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*)

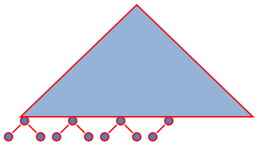
Representation Change

- 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:

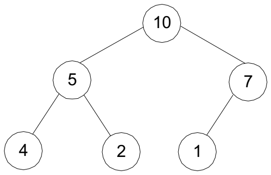
- It is essentially complete, i.e., all its levels are full except possibly the last level, where only some rightmost keys may be missing (*shape property*)



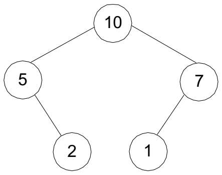
- The key at each node is ≥ keys at its children (*heap property*)

Illustration of the heap’s definition

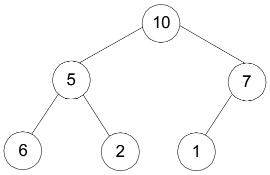
Which of the following is a heap?



Yes



No (shape fails)



No (heap fails)

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

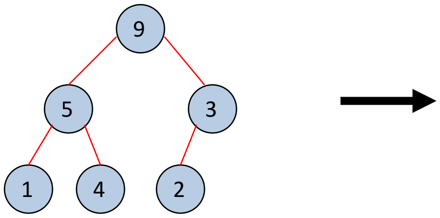
Some Important Properties of a Heap

- 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:



1	2	3	4	5	6
9	5	3	1	4	2

- Left child of node at index j is at index $2j$
- Right child of node at index j is at index $2j+1$
- Parent of node at index j is at index $\lfloor j/2 \rfloor$
- Parental nodes are represented in the first $\lfloor n/2 \rfloor$ locations

Heap Construction (bottom-up)

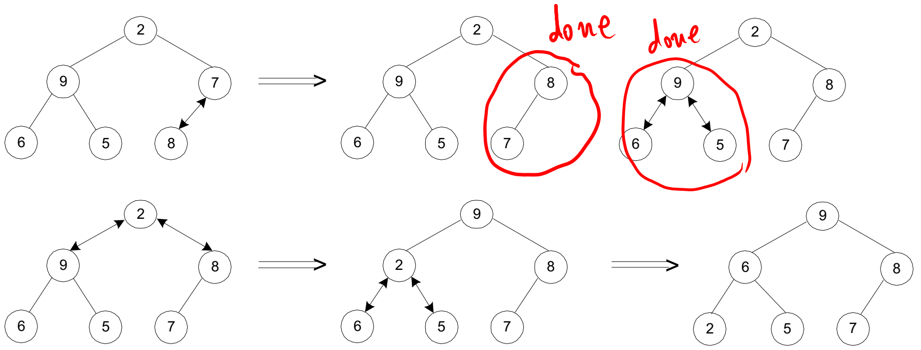
Step 0: Initialize the structure with keys in the order given

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)

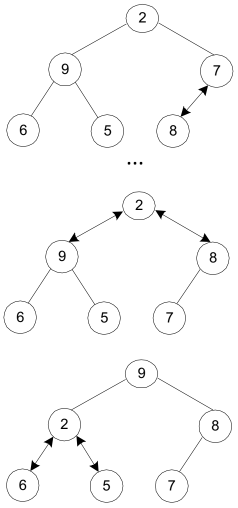
Construct a heap for the list 2, 9, 7, 6, 5, 8



- 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



=6/2

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

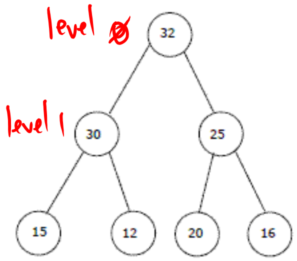
index k → children 2k, 2k+1

```
Heapify (k)
    j = 2k   /* index of left child */
    if (j > n) return   /* leaf */
    if (j < n) and H[j] < H[j+1] j++;
    if (A[k] < A[j])
        swap (A[k], A[j]);
        Heapify (j);
for k = ⌊n/2⌋ downto 1
    Heapify (k);
```

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          /* i = index of parental nodes/
    $k \leftarrow i$;  $v \leftarrow H[k]$                   /* k = index of descendant
    heap ← false                                       of i that needs to
    while not heap and $2 * k \le n$ do                     be heapified */
        $j \leftarrow 2 * k$
        if $j < n$   //there are two children         /* j = index of
            if $H[j] < H[j+1]$   $j \leftarrow j + 1$          the larger of the
        if $v \ge H[j]$                                  children of k */
            heap ← 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 = ___K-1___
- ❑ A key on level i (top level is 0) may need to travel to level ___h___ in the worst case
- ❑ Maximum number of levels traversed by key on level i is ___h-i___

- ❑ Moving a key to the next level down requires ___2___ comparisons
- ❑ Maximum number of comparisons for key on level i is ___2(h-i)___
- ❑ Number of keys on level i is ___2^i___

- ❑ Total number of key comparisons is $C(n) = \sum_{i=0}^{h-1} 2^i * 2(h-i) < \boxed{2n}$

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

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)

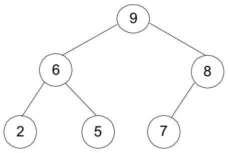
2 9 $\underline{7}$ 6 5 8

2 $\underline{9}$ 8 6 5 7

$\underline{2}$ 9 8 6 5 7

9 $\underline{2}$ 8 6 5 7

9 6 8 2 5 7



Stage 2 (root/max removal)

$\underline{9}$ 6 8 2 5 7

$\underbrace{7\ 6\ 8\ 2\ 5}_{\text{Heapify}}\ \boxed{9}$ in place

$8\ 6\ \boxed{7}\ 2\ 5$

$5\ 6\ 7\ 2\ \boxed{8}$ in place

$7\ 6\ 5\ 2$

$2\ 6\ 5\ \boxed{7}$ in place

$6\ 2\ 5$

$5\ 2\ \boxed{6}$ in place

$2\ \boxed{5}$ in place

Analysis of Heapsort

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

Worst-case:

$$C(n) = \sum_{i=0}^{h-1} 2(h-i) \ast 2^i < 2n = \Theta(n)$$

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

Worst-case:

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

$$= \Theta(n \log n)$$

After 1st removal: $\overbrace{\log_2(n-1) \ast 2}^{\#\ \text{comparison}}$

2nd removal: $2 \log_2 (n-2)$

3rd removal: $2 \log_2 (n-3)$

Analysis of Heapsort

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

Worst-case:

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

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

Worst-case:

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

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

In-place: $\boxed{\text{yes}}$ (advantage over mergesort)

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

Review of Major Sorting Algorithms

	Selection sort	Insertion sort	Mergesort	Quicksort	Heapsort
strategy	find min, swap, recurse	sort (n-1), insert nth	split in half, sort, merge	split based on pivot, recurse	B
worst time	$O(n^2)$	$O(n^2)$	$O(n \log n)$	$O(n^2)$	E
avg. time	$O(n^2)$	$O(n^2)$	$O(n \log n)$	$O(n \log n)$	S
in-place	yes	yes	NO	yes	T
stability	yes	yes	yes	no	

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)

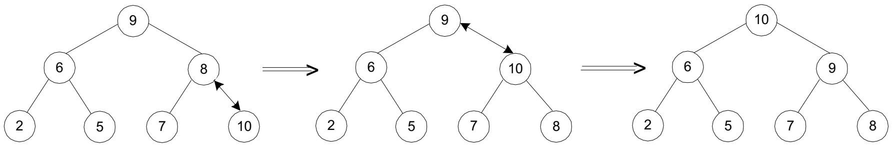
- Heap is a very efficient way for implementing priority queues

- 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

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

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

Two brute-force algorithms:

```
$p \leftarrow 0$
for $i \leftarrow n$ downto 0 do
    $power \leftarrow 1$
    for $j \leftarrow 1$ to $i$ do
        $power \leftarrow power * x$
    $p \leftarrow p + a_i * power$
return $p$
```

$\Theta(n^2)$

```
$p \leftarrow a_0$;  $power \leftarrow 1$
for $i \leftarrow 1$ to $n$ do
    $power \leftarrow power * x$
    $p \leftarrow p + a_i * power$
return $p$
```

$2n = \Theta(n)$

Counting # multiplications

Horner’ s Rule

Example: $p(x) = 2x^4 - x^3 + 3x^2 + x - 5$

$$= x(\underbrace{2x^3 - x^2 + 3x + 1}_{\text{recurse}}) - 5$$

$$= x(x\underbrace{(2x^2 - x + 3)}_{\text{recurse}} + 1) - 5$$

$$= x(x(x(2x-1)+3)+1) - 5$$

Horner’s Rule

Example: $p(x) = 2x^4 - x^3 + 3x^2 + x - 5 =$

$= x(2x^3 - x^2 + 3x + 1) - 5 =$

$= x(x(2x^2 - x + 3) + 1) - 5 =$

$= x(x(x(2x - 1) + 3) + 1) - 5$

Substitution into the last formula leads to a faster algorithm

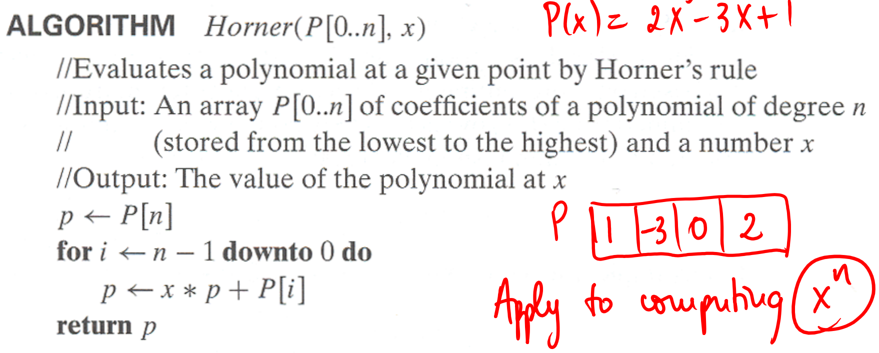
Same sequence of computations are obtained by simply arranging the coefficients in a table and proceeding as follows:

coefficients	2	-1	3	1	-5
x=3	2	$\underbrace{5}_{2 * 3 - 1}$	$\underbrace{18}_{5 * 3 + 3}$	55	$\boxed{160} \longrightarrow p(3)$

$p : (x-3)$

$= 2x^3 + 5x^2 + 18x + 55$

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 $p$
```

$P(x) = 2x^3 - 3x + 1$

P	1	-3	0	2

Apply to computing $\boxed{x^n}$

Efficiency of Horner's Rule: # multiplications = # additions = $\boxed{n}$

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, $n = 13 = 1101_2$.

Always same, don't count 'm efficiency

Binary rep. of 13:	1	1	0	1
	SM	SM	S	SM
Accumulator: 1 (computed left-to-right)	a	a^3	a^6	$\boxed{a^{13}}$

Efficiency: min M(n) $b-1$

max M(n) $2(b-1)$

$b = \#$ bits
$= \lfloor \log_2 n \rfloor + 1$

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, $n = 13 = 1101_2$.

$$a^{(2^3+2^2+2^0)} = a^{2^3} * a^{2^2} * a^{2^0}$$

$$= a^8 * a^4 * a^1$$

1		1		0	1		
a^8		a^4		a^2	a	:	a^{2^i} terms
a^8	*	a^4	*		a	:	product

(computed right-to-left)

Pseudocode for computing a^n (right to left)

Example for $n = 13 = 1101_2$:

1		1		0	1		
a^8	*	a^4	*		$\boxed{a}$	:	product

Pseudocode:

```
term = a
product = 1
for i = 0 to $\lfloor \log_2 n \rfloor$
    if ($bit_i$ = 1)
        product *= term
    term = term * term
```

term = a
Square term moving left

product *= term

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.

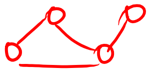
To be of practical value, the combined time of the transformation and solving the other problem should be smaller than solving the problem as given by another method.

Examples of Solving Problems by Reduction

- Computing lcm(*m*, *n*) via computing gcd(*m, n*)

  lcm(m,n) = m*n / 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