Judy IV Shop Manual

Original author: Alan Silverstein, ajs@fc.hp.com

Last update: 020131

Quick public excerpts created 020805
without rolling in other new material since 020131

invent ? ...

Judy!

1. Introduction

{Note: Public excerpts version 020805 without rolling in all new material since previous major edit 020131. Deleted from the master copy all irrelevant material (such as how to build Judy for HPUX) and all material possibly still “trade secret” beyond what is disclosed by the source package that was LGPL’d on SourceForge.net, 020627. Also there were some minor data structure changes, especially to bitmap branches and opportunistic uncompression, that are not reflected here, so the source code might seem out of sync in spots.}

This document is an internal specification! You should be familiar with the documents available at the external or internal website, including the manual entries and application notes, before trying to make sense of this one. Visit:

```
http://www.hp.com/go/judy             # HP-external.
{http://sourceforge.net/projects/judy  # LGPL sources, etc.}
```

“*Sorry I wrote you such a long letter; I didn’t have time to write a short one.*” (Sorry too, if necessary, for my weak attempts at humor to lighten up this long document.)

1.1 Document Purpose and Audience

This document serves a variety of purposes:

- A collection point for internal tutorials and references to support maintaining and modifying the Judy code.
- A starting point for a new engineer on the team -- hopefully one that is both necessary and sufficient.
- Explanation of Judy internals for outsiders, such as patent writers, managers, and peer engineers. Much of this document grew out of trying to explain Judy to patent attorneys.
- Useful as a reference when trying to envision data structures. Lots of drawings.

This is the default location for all Judy internal concepts, terminology, drawings, guidelines, recipes, and magic spells. Completeness, correctness, and accessibility came before brevity. “*Put all your eggs in one basket and* -- **watch that basket**.” -- Mark Twain (Is anyone watching the basket? ... At least the information will be correct at one point in time when it is written... After that I can only hope it is kept current... Reader discretion is advised.)

This document does **not** contain:

- A list of Judy tasks, issues, or enhancement ideas; those are scattered around elsewhere.
- Much Judy history.
- A lot of repetition of material already covered in external documents, including: Patent applications; in README files; in source file comments; or in other Judy internal documents. Instead hopefully they are all referenced here when and where appropriate.

- Enough humor. *“Trying to define humor is one of the definitions of humor.”* -- Saul Steinberg

Note: This document was written using Framemaker for ease of including drawings and of producing PDFs. If you must edit this document don’t panic, Framemaker is powerful, easy to learn, and portable. For simplicity this is a single, large, monolithic document. Unfortunately this document serves both tutorial and reference purposes and is not optimized for either. A new Judy engineer should read the entire document as a tutorial, but for other purposes it might be useful to look up particular drawings or explanations, but there is no index, sorry. (Should there be?) In lieu of that you might find ‘‘Glossary/Lexicon’’ on page 69 useful.

Thanks to Quincey Koziol of NCSA for lots of useful feedback in December 2002.

1.2 Document Title

Why is this called a “Shop Manual”? It’s more than that, and perhaps not completely that, either. Well I didn’t want to call it an IRS (internal reference specification) because that’s kind of vague and it’s not a complete specification anyway. I toyed with names and the analogy to a shop manual, at least in how it’s used, if not in what it contains, seemed most appropriate.

1.3 Brief Overview of Judy

Judy is a dessert topping **and** a floor wax, but it’s **not** intended for use as a house paint.

API: Judy is a programming library that provides a relatively simple interface (API) for array-like storage of word or string indexes with optional mapping of indexes to single-word values. (“Optional” means Judy1 does not, and JudyL and JudySL do; see below.) Functions are provided to insert, delete, and retrieve indexes; search for neighbor indexes (present or absent) in sorted order; count valid indexes in any range (subexpanse) or locate an index by its position (count); and free entire arrays. So what’s special about that? Judy arrays are remarkably fast, space-efficient, and simple to use. No initialization, configuration, or tuning is required or even possible, yet Judy works well over a wide dynamic range from zero to billions of indexes, over a wide variety of types of data sets -- sequential, clustered, periodic, random.

There are three types of Judy arrays and corresponding classes of access functions:

- Judy1 -- bit array; map long word index to Boolean (true/false)
- JudyL -- word array; map long word to long word value
- JudySL -- word array with string index; map string index to long word value

Note that Judy1 and JudyL support fixed-size indexes, and JudySL supports a particular type of variable-size indexes. These functions are documented in the Judy manual entries, available on the website... ‘nuff said.

Internally Judy is built using digital trees with highly adaptable nodes and a variety of compression tricks. These are explained in great detail starting at ‘‘Smarter Digital Trees’’ on page 26. Why is Judy special if it is a digital tree? Like other “sparse data set” data managers, it adapts itself to the population it is used to store. However, unlike the others, it begins by dividing the “expanse” of the indexes (keys) by expanse, not by population. Adaptation then occurs at each level of the tree -- branch and leaf compression into linear and bitmap forms; index compression based on remaining undecoded bits; level skipping based on “narrow pointers”; etc. All of this is done with a careful eye toward minimizing CPU cache misses, CPU time, and wasted memory, while maintaining source portability, broad dynamic ranges (see ‘‘Judy Dynamic Ranges’’ on page 24), and minimal configuration or tuning.

A word about “**expanses**”. As we use the term, the expanse of a data structure is the range of possible keys that can be used to address it. An unbounded variable-size key has an infinite expanse. A fixed-size key has a finite expanse, such as 0..2^32-1 for a one-word (32-bit) key. Any subexpanse of any expanse is itself an expanse (albeit a smaller one), down to the point where there is exactly one index left. For example, after decoding the first 2 bytes of a 4-byte index, the remaining two bytes have the expanse 0..65535, although perhaps offset by the leading 2 bytes. Decode another byte and the remaining subexpanse is 0..255; literally 0..255 if the first 3 bytes are zero, otherwise offset by their value, say to 256..511 if the first 3 bytes are 0x000001, etc.

You can think of digital trees as peeling (decoding) leading bits off a key until only one bit is left, but in the case of an unbounded variable-size key there is no definite “bottom” (that is, a definite last bit or maximum length for every key). However, there are always unpopulated subexpanses, except with a fixed-size key where every possible key value is stored in the data structure. When decoding keys top-down in this way, each (sub)expanse is defined by the bits already decoded and the number of bits remaining (if finite).

A **divide-by-expanse** hierarchical data structure has some special but unobvious properties, such as localizing insertions and deletions. Worst-case modify times are (rarely) perhaps 20x the average times, and you never need to rebalance the whole tree. However, early explorations of digital trees apparently discarded them as memory hogs not worth pursuing further. It is not obvious that the Judy approach can work, and work as well as it does, but the proof is in the software. Which was hard to write... Even harder than this document. It’s amazing really...

Platforms: As of October 2001, the Judy libraries are available to external customers with HP-UX PA32/64 releases beginning with 11i (11.11), June 2001; to internal customers for HP-UX/IPF (previously called IA64) and Linux/IA32, Linux/IPF, and Win32 (object only) via HP-internal web download; and hopefully later on other platforms including PowerPC and C++. The code makes some assumptions about compilers and hardware, but as few as possible (see ‘‘Machine Dependencies’’ on page 52), and attempts to be as portable as possible.

Just a little **history**: Judy grew out of explorations made by Douglas Baskins (see also ‘‘Some of the Inventor’s Thoughts About Judy’’ on page 80). Judy was named for Doug’s sister because we couldn’t think of a better name for it. A project team was formed to productize Judy in about January 2000. Judy III was made available by internal website in March 2000, and Judy IV was delivered to 11i OEUR for June customer shipments on April 2, 2001. The step from Judy III to

Judy IV turned out to be enormous and time-consuming... Obtaining a ~2x improvement in speed and space required ~5x lines of code and ~10x complexity, but we only lost 3-4 engineers to malnutrition during the implementation phase.

In retrospect this complexity is necessary due to the increased richness of the data structures required for flexibility, as you will read about here, and also the iterative and “unrolled for speed” nature of the Judy software. One of the authors of UNIX said, *“Controlling complexity is the essence of computer programming.*” -- Kernigan. However, in our drive for performance and efficiency we discovered that a remarkable amount of complexity was unavoidable. *“Everything should be made as simple as possible, but not simpler.*” -- Einstein

For more overview on Judy the reader is referred to the manual entries, the “Programming with Judy” book, various application notes, etc; all of which are available on the HP-external website:

```
http://www.hp.com/go/judy/
```

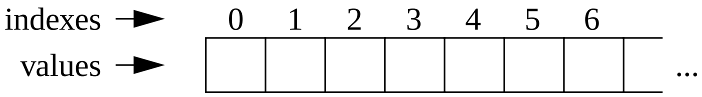
2. Background

2.1 Arrays and Digital Trees

2.1.1 Arrays and Alternatives

Suppose you want to map one or more keys, or equivalently, fields of a single key, to one or more related values. For example, suppose you want to map a phone number to a customer information record, referenced either as a memory address or a disk block address. If your key is short enough and you want speed and simplicity and you don't care about memory or flexibility, you just declare an array:

Figure 1: Simple Array



Usage: Each array value is looked up "positionally", or "digitally". This means multiplying the index by the size in bytes of each value, and adding that offset in bytes to the base address of the array. This is fast to compute, and it results in memory access(es) (CPU cache fills) only for the memory underlying the value of interest, typically only one cache fill. (Assuming the value is not already in cache.) At the computed address you find one "element" which is any number of bits constituting the target value or data associated with the index. In many cases the element is a single word pointer to an external memory location, or perhaps an address on mass storage.

Problems: The problems with a simple array are that it can't handle sparse indexes efficiently (where every possible index is not used or valid), and it can't even pretend to have as many elements as the index can have values, due to machine memory limitations.

Alternatives: There are many alternative data structures for representing sparse data efficiently, such as binary storage trees, b-trees, skiplists, and hash tables. I won't say much about them here, except to note that hashing is essentially a way of converting sparse, possibly multi-word indexes (such as strings) into dense array indexes. The typical hash table is a fixed-size array, and each index into it is the result of a hashing algorithm performed on the original index. (See also ‘‘Hashing Versus Caching’’ on page 18 for a picture of how simple hashing works.)

The problems with hashing are:

- To be efficient, you must use a hash algorithm with good "spectral properties" for the indexes you will store, so each array index is equally likely to be used and synonym chains (collision lists) remain short.
- You must size the hash table to match your data and hashing algorithm, again to keep memory usage down and keep synonym chains short.
- You must anticipate all uses of the data structure in order to tune it.

- Synonym management can be complex and/or destroy performance.
- Hashing causes myopia and possibly even brain damage among programmers who use it frequently.

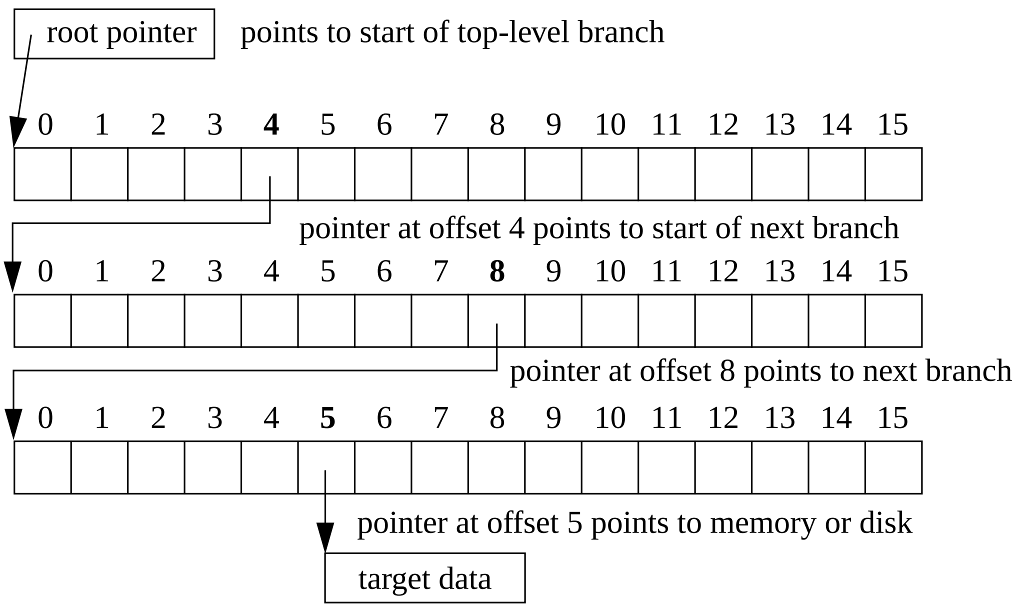
Copy of index: Note that hash tables (and many other data structures) require every data node to contain a copy of (or a pointer to) the original index (key) so you can tell which node is which in each synonym chain (or other type of collision list). If you don't need it, this is a bug (redundant data); if you do need it, it's a feature (self-identifying nodes outside the hash table context).

2.1.2 Digital Trees

In practical applications the index set, that is, the set of interesting valid (stored) key values, is often sparse, such as phone numbers. For handling sparse index sets at the cost of more indirections, that is, memory references (and possibly CPU cache fills), but often less wasted memory than a sparse, flat array, one alternative is a "digital tree", also called a "trie". (See also Knuth Volume 3. But not right now, you won't be back for a week...)

Whereas a flat array "decodes" the entire index in one address calculation as one large "digit", a simple trie is a tree of smaller arrays ("**branch nodes**"), each of which decodes 1..N bits of the index. The "order" of the digital tree is 2^N, and each "digit" consists of N bits. For example, a 4-bits-per-level digital tree has 16-way nodes. Given the decimal number (index) 1157, the equivalent hexadecimal is 0x485, and if each digit has 4 bits, the digits are 4,8,5.

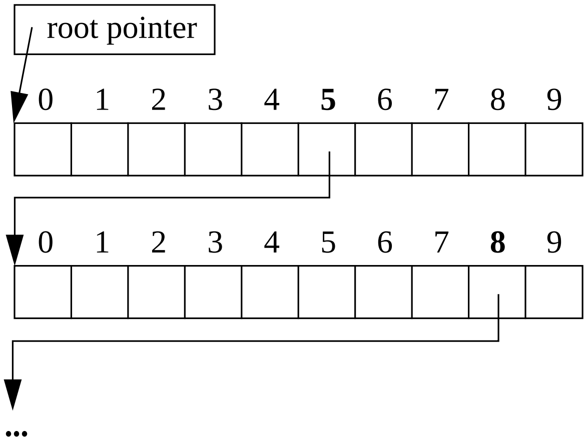
Figure 2a: Digital Tree, 16-way


root pointer
points to start of top-level branch
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
pointer at offset 4 points to start of next branch
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
pointer at offset 8 points to next branch
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
pointer at offset 5 points to memory or disk
target data

In this example, the three levels of the digital tree decode 3 digits (4-bit nibbles) with the values 4 (binary 0100), 8 (1000), and 5 (0101). Each element contains at least 1 word, a pointer to the next level (branch), but could contain multiple words, so long as all elements are the same size so address offsets are computable.

Another example would be using ASCII digits, such as in a phone number, in a digital tree. This picture shows the first two levels of decoding the phone number 581-2526.

Figure 2b: Digital Tree, 10-way



At the “bottom” of the digital tree, the last branch decodes the last bits of the index, and the element points to some storage specific to the index. The “leaves” of the tree are these memory chunks for specific indexes, which have application-specific structures.

Digital trees have been used for dictionaries, where each branch width is, say, 26 letters of the alphabet.

Advantages: A digital tree allows rapidly accessing an arbitrarily large index (series of bits) with no memory allocated in the branches where there are no indexes (zero population) under a given digit (expanse). The pointer corresponding to such digits is null. Furthermore, indexes stored in a digital tree are naturally accessible in (left-to-right) sorted order, and the neighbors of any index can quickly be located even though the index set is sparse and has large gaps.

Disadvantages: The higher the order of the digital tree, that is, the **wider** each branch and the more bits decoded by each digit, the greater the amount of memory that is wasted in null pointers for empty expanses. On the other hand, the **narrower** each branch, the more indirections (and possibly CPU cache fills) are required to access any index. Generally, access time is proportional to the number of significant digits in the index.

Because of the disadvantages, digital trees were not seriously used in the past. Doug didn’t set out to master them, either, he just stumbled into them out of necessity. “*Mater artium necessitas.*” (Necessity is the mother of invention, as Doug is fond of saying, though usually not in Latin.)

-- **Prior art**: Since the preceding was written I have read a lot of academic articles and some textbook sections about the theory and variety of digital trees. There’s a whole body of knowledge

I won’t repeat here. I will just summarize by observing that most prior art seemed rather ignorant of caching issues, probably for portability reasons, and also had a rather myopic desire for data structure simplicity. (Possibly due to excessive experience with hashing.) A few papers did talk about “hybrid” digital trees with variable branch and leaf nodes, but none came close to the Judy features.

On the other hand, all of the ~15 external papers I read focused on theories involving variable-size keys. But Judy grew out of a need to handle fixed-size keys, in particular, mapping single-word indexes to single-word values. Judy1 and JudyL still operate on fixed-size keys only, but we learned a lot about what I now call “meta-tries” such as JudySL that build on top of JudyL. We think Judy1 and JudyL solve problems of practical if not theoretical interest, and that JudyL supports meta-tries well for addressing variable-size key problems. Yes, I admit we are, academically speaking, a bunch of backwoods hicks, but we **do** know how to program for performance!

2.2 Some “Judy Physics”

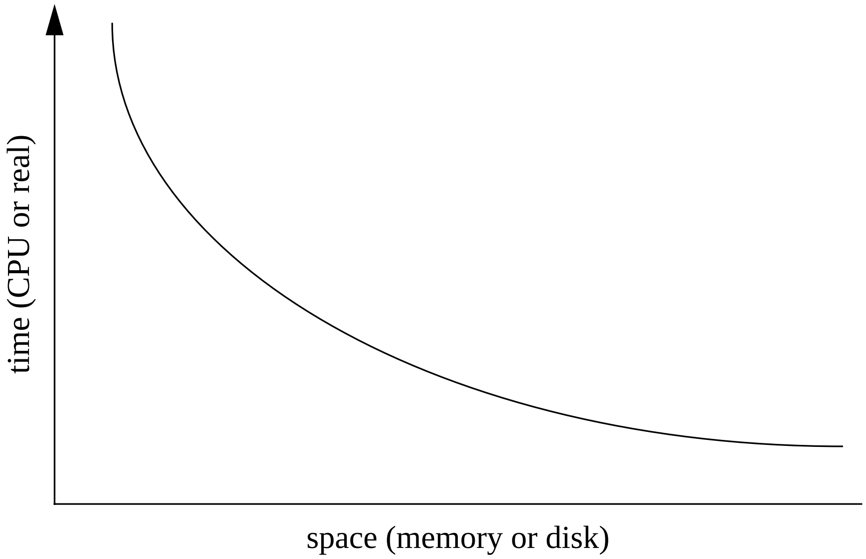
This section was adapted from a separate, earlier “Judy IV Concept Drawings” document written mainly to give as a slide presentation. While of course Judy does not violate any laws of physics (as far as we know), and in fact it does not reveal anything new about physics (only about how programmers think), some informal analogies may be helpful in understanding why Judy is special and how it operates.

2.2.1 Software Relativity (Time Versus Space)

Traditionally, computing problems trade off time and space. If you spend more space (memory, either RAM or disk), you can solve the same problem faster; and if you spend more time (CPU time and/or real time), you can solve the same problem using less space (memory).

Theoretically you can represent the tradeoff like this.

Figure 3: Time/Space Tradeoff



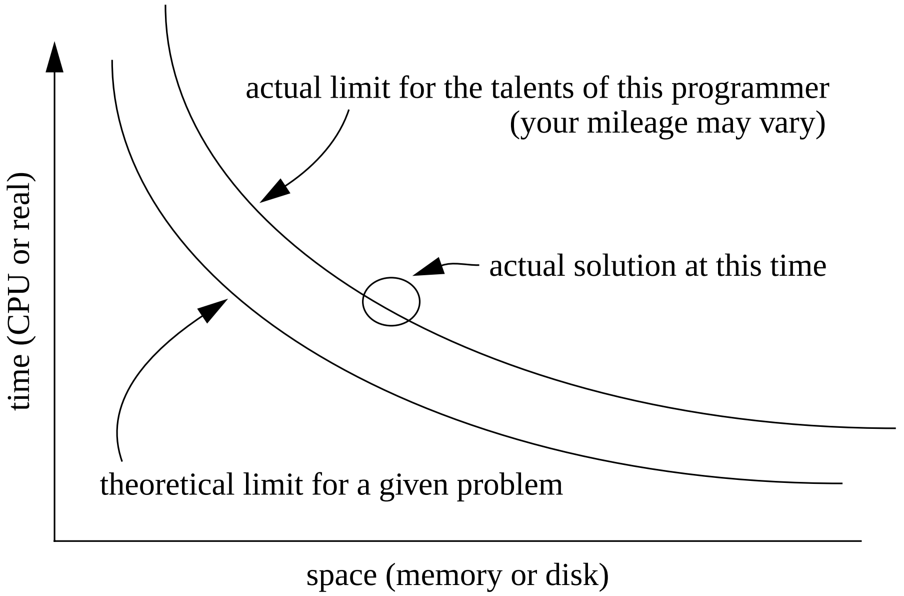
The frequent relationship between time and space in software engineering gives me an inkling that there might exist a “general theory of software relativity”. It would be cool if there was a uniform way to make time/space tradeoff decisions. But that’s a digression from the present discussion.

“*Time is an illusion perpetrated by the manufacturers of space.*”

2.2.1.1 Optimal Versus Suboptimal

It’s entirely possible, and all too frequent, to do worse than the theoretical optimum:

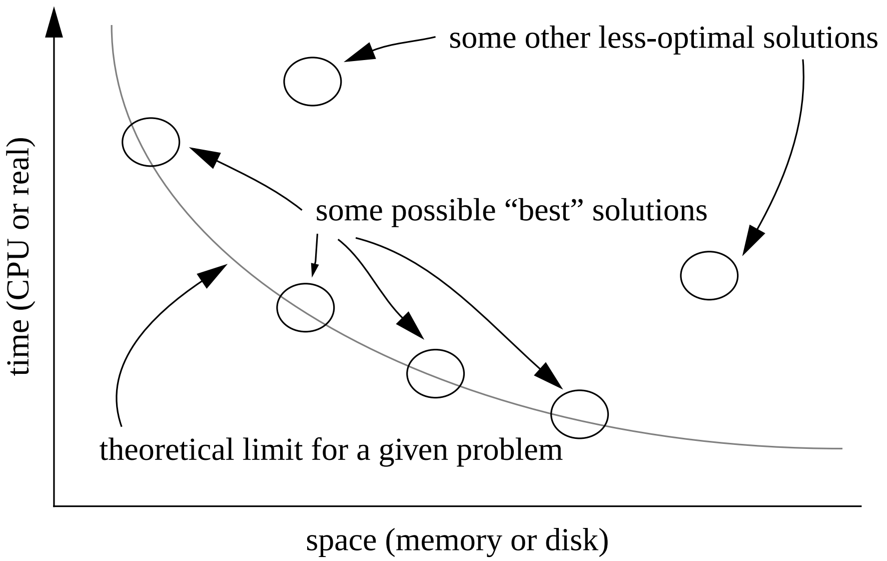
Figure 4: Suboptimal Time/Space Tradeoff


actual limit for the talents of this programmer
(your mileage may vary)
actual solution at this time
time (CPU or real)
theoretical limit for a given problem
space (memory or disk)

2.2.1.2 Practice Versus Theory

Also, in practice the possible ways to solve a given problem are usually quantized, not a smooth spectrum, so the best possible algorithms are a collection, not a curve:

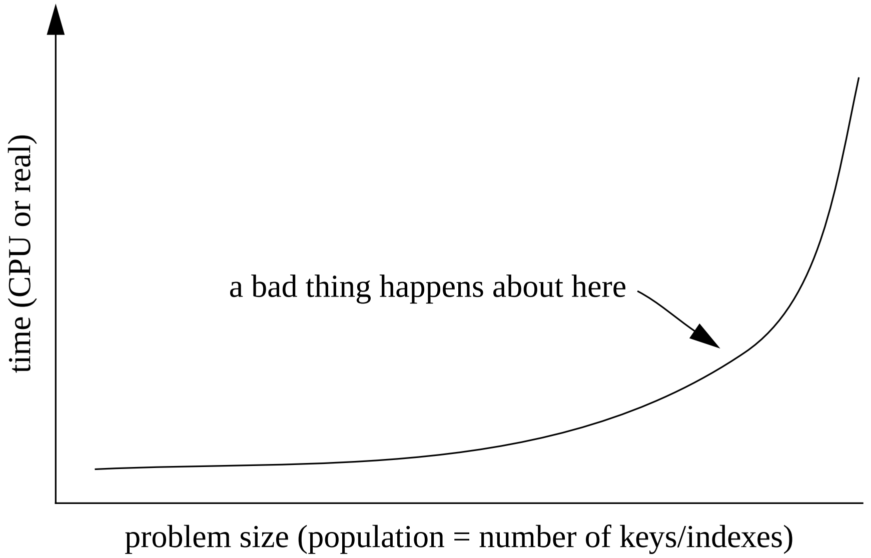
Figure 5: Quantized Solutions


some other less-optimal solutions
some possible “best” solutions
time (CPU or real)
theoretical limit for a given problem
space (memory or disk)

2.2.2 The Caching Cliff

Now, the preceding drawings illustrate behavior for a given “problem size” -- a fixed number of data points, keys, or indexes. Changing the X axis from space to population, observe that most problem solutions have an interesting but frustrating behavior:

Figure 6a: Caching Cliff (Upwards)



What happened here? The problem solution (software) encountered some sort of non-linear resource limit. Most of the time this means running out of RAM and having to start swapping memory pages to disk (“virtual memory”). The computer “caches” disk blocks in memory, but it can’t hold a whole disk worth of data in memory (or we’d just dispense with the disk, except for non-volatile backups). So the disk cache in memory is a limited resource that can run out. If this gets bad enough, the computer spends most of its time swapping (thrashing) and very little time making progress on the problem. Hence the saying, “Thrashing is virtual crashing.”

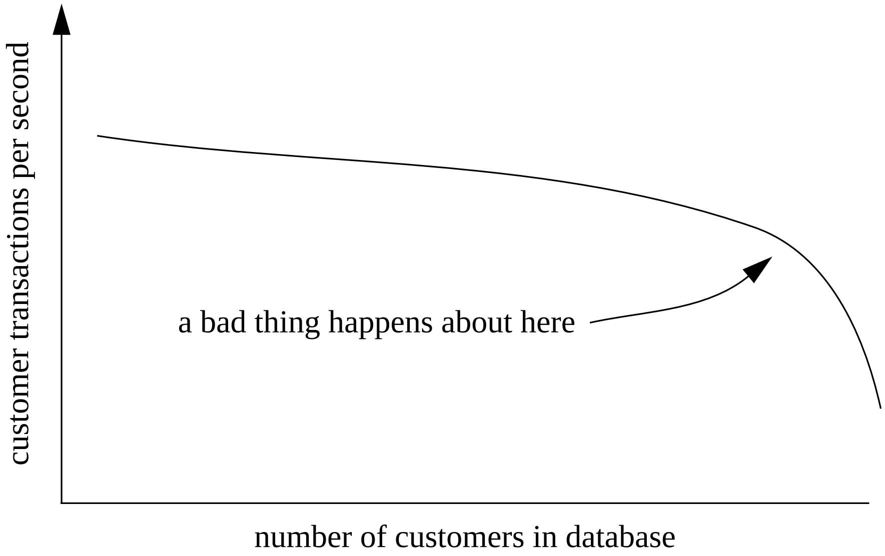
Another type of caching we use all the time and take for granted is network caching. For example, you skip (and update) the local cache when you use the Reload or Refresh button on a Web browser to refresh a page of interest from the Web server.

A relatively recent type of caching of special interest to Judy is CPU cache lines. These are small (such as 16-word) blocks of a relatively small pool of very fast memory (such as 1-2Mb) very close to the processor, sometimes on the same chip, sometimes consisting of multiple levels (such as L0, L1, and L2 cache). These cache lines are used to avoid delays due to reading from or writing to RAM (!), which can be 30-150 times slower than a cache hit. Modern processors and compilers try to make effective use of the CPU cache, but their intelligence is limited, with the result that assembly-level execution profiles show inordinate amounts of time attributed to register load and store instructions (or equivalents) due to cache fill delays.

The “knee” in the theoretical curve above can also be due to running out of CPU cache lines and having to load instructions or data more often from RAM. Of course this happens many times faster than loading data from disk, but the net effect on program performance is the same. I call this effect, “Falling off the caching cliff.”

Never mind that in this case the cliff falls off upward. I could flip over the curve by plotting problem solutions per second, or something like that:

Figure 6b: Caching Cliff (Downwards)

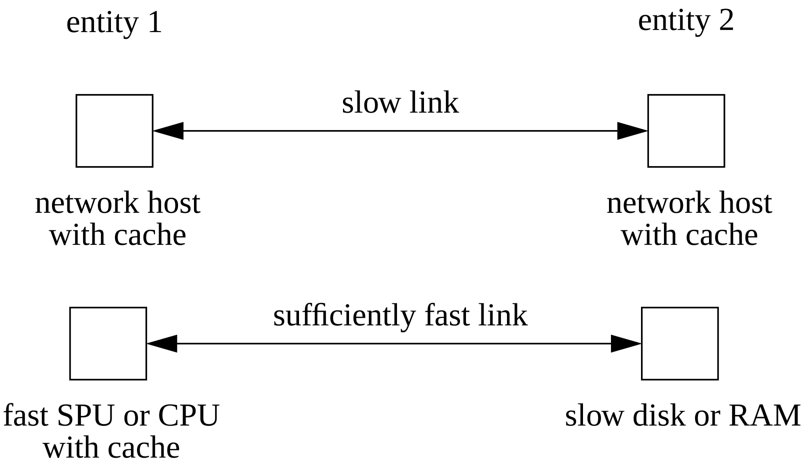


2.2.2.1 Why Cache?

“Real programs don’t eat cache.”

A brief digression... Why do computers bother to do any kind of caching? Caching adds complexity -- the software must worry about cache coherence, purging, flushing, and avoiding falling off the caching cliff. Still, caching is very useful any time two entities communicate over a relatively slow path, or where one entity is significantly faster than the other:

Figure 7: Linked Entities



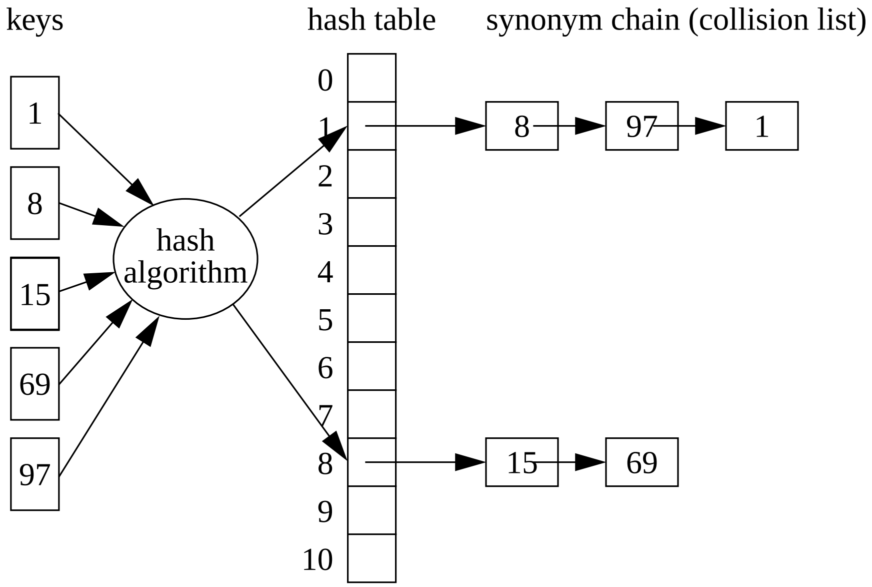
The purpose of a cache is to avoid unnecessary reloads of “hot” data that is likely to be reused soon or frequently. It is not primarily intended to support block transfers of data, nor prefetches, although caches are sometimes used this way by smart programs.

CPU cache lines exist because CPUs are getting so much faster than RAM.

2.2.2.2 Hashing Versus Caching

Another brief digression... A common solution for rapid and efficient access to large numbers of sparsely allocated “keys” is to hash their values and do a lookup through a hash table to one or more “synonyms” in a “synonym chain:”

Figure 8: Hashing



Here the keys 1, 8, and 97 are hashed to offset 1 in the hash table, and keys 15 and 69 are hashed to offset 8. The hash algorithm can be as simple as, say, “take the last four digits of the phone number or SSN,” if that spreads out the keys well across the hash table, or quite a bit more complex, and often surprisingly time-consuming.

It takes some CPU time to compute the hash value, one CPU cache fill to look up the pointer to the first synonym in the chain, and one cache fill to access each synonym to check if it matches the key being looked up. **If** the hash algorithm is well-fitted to the **actual** key values, **and** the hash table size is well-fitted to the **number** of key values (usually within a factor of 2), then hashing can perform very well.

A balanced hash table has short, uniform-length synonym chains. The above example shows a very unbalanced hash table... Which is all too common in practice as datasets outgrow their initially tuned algorithms without the software owners noticing (or in some cases even knowing how to retune the code).

Hashing has two other significant drawbacks. First, the value of each key, no matter how big it is (consider a long character string), must be stored in or pointed at by the corresponding synonym node in order to disambiguate the nodes. Second, hashing almost necessarily randomizes the keys as they are stored, so sorting is a whole ‘nother proposition.

-- After the preceding text was written, Doug discovered a way to merge hashing and JudyL to obtain the best of both. This is documented as an application note on the external website... We call it "scalable hashing", but I hope someday it proves out and is known as "Baskins hashing".

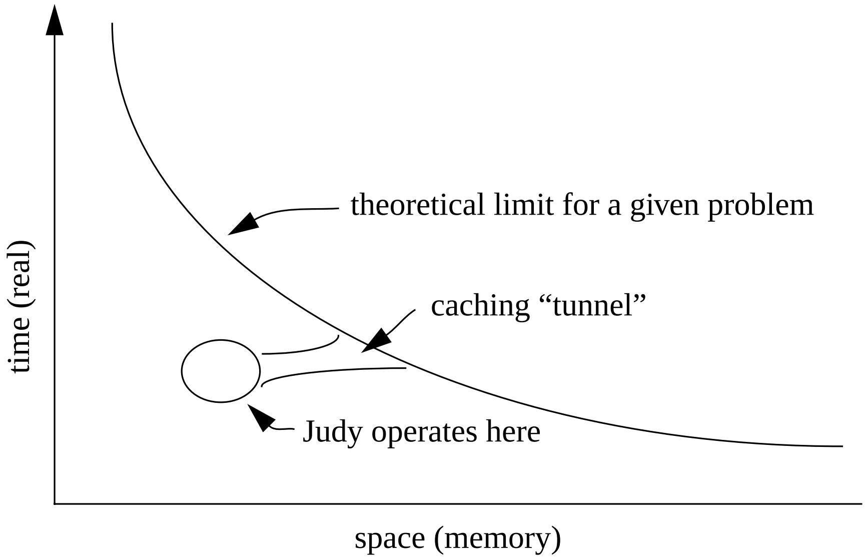
2.2.3 Beating the Curve

Putting the preceding together, Judy beats the theoretical curve (at least metaphorically speaking) essentially by...

- spending **more** CPU time,
- in order to store the dataset in **less** memory,
- that can be accessed **more** efficiently,
- resulting in **less real time** spent on solving the problem,
- due to avoiding falling off the caching cliff (for CPU cache lines), while simultaneously using an acceptable amount of space (memory) per key.

Of course Judy can't break any physical laws, but it takes advantage of new knowledge, mainly cache fills, to do the seemingly impossible. You can think of it this way. Note that in this plot, the Y axis is relabeled as real time only, not CPU time, and the X axis is relabeled as memory only, not disk space:

Figure 9: Beating the Curve



2.2.3.1 Digital Trees

I won't rehash here what a digital tree is, that's already covered elsewhere (see ''Digital Trees'' on page 10). What you need to know here is: A straight digital tree is not very attractive because, as the tree is made narrower and deeper to save memory with a sparse index set, the number of potential or average cache fills increases **faster** than the amount of memory saved by pruning empty expanses. The tree depth affects the number of indirections required to get from the top of the tree to a specific index. Of course, "faster" is a relative term, but for most problems a straight digital tree requires too much memory for obtaining a speed increase over other mechanisms, or else if the amount of memory is reasonable, the speed is lower (more cache fills).

Note: It’s always possible that a needed chunk of data is already in the cache, but for purposes of design and discussion, we assume it is not, so any indirection through a random (non-local) pointer is counted as a cache fill.

Note: As described in ‘‘Digital Trees’’ on page 10, a digital tree divides up the population (index set) uniformly by **expanse** (dividing and redividing the initial expanse evenly), while other methods, such as b-trees, divide up the population by the distribution of the **population** itself. The latter seems superior, but consider the need to keep the tree “balanced” so roughly the same population exists under each path at each level. This costs a lot of time, and can lead to terrible worst-case insert/delete performance.

The following table summarizes a representative spectrum of possible N-way digital trees on a 32-bit system. “Order” is the number of pointers that can be followed at each branch, which is typically 2^digit-size, where “digit-size” is the number of index bits decoded at each branch.

Table 1: Cache fills for different-width digital trees

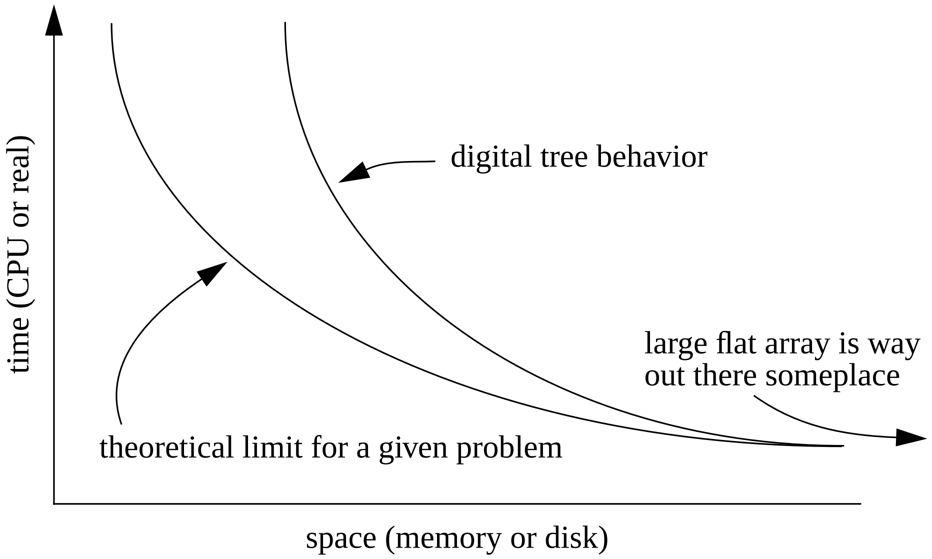
Order of tree (N-way)	Bits decoded per level (branch)	Indirections = cache fills
2^32 = flat array	32 = whole word	1
256	8 = byte	4
16	4	8
8	3	10.67
2 = binary tree	1 = bit	32

Note that a flat array is a degenerate digital tree where the entire index is decoded in one step: address of element = array base address + (index * element-size); a look up takes just one cache fill. However, to store, say, 2^32 indexes in a flat array requires an enormous amount of memory. This is very wasteful if, say, only 1 million indexes are stored out of the 2^32 = 4G (giga = billion) possible values. (Population = 1M, expanse = 4G.)

By narrowing the branches and decoding fewer bits from the index at each level, a digital tree gets deeper (more indirections = cache fills), but when the tree is very sparse, more memory can also be saved. In a “wide branch”, say 256-way, when there are few populated expanses (say 10), the others (246) must be null pointers.

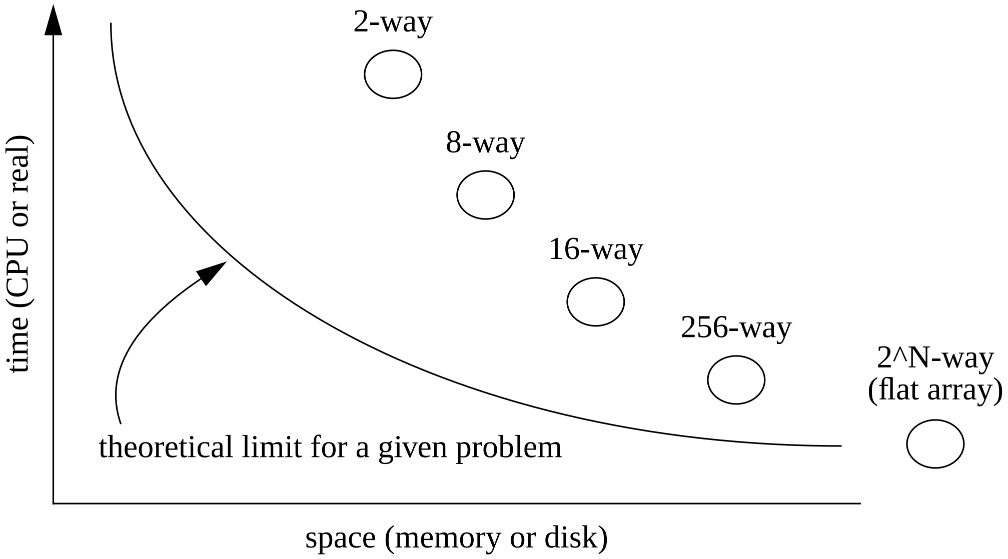
Here’s a picture to illustrate how digital trees are less than ideal due to memory consumption:

Figure 10a: Digital Tree Behavior



In fact the upper curve is not smooth, no more than the theoretical bottom curve. Each choice of branch width results in a different amount of memory used, depending on the index size (overall expanse):

Figure 10b: Digital Tree Behavior (Quantized)

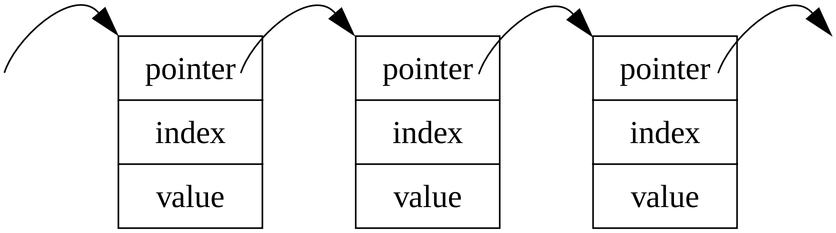


The exact placement of the circles also depends, of course, on the population of the tree. More heavily populated trees take more memory but have a lower number of bytes/index, which is the ultimate measure of space used or wasted.

2.2.3.2 The Bytes/Index Metric

Bear with me for another brief digression... One measure of the space efficiency of a data structure is the total amount of memory required by it, divided by the number of keys or indexes stored -- that is, **bytes per index**. For example, a simple, singly-linked linear list with 1-word values uses 3 words = 12 bytes/index on a 32-bit computer:

Figure 11: Linked List



We think it’s impressive that JudyL, which maps a word to a word, often uses < 10 bytes/index to do the job, **even while** it does it very fast. Consider how long it might take to look up an index in a lengthy linear linked list, especially if each pointer required another cache fill.

2.2.3.3 Compressed Digital Trees

In short, Judy gets the time (minimum cache fill) efficiencies of wide, shallow digital trees, while consuming only a reasonable amount of memory, by compressing unused expanses out of branches and leaves. The compression tricks are explained in more detail in ‘‘Smarter Digital Trees’’ on page 26. Here I’ll just summarize the types of objects used to build the 256-way Judy digital tree... Think of this as a preview, and don’t panic if it’s incomplete or overwhelming.

Table 2: Judy object types

	Linear	Bitmap	Uncompressed
Branch	count of populated subexpanses, followed by an ordered list of populated subexpanse numbers (each 0..255), followed by a matching list of pointers to next-level objects	256 bits, each representing whether one corresponding subexpanse is populated or empty, interspersed with 8 pointers to 8 ordered lists of up to 32 next-level pointers each	standard digital branch; an array of 256 pointers to next-level objects; some pointers are null for empty expanses
Leaf	same as linear branch, except for Judy1 there are no pointers, and for JudyL there are value areas instead of pointers	same as bitmap branch, except for Judy1 there are no pointers, and for JudyL there are value areas instead of pointers [and for 64-bit it’s 4 subsidiary pointers]	**no equivalent**; not needed; a leaf cannot grow beyond a certain size for multiple remaining undecoded index bytes, and a bitmap suffices for 1-byte indexes

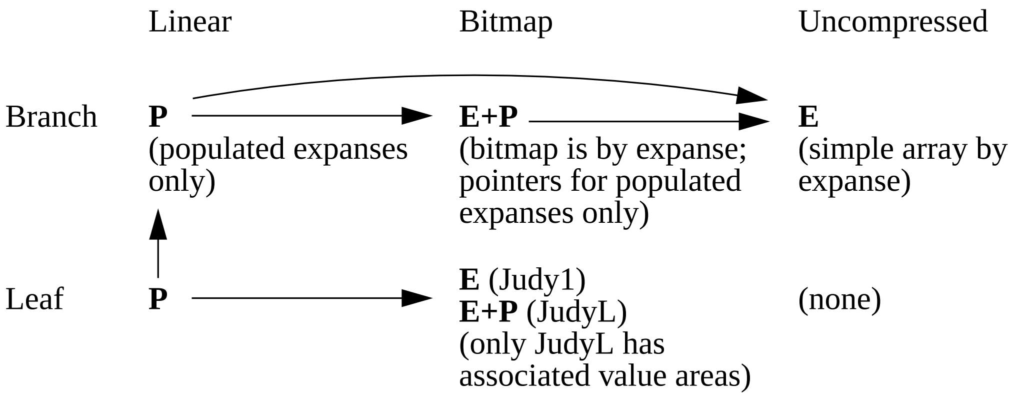
Some points to note:

- A linear branch takes 1 cache fill to traverse. A Judy bitmap branch always takes 2 cache fills. An uncompressed branch, being a simple array, takes 1 cache fill.
- Bitmaps are divided into 8 subexpanses of 32 bits (sub-subexpanses) each [except on 64-bit systems, bitmap leaves have 4 subexpanses of 64 bits each]. Each subexpanse has associated a first-tier pointer that's non-null if any of the corresponding 32 bits is set. otherwise null. Bits in the bitmap must be counted to determine the offset of the next-level pointer in each second-tier list of sub-subexpanse pointers.
- There are no null pointers in linear or bitmap branches, except in a bitmap branch the 8 first-tier pointers can be empty if none of the 32 corresponding sub-subexpanses is populated. The whole point is to "compress out" the null pointers for the unpopulated expanses so no memory is wasted, even on the null pointers themselves.

2.2.3.4 Judy Population/Expanse Organization and Growth

Here's a drawing illustrating the way the different Judy data structures organize the indexes, dividing them by population (P) or by expanse (E), and how the different data structures are used as indexes are inserted and the Judy array grows. This drawing will make more sense after you read about the data structures in more detail in ''Judy IV Data Structures'' on page 27, and if it doesn't, well, my email address is ajs@fc.hp.com.

Figure 12: Judy Population/Expanse Organization and Growth


Linear
Bitmap
Uncompressed
Branch
P
(populated expanses only)
E+P
(bitmap is by expanse; pointers for populated expanses only)
E
(simple array by expanse)
Leaf
P
E (Judy1)
E+P (JudyL)
(only JudyL has associated value areas)
(none)

When a linear branch overflows it becomes a bitmap branch, unless the population of the branch or the whole array is high enough to "amortize" the cost of an uncompressed branch (2Kb on a 32-bit system) while keeping the bytes/index low, in which case it's replaced by an uncompressed branch instead. In fact, this can happen **before** the linear branch overflows. Likewise, when a bitmap branch has sufficient population, or when the whole tree has a good enough bytes/index during insertion into a bitmap branch, it's converted to an uncompressed branch. (020130: Details are in flux and subject to further tuning.)

Similarly, when a linear leaf overflows it grows into a bitmap leaf, but only if the remaining undecoded index size is 1 byte; otherwise the leaf is either “compressed to a lower level” under a narrow(er) pointer, or else replaced by a new branch and “immediate” indexes (not otherwise discussed here) and/or more leaves. Meanwhile, a bitmap leaf cannot overflow; the bitmap can always represent 256 1-byte indexes.

2.2.4 Beyond Beating the Curve

When Doug “discovered” Judy by “reinventing” digital trees, he did not foresee where the path would lead into branch and leaf compression. However, he did realize early on that the UI/API to the Judy code would be remarkably simple -- “think of it as an array”.

As we developed Judy we set challenging goals for “balancing” many different aspects of Judy’s performance. We want Judy to be usable in an opaque way, in a wide range of applications, with no optimization or tuning required -- or perhaps even possible. Hence the following summary...

2.2.4.1 Judy Dynamic Ranges

Judy should be equally “good”, or nearly so, across all of the following:

- **Population**: 0, 1, ..., N, ..., “infinite”

  (or at least practically so, for a given machine word-size and memory)

  Judy supports small arrays as well as huge ones, although arrays known to be small, dense, and fixed-size are better implemented as simple arrays rather than in Judy. Use Judy to allow for a large or unpredictable population.

- **Population type**: Sequential .. clustered .. periodic .. random

  Behavior should be roughly equivalent regardless of the nature of the index set; in practice, within an order of magnitude for speed and space. In practice, random data is the worst case.

- **Performance**:

Table 3: Judy Time/Space Performance

		Average	Worst case
Time (usec/operation)	Insert/delete	“good”	“good”
	Retrieve	“excellent”	“good”
Space (bytes/index)		“excellent”	“good”

  TBD: Replace the ratings above with typical numbers.

- **Plus** sorting/searching!

  These features come “free” by the nature of Judy. Indexes are sorted in fixed orders that usually (but not always) are meaningful to the caller and allow rapid access in sorted order, including neighbor searching. This could be a desirable side-effect feature that hashing does not offer.

- **Plus** counting!

  Judy array counting capabilities allow fast solutions to problems in novel or unexpected ways, such as determining data density or stack depth. See for example the application note about the disk Work Load Analyzer on the external Judy website.

3. Smarter Digital Trees

The essence of Judy is to combine the advantages of the digital tree with smarter approaches to handling “branches”, that is, interior (non-terminal) nodes in the tree, and “leaves”, that is, terminal nodes in the tree. While this might seem obvious in retrospect, the path taken to discover the (hopefully) optimal data structure has been torturous and anything but obvious. For example, what seemed to work well for one type of data, such as clustered, did not work well for another, such as random. And, believe it or not, even if you understood all of the following, you probably **still** could not write the Judy code correctly; it’s very tricky. Other features beyond compression are also needed, such as, “always keep the tree in its least compressed form,” and, “write code that is portable but still runs fast.”

If you encounter any terms that confuse you, bear in mind the existence of ‘‘Glossary/Lexicon’’ on page 69. I can’t promise you’ll come back from there enlightened, but at least it might say the same thing in a different way and you can laugh at the inconsistencies.

3.1 Judy Compression Tricks

Judy uses a number of tricks to minimize both memory space and overall compute time, for both lookups and modifications (insert/delete), while attempting to keep worst-case (pathological) behavior acceptable for **any** (unpredictable) index set, and for any insertion/deletion to **any** index set. (See also ‘‘Beyond Beating the Curve’’ on page 24.) Most of the tricks used are forms of data compression. These tricks help Judy reduce space (memory required), minimize cache line fills required, and thus reduce execution time (the real goal) more than enough to make up for the additional CPU instructions required to support the tricks.

The text in this section was used to map out {some Judy patent applications}...

{Remainder deleted, sorry, you don’t need to read this to understand the released software.}

4. Judy IV Data Structures

Here’s a summary of the Judy IV data structures for Judy1 and JudyL for 32-bit [and 64-bit] systems. The real and current definition is in the Judy1.h, JudyL.h, and JudyPrivateBranch.h header files, but hopefully this presentation is easier to assimilate (even if it also happens to be wrong, becomes wrong over time... *“Half of what we taught you is wrong -- and we don’t know which half.”* -- Neifert)

Note: JudySL is “merely” a meta-trie built using JudyL arrays as branch nodes. See the related application note on the Judy external website.

This discussion proceeds more or less “top down” through the structures.

- Judy Array Pointer -- more commonly called a “root pointer”
- Root-Level Leaves -- for small arrays
- Judy Population/Memory Node (JPM) -- top node of larger arrays (trees)
- Judy Branches: Linear, Bitmap, Uncompressed -- the interior (non-terminal) nodes of larger trees
- Judy Pointer (JP) -- the “rich pointers” (also called subexpanse pointers) that populate all three types of branches:
    - Basic JP Data Structure
    - Decoding and Population
    - *JP Type* Field -- especially including Immediate Indexes
- Linear Leaves -- for populations too large for immediate indexes but otherwise relatively small
- Bitmap Leaves -- for level-1 leaves with high populations

4.1 Preliminary Notes

Miscellaneous notes before getting into the details...

Examples of Judy trees are in ‘‘Examples of Judy Trees’’ on page 67 as well as in this section.

Cache line size: Judy assumes a CPU cache line is always 16 words = 64 [128] bytes. This is not true on IA32 machines (cache line = 8 words) and possibly on some PA-RISC machines. On systems with smaller cache lines the code still works but not as fast. In general Judy tries to be cache-efficient without being too machine-dependent.

Tree level: Data storage trees are traditionally drawn with the levels numbered 1..N starting at the root of the tree. Judy trees are numbered bottom-up instead, meaning the root pointer is always at level 4 [8], for several reasons.

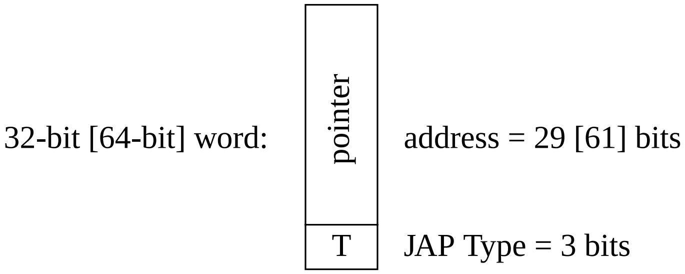
- The level of a node in the tree is equal to the number of undecoded index bytes remaining at that level.
- Similarly, the size of the *Population* field (in bytes) is equal to the level at which the field appears in a branch in the tree.
- Both 32-bit and 64-bit trees look the same in the lower 4 levels, which makes for much simpler common source code.
- This numbering convention works well for expanse-based digital trees, but would be unwieldy for purely population-based types of trees.

Judy1/JudyL: Remember, Judy1 maps indexes to “valid/invalid”, while JudyL maps indexes to one-word value areas (in addition to noting which indexes are valid).

4.2 Judy Array Pointer (JAP)

A **Judy Array Pointer** (root pointer) is 1 word = 4 bytes [8 bytes] containing an ordinary pointer (memory address) that points to a Judy array, except that it always references an object aligned to at least a 2-word = 8-byte [16-byte] boundary. Hence the 3 [or 4] least significant bits of the root pointer would always be 0, and are usable to encode the type of object to which the root pointer points. [In fact only the least 3 bits are needed and used, even on 64-bit systems.]

Figure 13: Judy Array Pointer (JAP)



The *JAP Type* field is defined as follows:

Table 4: Judy Array Pointer (JAP) Type values

JAP Type	Meaning
0	if address is also 0, an empty Judy array; otherwise invalid root pointer
1	JudyL root-level leaf containing exactly 1 index
2	JudyL root-level leaf containing exactly 2 indexes
3	JudyL root-level leaf with a *Population* word and >2 indexes
4	Judy1 root-level leaf containing exactly 2 indexes, **or** an invalid JudyL root (array) pointer

Table 4: Judy Array Pointer (JAP) Type values

5	JudyL top-level branch (L, B, or U under a JPM, see ‘‘Judy Population/ Memory Node (JPM)’’ on page 31)
6	Judy1 root-level leaf with a *Population* word and 1 or >2 indexes
7	Judy1 top-level branch (L, B, or U under a JPM, see ‘‘Judy Population/ Memory Node (JPM)’’ on page 31)

The *JAP Type* values are distributed so as to maximize the odds of detecting inappropriate use of a Judy array pointer by a caller, such as passing a Judy1 pointer to a JudyL function.

The “invalid JudyL root pointer” value (4) is reserved (see JLAP_MASK and JLAP_INVALID in Judy.h) to allow applications to construct arbitrary-shaped trees (meta-tries) of JudyL arrays (or even JudySL arrays, but not both mixed in one meta-trie) where each array value can either be a pointer to another Judy array or to some other non-Judy object defined by the application. In fact this feature is used by JudySL to construct trees of JudyL arrays including pointers to “shortcut leaves” in place of subsidiary JudyL arrays.

020130: Reader feedback: “...the different cases that you optimize for are very quirky. Why do you spend so much effort to optimize for root-level leaves with only 1 and 2 indices? I would have thought that using the JAP Type values for other variations of nodes would have been more beneficial...”

-- To make it possible to have huge numbers of small arrays, with good memory efficiency. However, this depends on the nature of the underlying memory manager and is still debatable. We might get rid of them in the future.

4.3 Root-Level Leaves

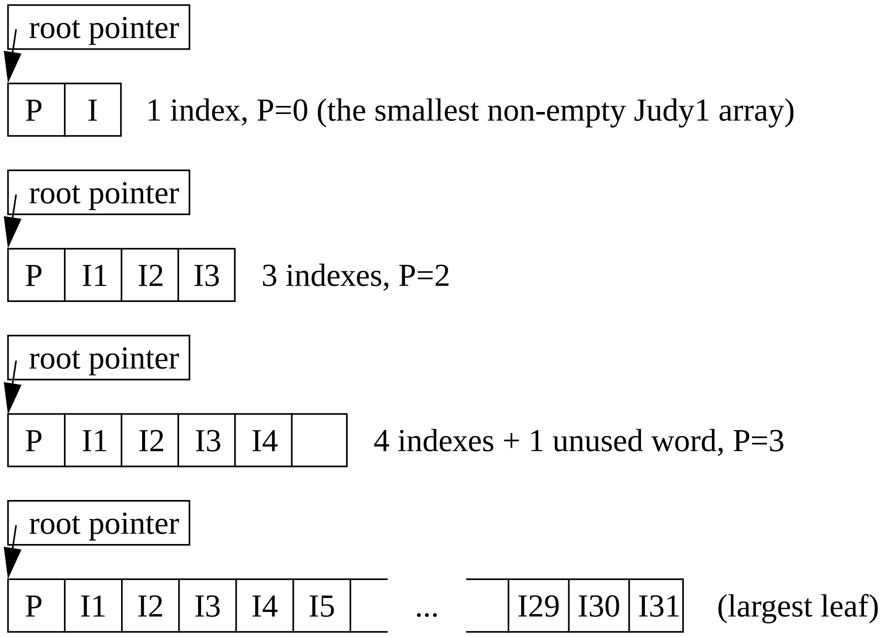
If a Judy1 array has a small enough population, it can fit into a single **root-level leaf**, up to 2 cache lines (16 * 2 = 32 words) in size, that typically contains a *Population* (P) word plus up to 31 additional words, each holding a 4-byte [8-byte] index.

Note: Technically a root-level leaf is a type of linear leaf aside from the population word, but by convention we refer to root-level leaves separately. In some cases you might read, “root-level linear leaf”, which is just a fancier name for a root-level leaf.

Note: Population values are always stored minus 1 from the actual value because the actual value can range 1..2^N, and 0..2^N-1 fits in an N-bit field. In the code we use the variable names “pop0” for the minus-1 value and “pop1” for the actual value to avoid confusing them.

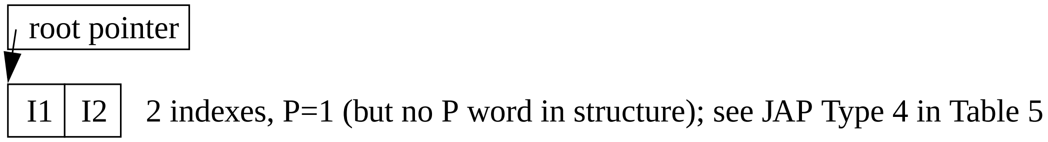
Here are examples of Judy1 root-level leaves, pointed at by root pointers, where each small box is 1 word:

Figure 14a: Judy1 Root-Level Leaves



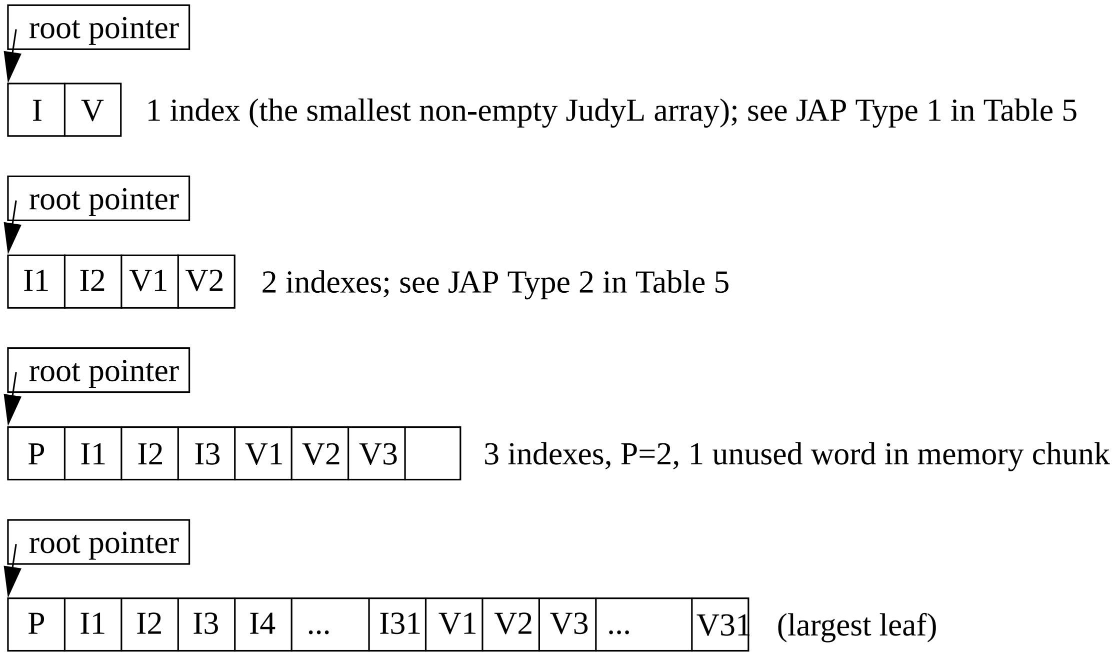
To support large numbers of small Judy1 arrays, the 2-index root-level leaf is special. The Judy memory manager code issues memory chunks in units of 2, 4, 6, 8, 12, 16, 24, 32... words to minimize fragmentation. Hence both 1-index and 2-index Judy arrays are stored in a 2-word root-level leaf (the smallest memory chunk). The 2-index root-level leaf contains no *Population* word, so instead the JAP's *Type* field specifies "2 indexes":

Figure 14b: Judy1 Root-Level Leaf with Population 2


root pointer
I1
I2
2 indexes, P=1 (but no P word in structure); see JAP Type 4 in Table 5

The preceding pictures show root-level leaves without value areas, that is, for Judy1 arrays. JudyL arrays must associate a value area with each index. Hence JudyL root-level leaves have value areas, are allowed to grow to 4 cache lines rather than 2, and it makes sense to distinguish a 1-index leaf, as well as a 2-index leaf, from a multi-index leaf.

Figure 14c: JudyL Root-Level Leaves


root pointer
I
V
1 index (the smallest non-empty JudyL array); see JAP Type 1 in Table 5
root pointer
I1
I2
V1
V2
2 indexes; see JAP Type 2 in Table 5
root pointer
P
I1
I2
I3
V1
V2
V3
3 indexes, P=2, 1 unused word in memory chunk
root pointer
P
I1
I2
I3
I4
...
I31
V1
V2
V3
...
V31
(largest leaf)

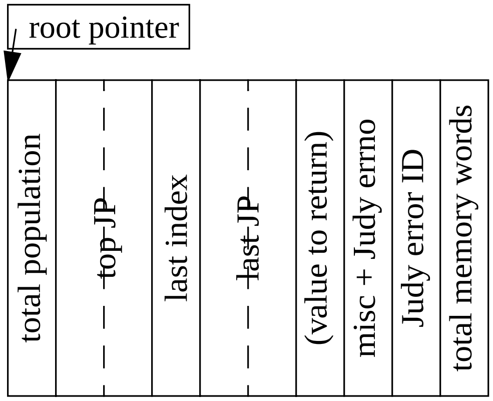
Note: Value areas are actually offset to an aligned location, meaning some unused words can exist between indexes and values. In general, alignment is used so fewer bytes must be moved during most insertions or deletions. That is, the insertions or deletions happen “in place” where possible.

4.4 Judy Population/Memory Node (JPM)

A **Judy Population/Memory** node is used when a Judy tree’s population is large enough to support the JPM by “**amortizing**” the memory required for it over a large enough population of indexes. Once a root level leaf fills (exceeds 31 indexes), the root pointer always points to a JPM instead of a root-level leaf or a branch, and the JPM in turn points to a linear, bitmap, or uncompressed branch. -- Why not directly to a leaf? A leaf of full-word indexes up to the allowed size (2 cache lines) would simply be a root-level leaf, and for any lower-level leaf to reside directly under a JPM it would have to be under a narrow pointer, but a JPM cannot contain a narrow pointer because there aren’t enough bytes in a JP (see ‘‘Basic JP Data Structure’’ on page 38), even in the JP within the JPM, to hold *Decode* plus *Population* bytes, and for code efficiency the JPM’s JP is treated like all others. This will become clearer as you read on.

The structure of a JPM is as follows, where each narrow rectangle is 1 word.

Figure 15: Judy Population/Memory Node (JPM)



The only field in the JPM absolutely required to make the tree work is the *JP Type* subfield in the *top JP* field. See ‘‘Judy Pointer (JP)’’ on page 38 for details about JPs. The rest of the fields in the JPM support simpler and more efficient tree traversal and modification software.

- The *total population* field prevents having to add up the population (of all the JPs) in a top branch.
- The *top JP* field (2 words) allows entry to tree traversal code with a JP always available, even at the top level of the tree. However, not all subfields or JP types in this *top JP* are used; see ‘‘Basic JP Data Structure’’ on page 38.
- The *last index*, *last JP*, and *misc* fields (the last of which includes last-offset and last-pop0 subfields not shown above) support stateful (more-efficient due to prior knowledge) repeated calls of the search functions (in progress, not yet checked in as of this writing). See the Judy1.h and JudyL.h header files for details.
- The *Judy errno* and *Judy error ID* fields are used to pass error information up from a lower level on the way to returning them through a caller-supplied PJError_t object. Note: On 64-bit systems the error ID might pack into the same word as misc + Judy errno because there are more bytes available per word; to be determined.
- The *value to return* field only exists for JudyL. Similarly, it is a way of getting a value area pointer back from a low level of recursion without passing a parameter separate from the JPM.
- The *total memory words* field is used to rapidly assess the overall memory efficiency of the tree (bytes/index) to support opportunistic branch decompression, and is also used to cross-check the Judy1FreeArray() and JudyLFreeArray() functions. Note: This field is placed “late” in the structure in case of a machine with an 8-word cache line, because this field is not so “hot” as the others.

Additional fields used for statistics, etc. could exist (in the future) to the right of the fields shown above but are not shown here.

Note that the *JAP Type* field described earlier is different from the *JPM* (and *JP*) *Type* field described in ‘‘JP Type Field’’ on page 40. However, the numeric *Type* values are disjoint and could coexist.

4.5 Judy Branches

Below a JPM, a Judy tree consists of some combination of branch nodes (linear, bitmap, or uncompressed), which are illustrated here, and leaf nodes (linear or bitmap), which are illustrated starting at ‘‘Linear Leaves’’ on page 44.

Each branch is a literal (uncompressed) or virtual (linear or bitmap) array of 256 **Judy Pointers** (JPs). That is, node fanout is 256, and a Judy tree is a 256-way digital tree. Why 256? Ultimately it’s because computers access and handle bytes more efficiently than other-sized bit fields. Indexes are broken into digits that happen to also be bytes. This is reflected in the remainder of the following diagrams. Also, even though a digital tree could have a variety of different fanouts at different levels or branch nodes, “there’s no profit in it” (as the Ferengi would say).

4.5.1 Linear Branches

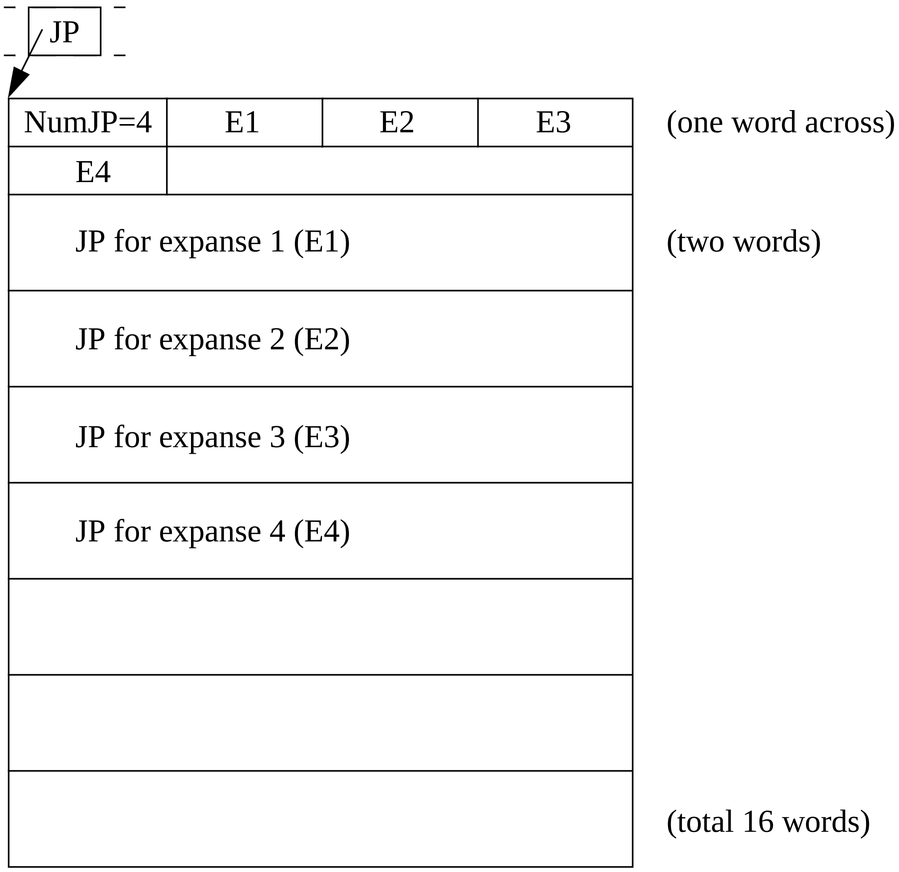
A linear branch node is used when the actual fanout, that is, the number of populated subexpanses, is relatively small, in practice, up to 7 JPs (out of 256 subexpanses per branch). A linear branch, as illustrated in the next figure below, consists of three consecutive regions:

- count of populated subexpanses (JPs);
- sorted list of populated subexpanses (digits, each 1 byte);
- list of corresponding JPs (2 words each).

A maximum linear branch consisting of 7 JPs takes 1 byte for the number of subexpanses and 7 bytes for the subexpanse list, hence 2 [1] words for the combination, followed by 14 words for the JPs, fitting in 16 words = 1 cache line total. (In the current implementation, for speed and simplicity of insertion and deletion, a linear branch is always allocated 16 words, even when it contains only 1 JP and could fit in just 4 words.)

Here’s a picture of a Judy linear branch with 4 populated subexpanses (JPs) on a 32-bit system. Each narrow rectangle all the way across the diagram represents 1 word of memory. The “E” (subexpanse list) fields are single digits, in the sense of a digital tree digit (although here each digit is a byte that takes two hexadecimal digits to represent), from the valid indexes representing the populated subexpanses. Blank rectangles are memory (1 or more bytes) unused when NumJP=4, which become used for larger NumJP = 5..7.

Figure 16a: Example of Judy Linear Branch (32-bit)



For example, it might be that E1 has the value 47 (hex), E2 = 9A, E3 = A5, and E4 = FD. In this case all of the indexes in this linear branch would have an Nth byte (where 5-N [9-N] is the level of the branch in the tree) that is either 47, 9A, A5, or FD; and there are no valid indexes with other values in that byte.

Note that the Judy insert functions also opportunistically convert linear branches to uncompressed branches when either the population under a top-level branch, or the overall memory efficiency of the tree in bytes per index for a lower-level branch, support doing so, in order to save access time. 020131: This is in flux and subject to revision.

4.5.2 Bitmap Branches

A bitmap branch node is used when the actual fanout of (that is, populated subexpanses under) a branch node exceeds the capacity of a linear branch but substantial memory can be saved compared to using an uncompressed branch.

Note that the Judy insert functions also opportunistically convert bitmap branches to uncompressed branches when either the population under a top-level branch, or the overall memory efficiency of the tree in bytes per index for a lower-level branch, support doing so, in order to save access time. 020131: This is in flux and subject to revision.

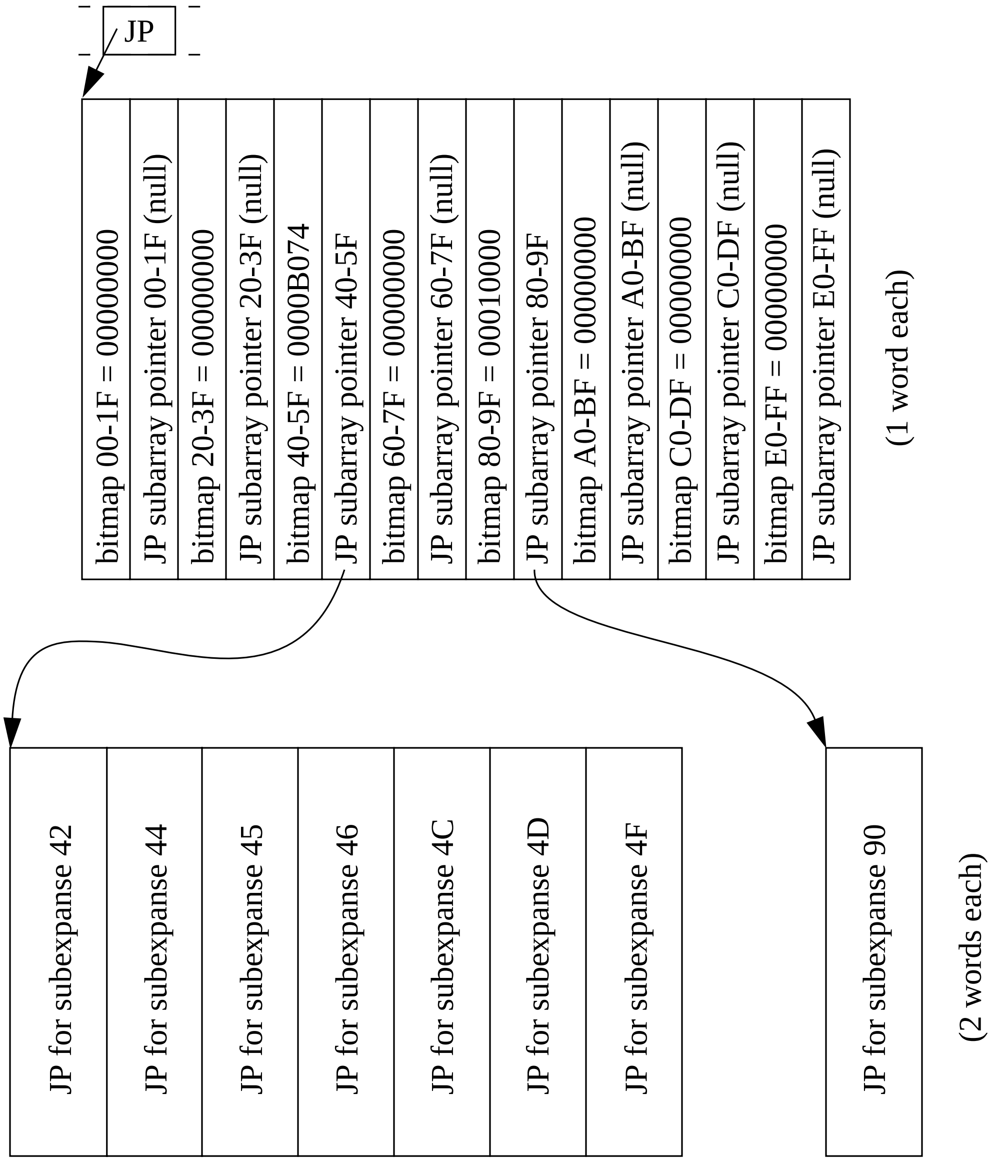
A bitmap branch is a 2-tier object more complex than a linear or uncompressed branch. The first level is the bitmap (always 256 bits = 32 bytes, on both 32-bit and 64-bit systems), subdivided into 8 subexpanses, interspersed with 8 corresponding ordinary pointers to second-level JP subarrays. Each JP subarray consists of a (packed) linear list of JPs, one JP for each bit set in the bitmap. On a 32-bit system that’s 32/4 = 8 words for the bitmap and 8 words for the pointers = 16 words = 1 cache line. [On a 64-bit system that’s 32/8 = 4 words for the bitmap and 8 words for the pointers with 4 words (8 half-words) wasted = 1 cache line.] Any empty 32-bit subexpanse of the bitmap has a corresponding null pointer and empty subarray.

Why use a 2-tier object? Otherwise the JP array (which could grow to 256 JPs = 512 words) becomes too large for fast insertions and deletions.

020130: Previously the bitmap was contiguous and preceded the 8 pointers. This was fine for systems with 16-word cache lines, but resulted in as many as 3 cache fills on systems with 8-word cache lines. Doug realized this could be avoided by interspersing the bitmaps and subarray pointers, which is what is now illustrated below.

Here’s a picture of a 32-bit Judy bitmap branch containing 8 populated subexpanses (JPs) that fall in 2 32-bit subexpanses of the bitmap. Each narrow vertical rectangle represents 1 word of memory and each wider vertical rectangle is 2 words.

Figure 16b: Example of 32-bit Judy Bitmap Branch



In this example, bitmap 40-5F = 0000B074, which looks like this in binary:

Figure 16c: Example of Judy Bitmap Branch Subexpanse

5F	5E	5D	5C	5B	5A	59	58	57	56	55	54	53	52	51	50	4F	4E	4D	4C	4B	4A	49	48	47	46	45	44	43	42	41	40
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	0	0	0	1	1	1	0	1	0	0
0				**0**				**0**				**0**				**B**				**0**				**7**				**4**			

Note: Late in Judy IV development, we realized that once a bitmap branch JP subarray grows to the full amount of memory it can occupy, it would be faster to access and modify the subarray if it were uncompressed. This means setting all of the bits in the corresponding subexpanse of the bitmap, even for subexpanses of indexes which are unpopulated; unpacking the JP subarray to be a simple, positionally-accessed array; and representing unpopulated index subexpanses with null JPs. However, this is not yet implemented. Currently null JPs can only occur in uncompressed branches.

Note: In Figure 16b above, the bitmap and subarray pointer for 80-9F points to a subarray containing a single JP. If this JP contained a single, immediate index, it would be possible to move the JP from the subarray into the bitmap branch itself, replacing the bitmap + pointer words. This has been considered but not yet implemented.

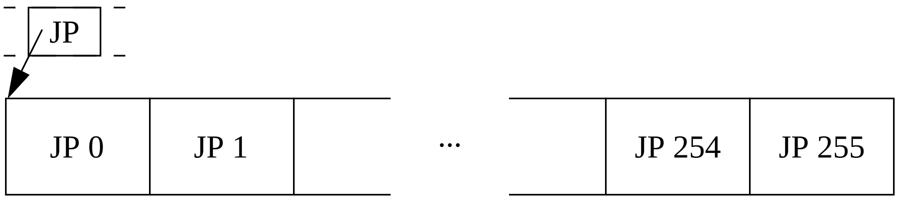
Note: The aspect ratio, currently 8 subexpanses of 32 JPs each, is macroized into the code; see BITMAP_BRANCH* and BITMAP_LEAF*, also NO_BRANCHU. Using an aspect ratio with fewer bits per bitmap than bits per word, such as 16 on a 32-bit system or 32 on a 64-bit system, opens the possibility of inserting a subexpanse population count into each bitmap word in a bitmap branch, at least for smaller populations, thereby speeding up counting because it would require fewer cache fills.

4.5.3 Uncompressed Branches

A Judy uncompressed branch is merely an array of JPs, in this case 256 JPs, with null JPs for empty subexpanses. At 2 words per JP, that takes 512 words or 2Kb [4Kb].

Here’s a picture of a Judy uncompressed branch. Each rectangle represents a 2-word object (JP).

Figure 16d: Judy Uncompressed Branch



4.6 Judy Pointer (JP)

Each JP occupies 2 words = 8 bytes [16 bytes]; that is, the digital tree’s branches are (literal or virtual) arrays of 2-word objects.

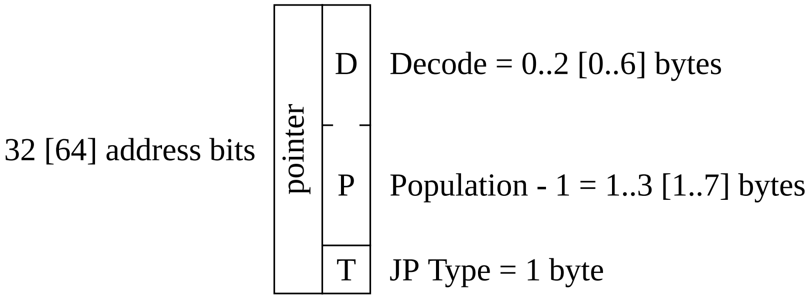
Each JP identifies the type of object to which it points, and can also “narrow the expanse” of that object while doing so (see ‘‘Decode and Population Fields’’ on page 39).

You might wonder why a JP is always 2 words in size, even at lower levels of a Judy tree where you might think you wouldn’t need that many bytes, when the *Population* field (see below) gets smaller, especially when no narrow pointers (also see ‘‘Decode and Population Fields’’ on page 39) are involved. It turned out to be simpler and more CPU-efficient to use a fixed-size “rich pointer” (JP), even at the cost of some “wasted space” especially at lower levels in the tree. This extra space turns out to be useful for corruption detection in that it can hold meaningful, though redundant, data.

4.6.1 Basic JP Data Structure

The basic JP data structure is 2 32-bit [64-bit] words structured as follows. Each narrow vertical rectangle represents 1 word.

Figure 17a: Judy Pointer (JP), Null or Pointing to Branch or Leaf Node

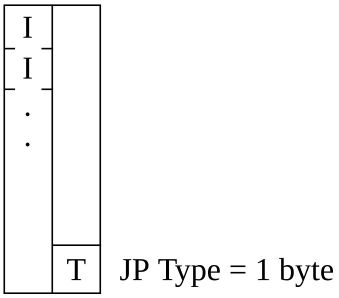

32 [64] address bits
pointer
D
Decode = 0..2 [0..6] bytes
P
Population - 1 = 1..3 [1..7] bytes
T
JP Type = 1 byte

For a null JP all bytes except the *Type* field are zero. Otherwise the first word is a pointer to (that is, address of) a subsidiary branch or leaf node. The *Decode* (decoded index) and *Population* fields together fill all but 1 byte of the second word; see ‘‘Decode and Population Fields’’ on page 39.

Note: Since a JP’s pointer field always points to an object at least 2 words in size, the least 3 bits of the pointer [or more on 64-bit systems] are always zero, but we found no need, ultimately, to pack any information into that space. Perhaps in a future implementation, we will, such as to mark a relative rather than absolute address.

When a JP is used to hold “immediate indexes” instead of pointing to some other node, it generally looks like this instead:

Figure 17b: Judy Pointer (JP) Containing Immediate Indexes



Note the presence of a *JP Type* field of the same size and in the same place as in any other JP. However, the remaining bytes of the JP are used to hold immediate indexes, and for JudyL, also either a value area or a pointer to a “value areas leaf”. See ‘‘Immediate Indexes JP’’ on page 41.

Note: The *top JP* field in a JPM always contains a valid pointer (to a top branch node) and an appropriate *JP Type* field, but the *Decode* and *Population* fields are null and unused. No bytes are decoded above the top level, and the *Population* field is too small to hold the entire array population, which is why it’s a separate field in the JPM. Also note that the *top JP* cannot contain a narrow pointer, that is, it always points to a top-level branch.

4.6.2 Decode and Population Fields

The *Decode* and *Population* fields share a common data field (1 word less 1 byte). (Note: This design depends on the fact that Judy1 and JudyL support fixed-size indexes of size 1 word.)

A normal JP always resides in a branch, below a JPM, so at least 1 digit (1 byte) of any index is already decoded in the course of accessing the JP. Hence there can be at most 4 - 1 = 3 [8 - 1 = 7] index bytes left to decode, and 2^24 [2^56] valid indexes (population) in the JP’s subexpanse.

Under any JP, at least 1 more digit (1 more byte) is always decoded at the leaf level, so at most 4 - 1 - 1 = 2 [8 - 1 - 1 = 6] next leading bytes could be **common** between all of the valid indexes in the JP’s subexpanse. The common bytes themselves must be stored in the *Decode* field, and the JP can be thought of as a “narrow-expanse pointer”.

In practice it’s more efficient to always store in the *Decode* field all the bytes (except the first one) decoded so far, not just the common bytes needed for a narrow pointer when applicable (that is, when the child node is more than one level below the parent branch node containing the JP). When a JP is not narrow, the *Population* field’s size is equal to the level of the subsidiary node under the pointer (counting up from the bottom of the tree), exactly 1 level below the branch in which the JP resides. When the JP is narrow, the *Decode* field is accordingly larger and the *Population* field is smaller, again matching the level of the subsidiary node (2 or more levels below the branch containing the JP). Note that the *Population* field is always at least 1 byte, even in a level-2 branch's JPs (which point to level-1 leaves).

Saying it differently, the boundary between the *Decode* and *Population* fields "floats". Lower in the tree the *Decode* field is larger and the *Population* field is smaller. Note how 3 [7] bytes are always sufficient to hold both the *Decode* bytes and the *Population* bytes -- in a normal JP. The *top JP* in the JPM doesn't decode 1 byte "off the top" and it cannot support a narrow pointer, so it has no *Decode* bytes, and the *Population* field takes a full word (separate from the *top JP*). As the number of common leading bytes increases, the subexpanse and the possible remaining population, hence the size of the *Population* field, decrease at the same rate.

Note: Common bit compression is done in units of bytes because this is the digit size for a 256-way digital tree.

4.6.3 JP Type Field

A JP can point or otherwise refer to only one of five broad classes (types) of subsidiary nodes:

- null (empty subexpanse)
- branch (linear, bitmap, or uncompressed)
- leaf (linear or bitmap)
- immediate indexes
- full population (Judy1 only)

However, the actual list of JP types is quite long (too long to put in a table here), and variable between Judy1 and JudyL, and also between 32-bit and 64-bit implementations. The reason is that the *JP Type* field is large enough (1 byte = 8 bits = 256 values) to enumerate every possible subsidiary node type, including the level at which the node resides, or for immediate indexes, the bytes per index (that is, its level) and the number of indexes present. See the Judy1.h and JudyL.h header files for the enumerations of the JP types.

We are aware that the use of a wide variety of node types is "academically impure" (in that many academic papers frown on pointers pointing to varieties of child node types), but it seems necessary for adaptability and performance, and the C language supports clean casting of pointer types.

Using a single large enumeration in a single *JP Type* field, rather than a collection of disjoint data fields, means the code that traverses or modifies a Judy tree can be written as a **state machine** whose inputs are the index being processed plus the nodes of the tree. The hallmark of a state machine is iteration or recursion through one large switch (or case) statement. In fact the Judy code for each API function largely consists of a single large switch statement with many small code chunks (cases) that are very specific to the operation being performed, such as, "insert an index in an immediate index JP already containing 2 indexes of 2 bytes each." Variations that might be computed at run-time are instead precomputed at compile time for efficiency.

For all but null and immediate JPs, before traversing through the JP's pointer field to the subsidiary node, the parent JP can decode some index bytes by use of the *Decode* field, thereby narrowing the expanse of the branch, leaf, or full expanse to which the JP points.

4.6.3.1 Null JP

Currently **null JP**s are only valid in uncompressed branches. The JP lists in linear and bitmap branches are "packed" rather than positional, that is, empty subexpanses are absent rather than being represented by null JPs. Note: We have an idea to create "uncompressed bitmap branch JP subarrays" for faster performance at no memory cost, which would change this.

4.6.3.2 Branch JP

A **branch JP** always lives in a JPM's *top JP* field or in a branch, and points to an (or another) branch node.

4.6.3.3 Leaf JP

A **leaf JP** always lives in a branch and points to a non-root-level linear or bitmap leaf node.

4.6.3.3.1 Bitmap Leaf JP

A **bitmap leaf JP** is used to reference a level-1 leaf containing, for Judy1 >= 25 [16] 1-byte indexes, or for JudyL >= 26 1-byte indexes. For Judy1, at this population the memory chunk size goes from 6 words to 8 words [2 words to 4 words] = 32 bytes = 256 bits. A bitmap turns out to be faster for set and test operations than a linear leaf, as well as cheaper in memory, for larger leaves.

For 32-bit JudyL, 26 indexes is the point where the leaf would grow from 2 to 3 cache lines. The same transition population is used for 64-bit JudyL in lieu of a better answer. The JudyL bitmap leaf structure is 2-tier (see ‘‘Bitmap Leaves’’ on page 47) and its layout depends on the distribution of the indexes, so a "correct" transition population is not a simple static number but would depend on the indexes stored, but it's not worth computing it in real time. -- As you can see, there are many subtle issues in the Judy design. "*People who deal with bits should expect to get bitten.*" -- Jon Bentley

4.6.3.4 Immediate Indexes JP

An **immediate indexes JP** essentially includes a small ("shortcut multi-index") leaf, but to avoid confusion we refer to it as an "immediate JP" or "immediate indexes" rather than an "immediate leaf". Immediate JPs are used to represent sparsely populated expanses where the indexes reside in the JP itself, in one of the following combinations of index counts (populations). These counts are just the numbers of N-byte indexes that can be packed into 2 words less 1 byte for Judy1, or 1 word less 1 byte for JudyL. (Don’t panic, drawings follow...)

Table 5: Populations of Immediate JPs

Judy1		**JudyL**		
32-bit	**[64-bit]**	**32-bit**	**[64-bit]**	**Index Size**
	[1..2]		[1]	[7-byte indexes]
	[1..2]		[1]	[6-byte indexes]
	[1..3]		[1]	[5-byte indexes]
	[1..3]		[1]	[4-byte indexes]
1..2	[1..5]	1	[1..2]	3-byte indexes
1..3	[1..7]	1	[1..3]	2-byte indexes
1..7	[1..15]	1..3	[1..7]	1-byte indexes

Immediate indexes are packed into immediate JPs starting at the “first byte” (farthest from the *Type* field), possibly leaving some unused bytes. However, in some cases the “first byte” is in the first word of the JP and in other cases it’s in the second word:

- For Judy1, there are no value areas associated with indexes, so all but 1 byte (the *Type* field) of each JP is available to hold index bytes, and the “first byte” is in the first word.
- For JudyL, however, the first word of the JP is either the value area for a single index or a pointer to a “values only leaf” for multiple immediate indexes, and the “first byte” is in the second word.
- Furthermore, when an immediate JP contains only a single index, no matter the index size (remaining undecoded bytes), there’s always space to store all but the first byte of that index -- in the second word. It turns out that this odd encoding makes index insertion and deletion much simpler, so it’s used for both Judy1 and JudyL.

Note: The structure of a linear leaf and the indexes portion of an immediate JP are identical once the starting address, index size, and population are known.

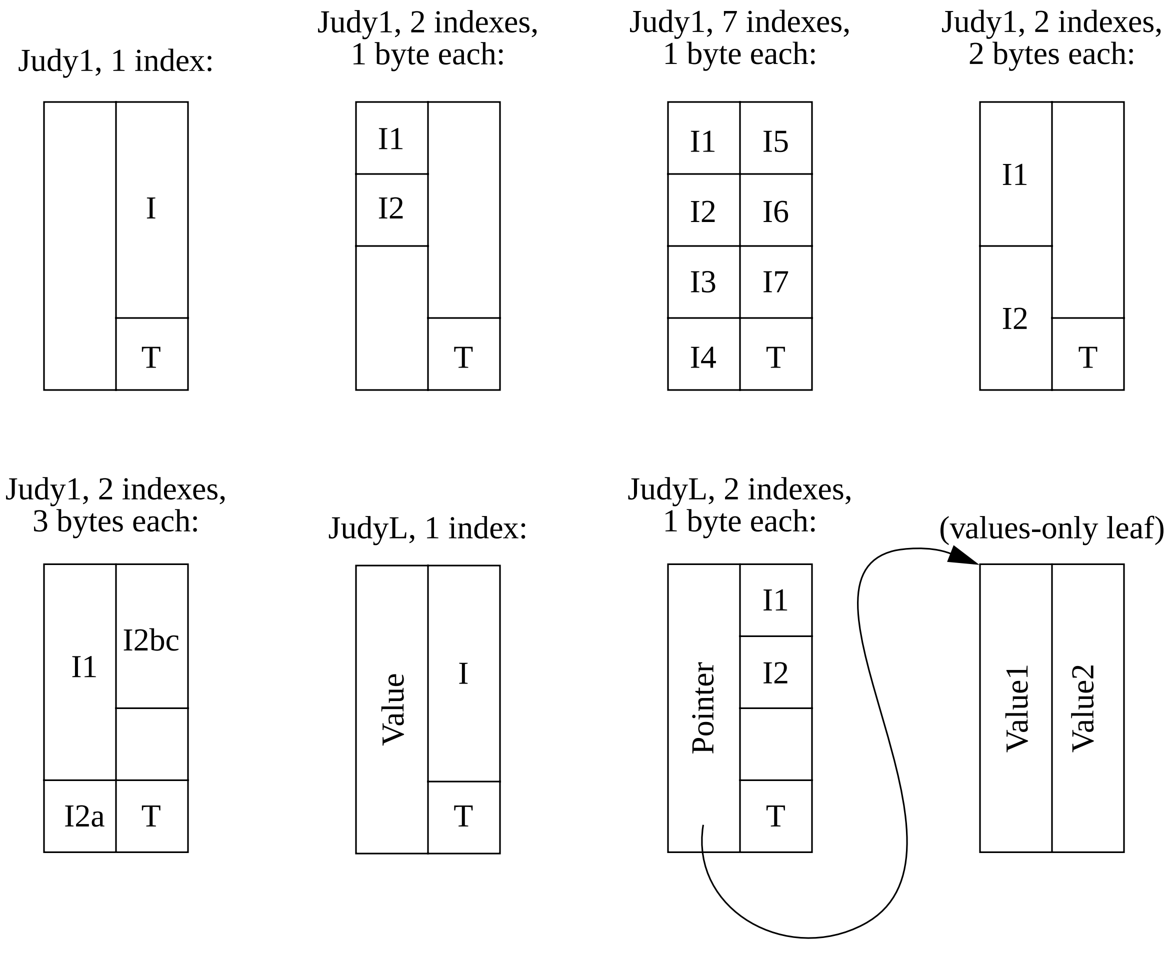
Following are some examples. In each case:

- The drawing (next figure below) is “scaled” for 32-bit, but extends as you would expect for 64-bit.
- “I” represents the N least significant bytes of an index; if more than one index is stored, the indexes appear in sorted order.
  - Note: For a solitary index, all but the first byte of the index is stored, regardless of the level in the tree.

  - Note: The current Judy IV implementation enumerates null and solitary-index immediate JPs by level, but since the JP format is identical for all single-index JPs this is not strictly required, merely handy when converting to/from multiple immediate indexes.

- The *JP Type* field (“T”) occupies 1 byte and encodes (enumerates) the fact that this is an immediate JP, how many indexes are present, and how many bytes are in each index (the index size). Presently we encode the index size even for solitary indexes, as noted previously, but we’ve realized that is an unnecessary complexity, and we might do away with it.
- Narrow vertical rectangles represent 1 word, and empty (unlabeled) rectangles represent unused bytes.

Figure 18: Examples of 32-bit Immediate JPs


Judy1, 1 index:
I
T
Judy1, 2 indexes, 1 byte each:
I1
I2
T
Judy1, 7 indexes, 1 byte each:
I1
I5
I2
I6
I3
I7
I4
T
Judy1, 2 indexes, 2 bytes each:
I1
I2
T
Judy1, 2 indexes, 3 bytes each:
I1
I2bc
I2a
T
JudyL, 1 index:
Value
I
T
JudyL, 2 indexes, 1 byte each:
Pointer
I1
I2
T
(values-only leaf)
Value1
Value2

“The difficult we do immediately; the impossible takes a little longer.”

020130: Reader feedback: “...for the Judy1 case where there are many indices clustered together, you might use a variant of the ‘expanse spans’ I mentioned above. Instead of listing each 1-byte index, you might be able to list ‘index/bitmap’ pairs. These would be a 1-byte index followed by a

1-byte bitmap of the 8 indices starting at that location. These might allow you to use immediate JPs more often...”

-- Right, but again, it’s a CPU versus space tradeoff to do any further compression. Earlier versions of Judy **did** have more complex encoding schemes... And we gave them up as being a net loss.

4.6.3.5 Full Expanse JP

A **full expanse JP** (also referred to as just a “full JP”) represents a subexpanse with a full population, that is, where all of the indexes in the subexpanse are valid. The JP’s pointer field is null. The *Decode* and *Population* fields are employed as usual to specify the location and size of the full expanse. Note that a full expanse can effectively reside under a narrow pointer. For example, a 32-bit Judy1 top-level linear branch might contain 2 JPs, one containing immediate indexes and the other a full JP with 2 *Decode* bytes and 1 *Population* byte with the value 0xff (= 256 - 1).

Full JPs only save significant memory at the lowest level of the tree (where there is 1 byte to decode), so they are only used at that level, although they can appear in a higher-level branch under a narrow pointer. Also, they only save significant memory for Judy1, so they are not used with JudyL.

4.7 Linear Leaves

In principle a digital tree ends in single-index leaves. In practice Judy arrays use multi-index leaves to save time and space.

A Judy1 linear leaf is simply a packed array of indexes that for each index stores (in sorted order) only the minimum number of bytes remaining to be decoded at the leaf’s level in the tree (below the root level), similar to immediate indexes described above. A JudyL linear leaf adds a value area (1 word) for each index, in a separate, corresponding list at a higher address than the index list.

Linear leaves have no *Population* (index count) field, unlike a root-level leaf. The parent JP carries the population for the leaf, unlike a root-level leaf where there’s no room in the root pointer for a *Population* field and it must be stored in the leaf itself.

A linear leaf contains one of the following populations of indexes.

Table 6: Populations of Linear Leaves

Judy1		**JudyL**		
32-bit	**[64-bit]**	**32-bit**	**[64-bit]**	**Index Size**

Table 6: Populations of Linear Leaves

	[3..36]		[2..34]	[7-byte indexes]
	[3..42]		[2..36]	[6-byte indexes]
	[4..51]		[2..39]	[5-byte indexes]
	[4..64]		[2..42]	[4-byte indexes]
3..42	[6..85]	2..36	[3..46]	3-byte indexes
4..64	[8..128]	2..42	[4..51]	2-byte indexes
8..24	(see below)	4..25	[8..25]	1-byte indexes

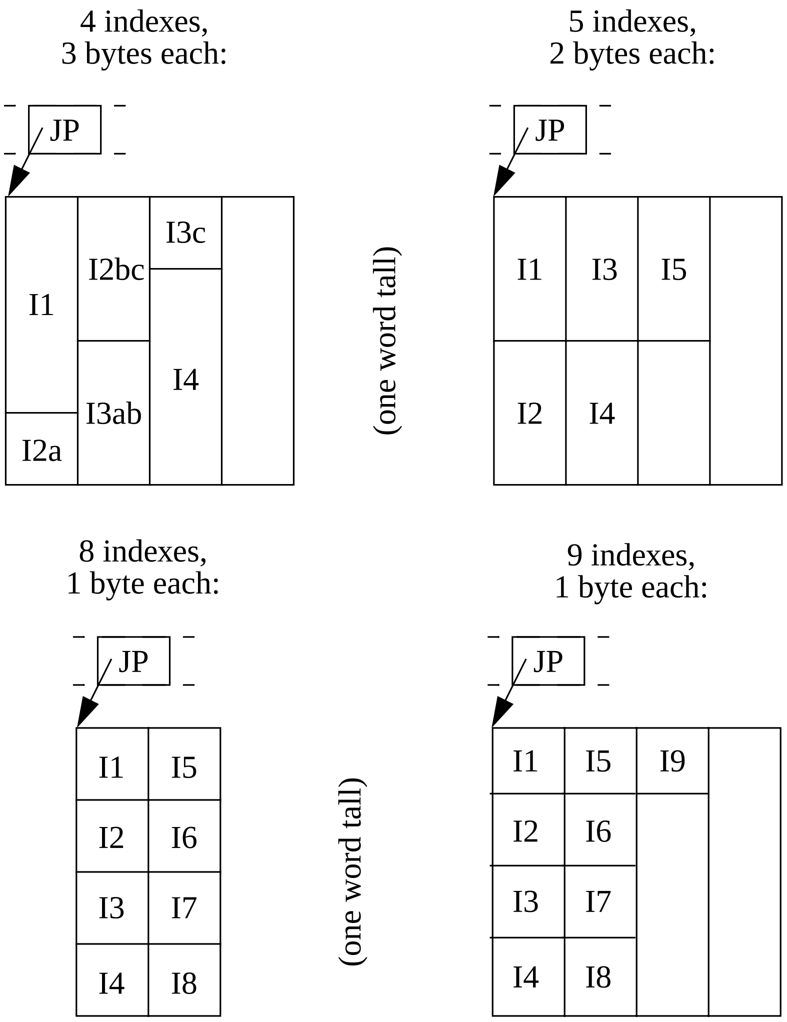
Notes:

- In each case the leaf's **index size**, that is, the number of remaining undecoded bytes in each index, is enumerated in the *JP Type* field as described earlier.
- The **minimum** leaf populations are based on how many indexes an **immediate** JP can hold. That is, smaller populations are immediatized (see "Immediate Indexes JP" on page 41).
- The **maximum** leaf populations are based on the capacity of 2 cache lines (32 words) for Judy1 linear leaves or 4 cache lines (64 words) for JudyL linear leaves. The maximum number is the point after which either a lower-level leaf under a narrow(er) pointer is used, a new branch node is inserted (for 2-byte and larger indexes), or the linear leaf is replaced by a bitmap leaf (for 1-byte indexes)

  [Note: The 64-bit Judy1 implementation switches directly from immediate indexes to a bitmap leaf upon reaching 16 indexes, to avoid creating a linear leaf for a single population size and then a bitmap leaf upon the next insertion, reaching 17 indexes, in the same subexpanse.]

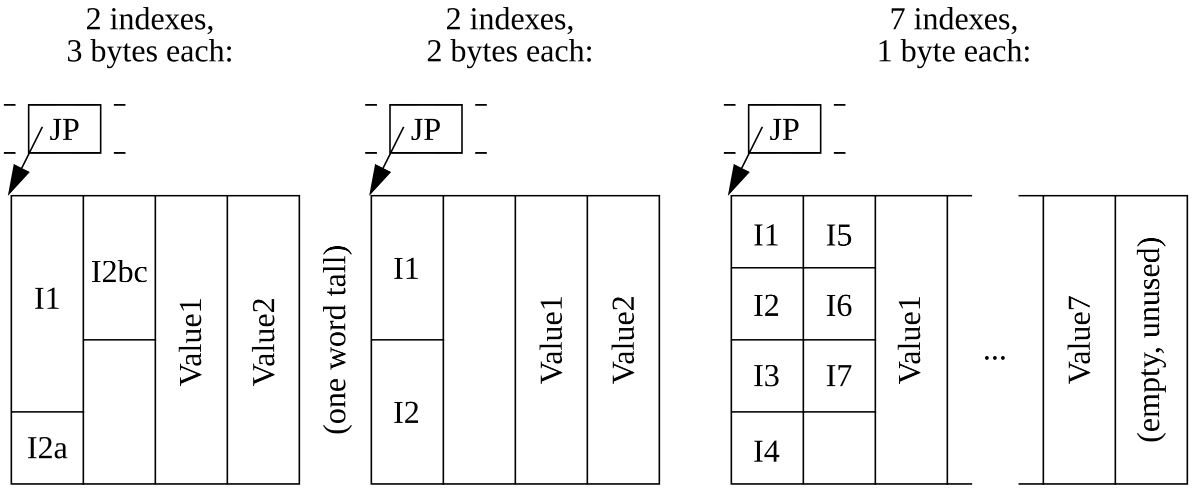
Here are examples of some linear leaves. Each narrow vertical rectangle represents 1 word and empty rectangles represent unused bytes. Note that nearly every linear leaf is at least 4 words in size because the next smallest allocatable memory chunk is 2 words, and any linear leaf that could fit in 2 words (less 1 byte for the *JP Type* field) would instead be stored as immediate indexes. The single exception is 8 1-byte indexes, as shown below.

Figure 19a: Examples of 32-bit Judy1 Linear Leaves



JudyL requires a value area corresponding to each stored index. Each value area requires 1 word. Values areas begin at an aligned location at a higher address than the index list so most insertions or deletions occur "in place" with minimum bytes moved. JudyL linear leaves are always at least 4 words in size, but the boundary with immediate indexes is a bit trickier than for Judy1. JudyL immediate indexes are used when the index bytes fit in 1 word (not 2 words) less 1 byte for the *JP Type*; otherwise a linear leaf is used.

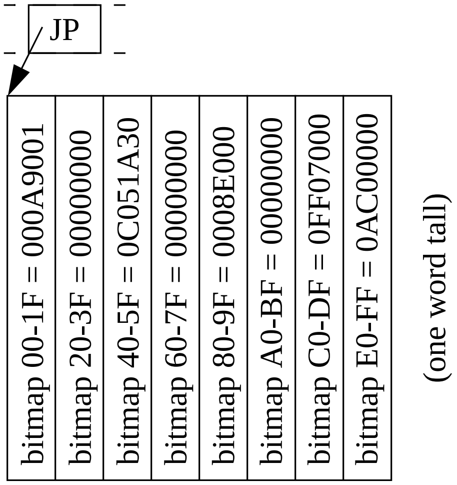
Figure 19b: Examples of 32-bit JudyL Linear Leaves



4.8 Bitmap Leaves

At the lowest level of the tree, where there is only a single index digit (byte) left to decode, when a 256-index subexpanse has sufficient population, it saves memory to represent the leaf as a bitmap with 1 bit for each index in the subexpanse, hence 256 total bits or 32 bytes. Here's an example of a bitmap leaf where each vertical rectangle represents 1 word. [On a 64-bit system the leaf looks similar except the words are bigger and there are half as many of them in the bitmap.]

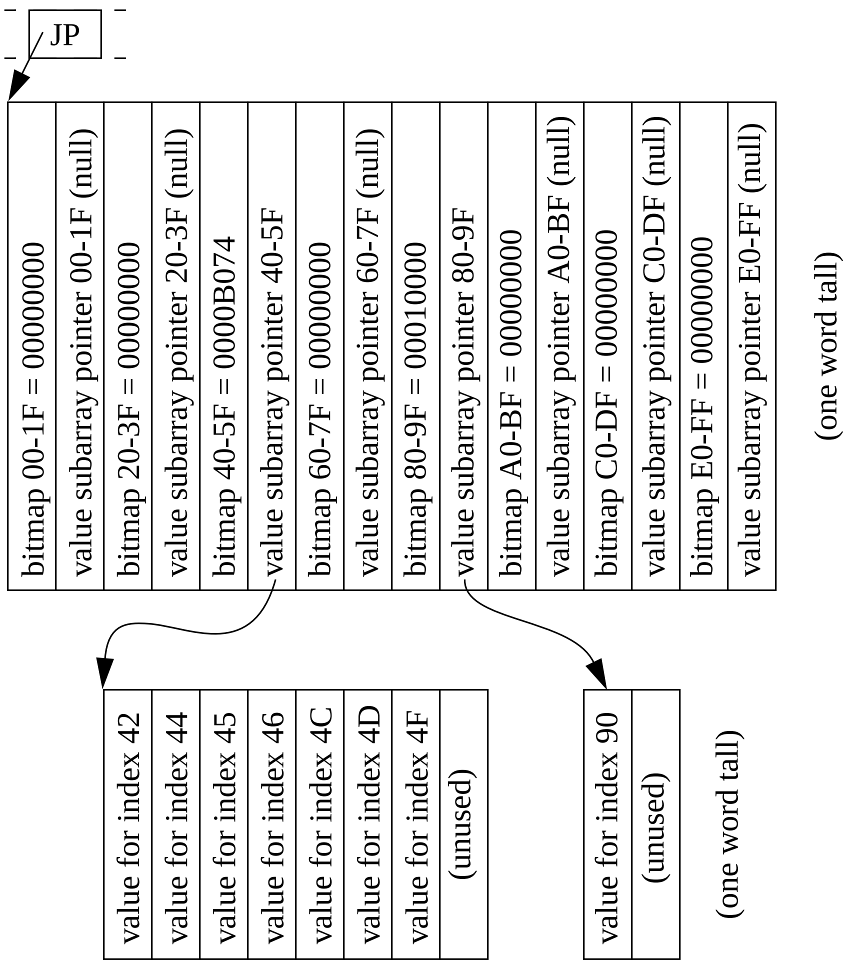
Figure 20a: Example of 32-bit Judy1 Bitmap Leaf



JudyL requires a value area (1 word) corresponding to each index. Like a bitmap branch, a JudyL bitmap leaf is a 2-tier object, except the JP subarrays (2 words per element) are instead value area subarrays (1 word per element). [On a 64-bit system the bitmap takes 4 words, there are 4 subarray pointers, and the remaining words are unused.]

020130: As described in ‘‘Bitmap Branches’’ on page 34, previously the bitmap words and the subarray pointers were not interspersed.

Figure 20b: Example of 32-bit JudyL Bitmap Leaf



Note: The preceding example is slightly misleading because for this small number (8) of valid indexes and associated values a linear leaf would be used instead. Imagine the bitmap leaf shown above but with more bits set and more corresponding value areas.

4.9 Symmetries

Note the degree of symmetry between branches and leaves, that is, between linear branches and linear leaves, and also between bitmap branches and bitmap leaves. (See also ‘‘Judy Population/ Expanse Organization and Growth’’ on page 23.) This symmetry is most apparent in the implementation (JudyL) wherein each index is mapped to an associated value. The interior nodes of the tree map portions (digits) of indexes to pointers to subsidiary nodes. The terminal nodes of the tree map fully decoded indexes to value areas that, in practice, often contain the addresses of, that is, pointers to, caller-defined objects external to the tree.

However, this symmetry fails in that there is no leaf equivalent to an uncompressed branch. When a higher-level leaf exceeds a specific population, it is converted to a lower-level leaf under a narrow(er) pointer or to a subtree under a new branch. When a lowest-level leaf exceeds a specific population, it is converted to a bitmap leaf.

You might ask, if it makes sense to convert a bitmap branch to an uncompressed branch to save one cache fill, why not a JudyL bitmap leaf to a large, uncompressed linear leaf? (Note that only JudyL has 2-tier bitmap leaves.) We think this would add complexity without a great benefit in performance compared with more-frequent tree traversals at higher levels. However, it’s an open question when you consider building meta-tries using JudyL as the branch nodes.

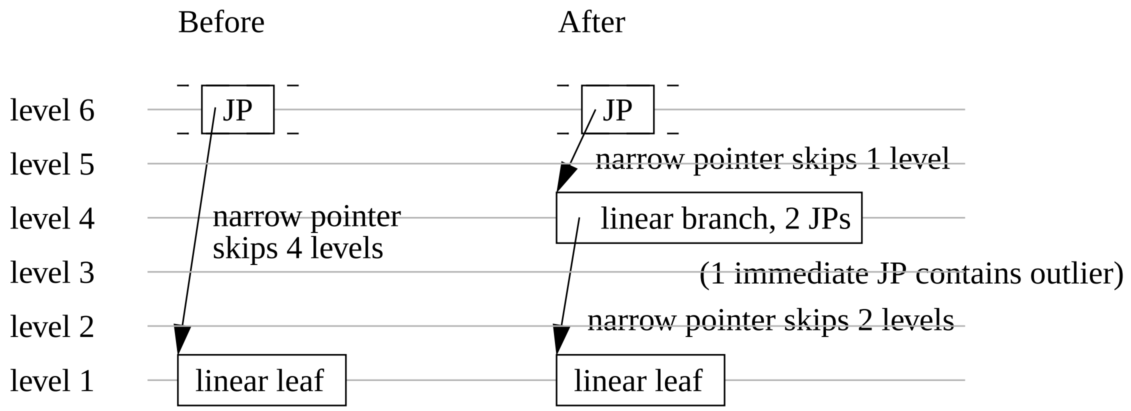
5. Usage of Data Structures

Here’s a summary of how the data structures previously described are employed. See also ‘‘Examples of Judy Trees’’ on page 67.

- An empty Judy array is represented by a null root pointer. See ‘‘Judy Array Pointer (JAP)’’ on page 28.
- Upon insertion of the first 1..31 indexes, the root pointer points to a root-level leaf, which requires 2, 4, 6, 8, 12, 16, 24, or 32 words = one memory chunk from the Judy memory manager, depending on population. This leaf begins with a *Population* word (except for population = 2, and for JudyL only, population = 1). See ‘‘Root-Level Leaves’’ on page 29.
- When a Judy array contains >= 32 indexes, the root pointer points to a JPM, which in turn points to a top-level branch. Note that a JPM’s *top JP* is never null, it never contains immediate indexes because it is only used when a root-level leaf overflows, and (by design) it never contains a narrow pointer or points directly to a leaf, either. See ‘‘Judy Population/Memory Node (JPM)’’ on page 31
- When possible indexes are stored in immediate JPs. When an immediate JP overflows it is converted to a leaf JP (see ‘‘Leaf JP’’ on page 41), and later perhaps to a bitmap JP (for 1-byte indexes only) or, for Judy1 only, to a full JP if the population grows large enough (see ‘‘Full Expanse JP’’ on page 44).
- The tree is kept in “least compressed form” in the leaves. For example, even if 32 3-byte indexes in a level-3 leaf have 1 or 2 leading bytes in common, no compression is done. The purpose of compression is to avoid multi-level cascades that would result in 2 or more (linear) branches each with a fanout of 1, that is, containing only a single non-null JP.
- When an index is inserted into a full leaf node, that is, one that is at capacity, the code checks if it is possible to do leaf compression and put the new index plus all present indexes in a lower-level leaf under a narrow pointer -- or a narrower pointer than the current one, if the full leaf is already under a narrow pointer. This is possible if the indexes are all in the same narrow subexpanse, that is, they have N digits in common under the parent branch, where N is greater than the parent branch’s level minus the full leaf’s level. See also ‘‘Linear Leaves’’ on page 44.
- As the Judy digital tree grows and a linear leaf with 2..4-byte [2..8-byte] indexes exceeds 2 cache lines (4 for JudyL), and if it cannot be converted to a lower-level leaf under a narrow(er) pointer, initially a branch replaces it. (A 1-byte-index leaf cannot overflow because it can be represented as a bitmap.) Typically, but not necessarily, the inserted branch is a linear branch. But with sufficiently random indexes in the leaf, a bitmap branch might be necessary. In either case the branch node could be opportunistically created as an uncompressed branch.
- When an index is inserted into any branch or leaf node under a narrow pointer, the code checks if it is an “outlier”, that is, it falls outside the expanse of the narrow pointer. If so, a new linear branch is inserted between the parent branch and the node under the narrow pointer. The new outlier index becomes an immediate index in the new linear branch, which has a fanout of 2 JPs. Note that a multi-level narrow pointer can have a new branch inserted “within it” and still result

in 1 or 2 narrow pointers above and/or below the new branch. (In practice, only on a 64-bit system; there aren't enough levels on a 32-bit system to support a 3-level or greater narrow pointer.) Here's an illustration.

Figure 21: Example of Inserting an Outlier



Note the "locality" of this operation. The old parent JP is copied into the new linear leaf unaltered and the linear leaf is unaltered. Most Judy insert/delete operations are similarly localized, hence faster.

- To minimize worst-case insert/delete time, **hysteresis** can be used. This means, for example, upon deleting an index, sometimes leaving a branch even when a leaf would use less memory, leaving a bitmap even when a 1-byte linear leaf would use less memory, etc.
- Finally, there's an interesting asymmetry between the insert and delete code. First, the insert code pre-grows the tree when necessary and then inserts the new index. Note that growing the tree is not always necessary. Sometimes the memory chunk already allocated has unused padding in it that allows a "grow-in-place" insertion. When this is not possible, the insert code rebuilds data structures as needed, thereby ensuring memory can be allocated, passing through a temporary state in which a branch or leaf has a lower-than-intended population, whereafter the new index is inserted now that there's room for it.

In contrast, the delete code uses hysteresis when possible to delay shrinking the tree until it is sure to fit in the next smaller sized object(s). A branch or leaf is only shrunk or deleted when the existing structure is already low-population enough to use a smaller or simpler structure, **before** the index is deleted. This makes for simpler delete code and in some cases prevents "thrashing" when a series of interleaved insertions and deletions occurs. Hysteresis is not possible for all data structures. In some cases the structure must exactly match the population because the population itself is the key to the choice of data structure, such as immediate indexes versus a leaf. Also, hysteresis greater than 1 is never used; at most, there is a one-index delay.

6. Machine Dependencies

Judy is designed to be as fast, small, and portable as possible on “modern processors” where the CPU cycle time is small compared to the memory (RAM) access time. However, its design has the following machine attribute dependencies.

- **Word size**: Judy is designed for modern processors with 32-bit or 64-bit words. For HP-UX PA-RISC and IPF we build 32-bit and 64-bit versions, including PA 2.0 32-bit and 64-bit versions. Naturally a larger word size means somewhat slower execution, but also an enormously larger expanse for each tree, consequently with much more frequent use of narrow pointers. The 32-bit version runs fine on a 64-bit PA-RISC system, but can only be linked with a 32-bit main program. We expect the main program’s choice of word width to be based on various external requirements, and then the appropriate version of Judy used to match.

- **Cache line size**: Judy is designed for modern processors with 16-word cache lines. (Size in **bytes** varies between 32-bit and 64-bit systems.) It’s difficult to determine the cache line size at run-time, and for efficiency the software must be compiled for a fixed size anyway. However, we expect Judy’s speed to degrade (or underperform the optimum) only a little on systems with 8-word, 32-word, or 64-word cache lines. In fact IA32 has an 8-word cache line size, and (I think) PA-RISC 64-bit systems also have 8-word cache lines. Dunno about IPF.

- **Endianness**: Judy is designed to operate equally well whether the architecture is little-endian or big-endian. Data is accessed through C structures in such a way (“endian-neutral”) as to hide the differences. However, leaf search routines for odd-byte-sized indexes might be faster if the machine’s endianness were known, that is, ifdef’d into the code.

- **Memory manager**: Judy contains its own memory manager that obtains large chunks of memory from the system and parcels it out in smaller units. To keep the memory manager simple and efficient, chunks are only available in the following word sizes: 2, 4, 6, 8, 12, 16, 24, 32, 48, 64, 96, 128, 192, 256, 384, and 512. These are each 2^N*3/4 or 2^N. The largest Judy object needed is a 256-way branch, 256 * 2 = 512 words. (Note: Even JudyL has no larger objects.)

  -- Caveat: Prior to 11i OEUR, the memory manager was modified to allocate smaller chunks and use straight malloc(3X) for larger chunks.

  While the memory manager is not directly a machine dependency, it intimately relates to the Judy design, such as in the handling of 2-index root-level leaves; the tradeoff point between linear and bitmap leaves; and a greater use of pointers to small objects rather than in some cases using larger, contiguous objects and fewer pointers. The last would require a more capable memory manager, better at coalescing odd-sized free blocks. Such a memory manager has been written, but as of September 2001 it has not yet been installed in Judy.

7. Working on Judy

Welcome to the Judy team! The **good news** is that Judy is an exciting state-of-the-art core technology with a lot of unexpected, synergistic applications, we’ve already built the basic product for you and gotten most of the bugs out, and we’ve written some documents, such as this one, for better or worse. The **bad news** is that like most software products Judy is large and complex, hence imperfect, and it comes with a fuzzy cloud of issues, tasks, and possible enhancements ranging from the urgent to the impractical. But I’m sure that you can fix all that for us old geezers...

“If youth but knew, if old age but could.” -- Henri Estienne

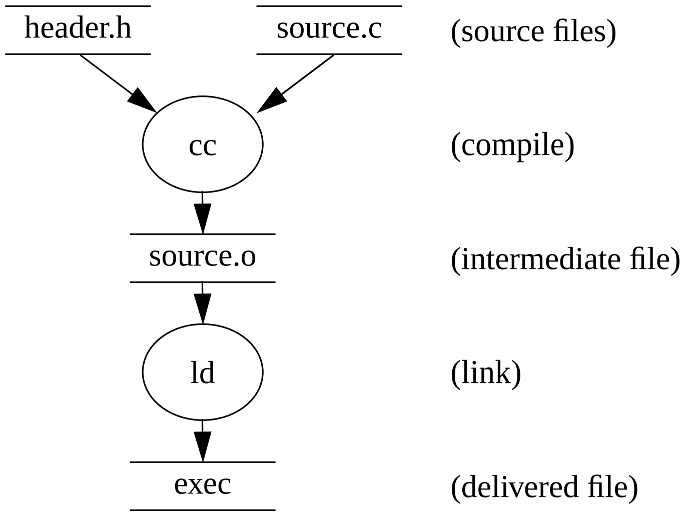
The purpose of this section is to give you an **overview** of the tools and processes used to implement, build, debug, and deliver Judy. However, this is not a complete description, to avoid redundancy with more specific documents about each topic. Consider this more of a tutorial than a reference, and forgive me if any of it is incomplete (even in concert with other material) or it gets out of date.

7.1 Source, Intermediate, and Delivered Files

Excuse the author for getting on the soapbox to preach a short bit of theory that you might find helpful in deciphering the Judy source tree and build process.

In the classical sense a source file is something you feed a compiler. Judy is written in ANSI C, so our source files are named *.h or *.c. In the more general sense, a build of a software product is a data flow, which can be represented as a data flow diagram (DFD). In a DFD there are repositories, usually files, and processes that transform data, such as a compiler. This document refers to as “**source files**” any files which appear in a build DFD with no input arrows, meaning they simply pre-exist in the build context.

Figure 22: Example Build DFD



Furthermore, any file with an arrow coming into it from one or more processes is a **constructed** file. There are two types of constructed files:

1. **Intermediate** files, which are not delivered to customers, and
2. **Delivered** files, which are shipped by some means.

In a “healthy” build-DFD, all source files have at least one arrow out of them; all intermediate files have both input and output arrows; and delivered files have at least input arrows, with output arrows possible but not required.

Note well that in a DFD a process step can be as simple as a “cp” (copy). That is, some source files, such as manual entries, are deliverable without further processing, but (and this is important), they are still copied to the delivery tree. (See below about source purity, and in ‘‘Makefile Concepts’’ on page 56 about the deliver/ tree.) For a bigger “real life” example of a DFD, see ‘‘Judy Build and Delivery Key Control and Data Flows’’ on page 79.

What does this have to do with Judy? Our private source files are maintained in a /judy/ tree on a SoftCM server (history manager) {prior to LGPL}. Other source files for our build process, such as system header files and libraries, are part of a build environment. We build Judy nightly for deliveries using the Common Process Framework (CPF), which includes:

- an **update.be** step that constructs a chroot-able augmented build environment (ABE);
- a **checkout** step that prepares all the private source files in a private “sandbox” that is a copy of the master source tree for a particular revision (tag or symbolic name, such as “J4”);
- a **build** step that invokes *make*(1) on the Judy makefile;

- a **package** step that constructs SD depots; and
- a **runtests** step that runs automated regression tests on the constructed deliverable files.

See also ‘‘Judy Build and Delivery Key Control and Data Flows’’ on page 79.

Delivery is semi-automated but lives outside the CPF (at least at this time). See ‘‘Delivering Judy’’ on page 66.

Furthermore, the Judy product is “**BE-pure**” in that it does not modify the build environment when it runs (a very bad practice). In particular this means that constructed files are placed under the source tree and not in their delivered locations on the host system or in the chrooted BE.

Also the Judy product is as “**source-pure**” as it can be, which means all constructed files are placed in constructed subsidiary directories and otherwise do not pollute the source tree itself. This allows multiple platforms, flavors, etc. to be built in the same source tree, albeit perhaps not simultaneously due to, for example, compiler temporary files. The exceptions to source purity have to do with tool files that must be modified in-place for packaging (see judy/tool/psf).

More is said in ‘‘Makefile Concepts’’ on page 56 about the source tree and constructed files layout.

7.2 History Manager (SoftCM)

The Judy private files source tree resides {resided} in /judy/ on a SoftCM server... {rest deleted as irrelevant}.

7.2.1 Convenient Shortcuts

Learn to use $PATH and $CDPATH in your shell; see the manual entry for details. The latter is very handy for popping back and forth between, say, judy/test/manual and judy/src via “cd manual” and “cd src”.

I also find it useful to declare the following (incomplete) set of shell environment parameters for working on the Judy code...

```
export hp='JudyCommon/JudyPrivate.h'
export hb='JudyCommon/JudyPrivateBranch.h'
export ht='JudyCommon/JudyPrivate1L.h'
export h1='Judy1/Judy1.h'
export hl='JudyL/JudyL.h'

export g='JudyCommon/JudyGet.c'
export in='JudyCommon/JudyIns.c'
export d='JudyCommon/JudyDel.c'
export f='JudyCommon/JudyFirst.c'
export pn='JudyCommon/JudyPrevNext.c'
export pne='JudyCommon/JudyPrevNextEmpty.c'
```

```
      export c='JudyCommon/JudyCount.c'
      export bc='JudyCommon/JudyByCount.c'
      export fa='JudyCommon/JudyFreeArray.c'
      export sl='JudySL/JudySL.c'

      export s='JudyCommon/JudySearchLeaf.c'
      export se='JudyCommon/JudySearchLeafEven.c'
      export so='JudyCommon/JudySearchLeafOdd.c'

      export cb='JudyCommon/JudyCreateBranch.c'
      export ib='JudyCommon/JudyInsertBranch.c'
      export cc='JudyCommon/JudyCascade.c'
      export dc='JudyCommon/JudyDecascade.c'

      export jp='JudyCommon/JudyPrintJP.c'
```

This is just to get you started and give you some ideas; your mileage will vary.

7.3 Makefile

Mostly you should read about the makefile (judy/src/makefile) in the makefile itself and in related README files, but here's a bit of overview. {Since then we added a rudimentary configure script, but Judy suffers from being platform-specific rather than attribute-specific, and is only set up now and ifdef'd to compile on six platforms, see below.}

7.3.1 Makefile Concepts

Per the persuasive argument found in a paper titled, "Recursive Make Considered Harmful", which you can locate on the Web, and also on past experience, the Judy makefile was designed to be relatively monolithic. That is, *make* does not call *make* again, recursively, any more than can be helped, so that the single invocation of *make* knows all dependencies globally. However, there are exceptions:

- Building a variety of similar libraries for different hardware versions, word-sizes, and library types (archive/shared) was simplest using a single level of recursive call where the parent *make* modifies some parameters for the child.
- The judy/src/makefile really belongs, now, one level up as judy/makefile, incorporating all knowledge about building other parts of the source tree that have their own makefiles or none at all. (However, some existing Makefile.deliver files are appropriate as written for execution in a delivered context.) I haven't had time to revise to this global makefile. {Half done now.}

  Furthermore, for ease of use, this global makefile should be accessible through wrapper scripts that know how to translate command line arguments and the present working directory to top-level *make* targets for subtree building, etc.

The existing judy/src/makefile does "reach up" to snag some files, such as ../doc/ext/Judy_3x.htm, to put them in the deliver tree.

The Judy makefile is also intended to be as portable as possible to a variety of build environments so that the same information need not be specified redundantly. {But alas, still platform-specific rather than attribute-specific.}

Aspects of Judy makes {using Makefile.multi rather than configure} are controlled through environment parameters. These can be set either by the shell (as in “FLAVOR=debug make”) or as *make* arguments (as in “make FLAVOR=debug”). The former is safer and more typical although neither form is as user-friendly as a command-line form might be (such as “makej debug” where *makej* is a shell script). The *make* command itself, of course, takes targets, not other values, as command line arguments, which can be confusing and/or limiting.

As described earlier, Judy builds are source-pure in that constructed files are placed in subdirectories segregated by platform and flavor. Also, deliverable files are placed in a pseudo-root subtree separate from intermediate files. The structure includes these components:

judy/src/*platform*/*flavor*/intermed/	# non-delivered files.
judy/src/*platform*/*flavor*/deliver/	# delivered files.
judy/src/hpux_pa/product/intermed	# an example.

Note that all platform names are unique with respect to other judy/src/ subdirectories. Also note that the deliver/ tree is complete unto itself (as much as possible) such that any product packaging step should be able to build its depot or equivalent from that tree without pulling in any files from other locations (necessarily with related path modifications). The package step might be **controlled** by external files, but it should not **deliver** any files other than those constructed in, including merely copied to, the deliver/ tree.

Note that not all “interesting things” you can build in the Judy source tree are known to judy/src/makefile. See in particular judy/test/manual/Jmake for building some test executables.

7.3.2 Using makelog

{Probably irrelevant for Linux.}

For most software products on most platforms the output from *make*(1) is long and verbose. Ideally you could ignore the output except for the bottom line, and assume if there’s no error there, the build was successful. However, this is a great way to overlook warnings, and even some errors that *make* itself does not detect. Moreover, *make* is often told to continue despite errors or warnings, say during an overnight build, so as to explore the build as thoroughly as possible. Finally, not every error or warning is clearly marked as such, especially those from subsidiary commands called by *make*, and some *make* output contains strings like “err” or “warn” due to filenames, etc.

To work toward reliable detection of build problems, I wrote a program called *makelog*. This is a wrapper around *make* that separates its output into summary and detail streams and does some automatic logging of the detail stream by default. The program is relatively simple-minded about what constitutes expected, non-error summary output, what is a detail, and what is an error or warning that should be flagged (marked with an inbound hyperlink) during HTML-enabled execution. For developer use, *makelog*'s summary output is a good way to scan for problems.

HP-UX *make* is well-behaved about indenting echoed command output, which supports *makelog* nicely. However, Gnu *make* (on Linux) is not so nice, so *makelog* would not work well in that context, and I haven't yet tried to even compile it there, which could also have problems. So assume *makelog* is only usable on HP-UX (sigh) unless you learn otherwise, and *make* do without it in other contexts.

Given a usable *makelog* program, which exists as judy/tool/makelog (HP-UX-only binary) to support nightly builds under the CPF, it's a simple matter of saying "makelog" where you would normally say "make". For the *makelog* sources, contact the MSL CPF team.

7.3.3 Useful Parameters and Targets

The Judy makefile is generally not recursive-descent. Often a recursive-descent makefile tree uses top-level common include files (makefile fragments) to minimize redundant rule definitions. Judy has none of that, although as a result the makefile is rather long with a lot of similar rules explicitly spelled out. It still uses some includes, though, but as a way of mapping from an environment parameter to platform-specific or flavor-specific variations. See judy/src/make/README for more on this.

Following is a summary of the useful environment parameters to pass to the Judy makefile.

Table 7: Judy Makefile Environment Parameters

Parameter	Description
PLATFORM	"hpux_pa" (default), "hpux_ipf", "linux_ia32", "linux_ipf", or "win_ia32" to select make/platform.*.mk; more values likely to come
FLAVOR	"product" (default), "debug", or "cov" to select make/flavor.*.mk; "product" bits are delivered, "debug" bits work with debuggers and include assertions, and "cov" bits are like "product" as much as possible but support C-Cover code coverage measurements (on platforms where C-Cover exists)
CCPRE	command to insert before $CC to preprocess files; mainly useful for invoking judy/tool/ccpre script ("CCPRE=../tool/ccpre" or just "CCPRE=ccpre" if ../tool is in $PATH); the *ccpre* script expands macros for more explicit debugging, such as "CCPRE=ccpre FLAVOR=debug makelog"; note that *ccpre* is **always** used with FLAVOR=cov; see ''Code Coverage'' on page 61
EXTCCOPTS	options to insert in all $CC command lines, such as "EXTCCOPTS=-DTRACEJP" to turn on building with TRACEJP defined

Note that it’s a feature of *make* to only rebuild targets that are out of date. So for example, if you need to debug a particular *.c file with *ccpre* invoked, “touch” the *.c file before doing the *make* with CCPRE=ccpre so the target is in fact rebuilt. In general be careful that you [re]build and link what you intend.

The Judy makefile has a number of useful targets, which you can reference as *make* command line arguments, such as “make all”. The complete, accurate list of available targets at any time resides in the makefile itself, but reading that (long) file to understand them is a chore. Here’s a summary of the most useful targets as of this writing, which are unlikely to change.

{Wrong, some have, see Makefile.}

Table 8: Judy Makefile Useful Targets

Target name	Description
all	the default target; make all deliverable files (only)
lib	make PA-RISC 1.1 32-bit archive lib only (libJudy.a), or equivalent on other platforms depending on $PLATFORM
libs	make all archive libraries
libs_all	make all libraries, including shared libraries, if any
libs_pic	for HP-UX on PA-RISC (and possibly IPF later), make libJudy-PIC.a libraries that are not normally constructed or delivered, for special delivery to people who build Judy into their own shared libs and don’t want to depend on a Judy shared lib at run-time
docs	make delivered documents, such as manual entries, including nroff versions auto-generated from the HTML versions using judy/tool/jhton.c
demos	make demo programs, which are not built by “all”
tarchive	build “all” and also a tarchive of the deliver/ tree
tarchive_contrib	build a tarchive of demo (contributed) programs, which must already be built using “demos”
tarchives	make both “tarchive” and “tarchive_contrib”
release	build “tarchive” and copy it to the internal website on judy.fc.hp.com; use cautiously! -- requires manual steps, updates website
release_contrib	build “tarchive_contrib” and copy it to the internal website on judy.fc.hp.com
releases	make both “release” and “release_contrib”
checkJh	compile Judy.h.check.c to test the Judy.h definitions

Table 8: Judy Makefile Useful Targets

Target name	Description
clean clobber	remove intermediate or all files from the constructed files tree, but these are not as fast or easy as just using “rm -rf”; they are here for completeness, but are not very useful since Judy is “source-pure”
list	emit to stdout a list of “interesting source files” for searching, for example, “grep foo $(make list)”
list_check	emit to stdout a comparison between “make list” and the files present in judy/src, to manually review to ensure “make list” is kept up to date
lint	run lint with waivers files on the Judy sources; however, the waivers are typically out of date and a lot of hand-review is required
lint1	run lint on the Judy1 sources only
lintL	run lint on the JudyL sources only
lintSL	run lint on the JudySL sources only

What if you need to build Judy a special way and you don’t know how? One trick is to build Judy normally using the makefile and capture its output, either directly or via *makelog,* to a temporary file, say “tempfile”. Then edit the output, usually one *cc* or *ld* command selected from it, to be what you want, and run the resulting file through the shell, such as “:w ! ksh” in *vi*, or “ksh tempfile” on the commandline. If you use this method to, say, rebuild a *.o file, then you can say “make lib” afterward to relink the library using your new *.o file. This is one nice use of *make*’s dependencies: It only rebuilds that which it thinks is out of date.

7.4 Running and Debugging Judy

Judy is a library. So to run it, you must build an executable program that links with the library. This seems obvious, but it’s the underlying cause of a lot of confusion and user errors.

7.4.1 Linking

Be sure you give the right options (-L, -l; or the name of the library itself) to your compile/link commands for the executable so you link with the intended version and flavor of the Judy library. See for example judy/test/manual/Jmake (until/unless we roll that into a global Judy makefile like it should be.) Ensure that the library itself was constructed as you intended, for example, using “FLAVOR=debug CCPRE=ccpre” and actually rebuilding the *.o’s you would like to debug with macros expanded, for example:

```
FLAVOR=debug CCPRE=ccpre makelog lib
```

As the inventor of Judy is fond of saying, “If you don’t know what you are measuring, you don’t learn anything from the results.” This is true for performance measurements and also for debugging.

7.4.2 Debugging

So how do you debug Judy? Well personally I like *xdb*, but it went away at 11.00 (or 11.11?), and even though I copied it to a newer OS and it works, it only works on 32-bit systems. For 64-bit I find myself using *dde* and learning to “love” its GUI. I won’t spend more time than that on debuggers. It’s up to you to locate and learn to use the debugger of your choice. Or even just resort to recompiling with printf() statements... Once you get up to speed, it’s pretty quick:

```
cd src # using $CDPATH in your .profile makes this simpler
# edit source file
# rerun previous makelog command such as “FLAVOR=debug CCPRE=ccpre
makelog lib”
cd - # back to, say, judy/test/manual
# rerun previous executable compilation command such as Jmake
# run your test again
```

Again, the tricky part about this is to be sure you rebuild exactly what you intend, while reusing previous commands. Be careful.

Be aware that there are a number of #ifdef’s in the Judy code that you can turn on to your advantage for debugging. See judy/doc/int/coding_policies for a start, and then search the code itself, especially for “DEBUG” and “TRACE”. In particular there are various TRACE capabilities, especially TRACEJP, which you can also search for to read about. To use them requires setting some environment parameters; read about this in source file header comments. You can also search for the following environment parameters in the source files (or just look for getenv()) to read about how to use some particular debugging features:

STARTADDR, KEYADDR, ENDADDR

STARTINDEX

STARTPOP, CHECKPOP

7.4.3 Code Coverage

We use the Bullseye C-Cover tool. Building with FLAVOR=cov produces a Judy library instrumented to measure run-time coverage of branches (decisions and conditions, weirdly referred to as “C/D” in the C-Cover commands), and a corresponding “covfile” (usually named test.cov). There is a lot to know about this process but it’s mostly documented elsewhere, so there’s no need to repeat it here. ....

When working with cov-flavor code you must be careful to have a corresponding covfile. There is one covfile corresponding to each libJudy.*; look in the cov/deliver/ tree to see them. Avoid swinstalling from the depot ..., and also getting a non-null covfile (with run data in it). Instead do something like this on the test system (currently judyj):

```
    cd /usr/lib/pa20_64
    what libJudy.a
    cp test_cov.a path/test_cov.a.auto
    cd /opt/Judy.cov/usr/lib/pa20_64
    what libJudy.a # must equal build ID from previous what.
    cp libJudy.a path/libJudy.a
```

Now in *path* you have a libJudy.a with a non-null matching covfile. If the two *what* outputs disagree only a little (same night), that's OK, use the latter libJudy.a because it's definitely the right one. The build ID in libJudy.a tells you which build you are working with.

Now you can "cp *path*/test_cov.a.auto *path*/test_cov.a.manual" and "covclear *path*/test_cov.a.manual", and use it, and then "covmerge *path*/test_cov.a.auto *path*/test_cov.a.manual".

The reason why *ccpre* is always used for FLAVOR=cov builds is that this is required for C-Cover to disambiguate multiple *.o files built from the same *.c file with different -D options. Without using *ccpre*, C-Cover only recognizes one set of branches, from one of the compilations. For example, if one compile pass generates Judy1Count() code and the other generates JudyLCount() code, the branches from JudyCount.c are not duplicated, resulting in a total C/D that's way too low. In many cases, both functions are not even represented in report output; and even if they are, it's incomplete, such as:

```
    Judy1Count ...src/JudyCommon/JudyCount.c 121 0 / 1 0 / 0
    JudyLCount ...src/JudyCommon/JudyCount.c 123 0 / 1 0 / 66 = 0%
```

(In this case I'm not even sure why Judy1Count() even showed up.)

Using *ccpre* means the compiler, hence C-Cover, sees a separate, preprocessed *.c file for each *.o file, as desired, but then to use *covbr* you must work with these "intermed" names, such as "JudyPrevNext_JudyLPrev_pre.c" (which is the source file derived from JudyCommon/JudyPrevNext.c using -DJUDYPREV and -DJUDYL).

7.4.4 Regression Tests

{Not yet public as of this writing? Or perhaps via CVS only?}

Once again there is a lot to say about this subject, but most of it is already documented elsewhere, so I'll be brief here. Start with judy/test/README and explore the tree below there. Be aware that the test structure might be arcane or unnecessarily complex. During the first Judy release, to 11.11 OEUR, we went through several gyrations trying to use variations of TET (Test Environment Toolkit, a testing environment) before giving up and working with a private subset that is portable but merely reminiscent of TET.

The general concept is that there is a directory for each test case which contains a file, normally a shell script called "prog", which is invoked to build, run, and clean up after each test. The judy/

test/test_registry.db file lists these scripts and indicates when they are to be run. In case of success the prog directory is cleaned out, otherwise various files are left around and copied to judy/test/BAD subdirectories. These files include “stdout” and “stderr”, which are the expected results, and “resout” and “reserr”, which are the actual results that differ (use diff to compare the files).

Moreover, the CPF nightly runs HTMLize their results..., browsing all the way down to detailed results for failed runs, although you can’t diff the output files this way. There’s lots of online documentation here too. I hope you can get up to speed by studying the files, including working examples.

For best results in the regression tests, for **successful** runs, do not write out things like dates, times, timings, directory paths, ... that might change from run to run. If the test **fails**, write everything you need (to stdout or stderr) to help debug the problem.

The tests are mostly run once for each flavor, but some are intended for FLAVOR=cov only to increase branch coverage. As a general rule you should keep the nightly processes running cleanly by fixing any problems that arise, and you should write new regression tests to keep coverage up to “100% or know why not” as new code or branches are added. For the original Judy release we were able to attain around 90% code coverage.

Some things to remember while working on regression tests:

- Currently we support regression tests on HP-UX PA-RISC only, and for CPF overnight invocation only -- developers must invoke each test by hand.
- Each test program must come with “stdout” and “stderr” reference files such that any deviation in the “resout” and “reserr” files (or non-zero return from the test program) is considered an error. Write your tests accordingly.
- Each test program (source form) is dropped into a directory with the stdout and stderr files and a “prog” script that you clone and customize from a model. (Have fun with this. After painstaking work on the Judy source code, here’s a place where you can go nuts, cut corners, be sloppy (if you must), etc. After all, it’s “just” internal tools.) This script builds the test program, hopefully in a standard way, and invokes it as many times as you like.
- Each test directory is registered in a database; see judy/test/test_registry.db. You (the author of the prog script and associated test program(s)) must decide if you want this prog invoked for all flavors, or if not, which flavors; and for all five (HP-UX PA-RISC) Judy libraries, or if not, for which libraries.
- Each invocation of a prog script (for one flavor + library) counts as **one** testcase to the CPF, even if there are many invocations of the embedded program. This is not ideal but it is simple. We really only care about N of N tests passing, and the coverage percentage; we don’t really care what is N.
- You can invoke any prog manually, outside the CPF, if you know how... Here’s an example:

```
JUDY_TEST=/users/ajs/judy/test FLAVOR=cov JUDYLIB=32a time prog
```

Look around at the prog scripts for how $JUDYLIB, etc. are used. Also study files under /judy/test/, starting with /judy/test/README.

- If you want to invoke a TET test via the “prog” script, and you want FLAVOR=cov, and you don’t want to share the test.cov file with other users on the system from a global location, you can use a private covfile. For example:

  ```
  $ cd /users/ajs/judy/test/auto/JudySL/multi  # test location.
  $ cp /usr/lib/test.cov_a test.cov           # private copy.
  $ covclear                                  # clear it.
  $ JUDY_TEST=/users/ajs/judy/test/ FLAVOR=cov JUDYLIB=32a prog ...
  ```

  Now this test runs automatically with the local covfile. Beware: If the last overnight test on this system, say judyj, was not cov, which it normally isn’t, the native Judy libraries on the system, such as /usr/lib/libJudy.a, are not FLAVOR=cov; you can copy them back from under /opt/Judy.cov/, but coordinate this with other users.

- Be careful when you use *covbr* that the source file you reference really does match the covfile you’re using. C-Cover is dumb about multiple *.o’s compiled from a single *.c, so we use *ccpre* for all FLAVOR=cov compilations, and we build cov flavor from preprocessed *.c files. (This also expands macros, which is nicer for *covbr* analysis too.) But now the *.c files are very sensitive to system header files. We build Judy in an 11.00 BE because there is (or at least was) no 11.11 BE. So the nightly intermed/*/*_pre.c files are different than those you might build in your sandbox on 11.11.

  So, how do you use the right source file for the covfile? It depends. If you build both the covfile and the sourcefile in your sandbox you are probably OK. But if you pull a nightly covfile, say for use in a TET prog directory (since you are linking against native libraries from last night’s build), and you want to use *covbr* too, you can’t just refer to a *_pre.c file in your sandbox.

7.4.4.1 Regression Tests -- Examples

For what it’s worth, here’s information written earlier by a Judy test creator:

When you are ready to create regression tests, please look at both of these examples.

The first sample shows a local regression test program, “hello.c”, which prog compiles, runs, and reports the status of. If there is an error, or mismatch of stderr or stdout, prog leaves the resout and reserr (and other work files and the SAVE directory) available.

```
$ ll test/auto/Judy1/template_c/
total 14
-r--r--r--   1 jer        users          267 Feb 22 10:39 hello.c
-r-xr-xr-x   1 jer        users         2172 Feb 22 10:39 prog
-r--r--r--   1 jer        users          104 Feb 22 10:39 stderr
-r--r--r--   1 jer        users          117 Feb 22 10:39 stdout
-rw-r--r--   1 jer        users           94 Feb 23 09:20 tet_captured
```

The second sample shows a benchmark test, where the “*.c” program is in the normal ~judy/src/apps/benchmark directory as “JLB_InsGet.c”. The *make* command runs in that directory and environment... and the program is run from there. There is no extra local copy of the executable.

```
$ ll test/auto/JudyL/JLB_InsGet/
total 12
-r-xr-xr-x   1 jer        users         2331 Feb 22 15:34 prog
-r--r--r--   1 jer        users          104 Feb 22 14:03 stderr
-r--r--r--   1 jer        users          225 Feb 22 15:31 stdout
-rw-r--r--   1 jer        users           94 Feb 23 09:20 tet_captured
```

Note that the output files, stderr and stdout, do contain some TET specific stuff, and a simple “Hello world.” or “SUCCESS”. I recommend a simple “PASS” or “FAIL” as the output. I also recommend following (name of engineer deleted to protect the innocent)’s lead, as he implemented a “-R” option for regression testing, so that the program only outputs “non-changing” stuff and the “PASS” or “FAIL”. Please be **sure** to give a return code of 0 with “PASS”. Lacking a 0 return code, TET assumes the test failed, no matter what it reports!

If there is a failure, the option “-R” can be removed and a manual test will show the problem.

Synopsis of the details for a successful regression test implementation:

- A local TET script, “prog”, does the setup, run and report analysis.
- The “prog” script is able to run in the TET environment or standalone.
- Be able to compile the test program in the Judy environment.
- Be able to use the /opt/Judy*/[lib|include]/* files in the standard makefile in the standard Judy tree.
- Implement a “-R” “regression = quiet” option for regression testing.
- Output only “non-changing” stuff and the “PASS” or “FAIL”.
- Give a return code of 0 with “PASS”, else the test fails.

(End of previously written text.)

7.4.4.2 Regression Tests -- Checkpointing

The Judy regression tests include an odd and little-used feature.

If you do **not** set $CHECKPOINT in the environment, the tests should run about as they did before the feature was added, with only trivially longer durations (< 1%).

If you **do** set $CHECKPOINT in the environment, the tests take a **lot** longer, perhaps 2-3x longer, but they report all cases where the $COVFILE is not updated by the test (a serious blunder marked as ERROR), or where the metrics in the $COVFILE did not change (marked as NOTE).

This is to allow us to find bugs in our regression tests where they fail to update the covfile as expected, and perhaps where they are worthless because they add no C/D coverage. However, the latter is debatable because so long as the tests run within the time allotted (whatever that is), we probably don’t care if they are redundant.

Note well: Currently the checkpointing feature writes its errors and notes to stdout, which makes tests appear to fail if $CHECKPOINT is enabled and there is anything to report.

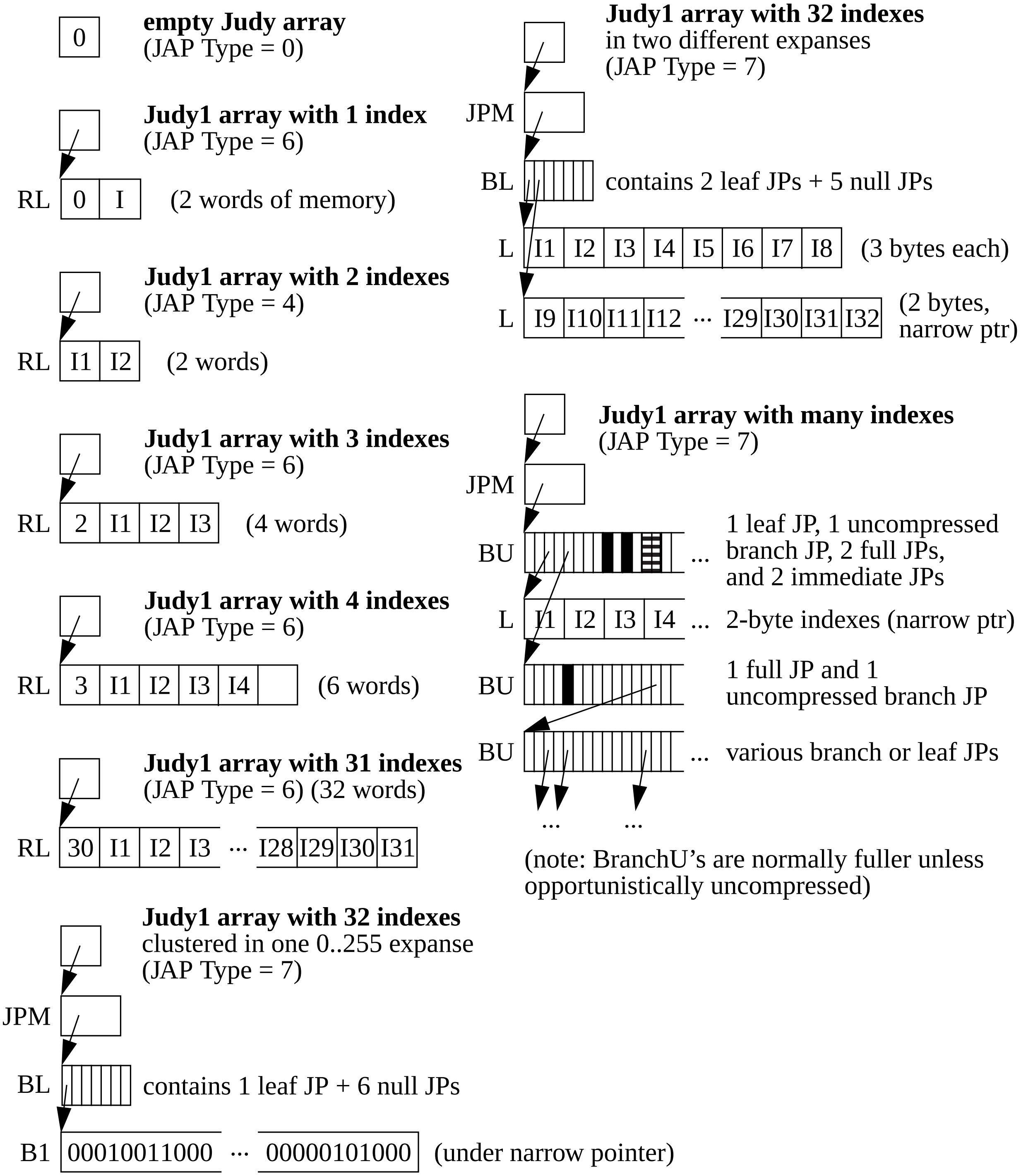
7.5 Delivering Judy

We wrote up the process separately... {and it’s irrelevant to non-HPUX}

Appendix A: Examples of Judy Trees

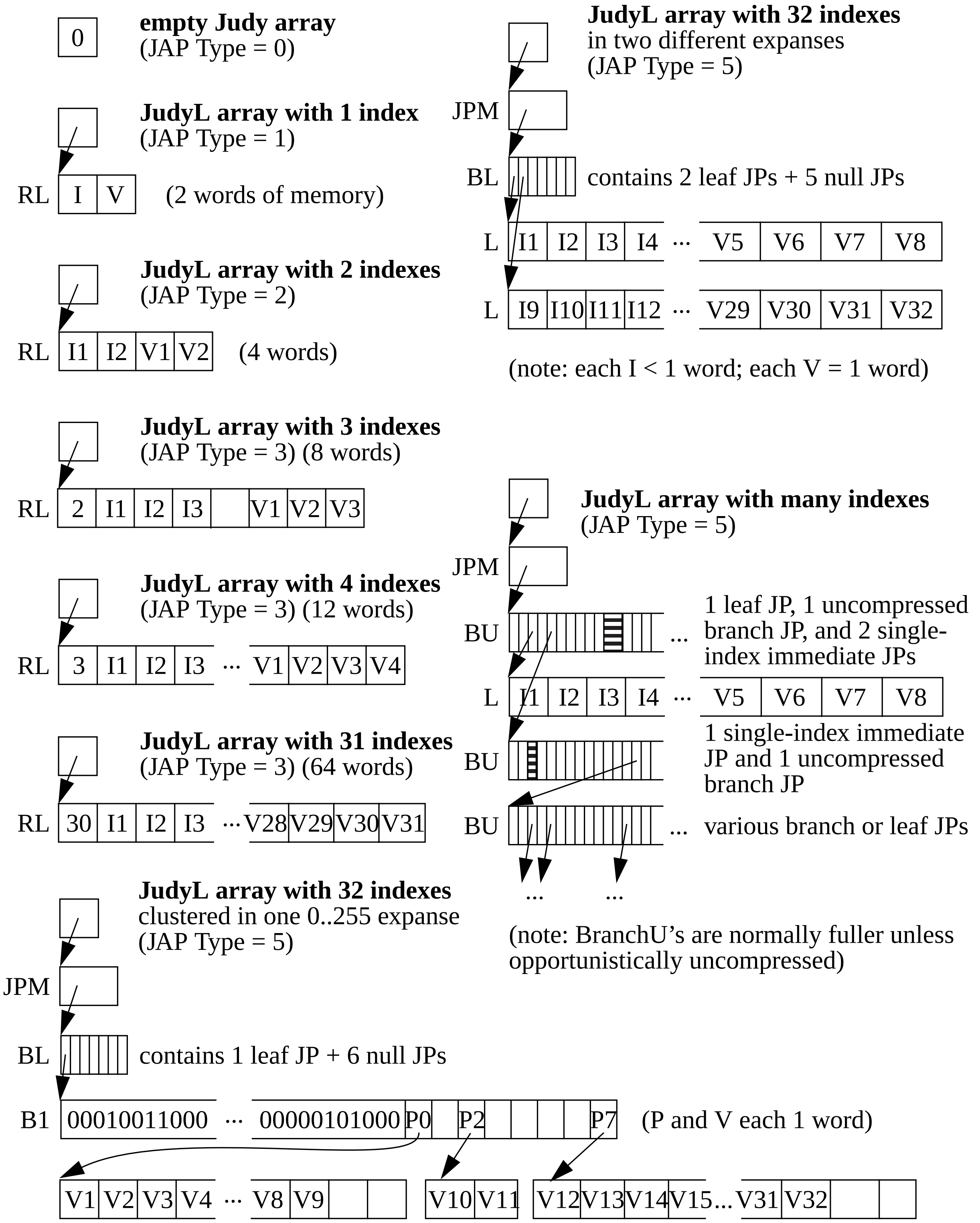
Empty Judy1 and JudyL arrays are identical; otherwise the trees vary because only JudyL associates a value with each index. These examples also illustrate conventional symbols for sketching Judy objects on paper or whiteboards.

Figure 23a: Examples of Judy1 Arrays (Trees)


0
empty Judy array
(JAP Type = 0)
Judy1 array with 1 index
(JAP Type = 6)
RL 0 I
(2 words of memory)
Judy1 array with 2 indexes
(JAP Type = 4)
RL I1 I2
(2 words)
Judy1 array with 3 indexes
(JAP Type = 6)
RL 2 I1 I2 I3
(4 words)
Judy1 array with 4 indexes
(JAP Type = 6)
RL 3 I1 I2 I3 I4
(6 words)
Judy1 array with 31 indexes
(JAP Type = 6) (32 words)
RL 30 I1 I2 I3 ··· I28 I29 I30 I31
Judy1 array with 32 indexes
clustered in one 0..255 expanse
(JAP Type = 7)
JPM
BL
contains 1 leaf JP + 6 null JPs
B1 00010011000 ··· 00000101000
(under narrow pointer)
Judy1 array with 32 indexes
in two different expanses
(JAP Type = 7)
JPM
BL
contains 2 leaf JPs + 5 null JPs
L I1 I2 I3 I4 I5 I6 I7 I8
(3 bytes each)
L I9 I10 I11 I12 ··· I29 I30 I31 I32
(2 bytes, narrow ptr)
Judy1 array with many indexes
(JAP Type = 7)
JPM
BU ...
1 leaf JP, 1 uncompressed branch JP, 2 full JPs, and 2 immediate JPs
L I1 I2 I3 I4 ...
2-byte indexes (narrow ptr)
BU
1 full JP and 1 uncompressed branch JP
BU ...
various branch or leaf JPs
... ...
(note: BranchU's are normally fuller unless opportunistically uncompressed)

JudyL adds value areas, which radically changes some of the structures.

Figure 23b: Examples of JudyL Arrays (Trees)


0
empty Judy array
(JAP Type = 0)
JudyL array with 1 index
(JAP Type = 1)
RL
I
V
(2 words of memory)
JudyL array with 2 indexes
(JAP Type = 2)
RL
I1
I2
V1
V2
(4 words)
JudyL array with 3 indexes
(JAP Type = 3) (8 words)
RL
2
I1
I2
I3
V1
V2
V3
JudyL array with 4 indexes
(JAP Type = 3) (12 words)
RL
3
I1
I2
I3
···
V1
V2
V3
V4
JudyL array with 31 indexes
(JAP Type = 3) (64 words)
RL
30
I1
I2
I3
···
V28
V29
V30
V31
JudyL array with 32 indexes
clustered in one 0..255 expanse
(JAP Type = 5)
JPM
BL
contains 1 leaf JP + 6 null JPs
B1
00010011000
···
00000101000
P0
P2
P7
(P and V each 1 word)
V1
V2
V3
V4
···
V8
V9
V10
V11
V12
V13
V14
V15
...
V31
V32
JudyL array with 32 indexes
in two different expanses
(JAP Type = 5)
JPM
BL
contains 2 leaf JPs + 5 null JPs
L
I1
I2
I3
I4
···
V5
V6
V7
V8
L
I9
I10
I11
I12
···
V29
V30
V31
V32
(note: each I < 1 word; each V = 1 word)
JudyL array with many indexes
(JAP Type = 5)
JPM
BU
...
1 leaf JP, 1 uncompressed branch JP, and 2 single-index immediate JPs
L
I1
I2
I3
I4
···
V5
V6
V7
V8
BU
1 single-index immediate JP and 1 uncompressed branch JP
BU
...
various branch or leaf JPs
...
...
(note: BranchU's are normally fuller unless opportunistically uncompressed)

Appendix B: Glossary/Lexicon

The follow list of terms was originally written in May 2000, based on Judy III and early Judy IV, without reference to outside academic sources (oh well), as an ordered series of concepts that progressed logically rather than alphabetically. In merging the old Lexicon with this document I decided to maintain a glossary as an appendix, but to simply order it alphabetically and try to bring it up to date. I also can't promise that our local definitions agree with consensus (academic) usage... Or even that every term or concept listed here is in fact used elsewhere in this document! But perhaps this glossary will be useful to readers anyway. It does "say the same thing differently" from the rest of this document, and even adds some new material.

See also the glossary in the external "Programming with Judy" book.

Abstract data type (ADT): Any data structure that can hold a variety of specific "key" data types, such as integers or characters. It's the relationship between the keys, not the type of the keys, that defines the ADT. We also use ADT to refer to cases in which a data structure has variable parts (nodes) to adapt to the data being stored.

Adjacent: Given all possible index sets, an adjacent of any index set is any other index set that differs from the first by the performing any one of a set of one or more "adjacency operations". In the Judy context, the useful and interesting adjacency operations are, "insert one index" and, "delete one index". In other words, the only way to get from one index set to another (except to the null set via Judy*FreeArray()) is by inserting or deleting a single index. By comparison, in attacking the Traveling Salesman Problem some useful adjacency operations might be, "rotate any subset of the route by any amount", or, "reverse the order of any subset of the route." Note: Adjacency is a kind of neighboring, but at a higher level of abstraction. It's a relationship between two index sets rather than between two indexes. To avoid confusion, a different term is used for the relationship between index sets.

ADT: See Abstract Data Type.

Amortization: Dividing some usage of time or memory space over a larger set of objects, such as indexes, such that the usage of interest is trivial in comparison. When a cascade occurs during index insertion, more memory is required, say to create a new branch node. If the number of indexes involved in (or rearranged by) the cascade is large enough, the relative amount of new memory needed to complete the cascade is small by comparison to the whole. To maximize locality and minimize the need for hysteresis, the data structure should maximize amortization for the average and/or worst case cascade. Amortization can also be applied to the fan-out (order) of a tree's nodes. The wider the nodes (branches), the less overhead there is in the tree relative to the "payload" (user or application data) in its leaves, unless the branches are many and sparse (containing many null pointers).

API: Application Programming Interface, a specification of the signatures of a collection of useful methods or functions and possibly global variables.

Bitmap branch [node]: A Judy digital tree node that uses 256 bits (32 bytes) to indicate which of the 256 subexpanses are populated (by indexes) and have corresponding JPs. The populated subexpanses are listed by-expanse in the bitmap, but the corresponding JPs are packed by-population (no null JPs, at least until/unless we add uncompressed bitmap branch subexpanses). Bitmap branches are two-tier objects to minimize insert/delete time; that is, the first level contains ordinary pointers to subarrays of 1..32 JPs for each of 8 subexpanses. See ‘‘Bitmap Branches’’ on page 34.

Bitmap leaf [node]: In a trie, once the expanse reduces to a small enough size, in practice in Judy, 256 indexes = 1 byte worth of undecoded indexes, if the population is not small enough to fit in a linear leaf node, it can be more efficient to represent the expanse’s (population’s) indexes (index set) as a bitmap (256 bits = 32 bytes) than as a list of index values (could be as many as 256 at one byte each). For Judy1 a bitmap leaf is just simply a bitmap; for JudyL there are also associated value areas in 1..8 [1..4] subarrays of 1..32 [1..64] value areas each. See ‘‘Bitmap Leaves’’ on page 47.

Branch [node]: A non-leaf node in a Judy tree other than a JPM node. This term is used as a noun, although this is a bit weird at first (but it is used this way in academic papers too), in order to give a more specific name to this type of node. Note that Judy branches contain more than just pointers (addresses); they contain JPs (Judy Pointers), which are a type of “rich pointer” that associates information with the address. See ‘‘Judy Branches’’ on page 33.

Branch width (fanout): The number of possible (virtual) or actual pointers in a branch, that is, the number of subtrees, which is related to, but not identical to, the branch’s memory size in words or bytes. A branch node divides an expanse into a collection of subexpanses, each with one associated pointer, some of which can be null (if the subexpanse is unpopulated). A linear or bitmap branch has an actual fanout (width) lower than its virtual width, whereas a Judy uncompressed branch has virtual fanout = actual fanout = 256.

By-expanse: Dividing an index set into subsets, as in a hierarchical data structure, by partitioning the indexes according to a prearranged series of subexpanses, also referred to in some literature as “a priori” storage. A digital tree is a by-expanse tree. In practical terms by-expanse storage means subdividing evenly, and with the same fanout at each level, in fact using whole numbers of bits (2^N fanout at each branch node). Judy divides index sets by-expanse with a very wide digit size of 8 bits (256 fanout). By-expanse trees are by nature unbalanced, but can also have predictable depth for fixed-size keys.

By-population: Dividing an index set into subsets, as in a hierarchical data structure, by partitioning the indexes according to which indexes are present. A binary storage tree is a by-population tree. The partitioning rules are prearranged as is true for by-expanse storage, but tend to be more about tree balancing than about locality of modifications, meaning the partitioning of any newly inserted indexes depends more on the index set already stored than about the value of the new index itself.

Cache fill: See CPU Cache Fill.

Cache line: See CPU Cache Line.

Cascade: During insertion of an index, this is when a leaf data structure overflows and more nodes must be added to a tree (or in some cases the leaf's indexes can be compressed into a lower-level leaf under a narrow pointer). A decascade (coalesce) is when the opposite occurs during an index deletion. Cascading results from changing an index set into one of its adjacents -- in the Judy context, by inserting or deleting one index.

Compressed branch: A linear or bitmap branch.

Compressed leaf: A bitmap leaf; also used to refer to index compression in lower-level leaves, wherein the already-decoded (at higher levels) leading digits are not present.

Counting tree: A tree where each branch node records the number of indexes stored under it, or equivalently, as in Judy1 and JudyL, where each pointer in the branch node has associated a count of the indexes stored under or in the branch or leaf pointed at by the pointer. These counts can be used to rapidly determine the number of valid indexes in any range or expanse, that is, between any pair of indexes. They are also useful internally when inserting, deleting, or searching for previous or next indexes.

Alternatively, each node or pointer could have associated a count which is a total of the application's values associated with each index in its index set. This can be useful in certain applications for determining index value density instead of index density. Note: Wider digital branches, even if otherwise efficient (meaning few null pointers), cause more work, and possibly more cache fills, while gathering counts, since every pointer in the wide branch must be visited. See ''Decode and Population Fields'' on page 39.

CPU cache fill (cache line fill): Needing to go to main memory (RAM), as the result of a cache miss, for one or more bytes not currently in any cache line on a CPU chip; results in filling a cache line; takes 30-150 times longer than a cache hit. Note that sometimes a cache fill results in a disk (buffer) cache miss and fill (swap in), by reading from mass storage, which is even more expensive. In this document, "cache" refers to CPU cache lines and does not even consider disk cache.

CPU cache line: A block of memory local to a CPU chip that caches the same-sized block from the computer's main memory. This is analogous to using a buffer cache to minimize disk I/O, but is implemented in hardware. Cache lines are size-aligned and typically 16 words in size in modern processors. Often there are hidden constraints on them, such as they can only map to certain offsets in memory.

Decascade: See Cascade.

Decoded index bits/bytes/digits: The most-significant bits/bytes/digits of an index that were already used, while traversing the tree, to do address calculations in higher-level branch nodes.

Density: A population divided by its expanse.

Digit: A key, or a field, or portion of a key, that is used to address through a branch node in a digital tree. For Judy each digit is one byte (eight bits), “very wide” as digital trees go.

Digital tree = trie: A tree data structure in which each node divides its expanse uniformly into N same-sized subexpanses, each with its own pointer to a next-lower node. In a pure digital tree the indexes need not be stored in the tree. They are used (piecemeal, digit-by-digit) to address into each node’s array of pointers. Hence N is typically a power of 2, so integer numbers of bits are “decoded” from the index while traversing down the tree. An ordinary array can be thought of as a one-level trie where the entire index is used in a single addressing calculation. Note that N need not be the same for every node in a trie, if there’s a way to determine the appropriate N for each node. For Judy, N = 256 (8 bits) at every level, but compressed branches have an actual fanout that is less than their virtual fanout. See ‘‘Digital Trees’’ on page 10.

Dynamic range: The population (size) or population-type (distribution of keys or indexes) over which an algorithm or data structure works well (untuned or with tuning). Judy has a wide dynamic range. See ‘‘Judy Dynamic Ranges’’ on page 24. Most ADTs work well only over specific, “tuned” or designed-in (limited) dynamic ranges.

Expanse: The range of possible indexes (keys) that can be stored under one pointer (root or lower) in a tree structure. For example, a root pointer to a 32-bit Judy array has an expanse of 0.. 2^32-1.

Fanout: See Branch Width.

Fixed-size index: A key to an array-like data structure which is always the same size, in practice one machine word in size. Judy1 and JudyL support fixed-size, one-word indexes only. Judy grew out of a need to solve this problem efficiently, although in the academic literature only variable-size index problems seem to be considered interesting. Ironically a meta-trie consisting of JudyL arrays as very-wide branch nodes can be used to support variable-size keys quite nicely. See also judy/doc/int/variable_size_keys.*.

Grow-in-place: Allocating memory in chunks that are often larger than the actual bytes required, by rounding up to a lesser number of different memory chunk sizes, means there is often some padding present in a given node. Thus the node can be modified in place during an insert operation, and it can also shrink in place during a deletion without having to be reallocated to a smaller size.

Horizontal compression: See Index Compression.

Hybrid data structures: Using different ADTs at different levels or for different nodes in a tree data structure. This means switching to a different ADT while traversing the tree as the expanse and/or population shrinks, to best represent the population’s indexes with the fewest cache fills and the least amount of memory.

Hysteresis: Leaving a system (in this case a Judy data structure) in a previous state even when it could be modified to a new state, with the result that the present state depends on the “direction” (such as insertion or deletion) from which it was approached. Any application that deletes indexes is likely to perform a series of insertions and deletions, often within a relatively small expanse of indexes. Hence it might be faster on average to not revise the data structure to the optimal form for each new index set (after each operation). Especially upon deletion followed by insertion, it can result in superior performance if decascades are postponed. The risk with hysteresis is that random or pathological sequences of insertions and deletions can result in a greatly suboptimal data structure -- and it's difficult to identify them or rule them out. Judy presently makes use of limited hysteresis (one index at most) in some cases of deletion.

Immediate indexes: Similar to how a machine instruction can contain an immediate (constant) rather than indirect (variable) value, if the population in a Judy expanse is low enough (perhaps just one index), depending on the data structure (memory space associated with each pointer in a branch), the population's indexes can be stored locally in a JP rather than in a separate leaf node under the JP, saving one indirection and possibly some memory. This requires encoding into the JP the special case of immediate indexes. See ''Immediate Indexes JP'' on page 41.

Index: A key to an array or to an array-like API.

Index compression = horizontal compression: At any level in a tree data structure, all indexes in the current population (index set within the current expanse) might have one or more most-significant bits (MSB) in common. These common bits can be stored just once in multi-index shortcut leaves, in a variety of possible "compression schemes", such as:

- Storing the first valid index in full and the others as relative offsets, either from the first, or from each previous.
- Storing the common bits only once, and for each valid index, only the bits that might differ. This is the same as relative offsets, all from the first value, which is a base value not necessarily equal to any one of the indexes.

In practice, due to machine instruction efficiencies, common MSBs are compressed only in units of whole bytes. For example, in a 32 bit system, there might be 0, 1, 2, or 3 common bytes, and thus 1, 2, 3, or 4 varying bytes, in the indexes in a leaf. At a low enough level in the tree, say where 3 bytes were already decoded, only 1 byte can vary. Furthermore, Judy does not bother with the kinds of complex compression schemes described above because we think the extra CPU time involved would outweigh the space savings. Judy does have the ability to compress a collection of indexes to a lower-level leaf, which has a greater capacity, under a narrow pointer, which skips 1 or more common leading bytes, if all of the indexes in the expanse are in the narrower leaf's expanse.

Index set: Given a data structure capable of holding 0..N indexes (keys, elements), any one [sub]set of those 0..N indexes is an index set. For a given expanse, each possible population's set of indexes is one index set. It must be possible to store and distinguish every unique index set in the data structure, regardless of the sequence of insert/delete operations used to arrive at that index set, or else the data structure is not very useful. Of course, Judy arrays pass this test.

Index: A key value to an ADT that appears array-like at the application interface. Since Judy appears array-like, keys into Judy ADTs are referred to as indexes.

Informational pointer: See JP.

Iteration: Reread this sentence until you are tired of reading it.

JP: Judy Pointer, a two-word object that populates branch nodes, also referred to... as a “rich pointer” or “informational pointer”. A JP contains an address field, except when the JP is null (represents an unpopulated subexpanse) or contains immediate indexes, plus other descriptive fields such as *Decode* bytes, *Population* count, and *Type*. See ‘‘Judy Pointer (JP)’’ on page 38.

JPM: Judy population/memory node, at most one per tree (Judy array), used when the array/tree population is too large to fit in a root-level leaf and hence is large enough to amortize the memory needed for the JPM. A root pointer points to a JPM which contains a top-level JP which in turn points to a top-level branch node. The JPM contains additional fields that make tree management faster and easier. See ‘‘Judy Population/Memory Node (JPM)’’ on page 31.

Judy array: An ADT that acts much like an ordinary array, but which appears unbounded (except by the expanse of the index) and is allocated by the first store/insert into the array. Judy arrays are hybrid digital trees consisting of a variety of branch and leaf node ADTs. Judy indexes can be inserted, retrieved, deleted, and searched in sorted order.

Judy pointer: See JP.

Judy population/memory node: See JPM.

Judy tree: The internal data structure used to implement the data stored in what is presented externally as a Judy array.

Judy1: Bit array that maps a long word index to Boolean (true/false).

JudyL: Word array that maps a long word index to a long word value.

JudySL: Word array with string index; map string index to long word value. Built as a meta-trie using JudyL arrays as extremely wide branch nodes. (Includes the use of JLAP_INVALID in JudyL value areas (subsidiary root pointers) to “escape” to shortcut leaves for unique string suffixes, a critical though subtle feature.)

Key: A data value (bitstring) used to look up related data values in a data structure that holds a collection of keys. See also Index.

Level compression = vertical compression: In the academic sense this means skipping levels in the tree when a branch node would otherwise have a fanout of 1. Read about Patricia tries for an example. Judy does this using narrow pointers. See ‘‘Decode and Population Fields’’ on page 39.

Linear branch [node]: A Judy non-leaf branch node for a low-fanout population. Linear branches are constrained to one cache line = 16 words, hence hold 1..7 JPs out of a virtual fanout of 256. Populated subexpanses are listed by-population (1 byte/digit each), along with a corresponding list of JPs. See ‘‘Linear Branches’’ on page 33.

Linear leaf [node]: A multi-index leaf (a terminal node) that holds a population too large to fit in an immediate index JP but small enough to fit in 2 cache lines (for JudyL, 4 cache lines including value areas). By convention a root-level leaf is referred to separately, and a linear leaf is always below the root level. Note that a linear leaf is never large enough to be fully populated, that is, to hold 256 indexes. See ‘‘Linear Leaves’’ on page 44.

Locality: Designing a data structure so a single insertion or deletion has the least wide-ranging effects on average and/or in the worst case. Ideally, every index set’s optimal data structure is very similar to every adjacent index set’s optimal data structure, implying excellent average and worst-case locality. If this cannot be achieved, at least instances of poor locality should be relatively rare. See ‘‘Usage of Data Structures’’ on page 50.

LSB: Least Significant Bytes/Bits.

Memory manager: Any dynamic multi-node data structure like Judy requires an underlying memory manager from which it can obtain chunks of memory, and to which it can possibly free (return) them. To minimize wasted memory and reduce the frequency and cost of pathological cases due to memory fragmenting into small and non-reusable chunks, the data structure should use the minimum number of different sizes of memory chunks, maximize the frequency of reuse of freed memory chunks, and/or the memory manager itself should be smart and efficient about merging adjacent free chunks into larger ones for reuse. (The *malloc*(3C) library is an example o a memory manager.)

Meta-trie: A hybrid digital tree (trie) whose branch nodes are in turn smaller tries. For example, a variable-size key trie might use fixed-size key tries like JudyL arrays as its branch nodes, especially given an “escape” mechanism like JLAP_INVALID to mark a JudyL value area as being other than a root pointer to a subsidiary JudyL array, such as a shortcut or multi-index leaf node. ...

MSB: Most Significant Bytes/Bits.

Multi-index leaf: If the population of an expanse in a trie is small enough, it can be more efficient to store the population’s indexes in a multi-index leaf object rather than under additional trie nodes leading to single-index leaves.

Narrow pointer: Suppose that high in a digital tree data structure there’s a large expanse, with a population large enough to not fit in a single leaf node, but whose members are clustered closely enough to have lots of index compressibility due to common leading bits/bytes/digits. A narrow pointer is a way of simultaneously skipping levels and unoccupied expanses in the tree. Associated with the narrow pointer must be the common index bits being “decoded” via the pointer, unless the whole indexes are stored in the leaves. Judy1 and JudyL support only fixed-size indexes (1 word each), so *Decode* bytes in the JPs in the branches describe the digits skipped by narrow pointers. See ‘‘Decode and Population Fields’’ on page 39.

Neighbor: In a sorted index set, the neighbor of any index (element) is the previous or next valid index in the index set. One of Judy’s strengths is its ability to rapidly locate neighbors.

Opaqueness: The property of hiding the internal details of a data structure or algorithm. Because Judy is very opaque, the application sees it purely as an array. Users don’t need to know the array is implemented as a tree, or actually as an even more complex hybrid ADT.

Optimal data structure: Given a set of unambiguous criteria, if they exist, by which to rank different data structures capable of holding the same index set, there should be exactly one best data structure to represent each possible index set. Ideally the implementation of each adjacency operation would then convert an optimal data structure (for one index set) into another optimal data structure (for its adjacent index set). It’s possible to have no unambiguous criteria for optimizing data structures. For example, Judy seeks to simultaneously:

- Minimize index access (get) time, both average and worst-case.
- Minimize index insert/delete time, both average and worst-case.
- Minimize memory consumed and thus maximize efficiency.
- Be as simple as possible -- but we had to give this up in the transition from Judy III to Judy IV. We learned that a wide dynamic range plus high-performance (although still portable) software requires a lot of different data structures (tree nodes) to pick from, plus maximally in-line code with minimal functions, loops, etc.

(See also ‘‘Judy Dynamic Ranges’’ on page 24.)

The optimal data structure for any index set depends sensitively on the tradeoffs chosen between conflicting criteria, and might not even be “computable”. (It worries me that whenever this ambiguity or complexity exists we might overlook better data structures or algorithms.) Ironically, space and time need not always be in conflict. Using data compression techniques to save memory, at the cost of some CPU time to compress/decompress, can reduce the overall size of a data structure, hence average CPU cache fills, resulting in less overall time to access an index.

Outlier: An index that falls within a given expanse but not within a narrow subexpanse of that expanse. For example, given indexes all beginning with 0x0102... in a leaf under a narrow pointer, an index beginning with 0x0103... would be an outlier under slot (digit) 0x01 of the top level branch. See ‘‘Usage of Data Structures’’ on page 50.

Parent [node]: A branch that contains a pointer to some other node (a child node).

Pathology: A pathological index set is one that brings out the worst-case behavior in a given algorithm and/or data structure. A pathological case is a sequence of insertions and deletions (or other adjacency operations) that result in a pathological index set and/or a suboptimal data structure, and/or which takes much longer to perform than the average.

Population: The number of indexes that are stored in one expanse, or the indexes themselves (that is, the index set), depending on the context.

Positional lookup: Data, such as pointers in a trie node, which are located via address calculations rather than by linear, binary, or any other type of search. Note that a positional lookup results in at most one cache fill -- so long as each data element itself does not cross cache line boundaries -- no matter how large the array or node in which the positional lookup is done.

Recursion: See Recursion. (Note: For performance reasons the Judy code uses very little of this. The “goto” statement is alive and well.)

Rich pointer: See JP.

Root pointer: A pointer to the top node of a tree structure. (Tree data structures are typically drawn upside down, with the root at the top and the leaves at the bottom.) See ‘‘Judy Array Pointer (JAP)’’ on page 28.

Root-level leaf: A simple linear list of whole indexes, possibly (in JudyL) with associated value areas, used for low-population Judy arrays (less than or equal to 31 indexes = 2 cache lines). The root pointer indicates the presence and type of the root-level leaf. A generic root-level leaf starts with a population count word. Small root-level leaves have their populations encoded in the root pointer to save memory. See ‘‘Root-Level Leaves’’ on page 29.

Shortcut leaf: By default, a full trie for a fixed-size index contains a fixed number of levels. If below a certain point there is only one index stored, that is, an expanse has a population of 1, it can be considerably more efficient to store that index in a non-trie leaf object directly below the last node containing 2 or more indexes in its expanse. A shortcut leaf must contain all of the remaining undecoded bits of the index it contains, since they are not encoded positionally in additional trie nodes. Shortcut leaves are mainly useful for variable-size key trees such as JudySL arrays, although multi-index leaves can be seen as a type of shortcut leaf too.

Sibling branch [node]: A branch node with the same parent as another branch; a peer.

Sorted index set: An index set whose members are uniquely sorted according to some rule, such as numerically or lexicographically.

Tree data structure: A hierarchical ADT in which the non-leaf nodes (in the graph sense) contain pointers to (that is, addresses of) other nodes, in an acyclical and non-multi-path fashion, possibly in addition to other data stored in the non-leaf nodes. A “storage tree”, such as a binary storage tree, is usually stored by-population and requires balancing, whereas a digital tree, such as a binary digital tree or a Judy tree, is stored by-expanse.

Trie: See Digital Tree.

Unbounded array: An array maps an index to a value. Normally an array is defined (at compile time) or allocated (at run time) with some fixed size, such as abc[10], which in C means indexes

0..9 are valid. An unbounded array is one where the size of the array is only limited by the size of the index. On a 32-bit system, for Judy1 and JudyL, where the index is a native machine word, this means 32 bits, or indexes 0..2^32-1. For JudySL, this means any string you can fit in memory. So unbounded doesn’t mean infinite, it means not arbitrarily bounded at less than the machine’s limits.

Uncompressed branch node: A simple array of 256 JPs, including null JPs for unpopulated subexpanses. See ‘‘Uncompressed Branches’’ on page 37.

Valid/invalid [index]: A valid index is one that appears in a index set; that is, it’s set or stored in a Judy array. An invalid index is one that could appear in the index set but does not; that is, it’s not currently stored in a Judy array. There are many synonyms for each word, such as: present/absent, set/unset, stored/free, full/empty.

Variable-size index: A key to an array-like data structure that handles a variety of index sizes, or possibly unbounded sizes. Bitstrings and character strings are examples of variable-size keys. JudySL is an array-like API implemented as a meta-trie that supports character string indexes.

Vertical compression: See Level Compression and Digital Tree.

Word: A unit of memory that is the same size as a pointer to pointer to void, and/or an unsigned long, in native C; typically 32 or 64 bits today. PA-RISC and IPF hardware support both 32-bit and 64-bit programs depending on compilation options. *“If any word is improper at the end of a sentence, a linked verb is.”* -- Safire

Appendix C: Judy Build and Delivery Key Control and Data Flows

Here on one (busy) page is an overview of all the Judy nightly build and occasional manual delivery steps... {HPUX only}

Appendix D: Some of the Inventor’s Thoughts About Judy

For what it’s worth, here’s something written in July 2000 by the inventor of Judy...

For me, Judy has been a fascinating exploration of numbers or rather collections of numbers.

When I, as a programmer, was faced with storing and retrieving a collection of numbers or data in a program, there were many published algorithms available for accomplishing this task. However, when it came to getting bug-free code, not much was available. In libc.a there are a few routines, but they often seemed awkward to use or slow or both.

In my years as a programmer I have seen countless mistakes in applying these algorithms. Hash tables that are static or too small, hash algorithms with long synonym chains, sorting algorithms with degenerate behavior, binary search algorithms that perform poorly, and remarkably, the most commonly found reason for poor performance problems -- the linear search.

The code for well established algorithms is generally not available out of the box and ready to go. This is because code modifications are frequently needed to make the algorithm fit the problem.

The Judy algorithm/code is an attempt to solve these shortfalls. Judy is:

- opaque -- no initialization, synonyms, dimensioning, or hashing
- scalable -- uses memory incrementally with population memory
- efficient -- uses compression for both speed and memory utilization
- naturally sorted -- allowing very fast neighbor searches
- counting[1] -- for very fast coalescing queues or stack depth problems
- ordinal determination[2] -- haven’t found an application for this capability yet

Judy uses the “infinite array” archetype as a simple way to think of and use it.

For example, a Judy1 array can be thought of as an array of bits, where the ordinal of the bit is the index. The ordinal or index can also be thought of as a key. A JudyL array can be thought of exactly like a Judy1 array with the addition of an associated ulong_t value with each bit. JudyV is another (hypothetical, not yet implemented) variation where the associated value is of variable size.

[1] Counting is implemented by determining the ordinal of the two passed indexes (keys) in a Judy array, then returning the difference of their ordinals + 1 (actually plus or minus the presence/absence of the two passed indexes themselves).

[2] It would be possible to pass an ordinal (a partial count of number of indexes stored in a Judy array) and return the index (and/or value) mapping to that ordinal. (In fact we implemented this later as Judy*ByCount().) The trivial example I can think of is finding the median index by passing the half count (population) of the array, then I would know the median index (assuming an odd number of indexes in the array).

However, in a Judy1 array, the ordinals can also be thought of as a collection of numbers that are stored in sorted order, or even a compressed collection in sorted order. It is interesting that sorting the numbers leads naturally to compression. For example, suppose one only stored the delta of the numbers in sorted order. Then the delta would take fewer bits than the actual number. How many fewer bits is a function of how close the numbers are. Note that a very densely populated Judy1 array takes very little memory.