CSCM712 – Software concepts and efficiency March 6th 2026
Cécilia Pradic & Deshan Sumanathilaka

Lab 6: linked lists and dynamic arrays

This week, there are three assessed tasks. Tasks 6.1 and 6.3 are about linked lists, and Task 6.2 is about dynamic arrays.

1. **Task 6.1: Linked lists** Open the file `llist.py` available on canvas **under the lab heading**. It is an expansion of the file we have started writing during the lecture that implements singly linked-lists in python. You will notice it contains more stuff, including:

   - placeholders that you should modify as part of your answer
   - some utility and diagnostic methods and functions
   - a blurb at the end that you may modify at will to test your functions.

   So first take a global readthrough. You should understand what is going on with the `cell` class and the first few three methods defined in `llist`, which is fairly close to what we did in the lectures. Then take a glance at the comments above the diagnostic functions `healthy` and `toDot`. Then take a look at the code beneath the last top-level comment, run the program and check that you can generate the picture that corresponds to the call to `toDot`.

   That was a long set-up, but hopefully that will be helpful! If you suspect you have a bug when dealing with the questions below, please try to test your functions using the diagnostic functions and figure out what is wrong.

   (a) Implement the method `prepend` that adds an element at the front of a linked list.
   (b) Implement the method `insert`.
   (c) Implement the method `concat`, making sure it runs in $\mathcal{O}(1)$.

2. **Task 6.2: Double-ended dynamic arrays** Recall that dynamic arrays exist, and that's how python lists are implemented. Recall that appending an element at the end of a dynamic array can be done in *amortized* constant time: the worst case complexity is linear, but over the lifetime of a dynamic array, we can pretend that `append` is constant time and get the correct overall complexity.

   Sadly, inserting a new element at the front of a python list using `insert` always runs in linear time.

   Now imagine that you are an employee in some "registered charity" and your boss Paul asked your colleague Rima to implement a datastructure that is able to have

- `prepend` in amortized constant time,
- `append` in amortized constant time,
- writing and reading an arbitrary cell of any index in constant time

Rima started working on this, and then was made redundant in the name of financial sustainability. She produced the file `deDynArr.py` that you may find on canvas.

(a) Read through the file and execute it. Describe in your own words how Rima tackled the problem.

(b) In the class `deDynArr`, implement the methods

```
def prepend(self, i):
def __getitem__(self, i):
def __setitem__(self, i, x):
```

as Paul intended. The last two methods allow python to overload the notation `a[i]` and `a[i] = x` when `a` is an object of a custom class. Paul wants your implementation to be somewhat compatible with Python's default convention for lists[1]:

- valid indexes for `a` are integers that can range from `- len(a)` to `len(a) - 1`. Negative indexes allow one to query things from the end of the collection.
- if an index outside of this range is given as input, your function should raise an `IndexError` exception.
- Paul does not want you to waste company time on implementing the cases where `i` is a `slice` object or check it is indeed an integer.

3. **Task 6.3 Merge sort** After our charity adventure, let us come back to linked lists. Our goal now will be to implement a memory-efficient sorting function `mergeSort`, an **in place** algorithm. This means that we want to only use up a constant[2] amount of memory on top of what `xs` occupies during the execution of `xs.mergeSort()`.

(a) First look at the code of the methods `isEmpty` and `peekFront`. Implement the method `popFront`.

(b) Now implement the method `split`

> **Tip.** *If you solution is starting to go over 15 lines, you are probably over-complicating it. Take advantage of the other methods in the class.*

[1] See `https://docs.python.org/3/reference/datamodel.html#object.__getitem__`

[2] In practice it's not quite true; if you follow the path of least resistance, the memory footprint will be *(poly)logarithmic* for reasons I'll be better equipped to explain next week. If you are nitpicky about counting (more nitpicky than we teach you to be in this module!) I doubt you can do much better anyway.

(c) There is a subtle bug in the function `merge` that can be fixed by a very small modification. Can you spot it?

> **Tip.** *If you don't see it after looking at the code, try using `healthy` and `toDot`. If you struggle too much, you can also skip this subquestion for now and get back to it later - `mergeSort` is going to have a minor bug but should be able to mostly run well.*

(d) Implement the method `mergeSort`.

> **Tip.** *Again, your solution should ideally not be longer than 15 lines.*

4. **Bonus tasks (not assessed)** Some questions to ponder if you like.

   (a) Write an *in place* `reverse` method for linked lists.

   (b) Consult `https://docs.python.org/3/glossary.html#term-iterator` and write more code in `llist.py` so that the following program should run and make sense:

```
x = llist(range(0,10))
for i in x:
  print(i)
```

   (c) For Task 3.2, Paul emails you to have a meeting. After settling on a meeting time, Paul explains to you over zoom that he wants you to modify the datastructure so that it has a `reverse` method running in constant time on top of all the other requirements. How do you plan on tackling that?

   (d) The `pop` method in python removes the last element of a list in amortized constant time. Can you implement a constant time method for linked list? Implement a *doubly-linked list* class that has both `pop` and `popFront` running in constant time.