Introduction to Algorithms and Data Structures
Lecture 7: Classic datatypes: lists, stacks, queues

John Longley

School of Informatics
University of Edinburgh

10 October 2023
‘Lists’ in general . . .

We’ve seen that ‘lists’ can be implemented in several ways, e.g. via arrays or linked lists. **How might we compare these?**

Start by listing the operations we’d like any impl to support.

E.g. for (unsorted) lists of items of type $X$, might want operations

- **get** : $\mathbb{int} \rightarrow X$  
  # read item at given pos
- **set** : $\mathbb{int} \times X \rightarrow \mathbb{void}$  
  # write item at given pos
- **cons** : $X \rightarrow \mathbb{void}$  
  # add item at start
- **append** : $X \rightarrow \mathbb{void}$  
  # add item at end
- **insert** : $\mathbb{int} \times X \rightarrow \mathbb{void}$
- **delete** : $\mathbb{int} \rightarrow \mathbb{void}$
- **length** : $\mathbb{void} \rightarrow \mathbb{int}$

Much like an **interface** in Java.
Abstract interfaces, concrete implementations

As in Java, we can consider various concrete implementations of this abstract interface.

Further points:

- For some purposes, could consider an interface with fewer operations, or with more: e.g. `reverse : void → void`.
- May be other operations that make sense for specific impl’s. E.g. for linked lists, ‘insert/delete at current position’ is useful.
- Some of our operations will be definable from others: e.g.

\[
\text{append}(x) \equiv \text{insert} (\text{length}(), x)
\]

But may want to include `append` in its own right: could be implementable more efficiently than general `insert`. 
**Implementation 1: Fixed-size arrays**

Use an **array** \( A \) of some **fixed** size \( m \).
Can store a list \( L = x_0, \ldots, x_{n-1} \) (where \( n \leq m \))
in the first \( n \) cells of \( A \) (so \( A[i] = x_i \) for each \( i < n \)).
Also want an integer variable \( n \) to store the value of \( n \).

List operations are easy to implement. E.g.

\[
\begin{align*}
\text{get}(i): & \quad \text{return } A[i] \\
\text{append}(x): & \quad A[n] = x \\
\text{insert} (i,x): & \quad \text{for } j = n-1 \text{ downto } i \\
 & \quad A[i] = x \\
 & \quad n = n+1
\end{align*}
\]

- **length**, **get**, **set** and **append** (when it works) take \( \Theta(1) \) time.
- **cons**, **insert**, **delete** require \( \Theta(n) \) time in worst case.
Fixed-size arrays have some strengths . . .

- Fast **get** and **set** operations – especially if we can keep the array on the stack!
- Fixed, predictable size good for memory management. (If on stack, can reclaim space immediately on expiry.)

. . . but a major weakness . . .

- Can’t cope with lists longer than pre-set limit $m$.
- If a computation involves a lot of lists, of unpredictable sizes, very likely we’ll either **under-cater** (some array will overflow) or **over-cater** (many arrays will contain a lot of wasted space).

So not a good choice for ‘general-purpose’ lists.
Implementation 2: Extensible arrays

Idea is simple: if array A overflows, replace it by a bigger one!

- If memory space ‘after’ A happens to be free, cheap to do.
- But if not, may have to allocate a fresh array B, and copy contents of A into it. E.g. for some real number $r > 1$:

\[
\text{append } (x):
\]
\[
\text{if } n = |A|
\]
\[
B = \text{new array (}[n \times r]\)
\]
\[
\text{copy contents of } A \text{ into } B \text{ (n items)}
\]
\[
A = B
\]
\[
\# \text{ Now do ordinary append:}
\]
\[
A[n] = x
\]
\[
n = n+1
\]

So a ‘normal’ append takes $\Theta(1)$ time – but occasionally we may get a bad one, taking $\Theta(n)$.

 Might seem ‘dirty’, but widely used in practice. Runtime analysis is interesting . . .
Amortized cost

Perhaps in some apps, even one bad append day could be fatal.

But often, we’re happy if over any long run of appends, the average time is reasonable. A bad one may be acceptable if we regard its cost as amortized (‘spread out’) over the next 100 good ones – i.e. if invested effort ‘pays for itself’ over time.

Does it?

Suppose array has initial capacity $a$, and starting from nil we do $m$ appends in succession, expanding by factor $r > 1$ when need be.

Array size grows as $a, ar, ar^2, ar^3, \ldots$. How many steps to reach $m$?

Solving ‘$ars = m$’ yields $s = \log_r(m/a)$ for the number of steps.

An item may get copied this many times!

Since potential number of copyings of an item grows with $m$, might suspect ‘average cost per append’ also grows with $m$ . . . ??

Let’s do the sums.
Calculating amortized cost of append

Example: Suppose $a = 100$, $r = 1.1$, $m = 5000$.

Note that $1.1^{41}a < m < 1.1^{42}a$. So will need 42 expansions.

Ignoring ‘rounding’, number of copyings ($B[i] = A[i]$) is basically

$$100 \times (1 + 1.1 + 1.1^2 + \cdots + 1.1^{41})$$

By ‘sum of geometric progression’ formula, this is

$$100 \times (1.1^{42} - 1)/(1.1 - 1) < 1.1m/0.1$$

So although some items get copied 42 times, average no. of copyings per item stays below $1.1/0.1 = 11$.

In general, total number of copyings is basically at most $m(r/(r-1))$. So average no. of copyings per item stays below $r/(r - 1)$.

\[IADS\ Lecture\ 7\ Slide\ 8\]
Again suppose $a = 100, \ r = 1.1$.

Imagine a copying costs 1p. Each time we do an **append**, we pay 11p into a **pension fund** to pay for future copyings.

Suppose we’ve just done our first expansion. Array now has 110 cells, with 100 filled. Next 10 **appends** pay for second expansion (110 copyings).

After second expansion, array has 121 cells, 110 filled. Next 11 **appends** pay for third expansion (121 copyings) . . .

So each **append** incurs a constant cost of 11 copyings.
Amortized cost: conclusion

So total time taken by expansion/copying is \( O(m) \).

But time taken by ordinary **append**s is also clearly \( O(m) \).

So may say the **amortized cost of append** is \( O(1) \) per operation.

- Lists in Python are implemented like this, essentially with \( r = 9/8 \). Underlying arrays may also be shrunk if proportion in use dips below 1/2. (For analysis, see CLRS 17.4.)

- Java class **ArrayList** also works like this. Precise expansion policy not prescribed, but it's required that amortized cost over a long run must be \( O(1) \) per operation.

Of course, **cons**, **insert**, **delete** still take time \( \Theta(n) \) in worst case (even amortized).
Implementation 3: Linked lists

We can also represent the lists over $X$ using linked lists, where each cell contains a key of type $X$.

Clearly, for a list of length $n$:

- **get** and **set** have $\Theta(n)$ worst-case time (but with small ‘$C$’)
- **cons** takes $\Theta(1)$ time, always.
- **insert**($i,x$), **delete**($i$) have $\Theta(n)$ worst-case time (or $\Theta(1)$ if we’ve already located the cell at position $i-1$).

Linked lists also naturally allow for sharing (unlike arrays).

**Offline exercise:** Show how the list of all $2^n$ binary lists of length $n$ can be stored in $\Theta(2^n)$ space with linked list impl. (Would take $\Theta(n.2^n)$ with arrays.)
List implementations: summary

Upper bounds on runtimes (where $n$ is length of list):

<table>
<thead>
<tr>
<th>Operation</th>
<th>Array impl</th>
<th>Linked-list impl</th>
</tr>
</thead>
<tbody>
<tr>
<td>get</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>set</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>cons</td>
<td>* $O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>append</td>
<td>* $O(n)$ (amortized $O(1)$)</td>
<td>$O(n)$ (can make it $O(1)$)</td>
</tr>
<tr>
<td>insert</td>
<td>* $O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>delete</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

Operations marked * may fail for fixed-array implementations, or trigger expansion for extensible-array ones.

So arrays offer fast **get/set**; linked lists offer fast **cons/append** and **insert/delete** at given position, plus sharing.

?? Is there some impl of lists for which **all** the above are ‘fast’ ??

**Find out in Lecture 9!**
Stacks and queues

Sometimes, we know that some list will only be manipulated in certain restricted ways, e.g. …

- Elements only ever added/read/removed at front of list (stack or Last-in-first-out buffer)
- Elements added at back, read/removed at front of list (queue or First-in-first-out buffer)

Knowing this may affect our choice of implementation.

**Interfaces** for stacks and queues (of items of type $X$):

<table>
<thead>
<tr>
<th>STACKS:</th>
<th>QUEUES:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>empty</strong>: $\text{void} \rightarrow \text{bool}$</td>
<td><strong>empty</strong>: $\text{void} \rightarrow \text{bool}$</td>
</tr>
<tr>
<td><strong>push</strong>: $X \rightarrow \text{void}$</td>
<td><strong>enqueue</strong>: $X \rightarrow \text{void}$</td>
</tr>
<tr>
<td><strong>peek</strong>: $\text{void} \rightarrow X$</td>
<td><strong>peek</strong>: $\text{void} \rightarrow X$</td>
</tr>
<tr>
<td><strong>pop</strong>: $\text{void} \rightarrow X$</td>
<td><strong>dequeue</strong>: $\text{void} \rightarrow X$</td>
</tr>
</tbody>
</table>
Implementing stacks

In principle, any impl of lists yields an impl of stacks:
But two obvious candidates:

- arrays (growing at end)
- linked lists (growing at start)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Extensible array impl</th>
<th>Linked list impl</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>push</td>
<td>* ( O(n) ) (amortized ( O(1) ))</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>peek</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>pop</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
</tbody>
</table>
Implementing queues

**Impl 1:** Wraparound array buffer (fixed-size/extensible)
**Impl 2:** Linked list with references to first and last cells
Implementing queues, ctd.

How would e.g. enqueue look in each case?

**Wraparound array:**

\[
\text{enqueue}(x):
\]

\[
j = (j+1) \mod |A|
\]

\[
\text{if } j = i \text{ fail (or expand)}
\]

\[
\text{else } A[j] = x
\]

**Linked list:**

\[
\text{enqueue}(x):
\]

\[
\text{last.next} = \text{new Cell}(x, \text{null})
\]

\[
\text{last} = \text{last.next}
\]

For further details, see Python Lab Sheet 3.

Situation similar to stacks:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Wraparound array impl</th>
<th>Linked-list impl</th>
</tr>
</thead>
<tbody>
<tr>
<td>enqueue</td>
<td>( O(n) ) (amortized ( O(1) ))</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>peek</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>dequeue</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
</tbody>
</table>
Reading

Stacks and queues: CLRS chapter 10.
Table expansion / amortized analysis: CLRS section 17.4.