Data Structures 1

NTIN066

Jirka Fink

Department of Theoretical Computer Science and Mathematical Logic
Faculty of Mathematics and Physics
Charles University in Prague

Summer semester 2025/26
Last change on February 24, 2026

Amortized complexity

Motivation

- Consider a data structure which is usually very fast
- However in rare cases, it needs to reorganize its internal structure
- So, the worst-case complexity is quite slow
- This data structure may be used by an algorithm
- We are interested in the total complexity or average complexity of many operations

Dynamic array

Problem description (Stack version)

- We need to store n elements in an array of length p
- How do we implement operations Push and Pop to ensure that $p = \Theta(n)$?

Problem description (Stack version)

- We need to store n elements in an array of length p
- How do we implement operations Push and Pop to ensure that $p = \Theta(n)$?
- If $n = p$ and an element has to be inserted, then the length of the array is doubled
- If $4n = p$ and an element has to be deleted, then the length of the array is halved
- How many elements are copied during k operations Push and Pop?

Problem description (Stack version)

- We need to store n elements in an array of length p
- How do we implement operations Push and Pop to ensure that $p = \Theta(n)$?
- If $n = p$ and an element has to be inserted, then the length of the array is doubled
- If $4n = p$ and an element has to be deleted, then the length of the array is halved
- How many elements are copied during k operations Push and Pop?

Aggregated analysis

- Let k_i be the number of operations between $(i-1)$-th and i-th reallocation

Problem description (Stack version)

- We need to store n elements in an array of length p
- How do we implement operations Push and Pop to ensure that $p = \Theta(n)$?
- If $n = p$ and an element has to be inserted, then the length of the array is doubled
- If $4n = p$ and an element has to be deleted, then the length of the array is halved
- How many elements are copied during k operations Push and Pop?

Aggregated analysis

- Let k_i be the number of operations between $(i-1)$-th and i-th reallocation
- The first reallocation copies at most $n_0 + k_1$ elements where n_0 is the initial number of elements
- The i-th reallocation copies at most $2k_i$ elements for $i \geq 2$
- Every operation without reallocation copies at most 1 element
- The total number of copied elements is at most $k + (n_0 + k_1) + \sum_{i \geq 2} 2k_i \leq n_0 + 3k$

Dynamic array

Potential method

Potential method

- Consider the potential

$$\Phi = \begin{cases} 0 & \text{if } p = 2n \\ n & \text{if } p = n \\ n & \text{if } p = 4n \end{cases}$$

and piece-wise linear function in other cases

Potential method

- Consider the potential

$$\Phi = \begin{cases} 0 & \text{if } p = 2n \\ n & \text{if } p = n \\ n & \text{if } p = 4n \end{cases}$$

  and piece-wise linear function in other cases
- Explicitly,

$$\Phi = \begin{cases} 2n - p & \text{if } p \leq 2n \\ p/2 - n & \text{if } p \geq 2n \end{cases}$$

Potential method

- Consider the potential

$$\Phi = \begin{cases} 0 & \text{if } p = 2n \\ n & \text{if } p = n \\ n & \text{if } p = 4n \end{cases}$$

  and piece-wise linear function in other cases
- Explicitly,

$$\Phi = \begin{cases} 2n - p & \text{if } p \leq 2n \\ p/2 - n & \text{if } p \geq 2n \end{cases}$$

- Change of the potential without reallocation is $\Phi_i - \Phi_{i-1} \leq 2$

Dynamic array

Potential method

- Consider the potential

$$\Phi = \begin{cases} 0 & \text{if } p = 2n \\ n & \text{if } p = n \\ n & \text{if } p = 4n \end{cases}$$

  and piece-wise linear function in other cases
- Explicitly,

$$\Phi = \begin{cases} 2n - p & \text{if } p \leq 2n \\ p/2 - n & \text{if } p \geq 2n \end{cases}$$

- Change of the potential without reallocation is $\Phi_i - \Phi_{i-1} \leq 2$
- Let T_i be the number of elements copied during i-th operation
- Observe that $T_i + \Phi_i - \Phi_{i-1} \leq 3$
- The total number of copied elements during k operations is $\sum_{i=1}^{k} T_i \leq 3k + \Phi_0 - \Phi_k \leq 3k + n_0$

Amortized complexity

Description

- The amortized complexity of an operation is the total time of k operations over k assuming that k is sufficiently large.
- For example, the amortized complexity of operations Push and Pop in the dynamic array is $\frac{\sum_{i=1}^{k} T_i}{k} \leq \frac{3k+n_0}{k} \leq 4 = \mathcal{O}(1)$ assuming that $k \geq n_0$.

Amortized complexity

Description

- The amortized complexity of an operation is the total time of k operations over k assuming that k is sufficiently large.
- For example, the amortized complexity of operations Push and Pop in the dynamic array is $\frac{\sum_{i=1}^{k} T_i}{k} \leq \frac{3k+n_0}{k} \leq 4 = \mathcal{O}(1)$ assuming that $k \geq n_0$.

Potential method

- Let Φ a non-negative potential which evaluates the internal representation of a data structure
- Let T_i be the actual time complexity of i-th operation
- Let Φ_i be the potential after i-th operation
- The amortized complexity of the operation is $\mathcal{O}(f(n))$ if $T_i + \Phi_i - \Phi_{i-1} \leq f(n)$ for every operation i in an arbitrary sequence of operations
- For dynamic array, $T_i + \Phi_i - \Phi_{i-1} \leq 3$, so the amortized complexity of operations Push and Pop is $\mathcal{O}(1)$

Can a potential be negative?

Can a potential be negative?

Naive dynamic array

- Every insertion increases the size of the array by one

Can a potential be negative?

Naive dynamic array

- Every insertion increases the size of the array by one
- Define a potential $\Phi(n) = -\binom{n}{2}$
- It holds that $T \leq 1 + \Phi(n) - \Phi(n+1)$
- From the choice $A = 1$ we obtain constant time amortized complexity

Therefore, the potential has to be non-negative to calculate amortized complexity.

Can a potential be negative?

Naive dynamic array

- Every insertion increases the size of the array by one
- Define a potential $\Phi(n) = -\binom{n}{2}$
- It holds that $T \leq 1 + \Phi(n) - \Phi(n+1)$
- From the choice $A = 1$ we obtain constant time amortized complexity

Therefore, the potential has to be non-negative to calculate amortized complexity.

Total time complexity of k consecutive operations

- The array in empty in the beginning, and it contains k elements in the end
- Total time of k operations is

$$k \cdot A + \phi(0) - \phi(k) = k - 0 + \binom{k}{2} = \binom{k+1}{2} = \mathcal{O}\left(k^2\right)$$

Can a potential be negative?

Naive dynamic array

- Every insertion increases the size of the array by one
- Define a potential $\Phi(n) = -\binom{n}{2}$
- It holds that $T \leq 1 + \Phi(n) - \Phi(n+1)$
- From the choice $A = 1$ we obtain constant time amortized complexity

Therefore, the potential has to be non-negative to calculate amortized complexity.

Total time complexity of k consecutive operations

- The array in empty in the beginning, and it contains k elements in the end
- Total time of k operations is

$$k \cdot A + \phi(0) - \phi(k) = k - 0 + \binom{k}{2} = \binom{k+1}{2} = \mathcal{O}\left(k^2\right)$$

The decrease of potential is accounted in the total time, so it can be negative.

Why dynamic array is doubling the size?

Consider that an array is full, and it has n elements.
What happens when we change its size to be

- $3n$
- $n + 10$
- n^2?

Queue

- Propose an efficient implementation of a queue.

Queue

- Propose an efficient implementation of a queue.
- You have two stacks, supporting only the POP and PUSH operations. Propose an algorithm, that would simulate a Queue with operations ENQUEUE and DEQUEUE. Besides the two stacks, you have only a constant amount of memory. Show that the queue operations have a constant amortized time complexity.

Queue

- Propose an efficient implementation of a queue.
- You have two stacks, supporting only the POP and PUSH operations. Propose an algorithm, that would simulate a Queue with operations ENQUEUE and DEQUEUE. Besides the two stacks, you have only a constant amount of memory. Show that the queue operations have a constant amortized time complexity.

```
class Queue:
  def __init__(self):
    self.input = [] # For elements that are enqueued
    self.output = [] # For elements that will be dequeued

  def enqueue(self, element):
    self.input.append(element)

  def dequeue(self):
    if not self.output: # If the output list is empty
      while self.input: # While the input list is not empty
        self.output.append(self.input.pop())
    return self.output.pop()
```

More details on Wikipedia

Dequeue

Propose an efficient implementation of a dequeue; e.i. an array that allows inserting, and removing elements from both ends.

Binary search tree

Definition: Binary tree

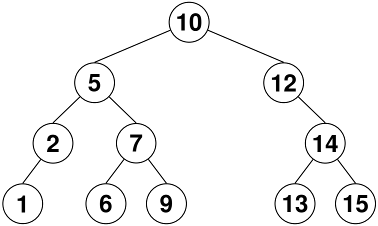
Binary tree is a tree data structure in which each node has at most two children, referred to as the left child and the right child. Each node except the root has a parent. A node without any child is called leaf. Every node contains some data of one element and every element is stored in one node. Every element is identified by a unique key and these keys are comparable.

Definition: Binary tree

Binary tree is a tree data structure in which each node has at most two children, referred to as the left child and the right child. Each node except the root has a parent. A node without any child is called leaf. Every node contains some data of one element and every element is stored in one node. Every element is identified by a unique key and these keys are comparable.

Definition: Binary search tree (BST)

Binary search tree is a binary tree data structure with the key of each internal node being greater than all the keys in the respective node's left subtree and less than the ones in its right subtree.

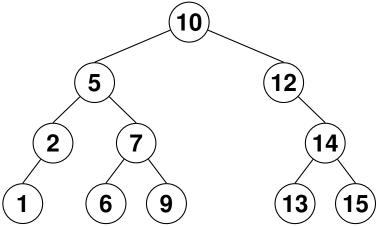


Basic operations in BST

Find a node containing a given key. Returns an invalid value if BST has no element with that key.

Insert a given element with its key. Returns False if the key is duplicate.

Delete the node containing a given key. Returns False if BST has no element with that key.



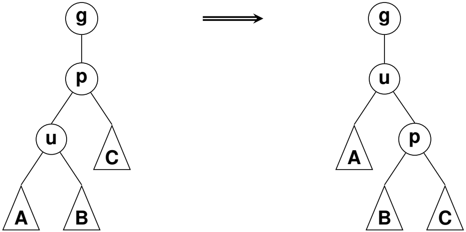
Task

Describe these operations and determine their complexity.

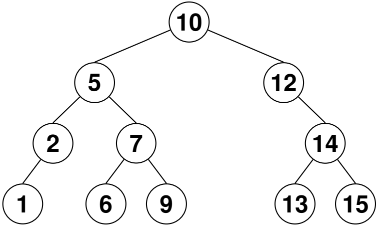
- Successor of a node A is the smallest node larger than A
- Given a tree and its node, find the successor
- If the given node is the last one, return None
- If the given node is None, return the first node
- When whole tree is traversed by your program, the total time complexity has to be $O(n)$

Single rotation in BST

Given a node u different from the root, the single rotation of u with its parent p changes BST so that u becomes a parent of p. Rotating a node u to the root means applying the single rotation on u until it becomes the root of BST.



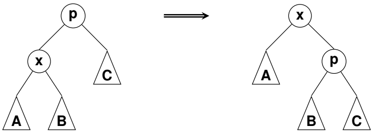
Exercise: Rotate the node 6 to the root.



Properties of single rotations

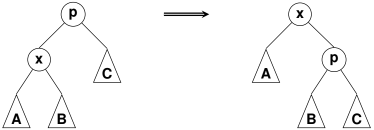
1. Consider an arbitrary BST containing elements with keys $1, \dots, n$ and apply rotation to the root on all elements in increasing order. Describe the resulting BST.
2. Rotate all elements to the root once more. Estimate the number of single rotations needed in this procedure.

- Zig rotation: The parent p of x is the root

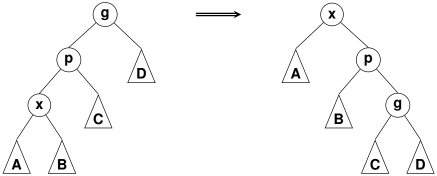


Splay tree (Sleator, Tarjan, 1985): Operation SPLAY of an node *x*

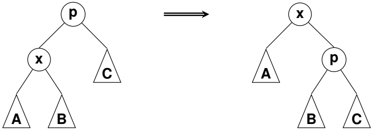
- Zig rotation: The parent *p* of *x* is the root



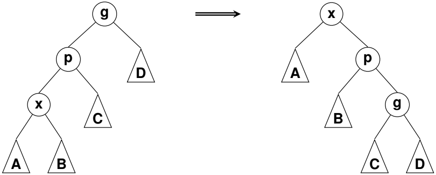
- Zig-zig rotation: Both *x* and *p* are left children of their parents



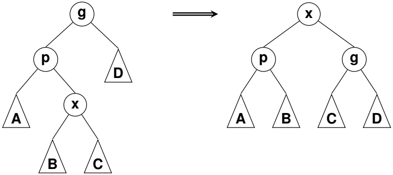
- Zig rotation: The parent *p* of *x* is the root



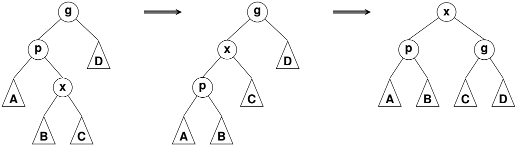
- Zig-zig rotation: Both *x* and *p* are left children of their parents



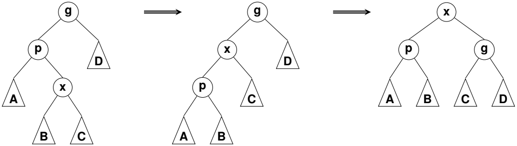
- Zig-zag rotation: Node *x* is the right child and *p* is the left child



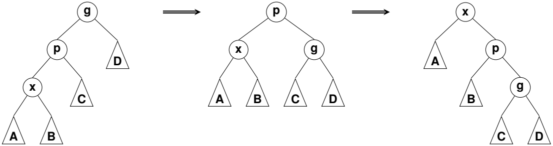
- Zig-zag are two single rotations of x:



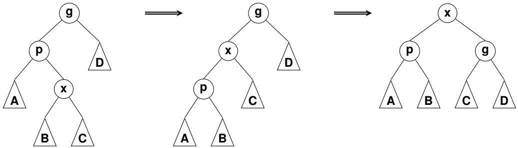
- Zig-zag are two single rotations of *x*:



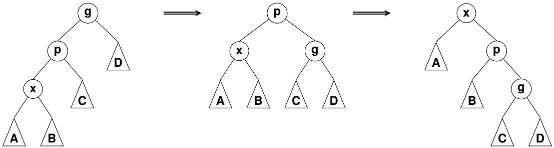
- Zig-zig can also be implemented as two rotations:



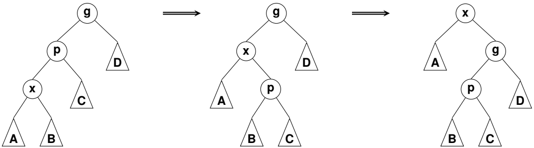
- Zig-zag are two single rotations of *x*:



- Zig-zig can also be implemented as two rotations:



- However, two rotations of the element *x* lead to an incorrect result:



Properties of SPLAY operation

1. Consider an arbitrary BST containing elements with keys $1, \dots, n$ and apply SPLAY operation on all elements in increasing order. Describe the resulting tree.
2. Then, apply the SPLAY operation on element 1 again. Estimate the height of the resulting tree.

Operations is SPLAY trees

The amortized complexity of operations Find, Insert and Delete in Splay tree is $O(\log n)$. This requires calling Splay on deepest visited node in every case. Give an example showing that this splaying is needed to achieve the desired complexity also in the following cases.

1. Splaying the last visited node in the operation Find when a given key is not present.
   - Describe Splay tree containing n nodes and a sequence of n unsuccessful operations Find with total complexity $\Theta(n^2)$.

Operations is SPLAY trees

The amortized complexity of operations Find, Insert and Delete in Splay tree is $O(\log n)$. This requires calling Splay on deepest visited node in every case. Give an example showing that this splaying is needed to achieve the desired complexity also in the following cases.

1. Splaying the last visited node in the operation Find when a given key is not present.
   - Describe Splay tree containing n nodes and a sequence of n unsuccessful operations Find with total complexity $\Theta(n^2)$.
2. Splaying the inserted element.
   - Show that inserting n elements into an empty Splay tree without splaying may have complexity $\Theta(n^2)$.
3. Splaying the original element when inserting a duplicate key.
   - Describe Splay tree containing n nodes and a sequence of n unsuccessful operations Insert with total complexity $\Theta(n^2)$.
4. Splaying the last visited element when a given key to be deleted is not present.
5. When operation Delete requires replacement, show that it is not sufficient to Splay the node which contained the deleted key before replacement.

2nd assignment: Implementation of Splay tree

Implement operations Splay, Insert, and Delete so that their amortized complexity is $O(\log n)$.