Algorithms and data structures

(adapted from Prof. Scott Sanner)

Prof. Dionne Aleman

MIE250: Fundamentals of object-oriented programming
University of Toronto

Note

Complexity, sorting, and hashing are covered in MIE335, so these topics will be covered very quickly just to give you an idea of what is coming in the future. More detailed coverage is in the pre-recorded lecture videos.

What is an algorithm?

- "...a finite sequence of well-defined, computer-implementable instructions, typically to solve a class of problems or to perform a computation." [Wikipedia, 2020]

- How do we measure the quality of an algorithm?
  - Accuracy
  - Computational complexity (speed)

- Speed is often determined by data structures, which is why algorithms often cannot be separated from the data structures used to implement them.

How efficient is your code?

- We can empirically measure speed by profiling
  - See classes that dominate memory usage
  - See methods that dominate runtime
- Great tools for IDEs
  - Netbeans built-in profiler: `https://profiler.netbeans.org/`
  - JVM Monitor for Eclipse: `http://www.jvmmonitor.org/doc/index.html`
- Profilers that examine running code
  - Valgrind: `https://valgrind.org/`
  - Mac's built-in Time Profiler in the Instruments program
- Let's see Java's own built-in JVisualVM in action.

Empirical analysis: A fair comparison?

- Hardware: processor(s), memory, cache, etc.
- OS, version of Java, libraries, drivers
- Programs running in the background
- Implementation dependent
- Choice of input
- Which inputs to test

Algorithm analysis

- As the "size" of an algorithm's input grows:
  - Time: How much longer does it run?
  - Space: How much memory does it use?

Algorithm analysis

- As the "size" of an algorithm's input grows:
  - Time: How much longer does it run?
  - Space: How much memory does it use?

- How do we answer these questions?
  - For now, we will focus on time only.

In general

- Evaluating an implementation? Use empirical timing.
- Evaluating an algorithm? Use asymptotic analysis (complexity analysis).

Complexity assumptions

- Basic operations take constant time
  - Arithmetic
  - Assignment
  - Access of one Java field or array index
  - Comparison of two simple values (e.g., is $x < 3$?)
- Other operations are summations or products
- Consecutive statements are summed
- Loops are (cost of loop body) $\times$ (number of loops)
- What about conditionals?

Worst-case analysis

- In general, we are interested in three types of performance:
  - Best-case (fastest)
  - Average-case
  - Worst-case (slowest)

- When determining worst-case, we tend to be pessimistic:
  - If there is a conditional, count the branch that will run the slowest.
  - This approach will give a loose bound on how slow the algorithm may run.

Examples

Code	Number of steps
`for (int i = 0; i < n; i++)` `    x = x + 1;`	

`for (int i = 0; i < n; i++)`
`    for (int j = 0; j < n; j++)`
`        x = x + 1;`	

`for (int i = 0; i < n; i++)`
`    for (int j = 0; j <= i; j++)`
`        x = x + 1;`	

Examples

Code	Number of steps
`for (int i = 0; i < n; i++)` `    x = x + 1;`	$\approx 4n$

`for (int i = 0; i < n; i++)`
`    for (int j = 0; j < n; j++)`
`        x = x + 1;`	

`for (int i = 0; i < n; i++)`
`    for (int j = 0; j <= i; j++)`
`        x = x + 1;`	

Examples

Code	Number of steps
`for (int i = 0; i < n; i++)` `    x = x + 1;`	$\approx 4n$

`for (int i = 0; i < n; i++)`
`    for (int j = 0; j < n; j++)`
`        x = x + 1;`
	$\approx 4n^2$

`for (int i = 0; i < n; i++)`
`    for (int j = 0; j <= i; j++)`
`        x = x + 1;`	

Examples

Code	Number of steps
`for (int i = 0; i < n; i++)` `    x = x + 1;`	$\approx 4n$

`for (int i = 0; i < n; i++)`
`    for (int j = 0; j < n; j++)`
`        x = x + 1;`
	$\approx 4n^2$

`for (int i = 0; i < n; i++)`
`    for (int j = 0; j <= i; j++)`
`        x = x + 1;`
	$\approx 4(1 + 2 + \ldots + n)$

Examples

<table>
<tr><th>Code</th><th>Number of steps</th></tr>
<tr><td><pre>for (int i = 0; i < n; i++)
    x = x + 1;</pre></td><td>$\approx 4n$</td></tr>
<tr><td><pre>for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        x = x + 1;</pre></td><td>$\approx 4n^2$</td></tr>
<tr><td><pre>for (int i = 0; i < n; i++)
    for (int j = 0; j <= i; j++)
        x = x + 1;</pre></td><td>$\approx 4(1 + 2 + \ldots + n)$
$\approx 4(n(n+1)/2)$</td></tr>
</table>

Examples

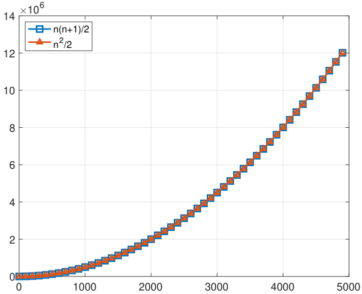
Code	Number of steps
`for (int i = 0; i < n; i++)` `    x = x + 1;`	$\approx 4n$

`for (int i = 0; i < n; i++)`
`    for (int j = 0; j < n; j++)`
`        x = x + 1;`
	$\approx 4n^2$

`for (int i = 0; i < n; i++)`
`    for (int j = 0; j <= i; j++)`
`        x = x + 1;`
	$\approx 4(1+2+\ldots+n)$
	$\approx 4(n(n+1)/2)$
	$\approx 2n^2+2n+2$

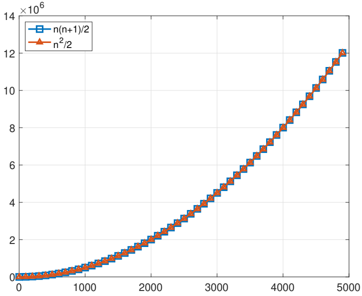
No need to be so exact

- Constants do not matter
  - Consider $6N^2$ and $20N^2$
  - When $N >> 20$, the N^2 is what drives the function's increase
- Lower-order terms are also less important
  - $N(N+1)/2$ v. just $N^2/2 \rightarrow$ linear term is inconsequential



No need to be so exact

- Constants do not matter
  - Consider $6N^2$ and $20N^2$
  - When $N >> 20$, the N^2 is what drives the function's increase
- Lower-order terms are also less important
  - $N(N+1)/2$ v. just $N^2/2 \rightarrow$ linear term is inconsequential



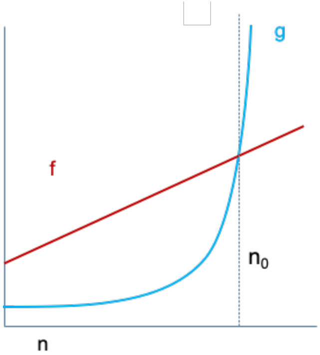
We need a better notation for performance that focuses on dominant terms only

Big O notation

Given two functions $f(n)$ and $g(n)$ for input n, we say $f(n)$ is in $O(g(n))$ ("order $g(n)$") iff there exist positive constants c and n_0 such that

$$f(n) \leq cg(n) \quad \forall n \geq n_0$$

Eventually, $g(n)$ is always an upper bound on $f(n)$ ignoring constants.



Big O only cares about big ticket items

- $5n + 3 \rightarrow O(n)$ Linear time
- $7n + 0.5n^2 + 2000 \rightarrow O(n^2)$ Quadratic time
- $2 \log n + 3 \rightarrow O(\log n)$ Logarithmic time
- $300n + 12 + n \log n \rightarrow O(n \log n)$ Linearithmic time (or quasi-linear or "n log n")
- $7n + 0.5n^2 + 2000 + 2^n \rightarrow O(2^n)$ Exponential time
- Note:
  - Cannot reduce exponents (n^3 is not $O(n^2)$)
  - Cannot reduce exponent bases (3^n is not $O(2^n)$)

Big O examples

Statement	True or false?
$4 + 3n$ is $O(n)$	

Big O examples

Statement	True or false?
$4 + 3n$ is $O(n)$	True

Big O examples

Statement	True or false?
$4 + 3n$ is $O(n)$	True
$n + 2\log n$ is $O(\log n)$	

Big O examples

Statement	True or false?
$4 + 3n$ is $O(n)$	True
$n + 2\log n$ is $O(\log n)$	False

Big O examples

Statement	True or false?
$4 + 3n$ is $O(n)$	True
$n + 2 \log n$ is $O(\log n)$	False
$\log n + 2$ is $O(1)$	

Big O examples

Statement	True or false?
$4 + 3n$ is $O(n)$	True
$n + 2\log n$ is $O(\log n)$	False
$\log n + 2$ is $O(1)$	False

Big O examples

Statement	True or false?
$4 + 3n$ is $O(n)$	True
$n + 2\log n$ is $O(\log n)$	False
$\log n + 2$ is $O(1)$	False
n^{50} is $O(1.1^n)$	

Big O examples

Statement	True or false?
$4 + 3n$ is $O(n)$	True
$n + 2\log n$ is $O(\log n)$	False
$\log n + 2$ is $O(1)$	False
n^{50} is $O(1.1^n)$	True (exponential always beats exponent, but is a loose upper bound)

Common Big O categories

From fastest to slowest

Category	Name
$O(1)$	constant (or $O(k)$ for constant k)
$O(\log n)$	logarithmic
$O(n)$	linear
$O(n \log n)$	linearithmic, quasi-linear, "n log n"
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(n^k)$	polynomial (where is k is constant)
$O(k^n)$	exponential (where constant $k > 1$)

Topics

- Searching
- Sorting
- Set and Map operations
- Decidability: do all algorithms terminate?

Searching and sorting

Searching

An ArrayList is implemented with an array:

32	87	17	79	95	76	24	1	23

- Complexity of `contains(Object o)` for array of length n?

Searching

An ArrayList is implemented with an array:

32	87	17	79	95	76	24	1	23

- Complexity of `contains(Object o)` for array of length n?
- Linear $O(n)$: Go through indices until
  - `Object o` is found $\rightarrow$ true
  - Reach the end $\rightarrow$ false

Searching a sorted list

An ArrayList is implemented with an array:

1	17	23	24	32	76	79	87	95

- Complexity of `contains(Object o)` for array of length n?

Searching a sorted list

An ArrayList is implemented with an array:

1	17	23	24	32	76	79	87	95

- Complexity of `contains(Object o)` for array of length n?
- Logarithmic $O(\log n)$: Repeatedly split search space in half
  - How many times k can we split before we reach singleton? $\left(\frac{1}{2}\right)^k n = 1$
  - $\left(\frac{1}{2}\right)^k = \frac{1}{n} \rightarrow 2^k = n \rightarrow k = \log n$ operations until we terminate!

Searching a sorted list

An ArrayList is implemented with an array:

1	17	23	24	32	76	79	87	95

But how do we sort?

- Complexity of `contains(Object o)` for array of length n?
- Logarithmic $O(\log n)$: Repeatedly split search space in half
  - How many times k can we split before we reach singleton? $\left(\frac{1}{2}\right)^k n = 1$
  - $\left(\frac{1}{2}\right)^k = \frac{1}{n} \to 2^k = n \to k = \log n$ operations until we terminate!

Bubble sort

32	87	17	79	95	76	24	1	23

Bubble sort

for $i = 0, \ldots, n-1$ **do**
    Find smallest element (index j) from indices $i, \ldots, n-1$, then swap (i, j)
end for

Bubble sort

32	87	17	79	95	76	24	1	23

Bubble sort

for $i = 0, \ldots, n-1$ **do**
  Find smallest element (index j) from indices $i, \ldots, n-1$, then swap (i, j)
end for

Iteration 1:

1	87	17	79	95	76	24	32	23

Bubble sort

32	87	17	79	95	76	24	1	23

Bubble sort

for $i = 0, \ldots, n-1$ **do**
    Find smallest element (index j) from indices $i, \ldots, n-1$, then swap (i, j)
end for

Iteration 1:	1	87	17	79	95	76	24	32	23
Iteration 2:	1	17	87	79	95	76	24	32	23

Bubble sort

32	87	17	79	95	76	24	1	23

Bubble sort

for $i = 0, \ldots, n-1$ **do**
  Find smallest element (index j) from indices $i, \ldots, n-1$, then swap (i, j)
end for

Iteration 1:	1	87	17	79	95	76	24	32	23
Iteration 2:	1	17	87	79	95	76	24	32	23
Iteration 3:	1	17	23	79	95	76	24	32	87

- Complexity?

Bubble sort

32	87	17	79	95	76	24	1	23

Bubble sort

for $i = 0, \ldots, n-1$ **do**
  Find smallest element (index j) from indices $i, \ldots, n-1$, then swap (i, j)
end for

Iteration 1:	1	87	17	79	95	76	24	32	23
Iteration 2:	1	17	87	79	95	76	24	32	23
Iteration 3:	1	17	23	79	95	76	24	32	87

- Complexity? $O(n^2)$ due to implicit double nested loop

We can do better: Start by merging sorted lists

List A:

1	17	23	43

List B:

2	19	23	31

Merged:

1	2	17	19	23	23	31	43

- Complexity?

We can do better: Start by merging sorted lists

List A:	1	17	23	43				
List B:	2	19	23	31				
Merged:	1	2	17	19	23	23	31	43

- Complexity? $O(n)$: just maintain indices of next element in each list

Merge sort

Subdivide until singletons, then repeatedly merge up to full array size n

43	1	17	23	19	31	23	2

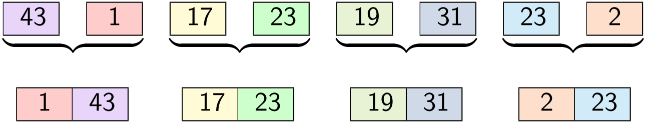
Merge sort

Subdivide until singletons, then repeatedly merge up to full array size n

$$\underbrace{\boxed{43}\quad\boxed{1}}\quad\underbrace{\boxed{17}\quad\boxed{23}}\quad\underbrace{\boxed{19}\quad\boxed{31}}\quad\underbrace{\boxed{23}\quad\boxed{2}}$$

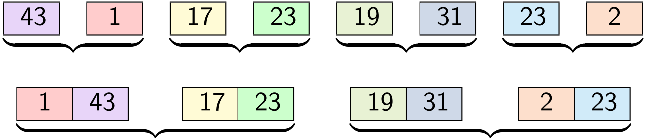
Merge sort

Subdivide until singletons, then repeatedly merge up to full array size n



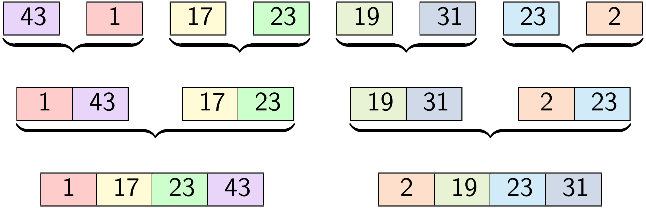
Merge sort

Subdivide until singletons, then repeatedly merge up to full array size n



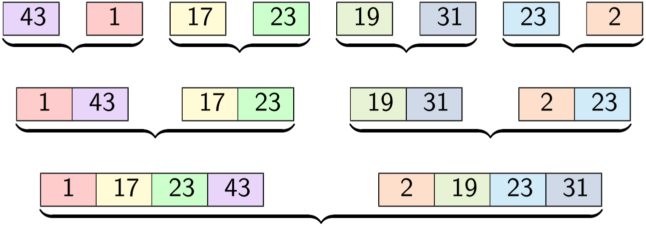
Merge sort

Subdivide until singletons, then repeatedly merge up to full array size n



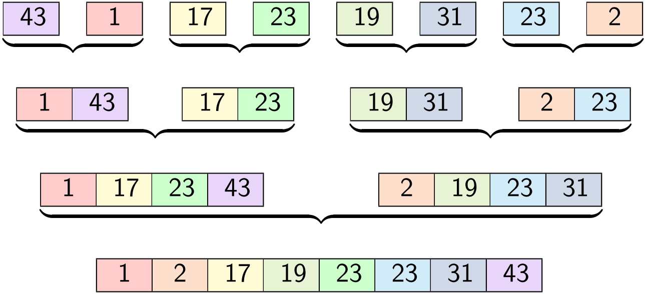
Merge sort

Subdivide until singletons, then repeatedly merge up to full array size n



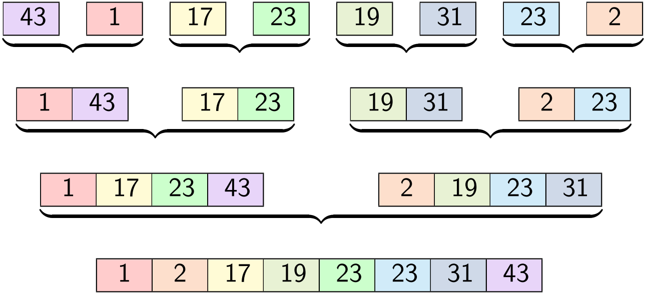
Merge sort

Subdivide until singletons, then repeatedly merge up to full array size n



Merge sort

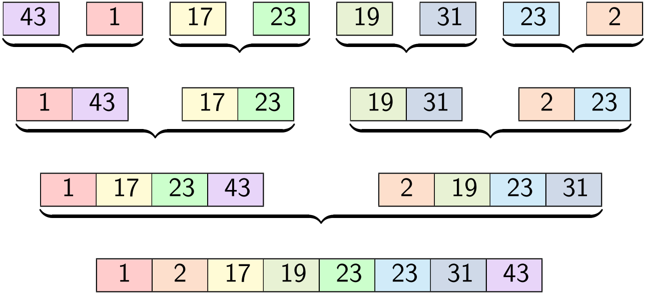
Subdivide until singletons, then repeatedly merge up to full array size n



- Complexity?

Merge sort

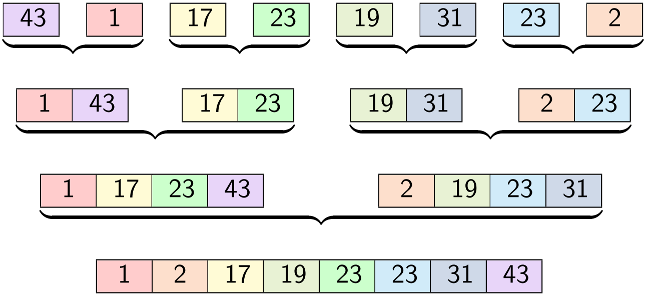
Subdivide until singletons, then repeatedly merge up to full array size n



- Complexity? $O(n \log n)$: n operations at each level, $\log n$ levels

Merge sort

Subdivide until singletons, then repeatedly merge up to full array size n



- Complexity? $O(n \log n)$: n operations at each level, $\log n$ levels
  - Best possible for sorting!

Hash sets and maps

HashSets and HashMaps

- **HashSet**: An unordered set of objects stored in way that allows for easy searching, insertion, and deletion.

- **HashMap**: Unordered collection of objects using `key-value` pairs; like an array, but with custom indices called map keys.

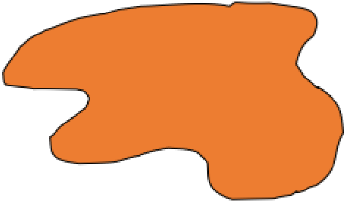
Hash tables for Sets and Maps

- Hash table: Like an array, but indices are a function of the original keys
- Order irrelevant, aim for constant-time ($O(1)$) `find`, `insert`, `delete`
  - “On average", under some reasonable assumptions

Hash tables for Sets and Maps

- Hash table: Like an array, but indices are a function of the original keys
- Order irrelevant, aim for constant-time ($O(1)$) `find`, `insert`, `delete`
  - “On average", under some reasonable assumptions

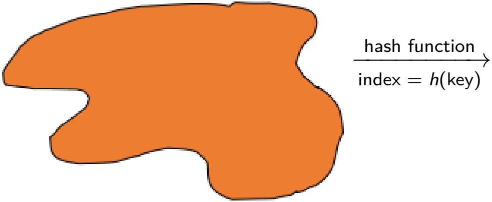
Key space (e.g., integers, strings):



Hash tables for Sets and Maps

- Hash table: Like an array, but indices are a function of the original keys
- Order irrelevant, aim for constant-time ($O(1)$) `find`, `insert`, `delete`
  - “On average", under some reasonable assumptions

Key space (e.g., integers, strings):

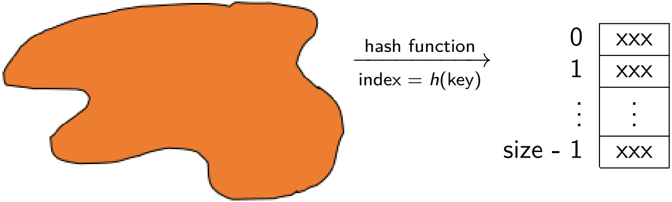


Hash tables for Sets and Maps

- Hash table: Like an array, but indices are a function of the original keys
- Order irrelevant, aim for constant-time ($O(1)$) `find`, `insert`, `delete`
  - “On average", under some reasonable assumptions

Key space (e.g., integers, strings):

Hash table



The ideal hash function

- Fast to compute
- Designed so that two keys rarely hash to the same index
  - Must happen if elements stored exceeds table size
  - Will handle collisions later

Example: Hashing strings

Key space $K = s_0 s_1 s_2 \dots s_{k-1}$, where s_i are chars: $s_i \in [0, 256]$

Which of these hash choices best avoids collisions?

1. $h(K) = \bmod\left(s_0, \text{TableSize}\right)$
2. $h(K) = \bmod\left(\sum_{i=0}^{k-1} s_i, \text{TableSize}\right)$
3. $h(K) = \bmod\left(\sum_{i=0}^{k-1} (s_i \times 37^i), \text{TableSize}\right)$

Collision resolution

- Collision: When two keys map to the same location in the hash table
- We try to avoid it, but cannot if number of keys exceeds table size
- So hash tables should support collision resolution
  - Ideas?

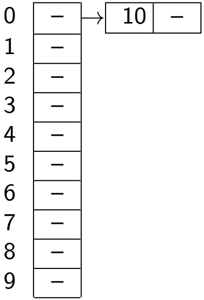
Separate chaining

- Chaining: All keys that map to the same table location are kept in a list (“chain" or “bucket")
- As easy as it sounds
- Example:
  insert 10, 22, 107, 12, 42
  with mod hashing
  and TableSize = 10

0	–
1	–
2	–
3	–
4	–
5	–
6	–
7	–
8	–
9	–

Separate chaining

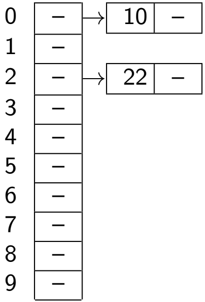
- Chaining: All keys that map to the same table location are kept in a list ("chain" or "bucket")
- As easy as it sounds
- Example:
  insert 10, 22, 107, 12, 42
  with mod hashing
  and TableSize = 10



$\mathrm{mod}(10, 10) = 0$

Separate chaining

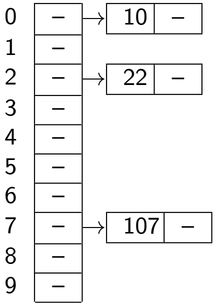
- Chaining: All keys that map to the same table location are kept in a list (“chain" or “bucket")
- As easy as it sounds
- Example:
  insert 10, 22, 107, 12, 42
  with mod hashing
  and TableSize $= 10$



$\text{mod}(22, 10) = 2$

Separate chaining

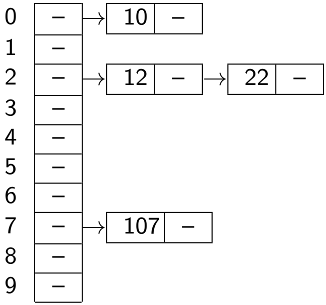
- Chaining: All keys that map to the same table location are kept in a list ("chain" or "bucket")
- As easy as it sounds
- Example: insert 10, 22, 107, 12, 42 with mod hashing and TableSize = 10



$\text{mod}(107, 10) = 7$

Separate chaining

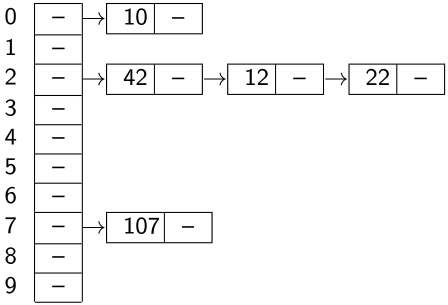
- Chaining: All keys that map to the same table location are kept in a list ("chain" or "bucket")
- As easy as it sounds
- Example:
  insert 10, 22, 107, 12, 42
  with mod hashing
  and TableSize $= 10$



$\text{mod}(12, 10) = 2$

Separate chaining

- Chaining: All keys that map to the same table location are kept in a list ("chain" or "bucket")
- As easy as it sounds
- Example:
  insert 10, 22, 107, 12, 42
  with mod hashing
  and TableSize = 10



$\mathrm{mod}(42, 10) = 2$

More rigorous chaining analysis

- The load factor (λ) of a hash table with N elements is

$$\lambda = \frac{N}{\text{TableSize}}$$

- Under chaining, the average number of elements per bucket is λ.
- So we like to keep $\lambda < 1$ for chaining: for good hash function $O(1)$ lookup!
- Java `HashSet`/`HashMap` will automatically grow the hash table when $\lambda > 0.75$.

Decision procedures and decidability

Not just how fast can it be computed, but can it be computed at all?

Can everything true be computed?

- **Decision procedure**: Code that terminates and returns true or false

Can everything true be computed?

- **Decision procedure**: Code that terminates and returns true or false
- **Decidable**: Always terminates
  - E.g., are there two positive primes that sum to N?

Can everything true be computed?

- **Decision procedure**: Code that terminates and returns true or false
- **Decidable**: Always terminates
  - E.g., are there two positive primes that sum to N?
- **Semidecidable**: Only guaranteed to terminate on true
  - E.g., are there two positive primes whose difference is N?

Can everything true be computed?

- **Decision procedure**: Code that terminates and returns true or false
- **Decidable**: Always terminates
  - E.g., are there two positive primes that sum to N?
- **Semidecidable**: Only guaranteed to terminate on true
  - E.g., are there two positive primes whose difference is N?
- **Undecidable**: May not terminate
  - E.g., is there a real number x that makes $f(x) = 1.0$?

Can everything true be computed?

- **Decision procedure**: Code that terminates and returns true or false
- **Decidable**: Always terminates
  - E.g., are there two positive primes that sum to N?
- **Semidecidable**: Only guaranteed to terminate on true
  - E.g., are there two positive primes whose difference is N?
- **Undecidable**: May not terminate
  - E.g., is there a real number x that makes $f(x) = 1.0$?

Corollary: There are true/false decisions that we cannot compute and there are true statements for which a proof can never be found!

Data structures (just heaps)

modified slides from Kevin Wayne
based on material from “Introduction to Algorithms" by Cormen, Leiserson, Rivest, and Stein
https://mitpress.mit.edu/books/introduction-algorithms-third-edition

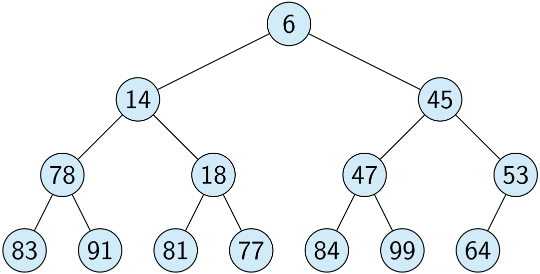
Smart choice of data structures = fast algorithms

- We have already seen HashSets and HashMaps for storing unordered data

- But what if we need to store ordered data?
  - Heaps: tree-based data structures, many flavors

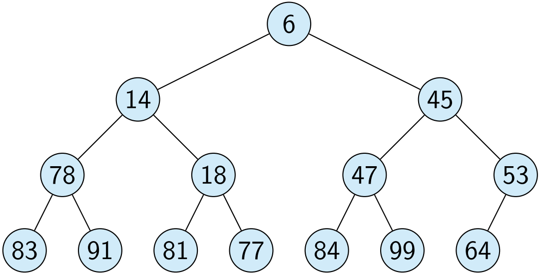
Binary heap: Definition

- An almost complete binary tree
  - Filled on all levels, except last, where filled from left to right
- Example: Min-heap ordered
  - Every child greater than (or equal to) parent



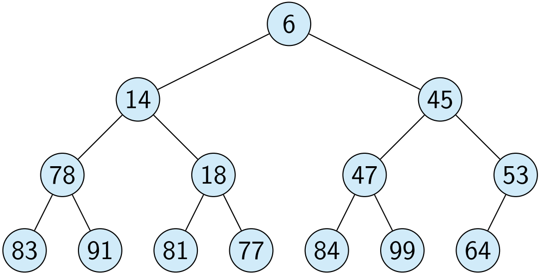
Binary heap: Definition

- An almost complete binary tree
  - Filled on all levels, except last, where filled from left to right
- Example: Min-heap ordered
  - Every child greater than (or equal to) parent
  - What would be different for a max-heap?



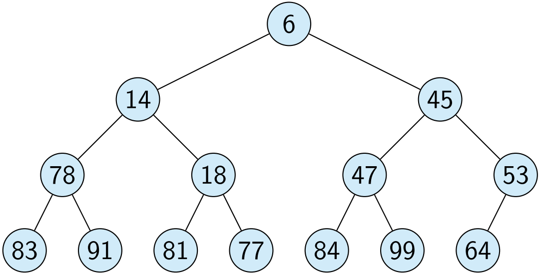
Binary heap: Properties

- Min element is the root
- Heap with N elements has height $= \lfloor \log_2 N \rfloor$



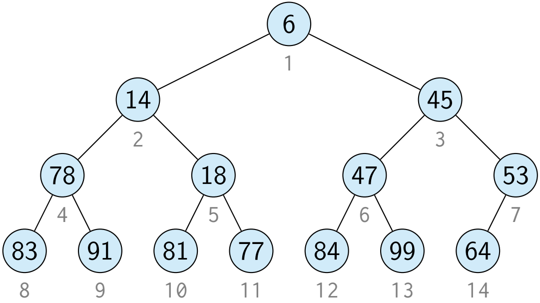
Binary heap: Properties

- Min element is the root
- Heap with N elements has height $= \lfloor \log_2 N \rfloor$
  - $N = 14$
  - height $= \lfloor \log_2(14) \rfloor = 3$ (starts at 0)



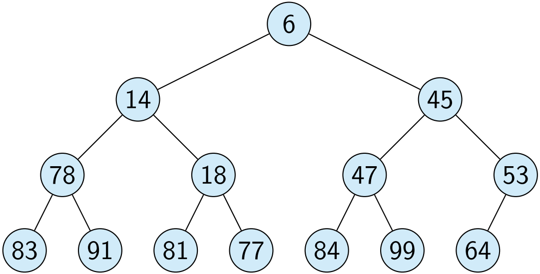
Binary heap: Array implementation

- Use as an array, no need for explicit parent or child pointers.
- For element `i` in the array:
  - `Parent(i)` $= \lfloor i/2 \rfloor$
  - `Left(i)` $= 2i$
  - `Right(i)` $= 2i + 1$



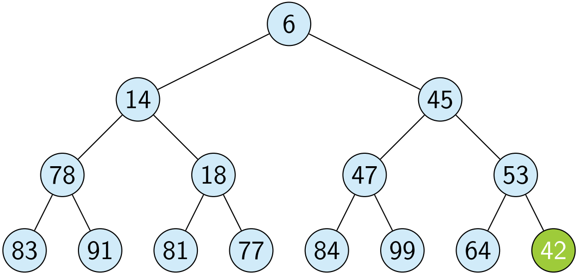
Binary heap: Insertion

- Insert into next available slot.
- Bubble up until heap is ordered.
  - Nodes rise as long as parent is larger.
- Complexity?



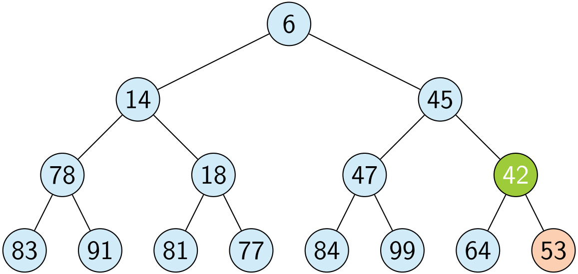
Binary heap: Insertion

- Insert into next available slot.
- Bubble up until heap is ordered.
  - Nodes rise as long as parent is larger.
- Complexity?



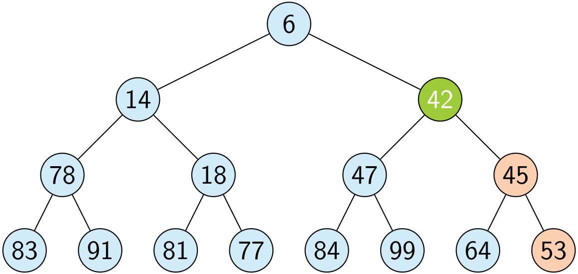
Binary heap: Insertion

- Insert into next available slot.
- Bubble up until heap is ordered.
  - Nodes rise as long as parent is larger.
- Complexity?



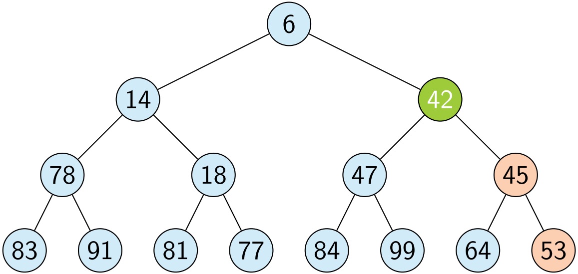
Binary heap: Insertion

- Insert into next available slot.
- Bubble up until heap is ordered.
  - Nodes rise as long as parent is larger.
- Complexity?



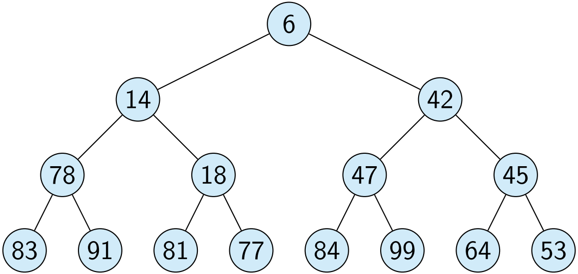
Binary heap: Insertion

- Insert into next available slot.
- Bubble up until heap is ordered.
  - Nodes rise as long as parent is larger.
- Complexity? $O(\log n)$ operations



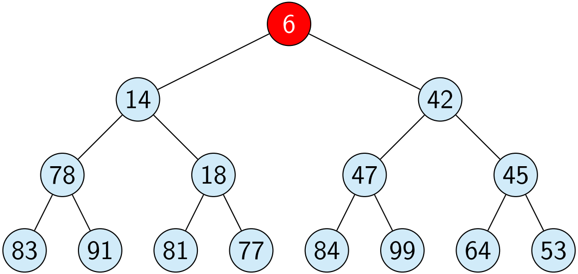
Binary heap: Delete minimum element

- Remove root, replace with rightmost leaf.
- Bubble down until heap is ordered.
  - Smaller child is promoted.
- Complexity?



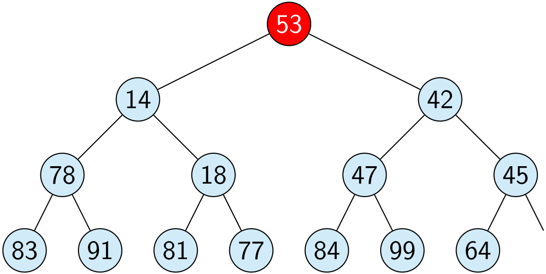
Binary heap: Delete minimum element

- Remove root, replace with rightmost leaf.
- Bubble down until heap is ordered.
  - Smaller child is promoted.
- Complexity?



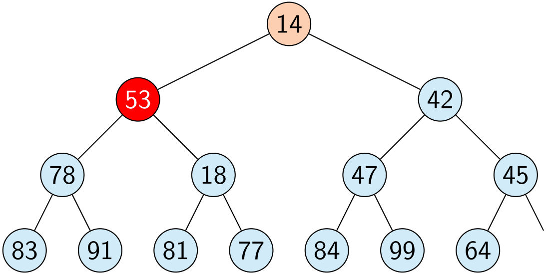
Binary heap: Delete minimum element

- Remove root, replace with rightmost leaf.
- Bubble down until heap is ordered.
  - Smaller child is promoted.
- Complexity?



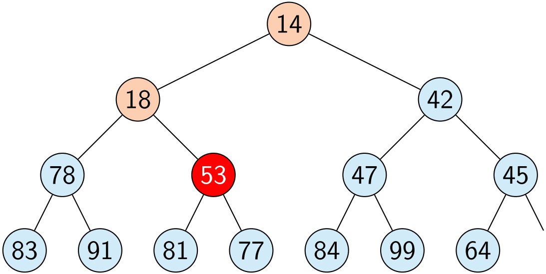
Binary heap: Delete minimum element

- Remove root, replace with rightmost leaf.
- Bubble down until heap is ordered.
  - Smaller child is promoted.
- Complexity?



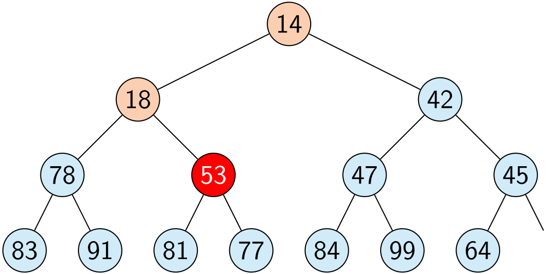
Binary heap: Delete minimum element

- Remove root, replace with rightmost leaf.
- Bubble down until heap is ordered.
  - Smaller child is promoted.
- Complexity?



Binary heap: Delete minimum element

- Remove root, replace with rightmost leaf.
- Bubble down until heap is ordered.
  - Smaller child is promoted.
- Complexity? $O(\log n)$ operations



Binary heap: Heapsort

- Insert N items into a binary heap.
- Perform N delete-min operations.
- Complexity?

Binary heap: Heapsort

- Insert N items into a binary heap.
- Perform N delete-min operations.
- Complexity? $O(N \log N)$

Binary heap: Heapsort

- Insert N items into a binary heap.
- Perform N delete-min operations.
- Complexity? $O(N \log N)$
- No extra storage required to store sorted result.
  - Why?

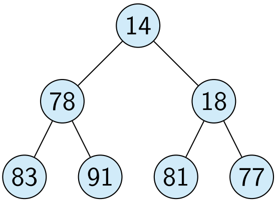
Binary heap: Heapsort

- Insert N items into a binary heap.
- Perform N delete-min operations.
- Complexity? $O(N \log N)$
- No extra storage required to store sorted result.
  - Why?
  - How does this approach compare with merge sort?

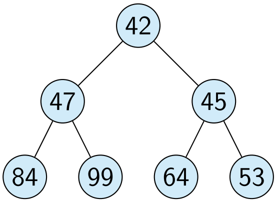
Binary heap: Union

- Combine two binary heaps, H_1 and H_2, into a single heap?
- No easy solution.

H_1

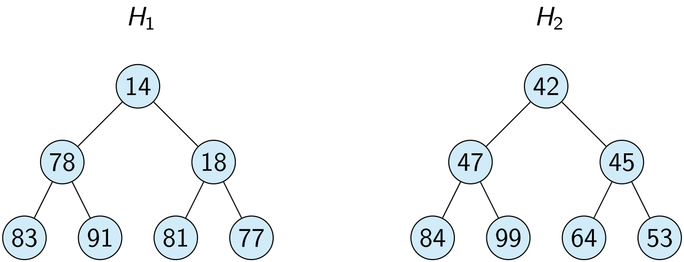


H_2



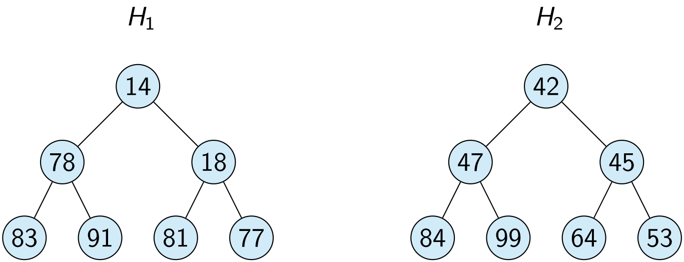
Binary heap: Union

- Combine two binary heaps, H_1 and H_2, into a single heap?
- No easy solution.
  - Why?



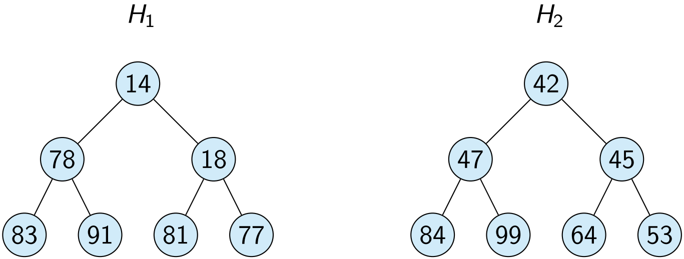
Binary heap: Union

- Combine two binary heaps, H_1 and H_2, into a single heap?
- No easy solution.
  - Why?
  - $\Omega(N)$ (best case) operations apparently required



Binary heap: Union

- Combine two binary heaps, H_1 and H_2, into a single heap?
- No easy solution.
  - Why?
  - $\Omega(N)$ (best case) operations apparently required
- Fast union supported with fancier heaps, e.g., binomial, Fibonacci



Heaps: Summary

- Heap data structure
  - Min-heap or max-heap (what properties?)
  - Store heap of N items compactly in an array
  - Note: repeats allowed
- Operations and time complexity
  - Perform $O(\log N)$ time insertion
  - Perform $O(1)$ lookup of min item for Min-heap
  - Perform $O(\log N)$ removal of min item for Min-heap
  - How to use above operations to make $O(N \log N)$ sorting algorithm?
- Questions
  - What is the $O(\cdot)$ time complexity of `contains`?
  - How to track top-k scoring N webpages in a Google search in $O(N \log k)$?