Greedy

Fractional Knapsack

Problem: You are a thief who breaks into a bank. Inside the vault, you find n piles of dust with weights $w_1, w_2, \ldots, w_n$ and potential profits $p_1, p_2, \ldots, p_n$. Your sack has total weight capacity W. How much of each pile do you steal so as to maximize your profit?

Example: $W = 15$ and

Pile	Weight	Profit
1	12	4
2	1	2
3	2	2
4	1	1
5	4	10

Fractional Knapsack

Algorithm: Arrange the piles in decreasing order of profit per weight. Continue stealing as much of the most valuable pile as you can until you run out of dust or your sack fills up.

Example: $W = 15$ and

Pile	Weight	Profit	Profit/Weight
1	12	4	$\frac{1}{3}$
2	1	2	2
3	2	2	1
4	1	1	1
5	4	10	$\frac{5}{2}$

Answer: Take all of piles 2, 3, 4, and 5, and 7 units of pile 1.

But how do we know *this* algorithms is correct? (Another greedy algorithm that *doesn't* work: Try arranging the piles in order of decreasing profit.)

Fractional Knapsack

Pile	Weight	Profit	Profit/Weight
1	12	4	$\frac{1}{3}$
2	1	2	2
3	2	2	1
4	1	1	1
5	4	10	$\frac{5}{2}$

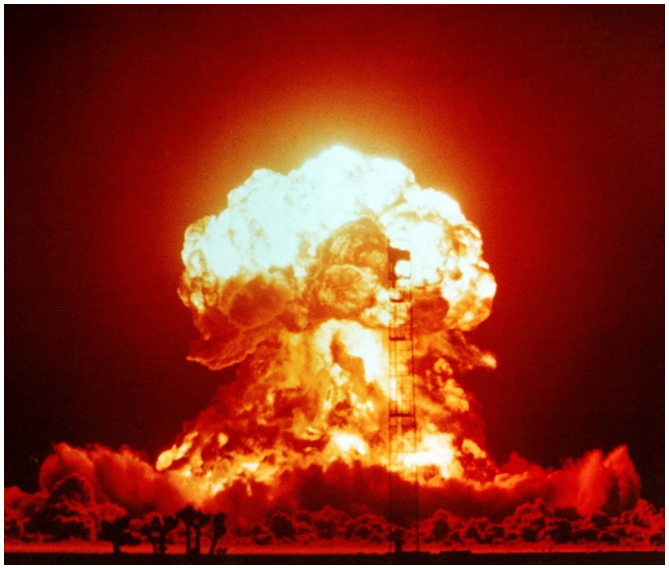
Claim: The algorithm (where rows with the same profit/weight are merged together) finds the optimal quantities to steal.

Proof: Let OPT be the quantities that the optimal algorithm takes, and let GREEDY be the quantities obtained by the greedy algorithm.

Assume that OPT and GREEDY are *different*.

Let pile i be the most valuable pile (as measured in profit/weight) where they are different. Which algorithm must take more of pile i? GREEDY

Let $\Delta w_i > 0$ be the difference in quantity of pile i that GREEDY took more than OPT. Let $\Delta r_i > 0$ be the difference in profit/weight between pile i and the next most valuable pile. Then OPT could have made at least $\Delta w_i \Delta r_i > 0$ more money if he had stolen pile i intead of some less valuable material.



Making Change

Motivation: Using standard American currency (no 50¢ pieces), make change for $3.68.

Answer: Three $1, two 25¢, one 10¢, one 5¢, and three 1¢

Ask yourself: Were you thinking about giving back 368 pennies? No

Problem: Given an infinite amount of coins of a particular currency, determine how to give back change for n using the minimum of coins.

Making Change

Problem: Given an infinite amount of coins of a particular currency, determine how to give back change for n using the minimum of coins.

Algorithm: Continue giving back the largest denomination less than or equal to what is owed.

Does the greedy algorithm always work?

Exercise: Consider an alien currency that consists of 7¢, 6¢, and 1¢ pieces. Does the greedy algorithm give back the minimum number of coins for 18¢ ?

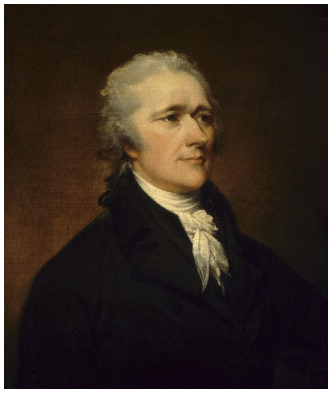
Answer: No

GREEDY: 6 coins (two 7¢ and four 1¢) vs. OPT: 3 coins (three 6 ¢)

Exercise: Consider what might happen if you ran out of nickels and had to make change for 40¢. Is the greedy algorithm optimal?

Answer: No

GREEDY: 7 coins (one 25¢, one 10¢, five 1¢) vs. OPT: 4 coins (four 10¢)



Interesting fact: Though it is not true for all worldwide currencies, if you use the standard American currency denominations, the greedy algorithm *will* always produce the optimal number of coins. There is historical evidence to suggest that not only did the designers of the currency know it, they may have specifically designed it that way.

Making Change

Problem: Assume that you are given a currency with a coin for all powers of 2 (1¢, 2¢, 4¢, 8¢, etc.). Does the greedy algorithm for making change produce the optimal number of coins?

Example: 37¢=32¢+4¢+1¢

Assume that OPT gives back the optimal number of coins and that, for some amount n, OPT and GREEDY give back change differently.

Let 2^x¢ be the largest denomination in which OPT and GREEDY give back a different number of coins.

Question: Can OPT ever give back 2 or more coins of a particular 2^i¢ denomination? No. Can GREEDY? No. So for every denomination, OPT and GREEDY must either give back 0 or 1 of that denomination.

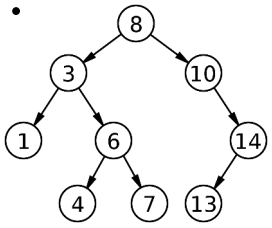
Math fact: $2^x - 1 = 1 + 2 + 4 + \ldots + 2^{x-1}$

Question: If OPT and GREEDY give back a different number of 2^x¢ coins, then one must have given back 0 and the other must have given back 1. Is that possible? No.

Interesting fact: Inside the CPU, numbers are represented by a sequence of bits inside registers. Generally, a 1 bit indicates the presence of a voltage (or something being *on* as opposed to *off*). Given the results to the left, why is this an extremely intelligent/efficient way to represent numbers as opposed to, say, making the 0 bits represent *on*?

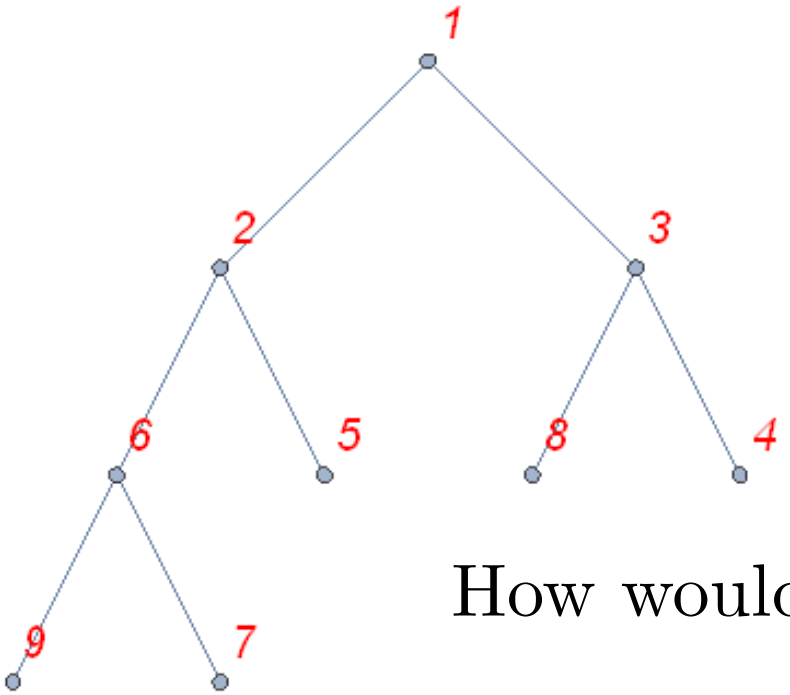
Heaps

Motivation: We require a fast data structure to store keys such that the smallest key is always quickly available. The data structure should support MAKE, INSERT, DELETE-MIN, and FIND-MIN (but not FIND).



Your first thought: Balanced binary search trees

Heap Rule: Any children of a node must be larger than the node itself.

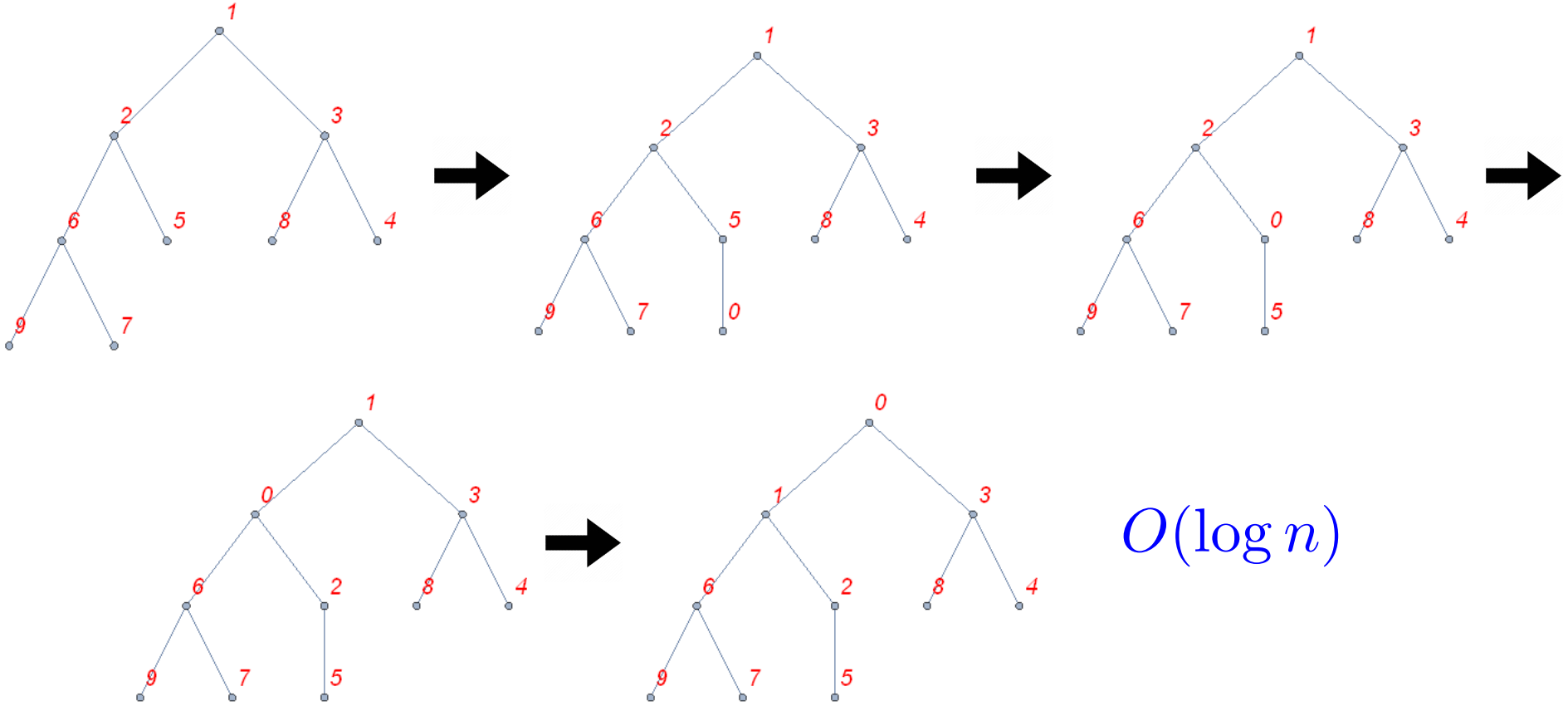


How would you FIND-MIN?

$O(1)$

Heaps

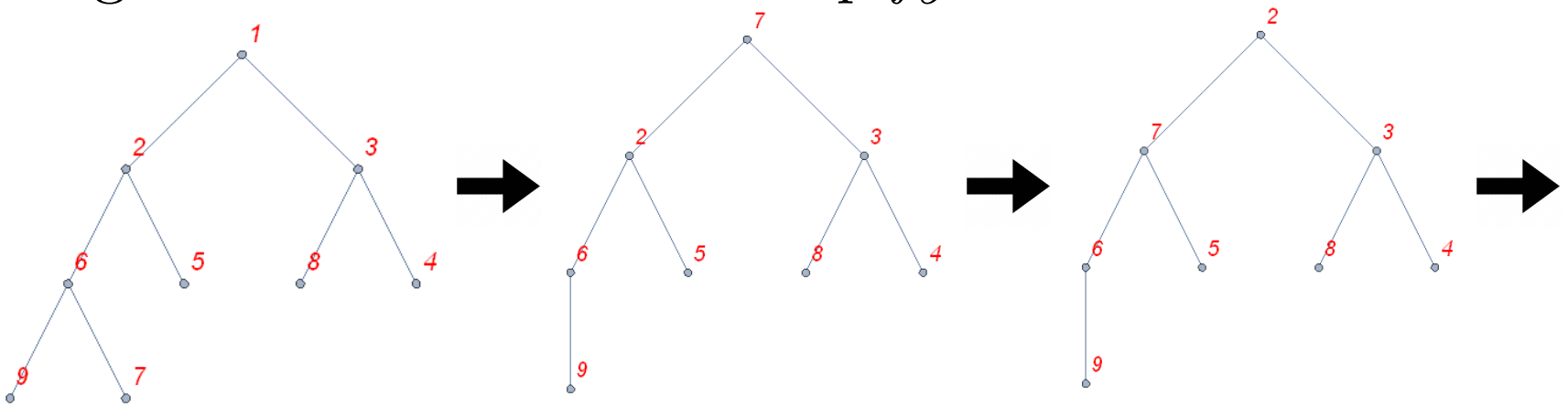
INSERT: To insert a node into a heap, place the new key into the bottom-most rightmost empty slot and then *heapify* up.

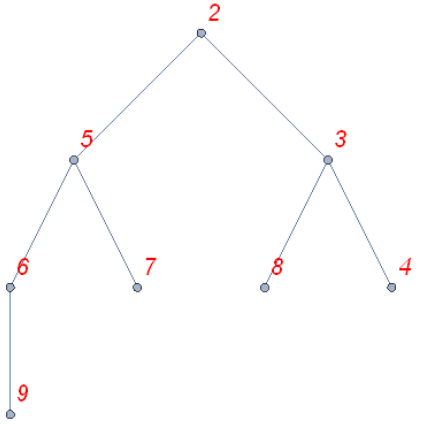


$O(\log n)$

Heaps

DELETE-MIN: Replace the root with the bottom-most, rightmost element and *heapify* down.



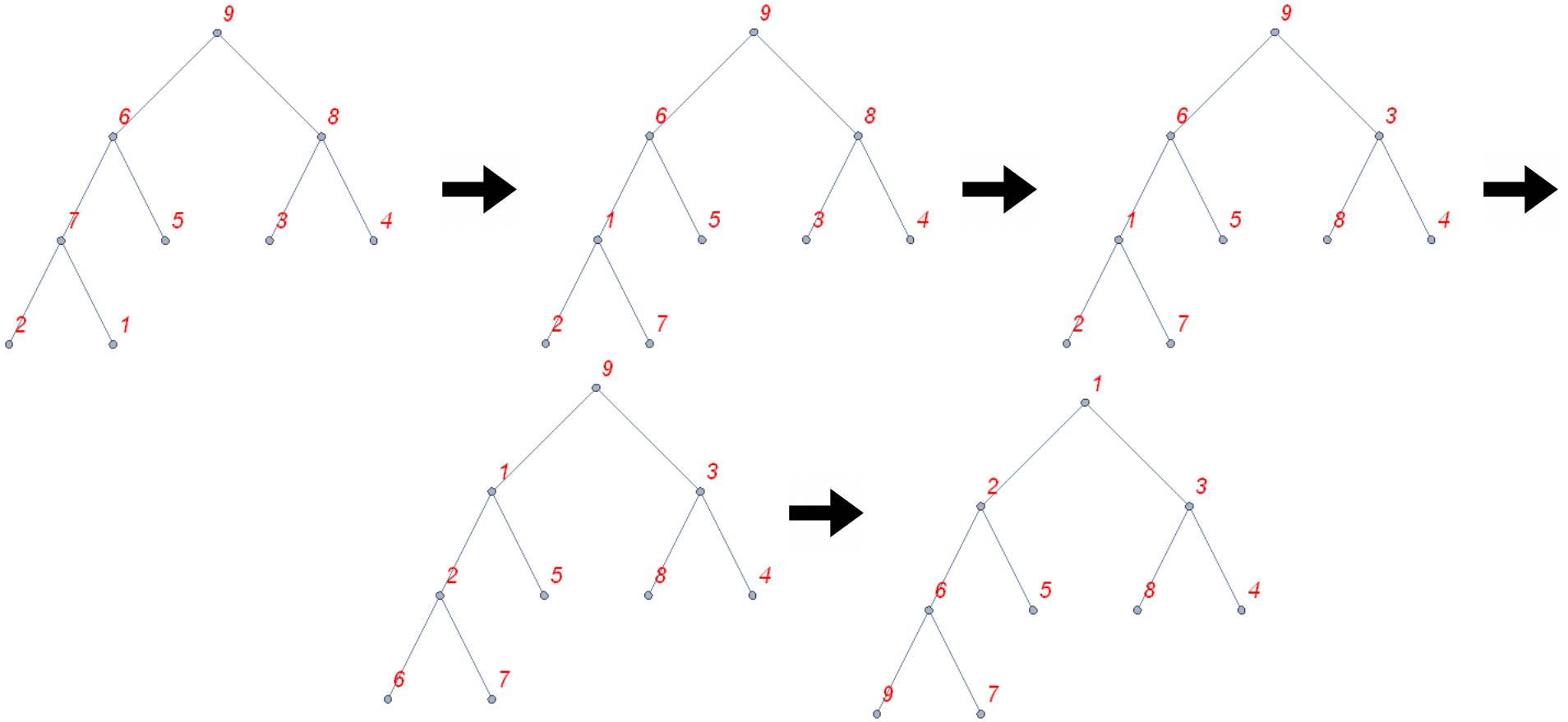


$O(\log n)$

Heaps

MAKE: You are given a list of n unsorted elements. Number the elements from 1 to n and form a binary tree out of them by having element i point downward to elements $2i$ and $2i+1$. Starting from the bottommost rightmost and continuing upwards (like a reverse-typewriter), *heapify* down.

Example: Assume that we want to call make on the following input: $(9, 6, 8, 7, 5, 3, 4, 2, 1)$



Heaps

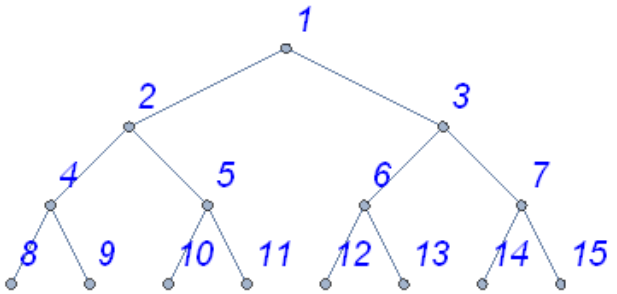
How long does it take to MAKE?

Assume that we call MAKE on n numbers. At the lowest level, there are at most $\frac{n}{4}$ nodes that require at most 1 swap. Each level we move up, there are half the number of nodes that potentially require one additional swap$\Rightarrow$

$\frac{n}{4}1 + \frac{n}{8}2 + \frac{n}{16}3 + \ldots + 1(\log_2 n - 1) =$

$\sum_{i=1}^{\log_2 n-1} \frac{n}{2^{i+1}} i = n \sum_{i=1}^{\log_2 n-1} \frac{i}{2^{i+1}} \leq n \sum_{i=1}^{\infty} \frac{i}{2^{i+1}} = \Theta(n)$

In fact, this data structure can be implemented *extremely* efficiently using *a single, continuous block of memory.*



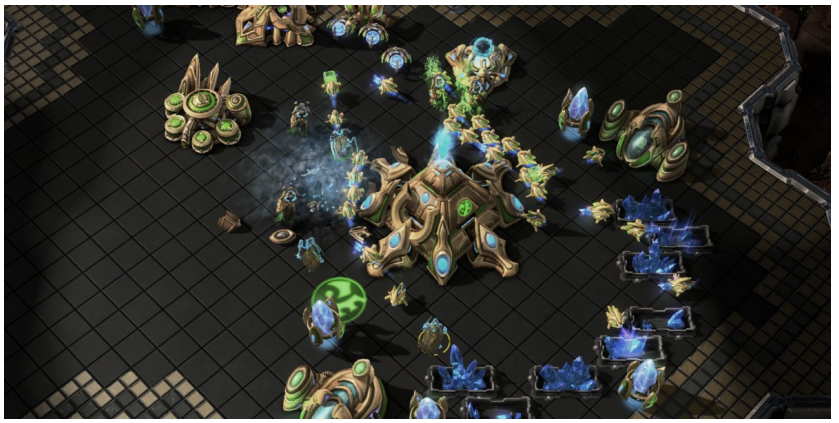
Index of parent? $\lfloor \frac{i}{2} \rfloor$

Index of left child? $2i$ Right child? $2i + 1$

Heaps

Applications:

- Heapsort
- Discrete Event Simulation
  - physical simulations (research)
  - operating systems
  - video games



- CECS 328 algorithms
- CECS 428 algorithms (more advanced versions)