Lecture 6

Algorithms & Data Structures

Goldsmiths Computing

November 12, 2018

Outline

Outline

Lecture

- Growth of functions
  - power law
  - logarithmic
  - exponential
- Recursion

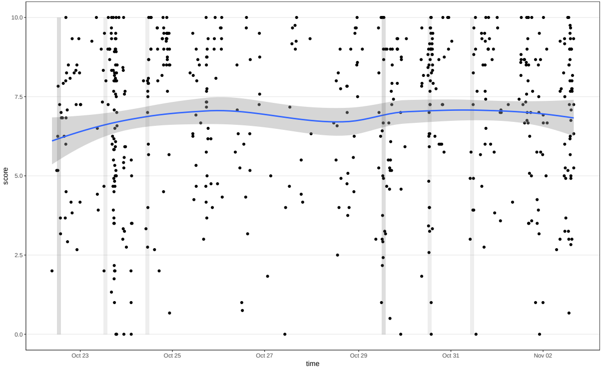
Lab

- Be a data structure implementor
    1. more methods on linked lists

VLE activities

Stacks and queues quiz

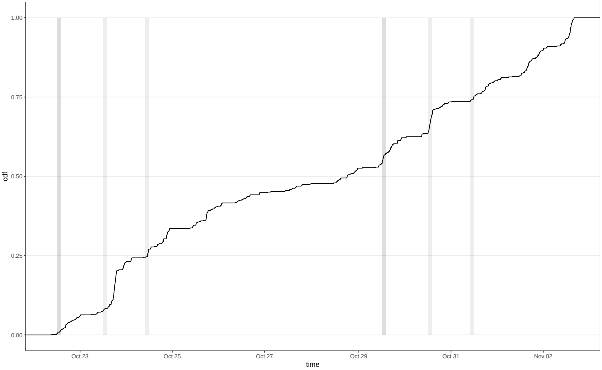
- 584 attempts: average mark 6.89
- 131 students: average mark 8.52
  - 5 under 4.00, 111 above 6.99, 41 at 10



VLE activities

Stacks and queues quiz

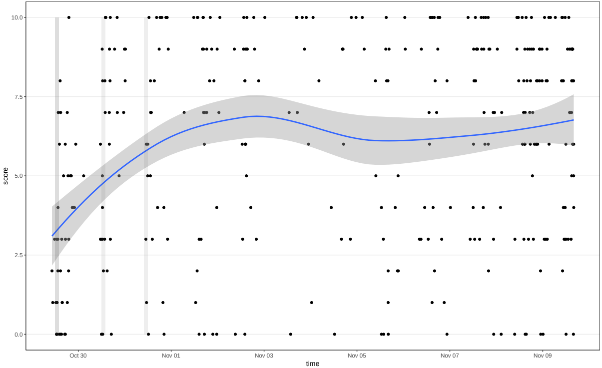
- 584 attempts: average mark 6.89
- 131 students: average mark 8.52
    - 5 under 4.00, 111 above 6.99, 41 at 10



VLE activities (cont'd)

Growth of functions quiz

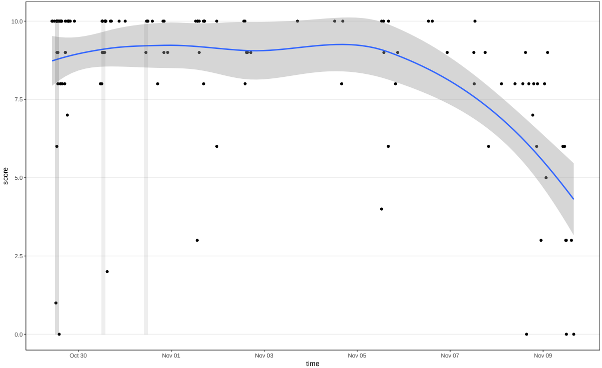
- 358 attempts: average mark 5.99
- 124 students: average mark 8.38
    - 11 under 4.00, 104 above 6.99, 58 at 10



VLE activities (cont'd)

Growth of functions quiz

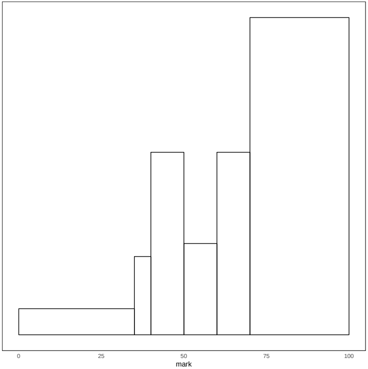
- 358 attempts: average mark 5.99
- 124 students: average mark 8.38
  - 11 under 4.00, 104 above 6.99, 58 at 10



VLE activities (cont'd)

Linked lists submission

- 118 final uploads: average mark 76.14



Outline

Motivation

- simplest form of tree data structure
- algorithms straightforward to understand
  - and (reasonably) simple to analyse
- generalise to practical applications
  - *e.g.* B-Trees for disk storage

Definition

A binary tree is an ordered collection of data

Operations

left	return the left-child of the tree
right	return the right-child of the tree
key	return the data stored at this node of a tree
parent	return the parent of the node

(and associated setters)

Operations

left return the left-child of the tree

right return the right-child of the tree

key return the data stored at this node of a tree

parent return the parent of the node

(and associated setters)

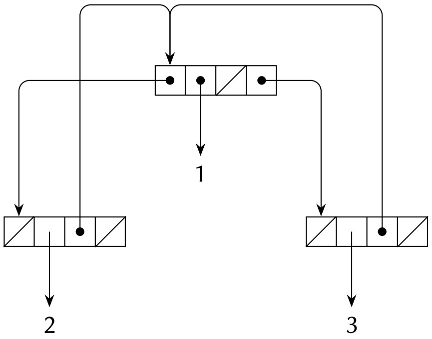
Collection operations

search[o] return `true` if o is in the collection

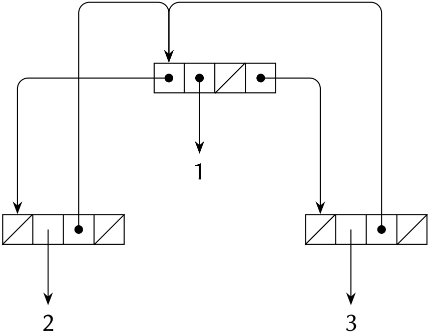
max return the maximum element (with respect to some ordering) of the collection

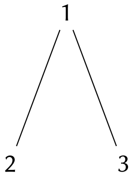
...

Implementation



Implementation





Complexity analysis

left, right, key, parent

single pointer reads (or writes for setters)

$$\Longrightarrow \Theta(1)$$

Traversal

vector start at index zero, and visit elements in order of index until you reach the end

dynamic array as vector

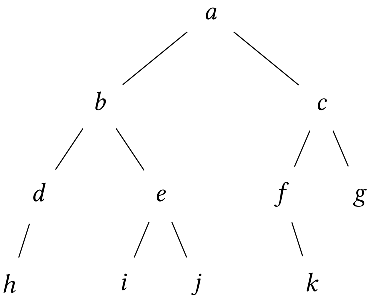
linked list start at the head of the list, and visit the FIRST of each successive REST

binary tree multiple possibilities!

Depth-first traversal

pre-order

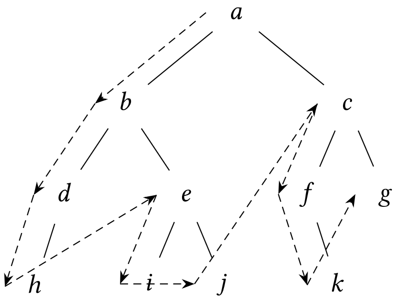
```
function PRE-ORDER(T)
    if ¬NULL?(T) then
        VISIT(T)
        PRE-ORDER(LEFT(T))
        PRE-ORDER(RIGHT(T))
    end if
end function
```



Depth-first traversal

pre-order

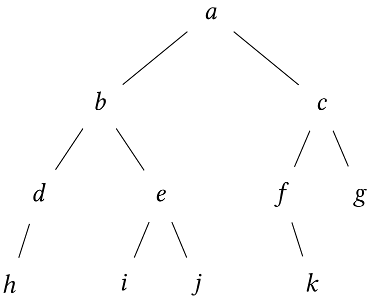
function PRE-ORDER(T)
    if ¬NULL?(T) **then**
        VISIT(T)
        PRE-ORDER(LEFT(T))
        PRE-ORDER(RIGHT(T))
    end if
end function



Depth-first traversal

post-order

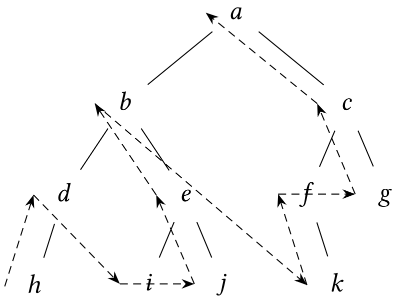
```
function POST-ORDER(T)
    if ¬NULL?(T) then
        POST-ORDER(LEFT(T))
        POST-ORDER(RIGHT(T))
        VISIT(T)
    end if
end function
```



Depth-first traversal

post-order

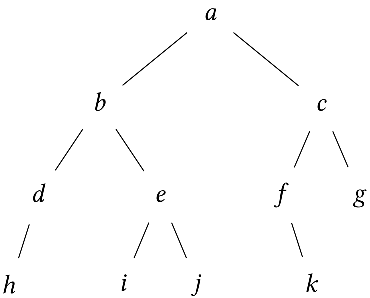
function POST-ORDER(T)
  if ¬NULL?(T) **then**
    POST-ORDER(LEFT(T))
    POST-ORDER(RIGHT(T))
    VISIT(T)
  end if
end function



Depth-first traversal

in-order

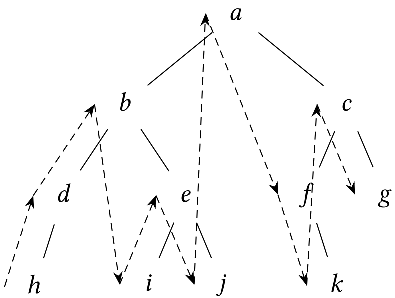
function IN-ORDER(T)
    if ¬NULL?(T) **then**
        IN-ORDER(LEFT(T))
        VISIT(T)
        IN-ORDER(RIGHT(T))
    end if
end function



Depth-first traversal

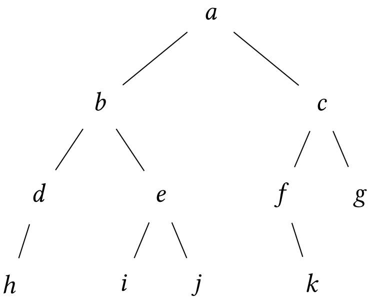
in-order

function IN-ORDER(T)
    if ¬NULL?(T) **then**
        IN-ORDER(LEFT(T))
        VISIT(T)
        IN-ORDER(RIGHT(T))
    end if
end function



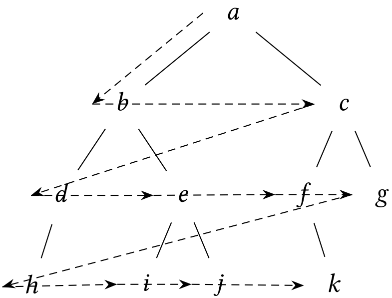
Breadth-first traversal

```
function ENQUEUE-IF!(Q,T)
    if ¬NULL?(T) then
        ENQUEUE!(Q,T)
    end if
end function
function BREADTH-FIRST(T)
    Q ← new Queue()
    ENQUEUE-IF!(Q,T)
    while ¬EMPTY?(Q) do
        t ← DEQUEUE!(Q)
        VISIT(t)
        ENQUEUE-IF!(Q,LEFT(t))
        ENQUEUE-IF!(Q,RIGHT(t))
    end while
end function
```



Breadth-first traversal

function ENQUEUE-IF!(Q,T)
    if ¬NULL?(T) **then**
        ENQUEUE!(Q,T)
    end if
end function
function BREADTH-FIRST(T)
    Q ← new Queue()
    ENQUEUE-IF!(Q,T)
    while ¬EMPTY?(Q) **do**
        t ← DEQUEUE!(Q)
        VISIT(t)
        ENQUEUE-IF!(Q,LEFT(t))
        ENQUEUE-IF!(Q,RIGHT(t))
    end while
end function



Height-balanced property

In a height-balanced tree:

- the heights of left- and right-subtrees of every node differ by at most 1

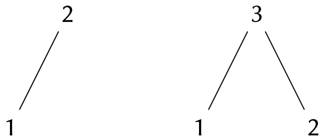
Example height-balanced trees

Height-balanced property

In a height-balanced tree:

- the heights of left- and right-subtrees of every node differ by at most 1

Example height-balanced trees

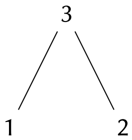


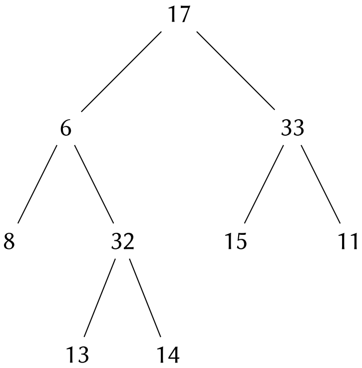
Height-balanced property

In a height-balanced tree:

- the heights of left- and right-subtrees of every node differ by at most 1

Example height-balanced trees



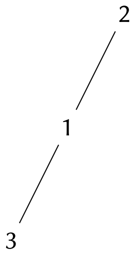


Height-balanced property

In a height-balanced tree:

- the heights of left- and right-subtrees of every node differ by at most 1

Example non-height-balanced trees

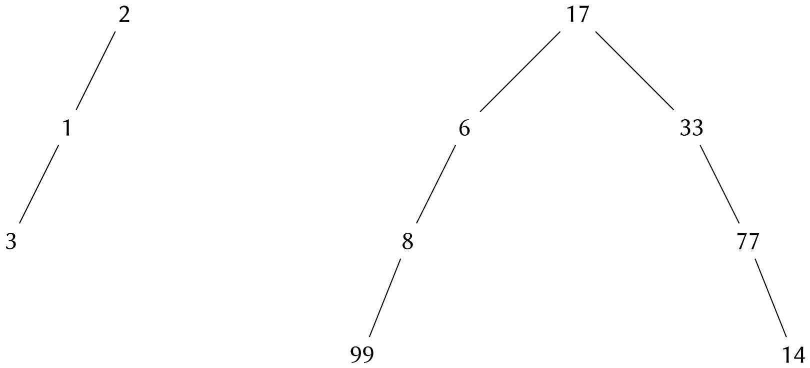


Height-balanced property

In a height-balanced tree:

- the heights of left- and right-subtrees of every node differ by at most 1

Example non-height-balanced trees



Weight-balanced property

In a weight-balanced tree:

- the number of nodes of left- and right-subtrees of every node differ by at most 1

Weight-balanced trees are automatically height-balanced.

Example weight-balanced trees

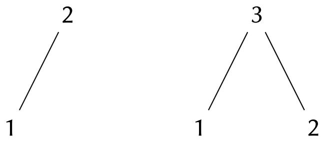
Weight-balanced property

In a weight-balanced tree:

- the number of nodes of left- and right-subtrees of every node differ by at most 1

Weight-balanced trees are automatically height-balanced.

Example weight-balanced trees



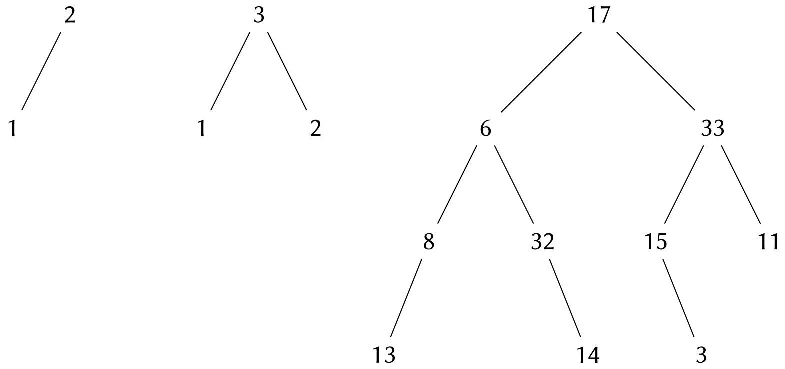
Weight-balanced property

In a weight-balanced tree:

- the number of nodes of left- and right-subtrees of every node differ by at most 1

Weight-balanced trees are automatically height-balanced.

Example weight-balanced trees



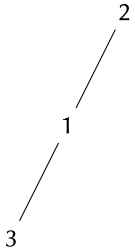
Weight-balanced property

In a weight-balanced tree:

- the number of nodes of left- and right-subtrees of every node differ by at most 1

Weight-balanced trees are automatically height-balanced.

Example non-weight-balanced trees



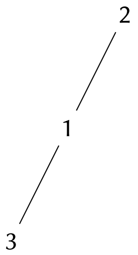
Weight-balanced property

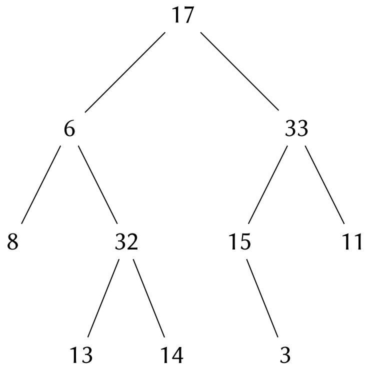
In a weight-balanced tree:

- the number of nodes of left- and right-subtrees of every node differ by at most 1

Weight-balanced trees are automatically height-balanced.

Example non-weight-balanced trees





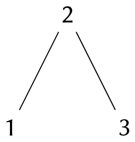
Nearly-complete property

In a nearly-complete tree:

- all levels except *possibly* the lowest level are completely filled;
- the lowest level is filled from the left;
- a complete tree (lowest level filled) is by convention also a nearly-complete tree.

Nearly-complete trees are automatically height-balanced (but not necessarily weight-balanced)

Example nearly complete trees



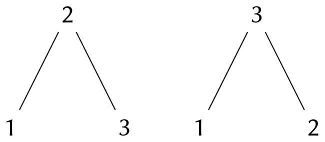
Nearly-complete property

In a nearly-complete tree:

- all levels except *possibly* the lowest level are completely filled;
- the lowest level is filled from the left;
- a complete tree (lowest level filled) is by convention also a nearly-complete tree.

Nearly-complete trees are automatically height-balanced (but not necessarily weight-balanced)

Example nearly complete trees



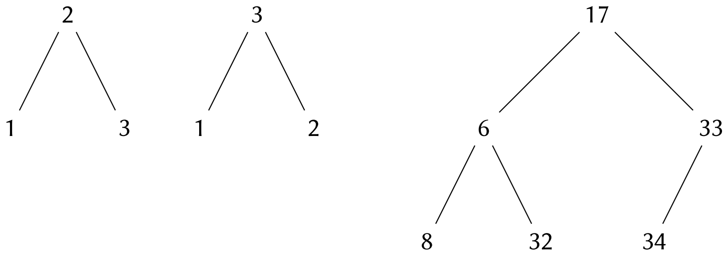
Nearly-complete property

In a nearly-complete tree:

- all levels except *possibly* the lowest level are completely filled;
- the lowest level is filled from the left;
- a complete tree (lowest level filled) is by convention also a nearly-complete tree.

Nearly-complete trees are automatically height-balanced (but not necessarily weight-balanced)

Example nearly complete trees



Nearly-complete property

In a nearly-complete tree:

- all levels except *possibly* the lowest level are completely filled;
- the lowest level is filled from the left;
- a complete tree (lowest level filled) is by convention also a nearly-complete tree.

Nearly-complete trees are automatically height-balanced (but not necessarily weight-balanced)

Example non-nearly complete trees

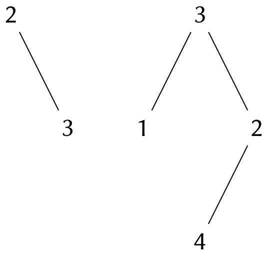
Nearly-complete property

In a nearly-complete tree:

- all levels except *possibly* the lowest level are completely filled;
- the lowest level is filled from the left;
- a complete tree (lowest level filled) is by convention also a nearly-complete tree.

Nearly-complete trees are automatically height-balanced (but not necessarily weight-balanced)

Example non-nearly complete trees



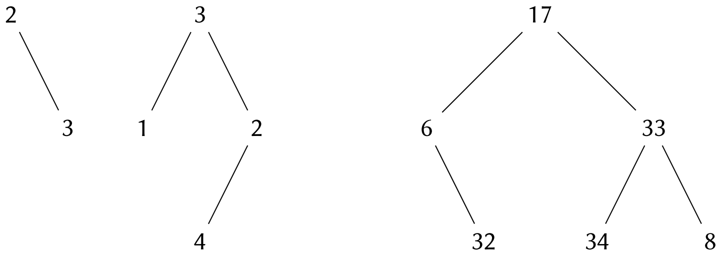
Nearly-complete property

In a nearly-complete tree:

- all levels except *possibly* the lowest level are completely filled;
- the lowest level is filled from the left;
- a complete tree (lowest level filled) is by convention also a nearly-complete tree.

Nearly-complete trees are automatically height-balanced (but not necessarily weight-balanced)

Example non-nearly complete trees

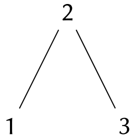


Outline

Binary search tree property

Let x be a node in a binary search tree. If y is a node in the left subtree of x, then $y.key < x.key$. If z is a node in the right subtree of x, then $z.key \geq x.key$.

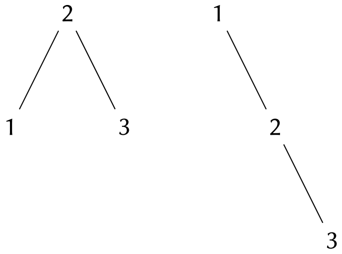
Example binary search trees



Binary search tree property

Let x be a node in a binary search tree. If y is a node in the left subtree of x, then y.key < x.key. If z is a node in the right subtree of x, then z.key ≥ x.key.

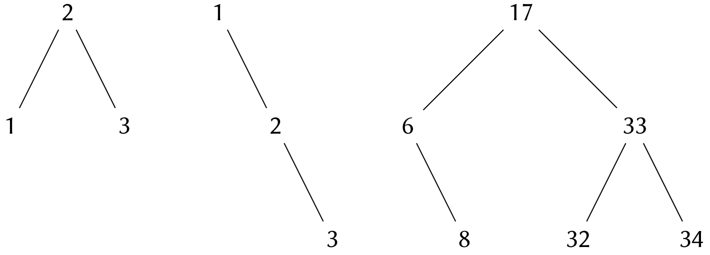
Example binary search trees



Binary search tree property

Let x be a node in a binary search tree. If y is a node in the left subtree of x, then y.key < x.key. If z is a node in the right subtree of x, then z.key ≥ x.key.

Example binary search trees



Find in binary tree

Require: tree :: binary tree
function FIND(tree,object)
  if NULL?(tree) **then**
    return false
  end if
  if tree.key = object **then**
    return true
  end if
  return FIND(tree.left,object) $\vee$ FIND(tree.right,object)
end function

Complexity analysis

Complexity analysis

find

$$T(N) = T(k) + T(N - k - 1) + \Theta(1)$$

$$\implies \Theta(N)$$

Complexity analysis

find

$$T(N) = T(k) + T(N - k - 1) + \Theta(1)$$
$$\implies \Theta(N)$$

max

(as with find: traverse all nodes)

Find in binary search tree

Require: tree :: binary search tree
  function FIND(tree,object)
    if NULL?(tree) **then**
      return false
    end if
    if tree.key = object **then**
      return true
    else if tree.key > object **then**
      return FIND(tree.left,object)
    else
      return FIND(tree.right,object)
    end if
  end function

Complexity analysis

Work in terms of the height h of the tree

find

key to find could in principle be on the lowest level of the tree

$$\Longrightarrow \Theta(h)$$

Complexity analysis

Work in terms of the height h of the tree

find

key to find could in principle be on the lowest level of the tree

$$\implies \Theta(h)$$

max

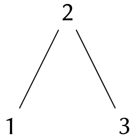
Descend right subtrees as far as possible

$$\implies \Theta(h)$$

Nearly complete binary search trees

A nearly complete binary search tree has both the binary search tree property and the complete property.

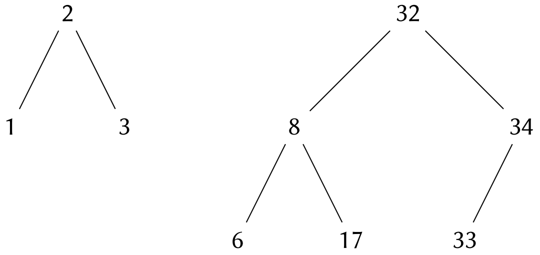
Example nearly complete binary search trees



Nearly complete binary search trees

A nearly complete binary search tree has both the binary search tree property and the complete property.

Example nearly complete binary search trees



Complexity analysis

For a complete binary search tree, h is $\lceil \log(N) \rceil$.

max

traverse just right-nodes

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

Complexity analysis

For a complete binary search tree, h is $\lceil \log(N) \rceil$.

max

traverse just right-nodes

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

find

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

$$\implies ...?$$

Complexity analysis

For a complete binary search tree, h is $\lceil \log(N) \rceil$.

max

traverse just right-nodes

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

find

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

$$\implies \dots?$$

constructor

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

Work

1. Reading
   - CLRS, section 10.4
   - CLRS, chapter 12
2. Questions from CLRS:

   Exercise 10.4-1

   Exercises 12.1-1, 12.2-1, 12.2-2, 12.3-1, 12.3-3

Outline

Motivation

Merge sort: a straightforward, efficient sorting algorithm, with some additional useful properties:

- stability
- genericity (linked lists and vectors)

Definition

To sort a sequence using merge sort: sort two half-length subsequences, then combine the results.

Merge (vector)

Require: a,b :: Vector
  function MERGE(a,b)
    al ← LENGTH(a); bl ← LENGTH(b); cl ← al + bl
    c ← **new** Vector(cl)
    ai ← bi ← ci ← 0
    while ci < cl **do**
      if ai = al **then**
        c[ci] ← b[bi]; bi ← bi + 1
      else if bi = bl ∨ a[ai] ≤ b[bi] **then**
        c[ci] ← a[ai]; ai ← ai + 1
      else
        c[ci] ← b[bi]; bi ← bi + 1
      end if
      ci ← ci + 1
    end while
    return c
  end function

Merge (linked list)

```
Require: a,b :: Linked List
  function MERGE(a,b)
     if NULL?(a) then
        return b
     else if NULL?(b) then
        return a
     else if FIRST(a) ≤ FIRST(b) then
        return CONS(FIRST(a), MERGE(REST(a), b))
     else
        return CONS(FIRST(b), MERGE(a, REST(b)))
     end if
  end function
```

Mergesort

```
function MERGESORT(s)
    sl ← LENGTH(s)
    if sl ≤ 1 then
        return s
    else
        mid ← ⌊sl/2⌋
        left ← MERGESORT(s[0...mid])
        right ← MERGESORT(s[mid...sl])
        return MERGE(left,right)
    end if
end function
```

Complexity analysis

Time complexity: merge

- each iteration:
  - two compares
  - two memory read/writes
  - one addition
- exactly LENGTH(a) + LENGTH(b) iterations

$$\implies \Theta(N_A + N_B)$$

Complexity analysis

Time complexity: merge

- each iteration:
  - two compares
  - two memory read/writes
  - one addition
- exactly LENGTH(a) + LENGTH(b) iterations

$$\implies \Theta(N_A + N_B)$$

Time complexity: mergesort

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

$$\implies \ldots?$$

Outline

Motivation

- (asymptotically) solves recurrence relationships
- including:
  - straightforward ones (from first year)
  - harder ones (from this course)
- you need to:
  - know the result
  - be able to apply the result
- (you don't need to know the proof)

Examples

Binary search

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

Binary tree traversal

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

Merge sort

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

Recursion trees

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

$$T(N)$$

Recursion trees

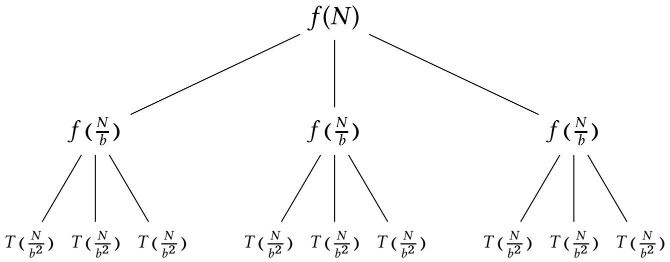
$$T(N) = aT\left(\frac{N}{b}\right) + f(N)$$

$$f(N)$$

$$T\left(\tfrac{N}{b}\right) \qquad T\left(\tfrac{N}{b}\right) \qquad T\left(\tfrac{N}{b}\right)$$

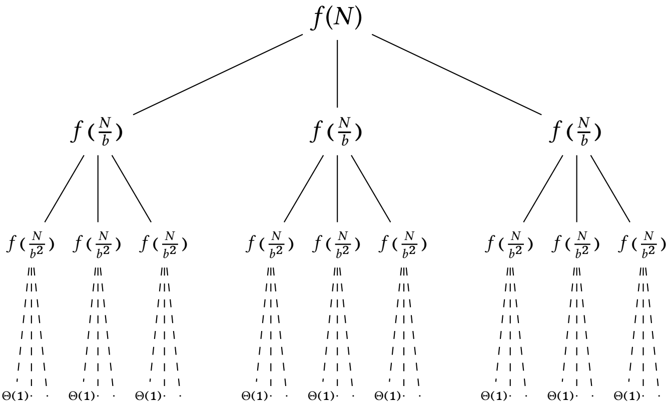
Recursion trees

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



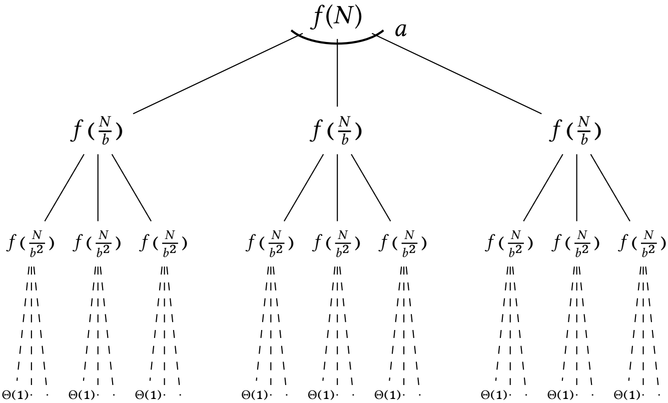
Recursion trees

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



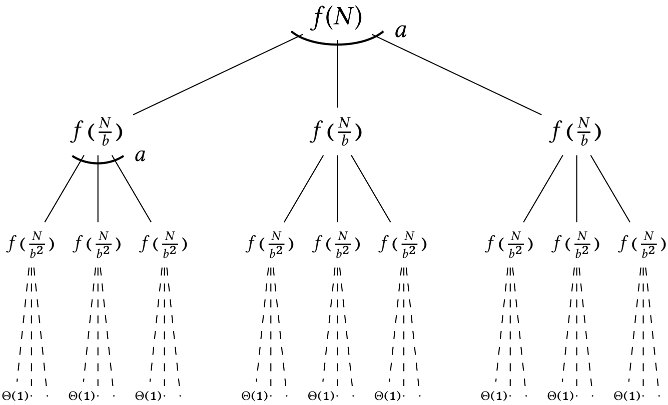
Recursion trees

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



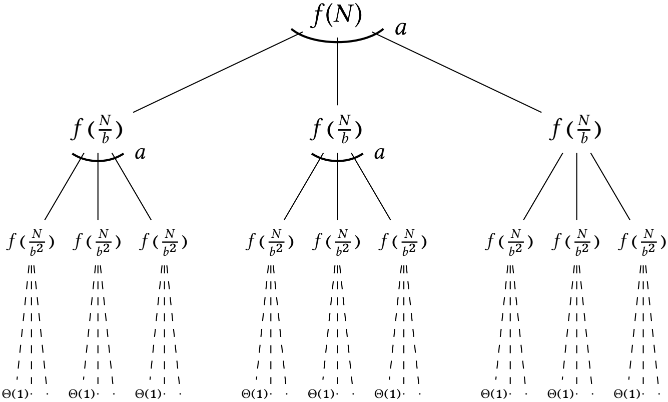
Recursion trees

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



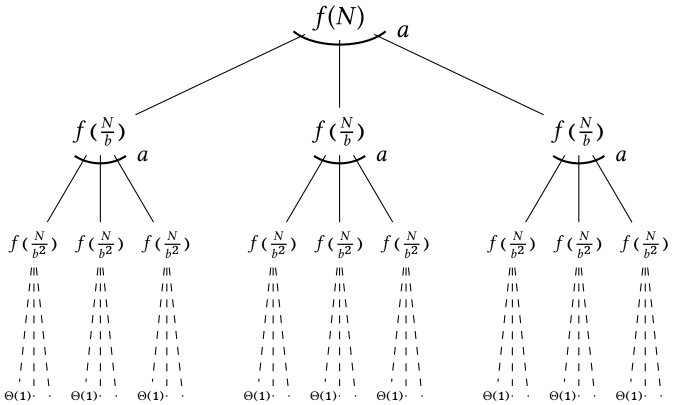
Recursion trees

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



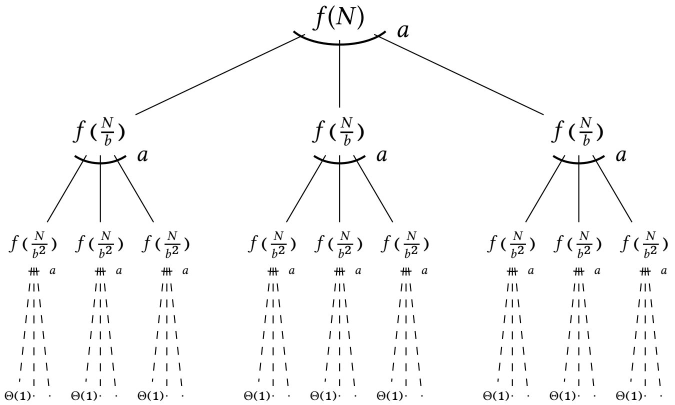
Recursion trees

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



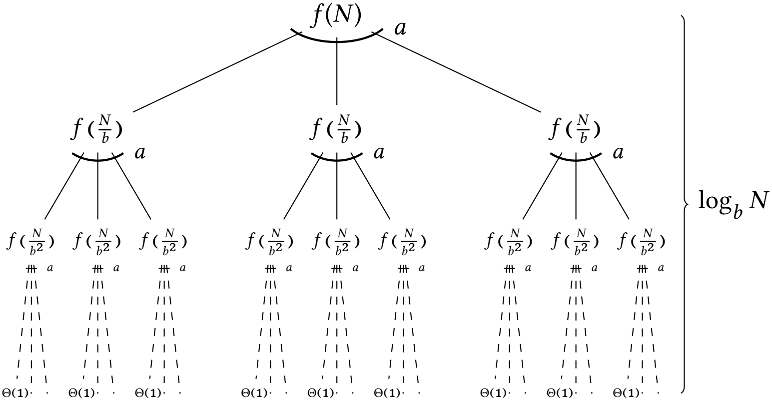
Recursion trees

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



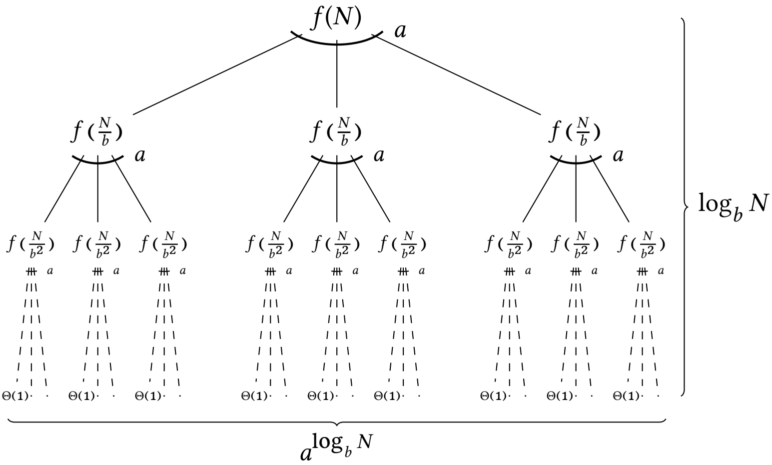
Recursion trees

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



Recursion trees

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



Theorem statement

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

Three cases:

1. $f(N) \in O\left(N^c\right)$ where $c < \log_b a$
2. $f(N) \in \Theta\left(N^c \log^k N\right)$ where $c = \log_b a$
3. $f(N) \in \Omega\left(N^c\right)$ where $c > \log_b a$

Theorem statement, case 1

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

$$f(N) \in O\left(N^c\right) \text{ where } c < \log_b a$$

Result

$$T(N) \in \Theta\left(N^{\log_b a}\right)$$

Example (binary tree traversal)

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

$$a = 2, b = 2, \log_b a = 1; f(N) = 1 \in O(N^0)$$

so

$$T(N) \in \Theta(N^1) = \Theta(N)$$

Theorem statement, case 2

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

$$f(N) \in \Theta\left(N^c \log^k N\right) \text{ where } c = \log_b a$$

Result

$$T(N) \in \Theta\left(N^{\log_b a} \log^{k+1} N\right)$$

Example (binary search)

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

$$a = 1, b = 2, \log_b a = 0; f(N) = 1 \in \Theta(N^0)$$

so

$$T(N) \in \Theta(N^0 \log^1 N) = \Theta(\log N)$$

Theorem statement, case 2

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

$$f(N) \in \Theta\left(N^c \log^k N\right) \text{ where } c = \log_b a$$

Result

$$T(N) \in \Theta\left(N^{\log_b a} \log^{k+1} N\right)$$

Example (merge sort)

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

$$a = 2, b = 2, \log_b a = 1; f(N) = N \in \Theta(N^1)$$

so

$$T(N) \in \Theta(N^1 \log^1 N) = \Theta(N \log N)$$

Theorem statement, case 3

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

$$f(N) \in \Omega\left(N^c\right) \text{ where } c > \log_b a$$

Result

$$T(N) \in \Theta\left(f(N)\right)$$

Example (quickselect)

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

$$a = 1, b = 2, \log_b a = 0; f(N) = N \in O(N^1)$$

so

$$T(N) \in \Theta(f(N)) = \Theta(N)$$

Proof

See CLRS for details:

- polynomially smaller/bigger;
- handling floor $\lfloor x \rfloor$ and ceiling $\lceil x \rceil$;
- regularity condition for case 3;

See also:

- Mohamad Akra and Louay Bazzi, *On the solution of linear recurrence equations*, Computational Optimization and Applications 10(2):195–210, 1998

Work

1. draw out the recursion trees for each of the examples given in the lecture (binary tree traversal, binary search, merge sort, quickselect) and convince yourself by adding up the contributions at each of the nodes that the results given in the lecture are correct.
2. Reading
   - CLRS, sections 4.4 and 4.5
3. Questions from CLRS

   Exercises 2.3-3, 2.3-4, 4.5-1, 4.5-2

   4-1 Recurrence examples

   4-2 Parameter passing costs

   4-3 More recurrence examples
4. do the quiz on Recurrence Relations on the VLE
   - open from 19th November
   - as many attempts as you like (but: 4 hours between attempts)