8 Transform-and-Conquer

Thursday, November 9, 2023 12:12 PM

Design technique :- 2-stage process:

1) Transform the input
2) Solve problem.

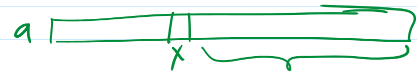
Main : transform the problem into a more convenient instance of the problem.

Problem #1

Check uniqueness in elements in an Array

try 1 :

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



$$C_{worst}(n) = (n-1) + (n-2) + (n-3) + \ldots + 1$$
$$= \frac{n(n-1)}{2} \in \Theta(n^2)$$

warning:
not all for loops should be for x in a.

a.index( )
$1 + 2 + 3 + 4 + 5 + \cdots + n$

a[k+1:]
$(n-1) + (n-2) + (n-3) + \ldots 1 + 0$

try 2 : transform the input

```
FUNCTION areUnique( a[0..n-1] )
    sort(a)                              } Sort
    FOR i <- 0 TO n-2 DO                 }
        IF a[i] = a[i+1] THEN            } scan
            RETURN false                 }
    RETURN true.                         }
```

Analysis.

Worst case: The elements in a are unique.

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

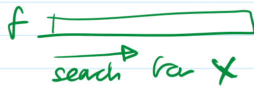
¡¡ Presorting!!

Problem #2 : Finding a "mode" in an array

"mode" is the value that repeats the most in the array

- try 1: brute force.
  - Need a list of items with their frequency
  - for every element in input

    find it in the frequency list and increase it's frequency by one (add to the list with frequency 1 if element is not found)

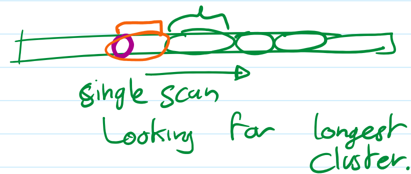


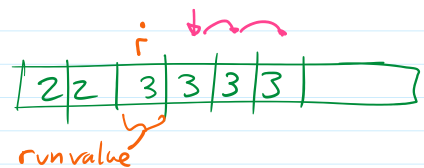
worst case: all elements are unique.

$1+2+3+4+\dots+n-1 \in \Theta(n^2)$

- try 2: Transform and conquer:

```
FUNCTION PresortMode ( a[0..n-1] )
    sort ( a )
    i <- 0                    .modevalue
    modefreq <- 0
    WHILE i <= n-1 DO
        .runlength <- 1
        .runvalue <- a[i]
        WHILE i + runlength <= n-1 and a[i+runlength] = runvalue DO
            runlength <- runlength + 1
        IF runlength > modefreq THEN
            modefreq <- runlength
            .modevalue <- runvalue
        i <- i + runlength.
    RETURN modevalue.
```





Analysis:

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

- Problem #3:

  Searching in an array

  brute force searching is $\Theta(n)$

  Transform and Conquer.
  - sort (a)
  - binary seach (a, x)

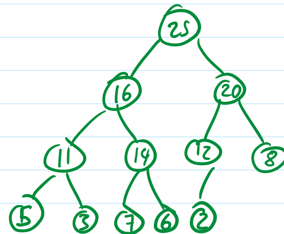
Analysis:

$$C_{worst}(n) = C_{sort}(n) + C_{binSearch}(n)$$
$$= \Theta(n \log n) + \Theta(\log n)$$
$$= \Theta(n \log n)$$

- HEAPS and HEAPSORT

Review of A.D.T. Heap:

- A Binary tree.
- semi-complete: every level except the last one is full; bottom level is full from left to right.
- "heap" property: every node is greater than both it's children
  Note: nothing is said about siblings.

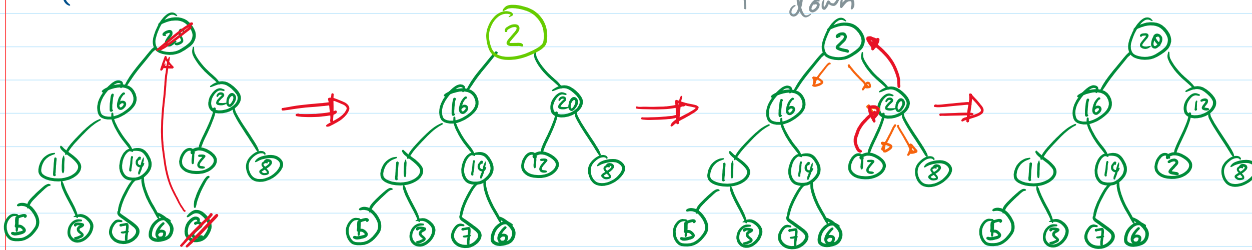


widely used to implement "priority Queues"

- find element with higest value → the root
- deleting element with highest value ↗
- adding a new element to the heap.

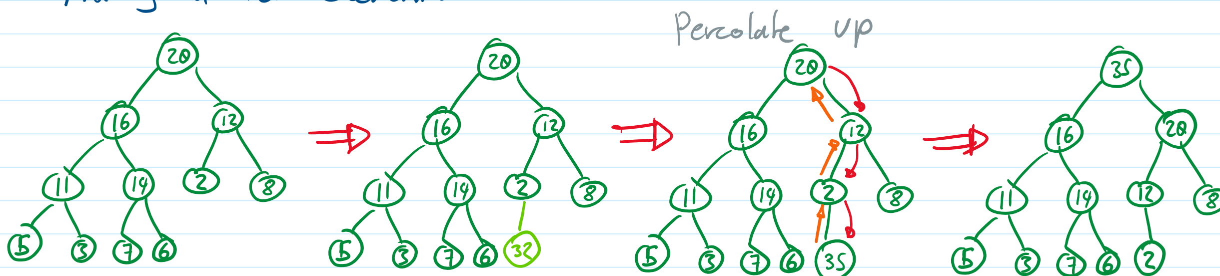
- Delete element. (with highest value)

percolate down



- Adding a new element.

Percolate up



Analysis:

Analysis:

Both operations need $\Theta(\log n)$ steps.

- Implementation the D.S. ArrayHeap.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
35	16	20	11	14	12	8	5	3	7	6	2				

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

- HEAPSORT

to sort:
1.- transform data into a heap.
2.- build sorted array in reverse order by picking biggest element in the heap

#1.- transform.

8	15	13	5	16	7	18	10	12	

heapify

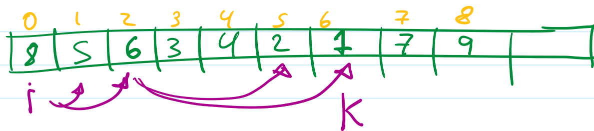
- for each intermediate node x percolate node down to it's appropiate level by swaping it the greatest of it's two children continously

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

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

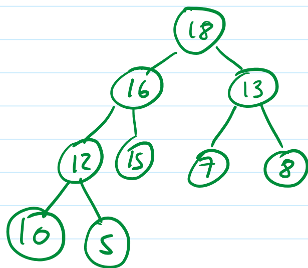
best_child. n = 9

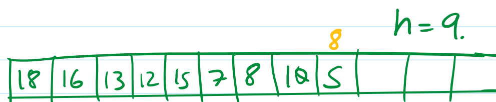
Trace: i=0
v=1



Analisys: heapify is linear. number of comparisons is < 2n.

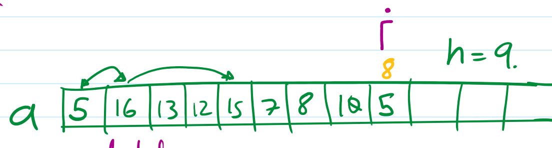
#2 sort.






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

Trace:



Trace:

n = 9.

a: | 5 | 16 | 13 | 12 | 15 | 7 | 8 | 10 | 5 | | | |

v = 18 (delete_max)

| 16 | 15 | 13 | 12 | 5 | 7 | 8 | 10 | 18 | | | |

v = 16 (delete_max)

| 15 | 12 | 13 | 10 | 5 | 7 | 8 | 16 | 18 | |

v = 15

| 13 | 12 | 8 | 10 | 5 | 7 | 15 | 16 | 18 | |

heap | sorted

Analysis:

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

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

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

Empirically:

heapsort is slightly better than merge_sort ⊕ heapsort is in place

heapsort is slightly slower than quicksort.

—o—o— EOF