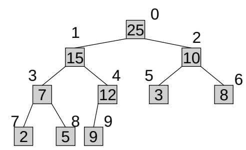
CSE 589
Applied Algorithms
Spring 1999

Cache Conscious Heapsort
Cache Conscious Static Search

Implicit Pointers of the Binary Heap



parent of i is (i-1)/2

children of i are 2i+1, 2i+2

0	1	2	3	4	5	6	7	8	9	10	11
25	15	10	7	12	3	8	2	5	9		

Heapsort
Williams 1964

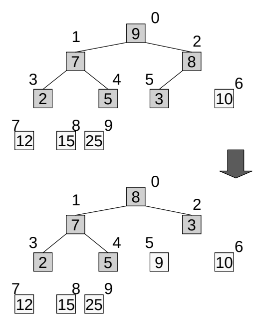
We will sort the array A[0..n-1] in-place
Build a heap in-place
For i = n-1 to 1
   A[i] := delete-max;

0	1	2	3	4	5	6	7	8	9
25	15	10	7	12	3	8	2	5	9
15	12	10	7	9	3	8	2	5	25
12	9	10	7	5	3	8	2	15	25
10	9	8	7	5	3	2	12	15	25
9	7	8	2	5	3	10	12	15	25

Invariants
Heap
Sorted

Heap ≤ Sorted

Logical Look at Heapsort



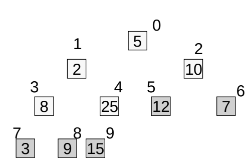
0	1	2	3	4	5	6	7	8	9
9	7	8	2	5	3	10	12	15	25

0	1	2	3	4	5	6	7	8	9
8	7	3	2	5	9	10	12	15	25

Building a Heap

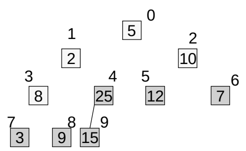
- Repeated Insertions
  - Worst case time O(n log n)
  - In practice O(n)
  - Good cache performance
- Floyd's Method
  - Worst case time O(n)
  - Poor cache performance

Floyd's Method (1)



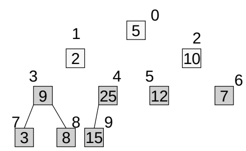
0	1	2	3	4	5	6	7	8	9
5	2	10	8	25	12	7	3	9	15

Floyd's Method (2)



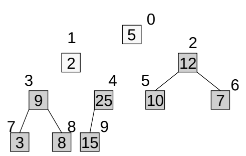
0	1	2	3	4	5	6	7	8	9
5	2	10	8	25	12	7	3	9	15

Floyd's Method (3)



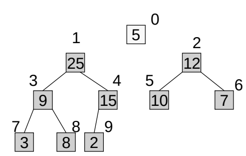
0	1	2	3	4	5	6	7	8	9
5	2	10	9	25	12	7	3	8	15

Floyd's Method (4)



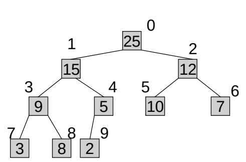
0	1	2	3	4	5	6	7	8	9
5	2	12	9	25	10	7	3	8	15

Floyd's Method (5)



0	1	2	3	4	5	6	7	8	9
5	25	12	9	15	10	7	3	8	2

Floyd's Method (6)



0	1	2	3	4	5	6	7	8	9
25	15	12	9	5	10	7	3	8	2

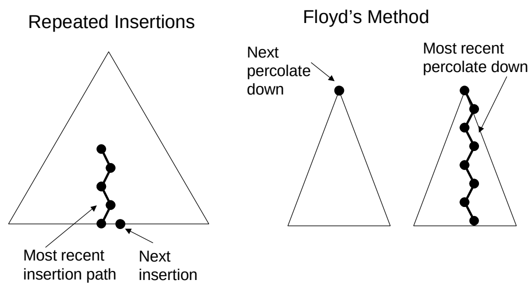
Analysis of Floyd's Method

- 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:

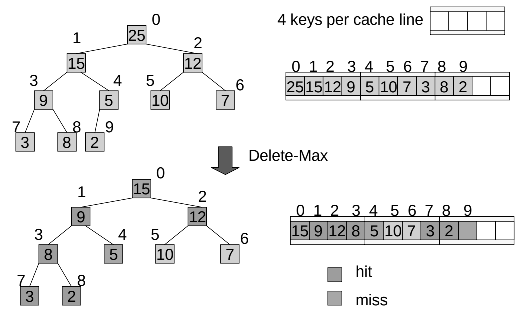
$$\sum_{i=1}^{k}(i-1)\frac{n}{2^i} \quad \text{where } k = \log_2 n$$

$$\sum_{i=1}^{k}(i-1)\frac{n}{2^i} \le n\sum_{i=1}^{\infty}(i-1)\frac{1}{2^i} \le 2n$$

Cache Performance of Build Heap

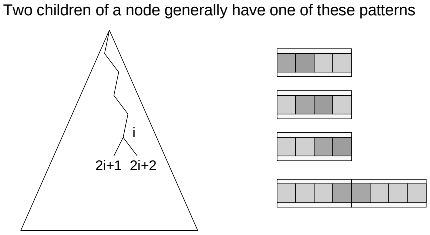


Spatial Locality of the Binary Heap



Poor Utilization of the Binary Heap

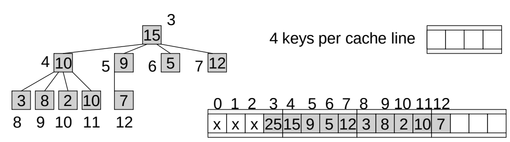
Two children of a node generally have one of these patterns



About $\log_2 n$ misses per delete-max that percolates down to the leaves.

Cache Conscious Heap

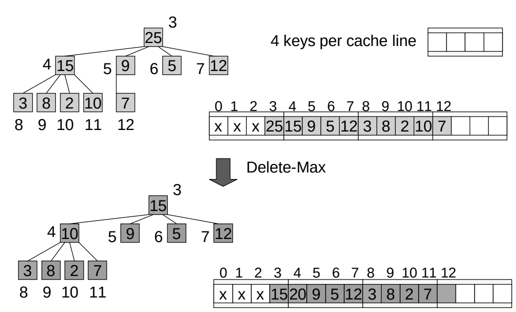
- Assume d keys fit in a cache line.
- d-heap - each node has d children.
- All children of a node are on the same cache line.



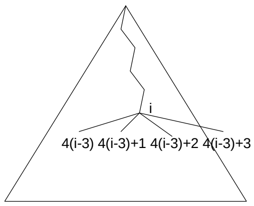
Computing the Children and Parents of an Aligned d-Heap

- 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

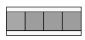
Delete-Max in CC Heap



Good Utilization of the 4-Heap



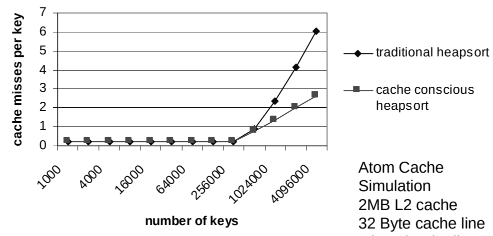
Four children of a node have this pattern



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

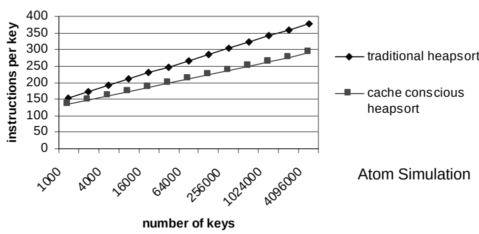
$\log_4 3n$ is approximately $(\log_2 n) / 2$

Cache Performance of Heaps



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

Instruction Count for Heaps



Atom Simulation

Height of a d-Heap

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

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

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

Time to Percolate Down in a d-Heap

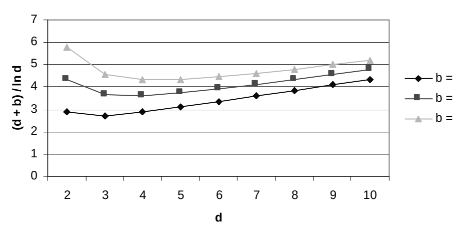
- We percolate down k levels where each step takes about $ad + b$ instructions for d comparisons and a swap.

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

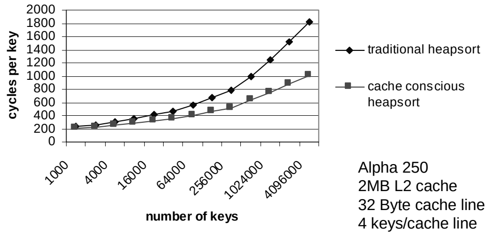
Plot of Instructions for Varying b

$$Insructions \approx \frac{ad + b}{\ln d} \ln n$$

a = 1
b = 0, 1, 2



Execution Time for Heapsort

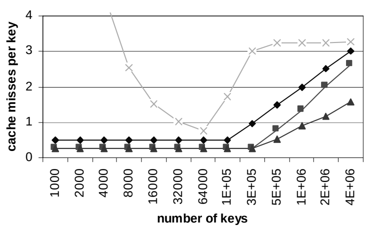


Alpha 250
2MB L2 cache
32 Byte cache line
4 keys/cache line

Heapsort Notes

- If d keys fit on a cache line then moving to a d-heap give better spatial locality and reduces cache misses.
- In addition, if d is small then 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.

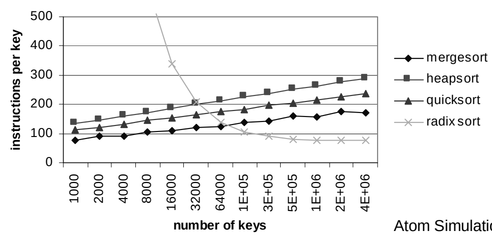
Cache Misses for Sorting Algorithms



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

Cache conscious versions of mergesort, heapsort, and quicksort

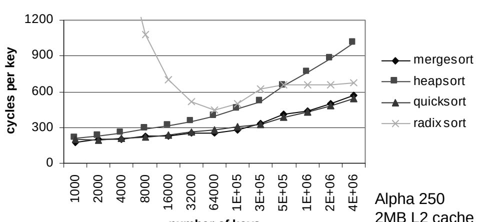
Instructions for Sorting Algorithms



Atom Simulation

Cache conscious versions of mergesort, heapsort, and quicksort

Execution Time for Sorting Algorithms



Alpha 250
2MB L2 cache
32 Byte cache line
4 keys/cache line

Cache conscious versions of mergesort, heapsort, and quicksort

Cache Conscious Static Search

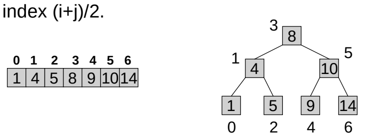
- Classic data structure for static search is a sorted array and binary search.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	4	5	8	9	10	14	**15**	17	20	25	26	27	29	30	31
1	4	5	8	9	10	14	15	17	20	25	**26**	27	29	30	31
1	4	5	8	9	10	14	15	17	20	25	26	27	**29**	30	31
1	4	5	8	9	10	14	15	17	20	25	26	27	29	**30**	31

search for 30

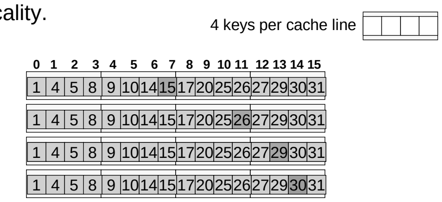
Tree Structure for Binary Search

- 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

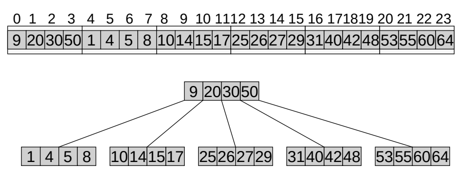
- Binary search does not have good spatial locality.



Cache line utilization is 1/4 = 25%.

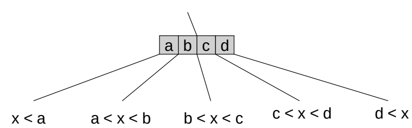
Multi-way Static Search

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



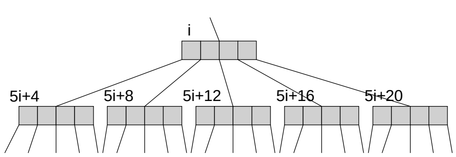
Organization of Search Structure

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

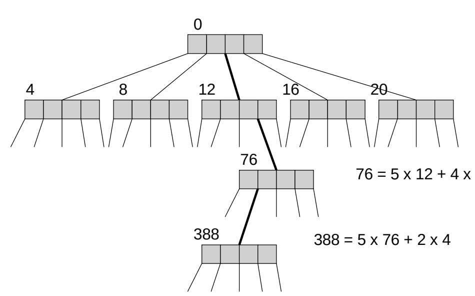


Calculating the Pointers

- 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

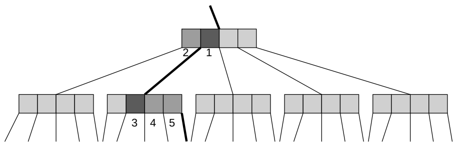


Example Search Path



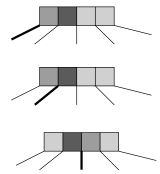
Search Algorithm

- Do binary search within a node to find the key in the node or find the branch to take.

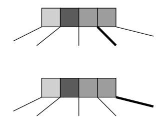


Average Cache Line Utilization

- 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.



$$\frac{3}{5}\cdot\frac{1}{2}$$

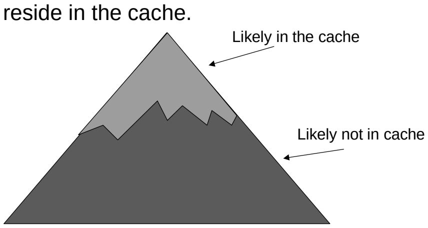


$$\frac{2}{5}\cdot\frac{3}{4}$$

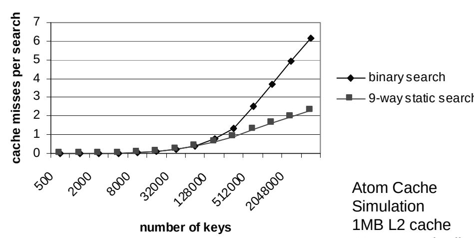
$$\frac{3}{5}\cdot\frac{1}{2}+\frac{2}{5}\cdot\frac{3}{4}=\frac{6}{10}=60\%$$

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.

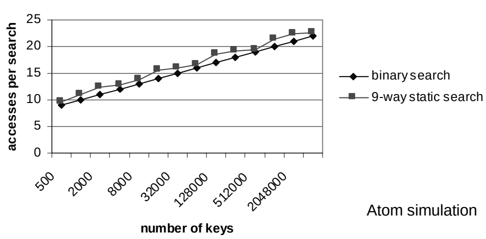


Cache Misses Per Search



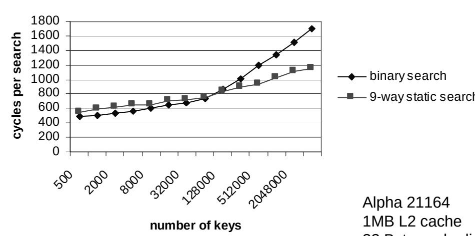
Atom Cache Simulation
1MB L2 cache
32 Byte cache line
8 keys/cache line

Accesses per Search



Atom simulation

Cycles per Search



Alpha 21164
1MB L2 cache
32 Byte cache line
8 keys/cache line