CSCM12: software concepts and efficiency
Datastructures for ordered collections

Cécilia Pradic

March 14th 2024

slido.com, code #1426271

Announcement: IT issue

Due to an IT incident, some CS systems are down including:

- **Autograder**:
  - this means I can't open the submission platform for coursework 2 yet
  - will see with the program director what to do with the deadline
- **the lab tracker**: we will record people who did lab for sign-offs and will port this back to the lab tracker at a later time, will not affect you besides your being unable to check what you are signed off for *temporarily*
- **maybe java on the lab machines**: unsure about that, will try to remember to test later today and make an announcement on canvas once I am sure; please bring your laptop tomorrow if you can.

It's likely this will be all resolved next week, fingers crossed.

What we did last week

- The notion of datastructure
- Interface for set-like datastructue (one motivating example)
- Linked lists (singly, doubly)
- Introduce CW2

What we did last week

- The notion of datastructure
- Interface for set-like datastructue (one motivating example)
- Linked lists (singly, doubly)
- Introduce CW2

Today

- Dynamic arrays
- Stacks
- Queues

What we did last week

- The notion of datastructure
- Interface for set-like datastructue (one motivating example)
- Linked lists (singly, doubly)
- Introduce CW2

Today

- Dynamic arrays
- Stacks
- Queues

For the latter two topics, I will use some (copyrighted) slides from Gary.

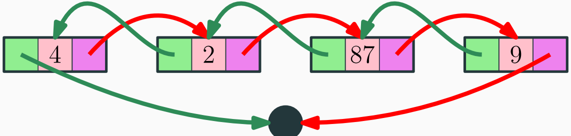
Reminders: CW2

Task: complete a doubly-linked implementation!

```
class Node<T> {
  Node<T> prev;
  T val;
  Node<T> next;
}

class DLList<T> {

  Node<T> first;
  Node<T> last;
  int length;
  }
```

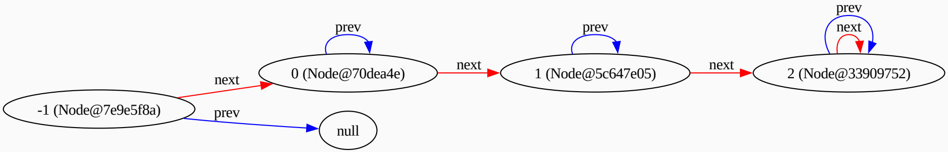


A question to you (answers in the room or slido)

Does the following bit of code build a correct doubly-linked list? If not, what are the issues? how should they be fixed?

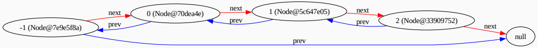
```
DLList<Integer> l = new DLList<Integer>();
Node<Integer> previous = new Node<Integer>();
previous.val = -1;
l.first = previous;
for(int i = 0; i < 3; ++i)
{
  Node<Integer> n = new Node<Integer>();
  n.val = i;
  previous.next = n;
  previous = n;
  n.prev = previous;
}
l.last = previous;
previous.next = l.last;
```

Answer: no


prev
prev
prev
next
next
next
next
-1 (Node@7e9e5f8a)
0 (Node@70dea4e)
1 (Node@5c647e05)
2 (Node@33909752)
prev
null

A correction

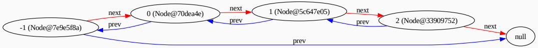
```java
DLList<Integer> l = new DLList<Integer>();
Node<Integer> previous = new Node<Integer>();
previous.val = -1;
l.first = previous;
for(int i = 0; i < 3; ++i)
{
  Node<Integer> n = new Node<Integer>();
  n.val = i;
  previous.next = n;
  n.prev = previous;
  previous = n;
}
l.last = previous;
```



Comments

Advice for the CW2

- Double-check that you handle forward and backward references properly!
- Be mindful of when references are set to `null`
- Test thoroughly your functions on diverse examples
  - Suggestion: at least three kind of list examples: an empty list, a list of size one, and a larger list



(I can make the code that generates the memory graphs for `DLList<T>` like the one above available on canvas if you are interested; it generates a textual description in `graphviz` that you can turn into a picture using either a local `graphviz` installation or an online tool like `https://viz-js.com/`).

Comparison list/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 list/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 list/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 list/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 list/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.

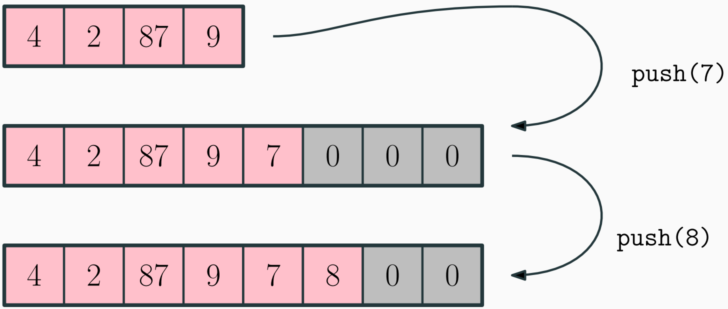
```
public class DynArrayInt {
private int[] internalArray;
private int size;
  ... }
```

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

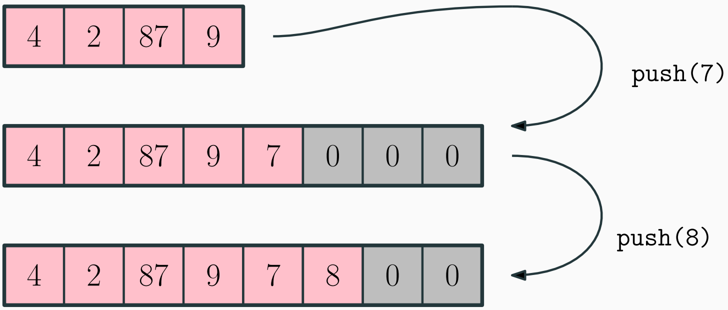
Adding an element in a dynamic array

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



Adding an element in a dynamic array

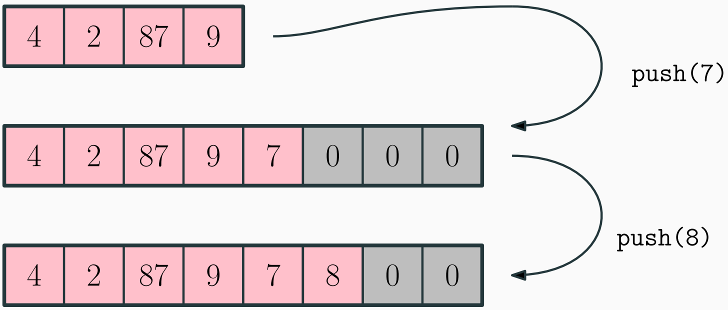
Let's picture adding 7 and 8 at the end of our running example:



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!

Adding k elements to an array of size n the empty array is $\mathcal{O}(n+k)$

Exercises about dynamic arrays

You can look up the 2022 exam of CSCM41J question III :)

(will upload it later today on the module's page)

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

(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)