T-treap and Cache Performance of Indexing Data Structures

Joshua P. MacDonald and Ben Y. Zhao

Abstract

As memory becomes cheaper and larger in capacity, more and more databases are being stored entirely in main memory. Furthermore, there is an increasing latency gap between on-chip caches and main memory. As a result, how main memory indexing data structures perform, especially with respect to local cache, is becoming an increasingly important factor in database performance.

One of these data structures is the treap, a binary tree using a Cartesian key pair to provide probabilistic self-balancing properties. The Cartesian keys in treaps have a great potential to be exploited for faster memory accesses. In this paper, we describe efforts to extend the treap to a multivalue node data structure called the T-treap, in an effort to improve the pointer to data ratio and overall performance.

We study several indexing data structures as well as our new data structure in a simulation environment. We derive some general results of the T-treap and also show that the relative performance of cache-conscious indexing structures is increasing with memory latency. In addition, we analyze the T-treap results, and propose optimizations to the T-treap. Finally, we show that top-down algorithms for maintaining these structures reduce the total instruction count, leading to a modest improvement in execution time over the corresponding bottom-up algorithms.

1. Introduction

The speed of computer processors is growing rapidly in comparison to the speed of DRAM chips. The processor clock cycle has been decreasing at a rate of roughly 70% every year, while the cycle time of common DRAM chips is decreasing by about 33% every 10 years [7]. These trends indicate that the cost of a cache miss, measured in processor clock cycles, is increasing at an exponential rate.

As memory becomes cheaper and larger in capacity, more and more databases are being stored entirely in main memory [1]. Furthermore, there is a large latency gap between on-chip caches and main memory. As a result, main memory databases using non cache-conscious data structures for indexing will suffer in performance due to the increasing cost of main memory access. This problem has already been studied for indexing in secondary storage where the cost of magnetic storage has been problematic for a long time [4]. The effects of caches on indexing performance are comparatively new, and have the following distinguishing characteristics:

- Cache blocks are much smaller than disk pages;
- Cache replacement policies are fixed and have limited associativity;
- Cache access times are much shorter than secondary storage, relative to the processor, so it is insufficient to simply equate a tree's depth with its access time as is done for secondary storage.

The treap is a binary tree that uses a Cartesian key pair to index data [13]. The naïve version of the treap uses a randomly generated second key to provide a reasonably well-balanced tree. For consistency, we will use Seidel's terminology and refer to the secondary key as the *priority*. The priority, sorted in heap order, can be exploited by read operations to migrate often-accessed data closer to the root, requiring fewer memory accesses. A main disadvantage of the treap is its high pointer to data ratio, which makes it perform relatively poorly on cached systems. One of the efforts of our work is to create a new data structure using the same Cartesian indexing mechanism while providing a better pointer to data ratio.

The idea of improving pointer to data ratios has been explored before, most notably in the T-tree data structure first presented in the context of main memory databases [10]. We introduce an initial design for a data structure called the T-treap, which combines the randomized secondary keys of the treap with the multivalued nodes of the T-tree. We hope to use the T-treap as a basis for experiments on isolating hotspots in data accesses and using the biasing of priorities to better exploit data locality within cache lines. We first test the balancing factor of the T-treap in comparison to a treap using Java prototypes, and then test the T-treap's operational latencies in the simulation environment.

We also study the main memory performance of a variety of existing techniques for maintaining balanced trees, including B^+-trees [4], AVL trees,

treaps [13], and two variants of the top-down deterministic skip list [11]. Experimental data is measured using both current hardware and the SimpleScalar tool set [2] for detailed simulation of a modern processor. Our results indicate that cache-conscious indexing data structures currently outperform their counterparts by approximately 48%. A factor of five (25) increase in memory latency increases the relative performance difference to 77% (84%).

Another aspect of balanced trees that is important to consider is their implementation difficulty. Munro et al. propose that deterministic skip lists using *top-down* operations are simpler to understand and implement than many of the alternatives [11]. Our simulation results confirm this, showing that top-down skip lists enjoy increased performance and reduction in instruction count relative to other tested data structures.

Top-down operations perform rebalancing in a single, downward pass through the tree, while *bottom-up* operations search on the way down and rebalance on the way back up, requiring the use of a stack or parent pointers [6]. Top-down operations also lead to an average reduction of 31% in total instruction count for our experiments. Because of this reduction a cache-conscious version of the basic skip list currently outperforms the corresponding B^+-tree by 12%, though this advantage will decrease with time as memory access time begins to dominate overall performance. Still, if the performance of top-down and bottom-up algorithms are comparable then the simplest implementation should be chosen.

2. T-Treap Design

The goals of the T-treap is to combine the properties of self-balancing using randomization, and that of clustered values in a single node, for better memory utilization. Associated with each T-treap is a node capacity range, which defines the minimum and maximum capacities inside its nodes. Leaf nodes are permitted to have less than the minimum number of nodes, and provides flexibility for maintaining optimal capacity in internal nodes. The T-treap is similar in operation to other data structures, providing indexing and retrieval of a key value pair.

Each T-treap node contains a cluster of values with adjacent keys, along with node pointers to adjacent nodes. Within each node, the individual priorities of the values are kept, along with a directional bit, and a node priority that is used for rotation operations between nodes. Figure 1 shows the internal representation of a T-treap node, along with a high level representation of a T-treap.

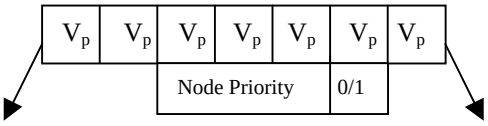


Figure 1a: T-treap Node

While the two key properties of the T-treap seem orthogonal on inspection, the combination of clustered nodes and priority-based balancing

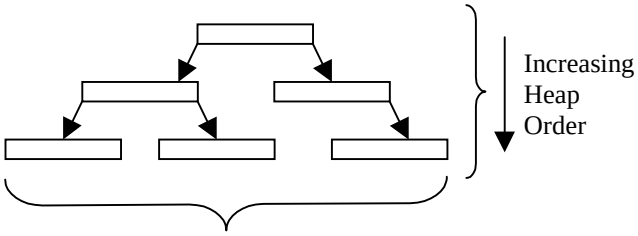


Figure 1b: T-treap

produces additional complexity in the insertion and deletion algorithms. This complexity can be costly in the worst case scenario, and is further discussed below.

2.1 Insertion/Deletion/Lookups

Inserts and deletes into the T-treap occur on the granularity of single values. The insertion algorithm is similar to that of a binary tree. First, each new value is assigned a priority, which is a randomized integer that corresponds to the second key in a Cartesian key pair. Then starting with the root, the value's key is compared to the minimum and maximum keys in each visited node. The insertion operation traverses the T-treap, stopping when the key falls inside the range of a node, or when the node has no children in the direction to be traversed. The key, value and priority are then added to the current node, and the priority of the node is adjusted for the new insert. Delete operations mirror insertions in the tree traversal. After a value has been removed, the node priority is again adjusted to reflect the priority of the current node occupants.

One of the key invariants of the T-treap node is that the number of values in a node always falls between the minimum and maximum node capacity defined for the T-treap. During insert and delete operations, values can "overflow" to an adjacent leaf node, or in the case of an "underflow," be borrowed from an adjacent leaf node. Since overflow and underflow are the only ways that a node can be created or destroyed, the way in which they are handled has an impact on the balancing nature of the overall T-treap.

There are several choices for choosing the direction of overflowing data. In general, we would want a function that balances overflows between left and right directions in such a way that the self-balancing properties from the treap are preserved. We do so by consulting a directional bit, which is flipped after each overflow. In implementations where data is statically allocated within the node, the cost of data copying is a factor, and the algorithm can overflow data in the direction which minimizes copies. If such cases, the directional bit is set to the direction chosen.

When a delete operation causes a node to underflow in capacity, we proceed to migrate values from an adjacent leaf node. The direction of underflow can be determined by flipping the directional bit. This allows the underflow to occur in the same direction as the last overflow, ensuring overall balance in the data movement.

The lookup function is simple compared to inserts and deletes. We simply traverse the T-treap by checking key boundaries at each node, until we find an inclusive node. The values in the node are then searched using binary search. Furthermore, a read operation in a biased version of the T-treap can be set to modify the priority of the value, possibly causing an entire node to be rotated from its position in the tree.

2.2 Priority management / Balancing

Due to the fact that the overall balance of the T-treap depends on the priorities, how those values are generated plays a part in determining how balanced a T-treap will be.

There are several algorithms one can choose to generate a node priority from. The three with the minimal storage overhead are: MIN, MAX, and AVG. For the MIN and MAX algorithms, reprioritizing after an insert or delete takes one comparison, except when the delete removes the value with the node priority, which then requires a full scan of individual priorities in a delete. For the AVG algorithm, the priority can be calculated with no comparison operations.

After node priorities are adjusted, the T-treap is checked for compliance with the heap order property[1]. Nodes are rotated if they violate the heap property.

One special case occurs when dealing with rotating leaf nodes. A leaf node that has capacity less than the defined node minimum is allowed to violate the heap order property and ignored during rotations. This is because only leaf nodes can adjust their minimum and maximum key values, and allowing a underflowing leaf to become an internal node guarantees it will not increase in capacity, and therefore extend the tree depth needlessly.

2.3 Areas for Optimization

There are several tunable parameters in the design of the T-treap. Due to the time constraints on this project, we were unable to fully explore the effects of each parameter on the overall performance of the structure. We briefly discuss here a few mechanisms that should be further explored:

- **Algorithms for read biasing**

There are several possible algorithms for adjusting the priority after a read access. A simple algorithm suggested by Seidel [13] is

$$\min(Max\,\mathrm{Pr}\,iority * Rand(\,),\mathrm{Pr}\,iority)$$

This places no constraint on the amount of change in priority, however, and could cause large sets of rotations after single read operations. Some more reasonable algorithms could be:

$$\mathrm{Pr}\,iority - X(\mathrm{Pr}\,iority_{parent} - \mathrm{Pr}\,iority)$$

or:

$$\mathrm{Pr}\,iority * (1 - Rand(\,))$$

- **Decoupling of node priority**

The current design calls for keeping a priority for each key-value pair. One potential variant could have a T-treap node where only the aggregate

[1] Our implementation called for lesser priority values at the top of the T-treap, and each node having a smaller priority than its children.

priority is stored, and all biasing operations modify the node priority directly. There is an obvious benefit in eliminating memory overhead which can be used for storing more values per node or cache line. Furthermore, modifying the node priority directly fixes the granularity of locality to the node size, which could allow the structure to more intuitively exploit spatial locality.

3. Skip Lists

Skip lists were originally introduced by Pugh as a randomized alternative to search trees [12]. Conceptually, a skip list represents the set of elements as a number of ordered chains equal to its height. All elements appear in the bottom chain, and each level above that contains a fraction *p* of the elements in the chain below it. Operations defined on this structure, illustrated in Figure 1, have expected logarithmic time.

A perfectly balanced skip list of order *k* requires that every k^{th} node of height at least *h* is of height at least *h*+1 [11]. In the probabilistic case, *k*=1/*p*. Two elements in a given chain are said to "skip" *k*-1 elements in the chain below them. The skipped elements form a gap. In order to give the skip list deterministic logarithmic time bounds, the perfect balance condition is relaxed to allow a variable width gap. A 1-2 skip list, for example, allows a gap of 1 or 2 and has a corresponding 2-3 tree [8].

Searching in a skip list begins at the head of the highest level chain, and at each point the decision is made to move either down or right. At each level, the search proceeds right as long as the next key at the current level is less than the search key, then it moves down and repeats the process. There are never more than the gap size number of rightward motions before the search moves down.

The correspondence between search trees and skip lists is easy to generalize. The elements forming a gap of size *k*-1 correspond to a multiway tree node with *k*-1 keys and *k* children. Top-down algorithms for insertion and deletion in 2-3-4 trees exist [6], and they are used as a basis for the top-down 1-2-3 skip list [11].

3.1 Top-down 1-2-3 Skip List

In its simplest form, the top-down 1-2-3 skip list uses a linked representation with fixed-size nodes, as illustrated in Figure 4. Several modifications have been made to the array representation. First, instead of using the next element's key for comparison, each key is stored at the node where it is actually used. Second, a new key named *max* that is greater than any key in the set is used to terminate each chain. The last key of any gap is always greater than the search key, and is called the *high key*. Though a comparison against the high key is redundant computation, it simplifies the code and has other uses as well. Additionally, there is a dummy **head** node and two sentinel nodes **bottom** and **tail**. Having made these modifications the representation is now somewhere between a skip list and a binary tree.

C code for search and lookup are given by Munro et al. using the following type definition:

```
typedef struct _Node Node;
struct _Node{
int k;
Node *r, *d;
};
Search is written as:
Node* search(int v){
Node* x = head;
bottom->k = v;
while(v != x->k)
      x = (v < x->k)?x->d:x->r;
return x;
}
```

The top-down insertion routine uses a precondition before descending into a gap ensuring that it can perform any necessary splits at the next level down. A full gap (of size three, in this case) must be split (into two gaps of size one) in order to meet this condition. In order to test whether the node x's gap is full, it uses the expression (x->k == x->d->r- >r->r->k). The function returns 1 on a successful insertion and 0 on failure:

```
int insert(int v){
node *t, *x = head;
bottom->key = v;
for(; x != bottom; x = x->d){
      while(v > x->k)
            x = x->r;
      if(x->d == bottom &&
             v == x->key)
            return 0;
      if(x->d == bottom ||
             x->k == x->d->r->r-
>r->k){
            t = new(Node);
            t->r = x->r;
```

```
            t->d = x->d->r->r;
            x->r = t;
            t->k = x->k;
            x->k = x->d->r->k;
      }
}
if(head->r != tail){
      t = new(Node);
      t->d = head;
      t->r = tail;
      t->k = max;
      head = t;
}
return 1;
}
```

Deletion uses the opposite precondition; it must ensure that it can perform any necessary concatenations at the next level down. A gap at minimum capacity (which is tested with the expression (x- >k == x->d->r->k)) is dealt with in four cases. An adjacent gap must be found either to the right or the left, these two cases are treated differently. If the adjacent gap is also at minimum capacity then they are concatenated to form a single gap of size three. If the adjacent gap is not at minimum capacity then an element is borrowed from that gap. When a match is found at the bottom of the skip list the node is removed from the list in one of two ways, depending on whether it is the first element of a gap or not. Finally, the first interior node with a key that matches the deleted key must be revisited and a second downward pass replaces the deleted key with the next greatest value in the set. Code for delete is supplied in Appendix A.

Note that there is no recursion, nor the use of stack or parent pointers in this code, therefore it is easier to understand. Bottom-up algorithms, on the other hand, require one of these mechanisms to keep track of nodes that have been visited, and this generally complicates program structure.

3.2 Higher order top-down skip lists

Higher order skip lists are constructed by allowing larger gaps. In general a skip list that allows a gap of (m/2)-1, ..., (m-1) corresponds to a B-tree of order m. To obtain the storage efficiency of a B-tree, however, it is necessary to place all the keys and down pointers in a gap into the same "page". Since all the keys in a skip list are stored in the bottom chain, the corresponding B-tree is actually a B^+- tree. Using this representation, the paged 1-2-3 skip list would appear as in Figure 5. The only structural differences between the paged skip list and the B^+- tree are:

- The high key occupies one extra key per skip list node
- Right links are used on internal and external nodes, whereas the B^+-tree has them only on external nodes.

Other than these, the only thing that distinguishes the two data structures is that B-trees are bottom-up while the skip list is top-down.

Rewriting the top-down algorithms given above to use the paged representation is surprisingly simple. Deletion is simplified by using an explicit count instead of key comparisons to detect overflow and underflow, making it is no longer necessary to revisit internal nodes after a key is deleted. Deleted keys are allowed to remain in the skip list just as in the B^+-tree. The borrowing operations are replaced with redistribution operations, to reduce the chances of future underflow.

The presence of right links in the paged skip list is a matter of convenience only, in order to support sequential access. Internal nodes have the same format as external nodes, so their right links are maintained as well, but they are not actually used by any of the dictionary operations.

3.3 Other concerns and applications

Concurrency is very important in database applications. Top-down algorithms are inherently concurrent, since they only require locking a fixed-size context around the current node during any operation. A related issue when secondary storage is involved is the efficient handling of blocked writers (allowing readers to pass through). It is interesting to note that the two structural extensions used in Lehman and Yao's B^{link}-tree for efficient concurrency in B^*-trees are the same two used in the skip list–a high key and right links on internal nodes [9]. Using their scheme, no read locks are required because the high key and right pointer allow a reader to find the adjacent sibling in case a concurrent modification has occurred. Their techniques should apply to top-down algorithms in a similar fashion.

The skip list thus presented can serve as a priority queue without modification since the

minimum key can be easily found in constant time and deleted in logarithmic time [3].

4. Experimental Results

In the analysis phase of our project, we provide some initial results of T-treap balancing behavior, in addition to a large set of detailed simulations on several existing indexing data structures.

4.1 T-treap Depth Measurements

The T-treap was first prototyped in Java. Several tests were performed on varying data sizes between the T-treap and the conventional treap. The main metric under question is the self-balancing nature of the T-treap.

The experiment involved inserting a large set (>380,000) of uniquely keyed data items into the structures. We compared a treap and 3 different versions of the T-treap, each using a different priority aggregation algorithm, as discussed in Section 2.2.

If we assume that multivalue nodes do not adversely affect the probabilistic self-balancing nature of the treap, we can predict the following decrease in average depth:

$$Avg\Delta D \approx Log_2(N) - Log_2\left(N/X\right)$$

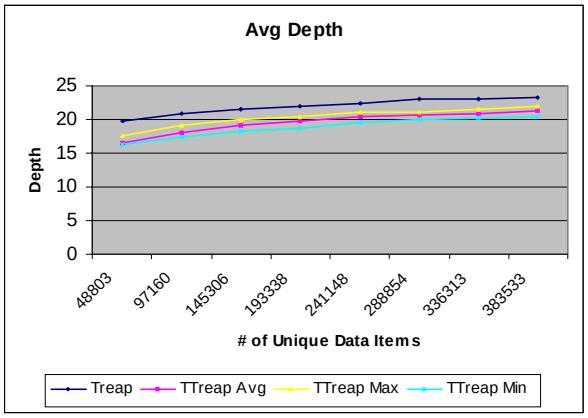


Figure 2

where:

N = number of total values

X = number of values per T-treap node.

In Figure 2, the results show the expected decrease in average depth due to the clustering of values per node. It supports the hypothesis that the self-balancing mechanism of the priorities was not adversely affected by the aggregation algorithms. More theoretical work is necessary to prove the hypothesis.

The data also shows that the MIN priority algorithm works best among the three choices examined. Other experiments on insertion time and max depth confirm this. We suspect this result is directly related of our choice of heap order $P_{parent} < P_{children}$. Similarly, in a heap order where $P_{parent} > P_{children}$, one would expect the MAX algorithm to perform the best.

4.2 Experiments in C[2]

For this portion of analysis, the linked skip list, paged skip list, B+-tree, AVL tree, treap and T-treap were each programmed in C. Two variations on the B^+-tree were tested, one using binary search within the node and one using sequential search. The paged skip list and B^+-tree allow their node size to be varied at compile-time in order to study the effect of tuning the data structure to the cache-block size. For plotting results as a function of node size, the non-variable-size structures are reported for every node size tested and appear as a flat line. Only node sizes that are a power of two were tested. For simplicity, it was assumed that each key, datum, and pointer occupies one 32-bit word. A cache-block of size $2m$ words can contain a B+-tree of order m. Due to the extra space occupied by the high key in the skip list, its branching factor is always one less than the corresponding B^+- tree. For example, a 32-word cache block has m=16 and a node size of 128 bytes. The version of the T-treap tested used the minimum priority algorithm.

An experiment was constructed using the UC Berkeley Home IP HTTP traces [5]. These traces consist of a total of 9,244,728 client HTTP requests taken over a period of 18 days. A key was generated from each request as follows. The high 16-bits of each key were taken as a hash of the server's IP address, and the low 16-bits were taken as a hash of the request string. There are 2,663,855 unique keys using this method. A tree is built by inserting every unique key into the tree in random order, then searching for each key in the order it appears in the trace, then deleting

[2] Unfortunately, analytical results of the C version of the T-treap are limited due to time constraints.

each key in random order. The intent of this setup was to preserve any spatial and temporal locality in the trace for the search portion of the workload. Keys are clustered spatially by the server each request was bound for, and have whatever temporal locality that was present in the original traces.

The experiments were run on an Intel Pentium II Xeon processor running at 450 MHz and simulated on the SimpleScalar architecture. Since the SimpleScalar out-of-order simulator is quite slow, a short 4-hour fragment of the traces were used for simulation. The shortened data set contains 95,769 requests and 61,308 unique keys.

Figure 6 shows the results of running the experiment on three processor configurations: the Intel processor and the SimpleScalar processor with memory latencies of 18 (the default value) and 100 cycles. The SimpleScalar processor was configured with a 32K split L1 I- and D-cache with 64 byte blocks and a 256K unified L2 cache with 128 byte blocks. The AVL tree is the best performing fixed- node-size data structure, and the paged skip list is the best performing variable-node-size data structure. Moreover, the gap between the cache-conscious data structures and their counterparts is growing, as shown by the 100 cycle memory latency configuration. The paged skip list outperforms the sequential-search B^+-tree by 12% on the Intel processor. It is clear that while the T-treap enjoys some performance gains relative to the treap, it does not exhibit the same cache-conscious behavior as the B-tree or the paged skip list. The performance difference between the AVL tree and the T-treap and treaps is due to the randomized balancing mechanism which on average provides a tree with

Table 1 lists various quantities describing each data structure for a node size of 128 bytes on the SimpleScalar processor. These values depend only on the specific algorithms used and not the memory latency. The cache-conscious data structures not only improve access time, but they also improve storage efficiency. The storage ratio is given as the number of words of overhead per key/datum pair in the index. The two skip list implementations require 31% fewer instructions, on average, than the bottom-up methods.

Table 2 lists the cache performance of each data structure. The three cache-conscious data structures produce significantly fewer cache misses in both levels of the cache, by a factor of five or more in the L2 cache. These values vary only slightly with memory latency, since latencies can effect the out- of-order pipeline. They were measured for the 18 cycle memory latency configuration.

Table 3 shows the execution time for each data structure with memory latencies of 18, 100, and 500 cycles and the corresponding cycles per instruction (CPI). The cache-conscious data structures are result in a 48% improvement in total cycles, on average, compared to their counterparts, with a memory latency of 18 cycles. Increasing the memory latency to 100 cycles (500 cycles) leads to an improvement of 77% (84%). These results indicate that the performance of the cache-conscious data structures is steadily increasing relative to the others.

Figure 7 shows the results of carrying the experiment out for large node sizes on the full data set with the binary-search B^+-tree, the paged skip list, and the AVL tree. Large nodes are inefficient for two reasons. First, their cache behavior is identical to a binary search tree, so the binary B^+-tree approaches the AVL tree for searching with large node sizes. Second, the cost of update operations becomes dominated by the cost of memory-to-memory copies for updating individual nodes. These copies also disrupt the cache. Node sizes larger than 1024 bytes (m=128) have worse performance on insert and delete than the AVL tree due to this effect. The cross-over point at which binary search begins to outperform sequential search occurs at a node size of 2048 bytes (m=256).

5. Conclusion

Cache performance is a growing bottleneck for dictionary data structures. The cache-conscious data structures measured in this study already outperform their counterparts by 48%. This value will increase steadily over the next few years to 77% or more when cache misses begin to cost hundreds of processor clock cycles.

Data structures that are optimized for reducing the number of accesses to secondary storage will be effected by this trend as well since they have the poor cache-performance of a binary search.

There is currently no work that we know of that adopts a two-level strategy for reducing both cache and disk accesses. Optimizing for both levels of the storage hierarchy will increase in importance as the cost of a cache miss increases.

Our initial design and implementation of the T-treap data structure shows improvement over the naive treap. The improvement, however, is proportional to the increase in cache size and the corresponding values/node. In a cached environment, the T-treap still performs relatively poorly, due to its binary branching factor. In an environment such as Java, however, where there is no control over data placement in memory, the biased versions of the T-treap and treap has potential to perform quite well. Given the number of possible optimizations, a closer look is warranted.

The top-down algorithms for maintaining skip lists are easier to understand and implement than the corresponding bottom-up algorithms for B-trees, and they reduce the instruction count as well. Given that their performance is similar in main memory, the paged skip list seems like a good alternative to B- trees.

Bibliography

[1] Phil Bernstein, et al. The Asilomar report on database research. *ACM Sigmod Record*, 27(4), 1998

[2] Doug Burger and Todd M. Austin. The SimpleScalar Tool Set, Version 2.0. University of Wisconsin-Madison Computer Science Technical Report #1342, June 1997.

[3] Seonghun Cho and Sartaj Sahni. Weight-Biased Leftist Trees and Modified Skip Lists. *Journal of Experimetnal Algorithmics*, 3(2), 1998.

[4] Douglas Comer. The Ubiquitous B-Tree. *ACM Computing Surveys*, 11(2), pp. 121-137, June 1979.

[5] Steven D. Gribble. UC Berkeley Home IP HTTP Traces. July 1997. Available at http://www.acm.org/sigcomm/ITA/.

[6] Leo J. Guibas and Robert Sedgewick. A Dichromatic Framework for Balanced Trees. *19th Annual Symposium on Foundations of Computer Science*, pp. 8-21, IEEE Computer Society Press, October 1978.

[7] J. L. Hennessy and D. A. Patterson. Computer Architecture: A Quantitative Approach, Morgan Kaufmann, 1996.

[8] Donald E. Knuth. Sorting and Searching, The Art of Computer Programming, Vol. 3, p. xiv + 780, Addison-Wesley, 1998.

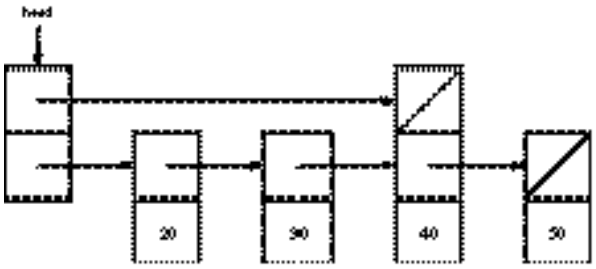
[9] Philip L. Lehman and S. Bing Yao. Efficient Locking for Concurrent Operations on B- Trees. *ACM Transactions on Database Systems*, 6(4), pp. 650-570, December 1981.

[10] Tobin J. Lehman and Michael J. Carey. Query Processing in Main Memory Database Management Systems. *Proceedings of the Conference on Management of Data (SIGMOD 1986).*

[11] J. Ian Munro, Thomas Papadakis and Robert Sedgewick. Deterministic Skip Lists. *Proceedings of the 3rd Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA '92), pp. 367-375, SIAM, January 1992.

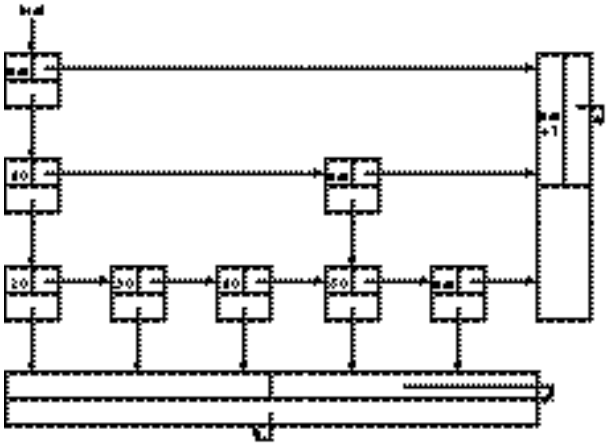
[12] William Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Proc. 1st Work. Algorithms and Data Structures*, WADS, Lecture Notes in Computer Science, LNCS, Vol. 382, pp. 437-449, Springer-Verlag, 17-19 August 1989.

[13] R. Seidel and C. R. Aragon. Randomized Search Trees. *Algorithmica*, Vol. 16, Number 4/5, pp. 464-497, October 1996



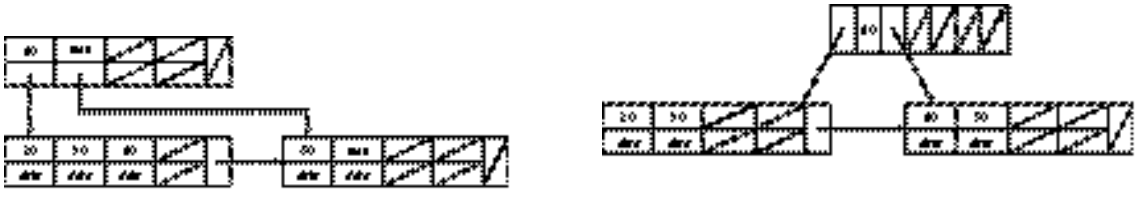
A skip list is comprised of a hierarchy of chains. Every key in the set resides in the bottom chain, and each level above that contains a fraction *p* of the elements in the chain below it.

Figure 3: A skip list with the standard array implementation



The linked skip list representation moves keys into the node where they are actually used and adds a high key.

Figure 4: A skip list with the linked list representation



(a) (b)

The skip list (a) resembles a B^+-tree (b) except that it has a high key and right links at all nodes, not just in the external nodes.

Figure 5: A skip list with the paged representation, and corresponding B^+-tree

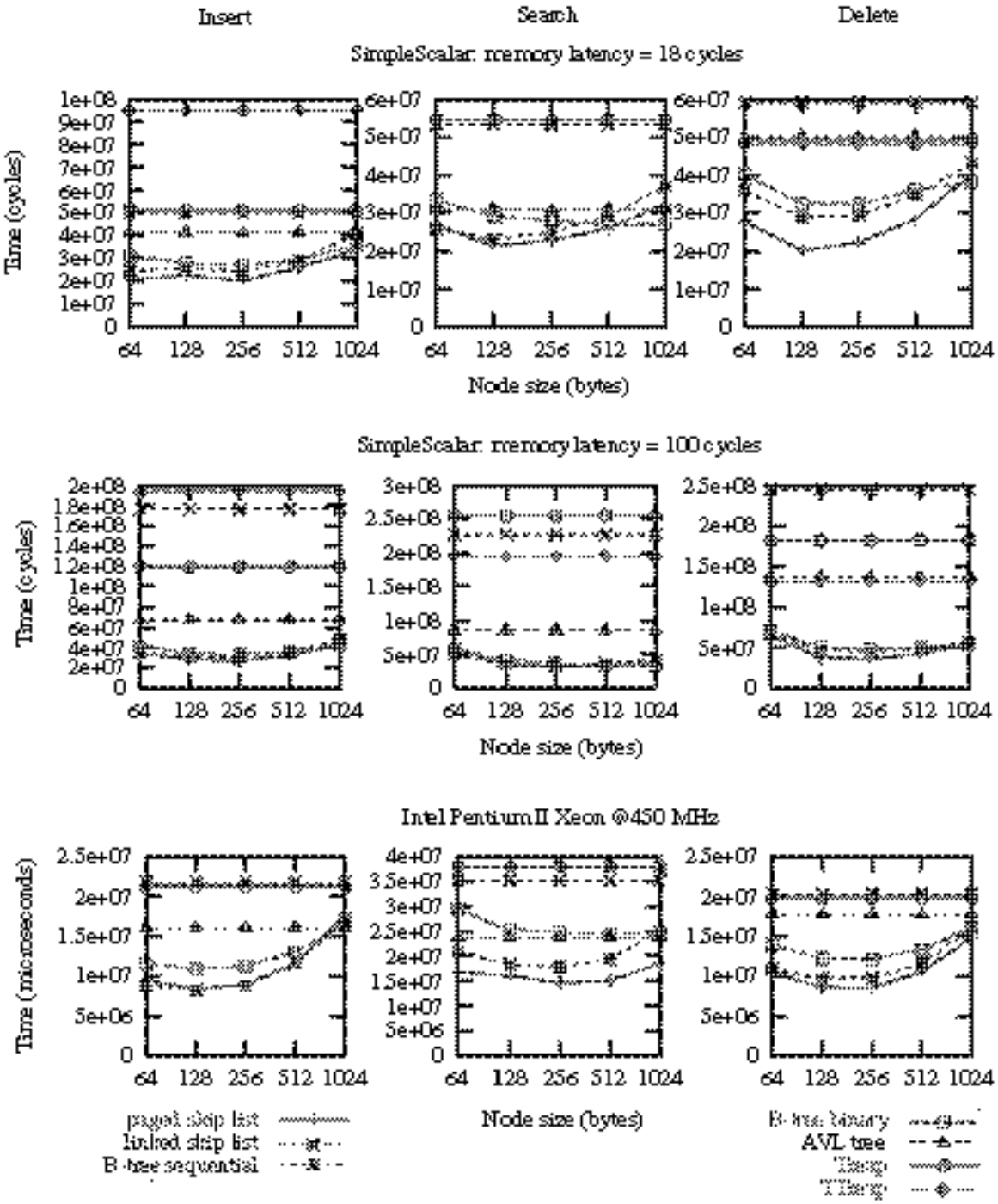
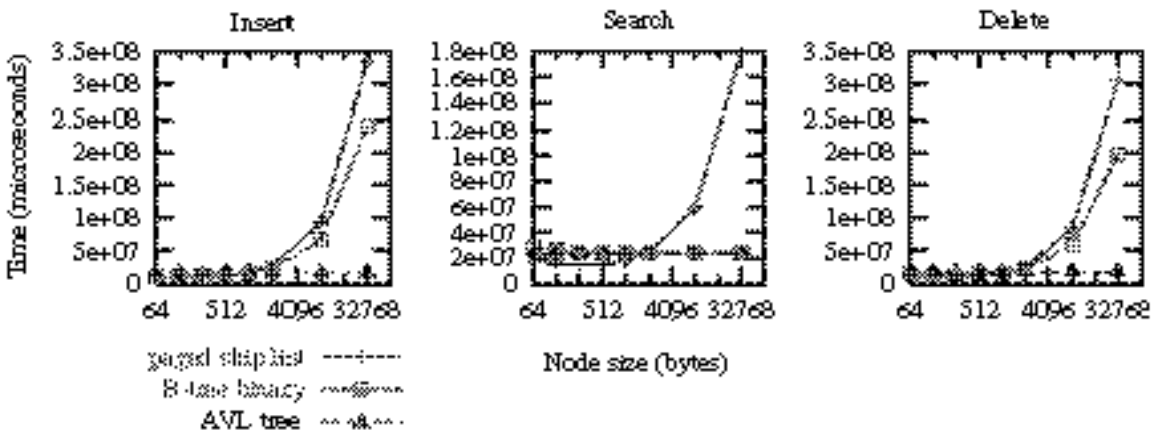


Figure 6: Performance comparison for three processor configurations



These graphs show the time taken by the dictionary operations for the full data set on the Intel processor. Searching in large nodes has the same performance as the (binary) AVL tree. For large nodes the cost of insertion and deleteion becomes dominated by the cost of memory-to-memory copies when updating individual nodes (which also disrupts the cache). They show the node size at which binary-search begins to outperform sequential search. This indicates that choosing disk-sized pages without regard for cache-performance incurs a serious (and growing) performance penalty in main memory.

Figure 7: Large node sizes

Tree	Height	Storage Ratio	Inst. Count	Loads	Stores	Branches
AVL	19	2.50	141M	38.3M (27%)	21.8M (15%)	34.3M (24%)
Treap	0	2.50	124M	34.4M (28%)	19.2M (15%)	29.1M (23%)
Skip List (P)	4	1.69	88.3M	19.6M (22%)	6.63M (7.5%)	17.7M (20%)
Skip List (L)	13	3.14	87.9M	30.9M (35%)	5.91M (6.7%)	23.1M (26%)
B+Tree (S)	4	1.66	125M	28.1M (22%)	14.2M (11%)	26.0M (21%)
B+Tree (B)	4	1.66	123M	26.7M (22%)	14.2M (12%)	25.5M (21%)

Table 1: Various quantities for each data structure (node size = 128 bytes)

Tree	L1			L2		
	Accesses	Misses	Writeback	Accesses	Misses	Writeback
AVL	62.5M	2.04M (3.3%)	1.12M (1.8%)	3.17M	910K (29%)	521K (16%)
Treap	56.8M	3.30M (5.8%)	1.47M (2.6%)	4.77M	1.48M (31%)	706K (15%)
Skip List (P)	27.5M	880K (3.2%)	364K (1.3%)	1.99M	220K (11%)	153K (7.7%)
Skip List (L)	42.7M	4.10M (9.6%)	507K (1.2%)	4.61M	2.02M (44%)	367K (8.0%)
B+Tree (S)	43.9M	888K (2.0%)	385K (0.88%)	1.96M	223K (11%)	156K (8.0%)
B+Tree (B)	41.1M	882K (2.1%)	383K (0.93%)	1.51M	221K (15%)	156K (10%)

Table 2: Cache performance for each data structure (node size = 128 bytes)

Tree	Inst. Count	18 cycles		100 cycles		500 cycles	
		Cycles	CPI	Cycles	CPI	Cycles	CPI
AVL	141M	126M	0.89	305M (+142%)	2.16 (+142%)	767M (+508%)	5.42 (+508%)
Treap	124M	155M	1.25	556M (+258%)	4.48 (+258%)	1.48G (+851%)	11.9 (+851%)
Skip List (P)	88.3M	64.0M	0.72	102M (+59%)	1.15 (+59%)	201M (+215%)	2.28 (+215%)
Skip List (L)	87.9M	162M	1.85	650M (+300%)	7.40 (+300%)	1.82G (+1021%)	20.7 (+1021%)
B+Tree (S)	125M	78.0M	0.62	115M (+48%)	0.92 (+48%)	217M (+179%)	1.74 (+179%)
B+Tree (B)	123M	89.8M	0.73	127M (+41%)	1.03 (+41%)	229M (+155%)	1.87 (+156%)

Table 3: Effect of increasing memory latency (node size = 128 bytes)

Appendix A: Code for delete

```
int delete(int k){
Node *l, *t, *t2, *r, *x;
Node *f = NULL;
int d = 0;
  for(l = NULL, x = head->d;
    x != bottom;
    l = NULL, x = x->d){
  for (; k > x->k;
        l = x, x = x->r) { }
  if(x->d == bottom){
    if(k == x->k){
      if(l == NULL){
        r = x->r;
        x->k = r->k;
        x->r = r->r;
        free(r);
      } else {
        l->r = x->r;
        free(x);
        x = l;
      }
      if(f)
        fixup(f,k,x->k);
      d = 1;
    }
  } else {
    if(k == x->k && f == NULL)
      f = x;
    if(x->d->r->k == x->k){
      if(l == NULL){
        r = x->r;
        if(r->d->r->k == r->k){
          x->k = r->k;
          x->r = r->r;
          free(r);
        } else {
          x->k = r->d->k;
          r->d = r->d->r;
        }
      } else {
        if (l->d->r->k == l->k){
          l->k = x->k;
          l->r = x->r;
          free(x);
        } else {
          t = l->d;
          while(t->r != x->d){
            t2 = t;
            t = t->r;
          }
          l->k = t2->k;
          x->d = t;
        }
      }
    }
  }
}
if(head->d->r == tail){
    t = head;
  head = head->d;
  free(t);
}
return d;
}
void fixup(Node *f,
          int oldk, int newk){
Node *x;
for(x = head->d;
      x != bottom; x = x->d){
  while(oldk > x->k)
    x = x->r;
  if(x->k == oldk)
    x->k = newk;
}
```