LSEQ: an Adaptive Structure for Sequences in Distributed Collaborative Editing

ConcoRDanT

Brice Nédelec
Pascal Molli Achour Mostefaoui Emmanuel Desmontils

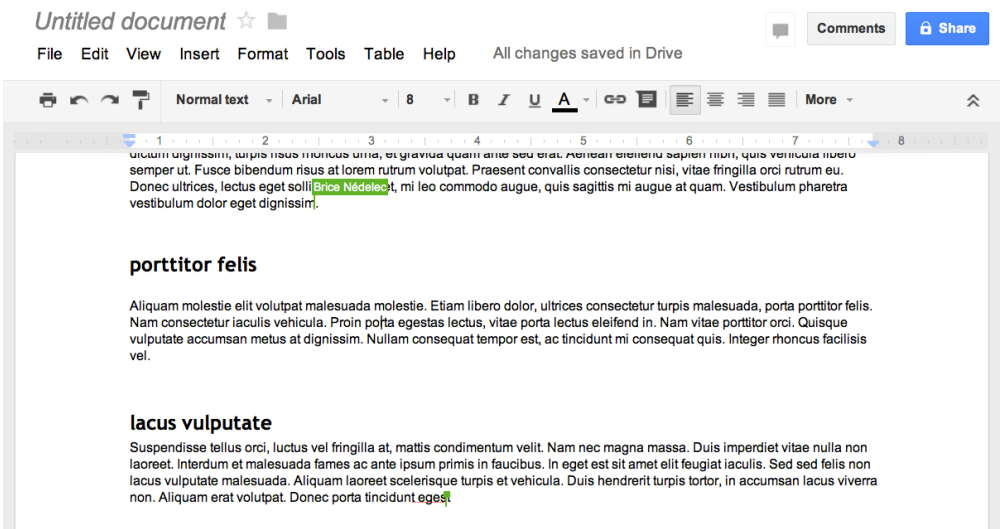
LINA, 2 rue de la Houssinière
BP02208, 44322 Nantes Cedex 03
first.last@univ-nantes.fr

UNIVERSITÉ DE NANTES

lina LABORATOIRE D'INFORMATIQUE DE NANTES ATLANTIQUE

Distributed Collaborative Editors

- DCE allow to **distribute the work** across **space, time, organizations.**
- *example : Google Docs, Google Wave, Etherpad, Git, Subversion, Wikipedia, CoVim. . .*



Current editors are not entirely satisfying

- **Single point of failure** : the server is down, I cannot work on my document anymore...
- **Privacy concern** : the owner of the server owns the data... He can sell it to third-party company. (*e.g. advertisement company*)

$\Rightarrow$ **As a user, I don't want my editor got any of these !**

- $\Rightarrow$ Deny the enslavement to a coordination provider !
- $\Rightarrow$ I still want something as appealing as Google Docs though...

Benefits of the optimistic replication

- most editors relies on the optimistic replication approach.
  - high **availability** of data : each user has its own local replica
  - **two phases** operations :
    1. **locally** prepare the result of the operation and send it
    2. integrate the result of the **remote** operations to the data
- when the data is a document $\rightarrow$ modifications are **insert** or **delete** an element

  $\Rightarrow$ element : character, line, paragraph...
- modifications of the document must ensure the 3 properties (CCI) :
  1. **Convergence** : all replicas eventually reach an identical state
  2. **Causality** : any operation needs to reflect the operations that occurred causally before it
  3. **Intention** : the effect of an operation needs to meet the intention of the user that ordered it

$\Rightarrow$ a **very efficient** approach that belongs to optimistic replication

$\Rightarrow$ **Conflict-free Replicated Data Types**

Benefits of the optimistic replication

- most editors relies on the optimistic replication approach.
  - high **availability** of data : each user has its own local replica
  - **two phases** operations :
    1. **locally** prepare the result of the operation and send it
    2. integrate the result of the **remote** operations to the data
- when the data is a document $\rightarrow$ modifications are **insert** or **delete** an element

  $\Rightarrow$ element : character, line, paragraph...

- modifications of the document must ensure the 3 properties (CCI) :
  1. **Convergence** : all replicas eventually reach an identical state
  2. **Causality** : any operation needs to reflect the operations that occurred causally before it
  3. **Intention** : the effect of an operation needs to meet the intention of the user that ordered it

$\Rightarrow$ a **very efficient** approach that belongs to optimistic replication

$\Rightarrow$ **Conflict-free Replicated Data Types**

Benefits of the optimistic replication

- most editors relies on the optimistic replication approach.
  - high **availability** of data : each user has its own local replica
  - **two phases** operations :
    1. **locally** prepare the result of the operation and send it
    2. integrate the result of the **remote** operations to the data
- when the data is a document $\rightarrow$ modifications are **insert** or **delete** an element

  $\Rightarrow$ element : character, line, paragraph...

- modifications of the document must ensure the 3 properties (CCI) :
  1. **Convergence** : all replicas eventually reach an identical state
  2. **Causality** : any operation needs to reflect the operations that occurred causally before it
  3. **Intention** : the effect of an operation needs to meet the intention of the user that ordered it

$\Rightarrow$ a **very efficient** approach that belongs to optimistic replication

$\Rightarrow$ **Conflict-free Replicated Data Types**

Benefits of the optimistic replication

- most editors relies on the optimistic replication approach.
  - high **availability** of data : each user has its own local replica
  - **two phases** operations :
    1. **locally** prepare the result of the operation and send it
    2. integrate the result of the **remote** operations to the data
- when the data is a document $\rightarrow$ modifications are **insert** or **delete** an element

  $\Rightarrow$ element : character, line, paragraph...

- modifications of the document must ensure the 3 properties (CCI) :
  1. **Convergence** : all replicas eventually reach an identical state
  2. **Causality** : any operation needs to reflect the operations that occurred causally before it
  3. **Intention** : the effect of an operation needs to meet the intention of the user that ordered it

$\Rightarrow$ a **very efficient** approach that belongs to optimistic replication

$\Rightarrow$ **Conflict-free Replicated Data Types**

CRDTs for sequences

Conflict-free Replicated Data Types

- simple abstract type for sequences that can model documents
- optimistic replication : convergent, two phases operations . . .
- avoids the difficult task of sovling conflicts
- 2 commutative operations : insert and delete
  - $\Rightarrow$ ins/ins, ins/del, del/del
  - $\Rightarrow$ except : ins(a)/del(a) $\Rightarrow$ require causality : ins(a) $\rightarrow$ del(a)

- 1 identifier for each element in sequence
  - document $\rightarrow$ character, line, paragraph. . .
  - unique, non mutable identifier
  - totally ordered
  - the order on ids makes the sequence

$\Rightarrow$ delete($id_{element}$)

$\Rightarrow$ insert(id_p, element, id_q) $\Rightarrow$ **alloc** (id_p, id_q) $\Rightarrow$ $id_p < id_{element} < id_q$

CRDTs for sequences

Conflict-free Replicated Data Types

- simple abstract type for sequences that can model documents
- optimistic replication : convergent, two phases operations . . .
- avoids the difficult task of sovling conflicts
- 2 commutative operations : insert and delete
  - $\Rightarrow$ ins/ins, ins/del, del/del
  - $\Rightarrow$ except : ins(a)/del(a) $\Rightarrow$ require causality : ins(a) $\rightarrow$ del(a)

- 1 identifier for each element in sequence
  - document $\rightarrow$ character, line, paragraph. . .
  - unique, non mutable identifier
  - totally ordered
  - the order on ids makes the sequence

$\Rightarrow$ delete($id_{element}$)

$\Rightarrow$ insert(id_p, element, id_q) $\Rightarrow$ **alloc** (id_p, id_q) $\Rightarrow$ $id_p < id_{element} < id_q$

Allocation of identifiers matters

D O C E N G

- linearly growing ids
  - document with 1000 lines $\Rightarrow max(id.size) = 1000$

Allocation of identifiers matters

- linearly growing ids
  - document with 1000 lines $\Rightarrow max(id.size) = 1000$

Allocation of identifiers matters

0 **1** 9

D O C E N G

- linearly growing ids
  - document with 1000 lines $\Rightarrow max(id.size) = 1000$

Allocation of identifiers matters

0	1	**2**				9
	D	O	C	E	N	G

- linearly growing ids
  - document with 1000 lines $\Rightarrow$ $max(id.size) = 1000$

Allocation of identifiers matters

0	1	2	**3**			9
	D	O	C	E	N	G

- linearly growing ids
  - document with 1000 lines $\Rightarrow max(id.size) = 1000$

Allocation of identifiers matters

0	1	2	3	**4**	**5**	**6**	9
	D	O	C	E	N	G	

- linearly growing ids
  - document with 1000 lines $\Rightarrow max(id.size) = 1000$

Allocation of identifiers matters

0	1	2	3	4	5	6	9
	D	O	C	E	N	G	
0							**9**
	E	**N**	**G**				

- linearly growing ids
  - document with 1000 lines $\Rightarrow max(id.size) = 1000$

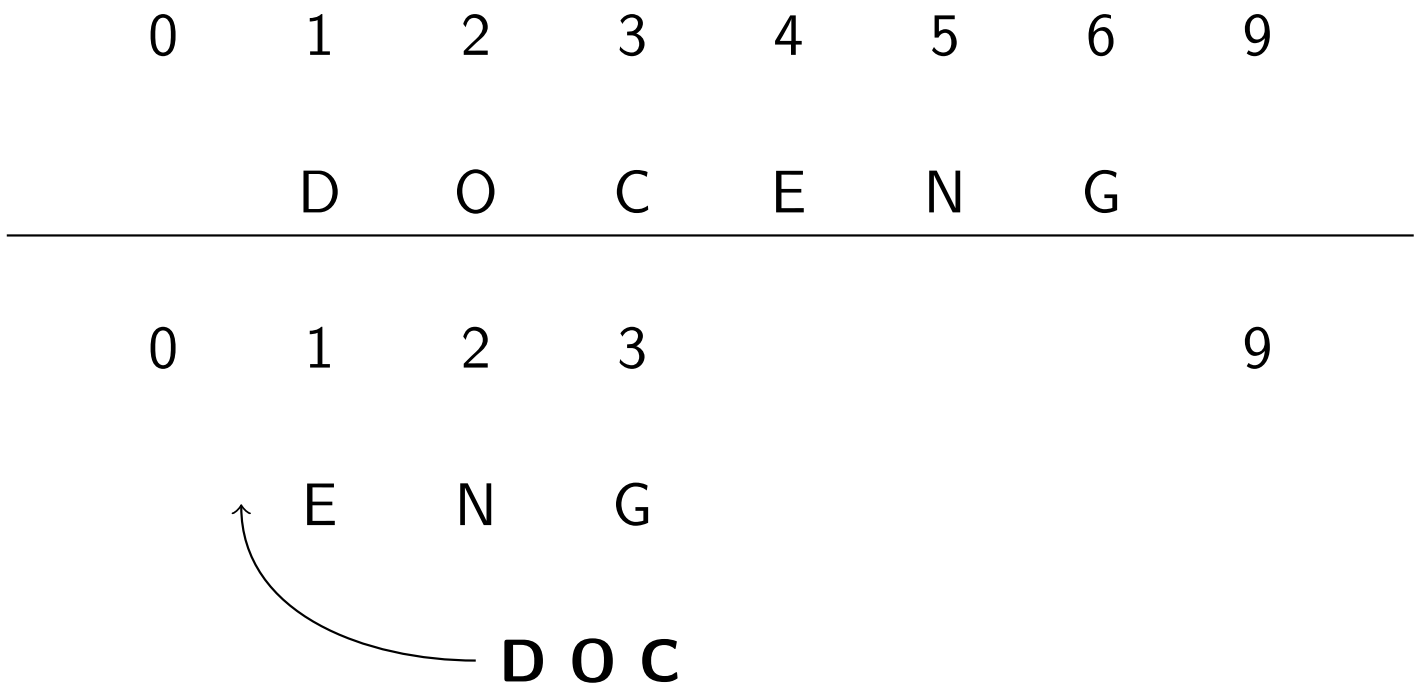
Allocation of identifiers matters

0	1	2	3	4	5	6	9
	D	O	C	E	N	G	

0	**1**	**2**	**3**				9
	E	N	G				

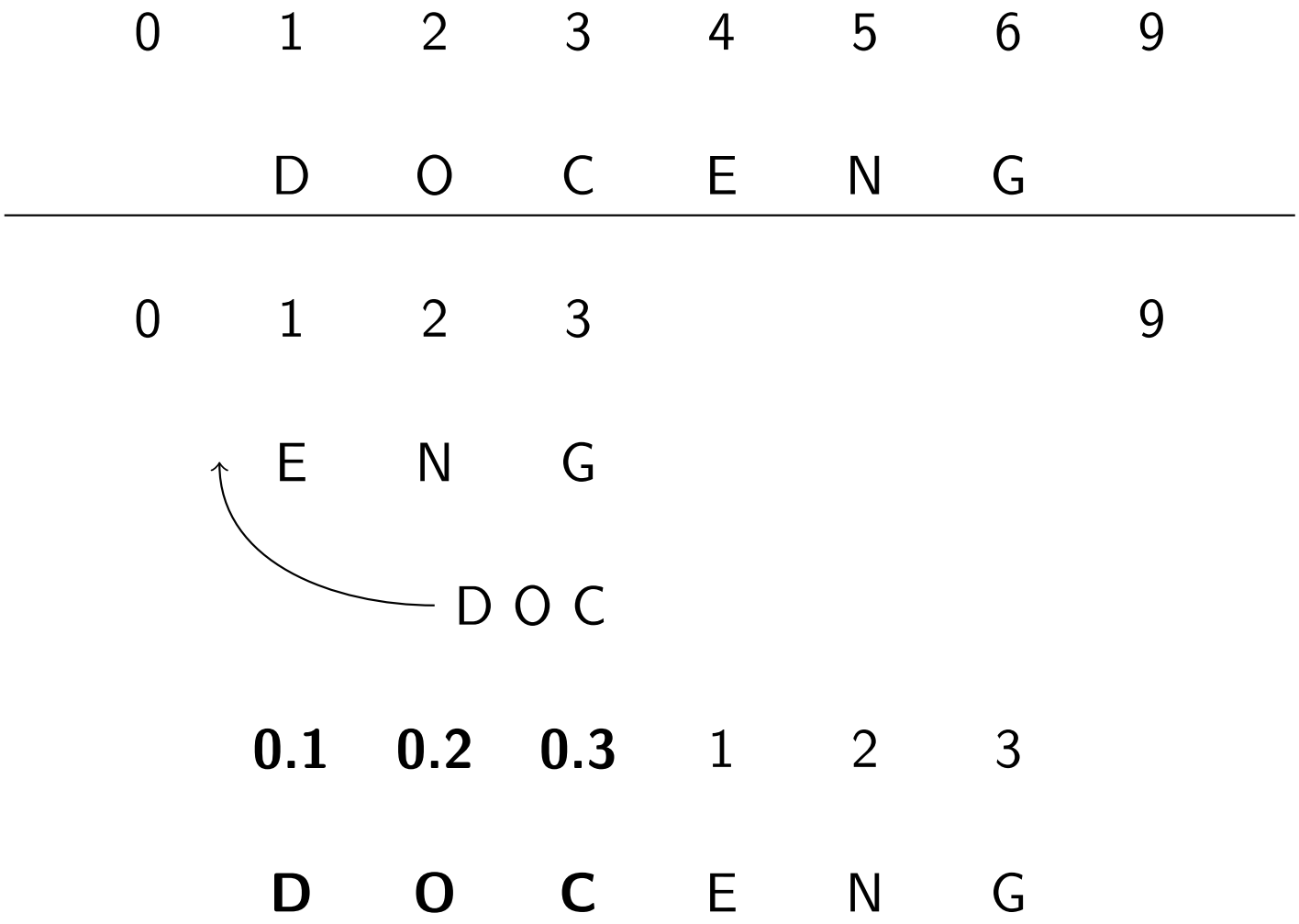
- linearly growing ids
  - document with 1000 lines $\Rightarrow max(id.size) = 1000$

Allocation of identifiers matters



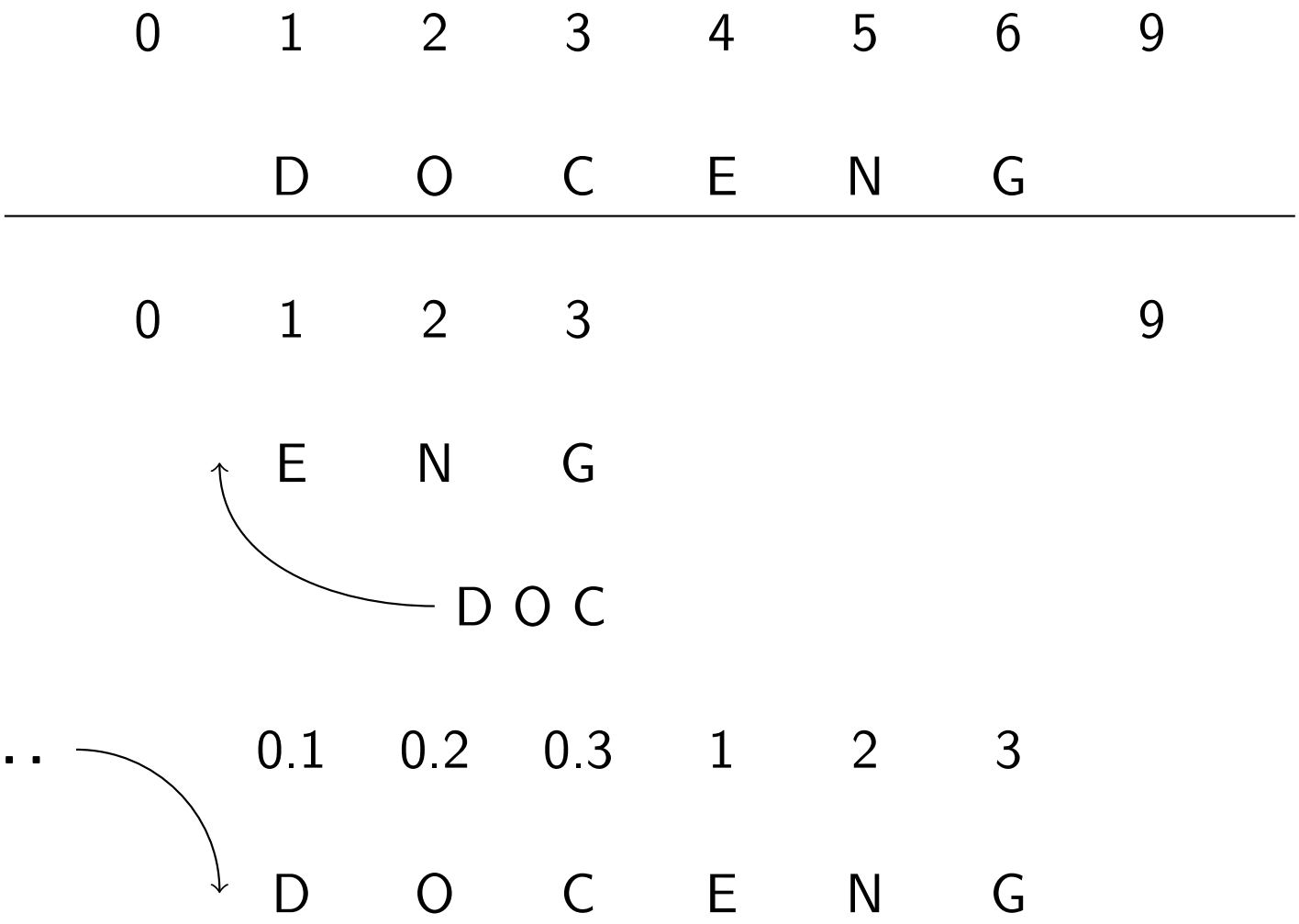
- linearly growing ids
  - document with 1000 lines $\Rightarrow max(id.size) = 1000$

Allocation of identifiers matters



- linearly growing ids
  - document with 1000 lines $\Rightarrow max(id.size) = 1000$

Allocation of identifiers matters

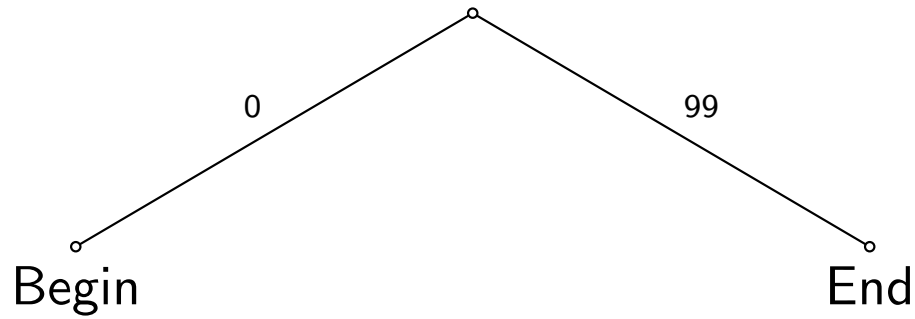


- linearly growing ids
  - document with 1000 lines $\Rightarrow$ $max(id.size) = 1000$

Problem : lowering the size of identifiers

Variable-size identifier

A variable-size identifier id is a sequence of numbers $id = [p_1.p_2 \dots p_n]$ which can designate a path in a tree.



- $Id_D = [10]$
- $Id_O = [10.13]$

Problem statement

Let $\mathcal{D}$ a document on which n insert operations have been performed. Let $\mathcal{I}(\mathcal{D}) = \{id | (\_, id) \in \mathcal{D}\}$. The function $alloc(id_p, id_q)$ should provide identifiers such as :

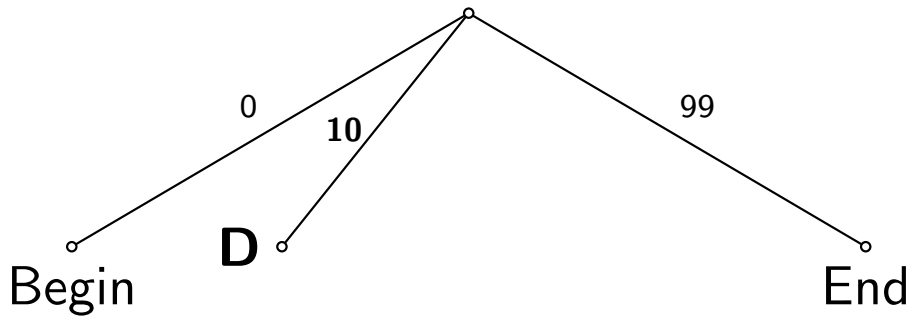
$$\sum_{id \in \mathcal{I}} \frac{|id|_2}{n} < O(n)$$

$\Rightarrow$ **No cleaning of the structure required !**

Problem : lowering the size of identifiers

Variable-size identifier

A variable-size identifier id is a sequence of numbers $id = [p_1.p_2 \ldots p_n]$ which can designate a path in a tree.



- $Id_D = [10]$
- $Id_O = [10.13]$

Problem statement

Let $\mathcal{D}$ a document on which n insert operations have been performed. Let $\mathcal{I}(\mathcal{D}) = \{id | (\_, id) \in \mathcal{D}\}$. The function $alloc(id_p, id_q)$ should provide identifiers such as :

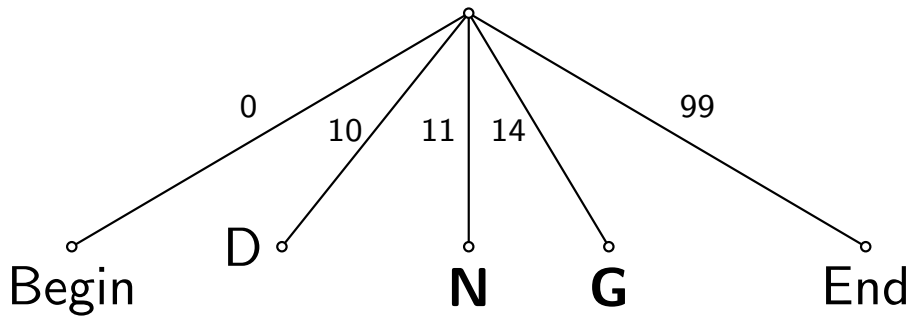
$$\sum_{id \in \mathcal{I}} \frac{|id|_2}{n} < O(n)$$

$\Rightarrow$ **No cleaning of the structure required !**

Problem : lowering the size of identifiers

Variable-size identifier

A variable-size identifier id is a sequence of numbers $id = [p_1.p_2 \dots p_n]$ which can designate a path in a tree.



- $Id_D = [10]$
- $Id_O = [10.13]$

Problem statement

Let $\mathcal{D}$ a document on which n insert operations have been performed. Let $\mathcal{I}(\mathcal{D}) = \{id | (\_, id) \in \mathcal{D}\}$. The function $alloc(id_p, id_q)$ should provide identifiers such as :

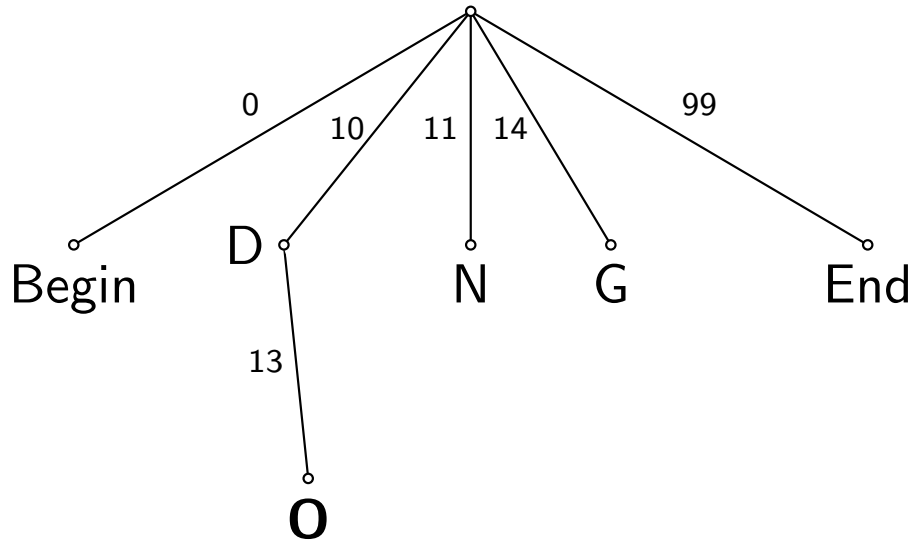
$$\sum_{id \in \mathcal{I}} \frac{|id|_2}{n} < O(n)$$

$\Rightarrow$ **No cleaning of the structure required !**

Problem : lowering the size of identifiers

Variable-size identifier

A variable-size identifier id is a sequence of numbers $id = [p_1.p_2 \dots p_n]$ which can designate a path in a tree.



- $Id_D = [10]$
- $Id_O = [10.13]$

Problem statement

Let $\mathcal{D}$ a document on which n insert operations have been performed. Let $\mathcal{I}(\mathcal{D}) = \{id | (\_, id) \in \mathcal{D}\}$. The function $alloc(id_p, id_q)$ should provide identifiers such as :

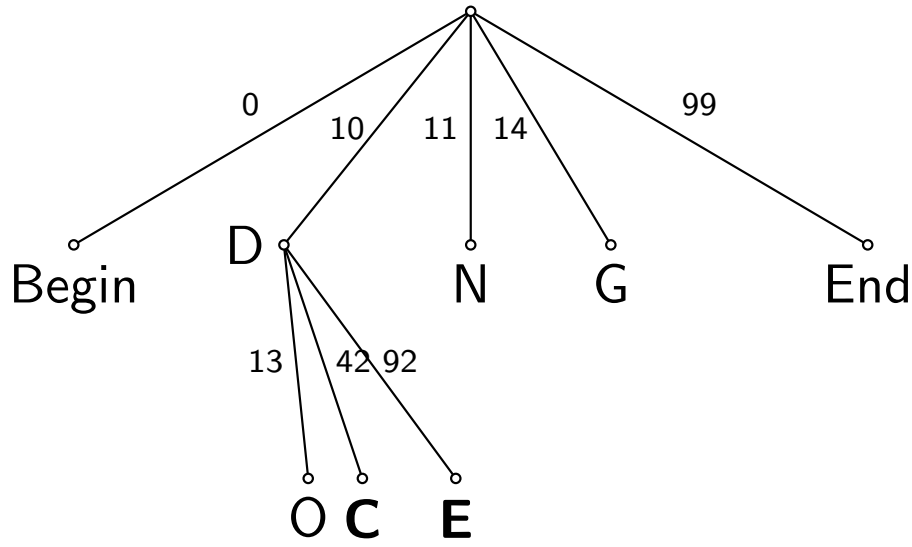
$$\sum_{id \in \mathcal{I}} \frac{|id|_2}{n} < O(n)$$

$\Rightarrow$ **No cleaning of the structure required !**

Problem : lowering the size of identifiers

Variable-size identifier

A variable-size identifier id is a sequence of numbers $id = [p_1.p_2 \dots p_n]$ which can designate a path in a tree.



- $Id_D = [10]$
- $Id_O = [10.13]$

Problem statement

Let $\mathcal{D}$ a document on which n insert operations have been performed. Let $\mathcal{I}(\mathcal{D}) = \{id | (\_, id) \in \mathcal{D}\}$. The function $alloc(id_p, id_q)$ should provide identifiers such as :

$$\sum_{id \in \mathcal{I}} \frac{|id|_2}{n} < O(n)$$

$\Rightarrow$ **No cleaning of the structure required !**

Proposal : **LSEQ**

LSEQ relies on **3** components :

- **exponential tree** : we assume that when ↗ depth, it means that a lot of insertions were done and more are probably upcoming. We need more available nodes
- **multiple allocation strategies** : a strategy designed for end-editing is not sufficient to handle any editing behaviour
- **random strategy choice** : I have multiple sub allocation strategies. Which one should I choose ?

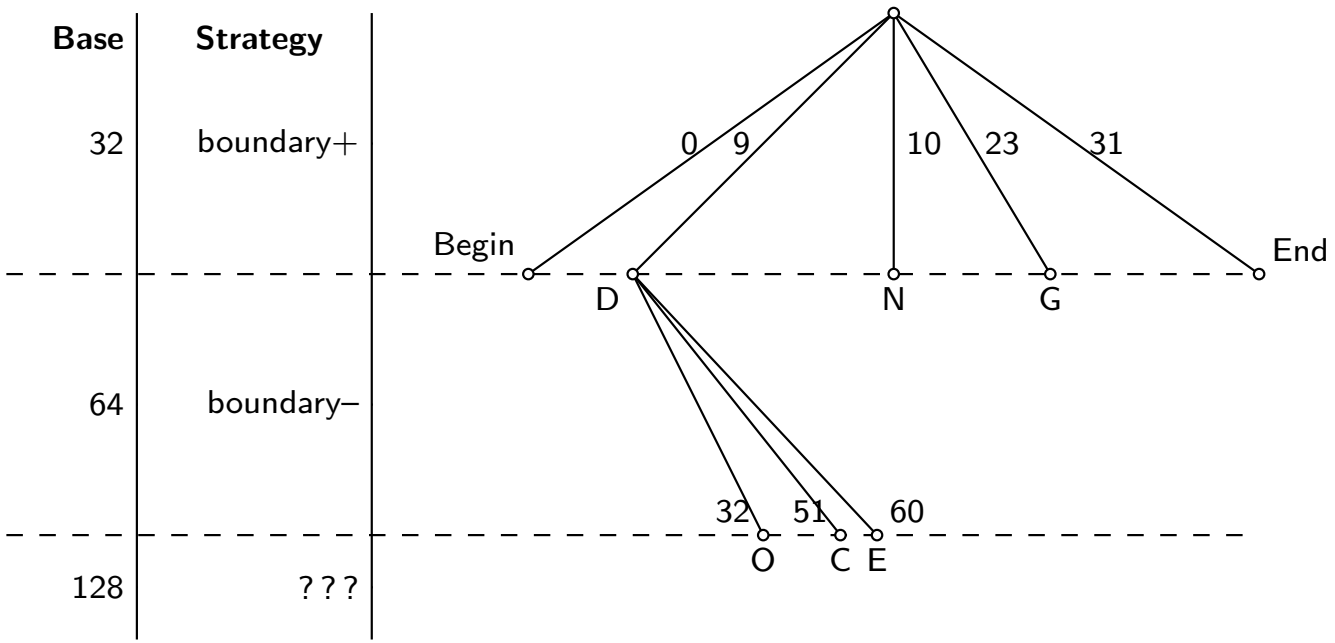
Intuition

As it is complex to predict the editing behaviour, some depths of the tree on a given path can be lost if the reward **compensates** the loss.
In other terms, even if LSEQ chooses the wrong strategy at a given time, it will eventually choose the good one, and that choice will **amortize** the cost of all previous lost depths.

Example : use case DNG ? Wait... DOCENG !

Three components :

- Exponential tree
  - ⇒ 32 nodes maximum at depth-1, 64 nodes maximum at depth-2...
- Two allocation strategies
  - ⇒ : designed for end-editing
  - ⇒ : designed for front-editing
- Random strategy choice

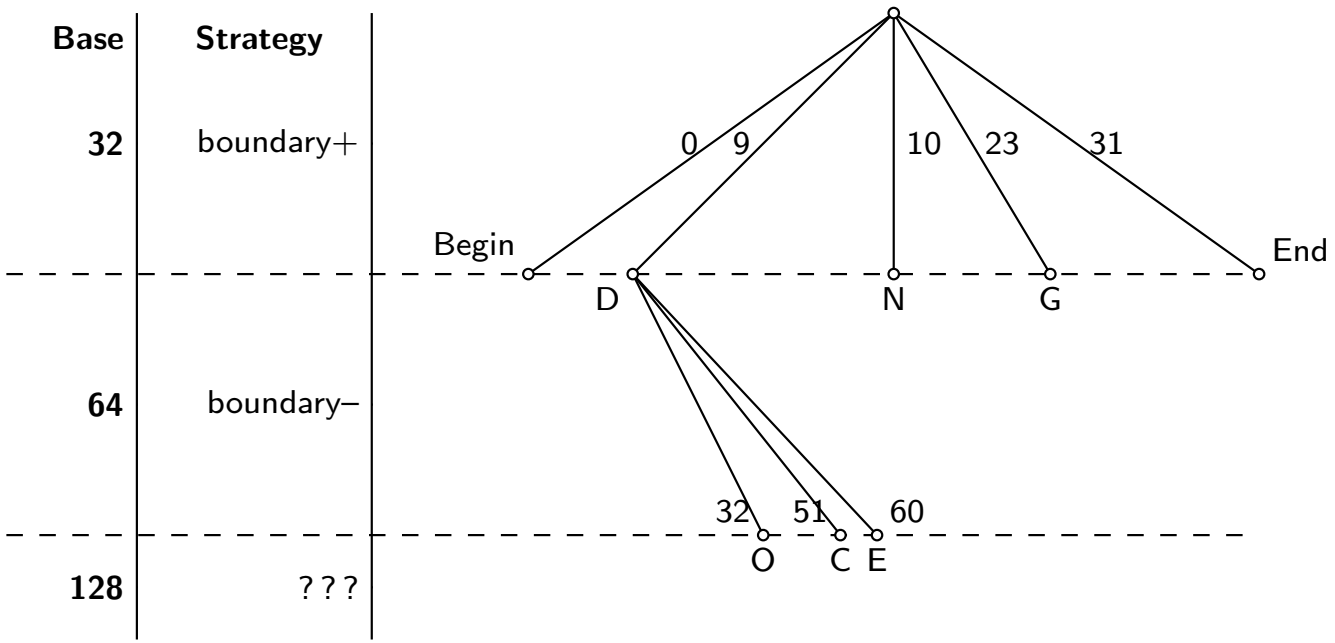


1. insert(Begin,D,End)
2. insert(D,N,End)
3. insert(N,G,End)
4. insert(D,E,N)
   - ⇒ no room
   - ⇒ randomize strat
   - ⇒ front-editing
5. insert(D,C,E)
6. insert(D,O,C)

Example : use case DNG ? Wait... DOCENG !

Three components :

- Exponential tree
  - $\Rightarrow$ 32 nodes maximum at depth-1, 64 nodes maximum at depth-2...
- Two allocation strategies
  - $\Rightarrow$: designed for end-editing
  - $\Rightarrow$: designed for front-editing
- Random strategy choice

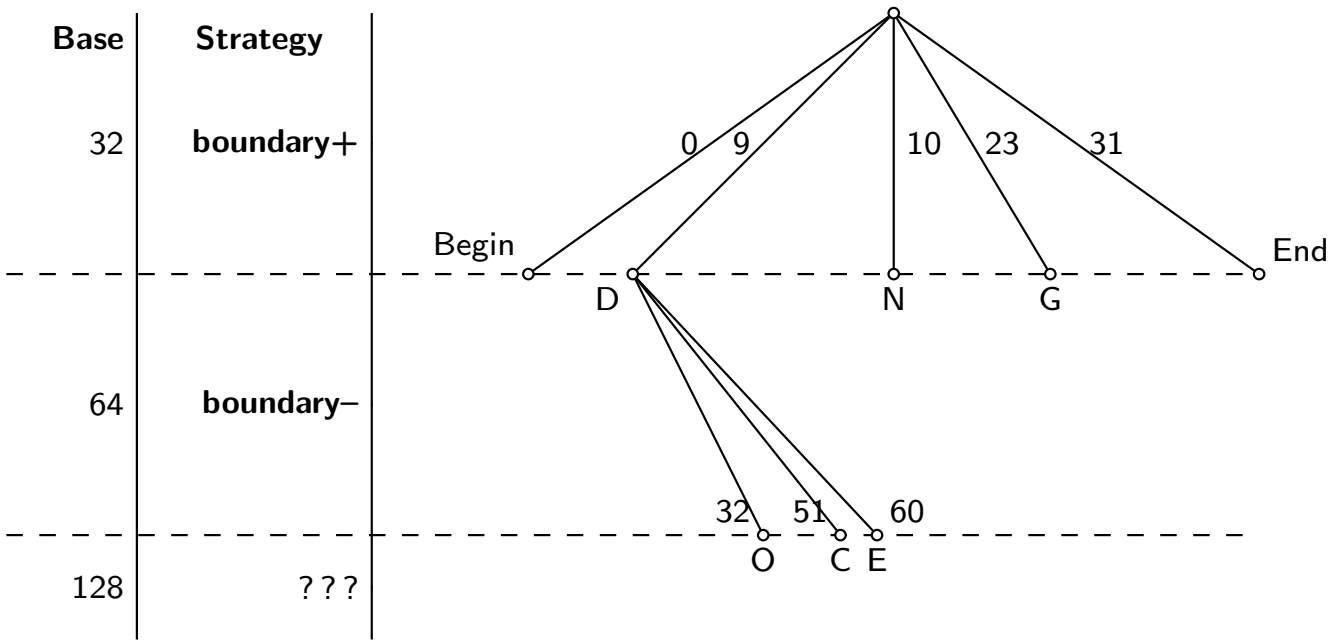


1. insert(Begin,D,End)
2. insert(D,N,End)
3. insert(N,G,End)
4. insert(D,E,N)
   - $\Rightarrow$ no room
   - $\Rightarrow$ randomize strat
   - $\Rightarrow$ front-editing
5. insert(D,C,E)
6. insert(D,O,C)

Example : use case DNG ? Wait... DOCENG !

Three components :

- Exponential tree
  - $\Rightarrow$ 32 nodes maximum at depth-1, 64 nodes maximum at depth-2...
- Two allocation strategies
  - $\Rightarrow$: designed for end-editing
  - $\Rightarrow$: designed for front-editing
- Random strategy choice

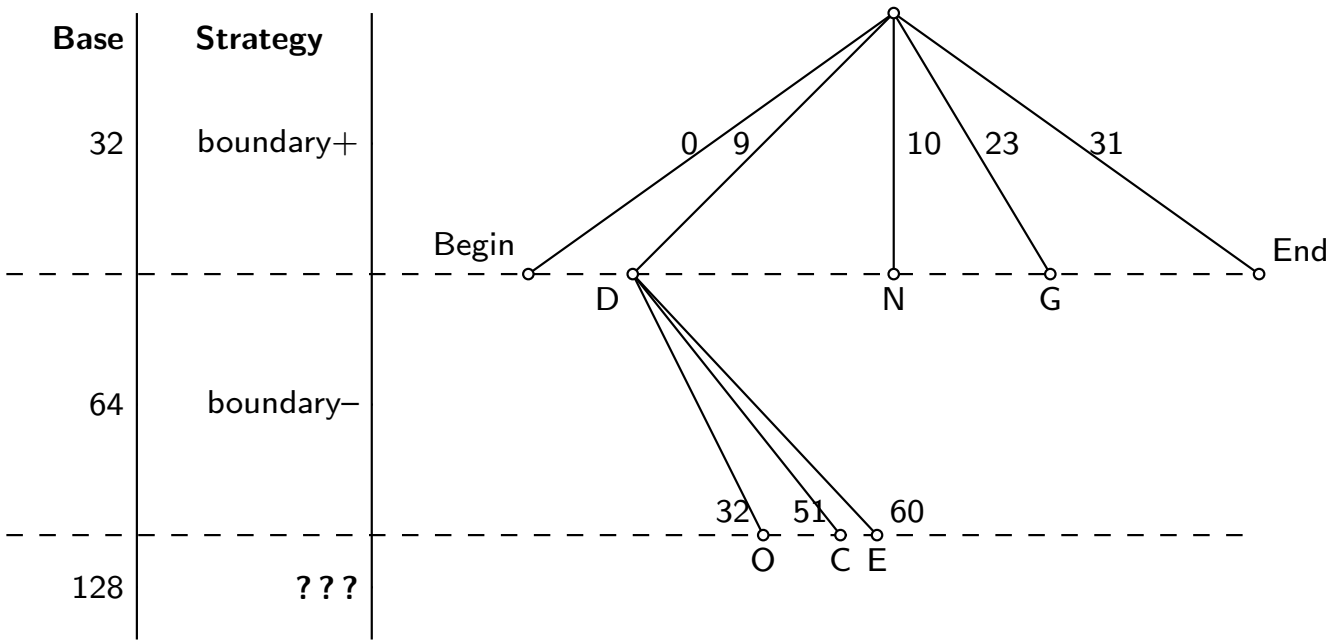


1. insert(Begin,D,End)
2. insert(D,N,End)
3. insert(N,G,End)
4. insert(D,E,N)
   - $\Rightarrow$ no room
   - $\Rightarrow$ randomize strat
   - $\Rightarrow$ front-editing
5. insert(D,C,E)
6. insert(D,O,C)

Example : use case DNG ? Wait... DOCENG !

Three components :

- Exponential tree
  - $\Rightarrow$ 32 nodes maximum at depth-1, 64 nodes maximum at depth-2...
- Two allocation strategies
  - $\Rightarrow$: designed for end-editing
  - $\Rightarrow$: designed for front-editing
- Random strategy choice

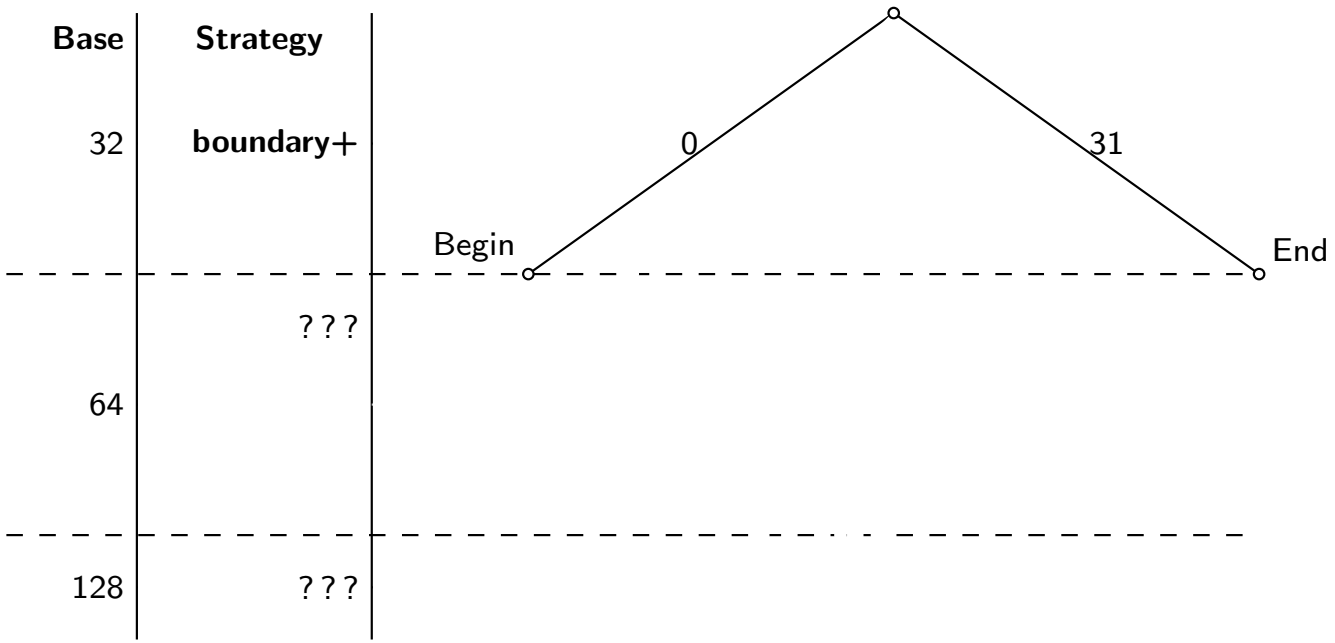


1. insert(Begin,D,End)
2. insert(D,N,End)
3. insert(N,G,End)
4. insert(D,E,N)
   - $\Rightarrow$ no room
   - $\Rightarrow$ randomize strat
   - $\Rightarrow$ front-editing
5. insert(D,C,E)
6. insert(D,O,C)

Example : use case DNG ? Wait... DOCENG !

Three components :

- Exponential tree
  - ⇒ 32 nodes maximum at depth-1, 64 nodes maximum at depth-2...
- Two allocation strategies
  - ⇒ : designed for end-editing
  - ⇒ : designed for front-editing
- Random strategy choice

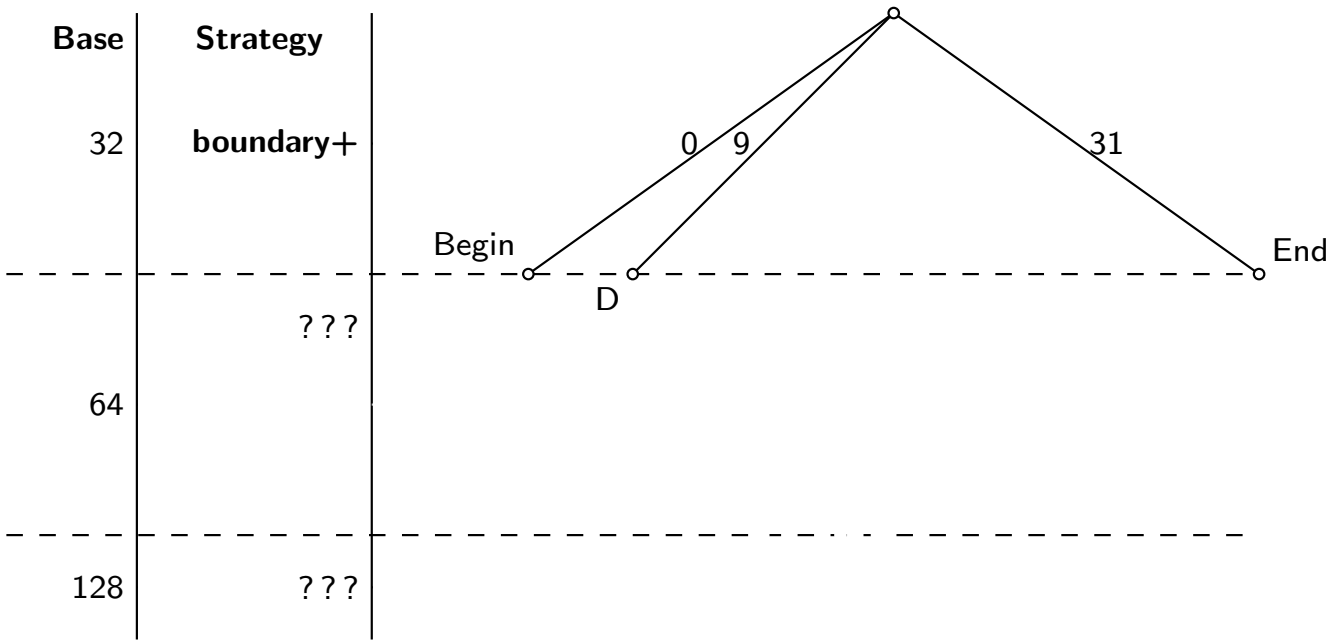


1. insert(Begin,D,End)
2. insert(D,N,End)
3. insert(N,G,End)
4. insert(D,E,N)
   - ⇒ no room
   - ⇒ randomize strat
   - ⇒ front-editing
5. insert(D,C,E)
6. insert(D,O,C)

Example : use case DNG ? Wait... DOCENG !

Three components :

- Exponential tree
  - ⇒ 32 nodes maximum at depth-1, 64 nodes maximum at depth-2...
- Two allocation strategies
  - ⇒ : designed for end-editing
  - ⇒ : designed for front-editing
- Random strategy choice

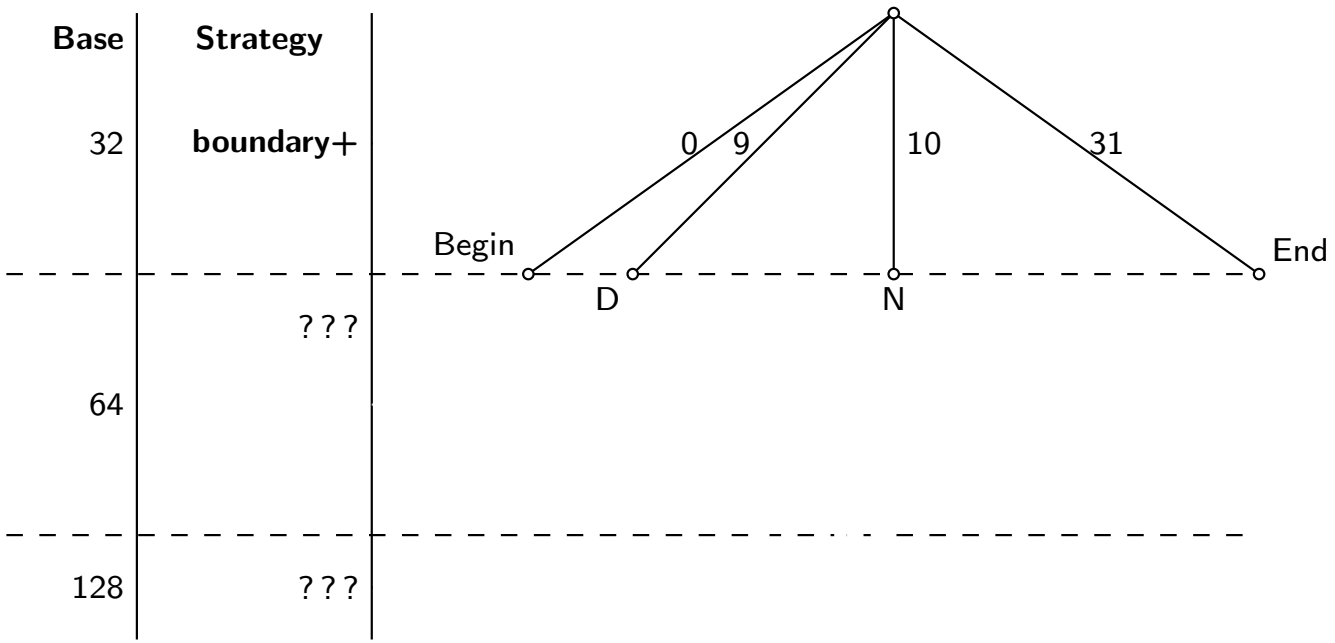


1. insert(Begin,D,End)
2. insert(D,N,End)
3. insert(N,G,End)
4. insert(D,E,N)
   - ⇒ no room
   - ⇒ randomize strat
   - ⇒ front-editing
5. insert(D,C,E)
6. insert(D,O,C)

Example : use case DNG ? Wait... DOCENG !

Three components :

- Exponential tree
  - ⇒ 32 nodes maximum at depth-1, 64 nodes maximum at depth-2...
- Two allocation strategies
  - ⇒ : designed for end-editing
  - ⇒ : designed for front-editing
- Random strategy choice

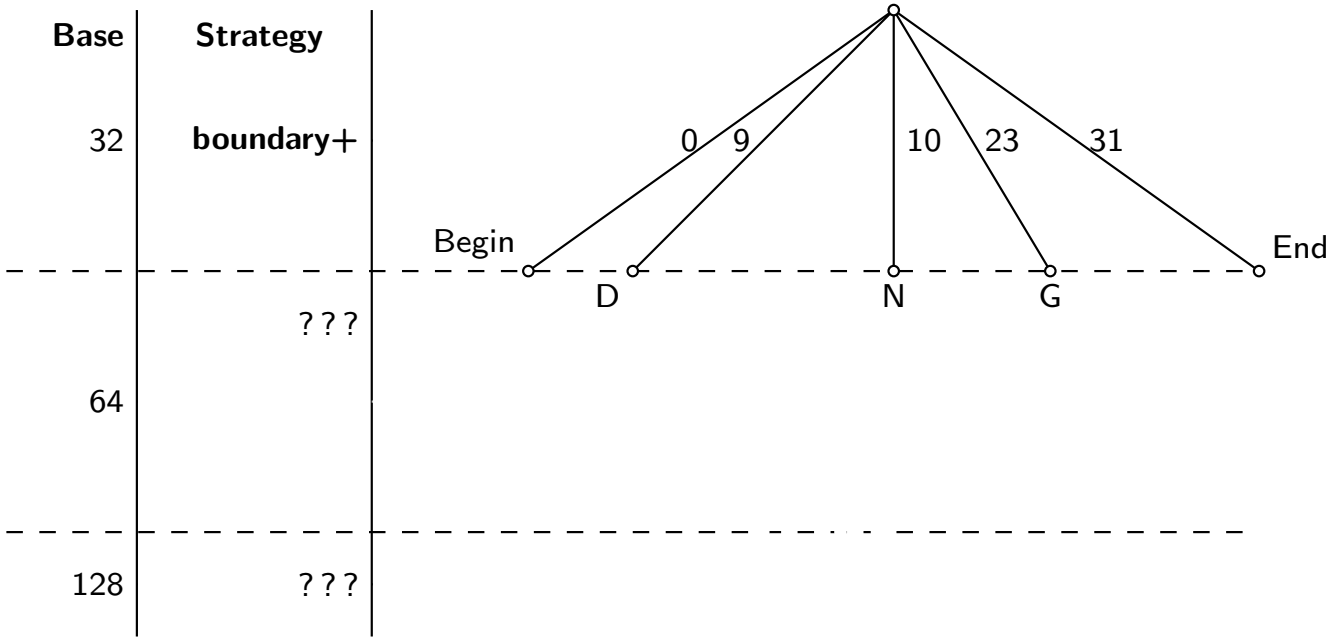


1. insert(Begin,D,End)
2. insert(D,N,End)
3. insert(N,G,End)
4. insert(D,E,N)
   - ⇒ no room
   - ⇒ randomize strat
   - ⇒ front-editing
5. insert(D,C,E)
6. insert(D,O,C)

Example : use case DNG ? Wait... DOCENG !

Three components :

- Exponential tree
  - ⇒ 32 nodes maximum at depth-1, 64 nodes maximum at depth-2...
- Two allocation strategies
  - ⇒ : designed for end-editing
  - ⇒ : designed for front-editing
- Random strategy choice

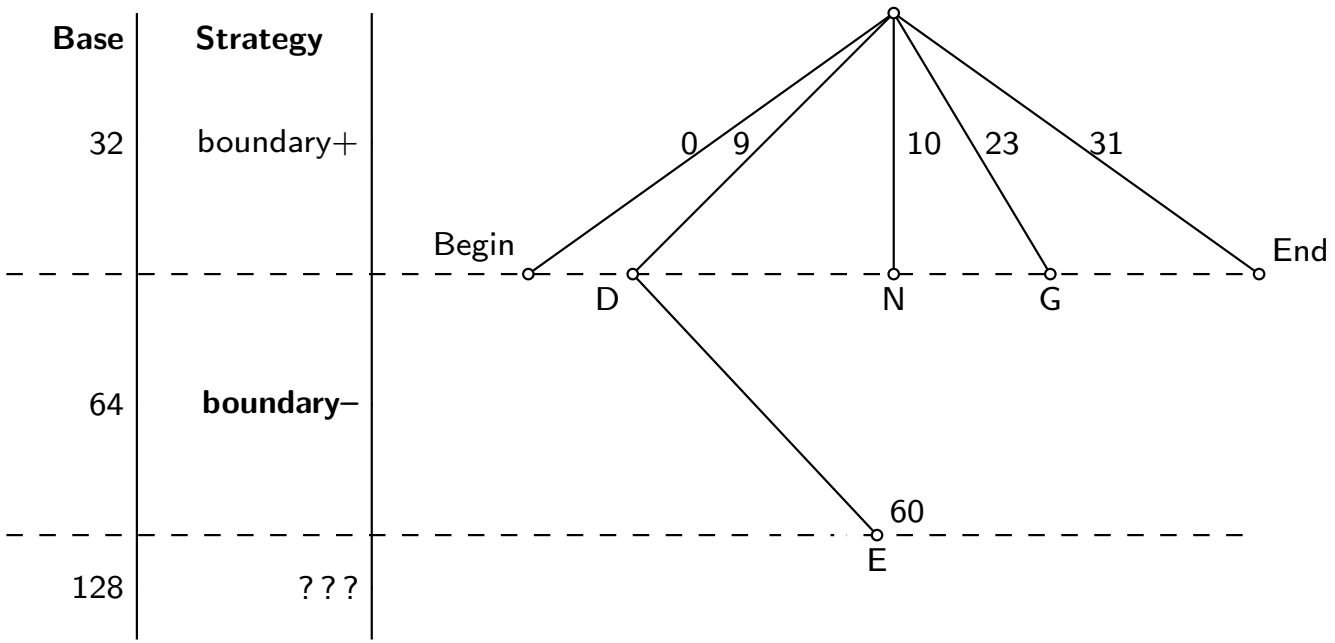


1. insert(Begin,D,End)
2. insert(D,N,End)
3. insert(N,G,End)
4. insert(D,E,N)
   - ⇒ no room
   - ⇒ randomize strat
   - ⇒ front-editing
5. insert(D,C,E)
6. insert(D,O,C)

Example : use case DNG ? Wait... DOCENG !

Three components :

- Exponential tree
  - ⇒ 32 nodes maximum at depth-1, 64 nodes maximum at depth-2...
- Two allocation strategies
  - ⇒ : designed for end-editing
  - ⇒ : designed for front-editing
- Random strategy choice

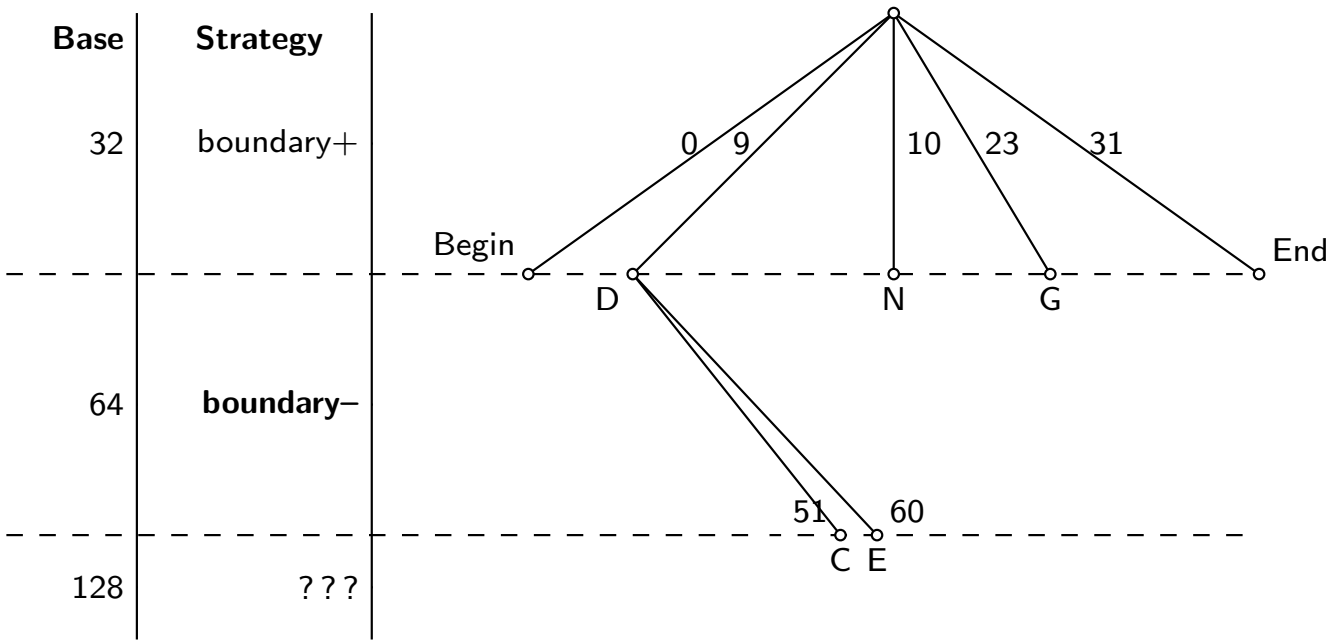


1. insert(Begin,D,End)
2. insert(D,N,End)
3. insert(N,G,End)
4. insert(D,E,N)
   - ⇒ no room
   - ⇒ randomize strat
   - ⇒ front-editing
5. insert(D,C,E)
6. insert(D,O,C)

Example : use case DNG ? Wait... DOCENG !

Three components :

- Exponential tree
  - $\Rightarrow$ 32 nodes maximum at depth-1, 64 nodes maximum at depth-2...
- Two allocation strategies
  - $\Rightarrow$: designed for end-editing
  - $\Rightarrow$: designed for front-editing
- Random strategy choice

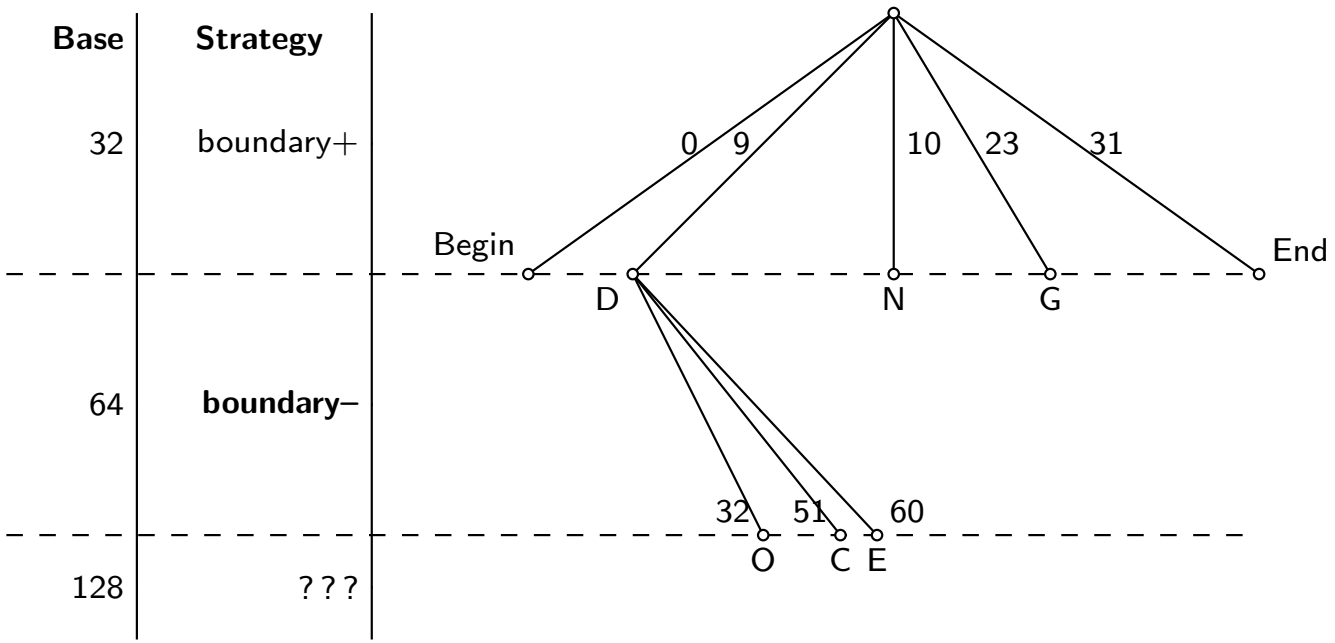


1. insert(Begin,D,End)
2. insert(D,N,End)
3. insert(N,G,End)
4. insert(D,E,N)
   - $\Rightarrow$ no room
   - $\Rightarrow$ randomize strat
   - $\Rightarrow$ front-editing
5. insert(D,C,E)
6. insert(D,O,C)

Example : use case DNG ? Wait... DOCENG !

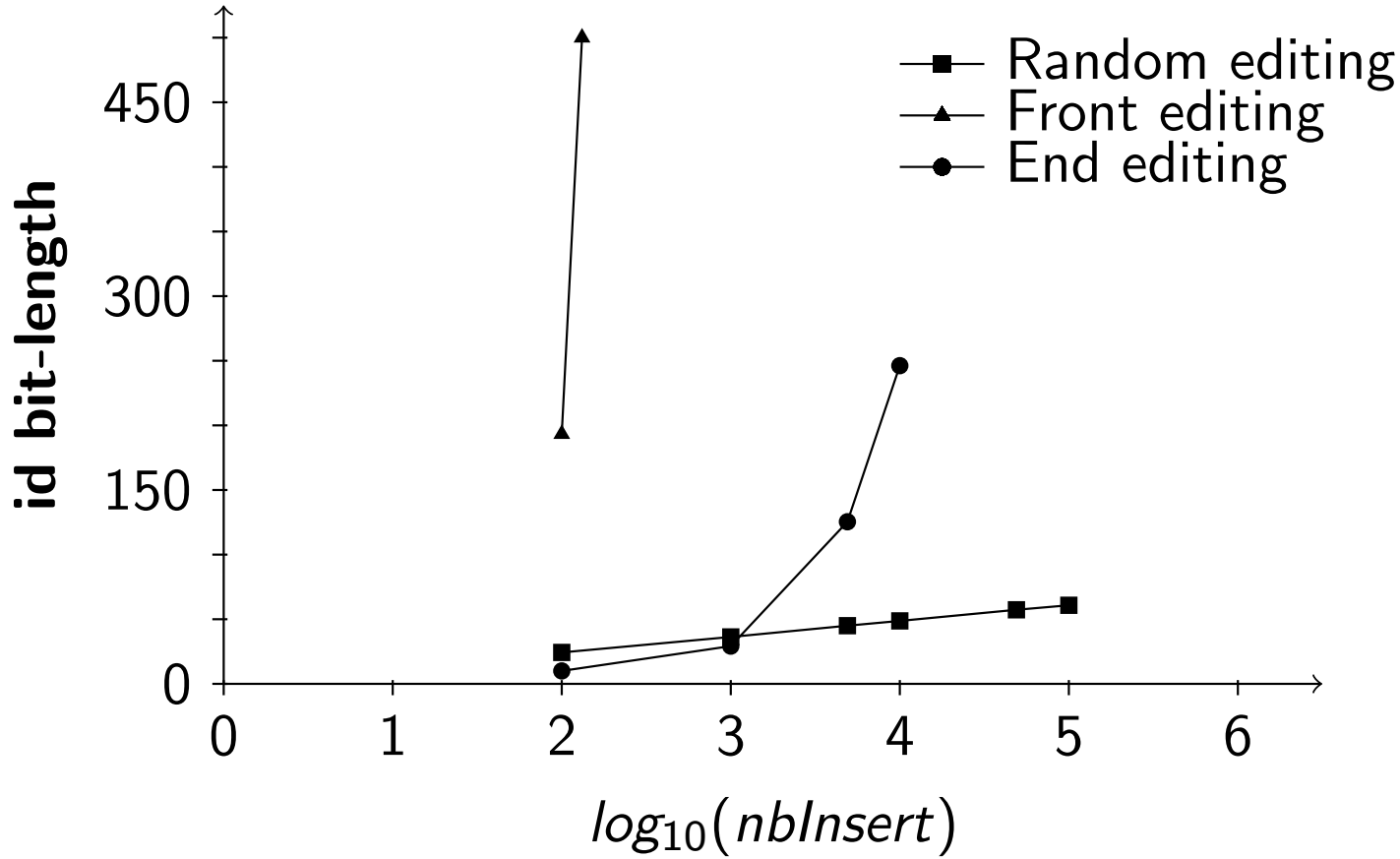
Three components :

- Exponential tree
  - $\Rightarrow$ 32 nodes maximum at depth-1, 64 nodes maximum at depth-2...
- Two allocation strategies
  - $\Rightarrow$: designed for end-editing
  - $\Rightarrow$: designed for front-editing
- Random strategy choice



1. insert(Begin,D,End)
2. insert(D,N,End)
3. insert(N,G,End)
4. insert(D,E,N)
   - $\Rightarrow$ no room
   - $\Rightarrow$ randomize strat
   - $\Rightarrow$ front-editing
5. insert(D,C,E)
6. insert(D,O,C)

Back in time : Our baseline



- random : logarithmic $\Rightarrow$ very good
- front and end : linear $\Rightarrow$ bad
- editing behaviour dependant $\Rightarrow$ bad

Exponential tree

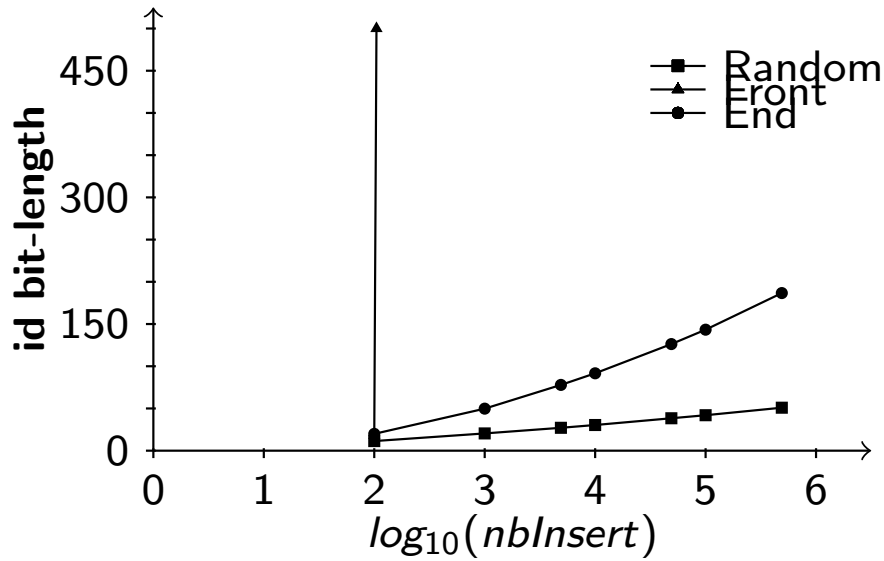
lower the space-complexity but still **editing behaviour dependant**

Each node has twice more children than its parent

$\Rightarrow$ exponential $\nearrow$ number of nodes when depth $\nearrow$

Each number in the id costs 1 more bit to encode

$+\mathbf{1}$ bit $\Rightarrow$ **x2** identifiers



Intuition

If the number of insert operations is **low**, the id bit-length can stay **small**. On the other hand, when the number of insertions **increases**, it is profitable to allocate **larger** identifiers.

Multiple allocation strategies & Random strategy choice

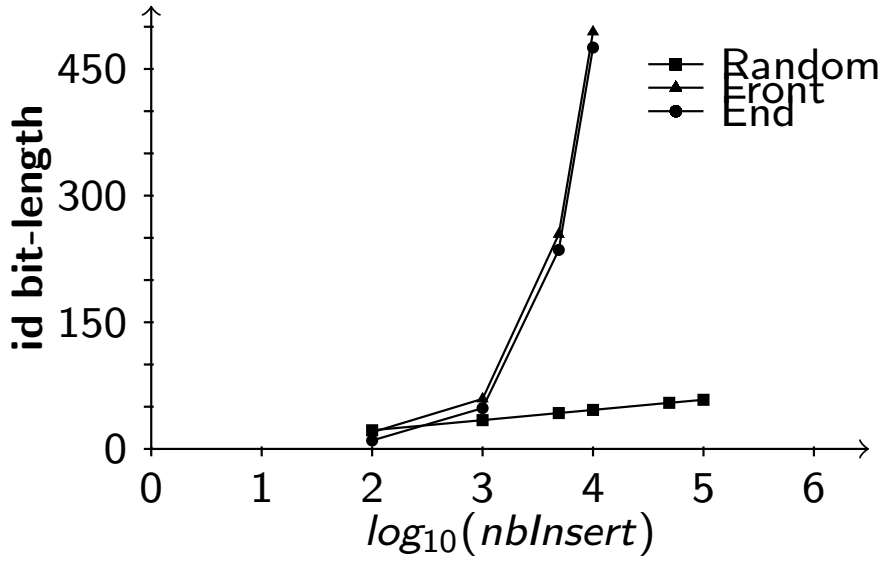
solve the editing behaviour dependency but still **linear**

Two sub allocation strategy :

- ⇒ 1 designed for end-editing
- ⇒ 1 designed for front-editing

Allocation strategy chosen :

- ⇒ randomly
- ⇒ when id size increases



Intuition : Why

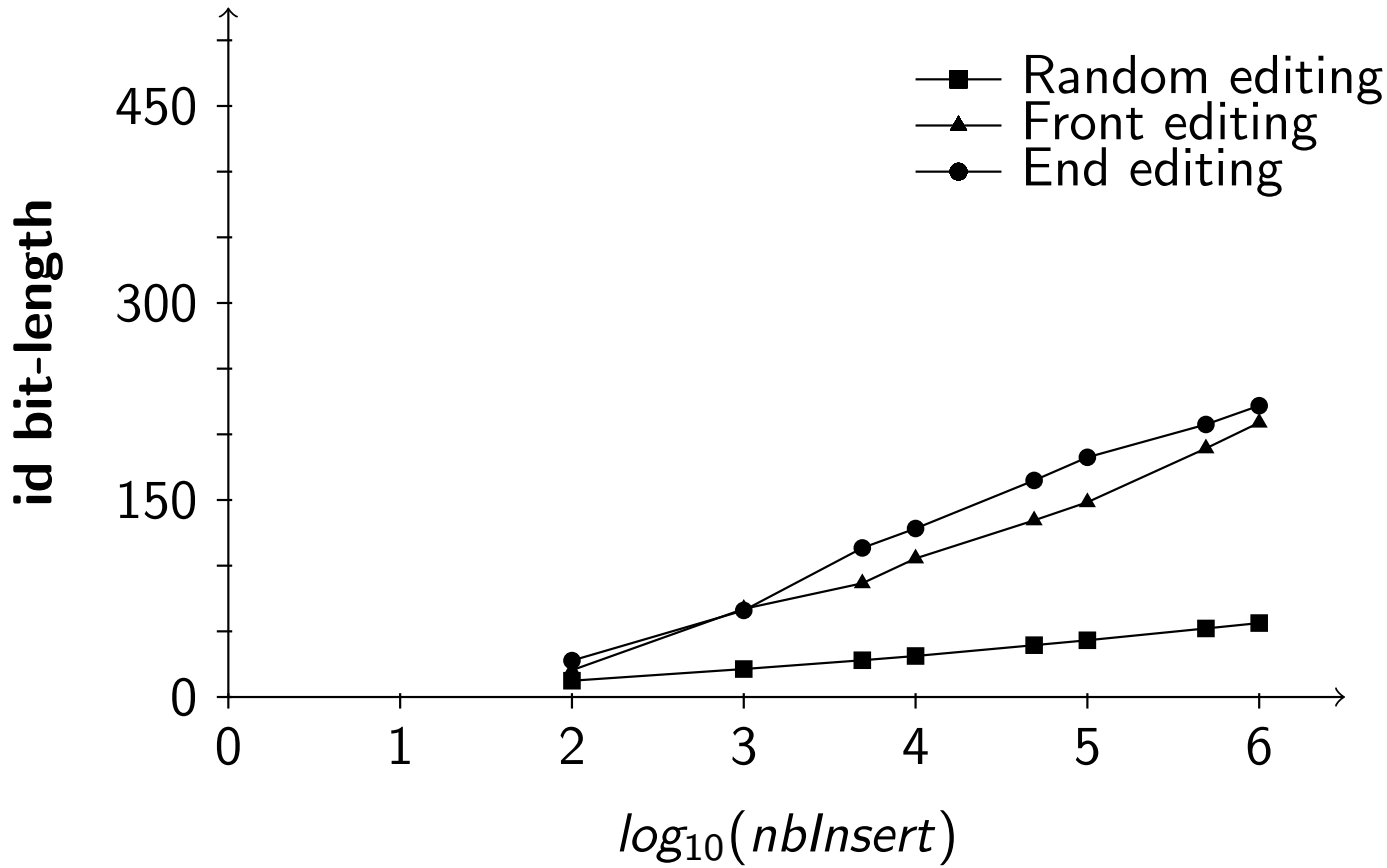
The end-editing allocation strategy is **not sufficient** to be employed as a safe allocation strategy. However, by using its antagonist strategy, each strategy **cancels** the **deficiency** of the other.

Intuition : Which

Since it is impossible to know *a priori* the editing behaviour, the strategy choice should **not favorize any behaviour**.

LSEQ solves everything !

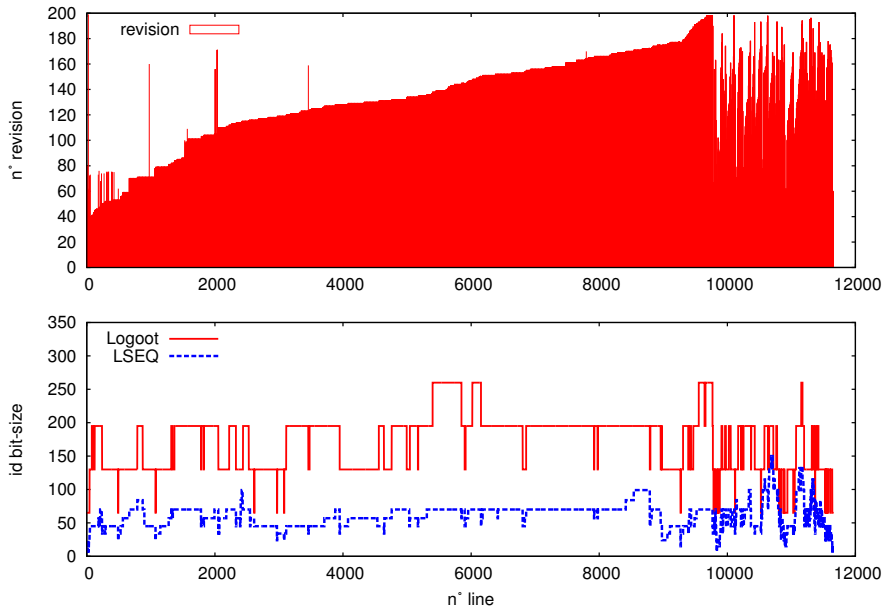
LSEQ = exponential tree + two sub allocation strategies + random strategy choice

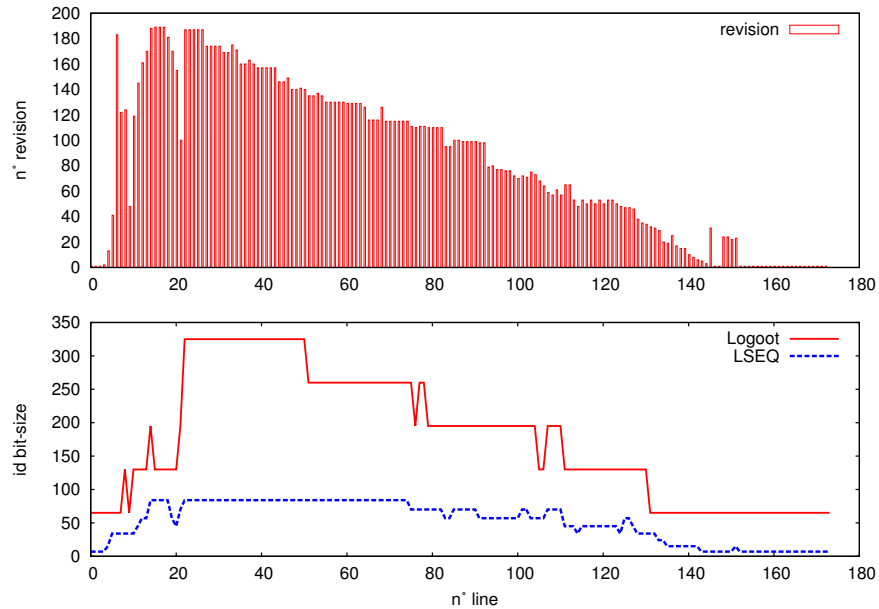


- front and end : sub-linear behaviour
- random : logarithmic

LSEQ outperforms variable-size CRDT on real documents

extracted from the English Wikipedia





		Logoot	**LSEQ**
id-bit-length	avg	169.7	**61.24**
	max	256	150

Page mostly end-edited.

		Logoot	**LSEQ**
id-bit-length	avg	172.25	**51.99**
	max	320	84

Page mostly front-edited.

Conclusion

LSEQ :

- Allocation strategy for variable-size CRDTs for sequences
  - $\Rightarrow$ reusable (*Example : Treedoc*)
- With 3 components :
  - exponential tree
  - 2 strategies : *boundary+* and *boundary–*
  - random strategy choice

Experiments show :

- LSEQ has a sub-linear behaviour
  - under the assumption that it works in context involving concurrency
  - $\Rightarrow$ No additionnal rebalance protocol required
  - $\Rightarrow$ safely usable : large open distributed network with churn
- Better performance over Logoot :
  - good case of Logoot – end editing
  - bad case of Logoot – front editing

Distributed Collaborative Editor $\not\Rightarrow$ single point of failure

- LSEQ is not safe when **multiple users** are involved due to different strategy choices. It has been improved by a shared hash function.
  $\Rightarrow$ upcoming talk...
- **Goal** : create Distributed Collaborative Editors which is
  1. decentralized (privacy, availability, repartition of charge...)
  2. scalable and simple
  3. short overhead

Future works

1. Proof on space complexity
   - n operations : uniform distribution $\Rightarrow O((log\ log\ n)^2)$
   - n operations : monotononic $\Rightarrow O((log\ n)^2)$
   - n operations : worst-case $\Rightarrow O(n^2)$
2. Proof : worst-case happens with a negligible probability
3. Concurrency effect : Latency, multiple-users ?
4. Causality tracking (still an issue in distributed systems with churn)
   - does not scale in term of user
   - or does not provide exact causality

   $\Rightarrow$ CRDTs for sequences require causality...
5. Develop the distributed collaborative editor

Thank you !