SFU CMPT-307 2008-2 Lecture: Week 2

Ján Maňuch

E-mail: jmanuch@sfu.ca

Lecture on May 13, 2008, 5.30pm-8.20pm

Divide and Conquer — Merge-Sort

- *divide* the problem into a number of subproblems
- *conquer* the subproblems by solving them recursively or if they are small, there must be an easy solution.
- *combine* the solutions to the subproblems to the solution of the problem

Example:

MERGE-SORT(A, p, r)

if $p < r$ **then**
    $q \leftarrow \lfloor (p + r)/2 \rfloor$
    MERGE-SORT(A, p, q)
    MERGE-SORT$(A, q + 1, r)$
    MERGE(A, p, q, r)
end if

Initial call for input array $A = A[1] \dots A[n]$ is MERGE-SORT$(A, 1, n)$.

Example:

sorted sequence

$\boxed{1\ 2\ 2\ 3\ 4\ 5\ 6\ 7}$

$\boxed{2\ 4\ 5\ 7}\ \boxed{1\ 2\ 3\ 6}$

$\boxed{2\ 5}\ \boxed{4\ 7}\ \boxed{1\ 3}\ \boxed{2\ 6}$

$\boxed{5}\ \boxed{2}\ \boxed{4}\ \boxed{7}\ \boxed{1}\ \boxed{3}\ \boxed{2}\ \boxed{6}$

initial sequence

MERGE-SORT

- splits length-ℓ sequence into two length-$\ell/2$ sequences
- sorts them recursively
- merges the two sorted subsequences

Merging

$\textsc{Merge}(A, p, q, r)$
Take the smallest of the two frontmost elements of sequences $A[p..q]$ and $A[q+1..r]$ and put it into a temporary array. Repeat this, until both sequences are empty. Copy the resulting sequence from temporary array into $A[p..r]$.

Assignment Problem 2.1. (deadline: May 20, 5:30pm)
Write a pseudo code for the procedure $\textsc{Merge}(A, p, q, r)$ used in Merge-sort algorithm for merging two sorted arrays $A[p \ldots q]$ and $A[q+1 \ldots r]$ into one *without using sentinels* (see Section 2.3.1 of the textbook for details).

Analyzing an D&Q algorithms

Let $T(n)$ be running time on problem of size n

- If n is small enough (say, $n \leq c$ for constant c), then straightforward solution takes $\Theta(1)$
- If division of problem yields a subproblems, each of which $1/b$ of original (Merge-Sort: $a = b = 2$)
- Division into subproblems takes $D(n)$
- Combination of solutions to subproblems takes $C(n)$

Then,

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

For Merge-Sort:

- $a = b = 2$
- $D(n) = \Theta(1)$ (just compute "middle" of array)
- $C(n) = \Theta(n)$ (merging has running time linear in length of resulting sequence; take my word for it)

Thus

$$T_{\mathrm{MS}}(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ T_{\mathrm{MS}}(\lfloor n/2 \rfloor) + T_{\mathrm{MS}}(\lceil n/2 \rceil) + \Theta(n) & \text{otherwise.} \end{cases}$$

That's what's called a **recurrence**

But: we want **closed form**, i.e., we want to *solve the recurrence*.

Solving recurrences

There are a few methods for solving recurrences, some easy and not powerful, some complicated and powerful.

Methods:

1. guess & verify (also called "*substitution method*")
2. *recursion-tree* method
3. *master* method
4. generating functions

and some others.

We're going to see 1., 2. and 3.

Substitution method

Basic idea:

1. "*guess*" the form of the solution
2. Use mathematical induction to find constants and *show that solution works*.

Usually more difficult part is the part 1.

Back to example: we had

$$T_{\mathrm{MS}}(n) \leq 2T_{\mathrm{MS}}(\lceil n/2 \rceil) + \Theta(n)$$

for $n \geq 2$.

If you hadn't seen something like this before, **how would you guess?**

There is no general way to guess the correct solutions. It takes experience.

Heuristics that can help to find a good guess.

- One way would be to have a **look at first few terms**. Say if we had $T(n) = 2T(n/2) + 3n$, then

$$\begin{aligned}
T(n) &= 2T(n/2) + 3n \\
&= 2(2T(n/4) + 3(n/2)) + 3n \\
&= 2(2(2T(n/8) + 3(n/4)) + 3(n/2)) + 3n \\
&= 2^3 T(n/2^3) + 2^2 3(n/2^2) + 2^1 3(n/2^1) + 2^0 3(n/2^0)
\end{aligned}$$

We can do this $\log n$ times

$$\begin{aligned}
& 2^{\log n} \cdot T(n/2^{\log n}) + \sum_{i=0}^{\log(n)-1} 2^i 3(n/2^i) \\
= \quad & n \cdot T(1) + 3n \cdot \sum_{i=0}^{\log(n)-1} 1 \\
= \quad & n \cdot T(1) + 3n \log n = \Theta(n \log n)
\end{aligned}$$

A *guess*! After guessing a solution you'll have to **prove the correctness**.

- **recursion-tree** method (in a moment)
- **similar recurrences** might have similar solutions

  Consider

$$T(n) = 2T(\lfloor n/2 \rfloor + 25) + n$$

Looks similar to last example, but is the additional 25 in the **argument** going to change the solution?

Not really, because for large n, difference between

$$T(\lfloor n/2 \rfloor) \quad \text{and} \quad T(\lfloor n/2 \rfloor + 25)$$

is not large: both cut n nearly in half: for $n = 2,000$ we have

$$T(1,000) \quad \text{and} \quad T(1,025),$$

for $n = 1,000,000$ we have

$$T(500,000) \quad \text{and} \quad T(500,025).$$

Thus, reasonable assumption is that now $T(n) = O(n \log n)$ as well.

- **stepwise refinement** – guessing loose lower and upper bounds, and gradually taking them closer to each other

  For $T(n) = 2T(\lfloor n/2 \rfloor) + n$ we see

  – $T(n) = \Omega(n)$ (because of the n term)

  – $T(n) = O(n^2)$ (easily proven)

  From there, we can perhaps "converge" on asymptotically tight bound $\Theta(n \log n)$.

Proving correctness of a guess

For Merge-Sort we have $T(n) \leq 2T(\lceil n/2 \rceil) + \Theta(n)$, which means, there is a constant $d > 0$ such that

$$T(n) \leq 2T(\lceil n/2 \rceil) + dn \qquad \text{for } n \geq 2.$$

We guessed that $T(n) = O(n \log n)$.

We prove $T(n) \leq cn \log n$ for appropriate choice of constant $c > 0$.

Induction hypothesis: assume that bound holds for any $n < m$, hence also for $n = \lceil m/2 \rceil$, i.e.,

$$T(\lceil m/2 \rceil) \leq c\lceil m/2 \rceil \log(\lceil m/2 \rceil)$$

and prove it for $m \geq 4$ (**induction step**):

$$\begin{aligned}
T(m) &\leq 2T(\lceil m/2 \rceil) + dm \\
&\leq 2(c\lceil m/2 \rceil \log(\lceil m/2 \rceil)) + dm \\
&\leq 2(c(m+1)/2 \cdot \log((m+1)/2)) + dm \\
&\leq c(m+1)\log((m+1)/2) + dm \\
&= c(m+1)\log(m+1) - c(m+1)\log 2 + dm \\
&\leq c(m+1)(\log m + 1/3) - c(m+1) + dm \\
&= cm\log m + c\log m + c/3(m+1) - c(m+1) + dm \\
&\leq cm\log m + cm/2 - 2/3c(m+1) + dm \\
&\leq cm\log m + (c/2 - 2/3c + d)m = cm\log m + (d - c/6)m \\
&\leq cm\log m \quad \text{for } c \geq 6d.
\end{aligned}$$

Base step

It remains to show that boundary conditions of recurrence ($m < 4$) are **suitable as base cases for the inductive proof**.

We have got to show that we can choose c large enough s.t.bound

$$T(m) \leq cm \log m$$

works for boundary conditions as well (when $m < 4$).

Assume that $T(1) = b > 0$.

For $m = 1$,

$$T(m) \leq cm \log m = c \cdot 1 \cdot 0 = 0$$

is a bit of a problem, because $T(1)$ is a constant greater than 0.

How to resolve this problem?

Recall that we wanted to prove that $T(n) = O(n \log n)$.

Also, recall that by def of $O()$, we are free to disregard a constant number of small values of n: $f(n) = O(g(n)) \;\Leftrightarrow\;$ there exist constants c, n_0 : $f(n) \leq c \cdot g(n)$ for $n \geq n_0$

A way out of our problem is to **remove** difficult boundary condition $T(1) = b > 0$ from consideration in inductive proof.

Note: for $m \geq 4$, $T(m)$ does **not** depend directly on $T(1)$
($T(2) \leq 2T(1) + 2d = 2b + 2d$,
$T(3) \leq 2T(2) + 3d = 2(2b + 2d) + 3d = 4b + 7d$, $T(4) \leq 2T(2) + 4d$)

This means:
We replace $T(1)$ by $T(2)$ and $T(3)$ as the base case in the inductive proof, letting $n_0 = 2$.

In other words, we are showing that the bound

$$T(n) \leq cn \log n$$

holds for any $n \geq 2$.

- For $m = 2$: $T(2) = 2b + 2d? \leq ?c.2 \log 2 = 2c$
- For $m = 3$: $T(3) = 4b + 7d? \leq ?c.3 \log 3$

Hence, set c to $\max(6d, b + d, \frac{4b+7d}{3 \log 3})$ and the above boundary conditions as well as the induction step will work.

Important:

General technique! Can be used very often!

Exercise: Show that the solution of the recurrence

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n),$$

is also in $\Omega(n \log n)$.

Assignment Problem 2.2. (deadline: May 20, 5:30pm)
Show that the solution of

$$T(n) = 2T(\lfloor n/2 \rfloor + 25) + n$$

is in $O(n \log n)$ and in $\Omega(n \log n)$.

Common pitfall

Might be tempted to "falsely" prove that for $T(n) = 2T(n/2) + n$ we have $T(n) = O(n)$.

Guess $T(n) \leq c \cdot n$.

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &\leq 2(c.n/2) + n \\ &\leq cn + n \\ &= (c+1) \cdot n \\ &= O(n) \end{aligned}$$

Although $(c+1) \cdot n$ certainly **is** in $O(n)$, we have **not** proved that $T(n) \leq c \cdot n$.

We **must** prove **exact** form of inductive hypothesis!

A neat trick called "changing variables"

Suppose we have

$$T(n) = 2T(\sqrt{n}) + \log n$$

Now rename $m = \log n \Leftrightarrow 2^m = n$. We know $\sqrt{n} = n^{1/2} = (2^m)^{1/2} = 2^{m/2}$ and thus obtain

$$T(2^m) = 2T(2^{m/2}) + m$$

Now rename $S(m) = T(2^m)$ and get

$$S(m) = 2S(m/2) + m$$

Looks familiar. We know the solution $S(m) = \Theta(m \log m)$.

Going back from $S(m)$ to $T(n)$ we obtain

$$T(n) = T(2^m) = S(m) = \Theta(m \log m) = \Theta(\log n \log \log n)$$

The recursion-tree method

- helps to come up with a good guess for the substitution method
- useful for recurrences describing the running time of a divide-and-conquer algorithm,
- build a *recursion tree* where each node represents the cost of single subproblem called somewhere during the computation of the main problem
- we can forget about the details: floors and ceilings

Some basic algebra

Sums

$$a + (a+1) + (a+2) + \cdots + (b-1) + b$$
$$= \sum_{i=a}^{b} i = \frac{(a+b)(b-a+1)}{2}$$

$$a + a \cdot c + a \cdot c^2 + \cdots + a \cdot c^{n-1} + a \cdot c^n$$
$$= \sum_{i=0}^{n} a \cdot c^i = a \cdot \frac{1-c^{n+1}}{1-c}$$

if $0 < c < 1$ then we can estimate the sum as follows

$$\sum_{i=0}^{n} a \cdot c^i = a \cdot \frac{1-c^{n+1}}{1-c} < a \cdot \frac{1}{1-c}$$

logarithms and powers

$\log_a n$ and a^n are inverse functions to each other:

$\boxed{\log_a(a^n) = n}$ for all n

$\boxed{a^{\log_a n} = n}$ for all $n > 0$

properties:

$\log_a(b \cdot c) = \log_a b + \log_a c$
$\log_a b = \frac{\log_c b}{\log_c a}$
$\log_a b = \frac{1}{\log_b a}$
$a^{\log_c b} = b^{\log_c a}$

Example. $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$

build a recursion tree for $T(n) = 3T(n/4) + cn^2$

see Figure 4.1 in textbook

summing the rows of the tree we get

$$\begin{aligned} T(n) =& cn^2 + \tfrac{3}{16}cn^2 + (\tfrac{3}{16})^2 cn^2 + \cdots + (\tfrac{3}{16})^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3}) \\ <& \frac{1}{1 - 3/16} cn^2 + \Theta(n^{\log_4 3}) \\ =& \tfrac{16}{13} cn^2 + \Theta(n^{\log_4 3}) \in \mathcal{O}(n^2) \end{aligned}$$

$\log_4 3 \doteq 1.262$, hence

$n^{\log_4 3} \in o(n^2)$ and so

$T(n) \in \mathcal{O}(n^2)$

then prove by the substitution method (MI) that the guess $\mathcal{O}(n^2)$ is correct

Assignment Problem 2.3. (deadline: May 20, 5:30pm)
Use a recursion tree to determine a good asymptotic upper bound on the recurrence

$$T(n) = 3T(\lfloor n/2 \rfloor) + n\,.$$

The "Master Method"

Recipe for recurrences of the form

$$T(n) = aT(n/b) + f(n)$$

with $a \geq 1$ and $b > 1$ constant, and $f(n)$ an asymptotically positive function ($f(n) = 5$, $f(n) = c \log n$, $f(n) = n$, $f(n) = n^{12}$ are just fine).

Split problem into a subproblems each of size n/b.

Subproblems are solved recursively, each in time $T(n/b)$.

Dividing problem and combining solutions of subproblems is captured by $f(n)$.

Deals with many frequently seen recurrences (in particular, our Merge-Sort example with $a = b = 2$ and $f(n) = \Theta(n)$).

Theorem. Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n),$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ can be bounded asymptotically as follows.

1. If $f(n) = \mathcal{O}(n^{(\log_b a)-\epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.

2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \cdot \log n)$.

3. If $f(n) = \Omega(n^{(\log_b a)+\epsilon})$ for some constant $\epsilon > 0$, and if $a \cdot f(n/b) \leq c \cdot f(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

Notes on Master's Theorem

2. If $f(n) = \Theta(n^{\log_b a})$, then
   $T(n) = \Theta(n^{\log_b a} \cdot \log n)$.

Note 1: Although it's looking rather scary, it really isn't. For instance, with Merge-Sort's recurrence $T(n) = 2T(n/2) + \Theta(n)$ we have $n^{\log_b a} = n^{\log_2 2} = n^1 = n$, and we can apply case 2.
The result is therefore $\Theta(n^{\log_b a} \cdot \log n) = \Theta(n \log n)$.

1. If $f(n) = \mathcal{O}(n^{(\log_b a)-\epsilon})$ for some constant $\epsilon > 0$, then
   $T(n) = \Theta(n^{\log_b a})$.

Note 2: In case 1,

$$f(n) = n^{(\log_b a)-\epsilon} = n^{\log_b a}/n^{\epsilon} = o(n^{\log_b a}) \, ,$$

so the ϵ **does** matter. This case is basically about "small" functions f. But it's not enough if $f(n)$ is just asymptotically smaller than $n^{\log_b a}$ (that is $f(n) \in o(n^{\log_b a})$), it must *polynomially smaller*!

3. If $f(n) = \Omega(n^{(\log_b a)+\epsilon})$ for some constant $\epsilon > 0$, and if $a \cdot f(n/b) \leq c \cdot f(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

Note 3: Similarly, in case 3,

$$f(n) = n^{(\log_b a)+\epsilon} = n^{\log_b a} \cdot n^{\epsilon} = \omega(n^{\log_b a}) \, ,$$

so the ϵ **does** matter again. This case is basically about "large" functions n. But again, $f(n) \in \omega(n^{\log_b a})$ is not enough, it must be *polynomially larger*. And in addition $f(n)$ has to satisfy the "**regularity condition**":

$$af(n/b) \leq cf(n)$$

for for some constant $c < 1$ and $n \geq n_0$ for some n_0.

The idea is that we *compare* $n^{\log_b a}$ to $f(n)$.

Result is (intuitively) determined by larger of two.

In case 1, $n^{\log_b a}$ is larger, so result is $\Theta(n^{\log_b a})$.

In case 3, $f(n)$ is larger, so result is $\Theta(f(n))$.

In case 2, both have same order, we multiply it by a logarithmic factor, and result is $\Theta(n^{\log_b a} \cdot \log n) = \Theta(f(n) \cdot \log n)$.

Important: Does not cover **all** possible cases.

For instance, there is a gap between cases 1 and 2 whenever $f(n)$ is smaller than $n^{\log_b a}$ but not **polynomially** smaller.

Using the master theorem

Simple enough. Some examples:

$T(n) = 9T(n/3) + n$

We have $a = 9$, $b = 3$, $f(n) = n$. Thus, $n^{\log_b a} = n^{\log_3 9} = n^2$. Clearly, $f(n) = \mathcal{O}(n^{\log_3(9)-\epsilon})$ for $\epsilon = 1$, so case 1 gives $T(n) = \Theta(n^2)$.

$T(n) = T(2n/3) + 1$

We have $a = 1$, $b = 3/2$, and $f(n) = 1$, so $n^{\log_b a} = n^{\log_{2/3} 1} = n^0 = 1$. Apply case 2 ($f(n) = \Theta(n^{\log_b a}) = \Theta(1)$, result is $T(n) = \Theta(\log n)$.

$T(n) = 3T(n/4) + n \log n$

We have $a = 3$, $b = 4$, and $f(n) = n \log n$, so $n^{\log_b a} = n^{\log_4 3} = \mathcal{O}(n^{0.793})$. Clearly, $f(n) = n \log n = \Omega(n)$ and thus also $f(n) = \Omega(n^{\log_b(a)+\epsilon})$ for $\epsilon \approx 0.2$. Also, $a \cdot f(n/b) = 3(n/4)\log(n/4) \leq (3/4)n \log n = c \cdot f(n)$ for any $c = 3/4 < 1$. Thus we can apply case 3 with result $T(n) = \Theta(n \log n)$.

Assignment Problem 2.4. (deadline: May 20, 5:30pm)
Use the master method to give tight asymptotic bounds for the following recurrences:

(a) $T(n) = 4T(n/2) + n$,

(b) $T(n) = 4T(n/2) + n^2$,

(c) $T(n) = 4T(n/2) + n^3$.

Examples when we cannot apply the master theorem

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

we have $a = 2$, $b = 2$, $f(n) = n \log n$ and $n^{\log_b a} = n$. Although, $f(n) = n \log n$ is asymptotically larger than n, it's not polynomially larger. The ratio

$$f(n)/n^{\log_b n} = (n \log n)/n = \log n$$

is asymptotically smaller than n^{ϵ} for any positive constant ϵ. The recurrence is in the “gap” between cases 2 and 3.

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

similarly, the recurrence is in the gap between cases 1 and 2

---

but we can get at least asymptotic lower and upper bounds!

Scheme how to use the master theorem

$$T(n) = aT(n/b) + f(n)$$

1. identify a, b and $f(n)$
2. compute the special function $s(n) = n^{\log_b a}$
3. compare the special function $s(n)$ to $f(n)$
   (a) $f(n)$ asymptotically smaller than $s(n)$
   - check if polynomially smaller:
     compute $\boxed{\dfrac{s(n)}{f(n)}}$
   - if in $\Omega(n^\epsilon)$ for some $\epsilon > 0$, then (case 1) answer is $\Theta(s(n))$
   - if in $o(n^\epsilon)$ for all $\epsilon > 0$, then we have a lower bound $\Omega(s(n))$ and an upper bound $\mathcal{O}(s(n) \log n)$

(b) $f(n)$ asymptotically equal to $s(n)$

- case 2, answer is $\Theta(f(n)\log n)$

(c) $f(n)$ asymptotically larger than $s(n)$

- check if polynomially larger:

  compute $\boxed{\dfrac{f(n)}{s(n)}}$
- if in $\Omega(n^\epsilon)$ for some $\epsilon > 0$, then check regularity condition; if ok, then (case 3) answer is $\Theta(f(n))$
- if in $o(n^\epsilon)$ for all $\epsilon > 0$, then we have a lower bound $\Omega(s(n)\log n)$ and an upper bound $\mathcal{O}(s(n)n^\epsilon)$ for all $\epsilon > 0$

Exercise. Problem 4–1 (page 85), Problem 4–4 (page 86)

Sorting

Input: A sequence of n numbers $A[1], \ldots, A[n]$;
Output: A reordering $B[1], \ldots, B[n]$ of the input sequence such that $B[1] \leq B[2] \leq \cdots \leq B[n]$.

Sorting in general is extremely important, as well in theory as in practice.

Sorting problems pop up all over the place, thus it's **extremely important** that sorting can be done **fast** (think of large databases).

We've already seen 2 sorting algorithms, *Selection-Sort* and *Merge-Sort*.

Selection-Sort has running time $O(n^2)$, but: it's **in-place**, and the constants are small. It's good for small inputs (like being recursion end for other algorithms).

in-place only a constant number of elements of the input array are ever stored outside the array

Merge-Sort is asymptotically faster, $O(n \log n)$, but: it's not in-place (Merge operation).

We're going to see a few others:

Heap-Sort: $O(n \log n)$, in-place

Quick-Sort: $O(n^2)$ worst-case, but $O(n \log n)$ on average, in-place, in practice generally faster than Heap-Sort. Small constants. Good for large inputs.

So far, all **comparison-based** algorithms.

We can prove the **lower bound** of $\Omega(n \log n)$ for such algorithms (later in the course).

There are other approaches, when we collect information about the input using different methods than comparing elements.

Using those approaches we can achieve a linear time (examples: **Radix-Sort** and **Bucket-Sort**), but we have to put additional conditions on the input (for example that elements are integers from the set $\{1, 2, \ldots, cn\}$ for some constant c).

We will see later...

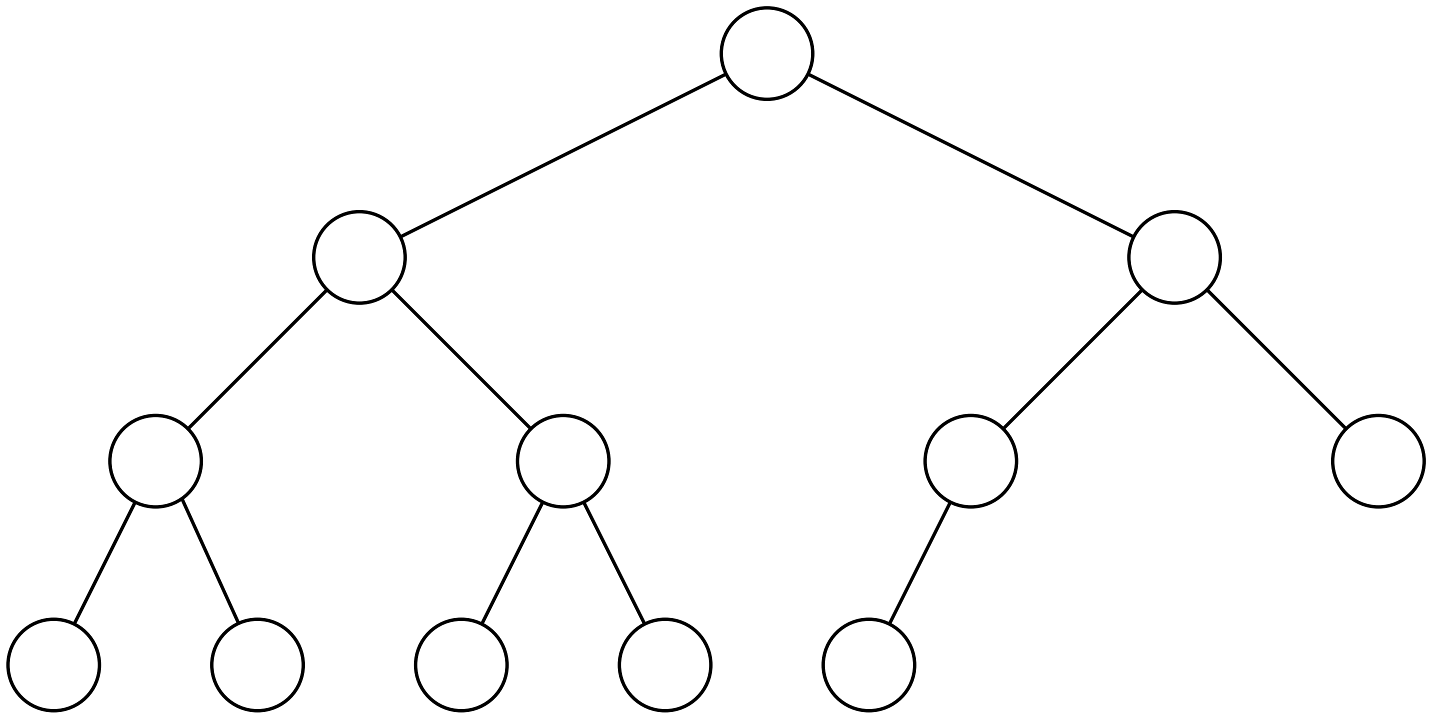
Heap

A **heap** (data structure) is a **linear array** that stores a nearly complete **tree**.

Only talking about **binary heaps** that store **binary trees**.

nearly complete trees:

- all levels except possibly the lowest one are filled
- the bottom level is filled from left to right up to some point



Want to store trees like that one such that certain properties are maintained.

Suppose that array A stores (or represents) a binary heap

Two major attributes:

- length(A) is **number of elements** in array A
- heap-size(A) is **number of elements in heap** stored within array A

Although $A[1], \ldots, A[\text{length}(A)]$ can contain **valid numbers**, only elements $A[1], \ldots, A[\text{heap-size}(A)]$ actually store **elements of the heap**, heap-size$(A) \leq$ length(A).

Will see what this is good for.

Assignment of tree vertices to array elements:

Very easy:

- root is $A[1]$
- given index i of some node, we have
  - $\textsc{Parent}(i) = \lfloor i/2 \rfloor$
  - $\textsc{Left}(i) = 2i$
  - $\textsc{Right}(i) = 2i + 1$

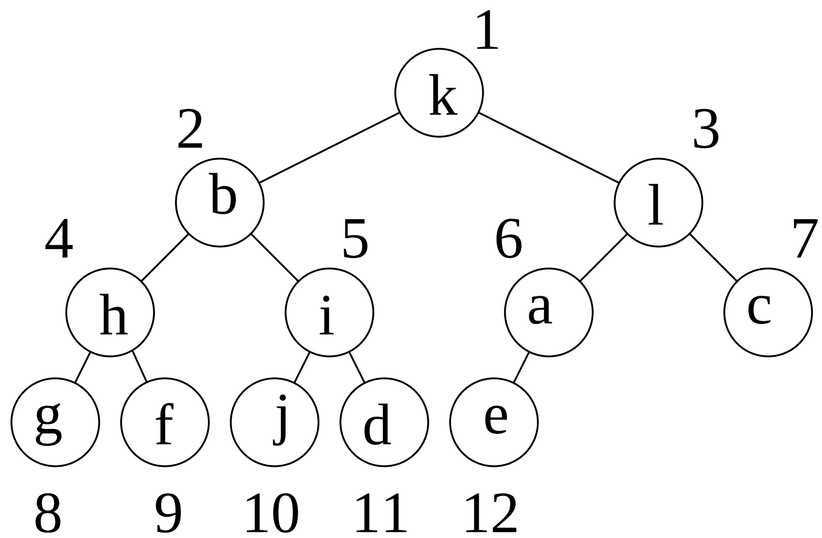
Implementation straightforward:

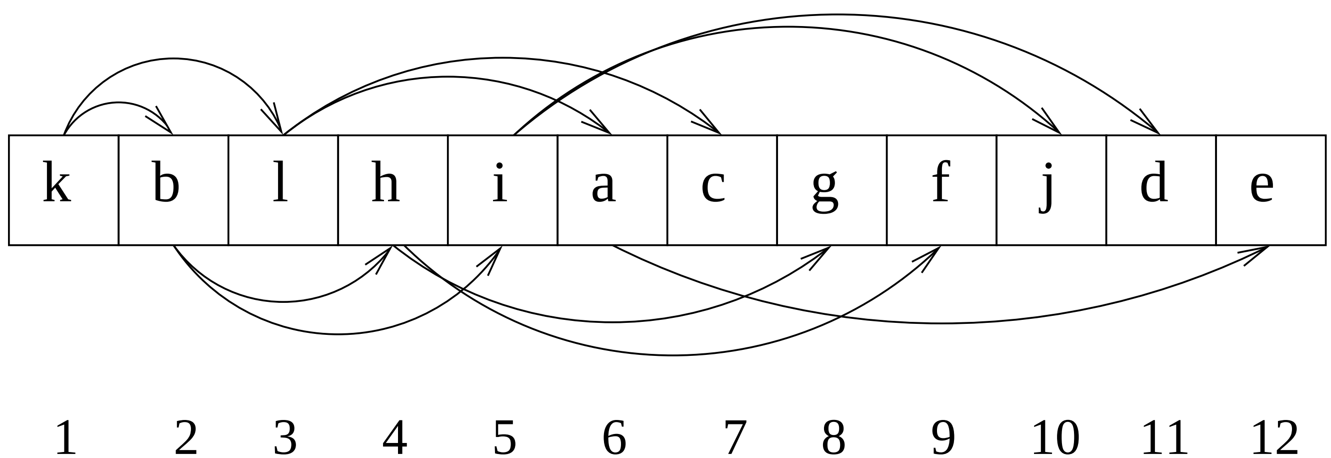
- $i \to 2i$

  left-shift by one of bit string representing i
- $i \to 2i + 1$

  left-shift by one plus adding a 1 to the last bit
- $i \to \lfloor i/2 \rfloor$

  right-shift by one


1
k
2
b
3
l
4
h
5
i
6
a
7
c
g
f
j
d
e
8
9
10
11
12


k b l h i a c g f j d e
1 2 3 4 5 6 7 8 9 10 11 12

This particular vertex numbering isn't the only requirement for the thing to be a proper heap

Two kinds: **min-heaps** and **max-heaps**

Both cases, values in nodes satisfy **heap property**

- **max-heap** with **max-heap property**:

  for every node i (other than root)

  $$A[\textsc{Parent}(i)] \geq A[i]$$

  meaning: value of node is **at most** value of parent, largest value is stored at root

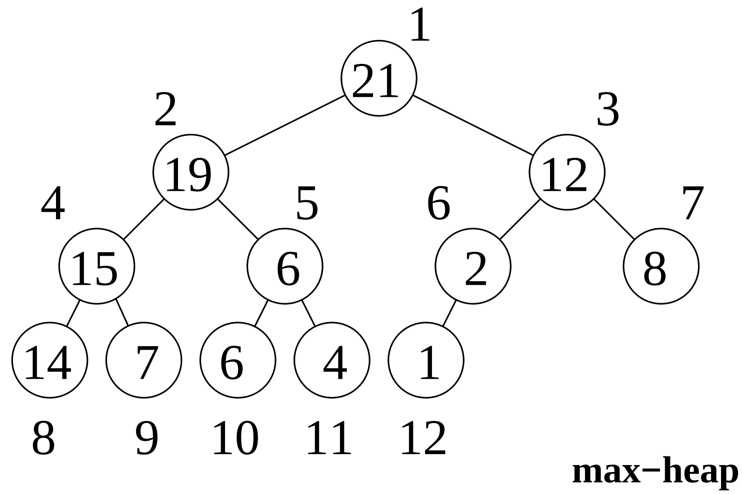
- **min-heap** with **min-heap property**:

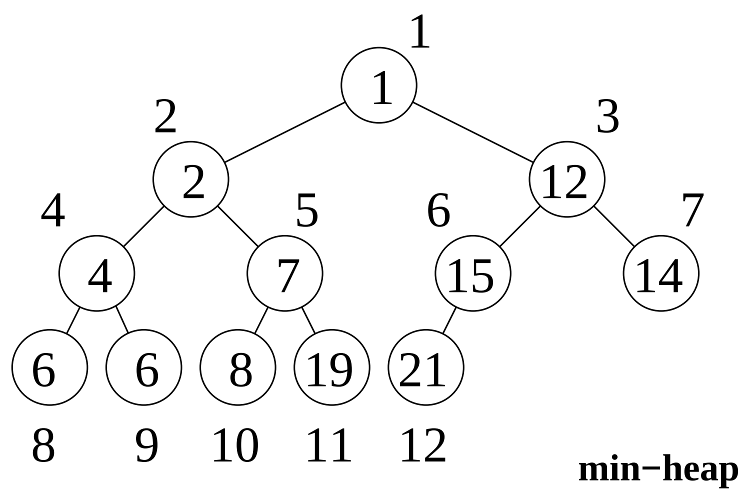
  for every node i (other than root)

  $$A[\textsc{Parent}(i)] \leq A[i]$$

  meaning: value of node is **at least** value of parent, smallest value is stored at root

Suppose given values 1,2,4,6,6,7,8,12,14,15,19,21





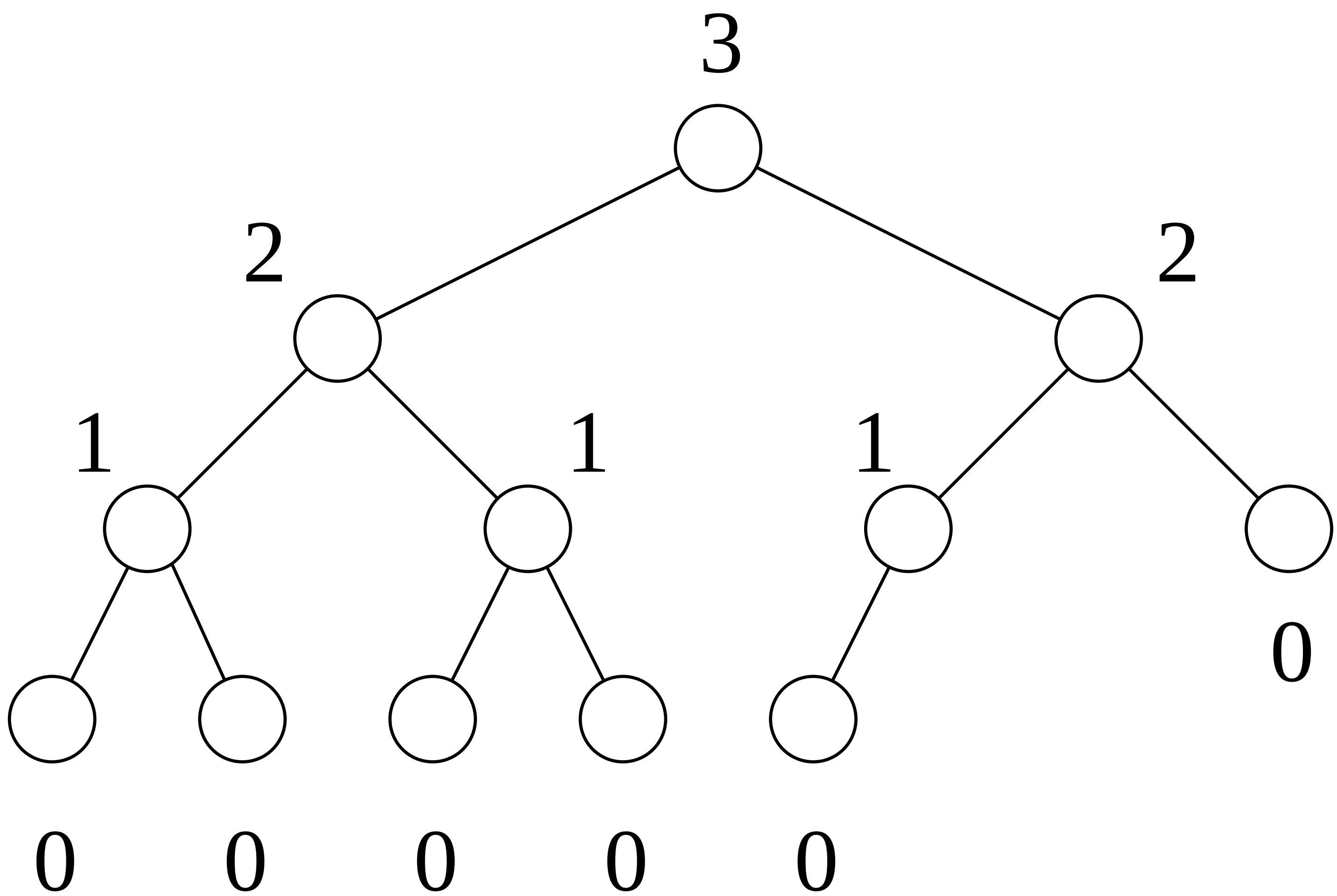
number inside vertices are values

numbers outside are array indices

Note: given fixed set of values, there are many possible proper min-heaps and max-heaps (except for what's at root)

We'll see applications for max-heaps (Heapsort) and min-heaps (priority queues, later)

If viewing heap as "ordinary" tree, define **height** of vertex as # of edges on longest simple downward path from vertex to some leaf



The height of a heap is the height of its root.

A heap of n elements is based on a complete binary tree, therefore its height is $\Theta(\log n)$

Assignment Problem 2.5. (deadline: May 20, 5:30pm)
What are the minimum and maximum numbers of elements in a heap of height h? Prove that an n-element heap has height $\lfloor \log_2 n \rfloor$.