Sortering

I. MERGE-SORT
QUICK-SORT

II. SAMMENLIKNINGSBASERTE SORTERINGSALGORITMER
NEDERE SKRANKE FOR KOMPLEKSITET

III. BUCKET-SORT

Kap.10: (kursorisk: 10.1.2, 10.1.3, 10.2, 10.4, 10.5.2, 10.6
unntatt: 10.3.2, 10.7, 10.8)

Sortering

- Bubble $\Theta(n^2)$

prioritetskø basert

- Selection $\Theta(n^2)$
- Insertion $O(n^2)$
- Heap $O(n \log n)$

‘‘splitt og hersk’’

- Merge $O(n \log n)$

Sortering

- Bubble $\Theta(n^2)$

prioritetskø basert

- Selection $\Theta(n^2)$
- Insertion $O(n^2)$
- Heap $O(n \log n)$

‘‘splitt og hersk’’

- Merge $O(n \log n)$
- **Quick** O **(n log n)**

Sortering

- Bubble $\Theta(n^2)$

prioritetskø basert

- Selection $\Theta(n^2)$
- Insertion $O(n^2)$
- Heap $O(n \log n)$

"splitt og hersk"

- Merge $O(n \log n)$
- Quick $O(n \log n)$

"Comparison based sorting" = Ω (n log n)

Sortering

- Bubble $\Theta(n^2)$

prioritetskø basert

- Selection $\Theta(n^2)$
- Insertion $O(n^2)$
- Heap $O(n \log n)$

‘‘splitt og hersk’’

- Merge $O(n \log n)$
- Quick $O(n \log n)$

‘‘Comparison based sorting’’ = $\Omega(n \log n)$

- Bucket $O(n + N)$
- Radix

Merge-Sort

MS(*S*) :

Splitt : hvis *S* har minst 2 elementer
del *S* i midten i to like lange (+/– 1) subsekvenser *S1* og *S2*

Rekursjon : sorter rekursivt **MS**(*S1*) og **MS**(*S2*)

Hersk : flett de sorterte resultatene til en sortert hele

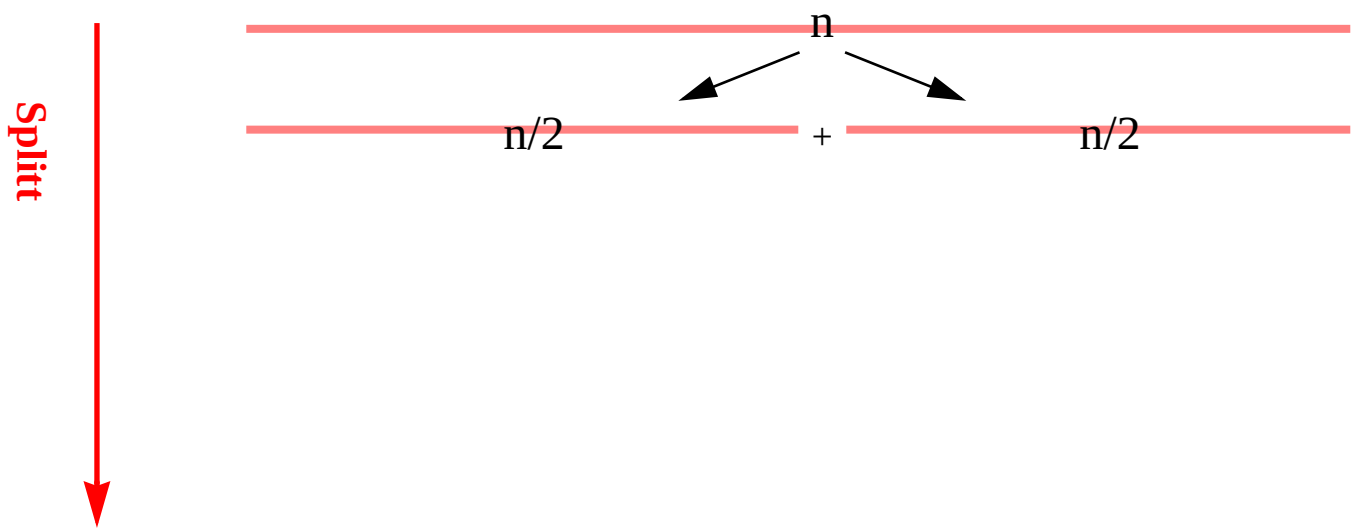
Merge-Sort

MS(*S*) :

Splitt : hvis *S* har minst 2 elementer
del *S* i midten i to like lange (+/– 1) subsekvenser *S1* og *S2*

Rekursjon : sorter rekursivt **MS**(*S1*) og **MS**(*S2*)

Hersk : flett de sorterte resultatene til en sortert hele



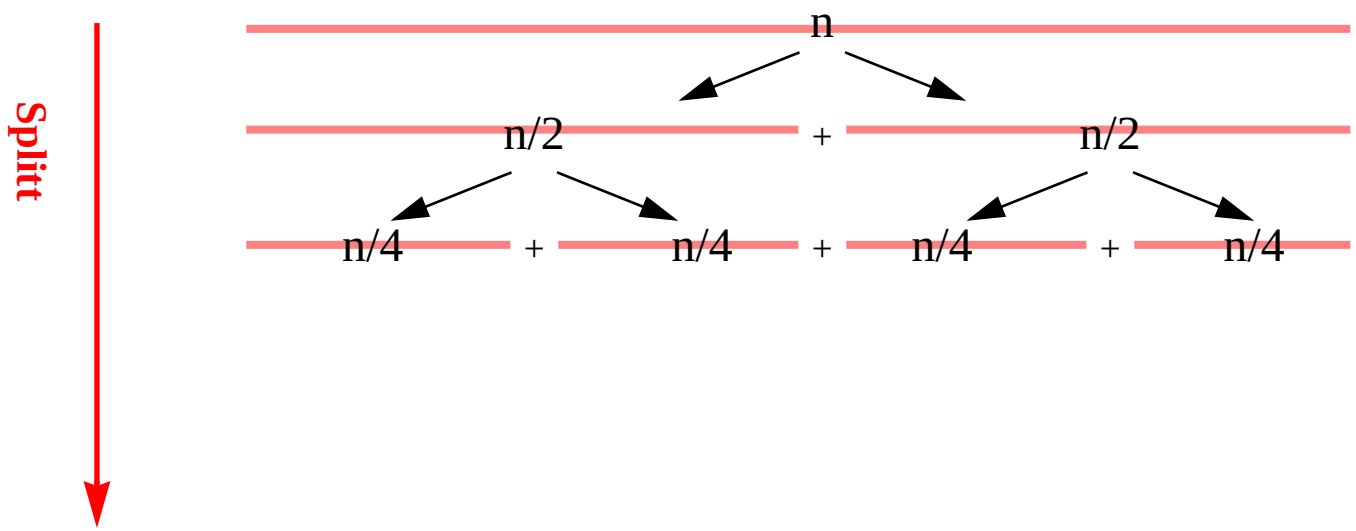
Merge-Sort

MS(*S*) :

Splitt : hvis *S* har minst 2 elementer
del *S* i midten i to like lange (+/– 1) subsekvenser *S1* og *S2*

Rekursjon : sorter rekursivt **MS**(*S1*) og **MS**(*S2*)

Hersk : flett de sorterte resultatene til en sortert hele



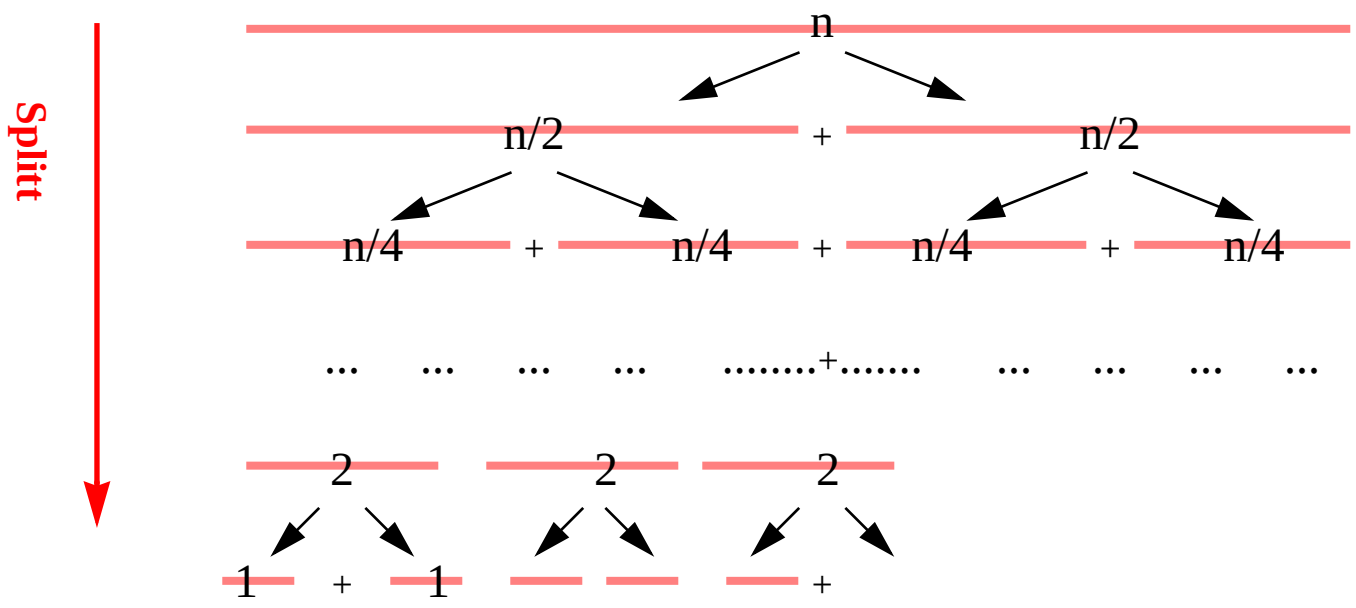
Merge-Sort

MS(*S*) :

Splitt : hvis *S* har minst 2 elementer
del *S* i midten i to like lange (+/– 1) subsekvenser *S1* og *S2*

Rekursjon : sorter rekursivt **MS**(*S1*) og **MS**(*S2*)

Hersk : flett de sorterte resultatene til en sortert hele



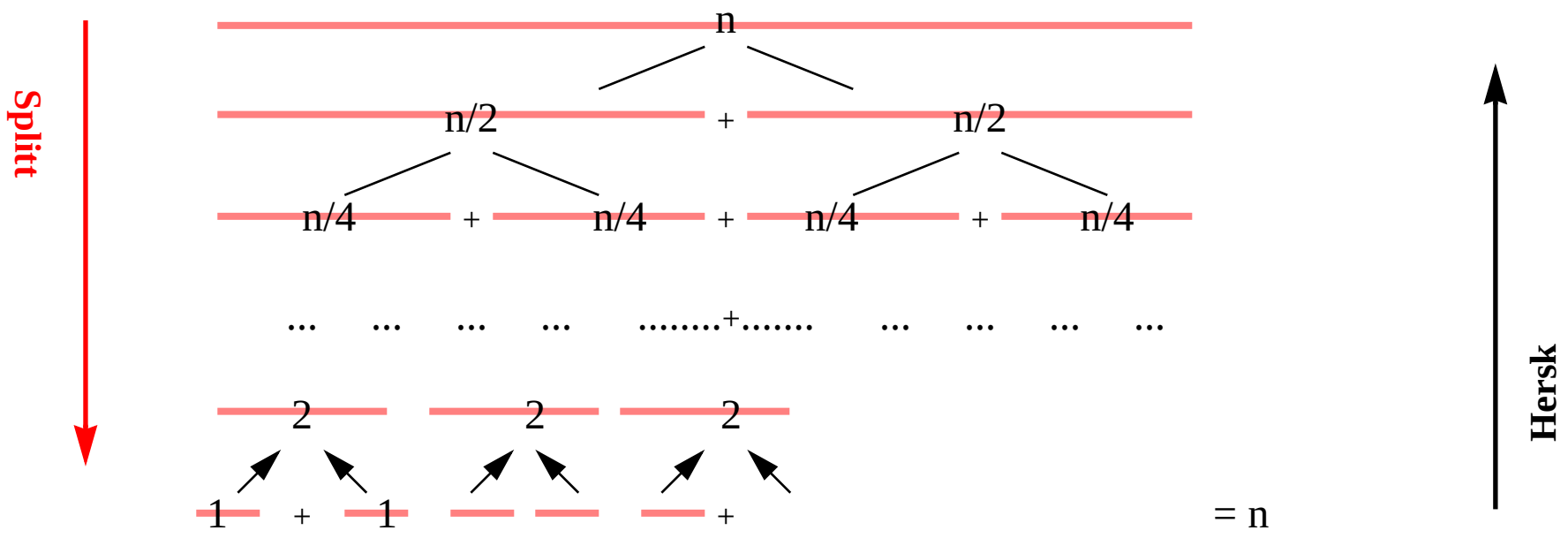
Merge-Sort

MS(*S*) :

Splitt : hvis *S* har minst 2 elementer
del *S* i midten i to like lange (+/– 1) subsekvenser *S1* og *S2*

Rekursjon : sorter rekursivt **MS**(*S1*) og **MS**(*S2*)

Hersk : flett de sorterte resultatene til en sortert hele



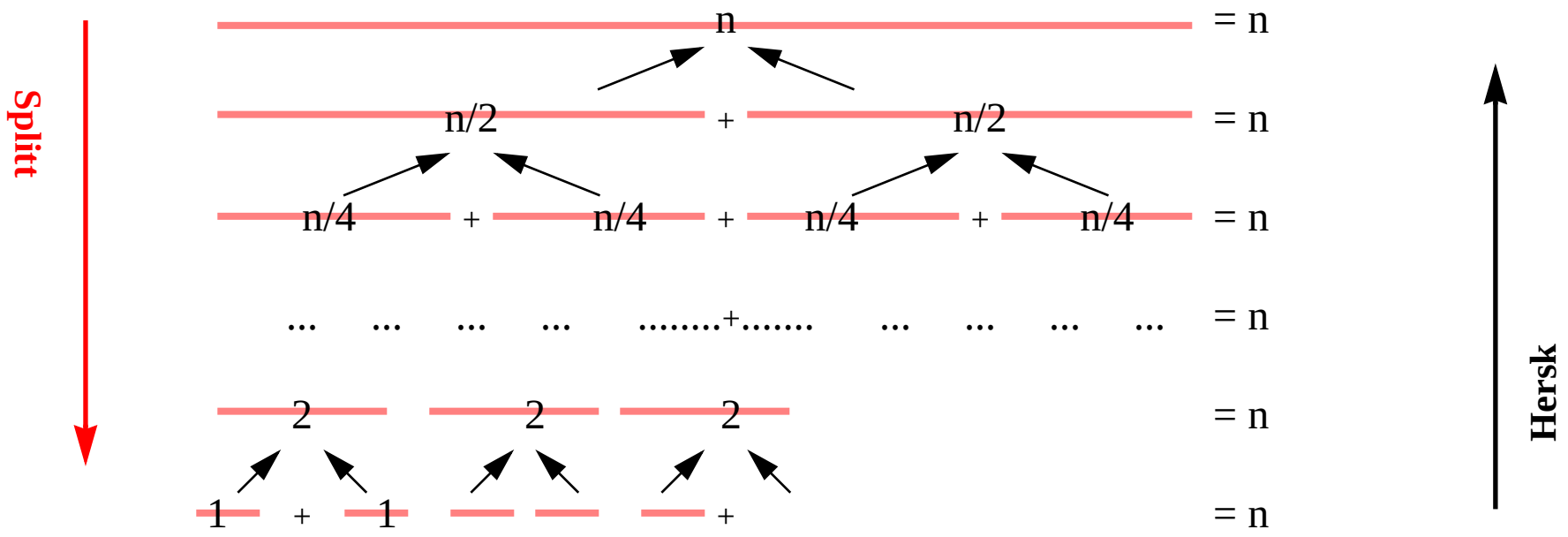
Merge-Sort

MS(*S*) :

Splitt : hvis *S* har minst 2 elementer
del *S* i midten i to like lange (+/– 1) subsekvenser *S1* og *S2*

Rekursjon : sorter rekursivt **MS**(*S1*) og **MS**(*S2*)

Hersk : flett de sorterte resultatene til en sortert hele



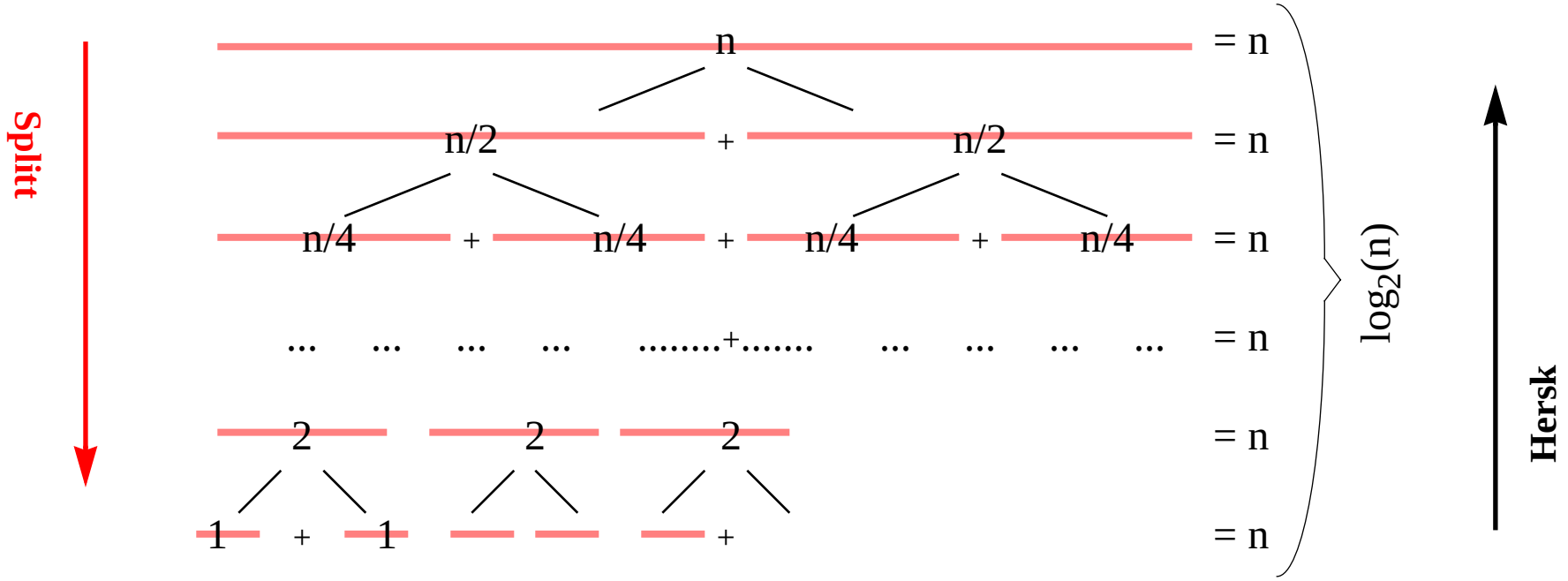
Merge-Sort

MS(*S*) :

Splitt : hvis *S* har minst 2 elementer
del *S* i midten i to like lange (+/– 1) subsekvenser *S1* og *S2*

Rekursjon : sorter rekursivt **MS**(*S1*) og **MS**(*S2*)

Hersk : flett de sorterte resultatene til en sortert hele



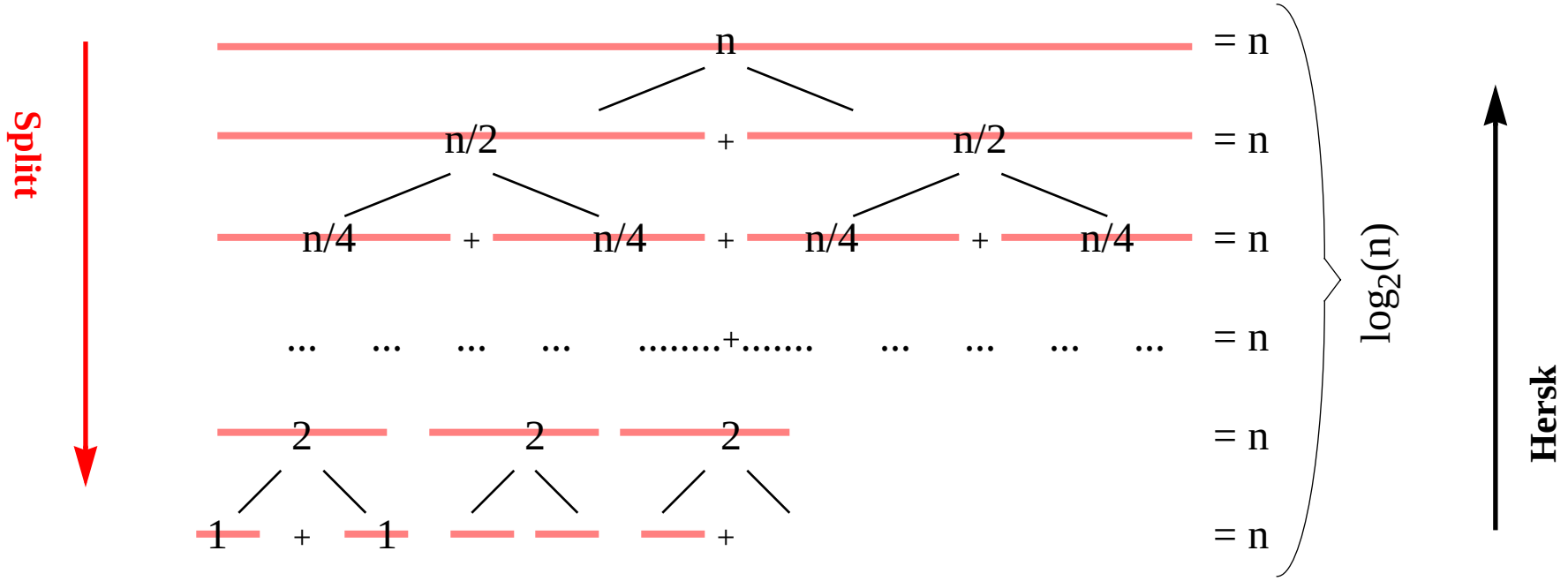
Merge-Sort

MS(*S*) :

Splitt : hvis *S* har minst 2 elementer
del *S* i midten i to like lange (+/– 1) subsekvenser *S1* og *S2*

Rekursjon : sorter rekursivt **MS**(*S1*) og **MS**(*S2*)

Hersk : flett de sorterte resultatene til en sortert hele



$$MS(n) = O\,(n * \log(n))$$

Quick-Sort

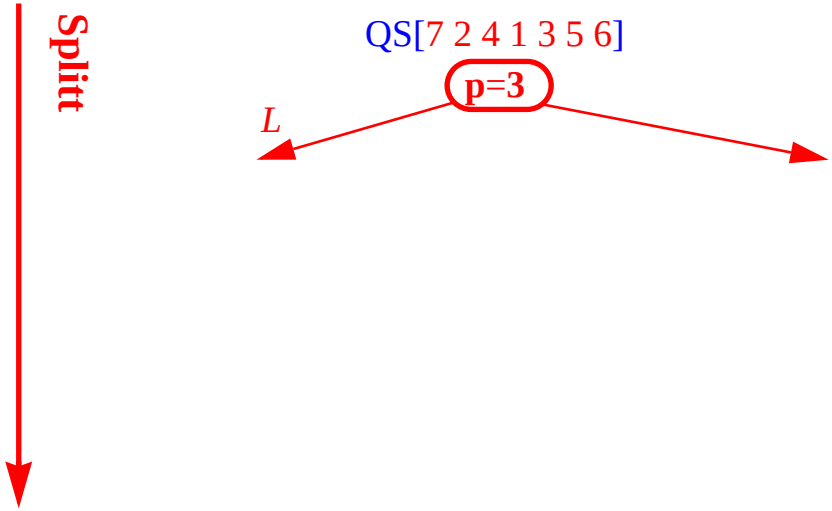
Splitt : hvis S har minst 2 elementer
– velg et element ***p*** fra S (pivot) og
– del S i tre mengder L, E og G slik at :
L inneholder alle elementene mindre enn ***p***
E inneholder alle elementene lik ***p***
G inneholder alle elementene større enn ***p***

Rekursjon : sorter rekursivt **QS**(L) og **QS**(G)

Hersk : returner sekvensen $QS(L)$–E–$QS(G)$

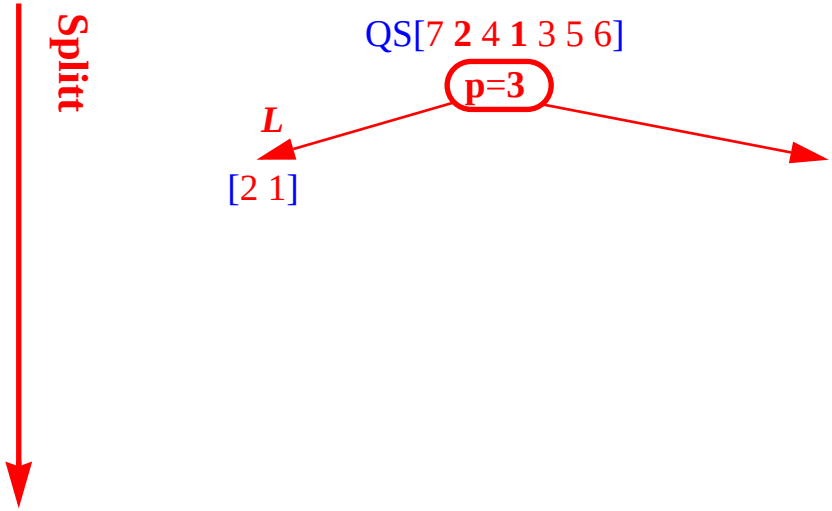
Quick-Sort

Splitt : hvis S har minst 2 elementer
– velg et element ***p*** fra S (pivot) og
– del S i tre mengder L, E og G slik at :
L inneholder alle elementene mindre enn ***p***



Quick-Sort

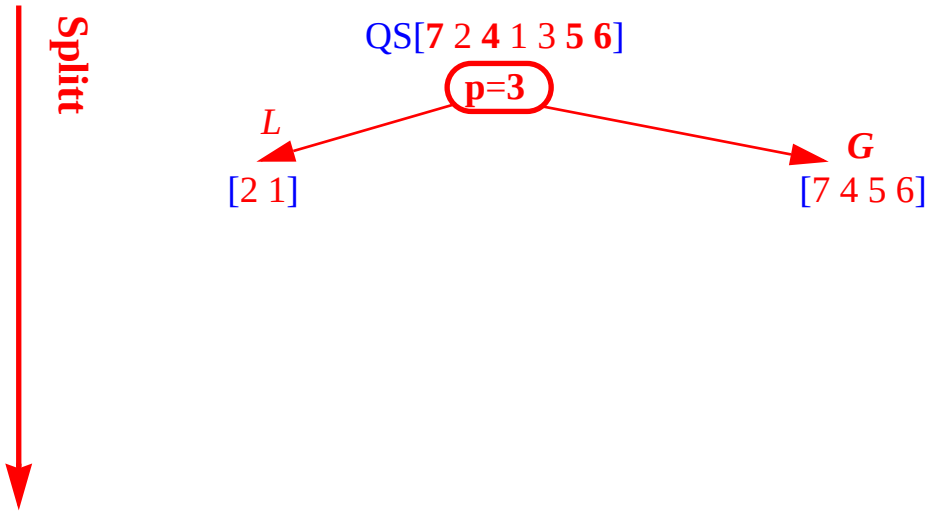
Splitt : hvis S har minst 2 elementer
– velg et element ***p*** fra S (pivot) og
– del S i tre mengder L, E og G slik at :
L inneholder alle elementene mindre enn ***p***



Quick-Sort

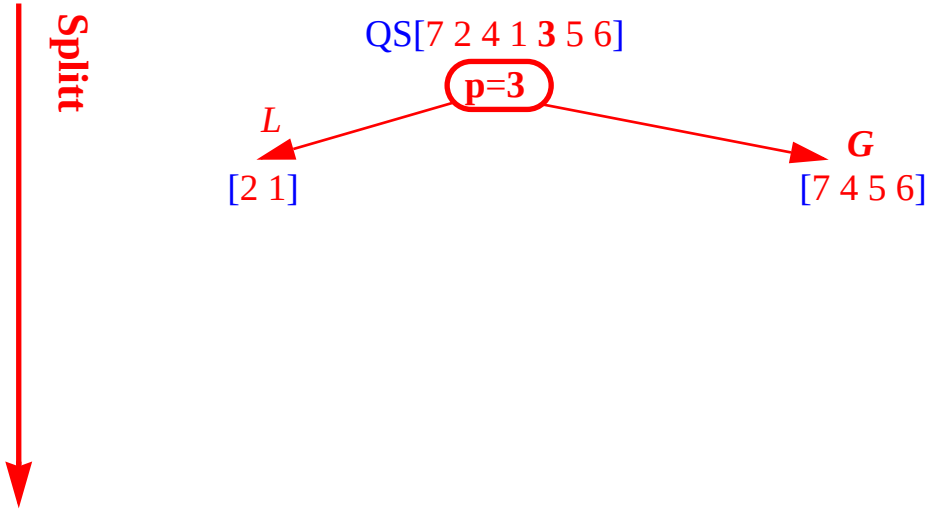
Splitt : hvis *S* har minst 2 elementer
– velg et element ***p*** fra *S* (pivot) og
– del *S* i tre mengder *L*, *E* og *G* slik at :
L inneholder alle elementene mindre enn ***p***

G inneholder alle elementene større enn ***p***



Quick-Sort

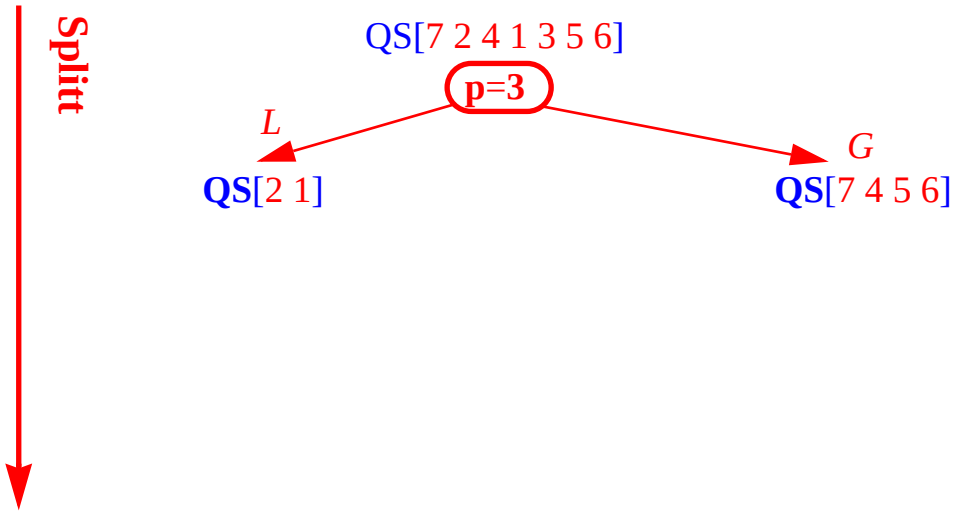
Splitt : hvis *S* har minst 2 elementer
– velg et element ***p*** fra *S* (pivot) og
– del *S* i tre mengder *L*, *E* og *G* slik at :
L inneholder alle elementene mindre enn ***p***
E inneholder alle elementene lik ***p***
G inneholder alle elementene større enn ***p***



Quick-Sort

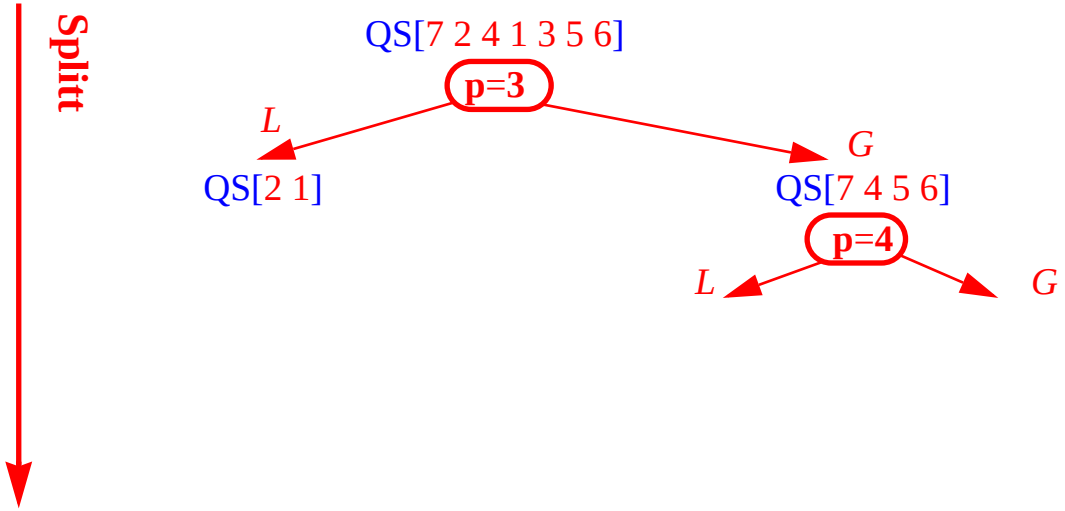
Splitt : hvis S har minst 2 elementer
– velg et element ***p*** fra S (pivot) og
– del S i tre mengder L, E og G slik at :
L inneholder alle elementene mindre enn ***p***
E inneholder alle elementene lik ***p***
G inneholder alle elementene større enn ***p***

Rekursjon : sorter rekursivt **QS**(L) og **QS**(G)



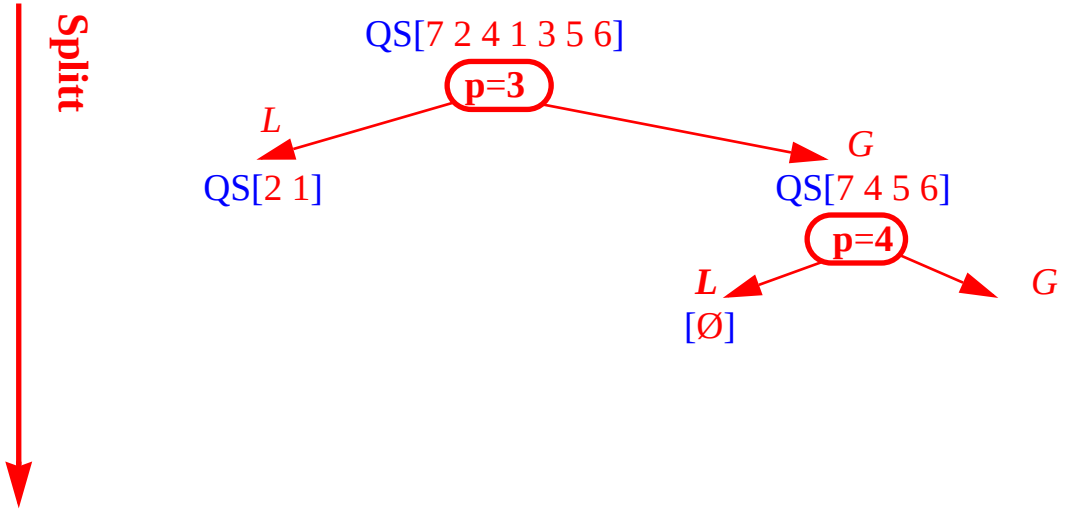
Quick-Sort

Splitt : hvis S har minst 2 elementer
– velg et element ***p*** fra S (pivot) og
– del S i tre mengder L, E og G slik at :



Quick-Sort

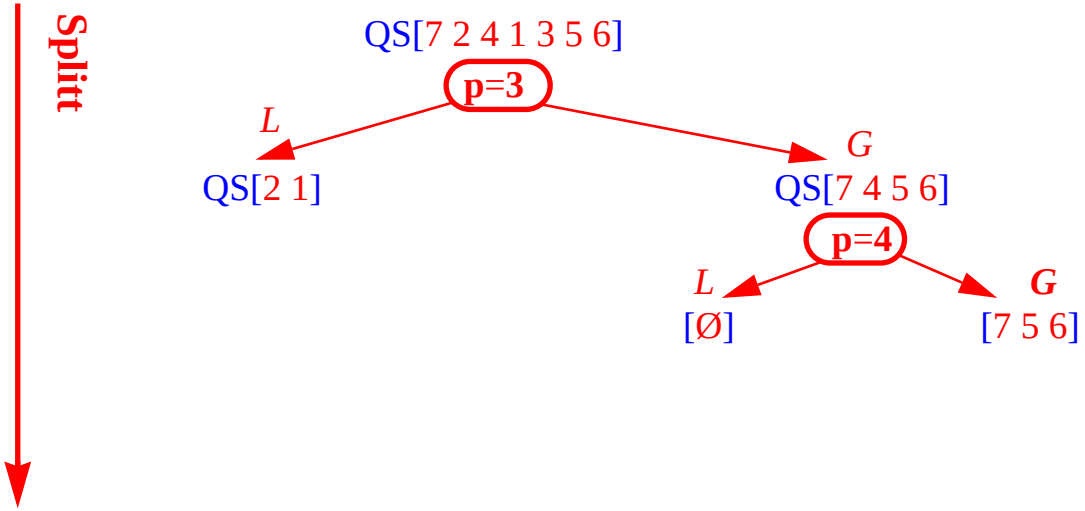
Splitt : hvis *S* har minst 2 elementer
– velg et element ***p*** fra *S* (pivot) og
– del *S* i tre mengder *L*, *E* og *G* slik at :
L inneholder alle elementene mindre enn ***p***



Quick-Sort

Splitt : hvis *S* har minst 2 elementer
– velg et element ***p*** fra *S* (pivot) og
– del *S* i tre mengder *L*, *E* og *G* slik at :
L inneholder alle elementene mindre enn ***p***

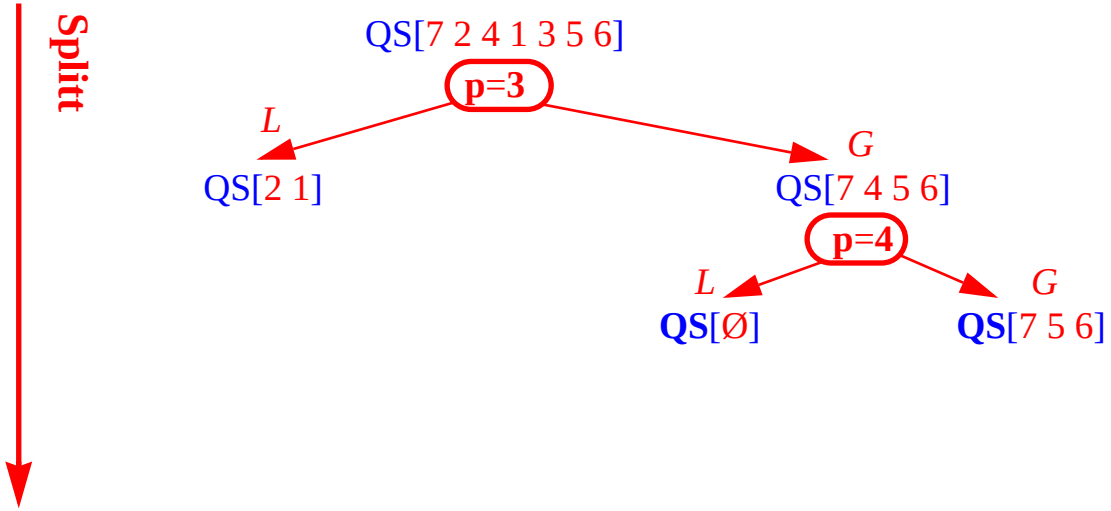
G inneholder alle elementene større enn ***p***



Quick-Sort

Splitt : hvis *S* har minst 2 elementer
– velg et element ***p*** fra *S* (pivot) og
– del *S* i tre mengder *L*, *E* og *G* slik at :
L inneholder alle elementene mindre enn ***p***
E inneholder alle elementene lik ***p***
G inneholder alle elementene større enn ***p***

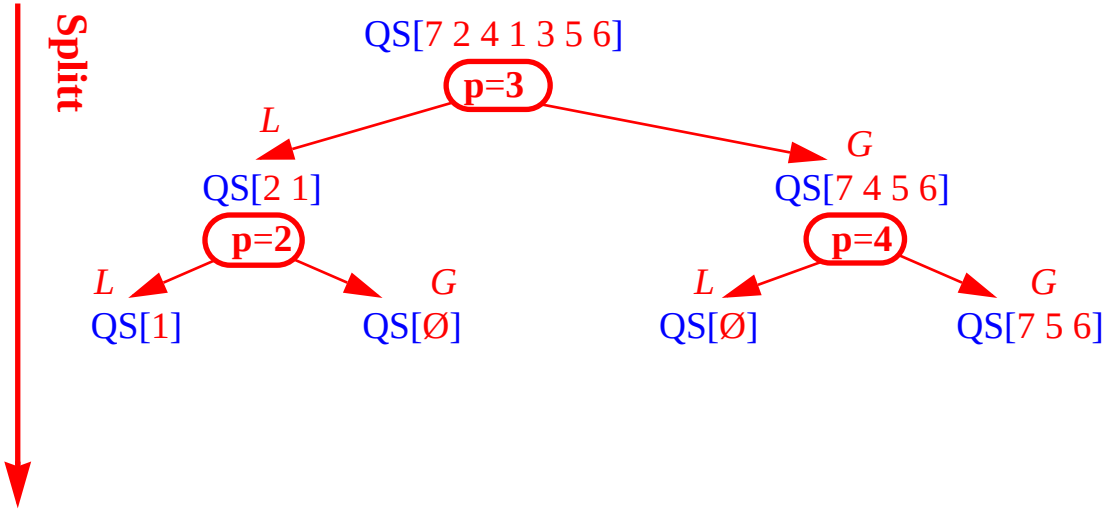
Rekursjon : sorter rekursivt **QS**(*L*) og **QS**(*G*)



Quick-Sort

Splitt : hvis *S* har minst 2 elementer
– velg et element ***p*** fra *S* (pivot) og
– del *S* i tre mengder *L*, *E* og *G* slik at :
L inneholder alle elementene mindre enn ***p***
E inneholder alle elementene lik ***p***
G inneholder alle elementene større enn ***p***

Rekursjon : sorter rekursivt **QS**(*L*) og **QS**(*G*)

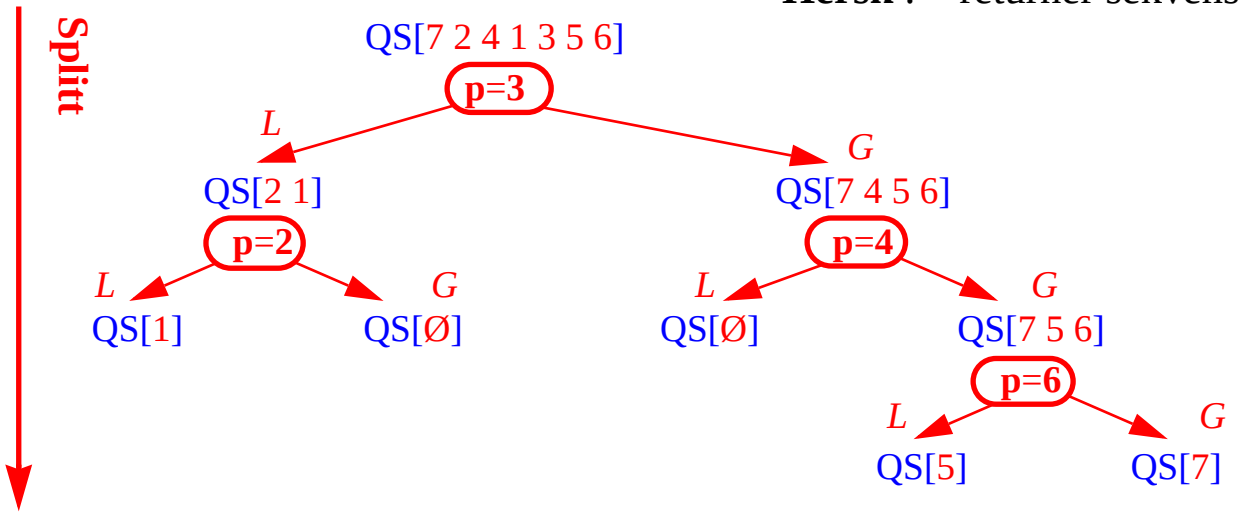


Quick-Sort

Splitt : hvis *S* har minst 2 elementer
– velg et element ***p*** fra *S* (pivot) og
– del *S* i tre mengder *L*, *E* og *G* slik at :

Rekursjon :

Hersk : returner sekvensen *QS(L)–E–QS(G)*

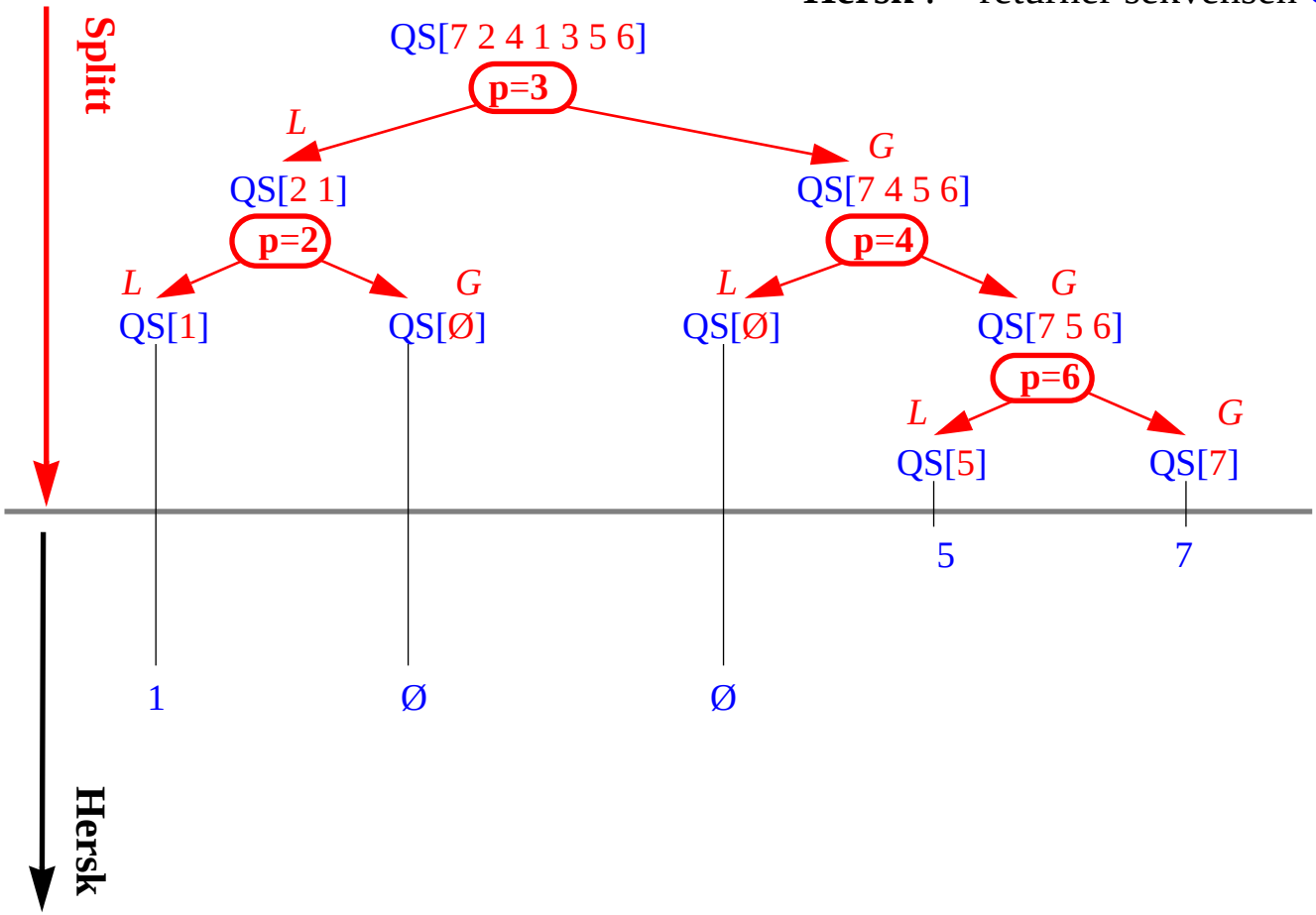


Quick-Sort

Splitt : hvis *S* har minst 2 elementer
– velg et element ***p*** fra *S* (pivot) og
– del *S* i tre mengder *L*, *E* og *G* slik at :

Rekursjon :

Hersk : returner sekvensen *QS(L)–E–QS(G)*

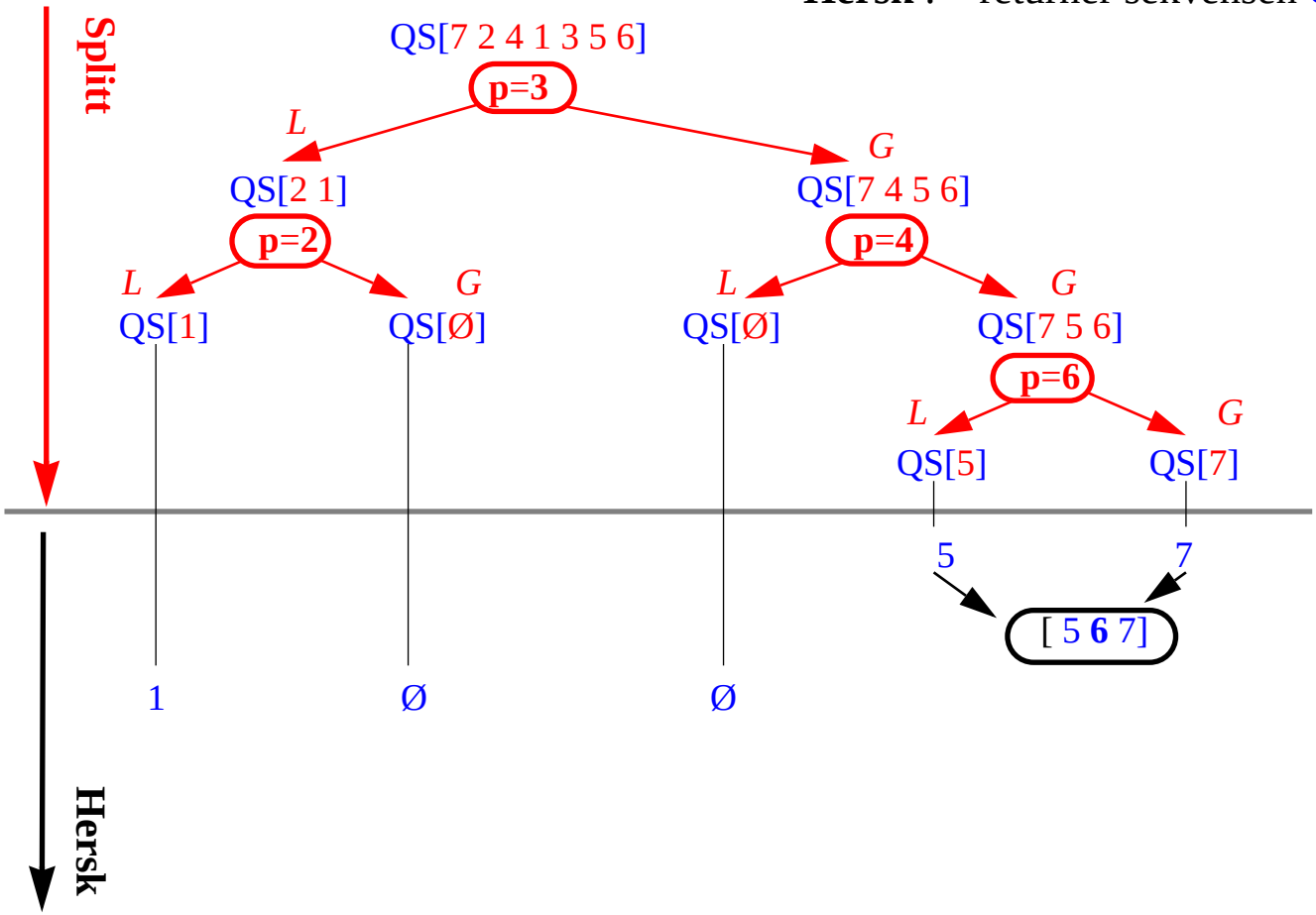


Quick-Sort

Splitt : hvis *S* har minst 2 elementer
– velg et element ***p*** fra *S* (pivot) og
– del *S* i tre mengder *L*, *E* og *G* slik at :
L inneholder alle elementene mindre enn ***p***
E inneholder alle elementene lik ***p***
G inneholder alle elementene større enn ***p***

Rekursjon : sorter rekursivt **QS**(*L*) og **QS**(*G*)

Hersk : returner sekvensen *QS(L)*–*E*–*QS(G)*

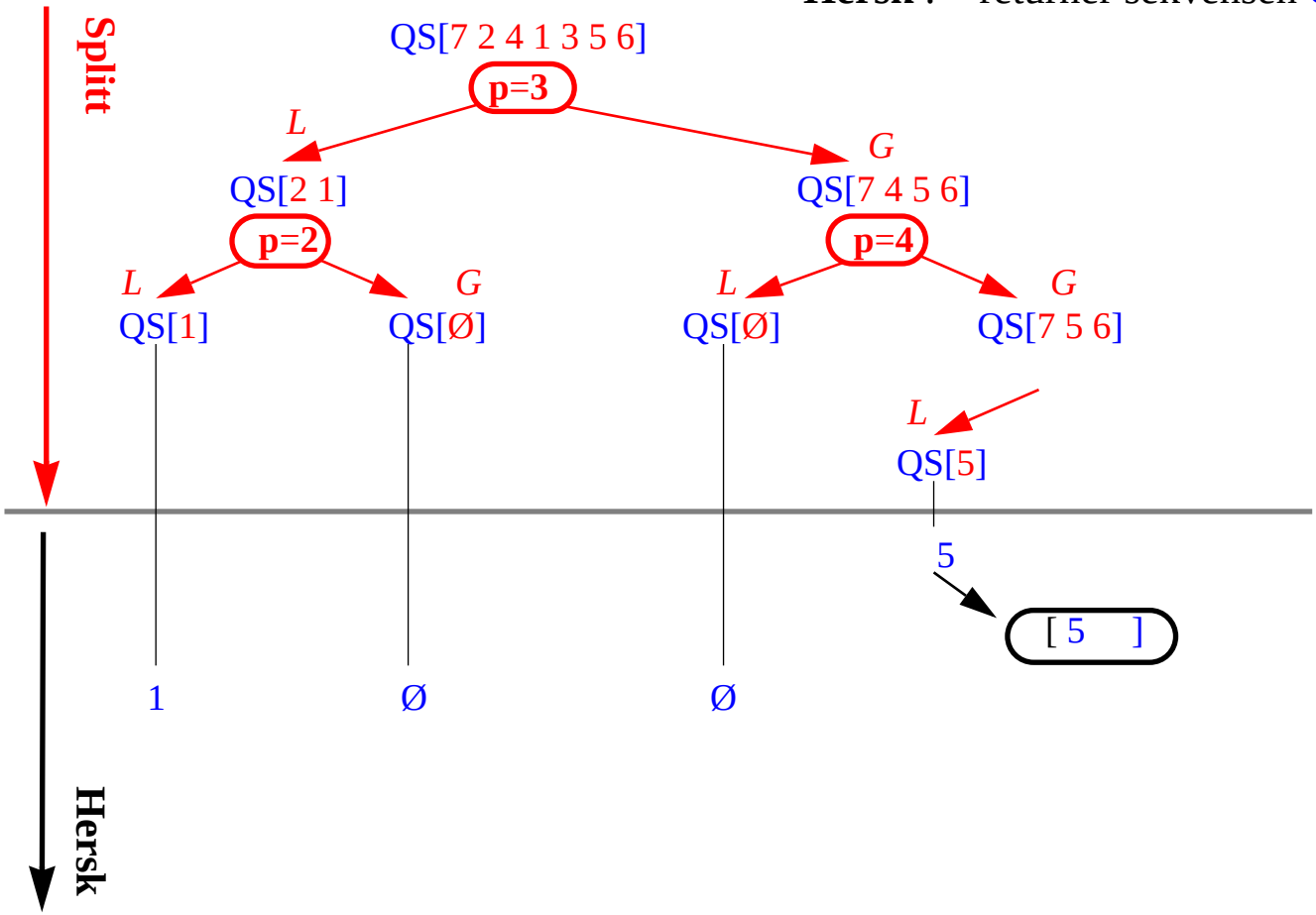


Quick-Sort

Splitt : hvis *S* har minst 2 elementer
– velg et element ***p*** fra *S* (pivot) og
– del *S* i tre mengder *L*, *E* og *G* slik at :
L inneholder alle elementene mindre enn ***p***
E inneholder alle elementene lik ***p***
G inneholder alle elementene større enn ***p***

Rekursjon : sorter rekursivt **QS**(*L*) og **QS**(*G*)

Hersk : returner sekvensen *QS(L)*–

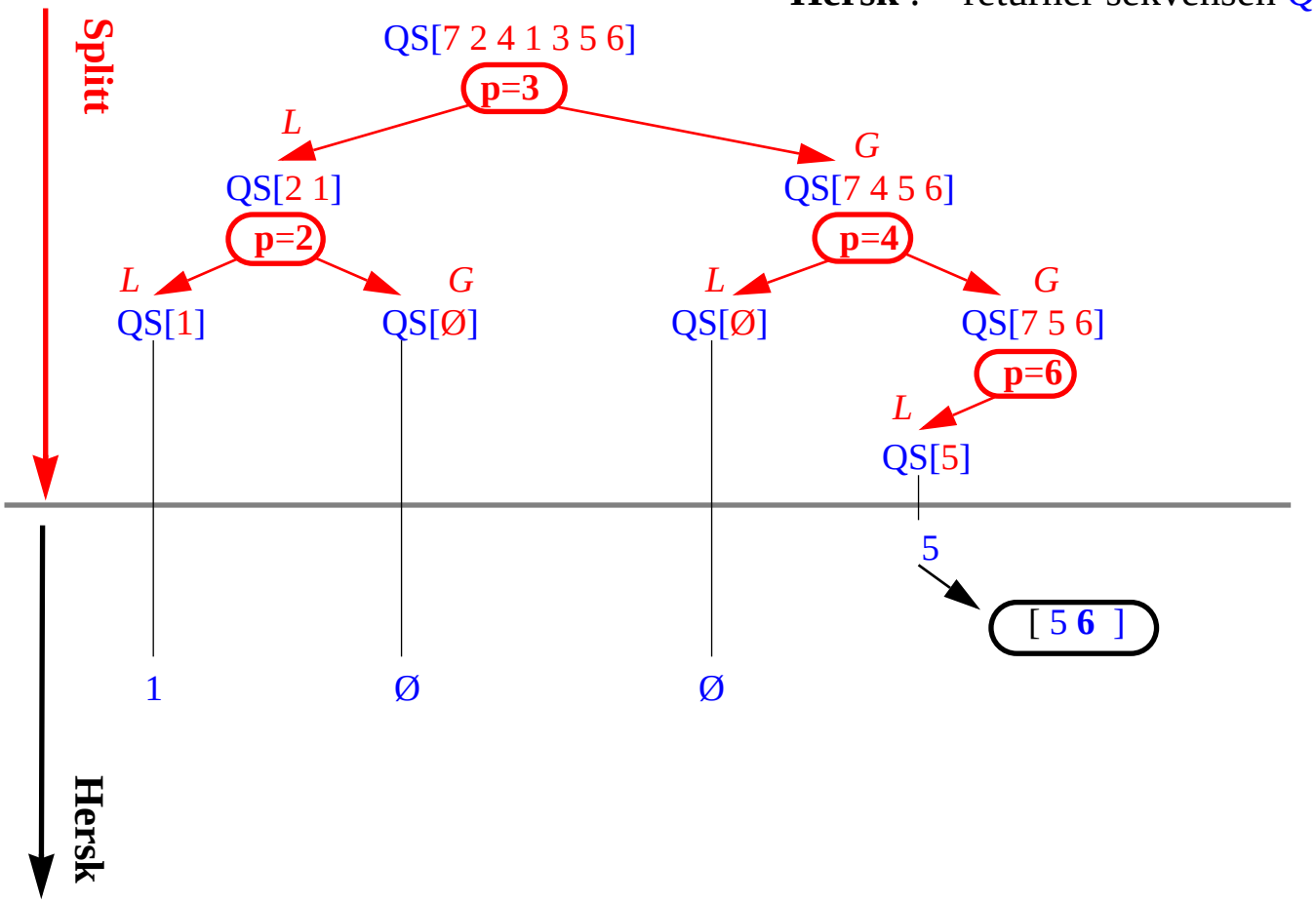


Quick-Sort

Splitt : hvis *S* har minst 2 elementer
– velg et element ***p*** fra *S* (pivot) og
– del *S* i tre mengder *L*, *E* og *G* slik at :
L inneholder alle elementene mindre enn ***p***
E inneholder alle elementene lik ***p***
G inneholder alle elementene større enn ***p***

Rekursjon : sorter rekursivt **QS**(*L*) og **QS**(*G*)

Hersk : returner sekvensen *QS(L)*–*E*–

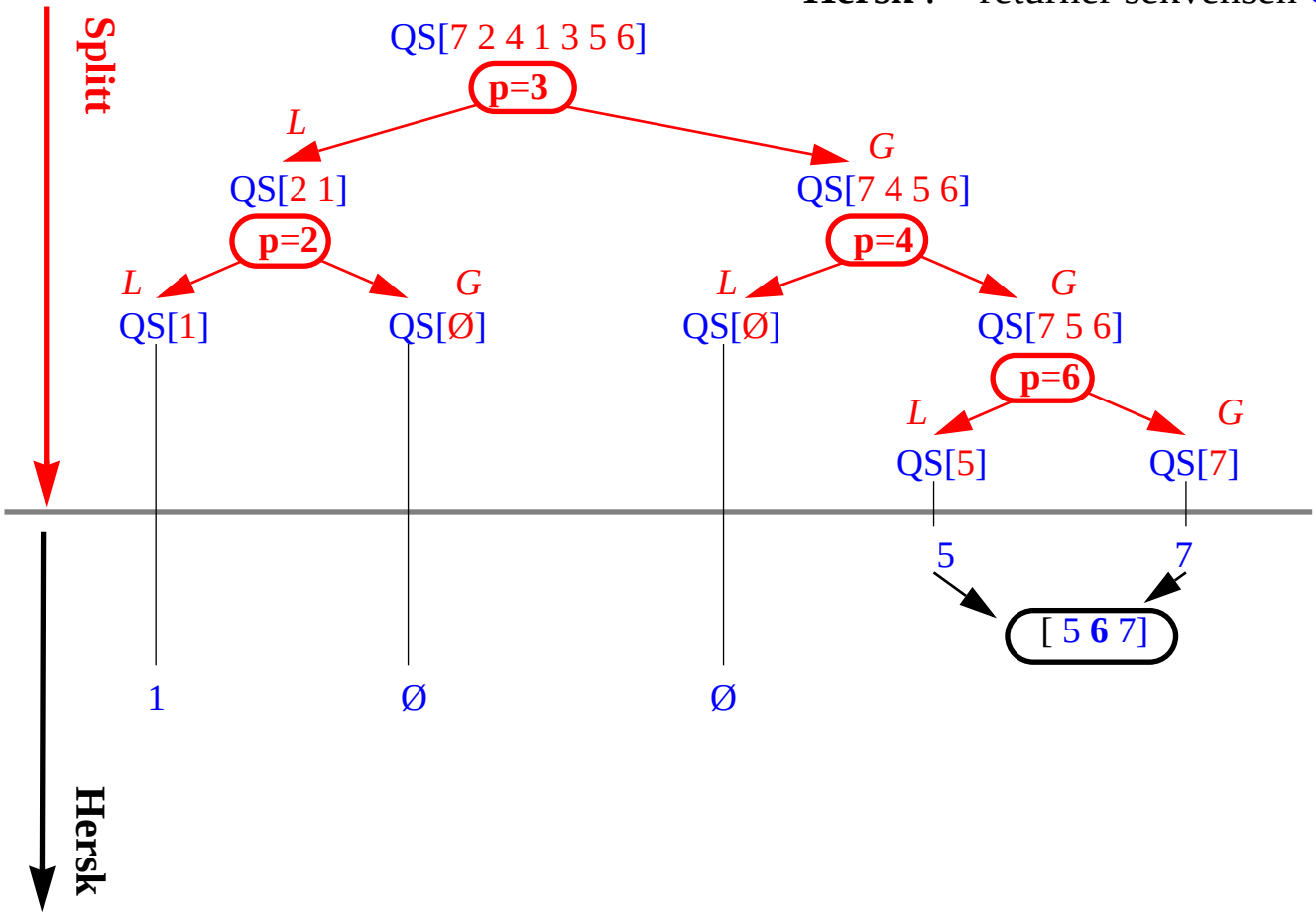


Quick-Sort

Splitt : hvis *S* har minst 2 elementer
– velg et element ***p*** fra *S* (pivot) og
– del *S* i tre mengder *L*, *E* og *G* slik at :
L inneholder alle elementene mindre enn ***p***
E inneholder alle elementene lik ***p***
G inneholder alle elementene større enn ***p***

Rekursjon : sorter rekursivt **QS**(*L*) og **QS**(*G*)

Hersk : returner sekvensen *QS(L)–E–QS(G)*

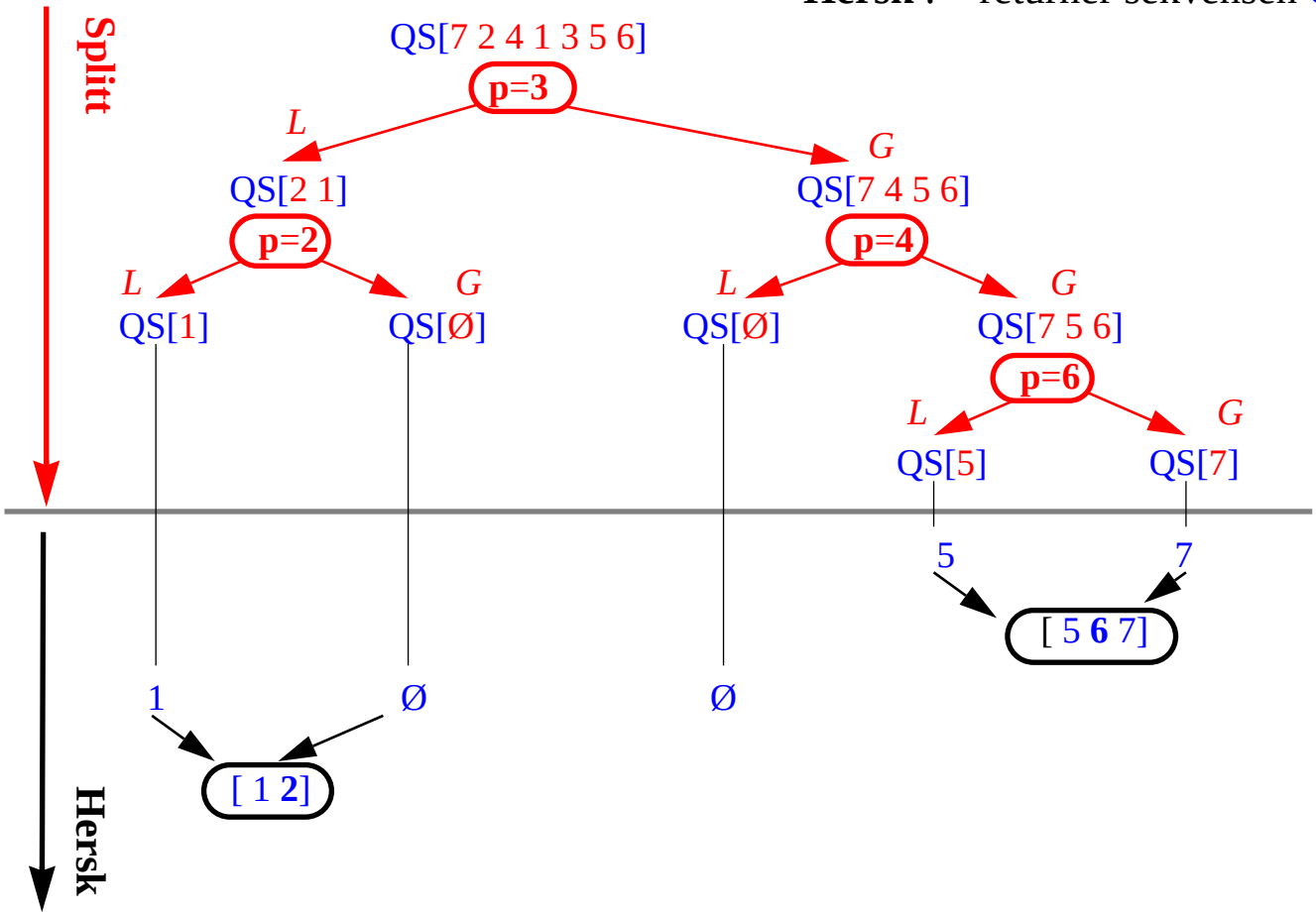


Quick-Sort

Splitt : hvis *S* har minst 2 elementer
– velg et element ***p*** fra *S* (pivot) og
– del *S* i tre mengder *L*, *E* og *G* slik at :
L inneholder alle elementene mindre enn ***p***
E inneholder alle elementene lik ***p***
G inneholder alle elementene større enn ***p***

Rekursjon : sorter rekursivt **QS**(*L*) og **QS**(*G*)

Hersk : returner sekvensen *QS(L)*–*E*–*QS(G)*

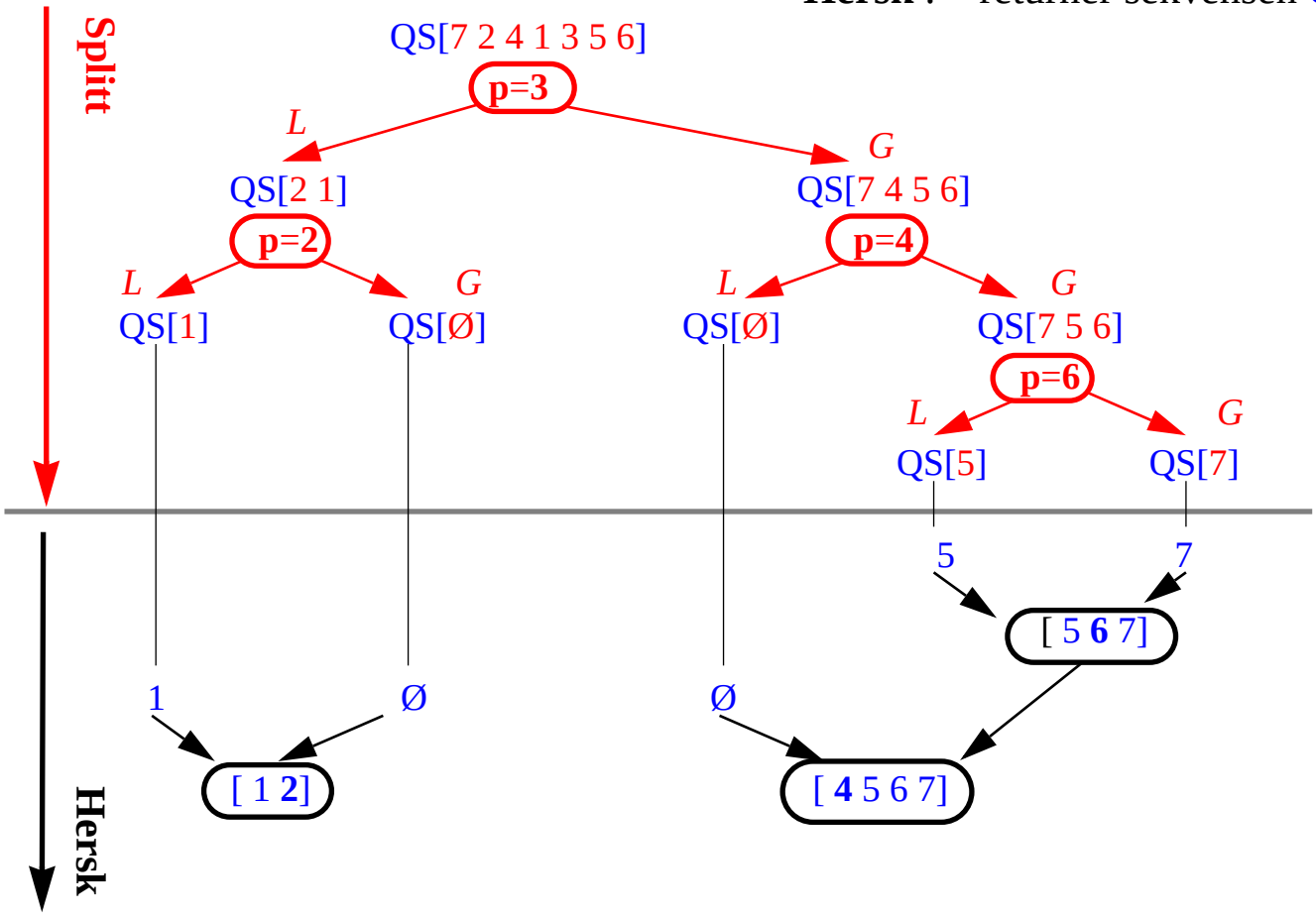


Quick-Sort

Splitt : hvis *S* har minst 2 elementer
– velg et element ***p*** fra *S* (pivot) og
– del *S* i tre mengder *L*, *E* og *G* slik at :
L inneholder alle elementene mindre enn ***p***
E inneholder alle elementene lik ***p***
G inneholder alle elementene større enn ***p***

Rekursjon : sorter rekursivt **QS**(*L*) og **QS**(*G*)

Hersk : returner sekvensen *QS(L)*–*E*–*QS(G)*

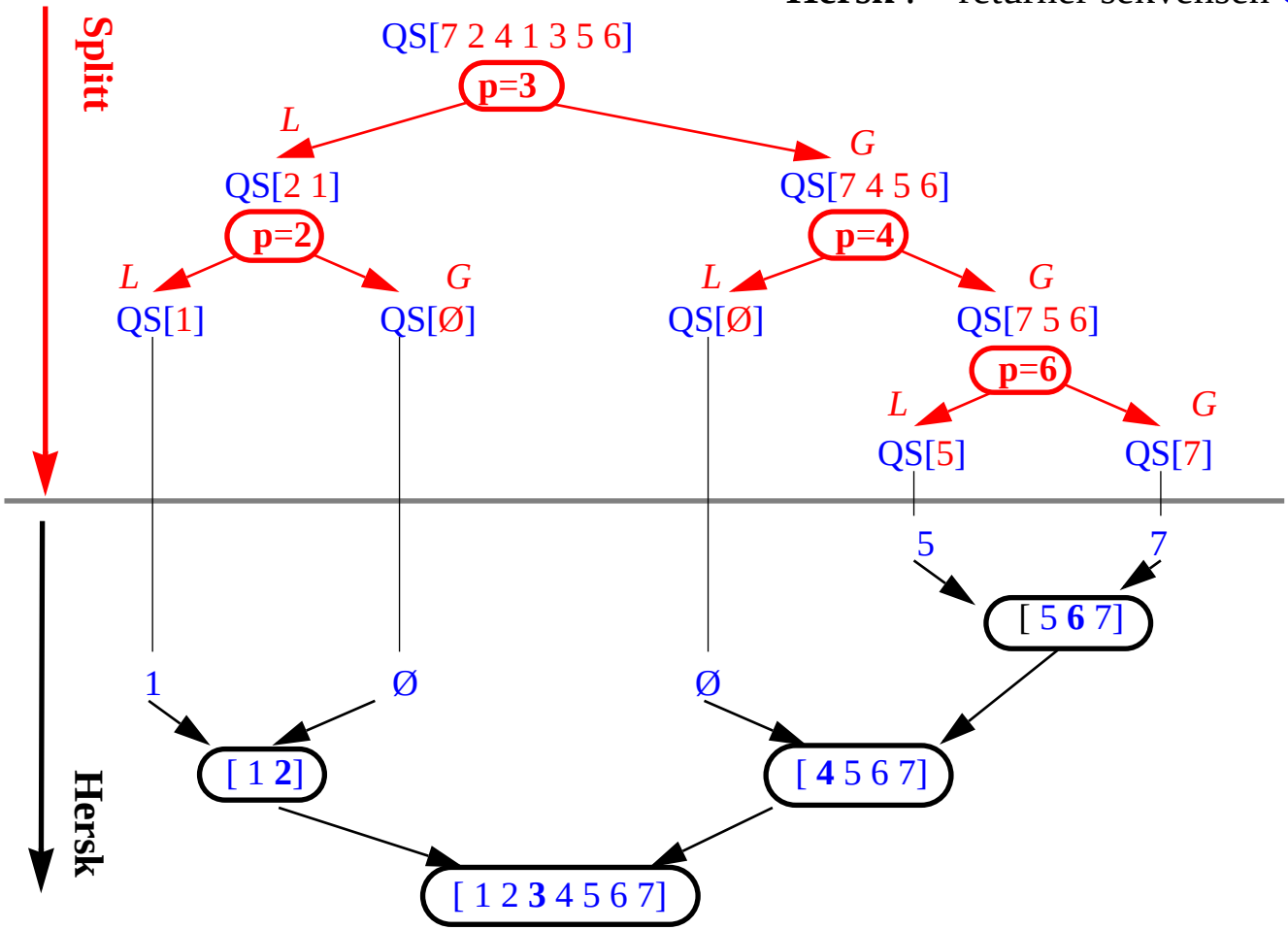


Quick-Sort

Splitt : hvis *S* har minst 2 elementer
– velg et element ***p*** fra *S* (pivot) og
– del *S* i tre mengder *L*, *E* og *G* slik at :
L inneholder alle elementene mindre enn ***p***
E inneholder alle elementene lik ***p***
G inneholder alle elementene større enn ***p***

Rekursjon : sorter rekursivt **QS**(*L*) og **QS**(*G*)

Hersk : returner sekvensen *QS(L)*–*E*–*QS(G)*

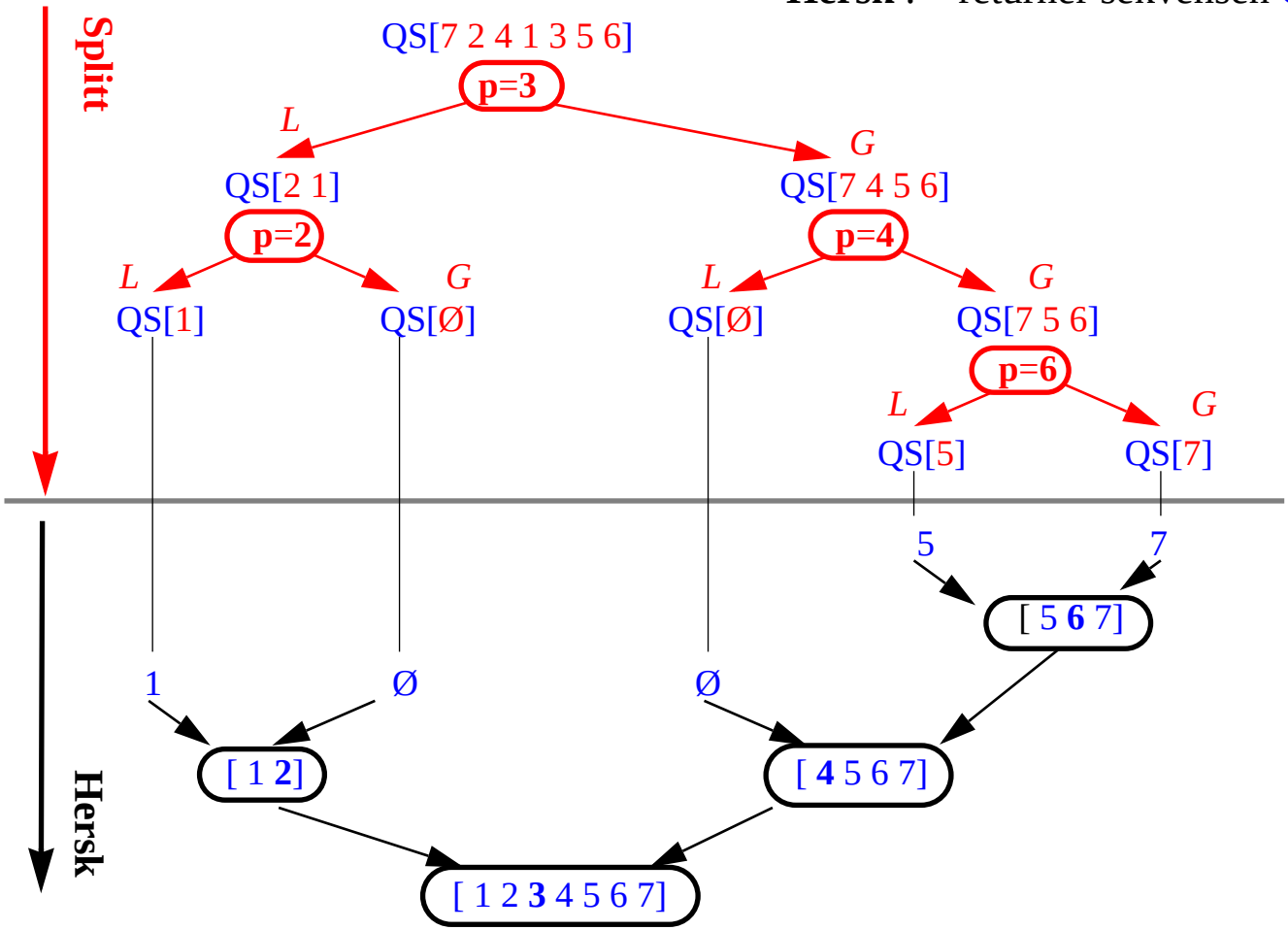


Quick-Sort

Splitt : hvis *S* har minst 2 elementer
– velg et element ***p*** fra *S* (pivot) og
– del *S* i tre mengder *L*, *E* og *G* slik at :
L inneholder alle elementene mindre enn ***p***
E inneholder alle elementene lik ***p***
G inneholder alle elementene større enn ***p***

Rekursjon : sorter rekursivt **QS**(*L*) og **QS**(*G*)

Hersk : returner sekvensen *QS(L)*–*E*–*QS(G)*



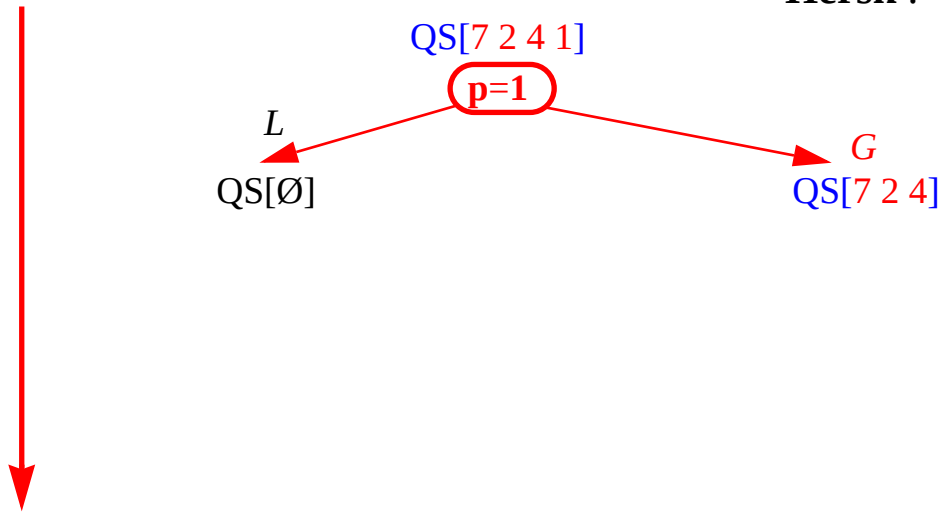
O (n * log(n)) ... ?

Quick-Sort

Splitt : hvis *S* har minst 2 elementer
– velg et element ***p*** fra *S* (pivot) og
– del *S* i tre mengder *L*, *E* og *G* slik at :
L inneholder alle elementene mindre enn ***p***
E inneholder alle elementene lik ***p***
G inneholder alle elementene større enn ***p***

Rekursjon : sorter rekursivt **QS**(*L*) og **QS**(*G*)

Hersk : returner sekvensen *QS(L)*–*E*–*QS(G)*

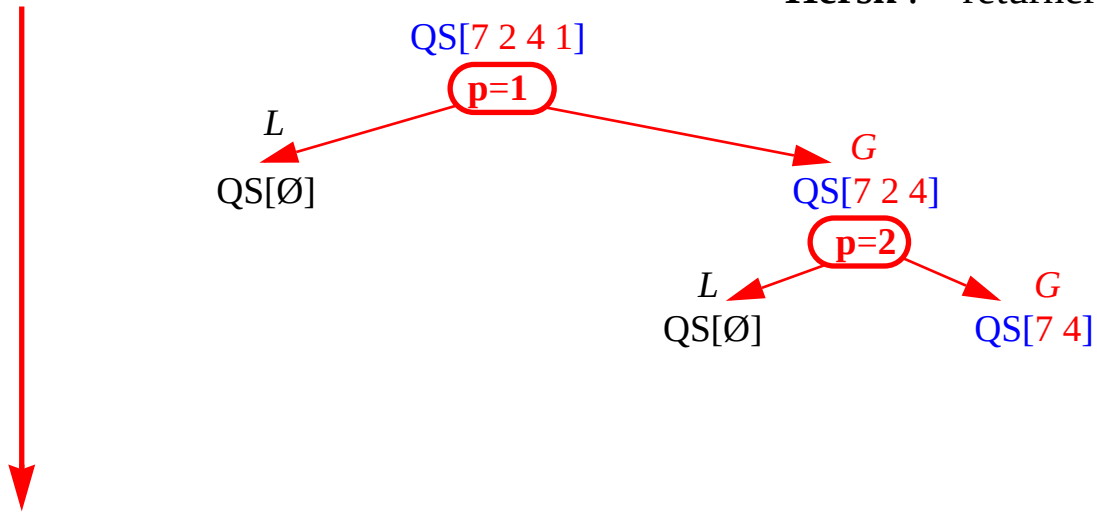


Quick-Sort

Splitt : hvis *S* har minst 2 elementer
– velg et element ***p*** fra *S* (pivot) og
– del *S* i tre mengder *L*, *E* og *G* slik at :
L inneholder alle elementene mindre enn ***p***
E inneholder alle elementene lik ***p***
G inneholder alle elementene større enn ***p***

Rekursjon : sorter rekursivt **QS**(*L*) og **QS**(*G*)

Hersk : returner sekvensen *QS(L)*–*E*–*QS(G)*

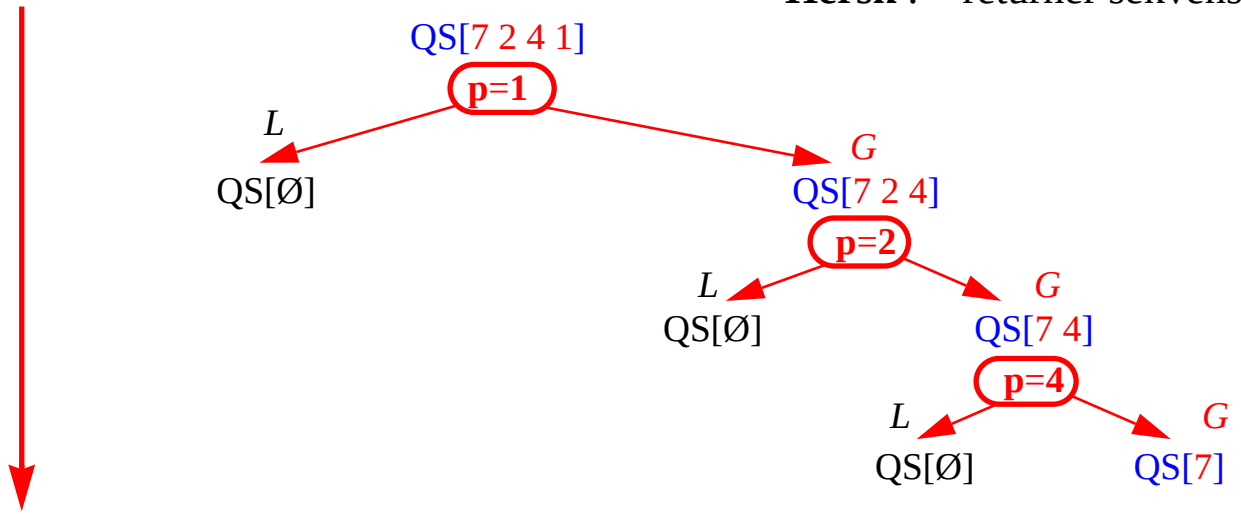


Quick-Sort

Splitt : hvis *S* har minst 2 elementer
– velg et element ***p*** fra *S* (pivot) og
– del *S* i tre mengder *L*, *E* og *G* slik at :
L inneholder alle elementene mindre enn ***p***
E inneholder alle elementene lik ***p***
G inneholder alle elementene større enn ***p***

Rekursjon : sorter rekursivt **QS**(*L*) og **QS**(*G*)

Hersk : returner sekvensen *QS(L)*–*E*–*QS(G)*

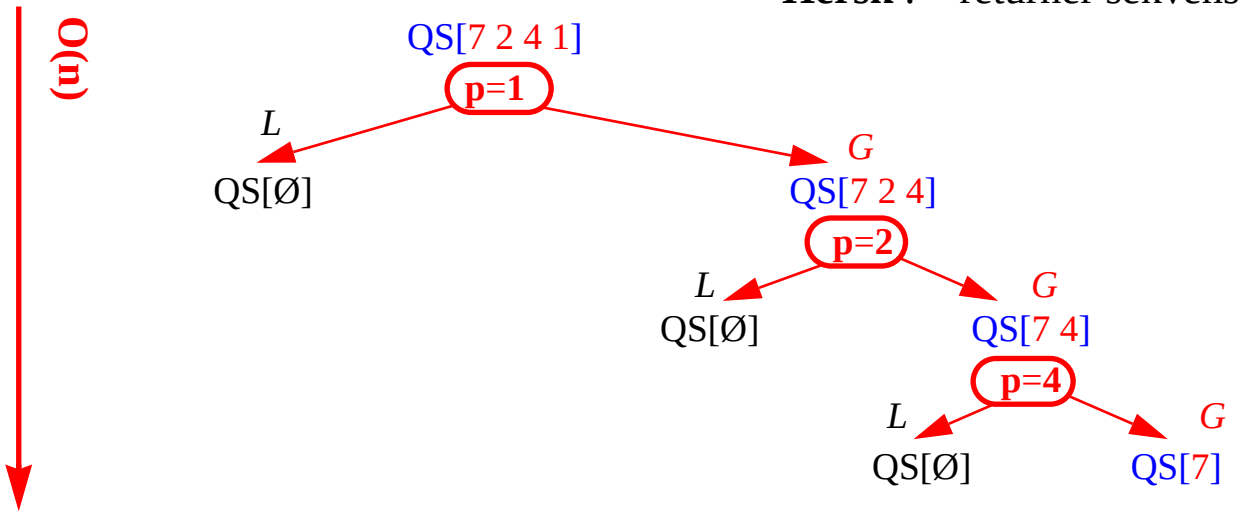


Quick-Sort

Splitt : hvis *S* har minst 2 elementer
– velg et element ***p*** fra *S* (pivot) og
– del *S* i tre mengder *L*, *E* og *G* slik at :
L inneholder alle elementene mindre enn ***p***
E inneholder alle elementene lik ***p***
G inneholder alle elementene større enn ***p***

Rekursjon : sorter rekursivt **QS**(*L*) og **QS**(*G*)

Hersk : returner sekvensen *QS(L)*–*E*–*QS(G)*

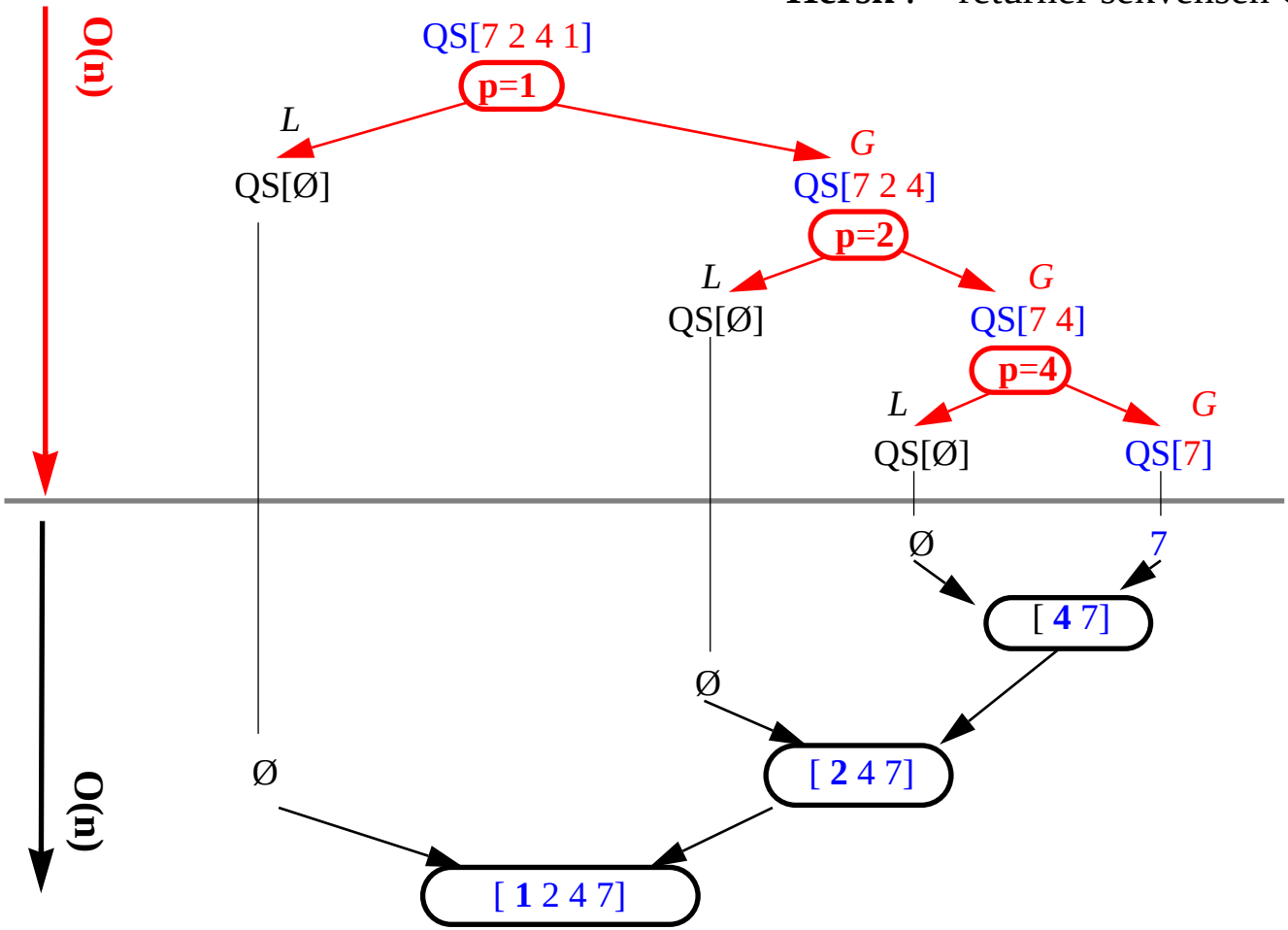


Quick-Sort

Splitt : hvis *S* har minst 2 elementer
– velg et element ***p*** fra *S* (pivot) og
– del *S* i tre mengder *L*, *E* og *G* slik at :
L inneholder alle elementene mindre enn ***p***
E inneholder alle elementene lik ***p***
G inneholder alle elementene større enn ***p***

Rekursjon : sorter rekursivt **QS**(*L*) og **QS**(*G*)

Hersk : returner sekvensen *QS(L)–E–QS(G)*

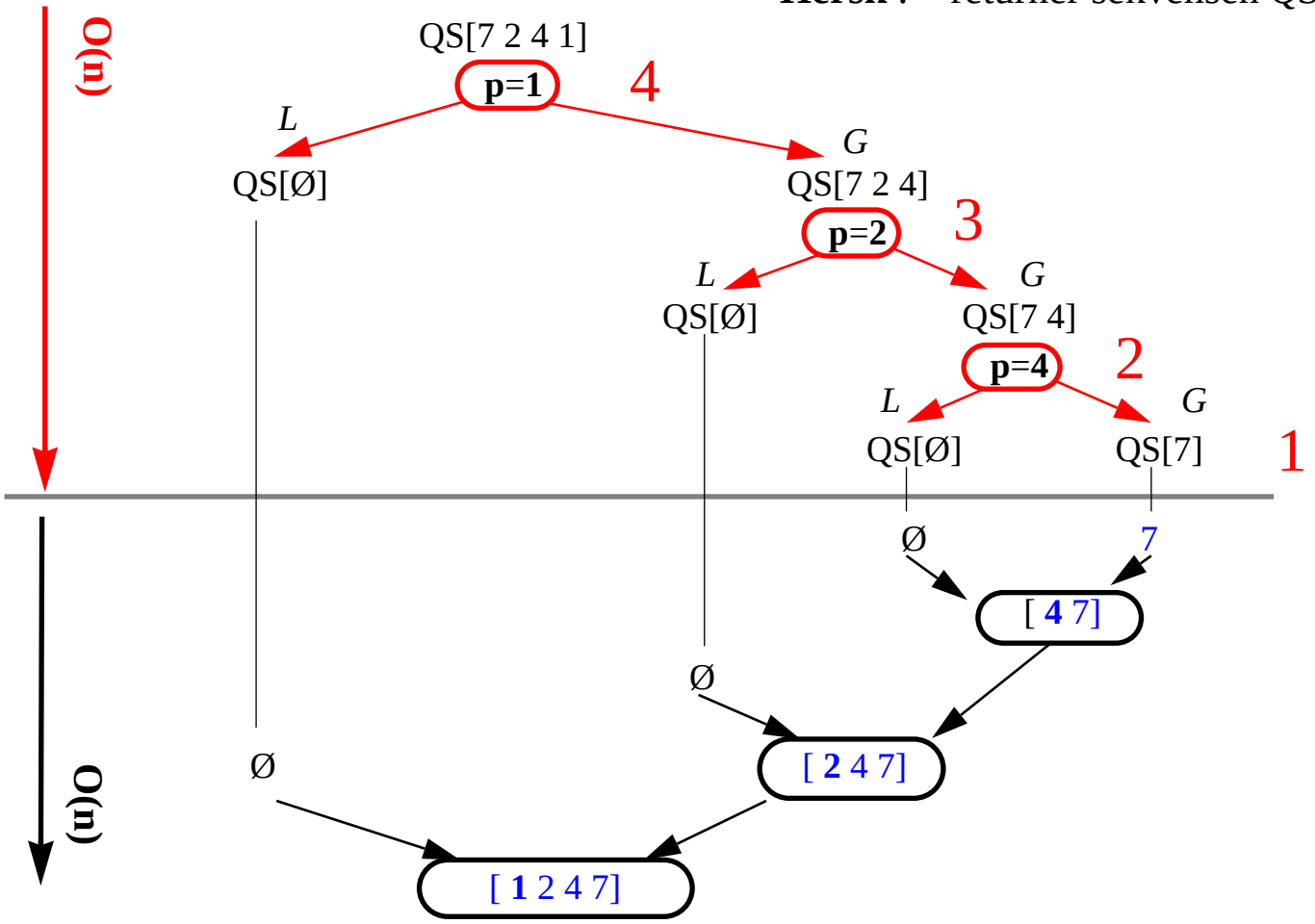


Quick-Sort

Splitt : hvis S har minst 2 elementer
– velg et element ***p*** fra S (pivot) og O(1)
– del S i tre mengder L, E og G slik at : O(|S|)
L inneholder alle elementene mindre enn ***p***
E inneholder alle elementene lik ***p***
G inneholder alle elementene større enn ***p***

Rekursjon : sorter rekursivt **QS**(L) og **QS**(G)

Hersk : returner sekvensen *QS(L)–E–QS(G)*

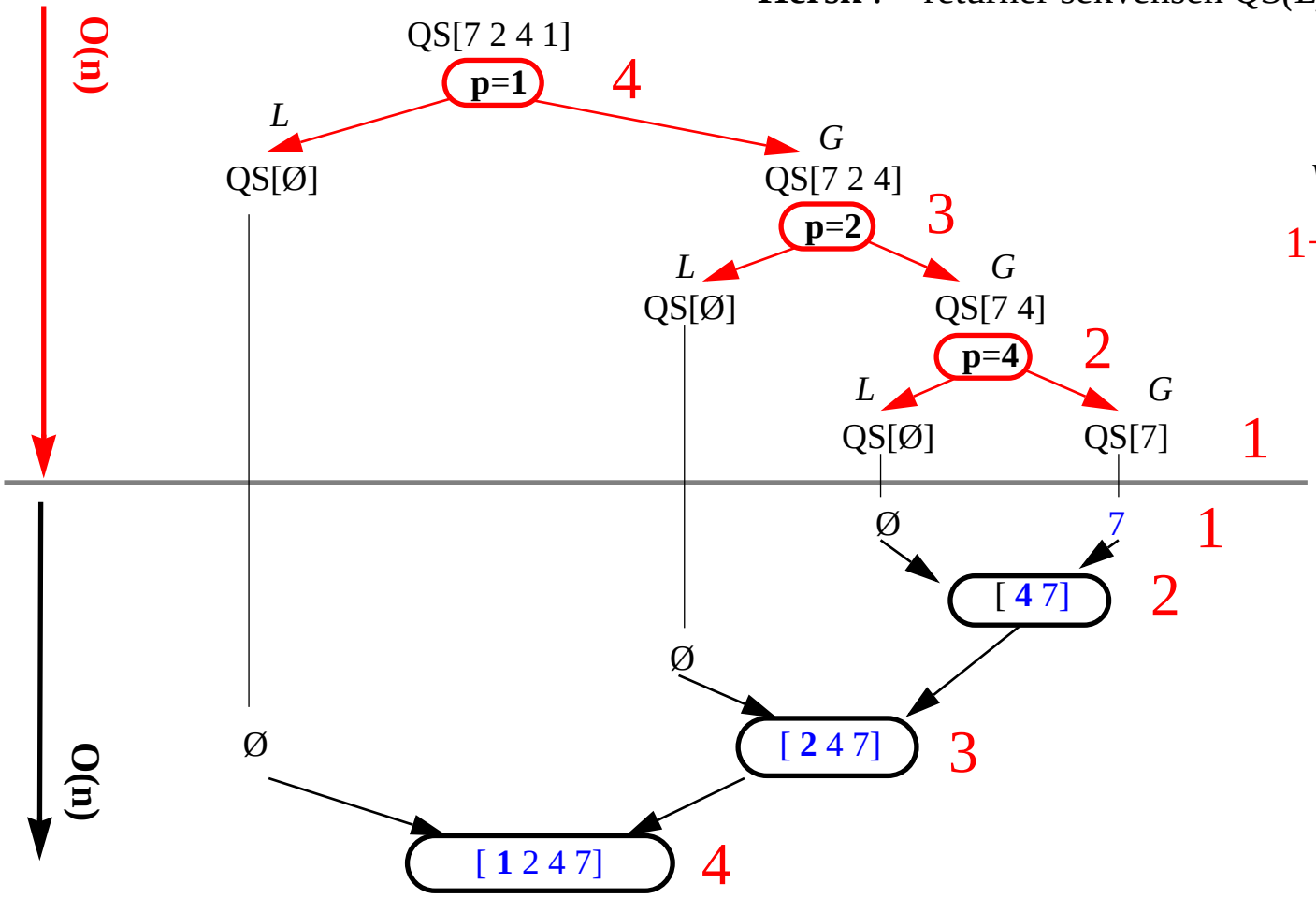


Quick-Sort

Splitt : hvis S har minst 2 elementer
– velg et element ***p*** fra S (pivot) og O(1)
– del S i tre mengder L, E og G slik at : O(|S|)
L inneholder alle elementene mindre enn ***p***
E inneholder alle elementene lik ***p***
G inneholder alle elementene større enn ***p***

Rekursjon : sorter rekursivt **QS**(L) og **QS**(G)

Hersk : returner sekvensen *QS(L)–E–QS(G)*



verste tilfelle kompleksitet er

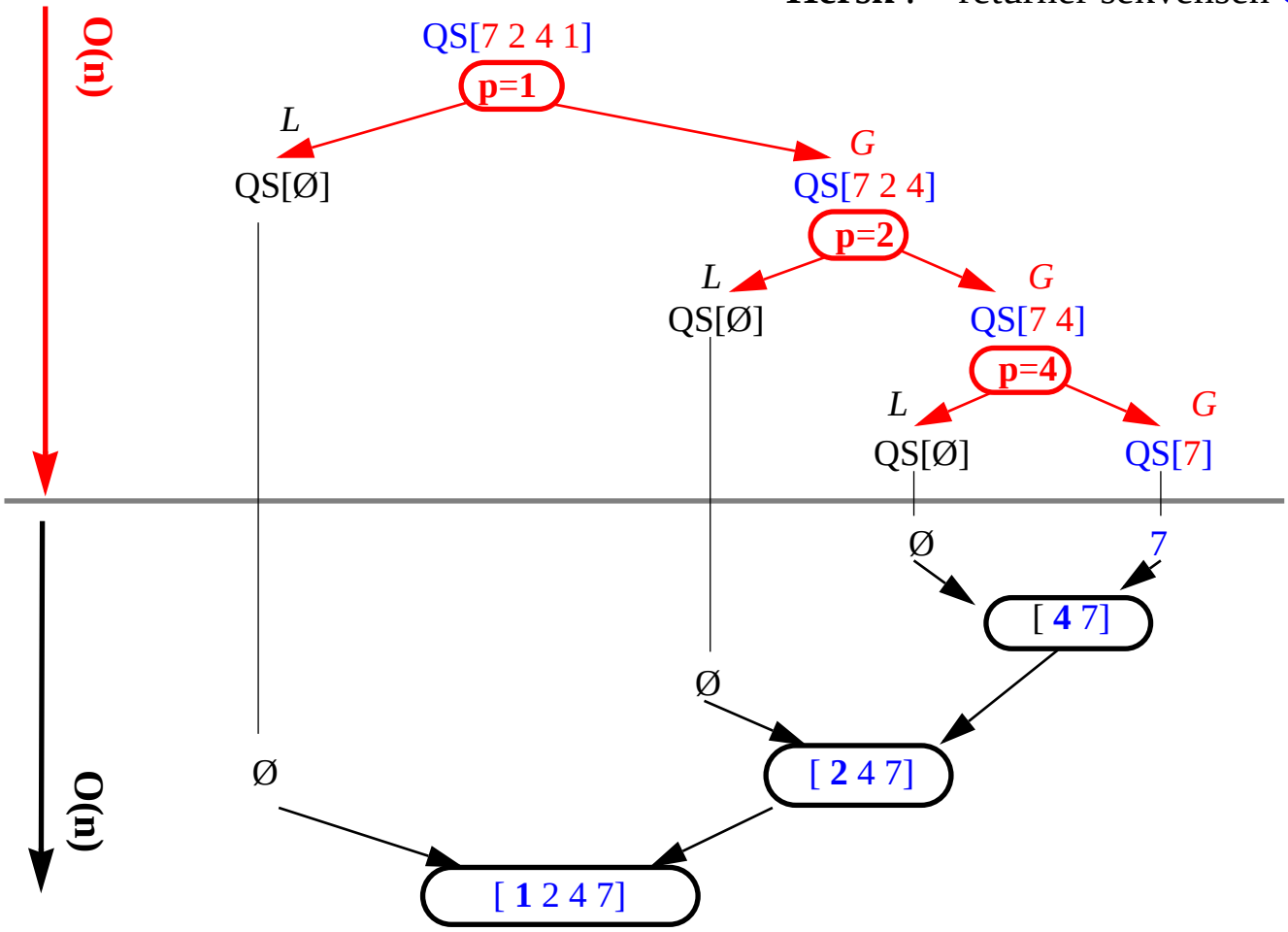
$1+2+...+n = O(n^2)$

Quick-Sort

Splitt : hvis *S* har minst 2 elementer
– velg et element ***p*** fra *S* (pivot) og
– del *S* i tre mengder *L*, *E* og *G* slik at :
L inneholder alle elementene mindre enn ***p***
E inneholder alle elementene lik ***p***
G inneholder alle elementene større enn ***p***

Rekursjon : sorter rekursivt **QS**(*L*) og **QS**(*G*)

Hersk : returner sekvensen *QS(L)–E–QS(G)*



verste tilfelle kompleksitet er

$1+2+...+n = O(n^2)$

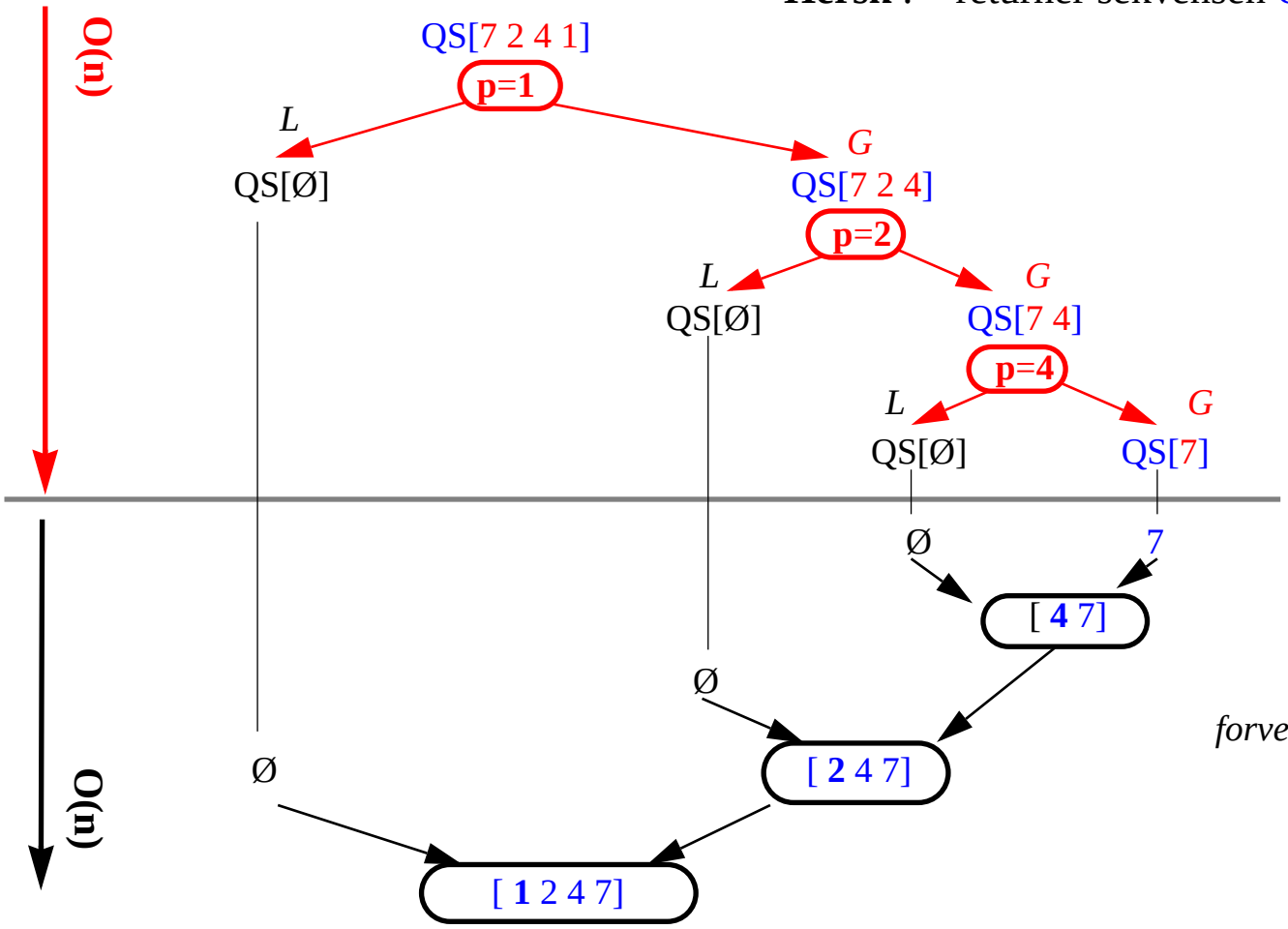
valg av pivot :
– siste i S
– midterste i S
– midterste bland {første, midterste, siste} i S
– random

Quick-Sort

Splitt : hvis *S* har minst 2 elementer
– velg et element ***p*** fra *S* (pivot) og
– del *S* i tre mengder *L*, *E* og *G* slik at :
L inneholder alle elementene mindre enn ***p***
E inneholder alle elementene lik ***p***
G inneholder alle elementene større enn ***p***

Rekursjon : sorter rekursivt **QS**(*L*) og **QS**(*G*)

Hersk : returner sekvensen *QS(L)–E–QS(G)*



verste tilfelle kompleksitet er

$1+2+...+n = O(n^2)$

valg av pivot :
– siste i S
– midterste i S
– midterste bland {første, midterste, siste} i S
– random

forventet/gjennomsnittlig kompleksitet med random valg av pivot er

$O(n * \log(n))$

“In-Place” sortering

En (sortering) algoritme er “in-place” dersom
den trenger kun konstat lagringsplass i tillegg til selve argumentet.

"In-Place" sortering

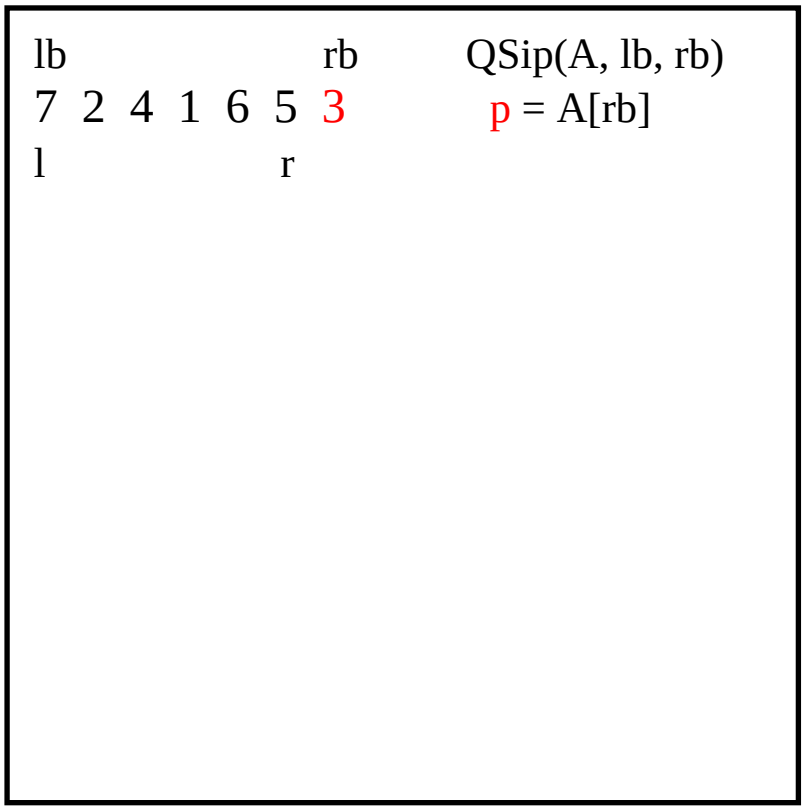
En (sortering) algoritme er "in-place" dersom
den trenger kun konstat lagringsplass i tillegg til selve argumentet.

```
void QSip(int[] A, int lb, int rb) {
   if (rb–lb < 1) return;
   p = A[rb];
   int l = lb, r = rb–1;
   while (l <= r) {
      while (l <= r && A[l] <= p) l++;
      while (r >= l && A[r] >= p) r--;
      if (l < r) A.swap(l, r);
   }
   A.swap(l,rb);
   QSip(A,lb,l–1);
   QSip(A,l+1;rb);
}
```

"In-Place" sortering

En (sortering) algoritme er "in-place" dersom
den trenger kun konstat lagringsplass i tillegg til selve argumentet.

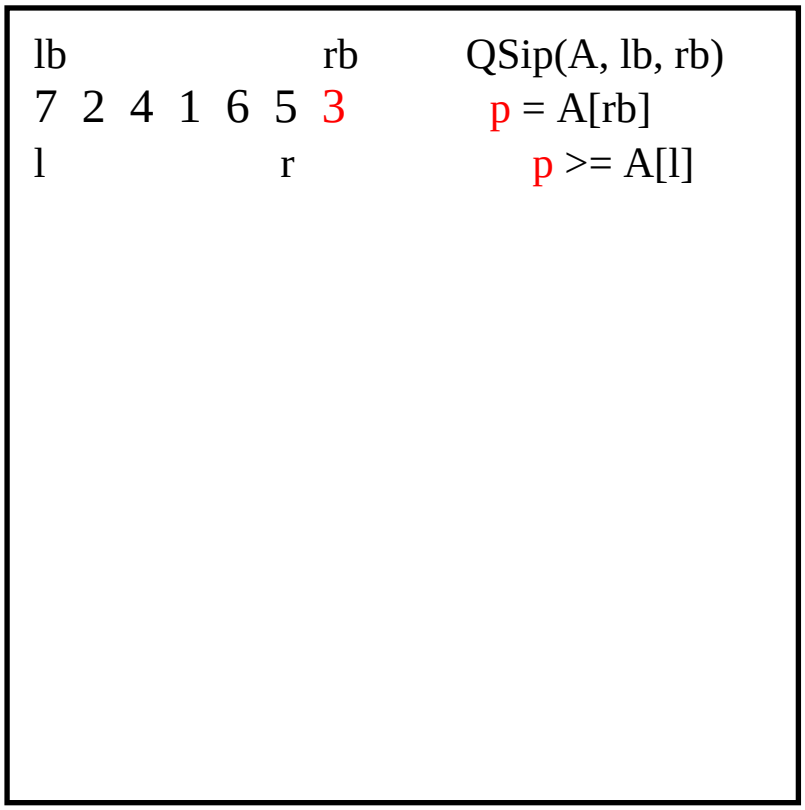
```
void QSip(int[] A, int lb, int rb) {
   if (rb–lb < 1) return;
   p = A[rb];
   int l = lb, r = rb–1;
   while (l <= r) {
      while (l <= r && A[l] <= p) l++;
      while (r >= l && A[r] >= p) r--;
      if (l < r) A.swap(l, r);
   }
   A.swap(l,rb);
   QSip(A,lb,l–1);
   QSip(A,l+1;rb);
}
```



"In-Place" sortering

En (sortering) algoritme er "in-place" dersom
den trenger kun konstat lagringsplass i tillegg til selve argumentet.

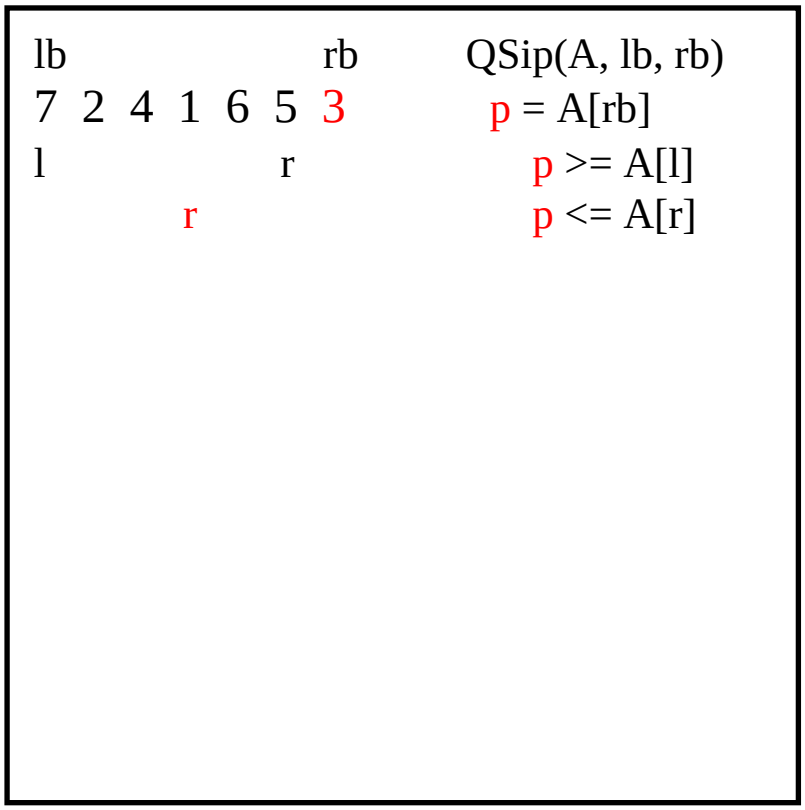
```
void QSip(int[] A, int lb, int rb) {
   if (rb–lb < 1) return;
   p = A[rb];
   int l = lb, r = rb–1;
   while (l <= r) {
      while (l <= r && A[l] <= p) l++;
      while (r >= l && A[r] >= p) r--;
      if (l < r) A.swap(l, r);
   }
   A.swap(l,rb);
   QSip(A,lb,l–1);
   QSip(A,l+1;rb);
}
```



"In-Place" sortering

En (sortering) algoritme er "in-place" dersom
den trenger kun konstat lagringsplass i tillegg til selve argumentet.

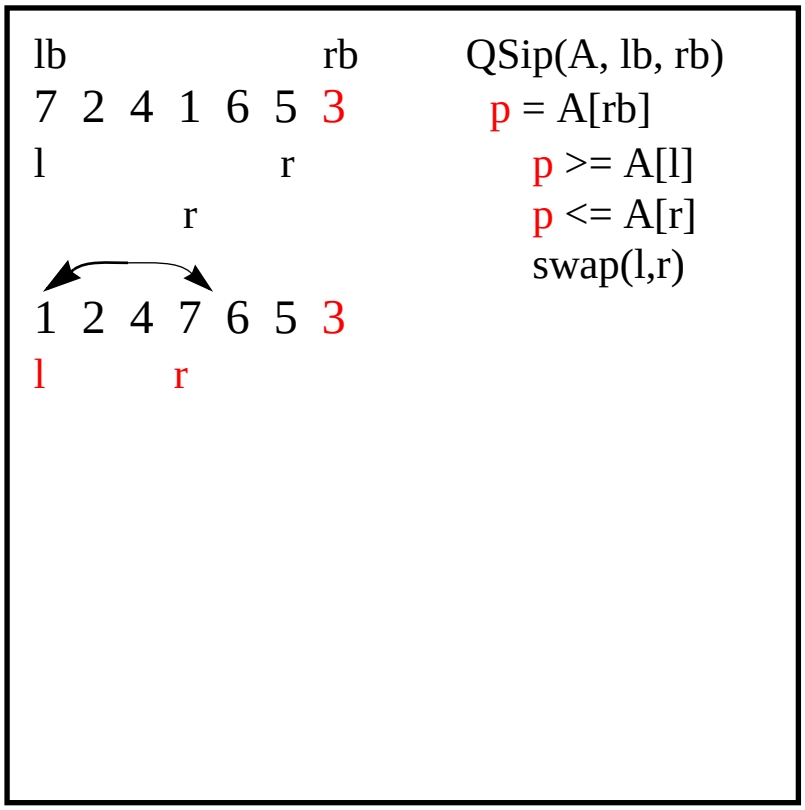
```
void QSip(int[] A, int lb, int rb) {
   if (rb–lb < 1) return;
   p = A[rb];
   int l = lb, r = rb–1;
   while (l <= r) {
      while (l <= r && A[l] <= p) l++;
      while (r >= l && A[r] >= p) r--;
      if (l < r) A.swap(l, r);
   }
   A.swap(l,rb);
   QSip(A,lb,l–1);
   QSip(A,l+1;rb);
}
```



"In-Place" sortering

En (sortering) algoritme er "in-place" dersom
den trenger kun konstat lagringsplass i tillegg til selve argumentet.

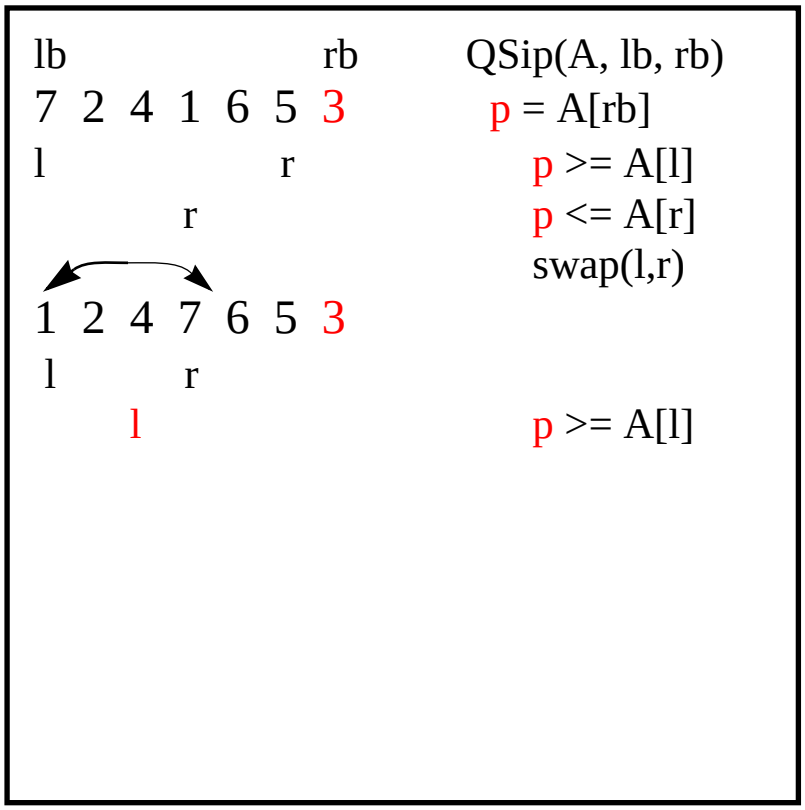
```
void QSip(int[] A, int lb, int rb) {
   if (rb–lb < 1) return;
   p = A[rb];
   int l = lb, r = rb–1;
   while (l <= r) {
      while (l <= r && A[l] <= p) l++;
      while (r >= l && A[r] >= p) r--;
      if (l < r) A.swap(l, r);
   }
   A.swap(l,rb);
   QSip(A,lb,l–1);
   QSip(A,l+1;rb);
}
```



"In-Place" sortering

En (sortering) algoritme er "in-place" dersom
den trenger kun konstat lagringsplass i tillegg til selve argumentet.

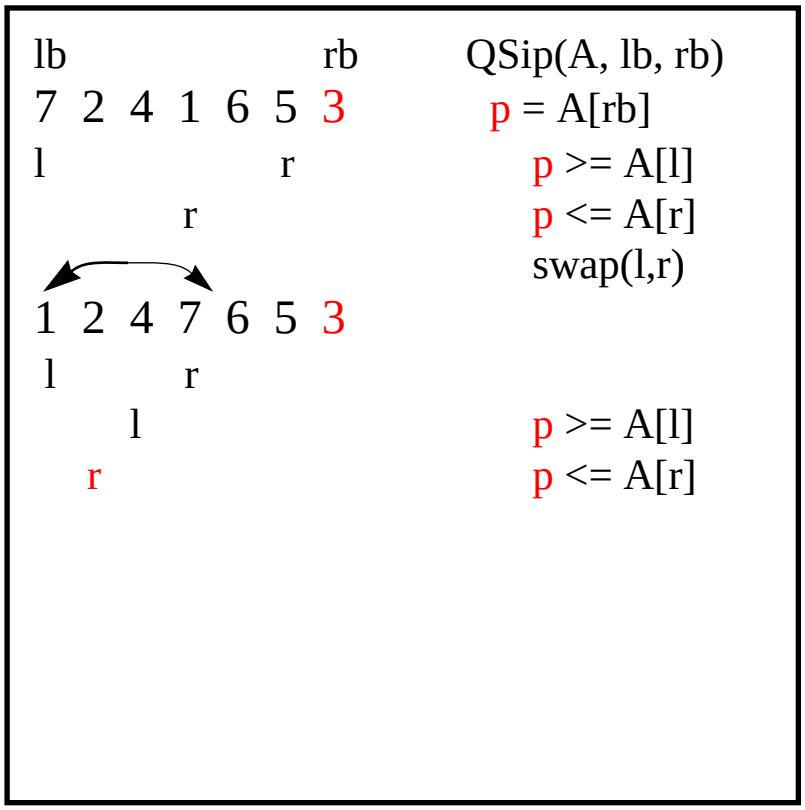
```
void QSip(int[] A, int lb, int rb) {
   if (rb–lb < 1) return;
   p = A[rb];
   int l = lb, r = rb–1;
   while (l <= r) {
      while (l <= r && A[l] <= p) l++;
      while (r >= l && A[r] >= p) r--;
      if (l < r) A.swap(l, r);
   }
   A.swap(l,rb);
   QSip(A,lb,l–1);
   QSip(A,l+1;rb);
}
```



"In-Place" sortering

En (sortering) algoritme er "in-place" dersom
den trenger kun konstat lagringsplass i tillegg til selve argumentet.

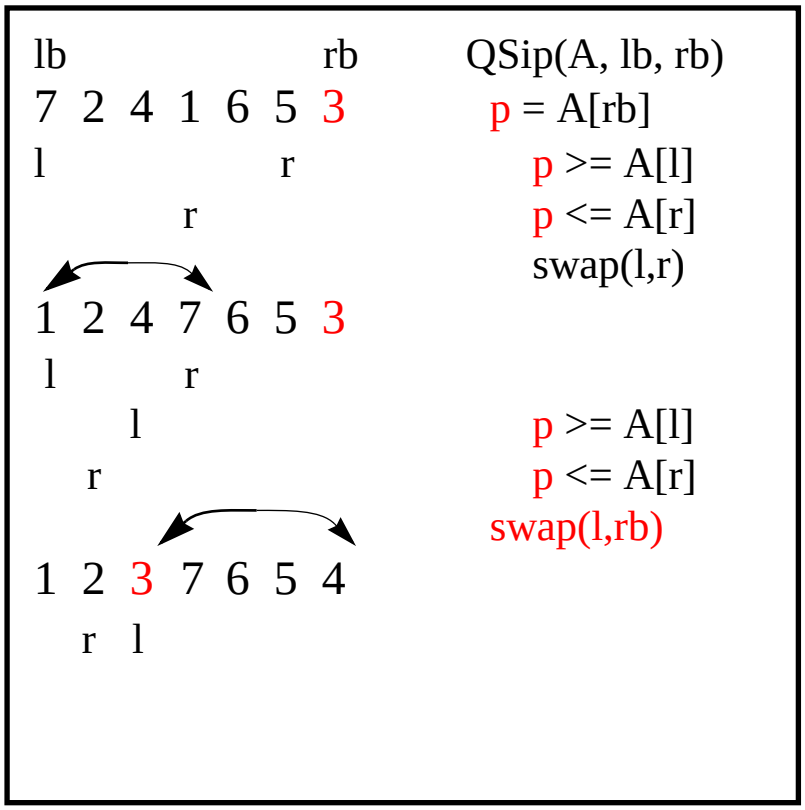
```
void QSip(int[] A, int lb, int rb) {
   if (rb–lb < 1) return;
   p = A[rb];
   int l = lb, r = rb–1;
   while (l <= r) {
      while (l <= r && A[l] <= p) l++;
      while (r >= l && A[r] >= p) r--;
      if (l < r) A.swap(l, r);
   }
   A.swap(l,rb);
   QSip(A,lb,l–1);
   QSip(A,l+1;rb);
}
```



"In-Place" sortering

En (sortering) algoritme er "in-place" dersom
den trenger kun konstat lagringsplass i tillegg til selve argumentet.

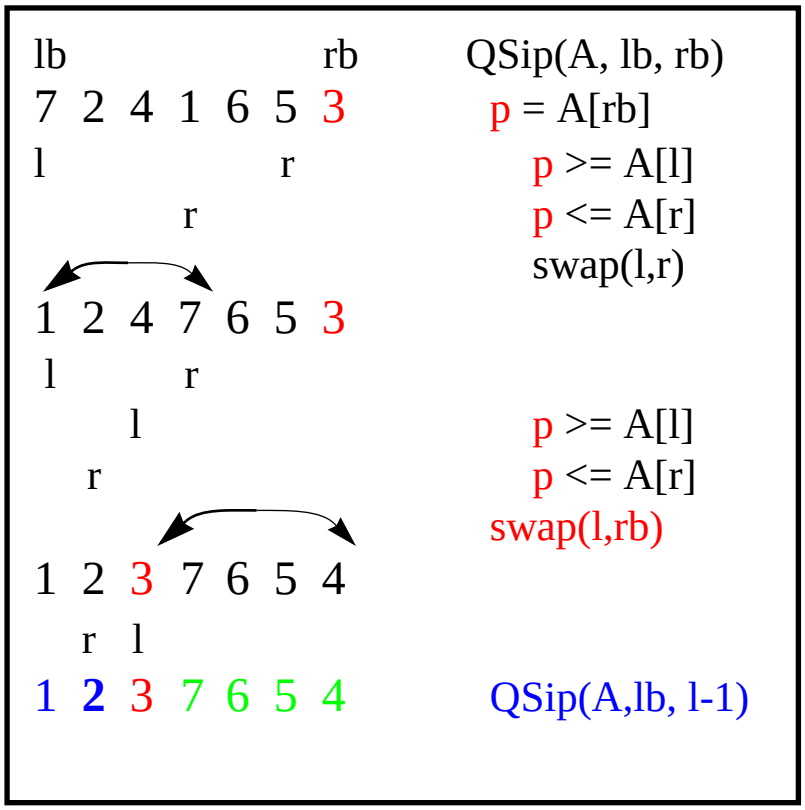
```
void QSip(int[] A, int lb, int rb) {
   if (rb–lb < 1) return;
   p = A[rb];
   int l = lb, r = rb–1;
   while (l <= r) {
      while (l <= r && A[l] <= p) l++;
      while (r >= l && A[r] >= p) r--;
      if (l < r) A.swap(l, r);
   }
   A.swap(l,rb);
   QSip(A,lb,l–1);
   QSip(A,l+1;rb);
}
```



"In-Place" sortering

En (sortering) algoritme er "in-place" dersom
den trenger kun konstat lagringsplass i tillegg til selve argumentet.

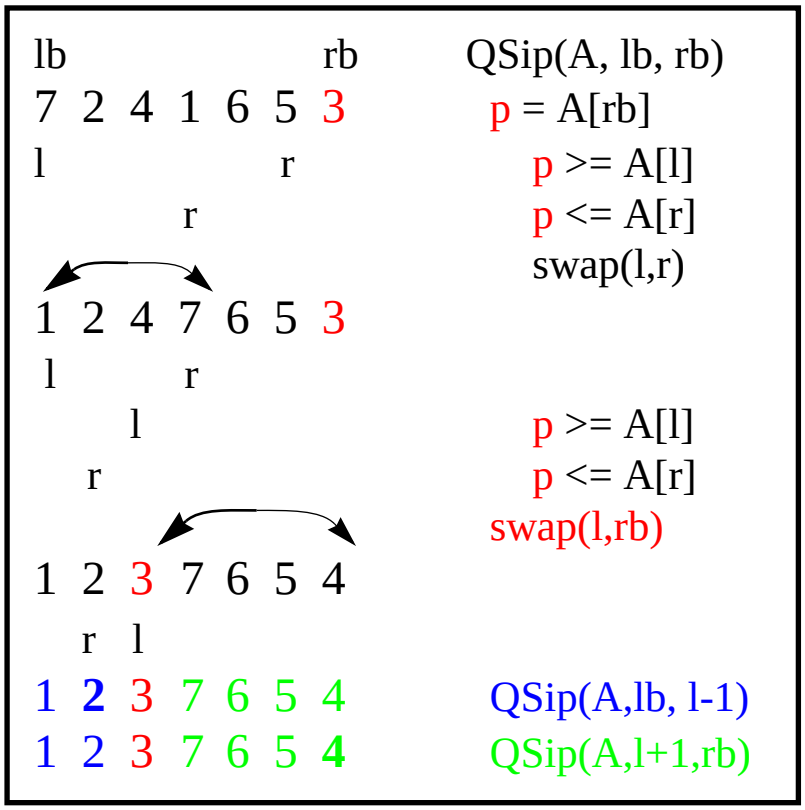
```
void QSip(int[] A, int lb, int rb) {
   if (rb–lb < 1) return;
   p = A[rb];
   int l = lb, r = rb–1;
   while (l <= r) {
      while (l <= r && A[l] <= p) l++;
      while (r >= l && A[r] >= p) r--;
      if (l < r) A.swap(l, r);
   }
   A.swap(l,rb);
   QSip(A,lb,l–1);
   QSip(A,l+1;rb);
}
```



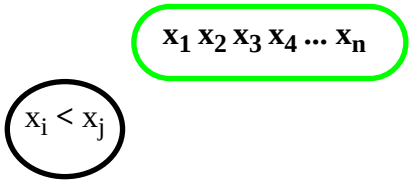
"In-Place" sortering

En (sortering) algoritme er "in-place" dersom
den trenger kun konstat lagringsplass i tillegg til selve argumentet.

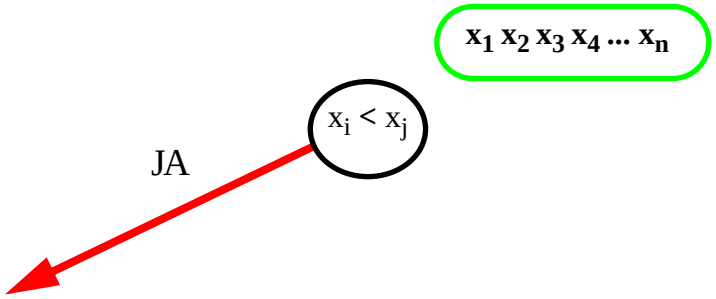
```
void QSip(int[] A, int lb, int rb) {
   if (rb–lb < 1) return;
   p = A[rb];
   int l = lb, r = rb–1;
   while (l <= r) {
      while (l <= r && A[l] <= p) l++;
      while (r >= l && A[r] >= p) r--;
      if (l < r) A.swap(l, r);
   }
   A.swap(l,rb);
   QSip(A,lb,l–1);
   QSip(A,l+1;rb);
}
```



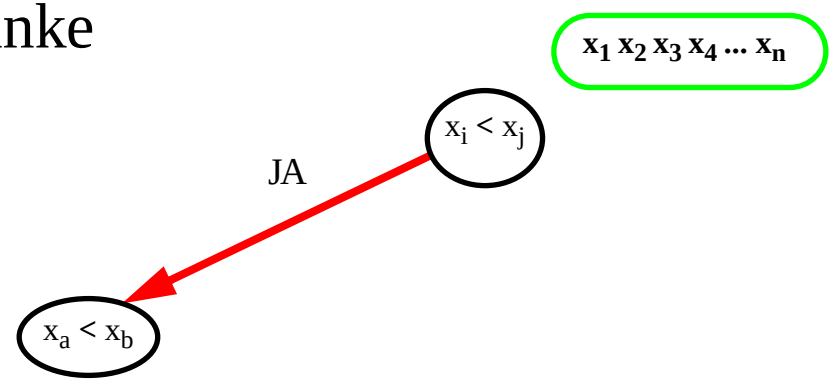
Sammenlikning-sortering: nedre skranke



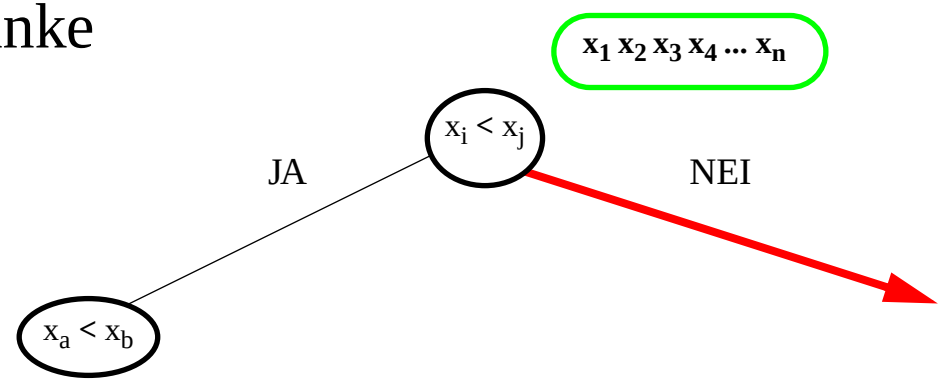
Sammenlikning-sortering: nedre skranke



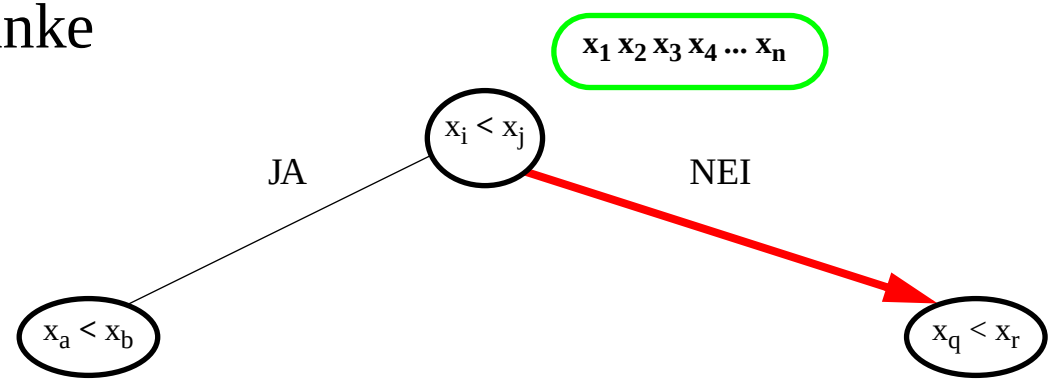
Sammenlikning-sortering: nedre skranke



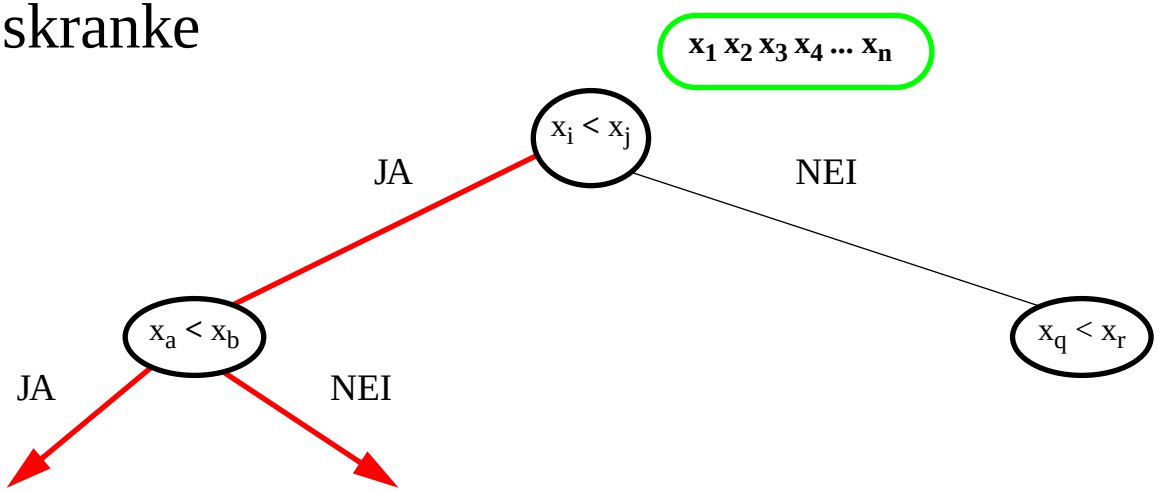
Sammenlikning-sortering: nedre skranke



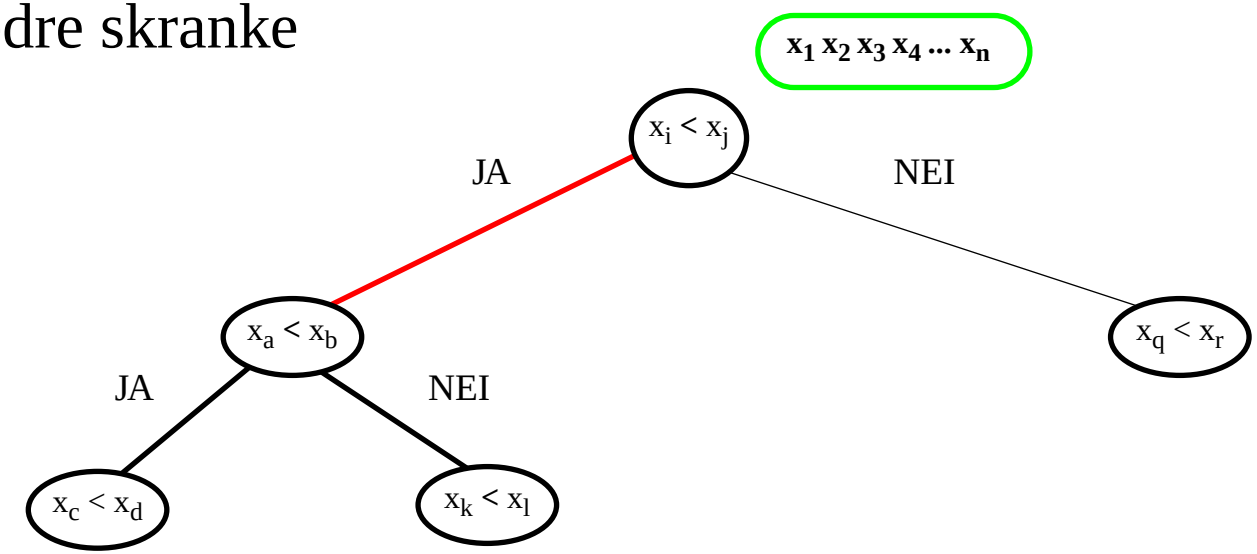
Sammenlikning-sortering: nedre skranke



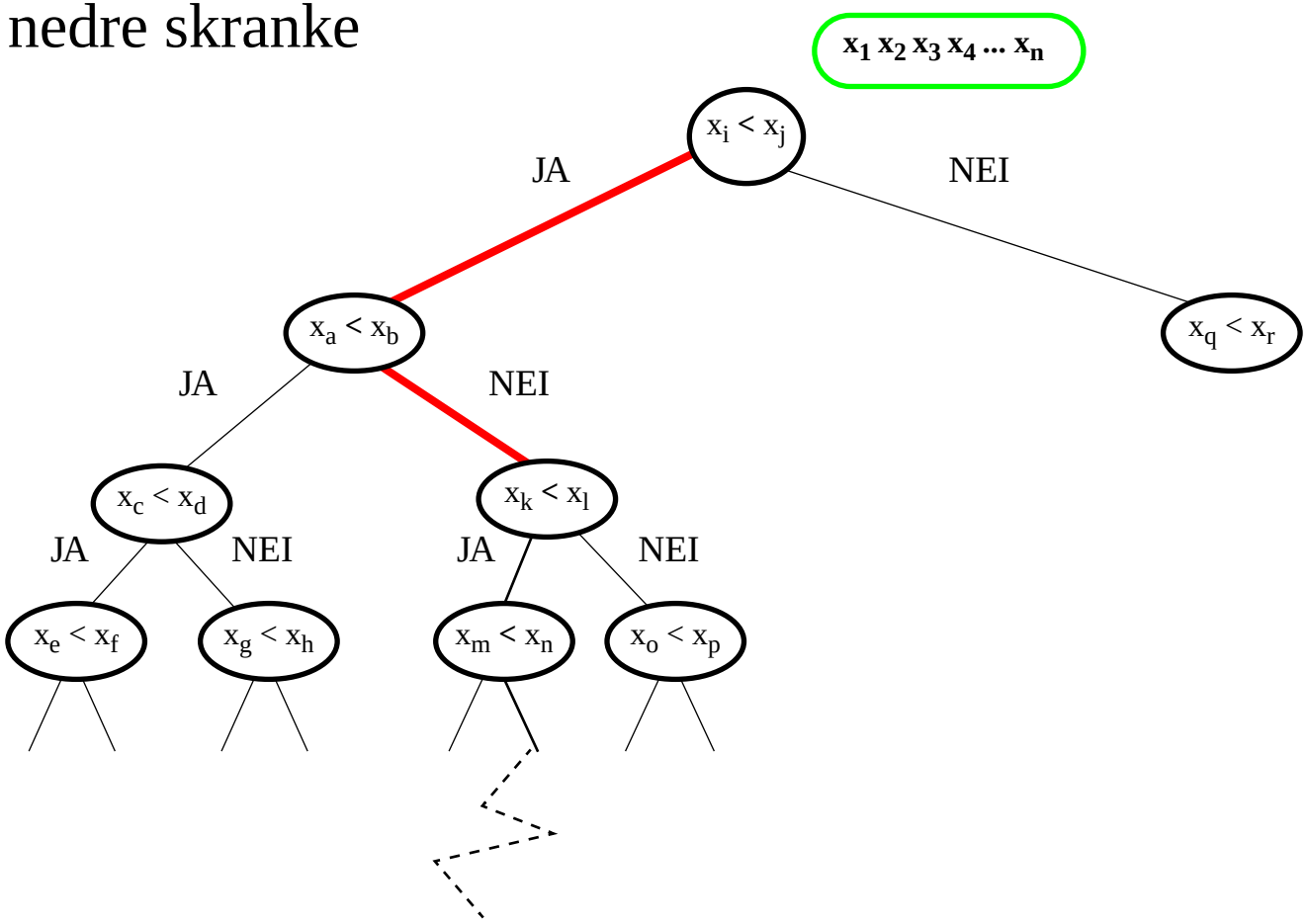
Sammenlikning-sortering: nedre skranke



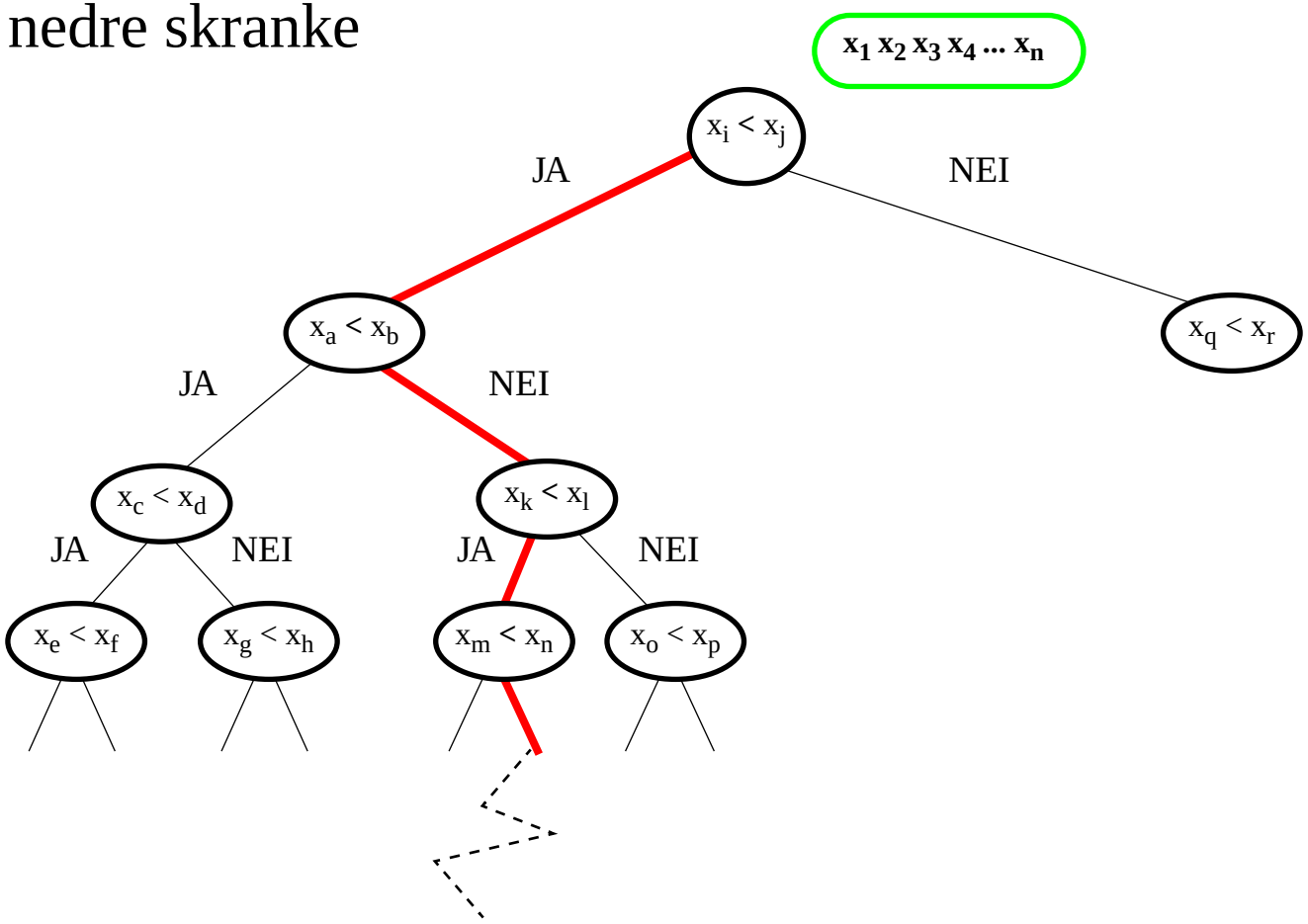
Sammenlikning-sortering: nedre skranke



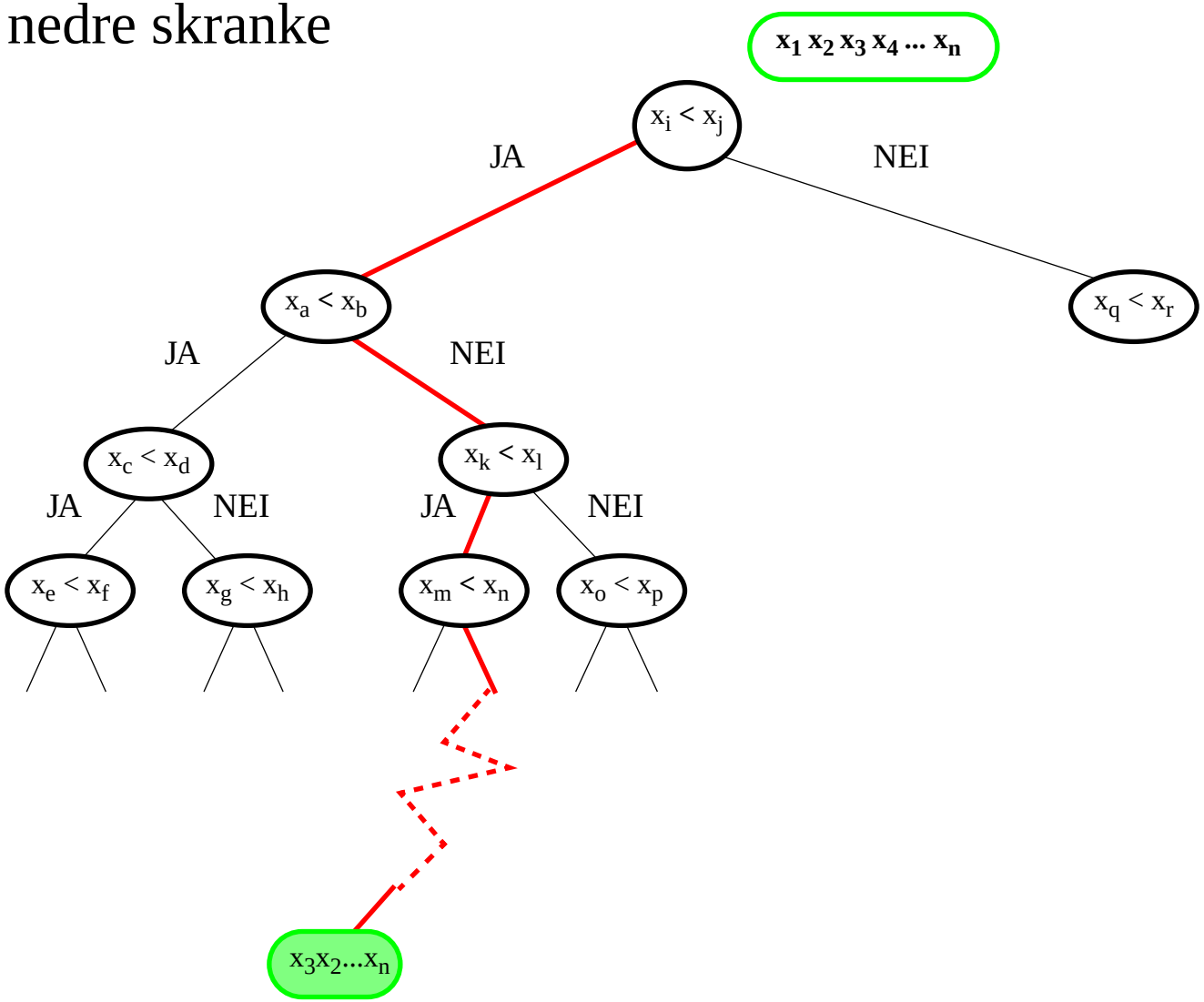
Sammenlikning-sortering: nedre skranke



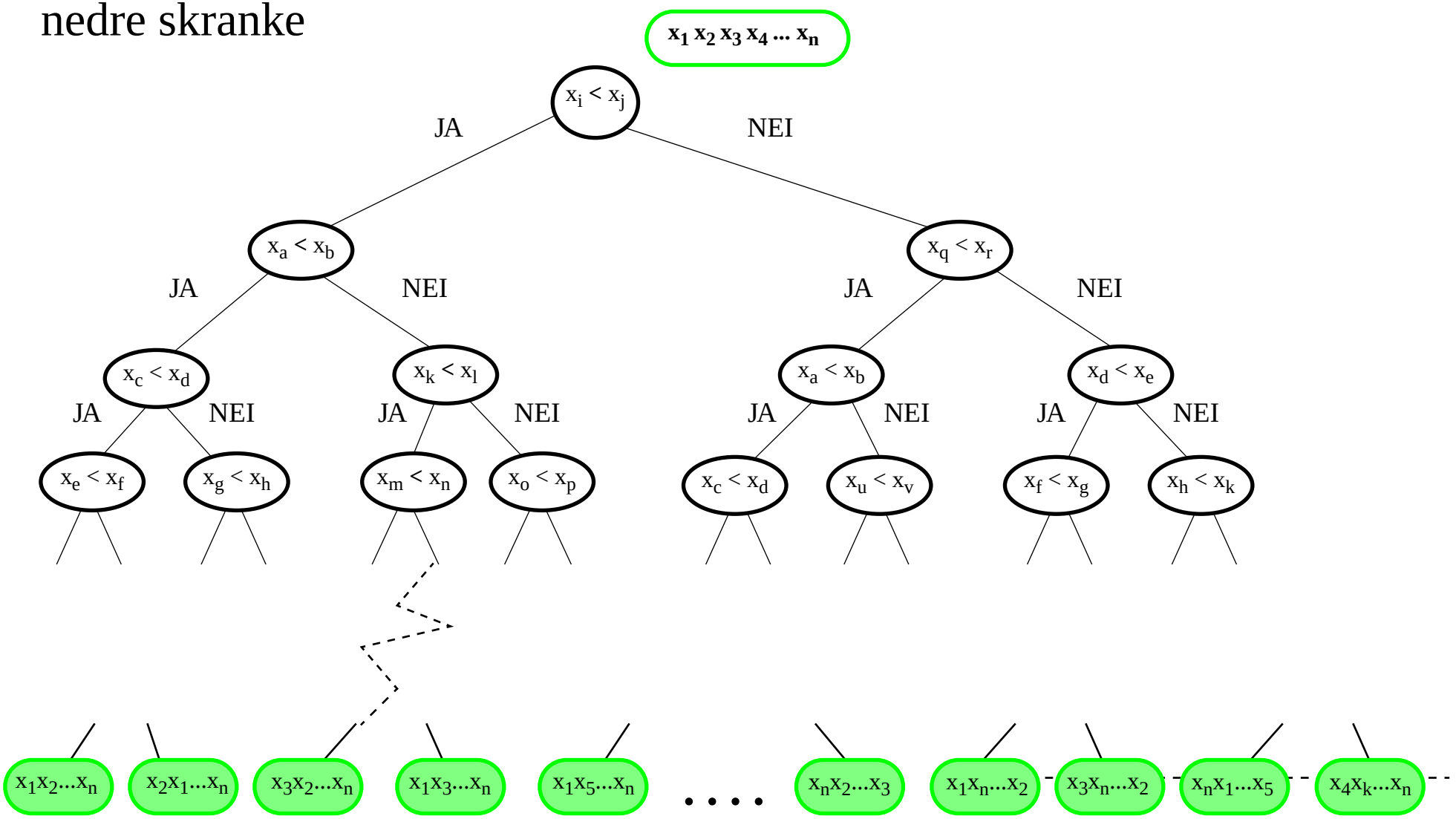
Sammenlikning-sortering: nedre skranke



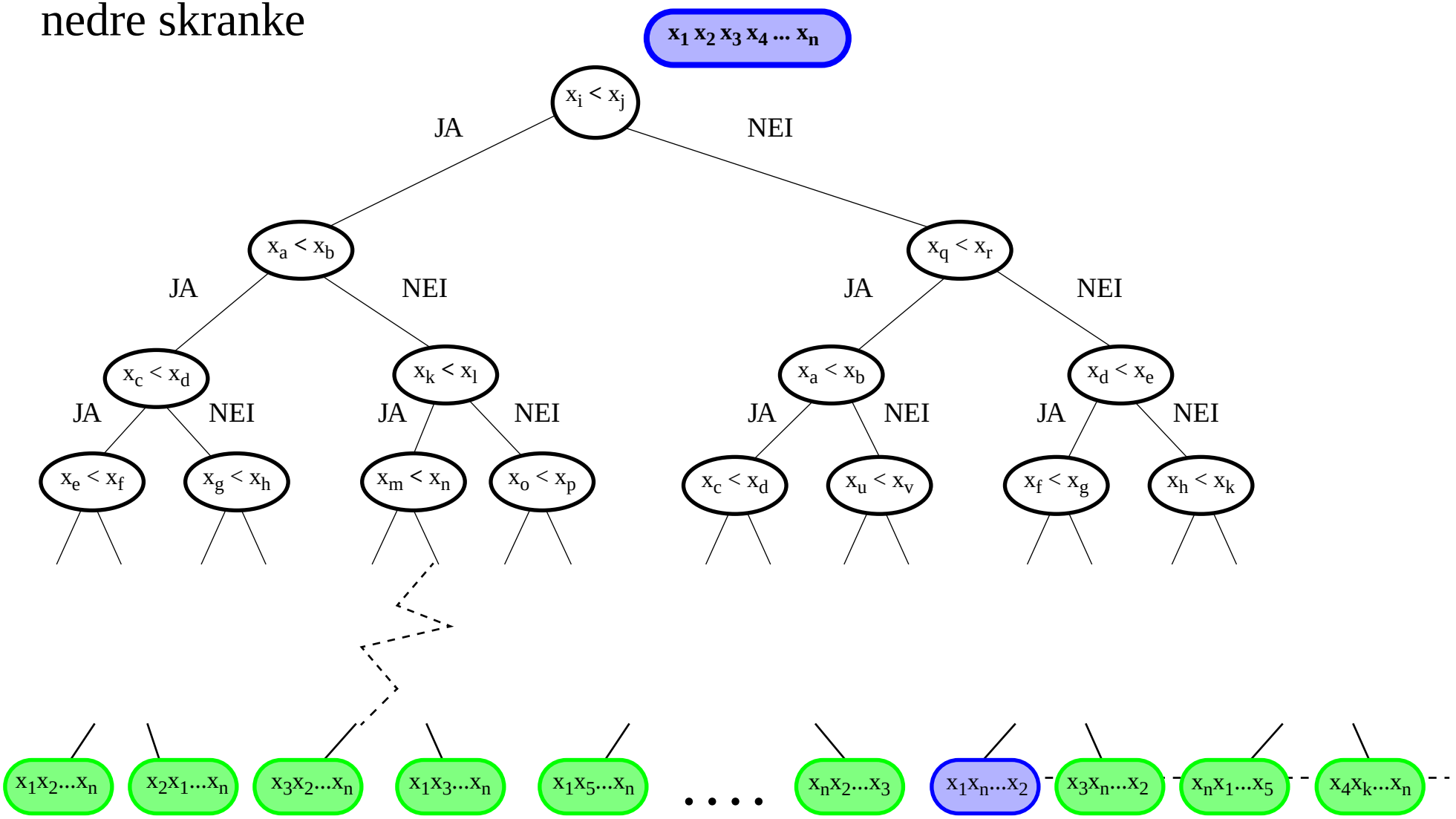
Sammenlikning-sortering: nedre skranke



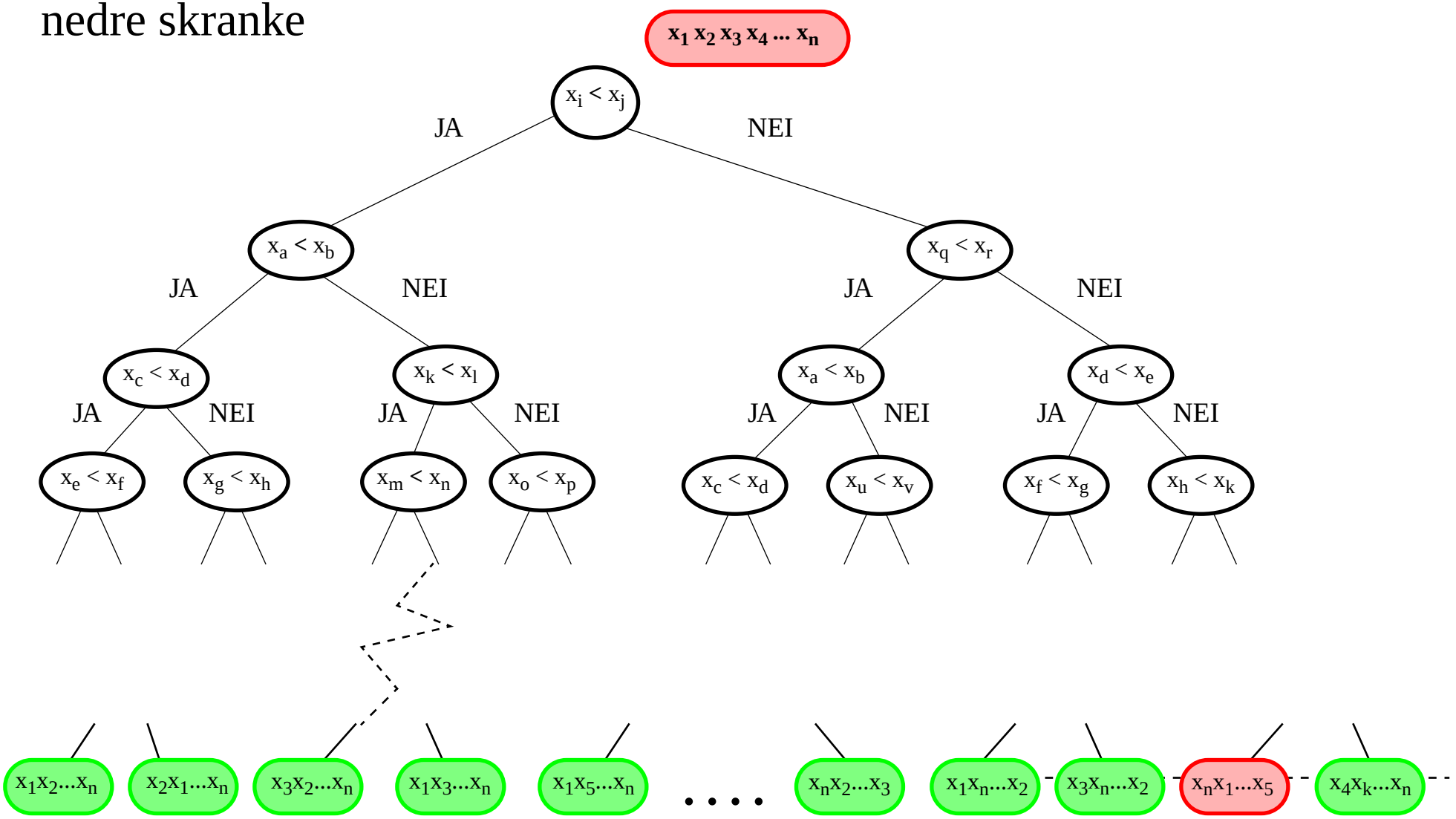
Sammenlikning-sortering: nedre skranke



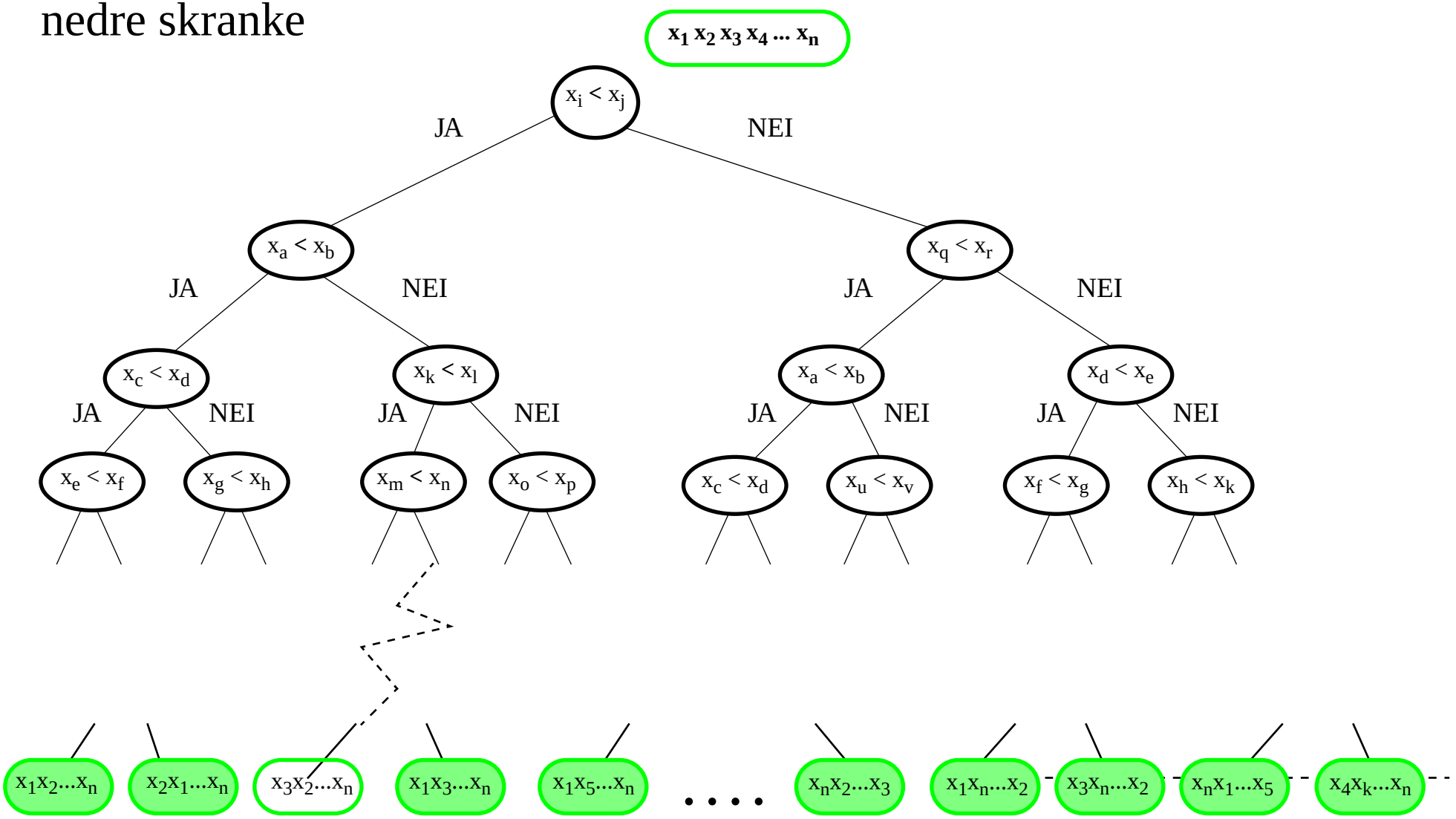
Sammenlikning-sortering: nedre skranke



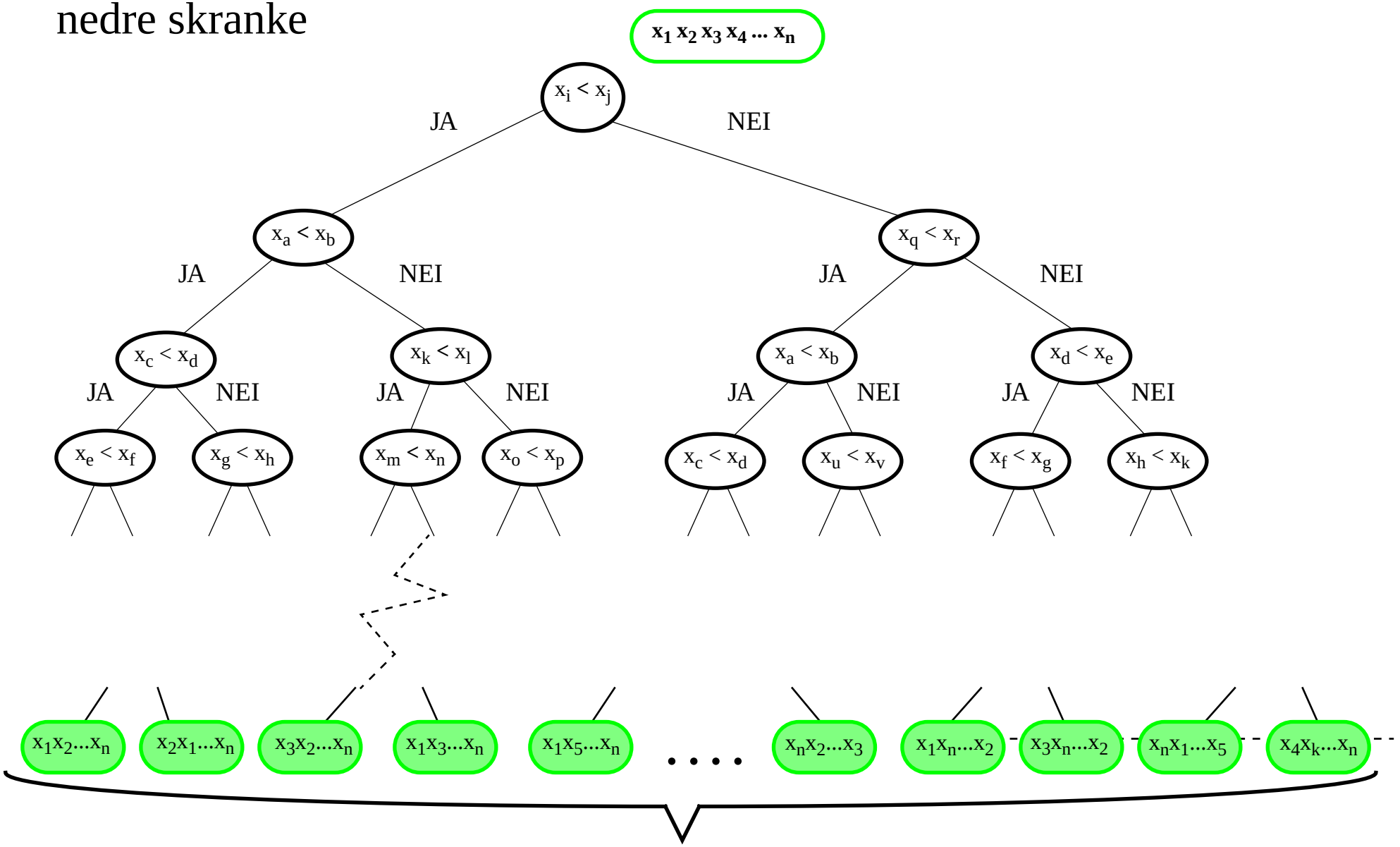
Sammenlikning-sortering: nedre skranke



Sammenlikning-sortering: nedre skranke

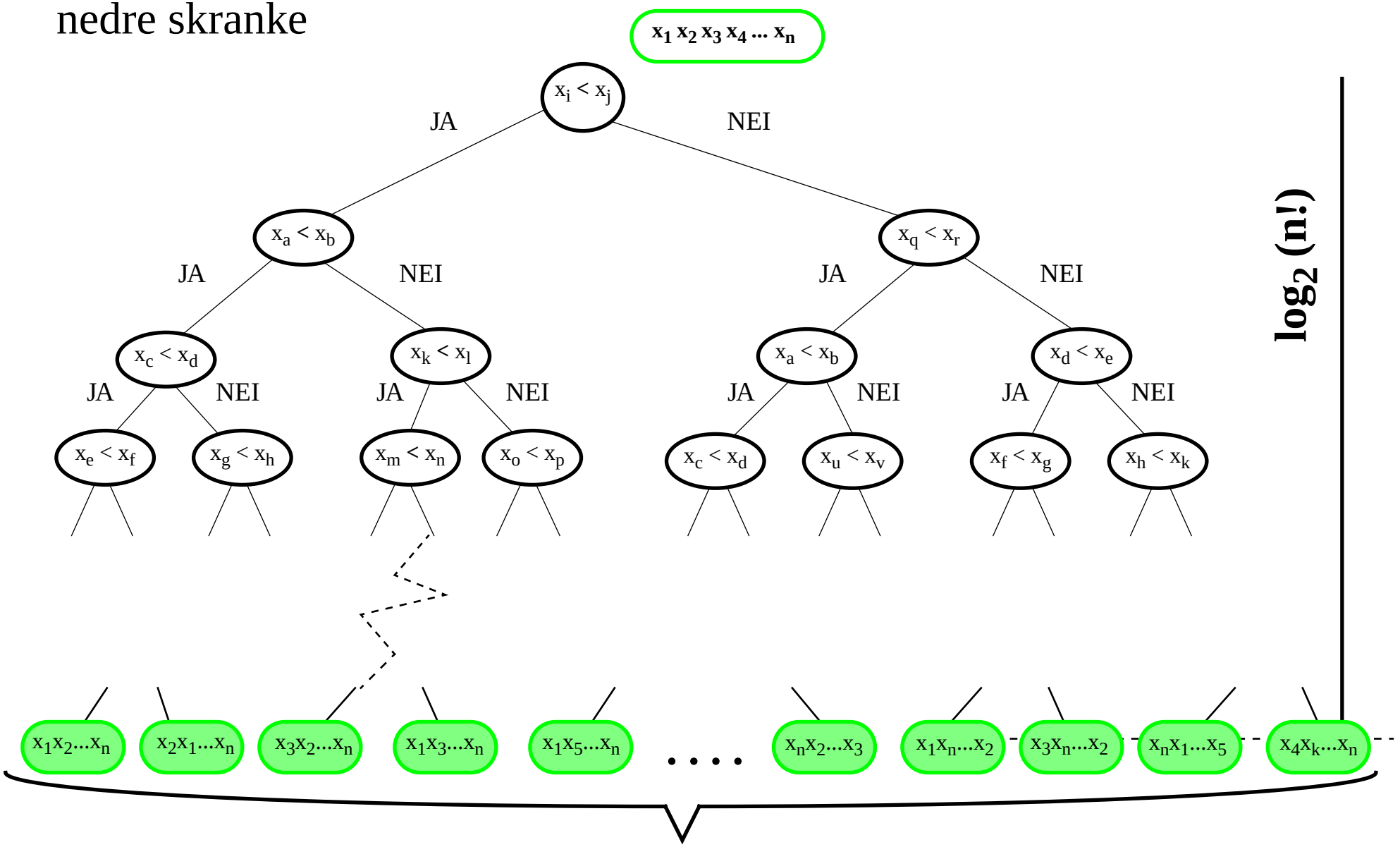


Sammenlikning-sortering: nedre skranke



antall permutasjoner av **n** elementer = **n!**

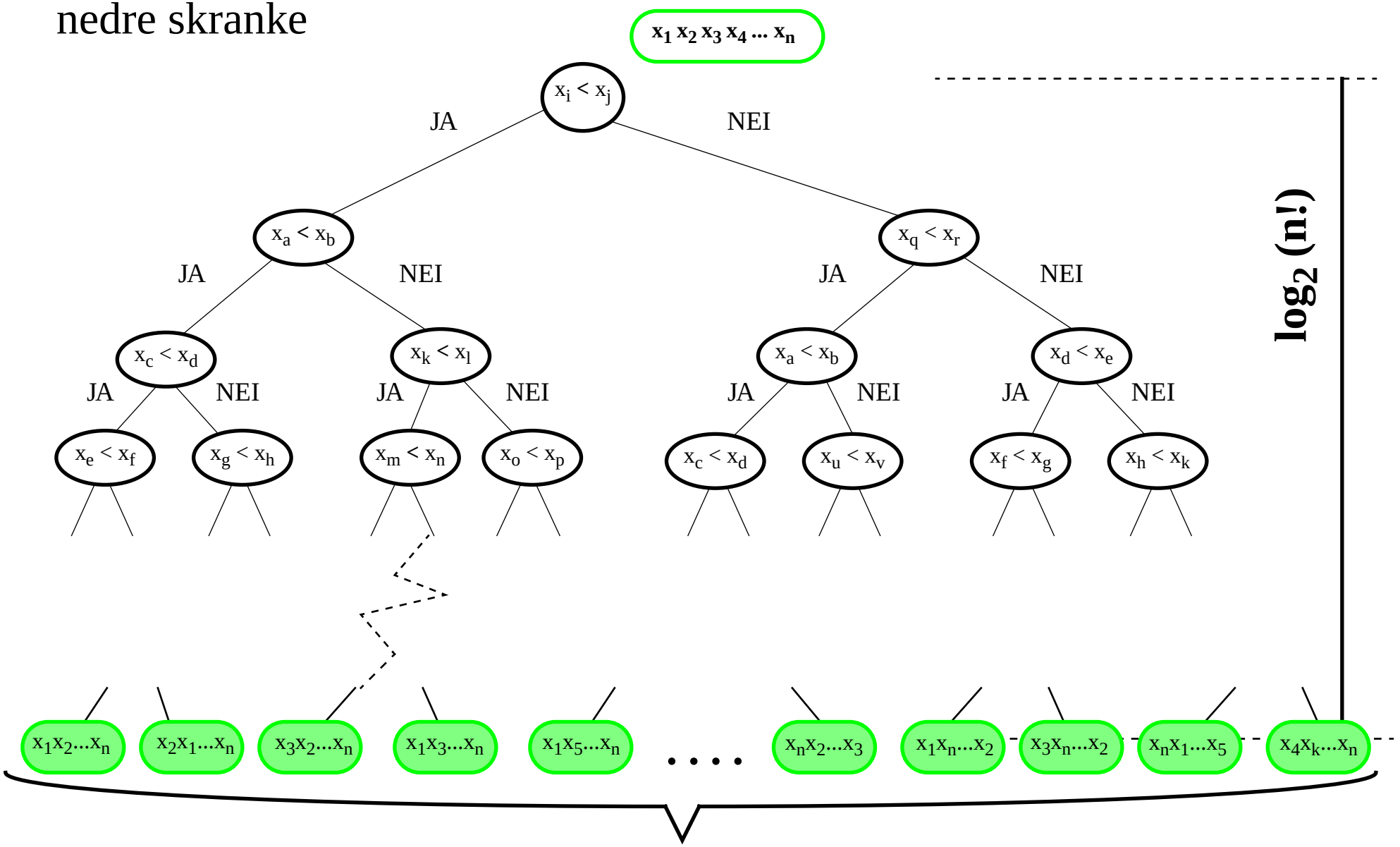
Sammenlikning-sortering: nedre skranke



antall permutasjoner av **n** elementer = **n!**

n! >= (n/2)$^{n/2}$

Sammenlikning-sortering: nedre skranke

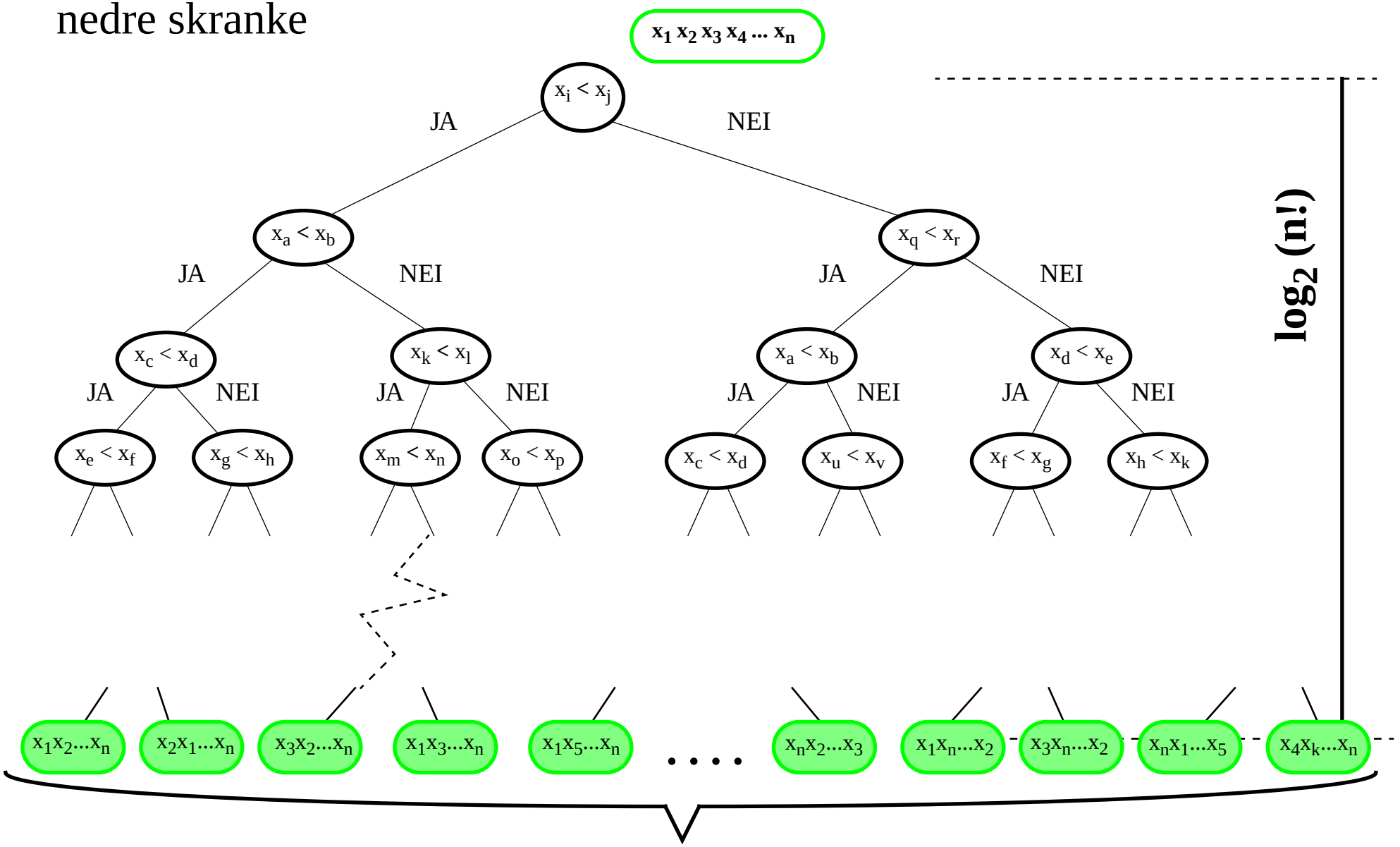


antall permutasjoner av **n** elementer = **n!**

Sammenlikning-sortering: nedre skranke

$n! >= (n/2)^{n/2}$

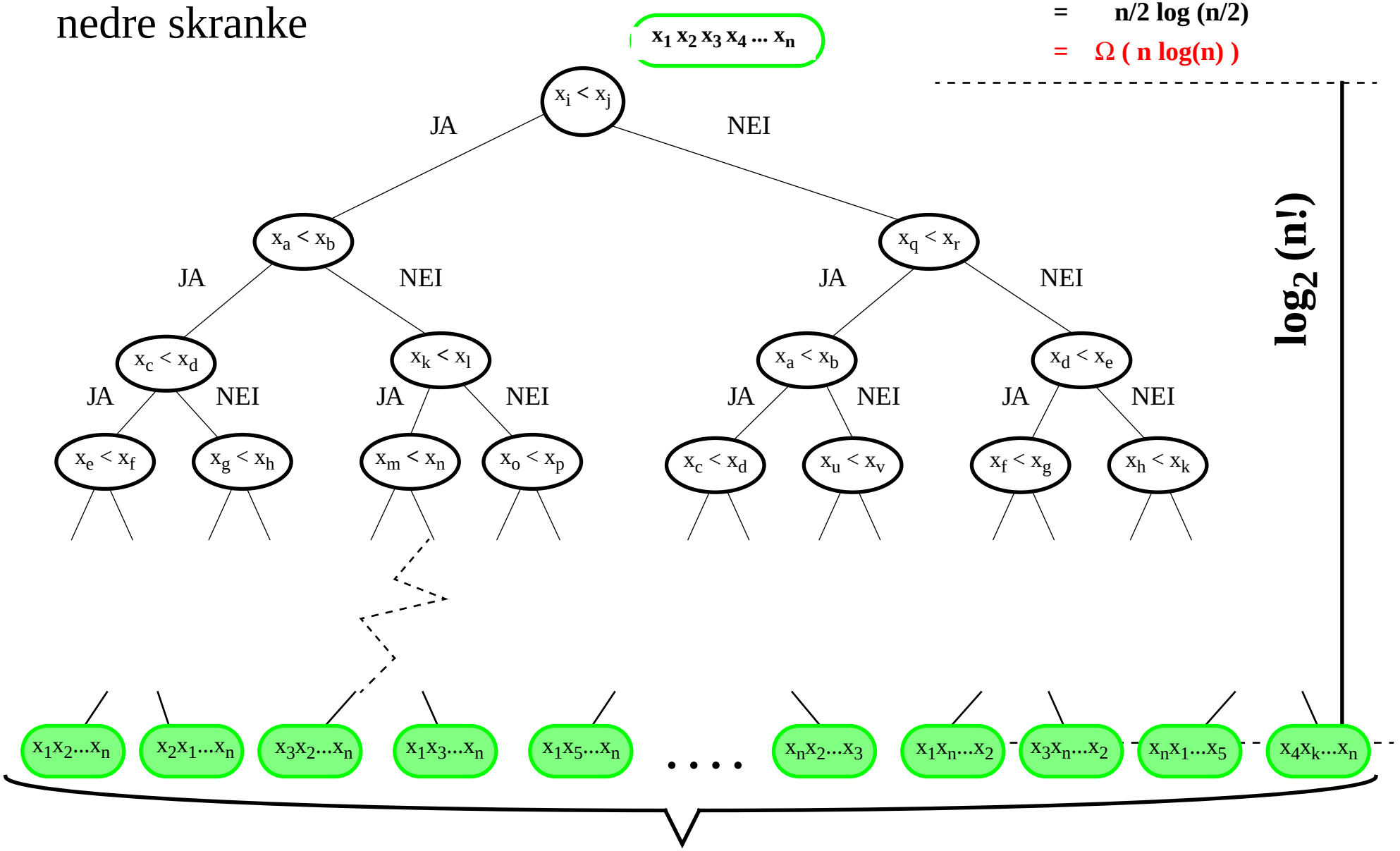
$\log(n!) >= \log (n/2)^{n/2}$



antall permutasjoner av **n** elementer = **n!**

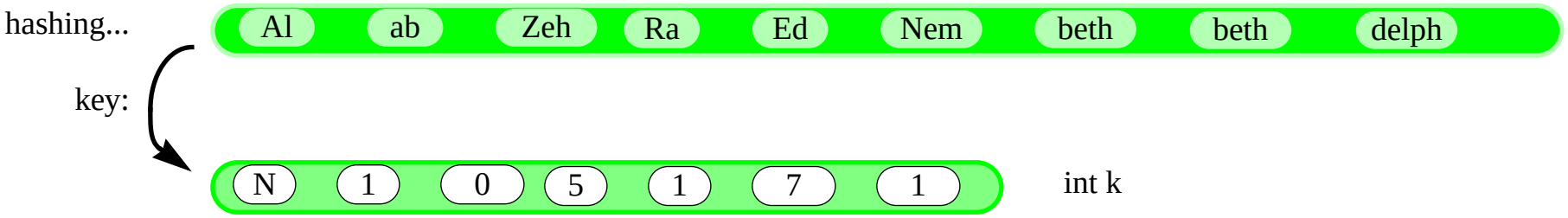
Sammenlikning-sortering: nedre skranke

$$n! >= (n/2)^{n/2}$$
$$\log(n!) >= \log (n/2)^{n/2}$$
$$= n/2 \log (n/2)$$
$$= \Omega (n \log(n))$$

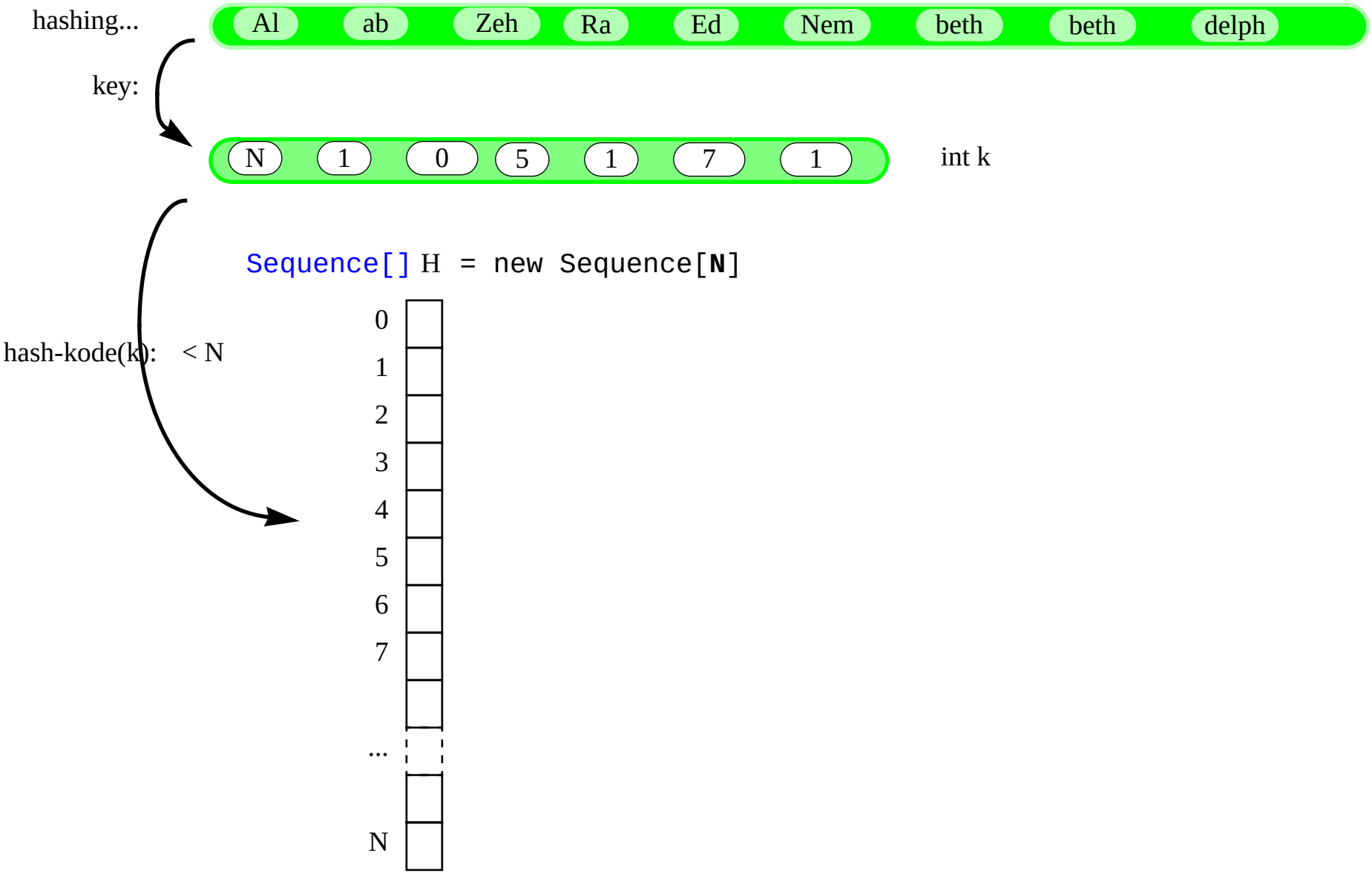


antall permutasjoner av **n** elementer = **n!**

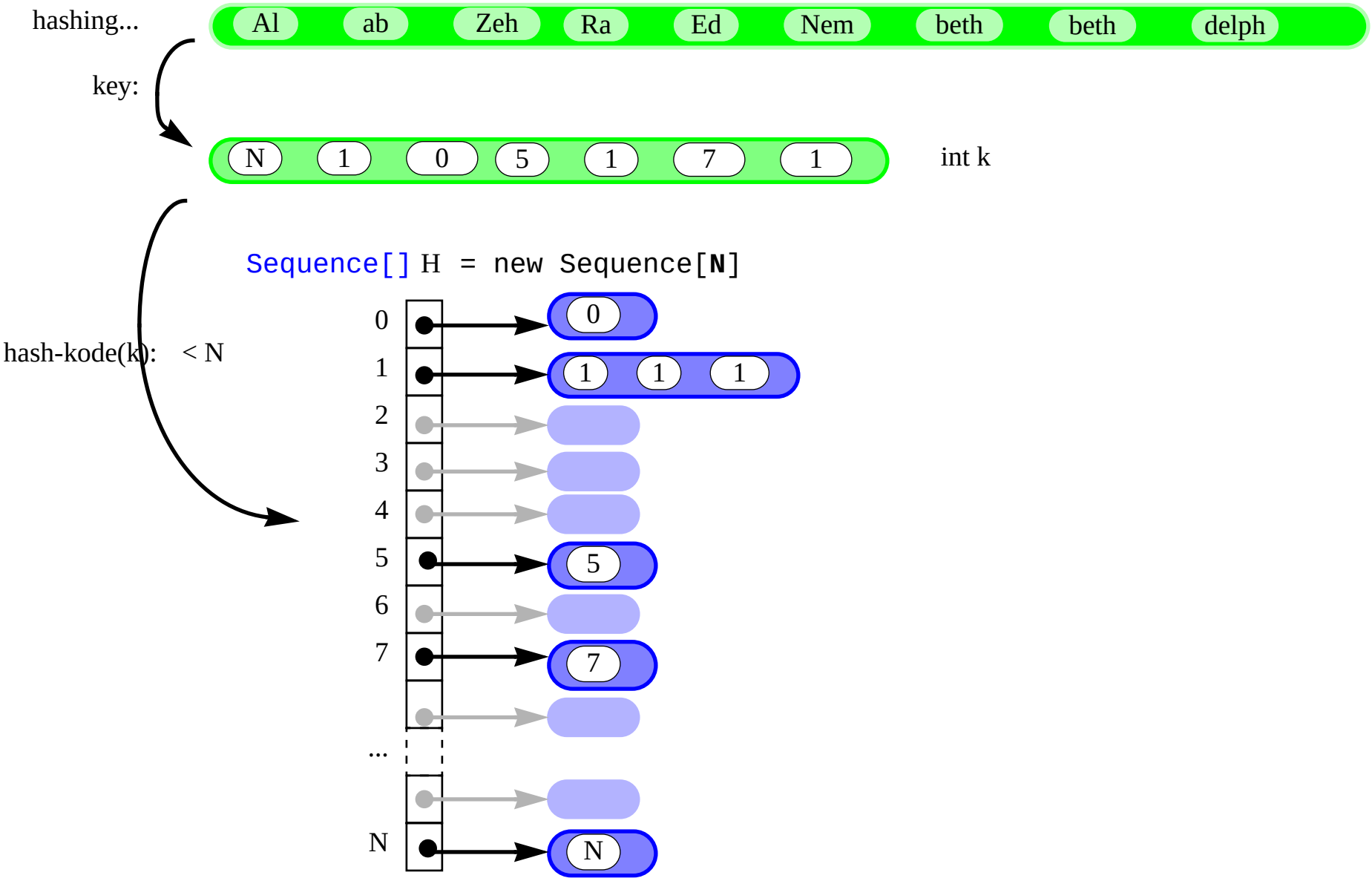
Bucket sort



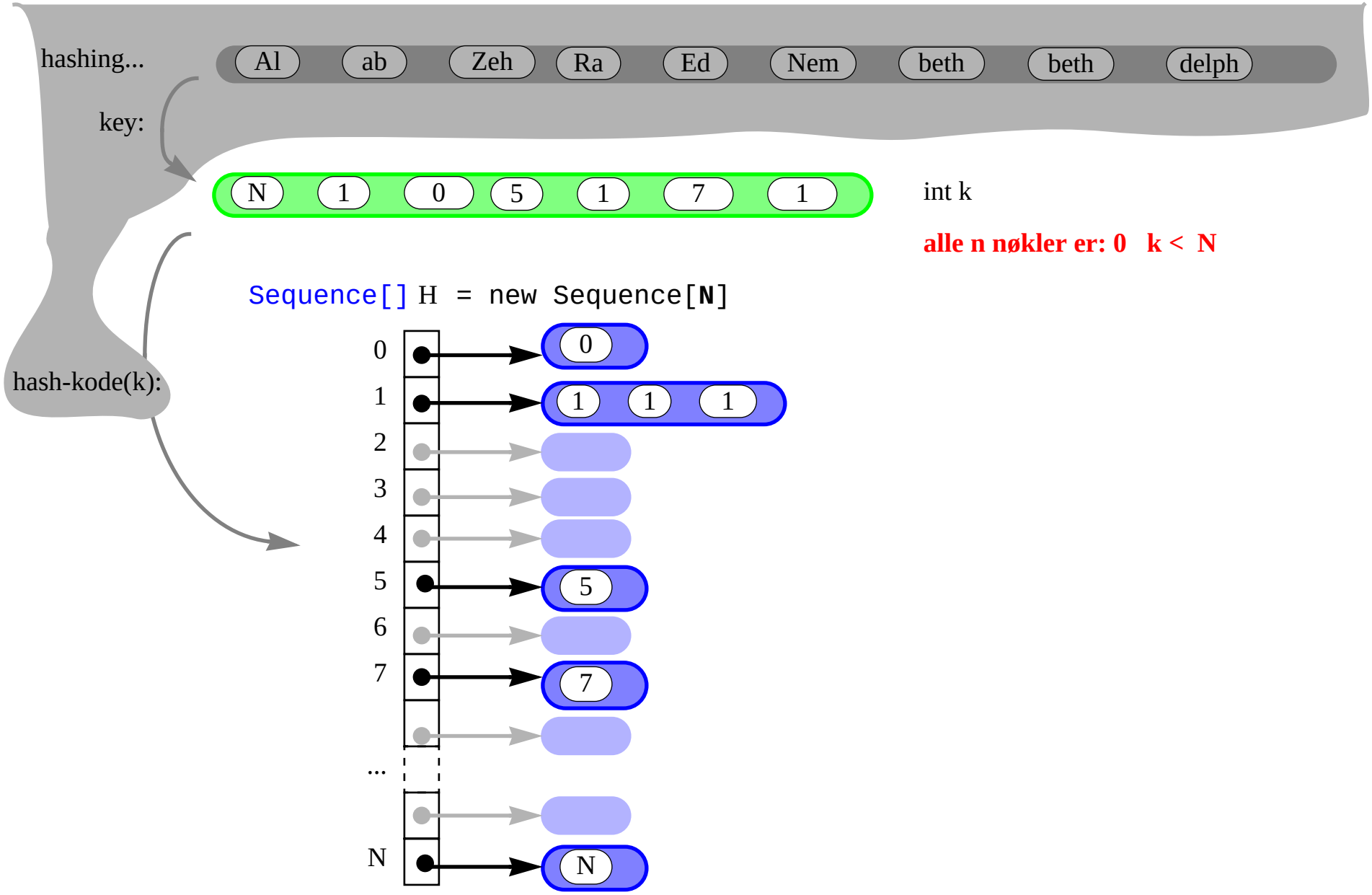
Bucket sort


hashing...
Al
ab
Zeh
Ra
Ed
Nem
beth
beth
delph
key:
N
1
0
5
1
7
1
int k
Sequence[] H = new Sequence[N]
hash-kode(k): < N
0
1
2
3
4
5
6
7
...
N

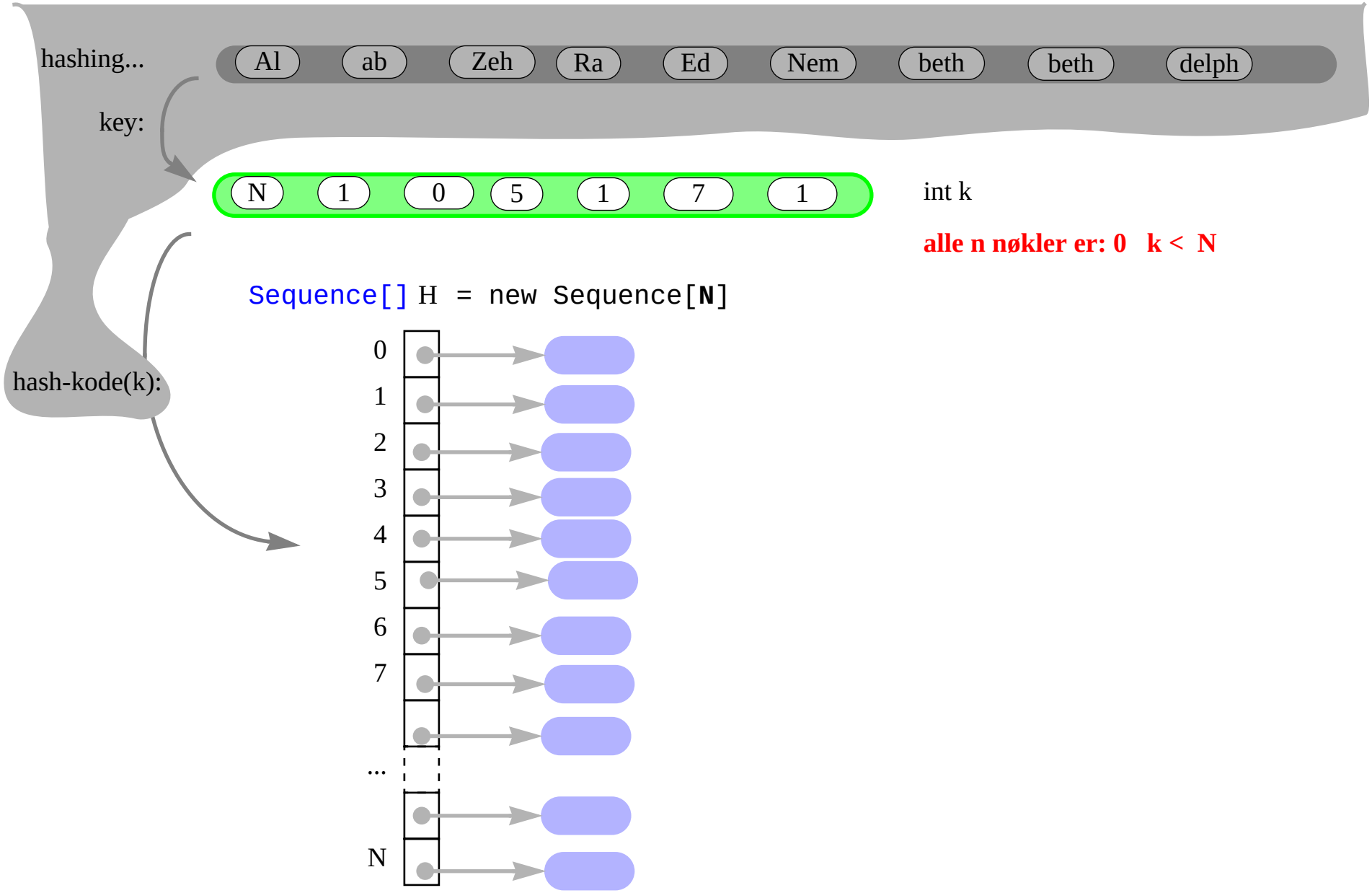
Bucket sort


hashing...
Al
ab
Zeh
Ra
Ed
Nem
beth
beth
delph
key:
N
1
0
5
1
7
1
int k
Sequence[] H = new Sequence[N]
hash-kode(k): < N
0
1
2
3
4
5
6
7
...
N
0
1
1
1
5
7
N

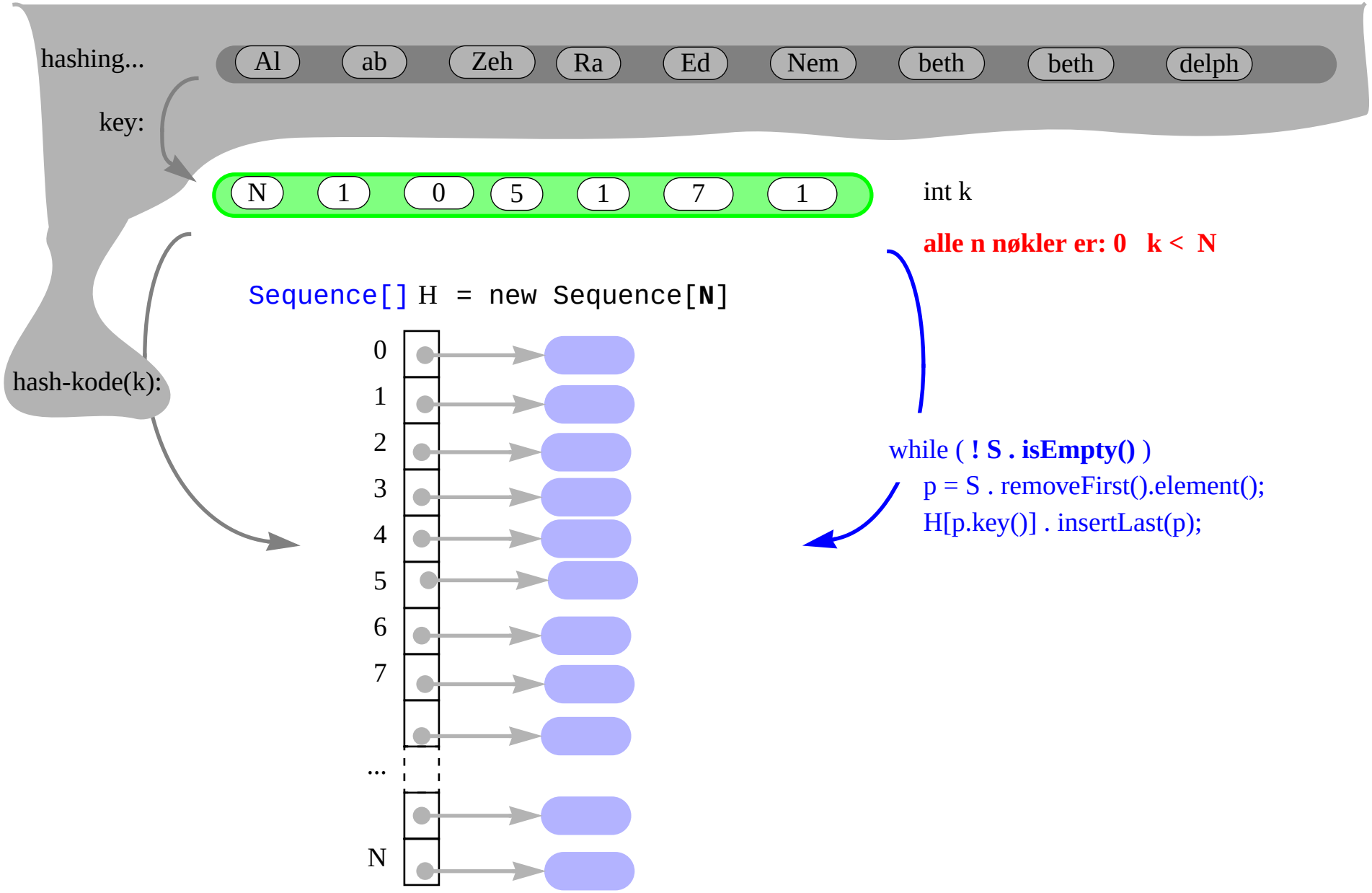
Bucket sort


hashing...
Al
ab
Zeh
Ra
Ed
Nem
beth
beth
delph
key:
N
1
0
5
1
7
1
int k
alle n nøkler er: 0 k < N
Sequence[] H = new Sequence[N]
hash-kode(k):
0
1
2
3
4
5
6
7
...
N
0
1
1
1
5
7
N

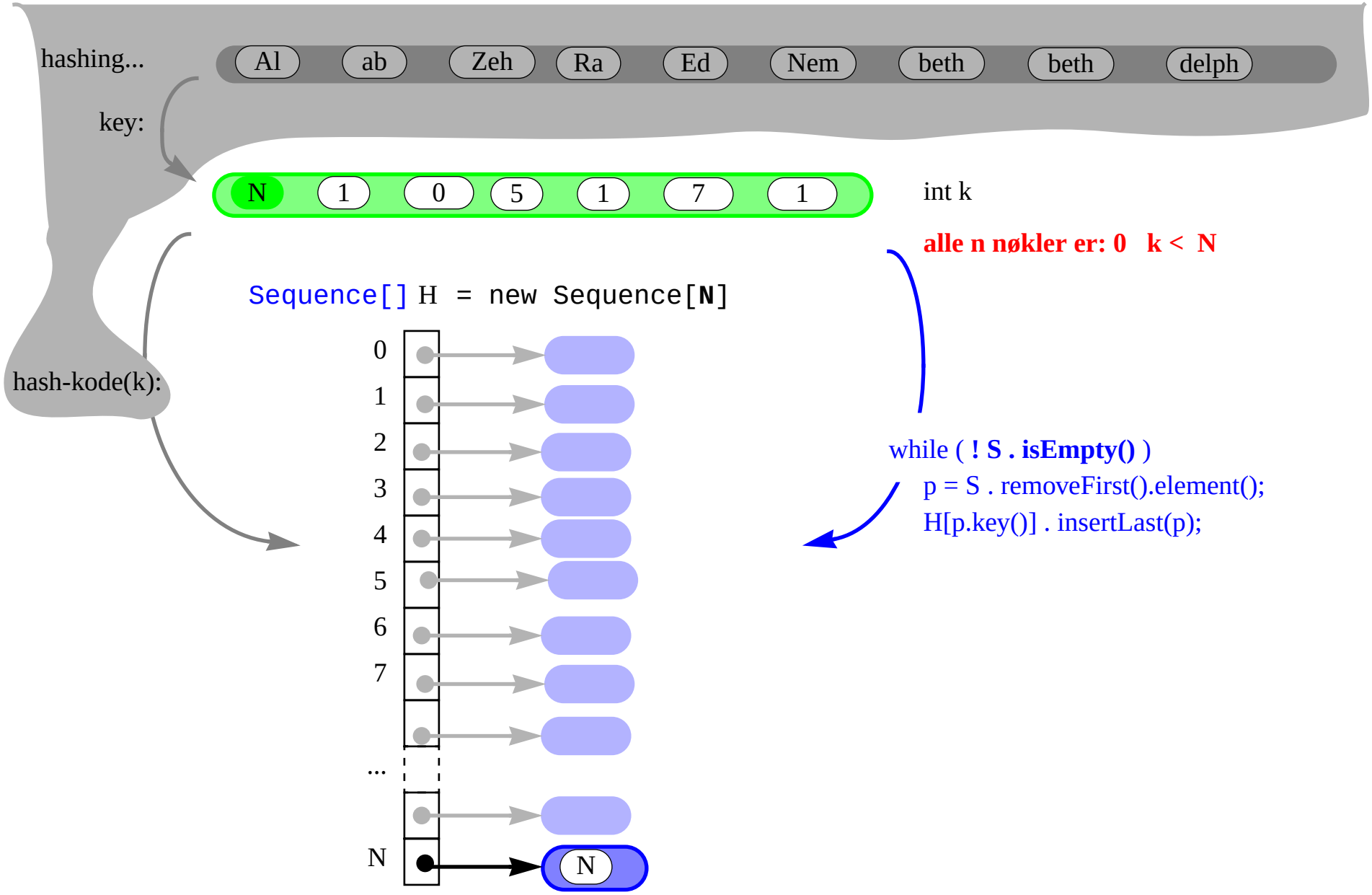
Bucket sort


hashing...
Al
ab
Zeh
Ra
Ed
Nem
beth
beth
delph
key:
N
1
0
5
1
7
1
int k
alle n nøkler er: 0 k < N
Sequence[] H = new Sequence[N]
hash-kode(k):
0
1
2
3
4
5
6
7
...
N

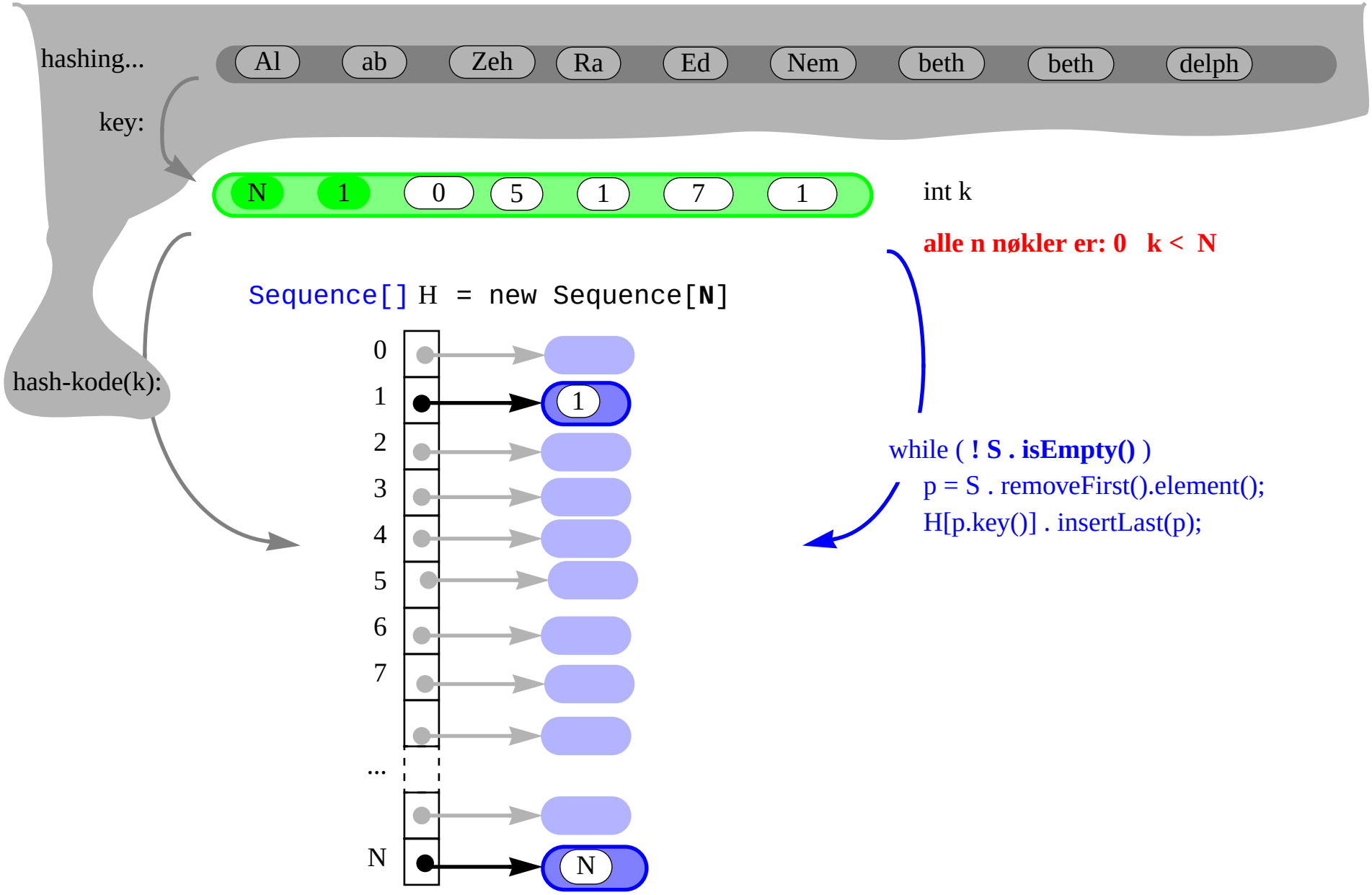
Bucket sort


hashing...
Al
ab
Zeh
Ra
Ed
Nem
beth
beth
delph
key:
N
1
0
5
1
7
1
int k
alle n nøkler er: 0 k < N
Sequence[] H = new Sequence[N]
hash-kode(k):
0
1
2
3
4
5
6
7
...
N
while (! S . isEmpty())
p = S . removeFirst().element();
H[p.key()] . insertLast(p);

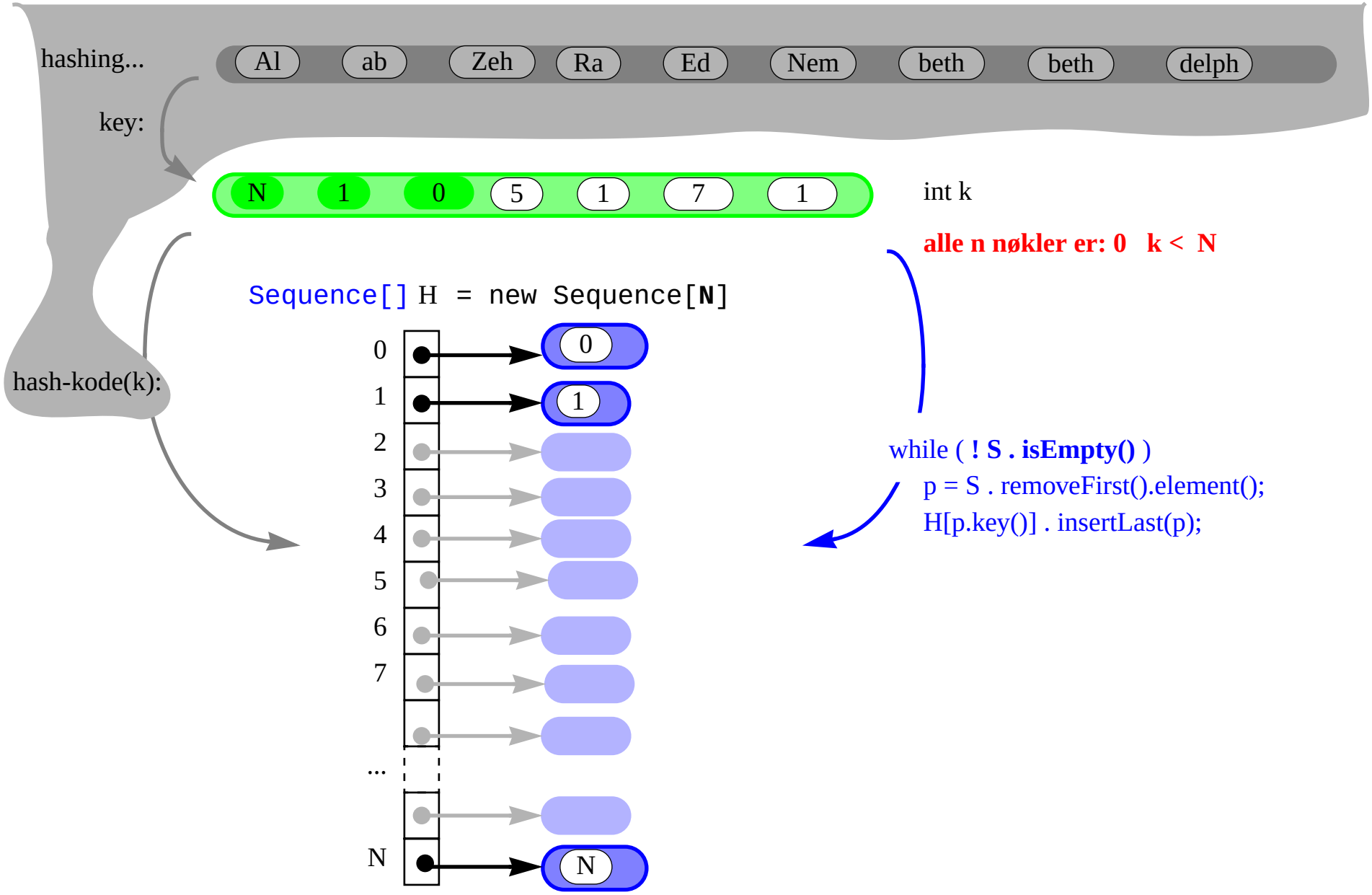
Bucket sort


hashing...
Al
ab
Zeh
Ra
Ed
Nem
beth
beth
delph
key:
N
1
0
5
1
7
1
int k
alle n nøkler er: 0 k < N
Sequence[] H = new Sequence[N]
hash-kode(k):
0
1
2
3
4
5
6
7
...
N
N
while (! S . isEmpty())
p = S . removeFirst().element();
H[p.key()] . insertLast(p);

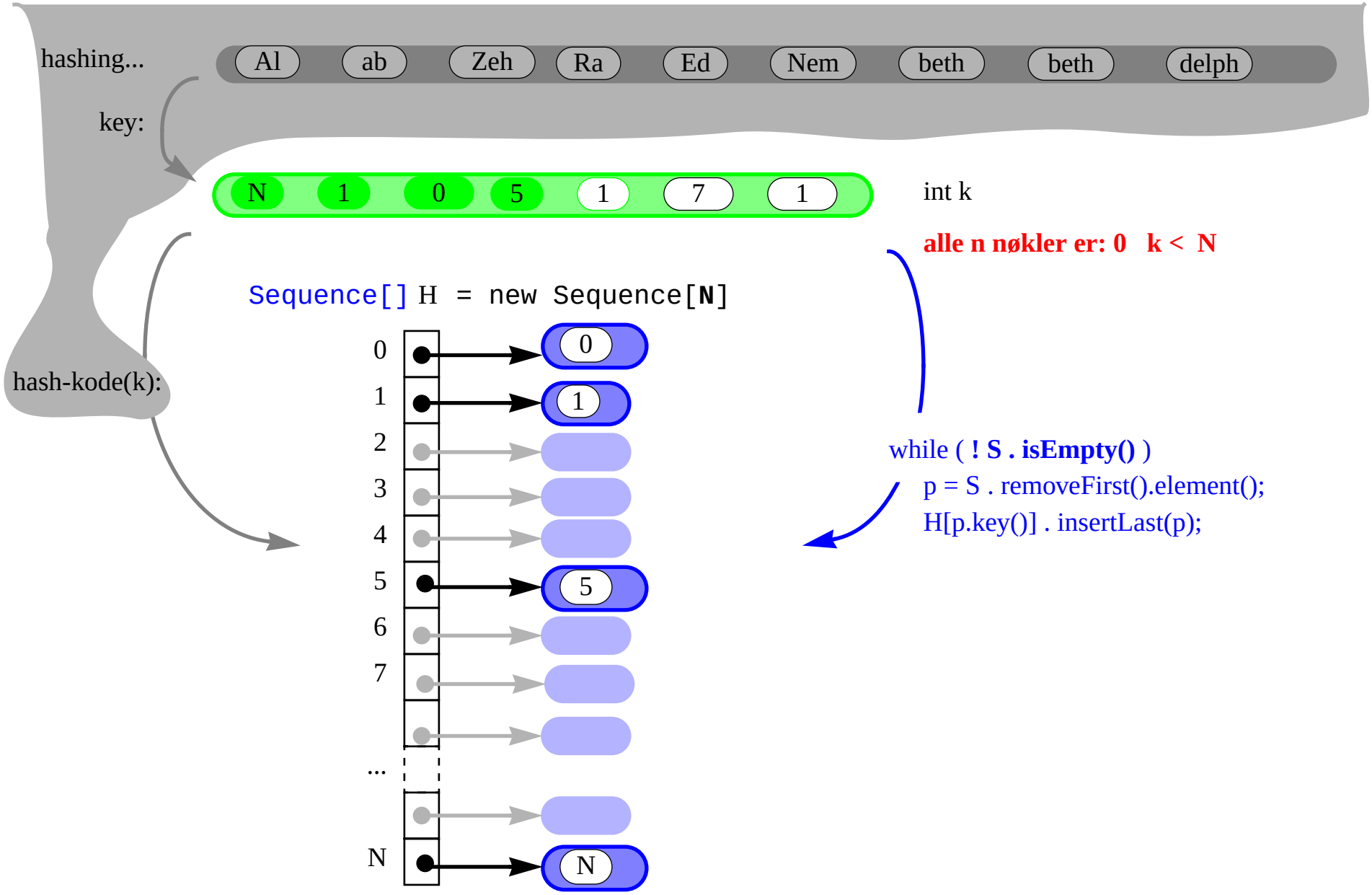
Bucket sort


hashing...
Al
ab
Zeh
Ra
Ed
Nem
beth
beth
delph
key:
N
1
0
5
1
7
1
int k
alle n nøkler er: 0 k < N
Sequence[] H = new Sequence[N]
hash-kode(k):
0
1
2
3
4
5
6
7
...
N
1
N
while (! S . isEmpty())
p = S . removeFirst().element();
H[p.key()] . insertLast(p);

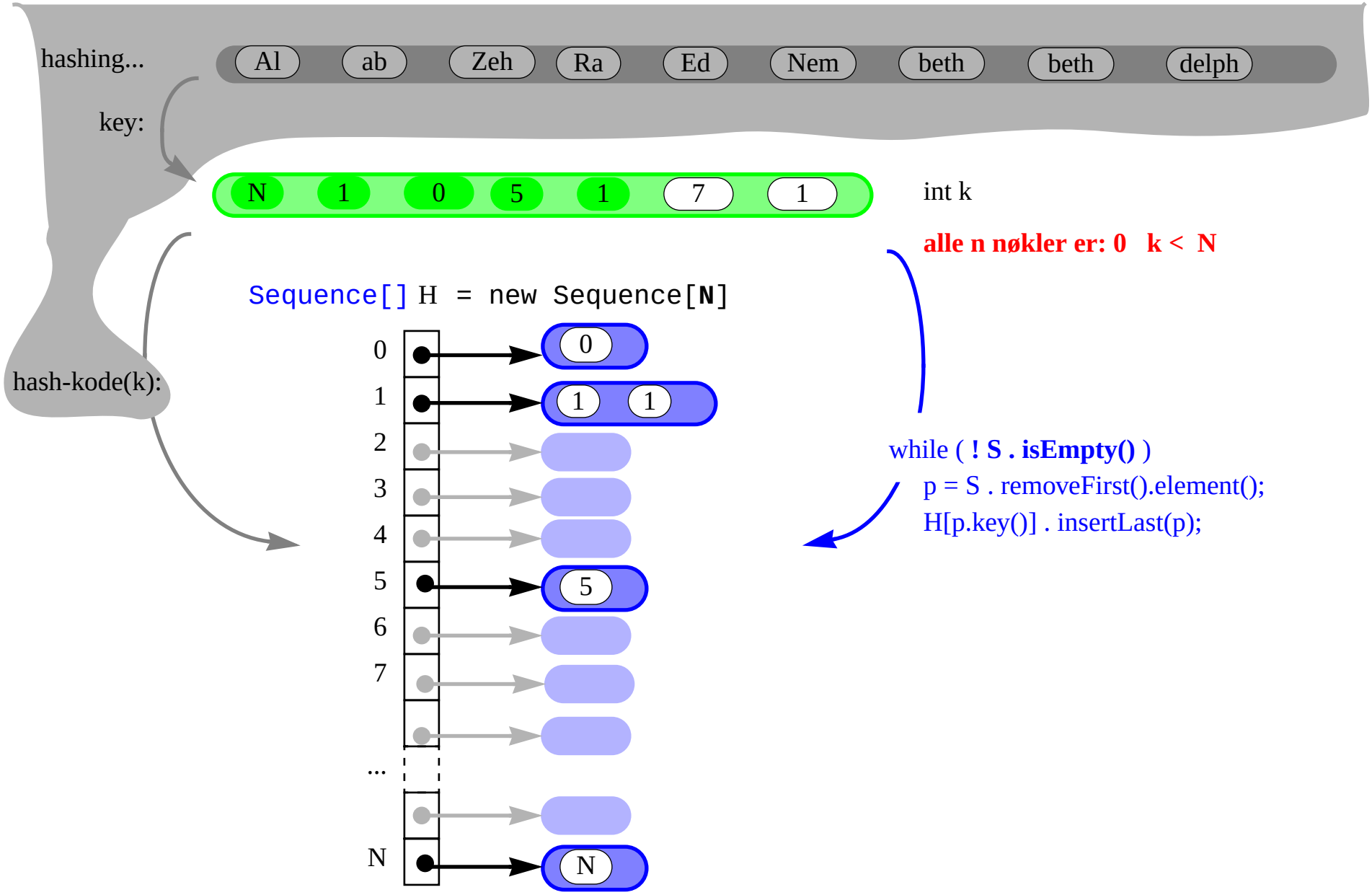
Bucket sort


hashing...
Al
ab
Zeh
Ra
Ed
Nem
beth
beth
delph
key:
N
1
0
5
1
7
1
int k
alle n nøkler er: 0 k < N
Sequence[] H = new Sequence[N]
hash-kode(k):
0
1
2
3
4
5
6
7
...
N
while (! S . isEmpty())
p = S . removeFirst().element();
H[p.key()] . insertLast(p);

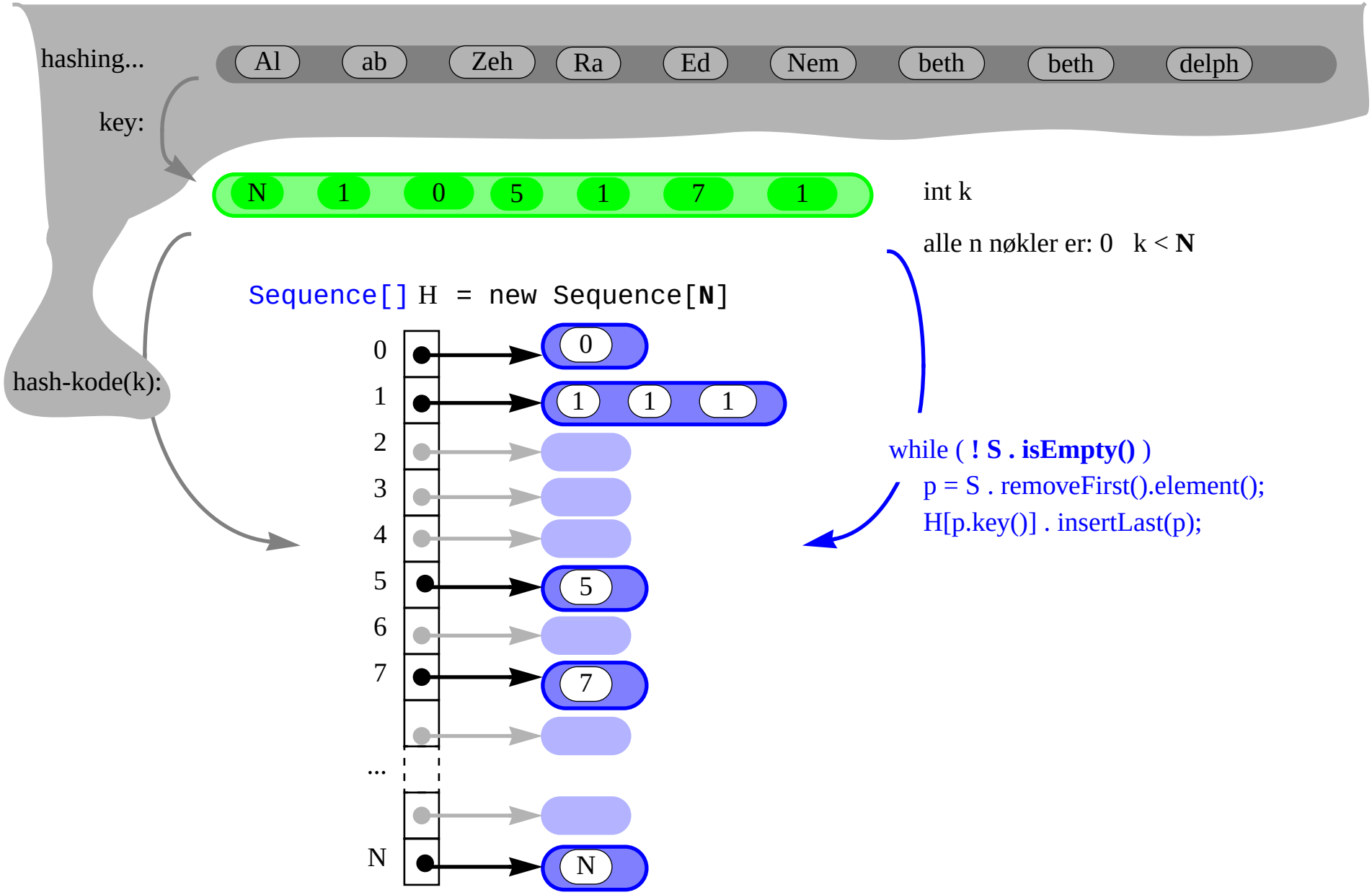
Bucket sort


hashing...
Al
ab
Zeh
Ra
Ed
Nem
beth
beth
delph
key:
N
1
0
5
1
7
1
int k
alle n nøkler er: 0 k < N
Sequence[] H = new Sequence[N]
hash-kode(k):
0
1
2
3
4
5
6
7
...
N
while (! S . isEmpty())
p = S . removeFirst().element();
H[p.key()] . insertLast(p);

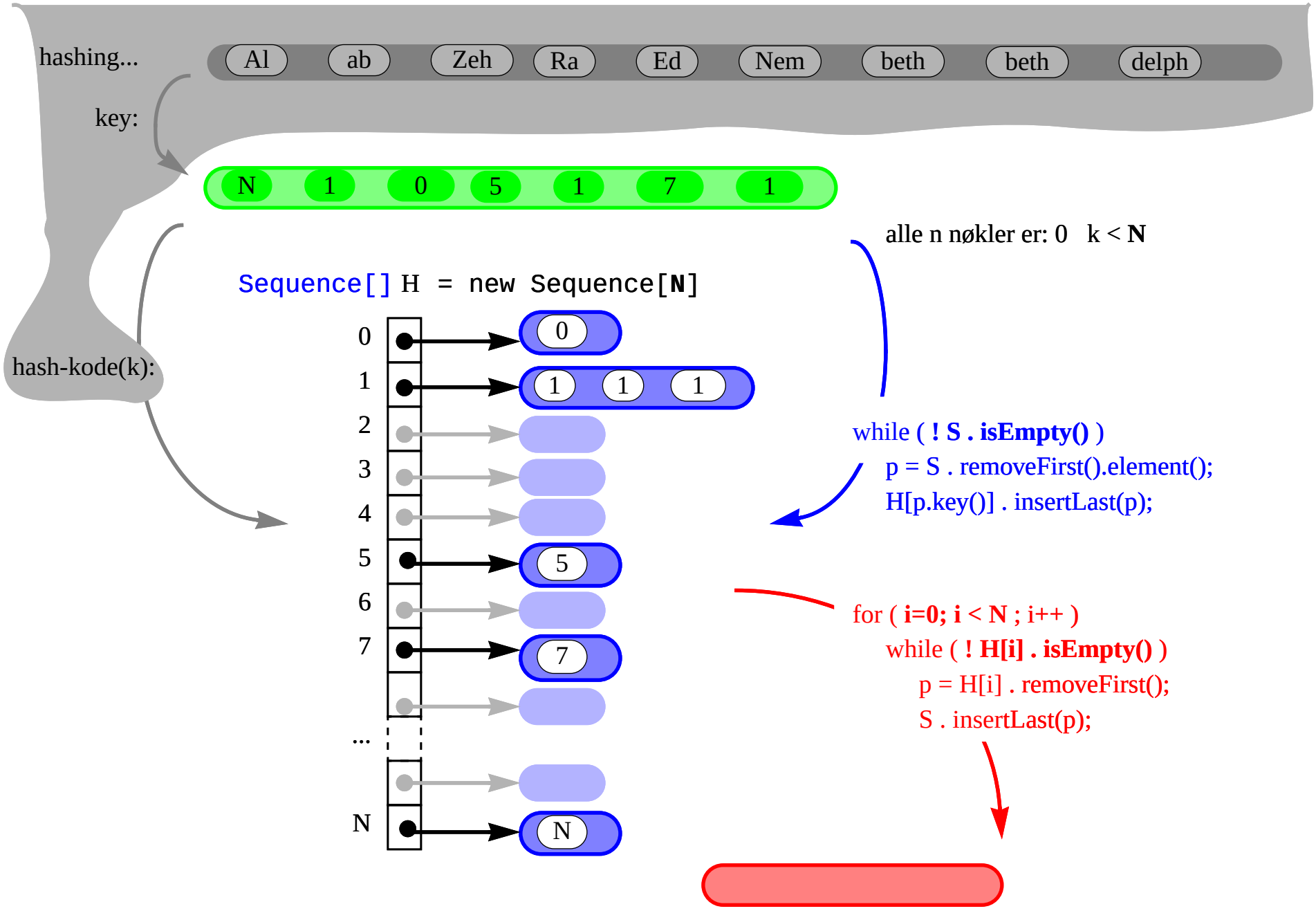
Bucket sort


hashing...
Al
ab
Zeh
Ra
Ed
Nem
beth
beth
delph
key:
N
1
0
5
1
7
1
int k
alle n nøkler er: 0 k < N
Sequence[] H = new Sequence[N]
hash-kode(k):
0
1
2
3
4
5
6
7
...
N
0
1
1
5
N
while (! S . isEmpty())
p = S . removeFirst().element();
H[p.key()] . insertLast(p);

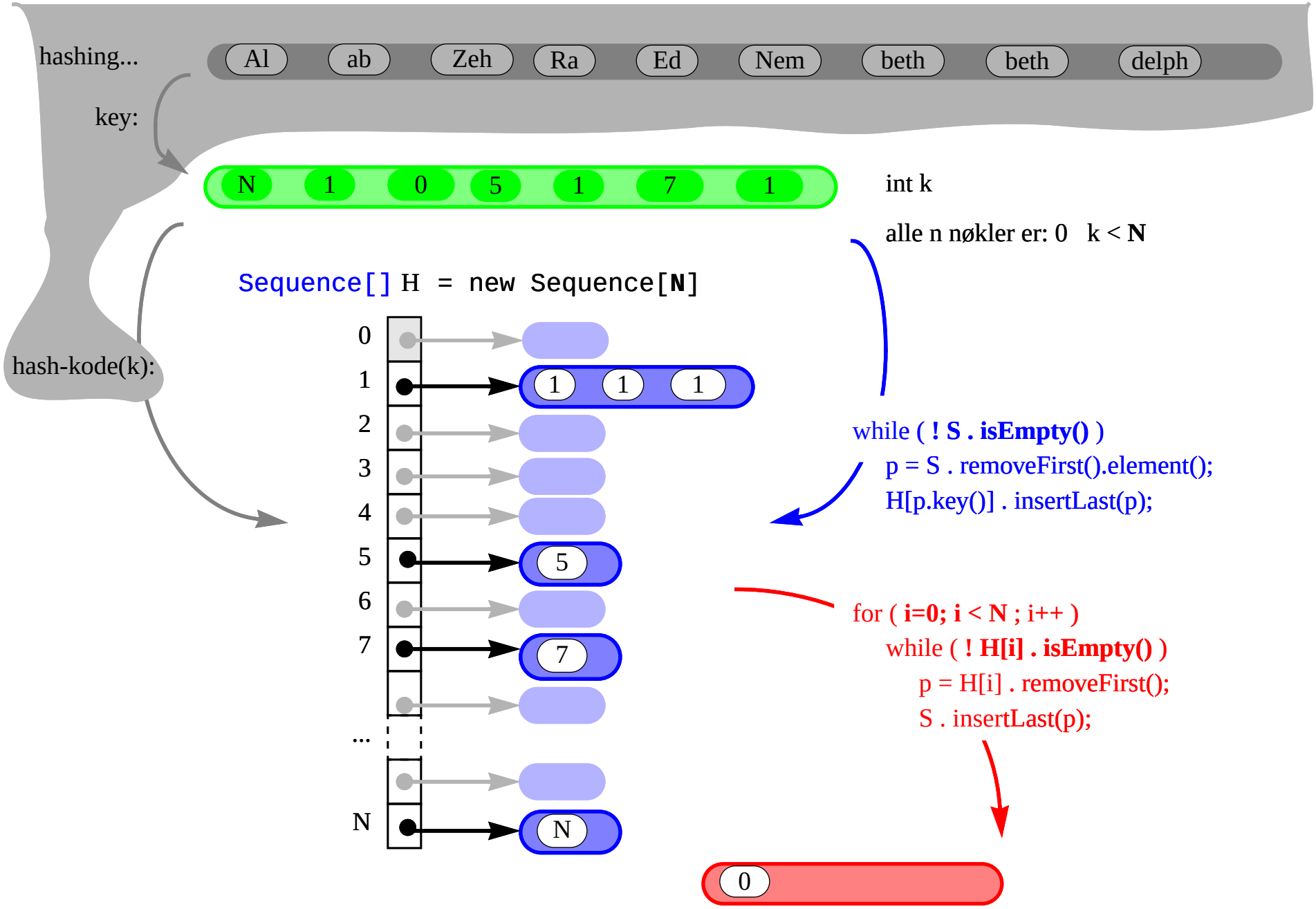
Bucket sort


hashing...
Al
ab
Zeh
Ra
Ed
Nem
beth
beth
delph
key:
N
1
0
5
1
7
1
int k
alle n nøkler er: 0 k < N
Sequence[] H = new Sequence[N]
hash-kode(k):
0
1
2
3
4
5
6
7
...
N
0
1
1
1
5
7
N
while (! S . isEmpty())
p = S . removeFirst().element();
H[p.key()] . insertLast(p);

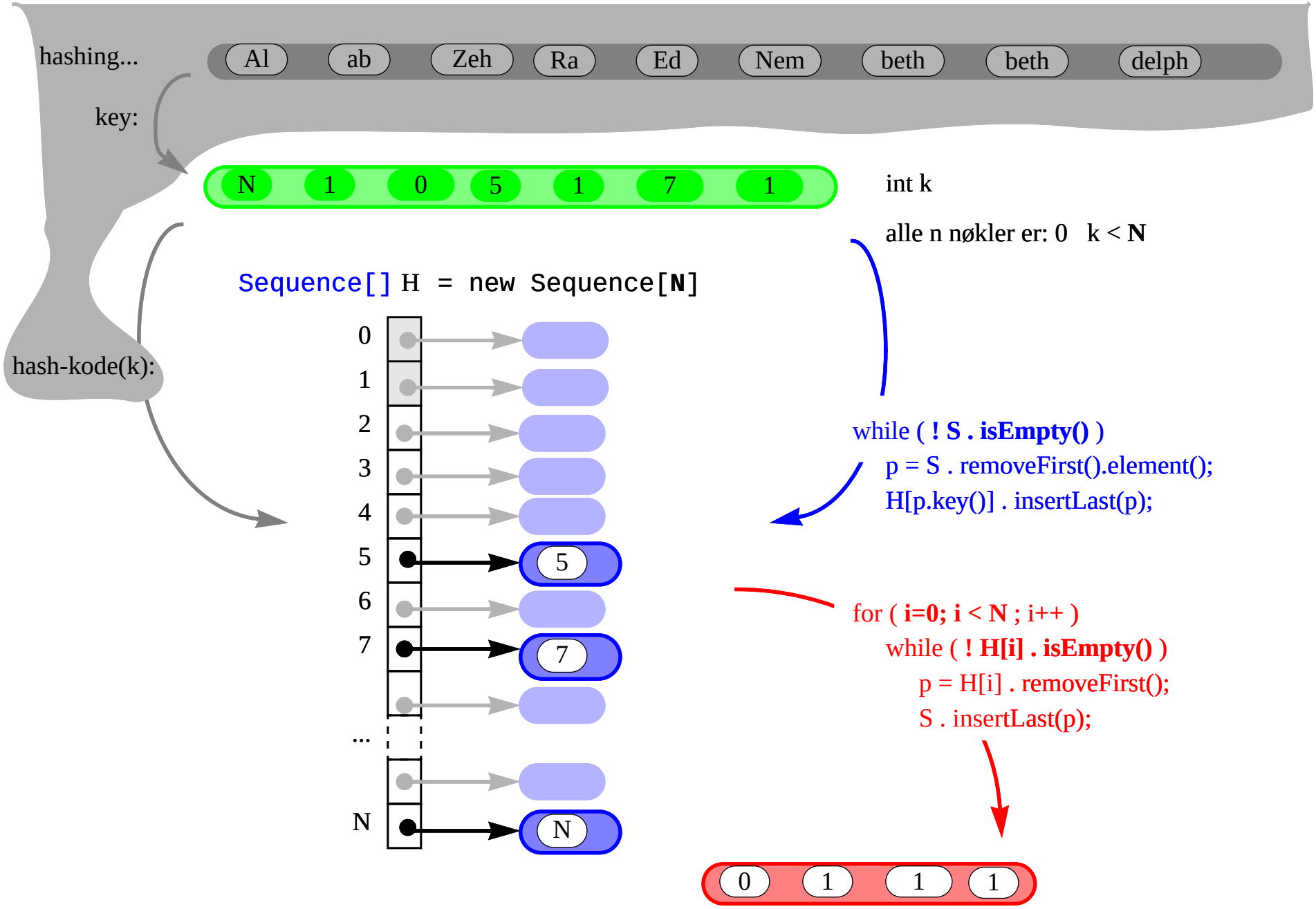
Bucket sort


hashing...
Al
ab
Zeh
Ra
Ed
Nem
beth
beth
delph
key:
N 1 0 5 1 7 1
alle n nøkler er: 0 k < N
Sequence[] H = new Sequence[N]
hash-kode(k):
0
1
2
3
4
5
6
7
...
N
while (! S . isEmpty())
p = S . removeFirst().element();
H[p.key()] . insertLast(p);
for (i=0; i < N ; i++)
while (! H[i] . isEmpty())
p = H[i] . removeFirst();
S . insertLast(p);

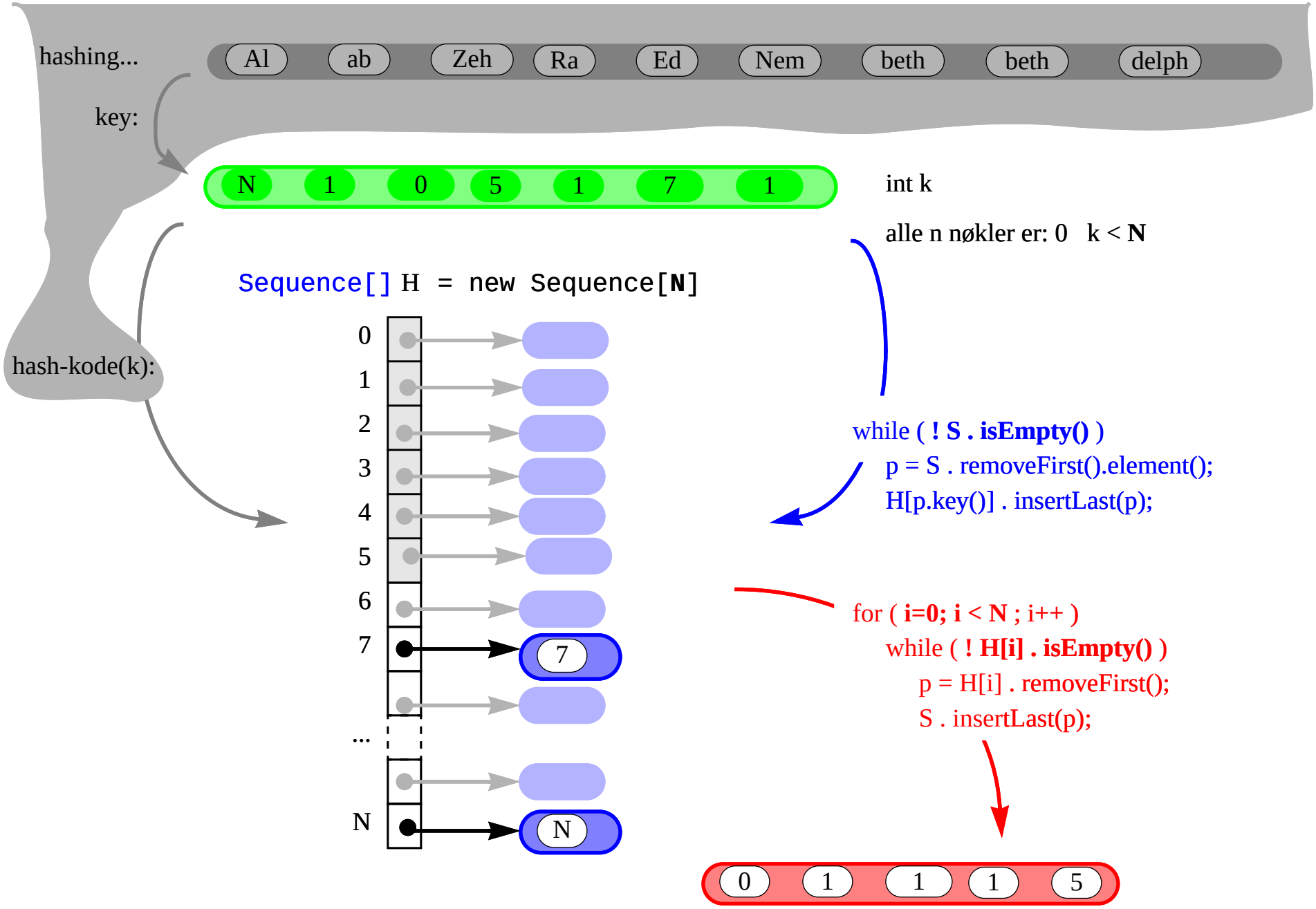
Bucket sort


hashing...
Al
ab
Zeh
Ra
Ed
Nem
beth
beth
delph
key:
N
1
0
5
1
7
1
int k
alle n nøkler er: 0 k < N
Sequence[] H = new Sequence[N]
hash-kode(k):
0
1
2
3
4
5
6
7
...
N
1
1
1
5
7
N
while (! S . isEmpty())
p = S . removeFirst().element();
H[p.key()] . insertLast(p);
for (i=0; i < N ; i++)
while (! H[i] . isEmpty())
p = H[i] . removeFirst();
S . insertLast(p);
0

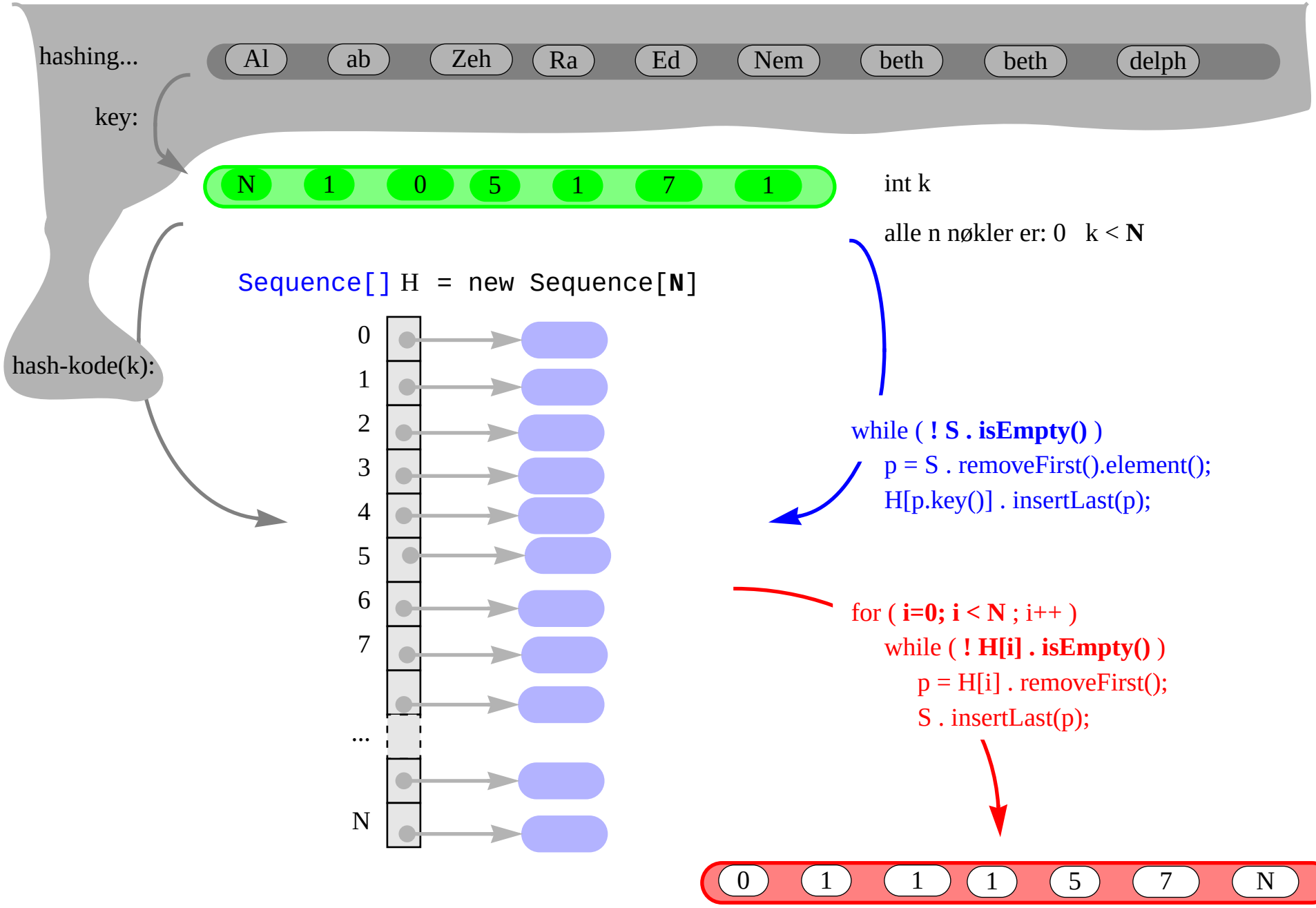
Bucket sort


hashing...
Al
ab
Zeh
Ra
Ed
Nem
beth
beth
delph
key:
N
1
0
5
1
7
1
int k
alle n nøkler er: 0 k < N
Sequence[] H = new Sequence[N]
hash-kode(k):
0
1
2
3
4
5
6
7
...
N
5
7
N
while (! S . isEmpty())
p = S . removeFirst().element();
H[p.key()] . insertLast(p);
for (i=0; i < N ; i++)
while (! H[i] . isEmpty())
p = H[i] . removeFirst();
S . insertLast(p);
0
1
1
1

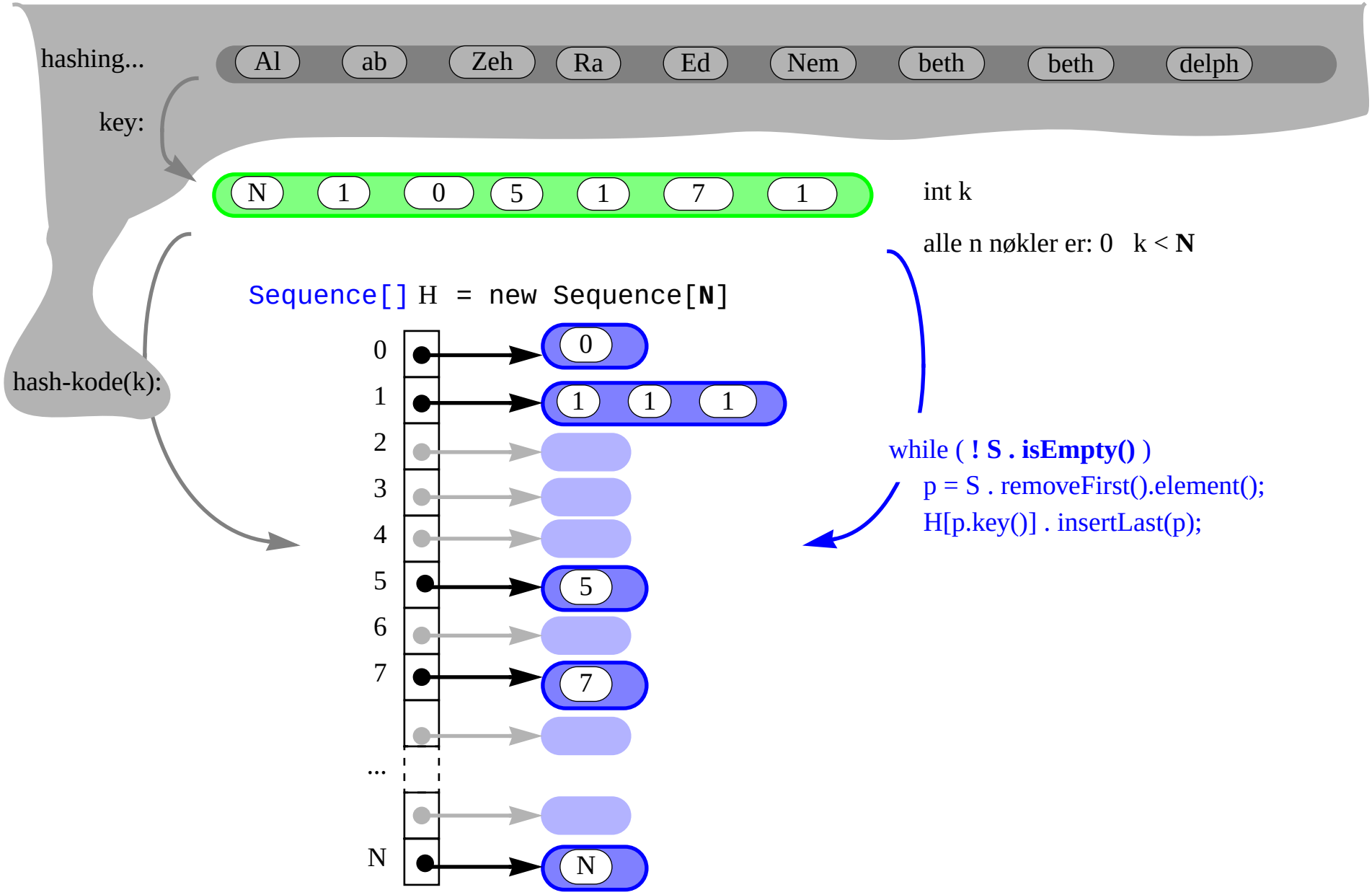
Bucket sort


hashing...
Al
ab
Zeh
Ra
Ed
Nem
beth
beth
delph
key:
N 1 0 5 1 7 1
int k
alle n nøkler er: 0 k < N
Sequence[] H = new Sequence[N]
hash-kode(k):
0
1
2
3
4
5
6
7
...
N
7
N
while (! S . isEmpty())
p = S . removeFirst().element();
H[p.key()] . insertLast(p);
for (i=0; i < N ; i++)
while (! H[i] . isEmpty())
p = H[i] . removeFirst();
S . insertLast(p);
0 1 1 1 5

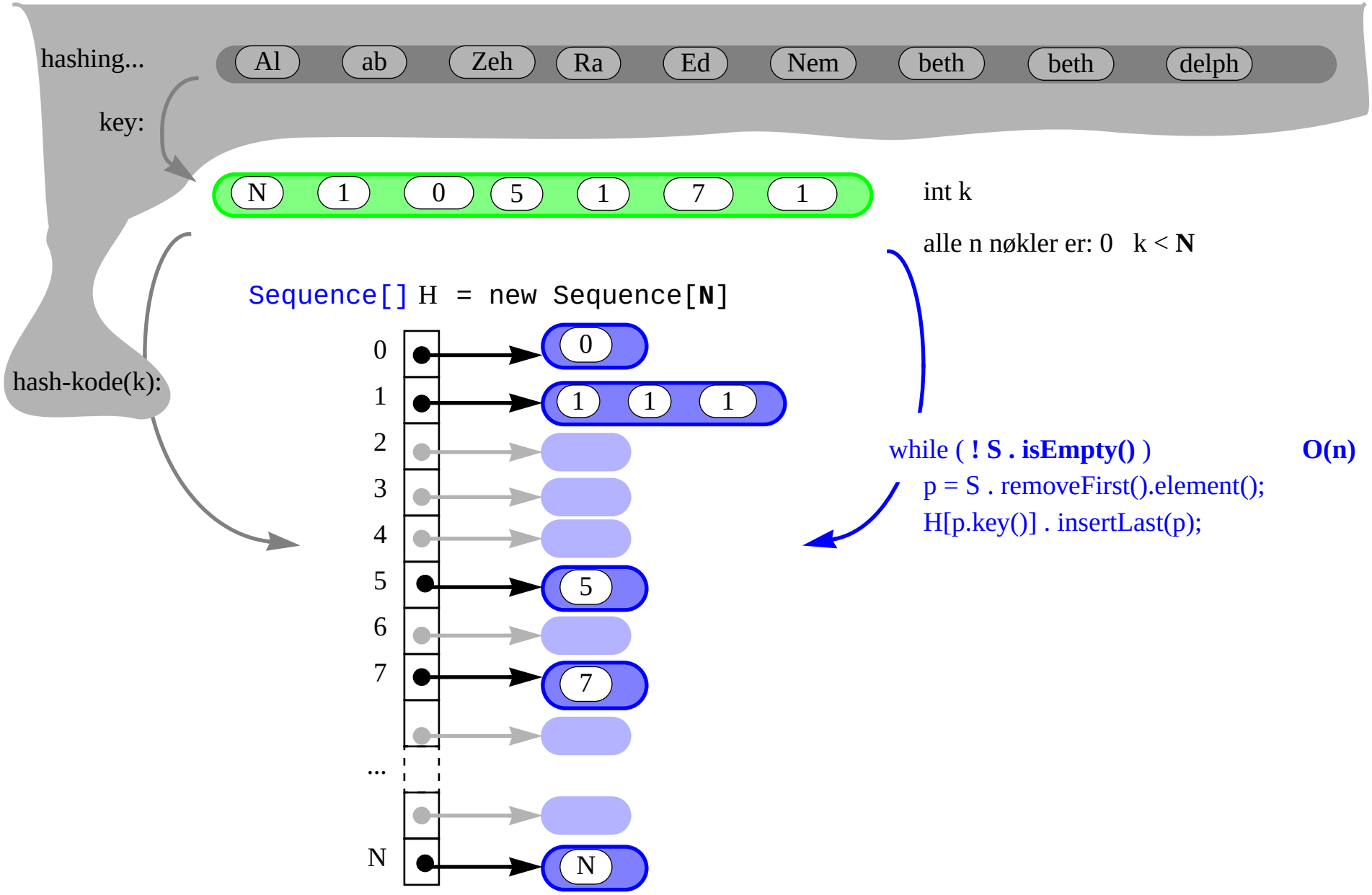
Bucket sort


hashing...
Al
ab
Zeh
Ra
Ed
Nem
beth
beth
delph
key:
N
1
0
5
1
7
1
int k
alle n nøkler er: 0 k < N
Sequence[] H = new Sequence[N]
hash-kode(k):
0
1
2
3
4
5
6
7
...
N
while (! S . isEmpty())
p = S . removeFirst().element();
H[p.key()] . insertLast(p);
for (i=0; i < N ; i++)
while (! H[i] . isEmpty())
p = H[i] . removeFirst();
S . insertLast(p);
0
1
1
1
5
7
N

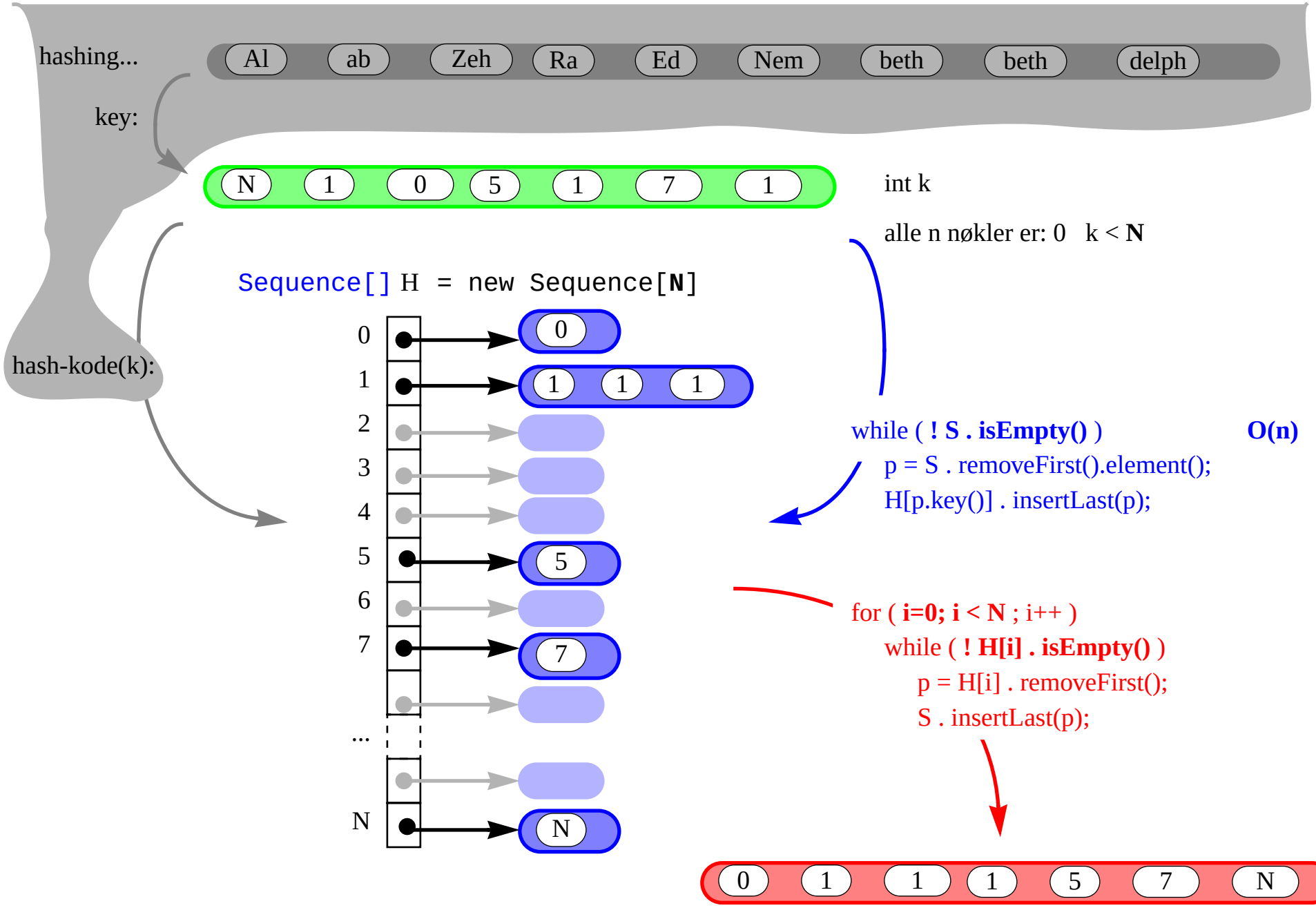
Bucket sort


hashing...
Al
ab
Zeh
Ra
Ed
Nem
beth
beth
delph
key:
N
1
0
5
1
7
1
int k
alle n nøkler er: 0 k < N
Sequence[] H = new Sequence[N]
hash-kode(k):
0
1
2
3
4
5
6
7
...
N
0
1
1
1
5
7
N
while (! S . isEmpty())
p = S . removeFirst().element();
H[p.key()] . insertLast(p);

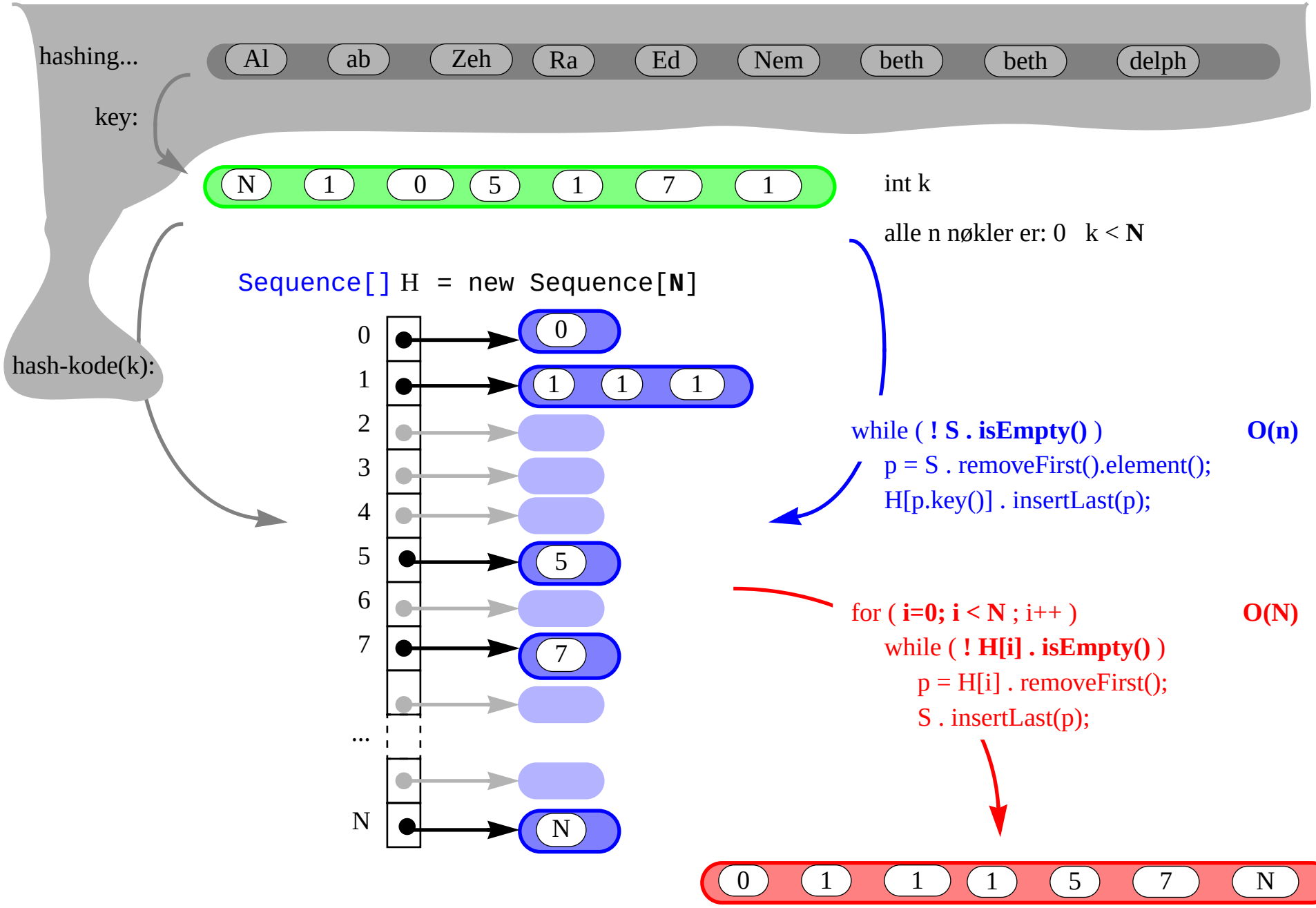
Bucket sort


hashing...
Al
ab
Zeh
Ra
Ed
Nem
beth
beth
delph
key:
N
1
0
5
1
7
1
int k
alle n nøkler er: 0 k < N
Sequence[] H = new Sequence[N]
hash-kode(k):
0
1
2
3
4
5
6
7
...
N
0
1
1
1
5
7
N
while (! S . isEmpty()) O(n)
p = S . removeFirst().element();
H[p.key()] . insertLast(p);

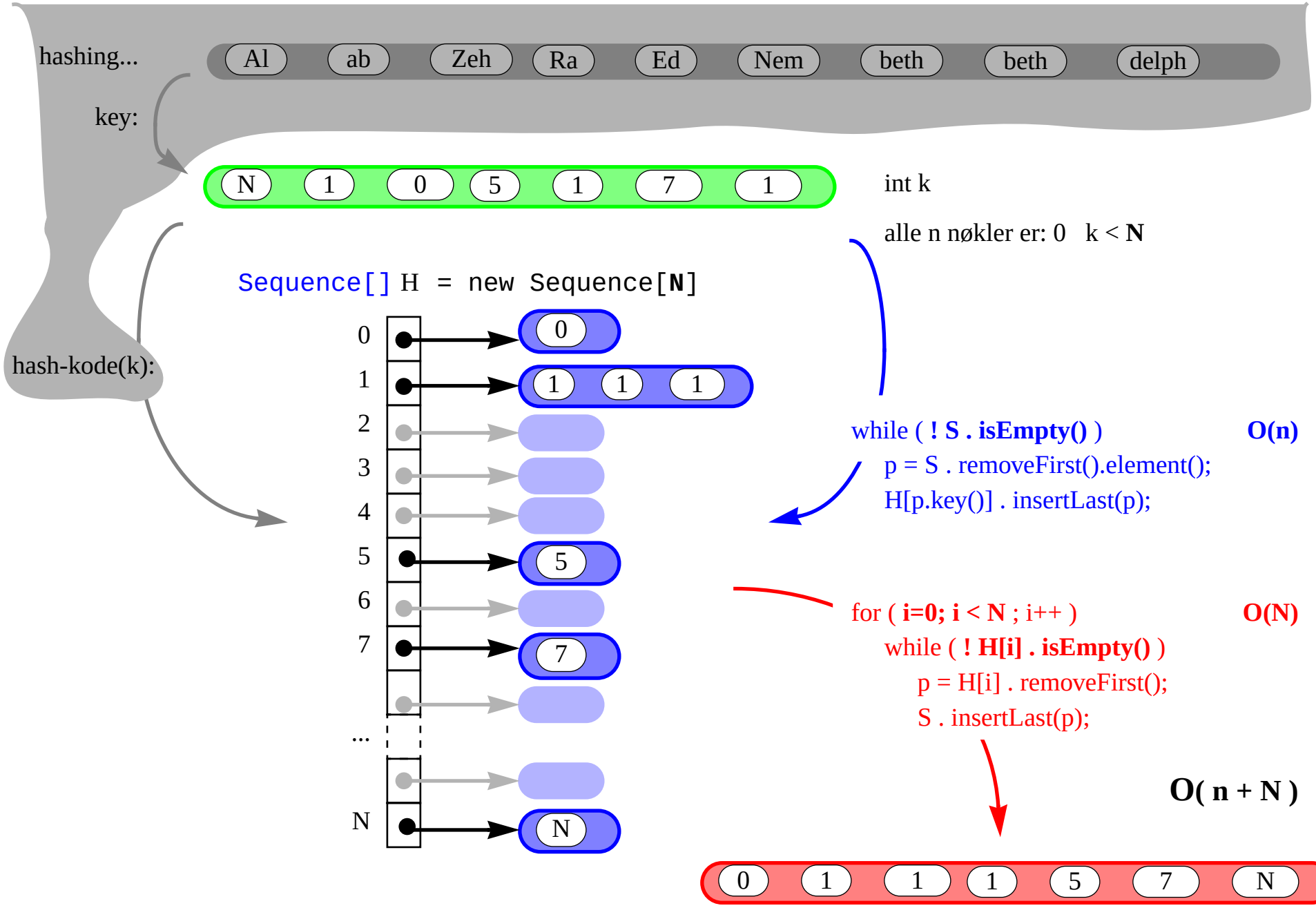
Bucket sort


hashing...
Al
ab
Zeh
Ra
Ed
Nem
beth
beth
delph
key:
N
1
0
5
1
7
1
int k
alle n nøkler er: 0 k < N
Sequence[] H = new Sequence[N]
hash-kode(k):
0
1
2
3
4
5
6
7
...
N
while (! S . isEmpty()) O(n)
p = S . removeFirst().element();
H[p.key()] . insertLast(p);
for (i=0; i < N ; i++)
while (! H[i] . isEmpty())
p = H[i] . removeFirst();
S . insertLast(p);
0
1
1
1
5
7
N

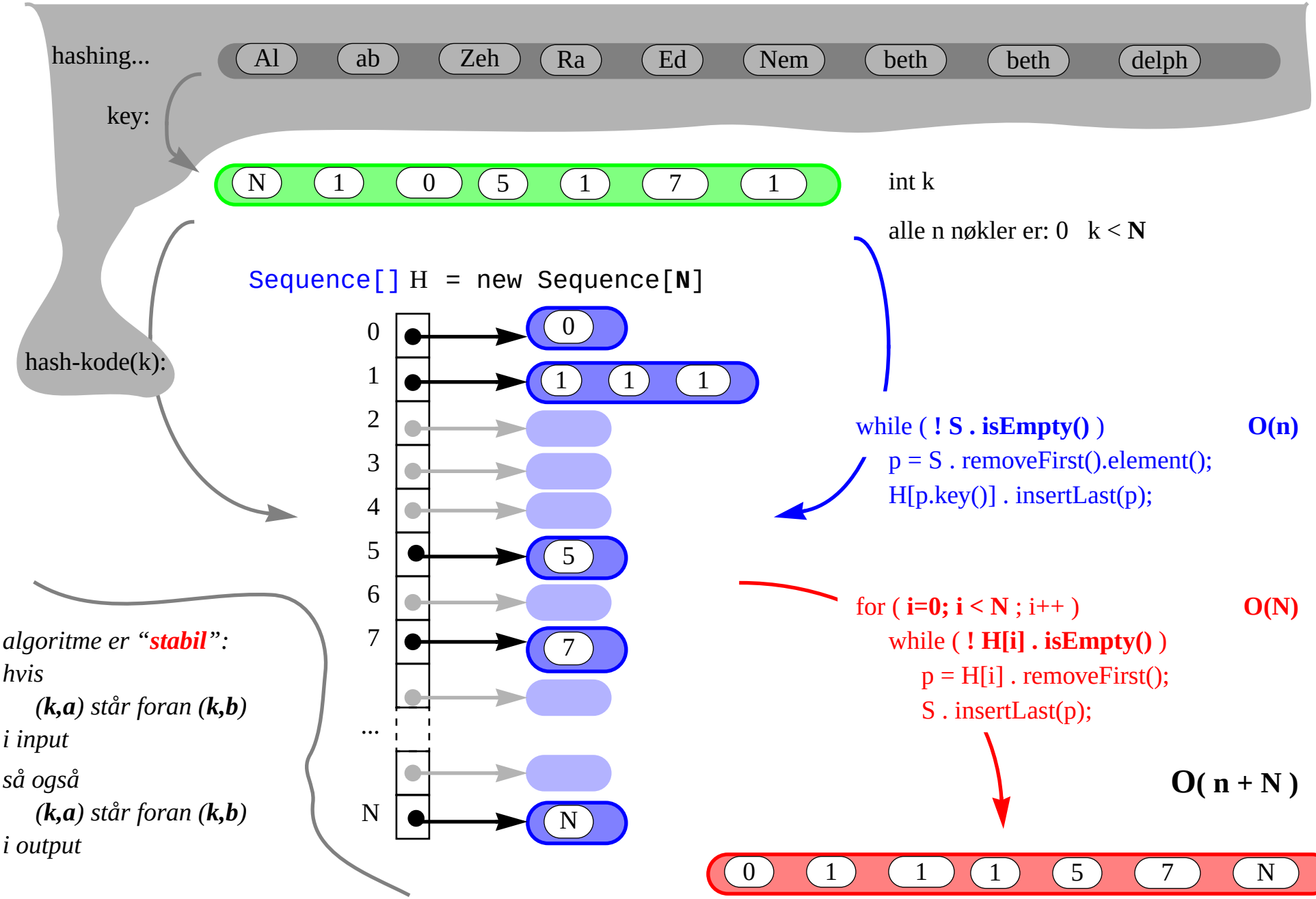
Bucket sort


hashing...
Al
ab
Zeh
Ra
Ed
Nem
beth
beth
delph
key:
N
1
0
5
1
7
1
int k
alle n nøkler er: 0 k < N
Sequence[] H = new Sequence[N]
hash-kode(k):
0
1
2
3
4
5
6
7
...
N
while (! S . isEmpty()) O(n)
p = S . removeFirst().element();
H[p.key()] . insertLast(p);
for (i=0; i < N ; i++) O(N)
while (! H[i] . isEmpty())
p = H[i] . removeFirst();
S . insertLast(p);
0
1
1
1
5
7
N

Bucket sort


hashing...
Al
ab
Zeh
Ra
Ed
Nem
beth
beth
delph
key:
N
1
0
5
1
7
1
int k
alle n nøkler er: 0 k < N
Sequence[] H = new Sequence[N]
hash-kode(k):
0
1
2
3
4
5
6
7
...
N
while (! S . isEmpty()) O(n)
p = S . removeFirst().element();
H[p.key()] . insertLast(p);
for (i=0; i < N ; i++) O(N)
while (! H[i] . isEmpty())
p = H[i] . removeFirst();
S . insertLast(p);
O(n + N)
0
1
1
1
5
7
N

Bucket sort


hashing...
Al
ab
Zeh
Ra
Ed
Nem
beth
beth
delph
key:
N 1 0 5 1 7 1
int k
alle n nøkler er: 0 k < N
Sequence[] H = new Sequence[N]
hash-kode(k):
0
1
2
3
4
5
6
7
...
N
while (! S . isEmpty()) O(n)
p = S . removeFirst().element();
H[p.key()] . insertLast(p);
for (i=0; i < N ; i++) O(N)
while (! H[i] . isEmpty())
p = H[i] . removeFirst();
S . insertLast(p);
O(n + N)
0 1 1 1 5 7 N
algoritme er "stabil":
hvis
(k,a) står foran (k,b)
i input
så også
(k,a) står foran (k,b)
i output

Sortering

- Bubble $\Theta(n^2)$

prioritetskø basert

- Selection $\Theta(n^2)$
- Insertion $O(n^2)$
- Heap $O(n \log n)$

‘‘splitt og hersk’’

- Merge $O(n \log n)$
- Quick $O(n \log n)$

Sortering

- Bubble $\Theta(n^2)$

prioritetskø basert

- Selection $\Theta(n^2)$
- Insertion $O(n^2)$
- Heap $O(n \log n)$

‘‘splitt og hersk’’

- Merge $O(n \log n)$
- Quick $O(n \log n)$

‘‘Comparison based sorting’’ = $\Omega(n \log n)$

Sortering

- Bubble $\Theta(n^2)$

prioritetskø basert

- Selection $\Theta(n^2)$
- Insertion $O(n^2)$
- Heap $O(n \log n)$

‘‘splitt og hersk’’

- Merge $O(n \log n)$
- Quick $O(n \log n)$

‘‘Comparison based sorting’’ = $\Omega(n \log n)$

- Bucket $O(n + N)$
- Radix

Sortering

- Bubble $\Theta(n^2)$

prioritetskø basert

- Selection $\Theta(n^2)$
- Insertion $O(n^2)$
- Heap $O(n \log n)$

‘‘splitt og hersk’’

- Merge $O(n \log n)$
- Quick $O(n \log n)$

‘‘Comparison based sorting’’ = $\Omega(n \log n)$

- Bucket $O(n + N)$
- Radix

- “in place”, stabil