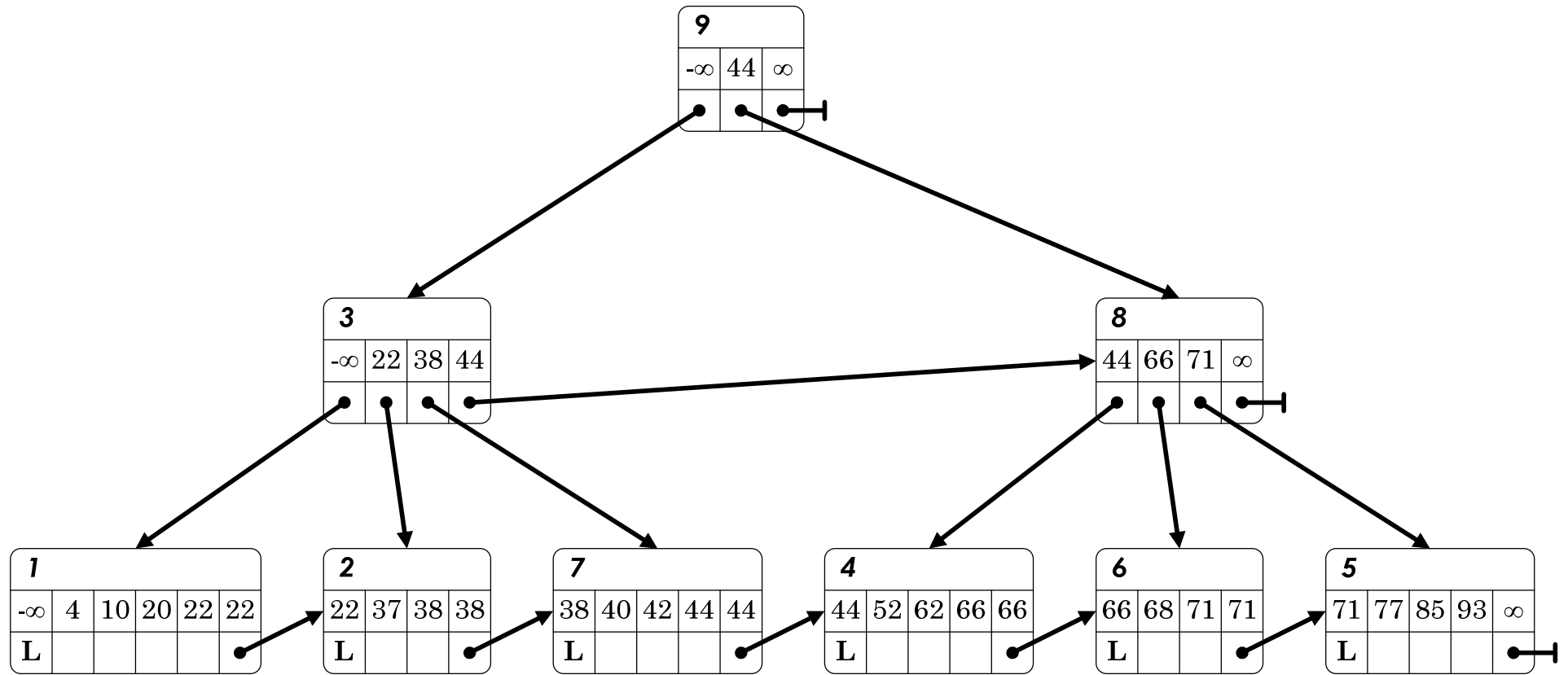
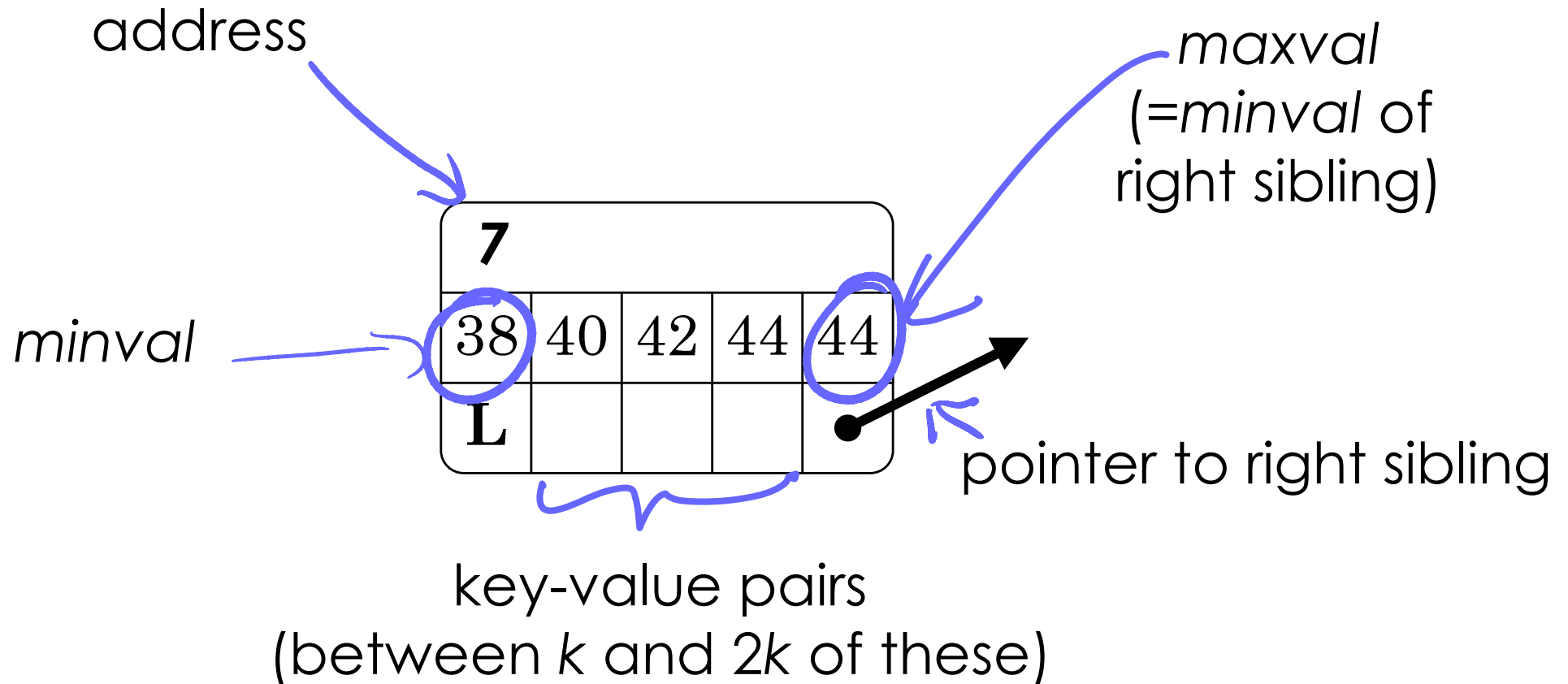




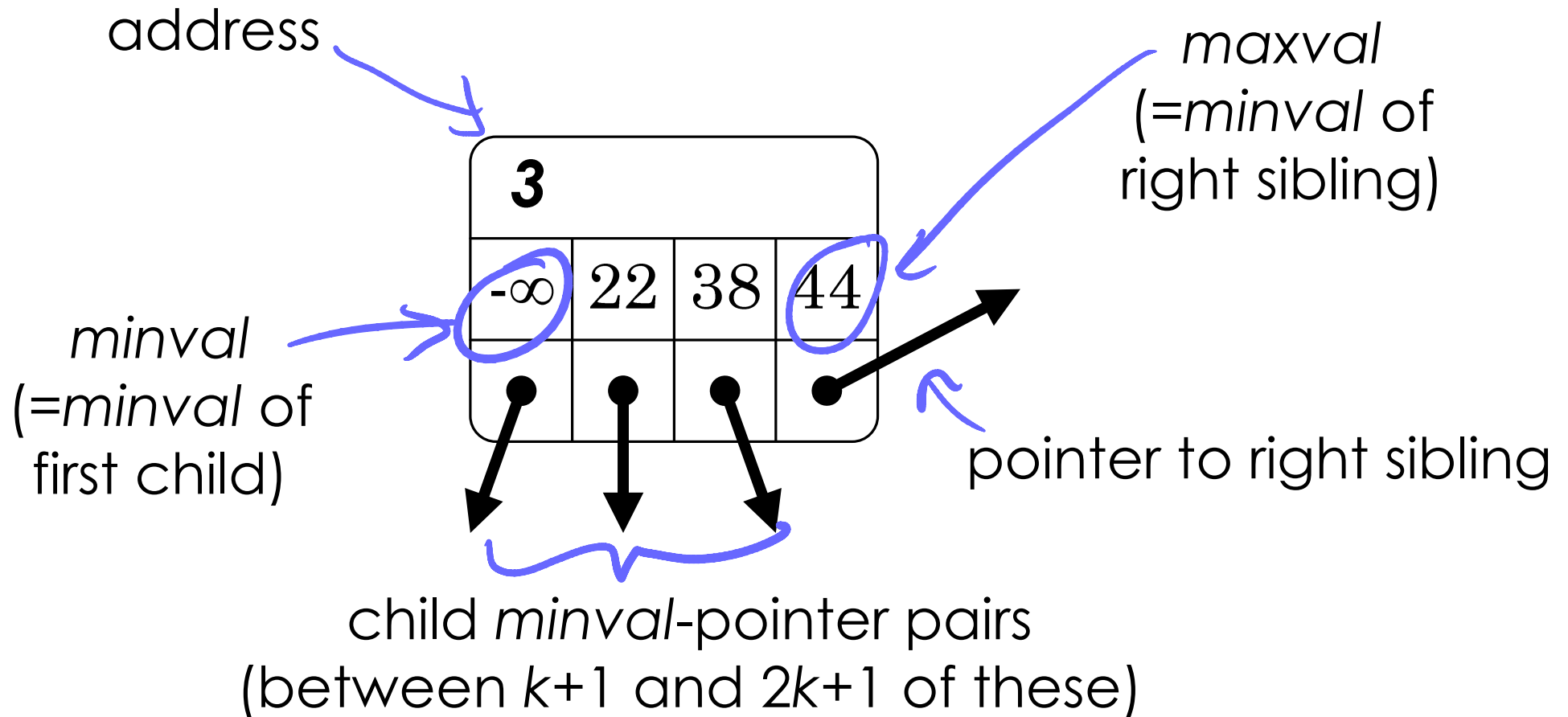
Concurrent B-Trees



Anatomy of a Leaf



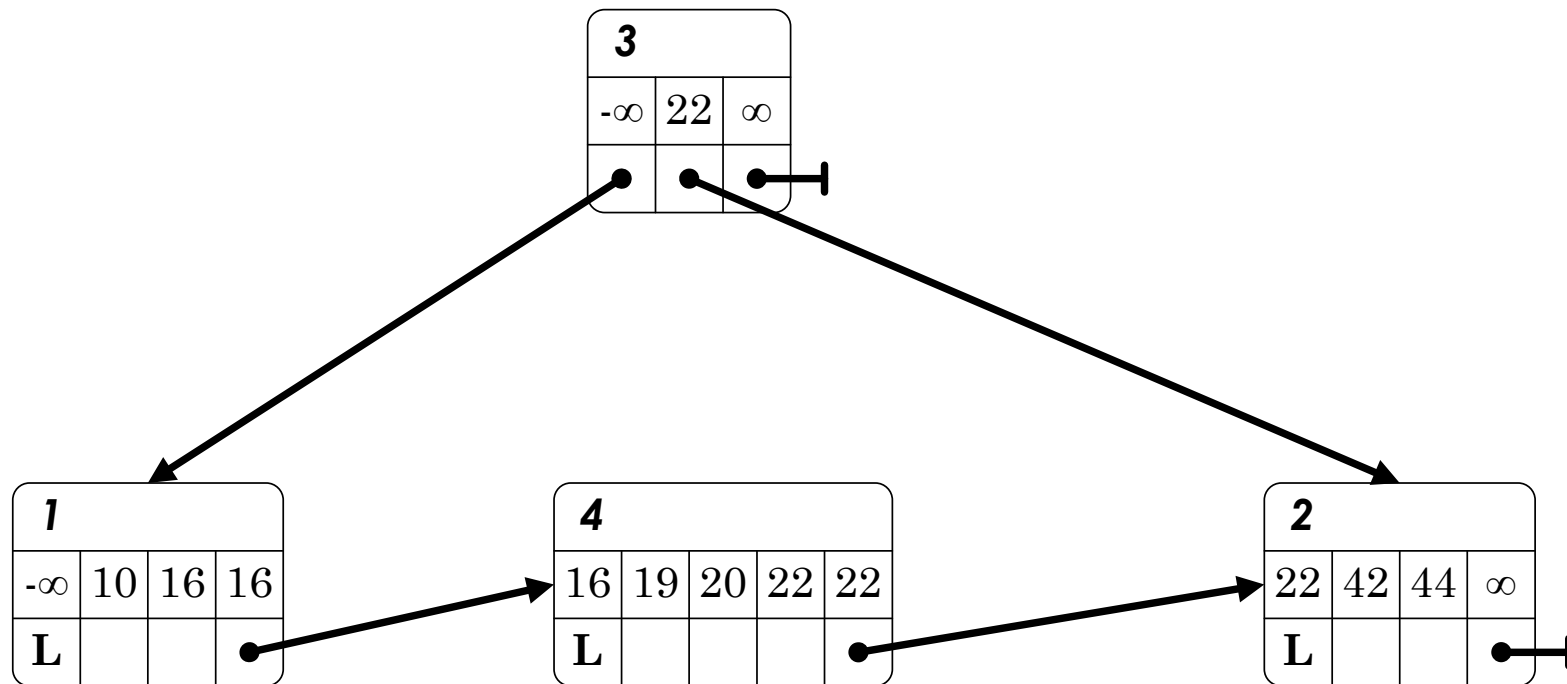
Anatomy of an Inner Node



Sagiv's Concurrent B-Trees

- Node accesses are atomic
- Reads acquire no locks
- Insertions and deletions acquire one lock at a time
- During insertions the B-Tree structure may be temporarily broken
- Compression phase repairs structure after deletions, and acquires up to three locks at once. (We ignore this for now.)

Broken B-Tree



Demo

Demo application available at:

`http://www.doc.ic.ac.uk/~td202/btree/`

Invariants

- The *minval* of a node is constant
- Corollary: the inner nodes always store the correct *minvals* of their linked-to children
- A leaf with *minval* u is reachable from a node with *minval* v if $v < u$
- Corollary: search always works

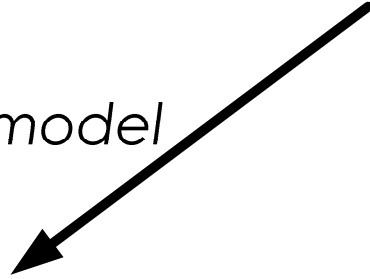


Separation Logic



Separation Logic

Change the model



Permissions, etc.

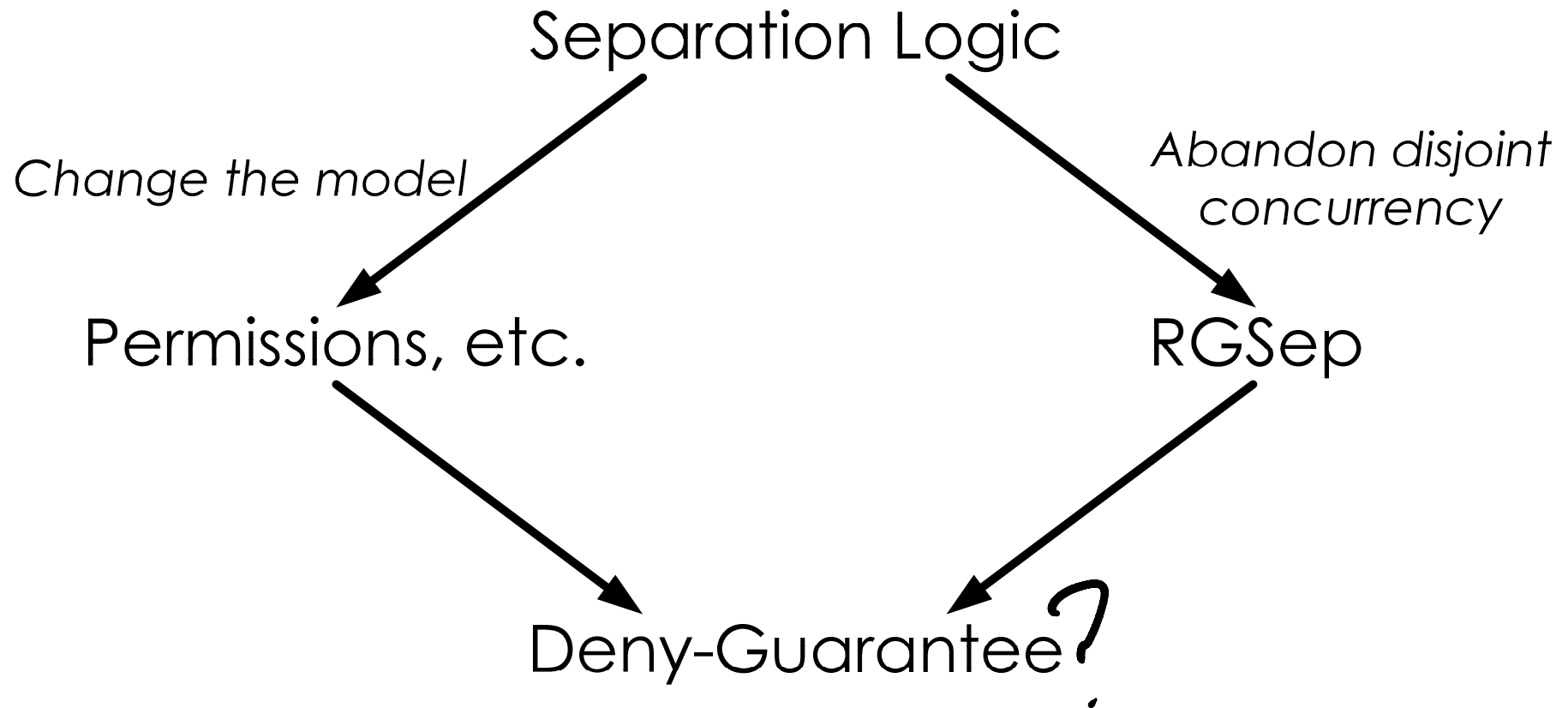
Separation Logic

Change the model

Permissions, etc.



*Abandon disjoint
concurrency*

RGSep



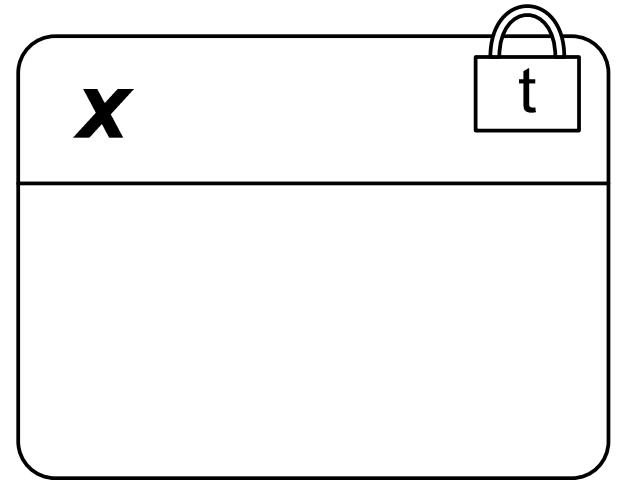
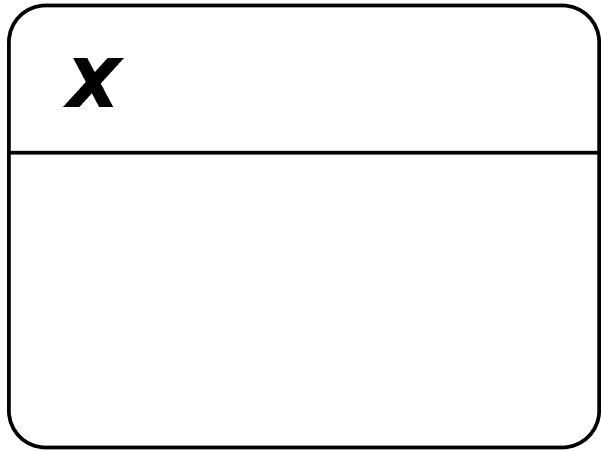
Actions

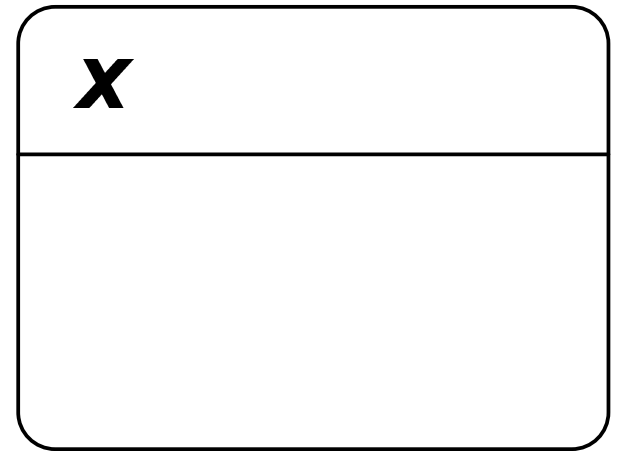
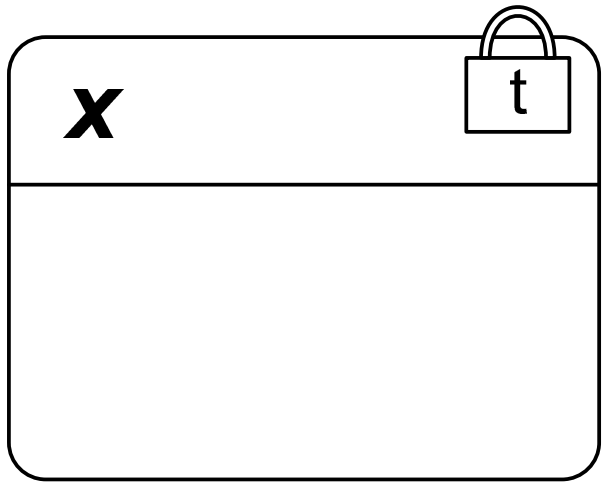
- Actions over-approximate the behaviours threads may exhibit on the shared state
 - Relation on shared state
 - Reflexive, transitive closure
- Guarantee – the actions of this thread
- Rely – the actions of all other threads
- Shared-state assertions should be stable under the Rely relation

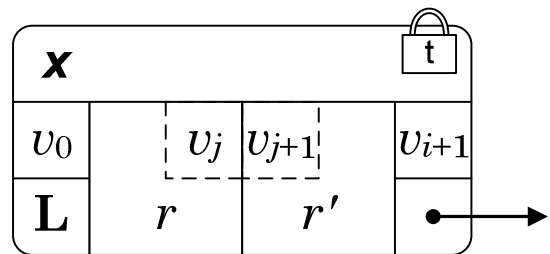

$$\begin{array}{l} \vdash C_1 \text{ sat } (p_1, R \cup G_2, G_1, q_1) \quad p_1 \text{ stable under } R \cup G_1 \\ \vdash C_2 \text{ sat } (p_2, R \cup G_1, G_2, q_2) \quad p_2 \text{ stable under } R \cup G_2 \\ \hline \vdash C_1 \| C_2 \text{ sat } (p_1 * p_2, R, G_1 \cup G_2, q_1 * q_2) \end{array}$$


B-Tree Actions

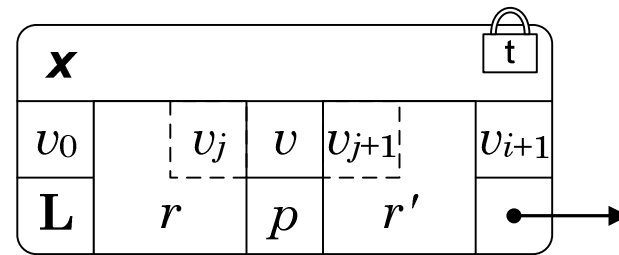
- Search does not alter shared state
 - Guarantee is the identity relation
- Insert alters shared state by
 - locking a node
 - unlocking a node
 - inserting into a leaf
 - inserting into an inner node
 - creating a new root





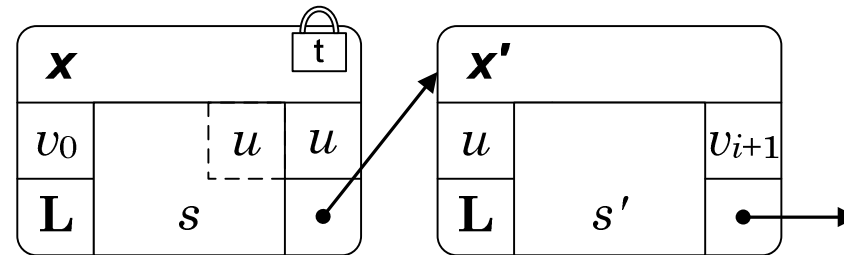


$$v_j < v < v_{j+1}$$

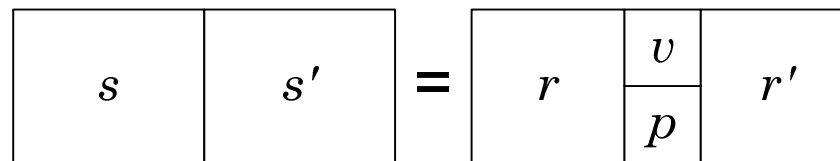


< 2k + 1 pairs

or

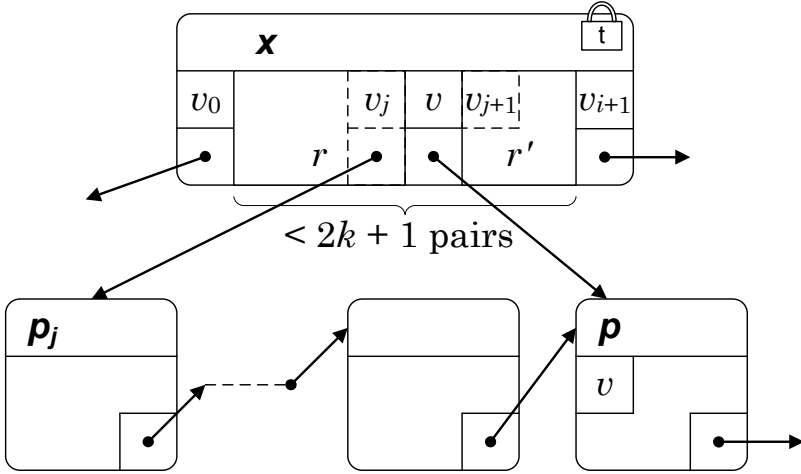
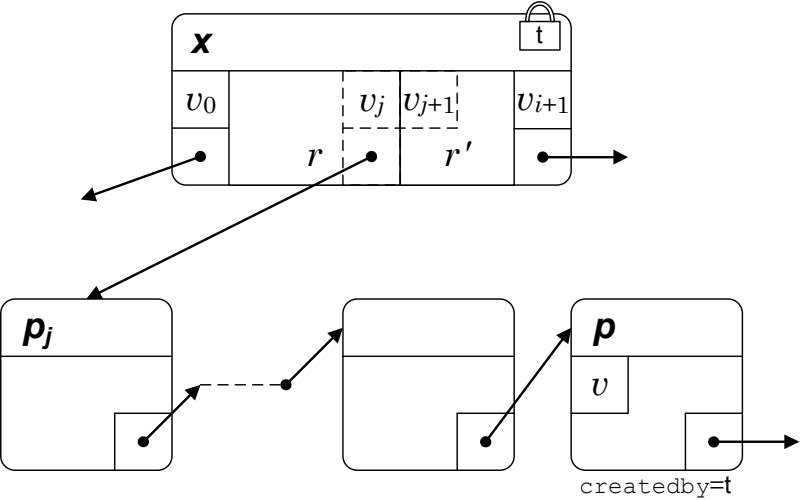


createdby=t

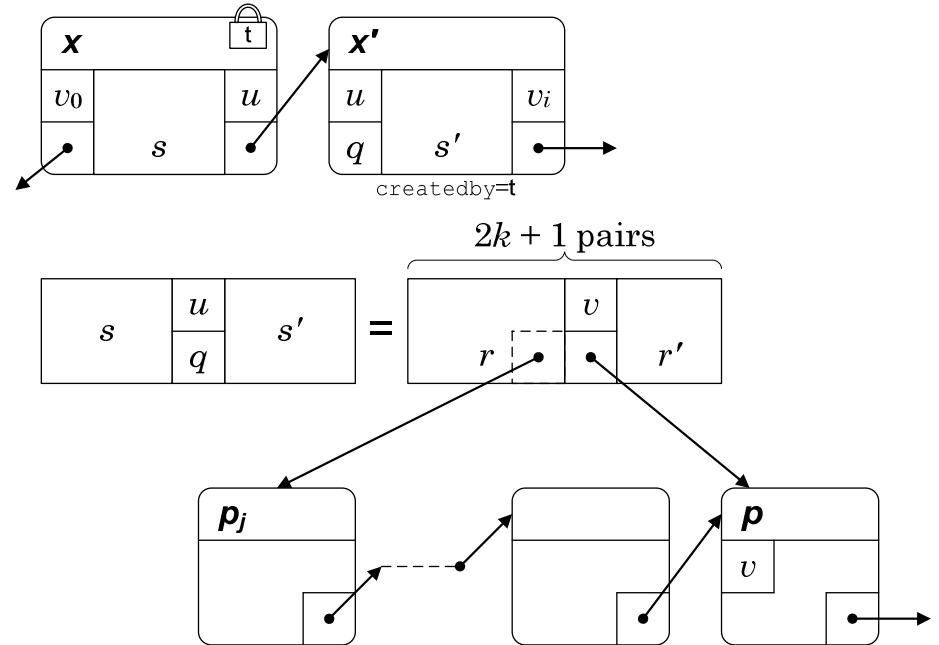


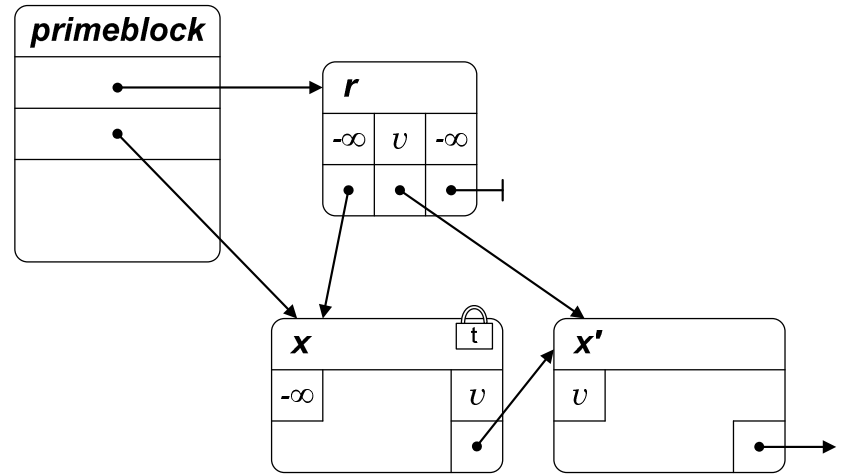
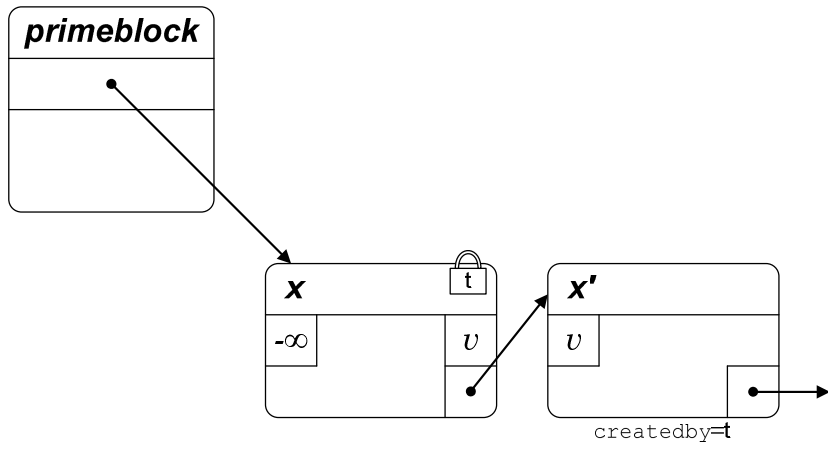
2k + 1 pairs

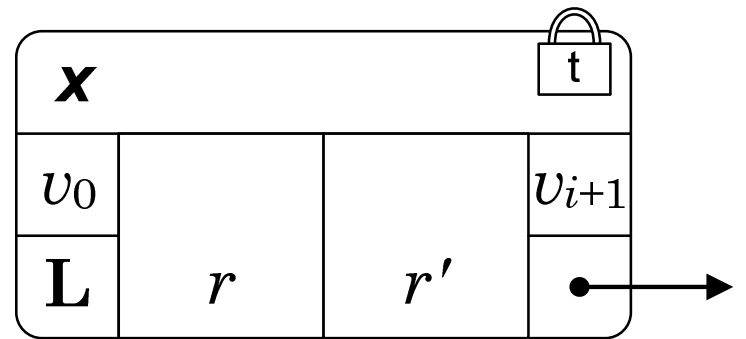
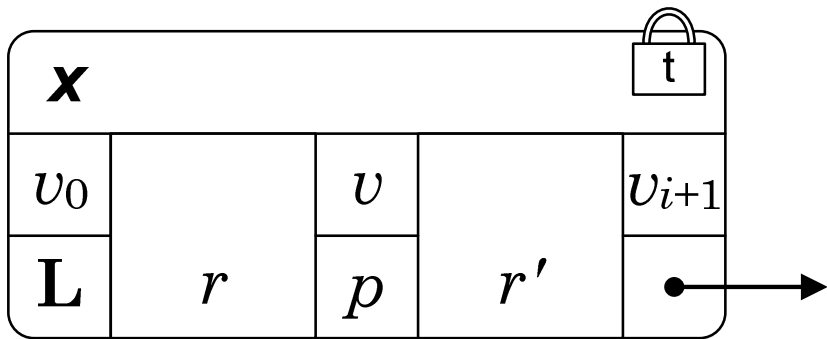
$$u_j < v < u_{j+1}$$



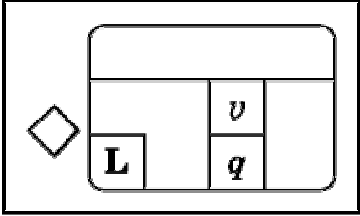
$$u_j < v < u_{j+1}$$

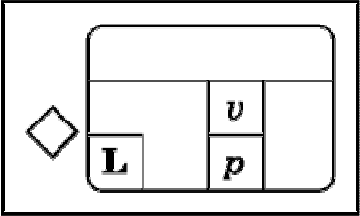






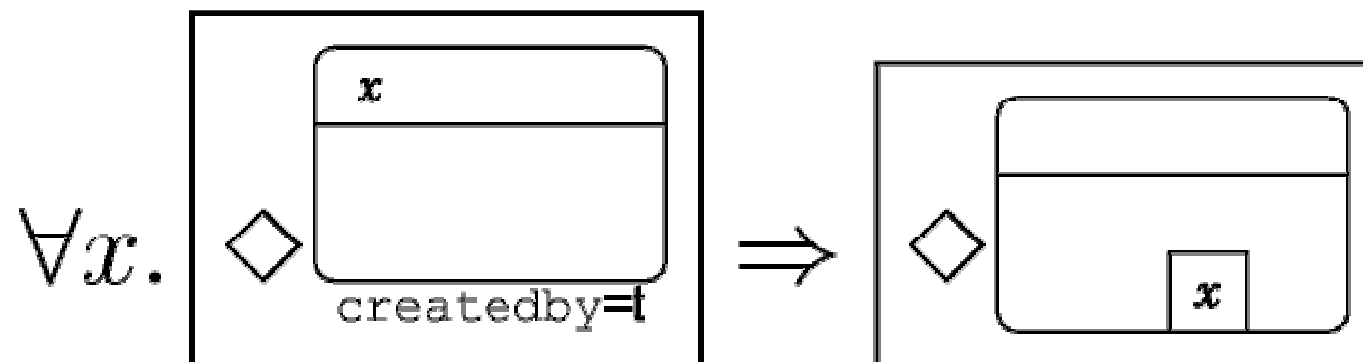
Specification of $insert(v,p)$

- Pre: $BTree \wedge \neg \exists q. \diamond$ 

The diagram shows a B-tree node structure. It consists of a large outer rectangle containing a smaller inner rectangle. The inner rectangle is divided into three sections: a left section containing the letter 'L', a middle section containing the variable 'v', and a right section containing the variable 'q'. A diamond symbol is positioned to the left of the inner rectangle, indicating a pointer to this node.
- Post: $BTree \wedge \diamond$ 

The diagram shows a B-tree node structure. It consists of a large outer rectangle containing a smaller inner rectangle. The inner rectangle is divided into three sections: a left section containing the letter 'L', a middle section containing the variable 'v', and a right section containing the variable 'p'. A diamond symbol is positioned to the left of the inner rectangle, indicating a pointer to this node.
- Rely: all actions, except inserting/deleting v (other threads)
- Guarantee: all actions, except deleting, or inserting anything other than v (this thread)

Fixing the B-Tree



High-level Specification

- We have given specs that work with the low-level details of the implementation
- What makes a good spec from the point of view of a user?
 - Simplicity
 - Compositionality
 - Completeness?

High Level

- What is a B-Tree providing at the high level?
- An index – a function from keys to values (or nothing)

$$\textit{Keys} \rightarrow \textit{Values} \cup \{\mathbf{Nothing}\}$$

- The commands only depend on a finite subset of the domain
 - We can use separation logic

High-level B-Tree Model

- We can consider partial B-Trees as a model of separation logic
 - Analogous to partial heaps
 - Partial B-Trees denote the information about a B-Tree for a finite set of keys
 - Partial B-Trees can be combined exactly when they talk about different sets of keys

$\{k \mapsto v\} \quad r = \text{search}(k) \quad \{k \mapsto v \wedge r = v\}$

$\{k \mapsto \mathbf{Nothing}\} \quad \text{insert}(k, v) \quad \{k \mapsto v\}$

$\{k \mapsto -\} \quad \text{delete}(k) \quad \{k \mapsto \mathbf{Nothing}\}$

$\{3 \Rightarrow \mathbf{Nothing} * 5 \Rightarrow \mathbf{Grapefruit}\}$

$r = search(5);$

$delete(5) \parallel insert(3, r)$

$\{3 \Rightarrow \mathbf{Grapefruit} * 5 \Rightarrow \mathbf{Nothing}\}$

What does $5 \mapsto \text{Grapefruit}$ mean?

- Implicit: we have a (broken) B-Tree
- Explicit: the key 5 is mapped to the value Grapefruit
- Locality: it does not matter what other keys are mapped to
- Disjoint Concurrency: the key 5 is not concurrently changed (but we do not care about other keys)

Action as Resource

- Interpreting these high-level resources at the low level requires us to view behaviours as part of the resource
- RGSep does not treat actions as resource
 - We cannot directly translate HL into LL
- How can/should we link the levels?
 - Deny-Guarantee?

$\{5 \mapsto \text{Grapefruit}\}$

$r = \text{search}(5);$
 $\text{insert}(5, \text{Pomegranate})$ \parallel $\text{delete}(5)$

$\{r = \text{nil} \Rightarrow 5 \mapsto \text{Pomegranate}\}$

Concluding Remarks

- We can use RGSep-style reasoning to show some correctness properties of concurrent B-Trees
 - Search in presence of Insert/Delete
 - Insert in presence of Insert/Delete
 - Compression...
- High-level view: action as resource?
- What about linearisability?