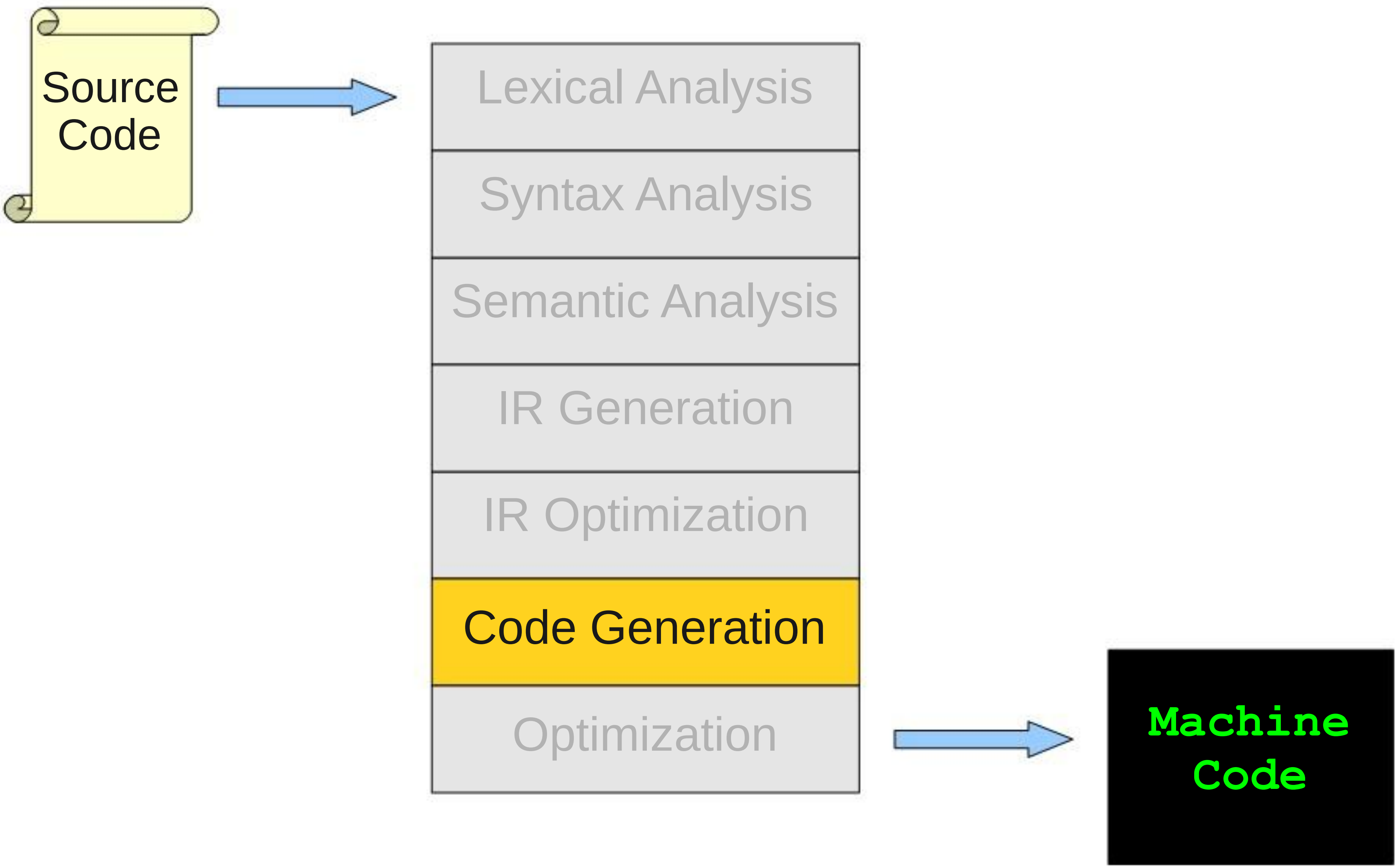
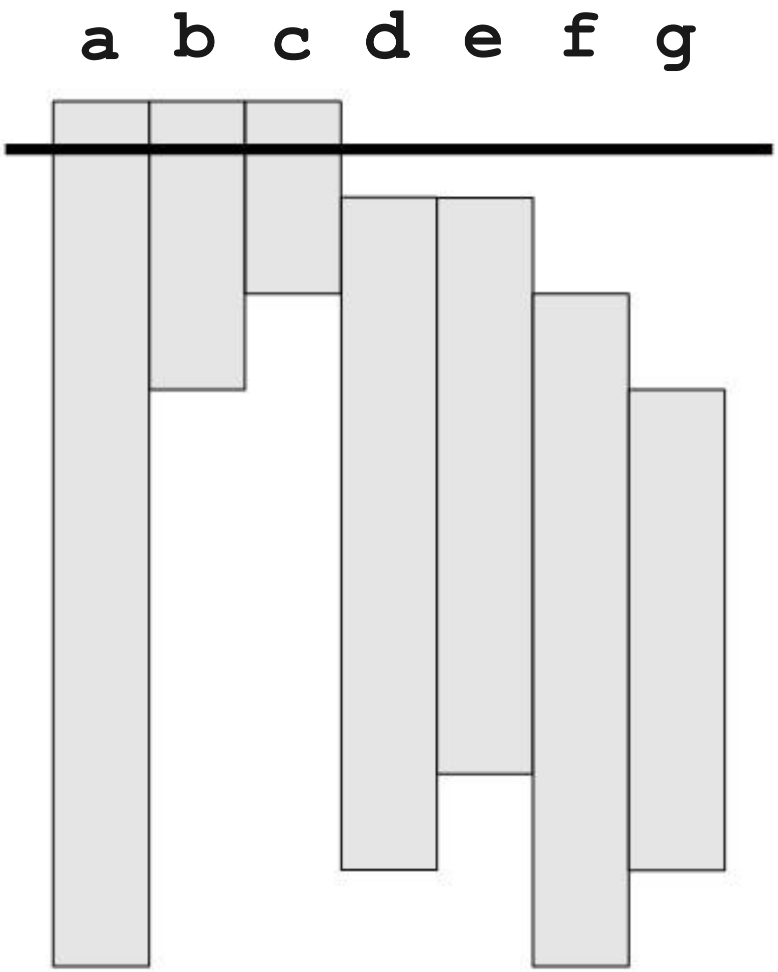
Where We Are


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

Register Spilling (Continue)

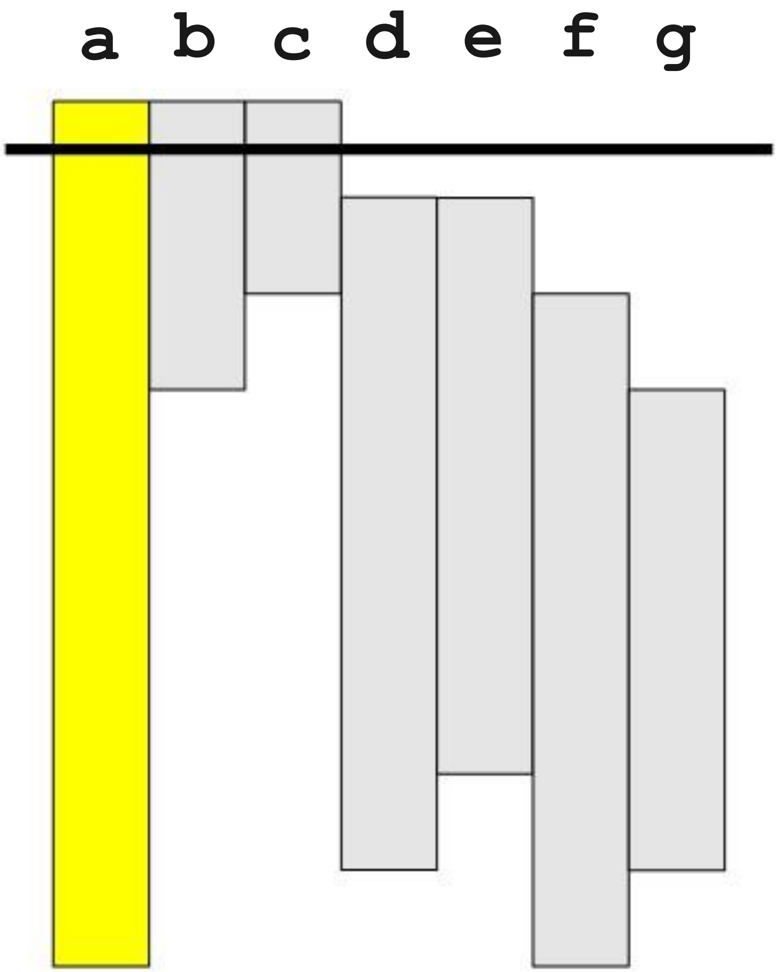
- If a register cannot be found for a variable v, we may need to **spill a variable into memory**.
- When we **need a register for the spilled variable,** temporarily **evict a register to memory.**
- When done with that register, write its value to the memory (if necessary) and load the old value back.
- Note: Some register allocation algorithms can handle spilling much more intelligently than this.
- Spilling is slow, but sometimes necessary.

Another Example



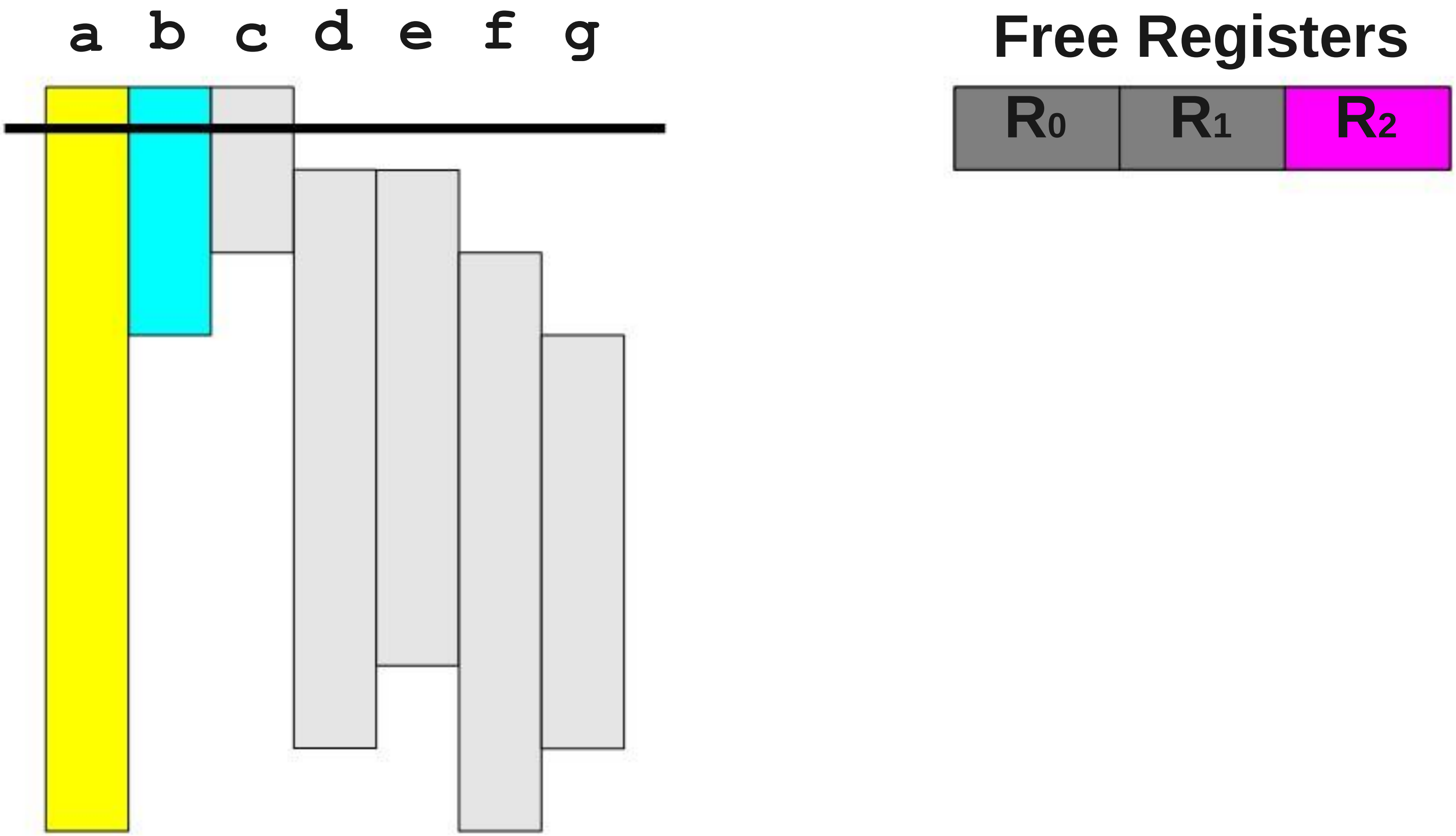
Free Registers

Another Example

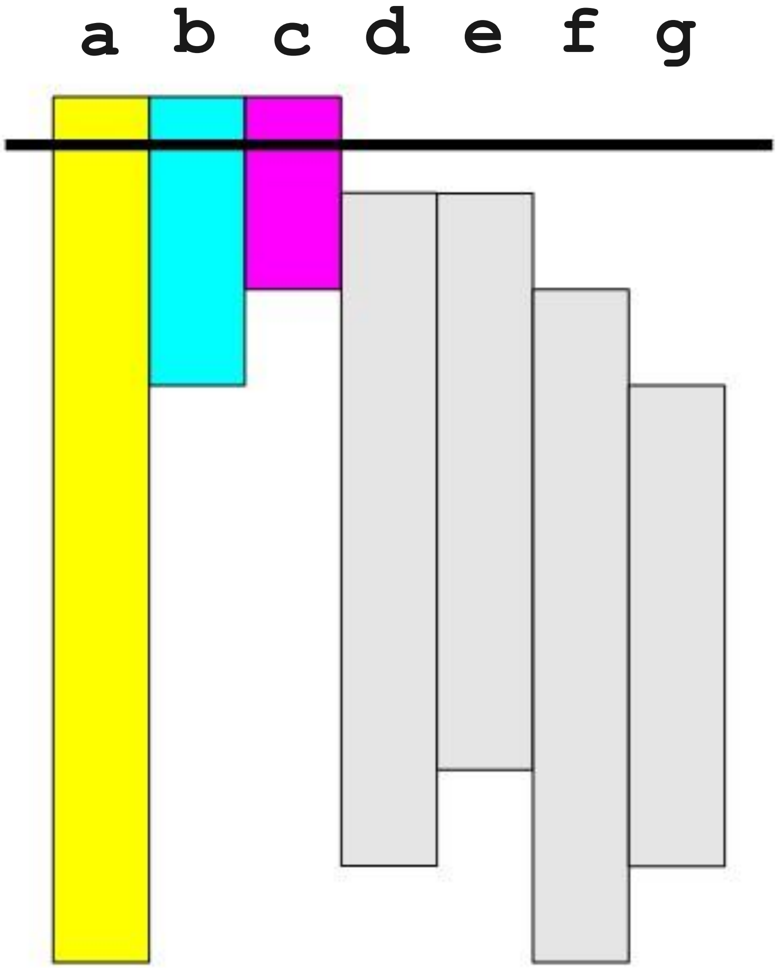


Free Registers

Another Example



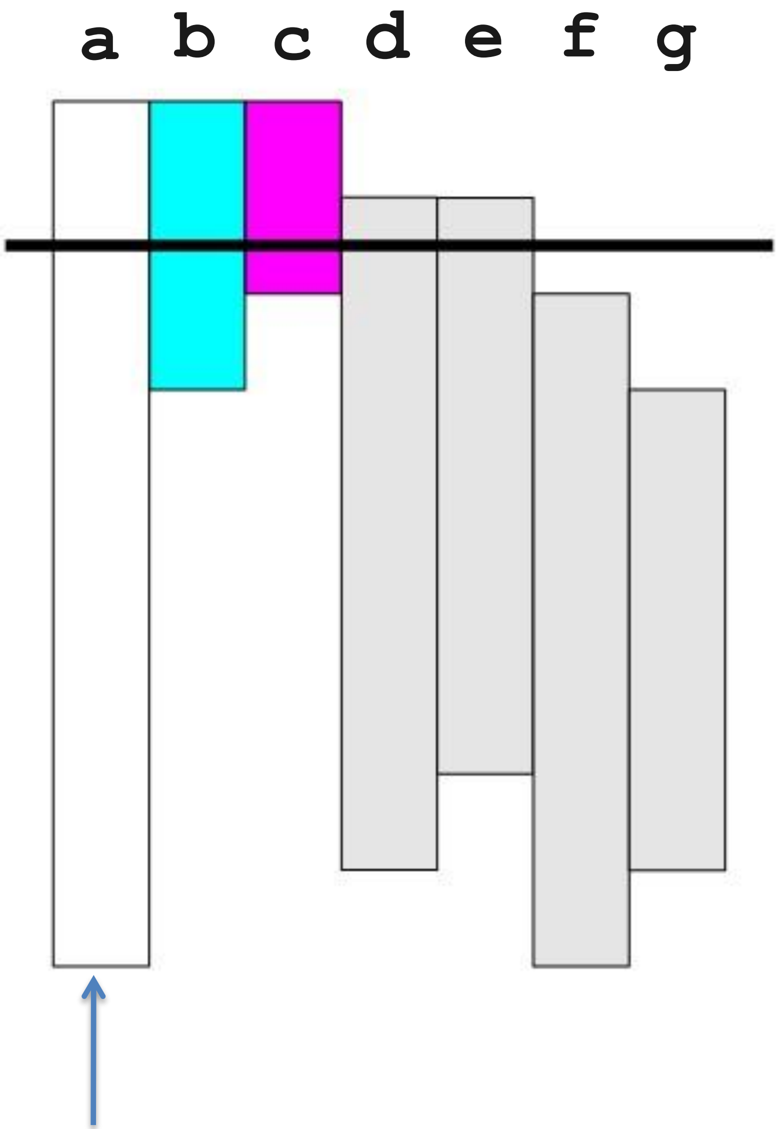
Another Example



Free Registers

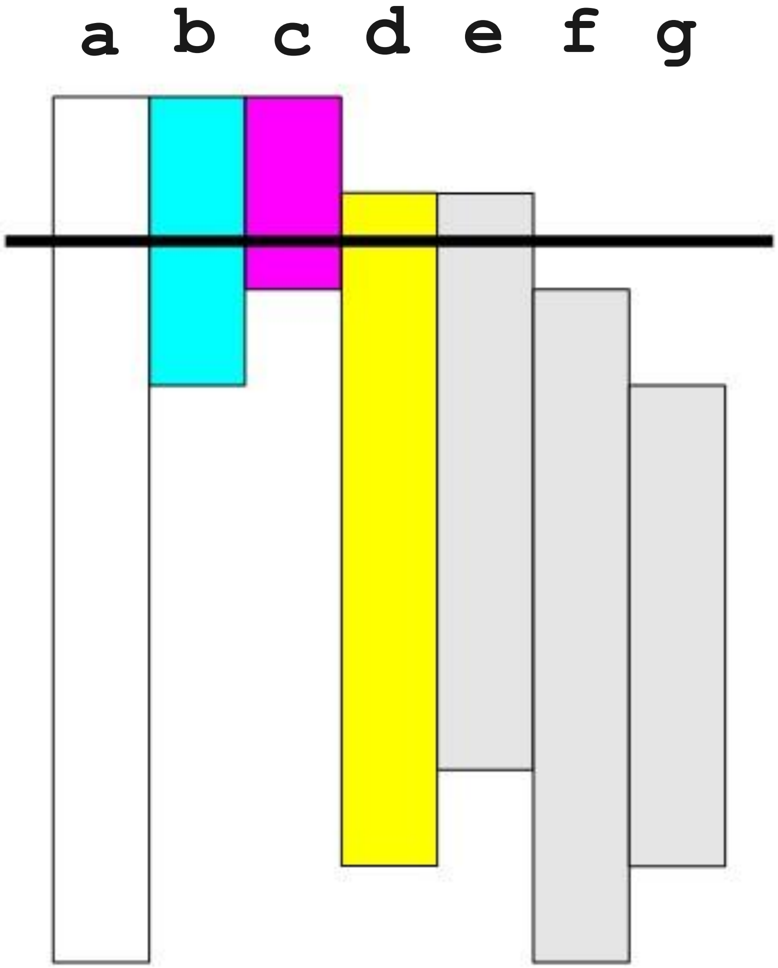
R_0	R_1	R_2

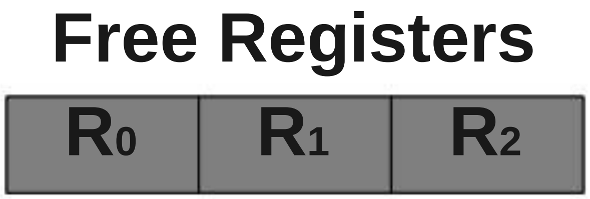
Another Example



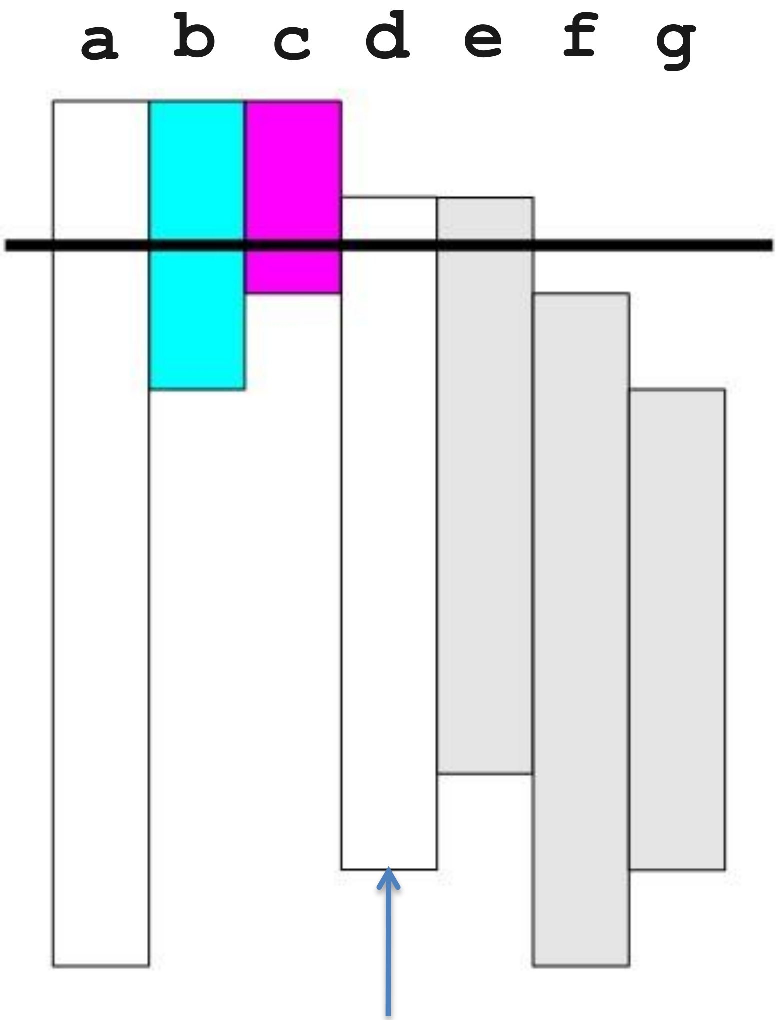
Spill the variable ‘a’ because it stays latest.

Another Example





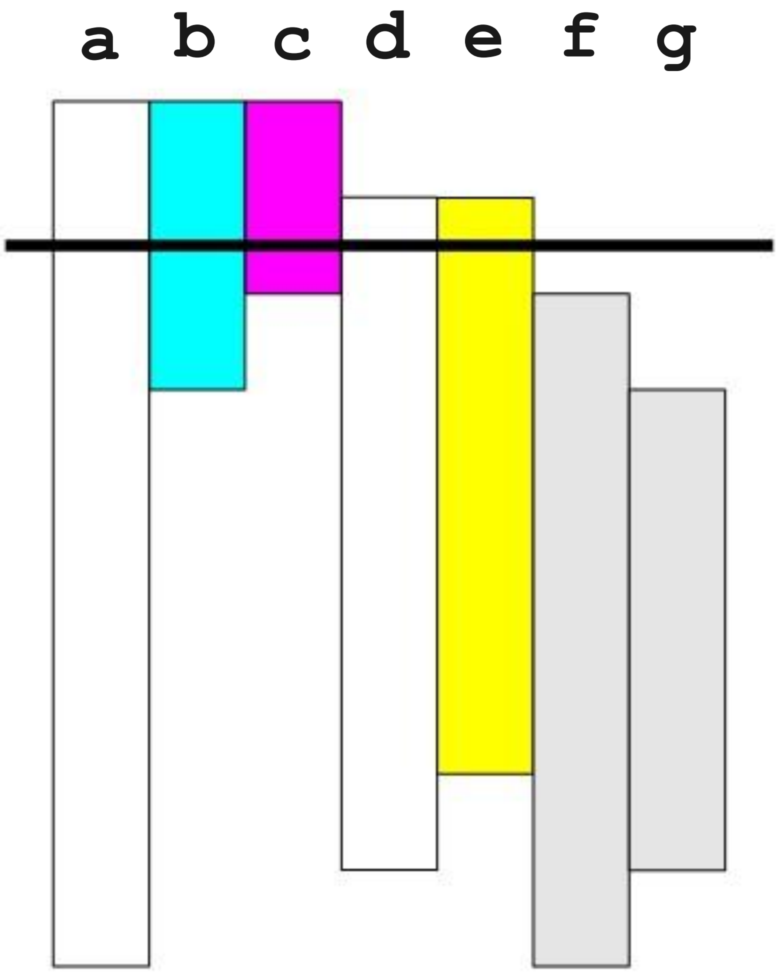
Another Example



Free Registers

Spill the variable ‘d’.

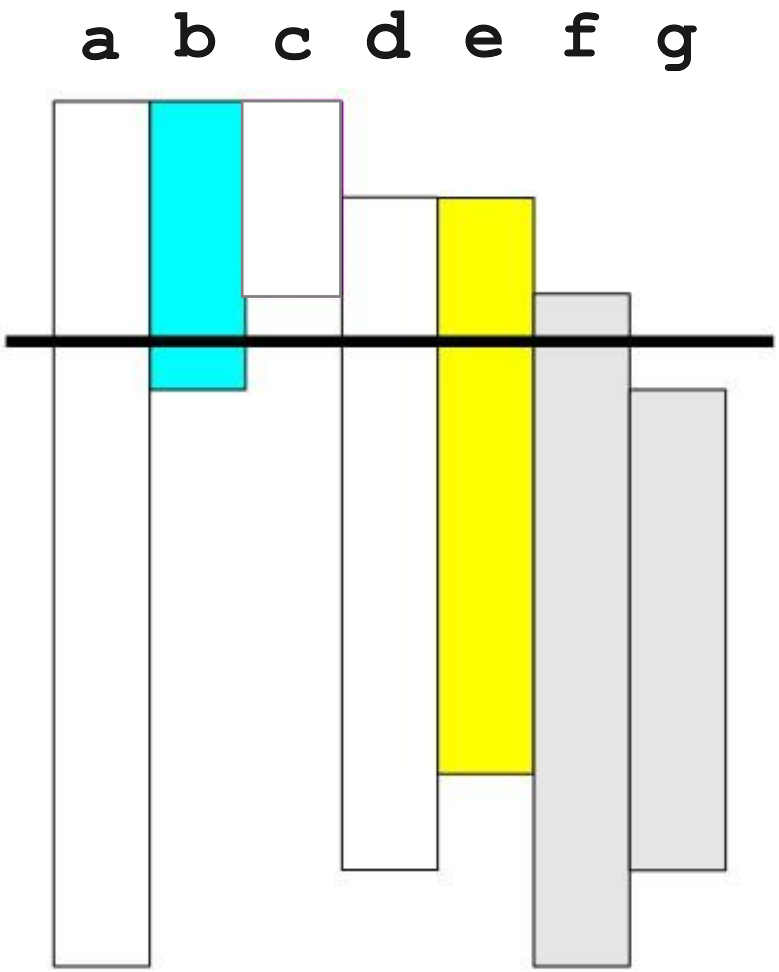
Another Example



Free Registers

R_0	R_1	R_2

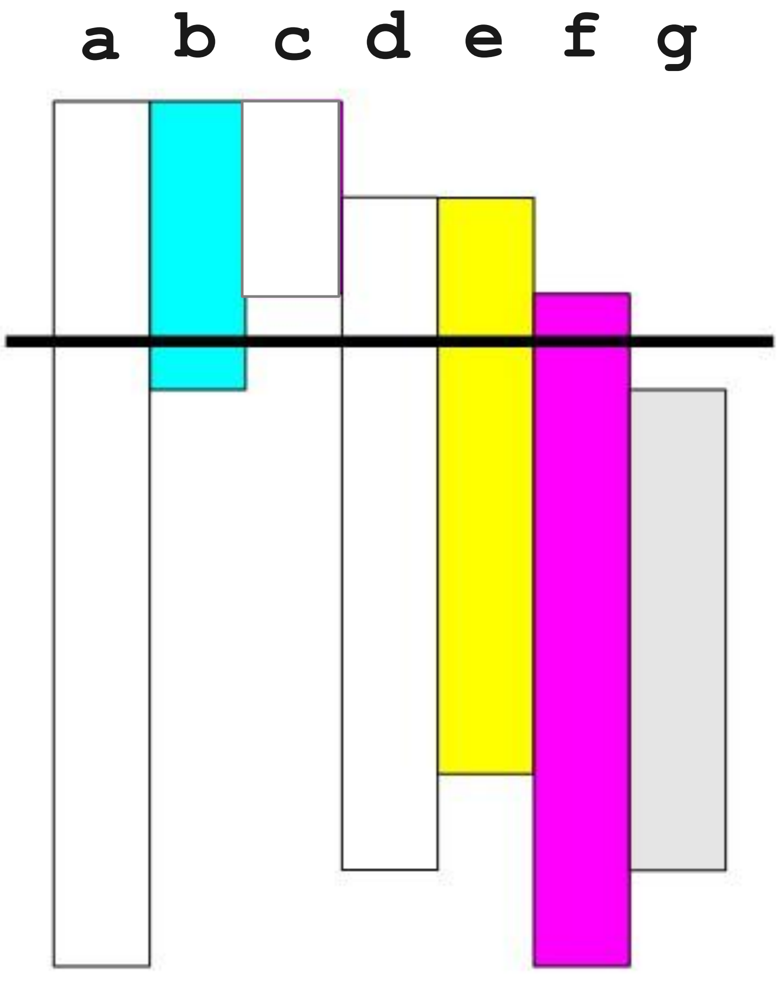
Another Example



Free Registers

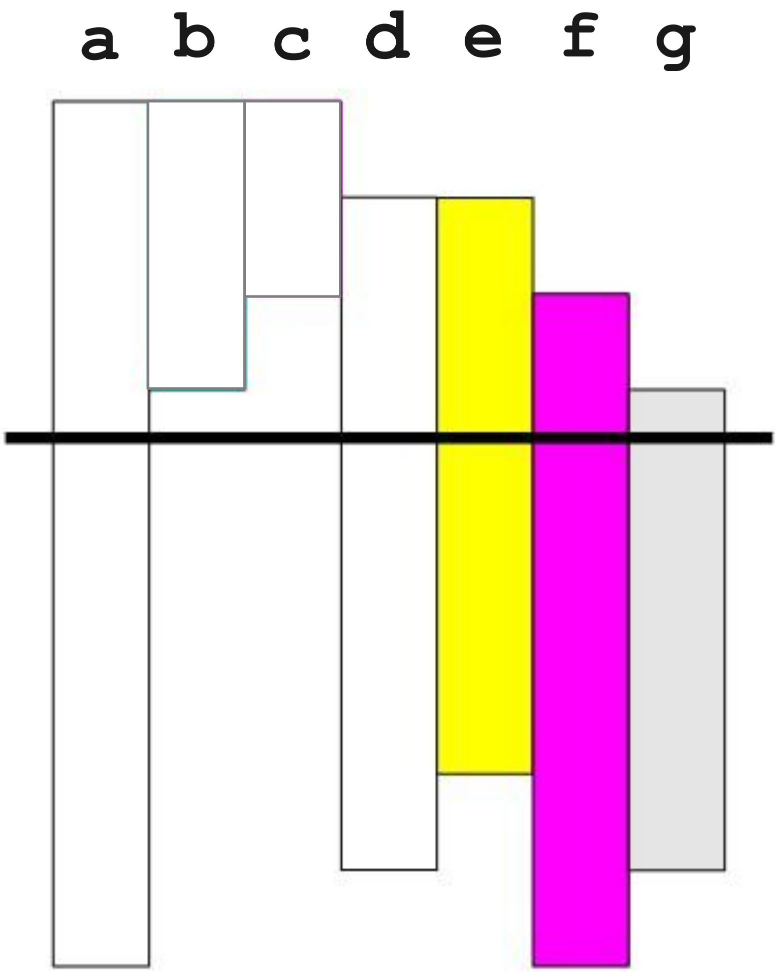
R_0	R_1	R_2

Another Example



Free Registers

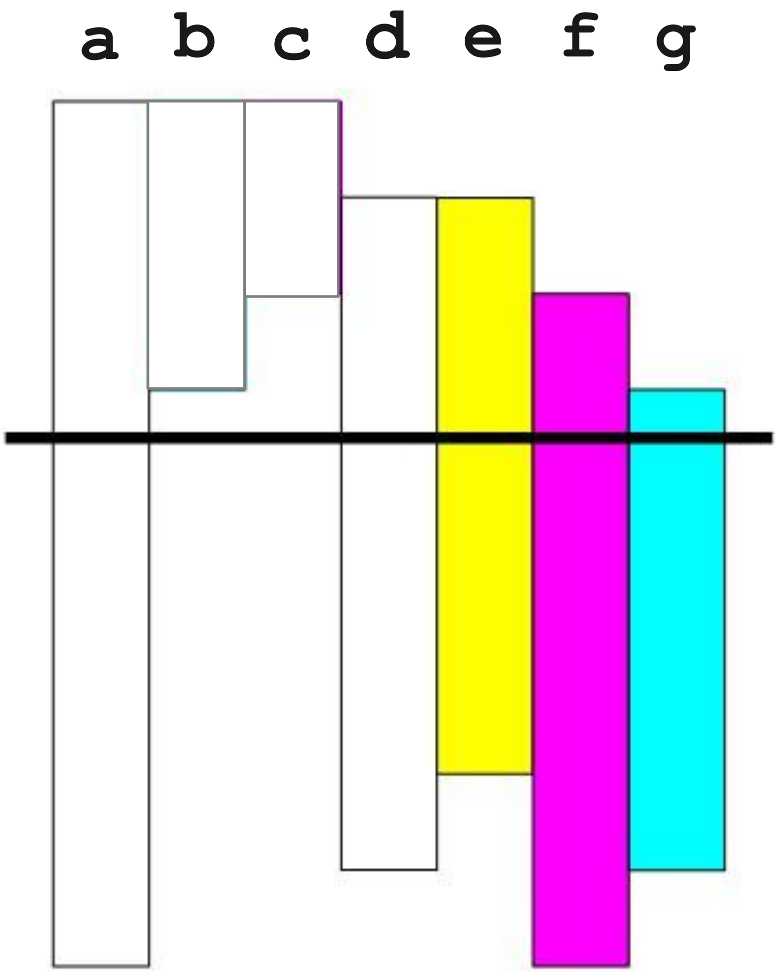
Another Example



Free Registers

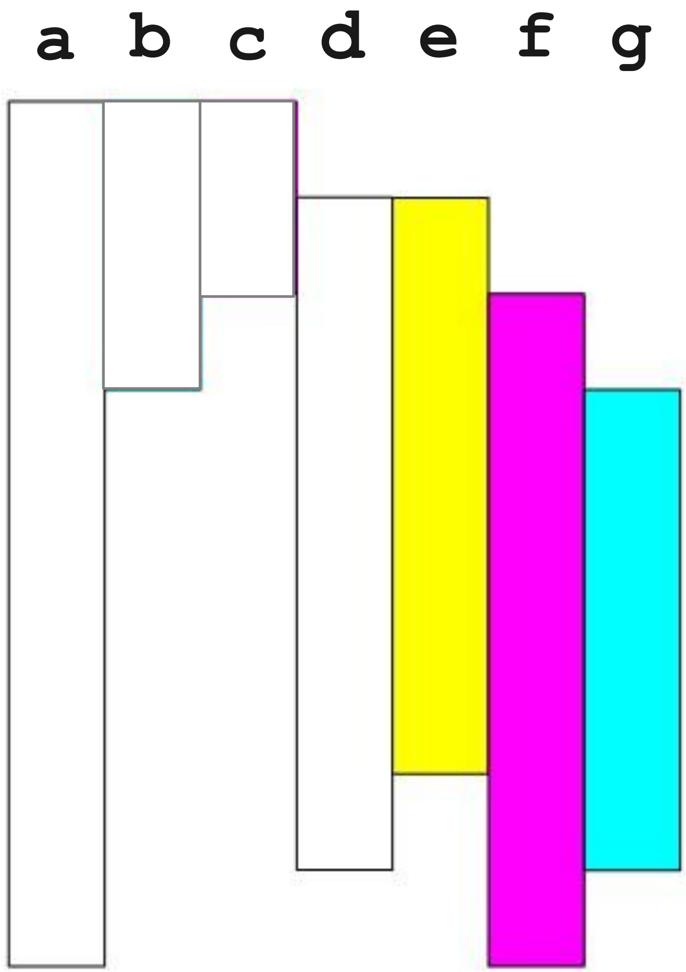
R_0	R_1	R_2

Another Example



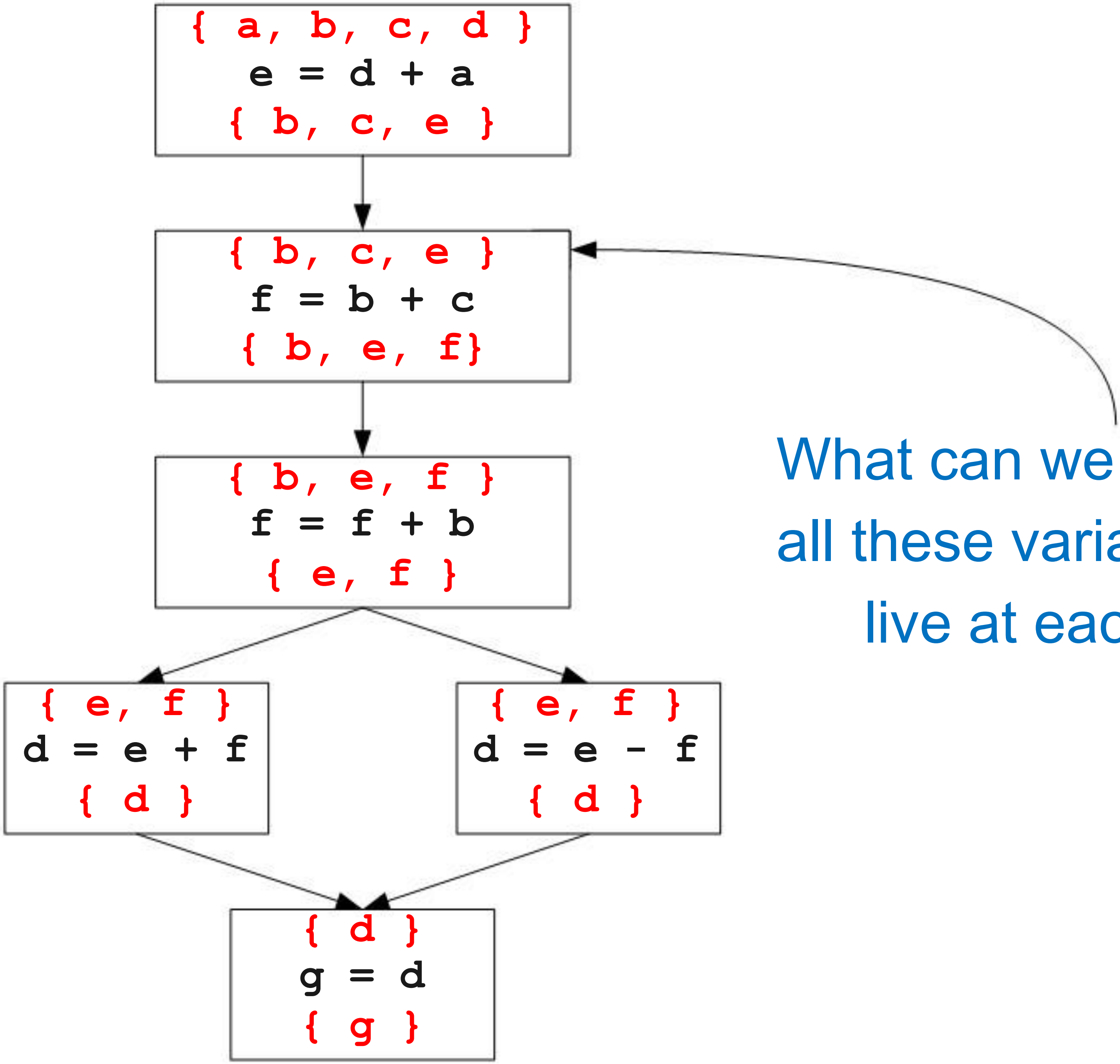
Free Registers

Another Example



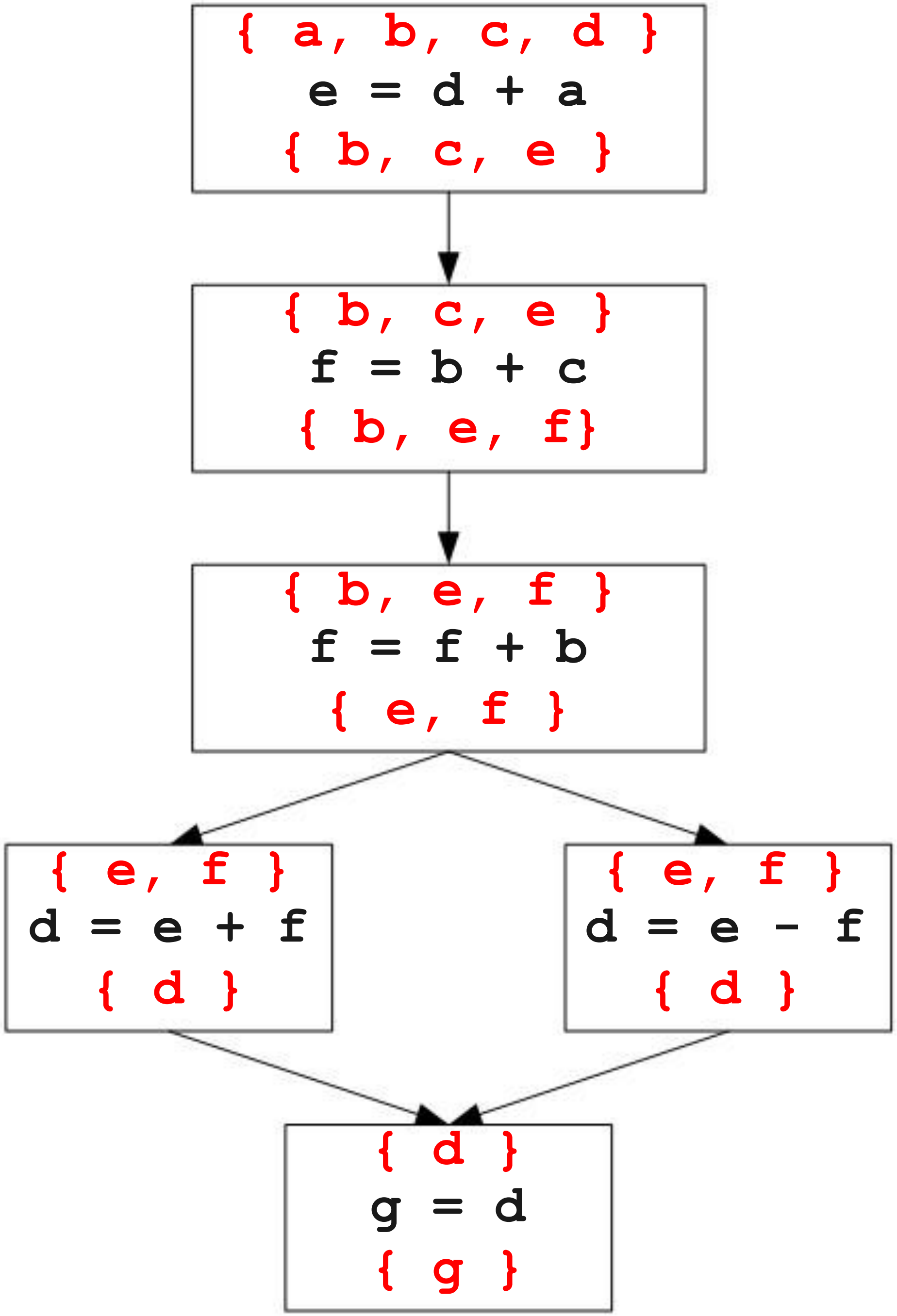
There is a smarter approach.

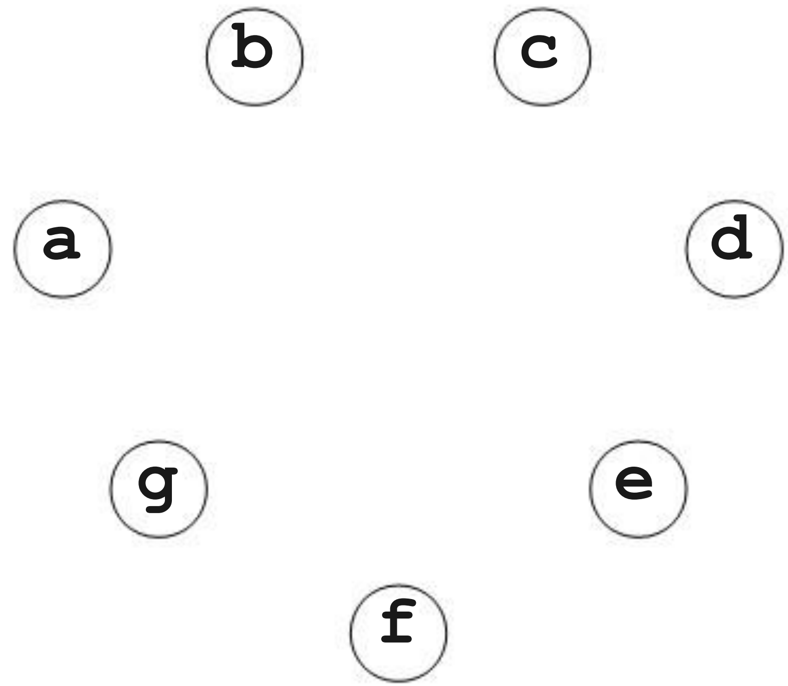
Smarter Approach



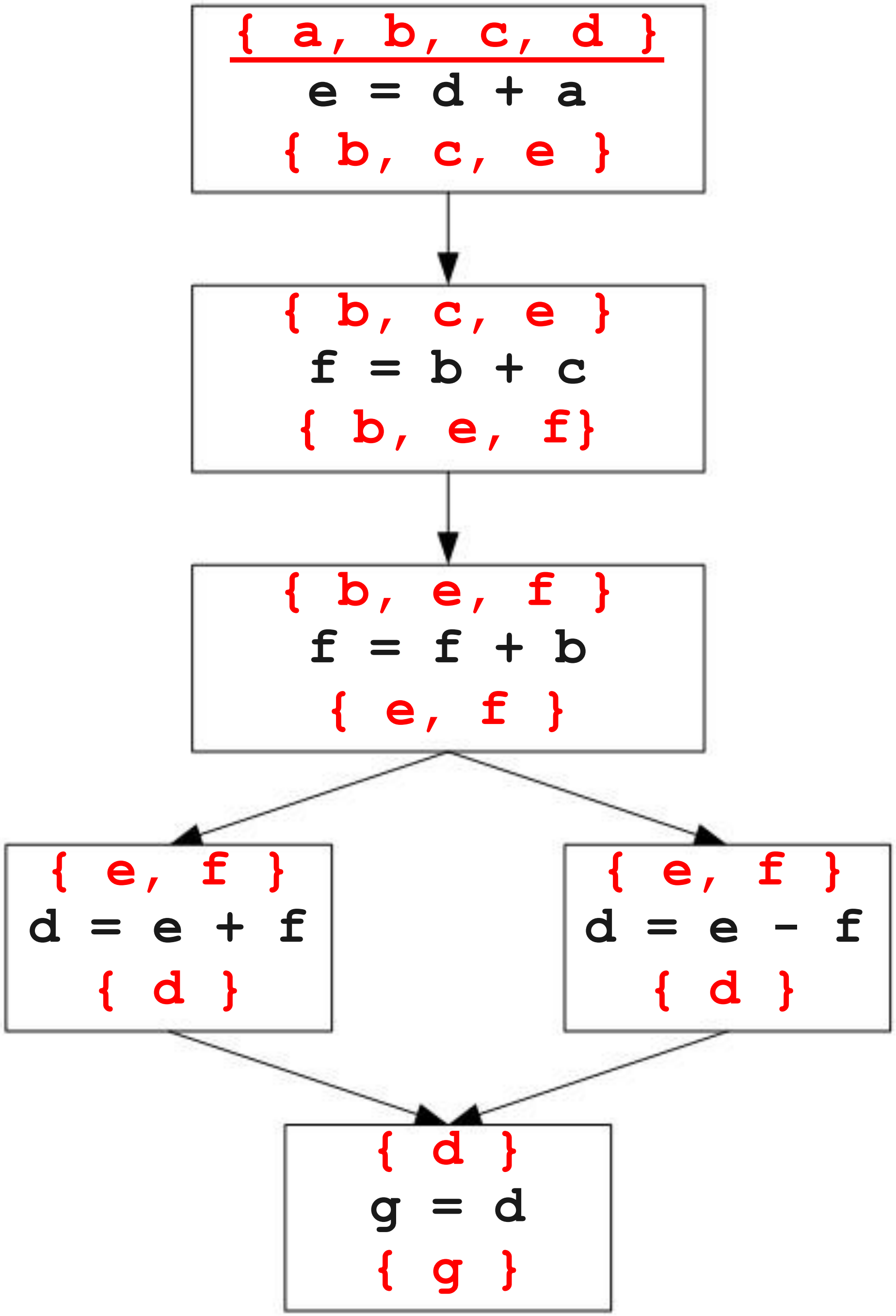
What can we infer from all these variables being live at each point?

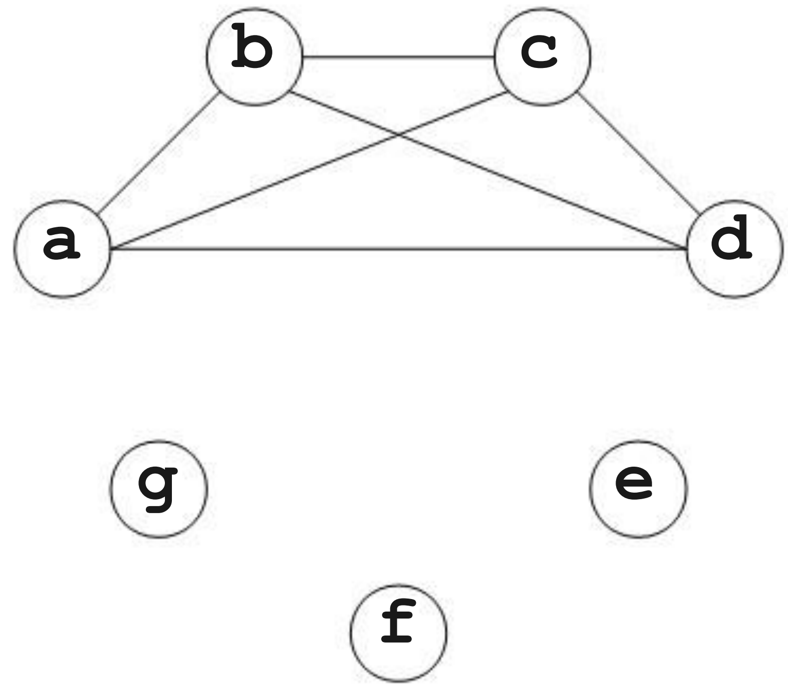
Smarter Approach



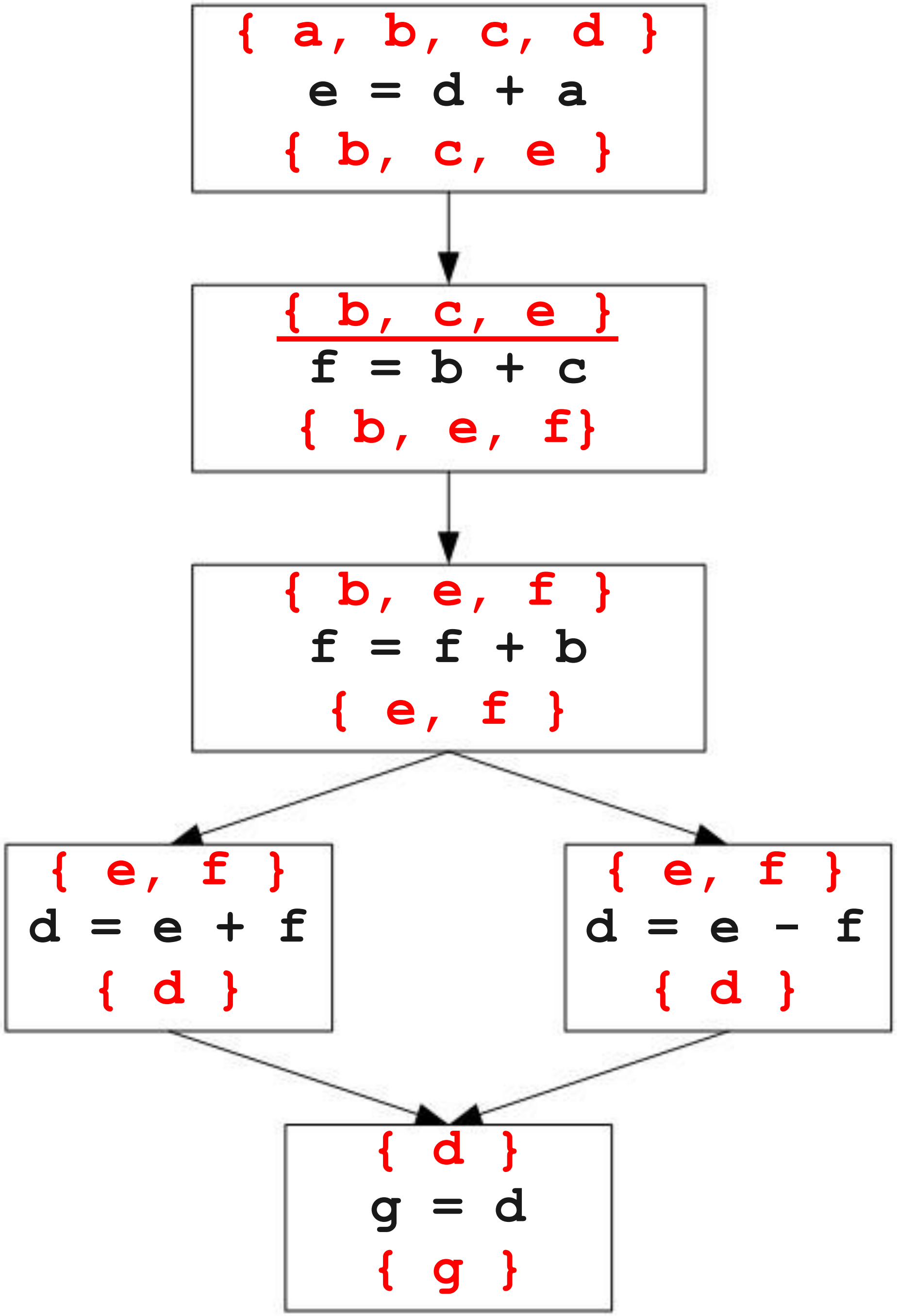


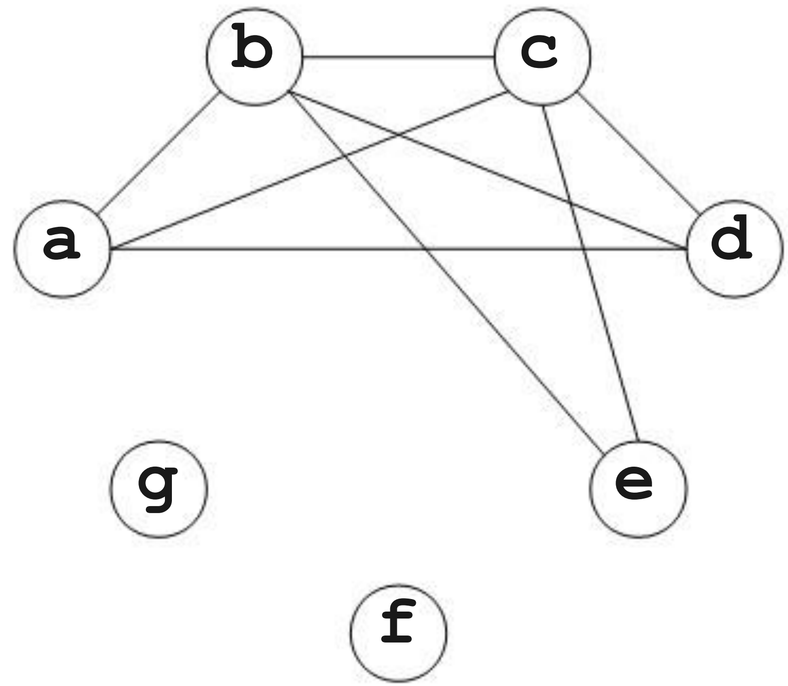
Smarter Approach



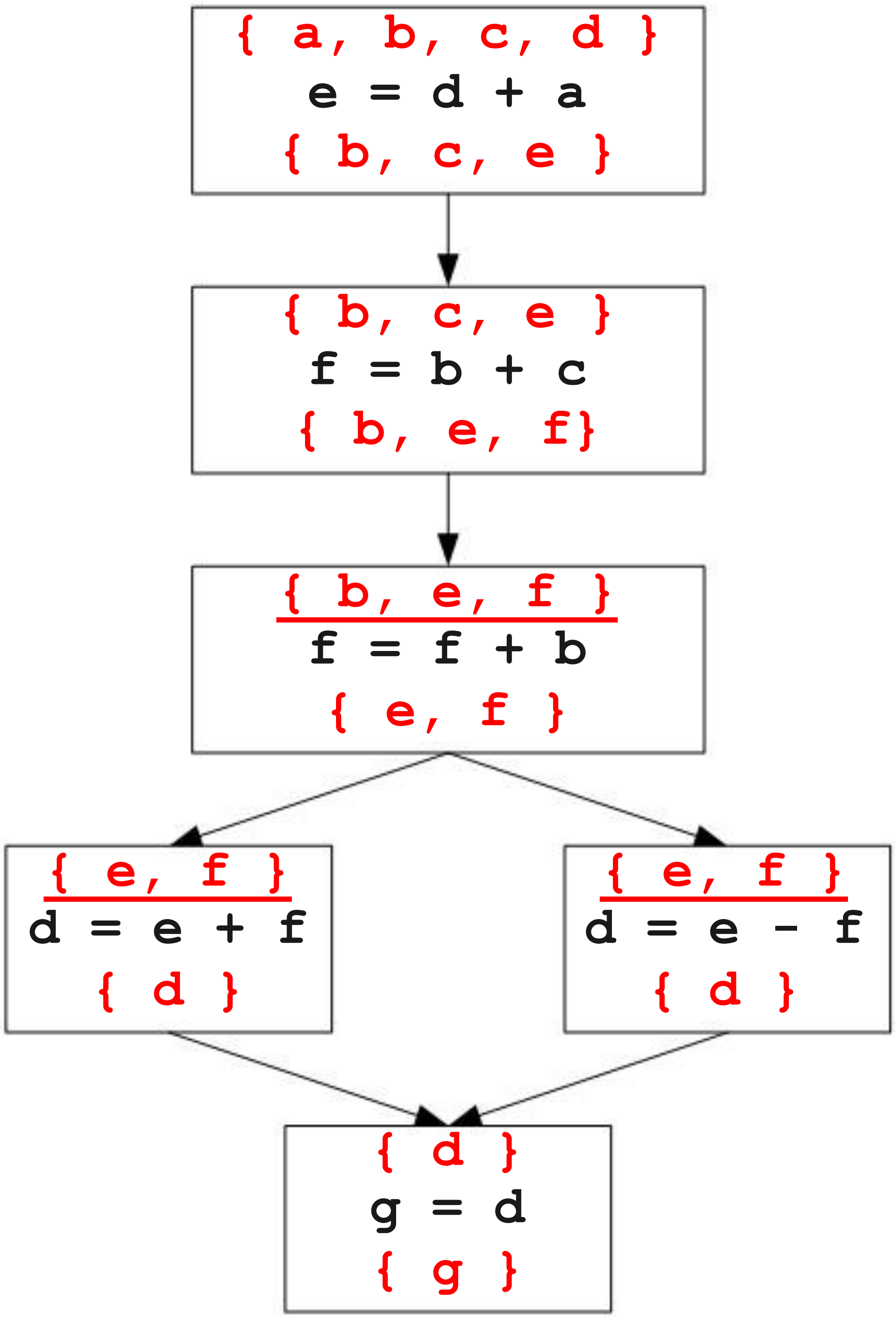


Smarter Approach

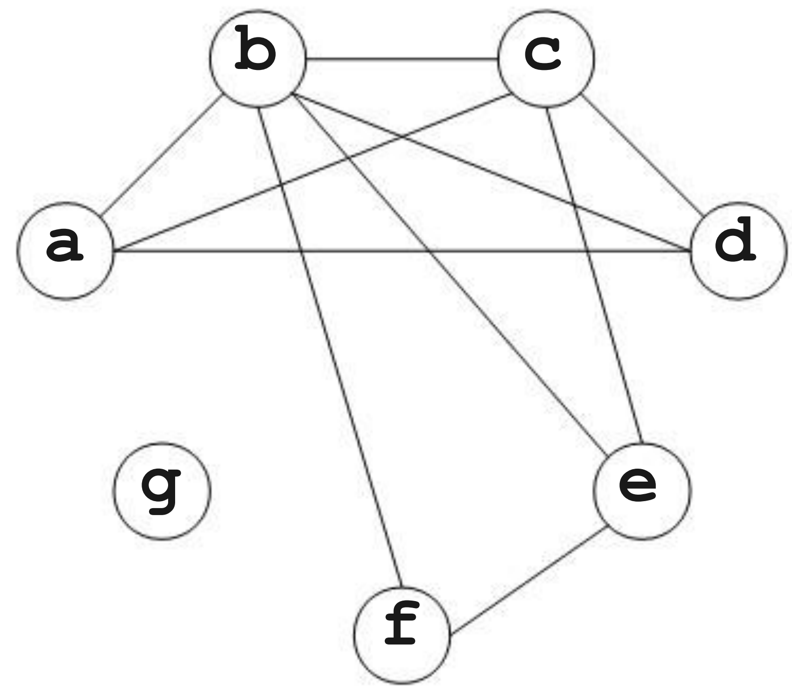




Smarter Approach

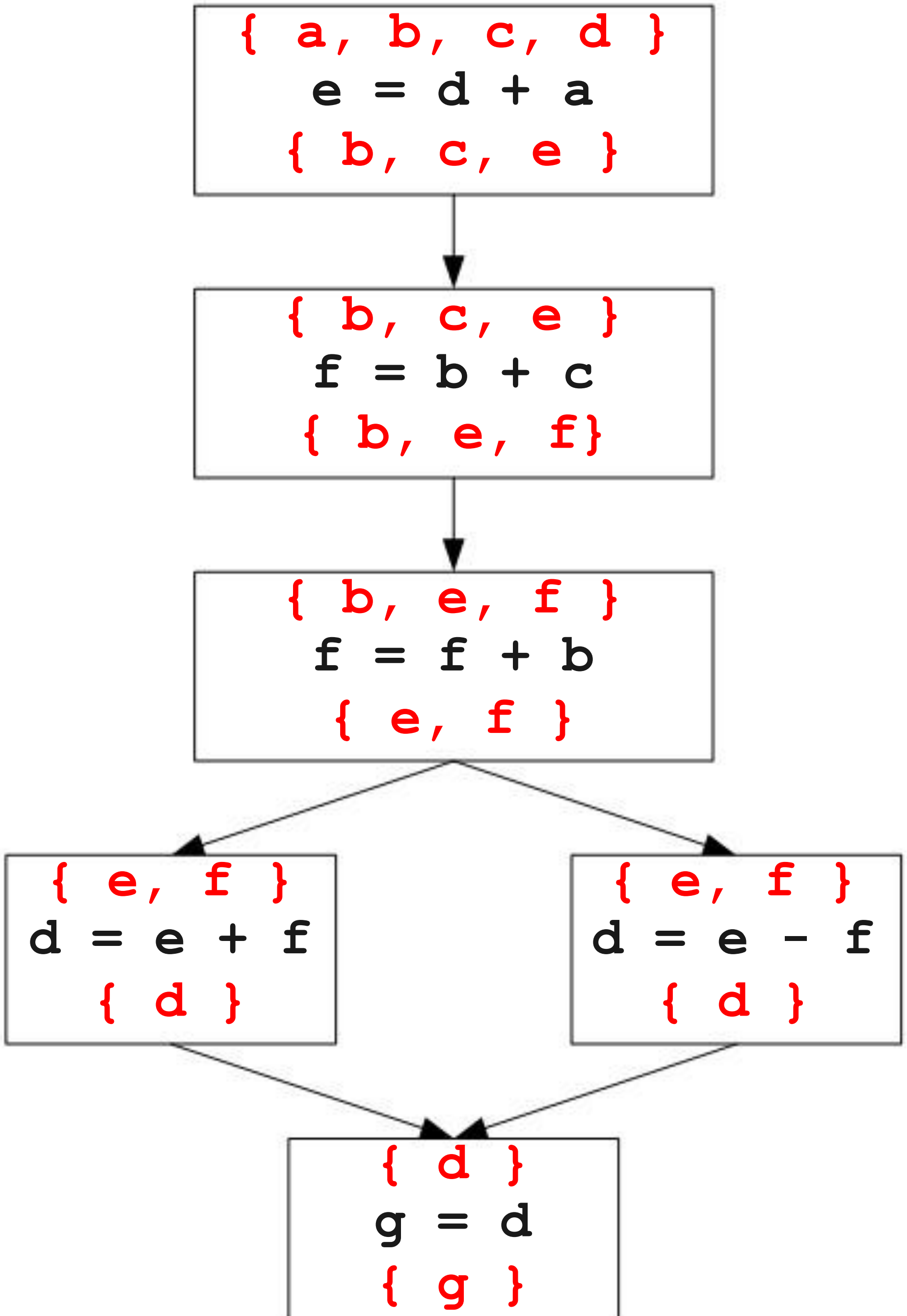


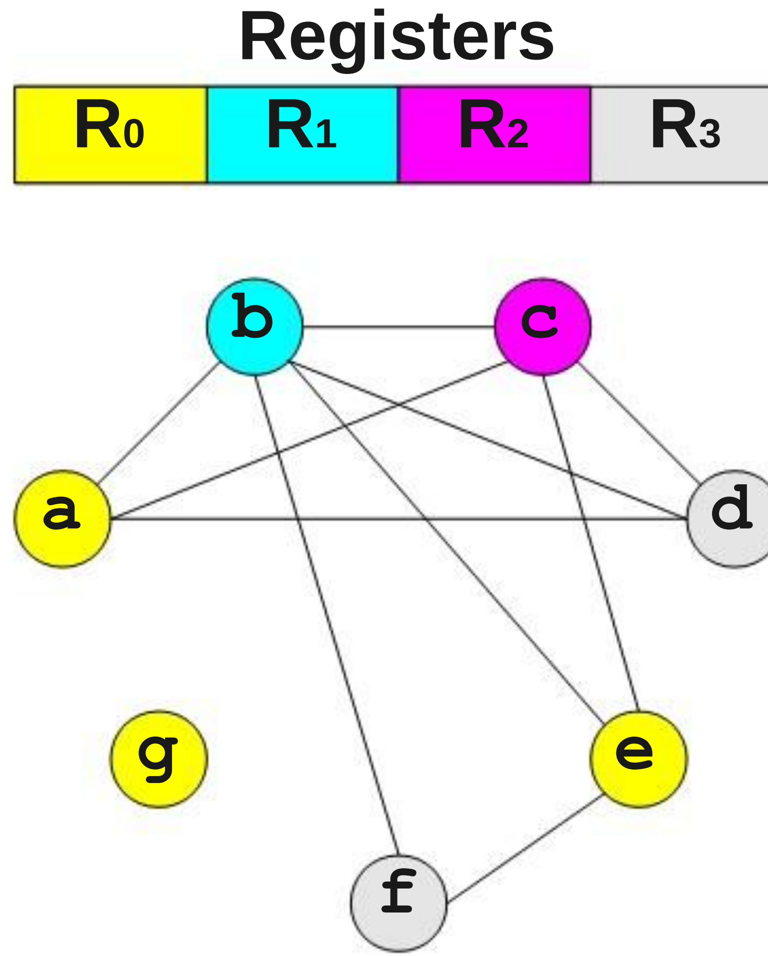
Register Interference Graph (RIG)



- Each node is a variable.
- An edge between two variables means they are live at the same program point.

Smarter Approach





Nodes which are adjacent cannot be assigned to the same register.

RIG & Graph-Coloring

- This problem is equivalent to **graph-coloring** (NP hard problem).
- No good polynomial-time algorithms are known for this problem.
- We have to use a heuristic* that is good enough in practice.

*A **heuristic** is a mental shortcut that allows people to solve problems and make judgments quickly and efficiently.

Chaitin's Algorithm

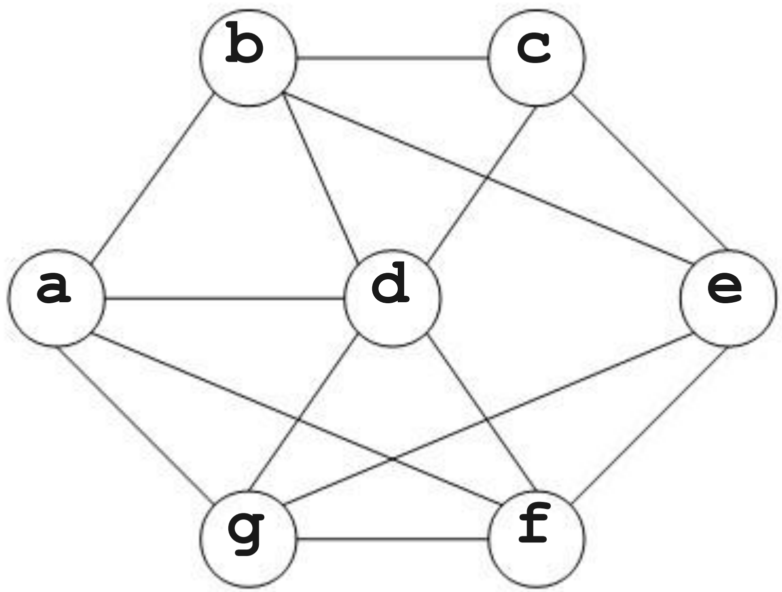
Intuition:

- Suppose we are trying to **k-color** a graph.
- Find a node with fewer than **k** edges.
- If we delete this node from the graph and color what remains, we can find a color for this node if we add it back in.

Algorithm:

- Find a node with fewer than k edges.
- Remove it from the graph.
- Recursively color the rest of the graph.
- Add the node back in.
- Assign it a valid color.

Chaitin's Algorithm

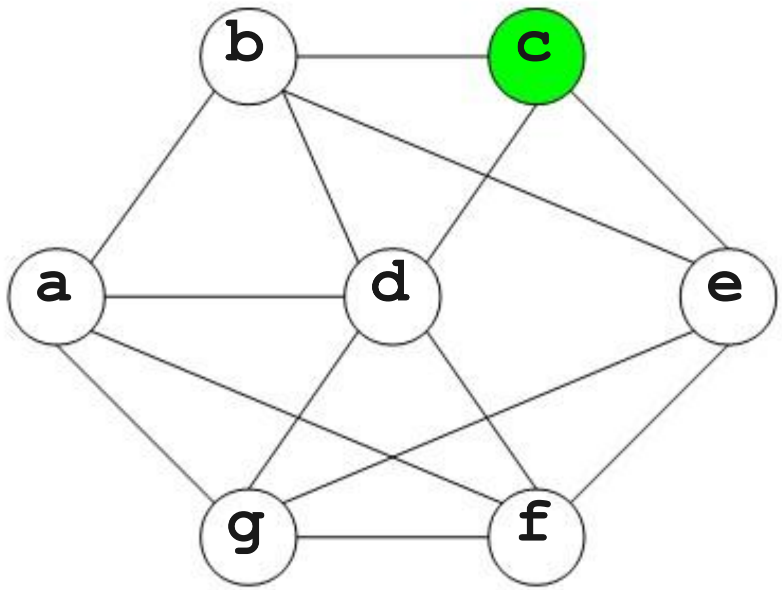


Registers

R_0	R_1	R_2	R_3

4-color

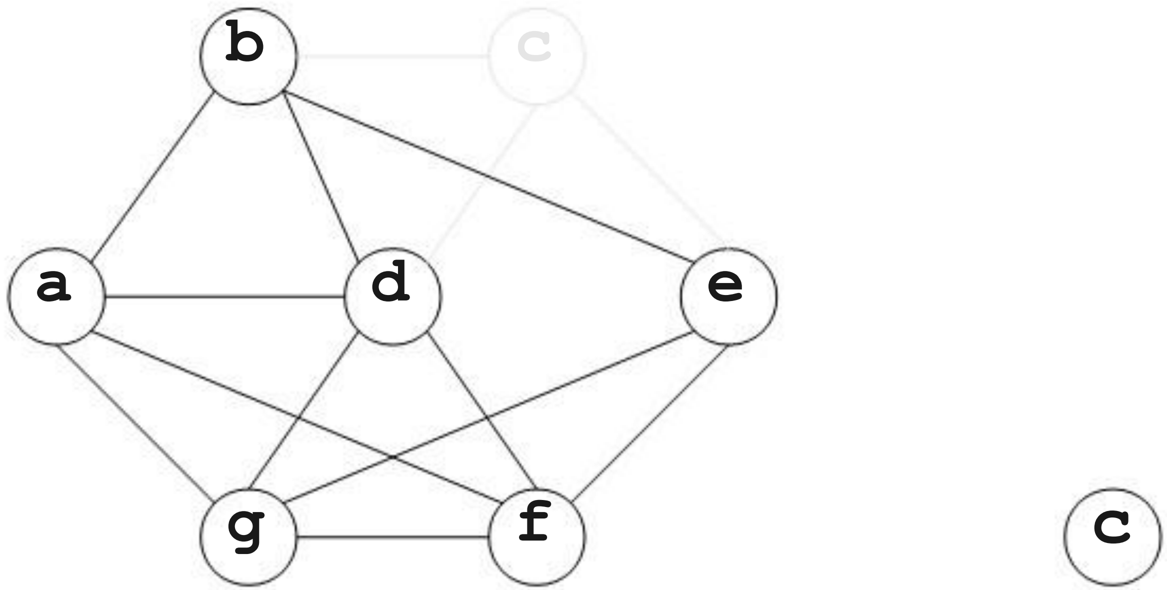
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

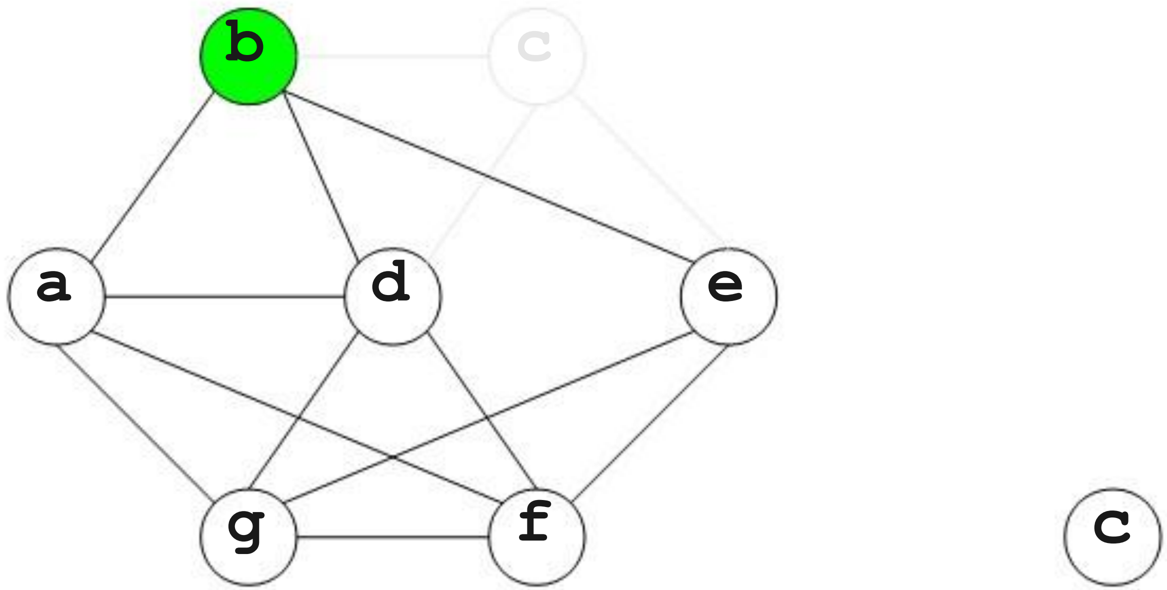
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

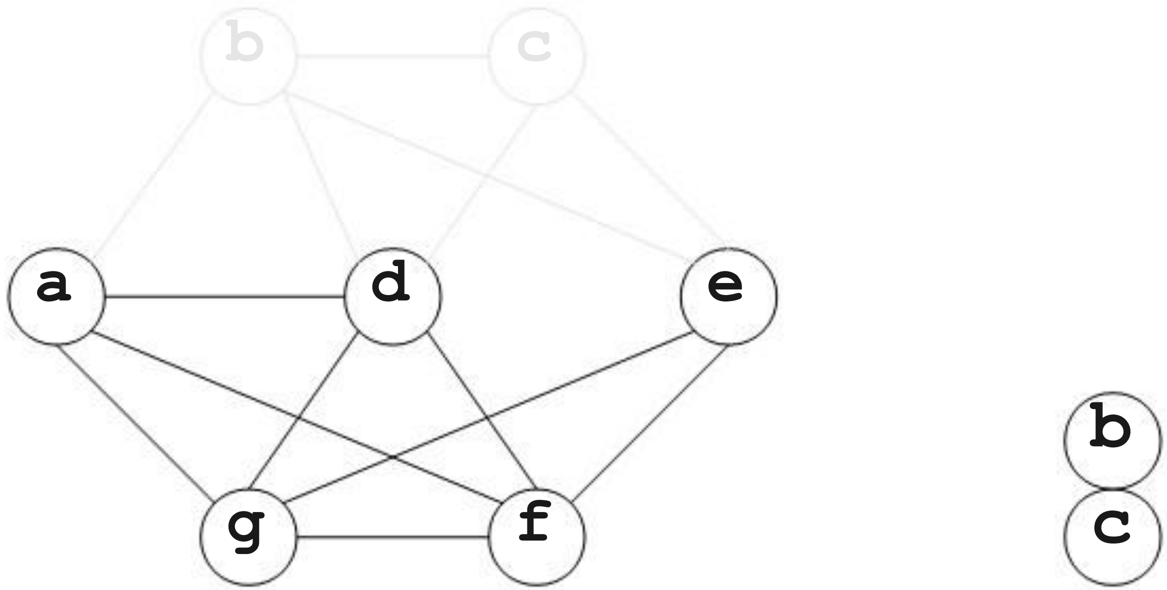
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

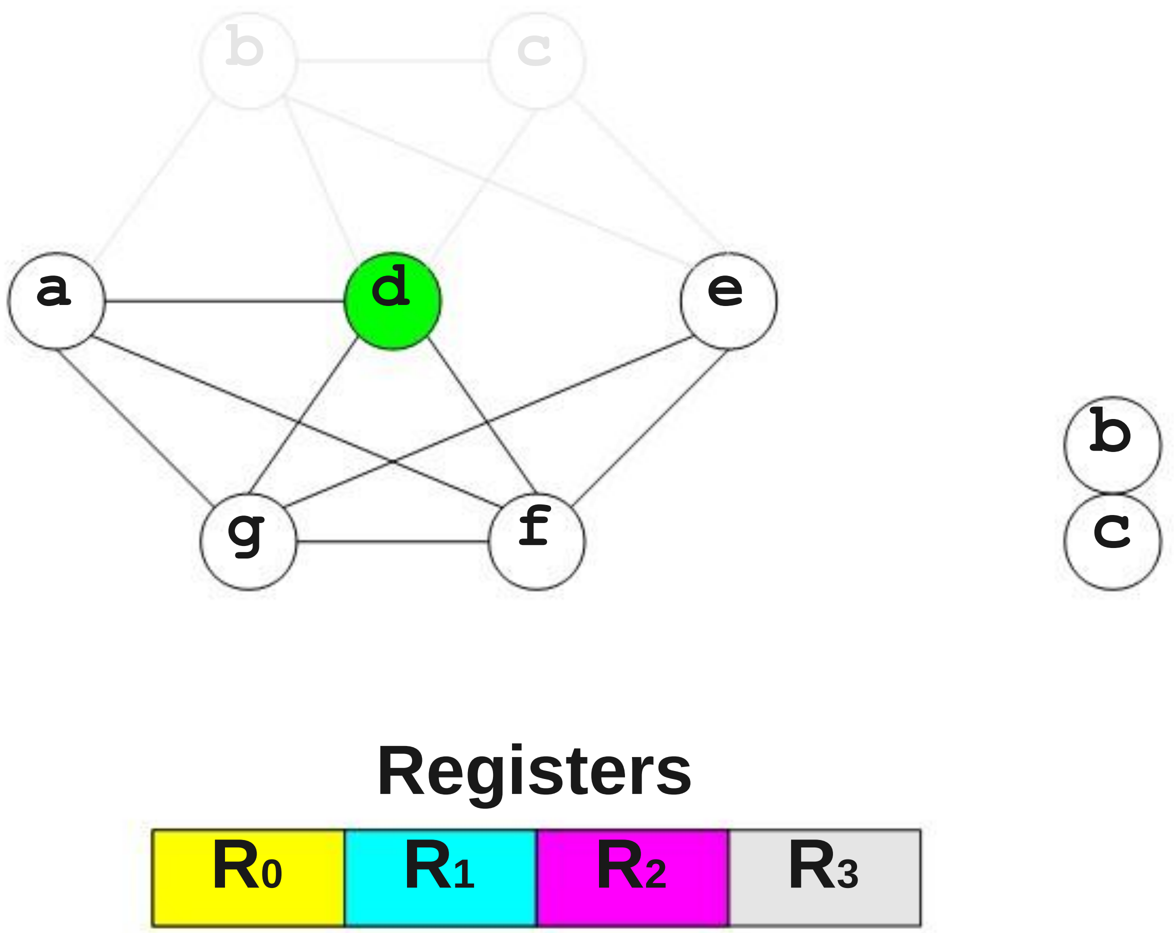
Chaitin's Algorithm



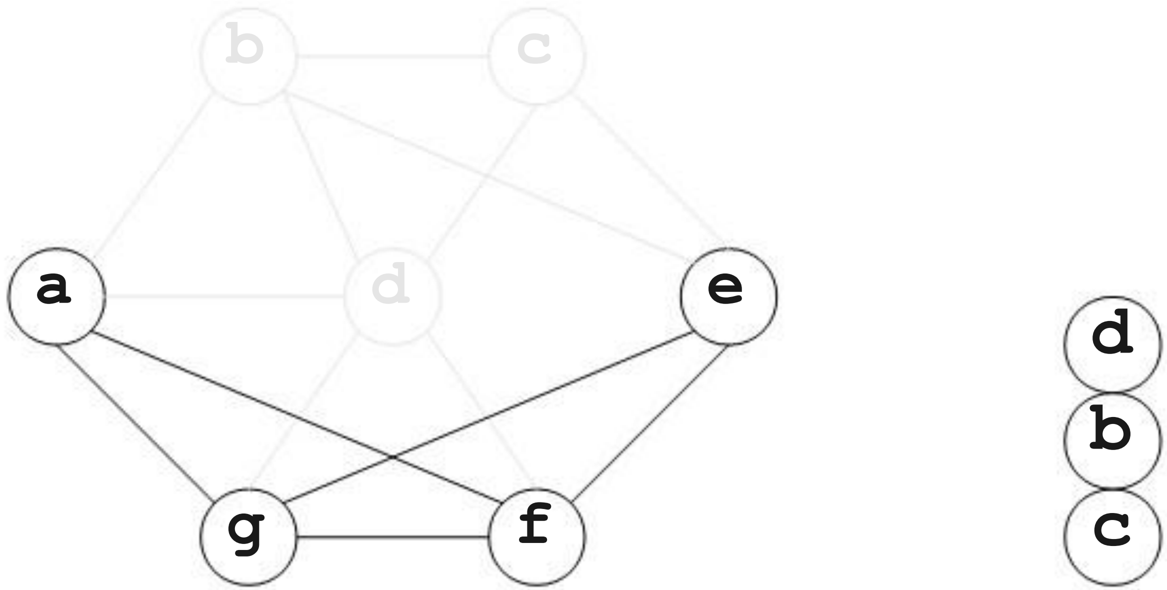
Registers

R_0	R_1	R_2	R_3

Chaitin's Algorithm



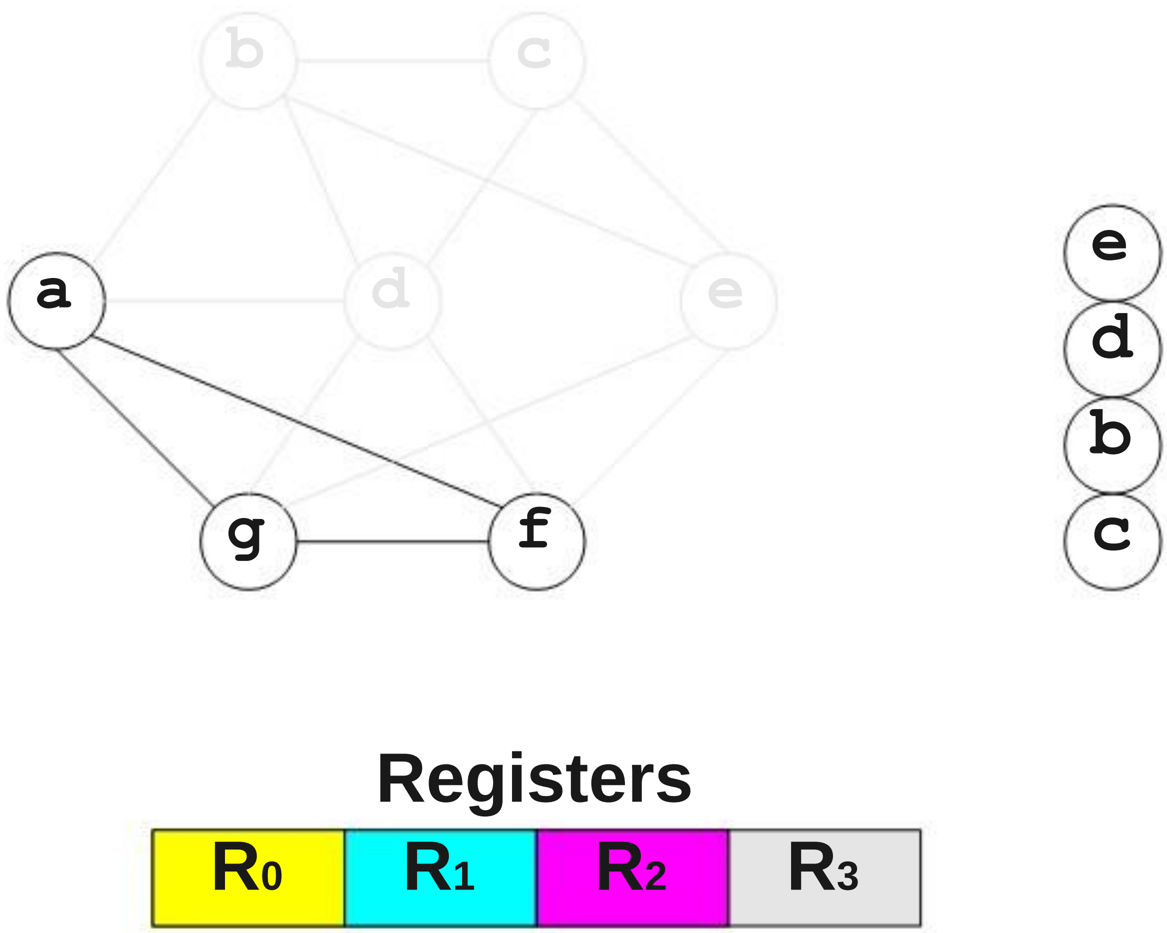
Chaitin's Algorithm



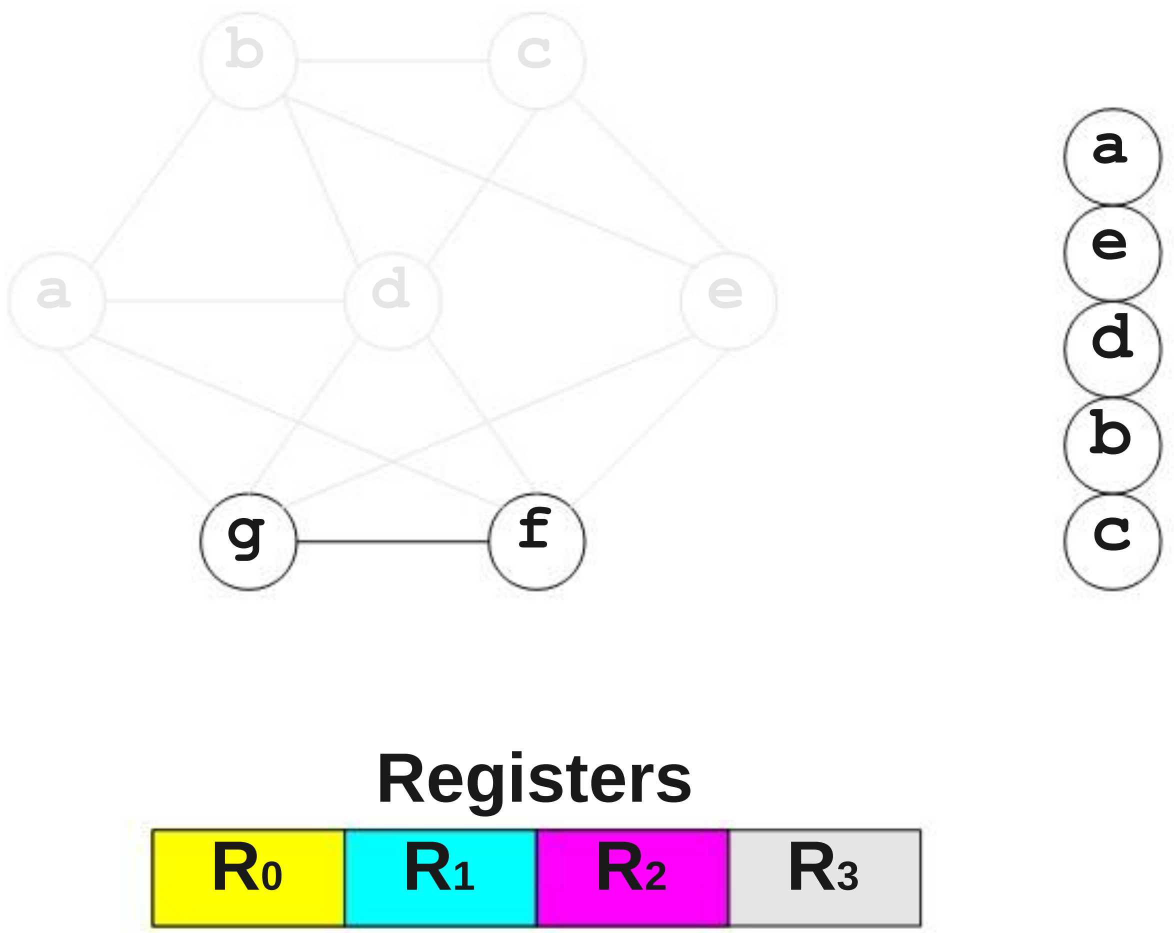
Registers

R_0	R_1	R_2	R_3

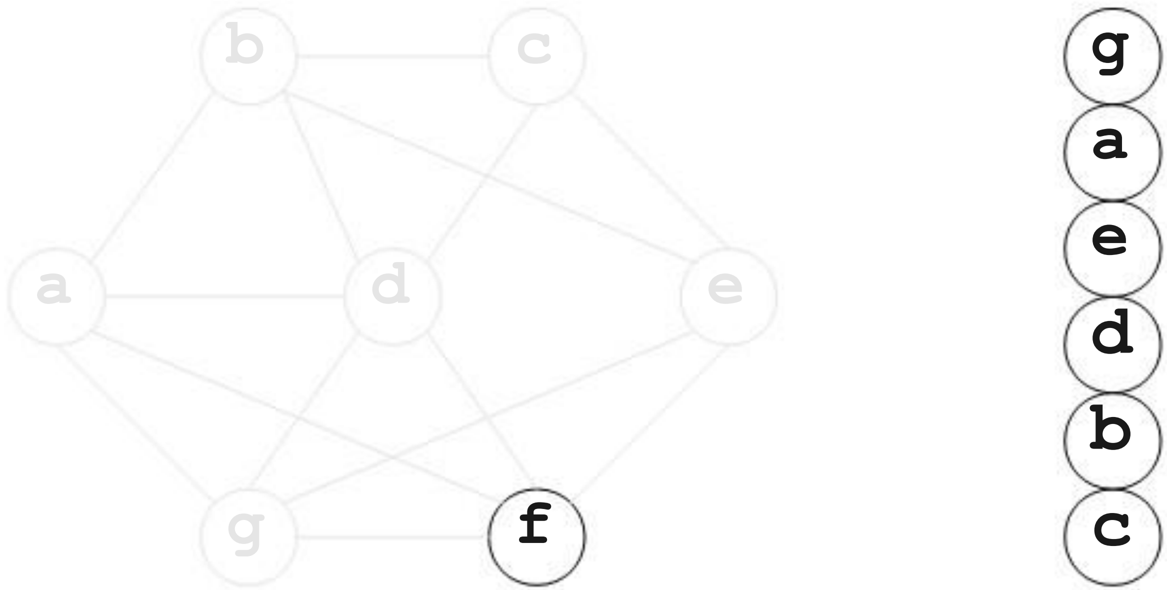
Chaitin's Algorithm



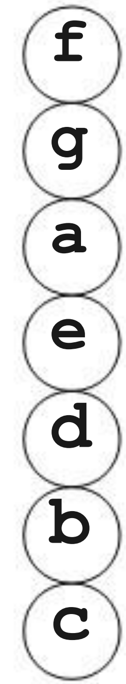
Chaitin's Algorithm



Chaitin's Algorithm



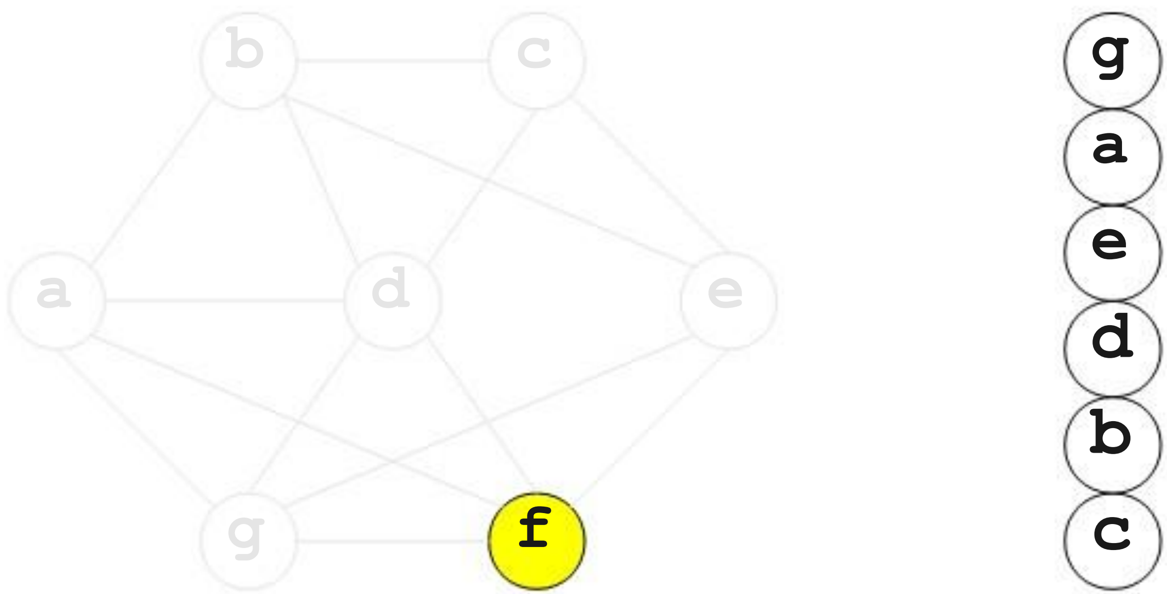
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

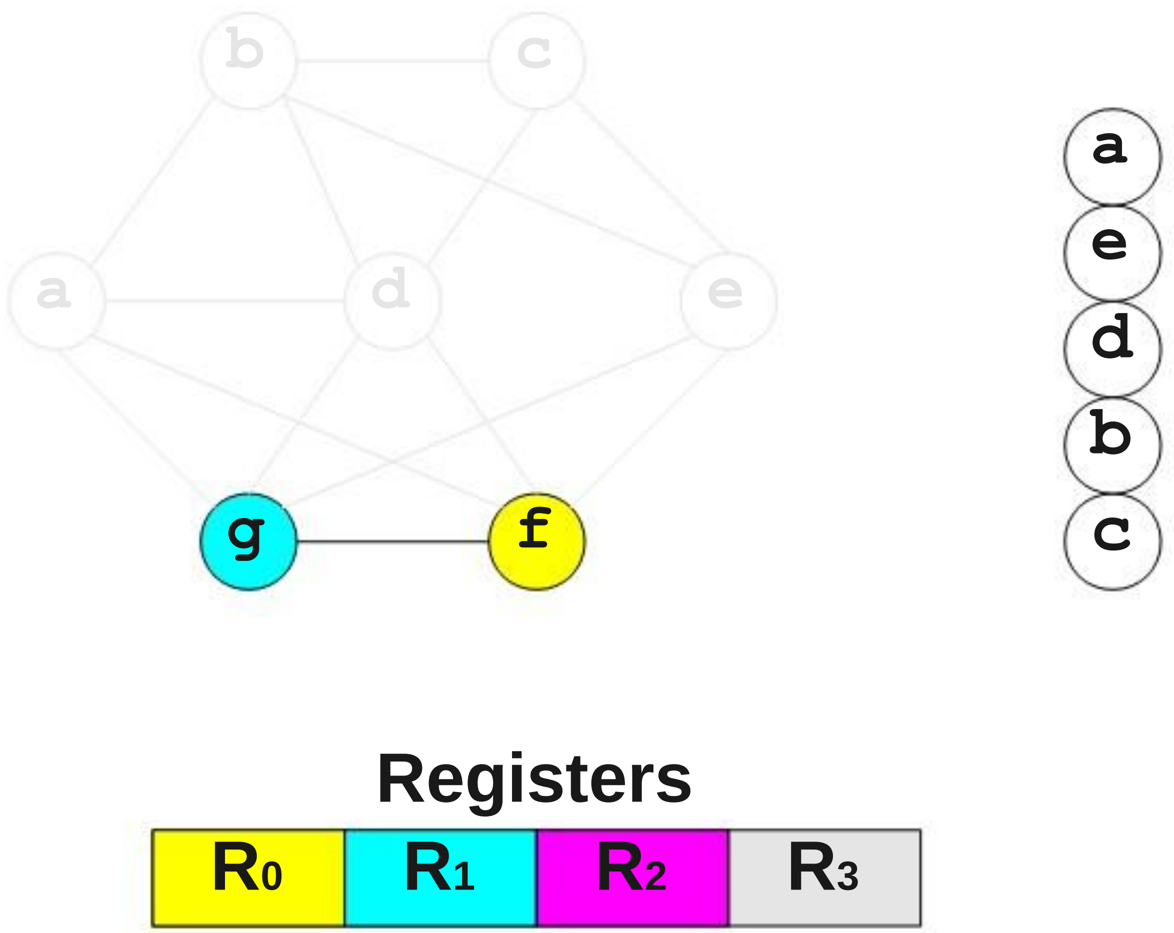
Chaitin's Algorithm



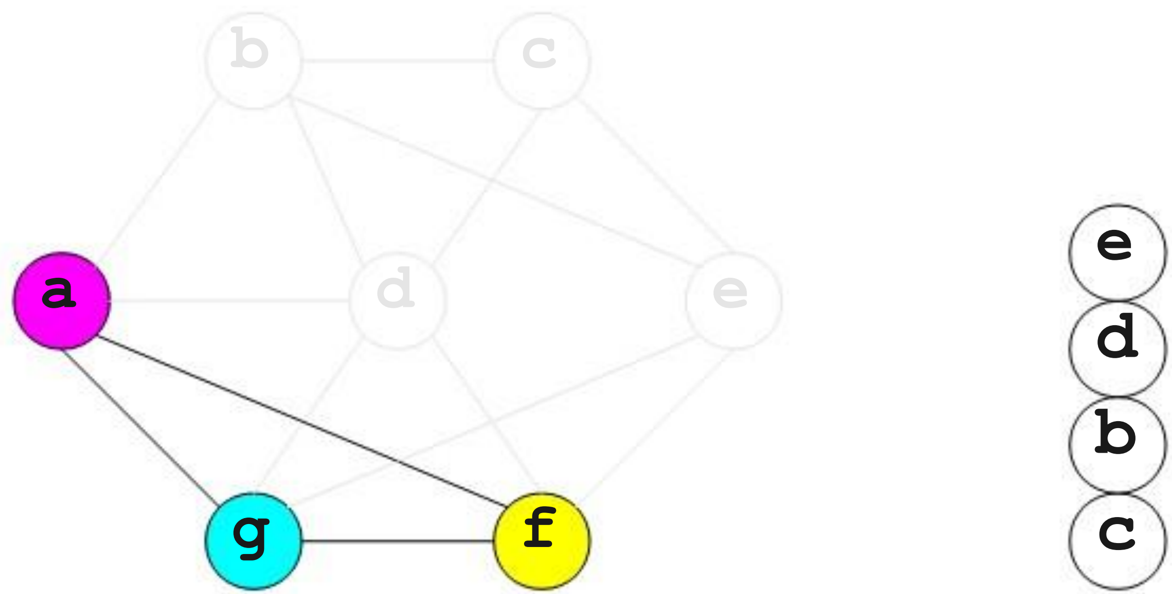
Registers

R_0	R_1	R_2	R_3

Chaitin's Algorithm



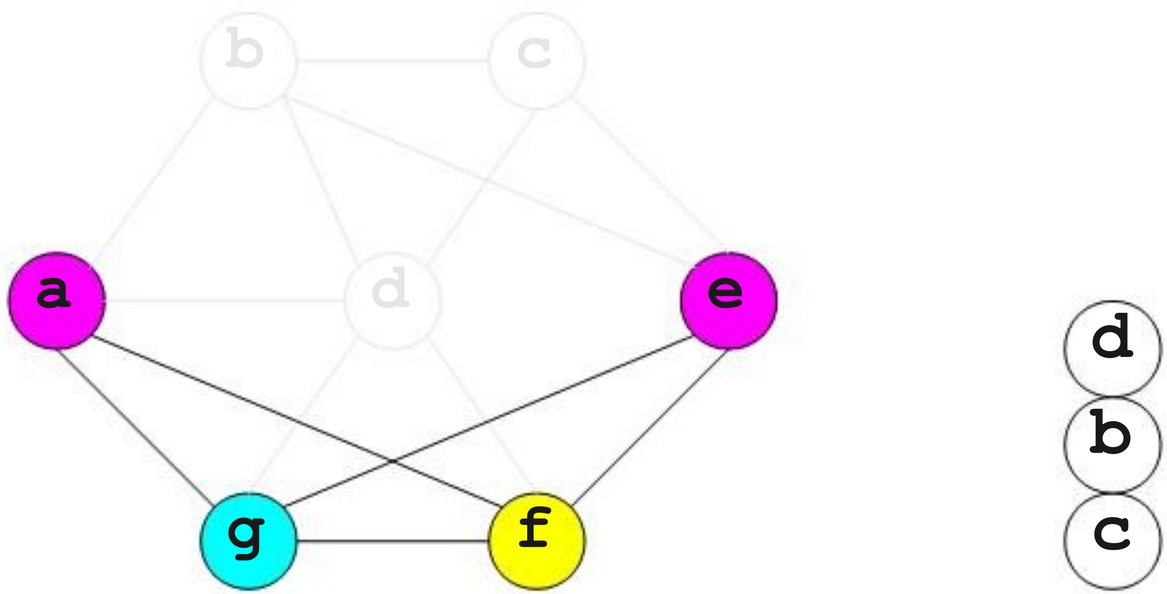
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

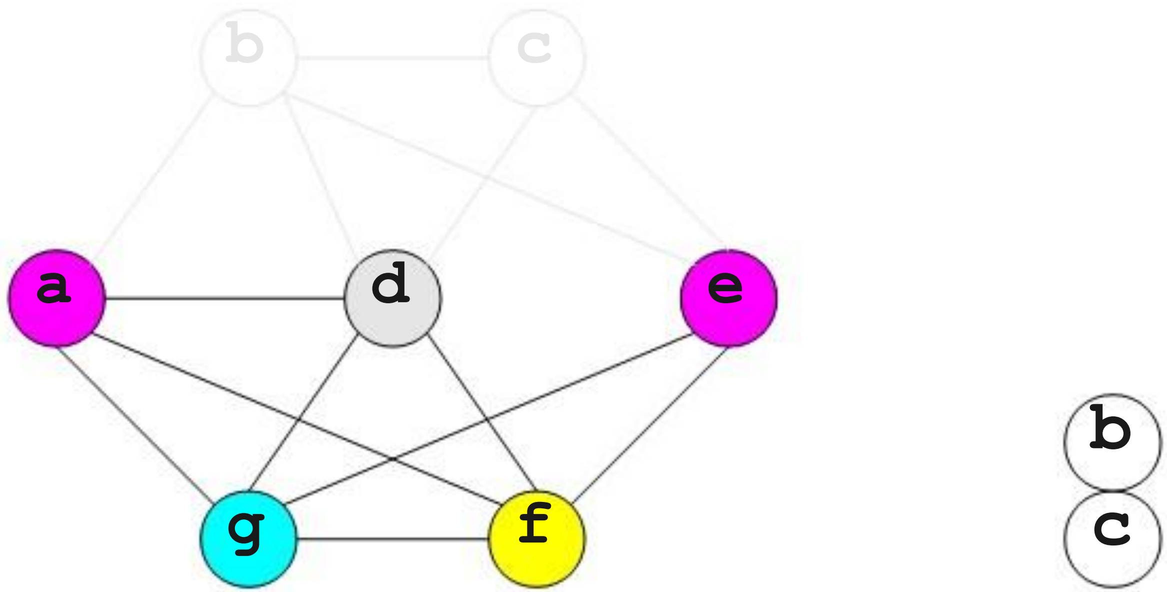
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

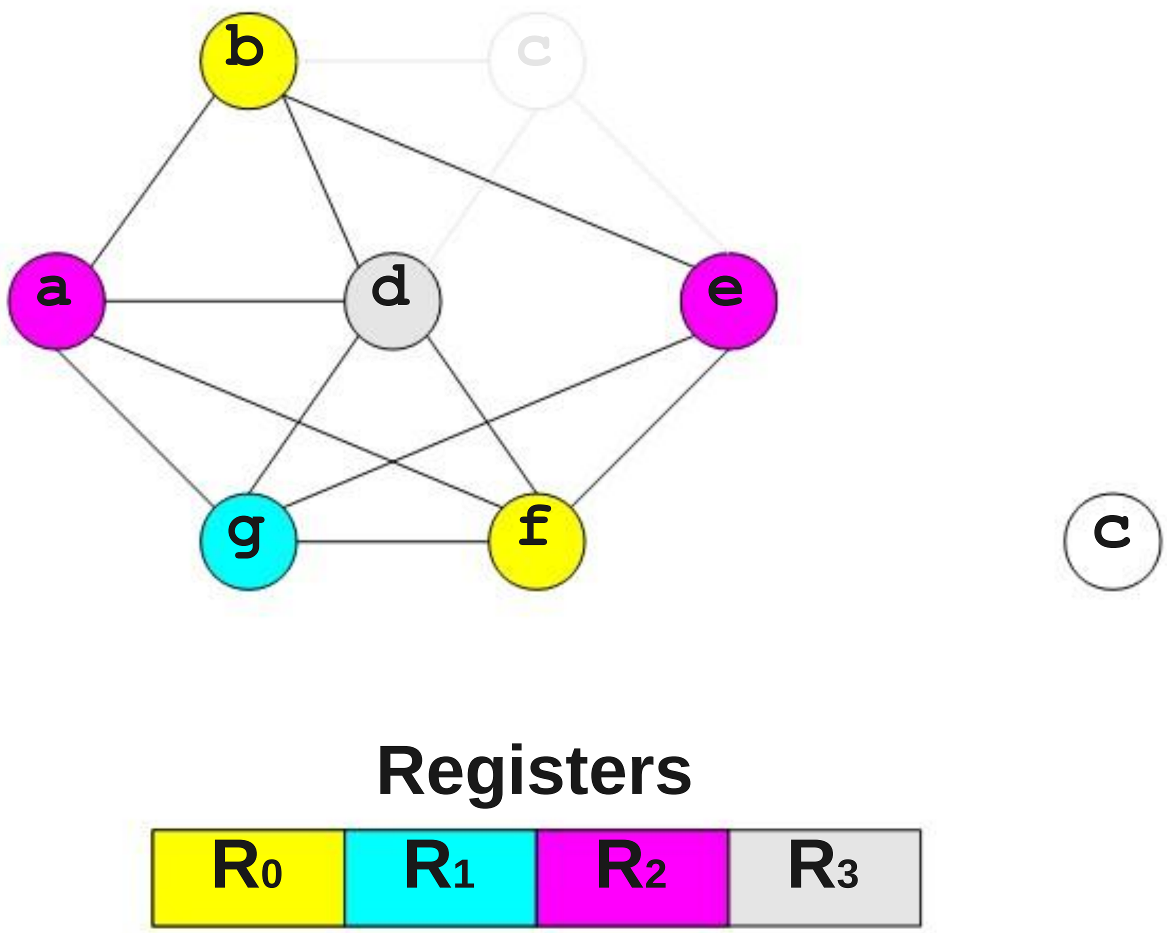
Chaitin's Algorithm



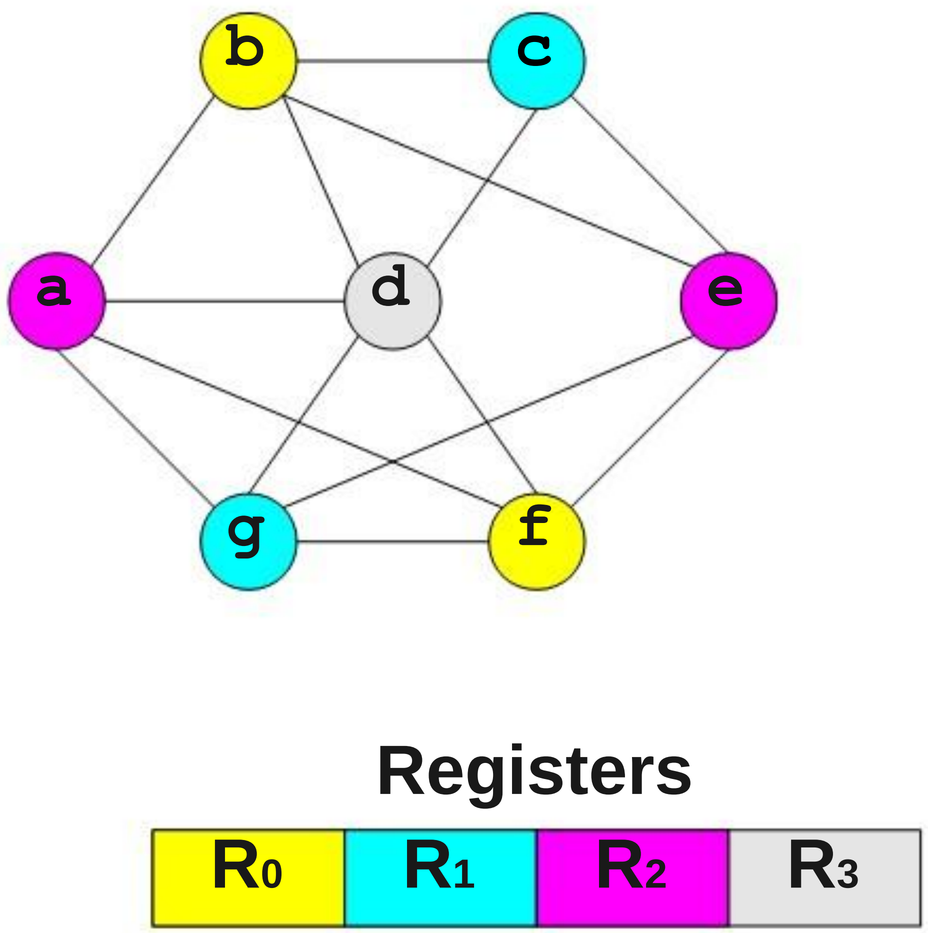
Registers

R_0	R_1	R_2	R_3

Chaitin's Algorithm



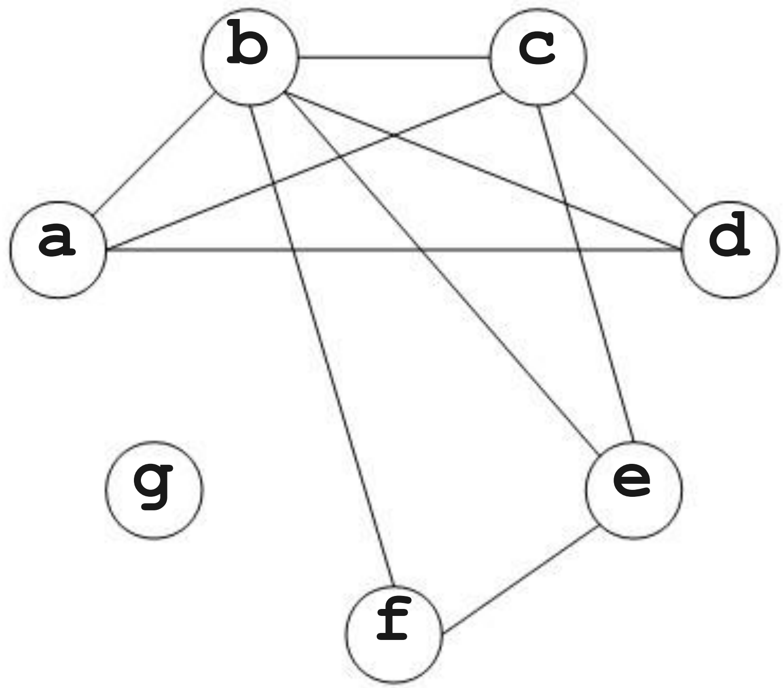
Chaitin's Algorithm



One Problem

- What if we can't find a node with < k neighbors?
- Choose and remove an arbitrary node
- When adding node back in, it may be possible to find a valid color.
- Otherwise, we have to spill that node.

Chaitin's Algorithm Reloaded

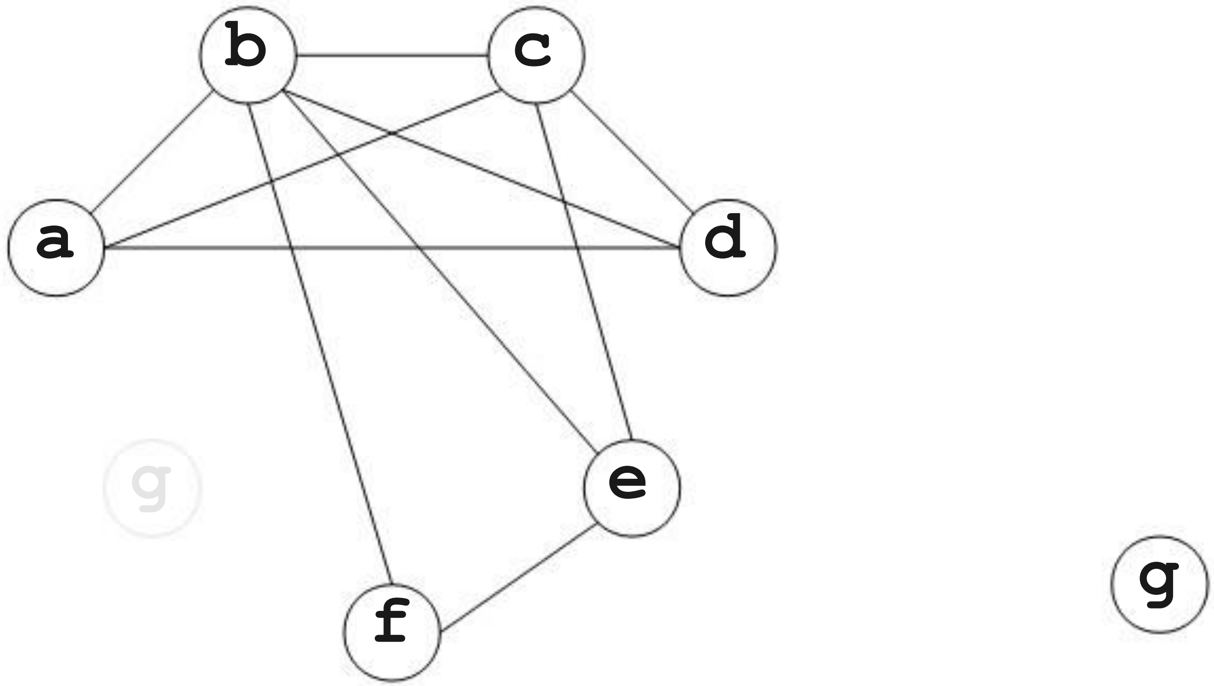


Registers

R_0	R_1	R_2

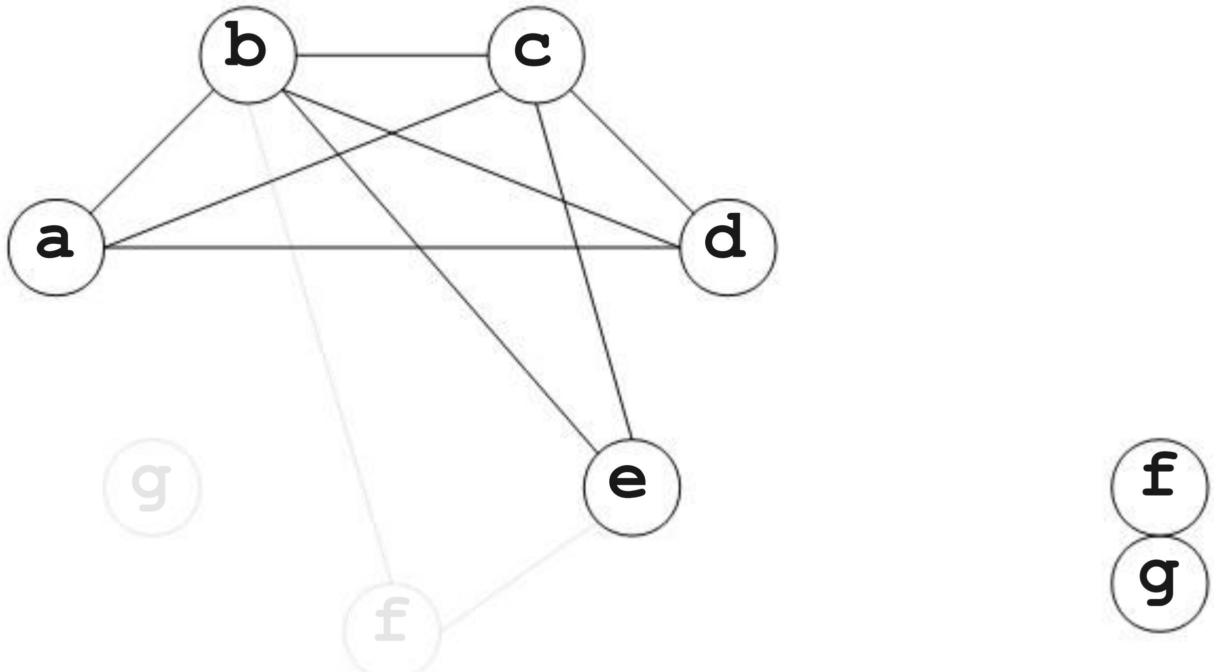
3-color

Chaitin's Algorithm Reloaded



Registers

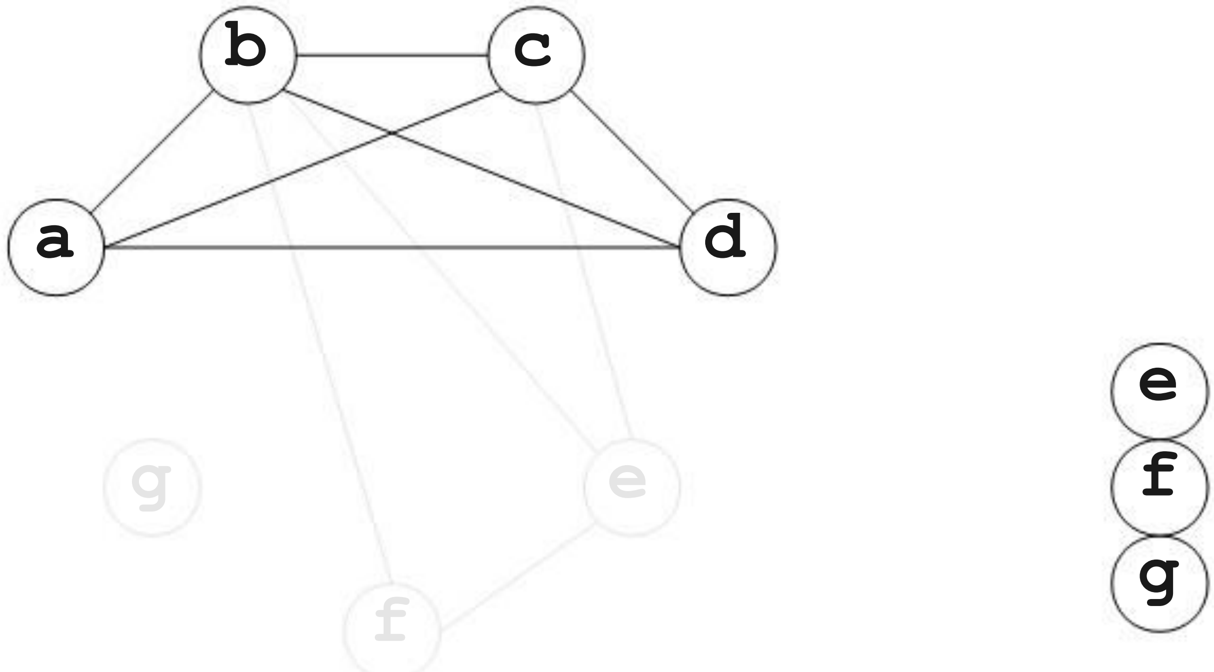
Chaitin's Algorithm Reloaded



Registers

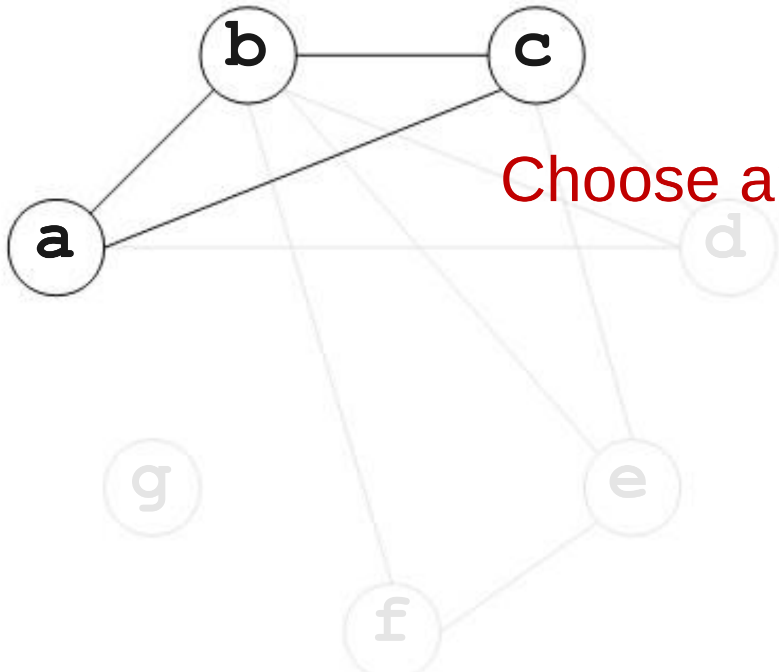
R_0	R_1	R_2

Chaitin's Algorithm Reloaded

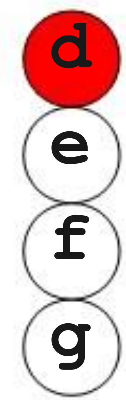


3-color

Chaitin's Algorithm Reloaded



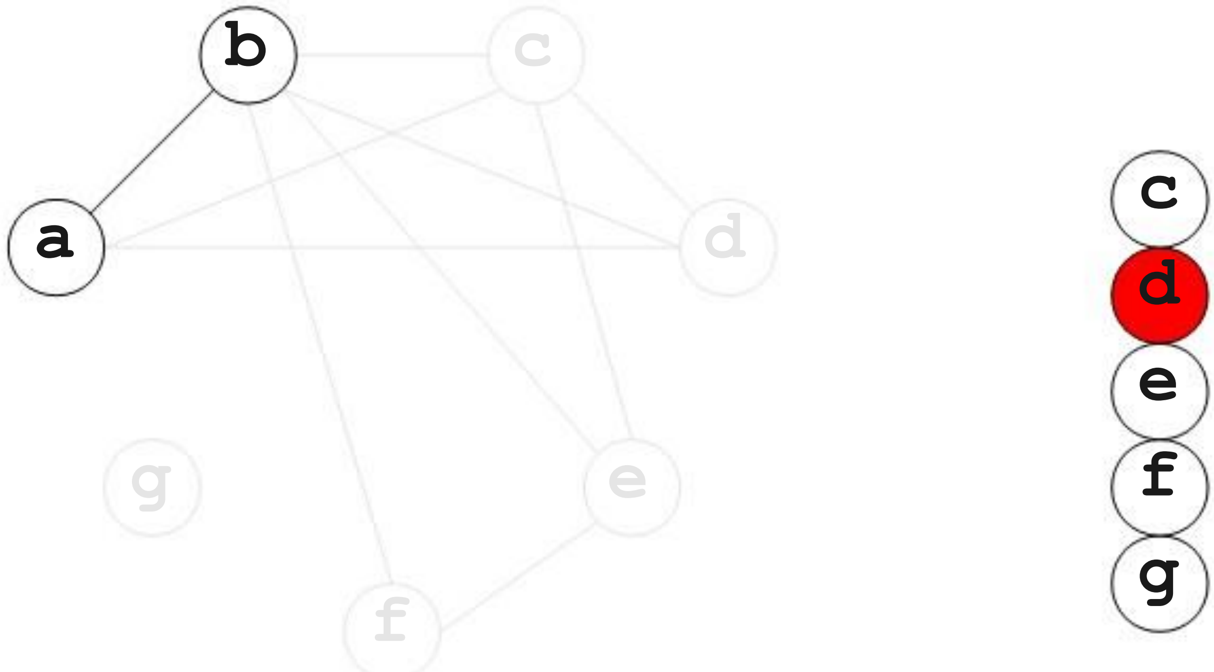
Choose and remove an arbitrary node.



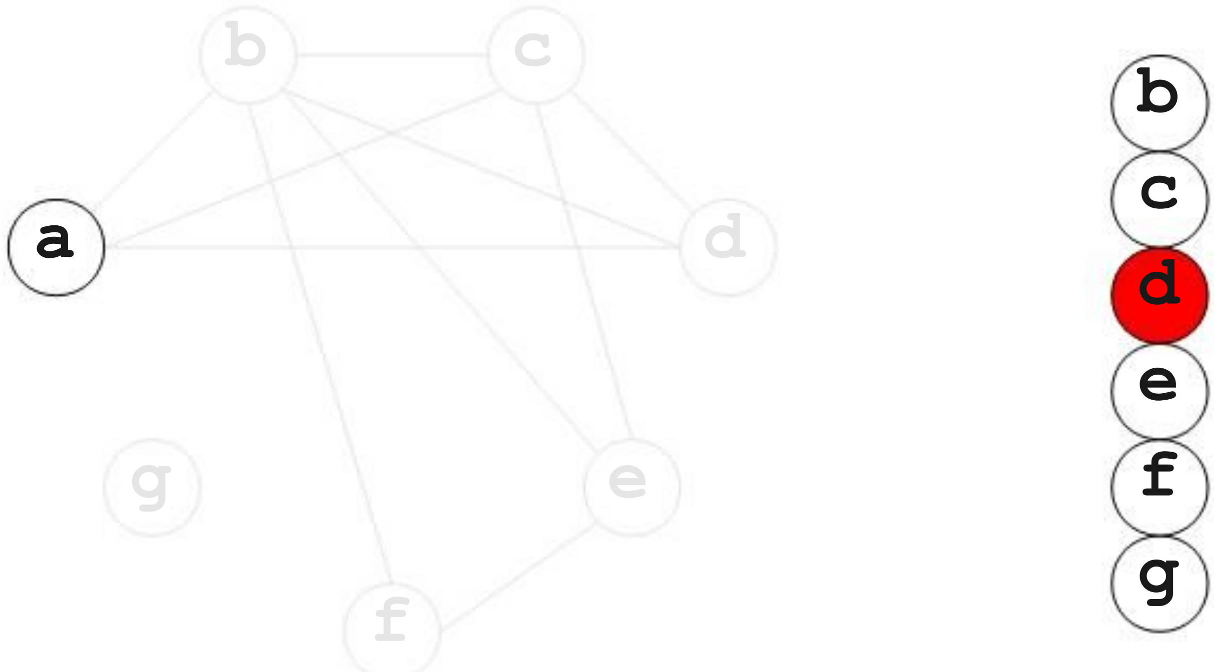
Registers

R_0	R_1	R_2

Chaitin's Algorithm Reloaded

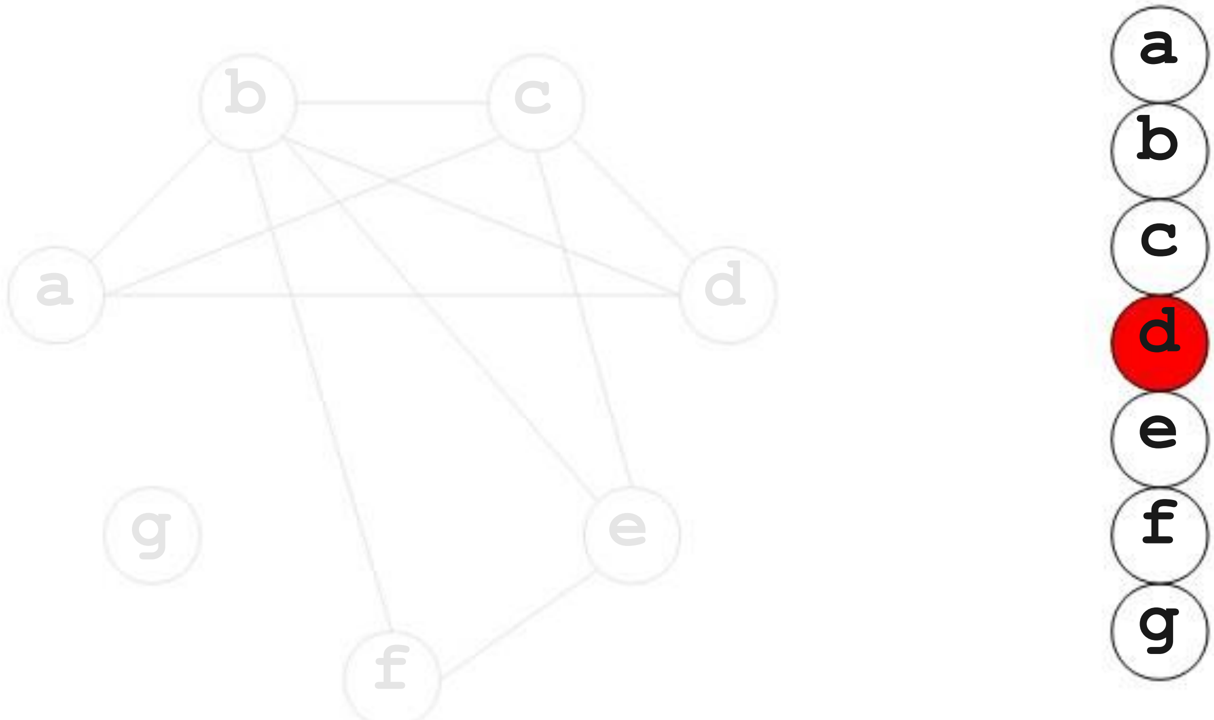


Chaitin's Algorithm Reloaded



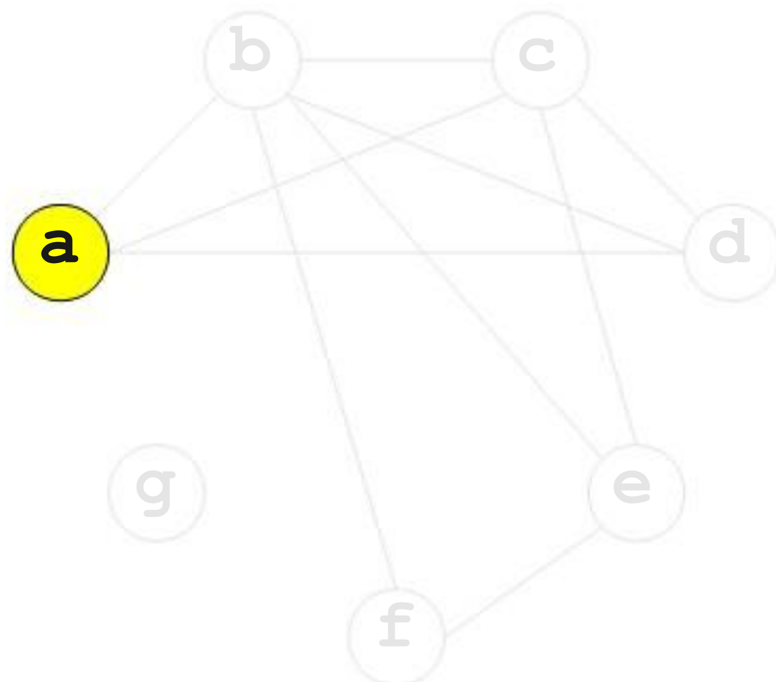
Registers

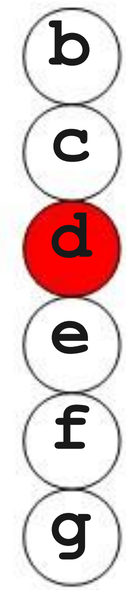
Chaitin's Algorithm Reloaded



Registers

Chaitin's Algorithm Reloaded

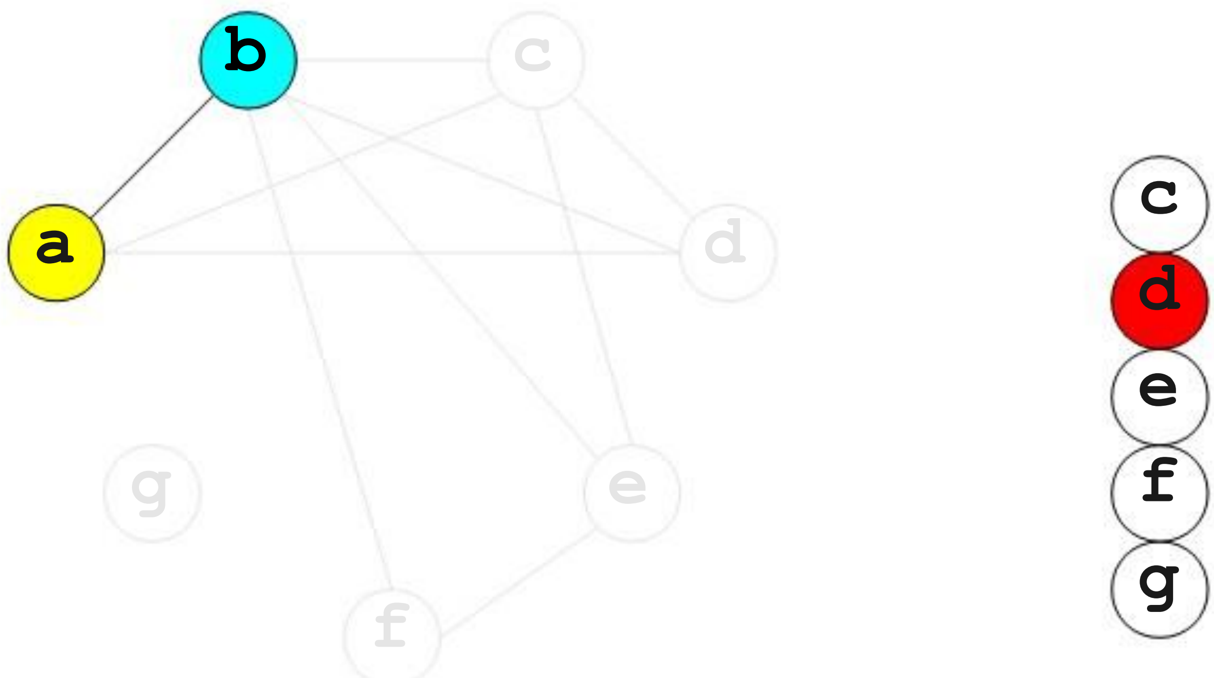




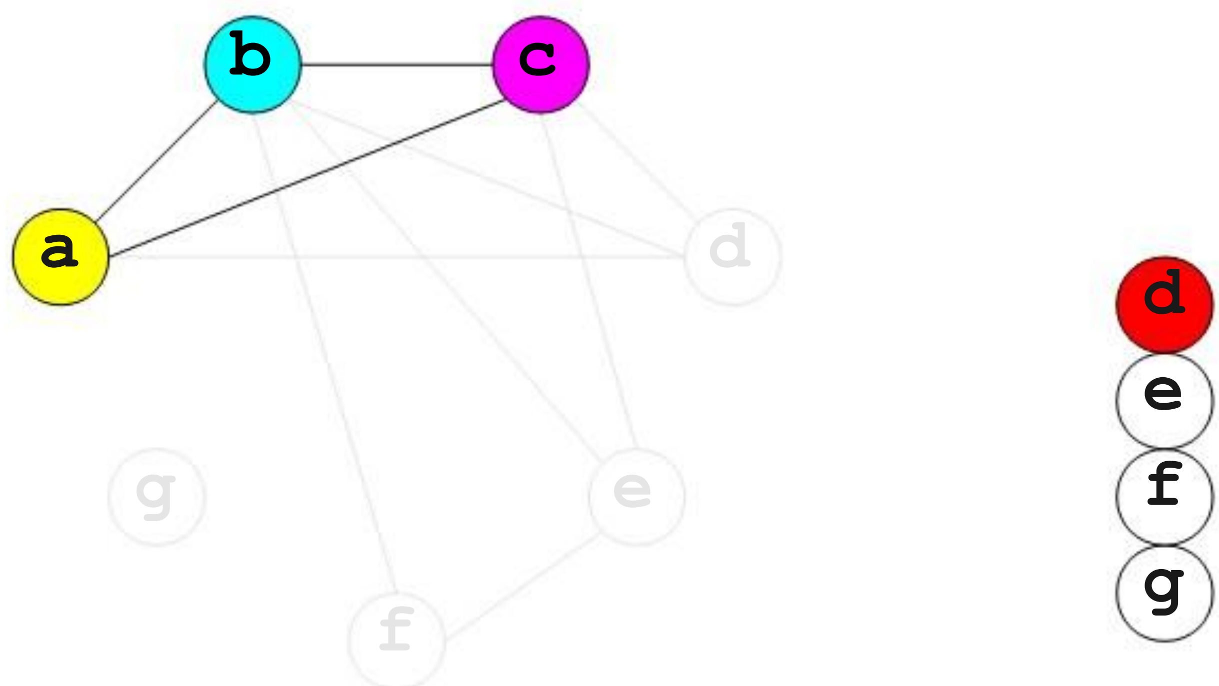
Registers

R_0	R_1	R_2

Chaitin's Algorithm Reloaded



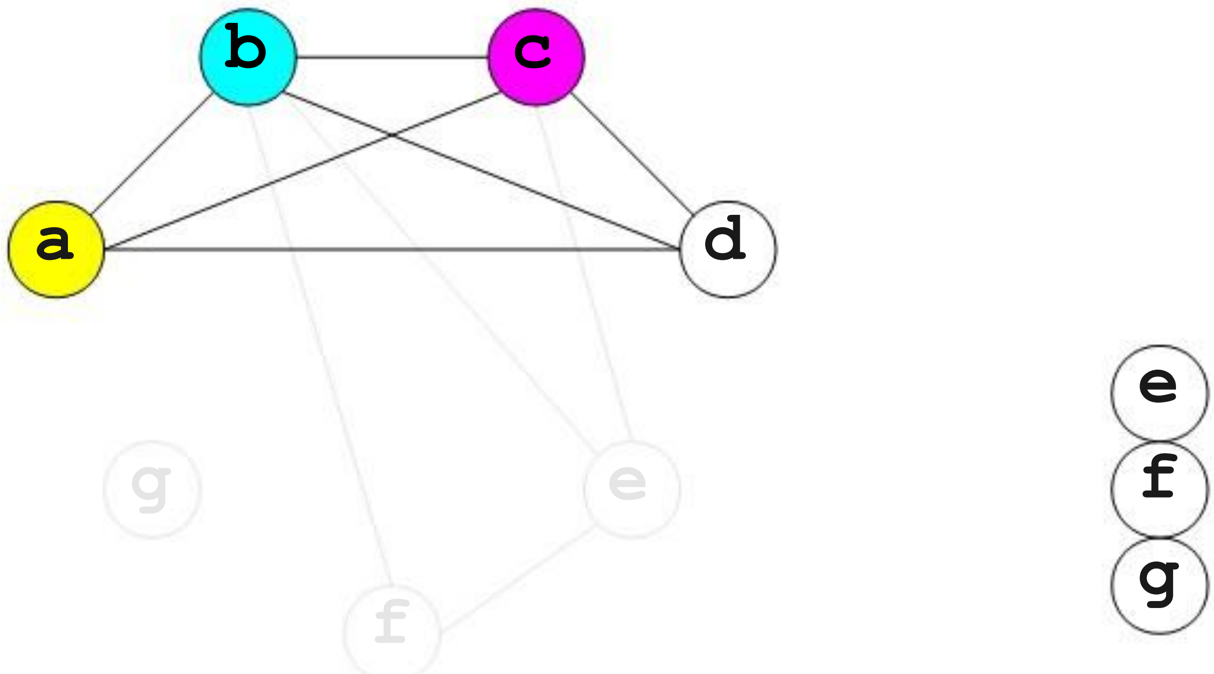
Chaitin's Algorithm Reloaded



Registers

R0	R1	R2

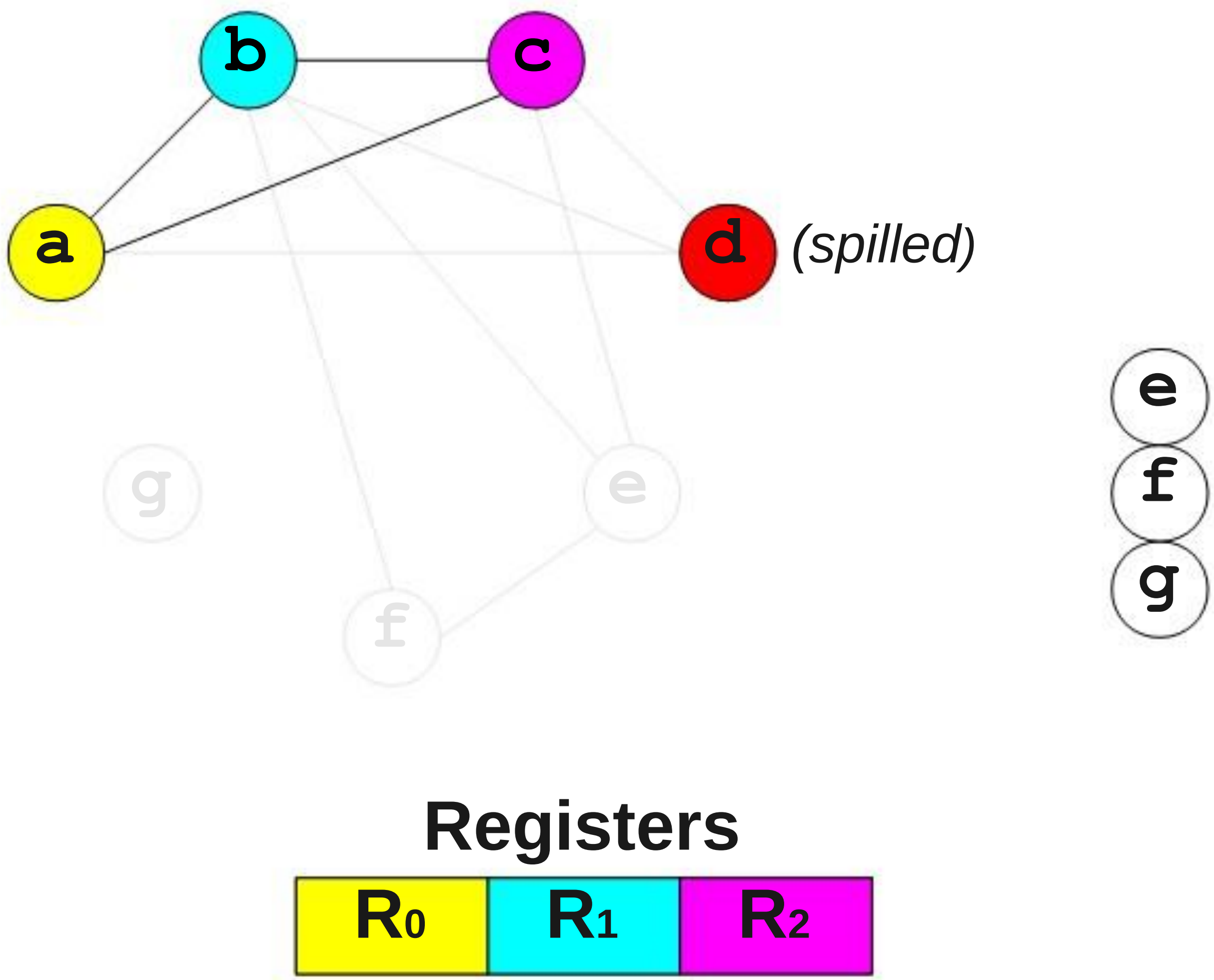
Chaitin's Algorithm Reloaded



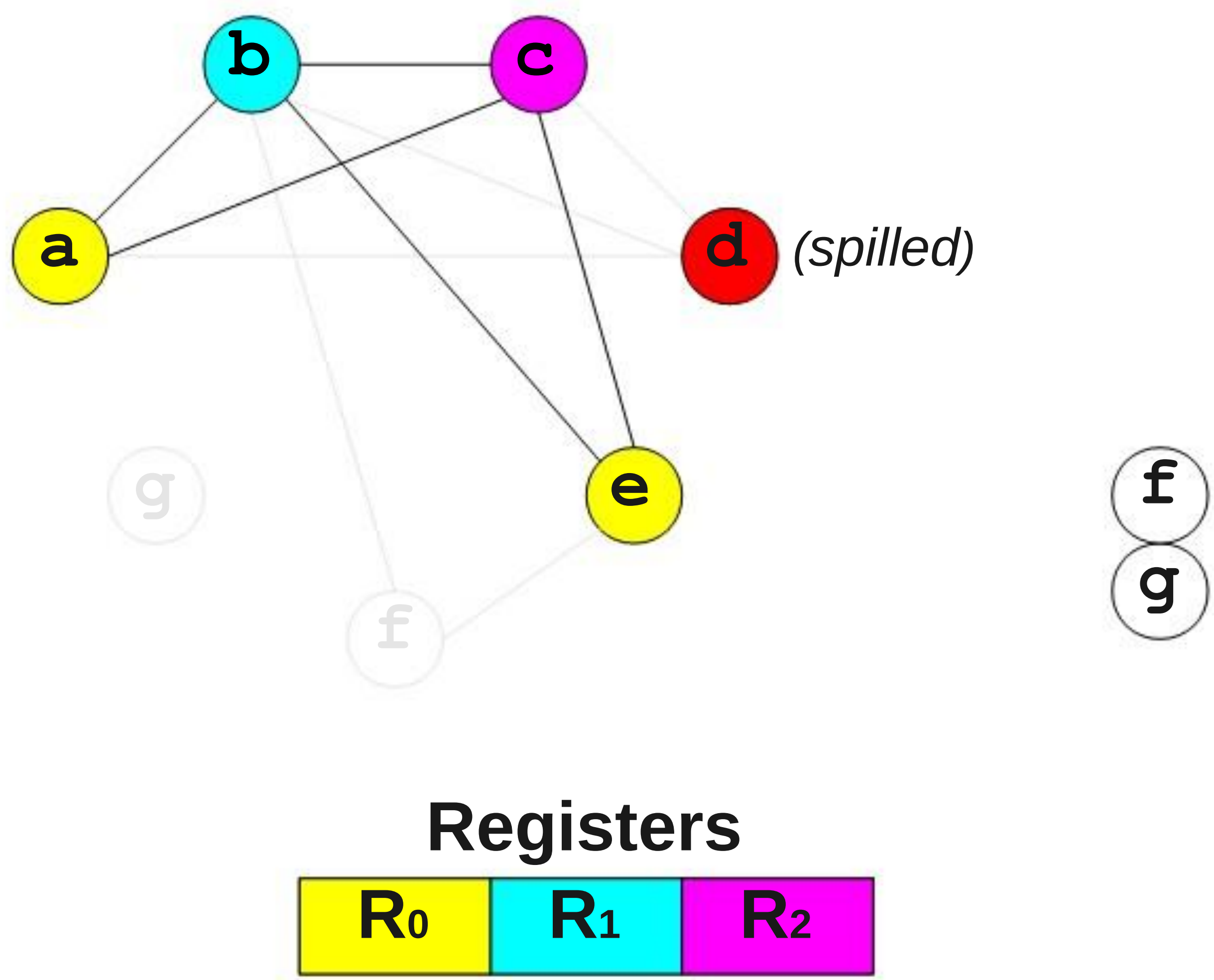
Registers

R_0	R_1	R_2

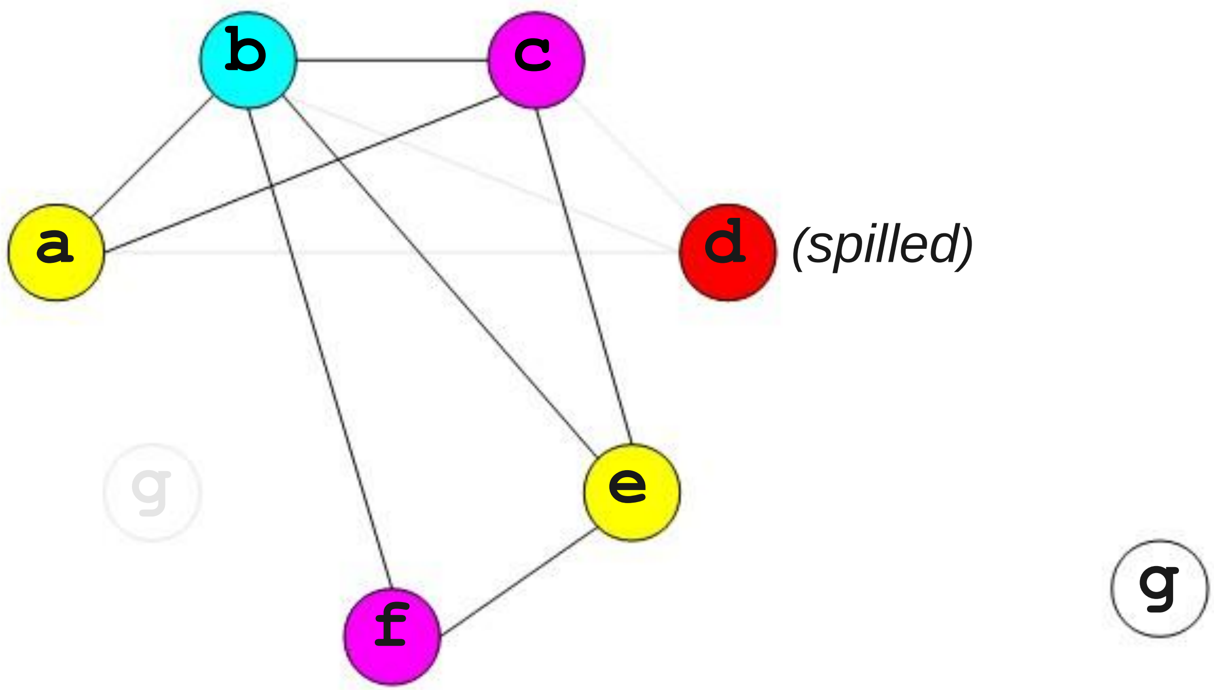
Chaitin's Algorithm Reloaded



Chaitin's Algorithm Reloaded



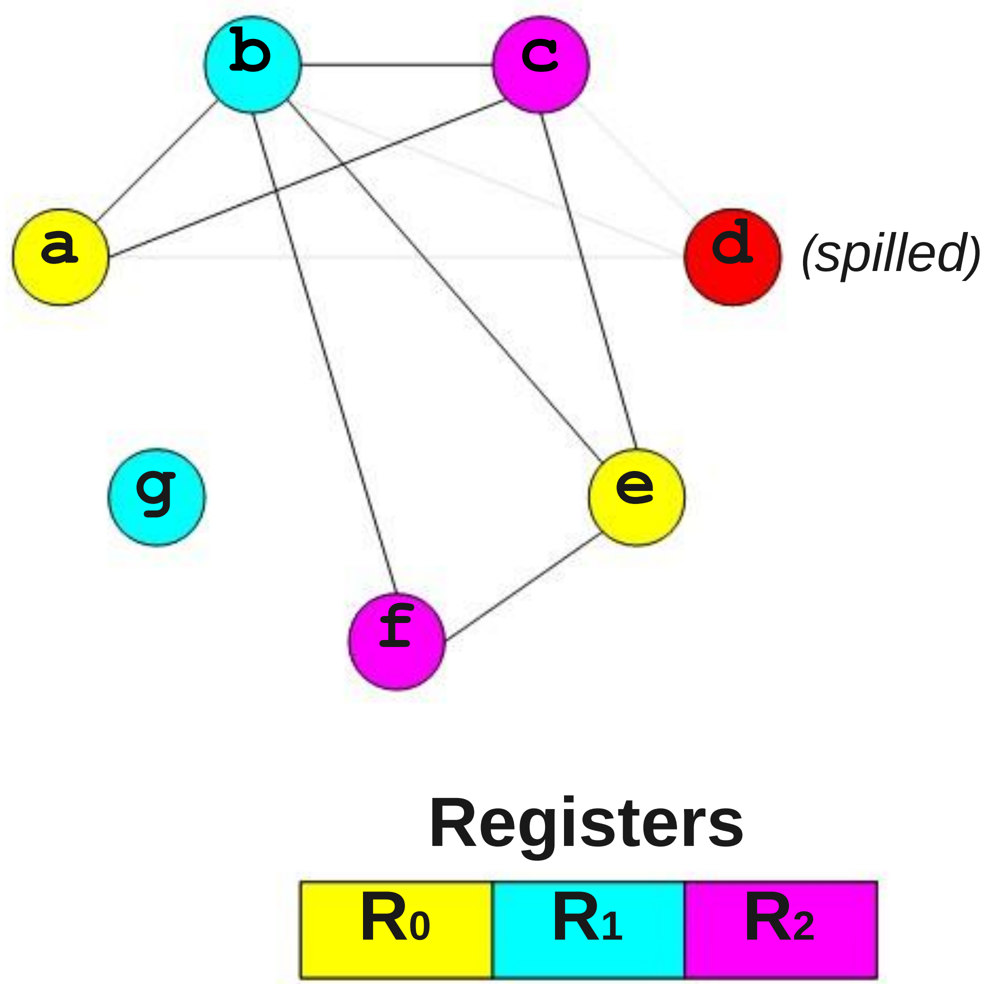
Chaitin's Algorithm Reloaded



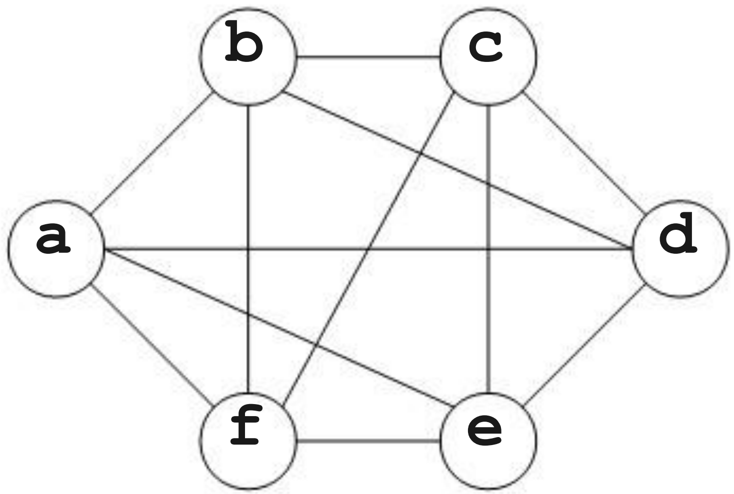
Registers

R_0	R_1	R_2

Chaitin's Algorithm Reloaded



Another Example

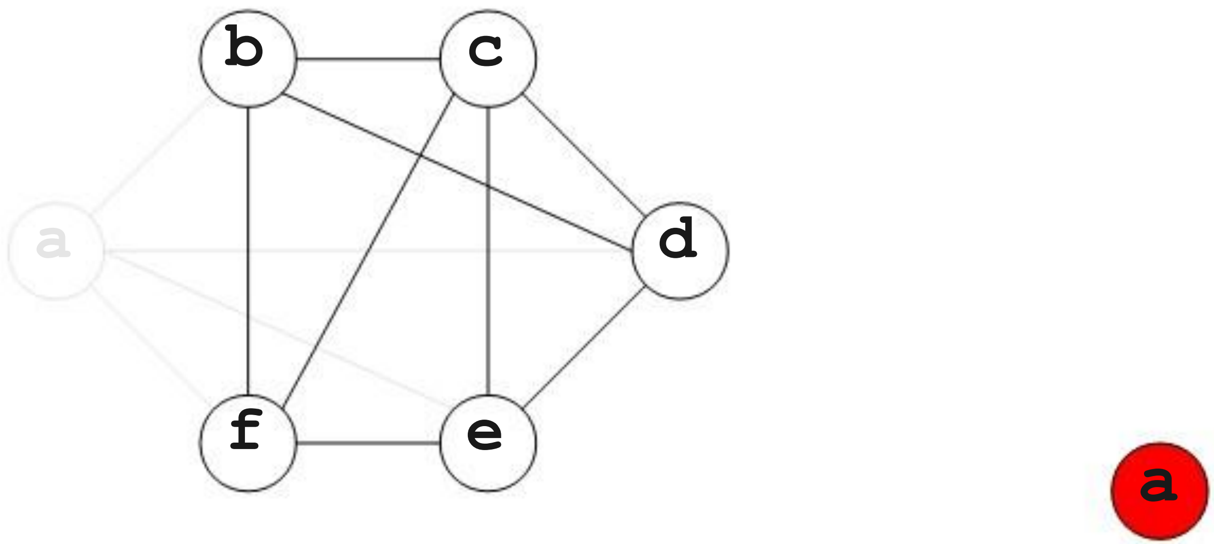


Registers

R_0	R_1	R_2

3-color

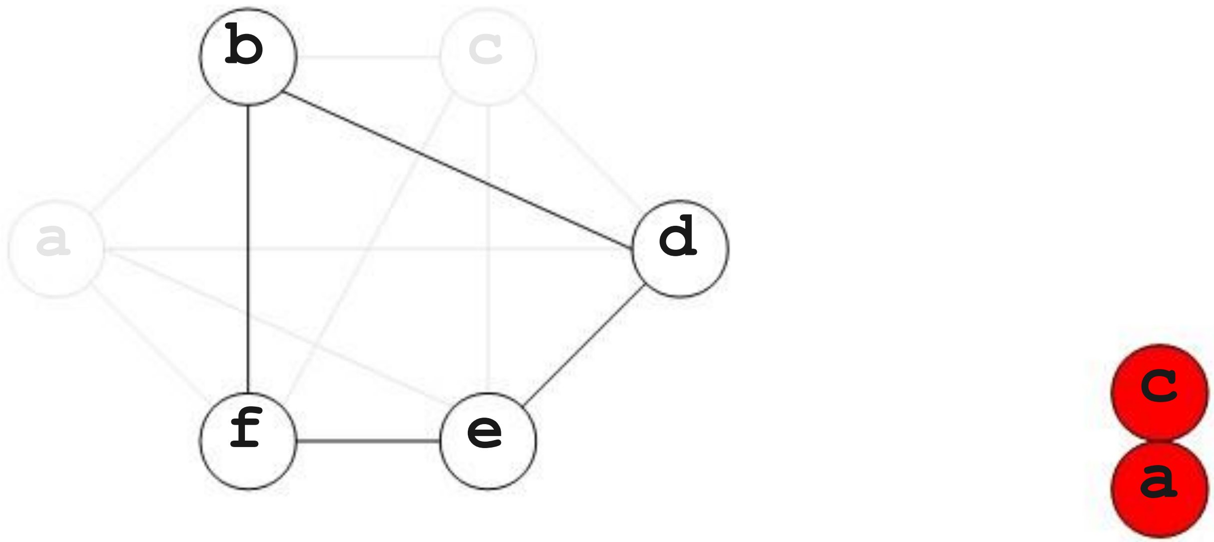
Another Example



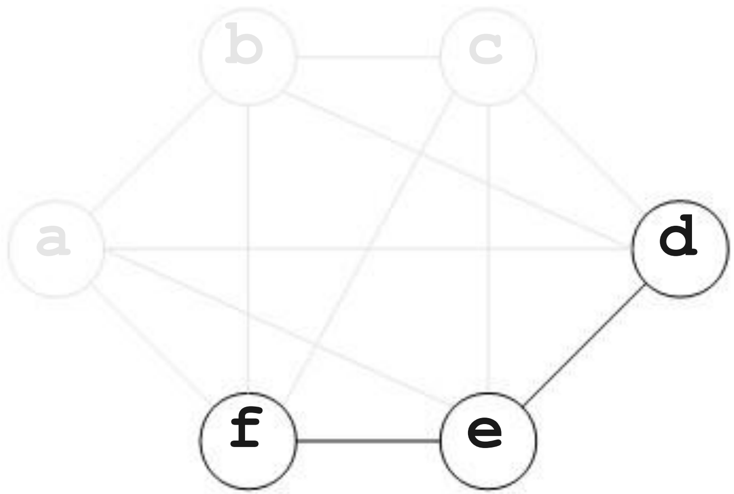
Registers

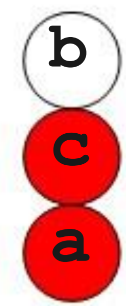
R0	R1	R2

Another Example



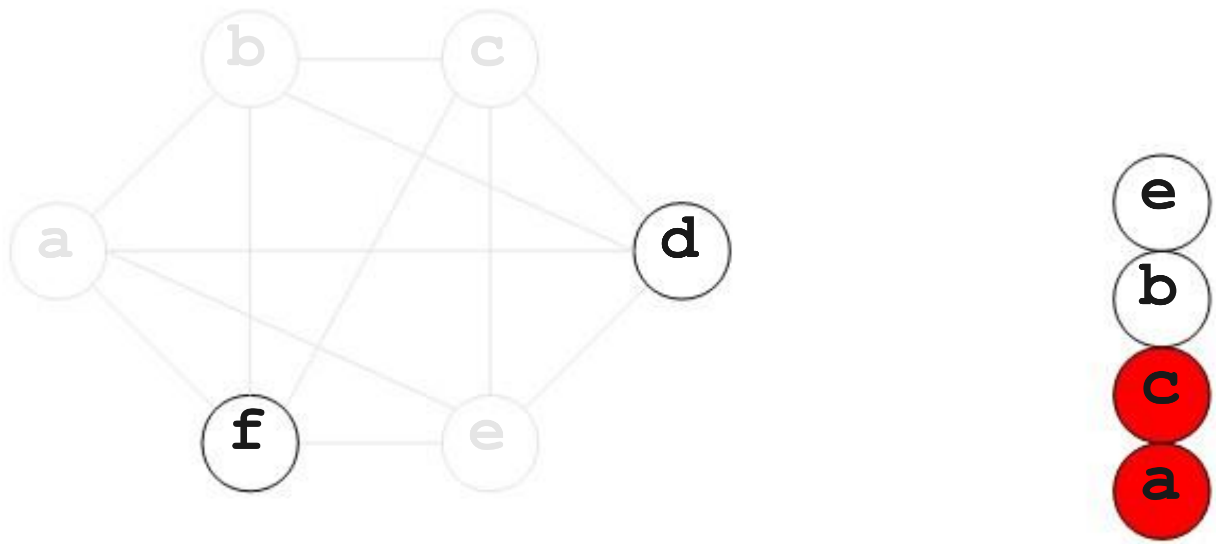
Another Example





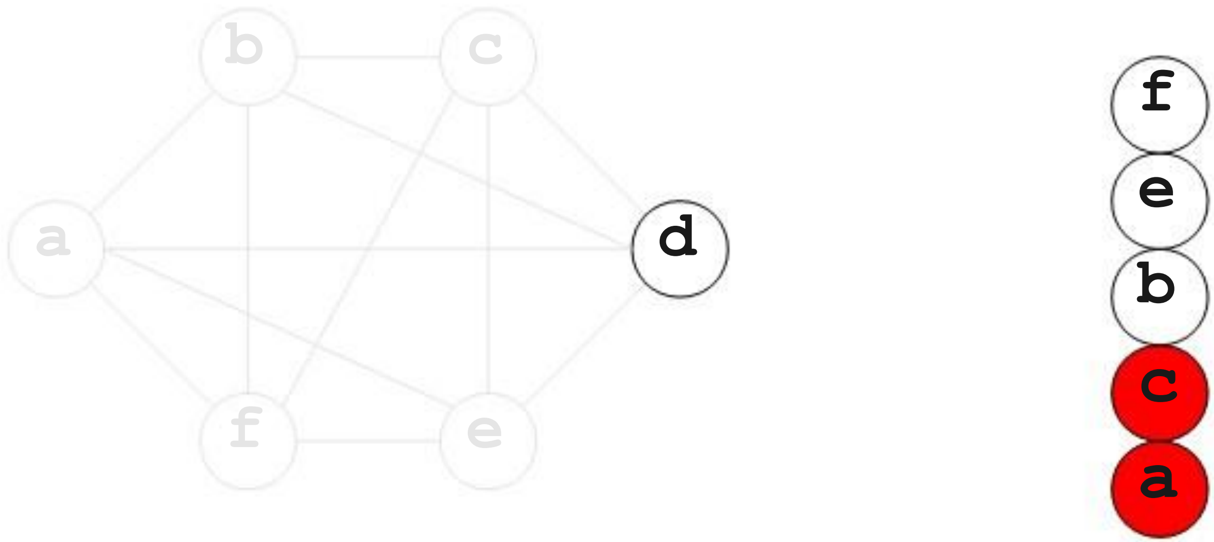
Registers

Another Example



Registers

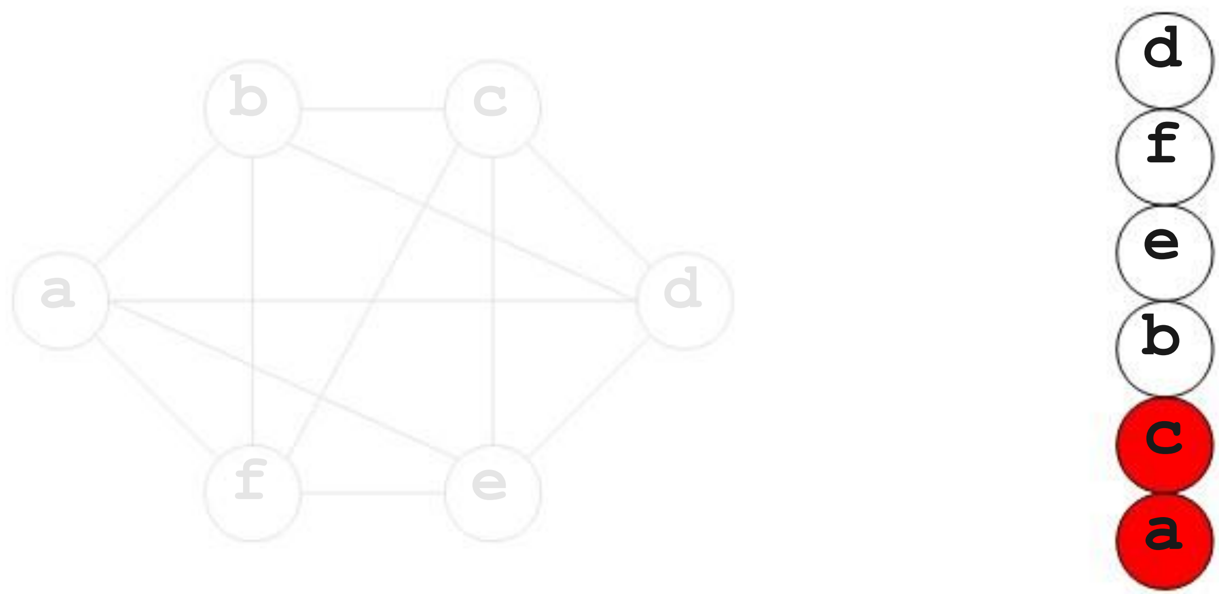
Another Example



Registers

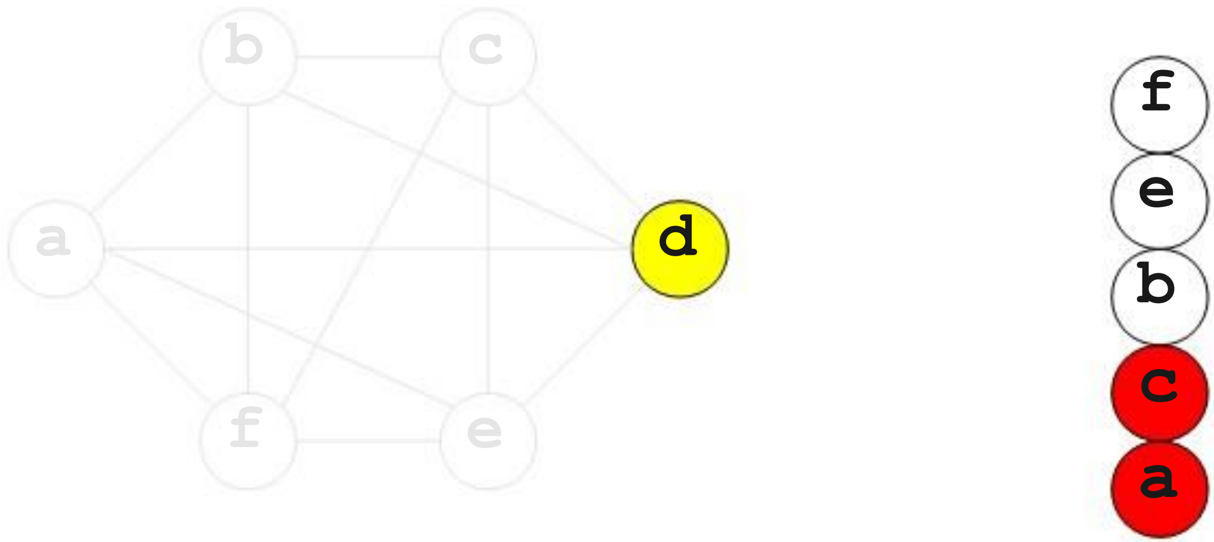
R_0	R_1	R_2

Another Example



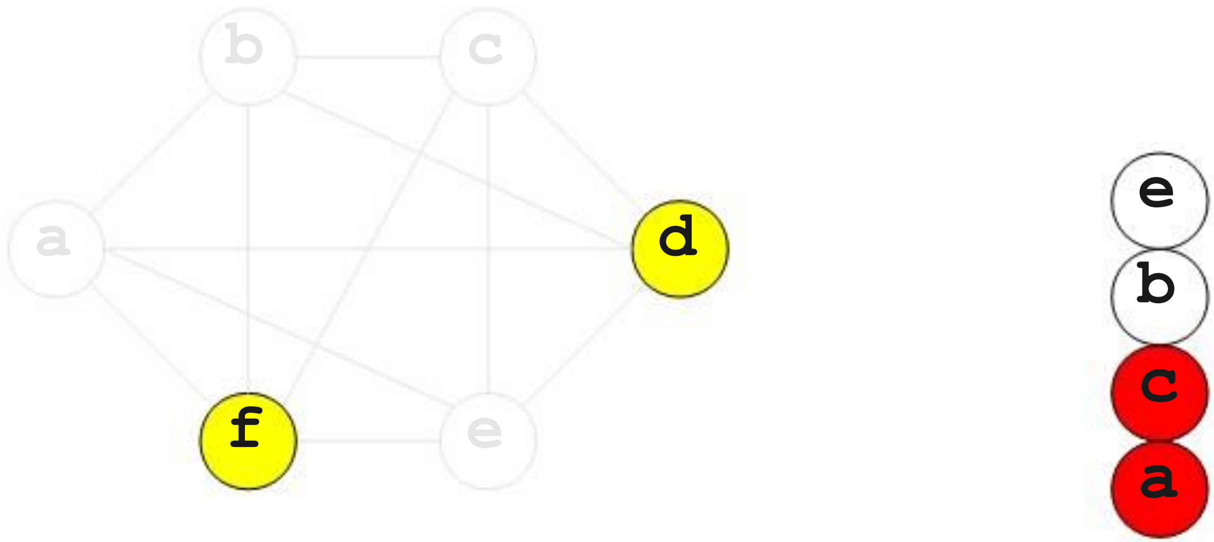
Registers

Another Example



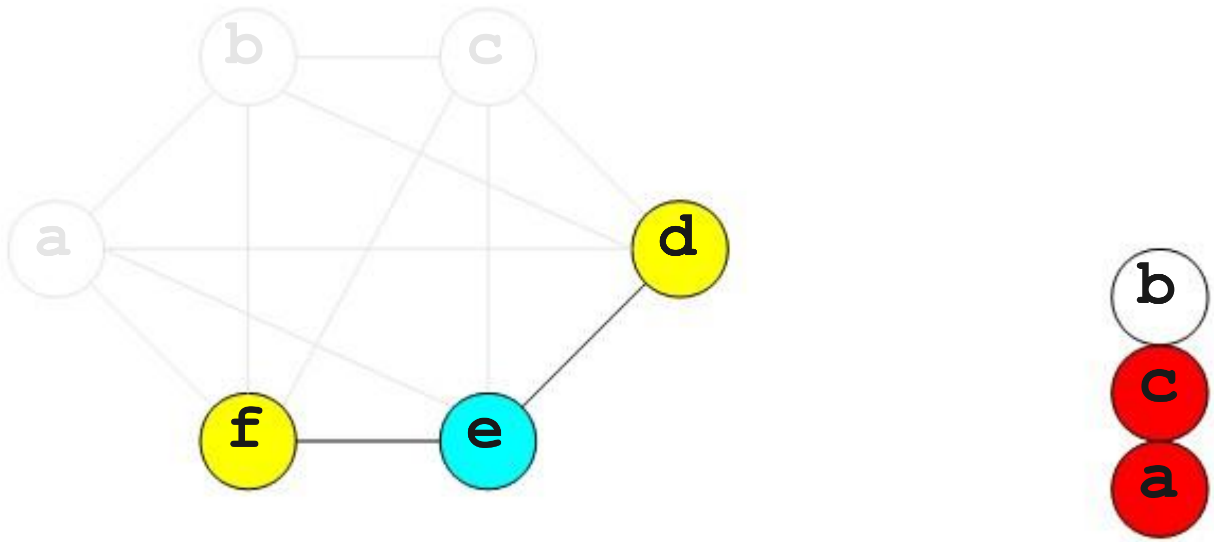
Registers

Another Example



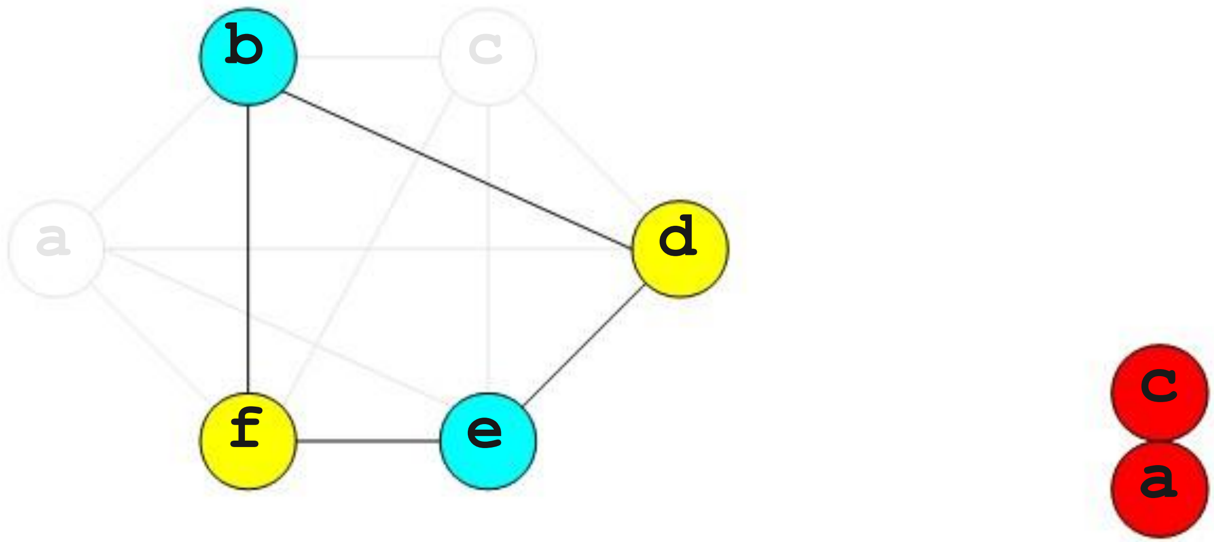
Registers

Another Example



Registers

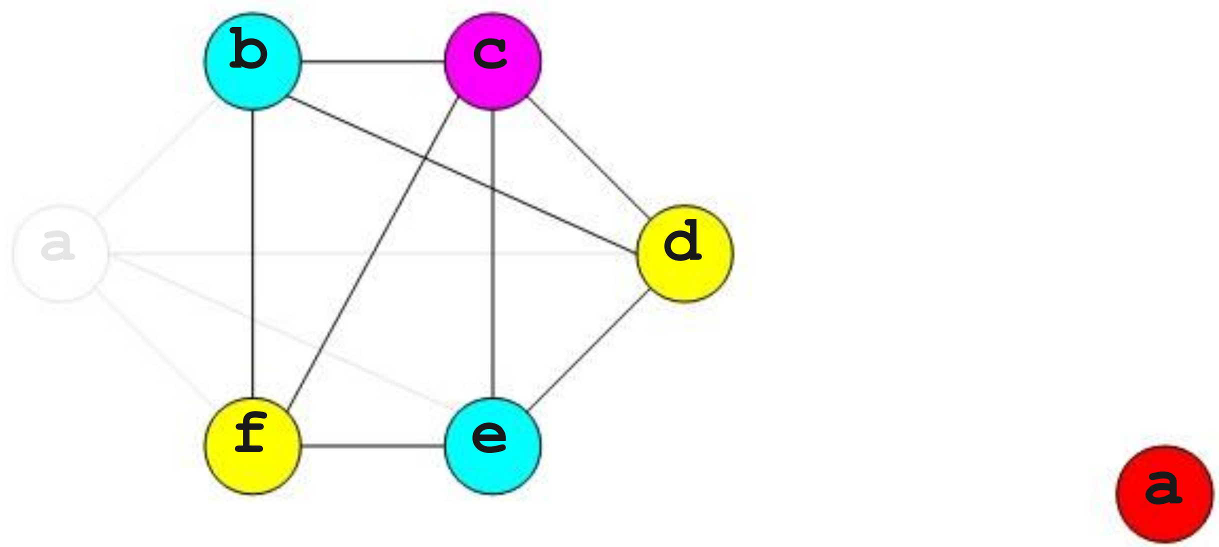
Another Example



Registers

R_0	R_1	R_2

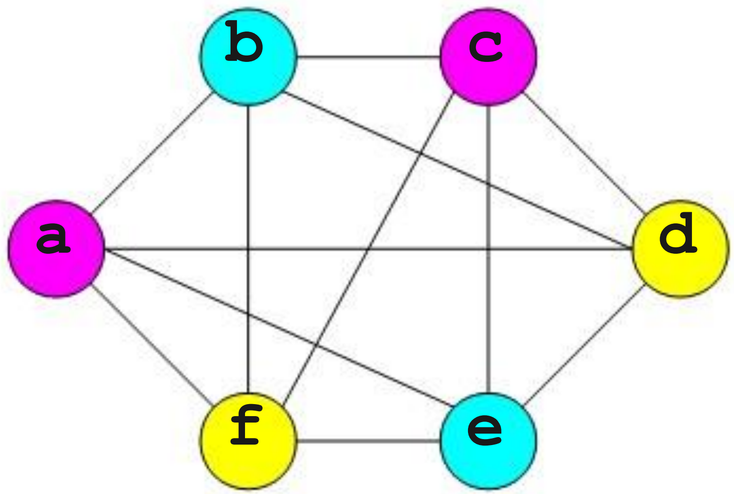
Another Example



Registers

R_0	R_1	R_2

Another Example



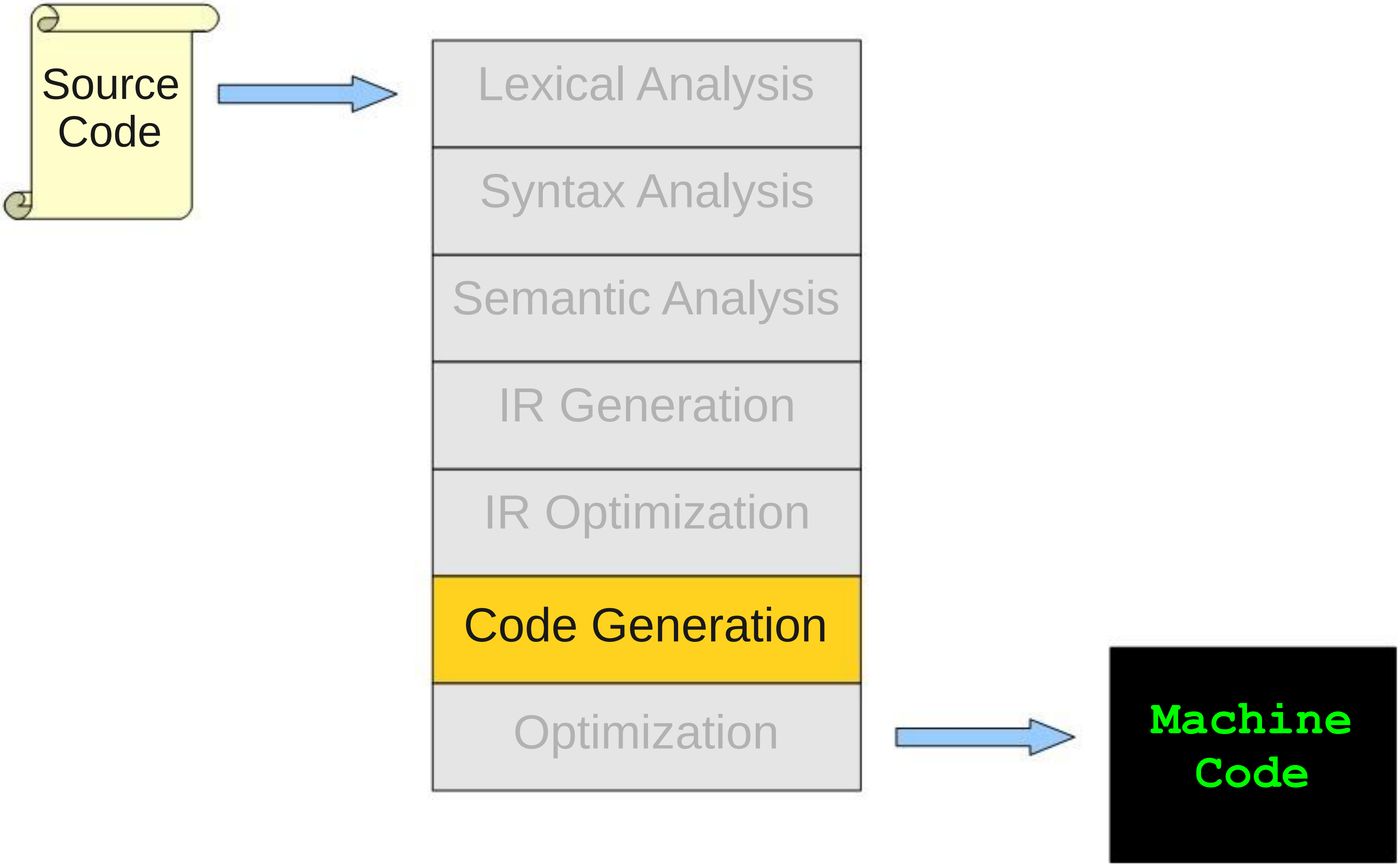
Registers

R_0	R_1	R_2

Note: Chaitin's algorithm often used in production compilers like GCC.

Garbage Collection

Where We Are


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

Memory Management So Far

- Some memory is preallocated and persists throughout the program:
  - Global variables, virtual function tables, executable code, etc.
- Some memory is allocated on the runtime stack:
  - Local variables, parameters, temporaries.
- **Some memory is allocated in the heap**:
  - Variables or Objects whose size don't know at compile time.

- Memory management for the first two is trivial.
- How do we **manage heap-allocated memory**?

Manual Memory Management

Option One: The programmer handles allocation and deallocation of dynamic memory.

- Approach used in C, C++, e.g. malloc(), free()
- Advantages:
  - Programmer can exercise precise control over memory usage.
- Disadvantages: (programmer's mistake)
  - **Memory leaks** where resources are never freed.
  - **Double frees** where a resource is freed twice
  - **Use-after-frees** where a deallocated resource is still used.

Automatic Memory Management

- **Idea:** Have the runtime environment automatically reclaim memory.
- Objects that won't be used again are called **garbage**.
- Reclaiming garbage objects automatically is called **garbage collection**.

Types of Garbage Collectors

- **Incremental** vs **stop-the-world**:
  - An **incremental** collector is one that runs concurrently with the program.
  - A **stop-the-world** collector pauses program execution to look for garbage.
- **Compacting** vs **non-compacting**:
  - A **compacting** collector is one that moves objects around in memory.
  - A **non-compacting** collector is one that leaves all objects where they originated.
- Garbage Collection Techniques
  - Reference Counting, Mark-and-Sweep, Stop-and-Copy,

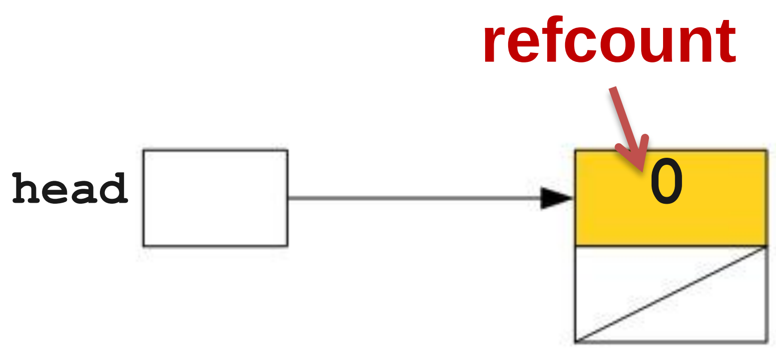
    Generational Garbage Collection

Reference Counting

- A simple framework for garbage collection.
- Idea: Store in each object a **reference count** (**refcount**) tracking how many references exist to the object.
- When an object has zero **refcount**, it is unreachable and can be reclaimed.

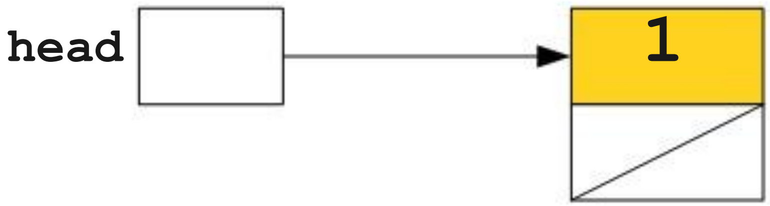
Reference Counting in Action

```
class LinkedList {
    LinkedList next;
}

int main() {
    LinkedList head = new LinkedList;
    LinkedList mid = new LinkedList;
    LinkedList tail = new LinkedList;

    head.next = mid;
    mid.next = tail;

    mid = tail = null;

    head.next.next = null;

    head = null;
}
```



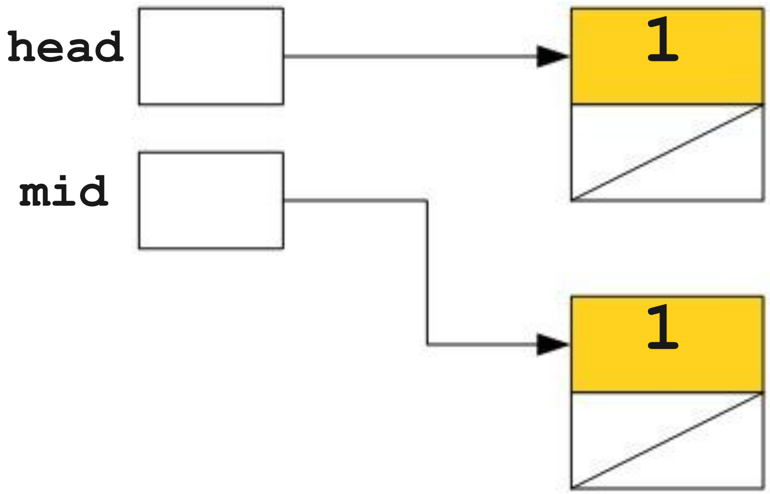
Reference Counting in Action

```
class LinkedList {
    LinkedList next;
}

int main() {
    LinkedList head = new LinkedList;
    LinkedList mid = new LinkedList;
    LinkedList tail = new LinkedList;

    head.next = mid;
    mid.next = tail;

    mid = tail = null;

    head.next.next = null;

    head = null;
}
```



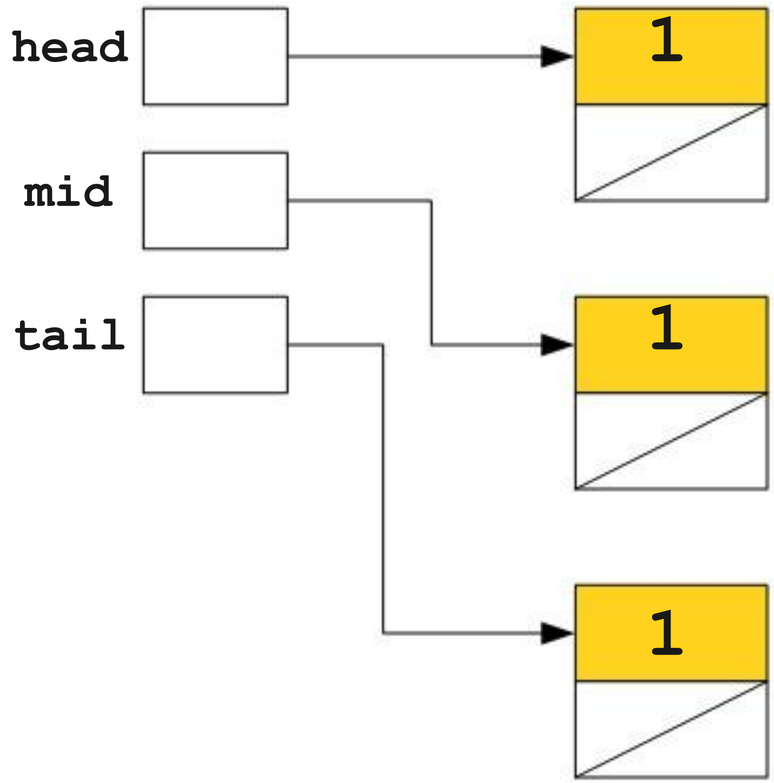
Reference Counting in Action

```
class LinkedList {
    LinkedList next;
}

int main() {
    LinkedList head = new LinkedList;
    LinkedList mid = new LinkedList;
    LinkedList tail = new LinkedList;

    head.next = mid;
    mid.next = tail;

    mid = tail = null;

    head.next.next = null;

    head = null;
}
```



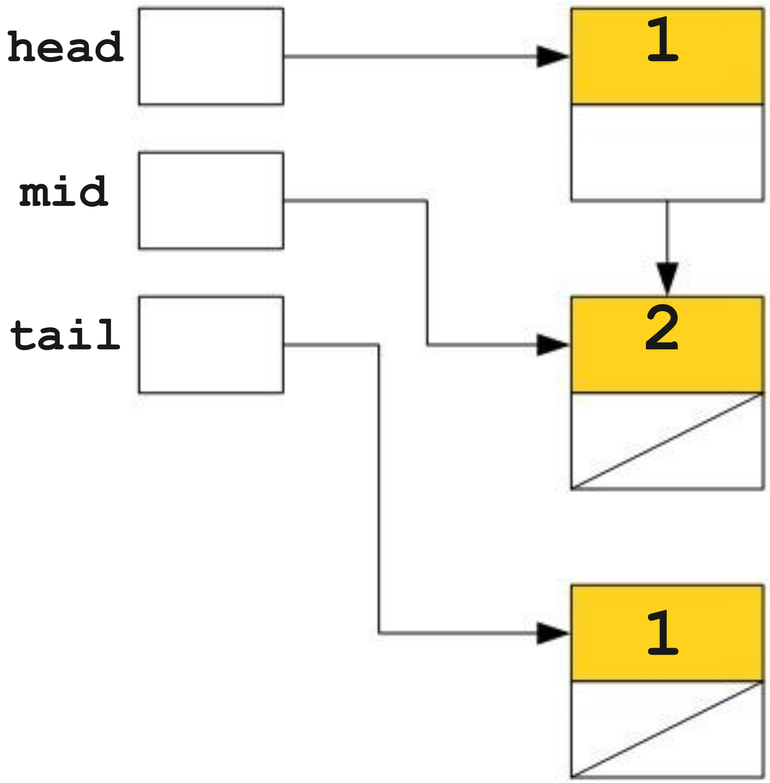
Reference Counting in Action

```
class LinkedList {
    LinkedList next;
}

int main() {
    LinkedList head = new LinkedList;
    LinkedList mid = new LinkedList;
    LinkedList tail = new LinkedList;

    head.next = mid;
    mid.next = tail;

    mid = tail = null;

    head.next.next = null;

    head = null;
}
```



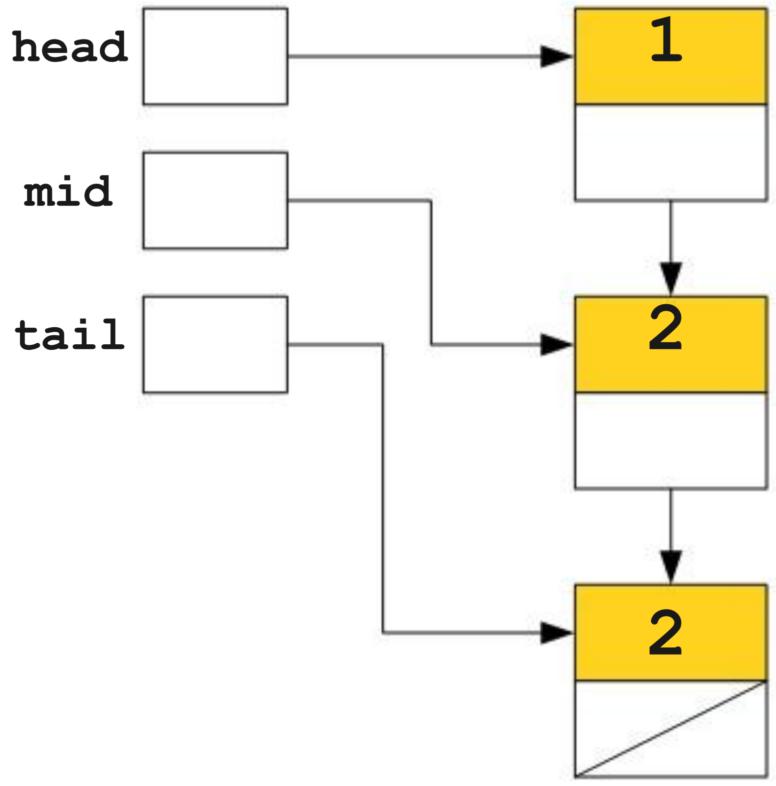
Reference Counting in Action

```
class LinkedList {
    LinkedList next;
}

int main() {
    LinkedList head = new LinkedList;
    LinkedList mid = new LinkedList;
    LinkedList tail = new LinkedList;

    head.next = mid;
    mid.next = tail;

    mid = tail = null;


    head.next.next = null;

    head = null;
}
```



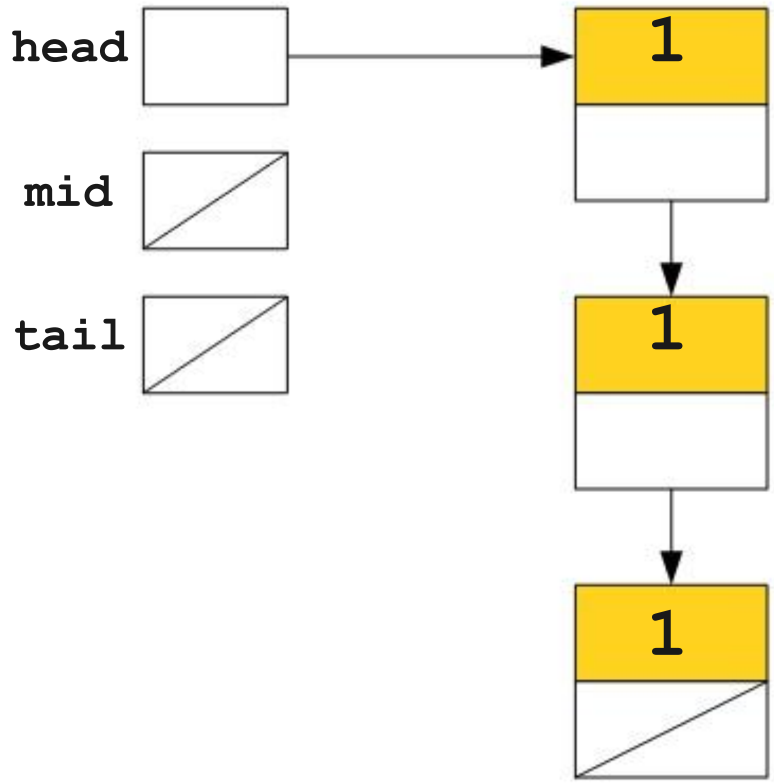
Reference Counting in Action

```
class LinkedList {
    LinkedList next;
}

int main() {
    LinkedList head = new LinkedList;
    LinkedList mid = new LinkedList;
    LinkedList tail = new LinkedList;

    head.next = mid;
    mid.next = tail;

    mid = tail = null;

    head.next.next = null;

    head = null;
}
```



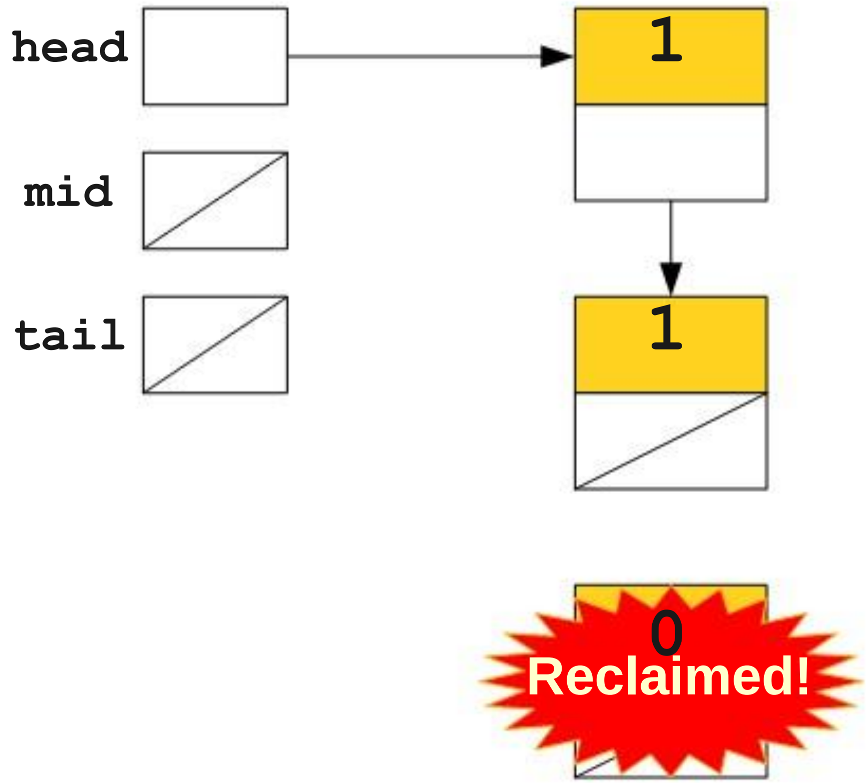
Reference Counting in Action

```
class LinkedList {
    LinkedList next;
}

int main() {
    LinkedList head = new LinkedList;
    LinkedList mid = new LinkedList;
    LinkedList tail = new LinkedList;

    head.next = mid;
    mid.next = tail;

    mid = tail = null;

    head.next.next = null;

    head = null;
}
```



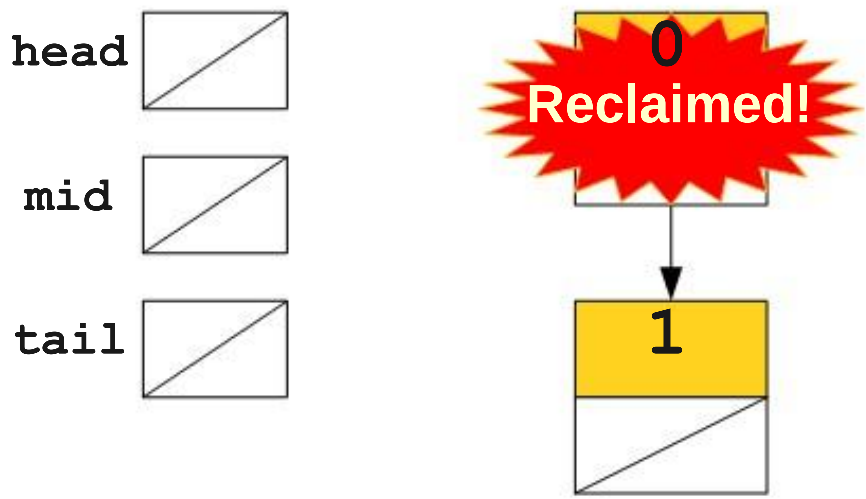
Reference Counting in Action

```
class LinkedList {
    LinkedList next;
}

int main() {
    LinkedList head = new LinkedList;
    LinkedList mid = new LinkedList;
    LinkedList tail = new LinkedList;

    head.next = mid;
    mid.next = tail;

    mid = tail = null;

    head.next.next = null;

    head = null;
}
```



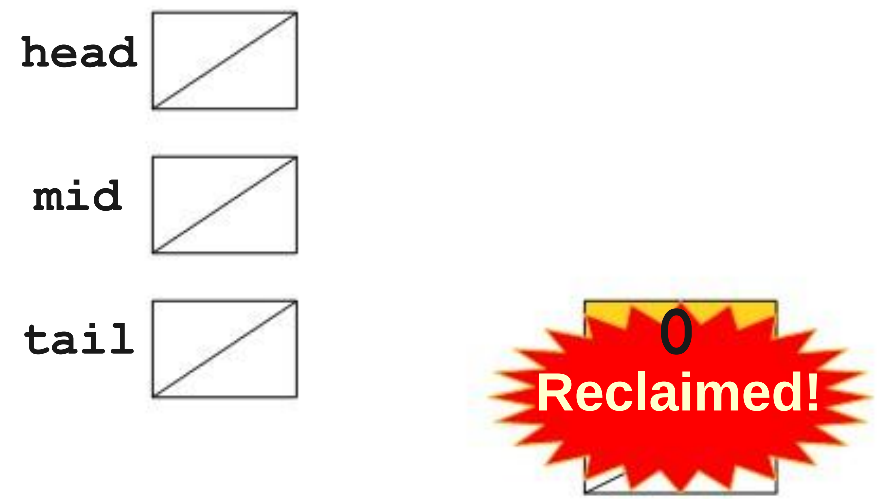
Reference Counting in Action

```
class LinkedList {
    LinkedList next;
}

int main() {
    LinkedList head = new LinkedList;
    LinkedList mid = new LinkedList;
    LinkedList tail = new LinkedList;

    head.next = mid;
    mid.next = tail;

    mid = tail = null;

    head.next.next = null;

    head = null;
}
```



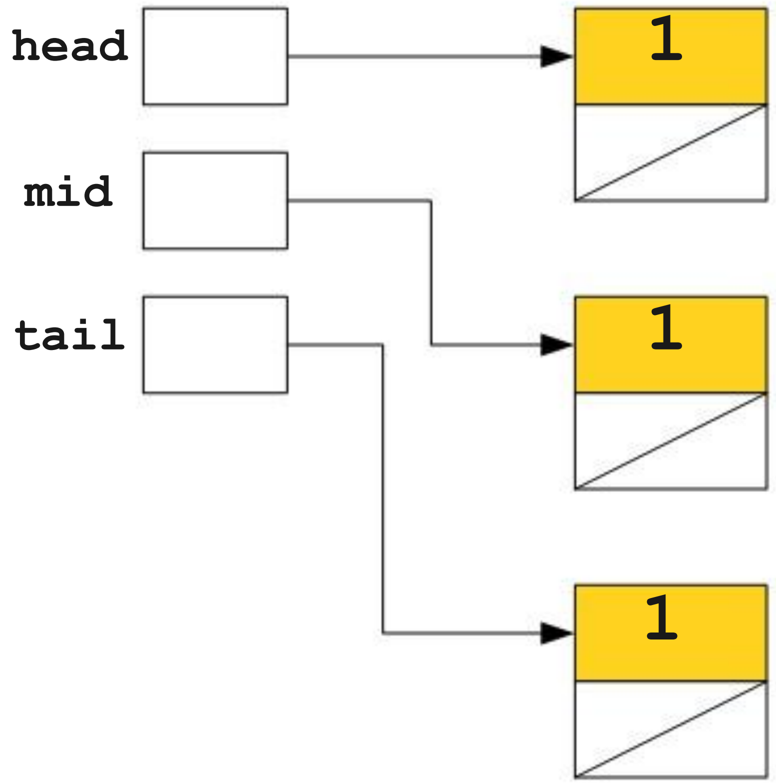
Reference Counting in Action

```
class LinkedList {
    LinkedList next;
}

int main() {
    LinkedList head = new LinkedList;
    LinkedList mid = new LinkedList;
    LinkedList tail = new LinkedList;

    head.next = mid;
    mid.next = tail;

    mid = tail = null;

    head.next.next = null;

    head = null;
}
```



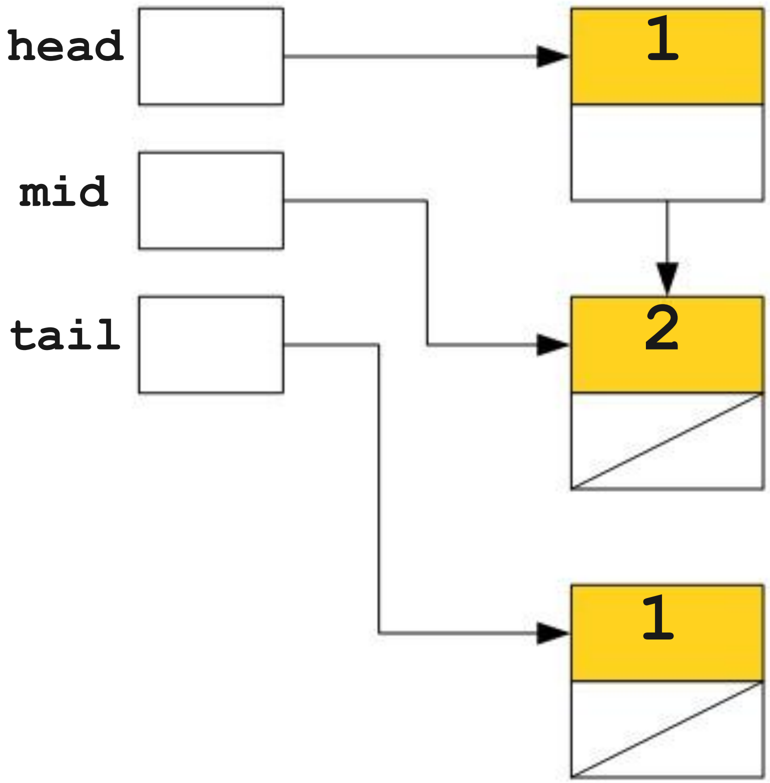
One Major Problem

```
class LinkedList {
    LinkedList next;
}

int main() {
    LinkedList head = new LinkedList;
    LinkedList mid = new LinkedList;
    LinkedList tail = new LinkedList;

    head.next = mid;
    mid.next = tail;
    tail.next = head;

    head = null;
    mid = null;
    tail = null;
}
```



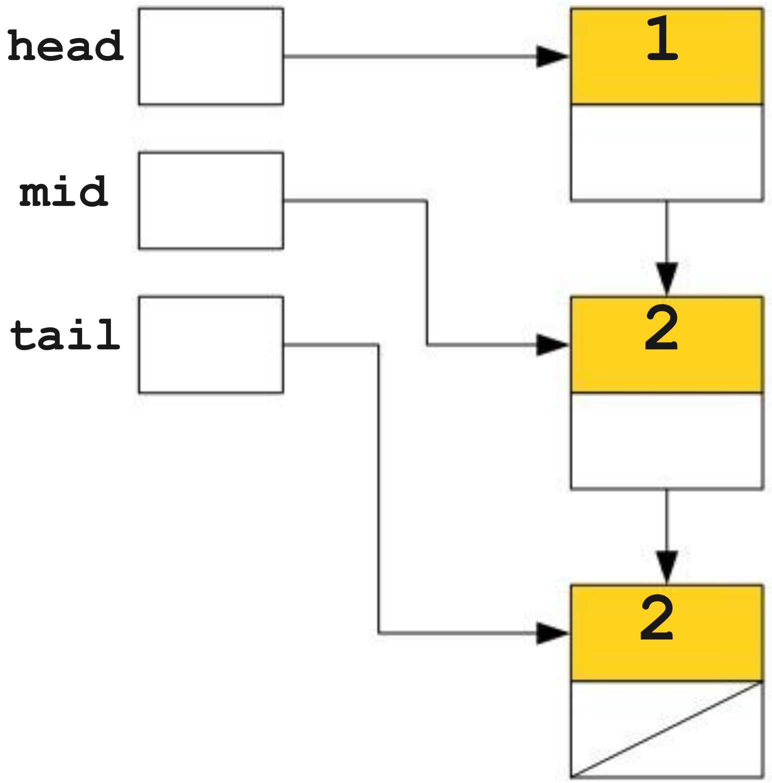
One Major Problem

```
class LinkedList {
    LinkedList next;
}

int main() {
    LinkedList head = new LinkedList;
    LinkedList mid = new LinkedList;
    LinkedList tail = new LinkedList;

    head.next = mid;
    mid.next = tail;
    tail.next = head;

    head = null;
    mid = null;
    tail = null;
}
```



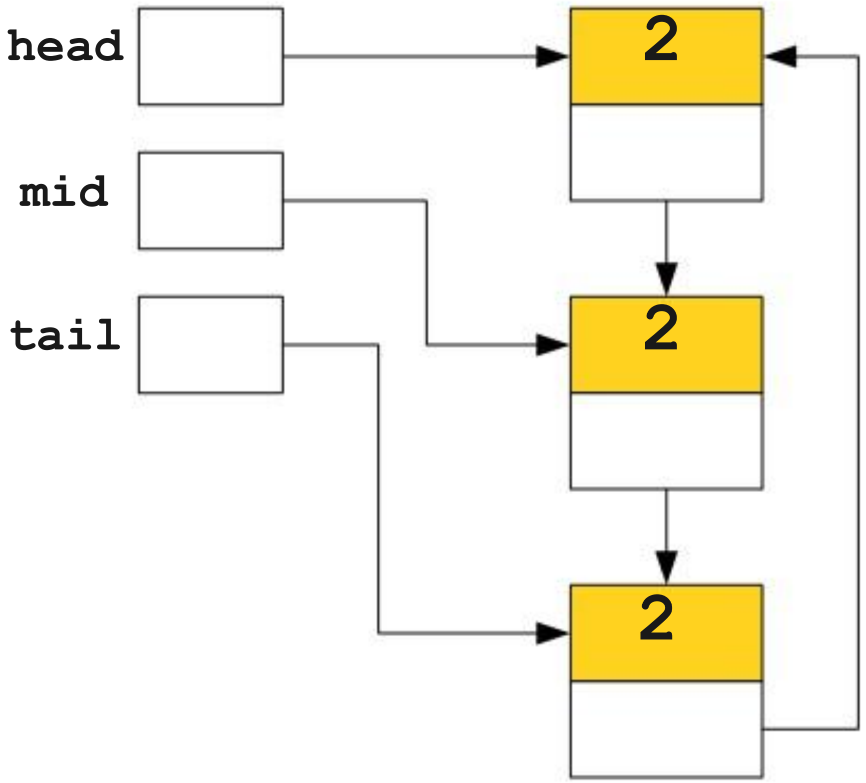
One Major Problem

```
class LinkedList {
    LinkedList next;
}

int main() {
    LinkedList head = new LinkedList;
    LinkedList mid = new LinkedList;
    LinkedList tail = new LinkedList;

    head.next = mid;
    mid.next = tail;
    tail.next = head;

    head = null;
    mid = null;
    tail = null;
}
```



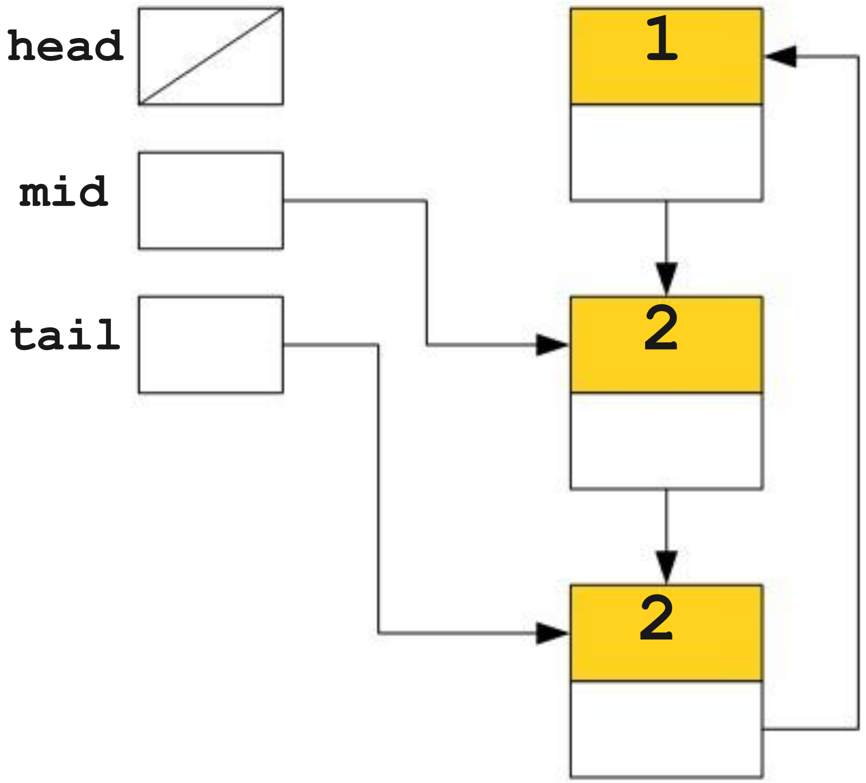
One Major Problem

```
class LinkedList {
    LinkedList next;
}

int main() {
    LinkedList head = new LinkedList;
    LinkedList mid = new LinkedList;
    LinkedList tail = new LinkedList;

    head.next = mid;
    mid.next = tail;
    tail.next = head;

    head = null;
    mid = null;
    tail = null;
}
```



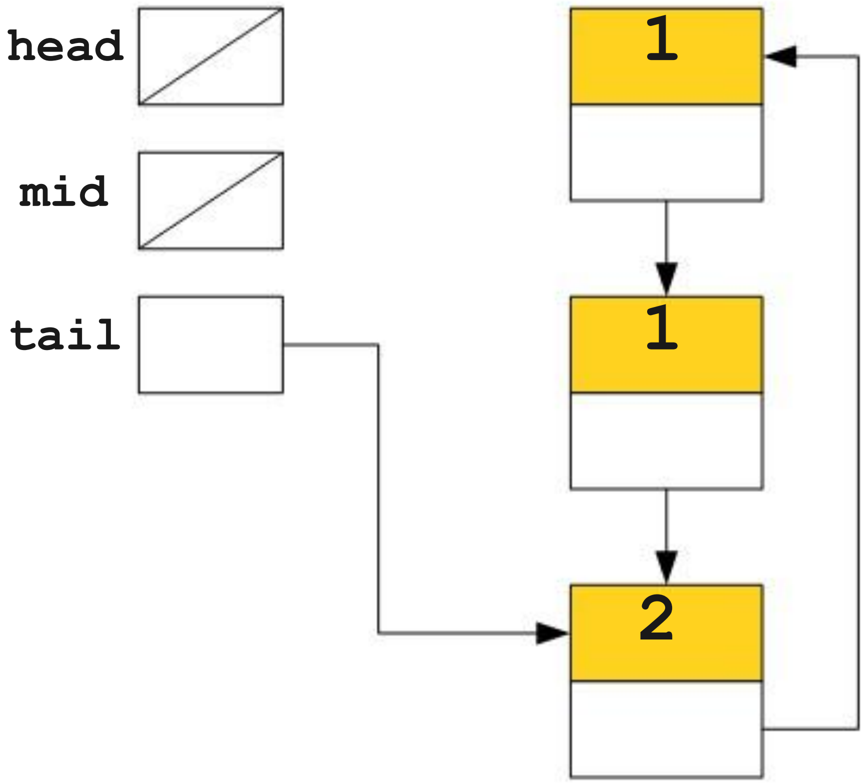
One Major Problem

```
class LinkedList {
    LinkedList next;
}

int main() {
    LinkedList head = new LinkedList;
    LinkedList mid = new LinkedList;
    LinkedList tail = new LinkedList;

    head.next = mid;
    mid.next = tail;
    tail.next = head;

    head = null;
    mid = null;
    tail = null;
}
```



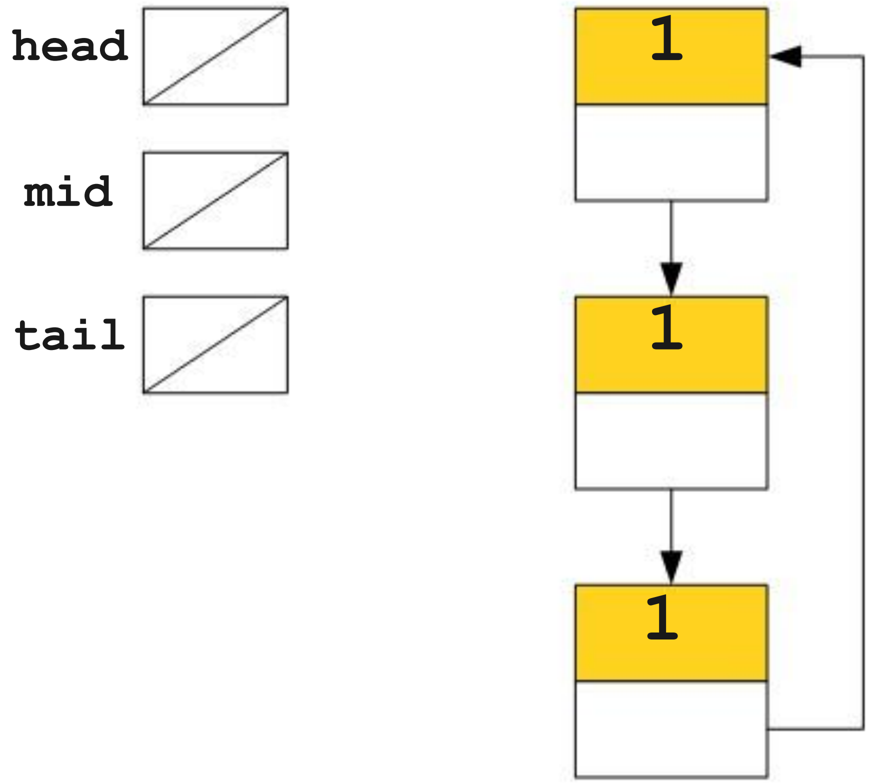
One Major Problem

```
class LinkedList {
    LinkedList next;
}

int main() {
    LinkedList head = new LinkedList;
    LinkedList mid = new LinkedList;
    LinkedList tail = new LinkedList;

    head.next = mid;
    mid.next = tail;
    tail.next = head;

    head = null;
    mid = null;
    tail = null;
}
```



One Major Problem

```
class LinkedList {
    LinkedList next;
}

int main() {
    LinkedList head = new LinkedList;
    LinkedList mid = new LinkedList;
    LinkedList tail = new LinkedList;

    head.next = mid;
    mid.next = tail;
    tail.next = head;

    head = null;
    mid = null;
    tail = null;
}
```



Cannot clear the circular garbage !

- Issue: Refcount tracks number of references,not number of *reachable* references.
- Major problems in languages/systems that use reference counting, e.g. Perl, Firefox 2.