CISC 320 Introduction to Algorithms Fall 2005

Lecture 5

HeapSort

An idea to speedup Insertion Sort is, for a to-be-inserted key, use Binary search to find the proper position in the sorted segment.

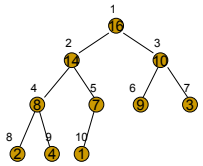
- When i-th element is to be inserted, the sorted segment has size (i-1), and binary search takes O(lg(i-1)) = O(lg(i) to find the right position. Therefore the total cost is

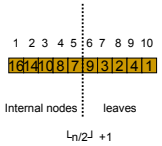
  $\Sigma_{i=1 \text{ to } n}$ O(lg(i)) = O(n lg n).
- But, how to actually insert an element into an array? Need to move other elements to their right, one by one, to fill the vacancy left by the to-be-inserted key. And it takes same number of shifting as does the original insertion sort.
- How about to store the sorted segment in a linked list, so inserting a new element will be easy? But, binary search does not work for a linked list.
- How about to use instead a binary search tree? Yes. But an ordinary BST has O(n) worst case time.
- Then a balanced BST, such as Red-black tree, will do it: in O(n lg n) time. This way, while we achieved lower bound in time, the algorithm is no longer "in-place".

Is there a data structure that help achieve both?

Heapsort

- Heap: a binary tree T that satisfies
  1. T is complete through depth h-1
  2. All paths to a leaf of depth h are to the left of all paths to a leaf of depth h-1, i.e., leaves at level h are filled from left to right.
  3. Key at any node is greater than or equal to the keys at each of its children (for maximizing heap). This is called heap property or partial order tree property.





Parent(i) = $\lfloor i/2 \rfloor$

Left(i) = 2i

Right(i) = 2i + 1

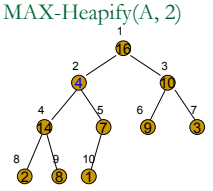
Maintain a Max-Heap

Left and right subtrees of i are already Max-Heaps
Make subtree rooted at i a Max-Heap
Max-Heapify(A, i)

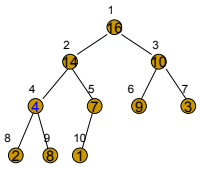
```
l ← left(i); // l = 2i
r ← right(i); // r = 2i+1
if ( l ≤ n and A[l] > A[i])
    then largest ← l;
    else largest ← i;
if (r ≤ n and A[r] > A[largest])
    then largest ← r;
if (largest ≠ i)
    then exchange A[i] ↔ A[largest];
        Max-Heapify(A, largest);
```

Analysis:
time is proportional to height of i
$\in$ O(lg n)

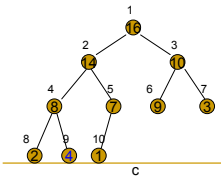
MAX-Heapify(A, 2)



a



b



c

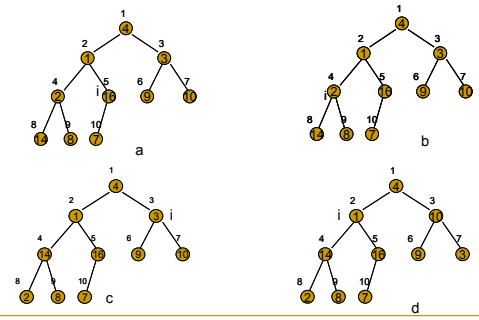
a. Exchange A[2] with A[4] and recursively call heapify(A,4)
b. Exchange A[4] with A[9] and recursively call heapify(A, 9)
c. Node 9 has no children, so it is done.

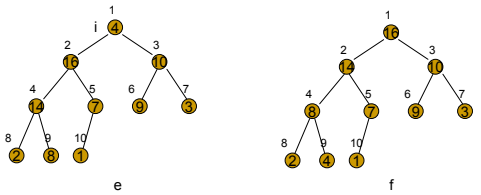
- Construct a Heap
  - Convert an array A[1..n] into a Max-Heap
  - Elements from n/2 +1 to n correspond to leaves, and themselves are 1-element heaps already.

Build-Max-Heap(A)

1. For (i= floor(n/2); i > 1; i--)
2. Max-Heapify(A, i);

	1	2	3	4	5	6	7	8	9	10
A	4	1	3	2	16	9	10	14	8	7





- Analysis of Build-Max-Heap

  n calls to heapify = n O(lg n) = O(n lg n)
- A tighter analysis

  $T(n) = T(n-r-1) + T(r) + 2 \lg(n)$

  left subtree, right subtree, heapify

  $= 2T((n-1)/2) + 2 \lg(n)$ // complete binary tree

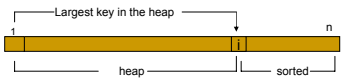
  $\in \Theta(n)$ // master theorem

Heapsort

Heapsort(A)

int heapsize = n;

1. **Build-Max-Heap(A);** // O(n)
2. **for(i = heapsize; i > 2; i--)** // n
3. **do exchange A[1] ↔ A[i];** // O(1)
4. **heapsize = heapsize – 1;** // O(1)
5. **Max-Heapify(A, 1);** // O(lg n)



Time analysis:

T(n) = O(n) + O(n lg n) = O(n lg n)

- A more exact analysis

$$T(n) = \Theta(n) + 2 \sum_{i=1 \text{ to } n-1} \lfloor \lg i \rfloor$$
$$\leq \Theta(n) + 2 \int_1^n (\lg e) \ln x \, dx$$
$$= \Theta(n) + 2 (\lg e)(n \ln n - n)$$
$$= \Theta(n) + 2(n \lg(n) - 1.443\, n)$$
$$= 2\, n \lg(n) + O(n)$$

- Heap used as priority queue
  - Max-Heap-Insert(S,x) inserts element x into the set S (time:)
  - Heap-Maximum(S) returns the element of S with the largest key (time:)
  - Heap-Extract-Max(S) removes and returns the element of S with the largest key (time:)
- A heap can support any priority-queue operations on a set of size n in _ time.

Heap-extract-max(A)

1. **If heap-size(A) < 1**
2. **then error "heap underflow"**
3. **max = A[1];**
4. **A[1] = A[heap-size(A)]**
5. **Heap-size(A) = heap-size(A) -1**
6. **Max-Heapify(A, 1)**
7. **Return max;**

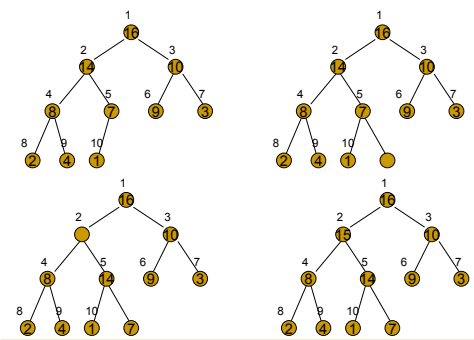
Note: Heap-extract-max takes only O(lg n) time

Max-Heap-Insert(A, key)

1. **heap-size(A) = heap-size(A) + 1**
2. **i = heap-size(A)**
3. **while (i > 1 and A[parent(i)] < key)**
4. **exchange A[i] ↔ A[parent(i)]**
5. **i = parent(i)**

Note: this is bubble-up heap, only requires one comparison at each level to float a big key to its right position (in contrast to heapify which requires two comparisons to filter down a small key)

Max-Heap-Insert(A, 15)

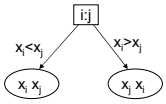


Comparison of sorting algorithms

Algorithm	Worst-case	Average	Space usage
Insertionsort	$n^2/2$	$\Theta(n^2)$	in place
Quicksort	$n^2/2$	$\Theta(n \log n)$	extra space log n
Mergesort	n lg n	$\Theta(n \log n)$	extra space n
Heapsort	2n lg n	$\Theta(n \log n)$	in place
Accl. Heapsort	n lg n	$\Theta(n \log n)$	in place

Lower bounds: worst-case

- Decision tree approach
  nodes ↔ comparisons of keys
  leaves ↔ possible permutation of n keys (=n!)



Height ↔ max # of comparisons
$\geq$ lg (# of leaves) = lg(n!)$\geq$ (n/2)lg(n/2)
$\in\Theta$(n log n)

Lower bounds

Average Case

- $T_{av}(n)$ = (sum of lengths of all paths from the root to a leaf) / (L, # of leaves)
- Balanced decision tree => lower $T_{av}(n)$
- A complete tree is most balanced: [L lg(L)] /L
- Therefore,

$$T_{av}(n) \geq \lg(L) = \lg(n!) \in \Theta(n \log n)$$