A note on Crochemore's repetitions algorithm, a fast space-efficient approach

F. Franek, W.F. Smyth, and X. Xia

Algorithms Research Group
Computing and Software
McMaster University
Hamilton, Ontario, Canada

PSC 2002, Prague Stringology Conference,
Prague, Czech Republic,
September 22-23, 2002

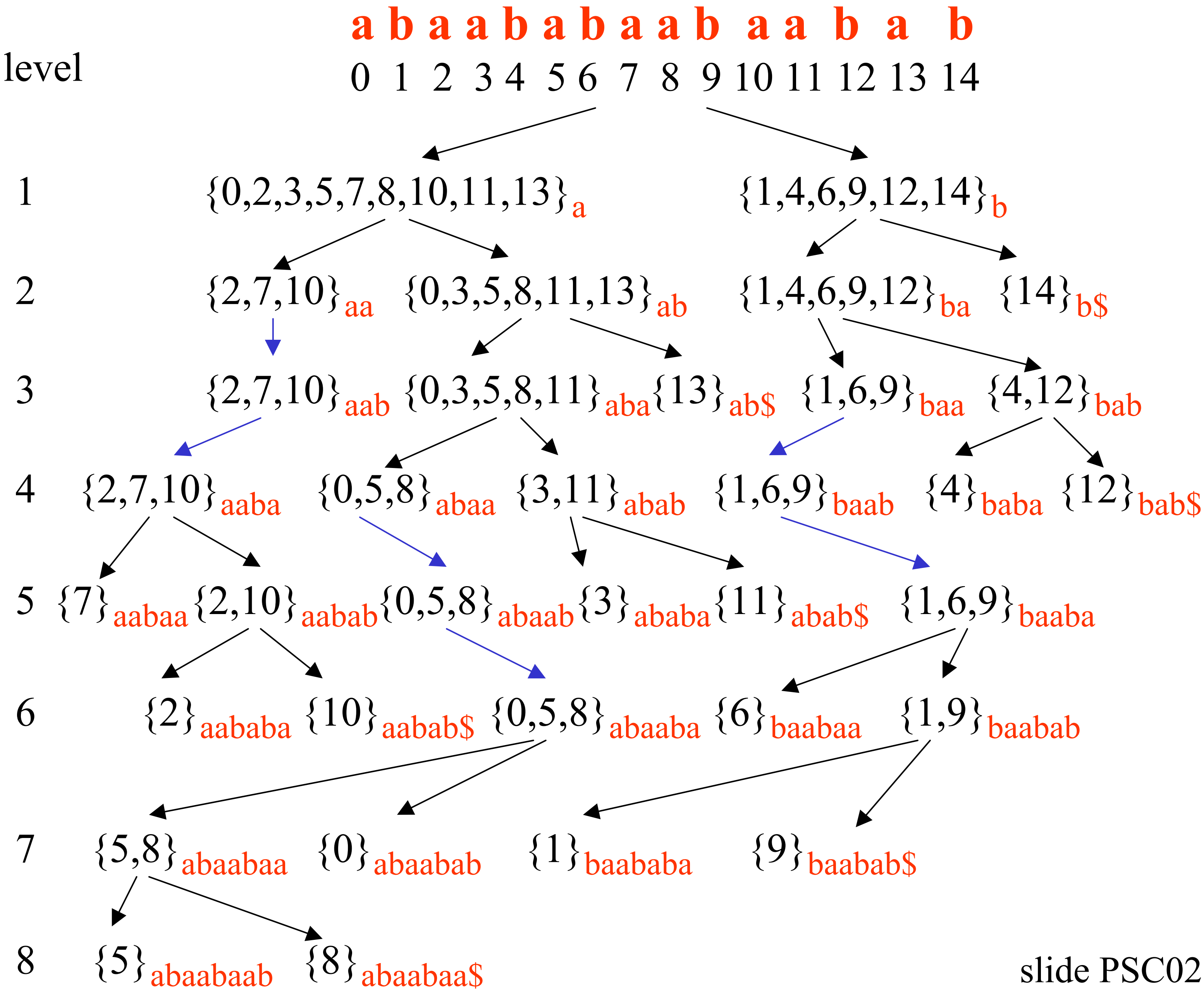
```
for(i = 0; i < N-2; i++) {
  for(k = 1;  k <= (N-i)/2; k++) {
    s = 1;
    for(j = 0; j < k; j++)
      if (x[i+j] != x[i+k+j]) {s=0; break; }
    if (s) printf("square of length %d at position %d\n",k,i);
  }
}
```

Trivial, brute force $O(n^3)$ algorithm for computing of all squares.

Crochemore (1981) designed the first O(n log n) algorithms to compute all the repetitions in a string.

One of the main ideas of the approach concerns successive refinements of classes of equivalence of indices (positions) of the input string.

Two positions on level p are equivalent, if two identical substrings of length p start there.


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

If we do the refinement in a brute force fashion again, immediately we have an $O(n^2)$ algorithm for computing of all squares (in fact it can be shown that the average-case complexity is O(n log n)).

The other main idea of Crochemore was to do the refinement using other classes and all of them, only the so-called small classes, which brings the worst-case complexity to O(n log n).

Let us remark that Crochemore's algorithm can be used for more than just repetitions, in fact it can be used to compute a suffix tree of the input string, a much stronger "description" of the structure of the string than the repetitions in the form of runs.

It is generally believed and all known implementations of Crochemore's algorithm needed about 20NM bytes of extra memory to work.

Since 1981 several linear (for fixed alphabet) or O(n log n) algorithms for suffix trees have been presented, in particular Ukkonen (1992), and for suffix arrays Manber, Myers (1993), all with small memory requirements.

So, why still bother with Crochemore's algorithm?

•Ease of implementation

•Implementations faster in reality

In this talk, we present a novel implementation of Crochemore's algorithm that requires about half as much memory as the standard ones: 10NM bytes, where N is the length of the input string and M is the size in bytes of the integer N.

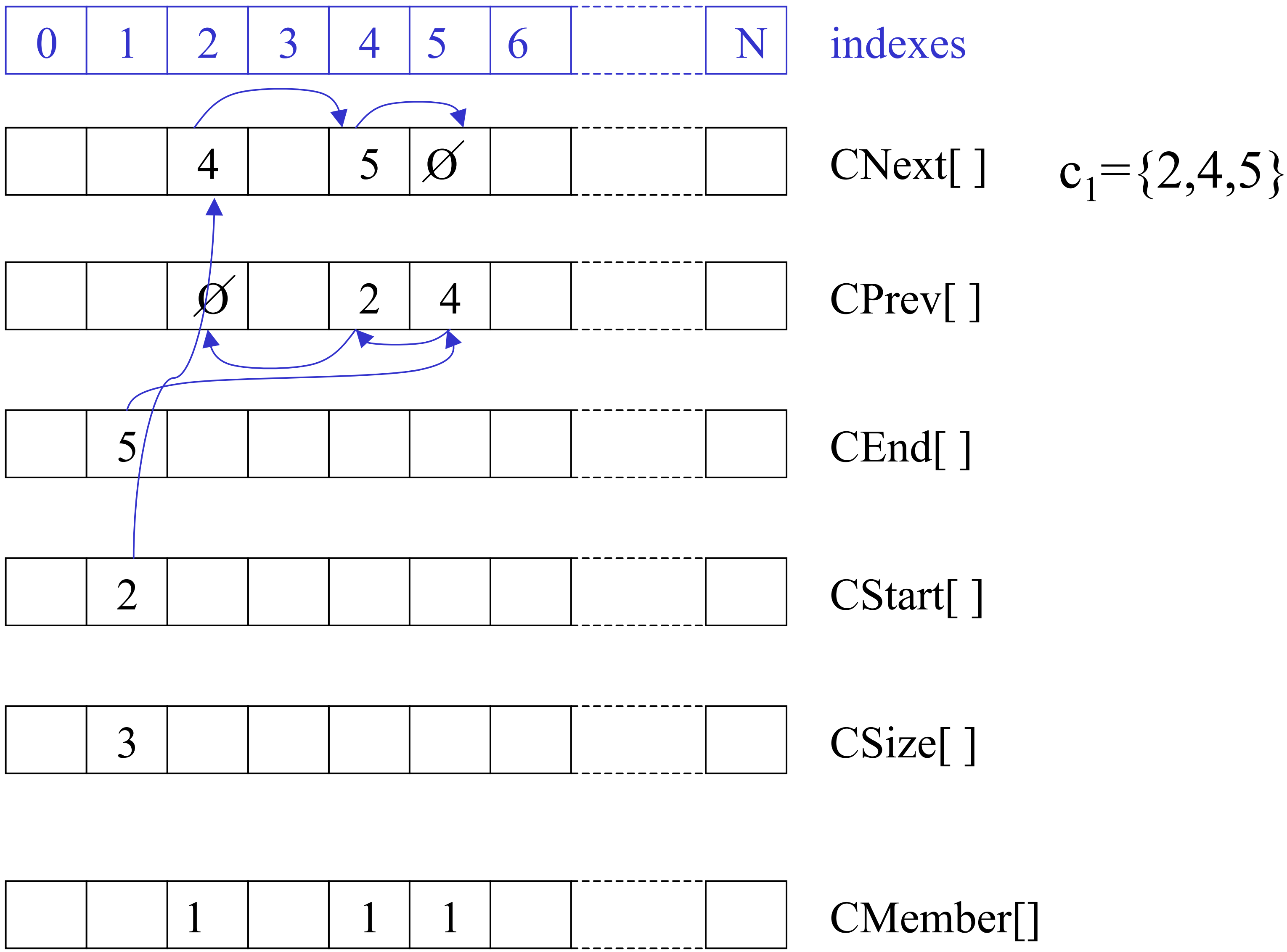
We present the implementation in two steps as it facilitates a better understanding.

•First we present a version requiring 15NM, the decrease being a result of a smarter handling of data structures representing the classes

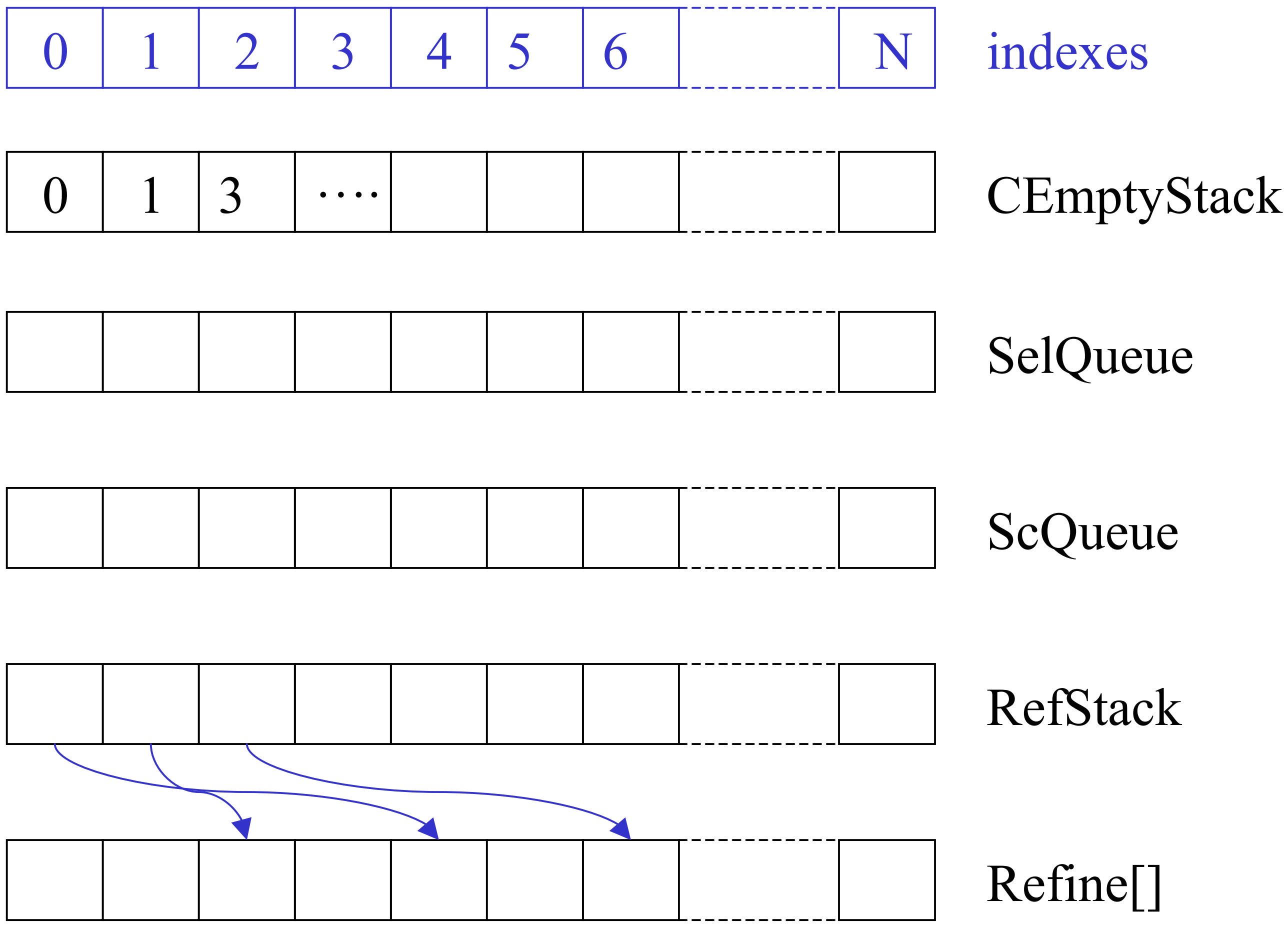
•Then we use some “tricks” to bring the memory requirements down to 10NM.

Of course, this requires a certain overhead, slowing down the execution.

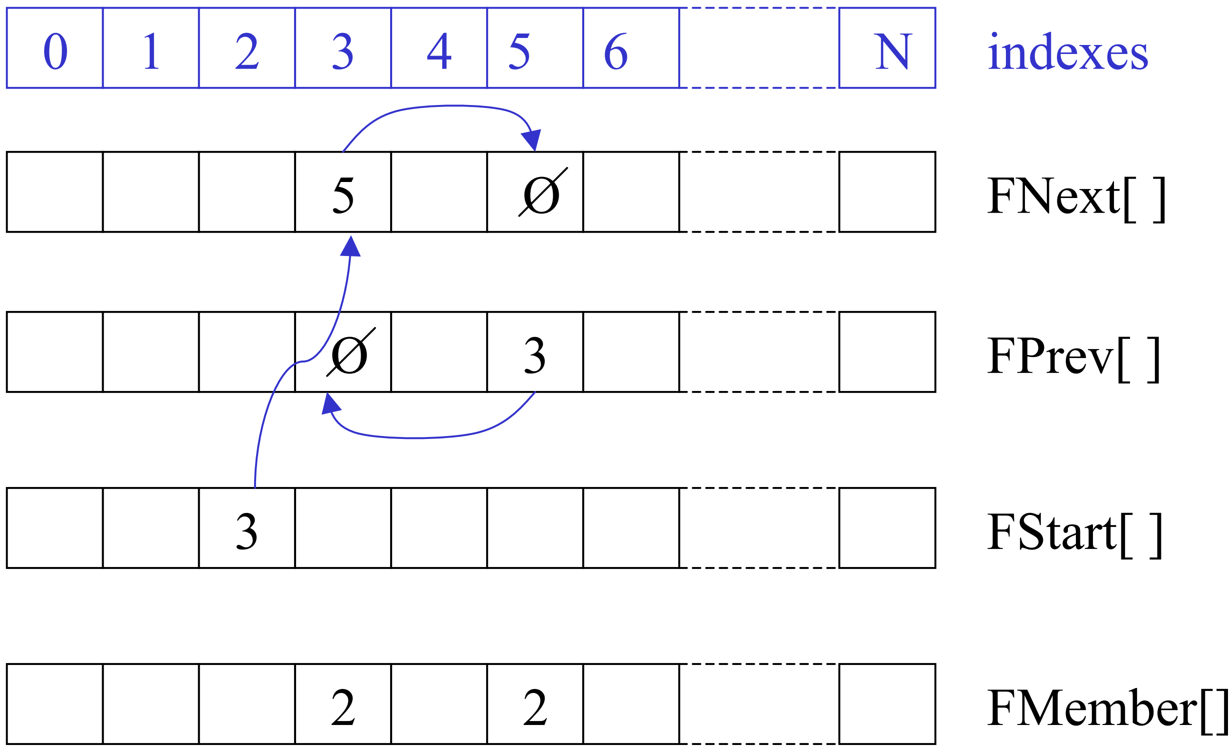
The experiments carried out by the third co-author Xia indicate a 20-30% slowdown.



Total this slide 6*N
subtotal 6*N



Total this slide 5*N
subtotal 11*N

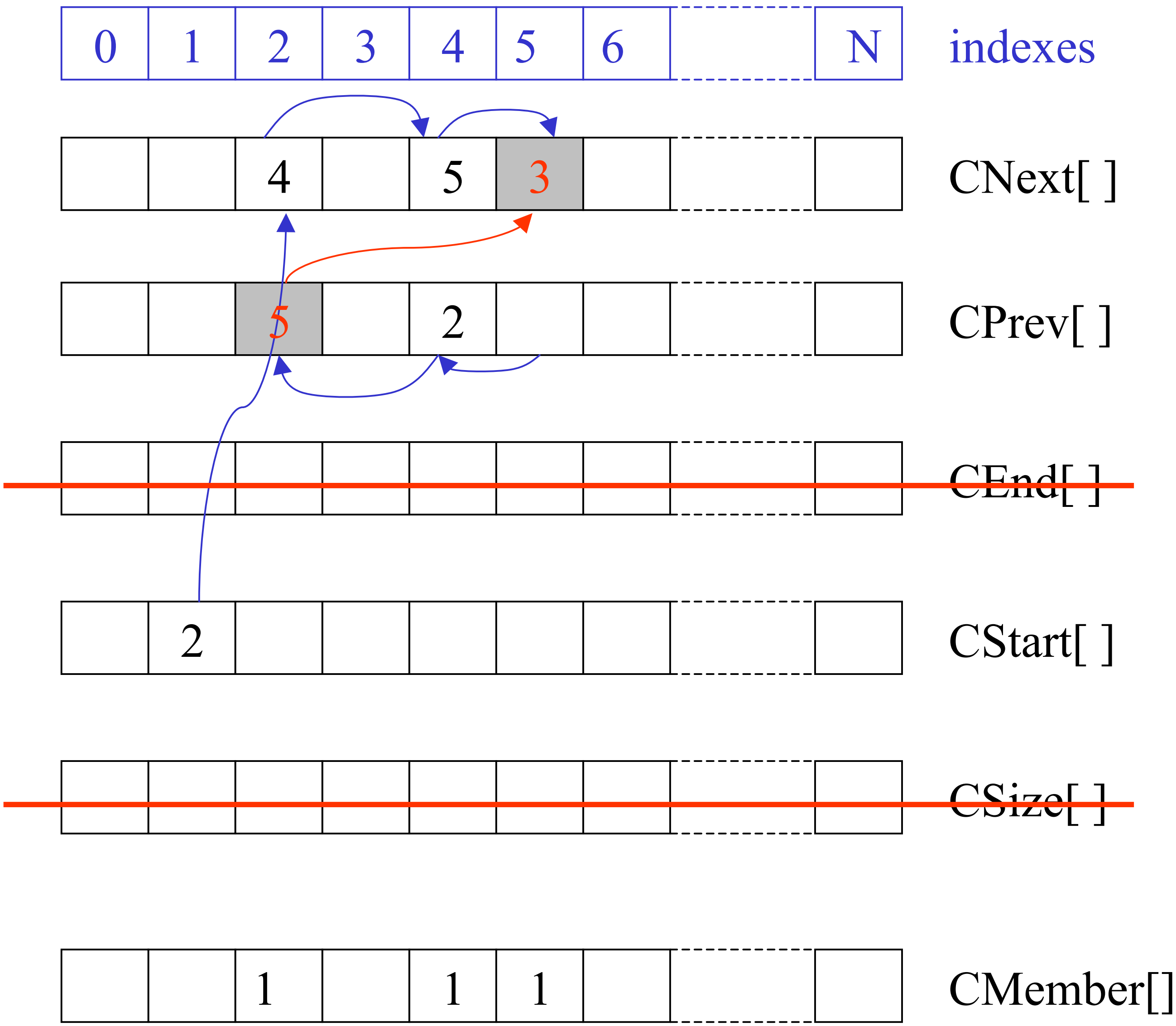


$f_2=\{3,5\}$

Total this slide 4*N
overall total 15*N

This completed the first step - an implementation of Crochemore’s algorithm that requires 15NM bytes of memory.

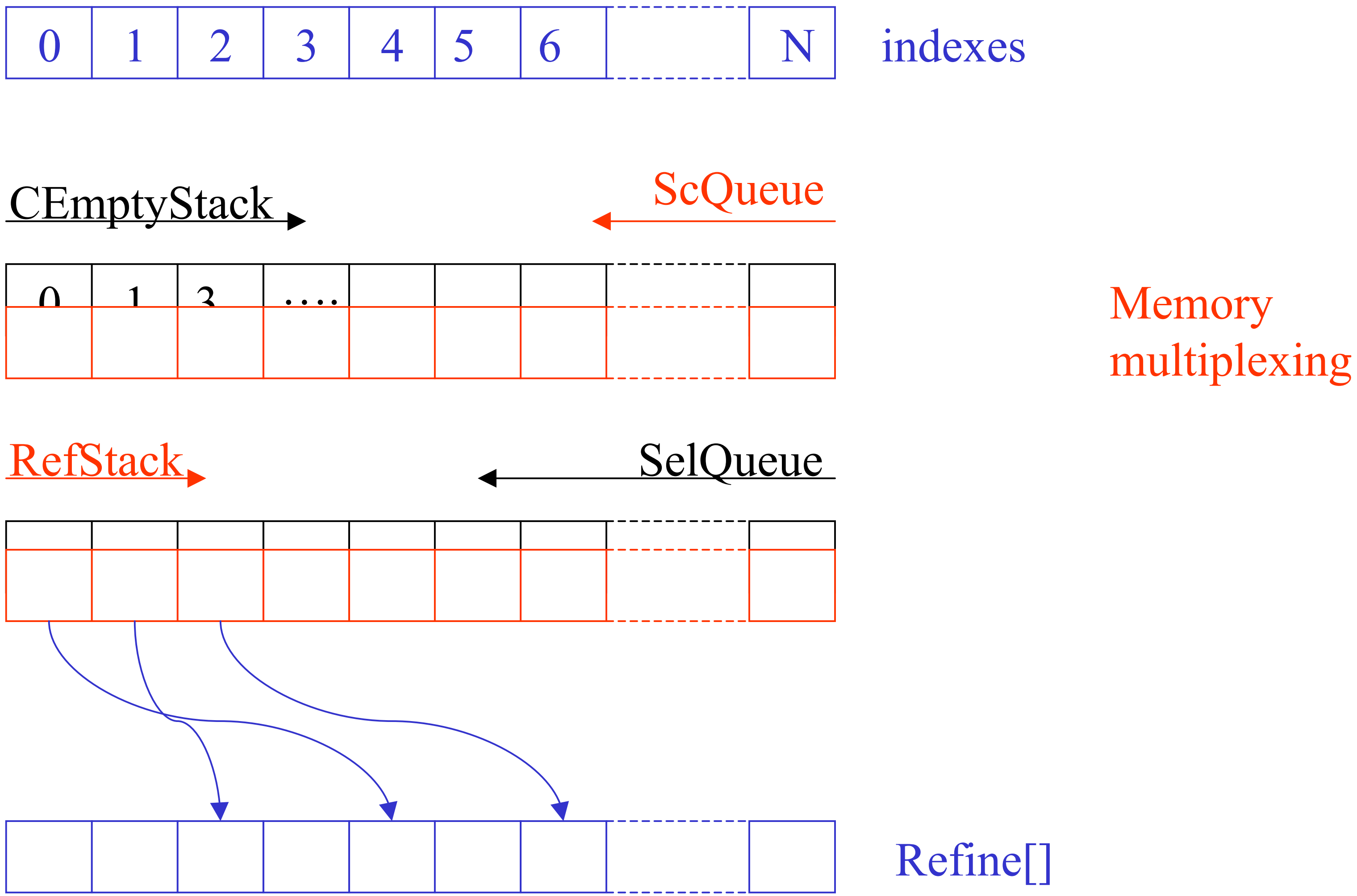
Now we are going to use “tricks” of memory multiplexing and memory virtualization to bring it down to 10NM.



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

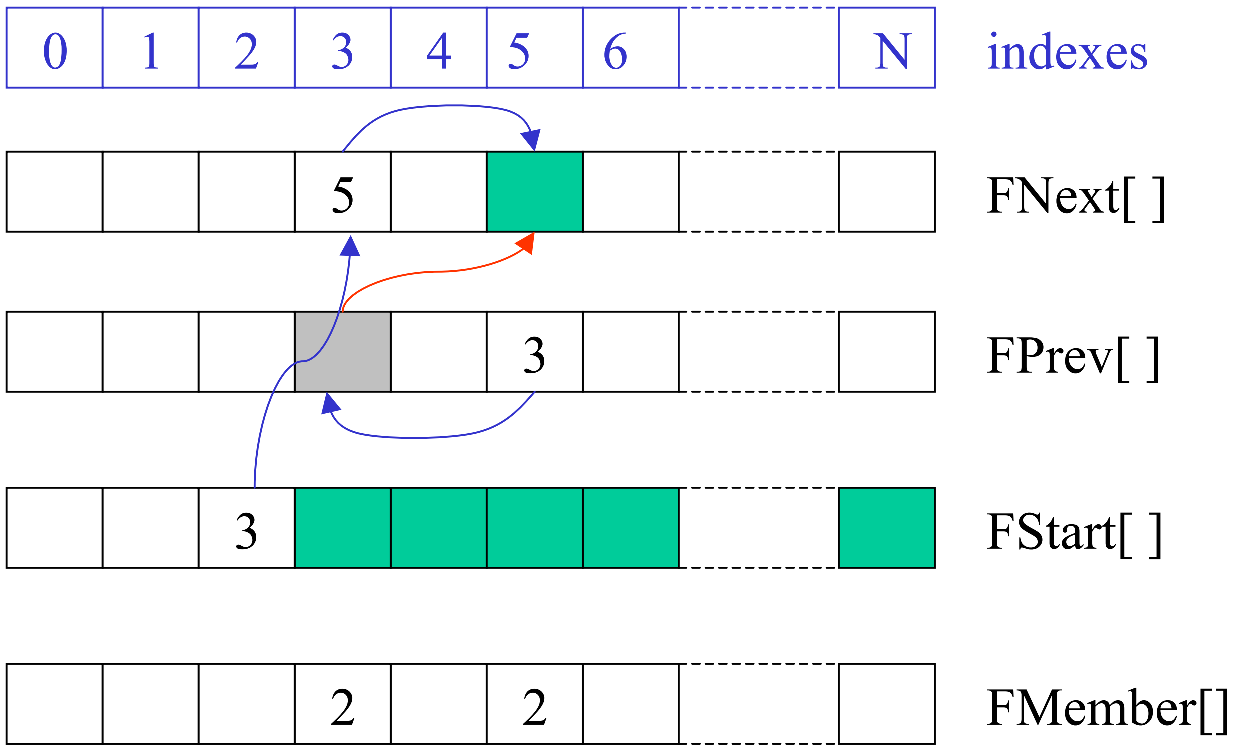
Memory virtualization

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