8 Transform and Conquer

Tuesday, November 19, 2024 10:57 AM

Problem Solving technique.

2 steps: 1) transform the input.

2) Solve the problem

Transform the input/problem to a more convenient form or shape.

E.G. Presorting.

Problem #1

Check whether all elements in a list are unique.

- Brute-Force.

(python)
```
def areUnique( l )
    for x in l:
        k = l.index(x)
        for y in l[k+1:] :
            if x == y :
                return False
    return true
```

NOTE:

all functions have a cost.

for x in l is nice, but not always.

Analysis: basic operation: ==

worst case: every element is unique.

$$C_{worst}(n) = (n-1) + (n-2) + (n-3) + \dots + 1$$

$$= \frac{(n-1)n}{2} \in \Theta(n^2)$$

$$C_{worst}(n) = \frac{(n-1)n}{2} \in \Theta(n^2)$$

- Presort. (Transform the input)
  - sort
  - check that no two consecutive elements are the same

```
FUNCTION areUnique( l[0...n-1])
    sort(l)                              } sort
    for i<-0 to n-2 do
        if l[i] == l[i+1] then           } scan
            return false
    return true.
```

Analysis:- basic operation ==

Worst case:- all elements are unique

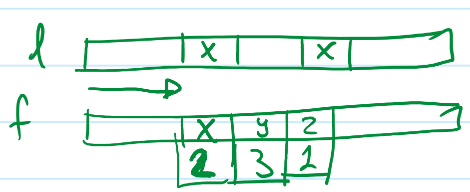
$$C_{worst}(n) = C_{sort}(n) + C_{scan}(n)$$
$$= \Theta(n \log n) + n-1$$
$$= \Theta(n \log n) + \Theta(n)$$
$$= \Theta(n \log n)$$

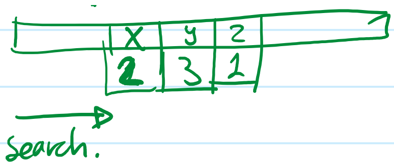
- Problem #2: Find the "mode" in a list.

mode: the value that occurs the most

try 1: Brute-force

- Need a list of items and their frequency
- for every element x in the input
  - find the frequency of x and increase it by 1
  - if x is not in the frequency list, add it with value 1
- find value with highest frequency.



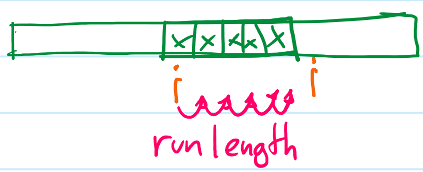
f 

Worst case: all elements are unique. $1+2+3+4+\dots+n-1 \in \Theta(n^2)$

try 2: presort!!

Sorted l 

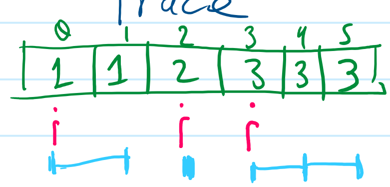
- look for the largest cluster:




```
FUNCTION PresortMode( l[0..n-1] )
    sort (l)
    mode freq ← 0
    mode value

    i ← 0
    while i ≤ n-1 do
        runlenght ← 1
        run value ← l[i]
        while i + runlenght ≤ n-1 and l[i+runlenght] = run value do
            runlenght ← runlenght + 1
        If runlenght > mode freq then
            Mode freq ← runlenght
            mode vale ← run value
        i ← i + runlenght
    return mode value.
```

trace



mode
~~1~~, ~~2~~
3, 3

Analysis basic operation.

$$C_{worst}(n) = C_{sort}(n) + C_{scan}(n)$$
$$= \Theta(n \log n) + n-1$$
$$= \Theta(n \log n) + \Theta(n)$$

$$= \Theta(n \log n) + n-2$$
$$= \Theta(n \log n) + \Theta(n)$$
$$= \Theta(n \log n)$$

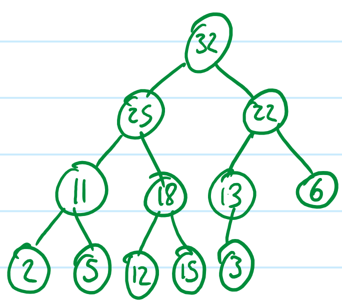
Heaps & HeapSort

What is a heap?

- A binary tree.
- Shape :- every level except the last one is complete
  the last level is filled from left to right.
- heap :- every node is greater* than both it's children

* maxheap.

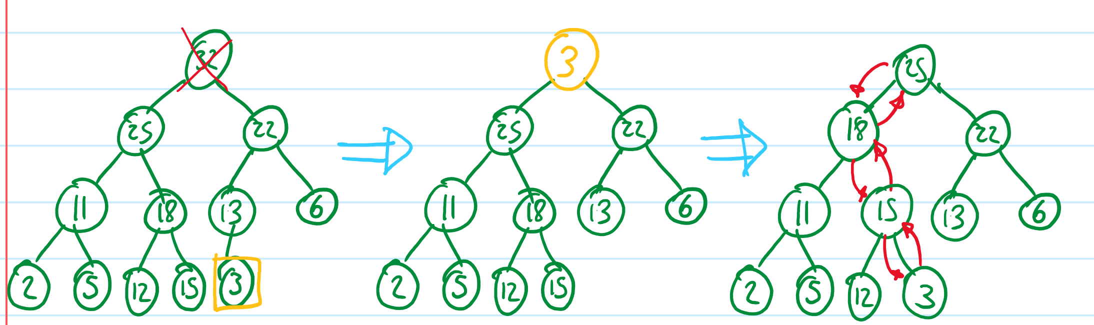
Note: nothing is said about siblings



Widely used to implement "priority Queues"

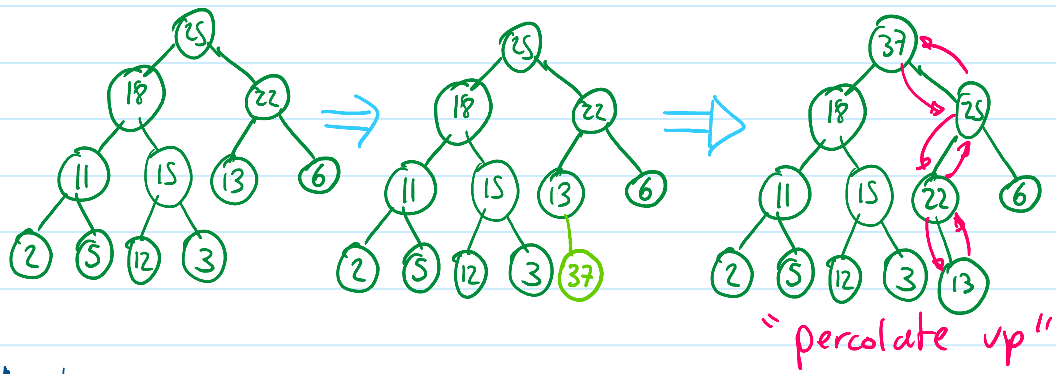
Operations
- find the greatest element. → the root
- delete element with greatest value
- add a new element to the heap

- Delete greatest



"percolate down"

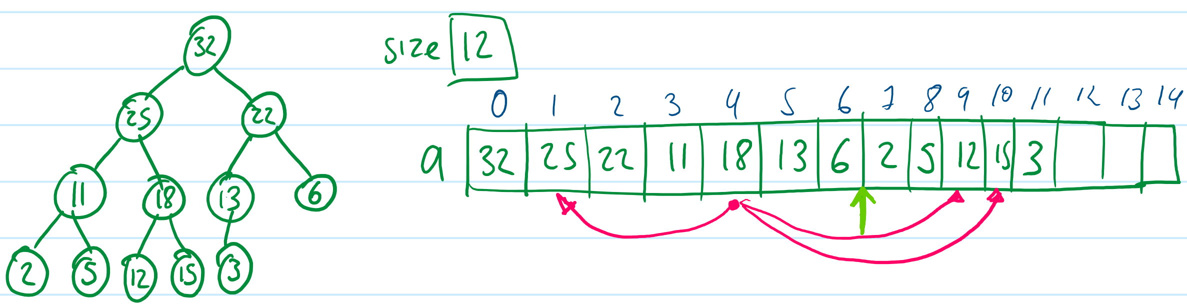
- Add an element



Analysis:

- A tree with n nodes has a height of $\log n$
- Both "Add Element" "Delete greatest" require $\Theta(\log n)$

- Implementation of a Heap.



Note:
- nodes in locations $0 .. \lfloor \frac{n}{2} \rfloor$ are internal nodes
- nodes in locations $\lfloor \frac{n}{2} \rfloor + 1$ to $n-1$ are the bottom layer
- the parent of node at i is in $\lfloor \frac{i-1}{2} \rfloor$
- the children of node at i are at positions $2i+1$ and $2i+2$

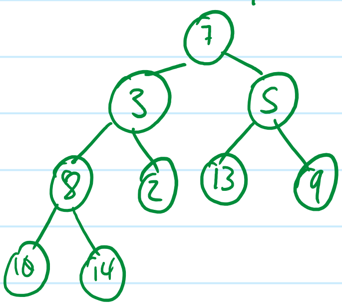
- HEAPSORT

To sort an list

1.- Transform the list into a heap
2.- Use the "find greatest" and "delete greatest" operations to sort the array from back-to-front

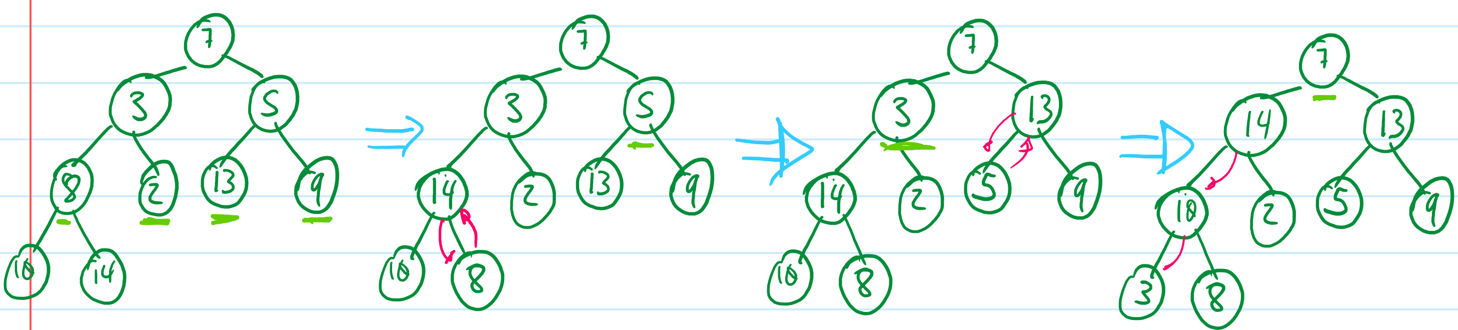
#1:- transform:

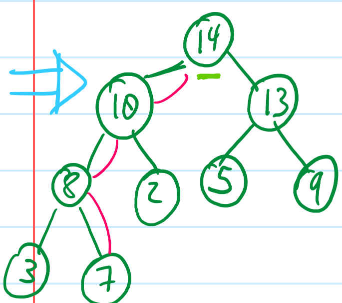
a									
	7	3	5	8	2	13	9	10	14



How can we "heapify" the array?

- for each intermediate node x percolate x down to it's appropiate place.





a									
	14	10	13	8	2	5	9	3	7

```
FUNCTION heapify ( a[0..n-1] )
    FOR i ← ⌊n/2⌋ DOWNTO 0 DO
        k ← i
        v ← a[k]
        done ← false
                                      while node k has children.
        WHILE not done and 2*k+1 < n DO
            best_child ← 2*k+1
            IF 2*k+2 < n THEN
                IF a[best_child] < a[best_child+1] THEN
                    best_child ← best_child+1

            IF v ≥ a[best_child] THEN
                done ← true
            ELSE
```

```
            done ← true
        ELSE
            a[k] ← a[best_child]
            k ← best_child
    a[k] ← v
```

Analysis:

$$C(n) = \frac{n}{2} \cdot \log n \quad \text{is} \quad \Theta(n \cdot \log n)$$

#2 Sort

```
FUNCTION heapSort(a[0..n-1])
    heapify(a)
    FOR i ← n-1 DOWNTO 1 DO
        v ← getMax(a)
        deleteMax(a)
        a[i] ← v
```

Trace:-

a	14	10	13	8	2	5	9	3	7

v=14

del_max a	13	10	9	8	2	5	7	3	14

V= 13

del_max a	10	8	9	3	2	5	7	13	14

V= 10

delete_max	9	8	7	3	2	5	10	13	14

Analysis:

$$C(n) = C_{heapify}(n) + n \cdot C_{delete\_max}(n)$$

Analysis:

$$C(n) = C_{heapify}(n) + n \cdot C_{\substack{delete\\max}}(n)$$

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

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

Empirically:

heapsort is slightly better than mergesort

⊕ heapsort is in-place.

heapsort is slightly slower than quicksort.