B-Trees

Mark-Jason Dominus

November 12, 1997

Introduction

B-Trees are often mentioned as a good way to store and retrieve data, especially to and from disk. They're efficient to use and easy to program, and they're often mentioned but not-so often discussed. They show up all over the place: In the Berkeley `DB_File` package, which is (semi-) standard with Perl, and in large high-performance databases. For example, B-Trees are the technology that underlies IBM's well-known ISAM files.

In this article we'll see how they work. First, we'll have a review of how we can use ordinary binary trees to store data, and we'll see the potential disadvantages of that. Then we'll look over the B-Tree algorithm in the abstract, and see how it corrects the flaws of binary trees. Then we'll look at the code of a module that implements the B-Tree algorithms and see how B-Trees can be implemented in Perl.

Binary trees and B-Trees are structures for storing and retrieving data, just like hashes. In fact, from the outside, they look exactly like hashes. Each contains a collection of keys, each of which is associated with a particular datum. In order to be useful, the data structure should allow you to look up the datum associated with a particular key very quickly. Binary trees and B-trees both provide that function.

You're supposed to learn about binary trees early in your programming career, so the part of this article about 'binary trees' is supposed to be a review; if you don't know about binary trees yet, you might want to check a book on data structures, such as the one in this article's bibliography first, or this article might not make any sense.

The Problem with Binary Trees

A binary tree has is made of *tree nodes*. Each node has a key-data pair, which from now on we'll call a *KDP*, and each node also has between zero and two *children*, which are also tree nodes. See figure 1.

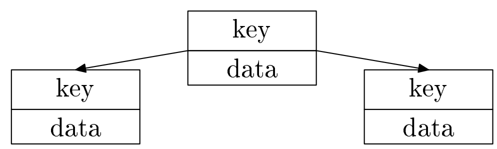


Figure 1: Binary tree node, with children

The children might have other children in turn, so the entire tree might be very big. From now on (until we actually do the implementation) we're going to forget about the data; just imagine it's stored in the nodes 'along with' the keys. The searching algorithms we'll see only look at the keys, never at the data, so we can forget about it for a while.

Figure 2 shows a little tree.

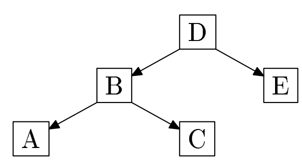


Figure 2: A little tree

In the little tree of figure 2, A and C are children of B, and we also say that B is the parent of A and C. Nodes with no children, such as A, C, and E, are called *leaves*. The topmost node, which has no parent, is called the *root*—node D is the root here. Notice that if you ignore D and E, B is the root of its own little tree of just A, B, and C. This mini-tree is called the *left subtree of* D. The right subtree of D is the even smaller tree that contains just E and nothing else.

Keys must have an ordering defined on them, so that you can say for sure when one is 'less than' another. The exact ordering doesn't matter much, so for this discussion we'll suppose that the keys are strings and that the ordering is the usual string ordering defined by Perl's string comparison operators such as `lt`. So from now on when I say that one key is 'less than' another, I mean the 'less than' defined by `lt`.

The essential property of the tree is this: If a certain node contains a key, say K, then the left subtree of that node contains only keys that are less than

K, and the right subtree contains only keys that are greater than K.

This means you can quickly search the tree for a particular key. Here's how: Suppose you're looking for K. Start at the root; if the key in the root is K, you've found it. If K is less than the key in the root, move down to the root of the left subtree and repeat the process; otherwise, move down to the right. Typically, each time you move down to a subtree, the number of nodes left below where you are will decrease by about half, so you'll quickly find the node you want.

Normally, if you search for a key and you don't find it, you've ended at a leaf, and you build a new node with your new key and data and you attach it as a subtree of this leaf, so that your tree grows. Let's see an example, where the keys are B, D, C, A, E, delivered in that order. Figure 3 shows each stage of the construction.

First we build a B node to be the root. Then we attach D as the right subtree of B, because D is greater than B. Then we attach C as the left subtree of D. (C comes after B but before D.) Then we attach A as the left subtree of B. (A comes before B.) Then we attach E as the right subtree of D. (E comes after B and after D.)

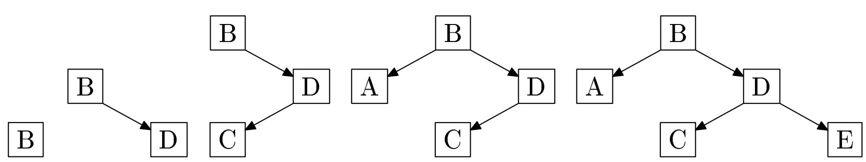


Figure 3: Constructing a binary tree

The average depth of a key here is 2.2, which means that to look up a key to see if it is in this tree, you expect to do about 2.2 comparisons with the keys in the tree.

The problem with binary trees is that sometimes if you're not careful you can build them wrong and then they're unbalanced. What does this mean? You'd like the tree to be pretty shallow, because that means you'll never have to do many comparisons to find out if a key is there or not. When the tree is shallow and bushy, no node's right subtree is much deeper than its left subtree. When this is true, the tree is said to be 'balanced'.

Let's see what happens when the tree isn't balanced. If, in the example above, we had encountered the keys A, B, C, D, E in alphabetical order, then instead of growing the tree we saw before, of maximum depth 3 and average depth 2.2, we would have the 'tree' of figure 4, of maximum depth 5 and average depth 3. Really it's more like a vine than a tree.

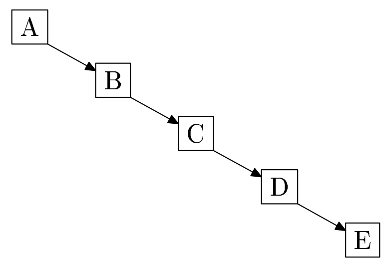


Figure 4: When binary trees go wrong

The average depth here is 3, which is 36% worse than the example tree we saw earlier. If you get unlucky in the way you build your tree, and you get a vine or a tall spindly thing that didn't get enough light, you can pay a stiff performance penalty. In this case when you search the vine you are making 36% more comparisons than you would be if the tree were the nice bushy one we saw before. For larger trees, the costs of vininess are even worse.

B-Trees are Always Balanced

B-trees avoid this vine problem by incorporating two improvements over ordinary binary trees.

First, the nodes contain many keys instead of only one, so the trees are not binary. Instead of nodes like the one in figure 5 we use nodes like the one in figure 6:

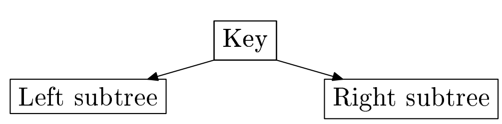


Figure 5: A Typical binary tree node

The nodes are analagous to the nodes in a binary tree. The keys in a node obey the ordering

$$\text{Key}_0 < \text{Key}_1 < \text{Key}_2 < \ldots < \text{Key}_{N-1}$$

so the keys in a node are always in sorted order. Furthermore,

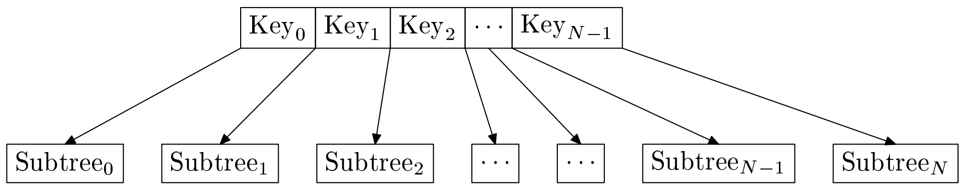


Figure 6: A Typical B-tree node

All the keys in Subtree$_0$	are less	than Key$_0$.	
All the keys in Subtree$_1$	are greater	than Key$_0$	and less than Key$_1$.
All the keys in Subtree$_2$	are greater	than Key$_1$	and less than Key$_2$.
...			
All the keys in Subtree$_{N-1}$	are greater	than Key$_{N-2}$	and less than Key$_{N-1}$.
All the keys in Subtree$_N$	are greater	than Key$_{N-1}$.	

To search the tree, you start at the root node, and you do a binary search on the keys in the node. (This is the kind of search you use when you look something up in the encyclopedia or the telphone book; you can do this because the keys in the node are in sorted order, like in the encyclopedia or the telephone book.)

If the key you want is in the node, you're done.

If the key you want isn't in the node, then you have found a key that is larger than the one you want and a key that is smaller, so you move down to the appropriate subtree in between and continue. Because the fan-out is greater, a B-tree is not as deep as a binary tree. This is the 'moving down' portion of the B-tree algorithm.

The real improvement, however, comes when you want to insert a new KDP (key-data pair) into a tree. When we insert, we have a trick that prevents the tree from getting too deep too quickly, and from turning into a vine. This is the 'moving up' portion of the algorithm:

The tree has a constant B, and each node in the tree is allowed to have as many as B keys, but no fewer than $B/2$. (B is always even.) For concreteness, let's suppose that B is 4. Nodes are allowed to have as few as 2 keys and no more than 4.

Now suppose we have searched for a key and we didn't find it; we have moved all the way down to a leaf of the tree, and we want to insert the new key. If the leaf has fewer than 4 keys already, there is no problem; we just put the new key into one of the empty slots in the node, and the tree is no deeper than before.

If the leaf node is full, because it already has $B = 4$ keys, we do something interesting. We insert the key into the node anyway. But now it's too big; it has 5 keys and it's only allowed 4. The overstuffed node looks like figure 7:

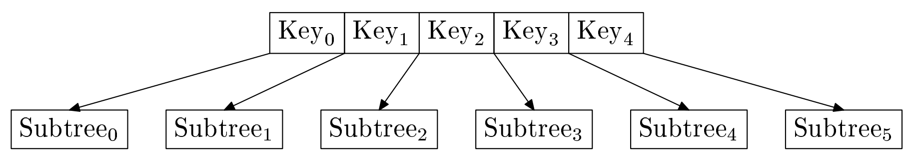


Figure 7: An overstuffed B-tree node

Then we break it in half, so that it looks like figure 8.

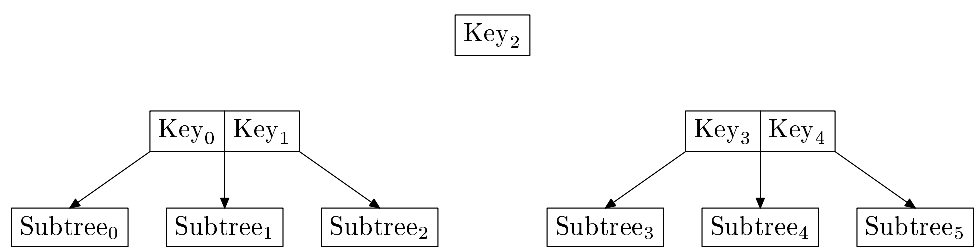


Figure 8: An overstuffed B-tree node, broken in half

There are now two nodes, one with Key_0 and Key_1, and one with Key_3 and Key_4, and a leftover key, Key_2.

And now the trick: If the leaf node's *parent* has room for a new key, we promote Key_2 into it in the appropriate spot. We attach the two new half-full nodes as subnodes of the parent, one just to the left of Key_2 and one just to the right. Everything is still in the correct order.

If we can do this, we've added a key into a full node without making the tree any deeper, by splitting the overfull node into two half-full nodes at the same level, and promoting the extra key up into the parent node, where there was room.

What if there wasn't room for Key_2 in the parent node? Then we repeat the process. We promote Key_2 into the full parent anyway, and then we split the parent node in two and promote the middle key from the parent node into the grandparent node. The splitting and promoting process continues until either there's a node somewhere up in one of the ancestors of the leaf that does have room, or until we get to the root.

If the root is full, we split it, and since there's nowhere to promote the middle key, it gets promoted into a new root node all by itself. This is the exception to the rule that says that there can't be fewer than $B/2$ keys in a node; in this case the root node has only one key. It's also the only time the tree gets any deeper. Since the tree grows from the root up instead of from the leaves down, the leaves are all always at the same depth, which means that the tree is always balanced and never gets all viney.

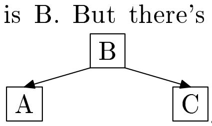
Let's see that A-B-C-D-E example again, the one that gave us a horrible spindly vine. Only this time let's insert these keys into a B-tree, with $B = 2$. $B = 2$ means that nodes are allowed to have no more than 2 keys, and no fewer than 1 key each.

First we make a new root node for A: $\boxed{\text{A}}$.

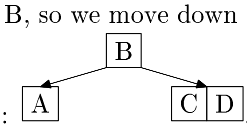
Then, since there's room in the root node for B, we add it there: $\boxed{\text{A}\,|\,\text{B}}$.

(We'll a double key box like $\boxed{\text{A}\,|\,\text{B}}$ to show that A and B are sharing living quarters in the same node.)

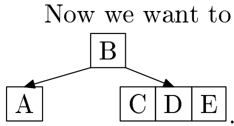
Then we need to insert C into the (only) node. But there's no room, so we split the node into two and try to promote the middle key of the three, which is B. But there's nowhere to promote to, so B gets its own new root node:



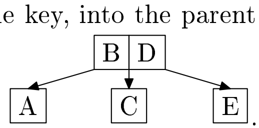
Now we need to insert D; it's greater than B, so we move down to C's node, in we can insert it there because there's room:

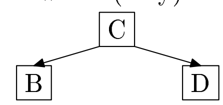


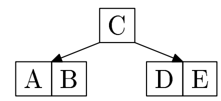
Now we want to insert E. We would have liked to put it into the C-D node:

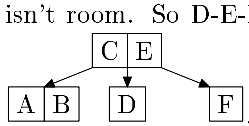


But we're out of room, so we split the C-D-E node, and promote D, the middle key, into the parent node (B). There is room for D there, so we're left with:



Just for kicks, let's see what happens if we get the keys in the order B, D, C, A, E, as in the very first binary tree example. First B goes into a new root node, and then D joins it. Then C wants to join also, but now the (only) node is full, so it splits, and C is promoted to a new root node: .

Now A goes into B's node and E goes into D's node: .

This isn't the same tree as before, but it still has the minimum possible depth. Just for fun, let's add F: It wants to go in with D and E, but there isn't room. So D-E-F splits, and E is promoted to live one level up, with C: .

When B is 2, as in the examples, the B-tree is called a *2-3 tree* because nodes always have either 2 or 3 subtrees. *Red-black* trees are 2-3 trees disguised as binary trees. If you've heard of these, but you didn't know what they were, you do now.

A Guided Tour of the Program

Now we'll see the implementation. The source code is at `http://www.plover.com/~mjd/perl/BTree/BTree.pm` if you'd like to see it all at once; there's a sample test program at `http://www.plover.com/~mjd/perl/BTree/testbt.pl`. The main part of the program, which we'll see in detail, is only about forty lines of Perl.

`BTree.pm` defines two classes. The important one is `BTree`, whose objects represent entire trees. These objects support methods for searching trees for keys, and for inserting new data into trees.

The file also defines a class that's used internally by BTree, called BTree::Node, whose objects are single tree nodes. This package includes methods for getting and setting the keys and data in a particular node. We'll look at the important `BTree` class first, and at the subsidiary `BTree::Node` class only as it becomes necessary.

A `BTree` has only two properties: It has a root node, and it has a constant B. We represent a B-Tree as a hash with two keys, named `B` and `Root`. If `$self` is a `BTree` object, then `$self->B` returns the B constant and `$self->root` returns the root node, which is a `BTree::Node` object.

Moving Down

The most important method in `BTree` is called `B_search`. `B_search` searches a B-Tree for a specified key, returns the associated datum, if there is one, and possibly adds new data to the tree.

There are several different behaviors that are useful here, and it turns out to be simpler to wrap them up into one function rather than to write four nearly-identical functions. For example, suppose that the search process fails to find your key. You might want to add that key with a new datum, or you might not. Similarly, if the search succeeds and your key is in the tree, you might have been looking for it because you wanted to know what data was associated with it, or you might have wanted to throw away the data and replace it with a new one.

`B_search` accepts arguments in 'named parameter' format, like this:

```
$btree->B_search(Key      => $your_key,       # Required
                 Data     => $your_new_data,  # Sometimes required
                 -Insert  => 1,               # Optional
                 -Replace => 1,               # Optional
                );
```

The `Key` is always required, and tells `B_search` what key to search for. Whether `Data` is required depends on whether the `-Insert` and `-Replace` arguments are present.

If the key is not in the tree, `B_search` might do one of two things. It might simply just return a failure code, or it might insert the new key. You select the latter behavior by including the `-Insert => 1` flag in the arguments. In this case the `Data` parameter is required; it is inserted into the tree along with the `Key`.

If the key is in the tree, `B_search` might simply return its associated data, or it might replace that value with new data. You select the latter behavior by including the `-Update => 1` flag in the arguments. In this case the `Data` parameter is required; it is used to replace the data that is already there.

If neither insert nor update mode is in effect, we say that `B_search` is in *search mode*. In search mode you may not supply a `Data` parameter. (What would it be used for, anyway?)

Let's look at `B_search` now in detail. The central idea here is that the method keeps track of a 'current node', with a variable called `$cur_node`. The current node starts at the root of the tree and moves downward until the key is found or until the search terminates at a leaf. In either case, what happens next depends on the flags: The method returns or modifies the associated data, or returns undef, or inserts the new KDP into the tree.

```
1  sub B_search {
2    my $self = shift;
3    my %args = @_;
4    my $cur_node = $self->root;
5    my $k = $args{Key};
6    my $d = $args{Data};
7    my @path;
```

Here we just initialize some important variables. `$self` is the tree we're searching; line 4 initializes the current node to be the root of that tree. Line 3 loads the parameters into an associative array, so that we can access them by name, as for example on lines 5–6.

```
 9    if ($cur_node->is_empty) {        # Special case for empty root
10      if ($args{'Insert'}) {
11        $cur_node->insert_kdp($k => $d);
12        return $d;
13      } else {
14        return undef;
15      }
16    }
```

Lines 9–16 handle the special case of a B-Tree which doesn't have any keys yet. Line 9 checks to see if the root node is empty by calling `$cur_node->is_empty`.[1] If the root node *is* entirely empty, the subroutine doesn't have much work to do: There are no keys in the tree at all, and the search fails immediately. If the subroutine is not in insert mode, it just returns undef to indicate failure. (Line 14.) In insert mode, the subroutine calls `$cur_node->insert_kdp` (line 11) to insert the new KDP into the root. Normally it would have to worry that it might be overfilling the node, but in this case it can be sure that there's room, because the node is entirely empty.

```
18    # Descend tree to leaf
19    for (;;) {
20
```

[1] `is_empty` is the first of several simple functions whose insides we'll see much later or not at all. Please see the full source code if you're interested.

With this trivial special case out of the way, the rest of `B_search` is a big endless loop, lines 19–51, in which `$cur_node`, the current node, moves down the tree one step for each pass through the loop. You can see `$cur_node` starting at the root node on line 4. The subroutine leaves the loop by executing a `return` when the search succeeds (line 28) or fails (line 40.)

```
21      # Didn't hit bottom yet.
22
23      my($there, $where) = $cur_node->locate_key($k);
```

On each pass through the loop, the subroutine checks to see if the desired key is in the current node. It does this on line 23 by calling `$cur_node->locate_key`.

`locate_key` returns two values, called `$there` and `$where`. They are the the answers to two questions, which are:

1. "Is this key in this node?" (`$there`)
2. "Where, exactly, is the key?" (`$where`)

`$there` is a boolean value that says whether the key is in the node or not. It answers the question 'Is it there?', so we put the results in the variable `$there` and write `if ($there)` ... to mean 'if the key was there in the current node...'. Line 24 checks to see if the key was in the current node in exactly this way.

```
24      if ($there) {                # Found it!
25        if ($args{'-Replace'}) {
26          $cur_node->kdp($where, $k => $d);
27        }
28        return $cur_node->data($where);
29      }
```

If `$there` is true, the key is in the current node, the search is done because the subroutine found the key that it was looking for.

The key is in the node, but it is one of many such keys, and `$where` is an index which identifies which one it is. Later on we can use this index in calls to `$node->kdp($where)` to get or replace the key and its associated data. Line 25 checks to see if the subroutine is in replace mode. If not, line 28 just uses `$cur_node->data($where)` to fetch the `$where`th data item from `$cur_node`, which happens to be the one associated with the key, and the subroutine returns the data item. In replace mode, line 26, the subroutine uses `$cur_node->kdp($where, $k => $d)`, which replaces the `$where`th KDP in the `$cur_node` with `$k` and `$d`. The subroutine then returns the new data.

```
31      # Not here---must be in a subtree.
32
33      if ($cur_node->is_leaf) {        # But there are no subtrees
34        return undef unless $args{Flags} & $BTREE_INSERT; # Search failed
35        # Stuff it in
36        $cur_node->insert_kdp($k => $d);
37        if ($self->node_overfull($cur_node)) { # Oops--there was no room.
38          $self->split_and_promote($cur_node, @path);
39        }
40        return $d;
41      }
```

If the key was not in the node, `$where` identifies which of `$cur_node`'s several subtrees contains the key. But if `$cur_node` is a leaf, it has no subtrees, and the search is finished, since we know that the key isn't anywhere to be found. Line 33 uses `$cur_node->is_leaf` to check to see if the current node is a leaf.

If the curent node is a leaf, then the search has failed, and the key is not in the tree. This part of the program is complicated, because this is the case where we might have to insert new keys, split nodes, move up the tree, promote keys to parent nodes, and possibly make a new root, and so we'll come back to it a little later.

```
43      # There are subtrees, and the key is in one of them.
44
45      push @path, [$cur_node, $where];        # Record path from root.
```

If the current node is *not* a leaf, control passes to line 45. Line 45 is responsible for making a record of the path that the search has taken, starting from the root, and making its way downwards. It does this so that if we get to the 'moving up' part of the algorithm, we can remember where we came from and where keys should be promoted to. The record is maintained in a variable called `@path`, which is a list of the nodes that the subroutine visited on the way down from the root, and also of the `$where` values that the subroutine used to get from one node to the next.

```
47      # Move down to search the subtree
48      $cur_node = $cur_node->subnode($where);
49
50      # and start over.
51    }                     # for (;;) ...
```

Line 48 then uses `$cur_node->subnode($where)` to get the identity of the

`$where`th subnode of the current node. The new node is the one that `locate_key` claimed would contain the search key. `B_search` sets `$cur_node` to be this new node, and then begins the loop over again.

Now, what if the search fails? This is on line 33. `$there` was false, so the key was not in the current node, and we would like to search in a subnode of the current node. But `$cur_node->is_leaf` was true, which means that the current node has no subnodes.

```
34        return undef unless $args{'-Insert'}; # Search failed
35        # Stuff it in
36        $cur_node->insert_kdp($k => $d);
37        if ($self->node_overfull($cur_node)) { # Oops--there was no room.
38          $self->split_and_promote($cur_node, @path);
39        }
40        return $d;
```

If the subroutine is not in insert mode, it just returns `undef` to indicate failure, at line 34.

In insert mode, the subroutine first inserts the new key and data into the appropriate place in the current node with `$cur_node->insert_kdp($k => $d)`, line 36. Then it checks to see if the current node is too full by calling `$self->node_overfull($cur_node)`, line 37.

If the node is *not* overfull, the subroutine's work is done, because `insert_kdp` already put the key and data into the right place in the current node, and so it just returns the data associated with the key.

If the node is overfull, however, control moves into the 'moving up' part of the algorithm; we have to split the current node, promote the middle key, and possibly repeat. For convenience and readability, this all happens in a separate subroutine, called `split_and_promote`.

Moving Up

```
sub split_and_promote {
  my $self = shift;
  my ($cur_node, @path) = @_;
```

`split_and_promote` gets two arguments. The first one is the current node, at which it starts. The other is `@path`, which you'll recall contains a complete record of how we got to `$cur_node` in the first place. The last item in `@path` mentions the last node we visited, and that's where `split_and_promote` will have to promote the middle key of `$cur_node`. The next-to-last item in `@path` mentions the next-to-last node we visited, and if `split_and_promote` has to promote a key up another level, the next-to-last node will be the one it'll go into.

```
5    for (;;) {
```

`split_and_promote`, like `B_search`, is an infinite loop, lines 5–21, broken by a `return` when it is done. It too has a notion of the current node, `$cur_node`, which starts out at the leaf node passed from `B_search`, and moves up, one step per pass through the loop.

```
6      my ($newleft, $newright, $kdp) = $cur_node->halves($self->B / 2);
```

The first thing `split_and_promote` does on each pass is to split the current node in two; it does this with the `$cur_node->halves` method. `halves` will break the node anywhere we tell it to, and passing it $B/2$ as an argument breaks it in the middle, so that key number $B/2$ is the leftover one. `halves` returns three things: `$newleft` and `$newright`, which contain the left and right halves of the old node, and `$kdp`, which contains the leftover key and data that will be promoted. If you hark back to figure 8 (page 6,) `$newleft` is the node on the left, with Key$_0$ and Key$_1$ and Subtree$_0$, Subtree$_1$, and Subtree$_2$. `$newright` is the node on the right, with Key$_3$ and Key$_4$ and Subtree$_3$, Subtree$_4$, and Subtree$_5$. `$kdp` is the leftover key and its associated data, which in the picture is Key$_2$.

After splitting the overfull node, `split_and_promote` figures out where to promote the leftover key and where to reattach the `$newleft` and `$newright`. This information is in the `@path` list. Line 7 gets the last element out of `@path`; this element mentions `$up`, the node above the current one, and `$where`, which says that `$cur_node` is the `$where`th sub-node of `$up`.

```
7      my ($up, $where) = @{pop @path};
8      if ($up) {
```

If, on line 7, the `@path` array was exhausted, that means that `$cur_node` is the root of the tree, that the root was overfull, and that line 6 actually split the root node. We test for this possibility on line 8, which checks to see if the `$up` node we thought we got from `path` was actually defined.

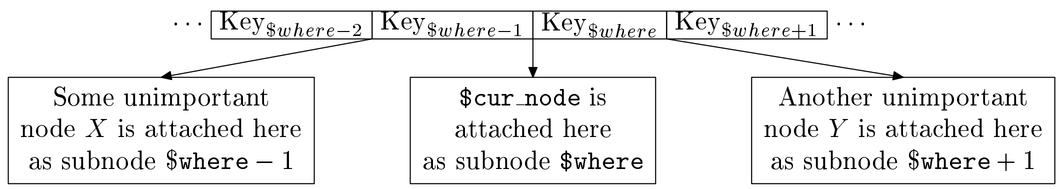


Figure 9: `$up` before promotion of `$kdp`

If so, `$up` looks like figure 9.

`split_and_promote`'s job then is to make `$up` look like figure 10 instead:

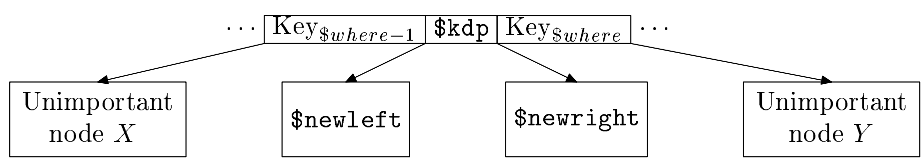


Figure 10: `$up` after promotion of `$kdp`

`split_and_promote` calls `$up->insert_kdp(@$kdp)` to insert the leftover KDP into the appropriate place in `$up`. (Line 9.) `insert_kdp` takes care of moving around the subnodes that are already there so that they stay in the right places. `split_and_promote` then attaches `$newleft` and `$newright` as subnodes of `$up`, adjacent to `$kdp`, by calling `$up->subnode()`. (Lines 10–11.)

If we were paranoid, we could check to make sure that `$kdp` went into `$up` at the place we expected it to by calling `$up->locate_key($kdp->[0])`, and seeing if `$there` was true and if the `$where` we got back matched the one that we got from `@path`.[2]

```
12        return unless $self->node_overfull($up);
13        $cur_node = $up;
```

After attaching the new subnodes, `split_and_promote` checks to see if `$up` is overfull, line 12. If it isn't, then `split_and_promote` is finished, and returns. Otherwise, it sets `$cur_node` to `$up` (line 13), to move one step up the tree, and it starts the infinite loop over again, to split `$up` and promote its middle key another step up.

[2]The `BTree.pm` module actually does include these checks, but I left them out of the article for clarity.

If the promotion goes all the way to the root, and then even the root is overfull, then we have split the root, and when line 7 tries to get the parent of the root off of the `@path` list, it gets nothing. In this case, control passes to line 14:

```
14     } else { # We're at the top; make a new root.
15       my $newroot = new BTree::Node ([$kdp->[0]],
16      [$kdp->[1]],
17      [$newleft, $newright]);
18       $self->root($newroot);
19       return;
20     }
```

Lines 15–17 call `new BTree::Node` to manufacture a new root node with the leftover key and associated data, and with `$newleft` and `$newright` as its only subnodes. Then line 18 sets the root of the tree to be the new root node, and line 19 returns because the whole process is done.

Details

That's been a long rough journey, but it covers the important methods and how they work; everything else is just details. The most important missing detail is what the internal structure of a `BTree::Node` is.

A node needs has three things: It has some keys, some data associated with the keys, and some subnodes. In this program, we store these as three lists, so that each node will be a reference to a list of three lists: The list of keys, the list of data, and the list of subnodes. If there are N keys, there are also N data, and there are $N + 1$ subnodes. Figure 11 is a picture of this.

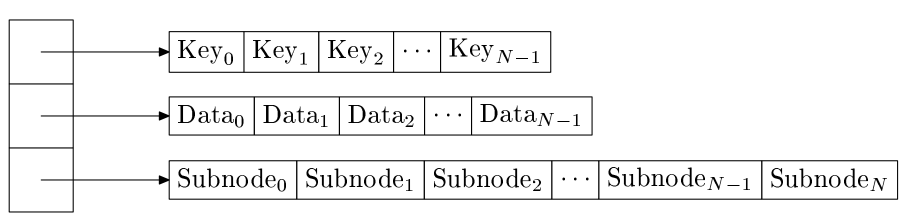


Figure 11: Internal structure of a `BTree::Node`

Empty nodes are represented by a completely empty list `[]`. They only occur as the root nodes of completely empty trees.

The node constructor, `BTree::Node::new`, accepts three parameters, which it installs as the three lists of the new node. If you omit the three lists, it installs nothing, and you get an empty node:

```
sub new {
  my $self = shift;
  my $package = ref $self || $self;
  bless [@_] => $package;
}
```

`split_and_promote` uses the `new` method when it constructs a new root.

The package contains a lot of simple methods for getting and setting keys and subnodes and the like; for example, there's a `subnode` method, which returns the `$n`th subnode of the node if you invoke it like this

```
$node->subnode($n)
```

and sets (and returns) the `$n`th subnode if you invoke it like this:

```
$node->subnode($n, $new_subnode);
```

Here it is:

```
sub subnode {
  my ($self, $n, $newnode) = @_;
  $self->[$SUBNODES][$n] = $newnode if defined $newnode;
  $self->[$SUBNODES][$n];
}
```

`$SUBNODES` is a constant. If `$self` is a `BTree::Node`, it has three lists, and the third of these is the list of subnodes of `$self`. `$SUBNODES` is just 2, so that we can write `$self->[$SUBNODES]` instead of `$self->[2]` when we want to get the list of subnodes, and similarly we write `$self->[$SUBNODES][$n]` instead of `$self->[2][$n]` to get the `$n`th subnode. Analagously, `$KEYS` and `$DATA`, not shown here, are constants equal to 0 and 1.

The `BTree::Node::locate_key` method might be instructive if you've never seen a binary search before. I won't show it, but I will point out a useful software engineering tactic: Binary search is notoriously hard to write correctly—it has a lot of funny boundary cases—and so for the early versions of the module, I didn't bother writing it correctly. I used an easy-to-program linear search instead, and I replaced slow linear search with quick binary search only once everything else was already working.

Other Directions

The most important point about B-Trees is that it's easy to implement a version of B-Trees that saves the tree on the disk. The tree nodes never need to get bigger or smaller, so you never have the problem of having to move one to a different place in the file when you insert a key. For this reason, they are frequently used for disk databases where the data have to be accessed by key. I didn't show this, because the basic algorithm is already complicated enough. Our Most Assiduous Reader might like to modify `BTree.pm` so that it stores and maintains its tree structure in a disk file instead of in memory.

By now you should be thinking of DBM files.[3] One DBM package commonly used with Perl is the Berkeley `DB_File` package, which does in fact use a B-Tree structure to store and retrieve data by key. OMAR might like to add a 'tied hash' interface to the BTree package presented here. The fetching and storing methods are quite simple; fetching is just a call to `B_search` in search mode, and storing is just a call in insert-and-replace mode. Only `nextkey`, which is used by the `keys`, `values`, and `each` functions, presents any real difficulty.

When used in DBM files, B-trees show one more big win over binary trees, even balanced binary trees. The binary search that occurs in a B-tree node takes about the same amount of CPU time as the binary search on the nodes of a binary tree with the sane number of keys. But when the tree lives on the disk, each node must be loaded into memory before it can be examined. In a B-tree, you can adjust B so that each entire node can be loaded with exactly one disk operation, and then searched quickly in memory. In a binary tree, each key resides in its own node, which must be loaded from disk separately. A binary tree therefore typically requires between $B/2$ and B times as many disk accesses as a B-tree of similar size, because it has fewer keys per node. When the tree lives on the disk, the time to search the tree is dominated by the disk access time, and the B-trees are therefore much faster than even balanced binary trees.

DBM files are almost invariably implemented with either B-trees or with hash tables. (Hash tables are the method that Perl uses for regular in-memory hash variables.) However, B-Trees present one enormous advantage over hash tables: The keys are stored in sorted order. Why is this important? Suppose you have a range, and you want to retrieve all the data for all the keys in that range. You can do this efficiently if your database is stored using B-trees: Locate the two keys corresponding to the upper and lower bounds of the range,

[3] A DBM file is a database on the disk. The data in the file are available to your Perl program through a special hash variable, called a 'tied' variable. The tied variable looks just like a regular hash variable, except that when you read from it the data come from the disk and when you store something to it the data are written back to the disk, so that they persist beyond the lifetime of your program.

and then take all the keys in between. (If you did the previous exercise, you can use your `nextkey` function to get the in-between keys.) With hash tables, the keys are not stored in any particular order, so there is no 'in between', and to retrieve a range of them, you must retrieve *all* the keys, extract the ones you want, sort them into order, and then query the hash once for each key, which is vastly less efficient.

Bibliography

Fundamentals of Data Structures in Pascal, Ellis Horowitz and Sartaj Sahni, Computer Science Press, 1984, pp. 491–512.

Notes from Last Time

Rujith S. de Silva pointed out that the `hamming` function from my article about lazy streams could be simplified. Gene Hsu and I discussed multi-way merge functions. Also, I posed an exercise about the sequence $4, 7, 10, 13, 19, 22, 25, 31, \ldots$ and other matters are discussed at `http://www.plover.com/~mjd/perl`.