Advanced Algorithm Design

- Algorithm Basics and asymptotic notation
- Divide and Conquer (Merge sort)
- Quick sort and Heap sort
- Binary search trees
- Red-black trees
- Graph searching algorithms
- Graph algorithm

Algorithm basics

Definition of algorithms

RAM computation model

Running time of algorithms

- Worst case running time
- Average case running time
- Best case running time
- Asymptotic notations

Definition

- An algorithm is a **finite** sequence of **unambiguous** instructions for solving a well-specified computational problem.
- Important Features:
  - Finiteness.
  - Definiteness.
  - Input.
  - Output.
  - Effectiveness.

RAM Model

- Run time expression should be machine-independent.
  - Use a model of computation or "hypothetical" computer.
  - Our choice – **RAM model** (most commonly-used).
- Model should be
  - Simple.
  - Applicable.

RAM Model

- Generic single-processor model.
- **Supports simple constant-time instructions** found in real computers.
  - Arithmetic (+, –, *, /, %, floor, ceiling).
  - Data Movement (load, store, copy, assignment statement).
  - Control (branch, subroutine call, loop control).
- Run time (**cost**) is uniform (**1 time unit**) for all simple instructions.
- Memory is unlimited.
- Flat memory model – no hierarchy.
- Access to a word of memory takes **1 time unit**.
- Sequential execution – **no concurrent operations**.

Running Time – Definition

- Call each simple instruction and access to a word of memory a "primitive operation" or "step."
- Running time of an algorithm for a given input is
  - The **number of steps** executed by the algorithm on that **input**.
- Often referred to as the ***complexity*** of the algorithm.

Complexity and Input

- Complexity of an algorithm generally depends on
  - **Size of input.**
    - Input size depends on the problem.
      - Examples: No. of items to be sorted.
      - No. of vertices and edges in a graph.
  - **Other characteristics of the input data.**
    - Are the items already sorted?
    - Are there cycles in the graph?

Worst, Average, and Best-case Complexity

- Worst-case Complexity
  - **Maximum** number of steps the algorithm takes for any possible input.
  - Most tractable measure.
- Average-case Complexity
  - **Average** of the running times of all ***possible* inputs**.
  - Demands a definition of probability of each input, which is usually difficult to provide and to analyze.
- Best-case Complexity
  - **Minimum** number of steps for any possible input.
  - Not a useful measure. Why?

A Simple Example – *Linear Search*

INPUT: **a sequence of *n* numbers, *key* to search for.**

OUTPUT: ***true* if *key* occurs in the sequence, *false* otherwise.**

LinearSearch(A, *key*)	*cost*	*times*
$i \leftarrow 1$	c_1	1
while $i \leq n$ **and** A[i] != *key*	c_2	x
do i++	c_3	x-1
if $i \leq n$	c_4	1
then return *true*	c_5	1
else return *false*	c_6	1

x ranges between 1 and n + 1.

So, the running time ranges between $c_1 + c_2x + c_3(x - 1) + c_4 + c_6$

$c_1 + c_2 + c_4 + c_5$ – **best case**

and

$c_1 + c_2(n+1) + c_3n + c_4 + c_6$ – **worst case**

A Simple Example – *Linear Search*

INPUT: a sequence of *n* numbers, *key* to search for.

OUTPUT: *true* if *key* occurs in the sequence, *false* otherwise.

LinearSearch(A, *key*)	*cost*	*times*
$i \leftarrow 1$	1	1
while $i \le n$ **and** A[i] != *key*	1	x
do i++	1	x-1
if $i \le n$	1	1
then return *true*	1	1
else return *false*	1	1

Assign a cost of 1 to all statement executions.

Now, the running time ranges between

$1 + 1 + 1 + 1 = 4$ – **best case**

and

$1 + (n+1) + n + 1 + 1 = 2n + 4$ – **worst case**

A Simple Example – *Linear Search*

INPUT: a sequence of *n* numbers, *key* to search for.

OUTPUT: *true* if *key* occurs in the sequence, *false* otherwise.

	cost	times
LinearSearch(A, *key*)	*cost*	*times*
1 $i \leftarrow 1$	1	1
2 **while** $i \le n$ **and** A[i] != *key*	1	x
3 **do** i++	1	x-1
4 **if** $i \le n$	1	1
5 **then return** *true*	1	1
6 **else return** *false*	1	1

If we assume that the *key* is equal to a random item in the list, on average, statements 2 and 3 will be executed $n/2$ times. Running times of other statements are independent of input. Hence, **average-case complexity** is

$$1 + n/2 + n/2 + 1 + 1 = n + 3$$

Order of growth

- Principal interest is to determine
  - how running time grows with input size – **Order of growth**.
  - the running time for large inputs – **Asymptotic complexity**.
- In determining the above,
  - **Lower-order terms and coefficient of the highest-order term are insignificant.**
  - **Ex: In $7n^5+6n^3+n+10$, which term dominates the running time for very large n? - n^5.**
- Complexity of an algorithm is denoted by the highest-order term in the expression for running time.
  - Ex: **$O(n)$, $\Theta(1)$, $\Omega(n^2)$**, etc.
  - Constant complexity when running time is independent of the input size – denoted $O(1)$.
  - Linear Search: **Best case $\Theta(1)$, Worst and Average cases: $\Theta(n)$.**
- More on O, Θ, and Ω in next classes. Use Θ for present.

Asymptotic notations

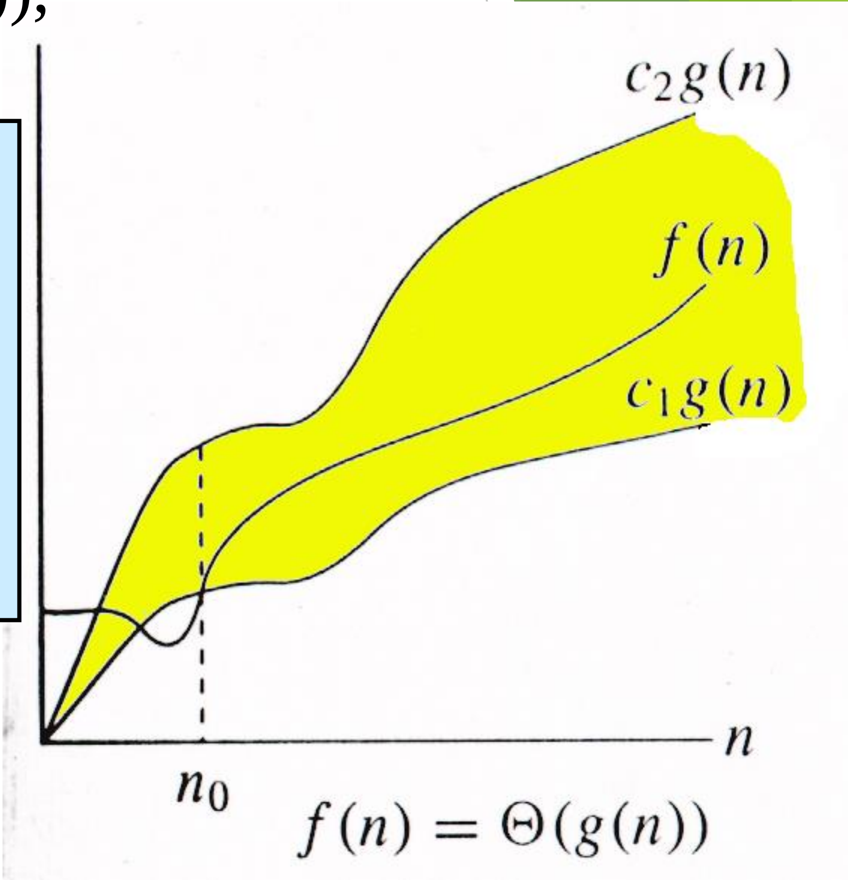
- Tight bound
- Upper bound
- Lower bound

Θ-notation

$g(n)$ = c (a constant), n, n^2, n^3, …

For function $g(n)$, we define $\Theta(g(n))$, big-Theta of n, as a set:

$\Theta(g(n)) = \{f(n)$:
∃ positive constants c_1, c_2, and n_0,
such that $\forall n \geq n_0$,
we have $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$
$\}$



Intuitively: Set of all functions that have the same *rate of growth* as $g(n)$.

$g(n)$ is an *asymptotically tight bound* for any $f(n)$ in the set.

Example

$\Theta(g(n)) = \{f(n) :$ **∃ positive constants c_1, c_2, and n_0, such that $\forall n \geq n_0$,** $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$

- $10n^2 - 3n = \Theta(n^2)$?
- What constants for n_0, c_1, and c_2 will work?
- Make c_1 a little smaller than the leading coefficient, and c_2 a little bigger.
- ***To compare orders of growth, look at the leading term (highest-order term).***
- Exercise: Prove that $n^2/2-3n = \Theta(n^2)$

Example

$\Theta(g(n)) = \{f(n) :$ ∃ **positive constants** c_1, c_2, **and** n_0, **such that** $\forall n \geq n_0$, $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$

- Is $3n^3 \in \Theta(n^4)$?
- If it is true, we can find c_1, c_2, and n_0 such that for $n > n_0$, we have

  $c_1 n^4 \leq 3n^3 \leq c_2 n^4$.

  $c_1 n^4 \leq 3n^3 \Rightarrow n \leq 3/c_1$.
- It is a contradiction. So, $3n^3 \notin \Theta(n^4)$?

Example

$\Theta(g(n)) = \{f(n) :$ **∃ positive constants c_1, c_2, and n_0, such that $\forall n \geq n_0$,** $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$

- How about $2^{2n} \in \Theta(2^n)$?
- If it is true, we can find c_1, c_2, and n_0 such that for $n > n_0$, we have

  $c_1 2^n \leq 2^{2n} \leq c_2 2^n$.

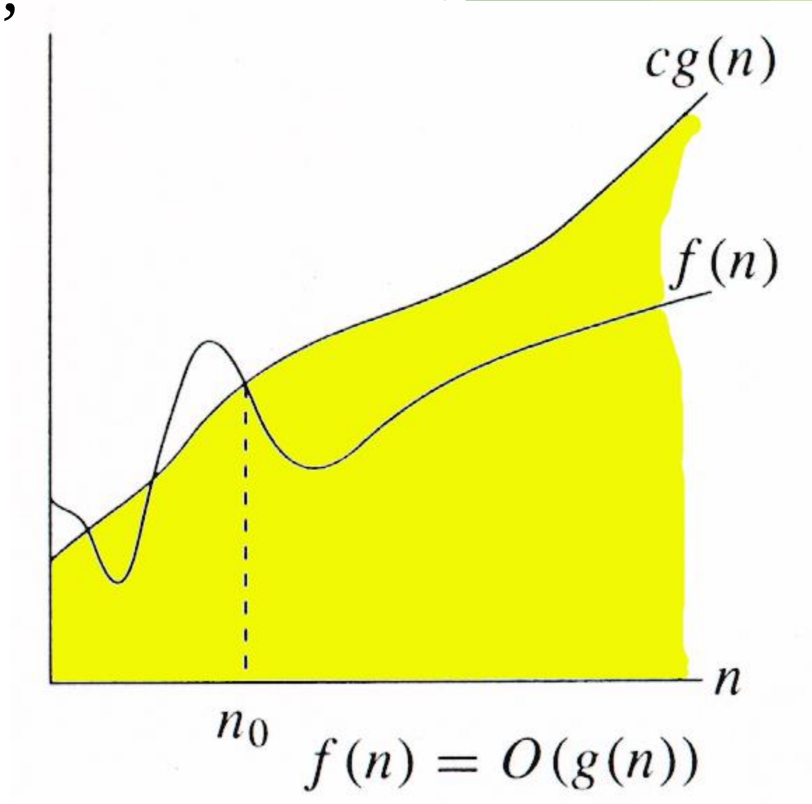
  $2^{2n} \leq c_2 2^n \quad \Rightarrow \quad 2^n \leq c_2 \Rightarrow \quad n \leq \log_2 c_2$.
- It is a contradiction. So, $2^{2n} \notin \Theta(2^n)$?

O-notation

For function $g(n)$, we define $O(g(n))$, big-O of n, as the set:

$O(g(n)) = \{f(n)$:
$\exists$ positive constants c and n_0, such that $\forall n \geq n_0$,
we have $0 \leq f(n) \leq cg(n)$ $\}$

Intuitively: Set of all functions whose *rate of growth* is the same as or lower than that of $g(n)$.



$g(n)$ is an *asymptotic upper bound* for any $f(n)$ in the set.

$f(n) = \Theta(g(n)) \Rightarrow f(n) = O(g(n))$.
$\Theta(g(n)) \subset O(g(n))$.

Examples

.

$O(g(n))$ = {$f(n)$: ∃ positive constants c and n_0, such that $\forall n \geq n_0$, we have $0 \leq f(n) \leq cg(n)$ }

- Any linear *function an* + *b* is in $O(n^2)$. **How?**
- To answer this question, we set $c = 1$, to see whether we have $an + b < n^2$ for $n >$ a constant n_0.
- To determine the value of n_0, we will solve an equation: $n^2 - an - b = 0$.
- We get $n_0 = \frac{a+\sqrt{a^2+4b}}{2}$

Examples

$O(g(n)) = \{f(n) :$ **$\exists$ positive constants c and n_0, such that $\forall n \geq n_0$,** we have $0 \leq f(n) \leq cg(n)$ $\}$

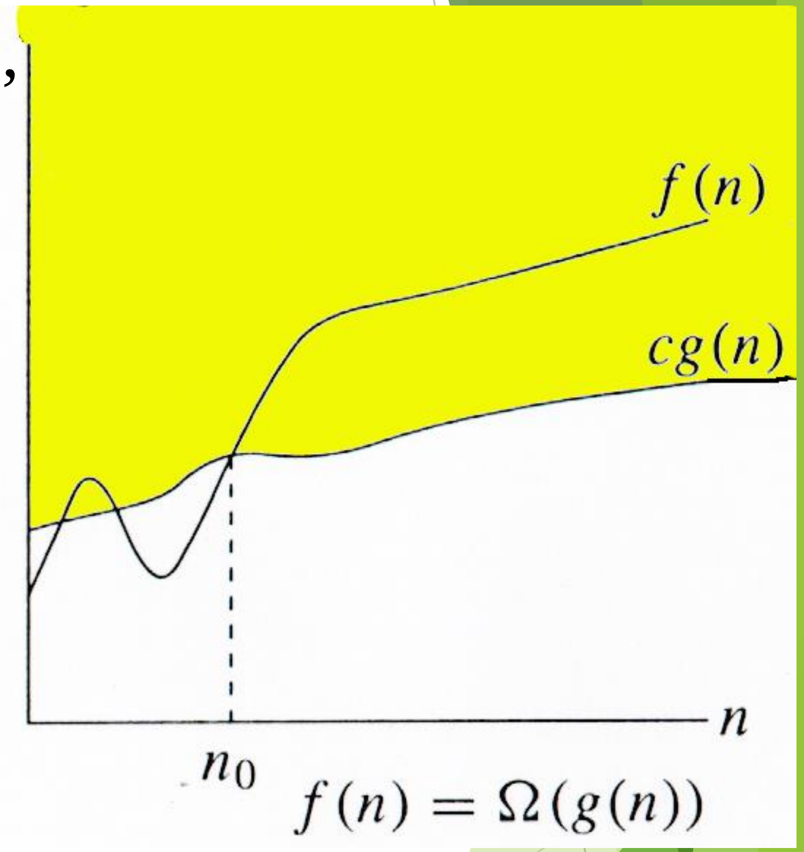
- Show that $3n^3 = O(n^4)$ for appropriate c and n_0.
- The answer is obviously *yes,* since for any $n > n_0 = 4$, we must have $n^4 > 3n^3$.
- Show that $3n^3 = O(n^3)$ for appropriate c and n_0.
- The answer is also *yes,* since we can take $c = 4$, and for any $n > n_0 = 1$, we must have $cn^3 > 3n^3$.

Ω -notation

For function $g(n)$, we define $\Omega(g(n))$, big-Omega of n, as the set:

$\Omega(g(n)) = \{f(n)$:
$\exists$ **positive constants c and n_0,**
such that $\forall n \geq n_0$,
we have $0 \leq cg(n) \leq f(n)\}$

Intuitively: Set of all functions whose *rate of growth* is the same as or higher than that of $g(n)$.



***g*(*n*) is an *asymptotic lower bound* for any *f*(*n*) in the set.**

$f(n) = \Theta(g(n)) \Rightarrow f(n) = \Omega(g(n))$.

$\Theta(g(n)) \subset \Omega(g(n))$.

Example

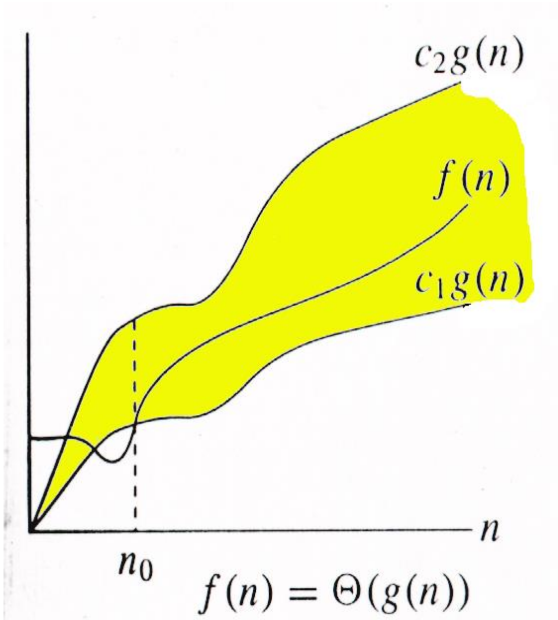
- $\Omega(g(n)) = \{f(n) : \exists$ positive constants c and n_0, such that $\forall n \geq n_0$, we have $0 \leq cg(n) \leq f(n)\}$

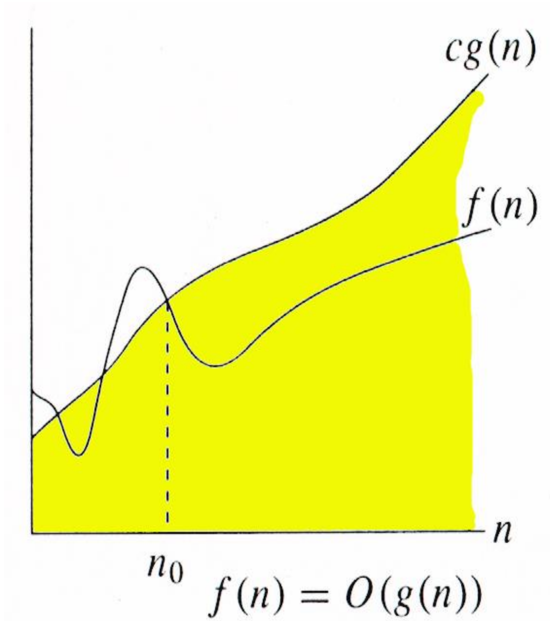
- $\sqrt{n} = \Omega(\log_2 n)$. Choose c and n_0.
- For this purpose, we need to determine constants c and n_0, such that for any $n \geq n_0$, we have

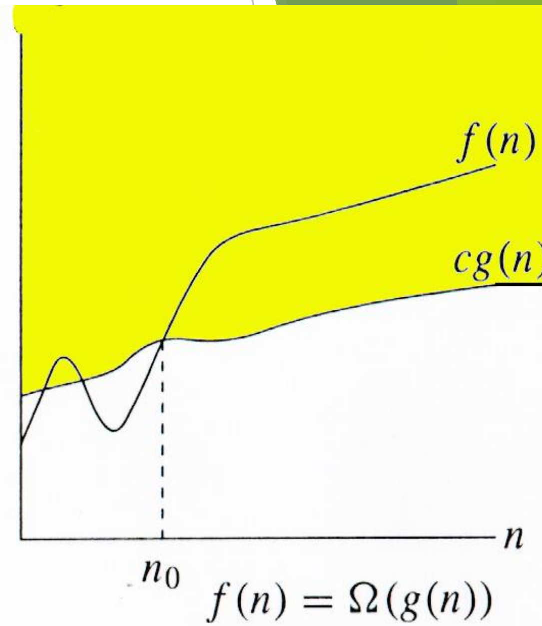
  $$C\log_2 n \leq \sqrt{n}$$

- We can $c = 1$ and $n_0 = 25$ since $\log_2 25 < \log_2 32 = 5 = \sqrt{25}$
- We can also prove that $\sqrt{n} - \log_2 n$ is an increasing function.

Relations Between Θ, *O*, Ω







Divide and Conquer (Merge sort)

Divide and conquer

Merge sort

- Basic merge sort
- Improved merge sort
- Running time analysis
- Correctness proof (loop invariant)

Divide and Conquer

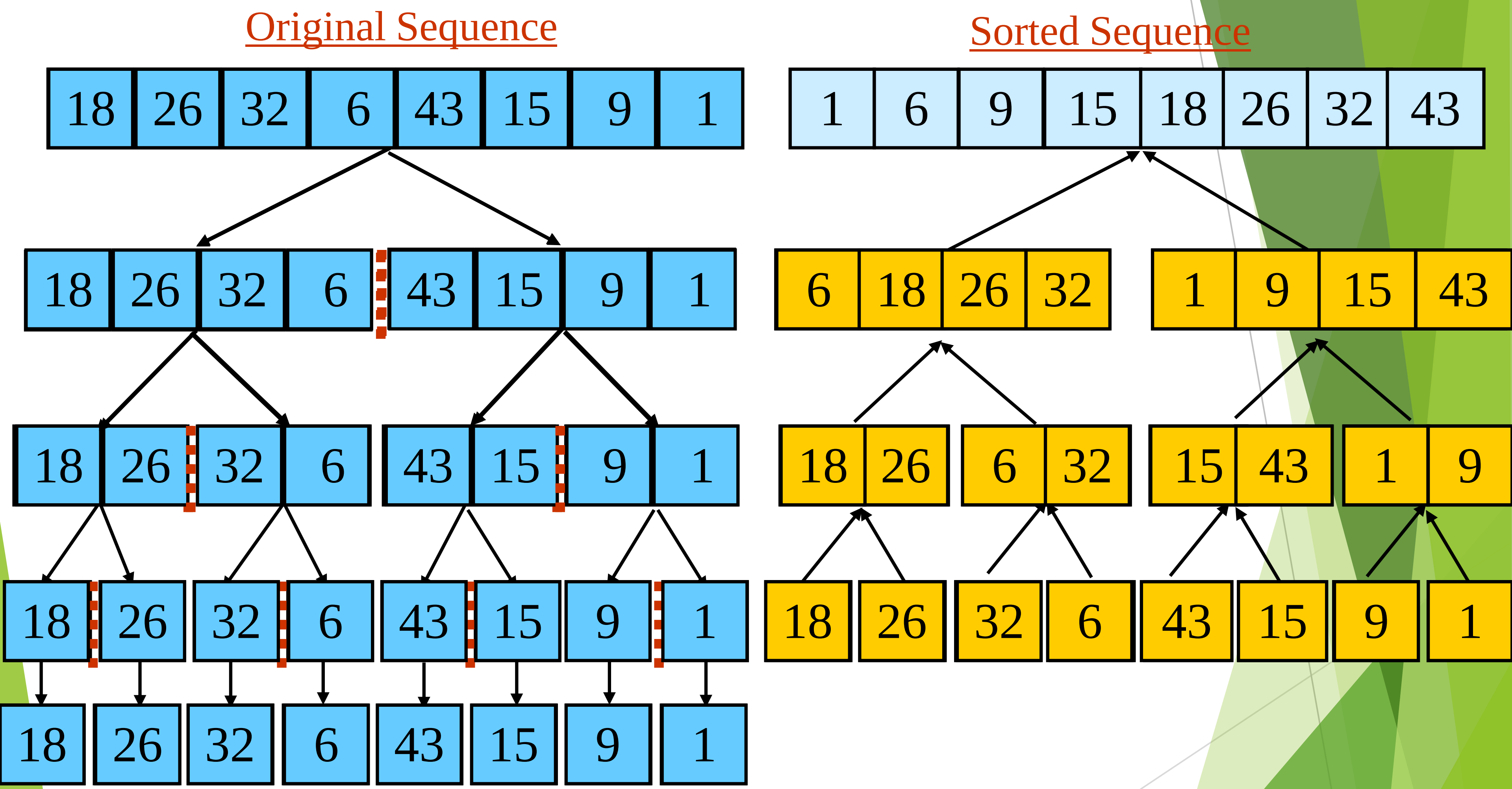
- Recursive in structure
  - ***Divide*** the problem into sub-problems that are similar to the original but smaller in size
  - ***Conquer*** the sub-problems by solving them recursively. If they are small enough, just solve them in a straightforward manner.
  - ***Combine*** the solutions of the sub-problems to create a global solution to the original problem

An Example: Merge Sort

Sorting Problem: Sort a sequence of *n* elements into non-decreasing order.

- ***Divide***: Divide the *n*-element sequence to be sorted into two subsequences of *n*/2 elements each
- ***Conquer:*** Sort the two subsequences recursively using merge sort.
- ***Combine***: Merge the two sorted subsequences to produce the sorted answer.

Merge Sort – Example



Merge-Sort (A, p, r)

INPUT: a sequence of *n* numbers stored in array A

OUTPUT: an ordered sequence of *n* numbers

```
MergeSort (A, p, r)   // sort A[p..r] by divide & conquer
    if p < r
        then q ← ⌊(p+r)/2⌋
            MergeSort (A, p, q)
            MergeSort (A, q+1, r)
            Merge (A, p, q, r) // merges A[p..q] with A[q+1..r]
```

Initial Call: *MergeSort*(*A*, 1, *n*)

Procedure Merge

Merge(*A*, *p*, *q*, *r*)

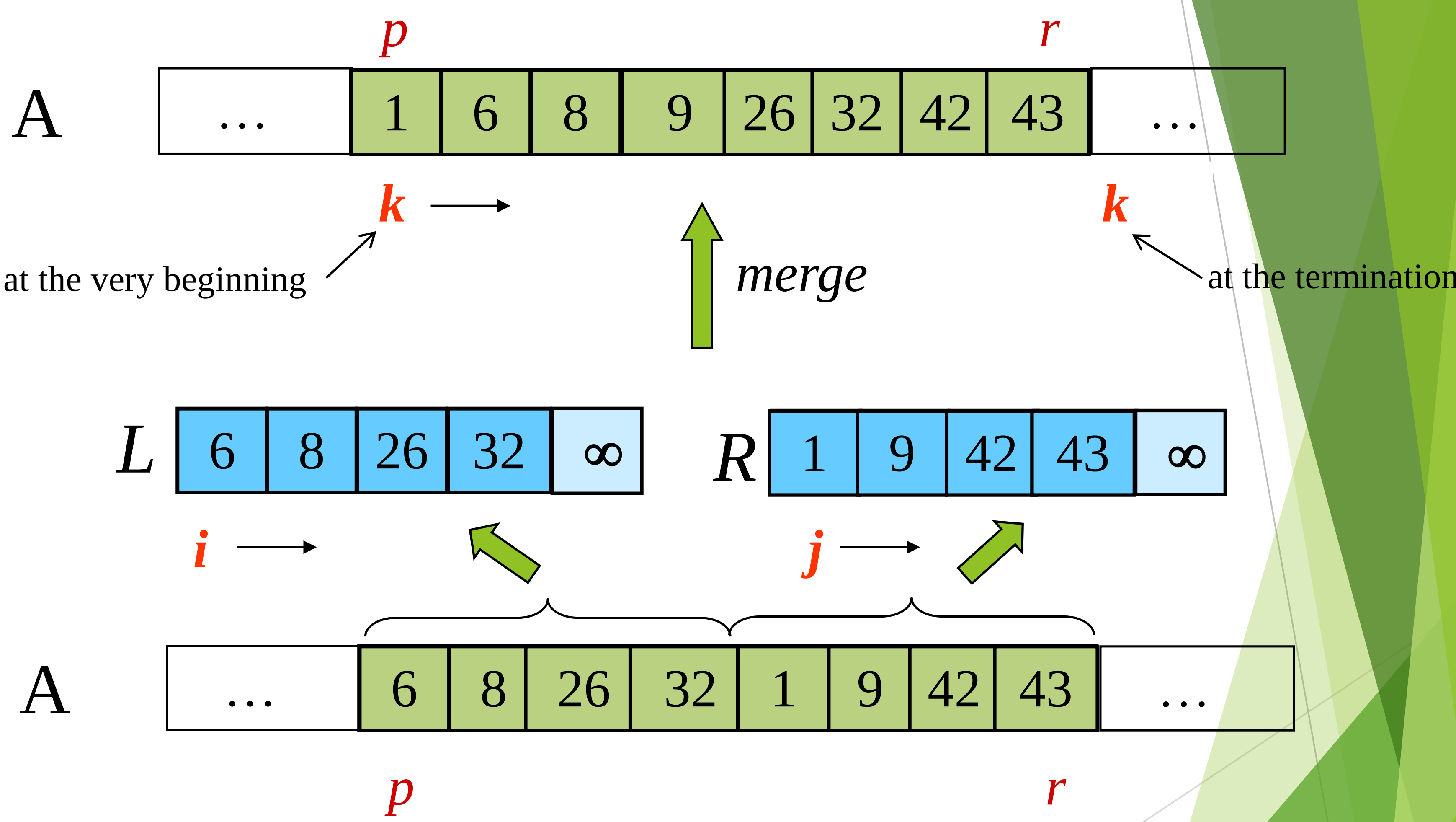
```
n1 ← q – p + 1
n2 ← r – q
    for i ← 1 to n1
        do L[i] ← A[p + i – 1]
    for j ← 1 to n2
        do R[j] ← A[q + j]
    L[n1+1] ← ∞
    R[n2+1] ← ∞
    i ← 1
    j ← 1
    for k ← p to r
        do if L[i] ≤ R[j]
            then A[k] ← L[i]
                i ← i + 1
            else A[k] ← R[j]
                j ← j + 1
```

Input: Array containing sorted subarrays $A[p .. q]$ and $A[q+1 .. r]$.

Output: Merged sorted subarray in $A[p .. r]$.

Sentinels, to avoid having to check if either subarray is fully copied at each step.

Merge - Example



Merge(*A*, *p*, *q*, *r*)

$n_1 \leftarrow q - p + 1$
$n_2 \leftarrow r - q$
for $i \leftarrow 1$ **to** n_1
  do $L[i] \leftarrow A[p + i - 1]$
for $j \leftarrow 1$ **to** n_2
  do $R[j] \leftarrow A[q + j]$
$L[n_1+1] \leftarrow \infty$
$R[n_2+1] \leftarrow \infty$
$i \leftarrow 1$
$j \leftarrow 1$
for $k \leftarrow p$ **to** r
  do if $L[i] \leq R[j]$
    then $A[k] \leftarrow L[i]$
      $i \leftarrow i + 1$
    else $A[k] \leftarrow R[j]$
      $j \leftarrow j + 1$

Loop Invariant for the *for* loop

- At the start of each iteration of the for loop:
  subarray $A[p\,..\,k-1]$ contains the $k - p$ smallest elements of L and R in sorted order.
- $L[i]$ and $R[j]$ are the smallest elements of L and R that have not been copied back into A.

Initialization:

Before the first iteration:

- $A[p\,..\,k-1]$ is empty.
- $i = j = 1$.
- $L[1]$ and $R[1]$ are the smallest elements of L and R not copied to A.

Correctness of Merge

Merge(A, p, q, r)

1 $n_1 \leftarrow q - p + 1$
2 $n_2 \leftarrow r - q$
3 **for** $i \leftarrow 1$ **to** n_1
4 **do** $L[i] \leftarrow A[p + i - 1]$
5 **for** $j \leftarrow 1$ **to** n_2
6 **do** $R[j] \leftarrow A[q + j]$
7 $L[n_1+1] \leftarrow \infty$
8 $R[n_2+1] \leftarrow \infty$
9 $i \leftarrow 1$
10 $j \leftarrow 1$
11 **for** $k \leftarrow p$ **to** r
12 **do if** $L[i] \leq R[j]$
13 **then** $A[k] \leftarrow L[i]$
14 $i \leftarrow i + 1$
15 **else** $A[k] \leftarrow R[j]$
16 $j \leftarrow j + 1$

Maintenance:

(We will prove that if after the kth iteration, the Loop Invariant (LI) holds, we still have the LI after the (k+1)th iteration.)

Case 1: $L[i] \leq R[j]$

- By Loop Invariant, A contains $k - p$ smallest elements of L and R in *sorted order*.
- Also, $L[i]$ and $R[j]$ are the smallest elements of L and R not yet copied into A.
- Line 13 results in A containing $k - p + 1$ smallest elements (again in sorted order). Incrementing i and k reestablishes the LI for the next iteration.

Similarly for Case 2: $L[i] > R[j]$.

Merge(*A*, *p*, *q*, *r*)

```
1  n1 ← q – p + 1
2  n2 ← r – q
3      for i ← 1 to n1
4          do L[i] ← A[p + i – 1]
5      for j ← 1 to n2
6          do R[j] ← A[q + j]
7      L[n1+1] ← ∞
8      R[n2+1] ← ∞
9      i ← 1
10     j ← 1
11     for k ← p to r
12         do if L[i] ≤ R[j]
13             then A[k] ← L[i]
14                     i ← i + 1
15             else A[k] ← R[j]
16                     j ← j + 1
```

Maintenance:

Case 1: $L[i] \le R[j]$

- By Loop Invariant (LI), A contains $k - p$ smallest elements of L and R in *sorted order*.
- By LI, $L[i]$ and $R[j]$ are the smallest elements of L and R not yet copied into A.
- Line 13 results in A containing $k - p + 1$ smallest elements (again in sorted order). Incrementing i and k reestablishes the LI for the next iteration.

Similarly for Case 2: $L[i] > R[j]$.

Termination:

- On termination, $k = r + 1$.
- By LI, A contains $r - p + 1$ smallest elements of L and R in sorted order.
- L and R together contain $r - p + 3 - (r - p + 1) = 2$ elements.

All but the two sentinels have been copied back into A.

Improvements

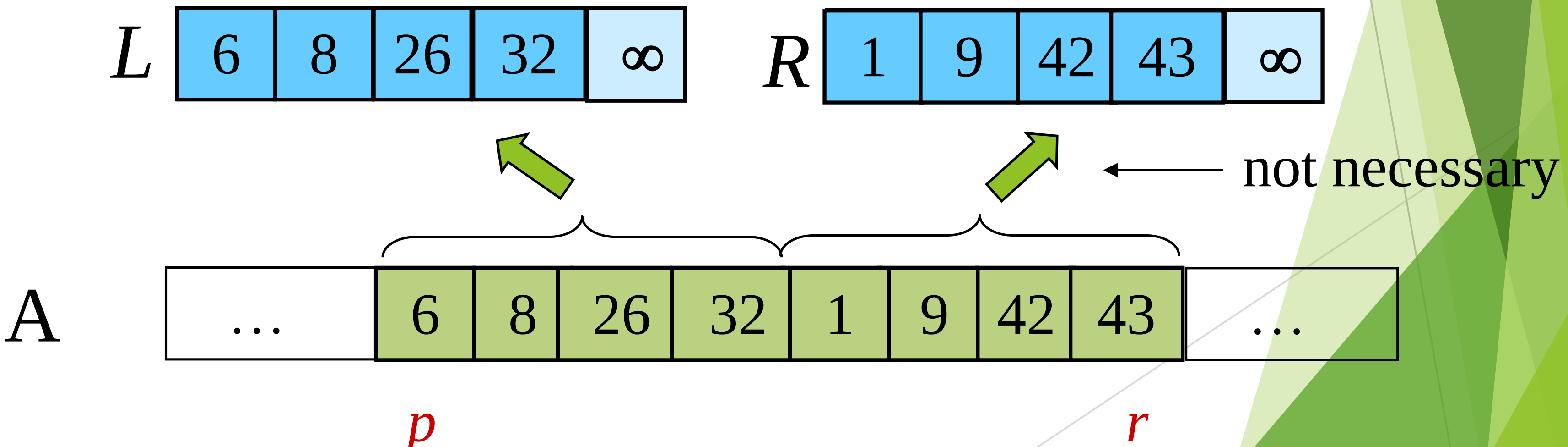
- Reduction of data movements
- Non-recursive Algorithm

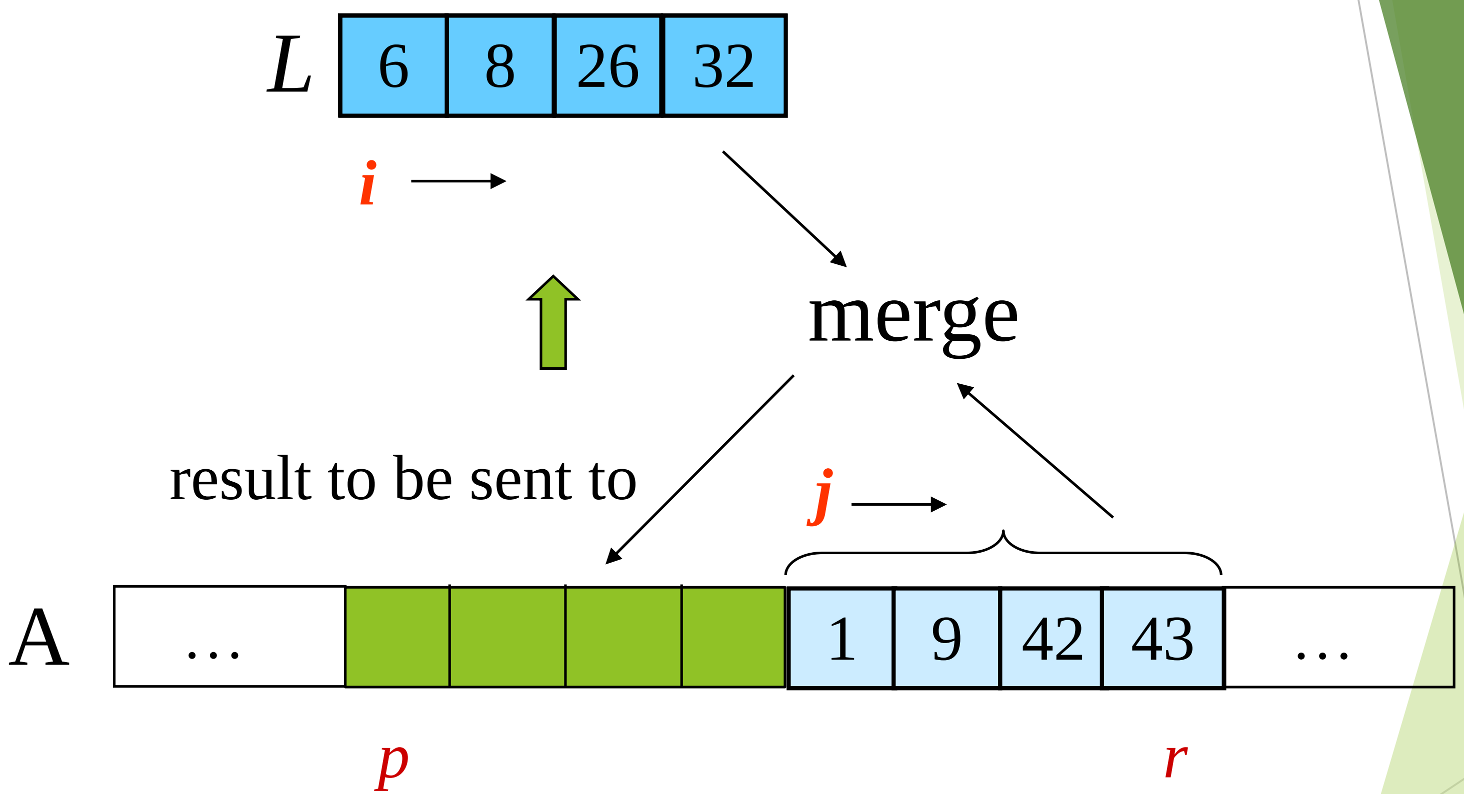
Y. Chen, and R. Su, Merge Sort Revisited, ACTA Scientific Computer Sciences, Vol. 4, No. 5, pp. 49 - 52, 2022.

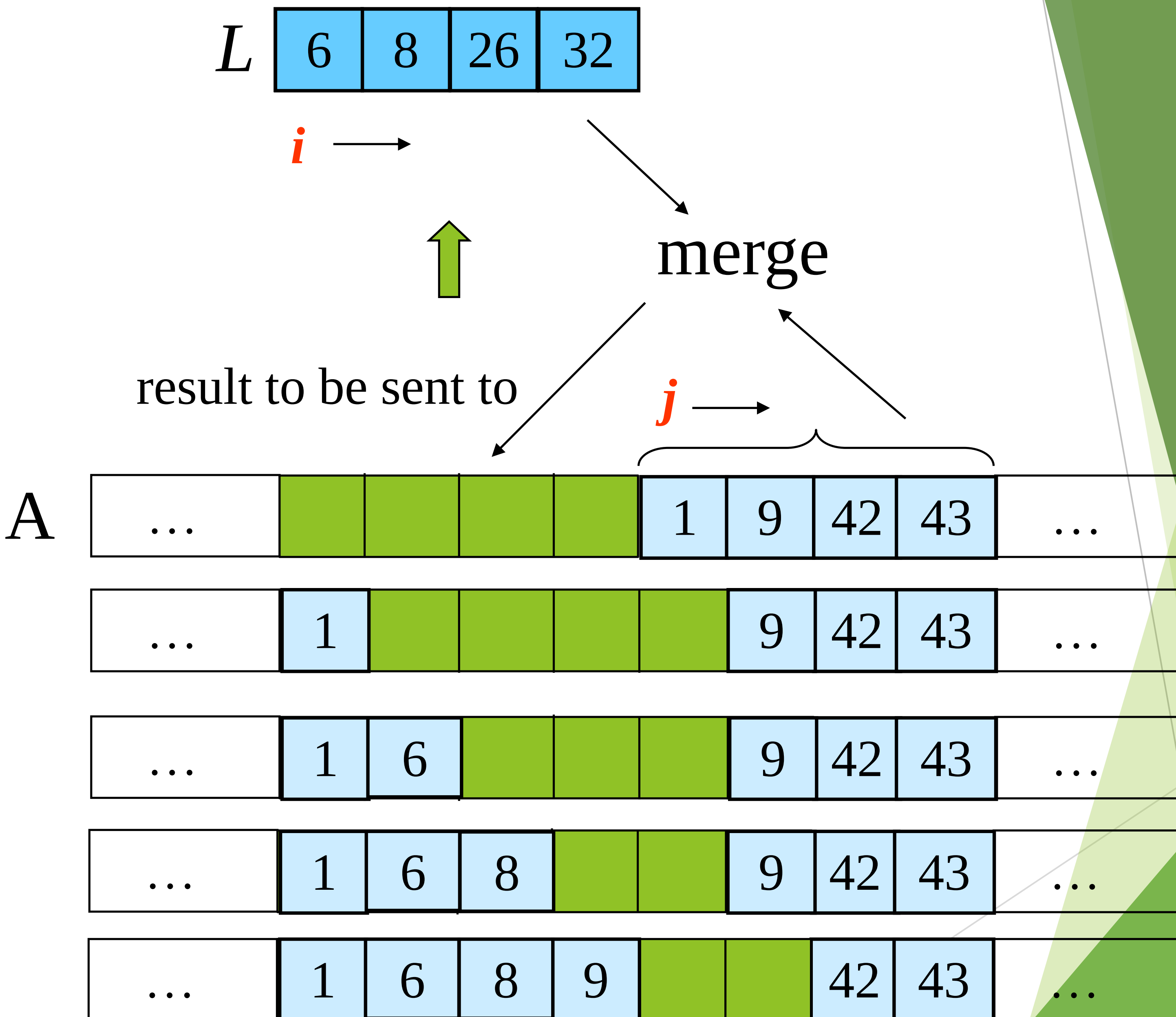
Improvements

- Reduction of data movements

We notice that in the procedure $merge(\ )$ of Merge sort the copying of $A[q + 1 \mathinner{..} r\ ]$ into R is not necessary, since we can directly merge L and $A[q + 1 \mathinner{..} r\ ]$ and store the merged, but sorted sequence back into A.




L
6
8
26
32
i
merge
result to be sent to
j
A
…
1
9
42
43
…
p
r



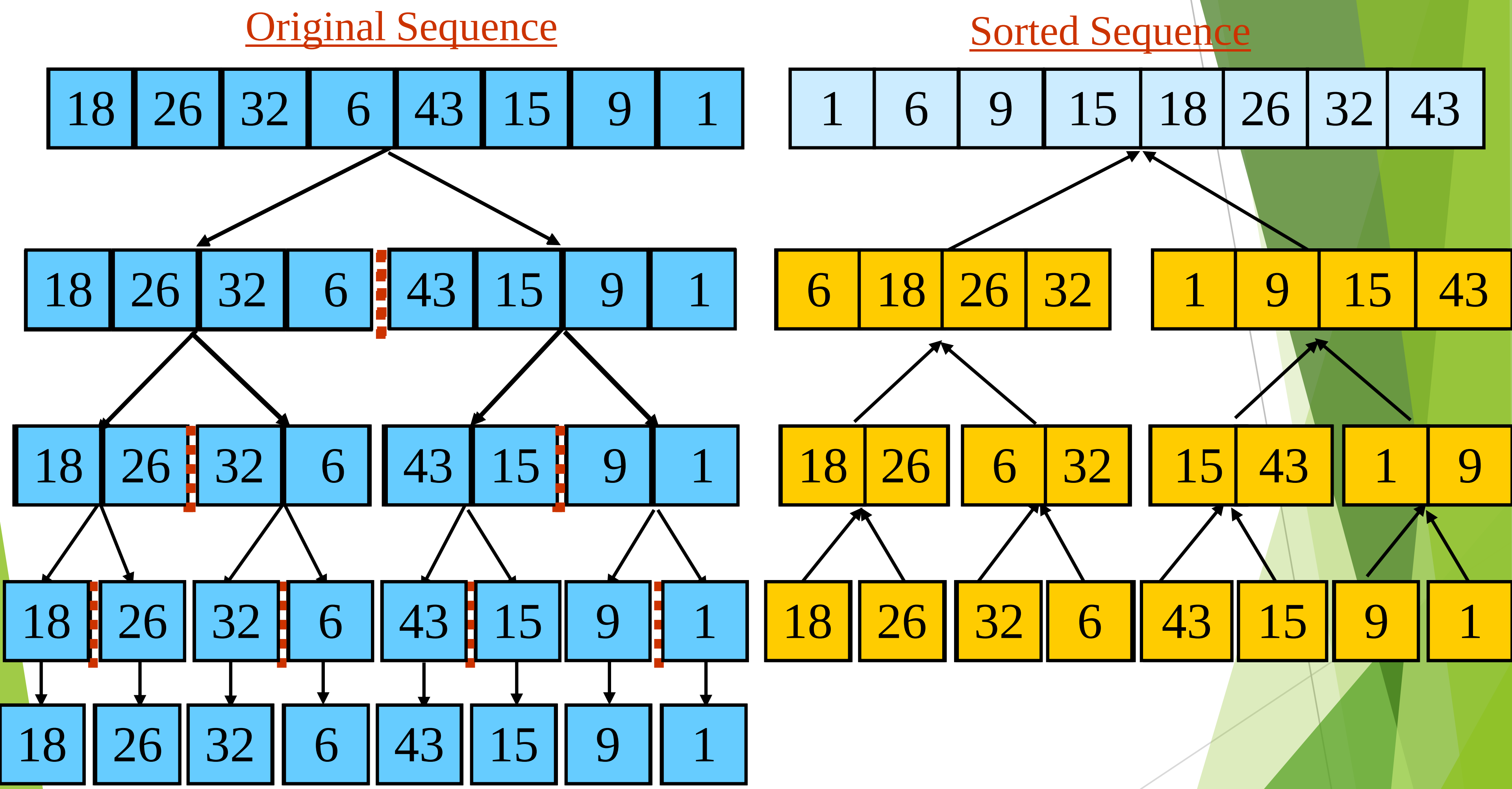
L
6
8
26
32
i
merge
result to be sent to
j
A
…
1
9
42
43
…
…
1
9
42
43
…
…
1
6
9
42
43
…
…
1
6
8
9
42
43
…
…
1
6
8
9
42
43
…

Improvements

Algorithm: $mergeImpr\ (A, p, q, r\ )$
Input: Both $A[p\ ..\ q]$ and $A[q + 1\ ..\ r\ ]$ are sorted; but A as a whole is not sorted
Output : sorted A

1. $n_1 := q - p + 1$; $n_2 := r - p + 1$; $k := p$;
2. let $L[1\ ..\ n_1]$ be a new array;
3. **for** i = 1 to n_1 **do**
4. $\quad L[i] := A[p + i - 1\ ]$
5. $\quad i := p$; $j := q + 1$;
6. **while** $i \leq n_1$ and $j \leq n_2$ **do**
7. $\quad$ **if** $L[i] \leq A[j]$ **then** $\{A[k] := L[i]; i := i + 1;\}$
8. $\quad$ **else** $\{A[k] := A[j]; j := j + 1;\}$
9. $\quad k := k + 1$;
10. **if** $j > n_2$ **then**
11. copy the remaining elements in L into $A[k\ ..\ r\ ]$;

When going out of while-loop,
we distinguish between two cases:
$i > n_1$,
$j > n_2$.


Original Sequence
18 26 32 6 43 15 9 1
18 26 32 6 43 15 9 1
18 26 32 6 43 15 9 1
18 26 32 6 43 15 9 1
18 26 32 6 43 15 9 1
Sorted Sequence
1 6 9 15 18 26 32 43
6 18 26 32 1 9 15 43
18 26 6 32 15 43 1 9
18 26 32 6 43 15 9 1

Non-recursive algorithm

Algorithm: $mSort\ (A)$
Input : A - a sequence of elements stored as an array;
Output : sorted A

1. if $|A| \leq 1$ **then** return A;
2. $r := |A|$;
3. $l := \lceil \log_2 r \rceil$;
4. $j := 2$;
5. for $i = 1$ to l **do**
6. **for** $k = 1$ to $\lceil r/j \rceil)$ **do**
7. $s := \lfloor (k - 1)j \rfloor$;
8. $mergeImpr(A, s + 1, s + \lceil j/2 \rceil, s + j)$;
9. $j := 2j$;

r: the length of A
l : the number of passes
j : the number of elements involved in a merging process in a pass

Analysis of Merge Sort

- Running time $T(n)$ of Merge Sort:
- Divide: computing the middle takes $\Theta(1)$
- Conquer: solving 2 subproblems takes $2T(n/2)$
- Combine: merging n elements takes $\Theta(n)$
- Total:

$$T(n) = \Theta(1) \qquad \text{if } n = 1$$
$$T(n) = 2T(n/2) + \Theta(n) \qquad \text{if } n > 1$$

$\Rightarrow T(n) = \Theta(n \lg n)$ (CLRS, Chapter 4)

Recurrence Relations

Equation or an inequality that characterizes a function by its values on smaller inputs.

Solution Methods (Chapter 4)

- Substitution Method.
- Recursion-tree Method.
- Master Theorem Method.

Recurrence relations **arise when we analyze the running time of iterative or recursive algorithms**.

- Ex: Divide and Conquer.

$$T(n) = \Theta(1) \qquad \text{if } n \le c$$
$$T(n) = a\,T(n/b) + D(n) \qquad \text{otherwise}$$

Substitution Method

- **Guess** the form of the solution, then **use mathematical induction** to show it correct.
  - Substitute guessed answer for the function when the inductive hypothesis is applied to smaller values.
- Works well when the solution is easy to guess.
- No general way to guess the correct solution.

Example – Exact Function

Recurrence: $T(n) = 1$ if $n = 1$

$T(n) = 2T(n/2) + n$ if $n > 1$

- Guess: $T(n) = n \lg n + n$.
- Induction:
  - **Basis:** $n = 1 \Rightarrow n \lg n + n = 1 = T(n)$.
  - **Hypothesis:** $T(k) = k \lg k + k$ for all $k < n$.
  - **Inductive Step:**

$$
\begin{aligned}
T(n) &= 2\,T(n/2) + n \\
&= 2\,((n/2)\lg(n/2) + (n/2)) + n \\
&= n\,(\lg(n/2)) + 2n \\
&= n \lg n - n + 2n \\
&= n \lg n + n
\end{aligned}
$$

Recursion Tree – Example

Running time of Merge Sort:

$$T(n) = \Theta(1) \qquad \text{if } n = 1$$

$$T(n) = 2T(n/2) + \Theta(n) \qquad \text{if } n > 1$$

Rewrite the recurrence as

$$T(n) = c \qquad \text{if } n = 1$$

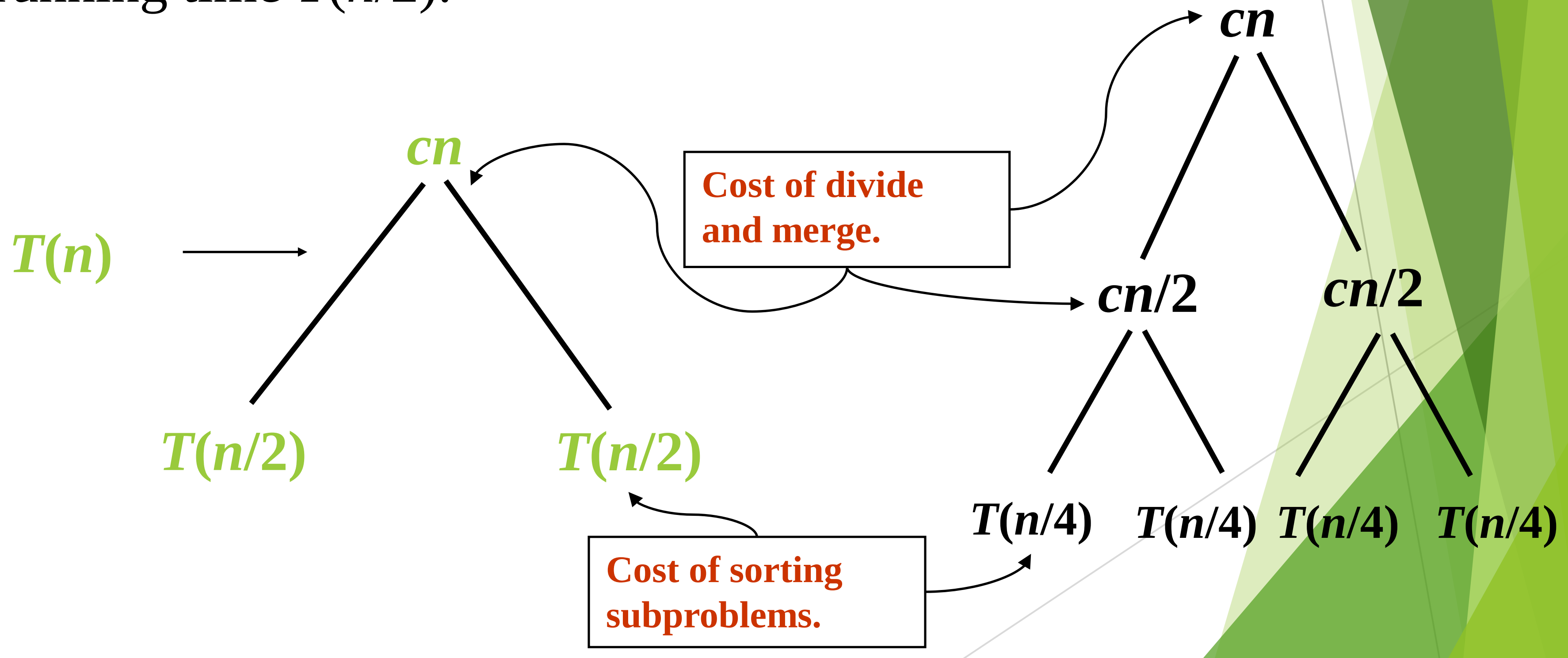
$$T(n) = 2T(n/2) + cn \qquad \text{if } n > 1$$

$c > 0$: Running time for the base case and time per array element for the divide and combine steps.

Recursion Tree for Merge Sort

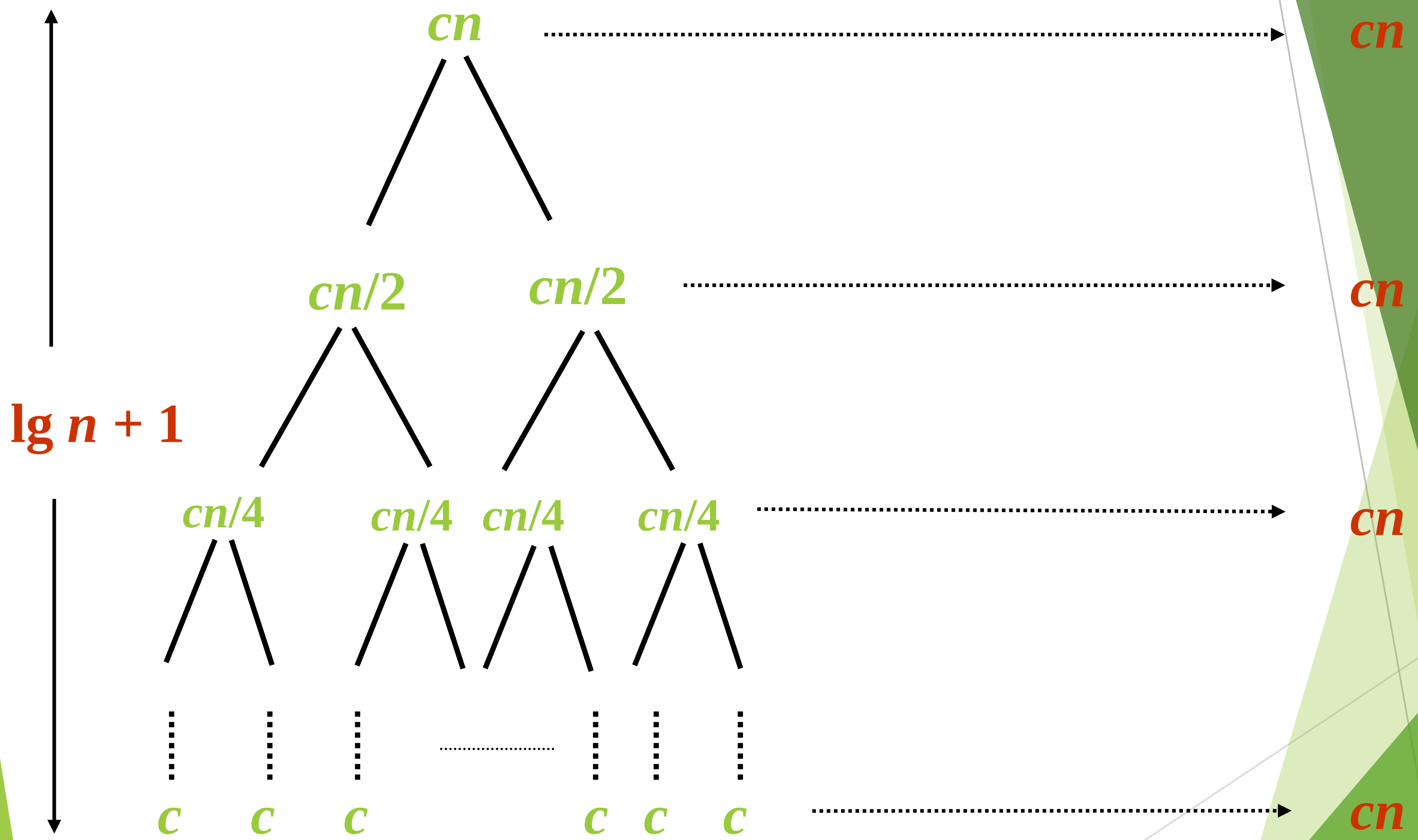
For the original problem, we have a cost of *cn*, plus two subproblems each of size (n/2) and running time T(n/2).

Each of the size n/2 problems has a cost of *cn*/2 plus two subproblems, each costing T(n/4).



Recursion Tree for Merge Sort

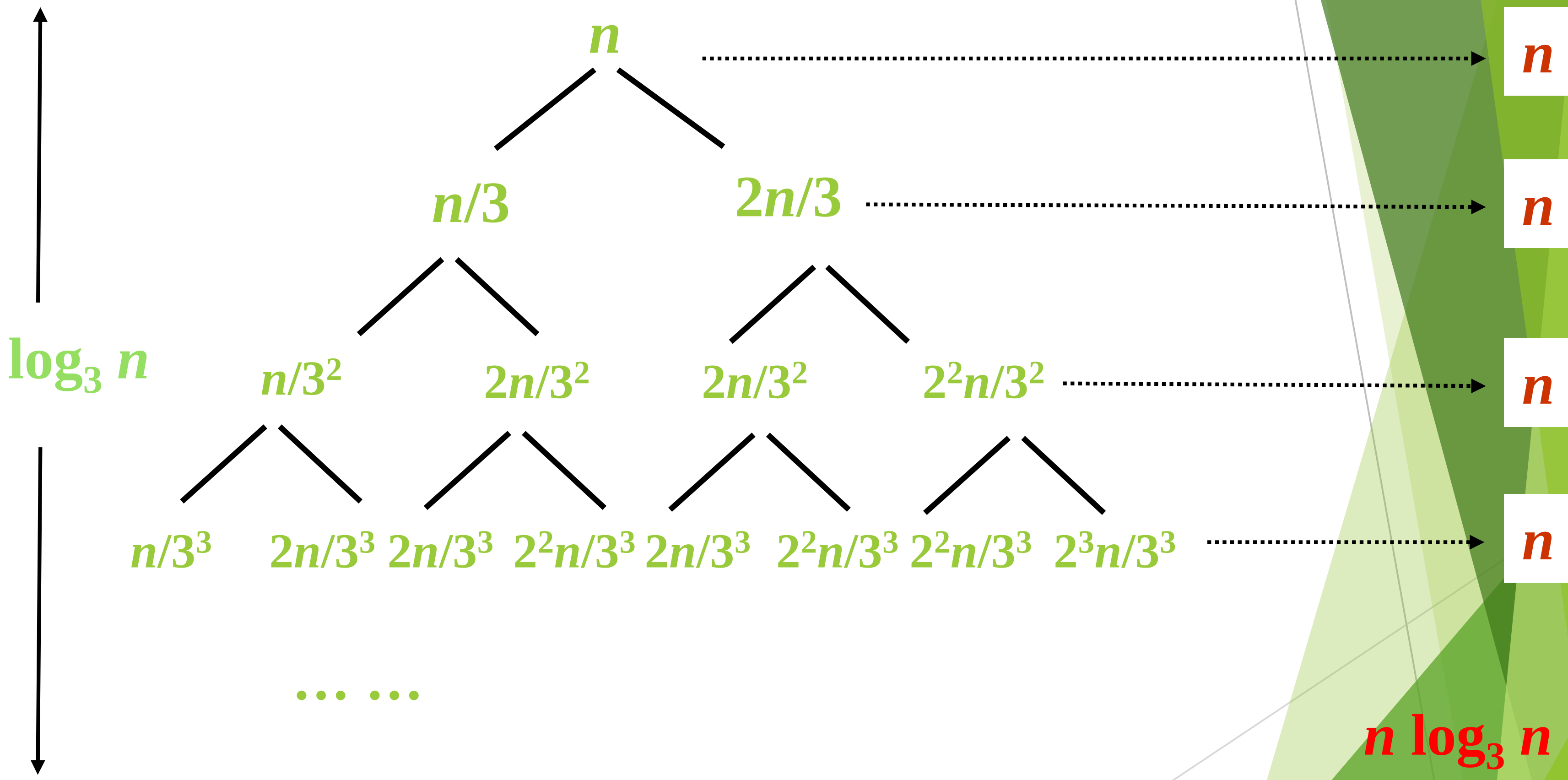
Continue expanding until the problem size reduces to 1.



Total: $cn\lg n + cn$

Other Examples

- $T(n) = T(n/3) + T(2n/3) + O(n)$.



The Master Theorem

Theorem 4.1

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on nonnegative integers by the recurrence $T(n) = aT(n/b) + f(n)$, where we can replace n/b by $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. $T(n)$ can be bounded asymptotically in three cases:

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

Quicksort

- Quick sort
- Correctness of partition
  - loop invariant
- Performance analysis
  - Recurrence relations

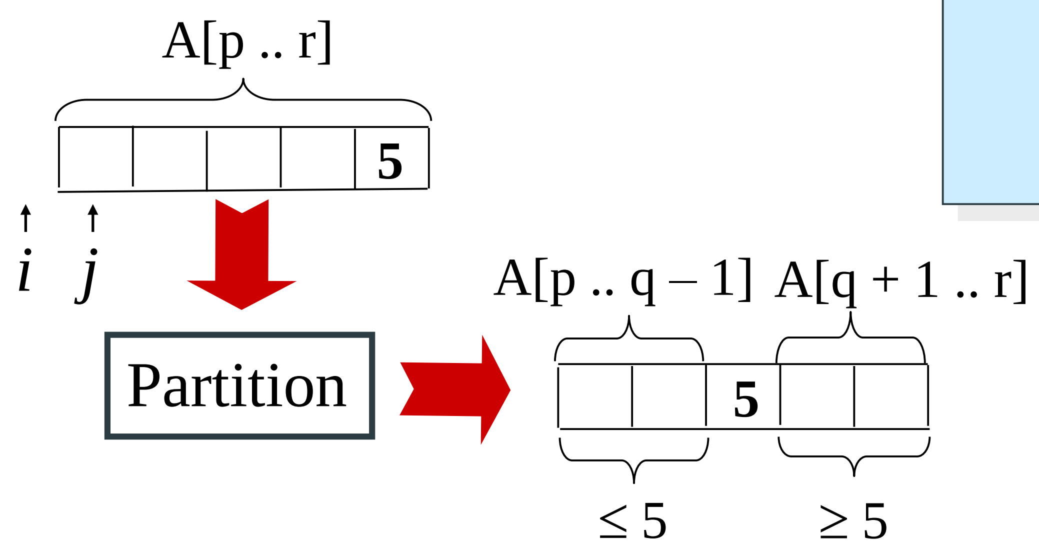
Design

- Follows the **divide-and-conquer** paradigm.
- ***Divide***: Partition (separate) the array $A[p \,..\, r]$ into two (possibly empty) subarrays $A[p \,..\, q\text{-}1]$ and $A[q\text{+}1 \,..\, r]$.
  - Each element in $A[p \,..\, q\text{-}1] \leq A[q]$.
  - $A[q] <$ each element in $A[q\text{+}1 \,..\, r]$.
  - Index q is often referred to as a pivot.
- ***Conquer***: Sort the two subarrays by recursive calls to quicksort.
- ***Combine***: The subarrays are sorted in place – no work is needed to combine them.
- How do the divide and combine steps of quicksort compare with those of merge sort?

Pseudocode

```
Quicksort(A, p, r)
    if p < r then
        q := Partition(A, p, r);
        Quicksort(A, p, q – 1);
        Quicksort(A, q + 1, r)
    fi
```

```
Partition(A, p, r)
    x, i := A[r], p – 1;
    for j := p to r – 1 do
        if A[j] ≤ x then
            i := i + 1;
            A[i] ↔ A[j]
        fi
    od;
    A[i + 1] ↔ A[r];
    return i + 1
```



Example

		p									r	
initially:		2	5	8	3	9	4	1	7	10	**6**	**note:** pivot (x) = 6
	i	j										
next iteration:		2	5	8	3	9	4	1	7	10	**6**	
		i	j									
next iteration:		2	5	8	3	9	4	1	7	10	**6**	
			i	j								
next iteration:		2	5	8	3	9	4	1	7	10	**6**	
			i		j							
next iteration:		2	5	3	8	9	4	1	7	10	**6**	
				i		j						

```
Partition(A, p, r)
    x, i := A[r], p – 1;
    for j := p to r – 1 do
        if A[j] ≤ x then
            i := i + 1;
            A[i] ↔ A[j]
        fi
    od;
    A[i + 1] ↔ A[r];
    return i + 1
```

Example (Continued)

next iteration:	2	5	3	8	9	4	1	7	10	**6**
			i		j					
next iteration:	2	5	3	8	9	4	1	7	10	**6**
			i			j				
next iteration:	2	5	3	4	9	8	1	7	10	**6**
				i			j			
next iteration:	2	5	3	4	1	8	9	7	10	**6**
					i			j		
next iteration:	2	5	3	4	1	8	9	7	10	**6**
					i				j	
next iteration:	2	5	3	4	1	8	9	7	10	**6**
					i					j
after final swap:	2	5	3	4	1	**6**	9	7	10	8
					i					j

```
Partition(A, p, r)
    x, i := A[r], p – 1;
    for j := p to r – 1 do
        if A[j] ≤ x then
            i := i + 1;
            A[i] ↔ A[j]
        fi
    od;
    A[i + 1] ↔ A[r];
    return i + 1
```

Partitioning

- Select the last element $A[r]$ in the subarray $A[p\,..\,r]$ as the *pivot* – the element around which to partition.
- As the procedure executes, the array is partitioned into four (possibly empty) regions.
  1. $A[p\,..\,i]$ — All entries in this region are ≤ ***pivot***.
  2. $A[i+1\,..\,j-1]$ — All entries in this region are > ***pivot***.
  3. $A[j\,..\,r-1]$ — Not known how they compare to *pivot*.
  4. $A[r]$ = *pivot*.
- The above hold before each iteration of the *for* loop, and constitute a *loop invariant*. (4 is not part of the LI - loop invariant.)

Correctness of Partition

Use loop invariant.

Initialization:

Before first iteration

A[*p*.. *i*] and *A*[*i* + 1 .. *j* - 1] are empty – Conds. 1 and 2 are satisfied (trivially).

r is the index of the *pivot* - Cond. 4 is satisfied.

Cond. 3 trivially holds.

Maintenance:

Case 1: *A*[*j*] > *x*

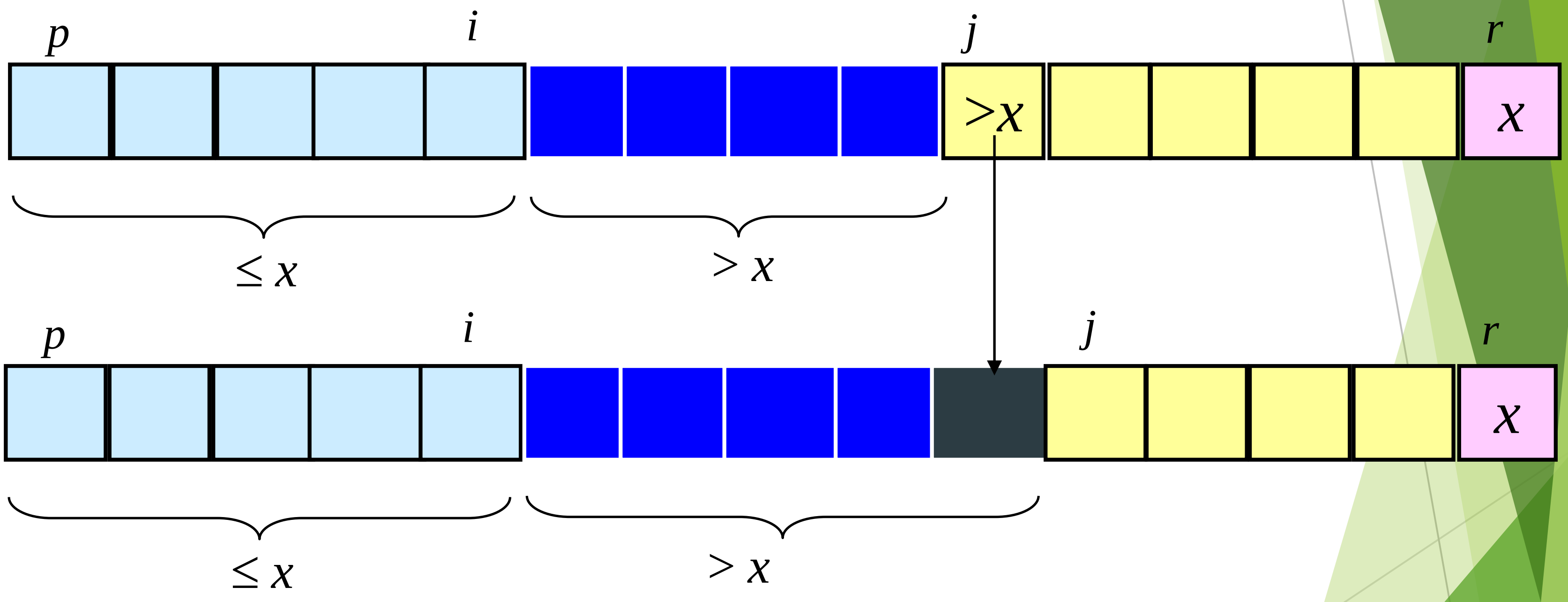
Increment *j* only.

LI is maintained.

```
Partition(A, p, r)
    x, i := A[r], p – 1;
    for j := p to r – 1 do
        if A[j] ≤ x then
            i := i + 1;
            A[i] ↔ A[j]
        fi
    od;
    A[i + 1] ↔ A[r];
    return i + 1
```

Correctness of Partition

Case 1: $A[j] > x$

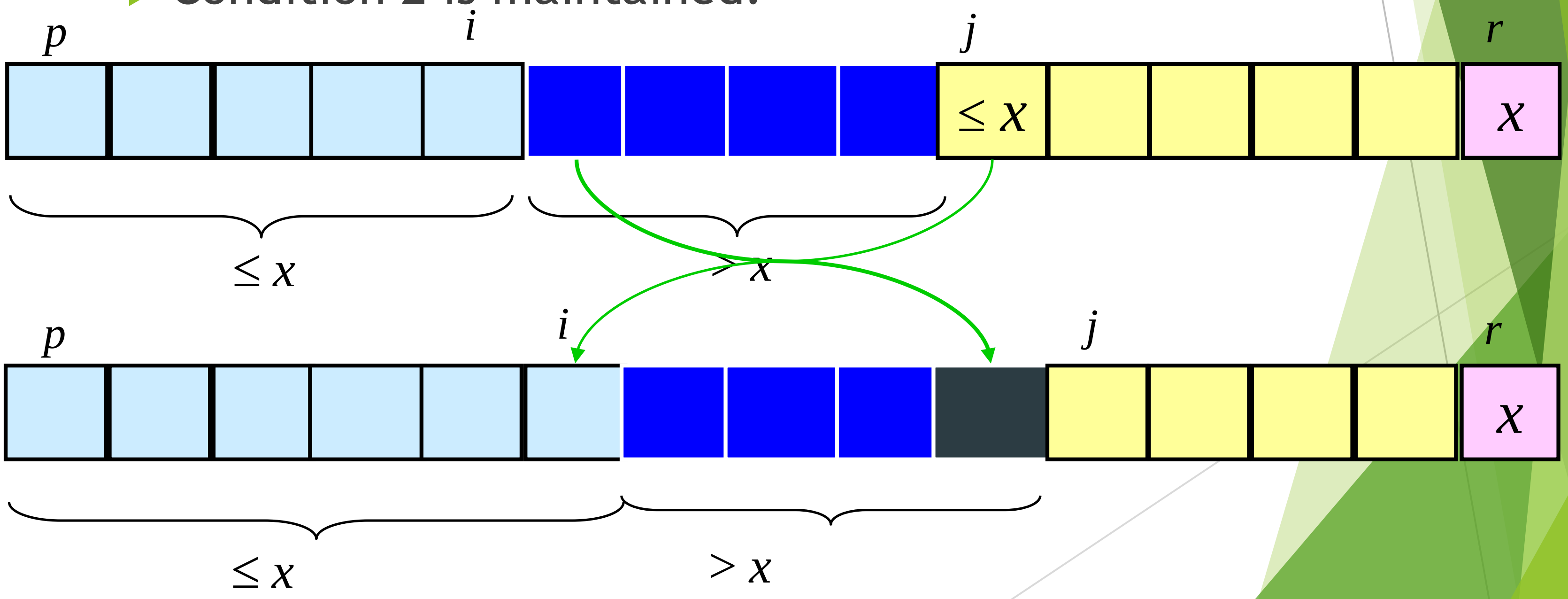


Correctness of Partition

- **Case 2:** $A[j] \leq x$
  - Increment i
  - Swap $A[i]$ and $A[j]$
    - Condition 1 is maintained.
  - Increment j
    - Condition 2 is maintained.

» $A[r]$ is unaltered.

- Condition 3 is maintained.



Correctness of Partition

Termination:

When the loop terminates, $j = r$, so all elements in A are partitioned into one of the three cases:

$A[p \,..\, i] \leq$ ***pivot***

$A[i + 1 \,..\, r - 1] >$ ***pivot***

$A[r] =$ ***pivot***

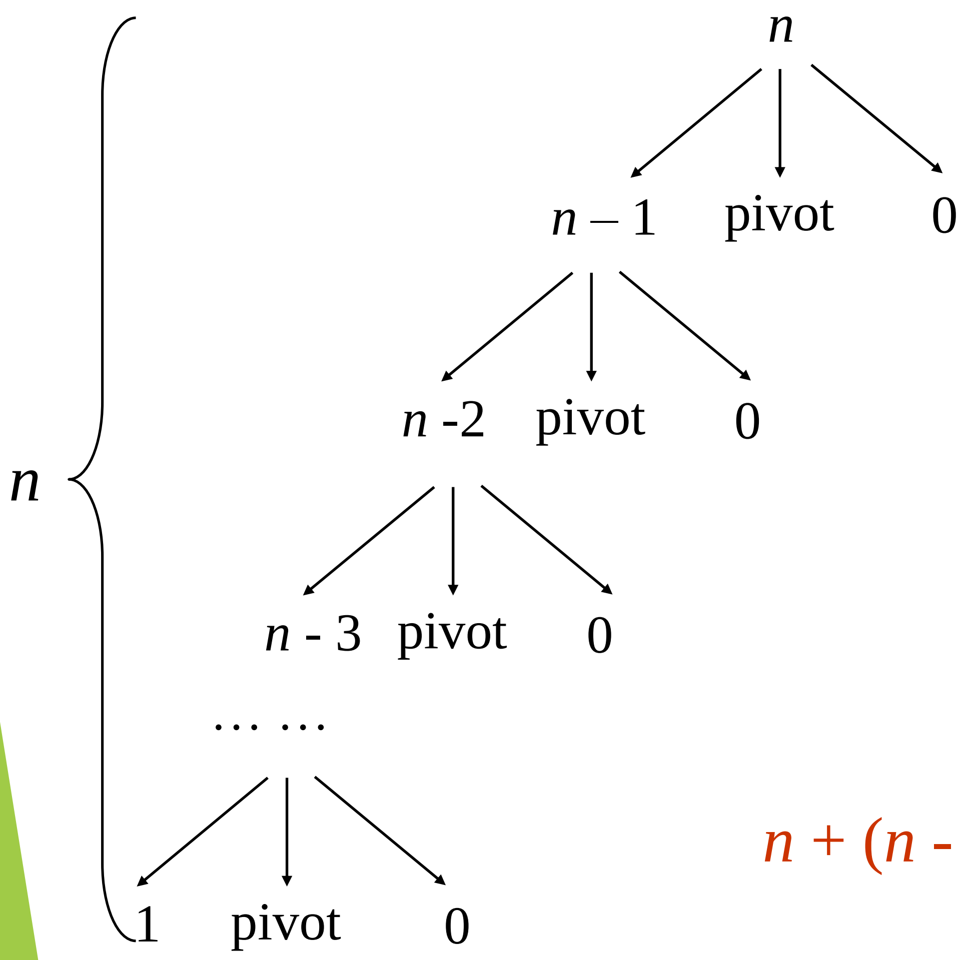
The last two lines swap $A[i + 1]$ and $A[r]$.

Pivot moves from the end of the array to between the two subarrays.

Thus, procedure *partition* correctly performs the divide step.

Worst-case Partition Analysis

Recursion tree for worst-case partition



Running time for worst-case partition at each recursive level:

$T(n) = T(n-1) + T(0) + \text{PartitionTime}(n)$

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

$= \sum_{k=1 \text{ to } n} \Theta(k)$

$= \Theta(\sum_{k=1 \text{ to } n} k)$

$= \Theta(n^2)$

$n + (n - 1) + \ldots + 1 = n(n + 1)/2 = O(n^2)$

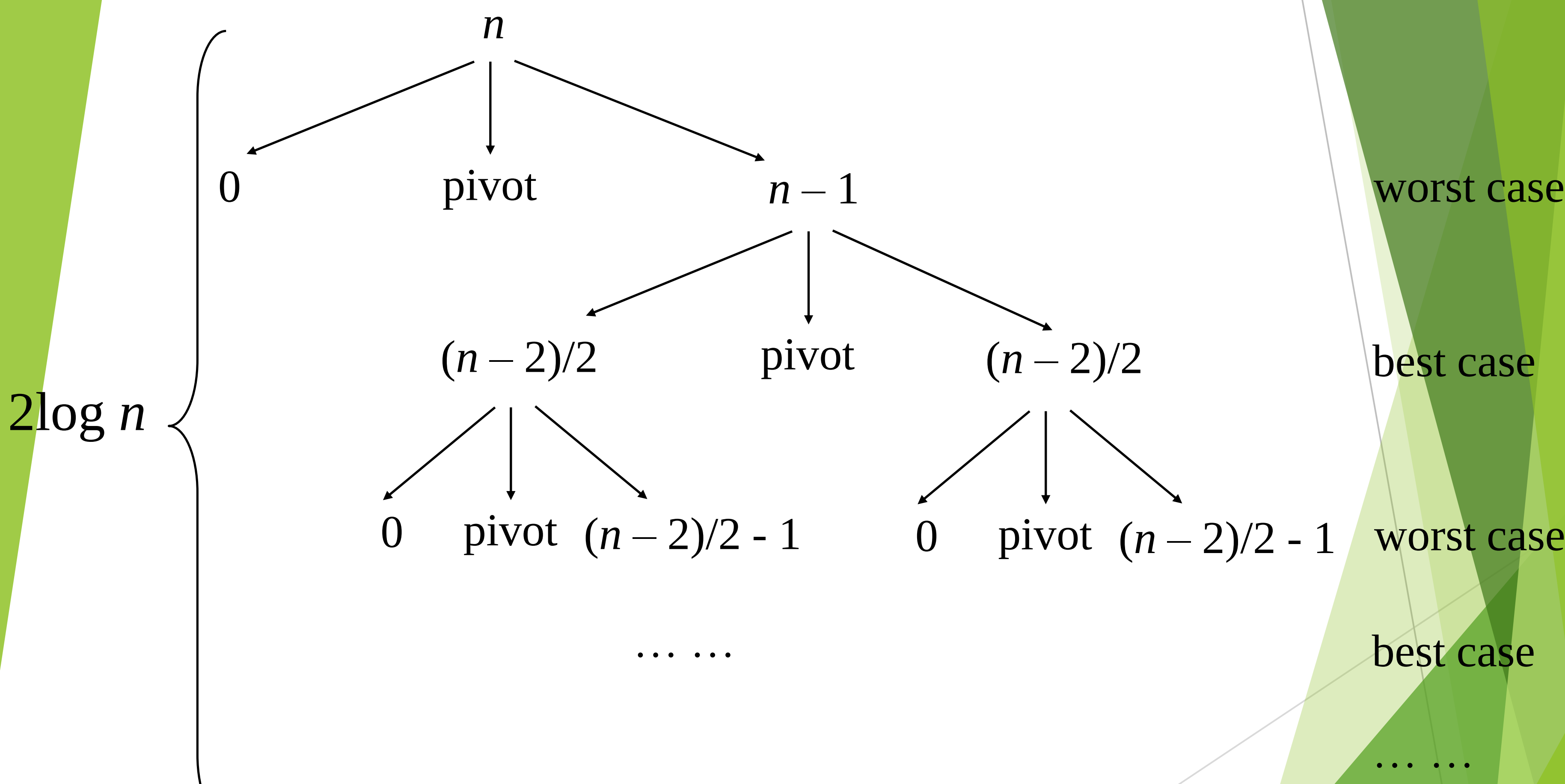
Best-case Partitioning

- Size of each subproblem $\leq n/2$.
  - One of the subproblems is of size $\lfloor n/2 \rfloor$
  - The other is of size $\lceil n/2 \rceil - 1$.
- Recurrence for running time
  - $T(n) \leq 2T(n/2) + \text{PartitionTime}(n)$

    $= 2T(n/2) + \Theta(n)$
- **$T(n) = \Theta(n \lg n)$**

Average-case Partitioning

Average case: Worst cases and best cases interleavingly appear.



Average case time complexity: $\leq 2n \log n$

Heapsort

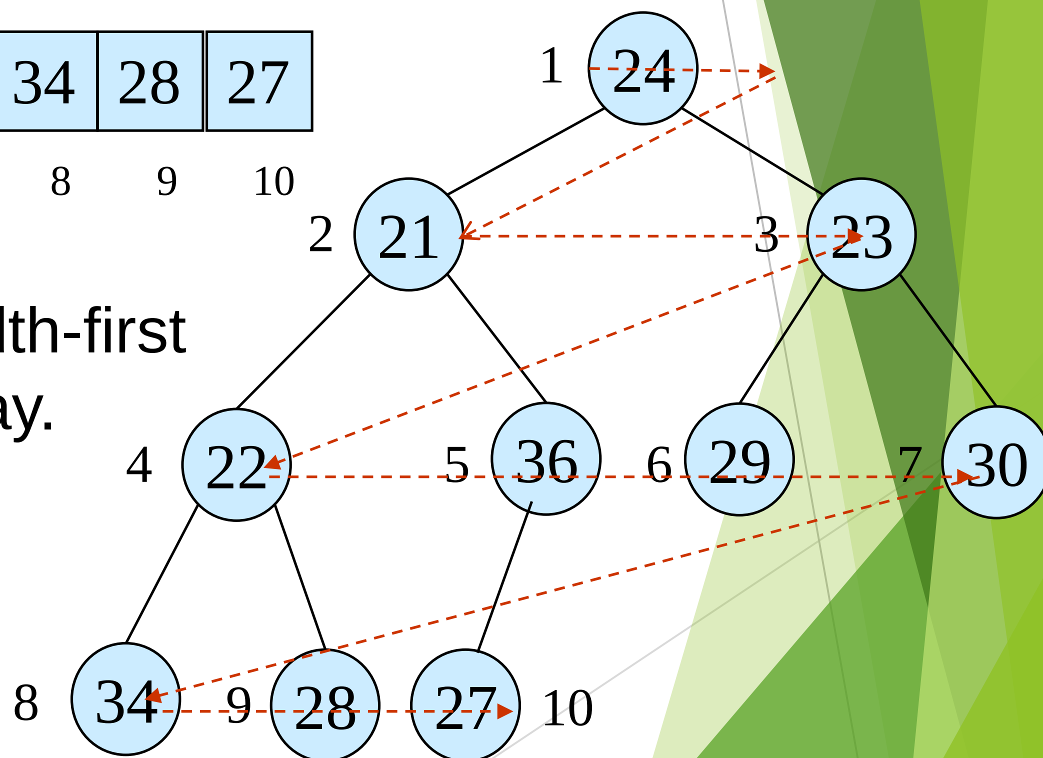
- What is a heap? Max-heap? Min-heap?
- Maintenance of Max-heaps
  - *MaxHeapify*
  - *BuildMaxHeap*
- Heapsort
  - Heapsort
  - Analysis
- Priority queues
  - Maintenance of priority queues

Data Structure Binary Heap

- length[*A*] – number of elements in array *A*.
- heap-size[*A*] – number of elements in heap stored in *A*.

heap-size[*A*] ≤ length[*A*]

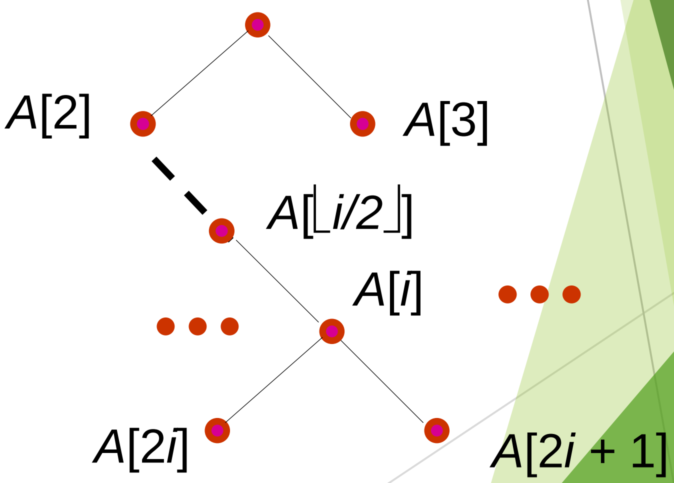
24	21	23	22	36	29	30	34	28	27
1	2	3	4	5	6	7	8	9	10



Searching the tree in breadth-first fashion, we will get the array.

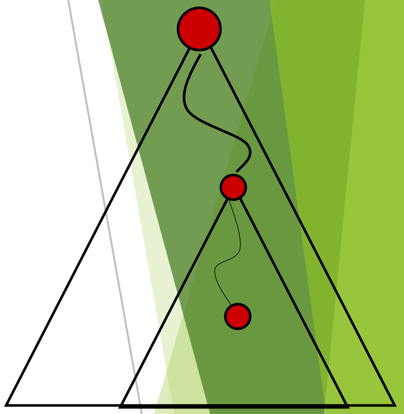
Data Structure Binary Heap

- Array viewed as a nearly complete binary tree.
  - Physically – linear array.
  - Logically – binary tree, filled on all levels (except lowest.)
- Map from array elements to tree nodes and vice versa
  - Root – *A*[1], Left[Root] – *A*[2], Right[Root] – *A*[3]
  - Left[*i*] – *A*[2*i*]
  - Right[*i*] – *A*[2*i*+1]
  - Parent[*i*] – $A[\lfloor i/2 \rfloor]$



Heap Property (Max and Min)

- Max-Heap
  - For every node excluding the root, the value stored in that node is at most that of its parent: $A[parent[i]] \geq A[i]$
- Largest element is stored at the root.
- In any subtree, no values are larger than the value stored at subtree's root.



- Min-Heap
  - For every node excluding the root, the value stored in that node is at least that of its parent: $A[parent[i]] \leq A[i]$
- Smallest element is stored at the root.
- In any subtree, no values are smaller than the value stored at subtree's root

Heapsort(*A*)

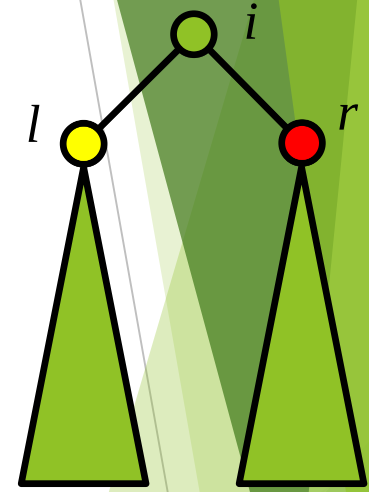
```
HeapSort(A)
Build-Max-Heap(A)
for i ← length[A] downto 2
    do exchange A[1] ↔ A[i]
        heap-size[A] ← heap-size[A] – 1
        MaxHeapify(A, 1)
```

Procedure MaxHeapify

<u>*MaxHeapify*(*A*, *i*)</u>

$l \leftarrow$ left(i) (* $A[l]$ is the left child of $A[i]$.*)
$r \leftarrow$ right(i)
if $l \leq$ *heap-size*[A] and $A[l] > A[i]$
    then *largest* $\leftarrow l$
    else *largest* $\leftarrow i$
if $r \leq$ *heap-size*[A] **and** $A[r] > A[largest]$
    then *largest* $\leftarrow r$
if *largest* $\neq i$
    then exchange $A[i] \leftrightarrow A[largest]$
MaxHeapify(A, *largest*)

<u>Assumption:</u>
Left(i) and Right(i) are max-heaps.



$A[largest]$ must be the largest among $A[i]$, $A[l]$ and $A[r]$.

Building a heap

Use *MaxHeapify* to convert an array *A* into a max-heap.

<u>How?</u>

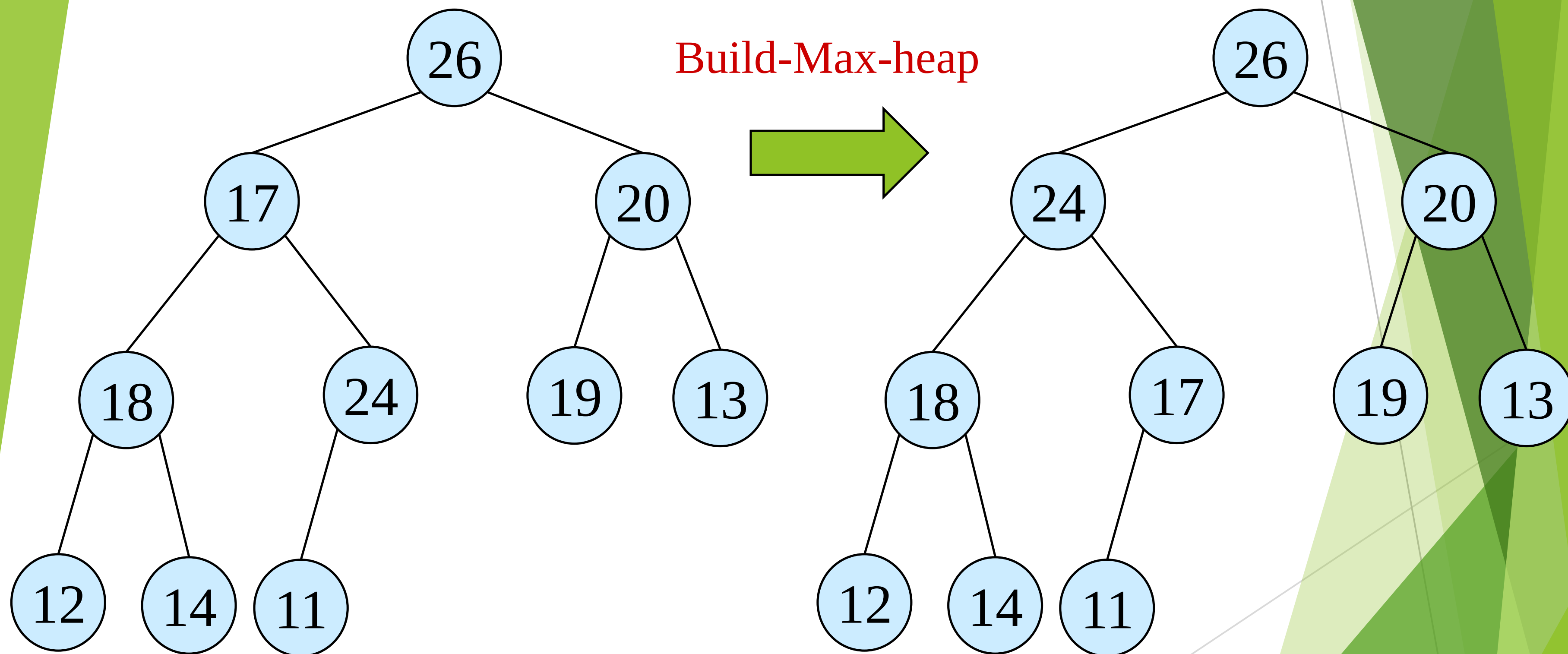
Call MaxHeapify on each element in a bottom-up manner.

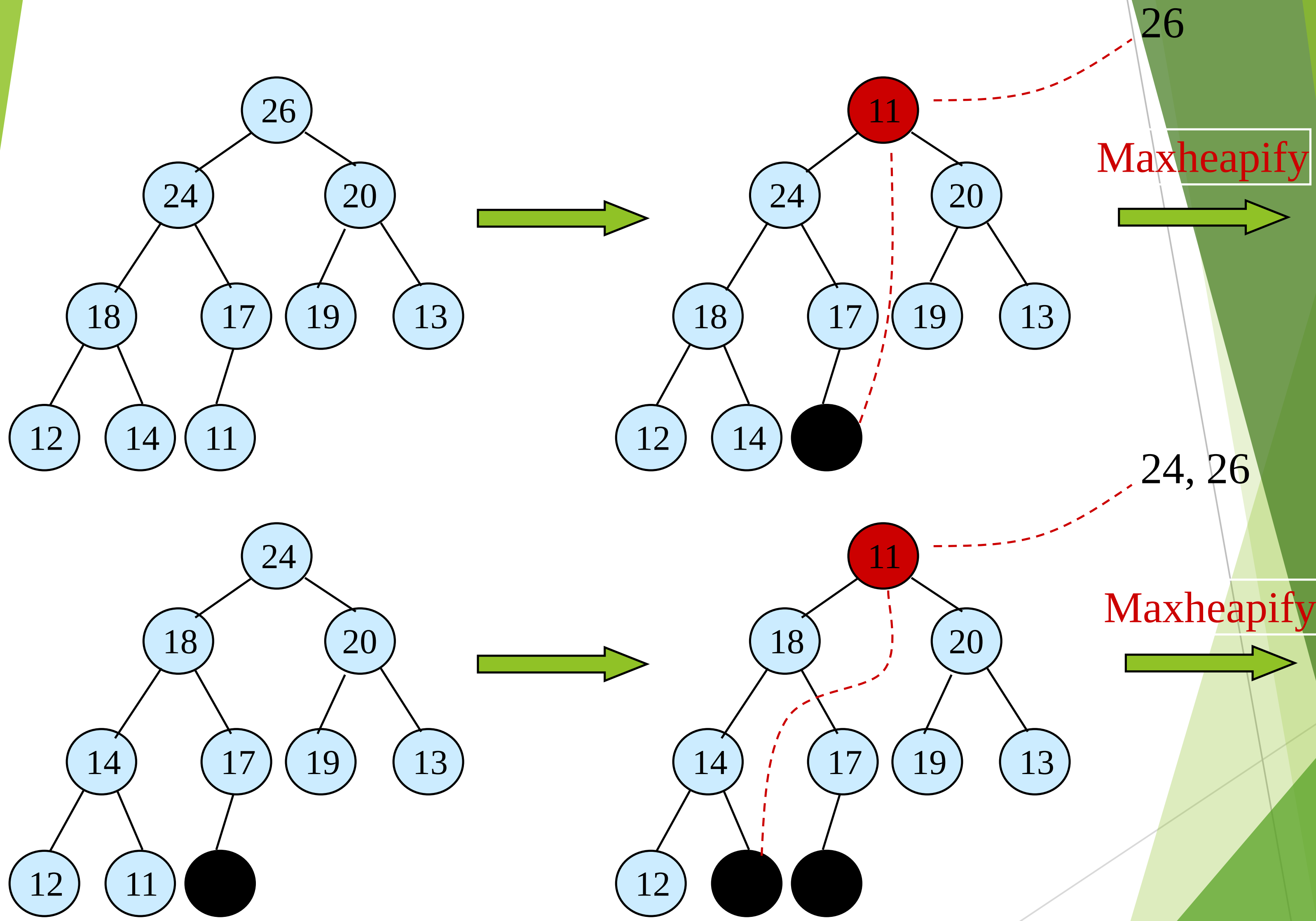
<u>*BuildMaxHeap*(*A*)</u>

1. *heap-size*[*A*] $\leftarrow$ *length*[*A*]
2. **for** *i* $\leftarrow \lfloor length[A]/2 \rfloor$ **downto** 1 (*$A[\lfloor length[A]/2 \rfloor +1]$, $A[\lfloor length[A]/2 \rfloor +2]$, … are leaf nodes.*)
3. **do** *MaxHeapify*(*A*, *i*)

Heapsort – Example

26	17	20	18	24	19	13	12	14	11
1	2	3	4	5	6	7	8	9	10




26
24
20
18
17
19
13
12
14
11
11
24
20
18
17
19
13
12
14
26
Maxheapify
24
18
20
14
17
19
13
12
11
11
18
20
14
17
19
13
12
24, 26
Maxheapify

Running Time of *BuildMaxHeap*

Tighter Bound for *T*(*BuildMaxHeap*)

T(*BuildMaxHeap*)

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

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \le \sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$$

$x = 1$ in (A.8)

$$O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(n)$$

Can build a heap from an unordered array in linear time.

Priority Queue

- Popular & important application of heaps.
- Max and min priority queues.
- Maintains a *dynamic* set *S* of elements.
- Each set element has a *key* – an associated value.
- Goal is to support insertion and extraction efficiently.
- Applications:
  - Ready list of processes in operating systems by their priorities – the list is highly dynamic
  - In event-driven simulators to maintain the list of events to be simulated in order of their time of occurrence.

Basic Operations

Operations on a max-priority queue:

- Insert(S, x) - inserts the element x into the queue S

  $S \leftarrow S \cup \{x\}$.

- Maximum(S) - returns the element of S with the largest key.

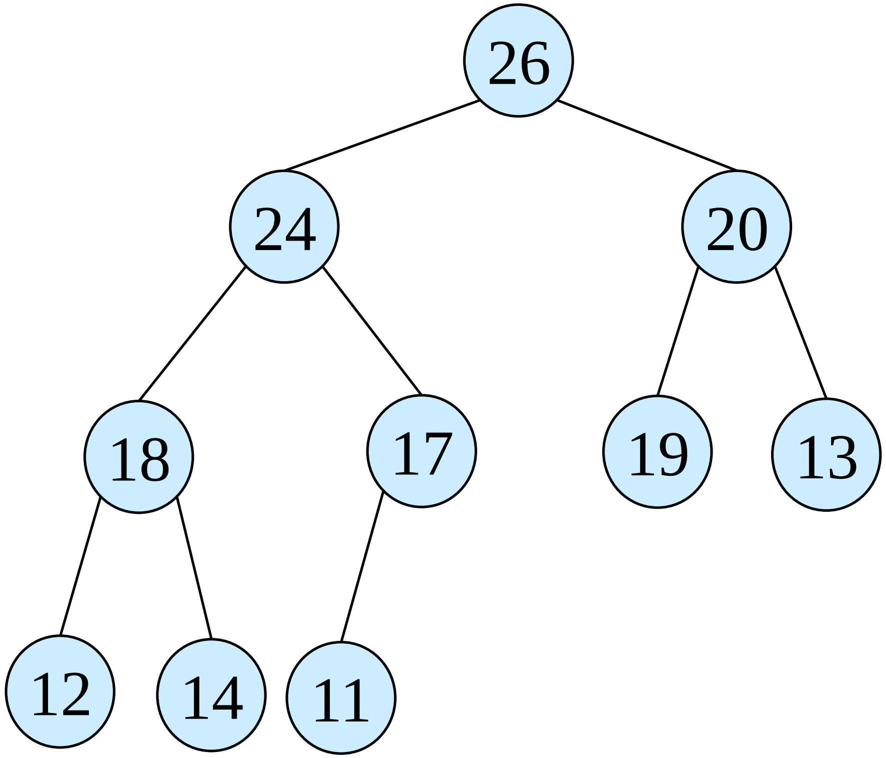
- Extract-Max(S) - removes and returns the element of S with the largest key.

- Increase-Key(S, x, k) – increases the value of element x's key to the new value k.

Min-priority queue supports Insert, Minimum, Extract-Min, and Decrease-Key.

Heap gives a good compromise between fast insertion but slow extraction and vice versa.

Priority queue as max-heap:



Heap-Extract-Max(*A*)

Implements the Extract-Max operation.

```
Heap-Extract-Max(A)
if heap-size[A] < 1
    then error “heap underflow”
max ← A[1]
A[1] ← A[heap-size[A]]
heap-size[A] ← heap-size[A] - 1
MaxHeapify(A, 1)
return max
```

Running time : Dominated by the running time of MaxHeapify
= $O(\lg n)$

Heap-Insert(*A*, *key*)

```
Heap-Insert(A, key)
heap-size[A] ← heap-size[A] + 1
i ← heap-size[A]
while i > 1 and A[Parent(i)] < key
        do A[i] ← A[Parent(i)]
                i ← Parent(i)
A[i] ← key
```

Running time is $O(\lg n)$

The path traced from the new leaf to the root has length $O(\lg n)$.

Heap-Increase-Key(*A*, *i*, *key*)

```
Heap-Increase-Key(A, i, key)
If key < A[i]
   then error "new key is smaller than the current key"
A[i] ← key
while i > 1 and  A[Parent[i]] < A[i]
        do exchange A[i] ↔ A[Parent[i]]
            i ← Parent[i]
```

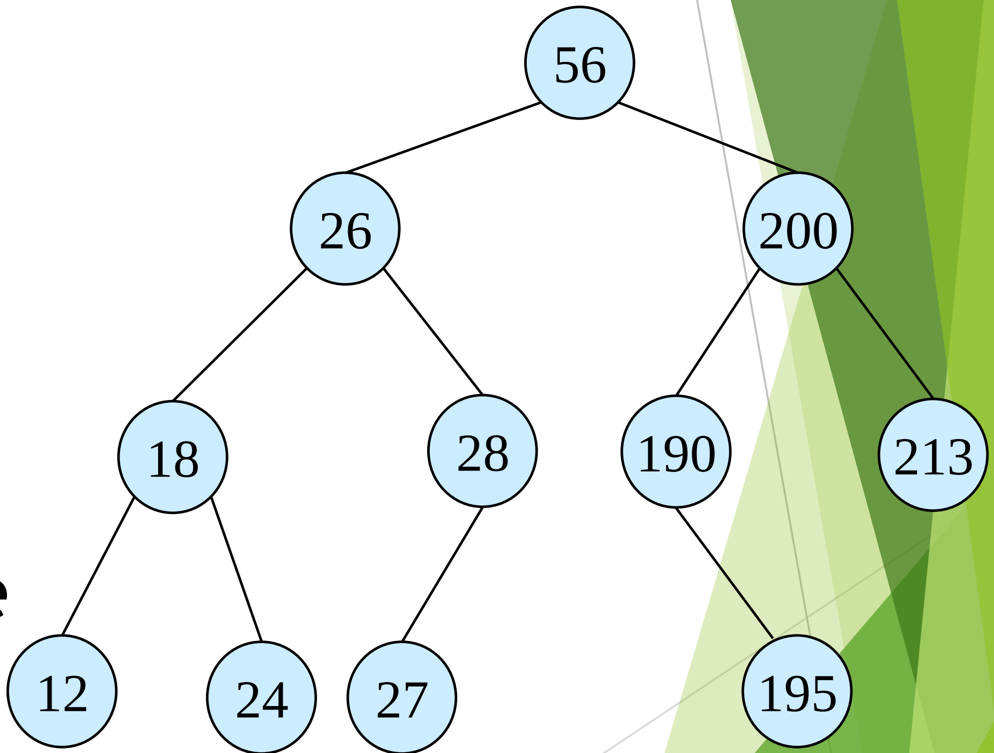
```
Heap-Insert(A, key)
heap-size[A] ← heap-size[A] + 1
A[heap-size[A]] ← – ∞
Heap-Increase-Key(A, heap-size[A], key)
```

Binary Search Trees

- What is a binary search tree?
- Tree searching
- Inorder traversal of a binary search tree
- Find Min & Max
- Predecessor and successor
- BST insertion and deletion

Binary Search Tree

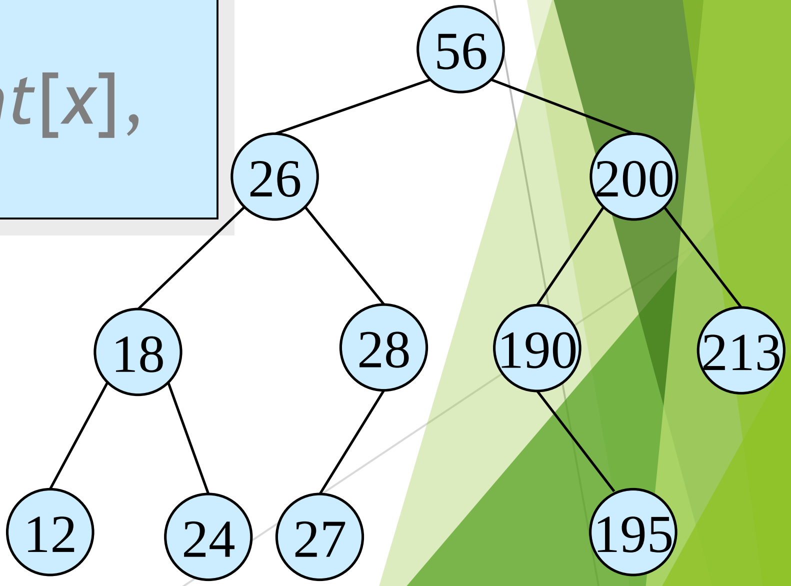
- Stored keys must satisfy the *binary search tree* property.
  - » ∀ y in left subtree of x, then $key[y] < key[x]$.
  - » ∀ y in right subtree of x, then $key[y] \geq key[x]$.



Tree Search

Tree-Search(x, k)

1. **if** x = NIL *or* k = $key[x]$
2. **then** return x
3. **if** $k < key[x]$
4. **then** return Tree-Search($left[x]$, k)
5. **else** return Tree-Search($right[x]$, k)



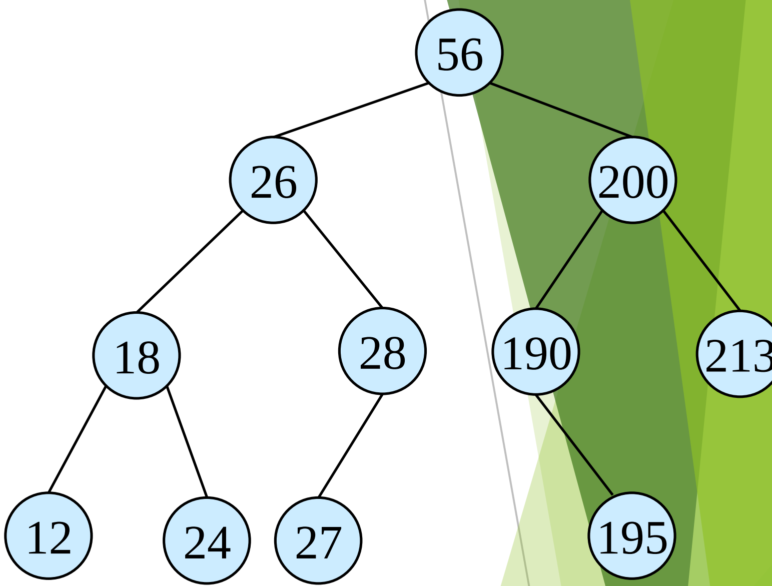
Running time: $O(h)$

Aside: tail-recursion

Iterative Tree Search

Iterative-Tree-Search(x, k)

```
while x ≠ NIL and k ≠ key[x]
    do if k < key[x]
        then x ← left[x]
        else x ← right[x]
return x
```



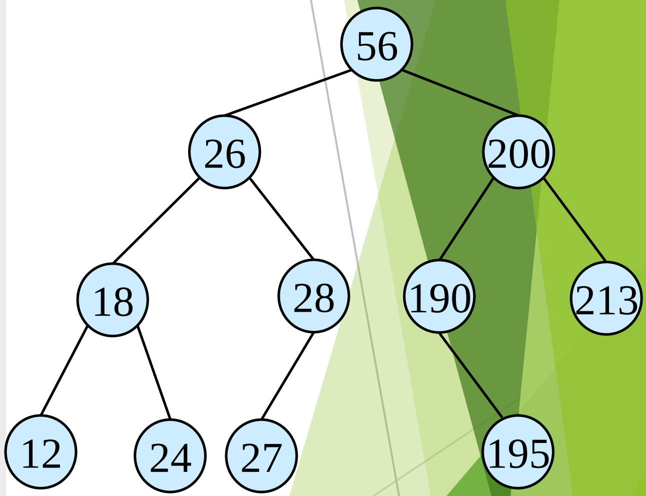
The iterative tree search is more efficient on most computers.
The recursive tree search is more straightforward.

Inorder Traversal

The binary-search-tree property allows the keys of a binary search tree to be printed, in (monotonically increasing) order, recursively.

Inorder-Tree-Walk (*x*)

if $x \neq$ NIL
then Inorder-Tree-Walk(*left*[*x*])
    print *key*[*x*]
    Inorder-Tree-Walk(*right*[*x*])



Finding Min & Max

- The binary-search-tree property guarantees that:
  - » The minimum is located at the left-most node.
  - » The maximum is located at the right-most node.

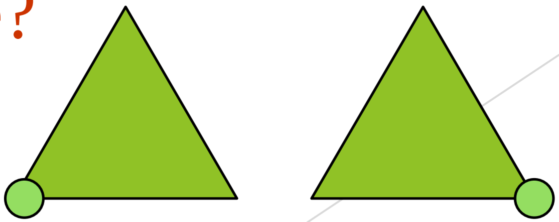
Tree-Minimum(*x*)

1. **while** *left*[*x*] $\neq$ *NIL*
2. **do** *x* $\leftarrow$ *left*[*x*]
3. **return** *x*

Tree-Maximum(*x*)

1. **while** *right*[*x*] $\neq$ *NIL*
2. **do** *x* $\leftarrow$ *right*[*x*]
3. **return** *x*

Q: How long do they take?



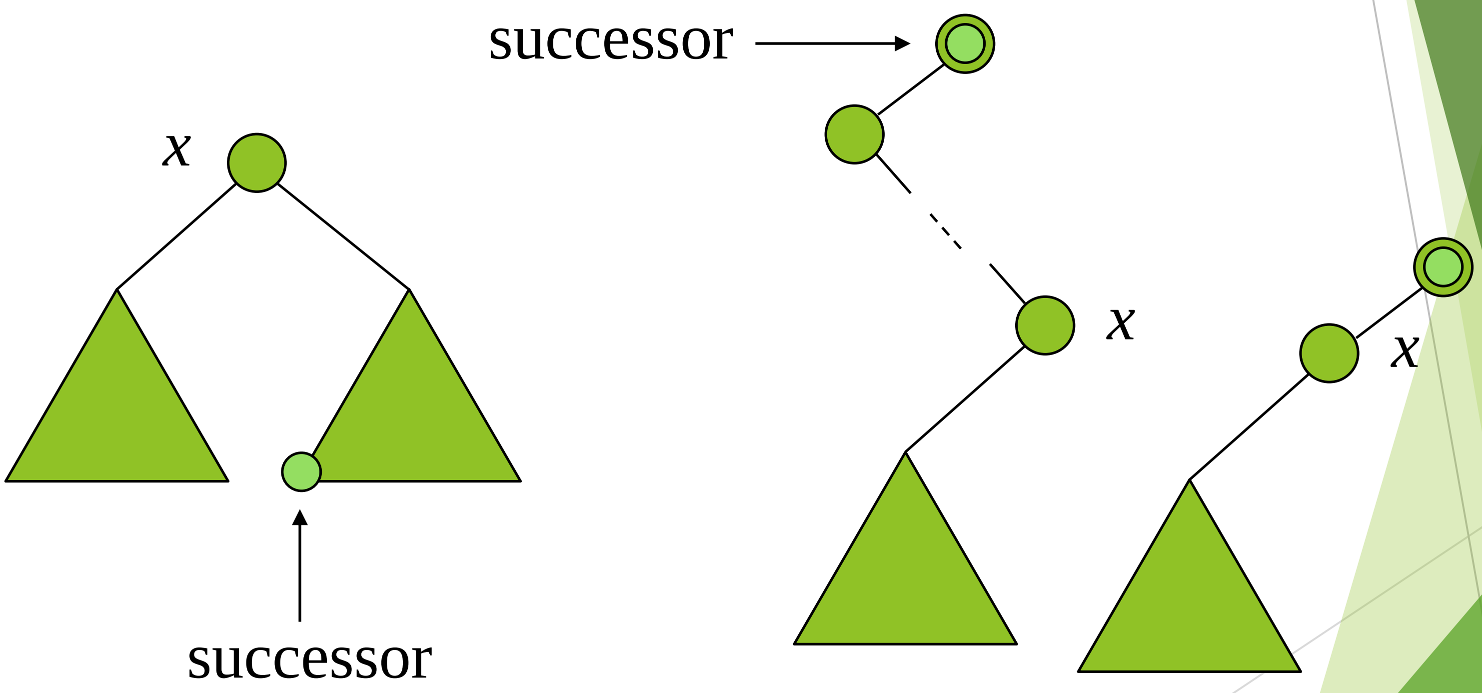
Predecessor and Successor

- Predecessor of node *x* is the node *y* such that *key*[*y*] is the greatest key smaller than *key*[*x*].
- Successor of node *x* is the node *y* such that *key*[*y*] is the smallest key greater than *key*[*x*].
- The successor of the largest key is NIL.
- Search consists of two cases.
  - If node *x* has a non-empty right subtree, then *x*'s successor is the minimum in the right subtree of *x*.
  - If node *x* has an empty right subtree, then:
    - As long as we move to the left up the tree (move up through right children), we are visiting smaller keys.
    - *x*'s successor *y* is the node that is the predecessor of *x* (*x* is the maximum in *y*'s left subtree).
    - In other words, *x*'s successor *y*, is the lowest ancestor of *x* whose left child is also an ancestor of *x* or is *x* itself.

Successor

Case 1: x has a non-empty right subtree.

Case 2: x has an empty right subtree.

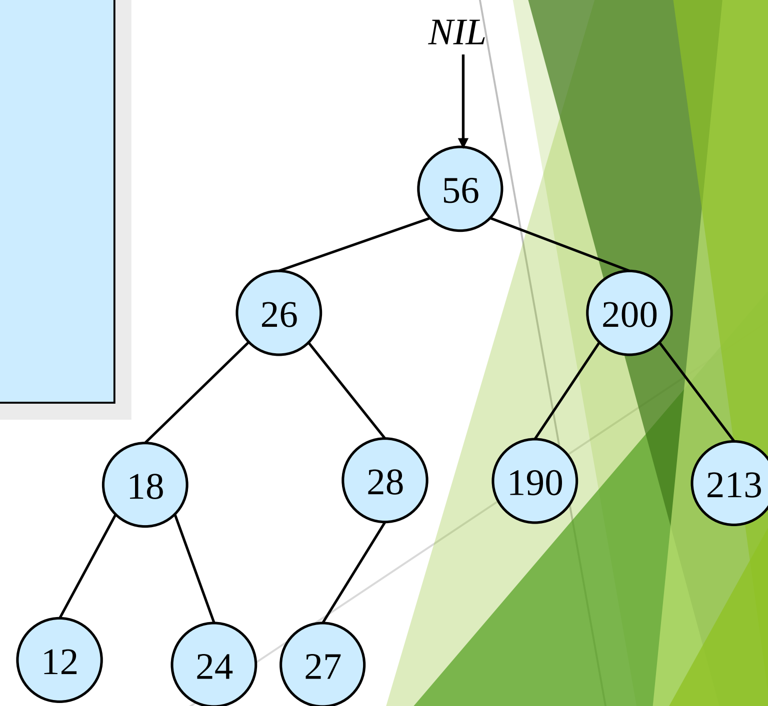


Pseudo-code for Successor

```
Tree-Successor(x)
if right[x] ≠ NIL
    then return Tree-Minimum(right[x])
y ← p[x]
while y ≠ NIL and x = right[y]
do x ← y
    y ← p[y]
return y
```

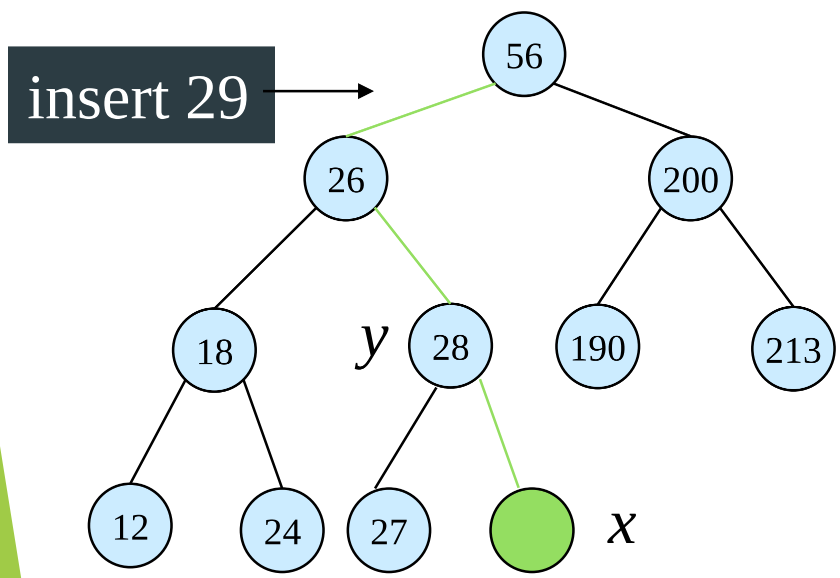
Code for ***predecessor*** is symmetric.

Running time: $O(h)$



BST Insertion – Pseudocode

- Change the dynamic set represented by a BST.
- Ensure the binary-search-tree property holds after change.
- Insertion is easier than deletion.



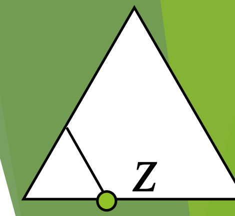
<u>Tree-Insert(*T*, *z*)</u>

```
y ← NIL
x ← root[T]
while x ≠ NIL
    do y ← x
        if key[z] < key[x]
            then x ← left[x]
            else x ← right[x]
p[z] ← y
if y = NIL
    then root[T] ← z
    else if key[z] < key[y]
        then left[y] ← z
        else right[y] ← z
```

Tree-Delete (T, z)

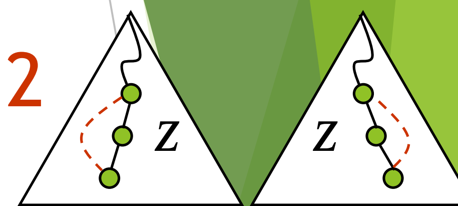
if z has no children ♦ case 1

then remove z



if z has one child ♦ case 2

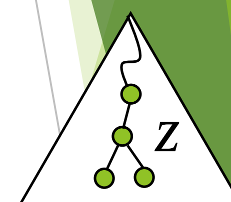
then make $p[z]$ point to child



if z has two children (subtrees) ♦ case 3

then swap z with its successor

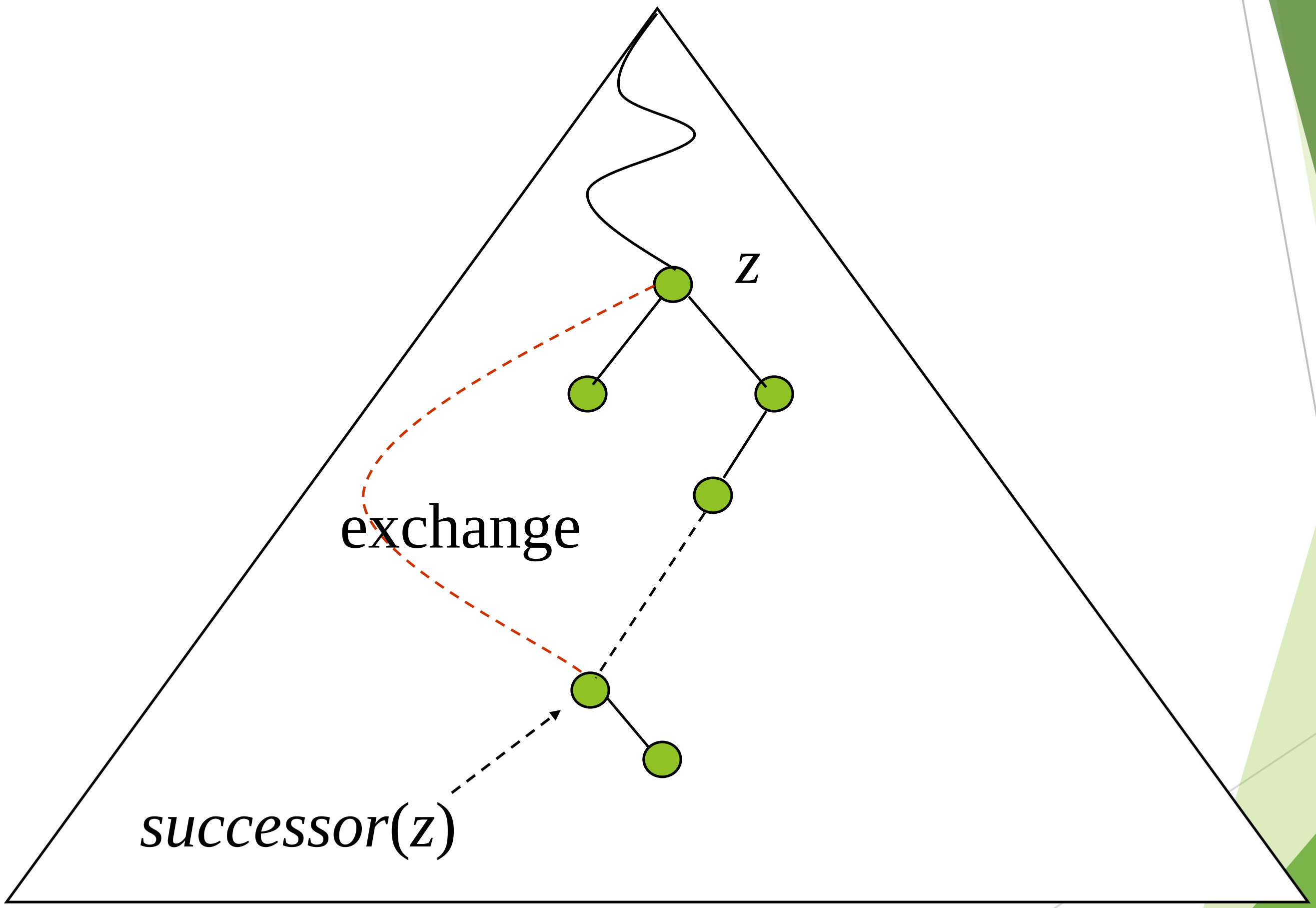
perform case 1 or case 2 to delete it



⇒ TOTAL: $O(h)$ time to delete a node

Tree-Delete (T, z)

Illustration for case 3:



Deletion – Pseudocode

```
Tree-Delete(T, z)
/* Determine which node to splice out: either z or z's successor.
    */
    if left[z] = NIL or right[z] = NIL
        then y ← z                              /*Case 1 or Case 2*/
        else y ← Tree-Successor[z]      /*Case 3*/
/* Set x to a non-NIL child of y, or to NIL if y has no children. */
    if  left[y] ≠ NIL                /*y has one child or no child.*/
         then x ← left[y]           /*x can be a child of y or NIL.*/
         else x ← right[y]
/* y is removed from the tree by manipulating pointers of  p[y]
    and x */
    if x ≠ NIL
        then p[x] ← p[y]
/* Continued on next slide */
```

y is the node be deleted, which has at most one child.

x is the unique child of *y*.

Deletion – Pseudocode

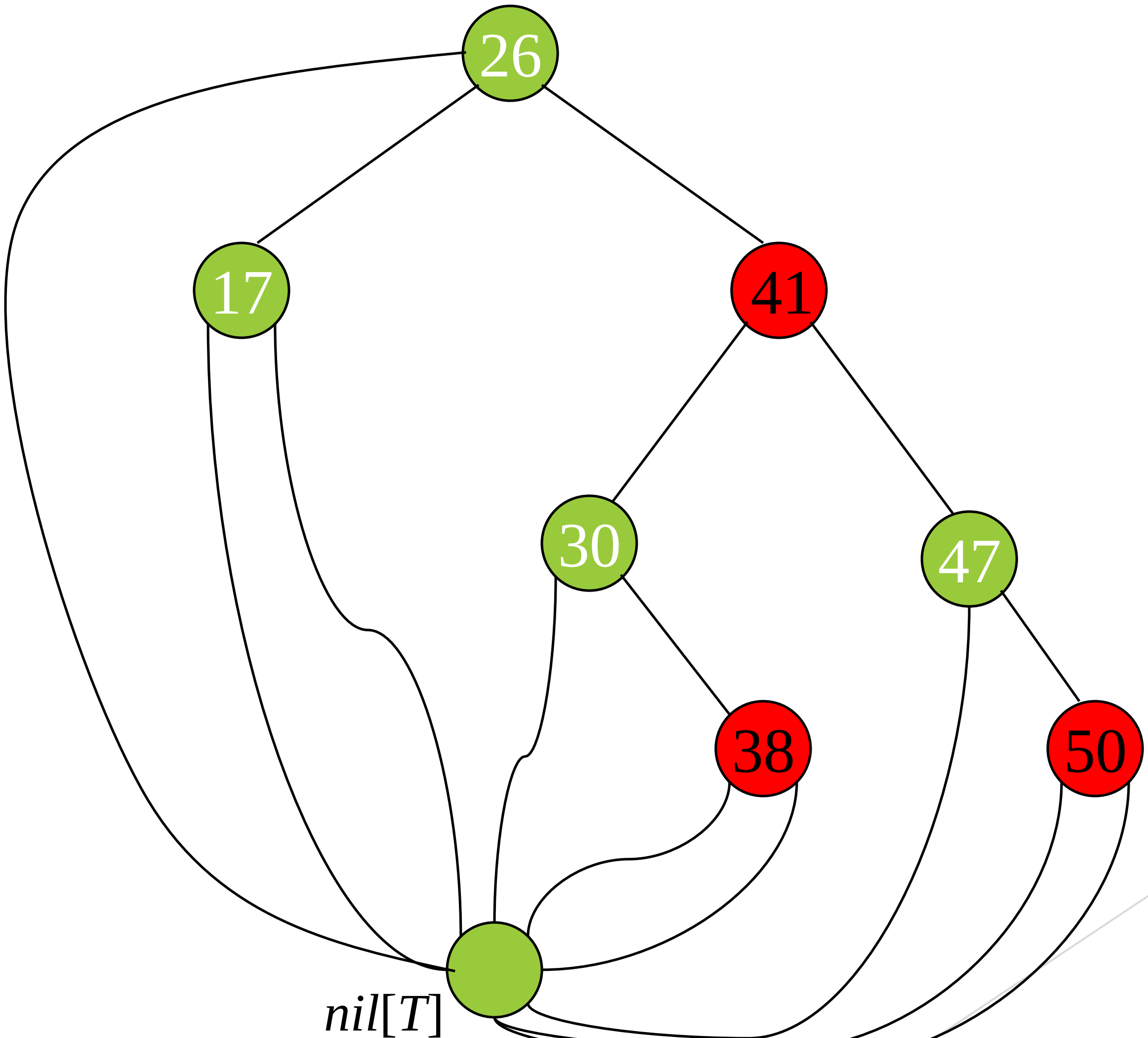
Tree-Delete(*T*, *z*) (Contd. from previous slide)

```
if p[y] = NIL                              /*if y is the root*/
    then root[T] ← x
    else if  y = left[p[y]]                /*y is a left child.*/
            then left[p[y]] ← x
            else right[p[y]] ← x
/* If z's successor was spliced out, copy its data into z */
if y ≠ z                                   /*y is z's successor.*/
    then  key[z] ← key[y]
              copy y's satellite data into z.
return y
```

Red-Black Trees

- What is a red-black tree?
  - node color: red or black
  - *nil*[*T*] and black height
- Subtree rotation
- Node insertion
- Node deletion

Red-black Tree – Example



Red-black Properties

1. Every node is either red or black.
2. The root is black.
3. Every *virtual* leaf (*nil*) is black.
4. If a node is red, then both its children are black.

5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

Height of a Red-black Tree

- Height of a node:
  - $h(x)$ = number of edges in a longest path to a leaf.
- Black-height of a node x, $bh(x)$:
  - $bh(x)$ = number of black nodes (including $nil[T]$) on the path from x to leaf, not counting x.
- Black-height of a red-black tree is the black-height of its root.
  - By Property 5, black height is well defined.

Height of a Red-black Tree

Example:

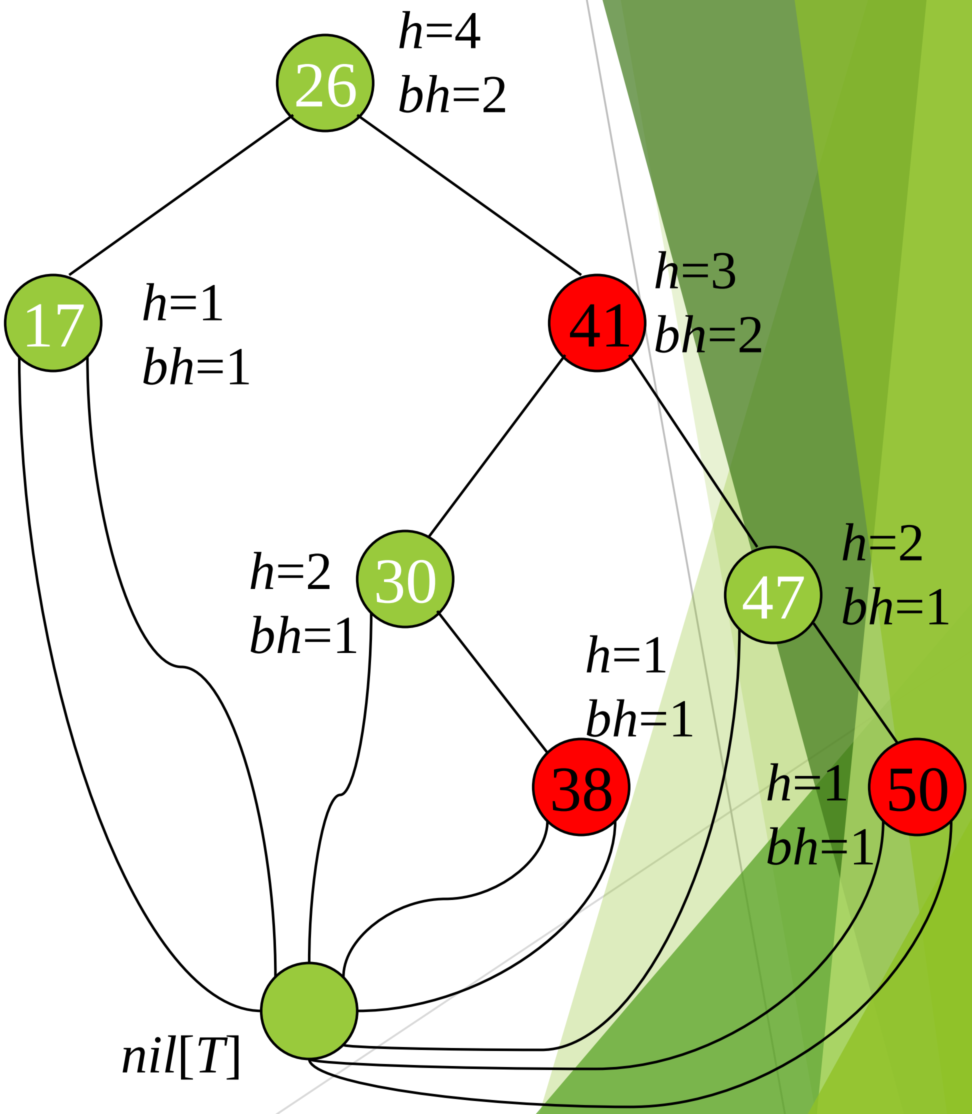
Height of a node:

h(*x*) = # of edges in a longest path to a leaf.

Black-height of a node *bh*(*x*) = # of black nodes on path from *x* to leaf, not counting *x*.

How are they related?

$bh(x) \le h(x) \le 2bh(x)$



Lemma "RB Height"

Consider a node x in an RB tree: The longest descending path from x to a leaf has length $h(x)$, which is at most twice the length of the shortest descending path from x to a leaf.

Proof:

black nodes on any path from x = $bh(x)$ (prop 5)

$\leq$ # nodes on shortest path from x, $s(x)$. (prop 1)

But, there are no consecutive red (prop 4),

and we end with black (prop 3), so $h(x) \leq 2\ bh(x)$.

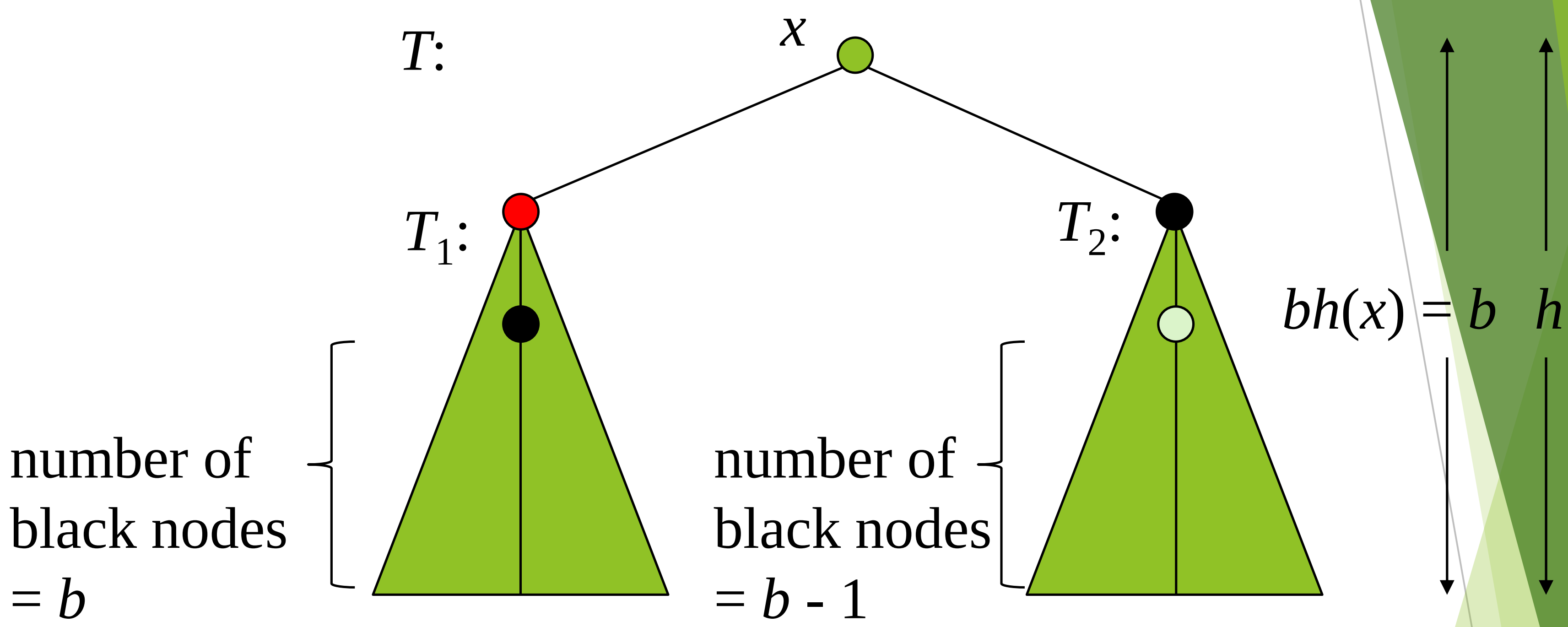
Thus, $h(x) \leq 2s(x)$. **QED**

Bound on RB Tree Height

- **Lemma:** The subtree rooted at any node x has $\geq 2^{bh(x)}-1$ internal nodes.
- **Proof:** By induction on height of x, $h(x)$.
  - Base Case: Height $h(x) = 0 \Rightarrow x$ is a leaf $\Rightarrow bh(x) = 0$. Subtree has $2^0-1 = 0$ nodes.
  - Induction Step: **Assume that for any node with height $< h$ the lemma holds.**

    Consider node x with $h(x) = h > 0$ and $bh(x) = b$.
    - Each child of x has height at most h - 1 and black-height either b (child is red) or b - 1 (child is black).
    - By ind. hyp., each child has $\geq 2^{bh(x)-1} - 1$ internal nodes.
    - Subtree rooted at x has $\geq 2(2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ internal nodes. (The +1 is for x itself.)

Bound on RB Tree Height



number of internal nodes of T_1 $> 2^b$-1

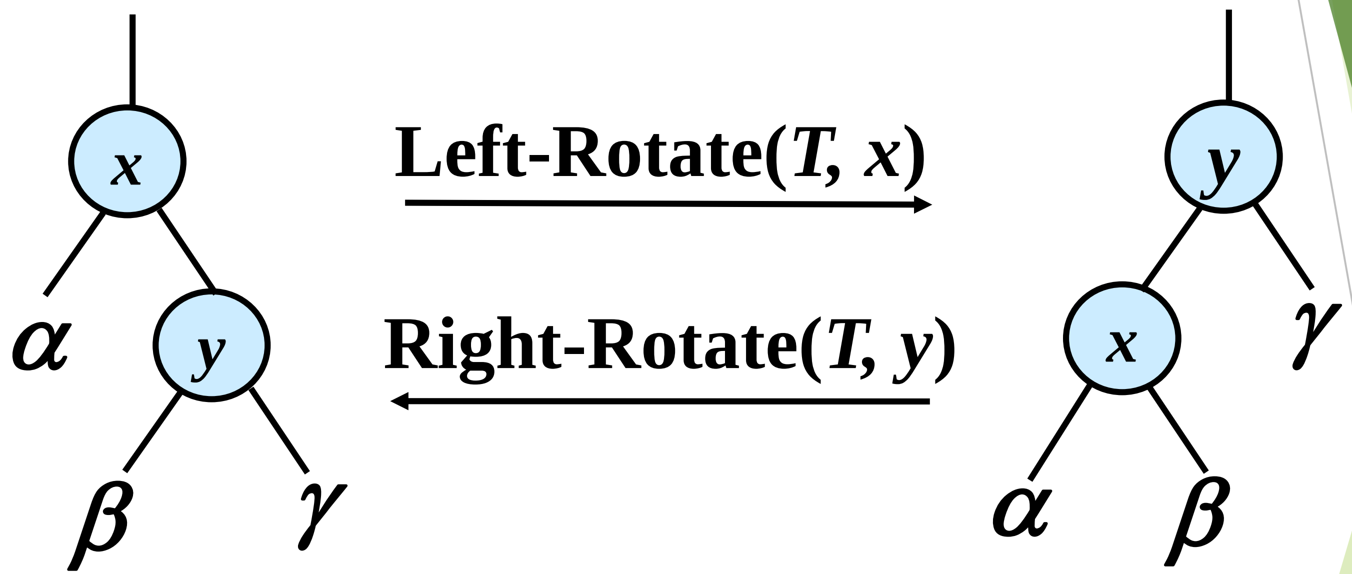
number of internal nodes of T_2 $> 2^{b-1}$-1

$$\text{number of internal nodes of } T \geq |T_1| + |T_2| + 1$$
$$\geq (2^b - 1) + (2^{b-1} - 1) + 1 \geq 2^b - 1$$

Bound on RB Tree Height

- Lemma: The subtree rooted at any node x has $\geq 2^{bh(x)}-1$ internal nodes.
- **Lemma 13.1:** A red-black tree with *n* internal nodes has height at most 2lg (*n*+1).
- **Proof:**
  - By the above lemma, $n \geq 2^{bh} - 1$,
  - and since $bh \geq h/2$, we have $n \geq 2^{h/2} - 1$.
  - $\Rightarrow \; h \leq 2\lg(n + 1)$.

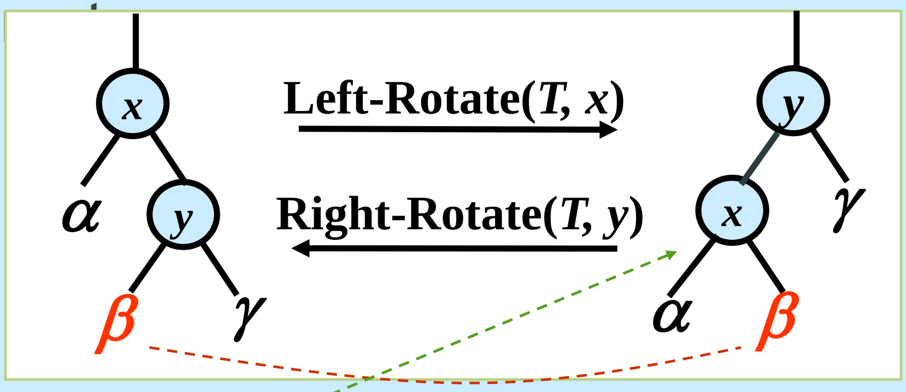
Rotations



Left Rotation – Pseudo-code

Left-Rotate (*T, x*)

```
y ← right[x]   // Set y.
right[x] ← left[y]   //Turn y's left subtree β into x's right
subtree.
if left[y] ≠ nil[T]
    then p[left[y]] ← x //Set x to be the parent of left[y] = β.
p[y] ← p[x]          //Link x's parent to y.
if p[x] = nil[T]     //If x is the
    then root[T] ← y
    else if x = left[p[x]]
        then left[p[x]] ← y
        else right[p[x]] ← y
left[y] ← x          // Put x as y's left child.
p[x] ← y
```



Insertion in RB Trees

- Insertion must preserve all red-black properties.
- Should an inserted node be colored Red? Black?
- Basic steps:
  - Use Tree-Insert from BST (slightly modified) to insert a node *z* into *T*.
    - Procedure **RB-Insert(z)**.
  - Color the node *z* red.
  - Fix the modified tree by re-coloring nodes and performing rotation to preserve RB tree property.
    - Procedure **RB-Insert-Fixup**.

Insertion

RB-Insert(*T*, *z*)

```
y ← nil[T]
x ← root[T]
while x ≠ nil[T]
    do y ← x
        if key[z] < key[x]
            then x ← left[x]
            else x ← right[x]
p[z] ← y
if y = nil[T]
    then root[T] ← z
    else if key[z] < key[y]
        then left[y] ← z
        else right[y] ← z
```

RB-Insert(*T*, *z*) Contd.

```
left[z] ← nil[T]
right[z] ← nil[T]
color[z] ← RED
RB-Insert-Fixup(T, z)
```

How does it differ from the Tree-Insert procedure of BSTs?

Which of the RB properties might be violated?

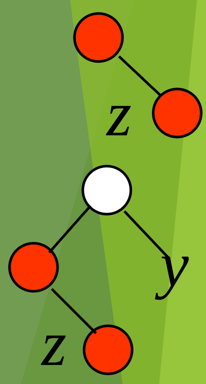
Fix the violations by calling RB-Insert-Fixup.

Insertion – Fixup

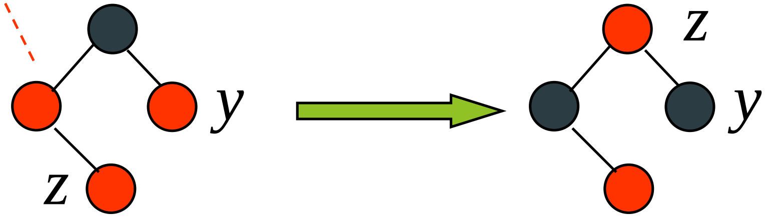
RB-Insert-Fixup (T, z)

z's parent is the left child of its own parent

```
while color[p[z]] = RED
    do if p[z] = left[p[p[z]]]  //for cases 1 – 3
        then y ← right[p[p[z]]]
            if color[y] = RED
                then color[p[z]] ← BLACK        // Case 1
                     color[y] ← BLACK           // Case 1
                     color[p[p[z]]] ← RED       // Case 1
                     z ← p[p[z]]                // Case 1
```



Case 1:

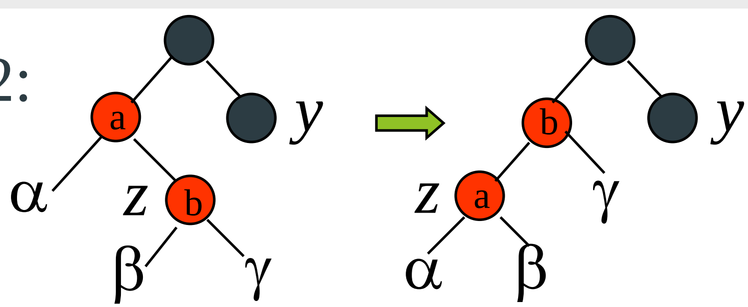


← Change this node to red to keep the number of black nodes not increased

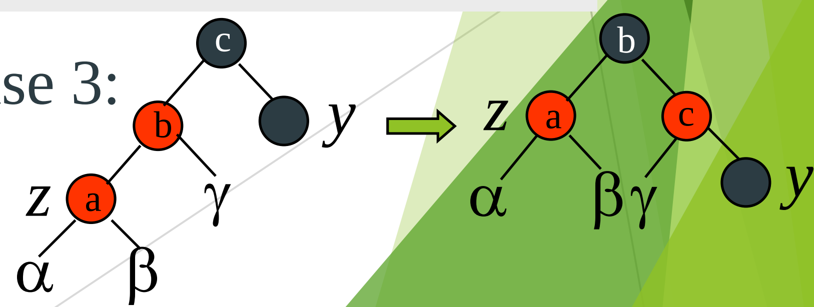
Insertion - Fixup

```
RB-Insert-Fixup(T, z) (Contd.)
9.                else if z = right[p[z]]          // color[y] ≠ RED
10.                    then z ← p[z]                  // Case 2
11.                         LEFT-ROTATE(T, z)         // Case 2
12.                    color[p[z]] ← BLACK            // Case 3
13.                    color[p[p[z]]] ← RED           // Case 3
14.                    RIGHT-ROTATE(T, p[p[z]])       // Case 3
15.         else (if p[z] = right[p[p[z]]]) (for cases 4 – 6, same
16.              as 3-14 with "right" and "left" exchanged)
17. color[root[T ]] ← BLACK
```

Case 2:



Case 3:



Deletion

- Deletion, like insertion, should preserve all the RB properties.
- The properties that may be violated depends on the color of the deleted node.
  - Red – OK. Why?
  - Black?
- Steps:
  - Do regular BST deletion.
  - Fix any violations of RB properties that may be caused by a deletion.

Deletion

RB-Delete(*T*, *z*)

```
if left[z] = nil[T] or right[z] = nil[T]
    then y ← z
    else y ← TREE-SUCCESSOR(z)
if left[y] ≠ nil[T ]
    then x ← left[y]
    else x ← right[y]
p[x] ← p[y]    // Do this, even if x is nil[T]
```

Deletion

RB-Delete (*T, z*) (Contd.)

```
if p[y] = nil[T]
    then root[T] ← x
    else if y = left[p[y]] (*if y is a left child.*)
            then left[p[y]] ← x
            else right[p[y]] ← x  (*if y is a right child.*)
if y ≠ z
    then key[z] ← key[y]
         copy y's satellite data into z
if color[y] = BLACK
    then RB-Delete-Fixup(T, x)
return y
```

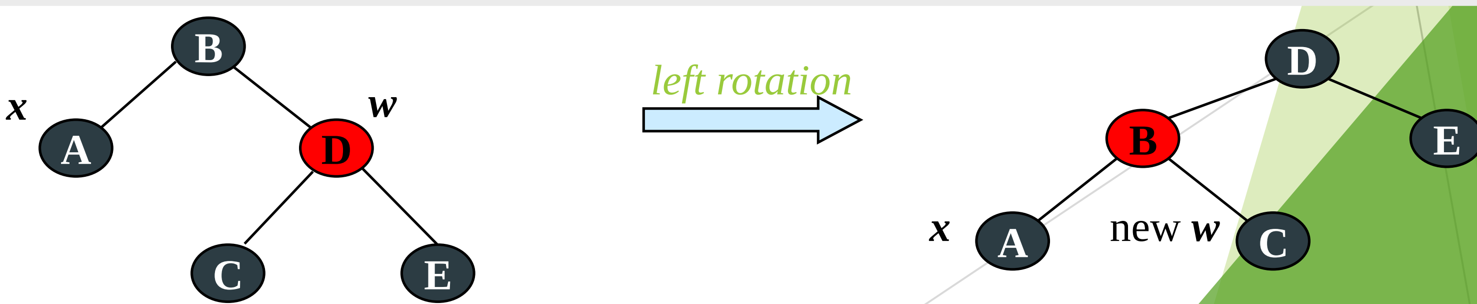
The node passed to the fixup routine is the only child of the spliced up node, or the sentinel.

Deletion – Fixup

RB-Delete-Fixup(*T*, *x*)

```
while x ≠ root[T] and color[x] = BLACK
    do if x = left[p[x]]                          // for cases 1 - 4
        then w ← right[p[x]]
            if color[w] = RED                     // Case 1
                then color[w] ← BLACK             // Case 1
                    color[p[x]] ← RED             // Case 1
                    LEFT-ROTATE(T, p[x])          // Case 1
                    w ← right[p[x]]               // Case 1
```

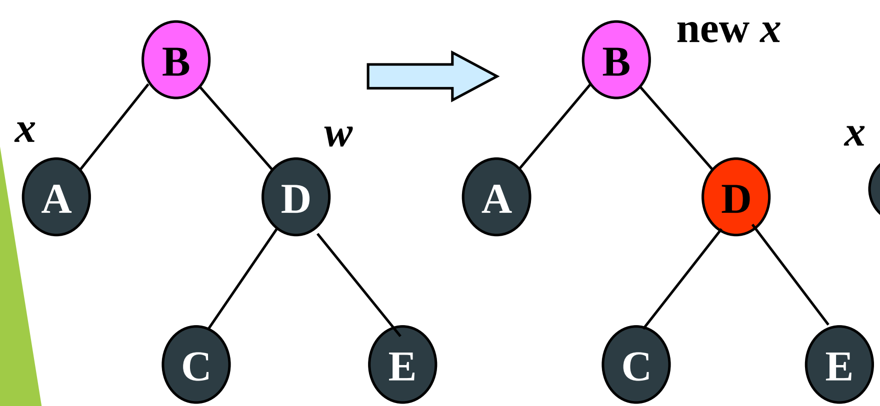
Case 1:



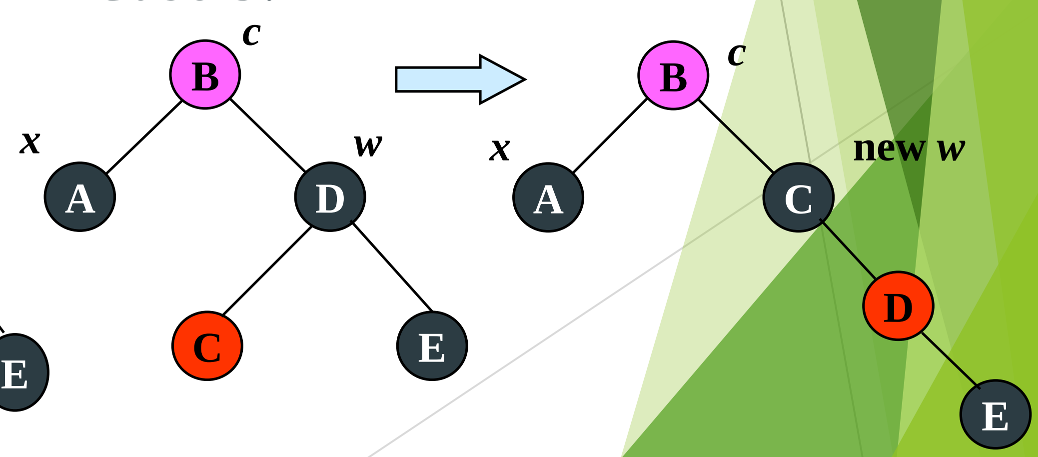
RB-Delete-Fixup(*T*, *x*) (Contd.)

```
        /* x is still left[p[x]] */
        if color[left[w]] = BLACK and color[right[w]] = BLACK
            then color[w] ← RED                               // Case 2
                 x ← p[x]                                     // Case 2
        else if color[right[w]] = BLACK                       // Case 3
                 then color[left[w]] ← BLACK                  // Case 3
                      color[w] ← RED                          // Case 3
                      RIGHT-ROTATE(T, w)                      // Case 3
                 w ← right[p[x]]                              // Case 3
```

Case 2:



Case 3:



RB-Delete-Fixup(*T*, *x*) (Contd.)

/* *x* is still *left*[*p*[*x*]] */

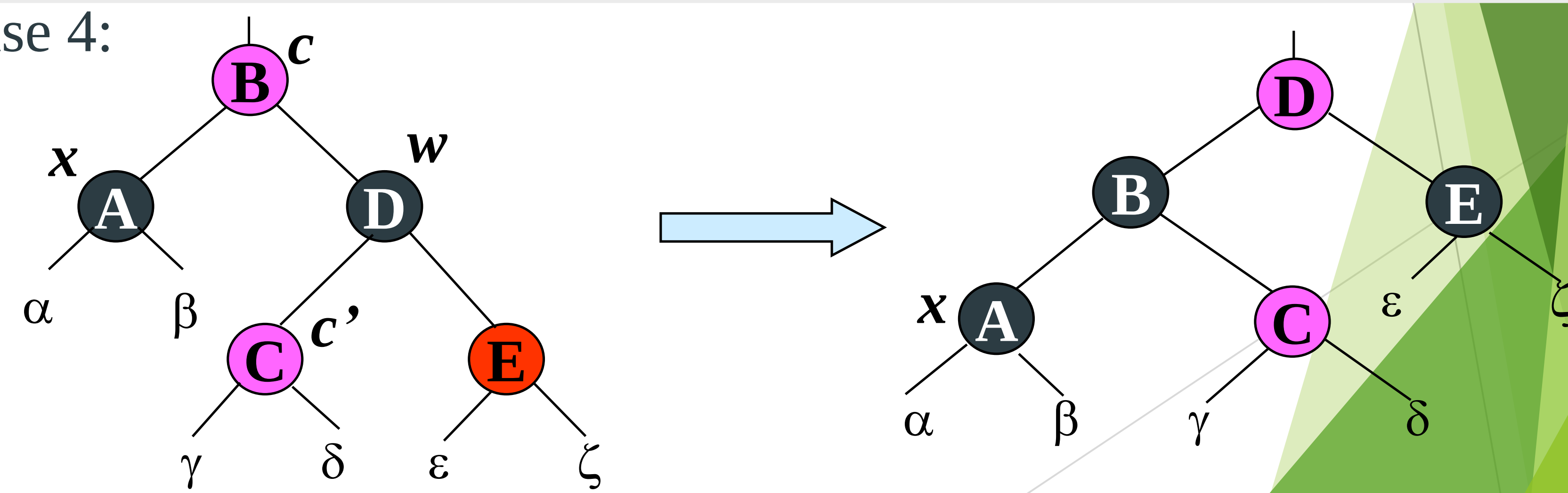
17. *color*[*w*] ← *color*[*p*[*x*]] // Case 4
18. *color*[*p*[*x*]] ← BLACK // Case 4
19. *color*[*right*[*w*]] ← BLACK // Case 4
20. LEFT-ROTATE(*T*, *p*[*x*]) // Case 4
21. *x* ← *root*[*T*] // Case 4

22. **else** (for cases 5 – 8, same as lines 3 - 21 with “right” and “left” exchanged)

23. *color*[*x*] ← BLACK

to go out the while-loop

Case 4:



Elementary Graph Algorithms

- Graph representation
- Graph traversal
  - Breadth-first search
  - Depth-first search
- Parenthesis theorem

Graphs

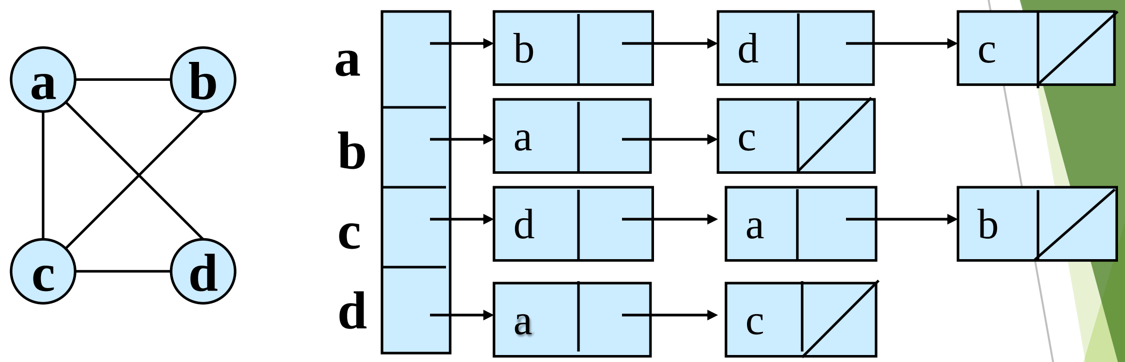
- Types of graphs
  - » Undirected: edge $(u, v) = (v, u)$; for all v, $(v, v) \notin E$ (No self loops.)
  - » Directed: (u, v) is edge from u to v, denoted as $u \rightarrow v$. Self loops are allowed.
  - » Weighted: each edge has an associated weight, given by a weight function $w : E \rightarrow R$. (R – set of all possible real numbers)
  - » Dense: $|E| \approx |V|^2$.
  - » Sparse: $|E| << |V|^2$.
- $|E| = O(|V|^2)$

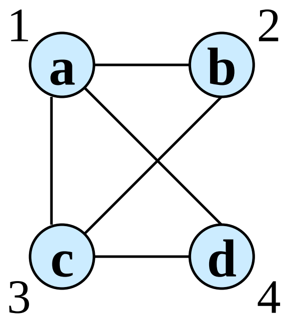
Graphs

- If $(u, v) \in E$, then vertex v is adjacent to vertex u.
- Adjacency relationship is:
  - Symmetric if G is undirected.
  - Not necessarily so if G is directed.
- If an undirected graph G is connected:
  - There is a path between every pair of vertices.
  - $|E| \geq |V| - 1$.
  - Furthermore, if $|E| = |V| - 1$, then G is a *tree*.
- If a directed graph G is connected:
  - Its undirected version is connected.
- Other definitions in Appendix B (B.4 and B.5) as needed.

Representation of Graphs

- Two standard ways.
  - Adjacency Lists.
  - Adjacency Matrix.





	1	2	3	4
1	0	1	1	1
2	1	0	1	0
3	1	1	0	1
4	1	0	1	0

Storage Requirement

- For directed graphs:
  - Sum of lengths of all adj. lists is

    $\sum_{v \in V}$ out-degree(v) = $\sum_{v \in V}$ in-degree(v) = $|E|$

    No. of edges leaving v
  - Total storage: $\Theta(|V| + |E|)$
- For undirected graphs:
  - Sum of lengths of all adj. lists is

    $\sum_{v \in V}$ degree(v) = $2|E|$

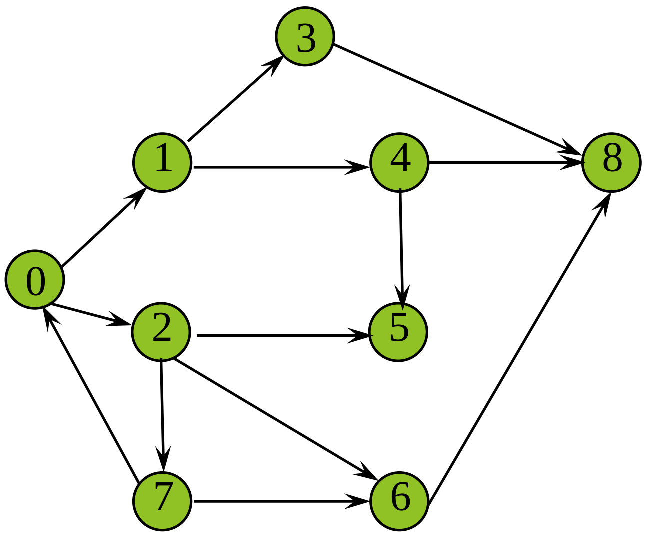
    No. of edges incident on v. Edge (u,v) is incident on vertices u and v.
  - Total storage: $\Theta(|V| + |E|)$

Sparse Matrix

	0	1	2	3
0	3	0	1	0
1	0	0	0	0
2	0	2	4	1
3	1	0	0	1

	row0		row2			row3	
Nonzero values data[ ]	{3	1	2	4	1	1	1}
Column indeces col_index[ ]	{0	2	1	2	3	0	3}
Row pointers row_ptr[ ]	{0	2	2	5	7}		

Sparse Graph



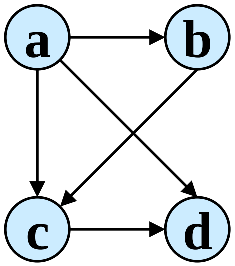
	0	1	2	3	4	5	6	7	8
0	0	1	1	0	0	0	0	0	0
1	0	0	0	1	1	0	0	0	0
2	0	0	0	0	0	1	1	1	0
3	0	0	0	0	1	0	0	0	1
4	0	0	0	0	0	1	0	0	1
5	0	0	0	0	0	0	1	0	0
6	0	0	0	0	0	0	0	0	1
7	0	0	0	0	0	0	1	0	0
8	0	0	0	0	0	0	0	0	0

Nonzero values data[ ]   1 1 1 1 1 1 1 1 1 1 1 1 1 1

destination[ ]   1 2 3 4 5 6 7 4 8 5 8 6 8 0 8

edges[ ]   0 2 4 7 9 11 12 13 15 16

Graph storage in a data file on hard disk



graph.txt

```
a  b
a  c
a  c
b  c
c  d
```

Breadth-first Search

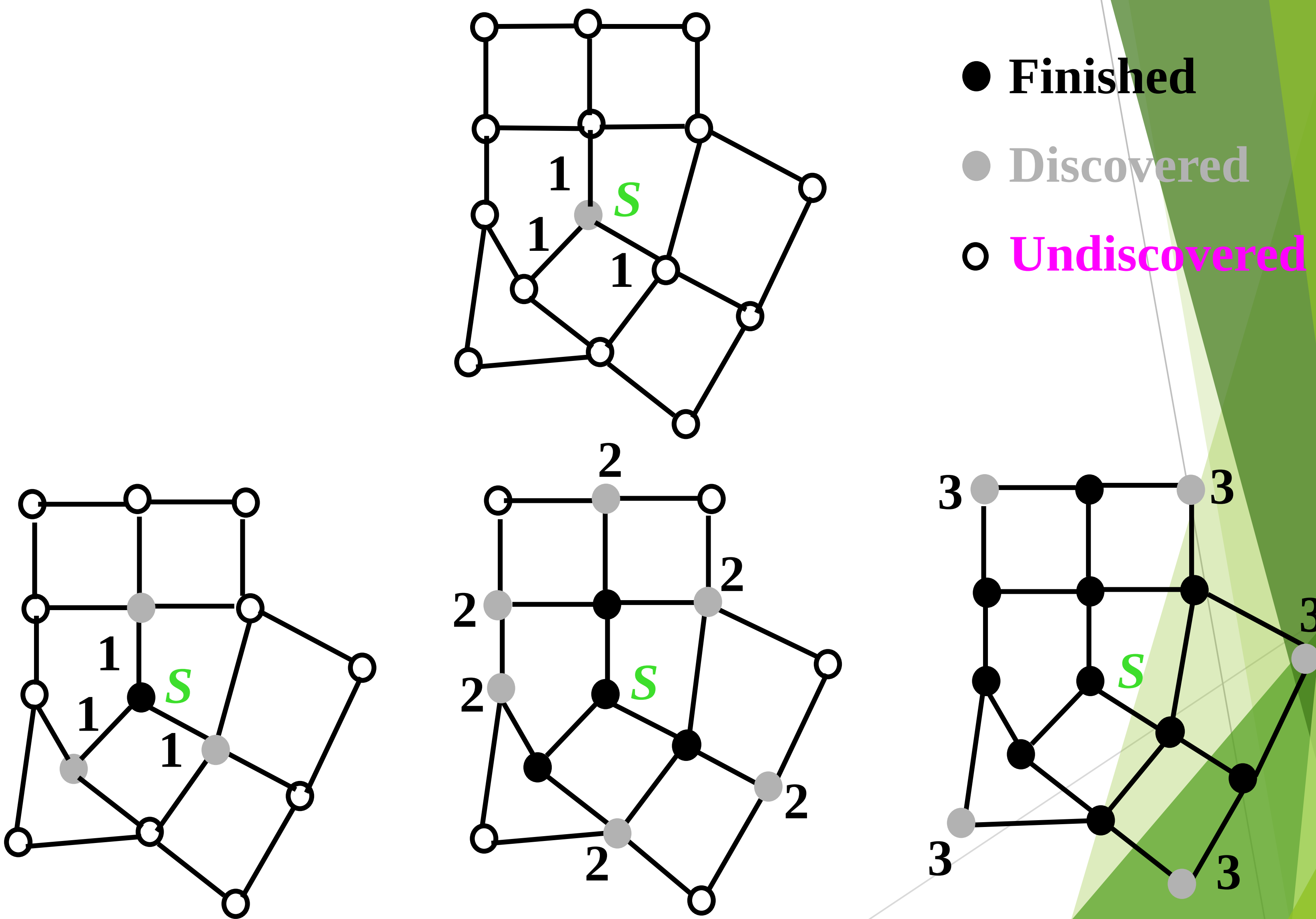
- **Input:** Graph $G = (V, E)$, either directed or undirected, and ***source vertex*** $s \in V$.
- **Output:**
  - $d[v]$ = distance (smallest # of edges, or shortest path) from s to v, for all $v \in V$. $d[v] = \infty$ if v is not reachable from s.
  - $\pi[v] = u$ such that (u, v) is last edge on shortest path $s \rightsquigarrow v$.
    - u is v's predecessor.
  - Builds breadth-first tree with root s that contains all reachable vertices.

Definitions:
Path between vertices u and v: Sequence of vertices $(v_1, v_2, \ldots, v_k)$ such that $u = v_1$ and $v = v_k$, and $(v_i, v_{i+1}) \in E$, for all $1 \leq i \leq k$-1.
Length of the path: Number of edges in the path.
Path is simple if no vertex is repeated.

Breadth-first Search

- Expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier.
  - A vertex is "discovered" the first time it is encountered during the search.
  - A vertex is "finished" if all vertices adjacent to it have been discovered.
- Colors the vertices to keep track of progress.
  - White – Undiscovered.
  - Gray – Discovered but not finished.
  - Black – Finished.

BFS for Shortest Paths



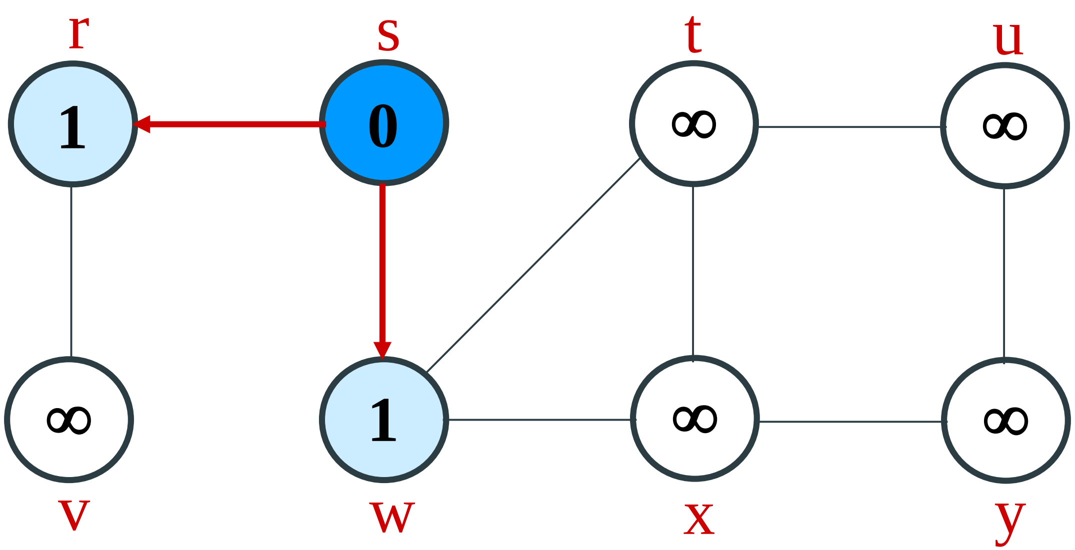
BFS(G,s)

```
for each vertex u in V[G] - {s}        ⎫
  do color[u] ← white                  ⎬ initialization
       d[u] ← ∝                        ⎪
       π[u] ← nil                      ⎭
color[s] ← gray                        ⎫
d[s] ← 0                               ⎬ access source s
 π[s] ← nil                            ⎭
Q ← Φ
enqueue(Q, s)
while Q ≠ Φ
  do u ← dequeue(Q)
  for each v in Adj[u] do
          if color[v] = white
          then color[v] ← gray
                  d[v] ← d[u] + 1
                  π[v] ← u
          enqueue(Q, v)
  color[u] ← black
```

white: undiscovered
gray: discovered
black: finished

Q: a queue of discovered vertices
color[v]: color of v
d[v]: distance from s to v
π[u]: predecessor of v

Example (BFS)

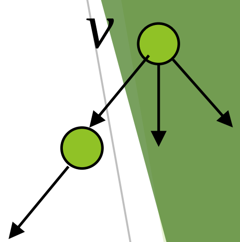


Q: w r
1 1

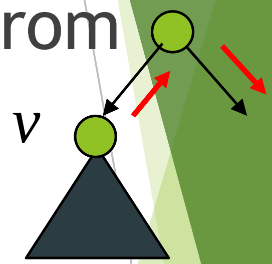
```
BFS(G,s)
for each vertex u in V[G] –  {s}
  do color[u] ← white
     d[u] ← ∝
     π[u] ← nil
color[s] ← gray
d[s] ← 0
 π[s] ← nil
Q ← Φ
enqueue(Q, s)
while Q ≠ Φ
  do u ← dequeue(Q)
  for each v in Adj[u] do
     if color[v] = white
     then color[v] ← gray
        d[v] ← d[u] + 1
        π[v] ← u
        enqueue(Q, v)
  color[u] ← black
```

Depth-first Search (DFS)

- Explore edges out of the most recently discovered vertex *v*.



- When all edges of *v* have been explored, backtrack to explore other edges leaving the vertex from which *v* was discovered (its *predecessor*).



- "Search as deep as possible first."
- Continue until all vertices reachable from the original source are discovered.
- If any undiscovered vertices remain, then one of them is chosen as a new source and search is repeated from that source.

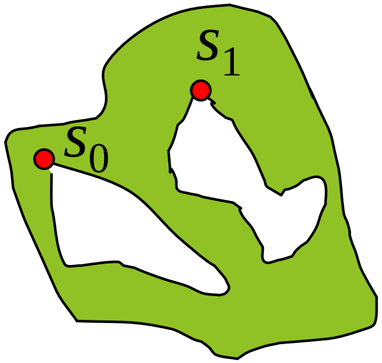
Depth-first Search

- **Input:** $G = (V, E)$, directed or undirected. No source vertex given!
- **Output:**
  - 2 **timestamps** on each vertex. Integers between 1 and 2|V|.
    - $d[v]$ = ***discovery time*** (v turns from white to gray)
    - $f[v]$ = ***finishing time*** (v turns from gray to black)
  - $\pi[v]$: predecessor of $v = u$, such that v was discovered during the scan of u's adjacency list.
- Coloring scheme for vertices as BFS. A vertex is
  - "undiscovered" (white) when it is not yet encountered.
  - "discovered" (grey) the first time it is encountered during the search.
  - "finished" (black) if it is a leaf node or all vertices adjacent to it have been finished.

Pseudo-code

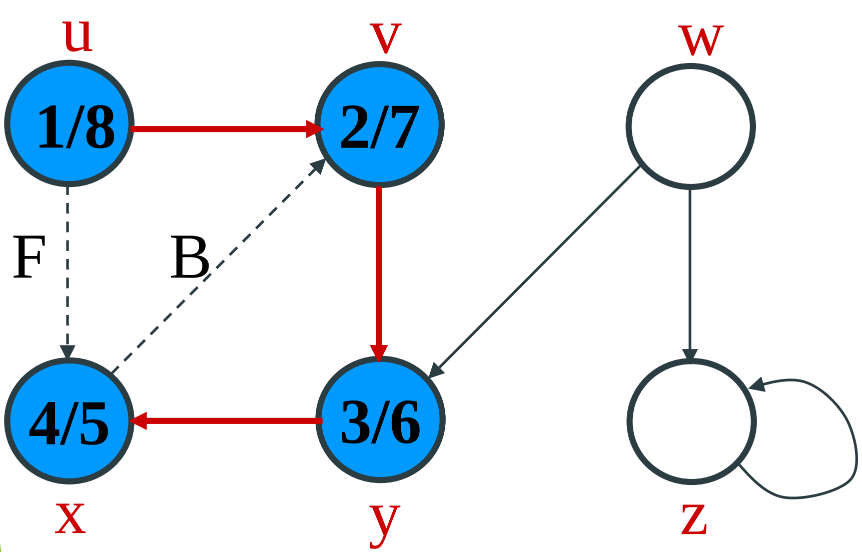
```
DFS(G)
for each vertex u ∈ V[G]
      do color[u] ← white
         π[u] ← NIL
time ← 0
for each vertex u ∈ V[G]
      do if color[u] = white
            then DFS-Visit(u)
```

Uses a global timestamp ***time***.




```
DFS-Visit(u)
    color[u] ← GRAY  // White vertex u
                        has been discovered
    time ← time + 1
    d[u] ← time
    for each v ∈ Adj[u]
        do if color[v] = WHITE
            then π[v] ← u
                DFS-Visit(v)
    color[u] ← BLACK     // Blacken u;
                            it is finished.
    f[u] ← time ← time + 1
```

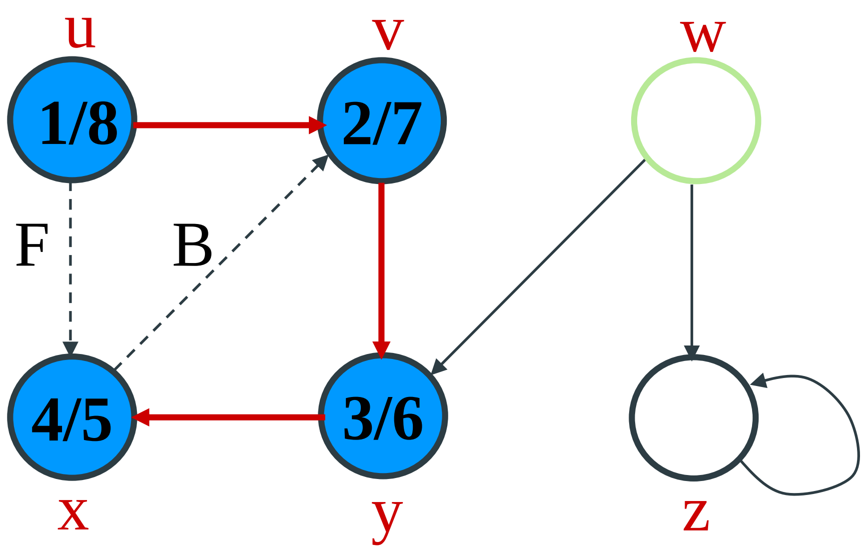
Example (DFS)



DFS-Visit(u)

```
1.  color[u] ← GRAY  // White vertex u
                        has been discovered
2.  time ← time + 1
3.  d[u] ← time
4.  for each v ∈ Adj[u]
5.      do if color[v] = WHITE
6.          then π[v] ← u
7.               DFS-Visit(v)
8.  color[u] ← BLACK     // Blacken u;
                           it is finished.
9.  f[u] ← time ← time + 1
```

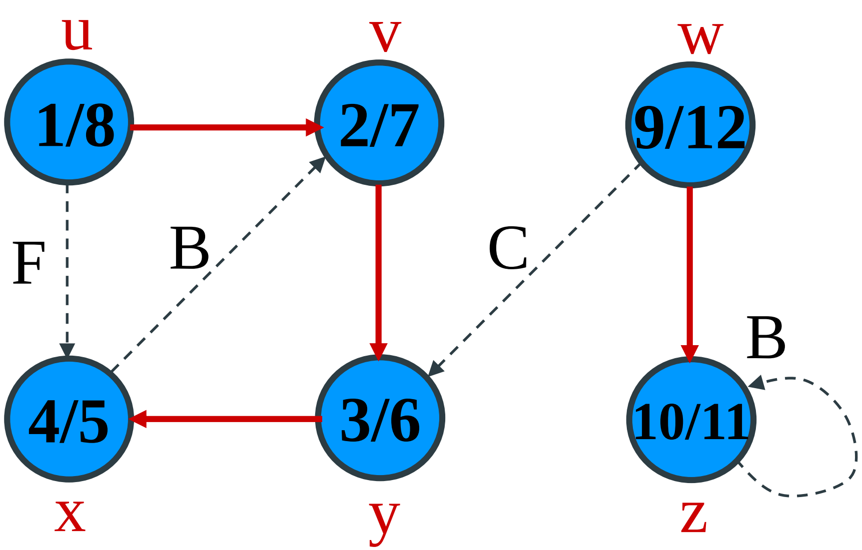
Example (DFS)



DFS(*G*)

for each vertex $u \in V[G]$
    do *color*[*u*] ← white
        $\pi[u]$ ← NIL
time ← 0
for each vertex $u \in V[G]$
    do if *color*[*u*] = white
        then DFS-Visit(*u*)

Example (DFS)

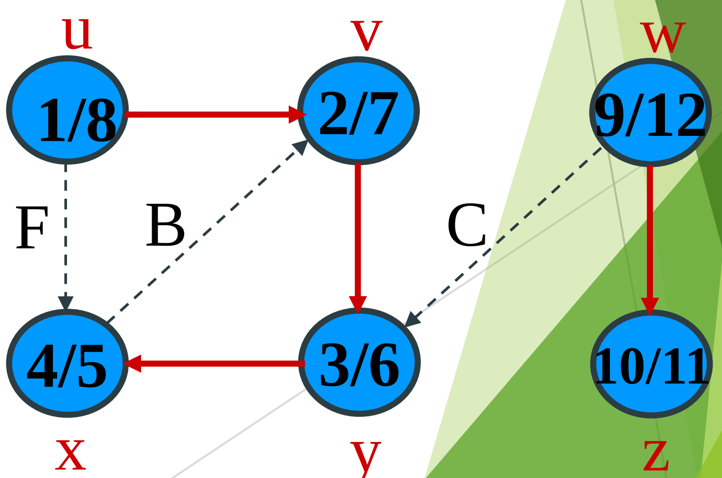


DFS-Visit(*u*)

1. *color*[*u*] ← GRAY // White vertex *u* has been discovered
2. *time* ← *time* + 1
3. *d*[*u*] ← *time*
4. **for** each *v* ∈ *Adj*[*u*]
5. **do if** *color*[*v*] = WHITE
6. **then** π[*v*] ← *u*
7. DFS-Visit(*v*)
8. *color*[*u*] ← BLACK // Blacken *u*; it is finished.
9. *f*[*u*] ← *time* ← *time* + 1

Depth-First Trees

- Predecessor subgraph defined slightly different from that of BFS.
- The predecessor subgraph of DFS is $G_\pi = (V, E_\pi)$ where $E_\pi = \{(\pi[v], v) : v \in V \text{ and } \pi[v] \neq nil\}$.
  - How does it differ from that of BFS?
  - The predecessor subgraph G_π forms a *depth-first forest* composed of several *depth-first trees*. The edges in E_π are called *tree edges*.



Definition:
Forest: An acyclic graph G that may be disconnected.

Parenthesis Theorem

Theorem 22.7

For all *u, v*, exactly one of the following holds:

1. $d[u] < f[u] < d[v] < f[v]$ or $d[v] < f[v] < d[u] < f[u]$ and neither *u* nor *v* is a descendant of the other in the *DF-tree*.
2. $d[u] < d[v] < f[v] < f[u]$ and *v* is a descendant of *u* in *DF-tree*.
3. $d[v] < d[u] < f[u] < f[v]$ and *u* is a descendant of *v* in *DF-tree*.

- So $d[u] < d[v] < f[u] < f[v]$ *cannot* happen.
- Like parentheses:
  - OK: ( ) [ ] ([ ]) [( )]
  - Not OK: ([ )] [( ])

$d[v]$ $f[v]$
([ )]
$d[u]$ $f[u]$

( )
$d[v]$ $f[v]$

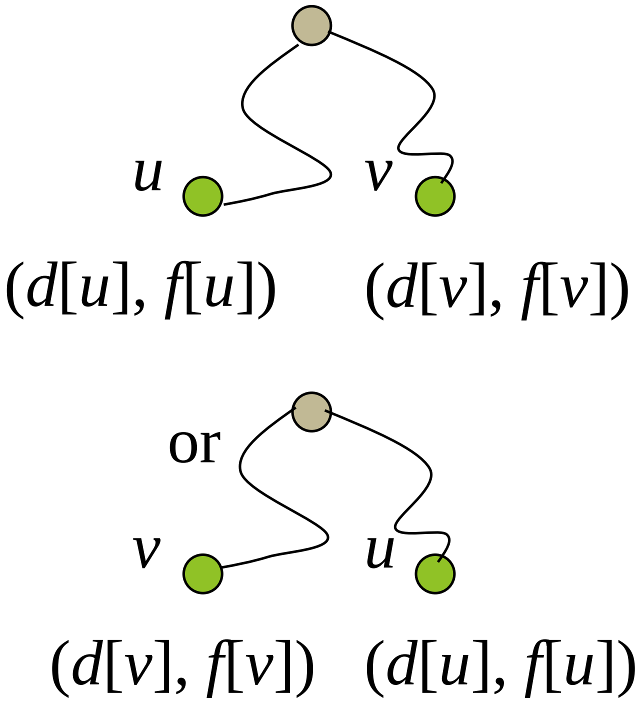
([ ])

Corollary

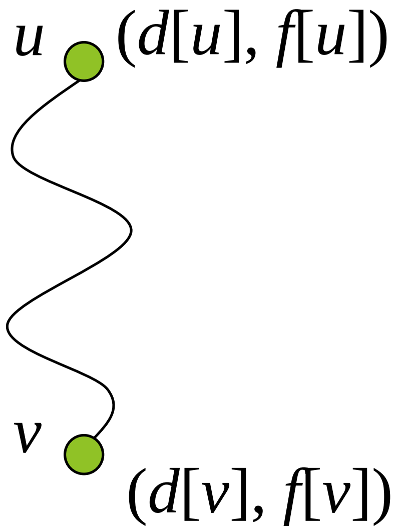
v is a proper descendant of *u* if and only if $d[u] < d[v] < f[v] < f[u]$.

Parenthesis Theorem

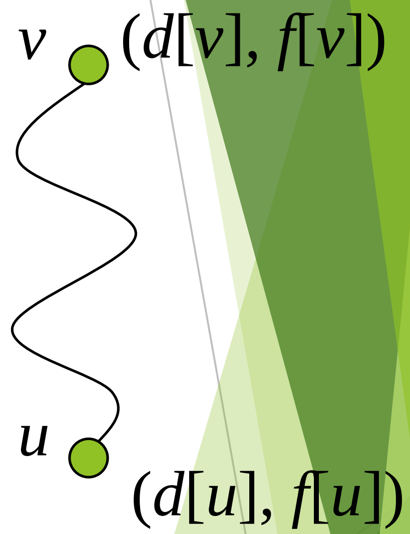
Case 1:



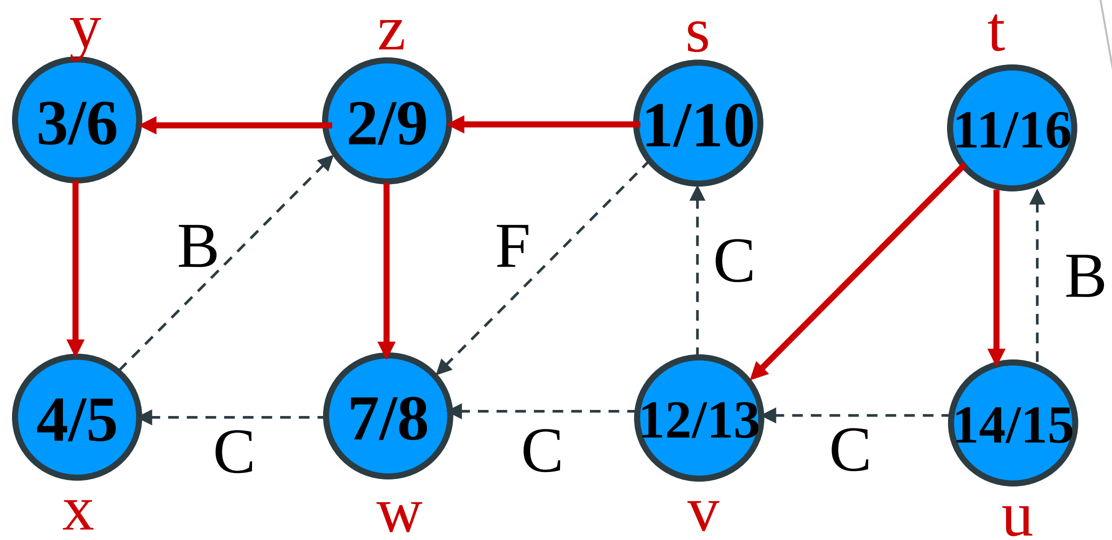
Case 2:



Case 3:



Example (Parenthesis Theorem)



(s (z (y (x x) y) (w w) z) s) (t (v v) (u u) t)

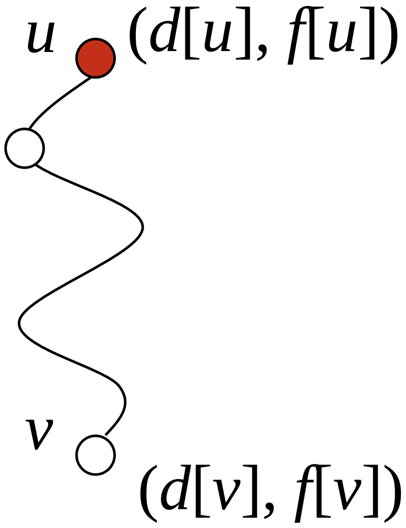
1<2<3<4<5<6<7<8<9<10 11<12<13<14<15<16

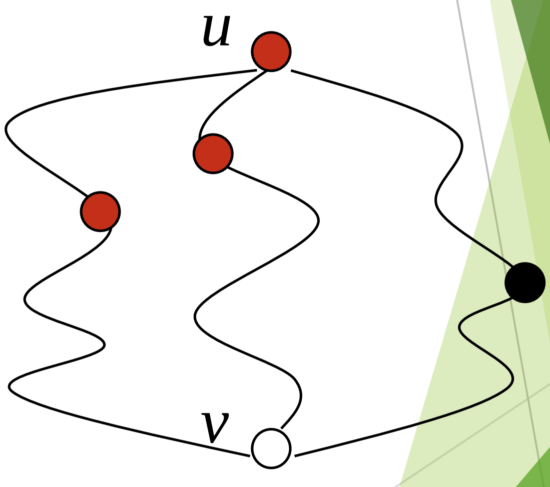
In general, if we use ‘(*v*’ to represent *d*[*v*], and ‘*v*)’ to represent *f*[*v*], the inequalities in the Parenthesis Theorem are just like parentheses in an arithmetical expression.

White-path Theorem

Theorem 22.9

v is a descendant of *u* in *DF-tree* if and only if at time *d*[*u*], there is a path *u* ⇝ *v* consisting of only white vertices. (Except for *u*, which was *just* colored gray.)

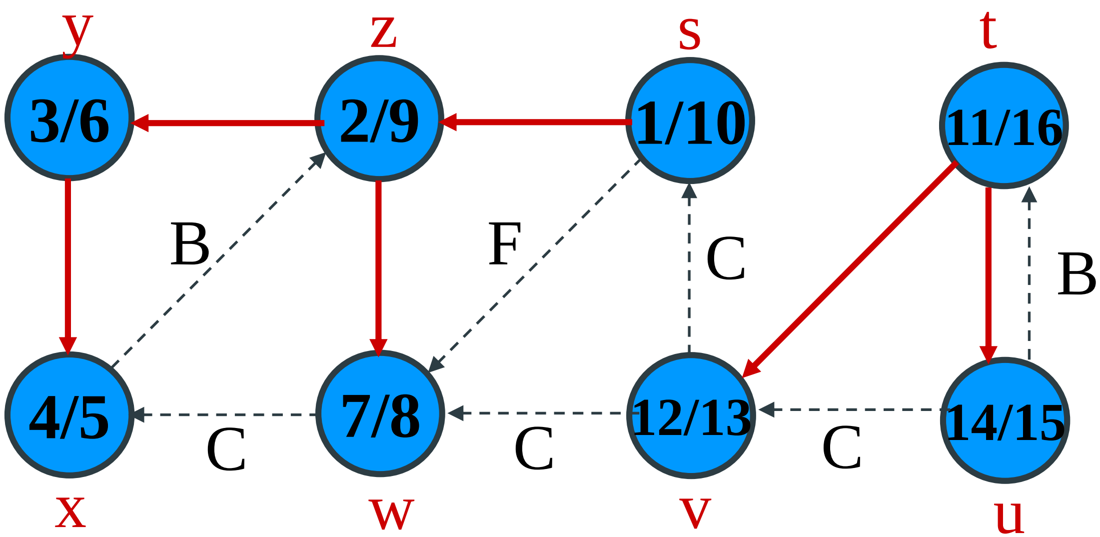


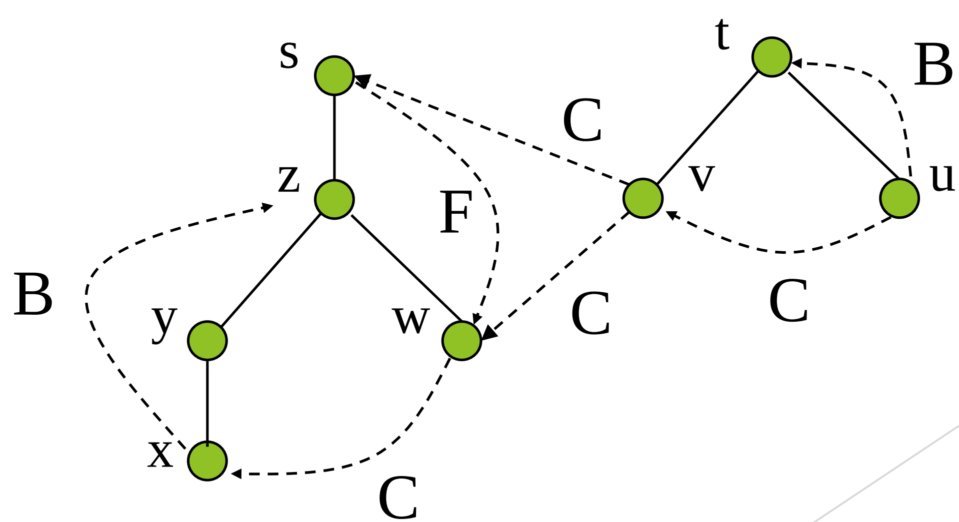


Classification of Edges

- **Tree edge:** in the depth-first forest. Found by exploring (*u*, *v*).
- **Back edge:** (*u*, *v*), where *u* is a descendant of *v* (in the depth-first tree).
- **Forward edge:** (*u*, *v*), where *v* is a descendant of *u*, but not a tree edge.
- **Cross edge:** any other edge (*u*, *v*) such that *u* is not a descendant of *v* (in the depth-first tree) and *vice versa*.

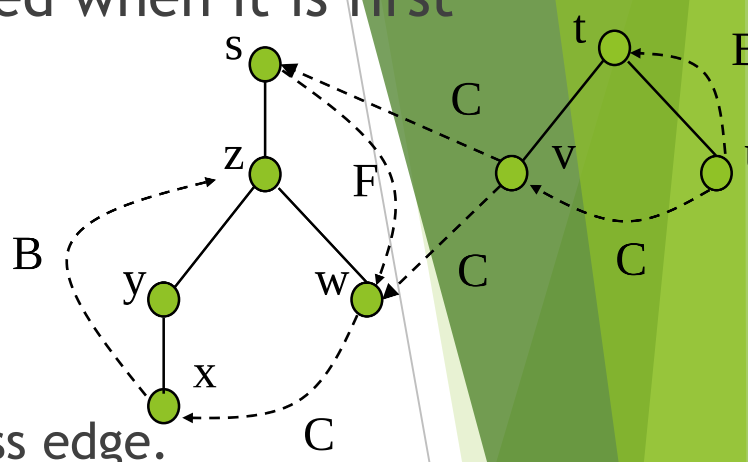
Theorem:
In DFS of an undirected graph, we get only tree and back edges. No forward or cross edges.

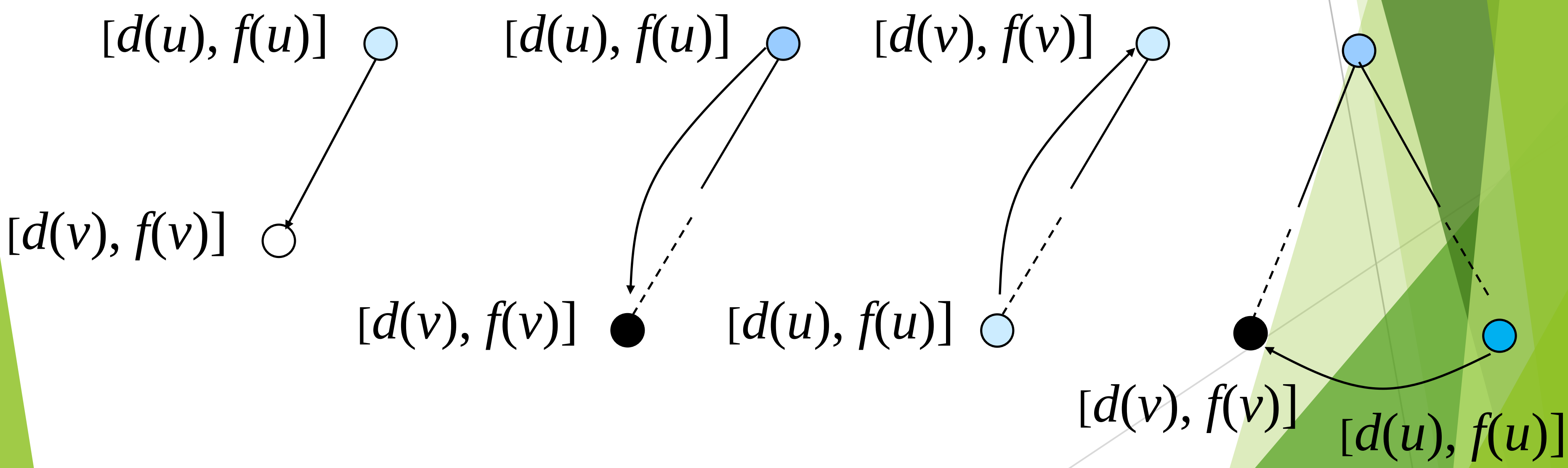

y
z
s
t
3/6
2/9
1/10
11/16
B
F
C
B
4/5
7/8
12/13
14/15
C
C
C
x
w
v
u


s
t
B
C
z
v
u
F
B
y
w
C
C
x
C

Identification of Edges

- Edge type for edge (u, v) can be identified when it is first explored by DFS.
- Identification is based on the **color of *v***.
  - If v is white, then (u, v) is a tree edge.
  - If v is gray, then (u, v) is a back edge.
  - If v is black, then (u, v) is a forward or cross edge.





Graph Algorithms – 2

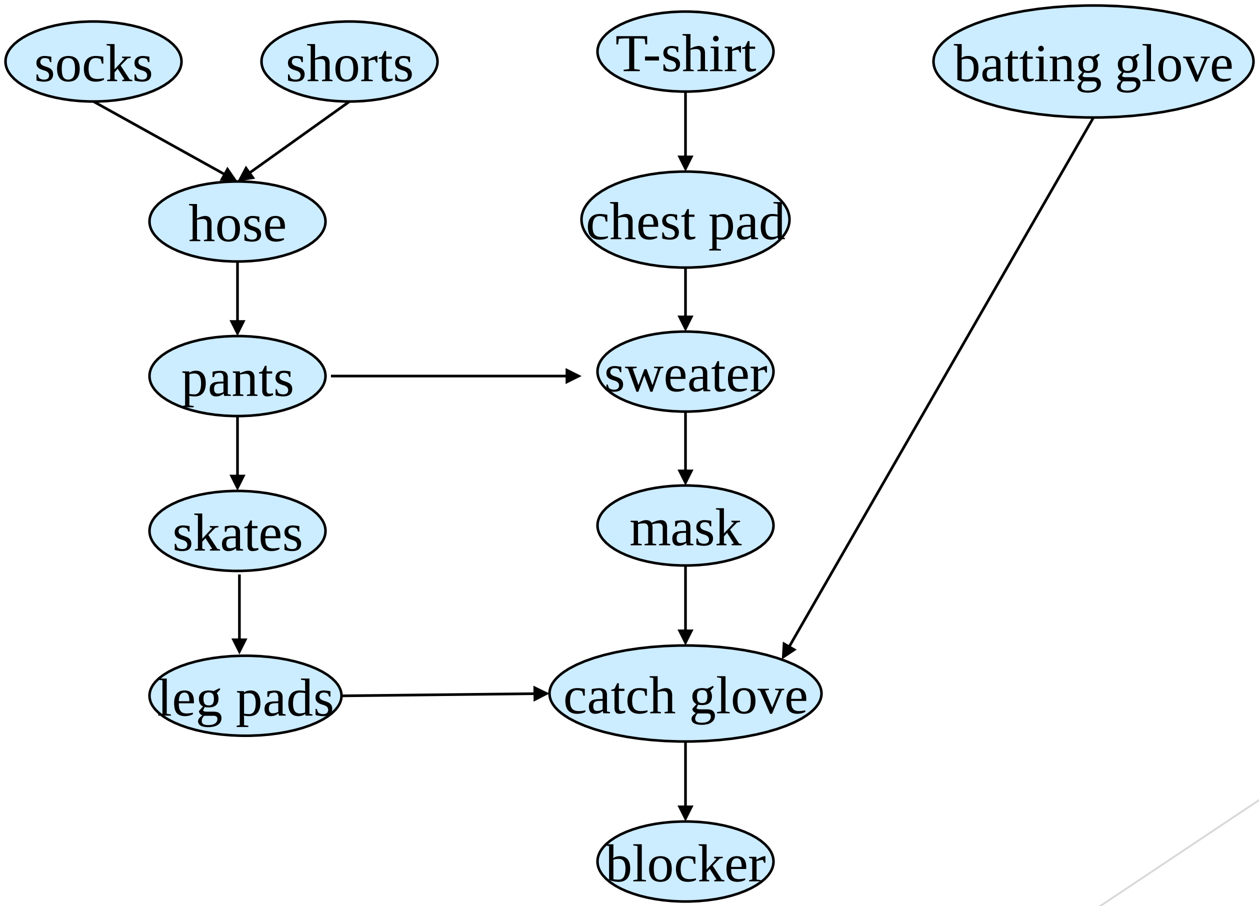
- DAGs
- Topological order
- Recognition of strongly connected components

Directed Acyclic Graph

- DAG – Directed Acyclic Graph (directed graph with no cycles)
- Used for modeling processes and structures that have a **partial order:**
  - Let a, b, c be three elements in a set U.
  - $a > b$ and $b > c \Rightarrow a > c$. (Transitivity)
  - But may have a and b such that neither $a > b$ nor $b > a$.
- We can always make a **total order** (either $a > b$ or $b > a$ for all $a \neq b$) from a partial order (by imposing a relation on any two elements whose relation is not specified with the original partial order, as long as the transitivity of this partial order not violated.)

Example

DAG of dependencies for putting on goalie equipment.

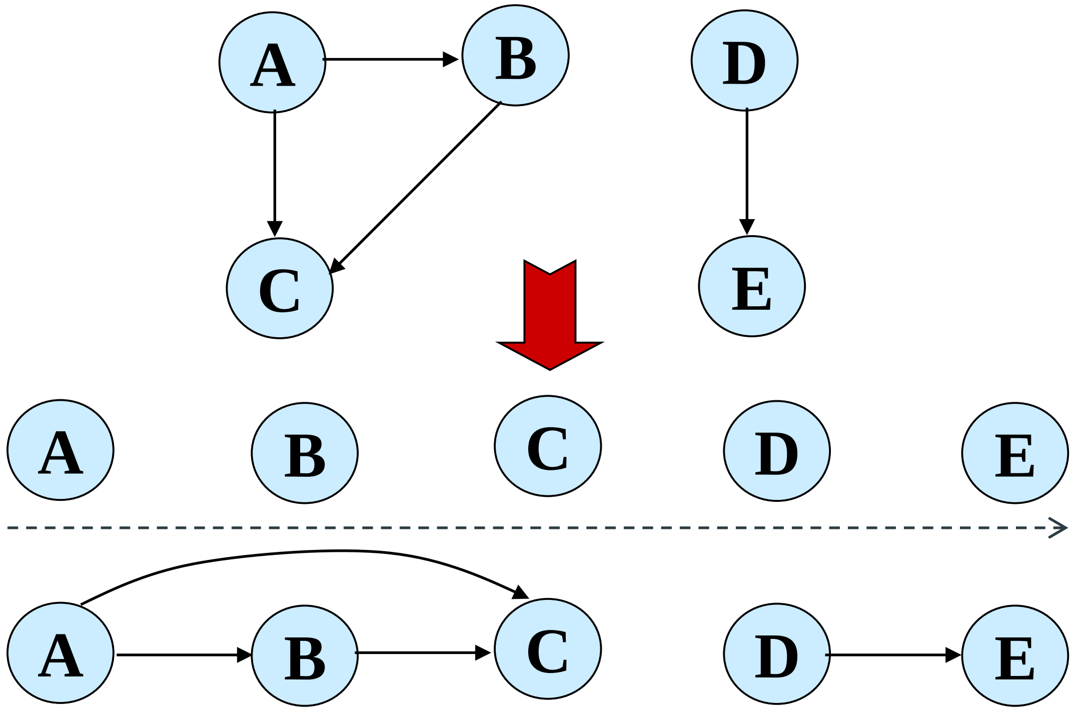


Topological Sort

- Performed on a DAG.
- Linear ordering of the vertices of $G(V, E)$ such that if $(u, v) \in E$, then u appears somewhere before v.

Topological Sort

Sort a directed acyclic graph (DAG) by the nodes' finishing times.



Think of original DAG as a **partial order**.

By sorting, we get a **total order** that extends this partial order.

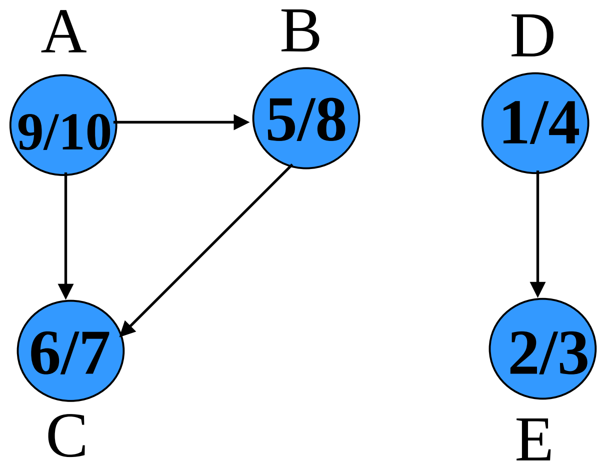
Topological Sort

- Performed on a DAG.
- Linear ordering of the vertices of G such that if $(u, v) \in E$, then u appears somewhere before v.

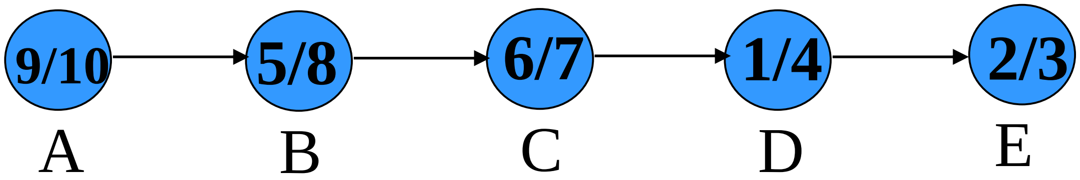
Topological-Sort (G)
1. call DFS(G) to compute finishing times $f[v]$ for all $v \in V$
2. as each vertex is finished, insert it onto the front of a linked list
3. **return** the linked list of vertices

Time: $\Theta(|V| + |E|)$.

Example 1

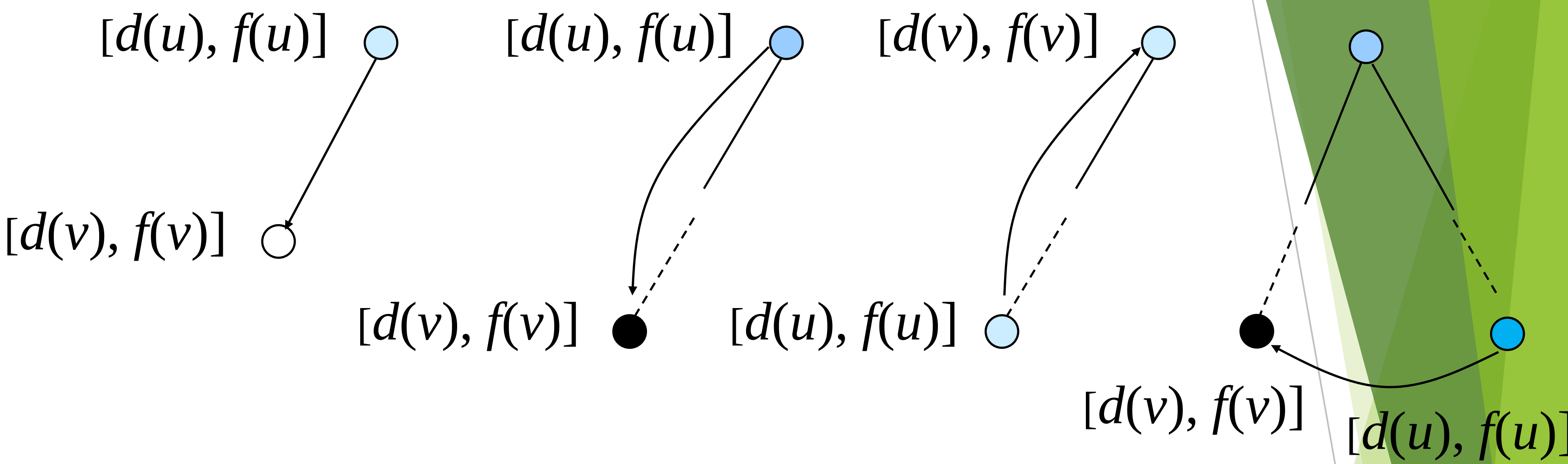


Linked List:

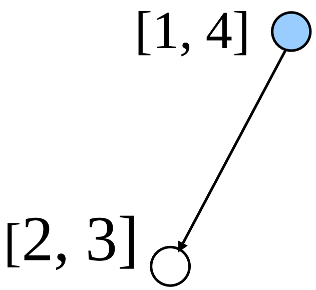


Correctness Proof

- Just need to show if $(u, v) \in E$, then $f[u] > f[v]$.
- When we explore (u, v), what are the colors of u and v?
  - u is gray.
  - Is v white?
    - Then becomes descendant of u.
    - By parenthesis theorem, $d[u] < d[v] < \underline{f[v] < f[u]}$.
  - Is v black?
    - Then v is already finished.
    - Since we're exploring (u, v), we have not yet finished u.
    - Therefore, $f[v] < f[u]$.
  - Is v gray, too?
    - No.
    - because then v would be ancestor of $u \Rightarrow (u, v)$ is a back edge.
    - $\Rightarrow$ contradiction of Lemma 22.11 (dag has no back edges).

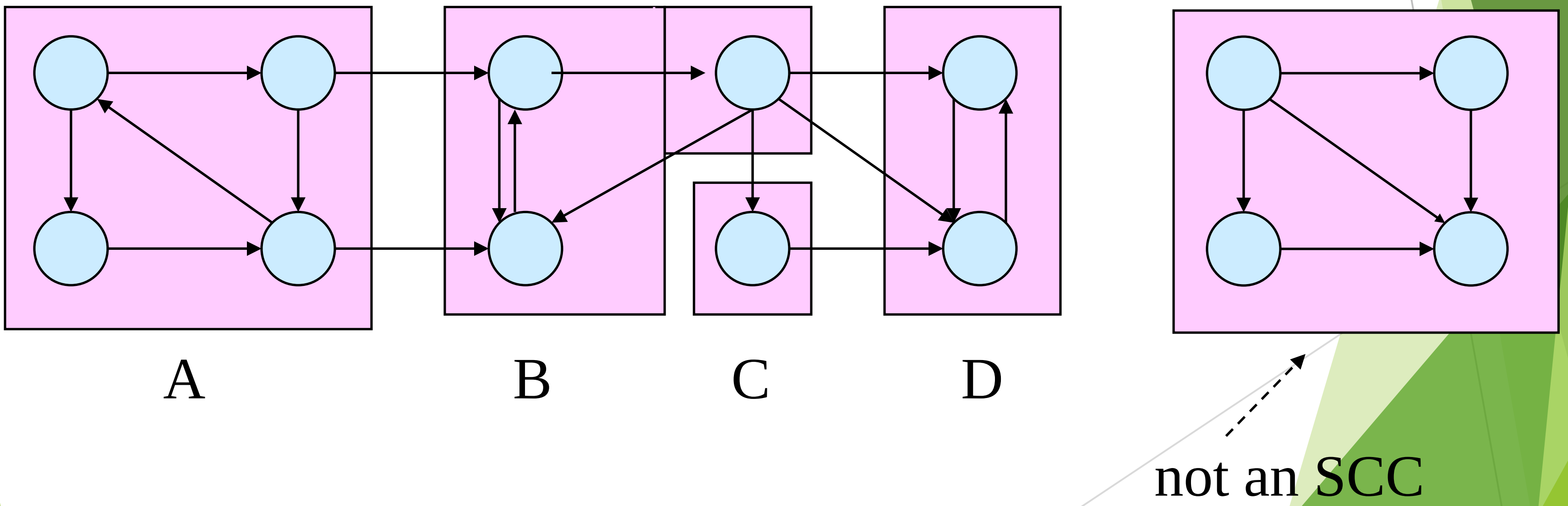


Example:



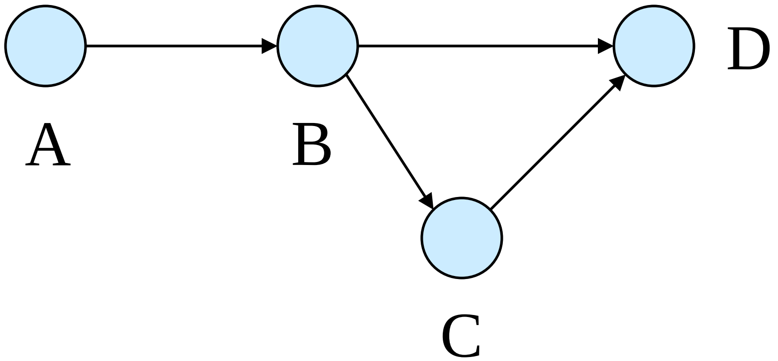
Strongly Connected Components

- G is strongly connected if every pair (u, v) of vertices in G is reachable from one another.
- A **strongly connected component** (***SCC***) of G is a maximal set of vertices $C \subseteq V$ such that for all $u, v \in C$, both $u \leadsto v$ and $v \leadsto u$ exist.



Component Graph

- $G^{SCC} = (V^{SCC}, E^{SCC})$.
- V^{SCC} has one vertex for each SCC in G.
- E^{SCC} has an edge if there's an edge between the corresponding SCC's in G.
- G^{SCC} for the example considered:



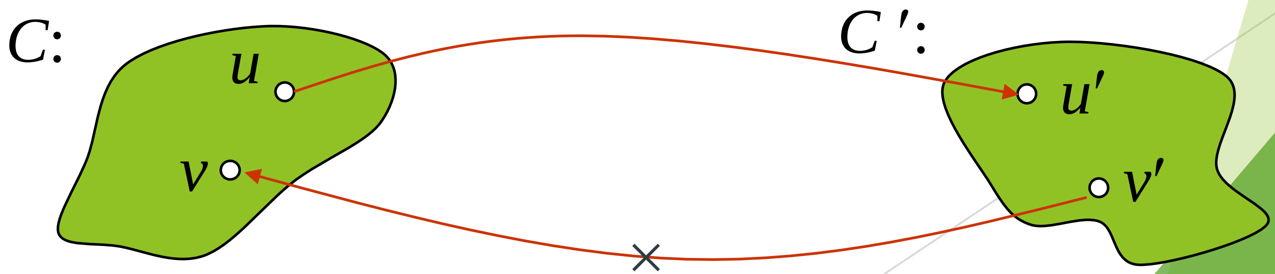
G^{SCC} is a DAG

Lemma 22.13

Let C and C' be distinct SCC's in G, let $u, v \in C$, $u', v' \in C'$, and suppose there is a path $u \rightsquigarrow u'$ in G. Then there cannot also be a path $v' \rightsquigarrow v$ in G.

Proof:

- Suppose there is a path $v' \rightsquigarrow v$ in G.
- Then there are paths $u \rightsquigarrow u' \rightsquigarrow v'$ and $v' \rightsquigarrow v \rightsquigarrow u$ in G.
- Therefore, u and v' are reachable from each other, so they are not in separate SCC's.



Transpose of a Directed Graph

- G^T = **transpose** of directed G.
  - $G^T = (V, E^T)$, $E^T = \{(u, v) : (v, u) \in E\}$.
  - G^T is G with all edges reversed.
- Can create G^T in $\Theta(|V| + |E|)$ time if using adjacency lists.
- G and G^T have the *same* SCC's. (u and v are reachable from each other in G if and only if reachable from each other in G^T.)

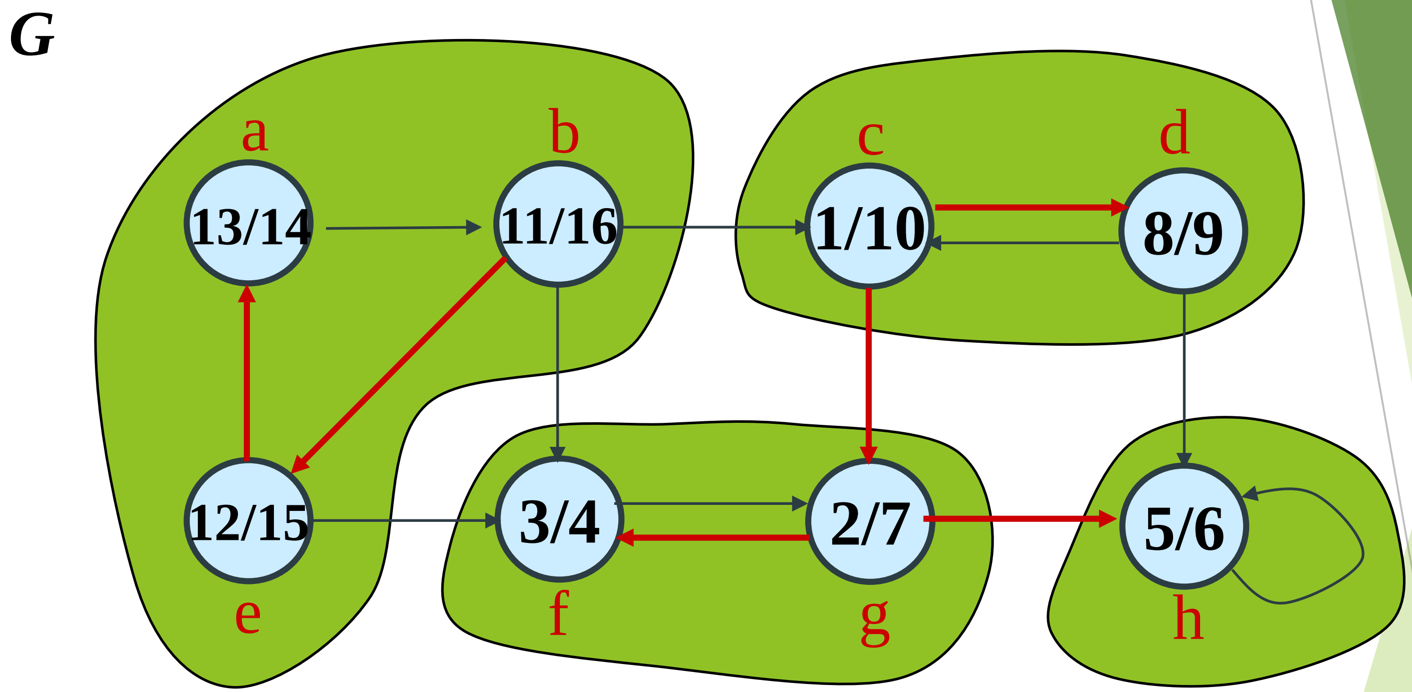
Algorithm to determine SCCs

<u>SCC(G)</u>

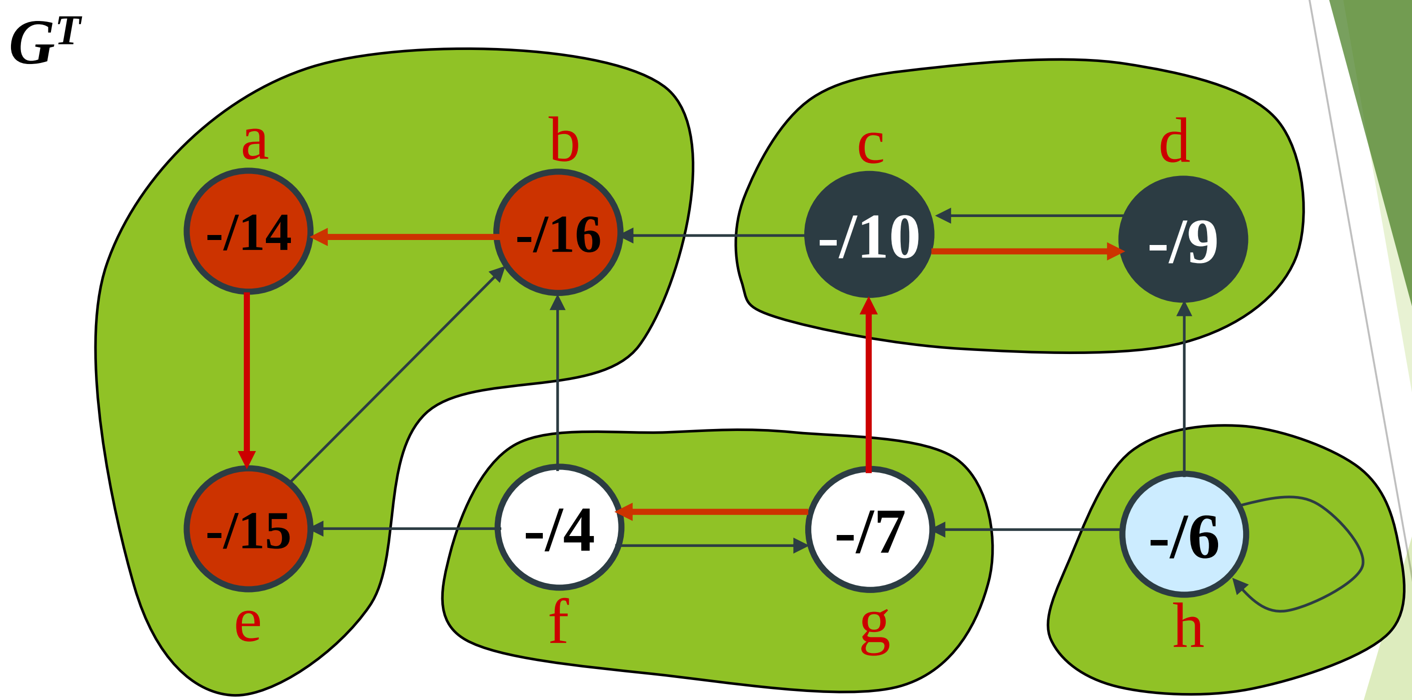
1. call DFS(G) to compute finishing times $f[u]$ for all u
2. compute G^T
3. call DFS(G^T), but in the main loop, consider vertices in order of decreasing $f[u]$ (as computed in the first DFS)
4. output the vertices in each tree of the depth-first forest formed in the second DFS as a separate SCC

Time: $\Theta(|V| + |E|)$.

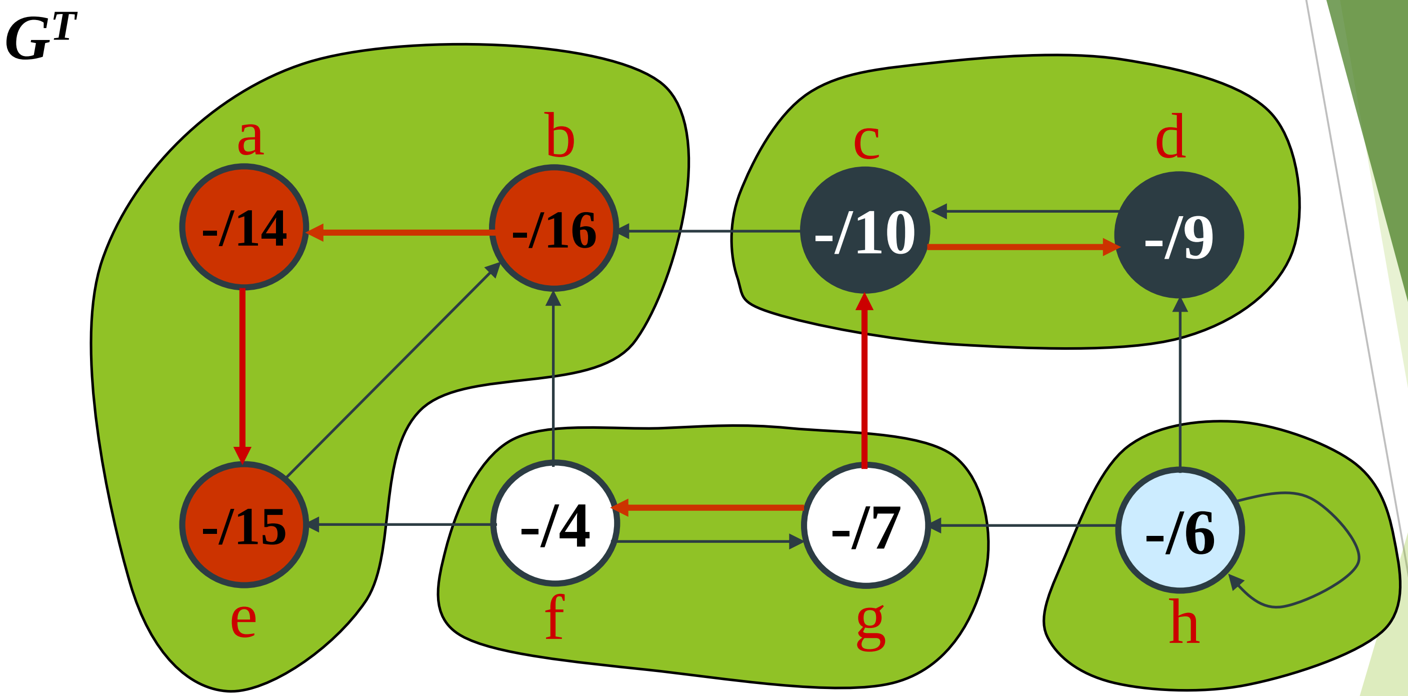
Example



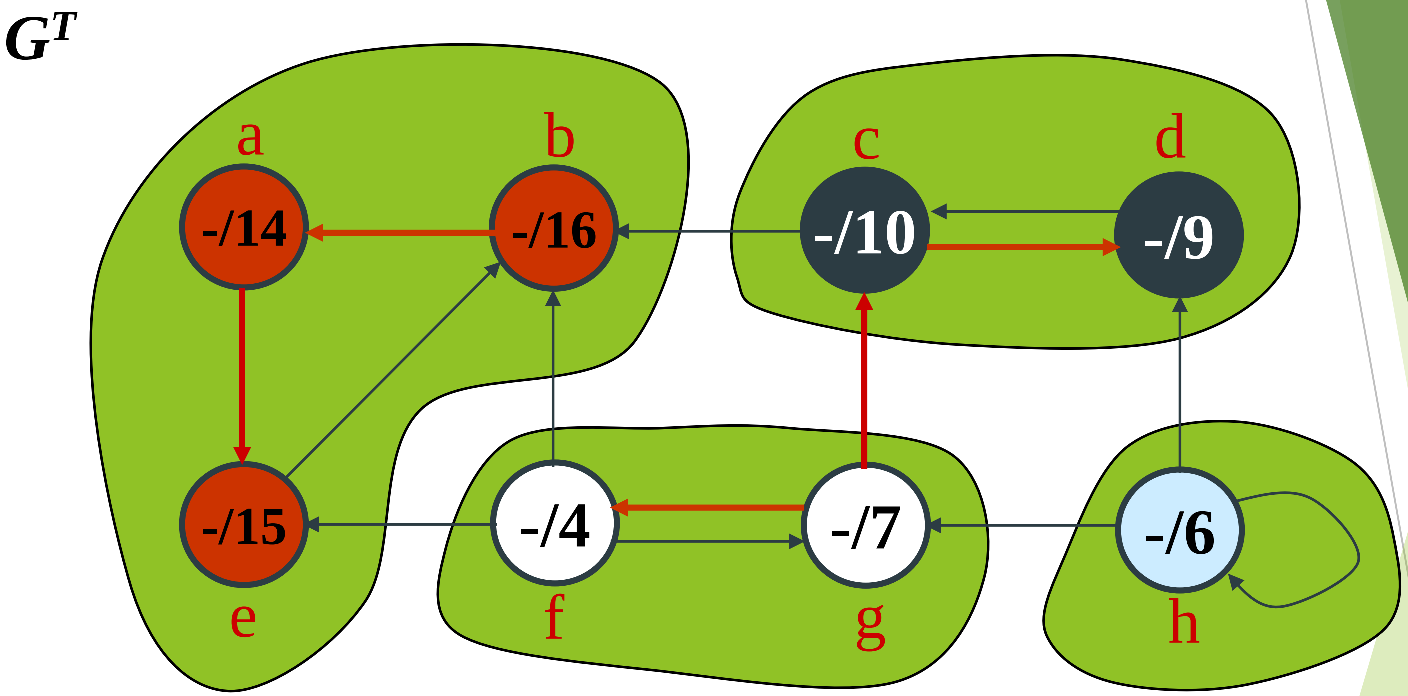
Example



Example



Example

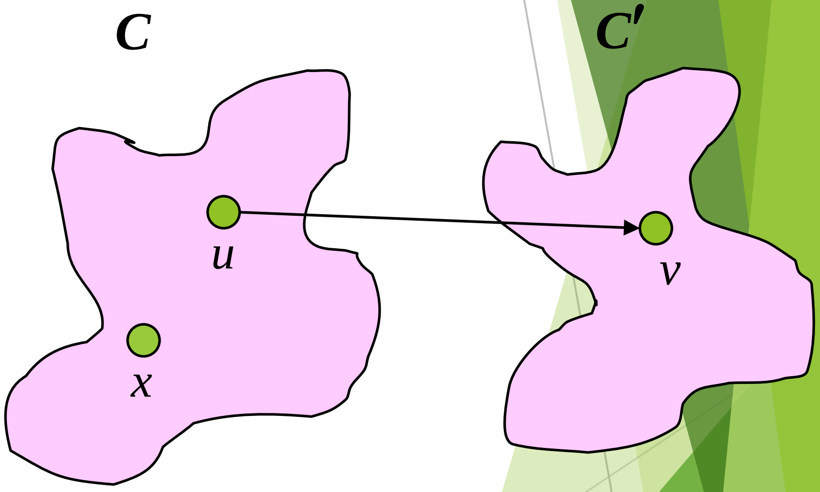


SCCs and DFS finishing times

Lemma 22.14
Let C and C' be distinct SCC's in $G = (V, E)$. Suppose there is an edge $(u, v) \in E$ such that $u \in C$ and $v \in C'$. Then $f(C) > f(C')$.

Proof:

- Case 1: $d(C) < d(C')$
  - Let x be the first vertex discovered in C.
  - At time $d[x]$, all vertices in C and C' are white. Thus, there exist paths of white vertices from x to all vertices in C and C'.
  - By the white-path theorem, all vertices in C and C' are descendants of x in depth-first tree.
  - By the parenthesis theorem, $f[x] = f(C) > f(C')$.



$d(x) < d(v) < f(v) < f(x)$

$d(C) = \min_{u \in C}\{d[u]\})$

$f(C) = \max_{u \in C}\{f[u]\}$

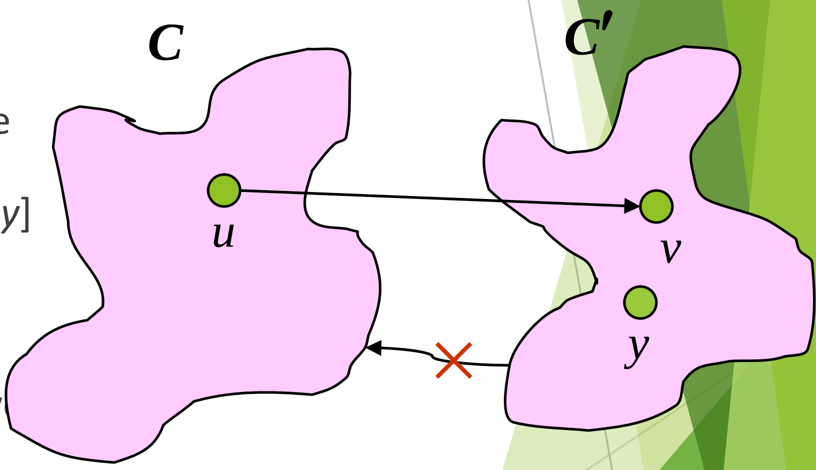
SCCs and DFS finishing times

Lemma 22.14

Let C and C' be distinct SCC's in $G = (V, E)$. Suppose there is an edge $(u, v) \in E$ such that $u \in C$ and $v \in C'$. Then $f(C) > f(C')$.

Proof:

- Case 2: $d(C) > d(C')$
  - Let y be the first vertex discovered in C'.
  - At time $d[y]$, all vertices in C' are white and there is a white path from y to each vertex in $C' \Rightarrow$ all vertices in C' become descendants of y. Again, $f[y] = f(C')$.
  - At time $d[y]$, all vertices in C are also white.
  - By earlier lemma, since there is an edge (u, v), we cannot have a path from C' to C.
  - So no vertex in C is reachable from y.
  - Therefore, at time $f[y]$, all vertices in C are still white.
  - Therefore, for all $v \in C$, $f[v] > f[y]$, which implies that $f(C) > f(C')$.



$$d(C) = \min_{u \in C}\{d[u]\})$$
$$f(C) = \max_{u \in C}\{ f[u]\}$$

SCCs and DFS finishing times

Corollary 22.15
Let C and C' be distinct SCC's in $G = (V, E)$. Suppose there is an edge $(u, v) \in E^T$, where $u \in C$ and $v \in C'$. Then $f(C) < f(C')$.

Proof:

- $(u, v) \in E^T \Rightarrow (v, u) \in E$.
- Since SCC's of G and G^T are the same, $f(C') > f(C)$, by Lemma 22.14.