Constraint Programming

Disjunctive Scheduling

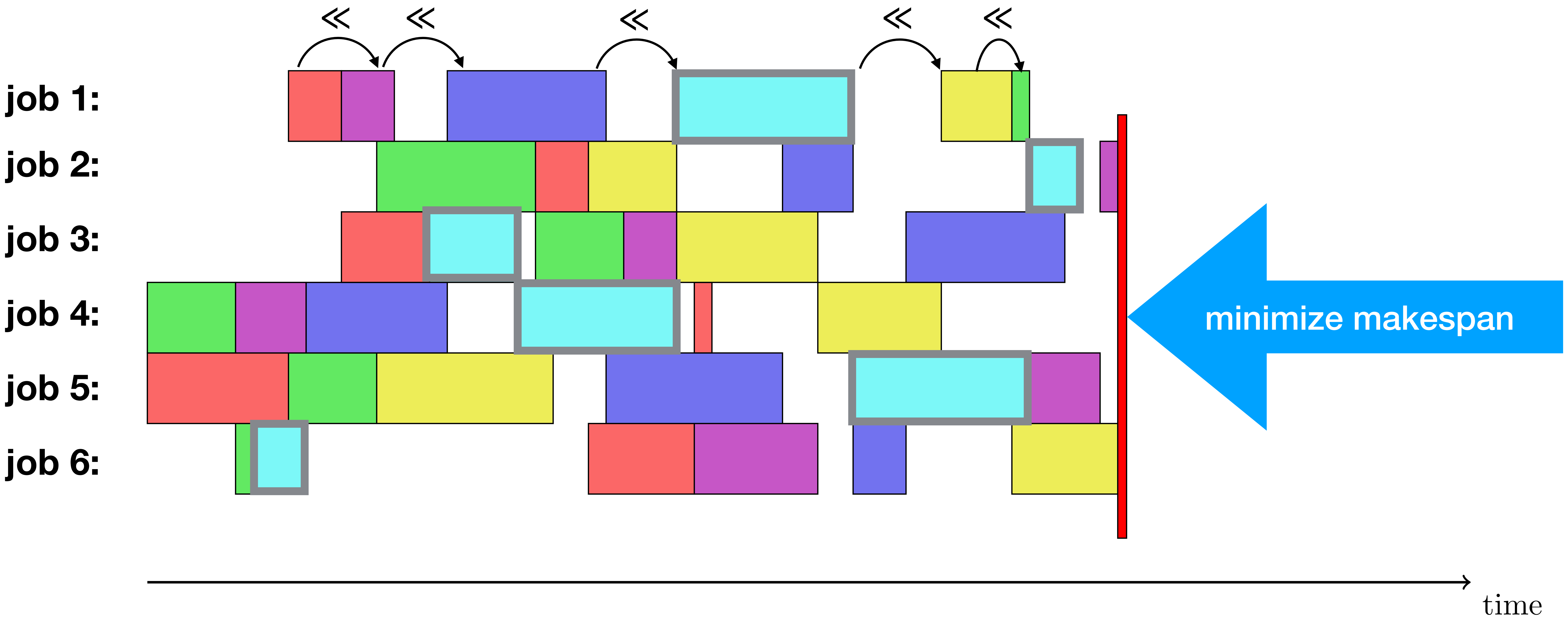
Pierre Schaus

- Disjunctive Decomposition

- Job Shop

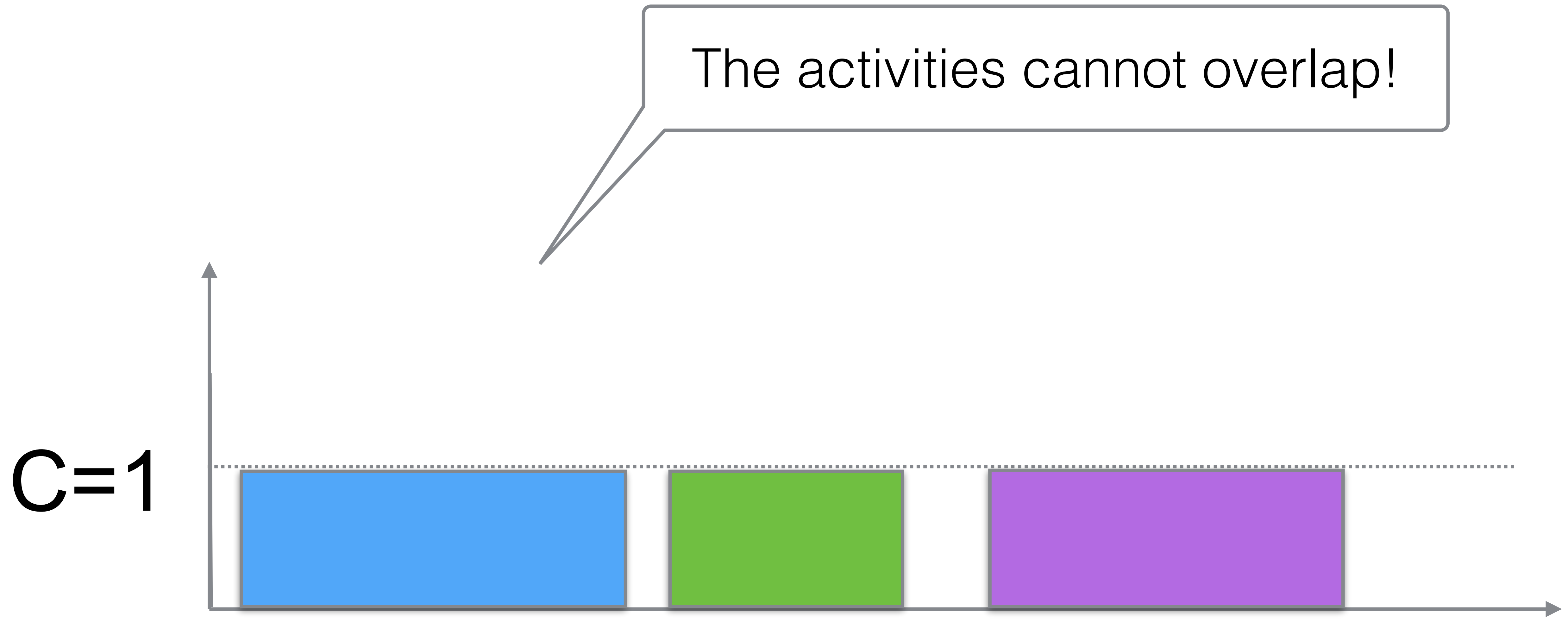
Job-Shop Problem

- Color = resource (or: machine), with capacity 1.
- Precedence constraints (denoted ≪) on the activities of a job.



Disjunctive Resource, aka Unary Resource

It would yield a Cumulative constraint
with all resource requirements $r_i = 1$ and capacity $C = 1$:



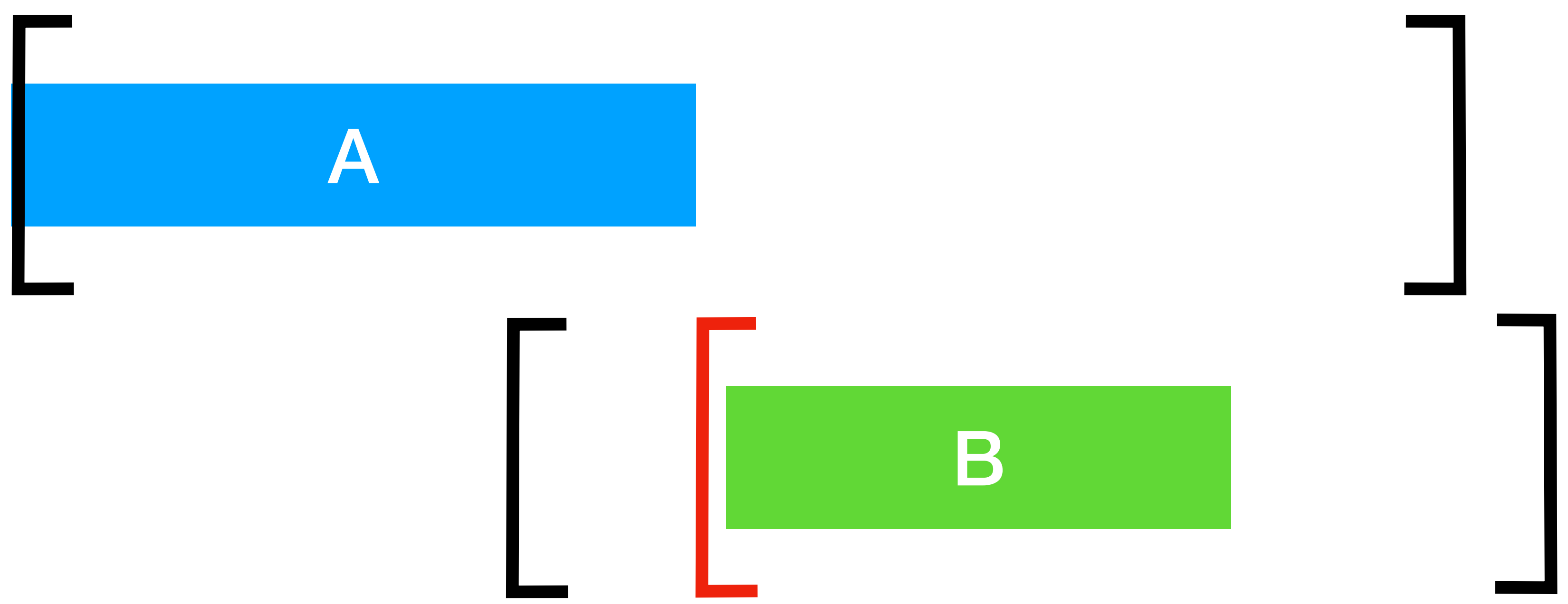
Binary Decomposition for a Unary Resource

- Let T be a set of n activities that cannot overlap.
- $\forall$ i, j $\in$ T where i < j:
  - $b_{ij} \equiv s_i + d_i \leq s_j$
  - $b_{ji} \equiv s_j + d_j \leq s_i$
  - $b_{ij} \neq b_{ji}$ (either i ends before j starts, or vice-versa)

- How does this binary decomposition compare with timetable filtering for Cumulative($[s_1,\ldots,s_n],[d_1,\ldots,d_n],[1,\ldots,1],1$)?

Binary Decomposition: Example

- The binary decomposition with reified constraints is at least as strong as timetable filtering for Cumulative.
- Example where the binary decomposition is *strictly* stronger:

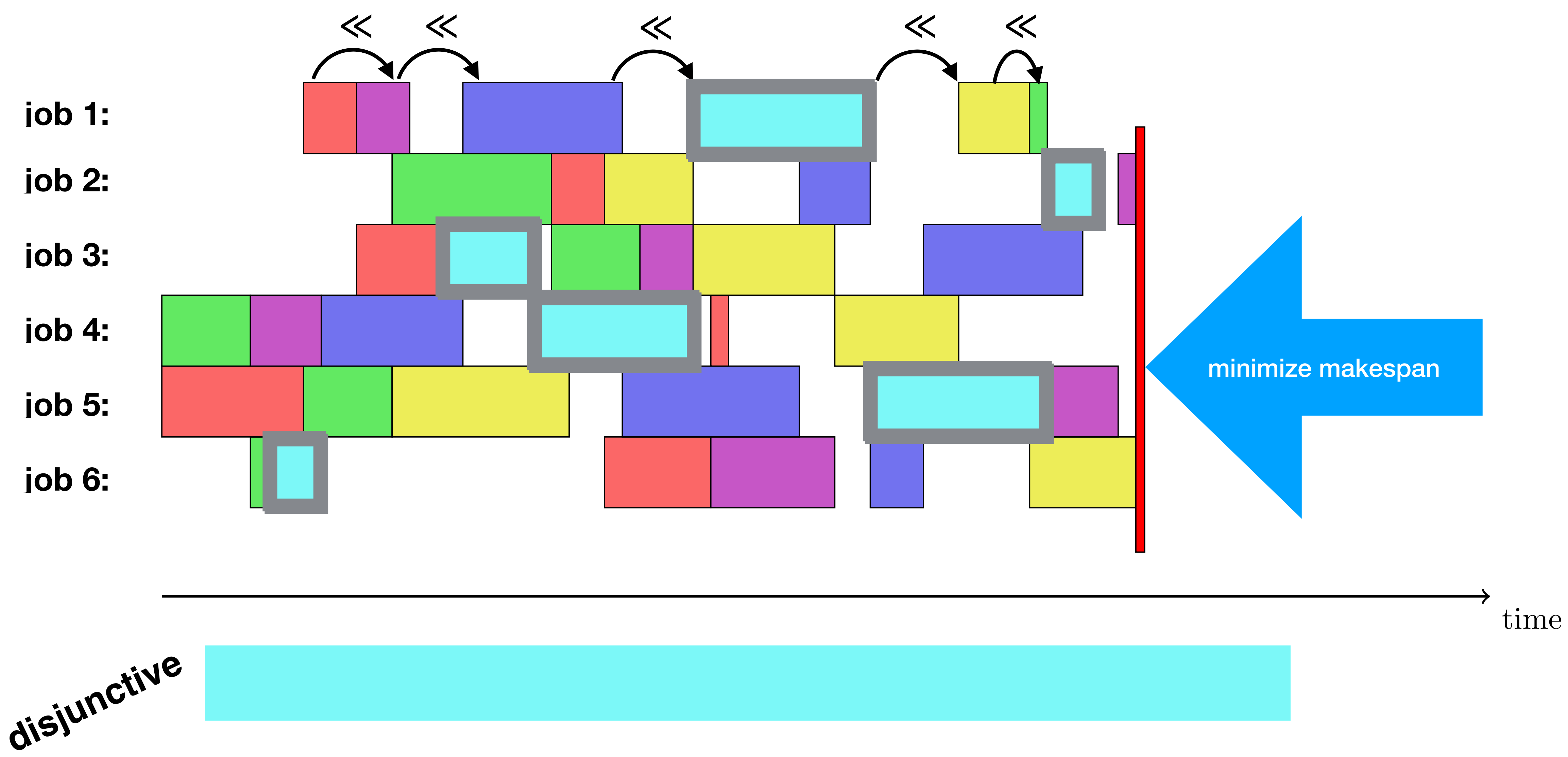


Activity A has no mandatory part:
no pruning for B with timetable filtering!

Job-Shop Model

```
JobShopInstance instance = new JobShopInstance("start");

Solver cp = makeSolver();
// variable creation
IntVar[][] start = new IntVar[instance.nJobs][instance.nMachines];
IntVar[][] end = new IntVar[instance.nJobs][instance.nMachines];
for (int i = 0; i < instance.nJobs; i++) {
    for (int j = 0; j < instance.nMachines; j++) {
        start[i][j] = makeIntVar(cp, 0, instance.horizon);
        end[i][j] = plus(start[i][j], instance.duration[i][j]);
    }
}
// job precedences
for (int i = 0; i < instance.nJobs; i++) {
    for (int j = 1; j < instance.nMachines; j++) {
        cp.post(lessOrEqual(end[i][j - 1], start[i][j]));
    }
}
// disjunctive constraints
for (int m = 0; m < instance.nMachines; m++) {
    // collect activities on machine m
    IntVar[] start_m = instance.collect(start, m);
    int[] dur_m = instance.collect(instance.duration, m);
    cp.post(new Disjunctive(start_m, dur_m));
}
// objective = makespan minimization
IntVar[] endLast = new IntVar[instance.nJobs];
for (int i = 0; i < instance.nJobs; i++) {
    endLast[i] = end[i][instance.nMachines - 1];
}
IntVar makespan = maximum(endLast);
Objective obj = cp.minimize(makespan);
// search to fix the start time of all activities
DFSearch dfs = makeDfs(cp, firstFail(flatten(start)));
```



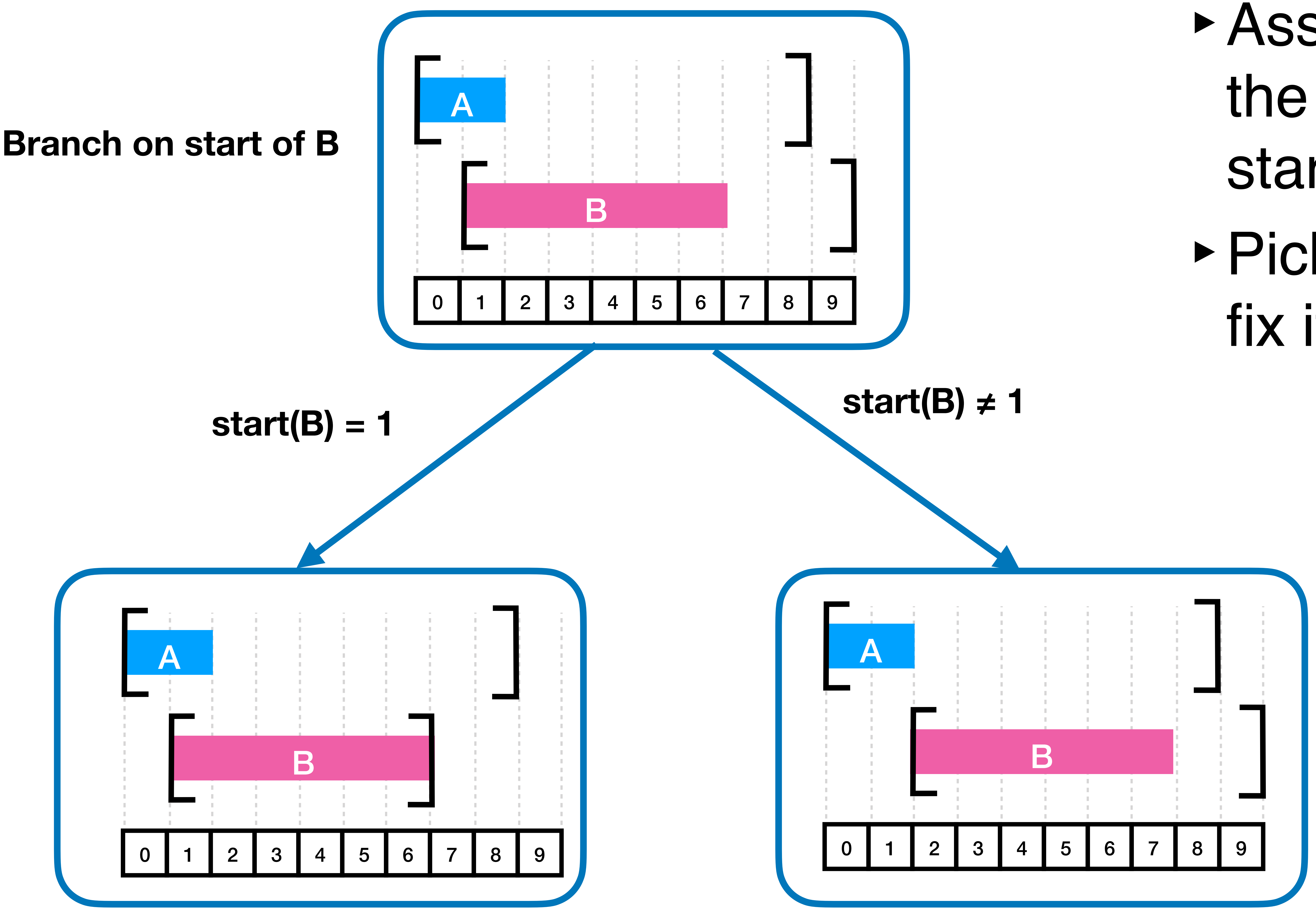
Search for Job Shop

Search for Job Shop

- Two alternatives :

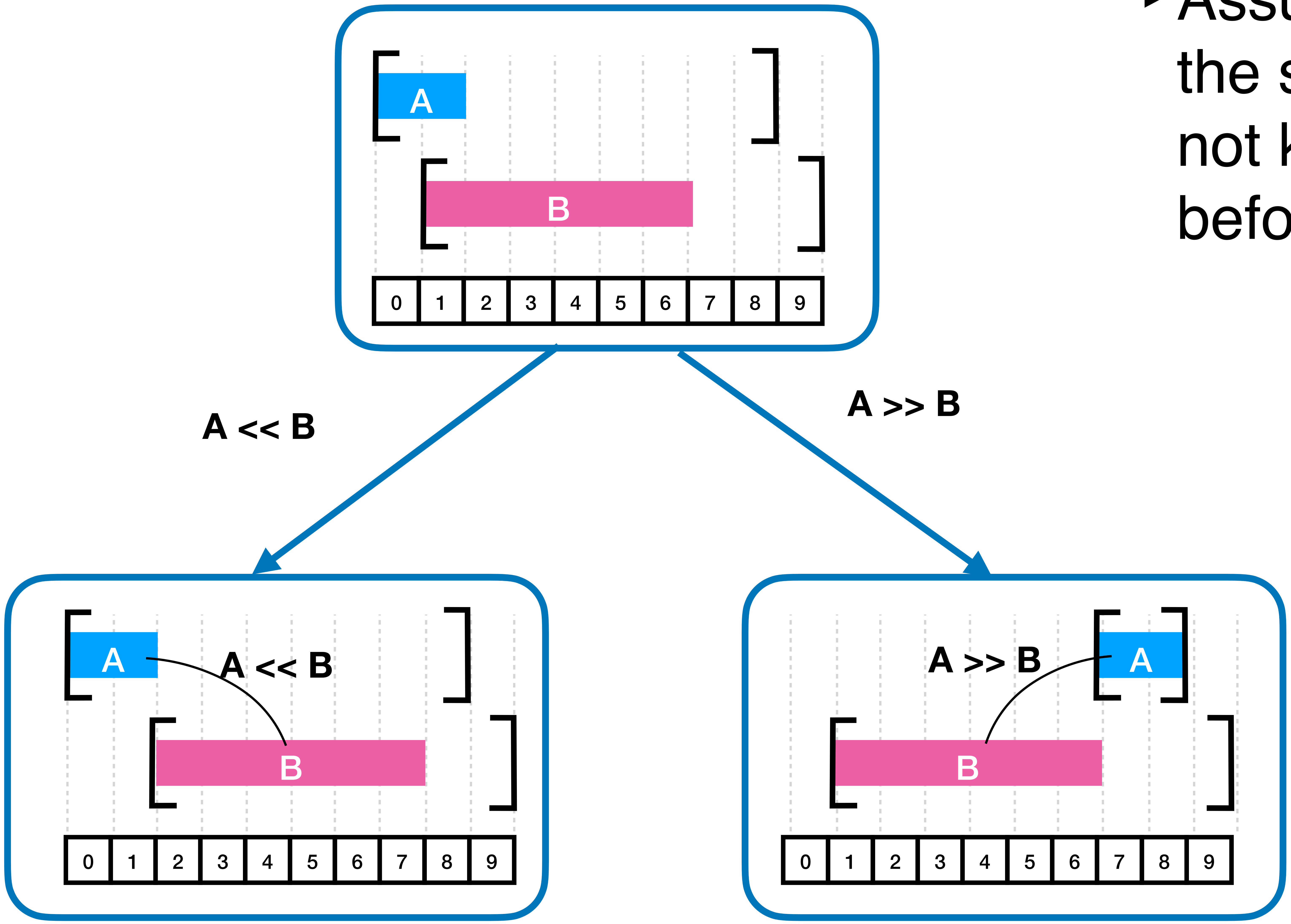
1. Fix the start variables
2. Fix the ordering on each machine (and eventually the start variables)

Search for Job Shop: fix the start variables



- Assume A and B execute on the same machine, and their starts are not yet fixed.
- Pick one, say B, and branch to fix its start.

Search for Job Shop: fix the ordering

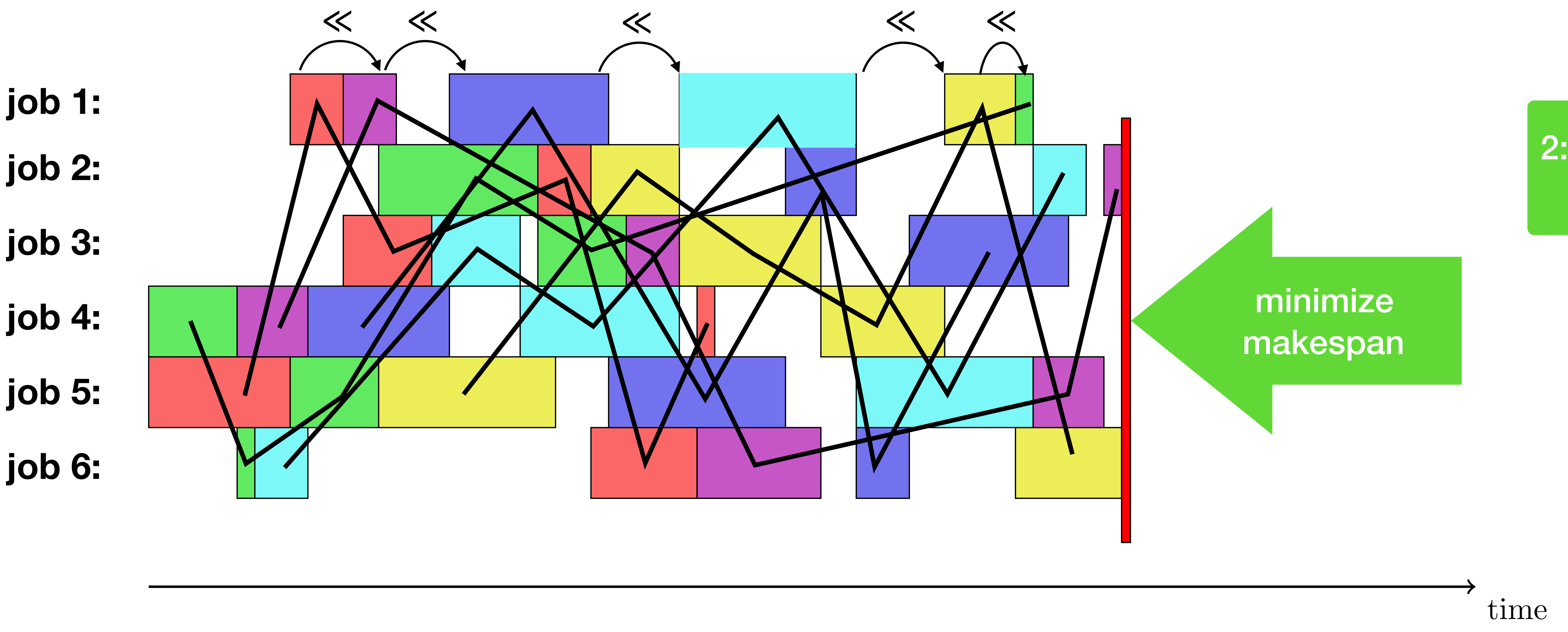


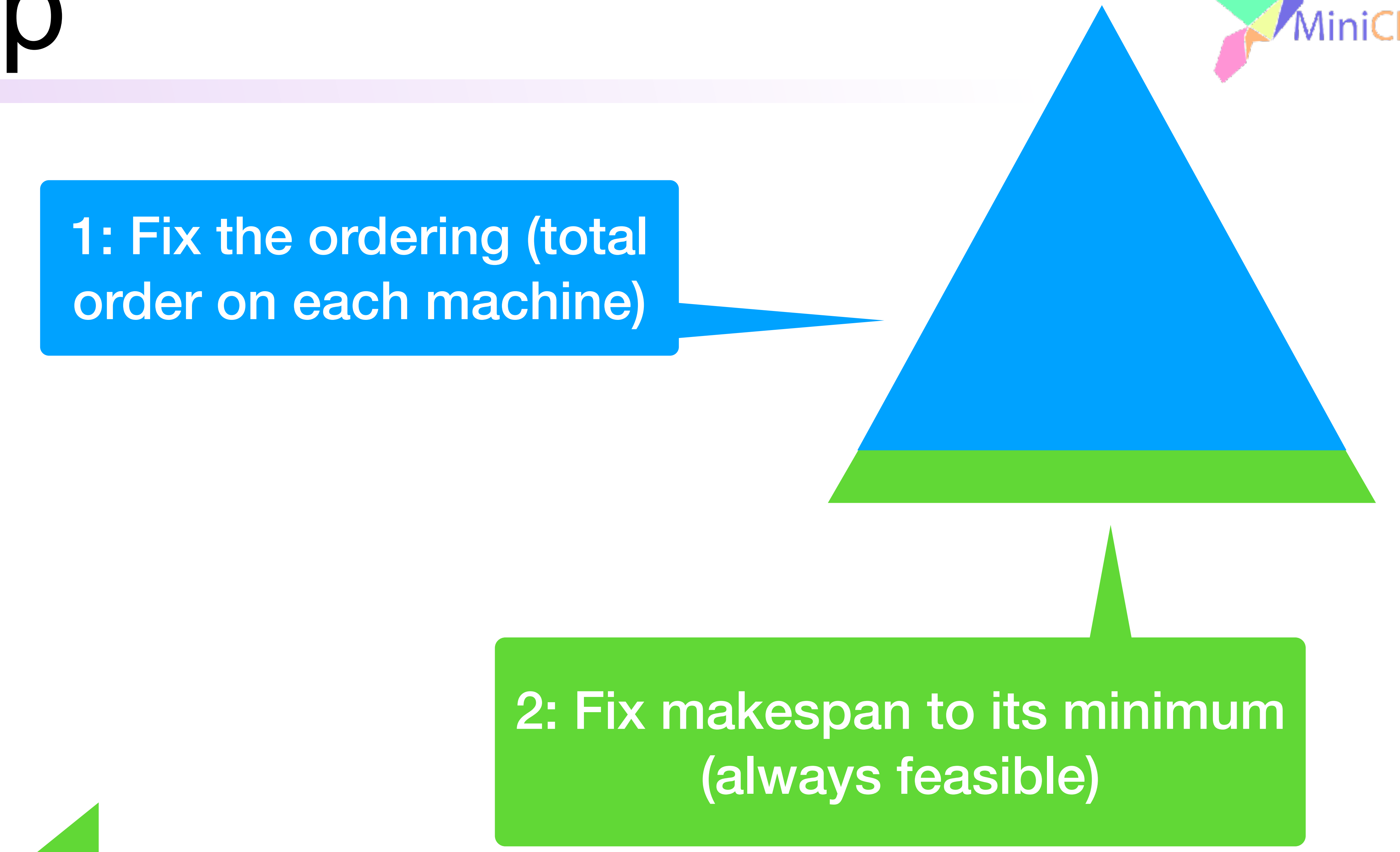
- Assume A and B execute on the same machine, and we do not know yet if A will execute before or after B.

Fixing the ordering

- Post the reified constraints in the model:
  - $\forall$ i, j $\in$ T where i < j:
    - $b_{ij} \equiv s_i + d_i \leq s_j$
    - $b_{ji} \equiv s_j + d_j \leq s_i$
    - $b_{ij} \neq b_{ji}$ (either i ends before j starts, or vice-versa)
- Branch on the b_{ij} variables during the search

Fixing the ordering for the Job Shop





Grimes, D., Hebrard, E., & Malapert, A. (2009). Closing the open shop: Contradicting conventional wisdom. In *International Conference on Principles and Practice of Constraint Programming*, 2009

Earliest Completion Time

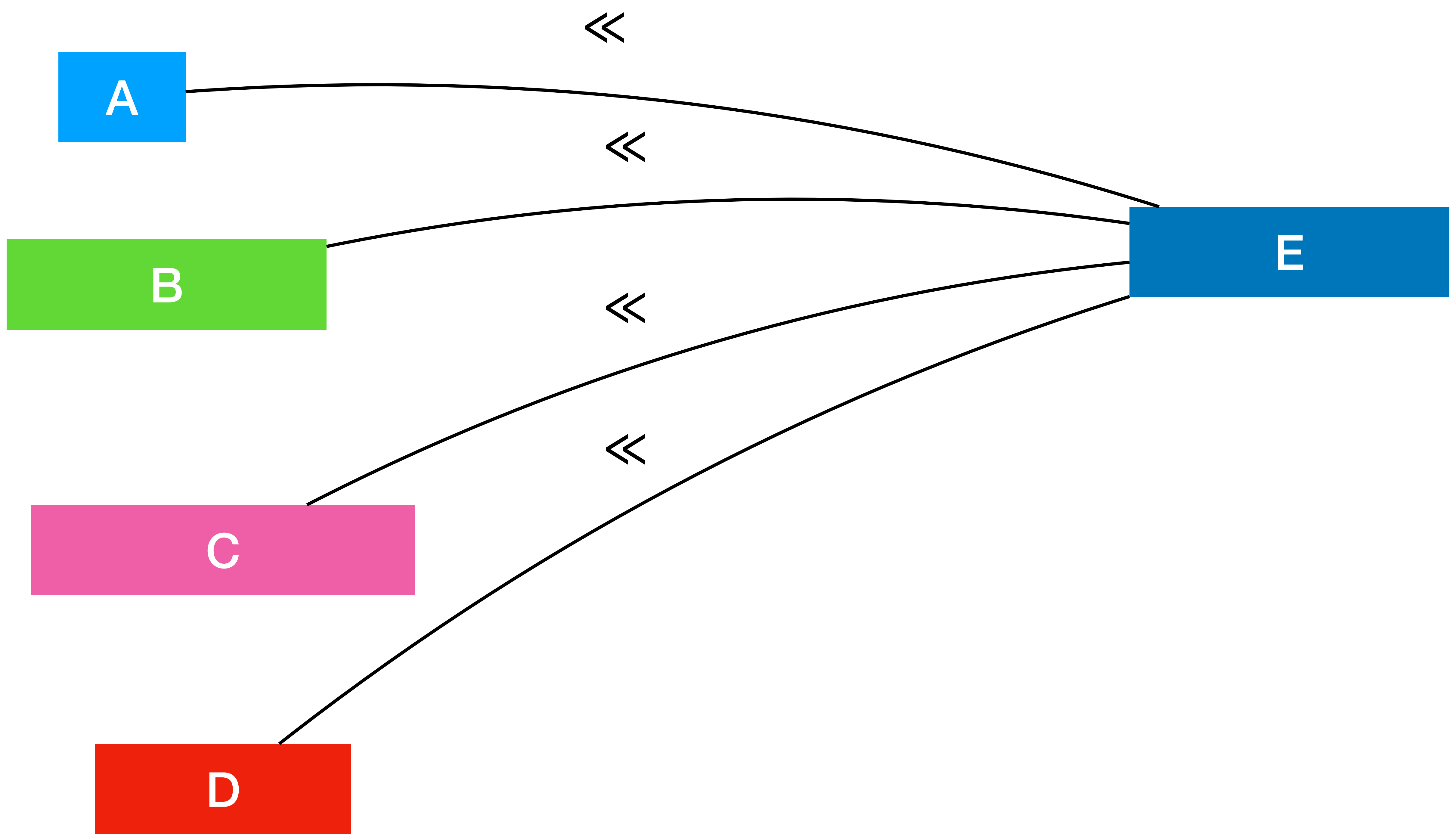
Notation and Definitions

- Let $\Omega \subseteq T$ be a subset of a set T of non-overlapping activities:
  - $est_\Omega = \min \{est_j \mid j \in \Omega\}$ = earliest starting time of Ω
  - $lct_\Omega = \max \{lct_j \mid j \in \Omega\}$ = latest completion time of Ω
  - $d_\Omega = \sum_{j \in \Omega} d_j$ = total duration of Ω

Earliest Completion Time? Why is it important?

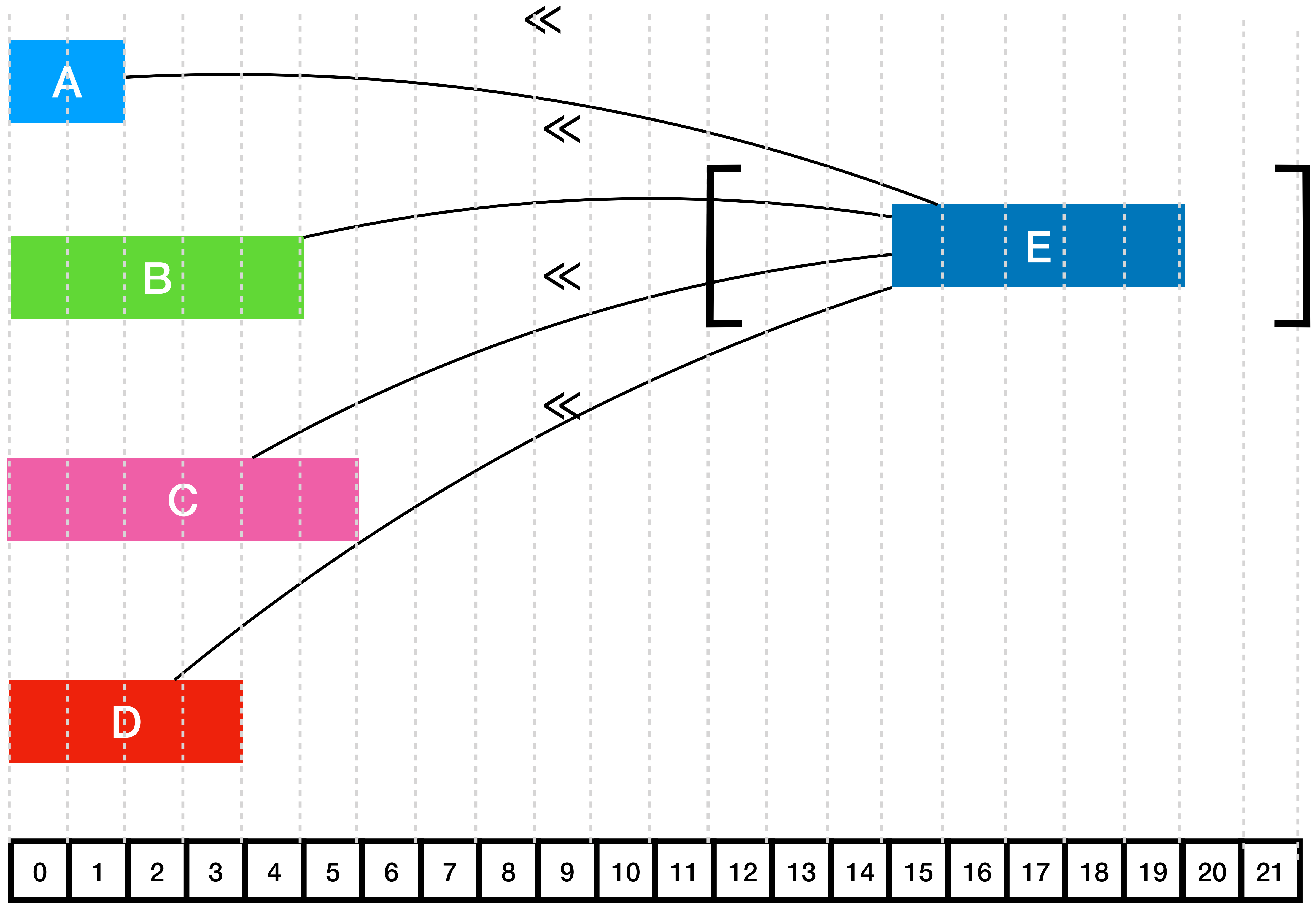
- Assume that we know that A, B, C, D must precede E

- Then E cannot start before the *earliest completion time* of the four activities



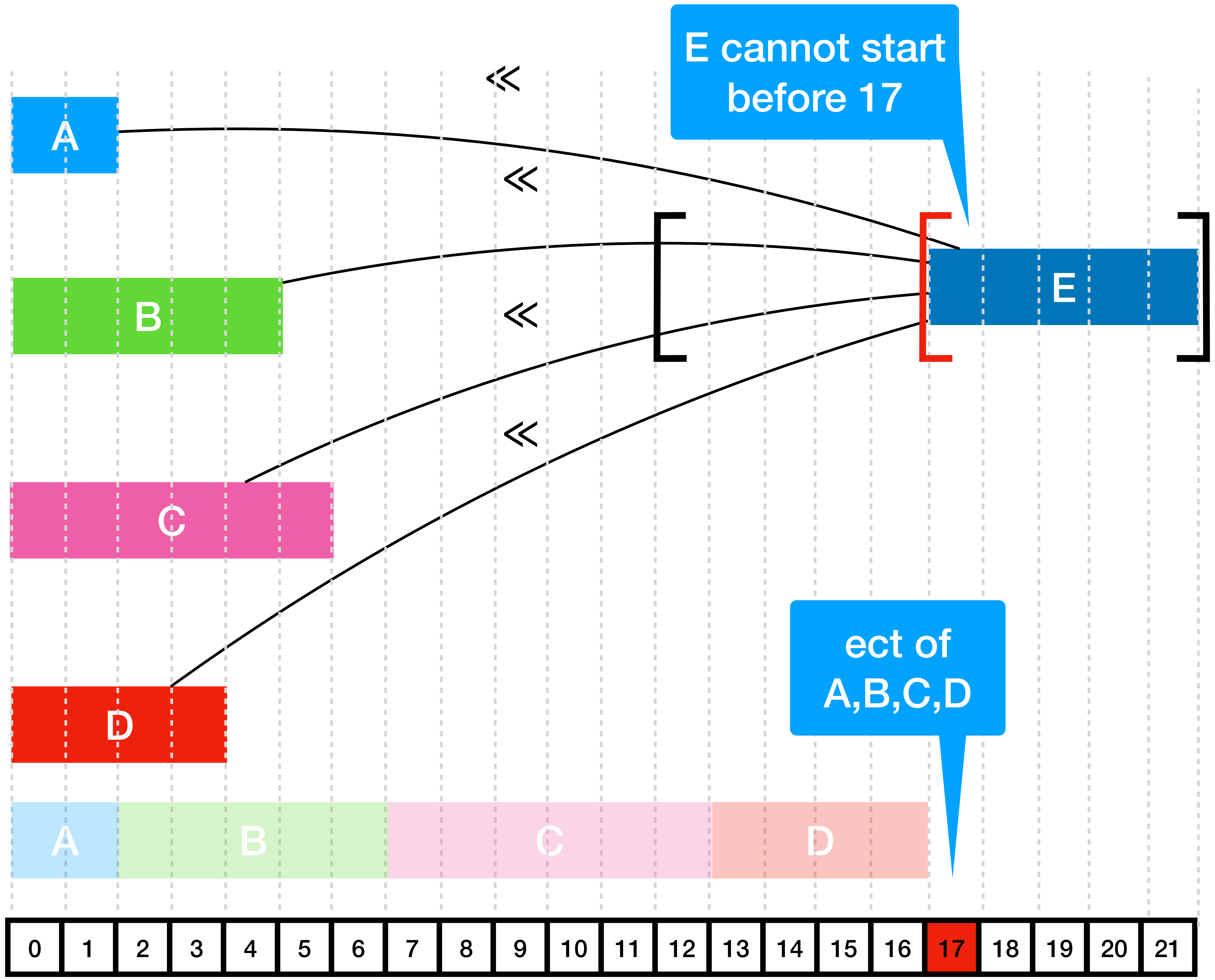
Earliest Completion Time? Why is it important?

- Assume that we know that A, B, C, D must precede E
- Then E cannot start before the *earliest completion time* of the four activities



Earliest Completion Time? Why is it important?

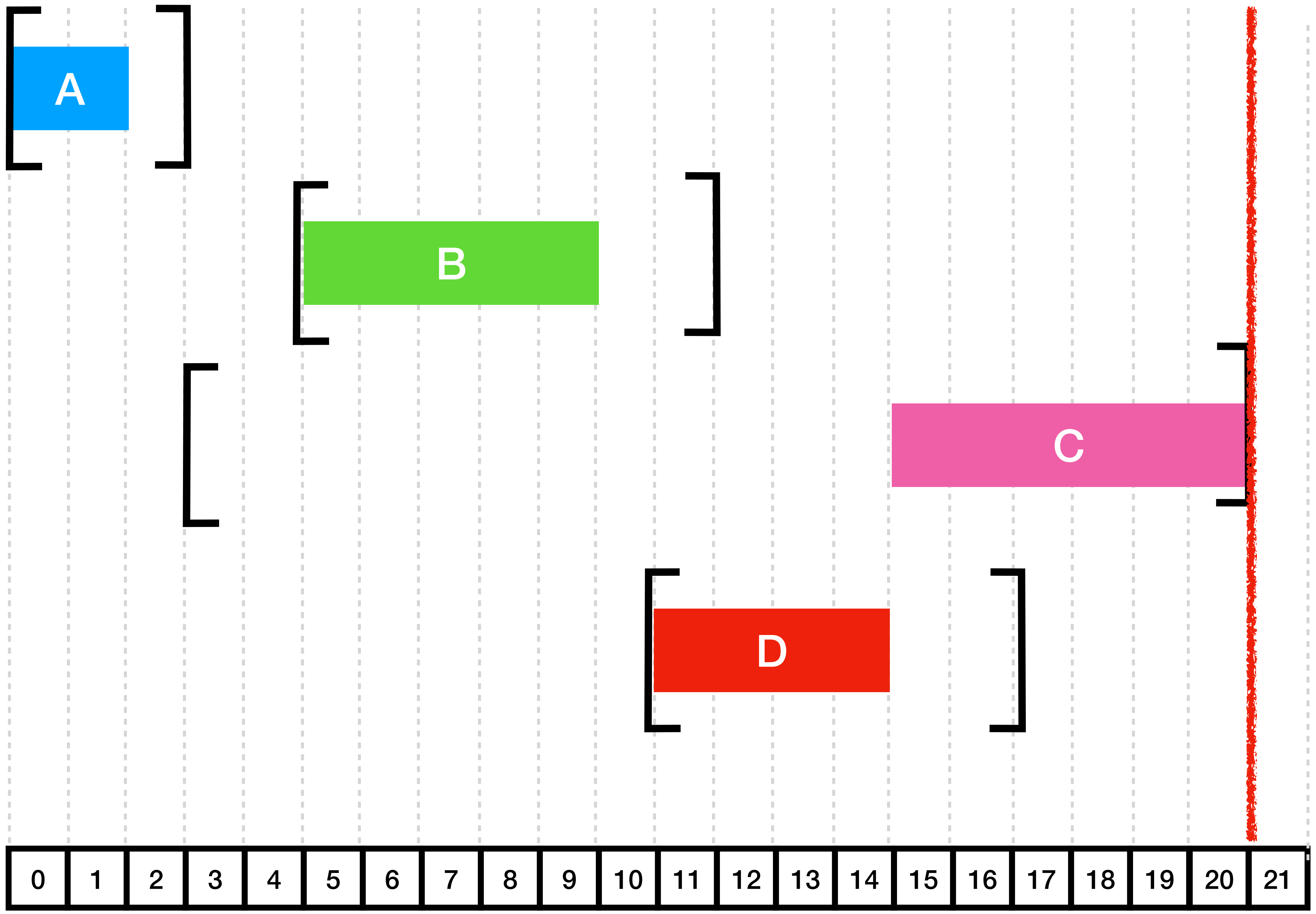
- Assume that we know that A, B, C, D must precede E
- Then E cannot start before the *earliest completion time* of the four activities



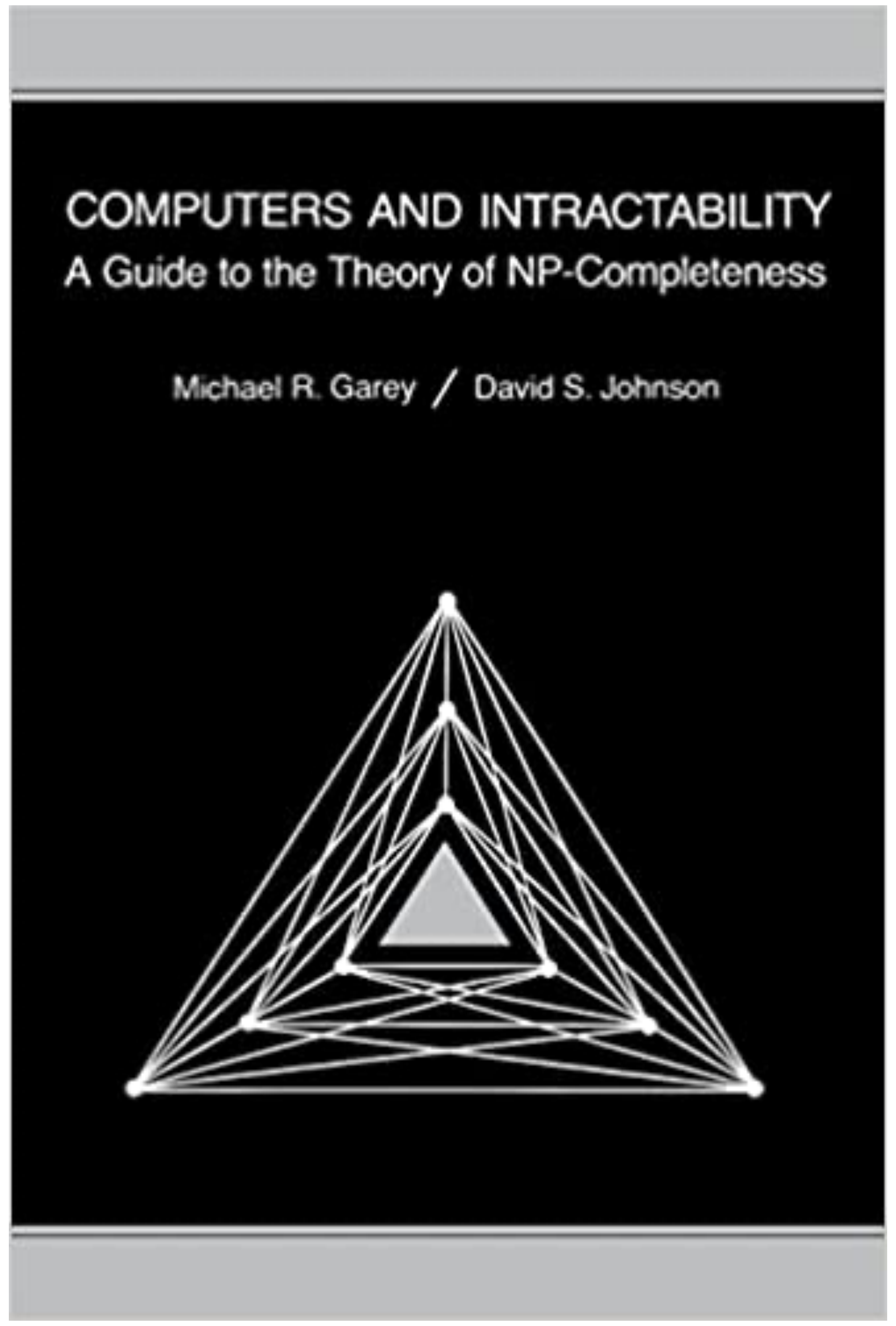
Earliest Completion Time

- Things get complicated when activities have time windows (domains)
- ect({A,B,C,D}) = 21

We cannot do better than 21



This problem is NP-hard 😢
See Garey and Johnson, problem SS1

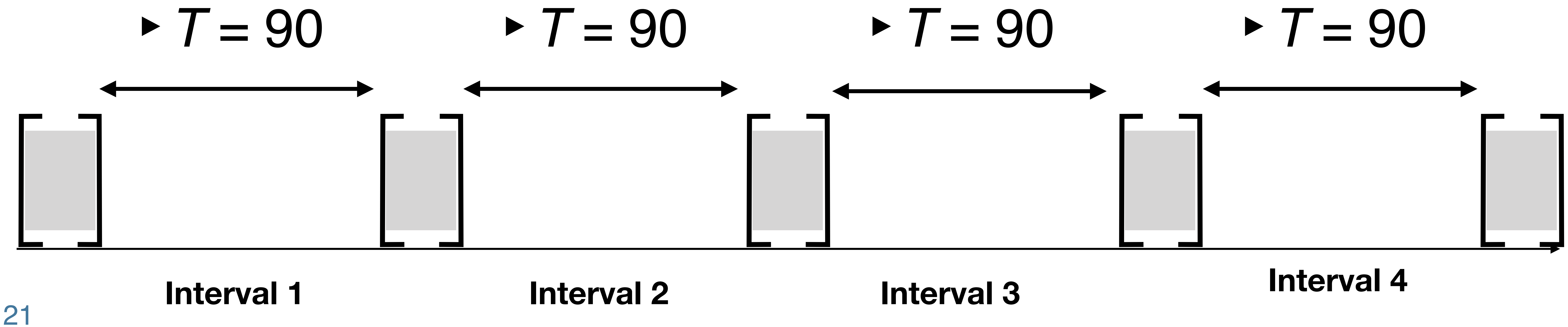


Sequencing with Time Windows is NP-Complete

- Reduction from the 3-Partition problem (known to be NP-complete) to our problem of interest
- 3-Partition (https://en.wikipedia.org/wiki/3-partition_problem):
  - The input is a multiset S of $n = 3m$ positive integers with sum $m\,T$.
  - The output is whether or not there exists a partition of S into m triplets S_1, S_2, …, S_m, each with sum T. (The S_1, S_2 , …, S_m must thus be disjoint and cover S.)

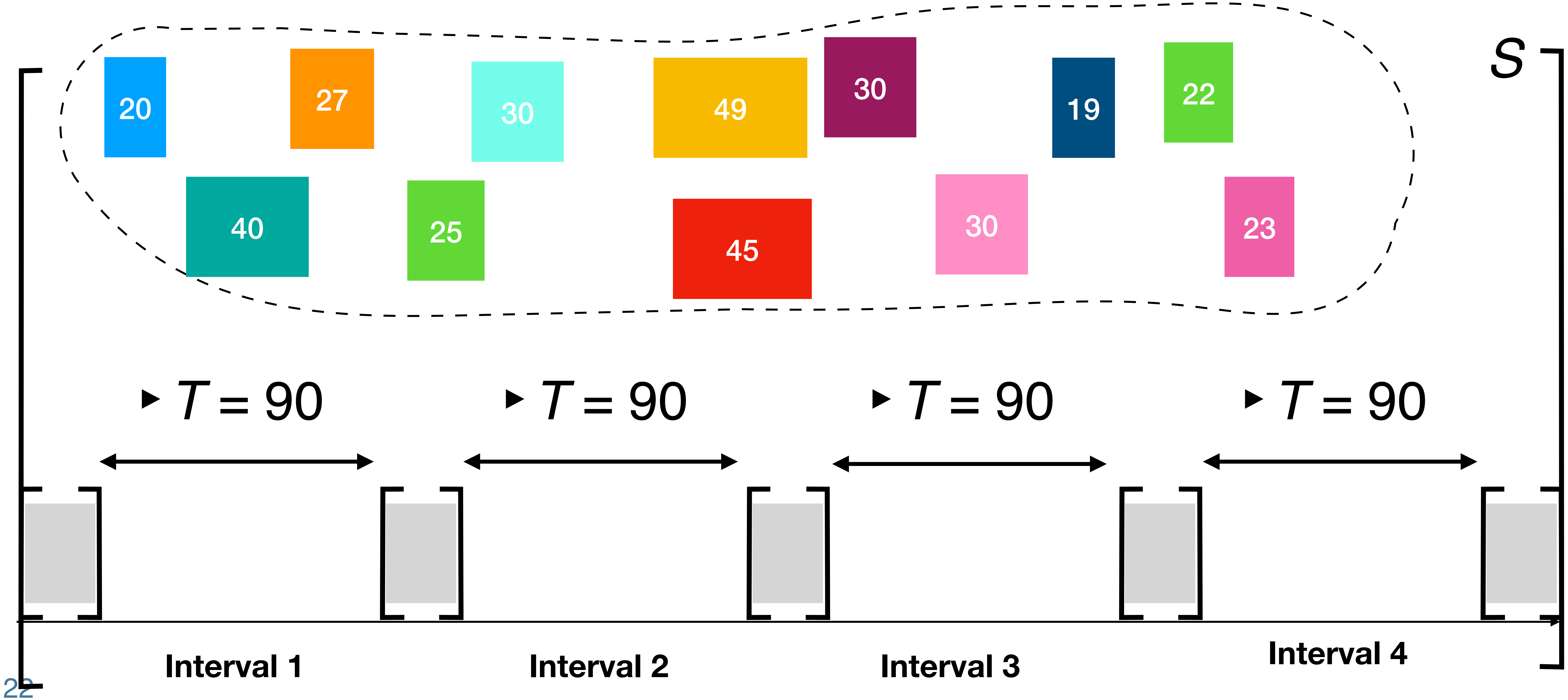
Sequencing with Time Windows is NP-Complete

- Example: The set $S = \{ 20, 23, 25, 30, 49, 45, 27, 30, 30, 40, 22, 19 \}$ can be partitioned into the four triplets $\{ 20, 25, 45 \}$, $\{ 23, 27, 40 \}$, $\{ 49, 22, 19 \}$, $\{ 30, 30, 30 \}$, each of which sums to $T = 90$.



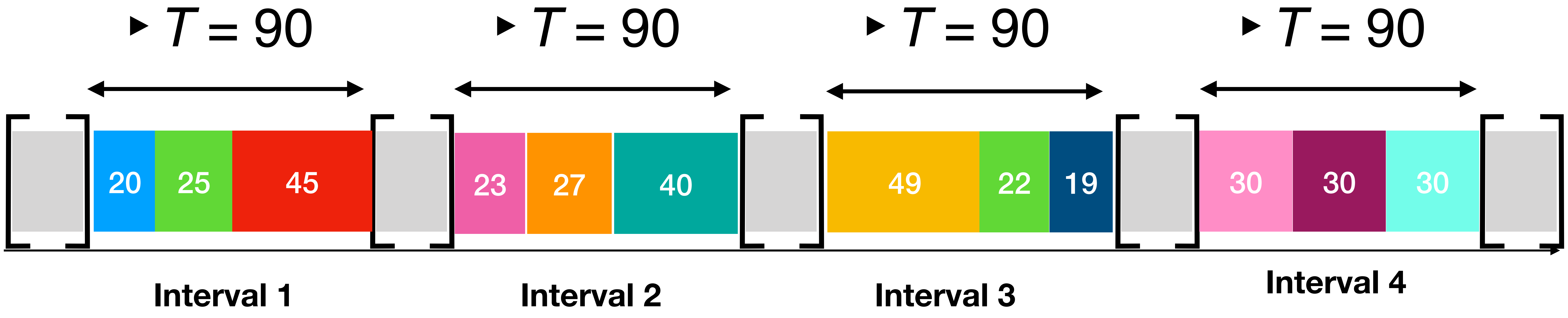
Sequencing with Time Windows is NP-Complete

- Example: The set $S = \{ 20, 23, 25, 30, 49, 45, 27, 30, 30, 40, 22, 19 \}$ can be partitioned into the four triplets $\{ 20, 25, 45 \}$, $\{ 23, 27, 40 \}$, $\{ 49, 22, 19 \}$, $\{ 30, 30, 30 \}$, each of which sums to $T = 90$.



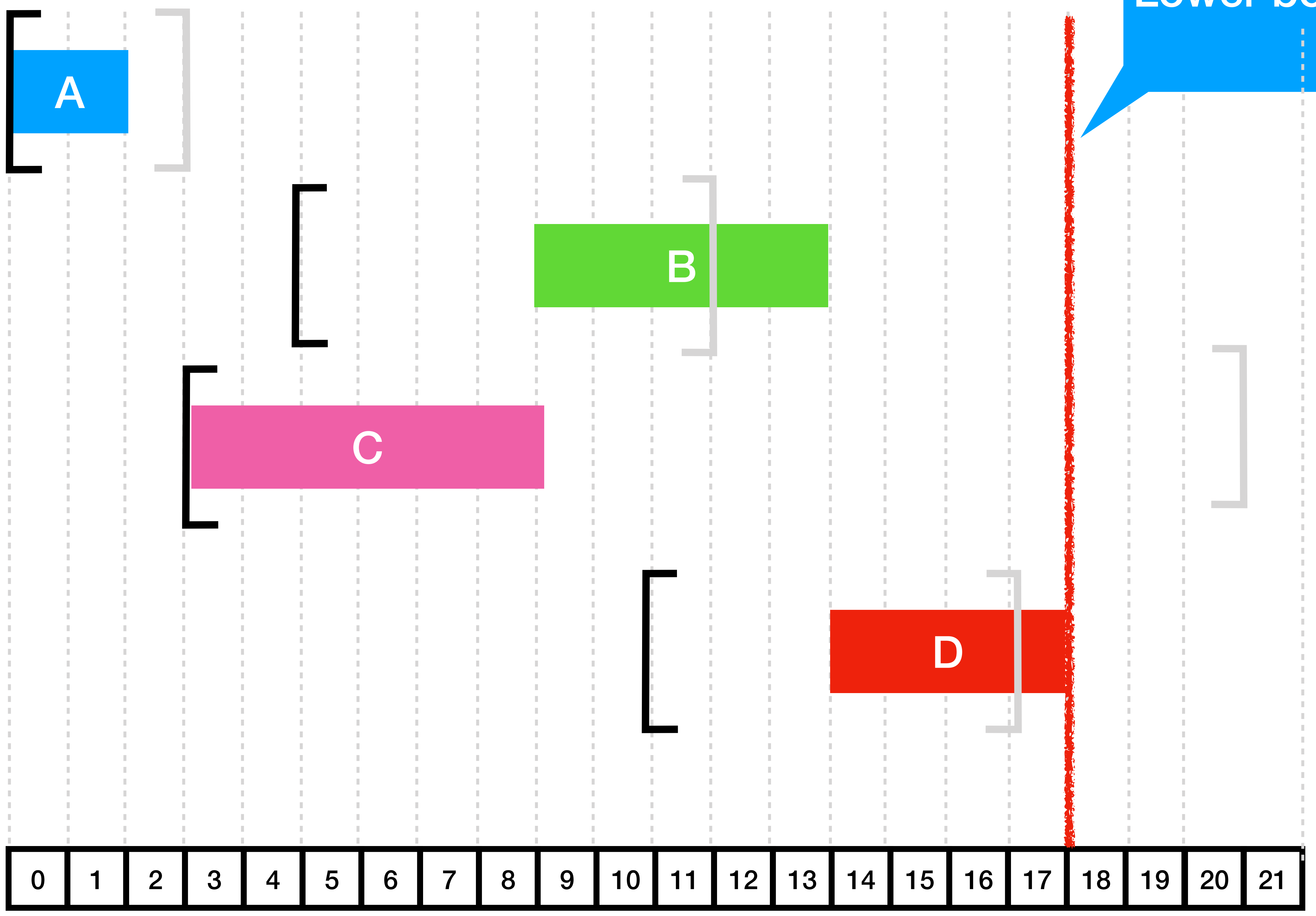
Sequencing with Time Windows is NP-Complete

- Example: The set $S = \{ 20, 23, 25, 30, 49, 45, 27, 30, 30, 40, 22, 19 \}$ can be partitioned into the four triplets $\{ 20, 25, 45 \}$, $\{ 23, 27, 40 \}$, $\{ 49, 22, 19 \}$, $\{ 30, 30, 30 \}$, each of which sums to $T = 90$.



Lower Bound on the Earliest Completion Time

- Relaxation of the time windows:
  keep the earliest start time but relax the latest completion time



Lower bound on the earliest completion time = 18 (<= 21)

This relaxation is easy to solve 🥳.
You can for instance sort the activities by earliest start time and schedule them as soon as possible.

Lower Bound on the Earliest Completion Time

```
ComputeECTLowerBound(T={1..n}) {
  Test ← sortAZ([1..n],sortKey = est) // O(n log n)
  ect = -inf
  for (i ← Test) {
    ect ← max(esti+di , ect+di)
  }
  return ect
}
```

Lower Bound on the Earliest Completion Time

- This lower bound can be formally defined as
  $\mathrm{ect}_{\Omega}^{\mathrm{LB}} = \max\{\mathrm{est}_{\Omega'} + d_{\Omega'} \mid \Omega' \subseteq \Omega\}$
- But, as just seen, we do not need to enumerate all the subsets,
  since we can compute it in O(n log n) time for n activities.
- In the following, by abuse of notation and since we will always use the lower bound, we drop “LB”:
  $\mathrm{ect}_{\Omega}^{\mathrm{LB}}$ is denoted by ect_{Ω}

Latest Starting Time (same idea)

- We also introduce an upper bound on the latest starting time (mirroring problem), which is $lst_{\Omega} = \min \{lct_{\Omega'} - d_{\Omega'} \mid \Omega' \subseteq \Omega\}$

Conventions for empty set

- By convention:
  - $est_{\emptyset} = ect_{\emptyset} = -\infty$
  - $lst_{\emptyset} = lct_{\emptyset} = +\infty$
  - $d_{\emptyset} = 0$

Earliest Completion Times of nested sets of activities

Earliest completion times of nested sets

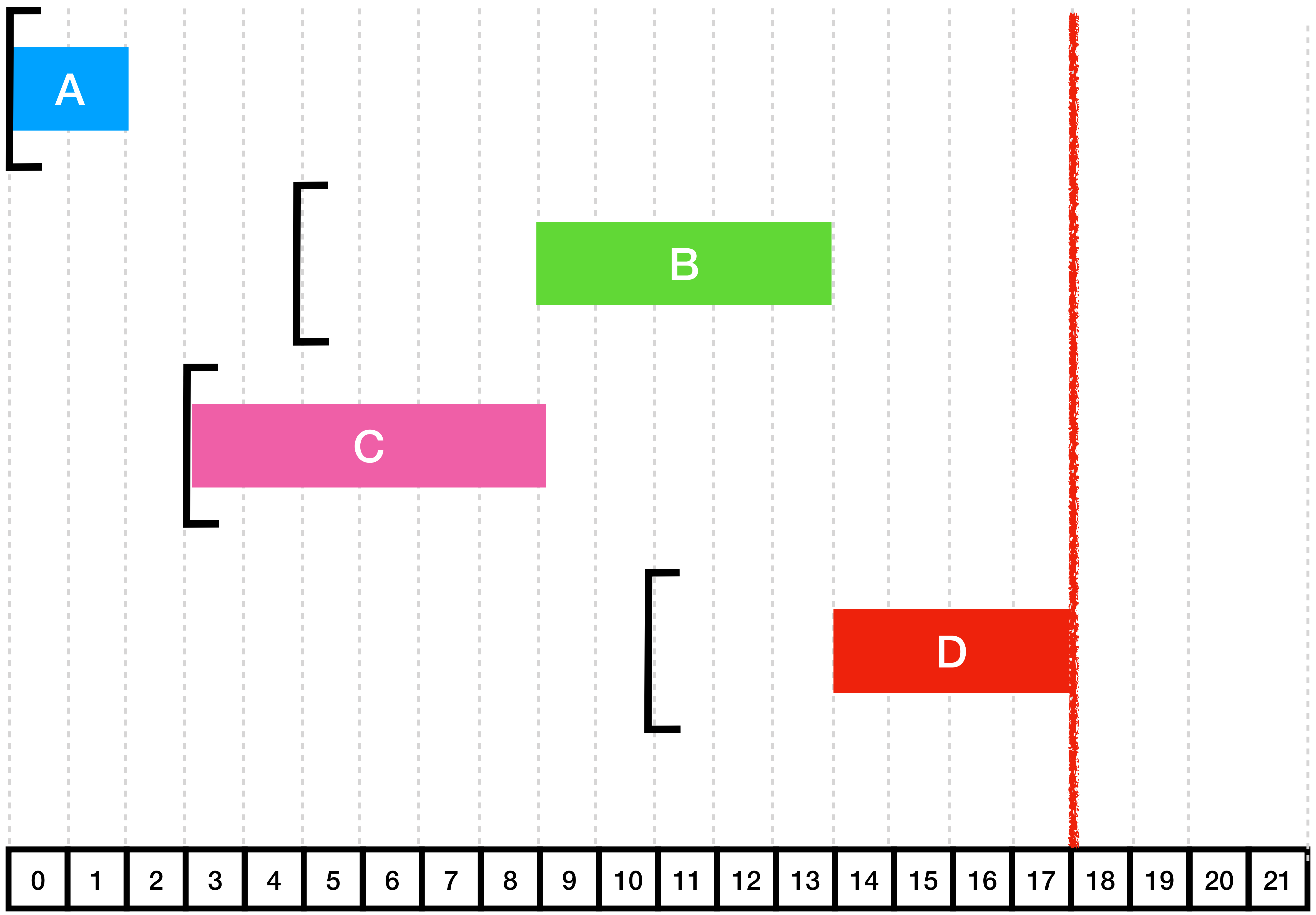
- Given n activities from the set T, given nested sets of activities

$$\Omega_1 = \{T_1\} \subset \Omega_2 \subset \Omega_3 \subset \cdots \subset \Omega_n = T \text{ with } \Omega_i = \Omega_{i-1} \cup \{T_i\}$$

- Can we compute all $\mathrm{ect}(\Omega_1), \mathrm{ect}(\Omega_2), \mathrm{ect}(\Omega_3), \ldots, \mathrm{ect}(\Omega_n)$ efficiently?

- Naïve approach: compute each independently: $O(n^2 \log n)$ time
- More efficient approach: use a data structure called a Θ-tree

Small example of nested sets



A	B	C	D

T

Ω_1 $\quad \mathrm{ect}(\{A\})?$

Ω_2 $\quad \mathrm{ect}(\{A, B\})?$

Ω_3 $\quad \mathrm{ect}(\{A, B, C\})?$

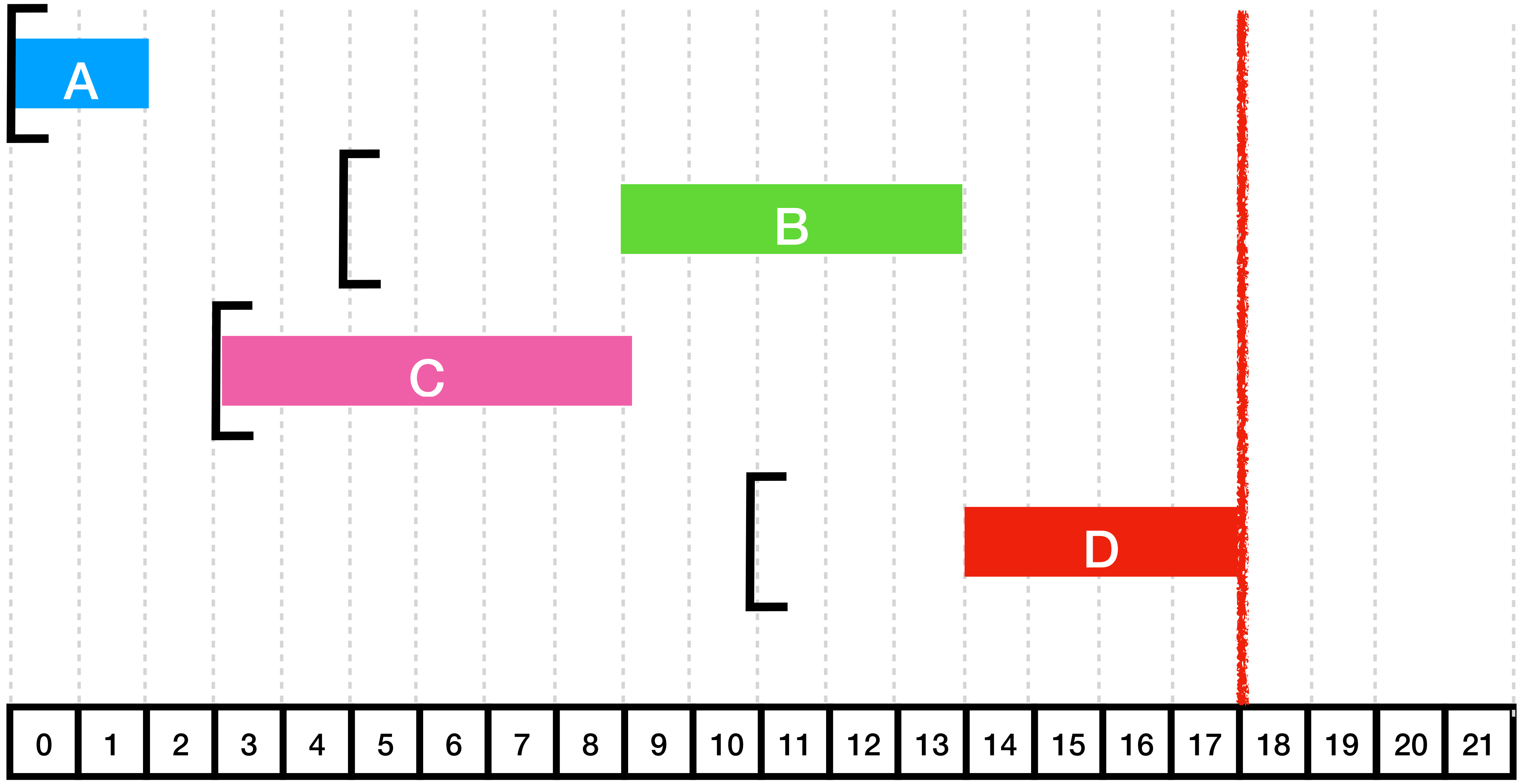
Ω_4 $\quad \mathrm{ect}(\{A, B, C, D\})?$

Θ-tree intuition

- The goal is to mimic the behavior of the seen algorithm:

```
ComputeECTLowerBound(T={1..n}) {
   Test ← sortAZ([1..n],sortKey = est)
   ect = -inf
   for (i ← Test) {
     ect ← max(esti+di , ect+di)
   }
   return ect
}
```

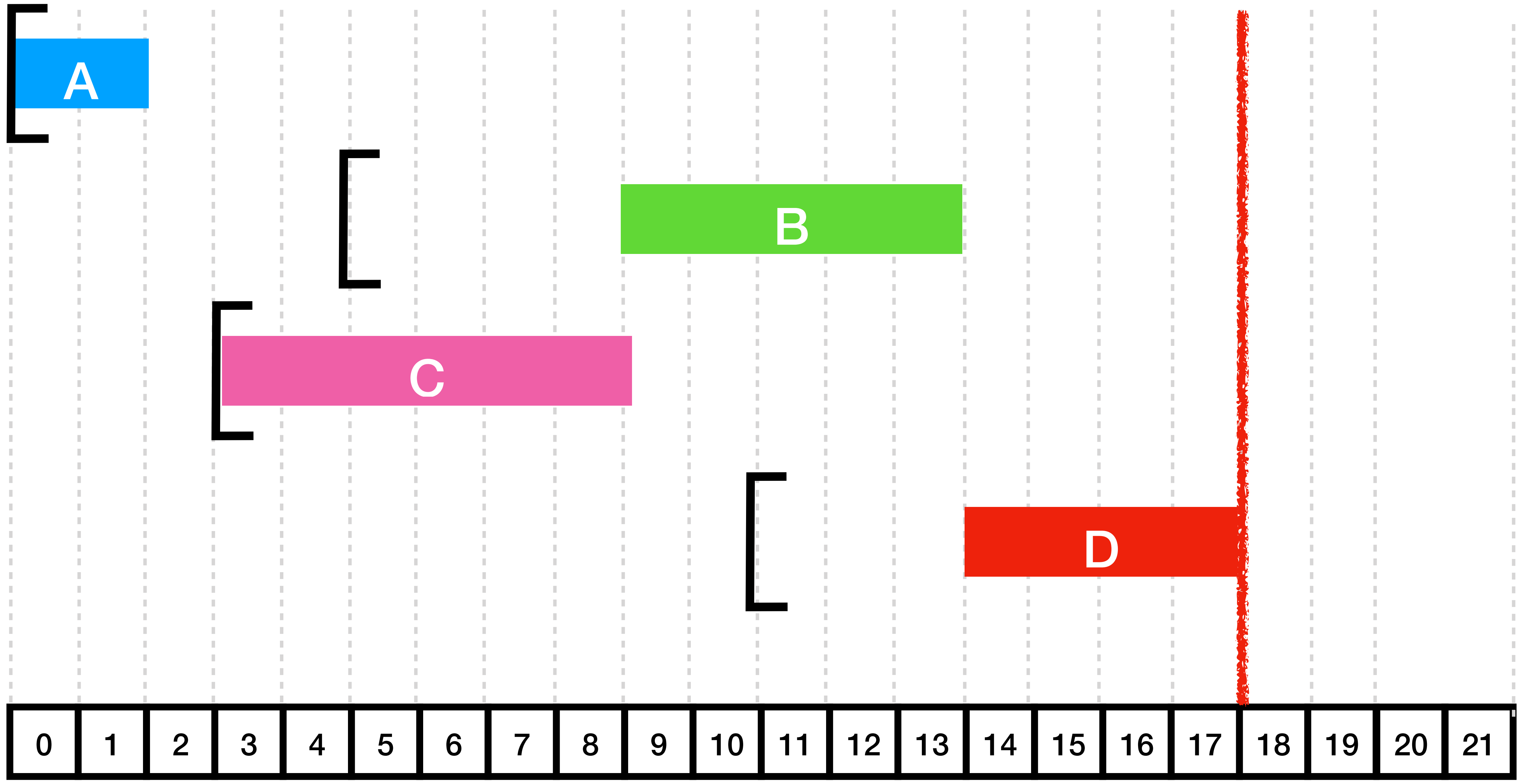
[A,C,B,D]

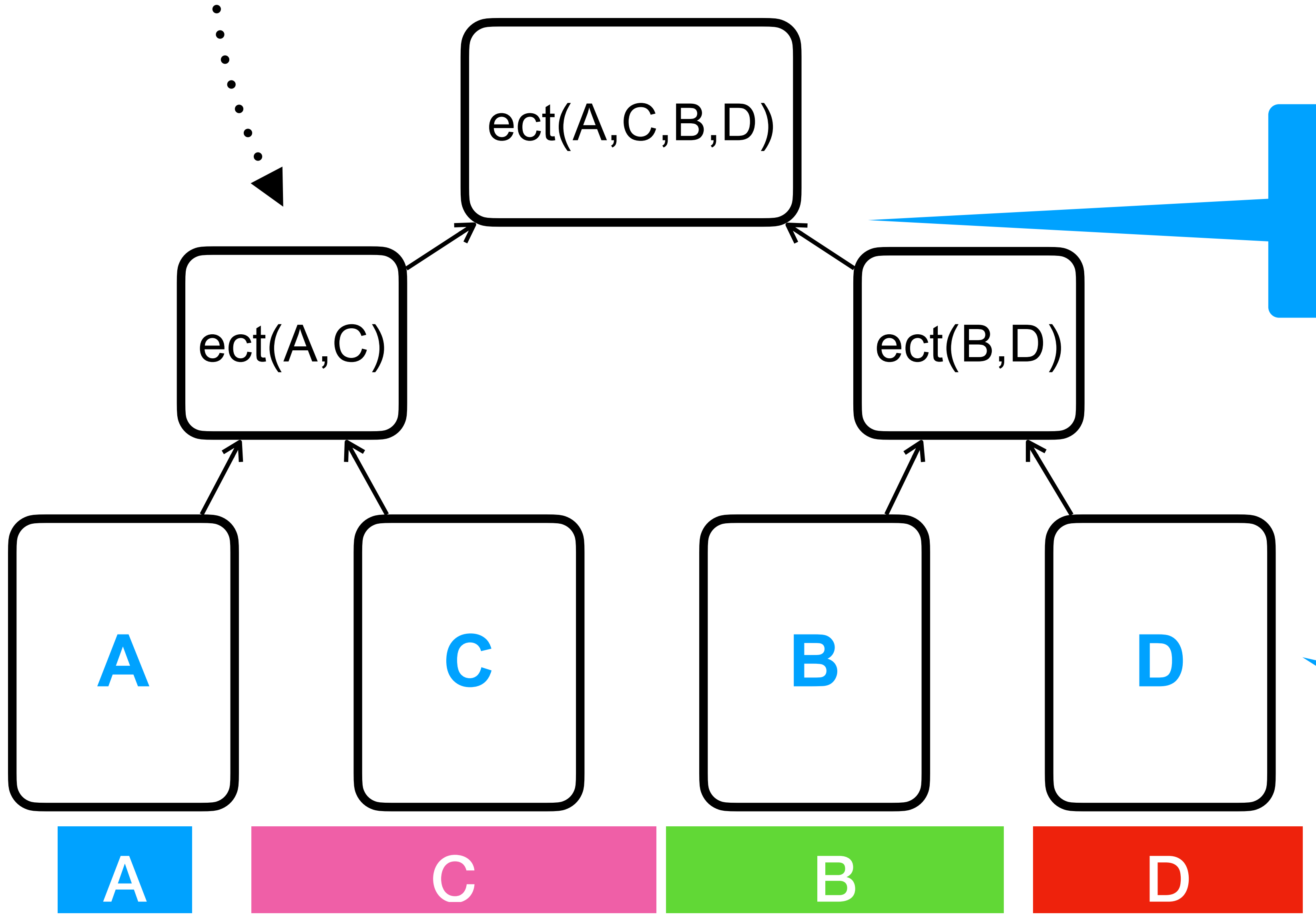


Θ-tree intuition

- The goal is to mimic the behavior of the seen algorithm:

```
ComputeECTLowerBound(T={1..n}) {
  T_est ← sortAZ([1..n],sortKey = est)
  ect = -inf
  for (i ← T_est) {
    ect ← max(est_i+d_i , ect+d_i)
  }
  return ect
}
```





2: bottom-up ect computation

1: activities are sorted wrt est

Bottom up computation

Update rule for each non-leaf v:

$\Delta_v = \Sigma P_{left(v)} + \Sigma P_{right(v)}$

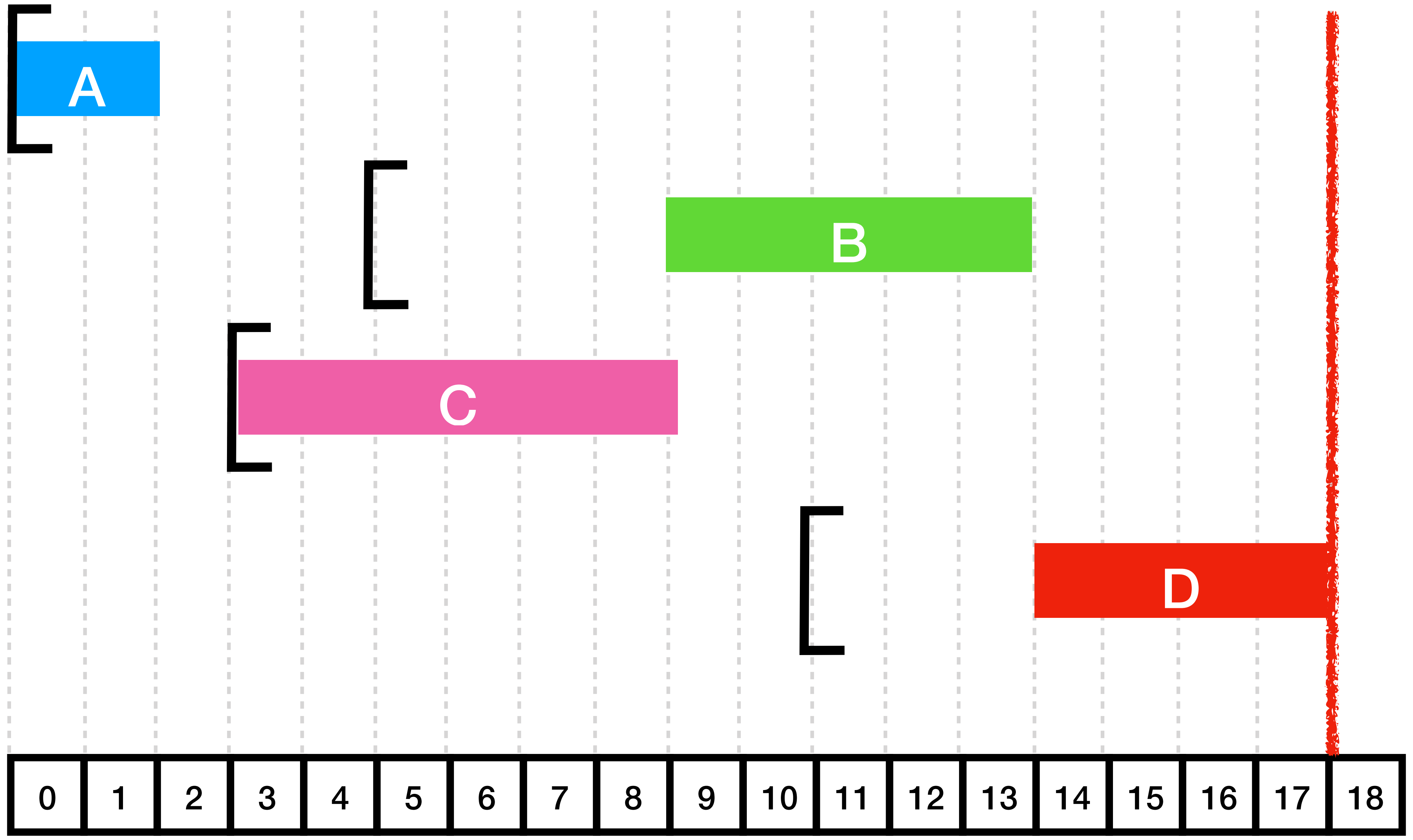
$ect_v = \max(ect_{left(v)} + \Sigma P_{right(v)}, ect_{right(v)})$

$\Delta_{ABCD} = 17$
$ect_{ABCD} = \max(9+9, 15) = 18$

$\Delta_{AC} = 8$
$ect_{AC} = \max(8, 9)$

$\Delta_{BD} = 9$
$ect_{BD} = 15$

A	C	B	D
$est_A = 0$	$est_C = 3$	$est_B = 5$	$est_D = 11$
$d_A = 2$	$d_C = 6$	$d_B = 5$	$d_D = 4$
$\Delta_A = 2$	$\Delta_C = 6$	$\Delta_B = 5$	$\Delta_D = 4$
$ect_A = 2$	$ect_C = 9$	$ect_B = 10$	$ect_D = 15$

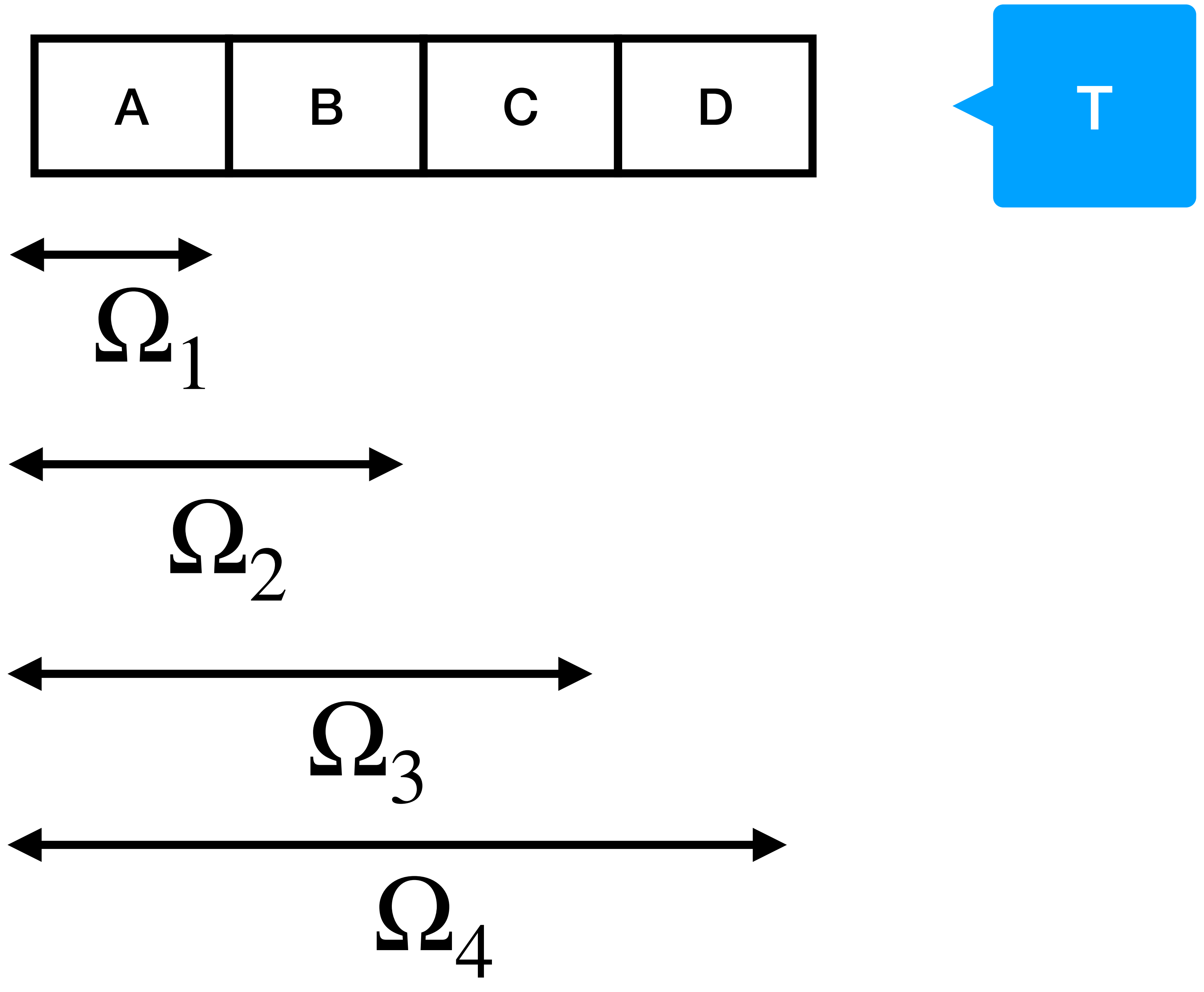


Time complexity?

What do we gain compared to simple algorithm?

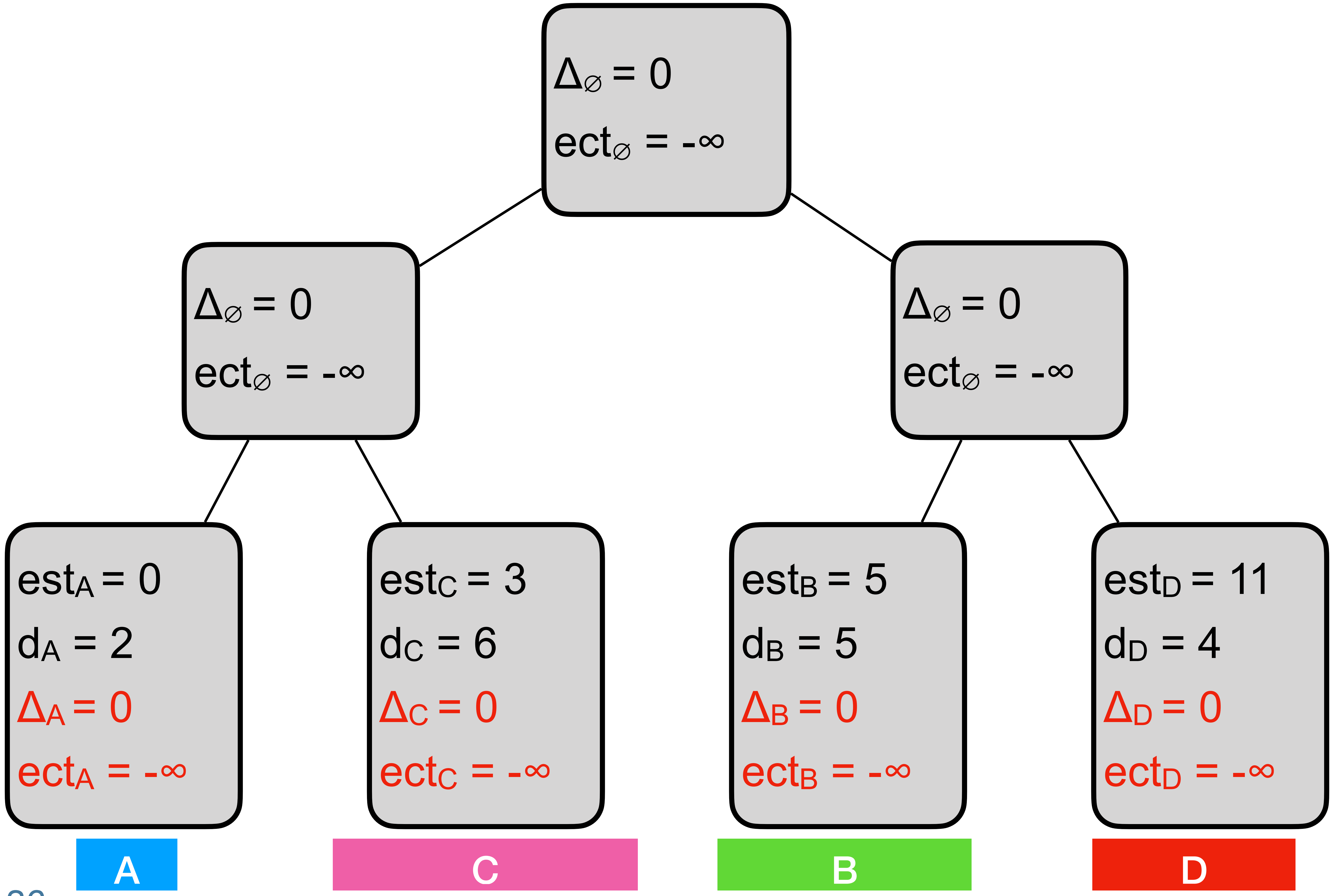
- Not the same problem
- We wanted to compute ect for nested sets
- Θ-tree can deal with it, not the simple algo

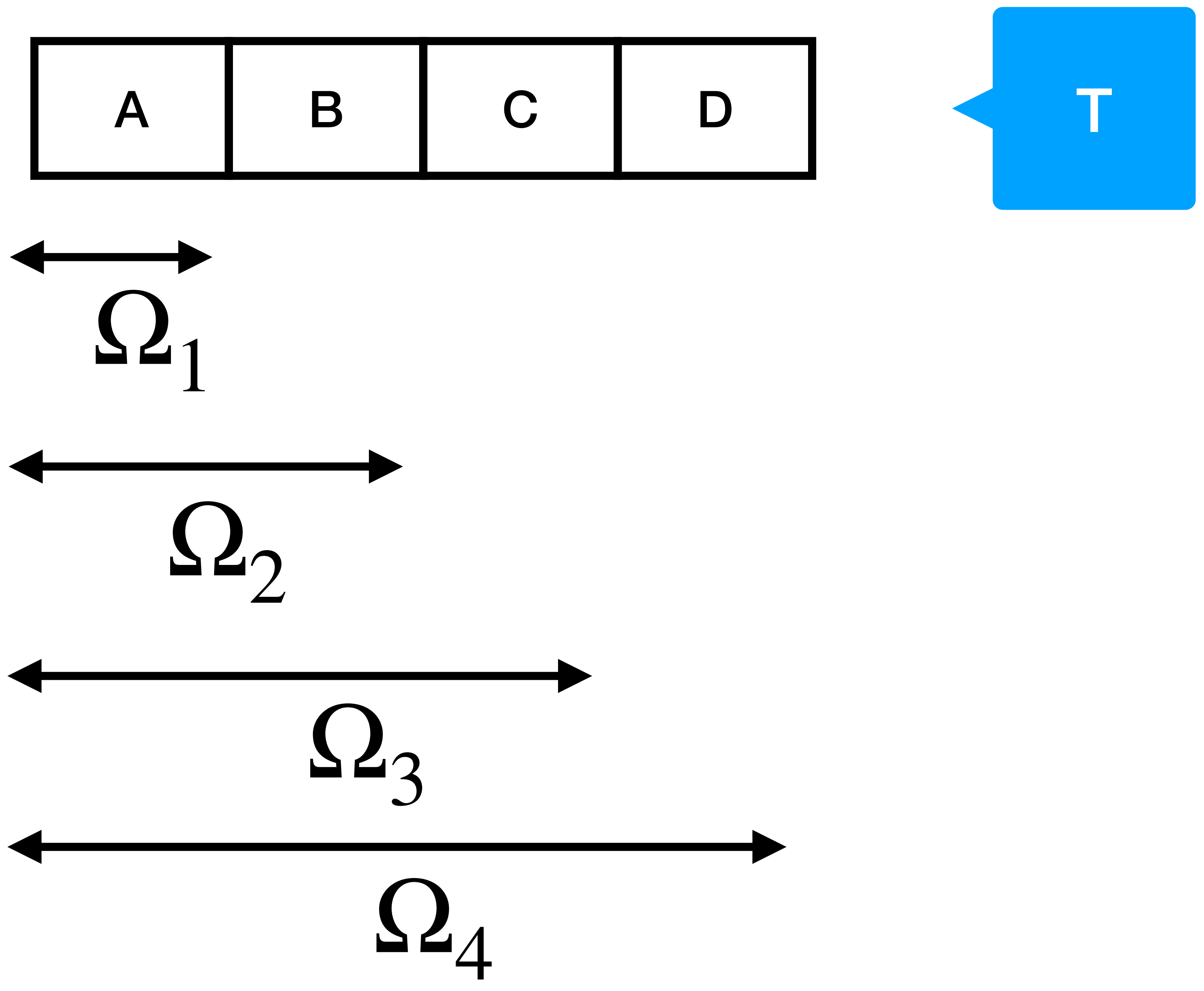
```
ComputeECTLowerBound(T={1..n}) {
  T_est ← sortAZ([1..n],sortKey = est)
  ect = -inf
  for (i ← T_est) {
    ect ← max(est_i+d_i , ect+d_i)
  }
  return ect
}
```



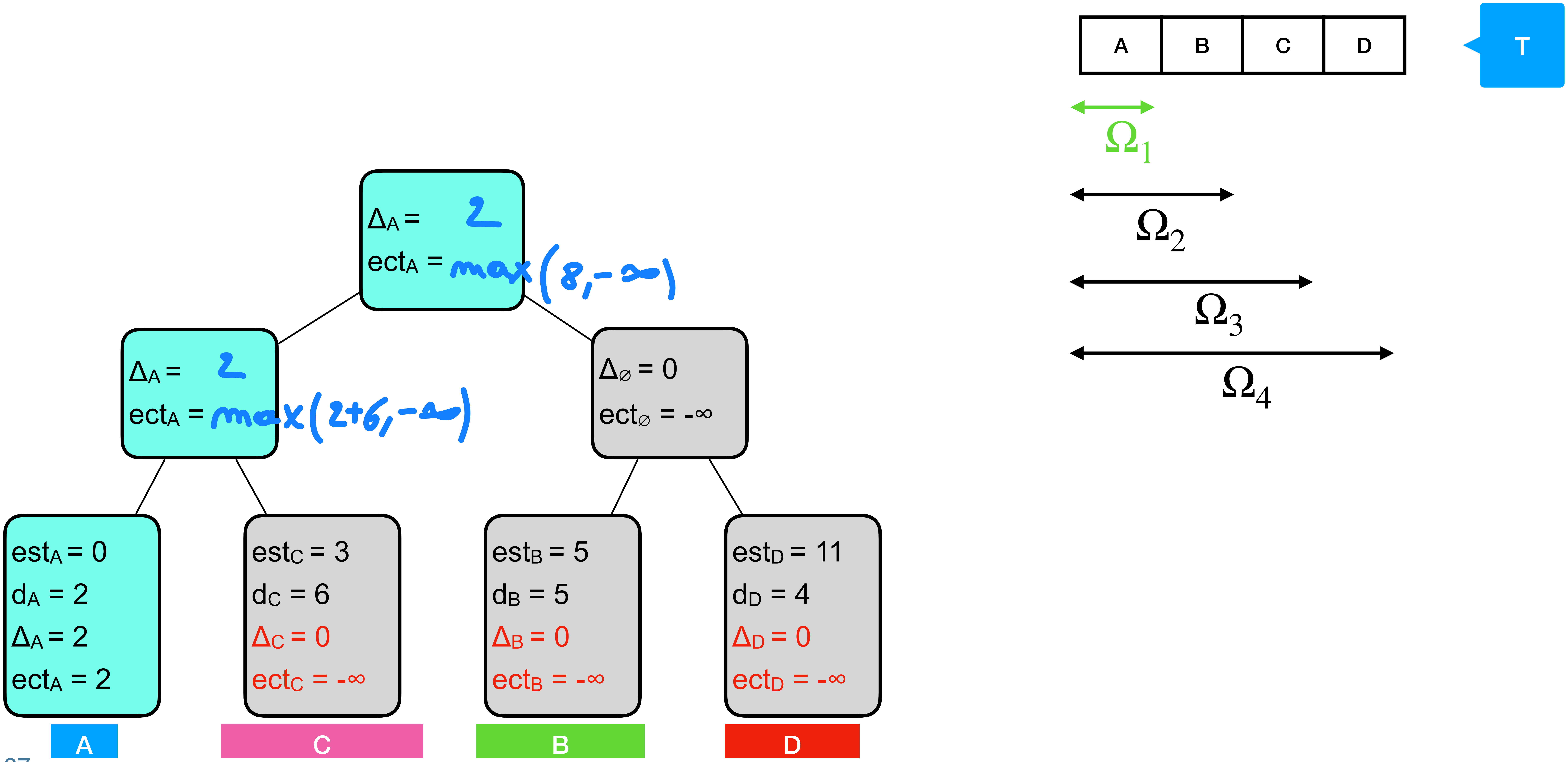
Θ-tree, initialization

- Empty set of activities

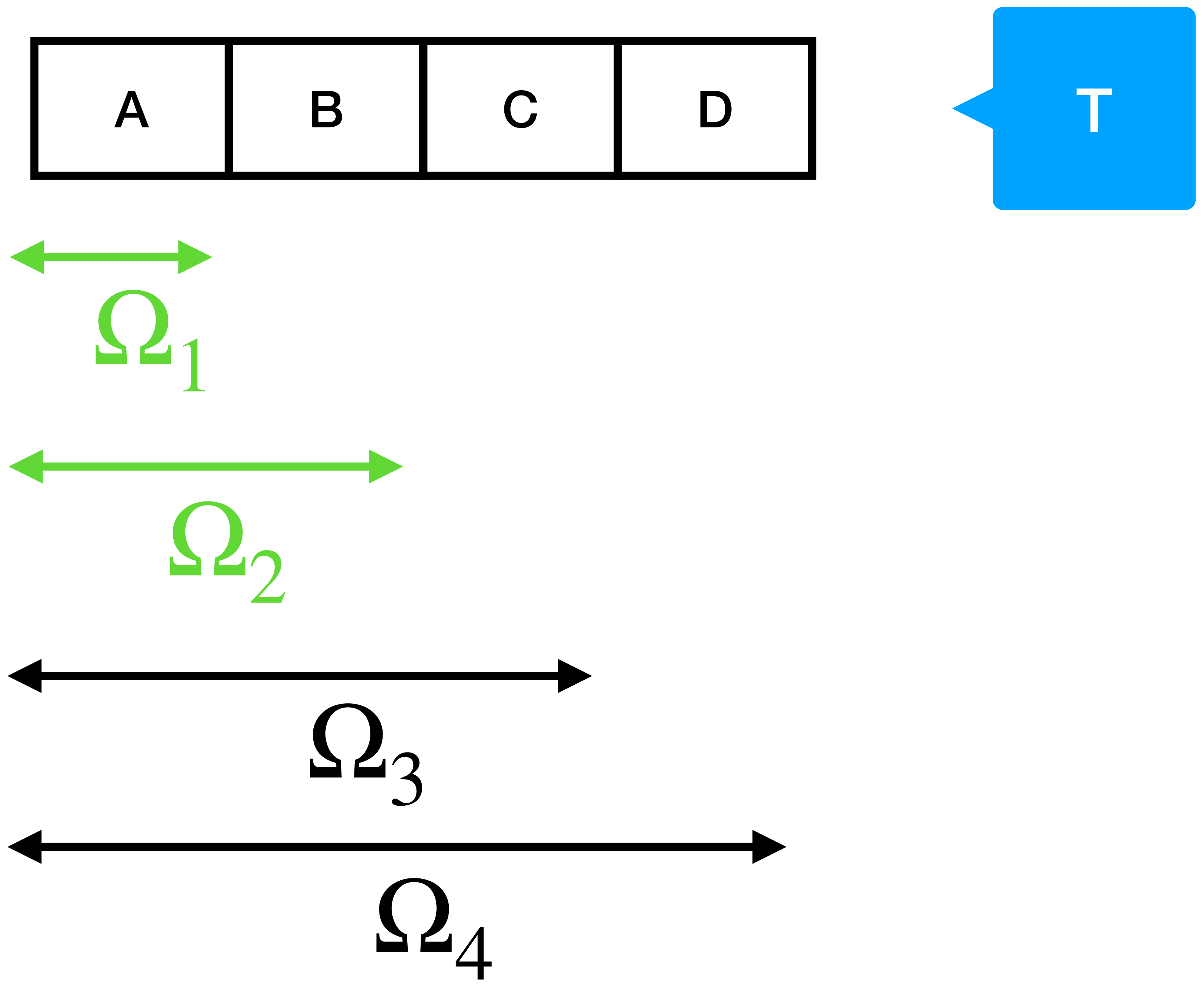


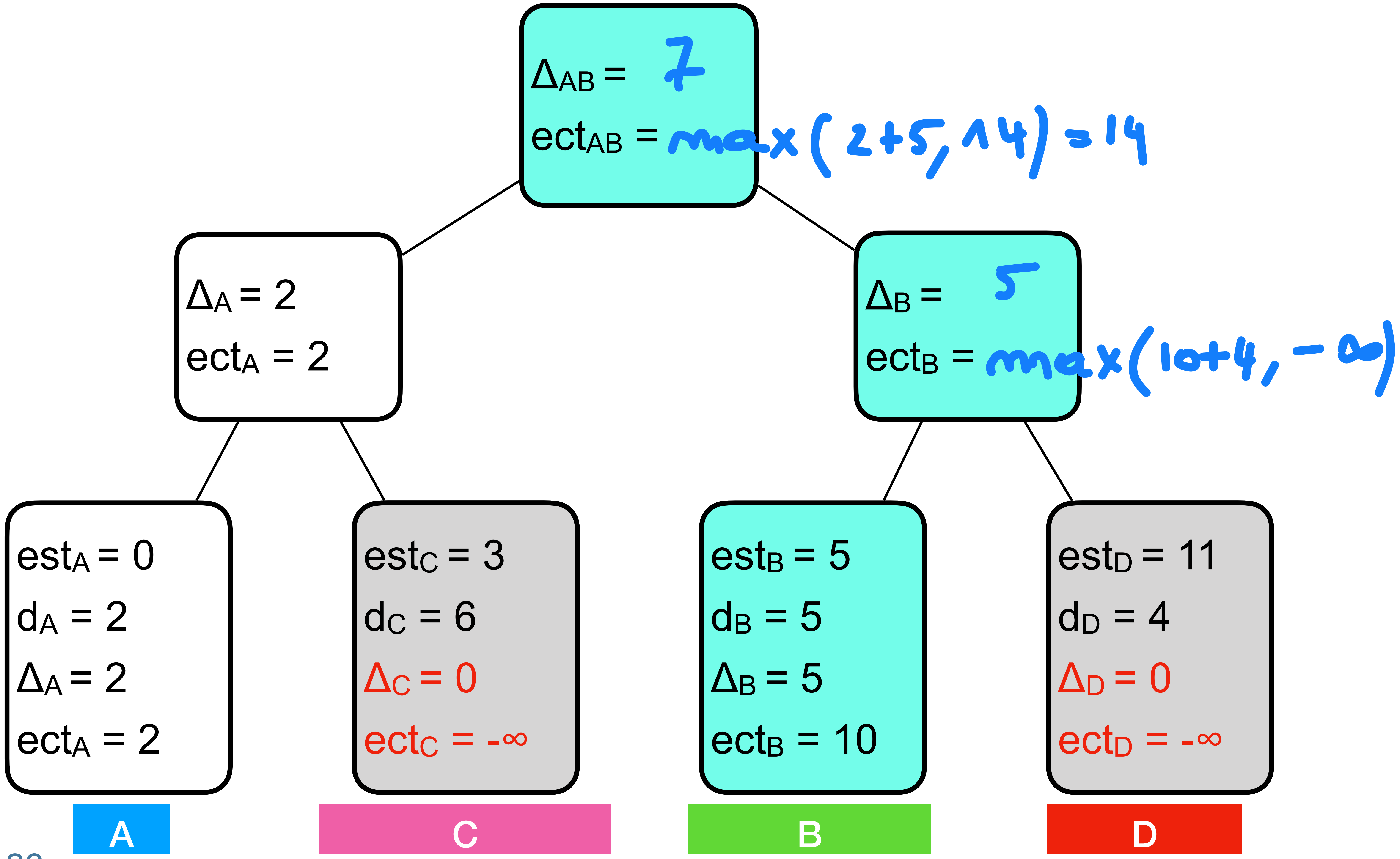


Insertion of A

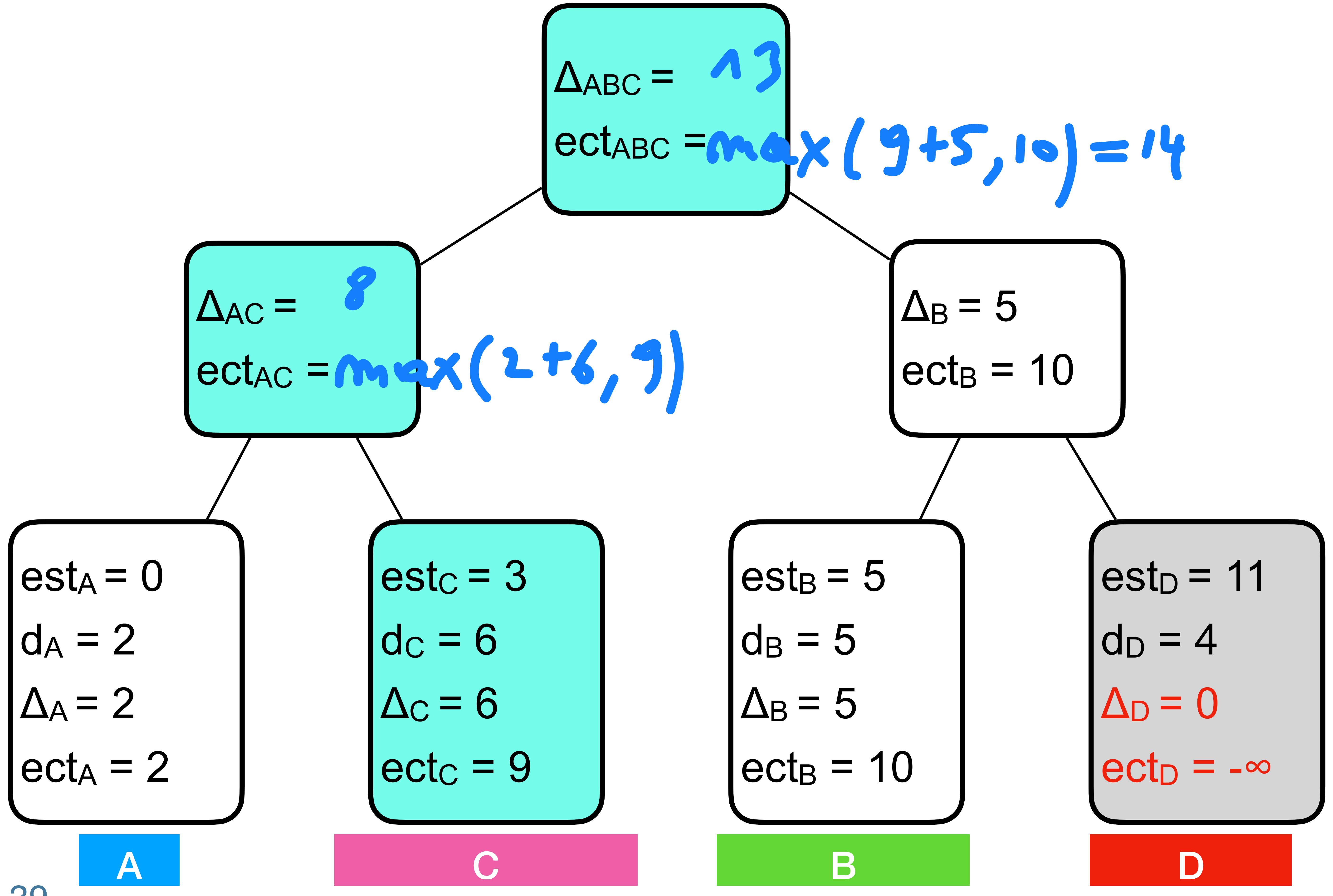


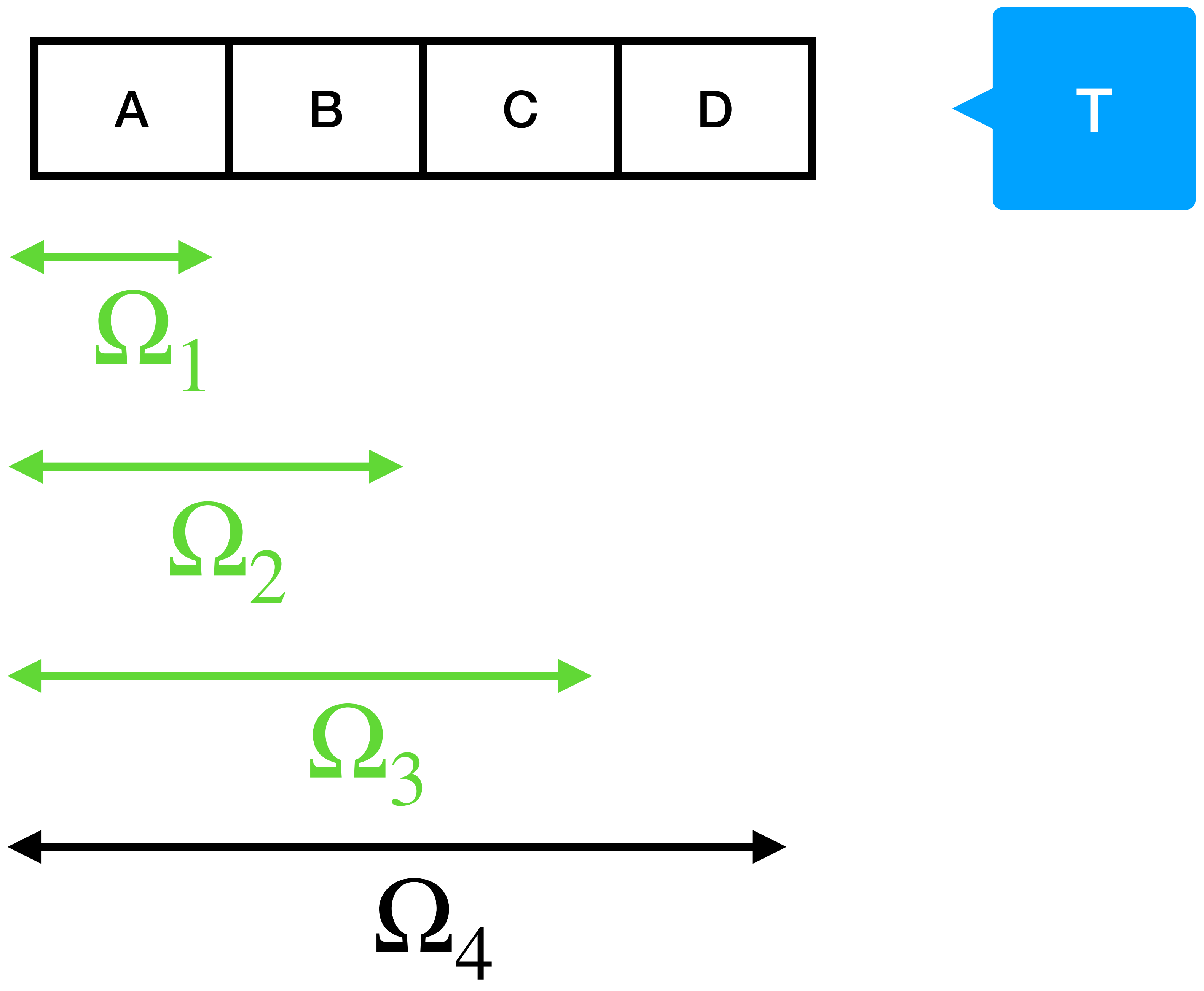
Insertion of B



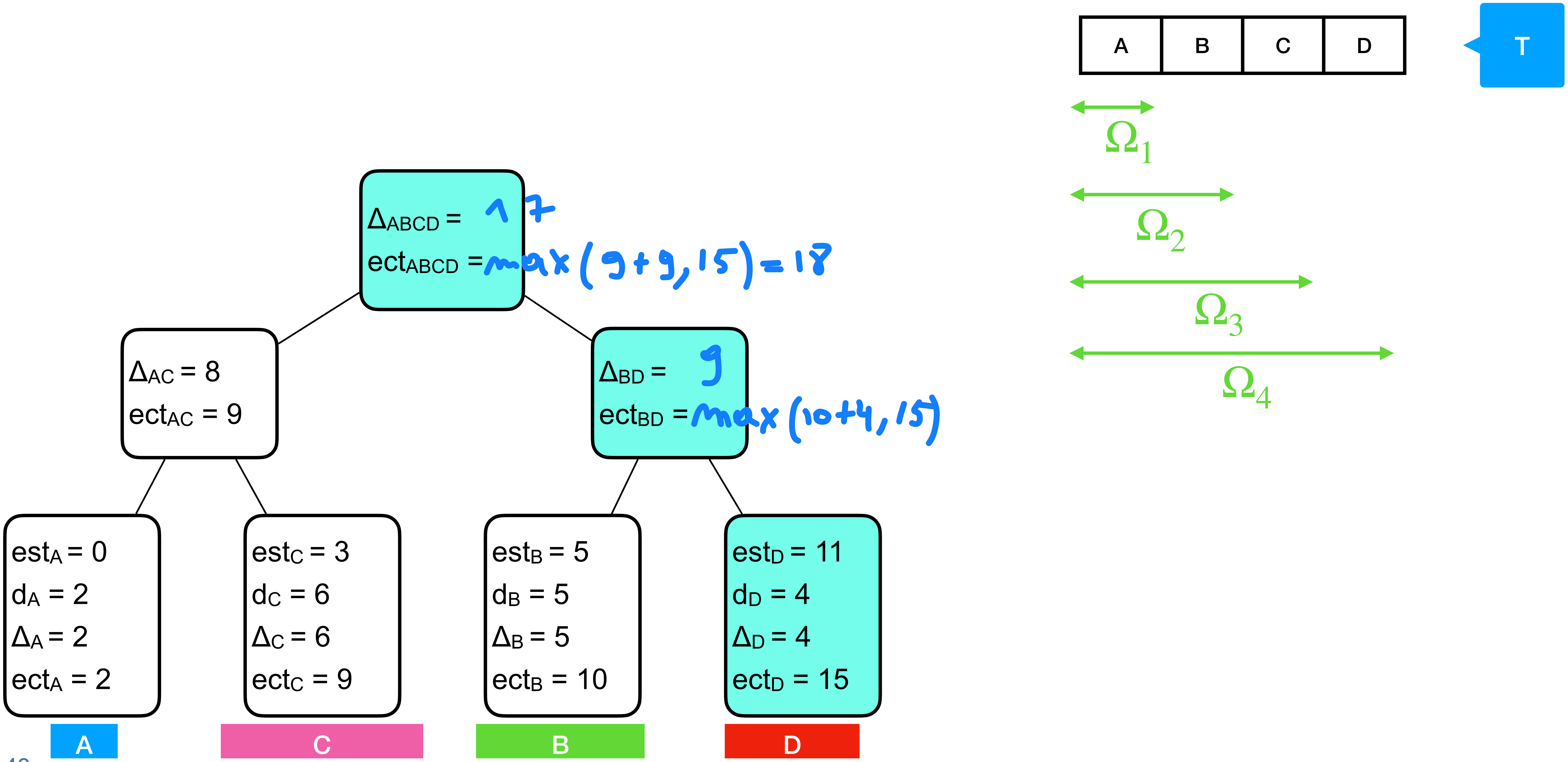


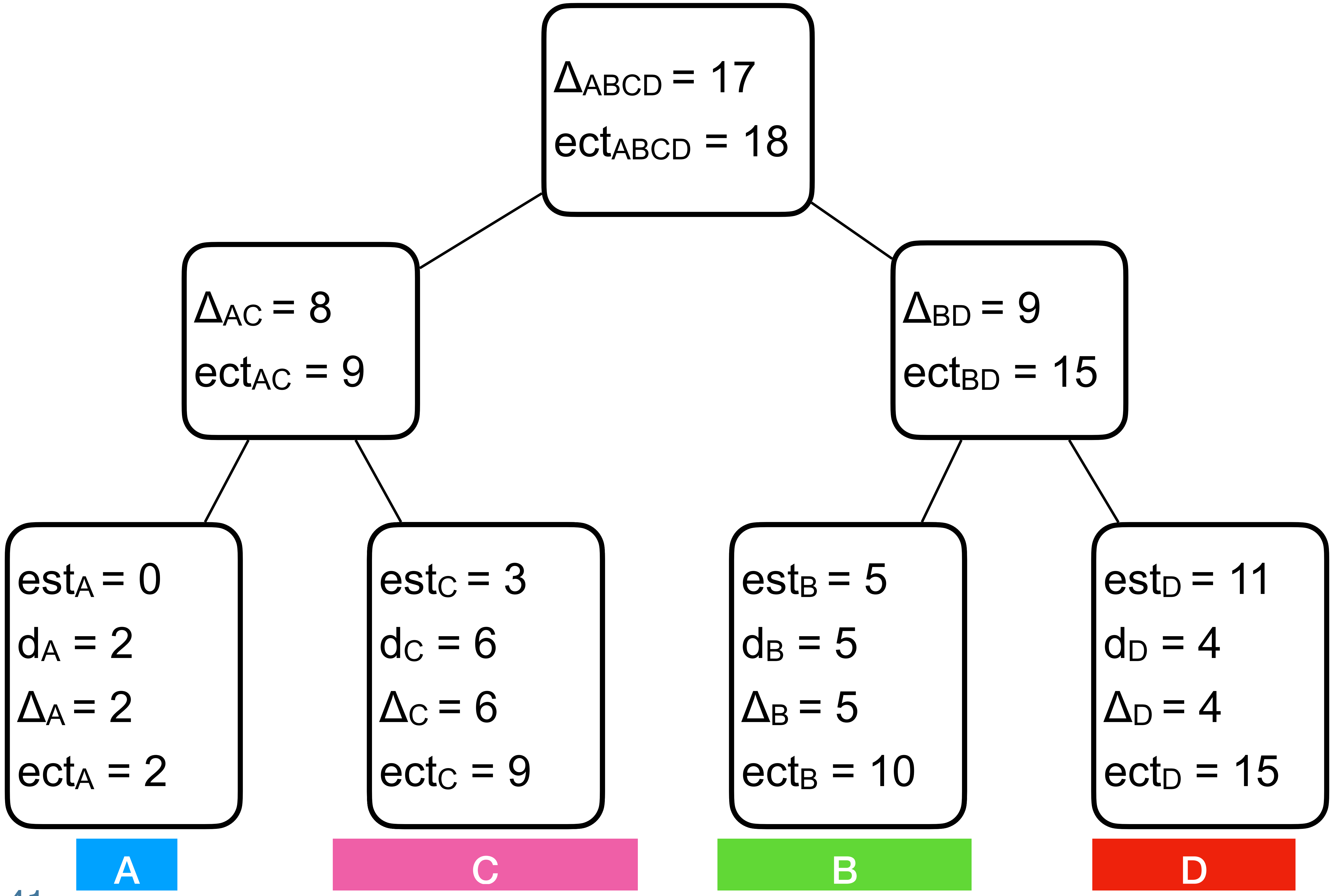
Insertion of C

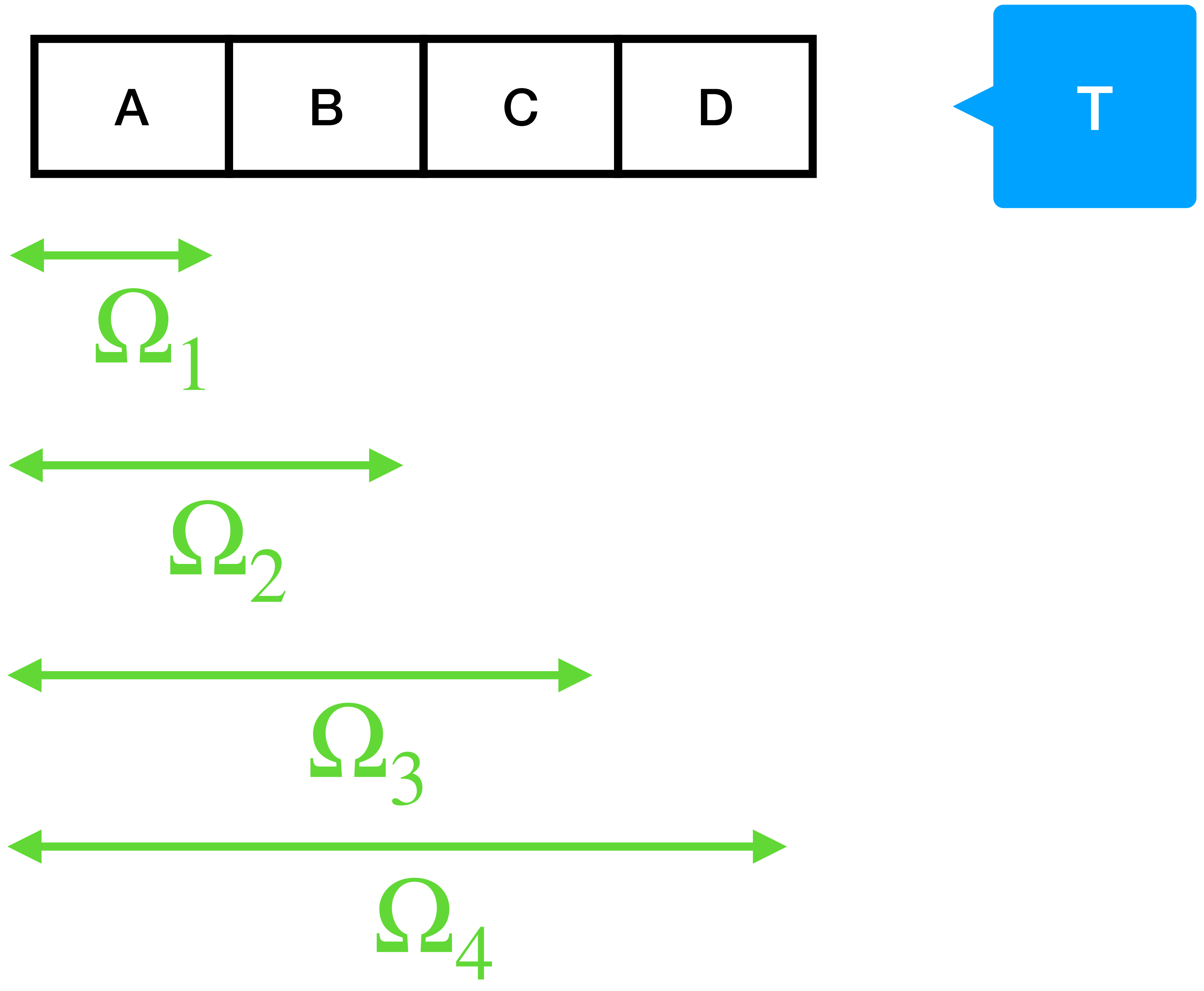




Insertion of D



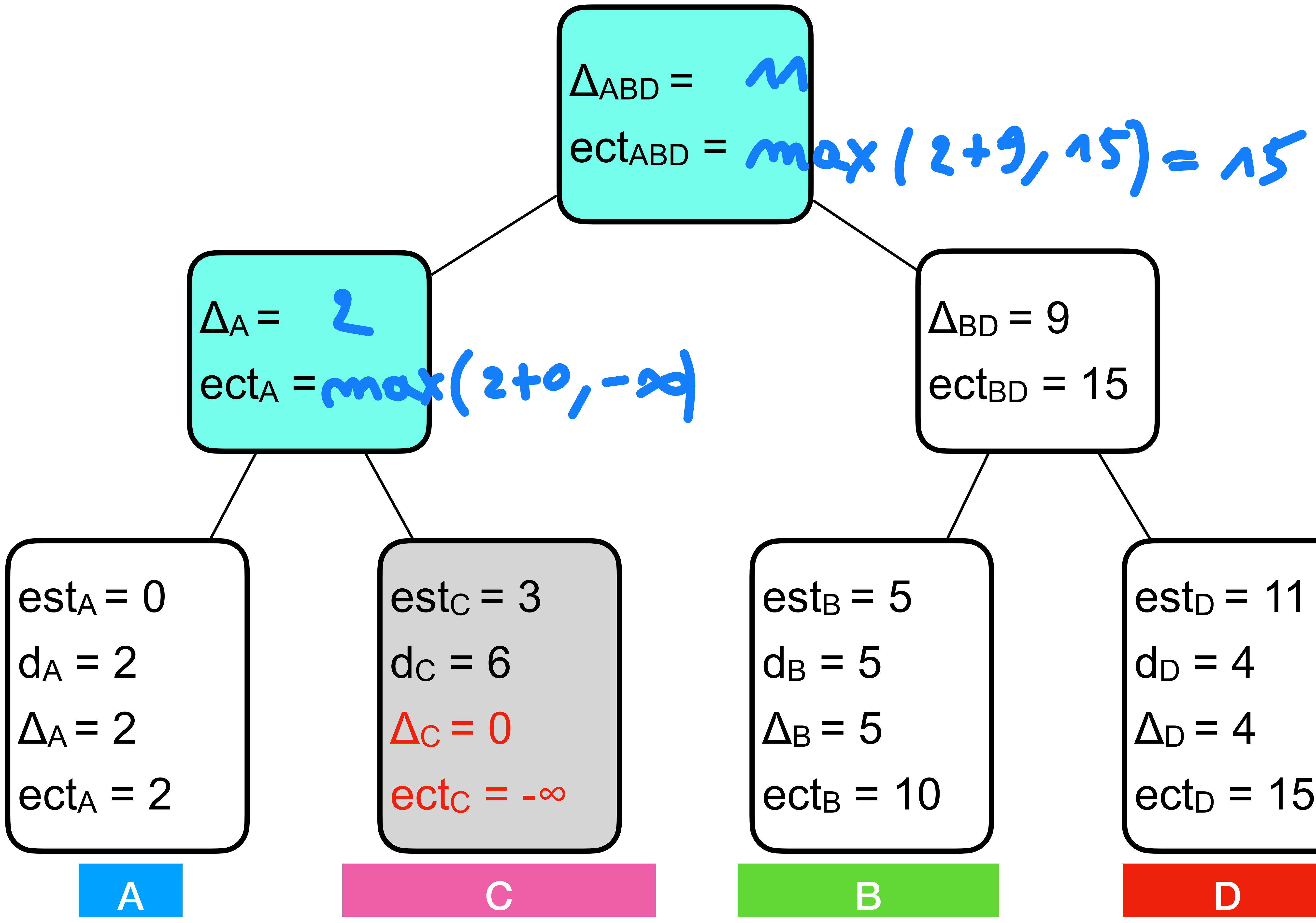




Total time complexity?

Θ-Tree: Incremental Removal of C

- To remove activity i from a Θ-tree: set $\Delta_i = 0$ and $ect_i = -\infty$.



Wrap-up on Θ-trees

A Θ-tree for a set Ω of n activities is

– a balanced binary tree,

– whose leaf nodes correspond to the activities of Ω (sorted according to est),

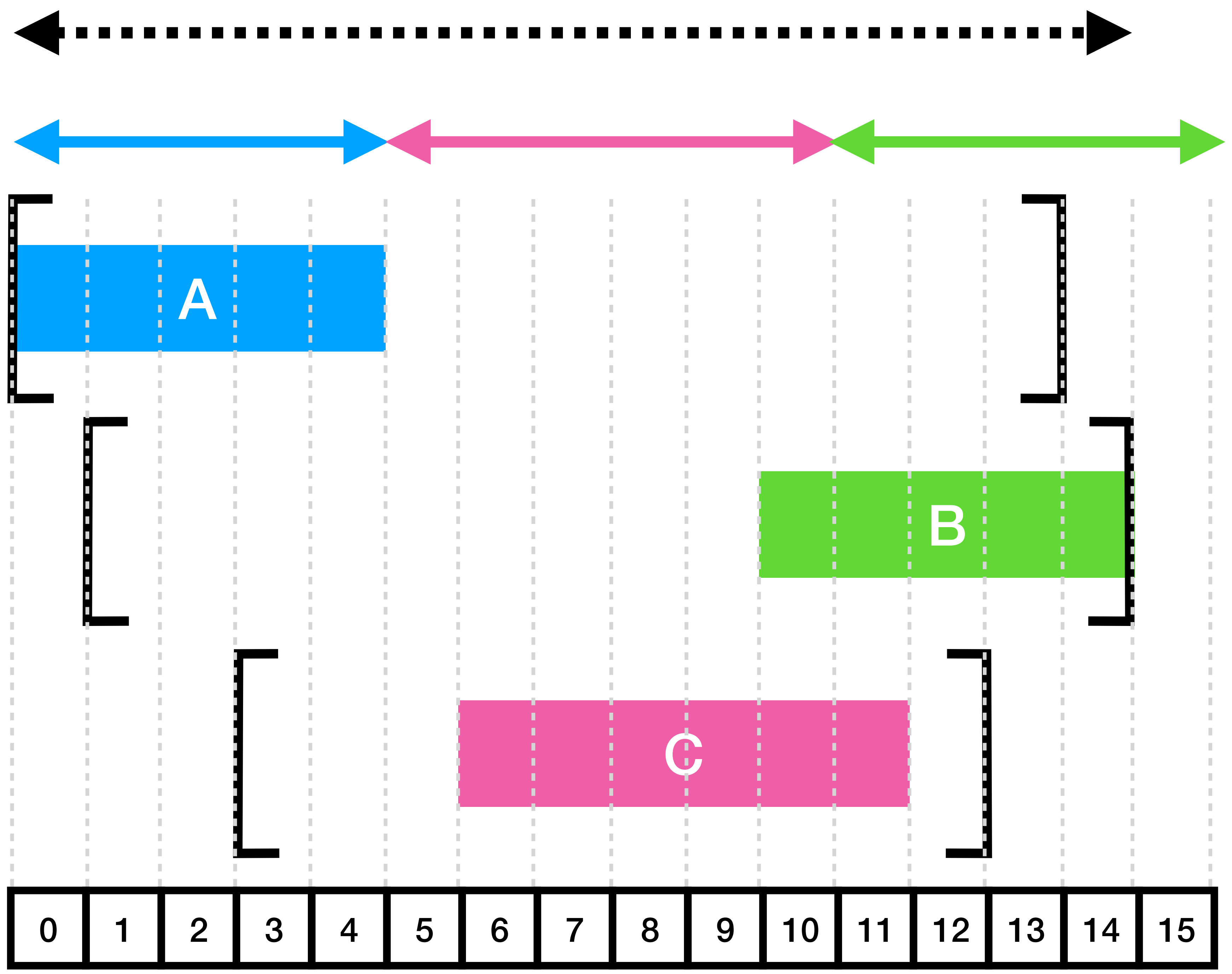
– whose internal nodes have intermediate Δ and ect values, and

– whose root node has ect_Ω.

Operation	Time complexity	Spec
init({1..n})	O(n log n)	Initialize an empty Θ-tree for the activites {1..n}
insert(i)	O(log n)	Insert activity i into the Θ-tree
remove(i)	O(log n)	Remove activity i from the Θ-tree
ect	O(1)	Return ect of the set of activities in the Θ-tree

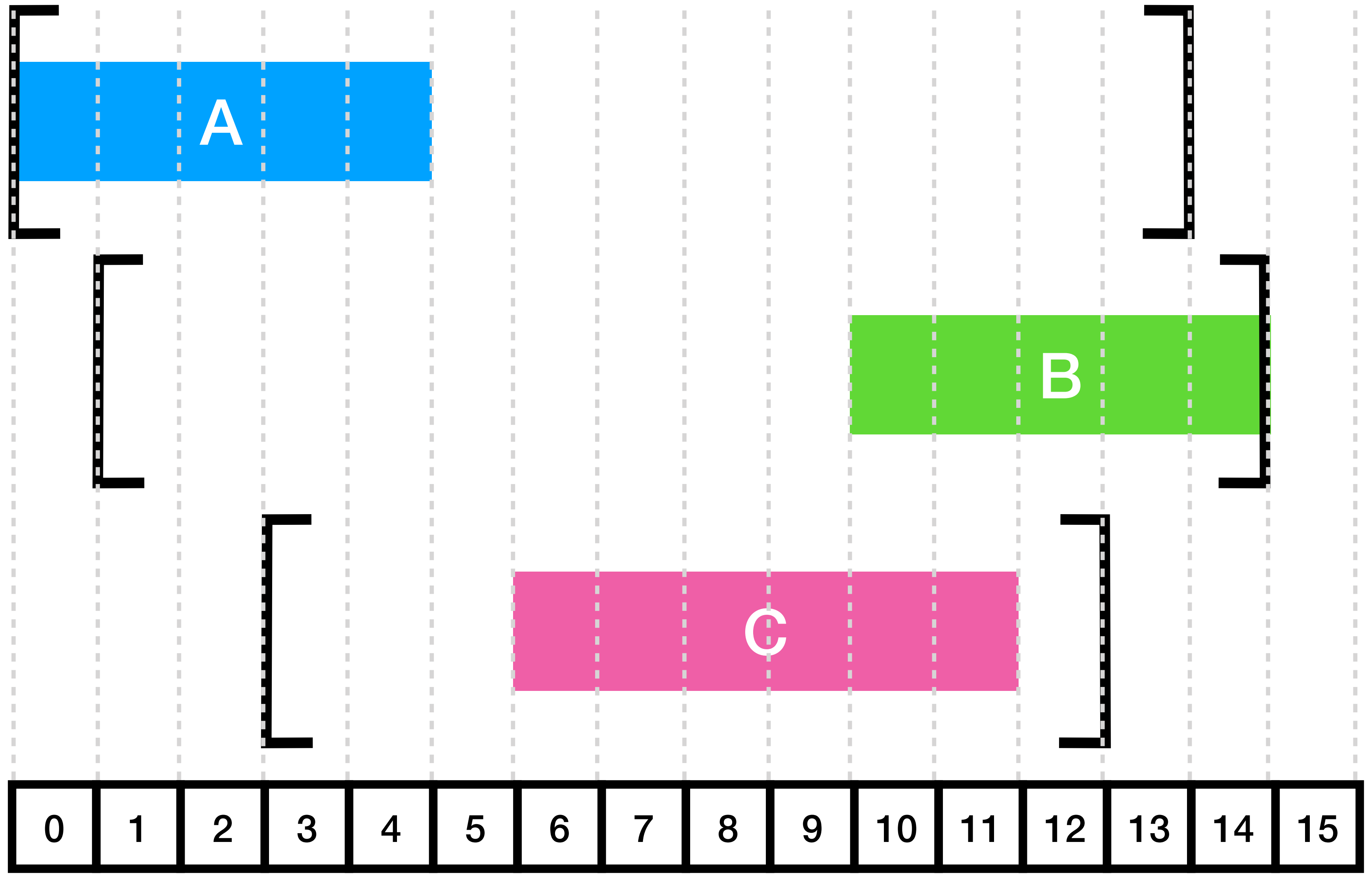
Overload Checker

Overload Checking = a feasibility check



Overload Checking = a feasibility check

- $\forall\, \Omega \subseteq T : (est_\Omega + d_\Omega > lct_\Omega \leadsto fail)$
- If there exists a subset of activities that cannot be processed within its bounds, then no solution exists.
  Example:



This failure
is *not* captured by the
binary decomposition
of Disjunctive.

- Take $\Omega = \{A,B,C\}$:
  $est_\Omega = 0$, $d_\Omega = 5+5+6 = 16$, $lct_\Omega = 15$, $0+16 > 15 \leadsto$ fail.

Overload Checking: time complexity?

- $\forall\ \Omega \subseteq T : (est_\Omega + d_\Omega > lct_\Omega \rightsquigarrow \text{fail})$
- We need to enumerate *all* subsets Ω of T, hence $2^{|T|}$ checks.
- It is not very practical to embed an algorithm of exponential time complexity in a propagator.
- We need something else…

Overload Checking: improve efficiency

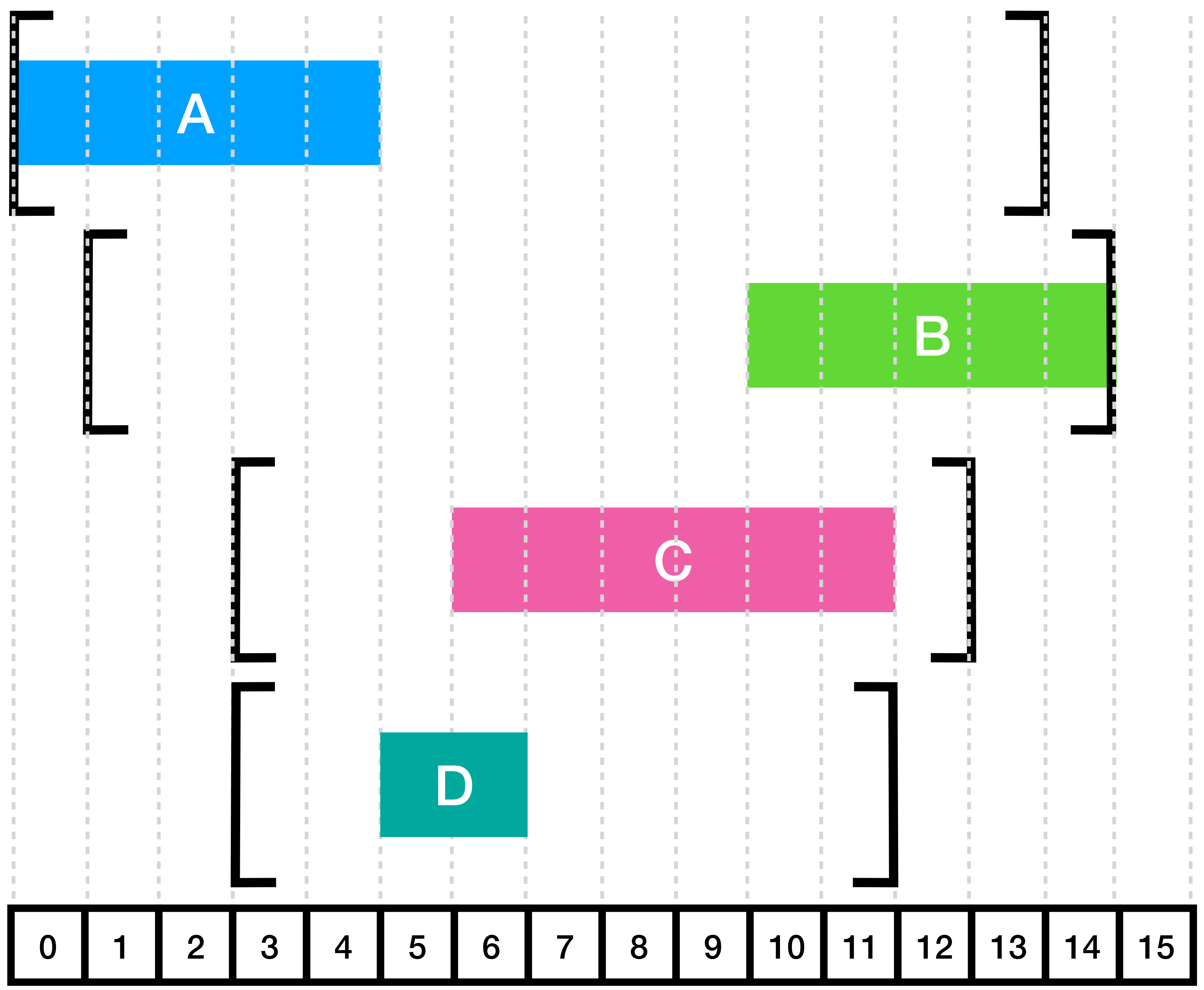
Left cut $LCut(T, j) = \{i \mid i \in T \;\&\; lct_i \leq lct_j\}$.

Example: $T = \{A,B,C,D\}$

$LCut(T,A) =$

$LCut(T,C) =$

$LCut(T,B) =$



Overload Checking: reformulation with LCut

$\forall\, \Omega \subseteq T : (est_\Omega + d_\Omega > lct_\Omega \rightsquigarrow \text{fail})$

can be reformulated as:

$\forall\, j \in T : ect_{LCut(T,j)} > lct_{LCut(T,j)} \rightsquigarrow \text{fail}$

equivalent to

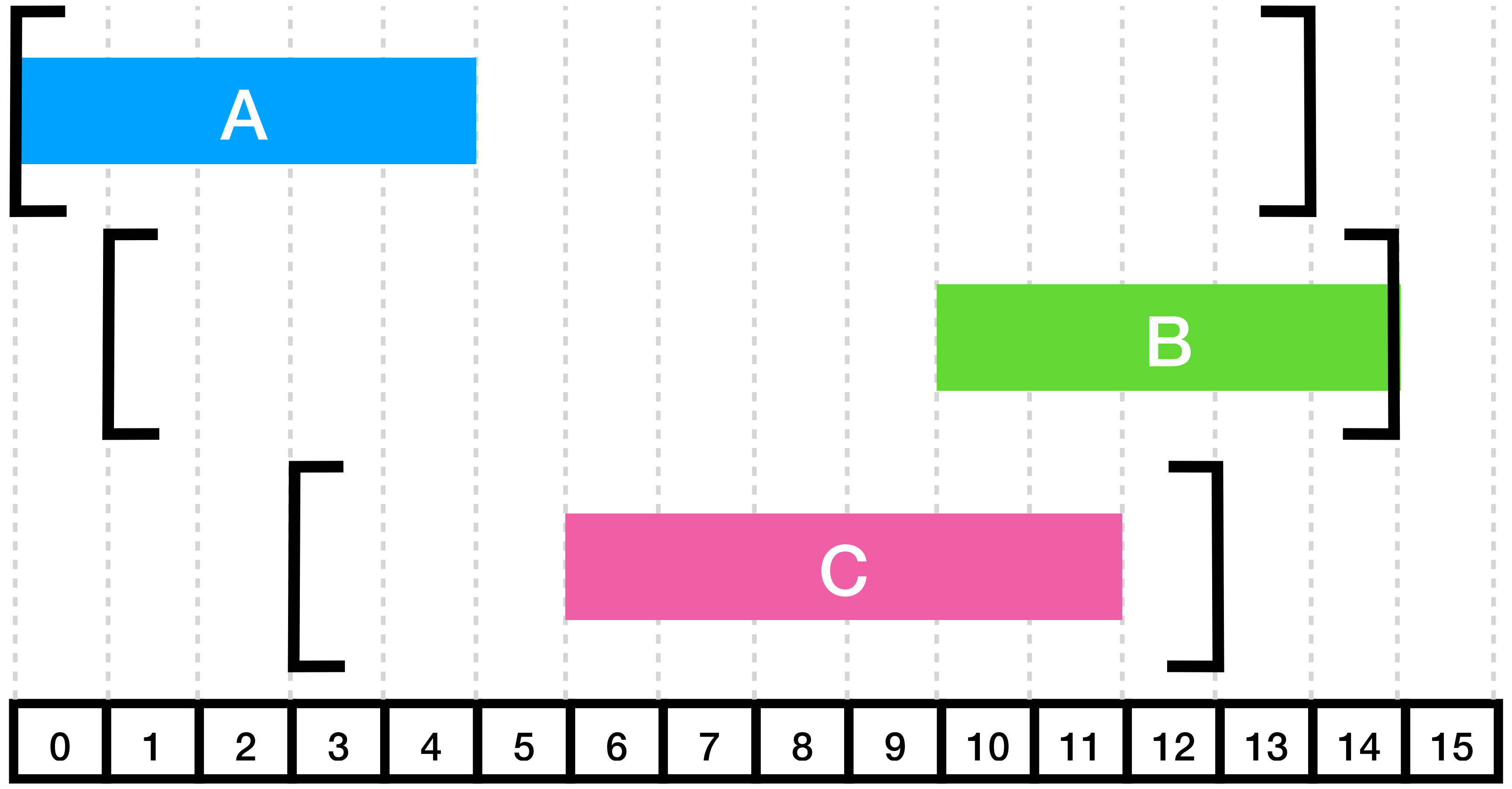
by definition

$\forall\, j \in T : ect_{LCut(T,j)} > lct_j \rightsquigarrow \text{fail}$

What do we gain? Complexity?

We can now compute it efficiently 💡

Overload Checking: example with LCut



For example, take j = B,
with LCut(T,B) = {A,B,C} = subset of activities ending by the end of B:

$ect_{LCut(T,B)} = 16 > lct_{LCut(T,B)} = 15 = lct_B$ (the red equality is true by definition).

Overload Checker taking O(n² log n) time

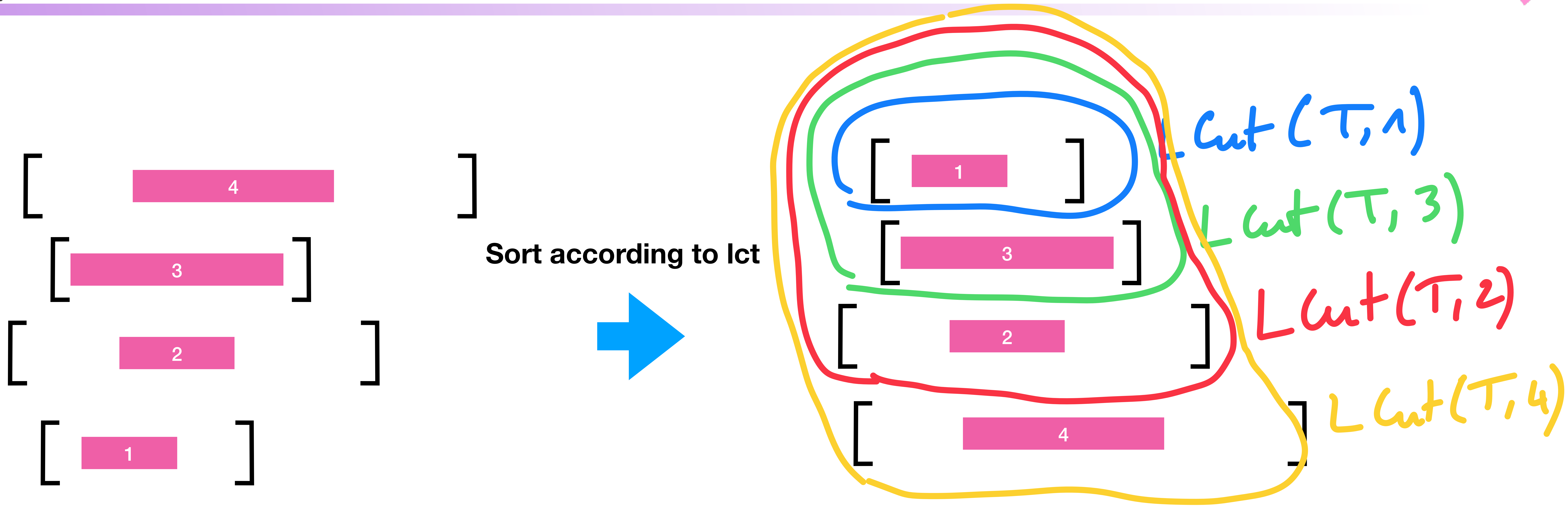
Overload checking rule:

$\forall\, j \in T : (ect_{LCut(T,j)} > lct_j \rightsquigarrow \text{fail})$

O(n² log n) time

```
OverloadCheckInefficient(T={1..n}) {
 for (j ← {1..n}) {
    Θ ← Θ-Tree.init({1..n}) // O(n log n) time
    for (i ← LCut(T,j)) {
       Θ.insert(i) // O(log n) time
    }
    if (Θ.ect > lctj) { // O(1) time
      throw InconsistencyException
    }
 }
}
```

Nested LCut



$LCut(T,j) = \{i \mid i \in T \ \& \ lct_i \leq lct_j\}$

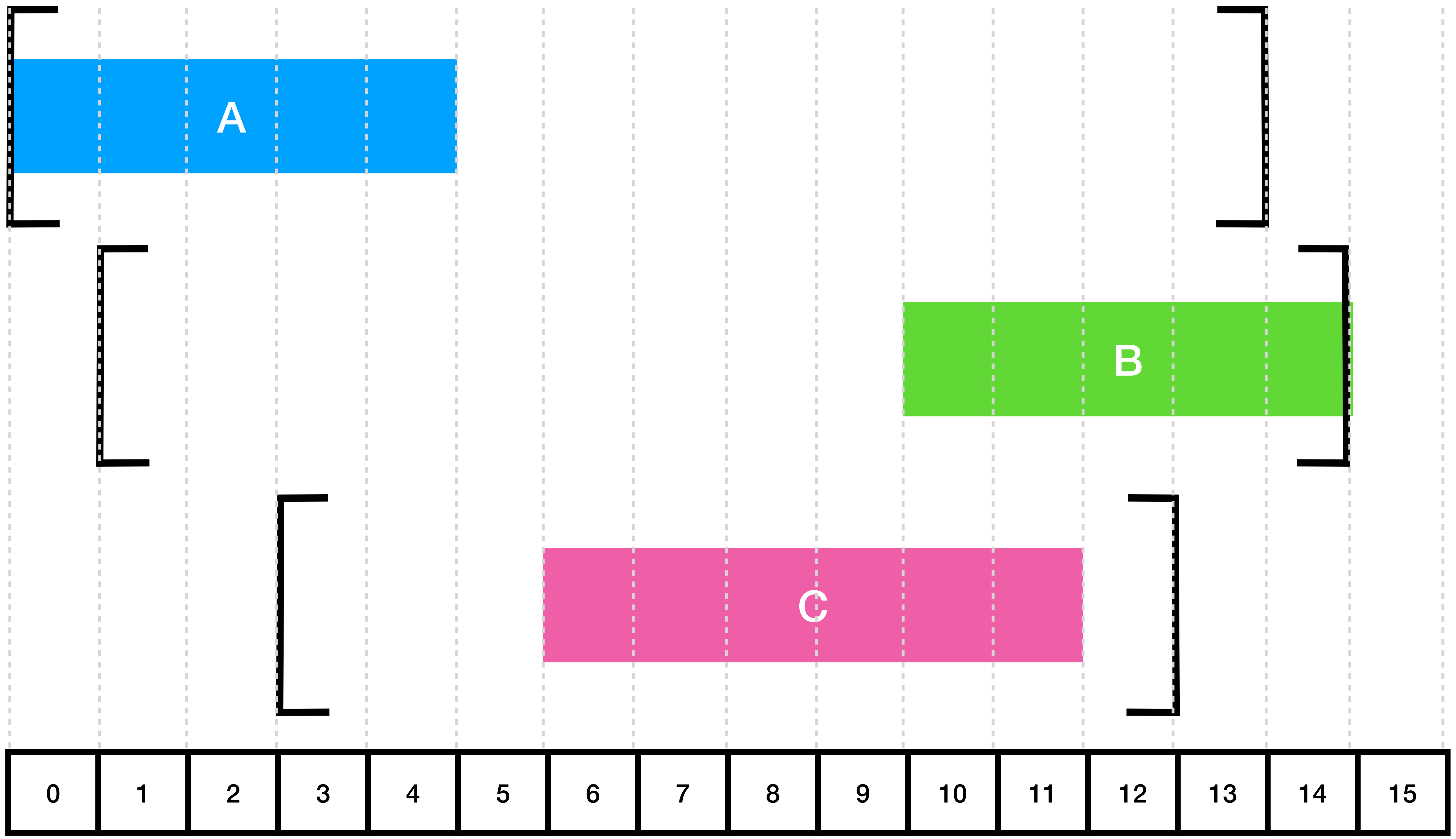
Overload Checker taking O(n log n) time

- Let T = {1..n} be ordered such that $lct_1 \leq \ldots \leq lct_n$.
- Then LCut(T,1) ⊆ LCut(T,2) ⊆ … ⊆ LCut(T,n) = T: *all* activities are eventually inserted.

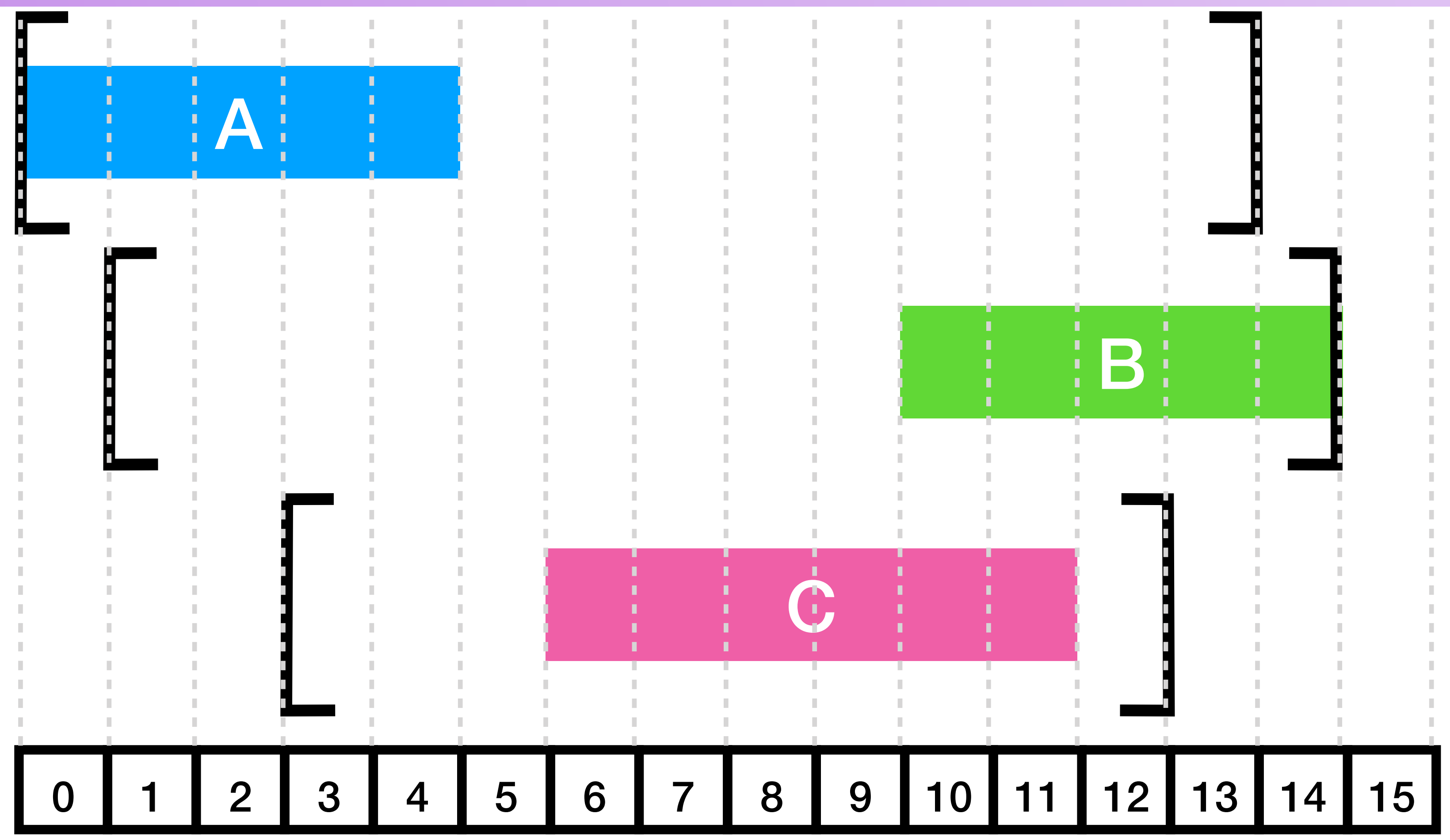
```
OverloadCheckEfficient(T={1..n}) {
  Θ ← Θ-Tree.init({1..n}) // O(n log n) time
  T ← sortAZ([1..n],sortKey = lct) // O(n log n) time
  for (j ← T) {
    Θ.insert(j) // O(log n) time
    // invariant: Θ contains LCut(T,j)
    if (Θ.ect > lctj) { // O(1) time
      throw InconsistencyException
    }
  }
}
```

Overload Checking with Θ-Tree: an example

- Application of *OverloadCheckEfficient* algorithm on this example

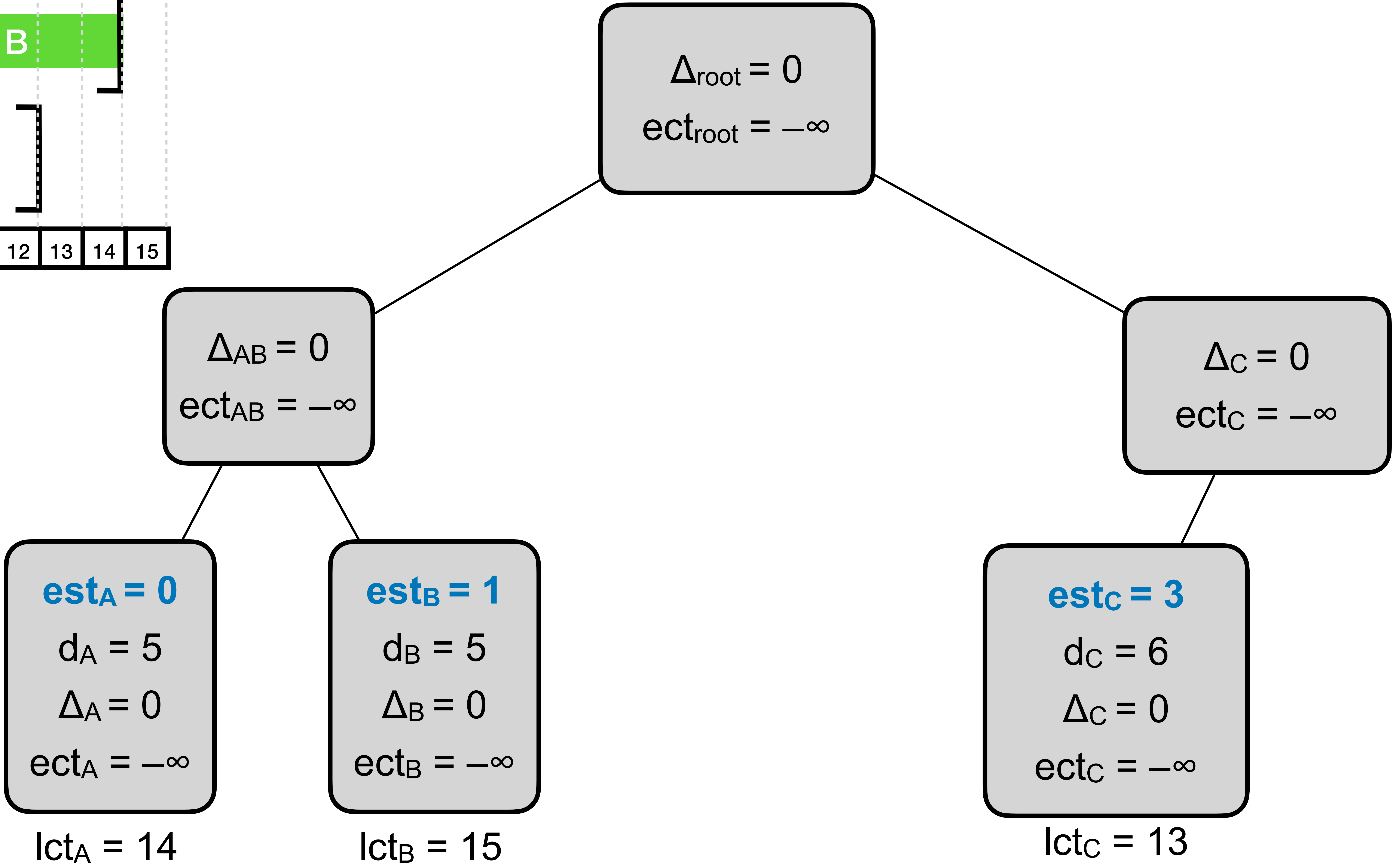


Overload Checking with Θ-Tree: an example

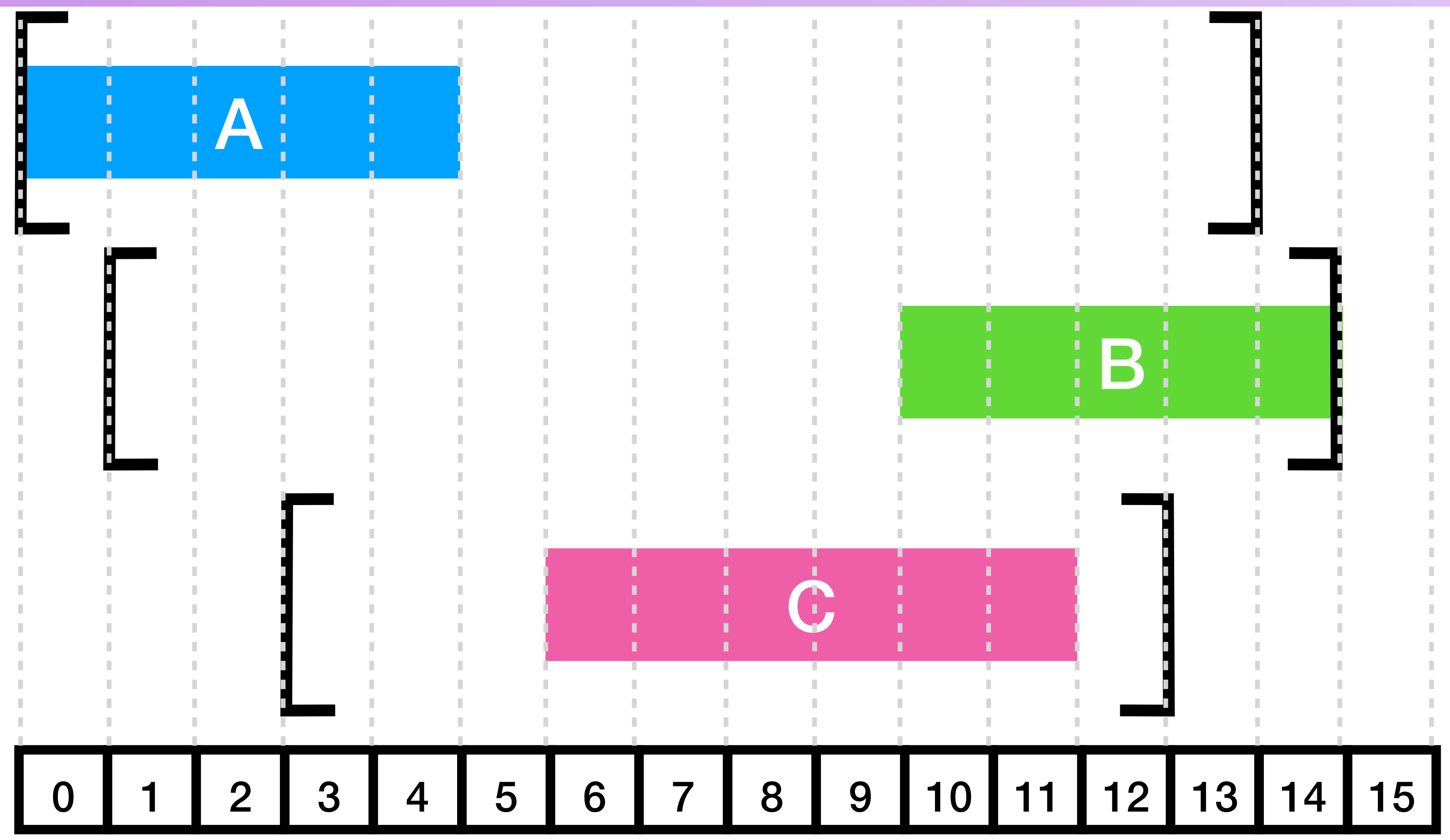



```
OverloadCheckEfficient(T={1..n}) {
  T ← sortAZ([1..n],sortKey = lct)
  Θ ← Θ-Tree.init({1..n})
  for (j ← T) { // [C,A,B]
    Θ.insert(j)
    if (Θ.ect > lctⱼ) {
      throw InconsistencyException
    }
  }
}
```

Empty Θ-Tree initialization

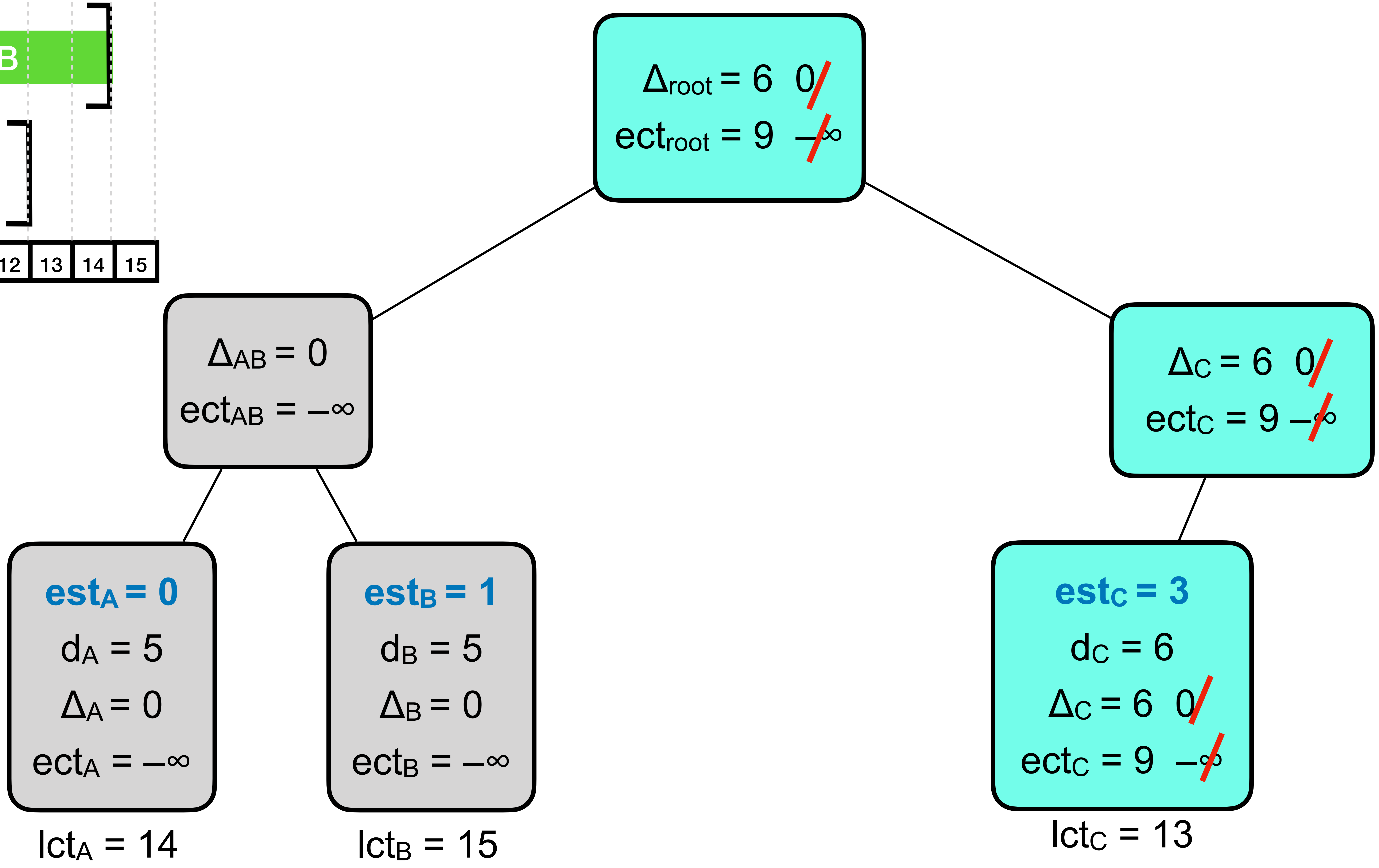


Overload Checking with Θ-Tree: an example

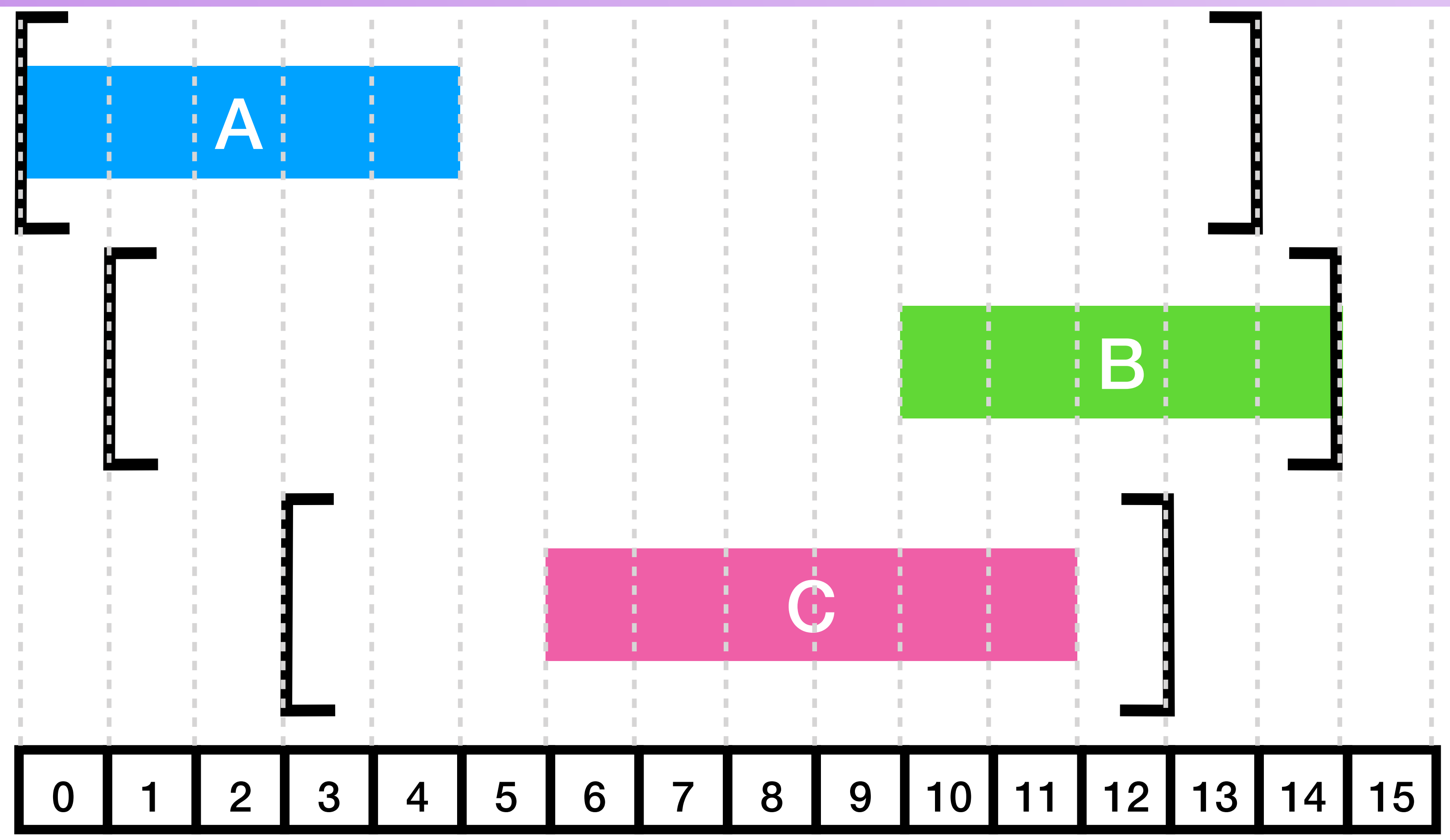



```
OverloadCheckEfficient(T={1..n}) {
  T ← sortAZ([1..n],sortKey = lct)
  Θ ← Θ-Tree.init({1..n})
  for (j ← T) { // [C,A,B]
    Θ.insert(j) // j = C
    if (Θ.ect > lctj) {
      throw InconsistencyException
    }
  }
}
```

Insertion of C

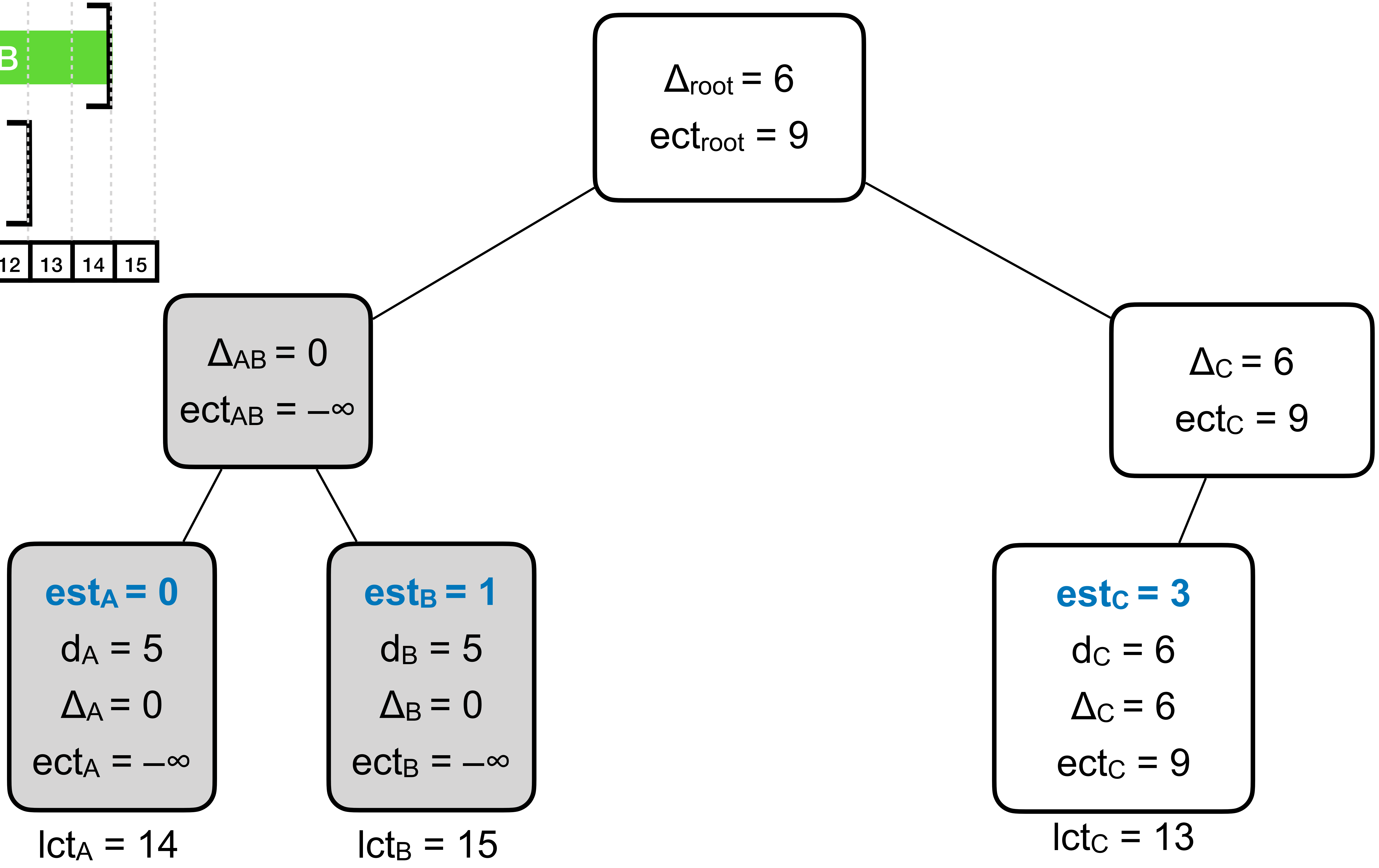


Overload Checking with Θ-Tree: an example

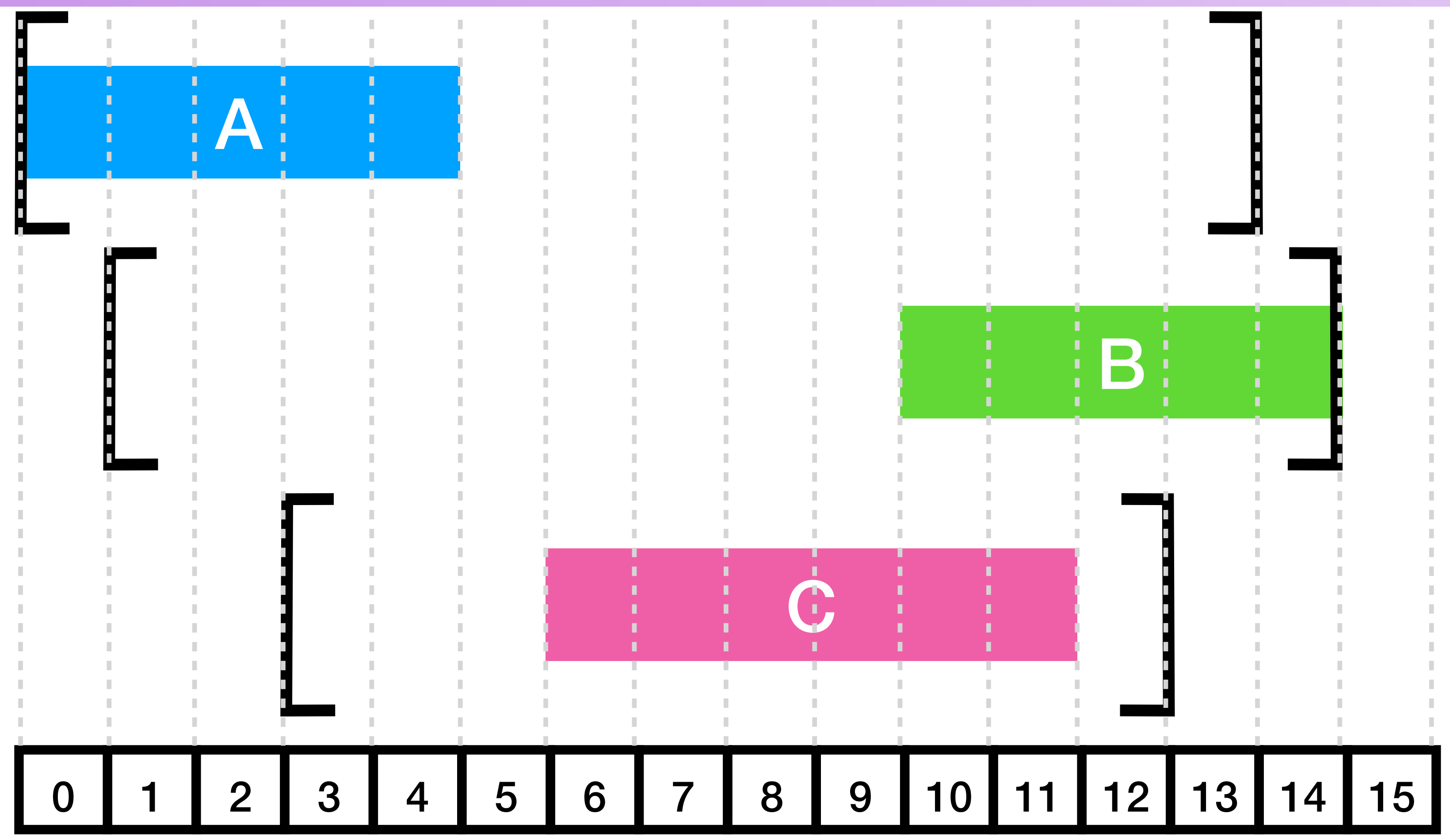



```
OverloadCheckEfficient(T={1..n}) {
  T ← sortAZ([1..n],sortKey = lct)
  Θ ← Θ-Tree.init({1..n})
  for (j ← T) { // [C,A,B]
    Θ.insert(j) // j = C
    if (Θ.ect > lct_j) { // 9 > 13 ✅
      throw InconsistencyException
    }
  }
}
```

Feasibility check

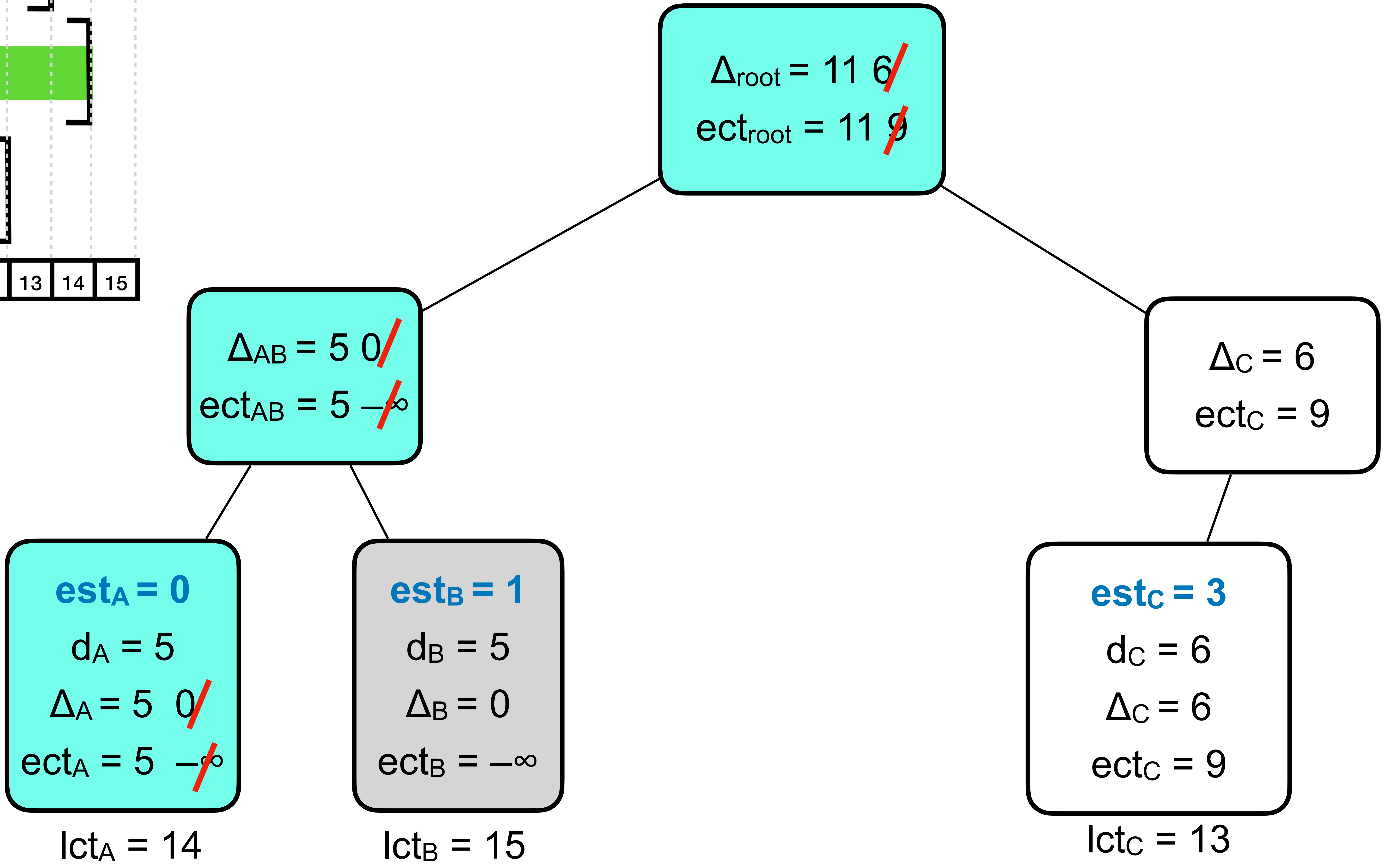


Overload Checking with Θ-Tree: an example

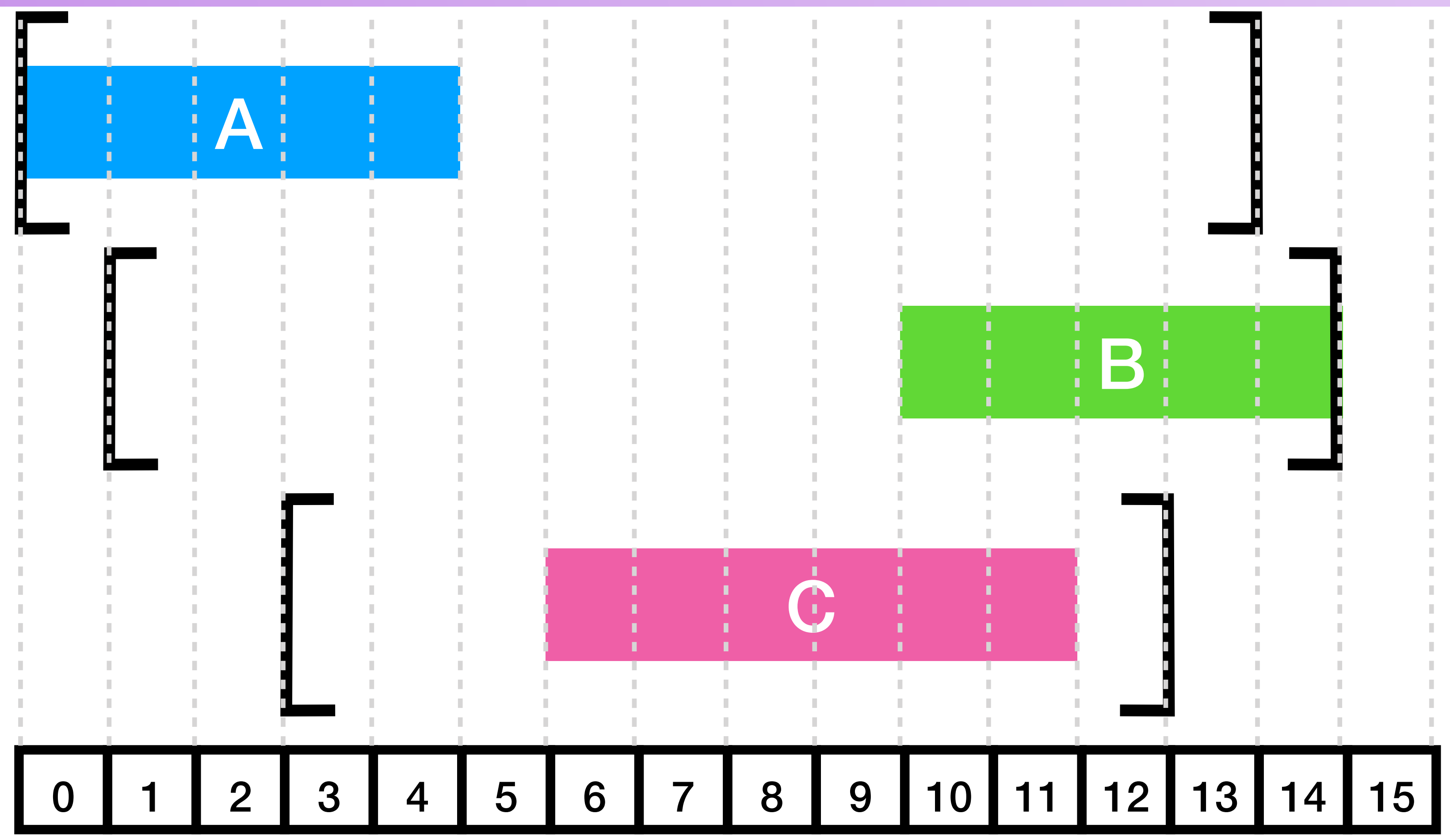



```
OverloadCheckEfficient(T={1..n}) {
  T ← sortAZ([1..n],sortKey = lct)
  Θ ← Θ-Tree.init({1..n})
  for (j ← T) { // [C,A,B]
    Θ.insert(j) // j = A
    if (Θ.ect > lctj) {
      throw InconsistencyException
    }
  }
}
```

Insertion of A

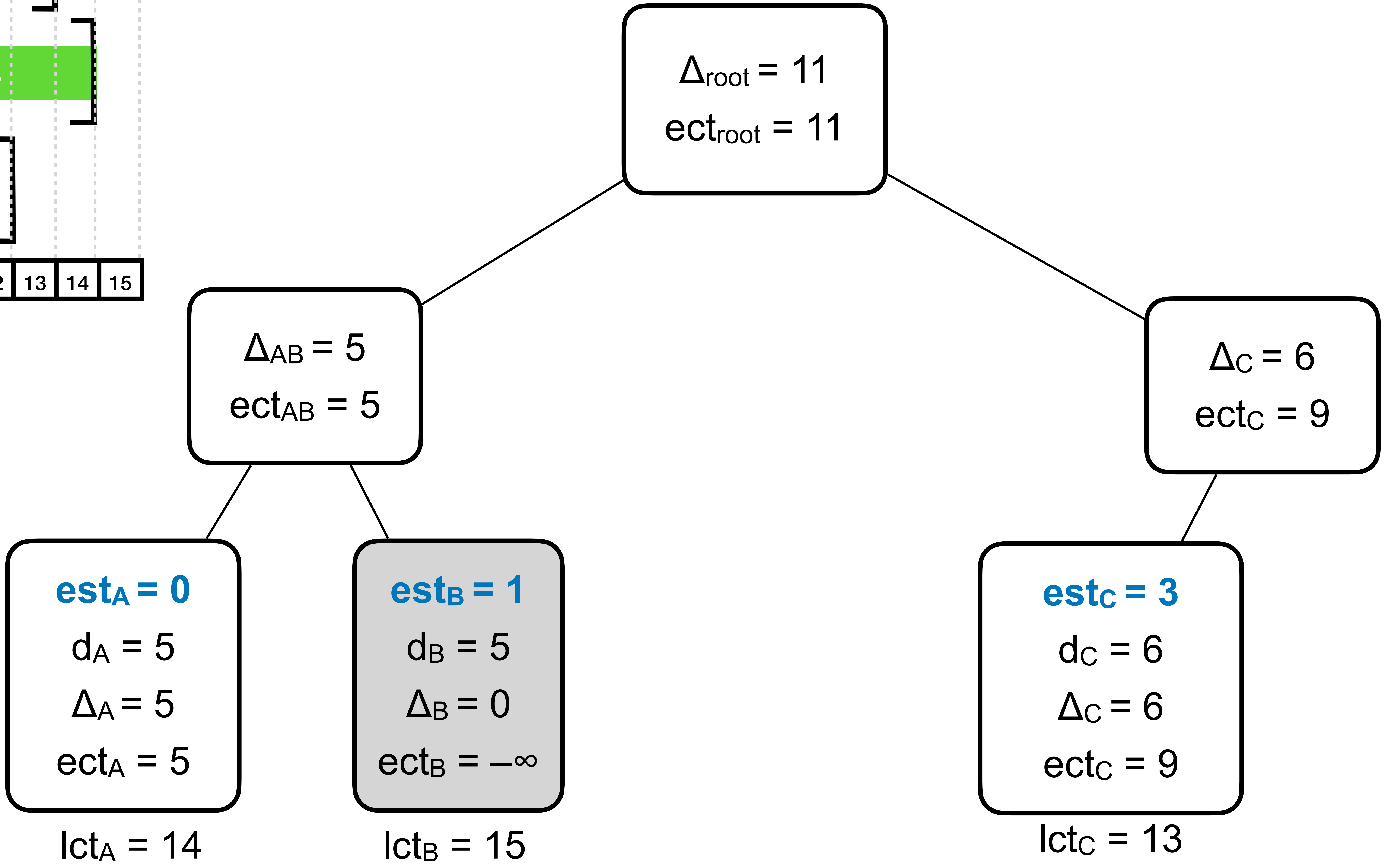


Overload Checking with Θ-Tree: an example

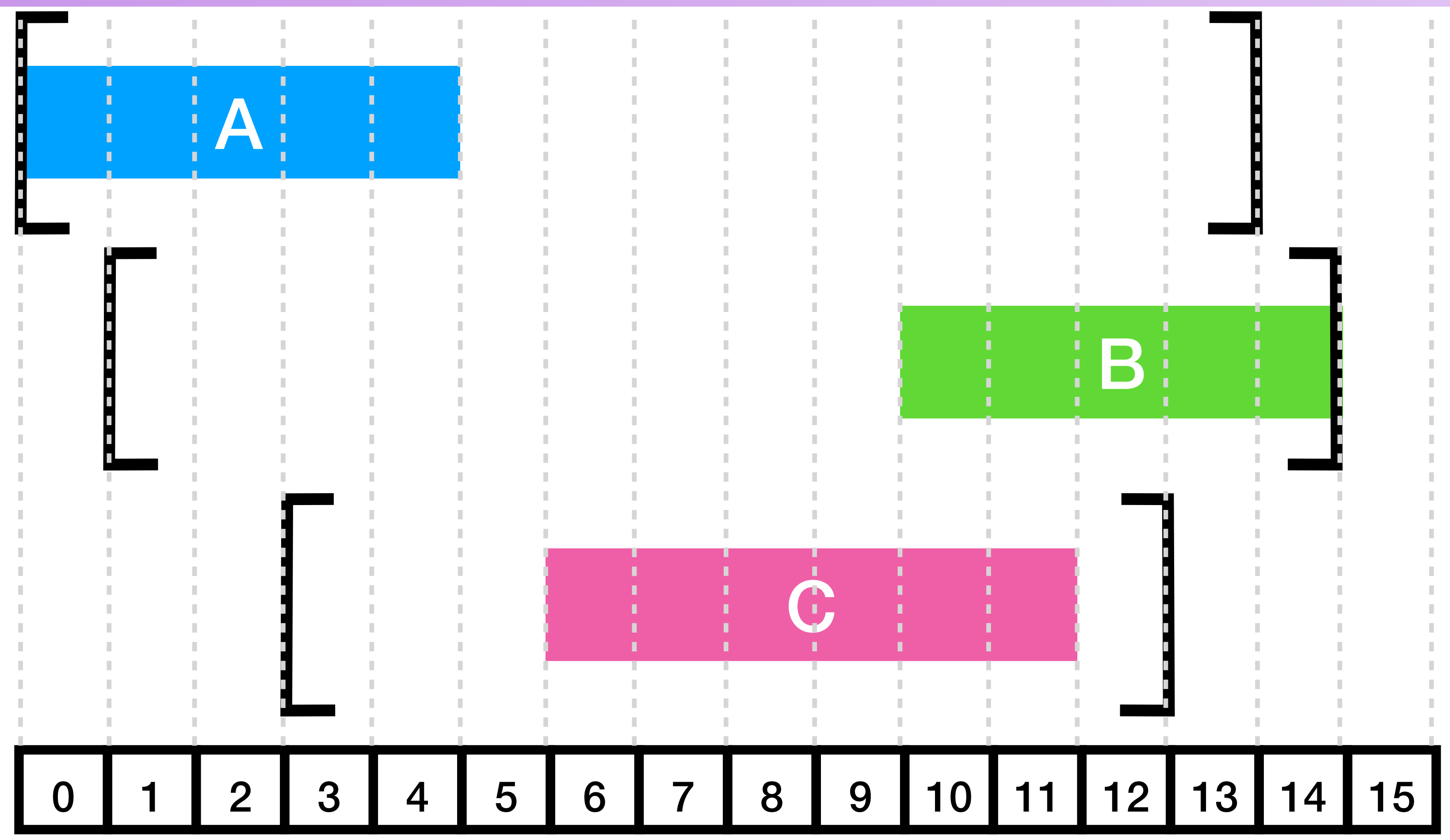



```
OverloadCheckEfficient(T={1..n}) {
  T ← sortAZ([1..n],sortKey = lct)
  Θ ← Θ-Tree.init({1..n})
  for (j ← T) { // [C,A,B]
    Θ.insert(j) // j = A
    if (Θ.ect > lctⱼ) { // 11 < 14 ✅
      throw InconsistencyException
    }
  }
}
```

Feasibility check

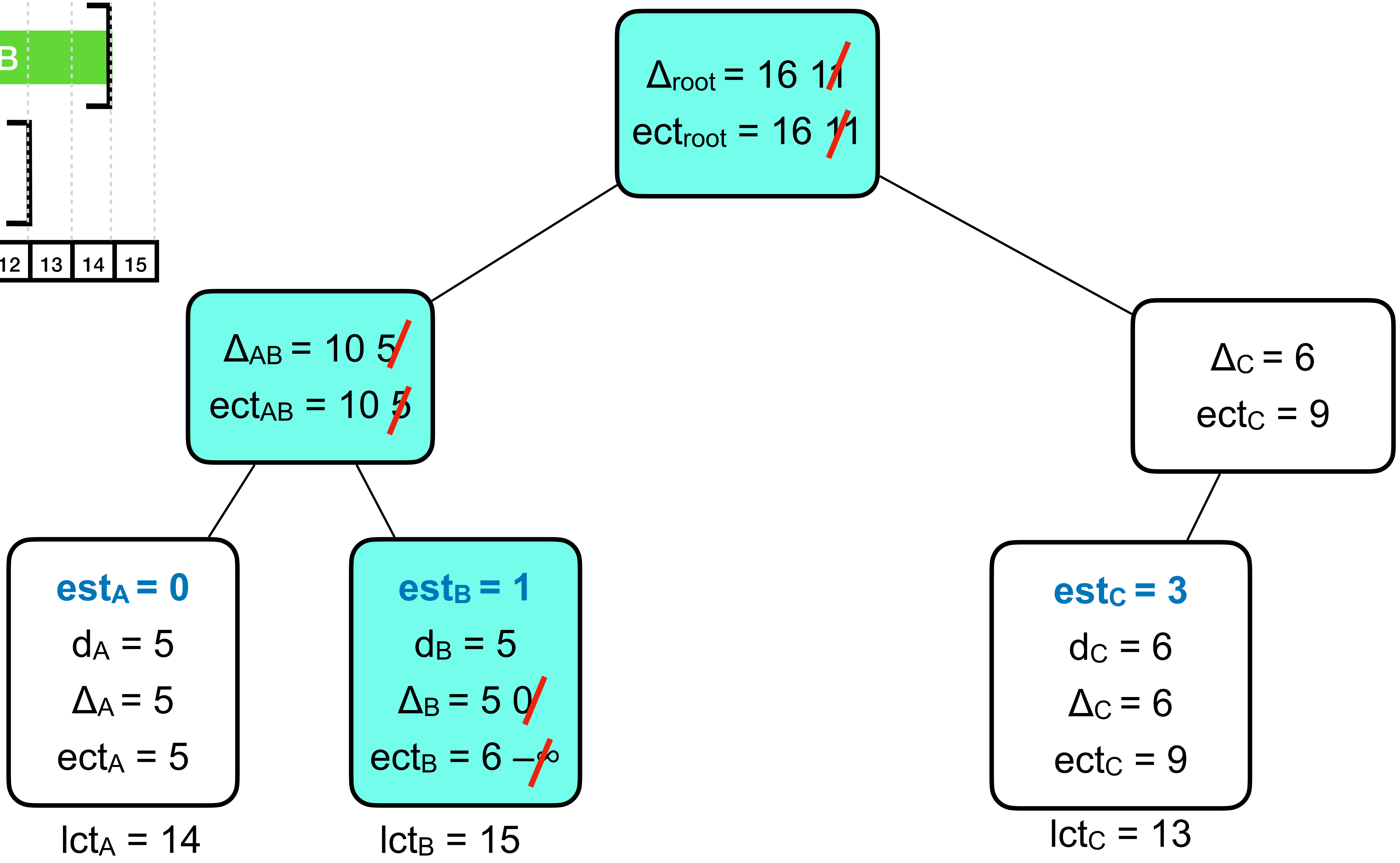


Overload Checking with Θ-Tree: an example

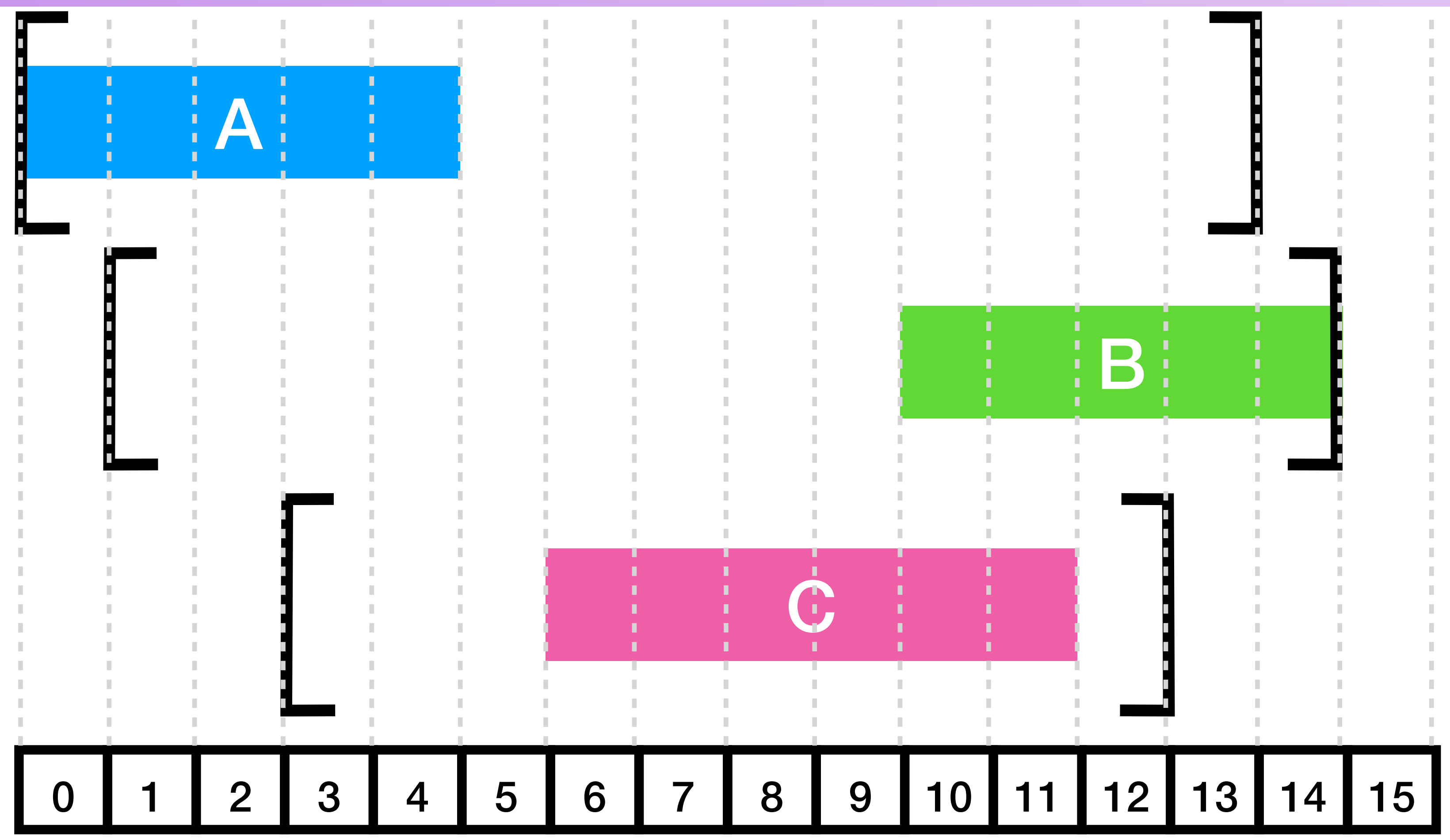



```
OverloadCheckEfficient(T={1..n}) {
  T ← sortAZ([1..n],sortKey = lct)
  Θ ← Θ-Tree.init({1..n})
  for (j ← T) { // [C,A,B]
    Θ.insert(j) // j = B
    if (Θ.ect > lctj) {
      throw InconsistencyException
    }
  }
}
```

Insertion of B

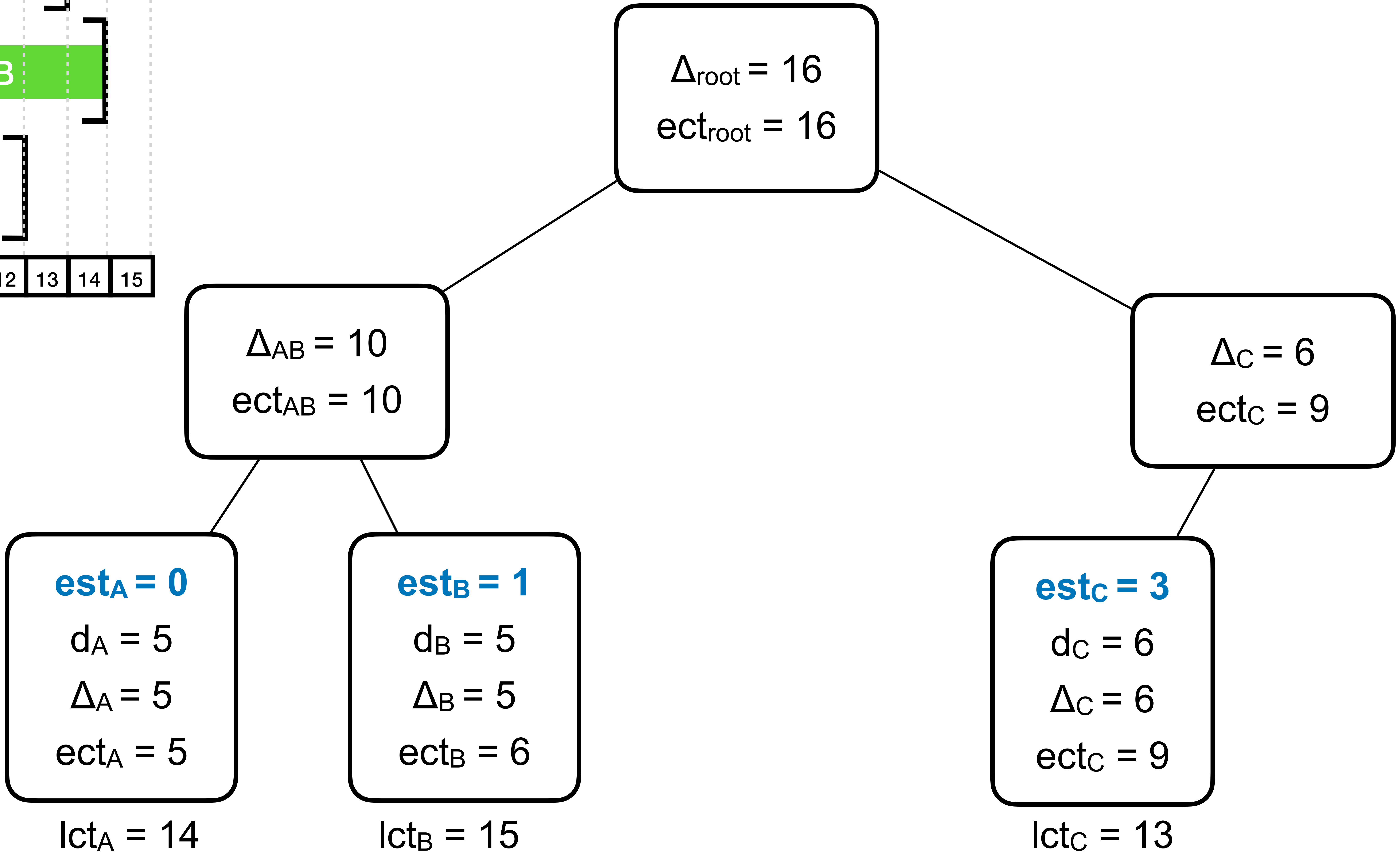


Overload Checking with Θ-Tree: an example




```
OverloadCheckEfficient(T={1..n}) {
  T ← sortAZ([1..n],sortKey = lct)
  Θ ← Θ-Tree.init({1..n})
  for (j ← T) { // [C,A,B]
    Θ.insert(j) // j = C
    if (Θ.ect > lctⱼ) { // 16 > 15 ✗
      throw InconsistencyException
    }
  }
}
```

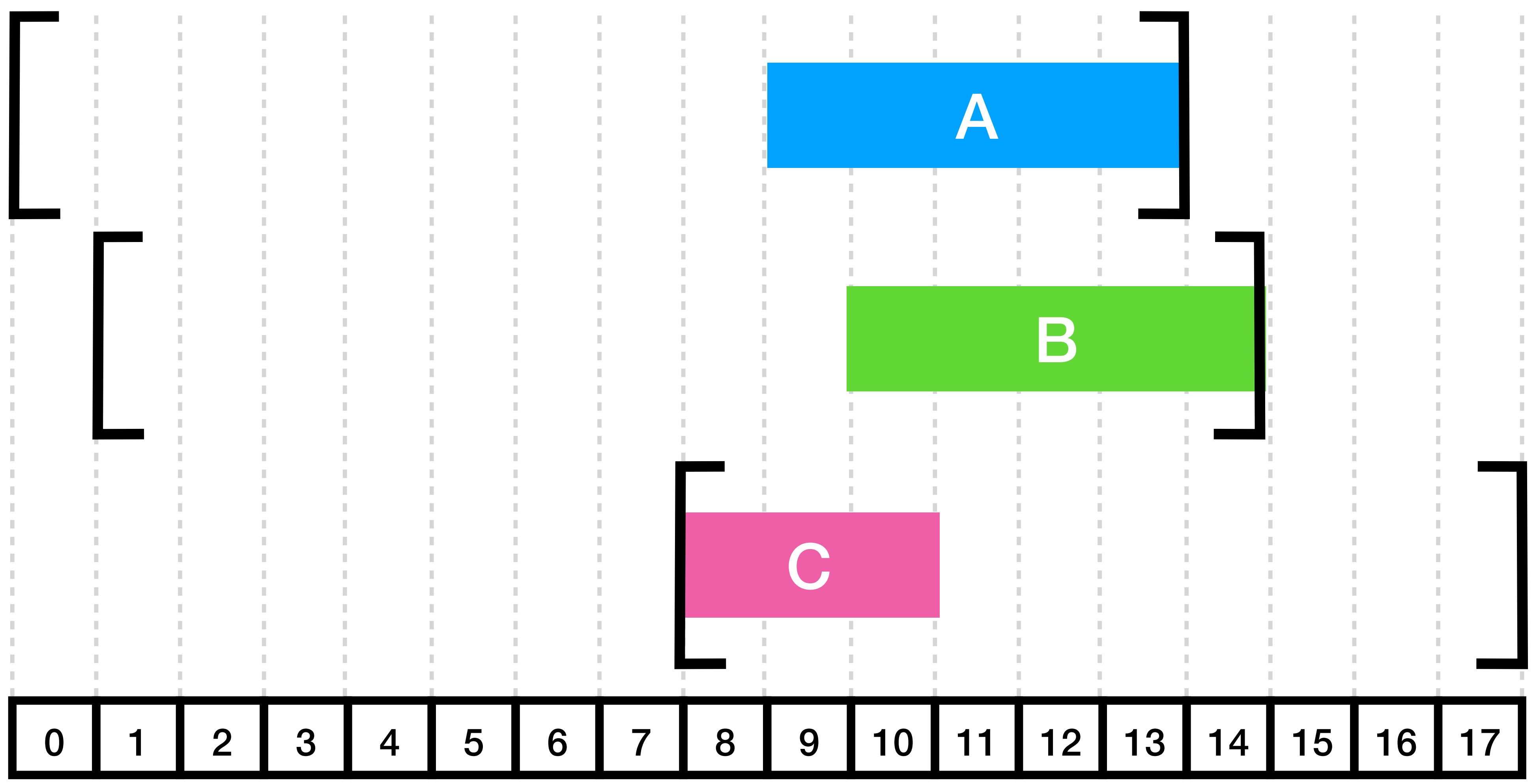
Feasibility check



Detectable Precedences

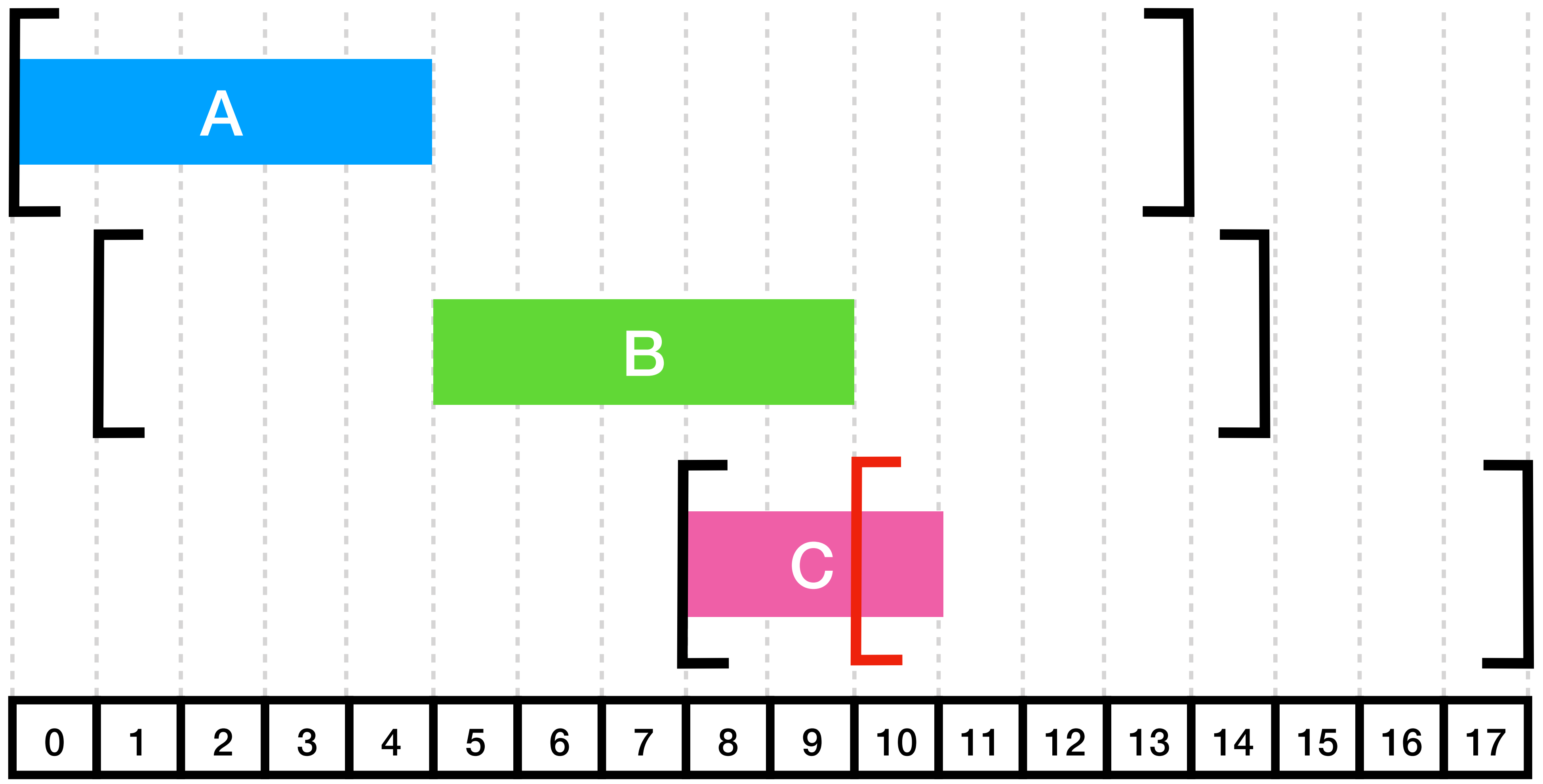
Detectable Precedences = a filtering rule

- Both A and B cannot be scheduled after C
- Therefore they must both be scheduled before



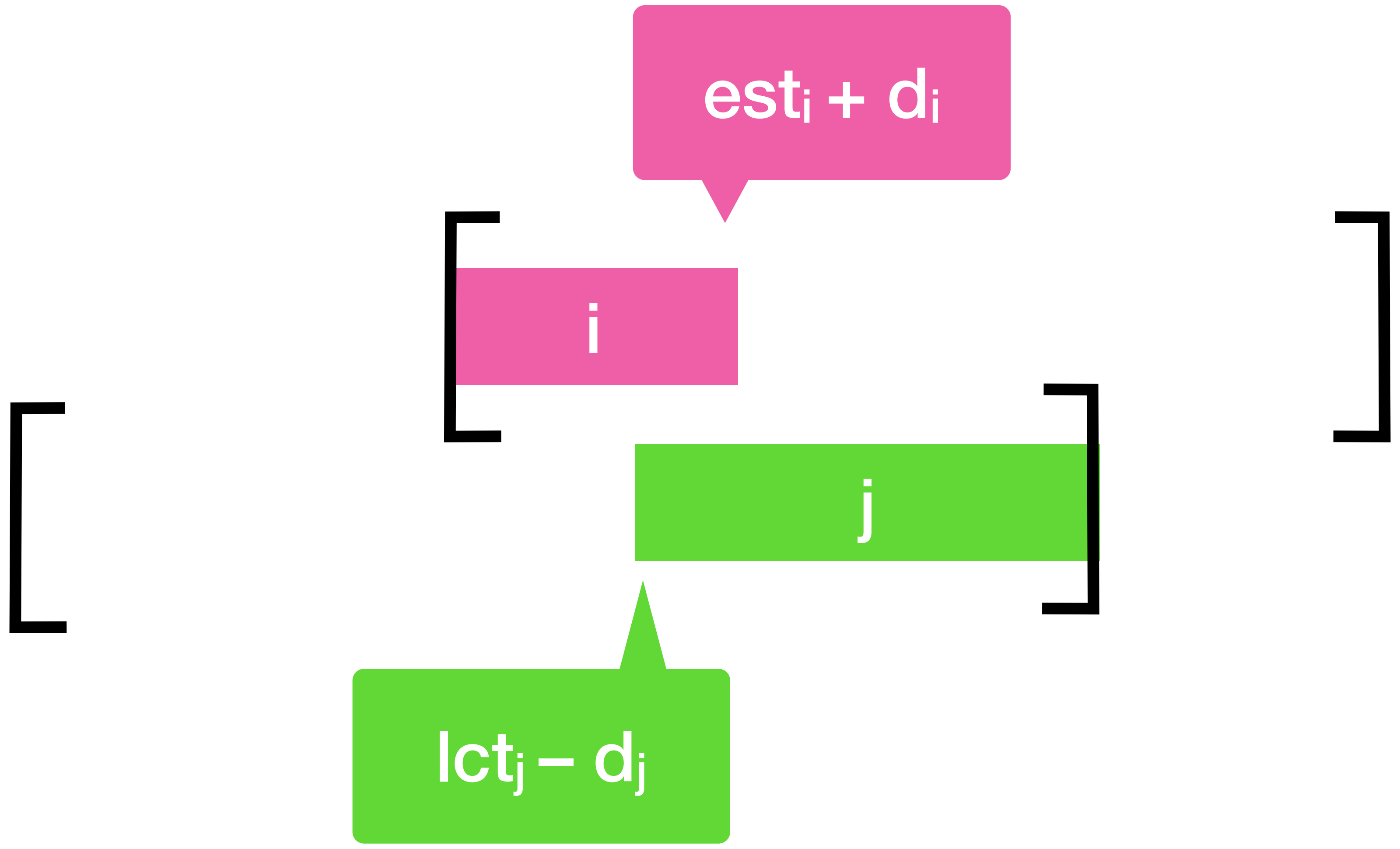
Detectable Precedences = a filtering rule

- Both A and B must end before C starts is denoted by {A,B} ≪ C
- By taking the earliest start of A and (duration A + duration B), we can filter (push) the start of C to 10



Detectable Precedences = a filtering rule

- A precedence $j \ll i$ is detectable if $est_i + d_i > lct_j - d_j$



that is if $ect_i > lst_j$ then activity j can*not* start after activity i ends.

Set of all activities with detectable precedence before i:
DPrec(T,i) $= \{ j \mid j \in T \setminus \{i\} \;\&\; est_i + d_i > lct_j - d_j \}$.

- Filtering: $est_i \leftarrow \max(est_i , ect_{DPrec(T,i)})$, for all $i \in T$.

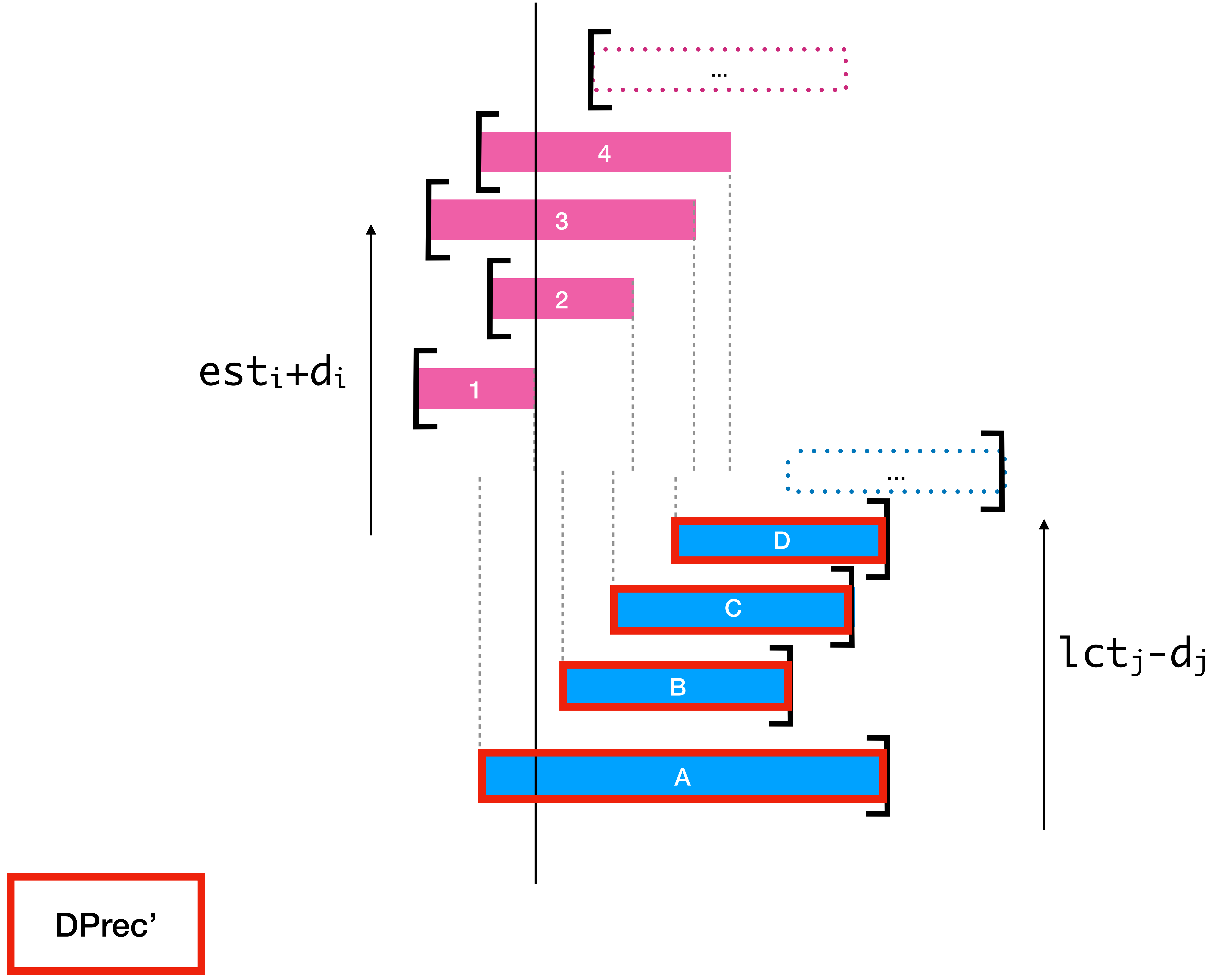
Nested sets?

- $DPrec'(T,i) = \{ j : j \in T \ \ \& \ \ est_i + d_i > lct_j - d_j \}$.
  Note that activity i is sometimes in DPrec'(T,i).

- Hence: $DPrec(T,i) = DPrec'(T,i) \setminus \{i\}$.

- In what order should the activities i be considered to have nested DPrec'(T,i) sets?

Order on i to have nested DPrec'(T,i) sets



Iterating on activities

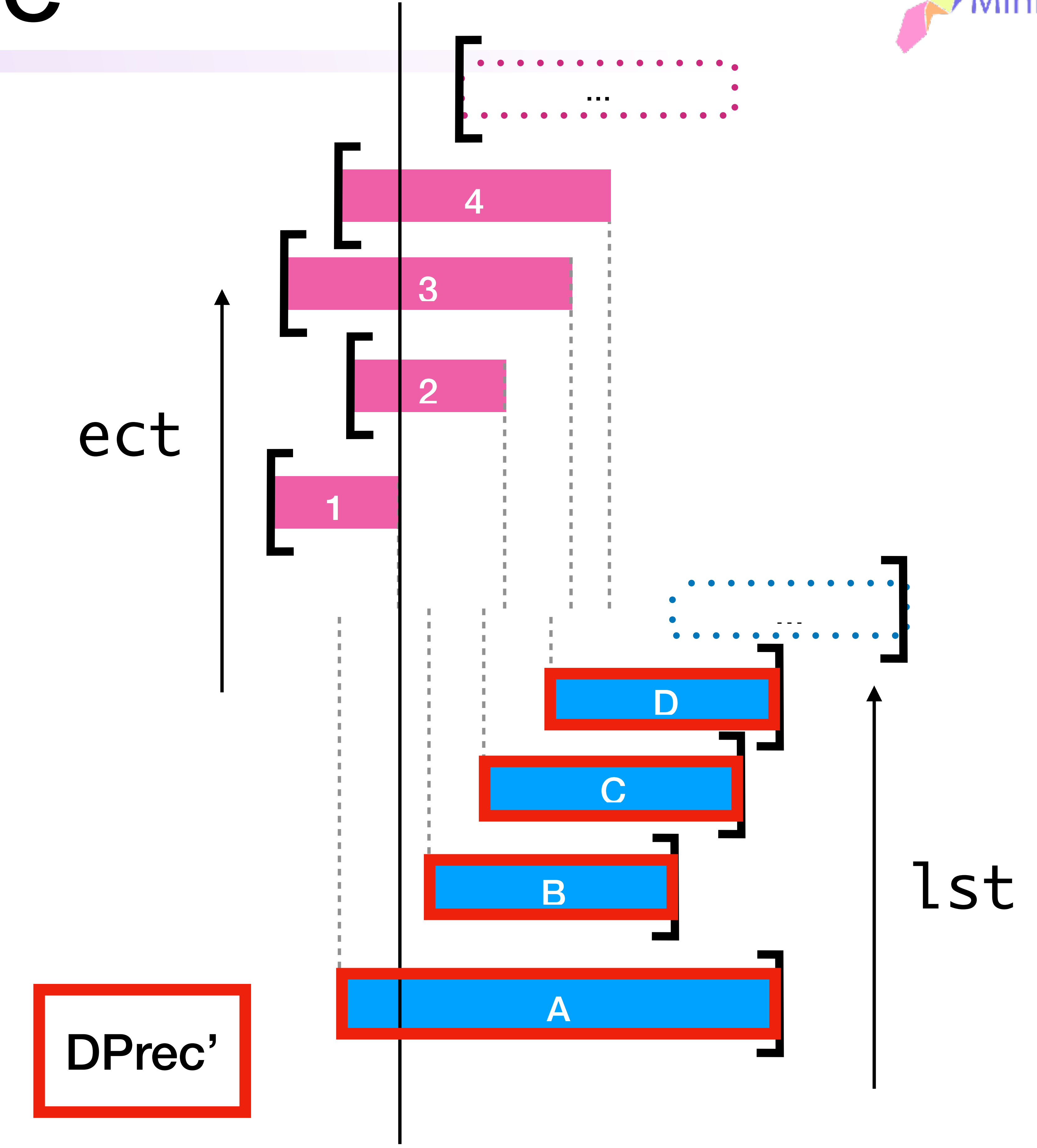
- Let $T = \{1..n\}$ be ordered such that
  - $est_1 + d_1 \leq est_2 + d_2 \leq \ldots \leq est_n + d_n$
  - Then: $\boxed{DPrec'(T,1) \subseteq DPrec'(T,2) \subseteq \ldots \subseteq DPrec'(T,n)}$
- This is exactly what we are looking for:
  an order to consider the activities i of T such that the detectable precedence set is growing monotonically, as this is very important for computing all $ect_{DPrec(T,i)}$ efficiently & incrementally with a Θ-tree.
- Note that DPrec'(T,n) is *not* necessarily T:
  not necessarily all activities are eventually inserted into the initialized Θ-tree.

Detectable Precedences: O(n log n) time

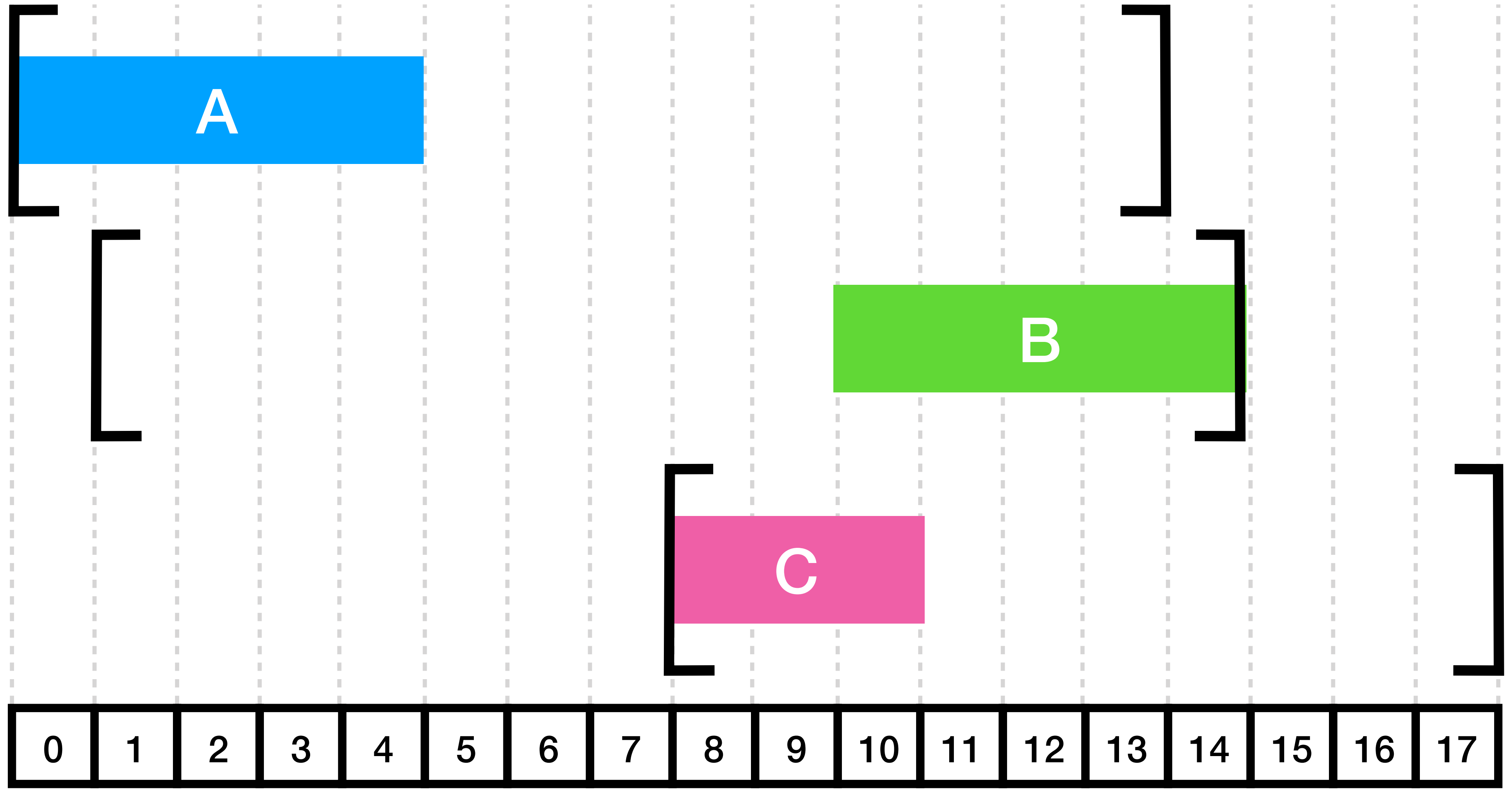
```
DetectablePrecedence(T={1..n}) {
  Tlst ← sortAZ([1..n],sortKey = lct-d) // O(n log n)
  Tect ← sortAZ([1..n],sortKey = est+d) // O(n log n)
  ite ← iterator(Tlst)
  j ← ite.next() // candidate precedence of i
  Θ ← Θ-Tree.init({1..n}) // O(n log n) time
  for (i ← Tect) {
    while (esti+di > lctj-dj) {
      Θ.insert(j) // O(log n) time
      if (ite.hasNext()) {j ← ite.next()} else {break}
    }
    est'i ← max(esti, ectΘ\i) // O(log n) time
  }
  esti ← est'i, ∀i∈T
}
```

This is executed at most n times

Because Θ contains DPrec'(T,i) and not DPrec(T,i): Θ.remove(i), use Θ.ect for max, Θ.insert(i).

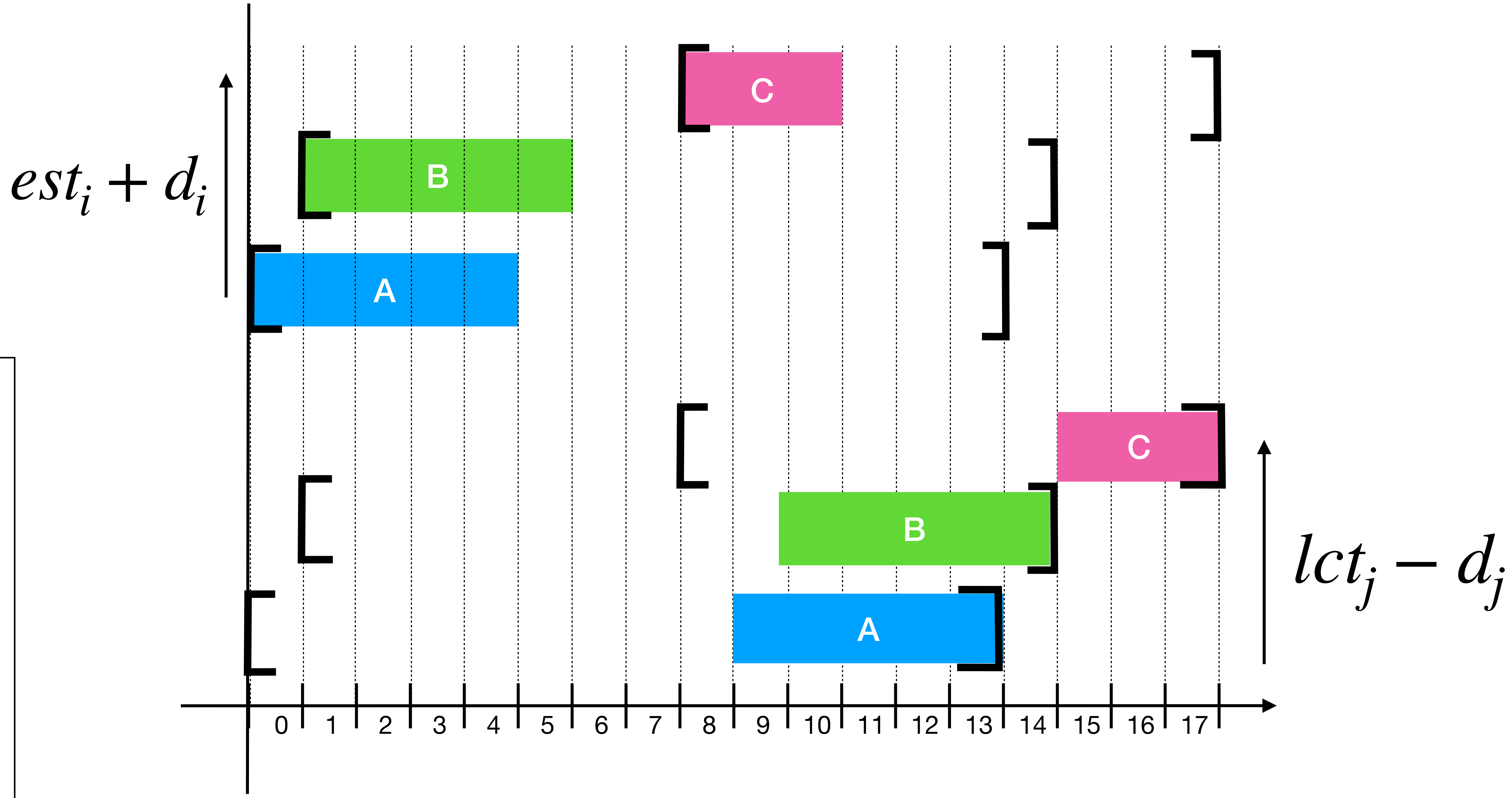


Detectable precedence filtering with Θ-Tree, an example

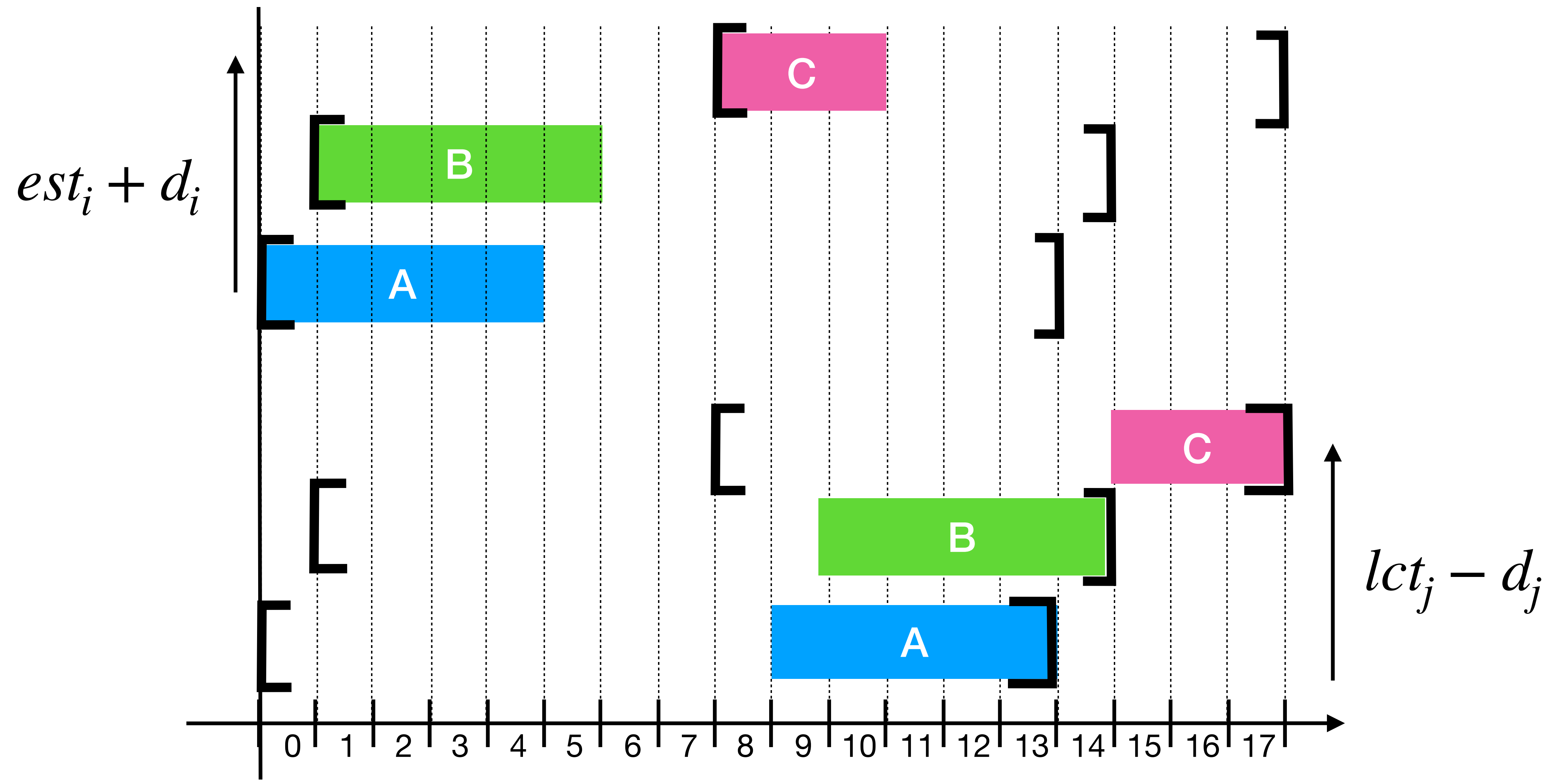



```
DetectablePrecedence(T={1..n}) {
  Tlst ← sortAZ([1..n],sortKey = lct-d) // [A,B,C]
  Tect ← sortAZ([1..n],sortKey = est+d) // [A,B,C]
  ite ← iterator(Tlst)
  j ← ite.next() // candidate precedence of i
  Θ ← Θ-Tree.init({1..n})
  for (i ← Tect) {
    while (esti+di > lctj-dj) {
       Θ.insert(j)
       if (ite.hasNext()) {j ← ite.next()} else {break}
    }
    est'i ← max(esti, ectΘ\i)
  }
  esti ← est'i, ∀i∈T
}
```

Sorting

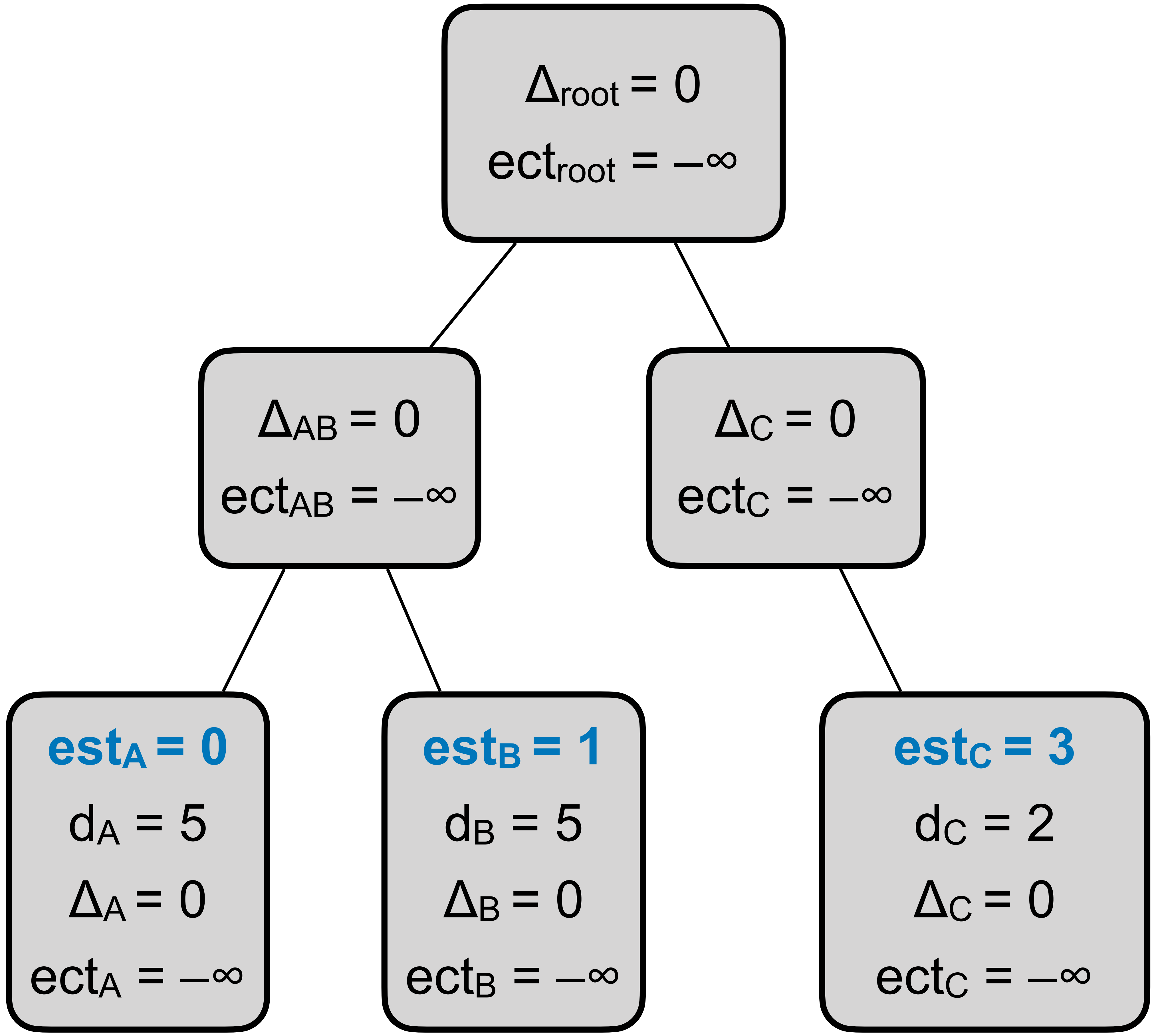


Detectable precedence filtering with Θ-Tree, an example

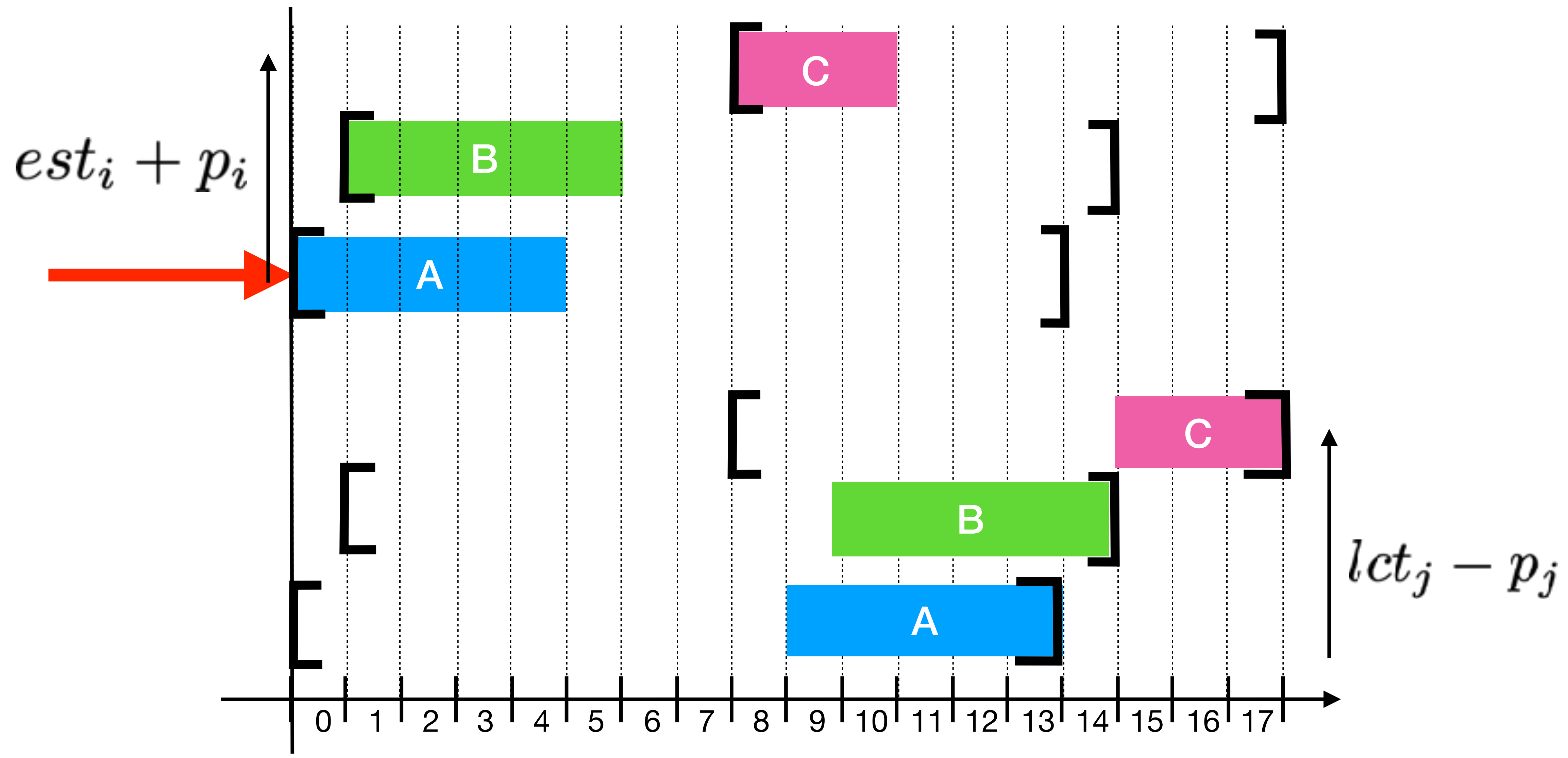



```
DetectablePrecedence(T={1..n}) {
  Tlst ← sortAZ([1..n],sortKey = lct-d) // [A, B, C]
  Tect ← sortAZ([1..n],sortKey = est+d) // [A, B, C]
  ite ← iterator(Tlst)
  j ← ite.next() // candidate precedence of i
  Θ ← Θ-Tree.init({1..n})
  for (i ← Tect) {
    while (esti+di > lctj-dj) {
       Θ.insert(j)
       if (ite.hasNext()) {j ← ite.next()} else {break}
    }
    est'i ← max(esti, ectΘ\i)
  }
  esti ← est'i, ∀i∈T
}
```

Θ-Tree initialiation

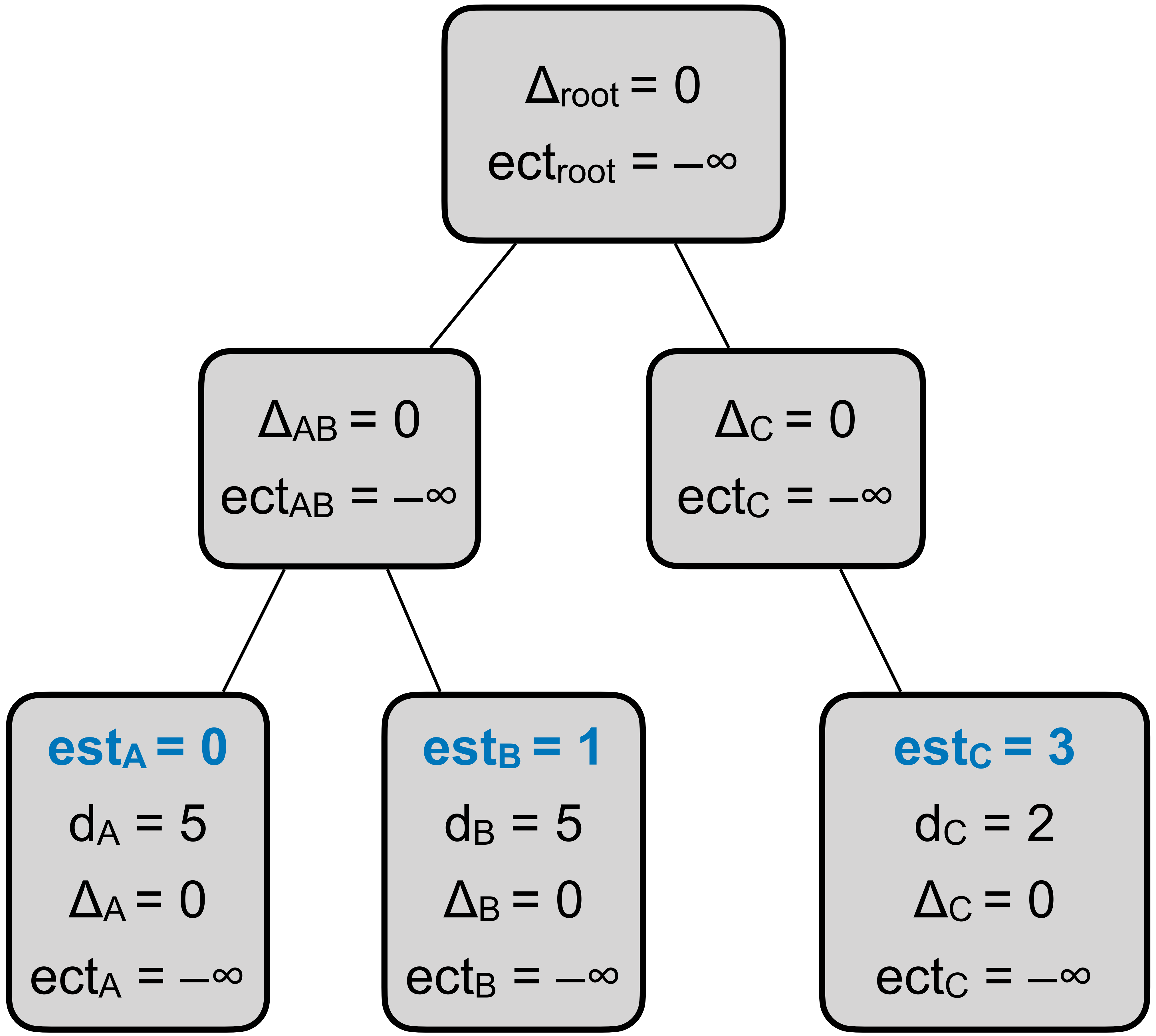


Detectable precedence filtering with Θ-Tree, an example

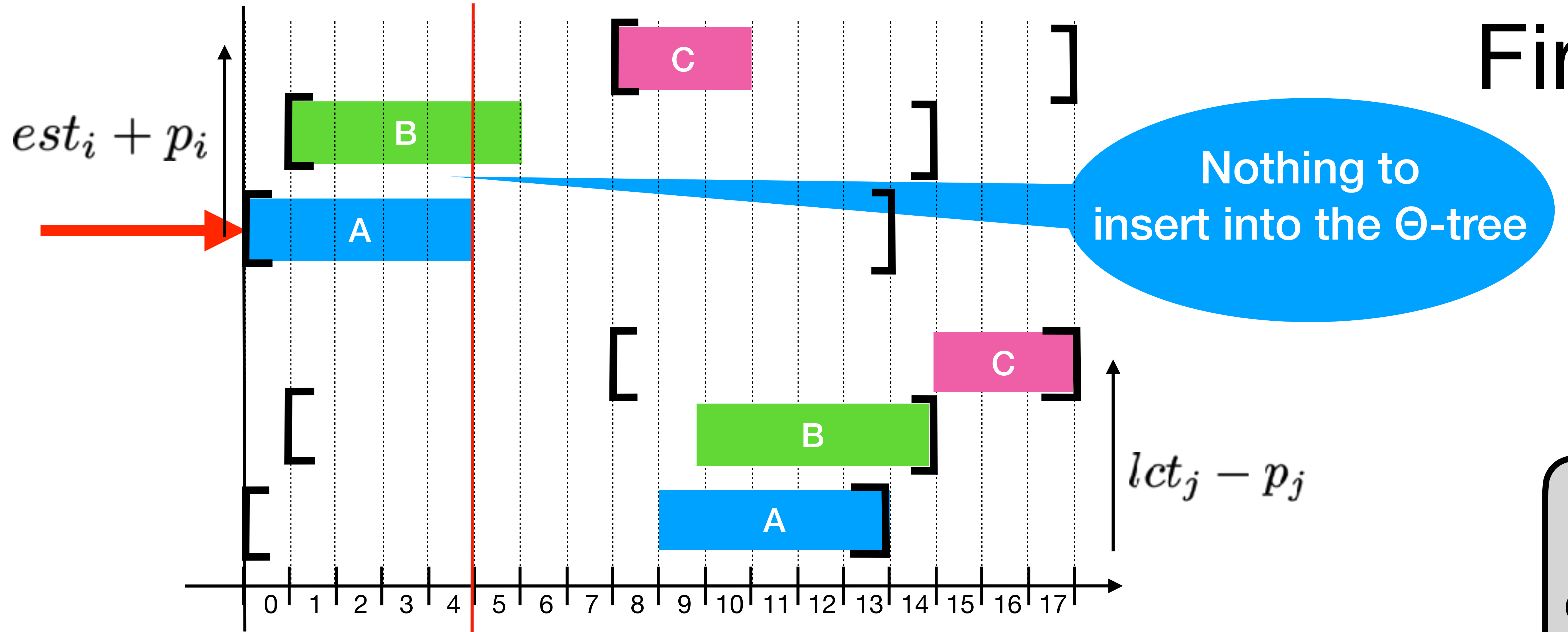



```
DetectablePrecedence(T={1..n}) {
  T_lst ← sortAZ([1..n],sortKey = lct-d) // [A, B, C]
  T_ect ← sortAZ([1..n],sortKey = est+d) // [A, B, C]
  ite ← iterator(T_lst)
  j ← ite.next() // candidate precedence of i
  Θ ← Θ-Tree.init({1..n})
  for (i ← T_ect) { // i ← A
    while (est_i+d_i > lct_j-d_j) {
       Θ.insert(j)
       if (ite.hasNext()) {j ← ite.next()} else {break}
    }
    est'_i ← max(est_i, ect_Θ\i)
  }
  est_i ← est'_i, ∀i∈T
}
```

First iteration: A is considered

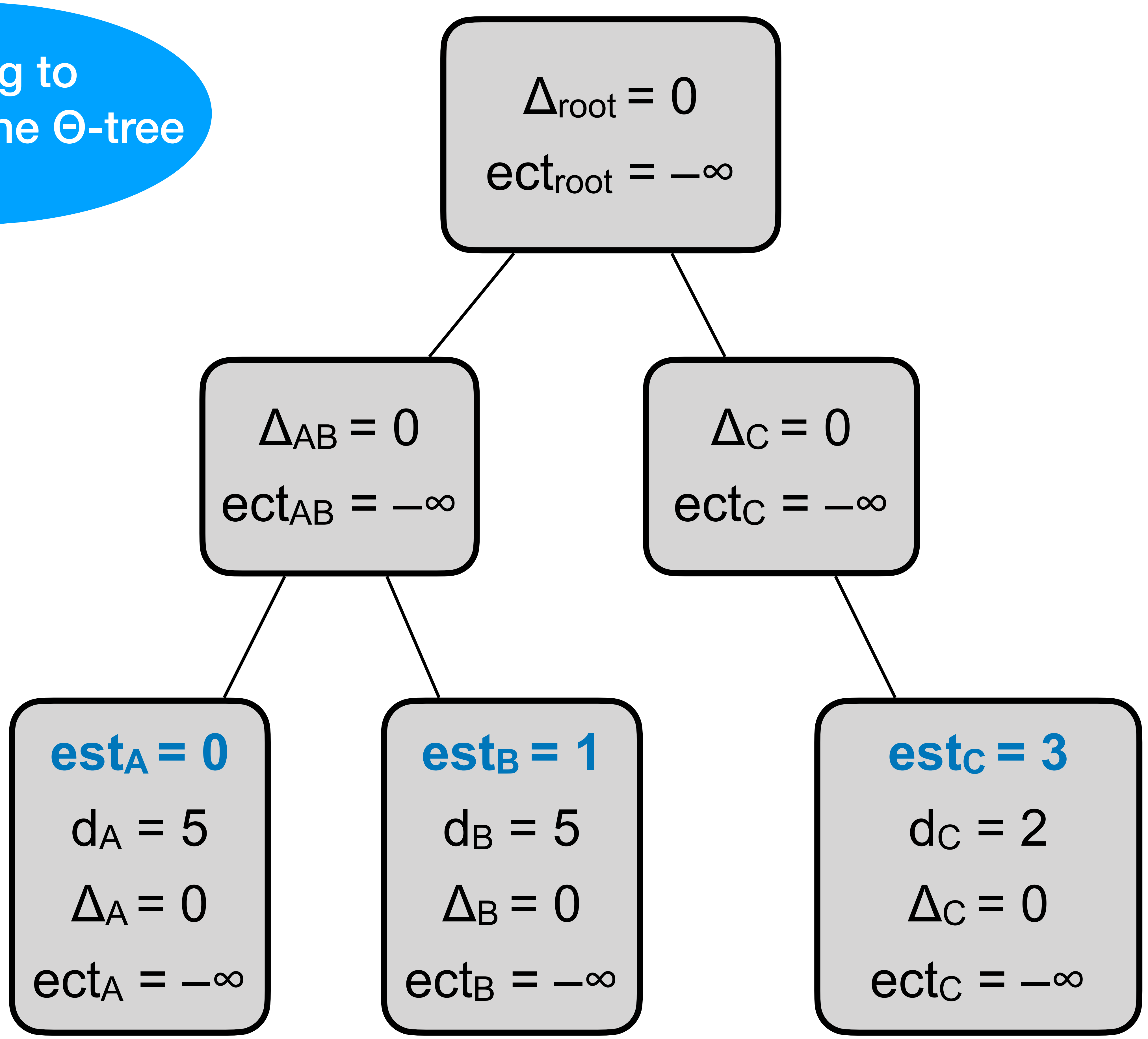


Detectable precedence filtering with Θ-Tree, an example

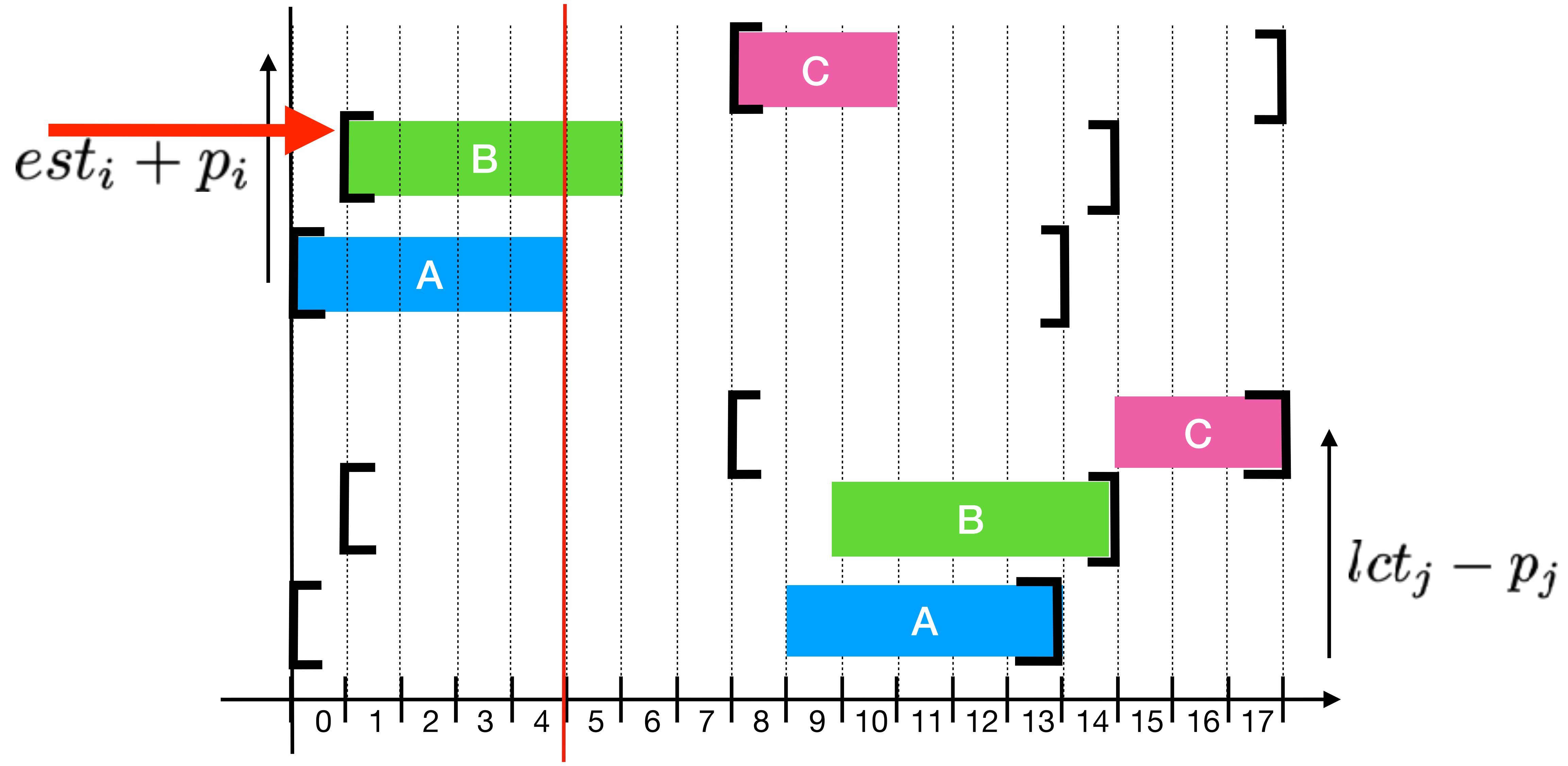



```
DetectablePrecedence(T={1..n}) {
  Tlst ← sortAZ([1..n],sortKey = lct-d) // [A, B, C]
  Tect ← sortAZ([1..n],sortKey = est+d) // [A, B, C]
  ite ← iterator(Tlst)
  j ← ite.next() // candidate precedence of i
  Θ ← Θ-Tree.init({1..n})
  for (i ← Tect) { // i ← A
    while (esti+di > lctj-dj) {
       Θ.insert(j)
       if (ite.hasNext()) {j ← ite.next()} else {break}
    }
    est'i ← max(esti, ectΘ\i)
  }
  esti ← est'i, ∀i∈T
}
```

First iteration: A is considered

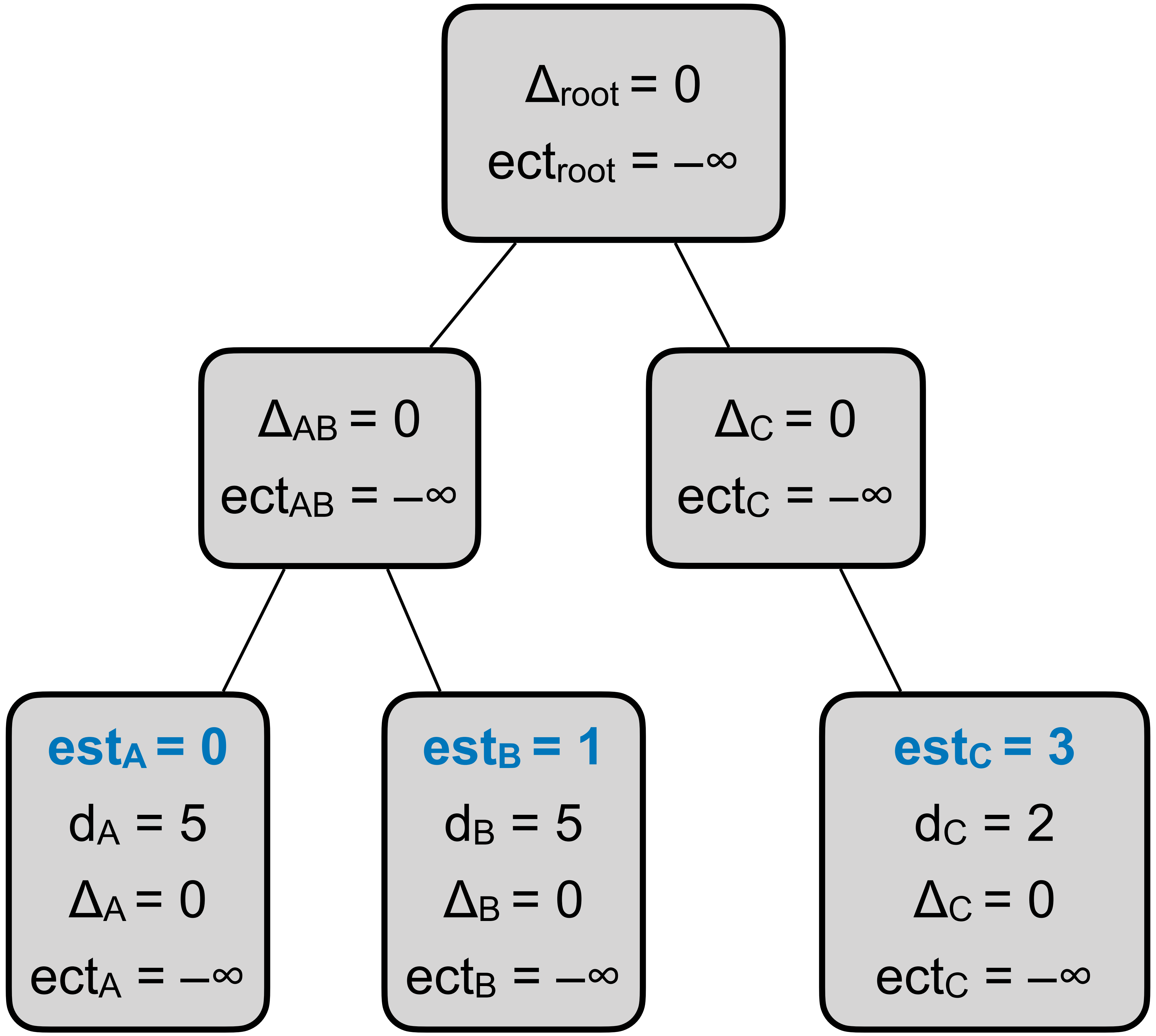


Detectable precedence filtering with Θ-Tree, an example

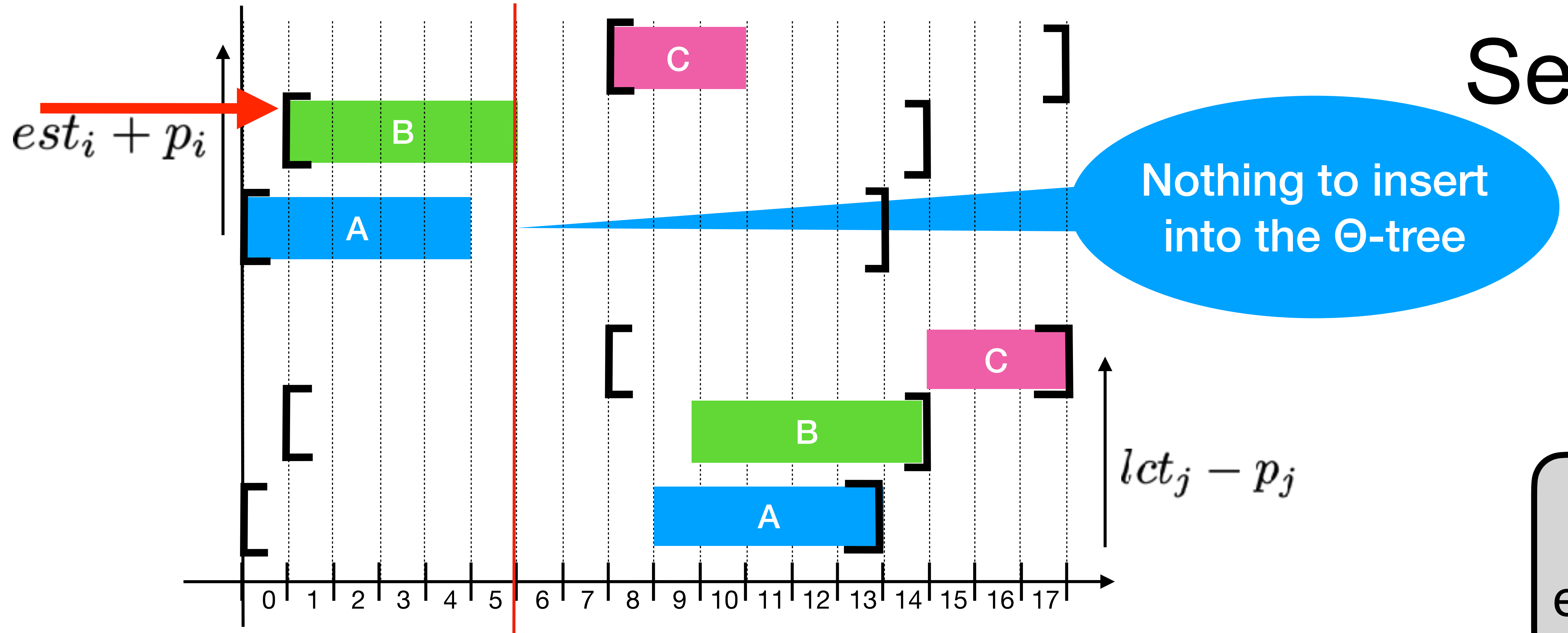



```
DetectablePrecedence(T={1..n}) {
  T_lst ← sortAZ([1..n],sortKey = lct-d) // [A, B, C]
  T_ect ← sortAZ([1..n],sortKey = est+d) // [A, B, C]
  ite ← iterator(T_lst)
  j ← ite.next() // candidate precedence of i
  Θ ← Θ-Tree.init({1..n})
  for (i ← T_ect) { // i ← B
    while (est_i+d_i > lct_j-d_j) {
       Θ.insert(j)
       if (ite.hasNext()) {j ← ite.next()} else {break}
    }
    est'_i ← max(est_i, ect_Θ\i)
  }
  est_i ← est'_i, ∀i∈T
}
```

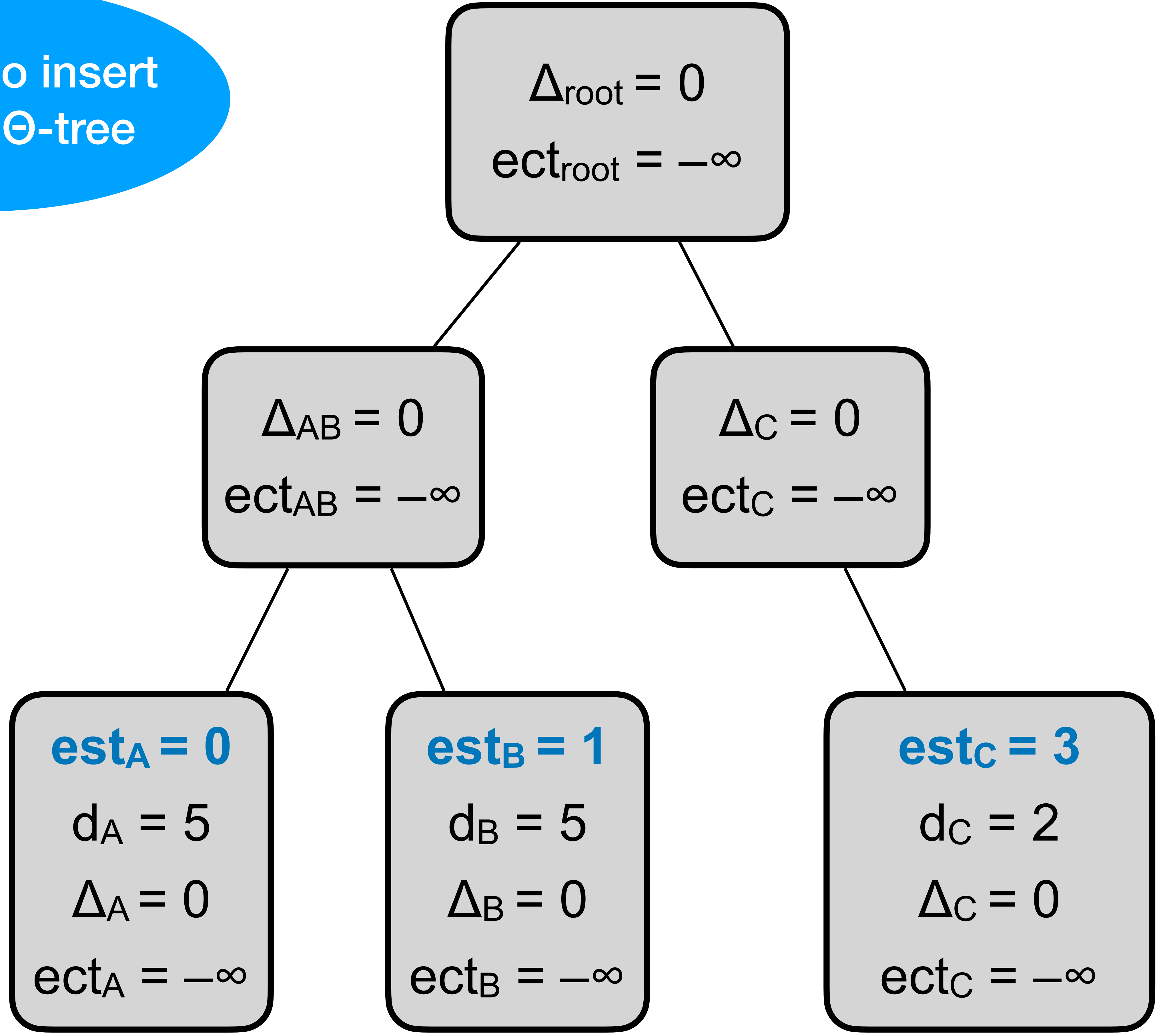
Second iteration: B is considered



Detectable precedence filtering with Θ-Tree, an example

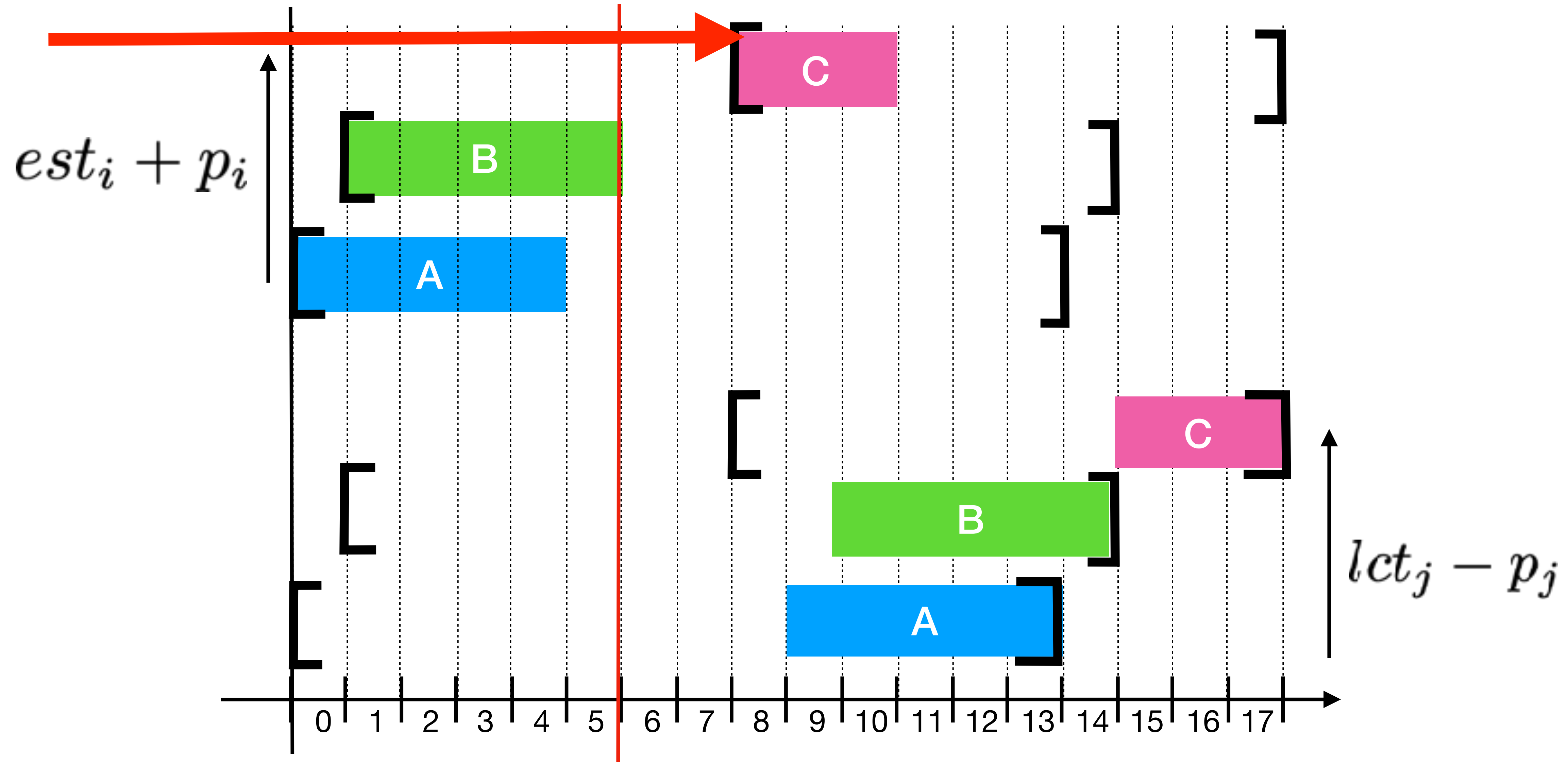


Second iteration: B is considered



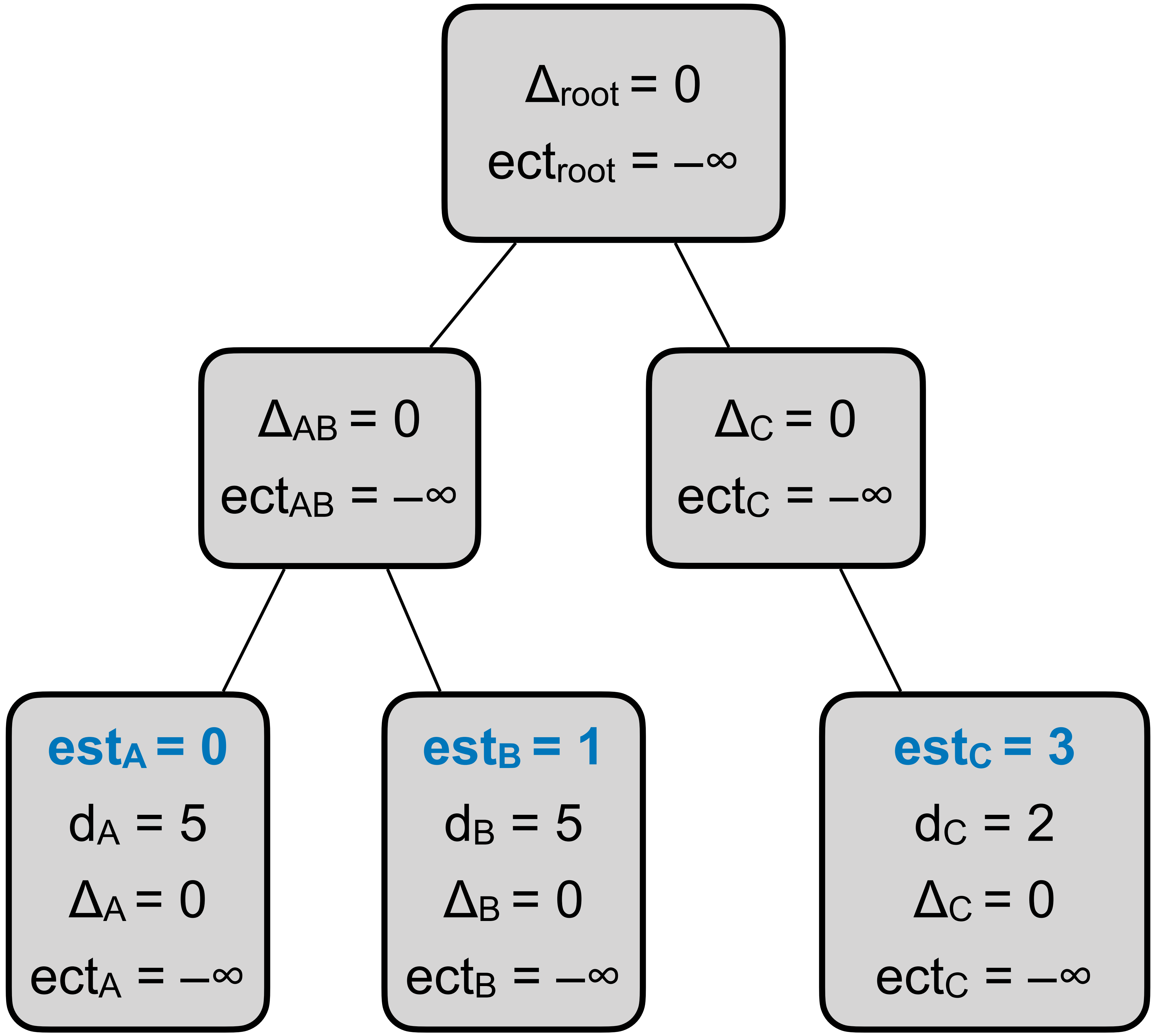

```
DetectablePrecedence(T={1..n}) {
  Tlst ← sortAZ([1..n],sortKey = lct-d) // [A, B, C]
  Tect ← sortAZ([1..n],sortKey = est+d) // [A, B, C]
  ite ← iterator(Tlst)
  j ← ite.next() // candidate precedence of i
  Θ ← Θ-Tree.init({1..n})
  for (i ← Tect) { // i ← B
    while (esti+di > lctj-dj) {
       Θ.insert(j)
       if (ite.hasNext()) {j ← ite.next()} else {break}
    }
    est'i ← max(esti, ectΘ\i)
  }
  esti ← est'i, ∀i∈T
}
```

Detectable precedence filtering with Θ-Tree, an example

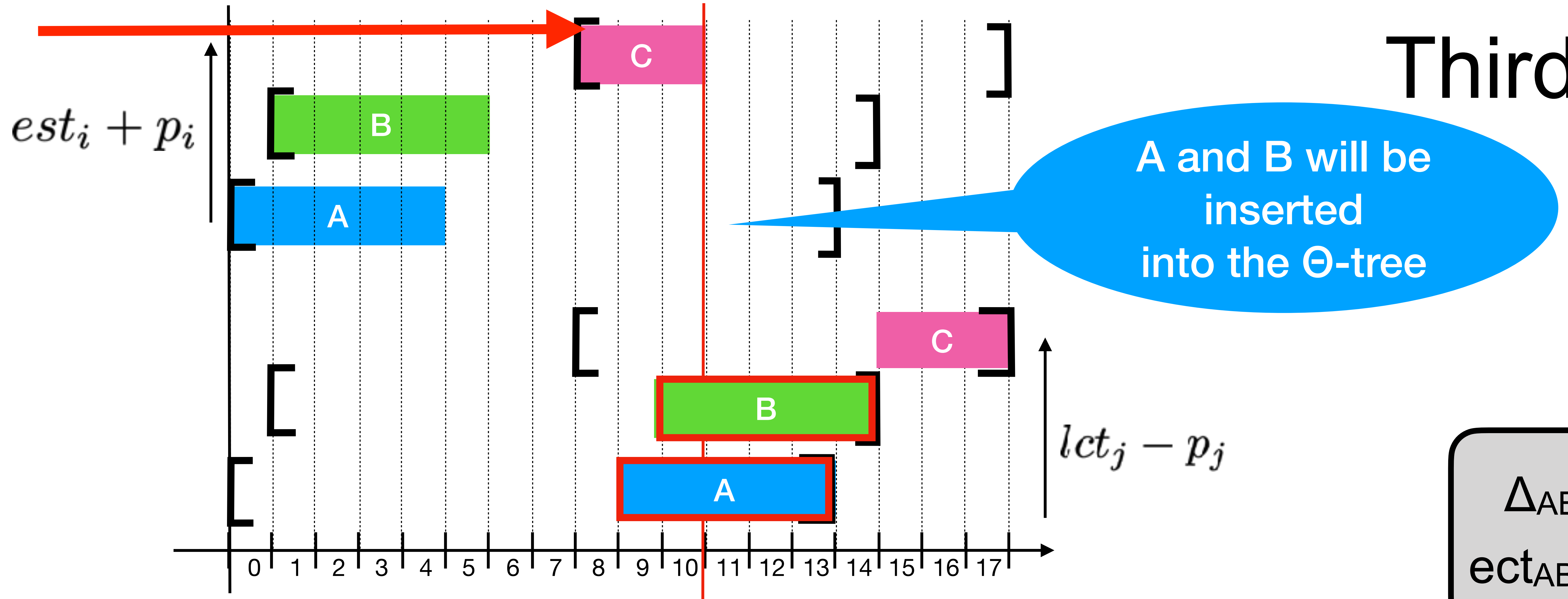



```
DetectablePrecedence(T={1..n}) {
  Tlst ← sortAZ([1..n],sortKey = lct-d) // [A, B, C]
  Tect ← sortAZ([1..n],sortKey = est+d) // [A, B, C]
  ite ← iterator(Tlst)
  j ← ite.next() // candidate precedence of i
  Θ ← Θ-Tree.init({1..n})
  for (i ← Tect) { // i ← C
    while (esti+di > lctj-dj) {
       Θ.insert(j)
       if (ite.hasNext()) {j ← ite.next()} else {break}
    }
    est'i ← max(esti, ectΘ\i)
  }
  esti ← est'i, ∀i∈T
}
```

Third iteration: C is considered

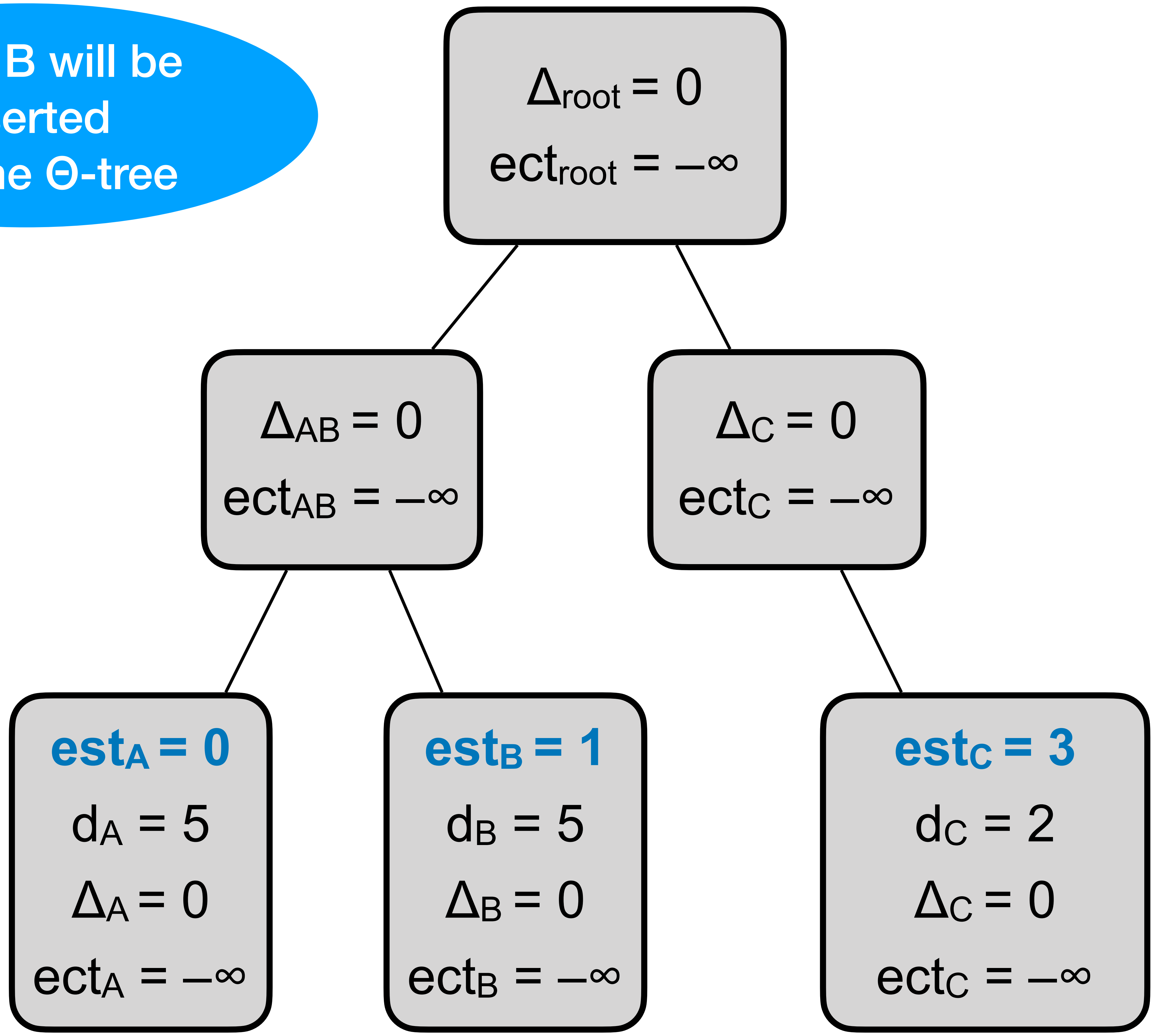


Detectable precedence filtering with Θ-Tree, an example

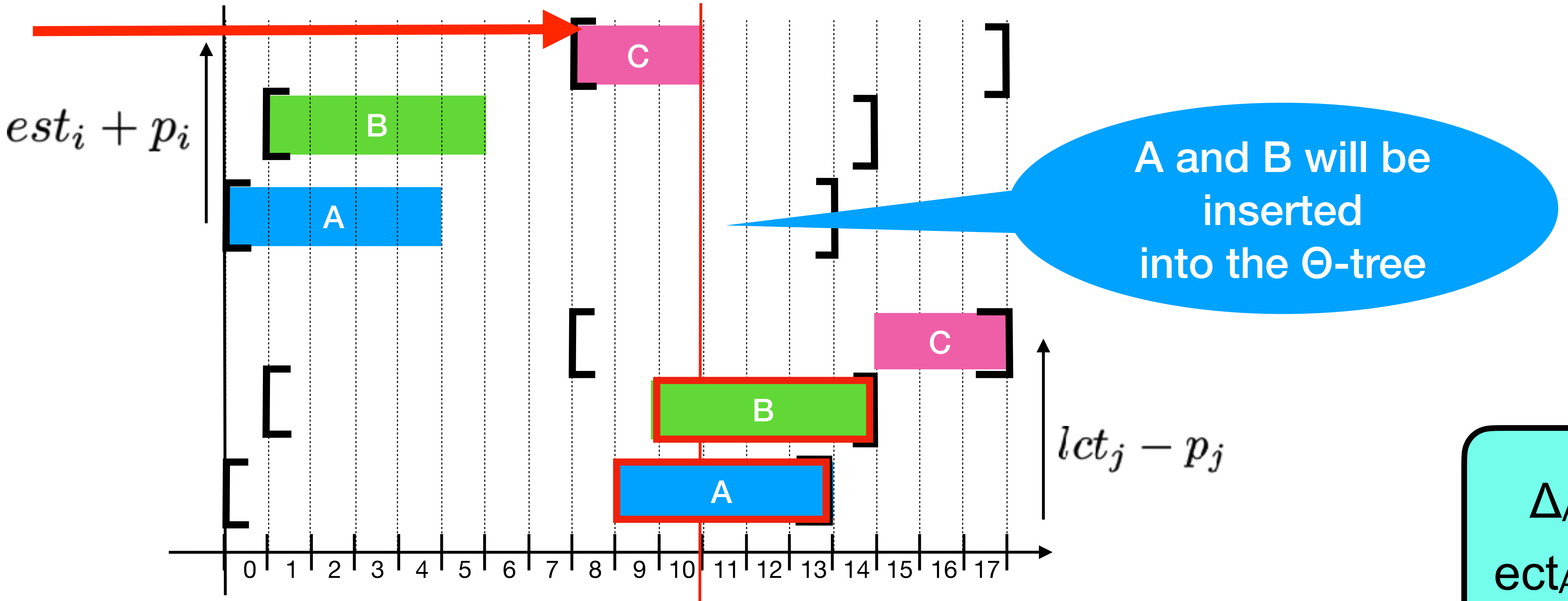



```
DetectablePrecedence(T={1..n}) {
  T_lst ← sortAZ([1..n],sortKey = lct-d) // [A, B, C]
  T_ect ← sortAZ([1..n],sortKey = est+d) // [A, B, C]
  ite ← iterator(T_lst)
  j ← ite.next() // candidate precedence of i
  Θ ← Θ-Tree.init({1..n})
  for (i ← T_ect) { // i ← C
    while (est_i+d_i > lct_j-d_j) {
       Θ.insert(j)
       if (ite.hasNext()) {j ← ite.next()} else {break}
    }
    est'_i ← max(est_i, ect_Θ\i)
  }
  est_i ← est'_i, ∀i∈T
}
```

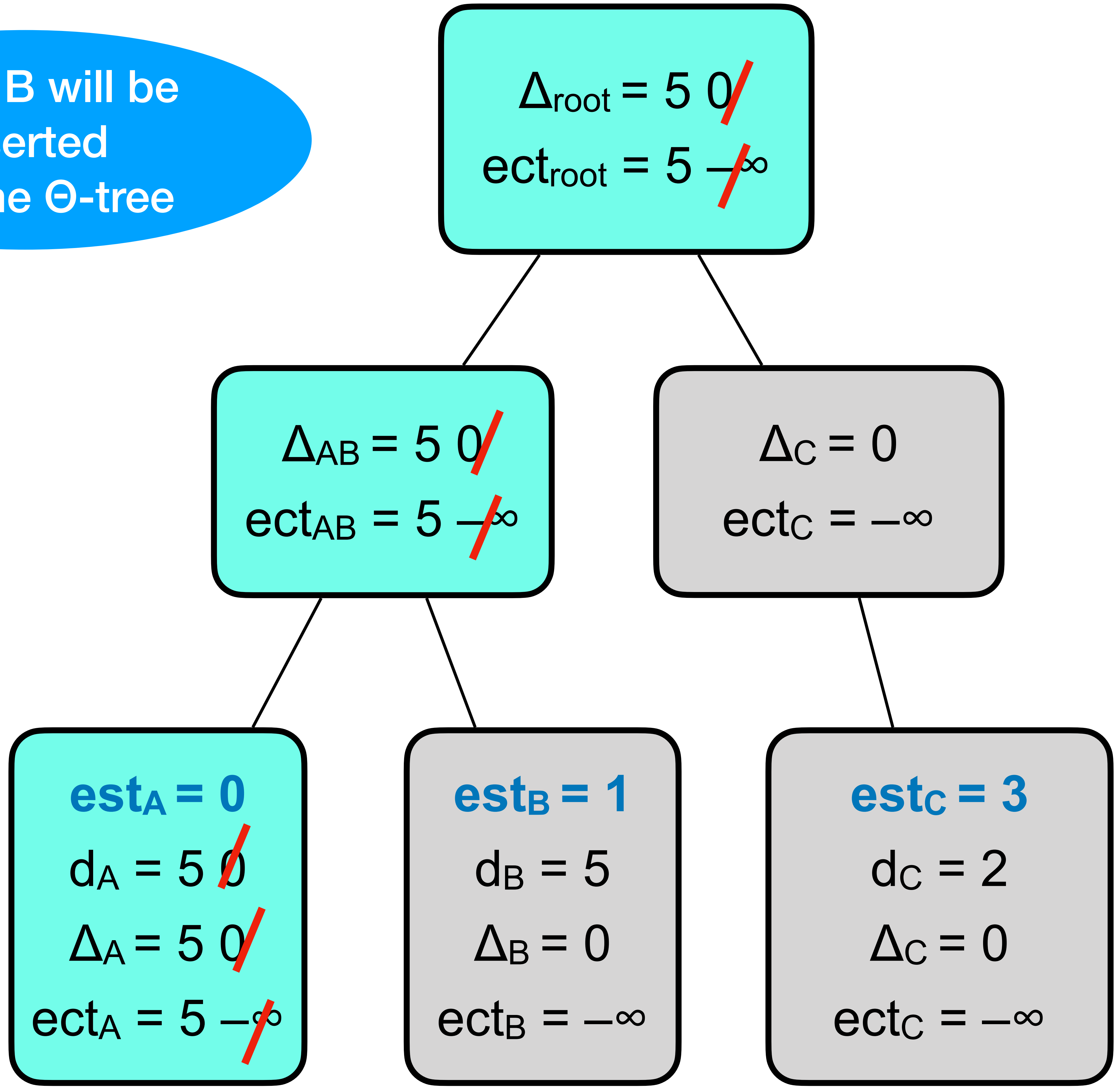
Third iteration: C is considered



Detectable precedence filtering with Θ-Tree, an example

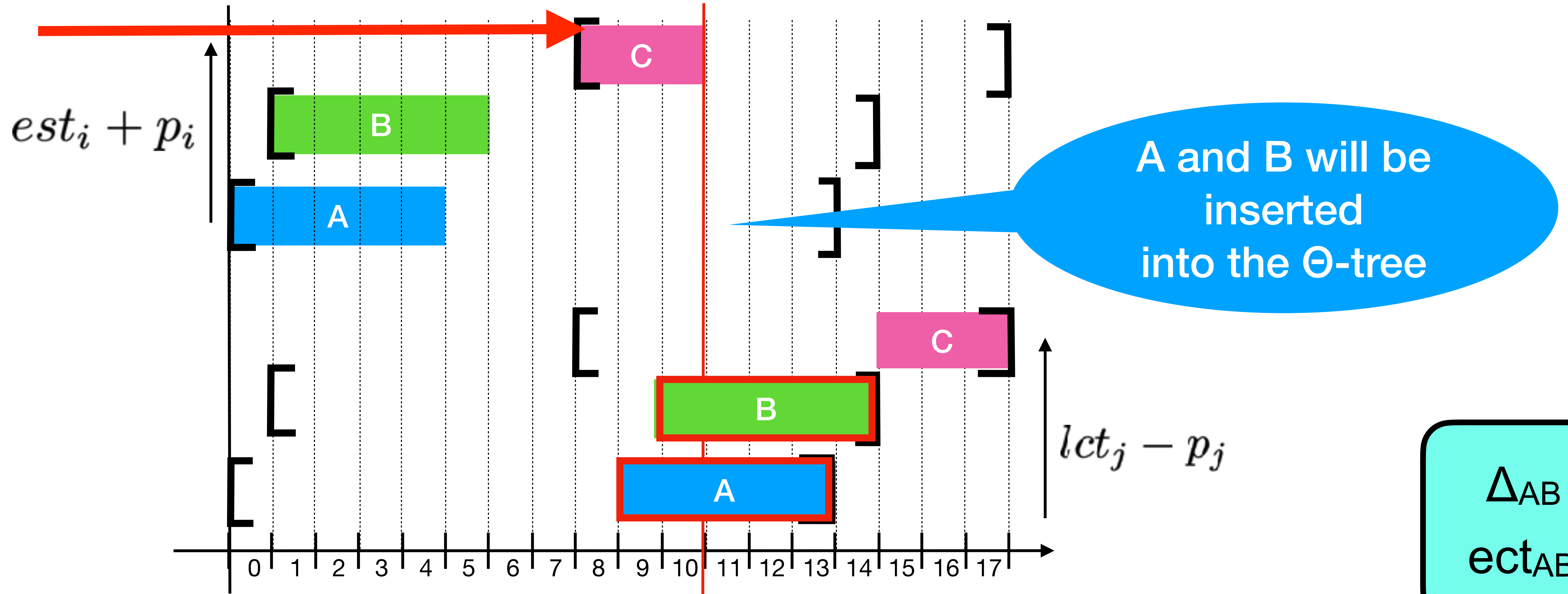


Insertion of A



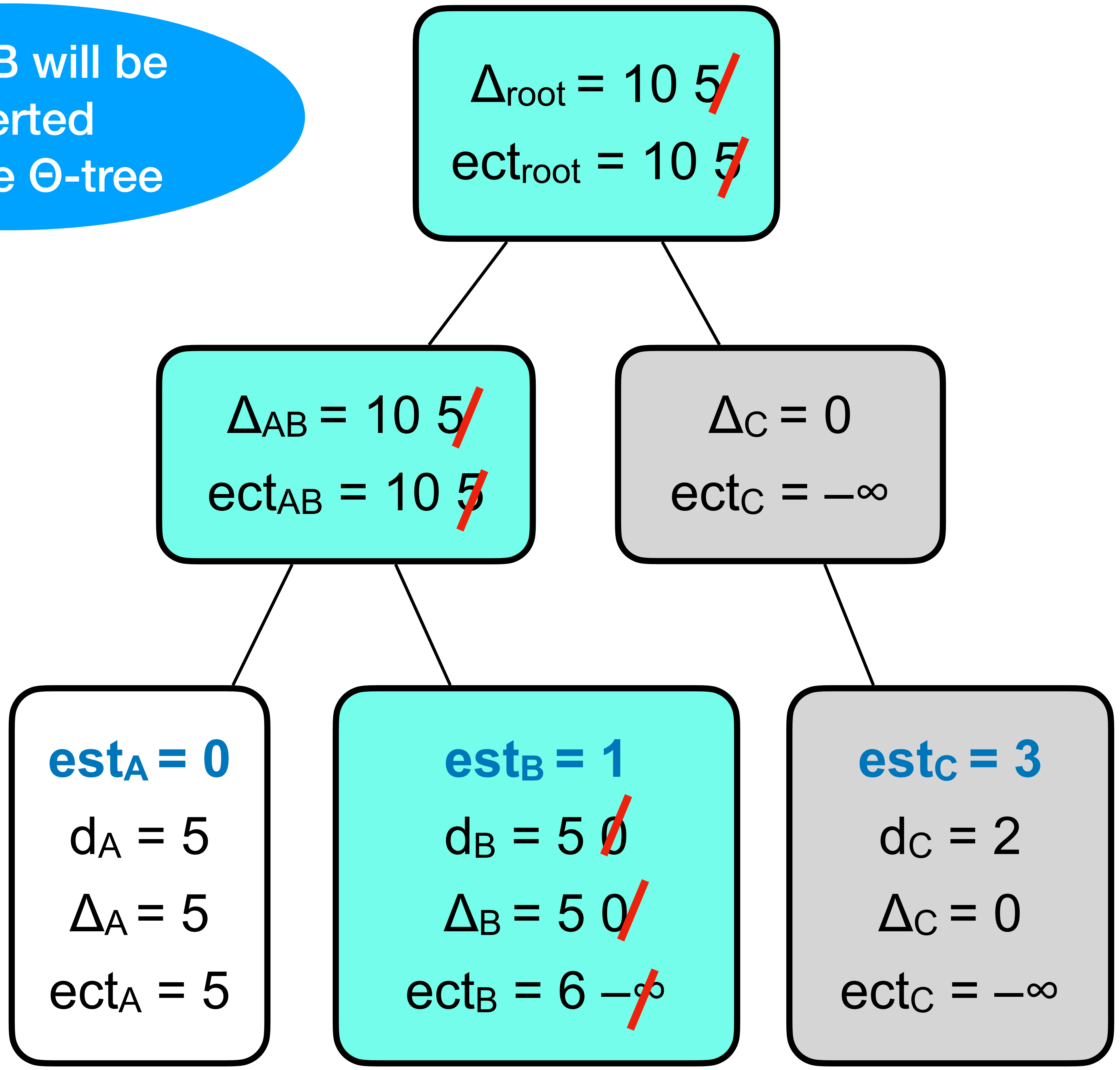

```
DetectablePrecedence(T={1..n}) {
  Tlst ← sortAZ([1..n],sortKey = lct-d) // [A, B, C]
  Tect ← sortAZ([1..n],sortKey = est+d) // [A, B, C]
  ite ← iterator(Tlst)
  j ← ite.next() // candidate precedence of i
  Θ ← Θ-Tree.init({1..n})
  for (i ← Tect) { // i ← C
    while (esti+di > lctj-dj) {
      Θ.insert(j)
      if (ite.hasNext()) {j ← ite.next()} else {break}
    }
    est'i ← max(esti, ectΘ\i)
  }
  esti ← est'i, ∀i∈T
}
```

Detectable precedence filtering with Θ-Tree, an example

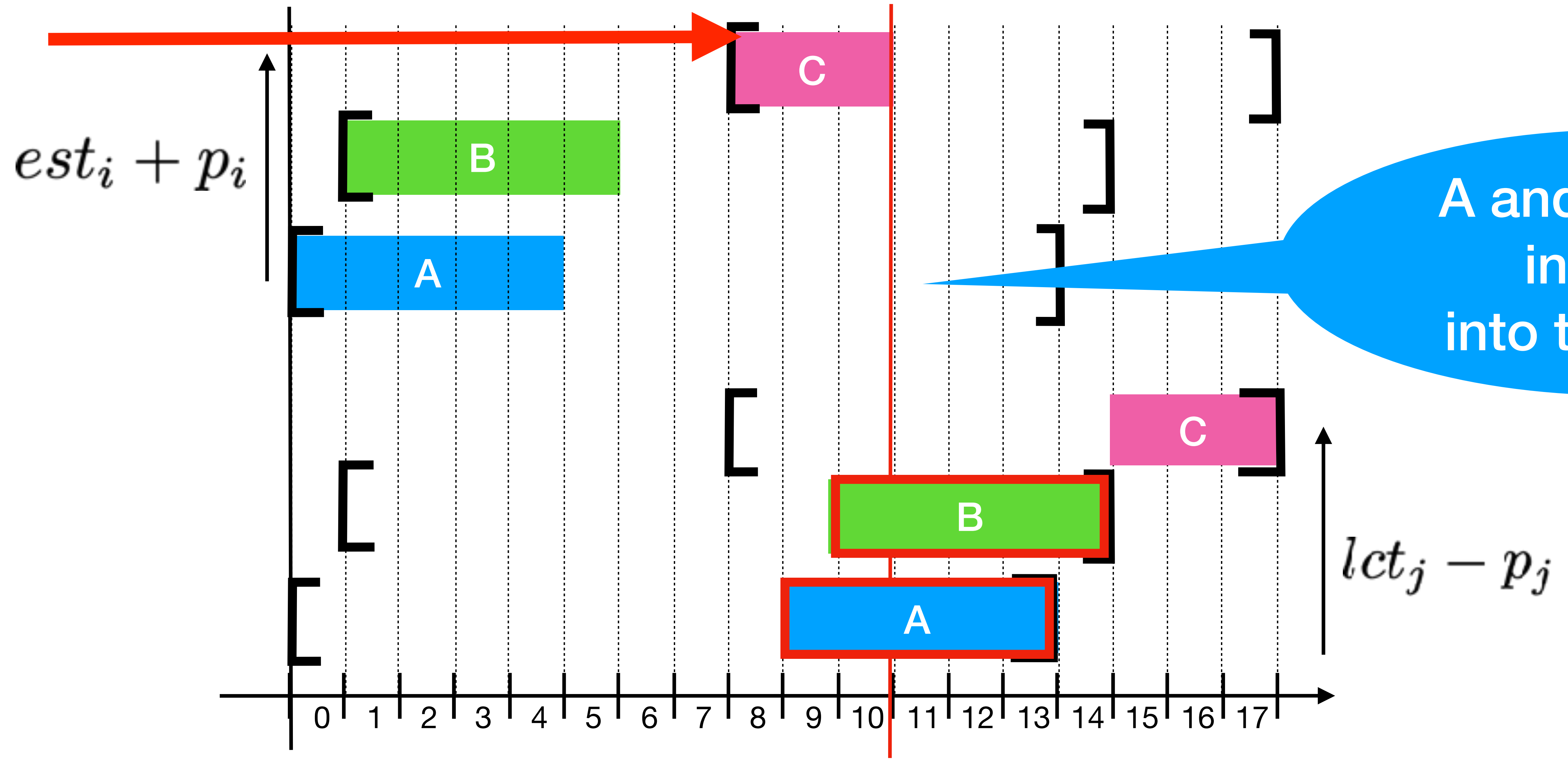



```
DetectablePrecedence(T={1..n}) {
  Tlst ← sortAZ([1..n],sortKey = lct-d) // [A, B, C]
  Tect ← sortAZ([1..n],sortKey = est+d) // [A, B, C]
  ite ← iterator(Tlst)
  j ← ite.next() // candidate precedence of i
  Θ ← Θ-Tree.init({1..n})
  for (i ← Tect) { // i ← C
    while (esti+di > lctj-dj) {
       Θ.insert(j)
       if (ite.hasNext()) {j ← ite.next()} else {break}
    }
    est'i ← max(esti, ectΘ\i)
  }
  esti ← est'i, ∀i∈T
}
```

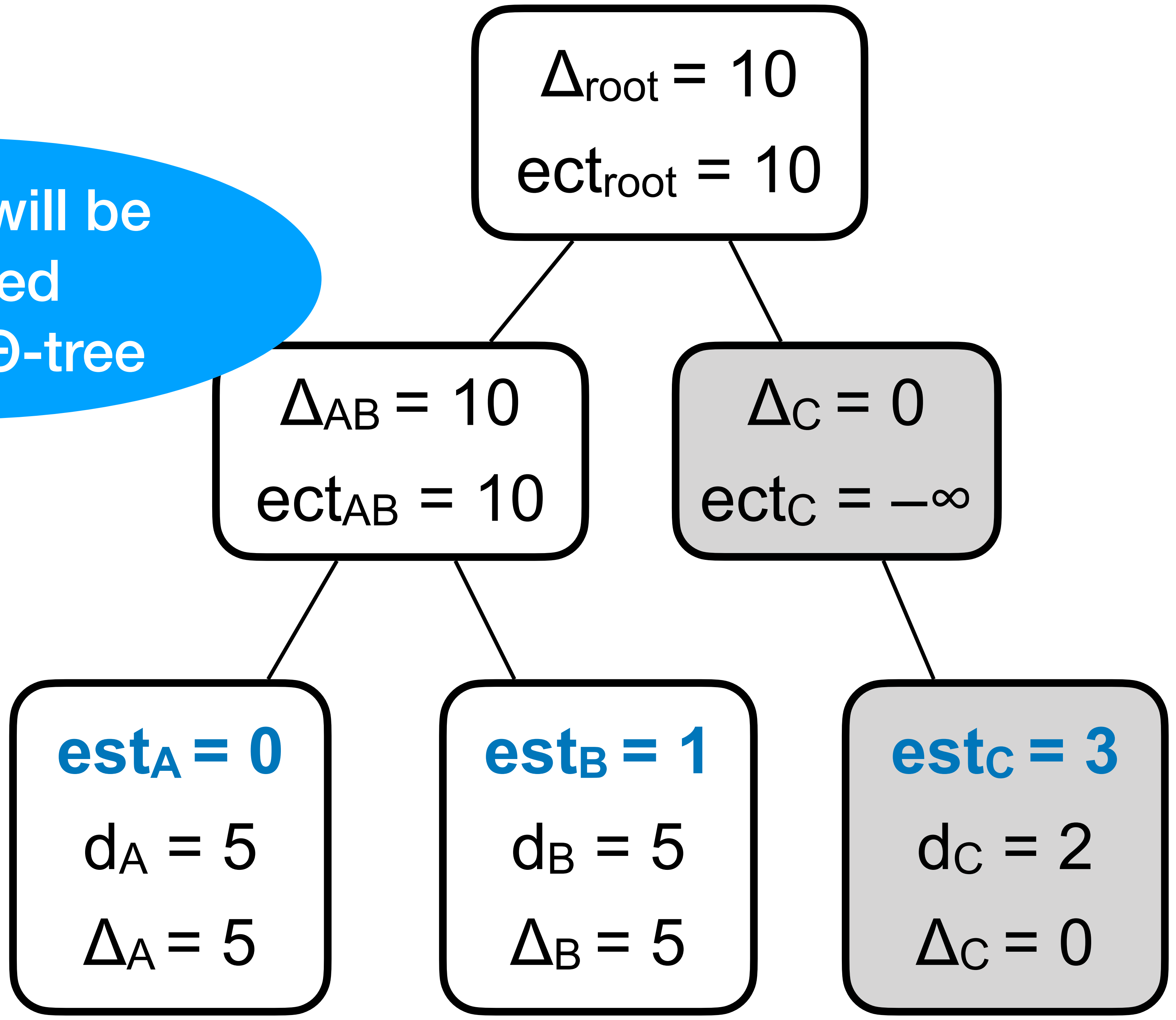
Insertion of B



Detectable precedence filtering with Θ-Tree, an example

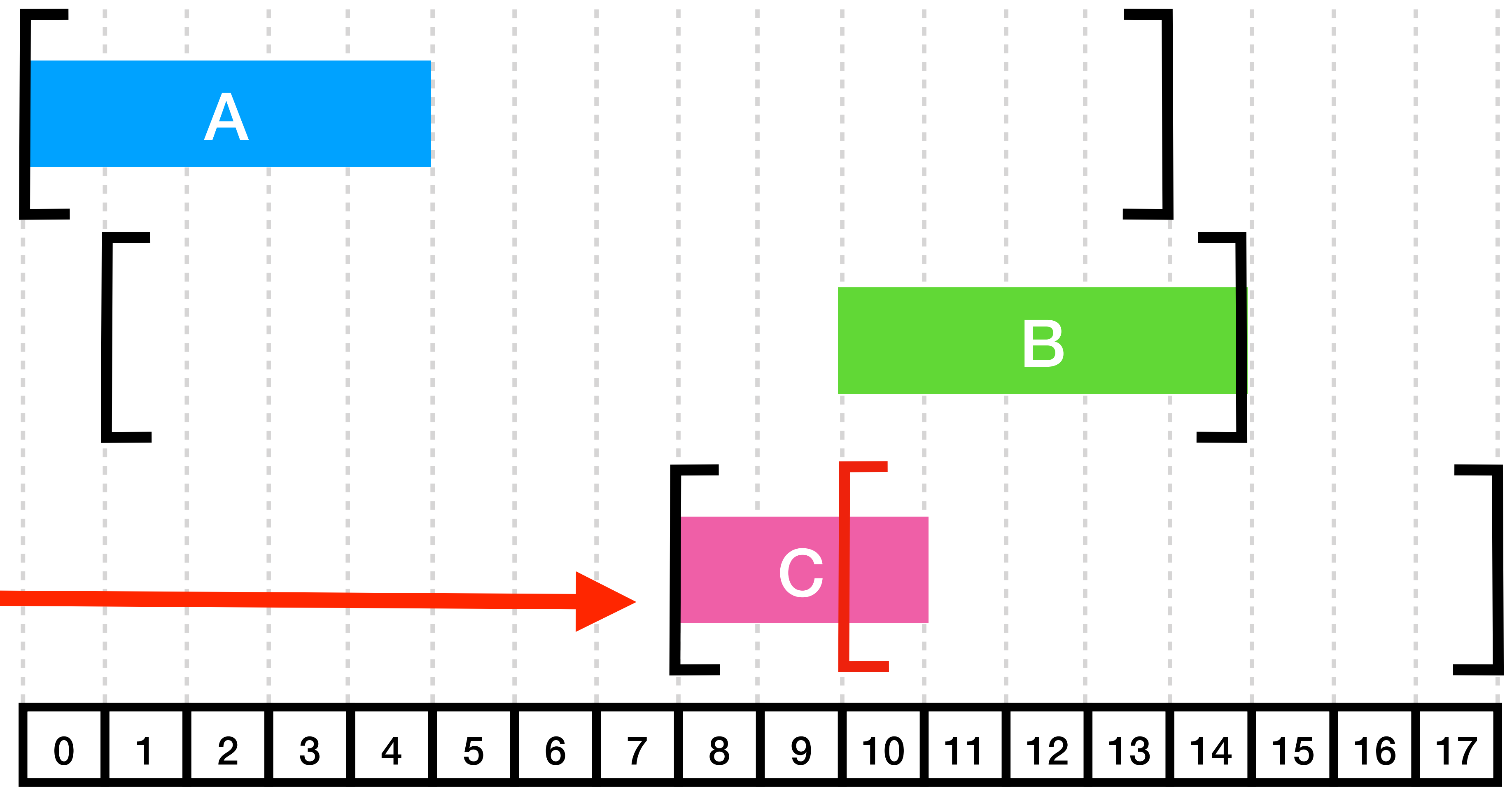


A and B will be inserted into the Θ-tree




```
DetectablePrecedence(T={1..n}) {
  Tlst ← sortAZ([1..n],sortKey = lct-d) // [A, B, C]
  Tect ← sortAZ([1..n],sortKey = est+d) // [A, B, C]
  ite ← iterator(Tlst)
  j ← ite.next() // candidate precedence of i
  Θ ← Θ-Tree.init({1..n})
  for (i ← Tect) { // i ← C
    while (esti+di > lctj-dj) {
       Θ.insert(j)
       if (ite.hasNext()) {j ← ite.next()} else {break}
    }
    est'i ← max(esti, ectΘ\i)
  }
  esti ← est'i, ∀i∈T
}
```

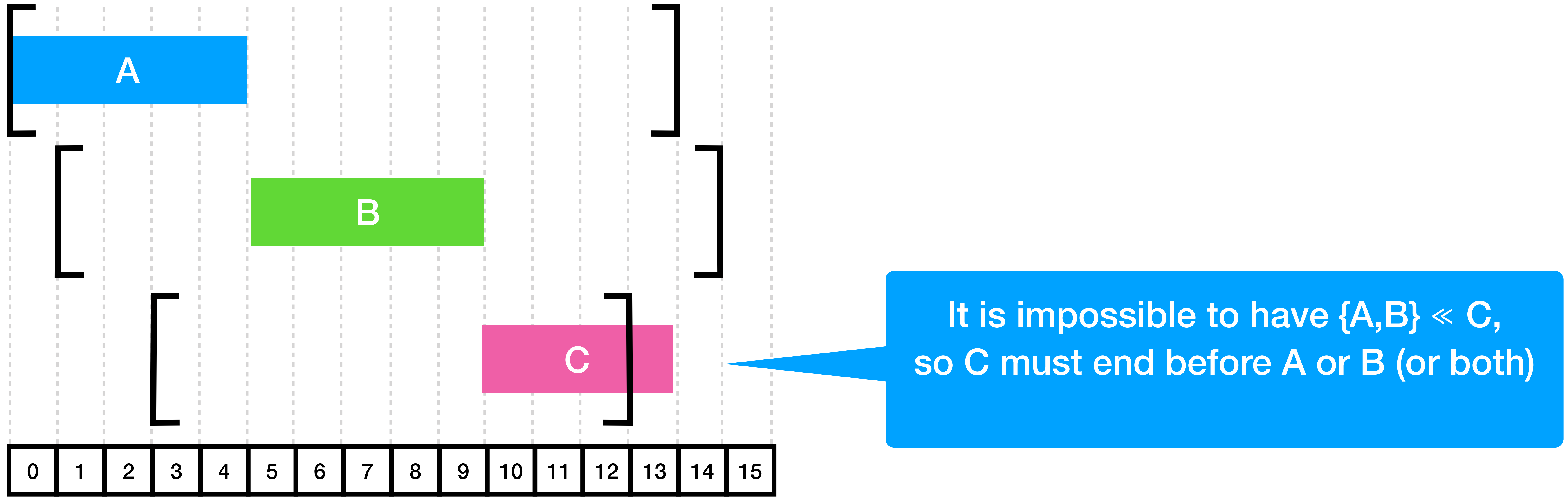
$est_C = est'_C = 10$



Not-Last

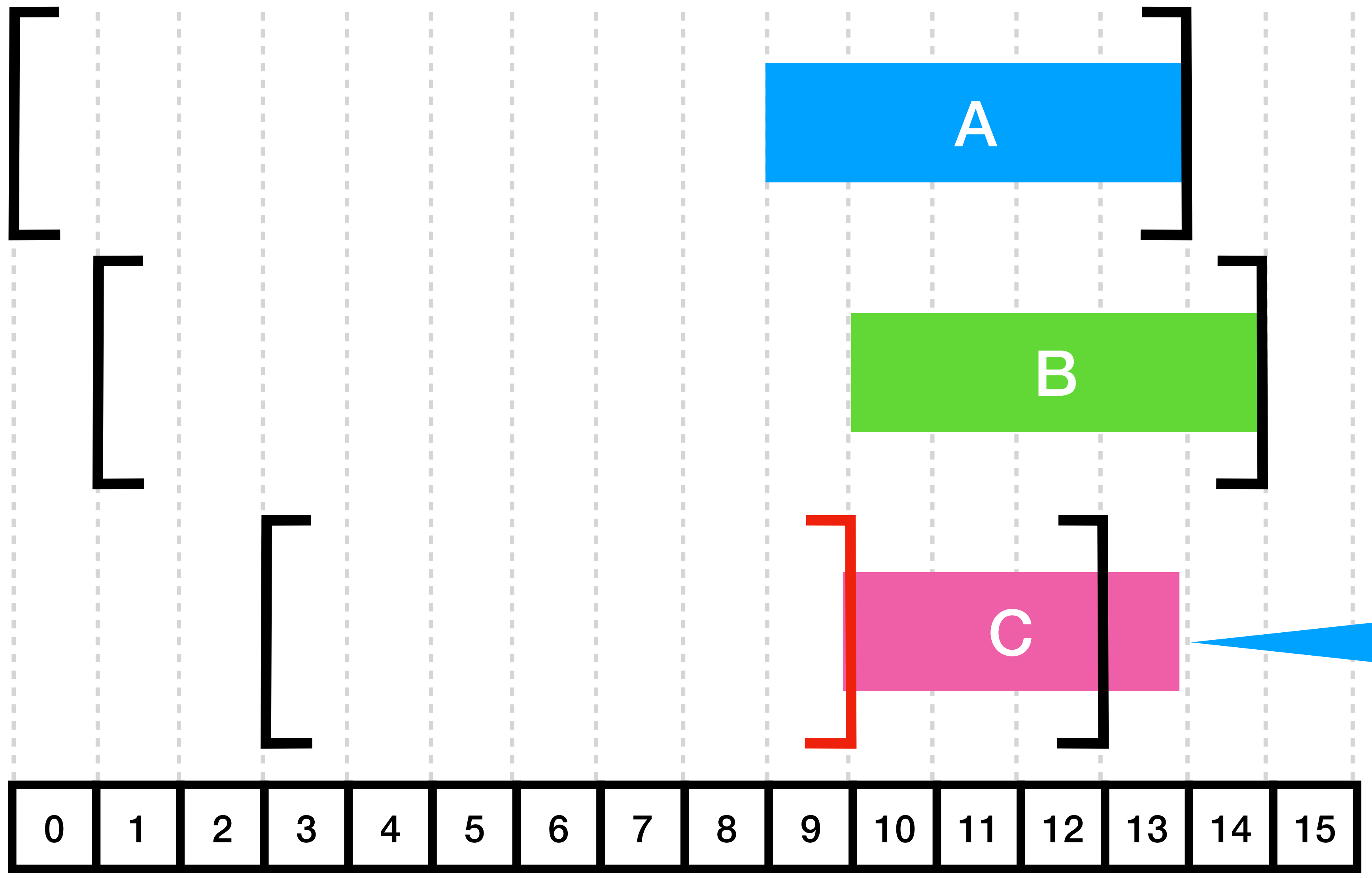
Not-Last = another filtering rule

- Activity C cannot be scheduled after (A and B):



Not-Last = another filtering rule

- Activity C cannot be scheduled after (A and B)



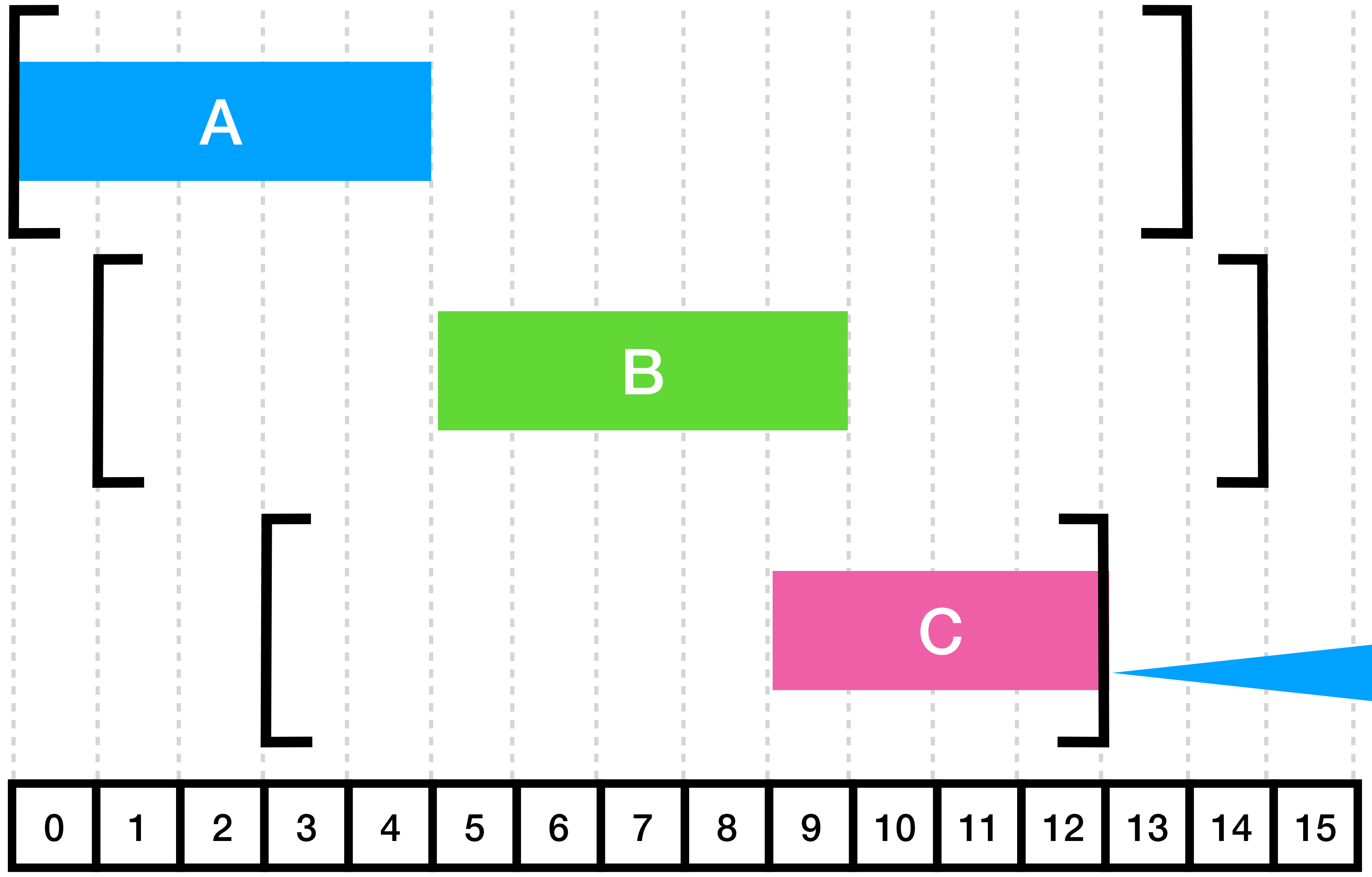
Take the minimum of the two cases:
$lct_C \leftarrow \min(lct_C, \max\{lct_B - d_B, lct_A - d_A\})$.

Not-Last filtering formally defined

- $\forall \Omega \subset T$ non-empty strict subset of T, $\forall i \in T \setminus \Omega$:

  $$est_\Omega + d_\Omega > lct_i - d_i \;\rightsquigarrow\; lct_i \leftarrow \min(lct_i\,,\ \max\,\{lct_j - d_j \mid j \in \Omega\}) \qquad \text{(NL)}$$

- Example: For $\Omega = \{A,B\}$, activity $i = C$ cannot start last:



It is impossible to have $\{A,B\} \ll C$,
so C must end before A or B (or both):
$lct_C \leftarrow \min(lct_C\,,\ \max\{lct_B - d_B,\ lct_A - d_A\})$.

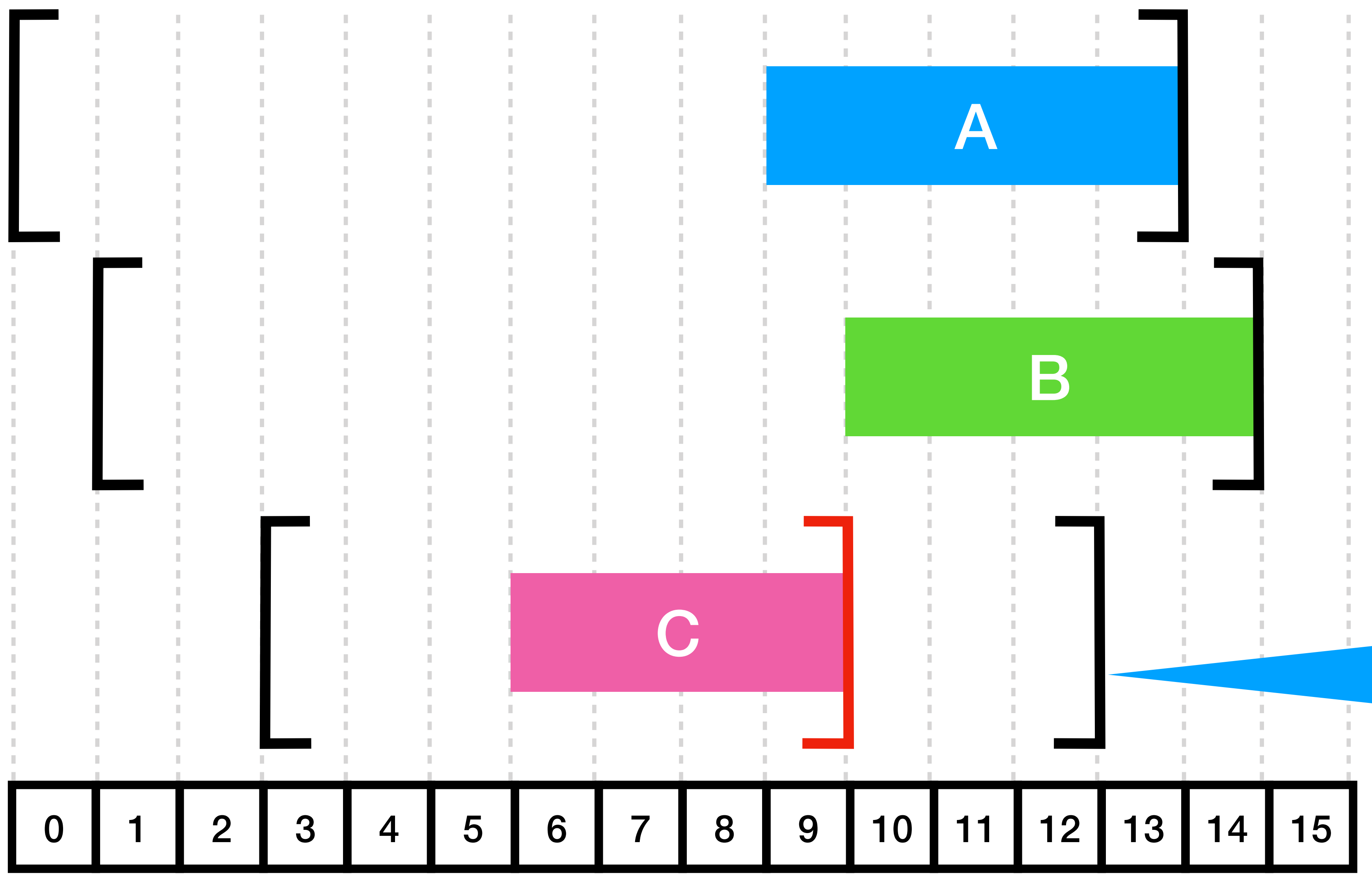
- Again, we need to find a way to enumerate the Ω in a nested way.

Not-Last filtering formally defined

- $\forall \Omega \subset T$ non-empty strict subset of T, $\forall i \in T \setminus \Omega$:

  $est_\Omega + d_\Omega > lct_i - d_i \;\rightsquigarrow\; lct_i \leftarrow \min(lct_i\,,\, \max\{lct_j - d_j \mid j \in \Omega\}) \qquad \text{(NL)}$

- Example: For $\Omega = \{A,B\}$, activity $i = C$ cannot start last:



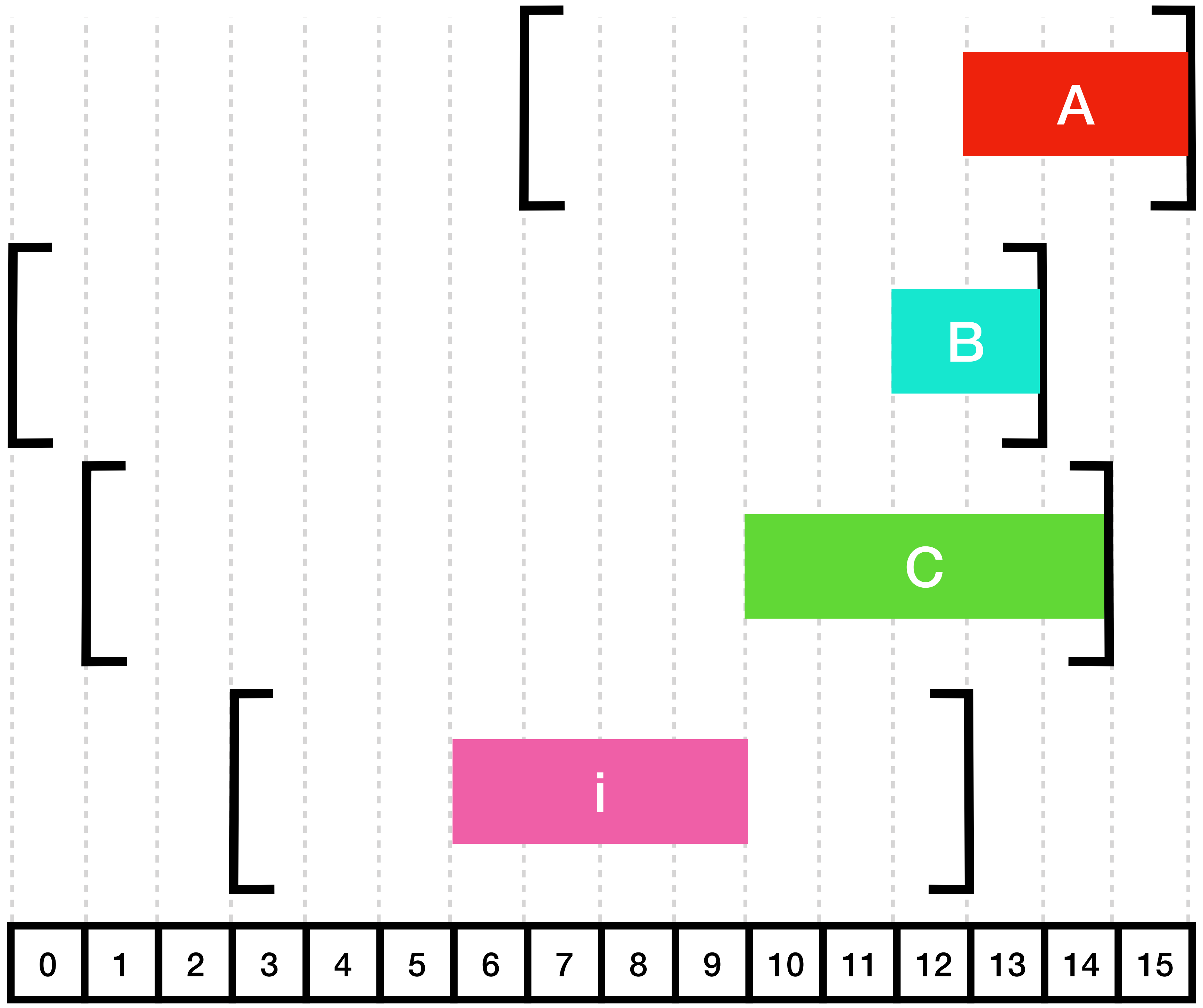
It is impossible to have $\{A,B\} \ll C$,
so C must end before A or B (or both):
$lct_C \leftarrow \min(lct_C\,,\, \max\{lct_B - d_B,\, lct_A - d_A\})$.

- Again, we need to find a way to enumerate the Ω in a nested way.

Not-Last Rule

- $est_\Omega + d_\Omega > lct_i - d_i \;\rightsquigarrow\; lct_i \leftarrow \min(lct_i\,,\, \max\{lct_j - d_j \mid j \in \Omega\})$ (NL)

- Observation: If there is a subset Ω for which this rule *actually filters*, then it is a subset of NLSet(T,i) = $\{\, j \mid j \in T \setminus \{i\} \;\&\; lct_j - d_j < lct_i \,\}$.



A is not in NLSet(T,i), but B and C are in it!

Not-Last Rule

- $est_\Omega + d_\Omega > lct_i - d_i \leadsto lct_i \leftarrow \min(lct_i, \max\{lct_j - d_j \mid j \in \Omega\})$ (NL)

- Observation: If there is a subset Ω for which this rule *actually filters*, then it is a subset of NLSet(T,i) $= \{ j \mid j \in T \setminus \{i\} \ \&\ lct_j - d_j < lct_i \}$.

- Does there exist a subset $\Omega \subseteq$ NLSet(T,i) for which the *detection* part of the rule (namely $est_\Omega + d_\Omega > lct_i - d_i$) *also* holds?

- Such a subset *exists* if and only if $\max\{est_{\Omega'} + d_{\Omega'} \mid \Omega' \subseteq NLSet(T,i)\} > lct_i - d_i$.

The left-hand side is the definition of $ect_{NLSet(T,i)}$: this probably means that a Θ-tree will be useful…

Not-Last Rule

Let us make this more efficient!

- The *existence* of a subset $\Omega \subseteq NLSet(T,i)$ triggering the rule can be tested as

$$ect_{NLSet(T,i)} > lct_i - d_i$$

- The problem is that we then do not have a subset Ω for filtering (we only test for the existence of it to trigger the rule).

- But do we really need it?
  No! if we accept to *relax* the filtering:

$$\max \{lct_j - d_j \mid j \in \Omega\} \leq \max \{lct_j - d_j \mid j \in NLSet(T,i)\} < lct_i$$

Because $\Omega \subseteq NLSet(T,i)$:
the advantage of this relaxation
is that we do *not* need a Ω!

Weaker Not-Last Rule

- $est_\Omega + d_\Omega > lct_i - d_i \;\rightsquigarrow\; lct_i \leftarrow \min(lct_i\,, \max\{lct_j - d_j \mid j \in \Omega\})$ (NL)
- $ect_{NLSet(T,i)} > lct_i - d_i \;\rightsquigarrow\; lct_i \leftarrow \max\{lct_j - d_j \mid j \in NLSet(T,i)\}$ (NL')
- Rule NL' may filter less than rule NL, but the fixpoint is the same.

Not-Last: Implementation

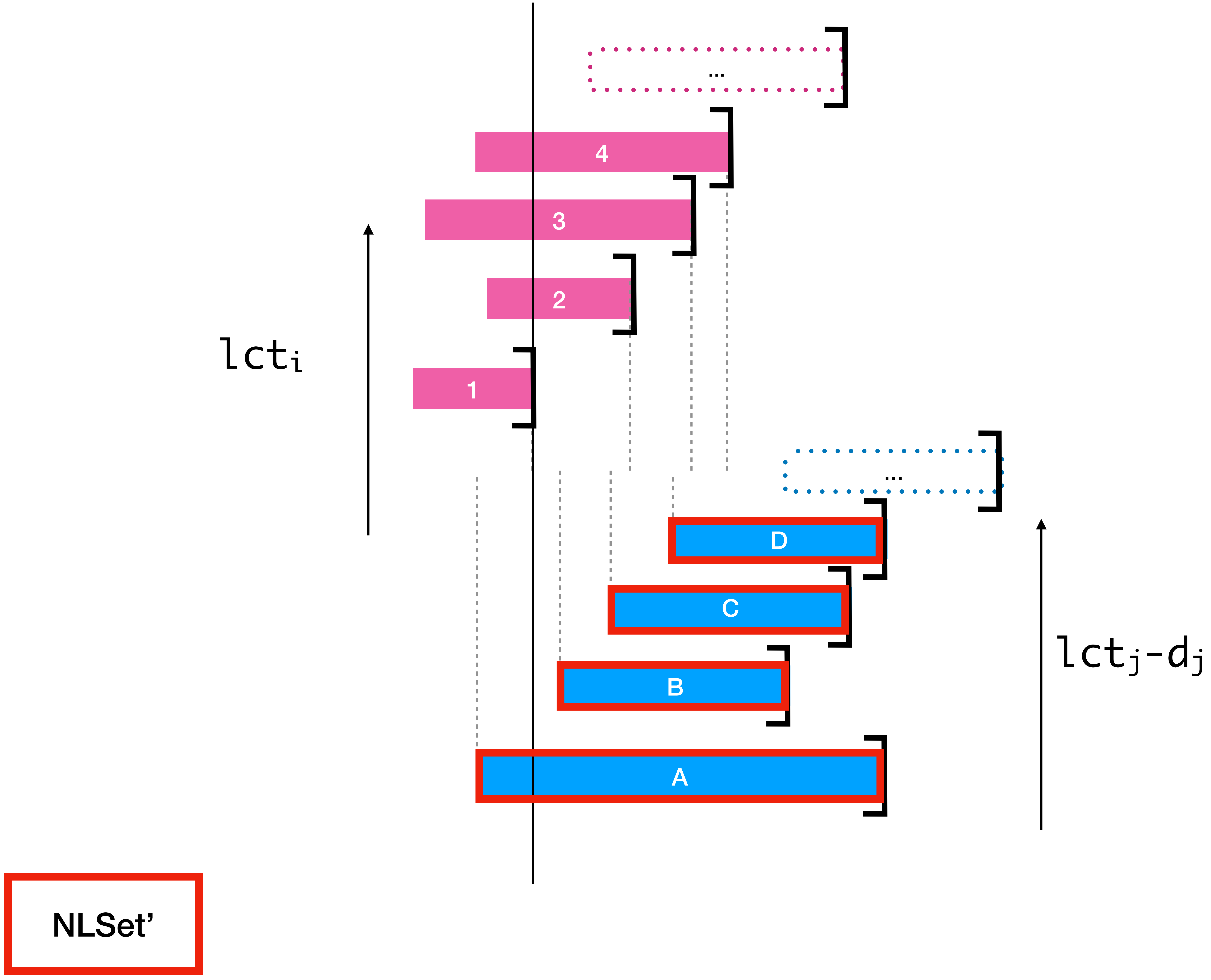
- Recall: $NLSet(T,i) = \{ j \mid j \in T \setminus \{i\} \;\&\; lct_j - d_j < lct_i \}$.

- We are looking for an order on i so as to have nested sets.

- Let NLSet'(T,i) $= \{ j \mid j \in T \;\&\; lct_j - d_j < lct_i \}$.
  Note that i is *always* in NLSet'(T,i).

- In what order should we consider activities to have nested NLSet'(T,i) sets?

Not-Last: Filtering Algorithm



Not-Last: Implementation

- Let NLSet'(T,i) = $\{ j \mid j \in T \;\&\; lct_j - d_j < lct_i \}$.
  Note that i is *always* in NLSet'(T,i).

- Let T = {1..n} be ordered such that $lct_1 \leq lct_2 \leq \ldots \leq lct_n$:
  then NLSet'(T,1) ⊆ NLSet'(T,2) ⊆ … ⊆ NLSet'(T,n) = T:
  all activities are eventually inserted into the initialised Θ-tree.

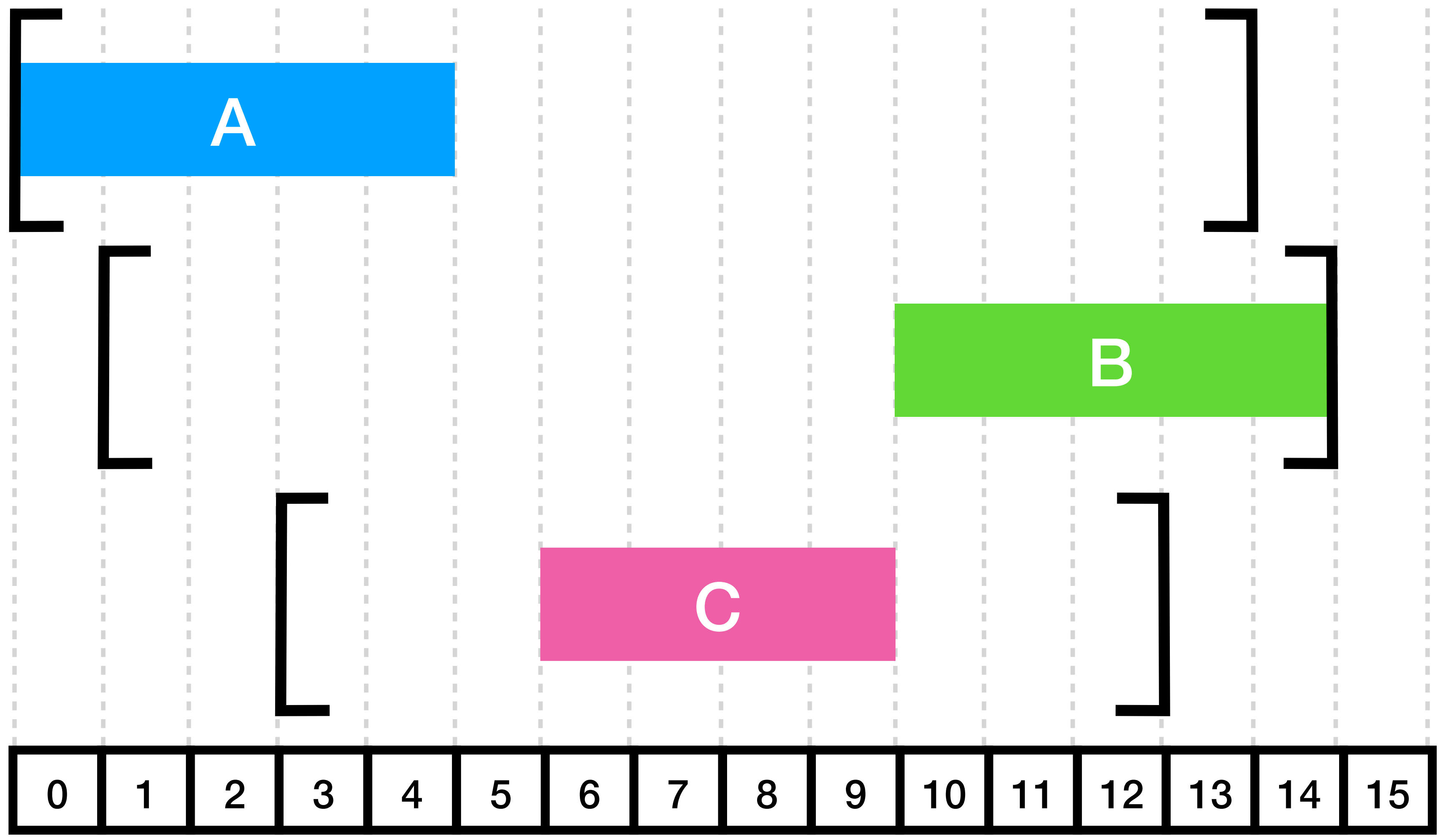
- Now we have a way to compute the NLSet(T,i) incrementally when using a Θ-tree.

Not-Last: Filtering Algorithm

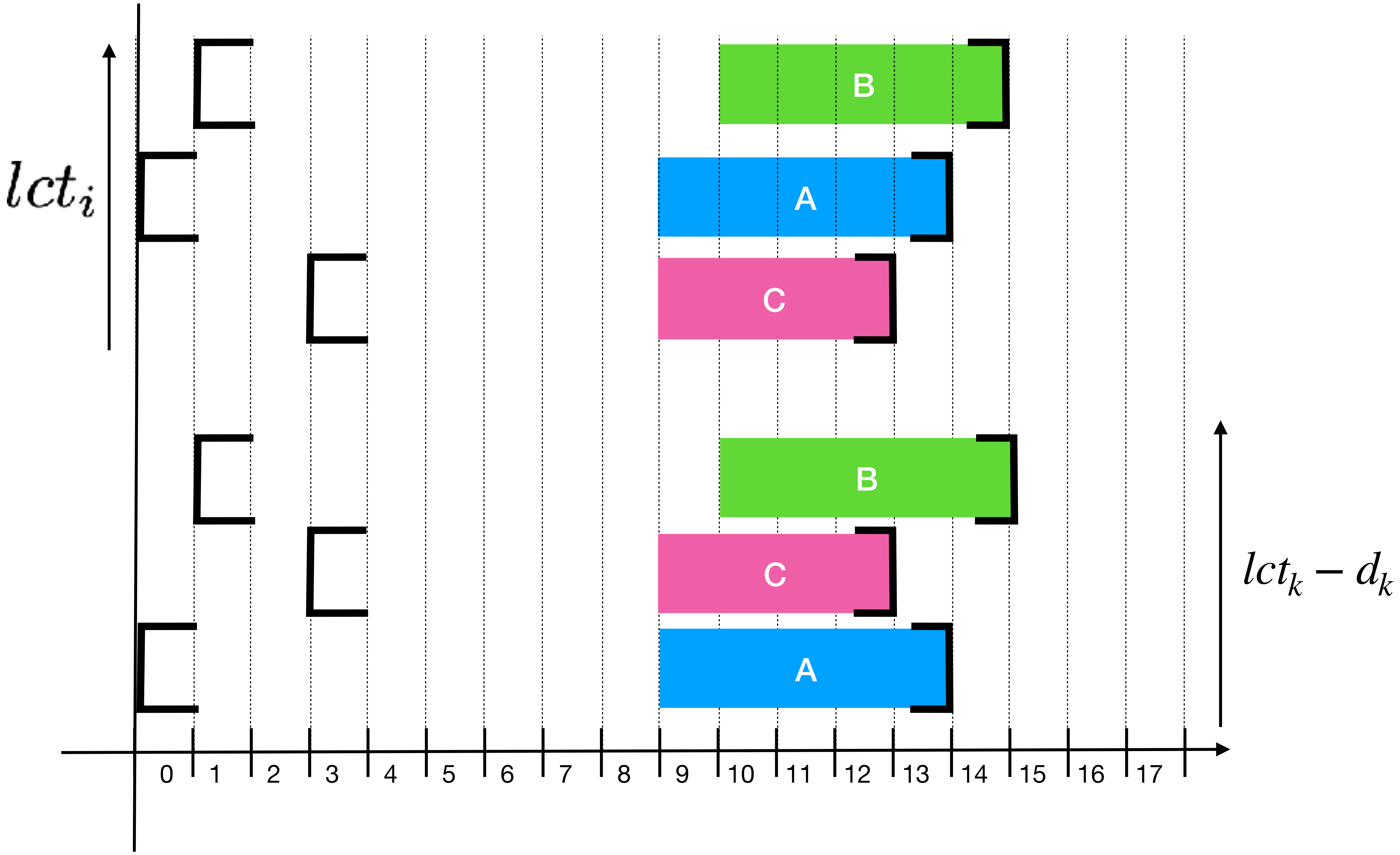
```
NotLast(T={1..n}) {
  lct'i ← lcti, ∀i∈T
  Tlst ← sortAZ([1..n],sortKey = lct-d) // O(n log n) time
  Tlct ← sortAZ([1..n],sortKey = lct) // O(n log n) time
  ite ← iterator(Tlst)
  k ← ite.next()
  j ← -1
  Θ ← Θ-Tree.init({1..n}) // O(n log n) time
  for (i ← Tlct) {
    while (lcti > lctk-dk) {
       Θ.insert(k) // O(log n) time
       j ← k // lctj-dj = max {lctk - dk : k ∈ NLSet(T,i)}
       k ← ite.next()
    }
    if (ectΘ\i > lcti-di) { // O(log n) time
       lct'i ← min(lcti, lctj-dj)
    }
 }
 lcti ← lct'i, ∀i∈T
}
```

Θ-tree contains *all* NLSet'(T,i).

Not last filtering with Θ-Tree, an example

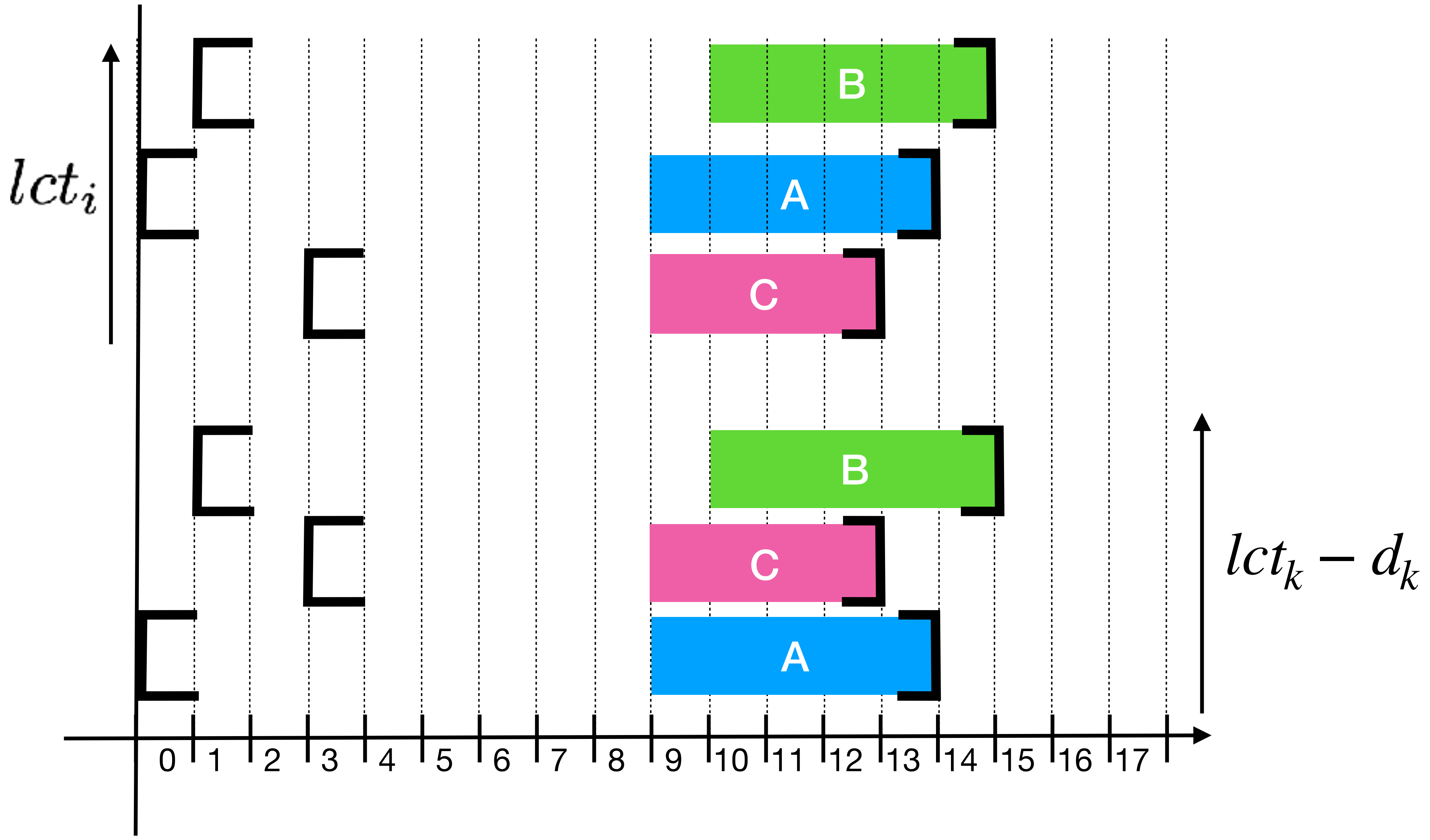


Sorting

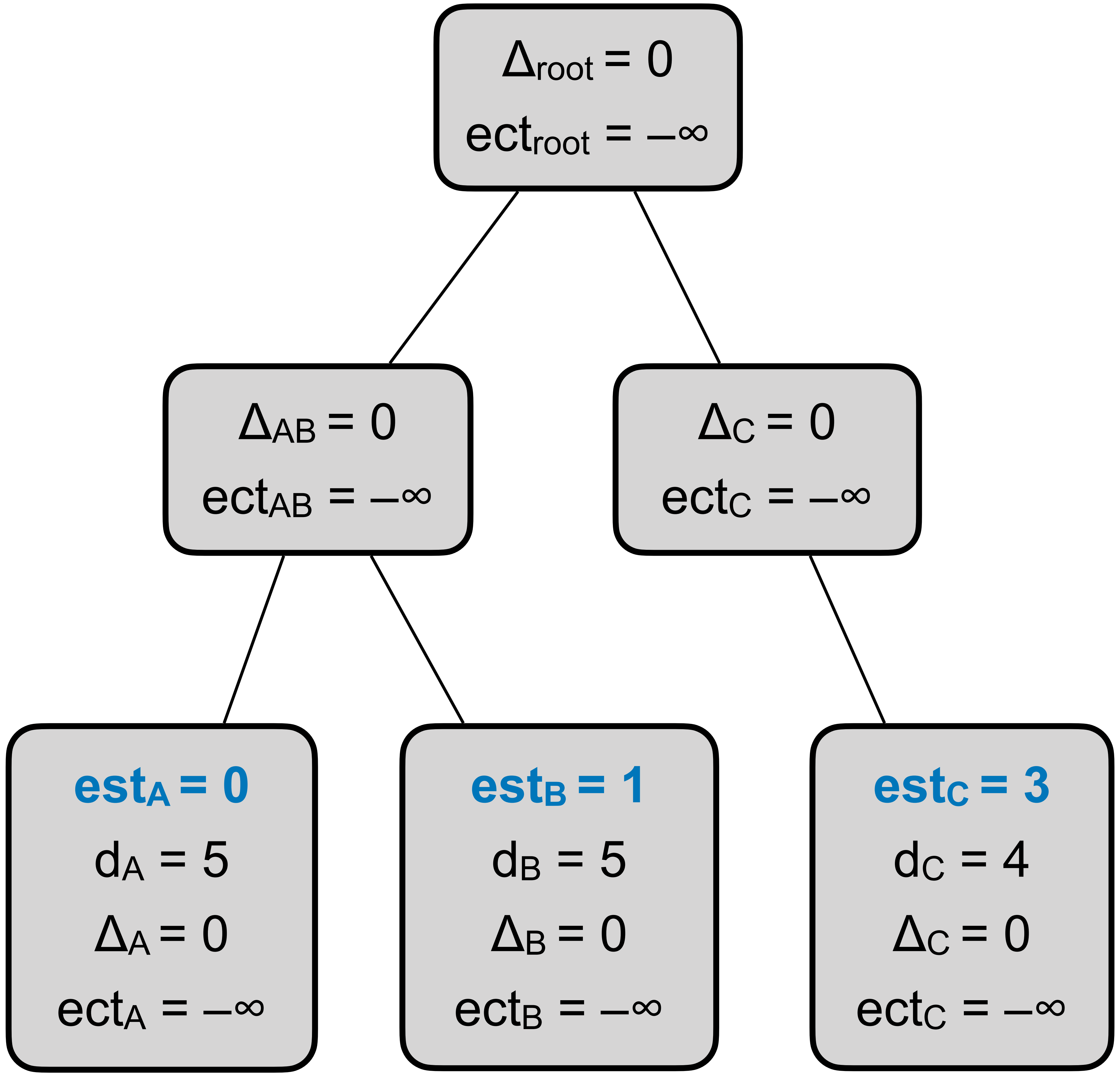



```
NotLast(T={1..n}) {
  lct'i ← lcti, ∀i∈T
  Tlst ← sortAZ([1..n],sortKey = lct-d) // [A, C, B]
  Tlct ← sortAZ([1..n],sortKey = lct) // [C, A, B]
  ite ← iterator(Tlst)
   k ← ite.next() // k = A
   j ← -1
  Θ ← Θ-Tree.init({1..n})
   ...
   ...
}
```

Not last filtering with Θ-Tree, an example

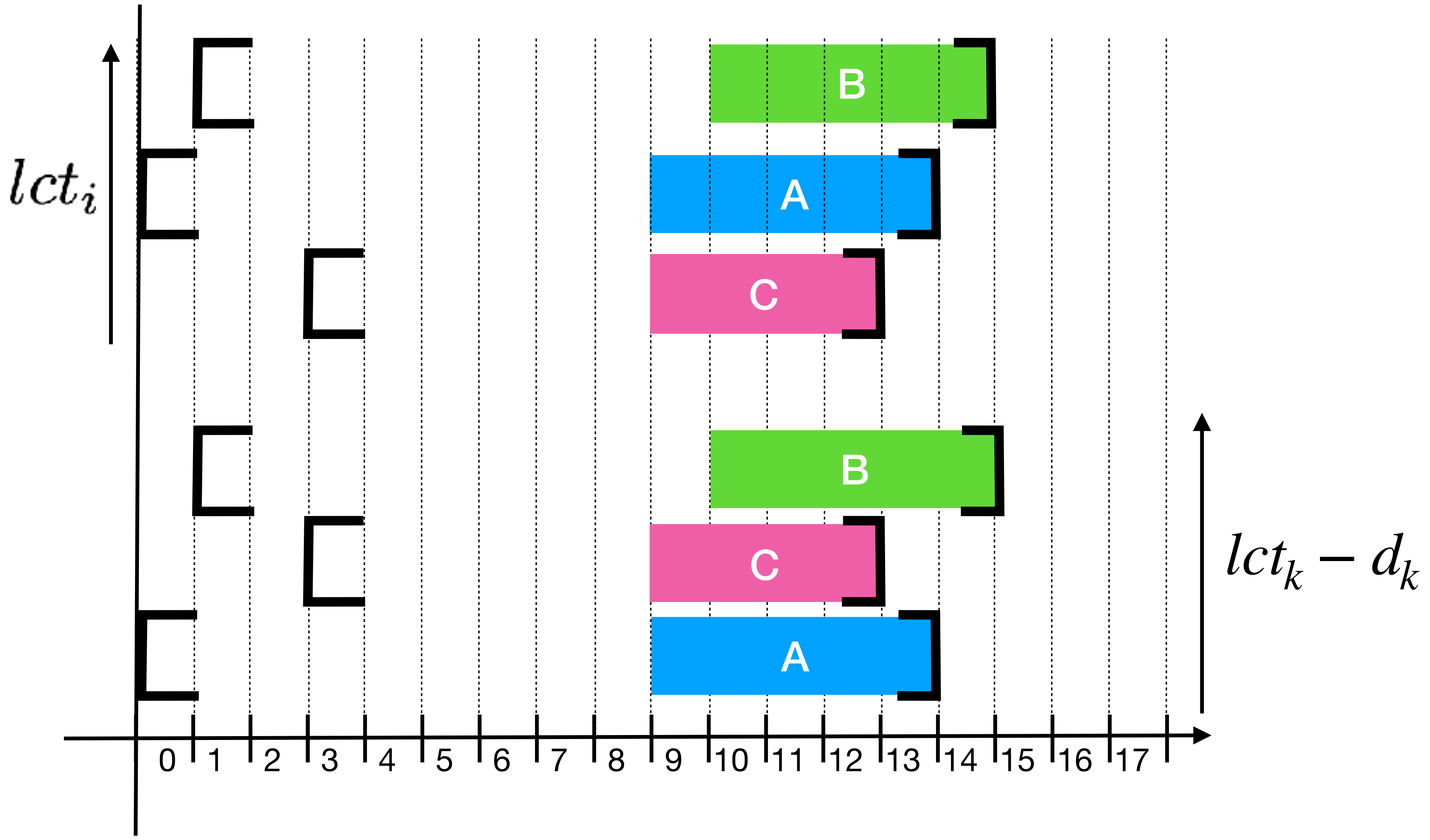


Θ-Tree initialisation



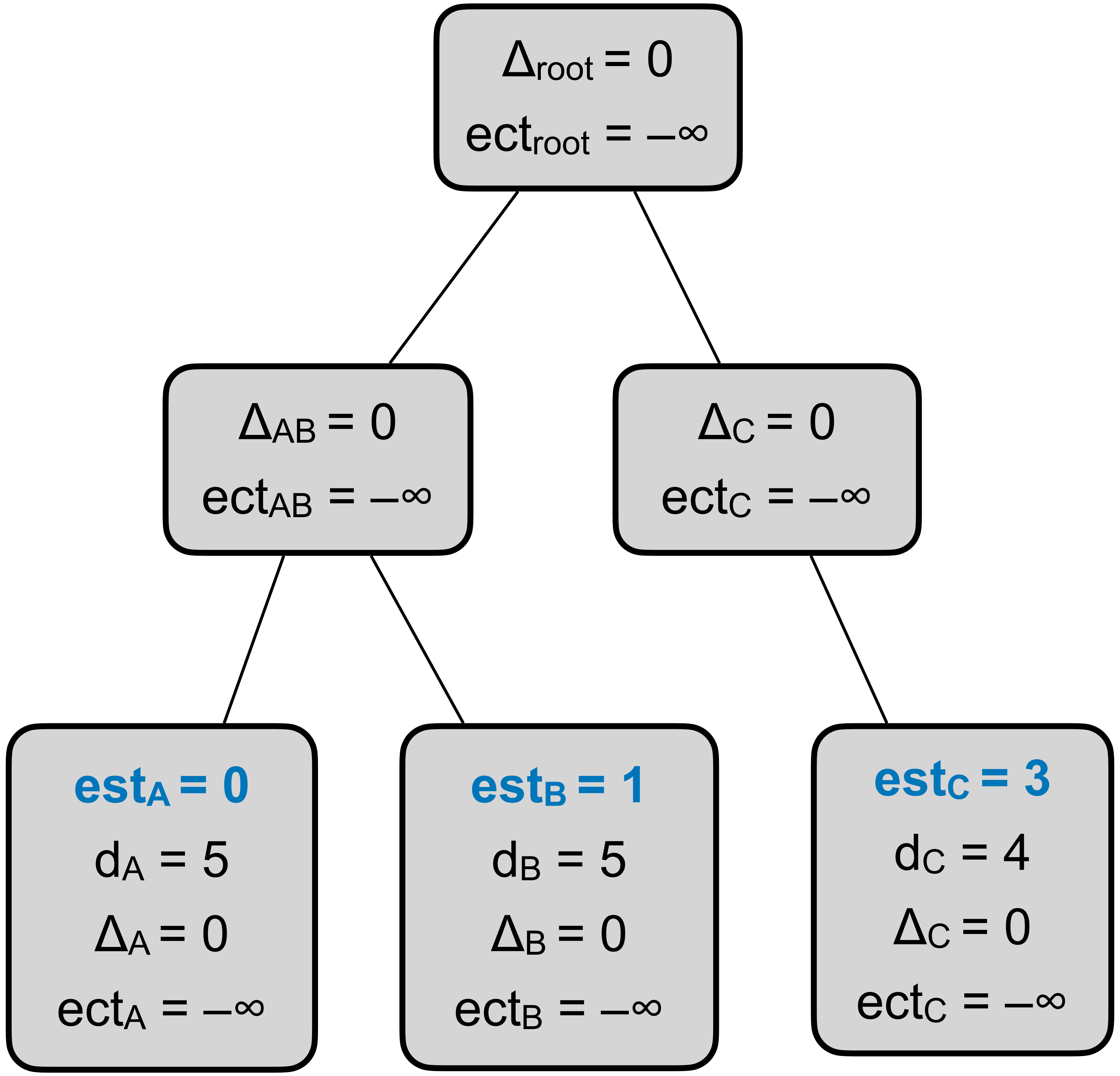

```
NotLast(T={1..n}) {
  lct'i ← lcti, ∀i∈T
  Tlst ← sortAZ([1..n],sortKey = lct-d) // [A, C, B]
  Tlct ← sortAZ([1..n],sortKey = lct) // [C, A, B]
  ite ← iterator(Tlst)
  k ← ite.next() // k = A
  j ← -1
  Θ ← Θ-Tree.init({1..n})
  ...
  ...
}
```

Not last filtering with Θ-Tree, an example

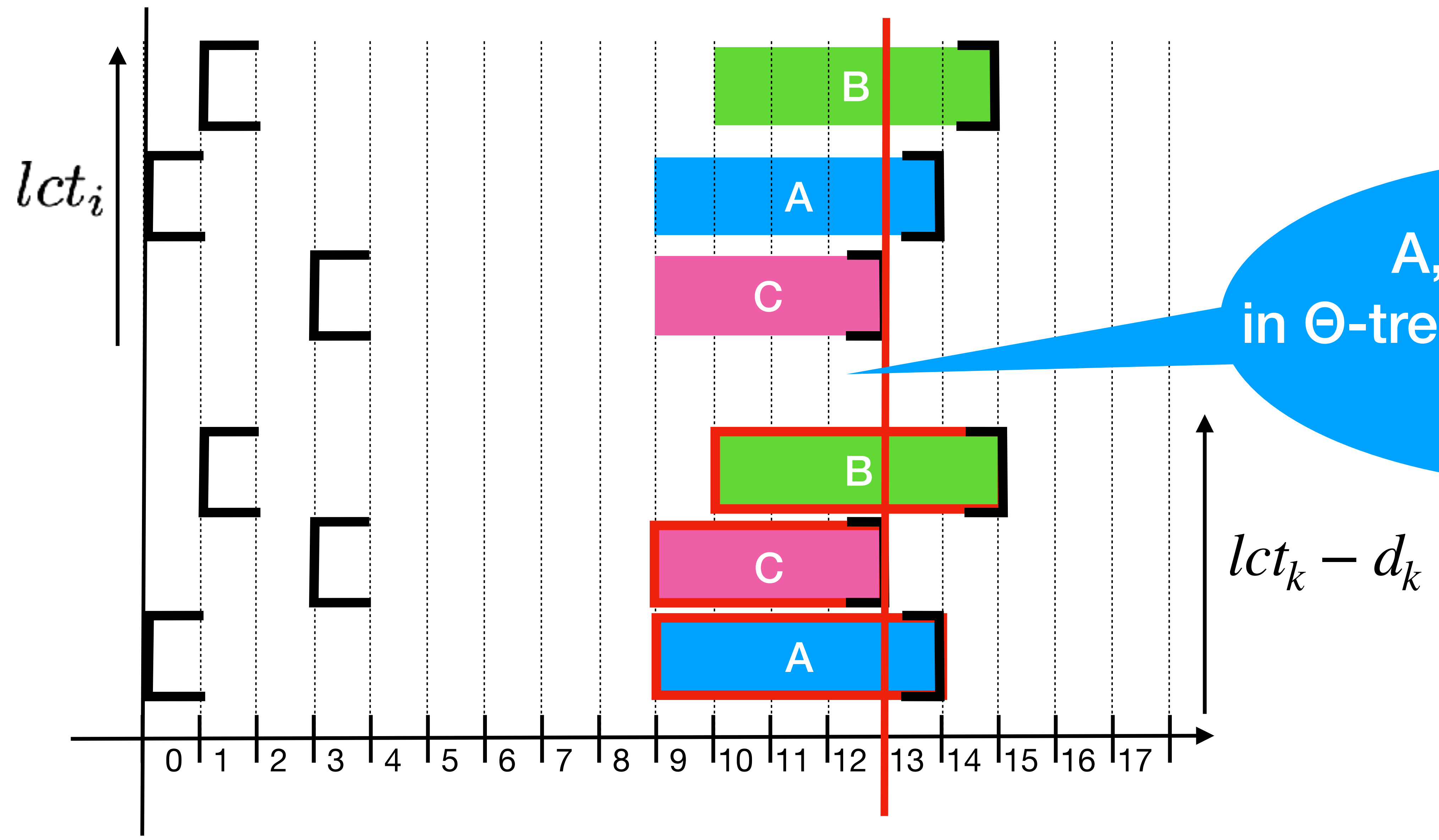



```
NotLast(T={1..n}) {
  ...
  ...
  Θ ← Θ-Tree.init({1..n})
  for (i ← T_lct) { // i ← C
    while (lct_i > lct_k-d_k) {
       Θ.insert(k) // O(log n) time
       j ← k // lct_j-d_j = max {lct_k - d_k : k ∈ NLSet(T,i)}
       k ← ite.next()
    }
    if (ect_Θ\i > lct_i-d_i) { // O(log n) time
       lct'_i ← min(lct_i, lct_j-d_j)
    }
 }
 lct_i ← lct'_i, ∀i∈T
}
```

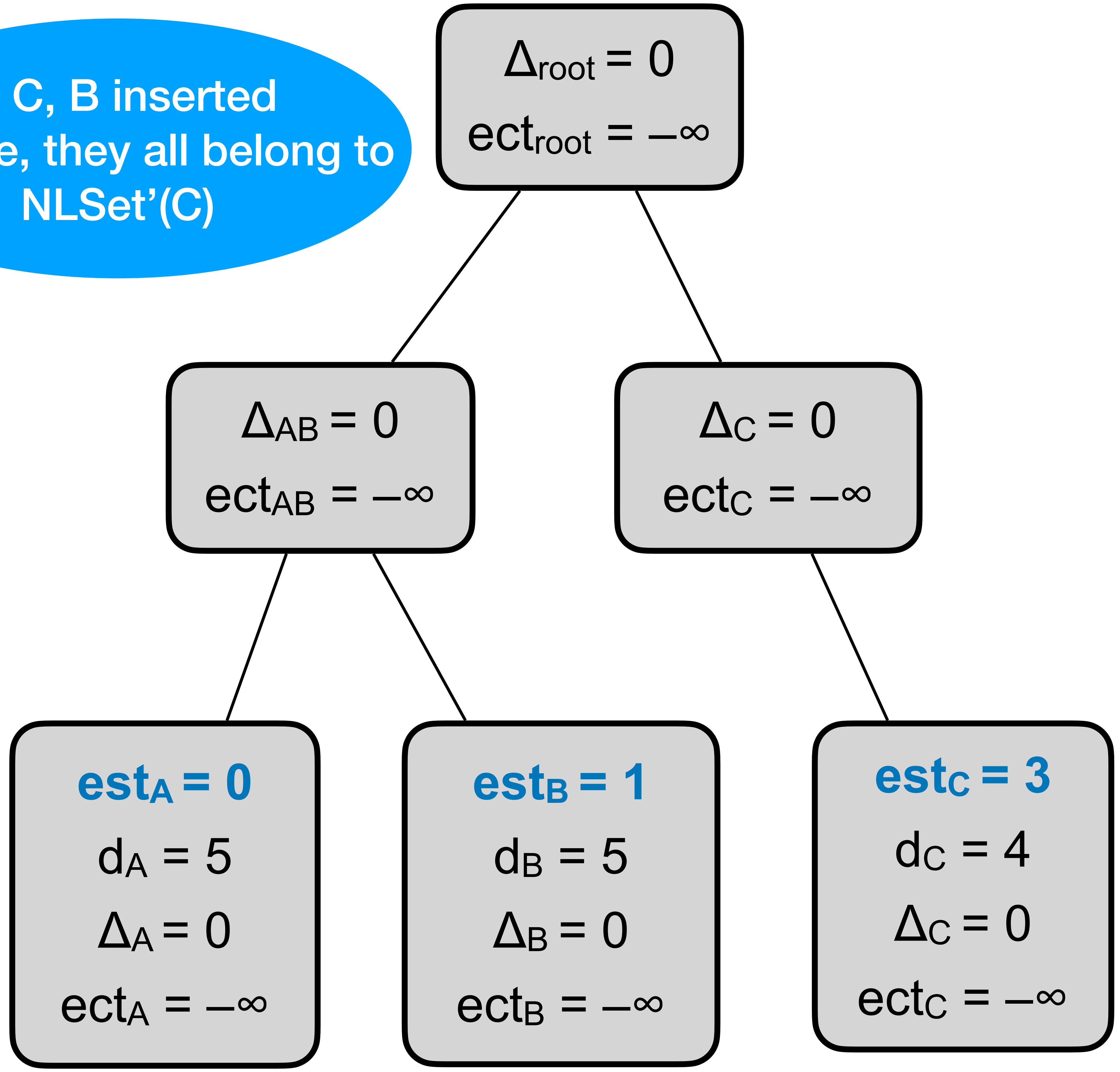
First iteration: C is considered



Not last filtering with Θ-Tree, an example

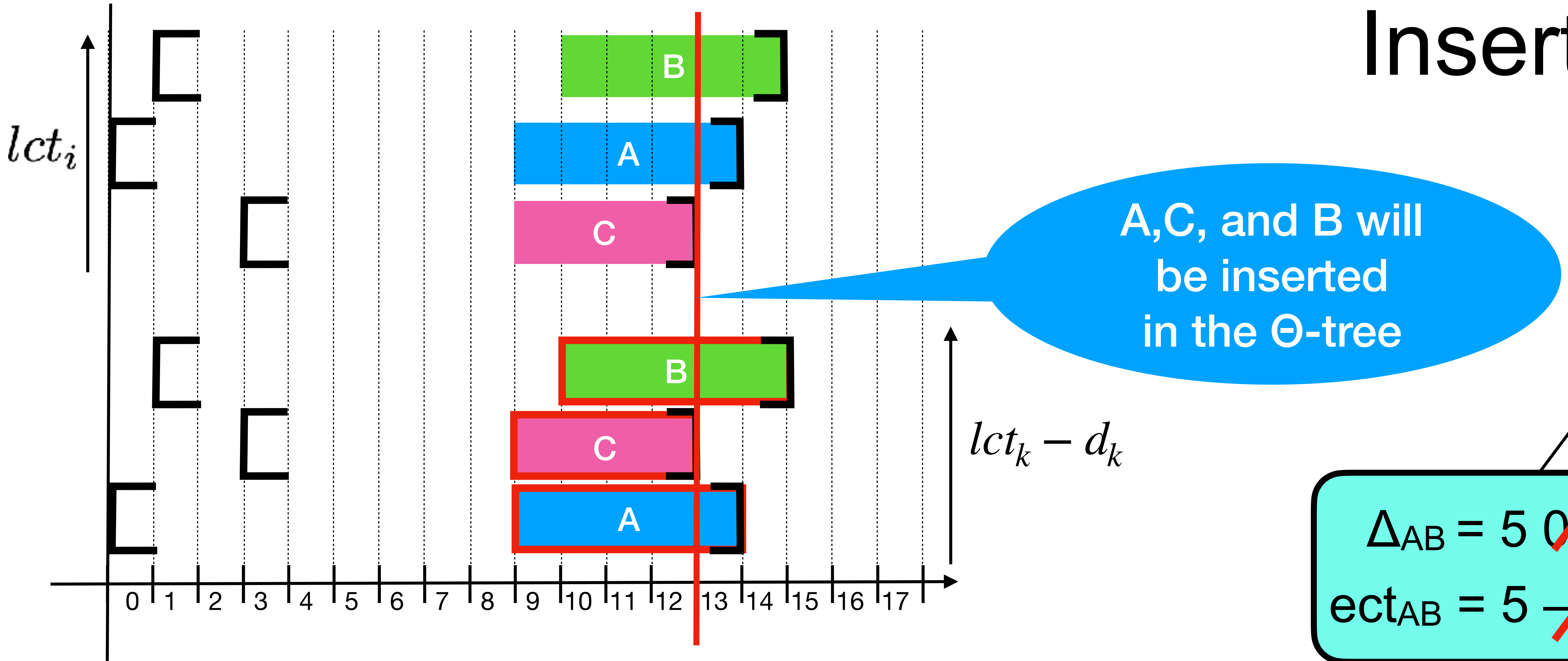


First iteration: C is considered

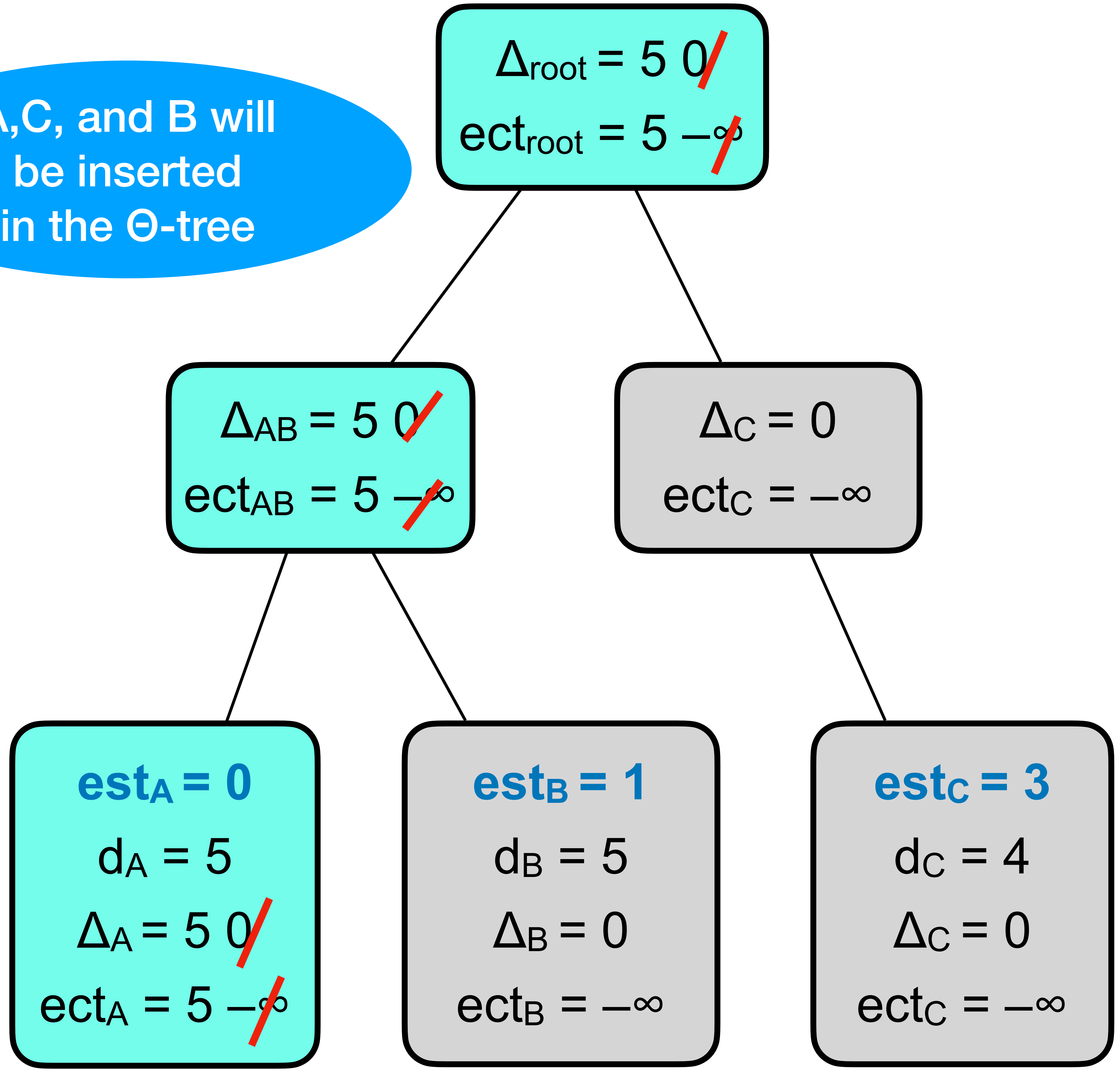



```
NotLast(T={1..n}) {
  ...
  ...
  Θ ← Θ-Tree.init({1..n})
  for (i ← Tlct) { // i ← C
    while (lcti > lctk-dk) {
       Θ.insert(k) // O(log n) time
       j ← k // lctj-dj = max {lctk - dk : k ∈ NLSet(T,i)}
       k ← ite.next()
    }
    if (ectΘ\i > lcti-di) { // O(log n) time
       lct'i ← min(lcti, lctj-dj)
    }
 }
 lcti ← lct'i, ∀i∈T
}
```

Not last filtering with Θ-Tree, an example

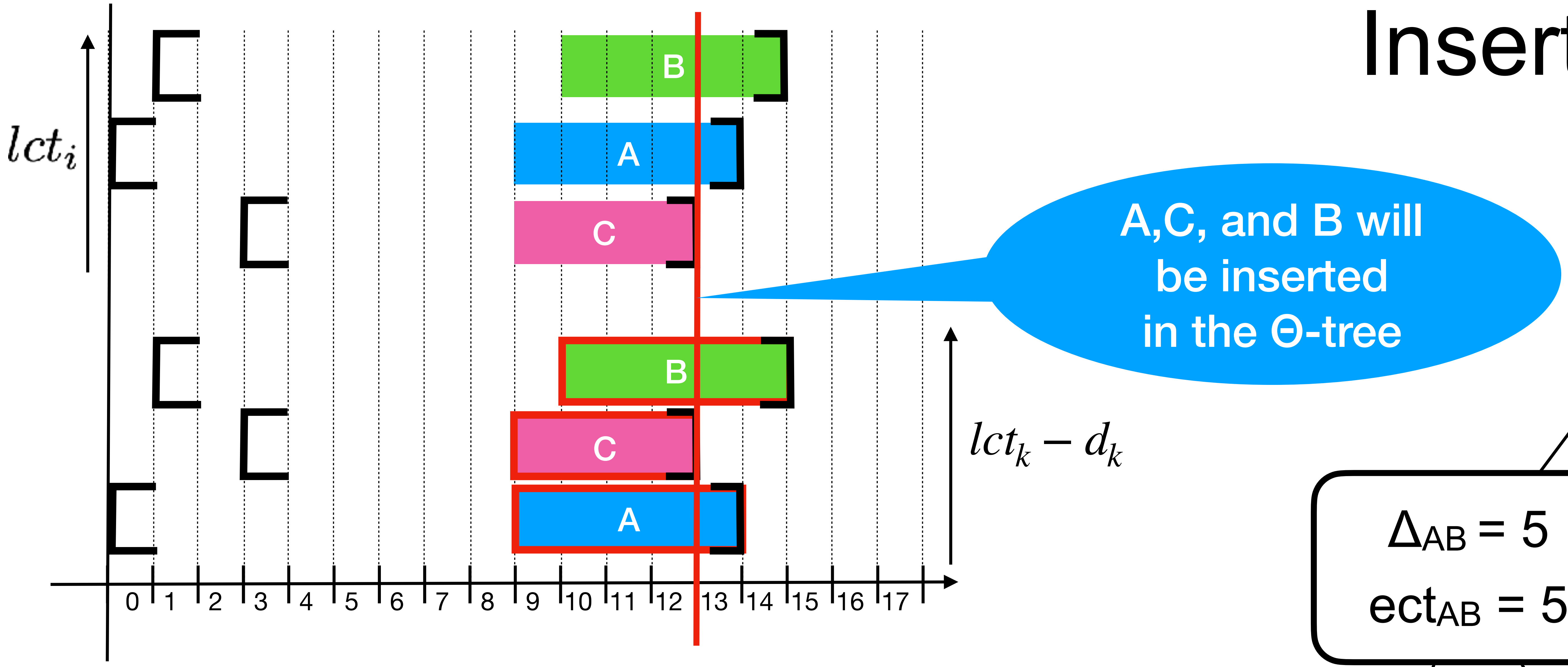


Insertion of A



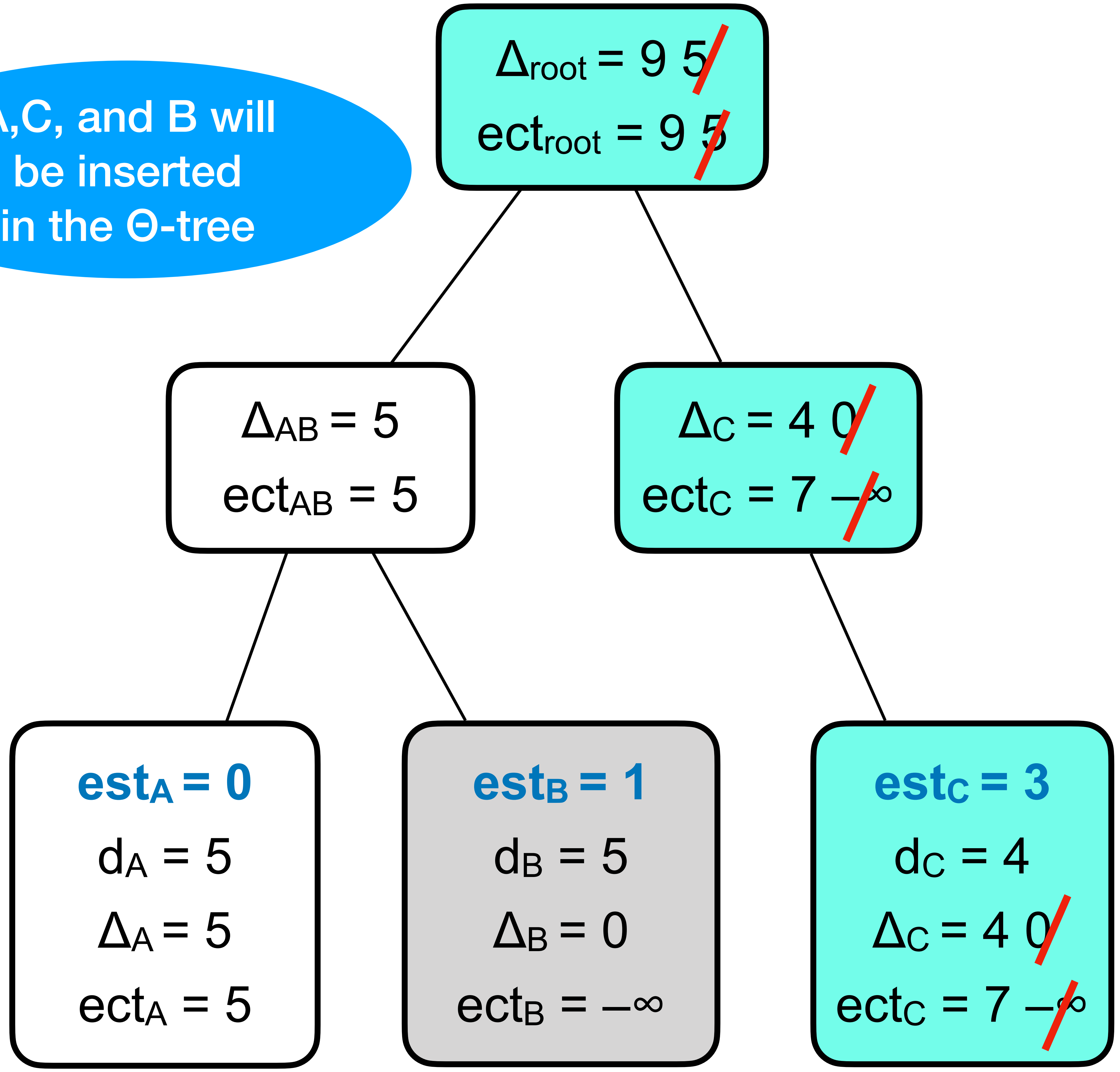

```
NotLast(T={1..n}) {
  ...
  ...
  Θ ← Θ-Tree.init({1..n})
  for (i ← Tlct) { // i ← C
    while (lcti > lctk-dk) { // k = A
       Θ.insert(k) // O(log n) time
       j ← k // lctj-dj = max {lctk - dk : k ∈ NLSet(T,i)}
       k ← ite.next()
    }
    if (ectΘ\i > lcti-di) { // O(log n) time
       lct'i ← min(lcti, lctj-dj)
    }
 }
 lcti ← lct'i, ∀i∈T
}
```

Not last filtering with Θ-Tree, an example

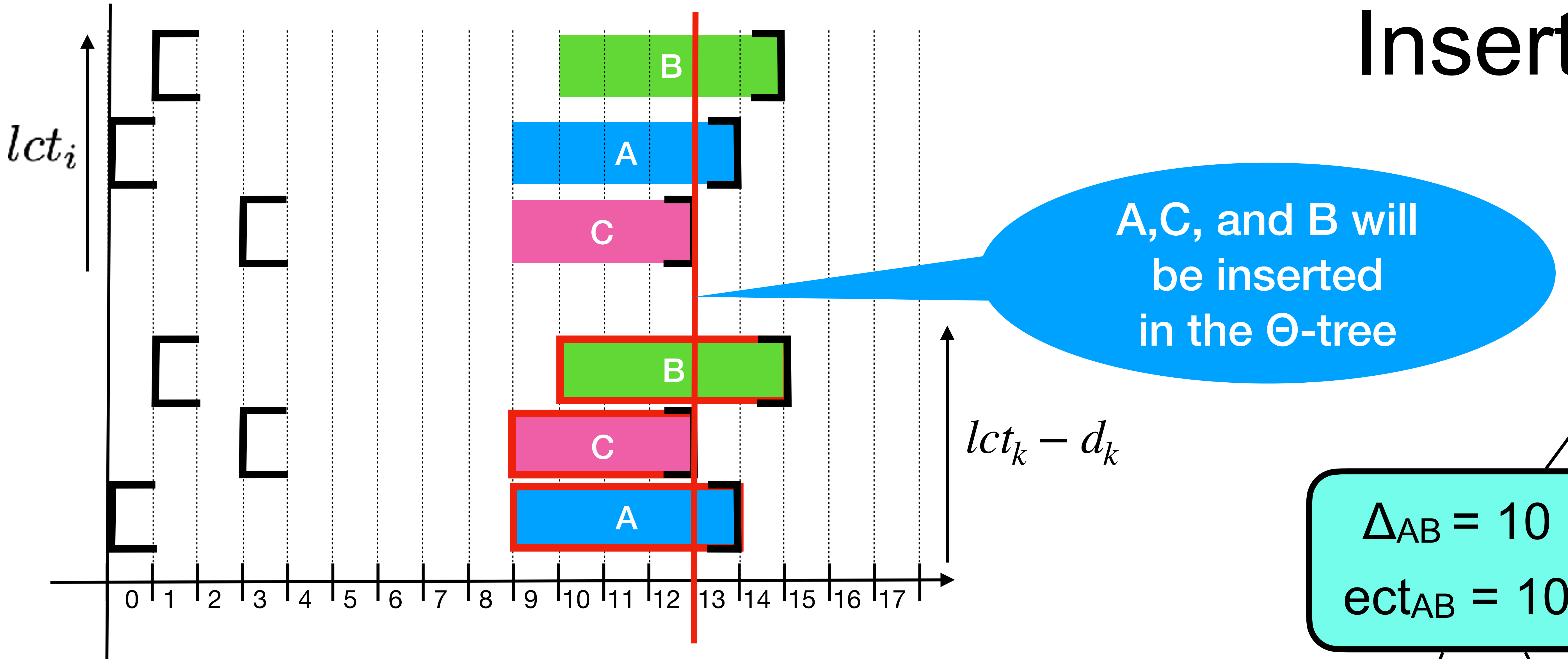



```
NotLast(T={1..n}) {
  ...
  ...
  Θ ← Θ-Tree.init({1..n})
  for (i ← T_lct) { // i ← C
    while (lct_i > lct_k-d_k) { // k = C
       Θ.insert(k) // O(log n) time
       j ← k // lct_j-d_j = max {lct_k - d_k : k ∈ NLSet(T,i)}
       k ← ite.next()
    }
    if (ect_Θ\i > lct_i-d_i) { // O(log n) time
       lct'_i ← min(lct_i, lct_j-d_j)
    }
 }
 lct_i ← lct'_i, ∀i∈T
}
```

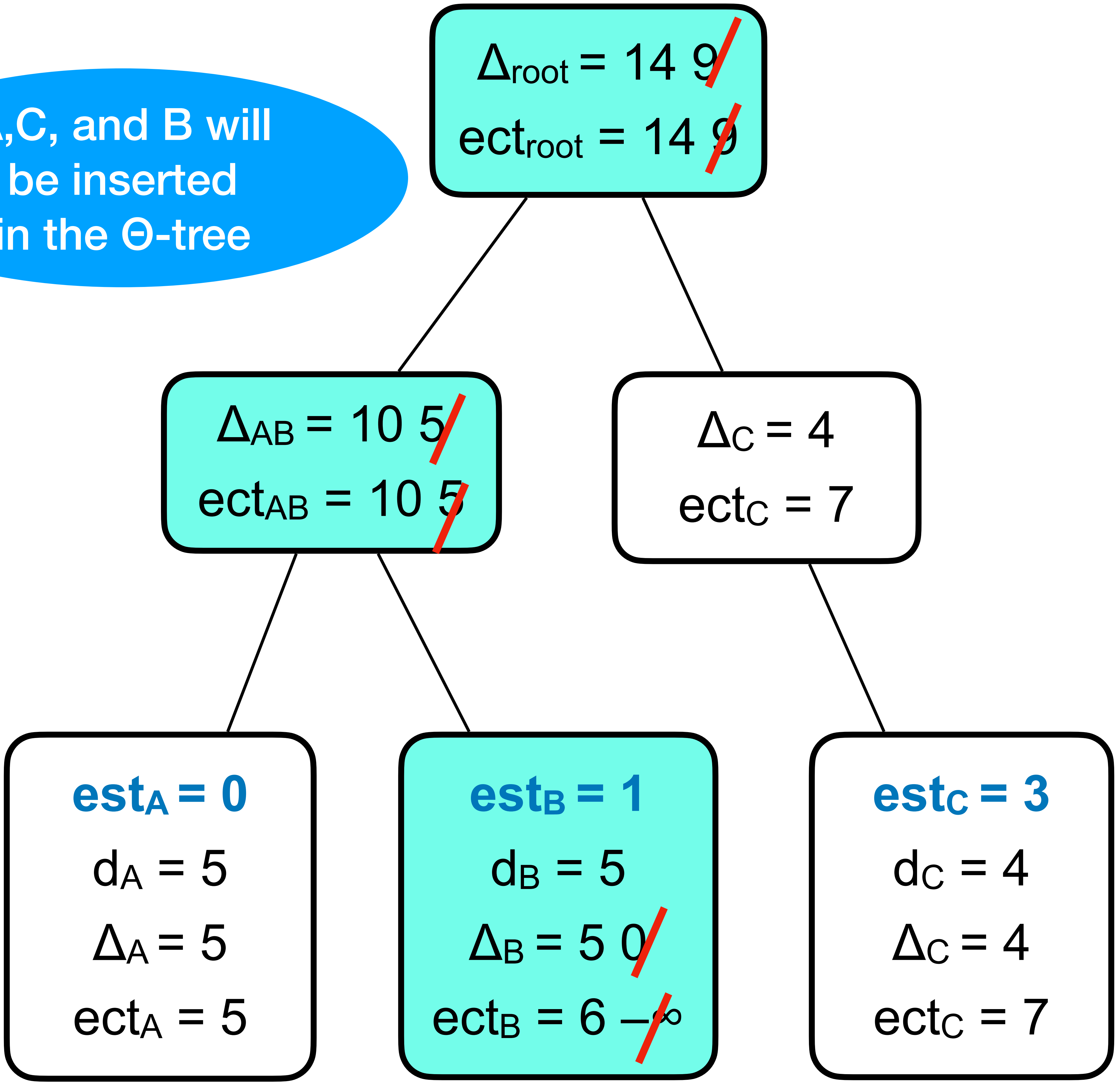
Insertion of C



Not last filtering with Θ-Tree, an example

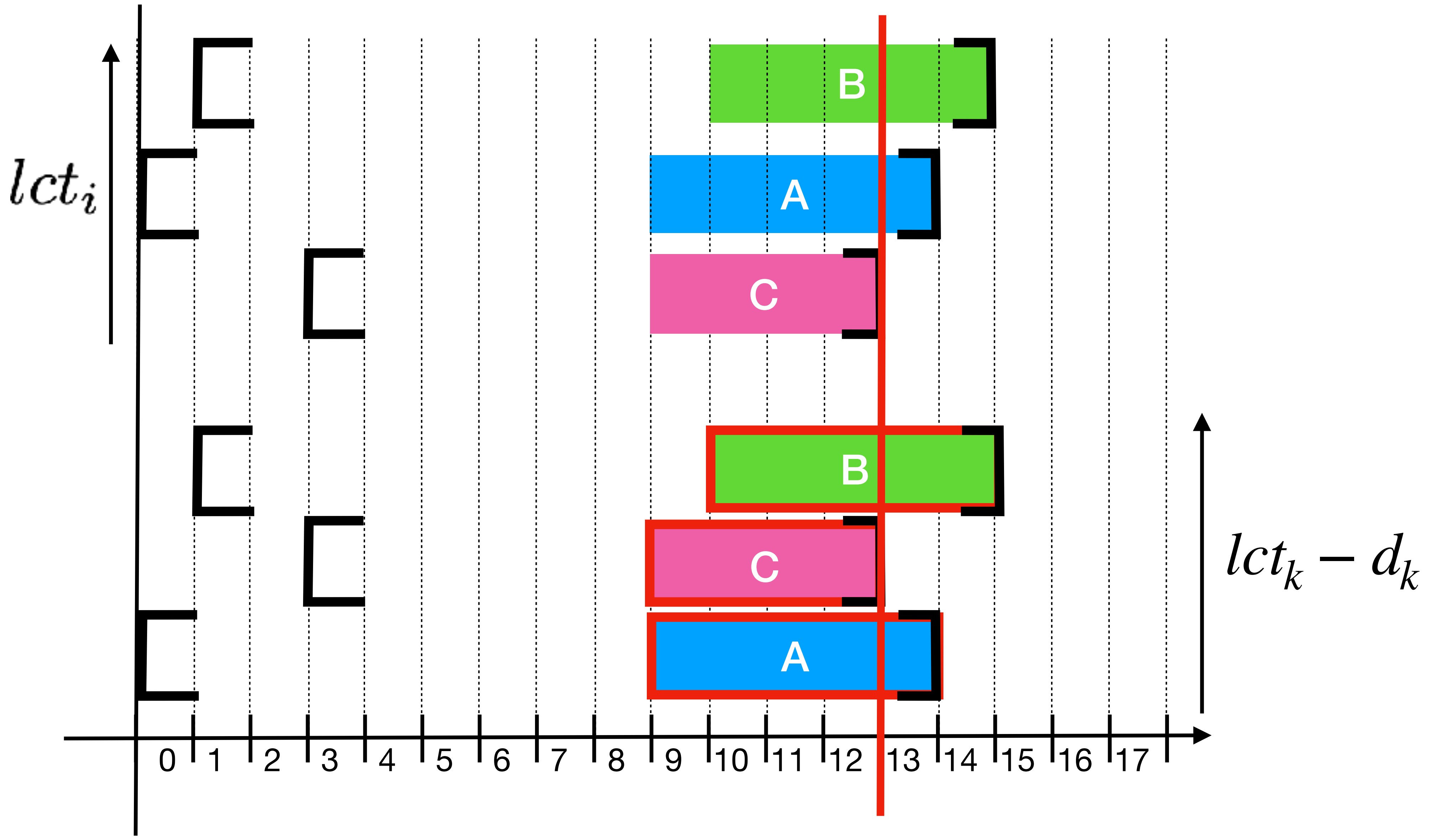


Insertion of B



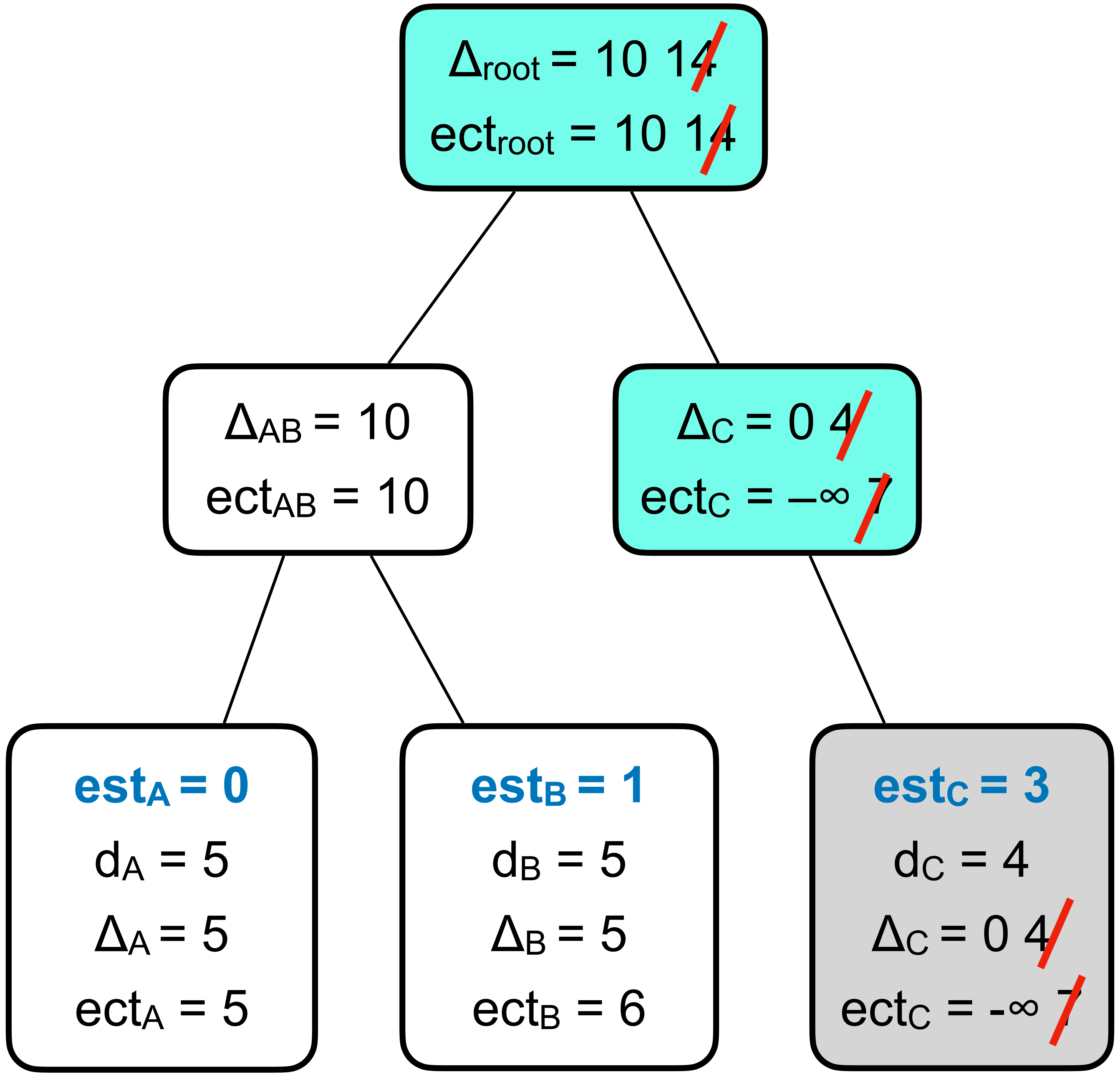

```
NotLast(T={1..n}) {
  ...
  ...
  Θ ← Θ-Tree.init({1..n})
  for (i ← Tlct) { // i ← C
    while (lctᵢ > lctₖ-dₖ) { // k = B
       Θ.insert(k) // O(log n) time
       j ← k // lctⱼ-dⱼ = max {lctₖ - dₖ : k ∈ NLSet(T,i)}
       k ← ite.next()
    }
    if (ectΘ\i > lctᵢ-dᵢ) { // O(log n) time
       lct'ᵢ ← min(lctᵢ, lctⱼ-dⱼ)
    }
 }
 lctᵢ ← lct'ᵢ, ∀i∈T
}
```

Not last filtering with Θ-Tree, an example



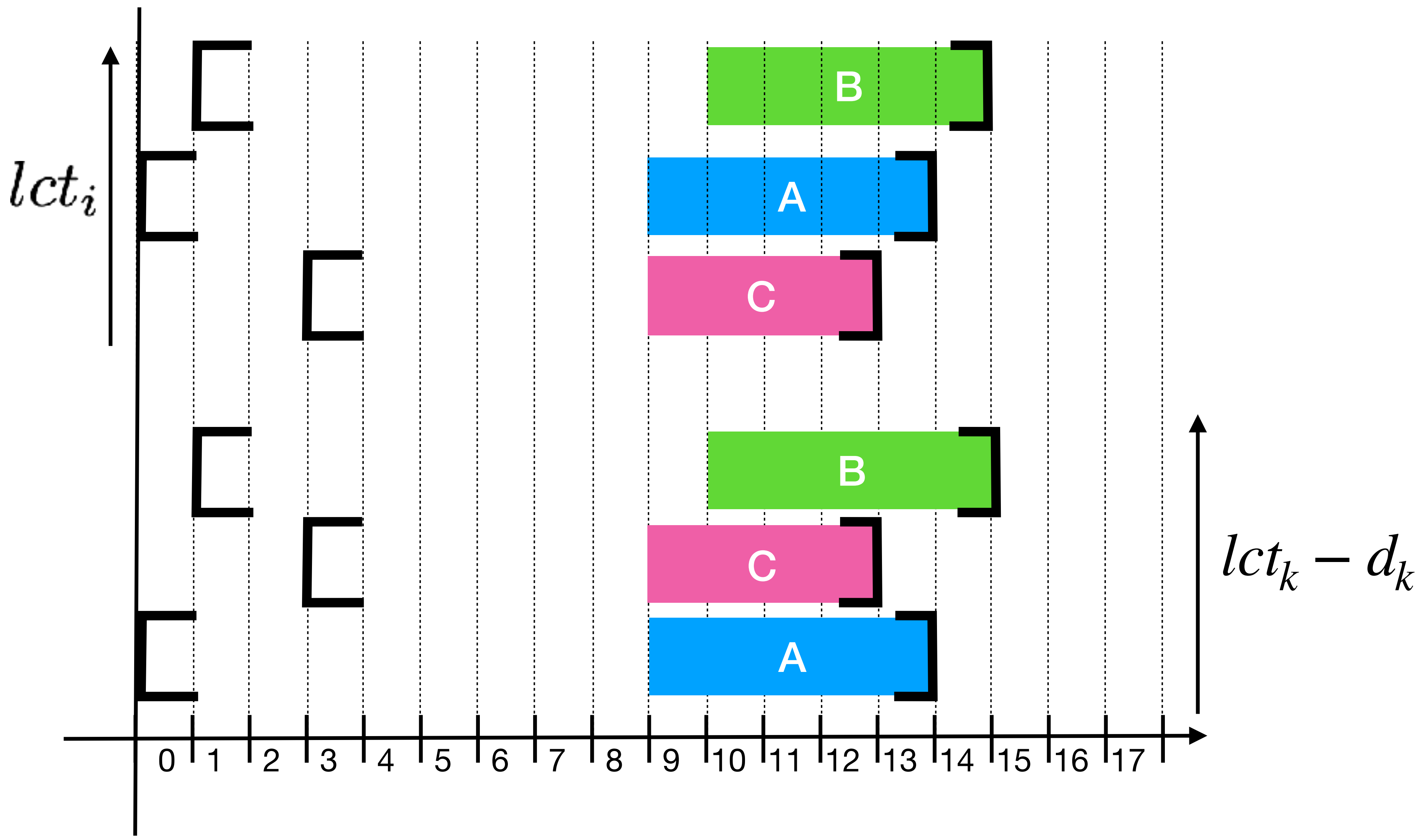

```
NotLast(T={1..n}) {
  ...
  ...
  Θ ← Θ-Tree.init({1..n})
  for (i ← Tlct) { // i ← C
    while (lcti > lctk-dk) { // k = B
      Θ.insert(k) // O(log n) time
      j ← k // lctj-dj = max {lctk - dk : k ∈ NLSet(T,i)}
      k ← ite.next()
    }
    if (ectΘ\i > lcti-di) { // ectΘ\C = 10 and lctC-dC = 9
      lct'i ← min(lcti, lctj-dj)
    }
  }
  lcti ← lct'i, ∀i∈T
}
```

$\Theta \setminus \{C\}$



lct'C = min(lctC, lctB-dB) = 10

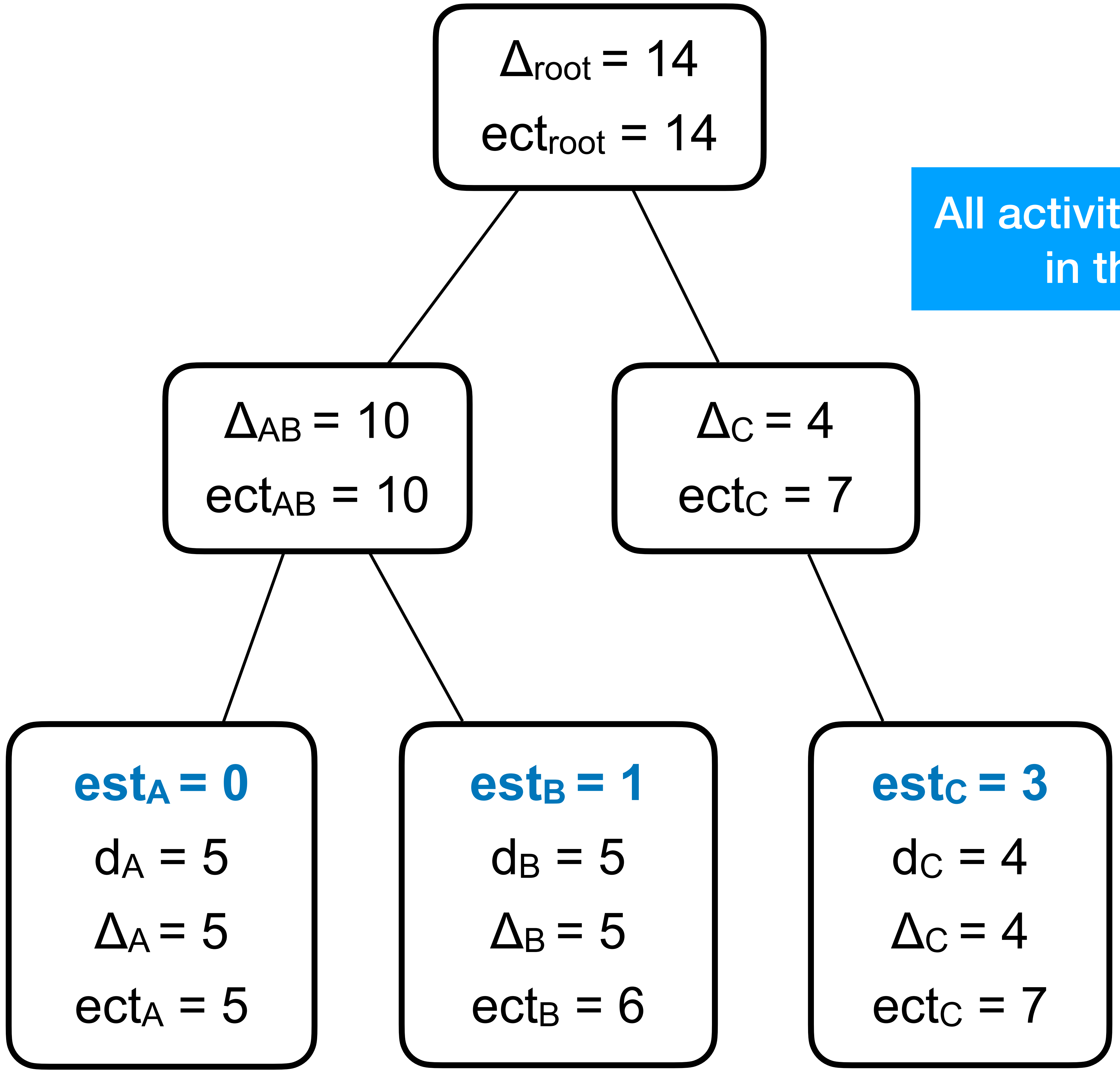
Not last filtering with Θ-Tree, an example



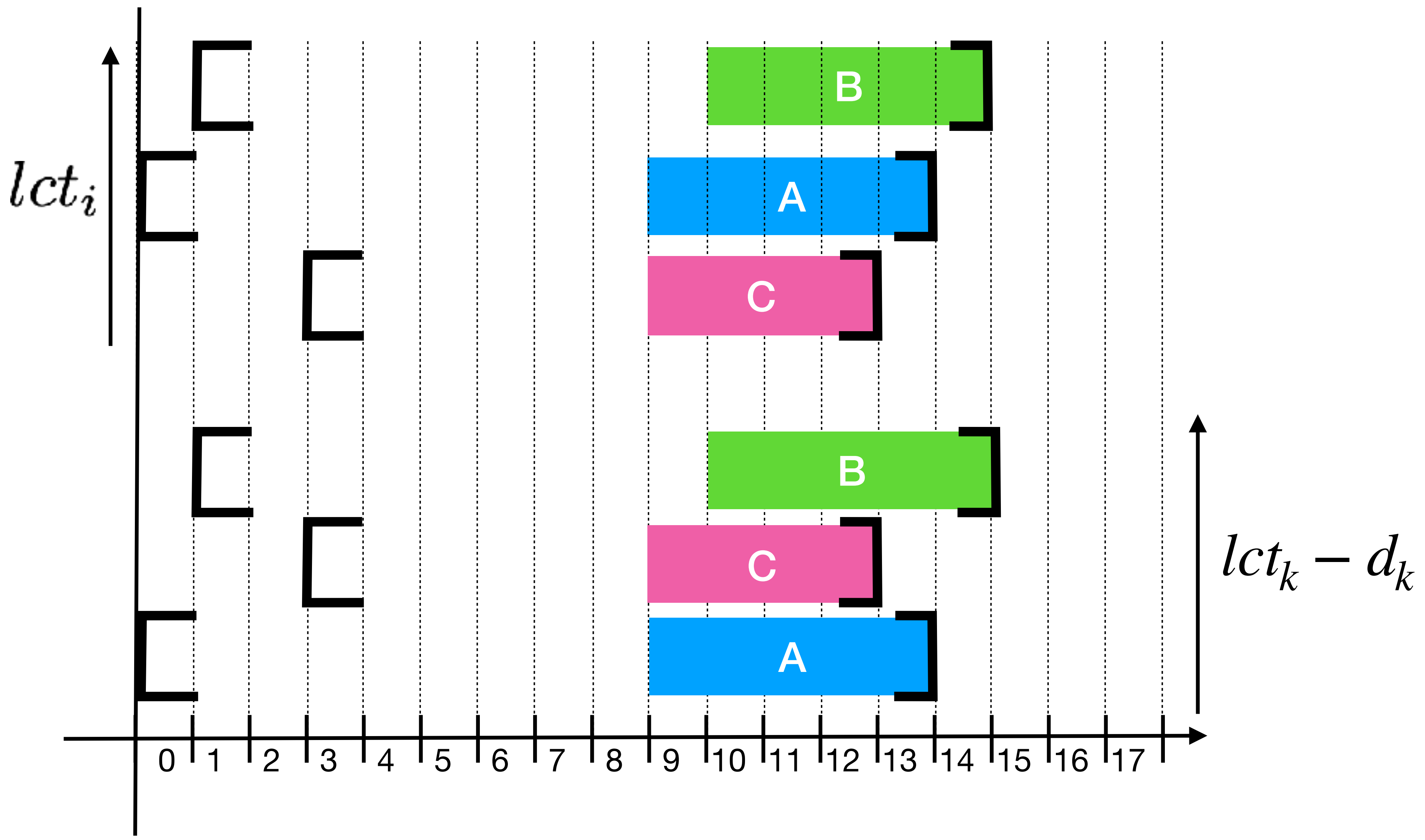

```
NotLast(T={1..n}) {
  ...
  ...
  Θ ← Θ-Tree.init({1..n})
  for (i ← T_lct) { // i ← A
    while (lct_i > lct_k-d_k) {
       Θ.insert(k) // O(log n) time
       j ← k // lct_j-d_j = max {lct_k - d_k : k ∈ NLSet(T,i)}
       k ← ite.next()
    }
    if (ect_Θ\i > lct_i-d_i) { // O(log n) time
       lct'_i ← min(lct_i, lct_j-d_j)
    }
  }
  lct_i ← lct'_i, ∀i∈T
}
```

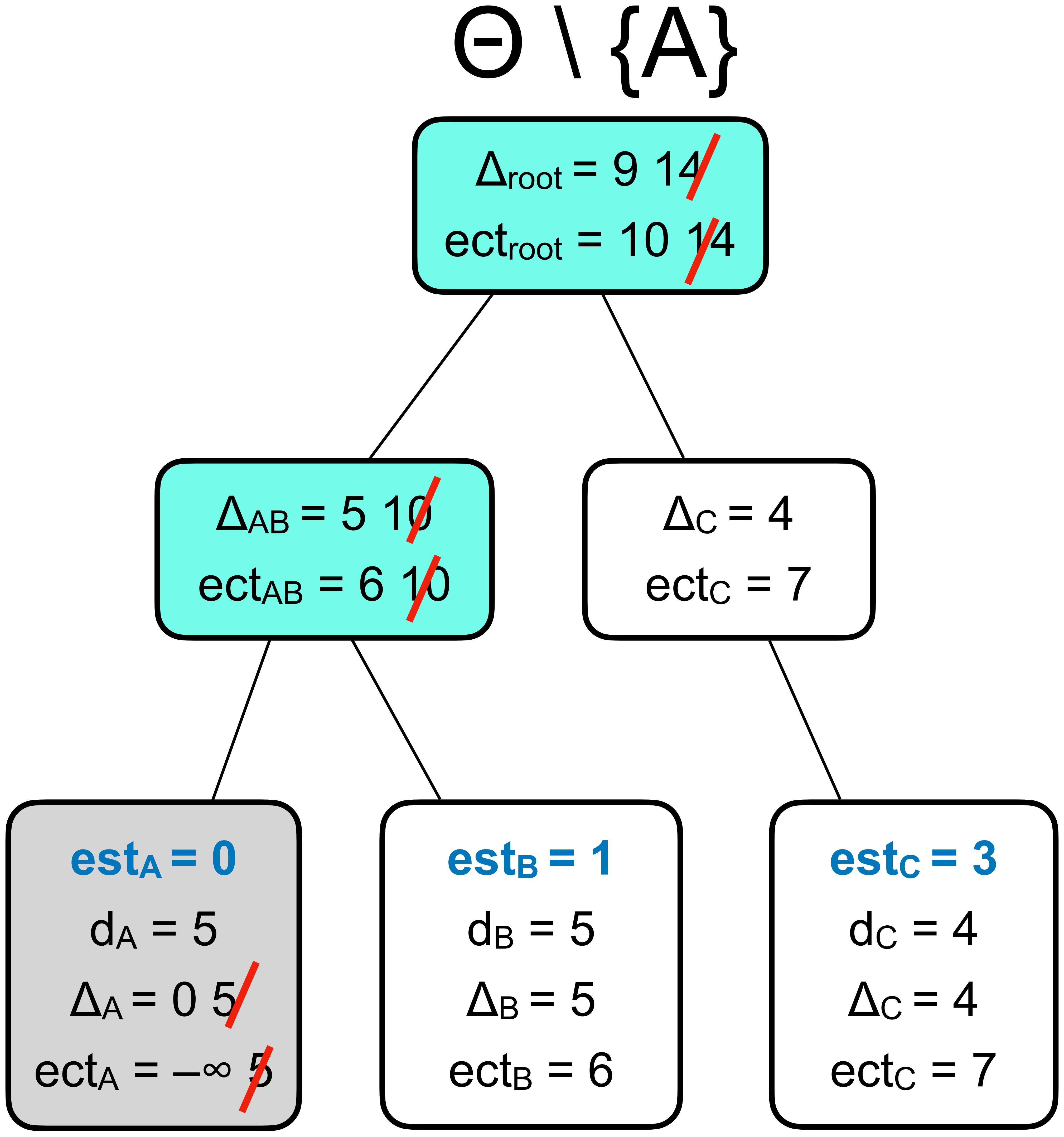
Second iteration: A is considered

All activities are already in the Θ-Tree



Not last filtering with Θ-Tree, an example

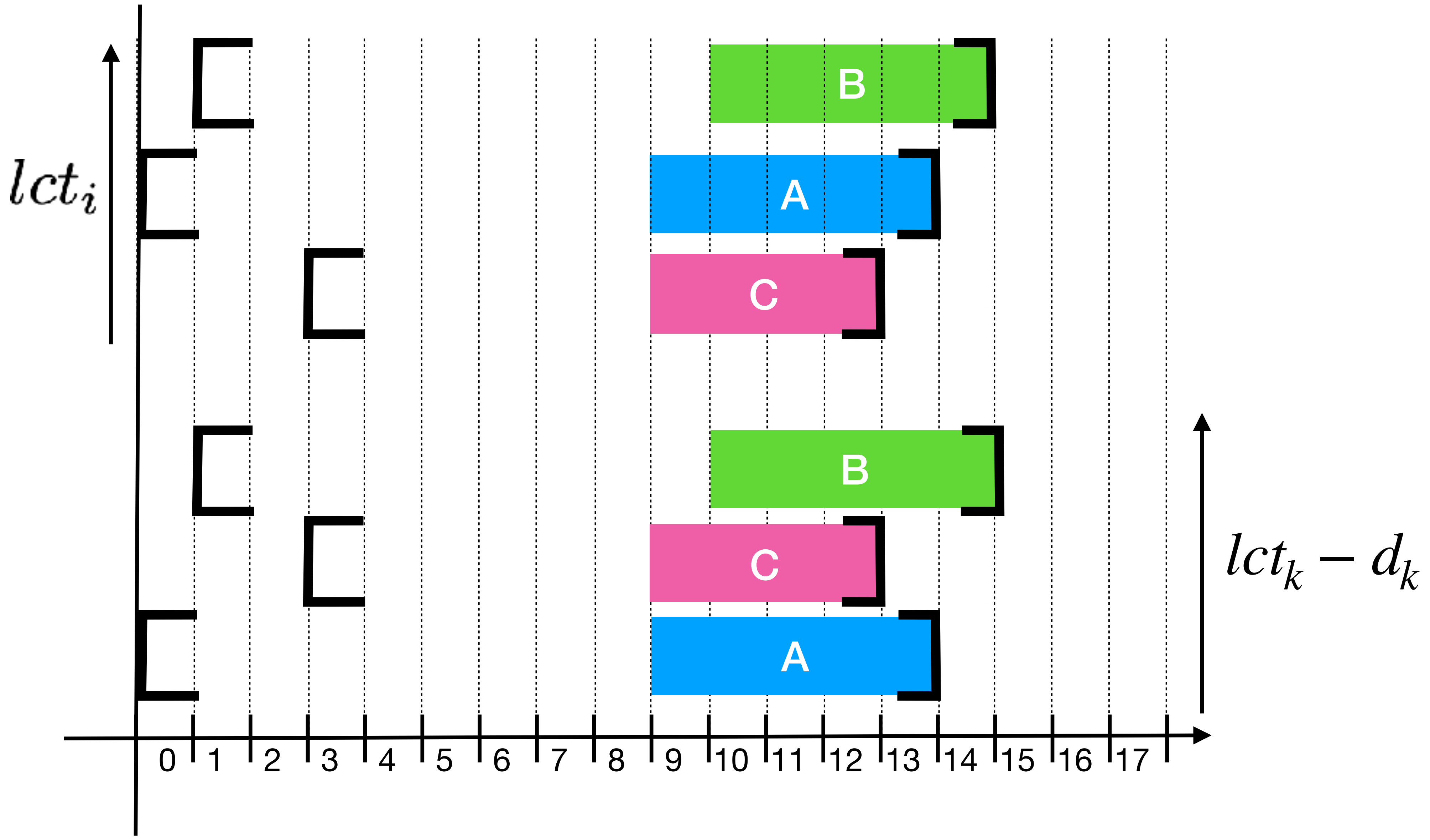




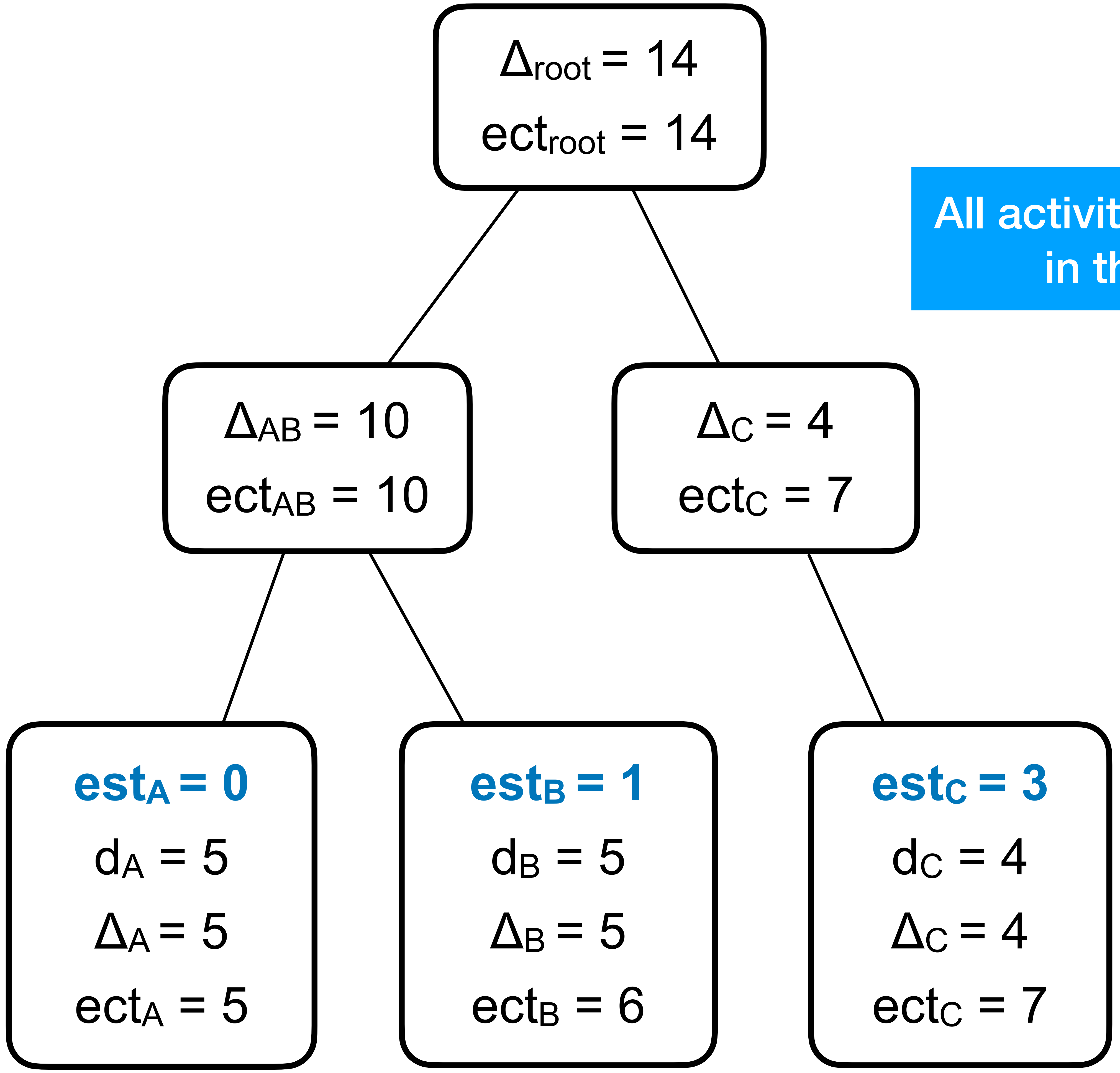

```
NotLast(T={1..n}) {
  ...
  ...
  Θ ← Θ-Tree.init({1..n})
  for (i ← Tlct) { // i ← A
    while (lcti > lctk-dk) {
       Θ.insert(k) // O(log n) time
       j ← k // lctj-dj = max {lctk - dk : k ∈ NLSet(T,i)}
       k ← ite.next()
    }
    if (ectΘ\i > lcti-di) { // ectΘ\A = 10 and lctA-dA = 9
       lct'i ← min(lcti, lctj-dj)
    }
 }
 lcti ← lct'i, ∀i∈T
}
```

$lct'_A = \min(lct_A, lct_B - d_B) = 10$

Not last filtering with Θ-Tree, an example



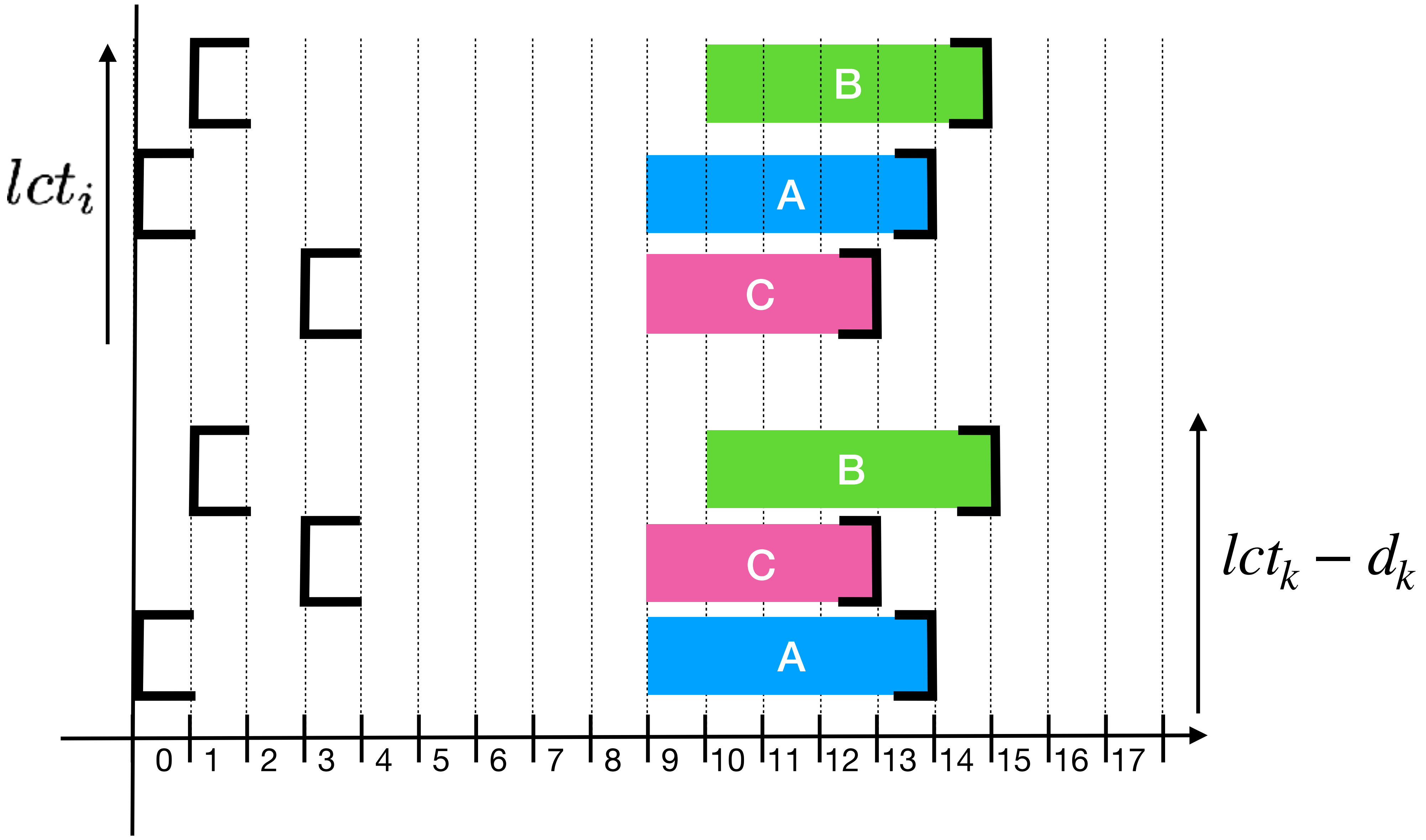
Third iteration: B is considered

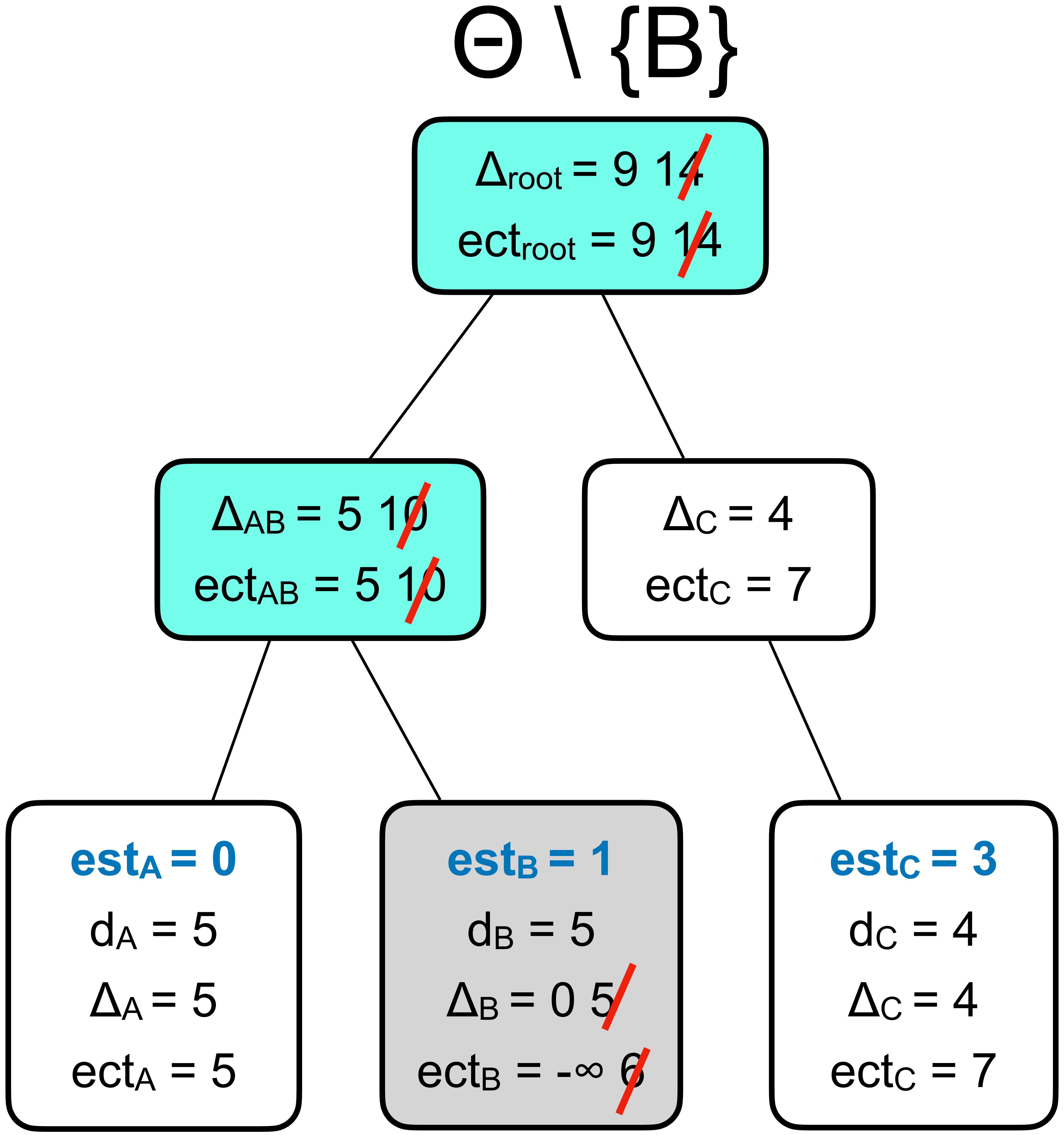


All activities are already in the Θ-Tree

```
NotLast(T={1..n}) {
  ...
  ...
  Θ ← Θ-Tree.init({1..n})
  for (i ← Tlct) { // i ← B
    while (lcti > lctk-dk) {
       Θ.insert(k) // O(log n) time
       j ← k // lctj-dj = max {lctk - dk : k ∈ NLSet(T,i)}
       k ← ite.next()
    }
    if (ectΘ\i > lcti-di) { // O(log n) time
       lct'i ← min(lcti, lctj-dj)
    }
  }
 lcti ← lct'i, ∀i∈T
}
```

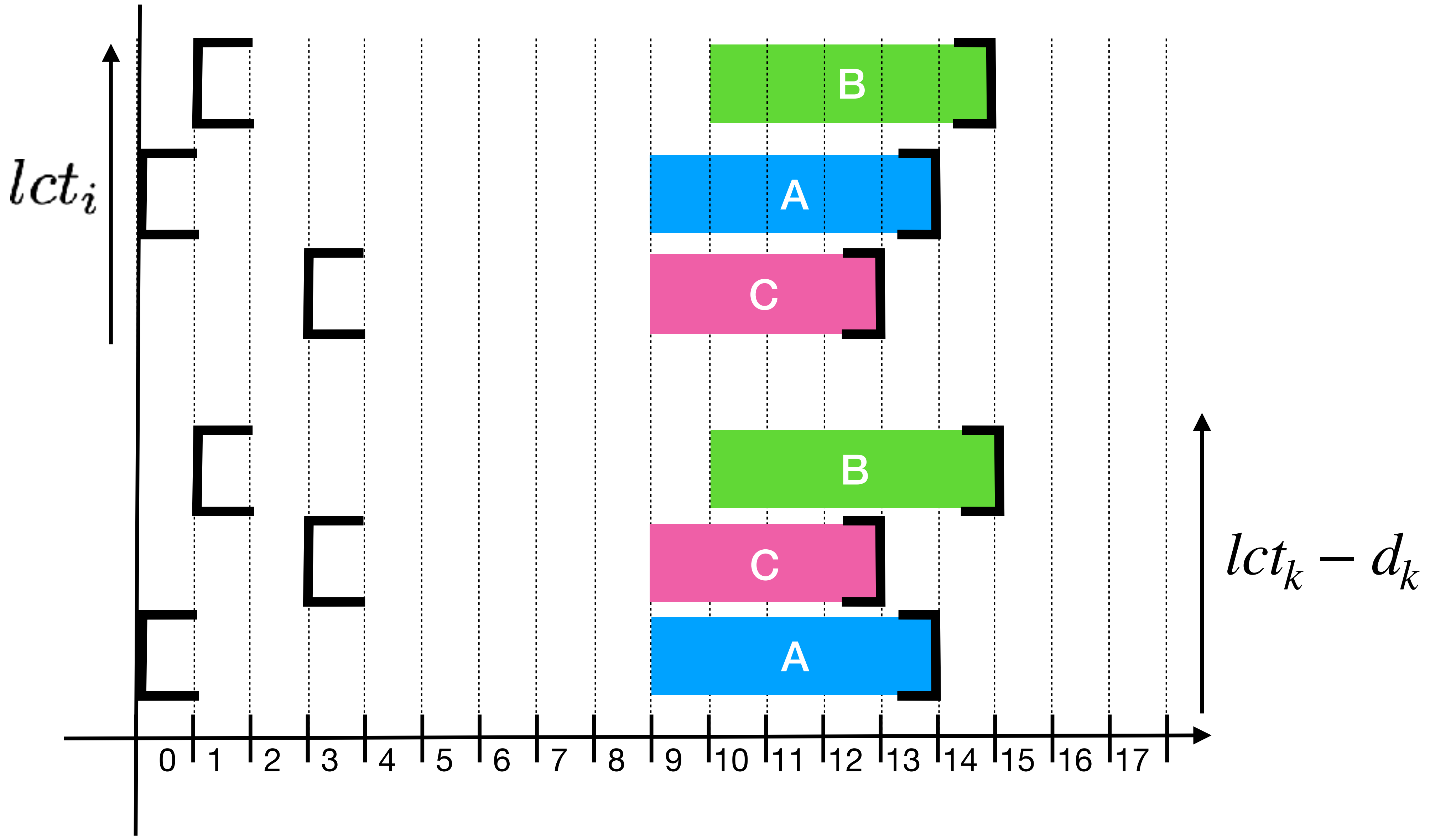
Not last filtering with Θ-Tree, an example

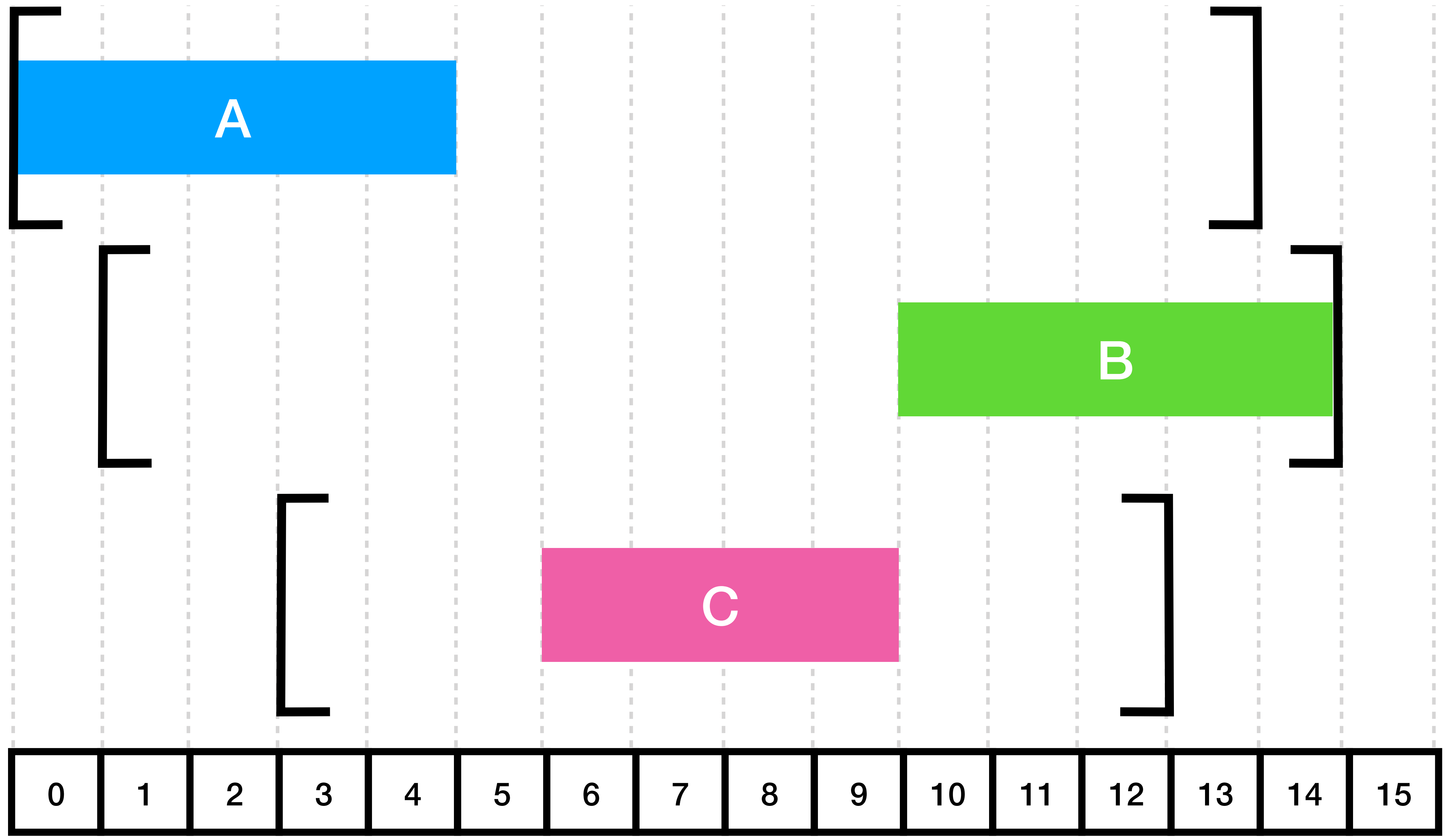





```
NotLast(T={1..n}) {
  ...
  ...
  Θ ← Θ-Tree.init({1..n})
  for (i ← T_lct) { // i ← B
    while (lct_i > lct_k-d_k) {
      Θ.insert(k) // O(log n) time
      j ← k // lct_j-d_j = max {lct_k - d_k : k ∈ NLSet(T,i)}
      k ← ite.next()
    }
    if (ect_Θ\i > lct_i-d_i) { // ect_Θ\B = 9 and lct_B-d_B = 10
      lct'_i ← min(lct_i, lct_j-d_j)
    }
  }
  lct_i ← lct'_i, ∀i∈T
}
```

Not last filtering with Θ-Tree, an example

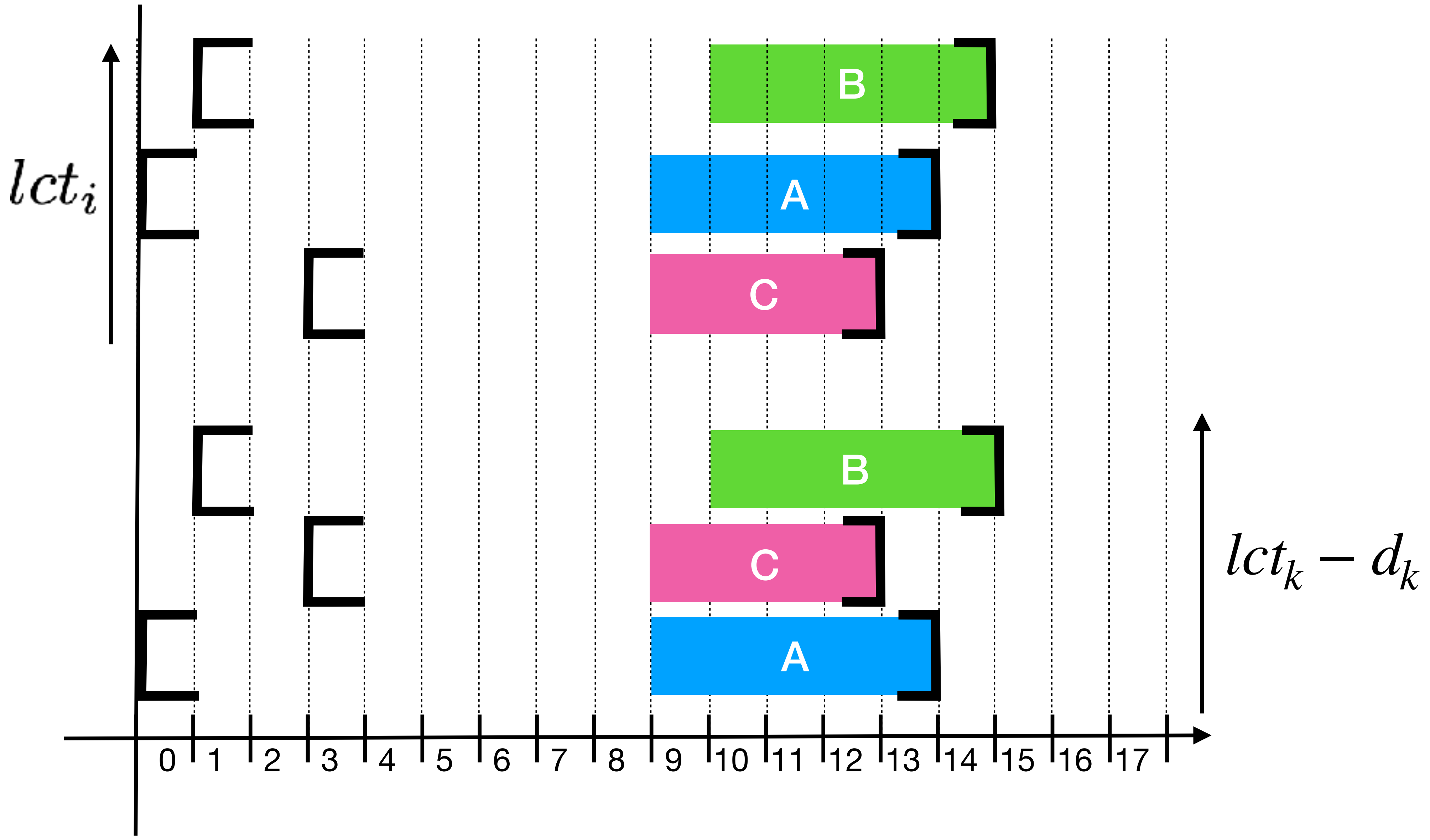


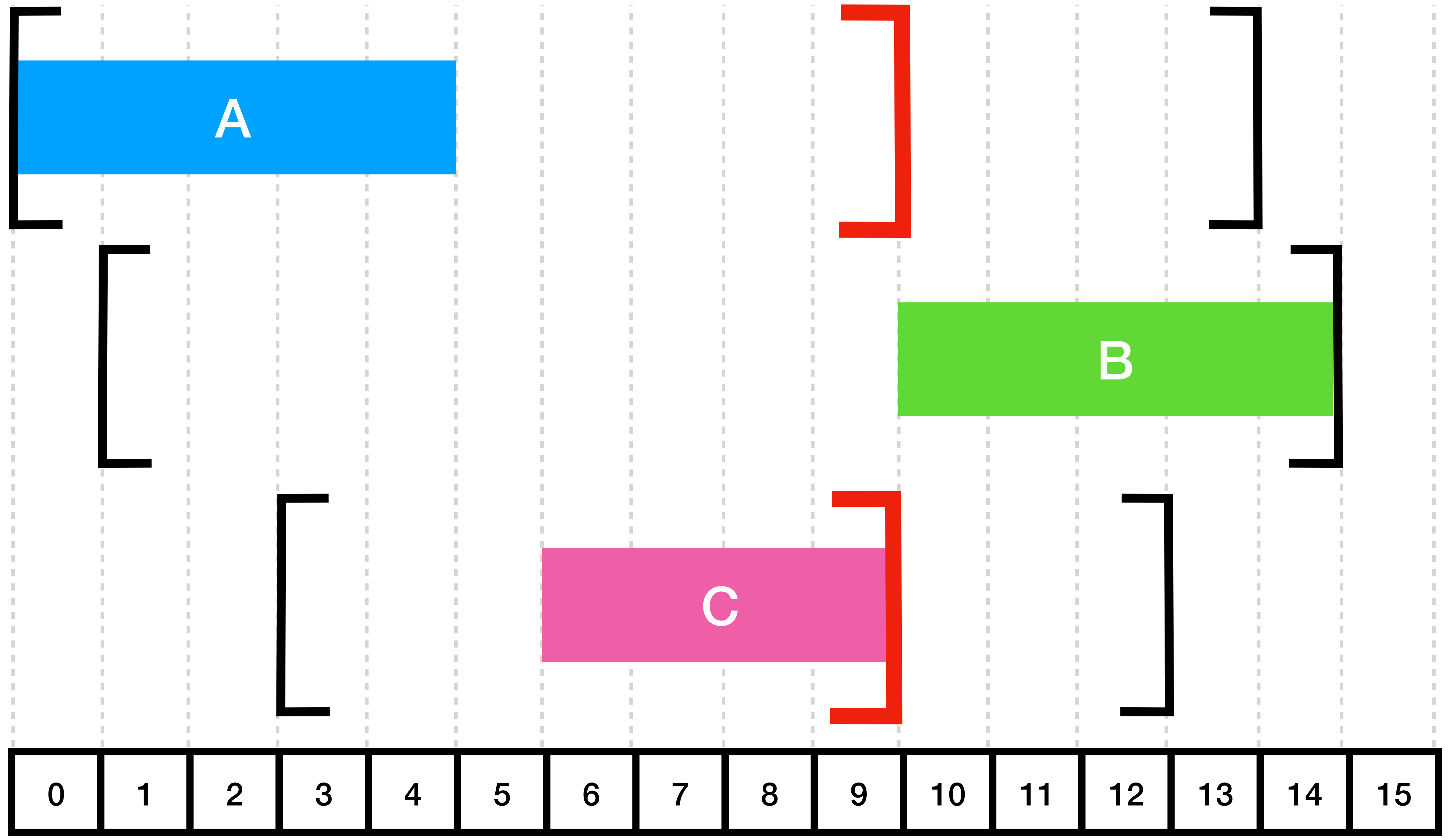



```
NotLast(T={1..n}) {
  ...
  ...
  Θ ← Θ-Tree.init({1..n})
  for (i ← Tlct) { // i ← B
    while (lcti > lctk-dk) {
       Θ.insert(k) // O(log n) time
       j ← k // lctj-dj = max {lctΩ - dΩ : Ω ⊆ NLSet(T,i)}
       k ← ite.next()
    }
    if (ectΘ\i > lcti-di) { // ectΘ\B = 9 and lctB-dB = 10
       lct'i ← min(lcti, lctj-dj)
    }
 }
 lcti ← lct'i, ∀i∈T
}
```

$lct_C = 10$
$lct_A = 10$
$lct_B = 15$

Not last filtering with Θ-Tree, an example





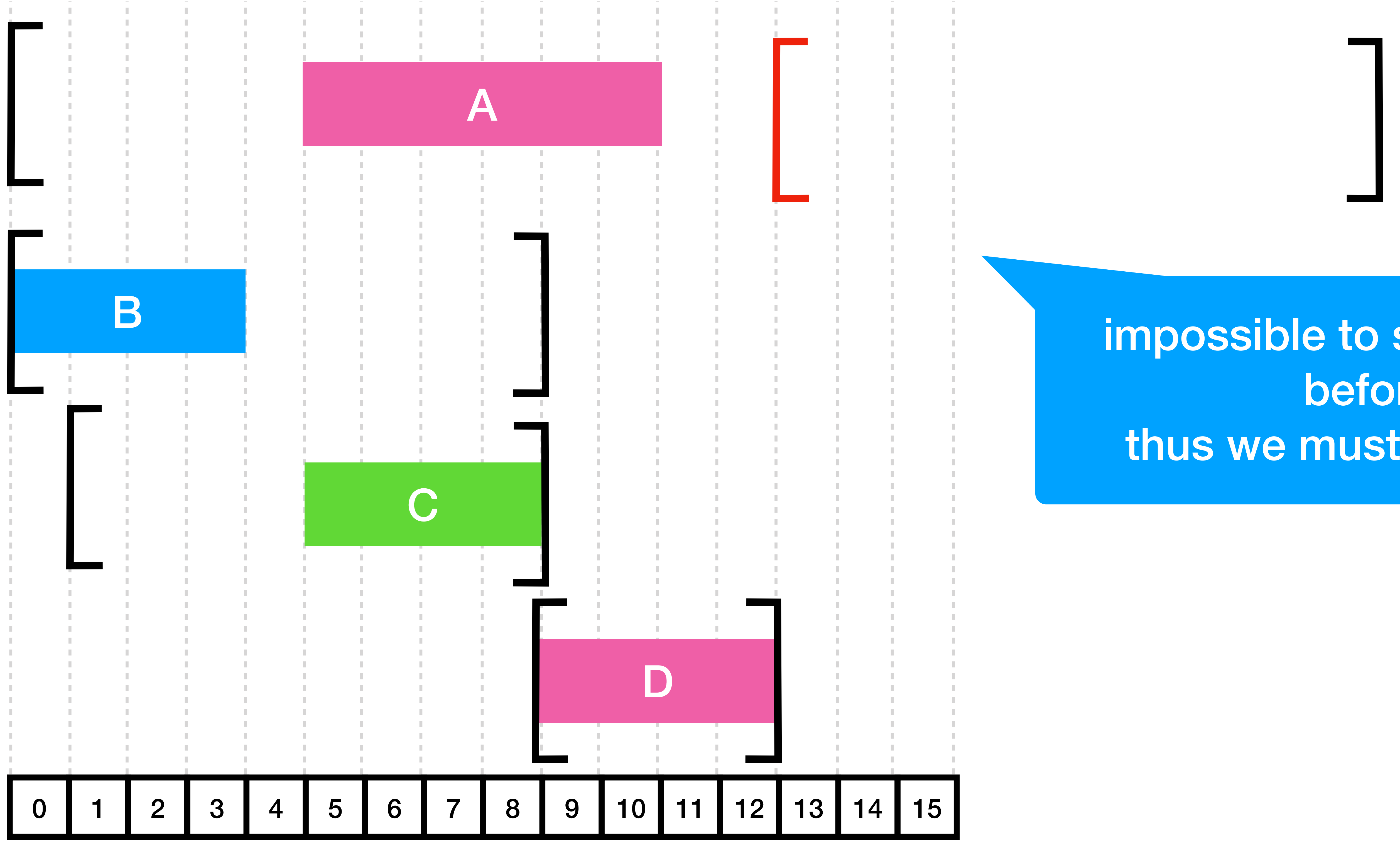

```
NotLast(T={1..n}) {
  ...
  ...
  Θ ← Θ-Tree.init({1..n})
  for (i ← Tlct) { // i ← B
    while (lcti > lctk-dk) {
      Θ.insert(k) // O(log n) time
      j ← k // lctj-dj = max {lctΩ - dΩ : Ω ⊆ NLSet(T,i)}
      k ← ite.next()
    }
    if (ectΘ\i > lcti-di) { // ectΘ\B = 9 and lctB-dB = 10
      lct'i ← min(lcti, lctj-dj)
    }
 }
 lcti ← lct'i, ∀i∈T
}
```

lct_C = 10
lct_A = 10
lct_B = 15

Edge Finder

Edge Finding

- $\forall \Omega \subset T, \forall i \in T \backslash \Omega$ = arbitrary non-empty subset of T
- $est_{\Omega \cup i} + d_{\Omega \cup i} > lct_{\Omega} \Rightarrow \Omega \ll i \rightsquigarrow est_i \leftarrow \max\{est_i, ect_{\Omega}\}$ (EF)
- i must be scheduled after the set Ω



impossible to schedule {A,B,C,D} before $lct_{\{BCD\}}$
thus we must have {B,C,D} « A

Edge Finding

- Reformulation of EF for easier implementation

$\forall\, j \in T,\ \forall\, i \in T \setminus LCut(T,j)$:

$LCut(T,j) = \{i \mid i \in T \;\&\; lct_i \leq lct_j\}$

$$ect_{LCut(T,j) \cup i} > lct_j \Rightarrow LCut(T,j) \ll i$$

$$\rightsquigarrow est_i \leftarrow \max\{est_i, ect_{LCut(T,j)}\} \quad (EF')$$

- Implementation using Θ-tree considering j and i wrt LCut(T,j)
  - $\Theta = LCut(T,j)$
  - Θ-Tree.insert(i), check if $ect_\Theta > lct_j$
  - Θ.remove(i)

O(log n) for testing one (i,j)
O(n^2 log n) overall => too slow!

Θ-Λ-Tree = generalization of Θ-Tree

white

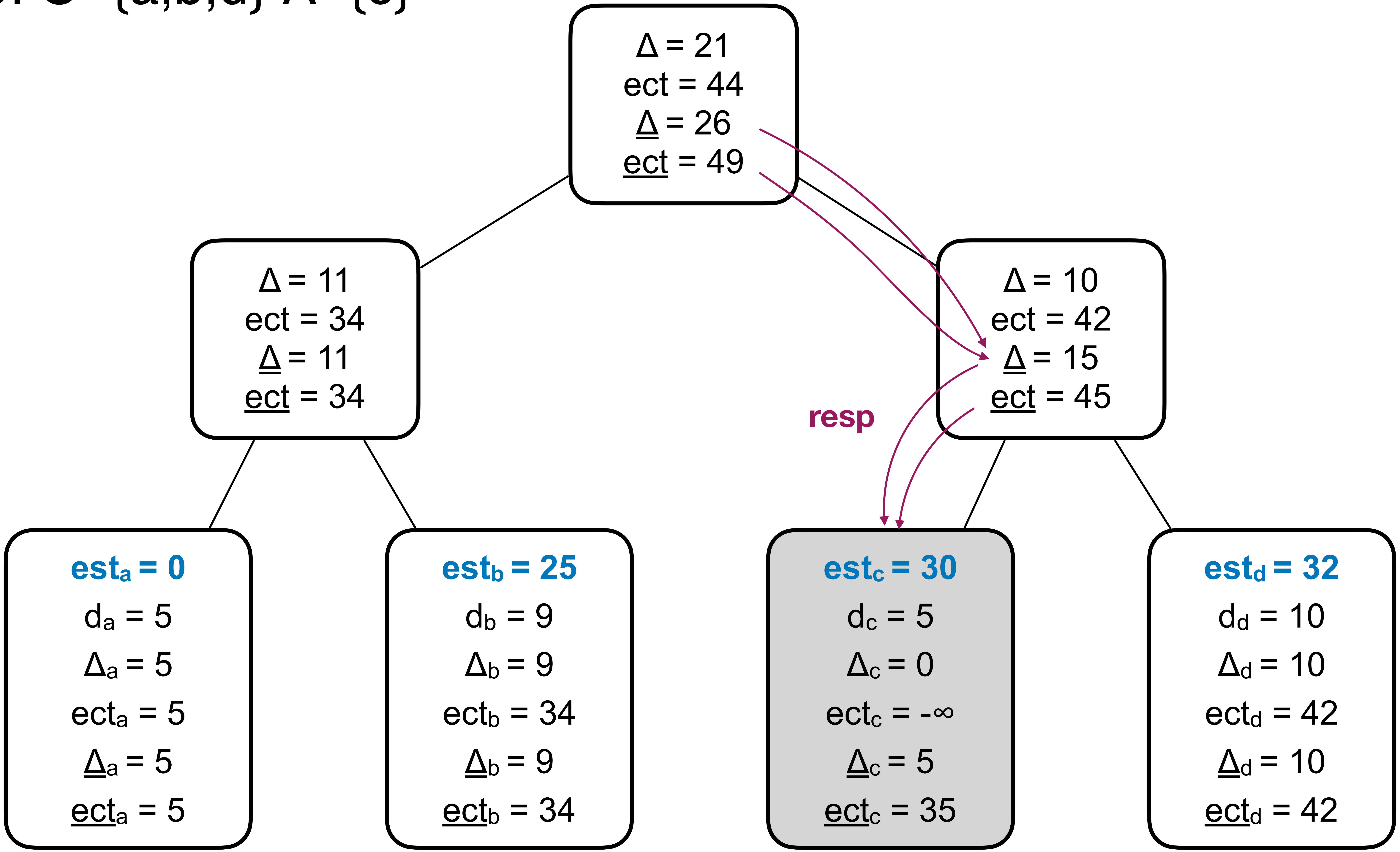
gray

Θ and Λ disjoint sets: $\Theta \cap \Lambda = \varnothing$

- $\underline{ect}(\Theta\text{-}\Lambda) = \max(\{ect_\Theta\},\{ect_{\Theta \cup i} : i \in \Lambda\})$
    - earliest completion time if at most one gray activity used
- New values stored in the nodes (in addition to Δ_v & ect_v)
    - $\underline{\Delta}_v = \max \{p_{\Theta'} \mid \Theta' \subseteq Leaves(v) \;\&\; |\Theta' \cap \Lambda| \leq 1\}$
    - $\underline{ect}_v = \underline{ect}_{Leaves(v)} = \max \{est_{\Theta'} + p_{\Theta'} \mid \Theta' \subseteq Leaves(v) \;\&\; |\Theta' \cap \Lambda| \leq 1\}$
- Update rule
    - $\underline{\Delta}_v = \max \{\underline{\Delta}_{left(v)} + \Delta_{right(v)}, \Delta_{left(v)} + \underline{\Delta}_{right(v)}\}$
    - $\underline{ect}_v = \max \{\underline{ect}_{right(v)}, \underline{ect}_{left(v)} + \Delta_{right(v)}, ect_{left(v)} + \underline{\Delta}_{right(v)}\}$

Example

- Θ-Λ-Tree: Θ={a,b,d} Λ={c}



Responsible Activities

- For each node v we can also compute the gray activity responsible for $\underline{\Delta}_v$ or $\underline{ect}_v$
- Leaf nodes:
  - $resp_{\underline{\Delta}}(i)$ = i if i is gray, undef otherwise
  - $resp_{\underline{ect}}(i)$ = i if i is gray, undef otherwise
- Internal nodes:
  - $resp_{\underline{\Delta}}(v) = resp_{\underline{\Delta}}(left(v))$ if $\underline{\Delta}_v = \underline{\Delta}_{left(v)} + \Delta_{right(v)}$,

    $resp_{\underline{\Delta}}(right(v))$ otherwise
  - $resp_{\underline{ect}}(v) = resp_{\underline{ect}}(right(v))$ if $\underline{ect}_v = \underline{ect}_{right(v)}$

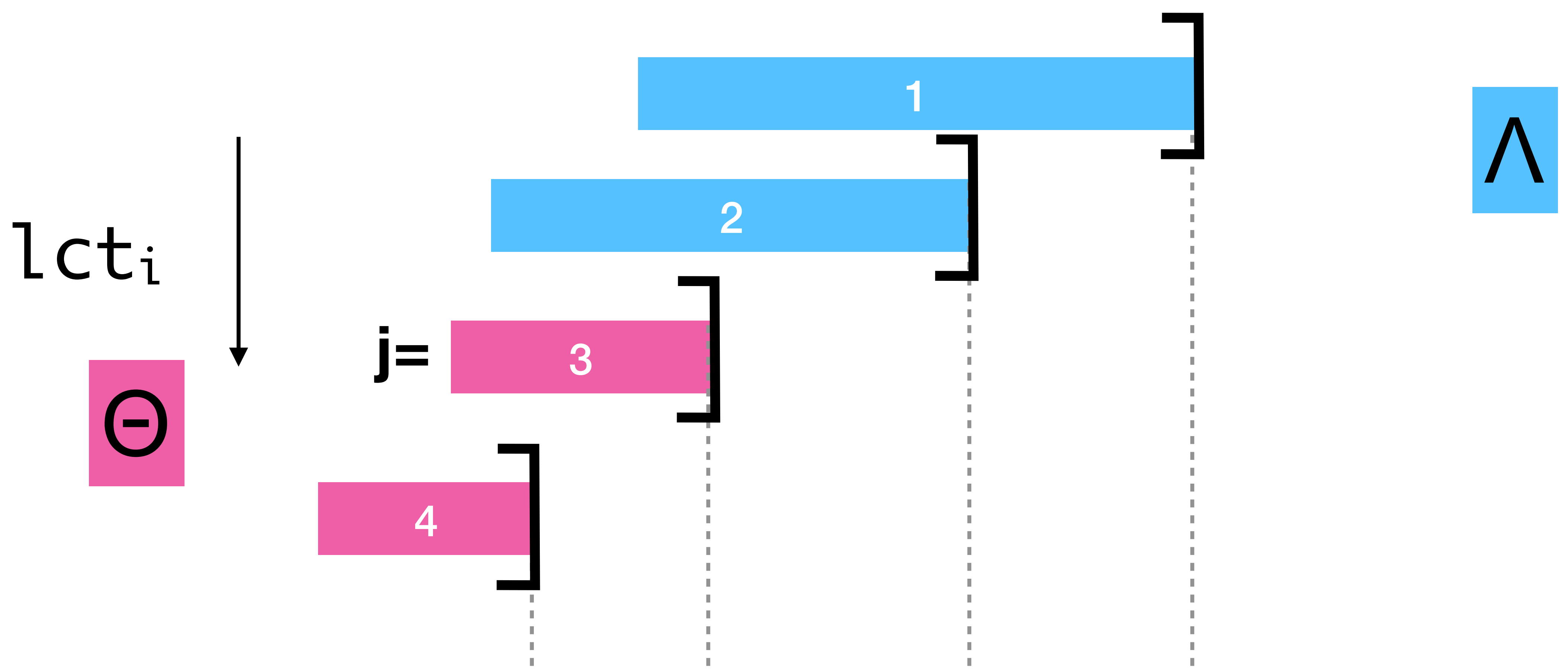
    $resp_{\underline{ect}}(left(v))$ if $\underline{ect}_v = \underline{ect}_{left(v)} + \Delta_{right(v)}$

    $resp_{\underline{\Delta}}(right(v))$ if $\underline{ect}_v = ect_{left(v)} + \underline{\Delta}_{right(v)}$

Complexities

Operation	Time Complexity
$(\Theta,\ \Lambda) := (\emptyset,\ \emptyset)$	$\mathcal{O}(1)$
$(\Theta,\ \Lambda) := (T,\ \emptyset)$	$\mathcal{O}(n \log n)$
$(\Theta,\ \Lambda) := (\Theta \setminus \{i\},\ \Lambda \cup \{i\})$	$\mathcal{O}(\log n)$
$\Theta := \Theta \cup \{i\}$	$\mathcal{O}(\log n)$
$\Lambda := \Lambda \cup \{i\}$	$\mathcal{O}(\log n)$
$\Lambda := \Lambda \setminus \{i\}$	$\mathcal{O}(\log n)$
$\overline{\mathrm{ect}}(\Theta, \Lambda)$	$\mathcal{O}(1)$
ect_{Θ}	$\mathcal{O}(1)$

Edge Finding: The big picture




```
while (ect(Θ-Λ) > lctⱼ) {
  i ← resp_ect(Θ-Λ)
  estᵢ ← max{estᵢ,ect_Θ}
  Λ ← Λ\i // O(log n)
}
```

Retrieve the activity of Λ responsible

Edge Finding Algorithm

```
EdgeFinding(T={1..n}) {
  (Θ,Λ) = (T,∅) // O(n log n) time
  Tlct ← sortZA([1..n],sortKey = lct) // O(n log n) time
  ite ← iterator(Tlct)
  j = ite.next()
  while (ite.hasNext()) {
    if (ectΘ >  lctj) throw InconsistencyException // overload
    (Θ,Λ) = (Θ\j,Λ∪j) // O(log n) time
    j ← ite.next()
    while (ect(Θ-Λ) > lctj) { // O(1) time
       i ← respect(Θ-Λ)
     esti ← max{esti,ectΘ}
       Λ ← Λ\i // O(log n) time
    }
  }
}
```

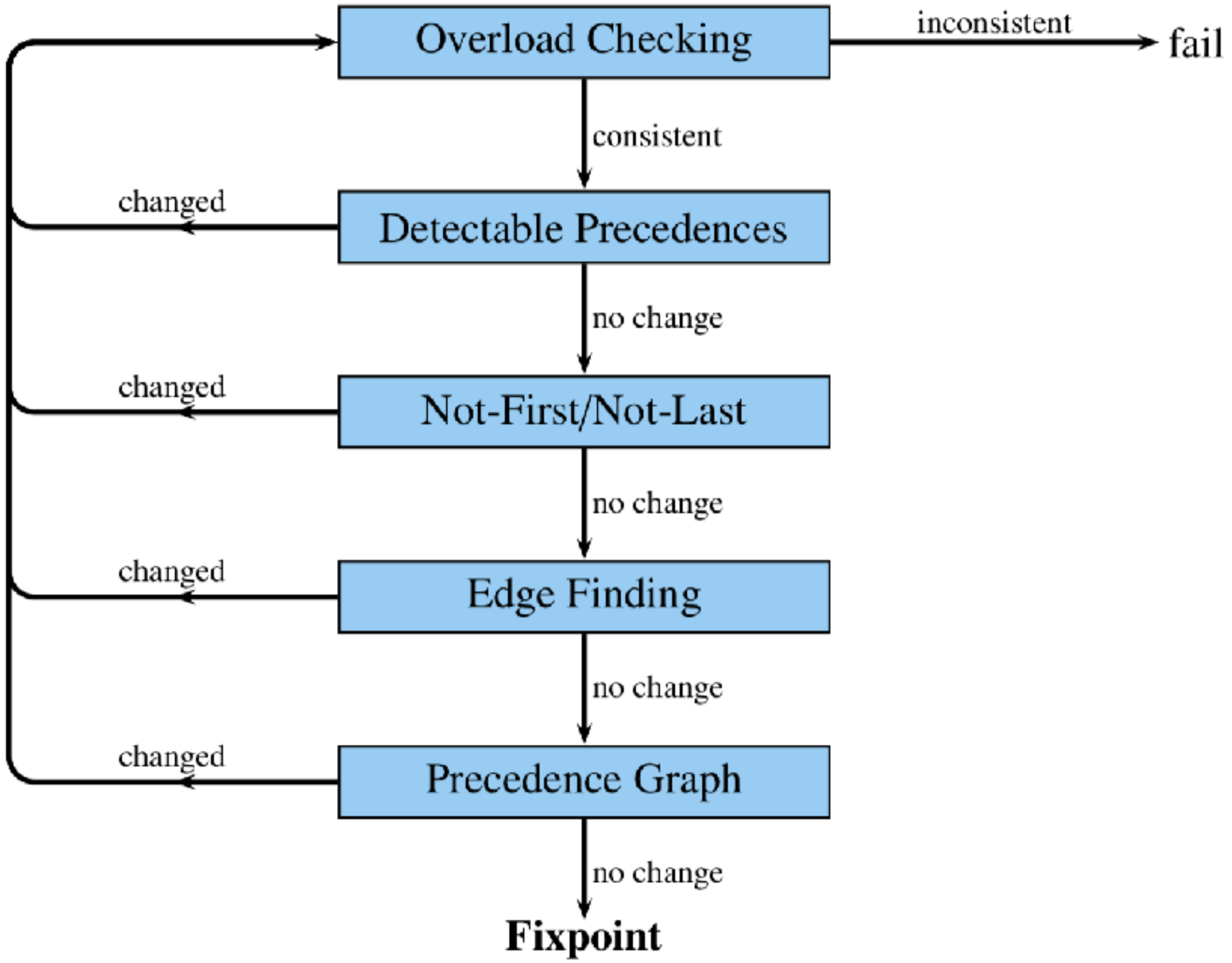
Executed at most n times

Fix-point

Reminder on Idempotency

Putting it all together

- None of the algorithms above is idempotent.
- According to Petr Vilím (see next slide),
  the following order for fixpoint computation is very efficient:



Bibliography

- Most of the notation, examples, … come from
  Petr Vilím’s PhD thesis (https://vilim.eu/petr/disertace.pdf),
  where all the proofs omitted here can be found.
- This thesis had a big impact on CP solvers because most of the algorithms for a disjunctive resource introduced by Petr Vilím take $O(n \log n)$ time instead of $O(n^2)$ or $O(n^3)$.