Inf 2B: Sequential Data Structures

Lecture 3 of ADS thread

Kyriakos Kalorkoti

School of Informatics
University of Edinburgh

Abstract Data Types (ADTs)

The “Specification Language" for Data Structures. An ADT consists of:

- a mathematical model of the data;
- methods for accessing and modifying the data.

An ADT does not specify:

- How the data should be organised in memory (though the ADT may suggest to us a particular structure).
- Which algorithms should be used to implement the methods.

An ADT is what, not how.

Data Structures

how . . .

A *data structure realising* an ADT consists of:

- collections of variables for storing the data;
- algorithms for the methods of the ADT.

In terms of JAVA:

ADT	$\leftrightarrow$	JAVA interface
data structure	$\leftrightarrow$	JAVA class

The data structure (with algorithms) has a large influence on the *algorithmic efficiency* of the implementation.

Stacks

A *Stack* is an ADT with the following methods:

- push(e): Insert element e.
- pop(): Remove the *most recently inserted* element and return it;
  - an error occurs if the stack is empty.
- isEmpty(): Returns TRUE if the stack is empty, FALSE otherwise.
- Last-In First-Out (**LIFO**).

Can implement *Stack* with worst-case time $O(1)$ for all methods, with *either* an array *or* a linked list.

The reason we do so well? . . . Very simple operations.

Applications of *Stacks*

- Executing Recursive programs.
- **Depth-First Search** on a graph (coming later).
- Evaluating (postfix) Arithmetic expressions.

Algorithm postfixEval($s_1 \dots s_k$)

for $i \leftarrow 1$ **to** k **do**
    if (s_i is a number) **then** push(s_i)
    else   *(s_i must be a (binary) operator)*
        $e2 \leftarrow$ pop();
        $e1 \leftarrow$ pop();
        $a \leftarrow e1\ s_i\ e2$;
        push(a)
return pop()

- Example: `6 4 - 3 * 10 + 11 13 - *`

Queues

A *Queue* is an ADT with the following methods:

- enqueue(e): Insert element e.
- dequeue(): Remove the element *inserted the longest time ago* and return it;
  - an error occurs if the queue is empty.
- isEmpty(): Return TRUE if the queue is empty and FALSE otherwise.
- First-In First-Out (**FIFO**).

Queue can easily be realised by a data structures based *either* on arrays *or* on linked lists.
Again, all methods run in $O(1)$ time (simplicity).

Sequential Data

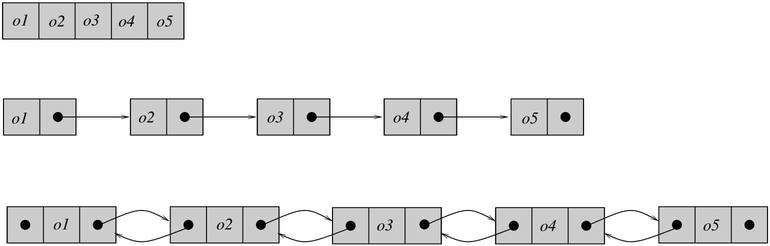
Mathematical model of the data: a linear *sequence* of elements.

- A sequence has well-defined *first* and *last* elements.
- Every element of a sequence except the last has a unique *successor*.
- Every element of a sequence except the first has a unique *predecessor*.
- The rank of an element e in a sequence S is the number of elements before e in S.

Stacks and *Queues* are sequential.

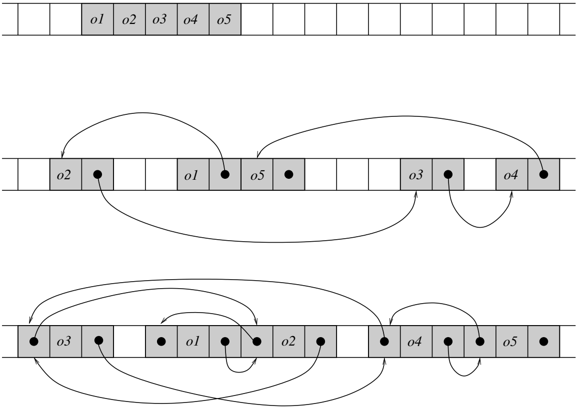
Arrays and Linked Lists abstractly

An array, a singly linked list, and a doubly linked list storing objects $o1, o2, o3, o4, o5$:



Arrays and Linked Lists in Memory

An array, a singly linked list, and a doubly linked list storing objects $o1, o2, o3, o4, o5$:



Vectors

A *Vector* is an ADT for storing a sequence S of n elements that supports the following methods:

- elemAtRank(r): Return the element of rank r; an error occurs if $r < 0$ or $r > n - 1$.
- replaceAtRank(r, e): Replace the element of rank r with e; an error occurs if $r < 0$ or $r > n - 1$.
- insertAtRank(r, e): Insert a new element e at rank r (this increases the rank of all following elements by 1); an error occurs if $r < 0$ or $r > n$.
- removeAtRank(r): Remove the element of rank r (this reduces the rank of all following elements by 1); an error occurs if $r < 0$ or $r > n - 1$.
- size(): Return n, the number of elements in the sequence.

Array Based Data Structure for *Vector*

Variables

- Array A (storing the elements)
- Integer n = number of elements in the sequence

Array Based Data Structure for *Vector*

Methods

Algorithm elemAtRank(r)

1. **return** $A[r]$

Algorithm replaceAtRank(r, e)

1. $A[r] \leftarrow e$

Algorithm insertAtRank(r, e)

1. **for** $i \leftarrow n$ **downto** $r + 1$ **do**
2. $\quad A[i] \leftarrow A[i - 1]$
3. $A[r] \leftarrow e$
4. $n \leftarrow n + 1$

insertAtRank assumes the array is big enough!
See later . . .

Array Based Data Structure for *Vector*

Algorithm removeAtRank(r)

1. **for** $i \leftarrow r$ **to** $n - 2$ **do**
2. $\quad\quad A[i] \leftarrow A[i+1]$
3. $n \leftarrow n - 1$

Algorithm size()

1. **return** n

Running times (for Array based implementation)
$\Theta(1)$ for **elemAtRank**, **replaceAtRank**, **size**
$\Theta(n)$ for **insertAtRank**, **removeAtRank** (worst-case)

Abstract Lists

List is a sequential ADT with the following methods:

- element(p): Return the element at position p.
- first(): Return position of the first element; error if empty.
- isEmpty(): Return TRUE if the list is empty, FALSE otherwise.
- next(p): Return the position of the element following the one at position p; an error occurs if p is the last position.
- isLast(p): Return TRUE if p is last in list, FALSE otherwise.
- replace(p, e): Replace the element at position p with e.
- insertFirst(e): Insert e as the first element of the list.
- insertAfter(p, e): Insert element e after position p.
- remove(p): Remove the element at position p.

Plus: last(), previous(p), isFirst(p), insertLast(e), and insertBefore(p, e)

Realising *List* with Doubly Linked Lists

Variables

- *Positions* of a *List* are realised by *nodes* having fields *element*, *previous*, *next*.
- List is accessed through node-variables *first* and *last*.

Method (example)

Algorithm insertAfter(p, e)

1. create a new node q
2. $q.element \leftarrow e$
3. $q.next \leftarrow p.next$
4. $q.previous \leftarrow p$
5. $p.next \leftarrow q$
6. $q.next.previous \leftarrow q$

Realising *List* using Doubly Linked Lists

Method (example)

Algorithm remove(p)

1. $p.previous.next \leftarrow p.next$
2. $p.next.previous \leftarrow p.previous$
3. delete p

Running Times (for Doubly Linked implementation).
All operations take $\Theta(1)$ time ...

ONLY BECAUSE of pointer representation (p is a direct link)

$O(1)$ bounds partly because we have simple methods.

search would be inefficient in this implementation of *List*.

Dynamic Arrays

What if we try to insert too many elements into a fixed-size array?

The solution is a Dynamic Array.

Here we implement a dynamic *VeryBasicSequence* (essentially a queue with no dequeue()).

VeryBasicSequence

VeryBasicSequence is an ADT for sequences with the following methods:

- elemAtRank(r): Return the element of S with rank r; an error occurs if $r < 0$ or $r > n - 1$.
- replaceAtRank(r, e): Replace the element of rank r with e; an error occurs if $r < 0$ or $r > n - 1$.
- insertLast(e): Append element e to the sequence.
- size(): Return n, the number of elements in the sequence.

Dynamic Insertion

Algorithm insertLast(e)

if $n < A.length$ **then**
    $A[n] \leftarrow e$
else $\triangleright$ $n = A.length$, i.e., the array is full
    $N \leftarrow 2(A.length + 1)$
    Create new array A' of length N
    for $i = 0$ **to** $n - 1$ **do**
        $A'[i] \leftarrow A[i]$
    $A'[n] \leftarrow e$
    $A \leftarrow A'$
$n \leftarrow n + 1$

Analysis of running-time

Worst-case analysis
elemAtRank, replaceAtRank, and size have $\Theta(1)$ running-time.
insertLast has $\Theta(n)$ worst-case running time for an array of length n (instead of $\Theta(1)$)

In **Amortised analysis** we consider the total running time of a sequence of operations.

Theorem

Inserting m elements into an initially empty VeryBasicSequence using the method insertLast *takes* $\Theta(m)$ *time.*

Amortised Analysis

- m insertions $I(1), \ldots, I(m)$. Most are *cheap* (cost: $\Theta(1)$), some are *expensive* (cost: $\Theta(j)$).
- Expensive insertions: $I(i_1), \ldots, I(i_\ell)$, $1 \leq i_1 < \ldots < i_\ell \leq m$.

  $i_1 = 1$, $i_2 = 3$, $i_3 = 7, \ldots, i_{j+1} = 2i_j + 1, \ldots$
  $\Rightarrow 2^{r-1} \leq i_r < 2^r$
  $\Rightarrow \ell \leq \lg(m) + 1$.

$$
\begin{aligned}
\sum_{j=1}^{\ell} O(i_j) + \sum_{\substack{1 \leq i \leq m \\ i \neq i_1, \ldots, i_\ell}} O(1) &\leq O\Big(\sum_{j=1}^{\ell} i_j\Big) + O(m) \\
&\leq O\Big(\sum_{j=1}^{\ell} 2^j\Big) + O(m) \\
&= O\Big(2^{\lg(m)+2} - 2\Big) + O(m) \\
&= O(4m - 2) + O(m) \\
&= O(m).
\end{aligned}
$$

Reading

- `Java Collections Framework: Stack`, `Queue`, `Vector`. (Also, `Java`'s ArrayList behaves like a dynamic array).
- Lecture notes 3 (handed out).
- If you have [GT]:
  Chapters on "Stacks, Queues and Recursion" and "Vectors, Lists and Sequences".
- If you have [CLRS]:
  "Elementary data Structures" chapter (except trees).