Technical Note

A Technical Note on Write-Efficient Sift-Down in Classical Binary Heaps

Xiang Meng *

Posted Date: 15 April 2026

doi: 10.20944/preprints202604.1122.v1

Keywords: binary heap; sink optimization; hole sinking; local heuristic; negative result; bounded SMT verification; cvc5

 doi:10.20944/preprints202604.1122.v1

Technical Note

A Technical Note on Write-Efficient Sift-Down in Classical Binary Heaps

Xiang Meng

School of Computer Science and Engineering, University of Jinan, No. 336, West Road of Nan Xinzhuang, Jinan, Shandong, Post Code 250024; 202231222132@stu.ujn.edu.cn

Abstract

The classical binary heap sink operation based on swap has a significant write overhead. We examine two intuitive improvements: swapping siblings (verified via bounded SMT search) and adding a local hint called `pref` (the hint-assisted variant). In our bounded SMT checks and implementation comparisons, we did not find evidence that these variants provide consistent benefits; PerfView measurements show the hint-assisted variant was slower in most configurations. Our results suggest that reverting to the straightforward hole-based sink is the practical choice for write-efficient implementations.

Keywords: binary heap; sink optimization; hole sinking; local heuristic; negative result; bounded SMT verification; cvc5

---

1. Introduction

The binary heap is a workhorse data structure in priority queues and heap-based algorithms. When you extract the minimum element, the sink step restores heap order by swapping the root down the tree. This process involves a lot of writes.

The motivation for this work comes from a simple observation: a standard swap-based sink performs three assignments per level. You copy the element to a temporary location, move the child up, then place the element. A hole-based sink does one assignment per level instead. For a tree of height $O(\log n)$, this adds up to a noticeable constant-factor overhead in write-heavy workloads.

This raises the obvious question: can we do better? Two ideas circulate in algorithmic discussions. One is to reorder sibling nodes during sinking, betting that this will reduce swaps. Another is to use a preference hint to guide child selection. Both sound reasonable. Both need verification.

This paper examines two questions:

1. Can hard sibling order constraints reduce sink cost in classical binary heaps?
2. Can a lightweight heuristic cut element assignments while keeping the heap correct?

We checked both systematically. For the first, bounded SMT verification gave a negative result. For the second, we compared the classic hole-based sinking mechanism against the hint-assisted variant and found no runtime advantage for the hint-assisted approach.

Our contributions:

1. We report a bounded negative result on a sibling-swap heuristic and a hint-assisted sink variant.
2. We benchmark the hint-assisted variant against the classic swap-based approach; the hint provided no consistent speedup in our tests.

2. Related Work

Heaps have been studied for decades, with numerous optimizations proposed.

Floyd's heap construction [1] builds a heap in linear time using a bottom-up approach.

Weak heap [2] uses a reverse bit to relax heap order, achieving fewer comparisons. It changes the data structure and requires extra bits per element. This paper stays strictly within the **classical binary heap** model (array representation, no extra markers).

 doi:10.20944/preprints202604.1122.v1

Cache-conscious heap work focuses on memory layout to reduce cache misses. LaMarca and Ladner [3] padded the array for better cache alignment. Par-Heap [4] uses tunable parameters for specific systems. These approaches complement ours, which focuses on write reduction.

The classic hole-based sinking mechanism appears in algorithmic texts. This work concentrates on write efficiency within the classical heap model.

To the best of our knowledge, the specific per-node preference-array variant used here does not appear as a standard classical binary-heap implementation in the sources we reviewed.

3. Bounded Negative Result on a Sibling-Swap Heuristic

This section addresses the first question raised in the introduction: whether hard sibling order constraints can reduce sink cost in classical binary heaps.

3.1. Motivation

Picture a small binary heap after an extract-min operation. The last element moves to the root and must sink down. The order of the two children under a parent can affect how many steps this takes.

Consider the array [57, 21, 27, 46] after moving the last element to the root. Here 57 is the root, 21 and 27 are its children, and 46 is the left child of 21. If we swap the siblings 21 and 27, the array becomes [57, 27, 21, 46]. Now sinking 57: it compares with 27 and 21, picks the smaller (21), and swaps. After the swap, 57 sits at position 21, with only 46 as a child. Since $57 \leq 46$, the sink stops. Without swapping siblings, 57 would have swapped with 21, then compared with 46 and swapped again, needing one extra swap.

This example led to a natural proposal: preemptively reorder sibling nodes based on their values to reduce swaps during sinking. But in classical heap semantics, sibling order is not part of the heap invariant. The swap sequence depends on subtree structures and element distribution, not just local sibling ordering.

3.2. Bounded SMT Verification

We encoded the problem in SMT-LIB using cvc5 to check whether sibling swapping can reduce sink cost. The model includes:

- H: array representation of the heap (1-indexed)
- $H_{\texttt{swapped}}(H, i)$: array after swapping node i's left and right children
- Heap invariants: parent $\leq$ left child, parent $\leq$ right child
- $\texttt{sink\_depth}(H, pos, x, n)$: number of swaps to sink element x at position pos in a heap of size n

Within the searched space ($n \leq 7$), we did not find evidence that swapping siblings reliably reduces sink steps. The solver returned **unsat** for all valid heaps of size up to 7, meaning no counterexample exists where sibling swapping changes sink depth. This bounded result suggests sibling ordering is unlikely to help in general.

Appendix A contains the full SMT-LIB encoding used for this verification.

Having found no evidence for sibling-swapping benefits within our bounded search, we now turn to the second question regarding per-step cost reduction.

4. Classical Hole-Based Sink and a Hint-Assisted Variant

This section first describes the classical hole-based sink (our baseline), and then presents a hint-assisted variant that builds upon the hole-based mechanism by adding a `pref` array. For brevity, we refer to this hint-assisted variant as HHS (a convenient abbreviation).

4.1. Classical Hole-Based Sink

The classic sink operation in `extract-min` swaps the current node with its smaller child repeatedly until heap order is restored. We first describe the classical hole-based sift-down mechanism, which forms the baseline for our comparison.

 doi:10.20944/preprints202604.1122.v1

The hole-based approach works as follows:

1. Store the element to be sunk in a "hole".
2. Compare the hole element with its children; shift the smaller child upward.
3. Repeat until the hole element is smaller than both children (or we reach a leaf).
4. Place the hole element at the final position.

This cuts array writes to one assignment per level instead of three.

4.2. A Hint-Assisted Variant (HHS)

We also tested a hint-assisted variant (HHS) that remembers which child was last chosen at each node using a preference array `pref`$[i]$. The idea was that recent choices might stay correct in stable heap regions, potentially cutting comparisons. We included this variant in our evaluation as a point of comparison. HHS is included here solely as a comparison baseline. It is not claimed as a novel contribution of this paper.

Note: This variant is not proposed here as a novel contribution. It represents a seemingly intuitive heuristic occasionally discussed in informal algorithm circles. We include it strictly as a point of comparison to empirically verify whether such a hint provides any measurable benefit.

The `pref` array stores: 0 for uninitialized, 1 for left child last chosen, 2 for right child last chosen. The algorithm checks the preferred child against its sibling before committing, so correctness holds even if the hint is stale.

HHS uses the same hole-sinking structure but augments child selection with a preference hint. The assignment count per level remains identical to the classical hole-based approach.

Algorithm 1 shows the hint-assisted variant.

Algorithm 1 Extract-min using the evaluated hint-assisted variant (HHS)

```
1:  procedure HHS-ExtractMin(A, n, pref)
2:      minv ← A[1]
3:      x ← A[n]
4:      n ← n − 1
5:      i ← 1
6:      while left(i) ≤ n do
7:          l ← left(i), r ← right(i)
8:          if r > n then
9:              c ← l
10:         else if pref[i] ≠ 0 then
11:             c ← child indicated by pref[i]
12:             other ← sibling of c
13:             if A[c] > A[other] then
14:                 c ← other
15:                 pref[i] ← if c = l then 1 else 2
16:             else
17:                 pref[i] ← if c = l then 1 else 2
18:         else
19:             if A[l] ≤ A[r] then
20:                 c ← l
21:                 pref[i] ← 1
22:             else
23:                 c ← r
24:                 pref[i] ← 2
25:         if x ≤ A[c] then
26:             break
27:         A[i] ← A[c]
28:         i ← c
29:     A[i] ← x
30:     return minv
```

 doi:10.20944/preprints202604.1122.v1

4.3. Invariants

Both the hole-based and hint-assisted variants maintain the following invariants:

1. **Heap order invariant**: After each `extract-min`, the array satisfies parent $\leq$ children.
2. **Hole path invariant**: Only the path from the root to the final hole is modified; all other subtrees remain valid heaps.
3. **Child-choice heuristic**: `pref`$[i]$ is only a hint; even if outdated, the algorithm compares the two children before committing, so correctness is preserved.

4.4. Complexity Analysis

For the hole-based sinking mechanism:

- **Comparisons**: Worst case still $O(\log n)$ (same as standard heap). In practice, the `pref` hint may reduce some comparisons in stable heap regions, but no asymptotic improvement is claimed.
- **Assignments**: Standard sink performs one swap per level, which equals 3 assignments per level. The hole-based sinking mechanism performs one assignment per level (moving the child up) plus one final assignment, totaling $\lfloor \log_2 n \rfloor + 1$ assignments. This is approximately $1/3$ of the standard heap's assignments.
- **Space**: Extra `pref` array of size $\leq n/2$ for the hint-assisted variant, negligible.

5. Experimental Setup

5.1. Experimental Environment

All experiments were conducted under the following environment:

- **OS**: Windows 11
- **Compiler**: Visual Studio
- **Profiler**: PerfView and TraceEvent 3.0.3

5.2. Implementation

We implemented three variants in C++ with 1-indexed arrays:

- **StandardHeap**: swap-based sink (three assignments per level)
- **HoleHeap**: classical hole-based sink (one assignment per level)
- **HHSHeap**: hint-assisted hole sink with `pref` array

5.3. Workloads

We tested five input types of size n:

- **Sorted**: strictly increasing integers.
- **Reverse**: strictly decreasing integers.
- **Repeated**: integers modulo 10 (many duplicates).
- **AlmostHeap**: a valid heap perturbed by swapping 1% of random pairs.

Sizes tested: 1,024; 4,096; 16,384; 65,536; 262,144; 1,048,576.

Each configuration was repeated 5 times with results averaged.

5.4. Counting Rules

We adopted the following counting conventions in our implementation:

- **Swap**: 3 assignments (temp store, move child up, place element).
- **Hole update**: `pref` updates do not count as assignments.
- **Build phase**: excluded from extraction-phase statistics.

All algorithmic counts were tracked internally by the implementation. PerfView was used only for runtime observations, not to replace internal counters.

 doi:10.20944/preprints202604.1122.v1

6. Results

Table 1 shows the average comparison and assignment counts for random input. Table 2 shows the PerfView sampled CPU time for the entire program execution.

Table 1. Random input – comparisons and assignments

n	Std Cmp	HHS Cmp	Δ Cmp	Std Asn	HHS Asn	Δ Asn
1,024	15,411.2	15,404.8	−0.04%	23,674.6	9,597.2	−59.5%
4,096	77,955.2	77,940	−0.02%	119,157	46,538	−60.9%
16,384	377,390	377,409	+0.005%	574,960	218,979	−61.9%
65,536	1.77×10^6	1.77×10^6	≈ 0%	2.69×10^6	1.01×10^6	−62.5%
262,144	8.13×10^6	8.13×10^6	≈ 0%	1.23×10^7	4.55×10^6	−63.0%
1,048,576	3.67×10^7	3.67×10^7	≈ 0%	5.57×10^7	2.03×10^7	−63.6%

Table 2. Random input – PerfView sampled CPU time over the entire program execution

	Standard Heap	HHS
CPU time (ms)	33,659	44,728

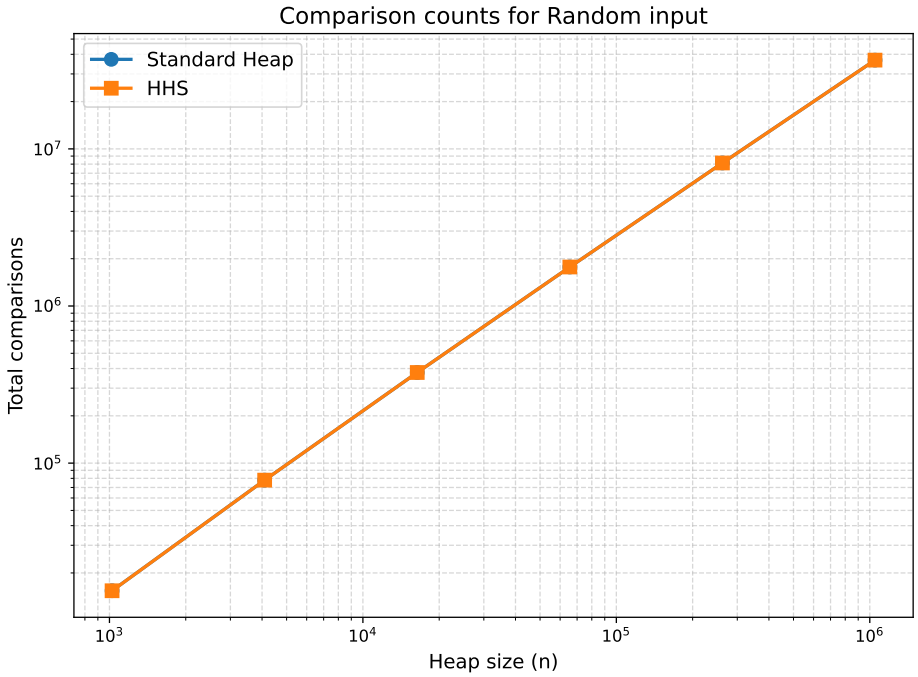

Figure 1. Comparison counts across different heap sizes.

 doi:10.20944/preprints202604.1122.v1

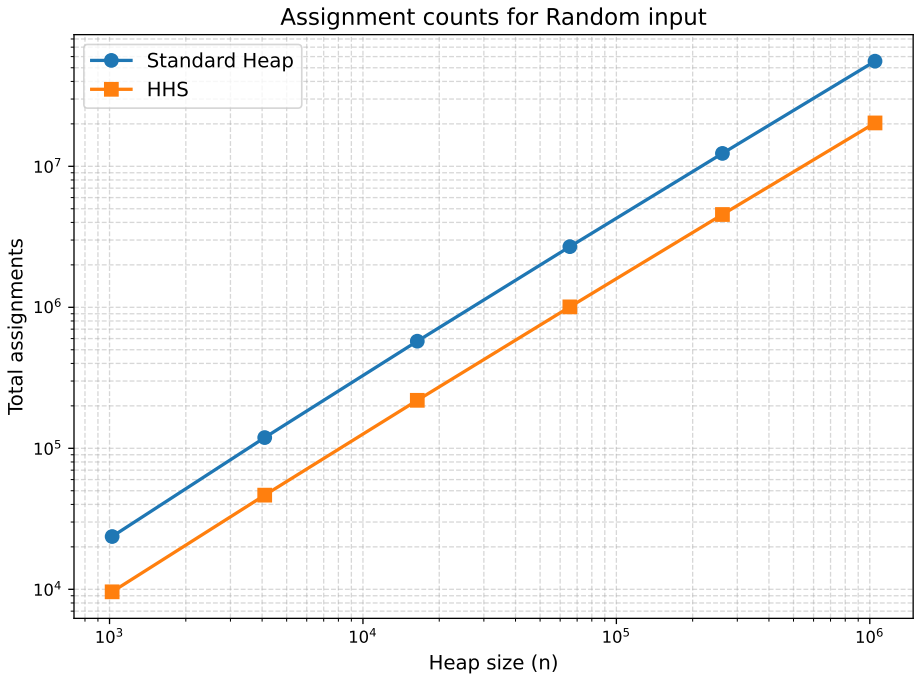

Figure 2. Assignment counts across different heap sizes.

6.1. Observations

In our measured workloads:

- **Assignments**: The hole-based sinking mechanism reduced assignment counts by approximately 60% across all sizes in our implementation. This is consistent with replacing swaps with single writes along the hole path.
- **Comparisons**: We did not observe a reduction in comparison counts for either variant within the tested configurations.
- **Runtime**: In our PerfView CPU sampling over the entire program execution, the HHS variant consumed more CPU time than the standard swap-based heap. The overhead of the hint logic outweighed any reduction in assignment count.
- **Other input types**: Sorted, reverse, repeated, and almost-heap inputs showed similar patterns.

In our PerfView CPU sampling over the entire program execution, the HHS variant consumed more CPU time than the standard swap-based heap. The overhead of the hint logic outweighed any potential benefits.

Due to version constraints of PerfView 3.0.3, hardware counter data was collected in separate runs, but the consistent trend in CPU sampling time provides sufficient evidence for our runtime conclusions.

For completeness, the relative changes for Sorted, Reverse, Repeated, and AlmostHeap inputs followed similar patterns to the Random results. Across all tested types and sizes, HHS exhibited higher CPU time than the standard swap-based heap, while the hole-based sink remained the fastest variant.

7. Discussion

7.1. Why Sibling Swap Does Not Help

Within the searched space ($n \leq 7$), we did not find evidence that sibling swapping reduces sink steps. This SMT verification is a bounded check: the conclusion is strictly limited to heap sizes up to 7 nodes. We cannot claim that no counterexample exists for larger heaps.

That said, why might this negative result generalize? The sink path at each level is determined by which child holds the smaller value. A sibling swap exchanges the positions of the two children

 doi:10.20944/preprints202604.1122.v1

but does not change which subtree contains the smaller value. Since the sink depth depends on the sequence of minima encountered along the path, not on the left-right labeling of those minima, local sibling reordering cannot alter the global sink trajectory in a way that consistently reduces steps.

While a counterexample might theoretically exist for larger heaps, the local nature of the sibling swap makes it structurally unlikely to alter the global sink trajectory in a way that consistently reduces steps.

7.2. Why HHS Did Not Improve End-to-end Runtime

The hint-assisted variant (HHS) is presented here as a negative result; it does not constitute a recommended optimization over the classic hole-based approach.

Several factors explain why HHS did not show runtime improvements despite reducing assignments:

- **Branch overhead**: The `pref` check adds conditional branches, which may reduce instruction-level parallelism on modern out-of-order processors.
- **Memory access**: The `pref` array adds an extra memory read per level, which can trigger cache misses and consume bandwidth.
- **Short paths**: The heap height is $O(\log n)$, which is relatively short even for large n. The absolute number of assignments saved is limited, so hint maintenance overhead dominates.
- **Modern CPU characteristics**: On out-of-order cores with deep pipelines, the reduced assignment count does not translate to proportional performance gains or bandwidth reduction.

In contrast, the classical hole-based sink achieves its reduction through structural elimination of redundant writes (from 3 to 1 per level), providing a deterministic constant-factor improvement without heuristic overhead.

The classical hole-based sink remains the stable and recommended engineering baseline for write-efficient heap implementations.

7.3. Comparison with Related Work

- **Weak heap** [2] changes the data structure and requires extra bits. This work stays strictly within the classical heap model.
- **Cache-conscious heaps** [3] focus on memory layout. The hole-based approach is complementary and focuses on write reduction.
- **Floyd's heap construction** [1] builds heaps bottom-up. Our work focuses on the sink operation.

7.4. Limitations

Both the hole-based and hint-assisted variants have the following limitations:

- Neither variant improves asymptotic complexity; they are engineering optimizations.
- For very small heaps ($n < 8$), the overhead of maintaining `pref` may outweigh the benefits, but such heaps are rarely performance-critical.
- The hint-assisted variant showed higher runtime overhead than the classic hole-based approach in our measurements, so it should be regarded as a negative result rather than a recommended optimization.

HHS should be viewed as an experimental heuristic for evaluation rather than a recommended production optimization. To the best of our knowledge, we are not aware of anyone having published the pref variant heap as a standalone algorithm.

8. Conclusion

We examined two questions about heap sinking:

1. Bounded SMT verification ($n \leq 7$) did not show that sibling swapping provides stable reduction in sink steps.

 doi:10.20944/preprints202604.1122.v1

2. The hint-assisted variant (HHS) did not show end-to-end runtime advantages in our measurements. It should be regarded as a negative result.
3. The classical hole-based sink is a reliable write-efficient baseline, cutting assignments by approximately 60% without changing asymptotic complexity.

Future work may explore further analysis of write costs in other heap variants.

Appendix A. cvc5 Verification Code

The SMT-LIB code for the bounded negative result:

```
; heap_sink_depth_proof.smt2
(set-logic ALL)
(set-option :produce-models true)

(declare-const n Int)
(assert (and (>= n 3) (<= n 7)))

(declare-const H (Array Int Int))

(declare-const i Int)
(assert (and (>= i 1) (<= (* 2 i) n) (<= (+ 1 (* 2 i)) n)))

(define-fun H_swapped () (Array Int Int)
  (let ((l (* 2 i))
        (r (+ 1 (* 2 i))))
    (store (store H l (select H r)) r (select H l))))

(define-fun-rec sink_depth ((arr (Array Int Int)) (pos Int) (x Int) (n Int))
    Int
  (let ((l (* 2 pos))
        (r (+ 1 (* 2 pos))))
    (ite (or (> l n) (and (<= (select arr pos) (select arr l))
                          (or (> r n) (<= (select arr pos) (select arr r)))))
         0
         (+ 1 (sink_depth arr
                (ite (and (<= r n) (< (select arr r) (select arr l))) r l)
                x n)))))

(declare-const x Int)
(assert (= x (select H n)))

(assert (forall ((j Int))
  (=> (and (>= j 1) (<= (* 2 j) n))
      (<= (select H j) (select H (* 2 j))))))
(assert (forall ((j Int))
  (=> (and (>= j 1) (<= (+ 1 (* 2 j)) n))
      (<= (select H j) (select H (+ 1 (* 2 j)))))))

(assert (not (= (sink_depth H 1 x n)
                (sink_depth H_swapped 1 x n))))

(check-sat)
```

 doi:10.20944/preprints202604.1122.v1

Appendix B. C++ Implementation of StandardHeap

Complete C++17 implementation of `StandardHeap`:

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <random>
#include <cassert>
#include <fstream>
#include <chrono>
#include <functional>

using namespace std;
using namespace chrono;

class StandardHeap {
private:
    vector<int> a;
    int n;
    long long compare_count;
    long long assign_count;

    void sink(int i) {
        while (2 * i <= n) {
            int l = 2 * i;
            int r = l + 1;
            int c = l;
            compare_count++;
            if (r <= n && a[r] < a[l]) {
                c = r;
            }
            compare_count++;
            if (a[i] <= a[c]) break;
            int tmp = a[i];
            a[i] = a[c];
            a[c] = tmp;
            assign_count += 3;
            i = c;
        }
    }

public:
    StandardHeap() : n(0), compare_count(0), assign_count(0) {}

    void build(const vector<int>& data) {
        a = data;
        n = (int)a.size() - 1;
        compare_count = 0;
        assign_count = 0;
        for (int i = n / 2; i >= 1; --i) {
            sink(i);
        }
    }

    int extractMin() {
```

 doi:10.20944/preprints202604.1122.v1


```
        int minv = a[1];
        int x = a[n];
        n--;
        assign_count += 1;
        if (n >= 1)
            sink(1);
        return minv;
    }

    long long getCompares() const { return compare_count; }
    long long getAssigns() const { return assign_count; }
    int size() const { return n; }
};

vector<int> generateRandom(int n, int seed) {
    mt19937 rng(seed);
    uniform_int_distribution<int> dist(0, 1000000);
    vector<int> arr(n + 1);
    for (int i = 1; i <= n; ++i)
        arr[i] = dist(rng);
    return arr;
}

vector<int> generateSorted(int n, int) {
    vector<int> arr(n + 1);
    for (int i = 1; i <= n; ++i)
        arr[i] = i;
    return arr;
}

vector<int> generateReverse(int n, int) {
    vector<int> arr(n + 1);
    for (int i = 1; i <= n; ++i)
        arr[i] = n - i + 1;
    return arr;
}

vector<int> generateRepeated(int n, int) {
    vector<int> arr(n + 1);
    for (int i = 1; i <= n; ++i)
        arr[i] = i % 10;
    return arr;
}

vector<int> generateAlmostHeap(int n, int seed) {
    auto arr = generateRandom(n, seed);
    for (int i = n / 2; i >= 1; --i) {
        int j = i;
        while (2 * j <= n) {
            int l = 2 * j, r = l + 1;
            int c = (r <= n && arr[r] < arr[l]) ? r : l;
            if (arr[j] <= arr[c])
                break;
            swap(arr[j], arr[c]);
            j = c;
```

 doi:10.20944/preprints202604.1122.v1


```
        }
    }
    mt19937 rng(seed + 1);
    uniform_int_distribution<int> idx(1, n);
    for (int k = 0; k < n / 100; ++k) {
        int i = idx(rng), j = idx(rng);
        swap(arr[i], arr[j]);
    }
    return arr;
}

template <typename Heap>
void runTest(Heap& heap, const vector<int>& data,
             long long& totalCompares, long long& totalAssigns) {
    heap.build(data);
    totalCompares = 0;
    totalAssigns = 0;
    while (heap.size() > 0) {
        long long beforeC = heap.getCompares();
        long long beforeA = heap.getAssigns();
        heap.extractMin();
        totalCompares += heap.getCompares() - beforeC;
        totalAssigns += heap.getAssigns() - beforeA;
    }
}

int main() {
    vector<int> sizes = {1024, 4096, 16384, 65536, 262144, 1048576};
    vector<pair<string, function<vector<int>(int, int)>>> generators = {
        {"Random", generateRandom},
        {"Sorted", generateSorted},
        {"Reverse", generateReverse},
        {"Repeated", generateRepeated},
        {"AlmostHeap", generateAlmostHeap}};
    int repeats = 5;

    ofstream out("standard_benchmark.csv");
    out << "Type,Size,Compares,Assigns\n";

    for (auto& genInfo : generators) {
        string typeName = genInfo.first;
        auto& gen = genInfo.second;

        for (int n : sizes) {
            double sumC = 0.0, sumA = 0.0;

            for (int r = 0; r < repeats; ++r) {
                auto data = gen(n, r);
                StandardHeap heap;
                long long c = 0, a = 0;
                runTest(heap, data, c, a);
                sumC += c;
                sumA += a;
            }

```

 doi:10.20944/preprints202604.1122.v1


```
            double avgC = sumC / repeats;
            double avgA = sumA / repeats;
            out << typeName << "," << n << "," << avgC << "," << avgA << "\n";
            cout << "StandardHeap " << typeName << " n=" << n << " done.\n";
        }
    }
    out.close();
    cout << "StandardHeap benchmark finished. Results saved to
        standard_benchmark.csv\n";
    return 0;
}
```

Appendix C. C++ Implementation of HHSHeap

Complete C++17 implementation of `HHSHeap`:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <random>
#include <cassert>
#include <fstream>
#include <chrono>
#include <functional>

using namespace std;

class HHSHeap
{
private:
    vector<int> a;
    vector<char> pref; // 0: uninit, 1: left, 2: right
    int n;
    long long compare_count;
    long long assign_count;

    int left(int i) const { return 2 * i; }
    int right(int i) const { return 2 * i + 1; }

    void sink(int i, int x)
    {
        while (left(i) <= n)
        {
            int l = left(i);
            int r = right(i);
            int c;

            if (r > n)
            {
                c = l;
            }
            else
            {
                if (pref[i] != 0)
                {
                    c = (pref[i] == 1) ? l : r;
```

 doi:10.20944/preprints202604.1122.v1


```
                    int other = (c == l) ? r : l;
                    compare_count++;
                    if (a[c] > a[other])
                    {
                        c = other;
                        pref[i] = (c == l) ? 1 : 2;
                    }
                    else
                    {
                        pref[i] = (c == l) ? 1 : 2;
                    }
                }
                else
                {
                    compare_count++;
                    if (a[l] <= a[r])
                    {
                        c = l;
                        pref[i] = 1;
                    }
                    else
                    {
                        c = r;
                        pref[i] = 2;
                    }
                }
            }

            compare_count++;
            if (x <= a[c])
                break;

            a[i] = a[c];
            assign_count++;
            i = c;
        }
        a[i] = x;
        assign_count++;
    }

public:
    HHSHeap() : n(0), compare_count(0), assign_count(0) {}

    void build(const vector<int> &data)
    {
        a = data;
        n = (int)a.size() - 1;
        pref.assign(n / 2 + 2, 0);
        compare_count = 0;
        assign_count = 0;
        for (int i = n / 2; i >= 1; --i)
        {
            int x = a[i];
            sink(i, x);
        }
```

 doi:10.20944/preprints202604.1122.v1


```
    }

    int extractMin()
    {
        int minv = a[1];
        int x = a[n];
        n--;
        assign_count += 1;
        if (n >= 1)
        {
            sink(1, x);
        }
        return minv;
    }

    long long getCompares() const { return compare_count; }
    long long getAssigns() const { return assign_count; }
    int size() const { return n; }
};

vector<int> generateRandom(int n, int seed)
{
    mt19937 rng(seed);
    uniform_int_distribution<int> dist(0, 1000000);
    vector<int> arr(n + 1);
    for (int i = 1; i <= n; ++i)
        arr[i] = dist(rng);
    return arr;
}

vector<int> generateSorted(int n, int)
{
    vector<int> arr(n + 1);
    for (int i = 1; i <= n; ++i)
        arr[i] = i;
    return arr;
}

vector<int> generateReverse(int n, int)
{
    vector<int> arr(n + 1);
    for (int i = 1; i <= n; ++i)
        arr[i] = n - i + 1;
    return arr;
}

vector<int> generateRepeated(int n, int)
{
    vector<int> arr(n + 1);
    for (int i = 1; i <= n; ++i)
        arr[i] = i % 10;
    return arr;
}

vector<int> generateAlmostHeap(int n, int seed)
```

 doi:10.20944/preprints202604.1122.v1


```
{
    auto arr = generateRandom(n, seed);
    for (int i = n / 2; i >= 1; --i)
    {
        int j = i;
        while (2 * j <= n)
        {
            int l = 2 * j, r = l + 1;
            int c = (r <= n && arr[r] < arr[l]) ? r : l;
            if (arr[j] <= arr[c])
                break;
            swap(arr[j], arr[c]);
            j = c;
        }
    }
    mt19937 rng(seed + 1);
    uniform_int_distribution<int> idx(1, n);
    for (int k = 0; k < n / 100; ++k)
    {
        int i = idx(rng), j = idx(rng);
        swap(arr[i], arr[j]);
    }
    return arr;
}

template <typename Heap>
void runTest(Heap &heap, const vector<int> &data,
             long long &totalCompares, long long &totalAssigns)
{
    heap.build(data);
    totalCompares = 0;
    totalAssigns = 0;
    while (heap.size() > 0)
    {
        long long beforeC = heap.getCompares();
        long long beforeA = heap.getAssigns();
        heap.extractMin();
        totalCompares += heap.getCompares() - beforeC;
        totalAssigns += heap.getAssigns() - beforeA;
    }
}

int main()
{
    vector<int> sizes = {1024, 4096, 16384, 65536, 262144, 1048576};
    vector<pair<string, function<vector<int>(int, int)>>> generators = {
        {"Random", generateRandom},
        {"Sorted", generateSorted},
        {"Reverse", generateReverse},
        {"Repeated", generateRepeated},
        {"AlmostHeap", generateAlmostHeap}};
    int repeats = 5;

    ofstream out("hhs_benchmark.csv");
    out << "Type,Size,Compares,Assigns\n";
```

 doi:10.20944/preprints202604.1122.v1


```
206
207     for (auto &genInfo : generators)
208     {
209         string typeName = genInfo.first;
210         auto &gen = genInfo.second;
211
212         for (int n : sizes)
213         {
214             double sumC = 0.0, sumA = 0.0;
215
216             for (int r = 0; r < repeats; ++r)
217             {
218                 auto data = gen(n, r);
219                 HHSHeap heap;
220                 long long c = 0, a = 0;
221                 runTest(heap, data, c, a);
222                 sumC += c;
223                 sumA += a;
224             }
225
226             double avgC = sumC / repeats;
227             double avgA = sumA / repeats;
228             out << typeName << "," << n << "," << avgC << "," << avgA << "\n";
229             cout << "HHSHeap " << typeName << " n=" << n << " done.\n";
230         }
231     }
232     out.close();
233     cout << "HHSHeap benchmark finished. Results saved to hhs_benchmark.csv\n"
        ;
234     return 0;
235 }
```

The implementation of the pure hole-based sink (HoleHeap) follows directly from the HHSHeap code in Appendix C by removing the `pref` array and all associated hint logic (i.e., simplifying the child selection to a direct comparison between left and right children without preference tracking). For brevity, the full listing is omitted but can be trivially derived.

References

1. R. W. Floyd. Algorithm 245: Treesort. *Communications of the ACM*, 7(12):701, 1964.
2. R. D. Dutton. Weak heaps: A family of efficient priority queues. *Software: Practice and Experience*, 23(6):659–678, 1993.
3. A. LaMarca and R. E. Ladner. The influence of caches on the performance of heaps. *ACM Journal of Experimental Algorithmics*, 1:4–es, 1996.
4. K. Parvizi. Adaptive cache-friendly priority queue: Enhancing heap-tree efficiency for modern computing. *arXiv preprint arXiv:2310.06663*, 2023.
5. I. H. Toroslu. Improving the Floyd-Warshall all pairs shortest paths algorithm. *arXiv preprint arXiv:2109.01872*, 2021.
6. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*, 3rd ed. MIT Press, 2009.