Transition system

Definition 1.3.1 A ***transition system*** S is a pair of the form

$$S = (C, T)$$

where C is the set of ***configurations*** and $T \subseteq C \times C$ is a relation, the ***transition relation***. □

Sequences generated by a transition system

Definition 1.3.3 Let $S = (C, T)$ be a transition system. S ***generates a set of sequences***, $\mathcal{S}(S)$, defined as follows:

1. the finite sequence $c_0, c_1, \ldots, c_n$ (for $n \geq 0$) belongs to $\mathcal{S}(S)$ if
   (ι) $c_0 \in C$
   ($\iota\iota$) for all i with $1 \leq i \leq n$: $(c_{i-1}, c_i) \in T$
2. the infinite sequence $c_0, c_1, \ldots, c_n, \ldots$ belongs to $\mathcal{S}(S)$ if
   (ι) $c_0 \in C$
   ($\iota\iota$) for all $i \geq 1$: $(c_{i-1}, c_i) \in T$

□

Processes generated by a transition system

Definition 1.3.5 Let $S = (C, T)$ be a transition system. The set of ***processes generated by*** S, written $\mathcal{P}(S)$, is the subset of $\mathcal{S}(S)$ containing

1. all infinite sequences of $\mathcal{S}(S)$
2. all finite sequences $c_0, c_1, \ldots, c_n$ $(n \geq 0)$ of $\mathcal{S}(S)$ for which it holds that there is no $c \in C$ with $(c_n, c) \in T$.

The final configuration of a finite process is called a ***dead configuration***. □

Football

Transition system Football

Configurations: $\{[t, X, a, b] \mid 0 \leq t \leq 90, X \in \{A, B, R\}, a, b \in \mathbf{N}\}$

$$
\begin{array}{lcll}
[t, A, a, b] & \rhd & [t+2, B, a, b] & \textbf{if } t \leq 88 \\
[t, A, a, b] & \rhd & [t+2, B, a+1, b] & \textbf{if } t \leq 88 \\
[t, A, a, b] & \rhd & [t+1, B, a, b] & \textbf{if } t \leq 89 \\
[t, A, a, b] & \rhd & [t+1, B, a+1, b] & \textbf{if } t \leq 89 \\
[90, A, a, b] & \rhd & [90, R, a, b] & \\
[t, B, a, b] & \rhd & [t+2, A, a, b] & \textbf{if } t \leq 88 \\
[t, B, a, b] & \rhd & [t+2, A, a, b+1] & \textbf{if } t \leq 88 \\
[t, B, a, b] & \rhd & [t+1, A, a, b] & \textbf{if } t \leq 89 \\
[t, B, a, b] & \rhd & [t+1, A, a, b+1] & \textbf{if } t \leq 89 \\
[90, B, a, b] & \rhd & [90, R, a, b] &
\end{array}
$$

Induction principle

Induction principle Let $P(0), P(1), \ldots, P(n), \ldots$ be statements. If

a) $P(0)$ is true

b) for all $n \geq 0$ it holds that $P(n)$ implies $P(n+1)$,

then $P(n)$ is true for all $n \geq 0$.

Invariance principle

Invariance principle for transition systems Let $S = (C, T)$ be a transition system and let $c_0 \in C$ be a configuration. If $I(c)$ is a statement about the configurations of the system, the following holds. If

a) $I(c_0)$ is true

b) for all $(c, c') \in T$ it holds that $I(c)$ implies $I(c')$

then $I(c)$ is true for any configuration c that occurs in a sequence starting with c_0.

Termination principle

Termination principle for transition systems Let $S = (C, T)$ be a transition system and let $\mu : C \to \mathbf{N}$ be a function. If

$$\text{for all } (c, c') \in T \text{ it holds that } \mu(c) > \mu(c')$$

then all processes in $\mathcal{P}(S)$ are finite.

Nim

Transition system Nim

Configurations: $\{A, B\} \times \mathbf{N}$

$$
\begin{array}{llll}
[A, n] & \triangleright & [B, n-2] & \textbf{if } n \geq 2 \\
[A, n] & \triangleright & [B, n-1] & \textbf{if } n \geq 1 \\
[B, n] & \triangleright & [A, n-2] & \textbf{if } n \geq 2 \\
[B, n] & \triangleright & [A, n-1] & \textbf{if } n \geq 1
\end{array}
$$

Towers of Hanoi

Transition system $\mathsf{Hanoi}(n)$

Configurations: $\{[A,B,C] \mid \{A,B,C\} \text{ a partition of } \{1,\ldots,n\}\}$

$$
\begin{array}{lll}
[A,B,C] & \rhd & [A\setminus\{r\}, B\cup\{r\}, C] \;\; \textbf{if } (r = \min A) \wedge (r < \min B) \\
[A,B,C] & \rhd & [A\setminus\{r\}, B, C\cup\{r\}] \;\; \textbf{if } (r = \min A) \wedge (r < \min C) \\
[A,B,C] & \rhd & [A\cup\{r\}, B\setminus\{r\}, C] \;\; \textbf{if } (r = \min B) \wedge (r < \min A) \\
[A,B,C] & \rhd & [A, B\setminus\{r\}, C\cup\{r\}] \;\; \textbf{if } (r = \min B) \wedge (r < \min C) \\
[A,B,C] & \rhd & [A\cup\{r\}, B, C\setminus\{r\}] \;\; \textbf{if } (r = \min C) \wedge (r < \min A) \\
[A,B,C] & \rhd & [A, B\cup\{r\}, C\setminus\{r\}] \;\; \textbf{if } (r = \min C) \wedge (r < \min B)
\end{array}
$$

Euclid's algorithm

Transition system Euclid

Configurations: $\{[m, n] \mid m, n \geq 1\}$

$$[m, n] \;\triangleright\; [m - n, n] \quad \textbf{if } m > n$$

$$[m, n] \;\triangleright\; [m, n - m] \quad \textbf{if } m < n$$

Expressions

Transition system Expressions

Configurations: $\{0, 1, +, \mathbf{E}, \mathbf{T}, (,)\}^*$

$$\alpha\mathbf{E}\beta \;\triangleright\; \alpha\mathbf{T}\beta$$
$$\alpha\mathbf{E}\beta \;\triangleright\; \alpha\mathbf{T}{+}\mathbf{E}\beta$$
$$\alpha\mathbf{T}\beta \;\triangleright\; \alpha 0\beta$$
$$\alpha\mathbf{T}\beta \;\triangleright\; \alpha 1\beta$$
$$\alpha\mathbf{T}\beta \;\triangleright\; \alpha(\mathbf{E})\beta$$

Expressions (context-free)

Transition system Expressions

Configurations: $\{0, 1, +, \mathbf{E}, \mathbf{T}, (,)\}^*$

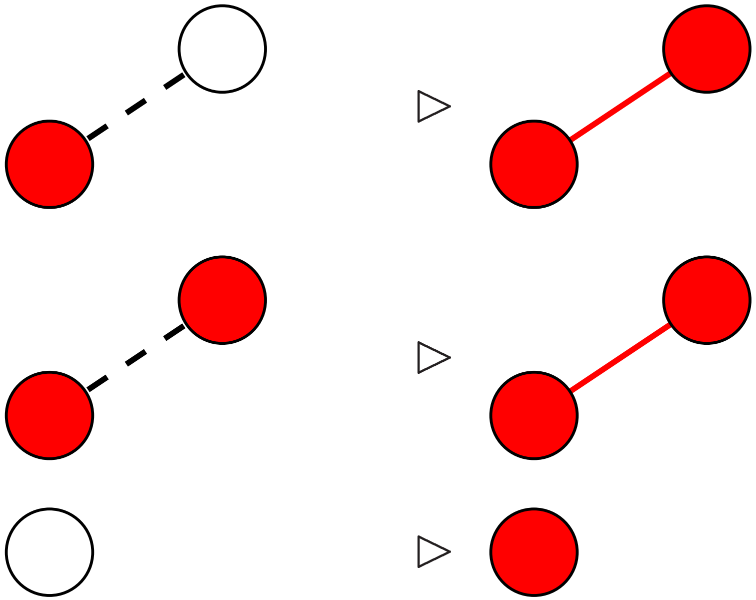
$\mathbf{E} \;\triangleright\; \mathbf{T}, \mathbf{T}{+}\mathbf{E}$

$\mathbf{T} \;\triangleright\; 0, 1, (\mathbf{E})$

Graph coloring

Transition system GraphColoring

Configurations: Danish graphs



if there are no pink nodes

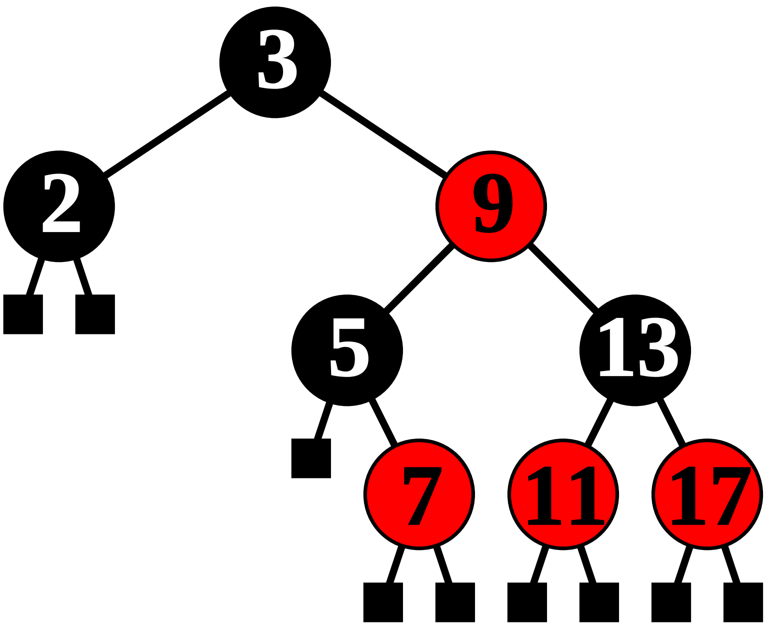
Red-black tree

Definition 1.5.7 A ***red-black tree*** is binary search tree in which all internal nodes are colored either red or black, in which the leaves are black, and

Invariant I_2 Each red node has a black parent.

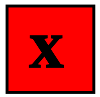
Invariant I_3 There is the same number of black nodes on all paths from the root to a leaf.

□



Insertion

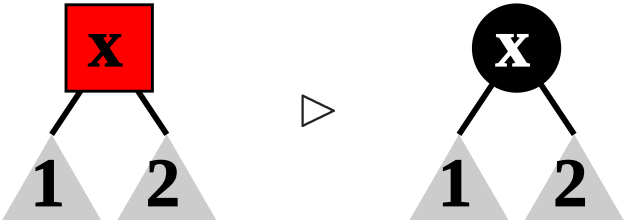
Illegitimate red node:



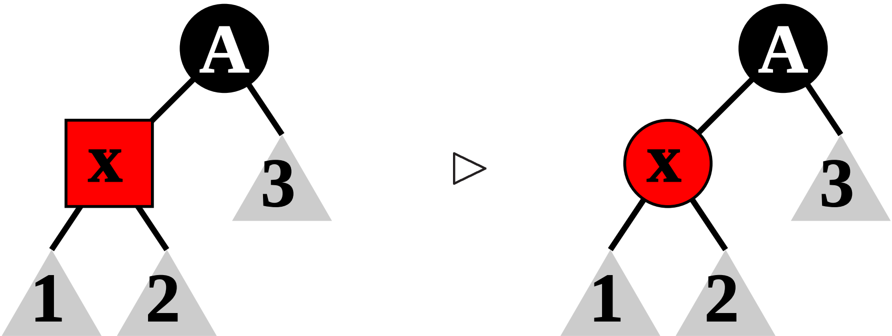
Invariant I_2'**:** Each *legitimate* red node has a black parent.

Insertion: transitions 1 and 2

The illegitimate node is the root of the tree:

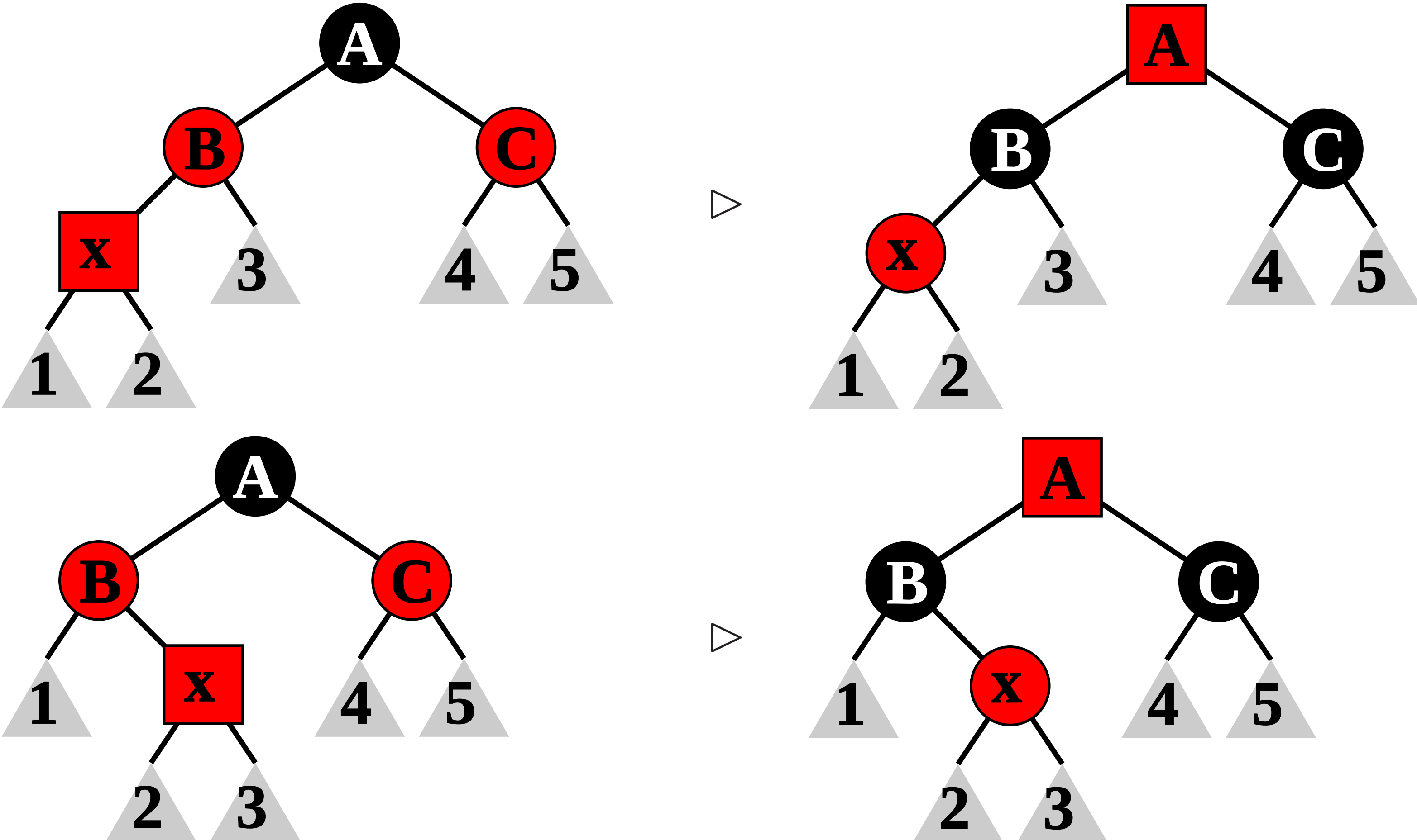


The illegitimate node has a black father:



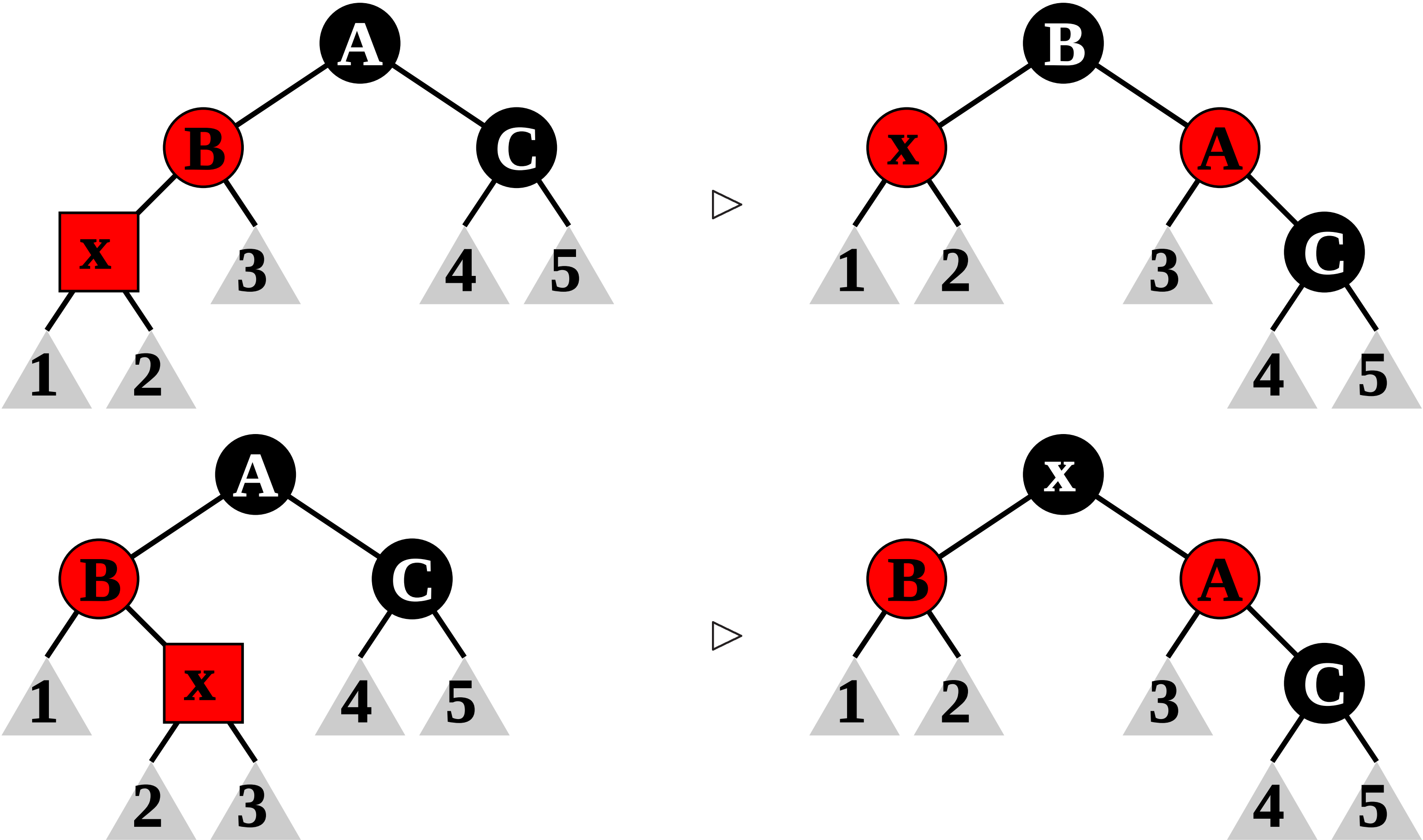
Insertion: transitions 3.1 and 3.2

The illegitimate node has a red father and a red uncle:



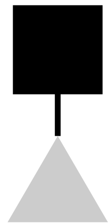
Insertion: transitions 4.1 and 4.2

The illegitimate node has a red father and a black uncle:



Deletion

Illegitimate black node:

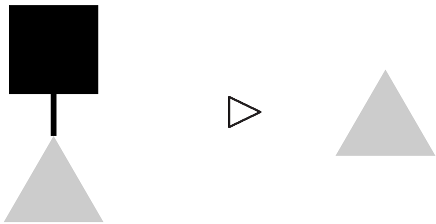


Invariant I_1' The tree satisfies I_1 if we remove the illegitimate node.

Invariant I_2' Each red node has a *legitimate* black father.

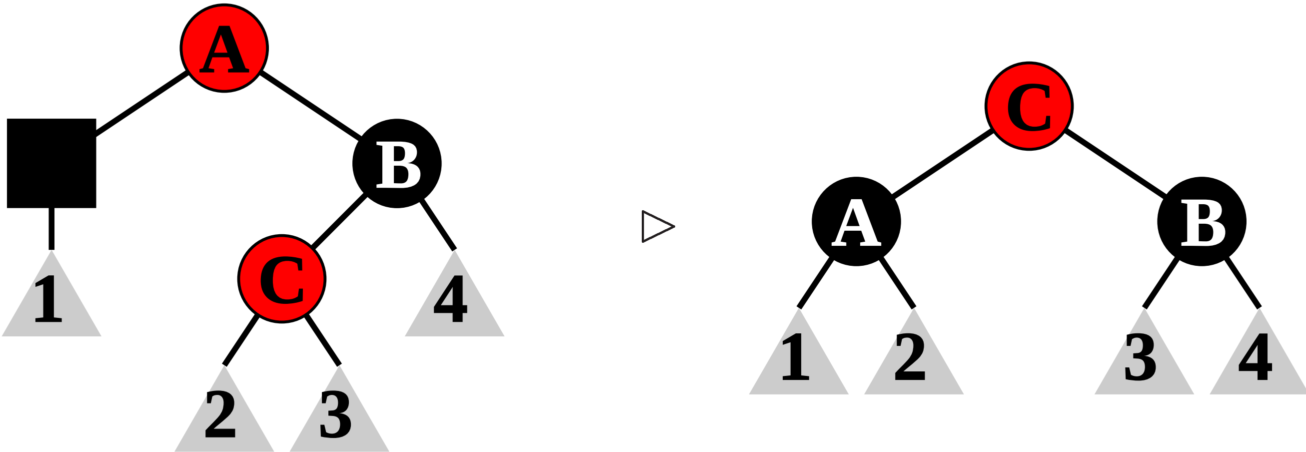
Deletion: transition 1

The illegitimate node is the root:

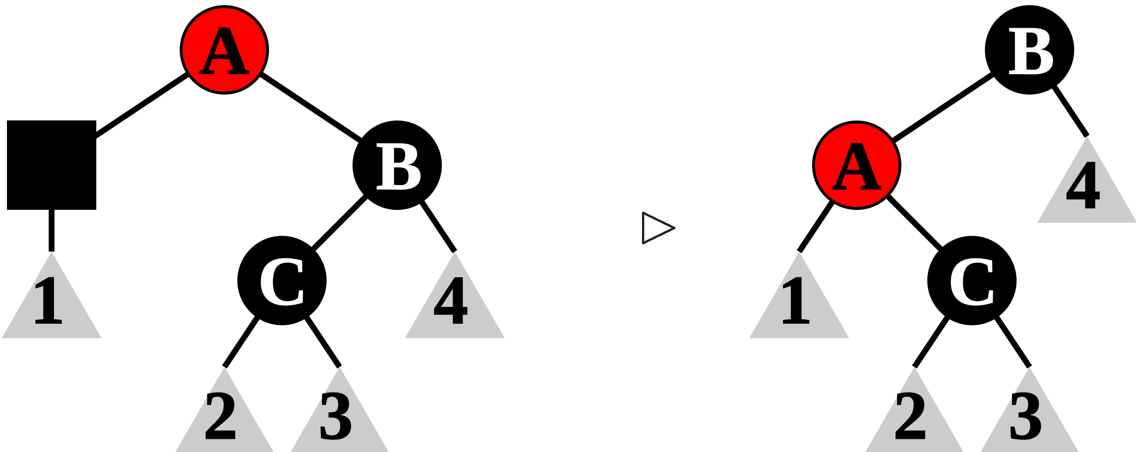


Deletion: transitions 2 and 3

The illegitimate node has a red father and a red closer nephew:

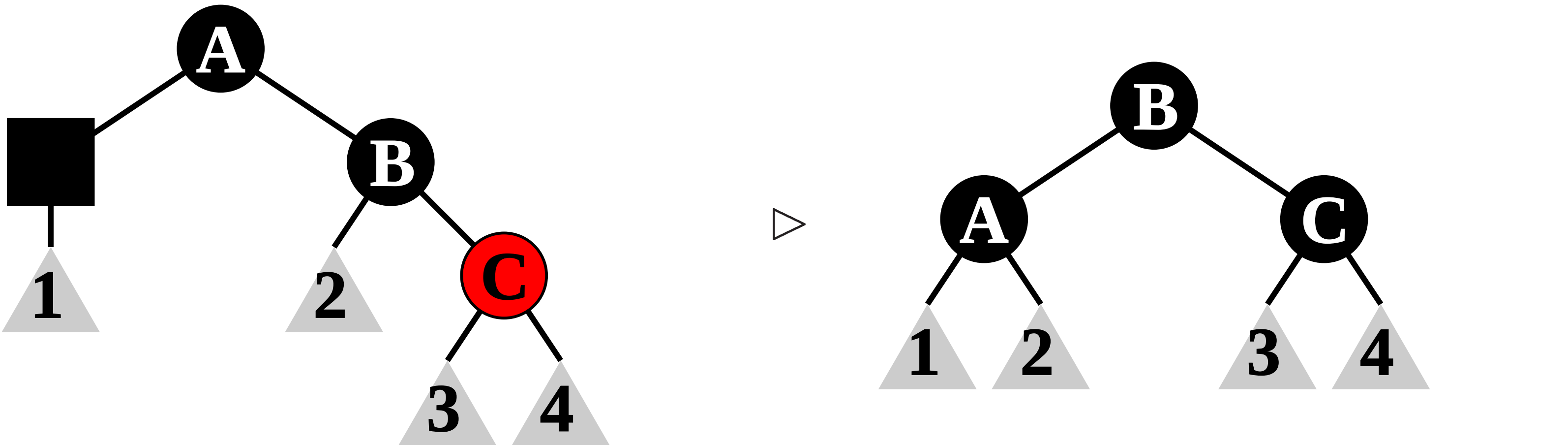


The illegitimate node has a red father and a black closer nephew:



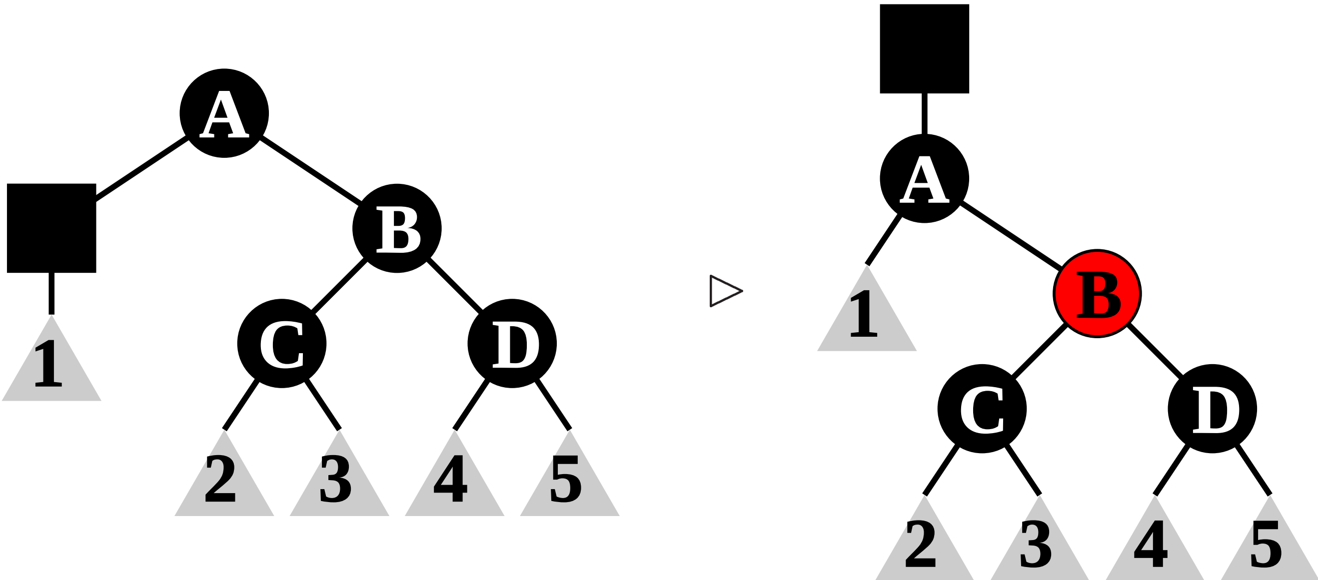
Deletion: transitions 4.1 and 4.2

The illegitimate node has a black father, a black sibling and one red nephew:



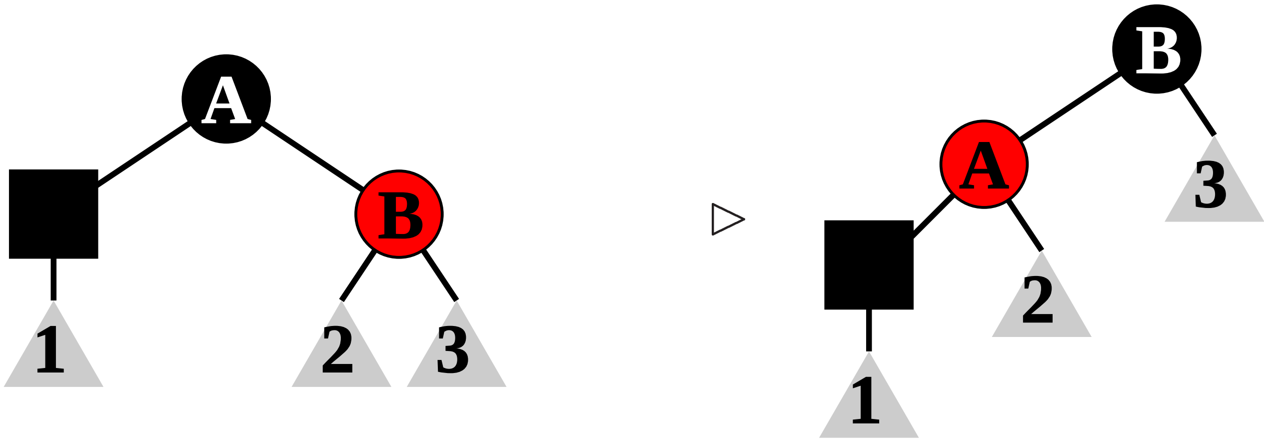
Deletion: transition 5

The illegitimate node has a black father, a black sibling and two black nephews



Deletion: transition 6

The illegitimate node has a black father and a red sibling:



The transition system Commands

Transition system Commands

Configurations: $\{[C, \sigma] \mid C$ a command-sequence, σ a state$\}$

$[x \leftarrow e;\ C', \sigma]$	$\triangleright$	$[C', \sigma\langle x{:}e\rangle]$	
$[\textbf{if}\ b\ \textbf{then}\ C_1\ \textbf{else}\ C_2;\ C', \sigma]$	$\triangleright$	$[C_1;\ C', \sigma]$	**if** $\sigma(b) = \mathsf{true}$
$[\textbf{if}\ b\ \textbf{then}\ C_1\ \textbf{else}\ C_2;\ C', \sigma]$	$\triangleright$	$[C_2;\ C', \sigma]$	**if** $\sigma(b) = \mathsf{false}$
$[\textbf{while}\ b\ \textbf{do}\ C;\ C', \sigma]$	$\triangleright$	$[C;\ \textbf{while}\ b\ \textbf{do}\ C;\ C', \sigma]$	**if** $\sigma(b) = \mathsf{true}$
$[\textbf{while}\ b\ \textbf{do}\ C;\ C', \sigma]$	$\triangleright$	$[C', \sigma]$	**if** $\sigma(b) = \mathsf{false}$

Euclid's algorithm

Algorithm Euclid(m, n)

Input : $m, n \geq 1$

Output : $r = \mathrm{gcd}(m_0, n_0)$

Method : **while** $m \neq n$ **do**

if $m > n$ **then**

$m \leftarrow m - n$

else

$n \leftarrow n - m;$

$r \leftarrow m$

Execution of *Euclid(72,45)*

Step#	**Action**	**State#**	**m**	**n**	**r**
		0	**72**	**45**	
1	m<>n				
		1	72	45	
2	m > n				
		2	72	45	
3	**m <– m-n**				
		3	**27**	45	
4	m<>n				
		4	27	45	
5	m > n				
		5	27	45	
6	**n <– n-m**				
		6	27	**18**	
7	m<>n				
		7	27	18	
8	m > n				
		8	27	18	
9	**m <– m-n**				
		9	**9**	18	
10	m<>n				
		10	9	18	
11	m > n				
		11	9	18	
12	**n <– n-m**				
		12	9	**9**	
13	m<>n				
		13	9	9	
14	**r <– m**				
		14	9	9	**9**

Correctness

Algorithm A$(\cdots)$	
Input	: *In*
Output	: *Out*
Method	: *C*

Definition 2.3.1 The algorithm A is ***correct*** if any process for the transition system Commands starting in a configuration $[C, \sigma]$, where σ satisfies *In*, is finite and ends with a configuration σ' satisfying *Out*. $\square$

Decorations

$$[\{I\}\textsf{while}\ b\ \textsf{do}\ C; C', \sigma] \rhd [C; \{I\}\textsf{while}\ b\ \textsf{do}\ C; C', \sigma]\ \textbf{if}\ \sigma(b) = \textsf{true}.$$

Definition 2.3.2 An assertion U of a decorated algorithm is ***valid for a process*** if for all configurations of the form $[\{U\}C, \sigma]$ in the process, the assertion U is satisfied by the state σ. □

Euclid's algorithm (decorated)

Algorithm Euclid(m, n)
Input : $m, n \geq 1$
Output : $r = \text{gcd}(m_0, n_0)$
Method : $\{I\}$**while** $m \neq n$ **do**
    if $m > n$ **then**
        $m \leftarrow m - n$
    else
        $n \leftarrow n - m$;
  $r \leftarrow m$

$$I : \text{gcd}(m, n) = \text{gcd}(m_0, n_0),$$

Validity

Algorithm A$(\cdots)$	
Input	: *In*
Output	: *Out*
Method	: *C*

Definition 2.3.3 The algorithm A is ***valid*** if all its assertions are valid for all processes starting in a configuration $[C, \sigma]$ where σ satisfies *In*. $\square$

Proof-burdens

$$\{U\}C\{V\}$$

For any state σ, if σ satisfies U and the execution of C in σ leads to σ' (ie. $[C, \sigma] \triangleright^* \sigma'$), then σ' must satisfy V.

Proof principle for simple proof-burdens

Proof principle for simple proof-burdens Let $C = c_1; \cdots ; c_k$ be a sequence of assignments and let $x_1, \ldots, x_n$ be the variables of the algorithm. Suppose that C executed in σ leads to σ'. Then the proof-burden $\{U\}C\{V\}$ is proved by proving the implication

$$U(x_1, \ldots, x_n) \Rightarrow V(x'_1, \ldots, x'_n)$$

—where $x'_1, \ldots, x'_n$ are the values of the variables in σ' expressed as functions of their values in σ.

Proof principle for compound proof-burdens

Proof principle for compound proof-burdens A proof-burden of the form $\{U\}C_1; C_2\{V\}$ gives rise to the proof-burdens

$$\{U\}C_1\{W\} \quad \text{and} \quad \{W\}C_2\{V\}.$$

A proof-burden of the form $\{U\}\textbf{if } b \textbf{ then } C_1 \textbf{ else } C_2\{V\}$ gives rise to the proof-burdens

$$\{U \wedge b\}C_1\{V\} \quad \text{and} \quad \{U \wedge \neg b\}C_2\{V\}.$$

A proof-burden of the form $\{U\}\textbf{while } b \textbf{ do } C\{V\}$ gives rise to the proof-burdens

$$\begin{array}{ll} U \Rightarrow I & \text{(basis)} \\ \{I \wedge b\}C\{I\} & \text{(invariance)} \\ I \wedge \neg b \Rightarrow V & \text{(conclusion).} \end{array}$$

Termination

Algorithm A$(\cdots)$	
Input	: *In*
Output	: *Out*
Method	: C

Definition 2.3.4 We say that A ***terminates*** if any process starting in a configuration $[C, \sigma]$, where σ satisfies *In*, is finite. □

Termination principle

Termination principle for algorithms Let $x_1, \ldots, x_n$ be the variables of an algoritm A. A terminates if for every loop

$$\{I\}\textbf{while } b \textbf{ do } C$$

in its method with I as valid invariant, there exists an integer-valued function $\mu(x_1, \ldots, x_n)$ satisfying

a) $I \Rightarrow \mu(x_1, \ldots, x_n) \geq 0$

b) $I \wedge b \Rightarrow \mu(x_1, \ldots, x_n) > \mu(x'_1, \ldots, x'_n) \geq 0$

—where $x'_1, \ldots, x'_n$ are the values of the variables after an iteration expressed as functions of their values before.

Extended version of Euclid's algorithm

Algorithm ExtendedEuclid(m, n)

Input : $m, n \geq 1$

Output : $(r = \gcd(m_0, n_0)) \wedge (s = \mathrm{lcm}(m_0, n_0))$

Method : $p \leftarrow m;\ q \leftarrow n;$

$\{I\}$**while** $m \neq n$ **do**

  if $m > n$ **then**

    $m \leftarrow m - n;\ p \leftarrow p + q$

  else

    $n \leftarrow n - m;\ q \leftarrow q + p;$

$r \leftarrow m;\ s \leftarrow (p + q)/2$

$$I : (mq + np = 2m_0n_0) \wedge (\gcd(m, n) = \gcd(m_0, n_0)).$$

Factorial

Algorithm Factorial(n)
Input : $n \geq 0$
Output : $r = n_0!$
Method : $r \leftarrow 1$;
$\{I\}$**while** $n \neq 0$ **do**
  $r \leftarrow r * n$;
  $n \leftarrow n - 1$;

$$I : (r = n_0!/n!) \wedge (n_0 \geq n \geq 0).$$

Power sum

Algorithm PowerSum(x, n)

Input : $(x \neq 0) \wedge (n \geq 0)$
Constants: x, n
Output : $r = \Sigma_{i=0}^{n} x^i$
Method : $r \leftarrow 1;\ m \leftarrow 0;$
$\{I\}$**while** $m \neq n$ **do**
$r \leftarrow r * x + 1;$
$m \leftarrow m + 1;$

$$I : (r = \Sigma_{i=0}^{m} x^i) \wedge (n \geq m \geq 0).$$

Finding maximum in an array

Algorithm ArrayMax(A)
Input : true
Constants: A
Output : $r = \max A$
Method : $r \leftarrow -\infty$; $i \leftarrow 0$;
$\{I\}$**while** $i \neq |A|$ **do**
if $r < A[i]$ **then** $r \leftarrow A[i]$;
$i \leftarrow i + 1$

$$I : (0 \leq i \leq |A|) \wedge (r = \max A[0..i]) \qquad \mu(A, i, r) = |A| - i$$

Time complexity

Algorithm $\mathsf{A}(x_1, \ldots, x_n)$	
Input	: In
Output	: Out
Method	: C

Definition 2.6.1 The ***time complexity of*** A is the function $T[\![\mathsf{A}]\!]$ taking a state σ satisfying In to the length of the process starting at $[C, \sigma]$. □

We can regard $T[\![\mathsf{A}]\!]$ as a function of the input parameters $x_1, \ldots, x_n$

Time complexity for ArrayMax

	Steps
$r \leftarrow -\infty;\ i \leftarrow 0;$	2
$\{I\}$**while** $i \neq \vert A\vert$ **do**	$\vert A\vert + 1$
if $r < A[i]$ **then** $r \leftarrow A[i];$	between $\vert A\vert$ and $2\vert A\vert$
$i \leftarrow i + 1$	$\vert A\vert$

Total number of steps:

$$3|A| + 3 \leq T[\![\mathsf{ArrayMax}]\!](A) \leq 4|A| + 3$$

More precisely, $T[\![\mathsf{ArrayMax}]\!](A)$ equals $3|A| + 3$ plus the number of times $A[i]$ is strictly larger than max $A[0..i]$, with i running through the indices $0, \ldots, |A| - 1$.

Time complexity for Euclid

while $m \neq n$ **do**
  if $m > n$ **then**
    $m \leftarrow m - n$
  else
    $n \leftarrow n - m$

m	n	T	m	n	T	m	n	T	m	n	T	m	n	T	$\cdots$
1	1	2	2	1	5	3	1	8	4	1	11	5	1	14	$\cdots$
1	2	5	2	2	2	3	2	8	4	2	5	5	2	11	$\cdots$
1	3	8	2	3	8	3	3	2	4	3	11	5	3	11	$\cdots$
1	4	11	2	4	5	3	4	11	4	4	2	5	4	14	$\cdots$
1	5	14	2	5	11	3	5	11	4	5	14	5	5	2	$\cdots$
1	6	17	2	6	8	3	6	5	4	6	8	5	6	17	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	

Worst-case time complexity

Algorithm $\mathsf{A}(x_1, \ldots, x_n)$	
Input	: In
Output	: Out
Method	: C

Definition 2.6.2 Let *size* be a function mapping states satisfying *In* to non-negative integers. The ***worst-case time complexity of*** A is the function mapping $n \geq 0$ to the maximum of $T[\![\mathsf{A}]\!](\sigma)$ for states σ (satisfying *In*) with $size(\sigma) = n$. □

We can regard *size* as a function of the input parameters $x_1, \ldots, x_n$, and write

$$T[\![\mathsf{ArrayMax}]\!](A) = 4|A| + 3 \in \mathcal{O}(|A|)$$

Exponentiation in linear time

Algorithm $\mathsf{LinExp}(x, p)$
Input : $p \geq 0$
Constants: x, p
Output : $r = x^p$
Method : $r \leftarrow 1;\ q \leftarrow p;$
$\{I\}$**while** $q > 0$ **do**
$r \leftarrow r * x;\ q \leftarrow q - 1$

$$I : (rx^q = x^p) \wedge (q \geq 0) \qquad \mu(x, p, r, q) = q$$

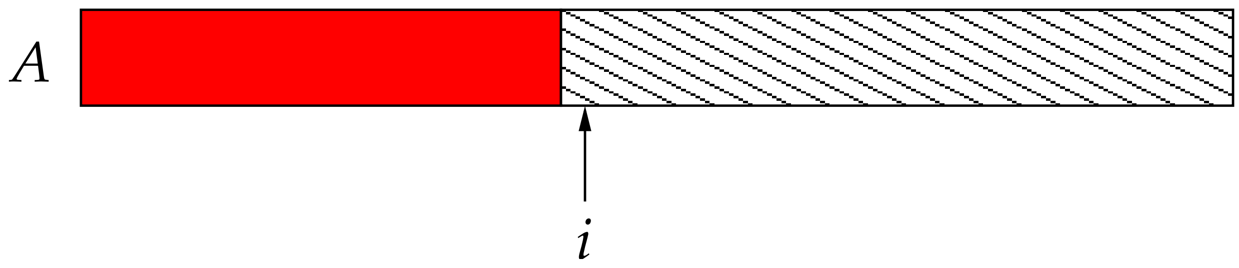
Exponentiation in logarithmic time

Algorithm LogExp(x, p)
Input : $p \geq 0$
Constants: x, p
Output : $r = x^p$
Method : $r \leftarrow 1;\ q \leftarrow p;\ h \leftarrow x;$

```
{I} while q > 0 do
        if q even then
                q ← q/2; h ← h * h
        else
                q ← q − 1; r ← r * h
```

$$I : (rh^q = x^p) \wedge (q \geq 0) \qquad \mu(x, p, r, q, h) = q$$

Scanning



$$I : (0 \leq i \leq |A|) \wedge I' \qquad \mu(A, i, \ldots) = |A| - i$$

$$\begin{array}{l} \ll \mathsf{init} \gg;\ i \leftarrow 0; \\ \{I\}\textbf{while } i \neq |A| \textbf{ do} \\ \qquad \ll \mathsf{update} \gg;\ i \leftarrow i + 1; \\ \ll \mathsf{end} \gg \end{array}$$

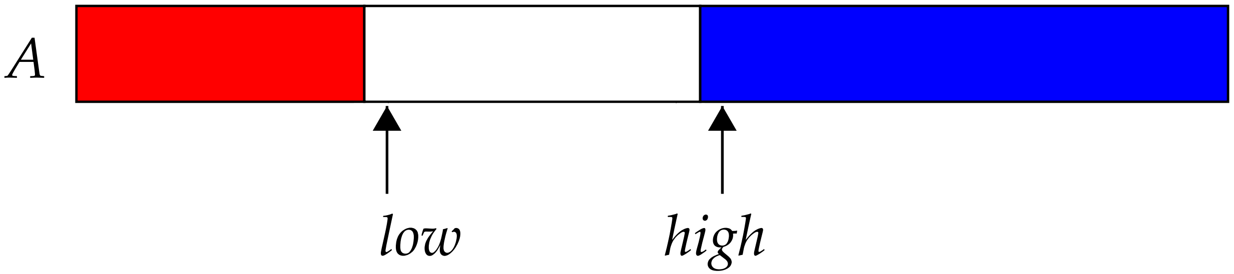
Linear search

Algorithm LinearSearch(A, s)
Input : true
Constants: A, s
Output : $(0 \leq r \leq |A|) \wedge (s \notin A[0..r]) \wedge (r = |A| \vee A[r] = s)$
Method : $r \leftarrow 0$;
$\{I\}$**while** $(r \neq |A|) \wedge (A[r] \neq s)$ **do**
$r \leftarrow r + 1$;

$$I : (0 \leq r \leq |A|) \wedge (s \notin A[0..r]) \qquad \mu(A, s, r) = |A| - r$$

Binary search: specification and invariant

Algorithm BinarySearch(A, s)
Input : A sorted
Constants: A, s
Output : $(0 \leq r \leq |A|) \wedge (A[0..r] < s) \wedge (s \leq A[r..|A|])$



$low \leftarrow 0;\ high \leftarrow |A|;$
$\{I\}$**while** $low \neq high$ **do**
    $\ll$ narrow the gap $\gg$;
$r \leftarrow low$

Binary search: the algorithm

Algorithm BinarySearch(A, s)
Input : A sorted
Constants: A, s
Output : $(0 \le r \le |A|) \wedge (A[0..r] < s) \wedge (s \le A[r..|A|])$
Method : $low \leftarrow 0;\ high \leftarrow |A|;$

```
{I}while low ≠ high do
      m ← (low + high)/2;
      if A[m] < s then
            low ← m + 1
      else
            high ← m;
r ← low
```

$$I : (0 \le low \le high \le |A|) \wedge (A[0..low] < s) \wedge (s \le A[high..|A|])$$

$$\mu(A, s, low, high, m, r) = high - low$$

Insertion sort

Algorithm InsertionSort(A)
Input : true
Output : $(A \text{ perm } A_0) \wedge (A \text{ sorted})$
Method : $i \leftarrow 0;$

```
{I}while i ≠ |A| do
    j ← i;
    {J}while (j ≠ 0) ∧ (A[j − 1] > A[j]) do
        A[j − 1] ↔ A[j];
        j ← j − 1;
    i ← i + 1
```

$$I : (A \text{ perm } A_0) \wedge (0 \leq i \leq |A|) \wedge (A[0..i] \text{ sorted})$$

$$J : (A \text{ perm } A_0) \wedge (0 \leq j \leq i < |A|) \wedge (A[0..j], A[j..i+1] \text{ sorted})$$

$$\mu_I(A, i, j) = |A| - i \qquad \mu_J(A, i, j) = j$$

Merge sort

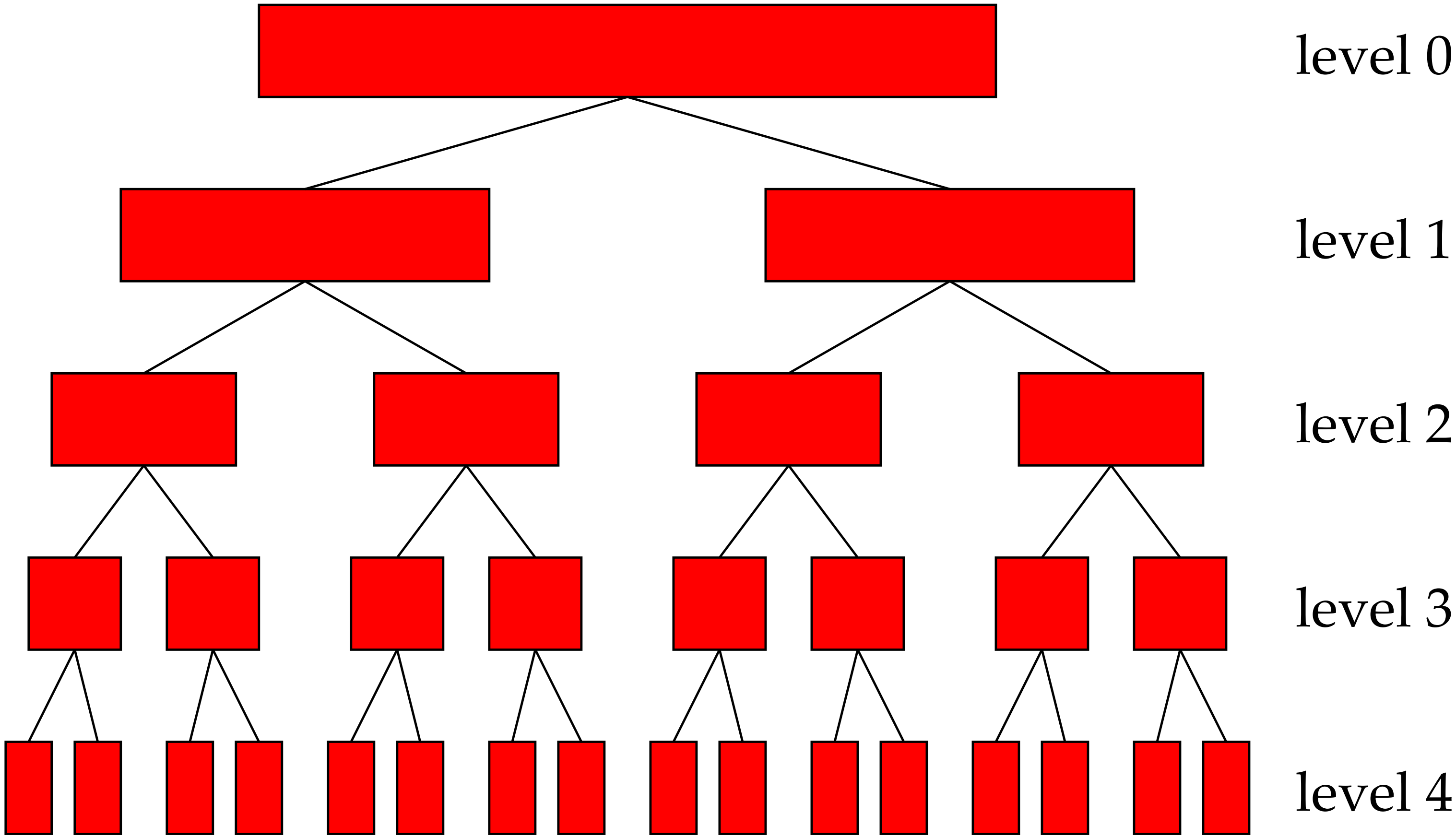
Algorithm $\mathsf{MergeSort}(A)$
Input : true
Output : $(A \text{ perm } A_0) \wedge (A \text{ sorted})$
Method : **if** $|A| > 1$ **then**
$B \leftarrow A[0..|A|/2];$
$C \leftarrow A[|A|/2..|A|];$
$\mathsf{MergeSort}(B);$
$\mathsf{MergeSort}(C);$
$\{W\}\mathsf{Merge}(A, B, C)$

Algorithm $\mathsf{Merge}(A, B, C)$
Input : $(|A| = |B| + |C|) \wedge (B, C \text{ sorted})$
Constants: B, C
Output : $(A \text{ perm } BC) \wedge (A \text{ sorted})$

$$W : (A = A_0) \wedge (B \text{ perm } A_1) \wedge (C \text{ perm } A_2) \wedge (B, C \text{ sorted})$$

$$\mu(A) = |A|$$

Merge sort: time complexity, assuming n a power of 2



Time spent at level i (for $0 \leq i \leq \log n$): $2^i \cdot n/2^i = n$ time units.

Merge: specification

Algorithm Merge(A, B, C)
Input : $(|A| = |B| + |C|) \wedge (B, C \text{ sorted})$
Constants: B, C
Output : $(A \text{ perm } BC) \wedge (A \text{ sorted})$

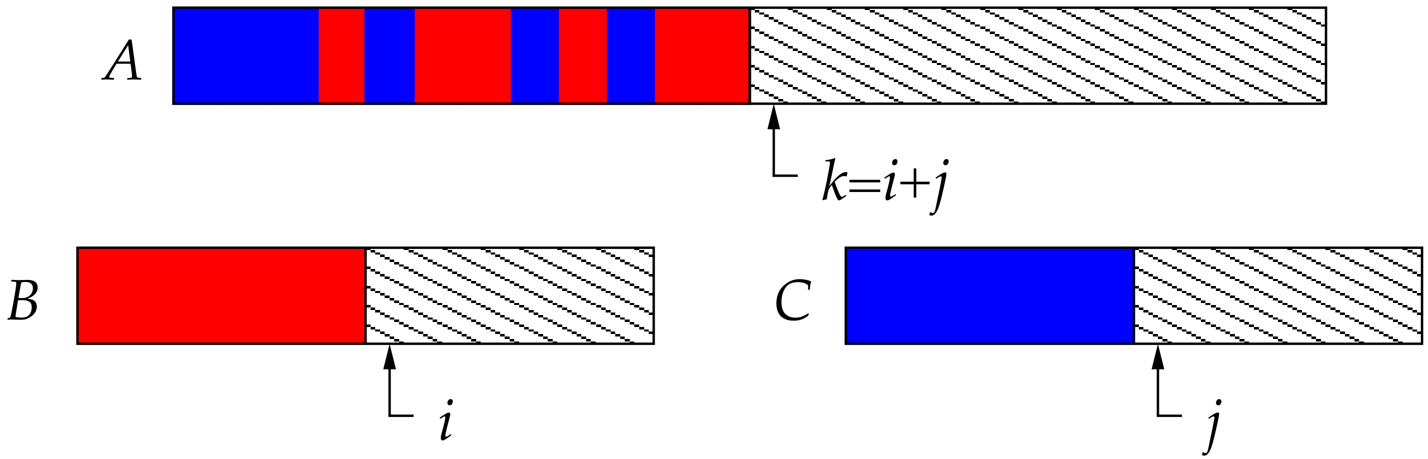
So, with $|A| = 11$ and

$$B = [1, 1, 4, 7, 8] \qquad \text{and} \qquad C = [1, 2, 2, 3, 4, 7]$$

we should obtain

$$A = [1, 1, 1, 2, 2, 3, 4, 4, 7, 7, 8].$$

Merge: scanning invariant



$$I: \ (|A| = |B| + |C|) \wedge (0 \leq i \leq |B|) \wedge (0 \leq j \leq |C|) \wedge$$
$$(A[0..i+j] \text{ perm } B[0..i]C[0..j]) \wedge (A[0..i+j] \text{ sorted})$$

$\ll$ init $\gg$;
$\{I\}$**while** $i + j \neq |A|$ **do**
  $\ll$ update $\gg$;
$\ll$ end $\gg$

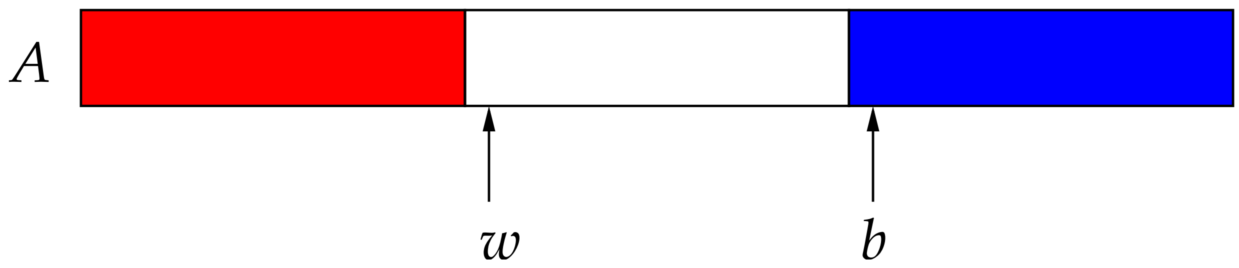
Merge: the algorithm

Algorithm Merge(A, B, C)
Input : $(|A| = |B| + |C|) \wedge (B, C \text{ sorted})$
Constants: B, C
Output : $(A \text{ perm } BC) \wedge (A \text{ sorted})$
Method : $i \leftarrow 0;\ j \leftarrow 0;$
$\{I\}$**while** $i + j \neq |A|$ **do**
  if $(i < |B|) \wedge (j = |C| \vee B[i] \leq C[j])$ **then**
    $A[i + j] \leftarrow B[i];\ i \leftarrow i + 1$
  else
    $A[i + j] \leftarrow C[j];\ j \leftarrow j + 1$

$$I:\ (|A| = |B| + |C|) \wedge (0 \leq i \leq |B|) \wedge (0 \leq j \leq |C|) \wedge (A[0..i + j] \text{ perm } B[0..i]C[0..j]) \wedge (A[0..i + j] \text{ sorted})$$

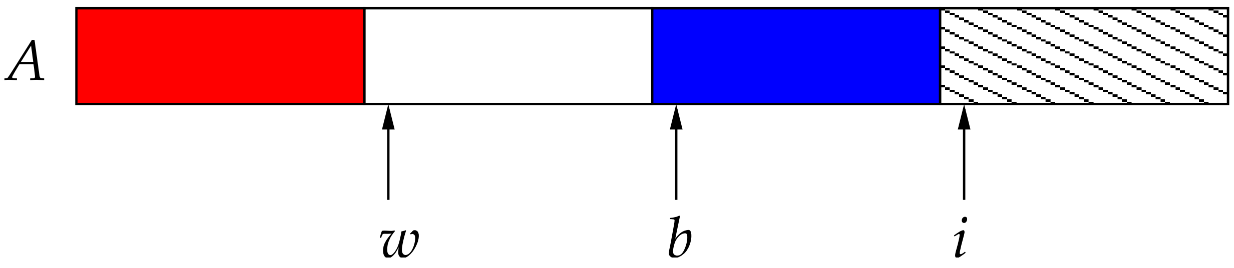
$$\mu(A, B, C, i, j) = |A| - (i + j)$$

Dutch flag: specification



Algorithm DutchFlag(A, s)
Input : true
Constants: s
Output : $(A \text{ perm } A_0) \wedge (0 \leq w \leq b \leq |A|) \wedge$
$(A[0..w] < s) \wedge (A[w..b] = s) \wedge (A[b..|A|] > s)$

Dutch flag: scanning invariant



$$I: \; (A \text{ perm } A_0) \wedge (0 \le w \le b \le i \le |A|) \wedge$$
$$(A[0..w] < s) \wedge (A[w..b] = s) \wedge (A[b..i] > s)$$

$\ll$ init $\gg$; $i \leftarrow 0$;
$\{I\}$**while** $i \neq |A|$ **do**
    $\ll$ update $\gg$; $i \leftarrow i + 1$;
$\ll$ end $\gg$

Dutch flag: basis

$$\{\mathsf{true}\} \ll \mathsf{init} \gg;\ i \leftarrow 0\{I\}$$

$$\ll \mathsf{init} \gg\ = w \leftarrow 0;\ b \leftarrow 0$$

Dutch flag: invariance

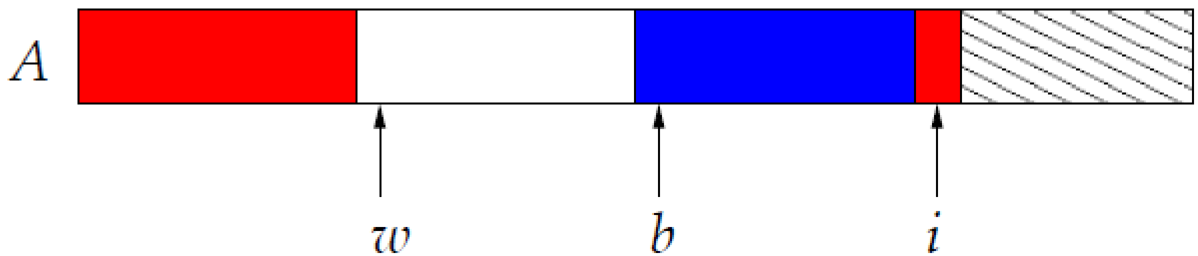
$$\{I \wedge i \neq |A|\} \ll \textsf{update} \gg;\ i \leftarrow i+1 \{I\}$$

$$\begin{aligned}
\ll \textsf{update} \gg = {} & \textbf{if } A[i] < s \textbf{ then} \\
& \qquad \ll \textsf{red} \gg \\
& \textbf{else if } A[i] = s \textbf{ then} \\
& \qquad \ll \textsf{white} \gg \\
& \textbf{else} \\
& \qquad \ll \textsf{blue} \gg
\end{aligned}$$

$$\begin{aligned}
1 :&\ \{I \wedge (i \neq |A|) \wedge (A[i] < s)\} \ll \textsf{red} \gg;\ i \leftarrow i+1 \{I\} \\
2 :&\ \{I \wedge (i \neq |A|) \wedge (A[i] = s)\} \ll \textsf{white} \gg;\ i \leftarrow i+1 \{I\} \\
3 :&\ \{I \wedge (i \neq |A|) \wedge (A[i] > s)\} \ll \textsf{blue} \gg;\ i \leftarrow i+1 \{I\}
\end{aligned}$$

Dutch flag: invariance, red

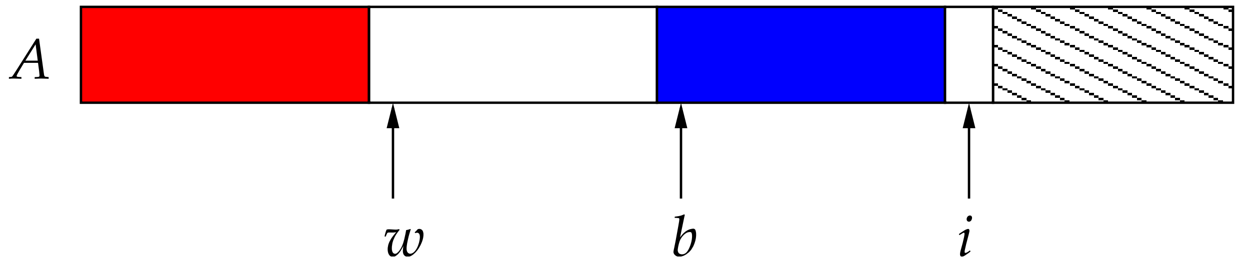
$$\{I \wedge (i \neq |A|) \wedge (A[i] < s)\} \ll \mathsf{red} \gg;\ i \leftarrow i + 1 \{I\}$$



$$\ll \mathsf{red} \gg \;=\; A[i] \leftrightarrow A[b];\ A[b] \leftrightarrow A[w];$$
$$w \leftarrow w + 1;\ b \leftarrow b + 1.$$

Dutch flag: invariance, white

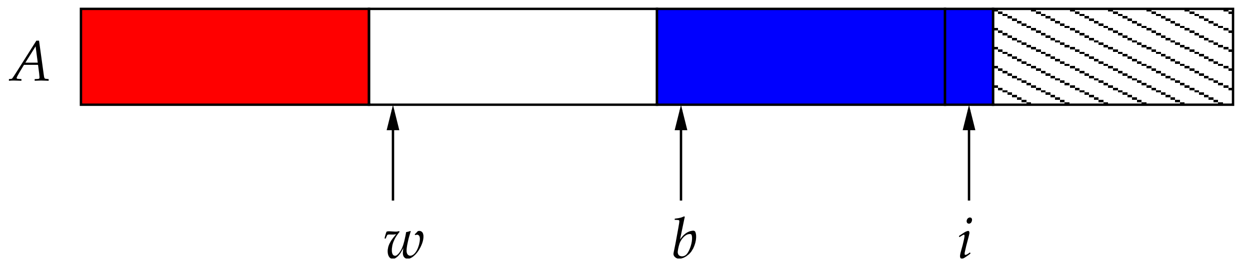
$$\{I \wedge (i \neq |A|) \wedge (A[i] = s)\} \ll \mathsf{white} \gg;\ i \leftarrow i + 1 \{I\}$$



$$\ll \mathsf{white} \gg\ = A[i] \leftrightarrow A[b];\ b \leftarrow b + 1$$

Dutch flag: invariance, blue

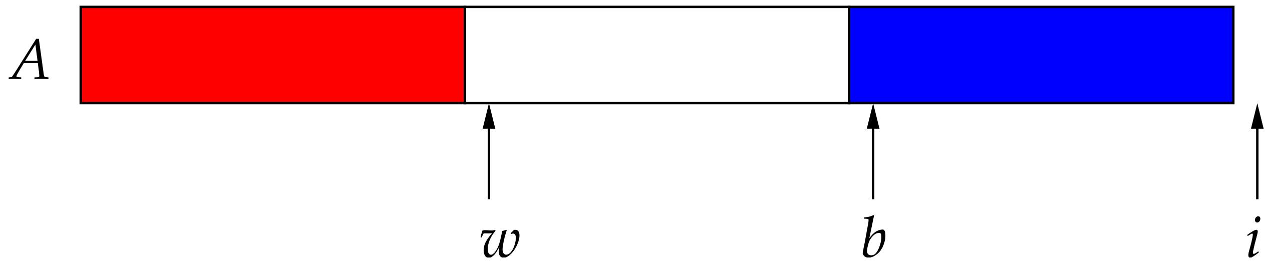
$$\{I \wedge (i \neq |A|) \wedge (A[i] > s)\} \ll \mathsf{blue} \gg;\ i \leftarrow i+1 \{I\}$$



$$\ll \mathsf{blue} \gg = \lambda$$

Dutch flag: conclusion

$$\{I \wedge (i = |A|)\} \ll \mathsf{end} \gg \{Out\}$$



$$\ll \mathsf{end} \gg = \lambda$$

Dutch flag: the algorithm

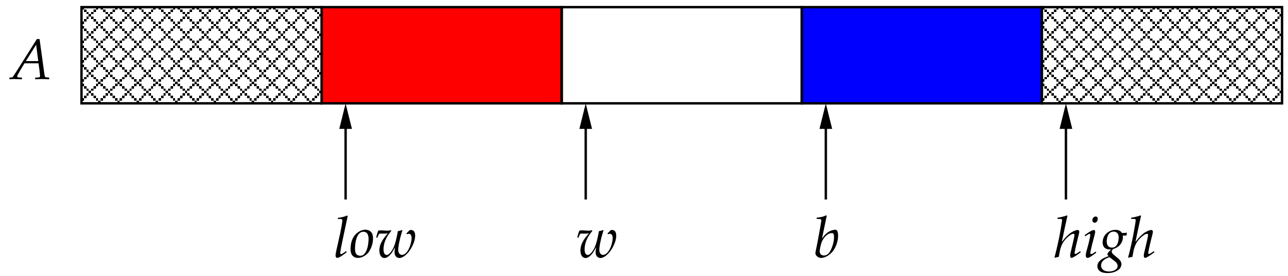
Algorithm DutchFlag(A, s)
Input : true
Constants: s
Output : $(A \text{ perm } A_0) \wedge (0 \leq w \leq b \leq |A|) \wedge$
$(A[0..w] < s) \wedge (A[w..b] = s) \wedge (A[b..|A|] > s)$
Method : $w \leftarrow 0;\ b \leftarrow 0;\ i \leftarrow 0;$

```
{I}while i ≠ |A| do
    if A[i] < s then
        A[i] ↔ A[b]; A[b] ↔ A[w];
        w ← w + 1; b ← b + 1
    else if A[i] = s then
        A[i] ↔ A[b]; b ← b + 1
    i ← i + 1
```

$$I:\ (A \text{ perm } A_0) \wedge (0 \leq w \leq b \leq i \leq |A|) \wedge$$
$$(A[0..w] < s) \wedge (A[w..b] = s) \wedge (A[b..i] > s)$$

$$\mu(A, s, w, b, i) = |A| - i$$

Dutch flag: from *low* to *high*



Algorithm DutchFlag$(A, low, high, s)$
Input : $0 \leq low \leq high \leq |A|$
Constants: $low, high, A[0..low], A[high..|A|], s$
Output : $(A[low..high]$ perm $A_0[low..high]) \wedge$
$(low \leq w \leq b \leq high) \wedge$
$(A[low..w] < s) \wedge (A[w..b] = s) \wedge (A[b..high] > s)$

Quick sort

Algorithm $\mathsf{QuickSort}(A, low, high)$
Input : $0 \le low \le high \le |A|$
Constants: $low, high, A[0..low], A[high..|A|]$
Output : $(A[low..high] \text{ perm } A_0[low..high]) \wedge (A[low..high] \text{ sorted})$
Method : **if** $high - low > 1$ **then**
$r \leftarrow \mathsf{random}(low, high)$;
$(w, b) \leftarrow \mathsf{DutchFlag}(A, low, high, A[r])$;
$\mathsf{QuickSort}(A, low, w)$;
$\mathsf{QuickSort}(A, b, high)$

$$\mu(A, low, high) = high - low$$

Quick select

Algorithm $\mathsf{QuickSelect}(A, low, high, k)$
Input : $(0 \leq low \leq k < high \leq |A|)$
Constants: $low, high, A[0..low], A[high..|A|], k$
Output : $(A[low..high] \text{ perm } A_0[low..high]) \wedge$
$(A[low..k] \leq A[k] \leq A[k..high])$
Method : $r \leftarrow \mathsf{random}(low, high);$
$(w, b) \leftarrow \mathsf{DutchFlag}(A, low, high, A[r]);$
if $k < w$ **then**
$\quad \mathsf{QuickSelect}(A, low, w, k)$
else if $k \geq b$ **then**
$\quad \mathsf{QuickSelect}(A, b, high, k)$

$$\mu(A, low, high) = high - low$$

Maximal subsum

Algorithm MaxSubsum(A)
Input : true
Constants: A
Output : $r = \text{ms}(A)$
Method : $\ll$ init $\gg$; $i \leftarrow 0$;
$\{I\}$**while** $i \neq |A|$ **do**
$\ll$ update $\gg$; $i \leftarrow i + 1$;
$\ll$ end $\gg$

$$I : (0 \leq i \leq |A|) \wedge (r = \text{ms}(A[0..i])) \wedge \cdots$$

Maximal subsum: maintaining the invariant

$\mathrm{ms}(A[0..i+1])$ is the maximum of

the maximal subsum obtained by not using $A[i]$, and
the maximal subsum obtained by using $A[i]$

So, we remember also the *maximal right subsum* $\mathrm{mrs}(A[0..i])$

$$I : (0 \leq i \leq |A|) \wedge (r = \mathrm{ms}(A[0..i])) \wedge (h = \mathrm{mrs}(A[0..i]))$$

Maximal subsum: the algorithm

Algorithm MaxSubsum(A)
Input : true
Constants: A
Output : $m = \text{ms}(A)$
Method : $r \leftarrow 0;\ h \leftarrow 0;\ i \leftarrow 0;$

$\{I\}$**while** $i \neq |A|$ **do**

$h \leftarrow \max\{h + A[i], 0\};$

$r \leftarrow \max\{r, h\};$

$i \leftarrow i + 1$

$$\mu(A, r, h, i) = |A| - i$$

Longest monotone sequence

Algorithm $\mathsf{LLMS}(A)$
Input : true
Constants: A
Output : $r = \mathrm{llms}(A)$
Method : $\ll \mathsf{init} \gg;\ i \leftarrow 0;$
$\{I\}$**while** $i \neq |A|$ **do**
$\ll \mathsf{update} \gg;\ i \leftarrow i + 1;$
$\ll \mathsf{end} \gg$

$$I : (0 \leq i \leq |A|) \wedge (r = \mathrm{llms}(A[0..i])) \wedge \cdots$$

Longest monotone sequence: maintaining the invariant

In the i'th iteration of the loop, $B[l]$ for $1 \leq l \leq |A|$ must contain

the necessary information about a monotone sequence from $A[0..i]$ of length l whose last element is minimal (if there is no sequence of length l, we just record that).

$$I: \ (0 \leq i \leq |A|) \wedge (r = \text{llms}(A[0..i])) \wedge \\ (B[l] = \text{ms}_l(A[0..i]) \text{ for } 0 \leq l \leq |A|)$$

$$\text{ms}_l(A[0..i]) = \begin{cases} -\infty & \text{if } l = 0 \\ \text{minimal last element in a monotone sequence of length } l \text{ from } A[0..i] & \text{if such exists} \\ \infty & \text{otherwise} \end{cases}$$

Longest monotone sequence: the algorithm

Algorithm $\mathsf{LLMS}(A)$
Input : true
Constants: A
Output : $r = \text{llms}(A)$
Method : $B \leftarrow [|A|+1 : \infty];\ B[0] \leftarrow -\infty;\ r \leftarrow 0;\ i \leftarrow 0;$
$\{I\}$**while** $i \neq |A|$ **do**
$l \leftarrow \mathsf{BinarySearch}'(B, A[i]);$
$B[l] \leftarrow A[i];\ r \leftarrow \max\{r, l\};\ i \leftarrow i+1$

$$\mu(A, B, r, i) = |A| - i$$

Algorithm $\mathsf{BinarySearch}'(B, s)$
Input : B sorted
Constants: B, s
Output : $(0 \le l \le |B|) \wedge (B[0..l] \le s) \wedge (s < B[l..|B|])$