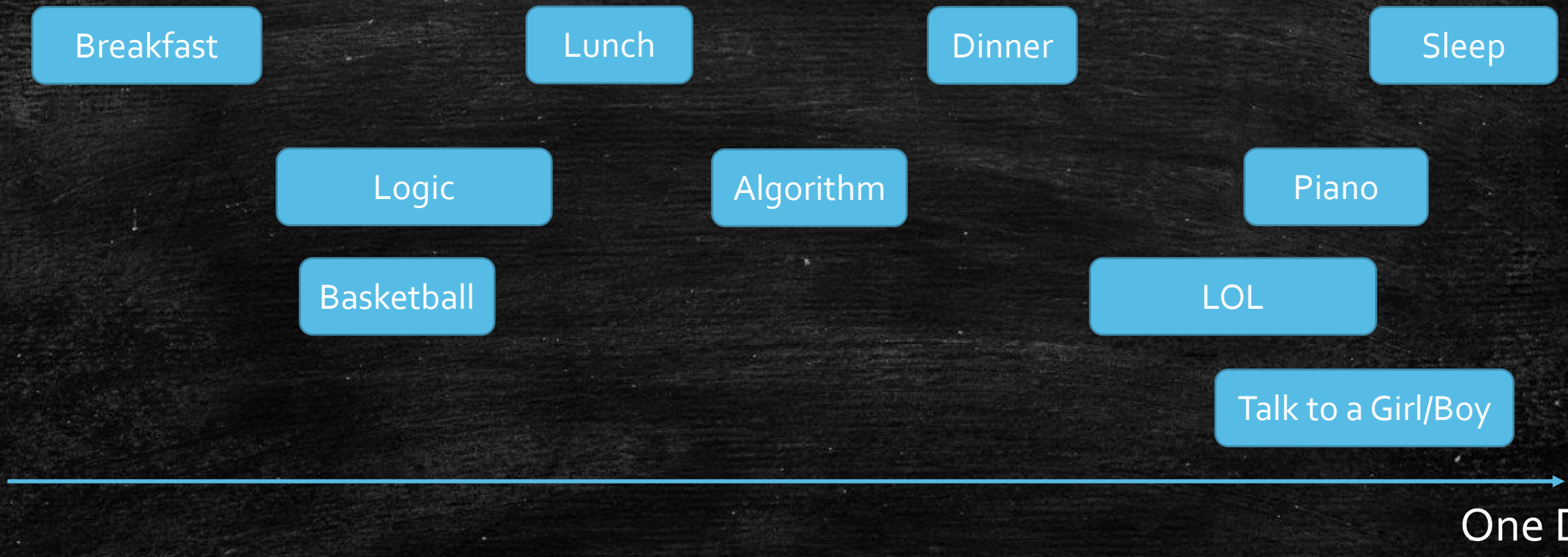
More Greedy Algorithms

How to select the correct greedy strategy.

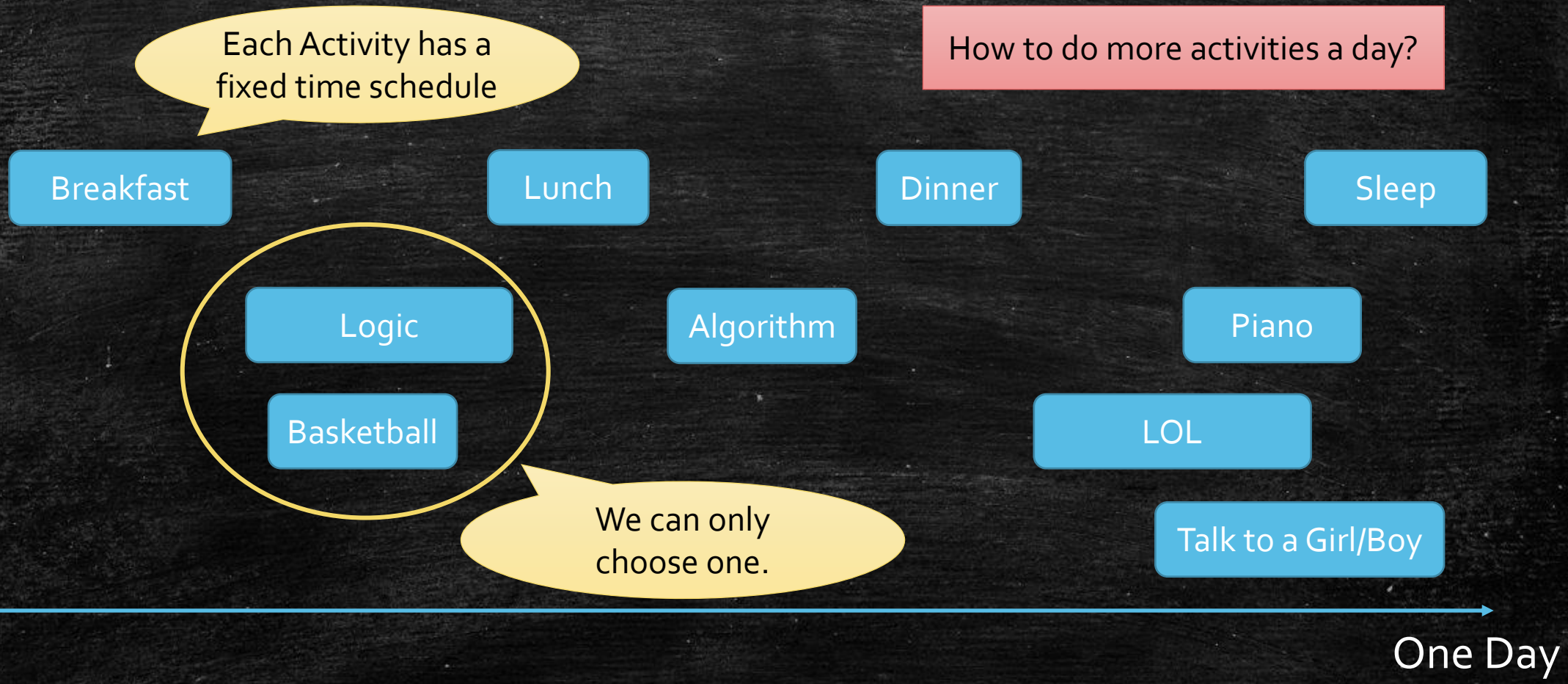
Greedy Homework Scheduling

- **Input:** n homework, each homework j has a size s_j, and a deadline d_j.
- **Output:** output a time schedule of doing homework!
- Greedy Approach
  - Keep finishing the homework with the **closest** deadline
- We have proved it is optimal!

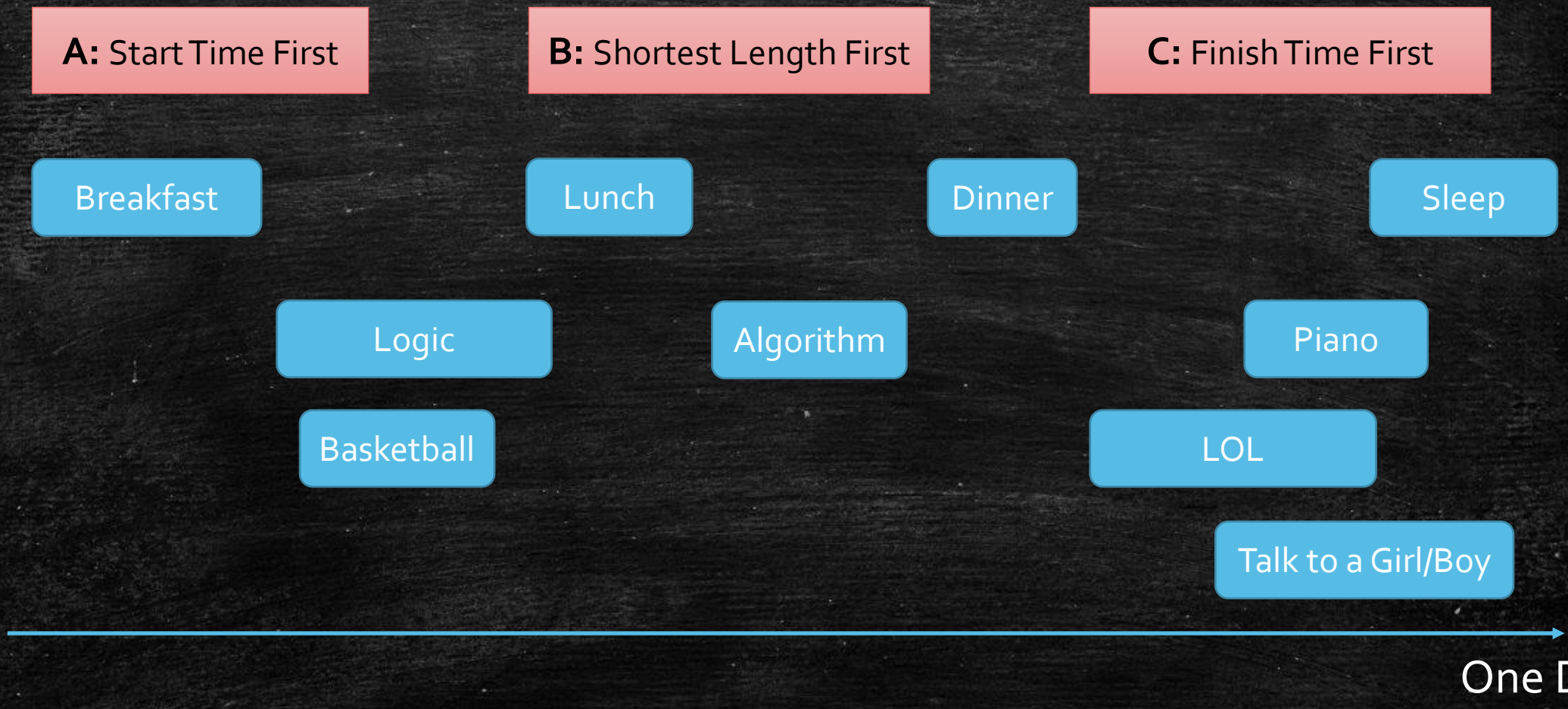
Another problem!



Let generalize the problem!

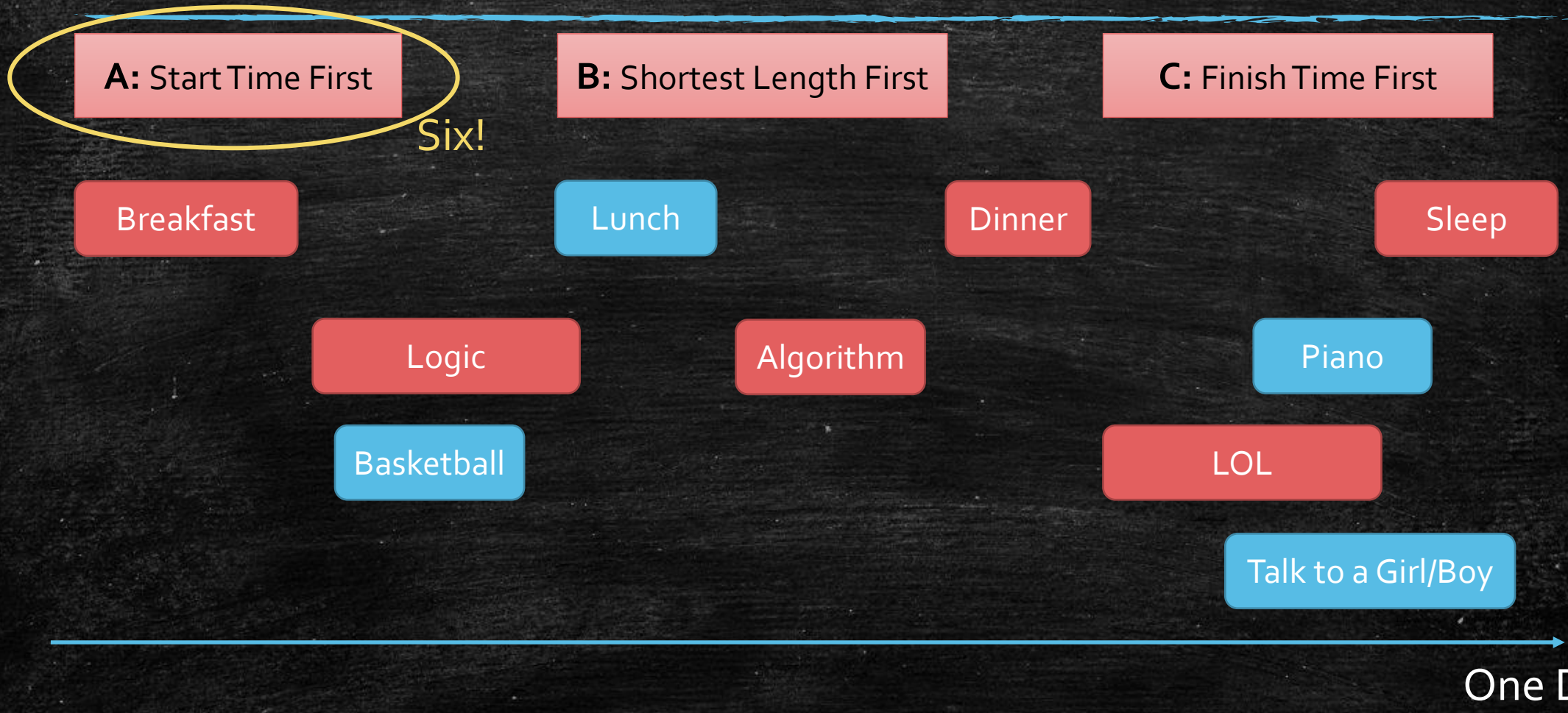


Three Greedy Ideas

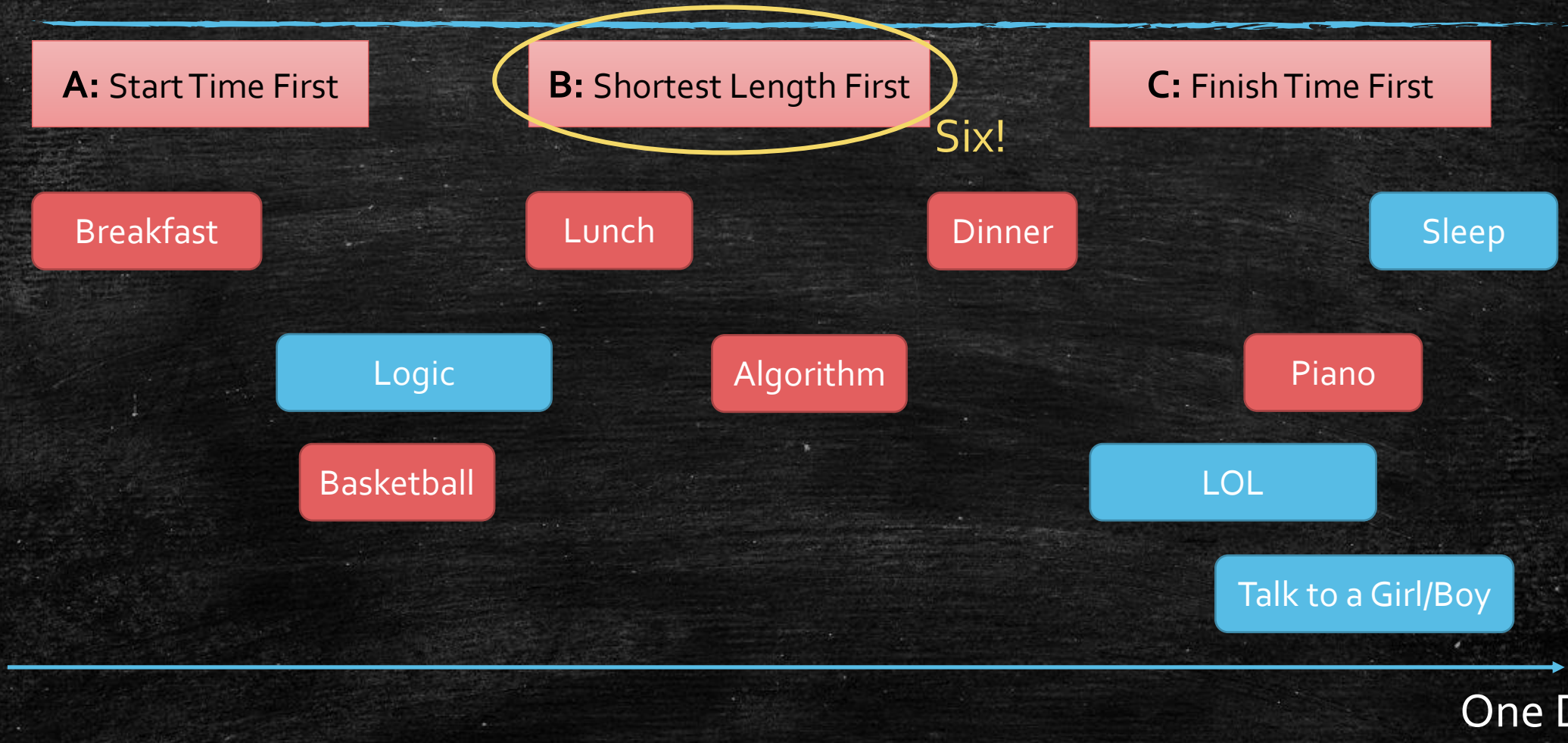


Which one is correct?

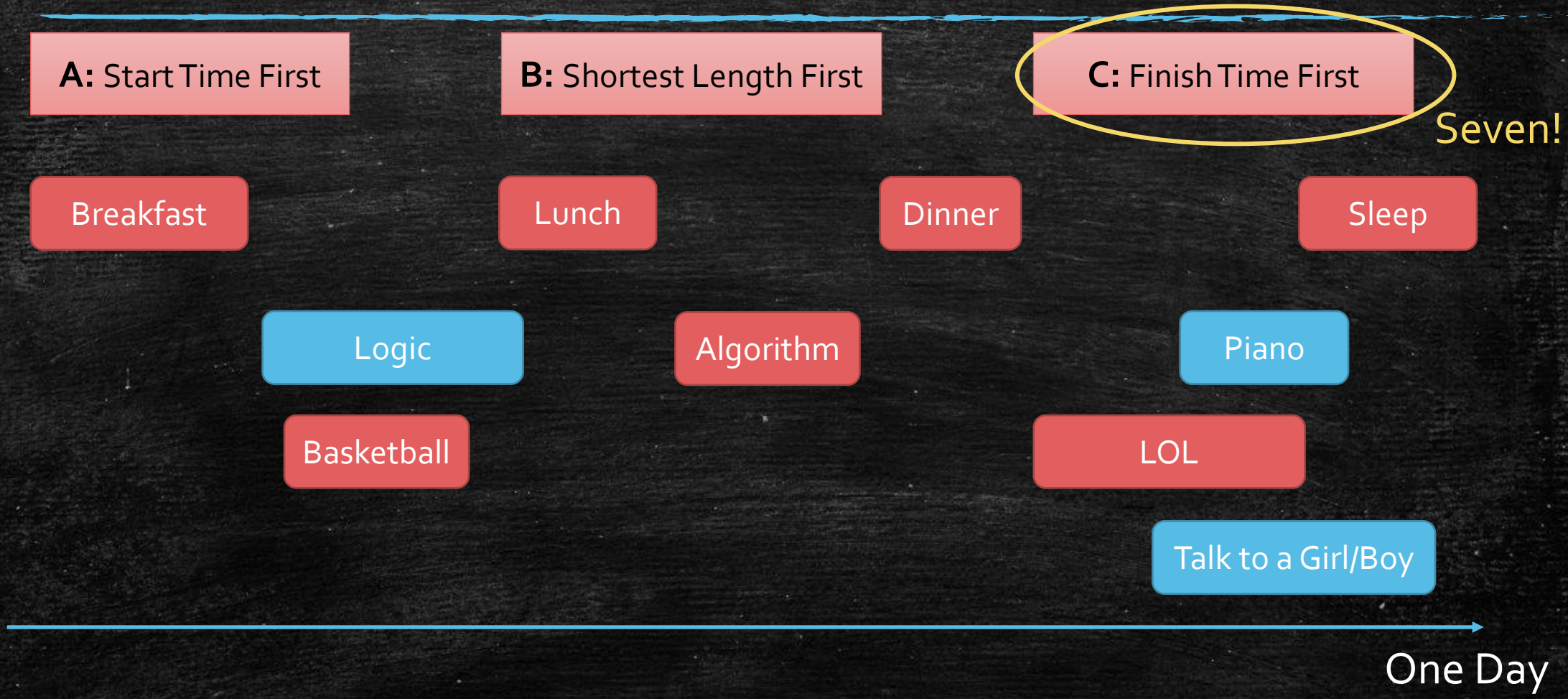
Three Greedy Ideas



Three Greedy Ideas



Three Greedy Ideas



Three Greedy Ideas

- Finish Time First is the only possible one!
- Is it correct?
- Intuition
  - Finish fast → Best for future
  - How to make a proof?

The Framework of Our Proof

- Dijkstra
  - Grow from small SPT to larger SPT
- Prime & Kruskal
  - Grow from small P-MST to larger P-MST
- We are correct if we **never** **ruin out** OPT!
- Or say: we are still in an Optimal Tunnel!

The Big Idea

The local greedy choice do not ruin out OPT

Induction

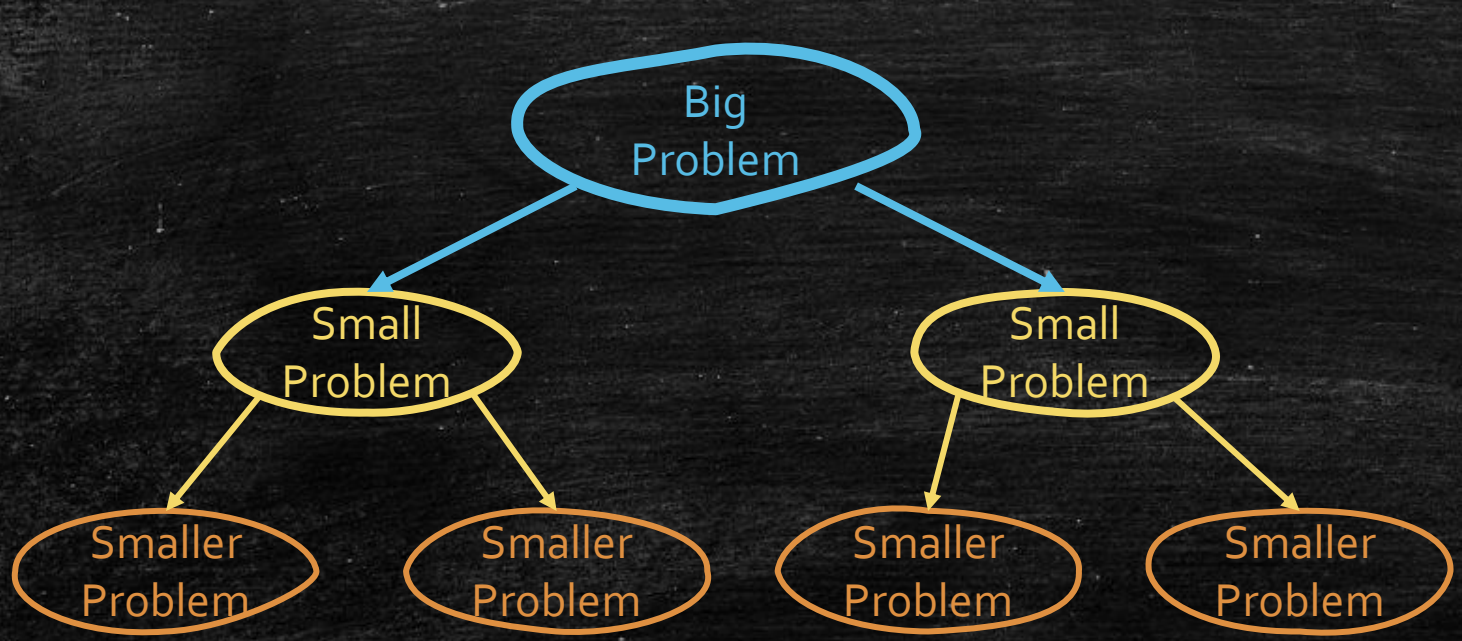
- Base step: $\emptyset$ is in an OPT.
- Hypothesis: the selected $k-1$ activities are in an OPT.
- Induction: After adding the k-th activity, we are still in an OPT.
- Conclusion: After adding the last activity, it is in an OPT. Nothing can be added, so it is OPT.

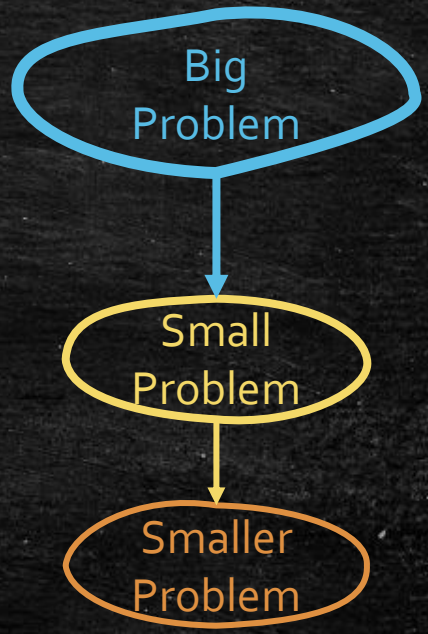
Proof of the Induction

- Hypothesis: the selected $k-1$ activities are in an OPT.
- Induction: After adding the k-th activity, we are still in an OPT.
- Can you prove it?
- Discussion!

Summarize

Divide and Conquer vs. Greedy





One more interesting Greedy!

General Question

- How to encode a book?
- Two steps:
  - Give alphabet encoding policy
  - Encode all sentences in the book

i am good at algorithms

Alphabet: Naïve Approach

a	0000
d	0001
g	0010
h	0011
i	0100
l	0101
m	0110
o	0111
r	1000
s	1001
t	1010
space	1011

- **i am good at algorithms**
- Cost Analysis
  - Each character & space: 4 digit
  - Totally: $23 \times 4 = 92$

Improvement: Why not shorter?

a	0
d	1
g	10
h	11
i	100
l	101
m	110
o	111
r	1000
s	1001
t	1010
space	1011

- **i am good at algorithms**
- Cost Analysis
  - Each character & space < 4 digit
  - Totally: $< 23 \times 4 = 92$
- Problem:
- When we decode
  - 10: is it ‘g’ or ‘da’ ?

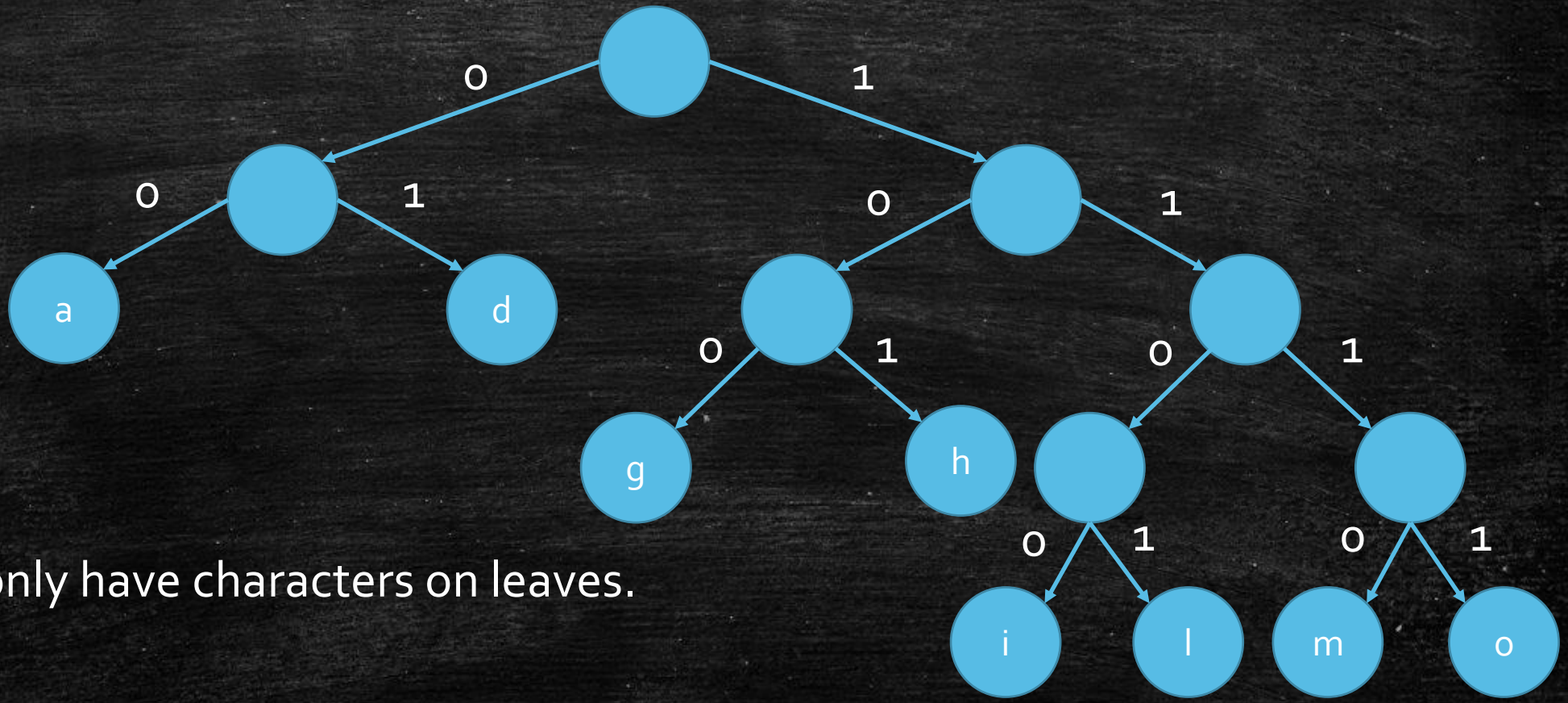
Can you conclude a property we need?

Solving the problem!

a	0
d	1
g	10
h	11
i	100
l	101
m	110
o	111
r	1000
s	1001
t	1010
space	1011

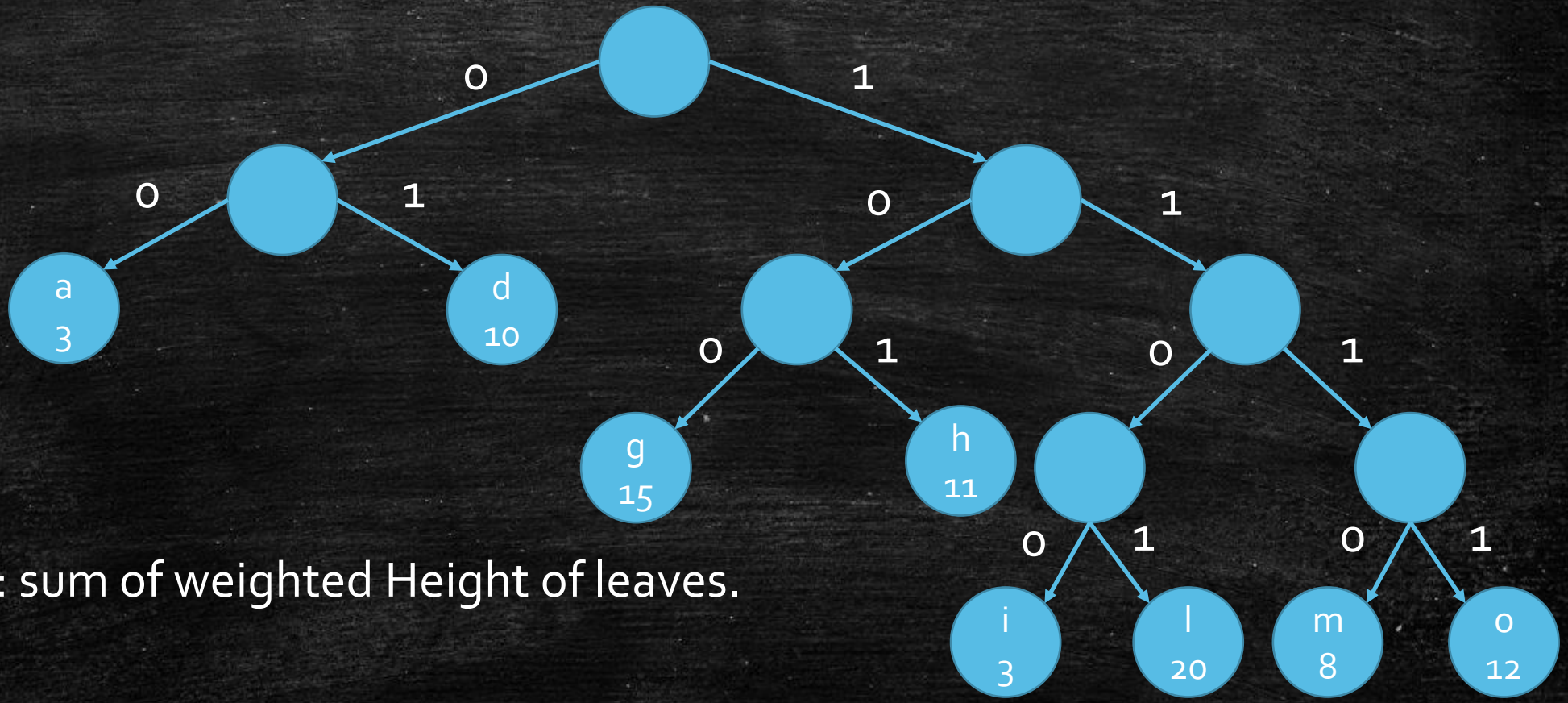
- **i am good at algorithms**
- Problem:
- When we decode
  - 10: is it ‘g’ or ‘da’ ?
- A **prefix-free** code
  - No symbol’s code is the prefix of another symbol’s code.
  - Example: ‘d’: 1 is the prefix of ‘g’: 10.
  - Think why it is good?

A prefix-free code is a tree!



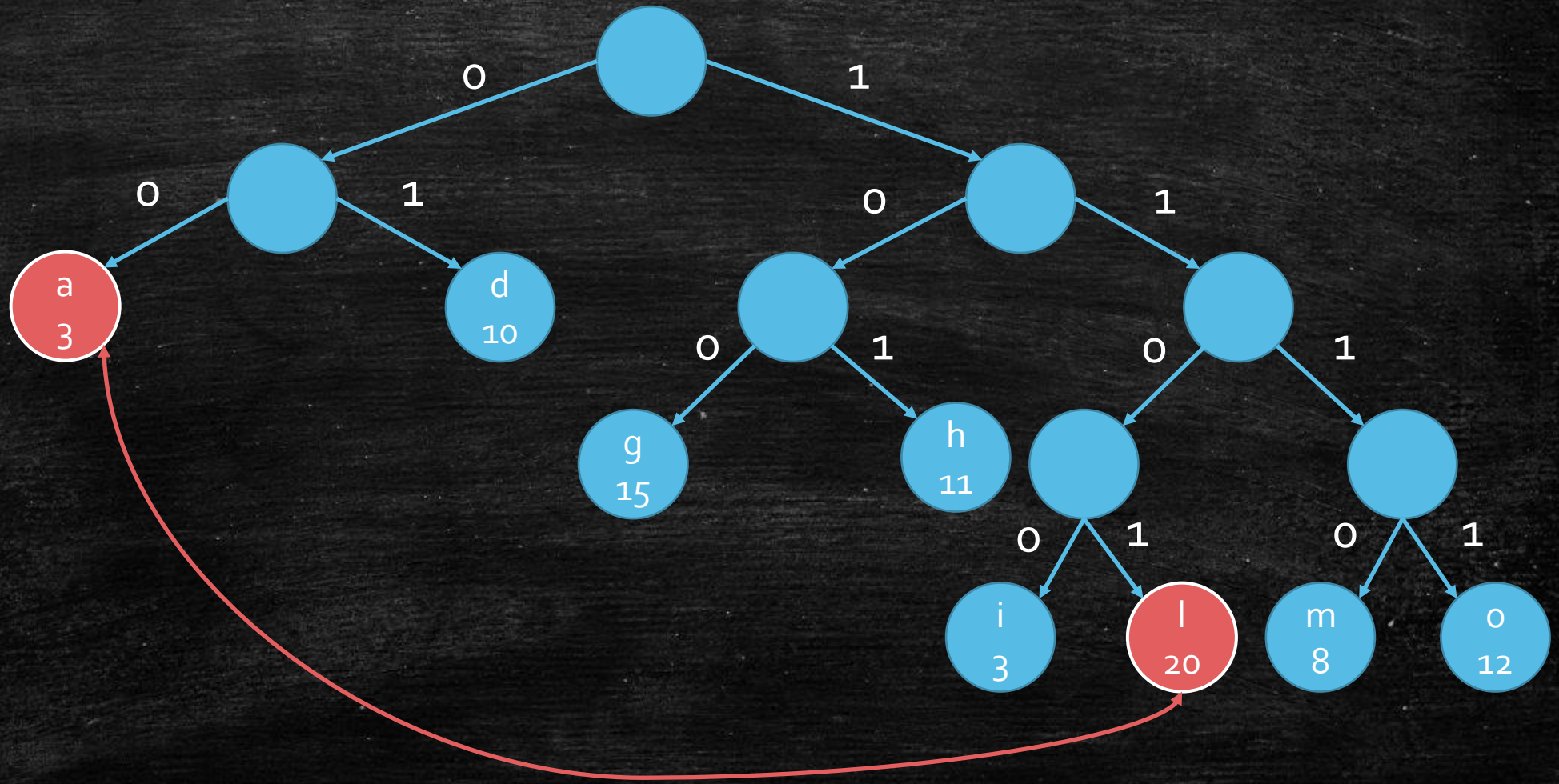
We only have characters on leaves.

Cost of A prefix-free code is a tree!



Cost: sum of weighted Height of leaves.

Minimize the cost?



The Optimization Problem

- We may learn the frequency from the history.
- **Input:** an alphabet A, a frequency function $f: A \to N$.
- Cost of a tree:
  - $lev(a)$: depth of $a \in \Sigma$. (the code length)
  - $\sum_{a \in A} lev(a) \cdot f(a)$: total cost.
- **Output:** a minimum cost prefix-free tree for A.

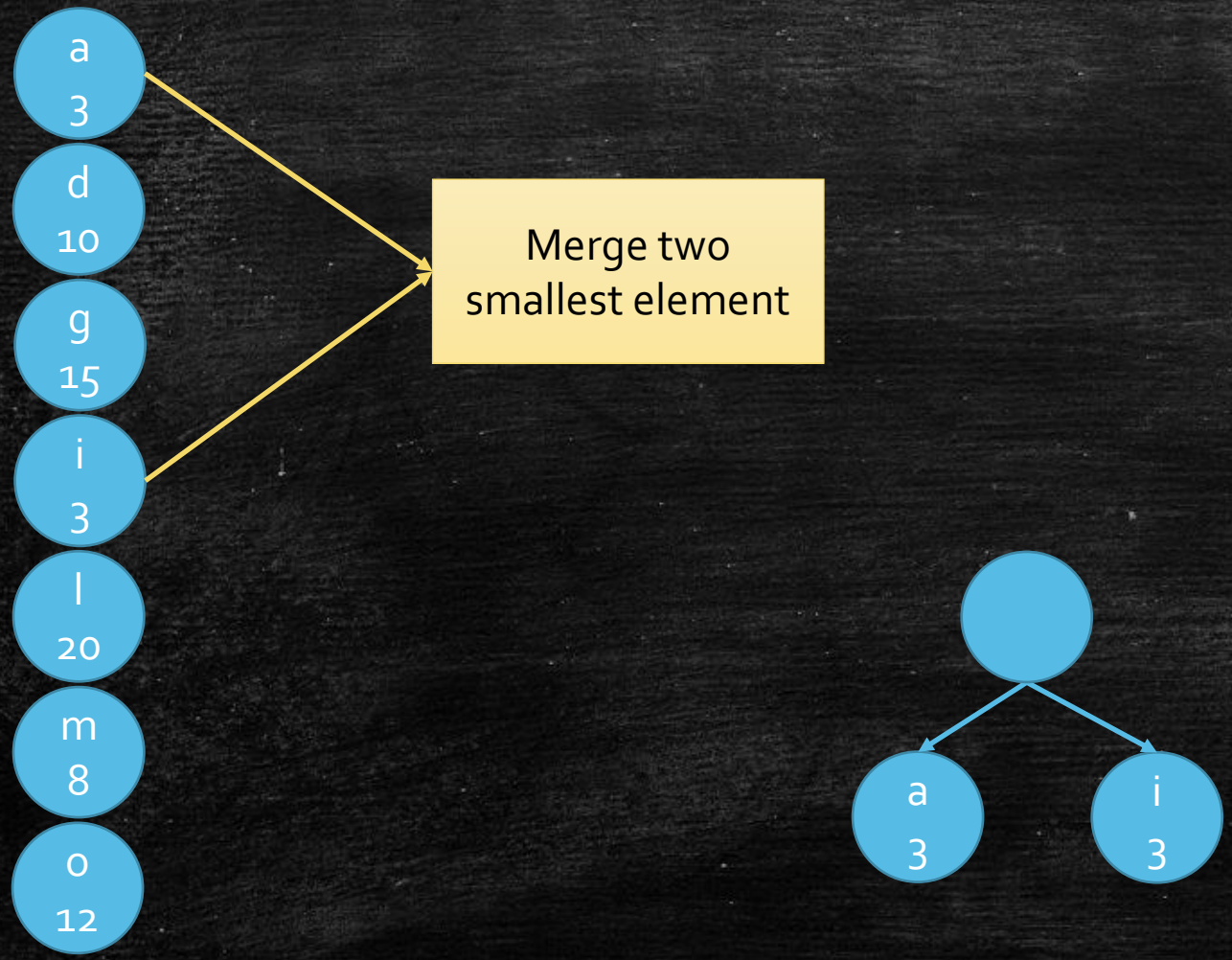
a	1
d	10
g	3
h	2
i	5
l	6
m	10
o	30
r	3
s	2
t	1
space	100

What is the greedy approach now?

- Build a tree from bottom.
- Put small cost character to bottom.
- I believe every know what is the first step!

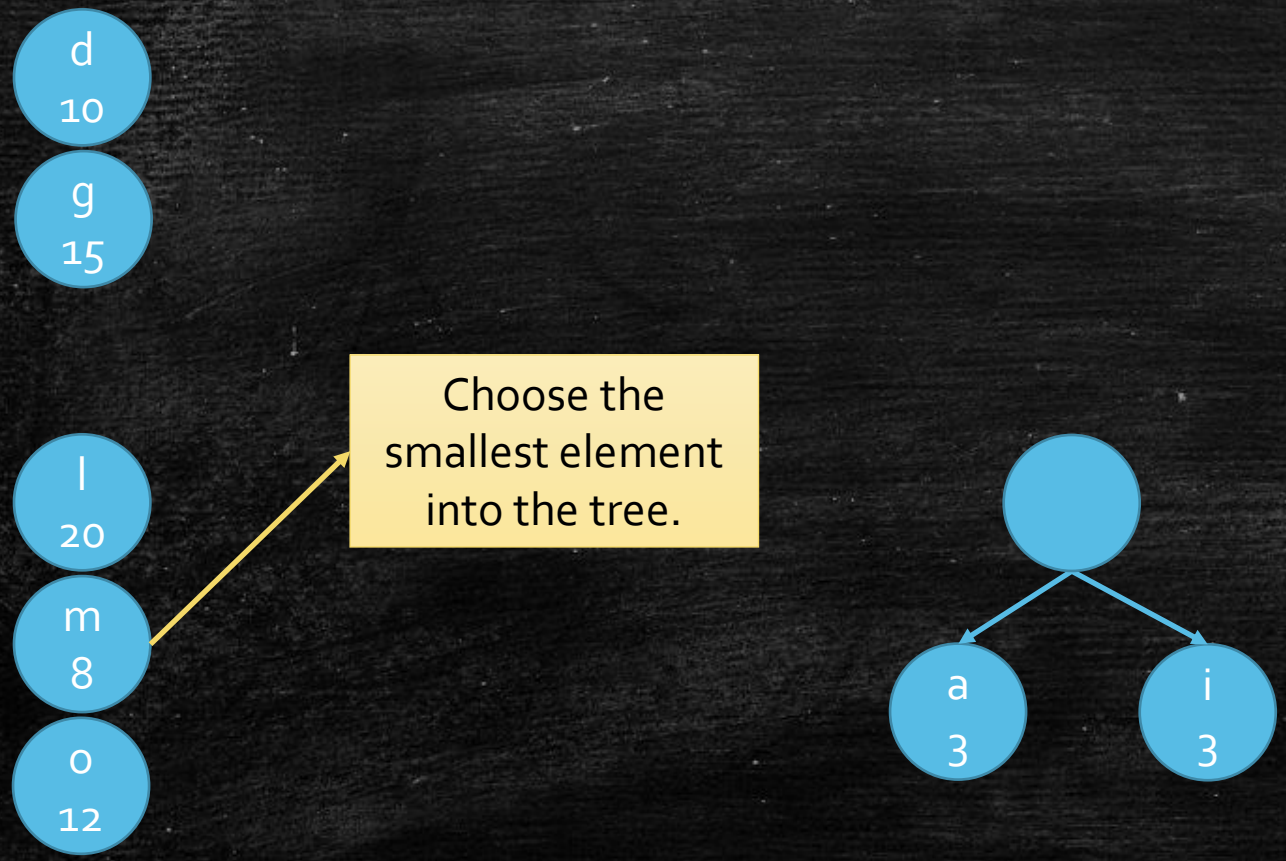
A bottom-up building

A bottom-up building

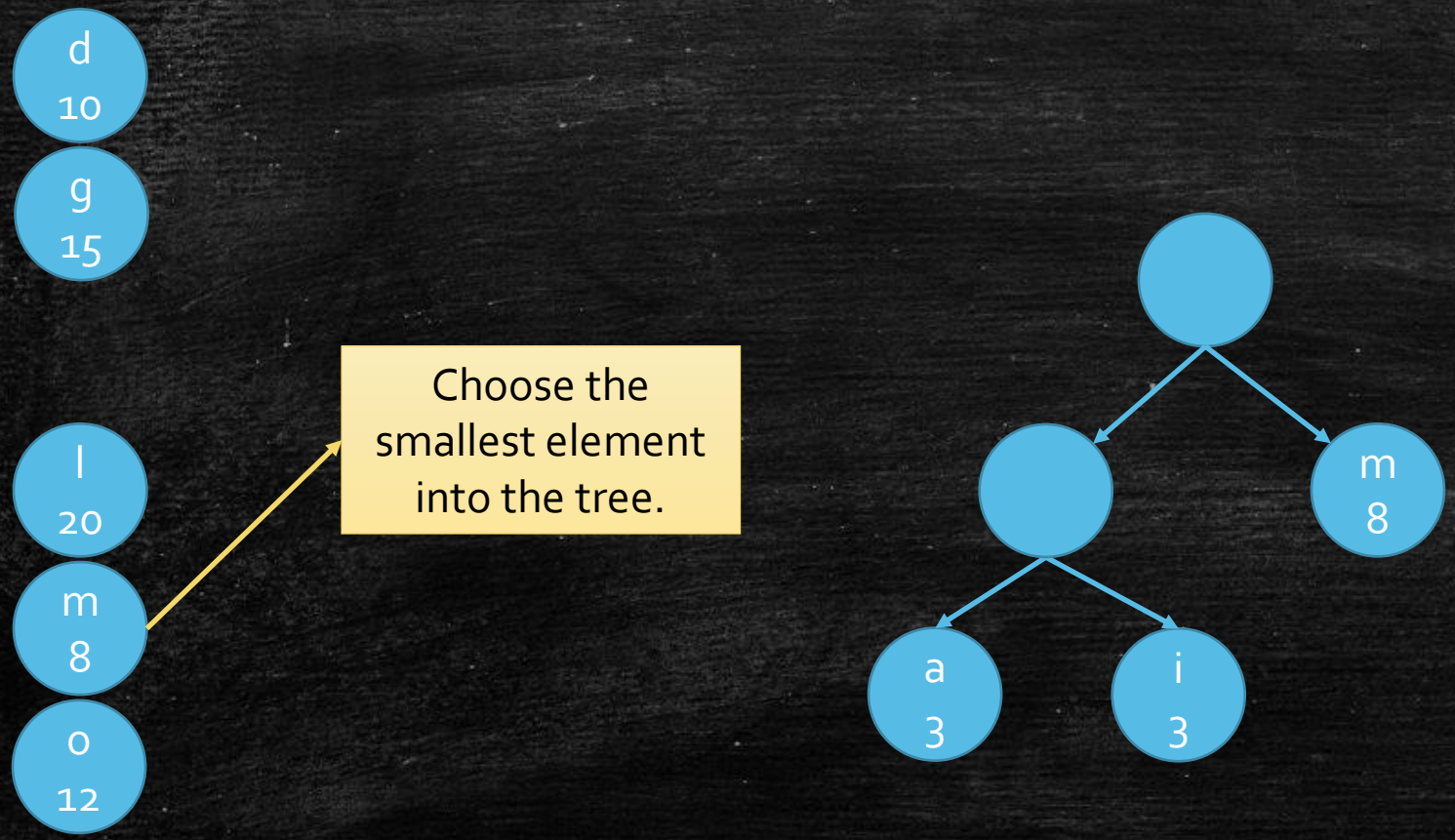


What is the next step?

Attempt One: Prim Like Greedy

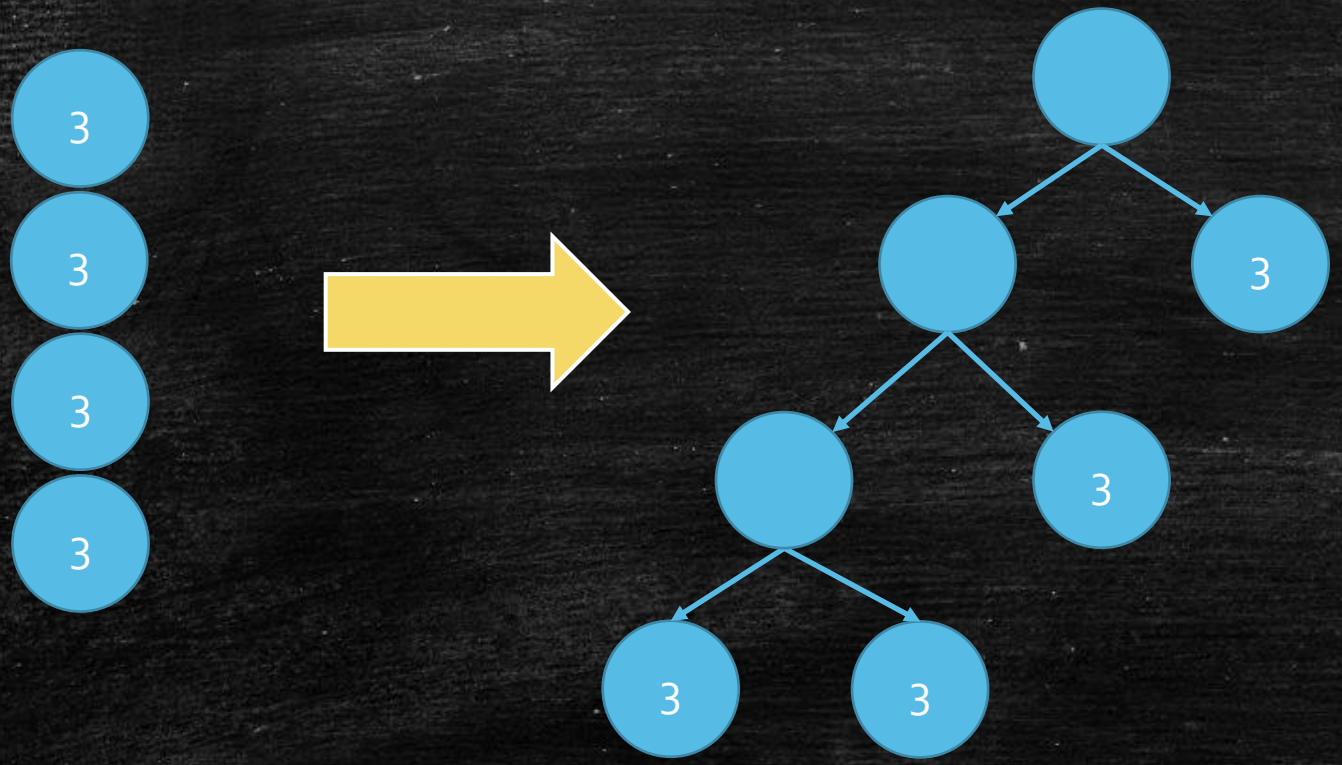


Attempt One: Prim Like Greedy

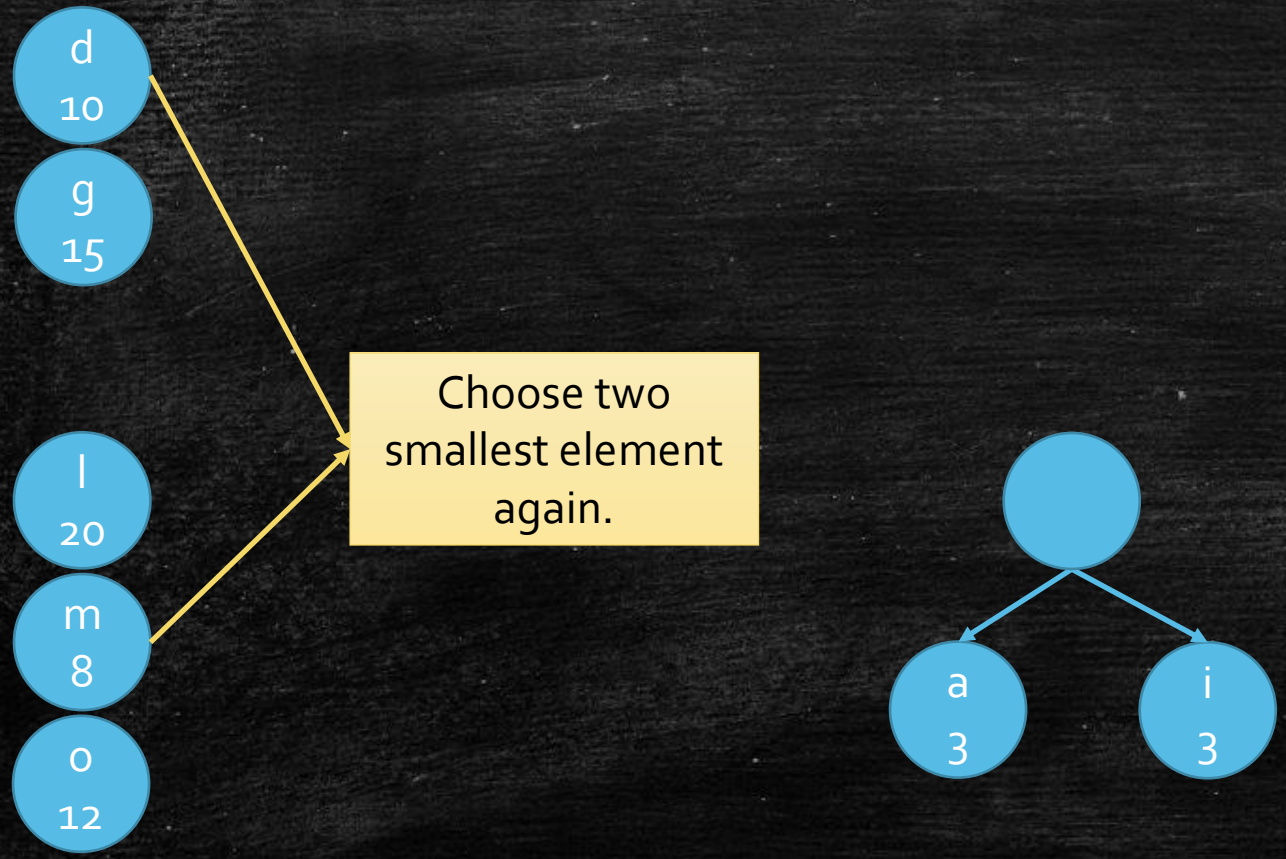


A Bad Case

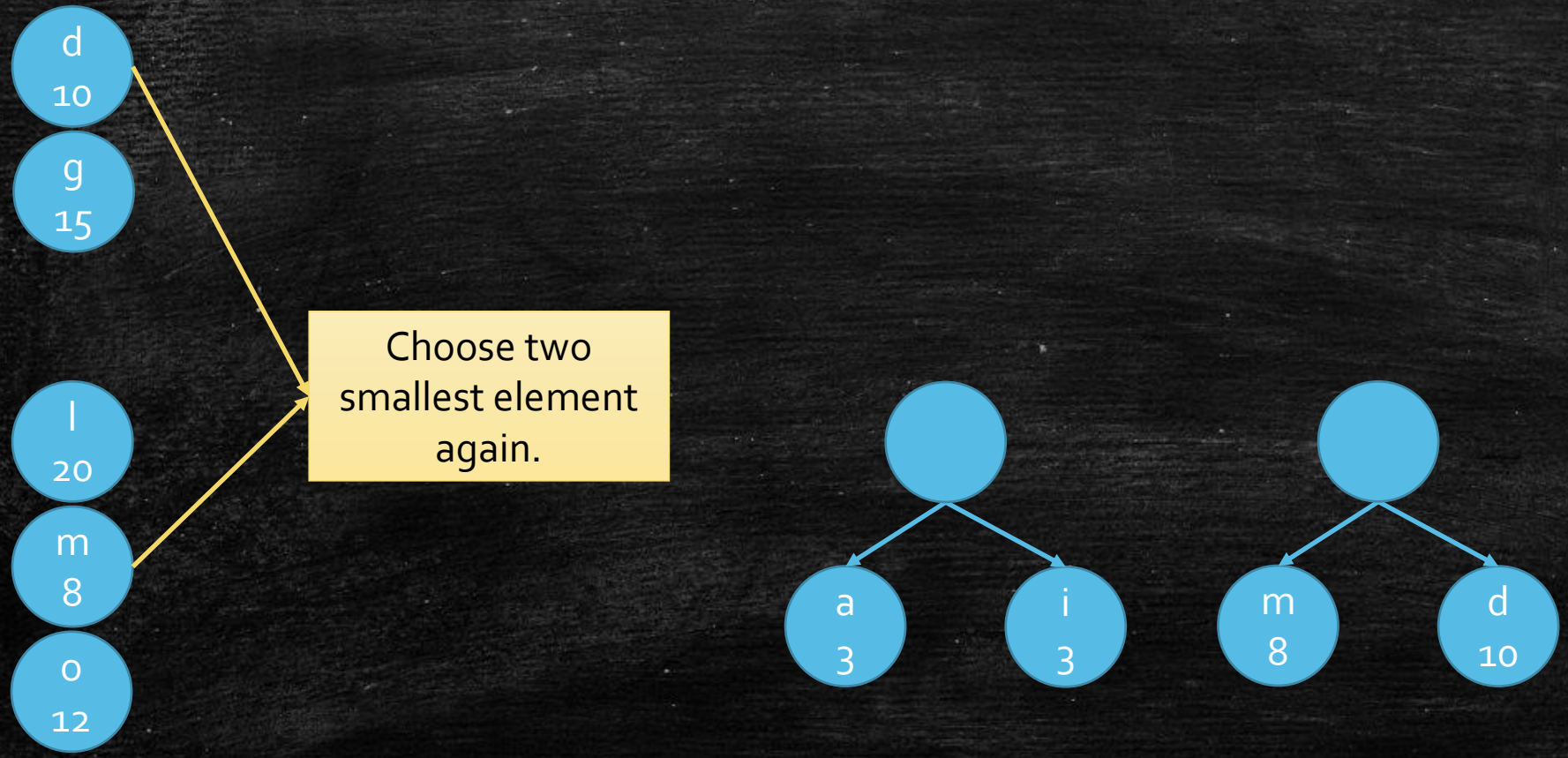
- Thinking: Is the cost minimized?



Attempt Two: Bug Fix!

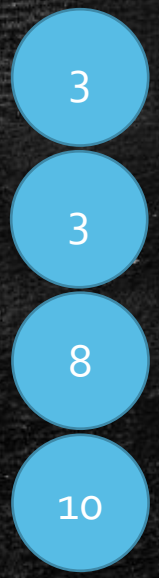


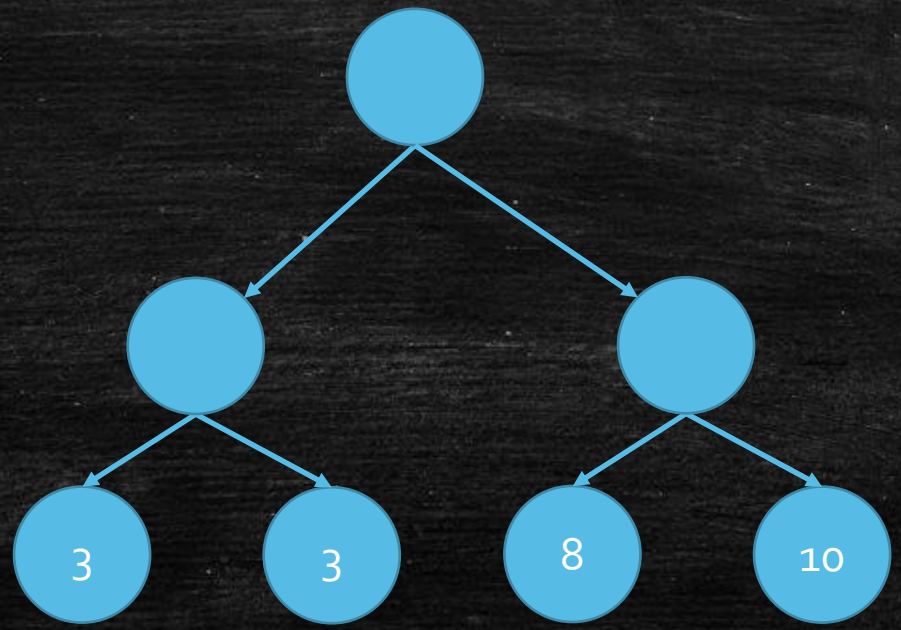
Attempt Two: Bug Fix!



A Bad Case

- Is the cost minimized?





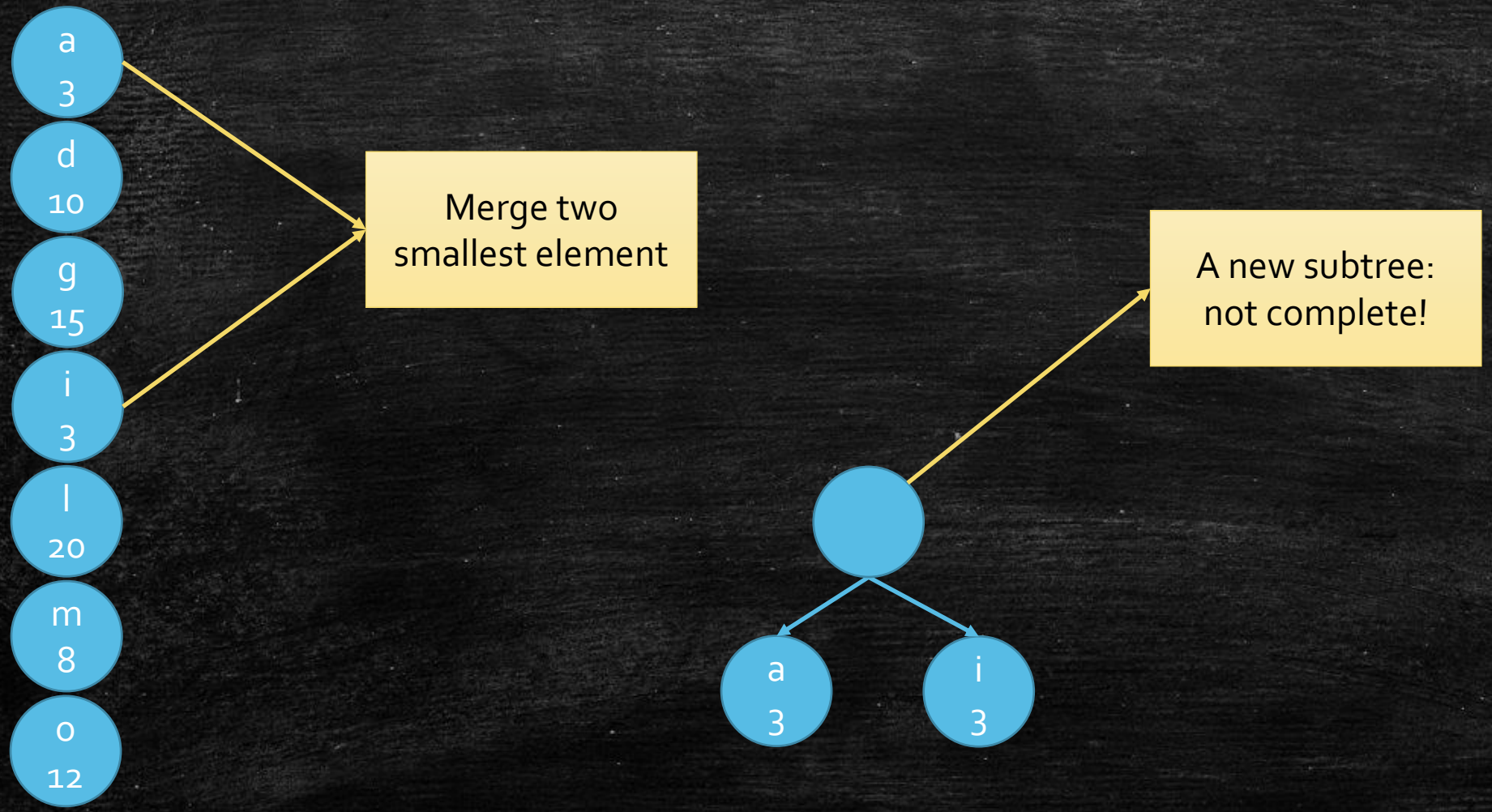
Thinking…

- When the first attempt is good?
- When the second attempt is good?
- Some observation:
  - The new vertex is small → the second attempt is good.
  - The new vertex is large → the first attempt is good.
- Another viewpoint
  - Merge two vertex = increasing the height of the two vertex (or subtree).
  - Go back and check!

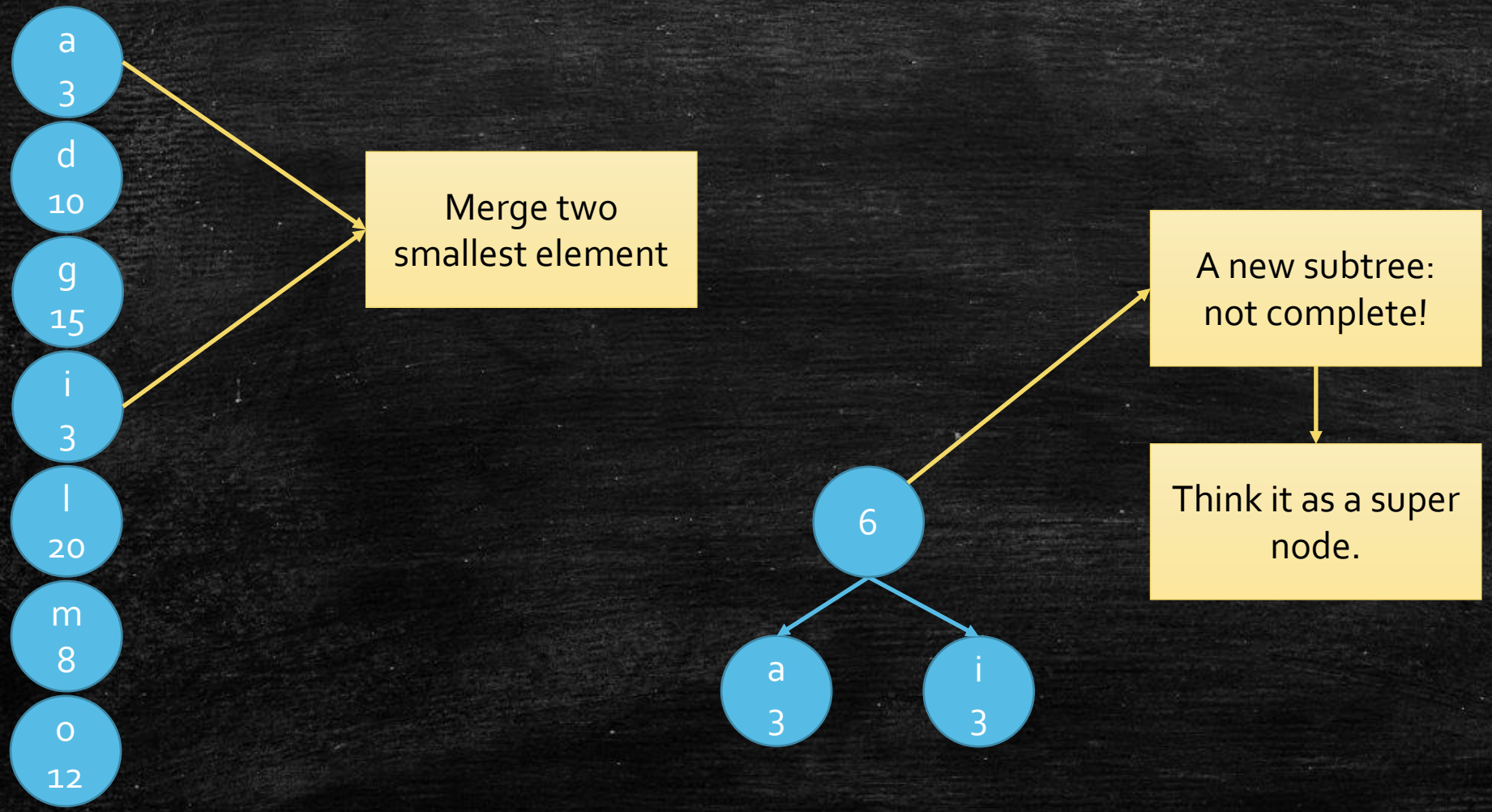
Move to A Better Attempt

Huffman Encoding

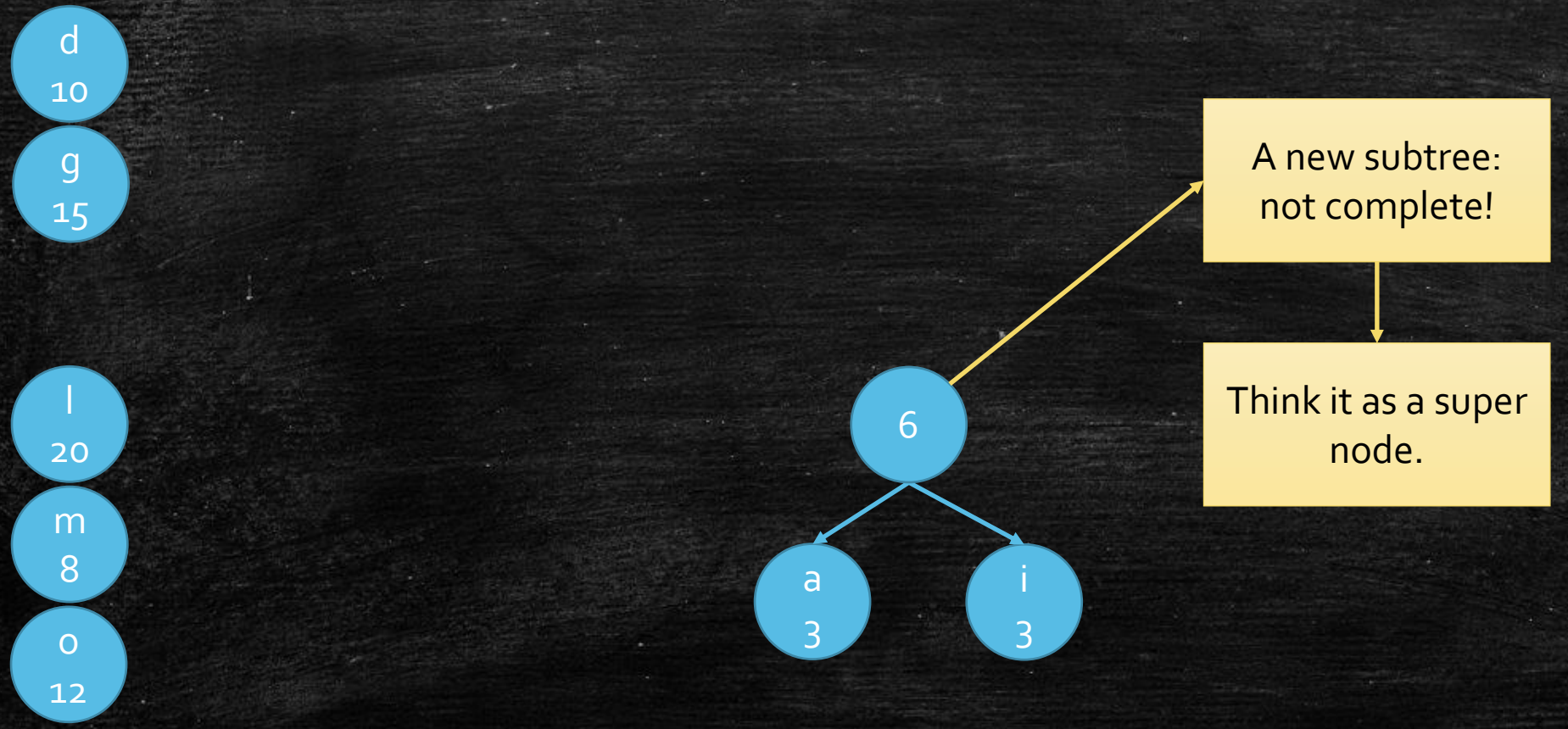
A bottom-up building



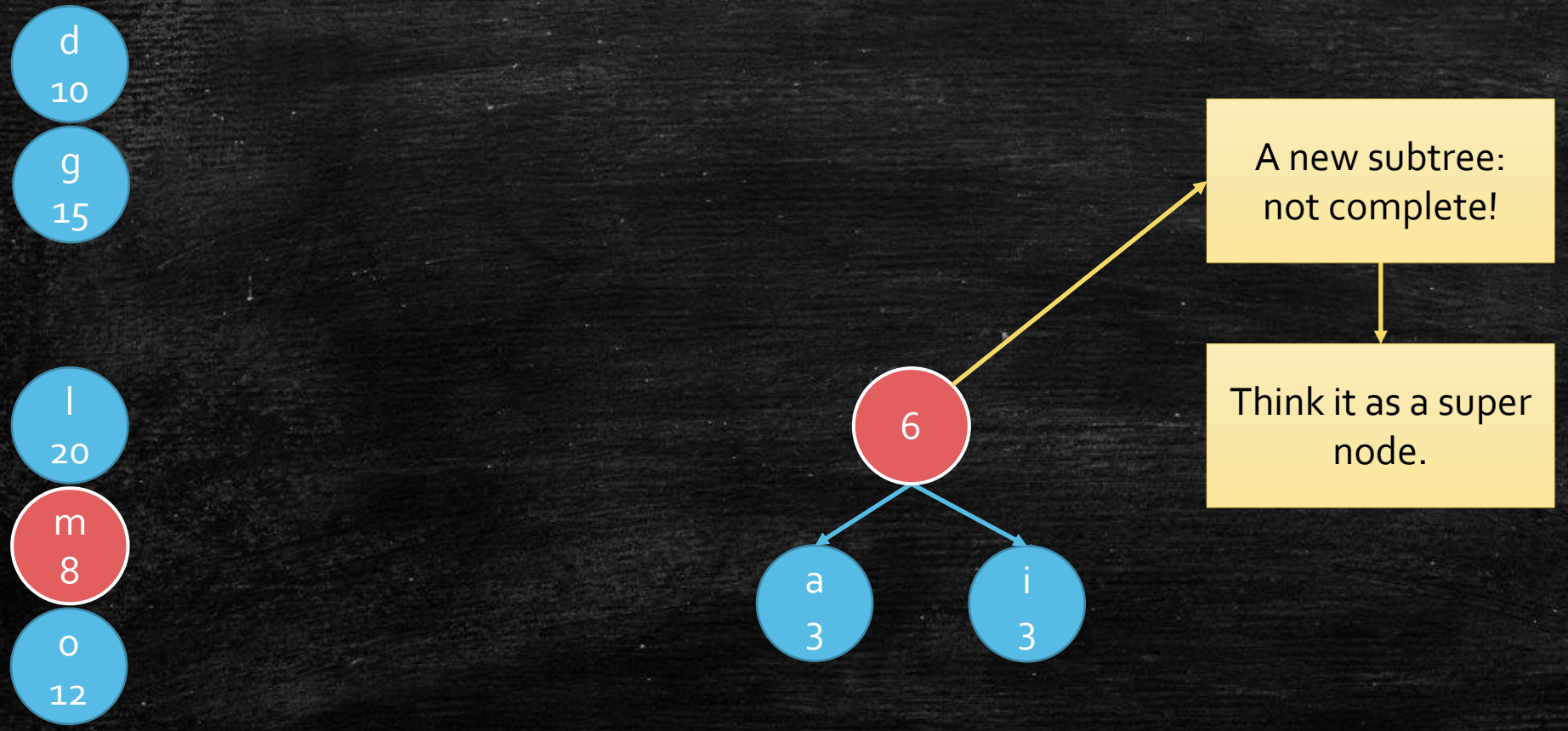
A bottom-up building



A bottom-up building



A bottom-up building



A bottom-up building

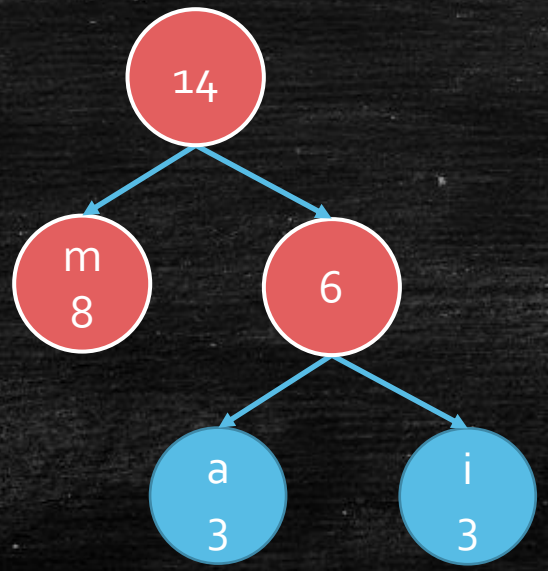
d
10

g
15

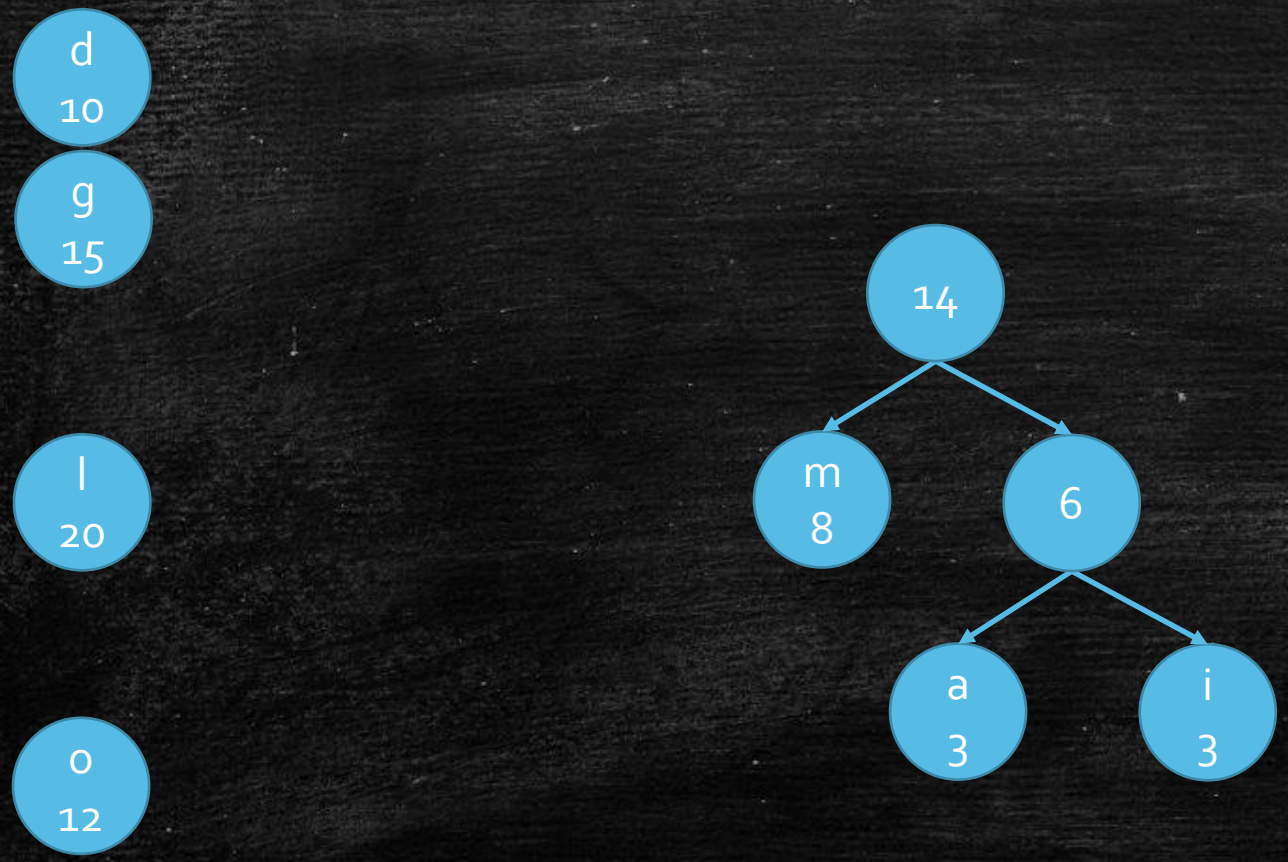
l
20

m
8

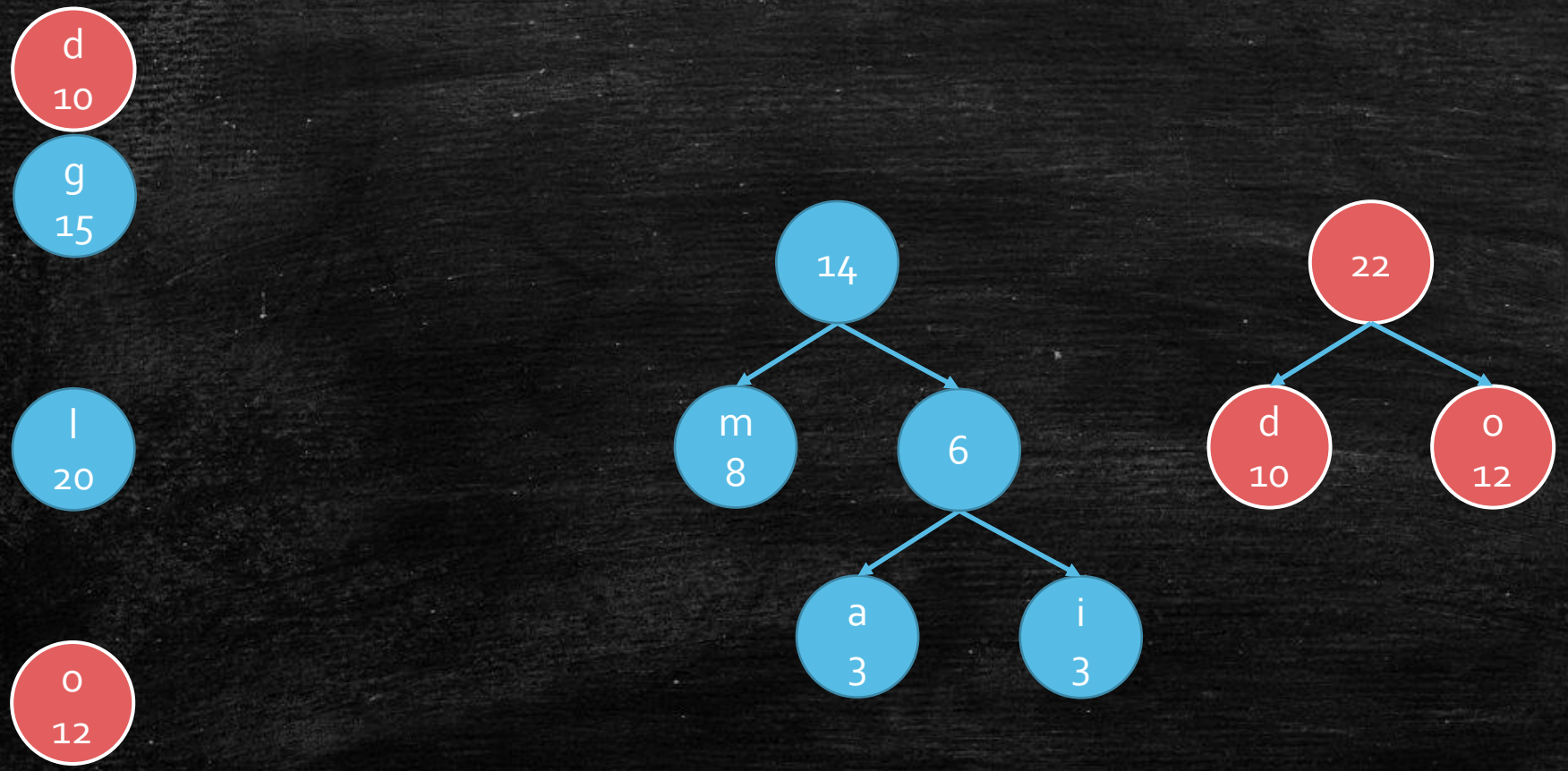
o
12



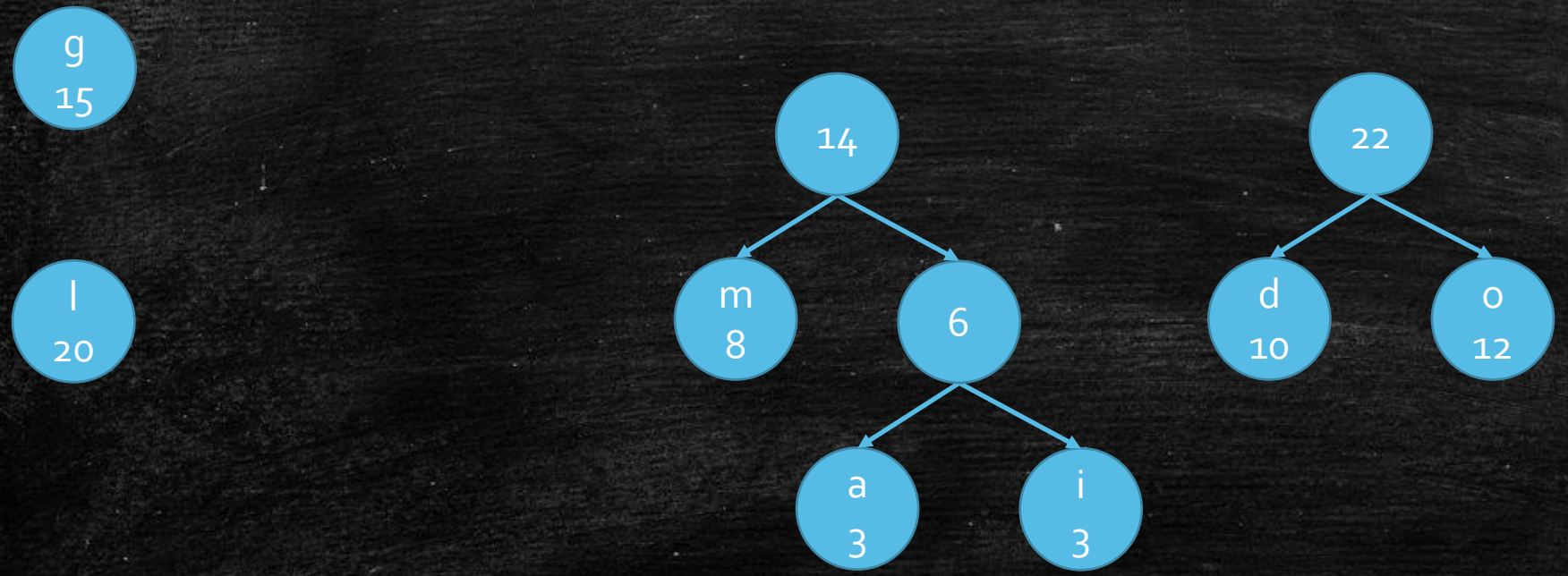
A bottom-up building



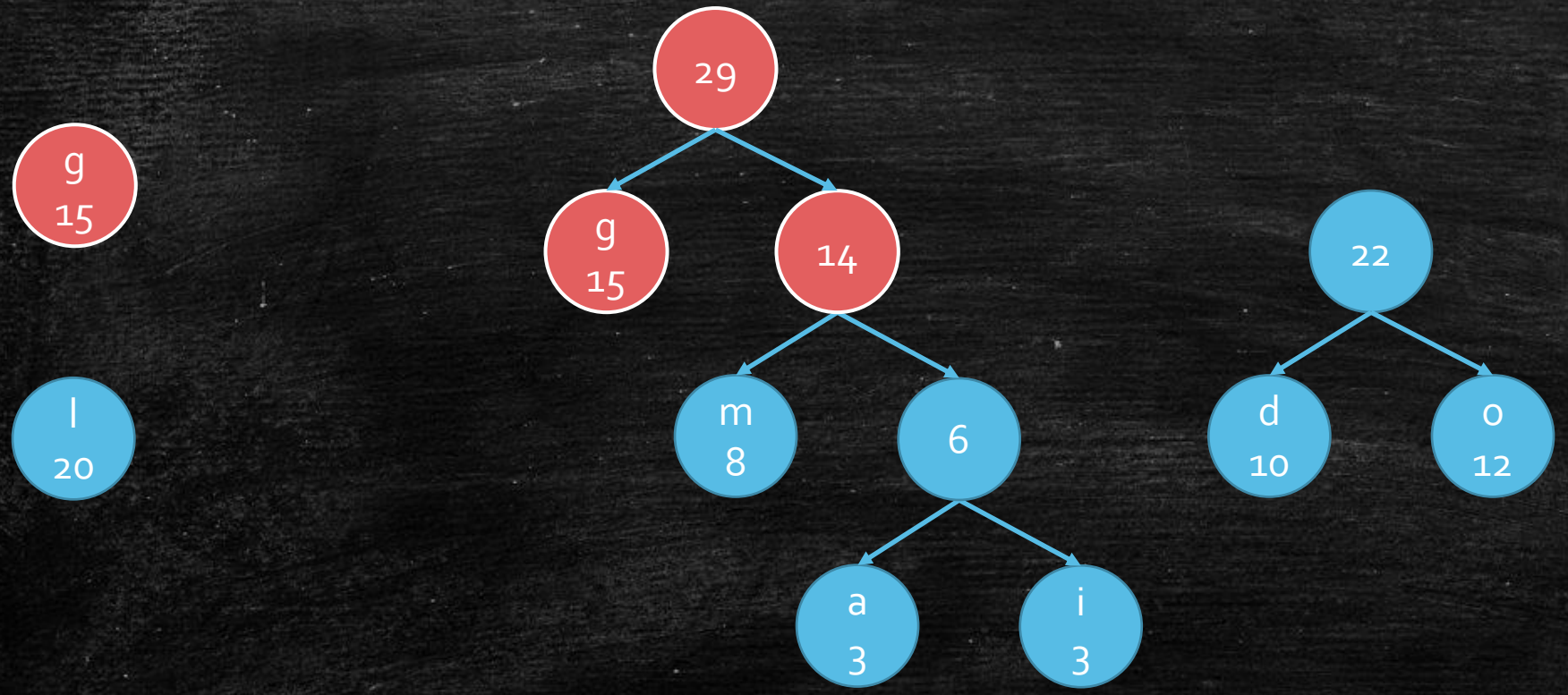
A bottom-up building



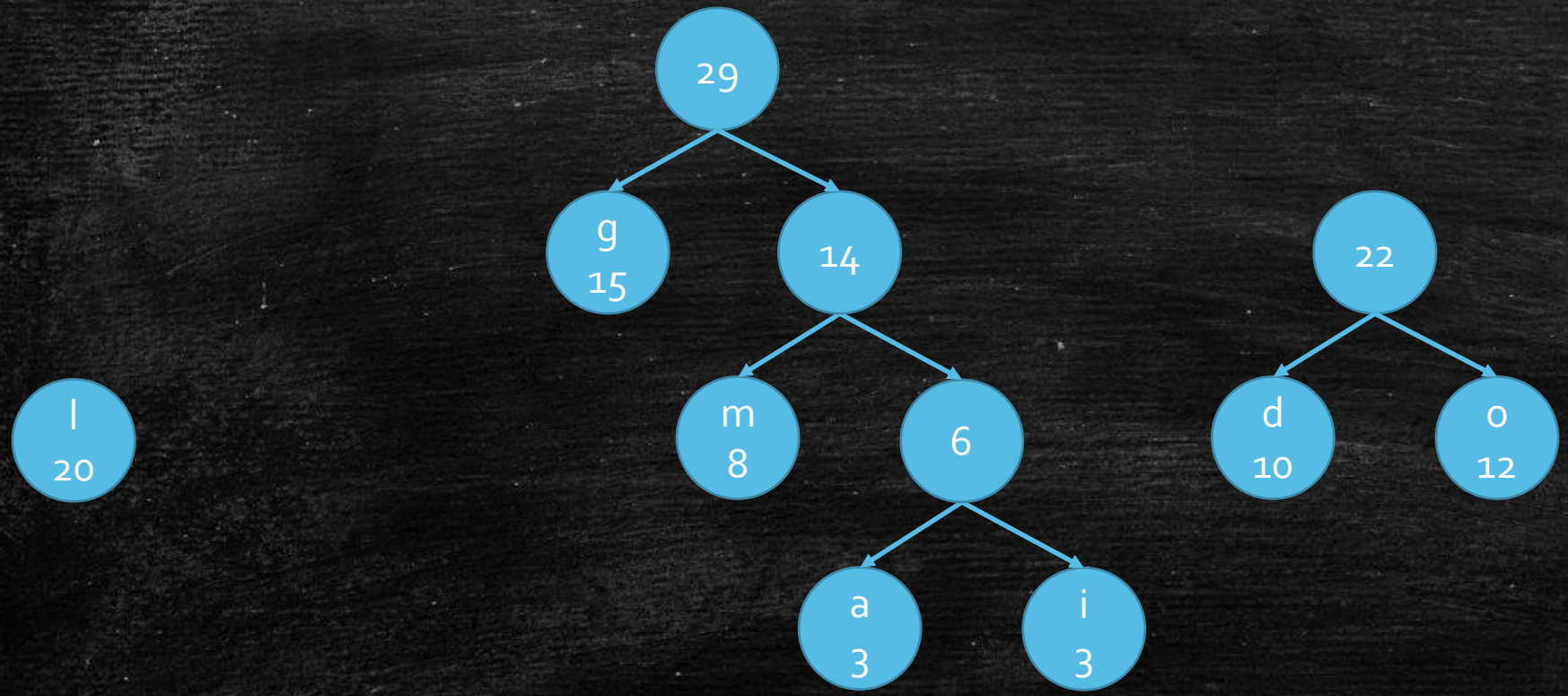
A bottom-up building



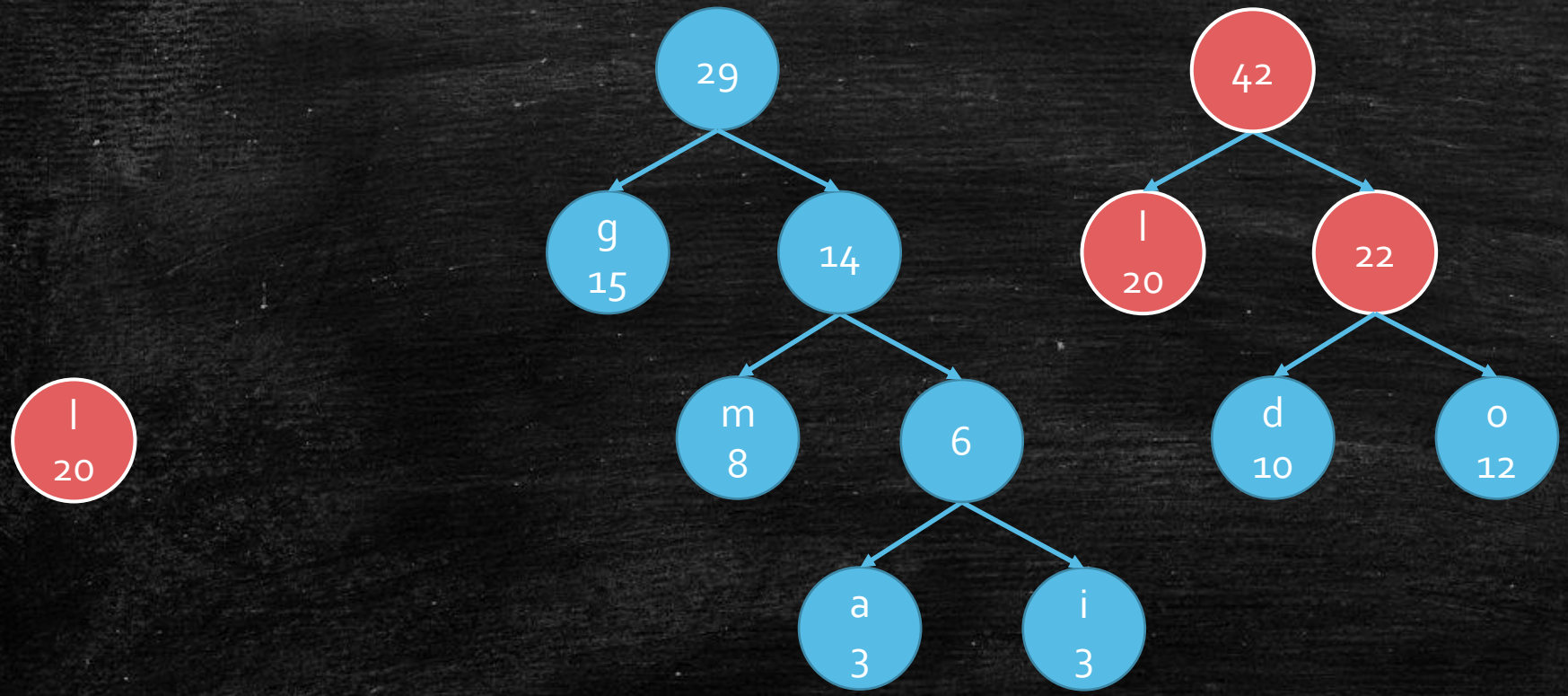
A bottom-up building



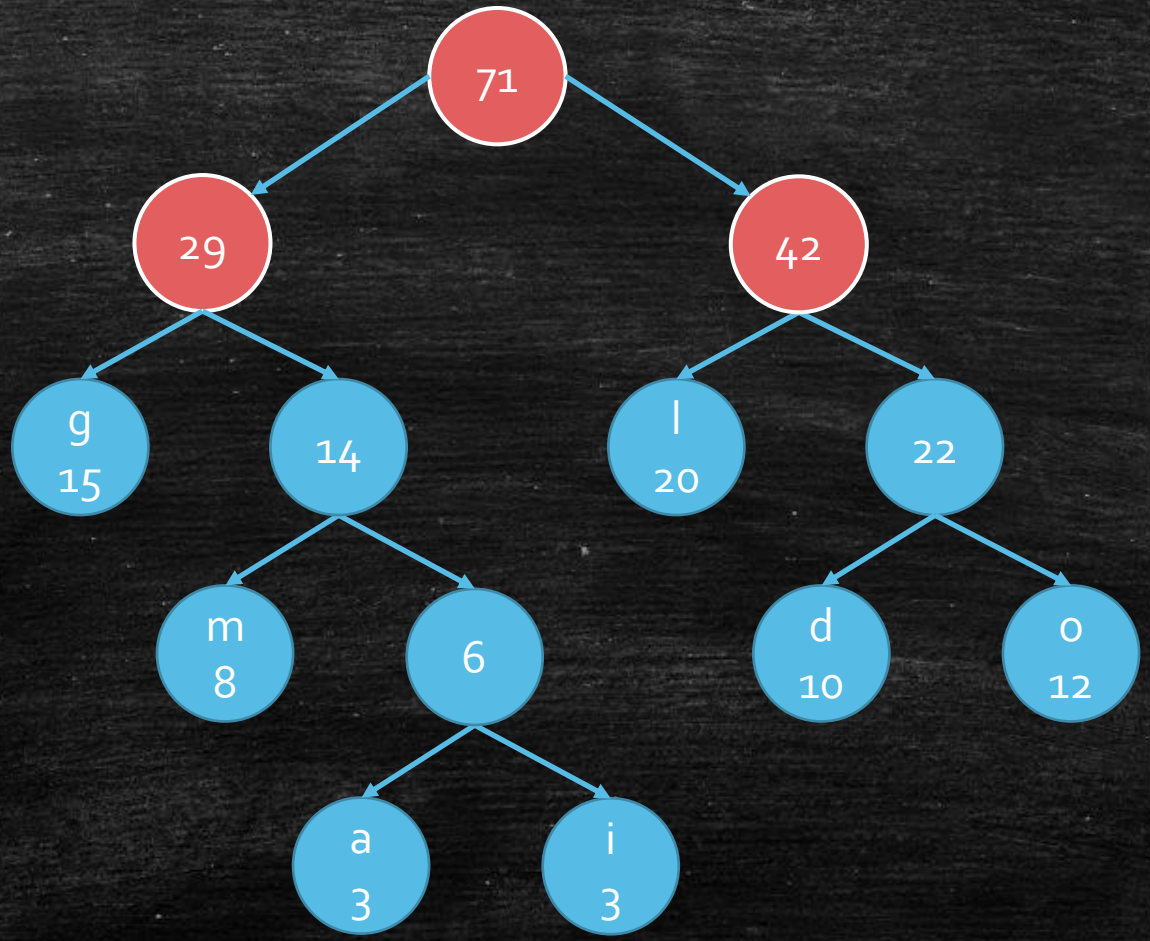
A bottom-up building



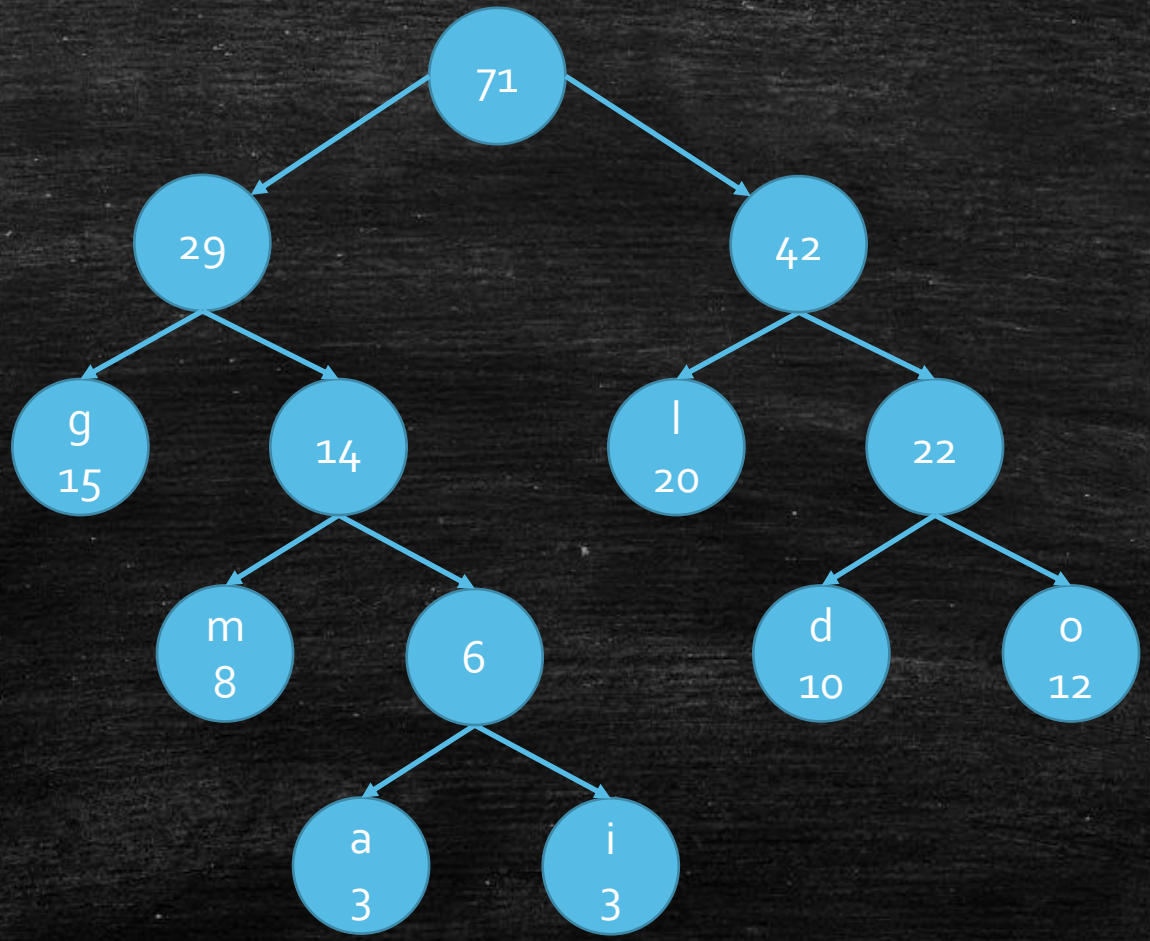
A bottom-up building



A bottom-up building

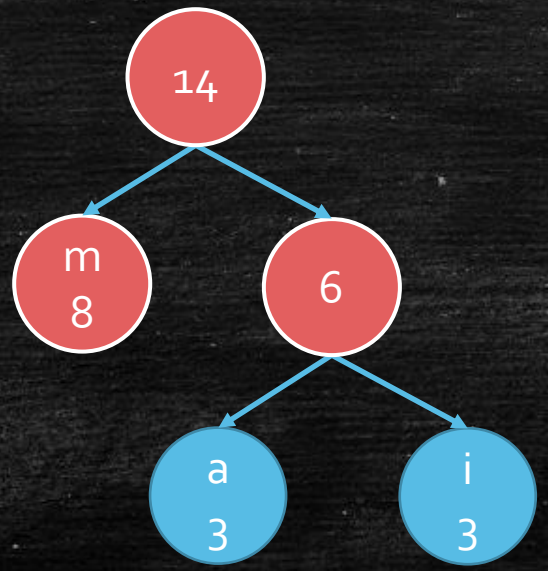


A bottom-up building



Is the cost minimized?

We create a new node with the **min cost**.



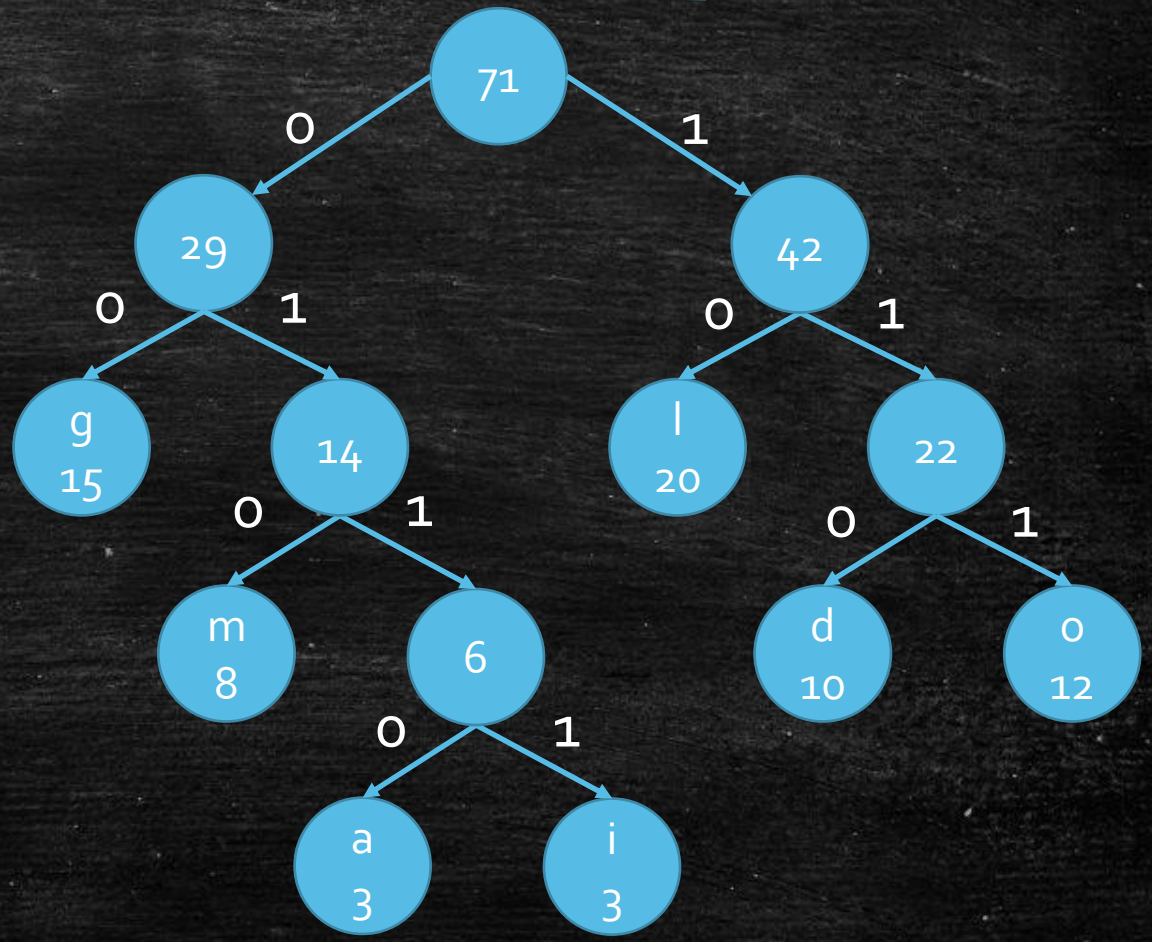
We pay the minimized **Augmentation Cost.**

Are the greedy approach optimal globally?

Move to Analysis

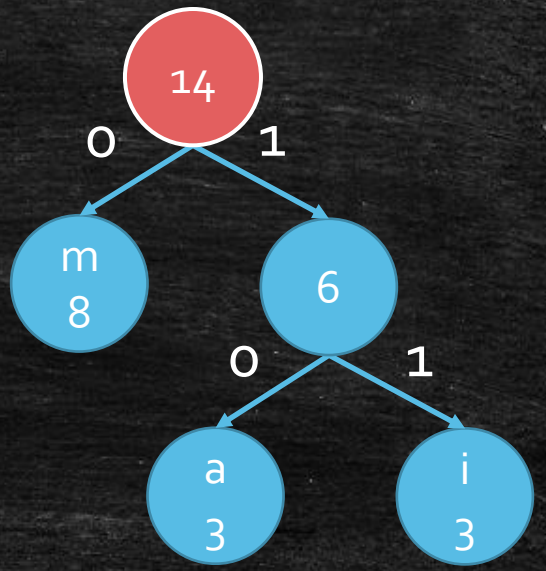
Reformulate The Cost

- Let $L =$ set of leaves
- $Cost = \sum_{v \in L} lev(v) \cdot f(v)$



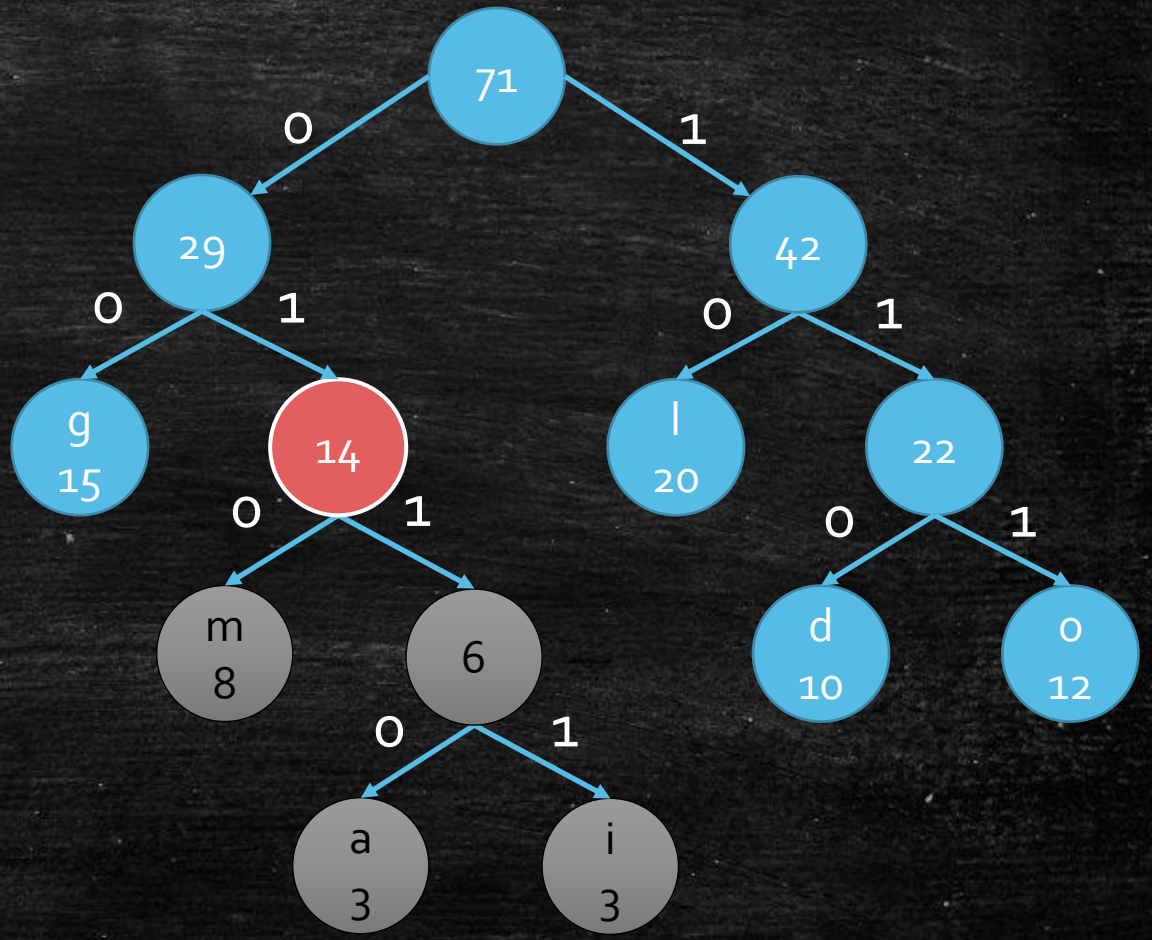
Augmentation View

- What do we mean when we merge 6 and 8?
- 6 + 8 nodes increase height of 1.



Another View

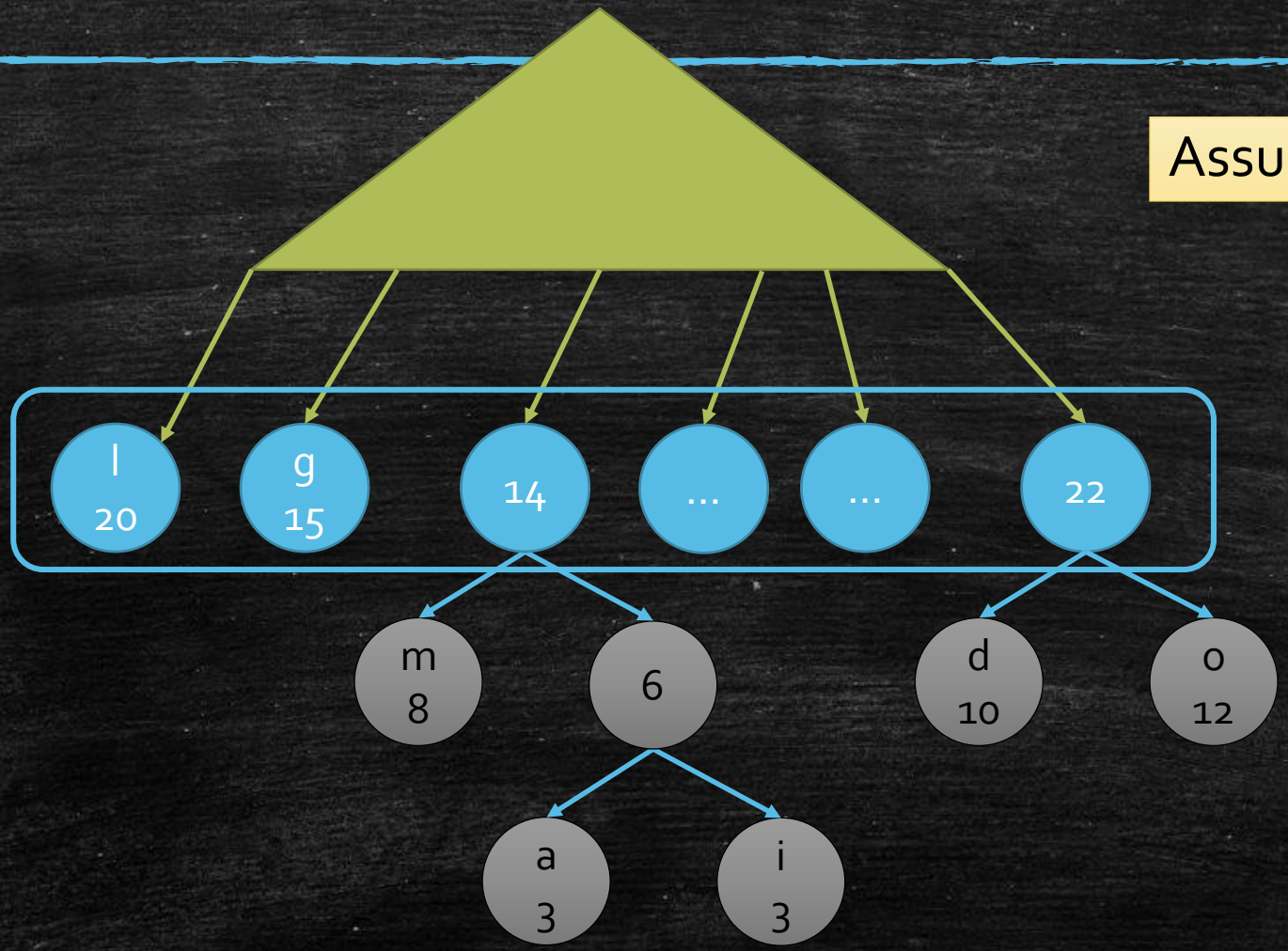
- Later, 14's level increase again.
- We will further pay a cost $lev(14) \cdot 14$
- The contribution of the cost by the subtree(14):
  - $3 \cdot 2 + 3 \cdot 2 + 8$
  - $14 \cdot lev(14)$



Proof of The Correctness

- The Big Idea
  - The local greedy choice do not ruin out OPT.
- Assume we are still in a partial-OPT,
- After Merging two smallest elements, prove we are still in a partial-OPT.

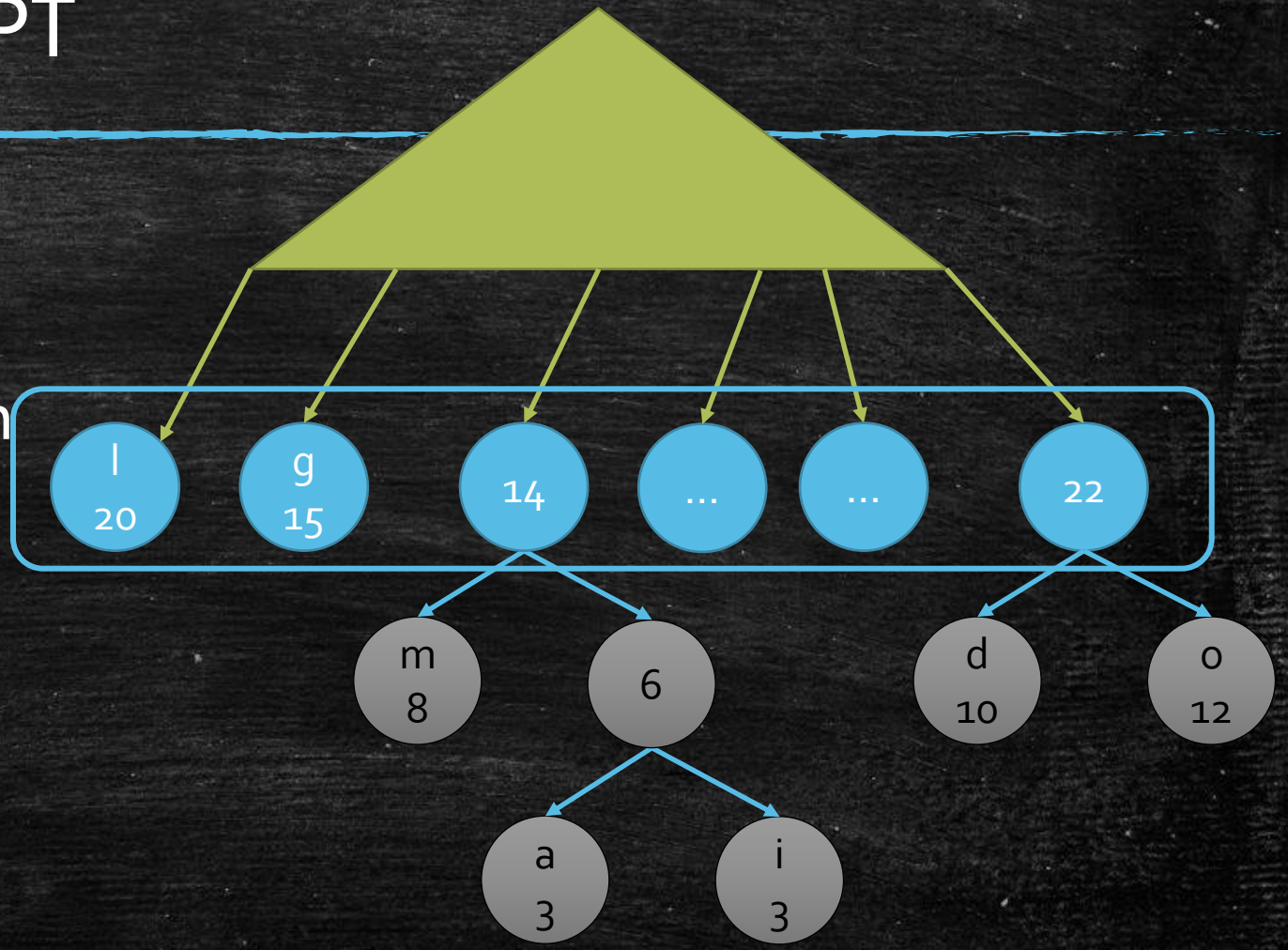
Induction



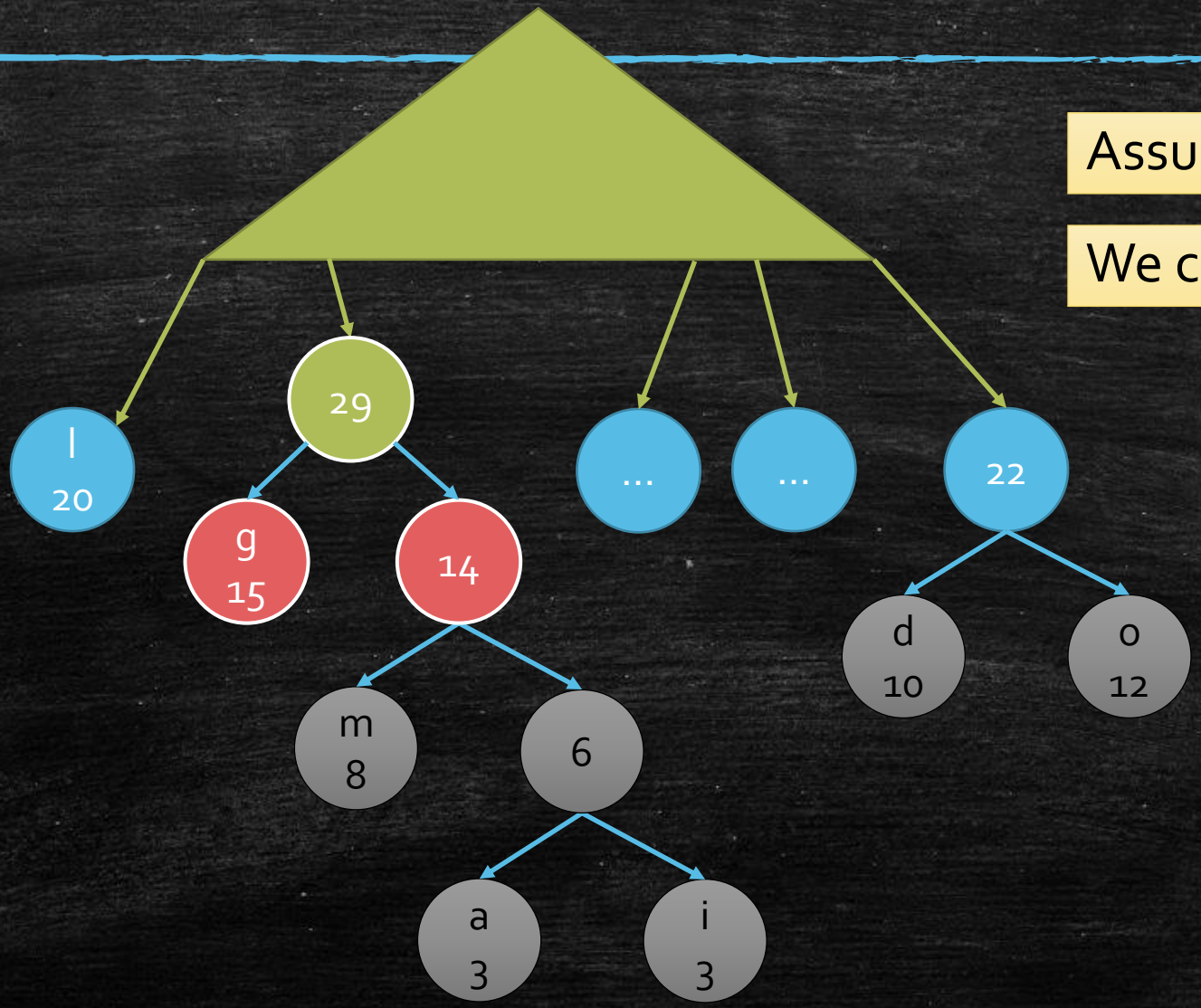
Assume we are in T^*.

We are in Partial-OPT

- Grey nodes pay the same cost as OPT.
- Further cost depends on the level of blue vertices.
- $Cost = Cost(grey) + \sum_{v\ is\ blue} lev(v) \cdot f(v)$.



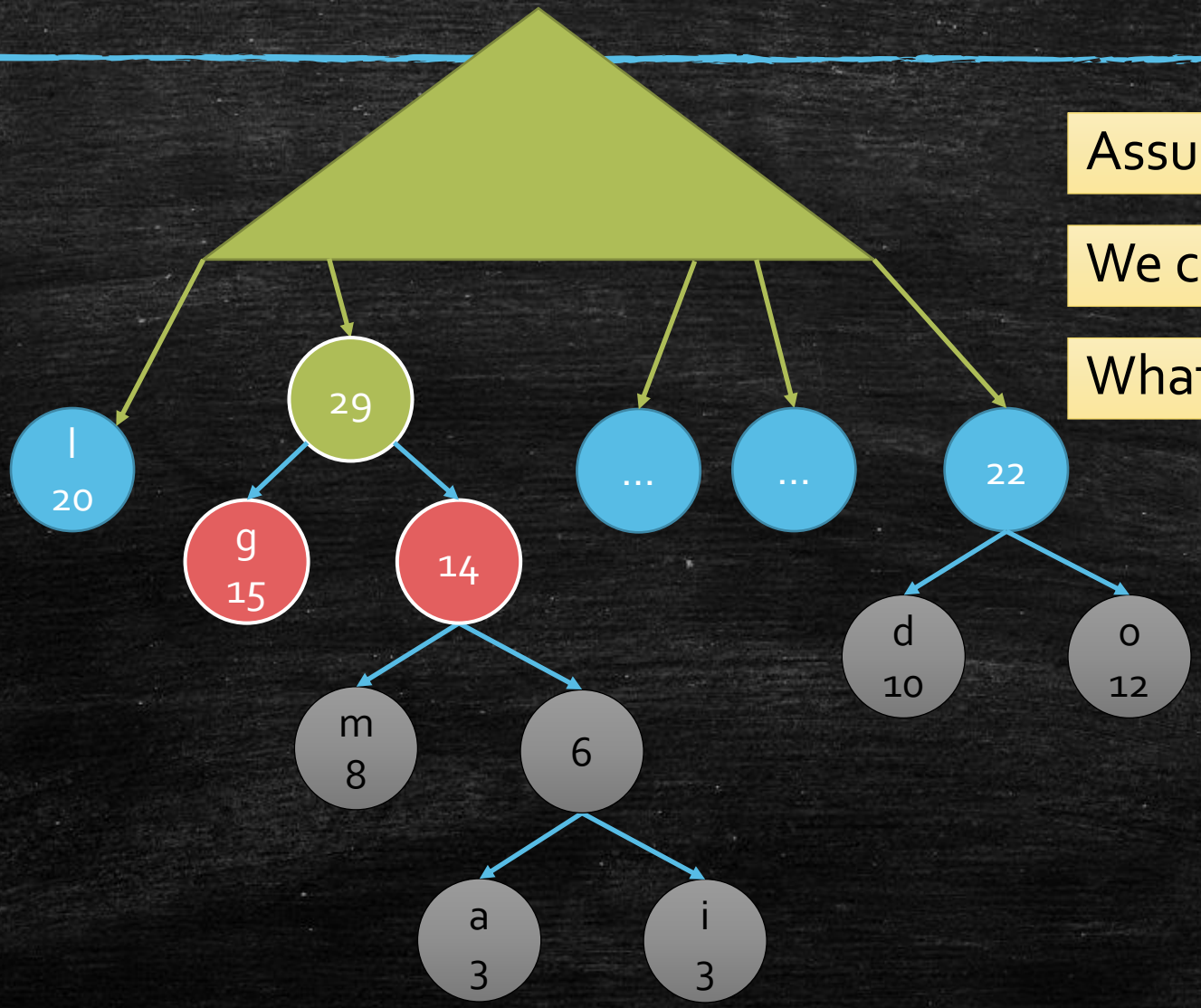
Induction



Assume we are in T^*.

We choose 14 and 15.

Induction

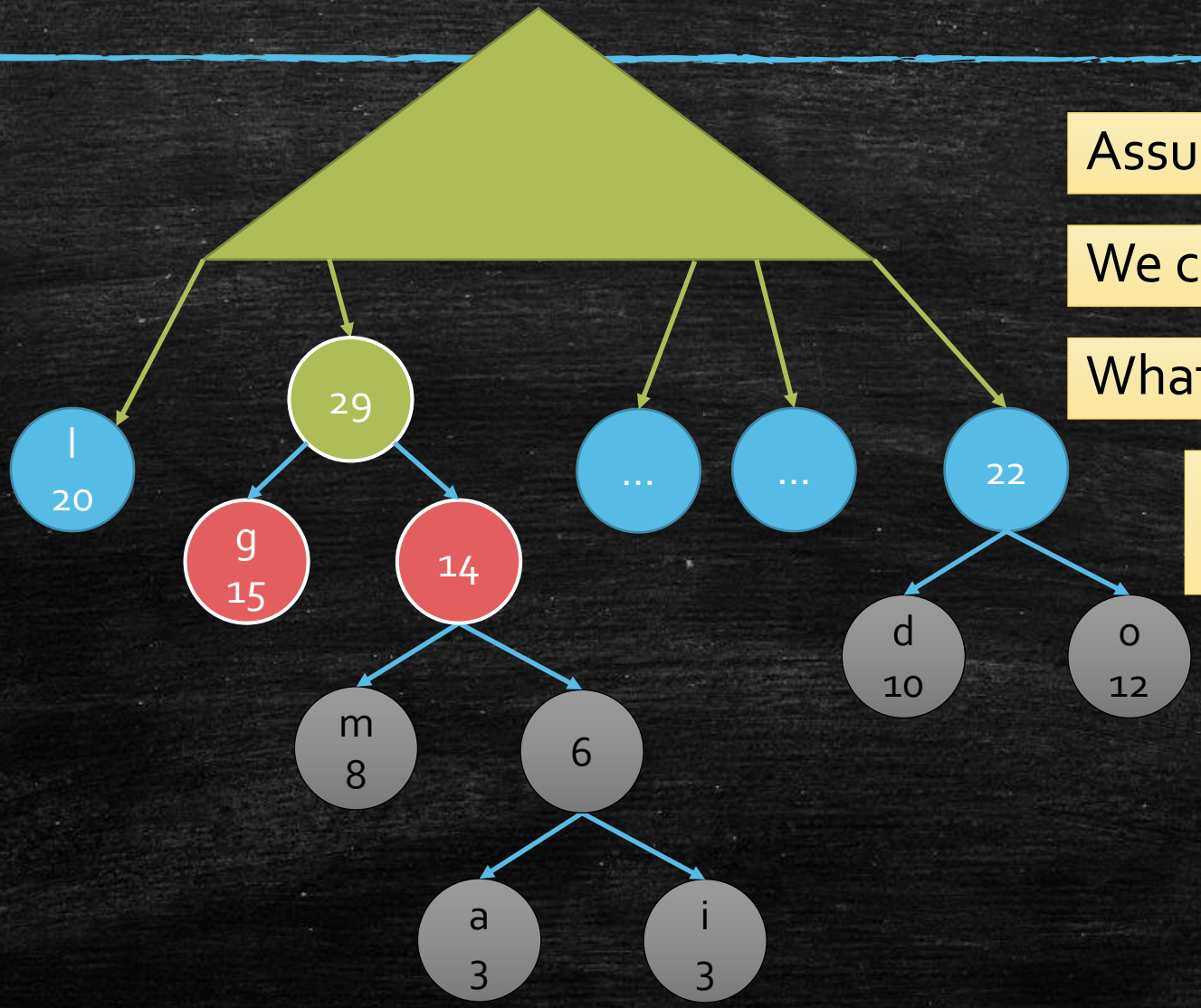


Assume we are in T^*.

We choose 14 and 15.

What if we are in trouble?

Induction



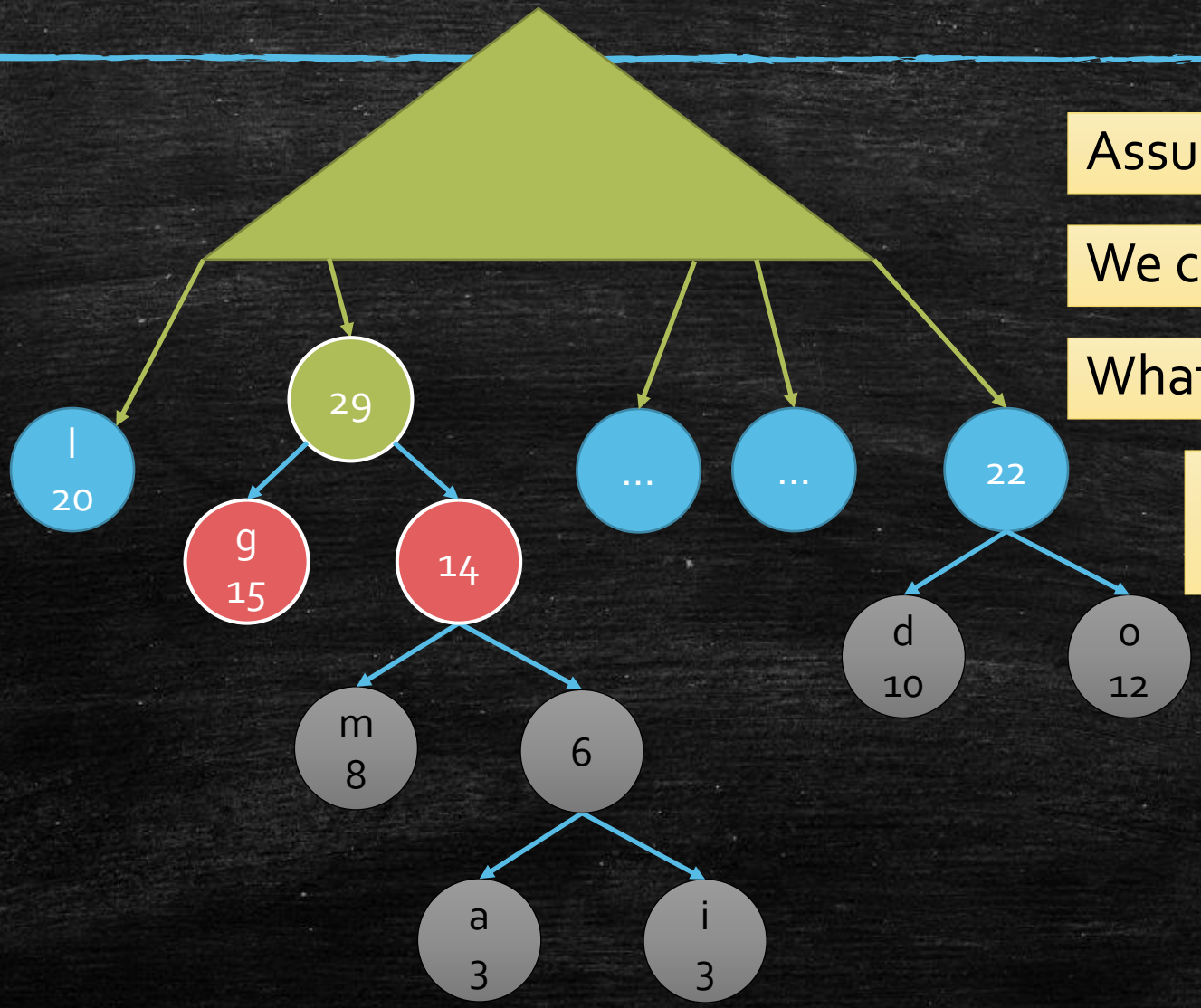
Assume we are in T^*.

We choose 14 and 15.

What if we are in trouble?

14 and 15 are not sibling in T^*

Induction



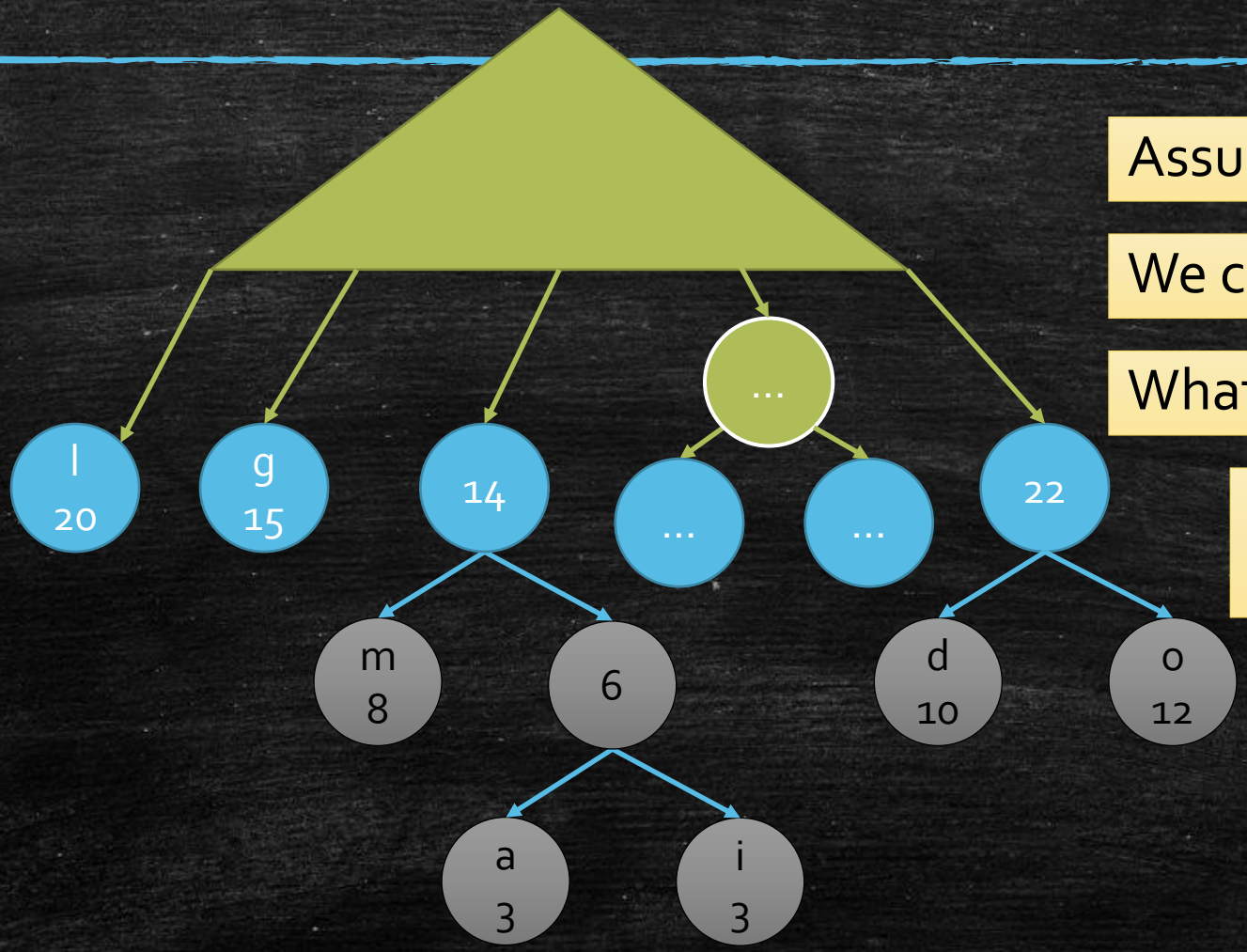
Assume we are in T^*.

We choose 14 and 15.

What if we are in trouble?

14 and 15 are not sibling in T^*

Induction



Assume we are in T^*.

We choose 14 and 15.

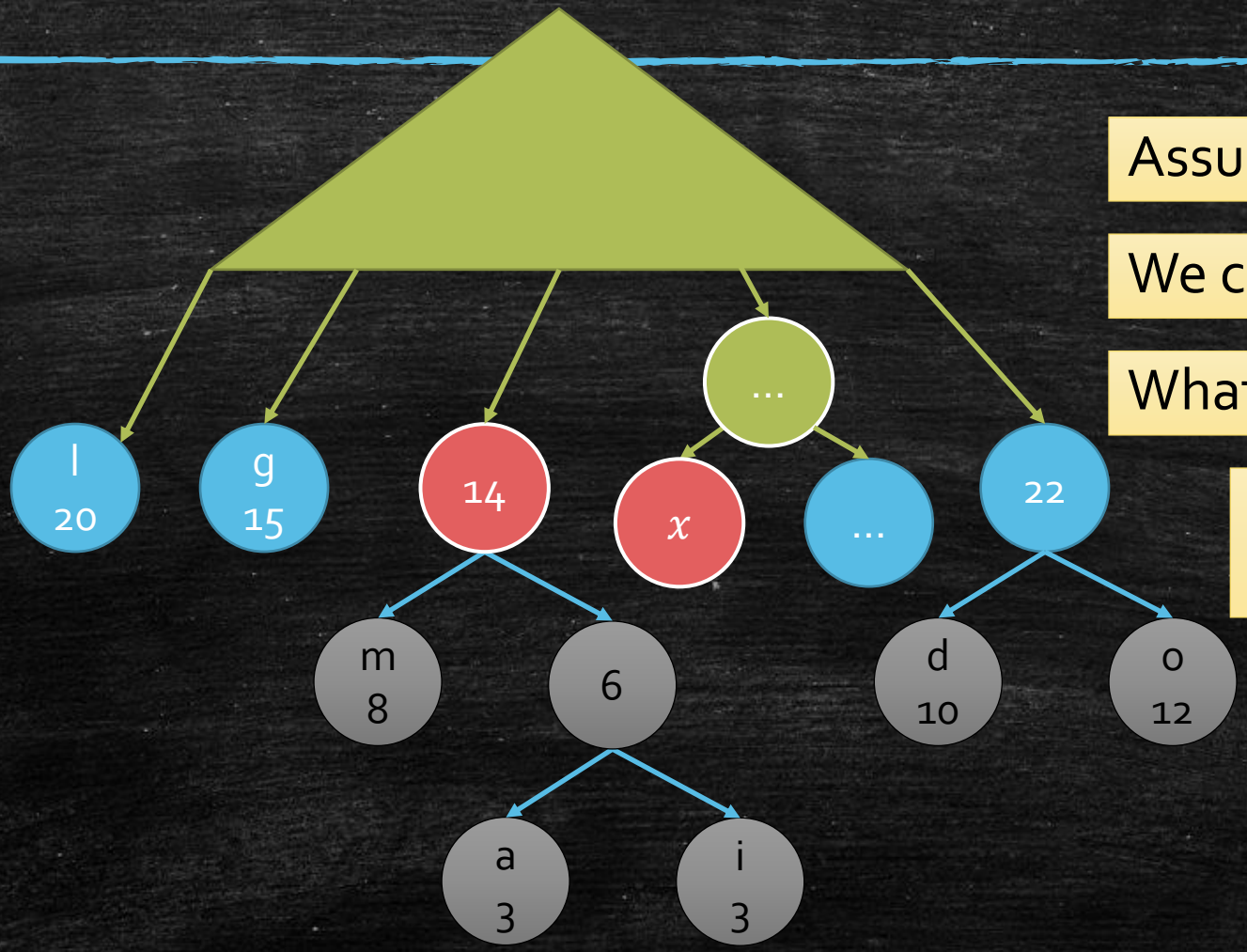
What if we are in trouble?

14 and 15 are not sibling in T^*

Check two lowest (largest lev) Siblings.

They are not 14 and 15.

Induction



Assume we are in T^*.

We choose 14 and 15.

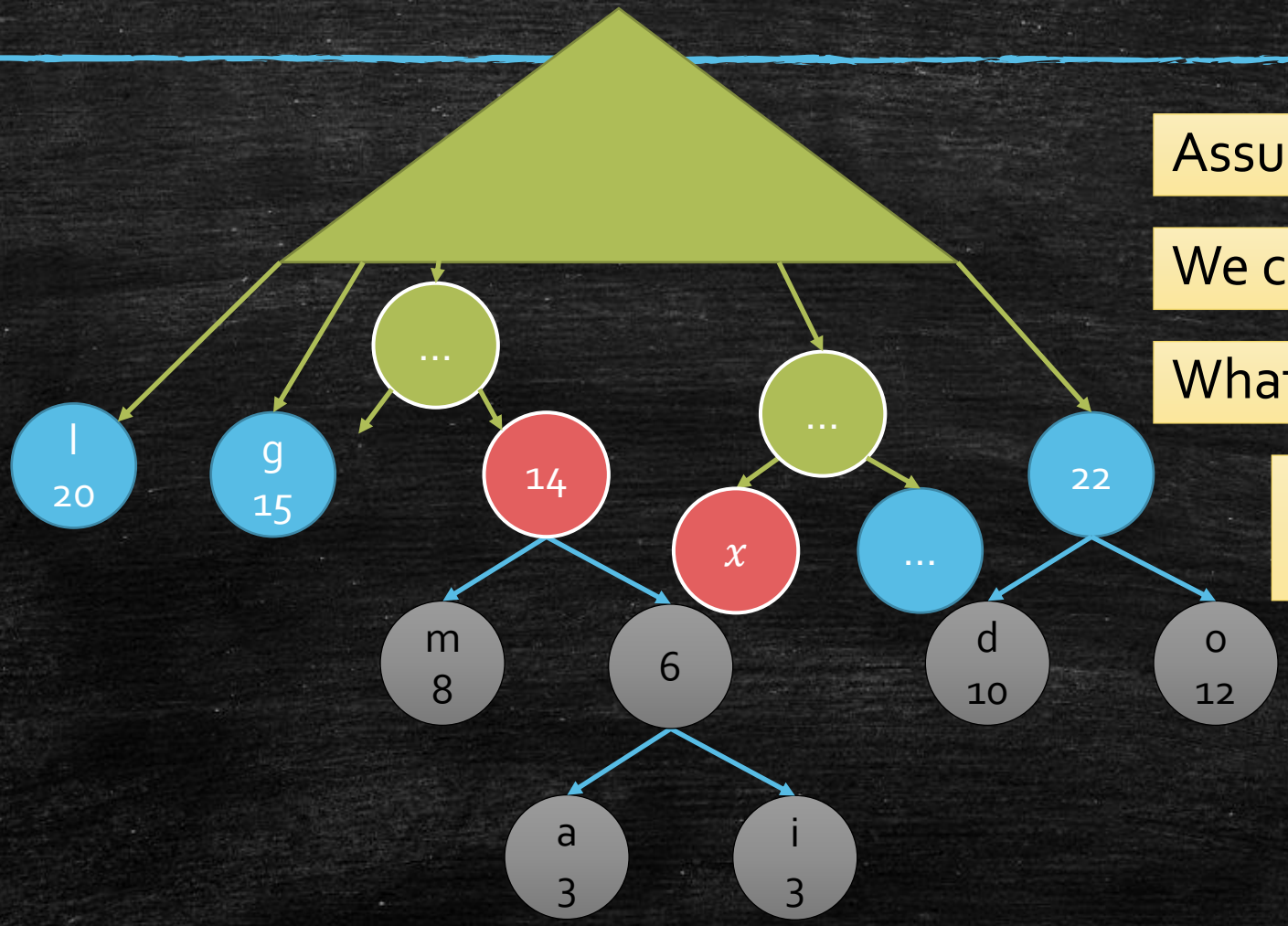
What if we are in trouble?

14 and 15 are not sibling in T^*

Check two lowest (largest lev) Siblings.

One of them is greater than 15.

Induction



Assume we are in T^*.

We choose 14 and 15.

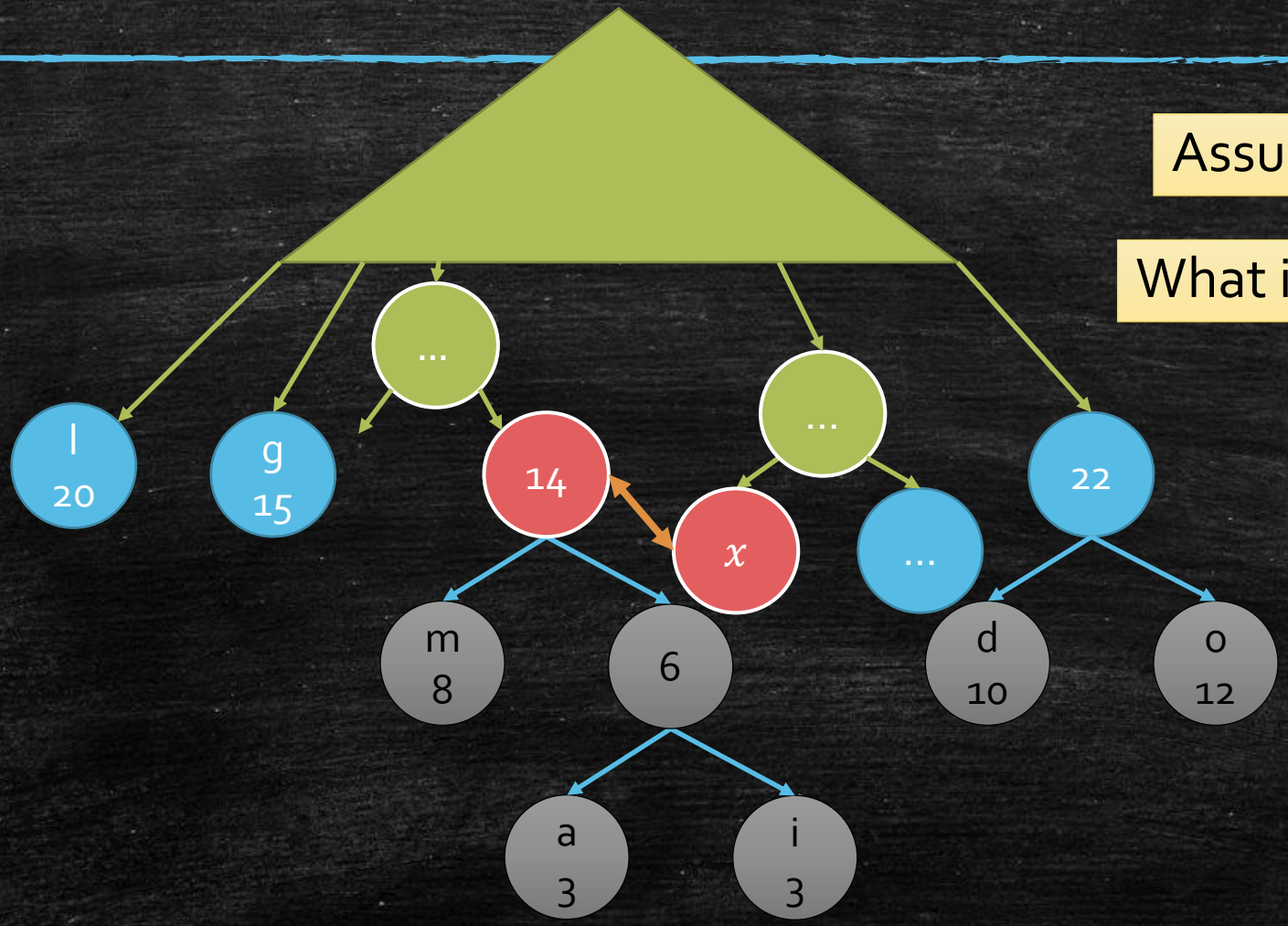
What if we are in trouble?

14 and 15 are not sibling in T^*

Check two lowest (largest lev) Siblings.

One of them is greater than 15.

Induction



Assume we are in T^*.

What if we switch 14 and x?

What happen?

- If we switch x and 14.
- Let L be the set of blue nodes.
- $Cost = Cost(grey) + \sum_{v \in L} lev(v) \cdot f(v)$
- Only two nods in L change lev.
- $lev[x]$ decrease h.
- $lev[14]$ increase h.
- $\Delta Cost = h \cdot (14 - x) \leq 0$!
- Switching x and 14 make the tree better!

Conclusion Formally

- **Lemma:** If we merge two vertices, then there is an OPT where the two vertices are siblings.
- It is enough to show we are still in an OPT after a round.
- **Conclusion:** The greedy strategy builds a prefix-free code with minimized cost.

Huffman Encoding: Time Complexity

- Sort the characters by their appearance.
  - $O(n \log n)$
- Repeat n rounds
  - Find two minimized appearance elements.
  - Delete two minimized element.
  - Insert a super node into the list.

Time Complexity

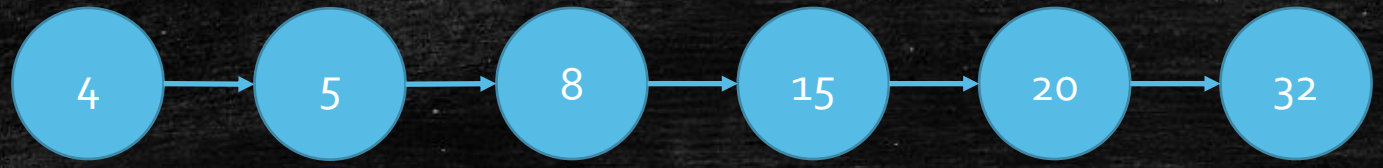
- Sort the characters by their appearance.
  - $O(n \log n)$
- Repeat n rounds
  - Find two minimized appearance elements.
  - Delete two minimized element.
  - Insert a super node into the list.
- Use a Heap?
  - Each round: $\boldsymbol{O(1) + O(\log n) + O(\log n).}$
- Totally: $O(n \log n)$ even if the characters are sorted.

Can we Improve?

- We have two bottlenecks
  - Sorting $O(\log n)$
  - Tree building $O(\log n)$

Can we Improve?

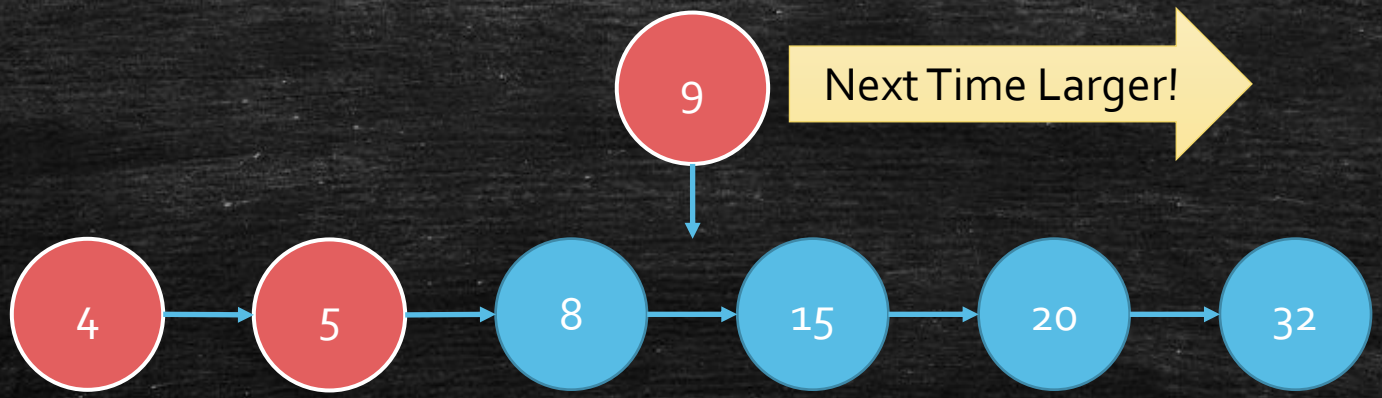
- We have two bottlenecks
  - ~~Sorting~~
  - Tree building
- Given a sorted list.
- Repeat n rounds
  - Find two minimized appearance elements.
  - Delete two minimized element.
  - Insert a super node into the list.



Can we Improve?

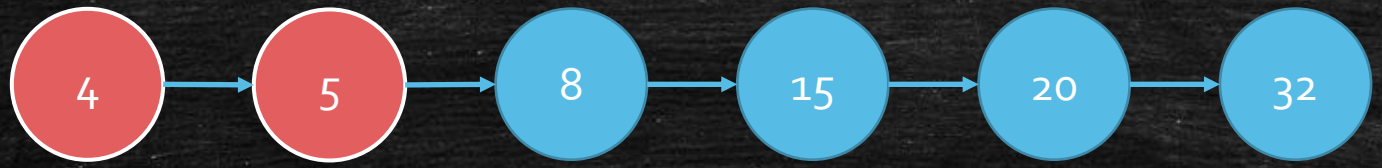
- Given a sorted list.
- Repeat n rounds
  - Find two minimized appearance elements.
  - Delete two minimized element.
  - Insert a super node into the list.
- Observation:
  - Inserted super nodes become **larger** and **larger**.

What if using only a sorted linked list?

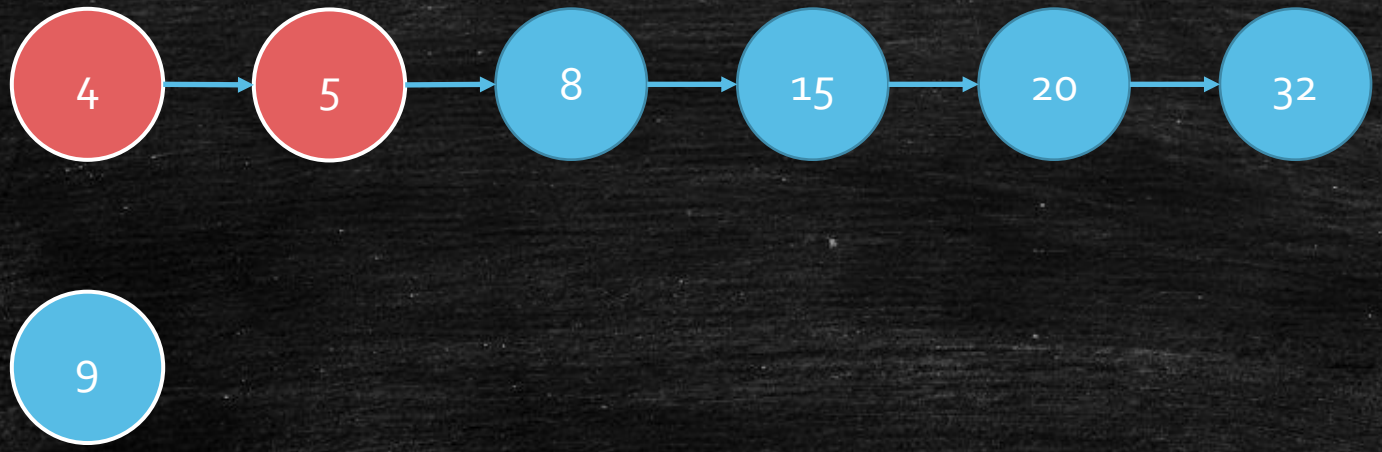


Use two priority queue to implement.

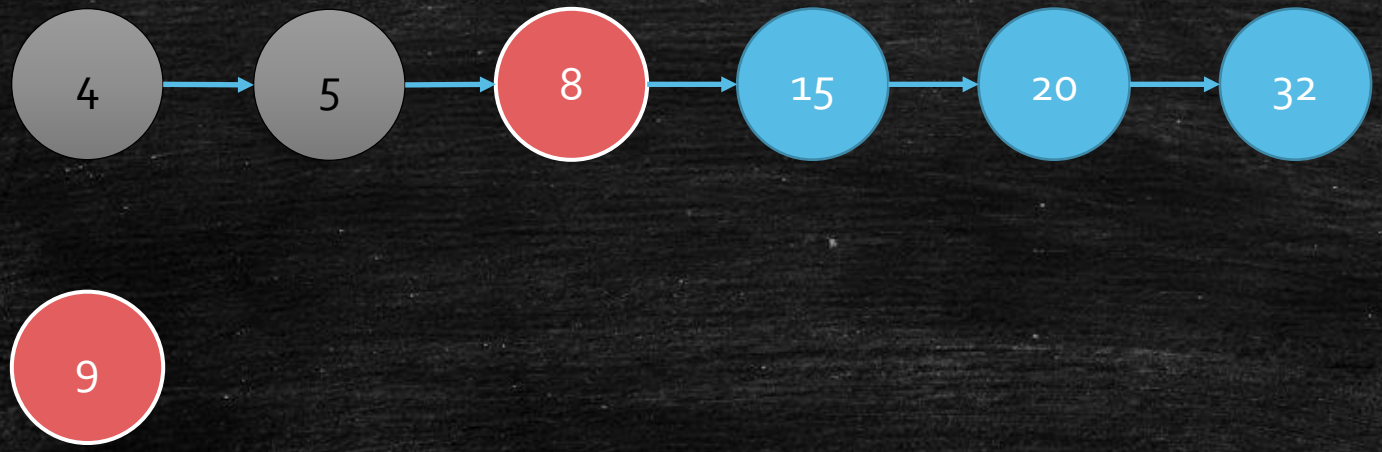
- Use two priority queue to implement.



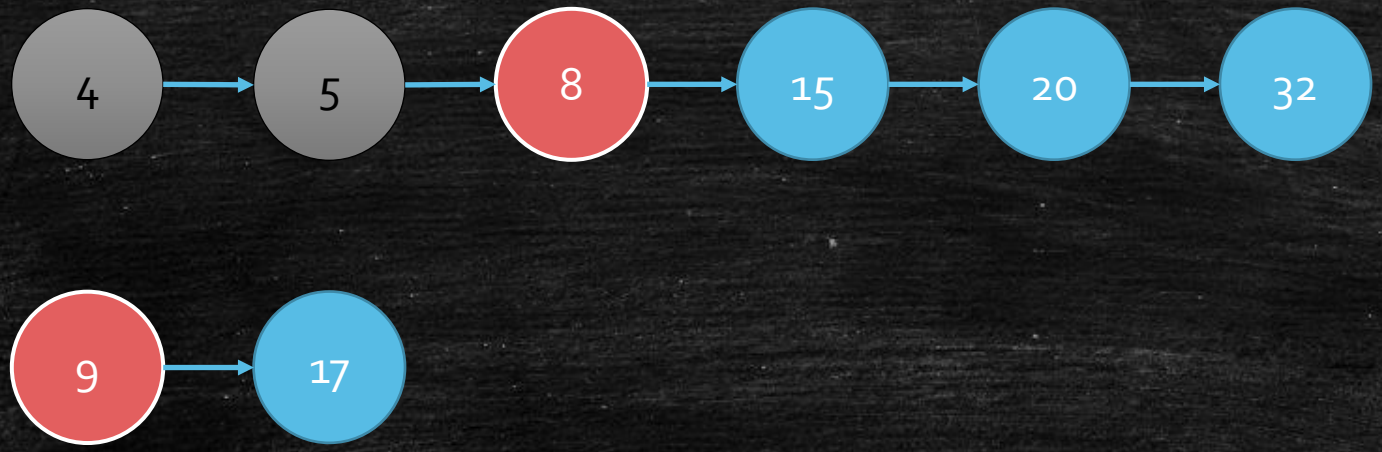
- Use two priority queue to implement.



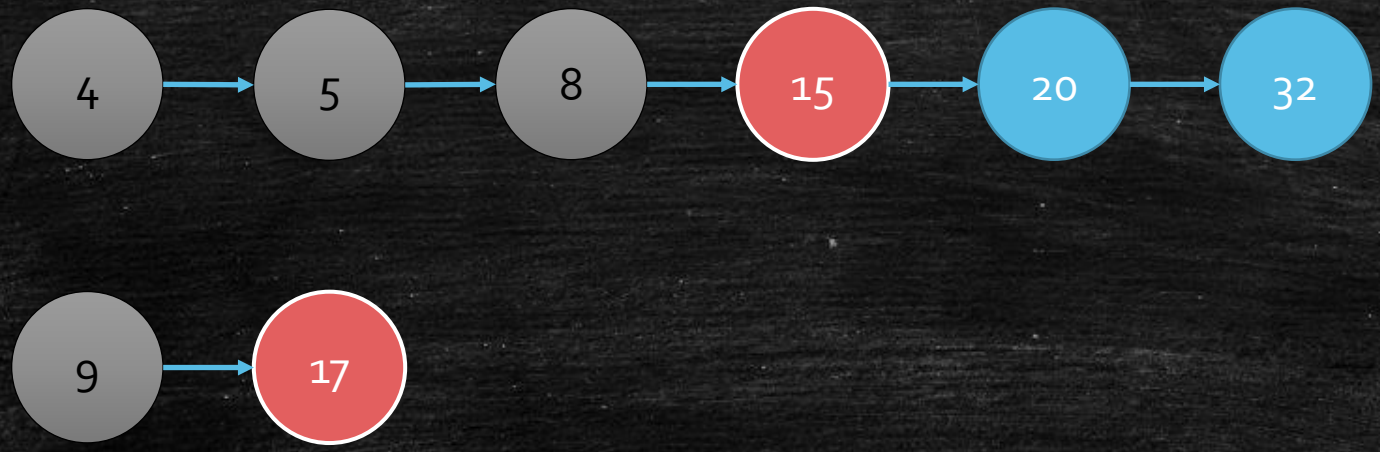
- Use two priority queue to implement.



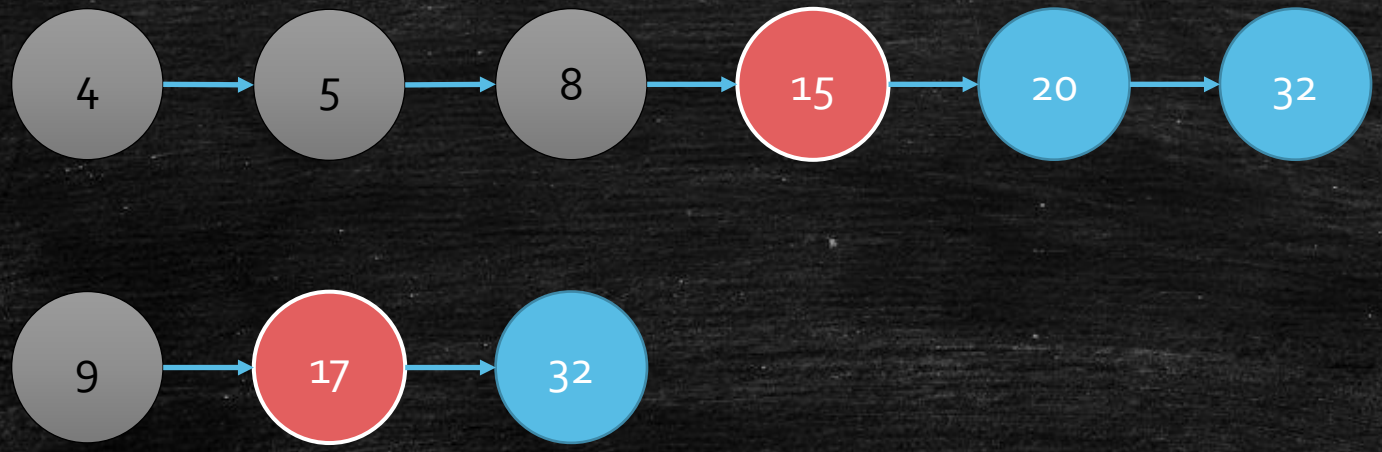
- Use two priority queue to implement.



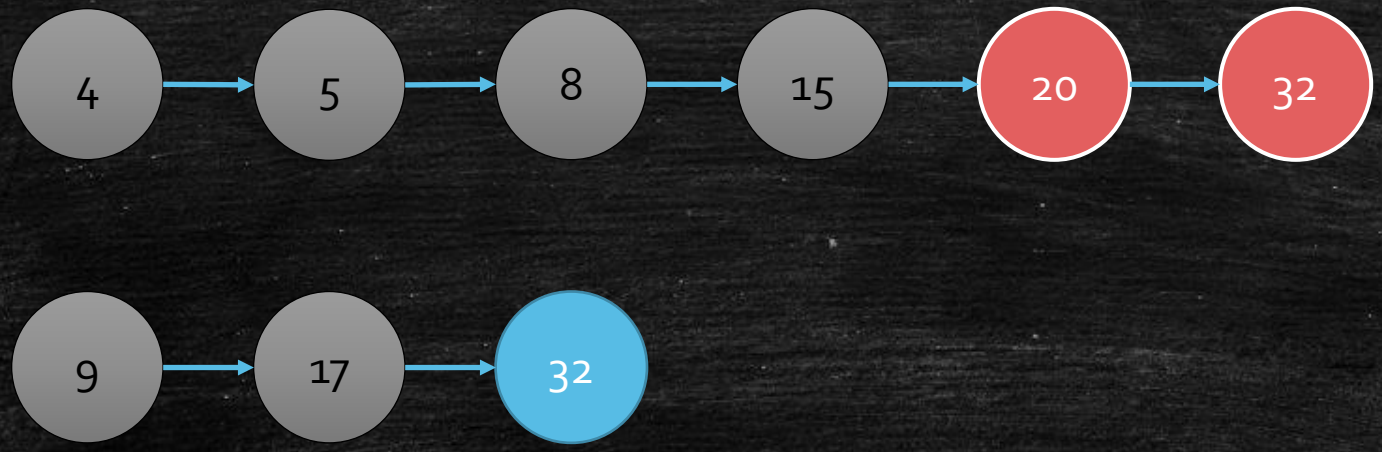
- Use two priority queue to implement.



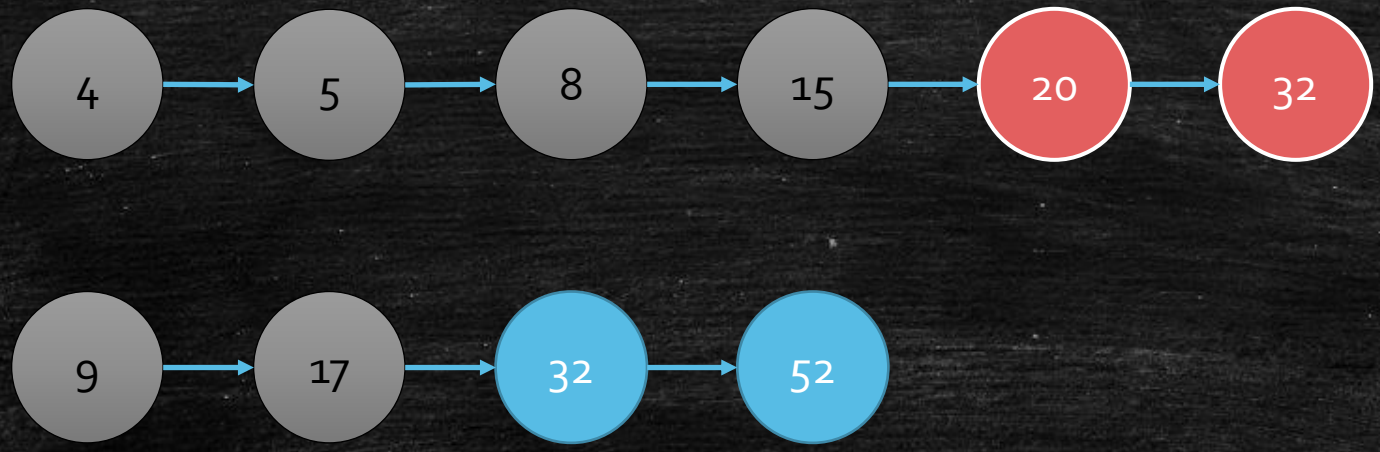
- Use two priority queue to implement.



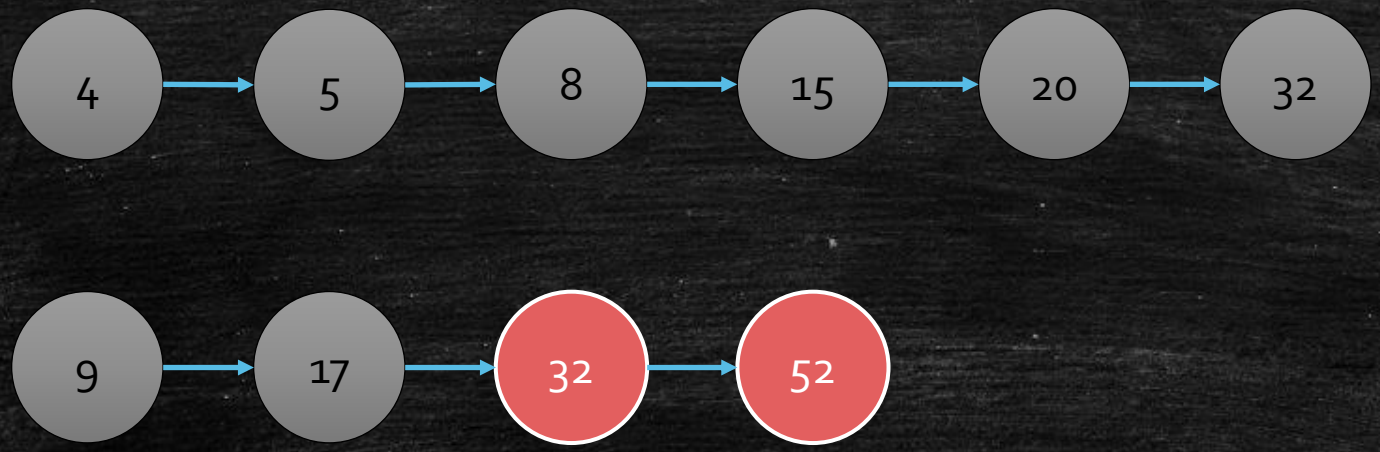
- Use two priority queue to implement.



- Use two priority queue to implement.



- Use two priority queue to implement.



Super Fun Story!!!

The story of the invention of Huffman codes is a great story that demonstrates that students can do better than professors. David Huffman (1925-1999) was a student in an electrical engineering course in 1951. His professor, Robert Fano, offered students a choice of taking a final exam or writing a term paper. Huffman did not want to take the final so he started working on the term paper. The topic of the paper was to find the most efficient (optimal) code. What Professor Fano did not tell his students was the fact that it was an open problem and that he was working on the problem himself. Huffman spent a lot of time on the problem and was ready to give up when the solution suddenly came to him. The code he discovered was optimal, that is, it had the lowest possible average message length. The method that Fano had developed for this problem did not always produce an optimal code. Therefore, Huffman did better than his professor. Later Huffman said that likely he would not have even attempted the problem if he had known that his professor was struggling with it.

From:

https://www.maa.org/press/periodicals/convergence/discovery-of-huffman-codes

Is there a better encoding?

Is Huffman code optimal?

What have we proved before?

Huffman code is somehow optimal.

- Input: an alphabet A, a frequency function $f: A \to N$.
- **Lemma 1:** The greedy algorithm can find a min cost prefix-free binary tree.
- **Lemma 2:** Huffman encoding is equivalent to the min cost prefix-free binary tree.

Why do we need a prefix-free binary tree?

Why do we need a prefix-free binary tree?

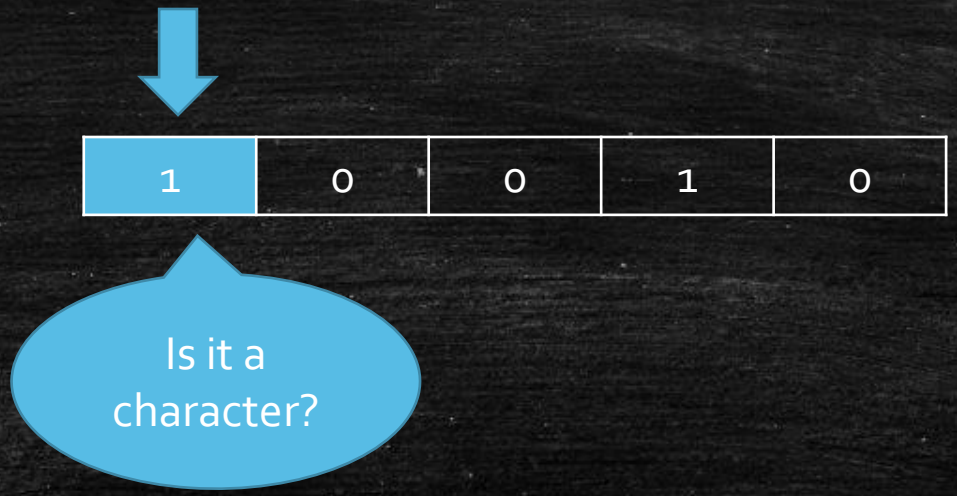
- If a code is not prefix free, we will have conflict with a **bit-by-bit** decoding.
- We want to decode the code of “BA”.

A	10
B	100

Why do we need a prefix-free binary tree?

- If a code is not prefix free, we will have conflict with a **bit-by-bit** decoding.
- We want to decode the code of “BA”.

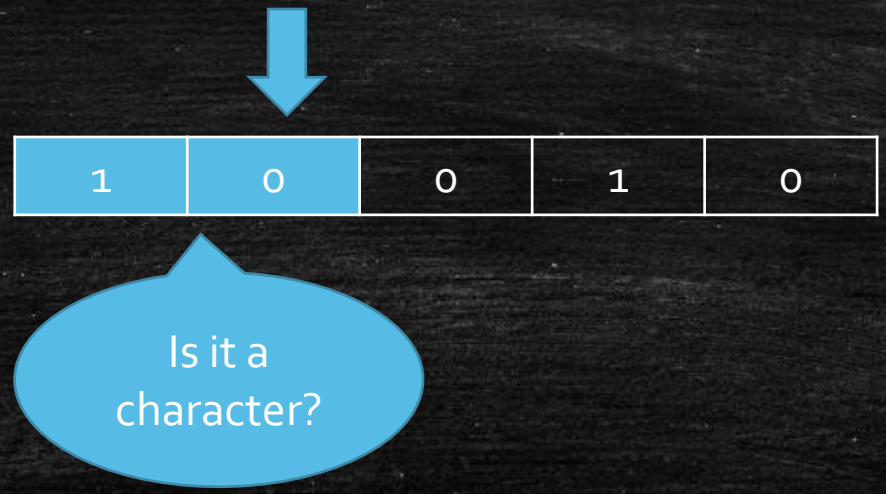
A	10
B	100



Why do we need a prefix-free binary tree?

- If a code is not prefix free, we will have conflict with a **bit-by-bit** decoding.
- We want to decode the code of “BA”.

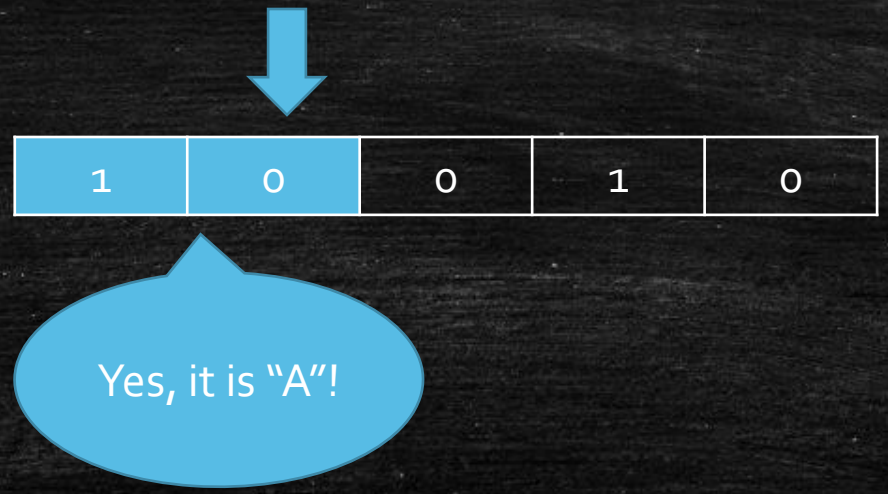
A	10
B	100



Why do we need a prefix-free binary tree?

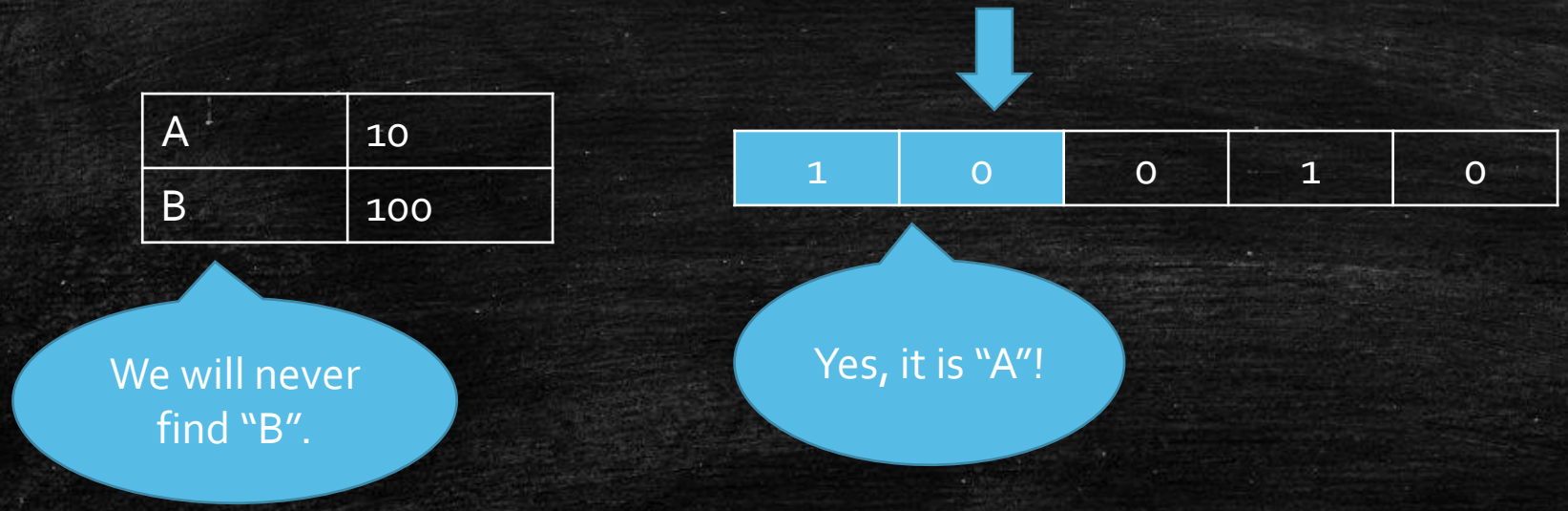
- If a code is not prefix free, we will have conflict with a **bit-by-bit** decoding.
- We want to decode the code of “BA”.

A	10
B	100



Why do we need a prefix-free binary tree?

- If a code is not prefix free, we will have conflict with a **bit-by-bit** decoding.
- We want to decode the code of "BA".



We are talking prefix-free based on binary code.

A 3-ary Code

- We do not have conflicts on this 3-nary code!

	binary	3-nary
A	10	2
B	100	11

When 3-ary code is better?

- We have 3 characters with same frequency.
- Huffman code (binary) waste some abilities.
  - Average length: (2+2+1)/3.

	frequency	binary	3-nary
A	10	00	0
B	10	01	1
C	10	1	2

- Binary cost: $20 + 20 + 10 = 50$; 3-ary cost: $30 \cdot \log_2 3 \approx 48$

When Huffman(binary) code is the best?

When Huffman(binary) code is the best?

- Case 1: We have 2^k characters with same frequency
- Intuition: The length of the code match the choices of the character.

	frequency	binary
A	10	00
B	10	01
C	10	10
D	10	11

Proof with Information Theory

- Formalize the message we want to encode.
  - We have an alphabet A, and a probability function $p(a)$ for $a \in A$.
  - The message is generated by m i.i.d. random variable X: $p(X = a) = p(a)$.
  - We want to minimized the expected bit length of our code.
- Entropy of the X
  - $Entropy(X) = \sum_{a \in A} p(a) \cdot \log(\frac{1}{p(a)})$.
  - Entropy is the minimized expected length to encode X.

Case 1

- Entropy of the X
  - $Entropy(X) = \sum_{a \in A} p(a) \cdot \log(\frac{1}{p(a)})$.
- Case 1: We have 2^k characters with same frequency
- Entropy of $X = 2^{\mathrm{k}} \cdot \frac{1}{2^k} \cdot \log 2^k = k$
- Cost of the code : Each code have length k.
- $Cost = m \cdot k = m \cdot Entropoy(X)$

When Huffman(binary) code is the best?

- Case 2: We have n characters with frequency
  - $m \cdot 2^{-(n-1)}, m \cdot 2^{-(n-1)}, m \cdot 2^{-(n-2)}, m \cdot 2^{-(n-3)}, \ldots, m \cdot 2^{-1}$
  - Example: Let $m = 40, n = 4$.
- Intuition: The length of the code match the probability of the character.

	frequency	binary
A	5	000
B	5	001
C	10	01
D	20	1

Case 2

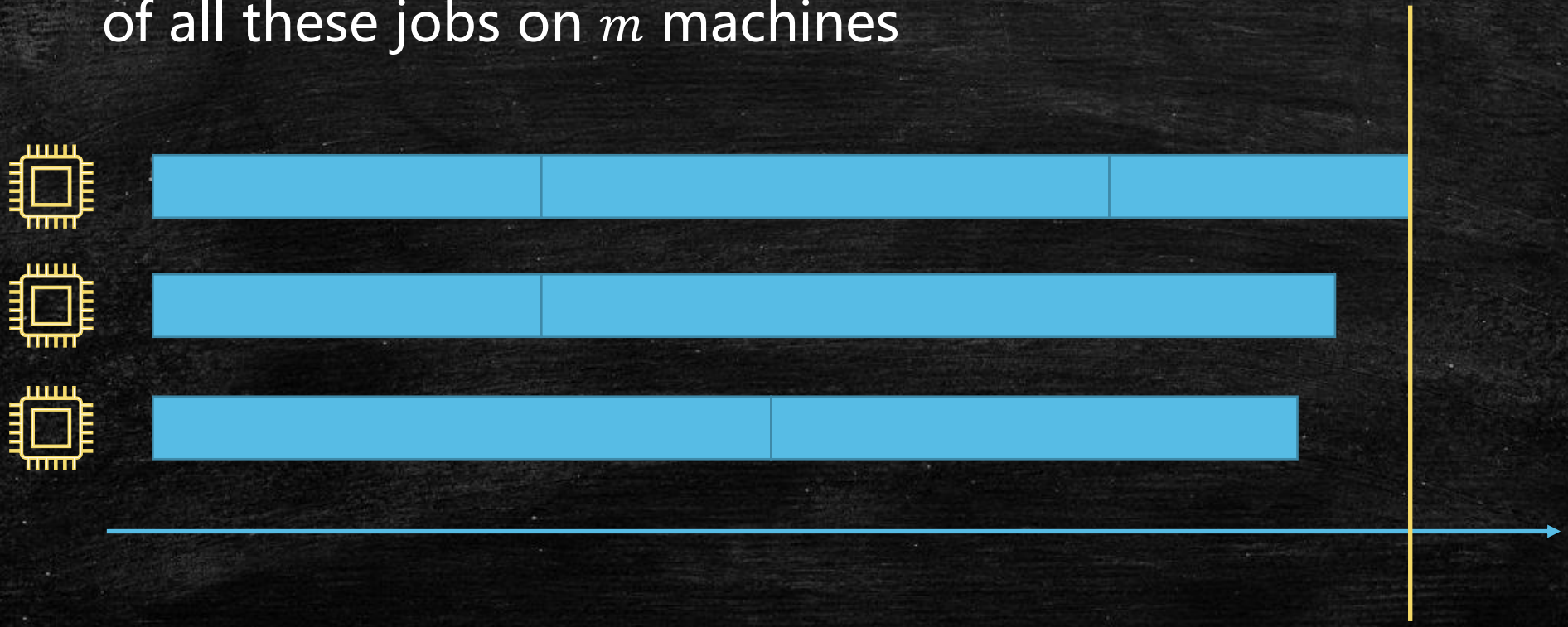
- Entropy of the X
  - $Entropy(X) = \sum_{a \in A} p(a) \cdot \log(\frac{1}{p(a)})$.
- Case 2: We have n characters with frequency
  - $m \cdot 2^{-(n-1)}, m \cdot 2^{-(n-1)}, m \cdot 2^{-(n-2)}, m \cdot 2^{-(n-3)}, \dots, m \cdot 2^{-1}$
- Cost of the code : Each code have length k.
- $\forall a \in A$: code length = $\log \frac{1}{p(a)}$.
- $Cost = m \cdot Entropy(X)$.

	frequency	binary
A	5	000
B	5	001
C	10	01
D	20	1

Even More Greedy!

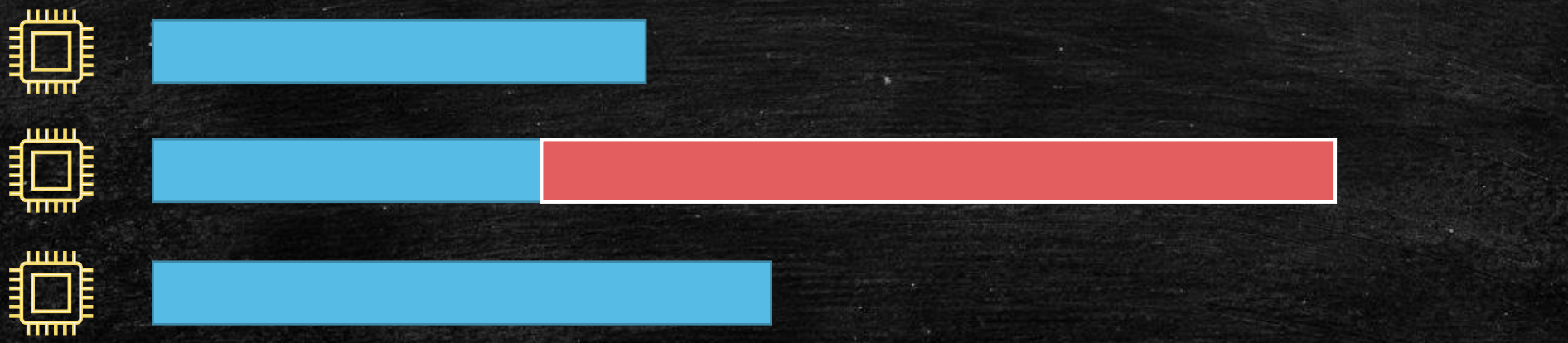
Makespan Minimization

- **Input:** m identical machines, n jobs with size p_i.
- **Output:** the minimized max completion time (**Makespan**) of all these jobs on m machines



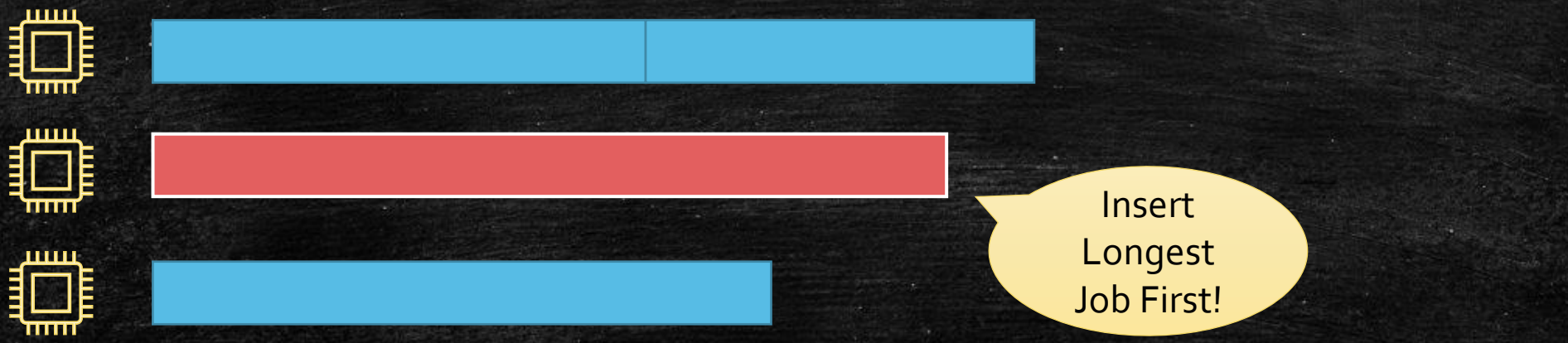
Greedy Attempt

- Schedule jobs to the earliest finished machine.
- In local view, it is optimal!



Is it optimal globally?

- No!
- Problem: the insertion order matters!



Greedy Attempt 2

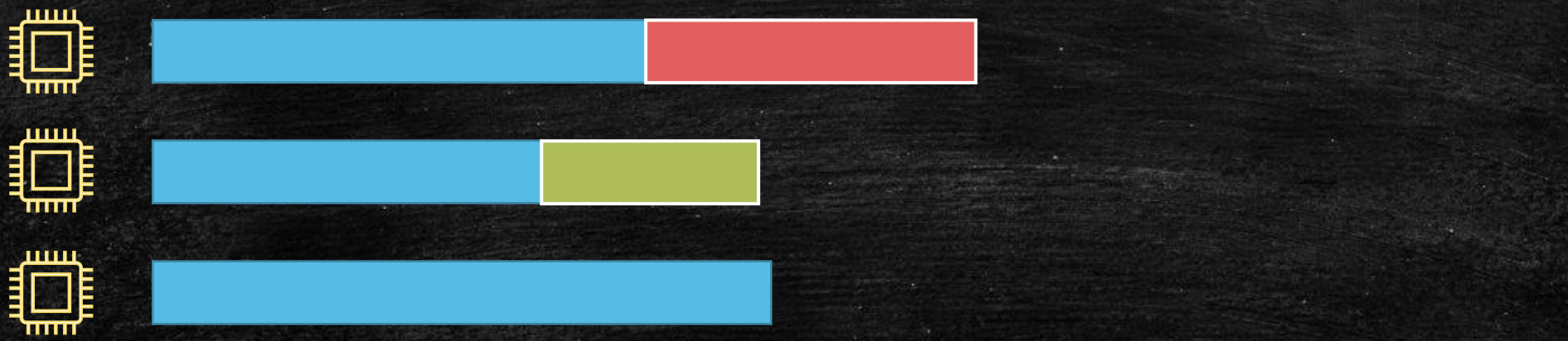
- **LPT** Algorithm
  - Longest Processing Time First.
  - Insert jobs into the earliest finished machine.

Proof of the correctness

- Assume we are still in OPT.
- We put the longest pending job onto the earliest finished machine.
- **Discussion! Are we still in an OPT?**

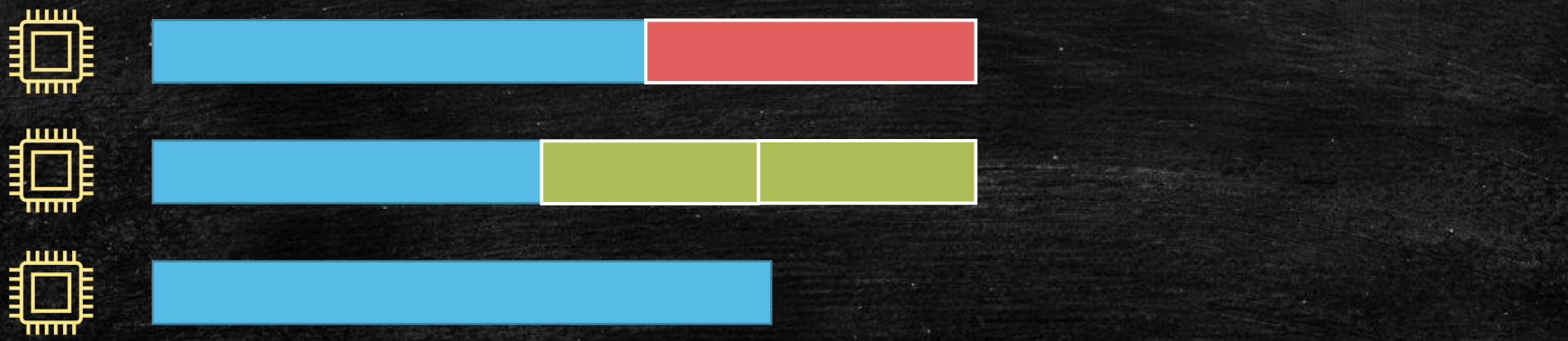
Proof of the correctness

- **Discussion! Are we still in an OPT?**
- Suppose not.
- We can swap red and green!



Proof of the correctness

- **Discussion! Are we still in an OPT?**
- Suppose not.
- But what if we have two green jobs?



Thinking: is it really bad?

Proof is a way for us to find the problems!

How to find a correct greedy algorithm?

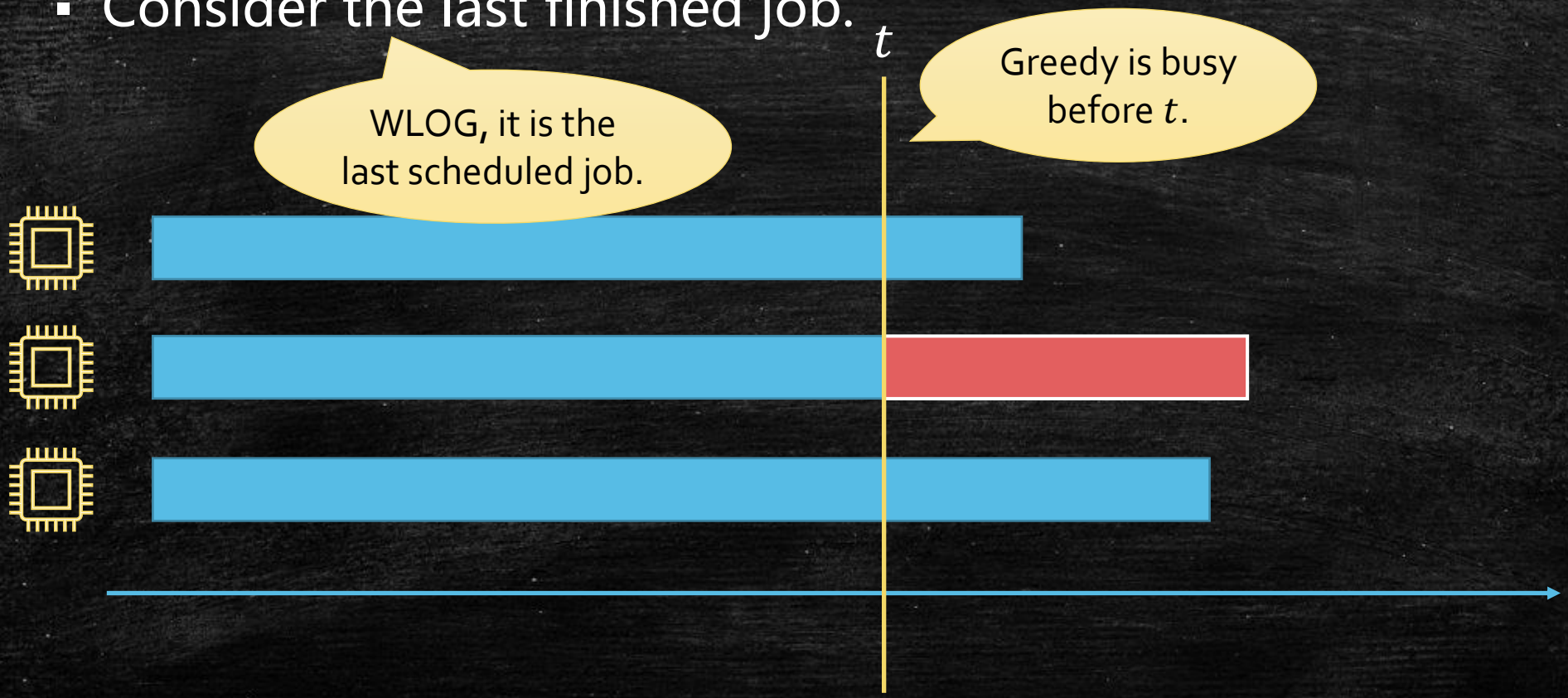
Makespan Minimization

- Makespan Minimization is a NP-hard problem.
- Find a poly time algorithm for it means $P = NP$.
- Is Simple Greedy or LPT very bad?
- At least, they are better than arbitrary scheduling.

Give a theoretical guarantee for Greedy and LPT.

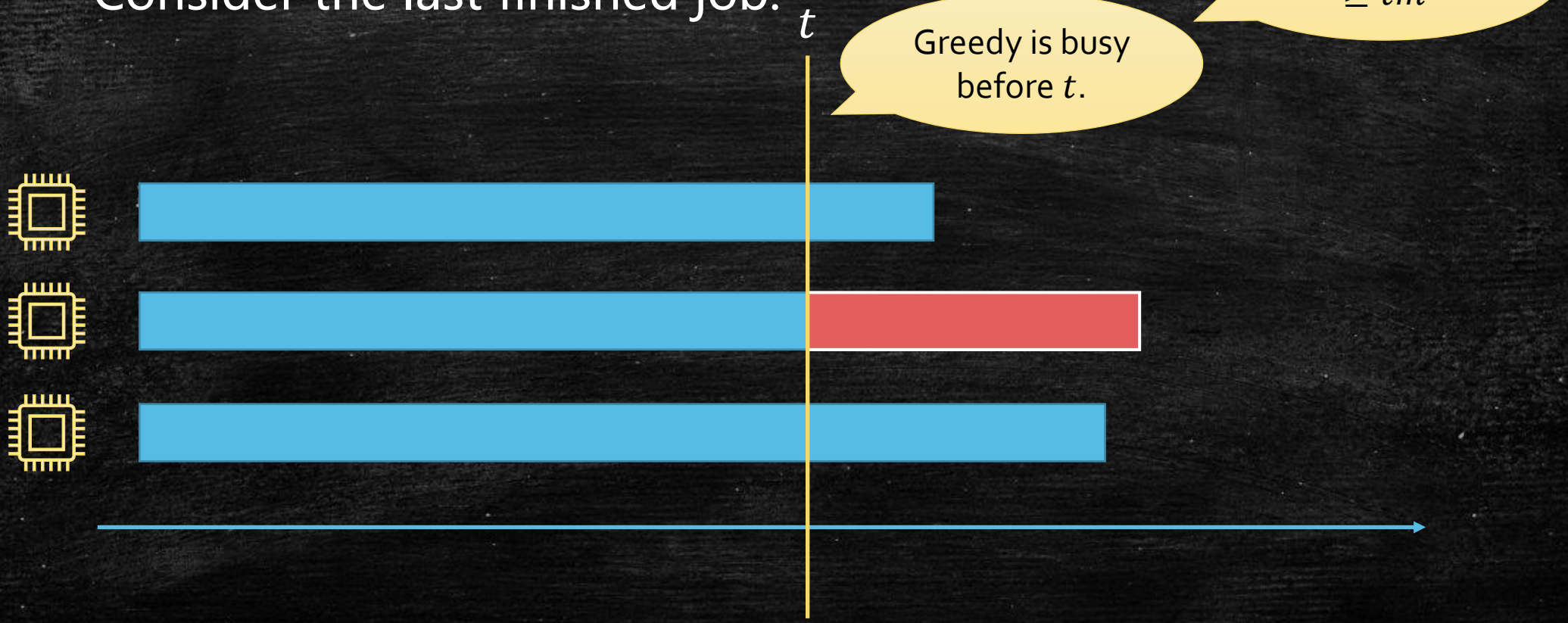
Relate Greedy Solution to OPT

- Schedule jobs to the earliest finished machine.
- Consider the last finished job.



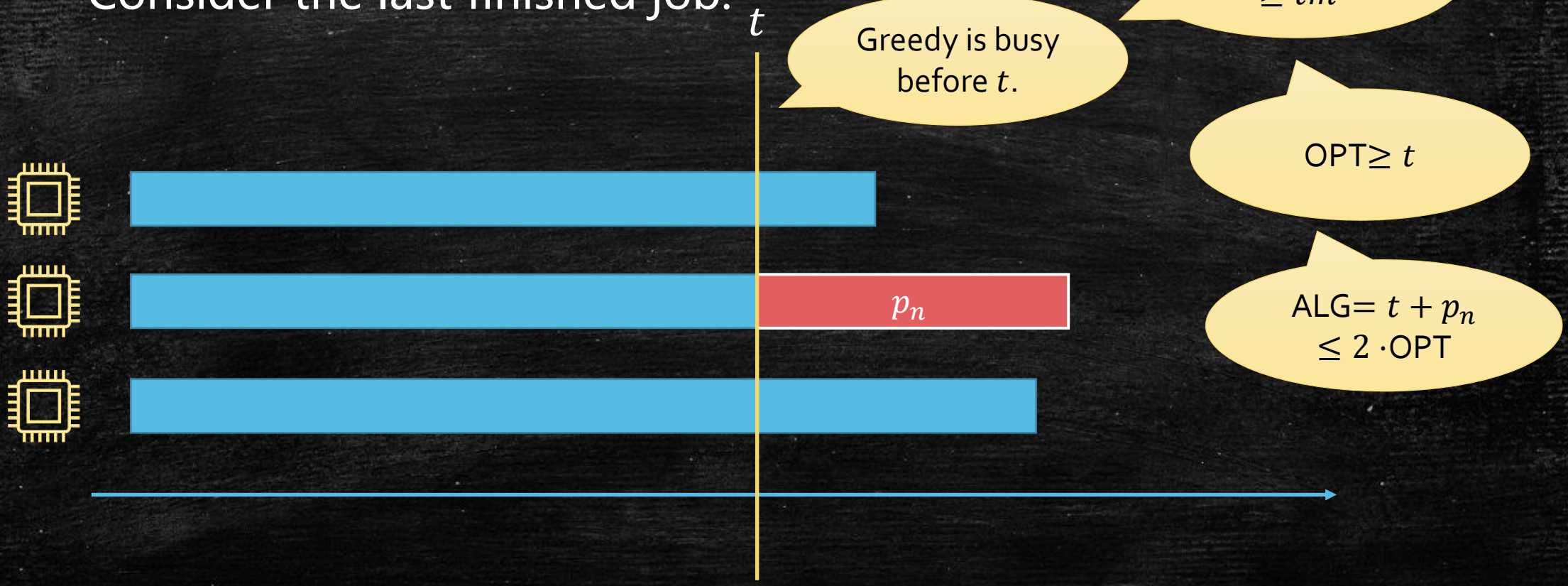
Relate Greedy Solution to OPT

- Schedule jobs to the earliest finished machine.
- Consider the last finished job.



Relate Greedy Solution to OPT

- Schedule jobs to the earliest finished machine.
- Consider the last finished job.

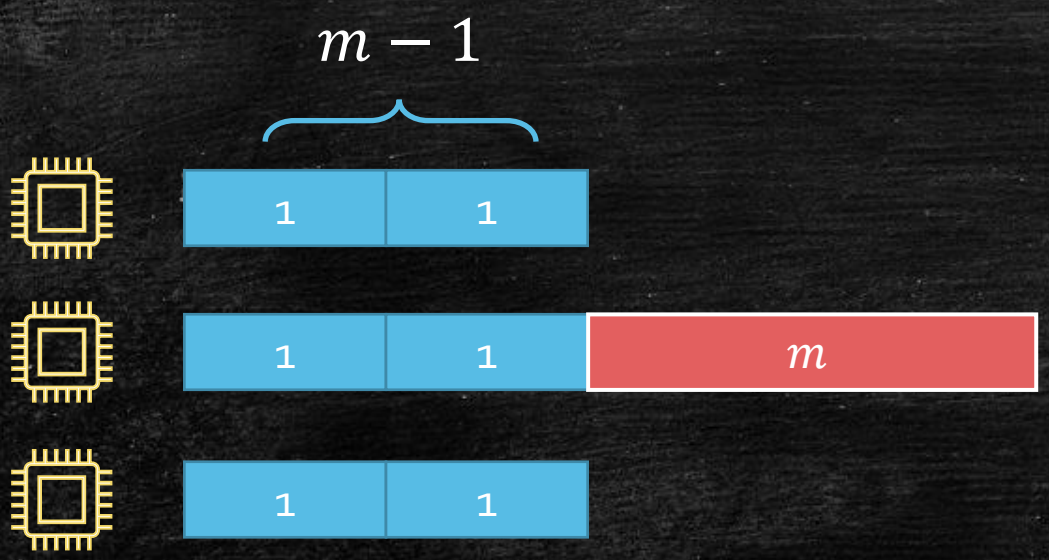


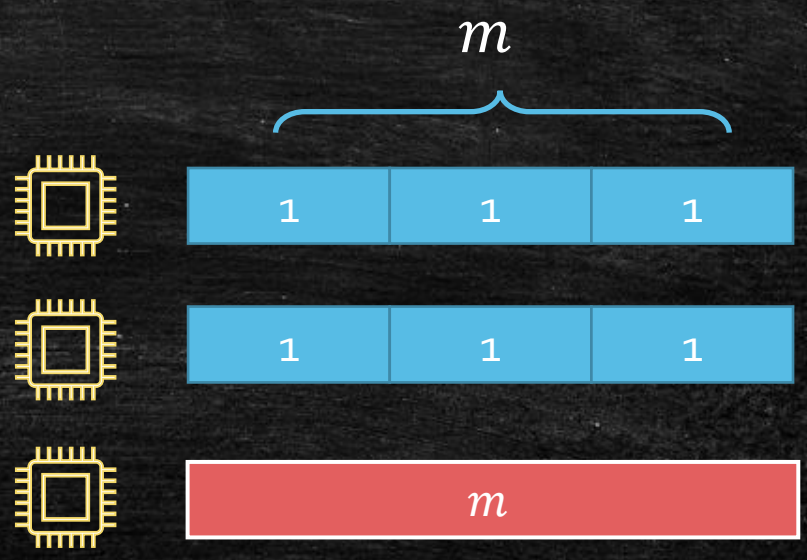
Relate Greedy Solution to OPT

- Greedy is at most 2 times of OPT in any input!
- We call Greedy is an Approximation Algorithm
  - Approximation Ratio is 2.
  - 2-approximate Algorithm.
- Can we make a better proof?

Counter-Example

Greedy get $2m - 1$, OPT get m.
Ratio $\to 2$ when m is large enough.

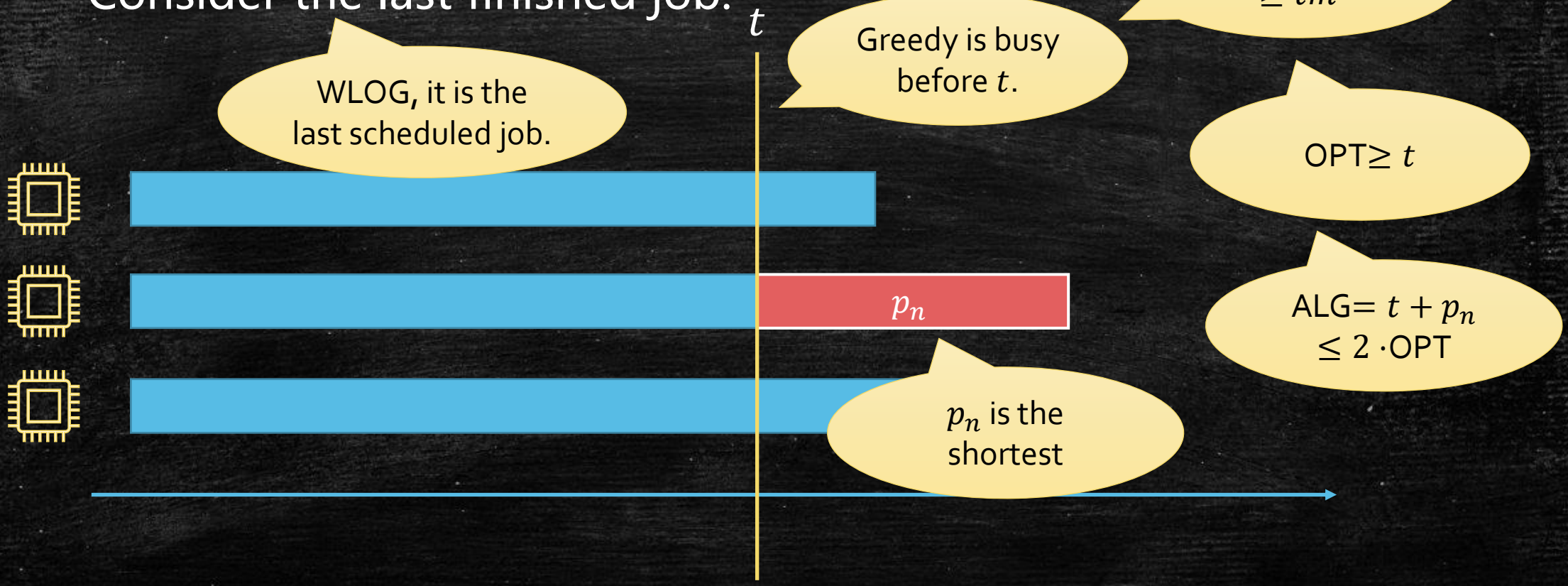




Can we improve the approximation ratio by **LPT**?

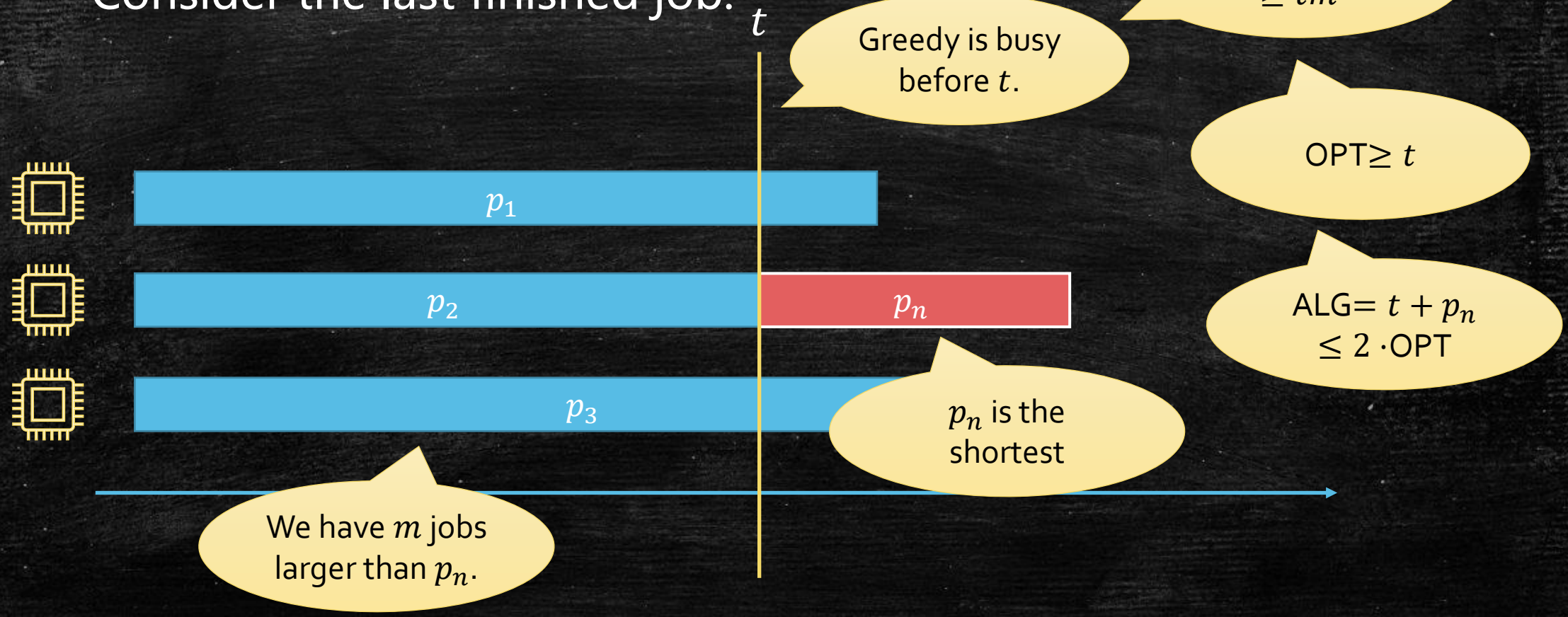
Relate LPT Solution to OPT

- Schedule jobs to the earliest finished machine.
- Consider the last finished job.



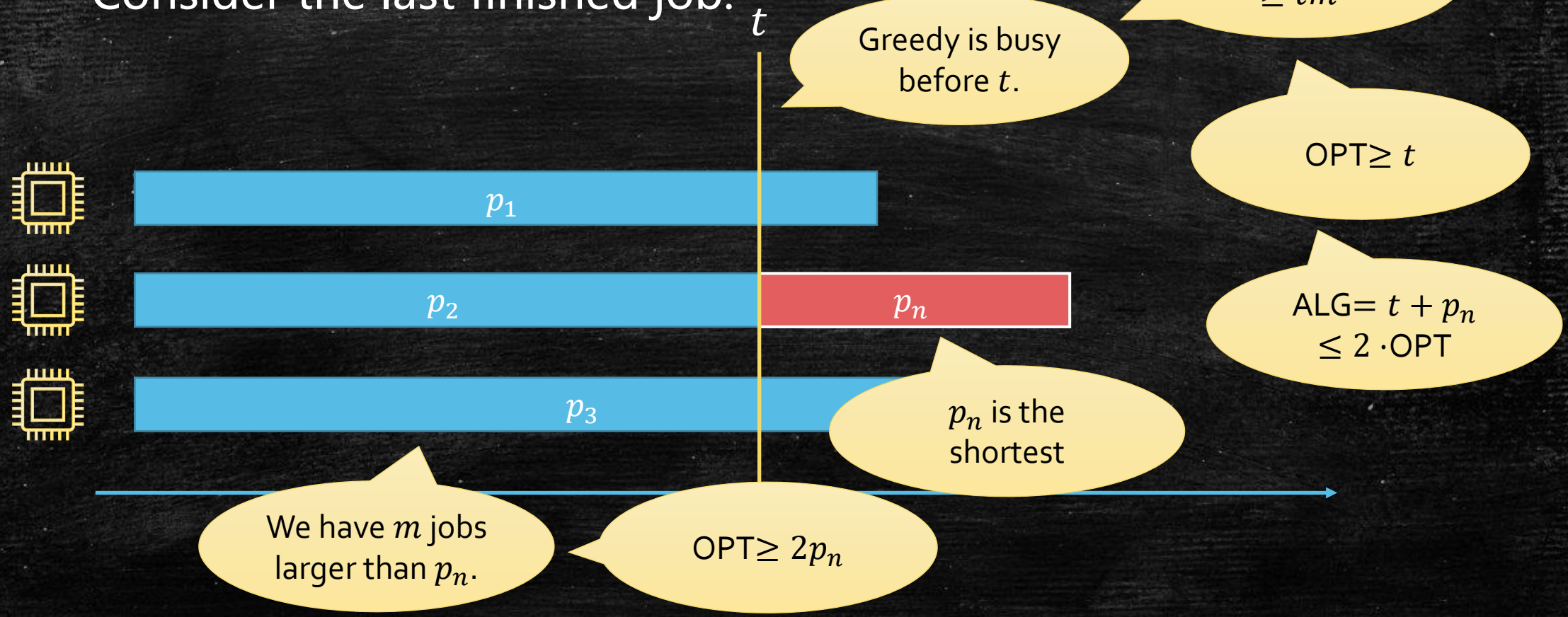
Relate LPT Solution to OPT

- Schedule jobs to the earliest finished machine.
- Consider the last finished job.



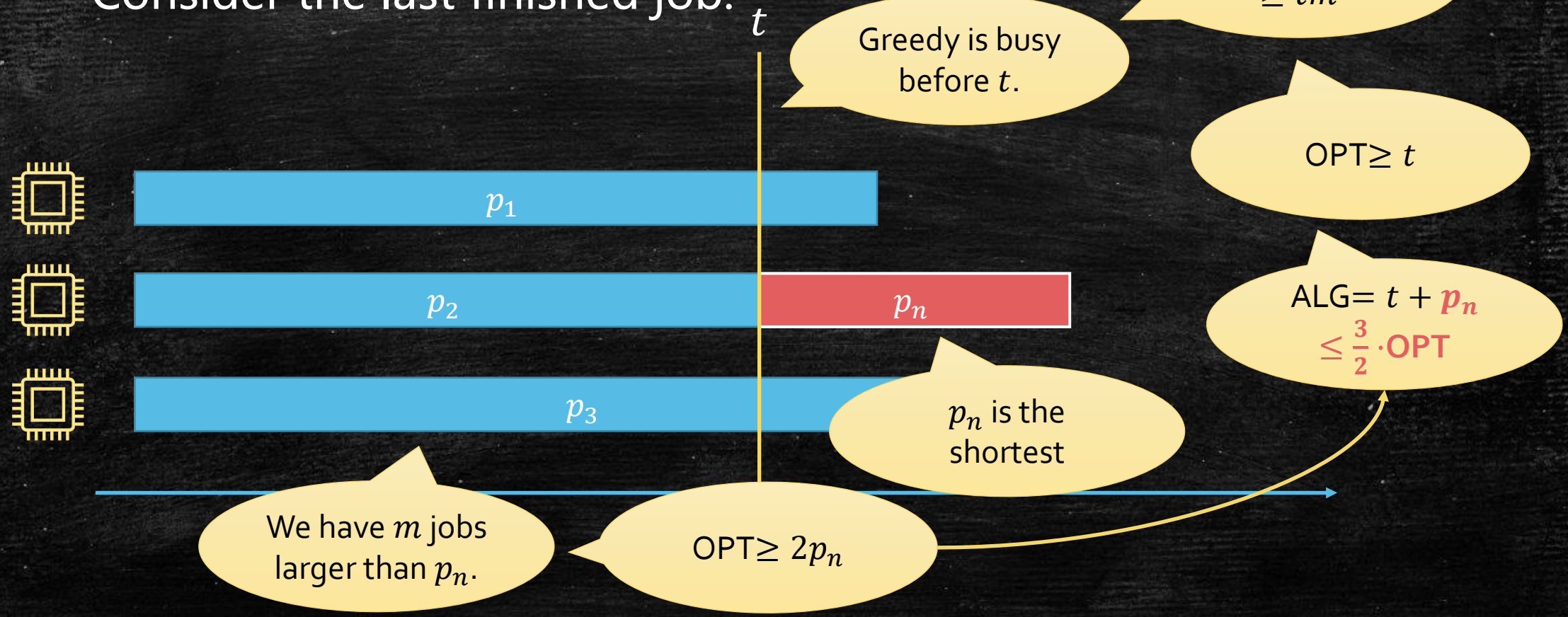
Relate LPT Solution to OPT

- Schedule jobs to the earliest finished machine.
- Consider the last finished job.



Relate LPT Solution to OPT

- Schedule jobs to the earliest finished machine.
- Consider the last finished job.



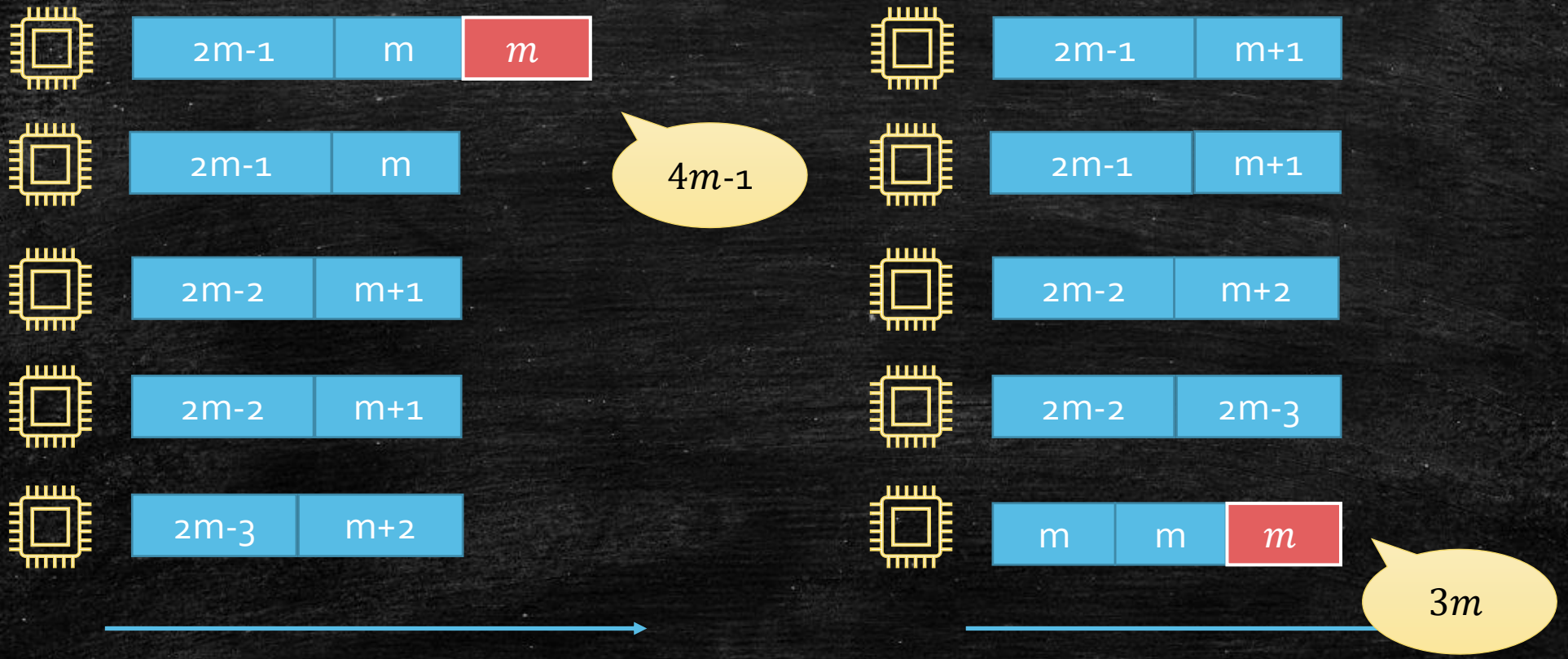
Are we done?

Is it finished?

- Not enough larger than p_n jobs?

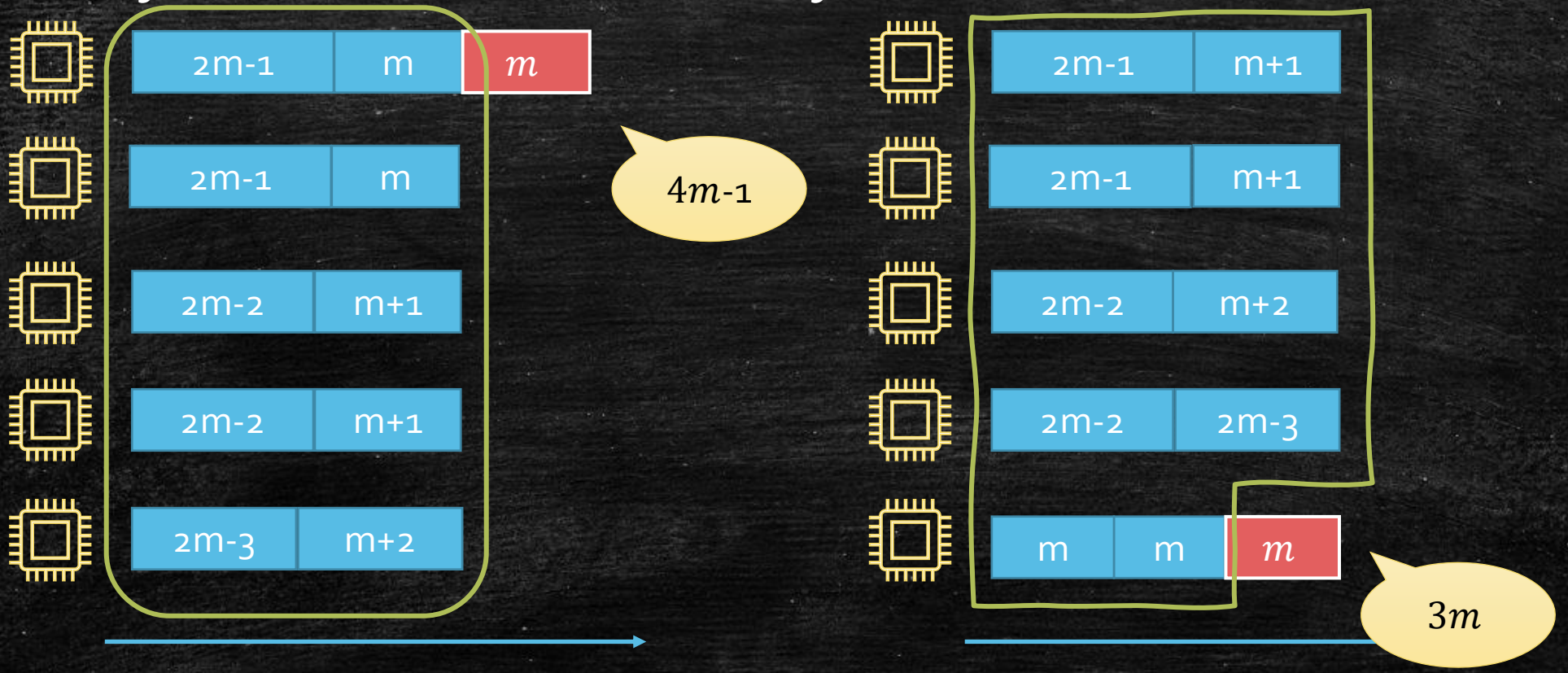
A counter example for LPT

- 2 jobs with size m~$2m-1$, 1 job with m.



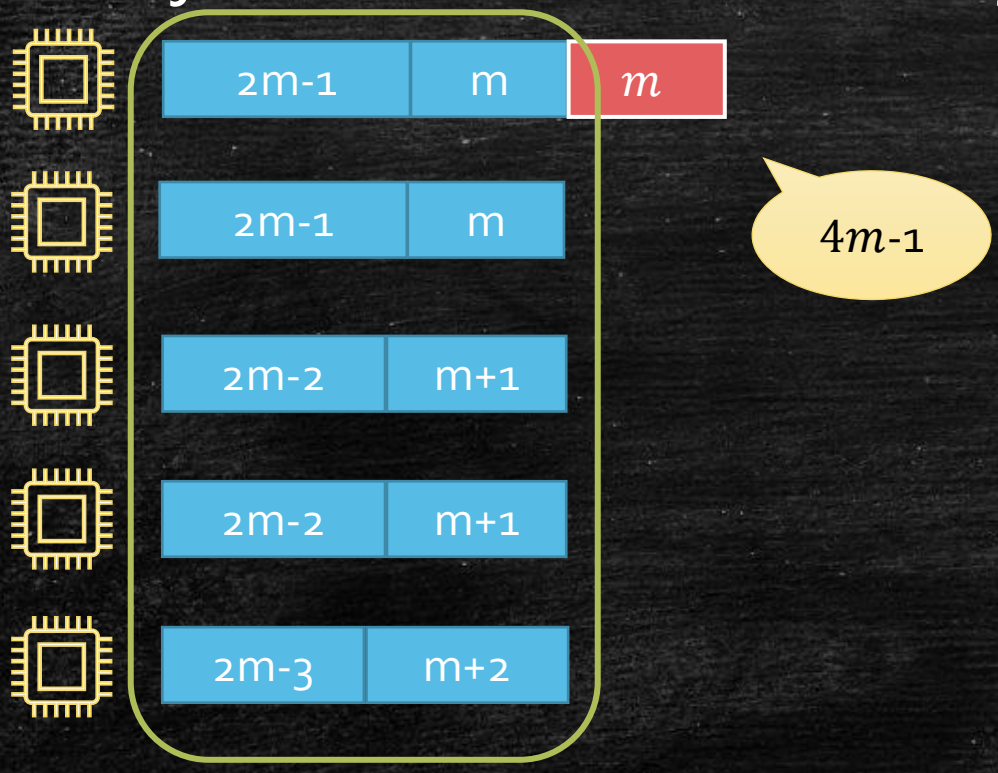
A counter example for LPT

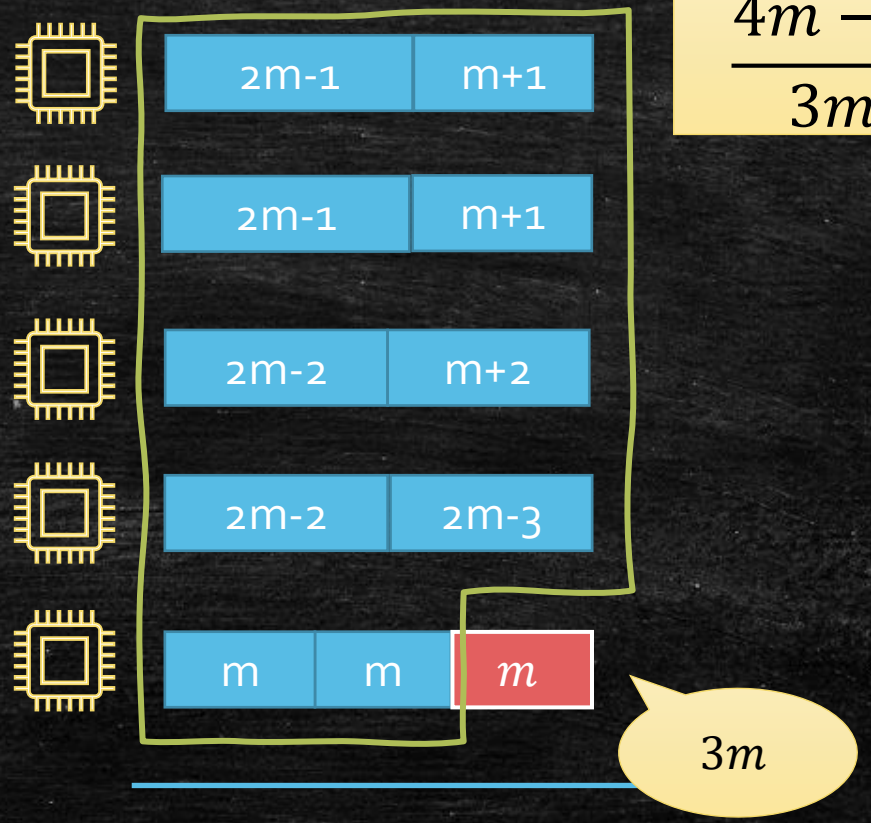
- 2 jobs with size m~$2m-1$, 1 job with m.



A counter example for LPT

- 2 jobs with size m~$2m-1$, 1 job with m.



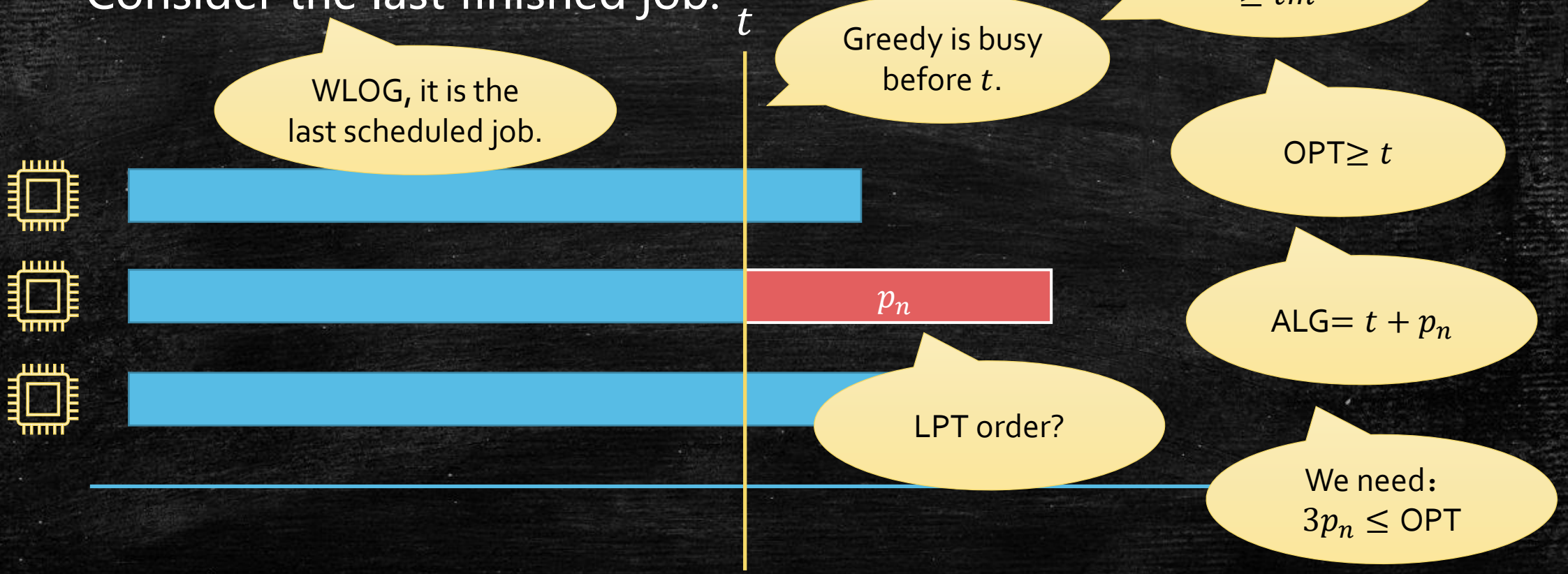


$$\frac{4m-1}{3m} \to 4/3$$

Can we improve the ratio to 4/3?

Connect LPT Solution to OPT

- Schedule jobs to the earliest finished machine.
- Consider the last finished job.

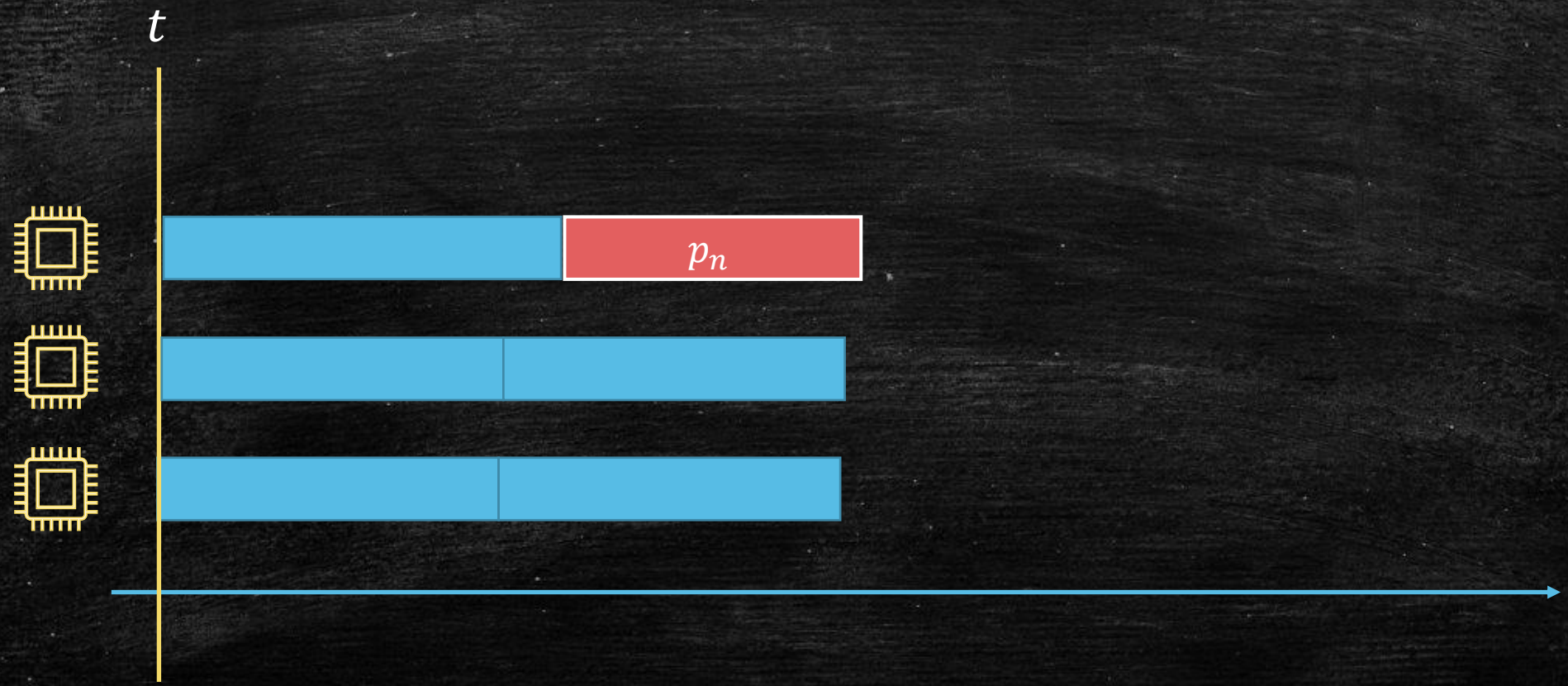


Prove: $3p_n \leq \text{OPT}$

- Recall p_n is the shortest job.
- Assume: $\text{OPT} < 3p_n$
- **Lemma 1:** Every machine in OPT only has two jobs.

OPT: Every machine in OPT only has two jobs.

- We have at most $2m$ jobs.

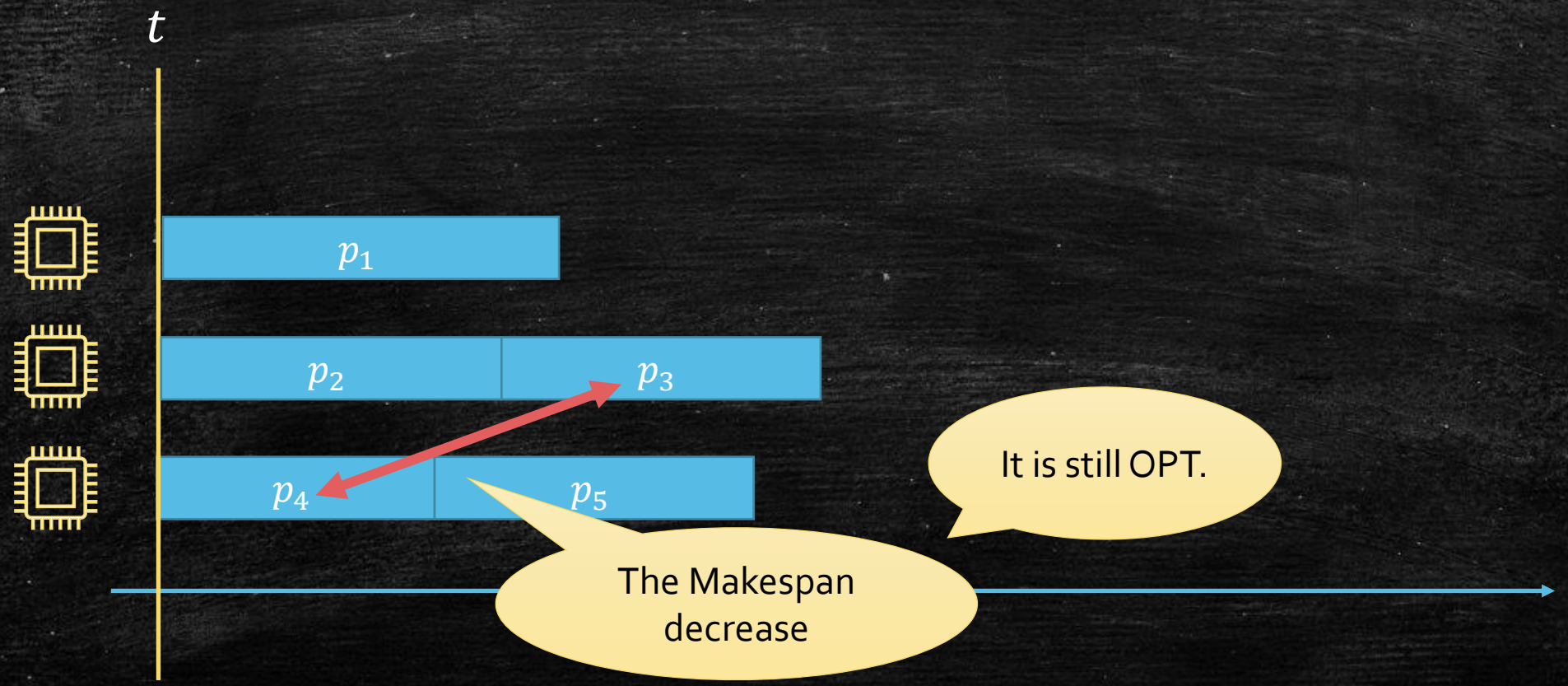


Run LPT for them is optimal.

- Proof:
- OPT is m job set $J_1, J_2, \ldots, J_m$, with size at most 2.
  - E.g., $J_1 = \{p_1, p_6\}$, $J_2 = \{p_2, p_5\}$ …
- **Lemma 2:** $p_1 \sim p_m$ are scheduled on different machines in OPT.

Lemma 1: $p_1 \sim p_m$ are scheduled on different machines in OPT.

- Suppose p_2 and p_3 is in the same machine in OPT.

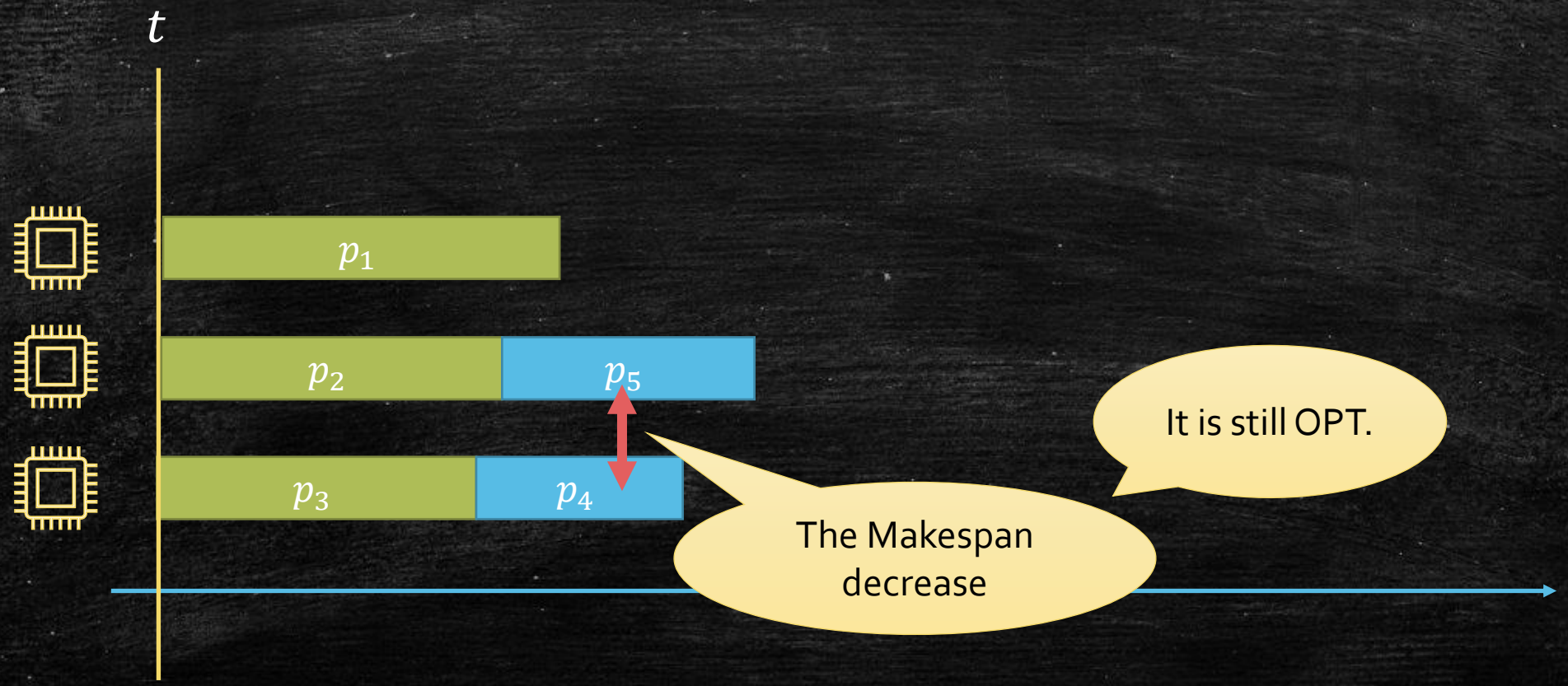


Run LPT for them is optimal.

- Proof:
- OPT is m job set $J_1, J_2, \ldots, J_m$, with size at most 2.
  - E.g., $J_1 = \{p_1, p_6\}$, $J_2 = \{p_2, p_5\}$ …
- **Lemma 2:** $p_1 \sim p_m$ are scheduled on different machines in OPT.
- **Lemma 3:** $p_m \sim p_n$ is correctly scheduled based on Lemma 1.

Lemma 1: $p_1 \sim p_m$ are scheduled on different machines in OPT.

- Suppose p_5 is scheduled later than p_4 in OPT.



Combining!

- Assume: $\mathrm{OPT} < 3p_n$
- **Lemma 1:** Every machine in OPT only has two jobs.
- **Lemma 2:** $p_1 \sim p_m$ are scheduled on different machines in OPT.
- **Lemma 3:** $p_m \sim p_n$ is correctly scheduled based on Lemma 1.
- **Lemma 4:** $\mathrm{ALG} = \mathrm{OPT}$ for $p_1 \sim p_n$ if $\mathrm{OPT} < 3p_n$.
- Conclusion
  - If $OPT \geq 3p_n$, $ALG = t + p_n \leq \frac{4}{3} OPT$
  - If $OPT < 3p_n$, $ALG = OPT$

More Questions

- How to improve the 2-analysis of **Greedy** to $2 - \frac{1}{m}$?
- How to improve the $\frac{4}{3}$ –analysis of **LPT** to $\frac{4}{3} - \frac{1}{3m}$?

Can we do better than Greedy?

- LPT: $\mathrm{ALG} \leq \frac{4}{3} \cdot \mathrm{OPT}$
- D. Hochbaum and D. Shmoys. JACM 1987
  - A PTAS for Makespan Minimization
  - An algorithm runs in $n^{O(\frac{1}{\epsilon^2})}$.
  - Performance guarantee: $\mathrm{ALG} \leq (1+\epsilon) \cdot \mathrm{OPT}$
- K. Jansen and L. Rohwedder. 2019
  - A EPTAS for Makespan Minimization
  - An algorithm runs in $f\left(\frac{1}{\epsilon}\right) n^c \to 2^{O\left(\frac{1}{\epsilon}\right) \log^2 \frac{1}{\epsilon}} + O(n)$
  - Performance guarantee: $\mathrm{ALG} \leq (1+\epsilon) \cdot \mathrm{OPT}$

Better: **FPTAS**
Runs in $poly\left(\frac{1}{\epsilon}\right) poly(n)$.

Today's goal

- Learn what is **Greedy**!
- Recap the difference of **Greedy** and **Divide and Conquer**.
- Learn to find the **problems** in a **Greedy attempt**.
- Learn to analyze Greedy Algorithm when it is **not optimal**.