CITS2200 Data Structures and Algorithms

Topic 8

Amortized Analysis

- Amortized Case Analysis for Data Structures
- An example of amortized analysis using Multipop stack
- The Simplist ADT
- An amortized analysis of *beforeFirst*.
- Complexity Examples

Reading: CLRS Chapter 17, Weiss Section 22.1

0.1 Amortized Case Analysis

Amortized analysis is a variety of worst case analysis, but rather than looking at the cost of doing the operation once, it examines the cost of repeating the operation in a sequence.

That is, we determine the worst case complexity $T(n)$ of performing a sequence of n operations, and report the amortized complexity as $T(n)/n$.

An alternative view is the accounting method: determine the individual cost of each operation, including both its execution time and its influence on the running time of future operations. The analogy: imagine that when you perform fast operations you deposit some "time" into a savings account that you can use when you run a slower operation.

Reading: Cormen, Leiserson, Rivest, and Stein, *Introduction to Algorithms*, Chapter 17.

0.2 Amortized Analysis for a Multi-delete Stack

A multi-delete stack is the stack ADT with an additional operation:

1. *mPop(i)*: delete the top i elements from the stack

Assuming a linked representation, the obvious way to execute `mPop(i)` is to perform pop i times.

If each *pop* takes *b* time units, *mPop(i)* will take approximately ib time units — linear in i!

Worst case is nb time units for stack of size n.

But. . .

Before you can delete i elements, need to (somewhere along the way. . .) individually insert i elements, which takes i operations and hence ic time for some constant c.

Total for those $i+1$ operations is $i(c+b)$. The time for i operations is approximately linear in i. The *average* time for each operation

$$\frac{i}{i+1}(c+b)$$

is approximately constant — independent of i.

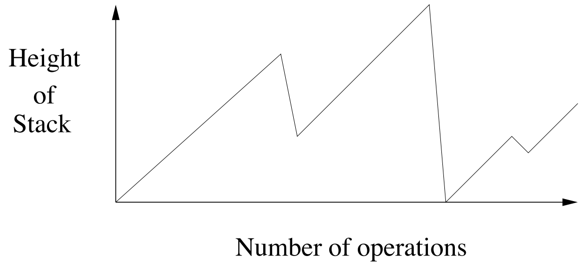
More accurate for larger i, which is also where its more important!

$$\left(\lim_{i\to\infty}\frac{i}{i+1}(c+b) = c+b\right)$$

This is called an *amortized analysis*. The cost of an expensive operation is amortized over the cheaper ones which *must* accompany it.

The Accounting Method for the Multi-delete Stack

Every time push is called we take a constant time (say a) to perform the operation, but we also put a constant amount of time (say b) in our "time-bank". When it comes time to perform multi-pop $mPop(i)$, if there are i items to delete, we must have at least ib time units in the bank.



Where Amortized Analysis Makes a Difference

In the block implementations of the data structures we have seen so far, we simply throw an exception when we try to add to a full structure.

Several implementations (e.g. `Java.util.ArrayList`) do not throw an exception in this case, but rather create an array *twice* the size, copy all the elements in the old array across to the new array, and then add the new element to the new array.

This is an expensive operation, but it can be shown that the *amortized* cost of the *add* operation is constant.

1. The Simplist ADT

The List ADT provides multiple explicit windows — we need to identify and manipulate windows in any program which uses the code.

If we only need a single window (eg a simple "cursor" editor), we can write a simpler ADT $\Rightarrow$ Simplist.

- single, implicit window (like Queue or Stack) — no need for arguments in the procedures to refer to the window position

We'll also provide only one window initialisation operation, *beforeFirst*

We'll show that, because of the single window, all operations except *beforeFirst* can be implemented in constant time using a singly linked list! Uses a technique called *pointer reversal* (or *reference reversal*).

We also give a useful amortized result for *beforeFirst* which shows it will not be too expensive over a collection of operations.

1.1 Simplist Specification

☐ **Constructor**

1. *Simplist()*: Creates an empty list with two window positions, before first and after last, and the window over before first.

☐ **Checkers**

2. *isEmpty()*: Returns true if the simplist is empty.
3. *isBeforeFirst()*: True if the window is over the before first position.
4. *isAfterLast()*: True if the window is over the after last position.

☐ **Manipulators**

5. *beforeFirst()*: Initialises the window to be the before first position.
6. *next()*: Throws an exception if the window is over the after last position, otherwise the window is moved to the next position.
7. *previous()*: Throws an exception if the window is over the before first position, otherwise the window is moved to the previous position.

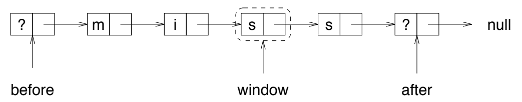
8. *insertAfter(e)*: Throws an exception if the window is over the after last position, otherwise an extra element *e* is added to the simplist after the window position.
9. *insertBefore(e)*: Throws an exception if the window is over the before first position, otherwise an extra element *e* is added to the simplist before the window position.
10. *examine()*: Throws an exception if the window is over the before first or after last positions, otherwise returns the value of the element under the window.
11. *replace(e)*: Throws an exception if the window is over the before first or after last positions, otherwise replaces the element under the window with *e* and returns the replaced element.
12. *delete()*: Throws an exception if the window is over the before first or after last positions, otherwise the element under the window is removed and returned, and the window is moved to the following position.

1.2 Singly Linked Representation

Again block and doubly linked versions are possible — same advantages/disadvantages as the List ADT. Our aim is to show an improvement in the singly linked representation.

Since the window position is not passed as an argument, we need to store it in the data structure. . .

```
public class SimplistLinked {

  private Link before;
  private Link after;
  private Link window;
```



1.3 Reference (or "Pointer") Reversal

The window starts at *before first* and can move up and down the list using *next* and *previous*.

Problem

As for the singly linked representation, *previous* can be found by link coupling, but this takes linear time.

Solution

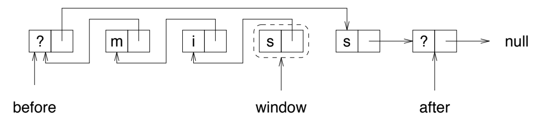
Q: What do you always do when you walk into a labyrinth?

Solution...

- point successor fields behind you backwards
- point successor fields in front of you forwards

Problem: window cell can only point one way.

Solution: the before first successor no longer needs to reference the first element of the list (we can always follow the references back). Instead, use it to reference the cell after the window, and point the window cell backwards.



$\Rightarrow$ *reference (pointer) reversal*

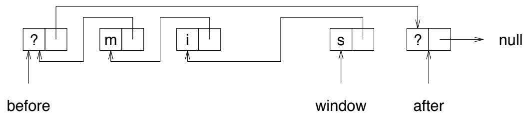
Exercise

```
public void previous() {
  if (!isBeforeFirst) {






  }
  else throw
    new OutOfBounds("calling previous before start of list");
}
```

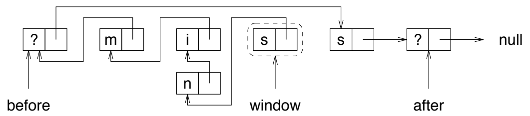
What is the performance of *previous*?

Other operations also require reference reversal.

delete. . .



insertBefore. . .



Disadvantage(?): A little more complex to code.

Advantage: Doesn't require the extra space overheads of a doubly linked list.

Advantage outweighs disadvantage — you only code once; might use many times!

Problem: These operations only reverse one or two references, but what about *beforeFirst*? Must reverse references back to the beginning. (Note that *previous* and *next* now modify the list *structure*.)

$\Rightarrow$ linear in worst case

What about amortized case?. . .

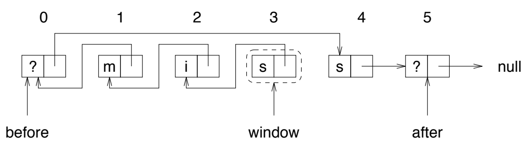
1.4 Amortized Analysis

Consider the operation of the window prior to any call to *beforeFirst* (other than the first one).

Must have started at the before first position after last call to *beforeFirst*.

Can only have moved forward by calls to *next* and *insertBefore*.

If window is over the *i*th cell (numbering from 0 at before first), there must have been i calls to *next* and *insertBefore*. Each is constant time, say 1 “unit”.



beforeFirst requires i constant time "operations" (reversal of i pointers) — takes i time "units".

Total time: $2i$. Total number of operations: $i + 1$.

Average time per operation: ≈ 2

Average time over a sequence of operations is (roughly) constant!

Formally: *Each sequence of n operations takes $O(n)$ time; ie each operation takes constant time in the amortized case.*

1.5 Performance Comparisons — Simplist

Operation	Block	Singly linked	Doubly linked
Simplist	1	1	1
isEmpty	1	1	1
isBeforeFirst	1	1	1
isAfterLast	1	1	1
beforeFirst	1	1^a	1
next	1	1	1
previous	1	1	1
insertAfter	n	1	1
insertBefore	n	1	1
examine	1	1	1
replace	1	1	1
delete	n	1	1

a — amortized bound

Complexity Examples: Insertion sort

For simple programs, we can directly calculate the number of basic operations that will be performed:

procedure INSERTION-SORT(A)
1 **for** $j \leftarrow 2$ **to** *length*$[A]$
2 **do** *key* $\leftarrow A[j]$
3 $i = j - 1$
4 **while** $i > 0$ and $A[i] >$ *key*
5 **do** $A[i+1] \leftarrow A[i]$
6 $i = i - 1$
7 $A[i+1] \leftarrow$ *key*

Lines 2-7 will be executed n times, lines 4-5 will be executed up to j times for j=1 to n.

Insertion Sort complexity

Insertion Sort can be shown to be $O(n^2)$.

A better sorting algorithm (in time)

procedure MERGE-SORT(A, p, r)
  if $p < r$ **then**
    $q \leftarrow \lfloor (p+r)/2 \rfloor$
    MERGE-SORT(A, p, q); MERGE-SORT$(A, q+1, r)$; MERGE(A, p, q, r)

procedure MERGE(A, p, q, r)
  $n_1 \leftarrow q - p + 1$; $n_2 \leftarrow r - q$
  for $i \leftarrow 1$ **to** n_1 **do** $L[i] \leftarrow A[p + i - 1]$
  for $j \leftarrow 1$ **to** n_2 **do** $R[j] \leftarrow A[q + j]$
  $i \leftarrow 1$; $j \leftarrow 1$; $k \leftarrow p$
  while $i \leq n_1$ **and** $j \leq n_2$ **do**
    if $L[i] \leq R[j]$ **then** $A[k++] \leftarrow L[i++]$
    else $A[k++] \leftarrow R[j++]$
  while $i \leq n_1$ **do** $A[k++] \leftarrow L[i++]$
  while $j \leq n_2$ **do** $A[k++] \leftarrow R[j++]$

Merge Sort complexity

Mergesort can be shown to be $O(n \lg n)$

A better sorting algorithm in space

Quicksort has worst case complexity worse than Merge-Sort, but it's average complexity and space usage is *better* than Merge-sort! (CLRS Chapter 7)

procedure QUICKSORT(A, p, r)
  if $p < r$
    then $q \leftarrow$ PARTITION(A, p, r)
      QUICKSORT$(A, p, q-1)$; QUICKSORT$(A, q+1, r)$

procedure PARTITION(A, p, r)
  $x \leftarrow A[r]$; $i \leftarrow p - 1$
  for $j \leftarrow p$ **to** $r - 1$
    do if $A[j] \leq x$
      then $i \leftarrow i + 1$
        exchange $A[i] \leftrightarrow A[j]$
  exchange $A[i+1] \leftrightarrow A[r]$
  return $i + 1$

QuickSort complexity

Quicksort can be shown to be $O(n^2)$

2. Summary

Amortized anaylsis allows us the judge the complexity of data structure operations in the context of the entropy they cause.

- A linked multi-pop stack requires takes time $O(n)$ to do a multi-pop, but this operation must be accompanied by n individual push operations.
- The beforeFirst method in Simplist requires time $O(n)$ but this must be accompanied by n individual next operations.