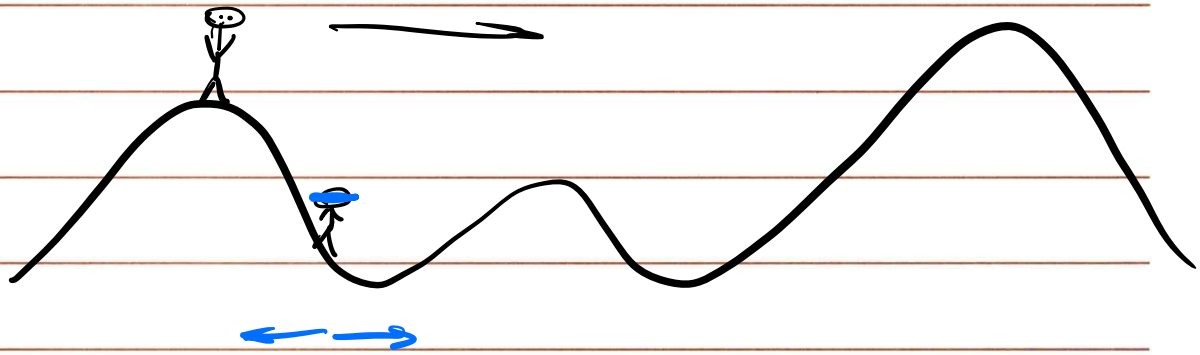


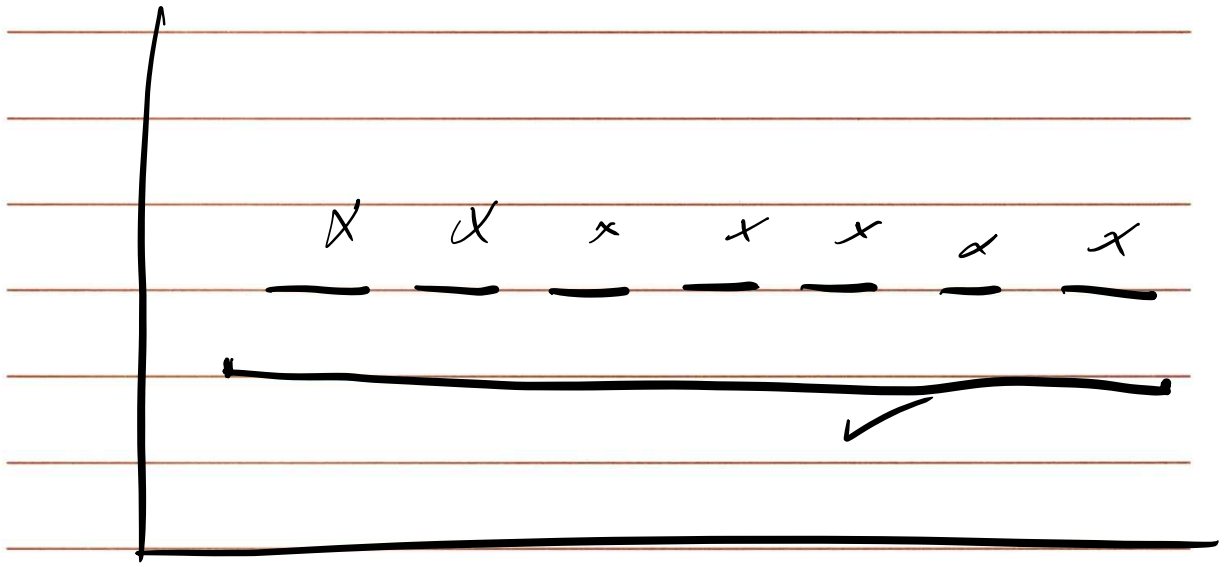
Interval scheduling Problem

Input: Set of requests {1..n}

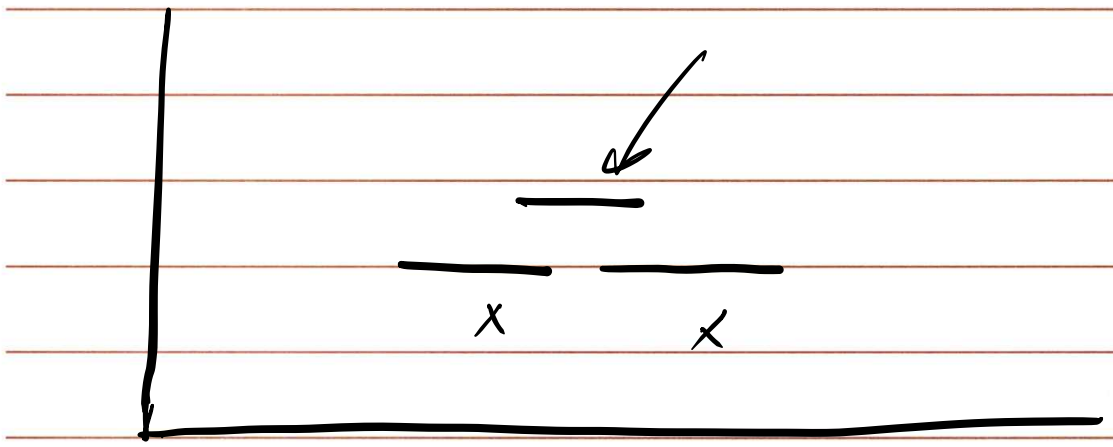
i^{th} request starts at $s(i)$ and ends at $f(i)$

Objective: To find the largest compatible subset of these requests

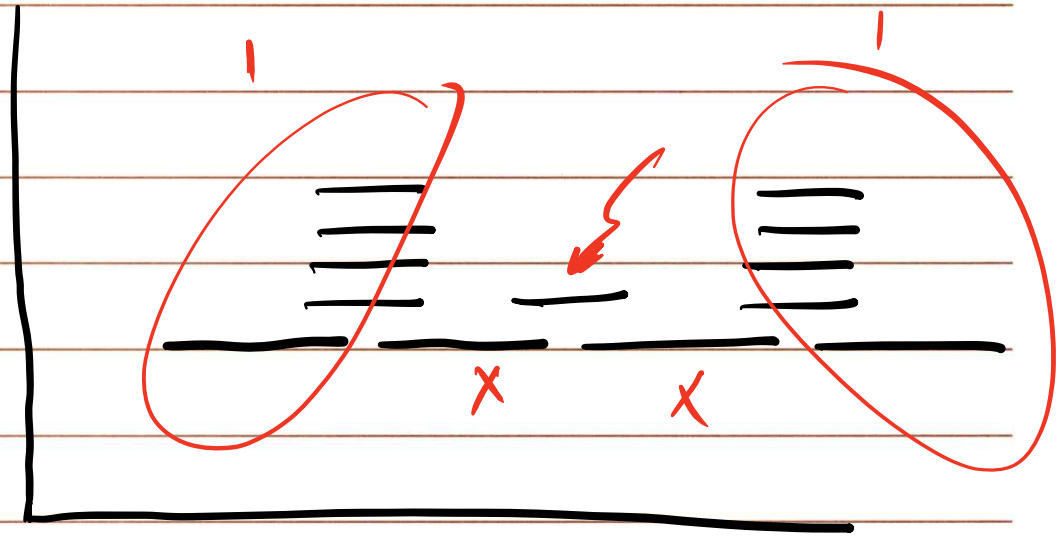
try #1 Earliest strat time X



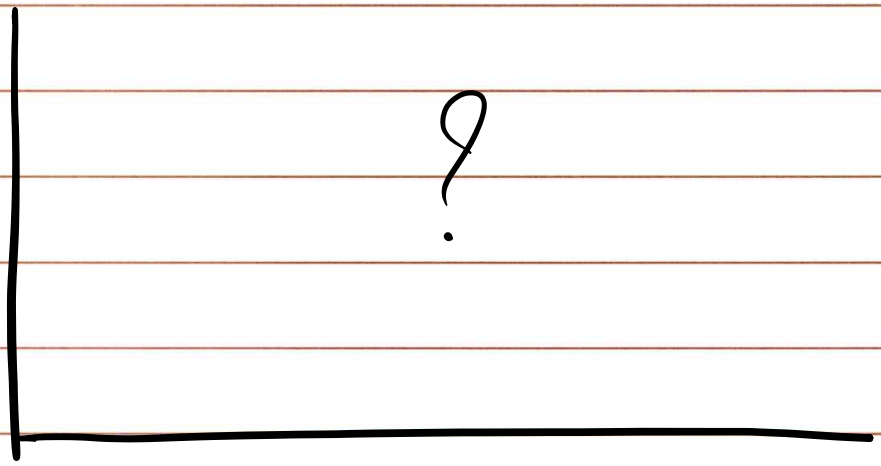
try #2 Smallest jobs first X



Try #3 Smallest no. of overlaps first



Try #4 Earliest finish time



Solution:

Initially R is the complete set of requests
& A is empty

While R is not empty

Choose a request $i \in R$ that has the smallest finish time

Add request i to A

Delete all requests from R that are not compatible w/ i

endwhile

Return A

Proof of Correctness

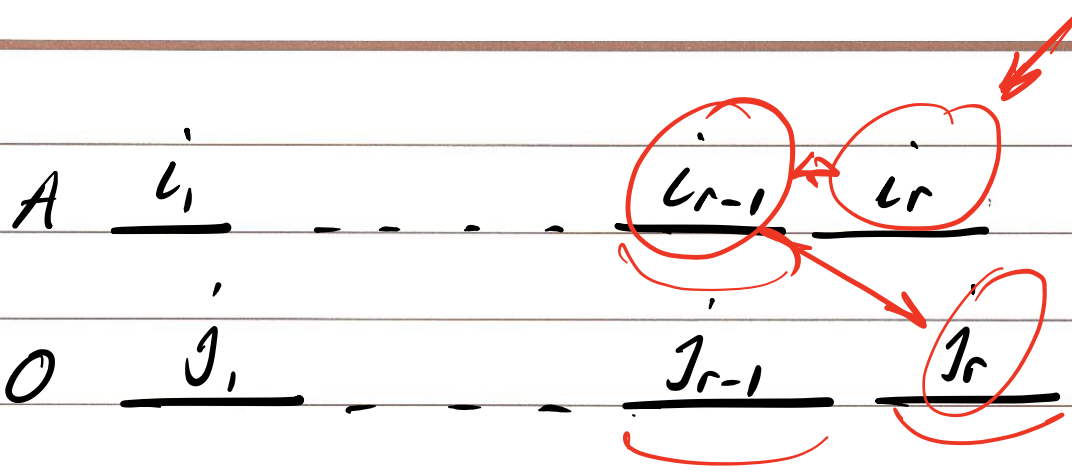
(1) Show that A is a compatible set

(2) Show that A is an optimal set

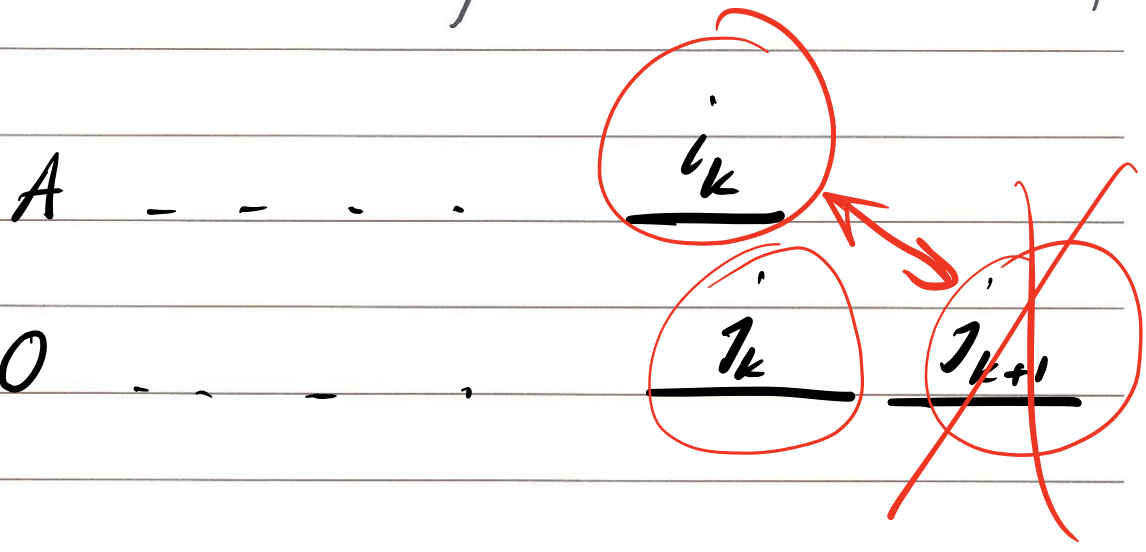
Say A is of size k
Say there is an opt. solution O

Requests in A: $i_1 \ldots\ldots i_k$
" " O: $j_1 \ldots\ldots j_m$

We will first prove that for all indices $r \leq k$, we have $f(i_r) \leq f(j_r)$



We can then easily prove that $|A| = |O|$

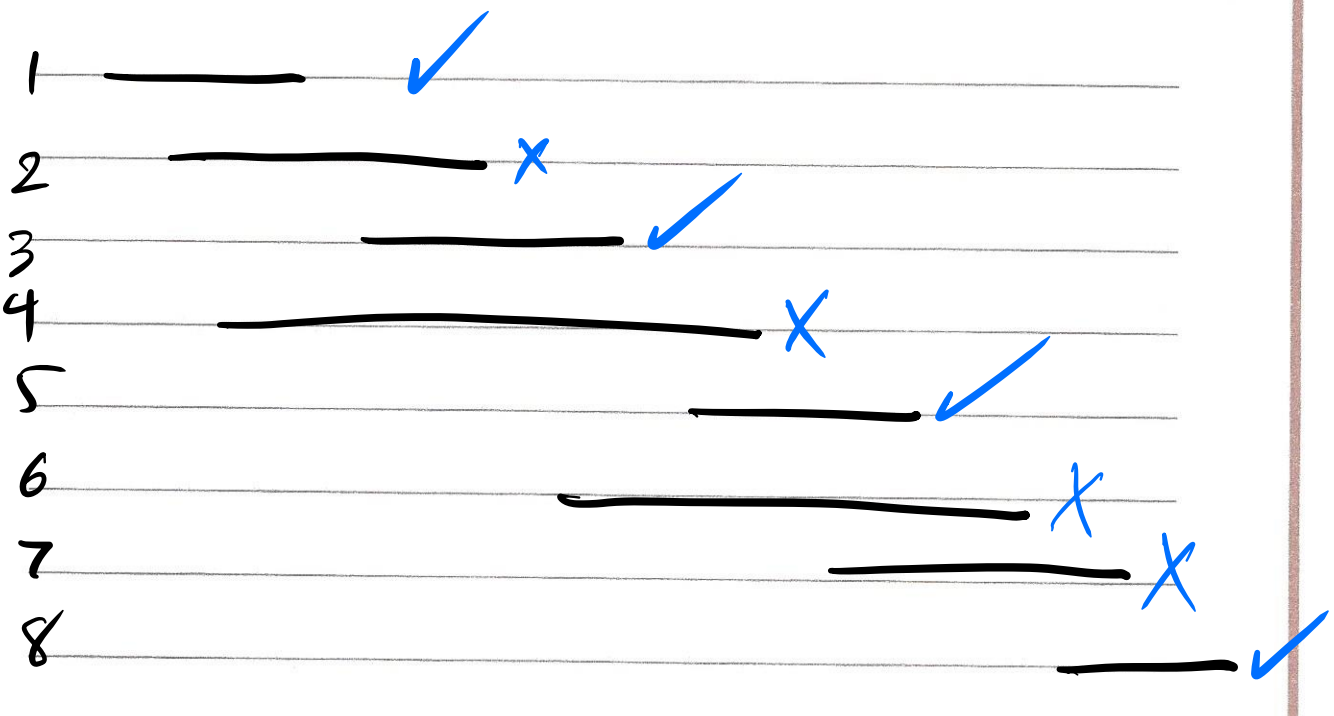


Implementation

$O(n \log n)$ — Sort requests in order of finish time and label in this order:

$f(i) \leq f(j)$ where $i < j$

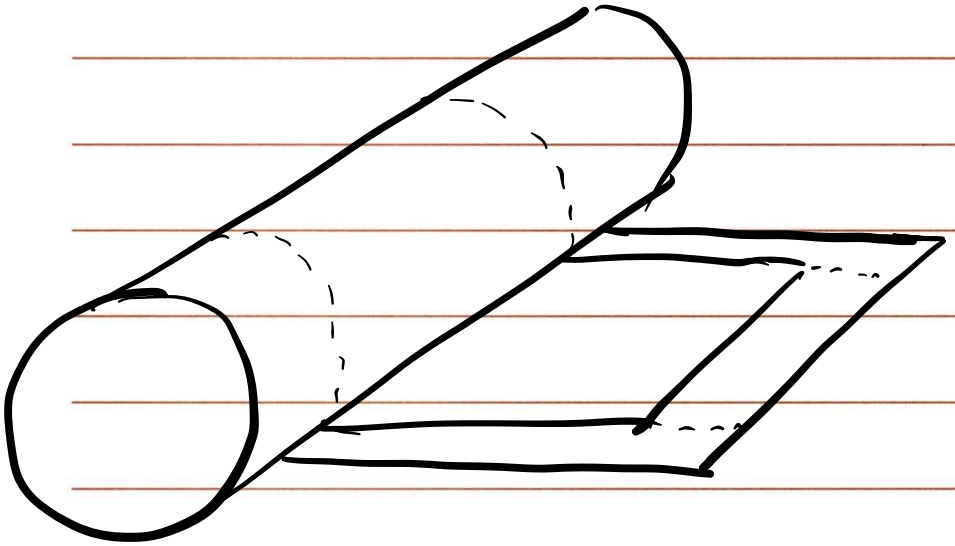
$O(n)$ — Select requests in order of increasing $f(i)$, always selecting the first. Then iterate through the intervals in this order until reaching the first interval for which $s(j) \geq f(i)$

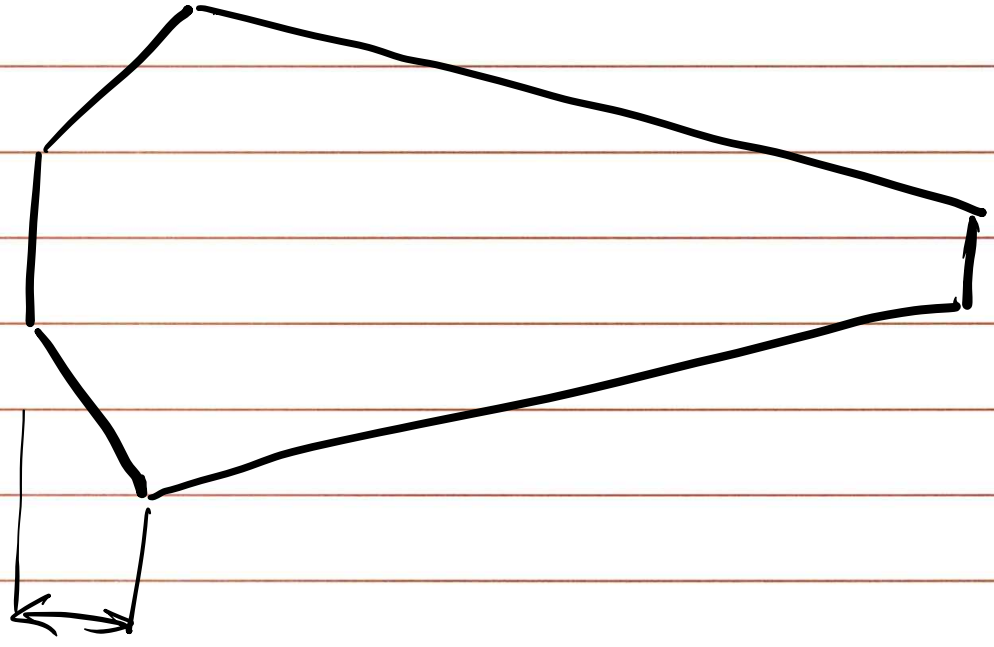


Overall performance = $O(n \log n)$

Orders:

- Qty
- width
- grade





Warm up

Fractional Knapsack

Knapsack has a weight capacity of W.

We are given as input, a set of n objects with weight w_i and value v_i.

Objective: Fill up the knapsack to its weight capacity such that the value of items in knapsack is maximized.

Ex. knapsack weight cap: 10

items	1	2	3	4	5
Values	10	20	15	2	8
weights	4	10	5	1	2

value/weight: 2.5 | 2 | 3 | 2 | 4

order: 5, 3, 1, 2, 4

total value: $8 + 15 + \frac{3}{4} * 10 = 30.5$

Scheduling to Minimize Lateness

Scheduling to Minimize Lateness

- Requests can be scheduled at any time
- Each request has a deadline
- Notation: $L_i = f(i) - d_i$

  L_i is called lateness for request i.

Goal: Minimize the Maximum Lateness $L = \max_i L_i$

Sol. 1

Job 1 Late by 5 hrs

Job 2 " " 6 hrs

} 6

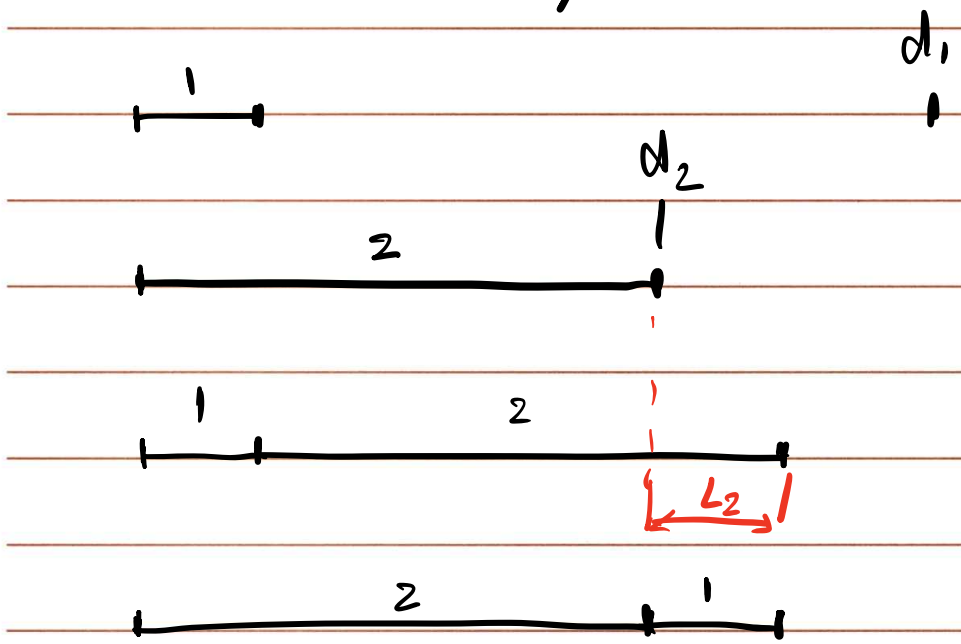
Sol. 2

job 1 Late by 0 hrs

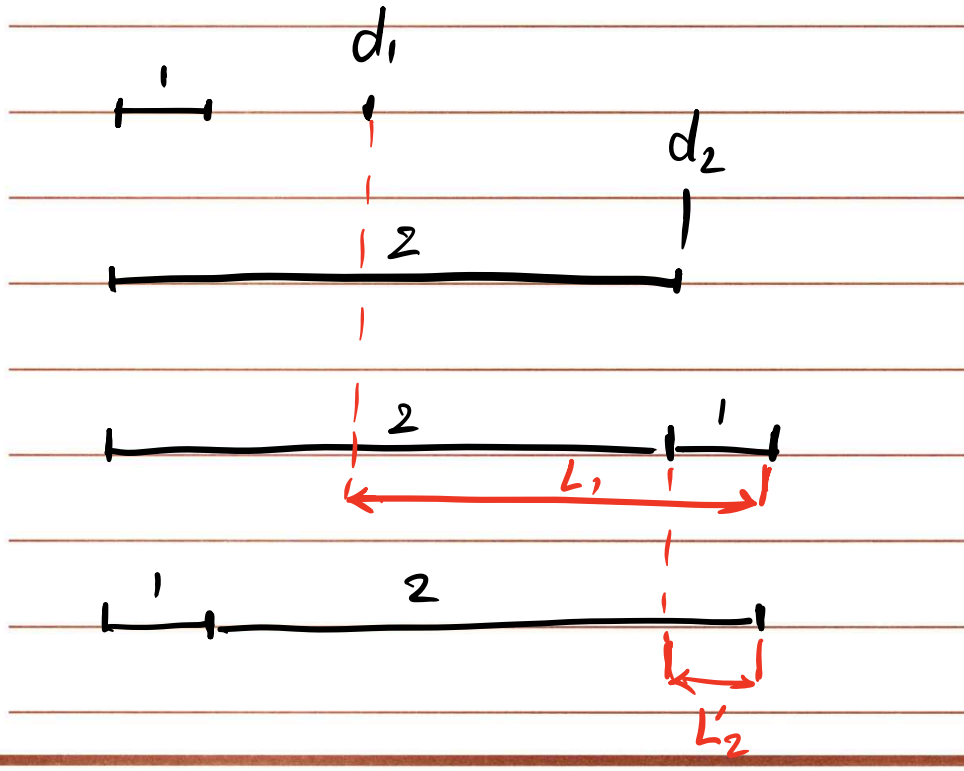
job 2 " " 7 hrs

} 7

try #1 Smallest requests first ✗



try #2 Shortest slack first ✗

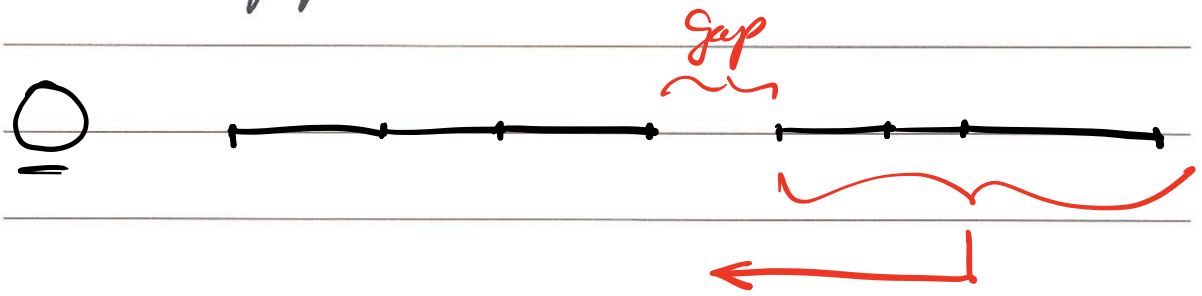


Solution:

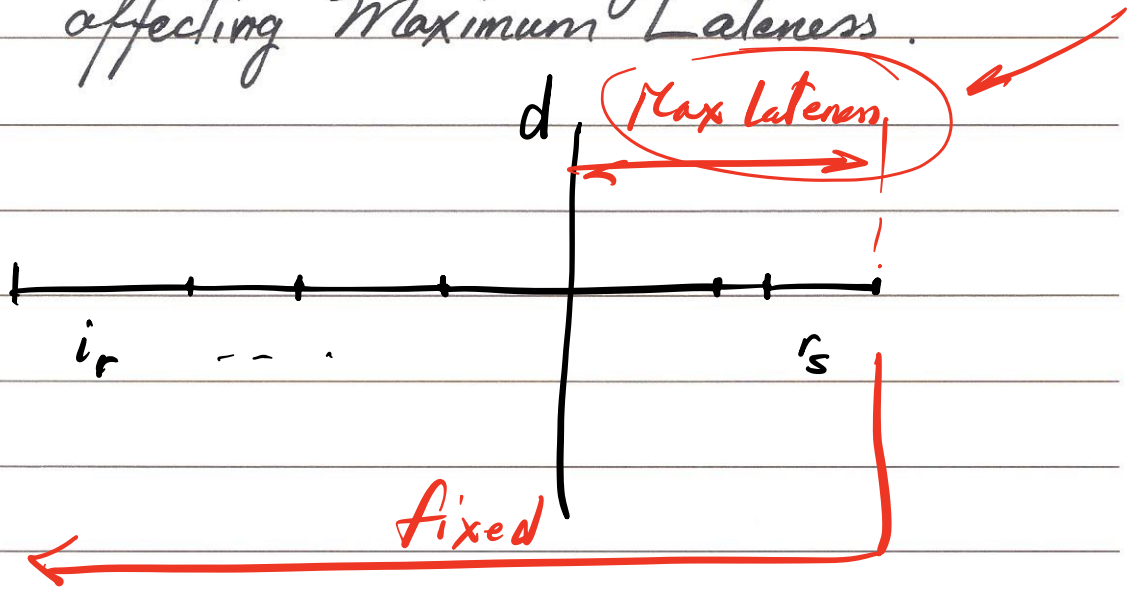
Schedule jobs in order of their deadline without any gaps between jobs.

Proof of Correctness

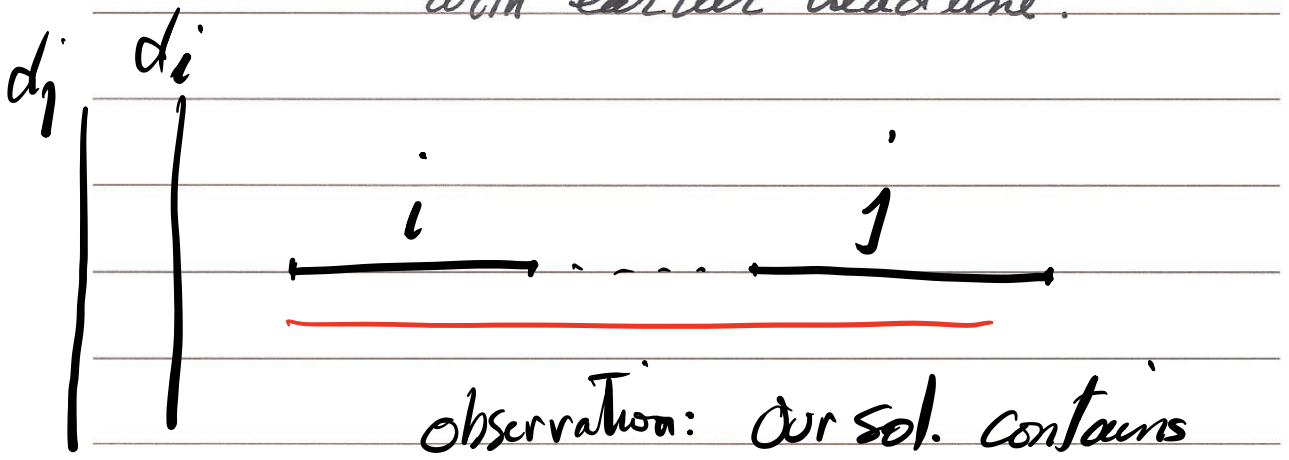
① There is an optimal solution with no gaps.



(2) Jobs with identical deadlines can be scheduled in any order without affecting Maximum Lateness.



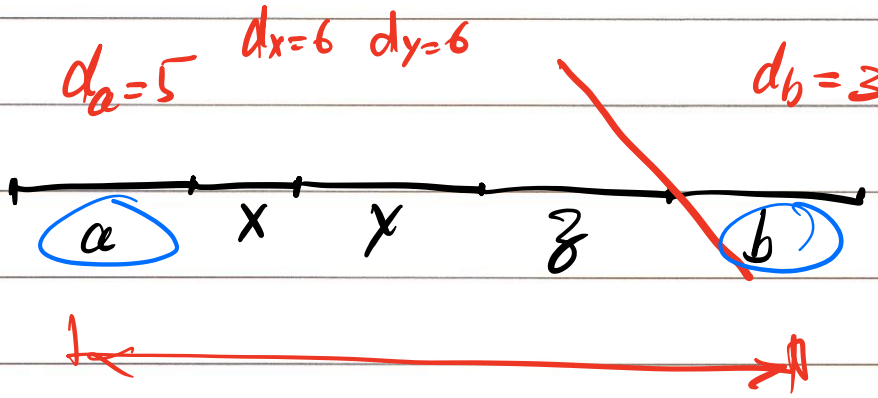
(3) Def. Schedule A' has an inversion if a job i with deadline d_i is scheduled before job j with earlier deadline.

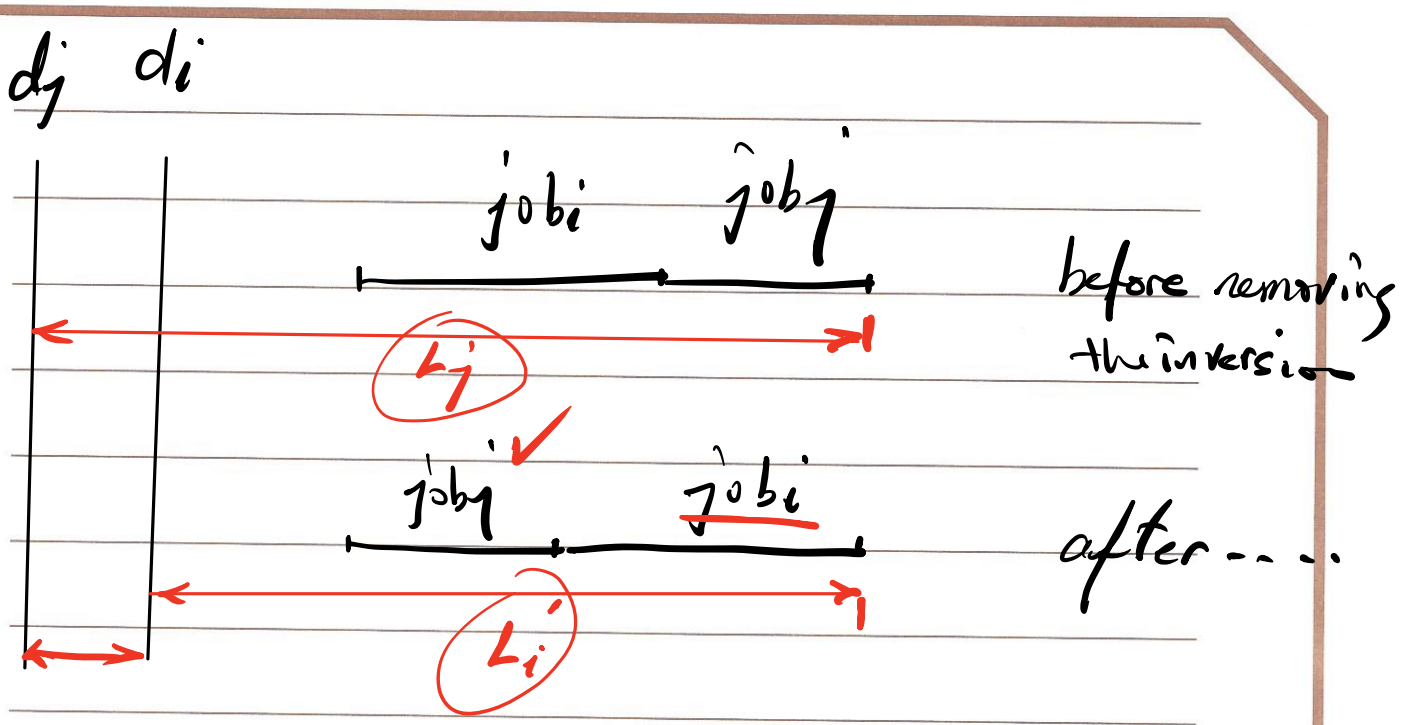


observation: our sol. contains no inversions

(4) All schedules with no inversions and no idle time have the same Maximum Lateness.

(5) There is an optimal schedule that has no inversions and no idle time.





So, if there is an optimal solution that has inversions, we can eliminate the inversions one by one as shown above until there are no more inversions. This solution will also be optimal.

(6) Proved that there exists an optimal schedule with no inversions and no idle time.

Also proved that all schedules with no inversions and no idle time have the same Maximum Lateness.

Our greedy algorithm produces one such solution ⇒ It will be optimal

Priority Queues

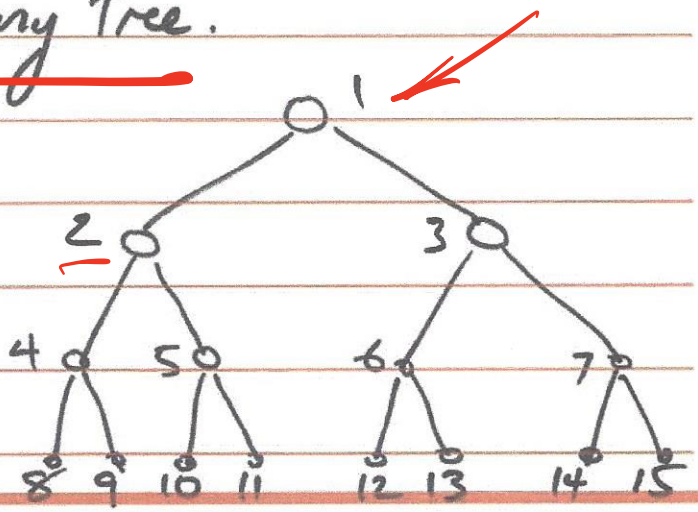
A priority queue has to perform these two operations fast!

1. Insert an element into the set
2. Find the smallest element in the set

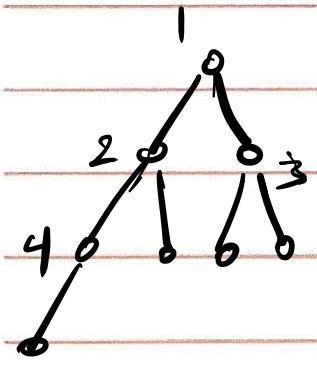
	Insert	Find
array	$O(1)$	$O(n)$
Sorted array	$O(n)$	$O(1)$
linked list	$O(1)$	$O(n)$
Sorted linked list	$O(n)$	$O(1)$

Background

Def. A binary tree of depth k which has exactly $2^k - 1$ nodes is called a full binary tree.



Def. A binary tree with n nodes and of depth k is complete iff its nodes correspond to the nodes which are numbered 1 to n in the full binary tree of depth k.



Traversing a complete binary tree stored as an array

Parent(i) is at $\lfloor i/2 \rfloor$ if $i \neq 1$
if $i = 1$, i is the root

Lchild(i) is at $2i$ if $2i \leq n$
otherwise it has no left child

Rchild(i) is at $2i+1$ if $2i+1 \leq n$
otherwise it has no right child

Def. A binary heap is a complete binary tree with the property that the value (of the key) at each node is at least as large as the values at its children (Max heap)

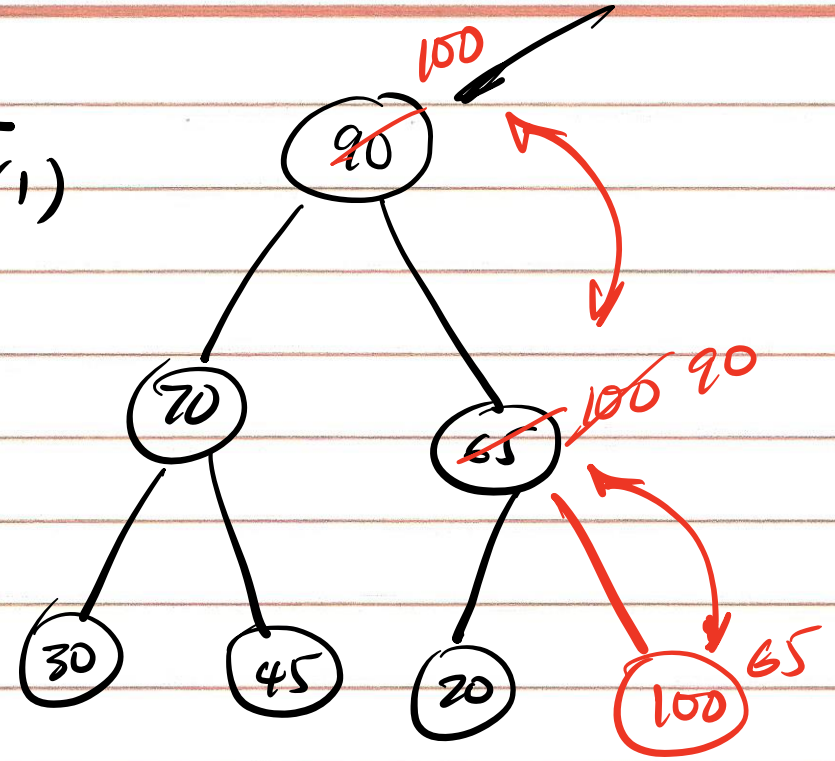
Find_Max

takes O(1)

Insert

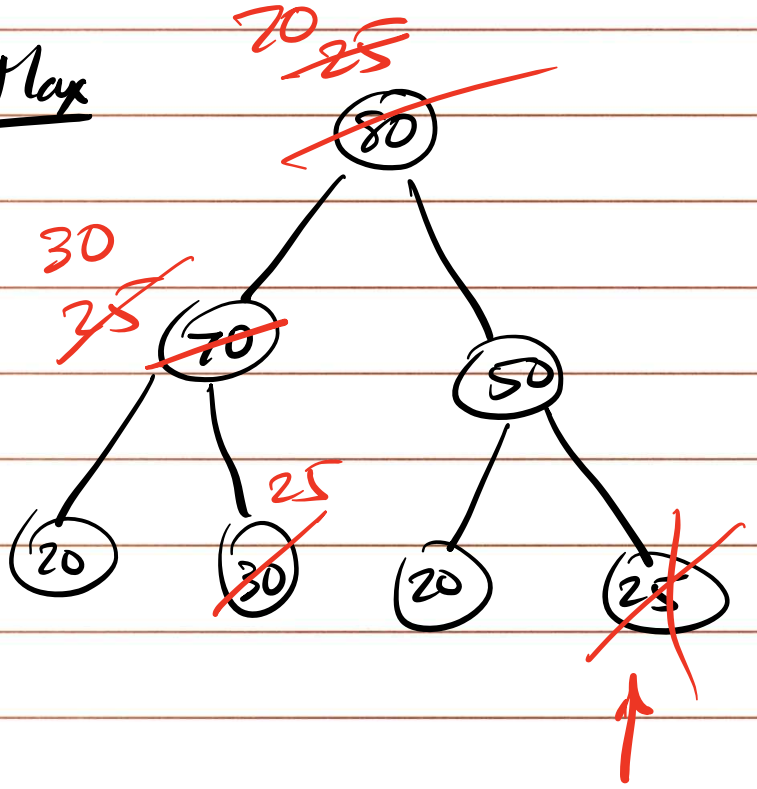
insert (100)

takes O(lgn)



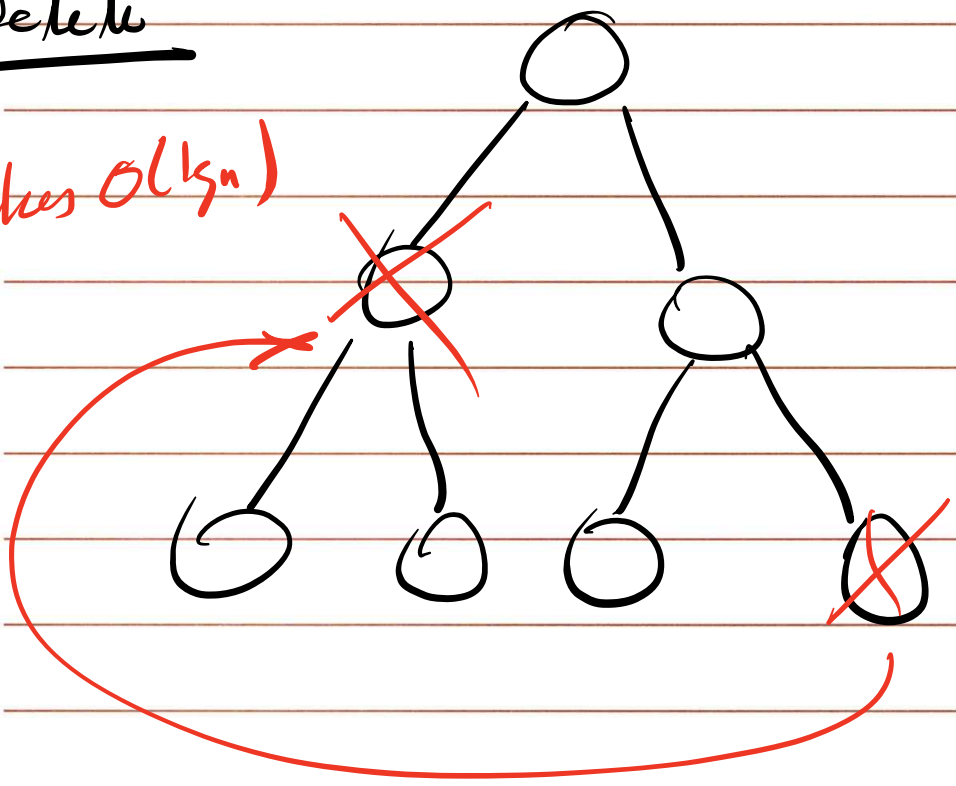
Extract_Max

takes $O(\lg n)$



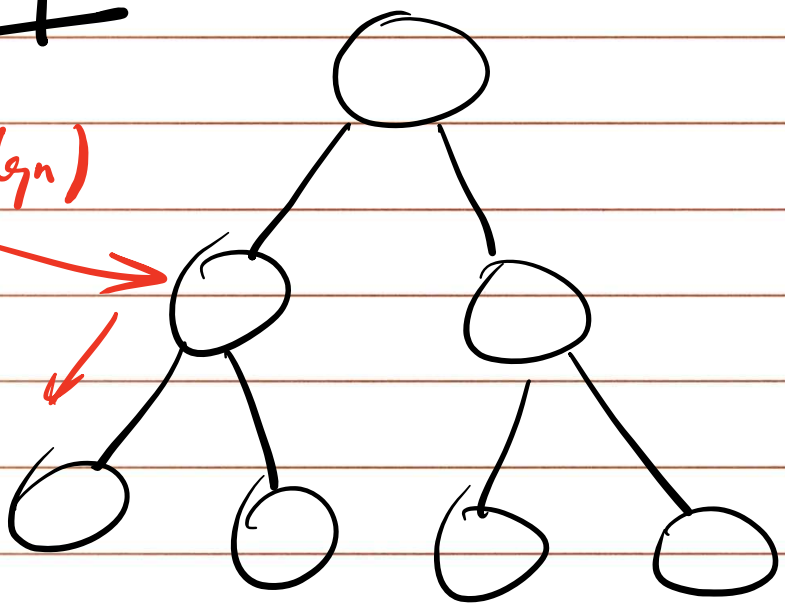
Delete

takes $O(\lg n)$



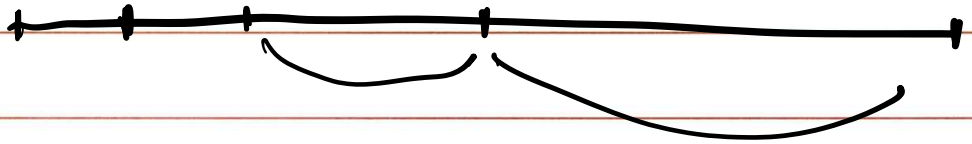
Decrease-key

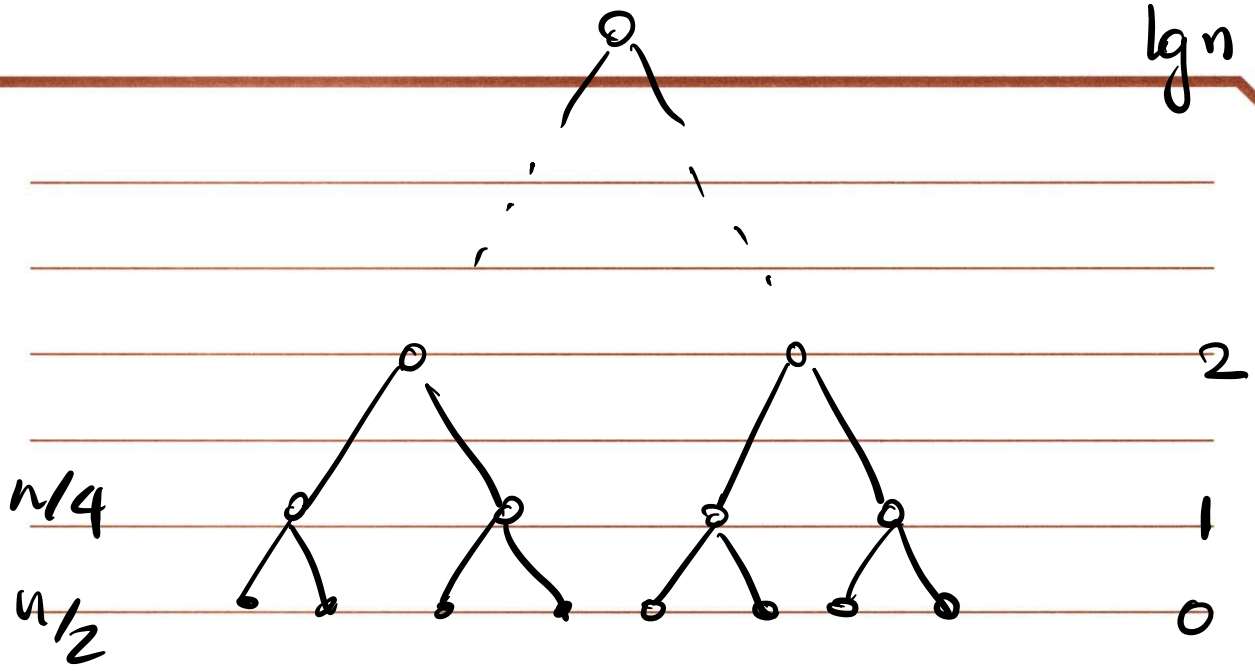
takes $O(\lg n)$



Construction

Can be done in $O(n \lg n)$ time using n insert op's.





$n/4$	1
$n/8$	2
$n/16$	3
⋮	
1	$\lg n$

$T = n/4 * 1 + n/8 * 2 + n/16 * 3 + \ldots$

$T/2 = n/8 * 1 + n/16 * 2 + n/32 * 3 + \ldots$

$T - T/2 = \underbrace{n/4 + n/8 + n/16 + \ldots}_{n/2}$

$T/2 = n/2$

$T = n$

$\underbrace{n/2 + n/4 + n/8 + \ldots}_{n}$

Bottom up construction takes $O(n)$

Merge op.

merging 2 binary heaps of size $\frac{n}{2}$ takes $O(n)$ using bottom up construction of the heap.

Problem Statement

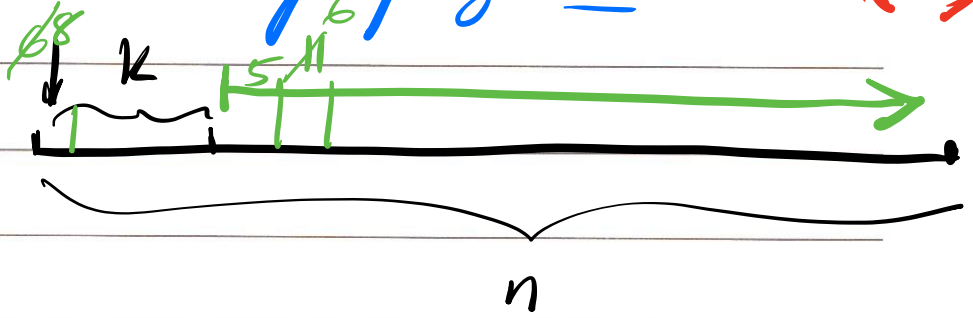
Input: An unsorted array of length n

Output: Top k values in the array ($k < n$)

Constraints:

- You cannot use any additional memory
- Find an algorithm that runs in time $O(n \lg k)$

Construct a Minheap of size k $O(k)$



Move through the rest of $(n-k)$ elements

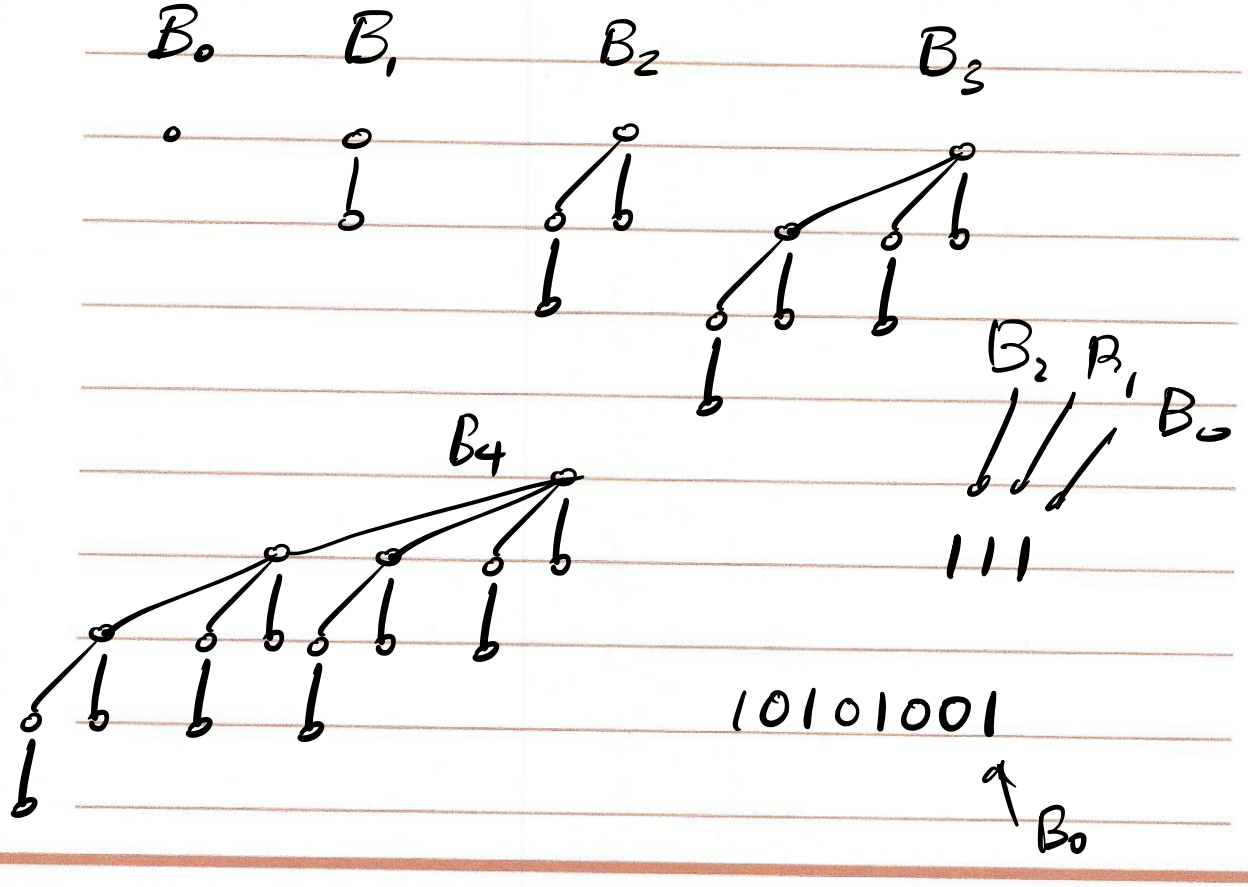
- extract-Min $O(\lg k)$
- insert $O(\lg k)$

overall complexity $= O(k) + O((n-k)\lg k)$

$= O(n \lg k)$

Def. A binomial tree B_k is an ordered tree defined recursively

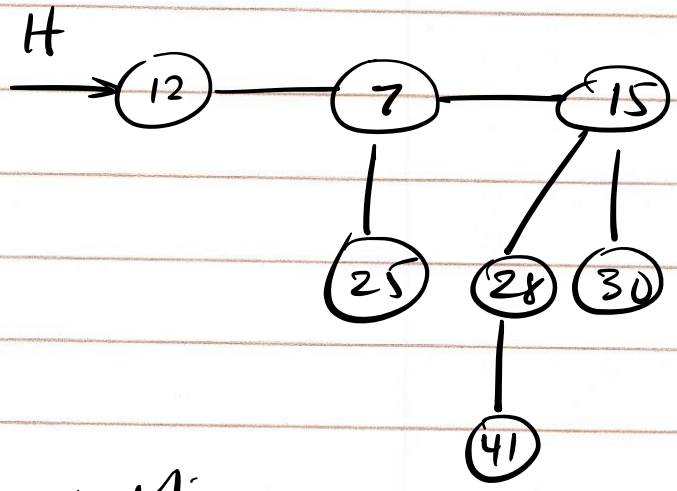
- Binomial tree B_0 consists of one node
- Binomial tree B_k consists of 2 binomial trees B_{k-1} that are linked together such that root of one is the leftmost child of the root of the other.



Def: A binomial Heap H is a set of binomial trees that satisfies the following properties:

1- Each binomial tree in H obeys the min-heap property

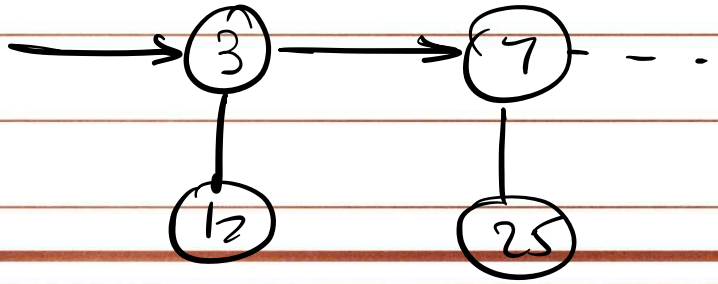
2- For any non-negative integer k, there is at most one binomial tree in H whose root has degree k.

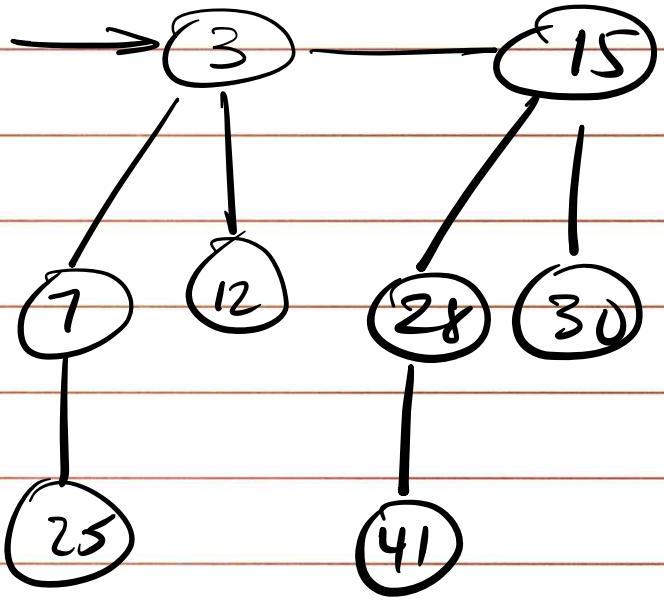


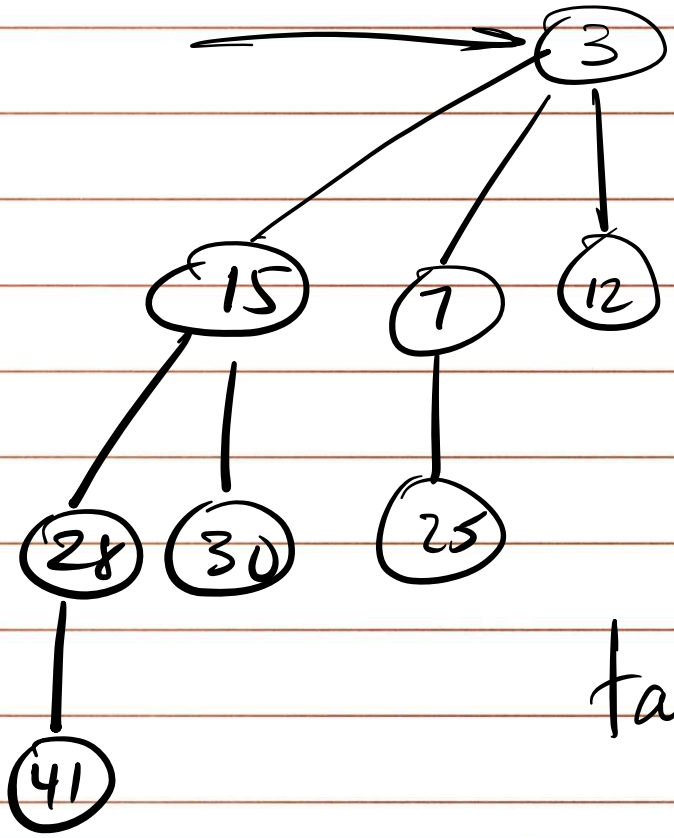
Find-Min takes $O(\log n)$

insert

insert (3)







takes $O(\lg n)$

Amortized Cost Analysis

Ex. 1

```
for i = 1 to n
    push or Pop
endfor
```

push takes $O(1)$
Pop takes $O(1)$

Our worst case runtime complexity = $O(n)$

Ex. 2

```
for i = 1 to n
    push or pop or multipop
endfor
```

push takes $O(1)$ worst-case
pop takes $O(1)$ worst-case
multipop takes $O(n)$ worst-case

Our worst case runtime complexity = $O(n^2)$

is this a tight bound?

Aggregate Analysis

- We show that a sequence of n operations (for all n) takes worst-case time $T(n)$ total.

- So, in the worst case, the amortized cost (average cost) per operation will be $T(n)/n$

observation: Multipop takes $O(n)$ time if there are n elements pushed on the stack

A sequence of n pushes takes $O(n)$
multipop takes $O(n)$ worst-case

$T(n) = O(n)$

when amortized over n operations,
average cost of an operation $= O(n)/n = O(1)$

Ex. 2

```
for i = 1 to n
    push or pop or multipop
end for
```

push → O(1) pop → O(1) multipop → O(1)

push takes O(1) amortized

pop " O(1) "

multipop " O(1) "

our worst case runtime complexity = $O(n)$

$= \Theta(n)$

Accounting Method

- We assign different charges (amortized costs) to different operations

- If the charge for an operation exceeds its actual cost, the excess is stored as credit.

- The credit can later helps pay for operations whose actual cost is higher than their amortized cost.

- Total credit at any time = total amortized cost − total actual cost

Total
- ^Credit can never be negative.

Ex. 2

```
for i = 1 to n
    push or pop or multipop
endfor
```

try # 1

assign a ~~cost~~ charge of 1 to each operation

op.	charge	Actual cost	total credit
push	1	1	0
push	1	1	0
multipop	1	2	-1

try # 2

assign charges as follows:

push 2 → O(1)

pop 0 → O(1)

multipop 0 → O(1)

op.	charge	Actual cost	total credit
push	2	1	1
push	2	1	2
multipop	0	2	0

Ex. 2

```
for i = to n
    push or pop or multipop
end for
```

	Amortized cost
push	$O(1)$
pop	$O(1)$
multipop	$O(1)$

worst-case runtime complexity = $O(n)$

$\Theta(n)$

- Fibonacci heaps are loosely based on binomial heaps.

- A Fibonacci heap is a collection of min-heap trees similar to Binomial heaps, however, trees in a Fibonacci heap are not constrained to be binomial trees. Also, unlike binomial heaps, trees in Fibonacci heaps are not ordered.

- Link to Fibonacci heap animations:

www.cs.usfca.edu/ngalles/JavascriptVisual/FibonacciHeap.html

Amortized costs

	Binary Heap	Binomial Heap	Fibonacci Heap
Find-Min	$O(1)$	$O(\lg n)$	$O(1)$
Insert	$O(\lg n)$	"	$O(1)$
Extract-Min	"	"	$O(\lg n)$
Delete	"	"	$O(\lg n)$
Decrease-key	"	"	$O(1)$
Merge	$O(n)$	"	$O(1)$
Construct	$O(n)$	$O(n)$	$O(n)$