Computer Science 160
Translation of Programming Languages

Instructor: Christopher Kruegel

Register Allocation

Register Allocation

- Main Idea: We want to replace temporary variables with some fixed set of registers

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

Liveness Analysis

- Live variable analysis (or simply liveness analysis) is a classic data-flow analysis to calculate the variables that are live at each point in the program.
- A variable is live at some point if it holds a value that may be needed in the future, or equivalently if its value may be read before the next time the variable is written to.
- Analysis is performed starting from the end of the function working towards the beginning → backwards analysis
- Compute def(inition) – use(age) regions: A variable is live between its (most recent) definition and (last) use

Liveness Analysis

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

Liveness Analysis

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

Liveness Analysis

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

Liveness Analysis

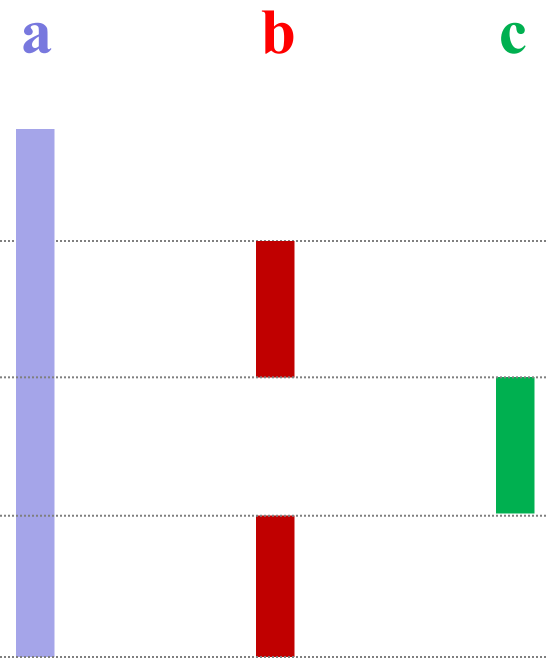
Instructions	Live vars
b = a + 2	
	b,a
c = b * b	
	a,c
b = c + 1	
	b,a
return b * a	

Liveness Analysis

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

Liveness Analysis

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



Interference Graph

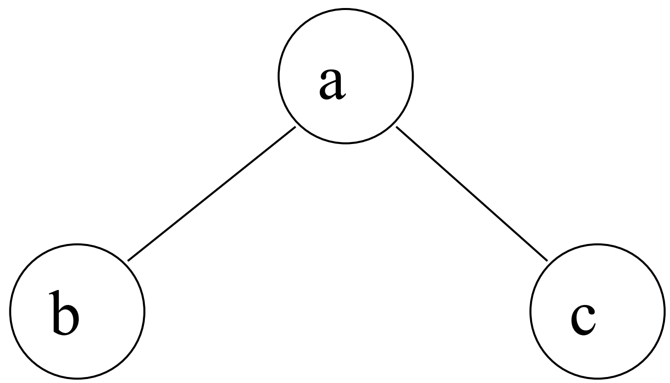
- Nodes of the graph = variables
- Edges connect variables that interfere with one another
- Nodes will be assigned a color corresponding to the register assigned to the variable
- Two colors cannot be next to one another in the graph

Interference Graph

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

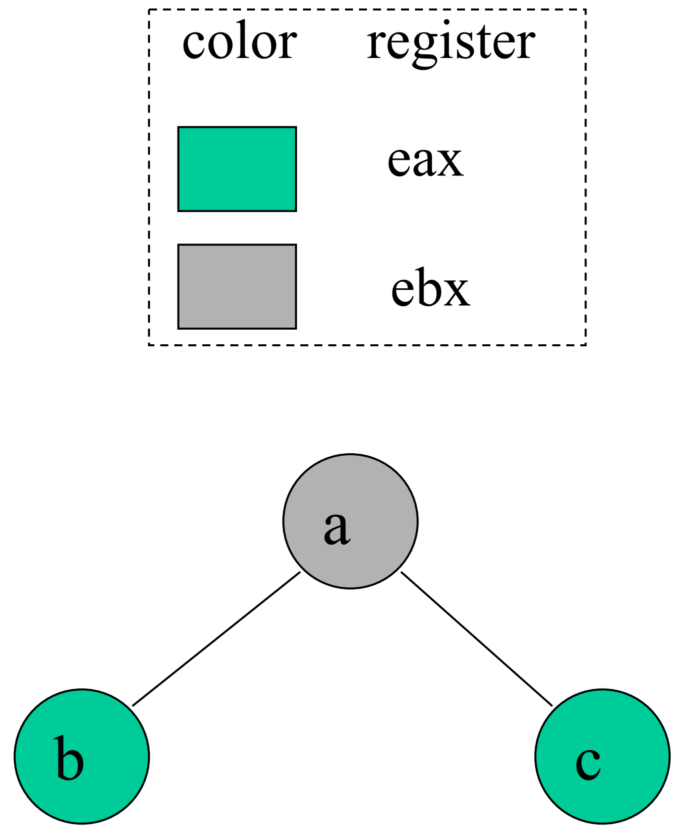
Interference Graph

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



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?
  - How do we choose registers to avoid move instructions?
  - 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

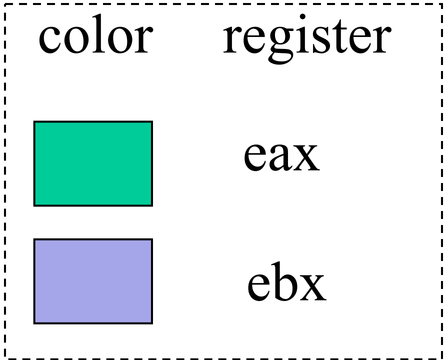
- Step 1 (Simplify): Find a node with at most K-1 edges and cut it out of the graph. Remember this node on a stack for later stages

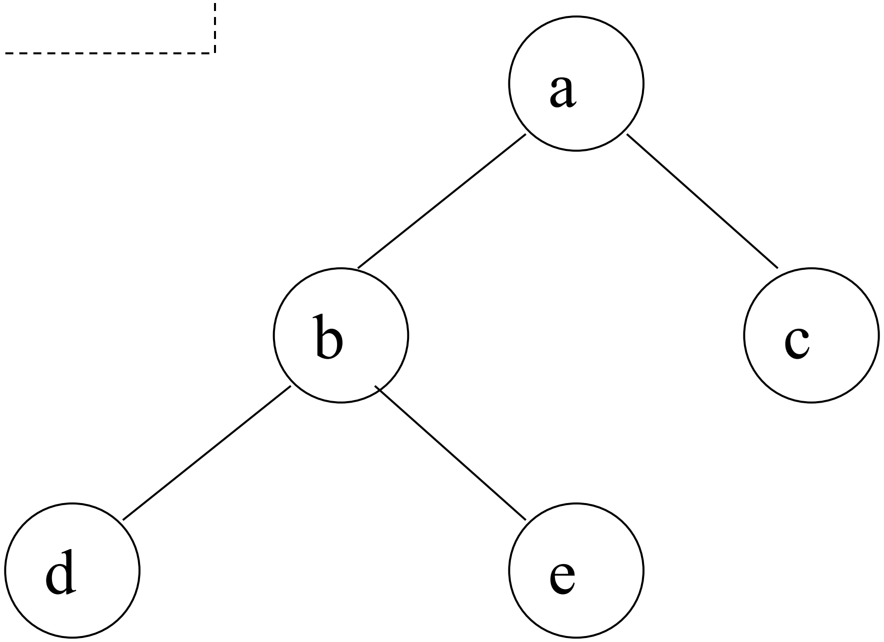
Coloring a Graph

- 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

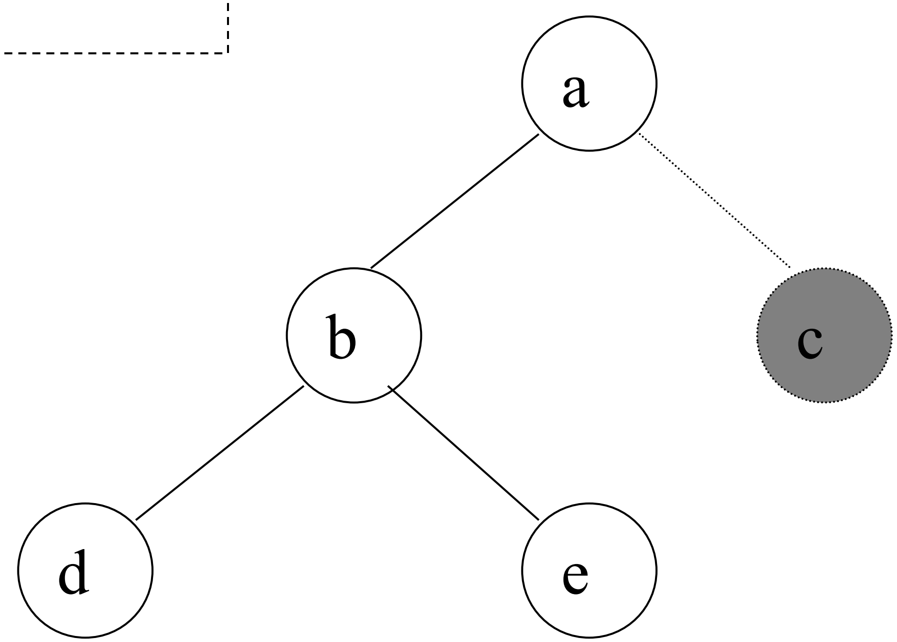




stack:

Coloring

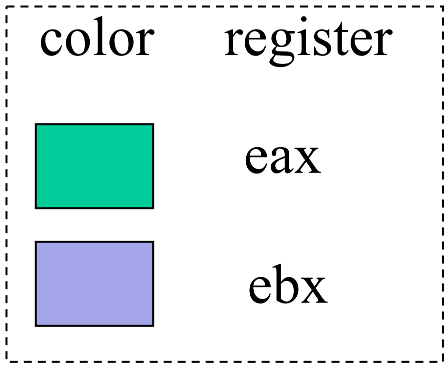
color	register
(green)	eax
(purple)	ebx

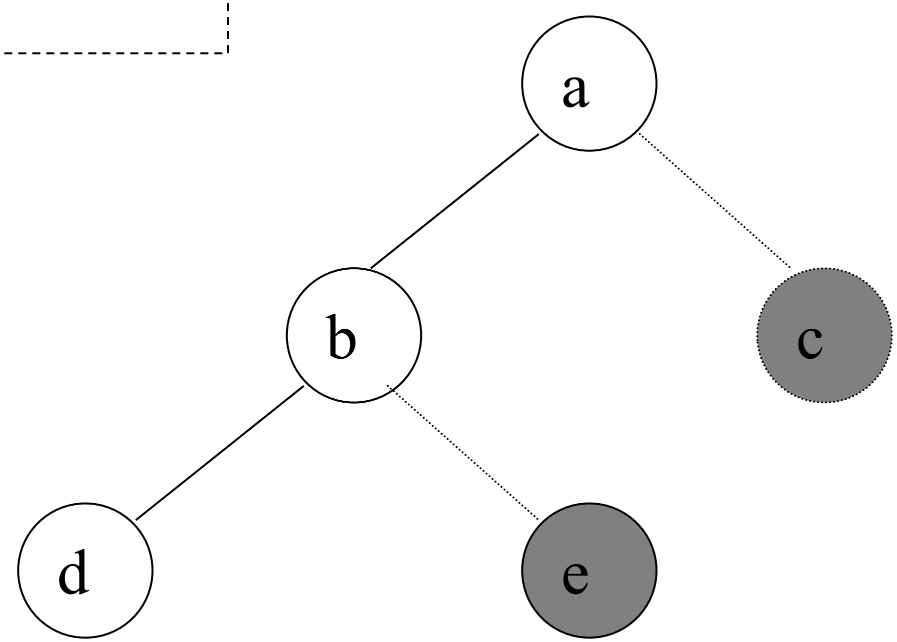


stack:

c

Coloring



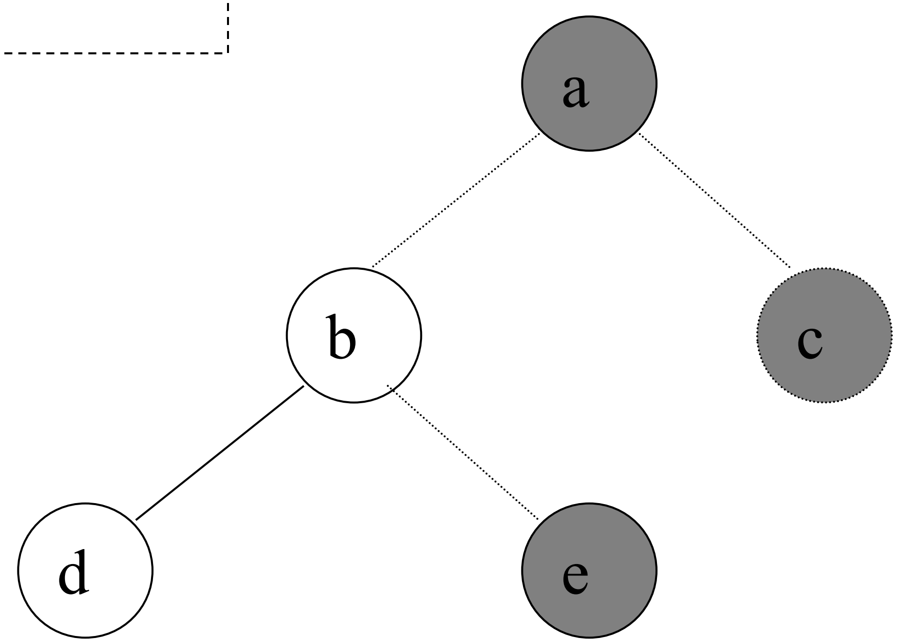


stack:

e
c

Coloring

color	register
(green)	eax
(purple)	ebx

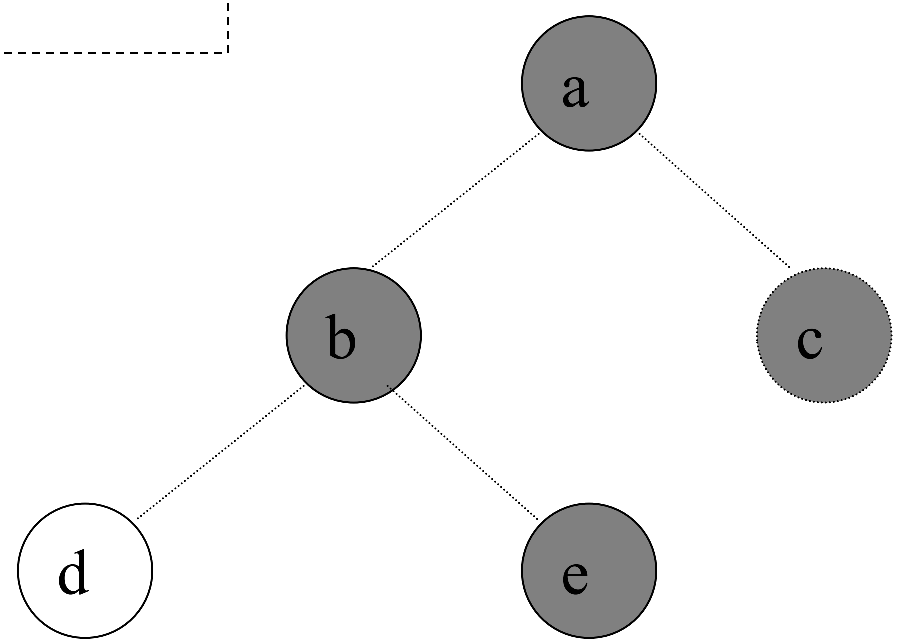


stack:

a
e
c

Coloring

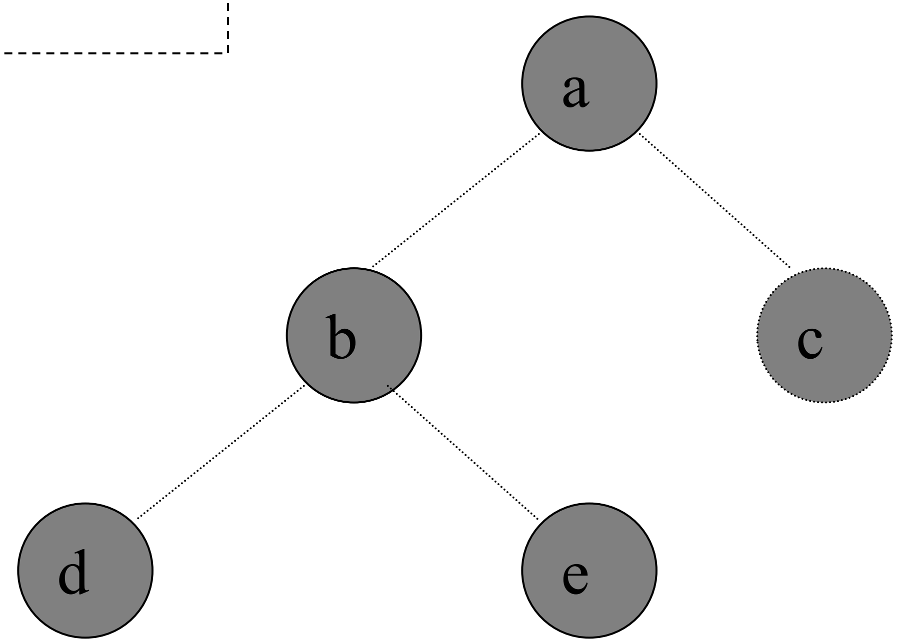
color	register
(green)	eax
(purple)	ebx



stack:
b
a
e
c

Coloring

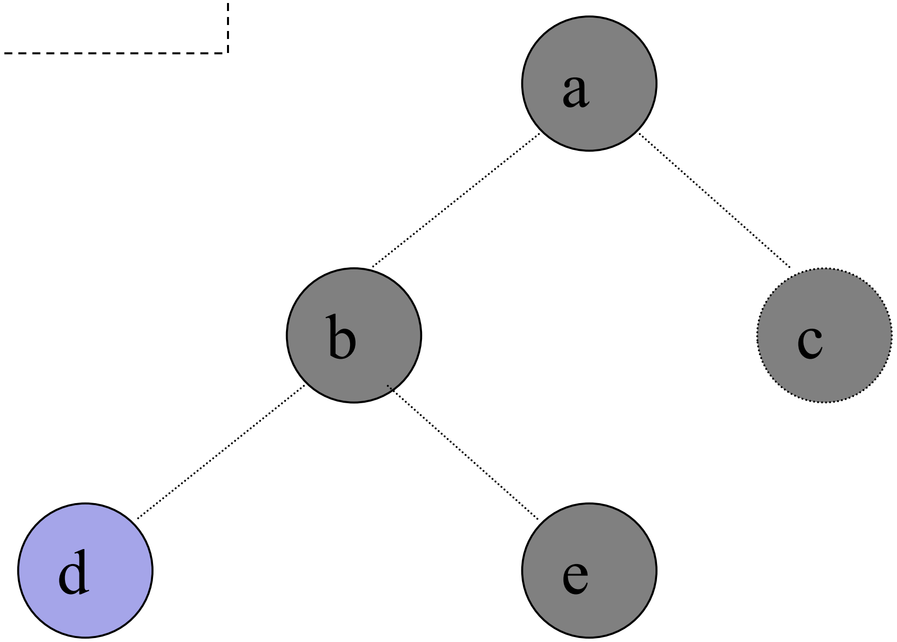
color	register
(green)	eax
(purple)	ebx



stack:
d
b
a
e
c

Coloring

color	register
(green)	eax
(blue)	ebx

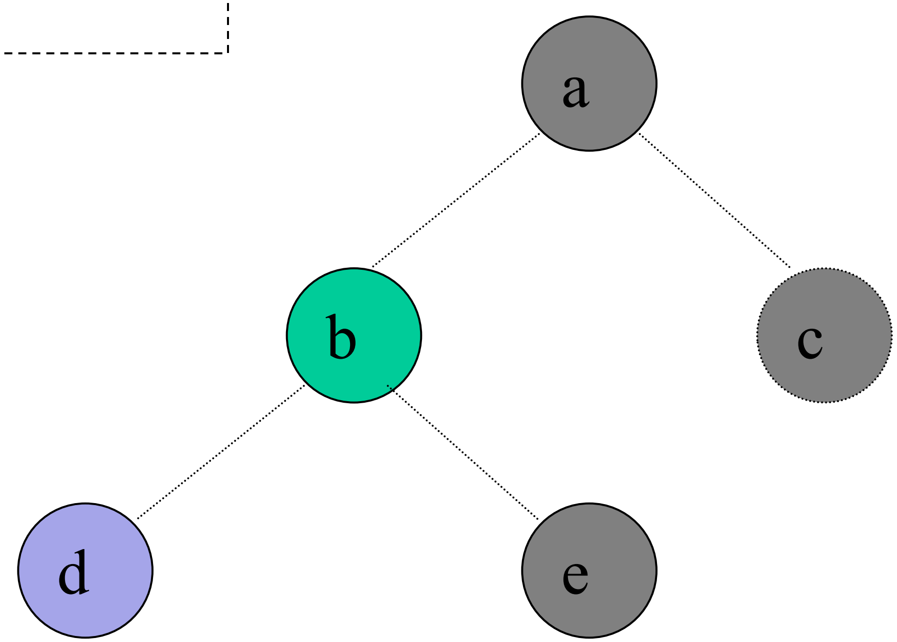


stack:

b
a
e
c

Coloring

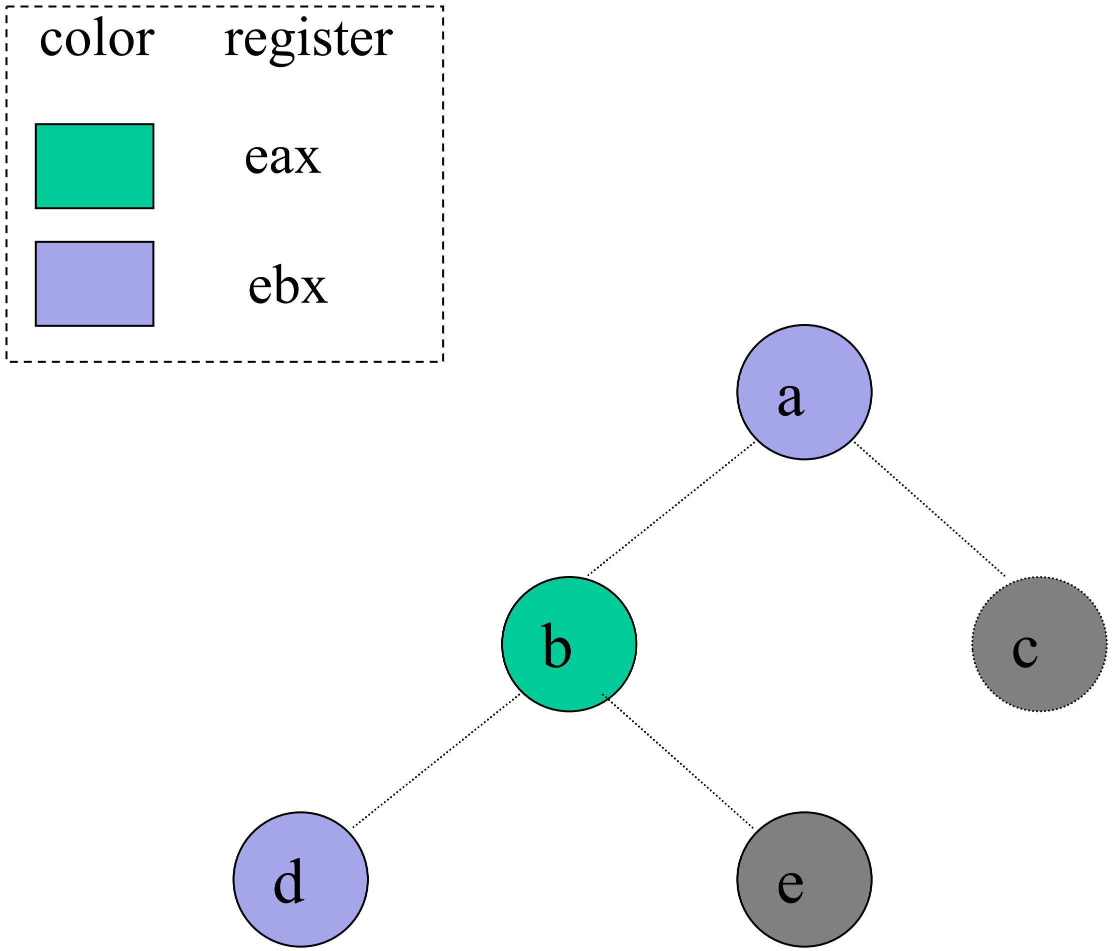
color	register
(green)	eax
(gray)	ebx



stack:

a
e
c

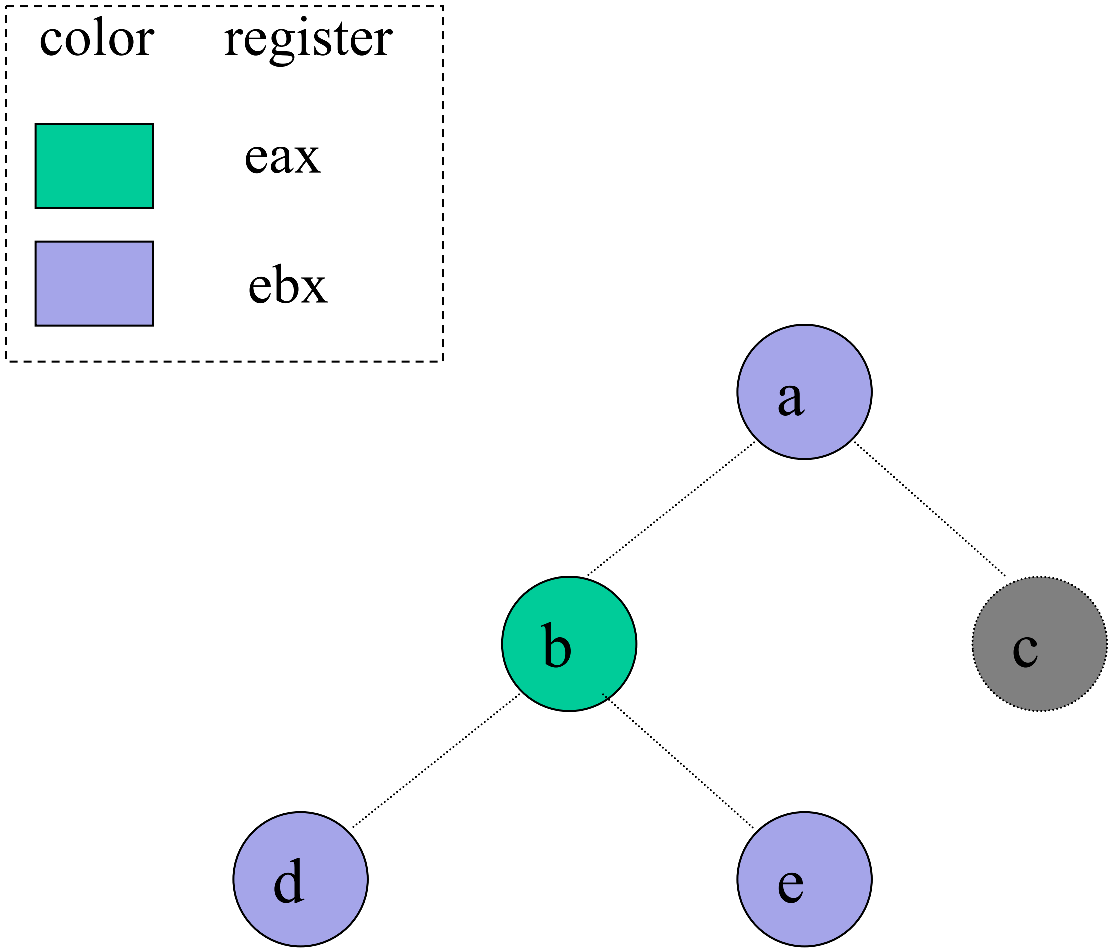
Coloring



stack:

e
c

Coloring

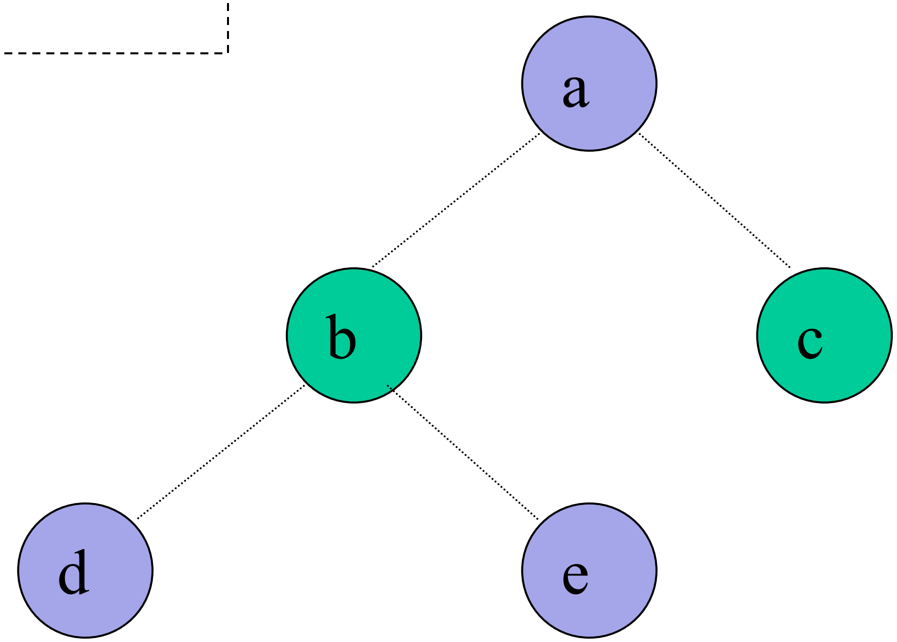


stack:

c

Coloring

color	register
green	eax
purple	ebx



stack:

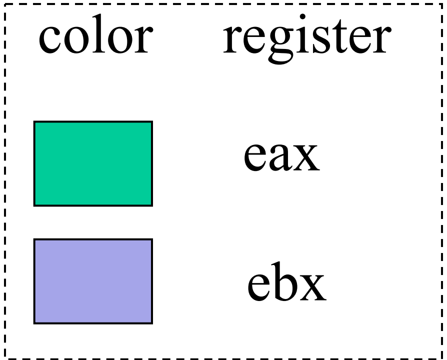
Failure

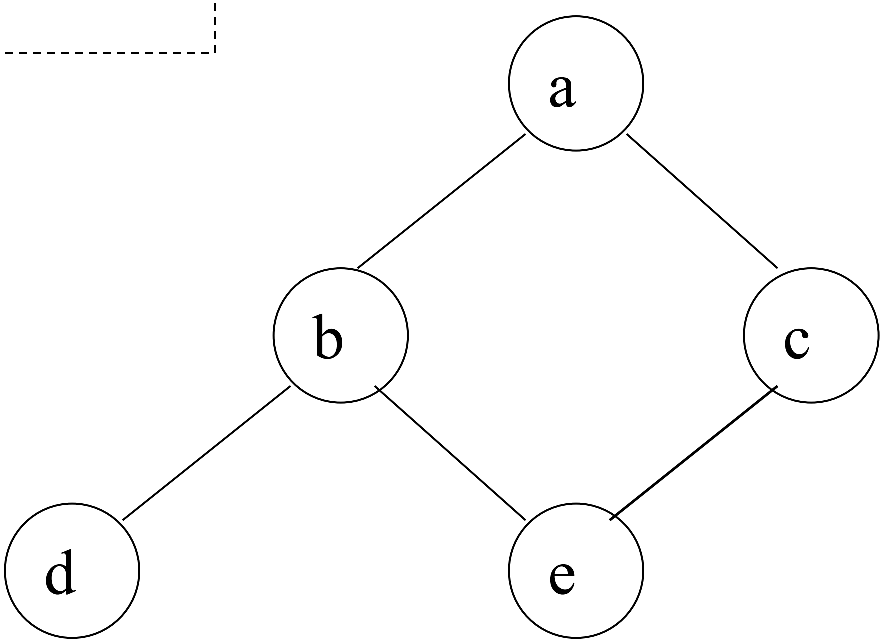
- If the graph cannot be colored, it will eventually be simplified to graph in which every node has at least K neighbors

- 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

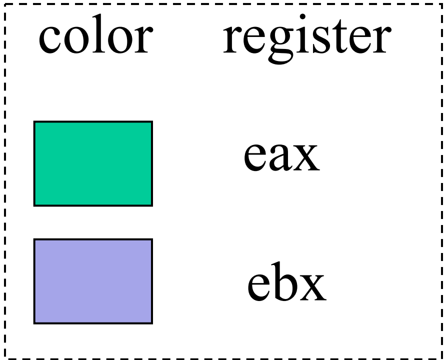
Coloring



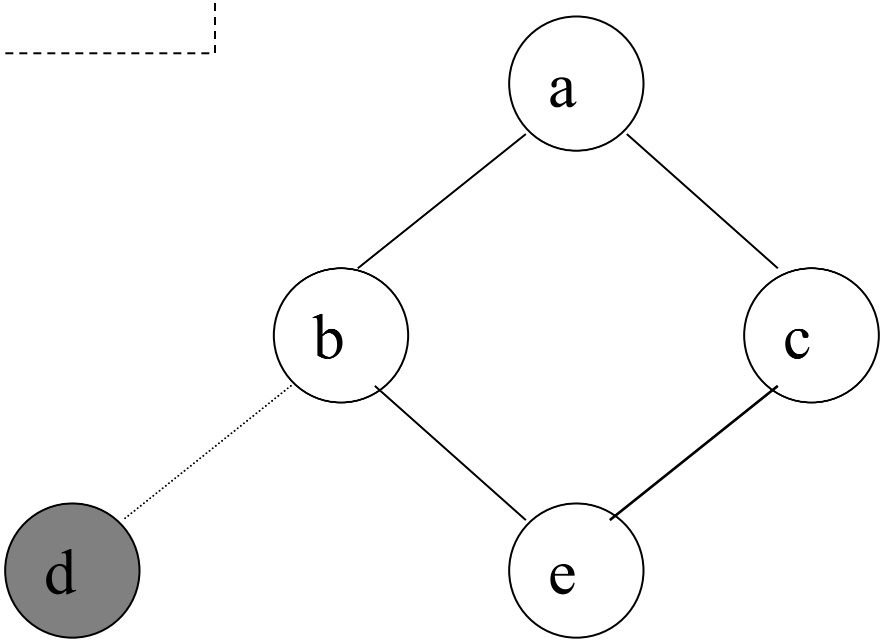


stack:

Coloring



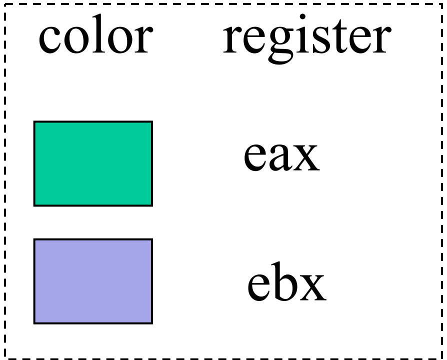
All nodes have
(at least) 2 neighbors!

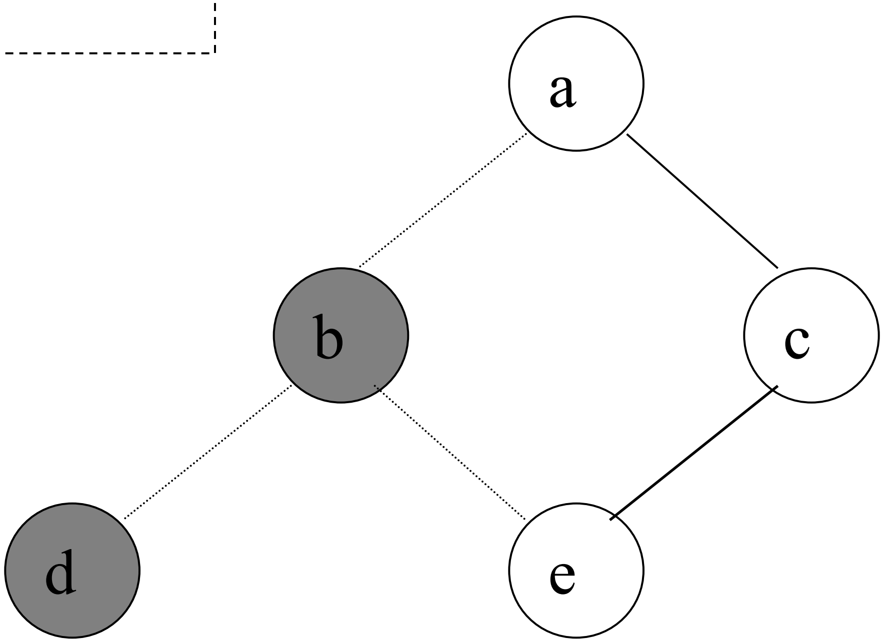


stack:

d

Coloring

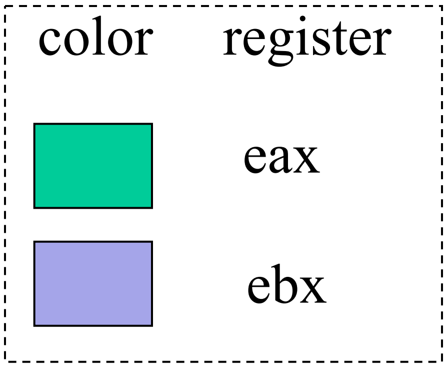


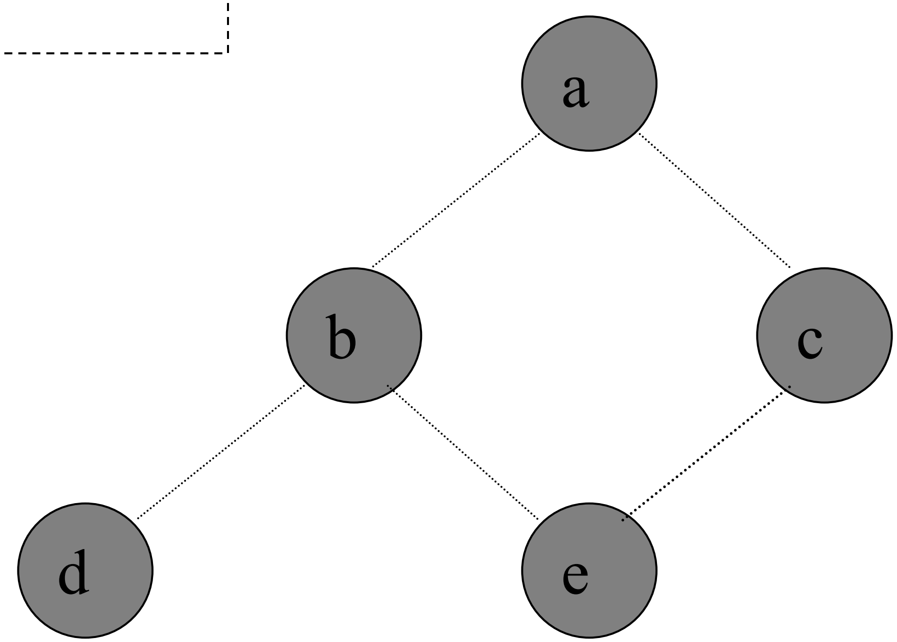


stack:

b
d

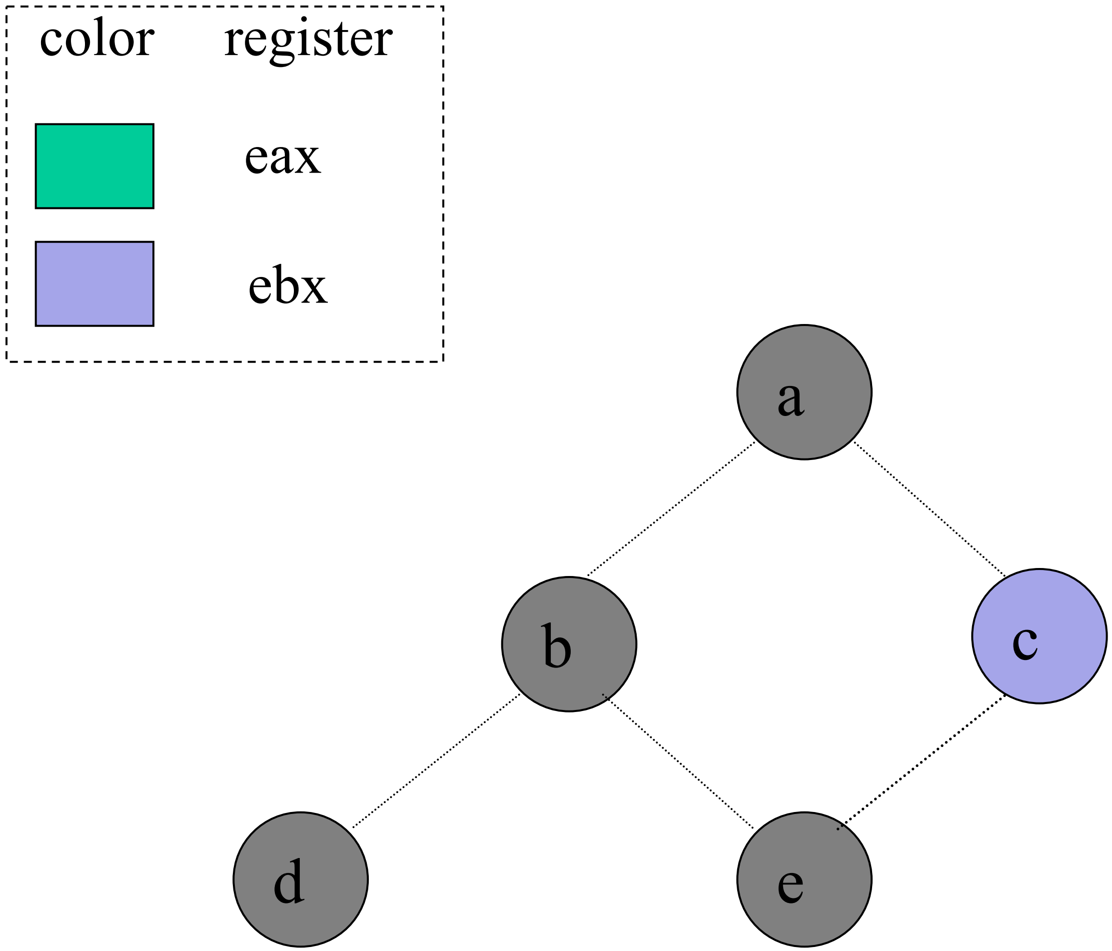
Coloring





stack:
c
e
a
b
d

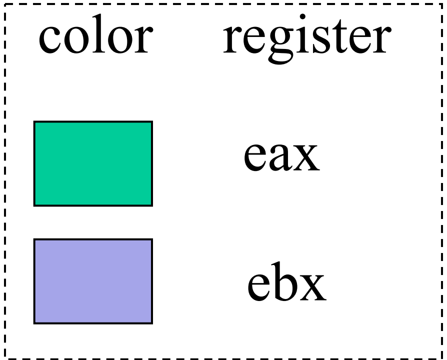
Coloring

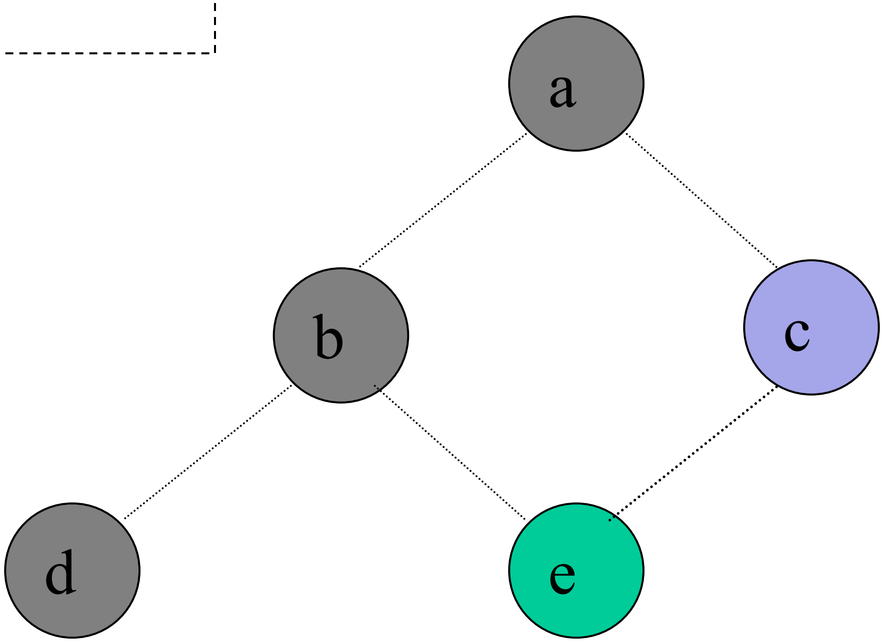


stack:

e
a
b
d

Coloring

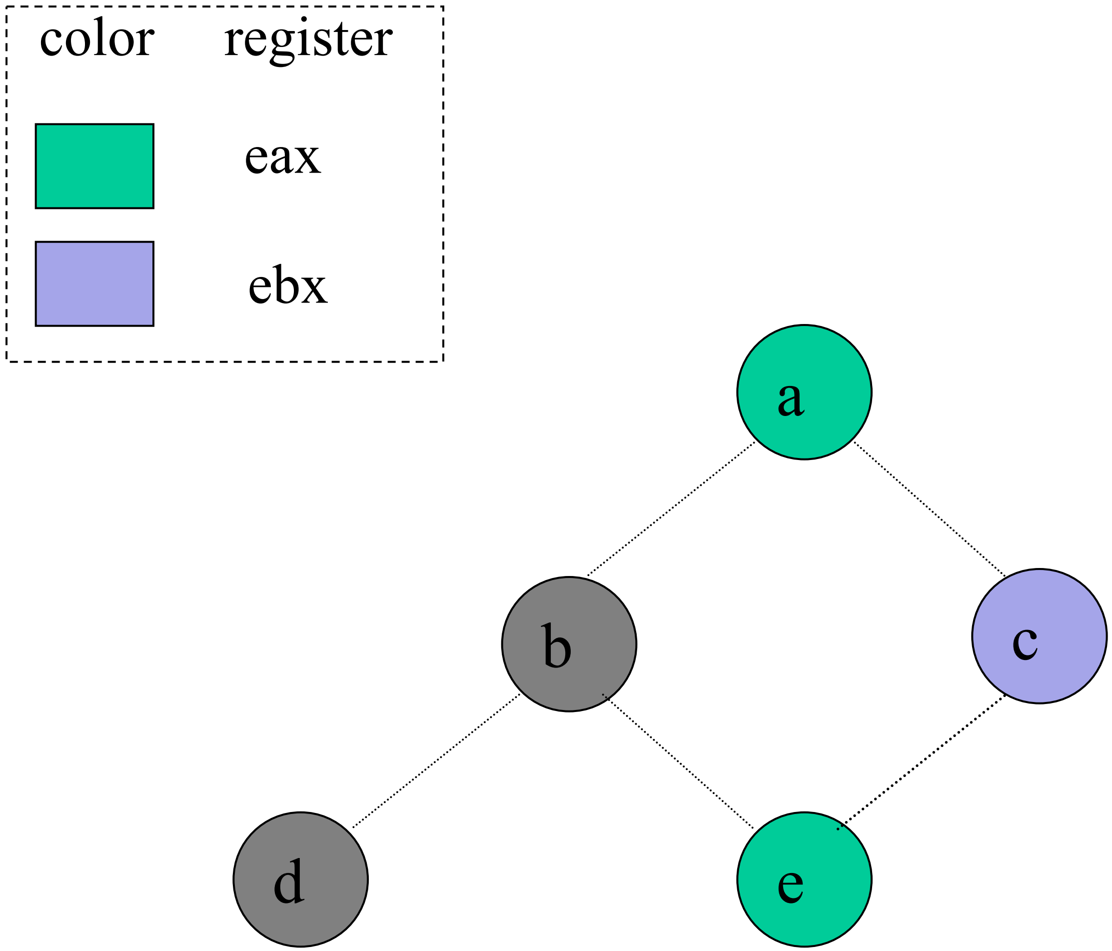




stack:

a
b
d

Coloring

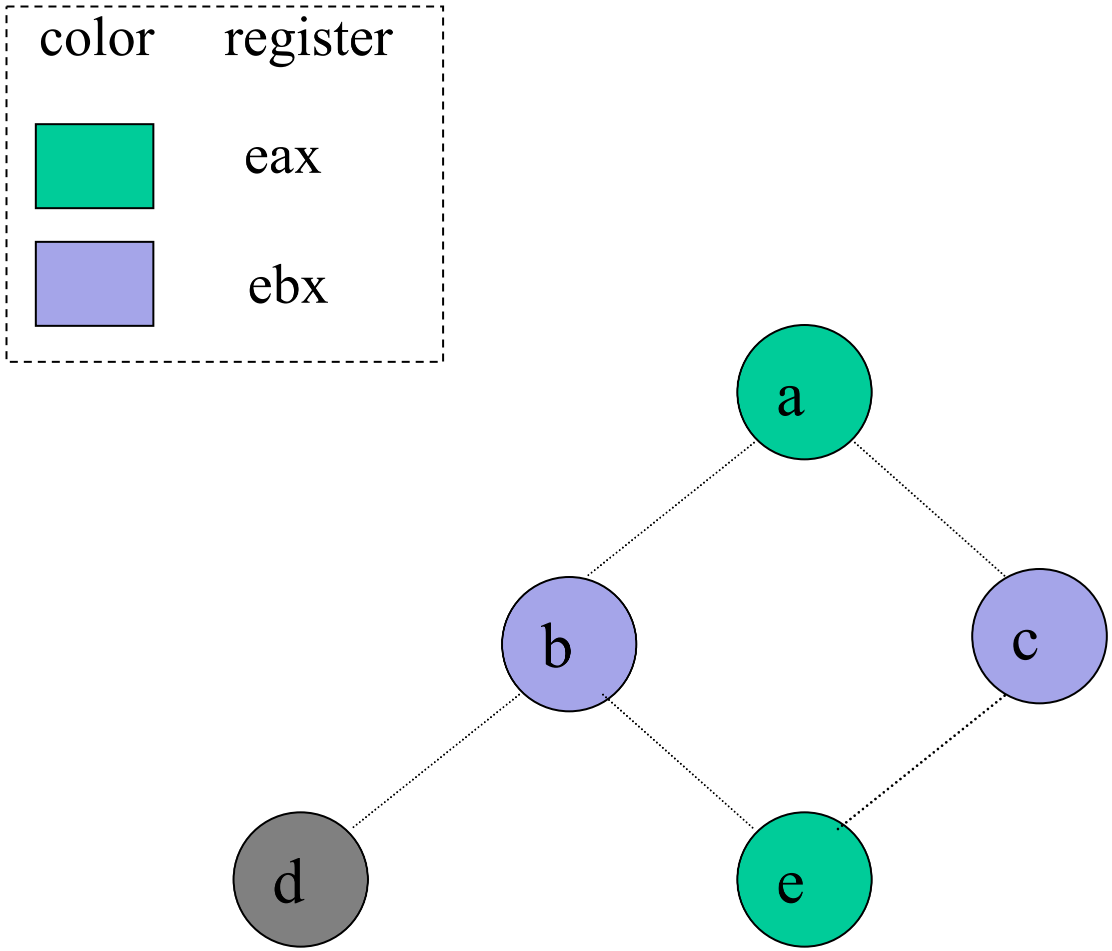


stack:

b

d

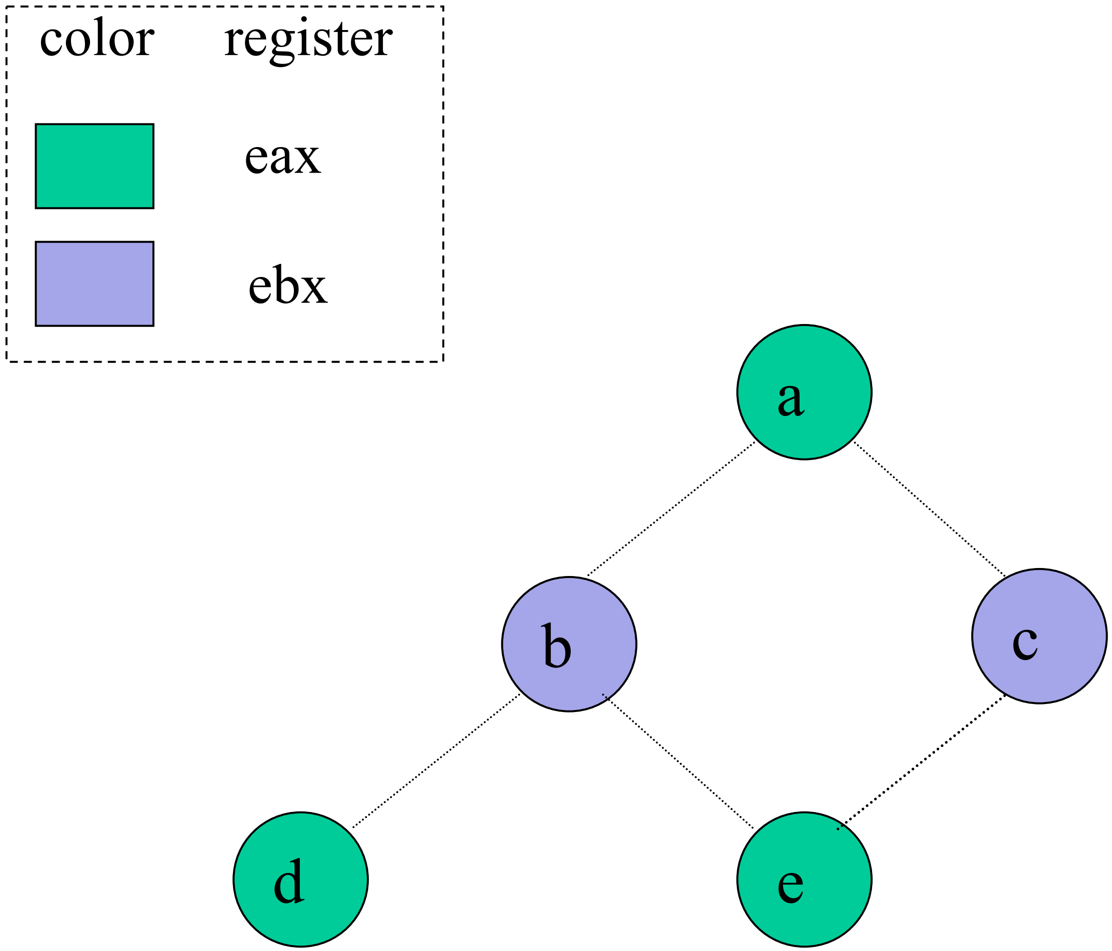
Coloring



stack:

d

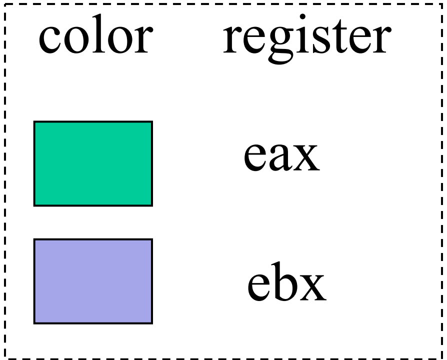
Coloring



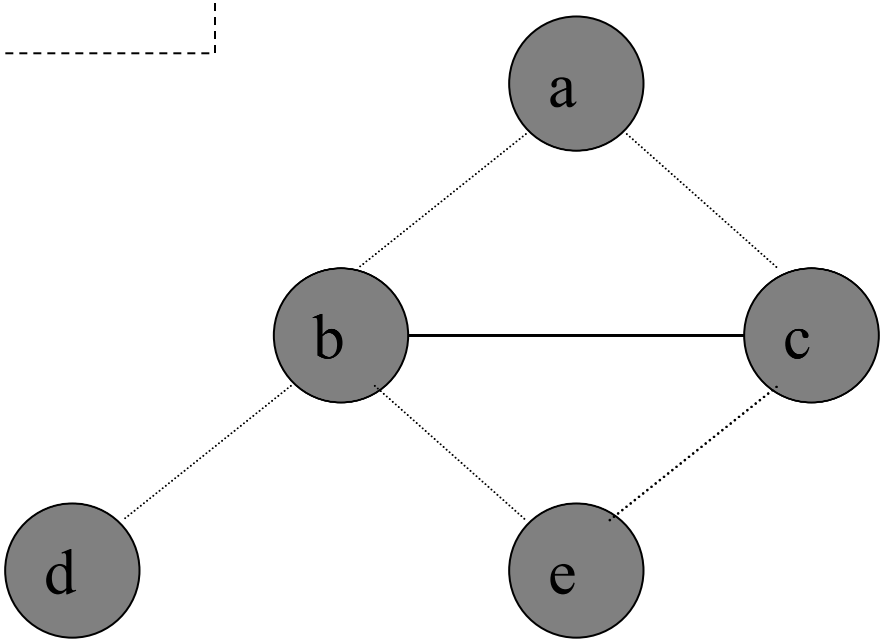
stack:

We got lucky!

Coloring

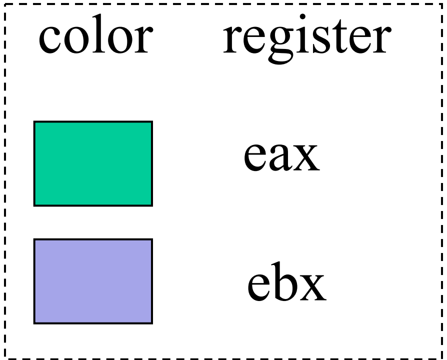


Some graphs cannot be colored in K colors

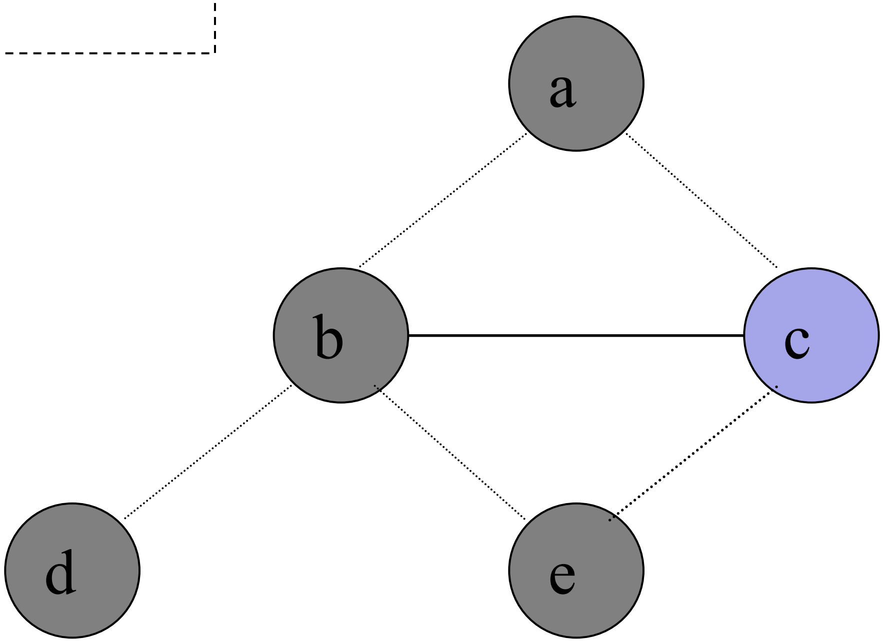


stack:
c
b
e
a
d

Coloring



Some graphs cannot be colored in K colors



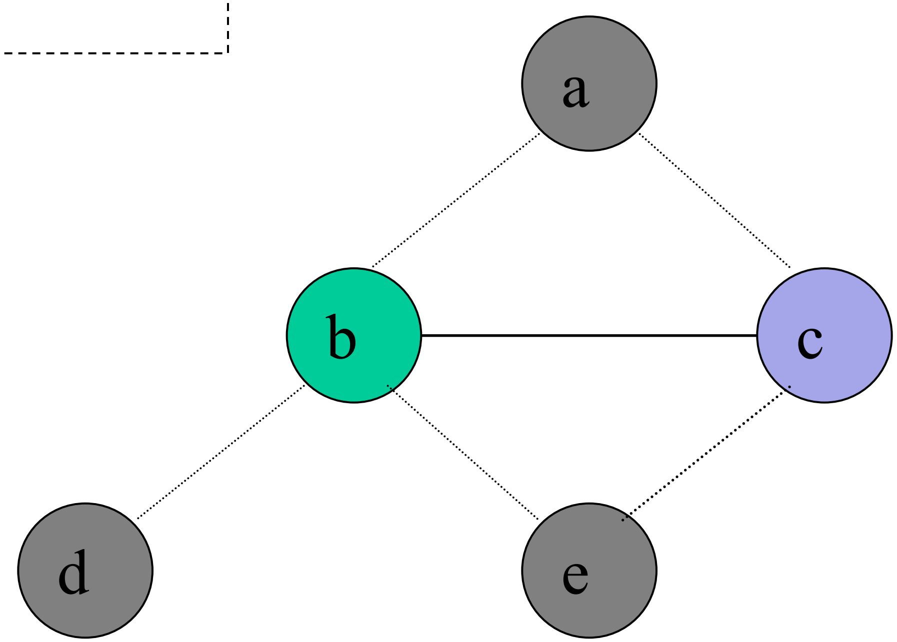
stack:

b
e
a
d

Coloring

color	register
(green)	eax
(purple)	ebx

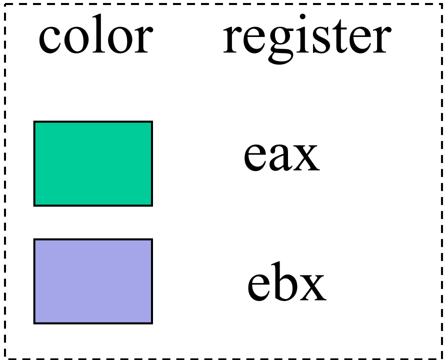
Some graphs cannot be colored in K colors



stack:

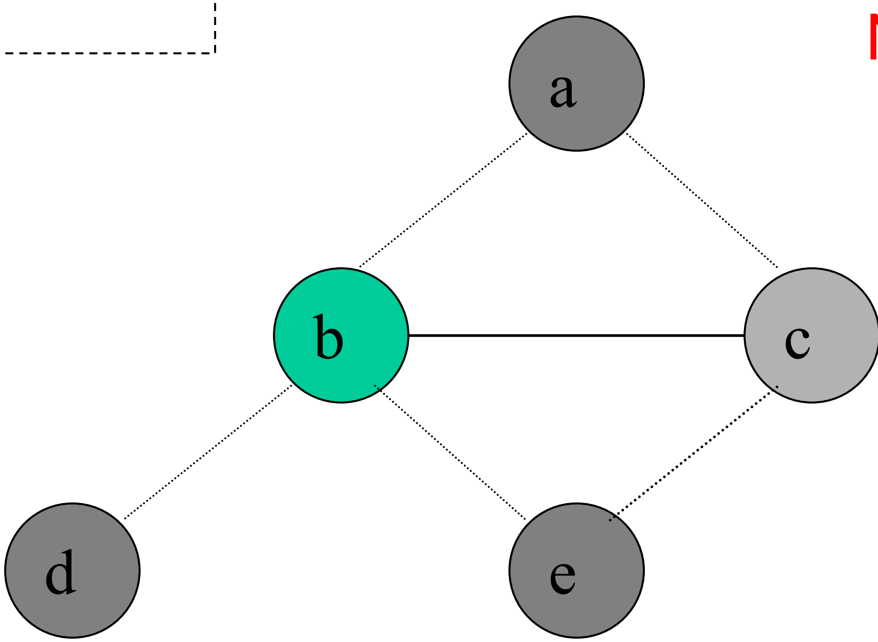
e
a
d

Coloring



Some graphs cannot be colored in K colors

No colors left for e!



stack:

e
a
d

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?
  - Naive 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?
  – Naive 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

Precolored Nodes

- Some variables are pre-assigned to registers
  - Eg: mul on x86/pentium
    - uses eax; defines eax, edx
  - Eg: call on x86/pentium
    - Defines (overwrites) caller-save registers eax, ecx, edx
- Treat these registers as special temporaries; before beginning, add them to the graph with their colors

Precolored Nodes

- Cannot 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

Summary

- Register allocation has three major parts
  - Liveness analysis
  - Graph coloring
  - Program transformation (spilling and optimizations)