Chinese Remainder List

Joseph Stanton
State University of New York
Institute of Technology

About Me

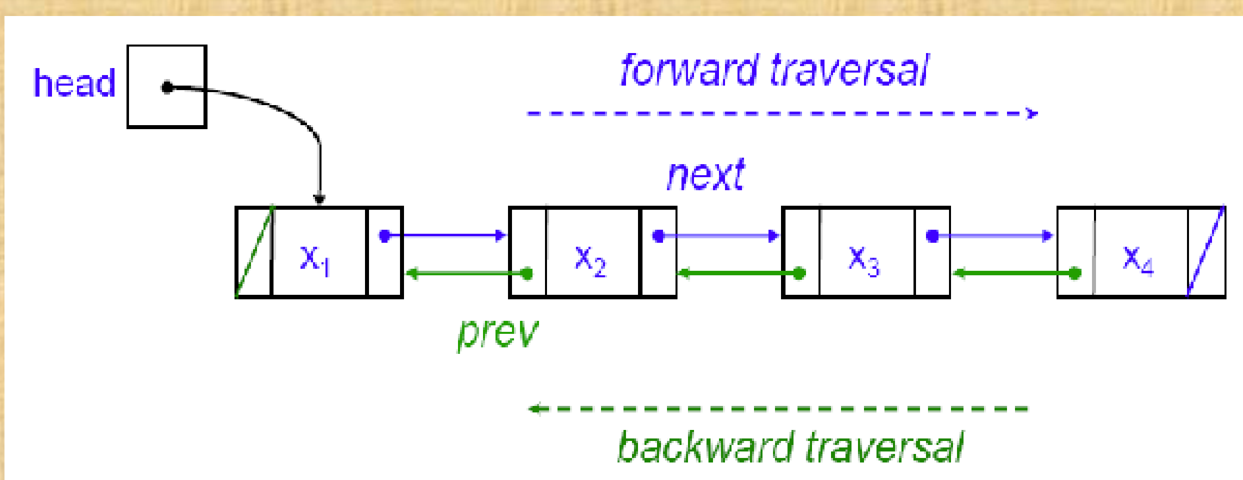
- Junior
- Accelerated BS/MS Computer Science
- Applied Mathematics

Terminology

- Pointer
  - Memory address
  - Positive integer less than architecture limit (typically)
  - (typically) in the range $0 - 2^n$

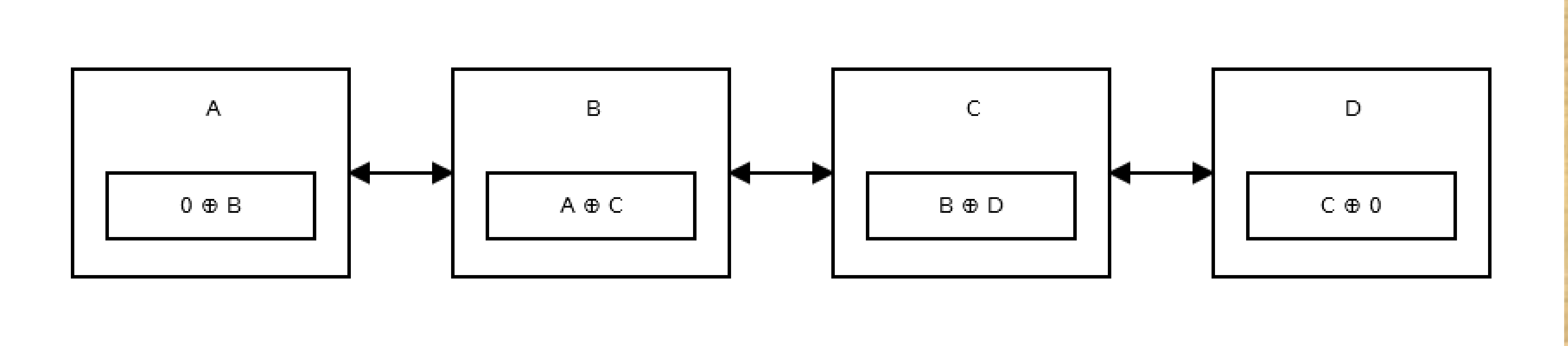
Linked List

- Data structure for dynamic manipulation of lists of items
- Commonly found as Doubly Linked Lists



Special Case

- XOR Linked List
- Advantage:
  – Reduces space required to store list
- Disadvantage:
  – Loss of some traversing abilities



Goal/Possible Solution

- Reduce the pointers to a single value while retaining traversing abilities
- Use the Chinese Remainder Theorem to compress the pointers

The problem

- *Problem Setup*
  - $a_1, a_2 \in \mathbb{Z}^+$ $and\ a_1, a_2 = addresses\ to\ be\ encoded$
  - $m_1, m_2, M, M_1, M_2, X \in \mathbb{Z}$
  - $x \in Z, x\ is\ the\ solution\ we\ are\ looking\ for$
  - $x \equiv a_1 (mod\ m_1)$
  - $x \equiv a_2 (mod\ m_2)$
  - $\gcd(m_1, m_2) = (m_1, m_2) = 1$
  - $M = m_1\ m_2$
  - $X = a_1 M_1 y_1 + a_2 M_2 y_2 \equiv x (mod\ M)$
- x is guaranteed to be unique modulo M

General Algorithm

1. Compute M
   – Computing M_n is trivial
2. Solve the congruencies (for y_n)
   – $M_1 y_1 \equiv 1(mod\, m_1)$
   – $M_2 y_2 \equiv 1(mod\, m_2)$
3. Compute the resulting equation
   – $X = a_1 M_1 y_1 + a_2 M_2 y_2 \equiv x(mod\, M)$

M

- Compute M_1 *and* M_2
- $M_1 = \frac{M}{m_1} = m_2$
- $M_2 = \frac{M}{m_2} = m_1$

Solve Congruencies

- Extended Euclidean Algorithm
  - Not going to go through this because I develop a better method for this specific application later on

Optimizations/Simplifications

- Assume $m_2 = m_1 + 1$
- This is can be proven by many different ways
- Bezout's Theorem
  - $\gcd(m_1, m_2) = a * m_1 + b * m_2 = a * m_1 + b * (m_1 + 1)$
  - $= a * m_1 + b * m_1 + b$
  - $= m_1(-1) + (1)m_2 + (1)$
  - $= -m_1 + m_1 + 1 = 1$

Effects on Modular Inverse

- Simplifies the calculation of y_1 immediately
  - $M_1 y_1 = m_2 y_1 = (m_1 + 1) y_1 \equiv 1 * y_1 \equiv 1 (mod\, m_1)$
  - $y_1 = 1$
- Slightly more work is required to simplify the second congruency

Second Congruency

- $M_2 y_2 = m_1 y_2 \equiv 1 (mod\, m_2)$
  - $m_1 y_2 = n * \ m_2 + 1$
  - $=> m_1 y_2 - n * \ m_2 = 1$
  - $=> m_1 y_2 - n * \ m_1 - n = 1$
  - Assuming $y_2 = -1$ and $n = -1$
  - $=> m_1(-1) - (-1) * \ m_1 - (-1) = -m_1 + m_1 + 1 = 0 + 1 = 1$

New Algorithm

1. Compute M
2. Compute the equation
   - $X = 1 * a_1 * m_2 + (-1) * a_2 * m_1 \equiv x(mod\, M)$

My implementation specifics

- let $m_1 = 2^n$ such that $m_1 \geq a_1, a_2$
- Have to store n, although it can be stored as a single byte in my implementation and work for architectures 64 bits and less
- Not completely solved…

Example

- Assume $a_1 = 3\ and\ a_2$ =5
- $m_1 = 2^3 = 8, m_2 = 9, M = 72$
- $X = (1) * 3 * 9 - (5) * 8 = -13 \equiv 59(mod\ 72)$
- $59\ mod\ 8 = 3, 59\ mod\ 9 = 5$

Problems

- Architecture limitations
  - Computer can’t handle that large of an integer
- Efficiency
  - Not as large a problem as the prior (depending on who you are)

More Realistic Example

- Assume $a_1 = 2^{33} and\ a_2 = 2^{32}$
- On a 64 bit system (i.e. max integer = 2^{64})
- $M = (2^{34})(2^{34} + 1) = 2^{68} + 2^{34}$
- $X = 1 * 2^{33} * (2^{34} + 1) + (-1)(2^{32})(2^{34}) = 2^{67} + 2^{33} - 2^{36}$
- $X = 2^{67} + 2^{33} - 2^{66} \equiv 73786976303428141056 (mod\ M)$

Hope

- Can possibly be used reduce storage requirements in the average case
- The average computer won't exceed 64gb of ram (without difficulty)
- Most hover around 4gb-8gb range
- Can possibly save a few bytes everywhere it is used (but the operating system may not allow this)