ReiserFS

- Reiser 3.6
- Reiser 4
- Reiser 5
- ????

Reiser FS

- **ReiserFS**, the **first of several journalling filesystems** we're going to be investigating.
- **ReiserFS 3.6.x** (the version included as part of Linux 2.4) is designed and developed by **Hans Reiser** and his **team of developers at Namesys.**
- **Hans and his team share the philosophy** that
- the **best filesystems are those that help create**
  - a **single shared environment**,
  - or **namespace**,
  - **where applications can interact more directly, efficiently and powerfully**.
- To do this, a filesystem should meet the performance and feature needs of its users.
- That way, **users can continue using** the **filesystem directly**
  - **rather** than **building special-purpose layers**
  - that **run on top of the filesystem**,
  - such **as databases and** the **like**.

Names and Objects

- A **name** is a means of **selecting an object**. An object is anything that acts as though it is a single unified entity. What is an object is context dependent. For instance, if you tell an object to delete itself, many distinctly named entities (that are distinct objects in other ways such as reading) might well disappear as though they are a single object in response to the delete request.

- A **namespace** is a **mapping of names to objects**. **Filesystems**, **databases**, **search engines**, environment variable names within shells, are all examples of namespaces. The early papers using the term tended to seek to convey that namespaces have commonality in their structure, are not fundamentally different, should be based on common design principles, and should be unified.

- Namespaces will never be unified, but the closer we can come to it, the more expressive power the OS will have. Reiser4 seeks to create a storage layer effective for such an eventually unified namespace, and gives it a semantic layer with some minor advantages over the state of the art. Later versions will add more and more expressive semantics to the storage layer.

Names and Objects

- **Finding objects is layered:**
- **1. semantic layer** takes names and **converts them into keys** (we call this "resolving" the name).
- **2. storage layer** (which contains the tree traversing code) takes keys and finds the bytes that store the parts of the object.

- **Keys are the fundamental name used by** the **Reiser4 tree**. They are the name that the storage layer at the bottom of it all understands. They can be used to find anything in the tree, not just whole objects, but parts of objects as well.

- **Everything** in the **tree has exactly one key**. Duplicate keys are allowed, but their use usually means that all duplicates must be examined to see if they really contain what is sought, and so duplicates are usually rare if high performance is desired. Allowing duplicates can allow keys to be more compact in some circumstances (e.g. **hashed directory entries**).
- An objectid cannot be used for finding an object, only **keys can**. Objectids are used to compose keys so as to ensure that keys are unique.

Small file performance

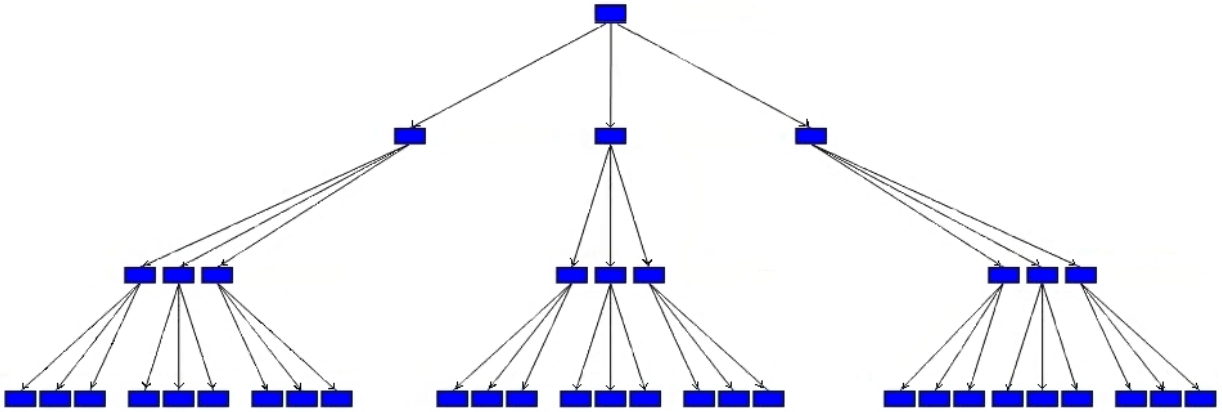
- **Namesys has decided to focus** on **one aspect of the filesystem**, **at least initially** -- **small file performance**.
- **In general**,
  - **filesystems like ext2** and **ufs don't do very well in this area**,
  - often **forcing developers** to **turn to databases or special organizational hacks** to **get the kind of performance they need**.
  - **special-purpose APIs**, which **isn't a good thing**.
- Well, that's the theory**. But how good is ReiserFS' small file performance in practice? Amazingly good**.
- In fact,
  - **ReiserFS** is around **8 to15 times faster than ext2**
  - when handling files **smaller than one k in size**!
- **Even better**,
  - **these performance improvements don't come**
  - at the **expense of performance for other file types**.
- In general, **ReiserFS outperforms ext2 in nearly every area**, but **really shines when it comes to handling small files**

ReiserFS technology

- So how does ReiserFS go about **offering such excellent small file performance?**
- **ReiserFS uses a specially optimized b* balanced tree**
  - (**one per filesystem)**
  - **to organize all filesystem data**.
- This in itself offers a nice performance boost, as well as easing artificial restrictions on filesystem layouts. It's now possible to have a **directory that contains 100,000 other directories**, for example.
- **Another benefit** of **using a b*tree** is that ReiserFS,
  - like most other next-generation filesystems,
  - **dynamically allocates inodes as needed**
  - **rather than creating** a **fixed set of inodes at filesystem creation time**.
- **This helps the filesystem to be more flexible**
  - to the **various storage requirements**
  - that **may be thrown at it,**
  - while at the same time allowing for **some additional space-efficiency**.

Basic Tree Concepts: Trees, Nodes, and Items

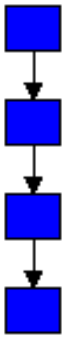
- One way of **organizing information** is to put it into *trees*.
- When we organize information in a computer, we typically sort it into piles (*nodes* we call them), and there is a name (a *pointer*) for each pile that the computer will be able to use to find the pile.



- Figure 1. One Example Of A Tree.
- Some of the nodes can contain pointers, and we can go looking through the nodes to find those pointers to (usually other) nodes.
- We are particularly interested in how to organize so that we can find things when we search for them. A tree is an organization structure that has some useful properties for that purpose.

Fine Points of the Definition

- **Figure 2. The simplest tree.**

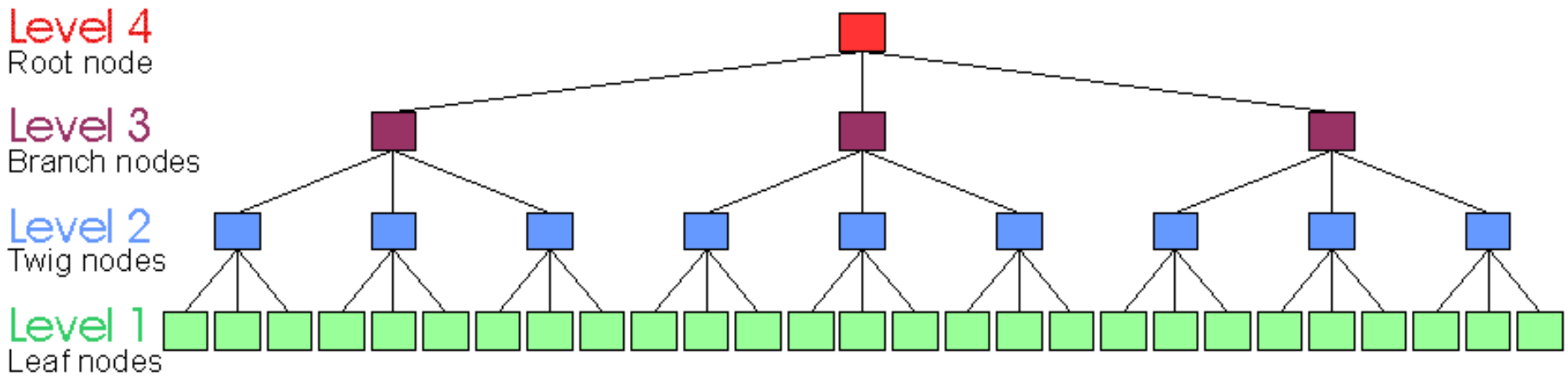


- **Figure 3. A trivial, linear tree.**
- It is interesting to argue over whether finite should be a part of the definition of trees. There are many ways of defining trees, and which is the best definition depends on what your purpose is. Donald Knuth (a well known author of algorithm textbooks) supplies several definitions of tree. As his primary definition of tree he even supplies one which has no pointers/edges/lines in the definition, just sets of nodes.

Ordering The Tree Aids Searching Through It

- **Keys**
- **We assign everything** stored in the tree a ***key***.
- We find things by their keys. Use of keys gives us additional flexibility in how we sort things, and if the keys are small, it gives us a compact means of specifying enough to find the thing. It also limits what information we can use for finding things.
- This limit restricts its usefulness, and so we have a *storage* layer, which finds things by keys, and a *semantic* layer, which has a rich naming system.
- The storage layer chooses keys for things solely to organize storage in a way that will improve performance, and the semantic layer understands names that have meaning to users. As you read, you might want to think about whether this is a useful separation that allows freedom in adding improvements that aid performance in the storage layer, while escaping paying a price for the side effects of those improvements on the flexible naming objectives of the semantic layer.

An example of a Reiser4 tree



- Figure 5. This Reiser4 tree is a 4 level, balanced tree with a fanout of 3.
- In practice Reiser4 fanout is much higher and varies from node to node, but a 4 level tree diagram with 16 million leaf nodes won't fit easily onto my monitor so I drew something smaller....;-)

unbalanced tree

- **Figure 6. This is an unbalanced tree**

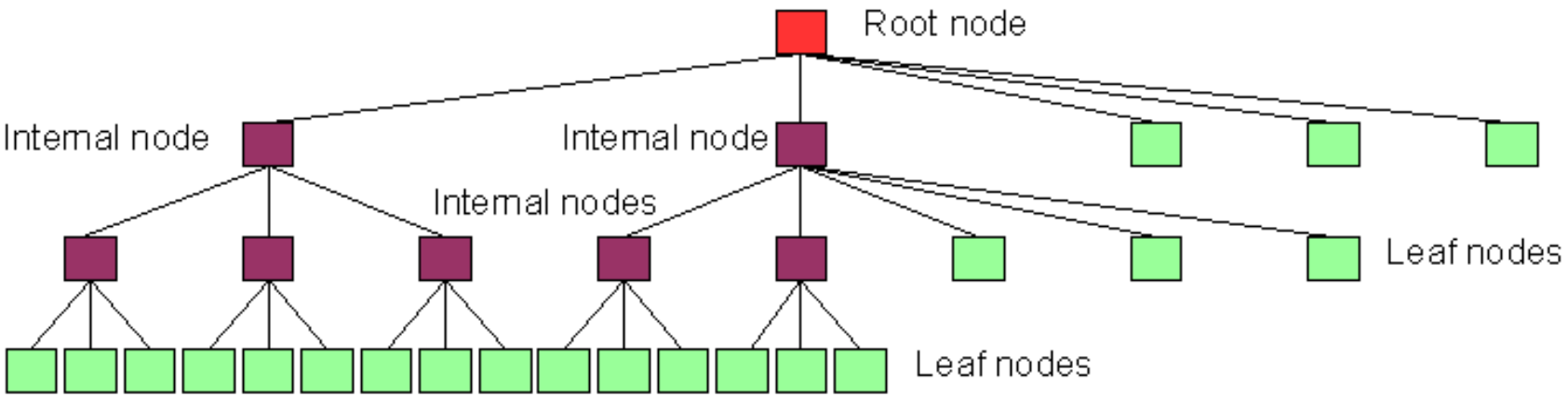
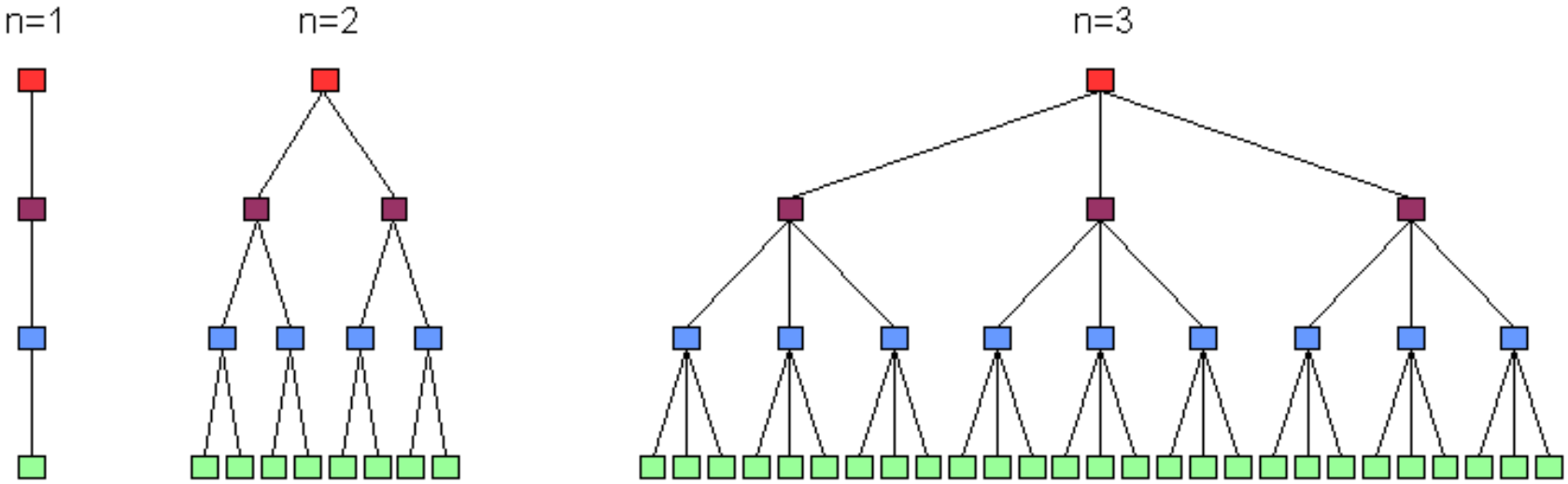


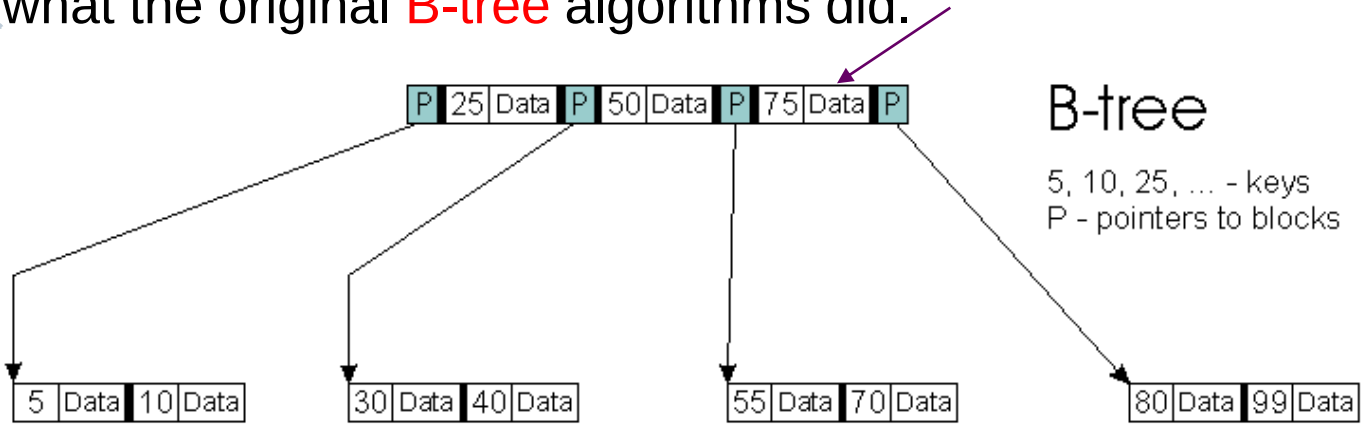
Figure 7. Three 4 level, height balanced trees with fanouts n = 1, 2, and 3.

- The first graph is a four level tree with fanout n = 1. It has just four nodes, starts with the (red) root node, traverses the (burgundy) internal and (blue) twig nodes, and ends with the (green) leaf node which contains the data. The second tree, with 4 levels and fanout n = 2, starts with a root node, traverses 2 internal nodes, each of which points to two twig nodes (for a total of four twig nodes), and each of these points to 2 leaf nodes for a total of 8 leaf nodes. Lastly, a 4 level, fanout n = 3 tree is shown which has 1 root node, 3 internal nodes, 9 twig nodes, and 27 leaf nodes.

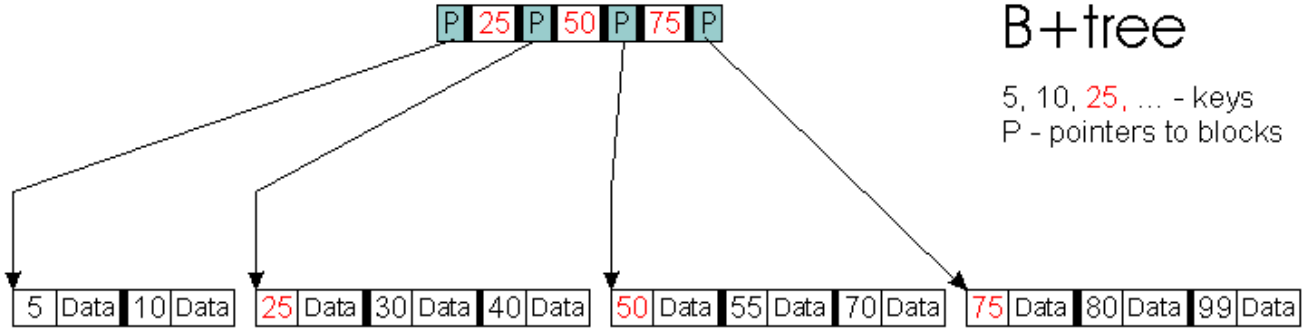


What Are B+Trees, and Why Are They Better than B-Trees

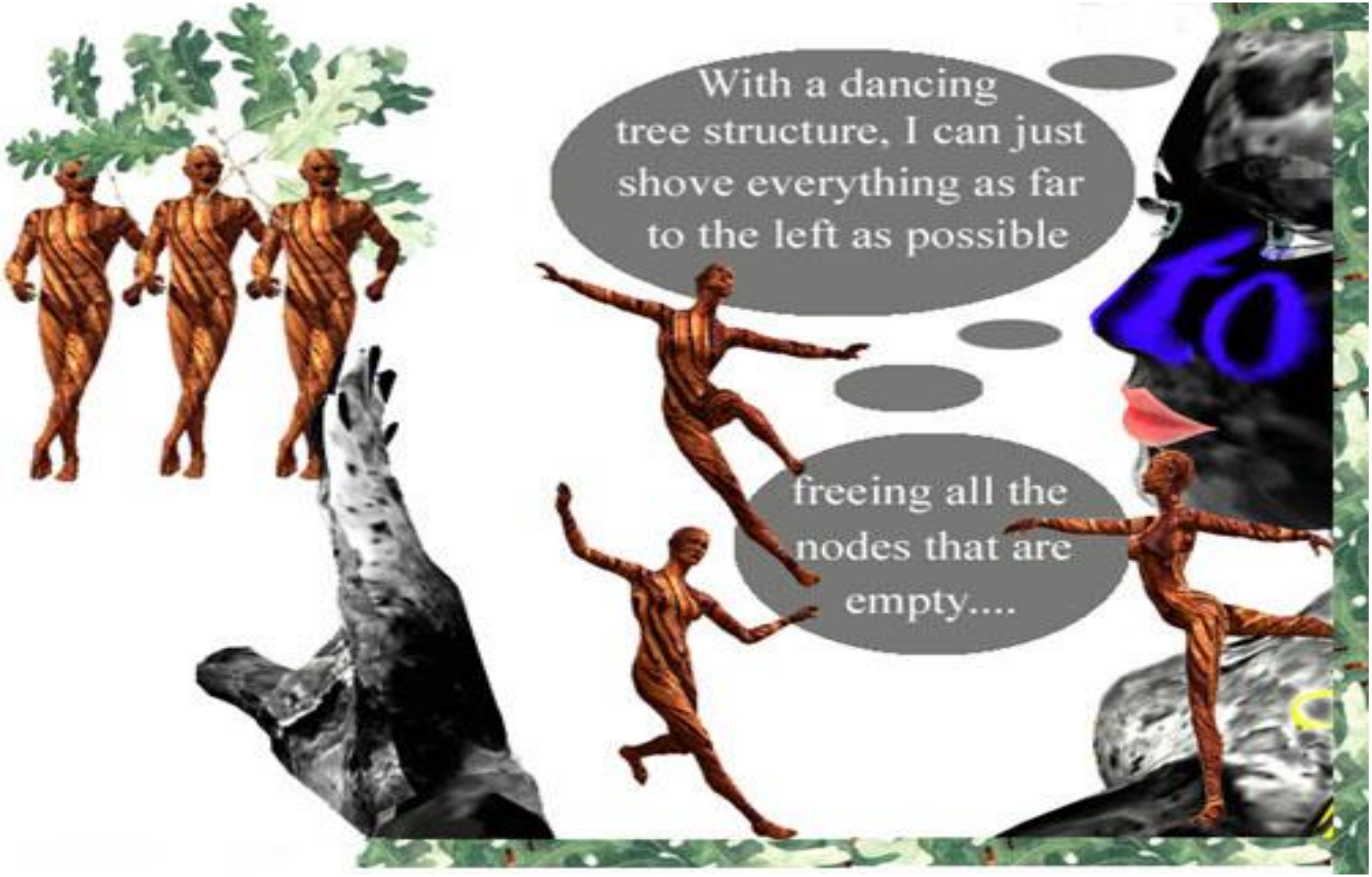
- It is possible to store not just pointers and keys in internal nodes, but also to store the objects those keys correspond to in the internal nodes. This is what the original B-tree algorithms did.



- Then B+trees were invented in which only pointers and keys are stored in internal nodes, and all of the objects are stored at the leaf level.



Dancing Trees Are Faster Than Balanced Trees



ReiserFS technology

- **ReiserFS** also **has a host of features aimed specifically at improving small file performance.**
- **Unlike ext2,**
  - **ReiserFS doesn't allocate storage space in fixed 1K or 4K blocks.**
  - **Instead, it can allocate the exact size it needs.**
- ReiserFS **also includes some special optimizations centered around tails**,
  - a **name** for **files** and end **portions of files**
  - that are smaller than a filesystem block.

ReiserFS technology

- **In order to increase performance**,
  - **ReiserFS is able**
  - **to store files inside** the **b*tree leaf nodes themselves**,
  - **rather than storing** the **data somewhere** else on the **disk**
  - **and pointing to it**.

- **This does two things**.
  - **First**, **it dramatically increases small file performance**.
  - Since the **file data** and the **stat_data (inode)** information
  - **are stored** right **next to each other**,
  - they **can normally be read with a single disk IO operation**.
- **Second**,
  - ReiserFS is **able to pack** the **tails together**,
  - **saving a lot of space**.
  - In fact, a ReiserFS filesystem with **tail packing enabled** (the default)
  - can store **six percent more data than the equivalent ext2 filesystem**.

Ext3 v Reiser