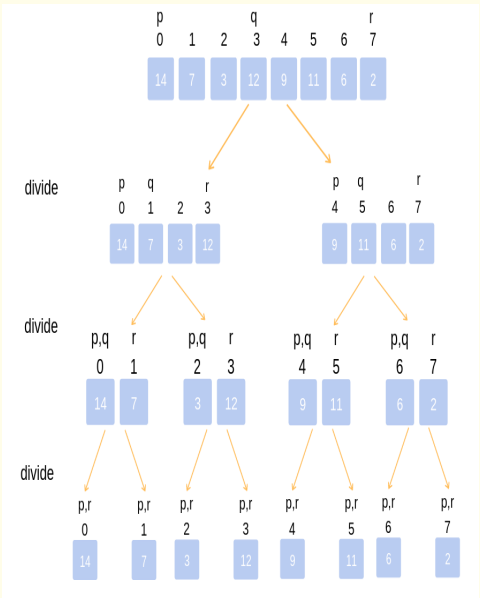
# Introduction to Algorithms
# Lecture 7 Amortized Analysis

Xue Chen
xuechen1989@ustc.edu.cn
2025 spring in

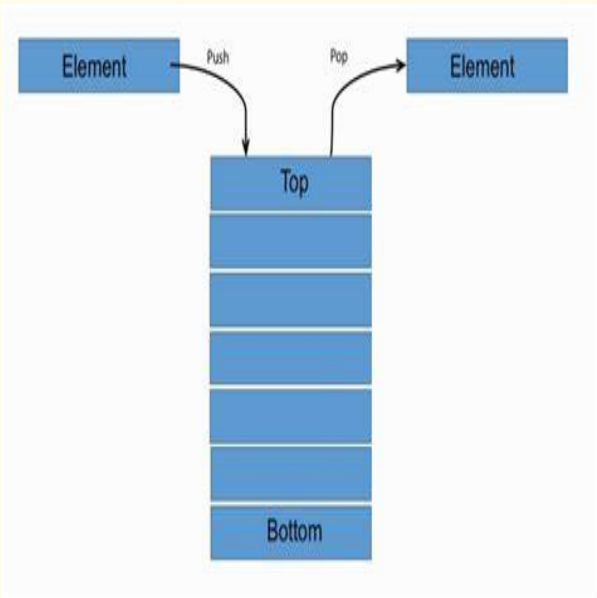
# Outline

# Introduction

1. A powerful method to analyze the running time of algorithms and data structures.
2. Various applications




(a) MERGESORT




(b) DATA STRUCTURE: STACK

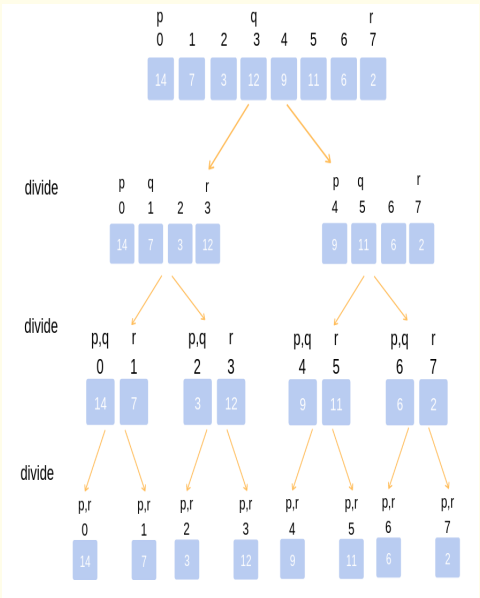
OPTIMAL-BST$(p, q, n)$

```
1  let e[1..n+1, 0..n], w[1..n+1, 0..n],
       and root[1..n, 1..n] be new tables
2  for i = 1 to n + 1
3      e[i, i − 1] = q_{i−1}
4      w[i, i − 1] = q_{i−1}
5  for l = 1 to n
6      for i = 1 to n − l + 1
7          j = i + l − 1
8          e[i, j] = ∞
9          w[i, j] = w[i, j − 1] + p_j + q_j
10         for r = i to j
11             t = e[i, r − 1] + e[r + 1, j] + w[i, j]
12             if t < e[i, j]
13                 e[i, j] = t
14                 root[i, j] = r
15 return e and root
```

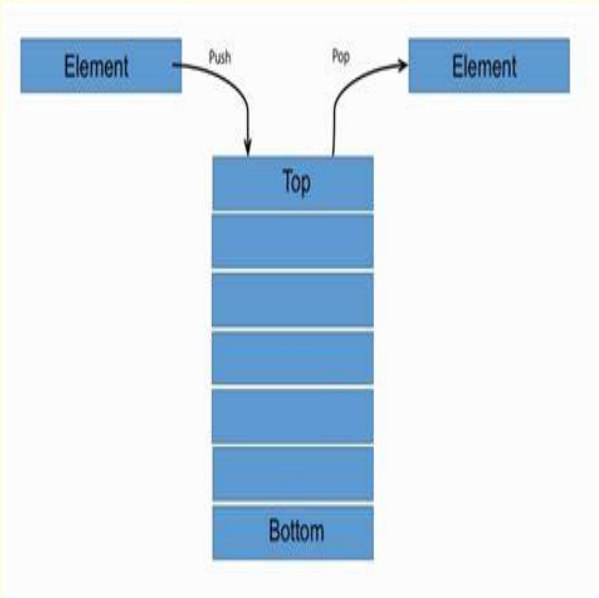
(c) IMPROVING DYNAMIC PROGRAMMING

# Introduction

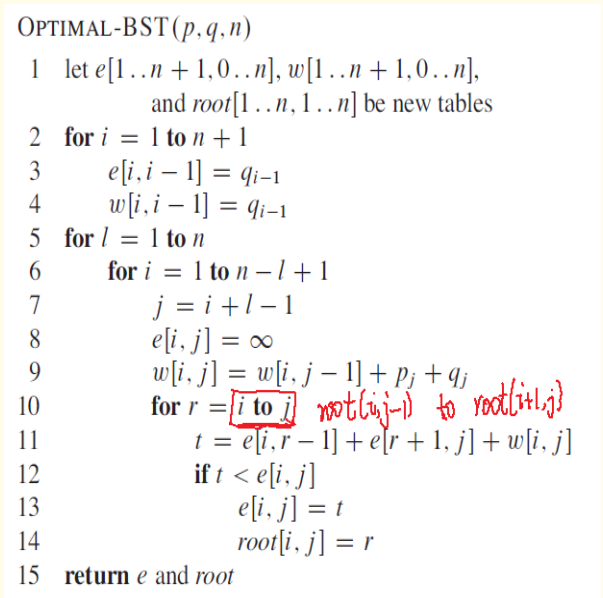
1. A powerful method to analyze the running time of algorithms and data structures.
2. Various applications




(d) MERGESORT




(e) DATA STRUCTURE: STACK



OPTIMAL-BST$(p, q, n)$

```
1  let e[1..n+1, 0..n], w[1..n+1, 0..n],
       and root[1..n, 1..n] be new tables
2  for i = 1 to n + 1
3      e[i, i − 1] = q_{i−1}
4      w[i, i − 1] = q_{i−1}
5  for l = 1 to n
6      for i = 1 to n − l + 1
7          j = i + l − 1
8          e[i, j] = ∞
9          w[i, j] = w[i, j − 1] + p_j + q_j
10         for r = i to j   root[i,j-1] to root[i+1,j]
11             t = e[i, r − 1] + e[r + 1, j] + w[i, j]
12             if t < e[i, j]
13                 e[i, j] = t
14                 root[i, j] = r
15 return e and root
```

(f) IMPROVING DYNAMIC PROGRAMMING

Let us introduce it formally and discuss its extensions.

# Overview

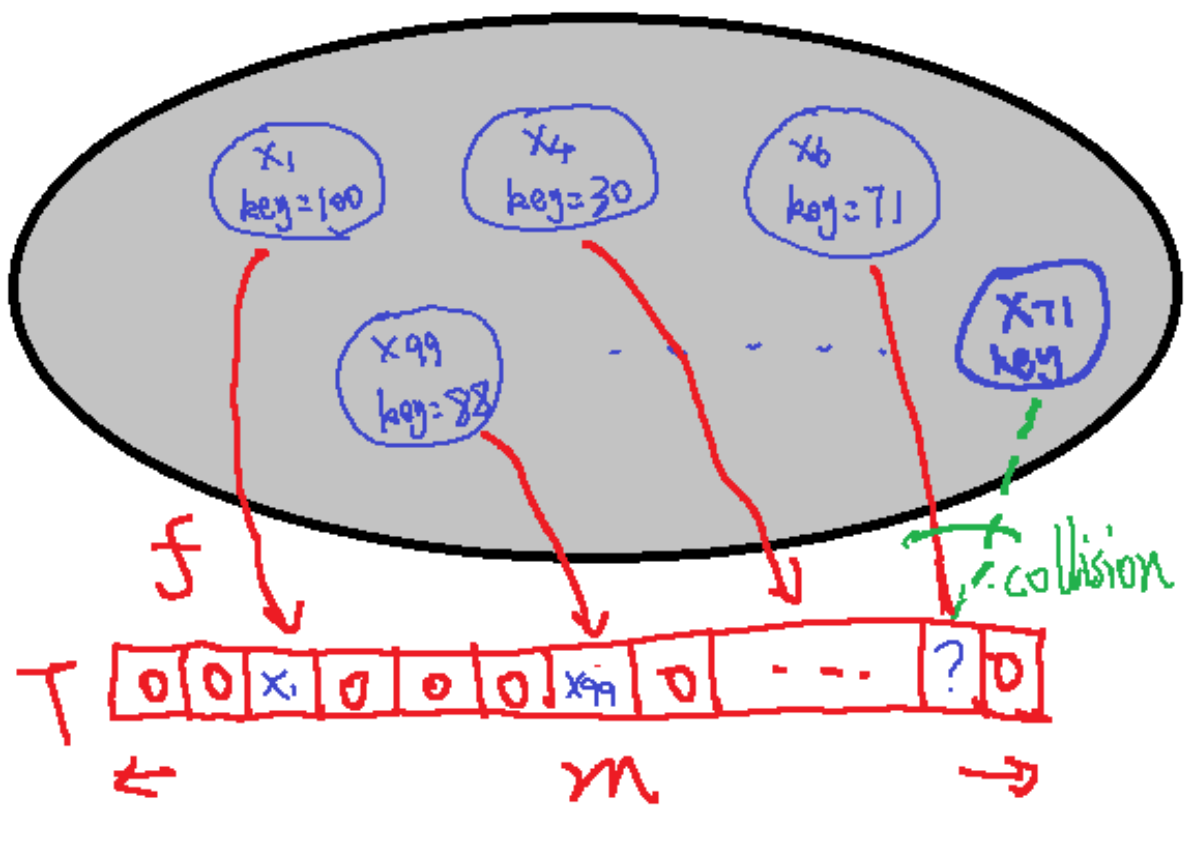
There are three major methods in amortized analysis:

1. Basic: Aggregate method
2. Advanced: Accounting method
3. Flexible: Potential method

Two examples: Dynamic Tables and Cartisian Trees

# Outline

# Problem Introduction




Goal: Make $m$ (the size of hash table) as small as possible, but never get overload

**Problem**

What if we don't know the proper size in advance?

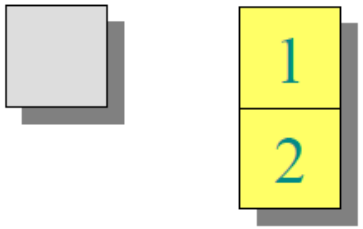
# Solution

Dynamic Tables — Whenever the table overloads, “grow” it by creating a larger table and move all items from the old table into the new one

# Solution

Dynamic Tables — Whenever the table overloads, “grow” it by creating a larger table and move all items from the old table into the new one

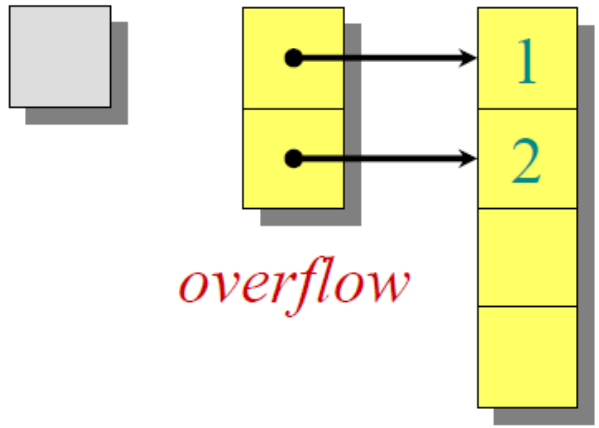
1. INSERT
2. INSERT



# Solution

Dynamic Tables — Whenever the table overloads, “grow” it by creating a larger table and move all items from the old table into the new one

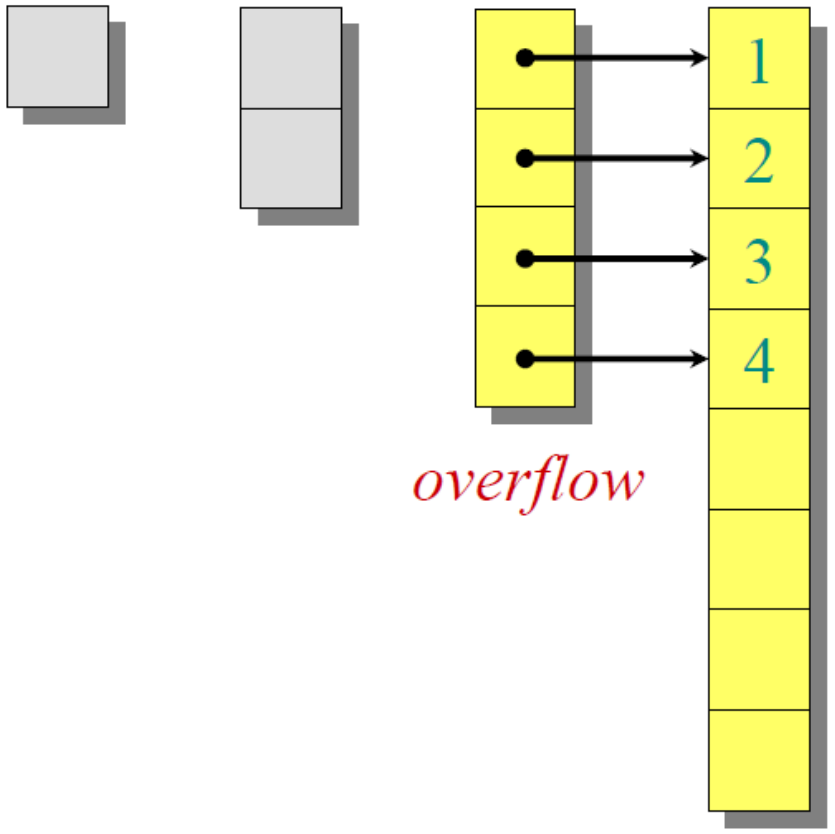
1. INSERT
2. INSERT
3. INSERT



# Solution

Dynamic Tables — Whenever the table overloads, “grow” it by creating a larger table and move all items from the old table into the new one

1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT



# Running Time

Consider $n$ insertions:

1. The worst-case time of one INSERT is $\Theta(n)$.
2. The total time of all insertions is $n \cdot \Theta(n) = \Theta(n^2)$.

# Running Time

Consider $n$ insertions:

1. The worst-case time of one INSERT is $\Theta(n)$.
2. The total time of all insertions is $n \cdot \Theta(n) = \Theta(n^2)$.

Not Tight! The total time is $\Theta(n)$ instead of $\Theta(n^2)$.

# Running Time

Consider $n$ insertions:

1. The worst-case time of one INSERT is $\Theta(n)$.
2. The total time of all insertions is $n \cdot \Theta(n) = \Theta(n^2)$.

**Not Tight!** The total time is $\Theta(n)$ instead of $\Theta(n^2)$.

## Analysis

$c_i$ denotes cost of the $i$th insertion, which is insertion + table change

| $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| $size_i$ | 1 | 2 | 4 | 4 | 8 | 8 | 8 | 8 | 16 | 16 |
| $c_i$ | 1 | 1 + 1 | 1 + 2 | 1 | 1 + 4 | 1 | 1 | 1 | 1 + 8 | 1 |

# Calculation

The total cost is $\Theta(n)$ and the average cost is $O(1)$.

$$
\begin{aligned}
\text{Cost of } n \text{ insertions} &= \sum_{i=1}^{n} c_i \\
&\le n + \sum_{j=0}^{\lfloor \lg(n-1) \rfloor} 2^j \\
&\le 3n \\
&= \Theta(n) .
\end{aligned}
$$

# Discussion

Amortized analyses: Proof strategies show that the average cost per operation is small, even though a single one could be expensive.

## Aggregate Method

Last 2 slides give an aggregate analysis
— we have seen similar ones in MERGESORT and DYNAMIC PROGRAMMING.

Though simple, lacks the flexibility and precision of the accounting and potential methods

# Accounting Method

Intro: Assign different charges to different operations (could be more or less)

1. Call the charged amount of an operation its amortized cost
2. The difference between amortized cost and actual time is the credit
3. Credit of one operation could be positive and negative
   — most credits are positive s.t. they cover those expensive operations with negative credits

# Accounting Method

Intro: Assign different charges to different operations (could be more or less)

1. Call the charged amount of an operation its amortized cost
2. The difference between amortized cost and actual time is the credit
3. Credit of one operation could be positive and negative
   — most credits are positive s.t. they cover those expensive operations with negative credits

## Formal Definition

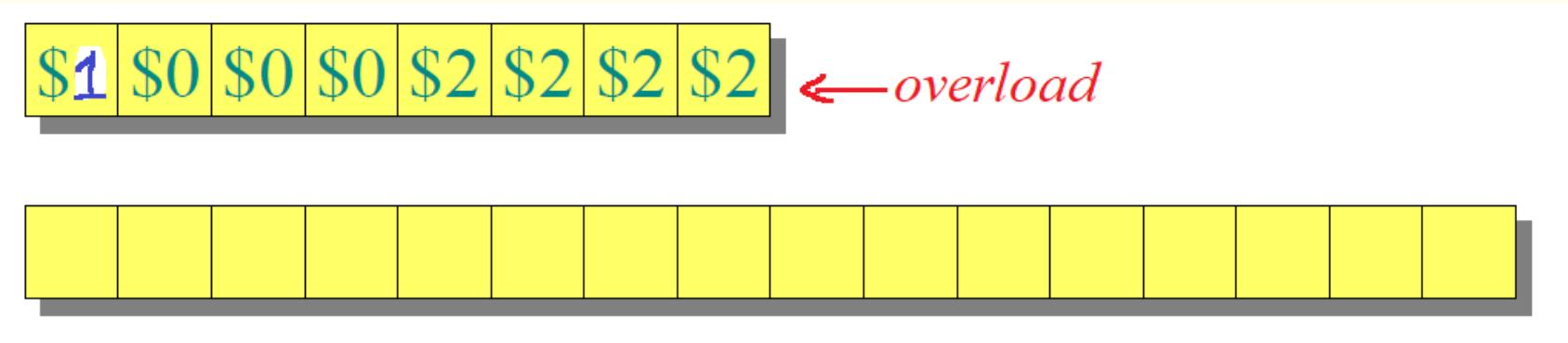
Let $c_i$ be the actual cost of operation $i$ and $\hat{c}_i$ be its amortized cost:

1. $\hat{c}_i$ is small for all $i$
2. $\sum_{i=1}^{n} \hat{c}_i \geqslant \sum_{i=1}^{n} c_i$ for any $n$

# Back to Dynamic Table

Charge $\hat{c}_i = 3$ for every INSERT.

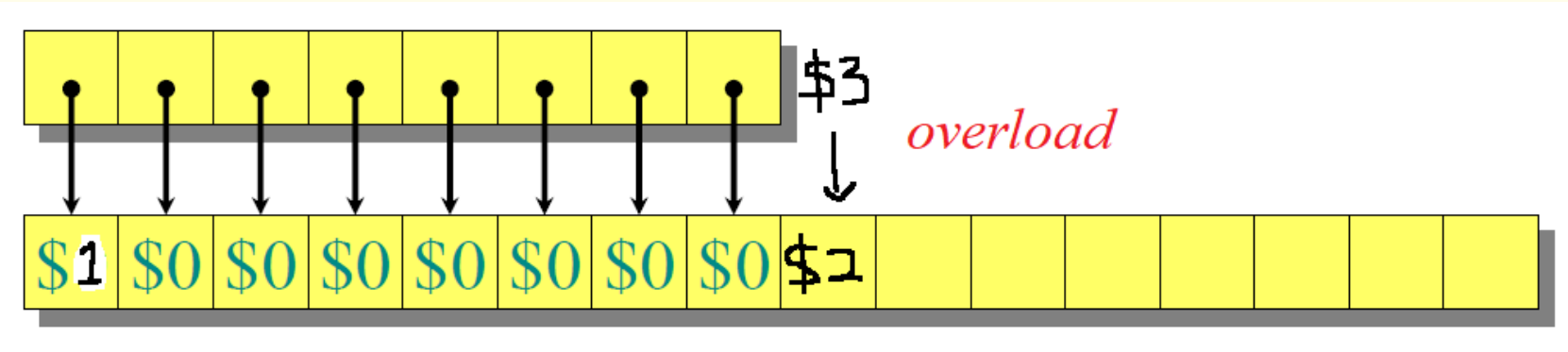
1. 1 pays for the immediate insertion.
2. 2 is stored for later table change — 1 for a recent item and 1 for an old item



# Back to Dynamic Table

Charge $\hat{c}_i = 3$ for every INSERT.

1. 1 pays for the immediate insertion.
2. 2 is stored for later table change — 1 for a recent item and 1 for an old item



# Accounting Analysis

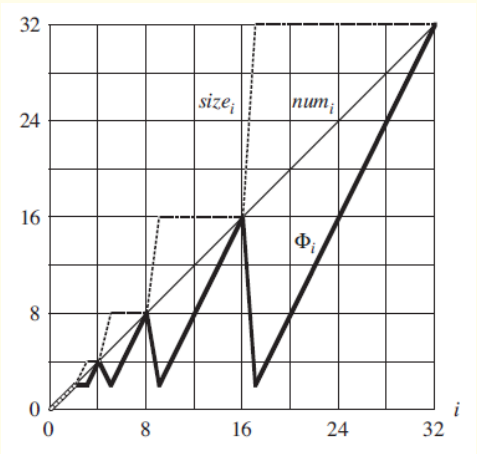
Bank balance is always non-negative

| $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| $size_i$ | 1 | 2 | 4 | 4 | 8 | 8 | 8 | 8 | 16 | 16 |
| $c_i$ | 1 | 2 | 3 | 1 | 5 | 1 | 1 | 1 | 9 | 1 |
| $\hat{c}_i$ | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| $bank_i$ | 2 | 3 | 3 | 5 | 3 | 5 | 7 | 9 | 3 | 5 |

# Potential Method

Basic Idea: View the bank account as the potential energy (a la physics) of the table

1. Start with an initial table (data structure/status) $D_0$
2. Operation $i$ changes $D_{i-1}$ to $D_i$ while its cost is $c_i$
3. Define a potential function $\Phi : D \to \mathbb{R}$ s.t. $\Phi(D_0) = 0$ and $\Phi(D_i) \geqslant 0$ for all $i \geqslant 1$

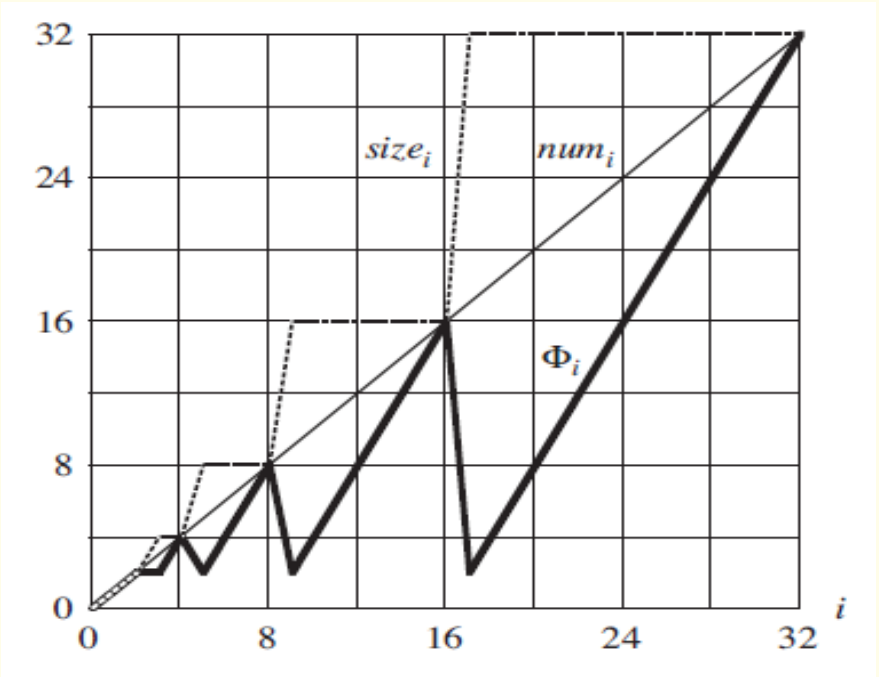



4. Goal: Minimize the amortized time $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

# More Notation

Let $\Delta(\Phi_i) = \Phi(D_i) - \Phi(D_{i-1})$ be the potential difference.

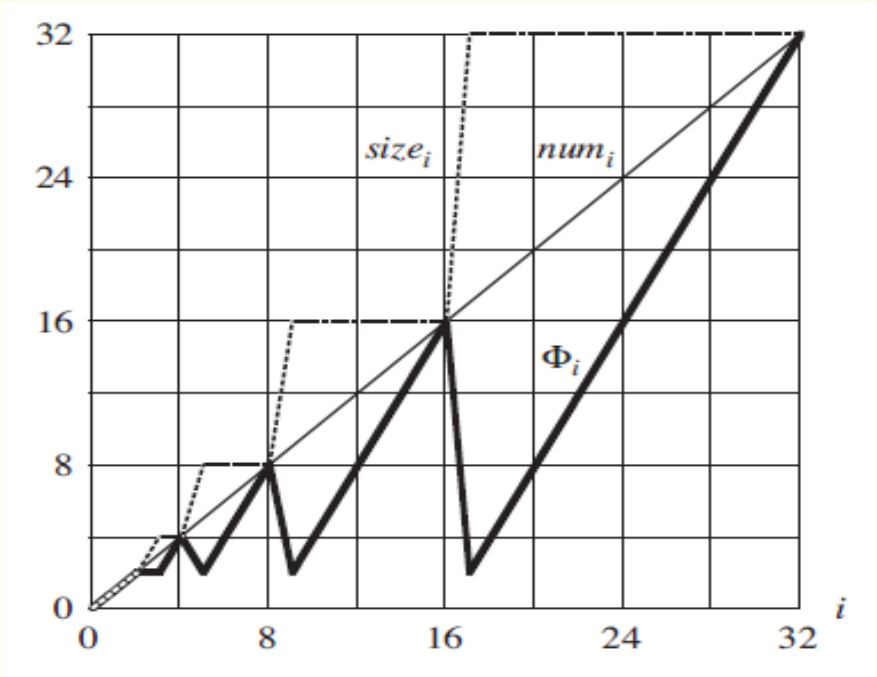
1. $\Delta(\Phi_i) > 0$ indicates $\hat{c}_i > c_i$ s.t. it stores energy for later use
2. $\Delta(\Phi_i) < 0$ indicates $\hat{c}_i < c_i$ s.t. it consumes energy



# More Notation

Let $\Delta(\Phi_i) = \Phi(D_i) - \Phi(D_{i-1})$ be the potential difference.

1. $\Delta(\Phi_i) > 0$ indicates $\hat{c}_i > c_i$ s.t. it stores energy for later use
2. $\Delta(\Phi_i) < 0$ indicates $\hat{c}_i < c_i$ s.t. it consumes energy




## Cost

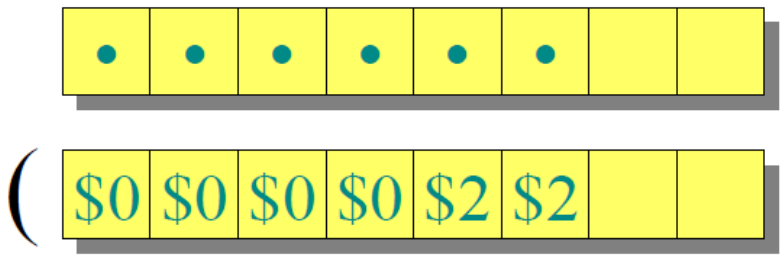
Amortize time $\sum_i \hat{c}_i = \sum_i (c_i + \Phi(D_i) - \Phi(D_{i-1})) = \sum_i c_i + \Phi(D_n) - \Phi(D_0)$
$\geqslant$ actual time $\sum_i c_i$.

# Back to Dynamic Table

One potential function for the current table $T$ is

$$\Phi(T) = 2T.num - T.size \text{ equivalent to } \Phi(D_i) = 2i - 2^{\lceil \log_2 i \rceil}$$

**Example:**




$\Phi = 2 \cdot 6 - 2^3 = 4$

(accounting method)

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\
&= 1 + i \cdot \mathbb{I}\{i = 2^k + 1\} + (2i - 2^{\lceil \log_2 i \rceil}) - (2(i-1) - 2^{\lceil \log_2 i - 1 \rceil}) \\
&= 1 + i \cdot \mathbb{I}\{i = 2^k + 1\} + 2 - 2^{\lceil \log_2 i \rceil} + 2^{\lceil \log_2 i - 1 \rceil} \\
&= O(1)
\end{aligned}$$

# Discussion

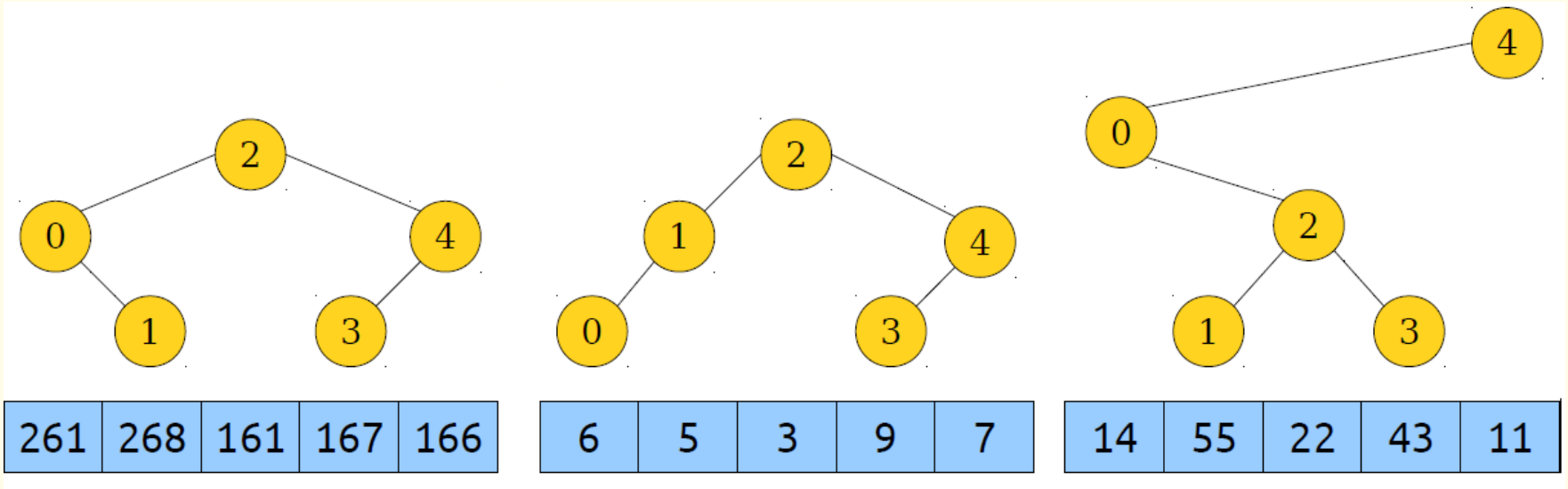
1. For many data structures (stack, binary search trees like Splay and Treap), amortized cost provide a clean statement
2. 3 methods: Aggregate method, accounting method, and Potential method.
3. Each method has some situation where it is arguably the simplest or most precise — Potential method is the most flexible one b.c. of $\Phi(D_i)$

# Outline

# Introduction

A Cartesian tree is a binary tree derived from an array:

1. Root stores the index of the minimum value
2. Its left and right children are Cartesian trees for the subarrays to the left and right



# Construction

1. Naive algorithm: $O(n^2)$
2. BST or Heap: $O(n \log n)$
3. Greedy Algorithm: $O(n)$

# Construction

1. Naive algorithm: $O(n^2)$
2. BST or Heap: $O(n \log n)$
3. Greedy Algorithm: $O(n)$

## Greedy Algorithm

Process $a_1, \ldots, a_n$ from left to right and maintain the tree for $a_1, \ldots, a_i$

1. If $a_i < a_{i+1}$, insert $v_{i+1}$ as the right child of $v_i$

# Construction

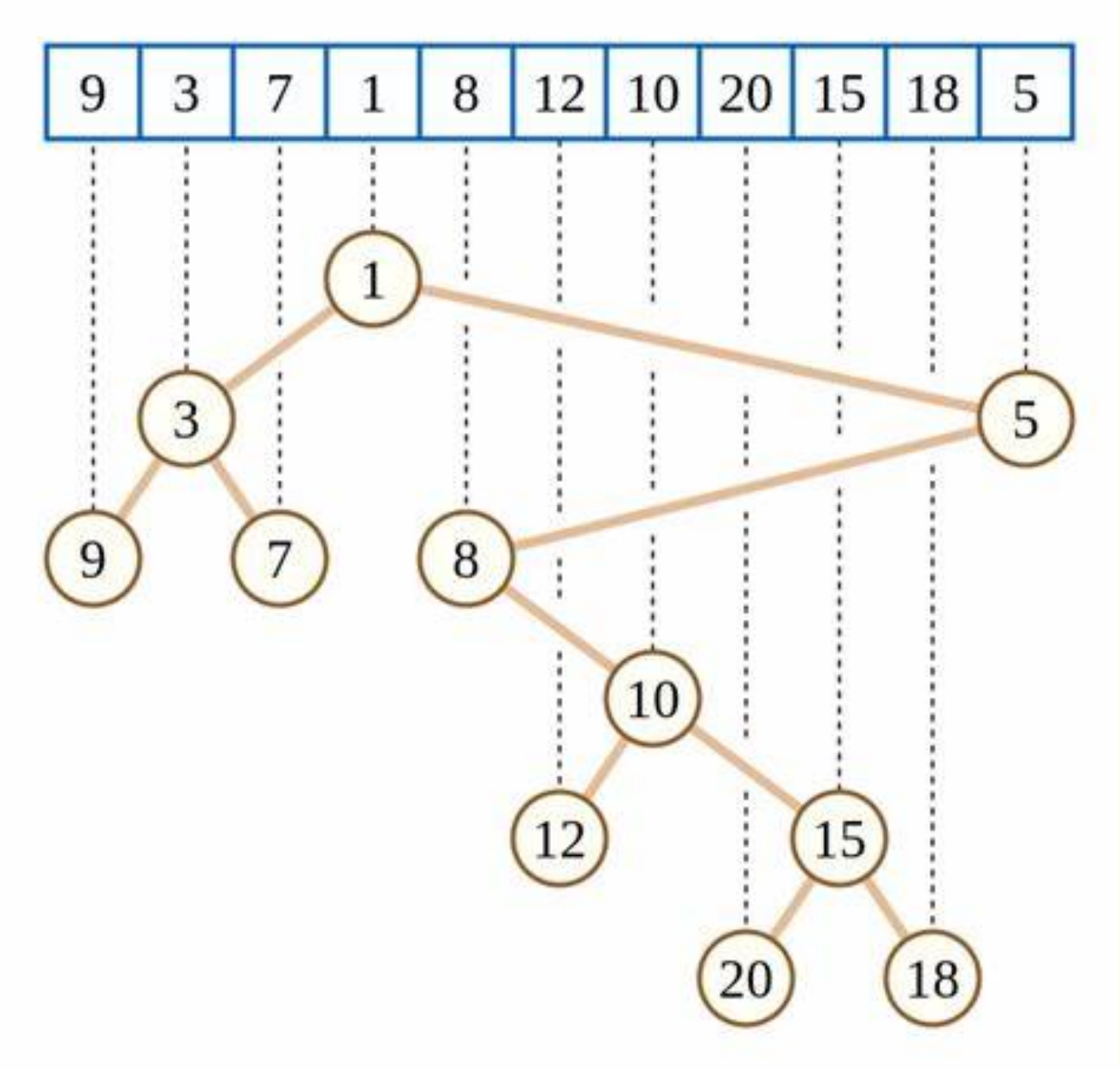
1. Naive algorithm: $O(n^2)$
2. BST or Heap: $O(n \log n)$
3. Greedy Algorithm: $O(n)$

## Greedy Algorithm

Process $a_1, \ldots, a_n$ from left to right and maintain the tree for $a_1, \ldots, a_i$

1. If $a_i < a_{i+1}$, insert $v_{i+1}$ as the right child of $v_i$
2. Otherwise consider $v_i$'s ancestors: Find the nearest ancestor $j$ with $a_j < a_{i+1}$ and insert $v_{i+1}$ as $v_j$'s right child (put $v_j$'s right child as $v_{i+1}$'s left child)
3. If no ancestor satisfies $a_j < a_{i+1}$, make $v_{i+1}$ as the root whose left child is the old root

# Example



# Analysis

Correctness follows by Induction: It always maintains the Cartesian tree of $a_1, \ldots, a_i$

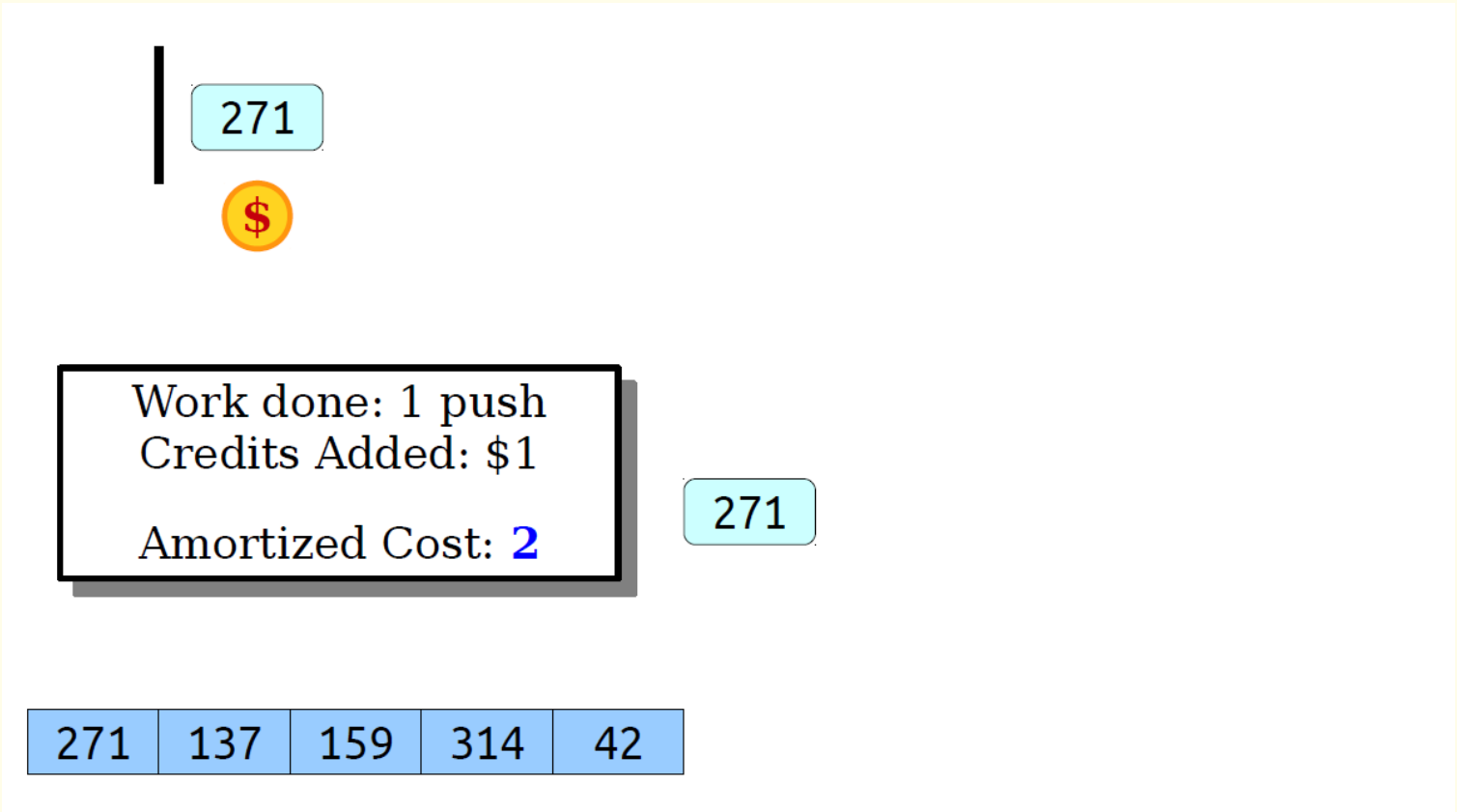
## Running Time

$O(n)$ — short answer is like a stack, each node gets at most one push and one pop

Let us try accounting method

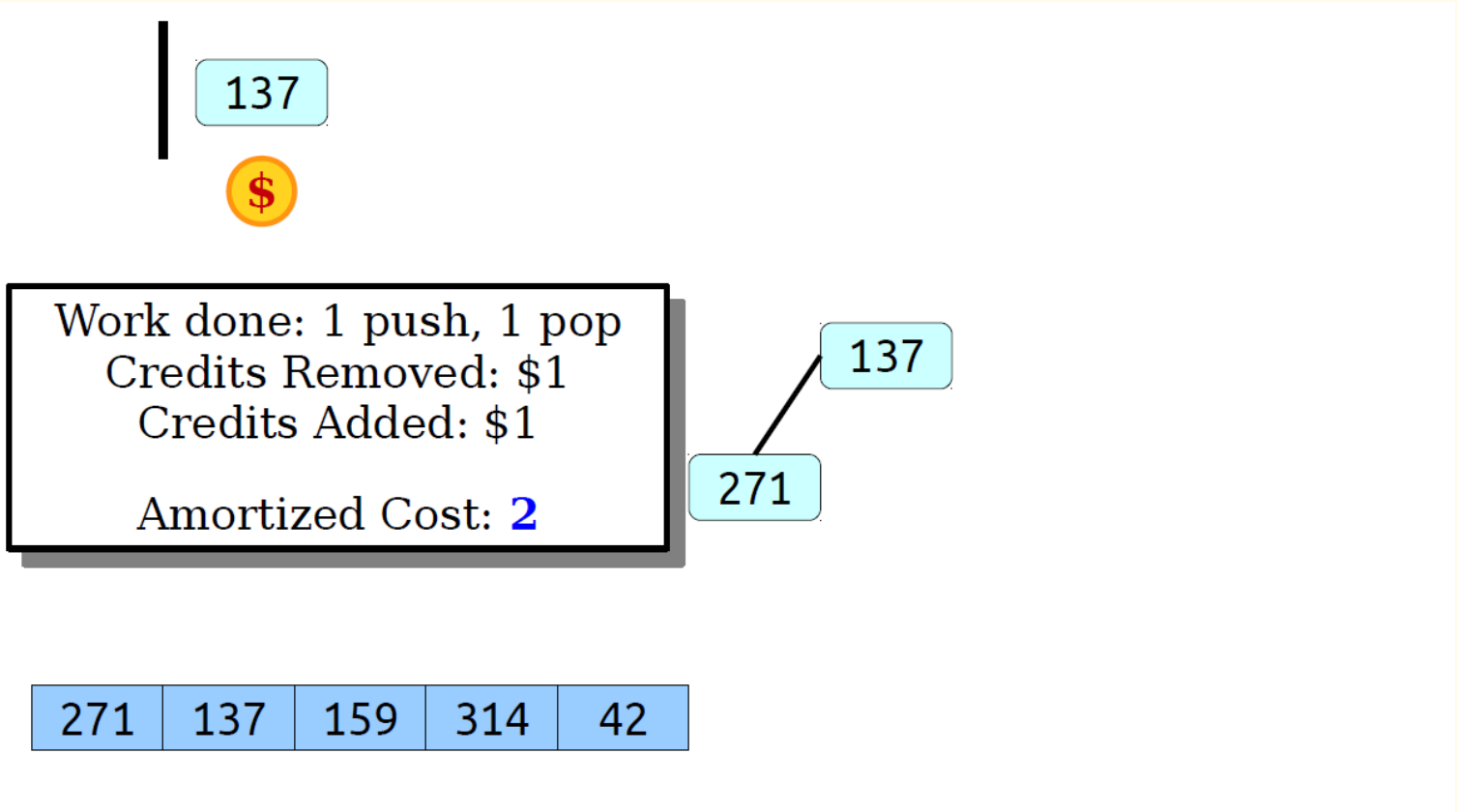
# Accounting Method

Define the credit as number of nodes from current node $v_i$ to root



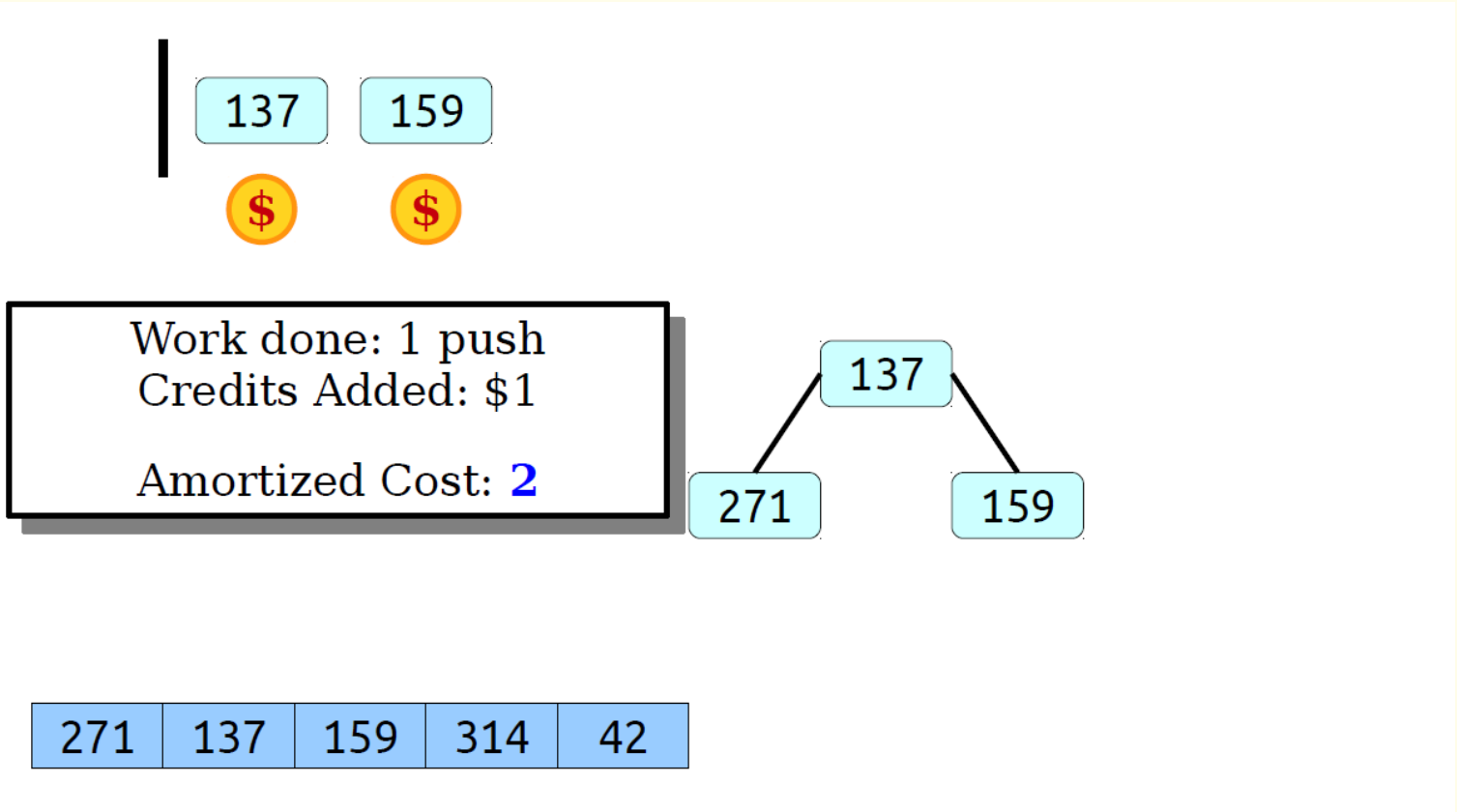
# Accounting Method

Define the credit as number of nodes from current node $v_i$ to root



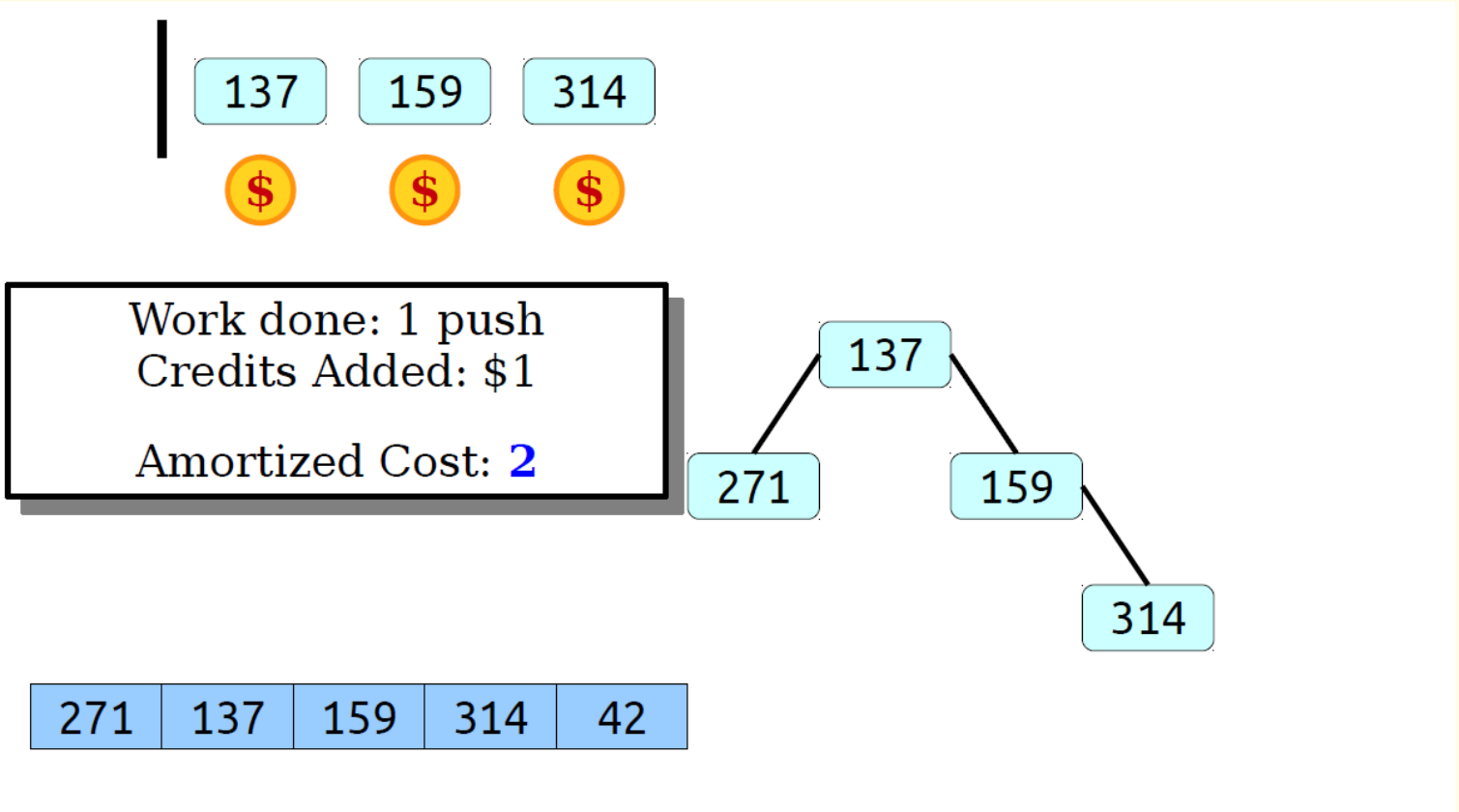
# Accounting Method

Define the credit as number of nodes from current node $v_i$ to root



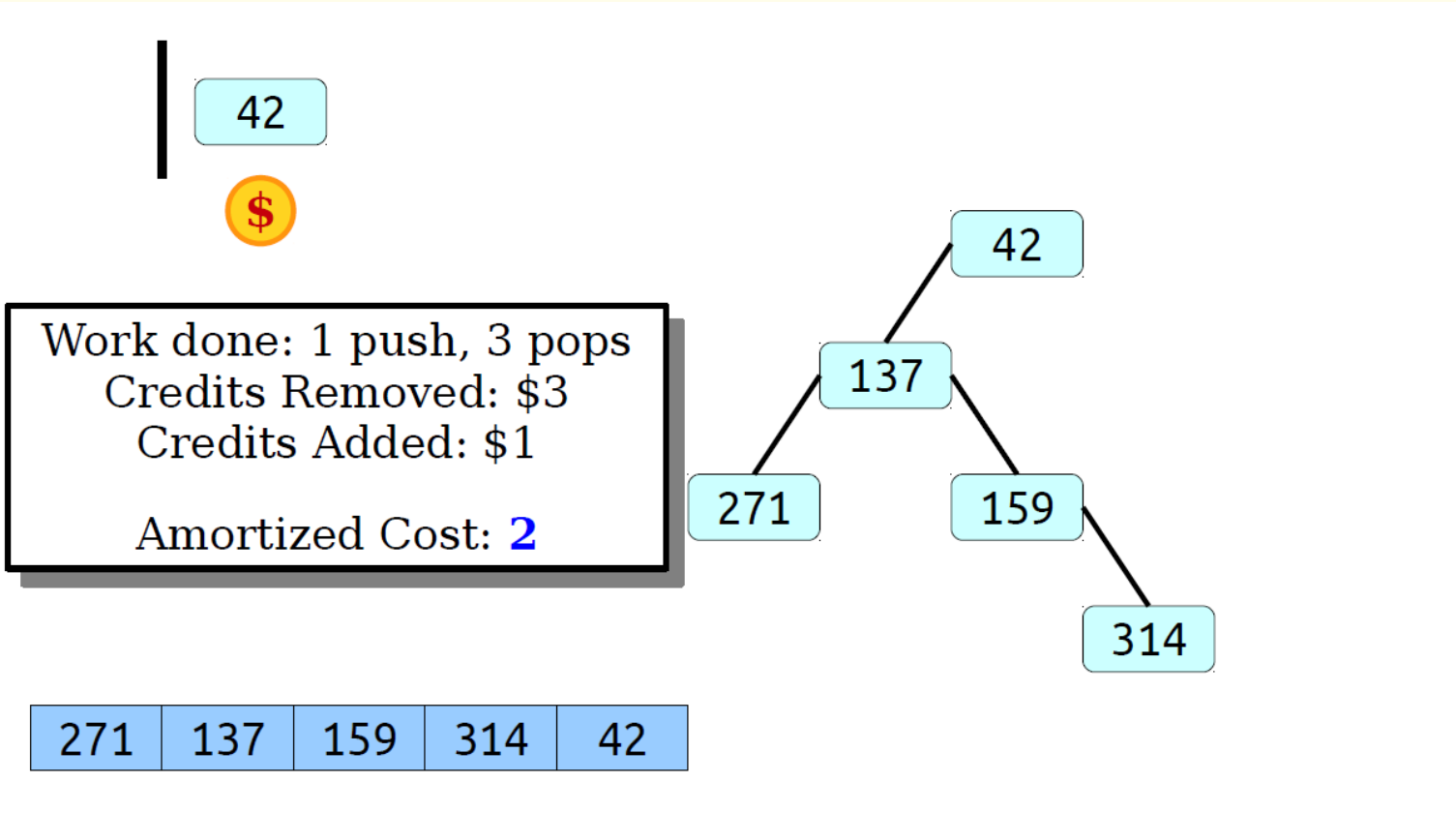
# Accounting Method

Define the credit as number of nodes from current node $v_i$ to root



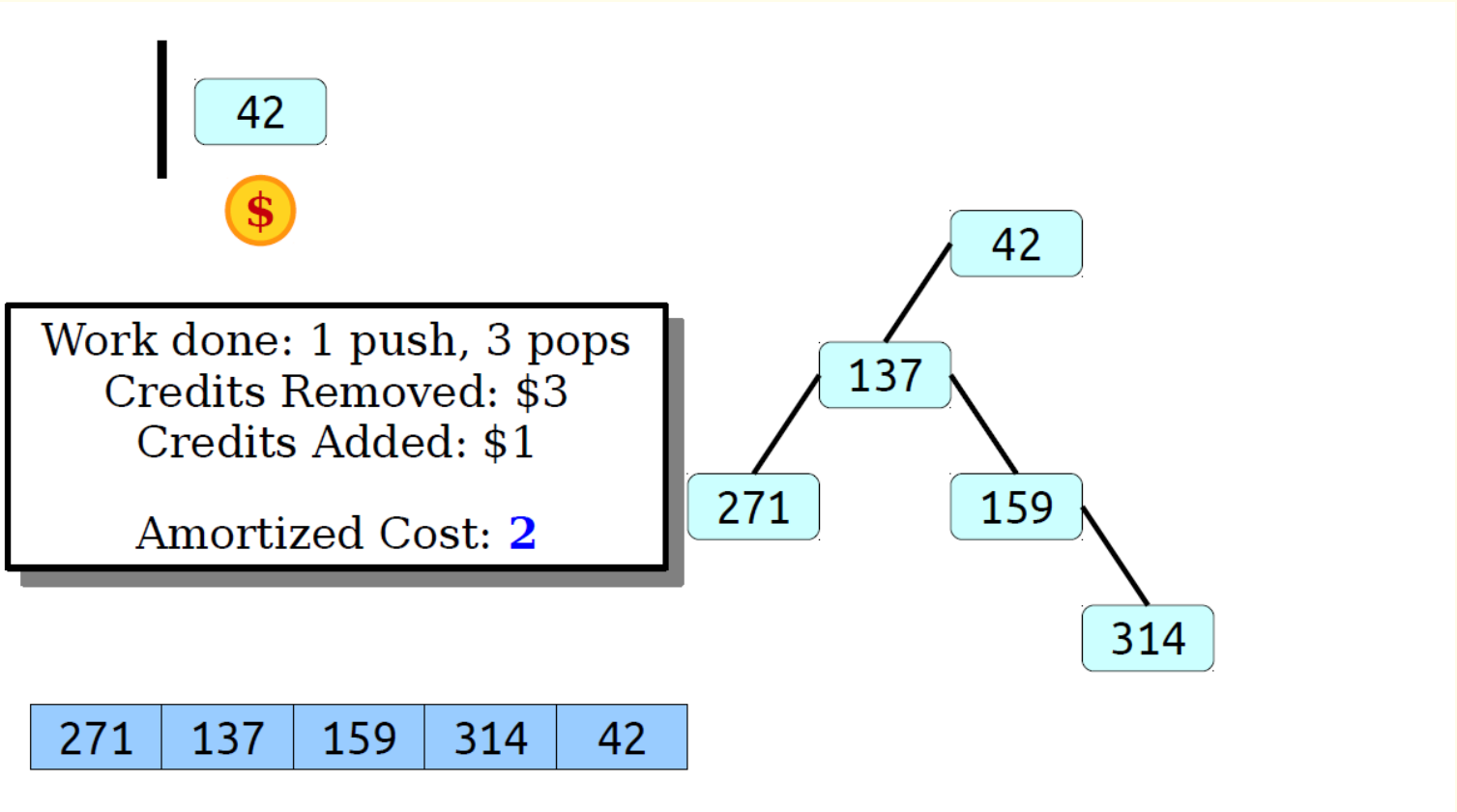
# Accounting Method

Define the credit as number of nodes from current node $v_i$ to root



# Accounting Method

Define the credit as number of nodes from current node $v_i$ to root




Similarly, define the potential function $\Phi(D)$ as the length.

# Questions?