CoLoSL

Compositional Reasoning At Last!

Azalea Raad Jules Villard Philippa Gardner

Imperial College London

31 March 2015

Global Shared Resources

$$P_1 \wedge P_2$$

$$\frac{\{P1\}\ C1\ \{Q1\} \qquad \{P2\}\ C2\ \{Q2\}}{\{P1 \wedge P2\}\ C1 \parallel C2\ \{Q1 \wedge Q2\}}$$

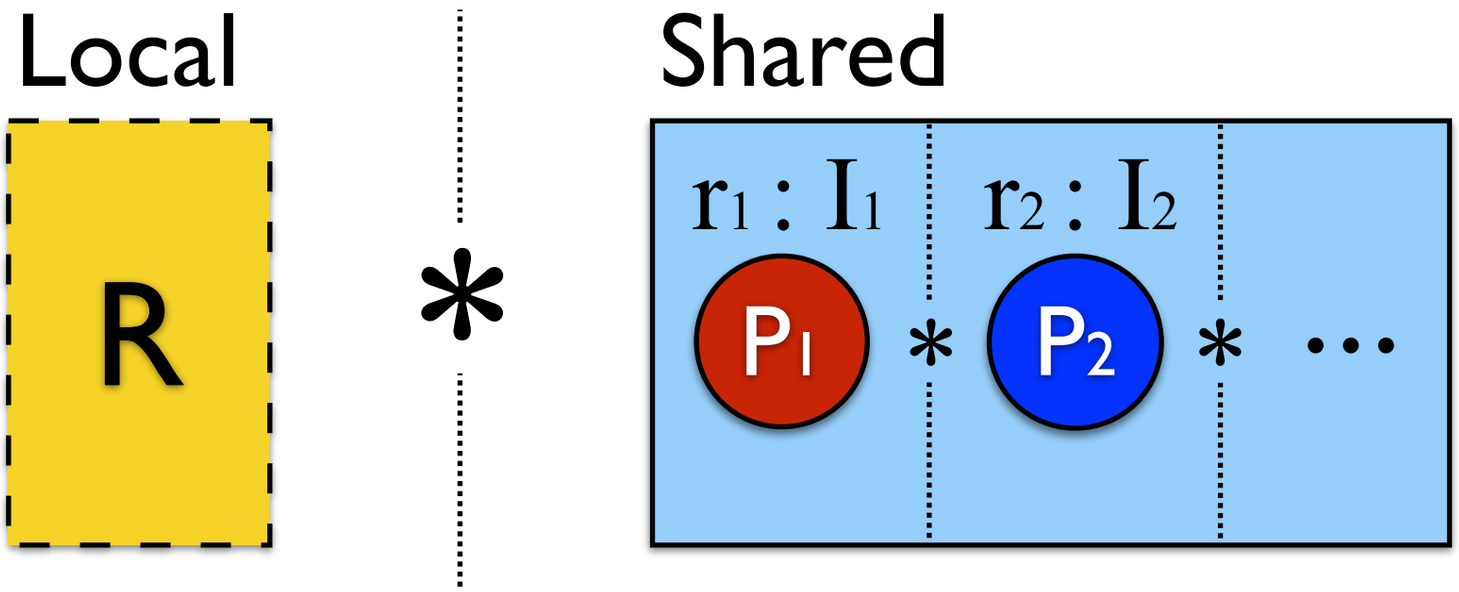
Owicki-Gries, Rely-Guarantee

Global Shared Resources

$$\frac{\{\boxed{P1}\}\ C1\ \{\boxed{Q1}\} \qquad \{\boxed{P2}\}\ C2\ \{\boxed{Q2}\}}{\{\boxed{P1 \wedge P2}\}\ C1 \parallel C2\ \{\boxed{Q1 \wedge Q2}\}}$$

- Shared Resources
  - **No framing** : reasoning on GLOBAL resources
- Interference
  - **No framing** : interference on ALL resources considered
- Extension
  - **No extension** : cannot dynamically share resources/extend interference

Disjoint Shared Resources



CSL, CAP, HOCAP, iCAP, TaDA

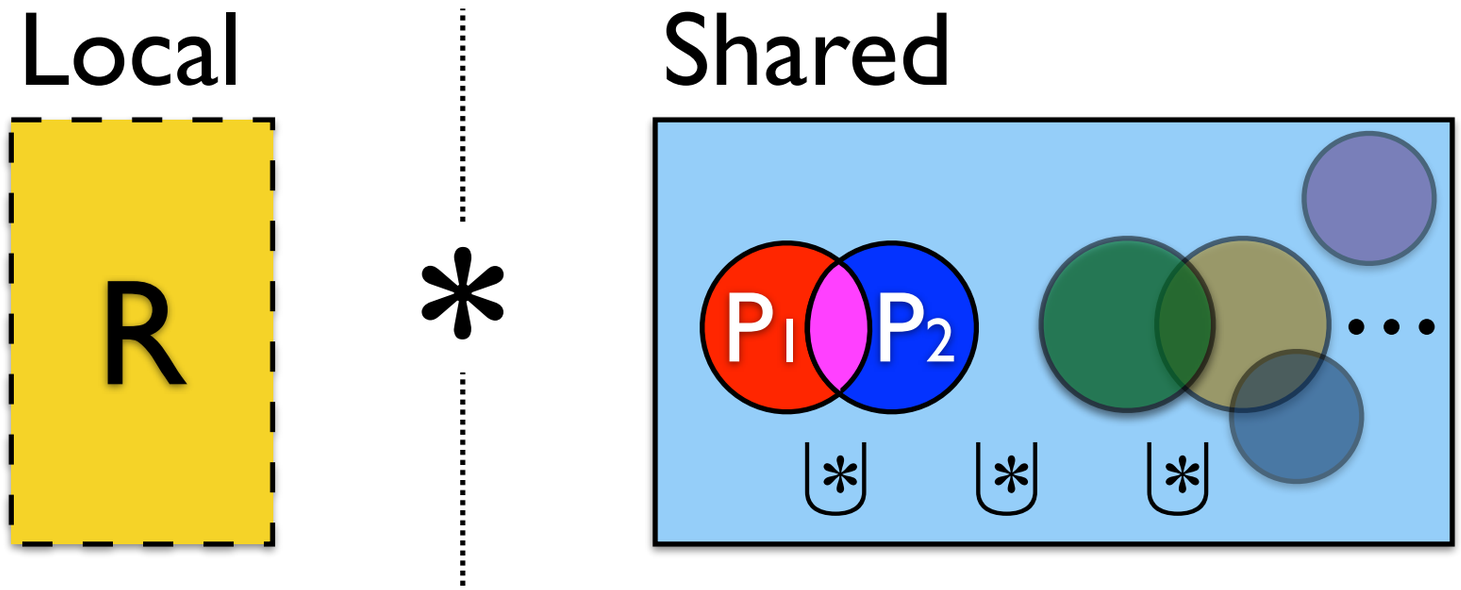
Disjoint Shared Resources

$$\frac{\{\boxed{P1}\}\ C1\ \{\boxed{Q1}\} \qquad \{\boxed{P2}\}\ C2\ \{\boxed{Q2}\}}{\{\boxed{P1} * \boxed{P2}\}\ C1 \parallel C2\ \{\boxed{Q1} * \boxed{Q2}\}}$$

- Shared resources / Interference
  - **Limited framing**:
    - **Static** (pre-determined) frames (regions/ invariants)
    - **Physically Disjoint** frames

- Extension
  - **Limited extension:**
    - Can create new regions/ invariants
    - **Cannot** extend regions with more resources/invariants

CoLoSL: Concurrent Local Subjective Logic



CoLoSL

CoLoSL: **Co**ncurrent **Lo**cal **S**ubjective **L**ogic

- Shared resources
  - **Dynamic** framing
  - **Overlapping** frames
- Interference
  - **Dynamic** framing/rewriting of interference
- Extension
  - **Dynamic** extension of shared state with new resource/interference

Why CoLoSL?

- **Local** Proofs
  - Proof reuse - proofs done for the largest possible context

Dijkstra's Token Ring

$C =$

```
while (x0 < 10){
  if (x0 == xsize)
    < x0 ++ >
}
```

$\|$

```
while (x1 < 10) {
  if (x1 < x0 )
    < x1 ++ >
}
```

$\| \cdots \|$

```
while (xn < 10) {
  if (xn < xn-1 )
    < xn ++ >
}
```

size	M: x_0	S_1: x_1	S_2: x_2	…	S_{n-1}: x_{n-1}	S_n: x_n
n	0	0	0		0	0

Dijkstra's Token Ring

$C =$

```
while (x0 < 10){
  if (x0 == xsize)
    < x0 ++ >
}
```

$\|$

```
while (x1 < 10) {
  if (x1 < x0 )
    < x1 ++ >
}
```

$\| \cdots \|$

```
while (xn < 10) {
  if (xn < xn-1 )
    < xn ++ >
}
```

size	M: x_0	S_1: x_1	S_2: x_2	…	S_{n-1}: x_{n-1}	S_n: x_n
n	1	0	0		0	0

Dijkstra's Token Ring

C =

```
while (x0 < 10){
  if (x0 == xsize)
    < x0 ++ >
}
```

||

```
while (x1 < 10) {
  if (x1 < x0 )
    < x1 ++ >
}
```

|| ... ||

```
while (xn < 10) {
  if (xn < xn-1 )
    < xn ++ >
}
```

size	M: x_0	S_1: x_1	S_2: x_2	…	S_{n-1}: x_{n-1}	S_n: x_n
n	I	I	0		0	0

II

Dijkstra's Token Ring

$C =$

```
while (x0 < 10){
  if (x0 == xsize)
    < x0 ++ >
}
```

$\|$

```
while (x1 < 10) {
  if (x1 < x0 )
    < x1 ++ >
}
```

$\| \cdots \|$

```
while (xn < 10) {
  if (xn < xn-1 )
    < xn ++ >
}
```

size	M: x_0	S_1: x_1	S_2: x_2	…	S_{n-1}: x_{n-1}	S_n: x_n
n	I	I	I		I	I

Dijkstra's Token Ring

$C =$

```
while (x0 < 10){
  if (x0 == xsize)
    < x0 ++ >
}
```

$\|$

```
while (x1 < 10) {
  if (x1 < x0 )
    < x1 ++ >
}
```

$\| \cdots \|$

```
while (xn < 10) {
  if (xn < xn-1 )
    < xn ++ >
}
```

size	M: x_0	S_1: x_1	S_2: x_2	…	S_{n-1}: x_{n-1}	S_n: x_n
n	10	10	10		10	10

Dijkstra's Token Ring : CoLoSL

$C =$

```
while (x0 < 10){
  if (x0 == xsize)
    < x0 ++ >
}
```

$\|$

```
while (x1 < 10) {
  if (x1 < x0 )
    < x1 ++ >
}
```

$\| \cdots \|$

```
while (xn < 10) {
  if (xn < xn-1 )
    < xn ++ >
}
```

size	M: x_0	S_1: x_1	S_2: x_2	…	S_{n-1}: x_{n-1}	S_n: x_n
n	0	0	0		0	0

M S1 S2 … Sn M

Dijkstra's Token Ring: Existing Approaches

$C =$

```
while (x0 < 10){
  if (x0 == xsize)
    < x0 ++ >
}
```

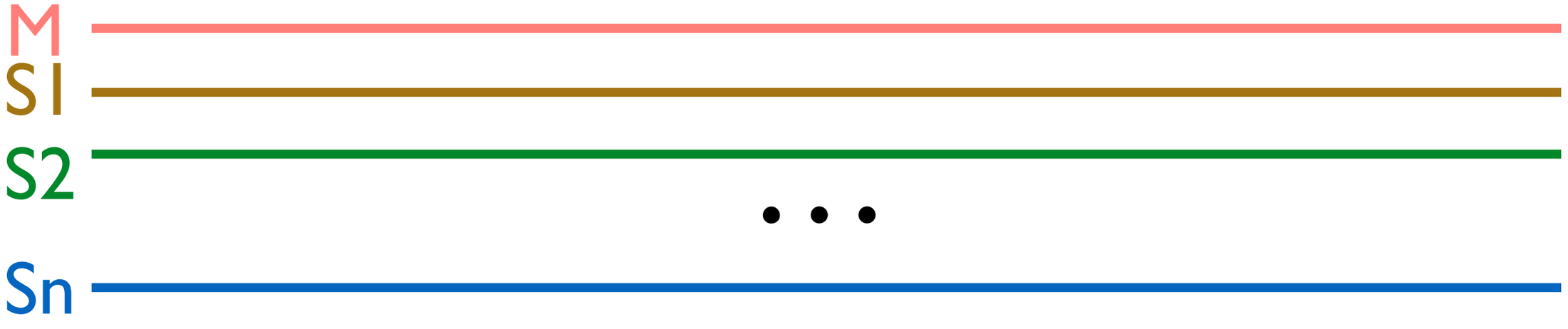
$\|$

```
while (x1 < 10) {
  if (x1 < x0 )
    < x1 ++ >
}
```

$\| \cdots \|$

```
while (xn < 10) {
  if (xn < xn-1 )
    < xn ++ >
}
```

size	M: x_0	S_1: x_1	S_2: x_2	…	S_{n-1}: x_{n-1}	S_n: x_n
n	0	0	0		0	0



Dijkstra's Token Ring: Possible Extension

E = while (true){
  if (random){
  $x_{(size+1)}$ = alloc(1);
  < $x_{(size+1)}$ = x_{size} ;
  size++ >
 }
 spawn (S_{size})
}

Dijkstra's Token Ring: Possible Extension
(CoLoSL)

C = M || S1 || S2 || … || Sn

C || E

size	M: x_0	S_1: x_1	S_2: x_2	…	S_{n-1}: x_{n-1}	S_n: x_n
n	0	0	0	…	0	0

M S1 S2 … Sn M

- The proofs (footprints) of slave threads are NOT affected
- Only the proof (footprint) of the master thread is affected
- Should not have to redo the proofs of slave threads

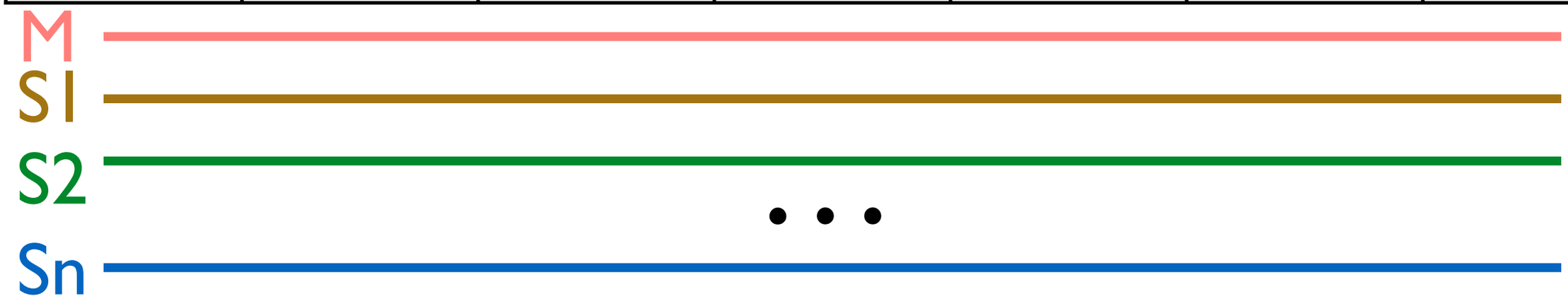
Dijkstra's Token Ring: Possible Extension

(Existing Approaches)

$C = M \,||\, S_1 \,||\, S_2 \,||\, \ldots \,||\, S_n$

C || E

size	M: x_0	S_1: x_1	S_2: x_2	…	S_{n-1}: x_{n-1}	S_n: x_n
n	0	0	0	…	0	0

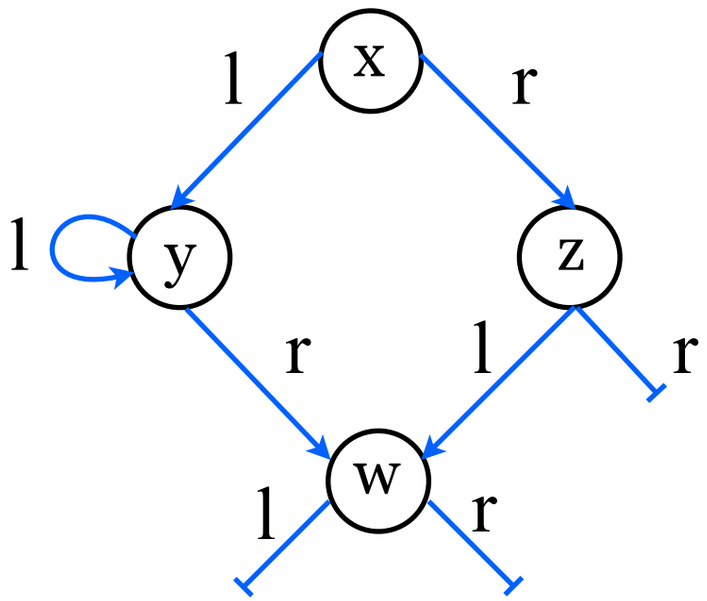


- The proofs (footprints) of ALL threads are affected!
- Have to redo all proofs

Why CoLoSL?

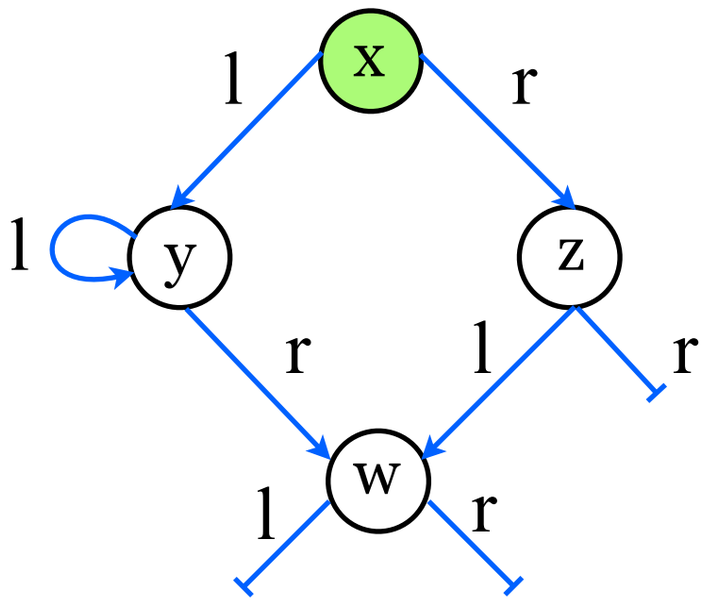
- **Local** Proofs
  - Proof reuse - proofs done for the largest possible context
  - Simpler/More intuitive proofs

Example - Spanning Tree



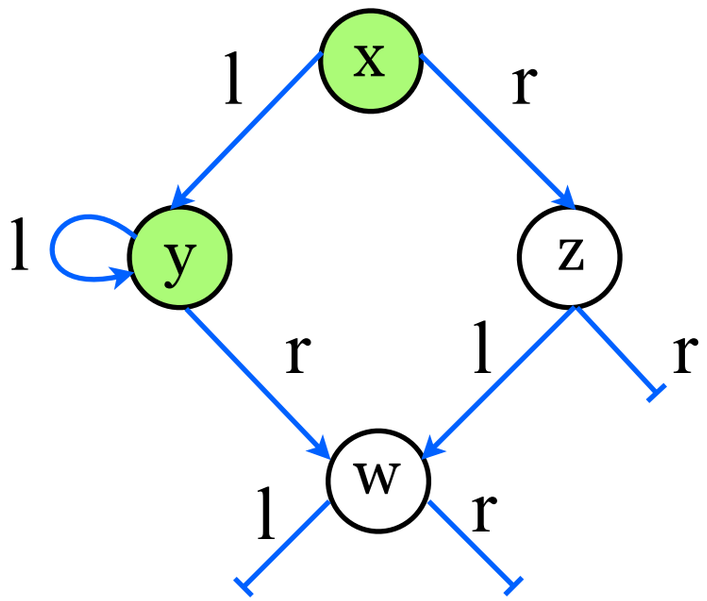

```
b:= spanning(x) {
   b:= <CAS(x.m, 0, 1)>;
   if (b) then {
      b1:= spanning(x.l) || b2:= spanning(x.r);
      if (!b1) then
         x.l:= null
      if (!b2) then
         x.r:= null
   }
   return b;
}
```

Example - Spanning Tree



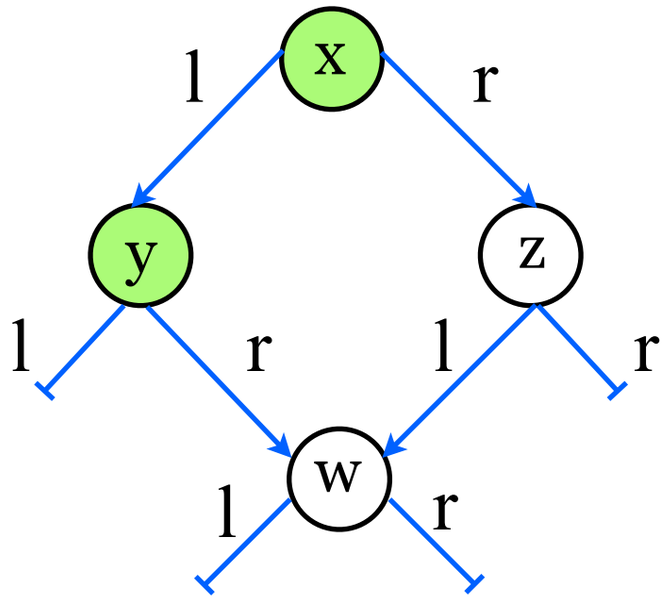

```
b:= spanning(x) {
   b:= <CAS(x.m, 0, 1)>;
   if (b) then {
      b1:= spanning(x.l) || b2:= spanning(x.r);
      if (!b1) then
         x.l:= null
      if (!b2) then
         x.r:= null
   }
   return b;
}
```

Example - Spanning Tree



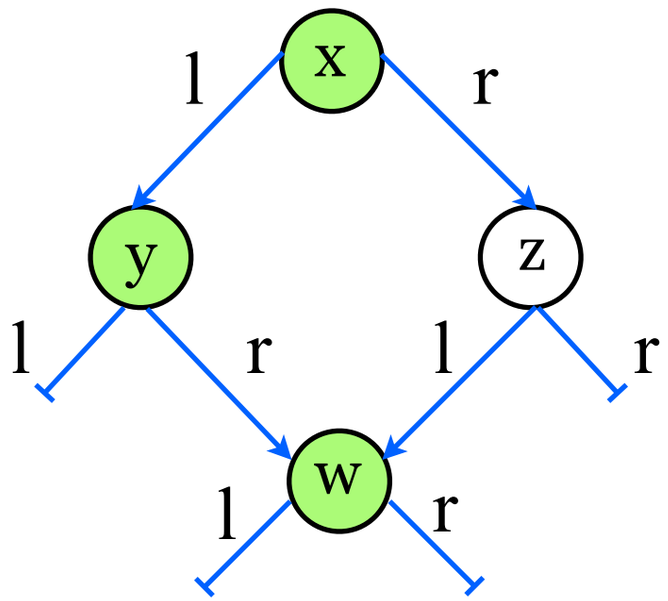

```
b:= spanning(x) {
   b:= <CAS(x.m, 0, 1)>;
   if (b) then {
      b1:= spanning(x.l) || b2:= spanning(x.r);
      if (!b1) then
         x.l:= null
      if (!b2) then
         x.r:= null
   }
   return b;
}
```

Example - Spanning Tree



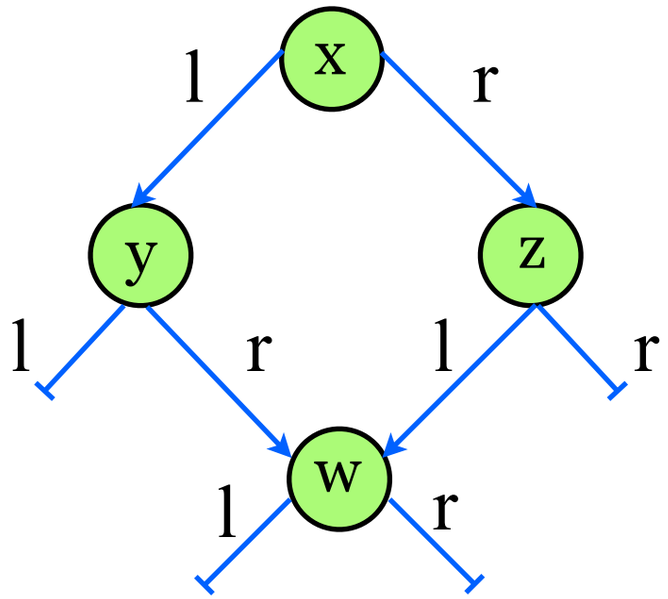

```
b:= spanning(x) {
   b:= <CAS(x.m, 0, 1)>;
   if (b) then {
      b1:= spanning(x.l) || b2:= spanning(x.r);
      if (!b1) then
         x.l:= null
      if (!b2) then
         x.r:= null
   }
   return b;
}
```

Example - Spanning Tree



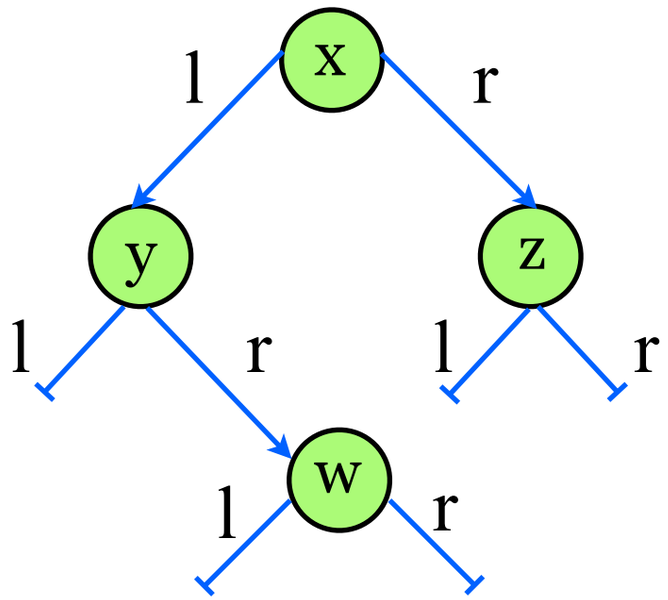

```
b:= spanning(x) {
   b:= <CAS(x.m, 0, 1)>;
   if (b) then {
      b1:= spanning(x.l) || b2:= spanning(x.r);
      if (!b1) then
         x.l:= null
      if (!b2) then
         x.r:= null
   }
   return b;
}
```

Example - Spanning Tree



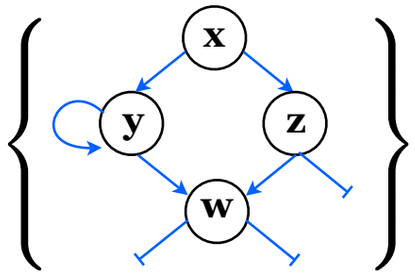

```
b:= spanning(x) {
   b:= <CAS(x.m, 0, 1)>;
   if (b) then {
      b1:= spanning(x.l) || b2:= spanning(x.r);
      if (!b1) then
         x.l:= null
      if (!b2) then
         x.r:= null
   }
   return b;
}
```

Example - Spanning Tree

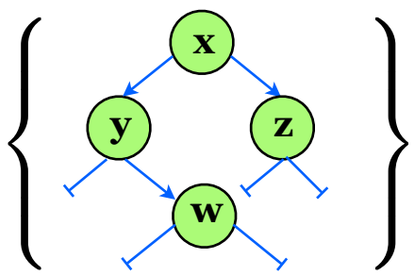



```
b:= spanning(x) {
   b:= <CAS(x.m, 0, 1)>;
   if (b) then {
      b1:= spanning(x.l) || b2:= spanning(x.r);
      if (!b1) then
         x.l:= null
      if (!b2) then
         x.r:= null
   }
   return b;
}
```

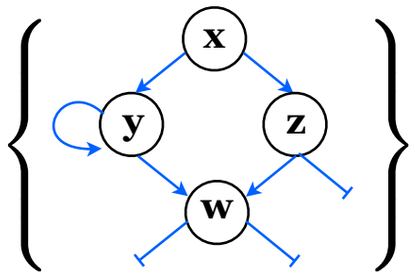
Example - Spanning Tree



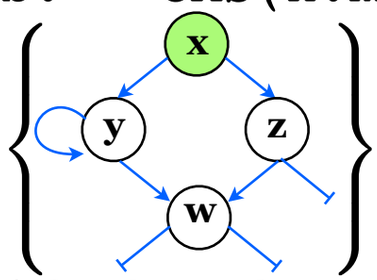

```
b:= spanning(x) {
   b:= <CAS(x.m, 0, 1)>;
   if (b) then {
      b1:= spanning(x.l) || b2:= spanning(x.r);
      if (!b1) then
         x.l:= null
      if (!b2) then
         x.r:= null
   }
   return b;
}
```



Example - Spanning Tree

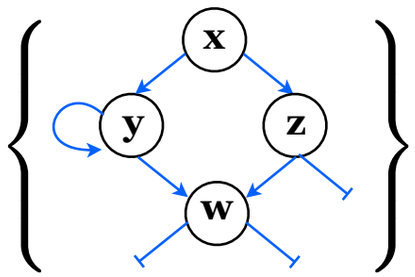



```
b:= spanning(x) {
   b:= <CAS(x.m, 0, 1)>;
```

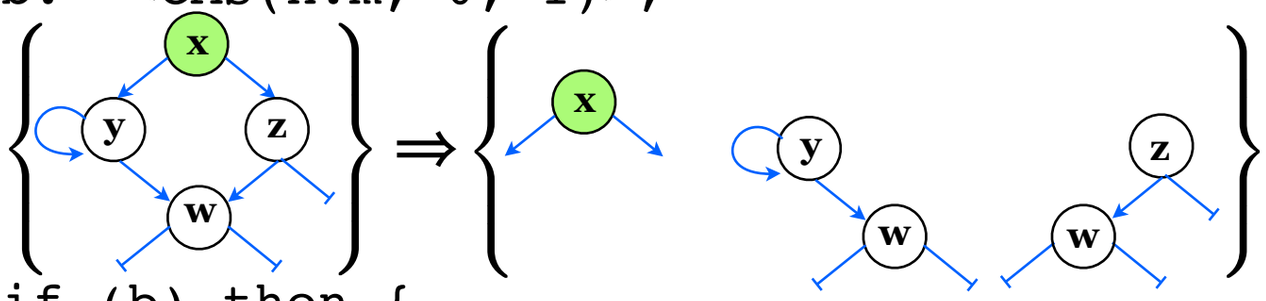



```
   if (b) then {
     b1:= spanning(x.l) || b2:= spanning(x.r);
     if (!b1) then
       x.l:= null
     if (!b2) then
       x.r:= null
   }
   return b;
}
```

Example - Spanning Tree

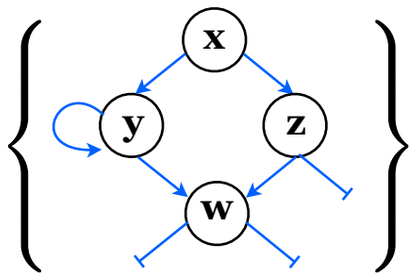



```
b:= spanning(x) {
   b:= <CAS(x.m, 0, 1)>;
```

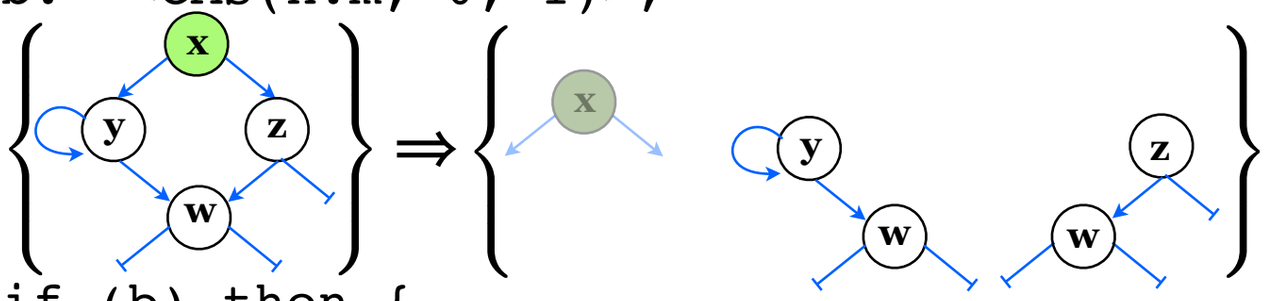



```
   if (b) then {
     b1:= spanning(x.l) || b2:= spanning(x.r);
     if (!b1) then
       x.l:= null
     if (!b2) then
       x.r:= null
   }
   return b;
}
```

Example - Spanning Tree

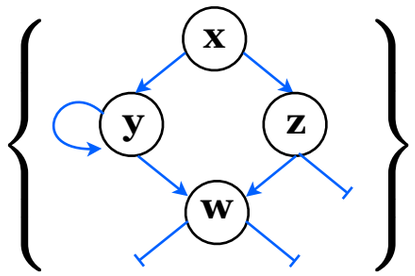



```
b:= spanning(x) {
   b:= <CAS(x.m, 0, 1)>;
```

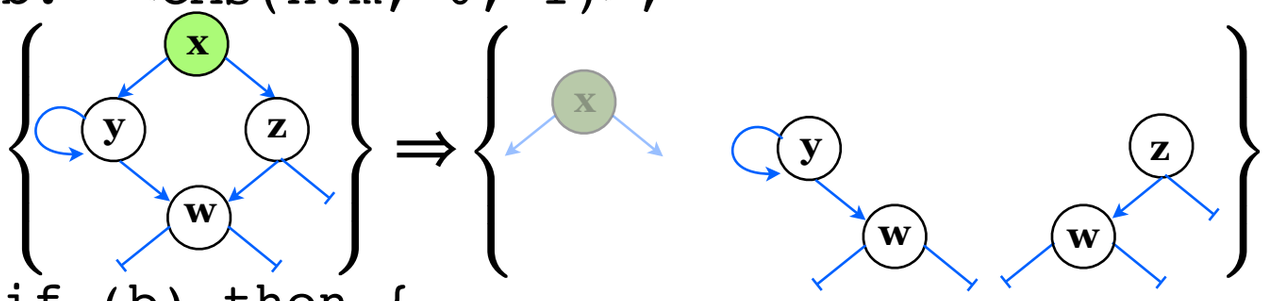



```
   if (b) then {
     b1:= spanning(x.l) || b2:= spanning(x.r);
     if (!b1) then
       x.l:= null
     if (!b2) then
       x.r:= null
   }
   return b;
}
```

Example - Spanning Tree



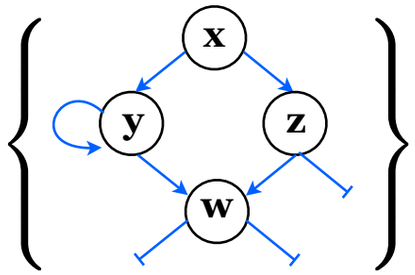

```
b:= spanning(x) {
   b:= <CAS(x.m, 0, 1)>;
```



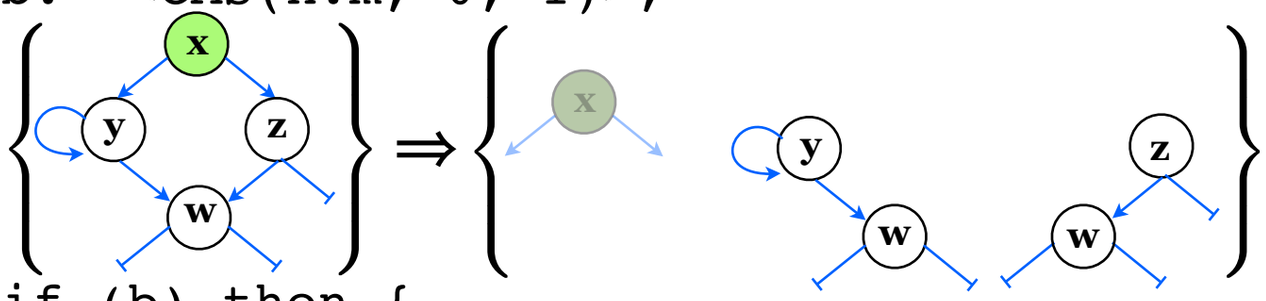

```
   if (b) then {
```


```
     b1:= spanning(x.l) || b2:= spanning(x.r);
     if (!b1) then
       x.l:= null
     if (!b2) then
       x.r:= null
   }
   return b;
}
```

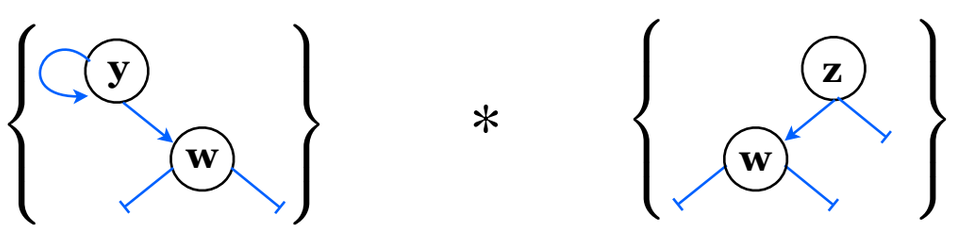
Example - Spanning Tree




```
b:= spanning(x) {
   b:= <CAS(x.m, 0, 1)>;
```

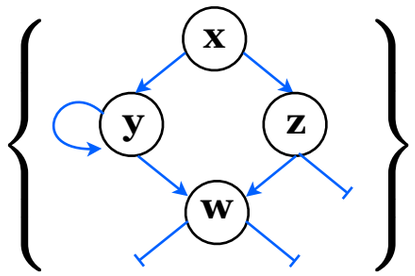



```
   if (b) then {
```

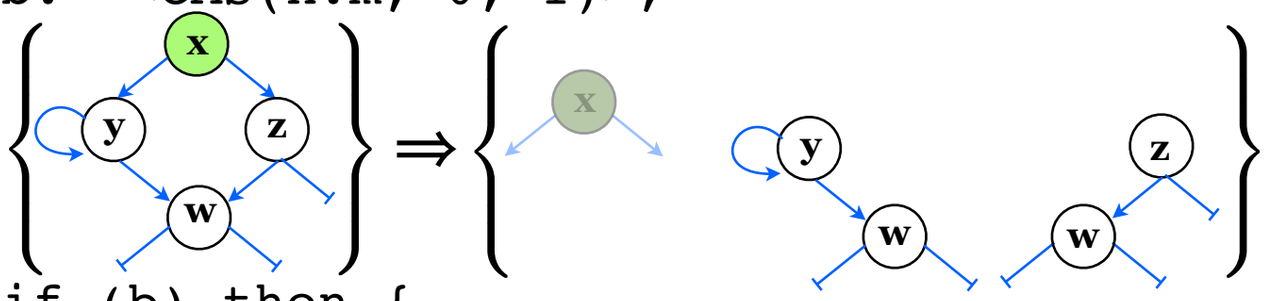



```
     b1:= spanning(x.l) || b2:= spanning(x.r);
     if (!b1) then
       x.l:= null
     if (!b2) then
       x.r:= null
   }
   return b;
}
```

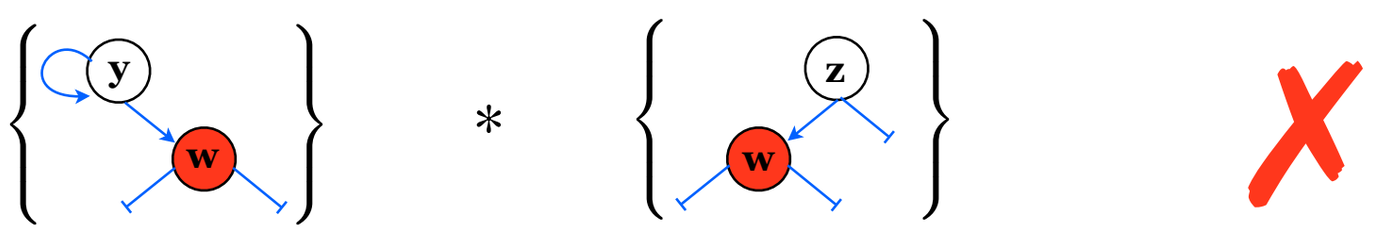
Example - Spanning Tree




```
b:= spanning(x) {
   b:= <CAS(x.m, 0, 1)>;
```

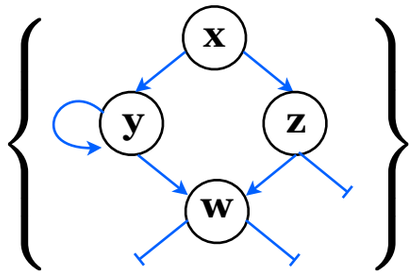



```
   if (b) then {
```

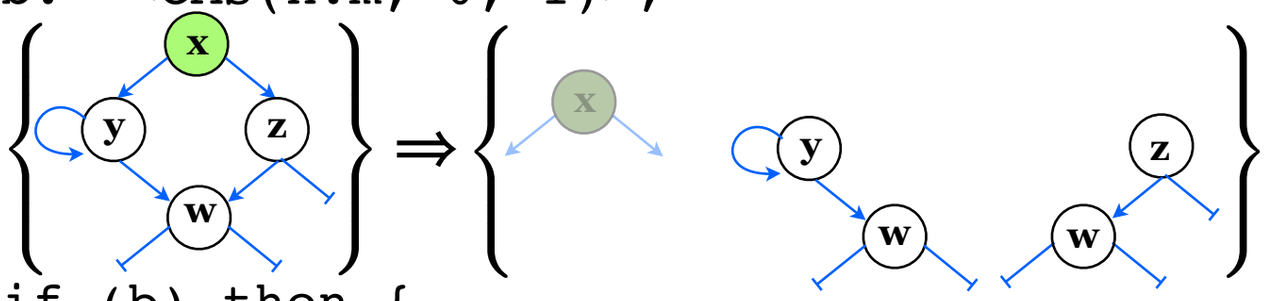



```
     b1:= spanning(x.l) || b2:= spanning(x.r);
     if (!b1) then
       x.l:= null
     if (!b2) then
       x.r:= null
   }
   return b;
}
```

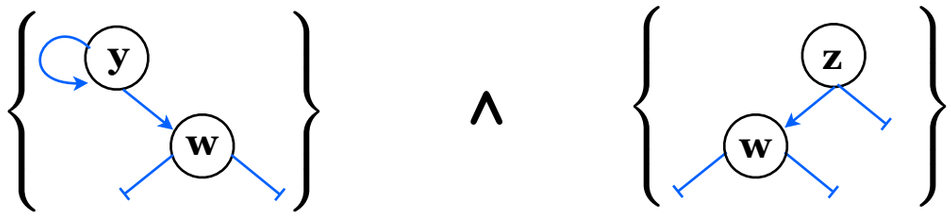
Example - Spanning Tree




```
b:= spanning(x) {
   b:= <CAS(x.m, 0, 1)>;
```

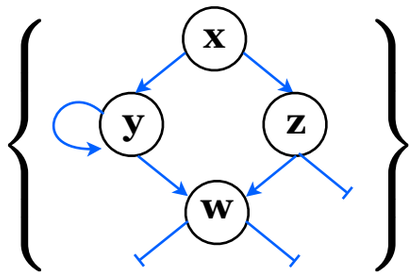



```
   if (b) then {
```

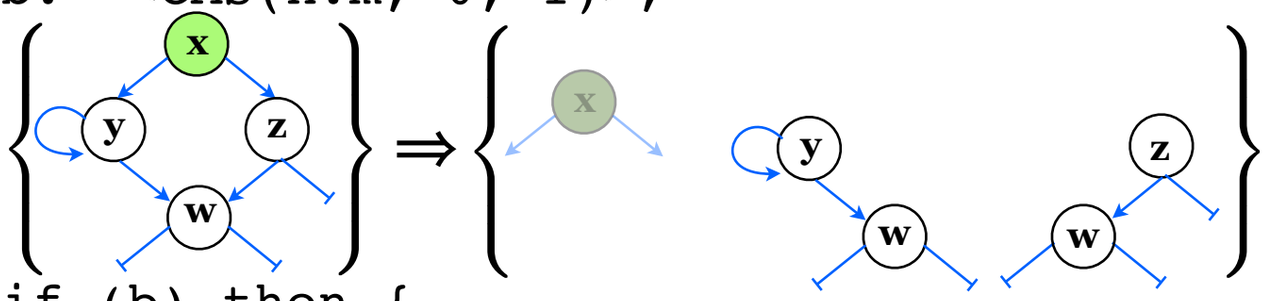


```
     b1:= spanning(x.l) || b2:= spanning(x.r);
     if (!b1) then
       x.l:= null
     if (!b2) then
       x.r:= null
   }
   return b;
}
```

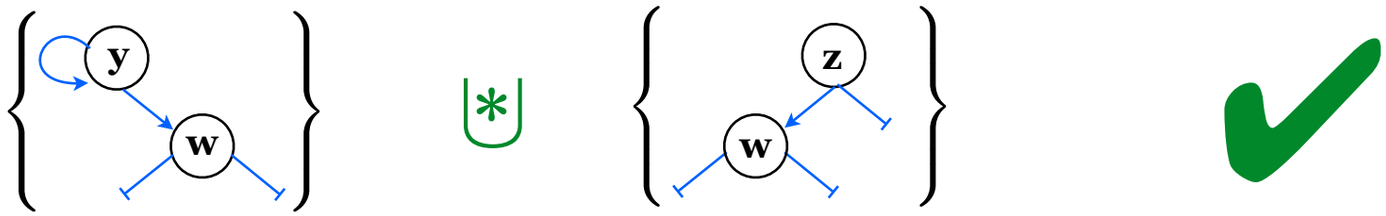
Example - Spanning Tree




```
b:= spanning(x) {
   b:= <CAS(x.m, 0, 1)>;
```

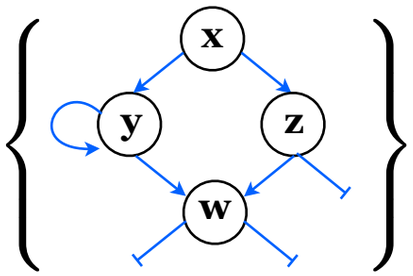



```
   if (b) then {
```

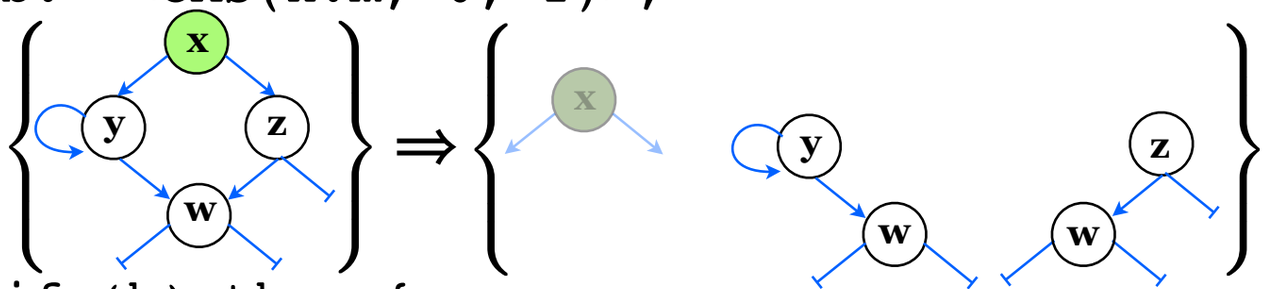



```
     b1:= spanning(x.l) || b2:= spanning(x.r);
     if (!b1) then
       x.l:= null
     if (!b2) then
       x.r:= null
   }
   return b;
}
```

Example - Spanning Tree




```
b:= spanning(x) {
   b:= <CAS(x.m, 0, 1)>;
```

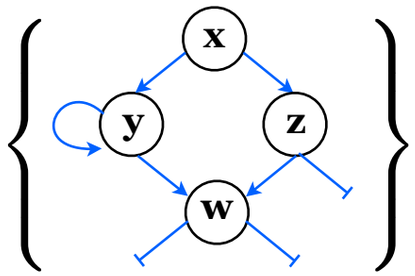



```
   if (b) then {
```

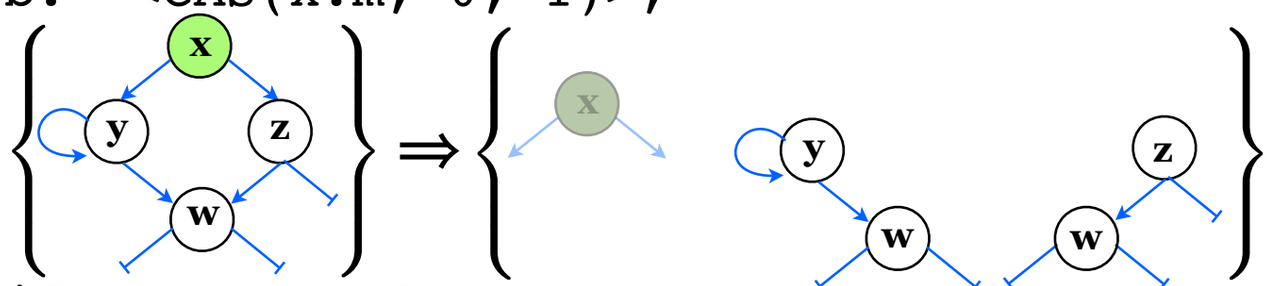

```
     b1:= spanning(x.l) || b2:= spanning(x.r);
```


```
     if (!b1) then
       x.l:= null
     if (!b2) then
       x.r:= null
   }
   return b;
}
```

Example - Spanning Tree

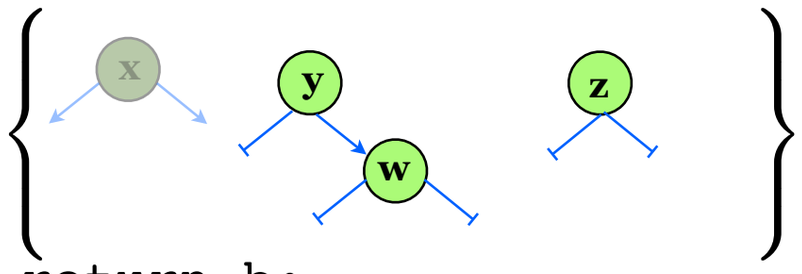



```
b:= spanning(x) {
   b:= <CAS(x.m, 0, 1)>;
```



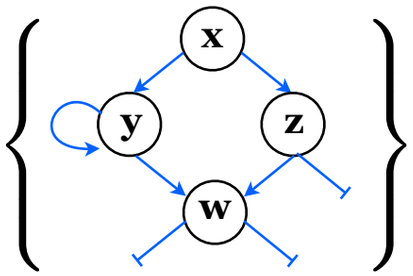

```
   if (b) then {
     b1:= spanning(x.l) || b2:= spanning(x.r);
```


```
     if (!b1) then
       x.l:= null
     if (!b2) then
       x.r:= null
   }
```

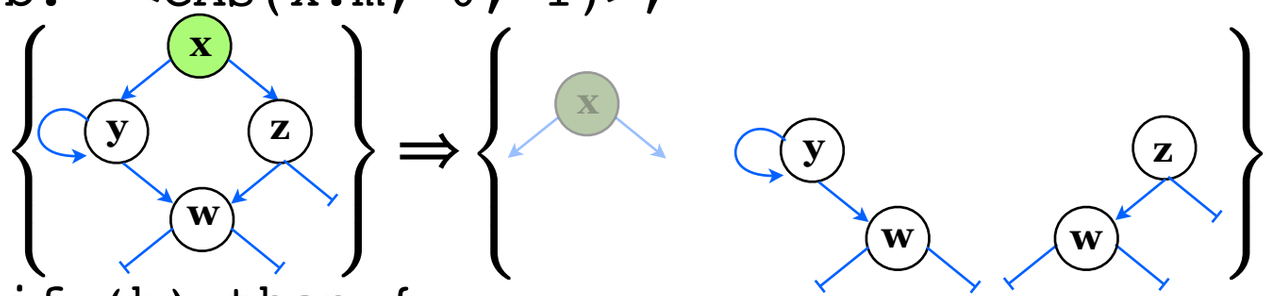



```
   return b;
}
```

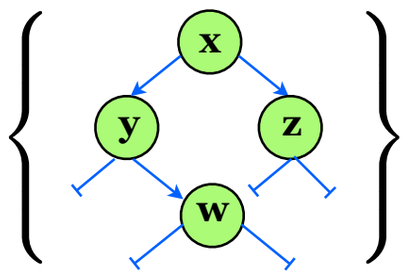
Example - Spanning Tree




```
b:= spanning(x) {
   b:= <CAS(x.m, 0, 1)>;
```

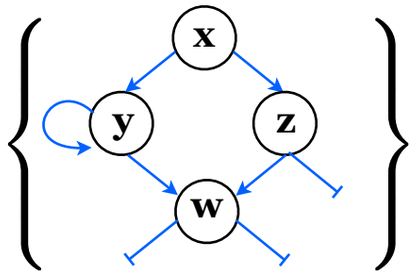



```
   if (b) then {
     b1:= spanning(x.l) || b2:= spanning(x.r);
     if (!b1) then
       x.l:= null
     if (!b2) then
       x.r:= null
   }
   return b;
```

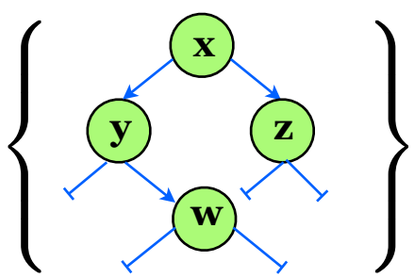



```
}
```

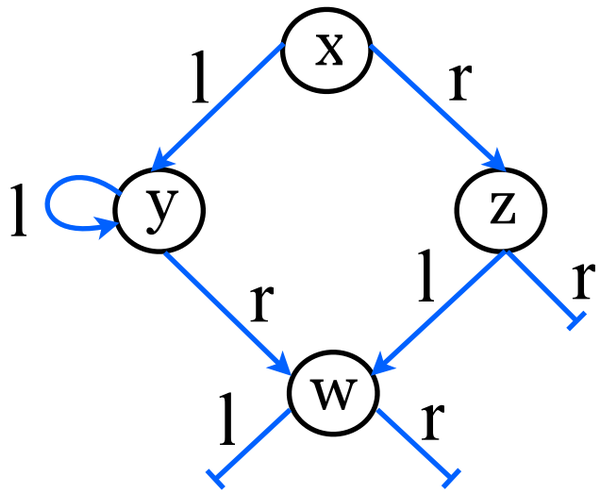
Example - Spanning Tree




```
b:= spanning(x) {
   b:= <CAS(x.m, 0, 1)>;
   if (b) then {
      b1:= spanning(x.l) || b2:= spanning(x.r);
      if (!b1) then
         x.l:= null
      if (!b2) then
         x.r:= null
   }
   return b;
}
```



Example - Spanning Tree



$g(x) = x \doteq \texttt{null} \ \vee$

$\exists\, m, l, r.\ \ x \mapsto m, l, r \ \ \uplus^* \ \ graph(l) \ \ \uplus^* \ \ graph(r)$

=

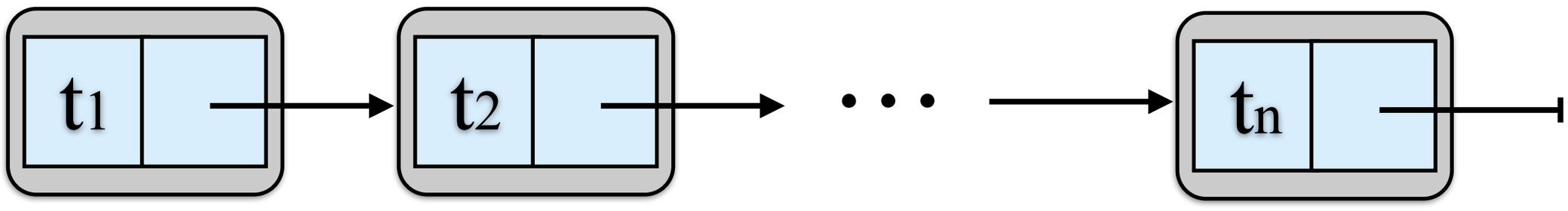
g(x)

Why CoLoSL?

- **Local** Proofs
  - Proof reuse - proofs done for the largest possible context
  - Simpler/More intuitive proofs
- **Subjective/ Overlapping** Shared Resources
  - More modular; better abstraction

Ordered Singly Linked-List <T>

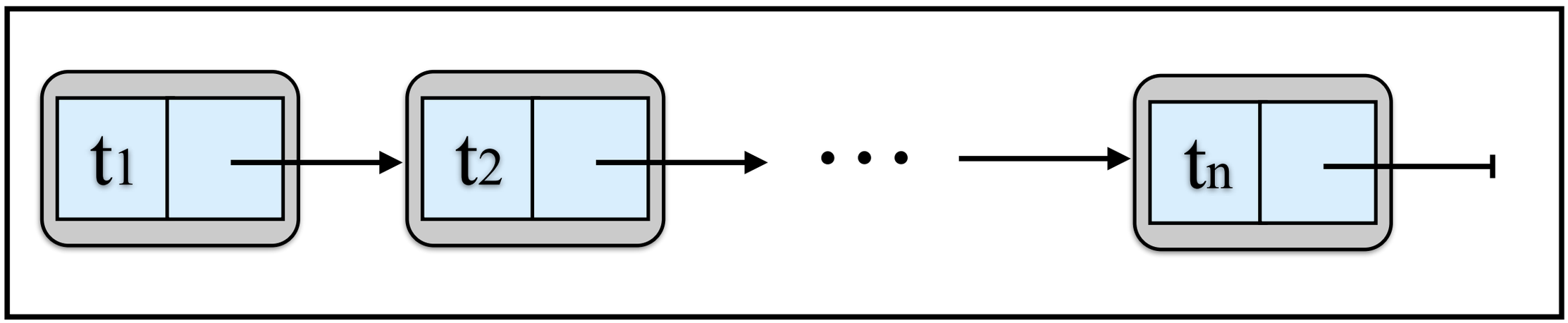
List([t_1, t_2, … , t_n]) =



- **<u>insertion/removal</u>** involves pointer surgery

Concurrent Ordered Singly Linked-List<T>

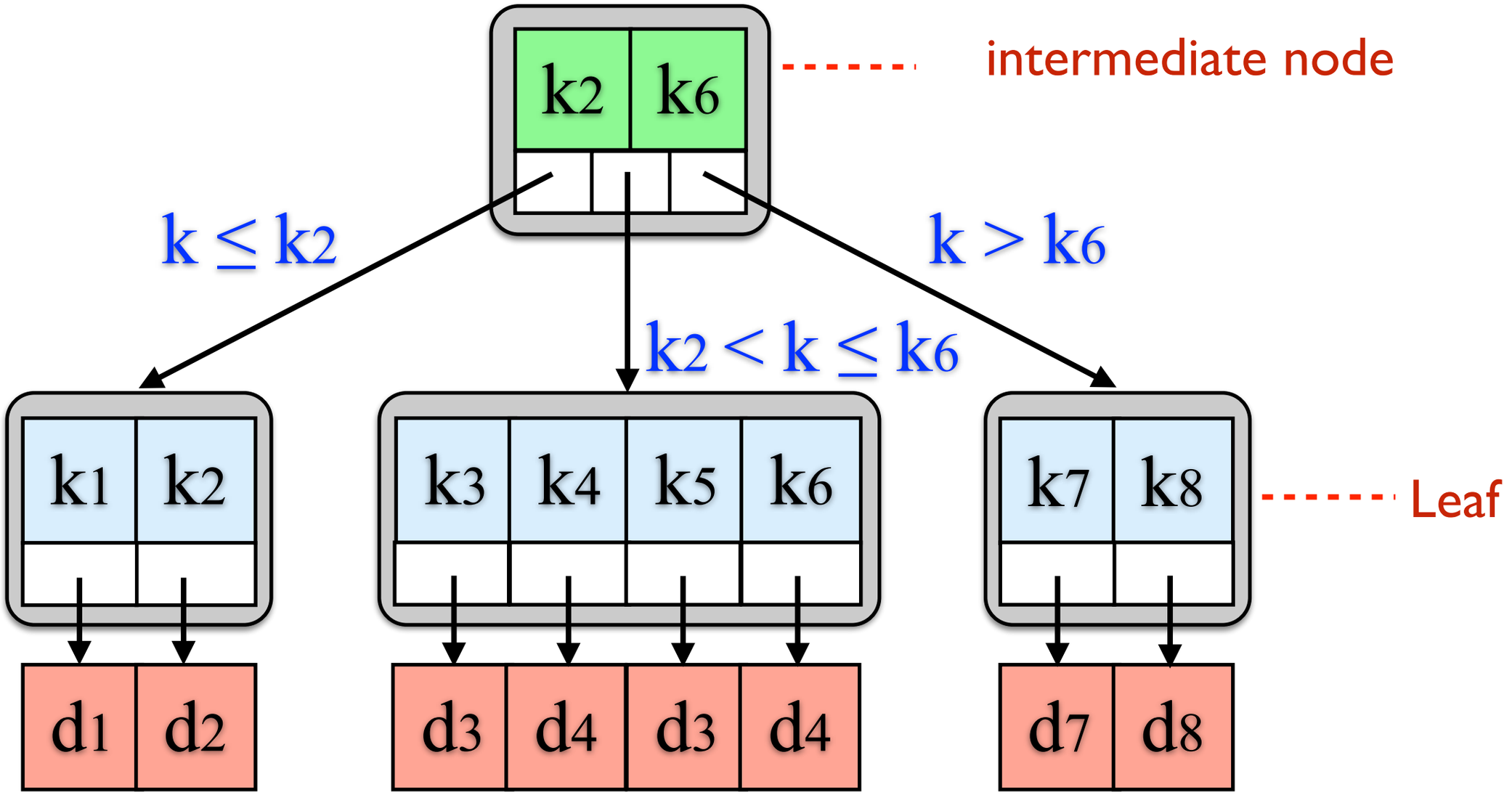
Con_List([t_1, t_2, … , t_n) =



$I_L = I_{add} \cup I_{rem}$

- **<u>insertion/removal</u>** involves pointer surgery

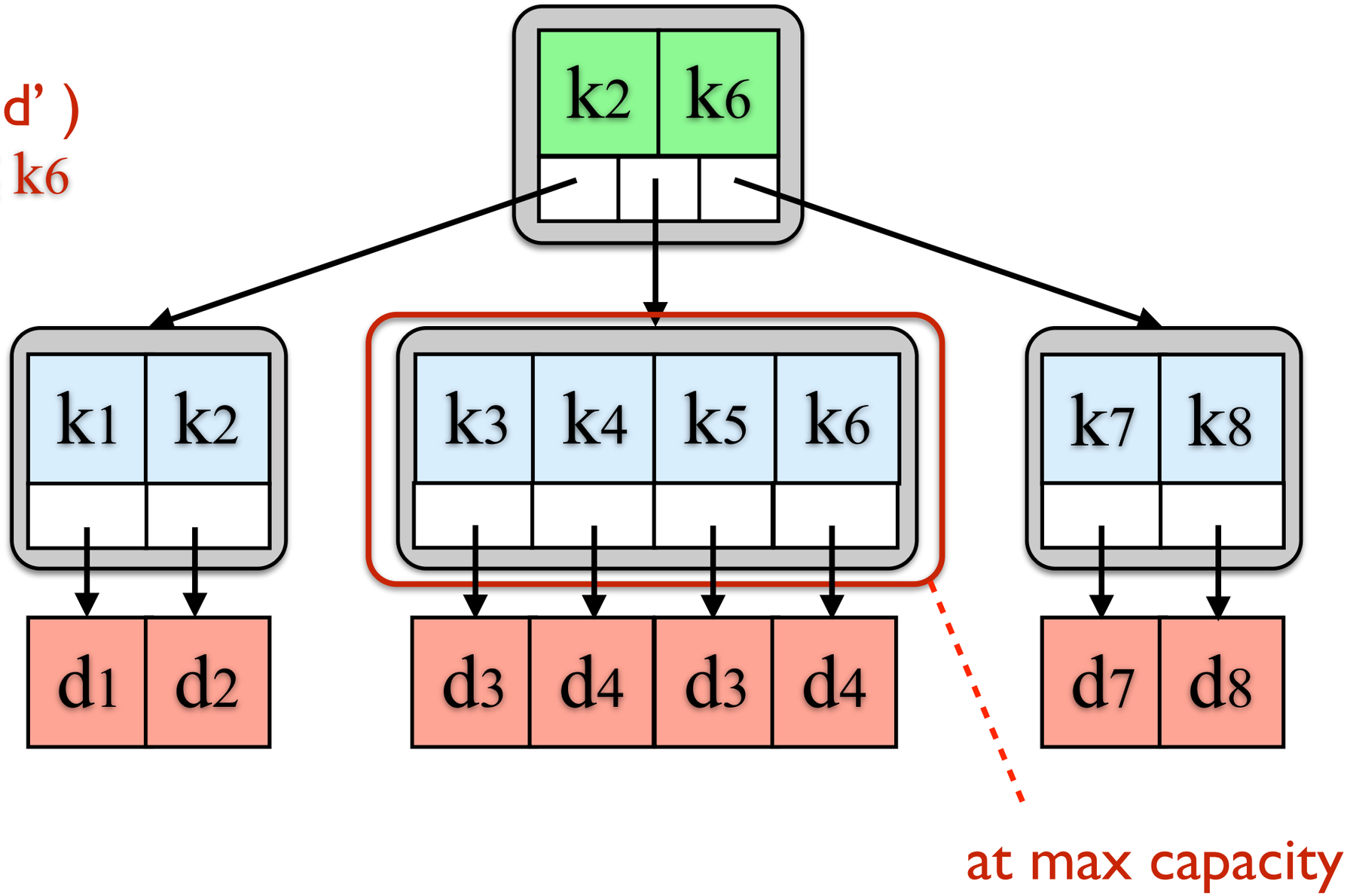
Balanced Search Tree<K,V> (Degree 2)



- **Balanced**: all immediate subtrees of a node have the same height
- **Leaf-heavy**: Data (values) stored in leaf nodes
- **Degree (d)**: no. of children (m) on each node $d \leq m \leq 2d$

Balanced Search Tree<K,V> (Degree 2)

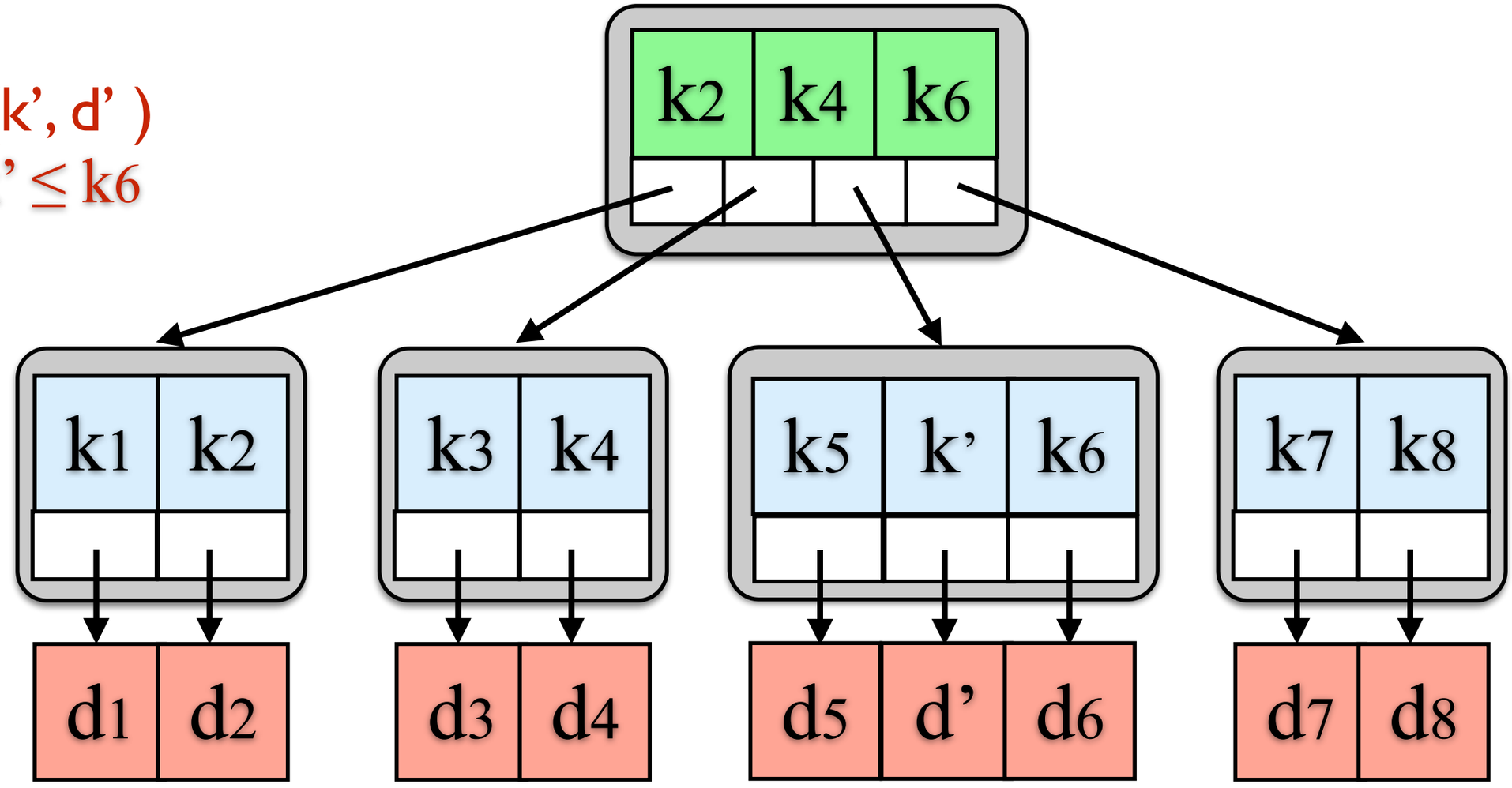
insert(k', d')
$k5 < k' \leq k6$



- **Insertion** may require **splitting**

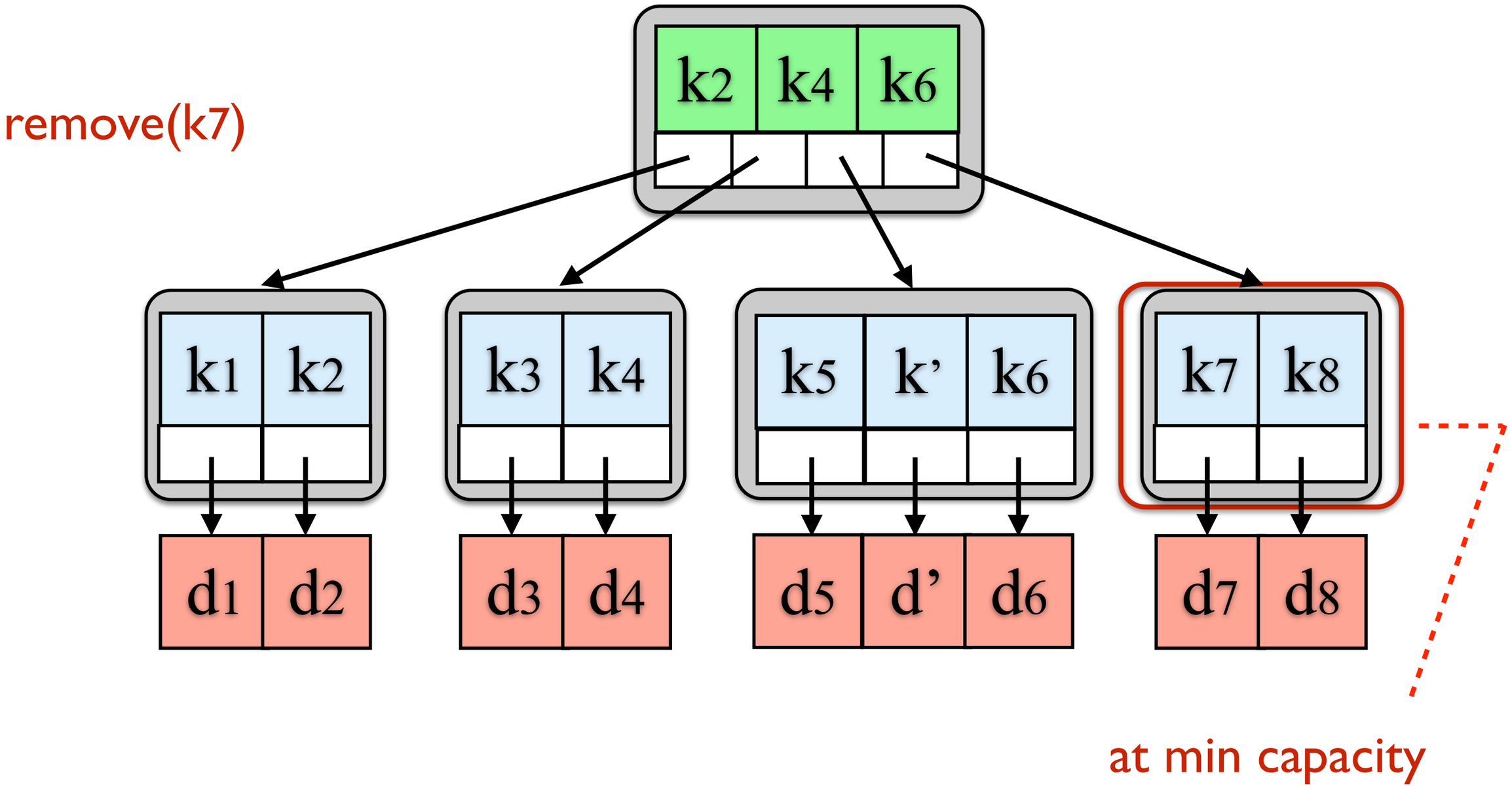
Balanced Search Tree<K,V> (Degree 2)

insert(k', d')
$k5 < k' \leq k6$



- **Insertion** may require **splitting**

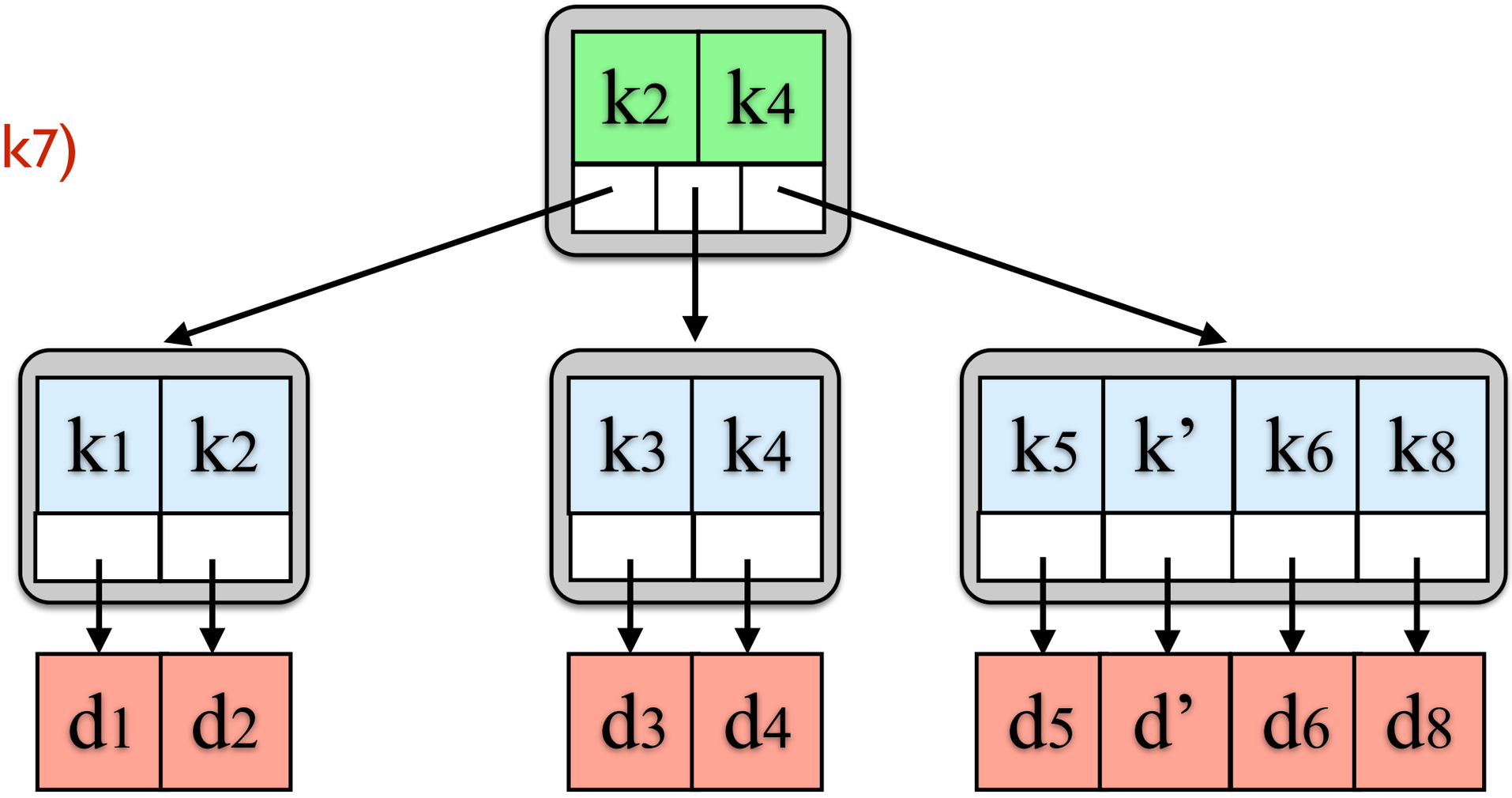
Balanced Search Tree<K,V> (Degree 2)



- **Removal** may require **merging**

Balanced Search Tree<K,V> (Degree 2)

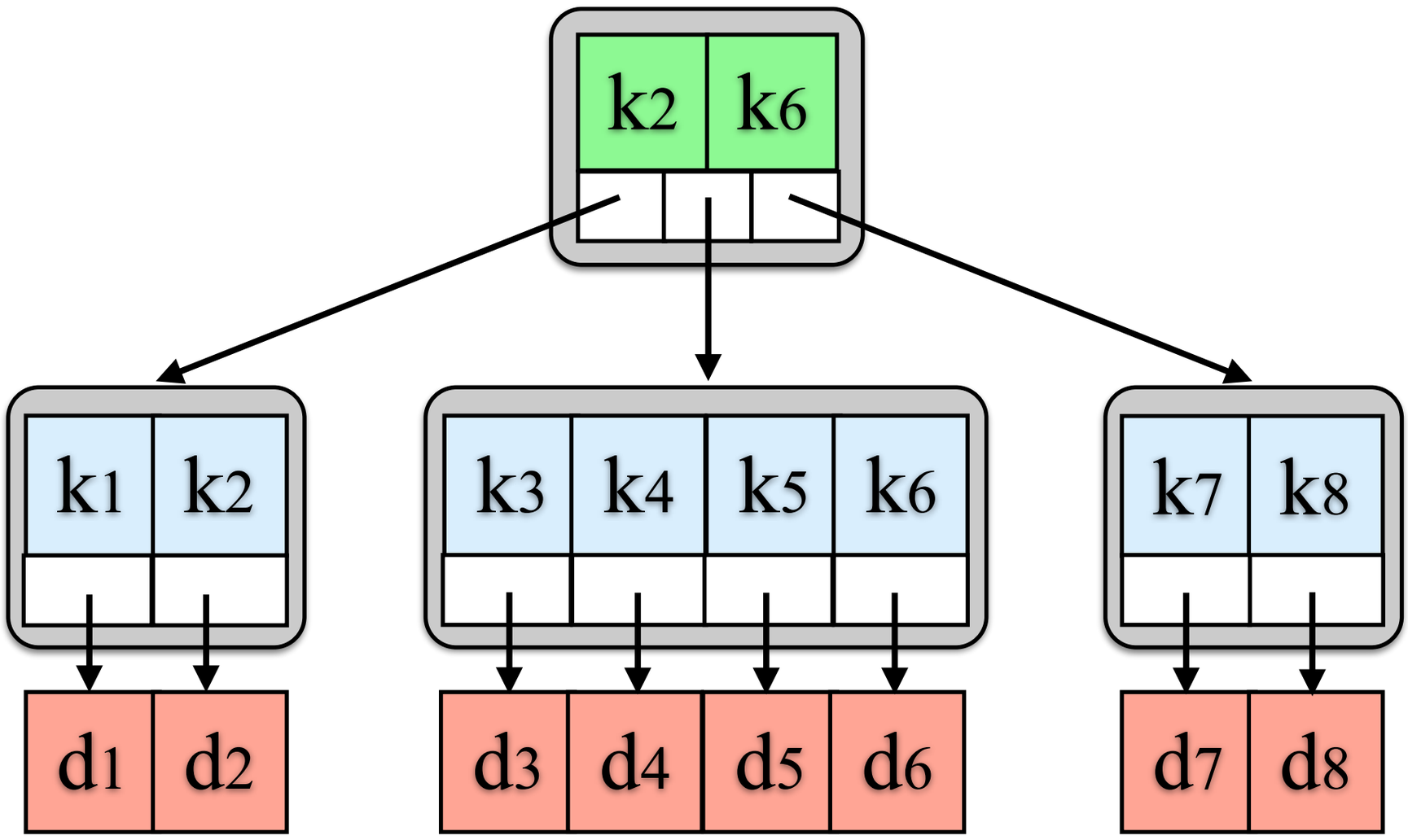
remove(k7)



- **Removal** may require **merging**

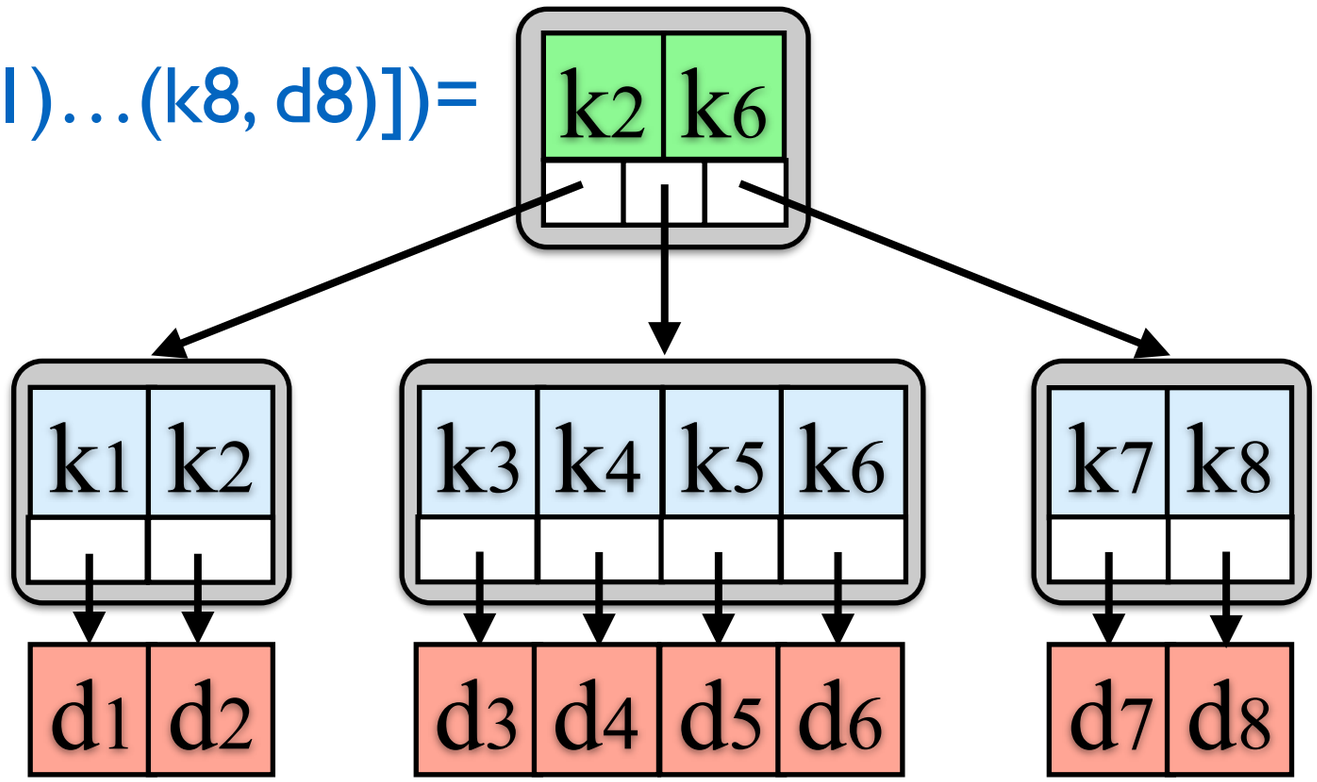
Balanced Search Tree<K,V> (Degree 2)

BTree 2 ([(k1, d1) … (k8, d8)]) =



Balanced Search Tree<K,V> (Degree 2)

BTree 2 ([(k1, d1)…(k8, d8)])=



BTree (h+1) L<K,V> =

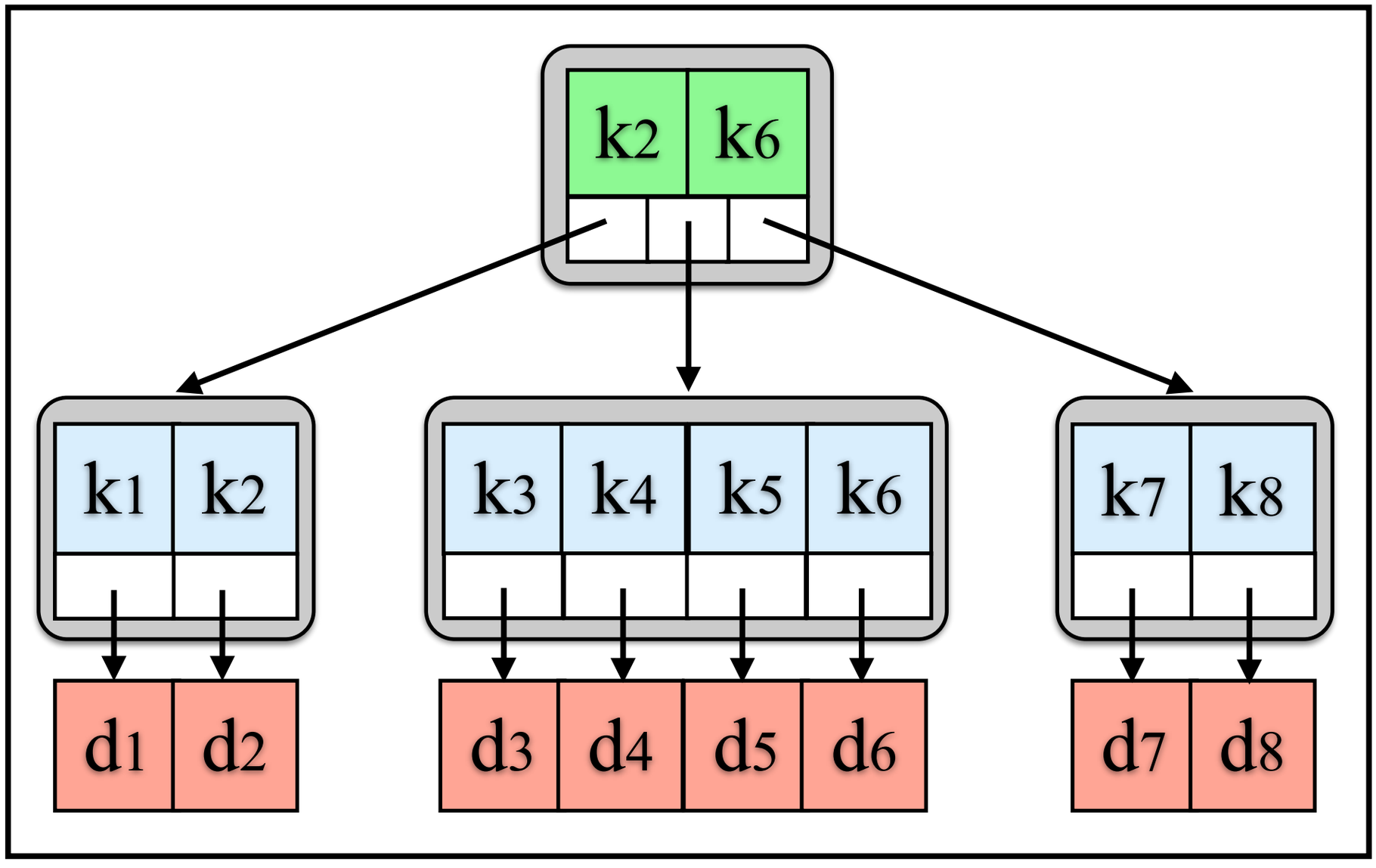
∃ k1,…, km, L1, …, L(m+1) . L= L1 ⊎ … ⊎ L(m+1)

Node (k1,…, km) (BTree h L1) … (BTree h L(m+1))

BTree 1 L<K,V> = Leaf L

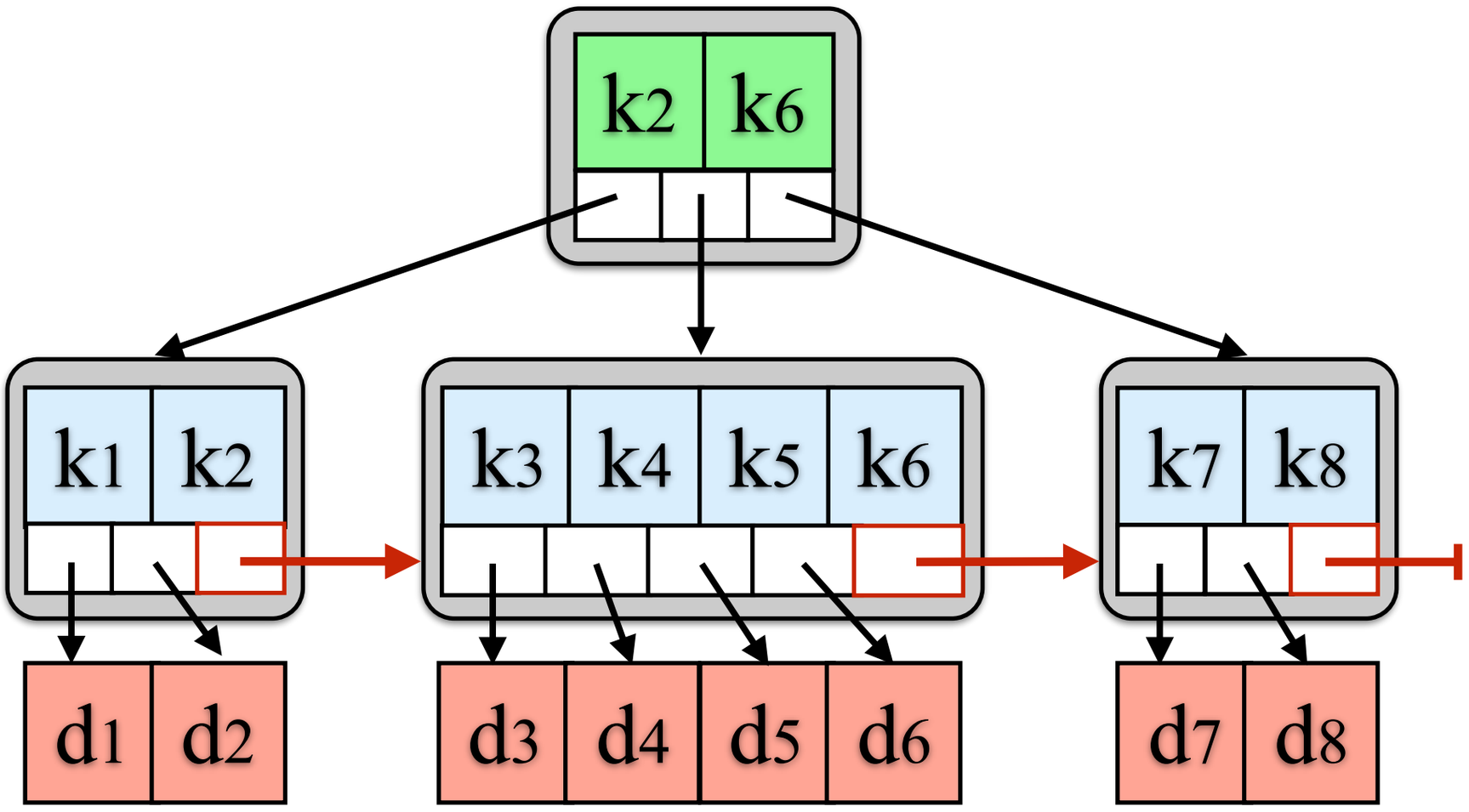
Concurrent Balanced Search Tree<K,V>

Con_BTree([(k1, d1) … (k8, d8)]) =

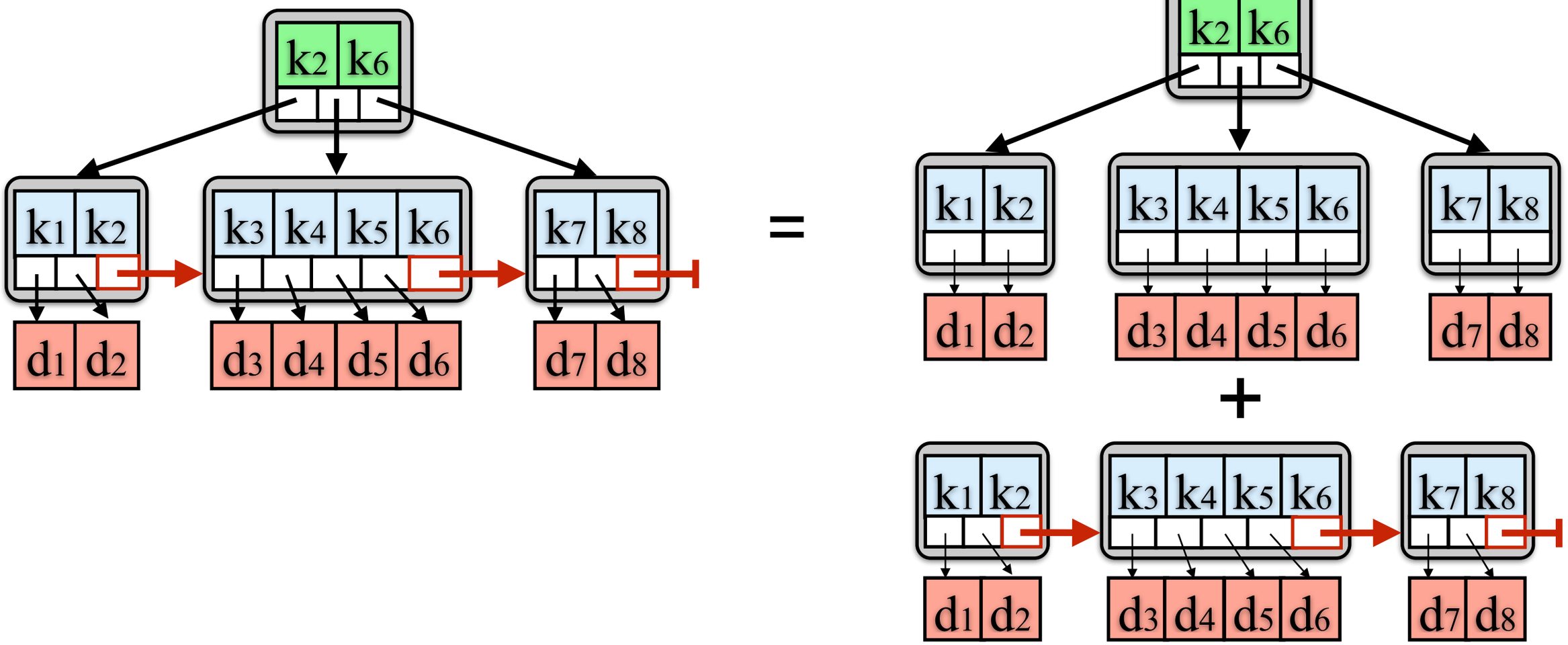


$I_B = I_{ins} \cup I_{del}$

B+ Tree <K,V> (Degree 2)



B+ Tree <K,V> (Degree 2)

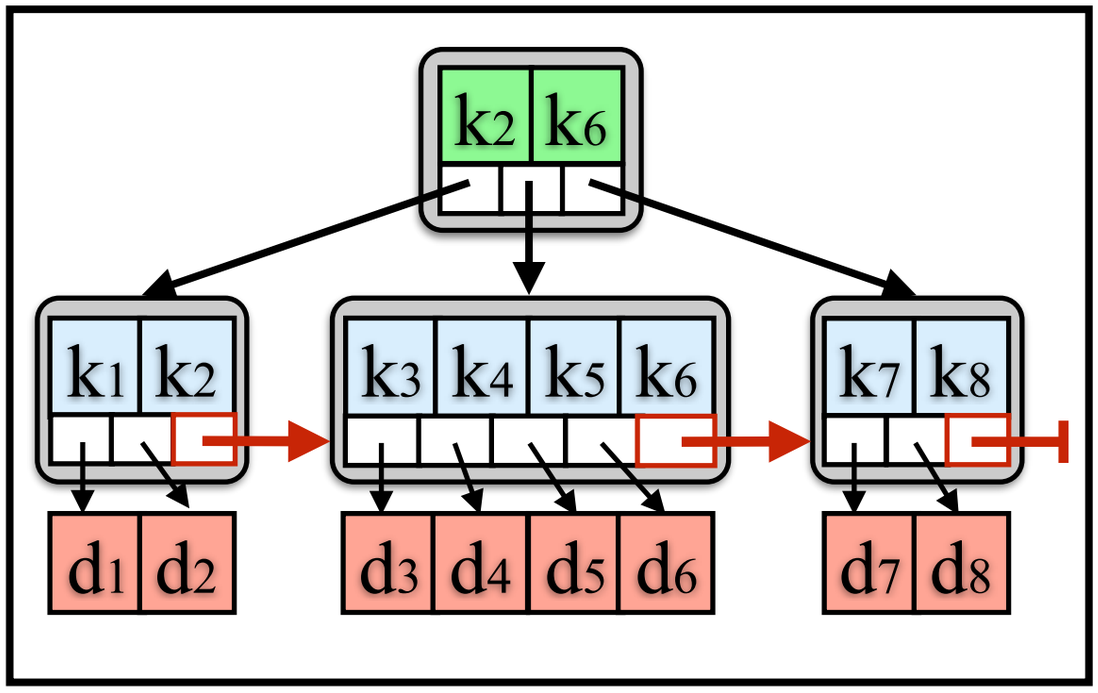


B^+ Tree([(k1, d1) … (k8, d8)]) = BTree([(k1, d1) … (k8, d8)])

+

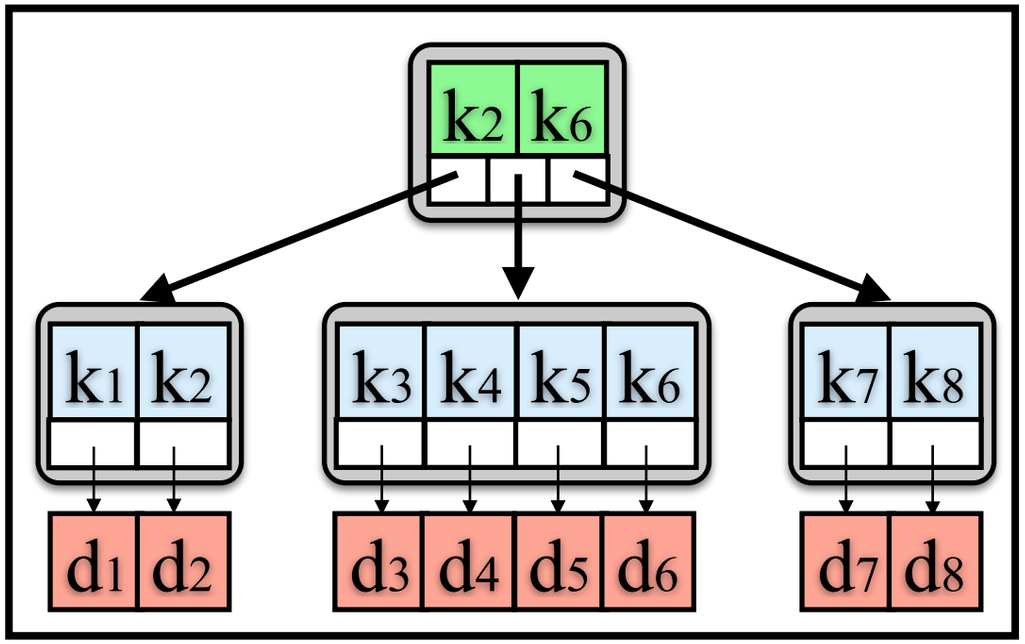
List[Leaf L1, Leaf L2, Leaf L3]

Concurrent B+ Tree <K,V> (Degree 2)



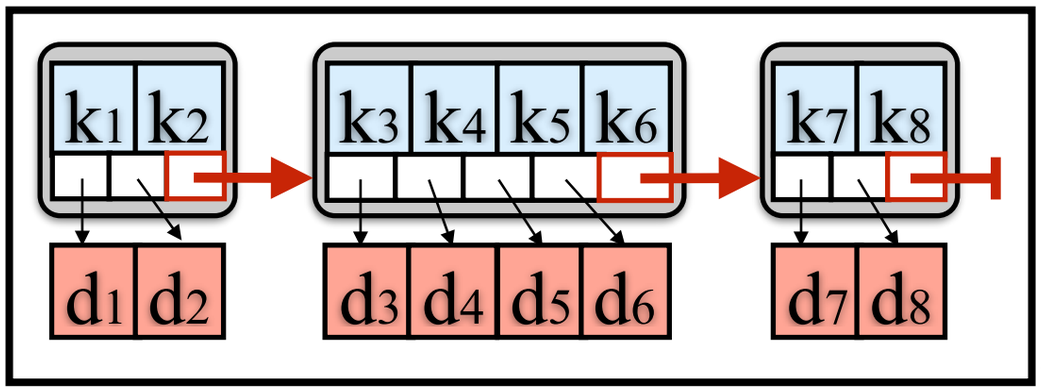
I

=



I_B

+



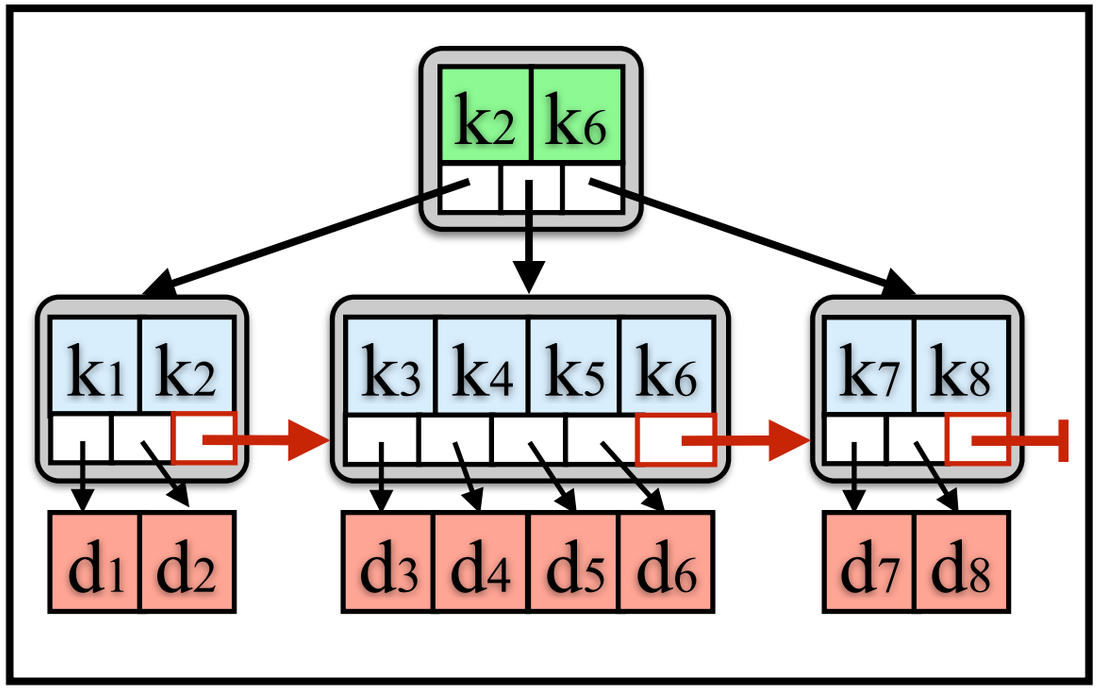
I_L

Con_B^+ Tree([(k1, d1) … (k8, d8)]) = Con_BTree([(k1, d1) … (k8, d8)])

+

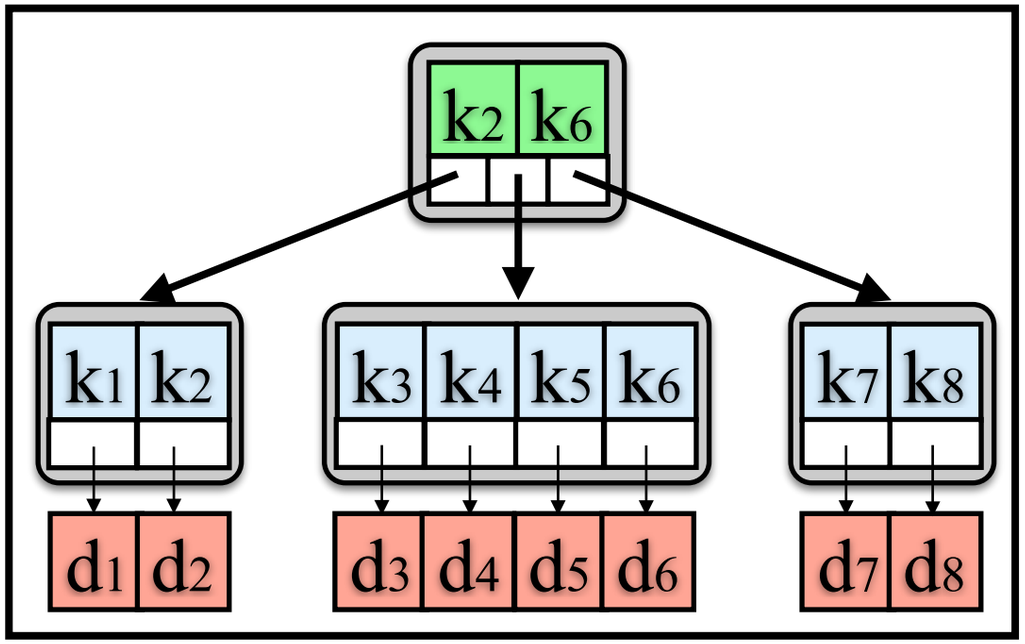
Con_List[Leaf L1, Leaf L2, Leaf L3]

Concurrent B+ Tree <K,V> (Degree 2)



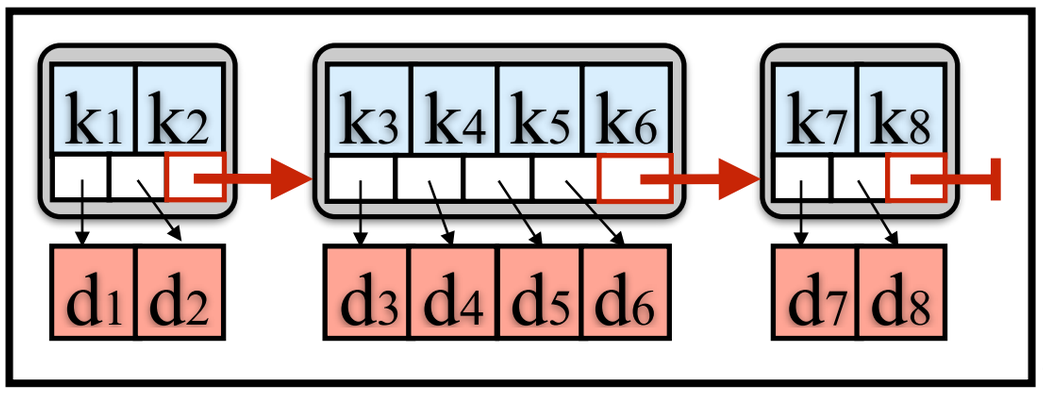
I

=



I_B

?



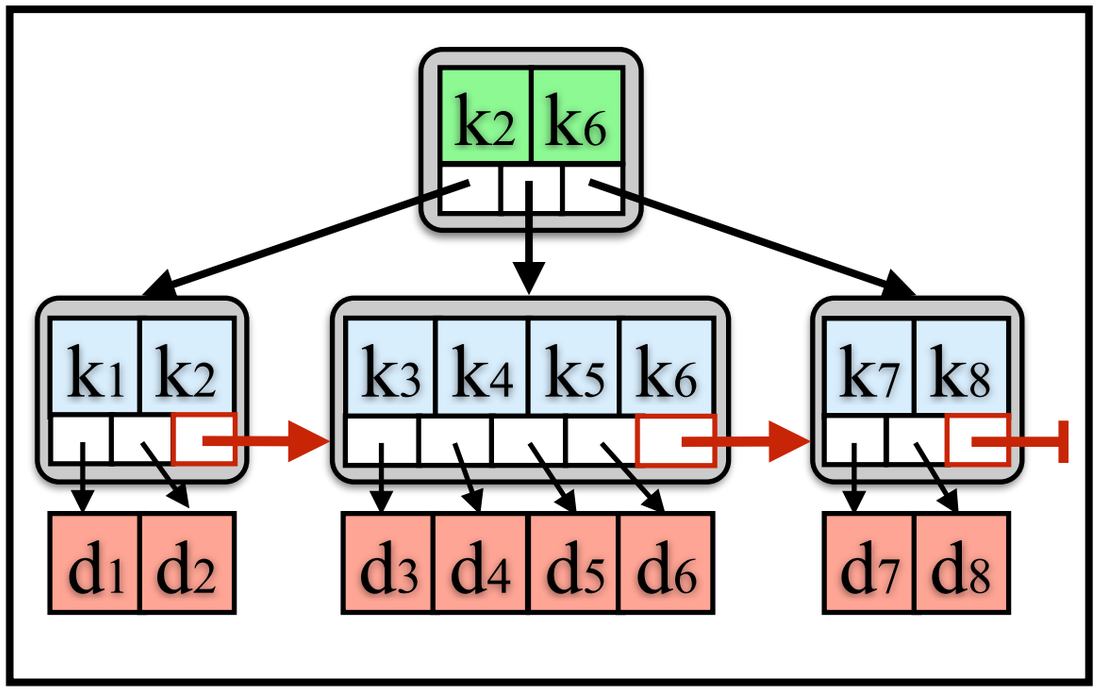
I_L

Con_B^+ Tree([(k1, d1) … (k8, d8)]) = Con_BTree([(k1, d1) … (k8, d8)])

+

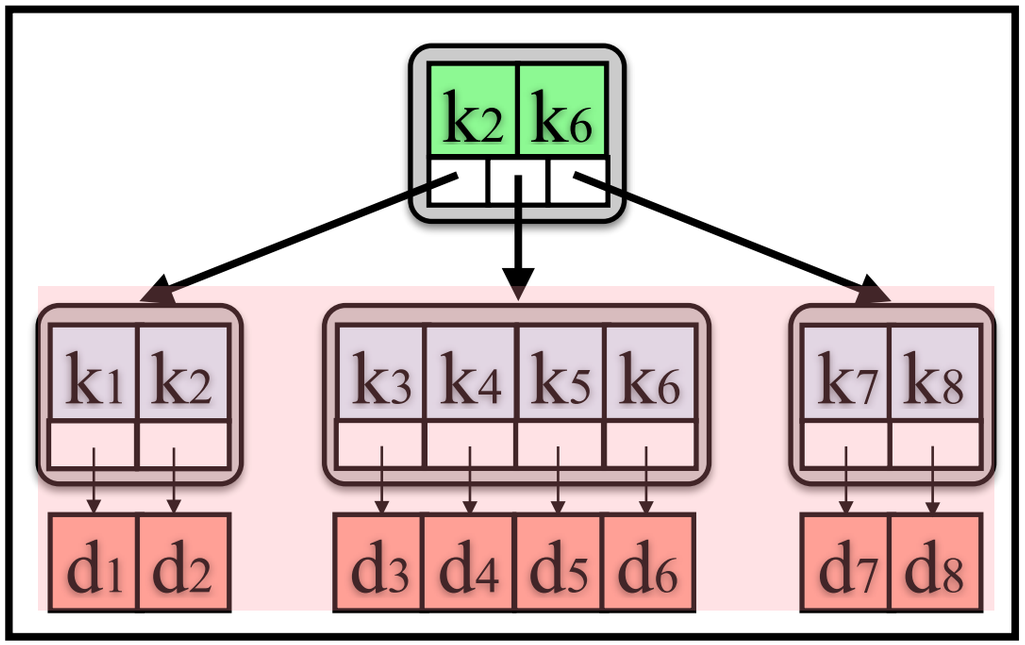
Con_List[Leaf L1, Leaf L2, Leaf L3]

B+ Tree <K,V> - Existing Approaches



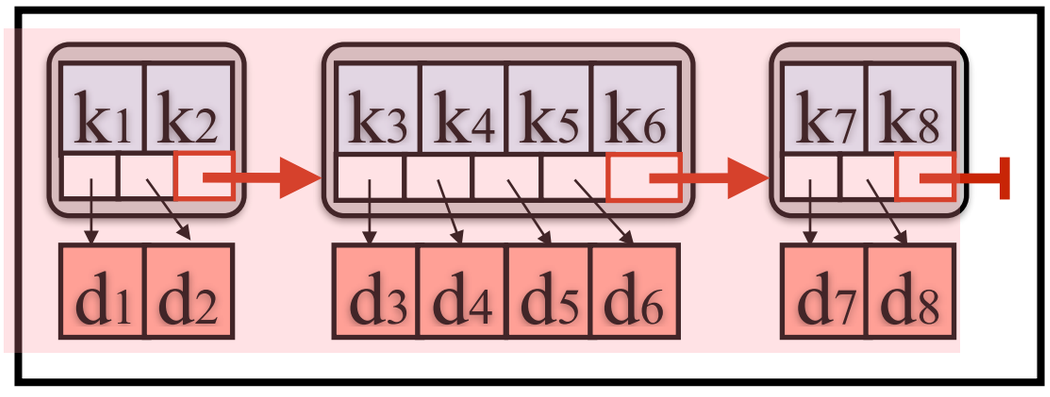
I

=



I_B

* ✗



I_L

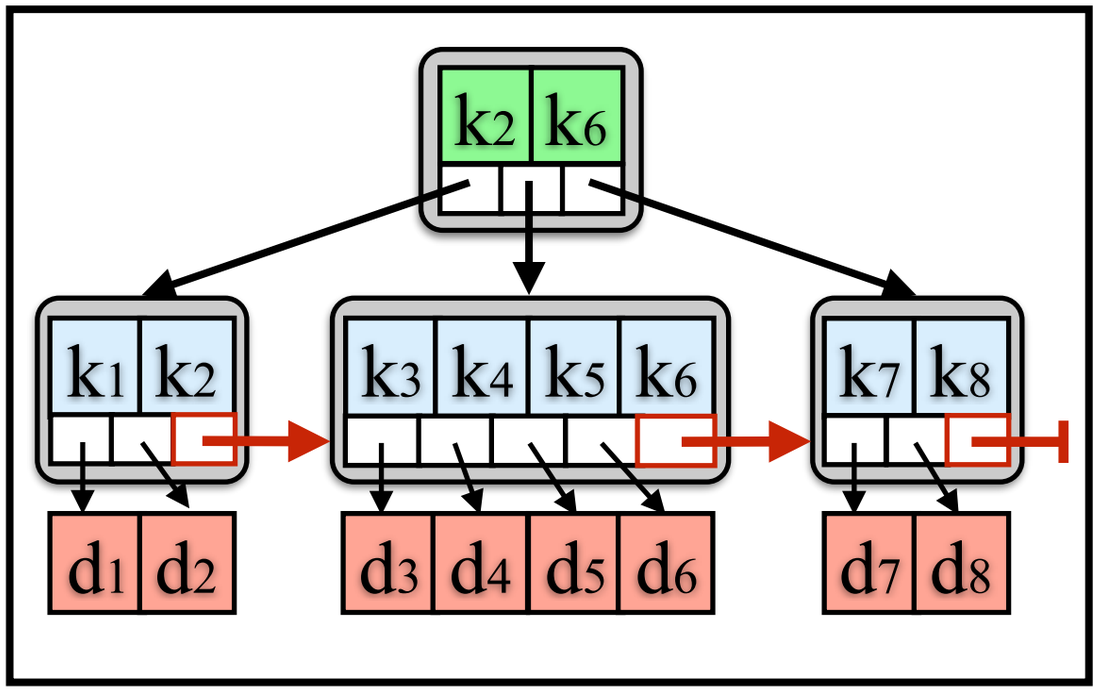
Con_B+Tree([(k1, d1) … (k8, d8)]) = Con_BTree([(k1, d1) … (k8, d8)])

+

Con_List[Leaf L1, Leaf L2, Leaf L3]

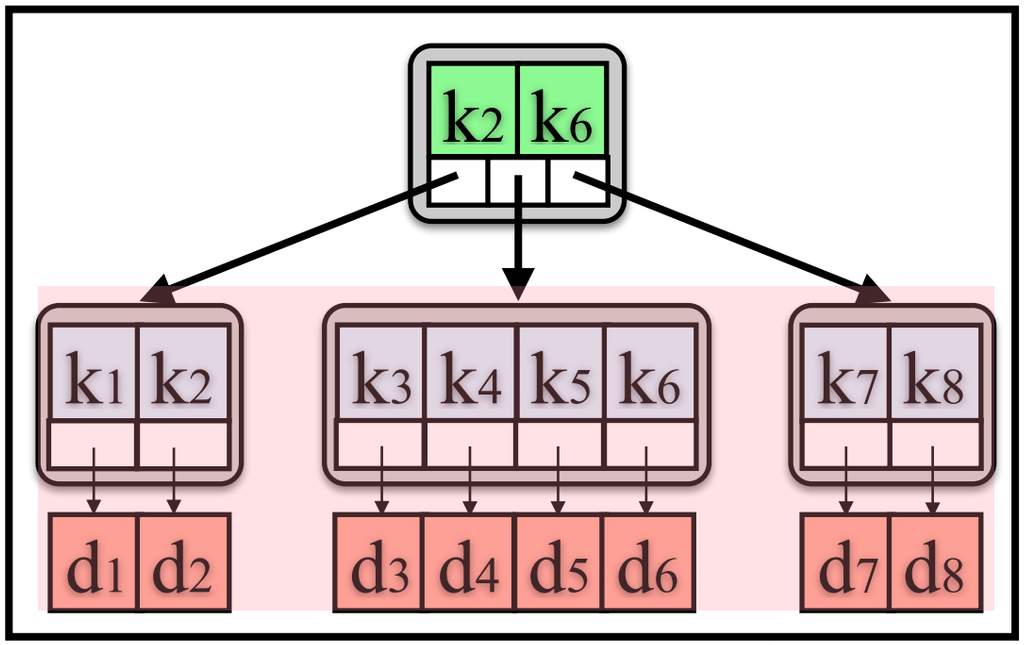
Existing approaches do not allow overlaps

B+ Tree <K,V> - CoLoSL



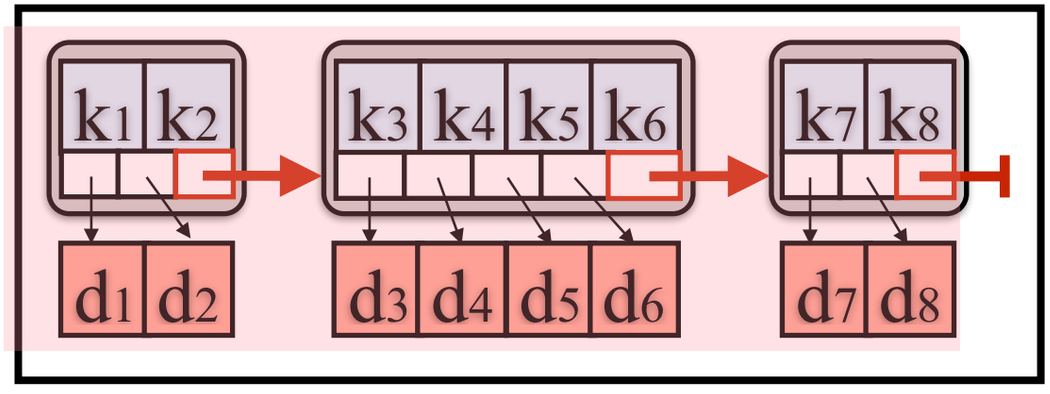
I

=



I_B

*



I_L

- CoLoSL allows overlapping subjective views

Why Overlapping Views?

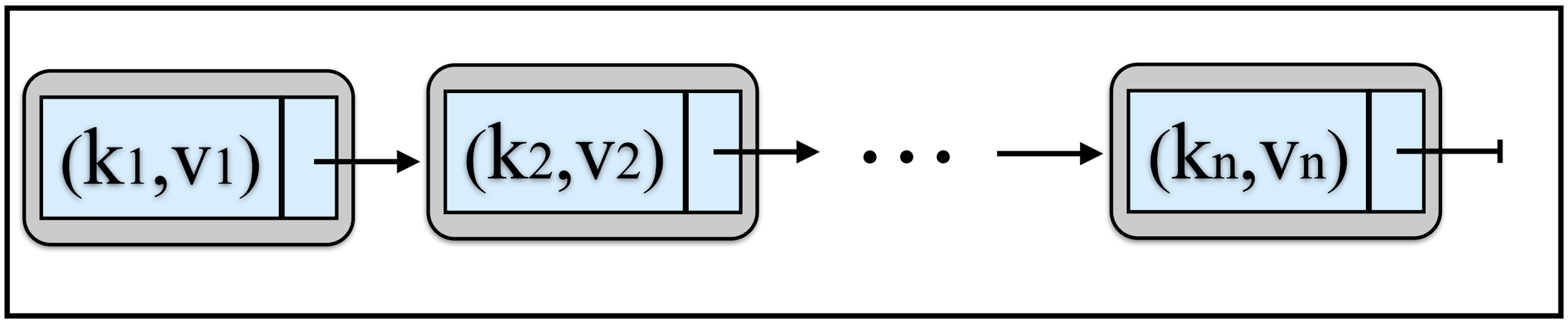
- Common to **combine** verified modules (BTree and List) to **build** new modules (B+ Tree)
- **Modularity**: reason about operations on each module **once**
- **Physically disjoint** composition impedes modularity
  - Define a new module (B+ Tree) and re-verify operations (e.g. adding/removing elements)
- **Overlapping** composition ☞ more ways to combine modules
  - More modular;
  - Better abstraction

Why CoLoSL?

- **Local** Proofs
  - Proof reuse - proofs done for the largest possible context
  - Simpler/More intuitive proofs
- **Subjective/ Overlapping** Shared Resources
  - More modular; better abstraction
- **Dynamic** Extension
  - No need to foresee all possible future behaviour

Concurrent List<K,V>: Possible Extension

Con_List([(k1, v1), (k2, v2), … , (kn, vn) =



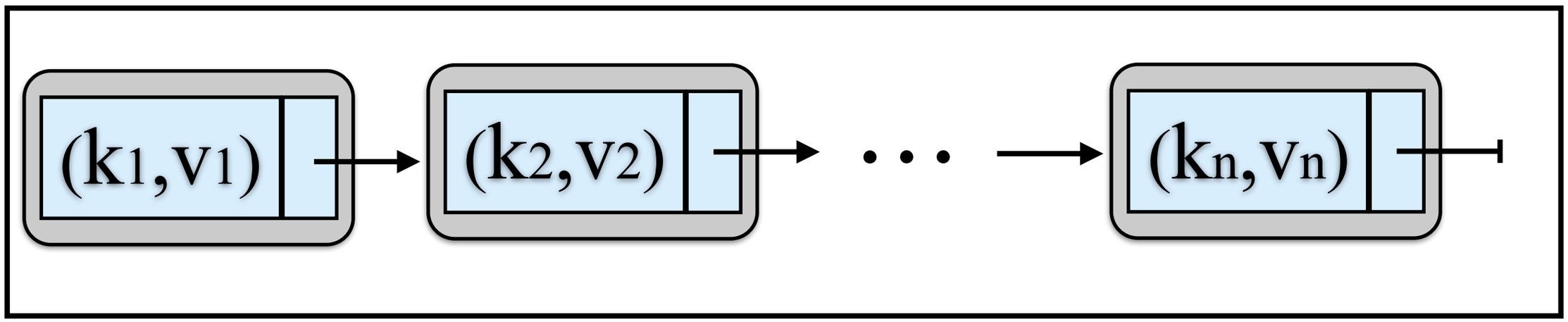
$I_{k1} = (k1, v1) \rightsquigarrow (k1, f1(v1))$

$I = I_{add} \cup I_{rem} \cup I_{k1}$

- **insertion/removal** involves pointer surgery
- Upon insertion specify how the value may be manipulated

Concurrent List<K,V>: Possible Extension

Con_List([(k1, v1), (k2, v2), … , (kn, vn) =



$I_{k1} = (k1, v1) \rightsquigarrow (k1, f1(v1))$

…

$I_{kn} = (kn, vn) \rightsquigarrow (kn, fn(vn))$

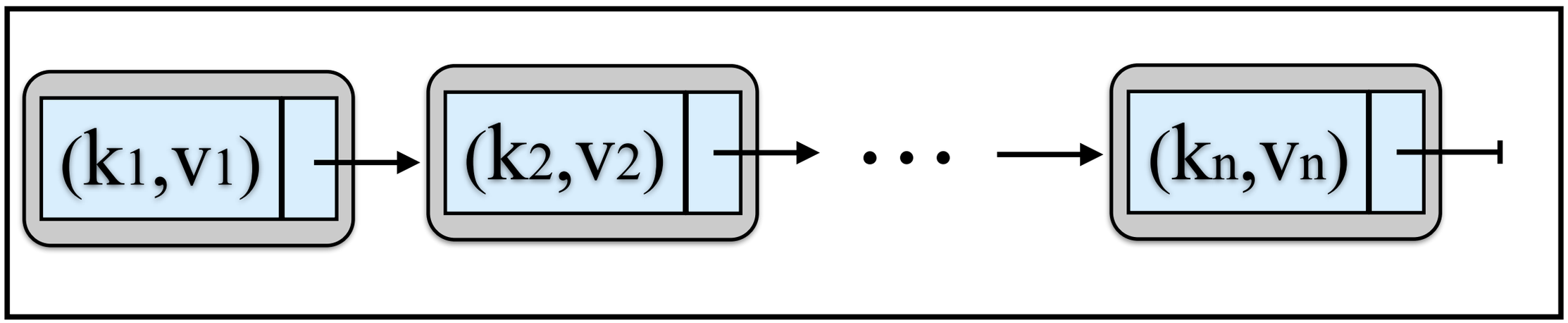
$I = I_{add} \cup I_{rem} \cup I_{k1} \cup \ldots \cup I_{kn}$

- **insertion/removal** involves pointer surgery
- Upon insertion specify how the value may be manipulated

Concurrent List<K,V>: Possible Extension

(Existing Approaches)

add(k', v')

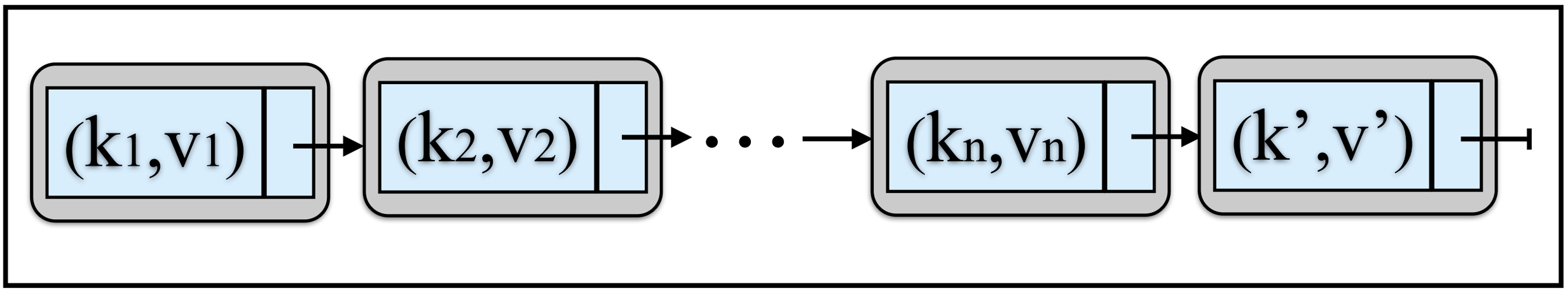


I

Concurrent List<K,V>: Possible Extension

(Existing Approaches)

add(k', v')



$I \cup I_{k'}$

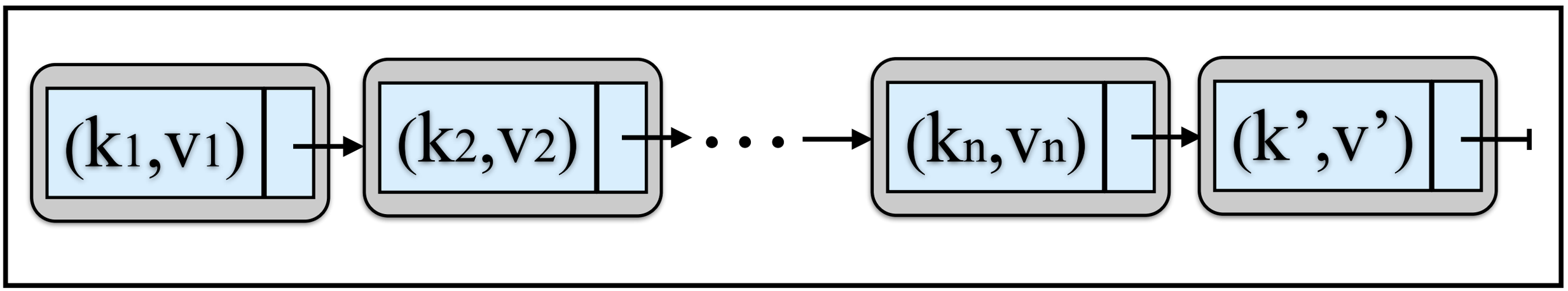
$I_{k'} = (k', v') \rightsquigarrow (k', f'(v'))$ ✗

- **Statically defined** interference
- Need to predict ALL future behaviour in advance

List<K,V>: Possible Extension

(CoLoSL)

add(k', v')



$I \cup I_{k'}$

$I_{k'} = (k', v') \rightsquigarrow (k', f'(v))$ ✔

- **Dynamic extension** :introduction of new interference on the fly

Conclusions

- From OG/Rg to CAP/TaDA
  - Huge steps towards compositionality/locality
  - Are we there yet? No!
- CoLoSL
  - Subjective/overlapping views
  - Dynamic framing on shared resource/interference
  - Dynamic extension
  - Are we there yet? Still No!
  - Better abstraction (atomicity), helping, spec

Questions?

Thank you for listening