Register Allocation

Xiao Jia

April 25th, 2012

Outline

- Introduction
- Graph coloring
- Linear scan

Introduction

- Want to replace temporary variables with some fixed set of registers

Introduction

- Want to replace temporary variables with some fixed set of registers

- We will judge this phase by …
  - Code review
  - Limiting # of executed instructions, e.g. no greater than 1 million
    - *Compare: only 160,000 for my optimized 8-queen*

Graph coloring

- First: need to know which variables are live after each instruction
  – Two simultaneously live variables cannot be allocated to the same register

Graph coloring

Control Flow Graph

- For every node n in CFG, we have out[n]
  - Set of temporaries live out of n
- Two variables *interfere* if
  - both initially live (i.e. function arguments), or
  - both appear in out[n] for any n, or
  - one is defined and the other is in out[n]
    - x = b - c   where x is dead & b is live interfere
- How to assign registers to variables?

Interference graph

- Nodes of the graph = variables
- Edges connect variables that interfere with one another
- Nodes will be assigned a color corresponding to the register assigned to the variable
- Two colors can’t be next to one another in the graph

Interference graph

Instructions	Live vars
b = a + 2	
c = b * b	
b = c + 1	
return b * a	

Interference graph

Instructions	Live vars
b = a + 2	
c = b * b	
b = c + 1	
	b,a
return b * a	

Interference graph

Instructions	Live vars
b = a + 2	
c = b * b	
	a,c
b = c + 1	
	b,a
return b * a	

Interference graph

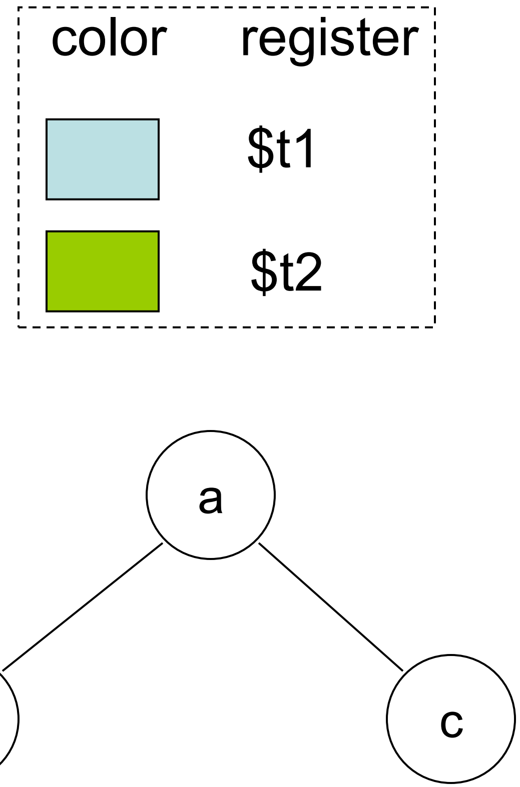
Instructions	Live vars
b = a + 2	
	b,a
c = b * b	
	a,c
b = c + 1	
	b,a
return b * a	

Interference graph

Instructions	Live vars
	a
b = a + 2	
	b,a
c = b * b	
	a,c
b = c + 1	
	b,a
return b * a	

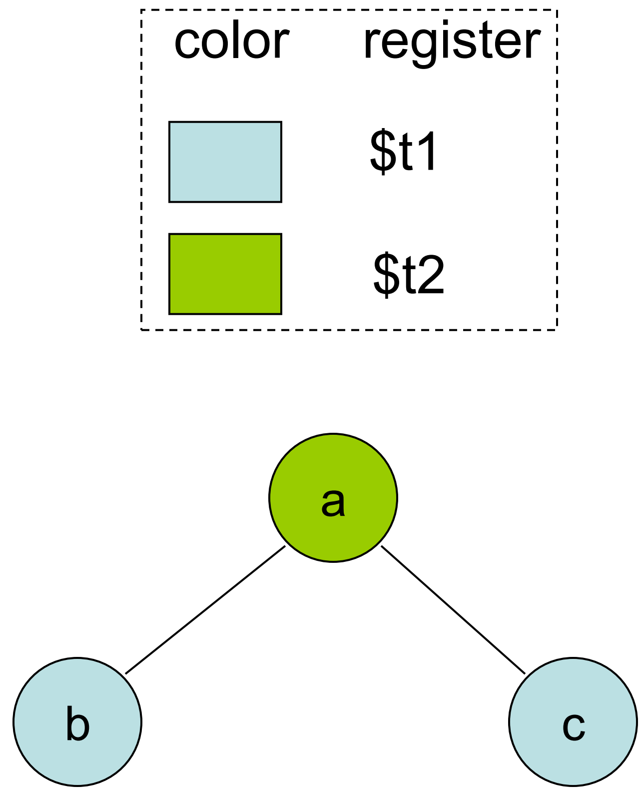
Interference graph

Instructions	Live vars
	a
b = a + 2	
	a,b
c = b * b	
	a,c
b = c + 1	
	a,b
return b * a	



Interference graph

Instructions	Live vars
	a
b = a + 2	
	a,b
c = b * b	
	a,c
b = c + 1	
	a,b
return b * a	



Graph coloring

- Questions:
  - Can we efficiently find a coloring of the graph whenever possible?
  - Can we efficiently find the optimum coloring of the graph?
  - What do we do when there aren’t enough colors (registers) to color the graph?

Coloring a graph

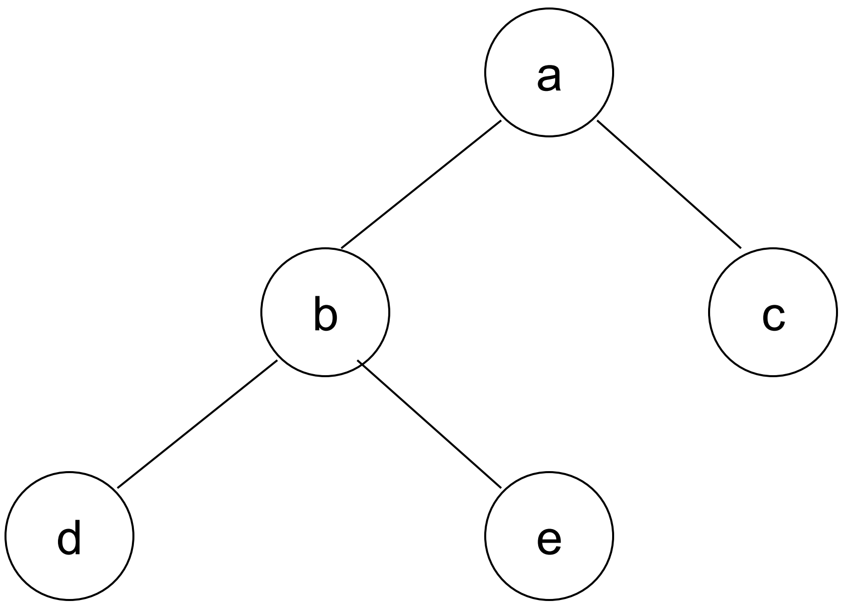
- Kempe's algorithm [1879] for finding a K-coloring of a graph
- Step 1 (simplify): find a node with at most K-1 edges and cut it out of the graph. (Remember this node on a stack for later stages.)

Coloring a graph

- Once a coloring is found for the simpler graph, we can always color the node we saved on the stack
- Step 2 (color): when the simplified subgraph has been colored, add back the node on the top of the stack and assign it a color not taken by one of the adjacent nodes

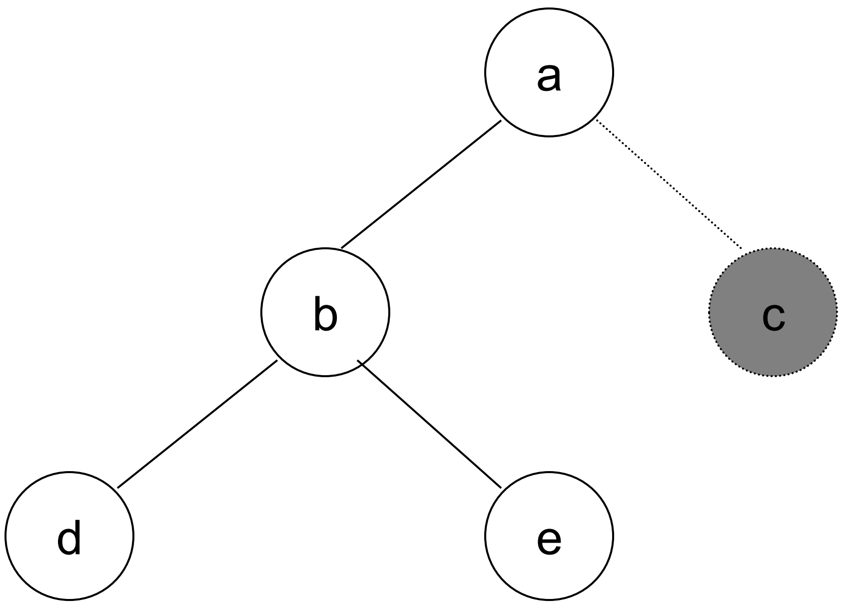
Coloring

color	register
	$t1
	$t2



stack:

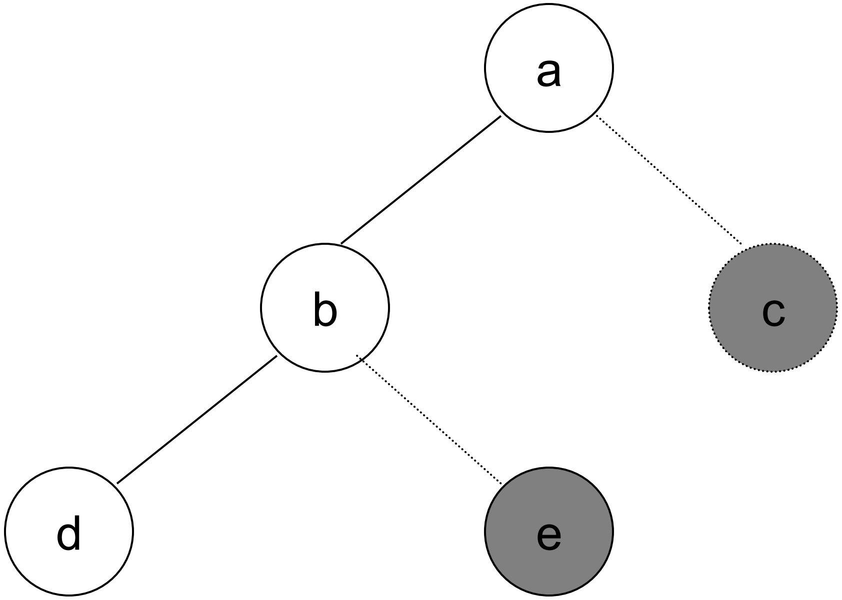
Coloring



stack:

c

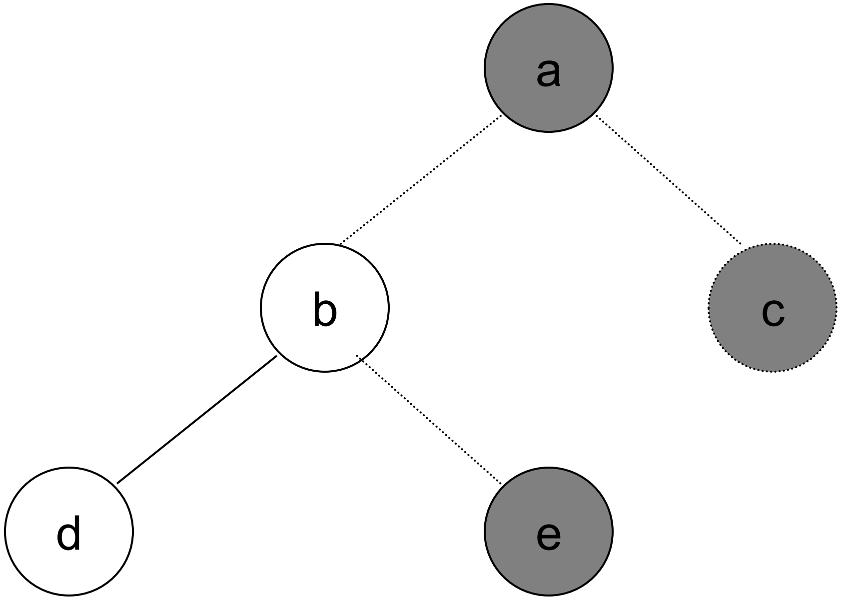
Coloring



stack:

e
c

Coloring

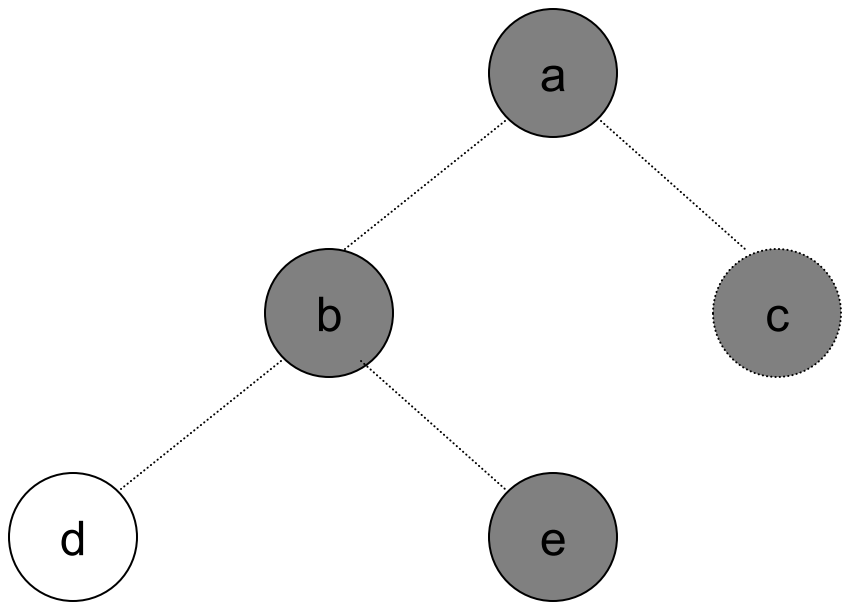


stack:

a
e
c

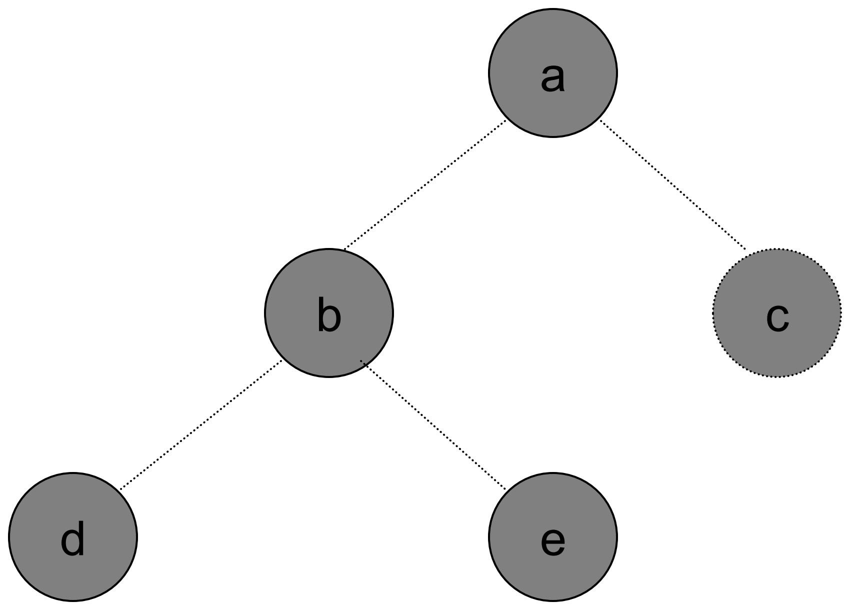
Coloring

color	register
	$t1
	$t2



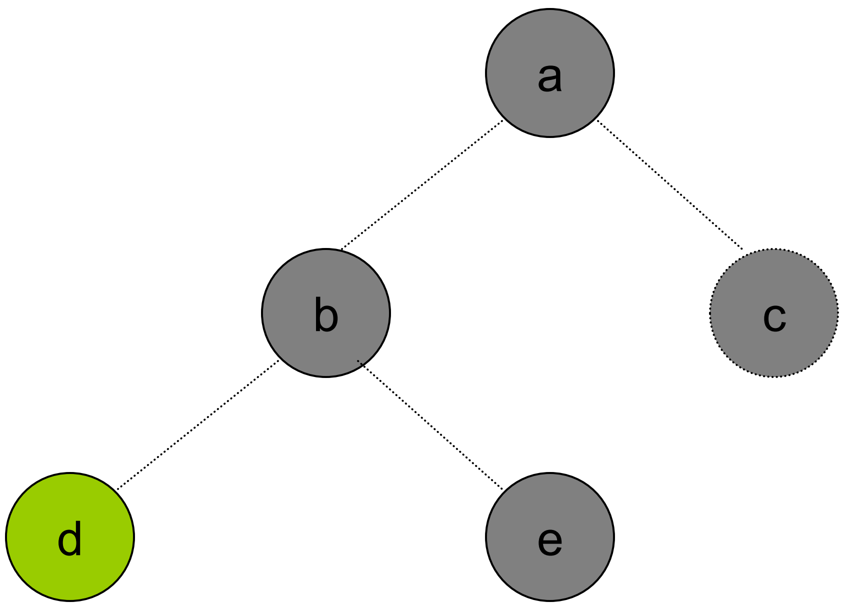
stack:
b
a
e
c

Coloring



stack:
d
b
a
e
c

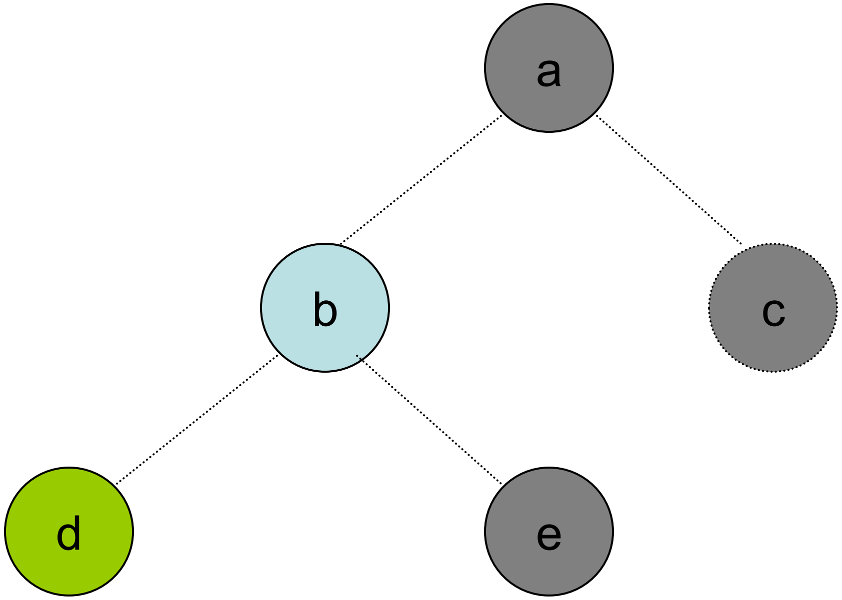
Coloring



stack:

b
a
e
c

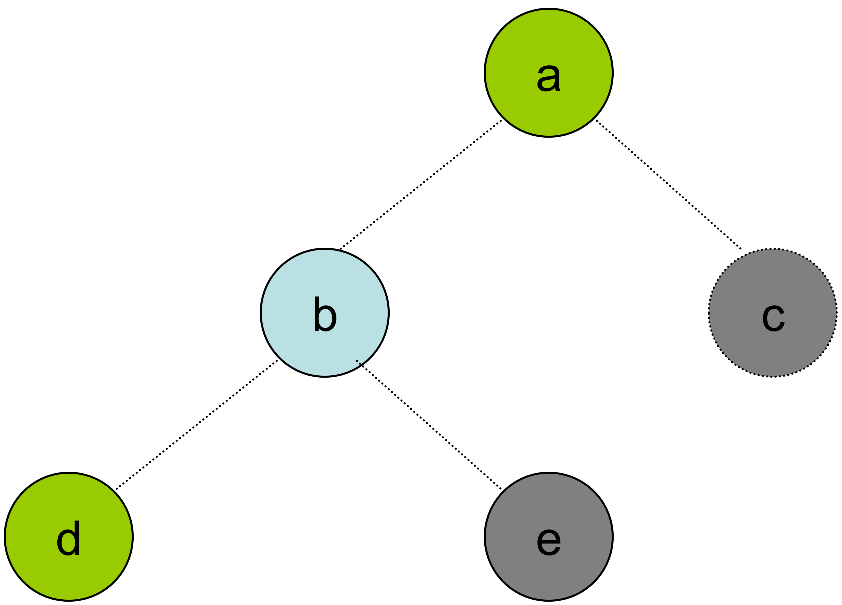
Coloring



stack:

a
e
c

Coloring

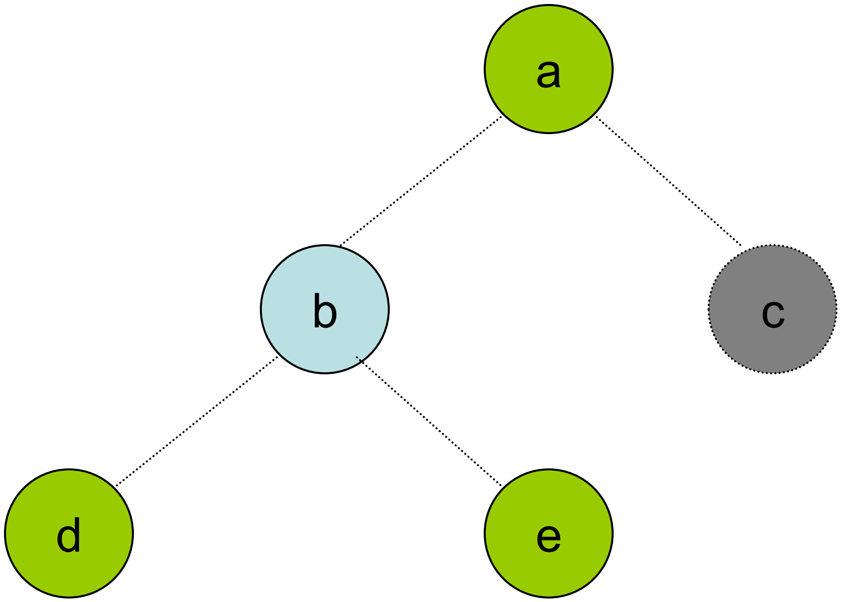


stack:

e
c

Coloring

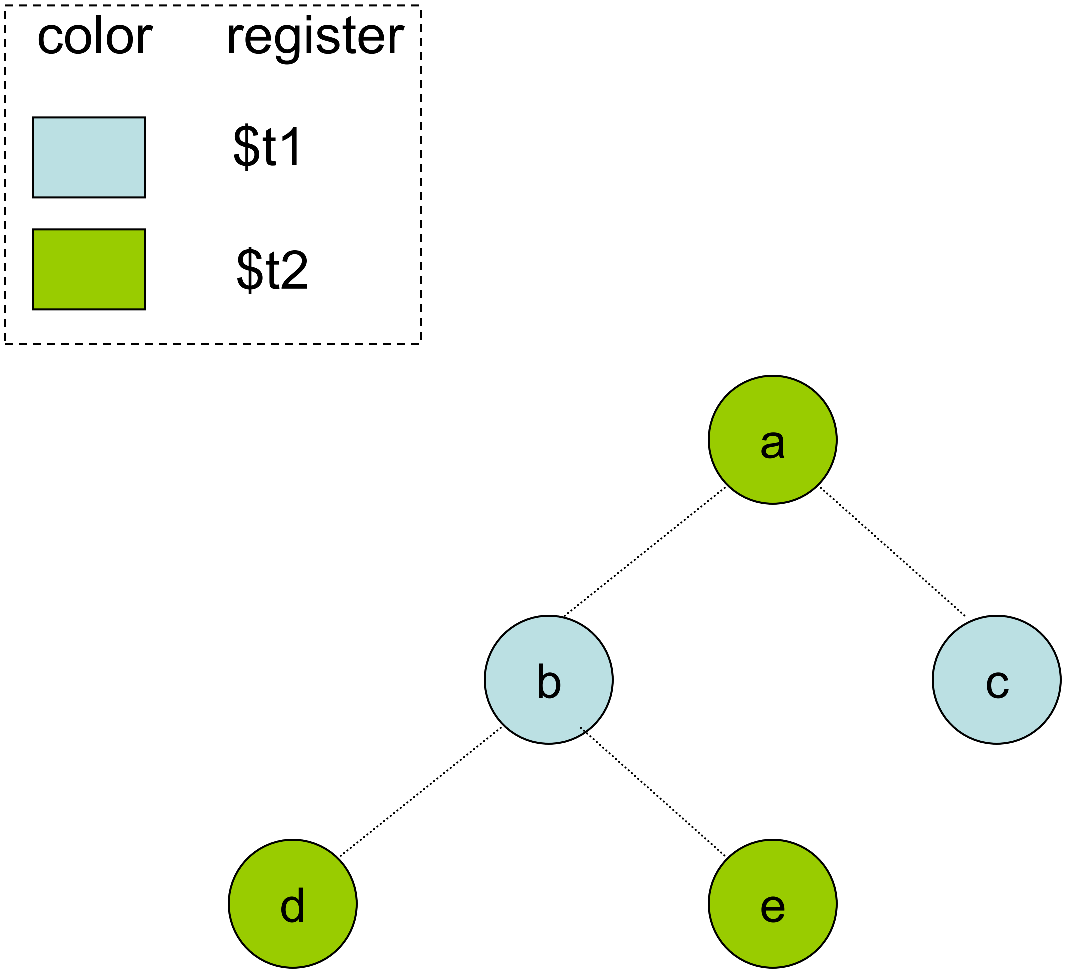
color	register
	$t1
	$t2



stack:

c

Coloring



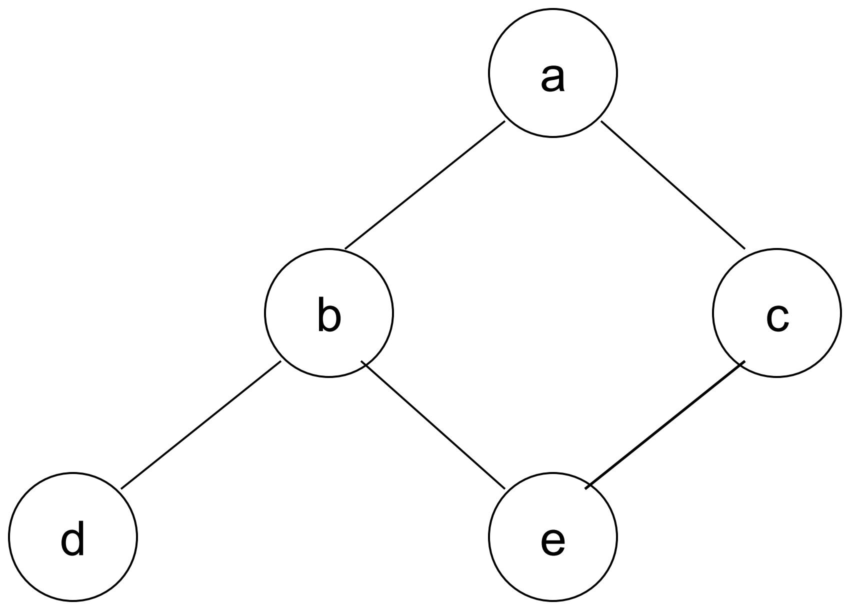
stack:

Failure

- If the graph cannot be colored, it will eventually be simplified to graph in which every node has at least K neighbors
- Sometimes, the graph is still K-colorable!
- Finding a K-coloring in all situations is an NP-complete problem
  - We will have to approximate to make register allocators fast enough

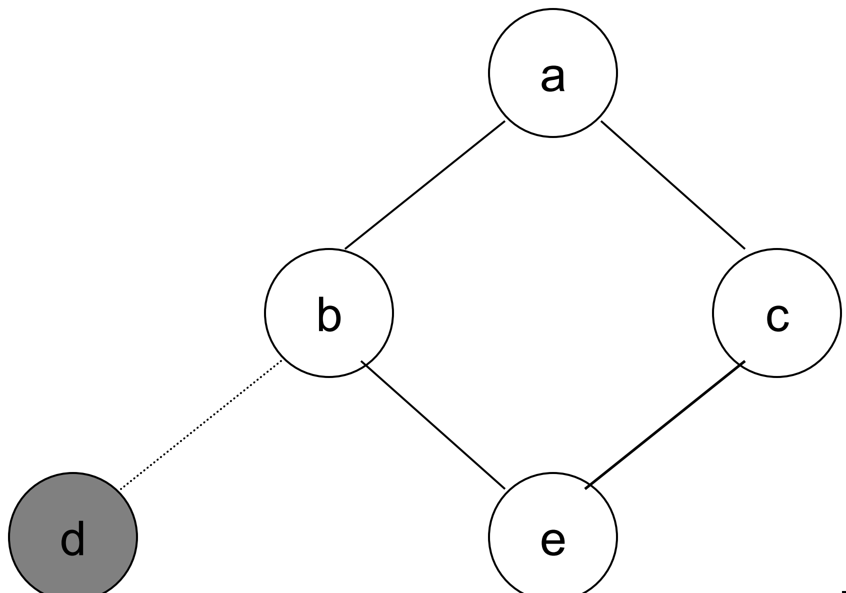
Coloring

color	register
	\$t1
	\$t2



stack:

Coloring

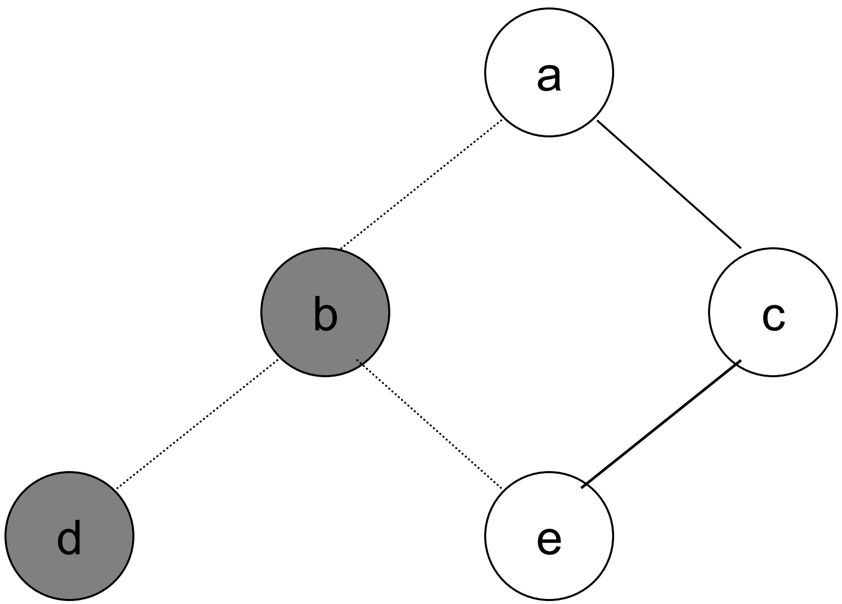


stack:
d

all nodes have
2 neighbours!

Coloring

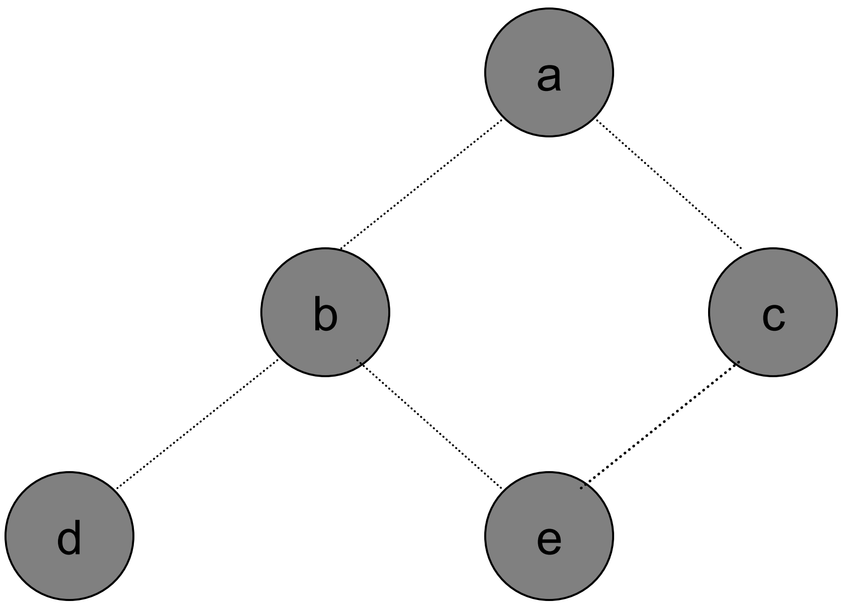
color	register
	$t1
	$t2



stack:

b
d

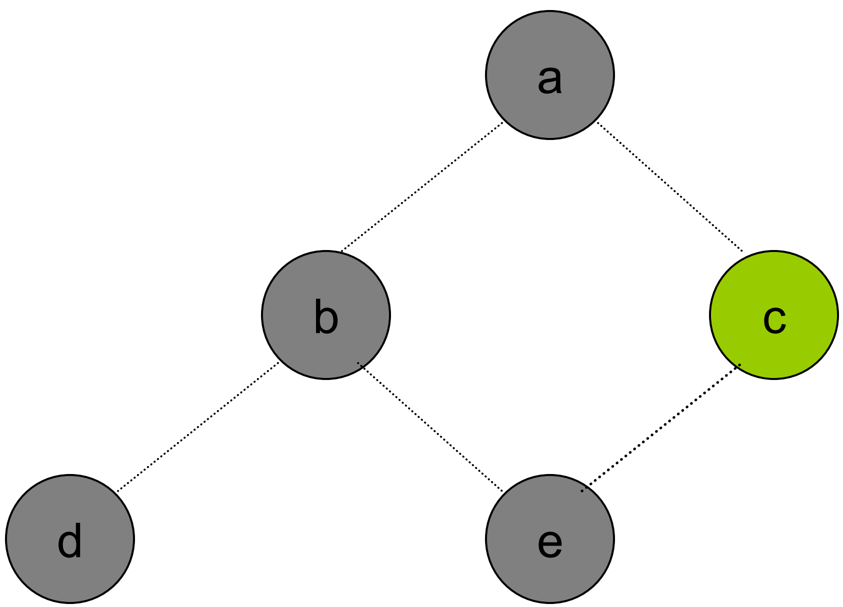
Coloring



stack:
c
e
a
b
d

Coloring

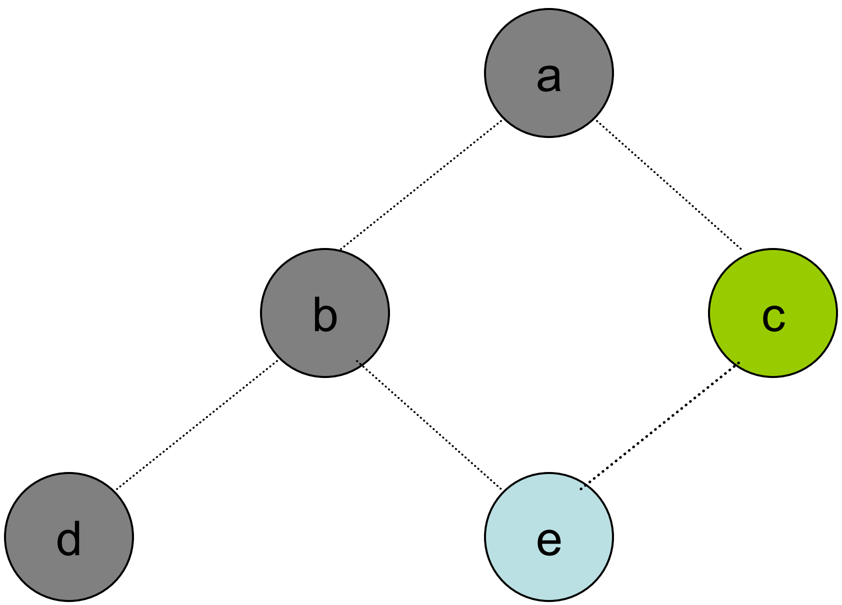
color	register
	$t1
	$t2



stack:

e
a
b
d

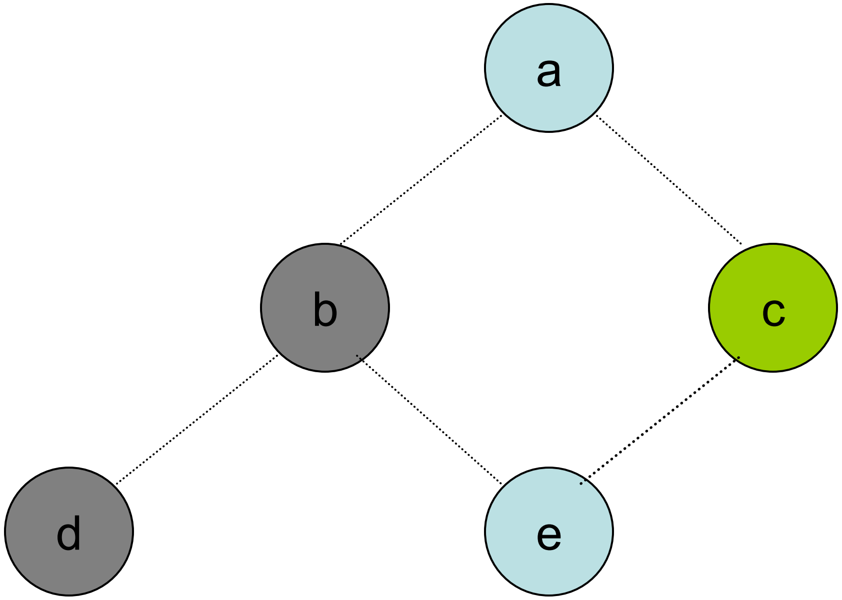
Coloring



stack:

a
b
d

Coloring

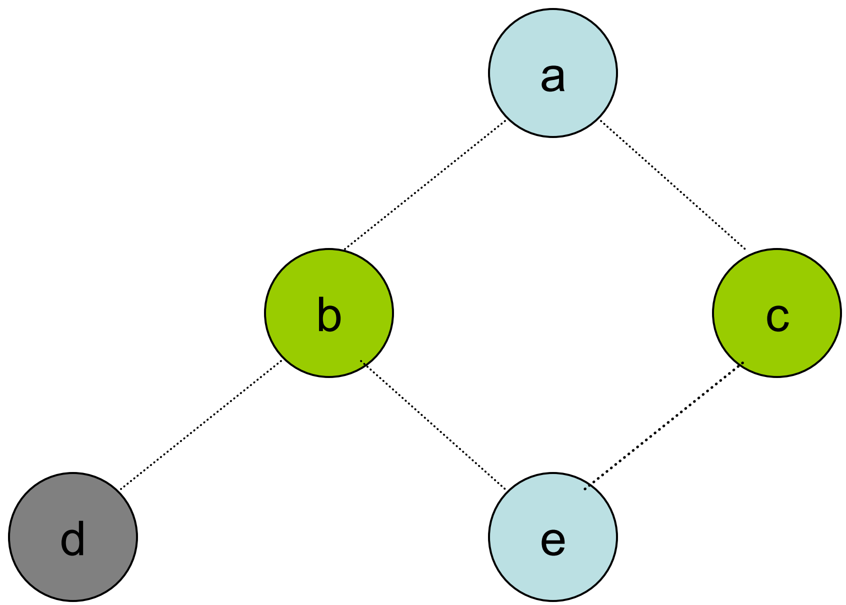


stack:

b
d

Coloring

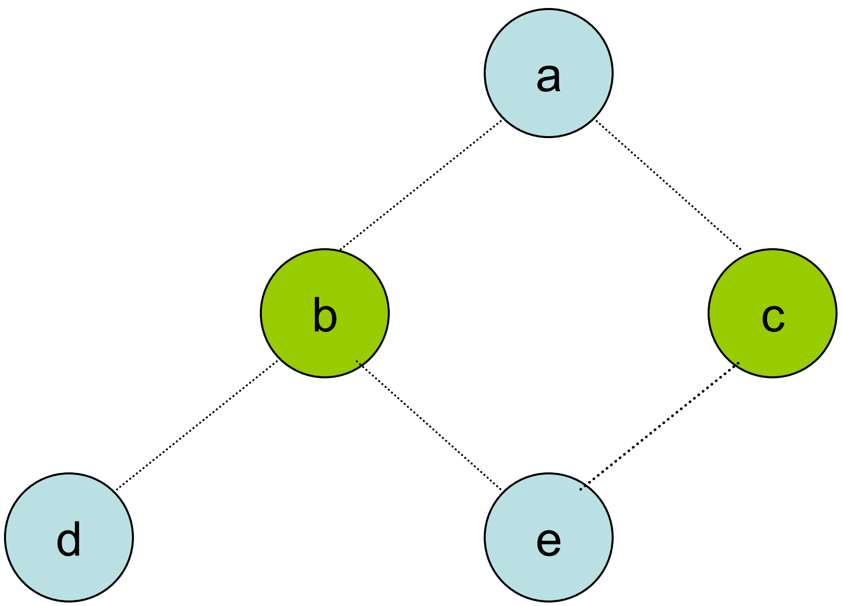
color	register
	$t1
	$t2



stack:

d

Coloring



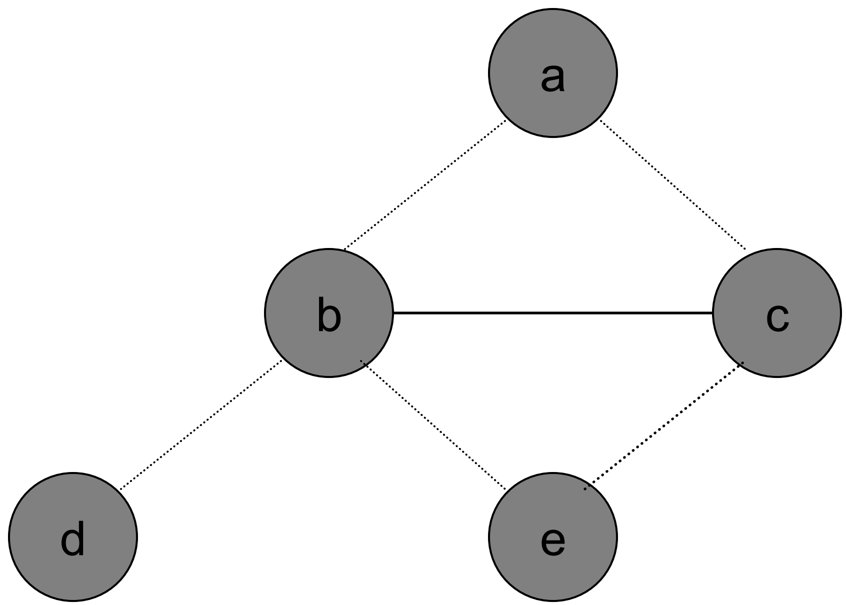
stack:

We got lucky!

Coloring

color	register
	$t1
	$t2

Some graphs can't be colored
in K colors:

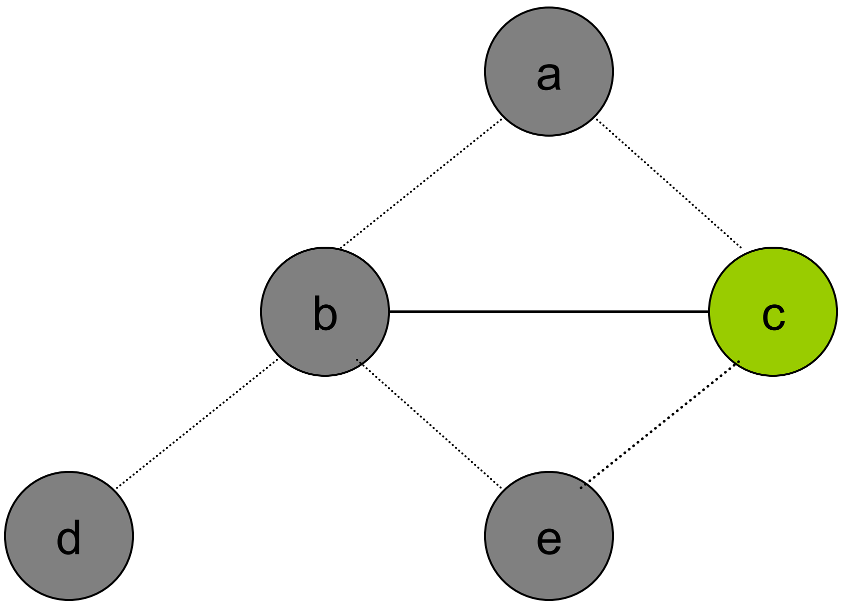


stack:

c
b
e
a
d

Coloring

Some graphs can't be colored in K colors:



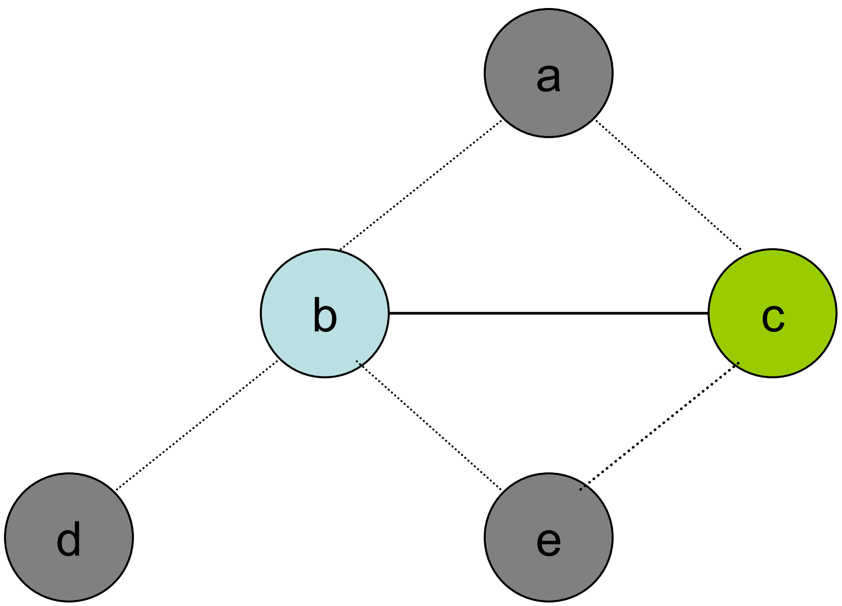
stack:

b
e
a
d

Coloring

color	register
(light blue)	$t1
(green)	$t2

Some graphs can't be colored in K colors:



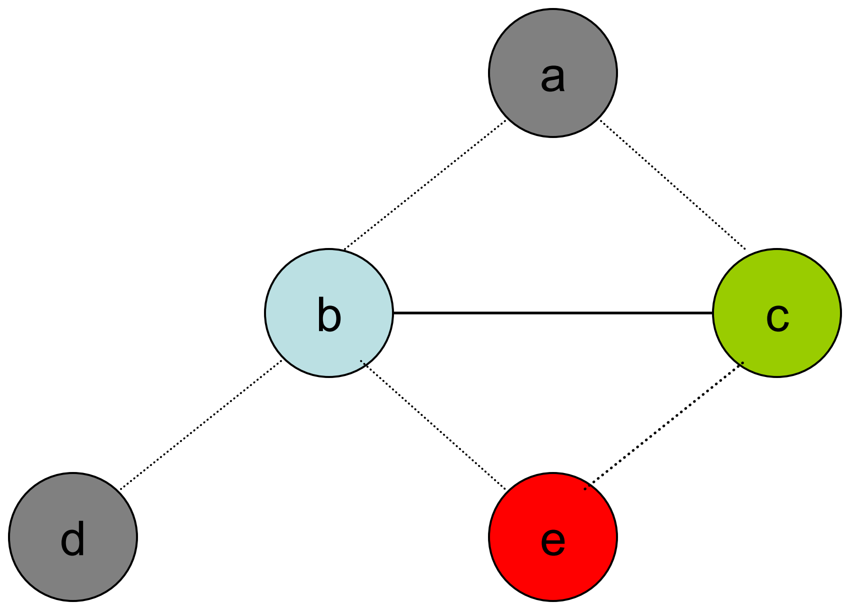
stack:

e
a
d

Coloring

color	register
(light blue)	$t1
(green)	$t2

Some graphs can't be colored
in K colors:



stack:

e
a
d

no colors left for e!

Spilling

- Step 3 (spilling): once all nodes have K or more neighbors, pick a node for spilling
  - Storage on the stack
- There are many heuristics that can be used to pick a node
  - not in an inner loop

Spilling code

- We need to generate extra instructions to load variables from stack and store them
- These instructions use registers themselves. What to do?

  Dedicated registers

  - Stupid approach: always keep extra registers handy for shuffling data in and out: what a waste!
  - Better approach: ?

Spilling code

- We need to generate extra instructions to load variables from stack and store them
- These instructions use registers themselves. What to do?
  - Stupid approach: always keep extra registers handy for shuffling data in and out: what a waste!
  - Better approach: rewrite code introducing a new temporary; rerun liveness analysis and register allocation

Rewriting code

- Consider: add t1, t1, t2
  - Suppose t2 is a selected for spilling and assigned to stack location 24($fp)
  - Introduce new temporary t3 for just this instruction and rewrite:
    - ld t3, 24($fp)
    - add t1, t1, t3
  - Advantage: t3 has a very short live range and is much less likely to interfere.
  - Rerun the algorithm; fewer variables will spill

Precolored Nodes

See Tiger book for more details

- Some variables are pre-assigned to registers
  - Frame pointer
  - Arguments ($a0, $a1, $a2, $a3)
  - Function call defines (trashes) caller-save registers
- Treat these registers as special temporaries; before beginning, add them to the graph with their colors

Precolored Nodes

See Tiger book for more details

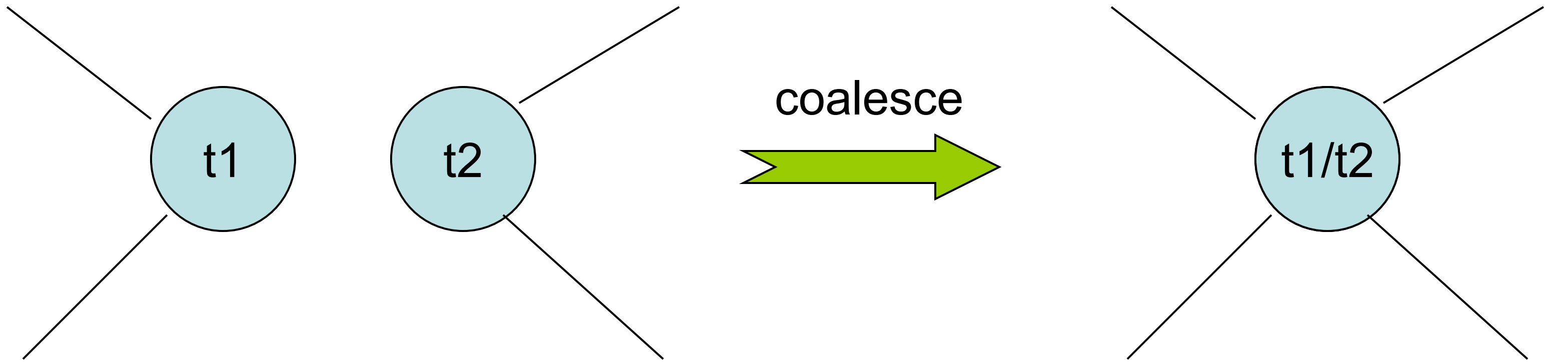
- Can’t simplify a graph by removing a precolored node
- Precolored nodes are the starting point of the coloring process
- Once simplified down to colored nodes start adding back the other nodes as before

Optimizing Moves

- Code generation produces a lot of extra move instructions
  - mov t1, t2
  - If we can assign t1 and t2 to the same register, we do not have to execute the mov
  - Idea: if t1 and t2 are not connected in the interference graph, we coalesce into a single variable

Coalescing

- Problem: coalescing can increase the number of interference edges and make a graph uncolorable

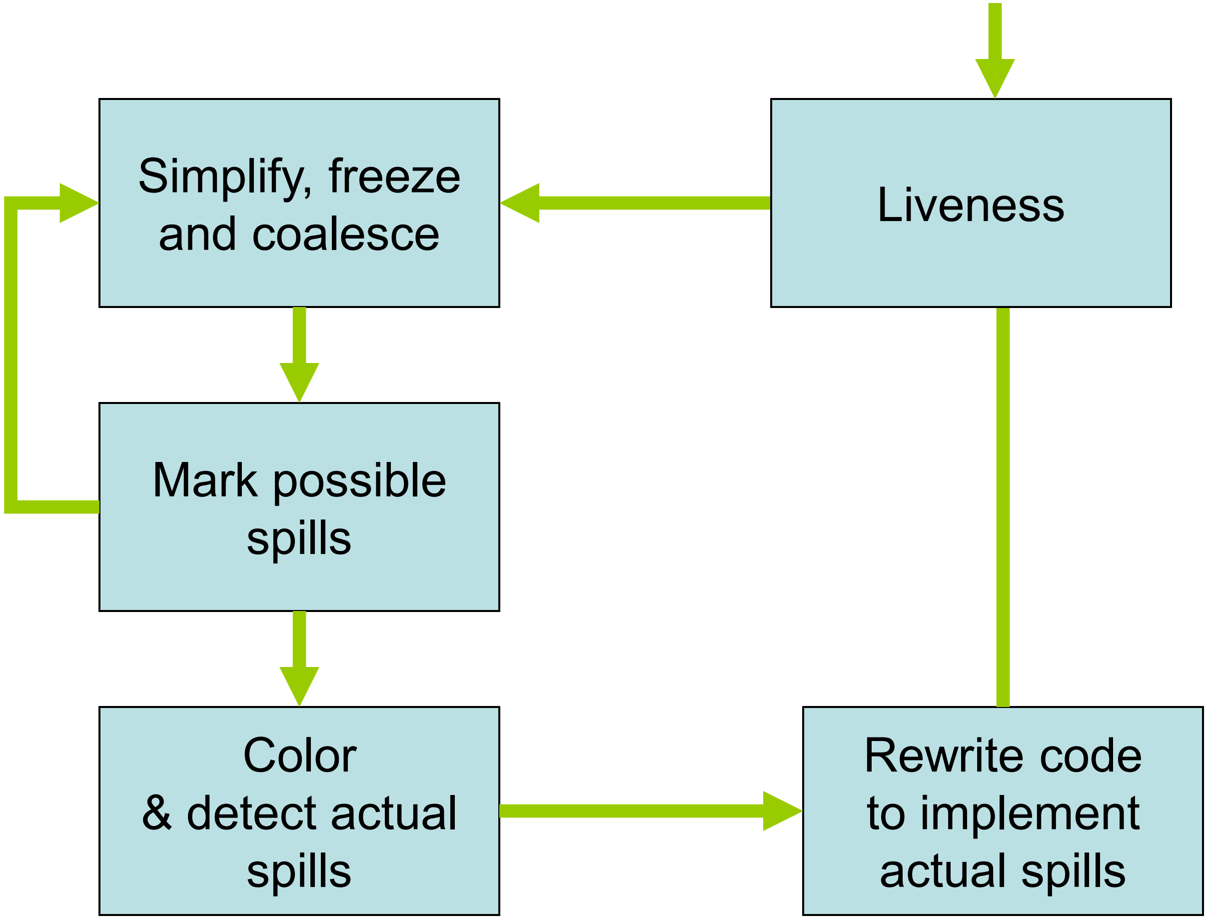


- Solution 1 (Briggs): avoid creation of high-degree (>= K) nodes
- Solution 2 (George): a can be coalesced with b if every neighbour t of a:
  - already interferes with b, or
  - has low-degree (< K)

Simplify & Coalesce

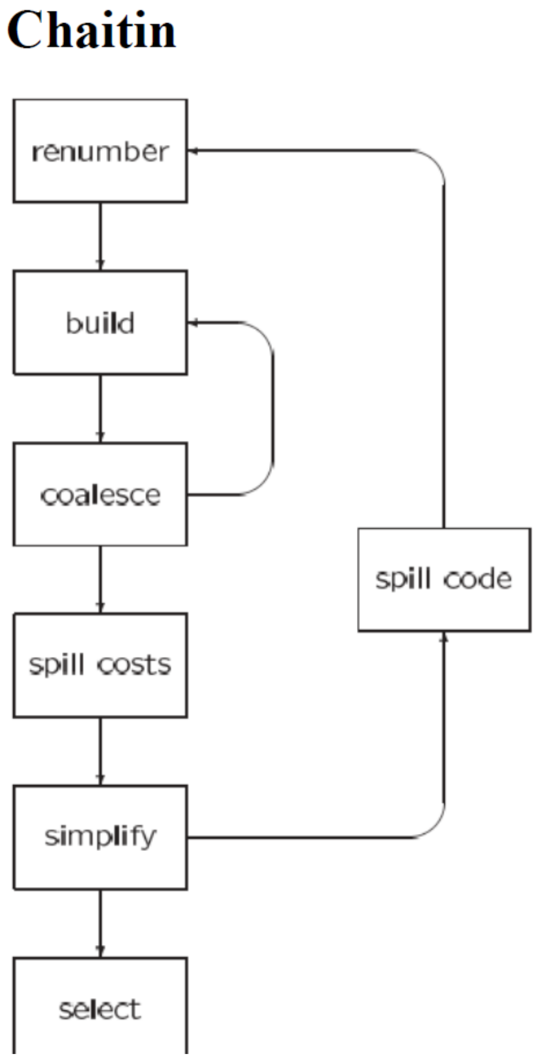
- Step 1 (simplify): simplify as much as possible without removing nodes that are the source or destination of a move (*move-related* nodes)
- Step 2 (coalesce): coalesce move-related nodes provided low-degree node results
- Step 3 (freeze): if neither steps 1 or 2 apply, freeze a move instruction: registers involved are marked not move-related and try step 1 again

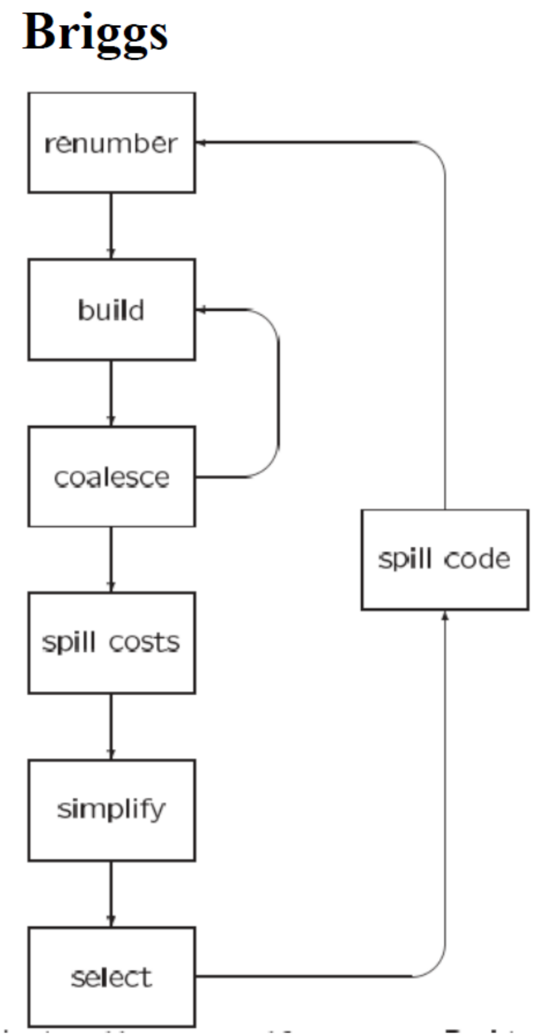
Overall Algorithm



Variations

May read papers for more variations





Questions?

Linear scan

- Given the live ranges of variables in a function, the algorithm scans all the live ranges in a single pass, allocating registers to variables in a greedy fashion.

- M. Poletto, V. Sarkar. *Linear scan register allocation*. 1999.

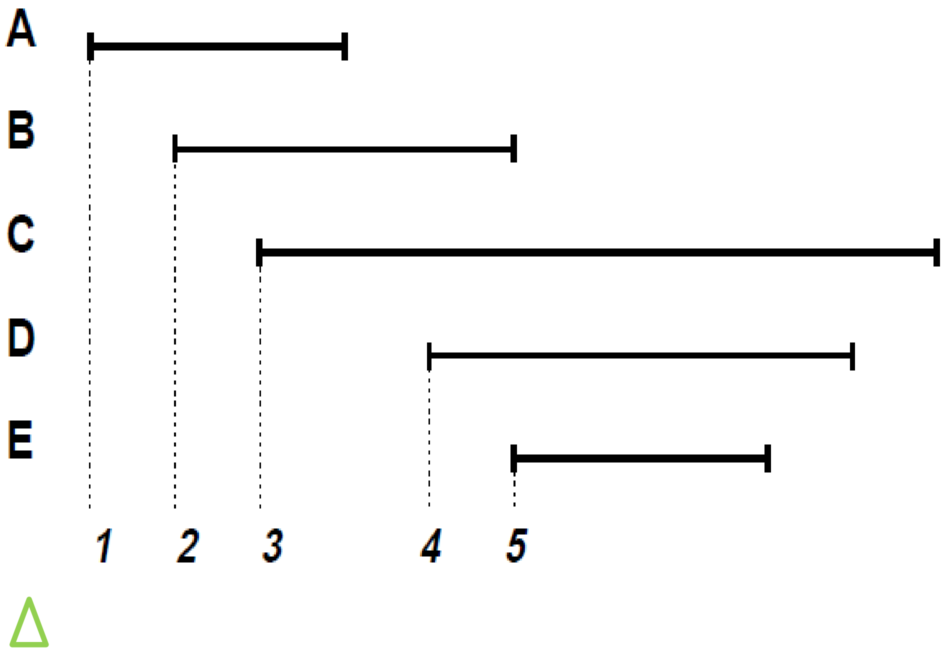
LINEARSCANREGISTERALLOCATION

```
active ← {}
foreach live interval i, in order of increasing start point
    EXPIREOLDINTERVALS(i)
    if length(active) = R then
        SPILLATINTERVAL(i)
    else
        register[i] ← a register removed from pool of free registers
        add i to active, sorted by increasing end point
```

```
EXPIREOLDINTERVALS(i)
    foreach interval j in active, in order of increasing end point
        if endpoint[j] ≥ startpoint[i] then
            return
        remove j from active
        add register[j] to pool of free registers
```

```
SpillAtInterval(i)
    spill ← last interval in active
    if endpoint[spill] > endpoint[i] then
        register[i] ← register[spill]
        location[spill] ← new stack location
        remove spill from active
        add i to active, sorted by increasing end point
    else
        location[i] ← new stack location
```

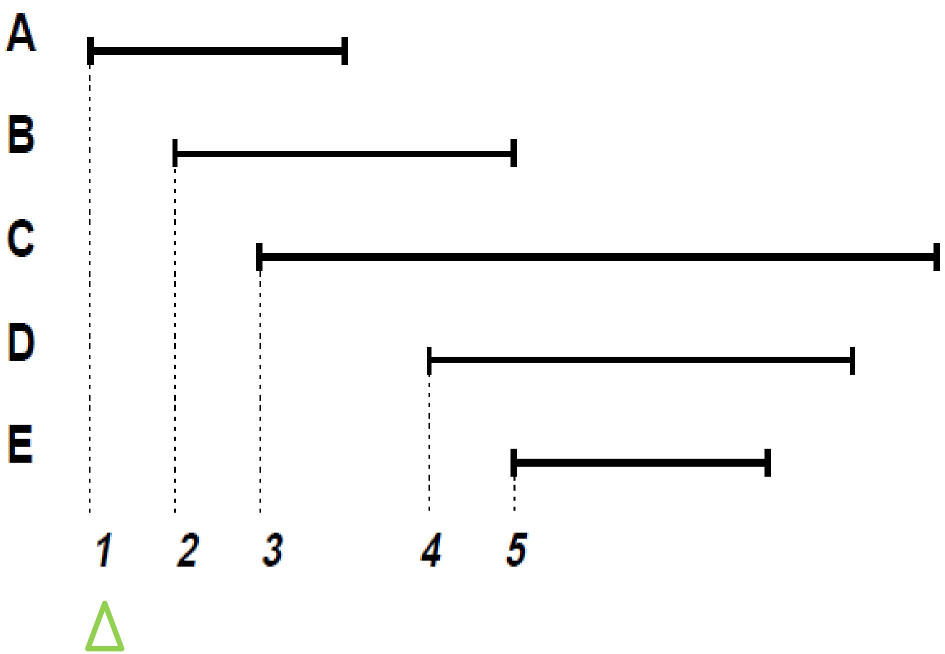
Example



- Initially, $active$ is empty

# of available registers is $R = 2$

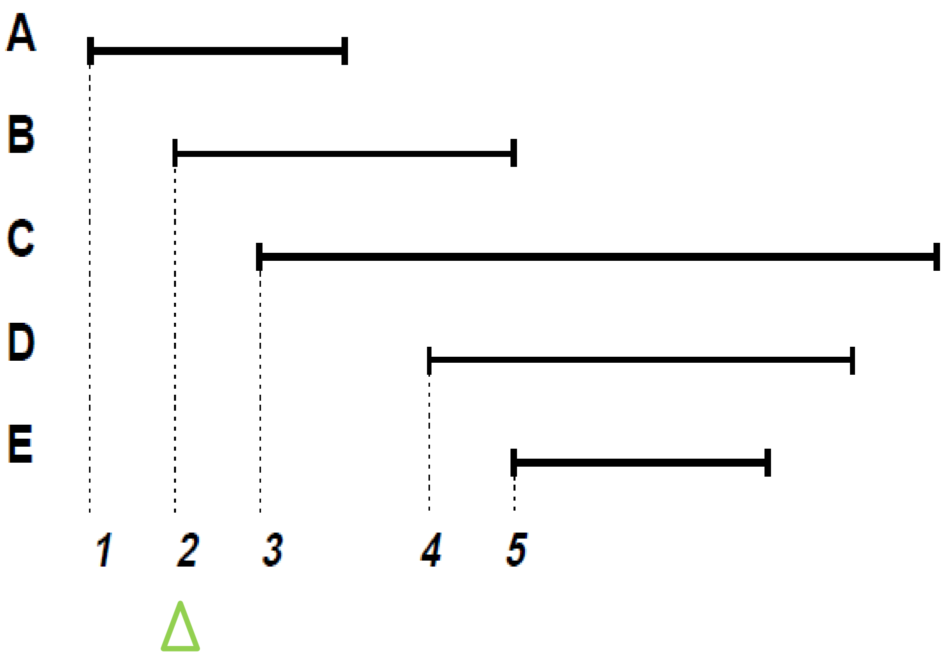
Example



- Step 1:
  $active = < A >$

# of available registers is $R = 2$

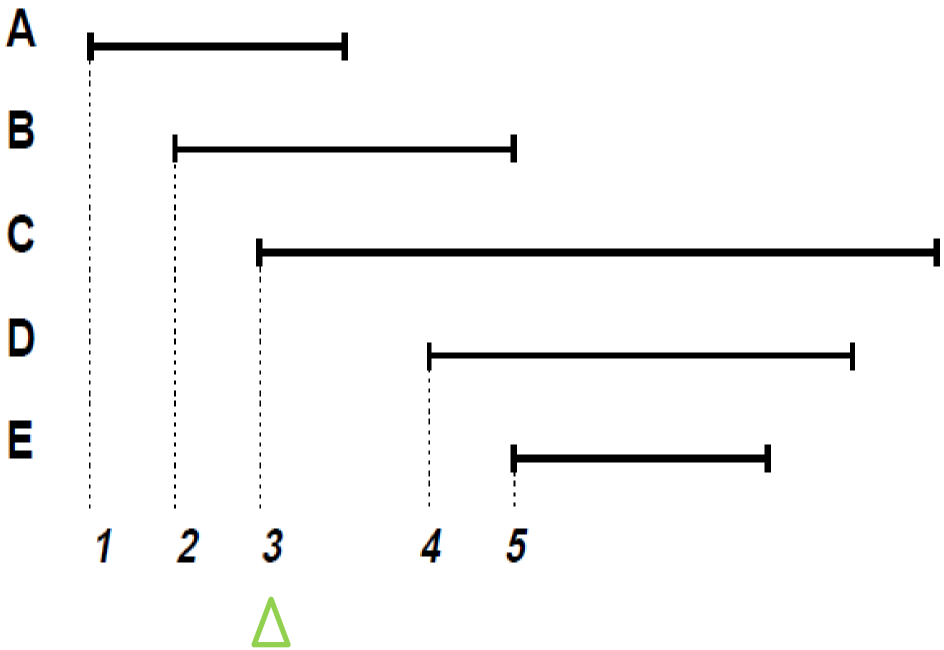
Example



- Step 2:
  $active = < A, B >$

# of available registers is $R = 2$

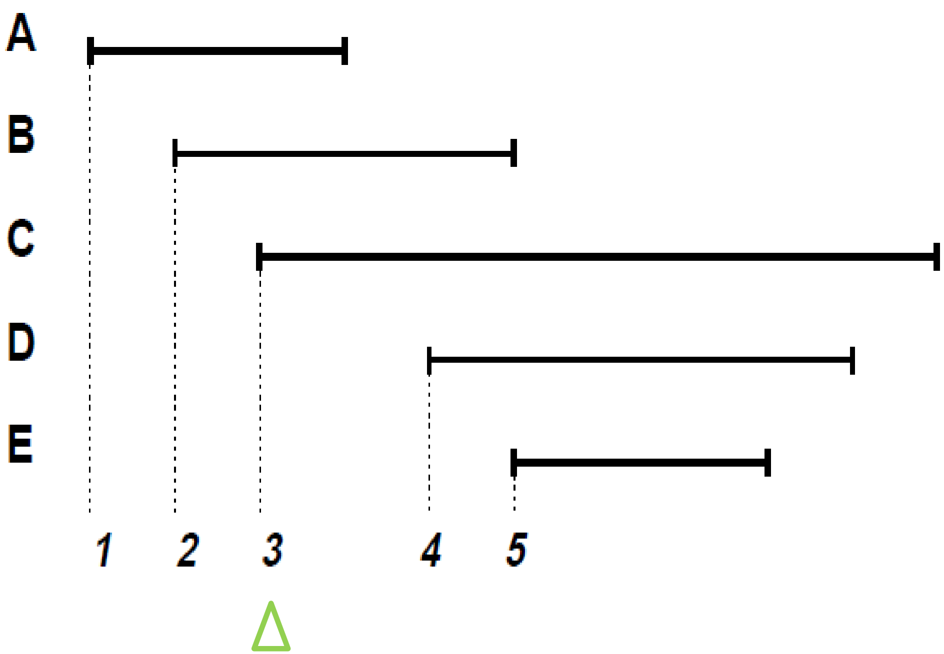
Example



- Step 3:
- 3 live intervals overlap

# of available registers is $R = 2$

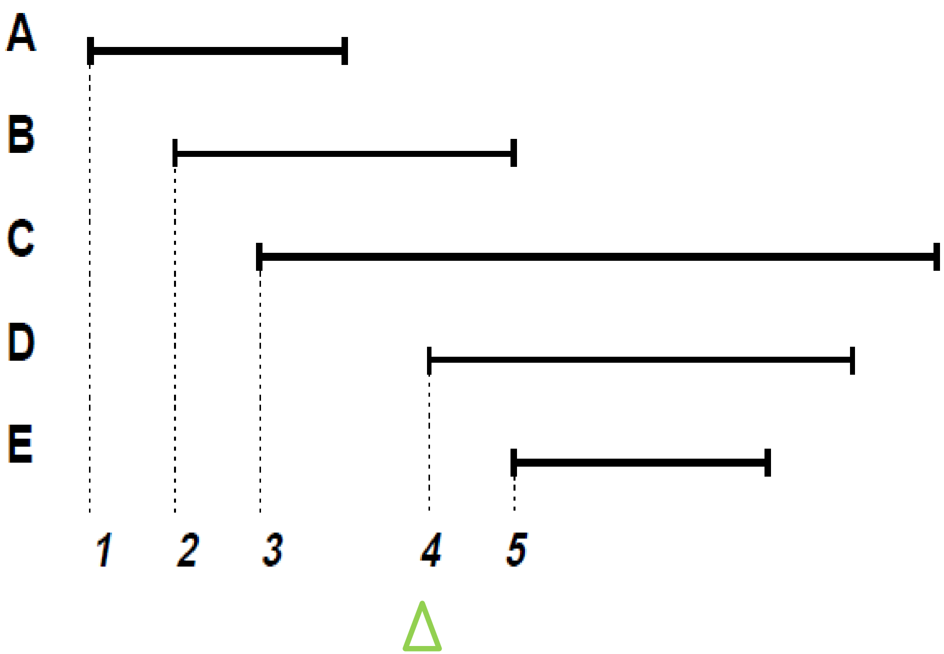
Example



# of available registers is $R = 2$

- Step 3:
- 3 live intervals overlap
- spills C, whose interval ends furthest away from the current point
- $active = < A, B >$

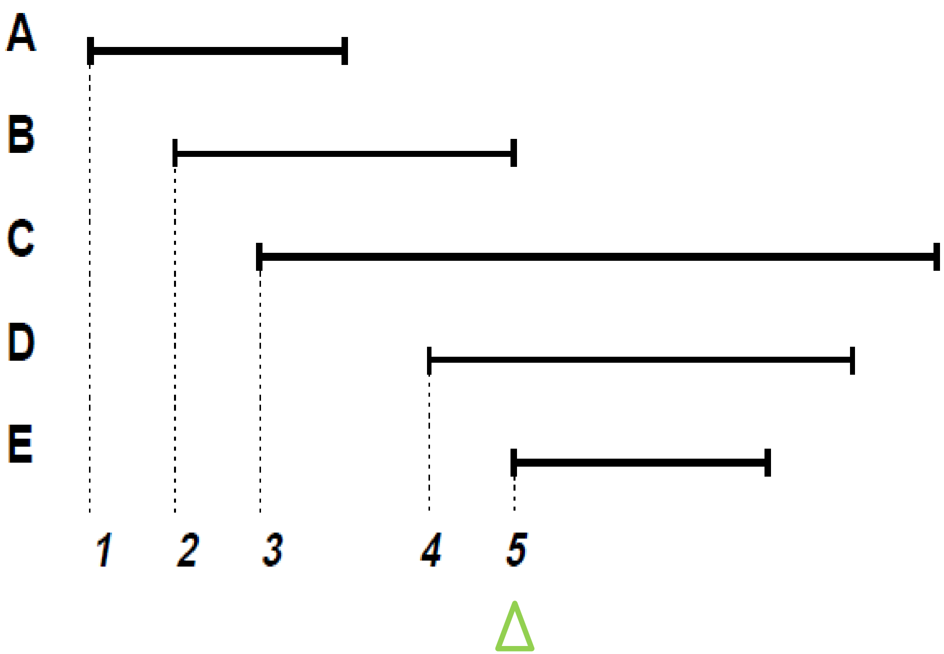
Example



- Step 4:
- A expires
- $active = < \cancel{A,} B, D >$

# of available registers is $R = 2$

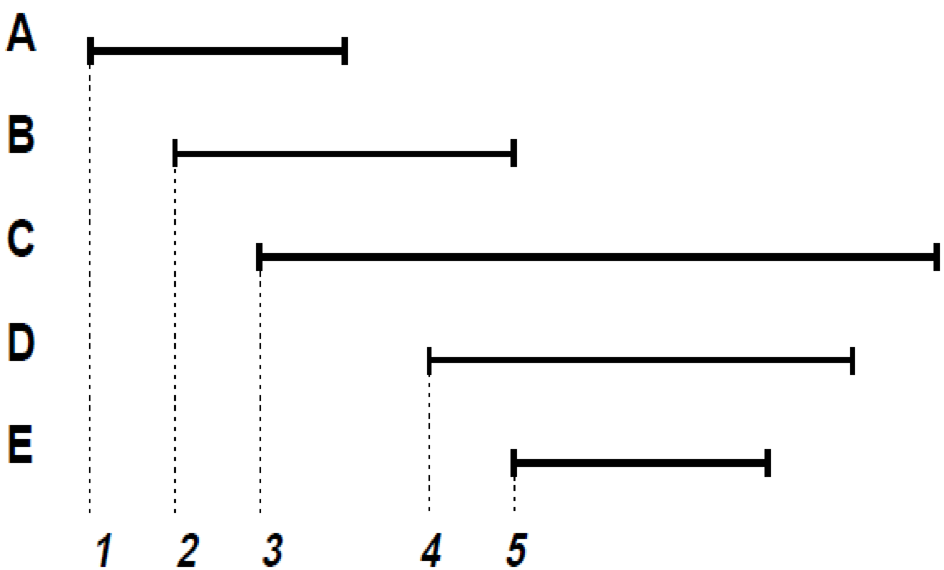
Example



- Step 5:
- B expires
- $active = < \cancel{B,} D, E >$

# of available registers is $R = 2$

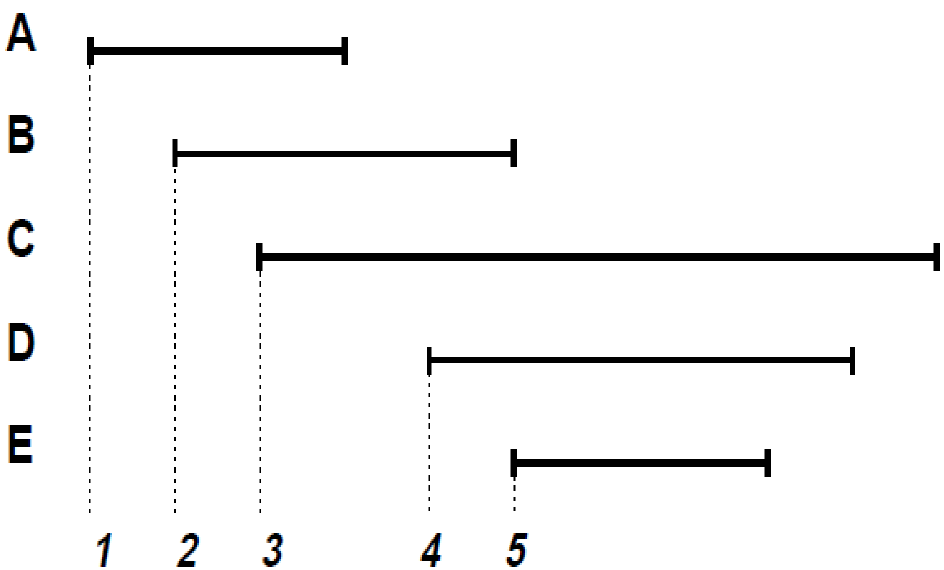
Example



- In the end, C is the only variable not allocated to a register.

\# of available registers is $R = 2$

Example



# of available registers is $R = 2$

- In the end, C is the only variable not allocated to a register.
- Otherwise, both one of A and B and one of D and E would have been spilled to memory.

Questions?

Conclusion

- # of slides
  - Graph coloring: ~50
  - Linear scan: ~10

Conclusion

- # of slides
  - Graph coloring: ~50
  - Linear scan: ~10
- Linear scan is much more simpler!
  - Only about 10% slower than a perfectly implemented graph coloring algorithm
  - And your code may not be that perfect ☺

Acknowledgements

- Graph coloring slides are adapted from *Register Allocation* by David Walker
- Linear scan pseudo-code and example are adapted from *Linear Scan Register Allocation* by M. Poletto and V. Sarkar.