Algorithms and Data Structures

Lists and Arrays

(c) Marcin Sydow

Topics covered by this lecture:

- Linked Lists
  - Singly Linked Lists
  - Doubly Linked Lists
- The Concept of **Abstract Data Structure**
  - Stack
  - Queue
  - Deque
- The Concept of **Amortised Analysis** of Complexity
  - **"Potential function"** method
  - "total cost" and "accounting" methods
  - Examples on Stack with multiPop
- Unbounded Arrays
- Comparison of Various Representations of Sequences

Sequences

Sequences of elements are the most common data structures.

Two kinds of basic operations on sequences:

- absolute access (place indentified by index), fast on arrays
- relative access (place indentified as a successor, predecessor of a given element.), slow on arrays

The simplest implementation of sequences: arrays, support fast (constant time) absolute access, however relative access is very slow on arrays (time $\Theta(n)$) (Assume a sequence has n elements, and assignment and value check are dominating operations)

Operations on ends of sequence

It is important to realise that in many practical applications, the operations on sequence **concern only the ends** of the sequence (e.g. removeFirst, addLast, etc.).

Any "insert" operation on array has pessimistic linear time (slow).

Thus, some other than arrays data structures can be more efficient for implementing them.

Linked Lists

Alternative implementation that supports **fast relative access** operations like:

- return/remove first/last element
- insert/remove an element after/before given element
- insert a list after/before an element
- isEmpty, size, etc.

Linked list consists of nodes that are linked.

- singly linked lists
- doubly linked lists
- cyclic lists (singly or doubly linked), etc.

Nodes contain:

- one element of sequence
- link(s) to other node(s) (implemented as pointers).

Singly Linked Lists


```
Class SLNode<Type>{
 Type element
 SLNode<Type> next //pointer to the next node or NULL if last
}

Class SList<Type>{
 SLNode<Type> head //points to the first element, the only access to lis
}
```

`head-> (2)-> (3)-> (5)-> (8)-> null`

Last element points to `null` (in empty list head points to `null`)

Example: printing the contents of a list:

```
print(SList l){
 node = l.head
 while(node not null)
  print node.element
  node = node.next
}
```

Double-Linked Lists


```
Class DLNode<Type>{
 Type element
 DLNode<Type> next
 DLNode<Type> prev
}

Class DLList<Type>{
 DLNode<Type> head //points to the first element, the only access to lis
}
```

Double links cost twice memory compared to singly-linked, but are useful when navigation in both directions is needed (e.g. insertionSort)

Cyclic Lists and Cyclic Arrays

In **cyclic list** variant the last node is linked to the first one

It can concern singly or doubly linked lists

In doubly linked case the following "invariant" holds for each node:
(next.prev) == (prev.next) == this

In some cases **cyclic arrays** are also useful. (an array of size n, `first` and `last` are kept to point to the ends of the sequence and they move "modulo n")

Operations on lists

Examples:

- isEmpty
- first
- last
- insertAfter (insertBefore)
- moveAfter (moveBefore)
- removeAfter (removeBefore)
- pushBack (pushFront)
- popBack (popFront)
- concat
- splice
- size
- findNext

Implementation of operations on linked lists

Algorithms and Data Structures

(c) Marcin Sydow

Introduction

Linked Lists

Abstract Data Structure

Stack, Queue, Deque

Amortised Analysis

Unbounded Arrays

Summary

Most modifier list operations can be implemented with a "technical" general operation `splice`.

INPUT: a,b,t - pointers to nodes into list; a anb b are in the same list, t is not between a and b

OUTPUT: cut out sublist (a,...,b) and insert it after t

Example implementation of splice in doubly linked lists:

(notice its **constant time complexity**, even if it concerns arbitrarily large subsequences!)

```
Splice(a,b,t){
 // cut out (a,...,b):
 a' = a.prev; b' = b.next; a'.next = b'; b'.prev = a'
 // insert (a,...,b) after t:
 t'= t.next;  b.next = t'; a.prev = t; t.next = a; t'.prev
}
```

Example: moveAfter(a,b){ splice(a,a,b)}

Extensions

Examples of additional cheap and useful attributes to the linked list structures:

- size (updated in constant time after each operation (except inter-list splice)
- last (useful in singly-linked lists, e.g. for fast pushBack)

Linked lists vs arrays

Linked Lists (compared with arrays):

- positive: fast "relative" operations, like "insert after", etc. (most of them in constant time!)
- positive: unbounded size
- negative: additional memory for pointers
- negative: slow (linear time) absolute access (vs fast (constant) in arrays)

Remarks:
Pointer size can be small compared to the element size, though.
Arrays have **bounded** size.

Abstract Data Structures

Abstract Data Structure

A very general and important concept ADS is defined by operations which can be executed on it
(or, in other words, by its **interface**).

ADS is not defined by the implementation (however, implementation matters in terms of time and space complexity of the operations).

Abstract Data Structure can be opposed to **"concrete"** data structure (as array or linked list)

The most basic examples of ADS: stack and queue.

Stack

The most basic example of abstract data structure.
It is defined by the following interface:

Stack (of elements of type T):

- push(T)
- T pop() (modifier)
- T top() (non-modifier)

(also called: "LIFO" data structure (last in - first out)

Applications of stack: undo, function calls, back button in web browser, parsing, etc.

Queue

(of elements of type T):

- inject(T)
- T out() (modifier)
- T front() (non-modifier)

FIFO (first in - first out)

Applications of queue: music files on iTune list, shared printer, network buffer, etc.

Deque

Double **E**nded **Que**ue (pronounced like “deck”). (of elements of type T):

- T first()
- T last()
- pushFront(T)
- pushBack(T)
- popFront()
- popBack()

Can be viewed as a generalisation of both stack and queue

Examples

Abstract Data Structure can be implemented in many ways (and using various "concrete" data structures), for example:

ADS	fast implementation* possible with:
Stack	SList, Array (how?)
Queue	SList, CArray (how?),(why not with Array?)
Deque	DList, CArray (how?), (why not with SList?, why not wi

* all the operations in constant time

Amortised Complexity Analysis

Amortised Complexity Analysis

Data structures are usually used in algorithms. A typical usage of a data structure is a **sequence of m operation calls** $s = (o_1, o_2, ..., o_m)$ on it.

Denote the cost of the operation o_i by t_i (for $1 \leq i \leq m$). Usually, the **total cost of the sequence of operations** $t = \sum_{1 \leq i \leq m} t_i$ is more important in analysis than the costs of separate operations.

Sometimes, it may be difficult to exactly compute t (total cost), especially if some operations are cheap and some expensive (we do not know the sequence in advance)

In this case, an approach of amortised analysis may be useful. Each operation o_i is assigned an amortised cost a_i so that:

$t = O(\sum_{1 \leq i \leq m} a_i)$ (i.e. t is upper bounded by the sum of amortised costs)

Methods for Amortised Analysis

The most general method for computing the amortised cost of a sequence of operations on a datastructure is the **method of "potential function"** (a non-negative function that has value dependent on the current state of the data structure under study).

Some less general (and possibly simpler) methods can be derived from the "potential method":

- "total cost" method (we compute the total cost of m operations)
- "accounting" method (objects in data structure are assigned "credits" to "pay" for further operations in the sequence)

Potential Function Method

After each operation o_i assign a **potential function** Φ_i to the state i of the data structure, so that $\Phi_0 == 0$, and it is always non-negative.

define $a_i = t_i + \Phi_i - \Phi_{i-1}$

thus we have:
$\sum_{1 \leq i \leq m} a_i = \sum_{1 \leq i \leq m}(t_i + \Phi_i - \Phi_{i-1}) = \sum_{1 \leq i \leq m} t_i + (\Phi_m - \Phi_0)$

thus, $t = \sum_{1 \leq i \leq m} t_i \leq \sum_{1 \leq i \leq m} a_i$

Example

Consider an abstract data structure `StackM` that supports additional `multiPop(int k)` operation[1]. Assume it is implemented in a standard way with a bounded array (of sufficiently large size).

- push(T e), pop(): real cost is O(1)
- multiPop(int k): real cost is O(k)

Question:

What is the pessimistic cost of any sequence of m operations from the above 2-element set of operations on **initially empty** stack?

[1] multiPop(k) is equivalent to applying standard stack's pop operation but k times in a row

Example: "potential" method on stackM

Define the potential function in our example as the current **size of the stack**:

$\Phi(stackM) = sizeOf(stackM)$

thus $amortisedCost(push) = 1 + 1 = 2$,
$amortisedCost(pop) = 1 - 1 = 0$,
$amortisedCost(multiPop(k)) = k + (-k) = 0$

Thus, m operations of push, pop or multiPop on initially empty stack, have total amortised cost $\leq 2m = O(m)$, so that amortised cost of each operation is constant $(O(m)/m)$.

Example, cont.: Total cost method

The problem is that the cost of multiPop(k) depends on the number of elements currently on the stack.

The real cost of multiPop(k) is min(k,n), which is the number of pop() operations executed.

Each element can be popped only once, so that the total number of pop() operations (also those used inside multiPop) cannot be higher than number of push() operations that is not higher than m. Thus all the operations have constant amortised time.

Example, cont.: Accounting Method

We "pay" for some operations in advance.

Amortised cost of operation is $a_i = t_i + credit_i$

Put a coin on each element pushed to the stack. (that is cost of push is: 1 (real cost) + 1 (credit))

Then, because the real cost of any pop() is 1, we always have enough "money" for paying any other sequence of operations

Indexable growing sequences

Consider an abstract data structure that supports:

- [.] (indexing)
- push(T element) (add an element to the end of sequence)

And additionally does not have a limit on size.

How to implement it with amortised constant time complexity of both operations?

Dynamically Growing Arrays

If full, **allocate 2 times bigger** and copy.

Now consider a sequence of n push operations (indexing has constant cost)

What is the pessimistic cost of push?

Dynamically Growing Arrays

If full, **allocate 2 times bigger** and copy.

Now consider a sequence of n push operations (indexing has constant cost)

What is the pessimistic cost of push?

What is amortised cost of push?

Dynamically Growing Arrays

If full, **allocate 2 times bigger** and copy.

Now consider a sequence of n push operations (indexing has constant cost)

What is the pessimistic cost of push?

What is amortised cost of push?(lets use the "global cost" method)

$t_i == i$ if $i - 1$ is a power of 2 (else $t_i == 1$)

$\sum_{i=1}^{n} t_i \leq n + \sum_{j=0}^{\lfloor lg(n) \rfloor} 2^j < n + 2n = 3n$

Thus, the total cost of n operations is bounded by $3n$ so that amoritsed cost is $3n/n = O(1)$

Dynamically Growing Arrays

If full, **allocate 2 times bigger** and copy.

Now consider a sequence of n push operations (indexing has constant cost)

What is the pessimistic cost of push?

What is amortised cost of push?(lets use the "global cost" method)

$t_i == i$ if $i - 1$ is a power of 2 (else $t_i == 1$)

$\sum_{i=1}^{n} t_i \leq n + \sum_{j=0}^{\lfloor lg(n) \rfloor} 2^j < n + 2n = 3n$

Thus, the total cost of n operations is bounded by $3n$ so that amoritsed cost is $3n/n = O(1)$

Exercise: What happens if the array grows by constant number

Example: Analysis of Growing Arrays, cont.

We can also use "accounting" method.

Each push "pays" 3 units to account: 1 for putting it, 1 for potential copying it in future, 1 for potential future copying of one of the previous "half" of elements already in the array. After each re-allocate, the credit is 0.

We can also use the potential method: $\Phi_i = 2n - w$ (where n is the current number of elements and w is the current size)

Dynamically Growing and Shrinking Arrays

Now, assume we want to extend the interface:

- [.] (indexing)
- push(T element) (add an element to the end of sequence)
- popBack() (take the last element in the sequence)

And wish that if there is “too much” unused space in the array it automatically **shrinks**

Unbounded Arrays

An unbound array u containing currently n elements, is emulated with w-element ($w \geq n$) static bounded array b with the following approach:

- first n positions of b are used to keep the elements, last $w - n$ are not used
- if n reaches w, a larger (say $\alpha = 2$ times larger) bounded array b' is allocated and elements copied to b'
- if n is to small (say $\beta = 4$ times smaller than w) a smaller (say $\alpha = 2$ smaller) array b' is reallocated and elements copied to b'

What is the (worst?,average?) time complexity of: index, pushBack, popBack in such implementation?

Example of Amortised Analysis on UArrays

pushBack and popBack on unbounded array with n elements have either O(1) (constant) or O(n) (linear) cost, depending on current size w of underlying bounded array b.

Lemma. Any sequence of m operations on (initially empty) unbounded array (with $\alpha = 2$ and $\beta = 4$) has $O(m)$ total cost, i.e. the amortised cost of operations of unbounded array is constant ($O(m)/m$).

Corollary. pushBack and popBack operations on unbounded array have amortised constant time complexity.

Exercise*: Prove the Lemma. Hint: define the potential $\Phi(u) = max(3n - w, w/2)$ and use the potential method

Exercise: Show that if $\beta = \alpha = 2$, it is possible to construct a sequence of m operations that have $O(m^2)$ total cost.

Comparison of complexity of sequence operations

Operation	SList	DList	UArray	CArray	meaning of '
[.]	n	n	1	1	
size	1'	1'	1	1	without external sp
first	1	1	1	1	
last	1	1	1	1	
insert	1	1'	n	n	only insertAfte
remove	1	1'	n	n	only removeAft
pushBack	1	1	1'	1'	amortised
pushFront	1	1	n	1'	amortised
popBack	n	1	1'	1'	amortised
popFront	1	1	n	1'	amortised
concat	1	1	n	n	
splice	1	1	n	n	
findNext	n	n	n'	n'	cache-efficient

(all the values are surrounded by $O()$, n is the number of elements in the

Summary

- Linked Lists
  - Singly Linked Lists
  - Doubly Linked Lists
- The Concept of Abstract Data Structure
  - Stack
  - Queue
  - Deque
- The Concept of Amortised Complexity
  - "Potential function" method
  - "total cost" and "accounting" methods
  - Examples on Stack with multiPop
- Unbounded Arrays
- Comparison of Various Representations of Sequences

Thank you for your attention