A dictionary interface.

```
interface Dictionary {
  public Data search(Key k);
  public void insert(Key k, Data d);
  public void delete(Key k);
}
```

A dictionary behaves like a many-to-one function.

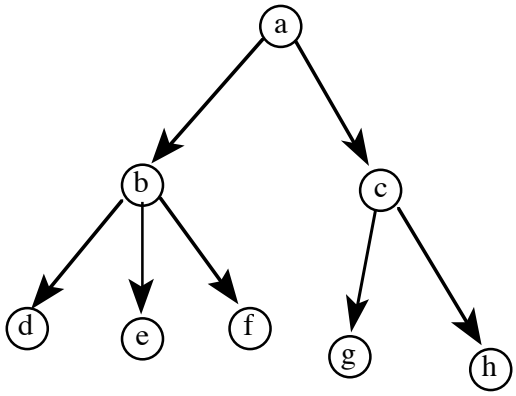
The *search* method returns (a reference to) the data which the dictionary associates with key *k*, or a null reference if there is no such data.

The *insert* method adds key *k* and its data to the dictionary; it will overwrite any earlier data associated with the same key.

The *delete* method changes the dictionary so that searching for key *k* gives a null reference.

Trees.

A *rooted directed tree* is a structure like this:



The circles are ***nodes*** of the tree; the lines are ***edges***.

> *Sometimes you will see* vertex *in place of node or* arc *in place of edge. Strictly, vertex goes with edge and node with arc, but I prefer it my way.*

This is a ***directed*** tree because the edges have a direction from one node to another, indicated here by arrows.

A ***node*** (a, b, c, d, e, f, g, h) can have any number of edges leaving it, including 0.

A ***leaf-node*** (d, e, f, g, h) is a node with no edges leaving it.

The ***root*** (a) has no edge entering it; all other nodes and leaves have exactly one edge entering them. A ***rooted tree*** has only one root.

A ***subtree*** starts at every node, with that node as its root.

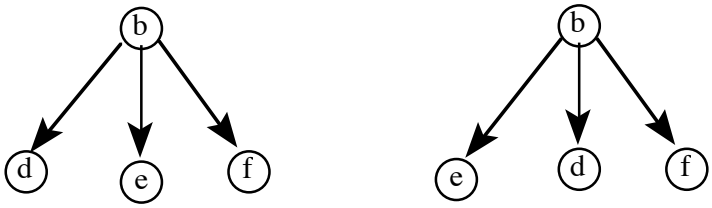
As a consequence of the definition, there is always exactly one way from the root of a tree to any node in the tree.

> *A tree is a special kind of* ***graph****. Graphs will be discussed later.*
>
> *Computer scientists* always *draw their trees with the root at the top. Except for proof trees ...*

We often say that a node is the ***parent*** of the nodes to which its outgoing edges lead, and those nodes as ***children*** nodes. The children of a particular node are ***sibling*** nodes.

We often talk about the ***subtrees*** of a tree, meaning the subtrees which start at the children of the root.

Computer scientists usually deal with rooted, directed, ***ordered*** trees in which the order in which the edges appears matters. For example, these two rooted, directed, ordered trees are distinct:



The only difference is the order of the edges (1st to d, 2nd to e, 3rd to f *vs* 1st to e, 2nd to d, 3rd to f).

As rooted directed trees they are the same: same root, same children; as rooted directed ordered trees they are different: same root, same children but children aren't in the same order.

Computer scientists don't often think about other kinds of tree: when we say 'tree' we usually mean rooted, directed, ordered tree.

Binary trees.

A ***binary tree*** is a rooted directed ordered tree in which each node has either 0, 1 or 2 children; the ordering is established by distinguishing ***left*** and ***right*** subtrees / children.

We distinguish between nodes which have a single subtree according to whether it occurs on the right or left. These are distinct binary trees:

Same root, same child, same number of children: but one has its child on the left and the other has it on the right.

Binary dictionary trees.

BDTs are designed to make it easy to represent a key→data database.

A BDT is either ***empty***, is a ***leaf*** or is a ***twonode***.

An empty BDT contains no information.

A leaf BDT contains a key plus its corresponding data.

A twonode contains a key (but no data) plus a reference to two BDTs.

> *and we shall see that it is convenient if neither of those sub-trees is empty.*

We exploit the edge-ordering in the tree by using a version of the binary chop idea; the records in a BDT are implicitly ordered by their keys.

Richard Bornat
Dept of Computer Science
QUEEN MARY AND WESTFIELD COLLEGE UNIVERSITY OF LONDON

Here is the interface which dictionary trees – BDTs or B-trees – will provide:

```
interface DictTree {
  public Data search(Key k);
  public DictTree insert(Key k, Data d);
  public DictTree delete(Key k);
}
```

Here is the 'wrapper' which allows me to use `DictTrees` as a kind of `Dictionary`:

```
class RecDict implements Dictionary {
  private DictTree t;
  public RecDict(DictTree init) { t = init; }
  public Data search(Key k); { return t.search(k); }
  public void insert(Key k, Data d) {
    t = t.insert(k,d);
  }
  public void delete(Key k); { t = t.delete(k); }
}
```

Searching in a Binary Dictionary Tree.

I define an `abstract class` of BDTs in order that they can share some semi-private information:

```
abstract class BDT implements DictTree {
  abstract public Data search(Key k);
  abstract public DictTree insert(Key k, Data d);
  abstract public DictTree delete(Key k);
}
```

Empty trees don't have much to do:

```
class EmptyBDT extends BDT {
  public Data search(Key k); { return null; }
  ...
}
```

Leaves store a key and its data:

```
class LeafBDT extends BDT {
  private Key k; private Data d;
  public Data search(Key k1); {
    return k.equals(k1) ? d : null;
  }
  ...
}
```

Twonodes contain a key value which they use – as in binary chop – to halve the domain of search.

I have chosen to build my trees so that (i) all the keys in the left subtree are ($\leq k$); (ii) ($k<$) all the keys in the right subtree; (iii) there is no *data* at the twonodes.

```
class TwoBDT extends BDT {
  private Key k; private DictTree left, right;
  public Data search(Key k1); {
    return k1.lesseq(k) ? left.search(k1) :
                          right.search(k1);
  }
}
```

> *My convention is that* `k1.lesseq(k)` *means* $k1 \leq k$. *That makes the code easier to read, even though it may not be the same convention as that used in other Java classes.*

Now ***provided that*** the tree is approximately ***balanced*** – that is, the left subtree is always about as leafy as and about the same height as the right – the search space will be cut in half at each twonode, and we automatically get $O(\lg N)$ search performance.

Inserting in binary dictionary trees.

Empty trees always turn into leaves on insertion:

```
class EmptyBDT extends BDT {
  public Data search(Key k); ...
  public DictTree insert(Key k, Data d) {
    return new LeafBDT(k,d);
  }
  ...
}
```

Leaves may re-assign their data or turn into twonodes:

```
class LeafBDT extends BDT {
  private Key k; private Data d;
  public LeafBDT(Key k0, Data d0) { k=k0; d=d0; }
  public Data search(Key k1); ...
  public DictTree insert(Key k1, Data d1) {
    if k.equals(k1) { d=d1; return this; }
    else {
      DictTree t = new LeafBDT(k1,d1);
      return k1.lesseq(k) ? new TwoBDT(t,k1,this)
                          : new TwoBDT(this,k,t);
    }
  }
  ...
}
```

Leaf nodes decide the keys for twonodes: they establish the ordering *left* ≤ *key* < *right* between key and subtrees.

Insertion into twonodes is remarkably simple:

```
class TwoBDT extends BDT {
  private Key k; private DictTree left, right;
  protected TwoBDT(DictTree l0, Key k0, DictTree r0) {
    left=l0; k=k0; right=r0;
  }
  public Data search(Key k1); ...
  public DictTree insert(Key k1, Data d1); {
    if (k1.lesseq(k)) left=left.insert(k1,d1);
    else right=right.insert(k1,d1);
    return this;
  }
}
```

Insertion into a twonode, ***provided that*** the tree is approximately balanced, will make about $O(\lg N)$ probes and finish either with the creation of a leaf node, an assignment to a leaf node or creation of a leaf node and a twonode.

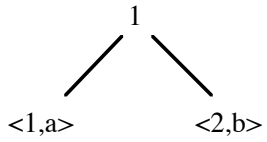
That is far better than the worst case of hash addressing or the average case of binary chop.

But insertion can make a balanced tree unbalanced and produce $O(N)$ search and insert performance.

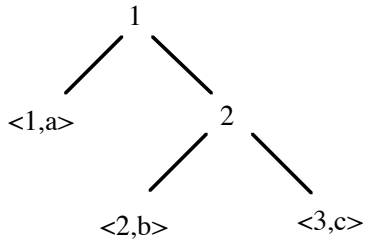
Insert <1,a> into an empty tree:

<1,a>

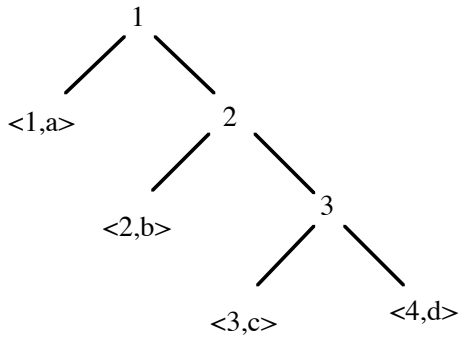
Then insert <2,b>:



Insert <3,c>:



Insert <4,d>:



... and so on.

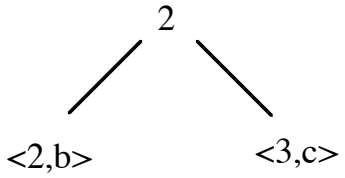
Insertion, as implemented above, is not yet exactly what is required.

If we insert the *same data* in a *different order* we can get a better result:

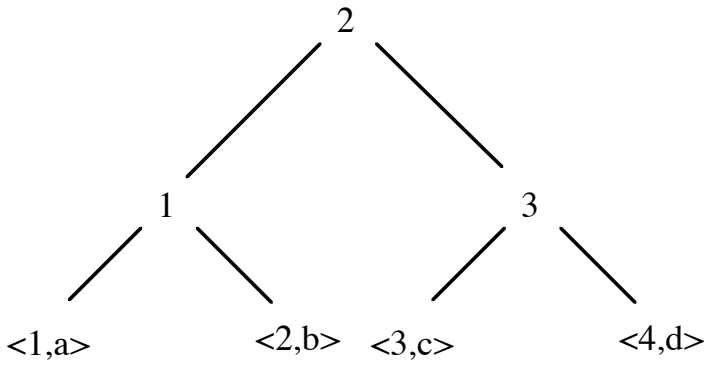
First insert <2,b>:

<2,b>

Then <3,c>:



Then <1,a> and <4,d> in either order:



But it would be oppressive to have to build our dictionaries in exactly the right order.

We shall see that is possible to build balanced *trees, by using a more sophisticated insertion method.*

Deletion in binary dictionary trees.

Deletion from an empty tree has no effect:

```
class EmptyBDT extends BDT {
  public Data search(Key k); ...
  public DictTree insert(Key k, Data d) ...
  public DictTree delete(Key k) { return this; }
  ...
}
```

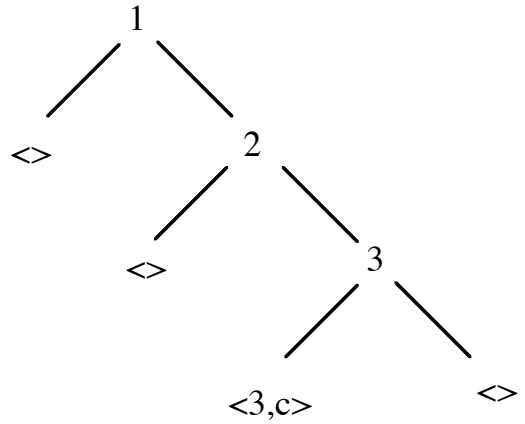
Deletion in a leaf may make it empty:

```
class LeafBDT extends BDT {
  private Key k; private Data d;
  public LeafBDT(Key k0, Data d0) ...
  public Data search(Key k1); ...
  public DictTree insert(Key k1, Data d1) ...
  public DictTree delete(Key k1) {
    if (k.equals(k1)) return new EmptyBDT();
    else return this;
  }
  ...
}
```

Deletion from a twonode is remarkably simple:

```
class TwoBDT extends BDT {
  private Key k; private DictTree left, right;
  protected TwoBDT(...) ...
  public Data search(Key k1); ...
  public DictTree insert(Key k1, Data d1); ...
  public DictTree delete(Key k1) {
    if (k1.lesseq(k)) left=left.delete(k1);
    else right=right.delete(k1);
    return this;
  }
}
```

Deletion can produce even nastier effects than insertion. Because a leaf may be replaced by an empty bdt, it's possible with judicious deletions to make a tree even *worse than unbalanced*:



Luckily, this is easy to fix:

```
class TwoBDT extends BDT {
  private Key k; private DictTree left, right;
  protected TwoBDT(...) ...
  public Data search(Key k1); ...
  public DictTree insert(Key k1, Data d1); ...
  public DictTree delete(Key k1) {
    if (k1.lesseq(k)) left=left.delete(k1);
    else right=right.delete(k1);
    return left instanceof EmptyBDT ? right :
           right instanceof EmptyBDT ? left : this;
  }
}
```

This keeps the trees maximally leafy, but it is obvious that deletion can still take a balanced tree and prune it to make it as unbalanced as insertion can.

It would be impossibly oppressive to have to delete things from our dictionaries in the right order!

AVL trees: height-balanced binary trees.

Adelson-Velskii and Landis invented this data structure in the early 1960s: see Weiss for the reference.

There are lots of other kinds of balanced binary trees: see Weiss chapter 18.

A binary dictionary tree can't be perfectly balanced unless it contains a number of leaves which is a power of 2.

The ***height*** (sometimes *depth*) of a rooted tree is the length of the longest path from root to leaf.

An AVL tree is *height-balanced* because

- at every twonode the height of the subtrees differs by at most 1.

Careless insertion or deletion in a height-balanced tree could make it unbalanced.

A-V & L invented a way of *rebalancing* a tree which has become just slightly unbalanced in this way.

Performance of AVL trees.

Searching in AVL trees is ***exactly*** like searching in a BDT.

If a tree is height-balanced, is it balanced? Answer (a) no; (b) we still get worst case $O(\lg N)$ search performance.

> *I neglect the case of the empty tree, and I rely on the fact that my non-empty AVL trees will never contain empty subtrees.*
>
> *My analysis is **NOT** the same as Weiss's, because my AVL trees are a little different from his: no empty subtrees for me, all the data at the leaves.*

For search performance, the worst case will be a tree with the fewest leaves for its height.

Clearly a tree of height 0 always has 1 leaf and a tree of height 1 always has 2.

A twonode of height h, in the worst case, will have one subtree of height $h-1$, and one of height $h-2$. So $\text{minleaves}(h) = \text{minleaves}(h-1) + \text{minleaves}(h-2)$

That's obviously a Fibonacci series: 1, 2, 3, 5, 8, ...

Fibonacci numbers aren't powers of 2.

But they are powers of *something*. Weiss shows (pp218 and 509) that $Fib_i \approx \phi^i / \sqrt{5}$ where ϕ is a constant. Thus in an AVL tree of height h containing N nodes, $N \geq K\phi^h$ where K is a constant factor. It follows, taking logarithms of both sides, that $\log_\varphi N \geq h + \log_\phi K$. Then, because logarithms of different bases differ by a constant multiplier (slides 4), we have $h \leq \lg_2 \phi \lg N - \log_\phi K$.

Height is logarithmic in the worst case, with some constant of proportionality.

Exact analysis shows (Weiss p509) that an height-balanced binary tree can have a height about 44% greater than an optimally-balanced binary tree.

AVL trees aren't optimally balanced, but they are at most a *constant factor* worse than an optimally-balanced binary tree. We are guaranteed $O(\lg N)$ search performance, if insertion and deletion preserve height-balance.

Implementation of AVL trees.

```
abstract class AVL implements DictTree{
  abstract protected int height();
  abstract public Data search(Key k);
  public DictTree insert(Key k, Data d); {
    return ainsert(k,d);
  }
  public DictTree delete(Key k); {
    return adelete(k);
  }
  abstract protected AVL ainsert(Key k, Data d);
  abstract protected AVL adelete(Key k);
}
```

The height of an empty tree or a leaf is fixed at 0:

```
class EmptyAVL extends AVL {
  protected int height(); { return 0; }
  ...
}

class LeafAVL extends AVL {
  protected int height(); { return 0; }
  ...
}
```

There's an obvious recursive calculation of the height of a twonode: 1+max(height of left subtree, height of right subtree).

This calculation will be needed when trees are rebalanced; performed recursively it would be an $O(N)$ calculation (because it has to visit every leaf in the tree)

Therefore we *cache* the result to avoid recalculation:

```
class TwoAVL extends AVL {
  private int h;
  private void calcheight() {
    h = 1+max(left.height(),right.height())
  }
  protected int height() { return h; }
  protected TwoAVL(Key k0, AVL l0, AVL r0) {
    k=k0; left=l0; right=r0; calcheight();
  }
  ...
}
```

We call *calcheight* when a twonode is created and again when its height changes because of insertion or deletion.

Insertion / deletion in an empty tree or a leaf is ***exactly*** as with BDTs, above.

Insertion and deletion in twonodes is similar to other BDTs, with the exception of a *balance* operation which makes sure that the height-balance property is preserved:

```
class TwoAVL extends BDT {
  ...
  protected AVL ainsert(Key k1, Data d1); {
    if (k1.lesseq(k)) left=left.insert(k1,d1);
    else right=right.insert(k1,d1);
    return balance();
  }
  public AVL adelete(Key k1) {
    if (k1.lesseq(k)) left=left.delete(k1);
    else right=right.delete(k1);
    return left instanceof EmptyAVL ? right :
           right instanceof EmptyAVL ? left :
           balance();
  }
  ...
}
```

The fun is all in the *balance* method.

Balancing an AVL tree.

Insertion doesn't always increase, nor deletion always decrease, the height of a tree.

So we check first to see if the tree is already balanced:

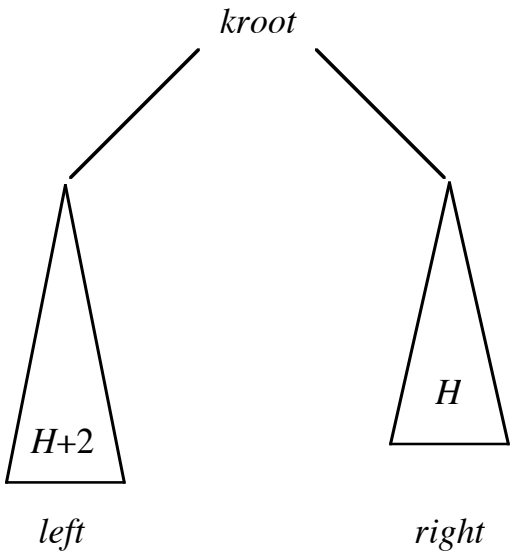
```
private AVL balance() {
  int lh=left.height(), rh=right.height();
  if (abs(lh-rh)<=1) return this;
  else ...
}
```

If we don't have balance, then the left may be too high, or it may be the right:

```
private AVL balance() {
  int lh=left.height(), rh=right.height();
  if (abs(lh-rh)<=1) return this;
  else
  if (lh-rh==2) ... // work on the left subtree
  else          ... // work on the right subtree
}
```

The two cases are symmetrical, so I shall depict only imbalance caused by a left subtree which is too high.

This is the problem:



Balance has been disturbed; to understand how to restore it, we must look at the various ways in which the left subtree might be built.

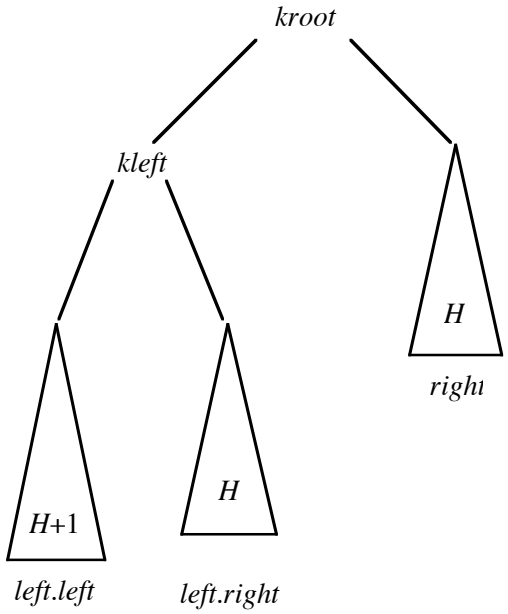
It can't possibly be empty or a leaf.

It might be built out of an H subtree and an $H+1$ subtree, or out of two $H+1$ subtrees, or an $H+1$ and an H subtree.

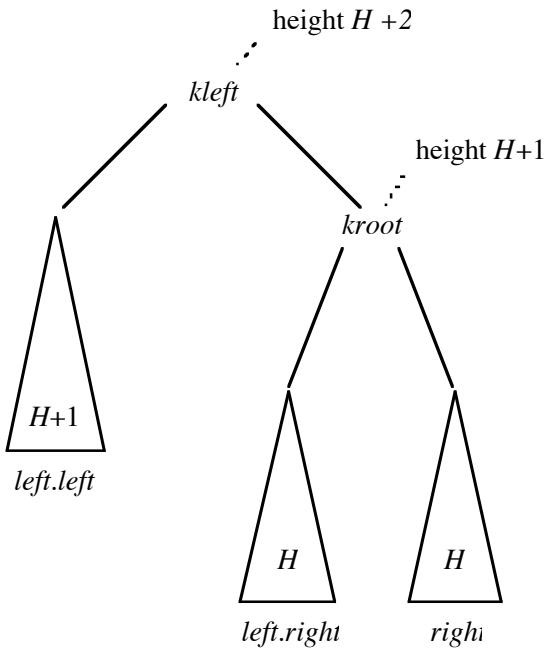
In two of these cases rebalancing is very easy; the other case can be handled by using the same trick more than once.

Case 1: higher on the outside.

Suppose that the situation is this:



Then this tree preserves the ordering of the original, but it's height-balanced:



The operation which changes one tree into the other is very simple, and it's easy to program:

```
protected AVL liftLeftChild() {
  TwoAVL oldleft=(TwoAVL)left;
  left=oldleft.right; oldleft.right=this;
  calcheight(); oldleft.calcheight();
  return oldleft;
}
```

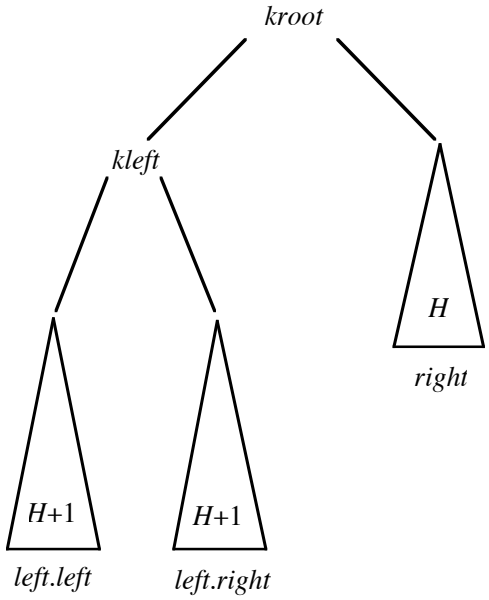
There's a symmetrical *liftRightChild*:

```
protected AVL liftRightChild() {
  TwoAVL oldright=(TwoAVL)right;
  right=oldright.left; oldright.left=this;
  calcheight(); oldright.calcheight();
  return oldright;
}
```

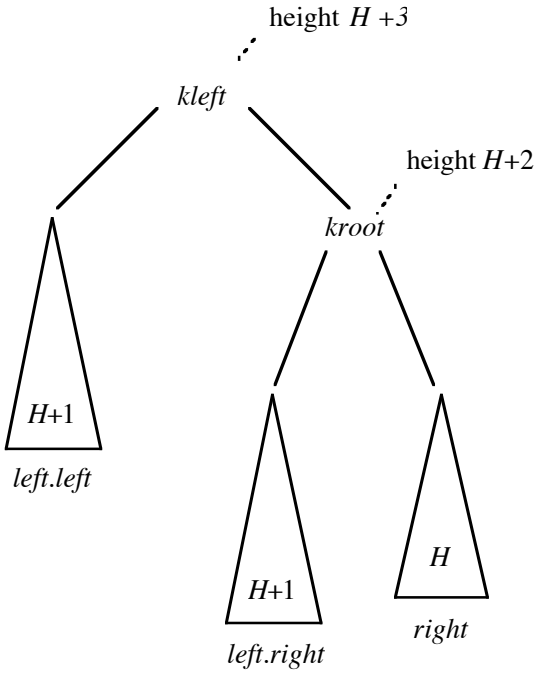
The technique is called ***rotation*** or ***pointer-swinging***. These methods each perform a ***single rotation***.

Case 2: neither is higher.

The first case was easy – so is this:



liftLeftChild will once again give us a height-balanced tree:

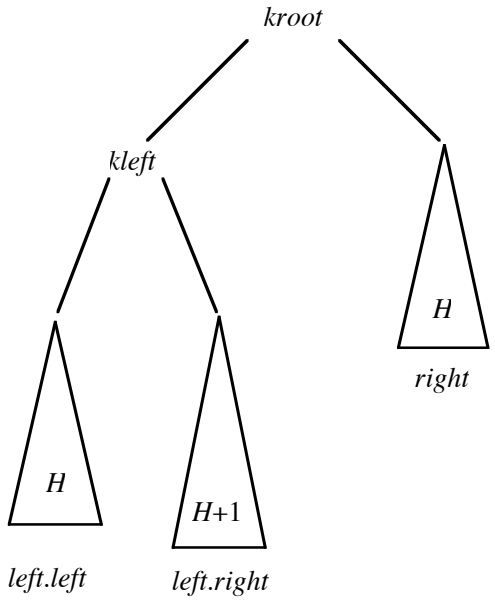


Richard Bornat
Dept of Computer Science
QUEEN MARY AND WESTFIELD COLLEGE UNIVERSITY OF LONDON

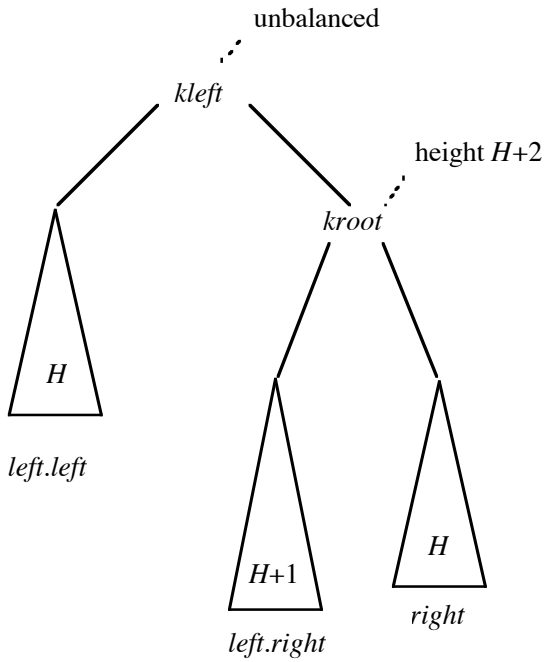
So we can deal with four cases of imbalance:

```
private AVL balance() {
  int lh=left.height(), rh=right.height();
  if (abs(lh-rh)<=1) return this;
  else
  if (lh-rh==2) {// work on the left subtree
    if (left.left.height()>=left.right.height())
      return liftLeftChild();
    else ...
  }
  else { // work on the right subtree
    if (right.right.height()>=right.left.height())
      return liftRightChild();
    else ...
  }
}
```

Case 3: higher on the inside.



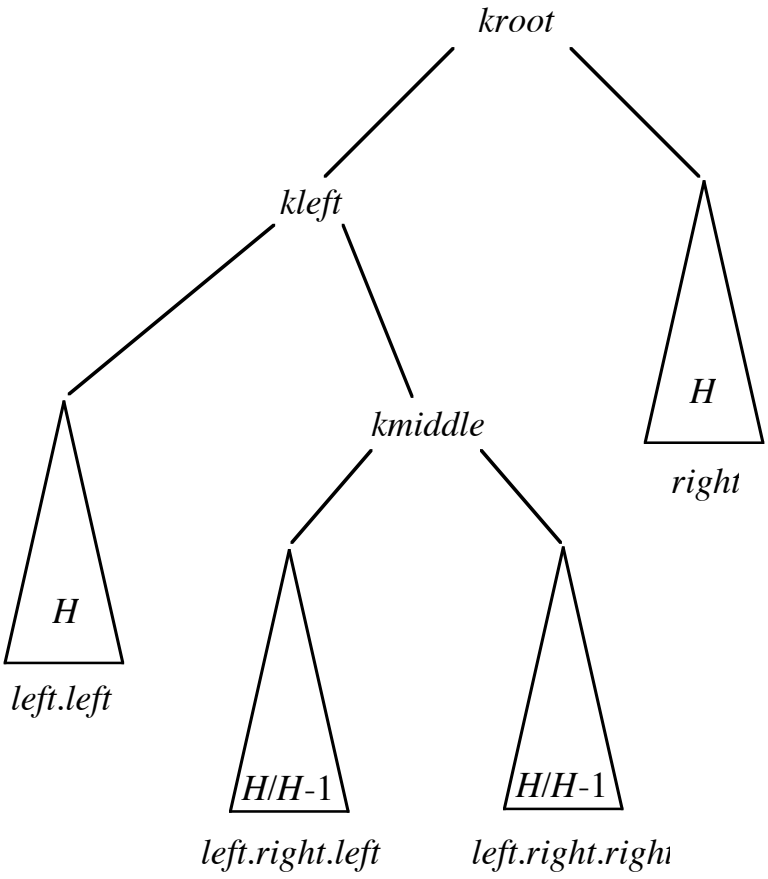
This one needs different treatment, because *liftLeftChild* doesn't give a height-balanced tree:



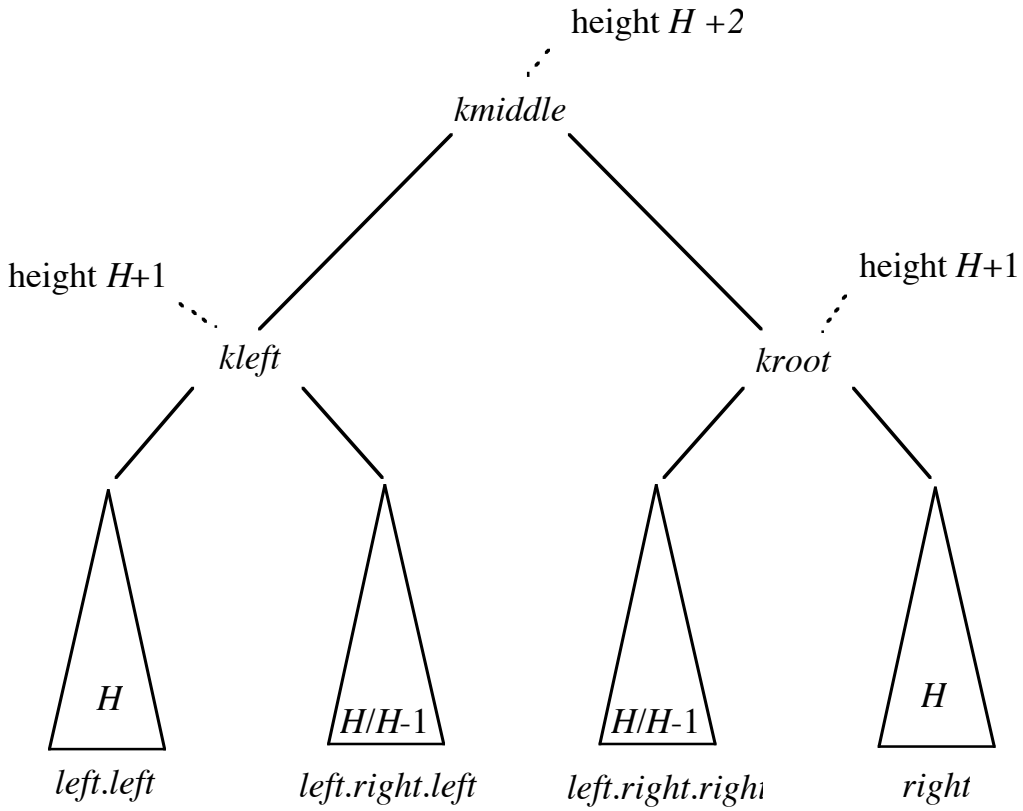
The solution is quite simple, but to see it we have to look at the structure of the middle subtree.

It can't be a leaf.

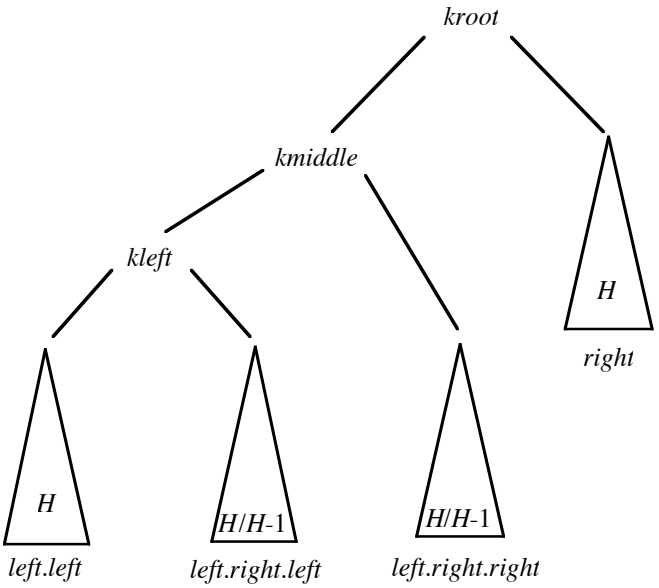
It can be made up of two H subtrees, or an H and an $H - 1$:



In either case, the same rearrangement produces a height-balanced result:

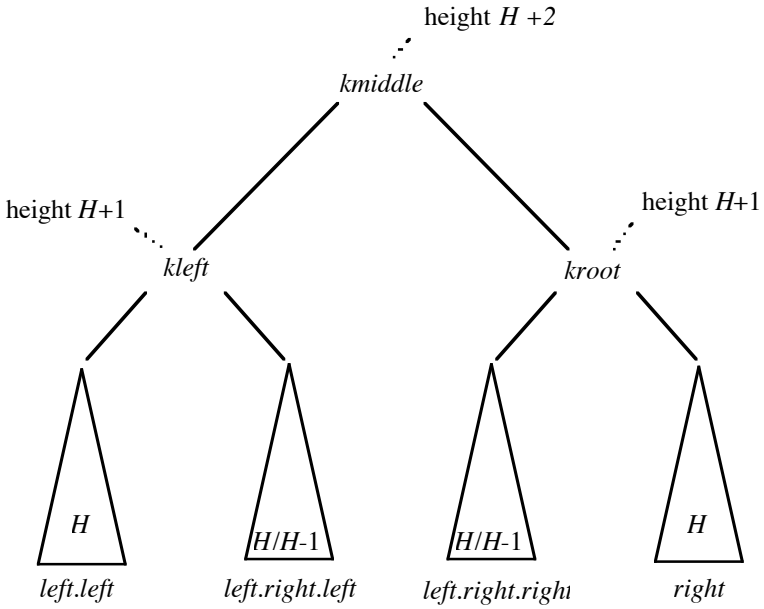


It looks tricky, but it's just a couple of rotations. First we change the *left* subtree by lifting its *right* child:



This may be an unbalanced tree, but that doesn't matter because there's another step to come.

Then lifting the left subchild solves the problem:



So this is the complete solution to balancing AVL trees:

```
private AVL balance() {
  int lh=left.height(), rh=right.height();
  if (abs(lh-rh)<=1) return this;
  else
  if (lh-rh==2) {// work on the left subtree
    if (left.left.height()<left.right.height())
      left = ((TwoAVL)left).liftRightChild();
    return liftLeftChild();
  }
  else { // work on the right subtree
    if (right.right.height()<right.left.height())
      right=((TwoAVL)right).liftLeftChild();
    return liftRightChild();
  }
}
```

And that is ***all there is to it***. Quite a bit of analysis has guaranteed that this very simple code preserves height-balance.

The code chooses between ***single rotation*** (cases 1 and 2) and ***double rotation*** (case 3).

Insert/delete performance in AVL trees.

The search for a position to insert or delete takes $O(\lg N)$ time.

The code which looks at the height of the subtrees to decide whether and how to rotate is $O(1)$, thanks to *calcheight*.

The rotations are $O(1)$ – a couple of assignments each.

So the work to balance the tree is $O(1)$ at each level, and it is performed only on the nodes on the path from root to insert/delete position.

Therefore insert/delete performance is $O(\lg N)$, as required.

How balanced is height-balanced?

The height of a tree is the length of the longest path it contains.

What is the length of the shortest path in an AVL tree?

In the worst case a tree of height h has a subtree of height $h-1$ and another of height $h-2$. The shortest path will obviously be
$\text{shortestpath}(h) = 1 + \text{shortestpath}(h-2)$.

So the shortest path in a tree of height 2 is 1, the shortest path in a tree of height 4 is 2, the shortest path in a tree of height 6 is 3, ... obviously
$\text{shortestpath}(h) \approx h \div 2$.

AVL trees can look very unbalanced: the local one-step height difference doesn't give global height balancing.

Despite that, we still get logarithmic performance.

B-trees.

A B-tree is a rooted, directed tree.

A leaf is an array of up to L <key,data> pairs, searched by binary chop.

A non-leaf (an *internal node*) is an array of up to I <key,subtree> pairs, also searched by binary chop.

In order that binary chop works on internal nodes, the <key,subtree> pairs are arranged so that (i) $\left(key_i \leq\right)$ all the nodes in subtree i; (ii) all the nodes in subtree $i-1$ are $\left(< key_i\right)$

In practice K and L are chosen so that a node just fits into a disc block (e.g. 8192 bytes); a subtree reference is then just a disc block number (4 bytes, 8 bytes, whatever).

> *We can't really implement it in Java, because Java doesn't give us control of array allocation, but we can pretend.*

Implementation of B-trees.

An empty B-tree is just a leaf with no entries.

A leaf contains a count of its number of entries, a key array and a data array.

An internal node contains a count of its number of entries, a key array and a subtree array.

Insertion into a leaf is straightforward, until the leaf becomes full: then insertion must make it split into two half-full leaves and put the extra element in one half or the other.

When a leaf splits the parent node – if there is one – must insert a new <key, subtree> pair. This may cause it to split also, and its parent must insert a new pair ... and so on.

If the root splits then it must be replaced by a new internal node which points to just the two halves of the original root.

In order to limit the depth of the tree, which reduces the number of disc-block reads required to search it, we require that a leaf should always have $L/2$ entries, and an internal node $I/2$ entries.

If deletion in a leaf reduces its entries too much, that leaf can be combined, in its parent, with a neighbour leaf. That may mean reduction of the number of entries in the parent node.

If deletion in an internal node reduces its entries too much, its parent may have to adjust.

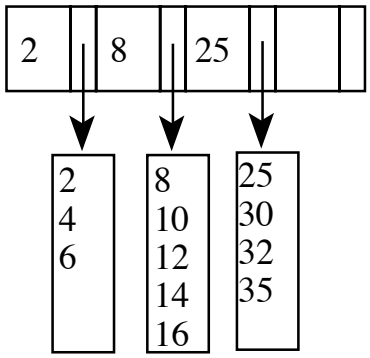
Only the root node is allowed to have less than $L/2$ entries (if it's a leaf) or $I/2$ entries (if it's an internal node).

> *Coding of B-trees is straightforward in principle, but fitting them into an Object-Oriented notation like Java is tricky. So I don't discuss that bit (though I do append some details).*

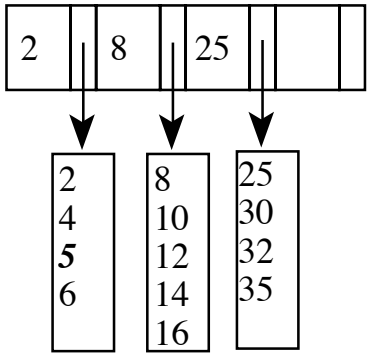
Illustrations of B-trees.

Weiss (p 541-544) has illustrations of B-trees, from which I have copied a style.

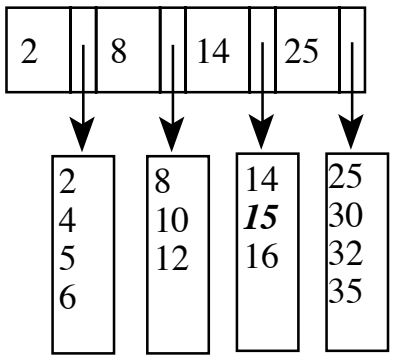
Here is a small B-tree ($I = 4$, $L = 5$) with data values not shown.



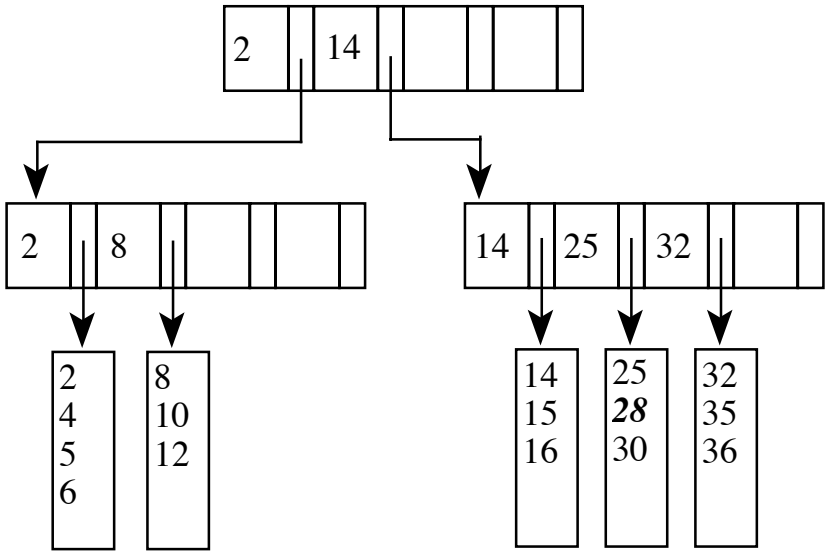
After insertion of *5*, one of the leaves changes:



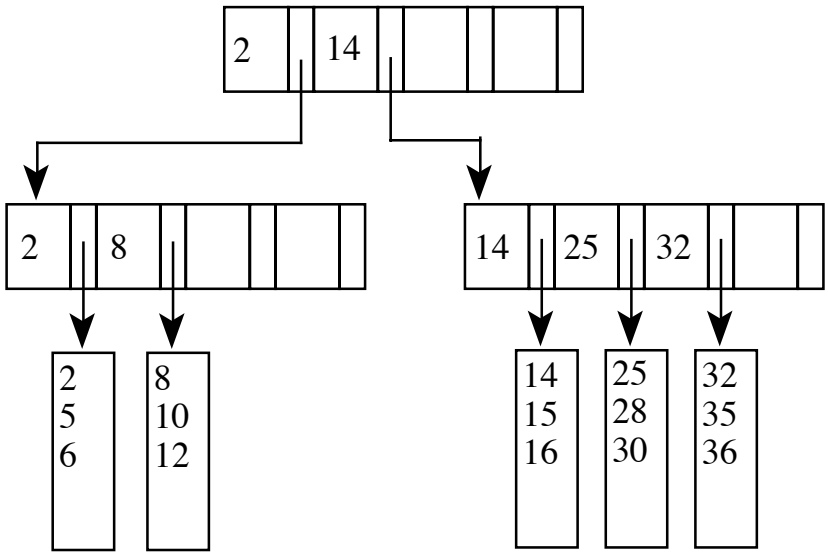
Insertion of *15* makes the middle leaf split, and the parent must add an extra entry:



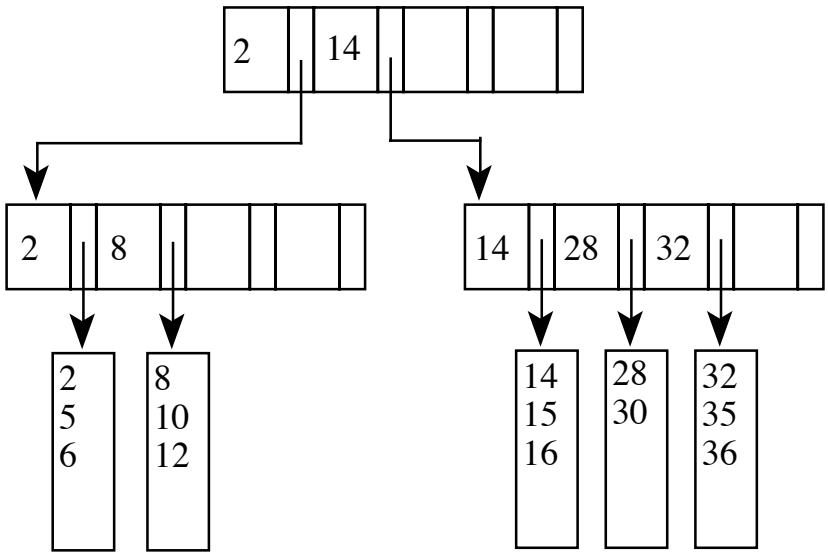
Insertion of *36* will fill the right-hand leaf. Then insertion of *28* will make it split: the parent is full so it must split as well; since the parent is the root we get a new root.



If we delete ***4*** from the tree nothing happens except that the left-hand leaf shrinks:

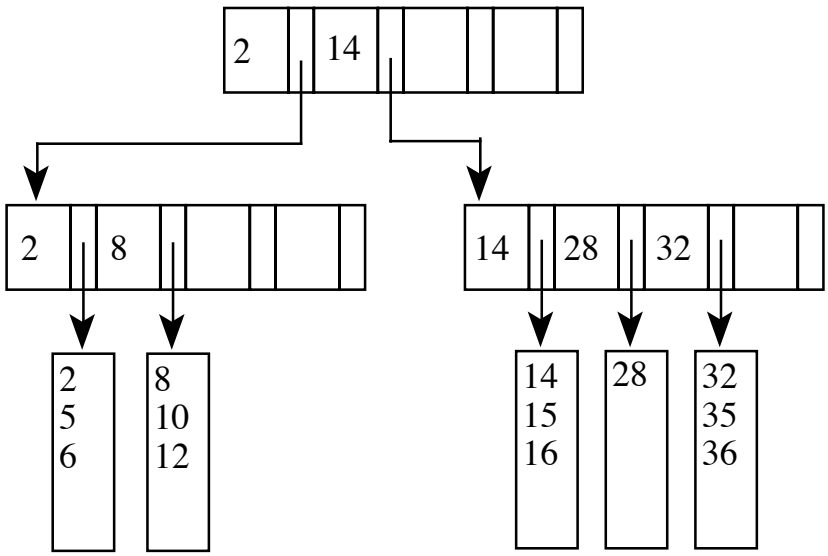


If we delete the first element in a node, its parent must alter the key value it associates with that subtree. For example, if we delete ***25***:

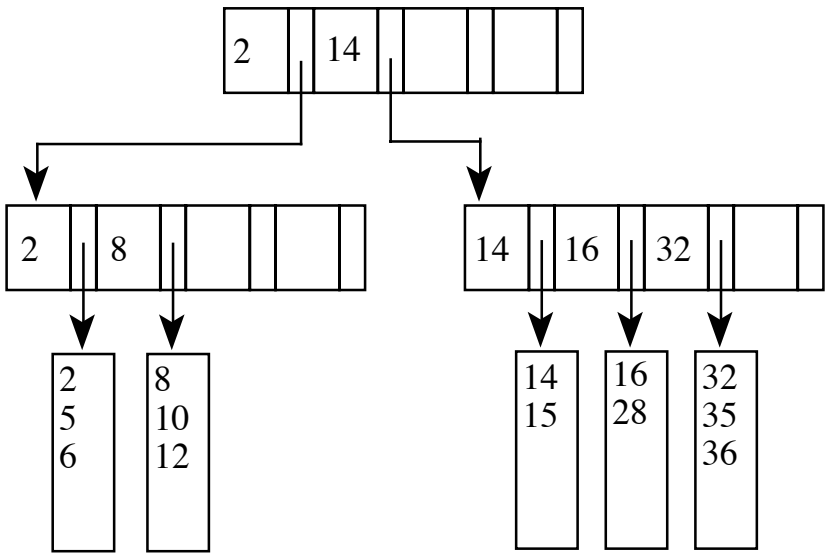


If deletion causes a node to shrink below the $L \div 2$ / $I \div 2$ boundary, then the tree must be rebalanced.

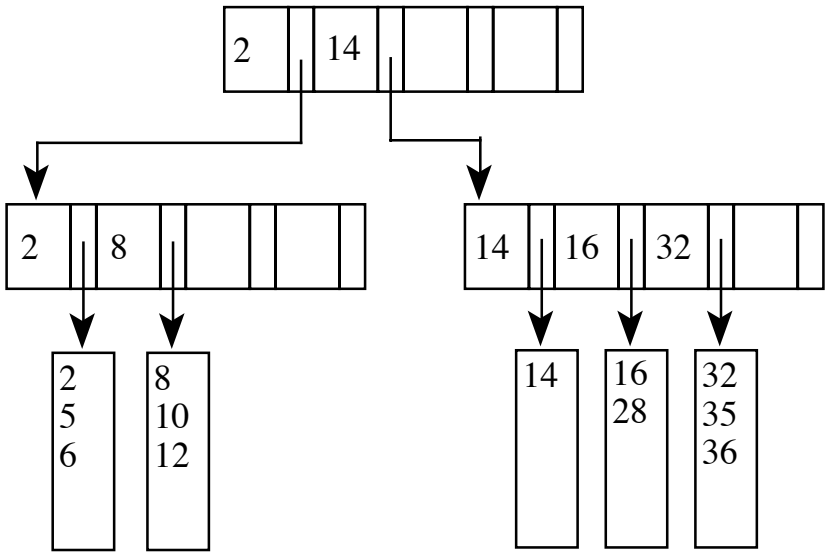
Suppose that we delete *30*. The tree would change to the following:



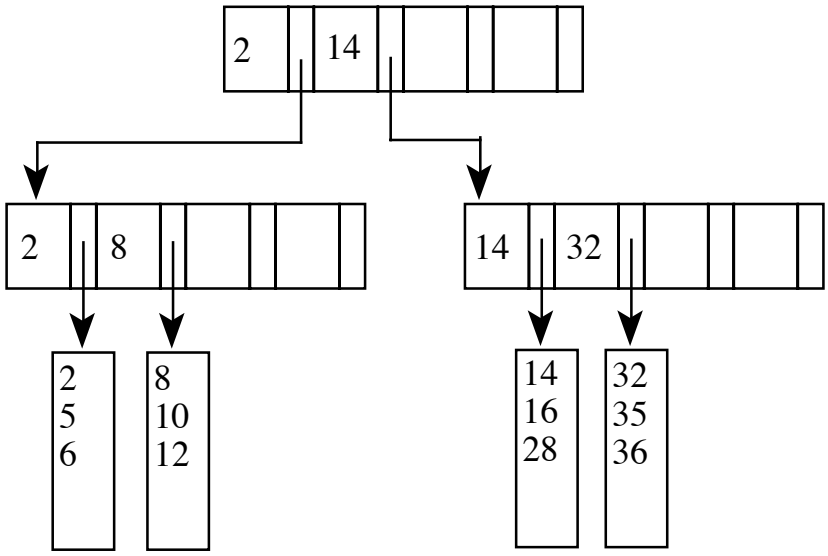
The fourth leaf is now too small, but each of its neighbours is above the $L \div 2$ boundary. We can move an element to balance the tree:



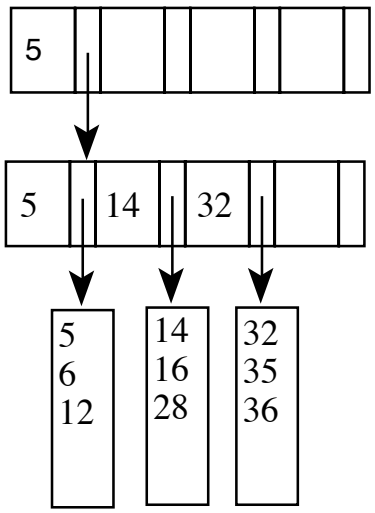
If we delete ***15*** then the neighbour is too small to give up a value:



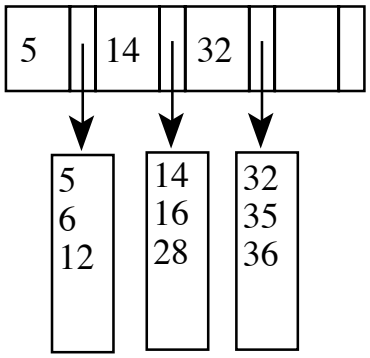
But we can amalgamate with that neighbour:



This process can percolate up the tree, and eventually the root may be reduced to a single-entry internal node:



In which case the root can be replaced by its single child:



Important facts about B-trees.

1. ***B-trees are always perfectly balanced***, in that all the path lengths are always equal.

> *This is because splitting only makes a tree* wider*; it will only get* deeper *if the root splits, and then every path grows by one step.*

2. ***B-trees always have high branching ratios and short paths***.

> *This is provided that K and I are reasonably large: then there are at most* $2N/K$ *leaf nodes, a branching ratio of* $I \div 2$ *and therefore a path length of* $\log_{I \div 2}(2N/K)$ *or* $(\log_{I \div 2} N + \log_{I \div 2} 2 - \log_{I \div 2} K)$.

3. Short path lengths means few node-accesses (disc block reads).

Key points

A dictionary is a mapping from keys to data: we are concerned about search, insert and delete performance.

Hash addressing gives the best search performance when you can work with arrays, but sometimes you can't work with arrays.

Hash addressing gives poor worst-case insert and delete performance.

Rooted directed ordered trees are a recursive organisation of data, a special kind of graph.

If we impose an ordering on the keys, we can use trees to implement dictionaries.

Binary dictionary trees give $O(\lg N)$ search, insert and delete performance in balanced trees, but insertion and deletion may easily make the tree unbalanced.

AVL trees are BDTs which are approximately height-balanced: they can be remarkably *un*balanced, but the balance is always good enough to guarantee $O(\lg N)$ search, insert and delete performance.

In detail AVL trees can be quite a bit more unbalanced than binary trees, but they are never more than a constant factor (about 1.44) worse than an optimally-balanced binary tree; they are a good engineering tradeoff.

The main trick which A-V and L used to preserve approximate balance on insertion or deletion was *rotation* or *pointer swinging*; it can easily be made a simple $O(1)$ operation.

B-trees use high branching ratios (large numbers of children per node) to guarantee short path lengths, and are thus useful when access to a node is much slower than access within a node.

B-trees are always perfectly height-balanced.

Deletion in a B-tree has to be able to move information from one child to another, in order to preserve the high branching ratio.