Université de Nantes

Dynamic Trees and Log-lists

Irena Rusu
Université de Nantes, LINA
Irena.Rusu@univ-nantes.fr

Outline

Å Motivations for an Array-List (but not the Java Collection é)

Å Dynamic Trees

Å A Sequence is a Filiform Tree

Å A Log-List is a Filiform tree too é but Needs Adjustments

Å Conclusion

Outline

- Motivations for an Array-List (but not the Java Collection …)
- Fundamentals: Dynamic Trees
- Easy observation: A Sequence is a Filiform Tree
- A step further: A Log-List too … but Needs Adjustments
- Conclusion

Motivations for an Array-List

Problem. Improve the running time of the existing algorithm for sorting a permutation P by prefix reversals and prefix transpositions (explained below).

Prefix reversal

7+1=8

P: 7 4 5 2 8 9 1 3 6 10

P_8^{-1} needed

P: 7 4 5 2 | 8 9 | 1 3 6 10

Need to:

Å Reverse

Å Update some P_i^{-1}

P: 2 5 4 7 8 9 1 3 6 10

P_8^{-1} better using arrays, reverse better using double-linked lists, update ???

Motivations for an Array-List

Prefix transposition

P: 2 5 4 7 8 9 1 3 6 10

P_1^{-1}, P_8, P_4^{-1} needed

P: 2 5 4 | 7 8 9 1 | 3 6 10

Need to:

Å Transpose

Å Update many P_i

Å Update many P_i^{-1}

P: 7 8 9 1 2 5 4 3 6 10

P_1^{-1}, P_8, P_4^{-1} better using arrays, transpose better using double-linked lists, update ??

Motivations for an Array-List : An Idea

Goal:

- choose and perform a reversal or transposition in $O(\log n)$
- improve the running time of the sorting algorithm from $O(n^2)$ to $O(n \log n)$

P: 2 5 4 | 7 8 9 1 | 3 6 10

Use trees:

Initial step: the permutation is a big tree

Coloring step: cut the big tree into subtrees

Block operation: change trees order and/or left-right orientation, and globally modify P and P^{-1} values at the root

Final step: link subtrees

Motivations for an Array-List : An Idea

P: 2 5 4 | 7 8 9 1 | 3 6 10

Trees need to support:

Å Cut several edges

Å Link several subtrees

Å Left-right flipping

in O(log n)

Binary balanced trees :

Å Height balanced trees

Å Red-Black trees

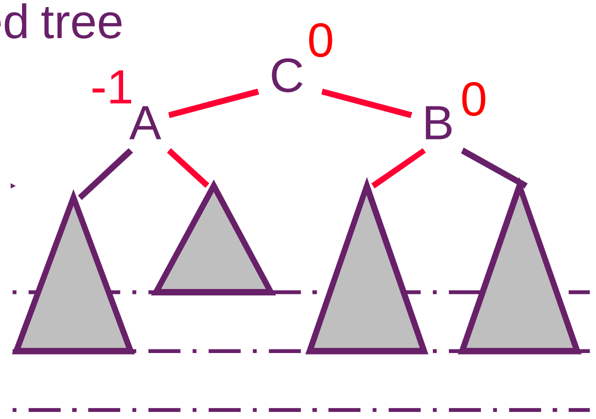
Å é

(Any other with height in O(log n), and which supports re-balancing)

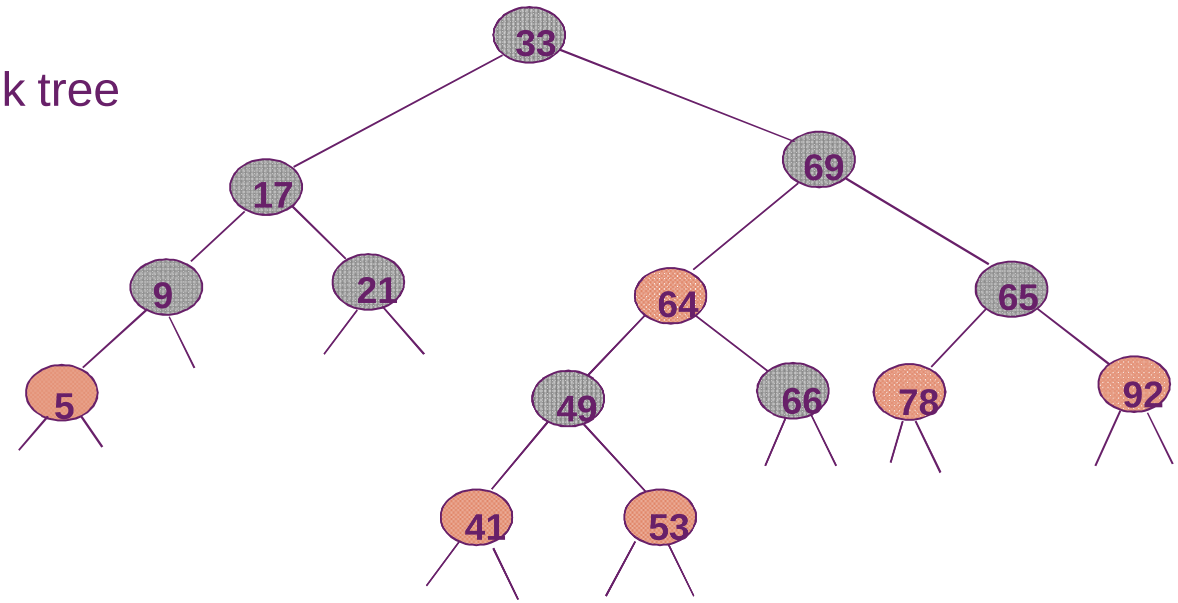
Rappels

Height-balanced tree

(Re-)balance



Red-black tree



Motivations for an Array-List : An Idea

P:	2	5	4	7	8	9	1	3	6	10

Trees need to support:

Å Cut several edges

Å Link several subtrees

Å Left-right flipping

in O(log n)

→

Binary balanced trees :

Å Height balanced trees

Å Red-Black trees

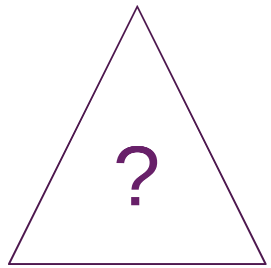
Å é

(Any other with height in O(log n), and which supports re-balancing)

é but also

Å global modifications of P and P^{-1} values

→



Outline

- Motivations for an Array-List (but not the Java Collection)
- Dynamic Trees
- Easy observation: A Sequence is a Filiform Tree
- A step further: A Log-List too … but Needs Adjustments
- Conclusion

Dynamic trees: Introduction (1)

Daniel D. Sleator and Robert E. Tarjan (1983)

Data structure proposed to maintain a forest of vertex disjoint rooted trees under link (add an edge) and cut (delete an edge) operations between trees, in O(log n) each.

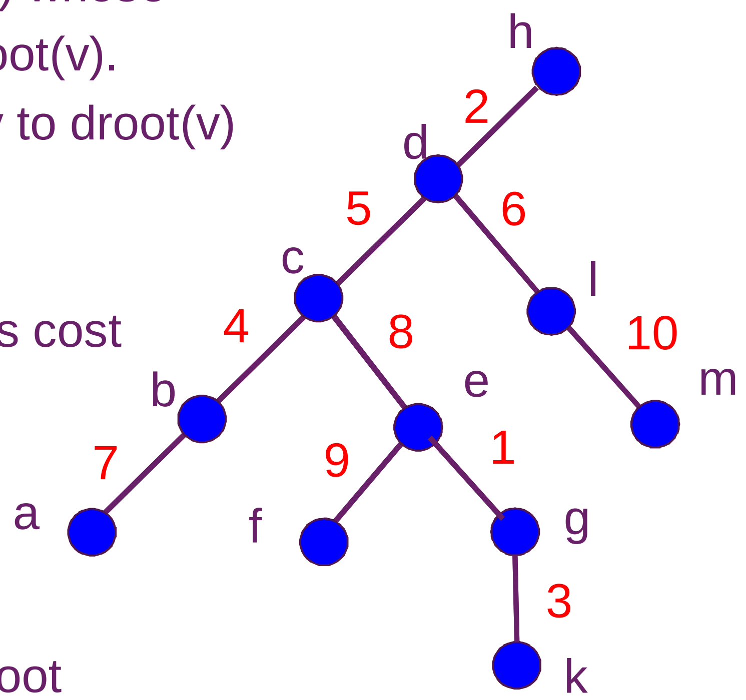
Applications:

- maximum flow problem and variants (Sleator, Tarjan, 1983)
- the Least Common Ancestor problem (Sleator, Tarjan, 1983)
- the transshipment problem (Sleator, Tarjan, 1983)
- the string matching problem in compressed strings (Farach, Thorup, 1995)
- é

Dynamic trees: Introduction (2)

More precisely, do in O(log n), given the forest of dynamic trees:

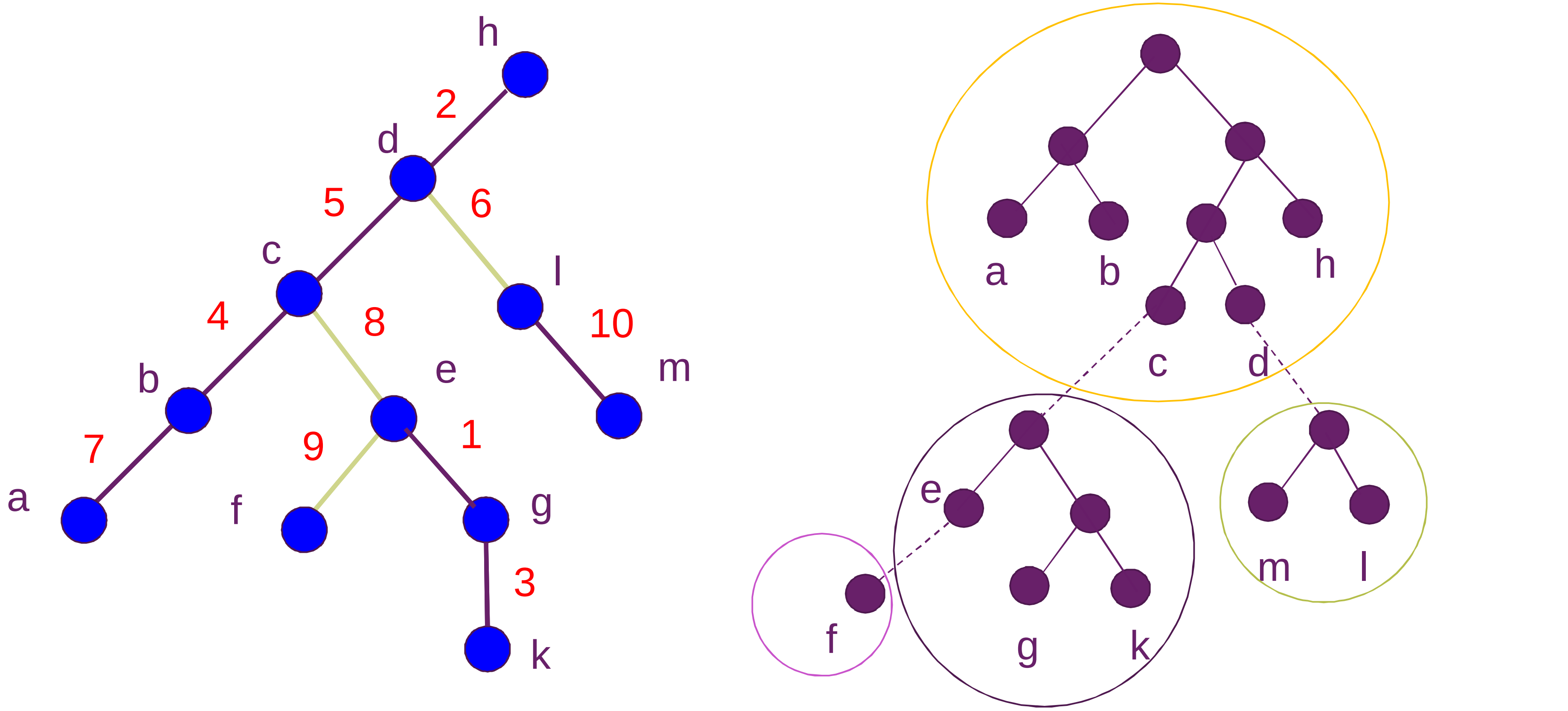
- dparent(v): return the parent of v in its tree, or null
- droot(v): return the root of the tree containing v
- dcost(v): return the cost of the edge (v,dparent(v))
- dmincost(v): return the vertex w closest to droot(v) whose
  dcost(w) is minimum on the path from v to droot(v).
- dupdate(v,u): add u to all costs on the path from v to droot(v)
- dlink(v,w,u): add edge (v,w) of cost u assuming
  v = droot(v), thus making w the parent of v
- dcut(v): delete the edge (v,parent(v)) and return its cost
- devert(v): make v become the root of its tree



Note. All these are operations on paths towards the root

Dynamic trees: Ideas

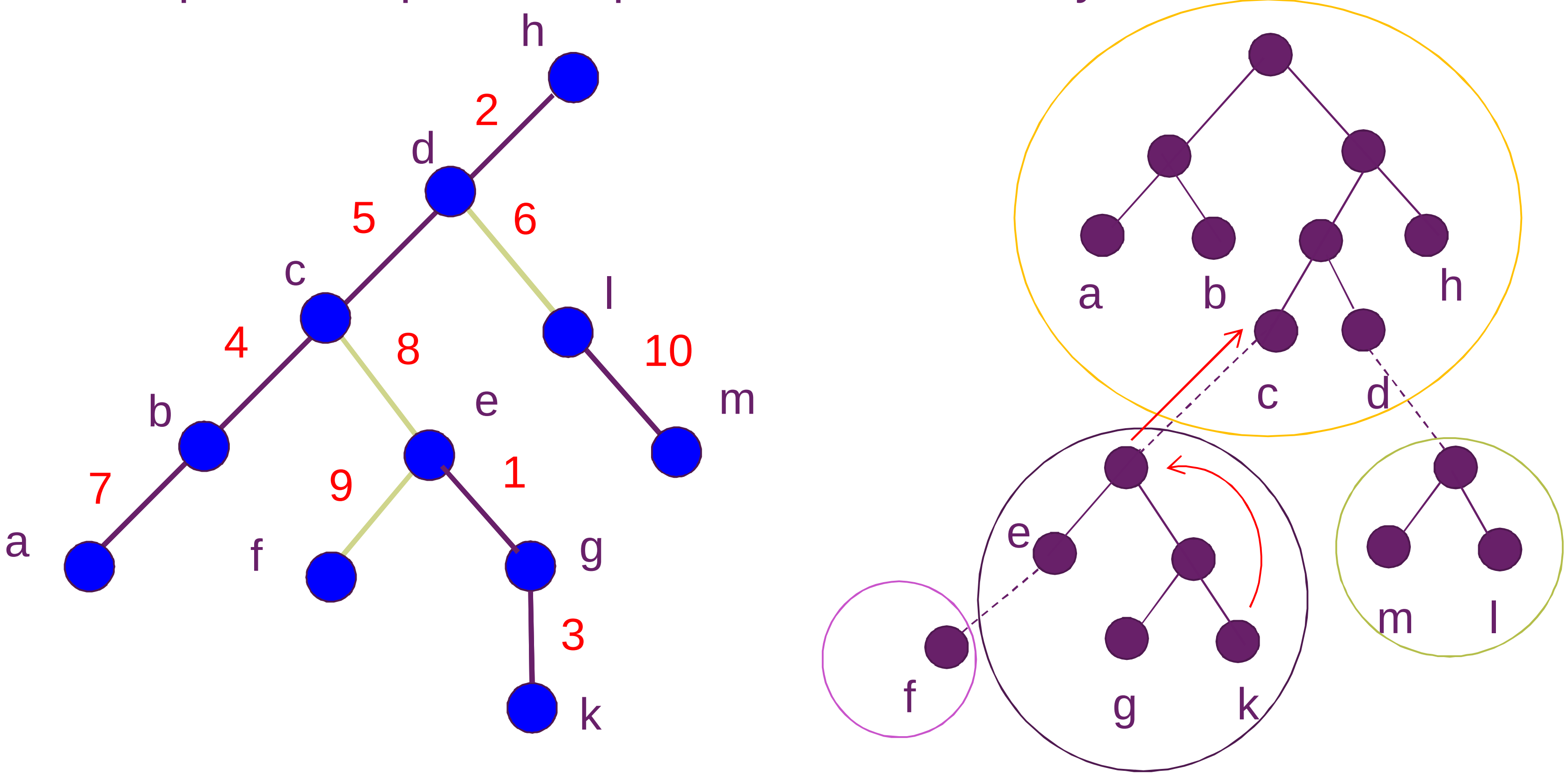
1. A tree is partitionned into « solid » paths going towards the root (edges are « solid » or « dashed »)



2. Each « solid » path is represented as a (particular) binary tree.

Dynamic trees: Ideas

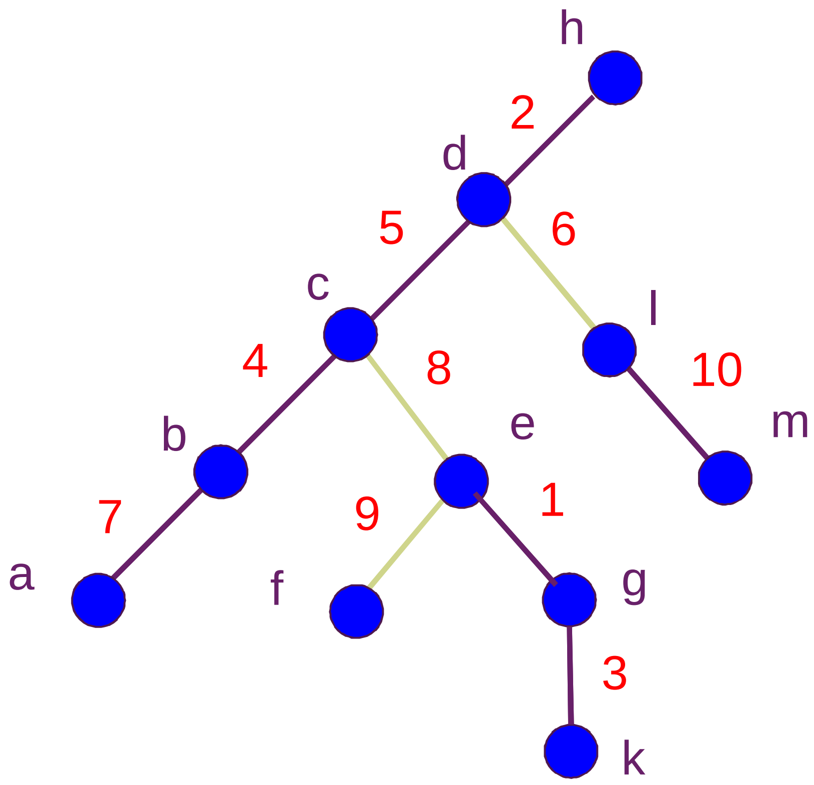
3. Operations on paths towards the root: turn the concerned path into a solid path, and perform operations on its binary tree

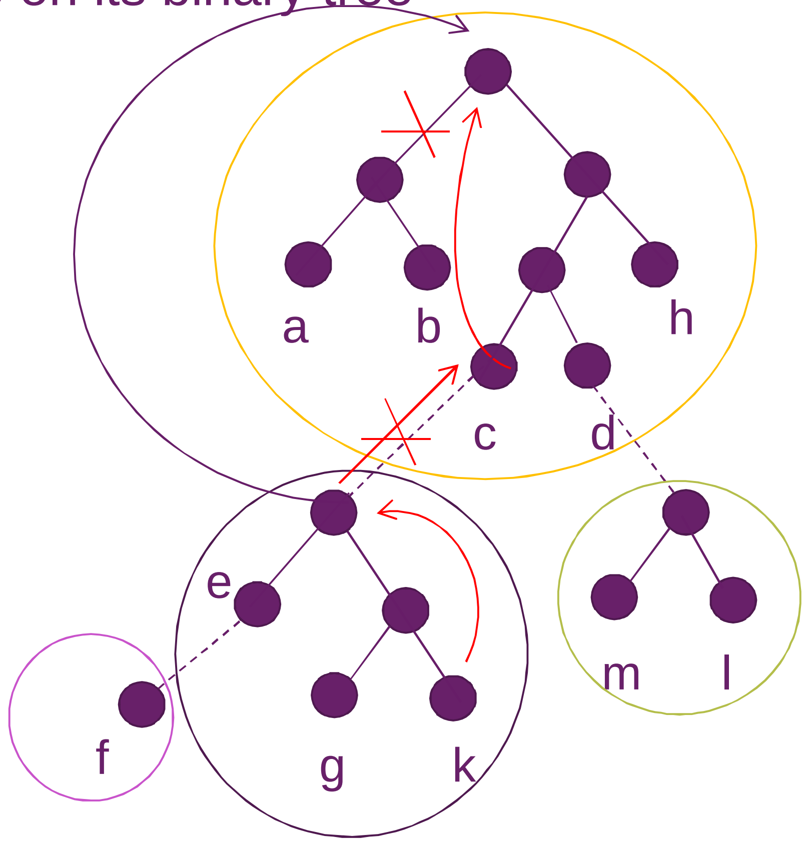


Here, the path from k to the root.

Dynamic trees: Ideas

3. Operations on paths towards the root: turn the concerned path into a solid path, and perform operations on its binary tree

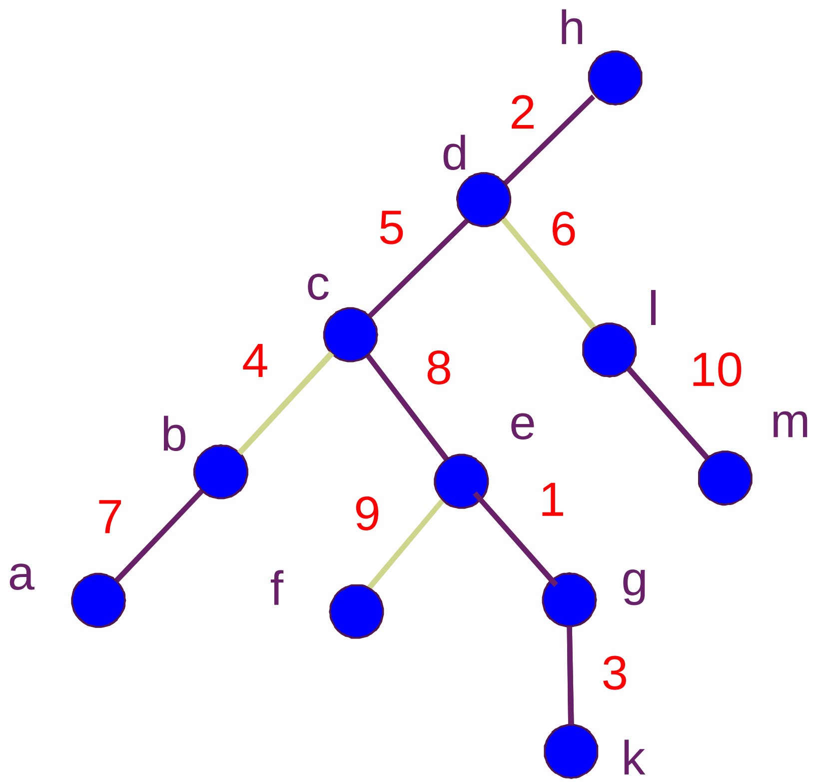


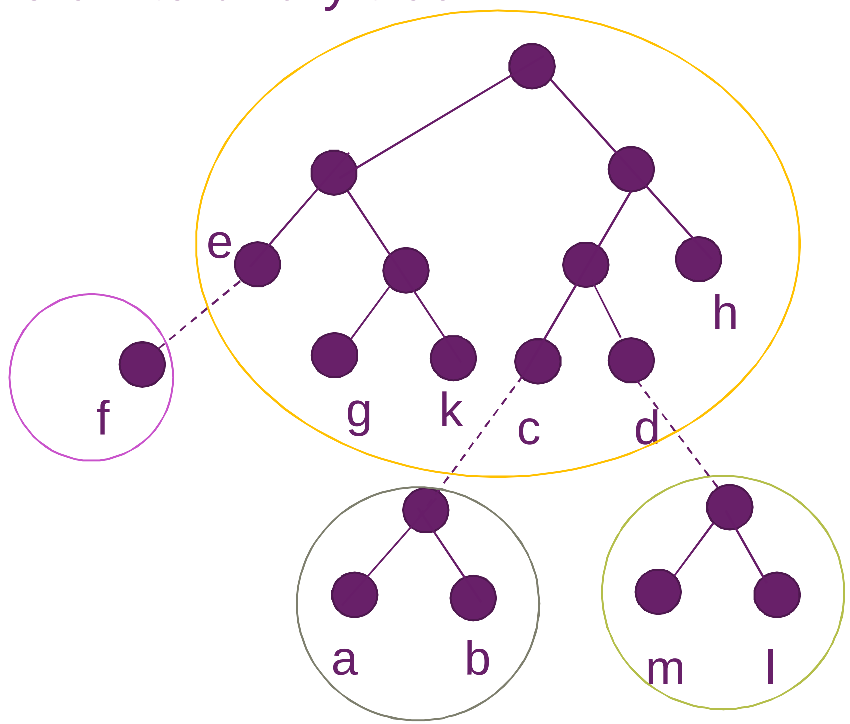


Here, the path from k to the root.

Dynamic trees: Ideas

3. Operations on paths towards the root: turn the concerned path into a solid path, and perform operations on its binary tree





Here, the path from k to the root.

Splice : switches one solid and one dashed arc
Needs O(log n) for binary balanced trees.
Expose : builds the entire path, uses t splices
Needs O(t log n) for binary balanced trees.

Dynamic trees: Choose the appropriate solid paths and binary trees

Trees / Solid paths	Standard balanced binary trees	Locally biased binary trees/ Splay trees	Globally biased binary trees
Initial: Arbitrary Later: As resulting from the operations performed	$O(\log^2 n)$ amortized	$O(\log n)$ amortized	
Initial and Later: Defined by the structure of the tree (Heavy/light edges*)			$O(\log n)$

Running time of expose

*(v,father(v)) is a heavy edge in the dynamic tree if 2*size(v)>size(father(v)), where size(v)=#nodes in the tree rooted at v

Dynamic trees: Choose the appropriate solid paths and binary trees

Trees / Solid paths	Standard balanced binary trees	Locally biased binary trees/ Splay trees	Globally biased binary trees
Initial: Arbitrary Later: As resulting from the operations performed	$O(\log^2 n)$ amortized	$O(\log n)$ amortized	
Initial and Later: Defined by the structure of the tree (Heavy/light edges*)			$O(\log n)$

Running time of expose

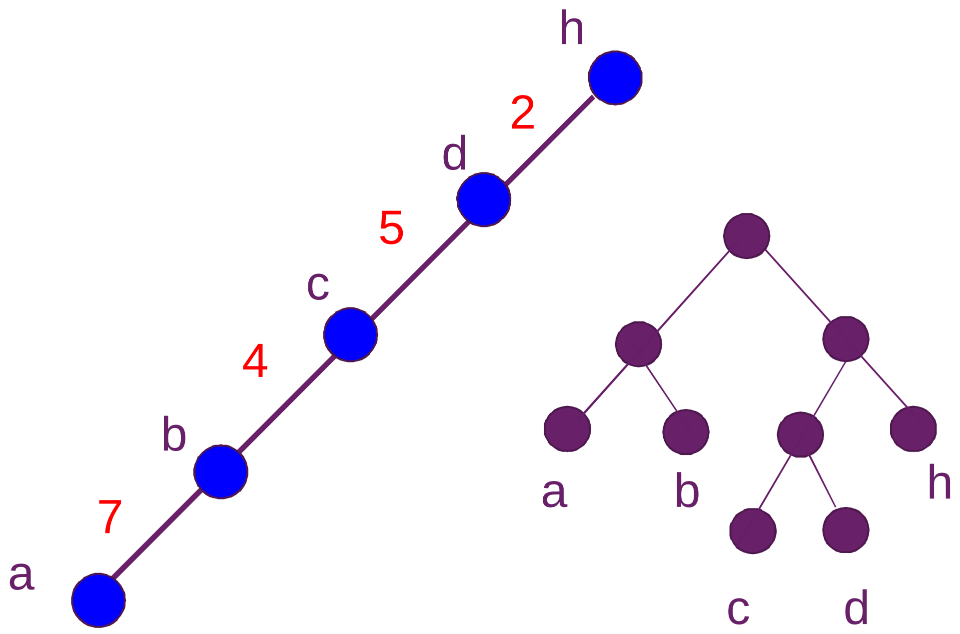
*(v,father(v)) is a heavy edge in the dynamic tree if 2*size(v)>size(father(v))

Lemma. With solid paths defined by the heavy edges, a path from v to droot(v) contains at most log n dashed edges.

Lemma. With globally biased binary trees, the running time of one splice operation is not constant, but summing over all splices, the total running time of expose is in O(log n).

Dynamic trees: Focus on some operations

Say v=a, and assume expose(a) has been done.



dparent(a)
droot(a)
} Search the binary tree

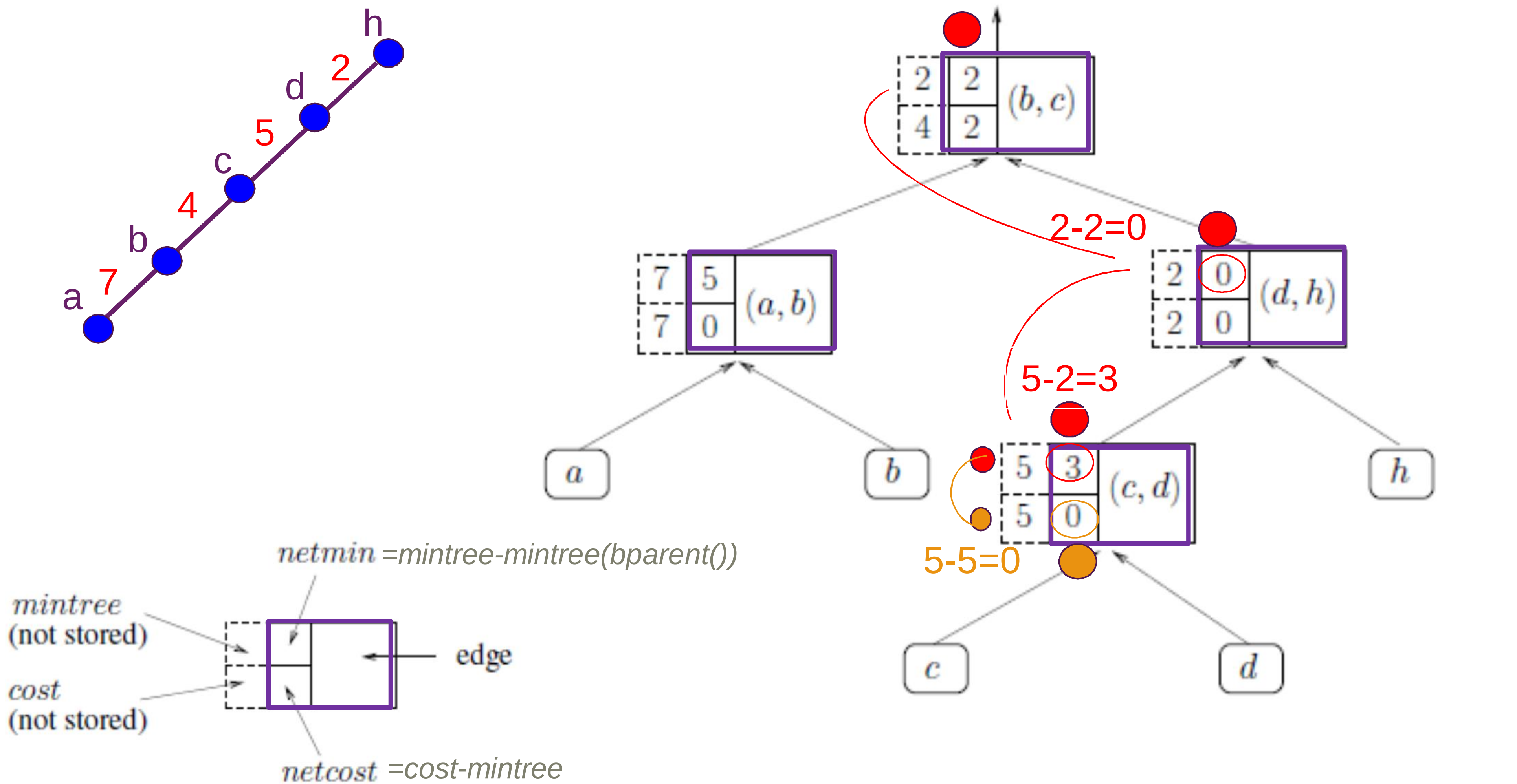
dcost(a)
dmincost(a)
dupdate(a,u)
} Store relative information at nodes and act on the root

dlink(v,c,u)
dcut(c)
devert(c)
} Cut/link/reverse/balance the binary trees

Ÿ whatever the type of the tree, it will have a lot of links and information at nodes

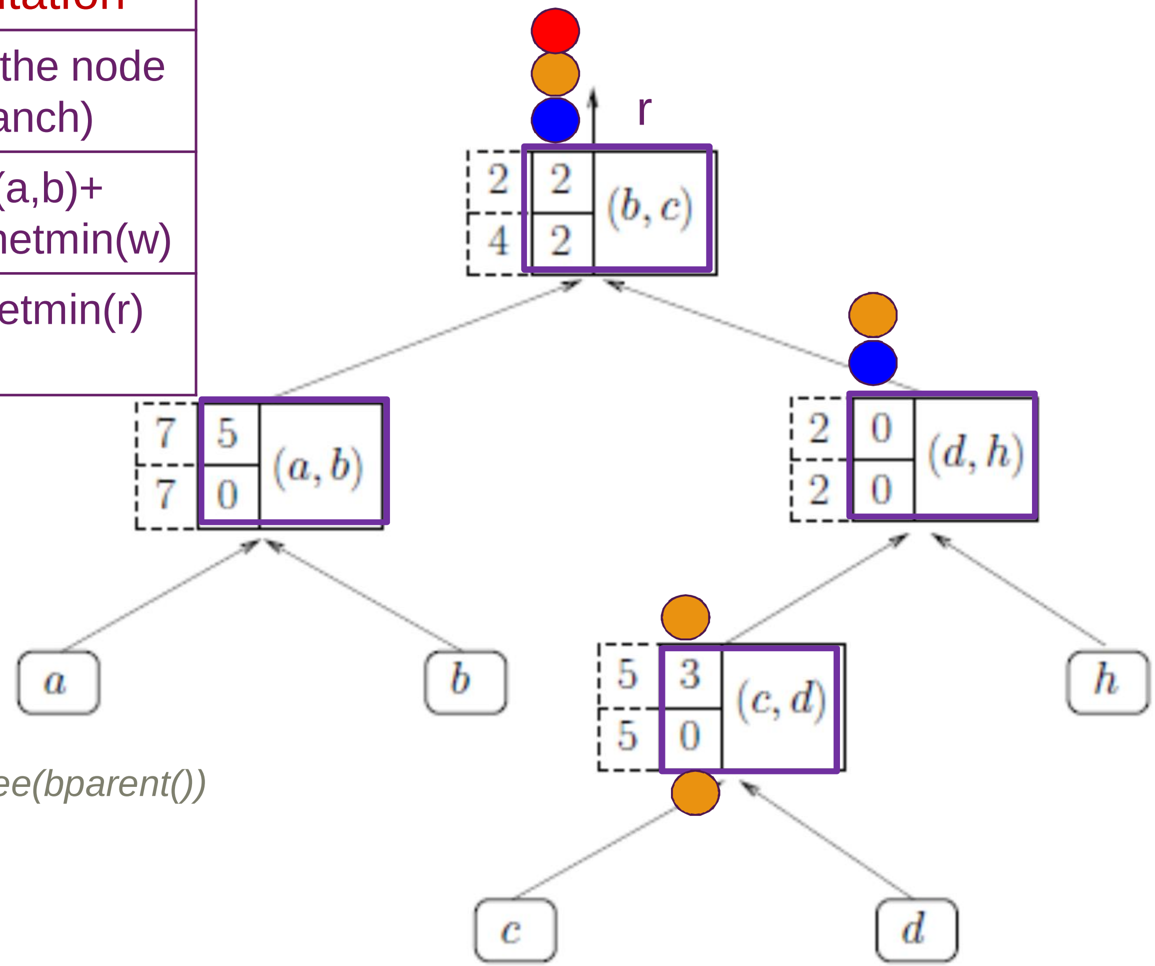
Dynamic trees: Focus on dcost(), dmincost(), dupdate()

Cost-related information in the binary tree.

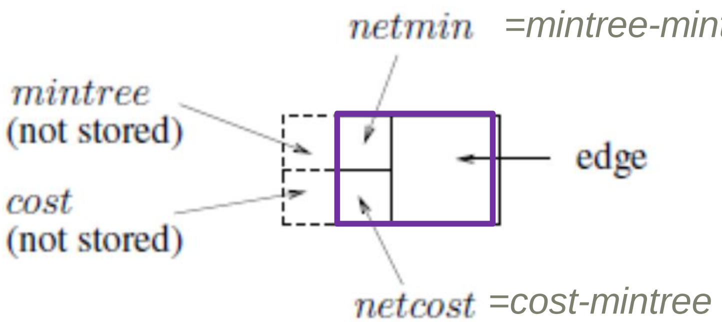


Dynamic trees: Focus on dcost(), dmincost(), dupdate()

Operation	Value	Computation
dmincost(a)	mintree(r)	Search the node (one branch)
dcost(a)	netcost(a,b)+ mintree(a,b)	netcost(a,b)+ $\times_{w=e,..,r}$ netmin(w)
dupdate(a,x)	+u on all costs	+u on netmin(r)



Notes.
1) *u on netmin(r) does not multiply all the values
2) several costs are possible



Dynamic trees: Conclusions (1)

With heavy/light edges and globally biased binary trees:

- Good news: All the operations are in O(log n)
- Principle: Create the solid path from v to r, then work on the associated binary tree
- Local operations dparent(v), droot(v) search the binary tree
- Aggregate operations dcost(v), dmincost(v), update(v,u) store relative information at nodes and sometimes search the binary tree
- Operations dlink(v,w,u), dcut(v), devert(v) modifying the structure of the forest link, cut, reverse, re-balance the binary trees involved.
- Several costs are possible

Dynamic trees: Conclusions (2)

Cannot be used in all situations:

- Not adapted to top-down visiting the dynamic trees
- Not adapted to searching a value in the forest/a dynamic tree
- Aggregate operations do not allow multiplications

Outline

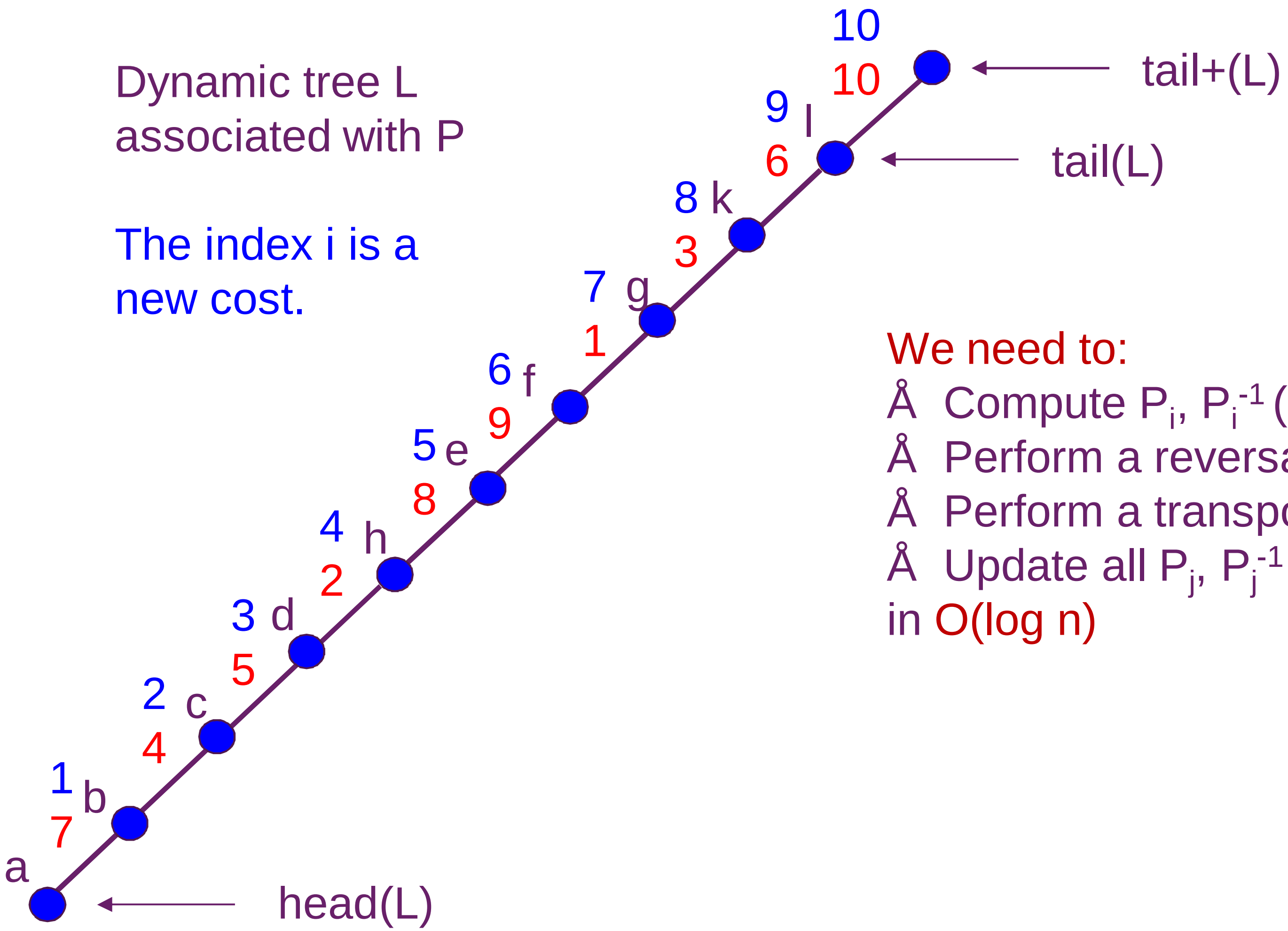
- Motivations for an Array-List (but not the Java Collection)
- Fundamentals: Dynamic Trees
- A Sequence is a Filiform Tree
- A step further: A Log-List too … but Needs Adjustments
- Conclusion

A sequence is a filiform tree: Implement a sequence as a dynamic tree

P: 7 4 5 2 8 9 1 3 6 10

Dynamic tree L associated with P

The index i is a new cost.



We need to:
- Compute P_i, P_i^{-1} (fixed i)
- Perform a reversal
- Perform a transposition
- Update all P_j, P_j^{-1}

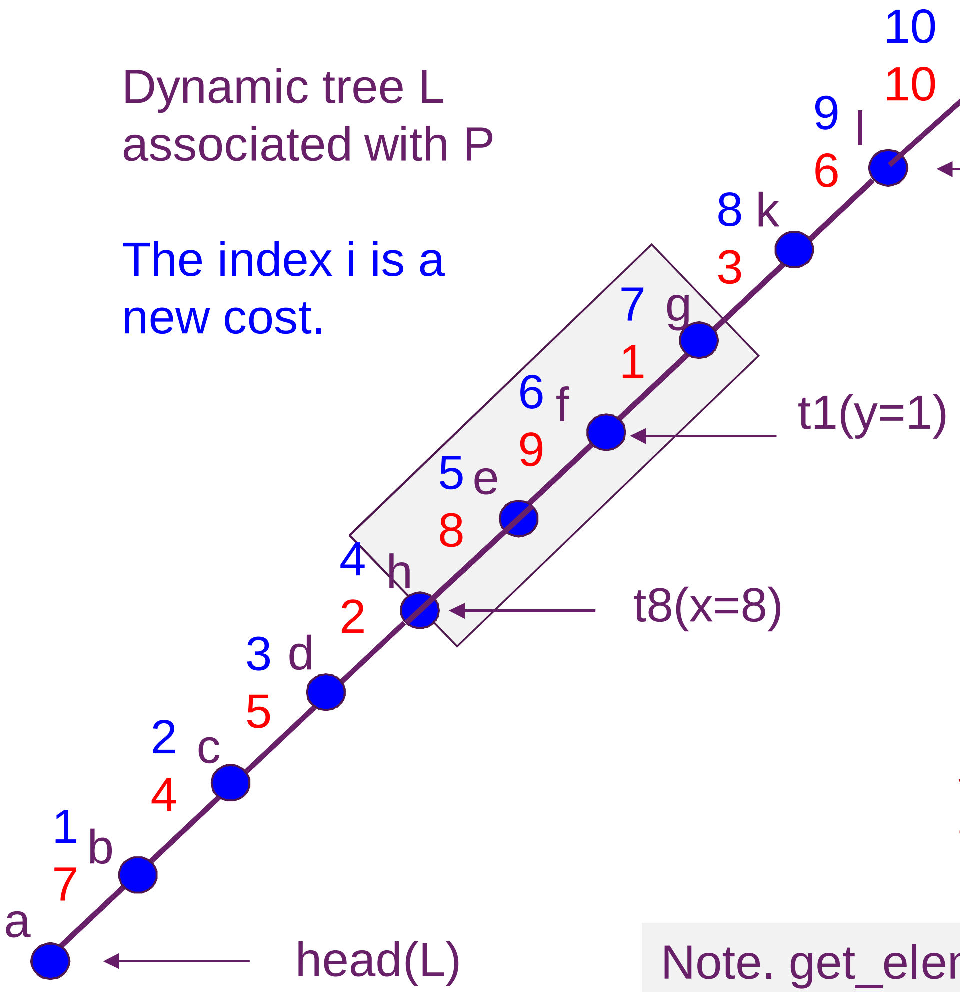
in O(log n)

A sequence is a filiform tree: what we get from dynamic trees

P: 7 4 5 2 8 9 1 3 6 10

Dynamic tree L associated with P

The index i is a new cost.



We are able to do more (tx,ty designate nodes with edge costs x, y) :

devert, dcut, dlink
- insert(L,L_1,tx)
- delete(L,tx,ty)
- reverse(L,tx,ty)

dmincost, dupdate
- find_min(L,tx,ty) (or max)
- add(L,tx,ty,u)

Not done yet with dynamic trees
- change_sign(L,tx,ty)
- find_rank(L,tx) (=P^{-1}[x])
- find_element(L,i) (=P[i])

in O(log n)

Note. get_element(), succ(), prev() in O(log n)

A sequence is a filiform tree: what remains to be done

1) Perform

Not done yet with dynamic trees

- change_sign(L,tx,ty)
- find_rank(L,tx) (=P^{-1}[x])
- find_element(L,i) (=P[i])

2) Update the index for delete, insert, reverse

Outline

- Motivations for an Array-List (but not the Java Collection)
- Fundamentals: Dynamic Trees
- Easy observation: A Sequence is a Filiform Tree
- A Log-List is a filiform tree too … but Needs Adjustments
- Conclusion

A log-list is a filiform tree too: Adjustments (1)

1) Perform

Not done yet with dynamic trees

- change_sign(L,tx,ty)
- find_rank(L,tx) (=P^{-1}[x])
- find_element(L,i) (=P[i])

Quite easy : find_element(L,i)

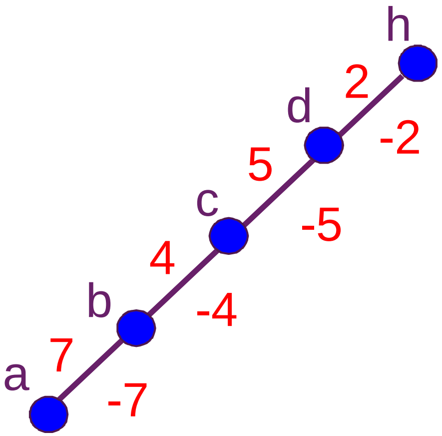
devert(tail+(L))	put the list in its standard from
expose(head(L))	all the list is a solid path with binary tree B
New: dsearchindex(B,i)	search i in the binary search tree B with values index()

Time: O(log n)

A log-list is a filiform tree too

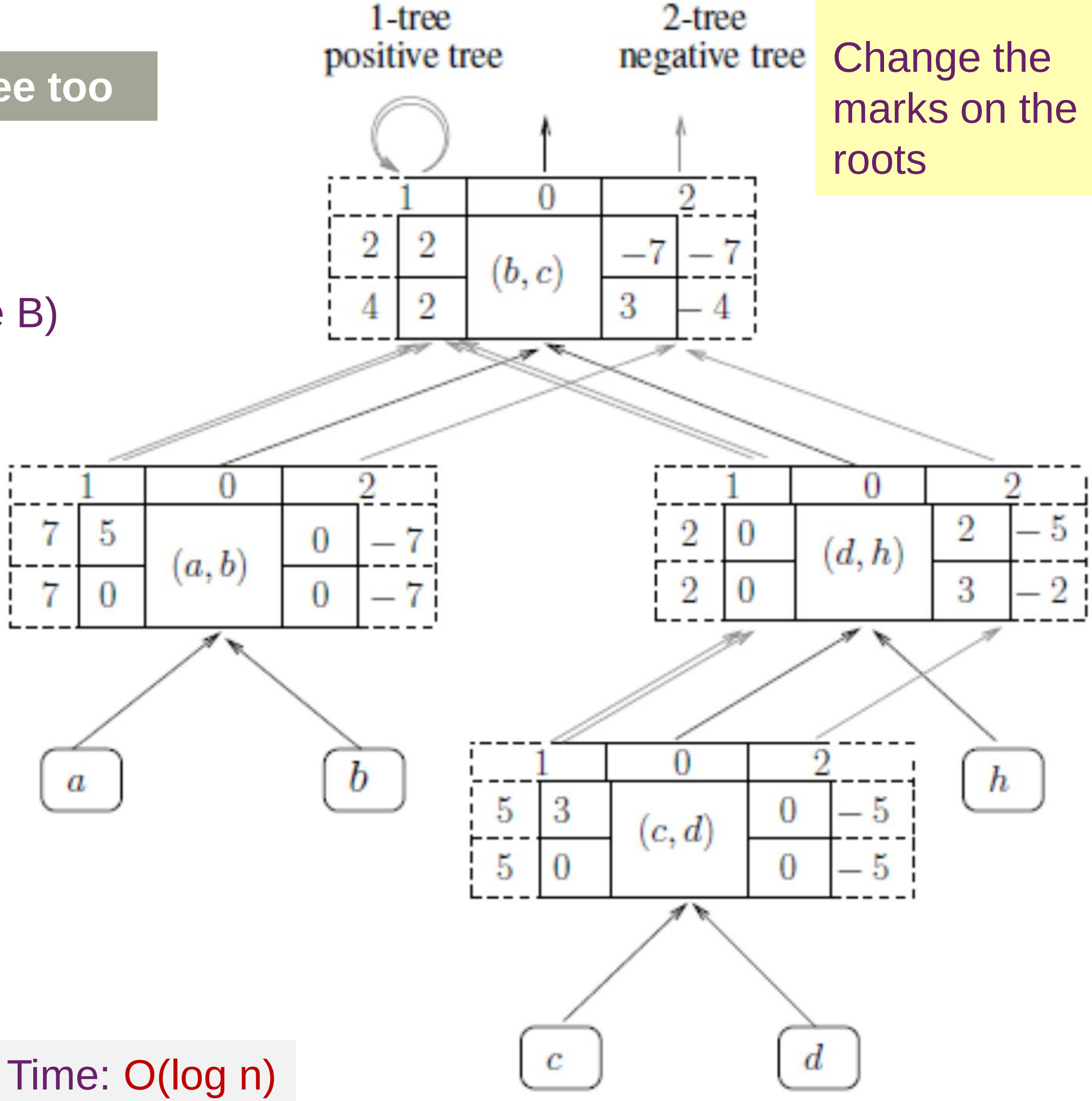
change_sign(L,tx,ty):
devert(dparent(ty)),
expose(tx) (Ÿ binary tree B)
dminuscost(B) New

Idea:
(±)binary tree



Time: O(log n)

cost() -cost()



Multiply by -1:

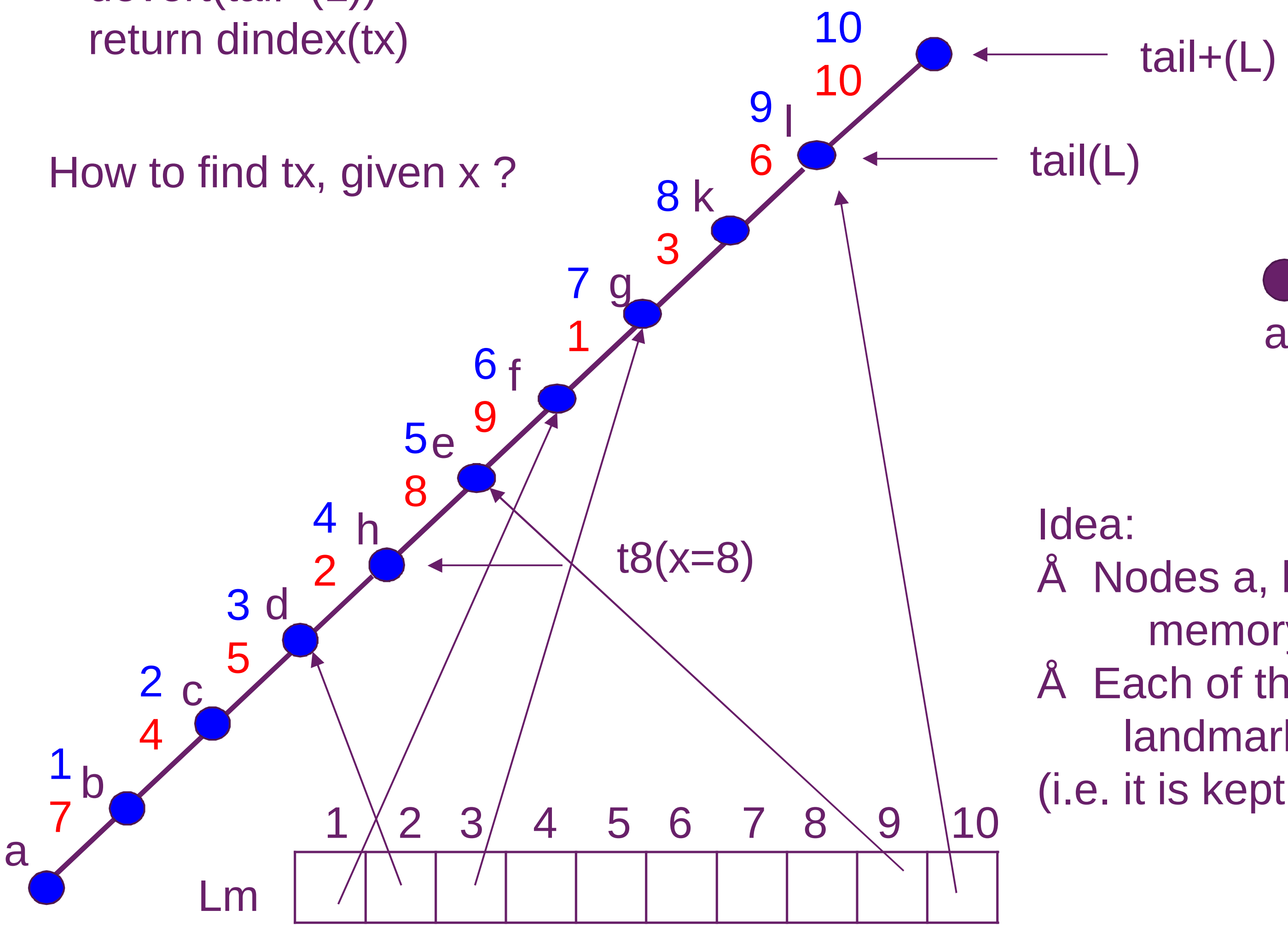
Change the marks on the roots

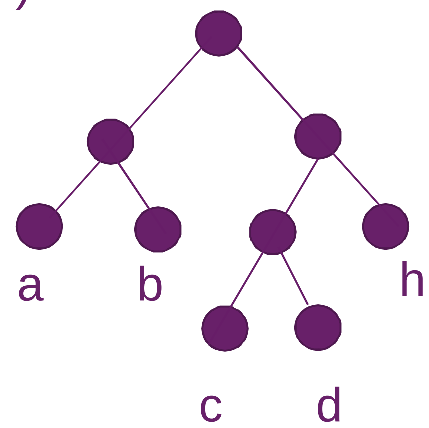
A log-list is a filiform tree too: Adjustments(1ter)

find_rank(L,tx)
devert(tail+(L))
return dindex(tx)

P: 7 4 5 2 8 9 1 3 6 10

How to find tx, given x ?





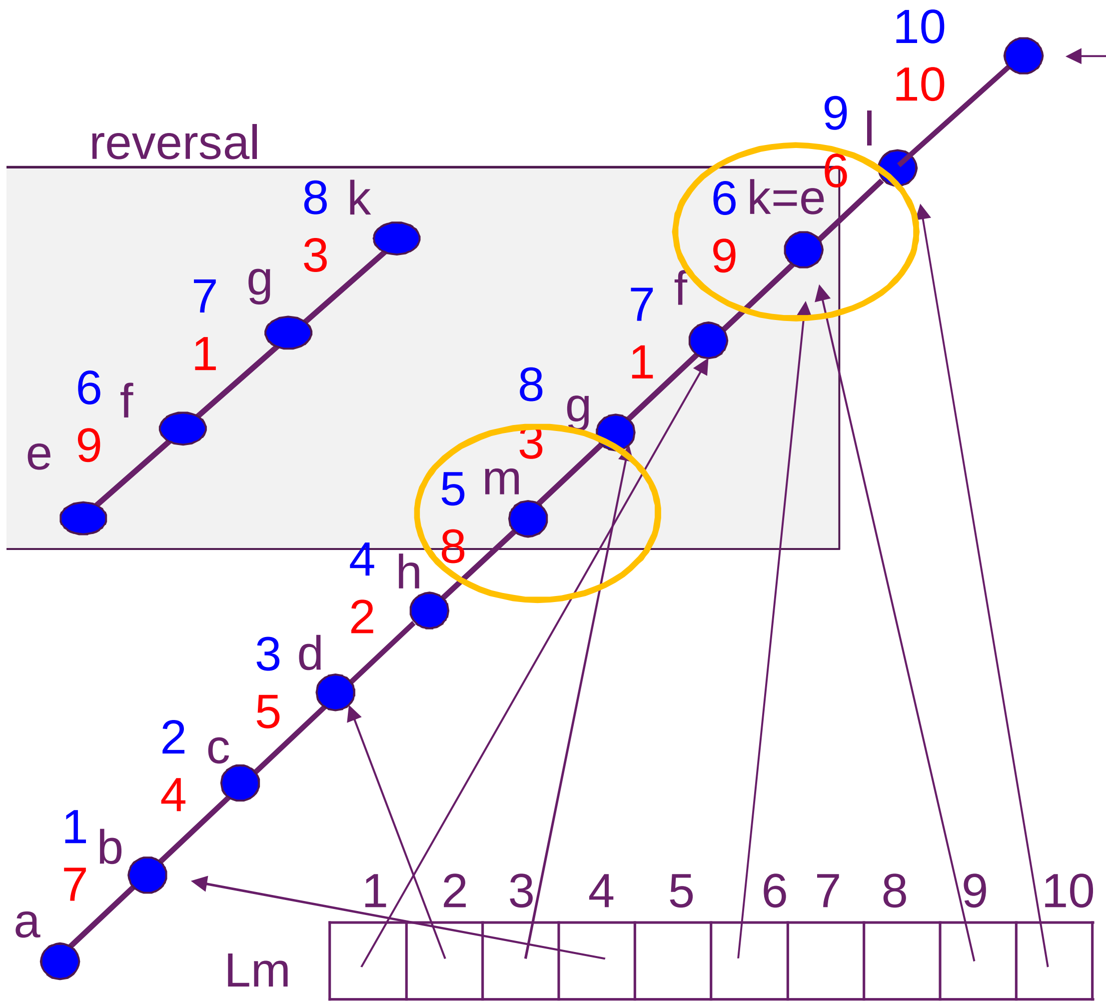
Idea:

Å Nodes a, b, c, é are fixed memory cases

Å Each of them may be a landmark for a given element (i.e. it is kept close to the element)

A log-list is a filiform tree too: Adjustments(1ter)

P: 7 4 5 2 8 9 1 3 6 10



tail+(L)

Idea:

Å Nodes a, b, c, é are fixed memory cases

Å Each of them may be a landmark for a given element (i.e. it is kept close to the element)

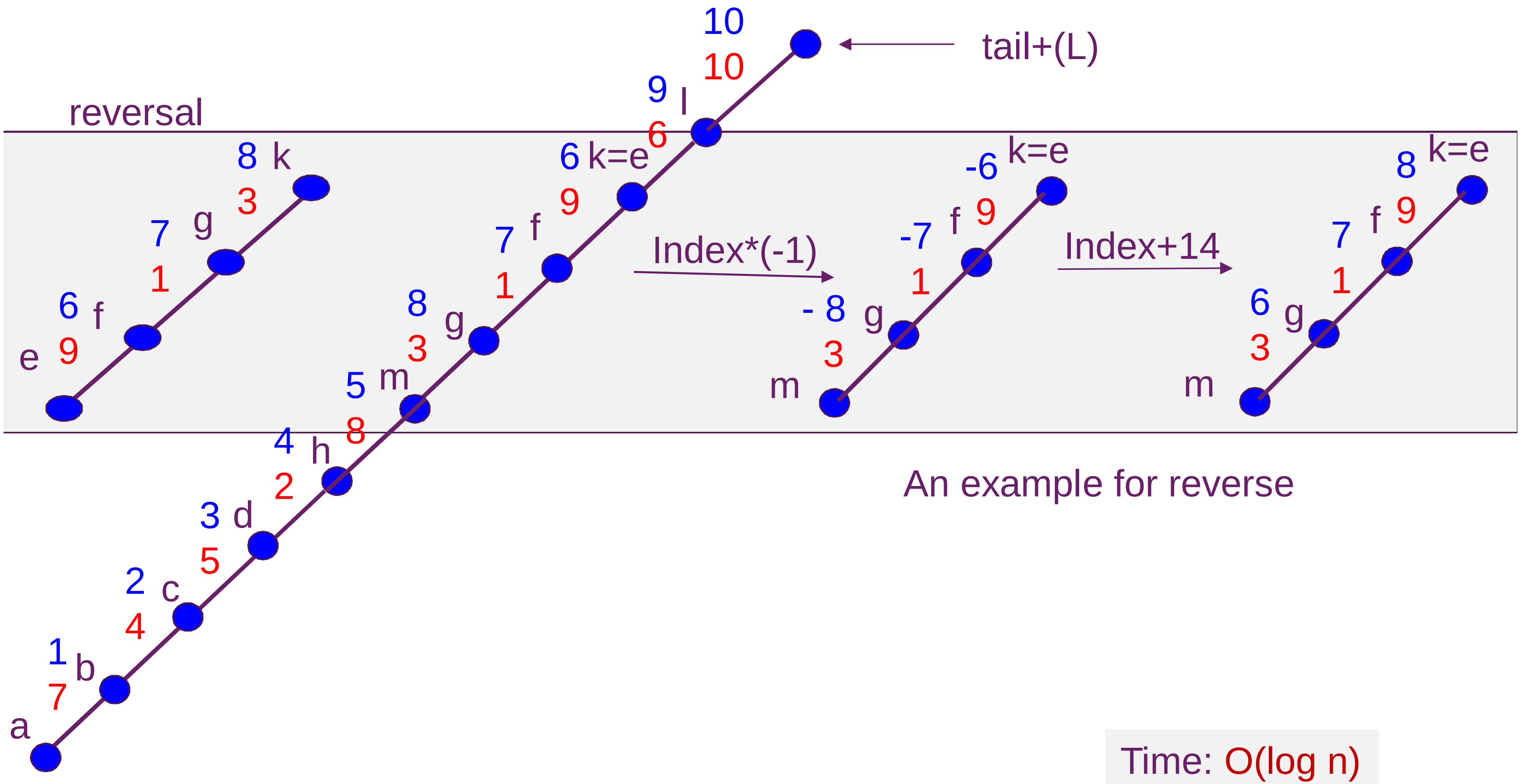
Å é with a few exceptions at the endpoints of the moved blocks

Lm[x] is one of the endpoints of the edge with cost x; tx is the lowest endpoint of it

Time: O(log n)

A log-list is a filiform tree too: Adjustments(2)

2) Update the index for delete, insert, reverse



An example for reverse

Time: O(log n)

A log-list is a filiform tree too: Summary

- What is a log-list ?
  - a filiform dynamic tree L (but temporarily a forest of such trees)
  - three pointers head(L), tail(L), tail+(L)
  - a toolbox of O(log n) time operations

- get_element(tx)
- succ(tx)
- prec(tx)
- insert(L,L_1,tx)
- delete(L,tx,ty)
- reverse(L,tx,ty)
- find_min(L,tx,ty) (or max)
- add(L,tx,ty,u)
- change_sign(L,tx,ty)
- find_rank(L,tx) (=P^{-1}[x])
- find_element(L,i) (=P[i])

  - (if needed) a landmark table Lm (allows to compute tx)

A log list is a filiform tree too: Summary

- Shall we deduce that visiting all the elements in a log-list takes O(n log n)?

  No, visiting directly the binary tree allows to do this in O(n) … but needs to go deeper into the implementation

- What about searching a given element in a log-list?

  If the list is not ordered, O(n) as above

  If the list is ordered and has distinct elements, the dynamic tree operation dsearchcost (we used it only for cost=index) does it in O(log n).

Outline

- Motivations for an Array-List (but not the Java Collection)
- Fundamentals: Dynamic Trees
- Easy observation: A Sequence is a Filiform Tree
- Conclusion

Conclusion

- Improving the algorithm for sorting by prefix reversals and prefix transpositions
  - n steps, each identifying and performing a block operation
  - $O(n \log n)$ time instead of $O(n^2)$ before
- In addition: four other variants of the sorting problem are improved
- Some of them resist however
- Log-list could have many other applications

Bibliography

Å Daniel D. Sleator and Robert E. Tarjan ï *A Data Structure for Dynamic Trees*, Journal of Computer and System Sciences 26, 362-291 (1983)

Å Irena Rusu ï *Log-Lists and Their Applications to Sorting by Transpositions, Reversals and Block-Interchanges*, arXiv 1507.01512 (2015).

Thank you !