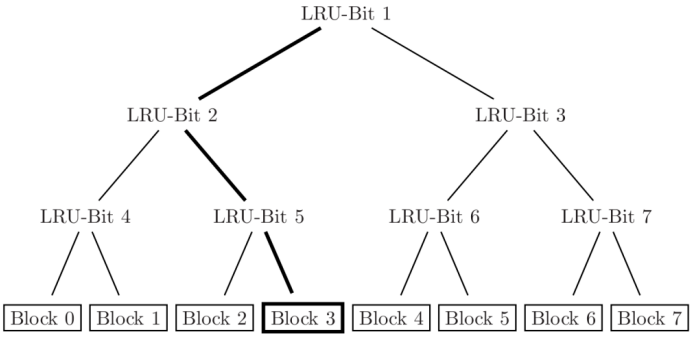
1. Cache, LRU, Pseudo-LRU

a)

LRU: Least Recently Used. Hier wird in einer Liste gespeichert, auf welche Cache Lines zuletzt zugegriffen wurde; nach jedem Zugriff auf eine Line wandert der Eintrag für diese Line ganz nach vorne in der Liste. Das letzte Element der Liste ist immer dasjenige, auf das „least recently" (dt.: „am wenigsten kürzlich", also am längsten nicht mehr) zugegriffen wurde.

Pseudo-LRU: Da die Verwaltung einer LRU-Liste aufwendig ist, kann man mit Pseudo-LRU ein einfacheres Verfahren nutzen, bei dem ein (binärer) Entscheidungsbaum verwendet wird, um eine Annäherung an die LRU-Liste zu speichern.



b) Bei LRU gibt es für n Blöcke $n!$ verschiedene Konfigurationen, bei Pseudo-LRU 2^{n-1} verschiedene Konfigurationen (weil der Baum mit n Blättern n-1 innere Knoten inkl. Wurzel hat und jeder dieser n–1 Knoten ein Bit speichert).

Die beiden Strategien sind genau dann identisch, wenn die Anzahl der darstellbaren Konfigurationen gleich ist, also: wenn $n! = 2^{n-1}$ gilt. Hier hilft eine Wertetabelle:

n	$n!$	2^{n-1}
1	1	1
2	2	2
3	6	4
4	24	8
5	120	16

...

Man sieht, dass $n!$ viel schneller wächst als 2^{n-1}. Darum gilt Gleichheit nur für die beiden Fälle n=1 und n=2.

c) Ich habe nur Pseudo-LRU implementiert, das Programm ist aber für eine Erweiterung um LRU vorbereitet.

```
METHOD = "PLRU"                 # LRU oder PLRU

SEQUENCE = [ "S", "R1", "R3", "S", "S", "R1", "R4" ]
                                # "Ri" = READ(i), "S" = Store

LEFT  = 0
RIGHT = 1

# plru: pseudo-LRU-bits
plru = [-100,1,1,1,1,1,1,1]   # init; Extrawert am Anfang, damit wir die 7
                              # Pseudo-LRU-Bits als plru[1] .. plru[7] ansprechen
                              # koennen.
                              # [1,1,1...]: Belegung nach Zugr. [0,1,2,3,4,5,6,7]
```

```
def neg(i): return 1-i        # tauscht 0->1, 1->0
def plru_update ( (a,b), (c,d), (e,f) ):
  # Paar (a,b) bedeutet: am Knoten a nach links oder rechts
  # a kann also aus [1..7] sein, b kann LEFT (0) oder RIGHT (1) sein
  plru[a] = b
  plru[c] = d
  plru[e] = f
  return

def plru_access(n):
  # Block n zuletzt benutzt
  print "plru_access(): Accessing", n
  if n == 0: plru_update ( (1,LEFT),  (2,LEFT),  (4,LEFT)  )
  if n == 1: plru_update ( (1,LEFT),  (2,LEFT),  (4,RIGHT) )
  if n == 2: plru_update ( (1,LEFT),  (2,RIGHT), (5,LEFT)  )
  if n == 3: plru_update ( (1,LEFT),  (2,RIGHT), (5,RIGHT) )
  if n == 4: plru_update ( (1,RIGHT), (3,LEFT),  (6,LEFT)  )
  if n == 5: plru_update ( (1,RIGHT), (3,LEFT),  (6,RIGHT) )
  if n == 6: plru_update ( (1,RIGHT), (3,RIGHT), (7,LEFT)  )
  if n == 7: plru_update ( (1,RIGHT), (3,RIGHT), (7,RIGHT) )
  return

def plru_get_lru():
  seek1 = 2+neg(plru[1])                # LRU-Bit 1; Ergebnis 2 oder 3
  block = 4 * (seek1-2)                 # Block: 0 oder 4
  seek2 = 2 * seek1 + neg(plru[seek1])  # naechste Stufe, seek2 ist aus [4,5,6,7]
  block = block + 2 * neg(plru[seek1])  # Block: 0,2,4 oder 6
  block = block + neg(plru[seek2])      # Block: 0,1,2,3,4,5,6 oder 7
  return block

def plru_print_tree():
  print "Baum:  1 :     ", plru[1]
  print "|     23 :   ",   plru[2]," ",plru[3]
  print "|____4567:  ",    plru[4],plru[5],plru[6],plru[7]
  return

def READ(i):
  if   METHOD == "PLRU": plru_access(i)
  elif METHOD == "LRU":  lru_access(i)
  return

def STORE():
  print "Store"
  if METHOD == "PLRU":
    i = plru_get_lru()
    plru_access(i)
  elif METHOD == "LRU":
    i = lru_get_lru()
    lru_access(i)
  return

# Hauptprogramm
for access in SEQUENCE:
  if access[0] == "S":
    STORE()
  elif access[0] == "R":
    char = access[1]
    i = ord(char) - ord("0")  # "3" in 3 umwandeln
    READ(i)
```

Eine leicht erweiterte Version liegt auf dem Webserver.