CSCM12: software concepts and efficiency
Datastuctures for ordered collections

Cécilia Pradic

February 27th 2023

Algorithms and datastructures

Starting from today we are going to talk about datastructures

- More complicated datatypes
- Designing more abstractions
    - interfaces
    - invariants

Algorithms and datastructures

Starting from today we are going to talk about datastructures

- More complicated datatypes
- Designing more abstractions
  - interfaces
  - invariants

We will still discuss algorithms and efficiency

- Introducing datastructures → new tools to
  - program efficiently in any context (small/large scale, interactive/batch)
  - representations for input/outputs for algorithmic problems

Today

- Some high-level considerations (not too long)
- Our first example: linked lists
- If time allows: dynamic arrays, amortized complexity

Wot's a datastructure?

Informal concept, high-level

Rough reductionist definition

1. A chunk of memory space layed out in a specified way

   (in java, often the attributes an object of a class)

2. A bunch of **operations** (in java, 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

   (in java, often the attributes an object of a class)

2. A bunch of **operations** (in java, the methods)

In Java both aspectss *often* materialize as

Caveat: not the only way to " do datastructure"

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

   (in java, often the attributes an object of a class)

2. A bunch of **operations** (in java, the methods)

In Java both aspectss *often* materialize as

Caveat: not the only way to " do datastructure"

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

   (in java, often the attributes an object of a class)

2. A bunch of **operations** (in java, the methods)

In Java both aspectss *often* materialize as   Caveat: not the only way to " do datastructure"

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 (`ArrayList`), strings (`String`)

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!

Solution: compare across **interfaces**

Interfaces (again, informal)

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

In Java: often using `interface`

Example: a fragment of the `Set` interface

- Define an interface for datastructure `Set`
- Represent a *set* of `T`s

(caveat: not quite the official java interface)

```
Set(); // creates an empty set
void remove(T e); // removes one element
boolean contains(T e); // do I contain the element?
void add(T e); // add one element
Set union(Set s2); // adds all elements of s2
...
```

Non-OO version of the same

For didactic purposes

```
Set(); // creates an empty set
static Set remove(Set s, T e); // returns s - {e}
static boolean contains(Set s, T e); // returns whether e is in s
static Set add(Set s, T e); // returns s unioned with {e}
static Set union(Set s1, Set s2); // returns s1 unioned with s2
...
```

But also: useful when considering **immutable data**

- Mutable = references, immutable = hard data like **`int`**...
- But for now, we will mostly consider **destructive update**

(i.e. inputs are consumed/modified)

(OO meshes well with this, not so well with immutable stuff (see all those **`static`**))

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	`Array`	`List`	`ArrayList`	`TreeSet`
`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, we are going to look at array-like interfaces

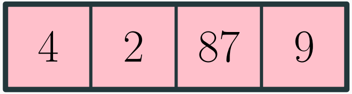
Typical operations

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

Impementation of arrays

Arrays are contiguously represented in memory by an address

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



Impementation of arrays

Arrays are contiguously represented in memory by an address

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

Impementation of arrays

Arrays are contiguously represented in memory by an address

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

Some properties

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

Impementation of arrays

Arrays are contiguously represented in memory by an address

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

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

Arrays are contiguously represented in memory by an address

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

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)

Impementation of arrays

Arrays are contiguously represented in memory by an address

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

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)

Source of the tradeoff for lists

Non-contiguous representation in memory, but still a linear stucture

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 MyLinkedList
{
    int head;
    MyLinkedList tail;

    MyLinkedList(int nHead, MyLinkedList nTail) {
      head = nHead; tail = nTail;
    }
}
```

Slight issue: the flag for the empty list

- Can be simulated `null`
- But bad practice here for java

(cascade of design issues wrt encapsuation, ...)

In practice

Still, let's use that for the lecture (proper implementation: tedious OO exercise)

```
class MyLinkedList {
  int head;
  MyLinkedList tail;
}
```

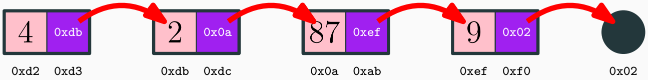
Model our example and get the third element:

```
MyLinkedList empty = null;
MyLinkedList tttail = new MyLinkedList(9,empty);
MyLinkedList ttail = new MyLinkedList(87,tttail);
MyLinkedList tail = new MyLinkedList(2,ttail);
MyLinkedList ex = new MyLinkedList(2,tail);
int 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 this is up to the implementation of **new**



Adding an element

The easiest thing is to add an element in front

- Non-OO-style:

```
static MyLinkedList push(MyLinkedList xs, int x){
  return new MyLinkedList(x, xs);
}
```

- OO-style:

```
MyLinkedList push(int x){
  return new MyLinkedList(x, this);
}
```

Careful: `xs.push(2)` does not modify `xs`

$\mathcal{O}(1)$!

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:

```
MyLinkedList insert(int i, int x){
  if(i == 0) then
    return push(x);
  tail.insert(i-1, x);
  return this;
}
```

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:

```
MyLinkedList insert(int i, int x){
  if(i == 0) then
    return push(x);
  tail.insert(i-1, x);
  return this;
}
```

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

The same with loops

Lists can also be rather easily handled with loops

```
MyLinkedList insert(int i, int x){
  if(i == 0)
    return push(x);
  MyLinkedList previousNode;
  for(tmp = this; i > 1; --i)
     tmp = tmp.tail;
  tmp.tail = tmp.tail.push(x);
  return this;
}
```

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	

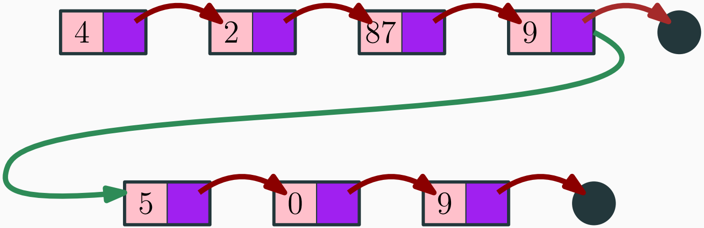
Exercises!

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 $\mathcal{O}(1)$

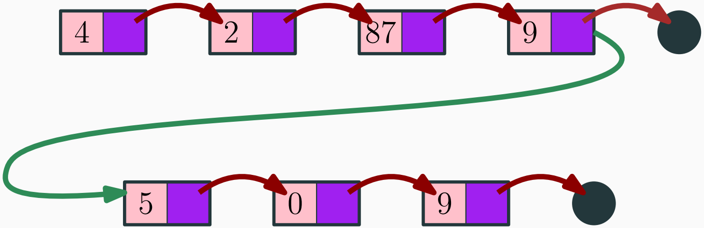
- Just modify the last tail pointer!



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

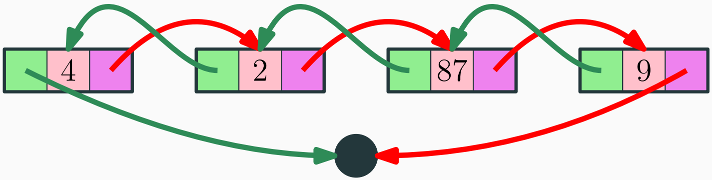
```
class MyCell {
  int head;
  MyCell tail;
}

class MyLinkedList {
  protected boolean empty;
  protected MyCell start;
  protected MyCell last;
  protected boolean flip;
  ...
}
```

The recursion is still essential, but not exposed by `MyLinkedList`.

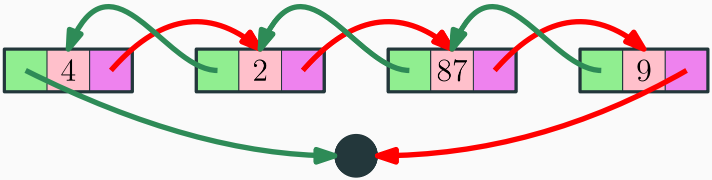
Further improvement: bidirectional links

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 $\rightarrow$ 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

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

class MyDoublyLinkedList {
  protected boolean empty;
  protected MyCell start;
  protected MyCell last;
  protected boolean flip;
  ...
}
```

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 $\rightarrow$ 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 $\rightarrow$ arrays win
- For **real-time** simulations (e.g. video games) with unbounded collections $\rightarrow$ lists 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 $\rightarrow$ arrays win
- For **real-time** simulations (e.g. video games) with unbounded collections $\rightarrow$ lists win

What about batch-processing with unbounded size?

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.

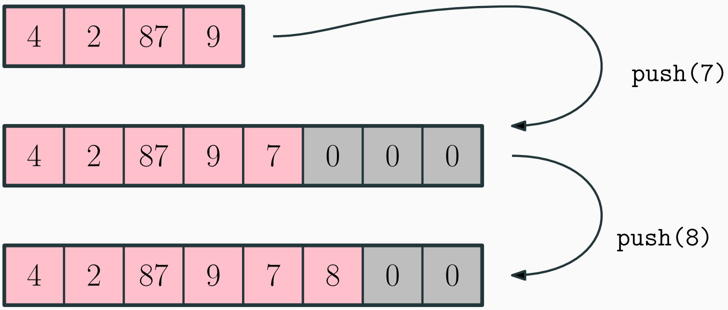
```
public class DynArrayInt {
private int[] internalArray;
private int size;
  ... }
```

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

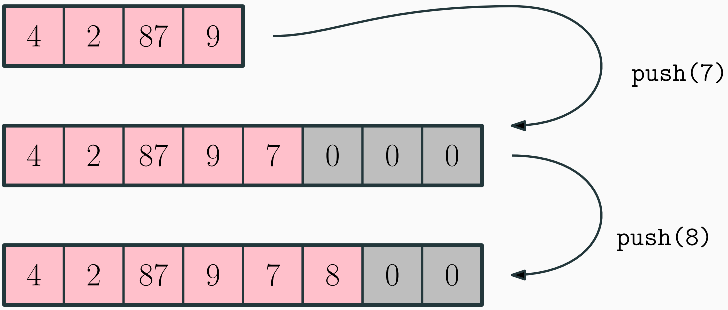
Adding an element in a dynamic array

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



Adding an element in a dynamic array

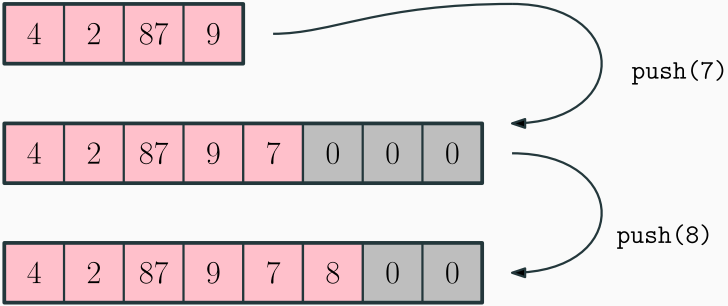
Let's picture adding 7 and 8 at the end of our running example:



Sometimes $\Theta(n)$, sometimes $\mathcal{O}(1)$...

Adding an element in a dynamic array

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



Sometimes $\Theta(n)$, sometimes $\mathcal{O}(1)$...

Constant amortized complexity! **(advanced material, maybe later)**

Adding k elements to the empty array is $\mathcal{O}(k)$

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`
`Set(T)`	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`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

Note that this is limited to set operations while we have considered more operations in the lecture (e.g. insertion; synamic arrays are not better than arrays at this)