

Algorithms and Data Structures

B2. Abstract Data Types: Stacks & Queues

Gabriele Röger

University of Basel

April 3, 2024

Abstract Data Type

Abstract Data Type

Abstract Data Type

Description of a data type, summarizing the possible data and the possible operations on this data.

- **User perspective:** How can I use the data type?
- In contrast to data structures, not specifying the concrete representation of the data.

Advantages of Abstract Data Types

- User codes against an interface.
- The underlying data structure (representation) is hidden/encapsulated.
 - Representation can be replaced at any time.
- Separating two aspects:
 - ① What is the data type doing (interface)?
 - ② How is this achieved (internal structure)?

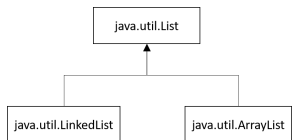
Advantages of Abstract Data Types

- User codes against an interface.
- The underlying data structure (representation) is hidden/encapsulated.
 - Representation can be replaced at any time.
- Separating two aspects:
 - ① What is the data type doing (interface)?
 - ② How is this achieved (internal structure)?

We can abstract away the dirty details and stay more flexible.

Abstract Data Types and Classes

- In object-oriented languages, abstract data types are often implemented as interfaces.
- For example, lists in Java:



```
interface List<E>:
    E get(int index);
    void add(E element);
    void add(int pos, E element);
    ...
```

Today: Stacks and Queues



Stack (of plates)



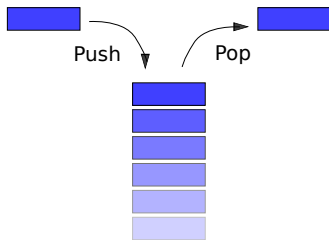
Queue (of persons)

Stack

Stack

A **stack** is a data structure following the **last-in-first-out (LIFO)** principle supporting the following operations:

- **push**: puts an item on top of the stack
- **pop**: removes the item at the top of the stack

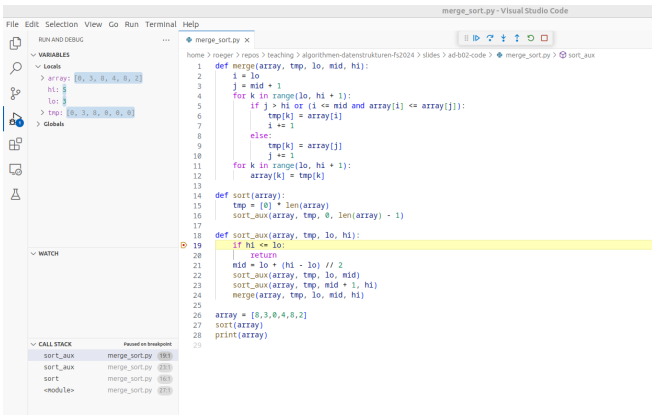


Both operations should take constant time.

Application: Call Stack

The **call stack** stores information when running subroutines of a computer program.

→ where to resume once the subroutine has terminated



The screenshot shows the Visual Studio Code interface with a Python file named `merge_sort.py` open. The code implements a merge sort algorithm. The call stack on the left shows the current execution state, with `sort_aux` at line 19 of `merge_sort.py` highlighted. The code is as follows:

```
1 def merge(array, tmp, lo, mid, hi):
2     i = lo
3     j = mid + 1
4     for k in range(lo, hi + 1):
5         if j > hi or (i <= mid and array[i] <= array[j]):
6             tmp[k] = array[i]
7             i += 1
8         else:
9             tmp[k] = array[j]
10            j += 1
11    for k in range(lo, hi + 1):
12        array[k] = tmp[k]
13
14 def sort(array):
15     tmp = [0] * len(array)
16     sort_aux(array, tmp, 0, len(array) - 1)
17
18 def sort_aux(array, tmp, lo, hi):
19     if hi <= lo:
20         return
21     mid = lo + (hi - lo) // 2
22     sort_aux(array, tmp, lo, mid)
23     sort_aux(array, tmp, mid + 1, hi)
24     merge(array, tmp, lo, mid, hi)
25
26 array = [0,3,0,4,8,2]
27 sort(array)
28 print(array)
29
```

Jupyter Notebook



Jupyter notebook: `fundamental-adts.ipynb`

Stack: Possible Implementation with Linked Lists

```
class Stack:
    def __init__(self):
        self.list = LinkedList()

    def push(self, item):
        self.list.prepend(item)

    def pop(self):
        if self.list.is_empty():
            raise Exception("popping from empty stack")
        else:
            return self.list.remove_first()
```

Questions



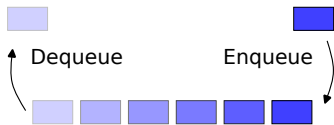
Questions?

Queue

Queue

A **queue** is a data structure following the **first-in-first-out (FIFO)** principle supporting the following operations:

- **enqueue**: adds an item to the tail of the queue
- **dequeue**: removes the item at the head of the queue



Both operations should take constant time.

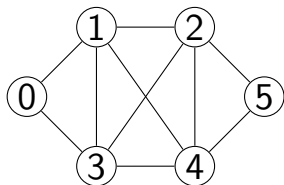
Application: Breadth-first Exploration

Queues are always helpful if we need to store elements and process them in the same order.

Application: Breadth-first Exploration

Queues are always helpful if we need to store elements and process them in the same order.

With a breadth-first exploration, we want to visit all reachable nodes in a graph in the order of their distance from a given start node.



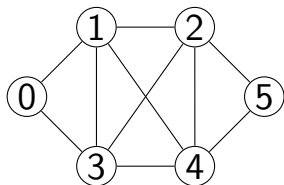
Starting from node 5, any of the following visitation orders would be fine:

- 5 2 4 1 3 0
- 5 4 2 1 3 0
- 5 2 4 3 1 0
- 5 4 2 3 1 0

Application: Breadth-first Exploration

Queues are always helpful if we need to store elements and process them in the same order.

With a breadth-first exploration, we want to visit all reachable nodes in a graph in the order of their distance from a given start node.



Starting from node 5, any of the following visitation orders would be fine:

- 5 2 4 1 3 0
- 5 4 2 1 3 0
- 5 2 4 3 1 0
- 5 4 2 3 1 0

Implementation with queue in Jupyter notebook

Jupyter Notebook



Jupyter notebook: `fundamental-adts.ipynb`

Queue: Possible Implementation with Doubly Linked Lists

```
class Queue:
    def __init__(self):
        self.list = DoublyLinkedList()

    def enqueue(self, item):
        self.list.append(item)

    def dequeue(self):
        if self.list.is_empty():
            raise Exception("dequeuing from empty queue")
        else:
            return self.list.remove_first()
```

Questions



Questions?

Deque

Dequeues

A **double-ended queue** (deque) generalizes both, queues and stacks:

- **append**: adds an item to the right side of the deque.
- **appendleft**: adds an item to the left side of the deque.
- **pop**: removes the item at the right end of the deque.
- **popleft**: removes the item at the left end of the deque.

Operation names can differ between programming languages.

All operations should take constant time.

Dequeues

A **double-ended queue** (deque) generalizes both, queues and stacks:

- **append**: adds an item to the right side of the deque.
- **appendleft**: adds an item to the left side of the deque.
- **pop**: removes the item at the right end of the deque.
- **popleft**: removes the item at the left end of the deque.

Operation names can differ between programming languages.

All operations should take constant time.

How would you implement a deque?

Summary

Summary

- **Abstract data types** (ADTs) specify the **behavior** of a data type, not the internal representation.

Summary

- **Abstract data types** (ADTs) specify the **behavior** of a data type, not the internal representation.
- **Stack**: follows last-in-first-out (LIFO) principle.
- **Queue**: follows first-in-first-out (FIFO) principle.
- **Deque**: generalizes stack and queue.

Summary

- **Abstract data types** (ADTs) specify the **behavior** of a data type, not the internal representation.
- **Stack**: follows last-in-first-out (LIFO) principle.
- **Queue**: follows first-in-first-out (FIFO) principle.
- **Deque**: generalizes stack and queue.
- All: in principle just lists with limited functionality.
- Limitations help clarifying intended usage and avoiding mistakes.

Summary

- **Abstract data types** (ADTs) specify the **behavior** of a data type, not the internal representation.
- **Stack**: follows last-in-first-out (LIFO) principle.
- **Queue**: follows first-in-first-out (FIFO) principle.
- **Deque**: generalizes stack and queue.
- All: in principle just lists with limited functionality.
- Limitations help clarifying intended usage and avoiding mistakes.

→ Preferably code against an ADT/interface.