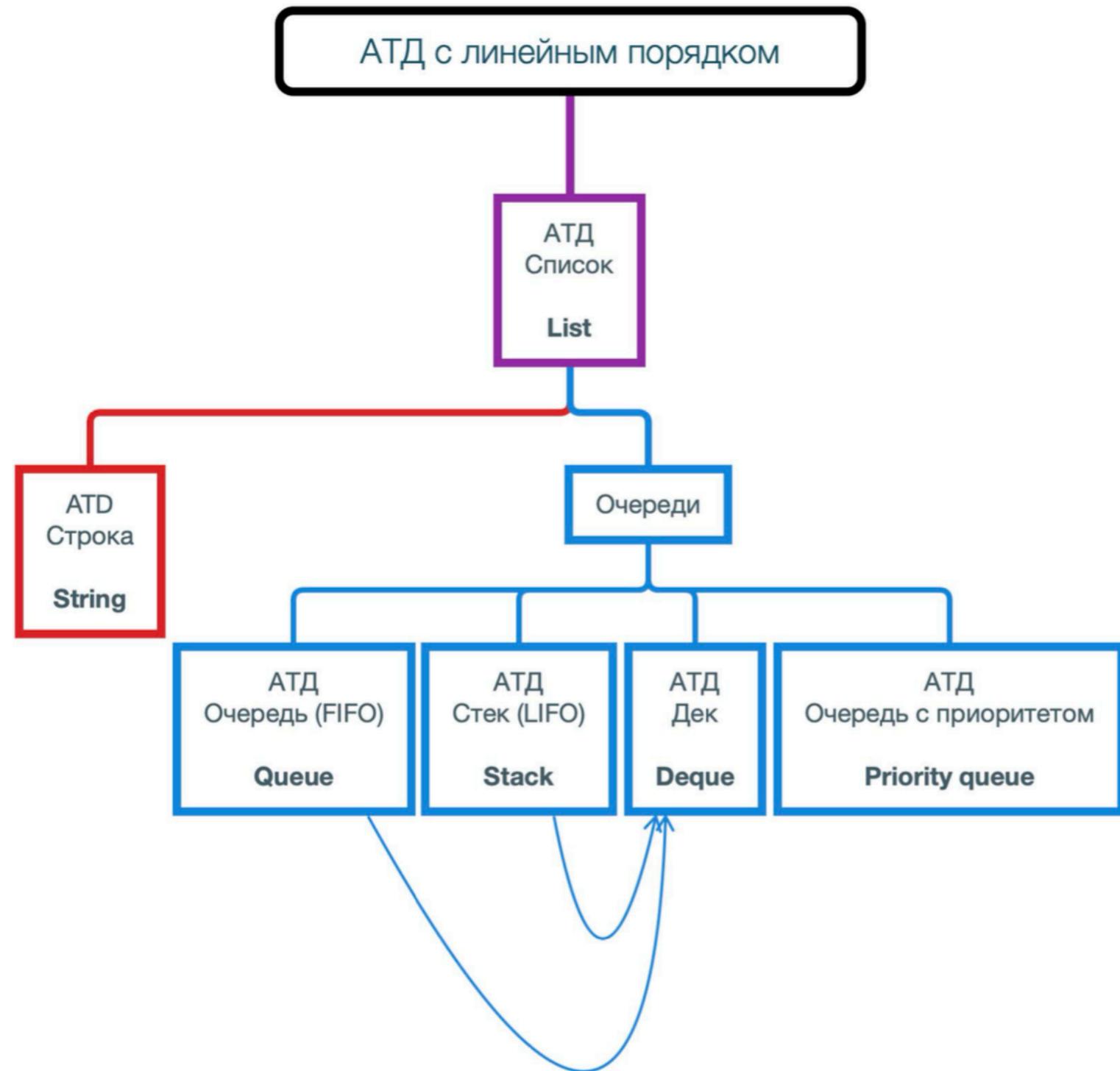


Информатика

Осень 2024

Баранов Максим Александрович

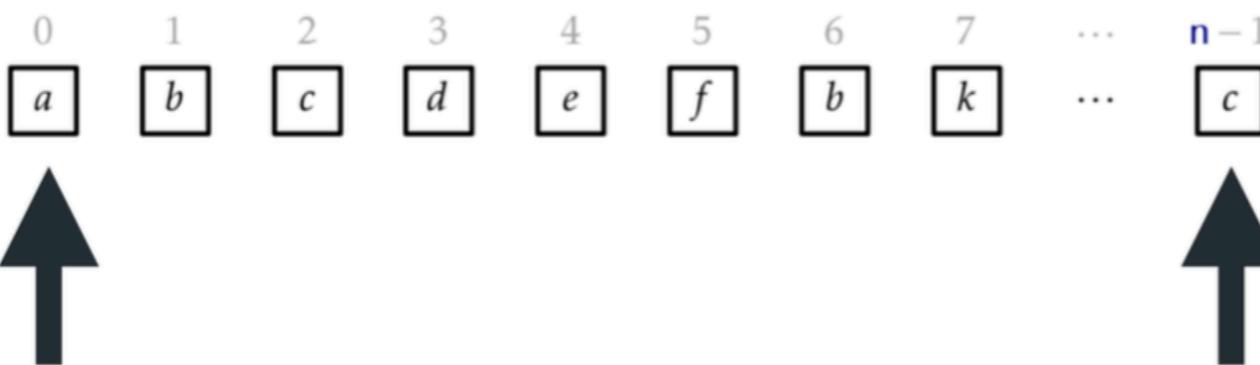
АТД с линейным порядком



Список

Список (list)

Строка (string)



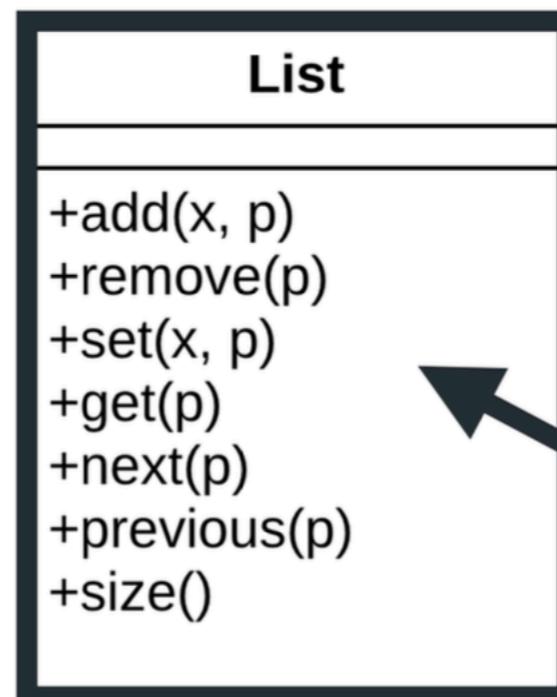
голова (head)

хвост (tail)

Список

Список (list)

Строка (string)



Могут быть
другие имена

Бывают и другие операции

Связный список

Это структура данных, в которой элементы (узлы) хранятся в линейной последовательности, но не занимают смежные участки памяти. Каждый узел содержит данные и ссылку на следующий элемент в списке.

- **Односвязный список:** Каждый узел содержит данные и ссылку на следующий узел.
- **Двусвязный список:** Каждый узел имеет ссылку на следующий и предыдущий узлы, что позволяет двигаться по списку в обе стороны.
- **Кольцевой связный список:** Конец списка связан с его началом, образуя замкнутый круг. Может быть как односвязным, так и двусвязным.

Преимущества и недостатки связных списков

Преимущества:

- Эффективное добавление и удаление элементов, особенно в начале списка.
- Легкое изменение размера, так как элементы не требуют смежной памяти.

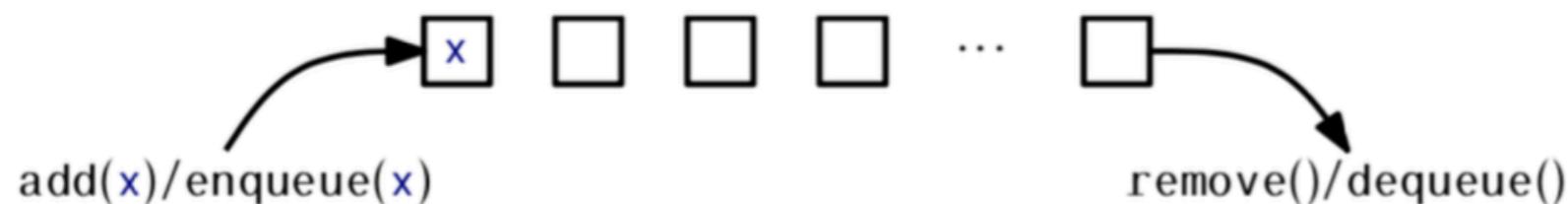
Недостатки:

- Отсутствие доступа по индексу (требуется обход от начала списка).
- Дополнительные затраты на память для хранения ссылок.
-

Очередь

Очередь (queue)

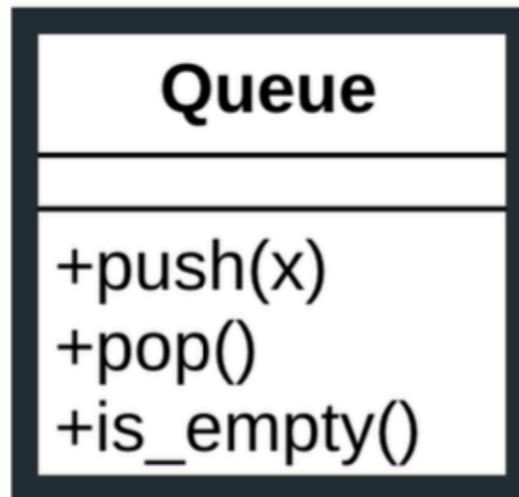
разновидность очереди: FIFO



Очередь

Очередь (queue)

разновидность очереди: FIFO



FIFO и списки

```
clients = list()

def check_in(client):
    clients.append(client)
    print(f"in: New client {client} joined the queue.")

def connect_to_associate(associate):
    if clients:
        client_to_connect = clients.pop(0)
        print(f"out: Remove client {client_to_connect}, connecting to {associate}.")
    else:
        print("No more clients are waiting.")
```

```
check_in("John")
check_in("Sam")
connect_to_associate("Emily")
check_in("Danny")
connect_to_associate("Zoe")
connect_to_associate("Jack")
connect_to_associate("Aaron")

# print out the following lines:
in: New client John joined the queue.
in: New client Sam joined the queue.
out: Remove client John, connecting to Emily.
in: New client Danny joined the queue.
out: Remove client Sam, connecting to Zoe.
out: Remove client Danny, connecting to Jack.
No more clients are waiting.
```

Очередь FIFO (First In, First Out)

```
from collections import deque

# Создание очереди FIFO
queue = deque()

# Добавление элементов в конец очереди
queue.append("Клиент 1")
queue.append("Клиент 2")
queue.append("Клиент 3")

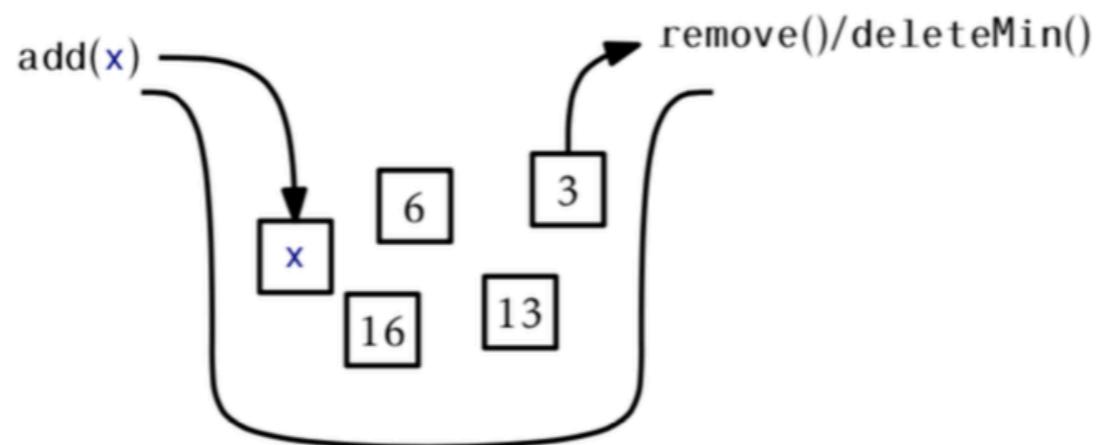
print("Очередь:", queue)

# Удаление элемента из начала очереди
first_client = queue.popleft()
print("Обслужен:", first_client)
print("Оставшаяся очередь:", queue)
```

Очередь с приоритетом

Очередь с приоритетом (priority queue)

разновидность очереди



Очередь с приоритетом

Очередь с приоритетом (priority queue)

разновидность очереди

| Priority queue |
|----------------|
| +push(x, c) |
| +pop_max() |
| +is_empty() |

Очередь с приоритетом

```
import heapq

# Создание очереди с приоритетом
priority_queue = []

# Добавление элементов с приоритетом
heapq.heappush(priority_queue, (3, "Задача 3")) # Приоритет 3
heapq.heappush(priority_queue, (1, "Задача 1")) # Приоритет 1 (высший приоритет)
heapq.heappush(priority_queue, (2, "Задача 2")) # Приоритет 2

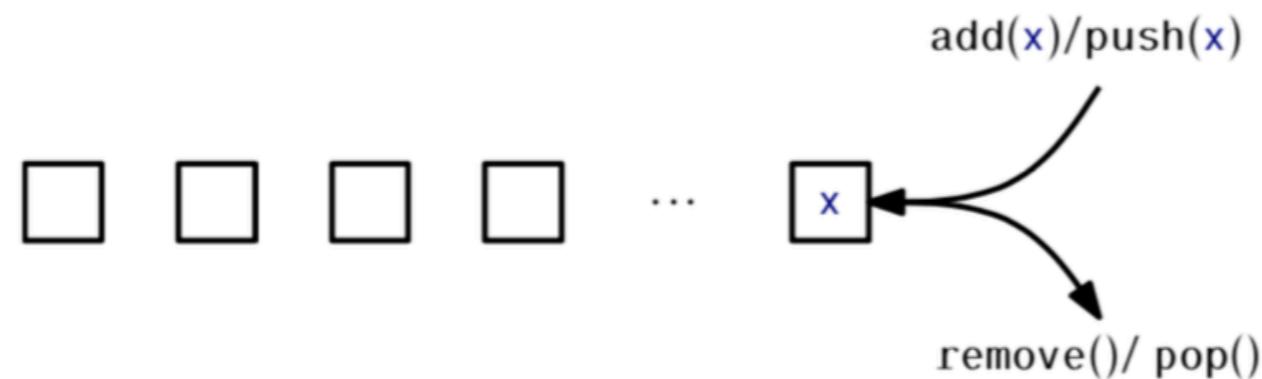
print("Очередь с приоритетом:", priority_queue)

# Извлечение элемента с наивысшим приоритетом
highest_priority_task = heapq.heappop(priority_queue)
print("Выполняется:", highest_priority_task)
print("Оставшиеся задачи:", priority_queue)
```

Стек

Стек (stack)

разновидность очереди: LIFO

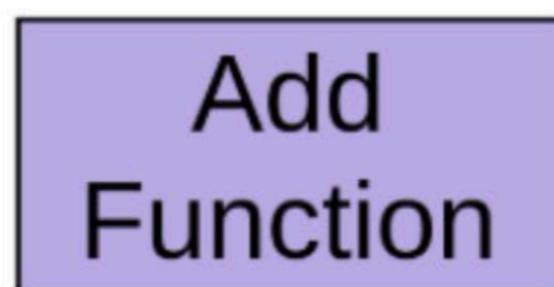


Before:



Undo
Stack

Push:

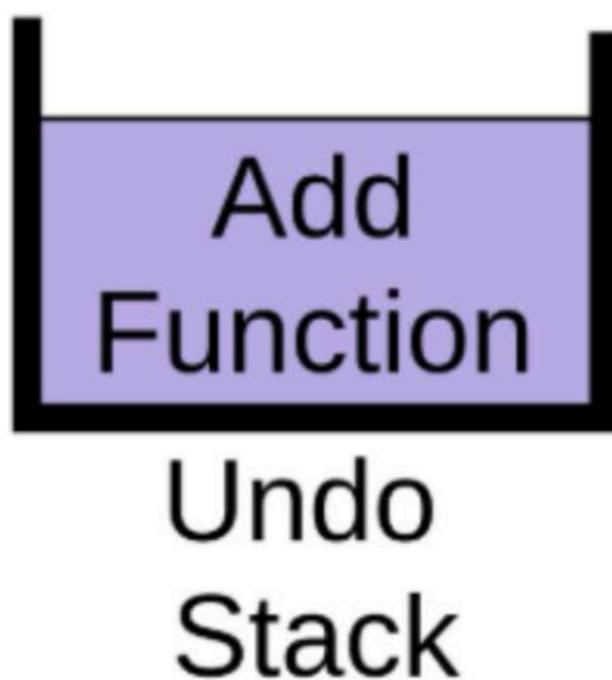


After:

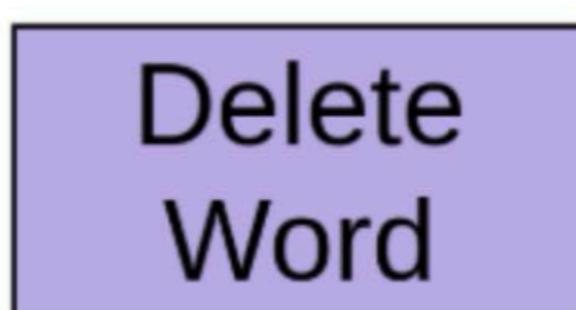


Undo
Stack

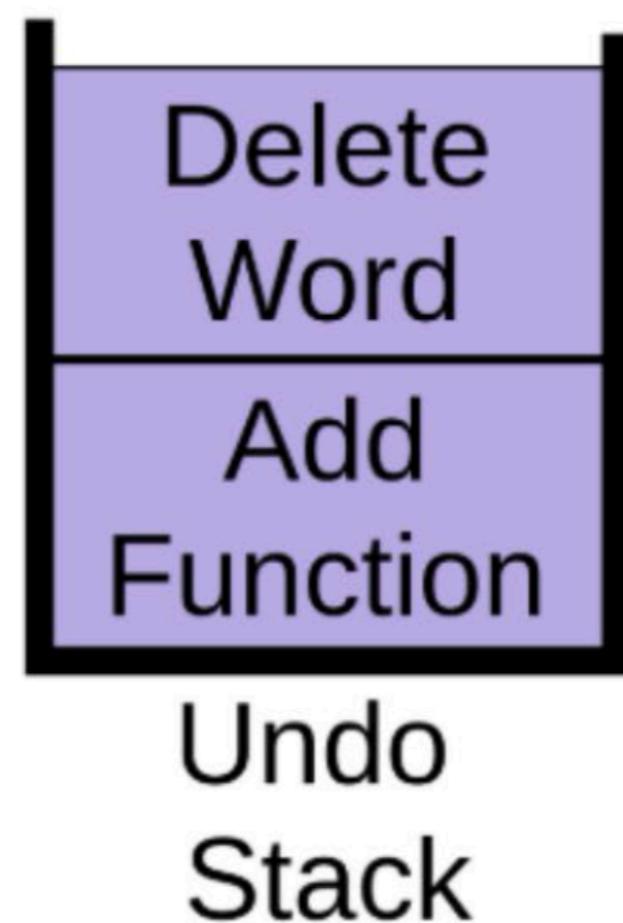
Before:



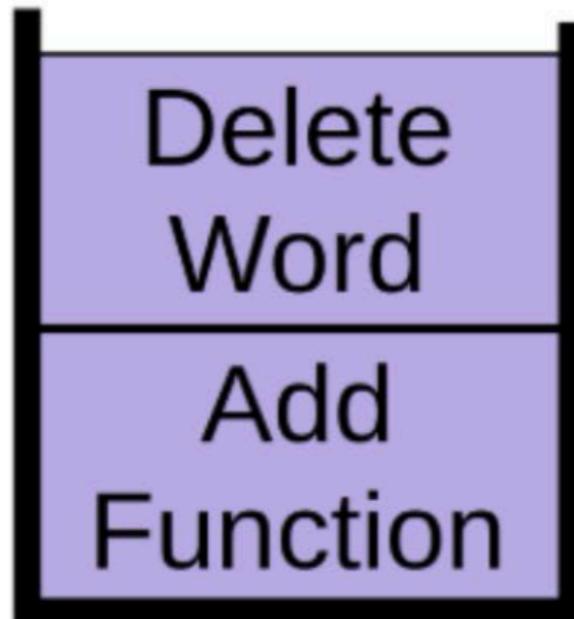
Push:



After:



Before:



Push:

Indent
Comment

After:

Indent
Comment

Delete
Word

Add
Function

Undo
Stack

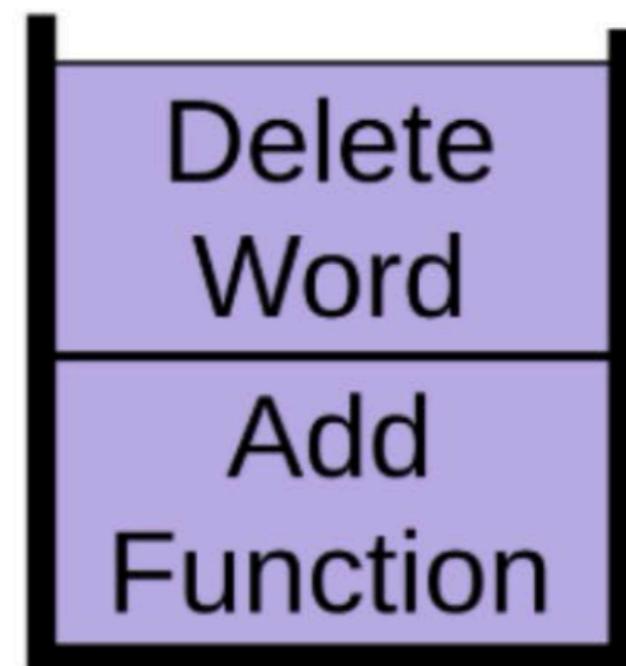
Before:



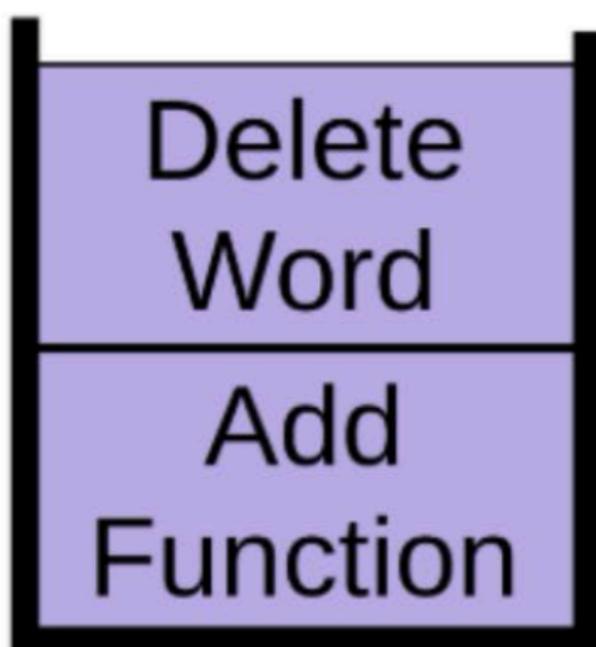
Pop:



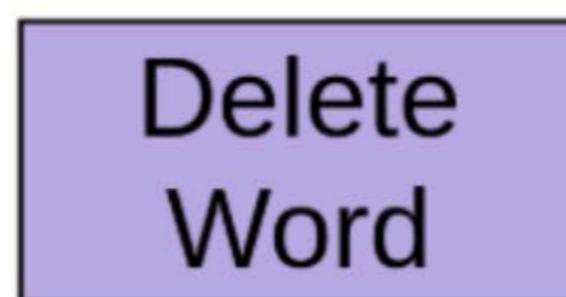
After:



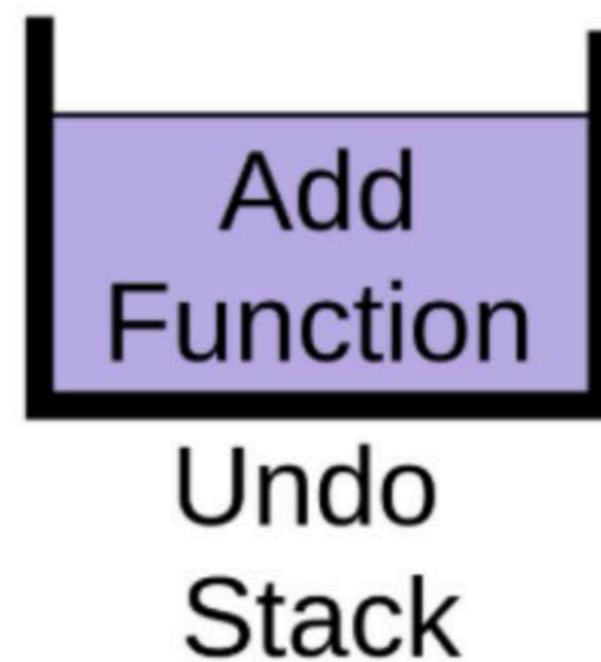
Before:



Pop:



After:



Очередь LIFO (Last In, First Out) или Стек

```
stack = []

# Добавление элементов в стек
stack.append("Действие 1")
stack.append("Действие 2")
stack.append("Действие 3")

print("Стек:", stack)

# Удаление последнего добавленного элемента (вершина стека)
last_action = stack.pop()
print("Отменено:", last_action)
print("Оставшиеся действия:", stack)
```

Стек с помощью deque

```
from collections import deque

# Создание стека с использованием deque
stack = deque()

# Добавление элементов в стек
stack.append("Действие 1")
stack.append("Действие 2")
stack.append("Действие 3")

print("Стек:", stack)

# Удаление элемента с вершины стека
last_action = stack.pop()
print("Отменено:", last_action)
print("Оставшиеся действия:", stack)
```

Дек

Дек (dequeue)

разновидность очереди: FIFO + LIFO



Дек

Дек (deque)

разновидность очереди: FIFO + LIFO

Deque

- +push_first(x)
- +push_last(x)
- +pop_first()
- +pop_last()
- +is_empty()

FIFO и LIFO

```
from collections import deque

# Создаем двухстороннюю очередь
dq = deque()

# Пример работы в режиме FIFO
# Добавление элементов
dq.append("Элемент 1")
dq.append("Элемент 2")
dq.append("Элемент 3")

print("Очередь FIFO:", dq)

# Удаление элемента с начала очереди (FIFO)
first_in = dq.popleft()
print("Обслужен (FIFO):", first_in)
```

FIFO и LIFO

```
print("Оставшиеся элементы (FIFO):", dq)

# Пример работы в режиме LIFO
# Добавление элементов (снова добавим для примера LIFO)
dq.append("Элемент 4")
dq.append("Элемент 5")

print("\nСтек LIFO:", dq)

# Удаление элемента с конца очереди (LIFO)
last_in = dq.pop()
print("Отменено (LIFO):", last_in)
print("Оставшиеся элементы (LIFO):", dq)
```

Пример реализации

```
from collections import deque
from timeit import timeit

def time_fifo_testing(n):
    integer_l = list(range(n))
    integer_d = deque(range(n))
    t_l = timeit(lambda: integer_l.pop(0), number=n)
    t_d = timeit(lambda: integer_d.popleft(), number=n)
    return f"{n: <9} list: {t_l:.6e} | deque: {t_d:.6e}"
```

```
numbers = (100, 1000, 10000, 100000)
for number in numbers:
    print(time_fifo_testing(number))
```

| | | | |
|--------|--------------------|--|---------------------|
| 100 | list: 2.186300e-05 | | deque: 1.461700e-05 |
| 1000 | list: 2.353830e-04 | | deque: 1.465280e-04 |
| 10000 | list: 1.561046e-02 | | deque: 1.386711e-03 |
| 100000 | list: 1.741322e+00 | | deque: 1.344412e-02 |

Временная сложность

Разным структурам данных присуща разная временная сложность операций

- В списке Python удаление первого элемента занимает $O(n)$ из-за сдвига оставшихся элементов, тогда как удаление в deque занимает $O(1)$.
- Операции вставки и удаления в двухсторонних очередях deque (при работе с концами) выполняются за $O(1)$, что делает их быстрее списков для ряда задач.

Применение АТД в Реальных Программах

АТД имеют широкое применение в программировании, и понимание этих областей поможет студентам увидеть ценность этих структур:

- **Стек:** Используется в анализе выражений, управлении вызовами функций, для хранения состояния при рекурсивных вызовах.
- **Очередь:** Входит в основу многопоточных систем, где задачи обрабатываются последовательно. Например, обработка сообщений в очередях брокеров сообщений (RabbitMQ, Kafka).
- **Очередь с приоритетом:** Планирование задач в операционных системах и алгоритмы поиска кратчайшего пути (алгоритм Дейкстры).
- **Deque:** Реализует функционал, как у очередей, так и у стеков. Подходит для задач, где элементы могут быть добавлены или удалены с любого конца.

Особенности Хранения в Памяти

- **Последовательными** структурами (например, массивы) — все элементы хранятся в непрерывных участках памяти, что ускоряет доступ по индексу, но замедляет вставку и удаление.
- **Связанными** структурами (например, двусвязный список, основа deque) — элементы могут быть распределены по разным участкам памяти и связаны указателями, что делает вставку и удаление быстрыми, но замедляет доступ к произвольному элементу.

Вопросы Стабильности и Потокобезопасности

- **Стабильность** очередей важна в задачах, где порядок добавления влияет на результат. Очередь должна возвращать элементы в том порядке, в котором они были добавлены.
- **Потокобезопасность** – важный аспект для многопоточных приложений. Например, deque в Python является потокобезопасным для операций append и pop, но для полностью безопасной работы в потоках предпочтительнее использовать queue.LifoQueue или queue.PriorityQueue.

Особенности Реализации АТД в Python

Python предоставляет несколько встроенных инструментов для работы с АТД:

- **list** – универсальная структура, но не оптимизированная для работы в качестве очереди из-за особенностей внутренней памяти.
- **collections.deque** – оптимизирован для операций вставки и удаления с обоих концов.
- **queue.Queue** и **queue.LifoQueue** – потокобезопасные очереди, подходящие для работы в многопоточном окружении.