Lecture 11

Algorithms & Data Structures

Goldsmiths Computing

January 14, 2019

Outline

Introduction

Implicit data structures

Multidimensional arrays

Binary search

Outline

Introduction

Implicit data structures

Multidimensional arrays

Binary search

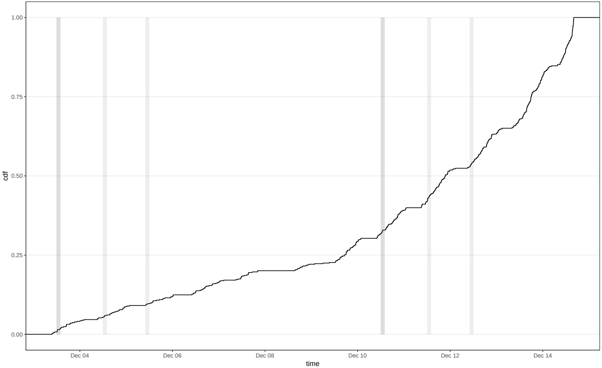
Lecture

1. Binary trees
2. Heaps

VLE activities

Binary trees quiz

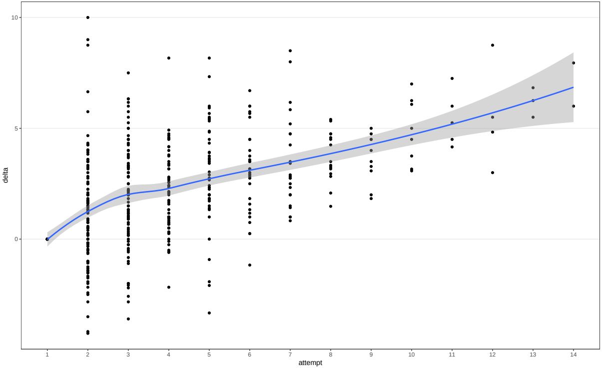
- 538 attempts: average mark 5.55
- 126 students: average mark 7.65
    - 8 under 4.00, 86 above 6.99, 29 at 10



VLE activities

Binary trees quiz

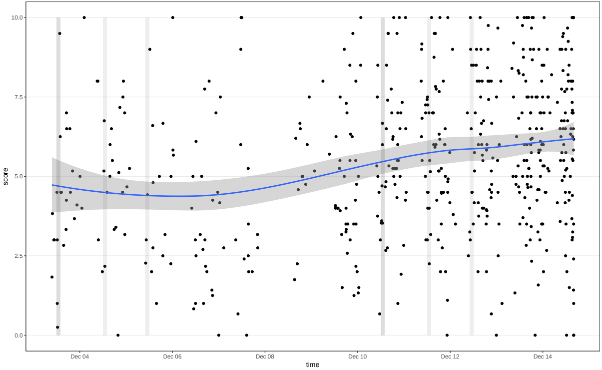
- 538 attempts: average mark 5.55
- 126 students: average mark 7.65
  - 8 under 4.00, 86 above 6.99, 29 at 10



VLE activities

Binary trees quiz

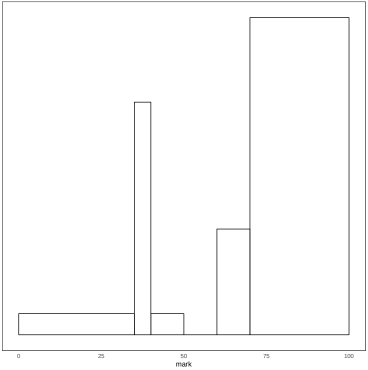
- 538 attempts: average mark 5.55
- 126 students: average mark 7.65
    - 8 under 4.00, 86 above 6.99, 29 at 10



VLE activities (cont'd)

Hash tables submission

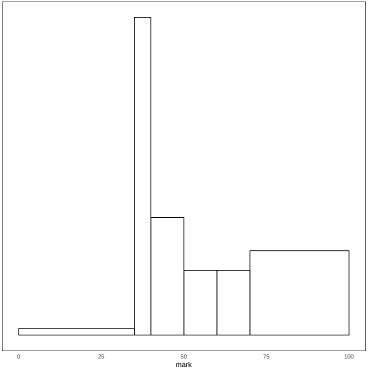
- 120 final uploads: average mark 80.03



VLE activities (cont'd)

String matching submission

- 116 final uploads: average mark 60.89



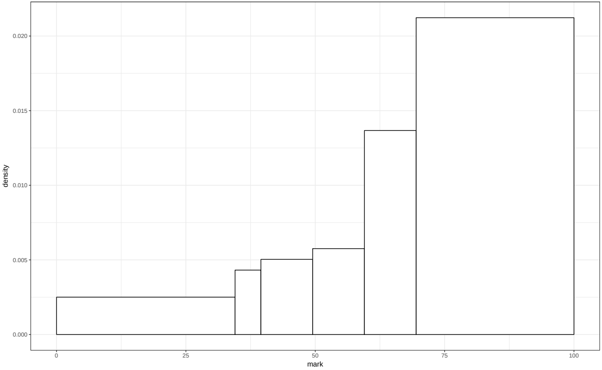
VLE activities (cont'd)

Module evaluation

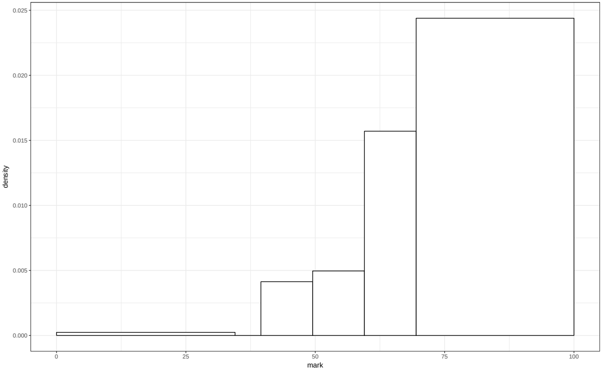
Module evaluation is open at this link (also from module page on learn.gold)

- answers held anonymously
- (also for your other first-term modules!)

Term 1 summary



Term 1 summary



Outline

Introduction

Implicit data structures

Multidimensional arrays

Binary search

Motivation

Pointers in data structures can be wasteful of space and cause inefficiencies on modern architectures. Encoding relationships (e.g. parent, left-child) between elements using storage location can help.

Motivation

Pointers in data structures can be wasteful of space and cause inefficiencies on modern architectures. Encoding relationships (e.g. parent, left-child) between elements using storage location can help.

Pointers/references can also be hard to work with. We're not going to solve *that* problem here.

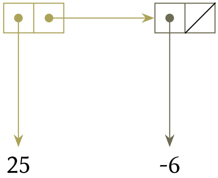
Definition

An implicit data structure is one where the space overhead for encoding the relationship between data contained in the structure is constant, regardless of the number of elements contained in the data structure.

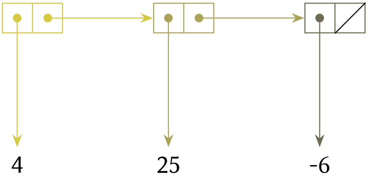
$$S(N) \in \Theta(1)$$

Linked list (review)

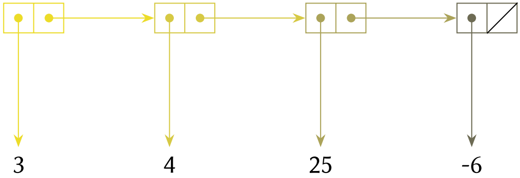
Linked list (review)



Linked list (review)



Linked list (review)



Example: linked list

Space overhead is linear

$$S(N) \in \Theta(N)$$

Example: linked list

Implement as a pair of static array and counter (A,c):

first return A[c]

Example: linked list

Implement as a pair of static array and counter (A,c):

first return A[c]

rest return (A,c+1)

Example: linked list

Implement as a pair of static array and counter (A,c):

first	return A[c]
rest	return (A,c+1)
set-first![o]	A[c] $\leftarrow$ o

Example: linked list

Implement as a pair of static array and counter (A,c):

first	return A[c]
rest	return (A,c+1)
set-first![o]	A[c] $\leftarrow$ o
set-rest![l]	?

Work

1. Reading:
   - J. Ian Munro and Hendra Suwanda, *Implicit data structures for fast search and update*, Journal of Computer and System Sciences 21:2, pp.236-250 (1980)

Outline

Introduction

Implicit data structures

Multidimensional arrays

Binary search

Motivation

Sometimes the data that you want to store is naturally expressed as a table with more than one dimension.

Definition

A multidimensional array is an array that is subscipted using more than one index

NB: the “multidimensional” in multidimensional arrays refers to the subscripting, not the data that is stored:

linear array of 3-component colours Vector

linear array of 3-dimensional vectors Vector

Definition

A multidimensional array is an array that is subscipted using more than one index
NB: the “multidimensional” in multidimensional arrays refers to the subscripting, not the data that is stored:

linear array of 3-component colours Vector

linear array of 3-dimensional vectors Vector

2d array of grayscale values Multidimensional array

3d array of temperature values Multidimensional array

Operations

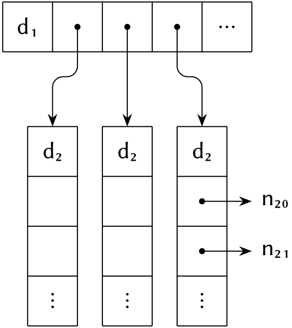
size return the number of elements in the multidimensional array

select[k,m,...,n] return the element at position k in the first dimension, m in the second, ..., and n in the last dimension

store![o,k,m,...,n] set the element at position k in the first dimension, m in the second, ..., and n in the last dimension to o.

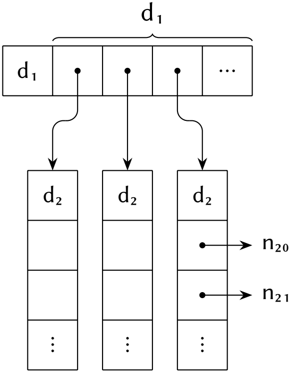
Implementation: Iliffe vector

Array of references to lower-dimensional arrays:



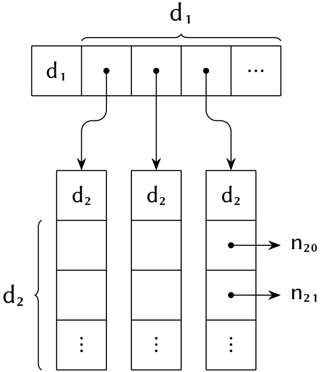
Implementation: Iliffe vector

Array of references to lower-dimensional arrays:



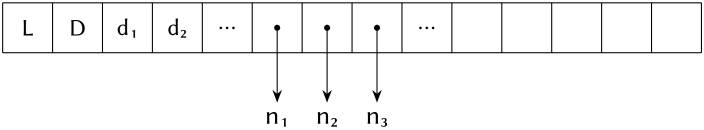
Implementation: Iliffe vector

Array of references to lower-dimensional arrays:



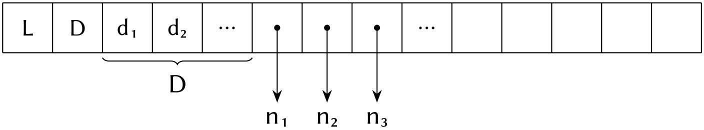
Implementation: dope vector

One-dimensional array with extra metadata (the “dope” on the array):



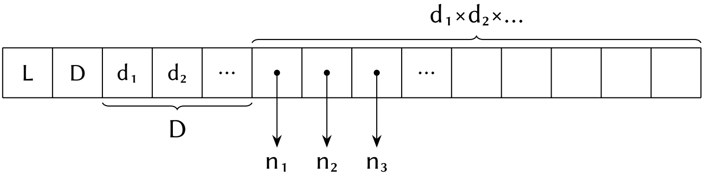
Implementation: dope vector

One-dimensional array with extra metadata (the “dope” on the array):



Implementation: dope vector

One-dimensional array with extra metadata (the “dope” on the array):



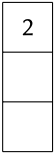
Implementation: example

Storing the 2×4 matrix

$$\begin{pmatrix} 1 & 2 & 4 & 8 \\ 2 & 3 & 5 & 7 \end{pmatrix}$$

Iliffe vector

Row-major ordering (compare earlier Iliffe vector diagram)



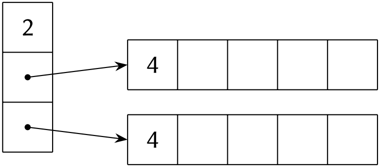
Implementation: example

Storing the 2×4 matrix

$$\begin{pmatrix} 1 & 2 & 4 & 8 \\ 2 & 3 & 5 & 7 \end{pmatrix}$$

Iliffe vector

Row-major ordering (compare earlier Iliffe vector diagram)



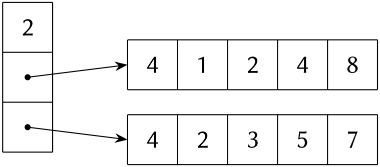
Implementation: example

Storing the 2×4 matrix

$$\begin{pmatrix} 1 & 2 & 4 & 8 \\ 2 & 3 & 5 & 7 \end{pmatrix}$$

Iliffe vector

Row-major ordering (compare earlier Iliffe vector diagram)



Implementation: example

Storing the 2×4 matrix

$$\begin{pmatrix} 1 & 2 & 4 & 8 \\ 2 & 3 & 5 & 7 \end{pmatrix}$$

dope vector

Row-major ordering

11											

Implementation: example

Storing the 2×4 matrix

$$\begin{pmatrix} 1 & 2 & 4 & 8 \\ 2 & 3 & 5 & 7 \end{pmatrix}$$

dope vector

Row-major ordering

11	2										

Implementation: example

Storing the 2×4 matrix

$$\begin{pmatrix} 1 & 2 & 4 & 8 \\ 2 & 3 & 5 & 7 \end{pmatrix}$$

dope vector

Row-major ordering

11	2	2	4								

Implementation: example

Storing the 2×4 matrix

$$\begin{pmatrix} 1 & 2 & 4 & 8 \\ 2 & 3 & 5 & 7 \end{pmatrix}$$

dope vector

Row-major ordering

11	2	2	4	1	2	4	8	2	3	5	7

Implementation: example

Storing the 2×4 matrix

$$\begin{pmatrix} 1 & 2 & 4 & 8 \\ 2 & 3 & 5 & 7 \end{pmatrix}$$

dope vector

Column-major ordering

11	2	2	4	1	2	2	3	4	5	8	7

Size

Iliffe vector

Require: A :: two-dimensional (Illife) array
  function SIZE(A)
    return LENGTH(A) × LENGTH(A[0])
  end function

Size

Iliffe vector

Require: A :: two-dimensional (Illife) array
  function SIZE(A)
    return LENGTH(A) × LENGTH(A[0])
  end function

dope vector

Require: A :: multidimensional (dope) array
  function SIZE(A)
    D ⟵ A[0]
    result ⟵ 1
    for $0 \le d < D$ **do**
      result ⟵ result × A[1+d]
    end for
    return result
  end function

Select

Iliffe vector

Require: A :: multidimensional (Iliffe) array
Require: ks :: list of indices

```
function SELECT(A,ks)
    if LENGTH(ks) = 1 then
        return A[FIRST(ks)]
    else
        return SELECT(A[FIRST(ks)],REST(ks))
    end if
end function
```

Select

dope vector

Require: A :: multidimensional row-major (dope) array
Require: ks :: tuple of indices

```
function SELECT(A,ks)
    D ← A[0]
    index ← 0
    for 0 ≤ d < D do
        index ← index × A[1+d] + ks[d]
    end for
    return A[1+D+index]
end function
```

Complexity analysis

time

Operations take time proportional to the number of dimensions D, but independent of the size of each dimension. For given D, all operations (size, select, store!) take time in $\Theta(1)$.

Complexity analysis

time

Operations take time proportional to the number of dimensions D, but independent of the size of each dimension. For given D, all operations (size, select, store!) take time in $\Theta(1)$.

space

Iliffe vector space overhead proportional to the size of each dimension (worst case, space overhead in $\Theta(N)$)

dope vector space overhead proportional to the number of dimensions (for a given dimension, space overhead in $\Theta(1)$)

Multidimensional array (with dope vector) is an example of an implicit data structure

Outline

Introduction

Implicit data structures

Multidimensional arrays

Binary search

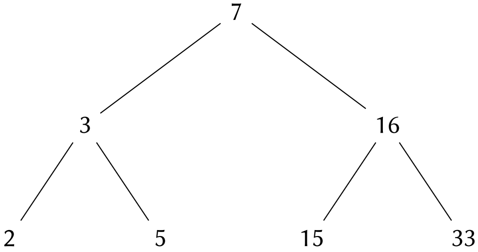
Motivation

- simple, efficient search algorithm
- one or two intersting practical lessons

Definition

Given a suitable data structure, binary search is a search algorithm for an item within that structure that can exclude half of the search space with a single comparison.

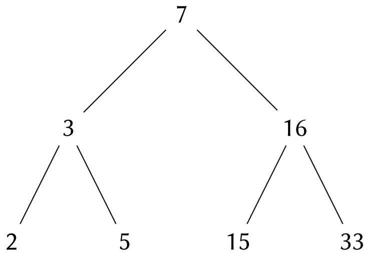
Tree representation



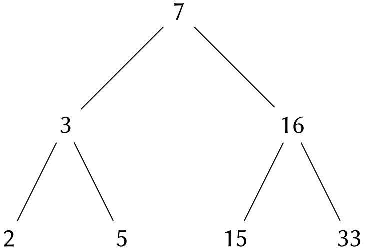
Binary search on trees

```
function BINARY-SEARCH(tree,k)
    if tree = NIL then
        return false
    else if tree.key = k then
        return true
    else if k < tree.key then
        return BINARY-SEARCH(tree.left,k)
    else
        return BINARY-SEARCH(tree.right,k)
    end if
end function
```

Sorted array (implicit tree) representation



Sorted array (implicit tree) representation



2	3	5	7	15	16	33

Binary search on sorted arrays

```
function BINARY-SEARCH(A,lo,hi,k)
    mid ← ⌊(lo+hi−1)/2⌋
    if lo = hi then
        return false
    else if A[mid] = k then
        return true
    else if k < A[mid] then
        return BINARY-SEARCH(A,lo,mid,k)
    else
        return BINARY-SEARCH(A,mid+1,hi,k)
    end if
end function
```

Complexity analysis

Recurrence relationship

$$T(N) = T\left(\frac{N}{2}\right) + 1$$

Recursion tree

$$T(N)$$

Complexity analysis

Recurrence relationship

$$T(N) = T\left(\frac{N}{2}\right) + 1$$

Recursion tree

$$\begin{array}{c} 1 \\ | \\ T\left(\frac{N}{2}\right) \end{array}$$

Complexity analysis

Recurrence relationship

$$T(N) = T\left(\frac{N}{2}\right) + 1$$

Recursion tree

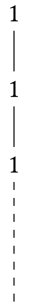
$$\begin{array}{c} 1 \\ | \\ 1 \\ | \\ T\left(\frac{N}{4}\right) \end{array}$$

Complexity analysis

Recurrence relationship

$$T(N) = T\left(\frac{N}{2}\right) + 1$$

Recursion tree



Complexity analysis

Recurrence relationship

$$T(N) = T\left(\frac{N}{2}\right) + 1$$

Recursion tree

$$\left.\begin{array}{c}1\\ |\\ 1\\ |\\ 1\\ \vdots\end{array}\right\}\log_2 N$$

Complexity analysis

Recurrence relationship

$$T(N) = T\left(\frac{N}{2}\right) + 1$$

Master theorem

$$T(N) = aT\left(\frac{N}{b}\right) + f(n)$$

- a = 1; b = 2; $f(n) \in \Theta(1) = \Theta(n^0)$ so c = 0
- $\log_b a = 0$ = c so case 2

$$\implies \Theta(\log N)$$

Work

1. as written in these slides, the algorithm binary search on sorted arrays contains a trap for the unwary: it is mathematically correct, but if translated directly into Java or C++ it would cause problems.
   - Reading: Jon Bentley, *Programming Pearls*, Column 4: Writing Correct Programs
   - Bentley's implementation of binary search in the above column has (at least) one serious bug
2. (week of 21st January) implement binary search (correctly!)