Crochemore's algorithm for repetitions revisited - computing runs

F. Franek, M. Jiang
Computing and Software
McMaster University
Hamilton, Ontario

Israel Stringology Conference
Bar-Ilan University, Tel-Aviv
March-April 2009

- Why we are interested in Crochemore's repetition algorithm
- A brief description of our implementation of Crochemore's algorithm.
- A simple modification of Crochemore's algorithm to compute runs (worsening the complexity to $O(n \log^2(n))$
- A modification of Crochemore's algorithm to compute runs while preserving the complexity $O(n \log(n))$
- Conclusion

Why we are interested in Crochemore’s repetition algorithm

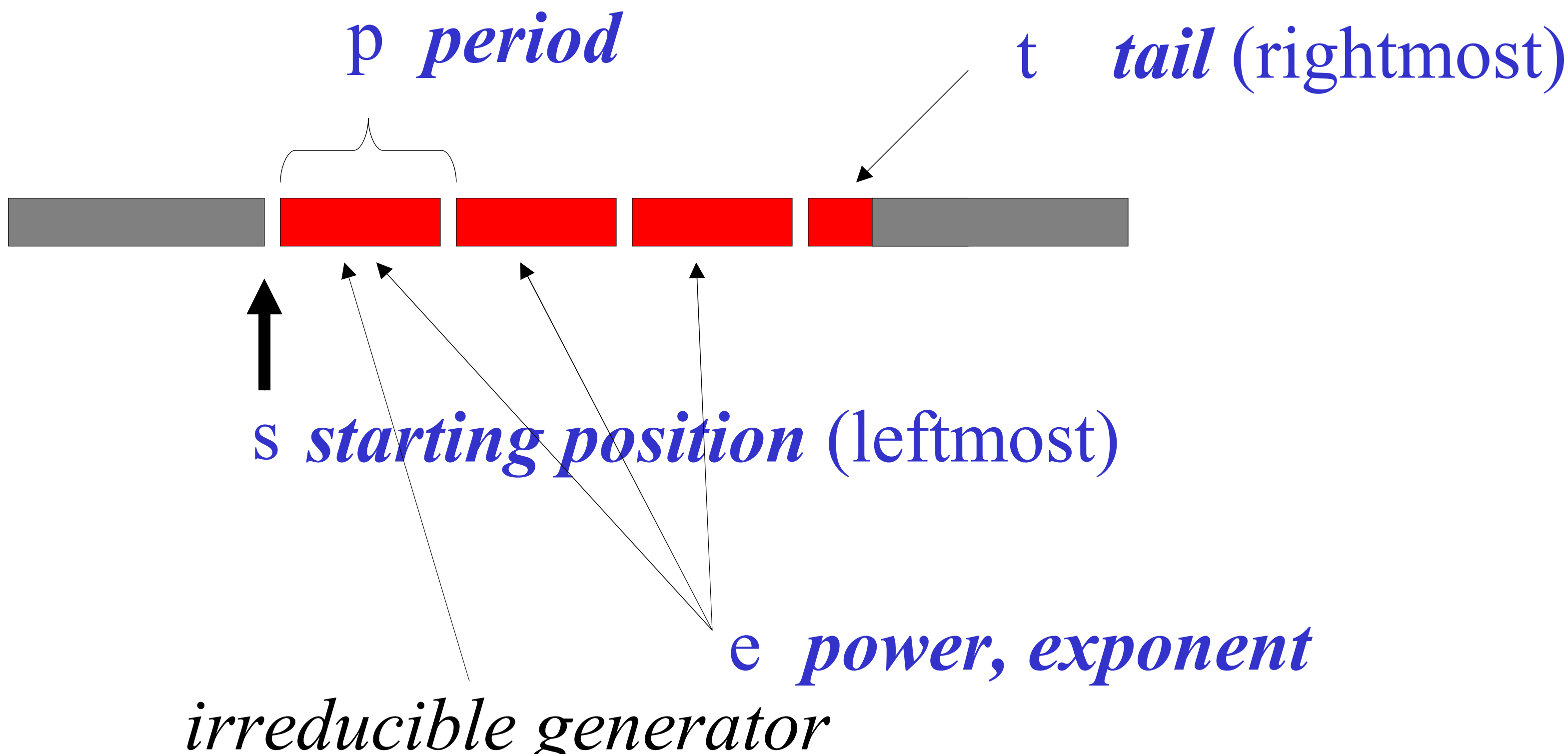
A **run** captures the notion of a maximal non-extendible repetition in a string ***x***

(s,p,e,t)

Alternative: (s,p,end)

e = (end - s + 1) / p

t = (end - s + 1) % p



Computing runs in linear time

Main (1989) introduced runs and gave the following algorithm to compute the leftmost occurrence of every run of a string ***x***:

(1) Compute a suffix tree for ***x*** (*linear, using Farach's algorithm*)

(2) using the suffix tree, compute Lempel-Ziv factorization of ***x*** (*linear, Lempel-Ziv)*

(3) using the Lempel-Ziv factorization, compute the leftmost runs (*linear, Main*)

Lempel-Ziv factorization can be computed in linear time using suffix array (Abouelhoda, Kurtz, & Ohlebusch *2004*)

Suffix array can be computed in linear time (Kärkkäinen, Sanders *2003*, Ko, Aluru *2003*)

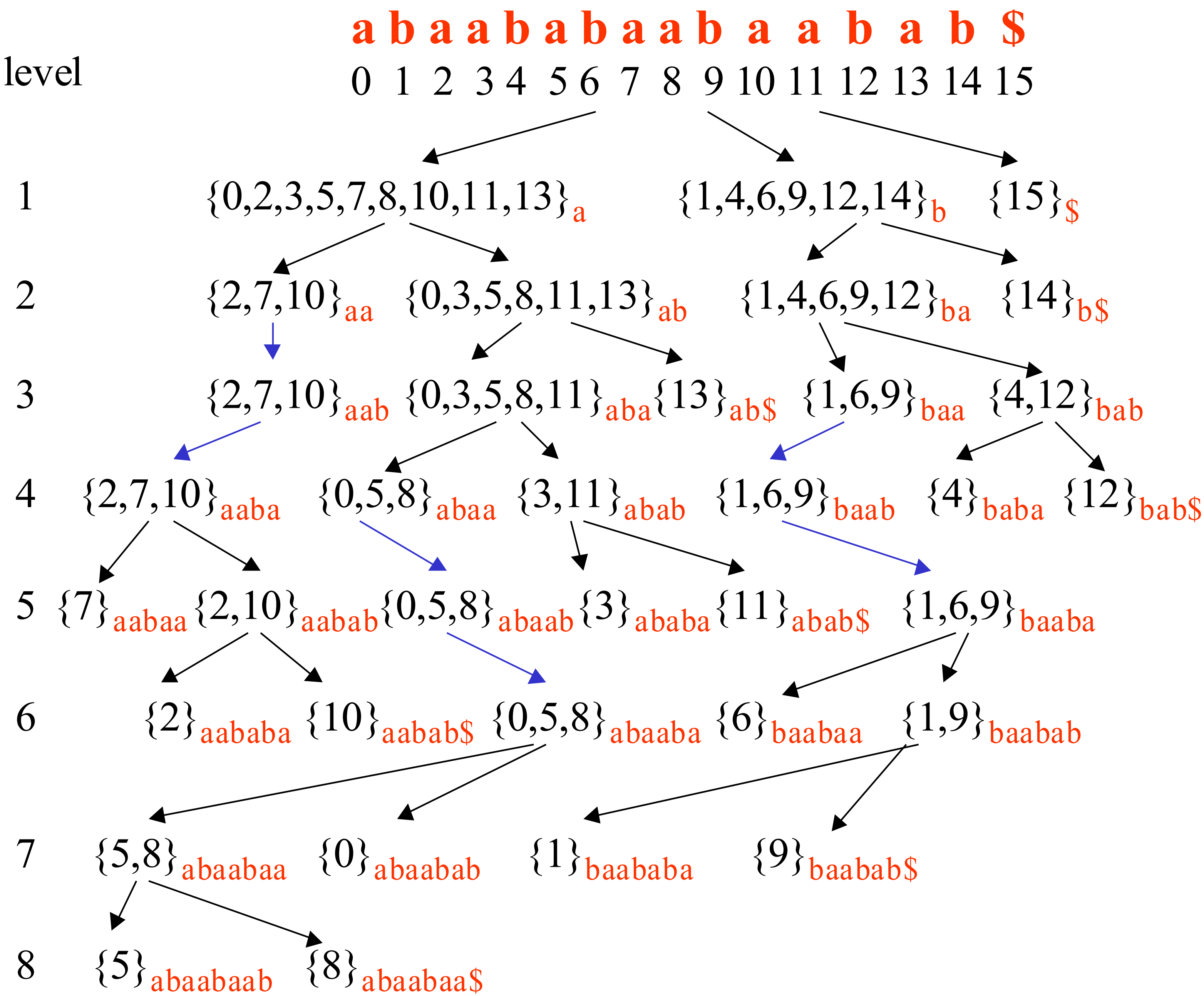
Chen, Puglisi, & Smyth *2007*, using suffix array and the lcp array (lcp can be computed from suffix array in linear time, Kasai *et al 2001*):) it computes Lempel-Ziv factorization in linear time using Ukkonen's on-line approach.

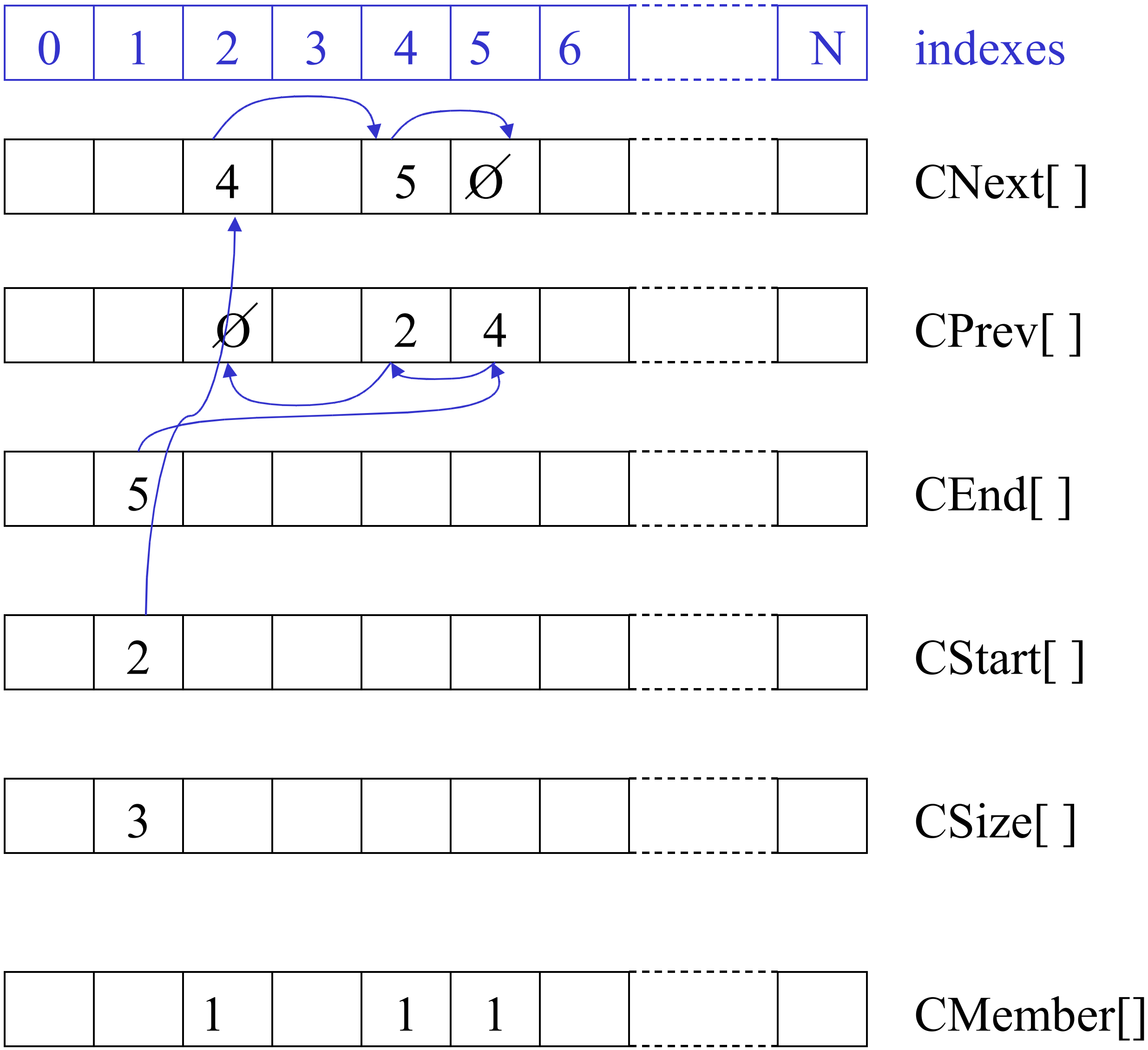
All these approaches are complicated and elaborate, and the implementations into code are not readily available.

Also, they do not lend themselves well to parallelization (see slide 9 -- the refinement of the classes can be done naturally in parallel as the refinement of one class is independent from the refinement of another class.)

We have a good and “space efficient” implementation of Crochemore’s algorithm, that naturally lends itself to parallelization.

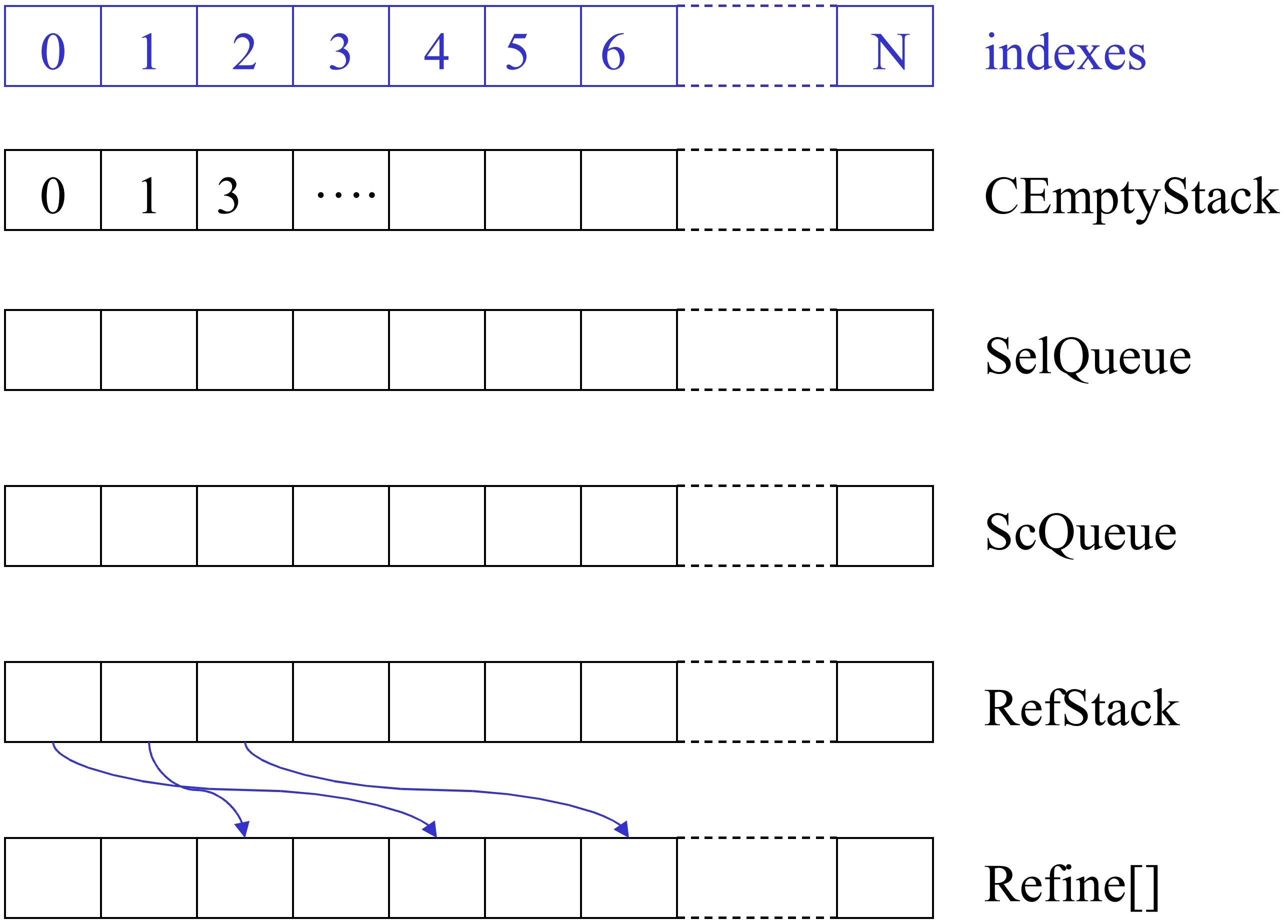
A brief description of our implementation of Crochemore's algorithm


a b a a b a b a a b a a b a b $
level
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
1
{0,2,3,5,7,8,10,11,13}a
{1,4,6,9,12,14}b
{15}$
2
{2,7,10}aa
{0,3,5,8,11,13}ab
{1,4,6,9,12}ba
{14}b$
3
{2,7,10}aab
{0,3,5,8,11}aba
{13}ab$
{1,6,9}baa
{4,12}bab
4
{2,7,10}aaba
{0,5,8}abaa
{3,11}abab
{1,6,9}baab
{4}baba
{12}bab$
5
{7}aabaa
{2,10}aabab
{0,5,8}abaab
{3}ababa
{11}abab$
{1,6,9}baaba
6
{2}aababa
{10}aabab$
{0,5,8}abaaba
{6}baabaa
{1,9}baabab
7
{5,8}abaabaa
{0}abaabab
{1}baababa
{9}baabab$
8
{5}abaabaab
{8}abaabaa$

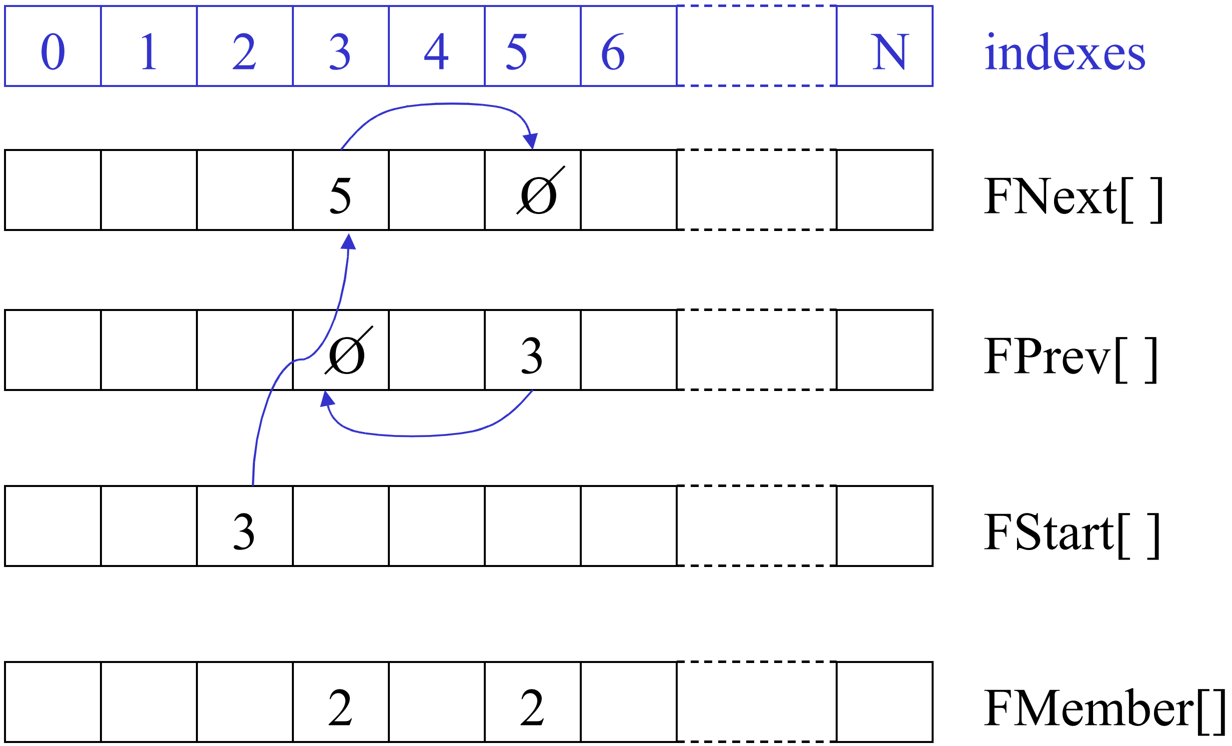


$c_1=\{2,4,5\}$

Total this slide 6*N
subtotal 6*N

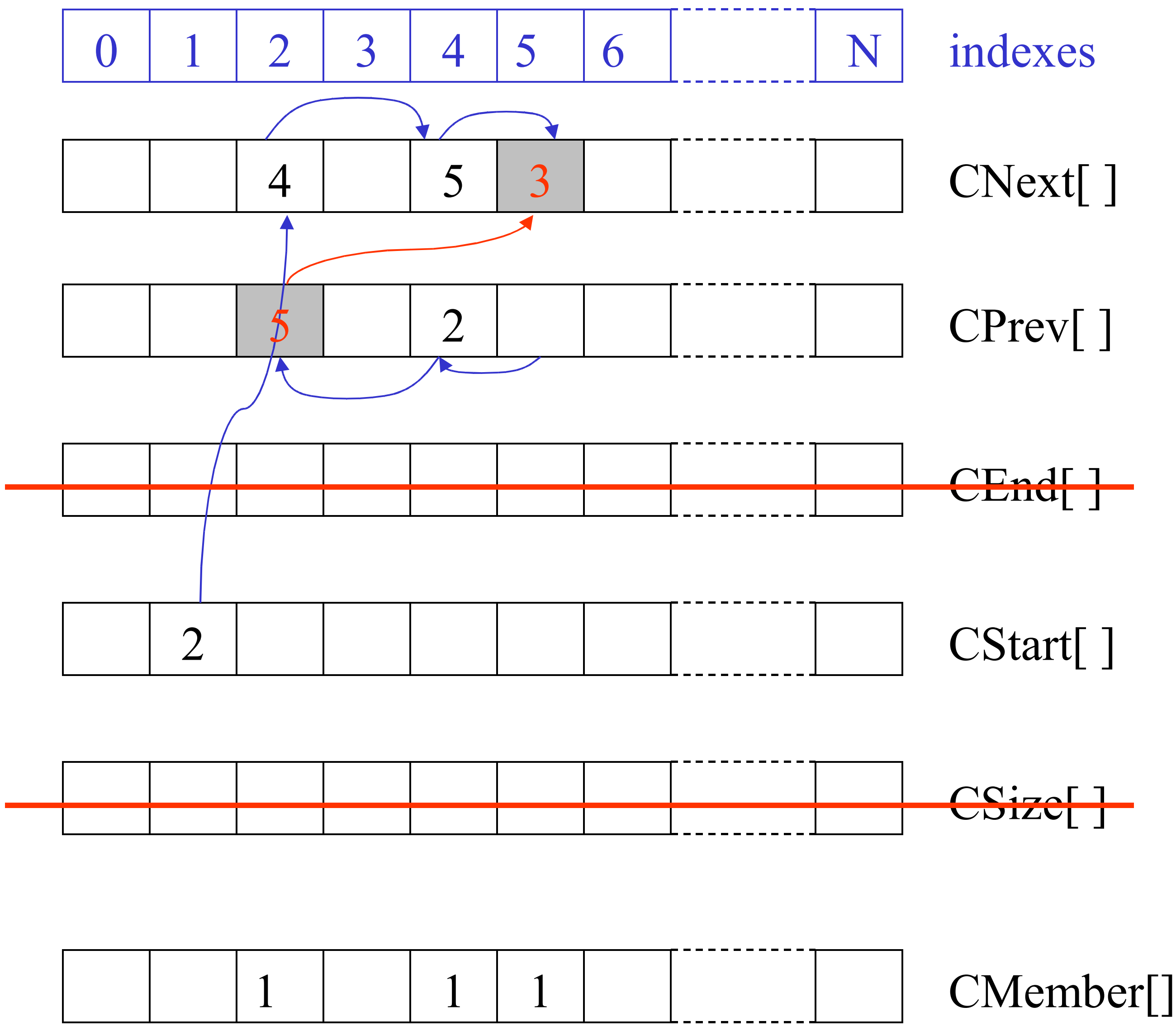


Total this slide 5*N
subtotal 11*N



$f_2=\{3,5\}$

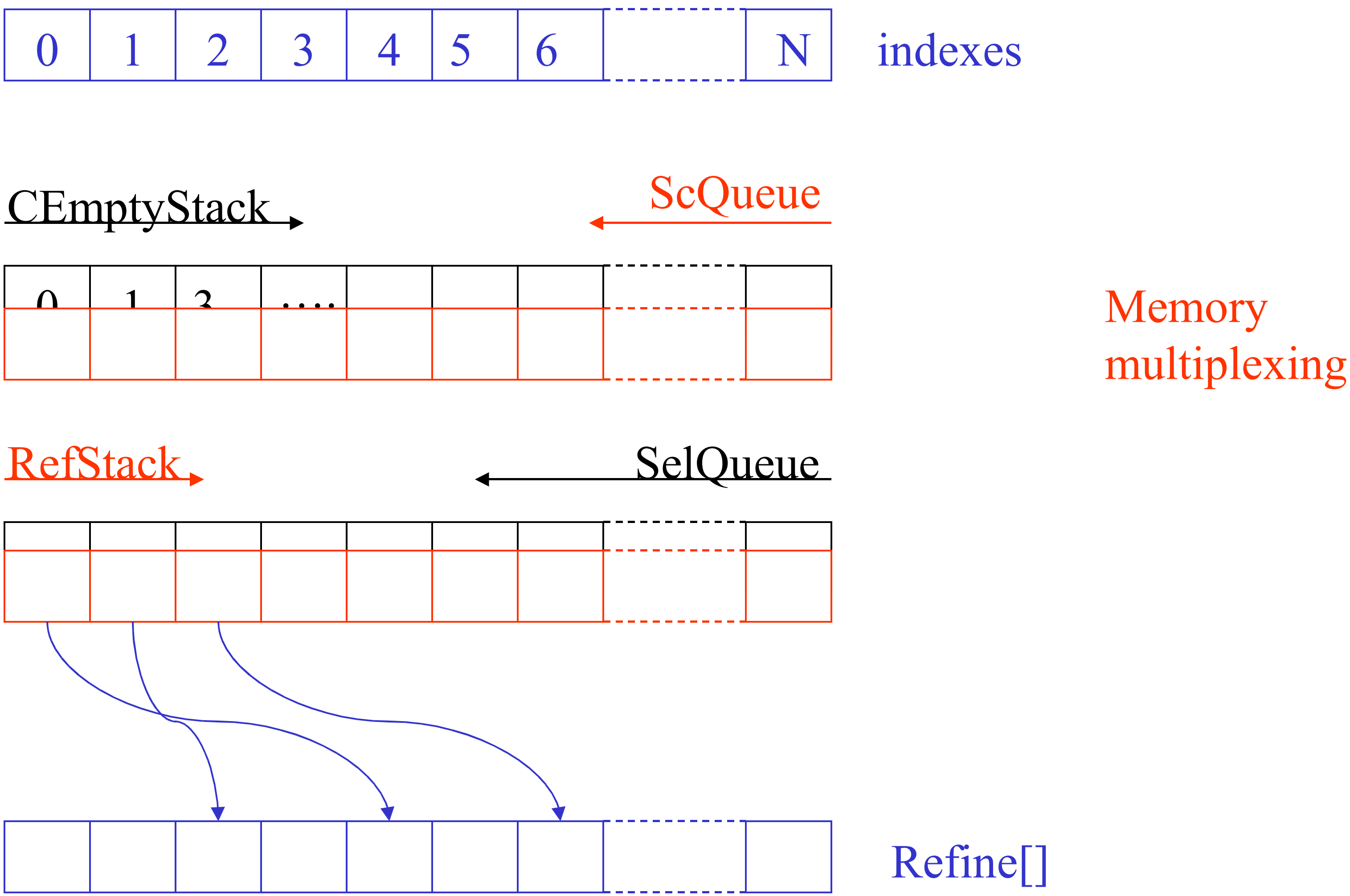
Total this slide 4*N
overall total 15*N



$c_1=\{2,4,5\}$

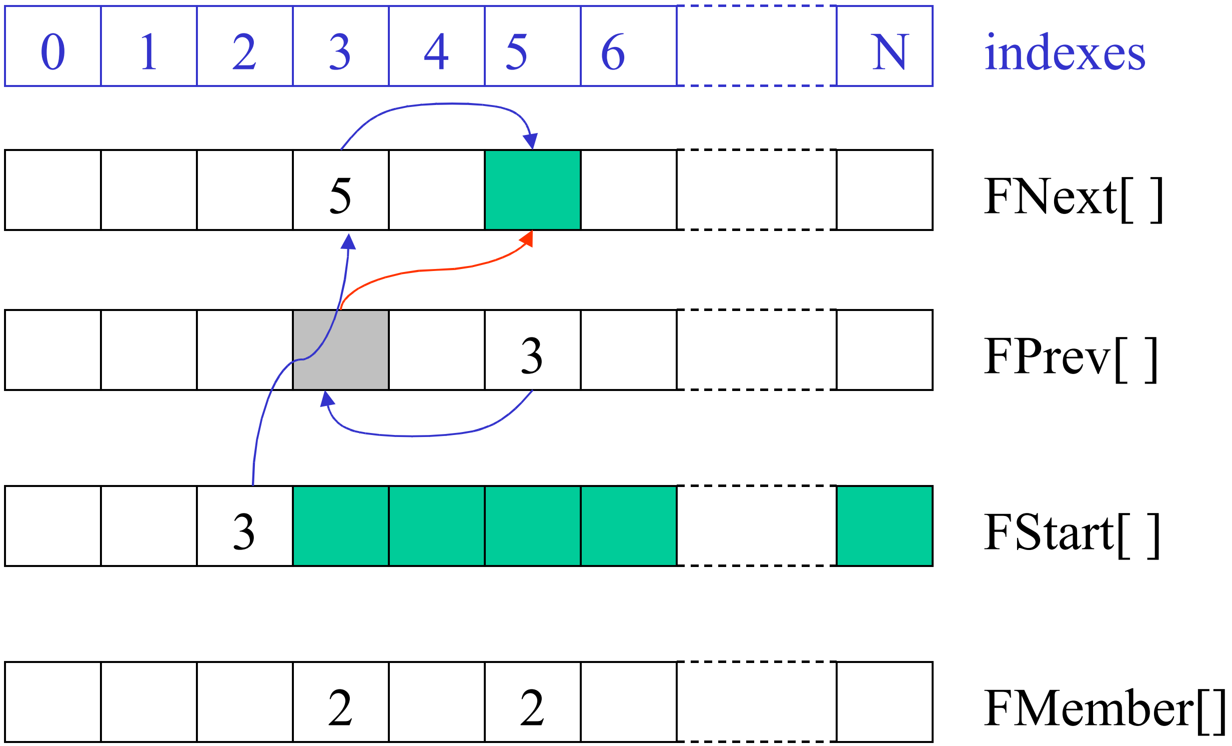
Memory virtualization

Total this slide 4*N
subtotal 4*N



Refine[] is virtualized over FNext[], FPrev[], and FStart[]

Total this slide 2*N
subtotal 6*N

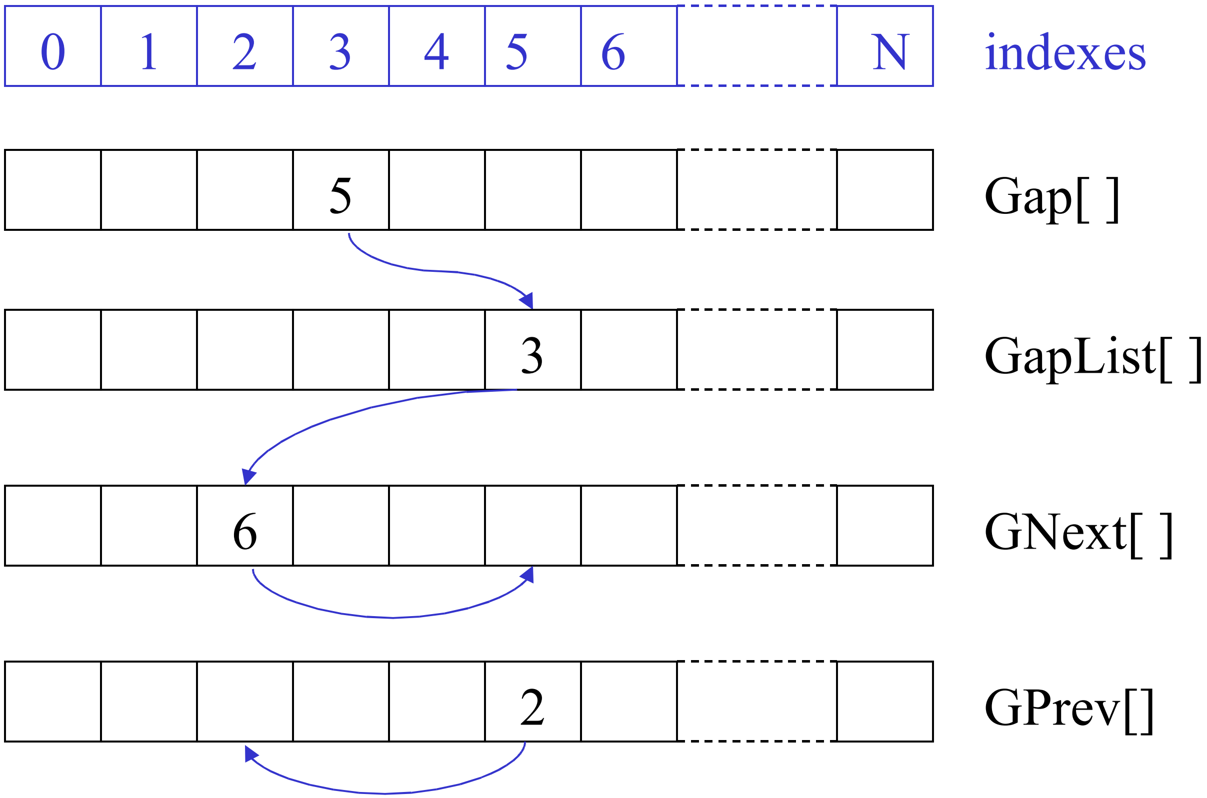


$f_2=\{3,5\}$

Memory virtualization

Refine[] is virtualized over

Total this slide 4*N
overall total 10*N



Total this slide 4*N
overall total 14*N

Though the repetitions are reported level by level, they are not reported in any appreciable order (caused by the manipulations of GapList)

a b a a b a b a a b a a b a b $
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

(10,1,2) a b a a b a b a a b a a b a b $
(7,1,2) a b a a b a b a a b a a b a b $
(2,1,2) a b a a b a b a a b a a b a b $

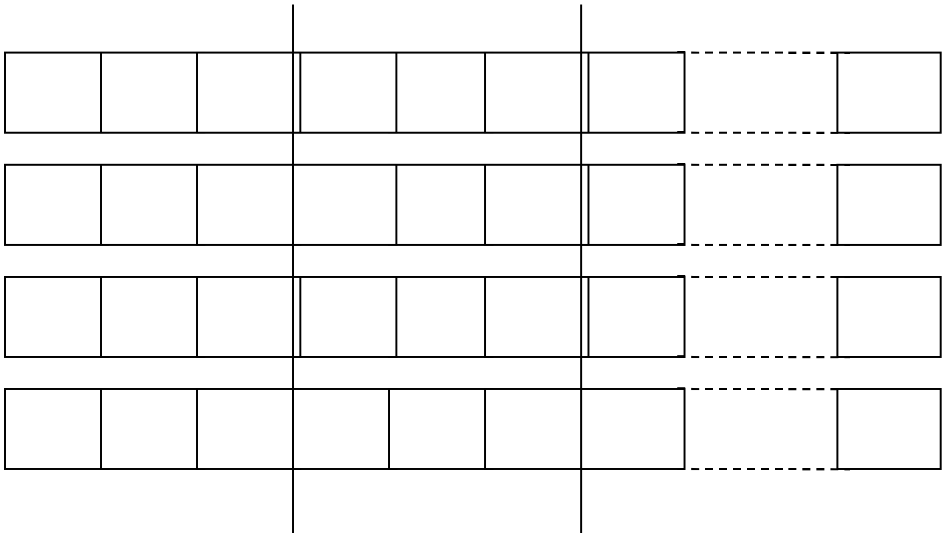
(11,2,2) a b a a b a b a a b a a b a b $
(3,2,2) a b a a b a b a a b a a b a b $
(4,2,2) a b a a b a b a a b a a b a b $
run

(6,3,2) a b a a b a b a a b a a b a b $
(5,3,3) a b a a b a b a a b a a b a b $
(0,3,2) a b a a b a b a a b a a b a b $
(7,3,2) a b a a b a b a a b a a b a b $
run

(0,5,2) a b a a b a b a a b a a b a b $
(1,5,2) a b a a b a b a a b a a b a b $
run

A simple modification of Crochemore's algorithm to compute runs (worsening the complexity to $O(n \log^2(n))$

We have to collect repetitions and “join” them into runs.
Collecting, “joining”, and reporting level by level, basically in a binary search tree:



RunLeft[ ] (reuse FNext[ ])

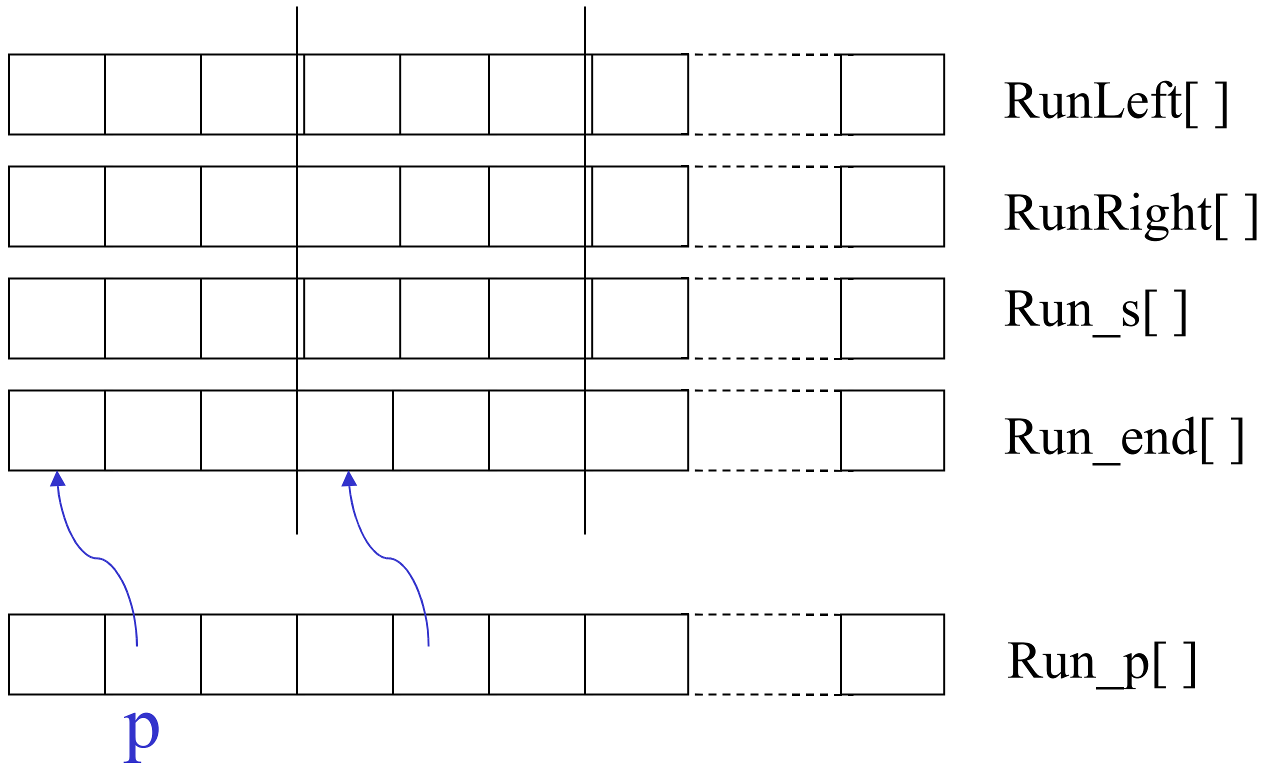
RunRight[ ] (reuse FPrev[ ])

Run_s[ ] (reuse FMember[ ])

Run_end[ ] (reuse FStart[ ])

Complexity: need $O(\log(n))$ for each repetition to place it in the tree, overall $O(n \log^2(n))$

Collecting and "joining" in a binary search tree, reporting at the end: the same complexity $O(n \log^2(n))$, memory requirement increased by 5*N



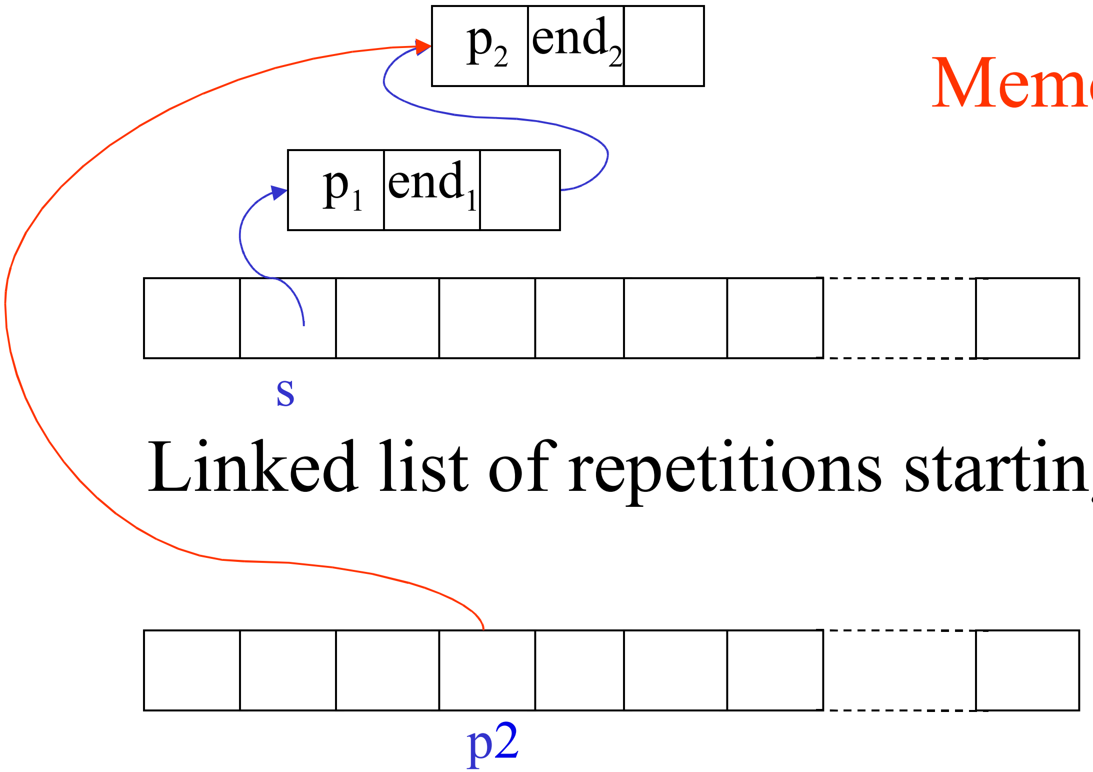
Total this slide 5*N
overall total 19*N

Points to the "root" of the search tree for runs of period p.

A modification of Crochemore's algorithm to compute runs while preserving the complexity $O(n \log(n))$

Collecting into buckets, "joining" and reporting at the end.

Memory: ? O(n log(n))



Linked list of repetitions starting at s

points to the last run with period p2, so we know with what to join the incoming repetition with (if at all), as we sweep from left to right.

Complexity: $O(n \log(n))$
Memory: $15*N + O(n \log(n))$
To avoid dynamic allocation of memory, we are using allocation from arena technique.

Conclusion

- Crochemore's algorithm is fast, though memory demanding
- Our implementation is as memory efficient as possible
- Great potential for parallel implementation
- Preliminary test very positive
- Further research
  (1) to compare performance with linear time algorithms (problem - lack of code)
  (2) to implement parallel version with little communication overhead

Thank You!