Branchless Search Programs*

Amr Elmasry[1] and Jyrki Katajainen[2]

[1] Department of Computer Engineering and Systems, Alexandria University
Alexandria 21544, Egypt
[2] Department of Computer Science, University of Copenhagen
Universitetsparken 5, 2100 Copenhagen East, Denmark

Abstract. It was reported in a study by Brodal and Moruz that, due to branch mispredictions, skewed search trees may perform better than perfectly balanced search trees. In this paper we take the search procedures under microscopic examination, and show that perfectly balanced search trees—when programmed carefully—are better than skewed search trees. As in the previous study, we only focus on the static case. We demonstrate that, by decoupling element comparisons from conditional branches and by writing branchless code in general, harmful effects caused by branch mispredictions can be avoided. Being able to store perfectly balanced search trees implicitly, such trees get a further advantage over skewed search trees following an improved cache behaviour.

1 Introduction

In traditional algorithm analysis each instruction is assumed to take a constant amount of time. In real computers pipelining and caching are omnipresent, so the unit-cost assumption may not always be valid. In this paper we study the impact of hardware effects on the efficiency of search programs for search trees. The outcome of the element comparisons performed in searches is often hard to predict. Also, sequential access is to be avoided in order to support searches in sublinear time. Both of these aspects make search algorithms interesting subjects of study.

We focus on the following hardware phenomena (for more details, see [15]):

Branch misprediction. In a pipelined processor, instructions are executed in parallel in a pipelined fashion. Conditional branches are problematic since the next instruction may not be known when its execution should be started. By maintaining a table of the previous choices, a prediction can be made. But, if this prediction is wrong, the partially processed instructions are discarded and correct instructions are fed into the pipeline.

Cache miss. In a hierarchical memory, the data is transferred in blocks between different memory levels. We are specifically interested in what happens between the last-level cache and main memory. Block reads and block writes

* © 2013 Springer-Verlag GmbH Berlin Heidelberg. This is the authors' version of the work. The final publication is available at `link.springer.com`.

are called by a common name: *I/Os*. When an I/O is necessary, the processor should wait until the block transfer is completed.

The motivation for the present study came from a paper by Brodal and Moruz [4], where they showed that skewed search trees can perform better than perfectly balanced search trees. This anomaly is mainly caused by branch mispredictions. Earlier, Kaligosi and Sanders [11] had observed a similar anomaly in quicksort; namely, when the pivot is given for free, a skewed pivot-selection strategy can perform better than the exact-median pivot-selection strategy.

In a companion paper [6] we proved that, when a simple static branch predictor is in use, any program can be transformed into an equivalent program that only contains $O(1)$ conditional branches and induces at most $O(1)$ branch mispredictions. That is, in most cases branch-misprediction anomalies can be avoided, but this program transformation may increase the number of instructions executed. In spite of the existence of this general transformation, current compilers can do branch optimization only in some special cases, and it is difficult for the programmer to force the compiler to do it.

As in the study of Brodal and Moruz [4], our goal is to find the best data representation for a static collection of N integers so that random membership searches can be supported as efficiently as possible. We restrict this study to classical comparison-based methods, so we will not utilize the universe size in any way. Our hypothesis is that, in this setup, balanced search trees are better than skewed search trees. We provide both theoretical and experimental evidence for this proposition. To summarize, the following facts support its validity.

1. The search procedure for a perfectly balanced search tree can be programmed such that it performs at most $\lg N + O(1)$ (two-way) element comparisons (Section 2). When element comparisons are expensive, their cost will dominate the overall costs.
2. The search procedure can be modified, without a significant slowdown, such that it induces $O(1)$ branch mispredictions (Section 4) and performs $O(1)$ conditional branches (Section 7). Hereafter most problems related to branch mispredictions are avoided and skewing does not give any advantage.
3. In several earlier studies (see, e.g. [2, 5, 14, 16–18]) it has been pointed out that the layout of a search tree can be improved such that it incurs $O(\log_B N)$ cache misses per search (Section 5), where B is the size of the cache lines measured in elements. By making the layout implicit, the representation can be made even more compact and more cache-friendly (Section 6).

In addition to providing branchless implementations of search algorithms, we investigated their practical performance. The experiments were carried out on a laptop (Intel$^{®}$ Core$^{\text{TM}}$ i5-2520M CPU @ 2.50GHz $\times$ 4) running Ubuntu 12.04 (Linux kernel 3.2.0-36-generic) using `g++` compiler (`gcc` version 4.6.3) with optimization `-O3`. The size of 12-way-associative L3 cache was 3 MB, the size of cache lines 64 B, and the size of the main memory 3.8 GB. Micro-benchmarks showed that in this computer unpredictable conditional branches were slow, whereas predictable conditional branches resulted in faster code than conditional-move primitives available at the assembly-language level. As expected, random access was much slower than sequential access, but the last-level cache (L3) was relatively large so cache effects were first visible for large problem instances.

In all experiments the elements manipulated were 4-byte integers; it was ensured that the input elements were distinct. All execution times were measured using the function `gettimeofday` accessible in the C standard library. All branch-misprediction and cache-miss measurements were carried out using the simulators available in `valgrind` (version 3.7.0). Each experiment was repeated 10^6 times and the mean over all test runs was reported.

Our main purpose was to use the experiments as a sanity check, not to provide a thorough experimental evaluation of the search procedures. We ported the programs to a couple of other computers and the behaviour was similar to that on our test computer. On the other hand, a few tests were enough to reveal that the observed running times are highly dependent on the environment—like the computer architecture, compiler, and operating system. Readers interested in more detailed comparison of the programs are advised to verify the results on their own platforms. The search programs discussed in this paper are in the public domain [12].

2 Search Procedure

We assume that a node in a binary search tree stores an element and three pointers pointing to the left child, right child, and parent of that node, respectively. For a node `x`, we let `(*x).element()`, `(*x).left()`, `(*x).right()`, and `(*x).parent()` denote the values of these four fields.

In a textbook description, search algorithms often rely on three-way comparisons having three possible outcomes: less, greater, or equal. We, however, assume that only two-way comparisons are possible. The simulation of a three-way comparison involves two two-way comparisons. Hence, a naive implementation of the search procedure essentially performs two element comparisons per visited node. The search procedure using two-way branching was discussed and experimentally evaluated by Andersson [1]. With this procedure only one element comparison per level is performed, even though more nodes may be visited. According to Knuth [13, Section 6.2.1], the idea was described in 1962. This optimization is often discussed in textbooks in the context of binary search, but it works for every search tree.

Assume that we search for element `v`. The idea is to test whether to go to the left at the current node `x` (when `v < (*x).element()`) or not. If not, we may have equality; but, if there is a right child, we go to the right anyway and continue the search until `x` refers to null. The only modification is that we remember the last node `y` on the search path for which the test failed and we went to the right. Upon the end of the search, we do one more element comparison to see whether `(*y).element() < v` (we already know that `(*y).element()` $\leq$ `v`). It is not hard to see that, if `v` $=$ `(*y).element()`, the last node on the search path is either `y` itself (if `(*y).right()` is null) or its successor (all subsequent tests fail and we follow the left spine from `(*y).right()`).

Let `N` denote the type of the nodes and `V` the type of the elements, and let `less` be the comparison function used in element comparisons. Using C++, the implementation of the search procedure requires a dozen lines of code.

```
bool is_member(V const& v) {
  N* y = nullptr; // candidate node
  N* x = root; // current node
  while (x != nullptr) {
    if (less(v, (*x).element())) {
      x = (*x).left();
    }
    else {
      y = x;
      x = (*x).right();
    }
  }
  if (y == nullptr || less((*y).element(), v)) {
    return false;
  }
  return true;
}
```

3 Models

We wanted to study the branch-prediction and cache behaviour separately. Therefore, instead of testing the search procedure in a real-world scenario, we decided to use models that capture its branch-prediction behaviour (see Fig. 1). The *three-way model* emulates the search procedure for a random search when three-way branching is used; the *two-way model* does the same when two-way branching is used; and the *branchless model* mixes Boolean and integer arithmetic to avoid the conditional branch altogether. In principle, all the models work in the same way: They reduce the search range from `N` to ℓ or to `N` $- 1 - \ell$ depending on the outcome of the branch executed; here ℓ is the border between the left and right portions. In the actual implementation the random numbers were generated beforehand, stored in an array, and retrieved from there.

```
k = random() * N;
while (N > 1) {
  ℓ = α * N;
  if (k < ℓ) {
    N = ℓ;
  }
  else if (k > ℓ) {
    k = k - 1 - ℓ;
    N = N - 1 - ℓ;
  }
  else {
    N = 1;
  }
}
```

Three-way model

```
k = random() * N;
while (N > 1) {
  ℓ = α * N;
  if (k < ℓ) {
    N = ℓ;
  }
  else {
    k = k - ℓ;
    N = N - 1 - ℓ;
  }
}
```

Two-way model

```
k = random() * N;
while (N > 1) {
  ℓ = α * N;
  Δ = (k < ℓ);
  k = k - ℓ + Δ * ℓ;
  N = Δ * ℓ + (1 - Δ) * (N - 1 - ℓ);
}
```

Branchless model

Fig. 1. The models considered

Table 1. Runtime performance of the models; execution time per $\lg N$ [in nanoseconds]

N	**Three-way**			**Two-way**			**Branchless**		
	$\alpha = \frac{1}{2}$	$\frac{1}{3}$	$\frac{1}{4}$	$\alpha = \frac{1}{2}$	$\frac{1}{3}$	$\frac{1}{4}$	$\alpha = \frac{1}{2}$	$\frac{1}{3}$	$\frac{1}{4}$
2^{16}	7.6	7.4	7.8	7.7	7.6	8.2	7.0	8.4	9.8
2^{24}	7.7	7.5	8.0	7.8	7.6	8.1	7.2	8.4	9.7
2^{32}	8.0	7.9	8.3	8.1	7.8	8.4	7.6	8.7	10.0

Table 2. Branch behaviour of the models; number of conditional branches executed (⧀) and branch mispredictions induced per search, both divided by $\lg N$

N	**Three-way** $\alpha = \frac{1}{3}$		**Two-way** $\alpha = \frac{1}{3}$		**Branchless** $\alpha = \frac{1}{2}$	
	⧀	**Mispred.**	⧀	**Mispred.**	⧀	**Mispred.**
2^{16}	2.67	0.44	2.15	0.45	1.00	0.06
2^{24}	2.67	0.44	2.16	0.44	1.00	0.04
2^{32}	2.79	0.44	2.17	0.44	1.00	0.03

We measured the execution time and the number of branch mispredictions induced by these models for different values of N. We report the results of these experiments in Tables 1 and 2. There are three things to note. First, the differences in the running times are not that big, but the branchless version is the fastest. Second, the model relying on three-way branching executes more conditional branches than the model relying on two-way branching. Third, since the outcome of at least one of the conditional branches inside the loop is difficult to predict, the first two models may suffer from branch mispredictions.

These experiments seem to confirm the validity of the experimental results reported by Brodal and Moruz [4]. For example, for the three-way model, for $N = 2^{32}$, the number of element comparisons increased from $2.4 \lg N$ ($\alpha = \frac{1}{2}$, not shown in Table 2) to $2.79 \lg N$ ($\alpha = \frac{1}{3}$), but the branch-misprediction rate went down from 0.51 ($\alpha = \frac{1}{2}$) to 0.44 ($\alpha = \frac{1}{3}$). This was enough to obtain an improvement in the running time. For larger values of α the running times got again higher because of the larger amount of work done. The behaviour of the two-way model was very similar to that of the three-way model. On the other hand, the situation was different for the branchless model; the value $\alpha = \frac{1}{2}$ always gave the best results and the branch-misprediction rate was 0.06 or lower.

4 Skewed Search Trees

A *skewed binary search tree* [4] is a binary search tree in which the left subtree of a node is always lighter than the right subtree. Moreover, this bias is exact so that, if *weight*(`x`) denotes the number of nodes stored in the subtree rooted at node `x`, *weight*(`(*x).left()`) $= \lfloor \alpha \cdot \mathit{weight}(\mathtt{x}) \rfloor$ for a fixed constant α, $0 < \alpha \leq \frac{1}{2}$. A *perfectly balanced search tree* is a special case where $\alpha = \frac{1}{2}$. We implemented skewed search trees and profiled their performance for different values of α.

To start with, we consider a memory layout where each node is allocated randomly from a contiguous pool of nodes. In addition to the search procedure relying on a two-way element comparison described in Section 2, we implemented a variant where the `if` statement inside the `while` loop was eliminated. The resulting program is interestingly simple.

```
N* choose(bool c, N* x, N* y) {
  return (N*)((char*) y + c * ((char*) x - (char*) y));
}

bool is_member(V const& v) {
  N* y = nullptr; // candidate node
  N* x = root; // current node
  while (x != nullptr) {
    bool smaller = less(v, (*x).element());
    y = choose(smaller, y, x);
    x = choose(smaller, (*x).left(), (*x).right());
  }
  if (y == nullptr || less((*y).element(), v)) {
    return false;
  }
  return true;
}
```

The statement `z = choose(c, x, y)` has the same effect as the C statement `z = (c) ? x : y`, i.e. it executes a conditional assignment. Here it would have been natural to use the conditional-move primitive provided by the hardware, but we wanted to avoid it for two reasons. First, it would have been necessary to implement this in assembly language, because there was no guarantee that the primitive would be used by the compiler. Second, our micro-benchmarks showed that the conditional-move primitive was slow in our test computer.

Assuming that the underlying branch predictor is static and that a `while` loop is translated by ending the loop conditionally, the outcome of the conditional branch at the end of the `while` loop is easy to predict. If we assume that backward branches are taken, this prediction is only incorrect when we step out of the loop. Including the branch mispredictions induced by the last `if` statement, the whole procedure may only induce $O(1)$ branch mispredictions per search.

The performance of the search procedures is summed up in Tables 3 and 4. Compared to the models discussed in the previous section, the running times are higher because of memory accesses. The results obtained are consistent with our

Table 3. Runtime performance of the search procedures for skewed search trees (random layout); execution time per $\lg N$ [in nanoseconds]

N	**Skewed random Two-way**			**Skewed random Branchless**		
	$\alpha = \frac{1}{2}$	$\frac{1}{3}$	$\frac{1}{4}$	$\alpha = \frac{1}{2}$	$\frac{1}{3}$	$\frac{1}{4}$
2^{15}	10.3	10.5	10.7	7.6	11.0	12.3
2^{20}	38.5	40.6	44.5	36.1	42.6	48.8
2^{25}	75.8	82.6	91.9	76.8	86.3	97.8

Table 4. Branch behaviour of the search procedures for skewed search trees (random layout); number of conditional branches executed (⧀) and branch mispredictions induced per search, both divided by $\lg N$

N	**Balanced random**		**Skewed random**		**Skewed random**		**Balanced random**	
	Two-way $\alpha = \frac{1}{2}$		**Two-way** $\alpha = \frac{1}{3}$		**Two-way** $\alpha = \frac{1}{4}$		**Branchless** $\alpha = \frac{1}{2}$	
	⧀	**Mispred.**	⧀	**Mispred.**	⧀	**Mispred.**	⧀	**Mispred.**
2^{15}	2.13	0.57	2.28	0.52	2.53	0.46	1.13	0.07
2^{20}	2.10	0.55	2.26	0.50	2.52	0.44	1.10	0.05
2^{25}	2.08	0.54	2.24	0.49	2.51	0.42	1.08	0.04

earlier experiments [6, 7], where branch removal turned out to be beneficial for small problem instances. For the problem sizes we considered, when the memory layout was random, we could not repeat the runtime results of Brodal and Moruz [4], although the branch-misprediction rate went down. For the branchless procedure there is a perfect match between theory and practice: It executes $\lg N + O(1)$ conditional branches but only induces $O(1)$ branch mispredictions.

5 Local Search Trees

By increasing the locality of memory references, improvements in performance can be seen at two levels:

- At the cache level, when more elements are in the same cache line, fewer cache misses will be incurred.
- At the memory level, when more elements are in the same page, fewer TLB (translation-lookaside buffer) misses will be incurred. At each memory access, the virtual memory address has to be translated into a physical memory address. The purpose of the TLB is to store this mapping for the most recently used memory addresses to avoid an access to the page table.

By running a space-utilization benchmark [3, Appendix 3], we observed that a binary-search-tree node storing one 4-byte integer and three 8-byte pointers required 48 bytes of memory (even though the raw data is only 28 bytes). Since in our test computer the size of the cache lines is 64 bytes, we did not expect much improvement at the cache level. However, as elucidated in an early study by Oksanen and Malmi [14], improvements at the memory level can be noticeable.

In the literature many schemes have been proposed to improve memory-access patterns for data structures supporting searching. The cache-sensitive schemes can be classified into three categories: one-level layouts [2, 14], two-level layouts [5, 17], and more complex multi-level layouts [2, 16, 18]. When deciding which one to choose, our primary criterion was the simplicity of programming. We were not interested in schemes solely providing good big-Oh estimates for the critical performance measures; good practical performance was imperative. Furthermore, cache-obliviousness [16] was not an important issue for us since we were willing to perform some simple benchmarks to find the best values for the tuning parameters in our test environment.

In a *local search tree* the goal is to lay out the nodes such that, when a path from the root to a leaf is traversed, as few pages are visited as possible. An interesting variant is obtained by seeing the tree as an F-ary tree, where each so-called *fat node* stores a complete binary tree. Since the subtrees inside the fat nodes are complete, F must be of the form 2^h for an integer h, $h > 0$, and each fat node of size (up to) $2^h - 1$. The F-ary tree can be laid out in memory as an F-ary heap [10] and the trees inside the fat nodes as a binary heap [19].

Jensen et al. [9] gave formulas how to get from a node to its children and parent in this layout. Using these formulas the pointers at each node can be set in a single loop. After this, it is easy to populate the tree provided that the elements are given in sorted order. Searching can be done as before. Each search will visit at most $\lceil \lceil \lg(1+N) \rceil / \lg F \rceil$ fat nodes. The fat nodes may not be perfectly aligned with the actual memory pages, but this is not a big problem since F is expected to be relatively large.

In our experiments we first determined the optimal value of F; in our test computer this turned out to be 16, but all values between 8 and 64 worked well. Then we compared our implementation of local search trees to the C++ standard-library implementation of red-black trees [8]. The results of this comparison are given in Tables 5 and 6. As seen from these and the previous results, red-black trees perform almost equally badly as perfectly balanced search trees when the memory layout is random. By placing the nodes more locally, almost a factor of two speed-up in the running time was experienced in our test environment.

Table 5. Runtime performance of the search procedures for two search trees; running time per search divided by $\lg N$ [in nanoseconds]

N	**Local**		**Red-black**
	Two-way	**Branchless**	**Two-way**
2^{15}	7.6	5.9	10.3
2^{20}	21.9	20.9	36.6
2^{25}	33.5	36.1	64.7

Table 6. Branch behaviour of the search procedures for two search trees; number of conditional branches executed ($\textcircled{<}$) and branch mispredictions induced per search, both divided by $\lg N$

N	**Local Two-way**		**Local Branchless**		**Red-black Two-way**	
	$\textcircled{<}$	**Mispred.**	$\textcircled{<}$	**Mispred.**	$\textcircled{<}$	**Mispred.**
2^{15}	2.16	0.57	1.12	0.07	2.12	0.58
2^{20}	2.05	0.55	1.05	0.05	2.09	0.56
2^{25}	2.13	0.54	1.09	0.04	2.08	0.59

6 Implicit Search Trees

If the data set is static, an observant reader may wonder why to use a search tree when a sorted array will do. A sorted array is an implicit binary search tree where arithmetic operations are used to move from one node to another; no explicit pointers are needed. Local search trees can also be made implicit using the formulas given in [9].

It was easy to implement implicit local search trees starting from the code for local search trees. Each time the left child (or the right child or the parent) of a node `x` was accessed, instead of writing `(*x).left()`, we replaced it with the corresponding formula. As for local search trees, we did not align the fat nodes perfectly with the cache lines. This would have been possible by adding some padding between the elements, but we wanted to keep the data structure compact and the formulas leading to the neighbouring nodes unchanged.

The results of our tests for implicit search trees are given in Tables 7 (running time), 8 (branch mispredictions), and 9 (cache misses). As a competitor to our search procedures, we considered two implementations of binary search (`std::binary_search` and our branchless modification of it) applied for a sorted array. Also here, before the final tests, we determined the best value for the parameter F; in our environment it was 16.

As to the running time (Table 7), for large problem instances implicit local search trees were the fastest of all structures considered in this study, but for small problem instances the overhead caused by the address calculations was clearly visible and implicit local search trees were slow.

Table 7. Runtime performance of the search procedures for implicit search trees; running time per search divided by $\lg N$ [in nanoseconds]

N	**Implicit local**		**Sorted array**	
	Two-way	**Branchless**	**Two-way**	**Branchless**
2^{15}	15.0	14.2	6.9	7.2
2^{20}	16.3	15.6	14.7	20.7
2^{25}	20.1	22.8	32.1	45.8

Table 8. Branch behaviour of the search procedures for implicit search trees; number of conditional branches executed (⧀) and branch mispredictions induced per search, both divided by $\lg N$

N	**Implicit local Two-way**		**Implicit local Branchless**		**Sorted array Two-way**		**Sorted array Branchless**	
	⧀	**Mispred.**	⧀	**Mispred.**	⧀	**Mispred.**	⧀	**Mispred.**
2^{15}	3.21	0.86	1.12	0.07	2.07	0.57	1.13	0.10
2^{20}	3.05	0.84	1.05	0.05	2.05	0.55	1.10	0.05
2^{25}	3.18	0.82	1.09	0.04	2.04	0.54	1.08	0.04

Table 9. Cache behaviour of three search trees; number of memory references performed, cache I/Os performed, and cache misses incurred per search, all divided by $\log_B N$, where B is the number of elements that fit in a cache line (16 in our test)

N	**Local** Two-way Refs.			**Implicit local** Two-way Refs.			**Sorted array** Two-way Refs.		
	Refs.	**I/Os**	**Misses**	**Refs.**	**I/Os**	**Misses**	**Refs.**	**I/Os**	**Misses**
2^{15}	9.19	2.43	0.00	9.46	0.40	0.00	6.93	2.00	0.00
2^{20}	8.60	3.27	1.32	8.80	0.81	0.12	6.70	3.20	0.73
2^{25}	8.84	3.77	2.27	9.00	1.01	0.51	6.56	3.37	3.01

One interesting fact of the formulas used for computing the indices of the neighbouring nodes is that they all contain an `if` statement. This is because a separate handling is necessary depending on whether we are inside a fat node or whether we move from one fat node to another. By inspecting the assembly-language code generated by the compiler, we observed that for the branchless version the compiler used conditional moves to eliminate these `if` statements, whereas for the non-optimized version conditional branches were used. This explains the discrepancies in the numbers in Table 8 (approximatively 3 vs. 1 conditional branches per iteration).

Finally, we compared the cache behaviour of local, implicit local, and implicit search trees (Table 9). We measured the number of memory references, cache I/Os, and cache misses. For small problem instances there were no cache misses since the data structures could be kept inside the cache. For implicit local search trees both the number of cache I/Os and the number of cache misses were smaller than the corresponding numbers for other structures. For $N = 2^{25}$, the $2.25 \lg N$ memory accesses generated $1.01 \log_B N$ cache I/Os, which is basically optimal when the size of the cache blocks is B; only half of the I/Os ended up to be a miss. One can explain the low number of cache misses by observing that the cache of our test computer is large enough so that, with an ideal cache replacement, the top portion of the search tree will be kept inside the cache at all times. A sorted array was another extreme; it made more than three times as many cache I/Os and almost every cache I/O incurred a cache miss.

7 Unrolling the Loop

It is well-known that loop unrolling can be used to improve the performance of programs in many respects (see, e.g. [3, Appendix 4]). Bentley [3, Column 4] gave a nice description of an ancient idea how to unroll binary search. The same technique applies for the search procedure of perfectly balanced search trees. In this section we describe how to do this unrolling so that the search procedure only contains a few branches and has no loops. An immediate corollary is that such a straight-line program cannot induce more than a constant number of branch mispredictions per search.

When the search tree is perfectly balanced such that the lengths of root-to-leaf paths only vary by one, the following kind of procedure will do the job.

```
bool is_member(V const& v) {
  N* y = nullptr; // candidate node
  N* x = root; // current node
  bool smaller;
  switch (height) {
  case 31:
    smaller = less(v, (*x).element());
    y = choose(smaller, y, x);
    x = choose(smaller, (*x).left(), (*x).right());
⋮
  case 1:
    smaller = less(v, (*x).element());
    y = choose(smaller, y, x);
    x = choose(smaller, (*x).left(), (*x).right());
  default:
    smaller = (x == nullptr) || less(v, (*x).element());
    y = choose(smaller, y, x);
  }
  if ((y == nullptr) || less((*y).element(), v)) {
    return false;
  }
  return true;
}
```

To manage skewed trees the bottom-most levels must be handled as in the normal search procedure, because we cannot be sure when `x` refers to null. For skewed trees the procedure should also be able to tolerate larger heights.

A theoretician may oppose this solution because the length of the program is not a constant. A practitioner may be worried about the portability since we assumed that the maximum height of the tree is 31. Both of these objections are reasonable, but neither is critical. The maximum height could be made larger and the program could even be generated on the fly after the user has specified the height of the search tree.

As a curiosity we tested the efficiency of the unrolled search procedure; we only considered perfectly balanced trees with random memory layout. The running-time performance did not improve at all. The main reason for this seems to be that in our test computer the cost of easy-to-predict branches is so low. On the other hand, both the branch-count rate and the branch-misprediction rate went to zero when N increased.

8 Conclusion

We were mainly interested in understanding the impact of branch mispredictions on the performance of the search procedures. We hope that we could make our case clear: Branch prediction is not a good enough reason to switch from balanced search trees to skewed search trees. In theory, any program can be transformed into a form that induces $O(1)$ branch mispredictions [6]. As shown in the present paper, in the context of searching, simple transformations can be used to eliminate branches and thereby avoid branch mispredictions. In most cases, these transformations work well in practice.

References

1. Andersson, A.: A note on searching in a binary search tree. Software Pract. Exper. 21(10), 1125–1128 (1991)
2. Bender, M.A., Demaine, E.D., Farach-Colton, M.: Efficient tree layout in a multilevel memory hierarchy. In: Möhring, R., Raman, R. (eds.) ESA 2002. LNCS, vol. 2461, pp. 165–173. Springer, Heidelberg (2002)
3. Bentley, J.: Programming Pearls. Addison-Wesley, Reading, 2nd edn. (2000)
4. Brodal, G.S., Moruz, G.: Skewed binary search trees. In: Azar, Y., Erlebach, T. (eds.) ESA 2006. LNCS, vol. 4168, pp. 708–719. Springer, Heidelberg (2006)
5. Chen, S., Gibbons, P.B., Mowry, T.C., Valentin, G.: Fractal prefetching B$^+$-trees: Optimizing both cache and disk performance. In: SIGMOD 2002. pp. 157–168. ACM, New York (2002)
6. Elmasry, A., Katajainen, J.: Lean programs, branch mispredictions, and sorting. In: Kranakis, E., Krizanc, D., Luccio, F. (eds.) FUN 2012. LNCS, vol. 7288, pp. 119–130. Springer, Heidelberg (2012)
7. Elmasry, A., Katajainen, J., Stenmark, M.: Branch mispredictions don't affect mergesort. In: Klasing, R. (ed.) SEA 2012. LNCS, vol. 7276, pp. 160–171. Springer, Heidelberg (2012)
8. Guibas, L.J., Sedgewick, R.: A dichromatic framework for balanced trees. In: FOCS 1978. pp. 8–21. IEEE Computer Society, Los Alamitos (1978)
9. Jensen, C., Katajainen, J., Vitale, F.: Experimental evaluation of local heaps. CPH STL Report 2006-1, Department of Computer Science, University of Copenhagen, Copenhagen (2006)
10. Johnson, D.B.: Priority queues with update and finding minimum spanning trees. Inform. Process. Lett. 4(3), 53–57 (1975)
11. Kaligosi, K., Sanders, P.: How branch mispredictions affect quicksort. In: Azar, Y., Erlebach, T. (eds.) ESA 2006. LNCS, vol. 4168, pp. 780–791. Springer, Heidelberg (2006)
12. Katajainen, J.: Branchless search programs: Electronic appendix. CPH STL Report 2012-1, Department of Computer Science, University of Copenhagen, Copenhagen (2012)
13. Knuth, D.E.: Sorting and Searching, The Art of Computer Programming, vol. 3. Addison-Wesley, Reading, 2nd edn. (1998)
14. Oksanen, K., Malmi, L.: Memory reference locality and periodic relocation in main memory search trees. In: 5th Hellenic Conference on Informatics. pp. 679–687. Greek Computer Society, Athens (1995)
15. Patterson, D.A., Hennessy, J.L.: Computer Organization and Design: The Hardware/Software Interface. Morgan Kaufmann, Waltham, revised 4th edn. (2012)
16. Prokop, H.: Cache-oblivious algorithms. Master's Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge (1999)
17. Rahman, N., Cole, R., Raman, R.: Optimised predecessor data structures for internal memory. In: Brodal, G.S., Frigioni, D., Marchetti-Spaccamela, A. (eds.) WAE 2001. LNCS, vol. 2141, pp. 67–78. Springer, Heidelberg (2001)
18. Saikkonen, R., Soisalon-Soininen, E.: Cache-sensitive memory layout for binary trees. In: 5th IFIP International Conference on Theoretical Computer Science. IFIP International Federation for Information Processing, vol. 273, pp. 241–255. Springer, New York (2008)
19. Williams, J.W.J.: Algorithm 232: Heapsort. Commun. ACM 7(6), 347–348 (1964)