Programming with Judy

C Language
Judy Version 4.0

For more information about the Judy technology, visit
the Judy web site at: http://devresource.hp.com/judy/

Manufacturing Part Number: B6841-90001

June 2001 Edition 1
Printed in USA

Legal Notices

The information in this document is subject to change without notice.

Hewlett-Packard makes no warranty of any kind with regard to this manual, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Hewlett-Packard shall not be held liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Warranty. A copy of the specific warranty terms applicable to your Hewlett-Packard product and replacement parts can be obtained from your local Sales and Service Office.

Restricted Rights Legend. Use, duplication or disclosure by the U.S. Government is subject to restrictions as set forth in sub-paragraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013. Rights for non-DOD U.S. Government Departments and Agencies are as set forth in FAR 52.227-19(c)(1,2).

Hewlett-Packard Company
3000 Hanover Street
Palo Alto, CA 94304 U.S.A.

Patent Notice. Hewlett-Packard has patents pending on the Judy Technology.

Trademark Notices. "Judy" is a trademark of Hewlett-Packard in the United States. UNIX is a registered trademark in the United States and other countries, licensed exclusively through The Open Group.

Revision History: June 2001, Edition 1.

This manual's printing date and part number indicate its current edition. The printing date changes when a new edition is printed (minor corrections and updates which are incorporated at reprint do not cause the date to change). The part number changes when extensive technical changes are incorporated. For the latest version, see the Judy documentation on the HP's Web:

`docs.hp.com/hpux/dev/`

Please direct comments regarding this guide to:

Hewlett-Packard Company
HP-UX Learning Products, MS 11
3404 East Harmony Road
Fort Collins, Colorado 80528-9599

Or, use our Web form:

`docs.hp.com/assistance/feedback.html`

Judy Technology Software License

ATTENTION: USE OF THE SOFTWARE IS SUBJECT TO THE HP SOFTWARE LICENSE AND WARRANTY TERMS SET FORTH BELOW. USING THE SOFTWARE INDICATES YOUR ACCEPTANCE OF THESE LICENSE AND WARRANTY TERMS. IF YOU DO NOT ACCEPT THESE TERMS, YOU MAY RETURN THE SOFTWARE FOR A FULL REFUND. IF THE SOFTWARE IS BUNDLED WITH ANOTHER PRODUCT, YOU MAY RETURN THE ENTIRE UNUSED PRODUCT FOR A FULL REFUND.

The following Terms govern your use of the accompanying Software unless you have a separate written agreement with HP.

License Grant. HP grants you a license to Use one copy of the Software. "Use" means storing, loading, installing, executing or displaying the Software. You may not modify the Software or disable any licensing or control features of the Software. If the Software is licensed for "concurrent use", you may not allow more than the maximum number of authorized users to Use the Software concurrently.

Ownership. The Software is owned and copyrighted by HP or its third party suppliers. Your license confers no title or ownership and is not a sale of any rights in the Software, its documentation or the media on which they are recorded or printed. Third party suppliers may protect their rights in the Software in the event of any infringement.

Copies and Adaptations. You may only make copies or adaptations of the Software for archival purposes or when copying or adaptation is an essential step in the authorized Use of the Software on a backup product, provided that copies and adaptations are used in no other manner and provided further that Use on the backup product is discontinued when the original or replacement product becomes operable. You must reproduce all copyright notices in the original Software on all copies or adaptations. You may not copy the Software onto any public or distributed network.

No Disassembly or Decryption. You may not disassemble or decompile the Software without HP's prior written consent. Where you have other rights under statute, you will provide HP with reasonably detailed information regarding any intended disassembly or decompilation. You may not decrypt the Software unless necessary for the legitimate use of the Software.

Transfer. Your license will automatically terminate upon any transfer of the Software. Upon transfer, you must deliver the Software, including any copies and related documentation, to the transferee. The transferee must accept these Terms as a condition to the transfer.

Termination. HP may terminate your license upon notice for failure to comply with any of these Terms. Upon termination, you must immediately destroy the Software, together with all copies, adaptations and merged portions in any form.

Export Requirements. You may not export or re-export the Software or any copy or adaptation in violation of any applicable laws or regulations.

U.S. Government Restricted Rights. The Software and any accompanying documentation have been developed entirely at private expense. They are delivered and licensed as "commercial computer software" as defined in DFARS 252.227-7013 (Oct 1988), DFARS 252.211-7015 (May 1991) or DFARS 252.227-7014 (Jun 1995), as a "commercial

item" as defined in FAR 2.101(a), or as "Restricted computer software" as defined in FAR 52.227-19 (Jun 1987)(or any equivalent agency regulation or contract clause), whichever is applicable. You have only those rights provided for such Software and any accompanying documentation by the applicable FAR or DFARS clause or the HP standard software agreement for the product involved.

DISCLAIMER. TO THE EXTENT ALLOWED BY LOCAL LAW, THIS HP SOFTWARE PRODUCT ("SOFTWARE") IS PROVIDED TO YOU "AS IS" WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, WHETHER ORAL OR WRITTEN, EXPRESSED OR IMPLIED. HP SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OR CONDITIONS OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT AND FITNESS FOR A PARTICULAR PURPOSE. Some countries, states and provinces do not allow exclusions of implied warranties or conditions, so the above exclusion may not apply to you. You may have other rights that vary from country to country, state to state, or province to province.

LIMITATION OF LIABILITY. EXCEPT TO THE EXTENT PROHIBITED BY LOCAL LAW, IN NO EVENT WILL HP OR ITS SUBSIDIARIES, AFFILIATES OR SUPPLIERS BE LIABLE FOR DIRECT, SPECIAL, INCIDENTAL, CONSEQUENTIAL OR OTHER DAMAGES (INCLUDING LOST PROFIT, LOST DATA, OR DOWNTIME COSTS), ARISING OUT OF THE USE, INABILITY TO USE, OR THE RESULTS OF USE OF THE SOFTWARE, WHETHER BASED IN WARRANTY, CONTRACT, TORT OR OTHER LEGAL THEORY, AND WHETHER OR NOT ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Your use of the Software is entirely at your own risk. Should the Software prove defective, you assume the entire cost of all service, repair or correction. Some countries, states and provinces do not allow the exclusion or limitation of liability for incidental or consequential damages, so the above limitation may not apply to you.

NOTE. EXCEPT TO THE EXTENT ALLOWED BY LOCAL LAW, THESE WARRANTY TERMS DO NOT EXCLUDE, RESTRICT OR MODIFY, AND ARE IN ADDITION TO, THE MANDATORY STATUTORY RIGHTS APPLICABLE TO THE LICENSE OF THE SOFTWARE TO YOU.

Contents

1. About Judy

When To Use Judy 11

2. The Judy Advantage

Judy Data Structures. 15
Compared with Other Search and Retrieval Methods. 18
Hashing versus JudyL 20

3. Using Judy

Using Judy1 28
Using JudyL 33
Using JudySL 44

4. Example of a Multi-dimensional Array

Concepts 49
Example Code. 50

5. Tuning HP-UX Memory Access

The chatr Command 55
Kernel Tunable Parameters 56
Compiler Switch 57
Swap Partition 57

A. Where Did Judy Come From?

B. Caching and Memory Management

Glossary

Contents

Preface

This book contains information about the Judy technology, a programming innovation from Hewlett-Packard that combines scalability and ease-of-use with a significant performance advantage in many data-intensive applications. The information in this book is designed for experienced programmers who have knowledge of at least one major programming language (preferably C, C++, or Java) and want to explore Judy's ability to replace existing data structures, such as arrays, sparse arrays, hash tables, B-trees, binary trees, linear lists, skip lists, sort and search algorithms, and counting functions with a much more scalable solution.

This book does not attempt to document all situations in which Judy can be used effectively or to analyze all of Judy's performance nuances compared with more traditional data structures. The Judy technology is new and many of its potential uses are open-ended and undiscovered. Therefore, this book can serve an introduction to Judy and an invitation to the programming community to adapt Judy's unique technology to new applications. This book is divided into these sections:

Chapter One: About Judy introduces the Judy technology, establishes a working definition of Judy, and describes the benefits of the technology.

Chapter Two: The Judy Advantage provides an overview of Judy's hybrid digital-tree structure and contains an analysis of Judy's performance when measured against other algorithms.

Chapter Three: Using Judy contains more detailed information about the types of Judy arrays and the functions they provide.

Chapter Four: Example of a Multi-dimensional Array shows a programming example demonstrating Judy functions.

Chapter Five: Tuning HP-UX Memory Access contains information about tuning HP-UX systems for better performance with Judy.

Appendix A: Where Did Judy Come From? contains background information about how the Judy technology was developed.

Appendix B: Caching and Memory Management provides more detailed information about how Judy achieves performance advantages through efficient cache memory management.

1 About Judy

The Judy technology brings the benefits of exceptionally efficient, dynamic arrays to the C programmer.

- Judy is designed to work well over a wide, dynamic range of values and maintains excellent performance across *all* array populations. Data-intensive applications using Judy can scale up without tuning as program requirements grow.
- Code creation and maintenance is much easier. Program data structures do not need to be periodically reworked for increased data-handling capability. Judy is a design-once solution proven over time.
- Judy provides an internal count that is exceptionally fast between any two array elements. In addition, data elements are stored in numerically sorted order.
- Productivity increases. Judy is a plug-in solution with an API that can be accessed through simple calls.

What is Judy?

The Judy technology is a C language library that provides an unbounded array capability. For example, suppose you could declare most arrays as:

```
array[max];
```

(Where *max* is 2^{32} on a 32-bit machine or 2^{64} on a 64-bit machine.)

Now imagine that the machine uses *only* RAM when data is stored into an array. Before you continue you need more information. How much RAM is required and how fast is storage and retrieval accomplished?

Normally you would expect a ratio of at least three words per element stored: one word for the index (key), one word for the value, and one or more words for the overhead to manage the structure. A linked list takes three words. A well written binary tree or skip list requires more memory.

By contrast, the Judy technology uses much less memory per element than traditional data structures (depending on the density and population of the indexes). It is remarkable that Judy often uses less than two words per element with very high populations. This is possible

because Judy stores the indexes in sorted order allowing compression of the common digits in the index word.

With binary-tree structures, you expect the speed to be close to a $log_2\ N$ function of the number of elements stored. Judy's speed is closer to $log_{256}\ N$. Unlike many data structures, Judy only slows a little when the population of indexes increases by a factor of 200. With this gentle slope, Judy performs well into the tera-element populations if that much memory is available. And excellent memory efficiency makes multi-dimensional Judy arrays possible (*probably best-in-class*).

Scalability is Judy's greatest strength. However, the Judy technology has many more benefits, including fast search and retrieval, and built-in sorting and counting functions.

Design goals

Judy is designed to replace many common data structures, such as arrays, sparse arrays, hash tables, B-trees, binary trees, linear lists, skiplists, and counting functions. Judy functions combine sparse arrays with innovative memory management and retrieval.

Judy is implemented as a tree and dynamically optimizes each node based on the population under that node. Judy also optimizes memory access to be more efficient than other algorithms that ignore memory caching. As a result, Judy exhibits better than $O(log_{256}\ N)$ retrieval behavior.[1] Judy has been used successfully for arrays with as little as one element (actually zero) to arrays with billions of elements. Since the technology is self-adapting, its design can support future machines with much larger memories than currently available.

Like many other algorithms, such as B-trees but notably excluding hashing, Judy keeps indexes in sorted order.[2] Judy also includes a powerful counting function that returns the number of data elements between any two keys or counts to any ordinal (position) within an array.[3] For example, Judy can store the first million prime numbers.

---

1. *Big O* notation is used to compare two algorithms that accomplish the same task. It is a theoretical measure of the execution of an algorithm, usually in terms of the time or memory needed. For more information, visit this web site: `www.wiu.edu/users/mflll/cs350/onot.htm`.
2. Because Judy necessarily defines the sorted order, the order cannot be modified to include user-defined sorting criteria (see page 26).

Then it can efficiently determine how many prime numbers exist between any random pair of numbers (prime or not). Because Judy maintains counts internally, the count function call can return an answer very quickly.

When To Use Judy

Designing programs with an unbounded array paradigm is one of the most powerful uses of Judy. Use Judy when your program requires:

- Sparse arrays with dynamic or unbounded indexes (0, 1, 2... *n*)
- The ability to scale effortlessly for future growth
- Fast search and retrieval
- Fast counting and sorting
- Good performance across various types of index sets: sequential, periodic, clustered, and random
- Memory efficiency: a low byte-per-index ratio for any array population
- Nearly linear performance from very small to very large populations

Array types

Judy offers three types of arrays:

Judy1 functions provide a way to store, retrieve, and locate Boolean values (bit maps) in a Judy array. See “Using Judy1” on page 28 for more information.

JudyL functions provide a way to store, retrieve, and locate long-word values in a Judy array. See “Using JudyL” on page 33 for more information.

JudySL functions provide a way to store, retrieve, and locate strings as indexes (similar to associative arrays in awk, Perl and Java). See“Using JudySL” on page 44 for more information.

3. Counting functions are available in Judy1 arrays (Judy1Count and Judy1ByCount) and in JudyL arrays (JudyLCount and JudyLByCount).

Judy libraries

The table below shows the location of the libraries that are provided with the Judy technology on the HP-UX system:

Hardware Architecture	Type	Location on system (from root)	
		32-bit	64-bit
HP-PA 1.1 (32-bit only)	archive	/usr/lib/libJudy.a	N/A
	shared	/usr/lib/libJudy.sl	N/A
HP-PA 2.0	archive	none	/usr/lib/pa20_64/ libJudy.a
	shared	/usr/lib/pa20_32/libJudy.sl	/usr/lib/pa20_64/libJudy.sl

NOTE

The 32-bit HP-PA 1.1 shared library (/usr/lib/libJudy.sl) is provided for compatibility only. For best performance on 32-bit machines, use the HP-PA 2.0 shared library (/usr/lib/pa20_32/libJudy.sl).

Development history

The Judy code has been tested and improved over several internally available iterations at HP. The most recent version runs about twice as fast as the previous version. This version of Judy was designed to maintain very good memory allocation of not more than 12 bytes per index for 32-bit machines and 24-bytes per index for 64-bit machines. In fact, for large populations of clustered data, memory efficiency can be less than 8 bytes per index on a 32-bit machine or 16 bytes per index on a 64-bit machine.

The Judy concept has been generating engineering interest at HP for years with early research and prototyping dating back to 1981. The current Judy technology has been under active engineering development for four years. Judy has been proven in several internal HP tools and products, including a performance profiling tool, a disk work-load analyzer (WLA), and the OpenGL (Graphics Library). (See Appendix A, “Where Did Judy Come From?,” on page 59.)

Judy is a technology with patents pending that was invented and implemented by Hewlett-Packard.

In summary

Judy can replace many commonly used data structures: arrays, sparse arrays, hash tables, B-trees, binary trees, linear lists, skiplists, and counting functions. Judy is designed to:

- ❏ Scale dynamically for future program requirements.
- ❏ Handle large, unbounded array structures particularly well with performance that is almost linear across large populations.
- ❏ Provide fast, built-in sorting and counting.
- ❏ Increase productivity through an easy-to-use API.

2 The Judy Advantage

The Judy advantage is a measurable performance benefit that can be demonstrated by benchmarking Judy against hash tables, skip lists, tree structures, stacks, and queues. The chapter describes how Judy uses the digital tree data structure in a new and unique way to provide enhanced performance. This chapter also contains real-time performance measurements that show Judy compared with a hash algorithm.

Judy Data Structures

Digital tree (trie) data structures

A digital tree or trie (pronounced *try*) is a multi-way or *M*-way tree whose nodes are vectors of *M* elements. Each element in a node is either a pointer to another node or to a leaf value. The tree is populated (and searched) using each *digit* of a key as an index into the nodes as they are traversed. A digit is determined by *M*. If *M* is 16, each digit is 4 bits. If *M* is 256 (as in Judy), each digit is 8 bits.

For example, assume we are storing a set of zip codes in 10-way digital tree. The zip codes are 87303, 92515, 92523, and 96503. Each level of the tree decodes one base-10 digit until a unique suffix remains. Zip code 87303 is stored at the top level of the tree hierarchy because the leading digit 8 is unique. Its position in the root node is node[8], as shown in Figure 2-1.

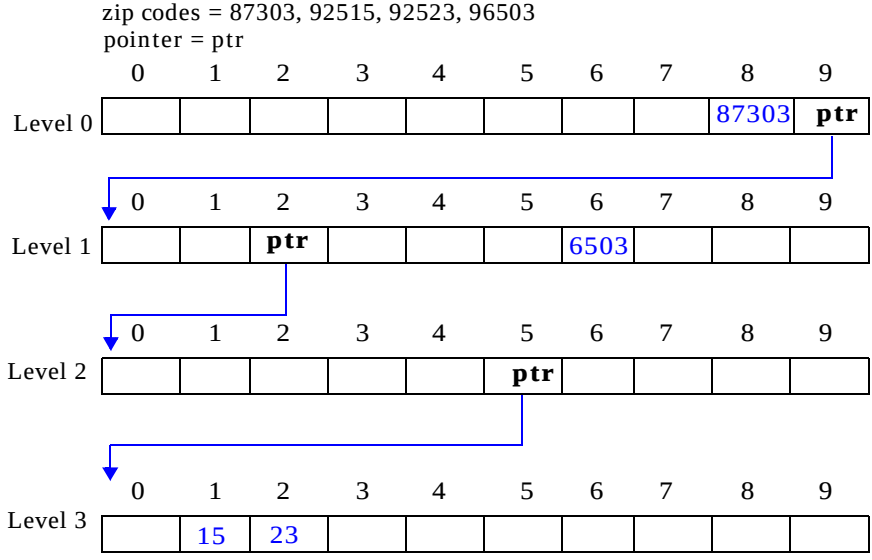
Since more than one zip code starts with 9, root node[9] points to a second level node. The second level node[6] contains a unique suffix *6503* (the leading 9 is decoded in the root level). The second level node[2] points to the third level.

The third level node[5] points to the last level with the remaining suffixes. Therefore, zip code 92515 is represented in the array by *level0[9]->level1[2]->level2[5]* and level 3 contains the unique suffix *15*. Zip code 92523 is represented in the array by *level0[9]->level1[2]->level2[5]* and level 3 contains the unique suffix *23*.

Unlike an analog tree (B-tree, binary tree, etc.) no comparisons are

necessary to search a digital tree until you reach the value. The index is divided into a series of array indexes until a unique index suffix is obtained.

Figure 2-1 **A ten-way digital tree**



Judy: a digital tree hybrid

Pure digital trees, like the one described above, typically exhibit very fast insertion and retrieval times; however, a digital tree often achieves fast performance at the cost of large memory use. A computation of 256^4 uses over four billion indexes, while a computation of 256^3 uses over 16 million indexes or at least 67 MB of storage (4 bytes/index).

Judy is a hybrid 256-way digital tree. Like a pure digital tree, each level in a Judy tree decodes one or more (8 bit) digits. Unlike a pure digital tree, each node is not necessarily a 256-way vector of elements.

Judy achieves scalability and excellent memory characteristics by choosing data structures for each node appropriate for the population below the node. For example, if you need a small associative array with 6 indexes, you would probably create a linear list to hold them. Judy does the same. If you need 600 million indexes, you would do something different. The same goes for Judy. As a Judy tree is populated, each node gets a data structure appropriate to the population under it.

To be this dynamic, Judy must know the population under any node. To

do this, Judy keeps internal population counts at each node. These counts are also the reason why you can call Judy1Count or JudyLCount to find all the indexes between *x* and *y* and get an answer back much faster than you can count a linear array of the same population.

Here is what you might find in an element of a Judy node:

Pointers	Like a traditional tree, Judy provides pointers to the next tree level.
Indexes	Because it is based on a digital tree, Judy always stores indexes in sorted order or infers them from their position if the node is an M-way vector. The order in which values are stored is based on numeric value.
Type	Judy uses type information to track the kind of structure being pointed to in the next level.
Count	The index population stored below this node element.

Since Judy inherits the advantages of a digital tree structure, a Judy tree with 32-bit indexes is a maximum of 4 levels deep. A Judy tree with 64-bit indexes is a maximum of 8 levels deep. When you consider that a 64-bit tree could have 1.845×10^{19} indexes (if you had enough memory), this is an efficient search and retrieval storage technique.

With *in-memory* data structures like the current version of Judy, the depth of the tree is only an indication the CPU cycles necessary to access the data. However, the time required to retrieve that data is even more important. Many search and retrieval algorithms have problems because memory cache-line fills frequently dominate in memory search and retrieval times. Modern processors have become so fast that it is advantageous to trade CPU cycles to avoid cache-line fills whenever possible. Judy is designed with this trade-off in mind. (See Appendix B, “Caching and Memory Management,” on page 61.)

Compared with Other Search and Retrieval Methods

The table below shows how Judy algorithms compare with other search and retrieval methods.

Table 2-1

Search and Retrieval method (data structure)	O number for lookup	Advantages	Disadvantages
Arrays		Fast indexing	Pre-allocation of entire array before storing a single element (inefficient for sparse arrays).
Hashing	Depends on algorithm	Fast access	• Synonym management is complex. • No near-neighbor lookup capability. • Must be tuned for size and nature of data being stored.
Binary Trees	O(log n)		Gets very deep with large data sets, requiring many memory accesses to reach target values.
AVL Trees (height-balanced binary trees)	O(log n)		• Expensive to keep balanced. • May also get very deep with large data sets, requiring many memory accesses to reach target values.
B-trees	O(log n)		• Slow to access. • Inherently difficult to balance.

Table 2-1

Search and Retrieval method (data structure)	O number for lookup	Advantages	Disadvantages
Skip Lists	O(log n)	• Simplify the tree-balancing problem by using a probabilistic method instead of a strictly enforced balancing method. • Their modest improvement in performance (over binary trees) is primarily due to the quaternary, rather than binary, nature of the data structure.	
Digital Trees (Trie)	O ($\log_m$ n)	Fast access	• Memory inefficient. • Requires access times proportional to the number of significant digits of the index value. The access time for storing an index that has a large number of significant digits can be lengthy.
Judy	Less than O($\log_{256}$ n)	• Fast access times. • Grows dynamically on demand. • Adjusts in size to the amount of stored data. • Minimal number of memory accesses. • Depth of tree does not increase significantly with the population of the tree. • Very fast lookup times for large sparse arrays. • Fast insertion, both average and worst case. • Requires no tree rebalancing. • External interface is similar to that of a sorted array. • Count available.	

Hashing versus JudyL

The figures below show JudyL compared to a commonly used hashing algorithm. The hash algorithm is a simple static table sized to be a *power of two* with a linear (linked list) collision chain. The table is a fixed size of 1,048,576 buckets (2^{20}). Each bucket is a pointer to a chain. Each chain element is 3 words (key, value, next pointer). The key, value and next pointer are all 64-bit quantities; therefore, the hash memory used should asymptotically approach 24 bytes per inserted data element.

To insert into the hash table, the key is hashed and then ANDed with the table size - 1 to find a bucket. It is also common to have a prime-number-sized hash table and use the *mod* function to get a table index. However, *mod* is very slow (~1 microsecond on the system we were using to acquire the above data) so we used the *power of two* table method instead. Chains are linear searched for a match. If there is no match, a new chain element is created. Finally, the data value is incremented (the data value becomes a usage count to verify the number of unique keys inserted).

The hash table was seriously overloaded by a factor of 40:1. Overloading is very undesirable but all too common with hashing. The resulting collision chains were an average of 39.15 in length, showing that our hash distribution was very good.

The hash table creation time is not shown in the data. Judy table creation is an integral part of JudyLIns since a Judy array starts at zero length and grows dynamically.

The hashing code for this benchmark was all *in-lined*. This is common with hashing but not a practical option with Judy. *Mallopt*(3C) was used to speed up the allocation of each hash element. Measurements were done on a 2GB HP 9000 J5000 workstation running HP-UX 11i at 440 MHz. The raw data is shown below.

NOTE

The data was collected using random numbers as keys. Random keys provide the best-case data for hashing and the *worst* case data for Judy. Judy's performance and memory usage will both improve with any data clustering.

Figure 2-2 Judy versus hash insertion

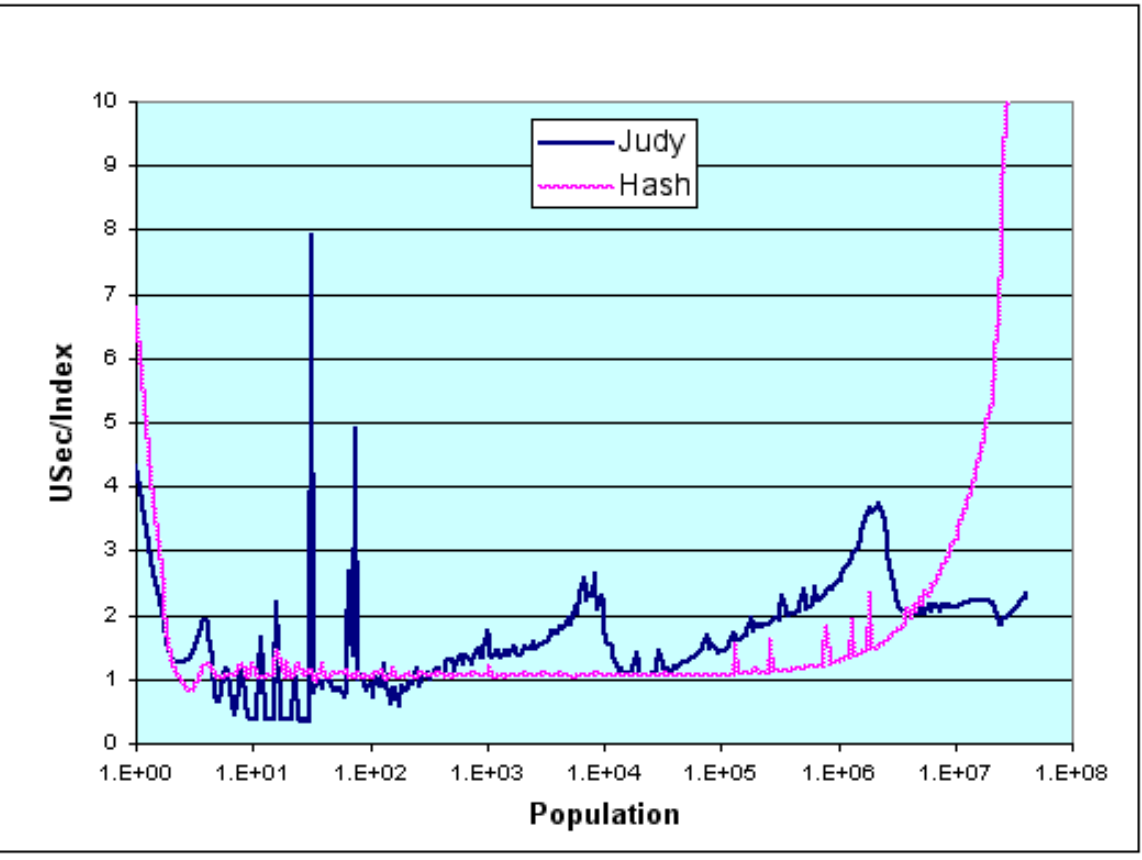


Figure 2-3 Judy versus hash retrieval

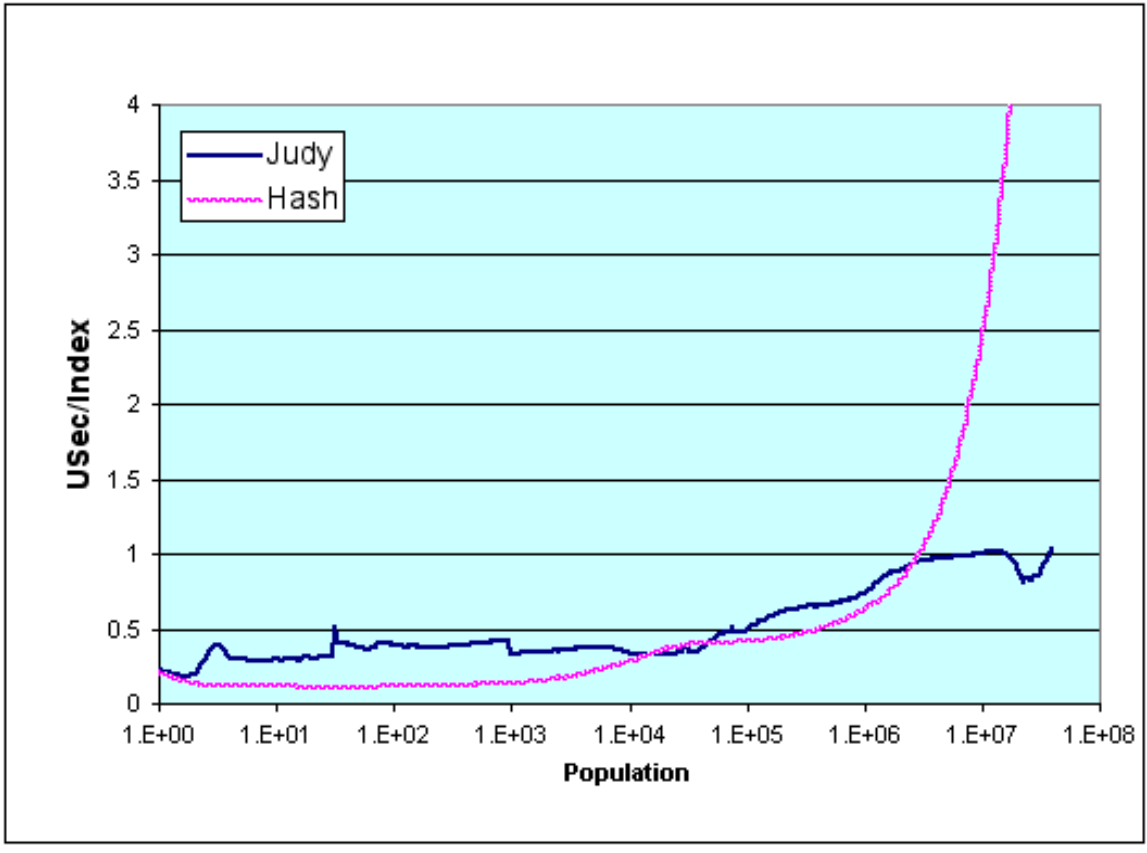
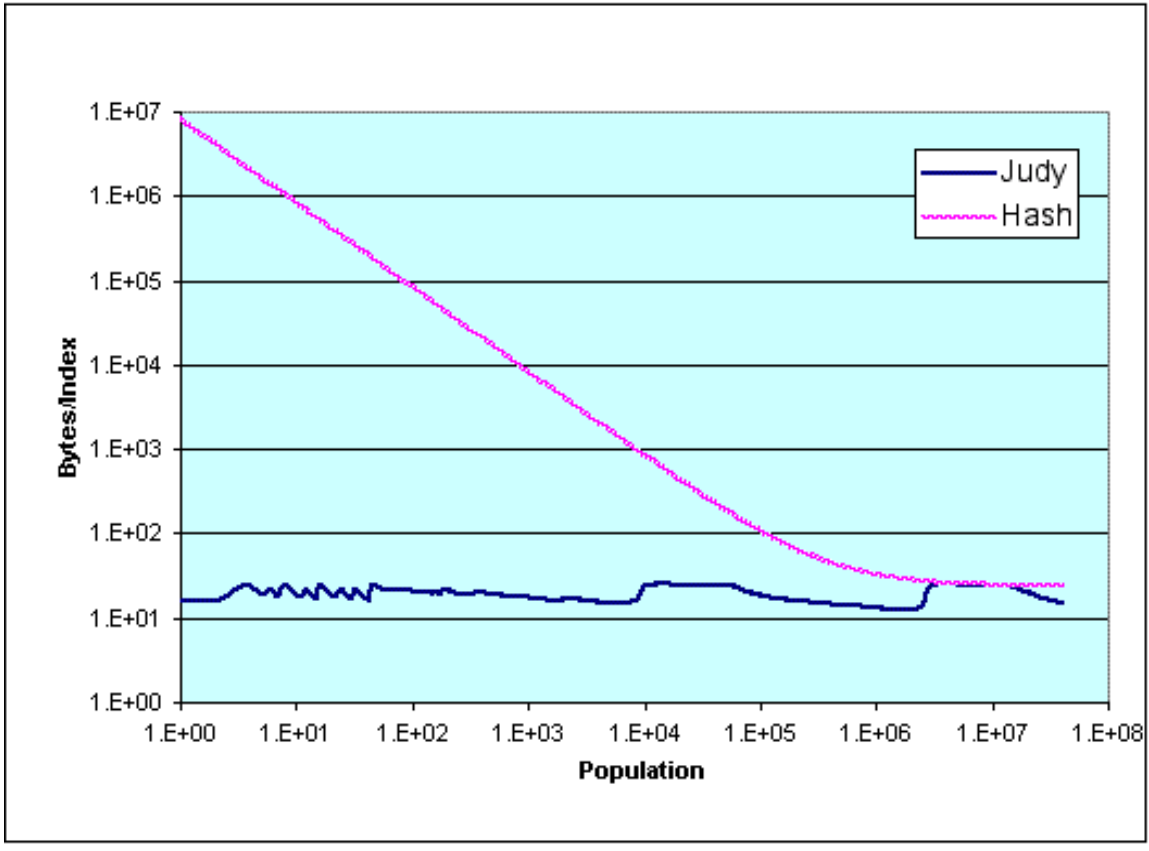


Figure 2-4 Judy versus hash memory performance



Most programmers understand that hashing with a static table can deliver excellent performance. However when you use hashing, you must remember to:

- Size your hash table correctly.
- Beware of operators like *mod* (i.e. %) on a PA-RISC system.
- Use any available optimizations, like *mallopt*(3C).
- In-line your code.

When you use Judy, you don't need to remember anything on this checklist! Judy takes care of all the optimizations for you.

In summary

Judy is a hybrid digital tree that chooses data structures for each node appropriate to the population below the node, thereby allowing tree-level compression and balanced memory usage.

Over large populations, the Judy advantage becomes clear when Judy is benchmarked against other search and retrieval methods, such as hashing.

3 Using Judy

The Judy library functions support small or large, dynamic arrays with 32- or 64-bit indexes depending on the processor type. Each type of Judy array has a different kind of index and/or value. You can think of a Judy array as:

```
value-type JudyArray [4294967296];
// or 2^32 on a 32-bit processor

value-type JudyArray [18446744073709551616];
// or 2^64 on a 64-bit processor
```

Judy arrays are quick to access and space efficient. A Judy array is created by storing (inserting) the first element and grows dynamically as elements are inserted.

The table below summarizes the Judy array types with functions provided. The Judy header file (*Judy.h*) provides definitions for these functions; the constants NULL, TRUE, and FALSE; and the macro equivalents for calling Judy functions.

Table 3-1 A summary of Judy functions

	Judy Array Type		
Function	**Judy1: maps ulong_t to bit**	**JudyL: maps ulong_t to ulong_t**	**JudySL: maps string to ulong_t**
Retrieve	Judy1Test	JudyLGet	JudySLGet
Set	Judy1Set	JudyLIns	JudySLIns
Clear/Delete	Judy1Unset	JudyLDel	JudySLDel
First Index	Judy1First	JudyLFirst	JudySLFirst
Last Index	Judy1Last	JudyLLast	JudySLLast
Previous Index	Judy1Prev	JudyLPrev	JudySLPrev
Next Index	Judy1Next	JudyLNext	JudySLNext

Table 3-1 A summary of Judy functions

	Judy Array Type		
Function	**Judy1: maps ulong_t to bit**	**JudyL: maps ulong_t to ulong_t**	**JudySL: maps string to ulong_t**
Count (between two indexes)	Judy1Count	JudyLCount	
Count (locate the *N*th index within an array, where *N*=Count)	Judy1ByCount	JudyLByCount	
Free Array	Judy1FreeArray	JudyLFreeArray	JudySLFreeArray

Sorting

As part of normal processing, Judy sorts indexes numerically using its own, internal sorting criteria. Therefore, you cannot modify the sorting order to include user-defined sorting criteria (such as requesting a sort using multi-byte characters).

While Judy is not primarily intended as a sorting algorithm, in many cases it's faster to store values in a Judy array and read them back in sorted order than to sort them using standard sorting algorithms. JudySL strings are treated as series of *N*-byte numbers (*N* = 4 on a 32-bit system or 8 on a 64-bit system).

Keep in mind that Judy arrays do not support synonyms, that is multiple values associated with the same index. The caller must build synonyms outside of Judy; for instance, by having the Judy value (from JudyL or JudySL) point to a linked list of nodes, each with a different value for the same index.

Each index can only be stored once. If you store value *xyz* and then store the value *xyz* again, Judy uses the same value area.

JudySL sorts two strings that start with the same prefix like *sort(1)* in UNIX (but without multi-byte support):

gazelle
gazette

If one string is a substring of another, the shorter one is first:

gazelle
gazelle2

JudySL sorts are case sensitive and JudySL literally stores strings as chunks of bytes or numbers. Sorting is numeric per the ASCII value of the character, so the letter "A" is sorted using a different numeric sequence than the letter "a".

For more information about sorting strings, see the “JudySL sorting example” on page 45.

Counting

You can use Judy counting functions to rapidly determine the number of valid indexes between any pair of indexes. Judy counts the population of any given expanse: the number of valid (stored) indexes in the expanse. Because each node in a Judy array maintains a count of the values under the node, Judy counts the number of keys between any two arbitrarily defined keys in nearly constant time.

Counting functions are available through Judy1Count() and JudyLCount(), which return counts of valid indexes between the specified indexes (including the indexes themselves if valid). In addition, you can use the JudyLByCount function to locate the *N*th index (where *N* = count) in an array and return a pointer to its value area.

Because counting happens in nearly constant time, Judy counting functions can support many novel problem solutions, such as repeated stack depth measurements. For more information about counting, refer to the *Judy1(3X)* and *JudyL(3X)* man pages.

Using Judy1

Judy1 functions represent Boolean values (bit maps) in a Judy array. These functions support arrays of one-bit (binary) values, representing true/false, valid/invalid, or present/absent.

The table below lists the Judy1 functions. For more information, see the *Judy1(3X)*man page.

Table 3-2

Judy1 functions	Actions
Judy1Test	Indicates whether the index is set in the array.
Judy1Set	Sets a bit in the array.
Judy1Unset	Clears a bit in the array.
Judy1First	Locates the first index set in the array (including the passed index).
Judy1Next	Typically continues a sorted order scan of the set indexes in a bit array or locates a neighbor of an index.
Judy1Last	Typically begins a reverse sorted-order scan of the set indexes in a bit array.
Judy1Prev	Typically continues a reverse sorted-order scan of the set indexes in a bit array or locates a neighbor of an index.
Judy1Count	Returns the count of set indexes between the specified indexes, including the specified indexes themselves.
Judy1ByCount	Locates the *N*th index (*N*=count) that is set in the bit array.
Judy1FreeArray()	Frees the entire bit array. This is much faster than using Judy1Next and Judy1Unset.

Judy1 example You can find the code for this example and others on the Judy web site: http://devresource.hp.com/judy/

```
// Judy 1 Example code: bit set operations.
// There should be a header file that defines the bit operation types:

#define JUDY1OP_AND    1L
#define JUDY1OP_OR     2L
#define JUDY1OP_ANDNOT 3L

#include "Judy.h"

/*********************************************************************
* Name: Judy1Op
* Description:
*       Logical set operations on Judy1 arrays.
*       All of these operations can be done on an unbounded array.
* Parameters:
*       PPvoid_t PPDest (OUT)
*            Ptr to the Judy destination array.
*            Any initial value pointed to by PPDest is ignored.
*       Pvoid_t PSet1 (IN)
*            First Judy1 set.
*            This will be NULL for an empty Judy1 array.
*       Pvoid_t PSet2 (IN)
*            Second Judy1 set.
*            This will be NULL for an empty Judy1 array.
*       ulong_t Operation (IN)
*            Operation to be performed (ie. PSet1 {Operation} PSet2)
*            Valid Operation values are:
*            JUDY1OP_AND     - intersection of two sets
*            JUDY1OP_OR      - union of two sets
*            JUDY1OP_ANDNOT  - set1 with set2 removed
```

```
*       JError_t * PJError (OUT)
*            Judy Error struct used to return Judy error.
* Returns:
*      !JERR if successful
*       JERR if an error occurs
 *       If the error is a caller error (invalid Operation or no PPDest)
 *       then the PJError error code will be JU_ERRNO_NONE.
 */
int Judy1Op (PPvoid_t PPDest, Pvoid_t PSet1, Pvoid_t PSet2,
                  ulong_t Operation, JError_t * PJError)
{
   Pvoid_t  PnewJArray = 0;  // empty Judy array
   ulong_t  Index1 = 0L;
   ulong_t  Index2 = 0L;
   int      Judy_rv;

   if (!PPDest) return JERR;

   switch (Operation)
   {
      case JUDY1OP_AND:
         // step through each array looking for index matches
         Judy_rv = Judy1First(PSet1, &Index1, PJError);
         Judy_rv += Judy1First(PSet2, &Index2, PJError);
         while(Judy_rv == 2)
         {
            if (Index1 < Index2)
            {
               Index1 = Index2;
               Judy_rv = Judy1First(PSet1, &Index1, PJError);
            }
            else
```

```
        if (Index1 > Index2)
        {
            Index2 = Index1;
            Judy_rv = Judy1First(PSet2, &Index2, PJError);
        }
        else
        {
            // do the AND
            Judy_rv = Judy1Set(&PnewJArray, Index1, PJError);
            if (Judy_rv == JERR) return JERR;

            // bump to the next bits
            Judy_rv  = Judy1Next(PSet1, &Index1, PJError);
            Judy_rv += Judy1Next(PSet2, &Index2, PJError);
        }
    }
    *PPDest = PnewJArray;
    break;

case JUDY1OP_OR:
    /* Set all the bits from PSet1 */
    for (Index1 = 0L, Judy_rv = Judy1First(PSet1, &Index1, PJError);
         Judy_rv == 1;
         Judy_rv = Judy1Next(PSet1, &Index1, PJError))
    {
        if (Judy1Set(&PnewJArray, Index1, PJError) == JERR)
            return JERR;
    }

    /* Set all the bits from PSet2 */
    for (Index1 = 0L, Judy_rv = Judy1First(PSet2, &Index1, PJError);
         Judy_rv == 1;
```

```
            Judy_rv = Judy1Next(PSet2, &Index1, PJError))
        {
           if (Judy1Set(&PnewJArray, Index1, PJError) == JERR)
              return JERR;
        }
        *PPDest = PnewJArray;
        break;

    case JUDY1OP_ANDNOT:
        // PSet1 with PSet2 removed
        // 0010 = PSet1(1010) ANDNOT PSet2(1100)

        for (Index1 = 0L, Judy_rv = Judy1First(PSet1, &Index1, PJError);
             Judy_rv == 1;
             Judy_rv = Judy1Next(PSet1, &Index1, PJError))
        {
           // if bit doesn't exist in PSet2, then add to result
           if (0 == Judy1Test(PSet2, Index1, PJError))
           {
              if (Judy1Set(&PnewJArray, Index1, PJError) == JERR)
                 return JERR;
           }
        }
        *PPDest = PnewJArray;
        break;

    default:
        return JERR;
   }
   return !JERR;
}  /* Judy1Op */
```

Using JudyL

JudyL functions provide a way to represent long-word values in a Judy array. These functions support large arrays of 32-bit or 64-bit values (long words). Since an initial (empty) JudyL array is represented by a null pointer, you can make JudyL arrays multi-dimensional, the equivalent of:

```
void * JudyLArray [4294967296][4294967296]...;
```

The table below shows the JudyL functions. For more information, see the *JudyL(3X)* man page.

Table 3-3

JudyL functions	Actions
JudyLGet	Retrieves a value from a JudyL array.
JudyLIns	Inserts a value into a JudyL array.
JudyLDel	Deletes the specified JudyL array element (index/value pair).
JudyLFirst	Typically begins a sorted-order scan of the set indexes in a JudyL array.
JudyLNext	Typically continues a sorted-order scan of the set indexes in a JudyL array or locates a neighbor of an index.
JudyLLast	Typically begins a reverse sorted-order scan of the set indexes in a JudyL array.
JudyLPrev	Typically continues a reverse sorted-order scan of the set indexes in a JudyL array or locates a neighbor of an index.

Table 3-3

JudyL functions	Actions
JudyLCount	Returns the count of set indexes between two specified indexes, including the specified indexes themselves.
JudyLByCount	Locates the *N*th index (*N* = count) that is set in the JudyL array.
JudyLFreeArray	Frees the entire JudyL array. This is much faster than using JudyLNext and JudyLDel.

JudyL example

You can find the code for this example and others on the Judy web site: http://devresource.hp.com/judy/

```
// FUNCTION HISTOGRAM; JUDYL DEMO PROGRAM
//
// This program uses JudyL to create a histogram of data generated by the
// function random(3M).  Other functions could be substituted in.
//
// The program generates n random numbers, stores them in a JudyL array
// and then prints a histogram.  All aspects of the generation and
// histogram are timed so the user can see how long various JudyL functions
// take.
//
// This demonstrates:
//           JudyLIns
//           JudyLFirst
//           JudyLNext
//           JudyLLast
//           JudyLCount
//           JudyLByCount
//           how to access a JudyL value.
//
// Notice that using JudyL gives you fast stores and lookups as with hashing
// but without having to define a hash function, initialize the hash table
```

```
// or having to predetermine a good hash table size.  It also gives you a
// sorted list at the same time. Also notice that JudyLCount is much faster than you
// can sequentially count items in an array.
//

#include <stdio.h>
#include <sys/time.h>
#include <errno.h>
#include "Judy.h"

// Default number of iterations (number of random numbers generated)
// This may be overridden on the command line

#define DEFAULT_ITER 1000000

// The number of buckets the histogram is divided into.

#define DEFAULT_HISTO_BUCKETS 32

// Macro for correction english output for plurals

#define PLURAL(count) ((count == 1) ? "" : "s")

// Generic fatal error handler, pass in JError_t

#define JUDY_ERROR(J) \
        fprintf(stderr, "File: '%s', line: %d: Judy error: %d\n", \
                __FILE__, __LINE__, JU_ERRNO(&(J))); \
        exit(2);

// timing routines
```

```
struct timeval TBeg, TEnd;
#define STARTTm         gettimeofday(&TBeg, NULL)
#define ENDTm           gettimeofday(&TEnd, NULL)
#define DeltaUSec \
        ((double)(TEnd.tv_sec * 1000000.0 + TEnd.tv_usec) \
        - (TBeg.tv_sec * 1000000.0 + TBeg.tv_usec))
// End of timing routines


main( int argc, char **argv)
{
   void    *PJArray = 0;        // main JudyL array
                                // key is random number, value is repeat count
   void    *PJCountArray = 0;   // second JudyL array
                                // key is repeat count (from PJArray)
                                // value is count of items with the same
                                // repeat count
   JError_t JError;             // Judy error structure
   ulong_t  Index;              // index in JudyLFirst/Next loop
   ulong_t *PValue;             // ptr to Judy array value
   ulong_t *PGenCount;          // ptr to generation count
   ulong_t  num_vals;           // number of randoms to generate
   ulong_t  iter;               // for loop iteration
   ulong_t  unique_nums;        // number of unique randoms generated
   ulong_t  random_num;         // random number
   ulong_t  median;             // random number
   ulong_t  tmp1, tmp2;         // temporary variable
   double   random_gen_time;    // time to generate random numbers
   ulong_t  histo_incr;         // histogram increment
   unsigned long long ran_sum = 0;  // sum of all generated randoms

// TITLE
```

```
   (void) puts ("\nJudyL demonstration: random(3M) histogram");

// SET NUMBER OF RANDOMS TO GENERATE

   if (argc != 2)
   {
      // SET TO DEFAULT_ITER
      num_vals = DEFAULT_ITER;
      (void) printf ("Usage: %s [numvals]\n", argv[0]);
      (void) printf ("  Since you did not specify a number of values, %d\n",
                     num_vals);
      (void) puts   ("  will be used as the number of random numbers to insert");
      (void) puts   ("  into the Judy array");
   }
   else
   {
// OVERRIDE NUMBER OF RANDOMS TO GENERATE

      num_vals = atol(argv[1]);
   }

// TIME THE GENERATION OF ALL THE RANDOM NUMBERS.  THIS TIME IS LATER
//
// This time is later subtracted from the insert loop time so that the
// time it takes to do the actual JudyLIns can be isolated from the
// total time.

   (void) puts ("\ntiming random number generation");
   STARTTm;
   for (iter = num_vals; iter; iter--)
   {
```

```
      random();
   }  /* end of random number generator time */
   ENDTm;
   random_gen_time = DeltaUSec;
   (void) printf("It took %.3f sec to generate %ld random numbers\n",
     random_gen_time/1000000, num_vals);
   (void) printf("  (ie. %.3f uSec/number)\n\n", random_gen_time/num_vals);

// REGENERATE RANDOMS AND INSERT THEM INTO A JUDYL ARRAY

   (void) puts("Please wait while the random numbers are inserted into");
   (void) puts("a JudyL array (with a usage count) ...");

   STARTTm;
   for (iter = num_vals; iter; iter--)
   {
      random_num = (ulong_t)random();
      if ((PValue = (ulong_t *)JudyLIns(&PJArray, random_num, &JError))
          == PJERR)
      {
         JUDY_ERROR(JError);  // exits
      }

      /* increment hit count */
      (*PValue)++;

      /* sum the random number */
      ran_sum += random_num;
   }  /* end of random number generator time */
   ENDTm;
   (void) printf("That took %.3f uSec/Index.\n",
                (DeltaUSec - random_gen_time)/num_vals);
```

```
// COUNT THE NUMBER OF ELEMENTS IN THE JUDYL ARRAY
// IE. COUNT THE NUMBER OF UNIQUE RANDOM NUMBERS

   STARTTm;
   unique_nums = JudyLCount(PJArray, 0, ~0L, &JError);
   ENDTm;
   (void) printf("\nThere were %ld unique random numbers generated\n",
                 unique_nums);
   (void) printf("It took %.3f uSec to count them in the Judy array.\n",
                 DeltaUSec);


// FIND HOW MANY NUMBERS WERE GENERATED ONCE, TWICE, ...
//
// Create a new JudyL array where the index is the count from PJArray
// and the value is a count of the number of elements with that count.

   if (unique_nums != num_vals)
   {
      (void) puts("\nLooping through the entire Judy array to create a");
      (void) puts("new Judy counting array (PJCountArray in the source code)");
      (void) puts("...");
      STARTTm;
      for (Index = 0L,
           PValue = (ulong_t *)JudyLFirst(PJArray, &Index, &JError);
           PValue;
           PValue = (ulong_t *)JudyLNext(PJArray, &Index, &JError))
      {
if ((PGenCount = (ulong_t *)JudyLIns(&PJCountArray, *PValue, &JError)) == PJERR)
         {
            JUDY_ERROR(JError);  // exits
```

```
        }
        (*PGenCount)++; // increment hit count
    }
    ENDTm;

    (void) printf("That took %.3f Secs or %.3f uSec/Index\n\n",
                    DeltaUSec/1000000, DeltaUSec/unique_nums);

// PRINT DUPLICATES HISTOGRAM

    (void) puts("Duplicates Histogram:");
    for (Index = 0L,
         PValue = (ulong_t *)JudyLFirst(PJCountArray, &Index, &JError);
         PValue;
         PValue = (ulong_t *)JudyLNext(PJCountArray, &Index, &JError))
    {
        (void) printf("  %ld numbers were generated %ld time%s\n",
                    *PValue, Index, PLURAL(Index));
    }
  }


// PRINT DISTRIBUTION HISTOGRAM

  (void) puts("\nCompute the random number distribution by counting index");
  (void) puts("ranges.");

  histo_incr = ((ulong_t)~0L / DEFAULT_HISTO_BUCKETS) >> 1;

  Index = 0L;
  for (iter = 0; iter < DEFAULT_HISTO_BUCKETS; iter++)
  {
```

```
      (void) printf("  %ld unique values from %8x - %8x\n",
                JudyLCount(PJArray, Index, Index + histo_incr, &JError),
                Index, Index + histo_incr);
      Index += histo_incr + 1;

   }


// PRINT MEAN (average),
//       MEDIAN (middle value, or average of two middle values)
//       RANGE (low and high value)

   tmp1 = (ulong_t)(ran_sum/(long long)num_vals);
   (void) printf("                  mean: 0x%08x\n", tmp1);

// If there were an even number of randoms generated, then average
// the two middle numbers.  Otherwise, the mean is the middle value
   if (num_vals & 1)
   {
      JudyLByCount(PJArray, num_vals/2, &tmp1, &JError);
      JudyLByCount(PJArray, (num_vals/2)+1, &tmp2, &JError);
      median = (tmp1 + tmp2) / 2;
   }
   else
   {
      JudyLByCount(PJArray, (num_vals+1)/2, &median, &JError);
   }
   (void) printf("                median: 0x%08x\n", median);

   Index = 0;
   JudyLFirst(PJArray, &Index, 0);
   (void) printf("first random generated: 0x%08x\n", Index);
```

```
    Index = ~0;
    JudyLLast(PJArray, &Index, 0);
    (void) printf(" last random generated: 0x%08x\n", Index);

    exit(0);
    /*NOTREACHED*/

}  // main()
```

Sample output

```
$ funhist 10000000

JudyL demonstration: random(3M) histogram

timing random number generation
It took 0.416 sec to generate 10000000 random numbers
  (ie. 0.042 uSec/number)

Please wait while the random numbers are inserted into
a JudyL array (with a usage count) ...
That took 1.619 uSec/Index.

There were 10000000 unique random numbers generated
It took 76.000 uSec to count them in the Judy array.

Compute the random number distribution by counting index
ranges.
  311914 unique values from        0 -  3ffffff
  311770 unique values from  4000000 -  7ffffff
  311556 unique values from  8000000 -  bffffff
  312178 unique values from  c000000 -  fffffff
```

```
  310960 unique values from 10000000 - 13ffffff
  311522 unique values from 14000000 - 17ffffff
  312201 unique values from 18000000 - 1bffffff
  312361 unique values from 1c000000 - 1fffffff
  312487 unique values from 20000000 - 23ffffff
  311927 unique values from 24000000 - 27ffffff
  313360 unique values from 28000000 - 2bffffff
  312025 unique values from 2c000000 - 2fffffff
  311236 unique values from 30000000 - 33ffffff
  310973 unique values from 34000000 - 37ffffff
  310617 unique values from 38000000 - 3bffffff
  312275 unique values from 3c000000 - 3fffffff
  311417 unique values from 40000000 - 43ffffff
  312189 unique values from 44000000 - 47ffffff
  311290 unique values from 48000000 - 4bffffff
  311570 unique values from 4c000000 - 4fffffff
  311224 unique values from 50000000 - 53ffffff
  311750 unique values from 54000000 - 57ffffff
  311372 unique values from 58000000 - 5bffffff
  311764 unique values from 5c000000 - 5fffffff
  311227 unique values from 60000000 - 63ffffff
  311947 unique values from 64000000 - 67ffffff
  311845 unique values from 68000000 - 6bffffff
  311913 unique values from 6c000000 - 6fffffff
  311941 unique values from 70000000 - 73ffffff
  311813 unique values from 74000000 - 77ffffff
  311703 unique values from 78000000 - 7bffffff
  335673 unique values from 7c000000 - 7fffffff
                 mean: 0x3fff3e9c
               median: 0x40220000
first random generated: 0x0000012b
 last random generated: 0x7fffff10
```

Using JudySL

The JudySL functions provide a way to use string values as indexes into a Judy array. In JudySL arrays, elements are sorted lexicographically (case-sensitive) by indexes:

```
void * JudySLArray ["Toto, I don't think we're in Kansas any more"];
```

Since an initial (empty) JudySL array is represented by a null pointer, you can make JudySL arrays multi-dimensional:

```
void * JudySLArray [char *][char *]...;
```

The table below shows the JudySL functions. For more information, see the *JudySL(3X)* man page.

Table 3-4

JudySL functions	Actions
JudySLGet	Retrieves a value from a JudySL string array.
JudySLIns	Inserts a value into a JudySL array.
JudySLDel	Deletes the specified JudySL array element.
JudySLFirst	Typically begins a sorted-order scan of the set indexes in a JudySL array.
JudySLNext	Typically continues a sorted-order scan of the set indexes in a JudySL array or locates a neighbor of an index.
JudySLLast	Typically begins a reverse sorted-order scan of the set indexes in a JudySL array.
JudySLPrev	Typically continues a reverse sorted-order scan of the set indexes in a JudySL array or locates a neighbor of an index.
JudySLFreeArray	Frees the entire JudyL array. This is much faster than using JudySLNext and JudySLDel.

JudySL sorting example

You can find the code for this example and others on the Judy web site: http://devresource.hp.com/judy/

```
// JudySort.c
// Judy demonstration code:  Judy equivalent of sort -u.  While Judy is not
// primarily intended as a sorting algorithm, in many cases it's faster to
// store values in a Judy array and read them back in sorted order than to sort
// them using standard algorithms.
//
// Usage:  <program> <file-to-sort>
//
// Code and comments are included if you want to modify the program to output
// duplicates as an exercise.
//
// Note:  If an input line is longer than BUFSIZ, it's broken into two or more
// lines.

#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>

#include "Judy.h"

main (int argc, char ** argv)
{
    Pvoid_t  PJArray = (Pvoid_t) NULL;  // Judy array.
    PPvoid_t PPValue;                   // Judy array element.
    char     Index [BUFSIZ];            // string to sort.
    char *   Pc;                        // place in Index.
    FILE *   fpin;                      // to read file.
    JError_t JError;                    // Judy error structure
```

```
// CHECK FOR REQUIRED INPUT FILE PARAMETER:

    if (argc != 2)
    {
        (void) fprintf (stderr, "Usage: %s <file-to-sort>\n", argv[0]);
        (void) fputs   ("Uses each line as a JudySL index.\n", stderr);
        exit (1);
    }

// OPEN INPUT FILE:

    if ((fpin = fopen (argv[1], "r")) == (FILE *) NULL)
    {
        (void) fprintf (stderr, "%s: Cannot open file \"%s\": %s "
                        "(errno = %d)\n", argv[0], argv[1], strerror (errno),
                        errno);
        exit (1);
    }

// INPUT/STORE LOOP:
//
// Read each input line (up to Index size) and save the line as an index into a
// JudySL array.  If the line doesn't overflow Index, it ends with a newline,
// which must be removed for sorting comparable to sort(1).

    while (fgets (Index, sizeof (Index), fpin) != (char *) NULL)
    {
        if ((Pc = strchr (Index, '\n')) != (char *) NULL)
            *Pc = '\0';                         // trim at newline.

        if ((PPValue = JudySLIns (& PJArray, Index, &JError))
```

```
            == PPJERR)
        {
            fprintf(stderr, "File: '%s', line: %d: Judy error: %d\n",
                    __FILE__, __LINE__, JU_ERRNO(&JError));
            exit (1);
        }

// To modify the program to output duplication counts, like uniq -d, or to emit
// multiple copies of repeated strings:

#ifdef notdef
        ++(*((ulong_t *) PPValue));
#endif
    }
// PRINT SORTED STRINGS:
//
// To output all strings and not just the unique ones, output Index multiple
// times = *((ulong_t *) PPValue).

    Index[0] = '\0';                            // start with smallest string.

    for (PPValue  = JudySLFirst (PJArray, Index, &JError);
         PPValue != (PPvoid_t) NULL;
         PPValue  = JudySLNext  (PJArray, Index, &JError))
    {
        (void) puts (Index);                    // see comment above.
    }

    exit (0);
    /*NOTREACHED*/

} // main()
```

In summary

The Judy technology provides three array types each supporting a different kind of index and/or value. These are:

Judy1	Map ulong_t to bit
JudyL	Map ulong_t to ulong_t
JudySL	Map char* to ulong_t

Each of these array types have associated functions described in Table 3-1 on page 25.

4 Example of a Multi-dimensional Array

The JudySLGet and JudySLIns functions shown in this chapter illustrate a multi-dimensional JudyL array. This code is the proof of concept code for the Judy Version 4.0 functions: JudySLGet() and JudySLIns(). Although it is similar to the current JudySL, this code should not be used with released JudySL functions.

Concepts

Copy each null-terminated string (index) into an array of *N* words (ulong_t's) with zero-padding in the last word if necessary. Use each word in this array as a level index in a multi-dimensional Judy array. For example, consider these two input strings:

"abcdefghijk"

"abcdefgk"

These are used as if they are arrays of 32- bit integers with the following keys:

"abcd", "efgh", "ijk0"
"abcd", "efgk", "0000"

Since a JudyL data element has both a key and a value, we can use the value to either point to another Judy array, or if the key contains a NULL, we can use the value area for user data. With this logic, the two strings above produce four Judy arrays:

Notes:

- Only 32 bit versions of the WORDCPY macros are shown in this example.
- You can find the code for this example and others on the Judy web site: `http://devresource.hp.com/judy/`

Example Code

```
#include "Judy.h"

#define TRUE  1L

#define CHNULL  '\0'
#define PCNULL  ((char *)   NULL)
#define PPVNULL ((PPvoid_t) NULL)

#define WORDSIZE (sizeof (ulong_t))     // bytes in a word = JudyL index.

// Copy a word from one location to another; typically one of which is not
// word-aligned:
//
// Like strncpy(), null-pad the destination, but based on performance
// measurements don't use strncpy().  Instead hardwire the code based on a
// parameter from JudyL.h; assume only two machine word sizes exist.

#define    CH(P,I)      (((char *) (P)) [I])    // shorthand for below.
#define   CPY(P1,P2,I)  (CH (P1, I) = CH (P2, I))
#define IFCPY(P1,P2,I)  if (CPY (P1, P2, I) != CHNULL)

#define WORDCPY(P1,P2) \
    { *((ulong_t *) (P1)) = 0L; \
      IFCPY(P1,P2,0L) IFCPY(P1,P2,1L) IFCPY(P1,P2,2L) CPY(P1,P2,3L); }

#define LASTWORD(indexword) (((char *) (& indexword)) [WORDSIZE - 1L] == CHNULL)
```

```
// ****************************************************************************
// J U D Y   S L   G E T

PPvoid_t JudySLGet (
        const void * PArray,
        const char * Index,
        JError_t   * PJError)
{
        char *   pos = (char *)Index;   // place in Index.
        ulong_t  indexword;     // next word to find.
        PPvoid_t PPvalue;       // from JudyL array.

assert (Index != PCNULL);



// SEARCH NEXT LEVEL OF JUDYL ARRAY IN TREE:
//
// Copy each word from the Index string and check for it in the next level
// JudyL array in the array tree.

        while (TRUE)                            // until return.
        {
// Since strings can begin on arbitrary byte boundaries, copy each
// chunk of N bytes from Index into a word-aligned object (a ulong_t)
// before using it as a Judy index.

            WORDCPY (& indexword, pos);

            if ((PPvalue = JudyLGet (PArray, indexword, PJError)) == PPJERR)
                return (PPVNULL);               // Index not in array tree.
```

```
// CHECK FOR END OF STRING IN CURRENT indexword:

            if (LASTWORD (indexword))
                return (PPvalue);       // is value for whole Index string.


// CONTINUE TO NEXT LEVEL DOWN JUDYL ARRAY TREE:
//
// If a previous JudySLIns() ran out of memory partway down the tree, it left a
// null *PPvalue; this is automatically treated here as a dead-end.

            pos   += WORDSIZE;
            PArray = *PPvalue;          // each value -> next array.

        } // while

        /*NOTREACHED*/

} // JudySLGet()


// ****************************************************************************
// J U D Y   S L   I N S

PPvoid_t JudySLIns (
        PPvoid_t     PPArray,
        const char * Index,
        JError_t   * PJError)
{
        char *   pos = (char *)Index;
        ulong_t  indexword;
        PPvoid_t PPvalue;
```

```
assert (PPArray != PPVNULL);
assert (Index   != PCNULL);

        while (TRUE)                            // until return.
        {
            WORDCPY (& indexword, pos);

            if ((PPvalue = JudyLIns (PPArray, indexword, PJError)) == PPJERR)
                return (PPVNULL);               // Index cannot be stored.

            if (LASTWORD (indexword))
                return (PPvalue);       // is value for whole Index string.

            pos    += WORDSIZE;
            PPArray = PPvalue;                  // each value -> next array.
        }

        /*NOTREACHED*/

} // JudySLIns()
```

In summary

This chapter contains an example that illustrates a Judy multi-dimensional array. Excellent memory efficiency makes multi-dimensional arrays a practical application of the Judy technology that may be “best in class”.

5 Tuning HP-UX Memory Access

Like other kinds of arrays, Judy arrays can potentially consume large amounts of memory. Because of this you should be aware of some HP-UX features that directly affect memory performance and the amount of memory you can access.

The *chatr* Command

The *chatr* command changes a program's internal attributes and can be used to modify the data page size for a specific executable. The larger the page size, the fewer virtual-to- physical, memory-page translations your process has to undergo. This can have a dramatic effect (sometimes *2X*) on the performance of data-intensive (RAM) applications.

For programs that use less than 64 MB of data space, use

chatr +pd L a.out

to allow the page size to grow from 16 Kb to 32 Kb or 64 Kb and up to the maximum it needs.

For programs that use more than 64 MB of data space, use

chatr +pd 64M a.out

This command causes the *a.out* to start with large pages (64 MB) and limits the page size to 64 MB, so that page sizes that are too large cannot cause a premature out of memory condition. For example, if you are using a 1GB page size and only have 0.99 GB left, the page allocation may fail (you run out of memory even though you have almost a gigabyte left).

For over 2GB of RAM, use *chatr* to enable the use of data space above 2GB:

chatr +q3p enable (only needed on a 32-bit *a.out*)

For over 3GB of RAM, use:

chatr +q4p enable (only needed on a 32-bit *a.out*)

See the *chatr(1)* man page for more information.

Kernel Tunable Parameters

Use the following parameters to tune the HP-UX kernel for more memory availability:

maxdsize	Limits the data-segment size of 32-bit processes. Limiting data segment size prevents programs from using up all available memory; however, if your executable needs memory, make sure this parameter is set high enough.
maxdsiz_64bit	Limits the data-segment size of 64-bit processes.
dbc_max_pct	Sets the maximum size of the kernel dynamic buffer cache. A large buffer cache (used to cache file system data) is very important for overall performance of most systems; however, a large buffer cache can take memory away from user programs. For machines with large amounts of memory you can free up memory for user processes by limiting the cache size to a couple of hundred megabytes (10% on a machine with 2GB RAM.

Compiler Switch

The -*N* C compiler switch (32-bit only) is passed through to the linker and directs it to lower the start of the heap to immediately after the text (code) segment. This allows you almost a 1GB larger data segment, but makes your *a.out* unsharable. See *ld(1)* for more information.

Swap Partition

This is the disk partition used for a system swap area. If your swap area is too small, the HP-UX kernel reserves some real memory to make sure it never “starves” for memory. This memory reservation takes away memory from user applications. To avoid this, set up a swap area that is the size of your RAM (either as a dedicated disk partition or through SAM). For example, if you have 2GB of RAM, use a 2GB swap area.

In summary

To ensure that Judy has good memory performance and access, use the following to tune your HP-UX system:

- The *chatr* command
- Kernel tunable parameters: *maxdsize* or *maxdsiz_64bit*, and *dbc_max_pct*
- -*N* C compiler switch (32-bit only)
- Disk partition for swapping

A Where Did Judy Come From?

The technological innovations that eventually became the Judy code were designed, implemented, and tested through more than a decade of research and development at Hewlett-Packard. The time line below shows the evolution of the Judy technology in HP internal products and code releases to the HP programming community.

1981	Some of the basic concepts of the Judy technology evolved from the RTE-6 Virtual Memory project (firmware/software implementation of large data indexing).
1991	Developing technology used in an internal performance profiling tool project (saving profiles in digital trees).
1994	The Judy array structure discovered (digital trees used instead of arrays for better performance).
1997	Judy re-engineered to handle random keys efficiently. Digital trees did not work well for random keys.
1997	HP funds Judy for internal development (developed efficient methods for storing random keys in digital trees).
1999	Judy used in an internal HP application that required the development of counting digital trees.
2000	HP staffs Judy as a project. In March 2000, a fully functional, untuned concept version of Judy is available internally.
2001	Judy Version 4.0, also known as Judy IV, available for HP-UX 11i during March 2001.

Judy was invented at HP's UNIX Software Enablement Laboratory at Fort Collins Colorado. Hewlett-Packard has patents pending on the Judy Technology.

Why is it called Judy?

We tried many other names and acronyms such as Sparse Array Memory Manager (SAMM) or Associative Memory Lookup Routine (AMLR), but none of them were adopted by the HP engineering community. The inventor, Doug, realized his relationship with the algorithm paralleled the relationship with his sister— Judy.

Figure A-1 **Doug with his sister Judy**

B Caching and Memory Management

Why use caching?

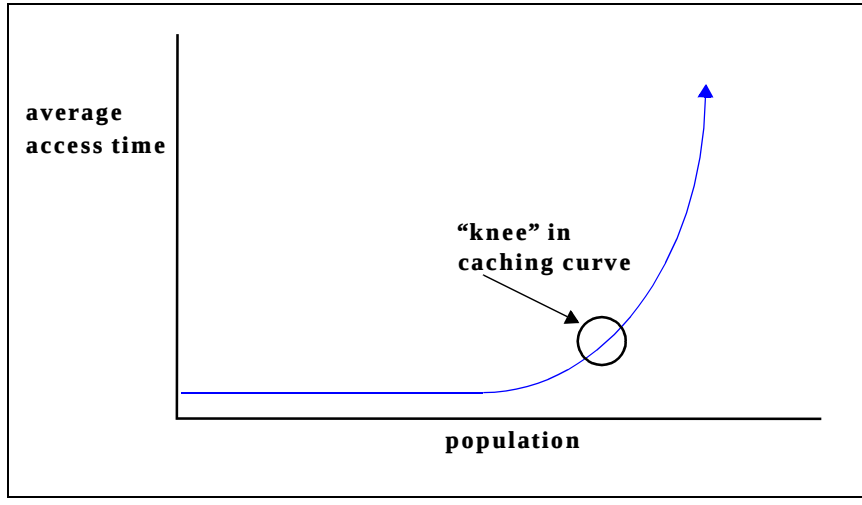
Cache is a small, fast memory holding recently accessed data. The purpose of caching is to avoid unnecessary reloads of *hot* data that is likely to be used soon or frequently. CPU caching is usually used to access processor memory but other types of caching can also be used for a local copy of data accessible over a network (such as web browser caching or disk buffer caching).

CPU “cache lines” exist because CPUs are becoming so much faster than RAM. When a computer must fill a cache line by accessing RAM (*cache line fill*), it can cause a 30-150 instruction delay (freeze) during which the CPU has to wait. Detailed profiles reveal that many programs spend a lot of time waiting for load and store instructions.

The caching curve

The figure below shows the *knee* in the caching curve that often occurs when the population (the number of valid keys or indexes) grows beyond a certain point. Note that as the population increases, the time to access any index in the population also increases exponentially showing a strong upward trend in the graph at a certain point.

Figure B-1 The caching curve



What happened here? The software requires access to data that is not in cache. Most of the time this means having to access an index that is not in cache memory and having to fill cache from main memory. The more frequently cache-line fills are required the more performance degrades over time. If this gets bad enough, the computer spends most of its time *thrashing* by repeatedly accessing main memory. Hence the saying, "Thrashing is virtual crashing."

Caching and memory efficiency

Judy uses minimal memory to achieve efficient access to stored data. Instead of caching data on local disk or network server, Judy arrays are memory resident.

Judy is cache-line optimized software: able to retrieve lines of cached data directly from the computer's cache with maximum speed. Judy minimizes cache-line fills, accessing an average of 4 to 6 cache-line fills to get to any value in an array. That means you can store one, one hundred, or 4 billion values in a Judy array. On average, Judy requires three cache-line fills to reach the data in a four gigabyte (4GB) element array. A one terabyte (1TB) element array would add one cache line fill.

Judy will never take more than 6 levels of indirection to reach the data.

Judy also minimizes the amount of memory required for array indexes. The amount of memory used is proportional to the array population, the density of the data, and the clustering of the data. JudyL was designed to allocate not more than 12 bytes per index for 32-bit machines and 24 bytes per index for 64-bit machines. For JudyL, eight bytes of memory or less per index is easily achievable on a 32-bit machine (16-bytes or less on a 64-bit machine), a ratio that is better than most other in-memory search and retrieval methods.

Glossary

Terms in this glossary attributed to (NIST) are derived from the National Institute of Standards and Technology (NIST) glossary.

Other terms are derived from the *Free Online Dictionary of Computing* and *whatis?com*.[1]

A

abstract data type (ADT) A mathematically specified collection of data-storing entities with operations to create, access, or change instances. (NIST) *See also data structure.*

algorithm A computable set of steps to achieve a desired result. The word comes from the Persian author Abu Ja'far Mohammed ibn Mûsâ al-Khowârizmî who wrote a book with arithmetic rules dating from about 825 A.D. (NIST)

array A set of items that are randomly accessible by numeric index. (NIST)

API See Application Programming Interface.

Application Programming Interface (API) The interface (calling conventions) by which an application program accesses an operating system and other services.

B

B-tree A balanced search tree in which every node has between *t-1* and *2t-1* children, where *t* is an arbitrary constant. This is a good structure if much of the tree is in slow memory (disk), since the height, and hence the number of accesses, can be kept small, say one or two, by picking a large *t*. (NIST)

big-O notation A theoretical measurement used to compare two algorithms that accomplish the same task, usually in terms of the time or memory needed.

binary tree A tree with at most two children for each node. (NIST)

bit array An array whose values are bits.

C

cache A small fast memory holding recently accessed data, designed to speed up subsequent access to the same data. Most often applied to CPU access but also used for a local copy of data accessible over a network or disk.

cache line A block of memory local to a CPU chip that caches the same sized block from the

1. Glossary web sites are at *http://hissa.nist.gov/dads/terms.html*, *http://www.instantweb.com* and *http://foldoc/whatis.techtarget.com*.

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.

cache-line fill Filling a cache line by accessing main memory as the result of a cache miss for one or more bytes not currently in any cache line. A cache-line fill takes ~30 times longer than a cache hit.

cardinal number A basic or primary value. Examples of cardinal numbers are 1, 7, 9, and 123. *Contrast with ordinal number.*

count In the Judy technology, the number of indexes between any two indexes, inclusive of the two indexes.

D

data structure An organization of information, usually in memory, for better algorithm efficiency, such as queue, stack, linked list, heap, dictionary, and tree. It may include redundant information, such as length of the list or number of nodes in a subtree. (NIST) *See also abstract data type.*

density A population divided by its expanse.

digital tree A tree in which the key is decoded one digit or byte at a time.

dynamic array An array whose size may change over time. Items are not only added or removed, but memory used changes, too. (NIST)

E

expanse The range of possible indexes 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}.

H

hash function A function which maps keys to integers, hopefully to get a more even distribution on a smaller set of values. (NIST)

hash table A dictionary in which keys are mapped to array positions by a hash function. Having more than one key map to the same position is called a collision. There are many ways to resolve collisions, but they may be divided into open addressing, in which all elements are kept within the table, and chaining, in which external data structures are used. (NIST)

hybrid data structures Using different abstract data types (ADTs) at different levels in a tree 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-line fills and the least amount of memory.

I

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.

J

Judy array An ADT that acts much like an ordinary array, but which appears unbounded (except by the available RAM and word size of the machine) and is allocated by the first store/insert into the array. Judy indexes can be inserted, retrieved, deleted, and searched in sorted order.

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

L

leaf node In tree structures, a node that has no nodes under it.

level compression Widening the nodes of a tree (branches of a Judy digital tree) in order to have fewer indirections (address calculations) and possible cache-line fills to access data.

linear list A simple, packed sequence of data items that are usually the same size and in some order (such as sorted numerically). For example, a linear list could be a list of valid indexes of one word each.

N

node (1) A unit of reference in a data structure. Also called a vertex in graphs and trees. (2) A collection of information which must be kept at a single memory location. (NIST)

O

O notation *See big-O notation.*

ordinal The sequence in which something is in relation to others of its kind. Examples of ordinal numbers are first, third, 11th, and 123rd. *Contrast with cardinal numbers.*

P

population The number of indexes that have been assigned or used.

S

scalability (1) The ability of a computer application or product (hardware or software) to continue to function well as it (or its context) is changed in size or volume in order to meet a user

need. Typically, the rescaling is to a larger size or volume. (2) The ability not only to function well in the rescaled situation, but to actually take full advantage of it.

skip list A probabilistic alternative to balanced trees that was invented by William Pugh in 1987. Parallel lists at higher levels skip geometrically more items. Searching begins at the highest level to quickly get to the right part of the list, then uses progressively lower level lists. With enough levels, searching is *O(log n)*.

sparse array An array in which some of the indexes may be invalid. A sparse data set may be represented by any of a large variety of data structures; an array is just one possibility. If the range of the data set is large and the valid indexes (keys) are sparse, a simple array is wasteful of memory although fast to access.

sort To arrange items in a predetermined order. There are dozens of sort algorithms, the choice of which depends on factors such as the number of items relative to working memory, knowledge of the orderliness of the items or the range of the keys, the cost of comparing keys versus the cost of moving items, etc. (NIST)

T

tree 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.

trie *See digital tree.*

U

ulong_t In C programming, an integer type that is set to *long* (maximum word length as defined by the machine) and to *unsigned* (a positive integer from 0 – *n*). On a 32-bit machine, *ulong_t* is 0 to 2^{32}-1. On a 64-bit machine, *ulong_t* is 0 to 2^{64}- 1.

unbounded array An array where the maximum size is not arbitrarily limited (or bounded) at less than the maximum word size available to the machine. For example, on a 32-bit machine an unbounded array could be from 0 to 2^{32}, while on a 64-bit machine an unbounded array could be from 0 to 2^{64}.

W

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.

Numerics

10-way digital tree, 15

A

advantages (of Judy), 15
algorithms
  counting, 27
  search and retrieval, 17
  sorting, 26
arrays
  binary, 28
  dynamic, 11, 16, 25
  in Judy, 11, 25, 59
  in memory, 62
  Judy1, 28
  JudyL, 33
  JudySL, 44
  multi-dimensional, 44, 49
  sparse, 11
  unbounded, 9, 11

B

big O notation, 10
Boolean values, 28

C

caching, 61
  cache line fills, 17, 61
  CPU, 61
  curve, 61
  defined, 61
  in Judy, 62
counting, 10, 11, 17, 27
CPU caching, 61

D

data structures, 10, 15
  digital trees, 15, 59
  in memory, 17
definition of Judy, 9
digital trees, 15, 59
  nodes in, 17

E

examples
  digital trees, 15
  Judy1 array, 29
  JudyL array, 34
  JudySL array, 45
  multi-dimensional arrays, 49
  zip code, 15

H

history (of Judy), 12, 59
HP-UX, 12, 49
hybrid digital tree, 16

I

indexes (Judy tree), 17

J

Judy header file, 25
Judy technology
  advantages, 15
  array types, 11
  defined, 9
  history, 12
  libraries, 12
  memory requirements, 9
  name, 60
  overview, 9
  performance, 10, 11, 12
  scalability, 10, 11
  summary of functions, 25
  unbounded arrays, 9, 11
  when to use, 11
Judy tree structure
  count, 17
  indexes, 17
  pointers, 17
  type, 17
Judy1 array
  counting, 27
  defined, 11
  examples, 29
  table of functions, 28
  using, 28, 44
JudyL array
  counting, 27
  defined, 11
  examples, 34
  table of functions, 33
  using, 33
JudySL array
  defined, 11
  examples, 45
  sorting, 26

table of functions, 44

L

libraries (in Judy), 12
long-word values, 33

M

memory management, 61
memory requirements, 9
multi-dimensional arrays, 44, 49

N

nodes (Judy tree), 17

O

overview of Judy, 9

P

performance, 10, 11, 12
pointers (Judy tree), 17

S

scalability, 10, 11
search and retrieval, 11, 15, 17
sorting, 10, 11, 26
  example of, 45
speed, 10
string values, 44
strings, sorting, 26
summary of functions (Judy), 25

T

trie *See* digital trees
type (Judy tree), 17

U

unbounded arrays, 9, 11

V

values
  Boolean, 28
  long-word, 33
  string, 44

Z

zip code example, 15