Notes on B*Trees

Thomas Dinsdale-Young, Philippa Gardner and Mark Wheelhouse

May 14, 2009

1 The Model

1.1 B^{Link}-Trees

Definition: Nodes

We define three types of node. Leaf nodes, which exist at the fringe of the tree and contain pointers to data, Intermediate nodes that contain pointers to other nodes, and Nodes where we don't know, or care, which kind of node we are deaing with.

$$\begin{aligned} \text{Leaf Node}: &\quad \textbf{leaf}(\mathtt{t}_{\mathrm{id}}, \mathtt{v}_0, \mathtt{s}, \mathtt{v}_{\mathtt{i+1}}, \mathtt{p}_{\mathtt{i+1}}) \\ \text{Intermediate Node}: &\quad \textbf{inner}(\mathtt{t}_{\mathrm{id}}, \mathtt{v}_0, \mathtt{p}_0, \mathtt{s}, \mathtt{v}_{\mathtt{i+1}}, \mathtt{p}_{\mathtt{i+1}}) \\ \text{Node}: &\quad \textbf{node}(\mathtt{t}_{\mathrm{id}}, \mathtt{v}_0, \mathtt{s}, \mathtt{v}_{\mathtt{i+1}}, \mathtt{p}_{\mathtt{i+1}}) \end{aligned}$$

The value $\mathtt{t}_{\mathrm{id}}$ tells us if the node has been locked by a thread, being set to 0 if the node is unlocked, and being set to the ID value of the locking thread otherwise. The value $\mathtt{v}_0$ is the lower bound on entries in the node and the value $\mathtt{v}_{\mathtt{i+1}}$ is the upper bound on entries in the node. $\mathtt{p}_0$ is the pointer to a subtree with values less than this node. Note that this is the null pointer in the leaf case. $\mathtt{p}_{\mathtt{i+1}}$ is the pointer to the next node at the current level of the tree and may be null if this is the rightmost node at the current level. $\mathtt{s}$ is a list of value-pointer pairs that allow for the quick search time of B^{link}-Trees.

Definition: Value-Pointer lists

$$\mathbf{s} = [\,] \mid (\mathbf{v}, \mathbf{p}){:}\mathbf{s}$$

where **v** is an integer key value and **p** is a pointer to a node or a data entry. In a leaf node all of the pointers in this list are to data entries, whilst in an inner node all of these pointers are to other tree nodes.

1.2 Program State

The program state consists of a *working heap h*, and a *variable store s*.

Definition: Heap

The heap h is a finite partial function that maps heap addresses to node data:

$$h : \textsc{Addrs} \rightharpoonup_{\text{fin}} \textsc{Nodes}$$

The heap is divided up such that each thread has its own private section of heap, and there is a single area of shared heap that contains the B*Tree.

Definition: Store

The store s is a set of finite partial functions that map integer variables $\text{Var}_{\mathbb{Z}} = \{i, j, ...\}$ to integers, address variables $\text{Var}_{\textsc{Addrs}} = \{\texttt{p}, \texttt{q}, ...\}$ to heap addresses, boolean variables $\text{Var}_{\mathbb{B}} = \{\texttt{b}, ...\}$ to boolean values and node data variables $\text{Var}_{\textsc{Nodes}} = \{\texttt{A}, \texttt{B}, ...\}$ to node data:

$$s : (\text{Var}_{\mathbb{Z}} \rightharpoonup_{\text{fin}} \mathbb{Z}) \times (\text{Var}_{\textsc{Addrs}} \rightharpoonup_{\text{fin}} \textsc{Addrs} \cup \{\textbf{null}\}) \times (\text{Var}_{\mathbb{B}} \rightharpoonup_{\text{fin}} \mathbb{B}) \times (\text{Var}_{\textsc{Nodes}} \rightharpoonup_{\text{fin}} \textsc{Nodes})$$

2 Basic Commands

2.1 Heap Commands

`A := get(x)`	*retuns node data of node x*
`put(A, x)`	*overwrites node x with node data A*
`lock(x)`	*acquires lock of node x*
`unlock(x)`	*releases lock of node x*
`q := new()`	*creates a node of size 2k and returns pointer to new node*

2.2 Store Commands

`r := root(PB)`	*returns the top pointer from the primeBlock info stored in PB*
`p := next(A, v)`	*returns the pointer from A to follow to find v*
`p := lookUp(A, v)`	*returns the data pointer for v, or nil*
`removePair(A, v)`	*removes pair (v,p) from A*
`addPair(A, v, p)`	*adds pair (v,p) to A in correct place*
`v := highValue(A)`	*returns high value of A*
`v := lowValue(A)`	*returns low value of A*
`b := isIn(A, v)`	*returns true if v is a value in A, false otherwise*
`B := rearrange(A, v, p, q)`	*split A into A and B with $k \leq i \leq 2k$*

2.3 Language Commands

$\mathbf{P}_1; \mathbf{P}_2$	*Sequencing of commands*
`if(B){`$\mathbf{P}_1$`}else{`$\mathbf{P}_2$`}`	*if B = true then run P_1 otherwise run P_2*
`while(B){`$\mathbf{P}$`}`	*while B = true run P*

2.4 Command Axioms

3 Algorithms

3.1 search

The search algorithm looks through the B*Tree trying to find the data entry associated with a certain key value val. When it gets to the node that would contain that key value, i.e. the node where $v_0 \leq val \leq v_{i+1}$, the algorithm tests to see if the key value is actually present in that node. If the key value is present then the algorithm returns a pointer to the data entry associated with that key value. If the key value is not present then the algorithm simply returns the **null** pointer.

```
r := search(val) ≜   {
        v := val;
        movedown;
        moveright;
        if(isIn(A, v)){
            r := lookUp(A, v)
        }
        else{
            r := null
        }
    }
```

where

```
movedown ≜   {
        PB := get(primeBlock)
        current := root(PB)
        A := get(current)
        while(type(A) ≠ leaf){
            current := next(A, v)
            A := get(current)
        }
    }
```

and

```
moveright ≜   {
        while(v > highValue(A)){
            current := next(A, v);
            A := get(current)
        }
    }
```

3.2 `insert`

The `insert` algorithm takes a value-pointer pair and inserts it into the B*Tree such that the search structure of the B*Tree is preserved.

3.3 `delete`

The `delete` algorithm searches through the B*Tree looking for the data entry associated with a certain key value. Once it finds the node that would contain the key value, ie.e the node where $v_0 \leq val \leq v_{i+1}$, the algorithm tests to see if the key value is actually present in that node. If the key value is present then the algorithm removes that key-value pair from the node and returns true. If the key value is not present then the algorithm simply returns false and makes no changes to the node. This operation may leave a node with less than **k** entries.

```
bool := delete(val) ≜   {
                          v := val;
                          movedown;
                          moveright;
                           if(isIn(A, v)){
                           locking???
                               removePair(A, v;
                           )    bool := true
                           }
                           else{
                               bool := false
                           }
                        }
```

3.4 `compress`

The `compress` algorithm works through (a level of?) the B*Tree and identifies nodes which have less than **k** entries. It compresses such nodes together so that the tree is now optimal for search times.

4 Rely Guarantee

4.1 Actions

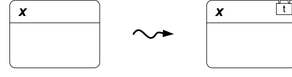
There are 5 different actions (at this stage) that other threads can perform that will modify the shared state. These actions are,

LOCK
UNOCK
DELETE
INSERT@LEAF
INSERT@INNER
NEWROOT

4.2 LOCK

This action allows another thread to lock a node that is currently unlocked.

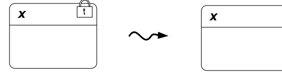
$$\mathtt{t} \in \mathrm{T}\backslash\{\mathtt{t}_{\mathrm{id}}\} \wedge \mathrm{unlocked}(\mathtt{x}) \rightsquigarrow \mathrm{locked}(\mathtt{x}, \mathtt{t})$$



4.3 UNLOCK

This action allows another thread to unlock a node which that same thread had previosuly locked.

$$\mathtt{t} \in \mathrm{T}\backslash\{\mathtt{t}_{\mathrm{id}}\} \wedge \mathrm{locked}(\mathtt{x}, \mathtt{t}) \rightsquigarrow \mathrm{unlocked}(\mathtt{x})$$



4.4 DELETE

This action allows another thread to remove a value-pointer pair from a leaf node if it holds the lock on that node. This action may leave a node with less than **k** entries.

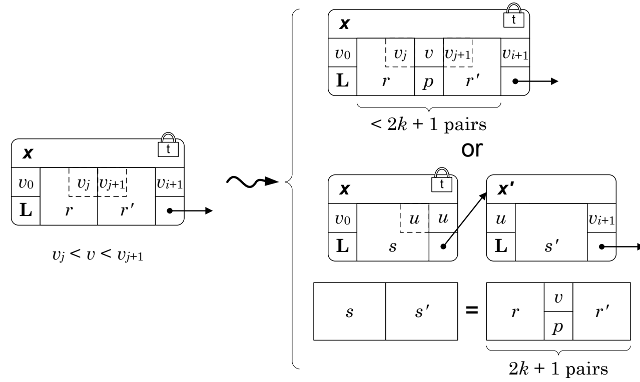
$$\begin{gathered}\mathtt{t} \in \mathrm{T}\backslash\{\mathtt{t}_{\mathrm{id}}\} \wedge \mathrm{leaf}(\mathtt{t}, \mathtt{v}_0, \mathtt{r} + (\mathtt{v}, \mathtt{p}) + \mathtt{r}', \mathtt{v}_{\mathtt{i}+1}, \mathtt{p}_{\mathtt{i}+1}) \\ \rightsquigarrow \\ \mathrm{leaf}(\mathtt{t}, \mathtt{v}_0, \mathtt{r} + \mathtt{r}', \mathtt{v}_{\mathtt{i}+1}, \mathtt{p}_{\mathtt{i}+1})\end{gathered}$$

...diagram here...

4.5 INSERT@LEAF

This action allows another thread to add a value-pointer pair to a leaf level node if it holds the lock on that node.

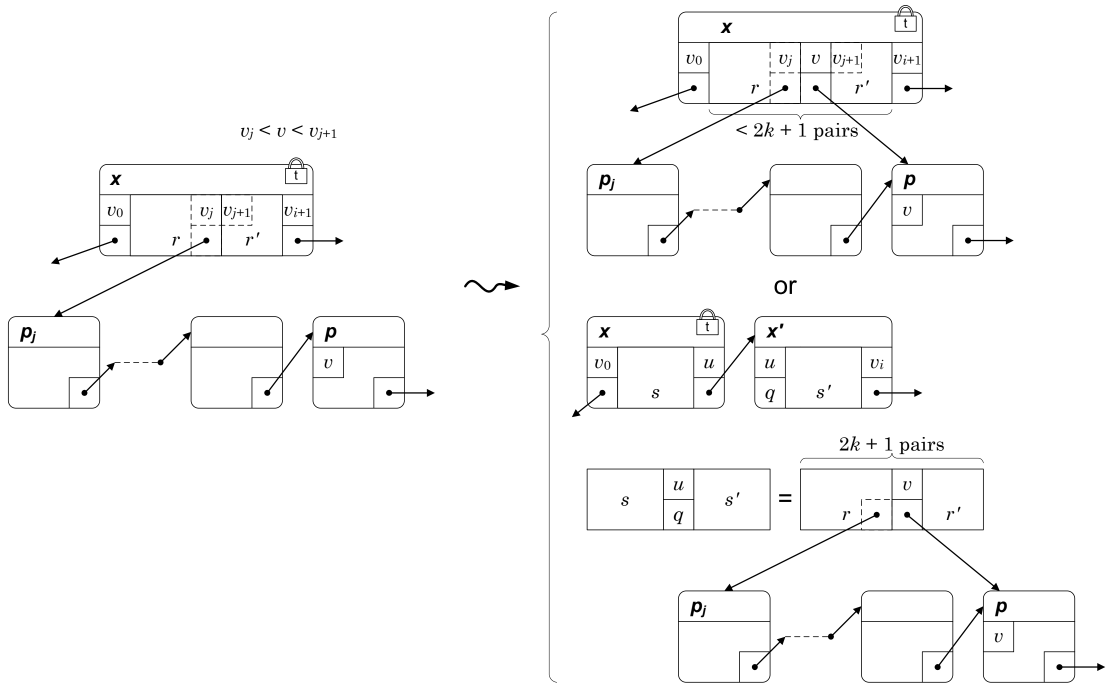
$$\begin{gathered}\mathtt{t} \in \mathrm{T}\backslash\{\mathtt{t}_{\mathrm{id}}\} \wedge \mathrm{lastVal}(\mathtt{r}) < \mathtt{v} < \mathrm{firstVal}(\mathtt{r}') \\ \wedge\ \mathtt{x} \mapsto \mathrm{leaf}(\mathtt{t}, \mathtt{v}_0, \mathtt{r} + \mathtt{r}', \mathtt{v}_{\mathtt{i}+1}, \mathtt{p}_{\mathtt{i}+1}) \\ \rightsquigarrow \\ |\mathtt{r} + \mathtt{r}'| < 2\mathbf{k} \wedge \mathtt{x} \mapsto \mathrm{leaf}(\mathtt{t}, \mathtt{v}_0, \mathtt{r} + (\mathtt{v}, \mathtt{p}) + \mathtt{r}', \mathtt{v}_{\mathtt{i}+1}, \mathtt{p}_{\mathtt{i}+1}) \\ \vee \\ |\mathtt{r} + \mathtt{r}'| = 2\mathbf{k} \wedge \mathtt{u} = \mathrm{lastVal}(\mathtt{s}) \wedge \mathtt{s} + \mathtt{s}' = \mathtt{r} + (\mathtt{v}, \mathtt{p}) + \mathtt{r}' \\ \wedge\ \mathtt{x} \mapsto \mathrm{leaf}(\mathtt{t}, \mathtt{v}_0, \mathtt{s}, \mathtt{u}, \mathtt{x}') * \mathtt{x}' \mapsto \mathrm{leaf}(0, \mathtt{u}, \mathtt{s}', \mathtt{v}_{\mathtt{i}+1}, \mathtt{p}_{\mathtt{i}+1}) \wedge \mathrm{created}(\mathtt{t}, \mathtt{x}')\end{gathered}$$



4.6 INSERT@INNER

This action allows another thread to add a value-pointer pair to an inner node if it holds the lock on that node.

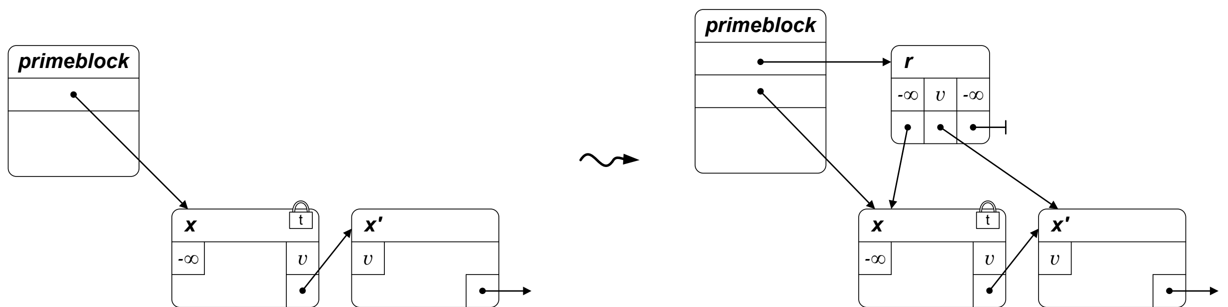
$$
\begin{gathered}
\mathtt{t} \in \mathrm{T}\backslash\{\mathtt{t}_{\mathrm{id}}\} \wedge \mathrm{lastVal}(\mathtt{r}) < \mathtt{v} < \mathrm{firstVal}(\mathtt{r}') \\
\wedge\ \mathtt{x} \mapsto \mathrm{inner}(\mathtt{t}, \mathtt{v}_0, \mathtt{p}_0, \mathtt{r} + \mathtt{r}', \mathtt{v}_{\mathtt{i+1}}, \mathtt{p}_{\mathtt{i+1}}) \\
*\ \mathrm{forest}(\mathtt{p}_{\mathtt{j}}, \mathtt{p}) \wedge \mathtt{F} * \mathtt{p} \mapsto \mathrm{node}(_, \mathtt{v}, _, _, _) \wedge \mathtt{P} \wedge \mathrm{created}(\mathtt{t}, \mathtt{p}) \\
\rightsquigarrow \\
|\mathtt{r} + \mathtt{r}'| < 2\mathbf{k} \wedge \mathtt{x} \mapsto \mathrm{inner}(\mathtt{t}, \mathtt{v}_0, \mathtt{p}_0, \mathtt{r} + (\mathtt{v}, \mathtt{p}) + \mathtt{r}', \mathtt{v}_{\mathtt{i+1}}, \mathtt{p}_{\mathtt{i+1}}) * \mathtt{F} * \mathtt{P} \\
\vee \\
|\mathtt{r} + \mathtt{r}'| = 2\mathbf{k} \wedge \mathtt{s} + (\mathtt{u}, \mathtt{q}) + \mathtt{s}' = \mathtt{r} + (\mathtt{v}, \mathtt{p}) + \mathtt{r}' \wedge \mathrm{created}(\mathtt{t}, \mathtt{x}') \\
\wedge\ \mathtt{x} \mapsto \mathrm{inner}(\mathtt{t}, \mathtt{v}_0, \mathtt{p}_0, \mathtt{s}, \mathtt{u}, \mathtt{x}') * \mathtt{x}' \mapsto \mathrm{inner}(0, \mathtt{u}, \mathtt{q}, \mathtt{s}', \mathtt{v}_{\mathtt{i+1}}, \mathtt{p}_{\mathtt{i+1}}) * \mathtt{F} * \mathtt{P}
\end{gathered}
$$



4.7 NEWROOT

This action allows another thread to create a new root for the B*Tree if it holds the lock of the current root node.

$$
\begin{gathered}
\mathtt{t} \in \mathrm{T} \backslash \{\mathrm{t_{id}}\} \wedge \textbf{primeBlock} \mapsto \mathtt{x} : \mathtt{xs} \\
* \, \mathtt{x} \mapsto \mathrm{node}(\mathtt{t}, -\infty, _, \mathtt{v}, \mathtt{x}') \wedge \mathtt{X} * \mathtt{x}' \mapsto \mathrm{node}(_, \mathtt{v}, _, _, _) \wedge \mathtt{X}' \wedge \mathrm{created}(\mathtt{t}, \mathtt{x}') \\
\rightsquigarrow \\
\textbf{primeBlock} \mapsto \mathtt{r} : \mathtt{x} : \mathtt{xs} * \mathtt{X} * \mathtt{X}' * \mathtt{r} \mapsto \mathrm{inner}(0, -\infty, \mathtt{x}, (\mathtt{v}, \mathtt{x}'), \infty, \textbf{null})
\end{gathered}
$$



5 Correctness of Algorithms

Now that we have a notion of the interference that can be caused by other threads manipulating the B*Tree, we can look at the correctness of the algorithms provided in section 3. (So far we only have the actions that correspond to other treads performing search or insert actions, we will need to extend our action set to also consider the effects of delete and compress actions.)

We will provide specifications in a rely-guarantee style, which is of the form,

$$\mathbb{C} \models \{P\}, R, G, \{Q\}$$

This means that a program $\mathbb{C}$ satisfies a specification if the program state satisfies P before the program initiates, the program state satisfies Q immediately after the program terminiates (if it terminates), the program operates normally under the effects on the shared state specificied in the Rely Set R and the program has the effects on the shared state specified by the Guarantee Set G. When we provide assertions at program points, these assertions must be stable under all of the actions allowed by the Rely Set R.

5.1 Correctness of `search`

To prove the correctness of search we wish to show that,

$$\mathtt{r} := \mathtt{search}(\mathtt{val}) \models \{\text{BTree}\}, R', \emptyset, \left\{ \begin{array}{c} (\text{BTree} \wedge \Diamond(\mathtt{val}, \mathtt{p}) \wedge (\mathtt{r} = \mathtt{p})) \\ \vee \\ (\text{BTree} \wedge \neg\Diamond(\mathtt{Val}, \mathtt{p}) \wedge (\mathtt{r} = \mathbf{null})) \end{array} \right\}$$

where BTree simply states that the shared state contains a 'well-formed' B*Tree. The Rely-Set R' is the set of actions described in section 4.1 without any actions that insert (or delete) at the key value `val` of the search, as these would lead to a non-deterministic race for that data.

Some of the mid points from our proof (a sketch outline)

$$\exists \mathtt{p}.\ \mathbf{primeBlock} \mapsto \mathtt{p} + \mathtt{PB}$$

$$(\mathtt{current} = \mathtt{n}) \wedge \mathtt{n} \mapsto \mathtt{N}\Diamond\mathtt{current}$$

$$(\mathtt{A} = \mathtt{N}) \wedge \text{niceNode}(\mathtt{N}, -\infty, ?)$$

$$\mathit{loopInv} \wedge (\mathtt{type}(\mathtt{A}) \neq \text{leaf})$$

$$\mathit{loopInv} \wedge (\mathtt{type}(\mathtt{A}) = \text{leaf})$$

where the loop invariant is something like,

$$(\mathtt{A} = \mathtt{N}) \wedge (\mathtt{v}' = \mathtt{lowValue}(\mathtt{A})) \wedge \text{niceNode}(\mathtt{N}, \mathtt{v}', ?) \wedge (\mathtt{v}' < \mathtt{v})$$