CSCM12: software concepts and efficiency
Datastructures for ordered collections

Cécilia Pradic

March 2nd 2026

Algorithms and datastructures

Starting from today we are going to talk about datastructures or

How stuff is layed out in memory matters

Algorithms and datastructures

Starting from today we are going to talk about datastructures or

How stuff is layed out in memory matters

We will still discuss algorithms and efficiency

- Introducing datastructures → tools to
  - program efficiently **and/or**
  - representations for input/outputs for algorithmic problems

Today

- Some high-level considerations (not too long)
- Lists in python: dynamic arrays
- Another kind of lists: linked lists

Wot's a datastructure?

Informal concept, high-level

Rough reductionist definition

1. A chunk of memory space layed out in a specified way
   (the attributes an object of a class)
2. A bunch of **operations** (the methods)

Wot's a datastructure?

Informal concept, high-level

Rough reductionist definition

1. A chunk of memory space layed out in a specified way
   (the attributes an object of a class)
2. A bunch of **operations** (the methods)

In OOP, both aspects *often* materialize as

1. the attribute of a class and its objects
2. the methods of the class

Wot's a datastructure?

Informal concept, high-level

Rough reductionist definition

1. A chunk of memory space layed out in a specified way
   (the attributes an object of a class)
2. A bunch of **operations** (the methods)

In OOP, both aspects *often* materialize as

1. the attribute of a class and its objects
2. the methods of the class

Purpose?

Language-independent designation for a useful reusable abstraction

Examples: arrays (`int[]`), dynamic arrays (python lists, `ArrayList`), strings, sets

Interface and comparing datastructures

What is a good datastructure?

- Depends on the application/purpose
- Point of comparison: the operations

Interface and comparing datastructures

What is a good datastructure?

- Depends on the application/purpose
- Point of comparison: the operations

Issue

Not all datastructures have the same operations!

Interface and comparing datastructures

What is a good datastructure?

- Depends on the application/purpose
- Point of comparison: the operations

Issue

Not all datastructures have the same operations!

Interfaces (again, informal)

The type signatures of operation and their **specification**

In the book: called **abstract datatypes**

Example: sets

You could imagine sets being any reasonable class that has the following methods:

```
class naiveSet:
  def __init__(self, c = []):
     self.elts = [i for i in c]
  def remove(self,e): # remove e
     self.elts = [i for i in self.elts, if i != e]
  def __contains__(self,e): # do I contain e the element?
     return e in self.elts
  def __add__(self,e); # add one element
     self.elts.append(e)
  def union(self, Set s); # adds all elements of ss
     self.elts += s.elts
```

Non-OO version of the same

For didactic purposes; more idiomatic in some programming languages

```
class naiveSet:
  def __init__(self, c = []):
     self.elts = [i for i in c]
def remove(s,e): # remove e from s
  s.elts = [i for i in s.elts, if i != e]
def contains(s,e): # do I contain e the element?
  return e in self.elts
def add(s,e); # add one element
  s.elts.append(e)
def union(s1, s2); # adds all elements of ss
  s1.elts += s2.elts
```

Comparing different implementations

Different valid **implementations** for a same inferface

- many parameters for comparison:

  time/space complexity, destructive or non-destructive update, ...

Comparing different implementations

Different valid **implementations** for a same inferface

- many parameters for comparison:

  time/space complexity, destructive or non-destructive update, ...

Complexities for some implementations of `Set` (we will compute those later)

Op \Data	arrays	linked lists	Dynamic arrays	Sets (AVL)
`Set(T)`	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`remove`	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(\log(n))$
`contains`	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(\log(n))$
`add`	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$ $\mathcal{O}(1)$ amortized	$\mathcal{O}(\log(n))$
`union`	$\mathcal{O}(n+m)$	$\mathcal{O}(1)$	$\mathcal{O}(n+m)$ $\mathcal{O}(m)$ amortized	$\mathcal{O}(m\log(n))$

Datastructure for collections

For today, we will be looking at datastructures for ordered collections

- I won't give a formal definition
- but essentially, the interface of lists

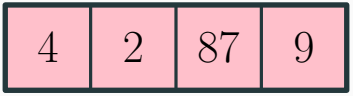
Typical operations

- Unique conversion to an array
- adding elements (arbitrarily or at a given indexed)
- removing by name/index.

Impementation of arrays (C, java, ...)

Arrays are just contiguous chunks in memory, represented by an address

(and an integer for the size in languages like java)



Impementation of arrays (C, java, ...)

Arrays are just contiguous chunks in memory, represented by an address

(and an integer for the size in languages like java)

Impementation of arrays (C, java, ...)

Arrays are just contiguous chunks in memory, represented by an address

(and an integer for the size in languages like java)

Low-level, you can *morally* think of RAM as a giant array and your CPU as a program with a fixed number of variable.

Some properties

- reading a cell at a given index is constant-time

Impementation of arrays (C, java, ...)

Arrays are just contiguous chunks in memory, represented by an address

(and an integer for the size in languages like java)

Low-level, you can *morally* think of RAM as a giant array and your CPU as a program with a fixed number of variable.

Some properties

- reading a cell at a given index is constant-time
- synergize well with hardware optimizations

(i.e., caching, nested loop parallelization)

Impementation of arrays (C, java, ...)

Arrays are just contiguous chunks in memory, represented by an address

(and an integer for the size in languages like java)

Low-level, you can *morally* think of RAM as a giant array and your CPU as a program with a fixed number of variable.

Some properties

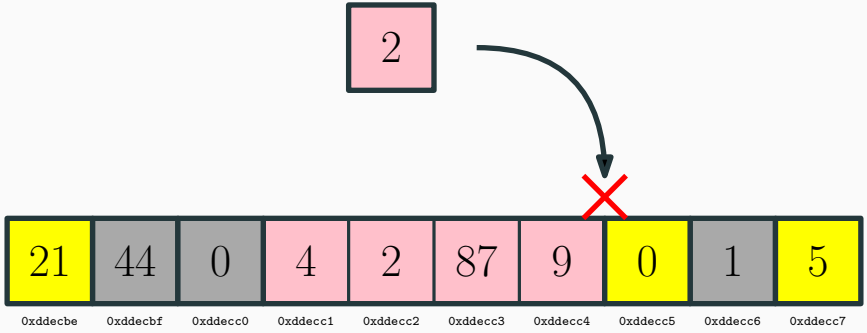
- reading a cell at a given index is constant-time
- synergize well with hardware optimizations

(i.e., caching, nested loop parallelization)

(this does not matter for *asymptotic* complexity)

Some things that are not too efficient with arrays

- Creating an array of size n is $\mathcal{O}(n)$
- **fixed size** $\Longrightarrow$ to extend, we need to re-create and copy an array
- Reason: your OS might have already allocated the memory around an array for some other purpose, so no guarantees of having anything free.



Can we do better?

Can we have $\mathcal{O}(1)$ insertion/deletion? At the end? Beginning? Middle?

Dynamic arrays

The answer is the workhorse behind `ArrayList<T>`

In a nutshell

An overlay on top of an array with a smart memory management policy.

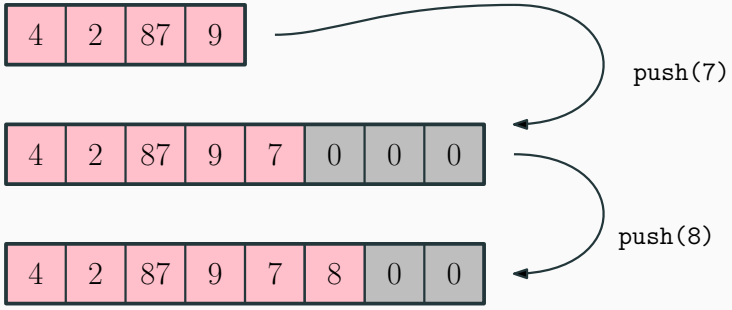
```
public class DynArrayInt {
private int[] internalArray;
private int size;
  ... }
```

Invariant: the size of internalArray is $= 2^{\lceil \log_2(\texttt{size}) \rceil}$

- This is more than needed
- Idea: plan ahead and reserve some space for future additions

Adding an element in a dynamic array

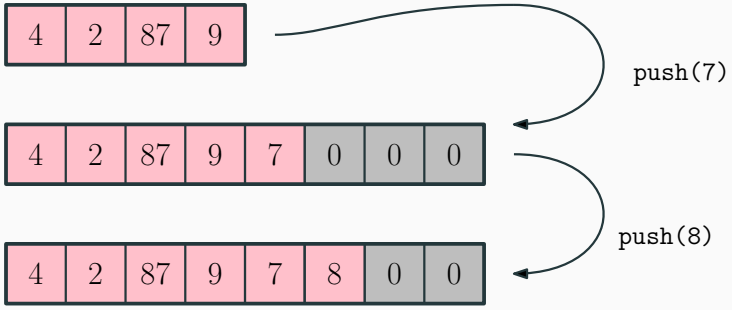
Let's picture adding 7 and 8 at the end of our running example:



Complexity: sometimes $\geq n$, sometimes $\mathcal{O}(1)\ldots$

Adding an element in a dynamic array

Let's picture adding 7 and 8 at the end of our running example:



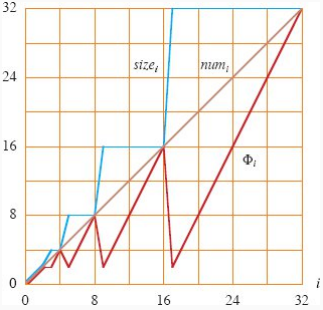
Complexity: sometimes $\geq n$, sometimes $\mathcal{O}(1)\ldots$

Constant amortized complexity!

Adding n elements to an *empty array* is $\mathcal{O}(n)$

Amortized complexity?

Idea: average during the whole lifecycle of a datastructure



(Cormen et. al., Intoduction to Algorithms 4th ed., fig 16.4)

Morally for python lists

- You can *mostly* pretend `append` is $\mathcal{O}(1)$
- But not always (e.g. real-time applications).
- (Slightly less space efficient than arrays)

But what if...

... we want $\mathcal{O}(1)$, even if we're starting from a big structure?

Simply linked lists: high-level idea

Recursive definition

A linked list is either

- a flag denoting an empty list
- or a cell containing a value and a reference to a linked list

Useful vocabulary for non-empty values

- **head** = value of the first cell
- **tail** = the remainder of the list

Example implementation in java

We need to use **recursively defined classes**

```
class llist:
  def __init__(self, x, n = None):
    self.head = x     # head = first value
    self.tail = n     # tail = rest of the list
                      #   (by default empty)
```

Slight issue: the empty list

- Can be simulated `None`
- But a better way would be to separate the recursive class from the global structure; what's above is kinda bad actually.

In practice

Still, let's use that for now (proper implementation: OOP exercise)

```
class llist:
  def __init__(self, x, n = None):
    self.head = x     # head = first value
    self.tail = n     # tail = rest of the list
                      #   (by default empty)
```

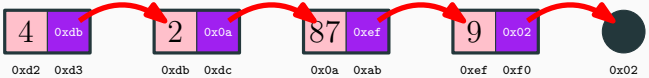
Model our example and get the third element:

```
tttail = llist(9);
ttail = llist(87,tttail);
tail = llist(2,ttail);
ex = llist(2,tail);
third = ex.tail.tail.head;
```

Quick comment about the memory layout

Not necessarily contiguous!

- Typically elements that are added in quick succession might be close, but up to implementation of the Python runtime



Adding an element in front: $\mathcal{O}(1)$!

Inserting an element (OO-style)

Suppose we want to insert an integer `x` at index `i`:

- Typically, recursion is nice to operate over recursively defined classes:

```
def insertAtIn(x, i, l):
  if i == 0:
    return llist(x, l)
  else:
    return llist(x, insertAtIn(x, i-1, l.tail))
```

Inserting an element (OO-style)

Suppose we want to insert an integer `x` at index `i`:

- Typically, recursion is nice to operate over recursively defined classes:

```
def insertAtIn(x, i, l):
  if i == 0:
    return llist(x, l)
  else:
    return llist(x, insertAtIn(x, i-1, l.tail))
```

Complexity: $\mathcal{O}(i)$

The same with loops

Lists can also be rather easily handled with loops

```
def insertAtIn(x, i, l):
  if i == 0:
    return llist(x, l)
  u = l
  while i > 1:
    u = u.tail
    --i
  u.tail = llist(x,u.tail)
  return l
```

(can you spot the subtle difference between this implementation and the previous one?)

Exercises!

Setting an element at index i	$\mathcal{O}(i)$
Deleting an element at index i	$\mathcal{O}(i)$
Reversing a list of size n	$\mathcal{O}(n)$
Array conversion	$\mathcal{O}(n)$
Concatenating	

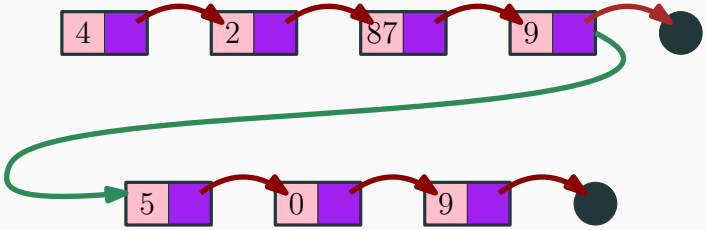
Exercises!

Setting an element at index i	$\mathcal{O}(i)$
Deleting an element at index i	$\mathcal{O}(i)$
Reversing a list of size n	$\mathcal{O}(n)$
Array conversion	$\mathcal{O}(n)$
Concatenating	$\mathcal{O}(n)$

The issue with concatenation

It seems concatenation should be $\mathcal{O}(1)$

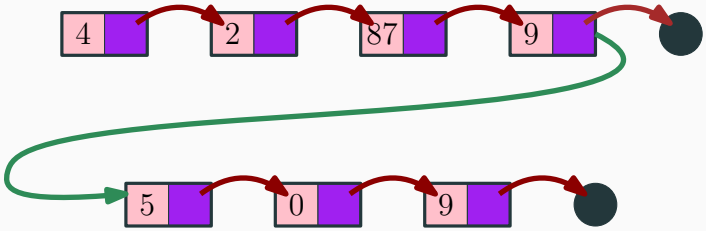
- Just modify the last tail pointer!



The issue with concatenation

It seems concatenation should be $\mathcal{O}(1)$

- Just modify the last tail pointer!



Solution: modify the datastructure to include a pointer to the end!

- To check: other operations doable with the same complexity

  that happens to be true here

- Similar exercise: adapt the datastructure so that reverse is $\mathcal{O}(1)$

  add a boolean to simulate reversing and adapt

Representing linked lists in OO properly

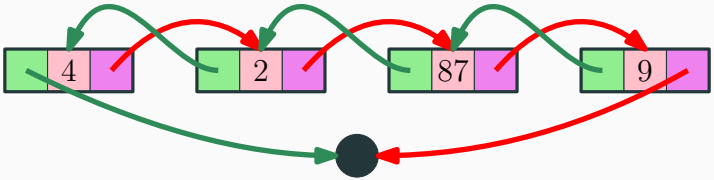
```
class cell:
  def __init__(self, v, n = None):
    self.head = v
    self.tail = n

class llist:
  def __init__(self, col = []): #coll is some iterable thing
    self.first = None # a cell
    self.last = None
    for i in col:
      self.append(i)
  # ... implementation
```

The recursion is still essential, but not in `llist`.

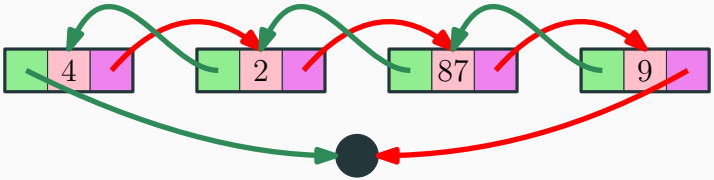
Further improvement: bidirectional links

Further improvement: doubly-linked lists



Further improvement: bidirectional links

Further improvement: doubly-linked lists



- In practice, that is what Java does for `List<T>`
- easier to navigate around → insertion in $\mathcal{O}(\min(i, n-i))$
- hard to do doubly-linked lists with *non-destructive* updates

(straightforward for singly linked-list, hence why they are useful)

In java

```
class MyCell {
  MyCell prev;
  int head;
  MyCell next;
}

class MyDoublyLinkedList {
  protected MyCell start;
  protected MyCell last;
  ...
}
```

Lab task: filling in (some of) the rest! (but in python)

Comparison with arrays

Op \Data	`Array`	`List`
deletion/insertion at i	$\mathcal{O}(n)$	$\mathcal{O}(i)$
getting/replacing the value at i	$\mathcal{O}(1)$	$\mathcal{O}(i)$
concatenating	$\mathcal{O}(n)$	$\mathcal{O}(1)$

Comparison with arrays

Op \Data	`Array`	`List`
deletion/insertion at i	$\mathcal{O}(n)$	$\mathcal{O}(i)$
getting/replacing the value at i	$\mathcal{O}(1)$	$\mathcal{O}(i)$
concatenating	$\mathcal{O}(n)$	$\mathcal{O}(1)$

Consequences:

- Note that everything is linear time (rather fast in the grand scheme of things)

Comparison with arrays

Op \Data	`Array`	`List`
deletion/insertion at i	$\mathcal{O}(n)$	$\mathcal{O}(i)$
getting/replacing the value at i	$\mathcal{O}(1)$	$\mathcal{O}(i)$
concatenating	$\mathcal{O}(n)$	$\mathcal{O}(1)$

Consequences:

- Note that everything is linear time (rather fast in the grand scheme of things)
- For batch processing where we can bound the size statically → arrays win

Comparison with arrays

Op \Data	`Array`	`List`
deletion/insertion at i	$\mathcal{O}(n)$	$\mathcal{O}(i)$
getting/replacing the value at i	$\mathcal{O}(1)$	$\mathcal{O}(i)$
concatenating	$\mathcal{O}(n)$	$\mathcal{O}(1)$

Consequences:

- Note that everything is linear time (rather fast in the grand scheme of things)
- For batch processing where we can bound the size statically → arrays win
- For **real-time** simulations (e.g. video games) with unbounded collections → lists win

To wrap up

Worth recalling the example comparison with the examples we have seen:

Complexities for some implementations of `Set`

Op \Data	`Array`	`List`	`ArrayList`
`{}`	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`remove`	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
`contains`	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
`add`	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$ $\mathcal{O}(1)$ amortized
`union`	$\mathcal{O}(n+m)$	$\mathcal{O}(1)$	$\mathcal{O}(n+m)$ $\mathcal{O}(m)$ amortized

(Table limited to set operations while we have considered more operations in the lecture)

(e.g. insertion; dynamic arrays are not better than arrays at this)

Immutable datastructures

Datatypes that *cannot* be modified.

Python strings and tuples = **immutable** arrays

```
x = ('a', [1, 2])
x[0] = 2 # Error!
x[1][0] = 5 # No error (brainteaser for you!)
```

What's the point?

- can enable optimizations
- e.g. CPython caches short strings
- (nicer for concurrent settings)

Syntactic sugar and abstract datatypes

You can define your own custom classes to be compatible with the nice syntax for list comprehension and loops.

```
xs = myClassOfCollection(blabla)
print([j * k for j in xs, k in ks, if j != k])
```

You need to implement a special method

```
def __iter__(self):
```

which returns an object implementing "a traversal"

See `https://docs.python.org/3/reference/datamodel.html` §3.3.7 for details (and this weeks' lab!)

Other methods nice to overload

```
def __str__(self): # pretty-printing for the REPL

def __len__(self): # length
def __getitem__(self, index) # a[i]
def __setitem__(self, index, val) # a[i] = v
def __contains__(self, val) # membership

def __eq__(self, other): # x == y
def __le__(self, other): # x <= y

def __add__(self, other): # x + y

...
```

See `https://docs.python.org/3/reference/datamodel.html` for more

What comes next?

Next week, three abstract datatypes with applications:

- stacks
- queues
- priority queues

That will wrap up Chapter 4 of the book.

Then we will start on tree-like datastructures (chapter 7)