Concurrent Table Accesses in Parallel Tabled Logic Programs

Ricardo Rocha Fernando Silva
{*ricroc,fds*}*@ncc.up.pt*
DCC-FC & LIACC, University of Porto, Portugal

Vítor Santos Costa
vitor@cos.ufrj.br
COPPE Systems & LIACC, University of Rio de Janeiro, Brazil

Tabling in Logic Programming

- **Tabling** is an implementation technique where results for subcomputations are stored in a **table space** and then reused when a repeated computation appears.

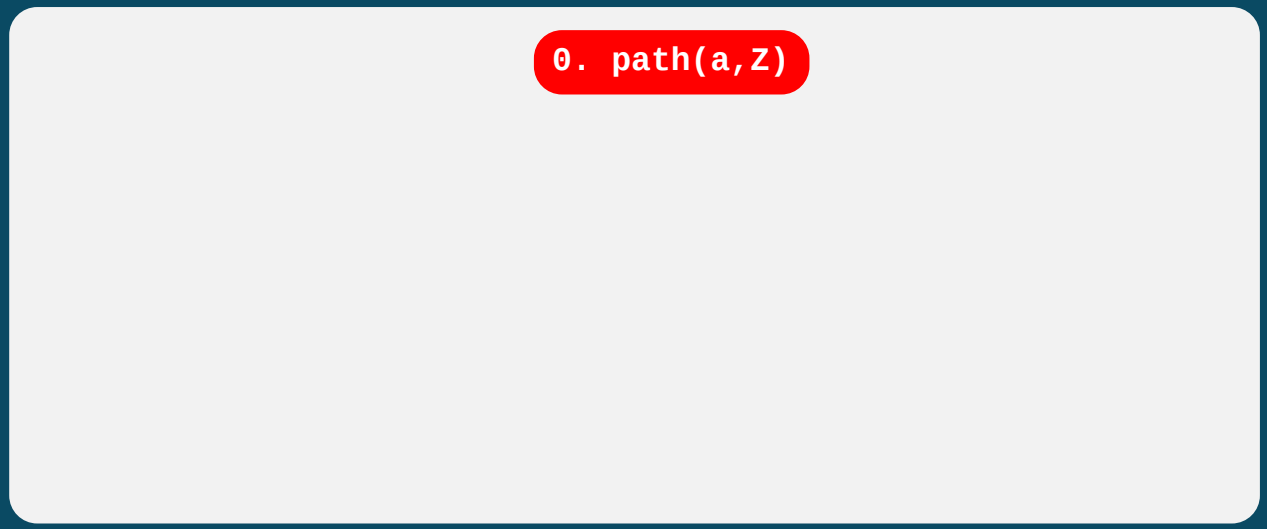
- Tabling has proven to be particularly effective in logic (**Prolog**) programs:
  - Avoids recomputation, thus reducing the search space.
  - Avoids infinite loops, thus ensuring termination for a wider class of programs.

- Tabling has been successfully applied to real applications:
  - Model Checking
  - Program Analysis
  - Deductive Databases
  - Non-Monotonic Reasoning
  - Natural Language Processing

Parallel Tabling

- Tabled programs show great potential for **parallel execution**:
  - Programs still need to exploit alternatives for solving goals.
  - Programs often perform search.
  - Programs do not depend on answer ordering.
  - Programs with long running times.
- We thus developed **OPTYap**:
  - The first parallel tabling system for logic programming.
  - Exploits implicit or-parallelism from tabled logic programs.
  - Designed for shared-memory machines.
- However, performance for real applications might never scale:
  - Parallel tabling requires a more **complex computational model** than traditional parallel Prolog models [ICLP'01, IPDPS'02, ICLP'04].
  - Parallelism introduces **concurrency** for table access [EuroPar'04].

Tabling Execution Model



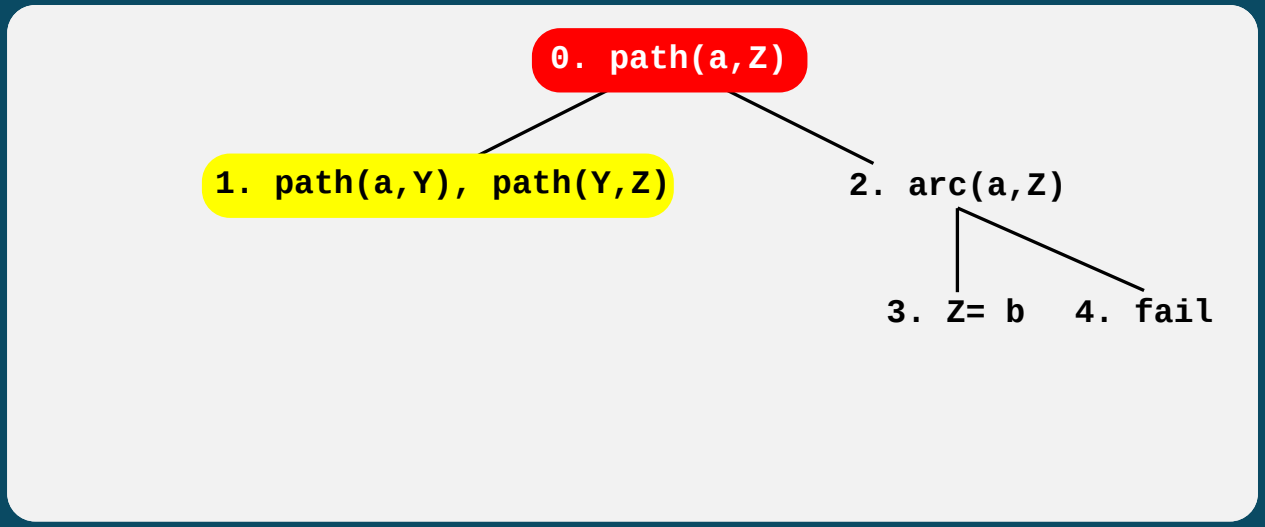
Program Code

```
:- table path/2.

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

arc(a,b).
arc(b,a).

?- path(a,Z).
```

Table Space

Subgoal	Answers
0. path(a,Z)	

Tabling Execution Model



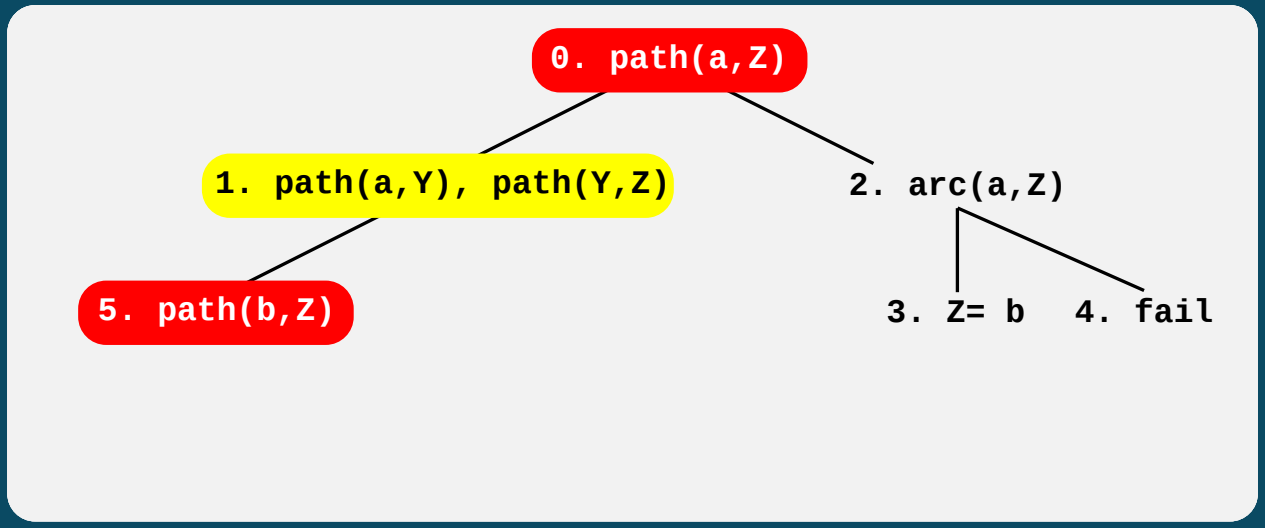
Program Code

```
:- table path/2.

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

arc(a,b).
arc(b,a).

?- path(a,Z).
```

Table Space

Subgoal	Answers
0. path(a,Z)	3. Z= b

Tabling Execution Model



Program Code

```
:- table path/2.

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

arc(a,b).
arc(b,a).

?- path(a,Z).
```

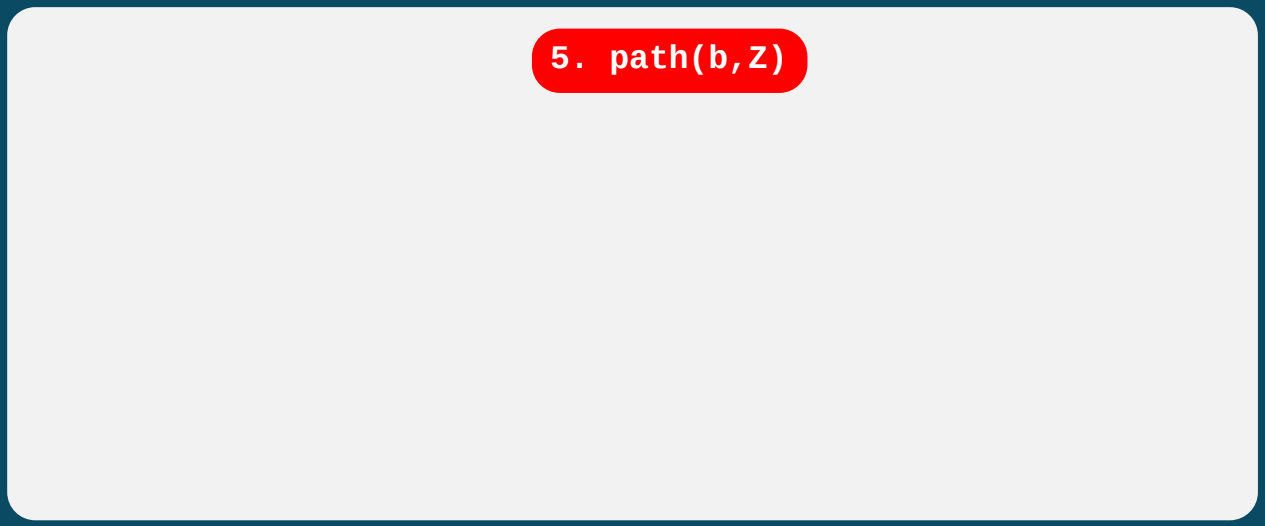
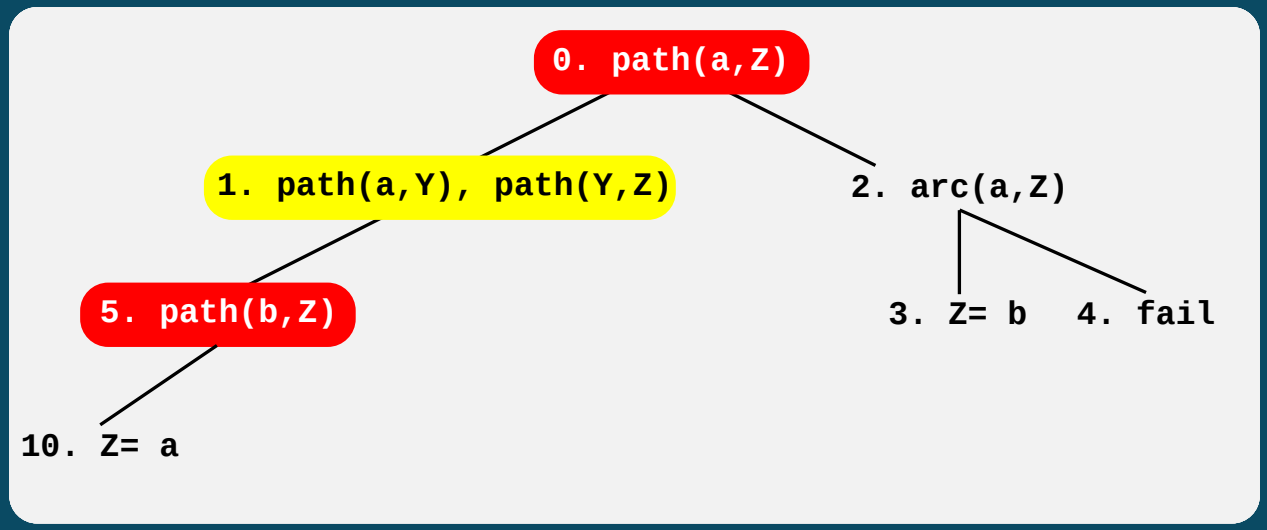


Table Space

Subgoal	Answers
0. path(a,Z)	3. Z= b
5. path(b,Z)	

Tabling Execution Model



Program Code

```
:- table path/2.

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

arc(a,b).
arc(b,a).

?- path(a,Z).
```

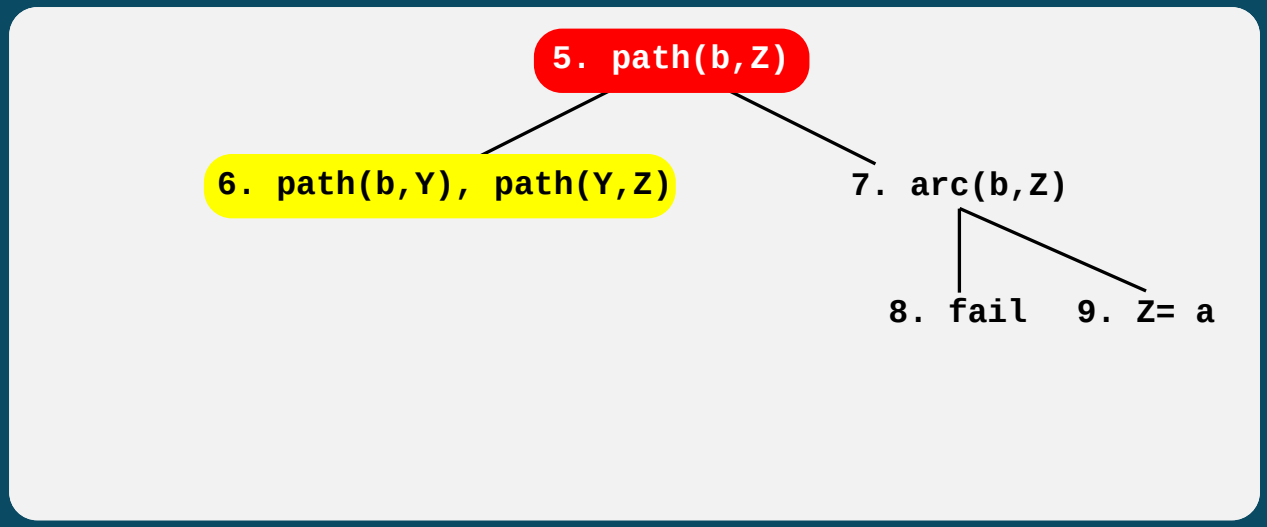
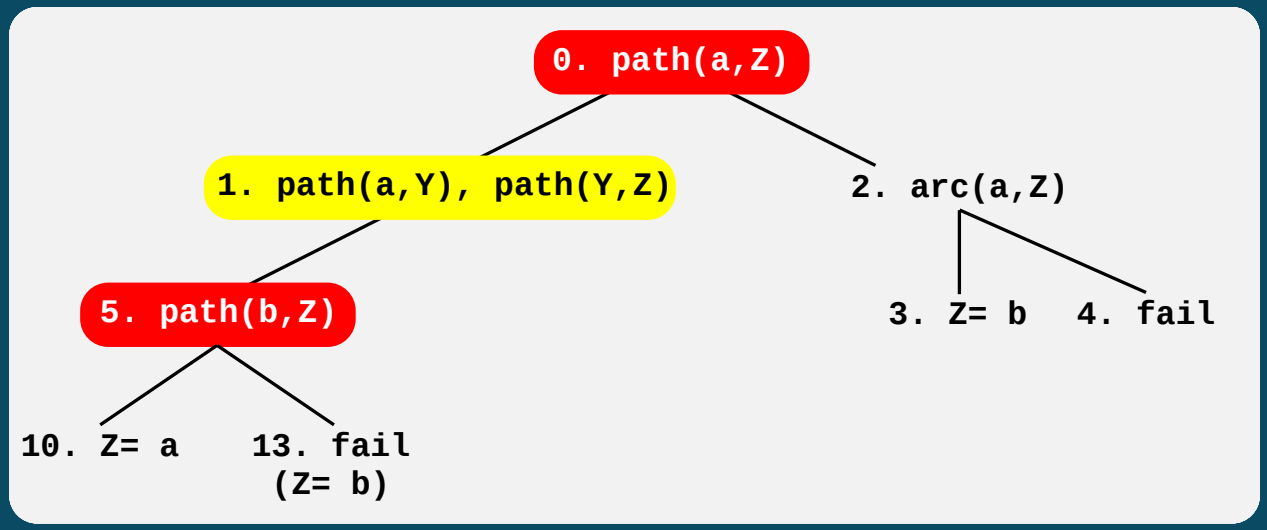


Table Space

Subgoal	Answers
0. path(a,Z)	3. Z= b 10. Z= a
5. path(b,Z)	9. Z= a

Tabling Execution Model



Program Code

```
:- table path/2.

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

arc(a,b).
arc(b,a).

?- path(a,Z).
```

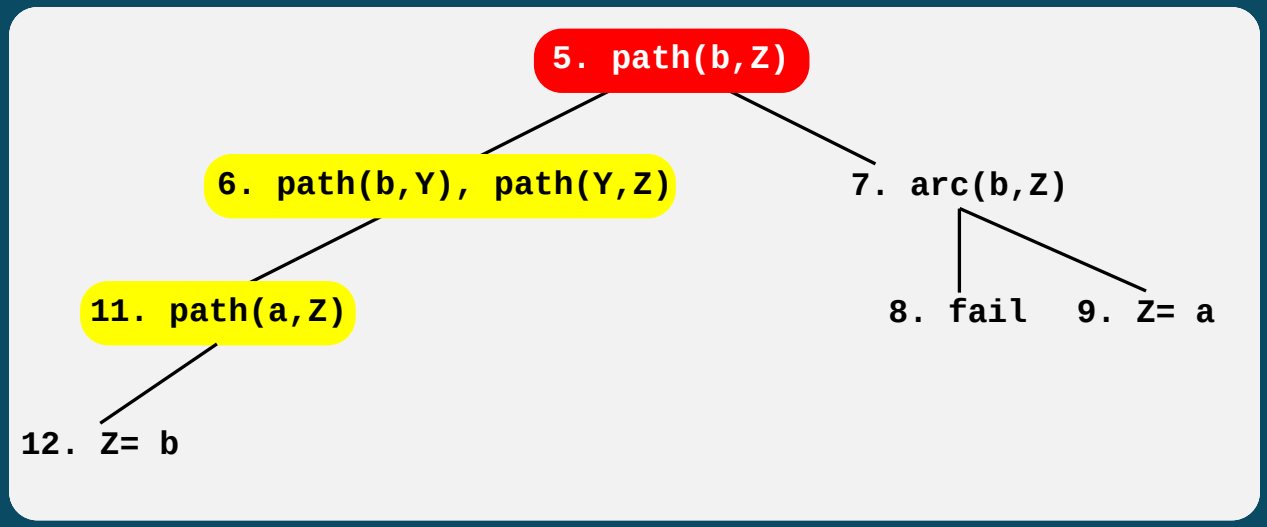
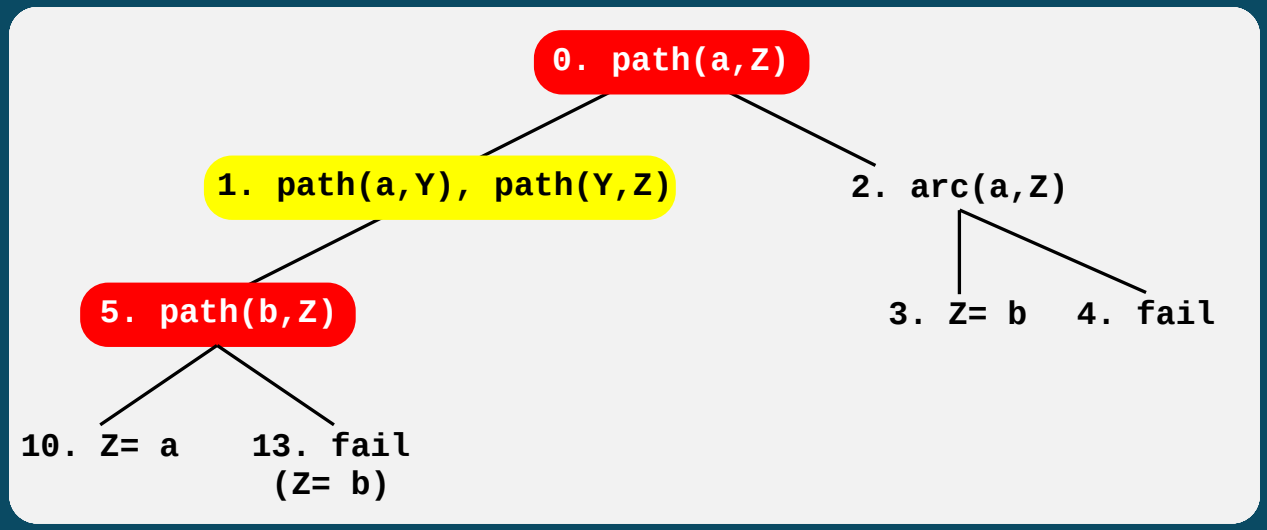


Table Space

Subgoal	Answers
0. path(a,Z)	3. Z= b 10. Z= a
5. path(b,Z)	9. Z= a 12. Z= b

Tabling Execution Model



Program Code

```
:- table path/2.

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

arc(a,b).
arc(b,a).

?- path(a,Z).
```

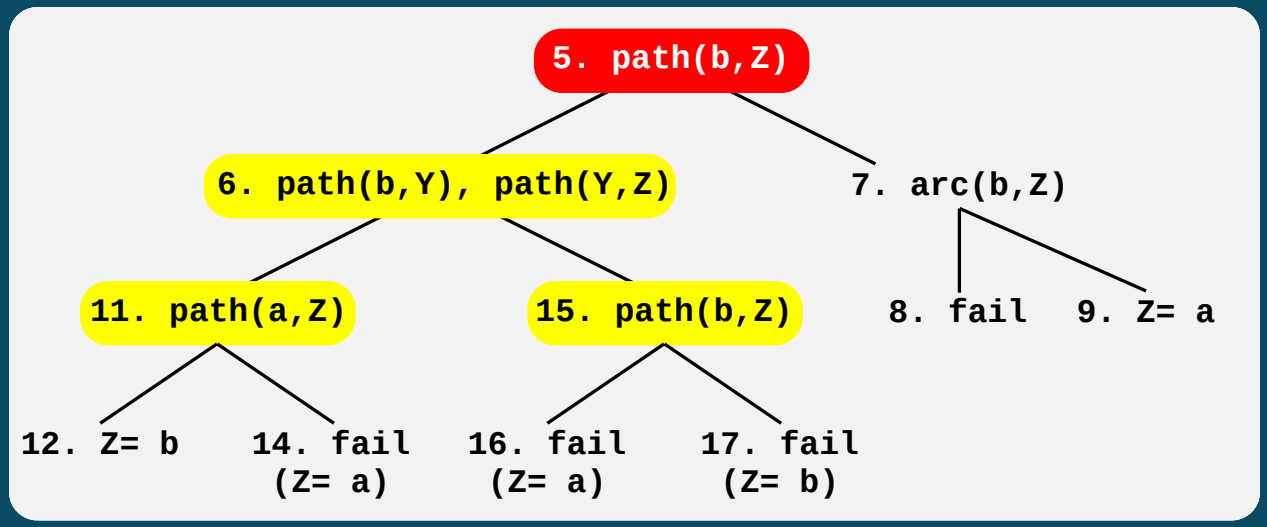
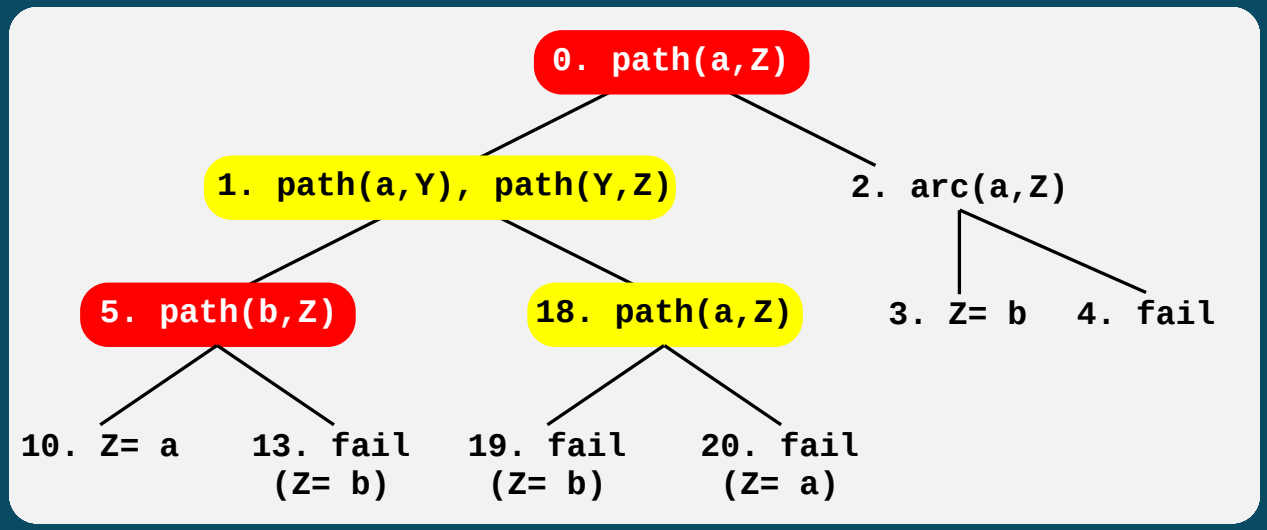


Table Space

Subgoal	Answers
0. path(a,Z)	3. Z= b 10. Z= a
5. path(b,Z)	9. Z= a 12. Z= b

Tabling Execution Model



Program Code

```
:- table path/2.

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

arc(a,b).
arc(b,a).

?- path(a,Z).
```

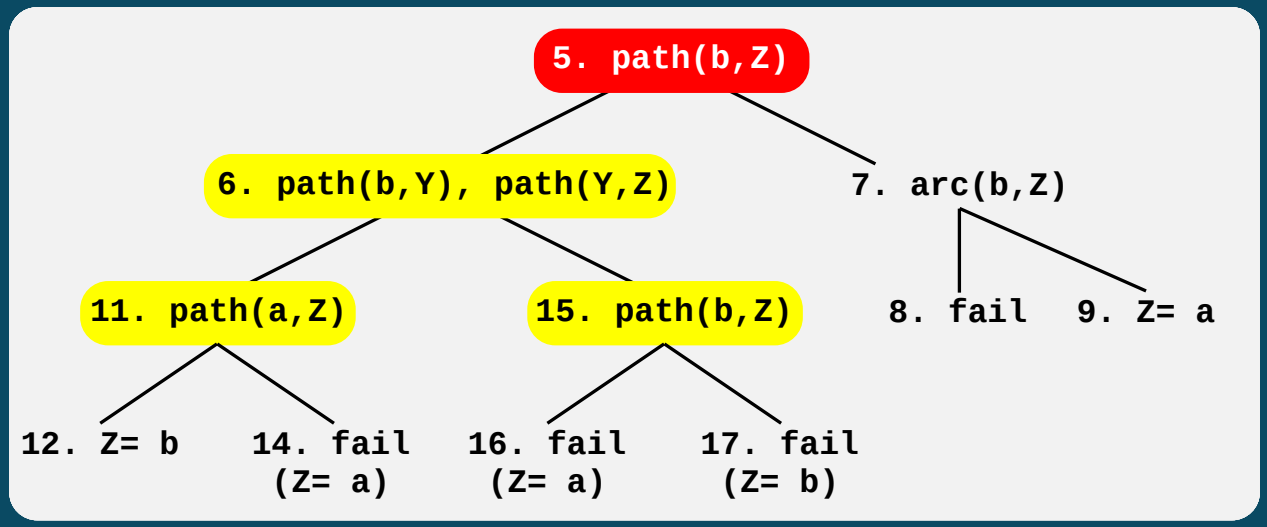


Table Space

Subgoal	Answers
0. path(a,Z)	3. Z= b 10. Z= a
5. path(b,Z)	9. Z= a 12. Z= b

Operating on the Table Space

Tabled Subgoal Call

- First call → Lookup and insert subgoal in the table.
- Variant call → Lookup subgoal in the table.

Found Tabled Answer

- New answer → Lookup and insert answer in the table.
- Repeated answer → Lookup answer and fail.

Consume Answer

- Newly unconsumed answer → Load answer and proceed.
- No unconsumed answers → Suspend execution and schedule a continuation.

Operating on the Table Space

Tabled Subgoal Call

- First call → Lookup and insert subgoal in the table.
- Variant call → Lookup subgoal in the table.

Found Tabled Answer

- New answer → Lookup and insert answer in the table.
- Repeated answer → Lookup answer and fail.

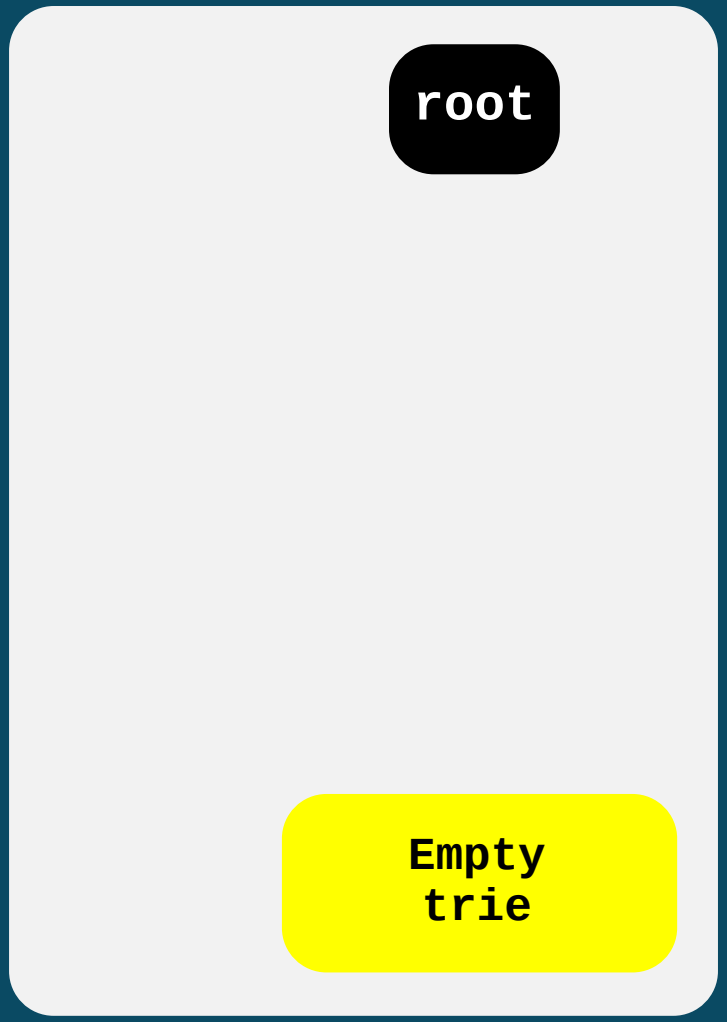
Consume Answer

- Newly unconsumed answer → Load answer and proceed.
- No unconsumed answers → Suspend execution and schedule a continuation.

To achieve an efficient implementation, we need to represent the table space with a data structure that is **compact** and allows **fast lookup and insertion** of terms.

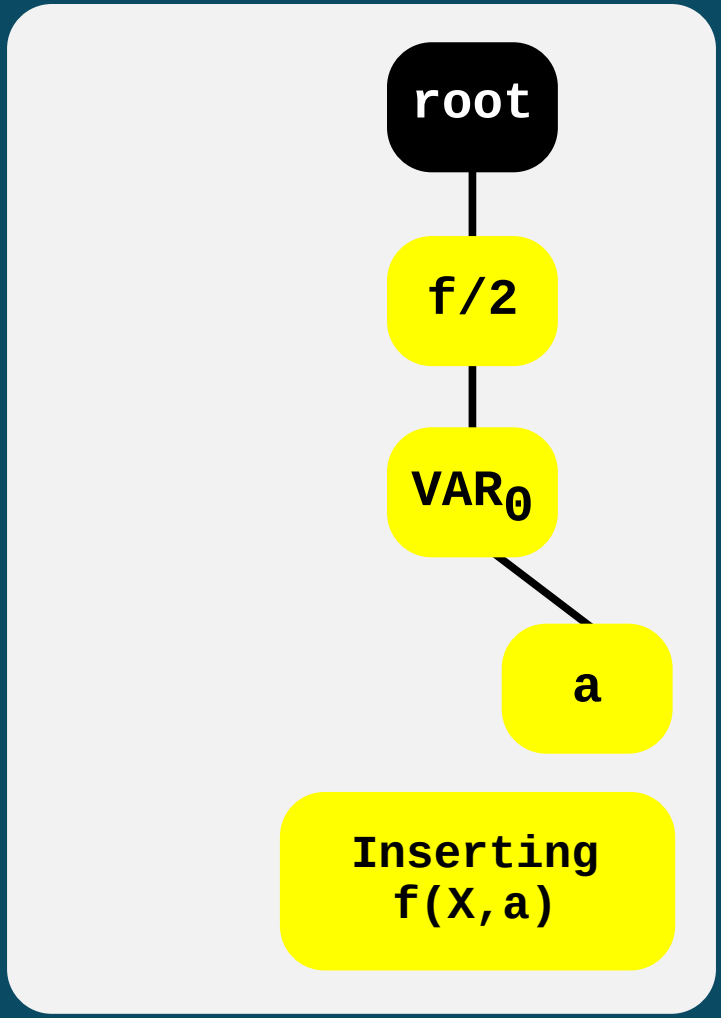
Using Tries to Represent Terms

- Tries are trees in which common prefixes are represented only once.



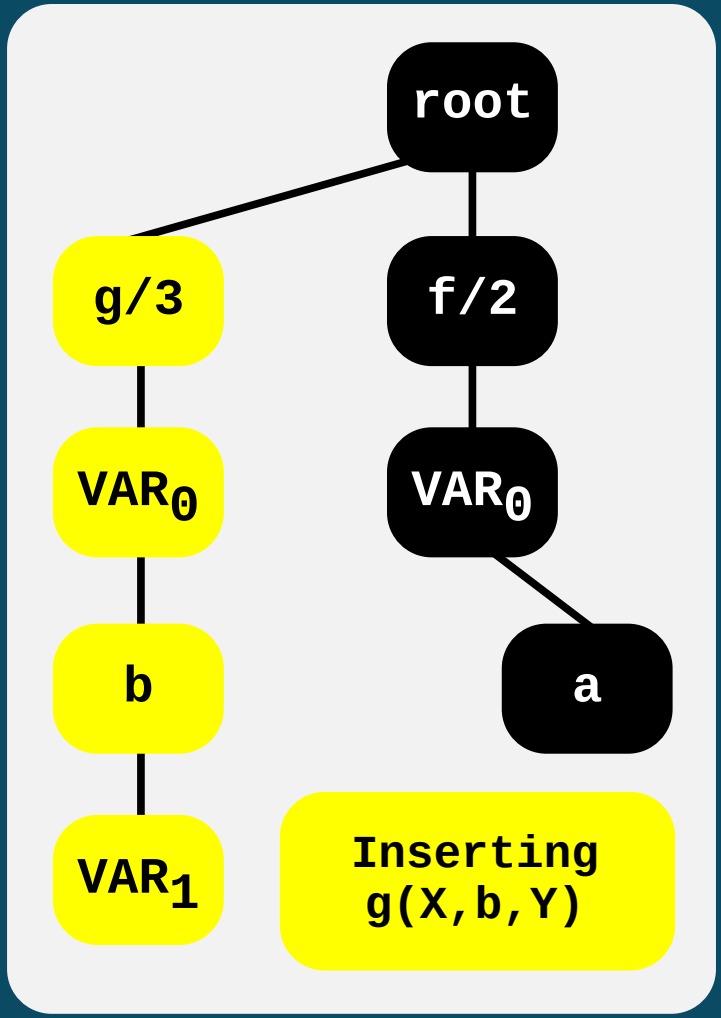
Using Tries to Represent Terms

- Tries are trees in which common prefixes are represented only once.
  - The entry point is called the root node, internal nodes represent symbols in terms and leaf nodes specify completed terms.



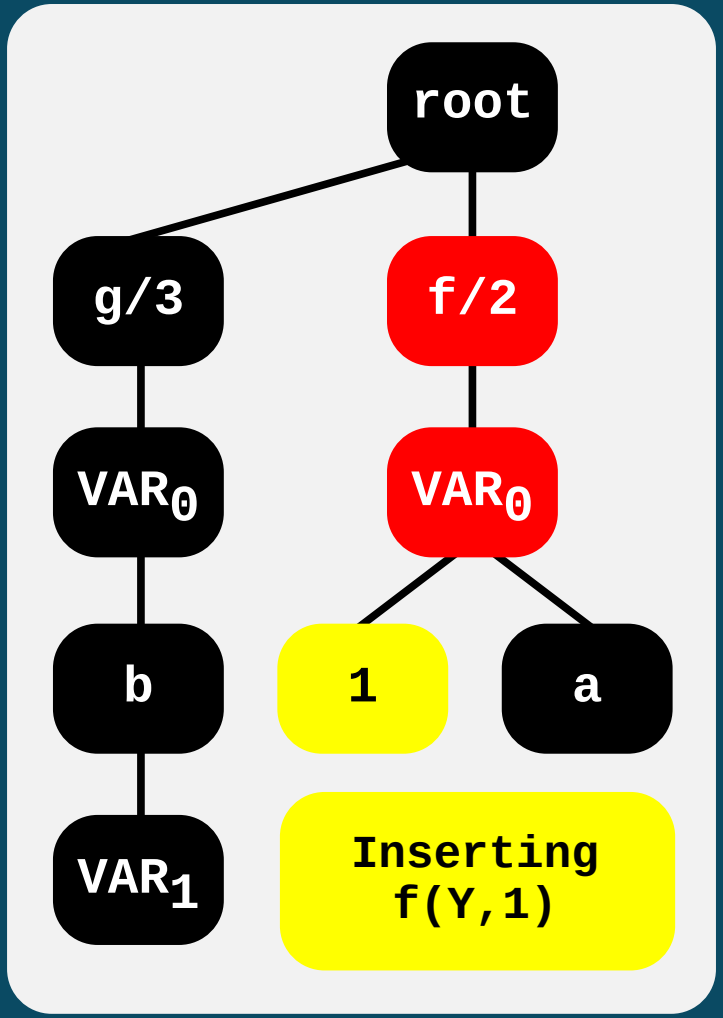
Using Tries to Represent Terms

- Tries are trees in which common prefixes are represented only once.
  - The entry point is called the root node, internal nodes represent symbols in terms and leaf nodes specify completed terms.
  - Each different path through the nodes in the trie corresponds to a term.



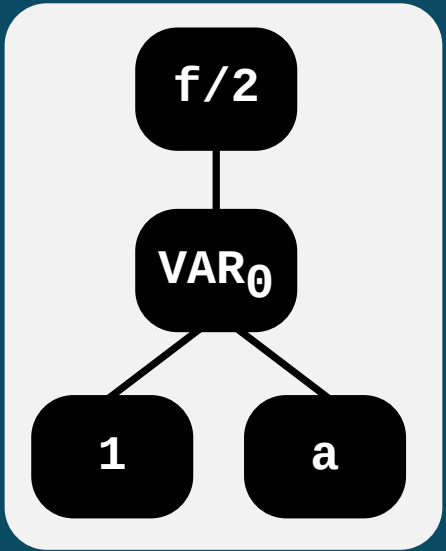
Using Tries to Represent Terms

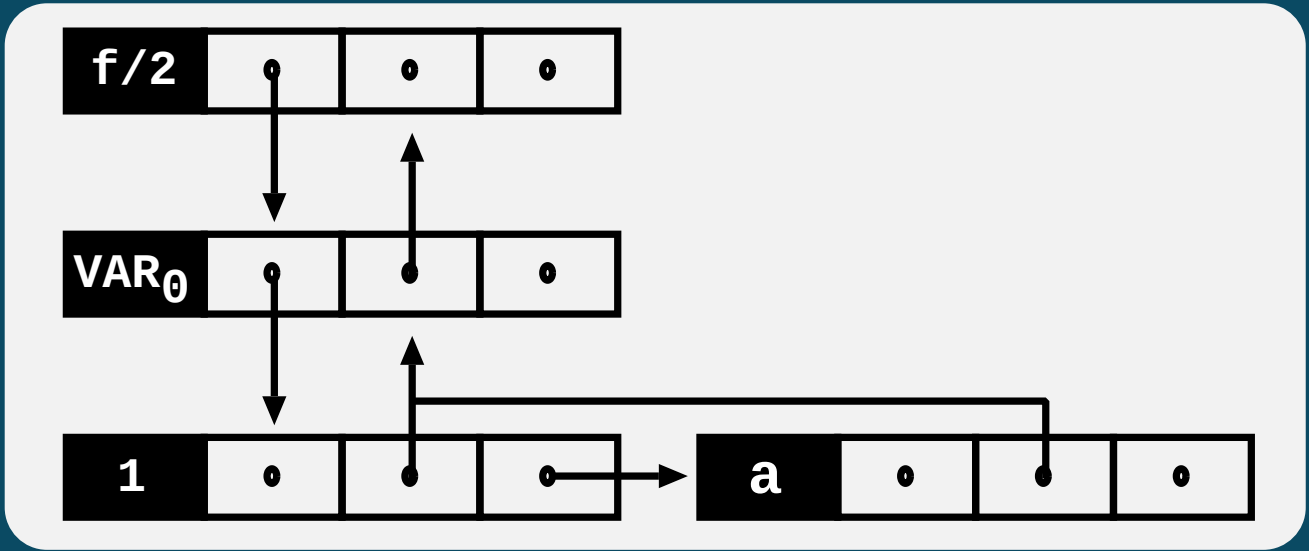
- Tries are trees in which common prefixes are represented only once.
  - The entry point is called the root node, internal nodes represent symbols in terms and leaf nodes specify completed terms.
  - Each different path through the nodes in the trie corresponds to a term.
  - Terms with common prefixes branch off from each other at the first distinguishing symbol.



Structure of a Trie Node

- A trie node has four fields:
  - TrNode_symbol: stores the symbol for the node.
  - TrNode_child: pointer to first-child node.
  - TrNode_parent: pointer to parent node.
  - TrNode_next: pointer to sibling node.

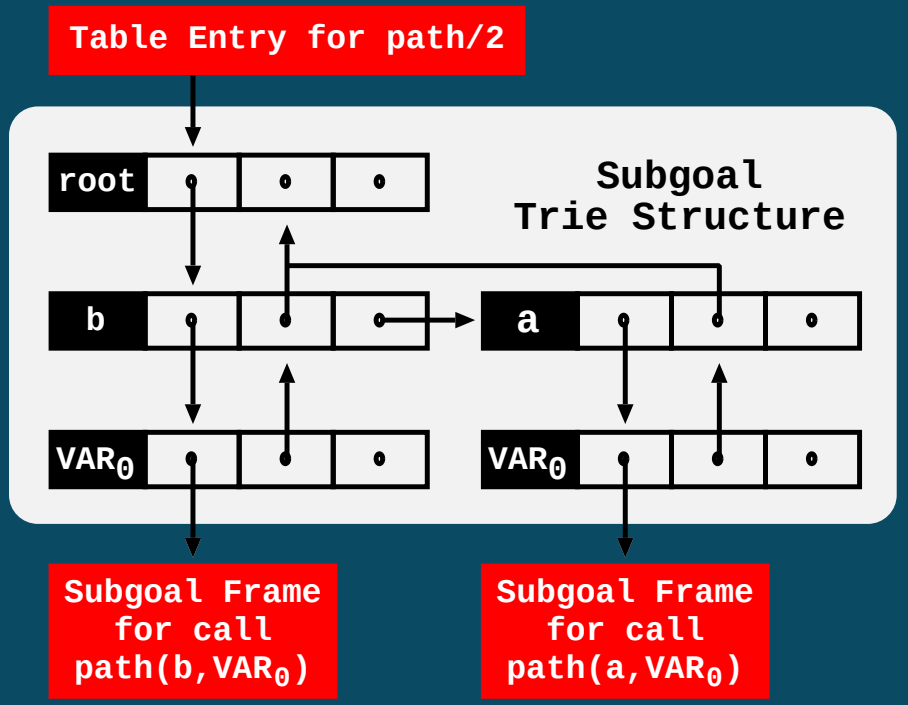




Using Tries to Organise the Table Space

Subgoal Trie Structure

- Stores the tabled subgoal calls.
- Starts at a table entry and ends with subgoal frames.
- A subgoal frame is the entry point for the subgoal answers.



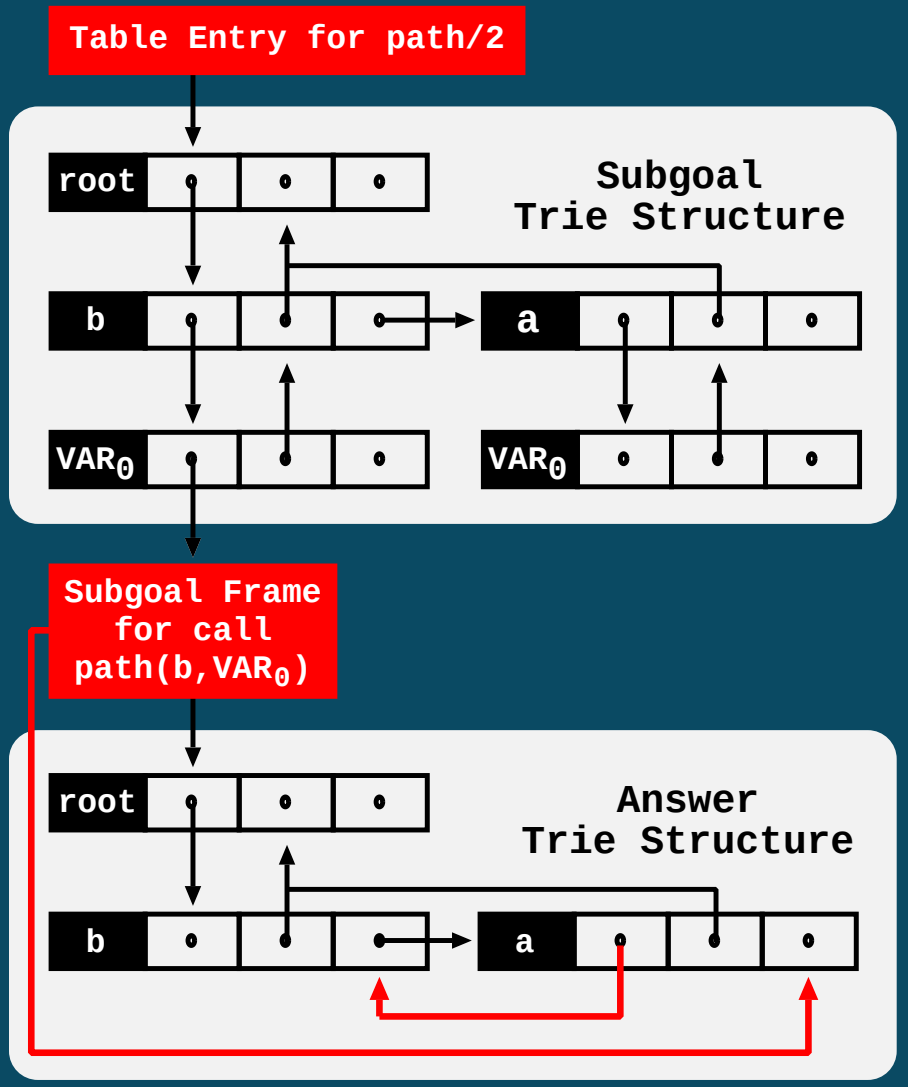
Using Tries to Organise the Table Space

Subgoal Trie Structure

- Stores the tabled subgoal calls.
- Starts at a table entry and ends with subgoal frames.
- A subgoal frame is the entry point for the subgoal answers.

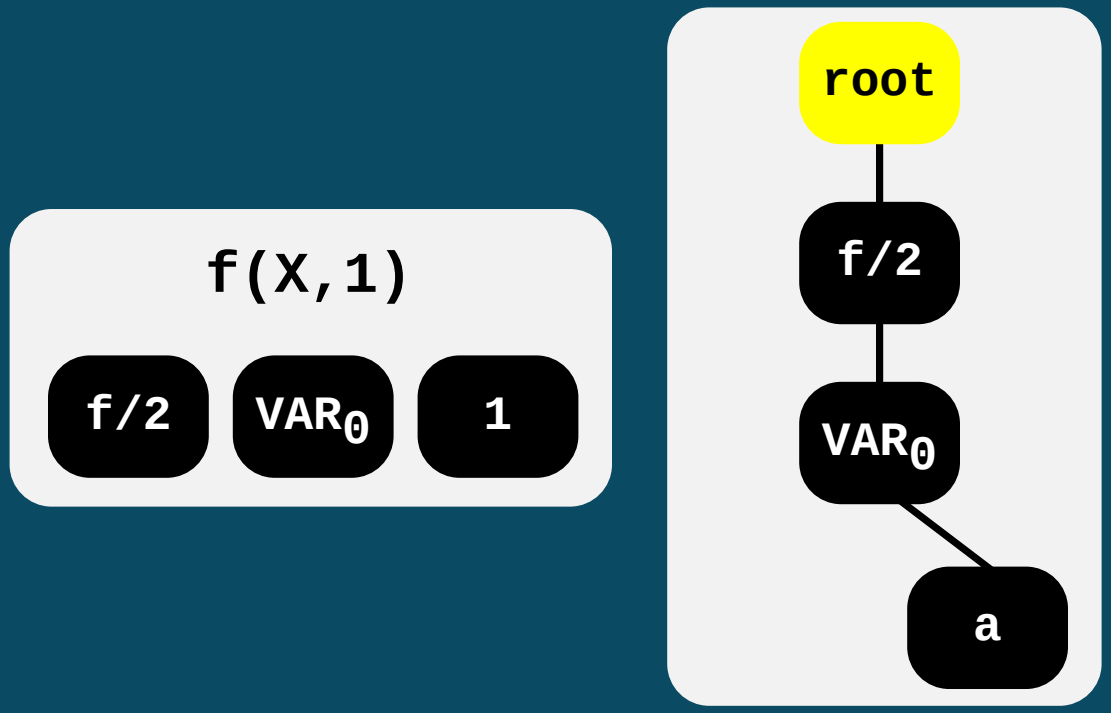
Answer Trie Structure

- Stores the tabled answers.
- Leaf nodes are chained in insertion time order.
- Variant calls keep a reference to the leaf node of the last consumed answer, hence can consume more answers by following the chain.



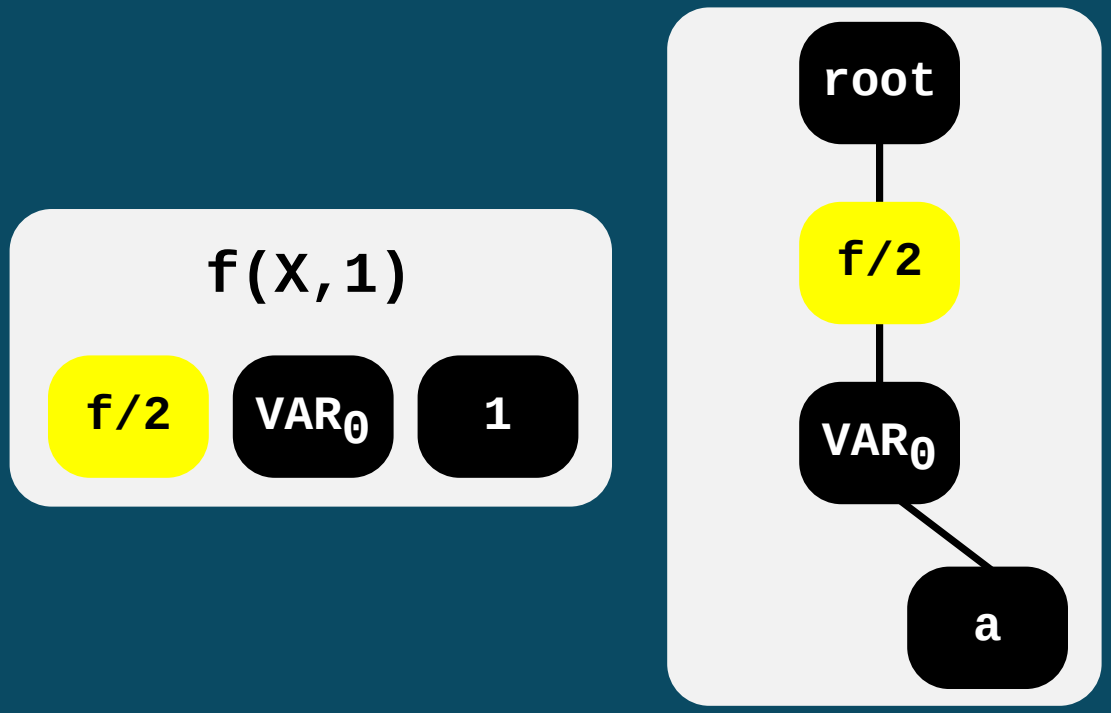
Lookup and Insertion of Terms

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



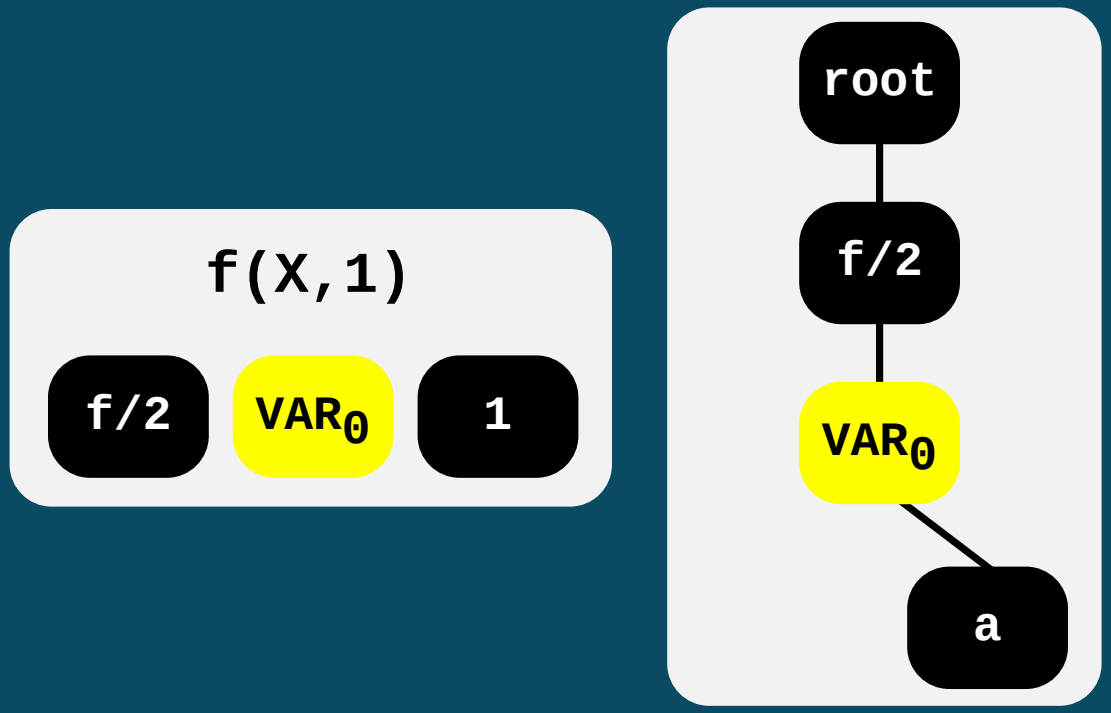
Lookup and Insertion of Terms

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



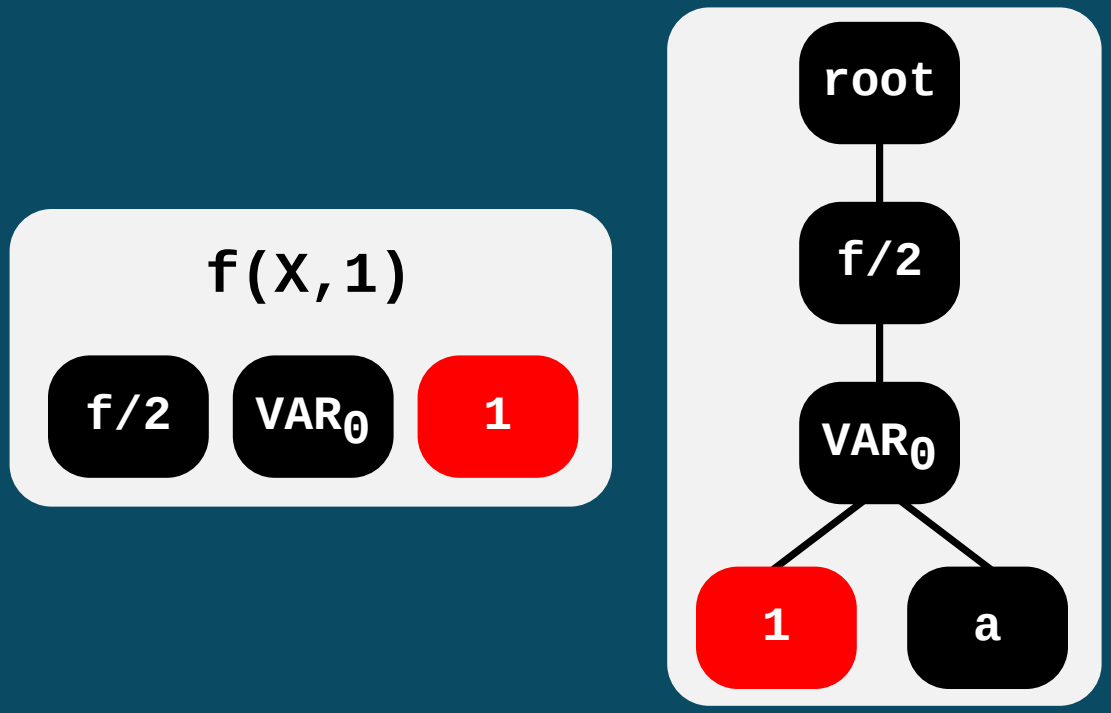
Lookup and Insertion of Terms

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



Lookup and Insertion of Terms

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

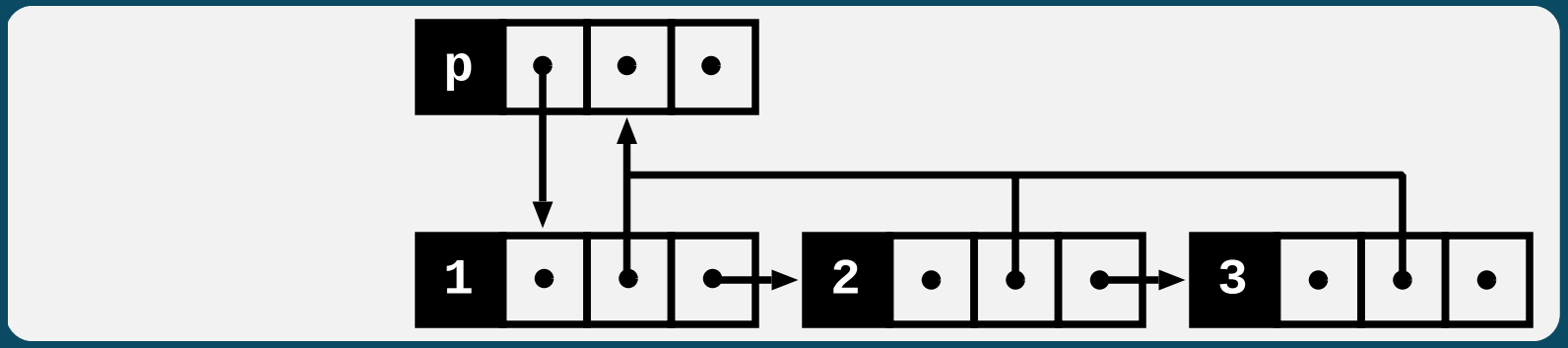


Lookup and Insertion of Terms

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

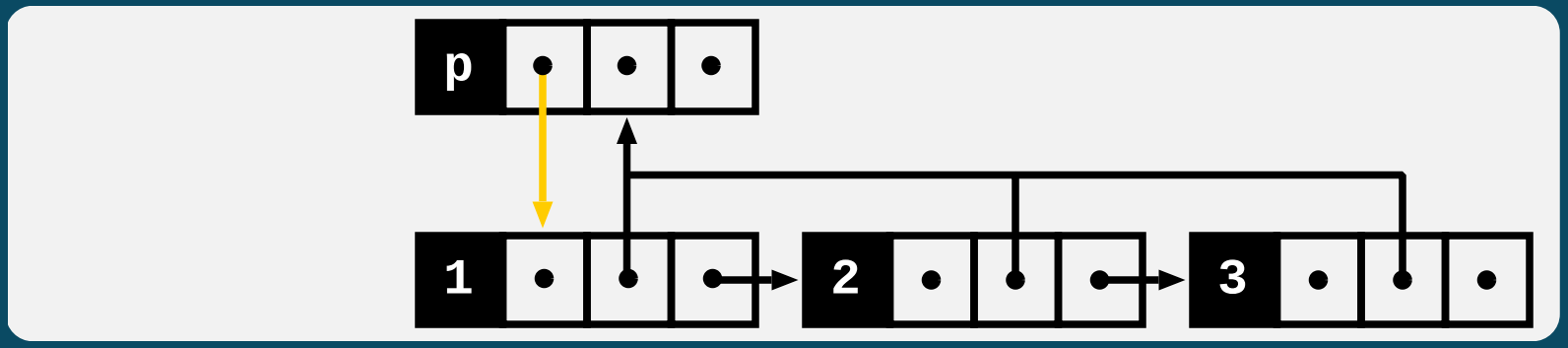
Lookup and Insertion of a Term Symbol

```
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
}
```



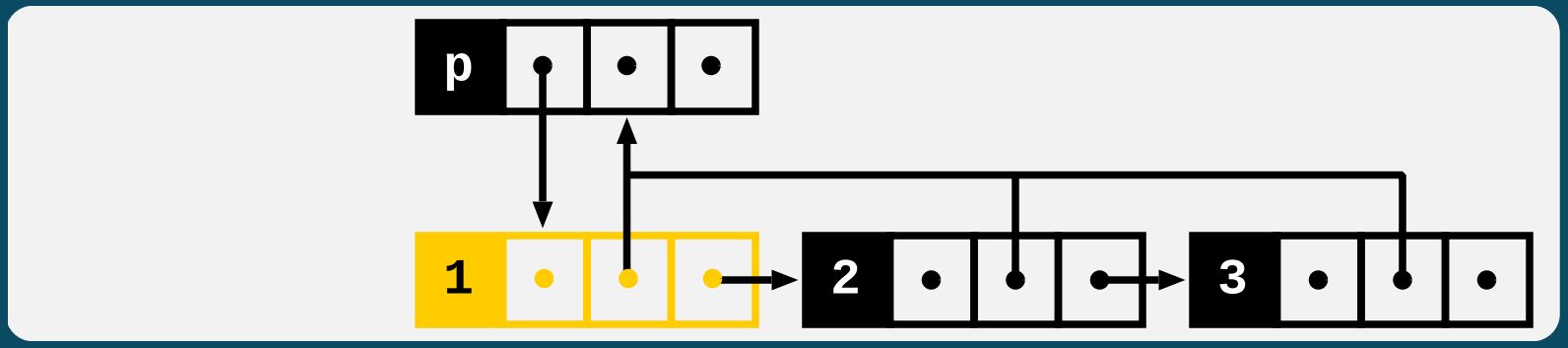
Lookup and Insertion of a Term Symbol

```
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
}
```



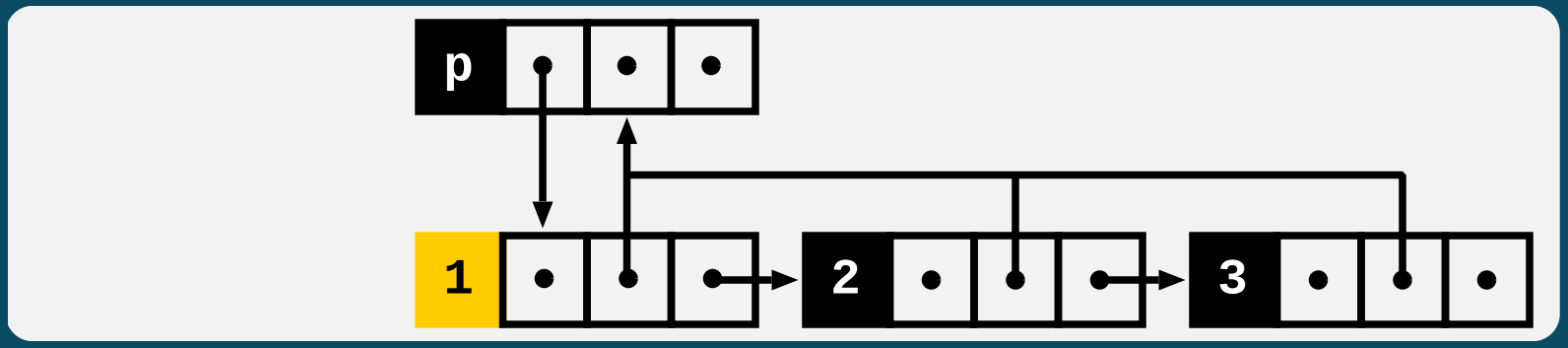
Lookup and Insertion of a Term Symbol

```
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
}
```



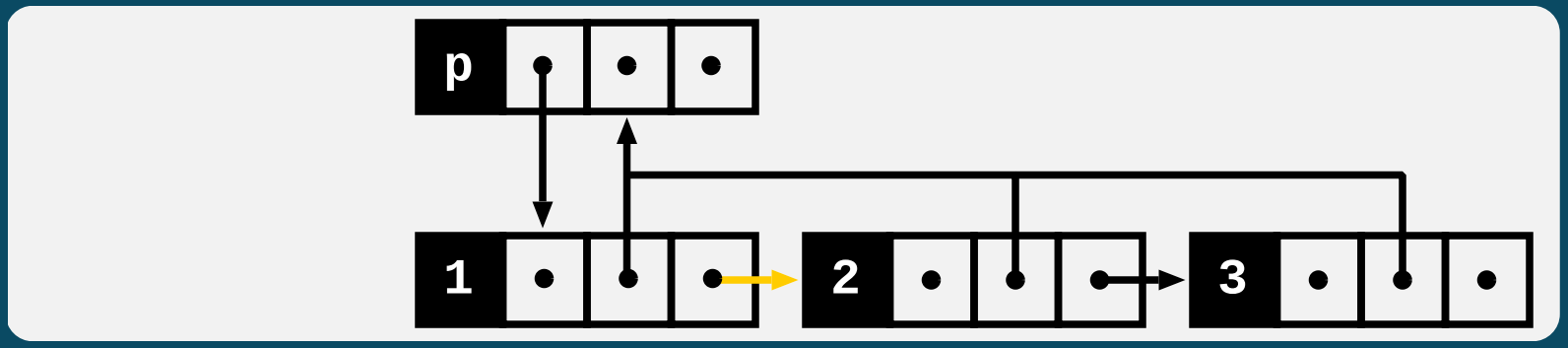
Lookup and Insertion of a Term Symbol

```
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
}
```



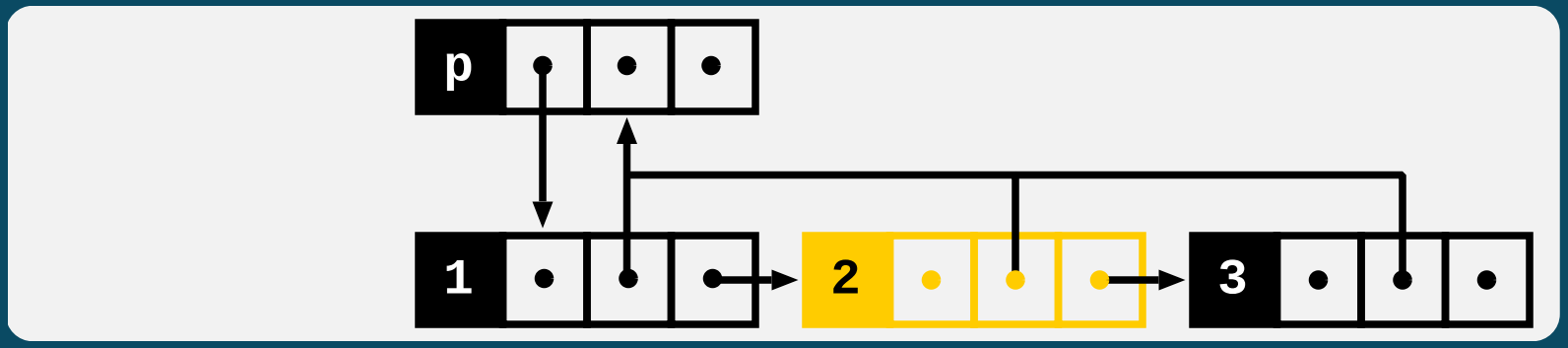
Lookup and Insertion of a Term Symbol

```
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
}
```



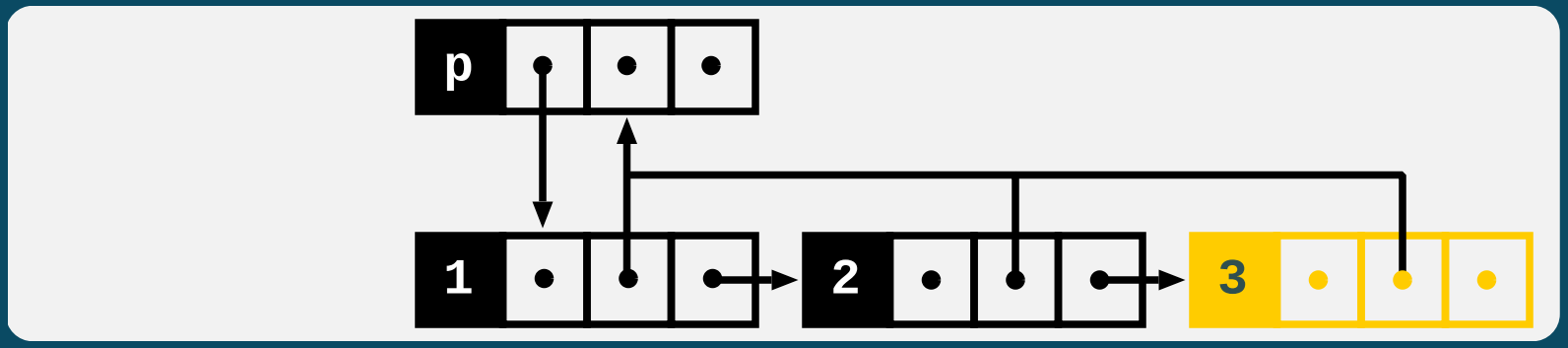
Lookup and Insertion of a Term Symbol

```
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
}
```



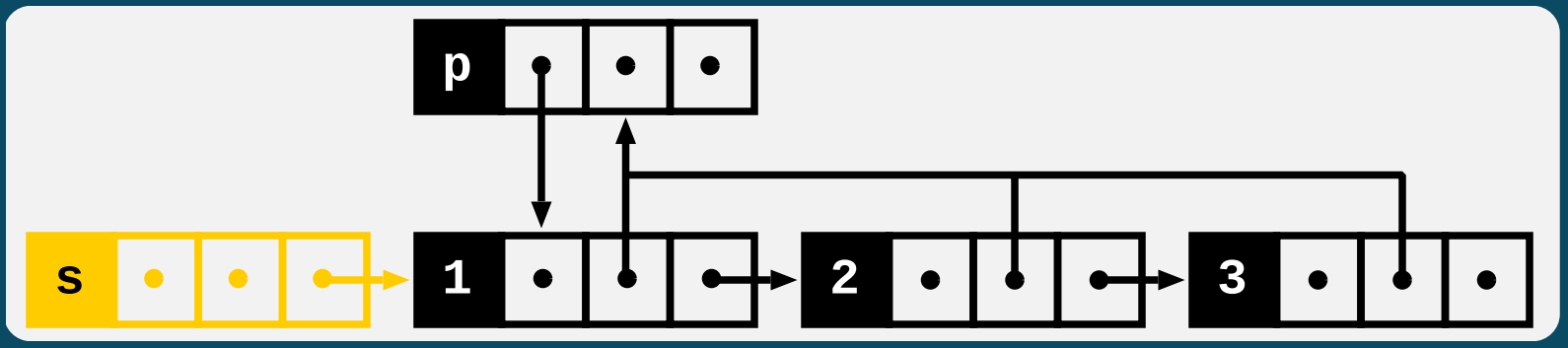
Lookup and Insertion of a Term Symbol

```
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
}
```



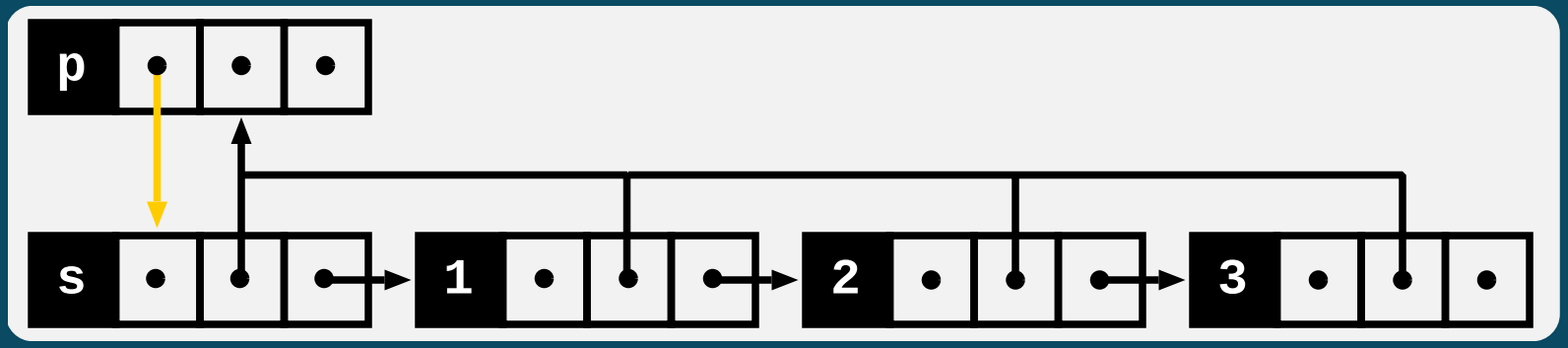
Lookup and Insertion of a Term Symbol

```
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
}
```



Lookup and Insertion of a Term Symbol

```
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
}
```



Lookup and Insertion of a Term Symbol

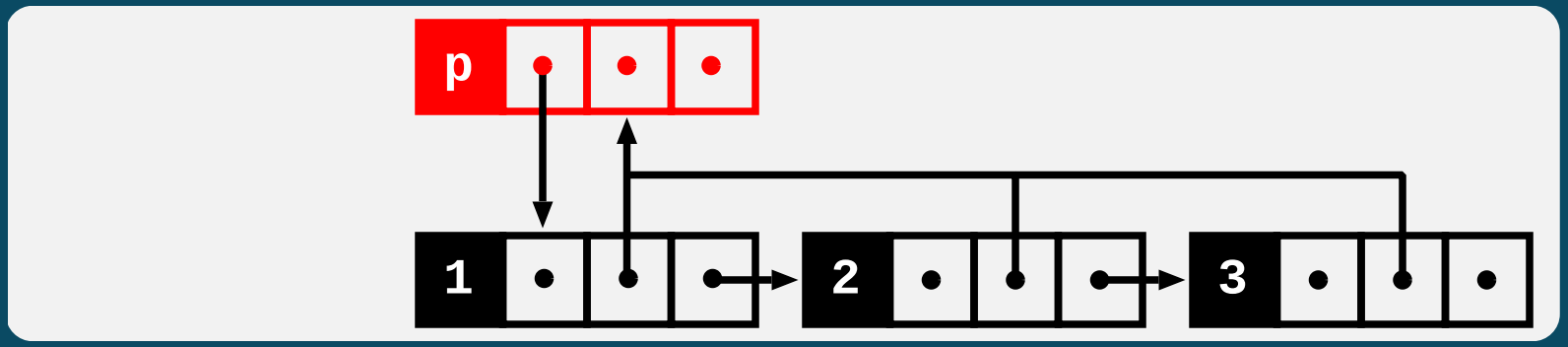
```
trie_check_insert(symbol s, trie node parent) {
  child = TrNode_child(parent)
  while (child) {
    if (TrNode_symbol(child) == s) return child
    child = TrNode_next(child)
  }
  child = new_trie_node(s, NULL, parent, TrNode_child(parent))
  TrNode_child(parent) = child
  return child
}
```

Concurrent Table Accesses

- Concurrent access to the table space requires mutual exclusion when adding new entries.
- We address concurrency by extending tries to support locking schemes.
- We have defined 3 locking schemes:
  - **TLNL**: Table Lock at Node Level
  - **TLWL**: Table Lock at Write Level
  - **TLWL-ABC**: Table Lock at Write Level-Allocate Before Check
- Two critical issues influence the efficiency of our locking schemes:
  - **Lock count** - the number of locks required to check/insert a term.
  - **Lock duration** - the amount of time a data structure is held.

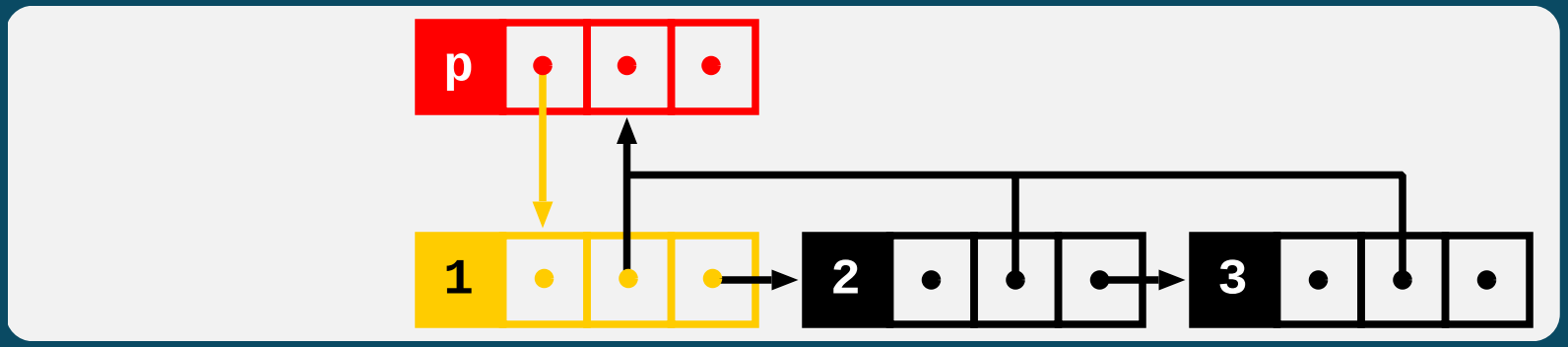
TLNL: Table Lock at Node Level

```
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
}
```



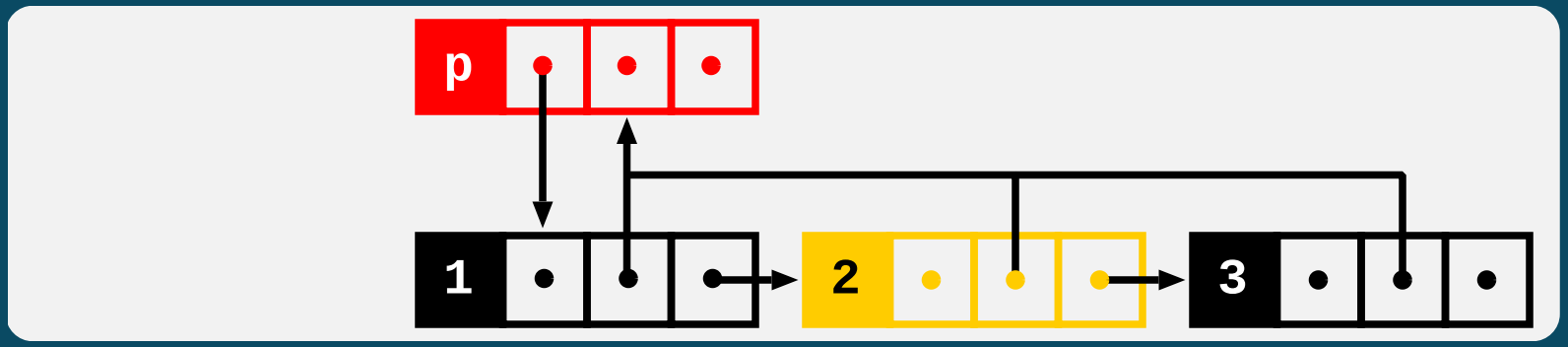
TLNL: Table Lock at Node Level

```
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
}
```



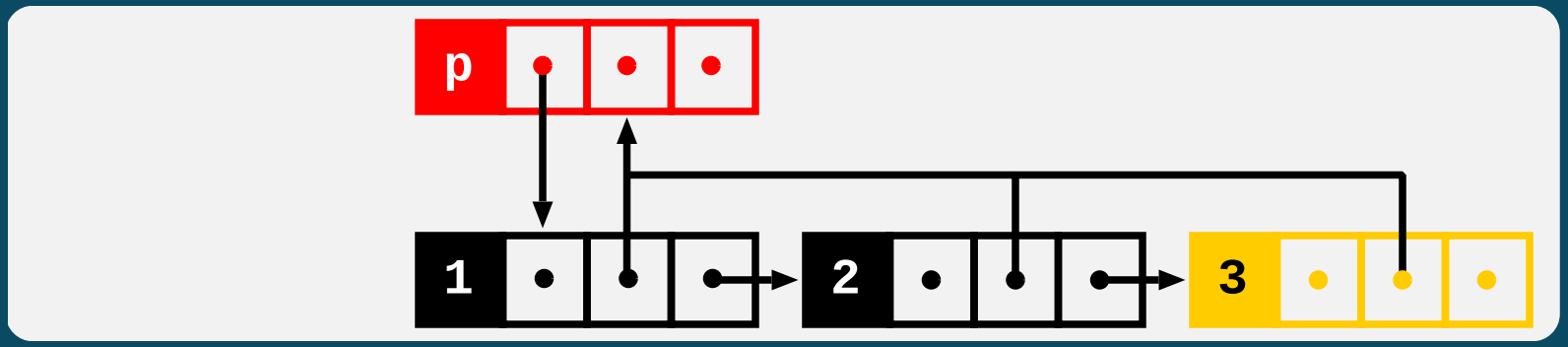
TLNL: Table Lock at Node Level

```
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
}
```



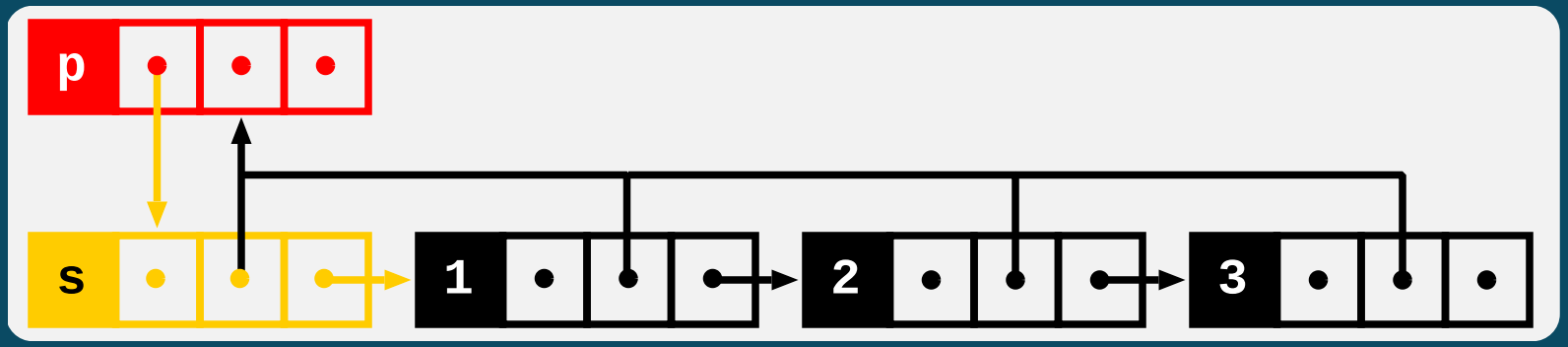
TLNL: Table Lock at Node Level

```
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
}
```



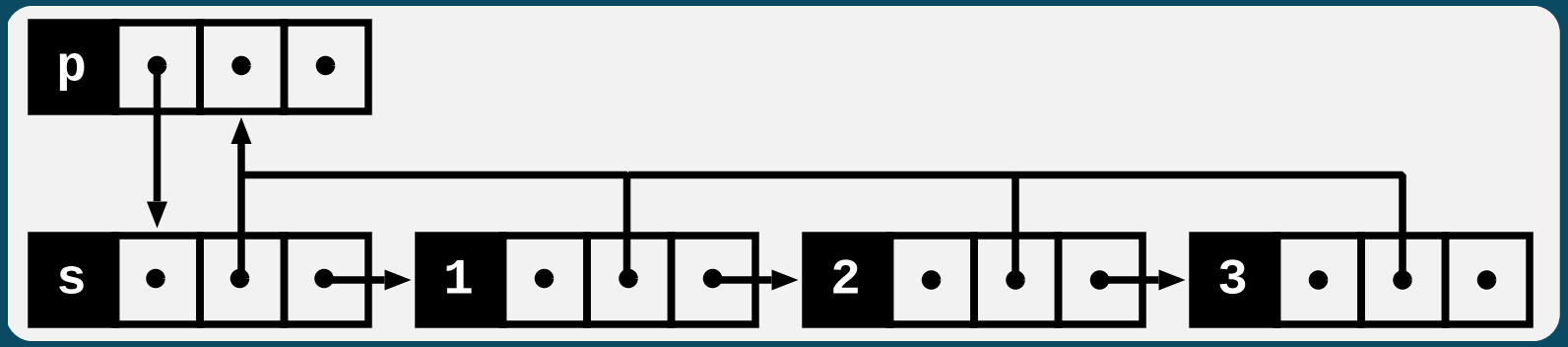
TLNL: Table Lock at Node Level

```
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

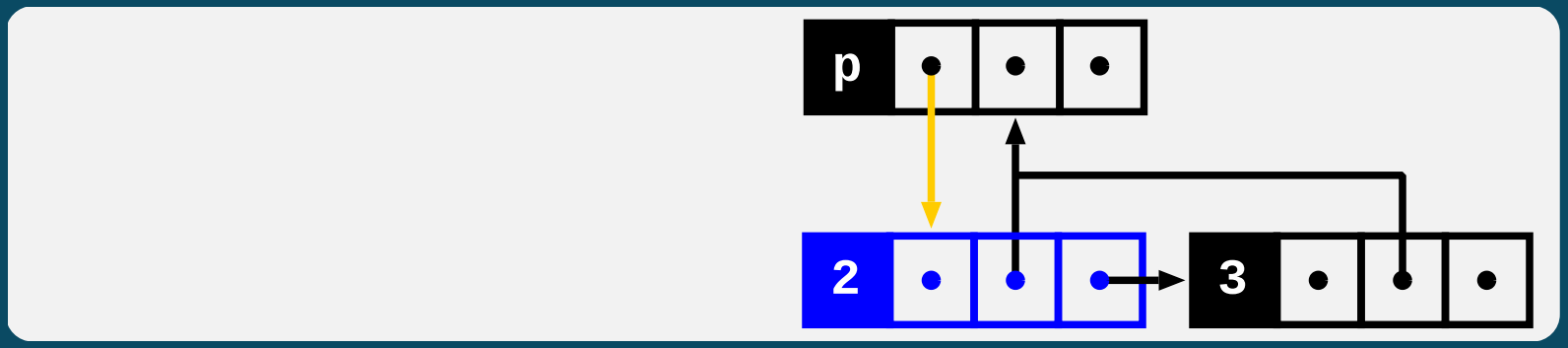
```
trie_check_insert(symbol s, trie node parent) {
   lock(parent)                          // locking the parent node
   child = TrNode_child(parent)
   while (child) {
      if (TrNode_symbol(child) == s) {
         unlock(parent)                  // unlocking before return
         return child
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

Locking Schemes Properties

- **TLNL**
  - **Lock count is proportional to the length of the term.**
  - **Lock duration is proportional to the time needed to traverse child nodes.**

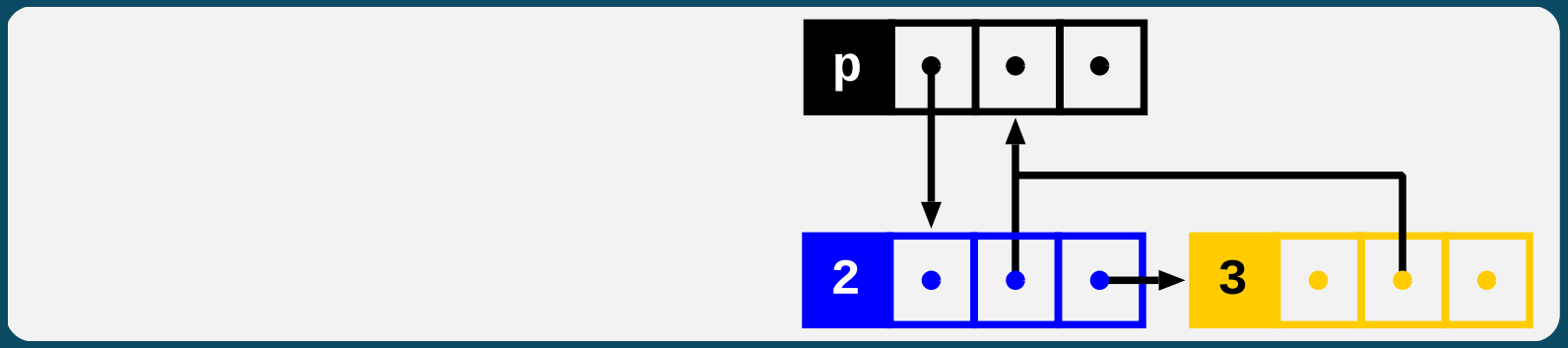
TLWL: Table Lock at Write Level

```
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
   }
   ...
}
```



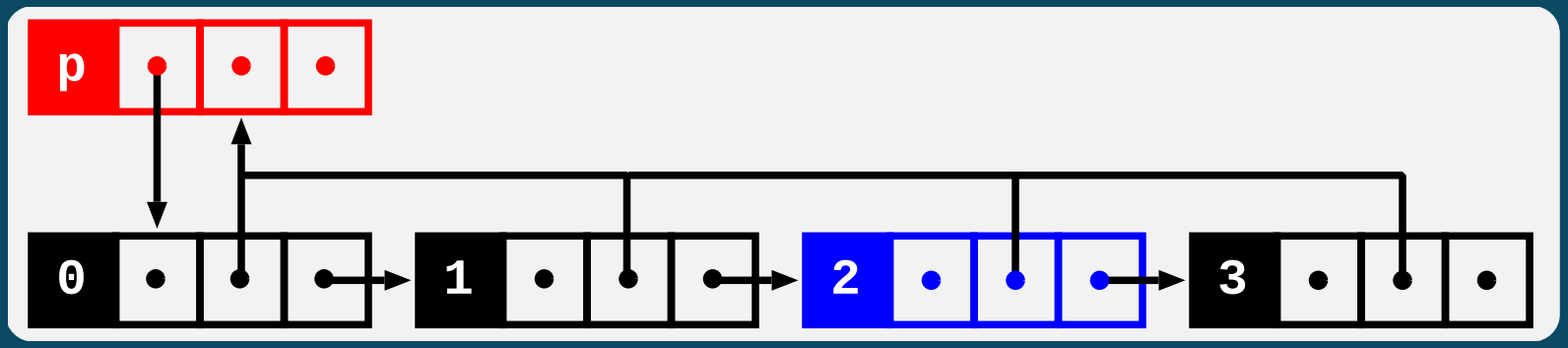
TLWL: Table Lock at Write Level

```
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
   }
   ...
}
```



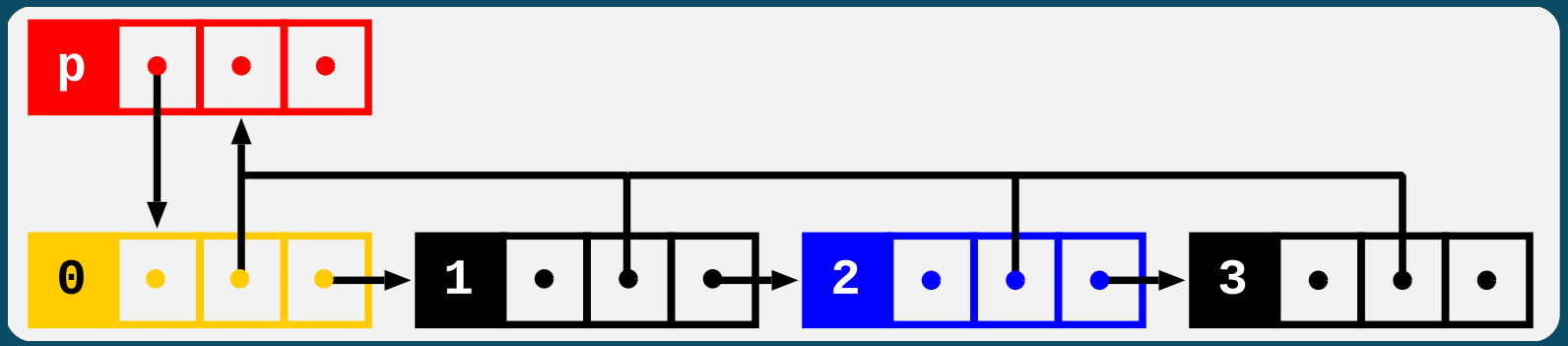
TLWL: Table Lock at Write Level

```
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
   }
   ...
}
```



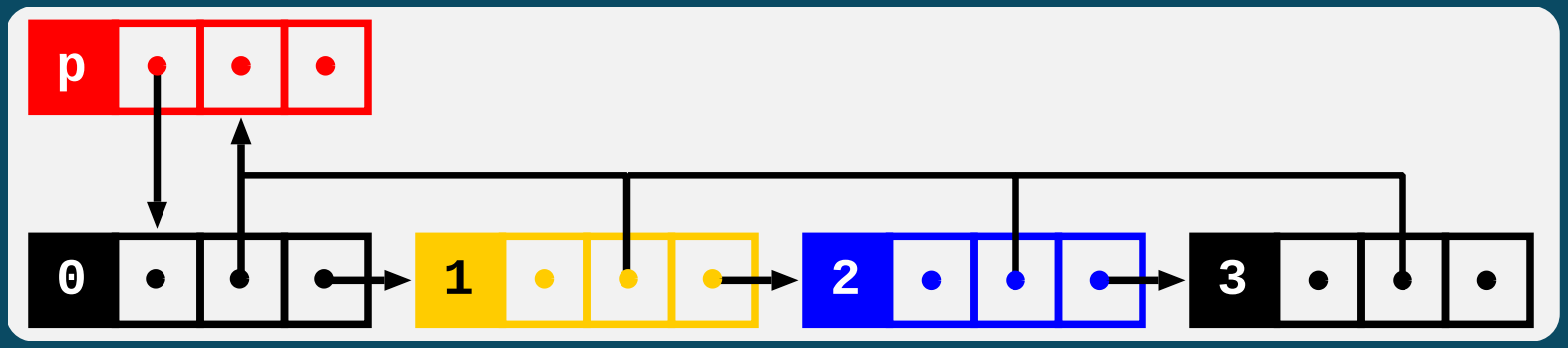
TLWL: Table Lock at Write Level

```
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
   }
   ...
}
```



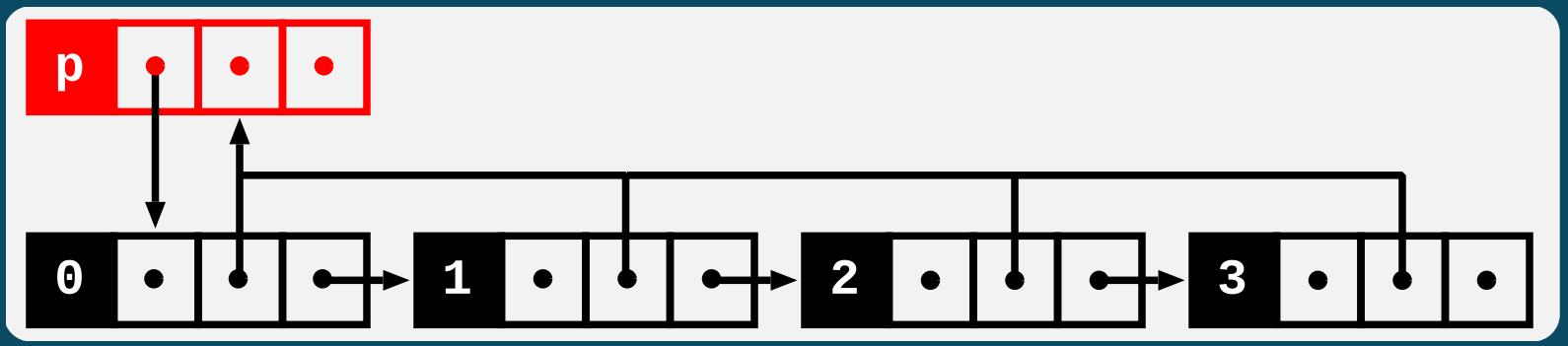
TLWL: Table Lock at Write Level

```
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

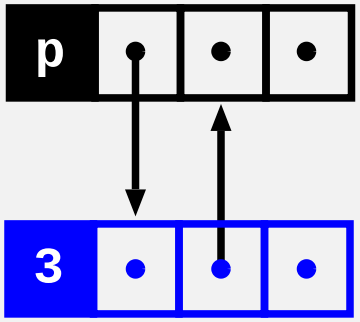
```
trie_check_insert(symbol s, trie node parent) {
   child = TrNode_child(parent)
   initial_child = child                    // keep the initial child node
   while (child) {
      if (TrNode_symbol(child) == s) return child
      child = TrNode_next(child)
   }
   lock(parent)
   child = TrNode_child(parent)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

Locking Schemes Properties

- **TLNL**
  - Lock count is proportional to the length of the term.
  - Lock duration is proportional to the time needed to traverse child nodes.
- **TLWL**
  - **Lock count varies from 0 to the length of the term.**
  - **Lock duration may be**
    - none, if the node already exists in the initial chain;
    - proportional to the child nodes added in the meantime plus the time to allocate the node.

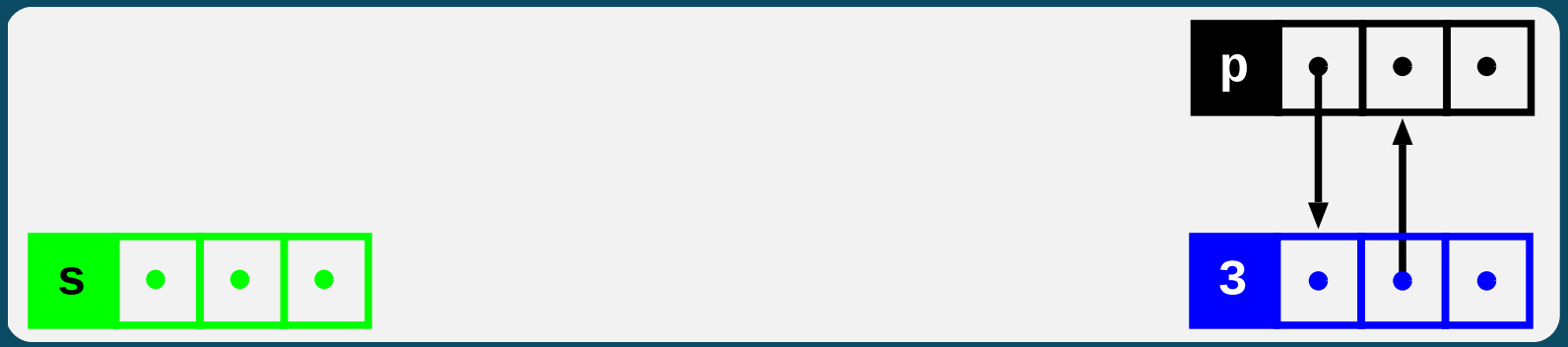
TLWL-ABC: Table Lock at Write Level-Allocate Before Check

```
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
}
```



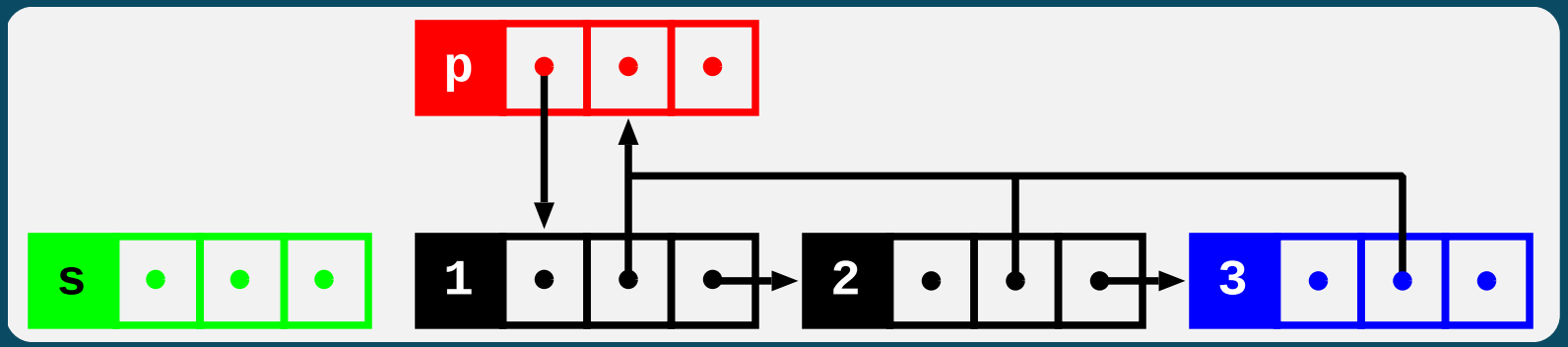
TLWL-ABC: Table Lock at Write Level-Allocate Before Check

```
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
}
```



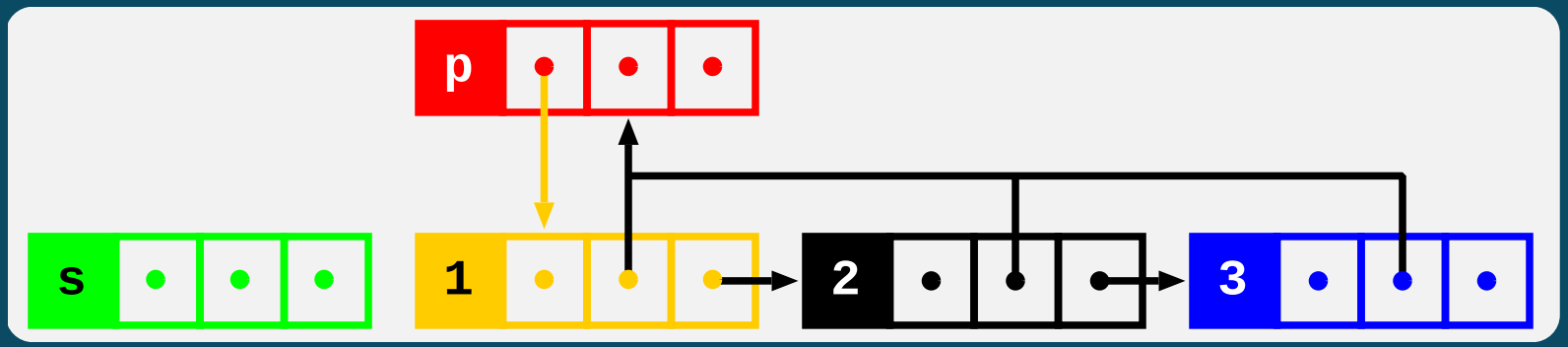
TLWL-ABC: Table Lock at Write Level-Allocate Before Check

```
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
}
```



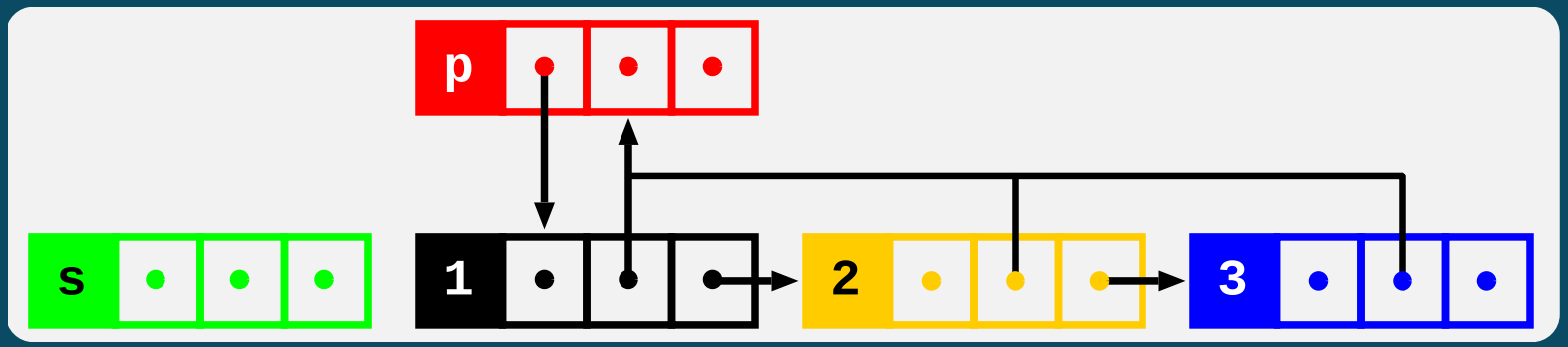
TLWL-ABC: Table Lock at Write Level-Allocate Before Check

```
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
}
```



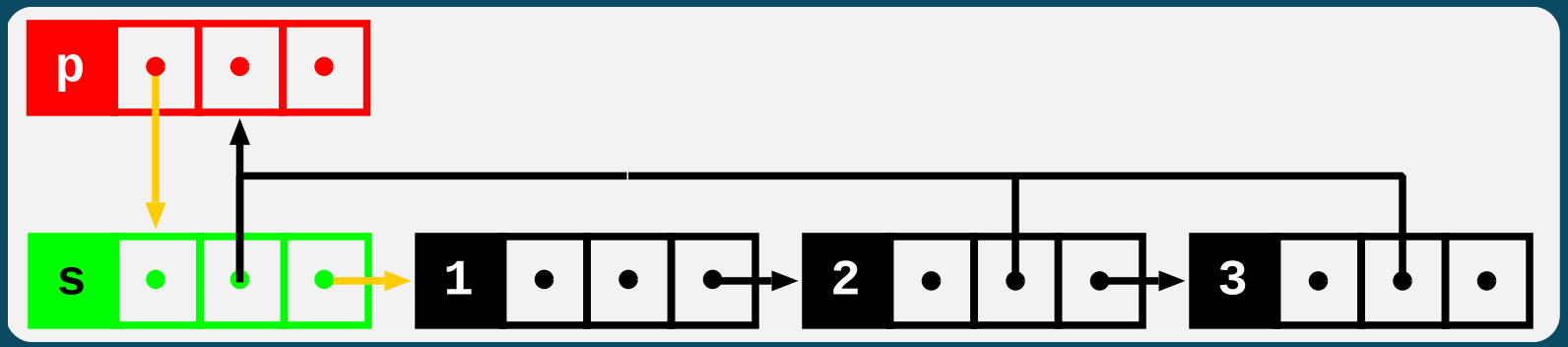
TLWL-ABC: Table Lock at Write Level-Allocate Before Check

```
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
}
```



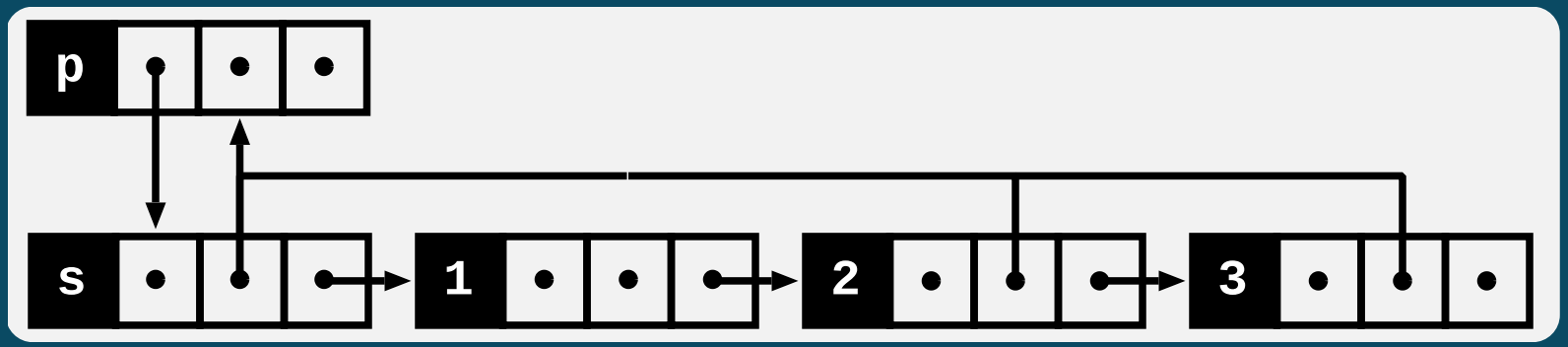
TLWL-ABC: Table Lock at Write Level-Allocate Before Check

```
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
}
```



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
}
```



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
}
```

Locking Schemes Properties

- **TLNL**
  - Lock count is proportional to the length of the term.
  - Lock duration is proportional to the time needed to traverse child nodes.
- **TLWL**
  - Lock count varies from 0 to the length of the term.
  - Lock duration may be
    - none, if the node already exists in the initial chain;
    - proportional to the child nodes added in the meantime plus the time to allocate the node.
- **TLWL-ABC**
  - **Lock count varies from 0 to the length of the term.**
  - **Lock duration may be**
    - **none, if the node already exists in the initial chain;**
    - **proportional to the child nodes added in the meantime.**

Preliminary Results

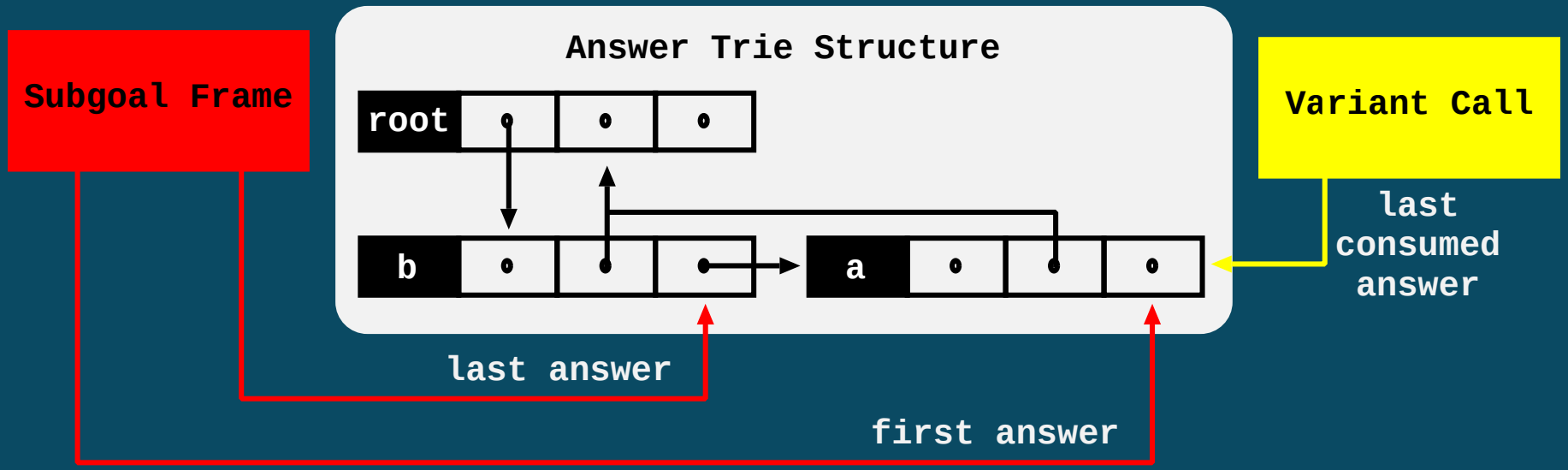
Schemes	Workers 8	16	24	32
mc-sieve				
TLNL	7.2	11.8	3.9	4.7
TLWL	**7.9**	**15.8**	**23.7**	**31.5**
TLWL-ABC	**7.9**	**15.8**	**23.7**	31.4
mc-iproto				
TLNL	2.6	1.8	1.0	1.0
TLWL	5.0	**9.0**	**8.8**	**7.2**
TLWL-ABC	**5.1**	7.7	8.4	7.1
samegen				
TLNL	**7.2**	13.8	19.6	24.0
TLWL	**7.2**	**13.9**	**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

Speedups

Programs	Answers Unique	Repeated	Time
mc-sieve	380	1386181	268
mc-iproto	134361	385423	24
samegen	23152	65597	26
lgrid	160000	449520	69

- TLWL and TLWL-ABC are the only schemes showing scalability.
- The more refined strategy of TLWL-ABC does not show to perform better than TLWL.
- In general, TLNL slows down for more than 16 workers. It pays the cost of performing locking even when writing is not likely.

Other Synchronization Points



- Locking is also required for:
  - **Subgoal Frames:** when inserting new answers in the table space.
  - **Variant Calls:** when synchronizing access to check for available answers.

Preliminary Results

Contention Points	Workers			
	8	16	24	32
mc-sieve	**33.9s**	**17.0s**	**11.3s**	**8.5s**
trie nodes	188	415	677	1979
subgoal frames	0	0	0	2
variant calls	0	1	0	4
mc-iproto	**4.8s**	**2.7s**	**2.7s**	**3.3s**
trie nodes	6579	10537	11816	11736
subgoal frames	9894	21271	**33162**	**33307**
variant calls	4685	25006	**66334**	**81515**
samegen	**3.6s**	**1.9s**	**1.3s**	**1.1s**
trie nodes	119	201	364	417
subgoal frames	52	112	283	493
variant calls	0	1	0	0
lgrid	**9.7s**	**5.1s**	**3.5s**	**2.8s**
trie nodes	5292	10341	12870	12925
subgoal frames	1124	7319	17440	27834
variant calls	1209	5987	23357	**35991**

Contention points with TLWL

- For **mc-sieve** and **samegen** locking is not a problem.
- **mc-iproto** and **lgrid** show high contention ratios per time unit.
- These are the programs that find more answers. The sequential order by which leaf answer nodes are chained in the trie seems to be a major problem when frequently accessing the table.

Conclusions

- We studied the impact of using alternative locking schemes to deal with concurrent table accesses in parallel tabling.
- We observed that there are locking schemes that can obtain good speedup ratios and achieve scalability.
- Our results show that a main problem is not only how we do locking, but also how we use auxiliary data structures to synchronize access to the table. A key issue is the sequential order by which leaf answer nodes are chained in the trie.
- We plan to investigate whether alternative designs can obtain scalable speedups even when frequently updating/accessing tables.