

CSC 8301- Design and Analysis of Algorithms

Lecture 8

Transform and Conquer II Algorithm Design Technique

Transform and Conquer

This group of techniques solves a problem by a *transformation*

- to a simpler/more convenient instance of the same problem (*instance simplification*)
- to a different representation of the same instance (*representation change*)
- to a different problem for which an algorithm is already available (*problem reduction*)

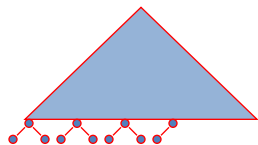
Representation Change

- Search Trees (binary, AVL, 2-3, 2-3-4, B-trees)
- **Heaps**
- **Horner's rule for polynomial evaluation**
- **Computing a^n (binary exponentiation)**

Heaps and Heapsort

Definition A *heap* is a binary tree with keys at its nodes (one key per node) such that:

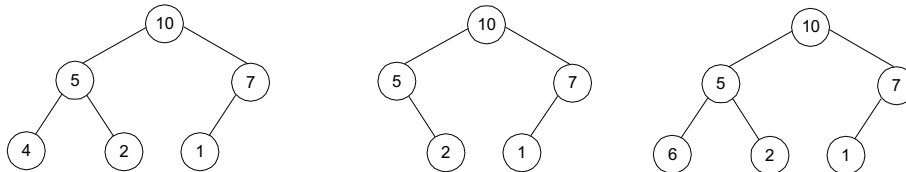
- It is essentially complete, i.e., all its levels are full except possibly the last level, where only some rightmost keys may be missing (*shape property*)



- The key at each node is \geq keys at its children (*heap property*)

Illustration of the heap's definition

Which of the following is a heap?



Note: Heap's elements are ordered top down (along any path down from its root), but they are not ordered left to right

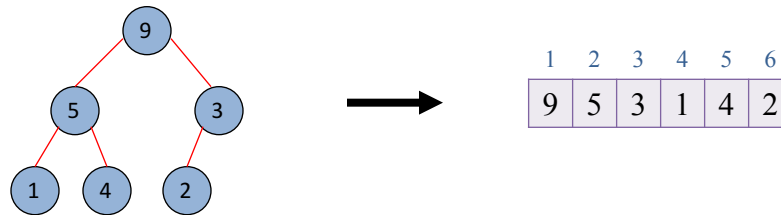
Some Important Properties of a Heap

- The root contains the largest key
- The subtree rooted at any node of a heap is also a heap
- A heap can be represented as an array
 - most important property that distinguishes it from other binary search trees; facilitated by the left-to-right fill at each level

Heap's Array Representation

Store heap's elements in an array (whose elements indexed, for convenience, 1 to n) in top-down left-to-right order

Example:



- Left child of node at index j is at index _____
- Right child of node at index j is at index _____
- Parent of node at index j is at index _____
- Parental nodes are represented in the first $\lfloor n/2 \rfloor$ locations

Heap Construction (bottom-up)

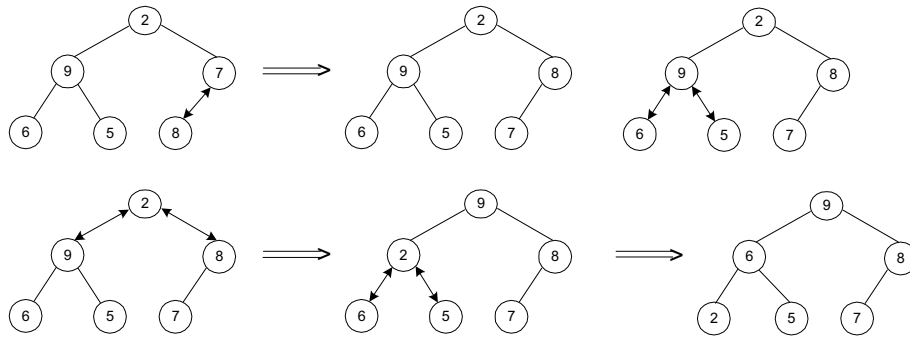
Step 0: Initialize the structure with keys in the order given

Step 1: (Heapify) Starting with the last (rightmost) parental node, fix the heap rooted at it, if it doesn't satisfy the heap condition: keep exchanging it with its largest child until the heap condition holds

Step 2: Repeat Step 1 for the preceding parental node

Example of Heap Construction (bottom-up)

Construct a heap for the list 2, 9, 7, 6, 5, 8

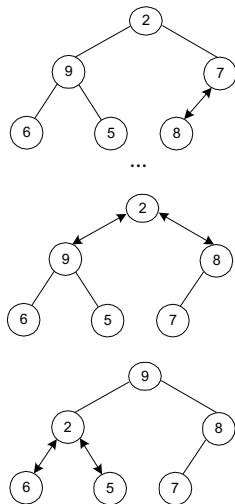


- Algorithm on heaps are easier to understand if we think of heaps as binary trees
- But their actual implementation is much simpler and more efficient with arrays

Bottom-up Heap Construction Algorithm

Construct a heap for the list

1	2	3	4	5	6	
2	9	7	6	5	8	H



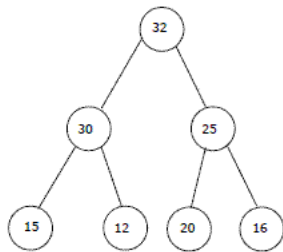
Pseudocode of Bottom-up Heap Construction

```

Algorithm HeapBottomUp( $H[1..n]$ )
//Constructs a heap from the elements of a given array
// by the bottom-up algorithm
//Input: An array  $H[1..n]$  of orderable items
//Output: A heap  $H[1..n]$ 
for  $i \leftarrow \lfloor n/2 \rfloor$  downto 1 do
     $k \leftarrow i$ ;  $v \leftarrow H[k]$ 
     $heap \leftarrow \text{false}$ 
    while not  $heap$  and  $2 * k \leq n$  do
         $j \leftarrow 2 * k$ 
        if  $j < n$  //there are two children
            if  $H[j] < H[j + 1]$   $j \leftarrow j + 1$ 
        if  $v \geq H[j]$ 
             $heap \leftarrow \text{true}$ 
        else  $H[k] \leftarrow H[j]$ ;  $k \leftarrow j$ 
     $H[k] \leftarrow v$ 

```

Efficiency of Bottom-up Heap Constr. Algorithm



- Assume $n=2^k-1$, so the heap is full
- Height is $h =$ _____
- A key on level i (top level is 0) may need to travel to level _____ in the worst case
- Maximum number of levels traversed by key on level i is _____
- Moving a key to the next level down requires _____ comparisons
- Maximum number of comparisons for key on level i is _____
- Number of keys on level i is _____
- Total number of key comparisons is $C(n) =$ _____

Heapsort

Stage 1: Construct a heap for a given list of n keys

Stage 2: Repeat operation of root removal $n-1$ times:

- Exchange keys in the root and in the last (rightmost) leaf
- Decrease heap size by 1
- “Heapify” the tree: if necessary, swap new root with larger child until the heap condition holds

Example of Sorting by Heapsort

Sort the list 2, 9, 7, 6, 5, 8 by heapsort

Stage 1 (heap construction)

2 9 7 6 5 8

1	2	3	4	5	6
2	9	7	6	5	8

Example of Sorting by Heapsort

Sort the list 2, 9, 7, 6, 5, 8

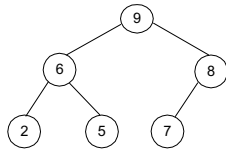
1	2	3	4	5	6
9	6	8	2	5	7

Stage 1 (heap construction)

2 9 7 6 5 8
 2 9 8 6 5 7
2 9 8 6 5 7
 9 2 8 6 5 7
 9 6 8 2 5 7

Stage 2 (root/max removal)

9 6 8 2 5 7



Analysis of Heapsort

Stage 1: Build heap for a given list of n keys

Worst-case:

$$C(n) =$$

Stage 2: Repeat operation of root removal $n-1$ times (fix heap)

Worst-case:

$$C(n) =$$

Analysis of Heapsort

Stage 1: Build heap for a given list of n keys

Worst-case:

$$C(n) = \sum_{i=0}^{h-1} \underbrace{2^{h-i}}_{\substack{\text{\# nodes at} \\ \text{level } i}} 2^i = 2(n - \log_2(n+1)) \in \Theta(n)$$

Stage 2: Repeat operation of root removal $n-1$ times (fix heap)

Worst-case:

$$C(n) = \sum_{i=1}^{n-1} 2 \log_2 i \in \Theta(n \log n)$$

Both worst-case and average-case efficiency: $\Theta(n \log n)$

In-place: yes

Stability: no (e.g., 1 1)

Review of Major Sorting Algorithms

	Selection sort	Insertion sort	Mergesort	Quicksort	Heapsort
strategy					
worst time					
avg. time					
in-place					
stability					

Priority Queue

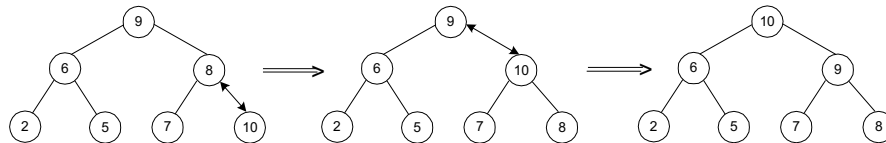
A *priority queue* is the ADT of a set of elements with numerical priorities with the following operations:

- find element with highest priority
 - delete element with highest priority
 - insert element with assigned priority (see next slide)
- Heap is a very efficient way for implementing priority queues
 - Two ways to handle priority queue in which highest priority = smallest number

Insertion of a New Element into a Heap

- Insert the new element at last position in heap.
- Compare it with its parent and, if it violates heap condition, exchange them
- Continue comparing the new element with nodes up the tree until the heap condition is satisfied

Example: Insert key 10



Efficiency: $O(\log n)$

Representation Change

- Search Trees (binary, AVL, 2-3, 2-3-4, B-trees)
- Heaps
- **Horner's rule for polynomial evaluation**
- **Computing a^n (binary exponentiation)**

Horner's Rule For Polynomial Evaluation

Given a polynomial of degree n

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$$

and a specific value of x , find the value of p at that point.

Two brute-force algorithms:

```

p ← 0
for i ← n downto 0 do
  power ← 1
  for j ← 1 to i do
    power ← power * x
  p ← p + ai * power
return p

```

```

p ← a0; power ← 1
for i ← 1 to n do
  power ← power * x
  p ← p + ai * power
return p

```

Horner's Rule

$$\text{Example: } p(x) = 2x^4 - x^3 + 3x^2 + x - 5$$

Horner's Rule

$$\begin{aligned} \text{Example: } p(x) &= 2x^4 - x^3 + 3x^2 + x - 5 = \\ &= x(2x^3 - x^2 + 3x + 1) - 5 = \\ &= x(x(2x^2 - x + 3) + 1) - 5 = \\ &= x(x(x(2x - 1) + 3) + 1) - 5 \end{aligned}$$

Substitution into the last formula leads to a faster algorithm

Same sequence of computations are obtained by simply arranging the coefficients in a table and proceeding as follows:

$$\begin{array}{cccccc} \text{coefficients} & & 2 & -1 & 3 & 1 & -5 \\ & x=3 & & & & & \end{array}$$

Horner's Rule Pseudocode

ALGORITHM *Horner*($P[0..n], x$)
 //Evaluates a polynomial at a given point by Horner's rule
 //Input: An array $P[0..n]$ of coefficients of a polynomial of degree n
 // (stored from the lowest to the highest) and a number x
 //Output: The value of the polynomial at x
 $p \leftarrow P[n]$
for $i \leftarrow n - 1$ **downto** 0 **do**
 $p \leftarrow x * p + P[i]$
return p

Efficiency of Horner's Rule: # multiplications = # additions = n

Byproduct – *synthetic division* of $p(x)$ by $(x-x_0)$:

□ Intermediate values are coefficients of the quotient of $p(x):(x-x_0)$

Computing a^n (revisited)

Left-to-right binary exponentiation

Initialize product accumulator by 1.

Scan n 's binary expansion from left to right and do the following:

- If the current binary digit is 0, square the accumulator (S)
- If it is 1, square the accumulator and multiply it by a (SM)

Example: Compute a^{13} . Here, $n = 13 = 1101_2$.

Binary rep. of 13: 1 1 0 1

Accumulator: 1 _____ _____ _____ _____

(computed left-to-right)

Efficiency: min $M(n)$ _____

 max $M(n)$ _____

Computing a^n (cont.)

Right-to-left binary exponentiation

Scan n 's binary expansion from right to left and compute a^n as the product of terms a^{2^i} corresponding to 1's in this expansion.

Example Compute a^{13} by the right-to-left binary exponentiation. Here, $n = 13 = 1101_2$.

$$\begin{array}{ccccccc}
 1 & & 1 & & 0 & & 1 \\
 a^8 & & a^4 & & a^2 & & a \\
 a^8 & * & a^4 & * & & & a \\
 & & & & & & : \text{ product}
 \end{array}
 \quad : \quad a^{2^i} \text{ terms}$$

(computed right-to-left)

Pseudocode for computing a^n (right to left)

Example for $n = 13 = 1101_2$:

$$\begin{array}{ccccccc}
 1 & & 1 & & 0 & & 1 \\
 a^8 & * & a^4 & * & & & a \\
 & & & & & & : \text{ product}
 \end{array}$$

Pseudocode:

Efficiency: same as that of left-to-right binary exponentiation

Transform and Conquer

Problem Reduction

Problem Reduction

This variation of transform-and-conquer solves a problem by transforming it into different problem for which an algorithm is already available.

To be of practical value, the combined time of the transformation and solving the other problem should be smaller than solving the problem as given by another method.

Examples of Solving Problems by Reduction

- Computing $\text{lcm}(m, n)$ via computing $\text{gcd}(m, n)$
- Counting number of paths of length n in a graph by raising the graph's adjacency matrix to the n -th power
- Transforming a maximization problem to a minimization problem and vice versa (also, min-heap construction)
- Linear programming
- Reduction to graph problems (e.g., solving puzzles via state-space graphs)

Homework

Exercises 6.4: 1, 3, 6, 7, 8

Exercises 6.5: 4, 7, 9

Exercises 6.6: 2, 9

Reading:

- Sec. 6.4, 6.5, and 6.6