Algorithms and Data Structures

B2. Abstract Data Types: Stacks & Queues

Gabriele Röger and Patrick Schnider

University of Basel

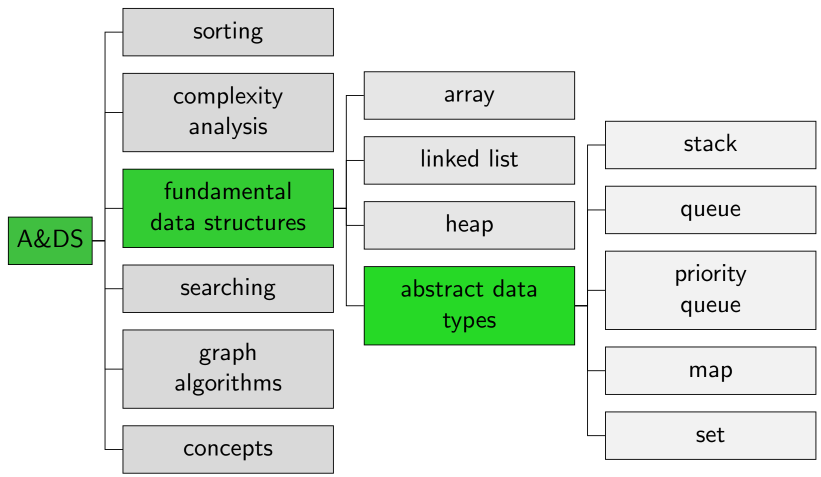
March 27, 2025

Algorithms and Data Structures

March 27, 2025 — B2. Abstract Data Types: Stacks & Queues

B2.1 Abstract Data Type

Content of the Course



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.

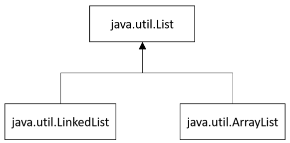
Advantages of Abstract Data Types

- 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)?

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);
  ...
```

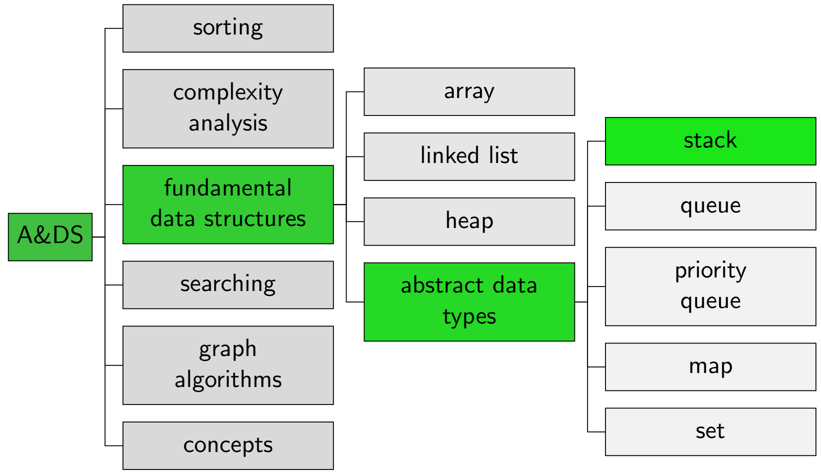

Today: Stacks and Queues

Stack (of plates)

Queue (of persons)

B2.2 Stack

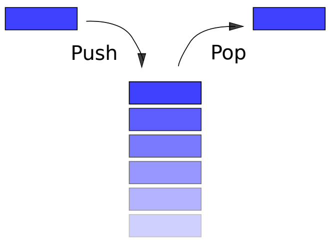
Content of the Course



Stack

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

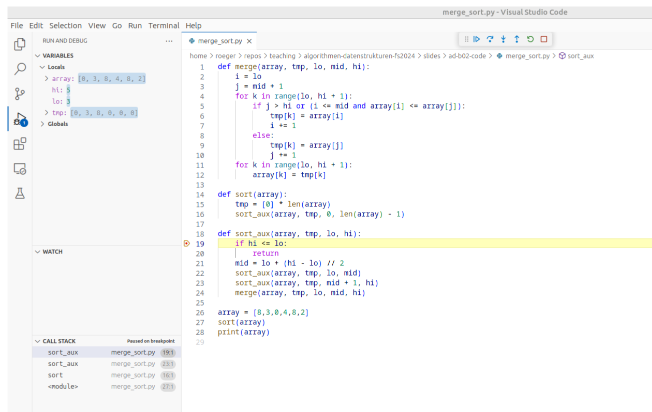
- push: puts an item on top of the stack
- pop: removes the item at the top of the stack



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



Jupyter Notebook

Jupyter notebook: `fundamental-adts.ipynb`

Stack: Possible Implementation with Doubly Linked Lists

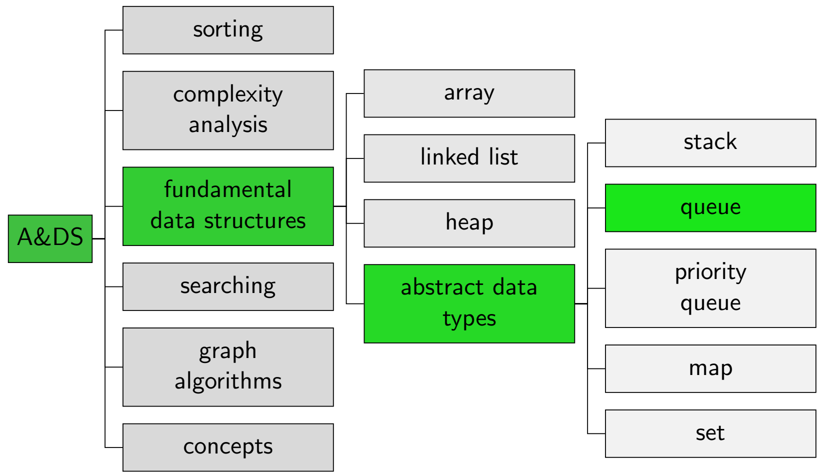
```
class Stack:
    def __init__(self):
        self.list = DoublyLinkedList()

    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()
```


B2.3 Queue

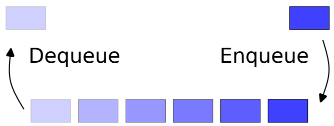
Content of the Course



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

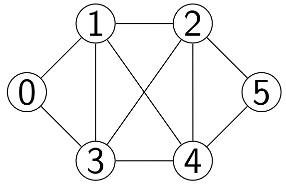


Both operations should take constant time.

Application: Breadth-first Exploration

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

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()
```

B2.4 Deque

Deques

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?

B2.5 Summary

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.