Lecture 8

Algorithms & Data Structures

Goldsmiths Computing

November 26, 2018

Outline

Outline

Lecture

- Collections
  - dynamic array, linked list…
  - … stack, queue, (binary tree)…
- Hash tables
  - constant-time access…
  - … if the data you have is kind to you
  - not an ordered collection.
- Finding cycles in linked lists

VLE activities

Recurrence relations quiz

Statistics so far:

- 204 attempts: average mark 4.30
- 72 students: average mark 4.60
  - 32 under 4.00, 23 over 6.99, 6 at 10.00

Quiz closes at 16:00 on Friday 30th November

- no extensions
- grade is
  - 0 (for no attempt)
  - $30 + 70 \times (\text{score}/10)^2$

VLE activities (cont'd)

List visualiser

Submissions so far: 26

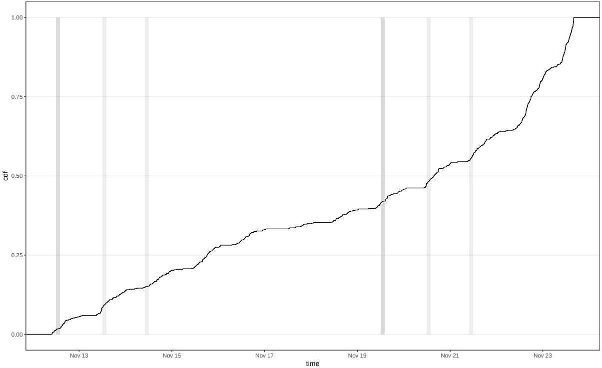
Submissions close at 16:00 on Friday 30th November

- absolutely no extensions
- grade is
  - 0 and barred from participation in assessment phase (for no submission)
  - 30 + score

VLE activities (cont'd)

Dynamic arrays quiz

- 604 attempts: average mark 5.23
- 130 students: average mark 7.30
    - 14 under 4.00, 84 above 6.99, 28 at 10



Outline

Motivation

We need (in general) the ability to insert an object whose reduced hash code is the same as that of an object already in the hash table.

Definition

A collision resolution strategy says what to do if a collision is found. Routines for insert, find and delete must all agree on the collision resolution strategy.

Separate chaining

aka: “Open hashing”, “Closed addressing”: each hash-table slot contains a *list* of values.

closed addressing
: the location in the hash table is always the place implied by the hash code

open hashing
: the object isn’t directly stored in the hash table

function FIND(o,h)
  i ← HASH(o,h)
  return FIND(o,h.table[i])
end function

Separate chaining

aka: “Open hashing”, “Closed addressing”: each hash-table slot contains a *list* of values.

closed addressing
: the location in the hash table is always the place implied by the hash code

open hashing
: the object isn't directly stored in the hash table

function FIND(o,h)
    i ← HASH(o,h)
    return FIND(o,h.table[i])
end function

function INSERT(o,h)
    i ← HASH(o,h)
    h.table[i] ← CONS(o,h.table[i])
end function

Separate chaining

aka: "Open hashing", "Closed addressing": each hash-table slot contains a *list* of values.

closed addressing
: the location in the hash table is always the place implied by the hash code

open hashing
: the object isn't directly stored in the hash table

```
function FIND(o,h)
    i ← HASH(o,h)
    return FIND(o,h.table[i])
end function
function INSERT(o,h)
    i ← HASH(o,h)
    h.table[i] ← CONS(o,h.table[i])
end function
function DELETE(o,h)
    i ← HASH(o,h)
    return DELETE(o,h.table[i])
end function
```

Open addressing

aka: "Closed hashing": if there's a collision, probe for a empty slot somewhere else

open addressing
: find a different location in the hash table than that implied by the hash code

closed hashing
: always directly store in the hash table

Open addressing

aka: “Closed hashing”: if there’s a collision, probe for a empty slot somewhere else

open addressing
: find a different location in the hash table than that implied by the hash code

closed hashing
: always directly store in the hash table

find

```
function FIND(o,h)
    i ← HASH(o,h); k ← 0
    repeat
        j ← PROBE(i,k,h); k ← k + 1
        if h.table[j] = o then
            return true
        end if
    until EMPTY?(h.table[j])
    return false
end function
```

Open addressing

aka: “Closed hashing”: if there’s a collision, probe for a empty slot somewhere else

open addressing
: find a different location in the hash table than that implied by the hash code

closed hashing
: always directly store in the hash table

insert

function INSERT(o,h)
  i ← HASH(o,h); k ← 0
  repeat
    j ← PROBE(i,k,h); k ← k + 1
  until FREE?(h.table[j])
  h.table[j] ← o
end function

Open addressing

Linear probing

If there's a collision, probe by looking at the next slot.

function PROBE(i,k,h)
  return (i + k) mod SIZE(h.table)
end function

- good locality of reference
- simple to implement

but

- similar hash codes lead to secondary collisions

Open addressing

Why FREE?, not EMPTY?, in INSERT?

Open addressing

Why FREE?, not EMPTY?, in INSERT?

Delete

```
function DELETE(o,h)
    i ← HASH(o,h); k ← 0
    repeat
        j ← PROBE(i,k,h); k ← k + 1
    until h.table[j] = o
    h.table[j] ← †
end function
```

Open addressing

Why FREE?, not EMPTY?, in INSERT?

Delete

```
function DELETE(o,h)
    i ← HASH(o,h); k ← 0
    repeat
        j ← PROBE(i,k,h); k ← k + 1
    until h.table[j] = o
    h.table[j] ← †
end function
function FREE?(value)
    return EMPTY?(value) ∨ value = †
end function
```

Open addressing

Quadratic probing

If there's a collision, probe by looking at slots square numbers away.

function PROBE(i,k,h)
  return (i - (-1)^k × k²) mod SIZE(h.table)
end function

- reduced primary clustering
- preserves some locality of reference
- SIZE(h.table) must be a prime number

Open addressing

Complexity of hash-table operations:

- no collisions: $\Theta(1)$
- everything collides: $\Theta(N)$
- usual case: somewhere in between

Improve the usual case:

- decrease probe length

Robin Hood linear probing

While inserting:

- if you find a value that is less far from its natural space
- steal that space and insert the value you've displaced instead

Extending and rehashing

Too many collisions?

1. make a bigger table
2. re-insert all the current contents
    - different reduction will mean different, fewer collisions

Work

1. Reading
    - CLRS, section 11.4
    - Drozdek, section 10.2
    - Pedro Celis, “Robin Hood Hashing”
2. Exercises

    CLRS 11.2-2, 11.4-1
3. Lab work (week of Monday 26th November)

Outline

Introduction

Hash table collision resolution

Characters

Strings

String matching

Motivation

In order to represent natural language, we need to be able to divide it up and represent individual components of text.

Definitions

grapheme cluster	roughly, a letter
grapheme	smallest meaningful unit in writing in a given language
symbol	individual member of an alphabet
code point	numeric value assigned to some kind of text unit

Definitions

grapheme cluster roughly, a letter

grapheme smallest meaningful unit in writing in a given language

symbol individual member of an alphabet

code point numeric value assigned to some kind of text unit

character highly context-dependent meaning: could be any of the above

Properties

numeric does the character represent some kind of number? 0, 3

Properties

numeric does the character represent some kind of number? 0, 3, Ⅹ

Properties

numeric	does the character represent some kind of number? 0, 3, X
lowercase	is the character lowercase? a, z

Properties

numeric	does the character represent some kind of number? 0, 3, Ⅹ
lowercase	is the character lowercase? a, z
uppercase	is the character uppercase? A, Z

Properties

numeric does the character represent some kind of number? 0, 3, X

lowercase is the character lowercase? a, z

uppercase is the character uppercase? A, Z, Dz

Properties

numeric	does the character represent some kind of number? 0, 3, Ⅹ
lowercase	is the character lowercase? a, z
uppercase	is the character uppercase? A, Z, ǅ
whitespace	is the character whitespace?

Character repertoires

ASCII

128 code points

- 10 digits
- 26 lowercase letters
- 26 uppercase letters
- 1 whitespace
- 32 punctuation
- 33 control-codes

Character repertoires

ASCII

128 code points

- 10 digits
- 26 lowercase letters
- 26 uppercase letters
- 1 whitespace
- 32 punctuation
- 33 control-codes

Characters in common use in USA

examples 5, e, Z, &, $

Character repertoires

Latin-1

256 code point superset of ASCII: includes everything there and:

- 32 lowercase letters
- 30 uppercase letters
- 1 whitespace
- 33 punctuation
- 32 control-codes

Character repertoires

Latin-1

256 code point superset of ASCII: includes everything there and:

- 32 lowercase letters
- 30 uppercase letters
- 1 whitespace
- 33 punctuation
- 32 control-codes

Adds characters useful in Western European languages

examples é, Ç, ÷, £

(but not €)

Character repertoires

Unicode

1114112 code points

- code points [0,1114111]
- (some code points do not correspond directly to characters)

Aims to standardise all human languages and text (*e.g.* Greek, Cyrillic, Arabic, Hebrew, Hangul, Ethiopic, Mongolian, Mathematical operators, Braille, CJK, mediaeval Latin)

examples Θ, Щ, ,א ,⌧ ⌧, ⌧, ⌧, ⌧, ⌧, ⌧, ⌧

(Klingon and Tengwar out of scope)

Combining characters

- e-acute: U+00E9, é
- a-acute: U+00E1, á
- z-acute: U+017A, ź
- v-acute:

Combining characters

- e-acute: U+00E9, é
- a-acute: U+00E1, á
- z-acute: U+017A, ź
- v-acute: U+0076 U+0301, v́

Some characters (grapheme clusters) have multiple representations:

- o-acute: U+00F3 ó or U+006F 0+0301 ó

Work

1. Reading
   - Unicode FAQ: Basic Questions
   - Marcus Kuhn, UTF-8 and Unicode FAQ

Outline

Motivation

Most language text is linear, so it makes sense to be able to store text in a linear collection, which we call strings.

Definition

A string is a linear collection specialized to hold characters.
(but what meaning of "character"? Usually code point)

Implementation

For now:

- as dynamic array of code points

  C++ `std::string`

Implementation

For now:

- as dynamic array of code points

  C++ `std::string`

- as vector of code points

  C `char []`

Implementation

For now:

- as dynamic array of code points

  C++ `std::string`

- as vector of code points

  C `char []`

- as *immutable* vector of code points

  Java `java.lang.String`

Implementation

For now:

- as dynamic array of code points

  C++ `std::string`

- as vector of code points

  C `char []`

- as *immutable* vector of code points

  Java `java.lang.String`

but beware:

- these data structures might not be optimal for the job
- there are many more exotic implementations and representations out there

Operations

Linear collection operations:

length	return how many characters are in the string
get[i]	return the character at position i in the string
find[c]	is the character c in the string?
position[c]	what position is the character c at?

Operations

Linear collection operations:

length	return how many characters are in the string
get[i]	return the character at position i in the string
find[c]	is the character c in the string?
position[c]	what position is the character c at?

Mutable collection operations:

push	add a character at the end (C++ only)

Operations

Linear collection operations:

length	return how many characters are in the string
get[i]	return the character at position i in the string
find[c]	is the character c in the string?
position[c]	what position is the character c at?

Mutable collection operations:

push	add a character at the end (C++ only)

String operations:

match[s]	is the string s contained in the string?

Outline

Motivation

- generalisation of search operation (sequences, not just single elements)
- applications include text editors, classifiers, information retrieval systems
- extensions used in
  - spelling checkers
  - DNA sequence matching
  - protein structure representations

Definition

String matching returns the smallest index at which the *pattern*, P, is found exactly in the *text*, T, or false if the pattern is not present in the text at all.

C++ `std::string::find()`

Java `java.lang.String.indexOf()`

String matching algorithm

```
function MATCH(T,P)
    m ← LENGTH(P)
    for 0 ≤ s ≤ LENGTH(T) - m do
        if T[s...s+m] = P[0...m] then
            return s
        end if
    end for
    return false
end function
```

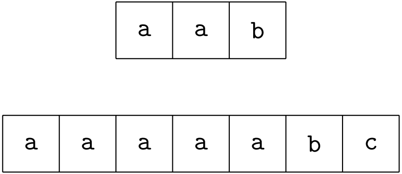
Naïve algorithm

```
function MATCH(T,P)
    m ← LENGTH(P)
    for 0 ≤ s ≤ LENGTH(T) - m do
        found ← true
        for 0 ≤ j < m do
            if T[s+j] ≠ P[j] then
                found ← false; break
            end if
        end for
        if found then
            return s
        end if
    end for
    return false
end function
```

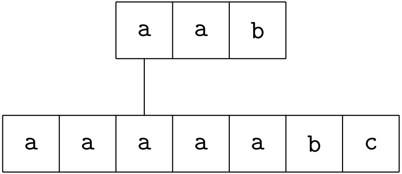
Diagram

a	a	b

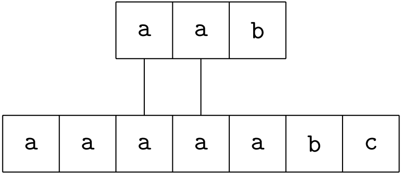
Diagram



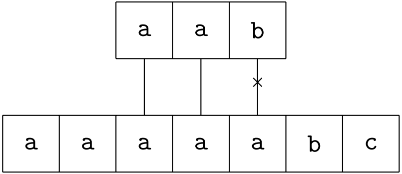
Diagram



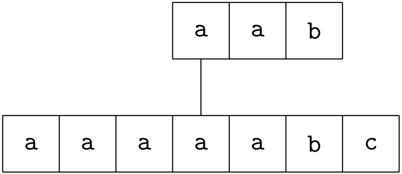
Diagram



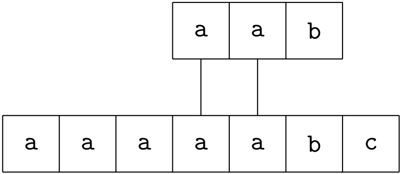
Diagram



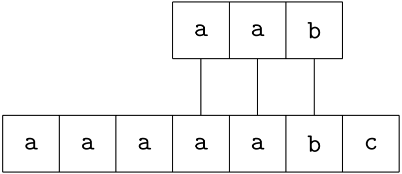
Diagram



Diagram



Diagram



Complexity analysis

space

- no particular requirements for additional storage

$$\implies \Theta(1)$$

time

- outer loop happens $n - m + 1$ times (worst case)
- inner loop m times (worst case)

$$\implies \Theta((n+1)m - m^2) \sim \Theta(nm)$$

For particular sizes of pattern:

small	$m \sim c \implies \Theta(n)$
large	$m \sim n \implies \Theta(n)$
intermediate	$m \sim \frac{n}{2} \implies \Theta(n^2)$

Work

1. Reading
   - CLRS, section 32.1
   - Drozdek, section 13.1.1 “Straightforward Algorithms”
2. Questions from CLRS

   Exercises 32.1-1, 32.1-2
3. Lab work
   - (week of 3rd December) implement naïve string match for strings of characters. Use `OpCounter` (remember that?) to count how many character comparisons happen in the worst case. Construct a table and verify the theoretical results in this lecture.