inserindo um Termo na Tabela

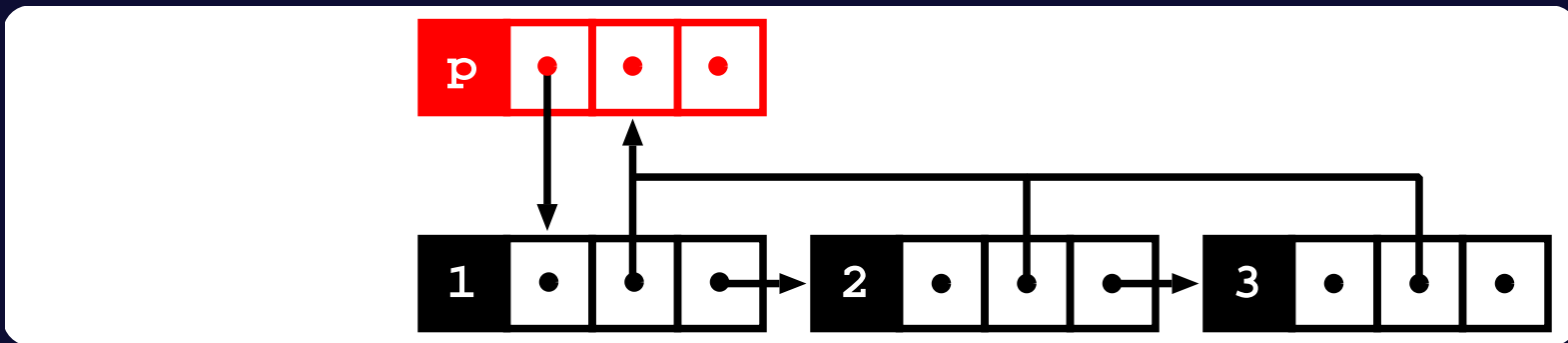
```
trie_check_insert(symbol s, trie node parent) {  
    child = TrNode_child(parent)  
    while (child) {  
        if (TrNode_symbol(child) == s) return child  
        child = TrNode_next(child)  
    }  
    child = new_trie_node(s, NULL, parent, TrNode_child(parent))  
    TrNode_child(parent) = child  
    return child  
}
```

TLNL: Table Lock at Node Level

```

trie_check_insert(symbol s, trie node parent) {
    lock(parent)                                // locking the parent node
    child = TrNode_child(parent)
    while (child) {
        if (TrNode_symbol(child) == s) {
            unlock(parent)                      // unlocking before return
            return child
        }
        child = TrNode_next(child)
    }
    child = new_trie_node(s, NULL, parent, TrNode_child(parent))
    TrNode_child(parent) = child
    unlock(parent)                              // unlocking before return
    return child
}

```

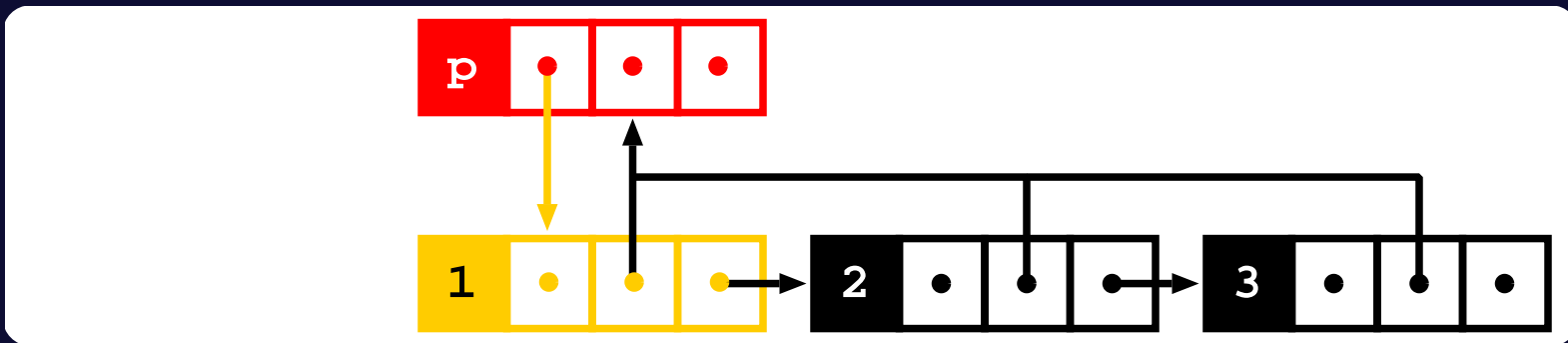


TLNL: Table Lock at Node Level

```

trie_check_insert(symbol s, trie node parent) {
    lock(parent)                                // locking the parent node
    child = TrNode_child(parent)
    while (child) {
        if (TrNode_symbol(child) == s) {
            unlock(parent)                      // unlocking before return
            return child
        }
        child = TrNode_next(child)
    }
    child = new_trie_node(s, NULL, parent, TrNode_child(parent))
    TrNode_child(parent) = child
    unlock(parent)                              // unlocking before return
    return child
}

```

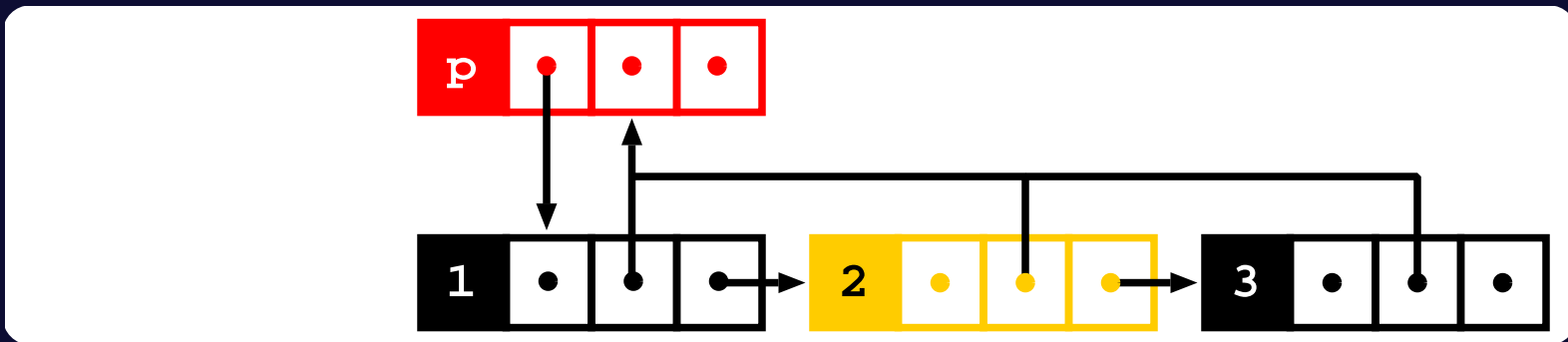


TLNL: Table Lock at Node Level

```

trie_check_insert(symbol s, trie node parent) {
    lock(parent)                                // locking the parent node
    child = TrNode_child(parent)
    while (child) {
        if (TrNode_symbol(child) == s) {
            unlock(parent)                      // unlocking before return
            return child
        }
        child = TrNode_next(child)
    }
    child = new_trie_node(s, NULL, parent, TrNode_child(parent))
    TrNode_child(parent) = child
    unlock(parent)                              // unlocking before return
    return child
}

```

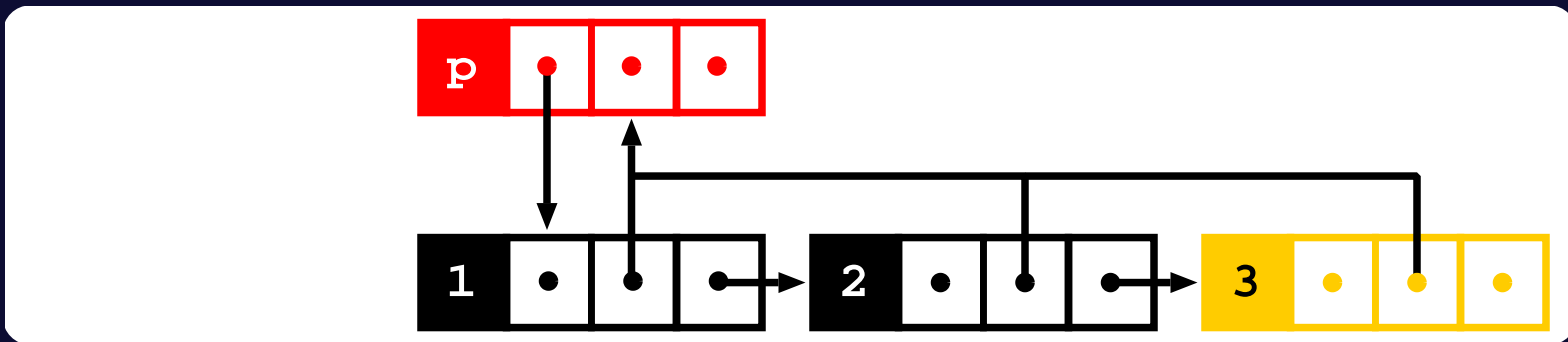


TLNL: Table Lock at Node Level

```

trie_check_insert(symbol s, trie node parent) {
    lock(parent)                                // locking the parent node
    child = TrNode_child(parent)
    while (child) {
        if (TrNode_symbol(child) == s) {
            unlock(parent)                      // unlocking before return
            return child
        }
        child = TrNode_next(child)
    }
    child = new_trie_node(s, NULL, parent, TrNode_child(parent))
    TrNode_child(parent) = child
    unlock(parent)                              // unlocking before return
    return child
}

```

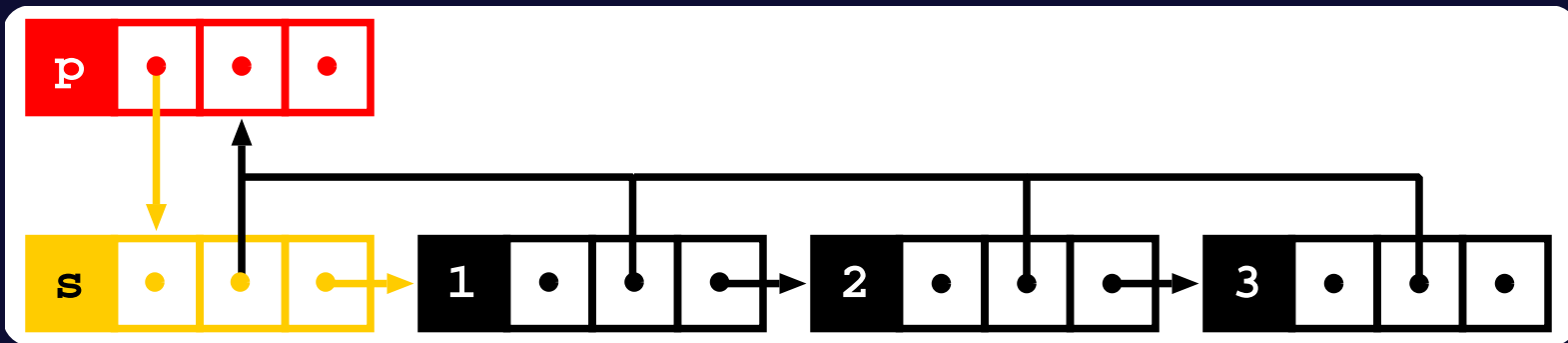


TLNL: Table Lock at Node Level

```

trie_check_insert(symbol s, trie node parent) {
    lock(parent)                                // locking the parent node
    child = TrNode_child(parent)
    while (child) {
        if (TrNode_symbol(child) == s) {
            unlock(parent)                      // unlocking before return
            return child
        }
        child = TrNode_next(child)
    }
    child = new_trie_node(s, NULL, parent, TrNode_child(parent))
    TrNode_child(parent) = child
    unlock(parent)                              // unlocking before return
    return child
}

```

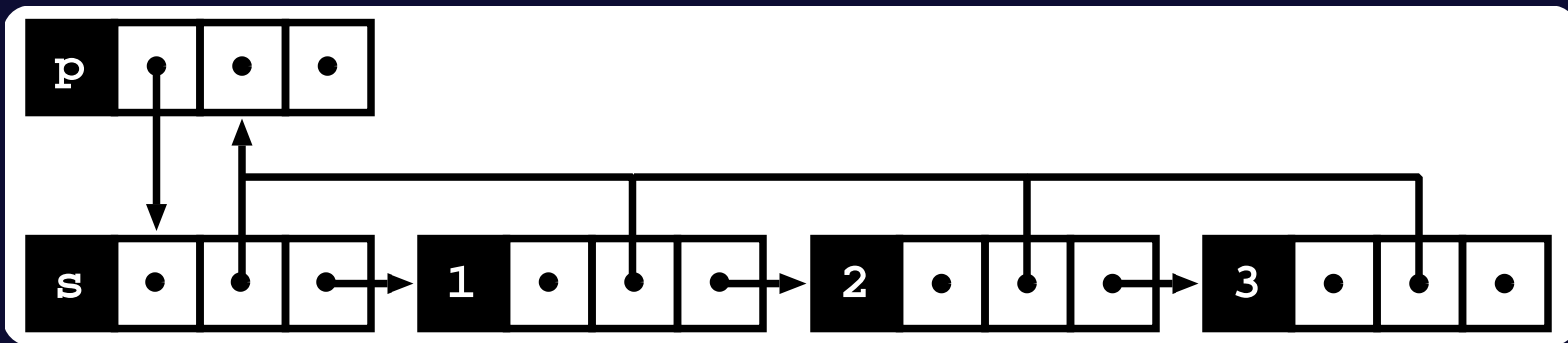


TLNL: Table Lock at Node Level

```

trie_check_insert(symbol s, trie node parent) {
    lock(parent)                                // locking the parent node
    child = TrNode_child(parent)
    while (child) {
        if (TrNode_symbol(child) == s) {
            unlock(parent)                      // unlocking before return
            return child
        }
        child = TrNode_next(child)
    }
    child = new_trie_node(s, NULL, parent, TrNode_child(parent))
    TrNode_child(parent) = child
    unlock(parent)                              // unlocking before return
    return child
}

```



TLNL: Table Lock at Node Level

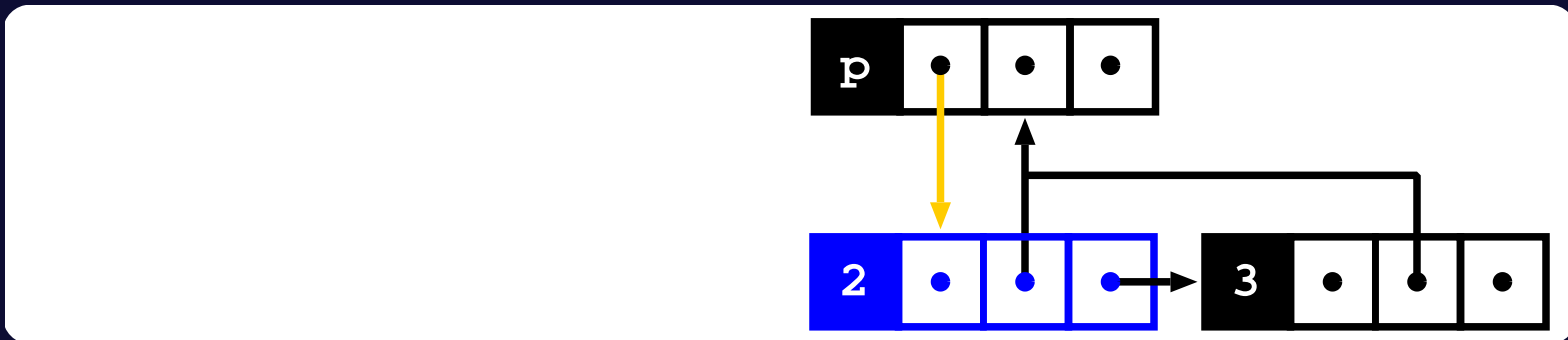
```
trie_check_insert(symbol s, trie node parent) {  
    lock(parent)                                // locking the parent node  
    child = TrNode_child(parent)  
    while (child) {  
        if (TrNode_symbol(child) == s) {  
            unlock(parent)                      // unlocking before return  
            return child  
        }  
        child = TrNode_next(child)  
    }  
    child = new_trie_node(s, NULL, parent, TrNode_child(parent))  
    TrNode_child(parent) = child  
    unlock(parent)                              // unlocking before return  
    return child  
}
```

TLWL: Table Lock at Write Level

```

trie_check_insert(symbol s, trie node parent) {
  child = TrNode_child(parent)
  initial_child = child                      // keep the initial child node
  while (child) {
    if (TrNode_symbol(child) == s) return child
    child = TrNode_next(child)
  }
  lock(parent)
  child = TrNode_child(parent)
  while (child != initial_child) {
    ...                                     // check nodes inserted in the meantime by others
  }
  ...
}

```

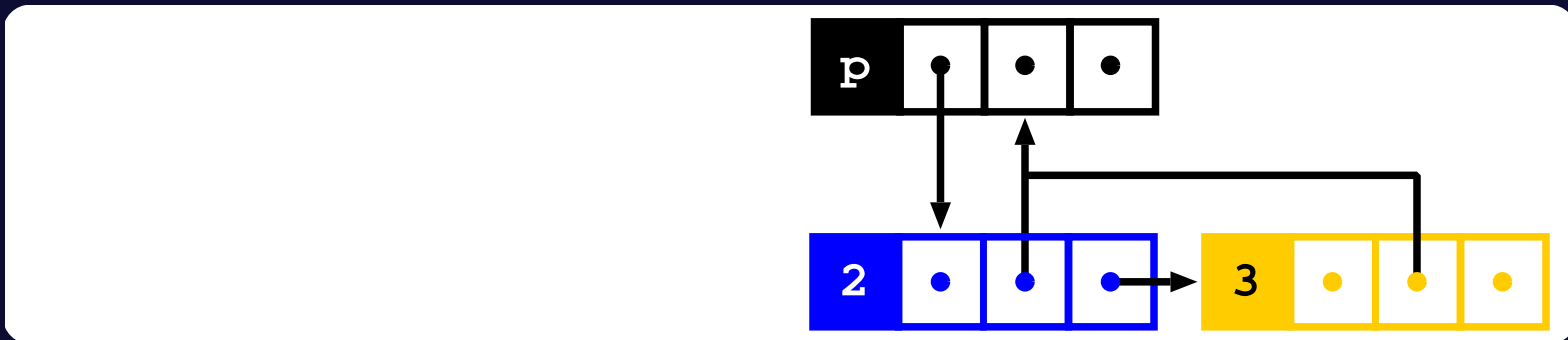


TLWL: Table Lock at Write Level

```

trie_check_insert(symbol s, trie node parent) {
  child = TrNode_child(parent)
  initial_child = child                                // keep the initial child node
  while (child) {
    if (TrNode_symbol(child) == s) return child
    child = TrNode_next(child)
  }
  lock(parent)
  child = TrNode_child(parent)
  while (child != initial_child) {
    ...                                                // check nodes inserted in the meantime by others
  }
  ...
}

```

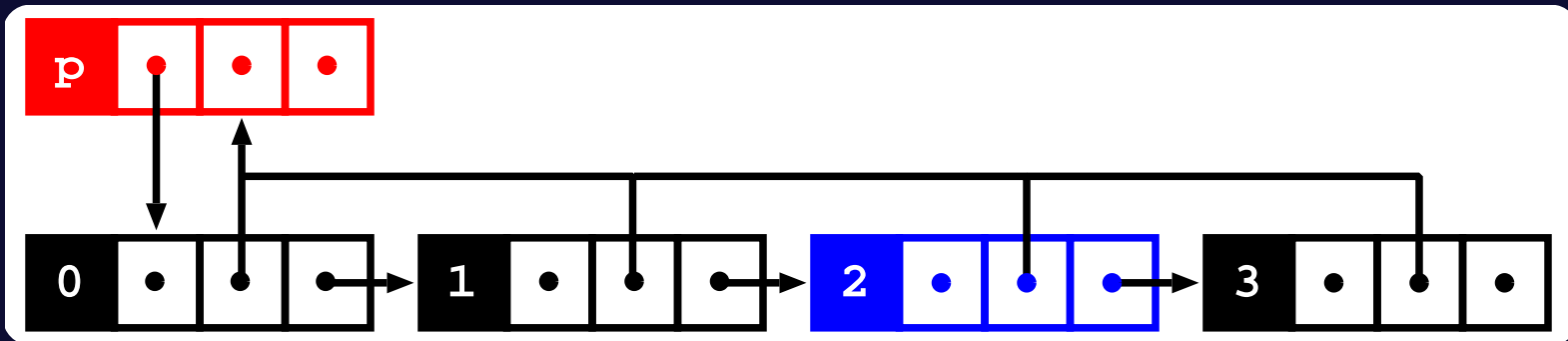


TLWL: Table Lock at Write Level

```

trie_check_insert(symbol s, trie node parent) {
  child = TrNode_child(parent)
  initial_child = child                                // keep the initial child node
  while (child) {
    if (TrNode_symbol(child) == s) return child
    child = TrNode_next(child)
  }
  lock(parent)
  child = TrNode_child(parent)
  while (child != initial_child) {
    ...                                                // check nodes inserted in the meantime by others
  }
  ...
}

```

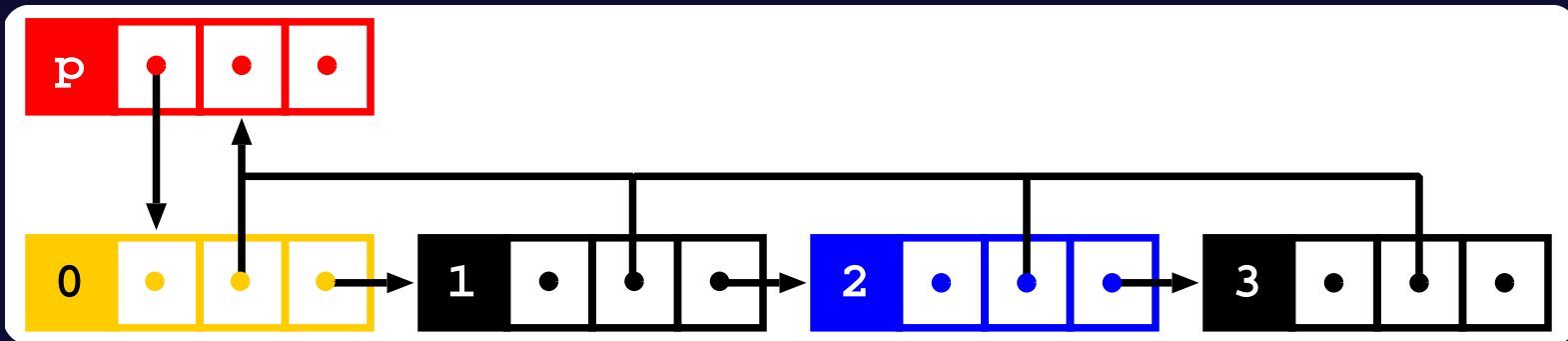


TLWL: Table Lock at Write Level

```

trie_check_insert(symbol s, trie node parent) {
  child = TrNode_child(parent)
  initial_child = child                                // keep the initial child node
  while (child) {
    if (TrNode_symbol(child) == s) return child
    child = TrNode_next(child)
  }
  lock(parent)
  child = TrNode_child(parent)
  while (child != initial_child) {
    ...                                                // check nodes inserted in the meantime by others
  }
  ...
}

```

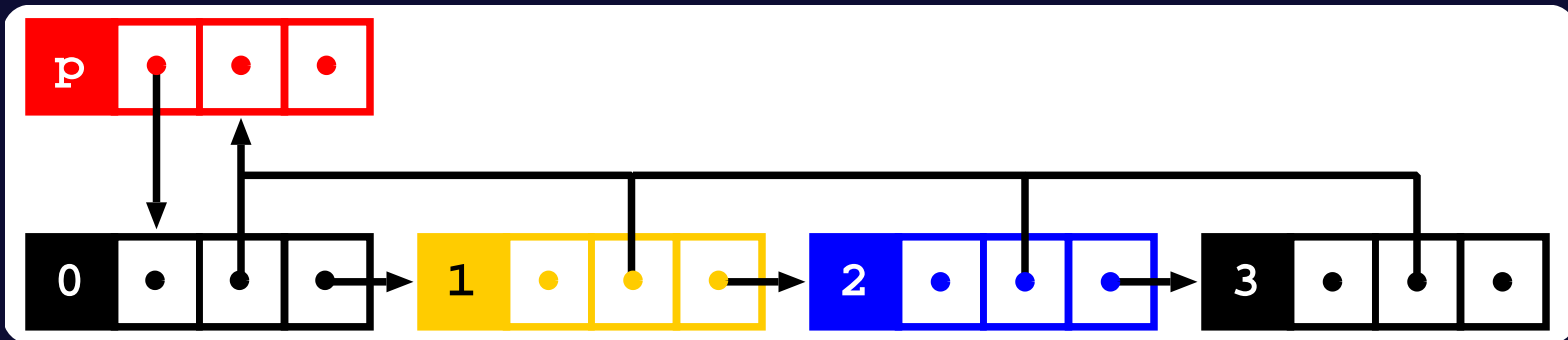


TLWL: Table Lock at Write Level

```

trie_check_insert(symbol s, trie node parent) {
  child = TrNode_child(parent)
  initial_child = child                                // keep the initial child node
  while (child) {
    if (TrNode_symbol(child) == s) return child
    child = TrNode_next(child)
  }
  lock(parent)
  child = TrNode_child(parent)
  while (child != initial_child) {
    ...                                                // check nodes inserted in the meantime by others
  }
  ...
}

```

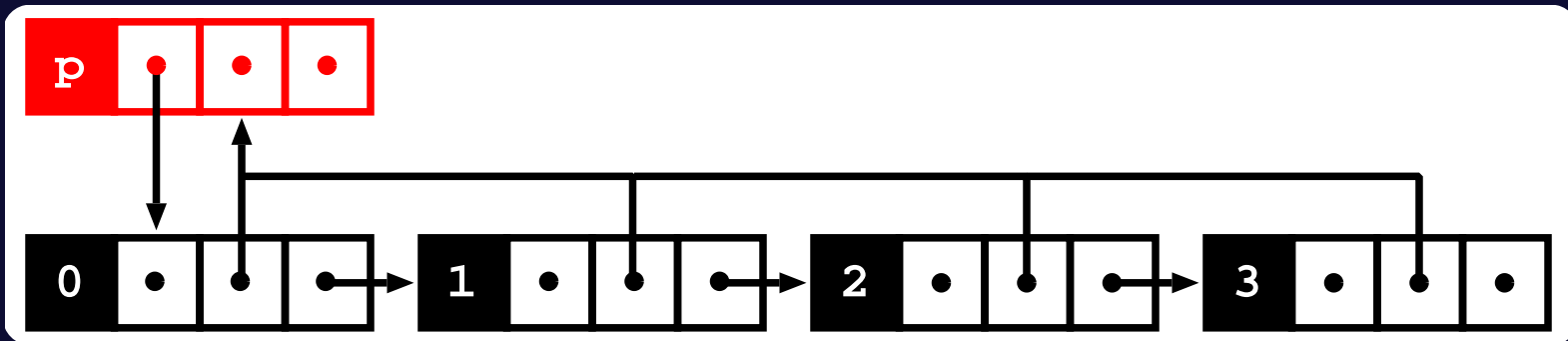


TLWL: Table Lock at Write Level

```

trie_check_insert(symbol s, trie node parent) {
  child = TrNode_child(parent)
  initial_child = child                                // keep the initial child node
  while (child) {
    if (TrNode_symbol(child) == s) return child
    child = TrNode_next(child)
  }
  lock(parent)
  child = TrNode_child(parent)
  while (child != initial_child) {
    ...                                                // check nodes inserted in the meantime by others
  }
  ...
}

```



TLWL: Table Lock at Write Level

```
trie_check_insert(symbol s, trie node parent) {  
    child = TrNode_child(parent)  
    initial_child = child                // keep the initial child node  
    while (child) {  
        if (TrNode_symbol(child) == s) return child  
        child = TrNode_next(child)  
    }  
    lock(parent)  
    child = TrNode_child(parent)  
    while (child != initial_child) {  
        ...                            // check nodes inserted in the meantime by others  
    }  
    ...  
}
```

TLWL-ABC: Table Lock at Write Level-Allocate Before Check

```

trie_check_insert(symbol s, trie node parent) {
    ...
    pre_alloc = new_trie_node(s, NULL, parent, NULL)
    lock(parent)
    child = TrNode_child(parent)
    while (child != initial_child) {
        if (TrNode_symbol(child) == s) {
            unlock(parent)
            free(pre_alloc)
            return child
        }
        child = TrNode_next(child)
    }
    TrNode_next(pre_alloc) = TrNode_child(parent)
    TrNode_child(parent) = pre_alloc
    unlock(parent)
    return pre_alloc
}

```

// the same as TLWL
 // pre-allocate ...
 // ... node before locking
 // free the pre-allocated node



TLWL-ABC: Table Lock at Write Level-Allocate Before Check

```

trie_check_insert(symbol s, trie node parent) {
    ...
    pre_alloc = new_trie_node(s, NULL, parent, NULL)
    lock(parent)
    child = TrNode_child(parent)
    while (child != initial_child) {
        if (TrNode_symbol(child) == s) {
            unlock(parent)
            free(pre_alloc)
            return child
        }
        child = TrNode_next(child)
    }
    TrNode_next(pre_alloc) = TrNode_child(parent)
    TrNode_child(parent) = pre_alloc
    unlock(parent)
    return pre_alloc
}

```

// the same as TLWL
 // pre-allocate ...
 // ... node before locking
 // free the pre-allocated node



TLWL-ABC: Table Lock at Write Level-Allocate Before Check

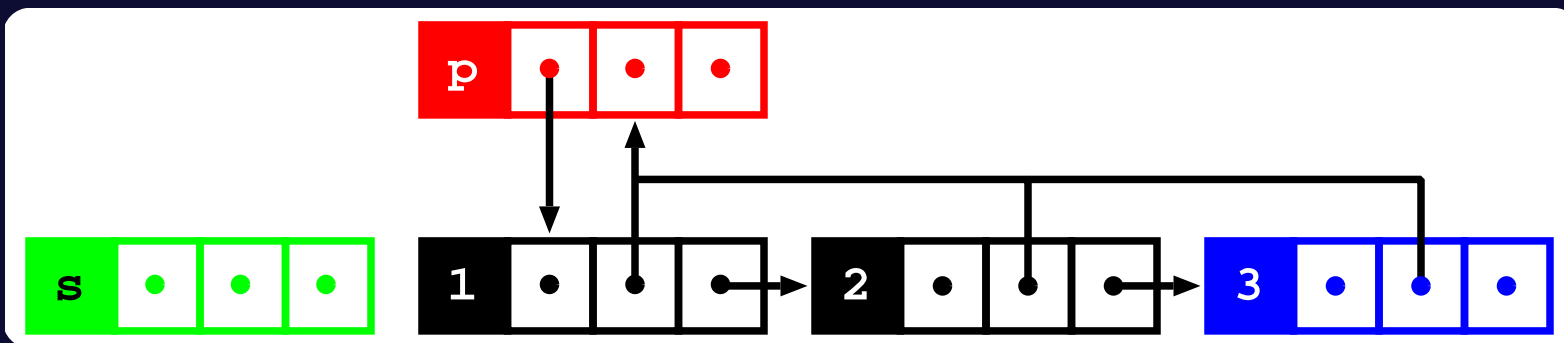
```

trie_check_insert(symbol s, trie node parent) {
    ...
    pre_alloc = new_trie_node(s, NULL, parent, NULL)
    lock(parent)
    child = TrNode_child(parent)
    while (child != initial_child) {
        if (TrNode_symbol(child) == s) {
            unlock(parent)
            free(pre_alloc)
            return child
        }
        child = TrNode_next(child)
    }
    TrNode_next(pre_alloc) = TrNode_child(parent)
    TrNode_child(parent) = pre_alloc
    unlock(parent)
    return pre_alloc
}

```

// the same as TLWL
 // pre-allocate ...
 // ... node before locking

 // free the pre-allocated node



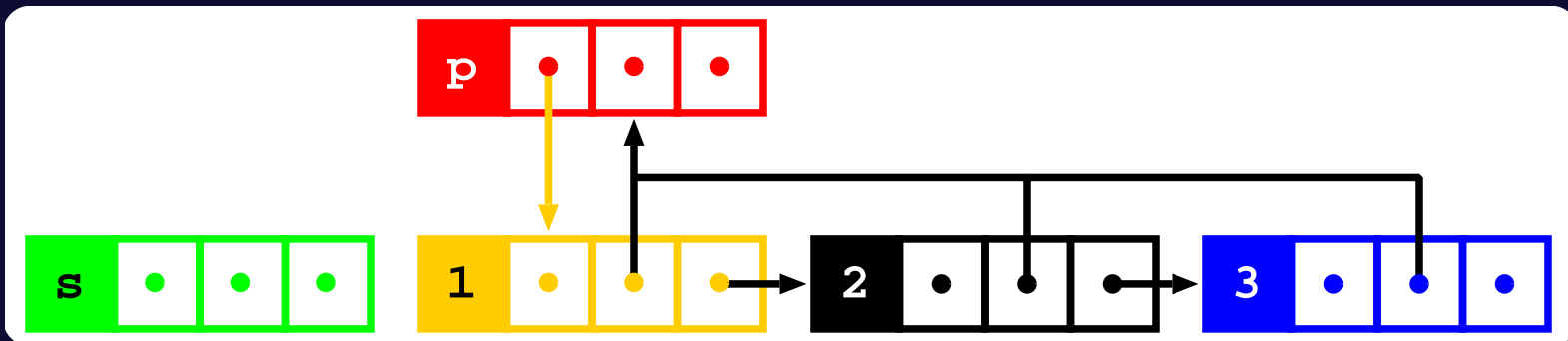
TLWL-ABC: Table Lock at Write Level-Allocate Before Check

```

trie_check_insert(symbol s, trie node parent) {
    ...
    pre_alloc = new_trie_node(s, NULL, parent, NULL)
    lock(parent)
    child = TrNode_child(parent)
    while (child != initial_child) {
        if (TrNode_symbol(child) == s) {
            unlock(parent)
            free(pre_alloc)
            return child
        }
        child = TrNode_next(child)
    }
    TrNode_next(pre_alloc) = TrNode_child(parent)
    TrNode_child(parent) = pre_alloc
    unlock(parent)
    return pre_alloc
}

```

// the same as TLWL
 // pre-allocate ...
 // ... node before locking
 // free the pre-allocated node



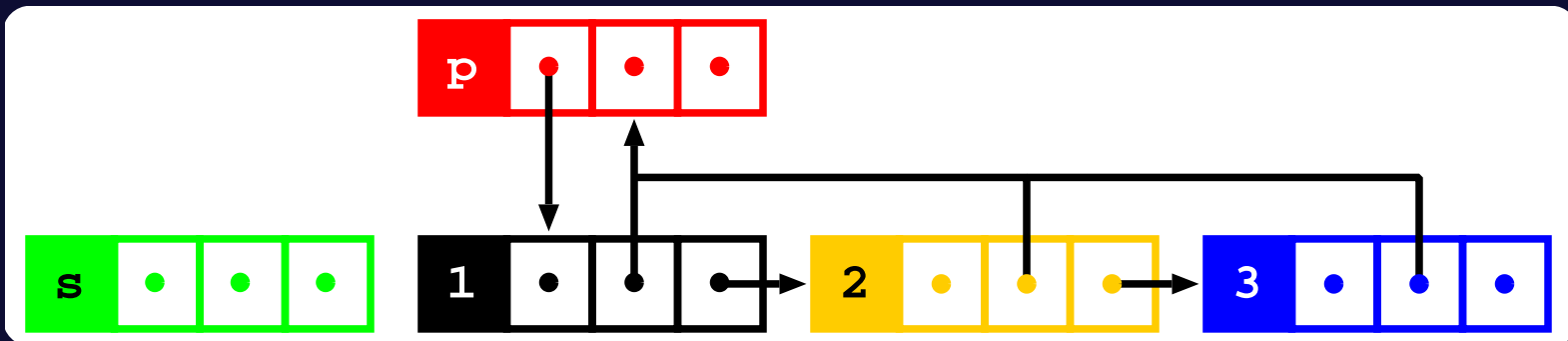
TLWL-ABC: Table Lock at Write Level-Allocate Before Check

```

trie_check_insert(symbol s, trie node parent) {
    ...
    pre_alloc = new_trie_node(s, NULL, parent, NULL)
    lock(parent)
    child = TrNode_child(parent)
    while (child != initial_child) {
        if (TrNode_symbol(child) == s) {
            unlock(parent)
            free(pre_alloc)
            return child
        }
        child = TrNode_next(child)
    }
    TrNode_next(pre_alloc) = TrNode_child(parent)
    TrNode_child(parent) = pre_alloc
    unlock(parent)
    return pre_alloc
}

```

// the same as TLWL
 // pre-allocate ...
 // ... node before locking
 // free the pre-allocated node



TLWL-ABC: Table Lock at Write Level-Allocate Before Check

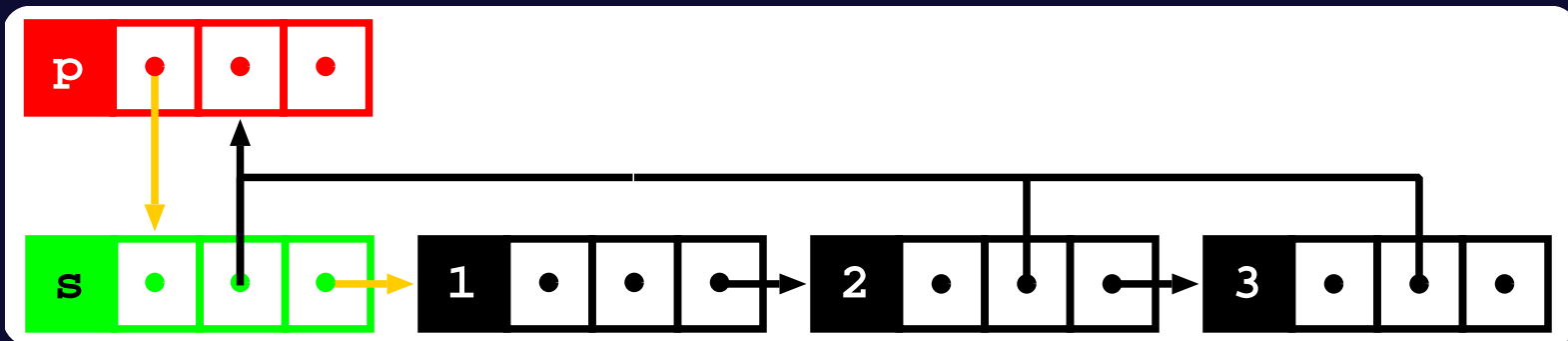
```

trie_check_insert(symbol s, trie node parent) {
    ...
    pre_alloc = new_trie_node(s, NULL, parent, NULL)
    lock(parent)
    child = TrNode_child(parent)
    while (child != initial_child) {
        if (TrNode_symbol(child) == s) {
            unlock(parent)
            free(pre_alloc)
            return child
        }
        child = TrNode_next(child)
    }
    TrNode_next(pre_alloc) = TrNode_child(parent)
    TrNode_child(parent) = pre_alloc
    unlock(parent)
    return pre_alloc
}

```

// the same as TLWL
 // pre-allocate ...
 // ... node before locking

 // free the pre-allocated node



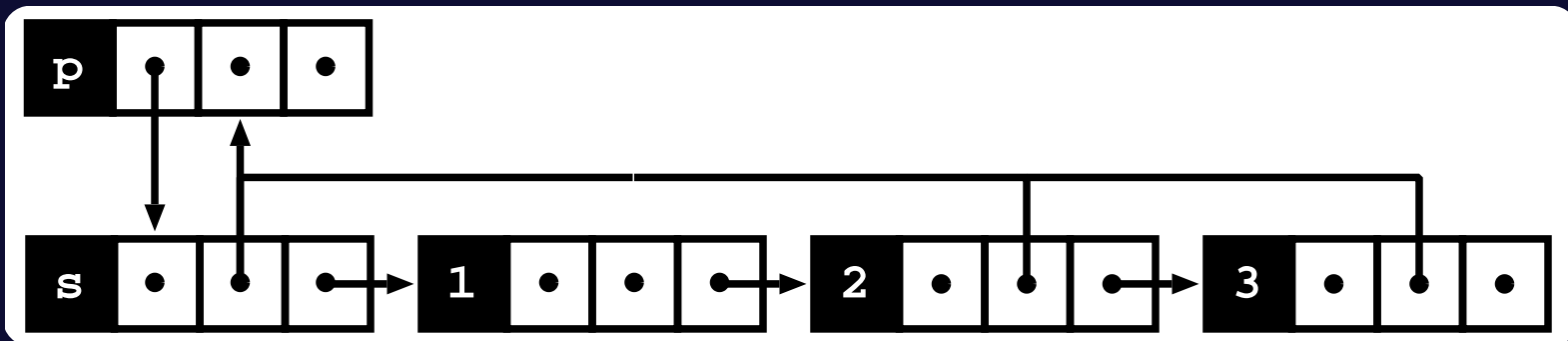
TLWL-ABC: Table Lock at Write Level-Allocate Before Check

```

trie_check_insert(symbol s, trie node parent) {
    ...
    pre_alloc = new_trie_node(s, NULL, parent, NULL)
    lock(parent)
    child = TrNode_child(parent)
    while (child != initial_child) {
        if (TrNode_symbol(child) == s) {
            unlock(parent)
            free(pre_alloc)
            return child
        }
        child = TrNode_next(child)
    }
    TrNode_next(pre_alloc) = TrNode_child(parent)
    TrNode_child(parent) = pre_alloc
    unlock(parent)
    return pre_alloc
}

```

// the same as TLWL
 // pre-allocate ...
 // ... node before locking
 // free the pre-allocated node



TLWL-ABC: Table Lock at Write Level-Allocate Before Check

```
trie_check_insert(symbol s, trie node parent) {  
    ... // the same as TLWL  
    pre_alloc = new_trie_node(s, NULL, parent, NULL) // pre-allocate ...  
    lock(parent) // ... node before locking  
    child = TrNode_child(parent)  
    while (child != initial_child) {  
        if (TrNode_symbol(child) == s) {  
            unlock(parent)  
            free(pre_alloc) // free the pre-allocated node  
            return child  
        }  
        child = TrNode_next(child)  
    }  
    TrNode_next(pre_alloc) = TrNode_child(parent)  
    TrNode_child(parent) = pre_alloc // insert the pre-allocated node  
    unlock(parent)  
    return pre_alloc  
}
```

Esquemas de Locking: Resumo

➤ TLNL

- ◆ Lock count é proporcional ao tamanho do termo.
- ◆ Lock duration é proporcional ao tempo necessário para percorrer os nós filho.

➤ TLWL

- ◆ Lock count varia entre 0 e o tamanho do termo.
- ◆ Lock duration pode ser
 - 0, se o nó já existe na cadeia inicial;
 - proporcional aos nós filhos adicionados entretanto mais o tempo necessário para alocar o novo nó.

➤ TLWL-ABC

- ◆ Lock count varia entre 0 e o tamanho do termo.
- ◆ Lock duration pode ser
 - 0, se o nó já existe na cadeia inicial;
 - proporcional aos nós filhos adicionados entretanto.

Os Nossos Protótipos

➤ **YapTab**

Extensão do Yap Prolog para a execução de Prolog com tabulação. Baseia-se no modelo SLG-WAM para implementar a execução de predicados tabelados.

➤ **OPTYap**

Extensão do Yap Prolog para a execução paralela de Prolog com tabulação. Baseia-se nos modelos de cópia de ambientes e SLG-WAM para explorar Paralelismo-Ou de forma implícita na execução de predicados tabelados.

Alguns Resultados: Programas Sem Tabulação

Programa	Yap	YapOr	YapTab	OPTYap	XSB
cubes	1.97	2.06 (1.05)	2.05 (1.04)	2.16 (1.10)	4.81 (2.44)
ham	4.04	4.61 (1.14)	4.28 (1.06)	4.95 (1.23)	10.36 (2.56)
map	9.01	10.25 (1.14)	9.19 (1.02)	11.08 (1.23)	24.11 (2.68)
nsort	33.05	37.52 (1.14)	35.85 (1.08)	39.95 (1.21)	83.72 (2.53)
puzzle	2.04	2.22 (1.09)	2.19 (1.07)	2.36 (1.16)	4.97 (2.44)
queens	16.77	17.68 (1.05)	17.58 (1.05)	18.57 (1.11)	36.40 (2.17)
Média		(1.10)	(1.05)	(1.17)	(2.47)

- Tempos de execução (em segundos) obtidos numa máquina paralela Silicon Graphics Cray Origin2000 com 96 processadores MIPS 195 MHz R10000.

Alguns Resultados: Programas Com Tabulação

Programa	YapTab	OPTYap	XSB
mc-sieve	235.31	268.13 (1.14)	433.53 (1.84)
mc-leader	76.60	85.56 (1.12)	158.23 (2.07)
mc-iproto	20.73	23.68 (1.14)	53.04 (2.56)
samegen	23.36	26.00 (1.11)	37.91 (1.62)
lgrid	3.55	4.28 (1.21)	7.41 (2.09)
lgrid/2	59.53	69.02 (1.16)	98.22 (1.65)
rgrid/2	6.24	7.51 (1.20)	15.40 (2.47)
Média		(1.15)	(2.04)

- Tempos de execução (em segundos) obtidos numa máquina paralela Silicon Graphics Cray Origin2000 com 96 processadores MIPS 195 MHz R10000.

Alguns Resultados: Programas Com Tabulação em Paralelo

Programa	Esquema	8 CPUs	16 CPUs	24 CPUs	32 CPUs
mc-sieve	TLNL	7.2	11.8	3.9	4.7
	TLWL	7.9	15.8	23.7	31.5
	TLWL-ABC	7.9	15.8	23.7	31.4
mc-iproto	TLNL	2.6	1.8	1.0	1.0
	TLWL	5.0	9.0	8.8	7.2
	TLWL-ABC	5.1	7.7	8.4	7.1
samegen	TLNL	7.2	13.8	19.6	24.0
	TLWL	7.2	13.9	19.7	24.1
	TLWL-ABC	7.2	13.9	19.7	24.2
lgrid	TLNL	6.7	12.1	6.2	5.3
	TLWL	7.1	13.5	19.9	24.3
	TLWL-ABC	6.9	13.4	18.9	24.2

- O esquema TLNL dificilmente consegue escalar para mais do que 16 agentes.
- A estratégia menos refinada do esquema TLWL obteve resultados semelhantes à do esquema TLWL-ABC.

Alguns Resultados: Programas Com Tabulação Incompleta

Modo	Configuração I	Configuração II
Sem Tabulação	> 1 day	> 1 day
Com Tabulação	278.2	137.9
Com Tabulação Incompleta	122.9	117.6

- Tempos de execução (em segundos) obtidos num Pentium M 1600MHz ao executar o sistema ILP April para o dataset Mutagenesis com 2 configurações diferentes.
- As configurações I e II chamam respectivamente 1479 e 1461 objectivos tabelados diferentes e ambas terminam com 76 tabelas incompletas.

Trabalho Futuro

➤ Projecto STAMPA

Sophisticated TAbling Mechanisms for Prolog and their Applications

- ◆ **Entidade Financiadora:** FCT (PTDC/EIA/67738/2006)
- ◆ **Financiamento:** 150.000 Euros
- ◆ **Data de Início:** Janeiro de 2008
- ◆ **Principais Objectivos:** o projecto visa estudar como combinar a tecnologia de tabulação com diferentes modelos de execução de Programação em Lógica, nomeadamente os modelos usados em *Answer Set Programming* e no *Extended Andorra Model*, e contribuir para o estado da arte dos actuais sistemas de tabulação através do desenvolvimento de novas técnicas e funcionalidades originais fortemente guiadas pela sua validação usando aplicações concretas de Indução de Programas em Lógica e Bases de Dados Dedutivas.
- ◆ **URL:** <http://www.dcc.fc.up.pt/~ricroc/homepage/projects/stampa>

Bibliografia

- **YapTab: A Tabling Engine Designed to Support Parallelism.** R. Rocha, F. Silva e V. Santos Costa. TAPD. 2000.
- **On a Tabling Engine That Can Exploit Or-Parallelism.** R. Rocha, F. Silva e V. Santos Costa. ICLP, Springer-Verlag LNCS 2237, páginas 43-58. 2001.
- **Concurrent Table Accesses in Parallel Tabled Logic Programs.** R. Rocha, F. Silva e V. Santos Costa. Euro-Par, Springer-Verlag LNCS 3149, páginas 662-670. 2004.
- **On Applying Tabling to Inductive Logic Programming.** R. Rocha, N. Fonseca e V. Santos Costa. ECML, Springer-Verlag LNAI 3720, páginas 707-714. 2005.
- **On Improving the Efficiency and Robustness of Table Storage Mechanisms for Tabled Evaluation.** R. Rocha. PADL, Springer-Verlag LNCS 4354, páginas 155-169. 2007.