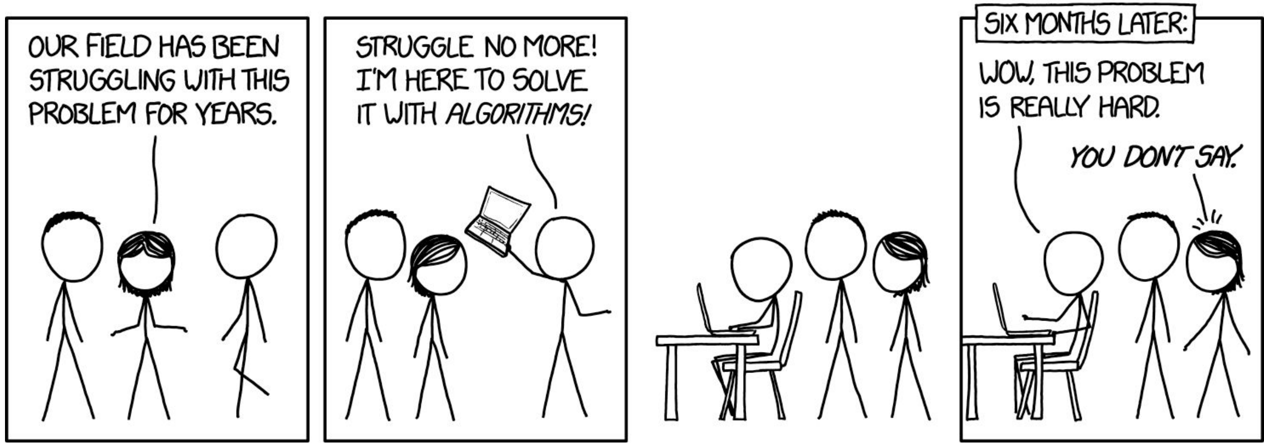
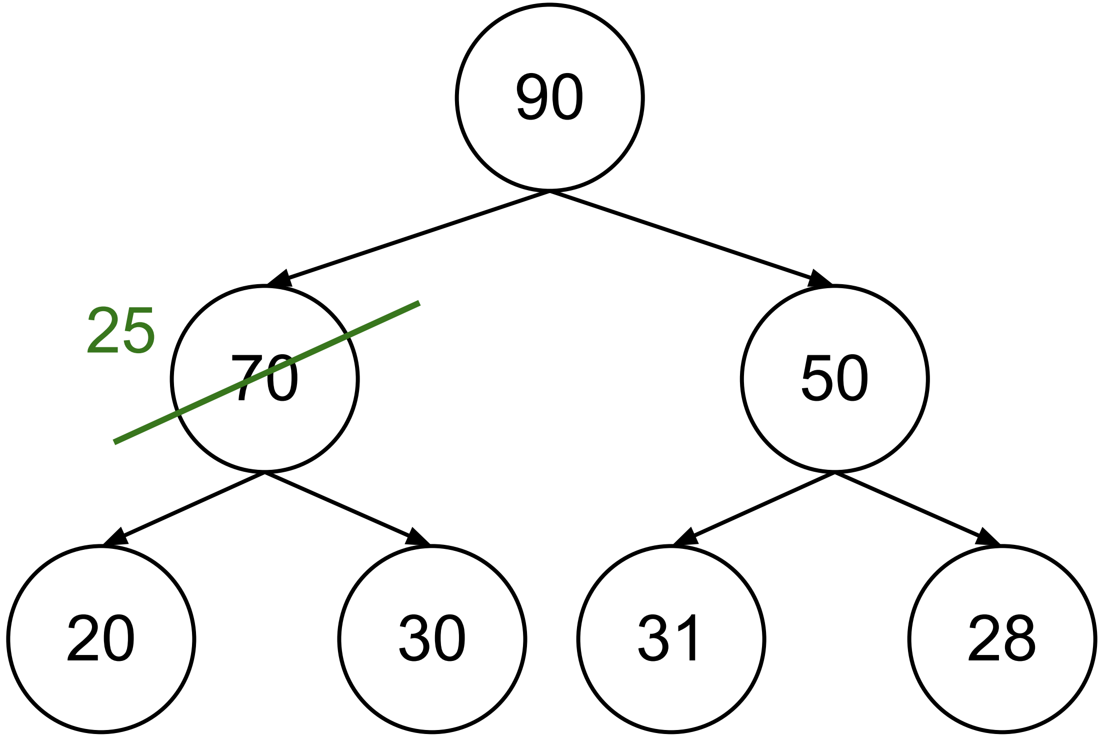
Greedy / Heaps

CSCI270 Discussion Week 3

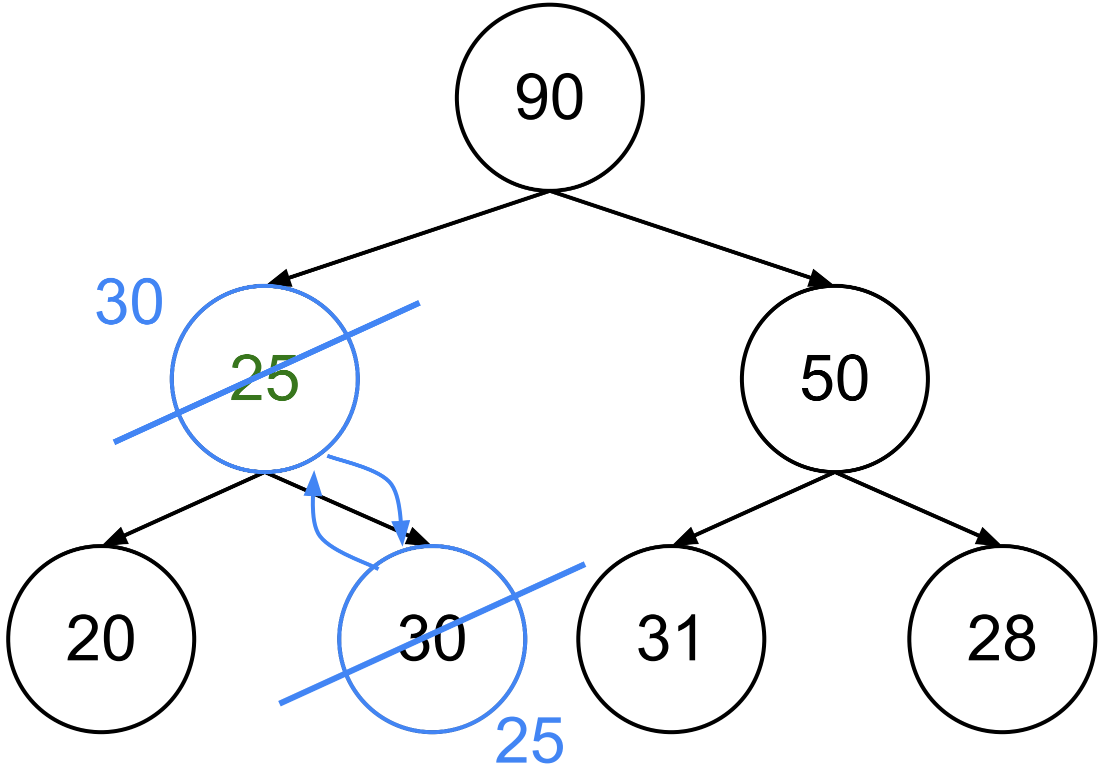
xkcd 1831


OUR FIELD HAS BEEN STRUGGLING WITH THIS PROBLEM FOR YEARS.
STRUGGLE NO MORE! I'M HERE TO SOLVE IT WITH *ALGORITHMS!*
SIX MONTHS LATER:
WOW, THIS PROBLEM IS REALLY HARD.
YOU DON'T SAY.

Heaps – decrease key



Heaps – decrease key



Heaps – decrease key

1. Find the entry in the heap ← ~~**linear search?**~~ Takes too long!

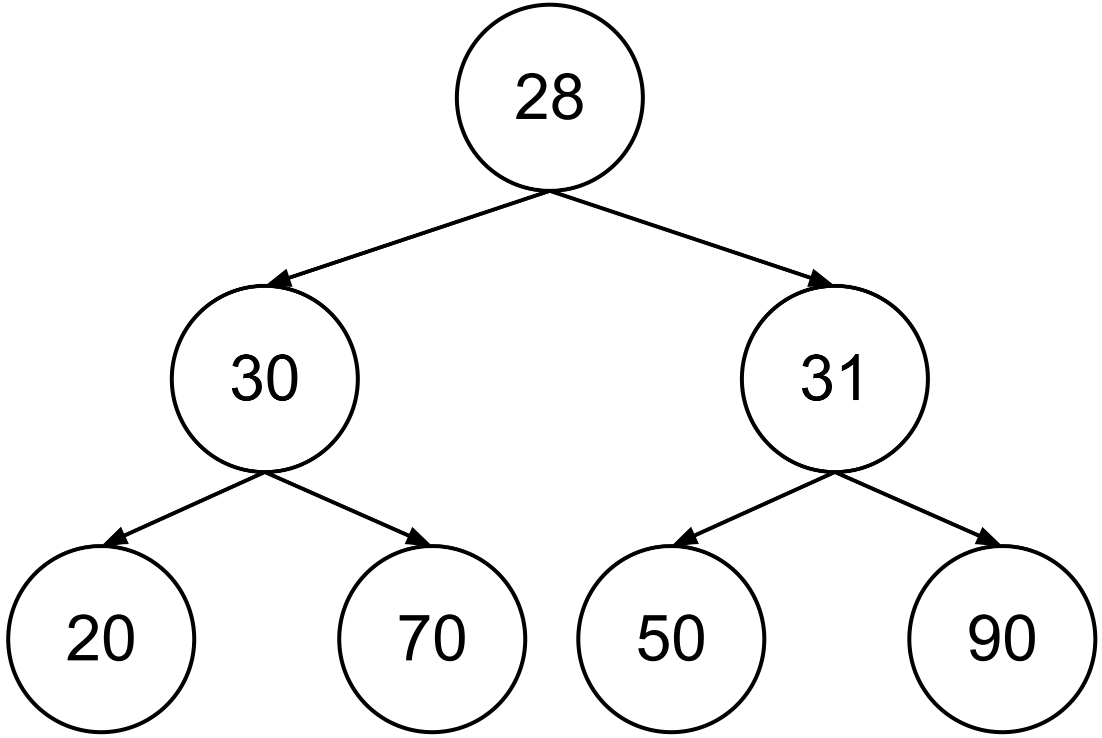
   Store an index describing where in the heap the element is located
   (Can use some sort of struct or array)
   → Lookup in constant time

2. Trickle down ← **O(logn)**

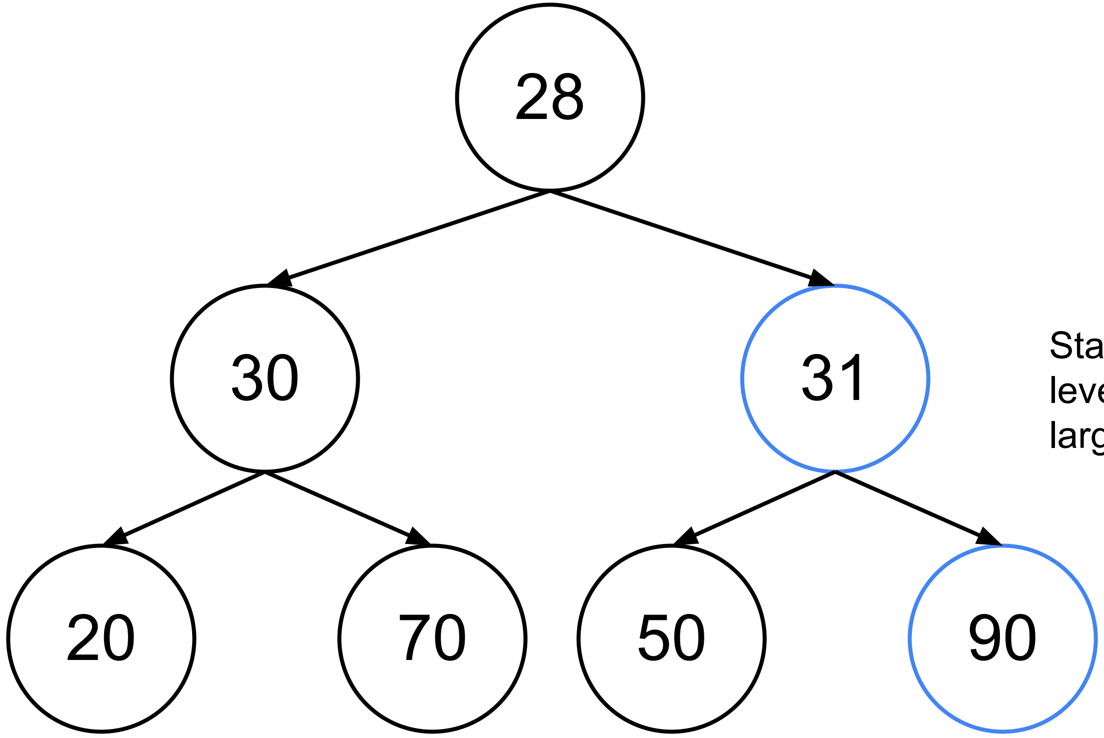
Heaps – construction

- n insert operations → O(nlogn)
- Bottom up construction → **O(n)**
  - Idea: starting from the bottom (leaf nodes), swap parent/children pairs to fix sub-heap structures, work upwards

Heaps – bottom up construction

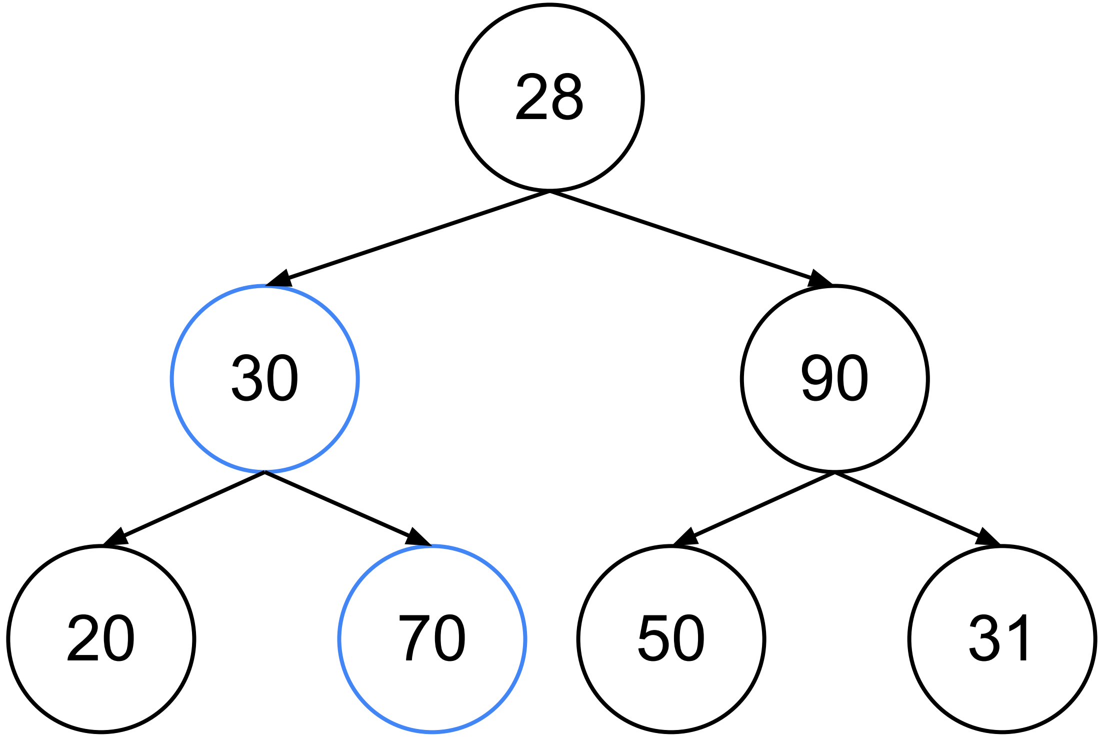


Heaps – bottom up construction

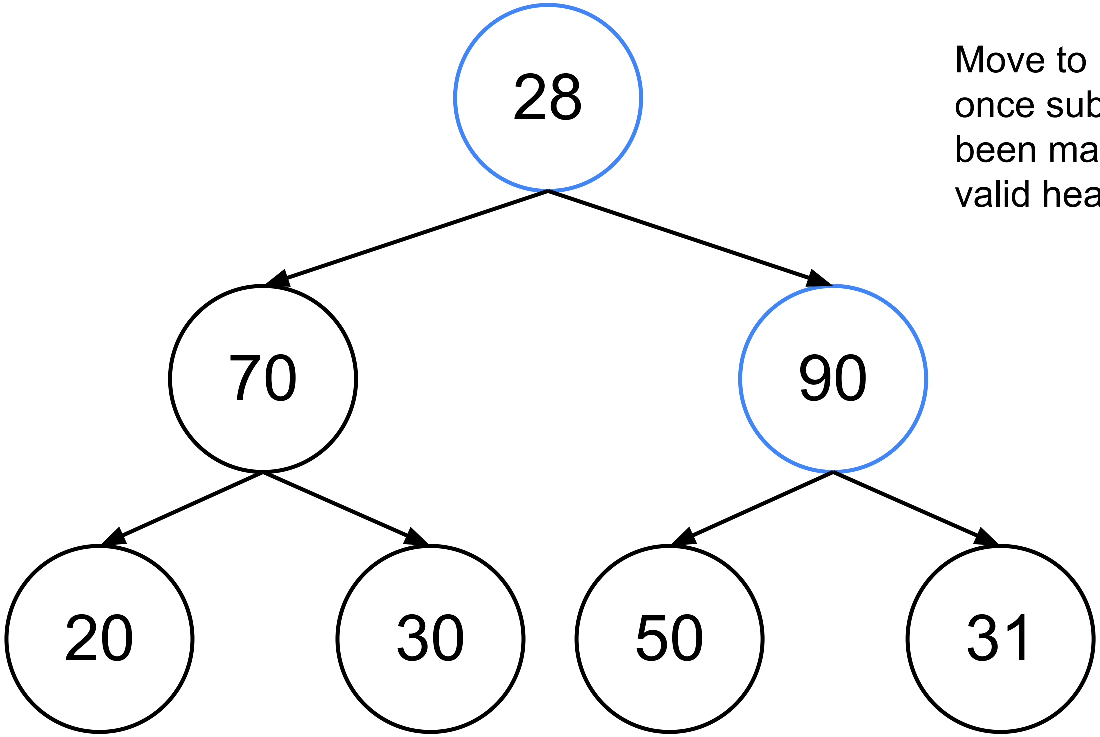


Start from bottom level, swap with largest child

Heaps – bottom up construction

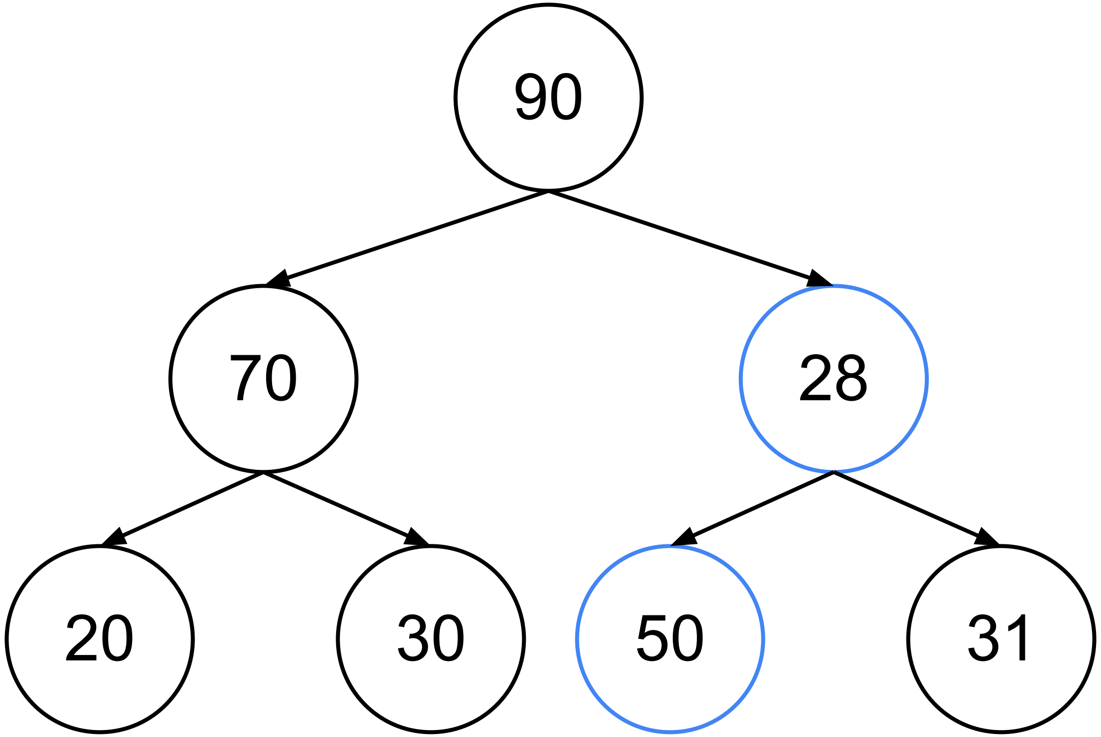


Heaps – bottom up construction

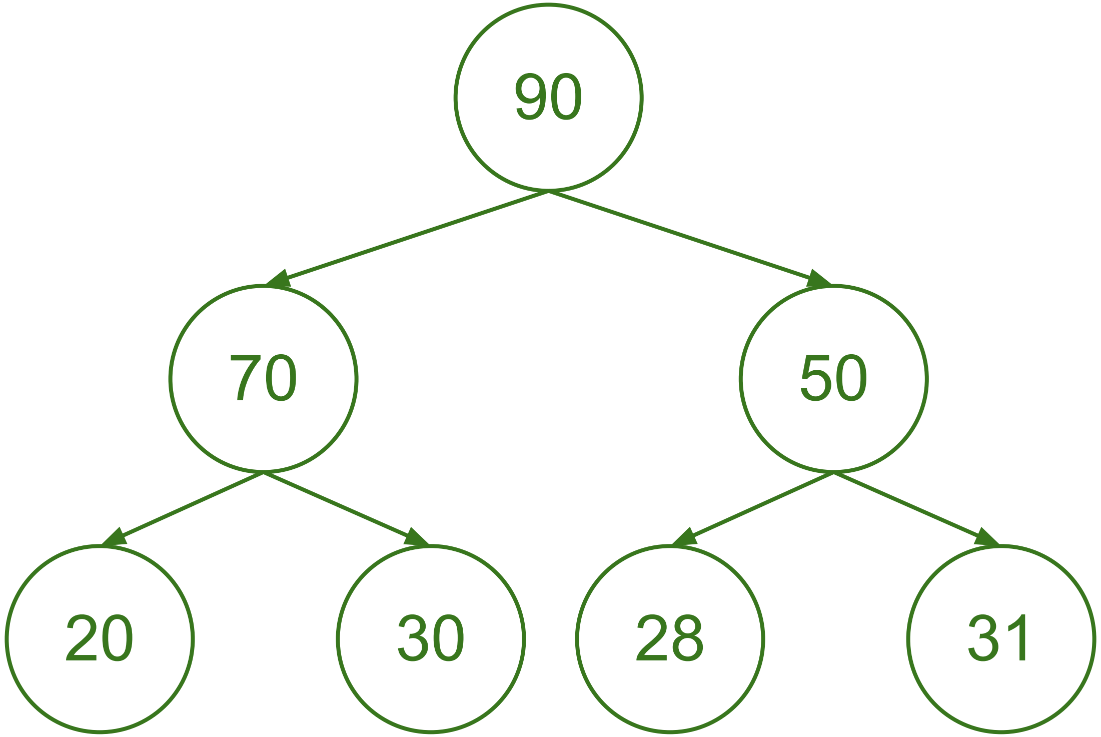


Move to next level once sub-trees have been made into valid heaps

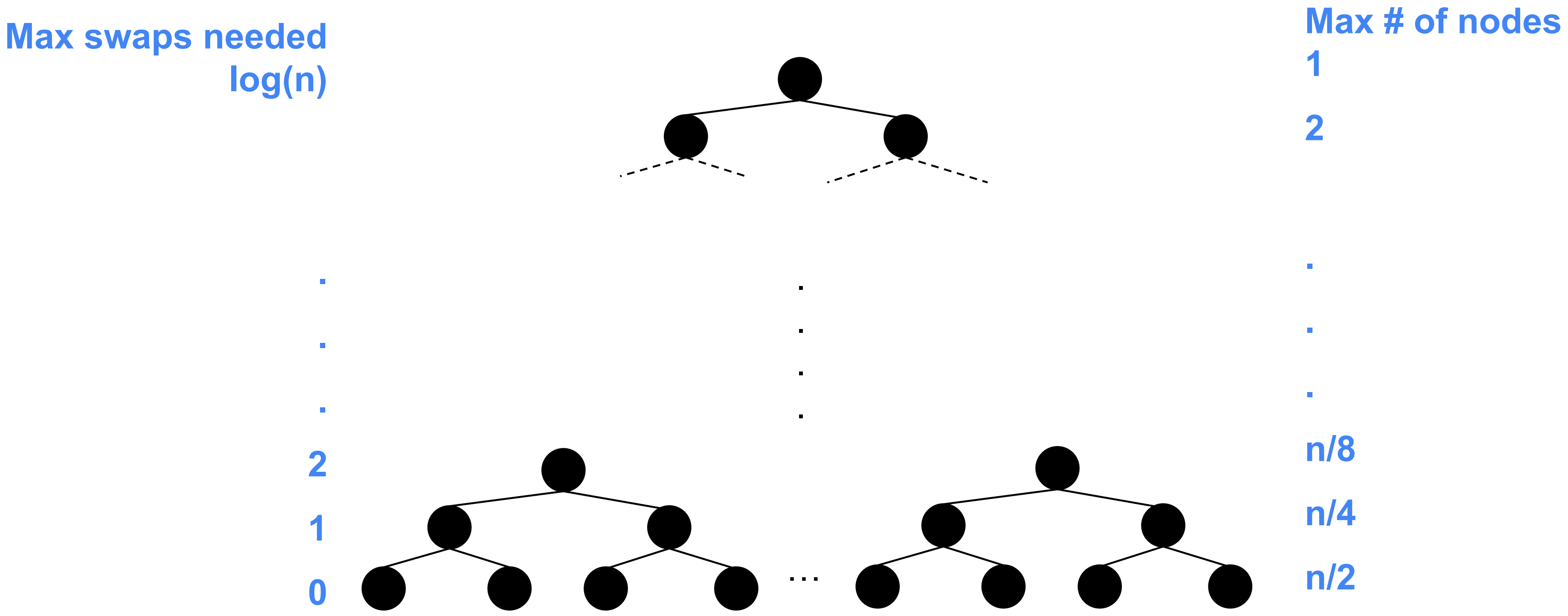
Heaps – bottom up construction



Heaps – bottom up construction



Heaps – bottom up construction



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

Heaps – bottom up construction

T = n/4 * 1 + n/8 * 2 + n/16 * 3 + … + 1 * logn

Consider T/2:

T/2 = n/8 * 1 + n/16 * 2 + … n/32 * 3 + …

Then

T - T/2 = n/4 + n/8 + n/16 + …

T/2 = n/2

→ T = O(n)

Heaps – merge

Merge two binary heaps of size n

- Takes linear time using linear time bottom-up construction

Problem 1

Let’s consider a long, quiet country road with houses scattered very sparsely along it. We can picture the road as a long line segment, with an eastern endpoint and a western endpoint. Further, let’s suppose that, despite the bucolic setting, the residents of all these houses are avid cell phone users. You want to place cell phone base stations at certain points along the road, so that every house is within four miles of one of the base stations.

Give an efficient algorithm that achieves this goal and uses as few base stations as possible. Prove that your algorithm correctly minimizes the number of base stations.

Problem 1

Solution: Find the first house from the left (western most house) and go four miles to its right and place a base station there. Eliminate all houses covered by that station and repeat the process until all houses are covered.

Complexity of our solution will be O(n logn) since we need to sort the houses first from left to right by their x coordinates. The actual positioning of the base stations will require O(n) time

Problem 1

Proof: The proof is similar to that for the interval scheduling solution we did in lecture. We first show (using mathematical induction) that our base stations are always to the right of (or not to the left of) the corresponding base stations in any optimal solution. Using this fact, we can then easily show that our solution is optimal (using proof by contradiction). Our base stations are always to the right of (or not to the left of) the corresponding base stations in any optimal solution:

Base case: we place our first base station as far to the right of the leftmost house as possible. If the base station in the optimal solution is to its right then the first house on the left will not be covered.

Problem 1

Inductive step: Assume our k th base station is to the right of the k th base station in the optimal solution. We can now show that our k+1st base station is also to the right of the k+1st base station in the optimal solution. To prove this we look at the leftmost house to the right of our k th base station which is not covered by our k th base station. We call this house H. If H is not covered by our k th base station, then it cannot be covered by the k th base station in the optimal solution since our base station is to the right of it. Our k+1st base station is placed 4 miles to the right of H. If the (k + 1)th base station in the optimal solution is further to the right of our (k + 1)th base station, then H is not going to be covered by neither the k th base station nor the (k + 1)th base station in the optimal solution so the (k + 1)th base station in our solution must also be to the right of the k+1st base station in the optimal solution.

Problem 1

Inductive step cont’d.

Now assume that our solution required n base stations and the optimal solution requires fewer base stations. We now look at our last based station. The reason we needed this base station in our solution was that there was a house to the right of our $(n-1)$th base station that was not covered by it. We call this house H. If H is not covered by our $(n-1)$th base station, then H will not be covered by the $(n-1)$th base station in the optimal solution either (since our base stations are always to the right of the corresponding base stations in the optimal solution). So, the optimal solution will also need another base station to cover H.

Problem 2

Your friend is working as a camp counselor, and he is in charge of organizing activities for a set of campers. One of his plans is the following mini-triathlon exercise: each contestant must swim 20 laps of a pool, then bike 10 miles, then run 3 miles. The plan is to send the contestants out in a staggered fashion, via the following rule: the contestants must use the pool one at a time. In other words, first one contestant swims the 20 laps, gets out, and starts biking. As soon as this first person is out of the pool, a second contestant begins swimming the 20 laps; 2 as soon as he or she is out and starts biking, a third contestant begins swimming, and so on. Each contestant has a projected swimming time, a projected biking time, and a projected running time. Your friend wants to decide on a schedule for the triathlon: an order in which to sequence the starts of the contestants. Let’s say that the completion time of a schedule is the earliest time at which all contestants will be finished with all three legs of the triathlon, assuming the time projections are accurate.

What is the best order for sending people out, if one wants the whole competition to be over as soon as possible? More precisely, give an efficient algorithm that produces a schedule whose completion time is as small as possible. Prove that your algorithm achieves this.

Problem 2

Solution:

Sort the athletes by decreasing biking + running time.

Proof:

The proof is similar to the proof we did for the scheduling problem to minimize maximum lateness. We first define an inversion as an athlete i with higher (bi + ri) being scheduled after athlete j with lower (bj + rj). We can then show that inversions can be removed without increasing the competition time. We then show that given an optimal solution with inversions, we can remove inversions one by one without affecting the optimality of the solution until the solution turns into our solution.

Problem 2

Proof cont’d.

Inversions can be removed without increasing the competition time. Remember that if there is an inversion between two items a and b, we can always find two adjacent items somewhere between a and b so that they have an inversion between them. Now we focus on two adjacent athletes (scheduled one after the other) who have an inversion between them, e.g. athlete i with higher (bi + ri) is scheduled after athlete j with lower (bj + rj). Now we show that scheduling athlete i before athlete j is not going push out the completion time of the two athletes i and j.

Problem 2

We do this one athlete at a time:

– By moving athlete i to the left (starting earlier) we cannot increase the completion time of athlete i

– By moving athlete j to the right (starting after athlete i) we will push out the completion time of athlete j but since the swimming portion is sequential and athlete j gets out of the pool at the same time that athlete i was getting out of the pool before removing the inversion, and since athlete j is faster than athlete i in the biking and running sections, then the completion time for athlete j will not be worse than the completion time for athlete i prior to removing the inversion.

Since we know that removing inversions will not affect the completion time negatively, if we are given an optimal solution that has any inversions in it, we can remove these inversions one by one without affecting the optimality of the solution. When there are no more inversions, this solution will be the same as ours, i.e. athletes sorted in descending order of biking+running time. So our solution is also optimal.

Problem 3

The values 1, 2, 3, . . . , 63 are all inserted (in any order) into an initially empty min-heap. What is the smallest number that could be a leaf node?

Solution: Since there are 2^6–1 elements in the heap, the heap will consist of a full binary tree with 6 levels. So, the smallest possible value at a leaf node would be 6.

Problem 4

Given an unsorted array of size n. Devise a heap-based algorithm that finds the k smallest elements in the array. What is its runtime complexity?

Solution: Create a max heap of size k. Keep inserting into the heap and when size of heap is k+1, pop out the highest element from the heap. At the end k elements will remain which would be the smallest k elements.

Time Complexity: $O(n \log k)$

Problem 5

Suppose you have two min-heaps, A and B, with a total of n elements between them. You want to discover if A and B have a key in common. Give a solution to this problem that takes time O(n log n) and explain why it is correct. Give a brief explanation for why your algorithm has the required running time. For this problem, do not use the fact that heaps are implemented as arrays; treat them as abstract data types

Problem 5

Solution: We can compare the element at the top of A (TA) with the element at the top of B (TB).

If TA < TB   Extract Min (A)

Elseif TA > TB   Extract Min (B)

Else A must be equal to B. We have found a common key

We repeat the above until we either find a common key or one of the heaps is empty. The complexity of the solution is O(n log n) since at each iteration we do two extract min operations at a cost of log n and there could be at most O(n) iterations.

Problem 6

Fractional Knapsack: Suppose you have a knapsack with a weight capacity of W. We are given an input of n objects, each with a respective weight, wi , and value, vi . You can take fractions of items. Provide an algorithm to fill the knapsack fully with objects such that the value of the items in the knapsack is maximized. Return this maximum value and prove your algorithm's correctness.

Problem 6

Solution: We can iterate through all n of the objects and form a ratio, r_i , by dividing value, v_i , over weight w_i . Because we want the most value possible in our knapsack, we want to maximize the ratios taken. Thus, we should sort our n ratios in non-ascending order such that r_1 > r_2 >, ..., > r_n.

Then we can loop through from i = 1...n and consider two options. If the entire weight of the item w_i is less than W (the weight capacity of our knapsack), we should add the whole thing. Else, we should add the fraction of the item (and its value) such that the total weight in the knapsack equals W.

Problem 6

Algorithm:

```
Int curr_value = 0
Int curr_weight = 0
For (i in 1…n):
    If w_i + curr_weight < W:
        Curr_weight += w_i
        Curr_value = v_i
    Else:
        Curr_value += ((W - curr_weight)/w_i) * v_i
        Curr_weight = W
        Break
    Return curr_value
```

Runtime: O(nlogn) ⟶ Dominated by sorting

Problem 6

Proof:

Greedy Stays Ahead: We can prove inductively by Greedy Stays Ahead that our greedy algorithm is correct. Let's say there exists some optimal solution called OPT that does not necessarily follow the same steps as our greedy. We'll prove that our greedy is as good, if not better in all instances. In our base case, we have either 0 or 1 item. Both of these are vacuously true as with 0, the empty set of items is the result for both algorithms, and for 1 item, it either exceeds W, in which case we take the proper fraction or doesn't, in which case we take the whole thing for both algorithms.

Problem 6

Proof cont’d.

Now, we can apply induction and assume that by the k th item chosen in our knapsack problem by greedy, our total value will be as good, if not better, than OPT. Let’s assume that the (k − 1)th element chosen in greedy is optimal at this point. By choosing the (k − 1)th item in the previous iteration in greedy, we were choosing the item with the (k − 1)th best “value”, meaning most value per unit of weight. We do the same thing with the k th item at the k th 4 iteration and see two options: either we take the whole thing (enter if statement) or we take a fraction of it (enter else).

If we enter the if statement, it means we are taking the entire item that provides us the most value for least weight at this point (k th), which is the ratio we want to optimize. Thus, that is the correct choice, and we should proceed. Similarly, if we enter the else statement, we take a fraction of this item due to weight constraints. However, this item is still the k th most value at the k th iteration, so taking an equivalent fraction of weight and value will still give us the most possible, which is at least as much as the most OPT could give. Thus, induction is shown in both cases. QED