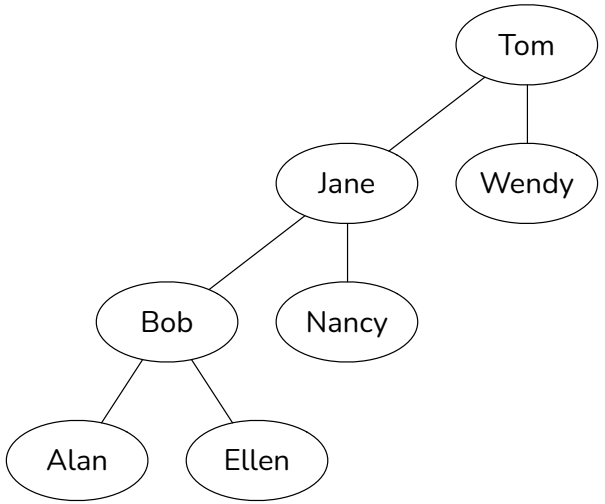
Red-Black Trees

Kristian Guillaumier

Binary Trees

- One way to index records using a search key is to use binary trees.
- If the tree is balanced then we can perform very efficient inserts, deletes, and lookups (binary search tree).
- However, the ‘basic’ binary tree can degenerate.



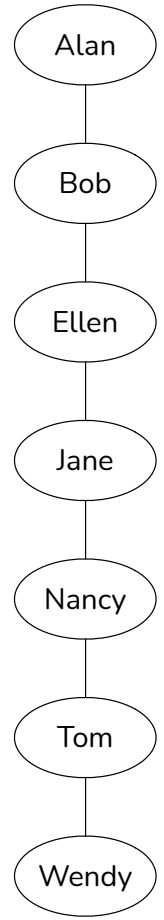
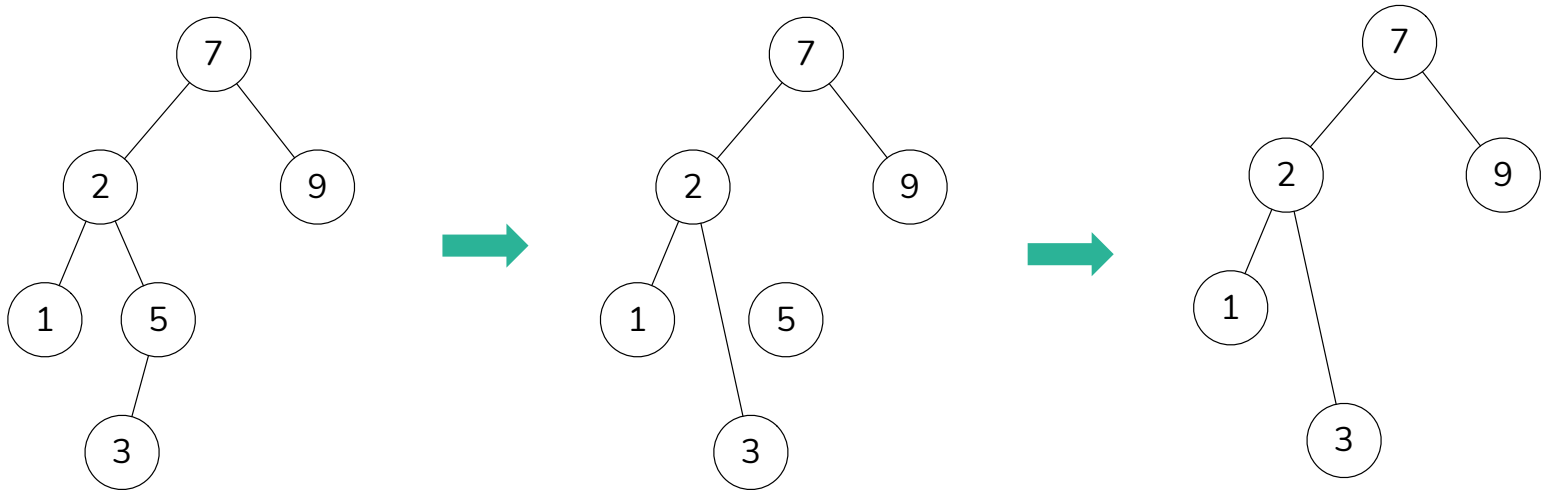


Image from Weiss

Some useful operations on binary trees

- Find minimum/maximum: trivial – start at root and branch left/right until there is no child.
- Find: recursive left/right decisions. Insert is a variation on find.
- Delete: most complex because it can disconnect the tree.
  - Scenario 1: Item is a leaf node – simply remove it since it will not disconnect the tree.
  - Scenario 2: Item X has only one child Y – set parent of X to point to Y and delete X…

Delete node 5:

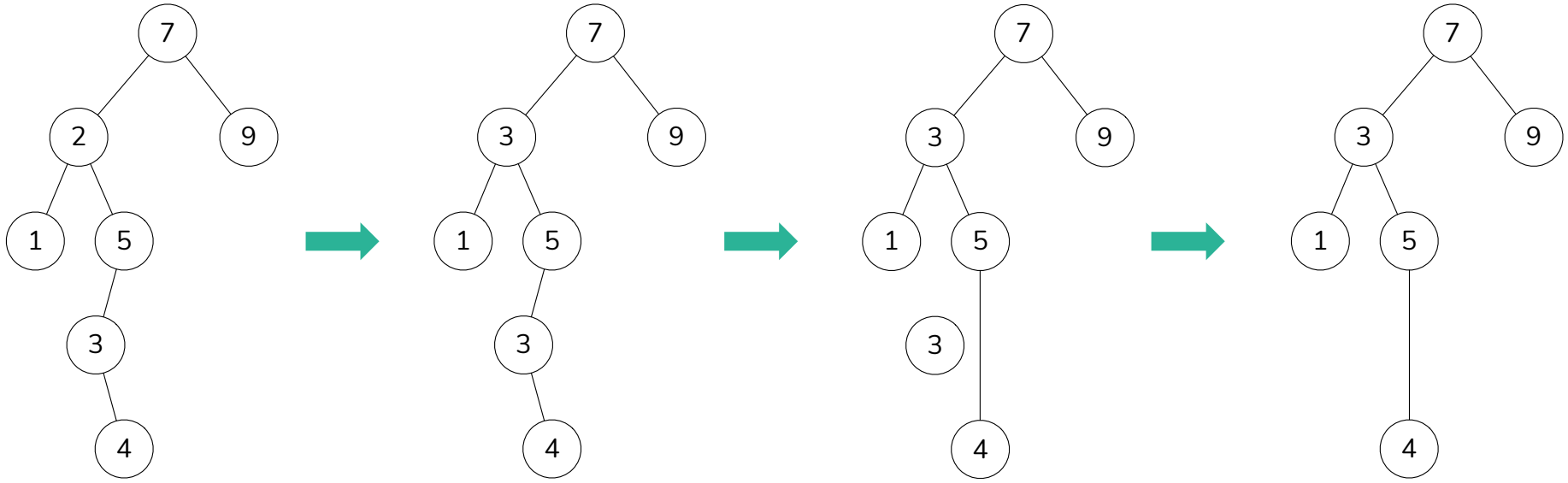


Example adapted from Weiss

...continued

- Delete scenario 3: node X has 2 children T^L and T^R,
  - Replace the item X with the smallest item Y in T^R.
  - The smallest item in T^R can be found efficiently.
  - Remove Y from T^R. Since Y in T^R can never have a left child, this will be simple.

Delete node 2:



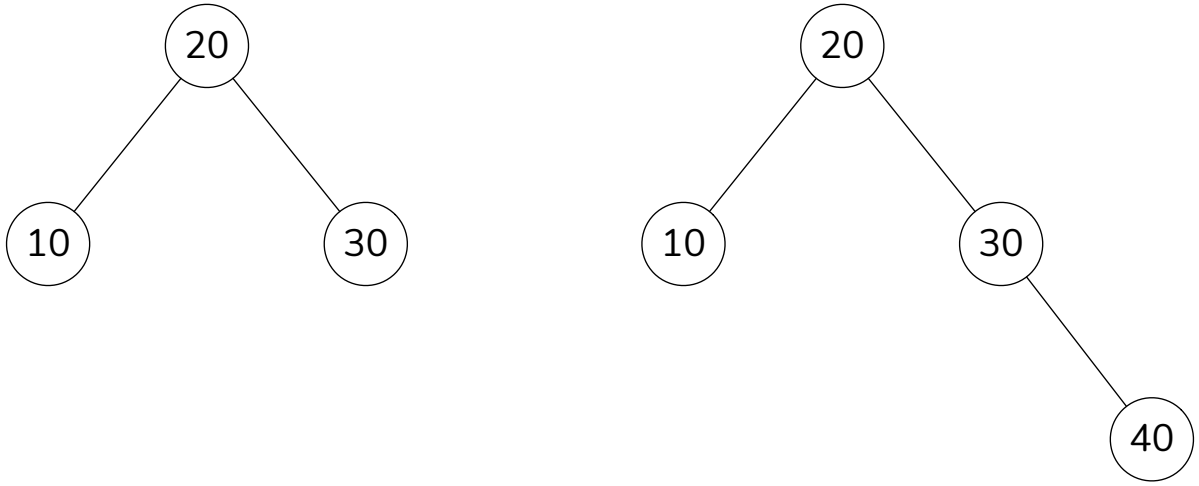
Example adapted from Weiss

Rotations in trees

- Recall AVL trees which maintain a height close to the minimum with little overhead.
- For any node in the tree the height of the left and right subtrees can differ by at most 1. The height of an empty tree is -1.
- After inserts or deletes the ‘shape’ of the tree is monitored to determine whether the AVL conditions have been violated.
- The height of the tree is ‘repaired’ using rotation operations.
- Note that not all inserts or deletes will necessarily cause a rotation to occur.

...continued

- Consider adding the value 40:



The tree is still balanced. No need to rotate.

...continued

- Consider adding the value 60:

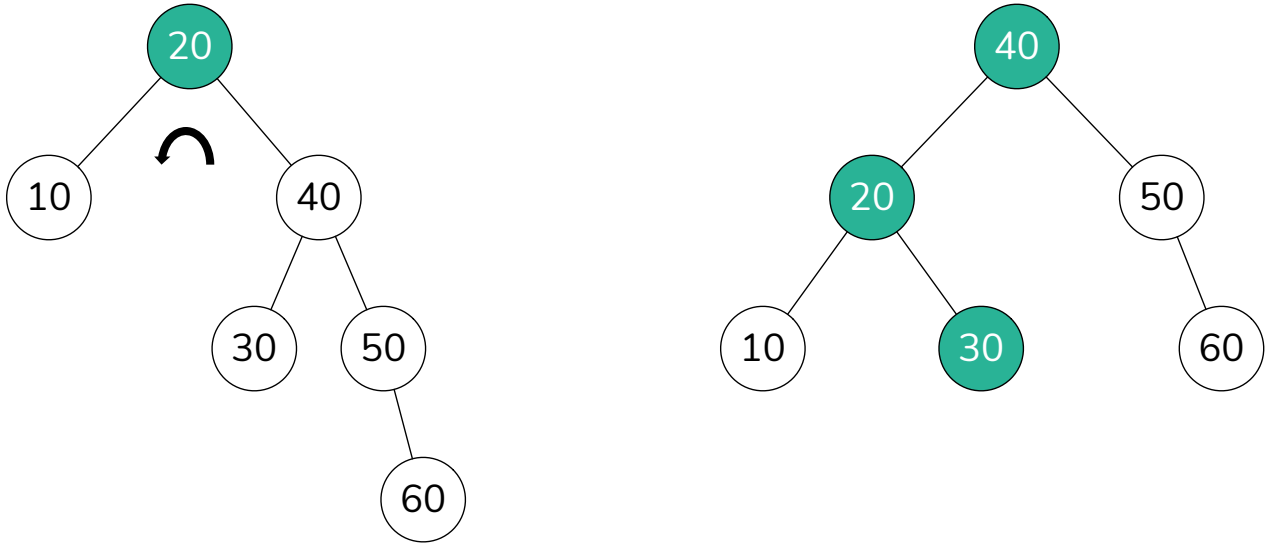


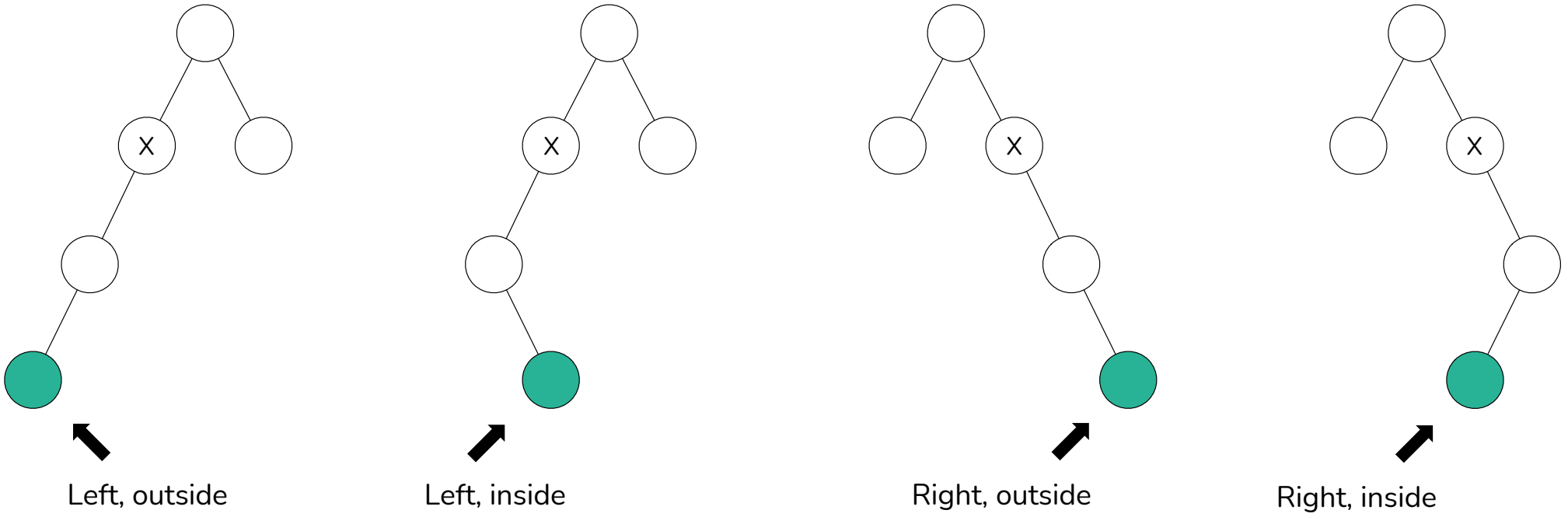
Image from Helman et al.

...continued

- We observe that after an insertion only a node on the path to the root might have the AVL property violated.
- We ‘walk up’ the tree from the insertion point until we find the node X that violates the AVL condition.
- The violation will occur in the following cases:
    1. Insertion in left subtree of left child of X.
    2. Insertion in right subtree of left child of X.
    3. Insertion in left subtree of right child of X.
    4. Insertion on right subtree of right child of X.

...continued

- Violations and notation:



...continued

- Example of LL rotation after inserting 1 (RR is the same by symmetry):

```
RotateLL(Node n) → Node {
    Node k  = n.Left
    n.Left  = k.Right
    k.Right = n
    return k
}
```

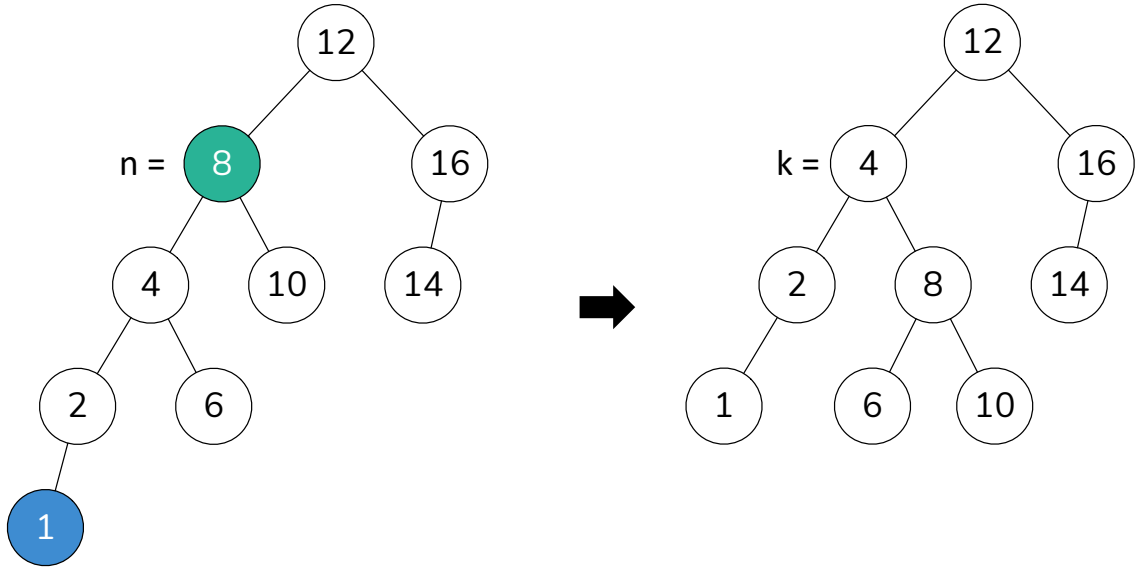
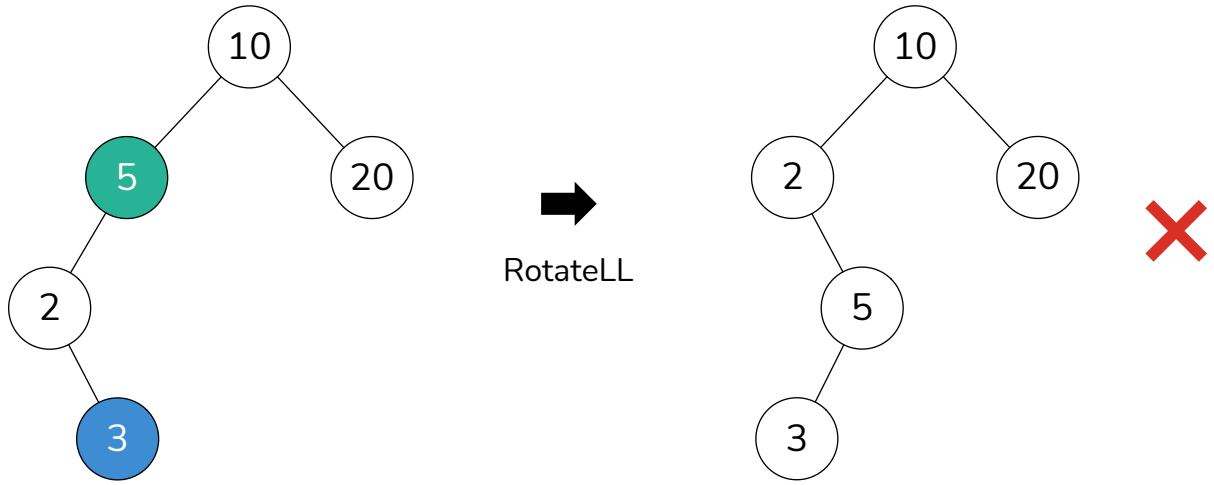


Image from Helman et al.

...continued

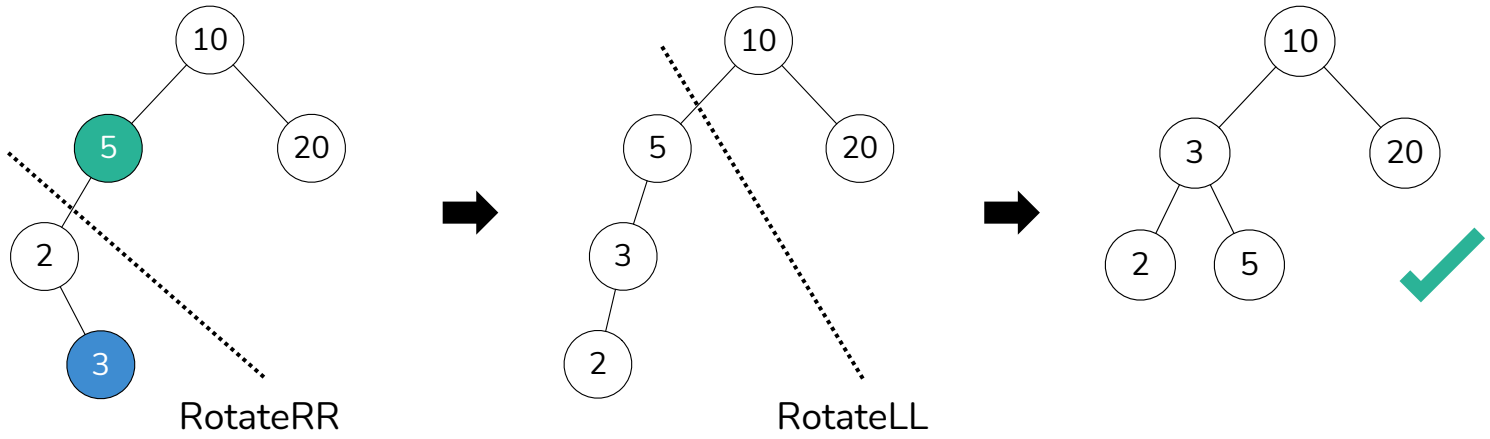
- RotateLL and RotateRR takes care of 'left and right outside' cases respectively, but not 'left and right inside' cases.
- Consider what happens if we execute RotateLL on a 'left, inside' case:



...continued

- The solution is to use double rotation:

```
RotateLR(Node n) → Node {
    n.Left = RotateRR(n.Left)
    return RotateLL(n)
}
```



Red-black trees (RBTs)

- A simple AVL tree implementation needs to:
  - Recursively go down the tree to find the insertion point.
  - Then go up to find the height information and rebalance (during recursive unwinding).
- RBTs can be implemented iteratively using single top-down pass in a relatively simple way. RBTs also tend to require fewer rotation operations during insertion and deletion.

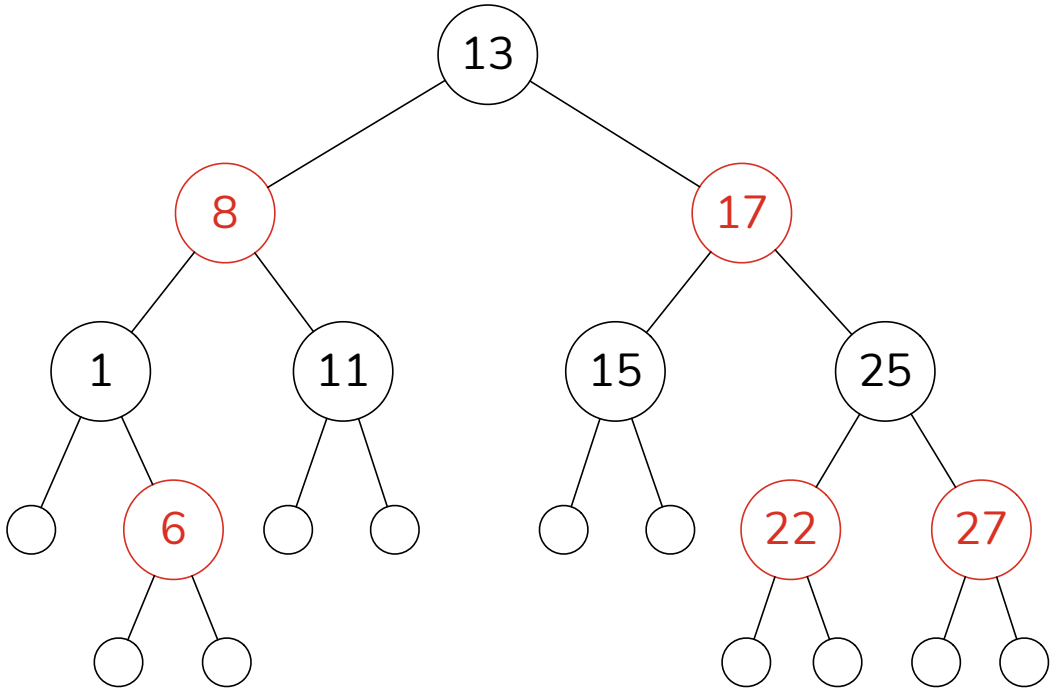
Properties of RBTs

A red-black tree is a self-balancing binary tree.

1. Nodes are coloured red or black.
2. The root is black.
3. Children of a red node are always black (there can never be two consecutive red nodes in a path).
4. Every path from any node to the null leaves has the same number of black nodes.
5. Null leaves are black.

An example

1. Nodes are coloured red or black.
2. The root is black.
3. Children of a red node are always black (there can never be two consecutive red nodes in a path).
4. Every path from any node to the null leaves has the same number of black nodes.
5. Null leaves are black.



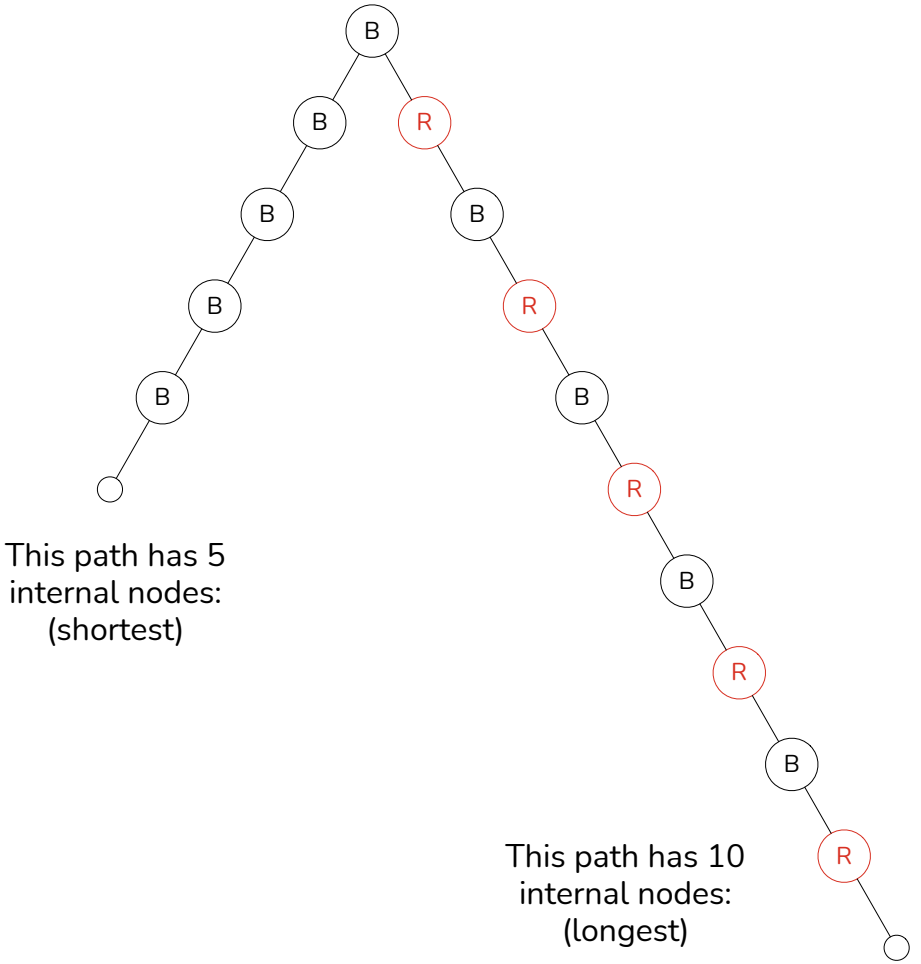
Note that the RBT properties do not specify that it must also be a BST – this example just happens to be a BST as well.

Image adapted from Wikipedia

A very important observation

- The shortest path will consist of only black nodes.
- The longest path will consist of alternating black and red nodes. This is because there can never be two consecutive red nodes due to property 3.

In other words, the longest path can never be more than twice the length of the shortest path.



The height of an RBT

- We want to show that the height h of a tree with n internal nodes is $O(\log_2 n)$.
- We specify internal nodes because the null leaves are not really nodes. The internal nodes are the keys.
- We will provide a sketch of the proof.
- The first step is the merge the tree – merge red nodes into the black ones.

Merging red nodes into black parents

- Consider this tree…

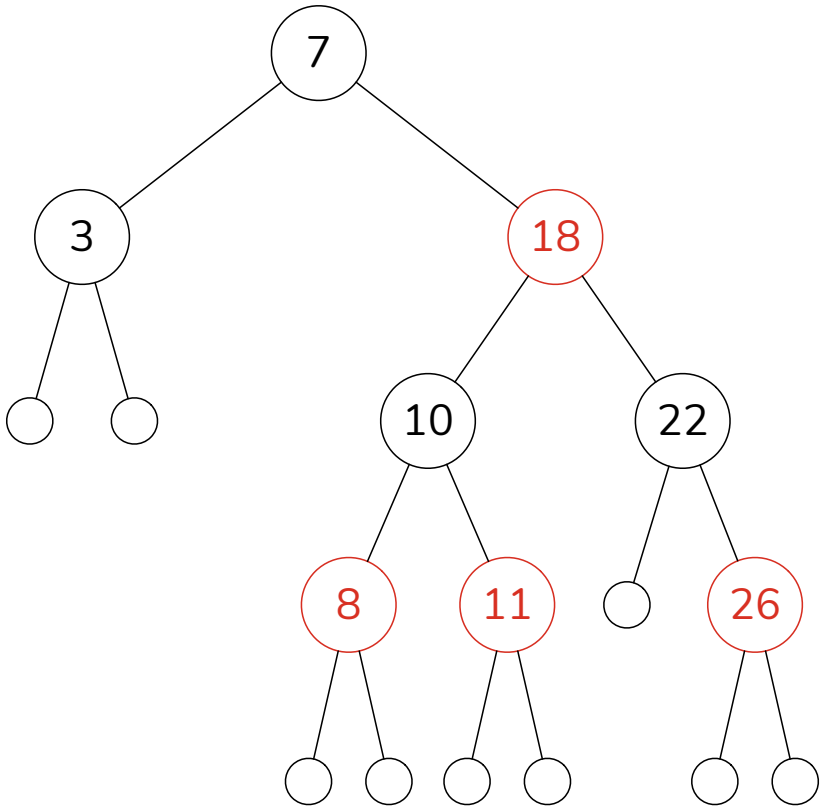
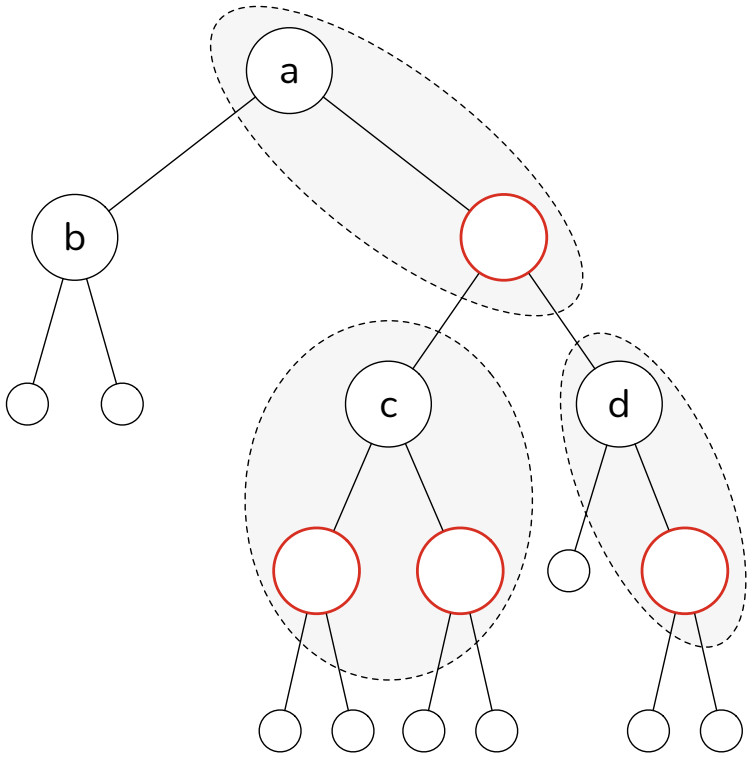
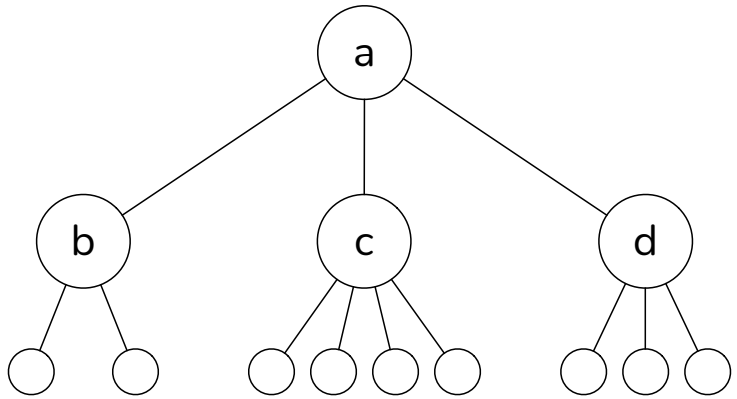


Image adapted from Eric Demaine

...continued

- After merging we get...





This is a 2-3-4 tree!

Image adapted from Eric Demaine

...continued

- After merging we get a 2-3-4 tree:
  - All nodes have 2, 3, or 4 children (except leaves who have none).
  - All leaves have the same height – because of property 4. This makes the tree balanced.
- Let:
  - h be the height of the original RBT.
  - h' be the height of the merged 2-3-4 tree.

The height of the merged tree

- Note that, in general, there are always $n + 1$ leaves in a tree that contains n internal (key) nodes.
- This is because every internal node has exactly 2 children.
- In the previous example, there are 8 internal nodes and 9 (i.e., $n + 1$) null leaves.

Proof: #Leaves = #Internal + 1

- Let n = number of internal nodes, and l = number of leaf nodes.
- Step 1: place 2 ‘tokens’ on each internal node…

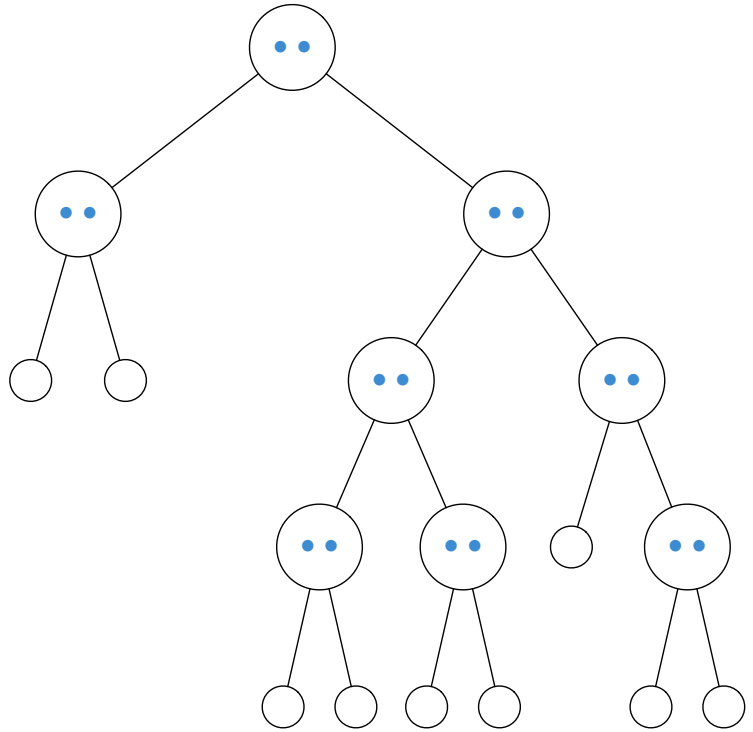


Image adapted from Eric Demaine

...continued

- Step 2: move one token to each child. Now all nodes except for the parent will have a token.

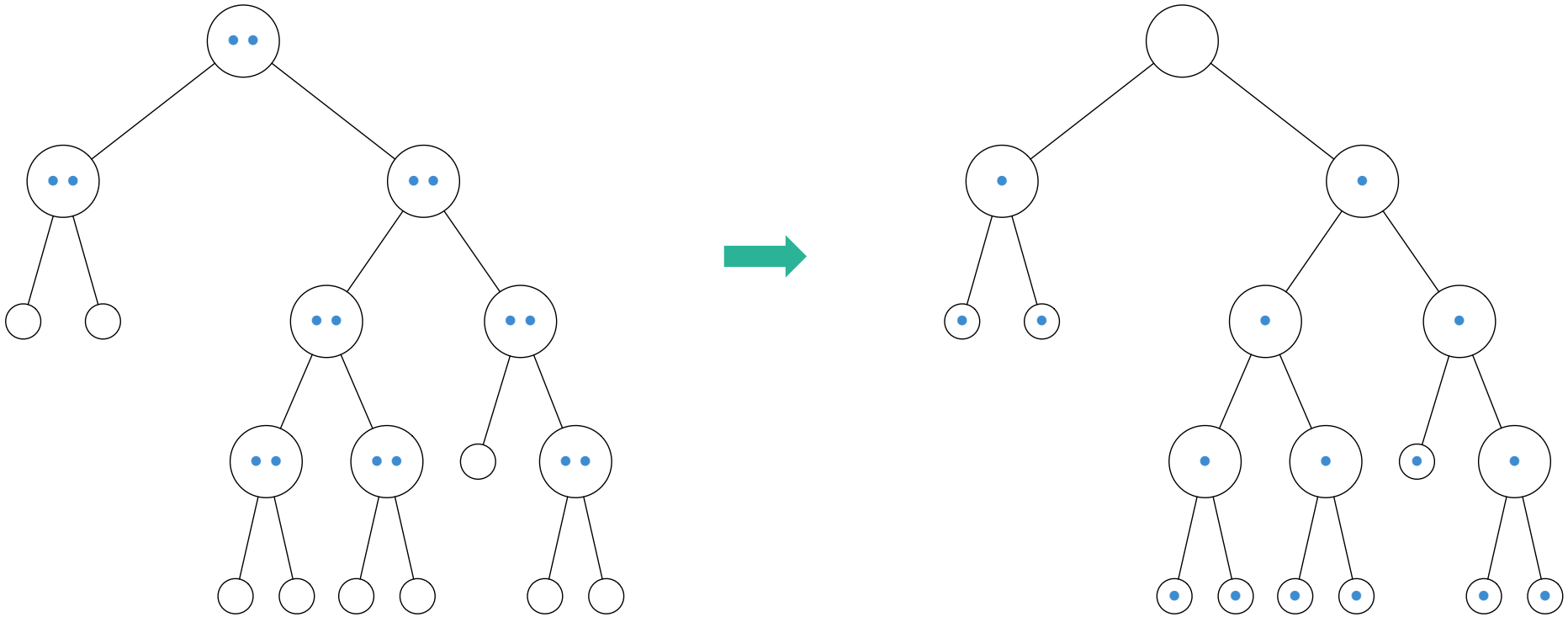


Image adapted from Eric Demaine

...continued

- We started with $2n$ tokens.
- We finished with l tokens (1 token per leaf) and $n-1$ tokens (1 token per internal excluding parent).
- Since we did not add or remove any tokens in the process, the number of tokens that we started with must be the same as when we finished...
- Rearrange:
  - $l + n - 1 = 2n$
  - $l + n - 1 = n + n$
  - $l - 1 = n$
  - $l = n + 1$

Height of RBT

- For a general 2-3-4 tree having height h':
  - The minimum number of leaves is $2^{h'}$ and the maximum number of leaves is $4^{h'}$.
- We know the merged red-black tree has $n+1$ leaves. So, the 2-3-4 tree has $n+1$ leaves too.
- So, we know that in my merged red-black tree, $2^{h'} \leq n+1$.
- If we log both sides, we get $h' \leq \log_2(n+1)$.
- Remember that the height of the merged RBT represents shortest possible path length (by rule 4) and earlier we showed that the longest possible path length is twice the shortest one.
- So, for the red-black tree having height h we get $h \leq 2 \times \log_2(n+1)$

So far...

- We have shown that we can perform operations in $O(\log_2 n)$ time if we follow the red-black tree rules.
- The next step is to create the insertion and deletion operations such that the red-black tree properties are not violated.

An insert strategy (this will not work in all cases)

- Use a normal BST search to find the insert location.
- We have a choice to make the node we insert either red or black.
- If we make the new node black, we violate rule 4 (the number of black nodes in any path is the same).
- If we make the node red, we might violate property 3 (consecutive reds).
- Since we must make a choice, we will go for red since a violation doesn't always occur, and such a violation happens to be easier to fix than a black violation.

A note on inserting

- If we insert 2 in this tree, we are OK. No rules are violated, and we are done (remember that we choose to always insert as red).
- In other words, if the parent is black, insertion is trivial.

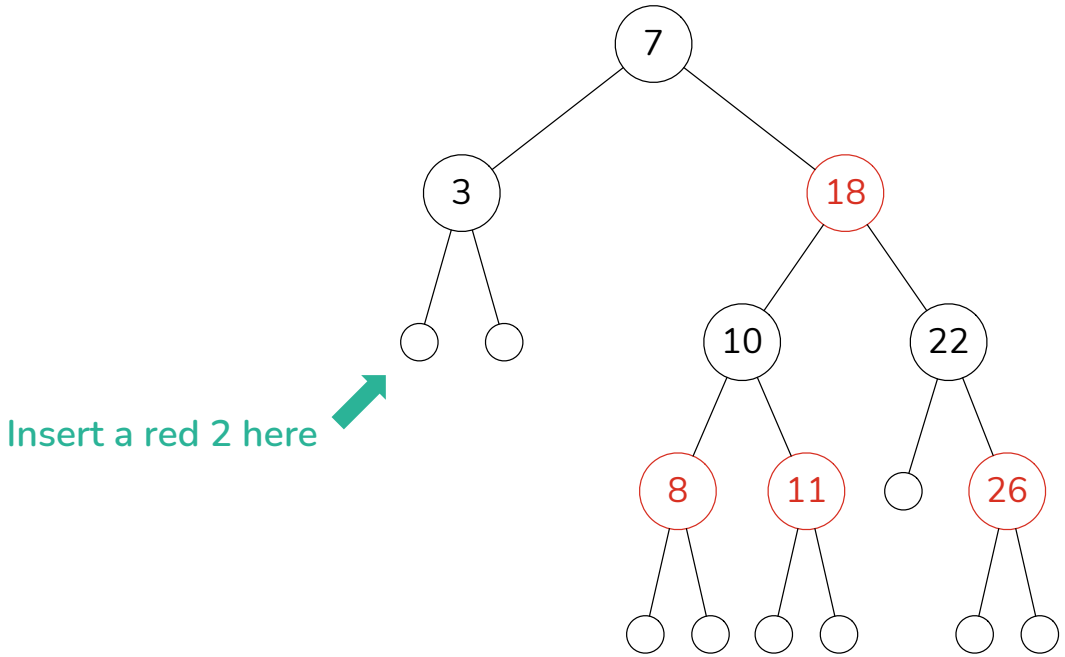


Image adapted from Eric Demaine

Inserting problem

- If we add 15, we have a problem (we get a 11-15 red+red violation).

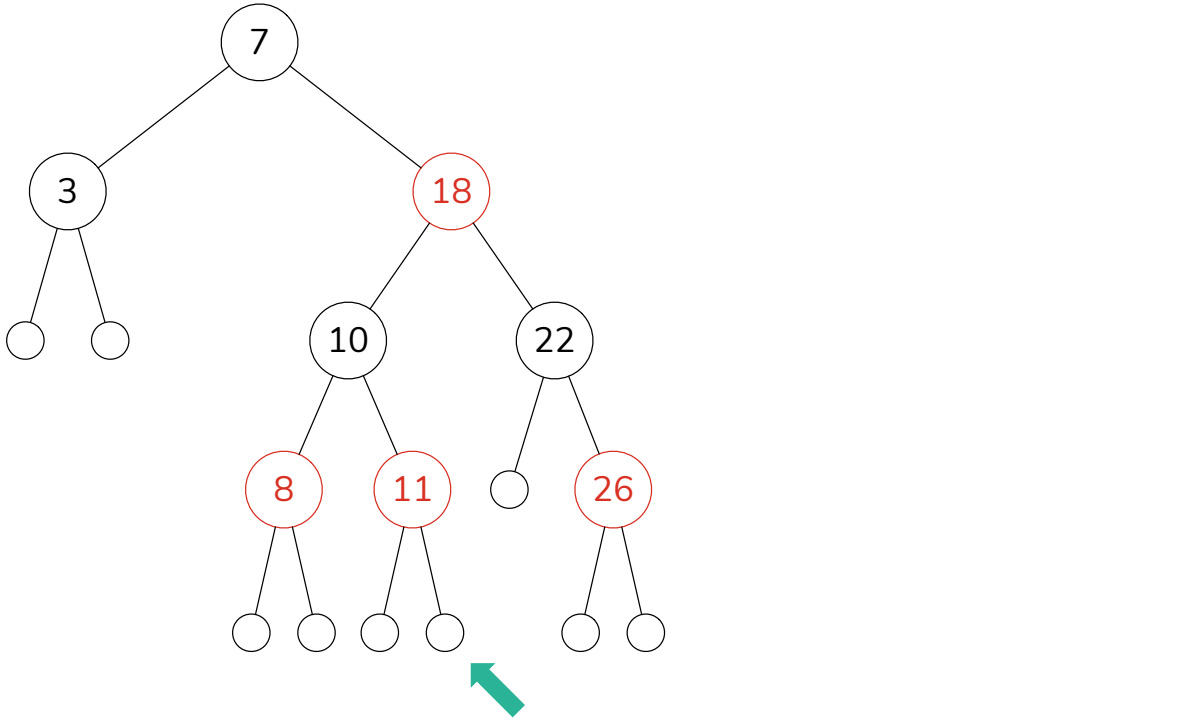


Image adapted from Eric Demaine

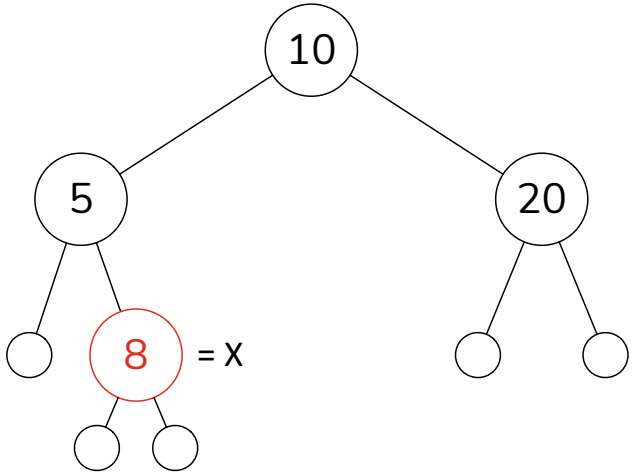
Our insert strategy so far…

- Always insert a red node X.
- If this is the root, colour it black and we are ready.
- If the parent P is black, we are ready.
- If the parent is red and the uncle U is black:
  - If relative to the grand parent G, X is outside:
    - Do an LL (or RR by symmetry if we inserted to right of P) and recolour old root (G) as red and new root to black.
  - If relative to G, X is inside:
    - Do an LR (or RL by symmetry) and recolour as above.
- If both the parent and uncle are red, then the scheme above will not work, and we will have to recursively fix further violations.

Some examples

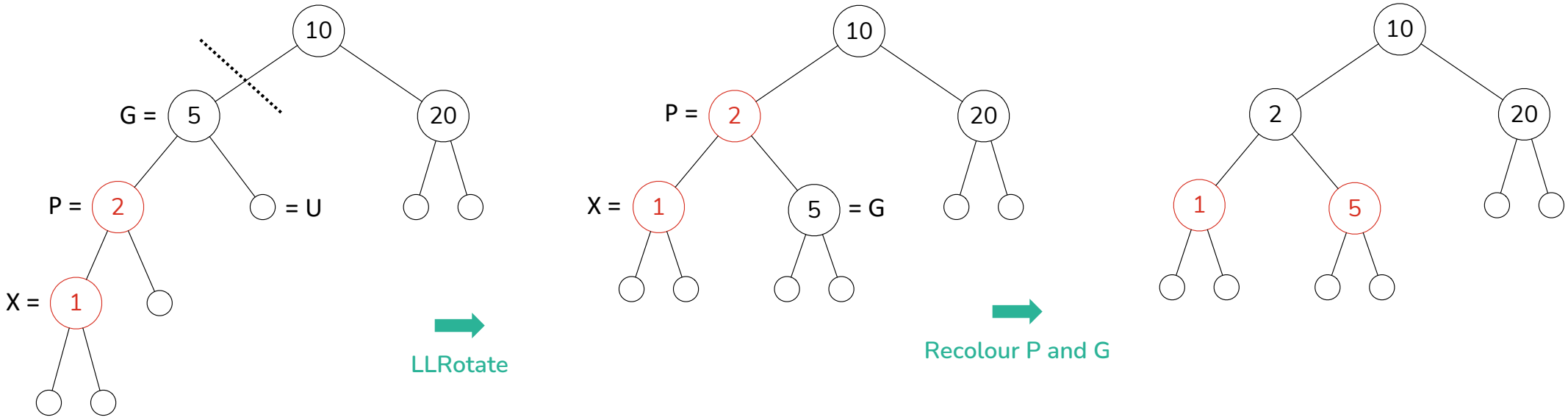
- The new node is the root – insert red X, recolour to black:

- The parent is black – insert red X and we're done:



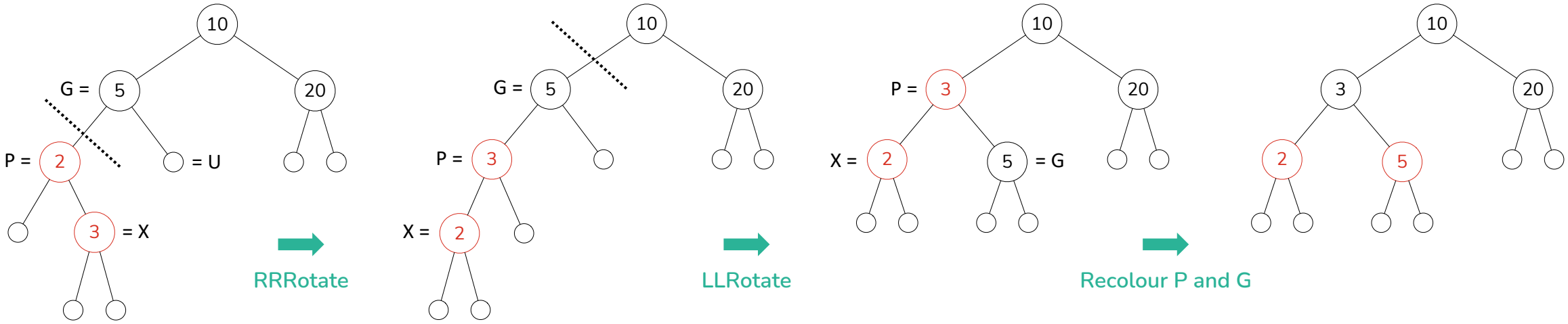
...continued

- Parent P is red, uncle U is black, X is outside grandparent G:



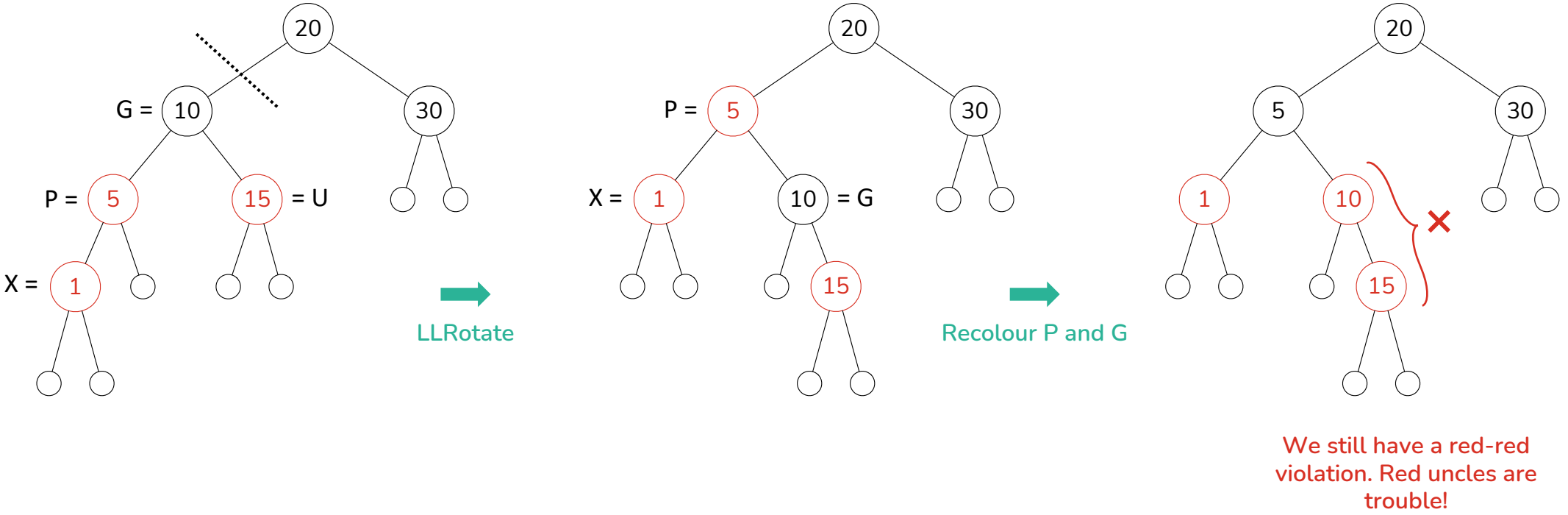
...continued

- Parent P is red, uncle U is black, X is inside G:



...continued

- Parent P is red, uncle U is red, X is outside G:



A working top-down insert

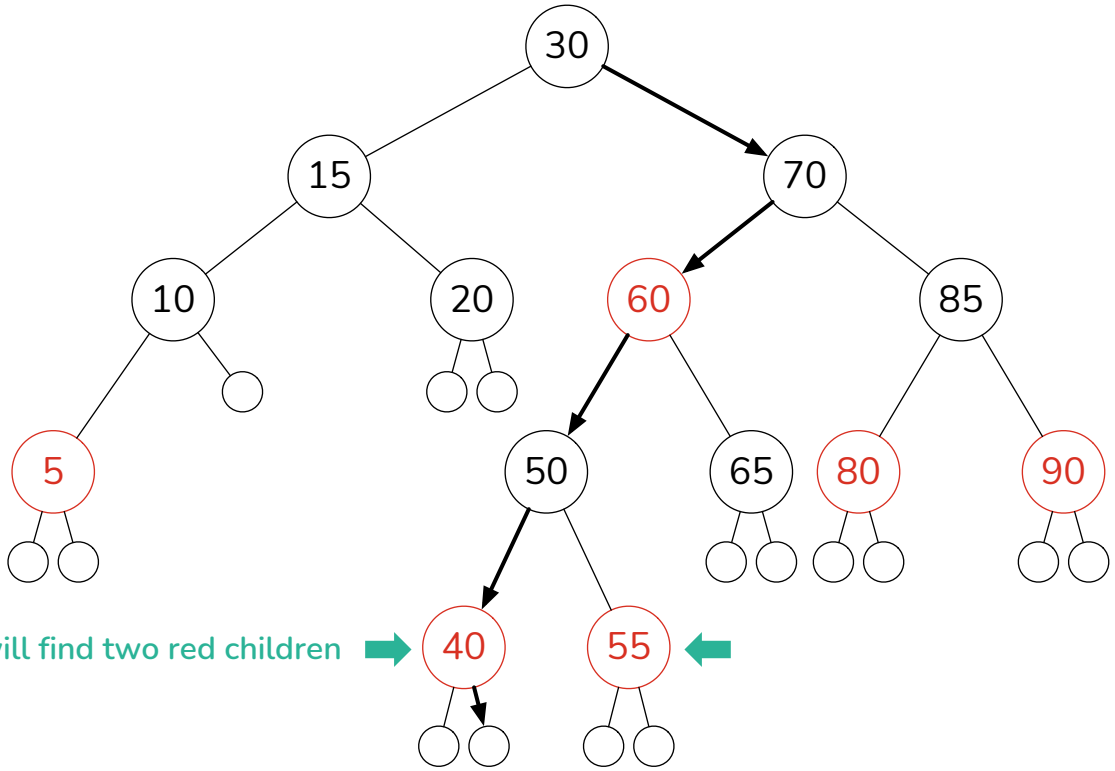
- When comparing AVL trees to RBTs, AVL trees are shown to be shallower.
  - AVL tree height <= 1.44 × $\log_2(n)$.
  - RBT height <= 2 × $\log_2(n)$.
  - So, for searches AVL trees may be slightly faster.
- AVL trees require a pass down to find the location of the new item and another pass up to rebalance the tree.
- So far, it appears that the RBT has the same problem: an insert and rebalance path is required.
- Additionally, RBTs so far require recursion to find the location and rebalance up.
- However, a modification solves this – top-down insertion.

...continued

- The basis of the algorithm is ensuring that an uncle is never red and then fix with one single or double rotation.
- When searching for the location where to place our new node X, if we find a node Y with 2 red children, we set Y to red and the children to black.
- If Y was the root, we will violate rule 2 but this is easily fixed at the end.
- This way we are guaranteeing that a red uncle can never exist.
- However, if Y's parent is red, we introduce a red-red violation. To fix this we do a single rotation (if outside) or a double rotation (if inside).

Example

- Let's start from this tree. We want to insert 45 and, on the way down, we see 50 which has two red children. Flip their colours.

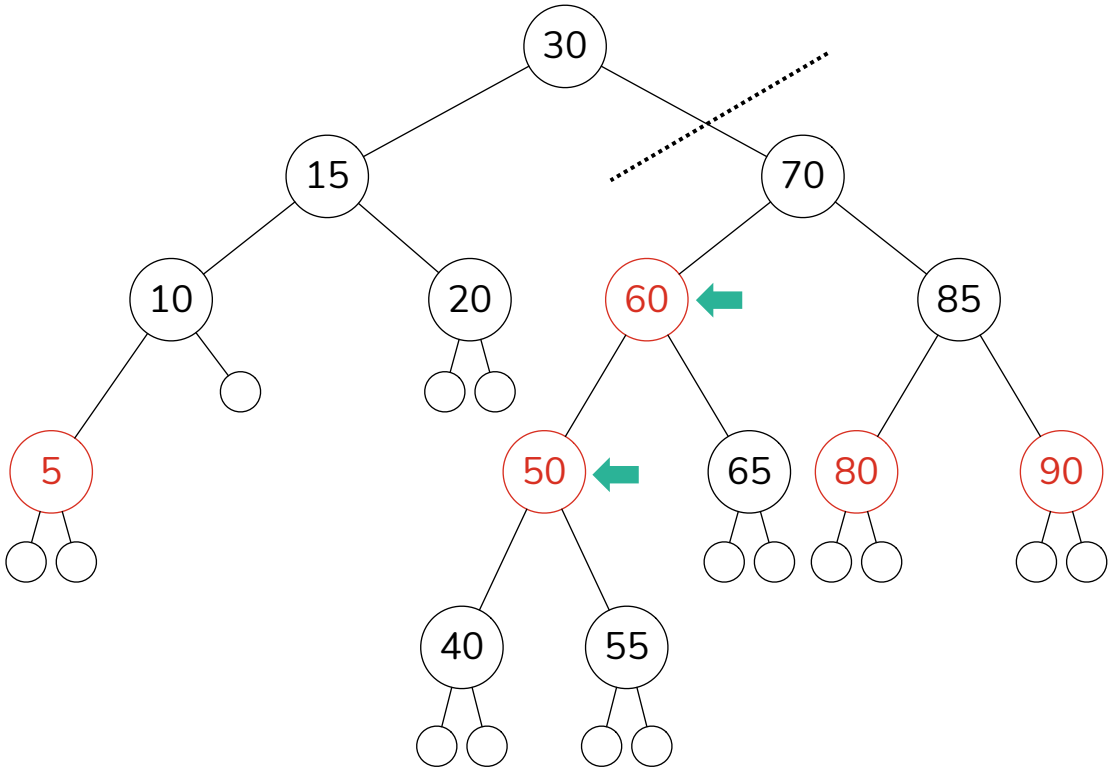


On the way down, at node 50, we will find two red children

Example adapted from Weiss

...continued

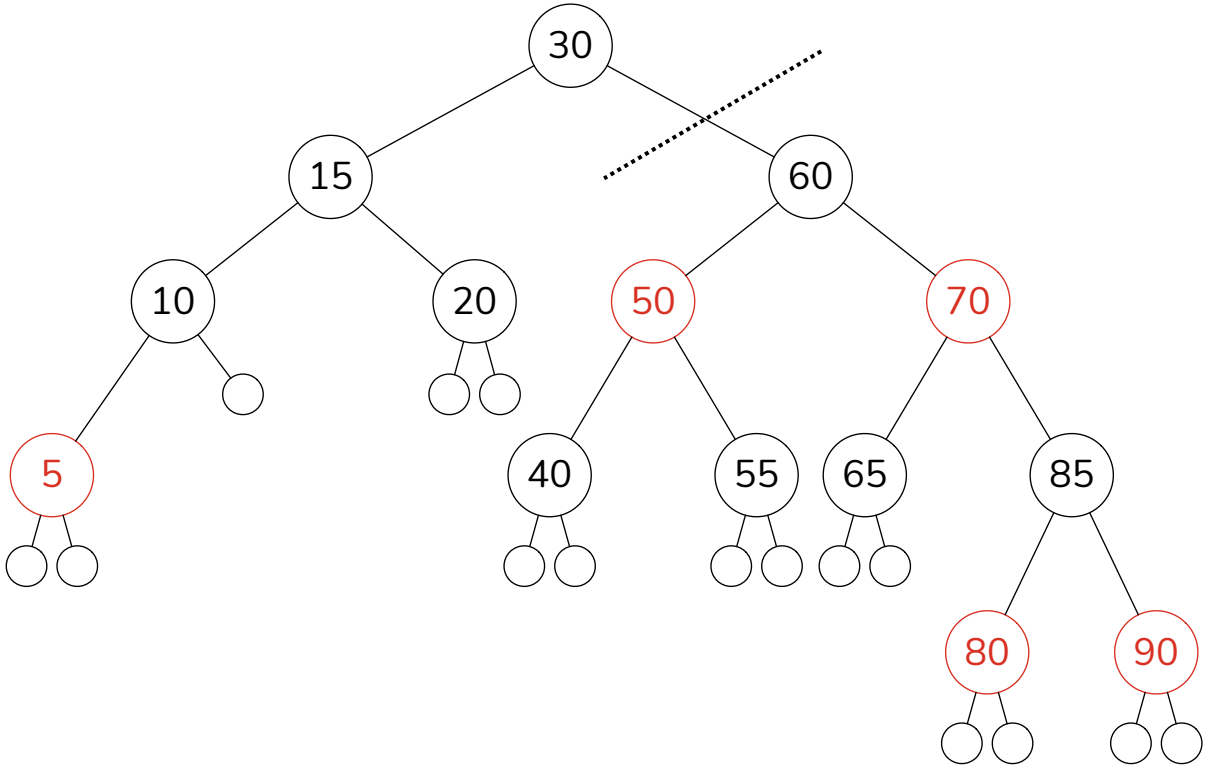
- Flipped children of 50 to black and set 50 to red. Now we have a violation at 60 and 50 because they are both red. 50 is outside of 70 so we do an LL rotation and recolour.



Example adapted from Weiss

...continued

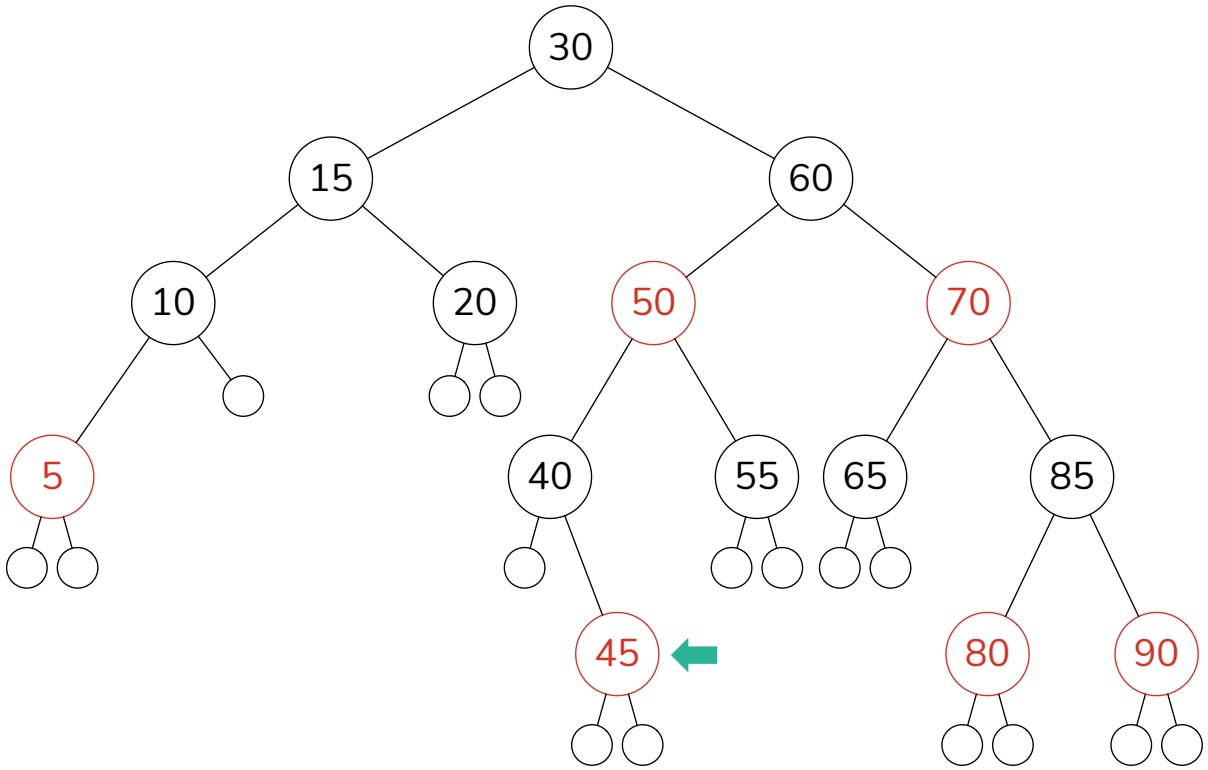
- Result of LL rotation and recolour at 70 fixes the problem.



Example adapted from Weiss

...continued

- We can proceed to insert 45. The parent is black, so we are done. If parent was red do another rotation. If the root was made red, make it black.



Example adapted from Weiss

Deletion

- Remember the 3 cases of ‘normal’ BST deletion.
- In RBT if we delete a red leaf node then there is no problem – it cannot violate any rules.
- If we delete a black node, we will cause a problem.
- We can implement a top-down deletion which is iterative and ‘fixes’ on the way down.
- Deletion is left as an exercise to the student (it is simple enough).

Further reading

- These notes should be supplemented by:
  - Introduction to Algorithms (Clifford Stein, Thomas H Cormen, Ronald L Rivest, Charles E Leiserson – MIT Press)
  - Red-Black Tree Online Videos (Erik Demaine, Massachusetts Institute of Technology, MIT, http://videolectures.net/mit6046jf05_demaine_lec10/)

Data Compression Basics

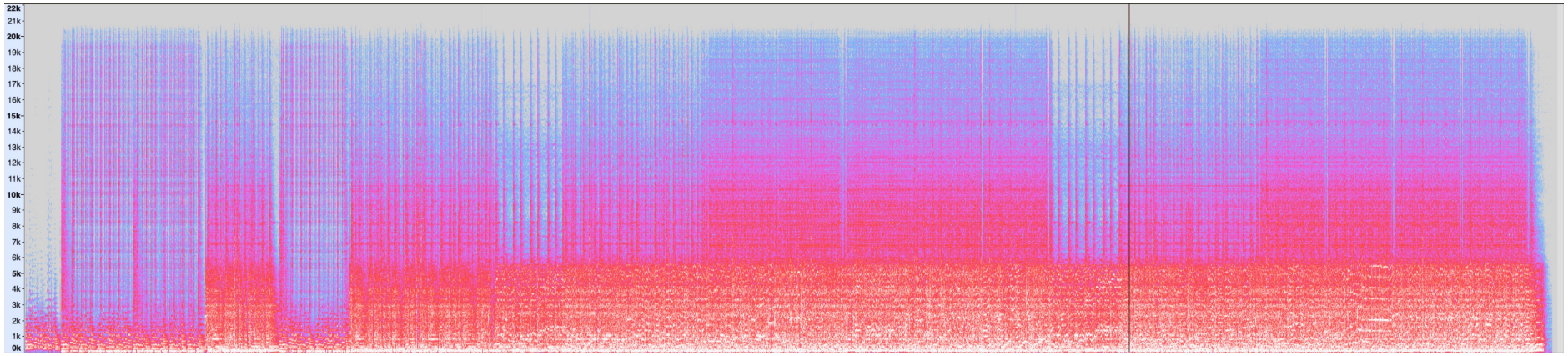
Kristian Guillaumier

We will not be covering lossy compression

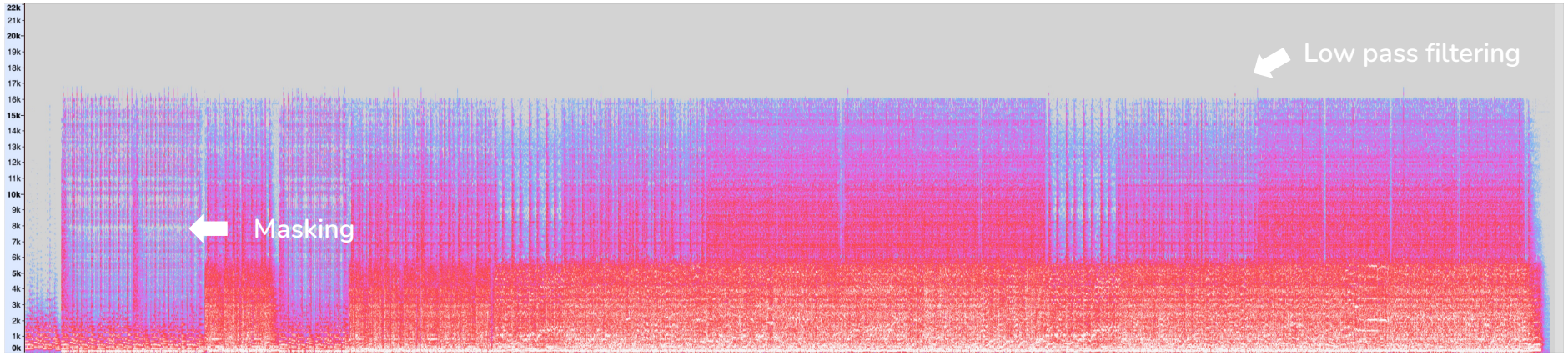
PNG lossless

JPG lossy

...continued



CDDA 44.1kHz, 16-bit



MP3 128kbps, 44.1kHz, 16-bit

Data compression

- The ASCII character set has about 100 printable characters.
- We can easily see that the number of bits required to encode the 100 characters in binary is given by $\lceil \log_2 100 \rceil = \lceil 6.64 \rceil = 7$ bits.
- The seven bits allow for 128 characters.
- ASCII adds an 8th bit for parity checking (or extensions).

In general, if the size of the character set is C then we need $\lceil log_2 C \rceil$ bits to encode each character.

...continued

- Let's say that we have a file that can only contain a, e, i, s, t, sp (space) and nl (new line) i.e., my character set C has 7 characters.
- Let's say that the file contains 10 'a' characters, 15 'e' characters, and so on...
- The table below shows some stats...

Character	Code	Frequency	Total bits
a	000	10	30
e	001	15	45
i	010	12	36
s	011	3	9
t	100	4	12
sp	101	13	39
nl	110	1	3
		Total:	174

← This is the file size

Example from Weiss

...continued

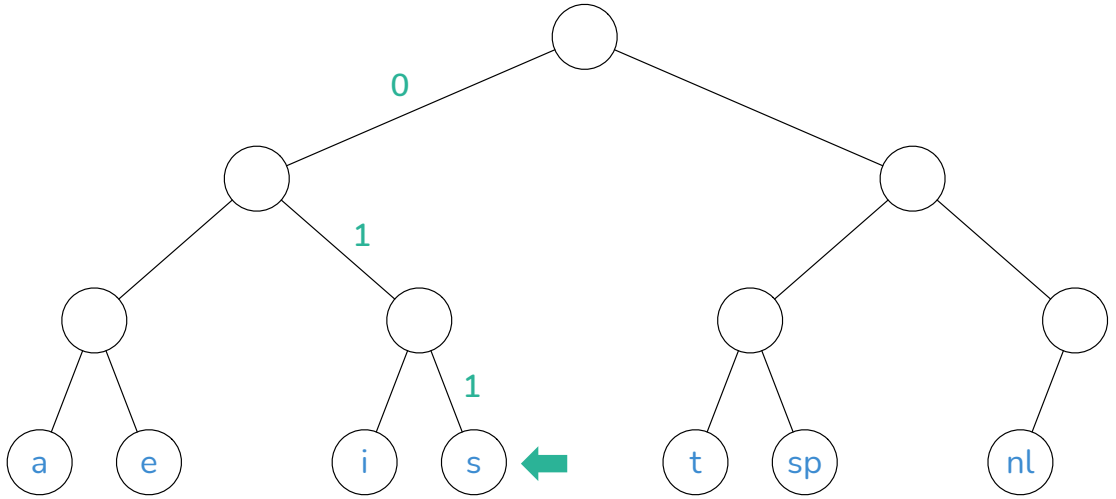
- In many data sequences (such text files) there is usually a large disparity between the most commonly occurring characters and the least commonly occurring ones.
- For example, the letter ‘e’ is more common than the letter ‘q’ in English text.
- We will use this property to our advantage and formulate a strategy where:
  - Character code length varies from one character to another.
  - Commonly used characters will have shorter codes.
  - Rarely used characters will have longer codes.

…continued

- Note that if all characters occur with the same frequency, good compression cannot be achieved.
  - Try compressing a file with random data – you'll barely get anything.
  - That is why plain text files compress much better than binaries.

Prefix codes

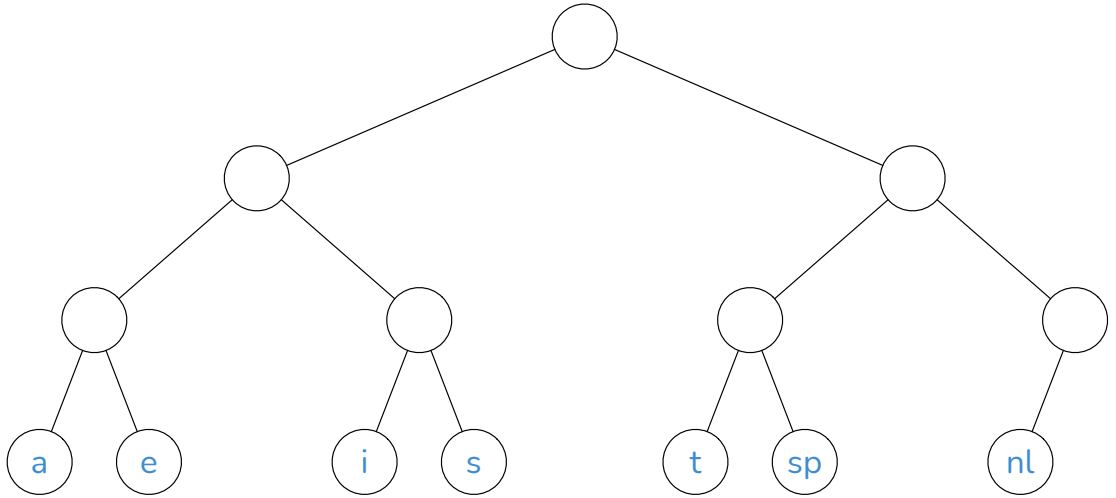
- The binary code shown in the table before can be represented by a prefix tree (values in the leaf nodes) below.
- Left branching indicates a 0, right branching indicates a 1. So, left → right → right is 011 and will encode the character ‘s’.



Example from Weiss

...continued

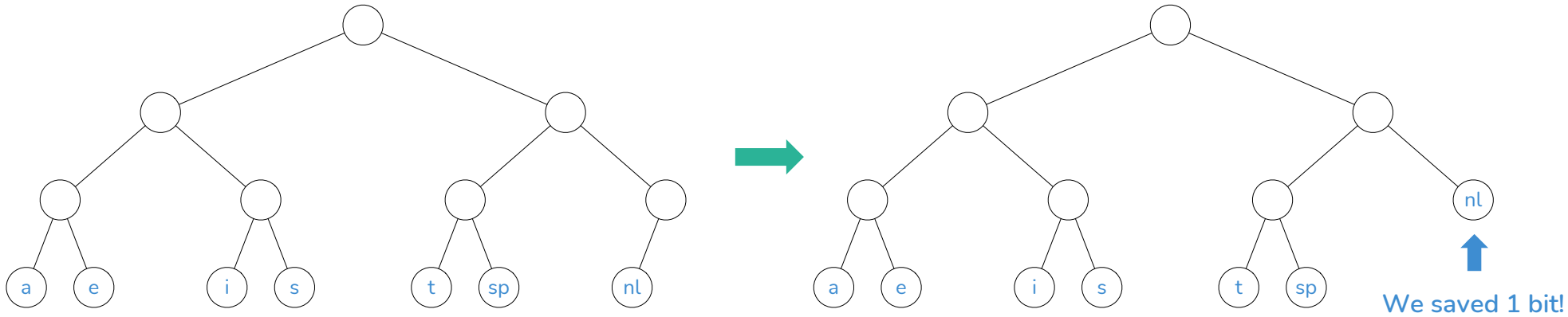
- If a character c is found at depth d and occurs f times, then $Cost(c) = d \times f$. In this example $Cost(\text{“}s\text{”}) = 3 \times 3 = 9$.
- The cost of the tree is the sum of the cost of every character in it. In this example, the cost will be 174.



Example from Weiss

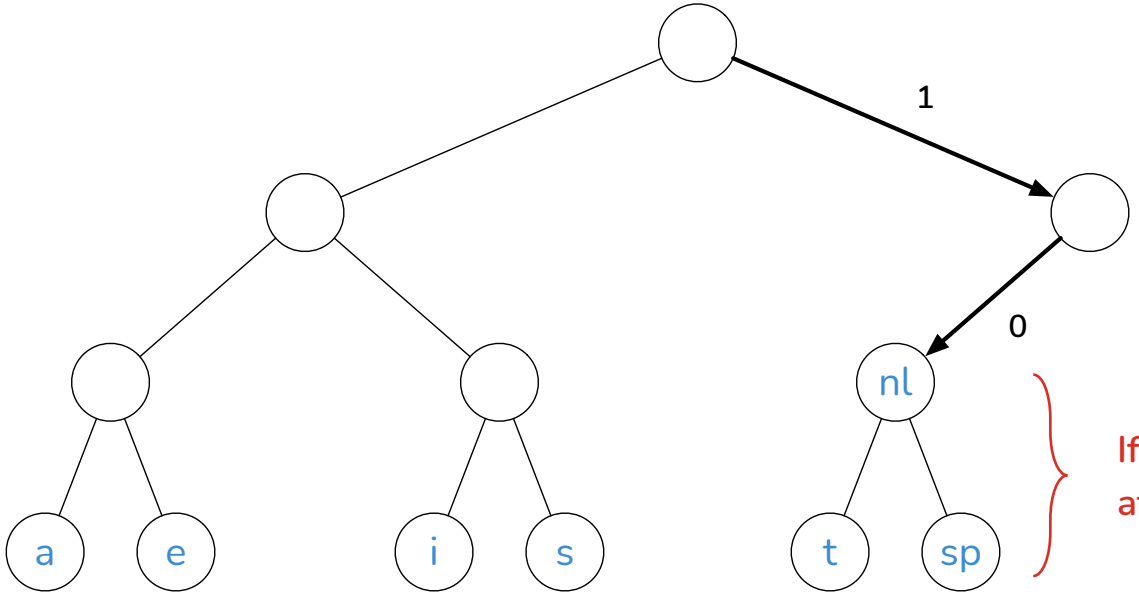
...continued

- If we reorganise the tree a bit and make leaves shallower, we we would also reduce the total cost of the tree.
- In this example, we shift the new line character one level up (without 'damaging' anything) and reduce the total cost of the tree from 174 to 173.



...continued

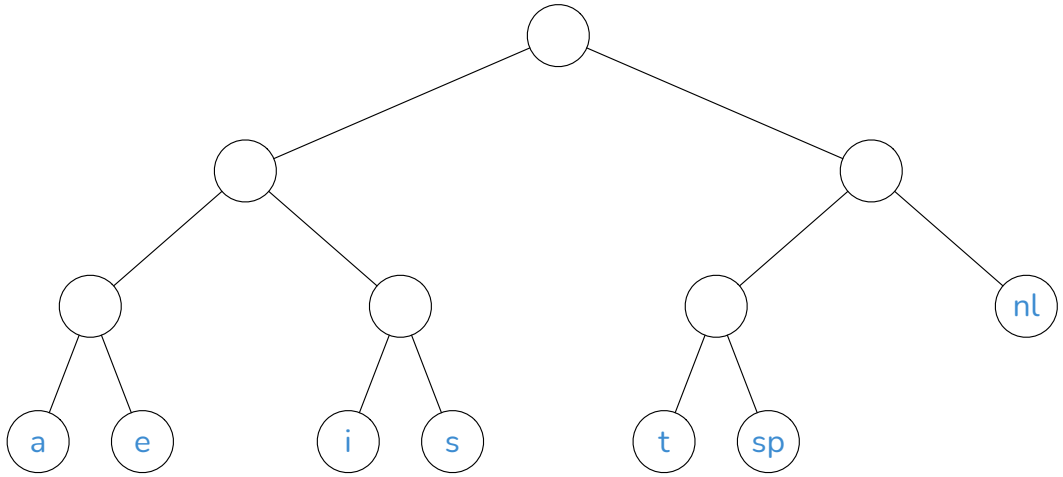
- Clearly, all characters must be in the leaves otherwise we will get ambiguous encoding (caused by common prefixes – this is very bad because we cannot decode).



If we read the sequence 10, do we stop at nl, or do we proceed to 't' or sp?

A simple example

- Consider the input string 010011111000101100010011 that we want to decode.
- We treat every 0 or 1 as a left or right move. When we reach a leave, we decoded a character. Then continue, restarting from the root.
- In this tree 010 (L → R → L) gives ‘i’, continuing with 011 (L → R → R) gives us ‘s’, continuing with 11 (R → R) gives us a nl, etc… Note that the character codes have different lengths.



Shannon-Fano coding

- Step 1: compute, or estimate, the frequency f of every character c in our character set.

Character	Frequency
e	10
a	100
d	50
c	80
b	90

Frequency table

...continued

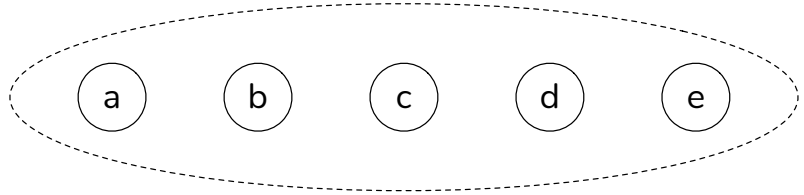
- Step 2: create a forest of trees containing a tree for each character (the tree is just a root node). The character table is sorted by frequency to help us visualise what's going on.

Character	Frequency
e	10
a	100
d	50
c	80
b	90

Frequency table

Character	Frequency
a	100
b	90
c	80
d	50
e	10

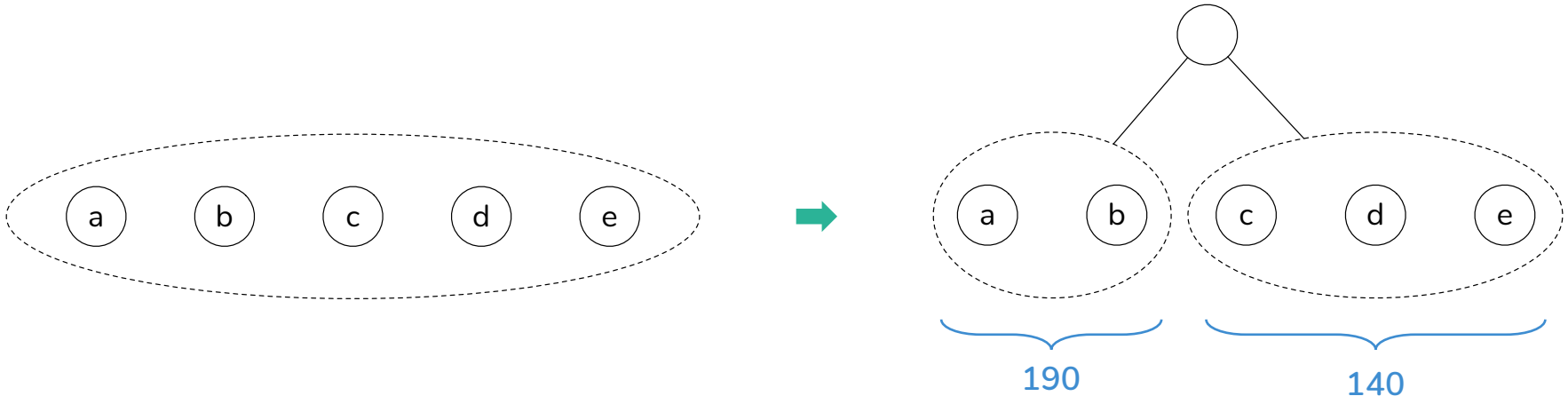
Sorted



Forest of trees

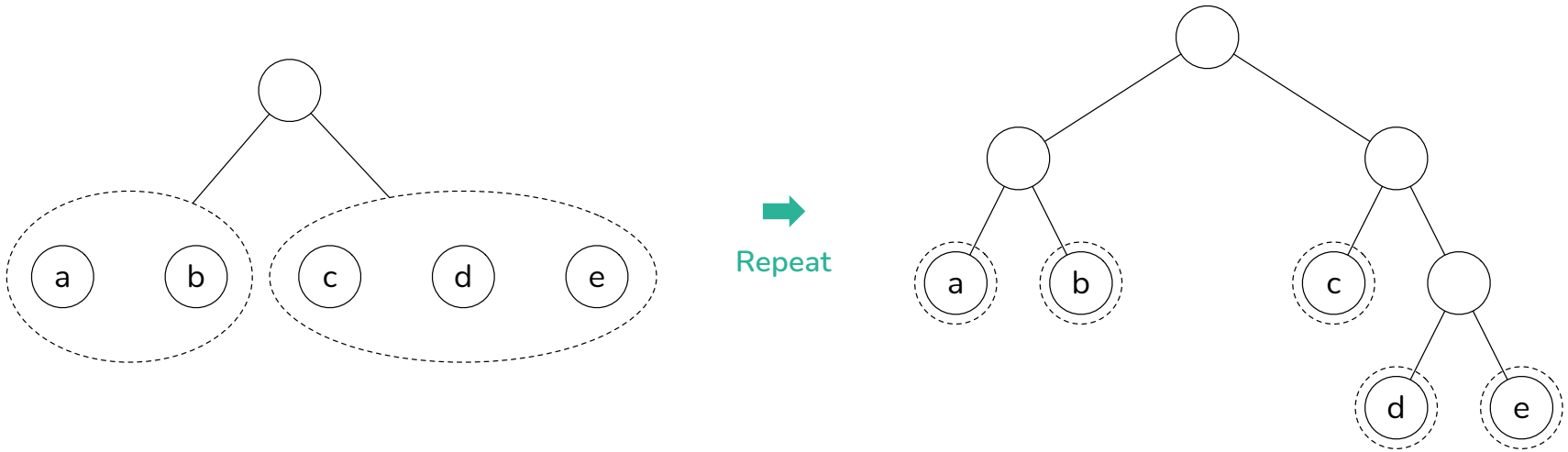
...continued

- Step 3: split the forest into two approximately equally-weighted parts. Merge the parts with a node and assign the left part a value of 0 and the right a value of 1.



...continued

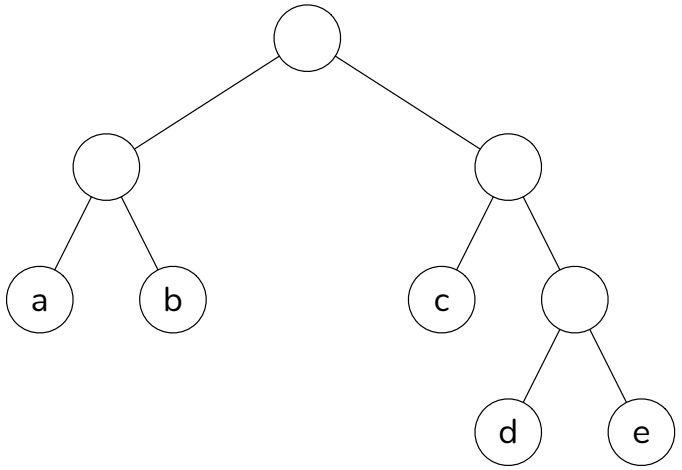
- Repeat this process recursively to every forest until each tree is alone in a forest (i.e., the symbol is a leaf in a tree).



The final prefix tree. Note that commonly occurring characters are shallower.

...continued

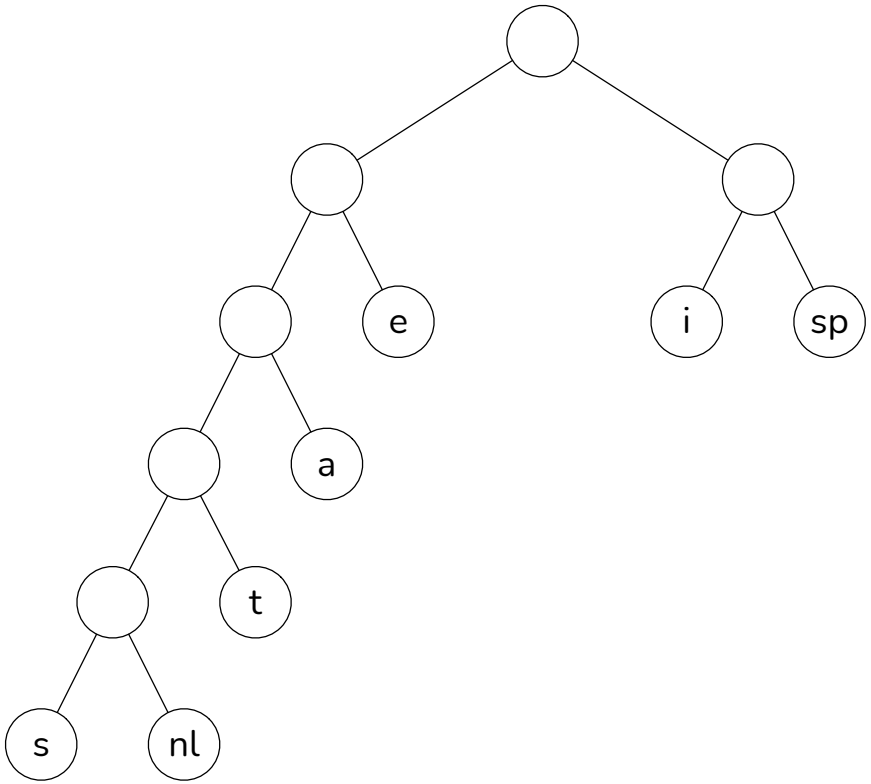
- Note that this is a basic method for creating a useful prefix tree and it does not guarantee that the prefix code is optimal.
- The final coding table we obtained:



Character	Frequency	Code
a	100	00
b	90	01
c	80	10
d	50	110
e	10	111

Commonly occurring characters have shorter codes.

The optimal prefix code tree



Character	Code	Frequency	Total bits
a	001	10	30
e	01	15	30
i	10	12	24
s	00000	3	15
t	0001	4	16
sp	11	13	26
nl	00001	1	5
		Total:	146

This is the file size

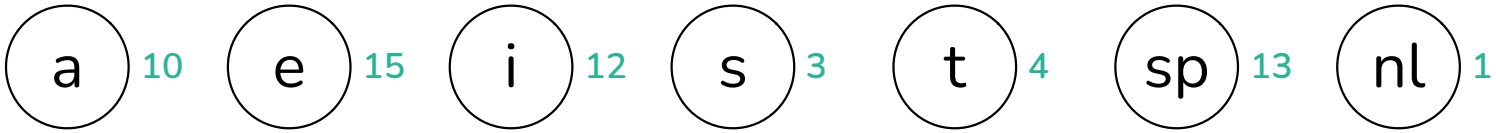
Example from Weiss

Huffman's algorithm

- An algorithm to construct an optimal prefix code tree is called Huffman's algorithm.
- It works by merging trees together (algorithm will be illustrated by example).
- Conventions:
  - A forest is a collection of trees.
  - C is the number of characters in the set.
  - The weight, w, of a tree is the sum of the frequencies of its leaves.
- In the beginning, there are C trees in the forest. Each tree is just a root/leaf node containing the character in question.
- The process iteratively merges (for $C - 1$ times) the trees until we get an optimal prefix tree.

Example

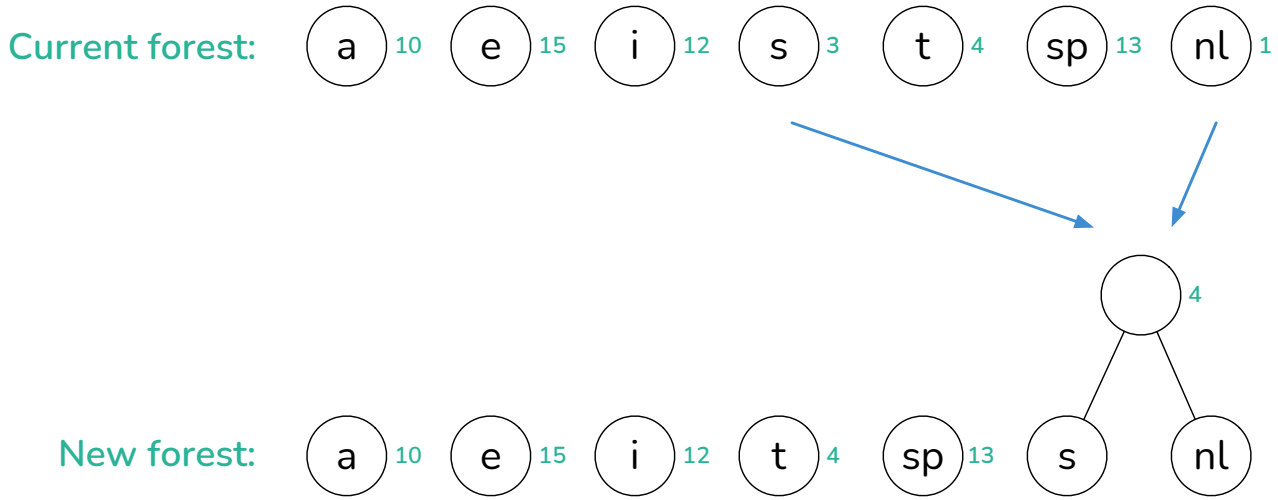
- Initial stage – forest containing C trees.
- Each tree has a single node with the character.
- The notes are annotated with the value w (weight, i.e., sum of frequencies).



Example from Weiss

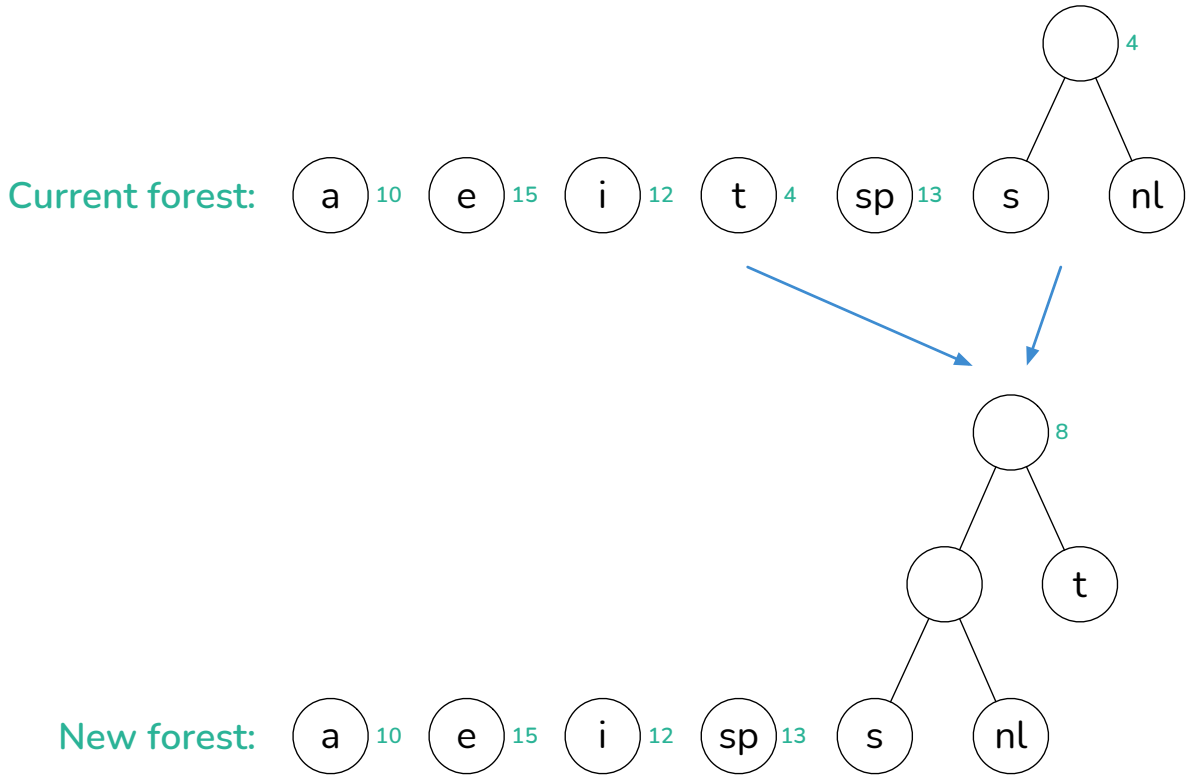
...continued

- Now we iteratively merge the two trees having the smallest weights and assign the weight of the new merged tree to the sum of the weights of the constituent trees.
- Note that ‘s’ and ‘nl’ have the smallest weights (3 and 1) so we join them with a parent node. The new tree has weight 3+1=4.



Example from Weiss

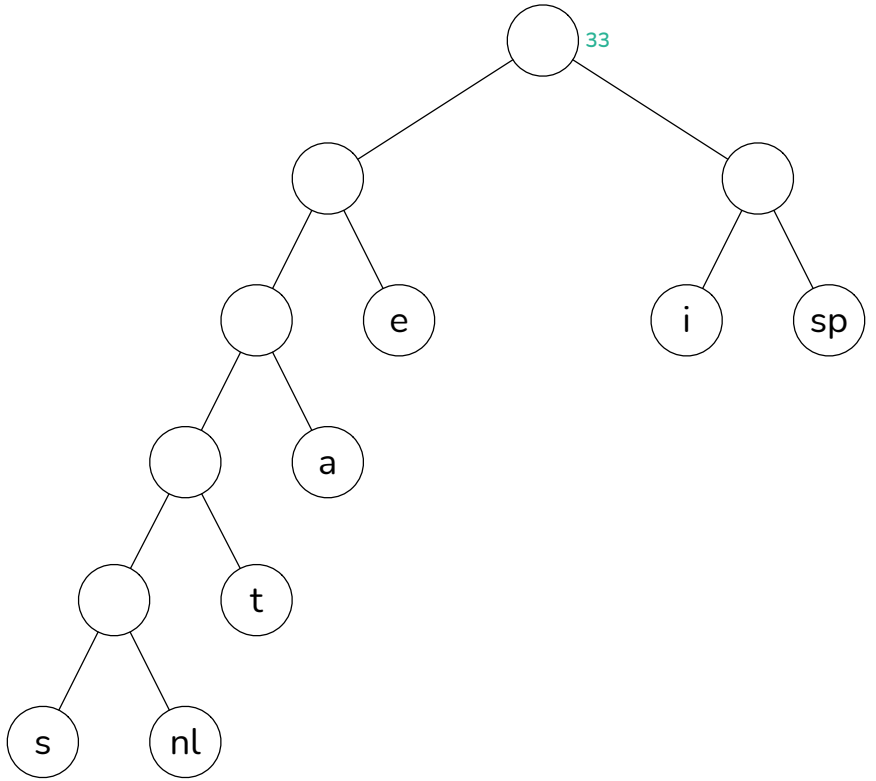
...continued



Example from Weiss

...continued

- The process is repeated until we end up with one tree. The final result is:



Example from Weiss

Huffman coding of images

- As practical example, consider Huffman encoding on 8-level grayscale images – normally we'd need 3 bits per pixel.

Source image
$128 \times 128 = 16384$

Gray level	Frequency
0	3441
1	4423
2	3932
3	1802
4	1311
5	819
6	492
7	164
Total:	16384 pixels

Gather statistics

Gray level	Huff. coding
0	11
1	01
2	10
3	001
4	0000
5	00010
6	000110
7	000111
Bits/pixel:	2.59

Huffman coding

The average bits per pixel is 2.59 vs. the 3 bits per pixel if not encoded.

Huffman coding using these statistics will give a 14% reduction in file size.

Example from http://academic.mu.edu/phys/matthysd/web226/L0423.htm

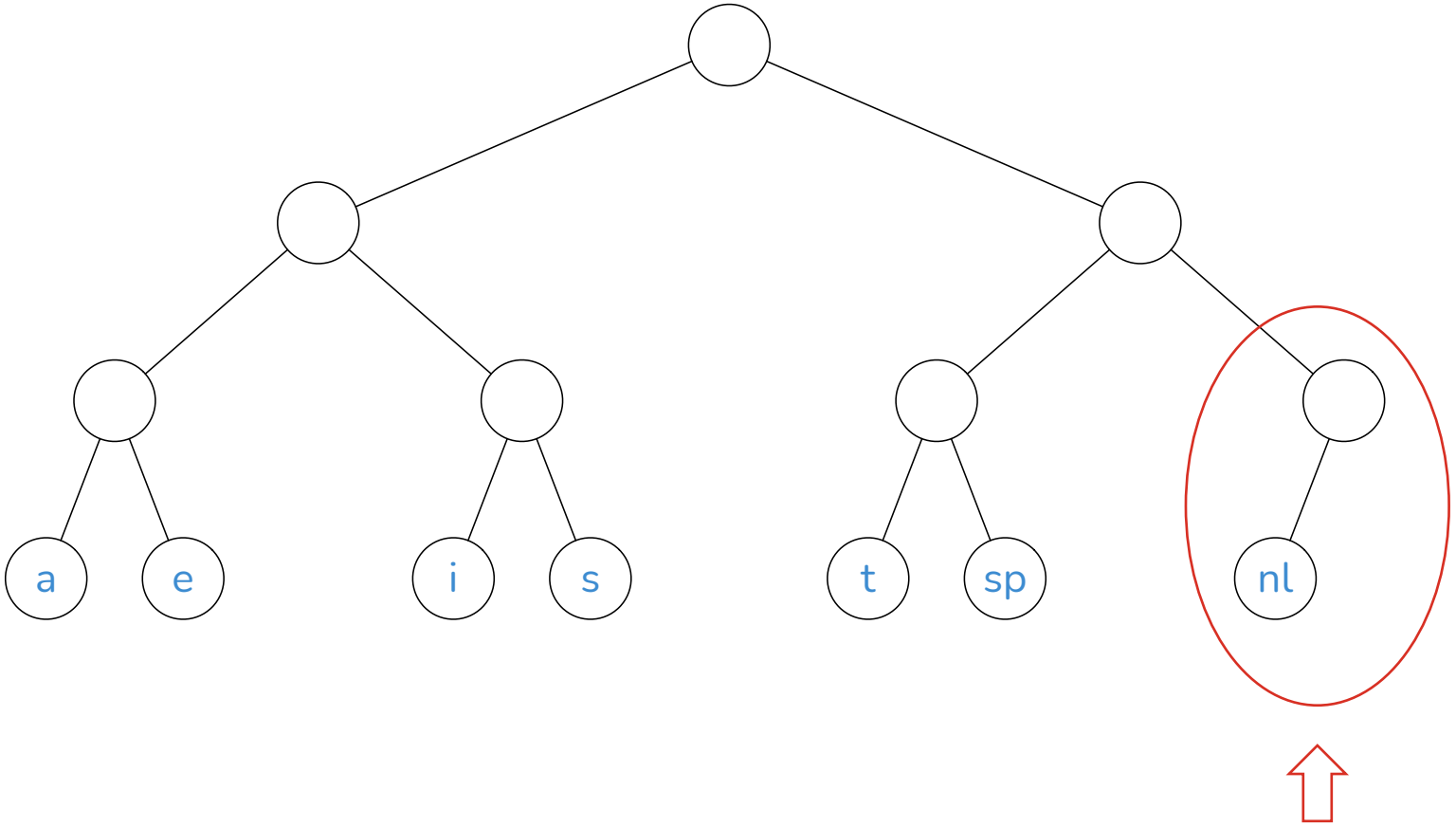
Proof of optimality

- Lemma 1:

An optimal coding tree is full – all internal nodes will have two child nodes.

Trivially shown by contradiction. Suppose that the tree T is optimal and has cost $cost(T)$. If there is an internal node N having only one child node M, we can simply remove N and replace it by the node M. This will reduce the depth of all the nodes in M's subtree, give a new subtree T' having a lower cost $cost(T')$.

…continued, lemma 1



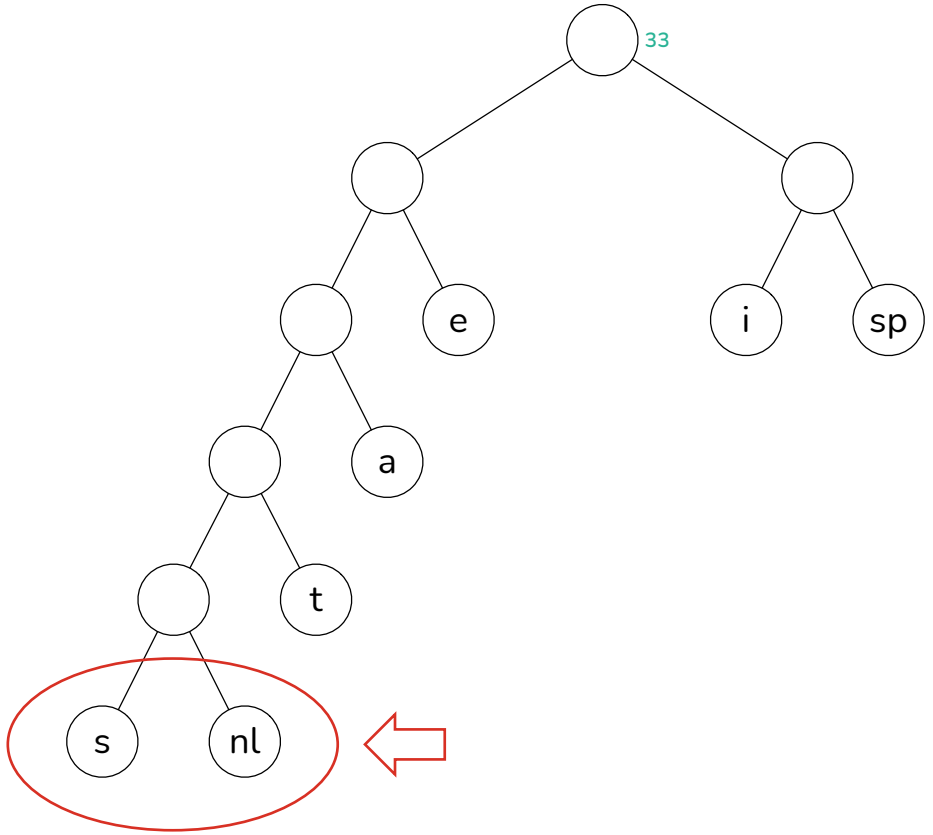
We've already seen this in this example before.

Another lemma

- Lemma 2:

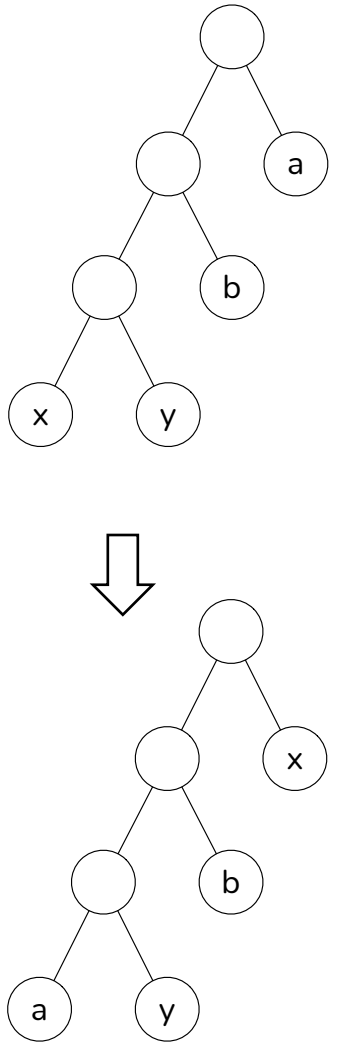
Consider two letters a and b having the smallest frequencies (least occurring).

There is an optimal code tree T in which these two letters are sibling leaves in the tree in the lowest level of the coding tree.



...continued, lemma 2

- Let T be an optimum prefix code tree having cost $cost(T)$.
- Let x and y be two siblings at the maximum depth of the tree. This case must exist due to lemma 1.
- Since, by our hypothesis, a and b have the two smallest frequencies, then $freq(a) \leq freq(x)$ and $freq(b) \leq freq(y)$.
- Since a and b are at the deepest level, $depth(a) \leq depth(x)$ and $depth(b) \leq depth(y)$.
- Case 1: swap the node for a with the node for x.
- This will give us a new tree T'.
- What is the cost of this new tree?

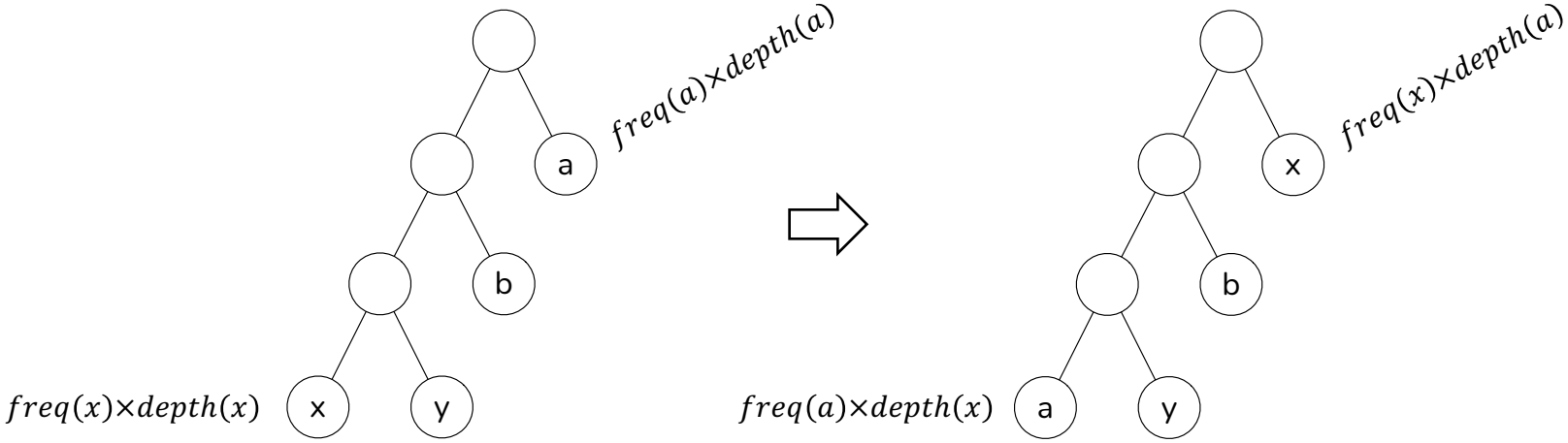


...continued, lemma 2

- The cost $cost(T')$ is given by:

$$\underbrace{cost(T')}_{} = \underbrace{cost(T)}_{} \underbrace{- freq(x)\times depth(x) - freq(a)\times depth(a)}_{} \underbrace{+ freq(x)\times depth(a) + freq(a)\times depth(x)}_{}$$

- i.e., the cost of the new tree is the cost of the previous one after removing the cost of x and the cost of a and then adding their costs back according to their new positions in the new tree.



...continued, lemma 2

- Rearranging...

$$cost(T') = cost(T) - freq(x)\times depth(x) - freq(a)\times depth(a) + freq(x)\times depth(a) + freq(a)\times depth(x)$$

- We get...

$$cost(T') = cost(T) - \big[\big(freq(a) - freq(x)\big)\times\big(depth(a) - depth(x)\big)\big]$$

Multiply/group binomials.

- So $cost(T') \leq cost(T)$
- But, by our hypothesis, $cost(T) \leq cost(T')$ since T is optimal.
- Therefore, $cost(T) = cost(T')$.
- This also holds for case 2: swap the node for b with the node for y where we get the same result.

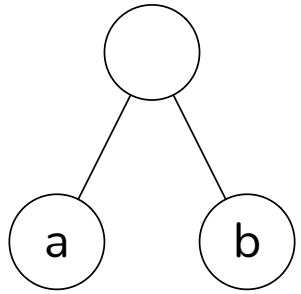
Theorem

Huffman coding has code efficiency (cost) which is lower than all prefix encodings of a given alphabet.

i.e., the algorithm is optimal.

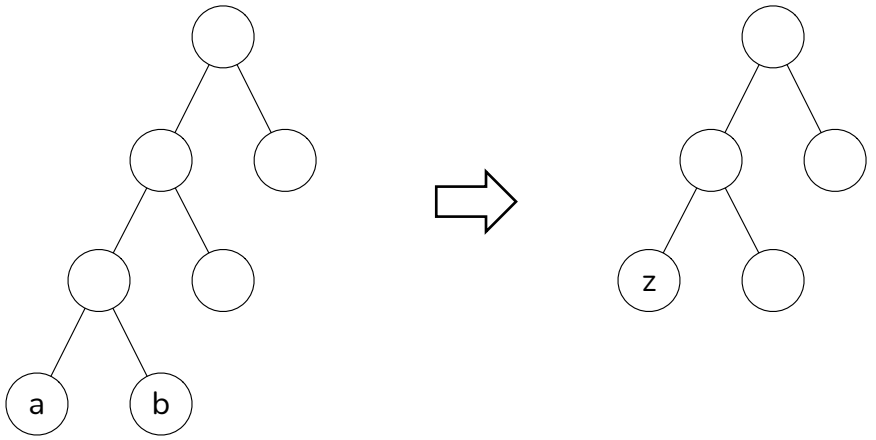
Proof

- We'll prove this by induction on the size n of the alphabet.
- Suppose that $n = 2$ (this is the smallest non-trivial alphabet size).
- In this case, the tree produced by the algorithm is obviously optimal; the tree will have one internal root note with two leaves for each symbol.



...continued

- Our inductive step will be to assume that the theorem holds true for all alphabets of size $n - 1$ and prove that it is also true for all alphabets of size n.
- Let A be an alphabet with n characters and let T be the optimal tree for A.
- T will have two leaves a and b which are siblings having minimum frequencies due to lemmas 1 and 2.
- Consider the tree T' which is a prefix code for the alphabet A' corresponding to the alphabet A after removing characters a and b then adding a new character z at their parent node in T.



...continued

- The frequency of the new character z will be $freq(a) + freq(b)$; remember that this is how Huffman 'weighs' internal nodes.
- So far, we have...
  - $depth(a) = depth(b) = depth(z) + 1$
  - Equivalently, $depth(z) = depth(a) - 1 = depth(b) - 1$
  - And $freq(z) = freq(a) + freq(b)$
  - And...

$$cost(T') = cost(T) - freq(a) \times depth(a) - freq(b) \times depth(b) + freq(z) \times depth(z)$$

...continued

- Let's clean up the previous expression for $cost(T')$ by removing all references to z in it.
- Using what we know:
  - $depth(z) = depth(a) - 1 = depth(b) - 1$
  - And $freq(z) = freq(a) + freq(b)$

$$cost(T') = cost(T) - freq(a) \times depth(a) - freq(b) \times depth(b) + freq(z) \times depth(z)$$

Becomes...

$$cost(T') = cost(T) - \big(freq(a) + freq(b)\big)$$

So...

$$cost(T) = cost(T') + \big(freq(a) + freq(b)\big)$$ ⇦ A constant!

...continued

- Let H' be the Huffman tree generated for the alphabet A'.
- Remember that H' is the tree prior to the merge that the algorithm does.
- So, by our inductive assumption, it is optimal.
- So, $cost(H') \leq cost(T')$ 1
- When we add the two symbols a and b under z, we get the Huffman tree H for the whole alphabet A.
- Remember what the algorithm is doing; when two symbols (trees in the forest) a and b are added to the Huffman tree, they are replaced by a new 'meta' symbol z (a new tree in the forest). Basically, whenever a merge is made, the size of the alphabet under consideration (the number of trees in the forest) is decreasing by one (1); minus 2 for each of a and b, and plus 1 for z.

...continued

- Using the same computation we made to calculate $cost(T)$, we get:

$$cost(H) = cost(H') + freq(a) + freq(b) \quad (2)$$

- So, from (1) and (2) ...

$$cost(H) = cost(H') + freq(a) + freq(b) \leq cost(T') + freq(a) + freq(b)$$

- We previously worked out that...

$$cost(T) = cost(T') + \big(freq(a) + freq(b)\big)$$

- So...

$$cost(H) \leq cost(T) \quad (3)$$

...continued

- Since T is optimal (by our initial hypothesis), we know that...

$$cost(T) \leq cost(H) \quad (4)$$

- Therefore, from (3) and (4) ...

$$cost(T) = cost(H)$$

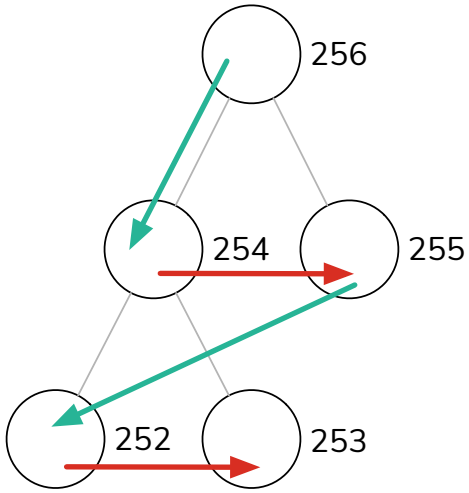
- Done.

Adaptive Huffman coding

- Does not require a priori knowledge of the character distribution (no need for a frequency table).
  - This is usually the case when data is transmitted and compressed in real time.
  - The prefix tree is rearranged as data is received.
  - Note: this makes the algorithm sensitive to transmission errors.
  - Also, useful when the computation of the statistical data may be very intensive.
- We will be describing Vitter's algorithm.

...continued

- Code is represented by a tree where:
  - Every node has a weight.
  - Every node has a unique number.
  - The unique numbers decrease downwards and increase from left-to-right.

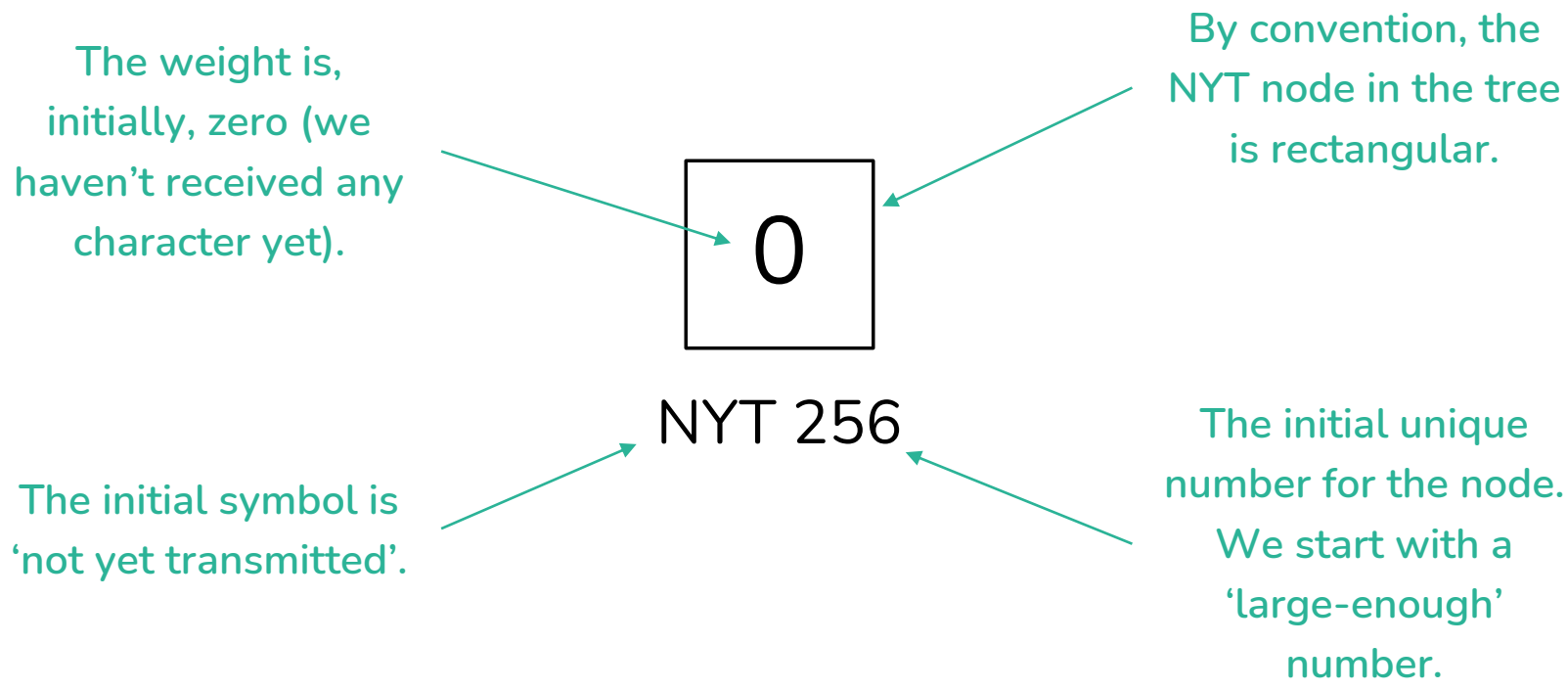


...continued

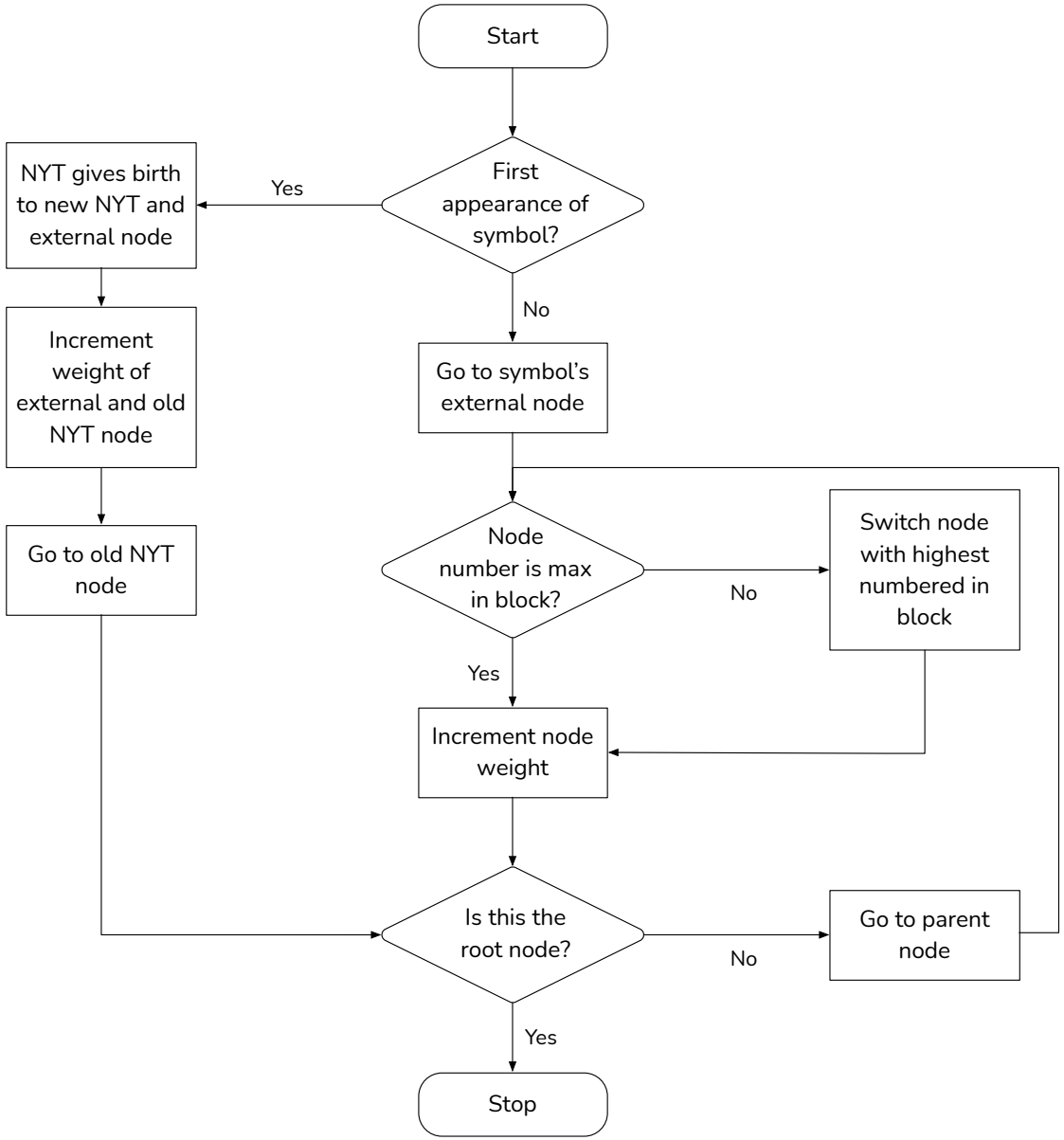
- Weights must satisfy the sibling property:
  - A node with a higher weight will have a higher number, and
  - A parent node will always have a higher node number than its children.
- The weight of a symbol is the number of times that symbol appeared so far.
- A set of nodes with the same weight is called a block.
- NYT represents the 'not yet transmitted' symbol.

...continued

- The starting point is simply a node containing the NYT symbol (in other words, this is starting prefix tree):



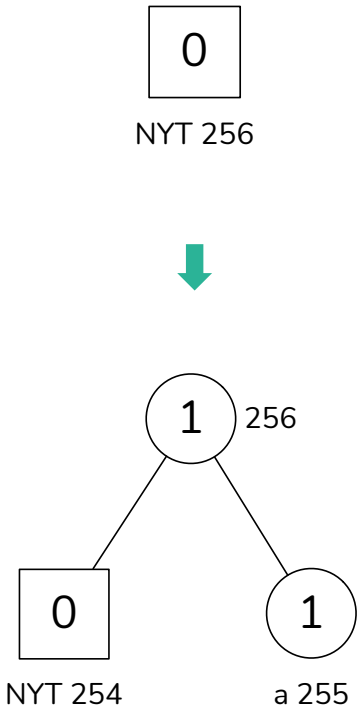
The algorithm

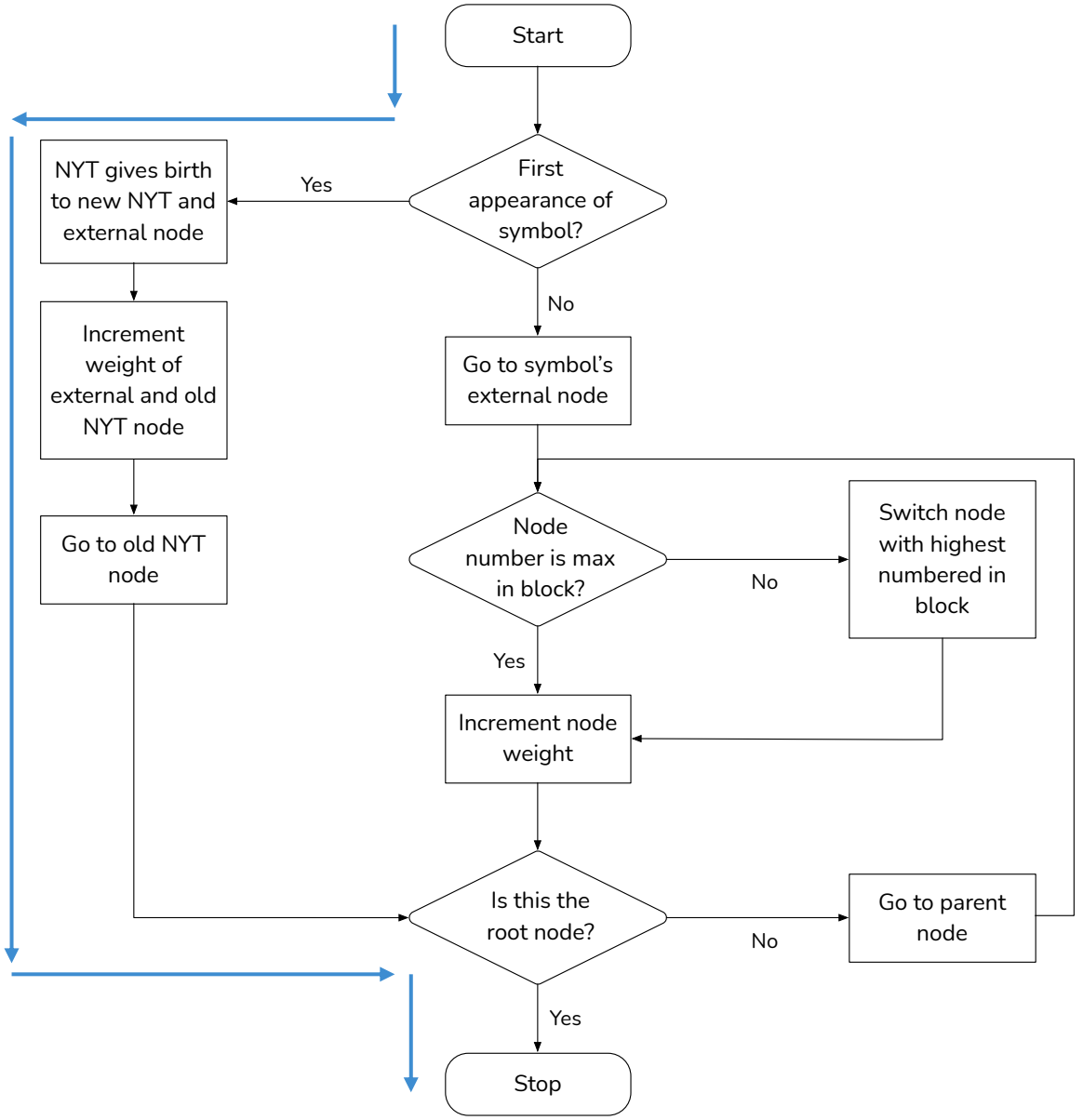


Flowchart from "Introduction to Data Compression" by by Sayood Khalid

Example

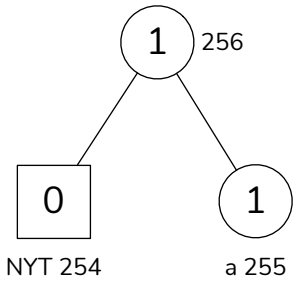
We received the symbol ‘a’

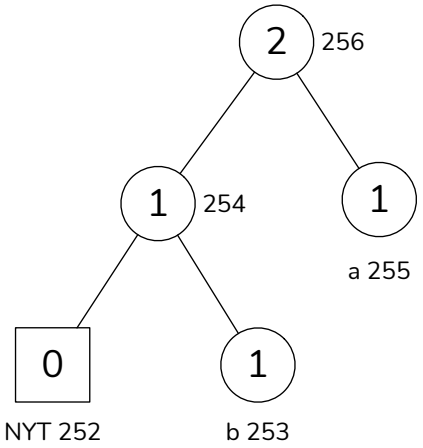


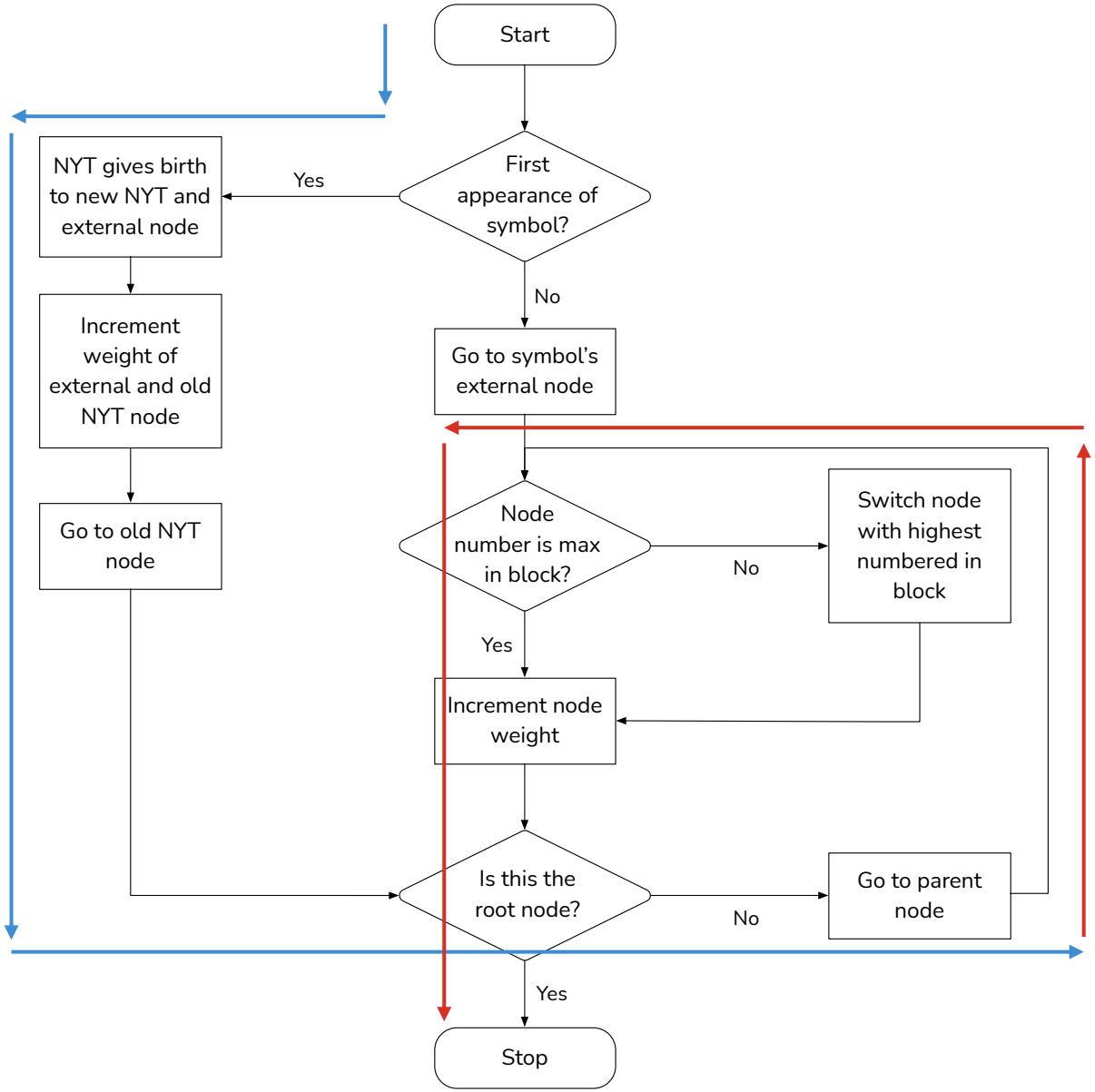


Example

We received the symbol ‘b’

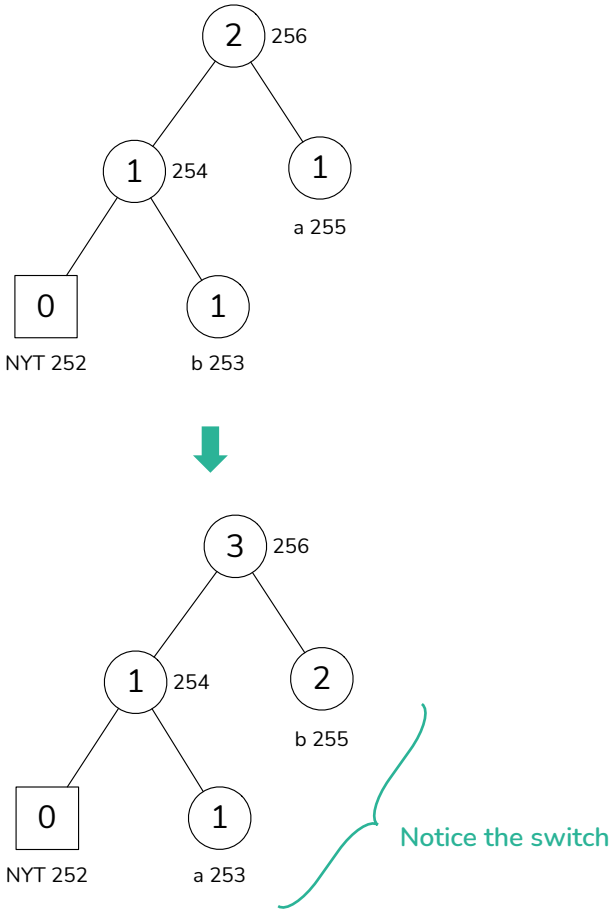


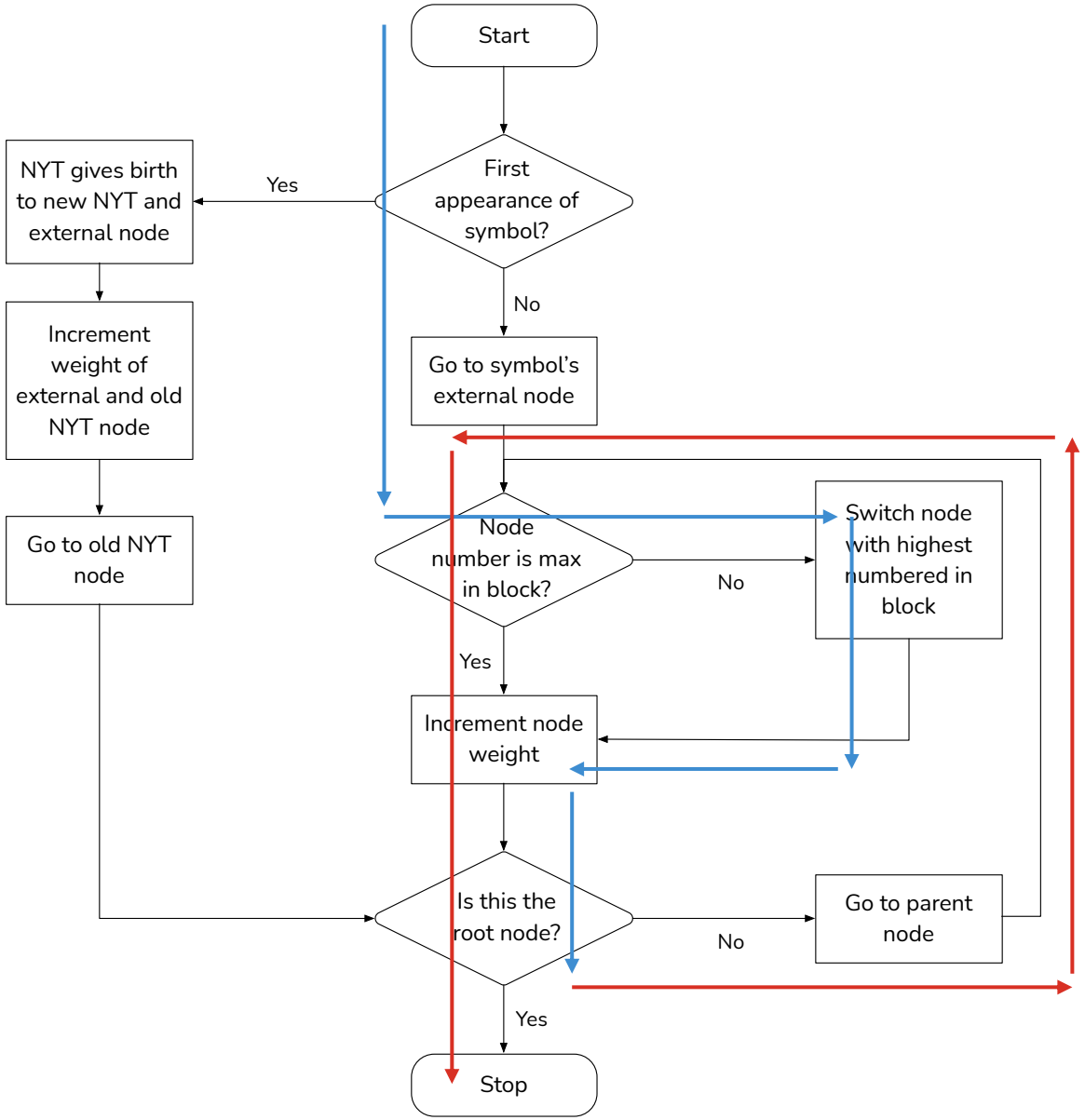




Example

We received another symbol ‘b’





Lempel-Ziv-Welch

- Lossless Data Compression Algorithm.
- It allows ‘longer’ sequence compression compared to Huffman coding which shortens (or lengthens) codes for sequences of only one character.
- In its most basic form:
  - Encode 8-bit data as 12-bit codes.
  - Codes 0 to 255 represent single character sequences and are automatically placed in the dictionary.
  - Codes 256 to 4095 make up a dictionary that is created for longer sequences encountered in the data.

Notice repeated sequences in this text...

```
<a href="/store/product/subscriptions/" class="header-highlight-link">Subscribe</a>
<div class="dropdown" id="header-search">
    <a href="/search/" class="dropdown-toggle search-toggle" aria-label="Search" aria-expanded="false">
        <span class="icon icon-search-mag-glass"></span>
    </a>
    <div class="dropdown-content">
        <form action="/search/" method="GET" id="search_form">
            <input type="hidden" name="ie" value="UTF-8">
            <input type="text" name="q" id="hdr_search_input" value="" aria-label="Search..." placeholder="Search...">
        </form>
        <a class="nav-search-close">Close</a>
    </div>
</div>
<div class="dropdown dropdown-mega" id="header-burger">
    <a href="#site-menu" class="dropdown-toggle" aria-label="Menu" aria-expanded="false">
        <span></span>
    </a>
    <div id="site-menu" class="dropdown-content">
        <section class="burger-navigate">
            <h3>
                <span class="icon icon-half-target"></span>

                    Navigate

            </h3>
            <ul>
                <li>
                    <a class="nav-link store" href="/store/">Store</a>
                </li>
                <li>
                    <a class="nav-link subscribe" href="/store/product/subscriptions/">Subscribe</a>
                </li>
                <li>
                    <a class="nav-link videos" href="http://video.arstechnica.com/">Videos</a>
                </li>
                <li>
                    <a class="nav-link section-features" href="/features/">Features</a>
                </li>
                <li>
                    <a class="nav-link section-reviews" href="/reviews/">Reviews</a>
                </li>
```

Basic method

- Read bytes and append until we meet a sequence that is not in the dictionary.
- By construction, a 1-byte sequence is always in the dictionary.
- When we append a byte that will create a sequence that is not in the dictionary we stop.
- Obviously, the sequence we read apart from the last read byte must be in the dictionary. For this, we emit the corresponding code.
- The sequence we read including the last read byte is added to the dictionary and associated with a new code.
- This last byte is then the starting point of the new scan.

A simple example

- Consider the following plain text to be compressed:
- Note the # that signals the end of transmission.
  - We call this the stop-code, and we give it the code 0.

TOBEORNOTTOBEORTOBEORNOT#

Example taken from Wikipedia

...continued

- There are 26 characters in the alphabet we are considering (A-Z), plus the stop-code.
- For these 27 combinations, we need at least 5 bits (2^5=32)
- The dictionary is initialised as follows:

Initial dictionary

Symbol	Code	Decimal
#	00000	0
A	00001	1
B	00010	
C	00011	
.		
.		
.		
Z	11010	26

...continued

- The 5 bits allow for 32 entries in our dictionary.
- If we meet/need the 33rd entry we have switch the dictionary to 6 bits.
- To encode...

Buffer input characters in a sequence ω until ω + [next character] is not in the dictionary. Emit the code for ω and add ω + [next character] to the dictionary. Start buffering again with that next character.

TOBEORNOTTOBEORTOBEORNOT#

Current Sequence	Next Char	Output Code	Output Bits	Extended Dictionary
NULL	T			
T	O	20 =	10100	27: TO
O	B	15 =	01111	28: OB
B	E	2 =	00010	29: BE
E	O	5 =	00101	30: EO
O	R	15 =	01111	31: OR
R	N	18 =	10010	32: RN
N	O	14 =	001110	33: NO
O	T	15 =	001111	34: OT

The first free slot in the dictionary.

We need to switch to a 6-bit dictionary.

Example from Wikipedia

TOBEORNOTTTOBEORTOBEORNOT#

Current Sequence	Next Char	Output Code	Output Bits	Extended Dictionary
...continued...				
T	T	20 =	010100	35: TT
TO	B	27 =	011011	36: TOB
BE	O	29 =	011101	37: BEO
OR	T	31 =	011111	38: ORT
TOB	E	36 =	100100	39: TOBE
EO	R	30 =	011110	40: EOR
RN	O	32 =	100000	41: RNO
OT	#	34 =	100010	
		0 =	000000	

Stop-code. Send current sequence and stop-code.

Results

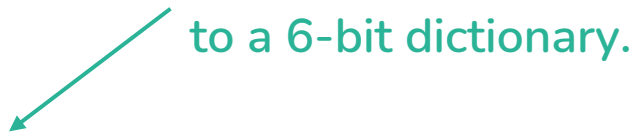
- Sending the transmission (25 characters) using 5 bits = 125 bits required.
- However, sending the compressed transmission requires 6×5 bits + 11×6 bits = 96 bits.
- We save almost 22%.

Decompression

- The decompressor only needs the initial dictionary used – the other entries can be constructed by the decompressor as well.
- In other words, the compressor and the decompressor only need to agree on the character set and how the initial dictionary is created.

Example

Input Bits	Output Code	Sequence	New Dictionary Entry Full	Conjecture
10100	= 20	T		27: T?
01111	= 15	O	27: TO	28: O?
00010	= 2	B	28: OB	29: B?
00101	= 5	E	29: BE	30: E?
01111	= 15	O	30: EO	31: O?
10010	= 18	R	31: OR	32: R?
001110	= 14	N	32: RN	33: N?
001111	= 15	O	33: NO	34: O?
010100	= 20	T	34: OT	35: T?



...continued

Input Bits	Output Code	Sequence	New Dictionary Entry Full	Conjecture
...continued...				
011011	= 27	TO	35: TT	36: TO?
011101	= 29	BE	36: TOB	37: BE?
011111	= 31	OR	37: BEO	38: OR?
100100	= 36	TOB	38: ORT	39: TOB?
011110	= 30	EO	39: TOBE	40: EO?
100000	= 32	RN	40: EOR	41: RN?
100010	= 34	OT	41: RNO	42: OT?
000000	= 0	#		

Further reading

- These notes should be supplemented by:
  - Introduction to Algorithms (Clifford Stein, Thomas H Cormen, Ronald L Rivest, Charles E Leiserson – MIT Press)
  - Data Structures and Algorithm Analysis in Java (Weiss)

Longest Common Subsequence

Kristian Guillaumier

Substrings and subsequences

- Substring:
  - Contiguous sequence (in order) of characters or symbols in a sequence/string.
- Subsequence:
  - Non-contiguous sequence (also in order) of characters or symbols in a sequence/string.
  - Equivalently, a subsequence is obtained by deleting zero or more symbols/characters in a sequence.

Some examples

- Substrings of `HelloWorld`:
  - `Hel` `HelloWorld`
  - `loWo` `HelloWorld`
  - `Hello` `HelloWorld`
  - `oWorld` `HelloWorld`
- Subsequences of `HelloWorld`:
  - `Hll` `HelloWorld`
  - `HWrd` `HelloWorld`
  - `Hello` `HelloWorld` (every substring is also a subsequence, not vice-versa)
  - ϵ `HelloWorld` (the empty string is a subsequence and a substring)
  - `HelloWorld` `HelloWorld` (the string itself is a subsequence and a substring of itself)

The longest common subsequence

- We have two strings or sequences X and Y of varying lengths.
- `X = X[1..n]` and `Y = Y[1..m]` where `n` and `m` are the lengths of the strings.
- We must find a longest subsequence in X and Y.

- `X: 3452345`
- `Y: 4541534`

In this example, the strings just happen to have the same length.

- One possible LCS(X,Y) is:
  - `X: 3452345`
  - `Y: 4541534`
  - LCS(X,Y) = 4545 and another one is 4534.

The LCS is not unique – its length is.

Dynamic programming

- Like divide and conquer, dynamic programming is a mathematical technique for ‘breaking down’ a problem into smaller parts (you should be familiar with the method).
- It can be used to solve optimisation problems.
- Here will use the Longest Common Subsequence (LCS) to illustrate DP.
- For example, finding common sequences in DNA sequences or online file backups.

A brute force method

- Create every subsequence in X and check if it is a subsequence of Y.
- Keep track of longest found so far.
- So, for the string `X:3452345` generate all the subsequences:

$$\epsilon,\ 3,\ 4,\ 5,\ \ldots,\ 34,\ 35,\ 32,\ \ldots,\ 345\ldots$$

- ...and check each one in the other string Y.

...continued

- How many steps does it take to check if a string is a subsequence of another?
- E.g., to check if 544 is a subsequence of 4541534.
- At worse $|4541534| = 7$ steps.
- So, to check if a string is a subsequence in $Y[1..m]$, we need O(m) time.

…continued

- How many checks need to be made (how many subsequences can be generated from X)?
- Answer is 2^n.
- To show this, let's use a bit string of length n (as long as the string X). Every bit set to 1 indicates a subsequence character selection in X.
- For example, …

...continued

0000000 = 3452345 = ϵ

0000001 = 3452345

0000010 = 3452345

...

0000011 = 3452345

...

1111111 = 3452345

There are 2^n of these. So, the worst-case running time is $O(m2^n)$.

An idea for improvement

- Step 1: Find the length of the LCS(X, Y).
- Note that we are finding the length only not the actual LCS.
- The length of LCS(X, Y) is unique.
- Step 2: Find the actual LCS(X, Y) knowing this length – this greatly simplifies the problem.
- Let's consider the prefixes of X and Y.
- We'll define:

```
c[i,j] = |LCS(X[1..i], Y[1..j])|
```

- For example, c[2, 3] is the length of the LCS between the first 2 characters of X and of the first 3 characters of Y.

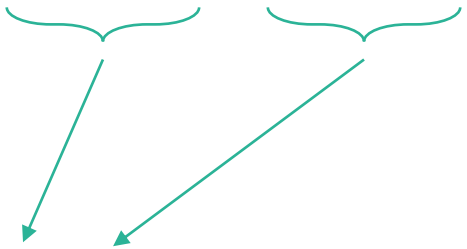
...continued

- Note that using:

```
c[i,j] = |LCS(X[1..i], Y[1..j])|
```

- The length of the entire LCS would be given when `i=n` and `j=m`:

```
c[n,m] = |LCS(X[1..n], Y[1..m])|
```



```
c[n,m] = |LCS(X, Y)|
```

Theorem

`c[i,j]` is

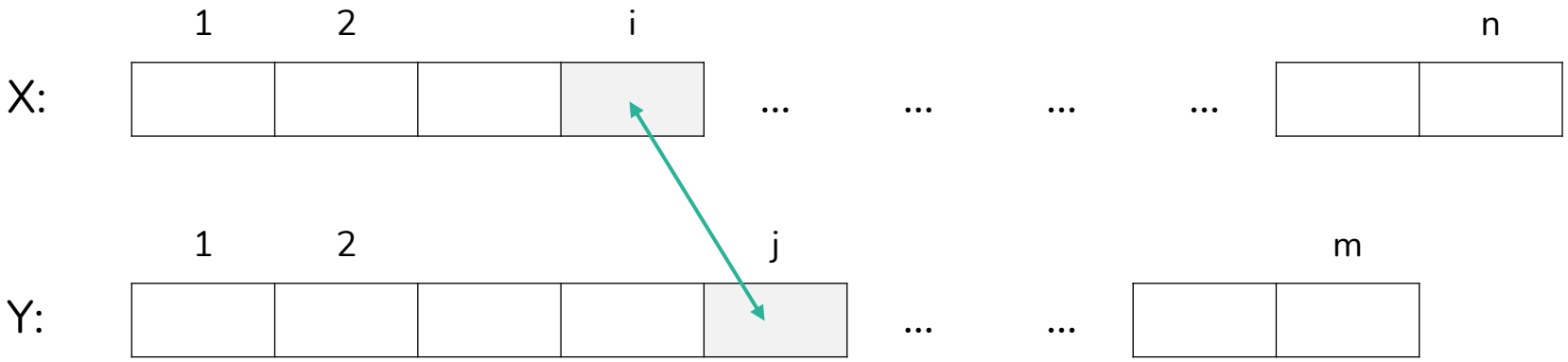
= `c[i-1,j-1] + 1` if `x[i] = y[j]`

= `max(c[i, j-1), c[i-1, j])` otherwise

Remember that we did something like this in edit string distance.

		0	1	2	3	4	5
		ϵ	w	h	a	l	e
0	ϵ						
1	h						
2	e						
3	l						
4	l						
5	o						

Proof for case `X[i]=Y[j]`



When these characters `x[i]` and `y[j]` are the same.

...continued

- Theorem for case 1:
  - `c[i,j] = c[i-1, j-1]+1` for `X[i]` is equal to `Y[j]`.

- Let:
  - `k = c[i,j]` be the length of the LCS in the prefix of X and prefix of Y
  - `Z[1..k] = LCS(X[1..i], Y[1..j])`

k is the length of the LCS in the prefix of X and prefix of Y

Z is the actual LCS in the prefix of X and prefix of Y

Example

X = 3452345

Y = 43141534

i = 2

j = 4

Remember that X[i] must be the same as Y[j] in this case: X[2] = Y[4], “4” = “4”.

c[2,4]

= |LCS(X[1..2], Y[1..4])|

= |LCS(“34”, “4314”)|

= |”34”|

= 2

So, the length of the LCS up to i in X and j in Y is 2,

and the actual LCS Z is “34”

...continued

```
X = 3452345
Y = 43141534
i = 2
j = 4

c[2,4] = |LCS(X[1..2], Y[1..4])|
       = |LCS("34", "4314")|
       = |"34"|
       = 2
```

- Remember we let:
  - `c[i,j] = k`
  - `Z[1..k] = LCS(X[1..i], Y[1..j])`
- We observe that:
  - `Z[k] = X[i] = Y[j]`
  - "4" = "4" = "4"

  } k^{th} character in the LCS Z, i^{th} character in X, and j^{th} character in Y must be the same!
- Thus:
  - `Z[1..k-1]` is a common subsequence (CS) of `X[1..i-1]` and `Y[1..j-1]`.
  - In our example, "3" is a CS of "3" and "431" which is true.
- We will now show that not only `Z[1..k-1]` is a CS but also an LCS.

...continued

- Claim:
  - Let Z[1..k] be a the LCS having length k of the strings `X[1..i]` and `Y[1..j]`.
  - `Z[1..k-1]` is not only a CS in `X[1..i-1]` and `Y[1..j-1]` but also an LCS.
- Suppose a sequence `w` exists which is a longer CS in `X[1..i-1]` and `Y[1..j-1]`:
  - i.e., `|w| > k-1`
- Argument:
  - Concatenate that sequence `w` with `Z[k]` giving `w+Z[k]`.
  - If `|w| > k-1`
  - Then `|w + Z[k]|` would be greater than `k` – contradicts our initial hypothesis.

Remember that `Z[k]` is the last character in the LCS of `X[1..i]` and `Y[1..j]`.

...continued

- Since we showed that `Z[1..k-1]` is an LCS of `X[1..i-1]` and `Y[1..j-1]`.
- Then `k-1` is the length of the LCS of `X[1..i-1]` and `Y[1..j-1]`.
- Thus...
  - `c[i-1, j-1] = k-1`
  - And `k` is `c[i,j]`, so...
  - `c[i,j] = c[i-1, j-1] + 1`.

- Proven first case.

...continued

- We have shown that if `Z = LCS(X, Y)`
- Then any prefix of Z = LCS (in a prefix of X, a prefix of Y).

This is what we would call the "optimal substructure" property in DP. We'll talk about this later.

...continued

- `Case 2:`
  - `c[i,j]= max(c[i, j-1), c[i-1, j]) for X[i] != Y[j]`

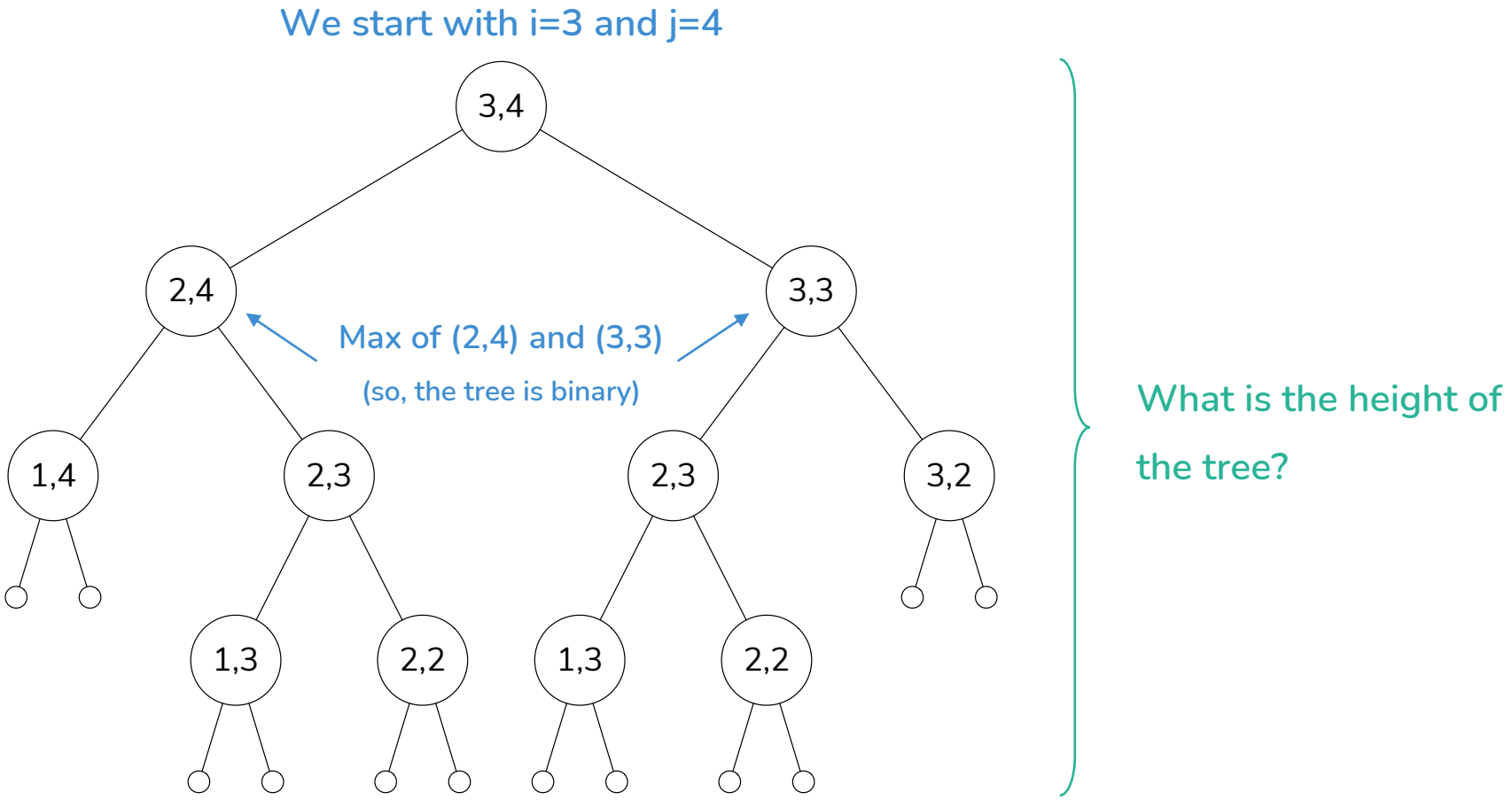
- After showing case 1, this is clearly true and will be left as an exercise for the student – work out an example.

The algorithm

```
c(i,j) {
  if (X[i] == Y[j])
       return c(i-1, j-1) + 1
  else
       return max( c(i-1,j) , c(i, j-1) )
}
```

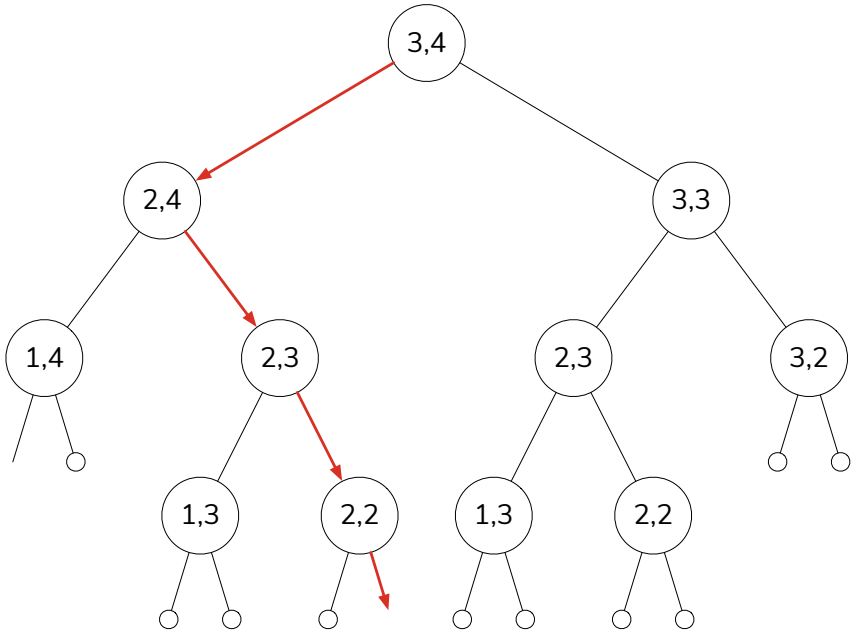
- Worst case is when `X[i] != Y[j]` for all `i, j` (the LCS will be the empty string).
- We must compute 2 sub problems to get the max.

Recursion tree analysis



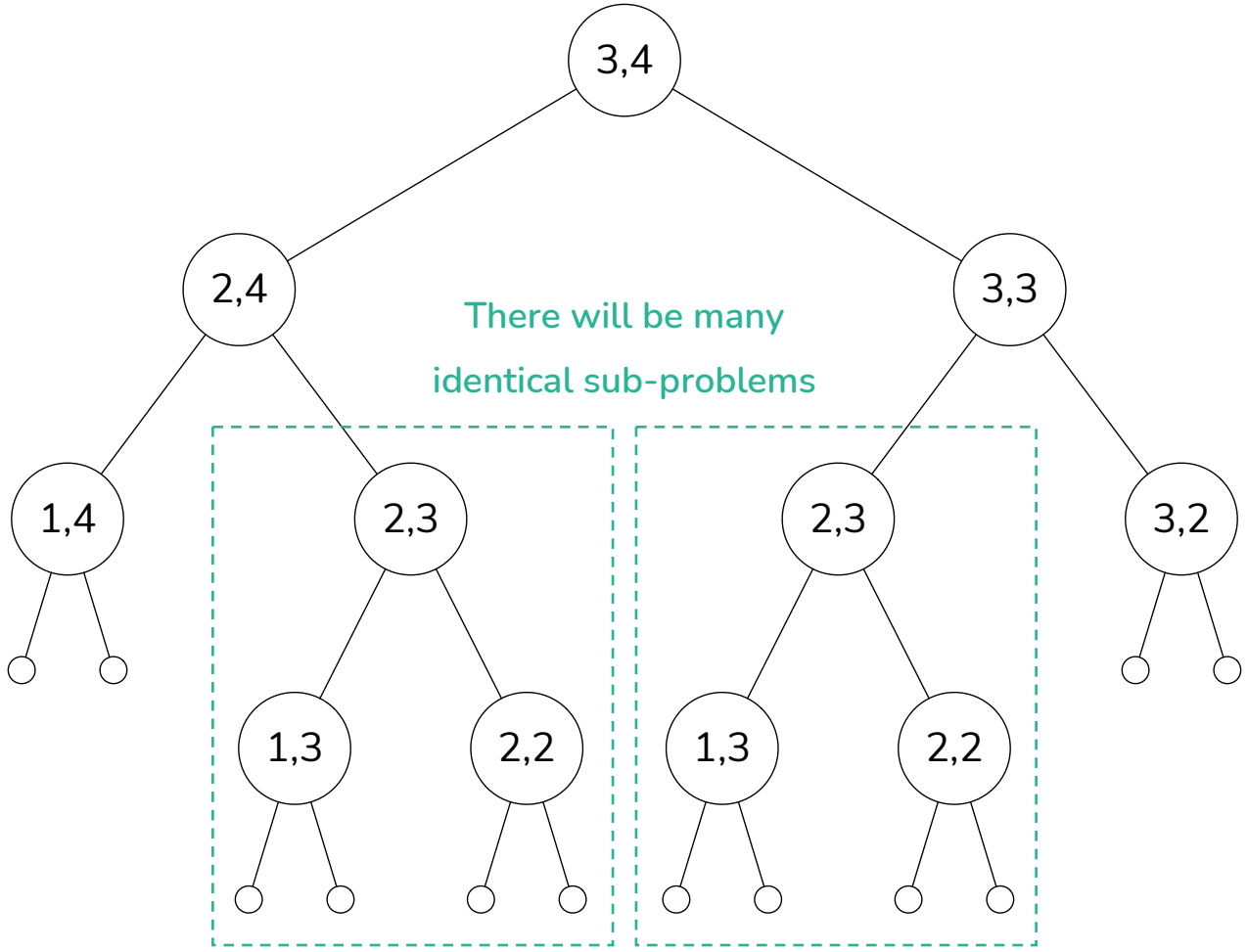
Example adapted from Eric Demaine

...continued



- Note that the height is n+m not max(n,m). Observe the repetition (rather than decrementing) of “2” down the red line.
- This gives us an exponential number of nodes $2^{(n+m)}$ to compute – not good... but...

...continued


3,4
2,4
3,3
There will be many
identical sub-problems
1,4
2,3
2,3
3,2
1,3
2,2
1,3
2,2

Memoization

- We will keep a memo. The idea is like using a lookup table.

Optimise by avoiding repeatedly calculating the results for previously processed inputs.

Example: Computing factorial without memoization

```
Factorial(n) {
  if (n==1)
       return 1
  else
       return n*Factorial(n-1)
}
```

Example: Computing factorial with memoization

```
Factorial(n) {
  if (Lookup(n) exists)
       return Lookup(n)
  else if (n==1)
       return 1
  else
       result       = n*Factorial(n-1)
       Lookup(n)    = result // put in lookup.
       return result
}
```

So, computing 5! requires 5 recursive calls, but subsequently computing 7! only requires 2.

Recursion tree analysis

- Back to the recursion tree, further analysis shows that there although there are $2^{(n+m)}$ nodes to evaluate, there are $n \times m$ distinct subtrees (sub problems) – one for every combination of n and m.
- We use a memoization algorithm to compute. This gives:
  - Time O(nm).
  - Space O(nm) ← required to store the lookup table.

Working LCS as a DP

	ε	A	B	C	B	D	A	B
ε	0	0	0	0	0	0	0	0
B	0							
D	0							
C	0							
A	0							
B	0							
A	0							

The zeroes are our base case.

The rest will be filled in using the results of our theorem.

The length of the LCS will be here.

Example adapted from Eric Demaine

Completed example

	ε	A	B	C	B	D	A	B
ε	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

```
if x[i]=y[i] c[i-1,j-1] + 1
else max(c[i, j-1), c[i-1, j])
```

There are 4 characters in the LCS.

Annotating for traceback

The arrows indicate when we chose to pick the diagonal +1 because the two characters at the two indices were the same.

	ε	A	B	C	B	D	A	B
ε	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

Finding the LCS by tracing backwards

We can move left or up when we have the same value and diagonally if there is an ‘arrow’.

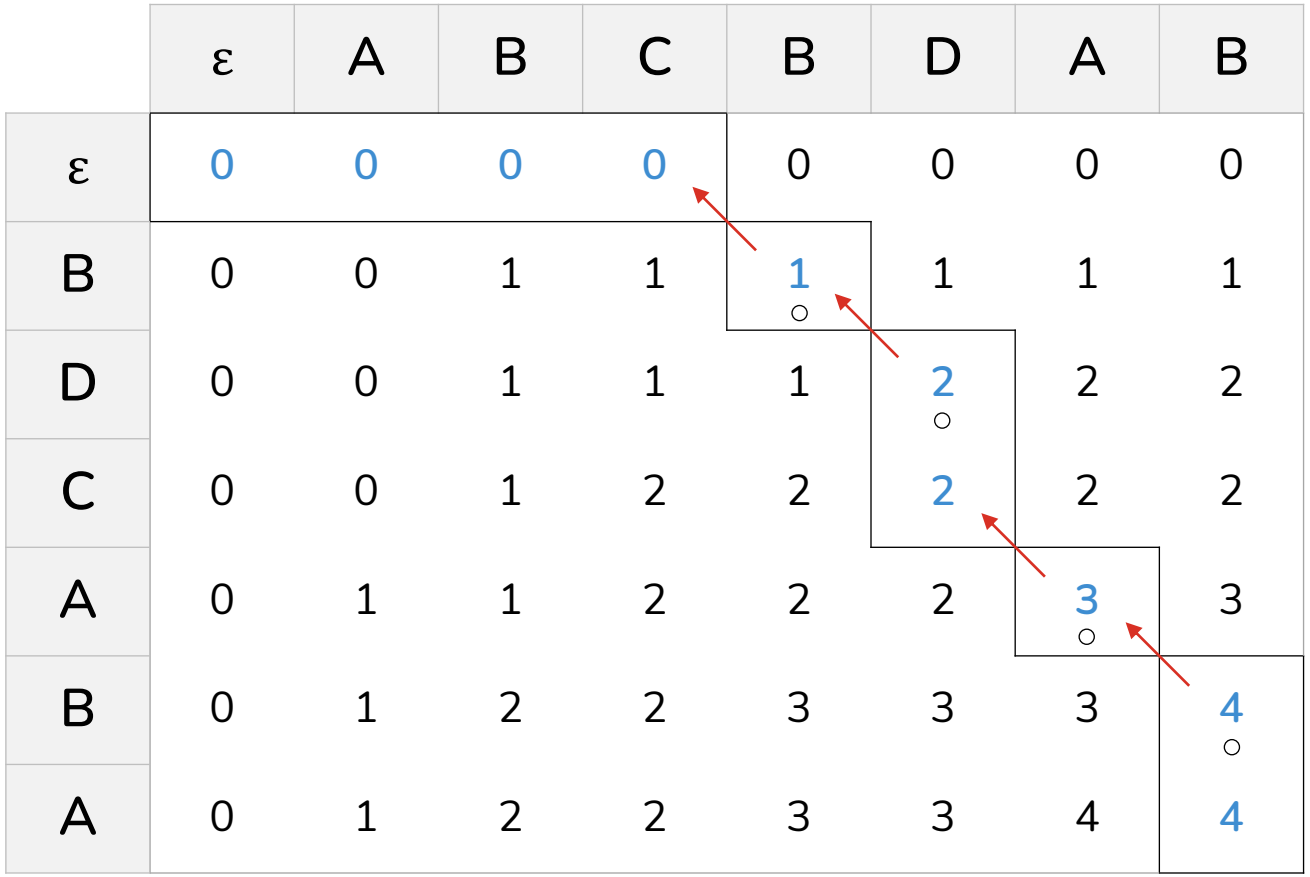
The dot shows a decrease in length. When the length decreases, we have a common character.

	ε	A	B	C	B	D	A	B
ε	0	0	0	0	0	0	0	0
B	0	0	1 ○	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2 ○	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3 ○	3	3	4
A	0	1	2	2	3	3	4 ○	4

Finding the LCS (a different trace backwards)

We can move left or up when we have the same value and diagonally if there is an ‘arrow’.

The dot shows a decrease in length. When the length decreases, we have a common character.



	ε	A	B	C	B	D	A	B
ε	0	0	0	0	0	0	0	0
B	0	0	1	1	1 ○	1	1	1
D	0	0	1	1	1	2 ○	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3 ○	3
B	0	1	2	2	3	3	3	4 ○
A	0	1	2	2	3	3	4	4

Conclusion

- Time = O(nm).
- Space = O(nm).
  - Note that space can be decreased substantially. One way is to keep only row above the current one or the column to the left of the current one (whichever one is smaller): O(min(n,m)).
  - This makes reconstruction of the LCS string a bit more complex but still possible.

Further reading

- These notes should be supplemented by:
  - Introduction to Algorithms (Clifford Stein, Thomas H Cormen, Ronald L Rivest, Charles E Leiserson – MIT Press)
  - Also see Eric Demaine: http://videolectures.net/mit6046jf05_leiserson_lec15/

Skip Lists

Kristian Guillaumier

Skip lists

- We want to keep a collection of items, and want to efficiently search, insert and delete in it.
- We could use:
  - Arrays: but they are not dynamic.
  - Self balancing trees (e.g., B-Trees, RBTs, AVL Trees).
- In 1989 the skip list data structure was proposed by William Pugh.
- The skip list is simpler to understand and implement than self-balancing trees, and it is very efficient.
- It is also a good example of a ‘probabilistic’ data structure.

...continued

- A skip list is based on a linked list.
- A basic linked list has a problem:
  - We don't have random access – in other words even though it might be sorted we cannot search in logarithmic time like an array.
- A skip list is a randomised (we'll explain later) data structure that solves the above problem.

Efficiency

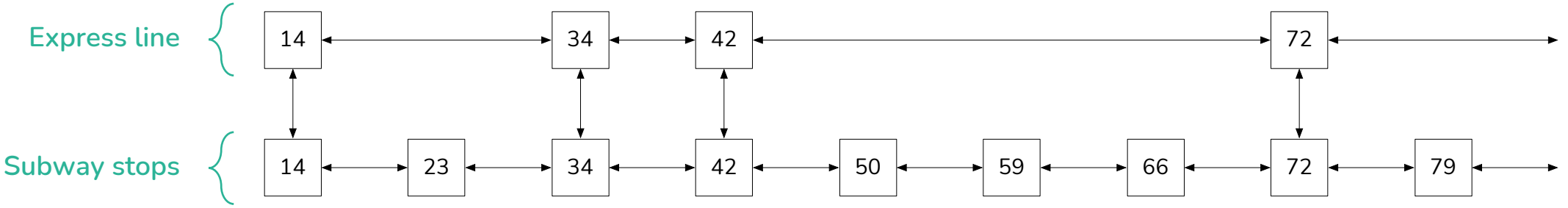
- In the previous data structures, we have seen, we have mentioned bounds (e.g., $O(\log_2 n)$ worst case for RBTs) on the performance.
- The performance of Skip Lists is in expectation:
  - i.e., we expect $O(\log_2 n)$ performance – note that this observation is not as powerful as an actual bound.
- But…
  - We will prove that it is $O(\log_2 n)$ with a very high probability.
  - This will be true for every operation.

Skip lists

- The basis of the algorithm is a linked list.
- We assume that the list is sorted – we'll make sure of this in the insert operation.
- Remember that we still cannot do a binary search on a linked list even if it is sorted.

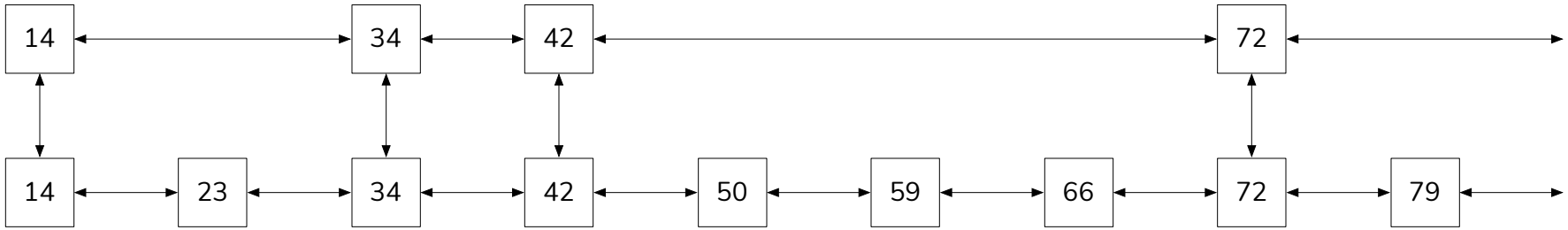
Illustration

- Overview of the method…
- These are street numbers with the following the subway stops (7th avenue line).
- The subway has an express line (this makes it a skip list).



Example from Demaine

A simple example



Note how the ‘local’ line has all the elements.

The express line has a subset.

If we want to go to 59:

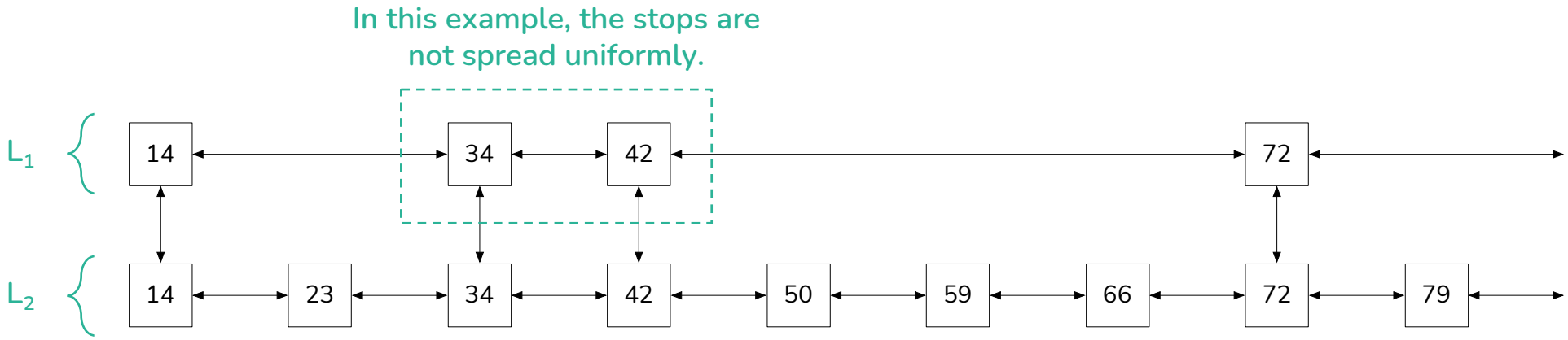
1. We take the express.
2. At 42, we are told that the next station is 72 (i.e., greater than the 59 we are looking for).
3. We get off the express.
4. We get on the local line and continue.

Overview of a search

- Init: Let L = topmost list.
- Move right in L until we either find what we are looking for and return it, or the next value is greater than we are looking for.
- If the next value is greater than we are looking for:
    1. If this is the bottom list, then the item is not found.
    2. Otherwise, move to the next lower list.
- Restart.

Skip lists

- If the lists are drawn from top to bottom (where bottom is the full list), what do we put in L_1?
- If this is a subway, we would put the most popular stops in it, but, in general, we do not have this kind of 'domain' knowledge.
- Ideally, we would spread the stops out uniformly.



Uniformly spread

Assuming $|L_1|$ is half $|L_2|$:

```
L1:  1   ↔   3   ↔   5   ↔   7   ↔   9
     ↕       ↕       ↕       ↕       ↕
L2:  1 ↔ 2 ↔ 3 ↔ 4 ↔ 5 ↔ 6 ↔ 7 ↔ 8 ↔ 9 ↔ 10
```

Another uniform spread:

```
L1:  1       ↔       5
     ↕               ↕
L2:  1 ↔ 2 ↔ 3 ↔ 4 ↔ 5 ↔ 6 ↔ 7 ↔ 8 ↔ 9 ↔ 10
```

Both are uniform spreads. Is one better than another?

Some analysis

- What is the maximum number of steps if we have two lists L1 and L2 in my skip list?
- If we spread out uniformly, in the worst-case scenario (last item) we would have to:
  - Go through $|L_1|$ steps.
  - Go down to $|L_2|$ and go through $\frac{|L_2|}{|L_1|}$ steps.
  - In total we have $|L_1| + \frac{|L_2|}{|L_1|}$ steps.
  - We want to minimise this.

...continued

- $|L_2|$ (the bottom list) is n, where n is the number of elements in the data structure.
- To minimize, we can only choose the size of L_1 since n is not a choice:

$$|L_1| + \frac{n}{|L_1|}$$

- This formula will be smaller when:

$$\underbrace{|L_1|}_{L_1 \text{ is smaller}} + \underbrace{\frac{n}{|L_1|}}_{L_1 \text{ is larger}}$$

We must find a balance

...continued

- So, we must balance:

$$\underbrace{|L_1|}_{\text{This half}} + \underbrace{\frac{n}{|L_1|}}_{\text{And this half}}$$

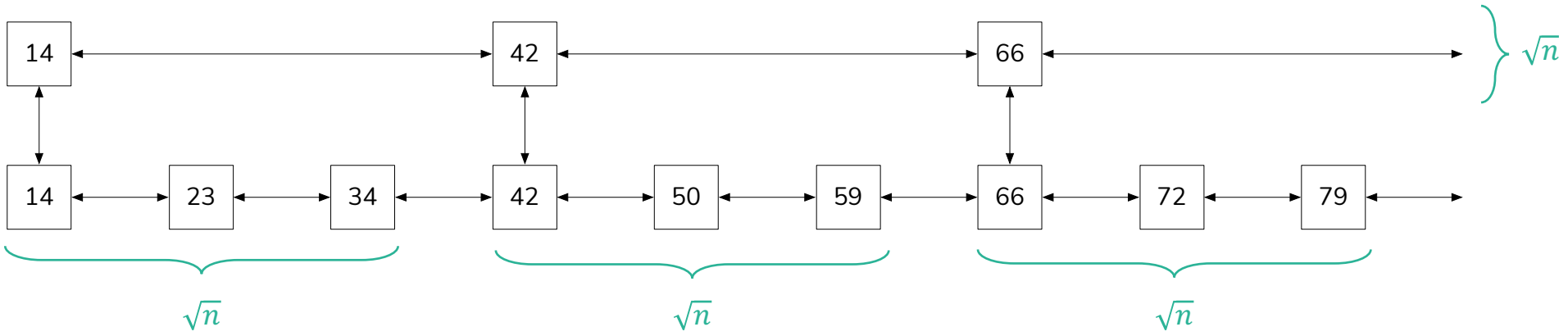
- The formula is minimised when the 2 ‘halves’ are the same:
  - $|L_1| = \frac{n}{|L_1|}$
  - Rearranged...
  - $|L_1|^2 = n$
  - $|L_1| = \sqrt{n}$

...continued

- Let's work out some examples for $n = 150$ items:
  - Let's try $|L_1| = 2$:
    - The formula $|L_1| + \frac{n}{|L_1|} = 2 + \frac{150}{2} = 77$
  - Let's try $|L_1| = 50$:
    - The formula $|L_1| + \frac{n}{|L_1|} = 50 + \frac{150}{50} = 53$
  - Let's try $|L_1| = \sqrt{150} = 12.25$:
    - The formula $|L_1| + \frac{n}{|L_1|} = 12.25 + \frac{150}{12.25} = 24.5$

...continued

- Since the two 'halves' of $|L_1| + \frac{n}{|L_1|}$ are the same and each half is $\sqrt{n}$ then the search cost is $\sqrt{n} + \sqrt{n} = 2\sqrt{n}$
- So, a normal linked list has a n 'time performance' and skip list with two lists gives me $2\sqrt{n}$ 'time performance' – better but not as good as it can get.



Example from Demaine

…continued

- O($\sqrt{n}$) is better than O(n) but still not nearly as good as O($\log_2 n$) time.
- The solution is to add a third ‘express line’ and a fourth, and fifth, and…
- So…
  - For two lists: $2\times\sqrt{n}$
  - For three lists: $3\times\sqrt[3]{n}$
  - For k lists: $k\times\sqrt[k]{n}$

$k \times \sqrt[k]{n}$

- Consider two lists. What is the search cost?

Search cost = $L1 + \frac{n}{L1}$

a $L1 = \frac{n}{L1} \Rightarrow L1 = \sqrt{n}$

$\sqrt{n} + \sqrt{n} = 2 \times \sqrt{n}$

But all 'parts' are equal

- Consider three lists. What is the search cost?

Search cost = $L2 + \frac{L1}{L2} + \frac{n}{L1}$

a $L2 = \frac{L1}{L2} \Rightarrow L2^2 = L1$

b $L2 = \frac{n}{L1} \Rightarrow L2 = \frac{n}{L2^2} \Rightarrow L2^3 = n \Rightarrow L2 = \sqrt[3]{n}$

$\sqrt[3]{n} + \sqrt[3]{n} + \sqrt[3]{n} = 3 \times \sqrt[3]{n}$

But all 'parts' are equal

$k \times \sqrt[k]{n}$

- Consider four lists. What is the search cost?

Search cost = $L3 + \frac{L2}{L3} + \frac{L1}{L2} + \frac{n}{L1}$

a $L3 = \frac{L2}{L3} \Longrightarrow L3^2 = L2$

b $L3 = \frac{L1}{L2} \Longrightarrow L3 = \frac{L1}{L3^2} \Longrightarrow L3^3 = L1$

c $L3 = \frac{n}{L1} \Longrightarrow L3 = \frac{n}{L3^3} \Longrightarrow L3^4 = n \Longrightarrow L3 = \sqrt[4]{n}$

$\sqrt[4]{n} + \sqrt[4]{n} + \sqrt[4]{n} + \sqrt[4]{n} = 4 \times \sqrt[4]{n}$

But all 'parts' are equal

- We can go on making this argument for k.

...continued

- The question now is, how many lists k do we need?
- We don't want k to be n otherwise we will have n levels and we will need to 'fall down' to a lower level n times and end up with O(n).
- We don't want k to be too small otherwise we will have 'root' time which is not as good as Log_2 time.
- We want k to be logarithmic to have logarithmic depth.
- So, for $\log_2 n$ lists, we get a search performance of $\log_2(n) \times \sqrt[\log_2(n)]{n}$.

...continued

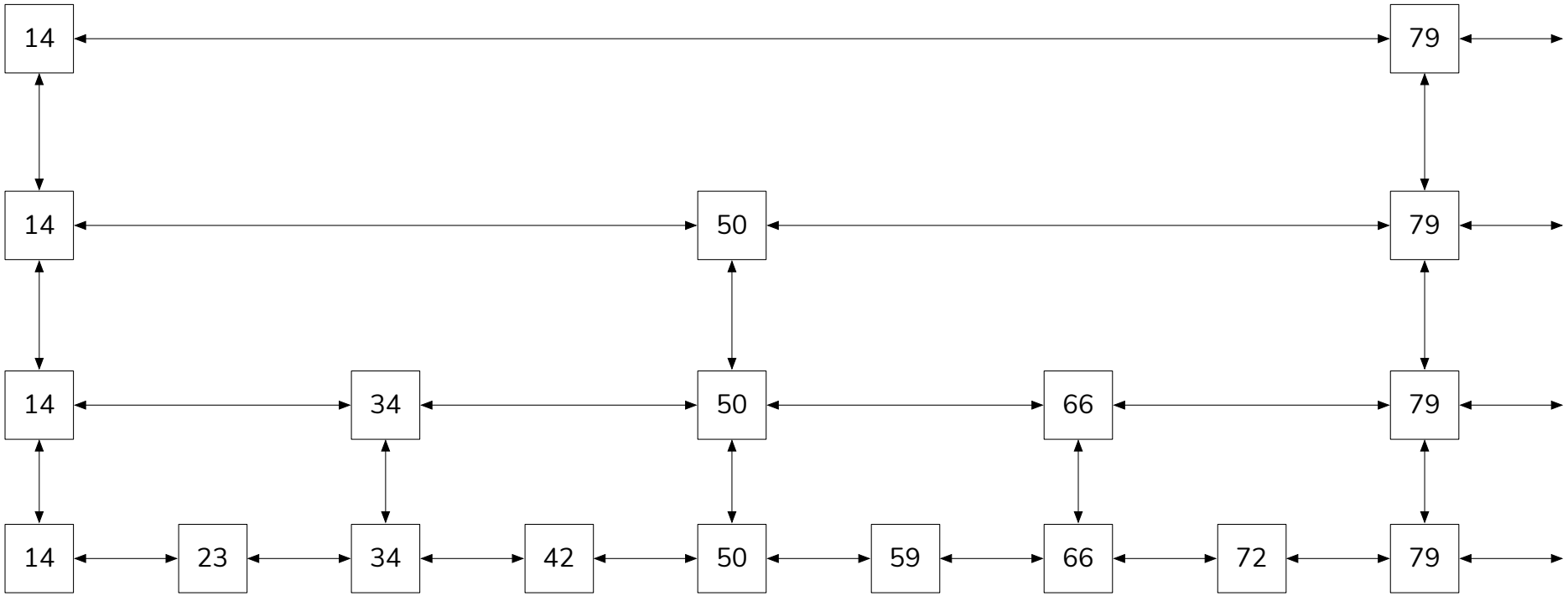
- Breaking down $\log_2(n) \times \sqrt[\log_2(n)]{n}$
- We know the y^{th} root of a number n is $\sqrt[y]{n} = n^{\frac{1}{y}}$
- So...
- $\log_2(n) \times \sqrt[\log_2(n)]{n} = \log_2(n) \times n^{\frac{1}{\log_2(n)}}$

Remember that $a^b = 2^{b \times \log_2(a)}$

$$n^{\frac{1}{\log_2(n)}} = 2^{\frac{1}{\log_2(n)} \times \log_2(n)} = 2^1 = 2$$

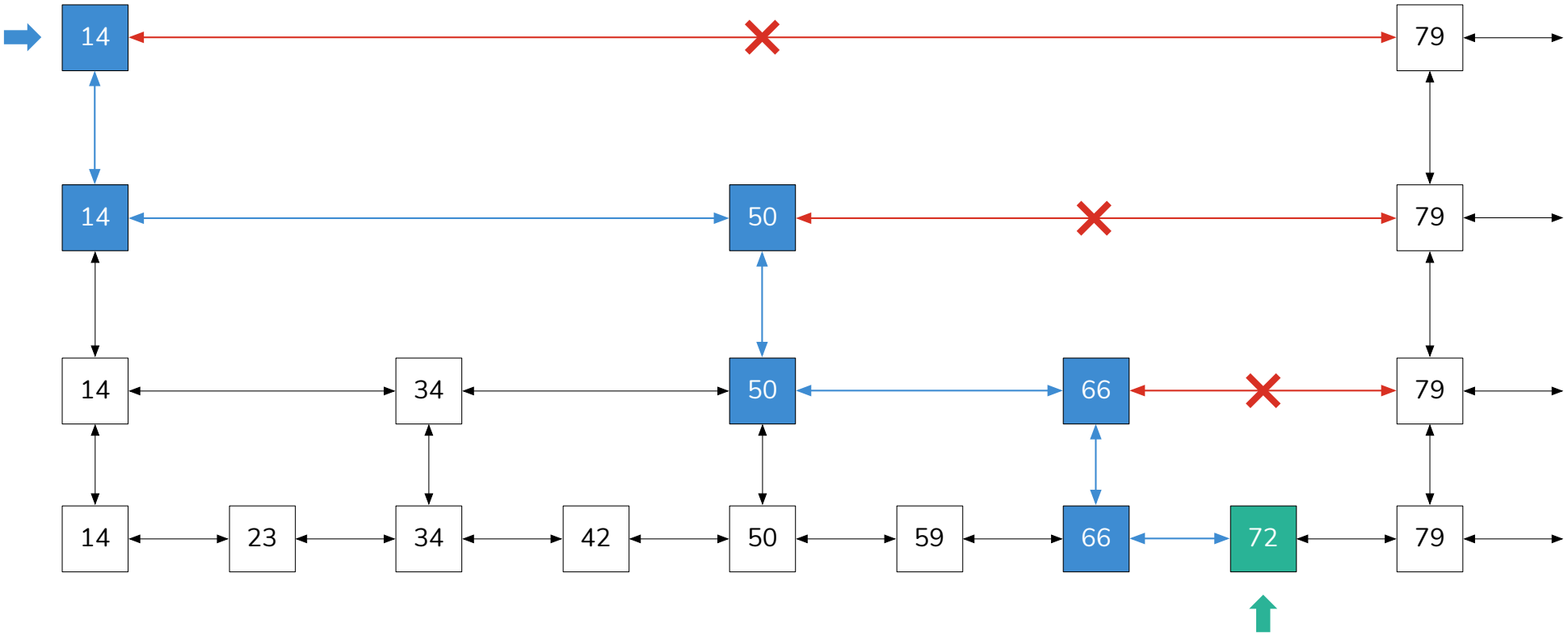
- Giving $2 \times \log_2(n)$
- So, for $\log_2(n)$ lists, we need to perform $2 \times \log_2(n)$ search steps – logarithmic!

A simple search example – find 72



Example from Demaine

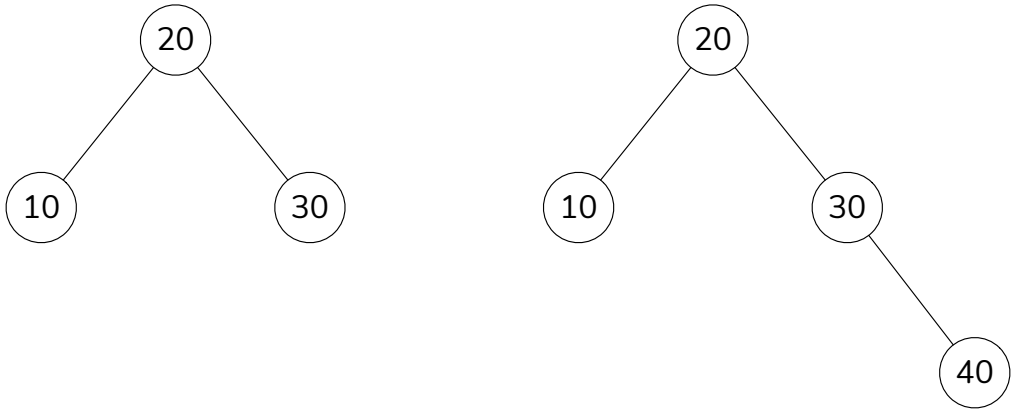
A simple search example – find 72



Example from Demaine

Skip lists

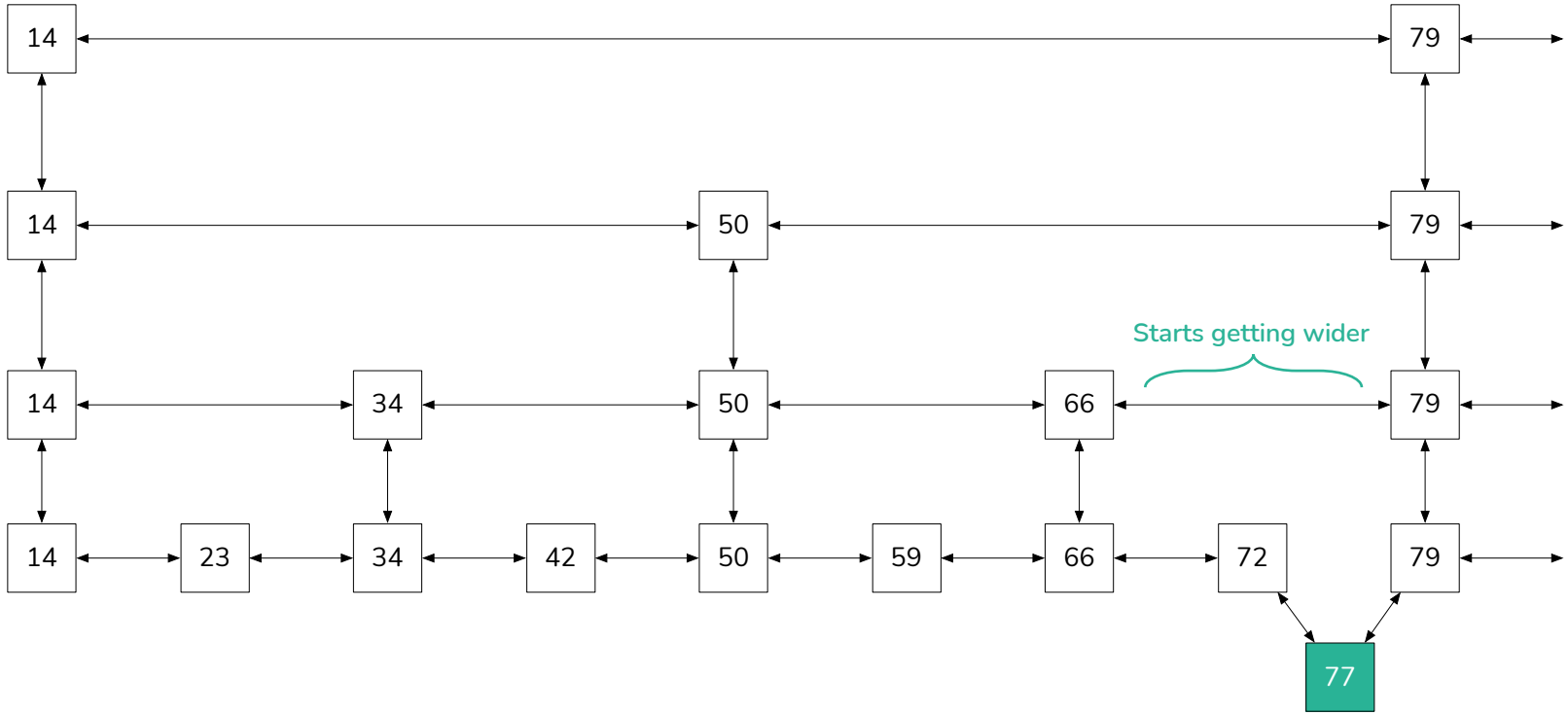
- The skip list in the previous illustration we have seen is an ideal skip list. This is like the concept of an ideal BST – insertions and deletions will ‘corrupt’ this structure.
- Ideal BST and insertion example:



Insertion

- As usual, we run a search to find where the item should fit in the bottom list.
- Remember that, by definition, the bottom list contains all the elements.
- Once we find the place in the bottom list, we insert it (in the bottom list). Note that we have now ‘de-idealised’ our skip list.

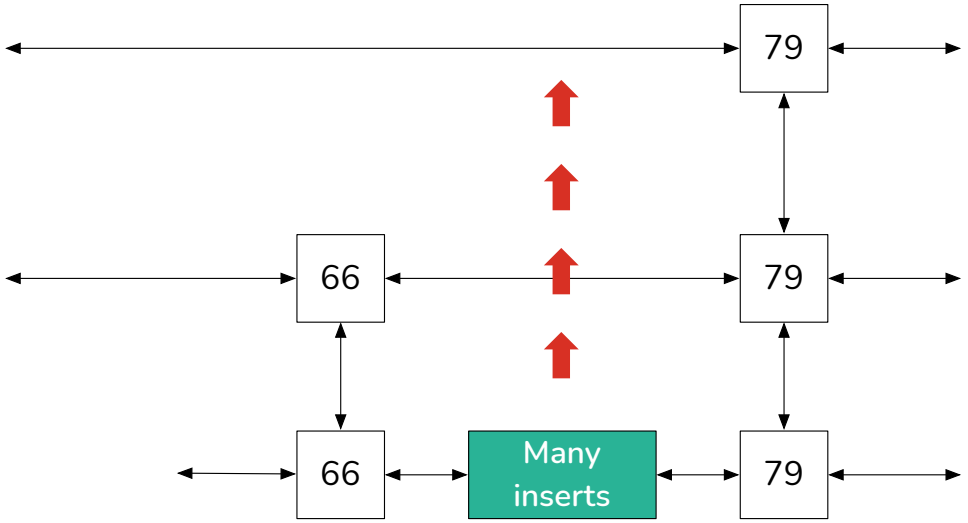
Inserting the value 77



Note how the length of this segment is longer, starting to make things inefficient.

...continued

- We can see that after a number of insertions, the ‘segment’ starts becoming longer and longer compared to the others (the skip list property is damaged).
- Clearly, some items must move to upper lists to maintain the skip list ‘balance’. How? What condition?



Some of the items we insert need to be promoted to higher levels. Which ones? How?

...continued

- Intuitively, some of the elements we insert must be promoted up a level (for each level) to maintain ratio of elements between lists.
- In skip lists, the answer to whether an inserted element gets promoted to an upper level is determined by flipping a coin (50/50 chance – half go up).
- Heads = promote up.
- Tails = don't promote.

We will show that, using this strategy, we can expect our operations to still be $O(\log_2 n)$*.*

A small problem when the list is empty

Insert 9. Do we promote? Toss → No. Good. So far:

```
[9]
```

Insert 20:

```
[9]↔[20]
```

Do we promote? Toss → yes:

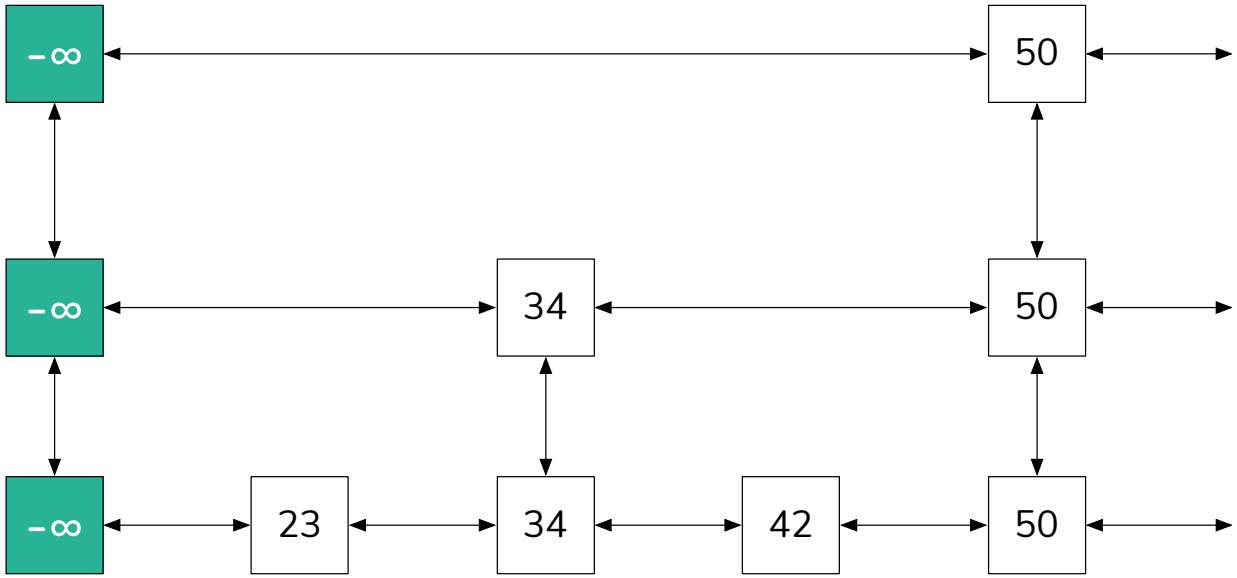
```
    [20]
      ↕
[9]↔[20]
```

Oops… I don't have a top-left from where to start searching:

```
❌ [20]
      ↕
[9]↔[20]
```

Solution

- Add a special value of $-\infty$ to each list:



Example from Demaine

Deletion

- Note that deletion is trivial.
- If I want to delete X, just delete its occurrence from every list.
- Done.

Theorem

With high probability, every search in an n element skip list costs $O(log_2 n)$

Preparation: what does WHP mean?

- Remember, $O(\log_2 n)$ is saying "order $\log_2(n)$".
- So…
  - $1 \times \log_2(n)$ is order $\log_2(n)$.
  - $1.3 \times \log_2(n)$ is order $\log_2(n)$ as well.
  - $10 \times \log_2(n)$ is order $\log_2(n)$ as well.
- In other words, if the cost is $c \times \log_2(n)$ where c is some constant, we still have "order $\log_2(n)$" performance, which is what we want.

Preparation: what does WHP mean?

- An event occurs WHP if for any number $\alpha \geq 1$, the event occurs with probability at least (greater than):

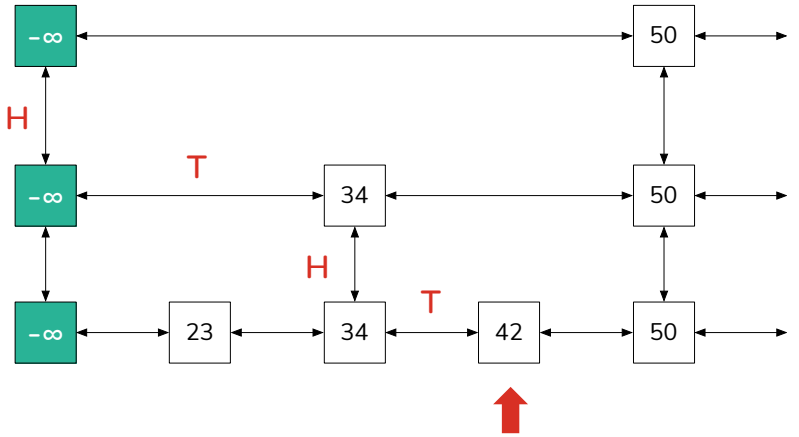
$$1 - O\left(\frac{1}{n^{\alpha}}\right)$$

- So, if the event E occurs with a probability of at least $1 - O\left(\frac{1}{n^{\alpha}}\right)$, this means that the event will not occur with a probability of at most (less than):

$$O\left(\frac{1}{n^{\alpha}}\right)$$

Preparation: reasoning about a search

- Suppose that we analyse a search backward.
- In a normal search we start at top-left −∞ then move right or down.
- In a backward search we…
  - Start at the result.
  - If we can move up, we got heads. Otherwise, if we can move left, we got tails.
  - Finish top-left −∞.
  - Think of any search as a string such as “HHHTTHTHTTTTTH”.



Preparation: some stats

- Recall that the notation $\binom{n}{r}$ denotes the number of ways that we can choose r objects from a set containing n distinct elements.
- Also, $\binom{n}{r} = \frac{n!}{r!(n-r)!}$
- So, if a flip a coin 4 times, in how many ways can we get 1 heads?

$$\frac{4!}{1!\,(4-1)!} = \frac{24}{6} = 4$$

HTTT, THTT, TTHT, TTTH

Preparation: some stats

- If I flip a coin 4 times, in how many ways I can get 2 heads:

$$\binom{4}{2} = \frac{4!}{2!\,(4-2)!} = \frac{24}{4} = 6$$

HHTT, HTHT, HTTH, THHT, THTH, TTHH

Preparation: some stats

- Let's say we flip a coin 5 times. What is the probability of getting exactly 2 heads?

$$\underbrace{\binom{5}{2}}_{\text{Number of different ways I can get this.}} \times \underbrace{\left(\frac{1}{2}\right)^{2}}_{\text{Probability of getting 2 heads.}} \times \underbrace{\left(\frac{1}{2}\right)^{3}}_{\text{Probability of getting 3 tails.}}$$

} This is obviously the total number of flips less what we are looking for.

Preparation: some stats

- Let's say I flip a coin 5 times. What is the probability of getting at most 2 heads?

$$\binom{5}{2} \times \left(\frac{1}{2}\right)^2 \times \left(\frac{1}{2}\right)^3$$

$$= \binom{5}{2} \times \left(\frac{1}{2}\right)^3$$

Another way of saying "getting at least 3 tails".

Preparation: some stats

- Also note that there is a known bound (we won't prove this here) that says:

$$\binom{n}{r} \leq \left(e.\frac{n}{r}\right)^r$$

e is Euler's number, the base of the natural log, 2.718...

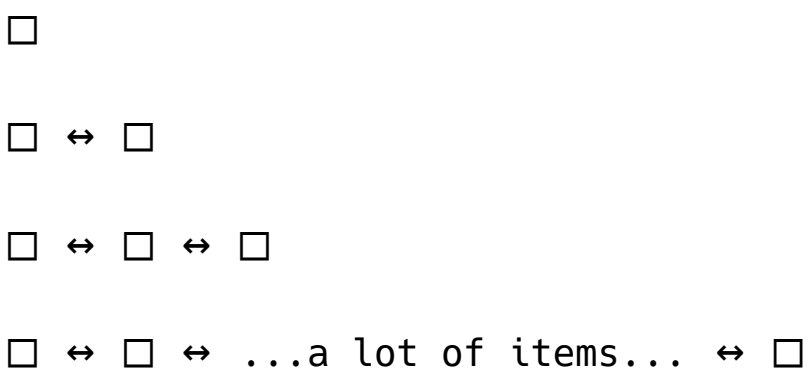
There is a nice explanation here:
https://en.wikipedia.org/wiki/Binomial_coefficient#Bounds_and_asymptotic_formulas

Back to our proof, our strategy will be…

- First, we will show that the number of levels in a skip list is bound by $O(\log_2 n)$.
- This means that in any search there will not be any more than $O(\log_2 n)$ down moves (or up moves if we're using the backward search idea).
- Which also means that in any sequence "HHTTTHTHH…" there will not be more than $O(\log_2 n)$ H moves.
- Additionally, by the time we have performed all our H moves, the search is complete.
- We finally need to show that by the time we perform $O(\log_2 n)$ H moves, the total cost of the search (the length of the H-T sequence) is also $O(\log_2 n)$.

Step 1: proving a bound on the height

- Remember we are trying to prove that all searches in a skip list with n elements take $O(\log_2 n)$ steps.
- In BSTs we know that if we can show that the number of levels in the tree is $O(\log_2 n)$ then we get $O(\log_2 n)$ search performance.
- Unfortunately, this is not true for skip lists. We could have $O(\log_2 n)$ levels but it could still be 'unbalanced'.
- An extreme example:



Height is bound but search is still very inefficient.

Step 1: proving a bound on the height

- Although the height does not help us establish performance of the search, proving a bound is useful to prove our result.
- So…

Lemma: WHP a skip list with n elements will not have more than $O(log_2 n)$ levels.

Step 1: proving a bound on the height

- Probability of an element going up 1 level is $\frac{1}{2}$
- Probability of an element going up 2 levels is $\frac{1}{2} \times \frac{1}{2} = \frac{1}{4}$
- Probability of an element going up 3 levels is $\frac{1}{2} \times \frac{1}{2} \times \frac{1}{2} = \frac{1}{8}$
- Probability of an element going up k levels is $\frac{1}{2} \times \frac{1}{2} \times \cdots \times \frac{1}{2} = \frac{1}{2^k}$
- Probability of an element going up $c.\log_2 n$ levels is $\frac{1}{2^{c.\log_2 n}}$

Step 1: proving a bound on the height

- So, the probability of an element going up more than $c.\log_2 n$ levels is $< \frac{1}{2^{c.\log_2 n}}$.
- But we are not dealing with only one element – we are inserting n elements.
- So, the probability of any one of the n elements going up more than $c \times \log_2 n$ levels is:

$$< n \times \frac{1}{2^{c.\log_2 n}}$$

$$< n \times \frac{1}{n^c}$$

$$< \frac{n}{n^c}$$

$$< \frac{1}{n^{c-1}}$$

So, the probability is polynomially small.
The larger we make c the less probability there is it will go beyond $c.log_2 n$

$$c - 1 = \alpha$$

Step 2: final proof

With high probability, every search in an n element skip list costs $O(log_2 n)$

To show this, we will use our ‘backwards’ search idea…

Step 2: final proof

- From our lemma we know that we cannot make more than $c.\log_2 n$ 'up' moves (WHP).
- So, we have a bound on the number of 'up' moves.
- Remember that in all the flips we make, we can never (WHP) generate more than $c.\log_2 n$ heads – we can never go beyond the highest level.

So, we want to show that WHP the total number of flips until we get $c.log_2 n$ heads is $O(log_2 n)$

Step 2: final proof

- Let's say that we make $10.c.\log_2 n$ total flips. What is the probability that we get exactly $c.\log_2 n$ heads?

$$\underbrace{\binom{10.c.\log_2 n}{c.\log_2 n}}_{\text{Number of different ways I can get this.}} \times \underbrace{\left(\frac{1}{2}\right)^{c.\log_2 n}}_{\text{Probability of getting heads.}} \times \underbrace{\left(\frac{1}{2}\right)^{9.c.\log_2 n}}_{\text{Probability of getting tails.}}$$

Step 2: final proof

- So, after making $10.c.\log_2 n$ total flips …

$$\Pr(\text{exactly } c.\log_2 n \text{ heads}) = \binom{10.c.\log_2 n}{c.\log_2 n} \times \left(\frac{1}{2}\right)^{c.\log_2 n} \times \left(\frac{1}{2}\right)^{9.c.\log_2 n}$$

- Then…

$$\Pr(\text{at most } c.\log_2 n \text{ heads}) = \binom{10.c.\log_2 n}{c.\log_2 n} \times \left(\frac{1}{2}\right)^{c.\log_2 n} \times \left(\frac{1}{2}\right)^{9.c.\log_2 n}$$

Step 2: final proof

- Now, we rearrange a bit using $\binom{n}{r} \leq \left(e.\frac{n}{r}\right)^r$ } We discussed this earlier in our preparation
- We fix:

$$\Pr(\text{at most } c.\log_2 n \text{ heads}) = \binom{10.c.\log_2 n}{c.\log_2 n} \times \left(\frac{1}{2}\right)^{9.c.\log_2 n}$$

- To get:

$$\Pr(\text{at most } c.\log_2 n \text{ heads}) \leq \left(e \times \frac{10.c.\log_2 n}{c.\log_2 n}\right)^{c.\log_2 n} \times \left(\frac{1}{2}\right)^{9.c.\log_2 n}$$

Step 2: final proof

$$\leq \left(e \times \frac{10.c.\log_2 n}{c.\log_2 n}\right)^{c.\log_2 n} \times \left(\frac{1}{2}\right)^{9.c.\log_2 n}$$

$$\left(\frac{1}{2}\right)^{9.c.\log_2 n} = \frac{1^{9.c.\log_2 n}}{2^{9.c.\log_2 n}} = \frac{1}{2^{9.c.\log_2 n}}$$

$$\leq (10.e)^{c.\log_2 n} \times \frac{1}{2^{9.c.\log_2 n}}$$

$$\leq \frac{(10.e)^{c.\log_2(n)}}{2^{9.c.\log_2 n}}$$

Remember that $a^b = 2^{b \times \log_2(a)}$

$$(10.e)^{c.\log_2 n} = 2^{\log_2(10.e)c.\log_2(n)}$$

$$\leq \frac{2^{\log_2(10.e).c.\log_2(n)}}{2^{9.c.\log_2(n)}}$$

Step 2: final proof

$$\leq \frac{2^{\log_2(10.e).c.\log_2(n)}}{2^{9.c.\log_2 n}}$$

$$\leq 2^{(\log_2(10e)-9).c.\log_2(n)}$$

$$\leq 2^{-\alpha.\log_2 n}$$

$$\leq \frac{1}{2^{\alpha.\log_2 n}}$$

$$\leq \frac{1}{n^{\alpha}}$$

Note that as the 10 approaches ∞, then the 9 increases **linearly** but the $\log_2(10e)$ increases **logarithmically** (more slowly), so it is:

- Negative
- $(\log_2(10e) - 9)$ goes to $-\infty$

Remember that $a^b = 2^{b \times log_2(a)}$

$$2^{\alpha.\log_2 n} = n^{\alpha}$$

Step 2: final proof

- So, we have shown that if we perform $Q.c.\log_2 n$ flips in total, the probability of getting $\leq c.\log_2 n$ heads is incredibly small…

$$\Pr(\text{at most } c.\log_2 n \text{ heads}) \leq \frac{1}{n^{\alpha}}$$

- Which means that by the time we perform $Q.c.\log_2 n = O(\log_2 n)$ search steps to find a result, we would have performed $c.\log_2 n = O(\log_2 n)$ up moves, and by lemma 1, we cannot perform more than that.

Recap

- An event E occurs WHP if probability is $\geq 1 - O\left(\frac{1}{n^{\alpha}}\right)$.
- An event E occurs WLP if probability is $< O\left(\frac{1}{n^{\alpha}}\right)$.

...continued

- The probability of an element reaching $c.\log_2 n$ levels is $\frac{1}{2^{c.\log_2 n}}$
- The probability of any of my n elements reaching $c.\log_2 n$ levels is $\leq$ $\frac{1}{2^{c.\log_2 n}} + \cdots + \frac{1}{2^{c.\log_2 n}}$ for n times (by Boole's inequality).
- Rearranged:
  - $\leq \frac{n}{2^{c.\log_2 n}}$
  - $\leq \frac{1}{n^{c-1}}$

Where $\alpha = c - 1$. So, the probability of any one of our n elements reaching $c.log_2 n$ levels is very small (within our definition of WLP).

...continued

- A search is performed from top-left down to bottom-right.
- We visualise the search in reverse.
- In reverse, we can:
  - Move to the left: i.e., we had gotten a tails for that element.
  - Move up: i.e., we had gotten a heads for that element.
- So, a search is a sequence HHTTTHHTH...

…continued

- We have shown that WHP we cannot make more than $c.\log_2 n$ up moves.
- Remember that the probability of any of our n elements going up more than $c.\log_2 n$ levels is $< \frac{1}{n^{\alpha}}$
- So, in any search HHTTTHHTH… we will not have more than $c.\log_2 n$ “H’s”.
- Clearly, the cost of a search is the length of a ‘string’ of HHTTTHHTH… which is equivalent to the total number of flips made to insert that item.
- Clearly, #Flips = #Heads + #Tails (and so far, we have a bound on #heads).

...continued

- Suppose that I start flipping coins.
- At some point, I will have flipped $c.\log_2 n$ heads.
- At this point, I cannot flip any more heads.
- That is, by now I have performed the total number of flips.
- I need to show that this total number of flips (cost of the search) is $O(\log_2 n)$.

…continued

- Let's say that we performed $O(\log_2 n)$ flips.
- What is the probability of getting at most ($\leq$) $c.\log_2 n$ heads?
- We have shown that this probability is $\leq \frac{1}{n^{\alpha}}$
- In other words, $\Pr(\text{getting } \leq c.\log_2 n \text{ heads}) \leq \frac{1}{n^{\alpha}}$
- The implication is: if we flip $O(\log_2 n)$ times, the probability of getting less than $O(\log_2 n)$ heads is negligible.

Finally…

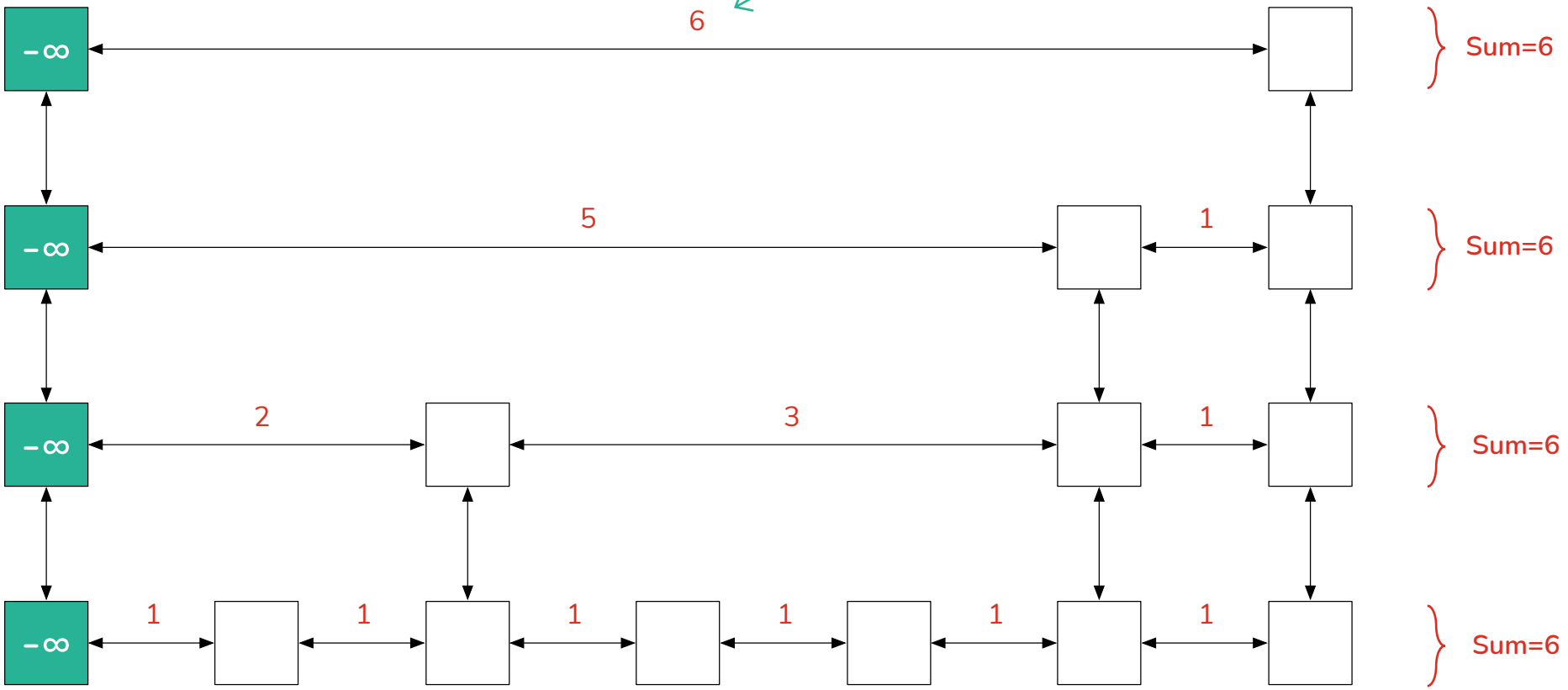
- By the time we flip $O(\log_2 n)$ heads the search is complete.
- By the time we flip $O(\log_2 n)$ heads, we have performed $O(\log_2 n)$ total flips WHP.
- We cannot perform more than $O(\log_2 n)$ heads WHP.
- So, we cannot perform more than $O(\log_2 n)$ total flips.

Indexable skip lists

- So far, our skip lists can do insertions and deletions in $O(\log_2 n)$.
- However, lookups at an index (accessing the ith element) still requires $O(n)$ time.
- This can be easily remedied by storing the 'width' of a link for every link. This way we can get $O(\log_2 n)$ performance to look up at an index.
- Consider the next example…

...continued

The width of a link is the sum of the link widths beneath it.



Note how every link has a width

Accessing the i^{th} element

- To access the i^{th} element, we simply traverse the list summing the widths of each traversed ‘horizontal’ link but go down a level if the adding a link width results in a value greater than i.
- When a new item is inserted, the widths can be efficiently updated as the skip list is traversed during the insert.
- See the technical document called “A skip list cookbook” by William Pugh for implementation details.

Further reading

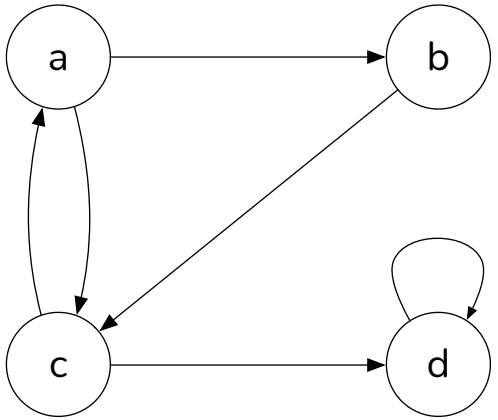
- These notes should be supplemented by:
  - Introduction to Algorithms (Clifford Stein, Thomas H Cormen, Ronald L Rivest, Charles E Leiserson – MIT Press)
- There are very good notes on skip lists:
  - http://videolectures.net/mit6046jf05_demaine_lec12/
- Original skip list paper:
  - ftp://ftp.cs.umd.edu/pub/skipLists/skiplists.pdf

Optimal Substructure and the Handshaking Lemma

Kristian Guillaumier

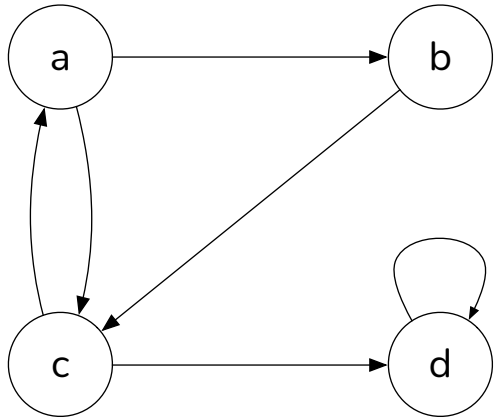
Revision: some graph jargon

- Vertices, edges.
- Labelled vertices, labelled edges.
- Directed, undirected graphs.
- Adjacent vertices (in $a \to b$ the vertex b is adjacent to a).
- The edge $a \to b$ emanates from a. The notation $A(v)$ denotes all the edges that emanate from v.
- The edge $a \to b$ is incident to b. The notation $I(v)$ denotes all the edges incident to the vertex v.



...continued

- Out-degree of a vertex:
  - The number of edges emanating from a node.
  - Written as $|A(v)|$
- In-degree of a vertex:
  - The number of edges incident on a node.
  - Written as $|I(v)|$



Vertex $\mathbf{v}$	$\mathbf{A(v)}$	Out-degree	$\mathbf{I(v)}$	In-degree
a	{(a, b), (a, c)}	2	{(c, a)}	1
b	{(b, c)}	1	{(a, b)}	1
c	{(c, a), (c, d)}	2	{(a, c), (b, c)}	2
d	{(d, d)}	1	{(c, d), (d, d)}	2

Example adapted http://www.brpreiss.com

...continued

- Subgraph:
  - The subgraph S of a graph G is a graph where:
    - Each vertex in S is a vertex in V of G.
    - Each edge in S:
      - Is a subset of the edges in G.
      - Each vertex in the edges of S is a vertex in S (this is implied because a subgraph is a graph).
- Connected graphs:
  - There is a path between every pair of vertices.

Graph representations

- Common representations are the adjacency list and the adjacency matrix.
- Suppose the vertices of a graph are labeled with numbers from 0 to $n-1$.
- Adjacency matrix:
  - An adjacency matrix would then be a 2D array (n by n) of Boolean values.
  - $A[i,j]$ true if there is an edge from i to j.
  - If the graph is weighted, then a value in the matrix is the weight.
  - If the graph is undirected $A[i,j] = A[j,i]$. This will make the matrix symmetrical about the diagonal.

...continued

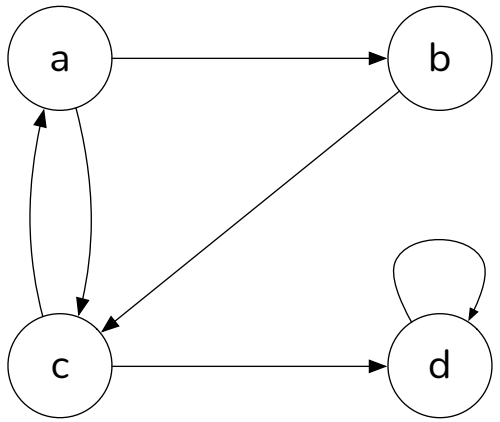
- Consider $G = (V, E)$ where $V = \{v_1, v_2, \ldots, v_n\}$.
- We use an n by n matrix A of Boolean values where:

$$A[i,j] = \begin{cases} 1 \text{ if } (v_i, v_j) \in E \\ 0 \text{ otherwise} \end{cases}$$

$A =$

	a	b	c	d
a	0	1	1	0
b	0	0	1	0
c	1	0	0	1
d	0	0	0	1

$=$



...continued

- In adjacency matrices, space is $O(|V|^2)$.
- This is irrespective of the number of edges in the graph.
- If $|E| \ll |V|^2$ then the matrix will be sparse and will be inefficient – most of it will be zeros.
- Discuss sparse vs. dense.

...continued

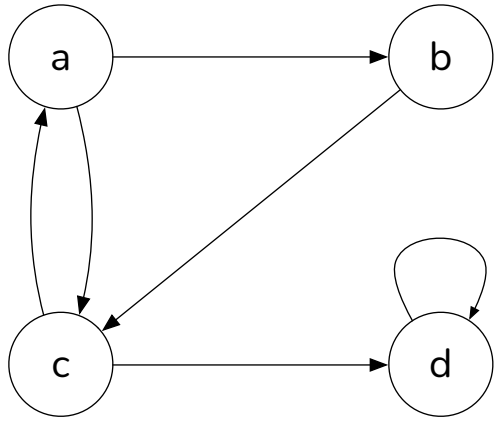
- Adjacency list:
  - Linked list of linked lists.
  - For a graph with n vertices, the primary linked list has n elements. That is, each vertex is a linked list of adjacent vertices.
  - Example:

$$a \rightarrow b \rightarrow c$$

$$b \rightarrow c$$

$$c \rightarrow a \rightarrow d$$

$$d \rightarrow d$$



Suitability

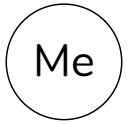
- Consider the operation of determining whether there is an edge between two vertices v_1 and v_2.
  - In adjacency matrix: examine value of $A[v_1, v_2]$.
  - In adjacency list: locate v_1 in primary list, look for v_2 in secondary list.
  - Winner: adjacency matrix.
- Find all vertices adjacent to a vertex v_1 in a graph having n vertices.
  - In adjacency matrix: go to the row for v_1 and iterate over n columns.
  - In adjacency list: go the element v_1 and the secondary list is the list of adjacent vertices.
  - Winner: adjacency list.

The handshaking lemma

- For undirected graphs.
- Colloquially:
  - There is a party.
  - Some people shake other people's hands.
  - Some people might not shake anyone’s hands.
  - I cannot shake hands with myself.
  - Note: if I shook hands with you it implies that you shook hands with me.
  - There will be an even number of handshakes.

Examples 1 and 2

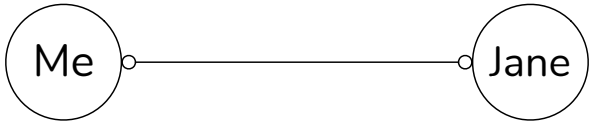
- I am alone. Zero shakes take place (even).



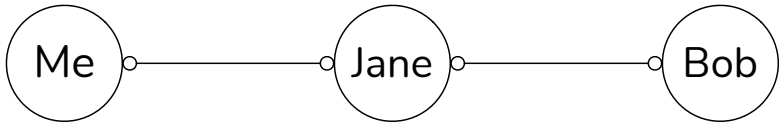
- There are two people at the party, but nobody shakes hands (even).

Examples 3 and 4

- Two people shake hands at the party. I shake hands with you, so you shook hands with me. There are two handshakes. Even.

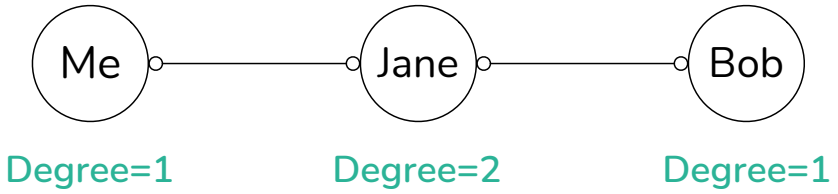


- Three people, shake hands as follows. There are four handshakes. Even.



The handshaking lemma

- Note that I am essentially summing the degrees of every vertex.



$$\sum_{v \in V} \text{Degree}(v) = 2 \times |E|$$

Clearly, every edge is contributing +2 to the result. +1 to one vertex and +1 to another.

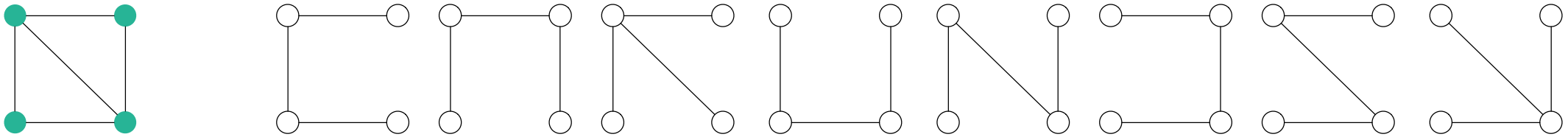
Important

We will be discussing a data structure called a spanning tree

This data structure and related algorithms with be covered as a dedicated topic in much more detail. We are mentioning it here to just as an example to illustrate the principle of optimal substructure and an application of the handshaking lemma.

Spanning trees

- Concerns connected and undirected graphs.
- A spanning tree is a tree that connects all the vertices in a graph and uses some edges.

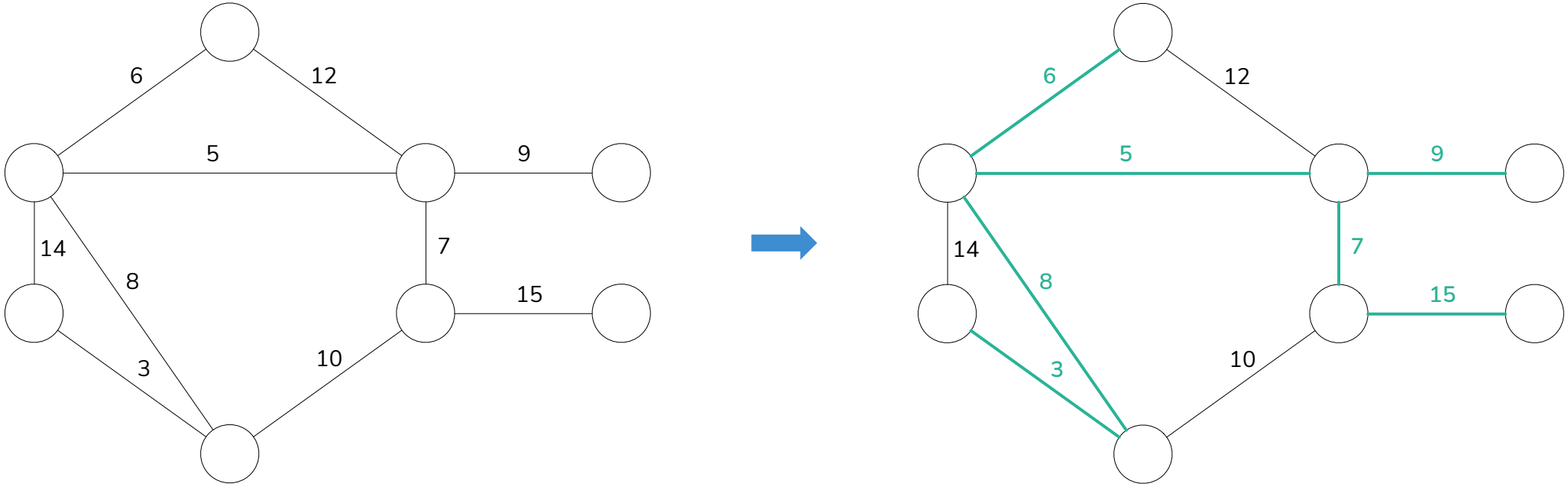


Example from Wolfram

The minimum spanning tree

- Again, concerns connected and undirected graphs.
- The graph is weighted (there is an edge weight function $w: E \to \mathcal{R}$).
- The weight of a spanning tree in the graph is the sum of the weights it uses.
- The minimum spanning tree is the spanning tree in the graph having the smallest weight.
- Major applications in distributed systems.
- From here on, we will assume that all edge weights are distinct.

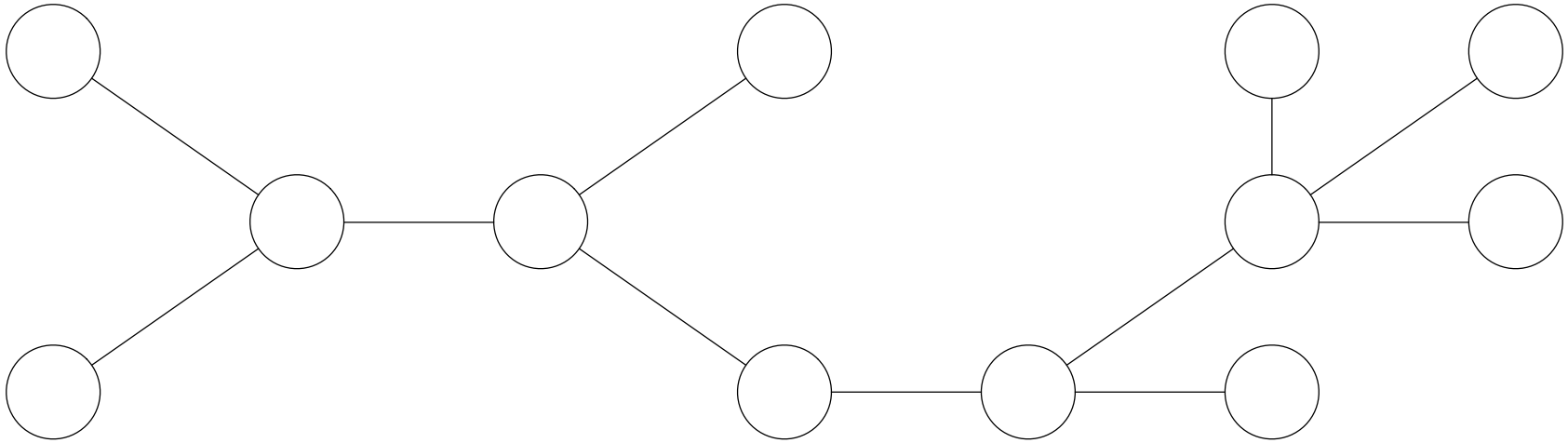
An example MST



Example from Eric Demaine

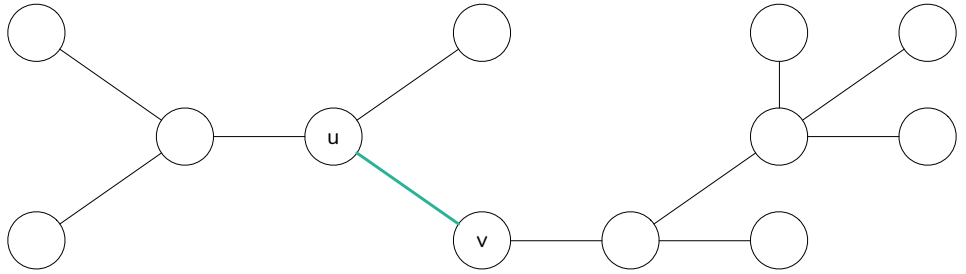
Claim: MSTs have optimal substructure

- Let’s assume we have the following MST.
- Only the edges in the MST are shown (so we’re just seeing the tree here).



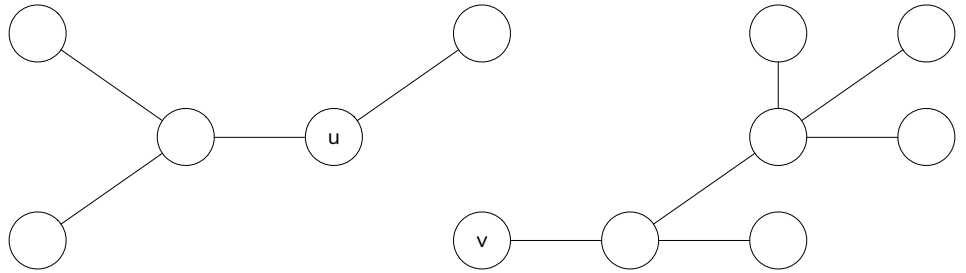
...continued

- If we remove any edge (u,v) we will end up partitioning the tree in two.



We will end up with two trees T1 and T2.

T1 =

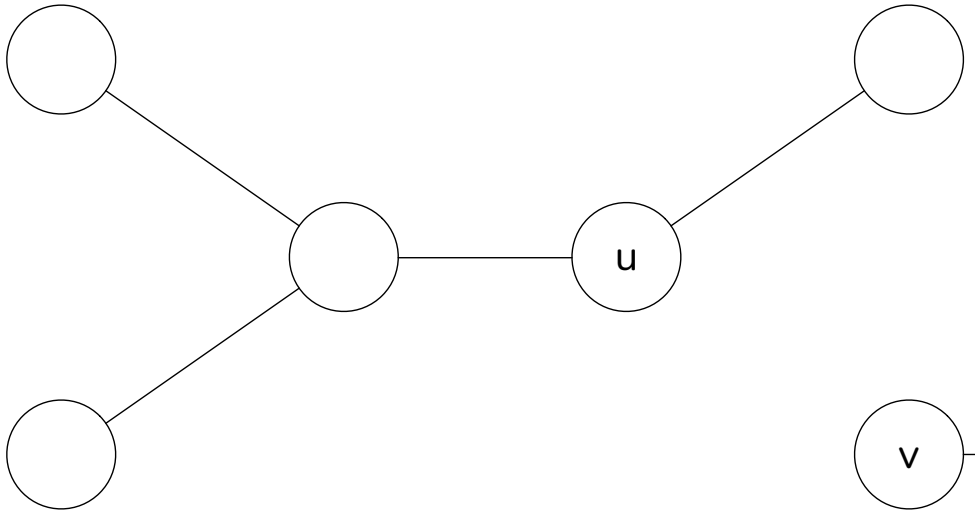


= T2

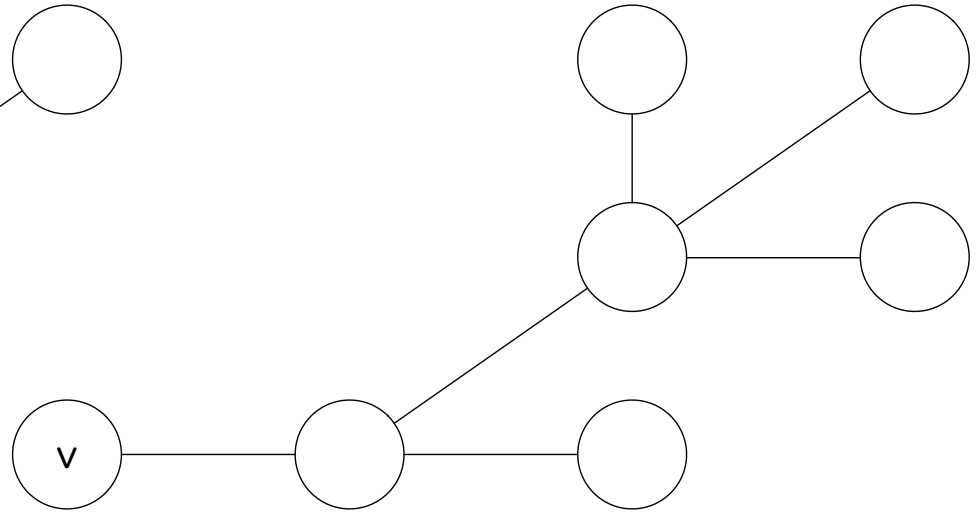
MSTs have optimal substructure

- Theorem – if T is an MST and we remove some edge (u,v):
  - T1 is an MST for the graph G1=(V1, E1) where G1 is the subgraph of G containing:
    - V1 = the vertices in T1.
    - E1 = (x,y) ϵ E : x, y ϵ V1.
  - T2 is an MST for the graph G2=(V2, E2) where G2 is the subgraph of G containing:
    - V2 = the vertices in T2.
    - E2 = (x,y) ϵ E : x, y ϵ V2.

...continued



T1 is an MST in this subgraph



T2 is an MST in this subgraph

Proof

- The weight of the whole MST is equal the weight of the edge removed plus the weight of the two subtrees.

$$w(T) = w(u, v) + w(T_1) + w(T_2)$$

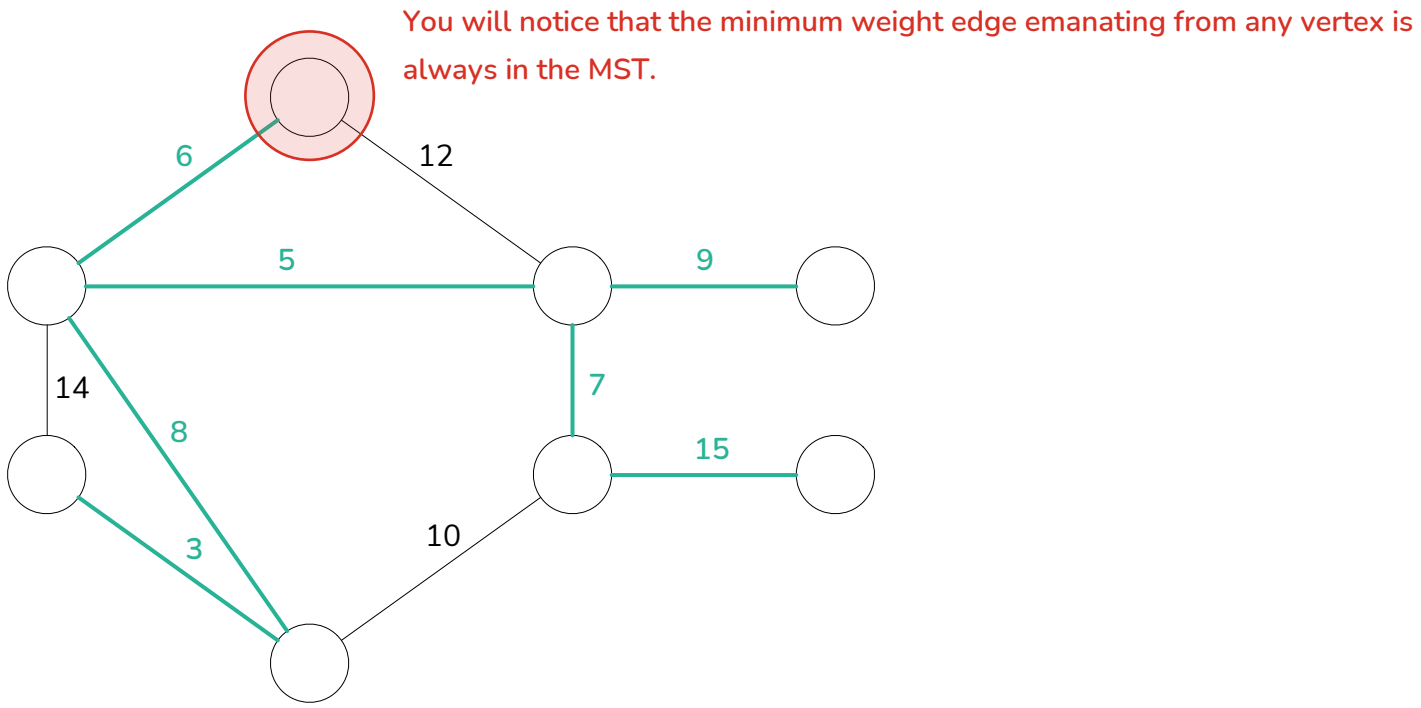
- Suppose some MST $T_1{}'$ existed that is lighter than T_1 for G_1 then:
  - T' is a spanning tree containing the edges: $\{u, v\} \cup T_1' \cup T_2$
  - Where T' is a lighter MST than T. This is a contradiction as this would imply that T was not an MST.
- Same applies when arguing for T2'.

Theorem

- Let T be the MST of $G = (V, E)$.
- Let $A \subseteq V$.
- Suppose $\{u, v\} \in E$ is the least-weight edge that connects A to $V - A$.
- Then $\{u, v\} \in T$. In other words, $\{u, v\}$ belongs to the MST T.

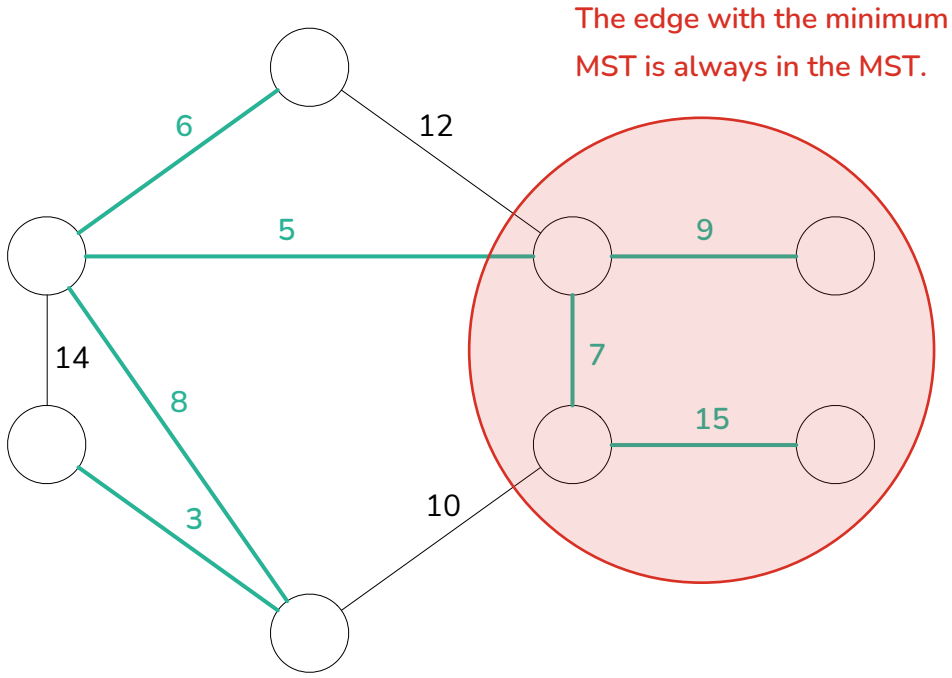
Illustration

- To illustrate, let's pick any single vertex:



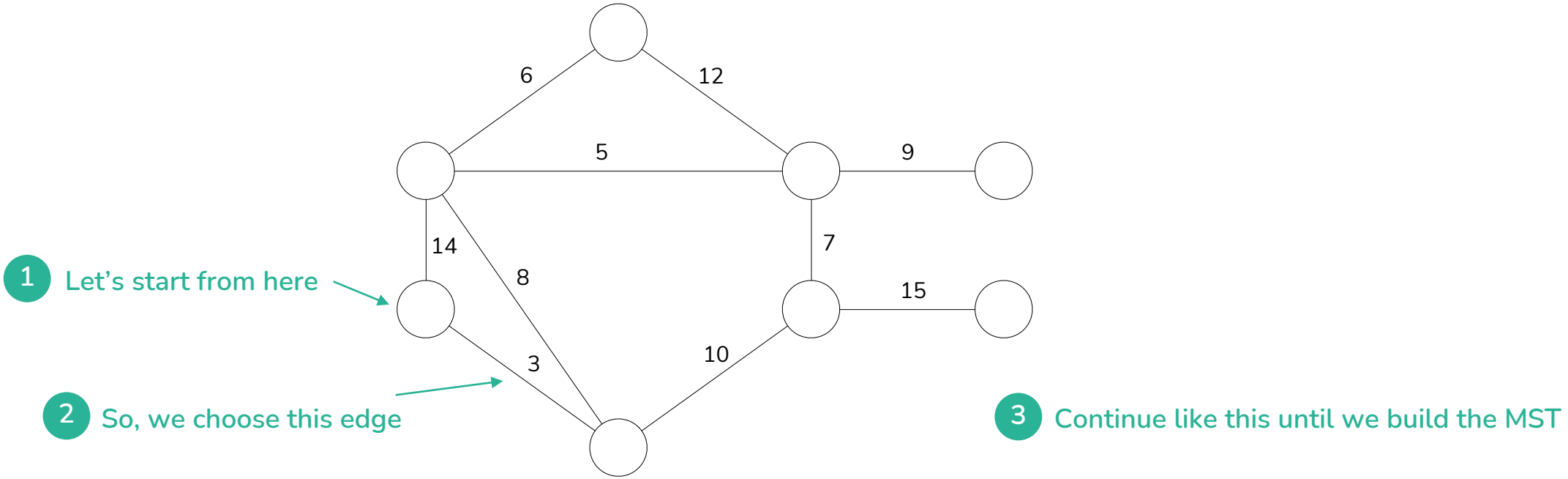
Illustration

- More than one vertex.



What does this mean?

- If I repeatedly connect a subtree of the MST to the rest of the graph and always pick the smallest weight edge to do so, I will end up with the MST for the graph.
- Try this for the following:



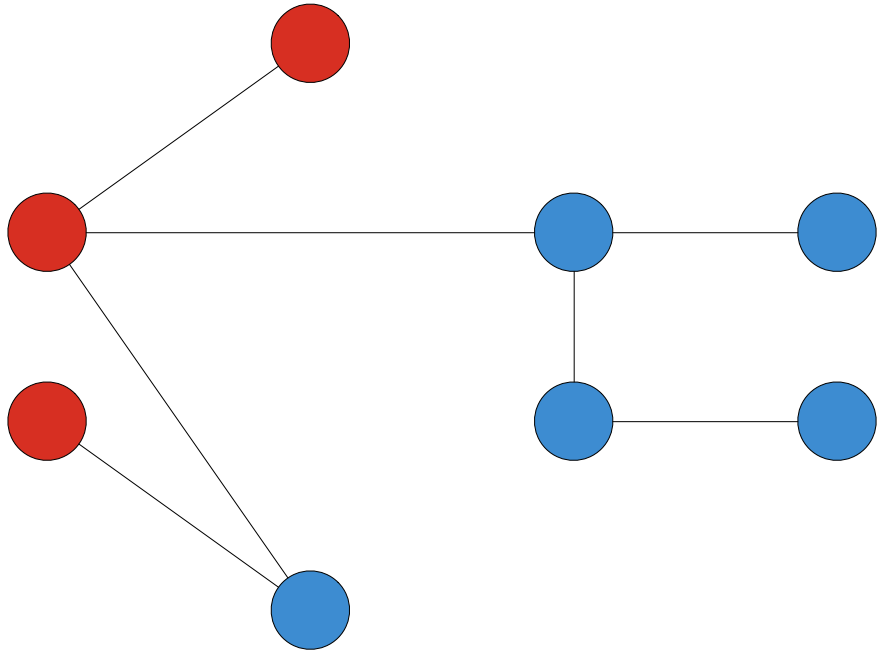
Proof

- Suppose I have this MST…

Our theorem:

- Let T be the MST of $G = (V, E)$.
- Let $A \subseteq V$.
- Suppose $\{u, v\} \in E$ is the least-weight edge that connects A to $V - A$.
- Then $\{u, v\} \in T$. In other words, $\{u, v\}$ belongs to the MST T.

- $\in A$
- $\in V - A$



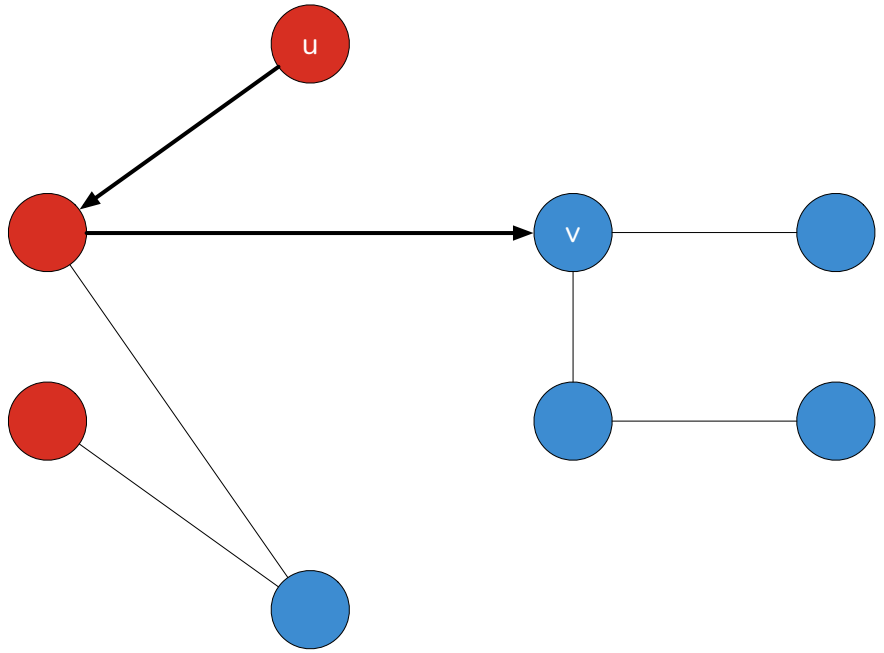
Proof

- There is a unique simple path from u to v...

Our theorem:

- Let T be the MST of $G = (V, E)$.
- Let $A \subseteq V$.
- Suppose $\{u, v\} \in E$ is the least-weight edge that connects A to $V - A$.
- Then $\{u, v\} \in T$. In other words, $\{u, v\}$ belongs to the MST T.

- $\in A$
- $\in V - A$



This ‘unique simple path’ property is true because we are dealing with a tree.

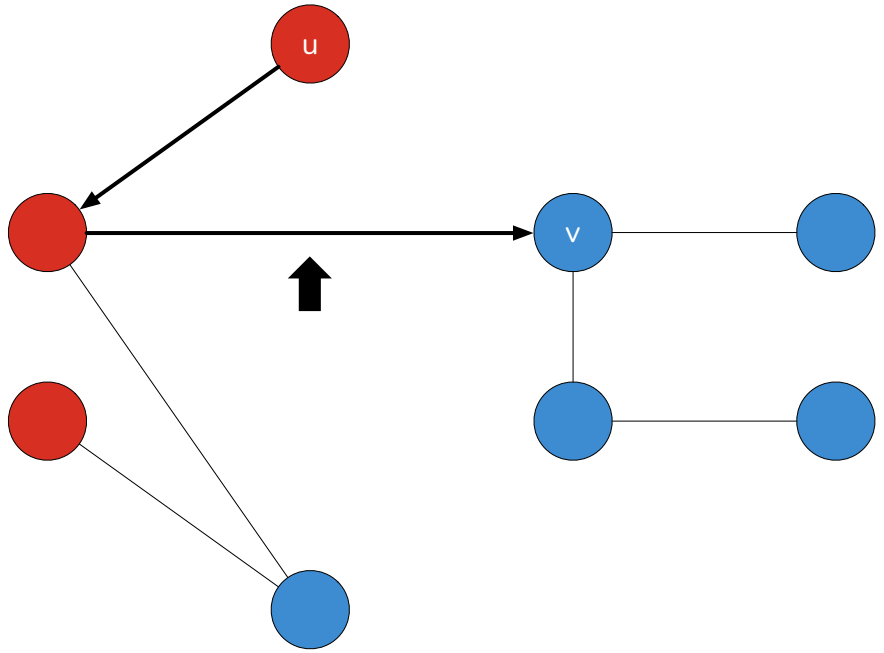
Proof

- At some point, the path will transition from A into (V-A)…

Our theorem:

- Let T be the MST of $G = (V, E)$.
- Let $A \subseteq V$.
- Suppose $\{u, v\} \in E$ is the least-weight edge that connects A to $V - A$.
- Then $\{u, v\} \in T$. In other words, $\{u, v\}$ belongs to the MST T.

- $\in A$
- $\in V - A$

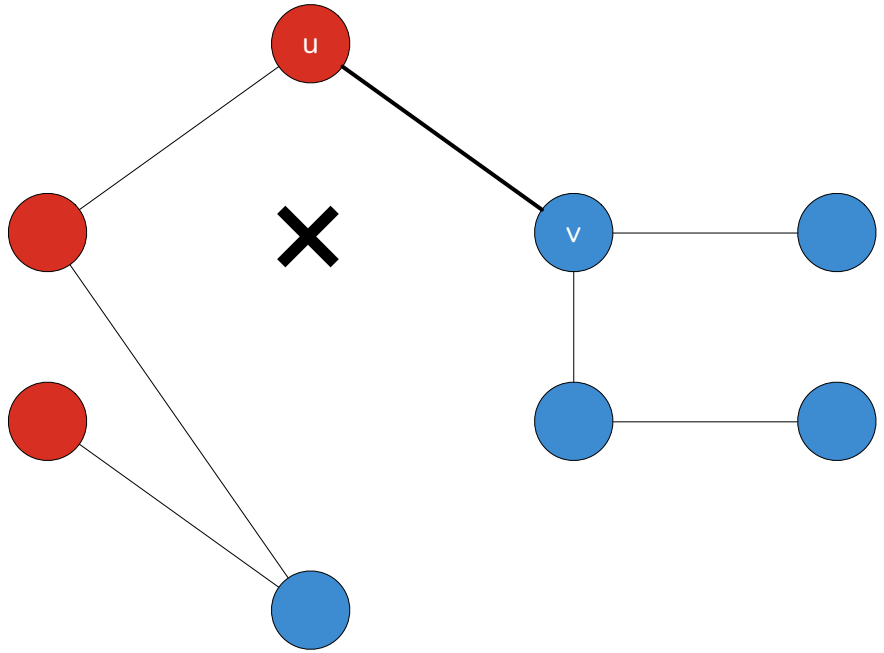


Obviously because {u,v} connects A and (V-A)

Proof

- Assume that {u,v} ϵ T.
- And swap it with the edge in the path that transitions from A to (V-A)…

- $\in A$
- $\in V - A$



Our theorem:

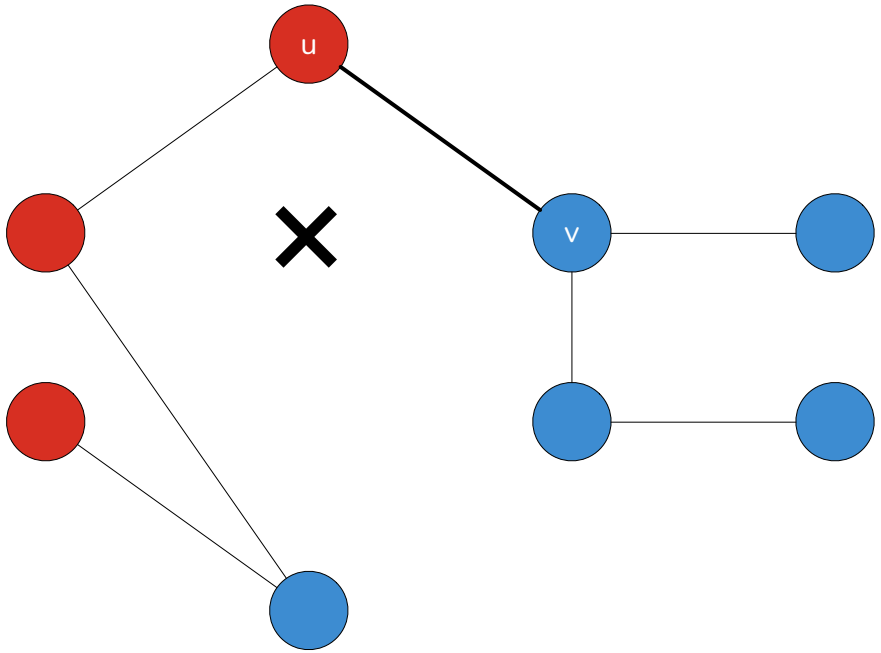
- Let T be the MST of $G = (V, E)$.
- Let $A \subseteq V$.
- Suppose $\{u, v\} \in E$ is the least-weight edge that connects A to $V - A$.
- Then $\{u, v\} \in T$. In other words, $\{u, v\}$ belongs to the MST T.

Proof

- If according to the requirement that the new edge is the lightest one connecting A and (V-A), then it is lighter than the edge we just removed.
- This is a contradiction because this would be a lighter tree than my starting one (which wouldn't have been an MST).

Our theorem:

- Let T be the MST of $G = (V, E)$.
- Let $A \subseteq V$.
- Suppose $\{u, v\} \in E$ is the least-weight edge that connects A to $V - A$.
- Then $\{u, v\} \in T$. In other words, $\{u, v\}$ belongs to the MST T.



Prim’s Algorithm

- Create a priority queue Q that contains the vertices (V-A), where:
  - The weight of a node in Q is the lightest node going to a vertex in A.
  - Initially A is empty.
  - The weight of each node in Q will be ∞. (A is empty so the lightest weight going to A is ∞).
- So far:
  - Q contains all of V (V-A actually, but A is empty).
  - All weights of the nodes in Q are ∞.
- Pick any node in Q and set its weight to zero.

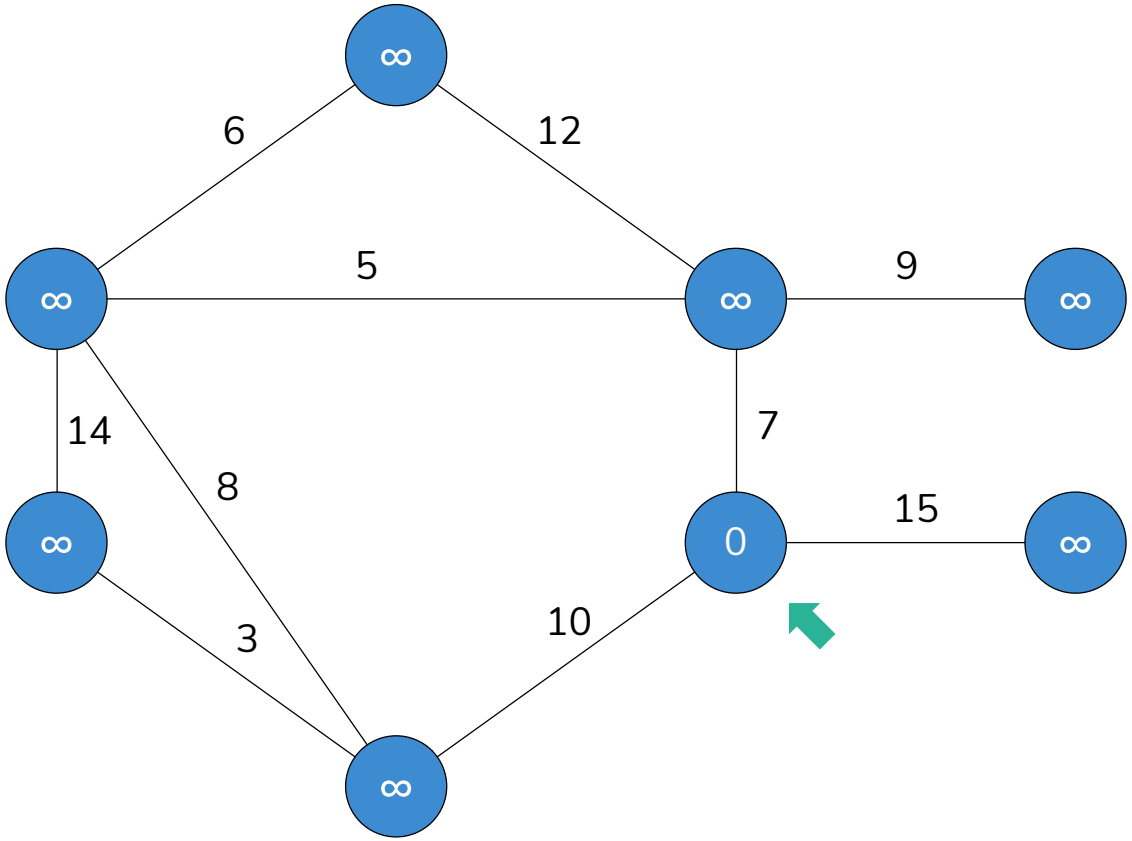
...continued

Initialising the algorithm...

```
Q = V          // Q is a priority queue
Key[v] = ∞     // ∀v ∈ V
Key[s] = 0     // for any s ∈ V
```

...continued

- $\in A$
- $\in V - A$



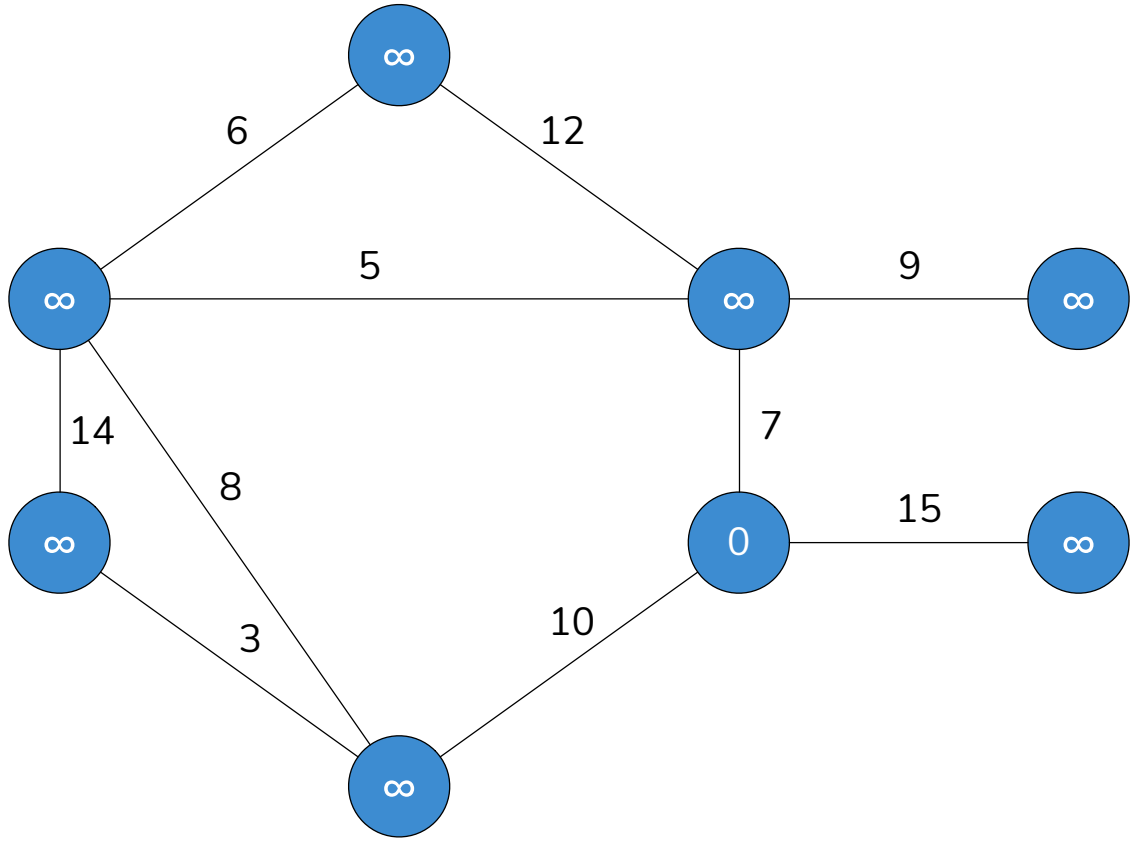
...continued

```
while Q ≠ empty {
  u = DequeueMin(Q) // Get the min (lightest) element.

  for each v ∈ Adjacent[u] {
    if v ∈ Q and w(u,v) < Key[v] then {
      Key[v] = w(u,v)
      Parent[v] = u
    }
  }
}
```

Start: initialise

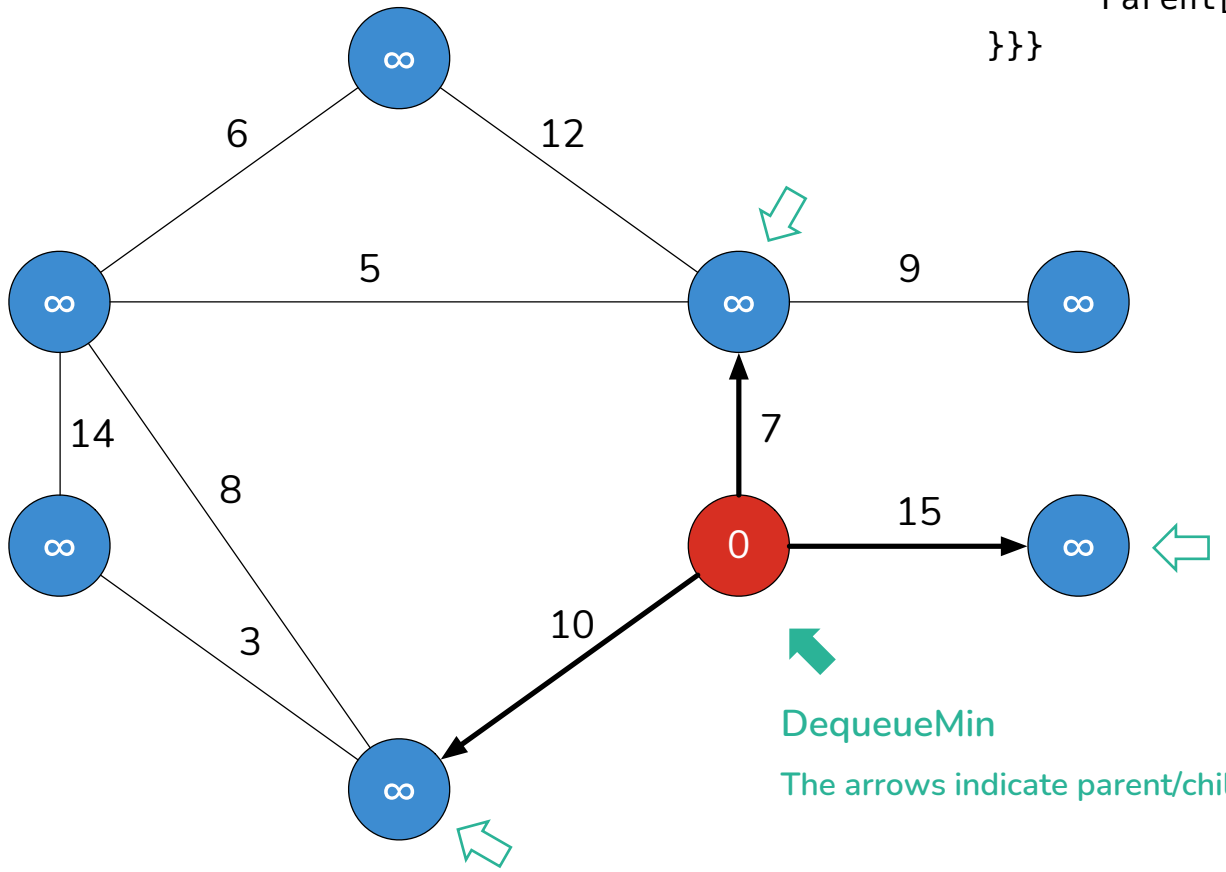
- $\in A$
- $\in V - A$



While queue is not empty

```
while Q ≠ empty {
  u = DequeueMin(Q)
  for each v ∈ Adjacent[u] {
    if v ∈ Q and w(u,v) < Key[v] then {
      Key[v] = w(u,v)
      Parent[v] = u
}}}
```

- $\in A$
- $\in V - A$



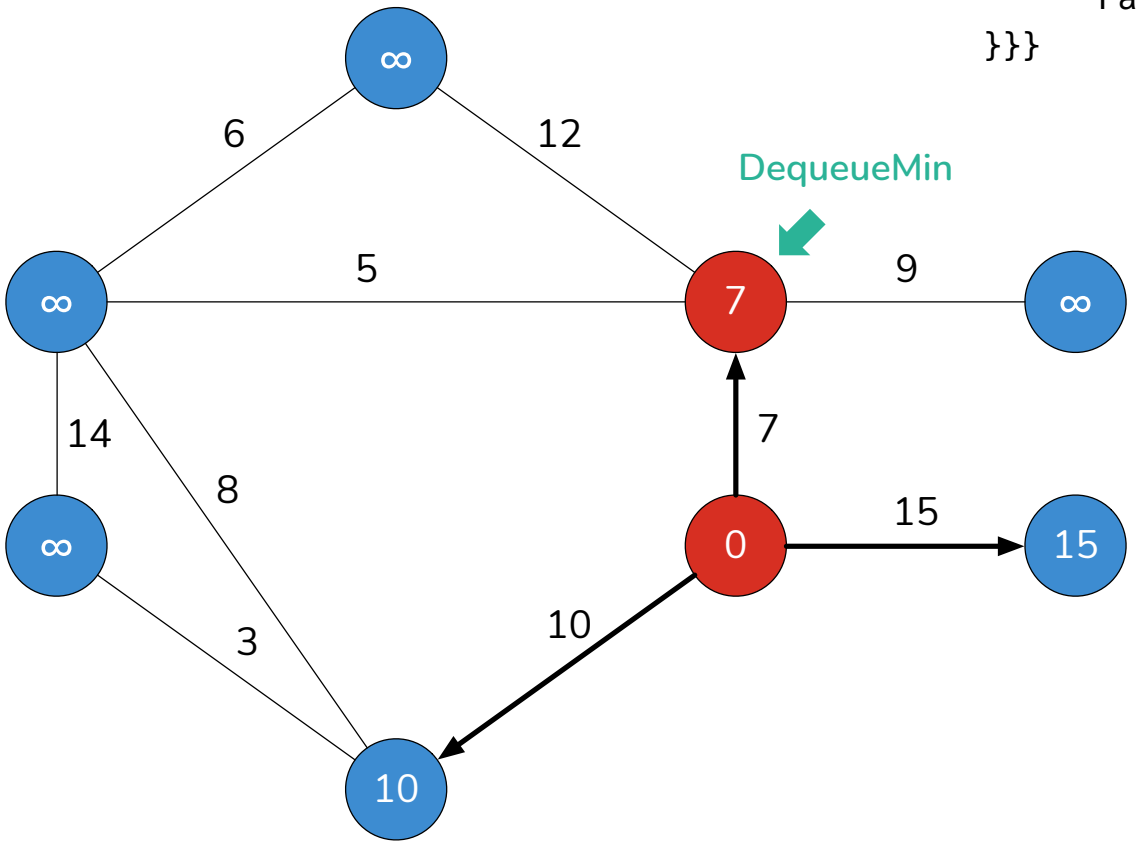
DequeueMin

The arrows indicate parent/child relationships in the tree.

...continued

```
while Q ≠ empty {
  u = DequeueMin(Q)
  for each v ∈ Adjacent[u] {
    if v ∈ Q and w(u,v) < Key[v] then {
      Key[v] = w(u,v)
      Parent[v] = u
}}}
```

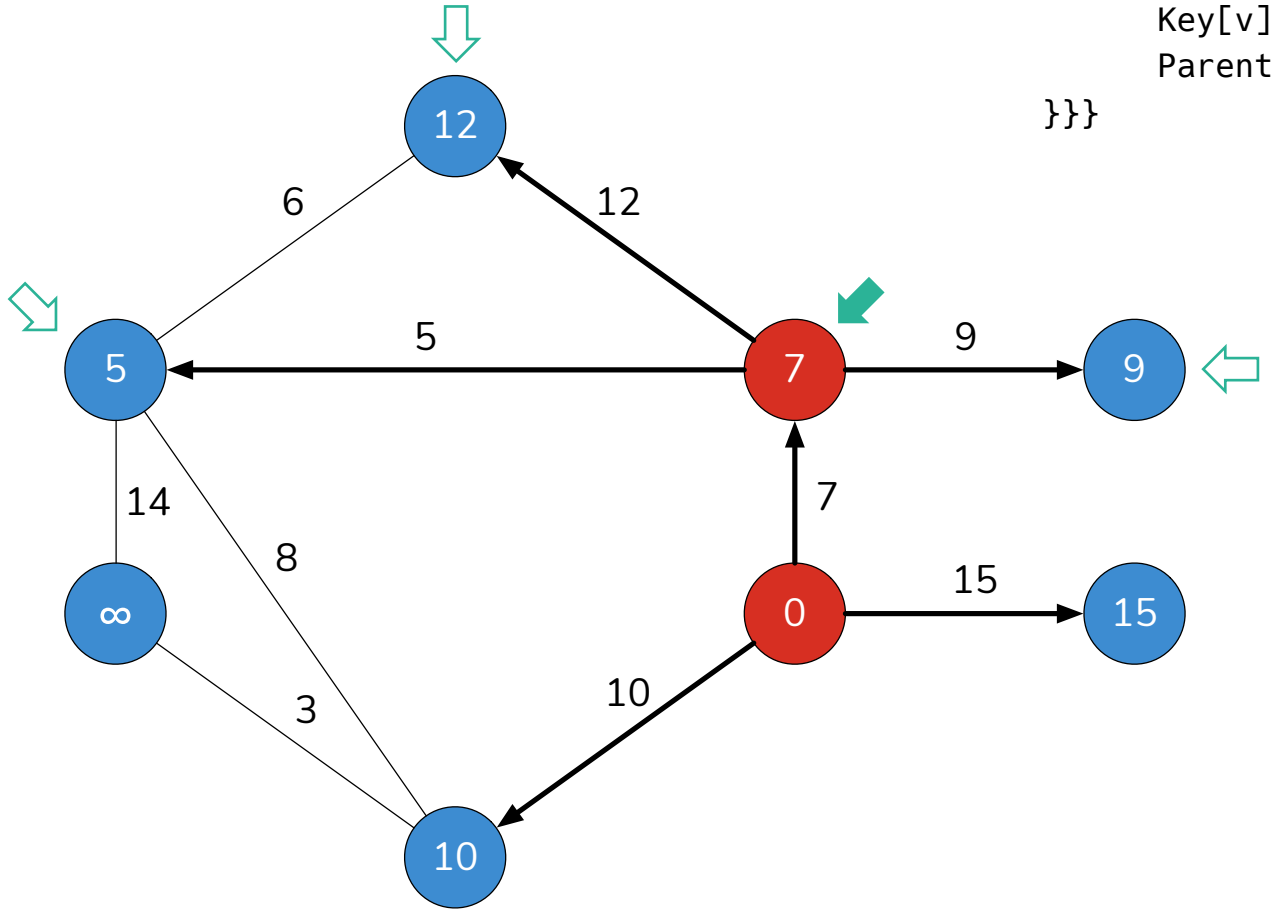
- $\in A$
- $\in V - A$



...continued

```
while Q ≠ empty {
  u = DequeueMin(Q)
  for each v ∈ Adjacent[u] {
    if v ∈ Q and w(u,v) < Key[v] then {
      Key[v] = w(u,v)
      Parent[v] = u
}}}
```

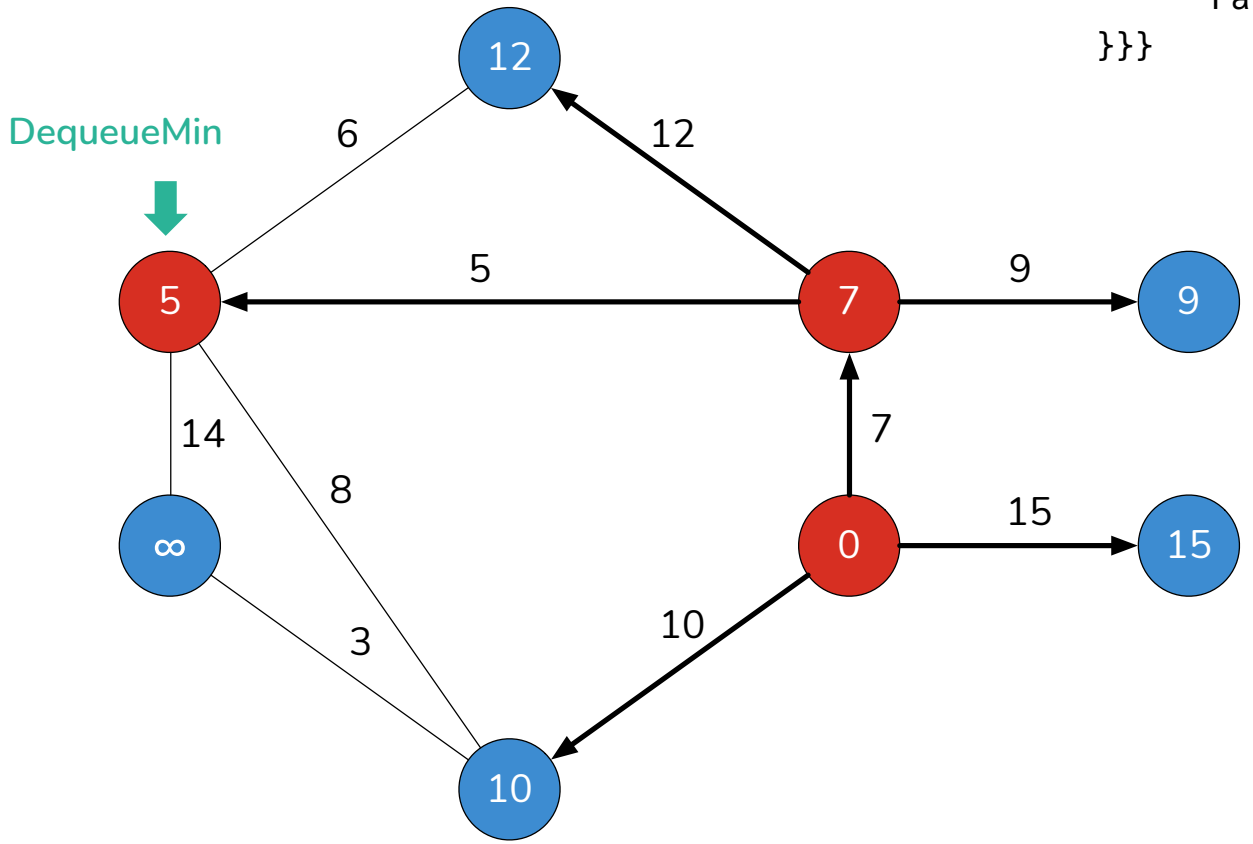
- $\in A$
- $\in V - A$



...continued

```
while Q ≠ empty {
  u = DequeueMin(Q)
  for each v ∈ Adjacent[u] {
    if v ∈ Q and w(u,v) < Key[v] then {
      Key[v] = w(u,v)
      Parent[v] = u
}}}
```

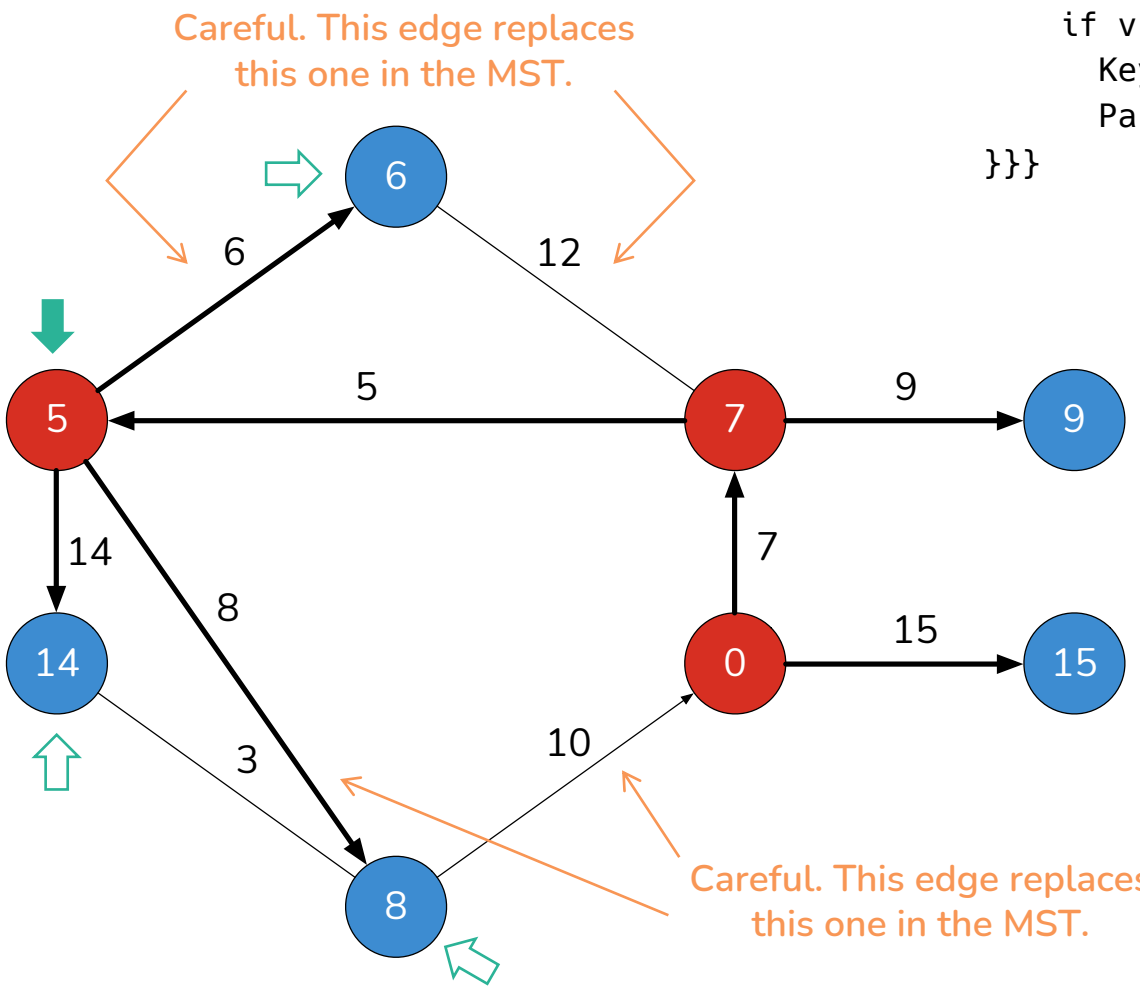
- $\in A$
- $\in V - A$



...continued

```
while Q ≠ empty {
  u = DequeueMin(Q)
  for each v ∈ Adjacent[u] {
    if v ∈ Q and w(u,v) < Key[v] then {
      Key[v] = w(u,v)
      Parent[v] = u
}}}
```

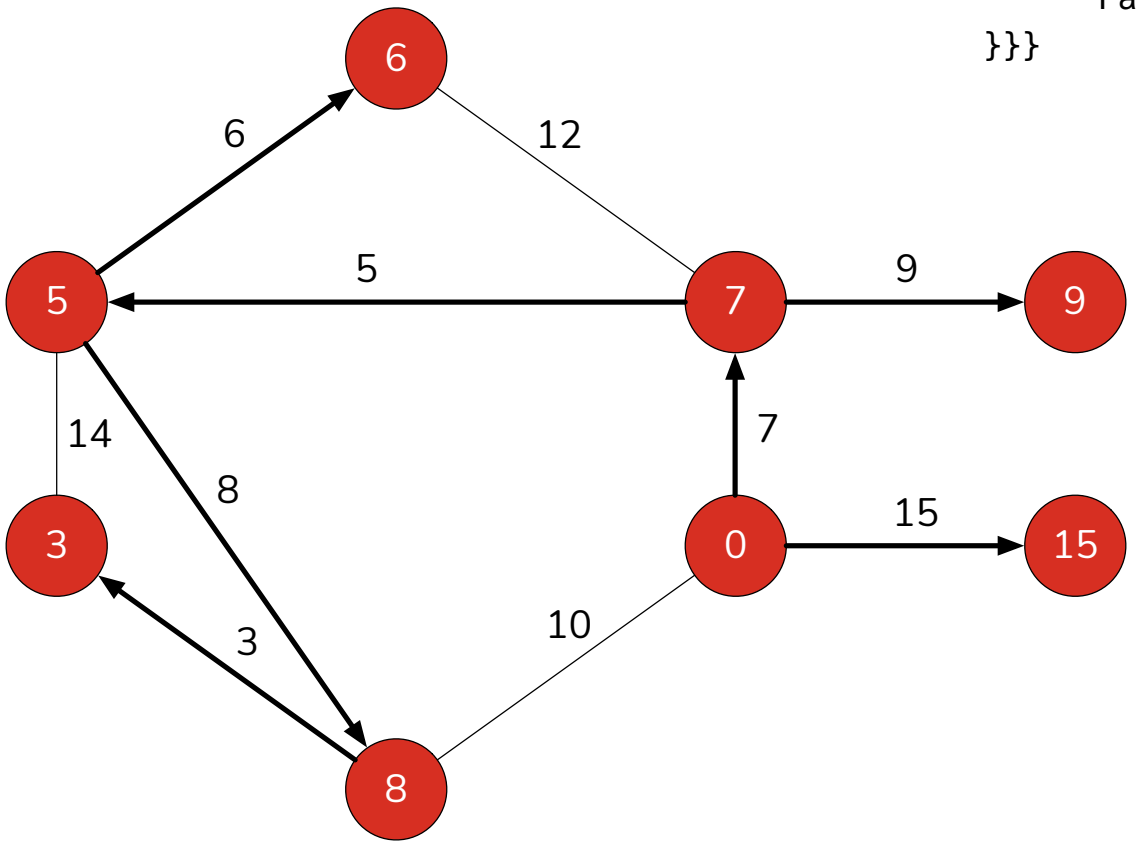
- $\in A$
- $\in V - A$



...continued

- And so on...

- $\in A$
- $\in V - A$




```
while Q ≠ empty {
  u = DequeueMin(Q)
  for each v ∈ Adjacent[u] {
    if v ∈ Q and w(u,v) < Key[v] then {
      Key[v] = w(u,v)
      Parent[v] = u
}}}
```

Analysis

```
Q = A
Key[v] = ∞
Key[s] = 0
```

Linear time.

```
while Q ≠ empty {
  u = DequeueMin(Q)
  for each v ∈ Adjacent[u] {
        if v ∈ Q and w(u,v) < Key[v] then {
              Key[v] = w(u,v)
              Parent[v] = u
        }
  }
}
```

Degree(u) times.

|V| times.

...continued

- So, we have |V| DequeueMins.
- And Degree(u) UpdateKeys for |V| times giving O(E) UpdateKeys.
- Total running time then is:
  - O(V)×TimeTo(DequeueMin) + O(E)×TimeTo(UpdateKey).
  - TimeTo() depends on the data structure I use to implement the Queue.

Remember the handshaking lemma:

$$\sum_{v \in V} \text{Degree}(v) = 2 \times |E|$$

Q Data Structure	DequeueMin	UpdateKey	Total
Unsorted Array	O(V)	O(1)	$O(V^2) + O(E) = O(V^2)$
Min Heap	$O(\log_2 V)$	$O(\log_2 V)$	$O(V.\log_2(V) + O(E.\log_2(V)) = O(E.\log_2(V))$

Note: if graph is dense, E approaches V^2 so Unsorted Array is better. If graph is sparse, Min Heap is better.

Further reading

- These notes should be supplemented by:
  - Introduction to Algorithms (Clifford Stein, Thomas H Cormen, Ronald L Rivest, Charles E Leiserson – MIT Press)
  - Also see Eric Demaine: http://videolectures.net/mit6046jf05_leiserson_lec16/
  - Also see previous material on graphs.

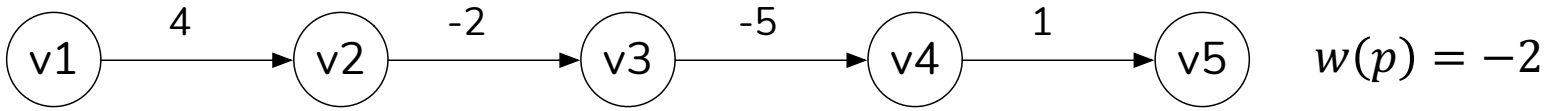
Shortest Paths

Kristian Guillaumier

Shortest paths

- Digraph $G = (V, E)$
- Edge weight function: $w: E \to \mathcal{R}$
- Weight of a path $v_1 \to v_2 \to \cdots \to v_k$ is given by:

$$w(p) = \sum_{i=1}^{k-1} w(v_i, v_{i+1})$$

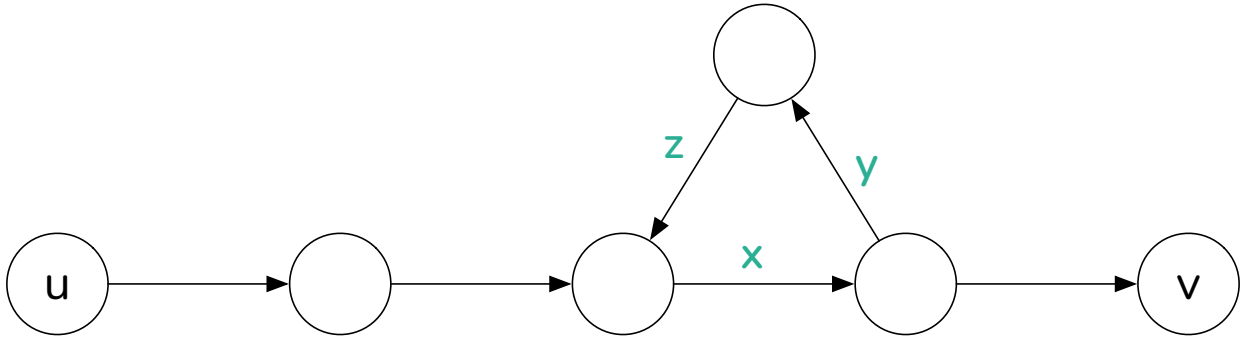


...continued

- A shortest path from u to v is a path having the minimum weight from u to v.
- The shortest path weight δ from u to v is:
  - $\delta(u, v) = \min\big(w(p)\big)$
  - Where p is a path from u to v.
  - Note that δ is the weight of the shortest path, not the shortest path itself.
- If graph is disconnected and there is no path from u to v, then $\delta(u, v) = \infty$.

...continued

- Note: we may encounter a problem if the edges can have negative weights and there are cycles...



If x+y+z is negative, then the length of the path from u to v can be infinitely small. That is: $w(p) = -\infty$

Optimal substructure

Claim: A subpath of a shortest path is a shortest path.

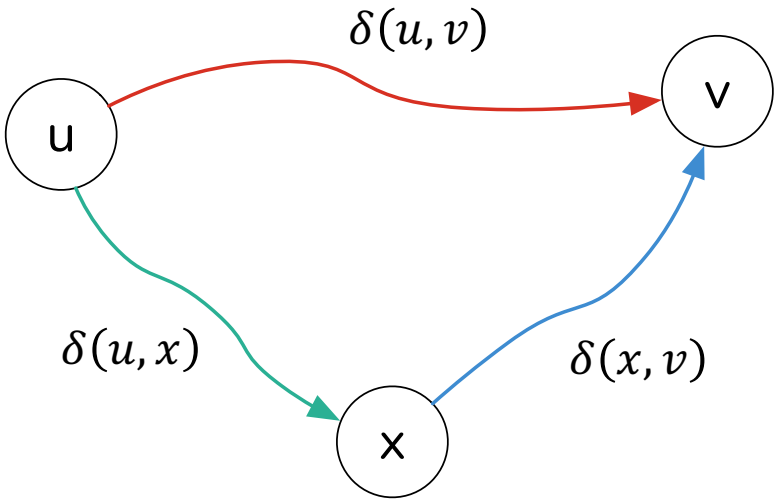
- Assume I have this shortest path:

- Suppose that $x \rightarrow y$ is not a shortest path, and there exists another one:

- This means that there is a shorter path from u to v. Contradiction.

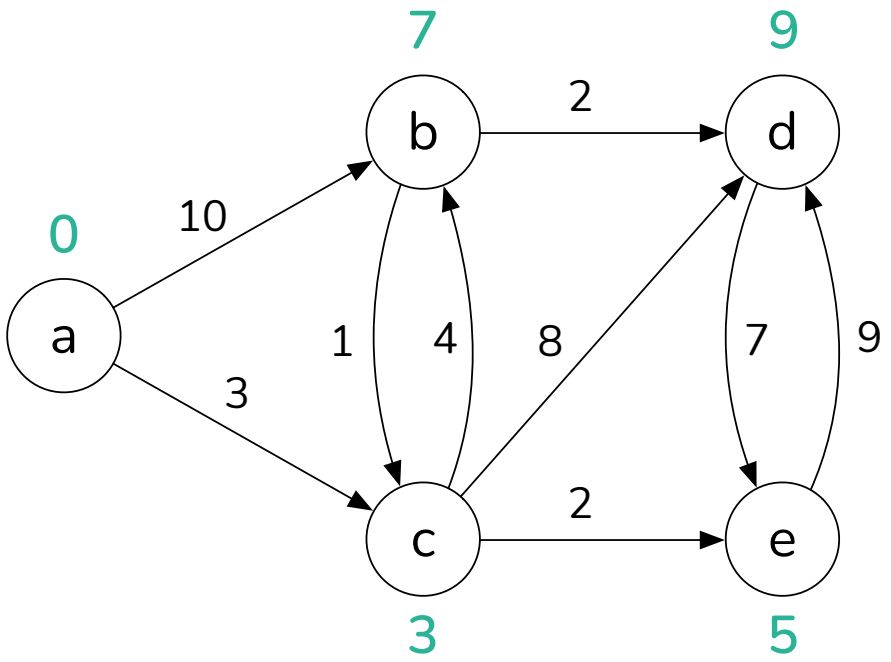
Triangle inequality

- For all $u, v, x \in V$
- $\delta(u, v) \leq \delta(u, x) + \delta(x, v)$



Single source shortest paths

- Given a source vertex $s \in V$ find the shortest path weight $\delta(s, v)$ to every vertex $v \in V$.
- Requirement: all weights are ≥ 0.



Green labels show the cost of the shortest path from the starting state to a vertex.

This is what an SSSP algorithm such as Dijkstra will find.

Example from Demaine

SSSP method

- Maintain a set S of vertices whose shortest path weight from some vertex s is known.
  - Note: the shortest path weight from s to s is known: $\delta(s, s) = 0$.
  - So, the vertex s is in the set S.
- For every vertex $v \in V - S$:
  - Add to S the vertex whose distance from s is minimal.
  - Update distance estimates for vertices adjacent to v.

Dijkstra's algorithm

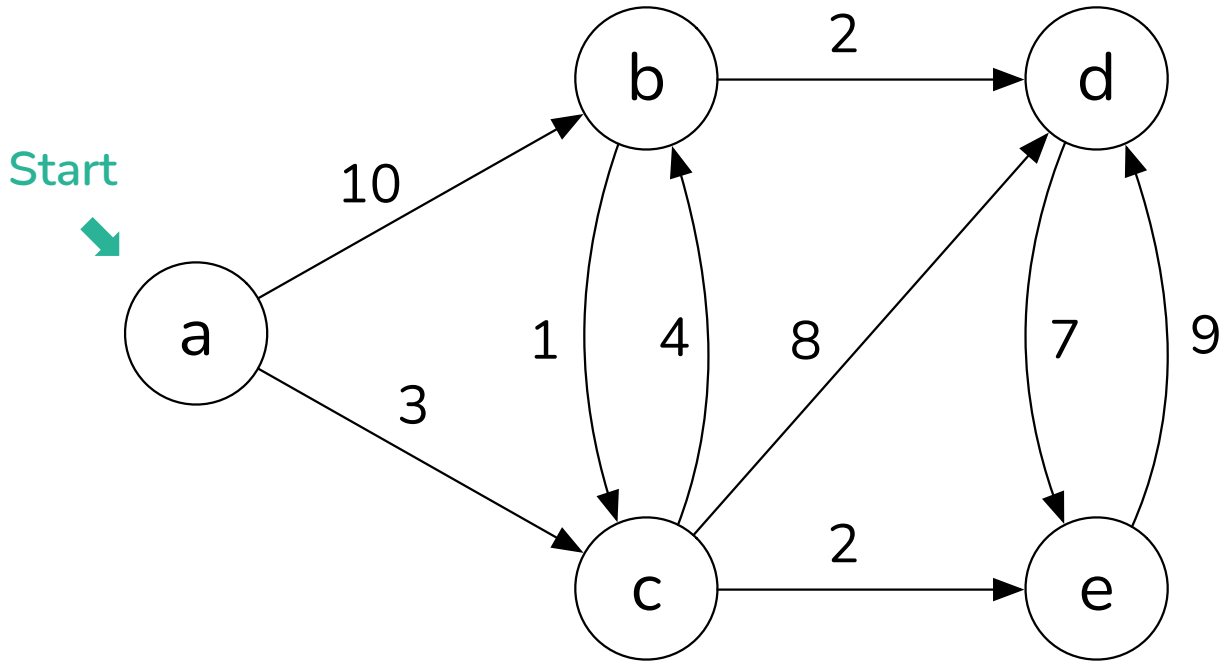
```
d[s] = 0
for each v in V-{s}
        d[v] = ∞
S = ∅
Q = V = {s}
while Q ≠ ∅
        u = DequeueMin(Q)
        S = S ∪ {u}
        for each v ∈ Adj[u]
                if d[v] > d[u] + w(u, v)
                        d[v] = d[u] + w(u, v)
```

d is an array indexed by vertex that keeps the estimate distance from s to that vertex.

Q is a priority queue of vertices indexed by distance estimate (i.e., distance is the key).

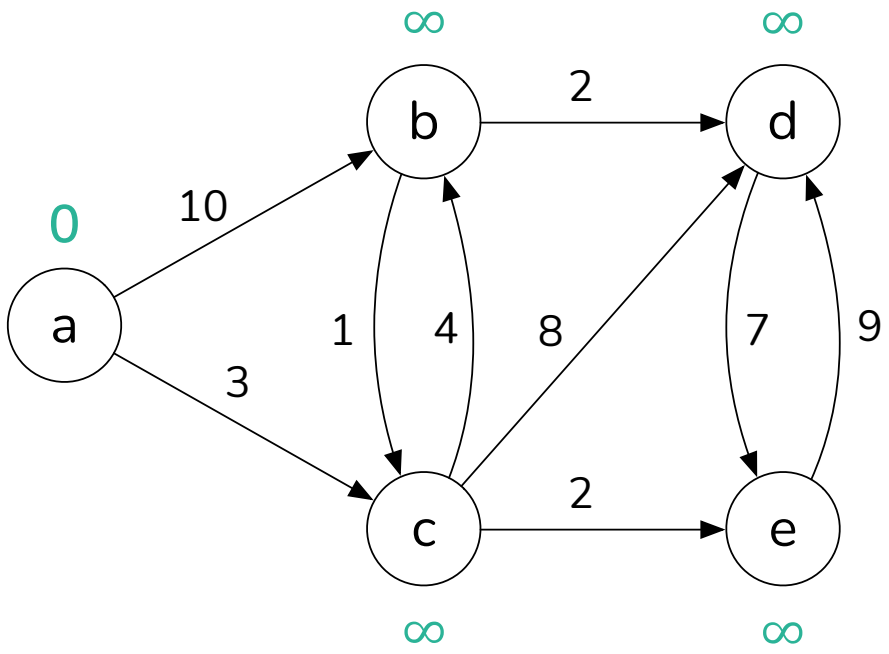
This is called the 'relaxation step'.

Dijkstra’s algorithm



Example from Demaine

Initialise

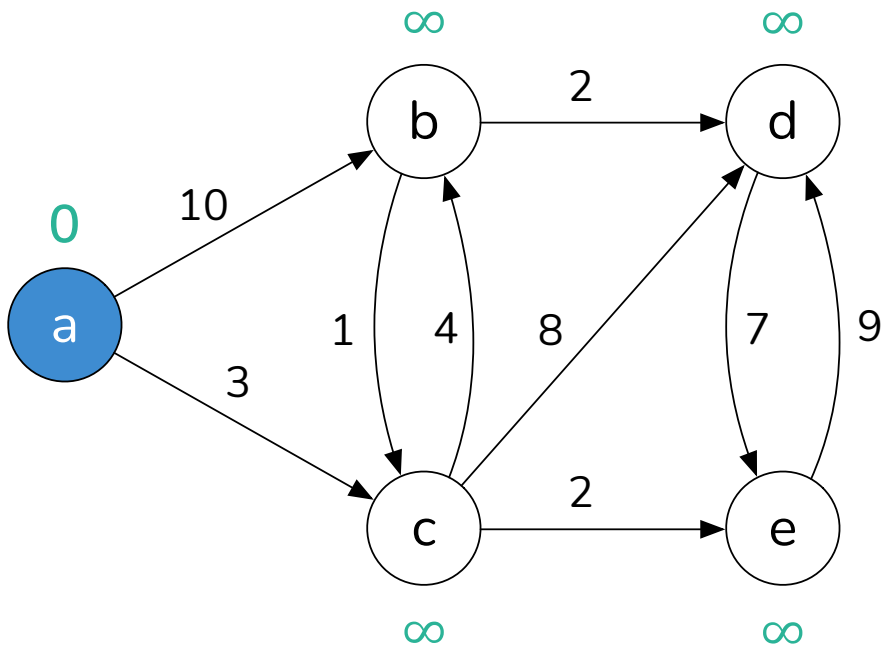


Queue				
a	b	c	d	e
0	∞	∞	∞	∞

Set
S={ }

```
d[s] = 0
for each v in V-{s}
        d[v] = ∞
S = ∅
Q = V
```

Dequeue minimum

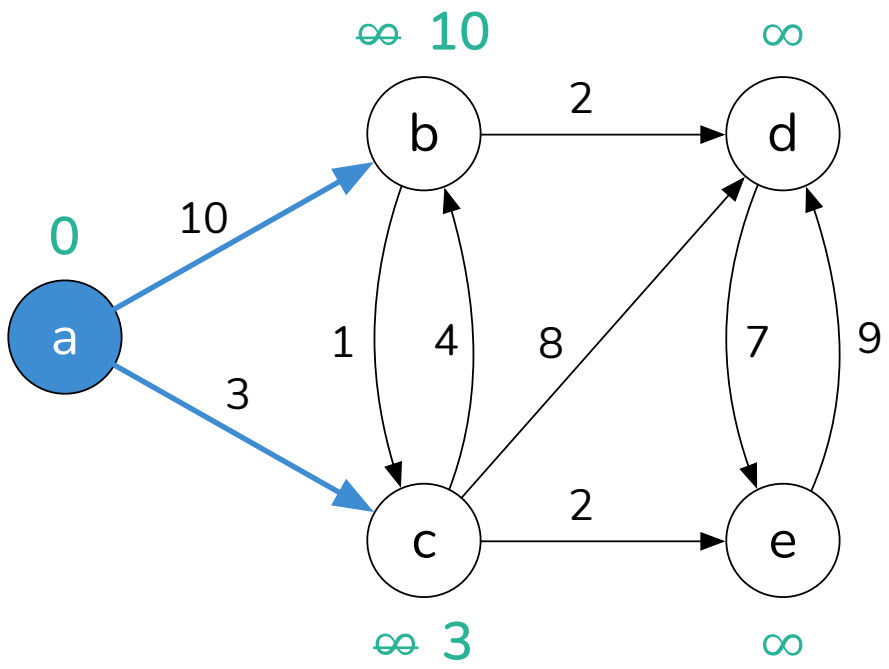


Queue				
~~a~~	b	c	d	e
~~0~~	∞	∞	∞	∞

Set
S={a}

```
u = DequeueMin(Q)
S = S ∈ {u}
for each  v ∈ Adj[u]
  if d[v] > d[u] + w(u, v)
    d[v] = d[u] + w(u, v)
```

Fix edges adjacent to ‘a’

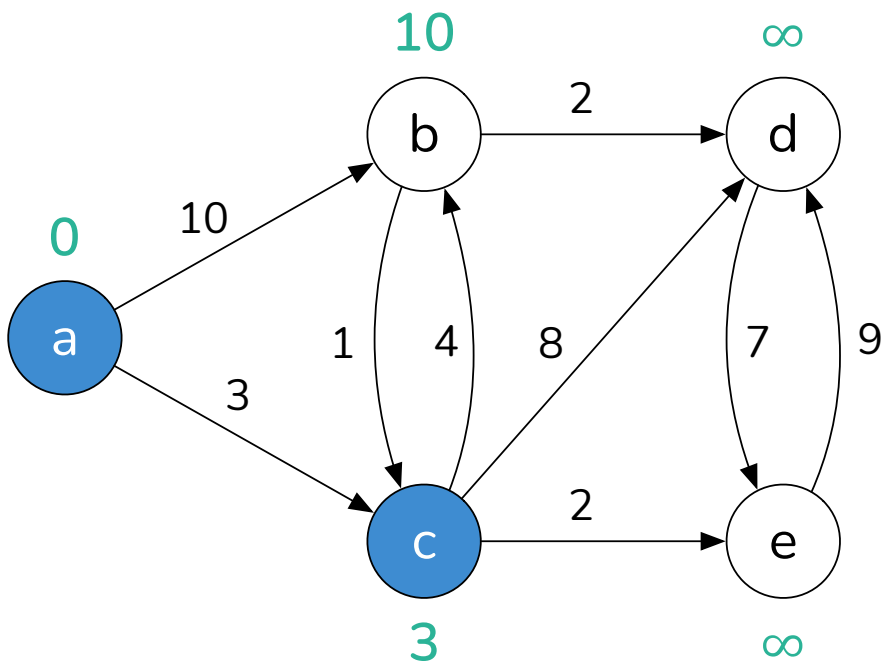


Queue				
~~a~~	b	c	d	e
~~0~~	∞	∞	∞	∞
	10	3	∞	∞

Set
S={a}

```
u = DequeueMin(Q)
S = S ∈ {u}
for each  v ∈ Adj[u]
  if d[v] > d[u] + w(u, v)
    d[v] = d[u] + w(u, v)
```

Dequeue minimum

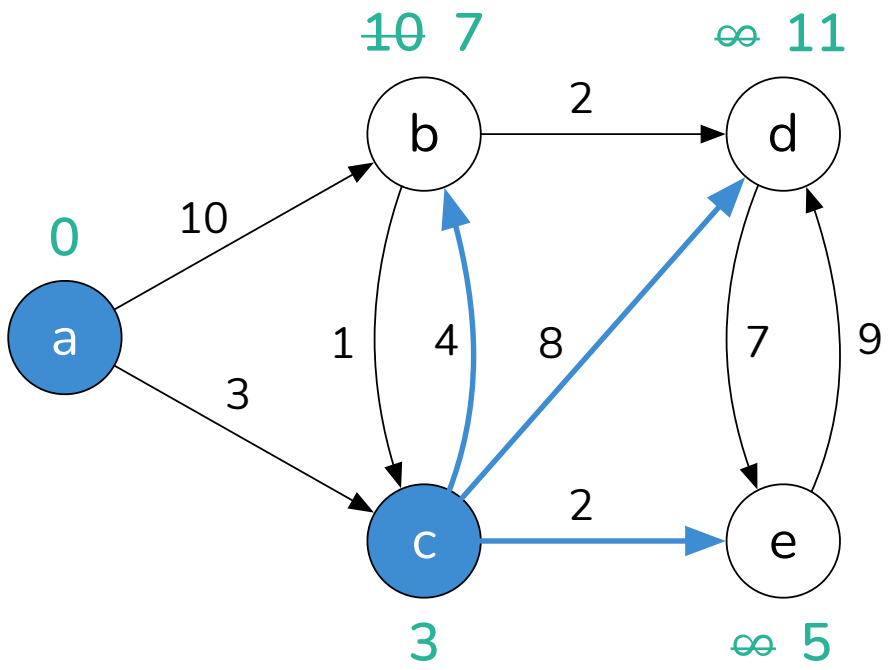


Queue				
~~a~~	b	~~c~~	d	e
~~0~~	∞	∞	∞	∞
	10	~~3~~	∞	∞

Set
S={a,c}

```
u = DequeueMin(Q)
S = S ∈ {u}
for each  v ∈ Adj[u]
  if d[v] > d[u] + w(u, v)
    d[v] = d[u] + w(u, v)
```

Fix edges adjacent to ‘a’

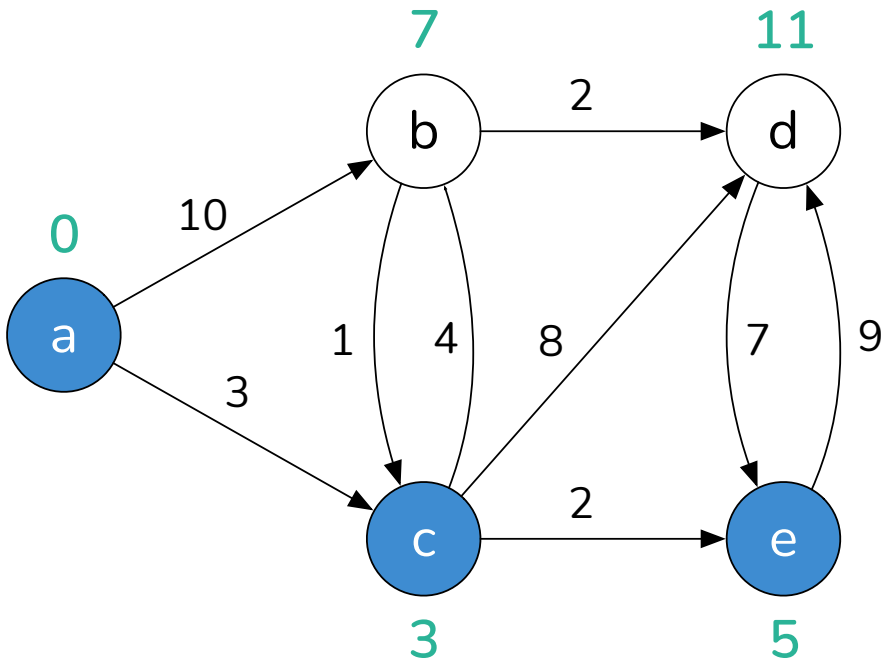


Queue				
~~a~~	b	~~c~~	d	e
~~0~~	∞	∞	∞	∞
	10	~~3~~	∞	∞
	7		11	5

Set
S={a,c}

```
u = DequeueMin(Q)
S = S ∈ {u}
for each  v ∈ Adj[u]
  if d[v] > d[u] + w(u, v)
    d[v] = d[u] + w(u, v)
```

Dequeue minimum

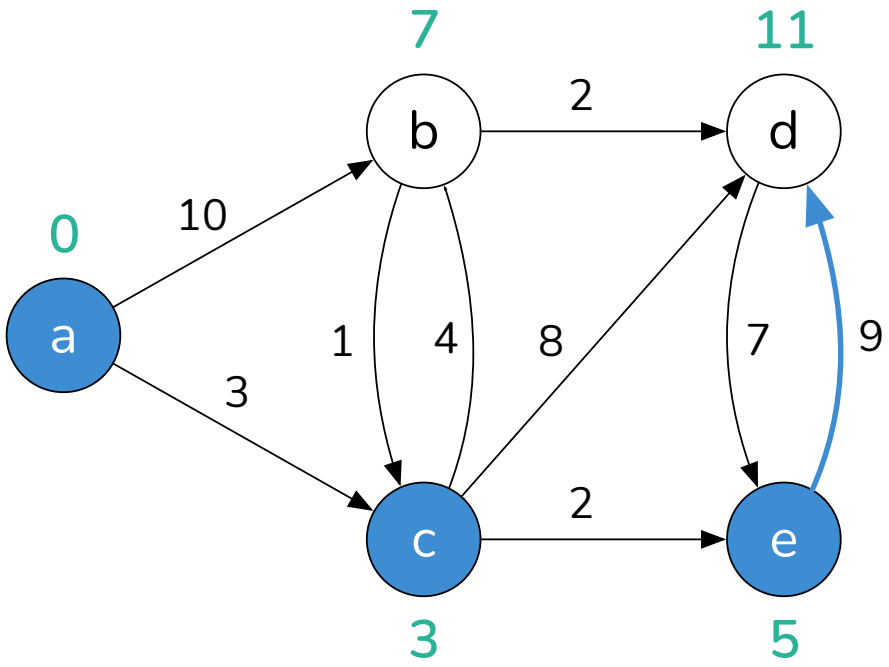


Queue				
~~a~~	b	~~c~~	d	~~e~~
~~0~~	∞	∞	∞	∞
	10	~~3~~	∞	∞
	7		11	~~5~~

Set
S={a,c,e}

```
u = DequeueMin(Q)
S = S ∈ {u}
for each  v ∈ Adj[u]
  if d[v] > d[u] + w(u, v)
    d[v] = d[u] + w(u, v)
```

Fix edges adjacent to ‘e’

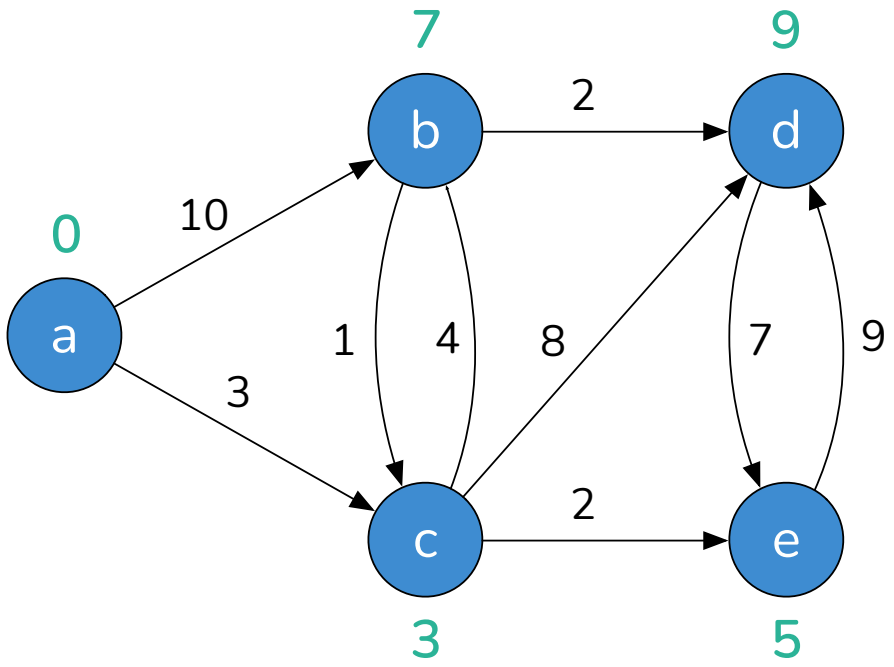


Queue				
~~a~~	b	~~e~~	d	~~e~~
~~0~~	∞	∞	∞	∞
	10	~~3~~	∞	∞
	7		11	~~5~~

Set
S={a,c,e}

```
u = DequeueMin(Q)
S = S ∈ {u}
for each  v ∈ Adj[u]
  if d[v] > d[u] + w(u, v)
    d[v] = d[u] + w(u, v)
```

And so on...



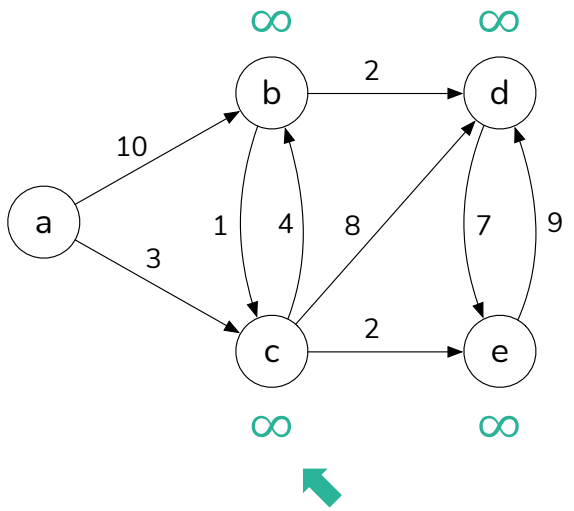
Queue				
~~a~~	~~b~~	~~c~~	~~d~~	~~e~~
~~0~~	∞	∞	∞	∞
	10	~~3~~	∞	∞
	7		11	~~5~~
	~~7~~		11	
			~~11~~	

Set
S={a,c,e,b,d}

Lemma 1

Initialising $d[s] = 0$ and $d[v] = \infty$ for all $v \in V - \{s\}$ means that $d[v] \geq \delta(s, v)$ for all $v \in V$ for any sequence of relaxation steps.

- This is the "overestimate" lemma.
- Base case: $d[s] = 0, d[v] = \infty$
  - Is $d[s] \geq \delta(s, s)$? Is $0 \geq 0$? Yes.
  - Is $d[v] \geq \delta(s, v)$? Is $\infty \geq$ any possible value? Yes.
  - Base case proven.



These values will always be overestimates (≥)

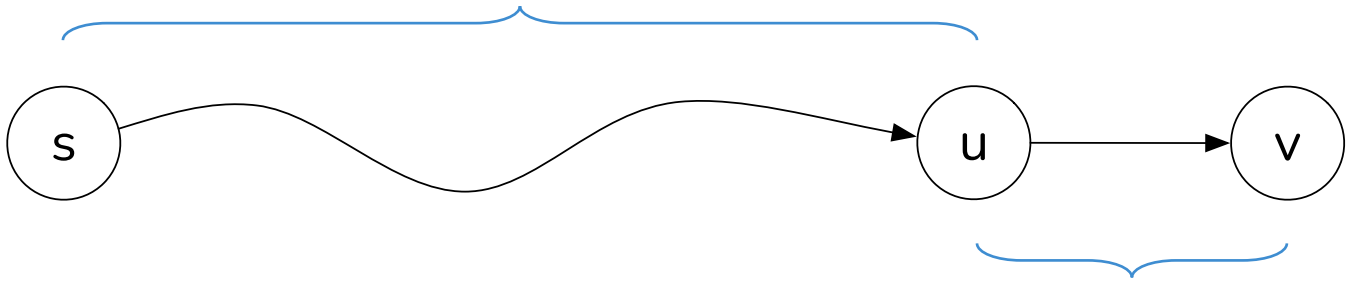
- Suppose that the lemma is not true.
- So, suppose that d[v] < δ(s, v). We are proving by induction so all other cases before d[v] did not fail.
- So, at some point we set d[v] = d[u] + w(u, v). ← Only part of the algorithm that can change d[v]
- So, we'd get d[u] + w(u, v) < δ(s, v). (a)
- But we know that d[u] >= δ(s, u) – remember that d[u] is before d[v] so by our induction hypothesis, it must be correct.
- So, d[u] + w(u, v) $\geq$ δ(s, u) + w(u, v). Note that I simply added a constant w(u, v) to both sides.
- w(u, v) is the weight of the edge u to v which is certainly $\geq$ the shortest path from u to v. The length of the shortest path from u to v is written as δ(u, v) and must then be $\leq$ w(u, v).
- So, d[u] + w(u, v) $\geq$ δ(s, u) + δ(u, v). By triangle inequality we get:
- d[u] + w(u, v) $\geq$ δ(s, v). (b)
- Contradiction. (a) (b)

Lemma 2

Suppose I have a shortest path…

$$s \to \cdots \to u \to v$$

if $d[u] = \delta(s,u)$ *and we relax edge* (u,v) *then* $d[v] = \delta(s,v)$.



Proving lemma 2

- Part 1:
  - $\delta(s, v) = w(s \to \cdots \to u) + w(u, v)$
  - By optimal substructure $w(s \to \cdots \to u) = \delta(s, u)$ so $\delta(s, v) = \delta(s, u) + w(u, v)$ - remember that, by assumption, $\delta(s, u) = d[u]$.
- Remember that relaxation can only decrease the values in d[ ]. The relaxation step is:

$$\texttt{if } d[v] > d[u] + w(u, v)$$
$$\texttt{then } d[v] = d[u] + w(u, v)$$

...continued

- Lemma 1 says that the values in d[ ] are always overestimates. So, for some v, the value in $d[v]$ is either already equal to $\delta(s, v)$ or it must be greater than $\delta(s, v)$.
- Case 1 – if $d[v]$ is equal to $\delta(s, v)$ then we won't 'pass' the `if` statement, and $d[v]$ will remain unchanged and equal to $\delta(s, v)$.

`if` $d[v] > d[u] + w(u, v)$

then $d[v] = d[u] + w(u, v)$

...continued

- Case 2 – $d[v]$ is greater than $\delta(s, v)$ so we will 'pass' the `if` statement.
  - Remember that in part 1, by optimal substructure, we know that $\delta(s, v) = \delta(s, u) + w(u, v) = d[u] + w(u, v)$.
  - If $d[v] > \delta(s, v)$ we will pass the `if` statement and set $d[v]$ to $d[u] + w(u, v)$ which is $\delta(s, v)$.
- Ready.

Proof of correctness

When Dijkstra terminates, $\boldsymbol{d[v] = \delta(s, v)}$ for all $\boldsymbol{v \in V}$.

- Recall that the algorithm does not change the value of $d[v]$ once we add it to the set S.
- So "all we have to do" is show that $d[v] = \delta(s, v)$ when v has been added to S.

...continued

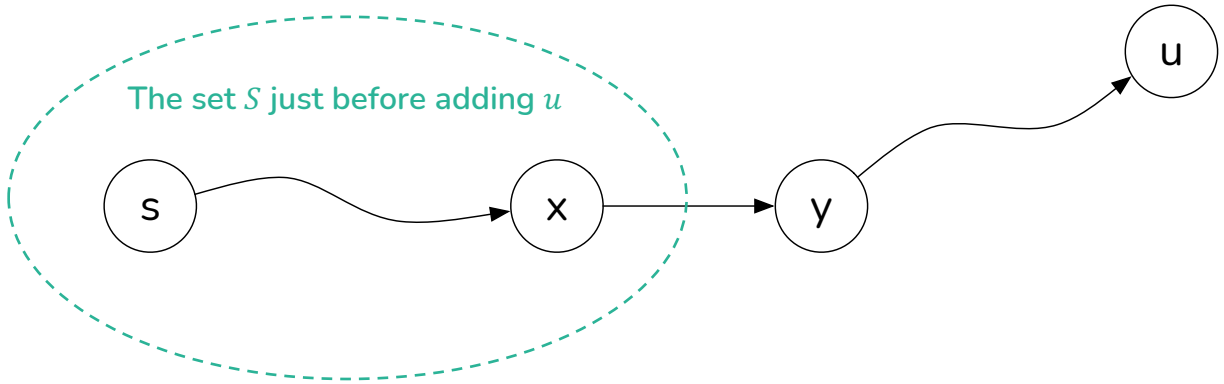
- We will prove by contradiction: suppose we are just about to add u to S and $d[u] \neq \delta(s, u)$.
- If it is not equal to, then by lemma 1 it must be greater:

$$d[u] > \delta(s, u) \quad \text{a}$$

- Now, let p be a shortest path from s to u.
- So, $w(p) = \delta(s, u)$.

...continued

- Remember that s is in S and u has not been added yet (we were just about to add it).
- All the values d[ ] in S are correct because they are in S.
- Adding $d[u]$ is going to be our first violation.
- Consider the first edge (x, y) where p exits S.



...continued

- $d[x] = \delta(s, x)$ is correct since it is in S.
- The algorithm says that after we put x in S we must relax all edges emanating from x. We will therefore relax the edge from x to y.
- By lemma 2, if $d[x]$ is the shortest path weight (and it is because its in S), if we relax the edge (x, y) then $d[y] = \delta(s, y)$.
- Now note that $\delta(s, y)$ can only be $\leq \delta(s, u)$.
- And my initial assumption is that $d[u]$ is strictly $> \delta(s, u)$. a
- So, organising everything we know so far, we get...
- $d[y] = \delta(s, y) \leq \delta(s, u) < d[u]$.

...continued

- Now, remember... which data structure are we using to storing the vertices outside of S? We are using a min-queue.
- This means that if we chose to add the vertex u in S and not the vertex y, it was because $d[u] \leq d[y]$.
- Arrange again...

$$\underbrace{d[u] \leq d[y] = \delta(s,y) \leq \delta(s,u)}_{} \underbrace{< d[u]}_{}$$

Contradiction

Analysis

```
d[s] = 0
for each v in V-{s}
        d[v] = ∞
S = ∅
Q = V
while Q ≠ ∅
        u = DequeueMin(Q)
        S = S ∪ {u}
        for each v ∈ Adj[u]
                if d[v] > d[u] + w(u, v)
                        d[v] = d[u] + w(u, v)
```

Linear time.

Degree(u) times.

|V| times.

...continued

- So, we have |V| DequeueMins.
- And Degree(u) UpdateKeys for |V| times giving O(E) UpdateKeys.
- Total running time then is:
  - O(V)×TimeTo(DequeueMin) + O(E)×TimeTo(UpdateKey).
  - TimeTo() depends on the data structure I use to implement the Queue.

Remember the handshaking lemma:

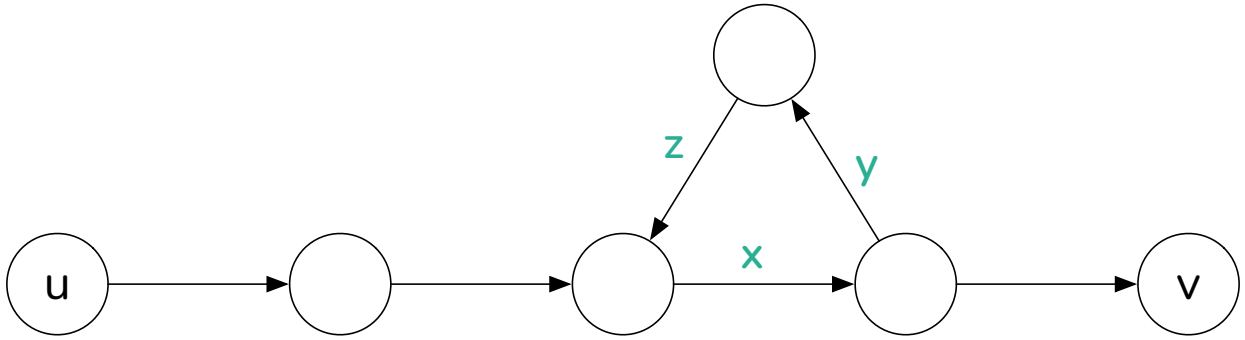
$$\sum_{v \in V} \text{Degree}(v) = 2 \times |E|$$

Q Data Structure	DequeueMin	UpdateKey	Total
Unsorted Array	O(V)	O(1)	$O(V^2) + O(E) = O(V^2)$
Min Heap	$O(\log_2 V)$	$O(\log_2 V)$	$O(V.\log_2(v) + O(E.\log_2(v)) = O(E.\log2(v))$

Note: if graph is dense, E approaches V^2 so Unsorted Array is better. If graph is sparse, Min Heap is better.

Bellman-Ford

- This will allow us to generate a shorter path than any other one. We cannot find a shortest path using Dijkstra.
- Bellman-Ford computes δ(s,v) even if we have negative weights but reports negative weight cycles (the shortest path will be a simple path – no vertex repetitions are allowed).



If x+y+z is negative, then the length of the path from u to v can be infinitely small. That is: $w(p) = -\infty$

...continued

```
d[s] = 0
```

This initial estimate of 0 may not necessarily be true since now we have negative weights.

```
∀v ∈ V-{s}

d[v] = ∞


for (|V|-1) times

        for each edge (u,v) ∈ E

                if d[v] > d[u] + w(u,v)

                        d[v] = d[u] + w(u,v)
```

The relaxation step.

Detecting negative weight cycles

```
d[s] = 0
∀ v ∈ V-{s}, d[v] = ∞

for (|V|-1) times
        for each edge (u,v) ∈ E
                if d[v] > d[u] + w(u,v)
                        d[v] = d[u] + w(u,v)
```

```
for (|V|-1) times
        for each edge (u,v) ∈ E
                if d[v] > d[u] + w(u,v)
                        Report -ve weight cycles
```

Look for problems here.

If no -ve weight cycles found then d[v] = δ(s, v)

Running time

```
d[s] = 0
∀ v ∈ V-{s}, d[v] = ∞
```

Linear time.

```
for (|V|-1) times
        for each edge (u,v) ∈ E
                if d[v] > d[u] + w(u,v)
                        d[v] = d[u] + w(u,v)
```

|E| × |V|

```
for (|V|-1) times
        for each edge (u,v) ∈ E
                if d[v] > d[u] + w(u,v)
                        Report -ve weight cycles
```

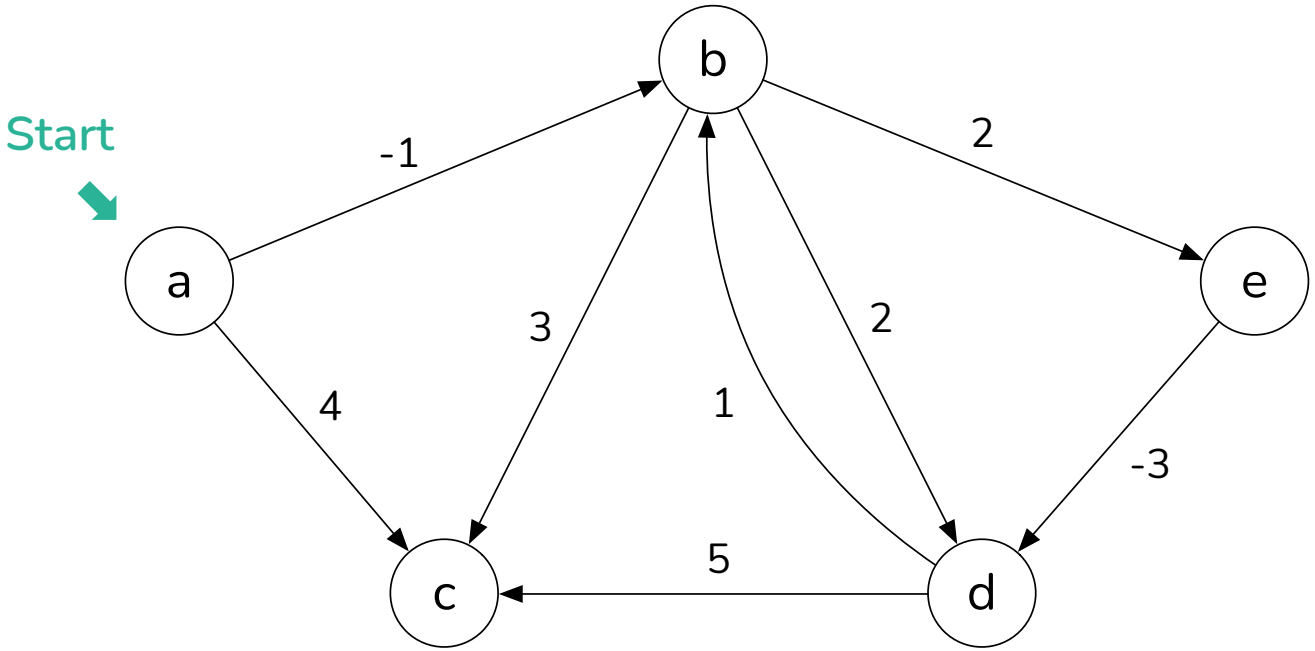
|E| × |V|

VE+VE = O(VE)...
so at least quadratic in V.

(the number of edges must be at least the V-1 since the graph needs to be connected)

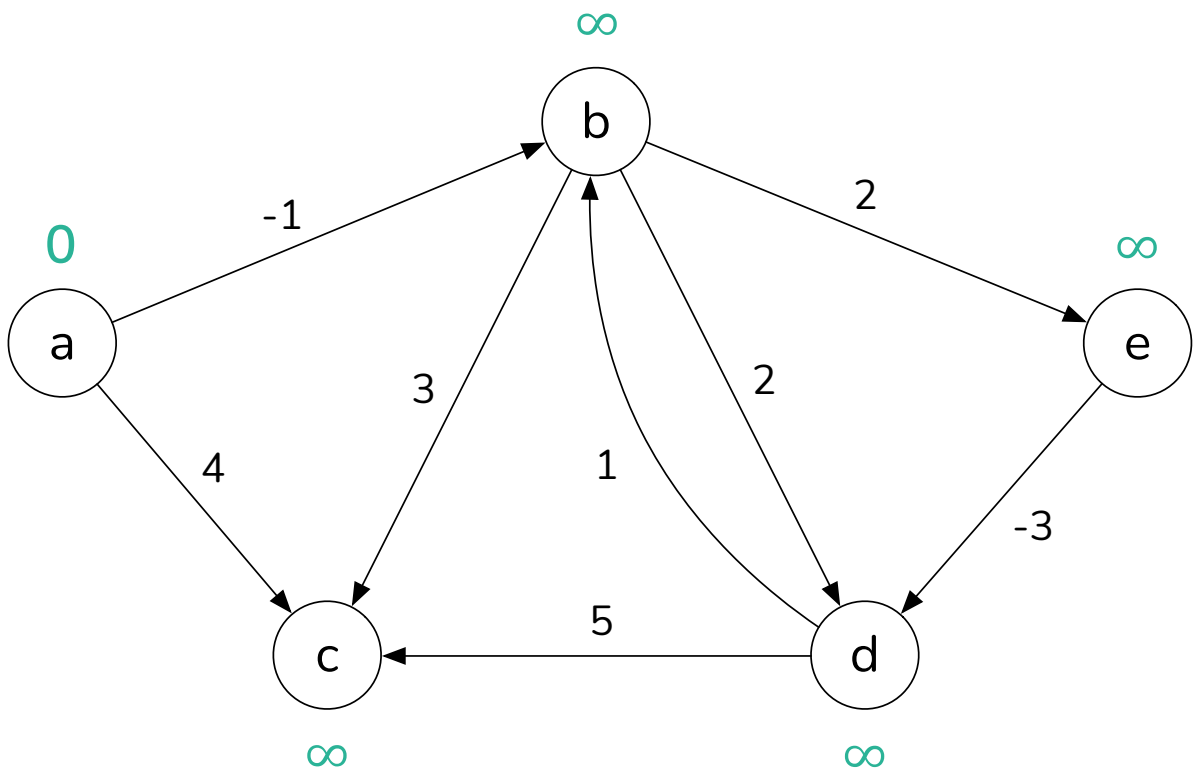
```
If no -ve weight cycles found then d[v] = δ(s, v)
```

A simple example



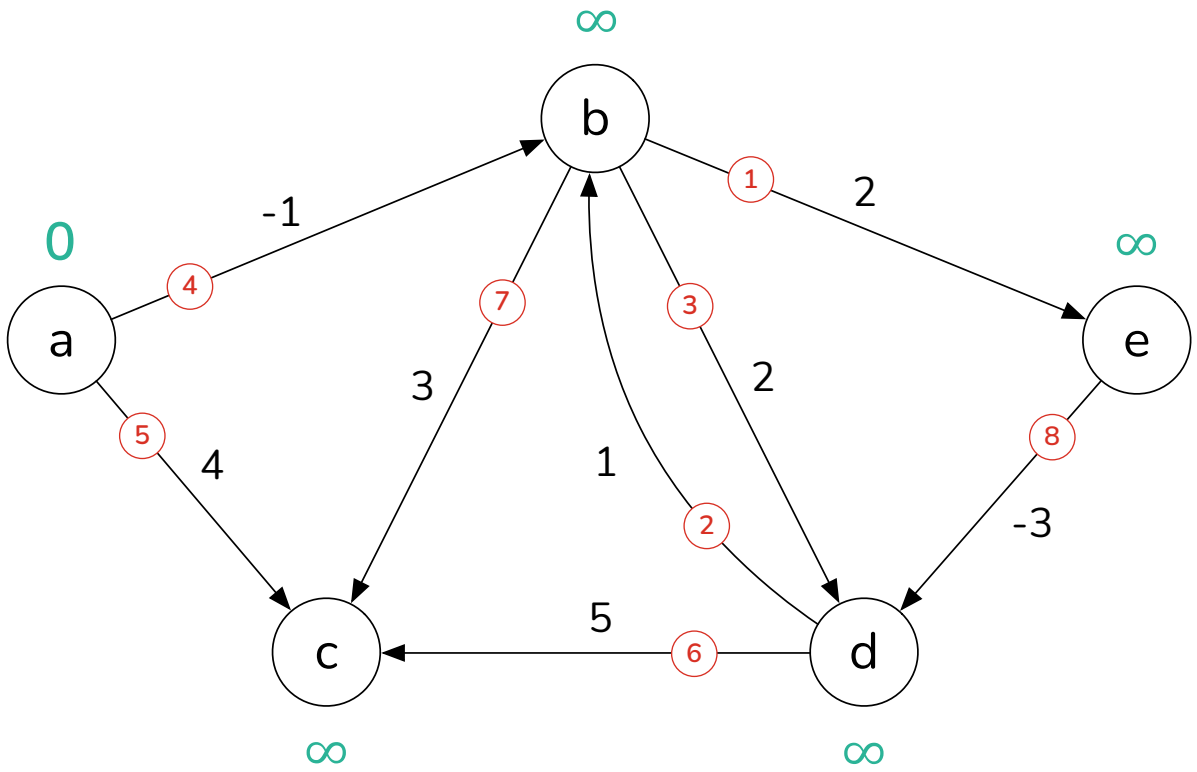
Example from Demaine

Initialisation




```
d[s] = 0
∀ v ∈ V-{s}
d[v] = ∞
```

Picking an edge order

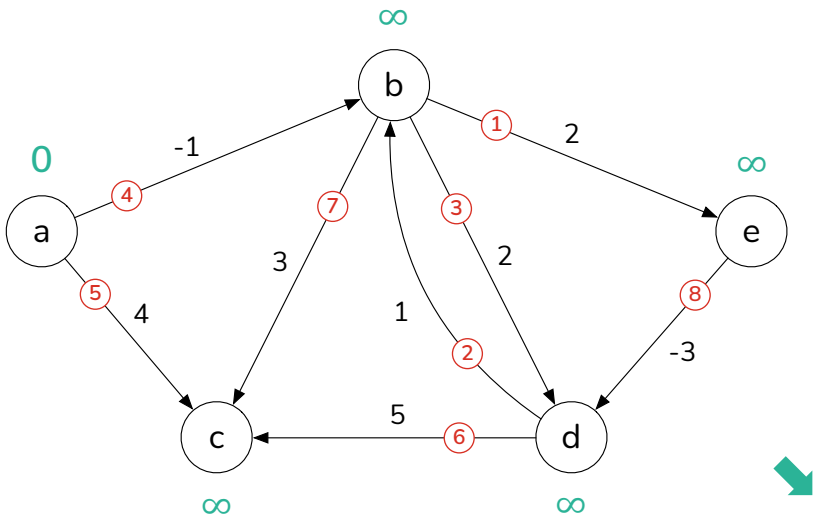


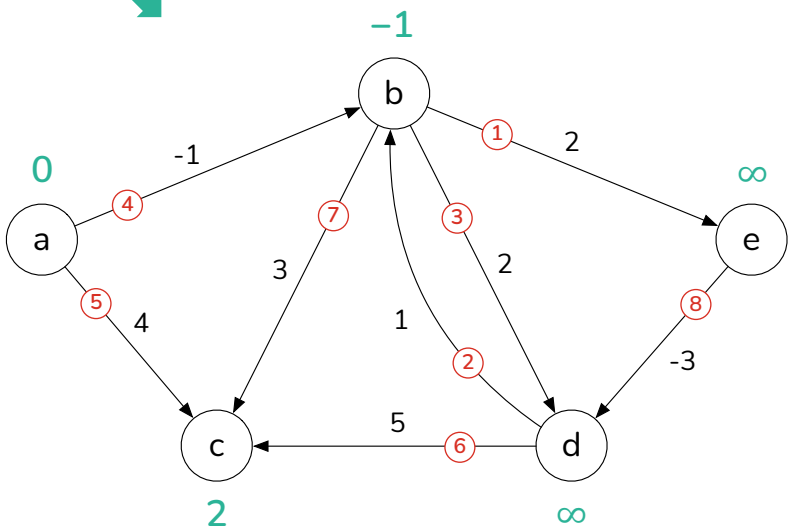
Note that the algorithm says that we must loop through all the edges in the graph (**`for each edge (u,v)` ∈ `E`**) but the order is not specified (and not important for the algorithm to work. As a matter of fact, the order can also change for every iteration of V).

However, to work out the example consistently we will pick an order to work with.

Remember that the edge processing order is not important, we are doing this only for the dry run.

Pass 1

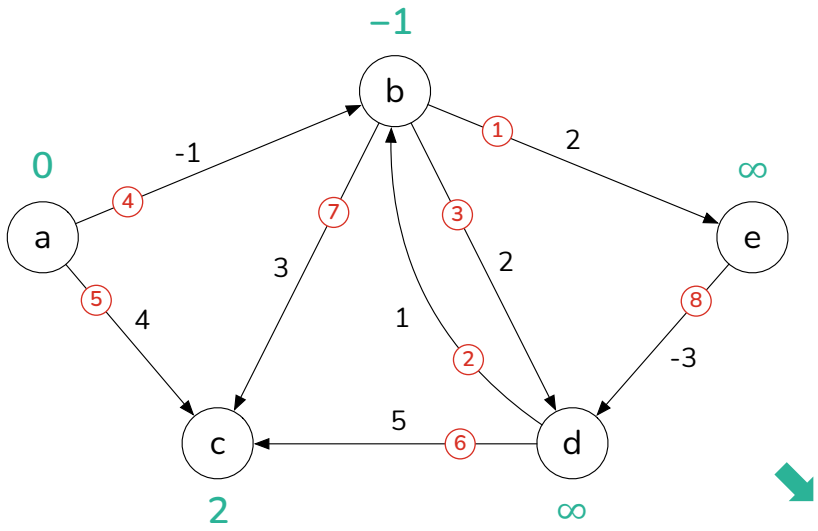


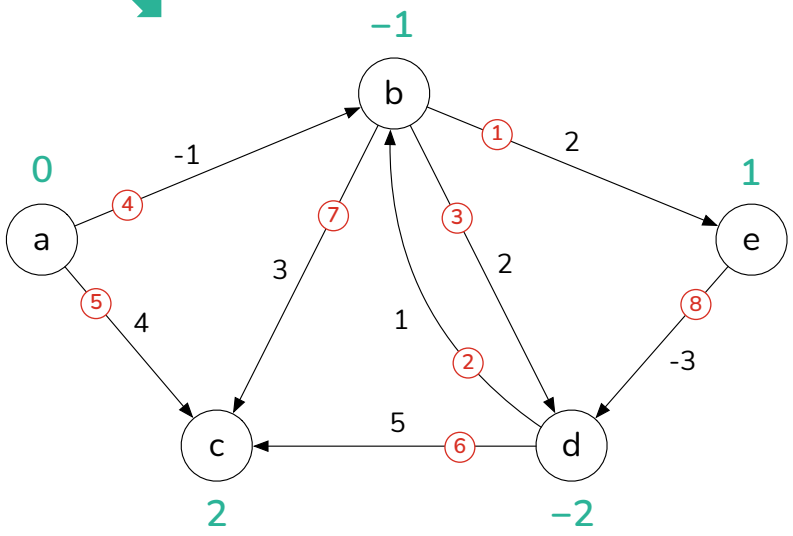



```
if d[v] > d[u] + w(u,v)
  then d[v] = d[u] + w(u,v)

1: Relax (b,e), so d[e] = ∞
2: Relax (d,b), so d[b] = ∞
3: Relax (b,d), so d[d] = ∞
4: Relax (a,b), so d[b] = -1
5: Relax (a,c), so d[c] = 4
6: Relax (d,c), so d[c] = 4
7: Relax (b,c), so d[c] = 2
8: Relax (e,d), so d[d] = ∞
```

Pass 2

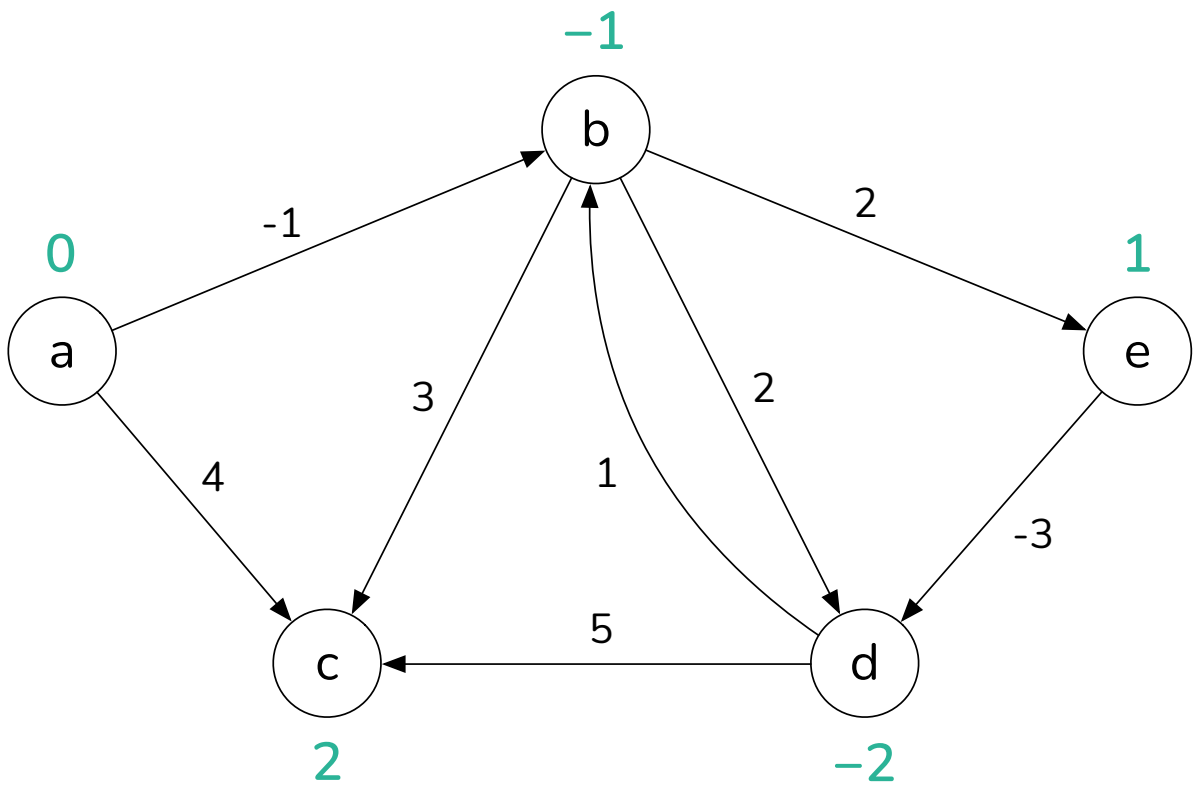





```
if d[v] > d[u] + w(u,v)
  then d[v] = d[u] + w(u,v)

1: Relax (b,e), so d[e] = 1
2: Relax (d,b), so d[b] = -1
3: Relax (b,d), so d[d] = 1
4: Relax (a,b), so d[b] = -1
5: Relax (a,c), so d[c] = 2
6: Relax (d,c), so d[c] = 2
7: Relax (b,c), so d[c] = 2
8: Relax (e,d), so d[d] = -2
```

Passes 3 and 4 (remember that 4 is $|V| - 1$)



Nothing happens in step 3, so we can just break and not bother with step 4.

Proof of correctness

If $G = (V, E)$ contains no negative weight cycles, after Bellman-Ford completes $d[v] = \delta(s, v)$ for all $v \in V$.

- Consider any $v \in V$ and let p be the shortest path from s to v having the minimum number of edges.

$$p = v_0 \rightarrow v_1 \rightarrow \cdots \rightarrow v_k$$

s v

- Remember the previous 'overestimate' lemma that showed $d[v] \geq \delta(s, v)$.

...continued

- Also remember that by enforcing that p has the minimum number of edges we are guaranteeing that p is a simple path.
  - Note that not all shortest paths are necessarily simple.
  - Negative weight cycles are not present because the algorithm is defined not to work for negative weight cycles in the first place.
  - This added 'minimum number of edges' constraint guarantees that we do not have any zero weight cycles.

...continued

- By optimal substructure (proven before for Dijkstra):

$$\delta(s, v_i) = \delta(s, v_{i-1}) + w(v_{i-1}, v_i)$$

- Base case:
  - $d[v_0]$ is initialised to 0, remember v_0 is s.
  - What is the shortest path from s to s? It must be 0 because the only case in which it could be less is if we have a negative weight cycle (negative loop) which is not allowed.
  - So $d[v_0]$ which is $d[s] = 0 = \delta(s, v_o) = \delta(s, s)$.
  - Base case ready.

...continued

- Repeat the following process i times:
  - During the i^{th} round, we relax all the edges, which will certainly include the edge (v_{i-1}, v_i).
  - By the previous lemma (Dijkstra lemma 2): If $d[v_{i-1}] = \delta(s, v_{i-1})$ and we relax the edge (v_{i-1}, v_i) then $d[v_i] = \delta(s, v_i)$
- Note that in the above process, after every i^{th} iteration, we optimize at least one edge in the path.
- How many rounds must we do? How large must i be? How many vertices do we have in a simple path with $|V|$ vertices? There are $V - 1$. This is exactly how much we are iterating in the outer loop.
- Done.

```
for (|V|-1) times
  for each edge (u,v) ∈ E
    relax edge
```

Corollary

If $d[v]$ fails to converge after $|V| - 1$ cycles, then there is a negative weight cycle in G from s.

Further reading

- These notes should be supplemented by:
  - Introduction to Algorithms (Clifford Stein, Thomas H Cormen, Ronald L Rivest, Charles E Leiserson – MIT Press)
  - Extra material from Erik Demaine, Massachusetts Institute of Technology, MIT, http://videolectures.net/mit6046jf05_demaine_lec17/)