Lab 5 – Stacks & Queues

Instructors: Lorenzo De Carli & Maan Khedr

Slides by Lorenzo De Carli

What we are going to focus on today

- Understanding implementation and performance of stacks and queues

Exercise #1: stack-based parser[1.5 pts]

- We are going to use a stack to build a primitive parser for *arithmetic S-expressions*. Such an expression consists of either:
  - An atom x, consisting of an integer number
  - An expression of the form *(o e1 e2)*, where *o* is an operand and *e1*, *e2* are S-expressions
  - Possible operands are +, -, *, /
- Examples:
  - `1`
  - `(+15)`
  - `(*(+15)2)`
  - `(-(*13)(/4(+12)))`

Exercise #1 /2

- Your code must:
  1. Receive a string representing an expression as a command line parameter [0.3 pts]
  2. Implement a stack data structure as discussed in lab and class [0.4 pts]
  3. Using the stack, compute the overall result of an expression [0.8 pts]
- For example:

```
python ex1.py '(+ 1 5)'
6

python ex1.py '(* (+ 1 5) 2)'
12

python ex1.py '(- (* 1 3) (/ 6 (+ 1 2)))'
1
```

Exercise 1 - What to deliver

- Submit a file named ex1.py with your implementation of the stack-based parser (questions 1, 2 and 3)

Exercise 2: simple priority queues [1.5 pts]

- We have seen that a priority queue always guarantees that the smallest (or largest) value is at the queue's head

1. Implement a priority queue class based on Python arrays (ref. to the discussion in the lecture on queues) [0.3 pts]
   - The priority queue must implement enqueue by appending at the end of the array, and then immediately sorting the array using mergesort, and dequeue by removing the first element
2. Implement another priority queue class [0.3 pts]
   - The priority queue must implement enqueue by inserting the new element in the appropriate location to ensure the array remains sorted at all times, and dequeue by removing the first element

Exercise #2 /2

3. Write a function which generates random lists of 1000 tasks. Each task is either an enqueue w/ probability 0.7, or a dequeue w/ probability 0.3 [0.3 pts]
4. Measure the performance of both implementations on 100 such lists using `timeit` and print the results [0.3 pts]
5. Discuss the results: which implementation is faster? Why? [0.3 pts]

- **Note:** your implementation should support dequeue from empty queue, returning None in that case

Exercise 2 - What to deliver

- Submit a file named ex2.py with answers to questions 1, 2, 3 and 4
- The file should also contain the answer to question 5 as code comments

Exercise 3 – stack performance [1.5 pts]

- In this exercise you will analyze the performance of different stack implementations

1. Implement a stack which internally uses Python arrays. push() must append an element at the tail, and pop() must remove an element from the tail [0.3 pts]
2. Implement a stack which internally uses a singly-linked list. push() must add an element at the head, and pop() must remove the head element [0.3 pts]

Exercise #3 /2

3. Write a function which generates random lists of 10000 tasks. Each task is either a push w/ probability 0.7, or a pop w/ probability 0.3 [0.3 pts]
4. Measure the performance of both implementations on 100 such lists of tasks using `timeit` and print the results [0.3 pts]
5. Plot the distribution of times (distributions for each implementation should be overlayed in the same plot; make sure to use consistent ranges) and discuss the results [0.3 pts]

Exercise 3 - What to deliver

- Submit a file named ex3.py with answers to questions 1, 2, 3, 4 and 5

Exercise 4 – queue performance [1.5 pts]

- In this exercise you will analyze the performance of different queue implementations

1. Implement a queue which internally uses Python arrays. enqueue() must insert an element at the head, and dequeue() must remove an element from the tail [0.3 pts]
2. Implement a queue which internally uses a singly-linked list. enqueue() must add an element at the head, and dequeue() must remove the tail element (make sure to keep a tail pointer!) [0.3 pts]

Exercise #4 /2

3. Write a function which generates random lists of 10000 tasks. Each task is either an enqueue w/ probability 0.7, or a dequeue w/ probability 0.3 [0.3 pts]
4. Measure the performance of both implementations on 100 such lists of tasks using `timeit` and print the results [0.3 pts]
5. Plot the distribution of times (distributions for each implementation should be overlayed in the same plot; make sure to use consistent ranges) and discuss the results [0.3 pts]

Exercise 4 - What to deliver

- Submit a file named ex4.py with answers to questions 1, 2, 3, 4 and 5

Exercise #5 – Running around [1 pt]

- In this exercise, we are going to implement a circular queue. A circular queue is a queue that uses a fixed-size array, and the front of the queue wraps around to the back of the array when it reaches the end.
- Requirement 1: The circular queue should provide the following methods:
  - enqueue adds an item to the queue and prints “enqueue <element>” to terminal
  - dequeue removes and returns the front-most item in the queue and prints “dequeue <element>” to terminal
  - peek just returns the front-most item in the queue without removing it and prints “peek <element>” to terminal
- Requirement 2: The queue should print “dequeue None” or “peek None” if you try to dequeue or peek from an empty queue.
- Requirement 4: The queue should print “enqueue None” if you try to to enqueue into a full queue

Exercise #5 /2

1. Implement such a queue based on a fixed-size Python array [0.4 pts]
2. Implement the queue again, this time using a circular linked list [0.4 pts]
3. Generate a list of 40 operations, together with expected output, that can be used to test correctness of implementation. [0.2 pts]
   - The list must include multiple peek, enqueue, dequeue operations. It should test regular operations and all corner cases (enqueue into full queue, peek into empty queue, push into empty queue)
   - The list can be implemented directly as Python code, e.g.:

```
Mylist.enqueue(1)
Mylist.dequeue()
…
```

Exercise 5 - What to deliver

- Submit a file named ex5.py with answers to questions 1, 2, and 3

How to submit

- Upload a zip file to the “Lab 5” dropbox on D2L, containing the required content for every exercise

Grading rubric

- You get **3 pts** for uploading a **partial solution** by **end of lab**
  - **Must not be an empty file or irrelevant material**
- Then, you'll have until **11:59PM of the day before the next lab** to upload the **complete solution**. That will be graded as follows:
  - Exercise 1-4: 1.5 pts each
  - Exercise 5: 1 pt
  - **Can upload the complete solution to the same dropbox**

That’s all folks!