Dynamic Data Structures

Dr Timothy Kimber

Dynamic Data Structures

Having efficient data structures is crucial for successful algorithms.

- The problems seen so far involved fixed length lists
- In m
  — ar
- Our

Other problems require dynamic data structures suc

- Lists, Stacks and Queues
- Sets and Dictionaries

These are designed to hold variable, essentially unlimited amounts of data.

Ordered Data Structures

A *list* is an ordered collection of {nodes, items, elements}.

- The key property of a list is the ordering of the nodes
- A list might support operations such as

unshift	adds an element to the front of the list
insert	adds an element at a given position
remove	removes the element at a given position
iterate	returns the items in order

- Plus sorting, searching, copying, joining, splitting ...
- The most appropriate implementation depends on which operations are needed.

Stacks

A *stack* is a last-in first-out (LIFO) list.

- Stacks support only

  push for adding elements

  pop for removing elements

- Sta

- Stacks support recursive algorithms including fundamental operations such as calling subprocedures and evaluating arithmetic expressions

Stacks

Question

How woul

- Must be able to add “unlimited” objects
- Push and Pop must implement LIFO behaviour

Performance of Push

Question

Given a stack containing N objects, what is the worst case time complexity of push?

- Ass c
- Ass

Performance of Push

Revised Question

Given an empty stack, what is the worst case time to push N objects?

- Assume: initial capacity is 4
- Ass

Amortisation

The time for N pushes is $O(N)$, so:

- A single push is *effectively* a constant time operation
- Mo
- NO

Amortis

- Related to accountancy method used to defer lar
- Amortised analysis considers a sequence of ope
- Cost of individual ops is “amortised” across the sequence
- Unlike accountancy, must never be in debt

Amortised Analysis

Rather than calculating cost of full sequence of N steps we can

- Pick a representative subsequence
- Sub
- Pic
- Show that paying amortised cost covers all costs (never in debt)

Exercise

Find a representative cycle (subsequence) of pushes
show that the amortised cost of $3c$ covers all costs.

Amortised Analysis

Begin cycle when ...

End cycle when ...

Amortised Analysis

Argument only works because array is initially empty and size is doubled

- Say we have N objects on stack after a copy
- Before next copy we always push N more
- Thi

- Multiplying by any factor will do - will affect amortisation constant

Queues

A *queue* is also a list, but the next object removed is either

- The earliest one added (FIFO Queue)
- The

Question

- How could you implement a priority queue (PQ)
- Given a PQ following your design that contains would be the worst case time to add a new object? (E a key attribute that determines its priority.)

Priority Queue Design

Heap: a Tree in an Array

We want to know where the "end" of the tree is:

- Build a tree within an array

- Tra
- Navigate by indices
- Leaving a[0] blank means:
  - parent of a[n] is a[n/2]
  - children of a[n] are a[2*n] and a[2*n+1]

Exercise

How should a new object be added to a max binary heap? (i.e. the greatest key should be at the root).

Heap: a Tree in an Array

- Tra
- Leaving a[0] blank means:
  - parent of a[n] is a[n/2]
  - children of a[n] are a[2*n] and a

Exercise

How should the object with the greatest key be removed from a max binary heap?

Binary Heap Performance

Question

Given a heap containing N objects, what is the time complexity for adding or removing one object?

Heapsort

Heaps also provide us with the Heapsort algorithm (JWJ Williams, 1964)

Heapsort (given a list L)

- Cre
- Re
- Re
- HALT

- What could be simpler?!
- Performance is again $\Theta(N log_2 N)$
- Can also be implemented in place by setting up list and heap partitions within a single array

Sets

A *set* is an unordered collection of objects each having a unique key.

- Should have "unlimited" capacity
- Wa
- A ke

Questions

- How could you implement a set?
- Given a set following your design that contains ould be the worst case time to get the object with key k?

A Search Tree?

A tree will divide the data but need a different ordering

- Start at the root (it's a tree)
- Go right: find/add larger keys
- Go left: find/add smaller keys

Binary Search Tree

In a Binary Search Tree

- Go right: find/add larger keys
- Go le

Exercise

- Dra

```
bst = new BST
keys = [5,3,10,1,6,9,8,0,4]
for i = 0 to 8
  bst.put(keys[i])
```


- What is the worst case time complexity of the put procedure?

Red-Black Trees

Red-Black Trees are binary search trees that maintain balance

- A BST can become (very) unbalanced, resulting in long branches
- Searching a BST takes $O(N)$ time in the worst case
- The

Red-Black Tree Properties

Definition (Red-Black Tree)

A binary search tree T is a red-black tree iff T satisfies the following five propertie

1. All n
2. The root node is black
3. Every leaf (all null) is black
4. Both children of a red node are black
5. All paths from a node to a descendant leaf contain the same number of black nodes

Insertion

A node is inserted using the ordinary BST procedure

- A new node is always colored red

Insertion

The insertion may result in a violation of the red-black tree properties

- The root might be coloured red
- A red node might have a red child

Insertion

Either recolour $\Theta(1)$ nodes

- There is still a red node with a red parent
- The problem has moved closer to the root (continue)

Insertion

Or perform a rotation of $\Theta(1)$ nodes and Stop

- Reduces height of the tree
- Preserves key ordering

Insertion

The properties are restored

Performance

By maintaining the red-black tree properties, we have $h \leq 2log_2(N+1)$

- Get p
- Hei

For Put, only the last part is different

- The extra work is still localised to one branch
- So, Put also runs in $O(log_2 N)$ time