CSE 589
Applied Algorithms
Spring 1999
Cache Conscious Heapsort
Cache Conscious Static Search

Heapsort Williams 1964
We will sort the array A[0..n-1] in-place
Build a heap in-place
For i = n-1 to 1
A[i] := delete-max

Building a Heap
• Repeated Insertions
  – Worst case time O(n log n)
  – In practice O(n)
  – Good cache performance
• Floyd’s Method
  – Worst case time O(n)
  – Poor cache performance

Implicit Pointers of the Binary Heap

Floyd’s Method (1)
Analysis of Floyd's Method

- n/2 keys percolate down 0 levels
- n/4 keys percolate down 1 level
- n/8 keys percolate down 2 levels

- Total time is proportional to:
  \[ \sum_{i=1}^{k} \frac{n}{2^i} \]

  where \( k = \log_2 n \)

  \[ \sum_{i=1}^{k} \frac{n}{2^i} \leq n \sum_{i=1}^{k} \frac{1}{2^i} \leq 2n \]
Cache Performance of Build Heap

Repeated Insertions

Floyd's Method

Most recent insertion path

Next insertion

Most recent percolate down

Next percolate down

Spatial Locality of the Binary Heap

Repeated Insertions

Floyd's Method

Most recent percolate down

Next percolate down

Cache Conscious Heap

• Assume d keys fit in a cache line.
• d-heap - each node has d children.
• All children of a node are on the same cache line.

Computing the Children and Parents of an Aligned d-Heap

• Root in position d-1
• Everything is offset by d-1 to align children to cache lines
  – Children of i are at d(i - d + 2), d(i - d + 2) + 1, ..., d(i - d + 2) + d - 1
  – Parent of i is at i/d + d - 2
• Example d = 4
  – Children of 5 are 4(5 - 4 + 2) = 12, 13, 14, 15
  – Parent of 12 is 12/4 + 4 - 2 = 5

Delete-Max in CC Heap
Good Utilization of the 4-Heap

Four children of a node have this pattern:

- 4(3i) - 4(3i+1) - 4(3i+2) - 4(3i+3)

About $\log_4 3n$ misses per delete-max that percolates down to the leaves.

$log_4 3n$ is approximately $(\log_2 n) / 2$

Cache Performance of Heaps

- Traditional heapsort
- Cache conscious heapsort

Atom Cache Simulation
- 2MB L2 cache
- 32 Byte cache line
- 4 keys/cache line

Instruction Count for Heaps

- Traditional heapsort
- Cache conscious heapsort

Atom Simulation

- Number of keys
- Instructions per key

- Traditional heapsort
- Cache conscious heapsort

Height of a d-Heap

- Suppose we have $n$ nodes in a $d$-heap of height $k$

$$n = 1 + d + d^2 + \ldots + d^k = \frac{d^{k+1} - 1}{d - 1}$$

$$k = \log_d ((d - 1)n) - 1 = \log_d n$$

Time to Percolate Down in a d-Heap

- We percolate down $k$ levels where each step takes about $ad + b$ instructions for $d$ comparisons and a swap.

$$Instructions = (ad + b)k = (ad + b) \log_d n = \frac{ad + b}{\ln d} \ln n$$

Plot of Instructions for Varying $b$

- $b = 0, 1, 2$
- $a = 1$
**Execution Time for Heapsort**

![Graph showing execution time for heapsort on a logarithmic scale.](image1)

- Traditional heapsort vs. cache-conscious heapsort.
- Alpha 250, 2MB L2 cache, 32-byte cache line, 4 keys/cache line.

**Heapsort Notes**

- If \( d \) keys fit on a cache line, then moving to a \( d \)-heap gives better spatial locality and reduces cache misses.
- In addition, if \( d \) is small, a \( d \)-heap has fewer instructions than a binary heap.
- Heapsort is not usually the sorting algorithm of choice, but it has some advantages: in-place, \( O(n \log n) \) performance, no stack or recursion.

**Cache Misses for Sorting Algorithms**

![Graph showing cache misses for sorting algorithms.](image2)

- Cache-conscious versions of mergesort, heapsort, and quicksort.
- Atom Cache Simulation, 2MB L2 cache, 32-byte cache line, 4 keys/cache line.

**Instructions for Sorting Algorithms**

![Graph showing instructions for sorting algorithms.](image3)

- Cache-conscious versions of mergesort, heapsort, and quicksort.
- Atom Simulation.

**Execution Time for Sorting Algorithms**

![Graph showing execution time for sorting algorithms.](image4)

- Cache-conscious versions of mergesort, heapsort, and quicksort.
- Alpha 250, 2MB L2 cache, 32-byte cache line, 4 keys/cache line.

**Cache Conscious Static Search**

- Classic data structure for static search is a sorted array and binary search.

```plaintext
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
1 4 5 6 9 0 4 8 7 5 2 9 8 2 1 6
1 4 5 9 0 4 8 7 5 2 9 8 2 1 6
search for 30
```
Tree Structure for Binary Search

- Each node defines a search interval.
  - Root is \([0, n-1]\)
  - Node defining interval \([i, j]\) has children with intervals \([i, k-1]\) and \([k+1, j]\) where \(k = (i+j)/2\).
  - A node defining interval \([i, j]\) has key stored at index \((i+j)/2\).

Spatial Locality of Binary Search

- Binary search does not have good spatial locality.

Multi-way Static Search

- If \(B\) keys fit per cache line then we use a \((B+1)\)-way branching search tree.

Organization of Search Structure

- Implicit pointers are calculated.
- Data in array is organized to support these calculations.

Calculating the Pointers

- If the base address of a node is \(i\), its \(B+1\) children are at base addresses
  - \((B+1)i + B\), \((B+1)i + 2B\), ..., \((B+1)i + (B+1)B\)
Search Algorithm
• Do binary search within a node to find the key in the node or find the branch to take.

Average Cache Line Utilization
• In the case that a node is not in the cache what is its average utilization, that is, on average what percentage is actually used.

Utilization is Not Everything
• If the search structure is used repeatedly then the nodes near the root tend to be reused and reside in the cache.

Cache Misses Per Search

Accesses per Search

Cycles per Search