Thinking Points

What is a data structure?

Why am I happy saying that insertion into a dynamic array is $O(1)$ when resizing is $O(n)$?

Consider the scenario:
I am in charge of an expensive machine at CERN. Each experiment is defined by 12 boolean flags. I don't want to repeat an experiment so I need to store the experiments I have already done and check them when considering a new experiment.

- What abstract operations on my data do I need?
- What is the most efficient way to store and access this data?

Lecture 8
Advanced Data Structures Part 1
Abstract Data Types and Hash Tables

COMP26120

Giles Reger

November 2019

Learning Objectives and Assessment

By the end of this section on data structures you should be able to:

- Explain the role of abstract data types (ADTs)
- Describe key ADTs such as vectors, lists, stacks, queues, sets, dictionaries, and priority queues
- Describe key data structures such as linked lists, dynamic arrays, hash tables, binary search trees, AVL trees, binary heaps and skip lists, including the main operations and their complexities
- Recall standard definitions about such datastructures
- Compute standard operations on such data structures
- Analyse a scenario and identify the most appropriate data structure based on the scenarios requirements

Reminder: January exam is online with 50% on complexity part and 50% on data structures. C programming is not examinable. See Blackboard quizzes for example of kinds of question and come to Revision Lecture.

Abstract Data Types and Data Structures

Abstract Data Type (ADT)

Abstractly defines the desired behaviour in terms of the supported operations and their effects. Java Interfaces are analogous to ADTs. Usually behaviour described informally but sometimes mathematically. May include expectations about complexity but rarely.

Data Structure

Concretely describes how data values are stored and operated on. Can be viewed as implementing one or more ADTs.

Key Abstract Data Types

In this course we will focus on the following key ADTs:

- Vector
- List
- Stack
- Queue
- Set
- Dictionary (a.k.a. Associative Array or Map)
- Priority Queue
- Tree
- (Graph, next semester)

For my purposes, these will be defined for reference in these slides. These are generally simpler than you may find elsewhere (e.g. in the course textbook) - you can usually assume extra operations.

Vector ADT

Let the rank of an element e in a sequence S be the number of elements that are before e in S.

A vector S storing n elements supports the following fundamental operations:

`get(r)`: Return the element of S with rank r.

`insert(r,e)`: Insert a new element e into S to have rank r

`remove(r)`: Remove from S the element at rank r

Each will have an error condition if $r < 0$ or $r > n - 1$.

Vectors usually support methods such as `size()` and `isEmpty()`

List ADT

A position is a simple ADT that has a single `element()` operation

A list S supports the following fundamental operations:

`first()`: Returns the position of the first element of S (error if S is empty)

`last()`: Returns the position of the last element of S (error if S is empty)

`before(p)`: Returns position of the element of S preceding the one at p (error if p is first position)

`after(p)`: Returns position of the element of S following the one at p (error if p is last position)

`insertBefore(p,e)`: Insert new element e into S before position p

`insertAfter(p,e)`: Insert new element e into S after position p

`remove(p)`: Remove from S the element at position p

Stack ADT

A stack S supports the following fundamental operations:

`push(o)`: Insert object o at the top of the stack

`pop()`: Remove and return the top object (the most recently inserted element that has not been popped)

(error if S is empty)

`peek()`: Return the top object in S without removing it

(error if S is empty)

Usually also support `size()` and `isEmpty()` operations.

Queue ADT

A queue S supports the following fundamental operations:

`enqueue(o)`: Insert object o at the rear of S

`dequeue()`: Remove and return the front object of S
(error if S is empty)

`peek()`: Return the front object in S without removing it
(error if S is empty)

Usually also support `size()` and `isEmpty()` operations.

Set ADT

A set S supports the following fundamental operations:

`add(o)`:	Insert object o into the set
`remove(o)`:	Remove object o from the set
`find(o)`:	Returns true if the set contains o and false otherwise

Usually also support `size()` and `isEmpty()` operations.

Dictionary ADT

Also known as an associative array or map.

A dictionary D supports the following fundamental operations:

`find(k)`: If D contains an item with key equal to k then return the value of that item, otherwise return special NULL item

`insert(k,v)`: Insert an item with value v and key k into D

`remove(k)`: Remove any item from D that has a key equal to k

Priority Queue ADT

A priority queue P supports the following fundamental operations:

`insert(k,e)`: Insert an element e with key k into P

`removeMin()`: Remove and return from P the element with the *smallest* key (may not be unique)

(error if P is empty)

Usually also support `size()` and `isEmpty()` operations.

Implementing a Dictionary or Priority Queue

So far we have met two data structures:

- Linked lists; and
- Dynamic arrays

We also assume you have some familiarity with trees from first year.

How might we use linked lists to implement a priority queue or dictionary? What about a dynamic array?

What will the complexity of the resulting operations be?

Is this the best we can do?

Further Data Structures

In the remainder we will explore:

- Hash Table (also Hash Set)
- Binary Search Tree / Ordered Binary Tree
- AVL Tree
- Binary Heap
- Skip List

In the labs you will get a chance to implement all of these!

Example

Hash Maps

Three main components:

1. Bucket array - an array A of size N where each cell contains a structure that initially holds a special EMPTY value
2. Hashing function h - from the sort of keys to the range 0 to $(N-1)$
3. Collision resolution strategy!

In the simple case:

- `insert(k,v)` puts a cell containing k and v at $A[h(k)]$
- `find(k)` looks in $A[h(k)]$ and returns the value if the keys match

But sadly it's not quite that simple...

Collisions

If 2,450 keys are hashed into a million buckets, even with a perfectly uniform random distribution, according to the birthday problem there is approximately a 95% chance of at least two of the keys being hashed to the same slot.

A collision occurs if there are two keys k and k' in the dictionary such that $h(k) = h(k')$

Collisions are unavoidable for any realistic sort of keys. We just need to handle them with a collision resolution strategy.

Hash Functions

We usually split the hash function into

- Computing the hash code of the key as an integer, and
- Mapping the hash code into the range of A via a compression map

The standard compression map is the division method e.g.

$$h(k) = |k| \bmod N$$

where the bucket array is of size N.

We need to be careful about the value of N with this. A bad choice can preserve patterns in key distribution. Ideally N is prime.

An alternative is the MAD method (see textbook).

Computing a Hash Code

A good hash code

- depends on all parts of the key, and
- is sensitive to their order

and necessarily hashes equal keys to the same hash code.

Let us assume a tuple key $x_0, \ldots, x_k$

We could try summing components e.g. $\sum_{i=0}^{k} x_i$

However, this is insensitive to order. A better approach is to use a polynomial

$$x_0 a^k + x_1 a^{k-1} + \ldots + x_{k-1} a + x_k$$

for a nonzero constant $a \neq 1$ - what is a good value for a?

Separate Chaining

This is the simplest solution to collision resolution

Each bucket stores a reference to a list that stores all items that hash to that bucket

Insertion simply appends the new item to this list; remove removes from it

Find must search the list

With a good hash function we expect each bucket to contain $\frac{n}{N}$ items where n are the number of items in the hash table and N is its capacity.

Separate Chaining Pseudo Code

```
insert(k,v):
  if A[h(k)] is empty
    Insert empty list into A[h(k)]
  Insert (k,v) into A[h(k)]


find(k):
  if A[h(k)] is empty
    Return NULL
  else
    For each (k',v) in A[h(k)]
      if k=k'
        Return v
    Return NULL
```

Open Addressing

An disadvantage of separate chaining is that we need to update lists and it takes more space

An alternative is to have a scheme for looking at alternative locations during each operation. The simplest is Linear Probing which scans through the bucket array to find an empty place.

However, this can cause clustering (what?). To handle this we can apply

- Quadratic Probing which iteratively tries $A[(i + j^2) \bmod N]$ for $j = 0, 1, 2, \ldots$ and $i = h(k)$
- Double Hashing which uses another hash function h' to iteratively try $A[(i + (j \cdot h'(k)))]$ for $j = 0, 1, 2, \ldots$, $i = h(k)$, and key k

Question: How do we deal with removal in open addressing?
Question: Will we always be able to insert an item?

Linear Probing Pseudo Code

```
insert(k,v):
    For j=0 to N
      if A[(h(k)+j)%N] is empty
        Insert (k,v) into A[(h(k)+j)%N]
        Return SUCCESS
    Return FAILURE


find(k):
  For j=0 to N
    if A[(h(k)+j)%N] is empty
      Return NULL
    Let (k',v) = A[(h(k)+j)%N]
    if k=k'
      Return v
  Return NULL
```

Rehashing

If a hash table becomes too full then both collision resolution approaches suffer.

The load factor of a hash table is $\frac{k}{N}$ items where k are the number of items in the hash table and N is its capacity.

It is a good idea to keep the load factor of a hash table below a certain constant (certainly below 1) by occasionally rehashing the table e.g. making it larger and reinserting its contents

Choices and Trade-Offs

Within the general Hash Map setting there are still choices to be made.

The choice of hash function (including the constant used in the polynomial hash code) may depend on the setting and the sort of the key. The cost of the operations may also be a consideration e.g. one might take 8 (deterministic) random characters from a string.

The choice of load factor may depend on the distribution of keys in your application and the collision resolution strategy chosen.

Such values can be set by experimentation. People have already done lots of experiments to suggest some good values.

Complexities

	Worst	Average
Space	O(n)	O(n)
Insert	O(n)	O(1)
Find	O(n)	O(1)
Remove	O(n)	O(1)

What assumptions are we making in the average case?

- Hashing is reasonably uniform (average $\frac{k}{N}$ per bucket)
- We ensure that the load factor is less than some constant (< 1)

Summary

Abstract Data Types (ADTs) describe the desired behaviour

Data Structures define the implementation

First identify the ADT we need, then consider which data structure to use - which operations are important, what are the complexity trade-offs?

Hash tables need

- A good hash function, and
- A good collision detection strategy

Should I use a hash table for implementing a Priority Queue?

Next time - Trees