Data Structures and Algorithms

Alexandre Duret-Lutz & Anupam Gupta
adl@lrde.epita.fr, anupamg@iitk.ac.in

September 9, 2010

Introduction

Objective
To acquire an “algorithmic thinking” to solve problems.

Means
- Practical culture:
  - learning basic data structures
  - learning classical algorithms for common problems
  - learning design strategies for algorithms
- Theoretical culture:
  - learning to reason about algorithms (proving that an algorithm does what it is designed to do, analyzing its complexity, ...)

Plan for the course

First half of the semester (we me):

Week 1 (this week): introducing concepts by studying how real-life algorithms can be adapted to computers,

Weeks 2–3 Defining the complexity of algorithms, and introducing tools to compute complexity.

Weeks 2–4 Many algorithms to sort data.

Weeks 5–6 Data structures.

mid semester exams

Second half of the semester (with Anupam Gupta):

Common design strategies for algorithms

Graph algorithms

Resources

- Lecture notes for this course (this document) in `http://www.lrde.epita.fr/~adl/ens/iitj/eso211/`.
- *Introduction to Algorithms (3rd edition)*, by Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein.
- You can find **many** web pages, and books dedicated to the topics discussed here.

Looking up a word in a dictionary (1/3)

We can think of many algorithms:

1. You have never seen a dictionary in your life and do not known how it is organized.
   Algo 1: Read the pages one by one until you find the word.
   Note: you can read the pages in any order as long as you read them all. (You may want to tear the pages you have read if you use a random order.)
2. You know a dictionary is alphabetically sorted.
   Algo 2: Open the dictionary in the middle. Look at the first word of the right page, and using the order on words tear the half of the dictionary that cannot contain the words you are looking. Repeat until you are left with one page..
3. You have an idea of the statistical distribution of words in the dictionary. Algo 3: If your word starts with `D`, you may want to open the dictionary near the beginning.

Looking up a word in a dictionary (2/3)

- We obviously expect "Algo 1" to be slower than "Algo 2" itself slower than "Algo 3".
- Algo 1 works even if the dictionary is not sorted: data can be arranged in any way.
- Algo 2 & 3 are faster because the data is organized in a way that helps: the words of the dictionary are sorted. In general is it always good to organize items in a way that ease operations on these items.[1]
- Algo 3 requires further knowledge on the data. Without this knowledge, you can make a educated guess on the distribution (e.g. uniform), because it is *likely* to speedup the search anyway. Such an approximation is called a **heuristic**.

[1] For another example: think of the way a library is organized in order to make the following three operations efficient: search a book in the library, remove a book from the library, and insert it back.

Looking up a word in a dictionary (3/3)

Can we compare the efficiency of these three algorithms?

- We can give a dictionary to three people, and ask them to look up the same word using each a different algorithm.
- This is only one test case: it does not mean anything.
  - For instance if we ask the three people to lookup for the word "A", **Algo 1** will terminate first since this is the first word of the dictionary.
  - Should we conclude that **Algo 1** is better than the other two algorithms?
- Can we define a best case and a worst case for each algorithm?
- Can we evaluate the speed of an algorithm independently of the people executing it (some people are faster than others, but that should not influence our algorithm comparison).
- Can we define an *average* case?

Searching for a track on a music tape

- A dictionary can be open anywhere: it takes the same time.
- A music tape (or audio cassette) works differently: if you are asked to play the fourth track, you first have to fast forward the tape to the start of that track (it costs some time) and then you can actually play that track.
- If a dictionary was recorded on the tape: you would not naturally consider the use **Algo 2 or 3**, because of the time it take to seek to the middle to tape, read a word, then seek elsewhere, etc.
- If the dictionary fill the whole tape, can you compute the distance of the tape we will have to fast-forward or rewind in the worst case during the execution of **Algo 2**?

In computer terms, we say that a dictionary has a **random access** (i.e. immediate access to arbitrary locations) while a tape (without rewind and fast-forward) only has **sequential access**: elements can only be accessed in a predefined order.

Searching a card in a deck of cards

You are given a (shuffled) deck of cards, and asked to remove the Jack of spades.

- Algo 1 still works.
- Algo 2 & 3 require you to sort the deck of cards before you actually search for the card. Is it worth it? Maybe we can sort the card so quickly than sorting plus searching with Algo 2 or 3 will still be faster than Algo 1 alone. Can you make an argument why it is impossible ?
  We will use this kind of arguments to prove **lower bounds** on the complexity of algorithms.

Here is a challenging exercise (for which we shall study the answer latter): You are given a huge pile of 256 exam papers that have already been graded. Your job is the find the median grade, i.e., the paper that would be in the middle of the pile if that pile was sorted. Can you do that in a way that is faster than sorting the pile?

Sorting cards (1/2)

- Can you describe an algorithm to sort a hand of cards?
  Here are two:
  - insertion sort: stack the unsorted cards in front of you, then pick the cards one by one and place it at the right place into your hand. It also work if you place the unsorted cards in one side of your hand and the sorted cards at the other side.
  - selection sort: put all the unsorted cards in your hand, remove the smallest one, and place it on a stack in front of you. Repeat until you have stacked all the cards in order.
- Would you use same sorting algorithms to sort an entire **pack** of cards? Why not?
  Here is a possible algorithm: make a first pass on the pack to build 4 stacks, one for each suit. Sort each stack as if it was a hand.

Sorting cards (2/2)

- We use a different algorithm because the number of card is too large for our hands.
- But it is also the case that sorting a hand of cards using the algorithm we use to sort a pack of cards would be slower.
- On a computer we might have similar tradeoffs:
  - Sometimes we have so much data to process it will not fit in memory: we need to devise way to process the data in smaller chunks.
  - It is often the case that an algorithm that is efficient for processing a lot of data, will be less efficient on a smaller number of data. Using another algorithm when the number of item is small is a common implementation trick.

Counting people in the room (1/2)

Here are two algorithms:

Algo A: Look at each person in order, and increment a counter in your head.
This assumes that is easy to define the order (for instance if everybody is seated in the room).

Algo B: This algorithm requires participation from everybody and goes as follows:

1. Everybody stand up and remember the number 1
2. If you are standing up and are not alone, find somebody else standing up, and add your numbers. One of you two should now seat down.
3. Repeat step 2 until you are standing up alone. Then shout your number, this is the number of people in the room.

Counting people in the room (2/2)

- We apparently have another good-for-large/bad-for-small tradeoff here: **Algo B** will be a lot faster than **Algo A** with a lot of people. **Algo A** will be faster with only a handful of people.
- **Algo B** is an example of parallel algorithm: there are several units of execution (the people) working at the same time, while **Algo A** is a sequential algorithm with only one unit of execution.
- If you imagine **Algo B** running in waves, where half of the people standing up seat down during each wave, you should see a similarity with **Algo 2** for dictionary lookup (discarding half of the dictionary at each step). Can you tell how many waves it will take to count a room of n people?

Random Access Machines

To study algorithm, we will work on an idealized computer: a Random Access Machine (or RAM).

A RAM has

- a unique processing unit that executes instructions sequentially
- random access to the memory (in constant time)
- infinite memory

To measure the time complexity of an algorithm, we will study the time it takes to execute. We can do that by counting the number of instructions executed.

We can also measure the space complexity of an algorithm by studying the size of the memory it requires to work.

(We will mostly focus on time complexity in this course.)

Pseudo Code

- To describe algorithms, we will use some pseudo-code.
- Pseudo code is midway between English and actual source code.
- It use conventions from computer languages (like using loops, functions) but without obeying syntax rules; the goal is to provide a compact and high-level description of the algorithm.
- It may include some mathematical expressions or natural language descriptions of some operations.

Dictionary Lookup: Data Structure

- How to represent the dictionary in memory in order to access each word easily?
- Problem: words in dictionary do not have constant size.

A first solution: Concatenate all words in memory, using a special symbol to separate words.

`a$aback$abandon$...$zucchini$zweiback`

Here the separating symbol is `$`, but on a C/C++ implementation you would likely use the string terminator `\0`.

Is this representation suitable for our three dictionary lookup algorithms?

Dictionary Lookup: Pseudo-Code for Algo 1

Input: an array D of characters, of size s, in which words are delimited by '\$'; a word w to search.
Output: an index $i \in \{0, \ldots, s-1\}$ such that $D[i]$ is the start of a word equal to w, or -1 if $w \notin D$.

DictionaryLookup(D, s, w) :

$pos \leftarrow 0$
while $pos < D$ do
    $end \leftarrow pos$
    repeat $end \leftarrow end + 1$ until $end > s$ or $D[end] =$'\$'
    if $w = D[pos..end]$ then
        return pos
    $pos \leftarrow end + 1$
done
return -1

Dictionary Lookup: Sentinel Value

A typical trick to get away with out-of-bounds checks such as $ens > s$ is to add a sentinel value at the end of the array. Here, if we replace
`a$aback$abandon$...$zucchini$zweiback`
by the following encoding
`a$aback$abandon$...$zucchini$zweiback$`

then we can simplify

repeat $end \leftarrow end + 1$ until $end > s$ or $D[end] =$'\$'

into

repeat $end \leftarrow end + 1$ until $D[end] =$'\$

because we know that we can always find a '\$' after a word.

Dictionary Lookup: Indirection

- Can we adapt **Algo 2** to this kind of encoding of a dictionary?
- Problem: because words have different sizes we cannot find the middle word easily. We can only easily find the middle character. Example: Consider `a$car$chance$schoolteacher` and search for a. You will only eliminate one word at a time.
- An idea: build an index table for all the words. I.e., an array that gives the starting position of each word.
- This second array may contain indices, or it may contain directly pointers to the actual words in the dictionary. We will now assume the latter and \0 termination.

Dictionary Lookup: Linear Search

Input: an array A of (pointers to) strings, the size s of A, and a string w to search.
Output: an index $i \in \{0, \ldots, s-1\}$ such that $D[i]$ is the start of a string equal to w, or -1 if $w \notin D$.

LinearSearch(A, s, w) :

for $pos \leftarrow 0$ to $s - 1$ do
    if $w = D[pos]$ then
        return pos
done
return -1

- This algorithm is not restricted to strings: it will work with any kind of data.
- Can you give an upper bound on the number of iterations of the loop if $w \notin A$? (easy!)

Dictionary Lookup: Linear Search Speedup

How to speedup the detection of $w \notin A$ if A is sorted?

Input: a sorted array A of (pointers to) strings, the size s of A, and a string w to search.
Output: an index $i \in \{0, \ldots, s-1\}$ such that $D[i]$ is the start of a string equal to w, or -1 if $w \notin D$.

```
LinearSearchSorted(A, s, w) :
for pos ← 0 to s − 1 do
    if w = D[pos] then
        return pos
    if w < D[pos] then
        return −1
done
return −1
```

- Can you see how to use a sentinel value to remove the last line?
- Can you give an upper bound on the number of iterations of the

Dictionary Lookup: Binary Search

Input: a sorted array $A[l..r]$ of strings, a string v to lookup
Output: an index $i \in \{l, \ldots, r\}$ such that $A[i] = v$ or -1 if $v \notin A[l..r]$.

```
BinarySearch(A, l, r, v)      :
1   while l ≤ r do
2       m ← ⌊(l + r)/2⌋
3       if v = A[m] then
4           return m
5       else
6           if v < A[m] then
7               r ← m − 1
8           else
9               l ← m + 1
10  done
11  return −1
```


Notes on Binary Search

- There are two ways to exit this algorithm: either at line 4 (if v is found) or at line 11 (if v is not found).
- How can we prove that it will exit the loop if v is not found ?
- Can you give an upper bound on the number of iterations if the loop if $v \notin A[l..r]$?
- This algorithm works on any type of data that is ordered.

Compare iterative and recursive BinarySearch

```
BinarySearch(A, l, r, v):
while l ≤ r do
    m ← ⌊(l + r)/2⌋
    if v = A[m] then
        return m
    else
        if v < A[m] then
            r ← m − 1
        else
            l ← m + 1
done
return −1
```

```
BinarySearch(A, l, r, v):
if l ≤ r then
    m ← ⌊(l + r)/2⌋
    if v = A[m] then
        return m
    else
        if v < A[m] then
            return BinarySearch(A, l, m − 1, v)
        else
            return BinarySearch(A, m + 1, r, v)
else
return −1
```

Representing a Hand of Cards

- How can we best represent a hand of cards?
  We assume the order of the cards in the hand matters.
- An array?
  - accessing the ith card is fast
  - swapping two cards is easy
  - moving one card to another place require to shift all cards in-between (costly)
- A linked list[2]?
  - accessing the ith card is slow
  - swapping two cars is easy (if you have pointers to them)
  - moving a card to another place is efficient if you know the destination

[2] http://en.wikipedia.org/wiki/Linked_list

Insertion Sort on an Array

Input: an array A of items (e.g. cards) to sort
Output: the array A sorted in increasing order

```
InsertionSortArray(A)
for j ← 2 to length(A) do
    key ← A[j]
    i ← j − 1
    while i > 0 and A[i] > key do
        A[i + 1] ← A[i]
        i ← i − 1
    A[i + 1] ← key
```


Insertion Sort on a Linked List

Input: a linked list L of items to sort
Output: the list L sorted in increasing order

```
InsertionSortList(L)
if L = ∅ then return L
res ← L; L ← L.next; res.next ← ∅
while L ≠ ∅
    tmp ← L.next
    if res.data > L.data then
        L.next ← res; res ← L
    else
        dst ← res
        while dst.next ≠ ∅ and dst.next.data ≥ L.data do
            dst ← dst.next
        L.next ← dst.next; dst.next ← L
    L ← tmp
return res
```


Insertion Sort: Array vs. List

- Note how the two data structures command slightly different algorithms even though the basic idea is the same.
- From distance the two algorithms do the same thing:
  - Consider all items from left to right.
  - For each item, find its place in the previous items and insert it there.
- Finding the place can be done using a search from left to right, or from right to left.
  - A Singly Linked List forbids a search from right to left, so we have to work from left to right.
  - Inserting in a array requires to shift all elements at the right of the insertion, so it is more efficient to shift these elements as we search from right to left.

Measuring complexity

Let us show how we can measure the time complexity of an algorithm.

- What we want is to see how the algorithm scales as the input grows larger.
- In other words, the time complexity is a function $T(n)$ where n is the size of the input.

We measure time formally by counting executed instructions.

- Different instructions may have different costs (=run time), so we will have to weight them.
- Actual cost of an instruction in pseudo-code is dependent on
  - the programmer who translated pseudo-code to a programming language
  - the compiler who translated to programming language into machine code
  - the CPU who is executing the machine code
- Eventually, we will abstract from these "implementation details" factors

Insertion Sort on an Array

Input: an array A of items (e.g. cards) to sort
Output: the array A sorted in increasing order

InsertionSortArray(A)	cost	occ.
1 for $j \leftarrow 2$ to $length(A)$ do	c_1	n
2 $key \leftarrow A[j]$	c_2	$n-1$
3 $i \leftarrow j-1$	c_3	$n-1$
4 while $i > 0$ and $A[i] > key$ do	c_4	$\sum_{j=2}^{n} t_j$
5 $A[i+1] \leftarrow A[i]$	c_5	$\sum_{j=2}^{n}(t_j-1)$
6 $i \leftarrow i-1$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7 $A[i+1] \leftarrow key$	c_7	$n-1$

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n}(t_j-1) + c_6 \sum_{j=2}^{n}(t_j-1) + c_7(n-1)$$

What are the best cases? worst cases?

InsertionSort: Best and Worst cases

Best case: the array is sorted. $t_j = 1$.

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1)$$

This is a linear function of the form $an + b$.

Worst case: the array is reversed. $t_j = j$.
Recall that $\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$ and $\sum_{j=2}^{n}(j-1) = \frac{n(n-1)}{2}$.

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) + c_5\frac{n(n-1)}{2} + c_6\frac{n(n-1)}{2} + c_7(n-1)$$

This is a quadratic function of the form $an^2 + bn + c$.

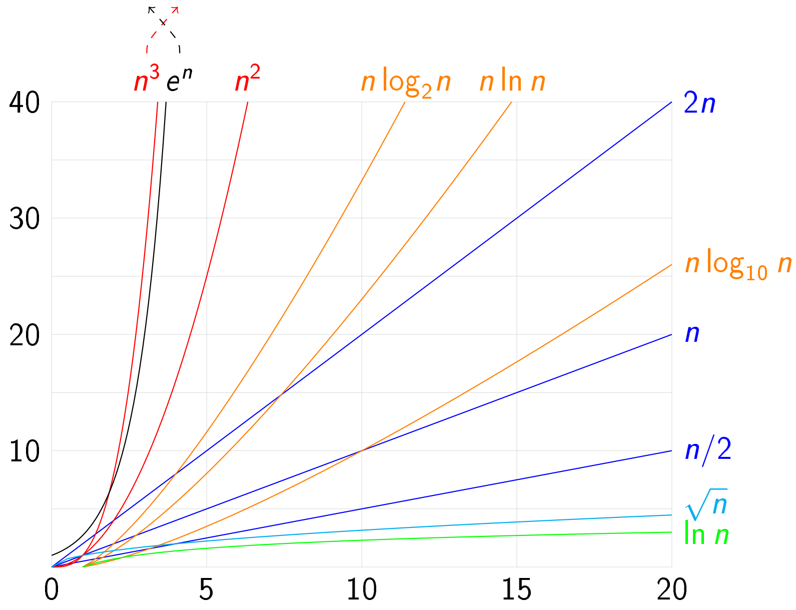
InsertSort: Average case

- The best case gives an upper bound for the complexity
- The worst case gives a lower bound for the complexity
- The general case is obviously in between
- We can also do average case analysis:
  Assume the array contains n randomly chosen numbers (following a uniform distribution). For a *key* picked at line 2, we expect half of the values of the array to be greater than *key*, and half less than *key*. Therefore $t_j = \frac{t}{2}$.
  We have $\sum_{j=2}^{n} \frac{t}{2} = \frac{n(n+1)-2}{4}$ and $\sum_{j=2}^{n} \frac{t}{2} - 1 = \frac{n(n-3)+2}{4}$

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\frac{n(n+1)-2}{4} + c_5\frac{n(n-3)+2}{4} + c_6\frac{n(n-3)+2}{4} + c_7(n-1)$$

This is again a quadratic function of the form $an^2 + bn + c$.

Function Comparison



In practice

Assuming 10^6 operations per second.

n	$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n
10^2	6.6 μs	0.1 ms	0.6 ms	10 ms	1 s	$4 \cdot 10^6$ y
10^3	9.9 μs	1 ms	10 ms	1 s	16.6 min	
10^4	13.3 μs	10 ms	0.1 s	1.6 min	11.6 d	
10^5	16.6 μs	0.1 s	1.6 s	2.7 h	347 y	
10^6	19.9 μs	1 s	19.9 s	11.6 d	10^6 y	
10^7	23.3 μs	10 s	3.9 min	3.17 y		
10^8	26.6 μs	1.6 min	44.3 min	317 y		
10^9	29.9 μs	16.6min	8.3 h	31709 y		

Machine Independence

In our formulas for $T(n)$, coefficients c_1, c_2, ..., c_7 are machine-dependent (and compiler-dependent, and programmer-dependent).

We would like to:

- ignore machine-dependent constants,
- study to growth of $T(n)$ when $n \to \infty$.

$\Longrightarrow$ Let's perform an asymptotic analysis of the run-time complexity.

Asymptotic Equivalence of Functions

$$\Theta(g(n)) = \{f(n) \mid \exists c_1 \in \mathbb{R}^{+\star}, \exists c_2 \in \mathbb{R}^{+\star}, \exists n_0 \in \mathbb{N}, \\ \forall n \geq n_0, 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

By convention (and abuse) we write "$f(n) = \Theta(g(n))$" instead of "$f(n) \in \Theta(g(n))$".

If $a_2 > 0$, we have $a_2 n^2 + a_1 n + a_0 = \Theta(n^2)$.
For instance let $c_1 = \frac{a_2}{2}$ and $c_2 = \frac{3a_2}{2}$, then show that

$$\frac{a_2}{2} \leq a_2 + \underbrace{\frac{a_1}{n} + \frac{a_0}{n^2}}_{\to 0 \text{ when } n \to \infty} \leq \frac{3a_2}{2}$$

Asymptotic Complexity of Insertion Sort

Best case

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1) = \Theta(n)$$

Worst case

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) + c_5\frac{n(n-1)}{2} + c_6\frac{n(n-1)}{2} + c_7(n-1) = \Theta(n^2)$$

Average case

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\frac{n(n+1)-2}{4} + c_5\frac{n(n-3)+2}{4} + c_6\frac{n(n-3)+2}{4} + c_7(n-1) = \Theta(n^2)$$

Let us Simplify the Calculations

One way to simplify the calculation is to pick one fundamental operation (or one familly of operations) for the problem and count only its number of executions.
The choice is good if the total number of operations is proportional to the count of fundamental operations executed.

Examples:

problem	fundamental operation
addition of binary numbers	all binary operations
matrix multiplication	scalar multiplication
sorting an array	comparisons of elements

Simplified Calculations

InsertionSortArray(A)	cost	occ.
for $j \leftarrow 2$ to $length(A)$ do	c_1	n
$key \leftarrow A[j]$	c_2	$n-1$
$i \leftarrow j-1$	c_3	$n-1$
while $i > 0$ and $A[i] > key$ do	c_4	$\sum_{j=2}^{n} t_j$
$A[i+1] \leftarrow A[i]$	c_5	$\sum_{j=2}^{n}(t_j - 1)$
$i \leftarrow i-1$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
$A[i+1] \leftarrow key$	c_7	$n-1$

Simplified Calculations

InsertionSortArray(A)	occ.
for $j \leftarrow 2$ to $length(A)$ do	
$key \leftarrow A[j]$	
$i \leftarrow j-1$	
while $i > 0$ and $A[i] > key$ do	$\sum_{j=2}^{n} t_j$
$A[i+1] \leftarrow A[i]$	
$i \leftarrow i-1$	
$A[i+1] \leftarrow key$	

Asymptotic Complexity of Insertion Sort

Best case

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1) = \Theta(n)$$

Worst case

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) + c_5\frac{n(n-1)}{2} + c_6\frac{n(n-1)}{2} + c_7(n-1) = \Theta(n^2)$$

Average case

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\frac{n(n+1)-2}{4} + c_5\frac{n(n-3)+2}{4} + c_6\frac{n(n-3)+2}{4} + c_7(n-1) = \Theta(n^2)$$

Asymptotic Complexity of Insertion Sort

Best case

$$T(n) = \quad n-1 = \Theta(n)$$

Worst case

$$T(n) = \quad \frac{n(n+1)}{2} - 1 = \Theta(n^2)$$

Average case

$$T(n) = \quad \frac{n(n+1)-2}{4} = \Theta(n^2)$$

Asymptotic Higher and Lower Bounds

$$\mathrm{O}(g(n)) = \{f(n) \mid \exists c \in \mathbb{R}^{+\star}, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq f(n) \leq cg(n)\}$$

$$\Omega(g(n)) = \{f(n) \mid \exists c \in \mathbb{R}^{+\star}, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq cg(n) \leq f(n)\}$$

Note that $f(n) = \Theta(g(n)) \iff f(n) = \mathrm{O}(g(n))$ et $f(n) = \Omega(g(n))$.

In other words $\Theta(g(n)) = \mathrm{O}(g(n)) \cap \Omega(g(n))$.

Since $\Theta(n) \subseteq \mathrm{O}(n^2)$ and $\Theta(n^2) \subseteq \mathrm{O}(n^2)$, we can say that the run-time complexity of Insertion Sort for n elements is in $\mathrm{O}(n^2)$.

Properties

- If $\lim_{n\to\infty} \frac{g(n)}{f(n)} = c > 0$ then $g(n) = \Theta(f(n))$.
- If $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$ then $g(n) = \mathrm{O}(f(n))$ and $f(n) \neq \mathrm{O}(g(n))$.
- If $\lim_{n\to\infty} \frac{g(n)}{f(n)} = \infty$ then $f(n) = \mathrm{O}(f(n))$ and $g(n) \neq \mathrm{O}(f(n))$.
- $f(n) = \mathrm{O}(g(n))$ iff $g(n) = \Omega(f(n))$.

If $f_1(n) = \Theta(g_1(n))$, $f_2(n) = \Theta(g_2(n))$, and k is constant, we have:

- $f_1(n) \times f_2(n) = \Theta(g_1(n) \times g_2(n))$
- $f_1(n) + f_2(n) = \Theta(g_1(n) + g_2(n))$
- $k \times f_1(n) = \Theta(g_1(n))$

The above rules are also true for O and Ω.

Exercises

- Find two functions f and g such that $f(n) \neq \mathrm{O}(g(n))$ and $g(n) \neq \mathrm{O}(f(n))$.
  We say that the two functions are incomparable using O.
- Show that $\Theta(f(n) + g(n)) = \Theta(\max(f(n), g(n)))$.
- If $f(n) = \Theta(g(n))$, do we have $2^{f(n)} = \Theta(2^{g(n)})$?
- Let us define the following (partial) order:

$$\Theta(f(n)) \leq \Theta(g(n)) \text{ si } f = \mathrm{O}(g(n))$$
$$\Theta(f(n)) < \Theta(g(n)) \text{ si } f = \mathrm{O}(g(n)) \text{ et } g \notin \mathrm{O}(f(n))$$

  Order the sets $\Theta(\ldots)$ containing the following functions:
  n, 2^n, $n \log n$, $\ln n$, $n + 7n^5$, $\log n$, $\sqrt{n}$, e^n, 2^{n-1}, n^2, $n^2 + \log n$, $\log\log n$, n^3, $(\log n)^2$, $n!$, $n^{3/2}$.

Complexity Analysis with Asymptotic Notations

Consider a general input of size n, and count the occurrences of each instruction using asymptotic notations.

InsertionSortArray(A)	
for $j \leftarrow 2$ to $length(A)$ do	$\Theta(n)+$
do $key \leftarrow A[j]$	$\Theta(n)+$
$i \leftarrow j - 1$	$\Theta(n)+$
while $i > 0$ and $A[i] > key$ do	$\mathrm{O}(n^2)+$
do $A[i+1] \leftarrow A[i]$	$\mathrm{O}(n^2)+$
$i \leftarrow i - 1$	$\mathrm{O}(n^2)+$
$A[i+1] \leftarrow key$	$\Theta(n)$
	$\mathrm{O}(n^2)$

Selection Sort

Idea: Find minimum of $A[1..n]$ then swap it with $A[1]$. Find minimum of $A[2..n]$ then swap it with $A[2]$. Etc. $A[1..k]$ is sorted after k iterations.

SelectionSort(A)	
for i from 1 to n do	$\Theta(n)$
$min \leftarrow i$	$\Theta(n)$
for j from i to n do	$\Theta(n^2)$
if $A[j] < A[min]$ then $min \leftarrow j$	$\Theta(n^2)$
$A[min] \leftrightarrow A[i]$	$\Theta(n)$
	$\Theta(n^2)$

It is worse than InsertionSort which is in $\mathrm{O}(n^2)$ but not in $\Omega(n^2)$.

Merge Sort

Input: an array A of integers, two indices l, r
Output: array A, with it subarray $A[l..r]$ sorted in increasing order

MergeSort(A, l, r)	$T(1)$	$T(n)$
if $l < r$ then	$\Theta(1)$	$\Theta(1)$
$m \leftarrow \lfloor (l+r)/2 \rfloor$		$\Theta(1)$
MergeSort(A, l, m)		$T(\lfloor n/2 \rfloor)$
MergeSort($A, m+1, r$)		$T(\lceil n/2 \rceil)$
Merge(A, l, m, r)		$\Theta(n)$

Using $n = r - l + 1$, we can express $T(n)$ using a recursive equation:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

Exercise: Write pseudo-code for Merge and prove its $\Theta(n)$ complexity.

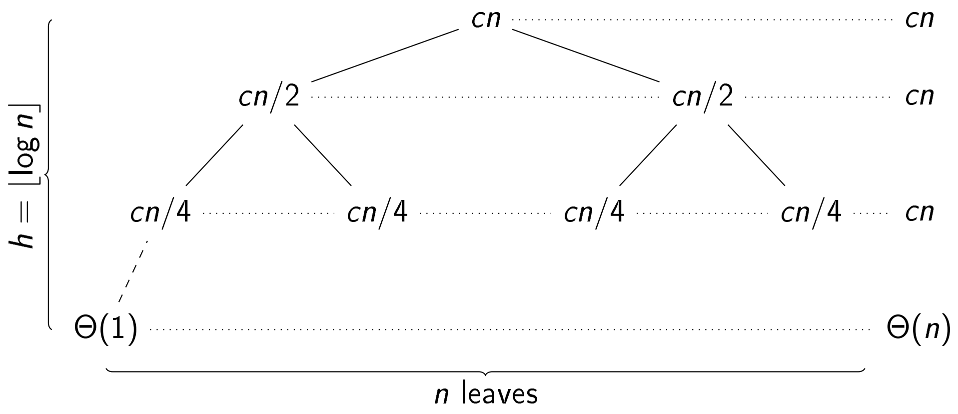
Merge

Input: an array A, three indices l, m, r such that $A[l..m]$ and $A[m+1..r]$ are sorted
Output: array A, with it subarray $A[l..r]$ sorted

Merge(A, l, m, r)	
$i \leftarrow l$; $j \leftarrow m+1$; $k \leftarrow l$	$\Theta(1)$
while $i \geq m$ and $j \geq r$ do	$\mathrm{O}(n)$
if $A[i] \geq A[j]$ then	$\mathrm{O}(n)$
$B[k] \leftarrow A[i]$; $k \leftarrow k+1$; $i \leftarrow i+1$	$\mathrm{O}(n)$
else	$\mathrm{O}(n)$
$B[k] \leftarrow A[j]$; $k \leftarrow k+1$; $j \leftarrow j+1$	$\mathrm{O}(n)$
if $i \geq m$ then	$\Theta(1)$
$B[k..r] \leftarrow A[i..m]$	$\mathrm{O}(n)$
else	$\mathrm{O}(1)$
$B[k..r] \leftarrow A[j..r]$	$\mathrm{O}(n)$
$A[l..r] \leftarrow B[l..r]$	$\Theta(n)$
return A	$\Theta(1)$
	$\Theta(n)$

Call Tree for MergeSort

Let us solve $T(n) = 2T(n/2) + cn$ graphically, for a constant $c > 0$.



We have $T(n) = \Theta(n \log n)$.

Solving Recursions by Unfolding

As second way to solve $T(n) = 2T(n/2) + cn$ is by unfolding.

$$
\begin{aligned}
T(n) &= cn + 2T(n/2) \\
&= cn + 2(cn/2) + 4T(n/4) = 2cn + 4T(n/4) \\
&= 2cn + 4(cn/4) + 8T(n/8) = 3cn + 8T(n/8) \\
&\vdots \\
&= kcn + 2^k T(n/2^k)
\end{aligned}
$$

We can continue unfolding until we reach $T(1) = \Theta(1)$. This happens when $2^k = n$. Substituting $k = \log_2 n$, we get:

$$
\begin{aligned}
T(n) &= cn \log_2 n + n \times T(1) \\
&= \Theta(n \log n) + \Theta(n) \\
&= \Theta(n \log n)
\end{aligned}
$$

General Theorem for Recurrence Equations

Let

$$
T(n) = \begin{cases} \Theta(1) & \text{si } n \leq n_0 \\ aT(n/b) + f(n) & \text{si } n > n_0 \end{cases}
$$

with $a \geq 1$, $b > 1$, $n_0 \in \mathbb{N}$.
Then

- if $f(n) = \mathrm{O}(n^{\log_b a - \varepsilon})$ for some $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
- if $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.
- if $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some $\varepsilon > 0$, **and** if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all large enough values of n, then $T(n) = \Theta(f(n))$.

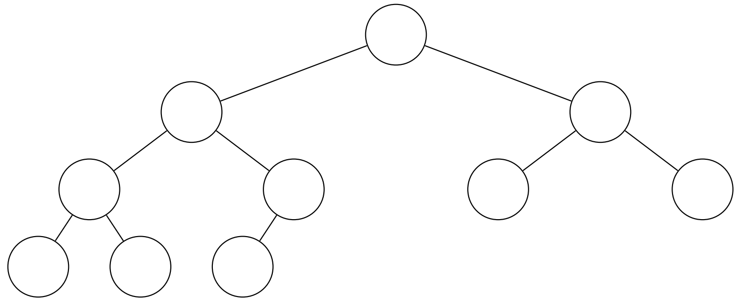
Beware, there are holes in this theorem: a function $f(n)$ may belong to none of the three cases. The ε constrains functions $f(n)$ to be polynomially smaller or greater than $n^{\log_b a}$.

Applications of the Theorem

- $T(n) = 2T(n/2) + \Theta(n)$
  $a = b = 2$, $n^{\log_b a} = n$. We have $T(n) = \Theta(n \log n)$.
- $T(n) = 4T(n/2) + \sqrt{n}$
  $a = 4$, $b = 2$, $n^{\log_b a} = n^2$. $\epsilon = 1$ and $\sqrt{n} = \mathrm{O}(n^{2-1})$, we thus have $T(n) = \Theta(n^2)$
- $T(n) = 3T(n/3) + n^2$
  $a = 3$, $b = 3$, $n^{\log_b a} = n$. $\epsilon = 1$ and $n^2 = \Omega(n^{1+1})$, furthermore $3(n/3)^2 \leq cn^2$ for $c = 1/3$, consequently $T(n) = \Theta(n^2)$.
- $T(n) = 4T(n/2) + n^2/\log n$
  $a = 4$, $b = 2$, $n^{\log_b a} = n^2$. We cannot find a $\varepsilon > 0$ such that $n^2/\log n = \mathrm{O}(n^{2-\varepsilon})$. Indeed, $n^2/\log n \leq cn^{2-\varepsilon}$ implies $n^\varepsilon \leq c \log n$. The theorem cannot apply.

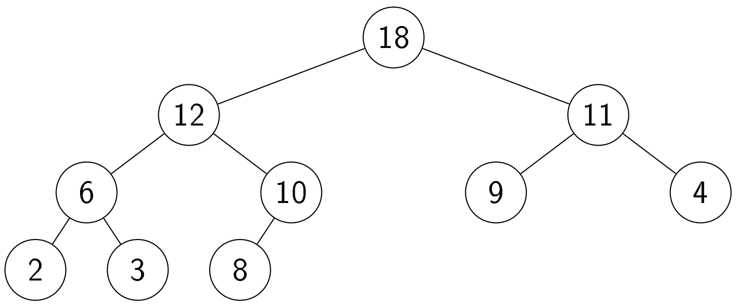
Perfect Tree

A perfect tree is a complete binary tree in which leaves from the deepest level are all grouped on the left (if that level is not complete).



Binary Heap

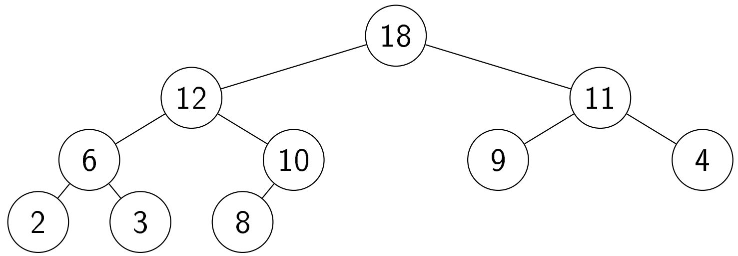
A binary heap is a perfect tree with the heap property: each node is greater than or equal to each of its children.



Any perfect tree can be efficiently stored as an array. This is how we will store binary heaps too.

18	12	11	6	10	9	4	2	3	8

Properties of a binary heap



$A =$

18	12	11	6	10	9	4	2	3	8

Index 1 in the array corresponds to the root of the tree

$\mathrm{Parent}(i) = \lfloor i/2 \rfloor$ (if $i > 0$)

$\mathrm{LeftChild}(i) = 2i$ (if it exists)

$\mathrm{RightChild}(i) = 2i + 1$ (if it exists)

Heap property: $\forall i > 0$, $A[\mathrm{Father}(i)] \geq A[i]$.

Operation on Binary Heaps (1): Heapify

Input: array A, and two indexes i and m such that $A[\text{LeftChild}(i)]$ and $A[\text{RightChild}(i)]$ are roots of heaps in $A[1..m]$,
Output: the array A such that $A[i]$ is the root of a heap.

Heapify(A, i, m)
$l \leftarrow$LeftChild(i)
$r \leftarrow$RightChild(i)
if $l \leq m$ and $A[l] > A[i]$
  then *largest* $\leftarrow l$
  else *largest* $\leftarrow i$
if $r \leq m$ and $A[r] > A[\mathit{largest}]$
  then *largest* $\leftarrow r$
if *largest* $\neq i$ then
  $A[i] \leftrightarrow A[\mathit{largest}]$
  Heapify(A, *largest*, m)

$T(n) \leq T(2n/3) + \Theta(1)$ with n the size of the subtree rooted at i, and $2n/3$ the maximum number of nodes of the subtree recursively explored.

We deduce $T(n) = \mathrm{O}(\log n)$

We can also write $T(h) = \mathrm{O}(h)$ with h the height of the tree rooted at i.

Operations on Heaps: BuildHeap

Input: an array A
Output: the array A organized as a heap.

BuildHeap(A)
for $i \leftarrow \lfloor \mathit{length}(A)/2 \rfloor$ down to 1 do
  Heapify($A, i, \mathit{length}(A)$)

The elements between $\mathit{length}(A)/2 + 1$ and the end of the array are the leaves: they are already heaps. We fix the rest of the array in a bottom-up way.

Intuitively if $n = \mathit{length}(A)$, $T(n) = \underbrace{\Theta(n)}_{\text{the loop}} \underbrace{\mathrm{O}(\log n)}_{\text{Heapify}} = \mathrm{O}(n \log n)$.

In fact the time spent in Heapify depends on the size of the subtree considered, not the entire tree. The complexity is better.

Complexity of BuildHeap

Let us calculate $T(n)$ for BuildHeap more precisely.

The height of a complete binary tree of n nodes is $\lfloor \log n \rfloor$.
The number of subtrees of height h in a heap is at most $\lceil n/2^{h+1} \rceil$.
The complexity of Heapify on a subtree of height h is est $\mathrm{O}(h)$.

We get:

$$T(n) = \sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \mathrm{O}(h) = \mathrm{O}\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^{h+1}} \right) = \mathrm{O}\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \right)$$

Since $\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$, we have $\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$.

Finally $T(n) = \mathrm{O}(n)$.

Heap Sort

HeapSort(A)
BuildHeap(A)
for $i \leftarrow \mathit{length}(A)$ down to 2 do
  $A[1] \leftrightarrow A[i]$
  Heapify($A, 1, i-1$)

The complexity is easy to express:

$$T(n) = \underbrace{\mathrm{O}(n)}_{\text{BuildHeap}} + \mathrm{O}(\underbrace{n}_{\text{loop}} \underbrace{\log n}_{\text{Heapify}}) = \mathrm{O}(n \log n)$$

Quick Sort

Origin

Sir Charles Antony Richard Hoare, 1962.

General idea

Partition the array in two parts, such that elements from the first part are smaller than elements from the second. Sort both parts recursively.

How to partition?

Pick a value and use it as pivot. Using successive swaps, arrange the array in two blocs such that

- elements at the beginning are less or equal to the pivot
- elements at the end are greater than or equal to the pivot

Our choice: the pivot is the first element.

Quick Sort: algorithm

Input: an array A, and two indices l and r
Output: the array A with $A[l..r]$ sorted in increasing order

```
QuickSort(A, l, r)
if l < r then
    p ← Partition(A, l, r)
    QuickSort(A, l, p)
    QuickSort(A, p + 1, r)
```

Input: an array A, two indices l and r
Output: an index p, the array A arranged so that $A[l..p] \leq A[p+1..r]$.

```
Partition(A, l, r)
x ← A[l]; i ← l − 1; j ← r + 1
repeat forever
    do i ← i + 1 until A[i] ≥ x
    do j ← j − 1 until A[j] ≤ x
    if i < j then
        A[i] ↔ A[j]
    else
        return j
```

$T_{\text{Partition}}(n) = \Theta(n)$ of $n = r - l + 1$.

Complexity of QuickSort

Unfavorable case

The choice of pivot is unlucky and yields a unbalanced partition. This happens if the input is already sorted (in any way).

$$\begin{aligned} T(n) &= \Theta(n) + \Theta(1) + T(n-1) \\ &= T(n-1) + \Theta(n) \\ &= \Theta(n^2) \end{aligned}$$

Best case

The partition always splits the array in the middle.

$$T(n) = \Theta(n) + 2T(n/2)$$

Same equation as MergeSort. We know the answer is $T(n) = \Theta(n \log n)$.

Intuition for Average Complexity

Let's assume an imbalance by a constant ratio: "1/10 : 9/10"

$$T(n) = T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) + \Theta(n)$$

Draw the call tree. The smallest branch has height $\log_{10} n = \Theta(\log n)$: the complexity of the call tree limited to this level is $\Theta(n \log n)$. We deduce that $T(n) = \Omega(n \log n)$. The longest branch has height $\log_{10/9} n$ and the complexity at each level is $\leq n$. We get that $T(n) \leq n \log_{10/9} n = \mathrm{O}(n \log n)$.

Finally $T(n) = \Theta(n \log n)$.

Any partition using a constant ratio implies $T(n) = \Theta(n \log n)$.

Calculation of the Average Complexity (1)

Let us assume a uniform distribution on the possible partitions. The partition cuts $A[1..n]$ in $A[1..i]$ and $A[i+1..n]$ with $n-1$ possible choices for i.

$$T(n) = \frac{1}{n-1}\sum_{i=1}^{n-1}(T(i) + T(n-i)) + \Theta(n) = \frac{2}{n-1}\sum_{i=1}^{n-1} T(i) + cn$$

Furthermore:

$$T(n-1) = \frac{2}{n-2}\sum_{i=1}^{n-2} T(i) + c(n-1)$$

Let's try to make $T(n-1)$ appear in $T(n)$:

$$T(n) = \frac{2(n-2)}{(n-1)(n-2)}\left(T(n-1) + \sum_{i=1}^{n-2} T(i)\right) + c(n-1+1)$$

Calculation of the Average Complexity (2)

$$\begin{aligned} T(n) &= \frac{2(n-2)}{(n-1)(n-2)}\left(T(n-1) + \sum_{i=1}^{n-2} T(i)\right) + c(n-1+1) \\ &= \frac{2}{n-1}T(n-1) + \frac{n-2}{n-1}T(n-1) + c(n-1)\left(1 - \frac{n-2}{n-1}\right) + c \\ &= \frac{n}{n-1}T(n-1) + 2c \end{aligned}$$

Divide left and right by n:

$$\frac{T(n)}{n} = \frac{T(n-1)}{n-1} + \frac{2c}{n}$$

Calculation of the Average Complexity (3)

$$\frac{T(n)}{n} = \frac{T(n-1)}{n-1} + \frac{2c}{n}$$

Let's introduce $Y(n) = \frac{T(n)}{n}$

$$Y(n) = Y(n-1) + \frac{2c}{n} = 2c\sum_{i=1}^{n}\frac{1}{i}$$

$$T(n) = 2cn\sum_{i=1}^{n}\frac{1}{i}$$

Using Euler's formula $\sum_{i=1}^{n}\frac{1}{i} = \ln n + \gamma + o(1) = \Theta(\log n)$ we get

$$T(n) = \Theta(n)\Theta(\log n) = \Theta(n\log n)$$

Questions

- What happens of all the elements of the array have the same value?
- It seems that a random array will be sorted more efficiently than a sorted array. How can we modify QuickSort to ensure that it will have the same (averge) complexity on random arrays and sorted arrays?

Stochastic QuickSort

A simple Idea: choose the pivot randomly in the array.

```
RandomizedPartition(A, l, r)
x ← A[Random(l, r)]; i ← l − 1; j ← r + 1
repeat forever
    do j ← j − 1 until A[j] ≤ x
    do i ← i + 1 until A[i] ≥ x
    if i < j then
        A[i] ↔ A[j]
    else
        return j
```

The effect is as if we had randomized the array before calling QuickSort.

Pro: no particular input is known to always provoke the worst case.
Cons: Random() is a slow function. Calling it so much (how many time is it called?) is a sure way to slow-down your implementation.

Another idea: median pivot

(The median of $2k+1$ values is the $(k+1)$st largest value.)

Idea: use as pivot the median of some values of the arrays (not all values, it would take too long to find the median).

For instance use the median of the first three value, or better (why?) the median of $A[l]$, $A[\lfloor\frac{l+r}{2}\rfloor]$ and $A[r]$.

Conclusion on Quick Sort

- We have $T(n) = O(n^2)$ in general but $T(n) = \Theta(n \log n)$ on the average.
- In practice Quick Sort is faster than the other sorting algorithms presented so far (assuming n is not ridiculously small).
- For a smaller n, Insertion Sort is a better choice.
- The `qsort()` implementation in GNU Libc (and others) use these tricks:
  - Use median-of-3 pivot (extremities and middle).
  - Switch to Insertion Sort if the array has ≤ 4 elements.
  - Order the two recursive calls such that the first one sees the smallest subarray, and the latter one (which is a tail recursion) use the largest subarray.
  - Do not actually perform recursive calls: tail recursion can be replaced by a loop, and the first call to QuickSort requires an explicit stack.

Introspective Sort

Origin
David Musser, 1997
Used in SGI's Standard Template Library. (`std::sort`)

Interest
Modification of Quick Sort so that $T(n) = \Theta(n \log n)$ always.

Idea
Detect when the values to sort are causing trouble to Quick Sort, and use a Heap Sort in this case.

In practice
We bound the number of recursive calls to $O(\log n)$.
Musser suggests $2\lfloor \log n \rfloor$.

Introspective Sort: Algorithm

```
IntroSort(A, l, r)
  IntroSort'(A, l, r, 2⌊log(r − l + 1)⌋)

IntroSort'(A, l, r, depth_limit)
  if depth_limit = 0 then
      HeapSort(A, l, r)
      return
  else
      depth_limit ← depth_limit − 1
      p ← Partition(A, l, r)
      IntroSort'(A, l, p, depth_limit)
      IntroSort'(A, p + 1, r, depth_limit)
```

This is a nifty implementation trick that you cannot think of without studying the complexity of your algorithms.

Sorting as a Problem

Input: A sequence of n numbers $\langle a_1, a_2, \ldots a_n \rangle$
Output: A permutation $\langle a'_1, a'_2, \ldots a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

These numbers might be attached to other data. For instance they might be a field in a record, and we want to sorts to records according to this field (called the key for the purpose of sorting).

The structure used to represent A is usually an array. (We have also looked at InsertionSort on a list.)

For now, we have only looked at "comparison sorts", i.e. algorithms that perform comparisons to order the elements.

In place and Stable Sorting Algorithm

In Place Sort

A sorting algorithm is **in place** if the number of memory it requires in addition to the input is independent of n, or at most $\Theta(\log n)$. Especially you are not allowed to use a temporary array of size n in an in place sort, since that would require $\Theta(n)$ memory.

Stable Sort

A sorting algorithm is **stable** if the order of equal elements is preserved. This matters when the key used for sorting is part of a larger structure (with other data attached), and several sorts are chained using different fields as key.

Summary of Sorting Algorithms Studied so far

	complexity	average	in place?	stable?
insertion sort	$O(n^2)$	$\Theta(n^2)$	yes	yes
selection sort	$\Theta(n^2)$		yes	no
merge sort	$\Theta(n \log n)$		no	yes
heap sort	$O(n \log n)$[1]		yes	no
quick sort	$O(n^2)$	$\Theta(n \log n)$	yes[2]	no
intro sort	$\Theta(n \log n)$		yes[2]	no

[1] The complexity is in fact $\Theta(n \log n)$, but we have not proved it.
[2] The number of temporary variables used locally by QuickSort and Partition is constant, but because of the recursive calls we are actually creating several copies of them (as many copies as the depth of the call tree). QuickSort requires $O(\log n)$ extra memory when ordering the partition so that the largest part is handled by the tail recursion (it requires $O(n)$ memory if you do not use such a trick).

Complexity of a problem

Definition

The complexity $C(n)$ of a problem P is the complexity if the best algorithm that solves P.

Consequences

- If an algorithm A solves P in $\mathrm{O}(f(n))$, then $C(n) = \mathrm{O}(f(n))$.
- If we can prove that all algorithms that solve P have a complexity in $\Omega(g(n))$, then $C(n) = \Omega(g(n))$.
- If these two bounds are equivalents (i.e., $f(n) = \Theta(g(n))$) then $C(n) = \Theta(f(n)) = \Theta(g(n))$, and this is the complexity of the problem.

For now, we have proved that "sorting n numbers" is in $\mathrm{O}(n \log n)$.

- That does not mean it is impossible to do better
- It is always possible to do worse :-)

Lower Bound for Worst Case of Comparison Sort

Recall the problem

Input: A sequence of n numbers $\langle a_1, a_2, \dots a_n \rangle$
Output: A permutation $\langle a'_1, a'_2, \dots a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

Argumentation

Sorts can be represented by a binary tree (decision tree). Internal nodes are comparisons between two elements: the left child represent a negative answer and the right child a positive answer. Leaves of the tree represent a permutation to apply to sort the array.

There exists $n!$ possible permutations of $\langle a_1, a_2, \dots a_n \rangle$. Our binary tree of $n!$ leaves, should therefore have a height of at least $\lceil \log n! \rceil$. Using Stirling's formula[3] we obtain that $\Omega(\log(n!)) = \Omega(n \log n)$.

The worst case of any comparison sort uses $\Omega(n \log n)$ comparisons.

[3] $n! = \sqrt{2\pi n}(n/e)^n(1 + \Theta(1/n))$

Counting Sort

Characteristics

Stable sort, not in place.
May be used only if the keys belong to a small interval. Here we assume they are in $\{1, \dots, k\}$.

Algorithm

CountingSort(A, B, k)

for $i \leftarrow 1$ to k do $C[i] \leftarrow 0$	$\Theta(k)$
for $i \leftarrow 1$ to $length(A)$ do $C[A[j]] \leftarrow C[A[j]] + 1$	$\Theta(n)$
for $i \leftarrow 1$ to k do $C[i] \leftarrow C[i] + C[i-1]$	$\Theta(k)$
for $i \leftarrow length(A)$ down to 1 do	$\Theta(n)$
$B[C[A[i]]] \leftarrow A[i]$	$\Theta(n)$
$C[A[i]] \leftarrow C[A[i]] - 1$	$\Theta(n)$
	$\Theta(n) + \Theta(k)$

Complexity

If $k = \mathrm{O}(n)$, then $T(n) = \Theta(n)$.

Bucket Sort

Characteristics

Unstable sort, not in place. Assume the elements are uniformly distributed. Here we assume they are in the interval $[0, 1[$.

Algorithm

BucketSort(A)

$n \leftarrow length(A)$	$\Theta(n)$
for $i \leftarrow 1$ to n do	$\Theta(n)$
insert $A[i]$ into the list $B[\lfloor n \cdot A[i] \rfloor]$	$\Theta(n)$
for $i \leftarrow 0$ to $n-1$ do	$\Theta(n)$
sort $B[i]$ with InsertionSort	$\sum_{i=0}^{n-1} \mathrm{O}(n_i^2)$
concatenate $B[0]$, $B[1]$, ..., $B[n-1]$ together in order	$\Theta(n)$

Complexity

It depends on the size n_i of the buckets $B[i]$ do sort.

Complexity of Bucket Sort

Best case
If each $B[i]$ has size $n_i = 1$, the n calls to InsertionSort at line 5 all cost $\Theta(1)$.
The complexity is $\Theta(n)$.

Worst case
If (1) all elements land in the same bucket, and (2) this bucket happens to be sorted in reverse order (worst case of InsertionSort) then line 5 costs $\Theta(n^2)$.
The final complexity is $\Theta(n^2)$.

Average case
What is n_i on the average? i.e. $E[n_i]$
What is n_i^2 on the average? i.e. $E[n_i^2]$
We eventually want to compute $\sum_{i=0}^{n-1} \mathrm{O}(E[n_i^2])$.

Binomial Distribution Reminder

Expected value of a random variable
It is its mean: $\mathrm{E}[X] = \sum_x x\Pr\{X = x\}$

Variance
$\mathrm{Var}[X] = \mathrm{E}[(X - \mathrm{E}[X])^2] = \mathrm{E}[X^2] - \mathrm{E}^2[X]$

Binomial Distribution
Throw n balls in r baskets, and assume balls have equal chances to land in each basket ($p = 1/r$). Let X_i denote the number of balls in basket i. We have $\Pr\{X_i = k\} = \binom{n}{k} p^k (1-p)^{n-k}$. It can be shown that $E[X_i] = np$ and $\mathrm{Var}[X_i] = np(1-p)$.

Probabilistic Study of Bucket Sort

Let n_i denote the size of a bucket to sort with Insertion Sort.

If the value to sort are uniformly distributed, they have equal chances to land in each bucket. It is like throwing n balls into n baskets (i.e. $p = 1/n$).

Therefore $E[n_i] = np = 1$ and $\mathrm{Var}[n_i] = np(1-p) = 1 - \frac{1}{n}$.

Insertion Sort of n elements takes $\mathrm{O}(n^2)$, so for all $B[i]$ we have

$$\sum_{i=0}^{n-1} \mathrm{O}(E[n_i^2]) = \mathrm{O}\left(\sum_{i=0}^{n-1} \mathrm{E}[n_i^2]\right) = \mathrm{O}\left(\sum_{i=0}^{n-1} (\mathrm{E}^2[n_i] + \mathrm{Var}[n_i])\right)$$
$$= \mathrm{O}\left(\sum_{i=0}^{n-1} \left(1 + 1 - \frac{1}{n}\right)\right) = \mathrm{O}(n)$$

Finally $T(n) = \mathrm{O}(n) + \Theta(n) = \Theta(n)$ on the average.

Data structures

You know of a couple of **data structures**, i.e., ways to organize data in memory to ease certain operations.

- Array
- Singly linked list

When you programmed last year, you probably used C++ container classes like `std::vector` (an array that can change its size) and list `std::list` (a doubly linked list).

As we saw with the dictionary lookup example, we can often choose between several data structures. The difference is in the operations they allow to perform, and the complexity of these operations.

Abstract Data Type

An abstract data type is a mathematical specification of a data set, and of a set of operations you can apply to this set. It is a contract that a data structure has to implement.

For instance the stack abstract data type that represents an ordered set allowing two operation

- **push** adds an item to the set in $\Theta(1)$,
- **pop** removes and returns the last item added to the set in $\Theta(1)$.

Such a abstract data type can be implemented using a singly linked list or array (Question: how would you implement these operations so that the complexity constraints are honored?)

Algorithms may be described using abstract data types, since such an abstractions precisely specify the expected behavior. The choice of the data structure used to implement the abstract data type can be delayed until the algorithm is actually implemented.

Some Data Structures to Represent a Set of Data

- Sequences:
  - array, vector
  - linked list (singly-, doubly-)
  - stack
  - queue
    - priority queue.
  - double entry queue (a.k.a deque)
- Associative array, search structures
  - hash table
  - self-balancing binary search tree
  - skip list

You pick a data structure according to the operations you need to execute on your data and the complexity of these operations for this data structures.

Some Operations on Sequences

$v \leftarrow$Access(S,k) Return the kth element.

$p \leftarrow$Search(S,v) Return a pointer (or index) to an element of S whose value is v.

Insert(S,x) Add element x to S.

Delete(S,p) Delete the element of S that is at position (pointer or index) p.

$v \leftarrow$Minimum(S) Return the minimum of S.

$v \leftarrow$Maximum(S) Return the maximum of S.

$p' \leftarrow$Successor(S,p) Return the position of the successor of (= the smallest value greater than) the element at position p in S .

$p' \leftarrow$Predecessor(S,p) Guess.

These are just some operations we will study for all data structures we present. Of course more operations exist (like sort, union, split, ...) and would deserve to be studied too.

Arrays

No need to present arrays

operation	unsorted array	sorted array
$v \leftarrow \text{Access}(S,k)$	$\Theta(1)$	$\Theta(1)$
$p \leftarrow \text{Search}(S,v)$	$O(n)$	$O(\log n)$
$\text{Insert}(S,x)$	$\Theta(1)$	$O(n)$
$\text{Delete}(S,p)$	$\Theta(1)$[4]	$O(n)$
$v \leftarrow \text{Minimum}(S)$	$\Theta(n)$	$\Theta(1)$
$v \leftarrow \text{Maximum}(S)$	$\Theta(n)$	$\Theta(1)$
$p' \leftarrow \text{Successor}(S,p)$	$\Theta(n)$	$\Theta(1)$
$p' \leftarrow \text{Predecessor}(S,p)$	$\Theta(n)$	$\Theta(1)$

[4] Since the order does not matter, we can replace the element that is deleted by the last element of the array.

Dynamic Arrays (1/2)

Array whose size can vary.

In C, we need to call `realloc()` when there is not enough unused entries left for insertion. In C++, `std::vector` will perform `realloc()` by itself. Reallocating an array requires $\Theta(n)$ time, since it has to be copied. Inserting in an array, usually is a $\Theta(1)$ operation, becomes $\Theta(n)$ if a reallocation is required.

You do not want to reallocate an array just to add one entry, because insertion would then cost $\Theta(n)$ every time. What is a suitable reallocation scheme?

We can study the **amortized complexity** of an insertion in a sequence of insertions.

Dynamic Arrays (2/2)

Let us consider the case of an insertion that leads to reallocation, with two different ways to enlarge the array:

Add k new entries. There will be a reallocation every k insertions, so the average cost of the last k insertions is

$$\frac{(k-1)\Theta(1) + 1\Theta(n)}{k} = \Theta(n)$$

Double the size. Since the last reallocation there have been $n/2 - 1$ insertions in $\Theta(1)$ followed by one insertion in $\Theta(n)$. The average cost of the last $n/2$ insertions is

$$\frac{(n/2-1)\Theta(1) + 1\Theta(n)}{n/2} = \frac{\Theta(n)+\Theta(n)}{n} = \Theta(1)$$

We say that insertion is in amortized $\Theta(1)$ when the operation is usually in $\Theta(1)$, and the slow cases are infrequent enough so that their cost can be amortized on the fast cases.

Lists

We distinguish between singly linked lists (where only the next item is known) and doubly linked lists (where a predecessor is also known).

	S.L.L.		D.L.L.	
operation	unsorted	sorted	unsorted	sorted
$v \leftarrow \text{Access}(S,k)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
$p \leftarrow \text{Search}(S,v)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
$\text{Insert}(S,x)$	$\Theta(1)$	$O(n)$	$\Theta(1)$	$O(n)$
$\text{Delete}(S,p)$	$O(n)$[1]	$O(n)$[1]	$\Theta(1)$	$\Theta(1)$
$v \leftarrow \text{Minimum}(S)$	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
$v \leftarrow \text{Maximum}(S)$	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
$p' \leftarrow \text{Successor}(S,p)$	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
$p' \leftarrow \text{Predecessor}(S,p)$	$\Theta(n)$	$O(n)$	$\Theta(n)$	$\Theta(1)$

[1] In practice the deletion can be in $\Theta(1)$ if you know the previous element somehow.

Stack and Queues

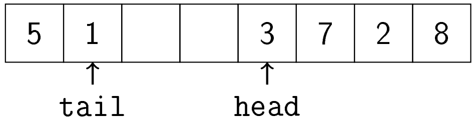
Stack Sequence in which insertions and deletions are always done at the same end. Insert() and Delete() are usually called Push() and Pop(). LIFO = Last In First Out
- Usually implemented on top of an array or singly linked list.

Queue Sequence in which insertions and deletions are done at opposite ends. Insert() and Delete() are usually called Push() and Pop(). FIFO = First In First Out
- Usually implemented on top of a singly linked list with tail pointer. Insert at tail in $\Theta(1)$, delete at head with $\Theta(1)$.

Double ended queue (deque) Insertion and deletion can be done at both ends.
- Can be implemented using a doubly linked list. Insertion and deletion in $\Theta(1)$.

Bounded Queues and Circular Arrays

If the size of a queue (simple or double ended) is bounded it can be implemented efficiently using a "circular array".



In that case, access to the kth element can be done in $\Theta(1)$ instead of $\Theta(n)$ with a list.

$$\text{Access}(S,k) = A[(head + k - 1) \mod n]$$

The counterpart is that, Insert() and Erase() at position r become $\mathrm{O}(\min(r, n-r))$ instead of $\mathrm{O}(1)$.

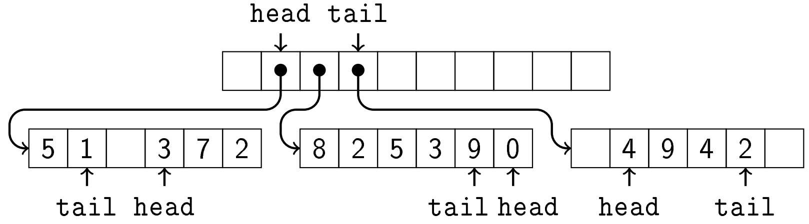
How can we extend this circular scheme to unbounded queues?

Unbounded Queues and Circular Arrays

How to add an entry to a circular array that is full?

1st idea Augment the size of the array. In practice: new memory allocation then copy. The insertion becomes in $\Theta(n)$ when it happens. Complexity stays in amortized $\Theta(1)$ with proper growth.

2nd idea Make a dynamic circular array of dynamic arrays that have constant size. Only the arrays at both ends are not full.



It is again amortized $\Theta(1)$ because we sometimes (but less often) have to reallocate the master array. This is the implementation of the `std::deque` container in C++.

Priority Queues

The element removed from a priority queue is always the greatest. Some say "*Largest In, First Out*" (but do not mistake with "LIFO = *Last In First Out*").

- If the priority queue is done with a sorted list, then Push() is in $\mathrm{O}(n)$ and Pop() in $\Theta(1)$.
- If the priority queue is done with a heap, then Push() is in $\mathrm{O}(\log n)$ and Pop() in $\mathrm{O}(\log n)$.

  (To Pop() a heap: like in Heap Sort you remove the first value of the heap, replace it by the last, and call Heapify to fix the heap structure.)

  Can you explain how to do Push() on a heap?

Push for Heap

Input: An array $A[1..m]$ with heap property, a value v to insert,
Output: An array $A[1..m+1]$ with heap property and containing v.

```
HeapPush(A, m, v)
  i ← m + 1
  A[i] ← v
  while i > 1 and A[Parent(i)] < A[i] do
      A[Parent(i)] ↔ A[i]
      i ← Parent(i)
```

In the worst case the number of operations is proportional to the height of the heap, so $T(n) = \mathrm{O}(\log n)$.

Associative Array

An associative array (or dictionary or map) is an abstract data type that can be seen as a generalization of arrays for non-consecutive indices. Because these indices may not be integers, we call these keys. (These keys can be associated to auxiliary data as in sorting algorithms.)

Typical operations:

- Adding
- Deleting
- Searching (by key).
- Updating (of auxiliary data).

In the sequel we shall not show the auxiliary data for simplicity, but you have to assume that they follow the key every time it is copied (but they are not used during comparisons).

Hash Table

Goal: represent a set $\mathcal{F}$ of items (the keys), let's say a subset of a domain $\mathcal{K}$. We want to quickly test membership to this subset.

If $\mathcal{K} = \mathbb{N}$, we can use an array to represent $\mathcal{F}$. Assuming 0-based array we then have $n \in \mathcal{F}$ iff $A[n] \neq 0$.
However if $\max(\mathcal{F})$ is big this array will take a lot of place even if $|\mathcal{F}|$ is small.
Furthermore, this scheme won't work if $\mathcal{K}$ does not represent integers.

Idea: for any $\mathcal{F} \subseteq \mathcal{K}$, let's find a function $f : \mathcal{K} \mapsto \{0, \ldots, m\}$ so we can then test set-membership as follows: $x \in \mathcal{F}$ iff $A[f(x)] \neq 0$.
f has to be injective for this test to be correct (and such function f can exist only if $m - 1 \geq |\mathcal{K}|$).

These membership tests are in $\Theta(1)$ if f is simple.

Injectivity in $\mathcal{K}$ or $\mathcal{F}$

Given $\mathcal{F} = \{$"chat", "chien", "oie", "poule"$\}$ to map to $\{0, \ldots, 30\}$. Let's take the function $f(mot) = (mot[2] - \texttt{'a'})$.

This functions distinguishes words of $\mathcal{F}$ by their third letter.
$f(\texttt{chat}) = 0$
$f(\texttt{chien}) = 8$
$f(\texttt{oie}) = 4$
$f(\texttt{poule}) = 20$
It is not injective in $\mathcal{K}$:
$f(\texttt{loup}) = 20$
A solution is to represent the key in the array. Then $x \in \mathcal{F}$ iff $A[f(x)] = x$.
If $m \geq |\mathcal{F}|$ we can find an injective function in $\mathcal{F}$.

i	$A[i]$
0	chat
1	/
2	/
3	/
4	oie
⋮	/
8	chien
⋮	/
20	poule
⋮	/

Dearth of Injective Functions

Such injective function f can hardly be found by luck.

Let $\mathcal{F}$ be a set of $n = 30$ items that we want to represent in an array of $m = 40$ entries.

There are $40^{30} \approx 10^{48}$ functions from $\mathcal{F}$ to $\{0, \ldots, m-1\}$. Among these functions, only $40 \cdot 39 \cdots 11 = 40!/10! \approx 2.10^{41}$ are injective. We therefore have one against 5 million chances to pick an injective function at random.

Another typical example of the dearth of injective functions is the birthday problem: with 23 people, the probability that two people are born on the same day is more than 1/2. Still the *birthday* function offers 365 possible choices!

http://en.wikipedia.org/wiki/Birthday_problem

GPerf

When the set $\mathcal{F}$ is known beforehand, it is possible to find (algorithmically) a function f that maps $\mathcal{F}$ to $\{0, \ldots, m\}$ with $m - 1 \geq \mathcal{F}$ and without collision. Such a function is called a perfect hashing function, and it is minimal if $m - 1 = |\mathcal{F}|$.

The purpose of the tool GNU `gperf` is to find such functions. It inputs a list of words to recognize, a value m, and produces a C file containing a function f and an array A with the words supplied at the right place (i.e. such that $A[f(w)] = w$ for any word w).

Other tools exist for the same task, e.g. CMPH (C Minimal Perfect Hashing Library).

Hashing with Chaining

When a perfect hashing function is not available (either m is too small, or $\mathcal{F}$ is always updated) two elements can be hashed to the same index and we have to deal with a collision.

As with bucket sort, the easiest way is to keep the list of possible values for each index in the array.

E.g. $\mathcal{F} = \{$`"chat"`, `"chien"`, `"oie"`, `"poule"`, `"loup"`$\}$.

Then $x \in \mathcal{F}$ iff $Search(A[f(x)], x) \neq 0$.
This membership test is no longer $\Theta(1)$ because you have to search the list. In the worst case the size of this list is $n = |\mathcal{F}|$. In the best case you hope for n/m items on the list.

i	$A[i]$
0	`chat`
1	/
2	/
3	/
4	`oie`
$\vdots$	/
8	`chien`
$\vdots$	/
20	`poule`, `loup`
$\vdots$	/

Chaining with Uniform Hashing

Uniform hashing is when you assume that f spreads keys uniformly in $\{0, \ldots, m-1\}$. (You have to make some hypotheses about the distribution law in order to compute average complexity.)

For uniform hashing, search is in $\Theta(1 + n/m)$.

If you can arrange for the array size m to be proportional to the number of keys n, then $n/m = O(1)$ and search is in $\Theta(1)$.
Insertion is then also in **amortized** $\Theta(1)$ and deletion in $\Theta(1)$.
Reallocating the array requires to change the hashing function (since m is changing) and to move all elements around to their new place.

Hashing Function Example: Division

$f(x) = x \mod m$
Avoid a power of two such as $m = 256$ because it amounts to ignoring the 8 lower bits of x, often those that change the most. A common suggestion for m is a prime numbers away from powers of two.
For instance to represent 3000 items with an average of 2 items per list, you may choose $m = 1543$.
In an implementation of hashing table using this methode, you usually find an hard-coded list of prime numbers to use. For instance if you plan to double the size of the array during reallocation, you may use the following prime number list for the successive values of m: 53, 97, 193, 389, 769, 1543, 3079, 6151, 12289, 24593, 49157, 98317, 196613, 393241, etc.
C++'s `std::hash_map` class uses this list and a hashing function $f(x) = g(x) \mod m$ where g is given by user (to convert anything to an integer) and m is a prime number from this list.

Open Addressing

A compact hashtable encoding where all items are stored in the array, without list nor pointers. Collisions are handled without chaining, but you have to probe several locations until you find the right one.

In open adressing the hashing function $f(x,i)$ takes a value x and an iteration number i.

To insert x in the table first check if $A[f(x,0)]$ is free, otherwise try $A[f(x,1)]$, then $A[f(x,2)]$, etc. We say we probe different positions. Function f should be such that $f(x,i)$ covers the entire range $\{0,\ldots,m-1\}$ when i covers $\{0,\ldots,m\}$. The order should depend on key x.

Searching can be done in the same way until the value or an empty entry is found.

Beware while deleting: why cannot you empty the entry?

Hashing Functions for Open Addressing

Linear probe $h(x,i) = (h'(x)+i) \mod m$
Problem: a sequence of occupied entries tends to grows, making the search longer.

Quadratic proble $h(x,i) = (h'(x)+c_1 i + c_2 i^2) \mod m$
It is better, but here again the first probe determines the entire sequence: $h(x,0) = h(x',0) \implies h(x,i) = h(x',i)$.

Double hachage $h(x,i) = (h_1(x) + ih_2(x)) \mod m$
This time $h(x,0) = h(x',0) \not\Rightarrow (x,i) = h(x',i)$.

Search Complexity in Open Addressing

The number of probes during an unsuccessful search is $1/(1-n/m)$ assuming uniform hashing (i.e., all probe sequence over $\{1,\ldots,m\}$ are assumed to appear with the same probability).

Insertion also requires $1/(1-n/m)$ probes on this average.

If n/m is constant, we conclude that insertion, search, and deletion are in $\Theta(1)$. In practice, m is of course not changed as often as n.

The point of open addressing is to get rid of pointers. This saves memory, allowing to store larger table. The counterpart is that it is slightly slower.

Critical Size of Hash Tables

The birthday problem tells us that if a hash table can represent N entries, the number of elements to insert to get collisions with probability p is

$$n(p,N) \approx \sqrt{2N \ln\left(\frac{1}{1-p}\right)}$$

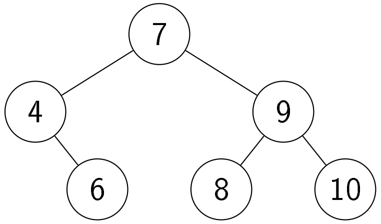
Let us just remember a simplified form:

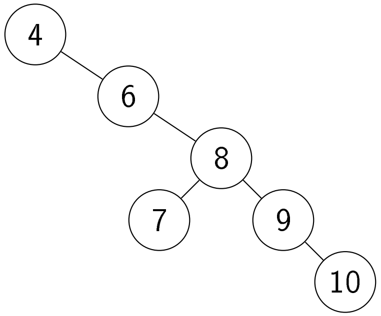
$$n(0.5,N) \approx 1.177\sqrt{N} = \Theta(\sqrt{N})$$

In other words, after $\sqrt{N}$ values in a hash table, there is $1/2$ chances that there is a collision somewhere in the table.

Binary Search Trees

A binary tree whose nodes are labelled is a search tree if for any node r labelled by v, all labels from the left subtree are $\leq v$ and all labels from the right subtree are $\geq v$.





All *BST* are not balanced. The complexity of search is $O(h)$ where h is the height of the tree. Finding minimum and maximum is also $O(h)$.

Infix Order

Traversing the tree in infix order makes it possible to visit all keys in increasing order.

```
InfixPrint(T, z)
  if LeftChild(z) ≠ NIL then
      InfixPrint(T, LeftChild(z))
  print key(z)
  if RightChild(z) ≠ NIL then
      InfixPrint(T, RightChild(z))
```

```
PrefixPrint(T, z)
  print key(z)
  if LeftChild(z) ≠ NIL then
      InfixPrint(T, LeftChild(z))
  if RightChild(z) ≠ NIL then
      InfixPrint(T, RightChild(z))
```

```
SuffixPrint(T, z)
  if LeftChild(z) ≠ NIL then
      InfixPrint(T, LeftChild(z))
  if RightChild(z) ≠ NIL then
      InfixPrint(T, RightChild(z))
  print key(z)
```


Insertion in BST is Easy

Input: a BST T and a node z to insert
Output: the BST T updated to include z

```
TreeInsert(T, z)
  y ← NIL
  x ← Root(T)
  while x ≠ NIL do
      y ← x
      if key(z) < key(x)
          then x ← LeftChild(x)
          else x ← RightChild(x)
  Parent(z) ← y
  if y = NIL then
      Root(T) ← z
  else
      if key(z) < key(y)
          then LeftChild(y) ← z
          else RightChild(y) ← z
```

$$T(h) = O(h)$$

Deletion (1/2)

Three cases to consider:

- Deleting a leave is easy.
- Deleting a node with one child: easy too.
- Deleting a node with two children is harder: we should replace the node by its successor, i.e. the minimum of the right tree (that has to be deleted).

Deletion (2/2)

```
TreeDelete(T, z)
x ← NIL
if LeftChild(z) = NIL or RightChild(z) = NIL
    then y ← z
    else y ← TreeSuccessor(z)
if LeftChild(y) ≠ NIL
    then x ← LeftChild(y)
    else x ← RightChild(y)
if x ≠ NIL then Parent(x) ← Parent(y)
if Parent(y) = NIL then
    Root(T) ← x
else
    if y = LeftChild(Parent(y))
        then LeftChild(Parent(y)) ← x
        else RightChild(Parent(y)) ← x
if y ≠ z then key(z) ← key(y)
```


Complexity of BST Operations

Insert, Delete, Search, Predecessor, Successor, Minimum and Maximum all run in $\mathrm{O}(h)$ and

$$\underbrace{\lfloor \log n \rfloor}_{\text{balanced}} \leq h \leq \underbrace{n}_{\text{unbalanced}}$$

So all these algorithms are in $\mathrm{O}(n)$...

However it can be shown that the average height of a randomly constructed BST is in $\Theta(\log n)$.

It would be best to modify these Insert() and Delete() operations so that they preserve the balancing of the tree. Such a tree is called a self-balancing tree.

Read-Black Trees

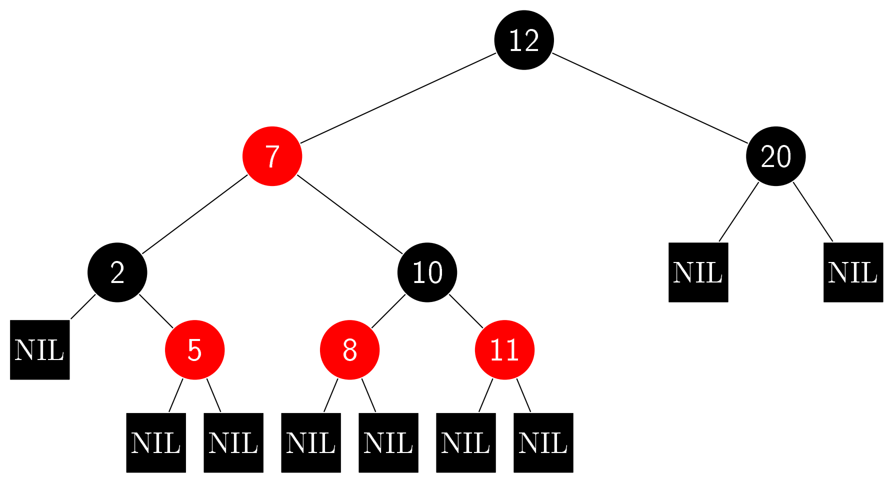
RBT are self-balancing trees in which each node has a bit indicating its color: red or black. Some constraints on colors ensure that the longest branch of the tree is at most twice longer that the smallest branch. (This is balanced enough to ensure $h = \Theta(\log n)$)

Here are the constraints:

- A node is either black or red
- Root and leaves (NIL) are black
- The two children of a red node are black
- All paths leaving a node down to a leave have the same number of black nodes

The black height of a node x, denoted $bh(x)$ is the number of black nodes between x (excluded) and a leave in its descendants (included).

Example of RBT



Property

An RBT with n internal nodes has a height of at most $\lfloor 2\log(n+1) \rfloor$.

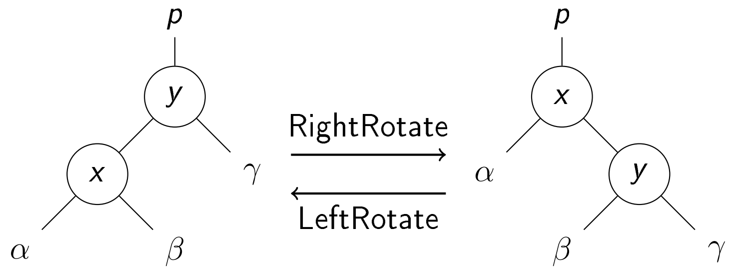
- If you ignore red nodes in the tree, each black node has between 2 and 4 black children, and all branches have the same height h'.
- The height for the complete tree is $h \leq 2h'$ because there cannot be more red nodes than black nodes on a branch.
- The number of leaves on the tree is $n+1$.
  Therefore

$$n+1 \geq 2^{h'} \quad \Longrightarrow \quad \log(n+1) \geq h' \geq h/2 \quad \Longrightarrow \quad h \leq 2\log(n+1)$$

Furthermore the minimal size of a branch is $\log(n+1)$ (half the height). Consequently, Search, Minimum, Maximum, Successor and Predecessor are all in $\Theta(\log n)$. It is not as obvious for Insert et Delete.

Rotations

Insertion with `TreeInsert` do not preserve RBT properties. We can fix this by changing the colors, and performing local rotations in the tree.



These rotations can be applied to any BST: they preserve the infix order.

Left Rotation

LeftRotate(T, x)

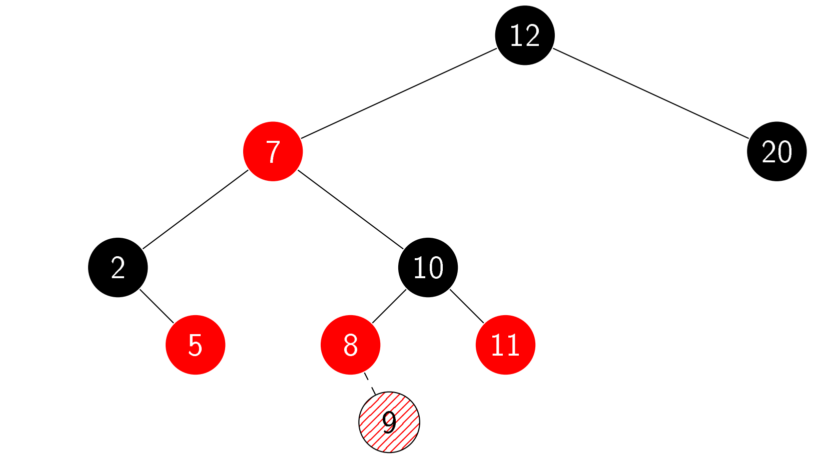
$y \leftarrow RightChild(x)$
$\beta \leftarrow LeftChild(y)$
$RightChild(x) \leftarrow \beta$
if $\beta \neq \mathrm{NIL}$ then $Parent(\beta) \leftarrow x$
$p \leftarrow Parent(x)$
$Parent(y) \leftarrow p$
if $p = \mathrm{NIL}$
    then $Root(T) \leftarrow y$
    else if $x = LeftChild(p)$
        then $LeftChild(p) \leftarrow y$
        else $RightChild(p) \leftarrow y$
$LeftChild(y) \leftarrow x$
$Parent(x) \leftarrow y$

$$T(n) = \Theta(1)$$

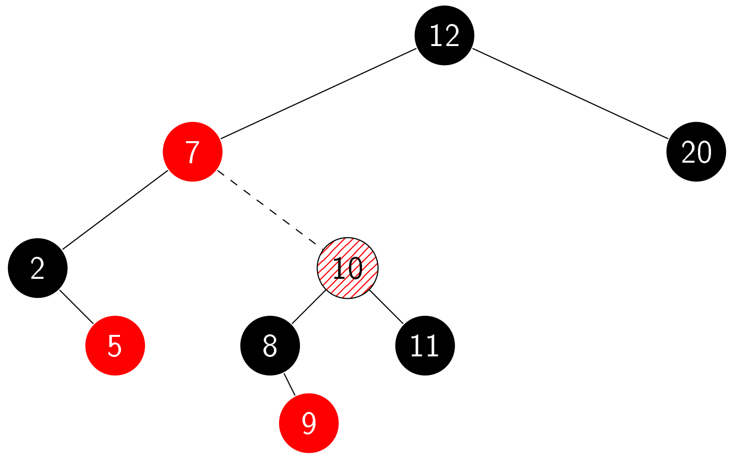
Insertion in a RBT

- Insert the node in the tree as if it was a BST, and give it the red color. The property "the two children of a red node are black" might be violated.
- Fix the violation by recoloring the parents, and moving the problem up until it can be fixed by one or two rotations.

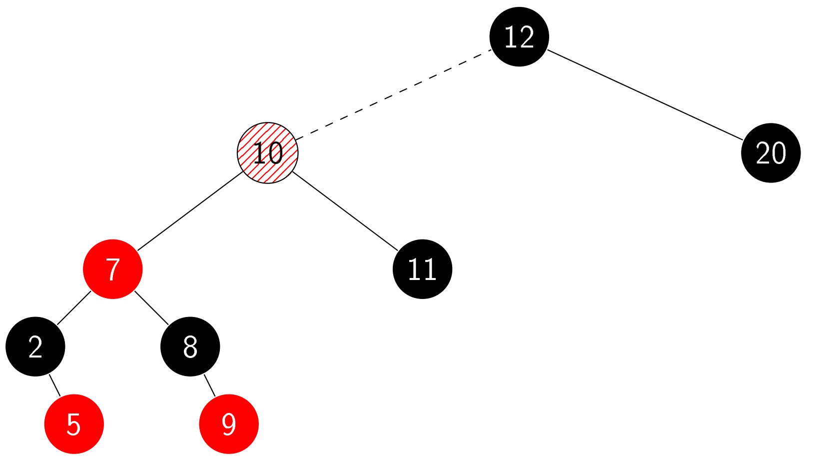
Insertion of 9



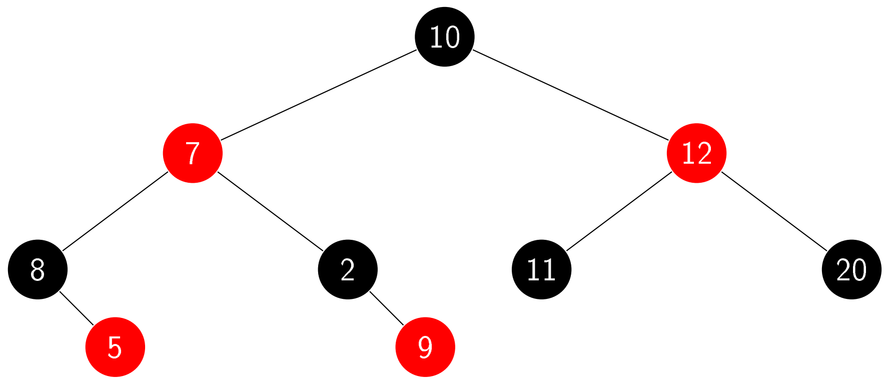
Insertion of 9



Insertion of 9

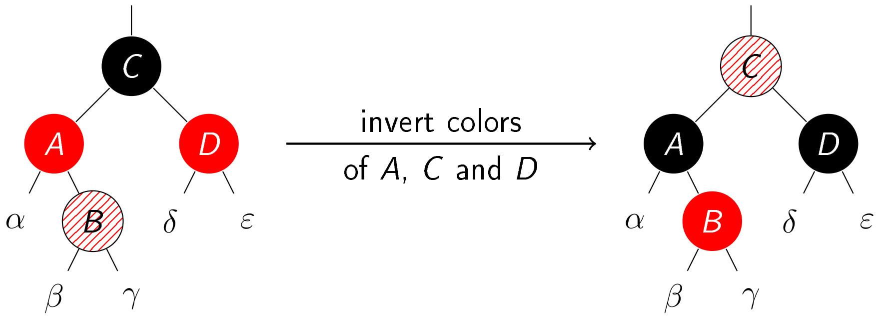


Insertion of 9



Three Cases to Handle: Case 1

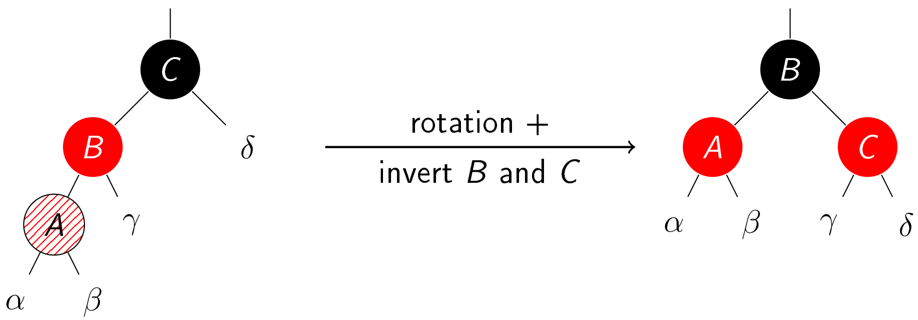
Father and uncle are both red.
(In all cases, Greek letter represent subtrees with the same back height.)



Continue from grandfather if the grand-grandfather is red.

Three Cases to Handle: Case 3

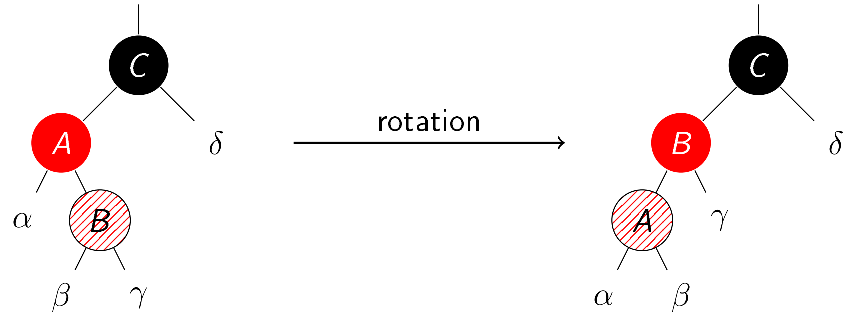
The father is red, the uncle is black, and the current node is aligned with the axe father–grandfather.



After this correction the tree is fixed and follows the RBT constraints.

Three Cases to Handle: Case 3

The father is red, the uncle is black, and the current node is not aligned with the axe father–grandfather.



A rotation can align son, father, and grandfather. The problem isn't fixed, but it has been transformed into "case 3".

Complexity of Insertion

- Add the (red) node with TreeInsert. $\Theta(h)$
- Apply case 1 at most $h/2$ times. $O(h)$
- Apply case 2 at most once. $O(1)$
- Apply case 3 at most once. $O(1)$

Finally we did $\Theta(h) = \Theta(\log n)$ operations.

RBTreeInsert

```
RBTreeInsert(T, z)
  TreeInsert(T, z)
  Color(z) ← red
  while Color(Parent(z)) = red do
    if Parent(z) = LeftChild(Parent(Parent(z))) then
      uncle ← RightChild(Parent(Parent(z)))
      if Color(uncle) = red then
        Color(Parent(z)) ← black
        Color(uncle) ← black                      case 1
        z ← Parent(Parent(z))
        Color(z) ← red
      else if z = RightChild(Parent(z)) then
        z ← Parent(z)                             case 2
        LeftRotate(T, z)
      else
        Color(Parent(z)) ← black
        Color(Parent(Parent(z))) ← red            case 3
        RightRotate(T, Parent(Parent(z)))
    else like "then", but swap "Left" and "Right"
  Color(Root(T)) ← black
```


RBT: Deleting

RBTreeDelete() can also be done in $\Theta(\log n)$.
If the node to delete is red, TreeDelete can be used.
If it is black, the tree will have to be fixed in a way similar to insertion (but with 4 cases to consider instead of 3).

Conclusion on RBT

Red-Black Trees can execute all the following operations in $\Theta(\log n)$:

- Insert
- Delete
- Minimum
- Maximim
- Successor
- Predecessor

Furthermore Search is in $\mathrm{O}(\log n)$.

On advantage over hash table is that the elements are sorted in the structure: it is possible to output them in order with $\Theta(n)$ operations.

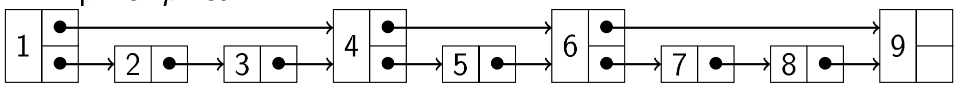
This data structure is used by `std::map` in C++.

Skip list

A generalization of the sorted list.

A probabilistic structure, with the same average complexity as RBT (i.e. $\Theta(\log n)$ on the average for all operations), and easier to implement.

Example *skip list* with two levels:

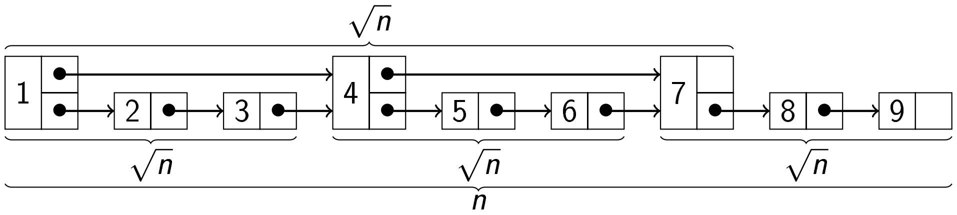


First locate the interval of the element you are looking for in the first list, than go down one level to refine the search.

Two-Level Skip list

Where should we connect the two levels ?

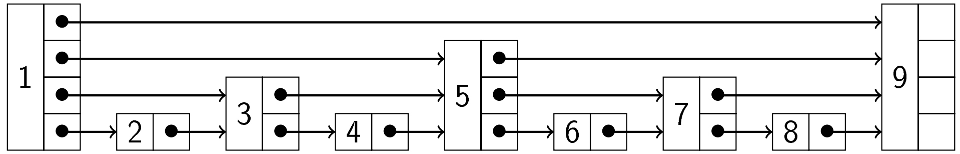
We want regular spacing, but with wich stride?
Let t_1 and t_2 denote the sizes of both lists. Searching an element in the *skip list* is in $O(t_1 + t_2/t_1)$. This sum is minimal when the two terms are equal: $t_1 = t_2/t_1$ in other words $t_1 = \sqrt{t_2}$.



The cost of searching is then proportional to $2\sqrt{n} = O(\sqrt{n})$.

Skip List With More Levels

- 2 lists: $2 \cdot \sqrt{n}$
- 3 lists: $3 \cdot \sqrt[3]{n}$
- k lists: $k \cdot \sqrt[k]{n}$
- $\log n$ lists: $\log n \cdot \sqrt[\log n]{n} = \log n \cdot e^{\frac{1}{\log n}\ln n} = \log n \cdot e^{\ln 2} = 2\log n$



Above example is an ideal skip list: searchs are always in $\Theta(\log n)$.

How to perform an insertion?

Insertion in a *skip list*

To insert x in a *skip list*.

- Make a search to find the place to insert x (in the lower level)
- Insert x in the lower list (level 0).
- With probability $1/2$, add x to the above list (level 1).
- If x was added to level 1, add it to level 2 with probability $1/2$.
- Repeat until destiny says to stop, or you reach the top list.

Eventually x appears

- at level 0 with probability 1
- at level 1 with probability $1/2$
- at level 2 with probability $1/4$
- at level k with probability 2^{-k}

Analysis of the Complexity of Search (1/2)

We evaluate the cost of Search by counting the moves backwards from the end: from level 0 we have to go back to level k, moving left if you cannot move up.

Let $C(k)$ denote the cost of moving up k levels, and let $p = 1/2$ be the probability that there is a level above the current node.
Two cases may occur:

- with probability p we go up one level, and there are $k - 1$ levels left to climb (cost: $1 + C(k-1)$ moves)
- with probability $1 - p$ we cannot move up: we move left once and there are still k levels to climb (cost: $1 + C(k)$ moves)

We can thus write:

$$C(0) = 0$$
$$C(k) = (1-p)(1 + C(k)) + p(1 + C(k-1))$$

Analysis of the Complexity of Search (2/2)

$$C(0) = 0$$
$$C(k) = 1/p + C(k-1) = k/p$$

This is an upper bound for moving up n levels, because if we reach the head of the list before reaching the top-level the probability to move up becomes 1.

Let $L(n)$ denote the height of a *skip list* of n items. The cost to move to the last level is $C(L(n) - 1)$.

In our case, $L(n) = \log n$. Thus we have

$$T(n) = \mathrm{O}\left(\frac{(\log n) - 1}{p}\right) = \mathrm{O}(\log n)$$

Complexity of Search Data Structures

operation	hash table	RBT	skip list
$p \leftarrow$Search(S, v)	$\Theta(1)$ avg.	$\mathrm{O}(\log n)$	$\mathrm{O}(\log n)$ avg.
Insert(S, x)	$\Theta(1)$ am.	$\Theta(\log n)$	$\mathrm{O}(\log n)$ avg.
Delete(S, p)	$\Theta(1)$ am.	$\Theta(\log n)$	$\Theta(1)$
$v \leftarrow$Minimum(S)	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$
$v \leftarrow$Maximum(S)	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$
$p' \leftarrow$Successor(S, p)	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$
$p' \leftarrow$Predecessor(S, p)	$\Theta(n)$	$\Theta(\log n)$	$\mathrm{O}(\log n)$ avg. or $\Theta(1)$

am. = amortized; avg. = on average.