TDDE73 - Programmering, datastrukturer och algoritmer

Grafer

Christoffer Holm

Institutionen för datavetenskap

LINKÖPING UNIVERSITY

1. Repetition: Grafer
2. Repetition: Riktade grafer
3. Representation av grafer
4. Grafsökning
5. Heap

LINKÖPING UNIVERSITY

1 Repetition: Grafer
2 Repetition: Riktade grafer
3 Representation av grafer
4 Grafsökning
5 Heap

LINKÖPING UNIVERSITY

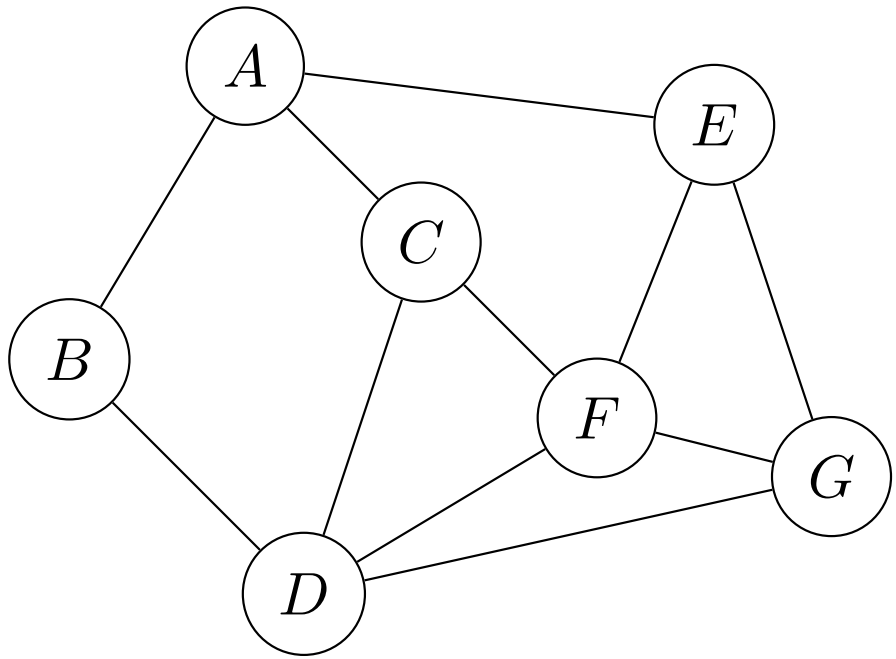
Grafer

Sammanfattning

- Vi börjar denna föreläsning med att sammanfatta och påminna om grundläggande begrepp och terminologi kring grafer.
- Detta ämne har mycket överlap med optimeringslära och andra kurser, men i den här föreläsningen lägger vi större vikt vid den datavetenskaplig sidan.

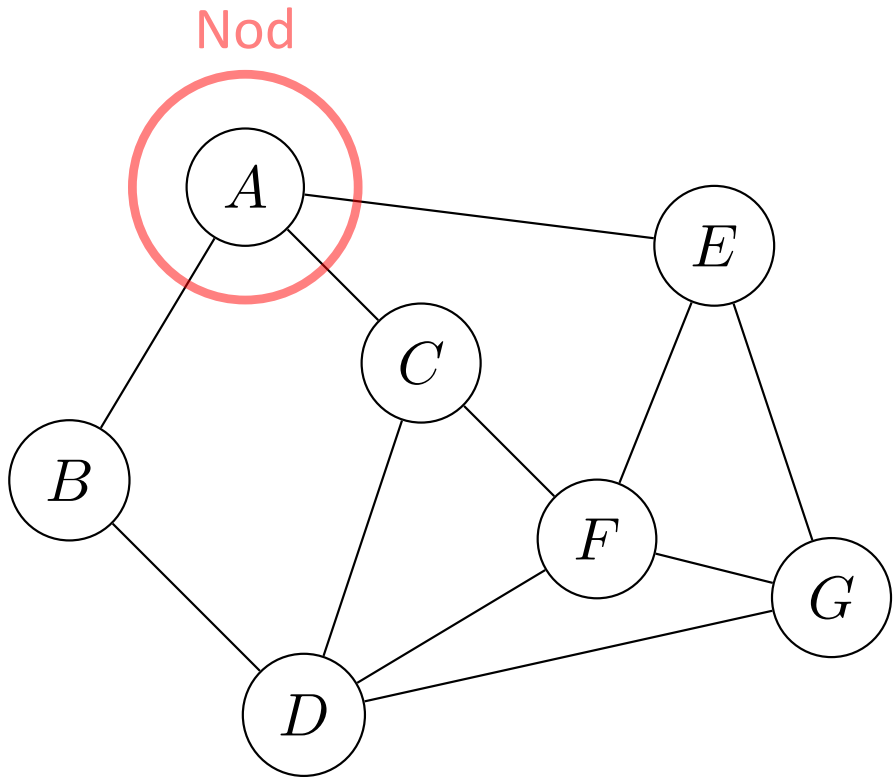
Grafer

Vad är en graf?



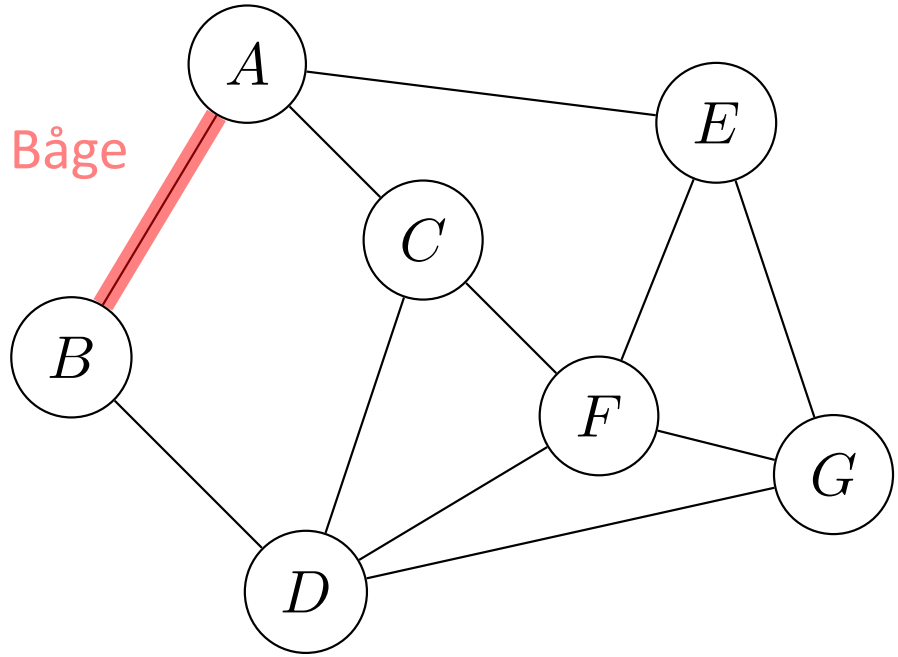
Grafer

Vad är en graf?



Grafer

Vad är en graf?



Grafer

Vad är en graf?

- Grafer studeras flitigt inom matematiken, t.ex. i kurserna TATA32 (diskret matematik) och TATA64 (grafteor).
- Grafer används i allmänhet för att modellera relationer/associationer mellan olika objekt. Det är en matematisk abstraktion som är oerhört användbar inom många fält.
- I datavetenskapen ser vi grafer som en icke-linjär datastruktur som tillåter mer flexibilitet än andra mer traditionella datastrukturer.

Grafer

Formell definition

Definition: Graf

En graf $G = (V, E)$ är ett par av två mängder: V och E, där E består av par av element från V, d.v.s. $E \subseteq V \times V$.

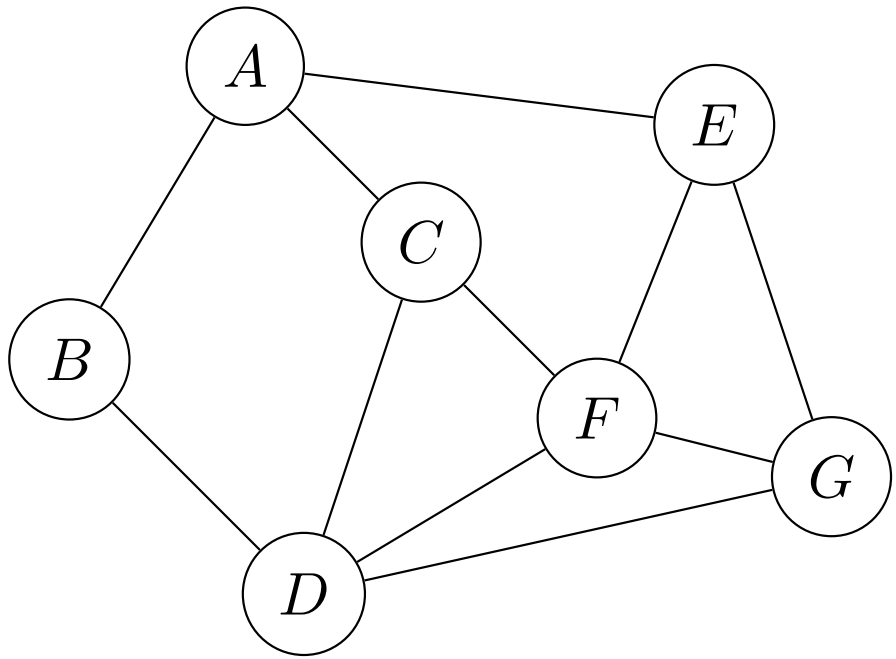
Grafer

Formell defintion

- V motsvarar noderna i grafen. Mängden består av *godtyckliga* objekt, exakt vad det är beror på tillämpningsområde.
- E motsvarar bågarna i grafen. Dessa definieras av vilka noder som bågen länkar ihop. Vad denna association av noder betyder beror på tillämpningsområde.

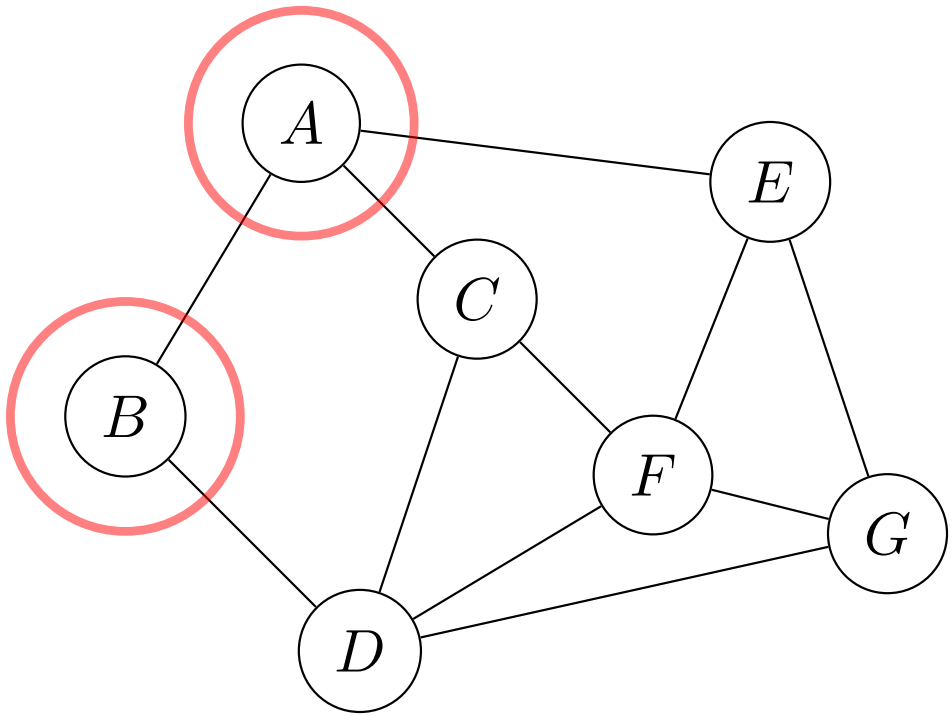
Grafer

Grannskap



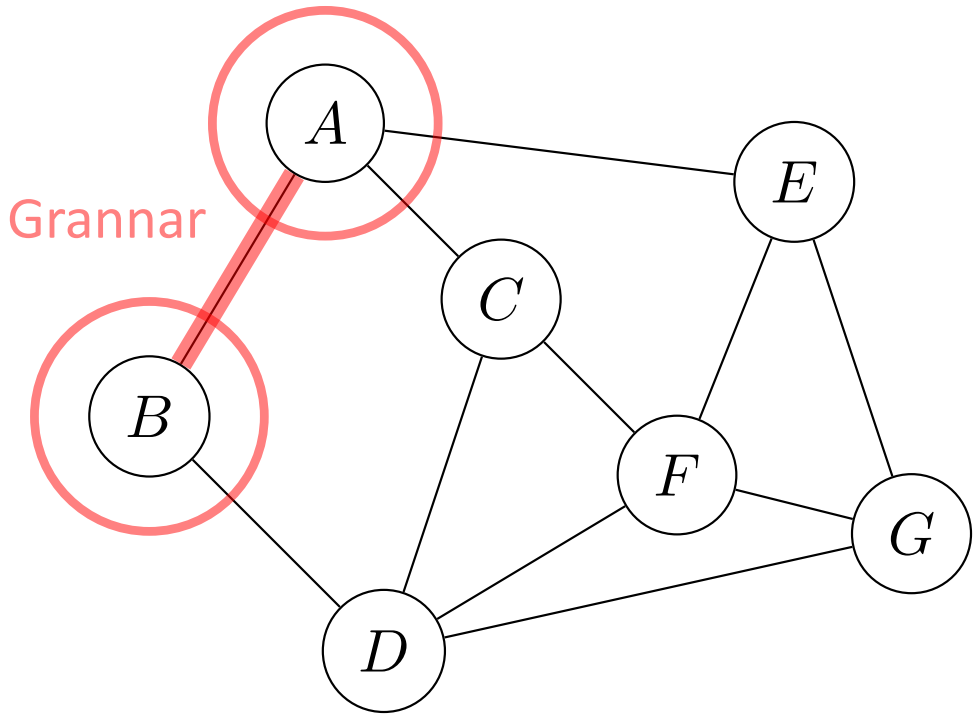
Grafer

Grannskap



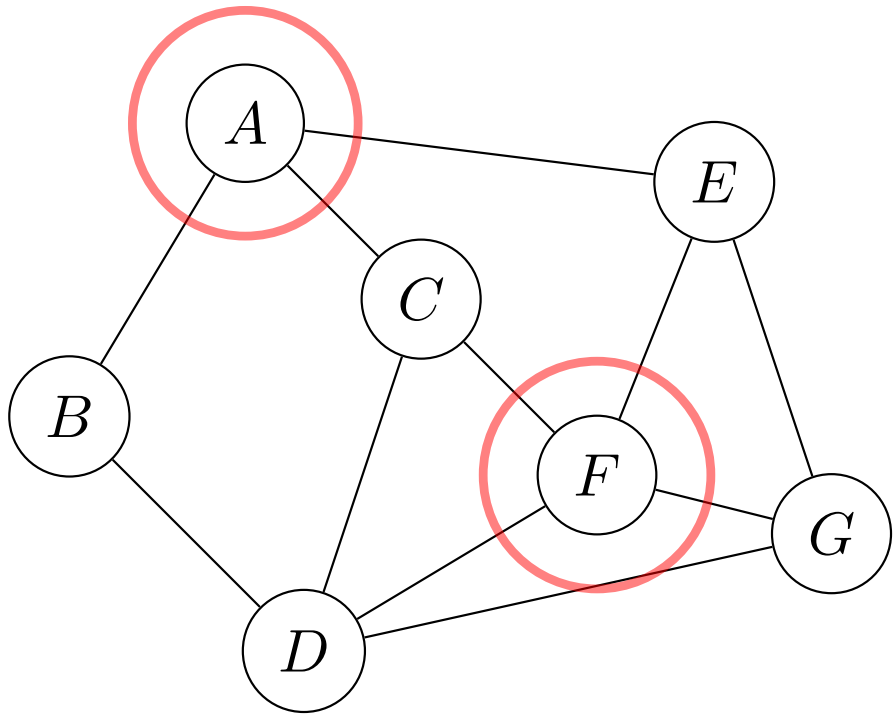
Grafer

Grannskap



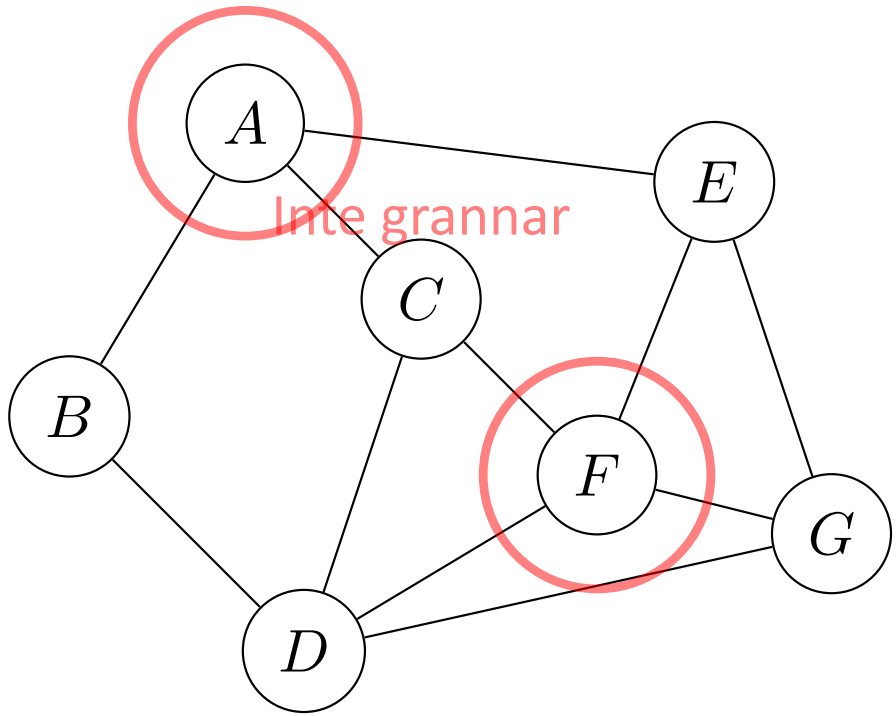
Grafer

Grannskap



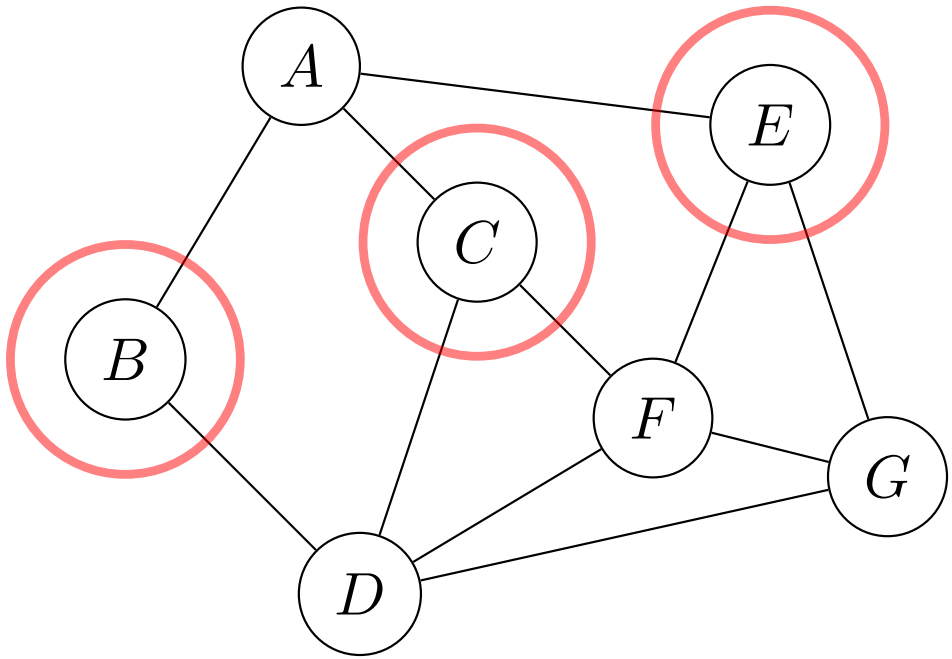
Grafer

Grannskap



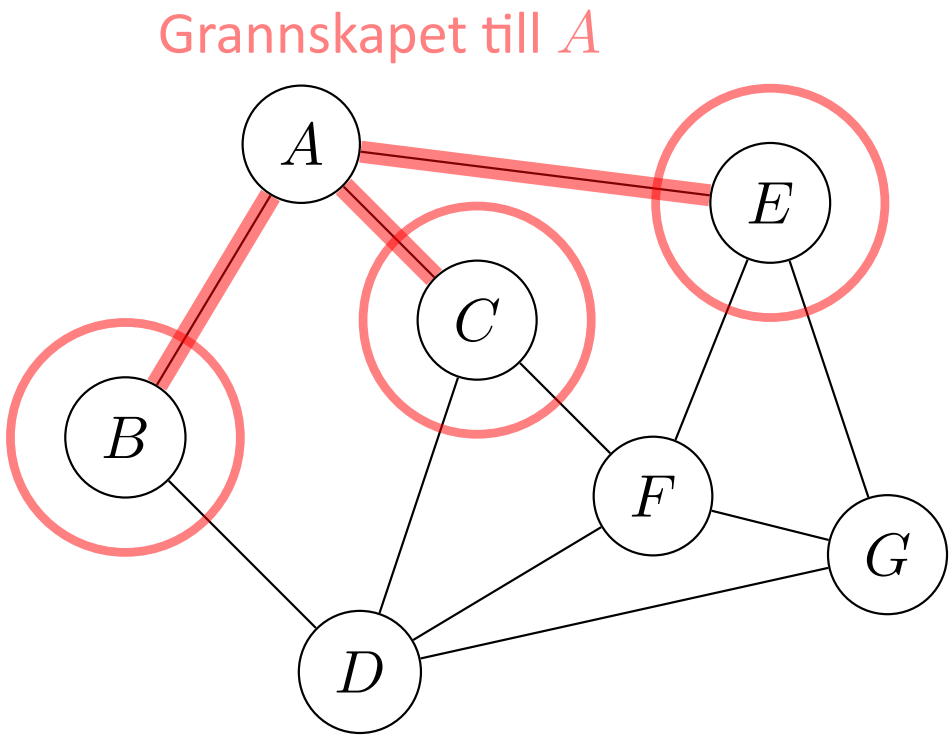
Grafer

Grannskap



Grafer

Grannskap



Grafer

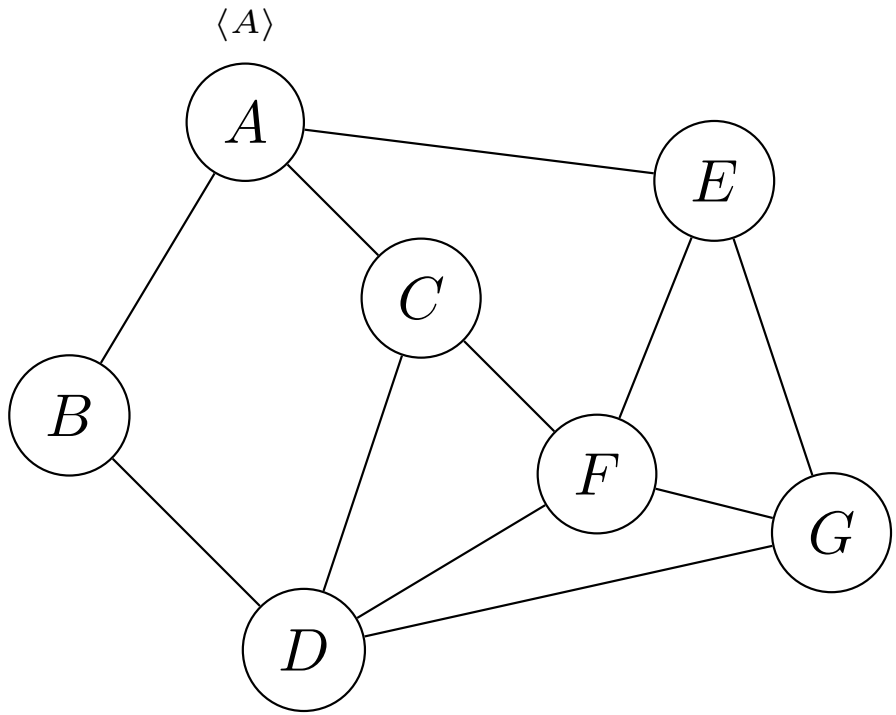
Grannskap

Defintion: Grannskap

- Två noder $u, v \in V$ sägs vara *grannar* om $(u, v) \in E$ (eller $(v, u) \in E$).
- Grannskapet av en nod u är mängden av alla noder v sådana att u och v är grannar.

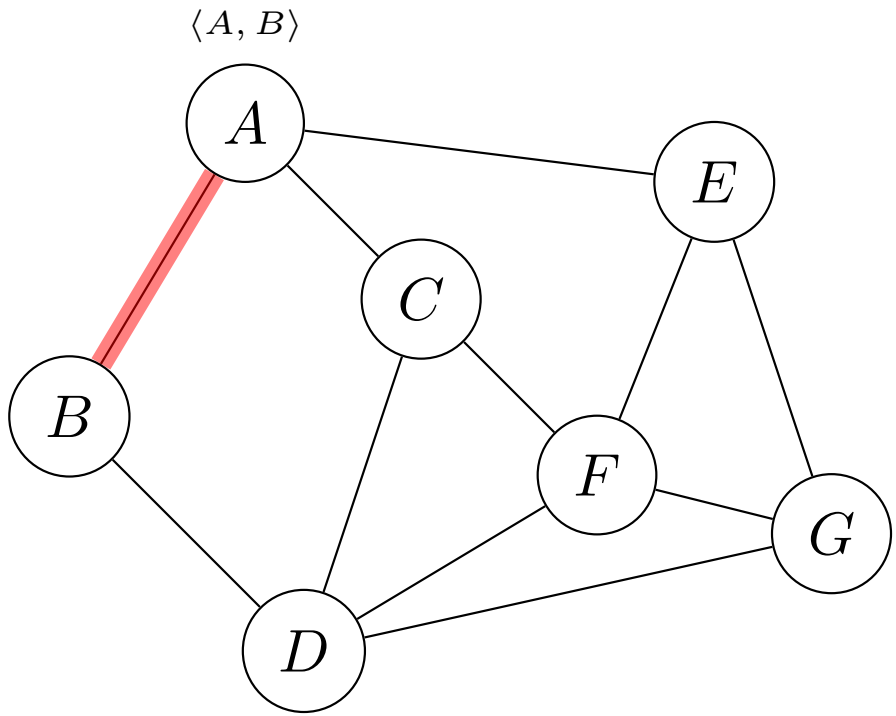
Grafer

Vandring



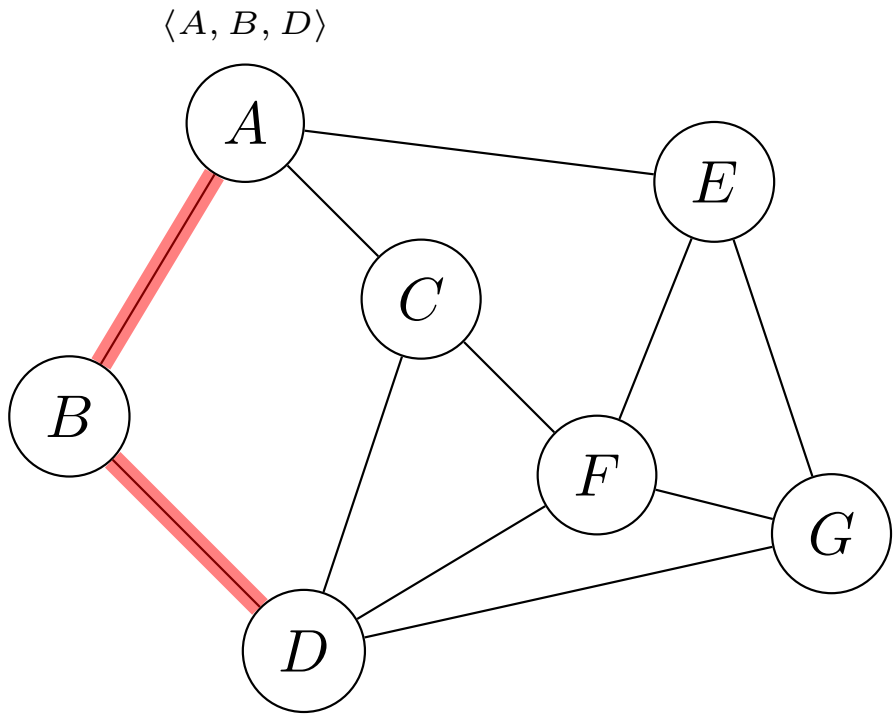
Grafer

Vandring



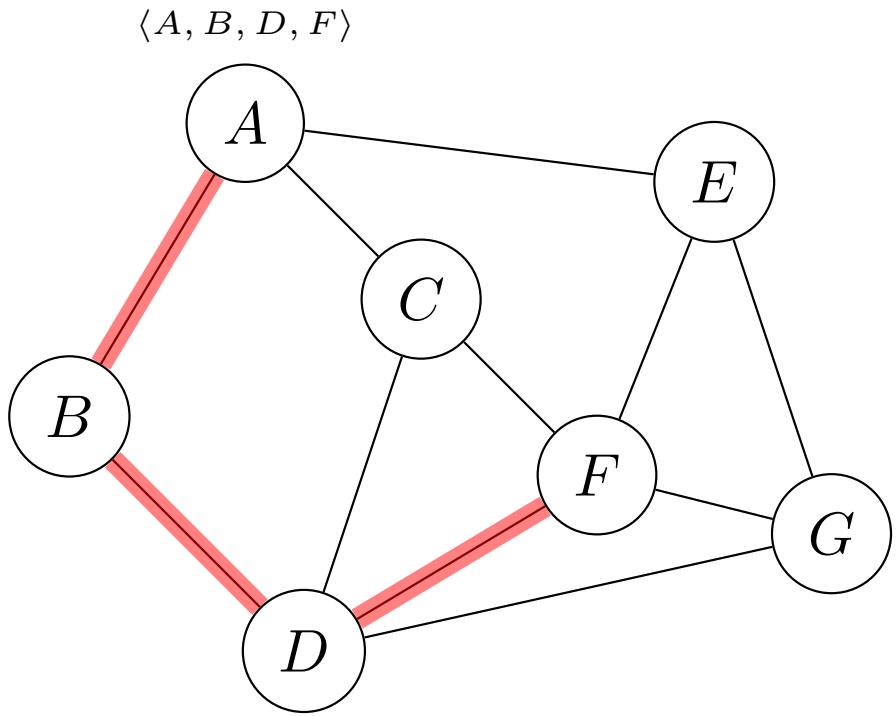
Grafer

Vandring



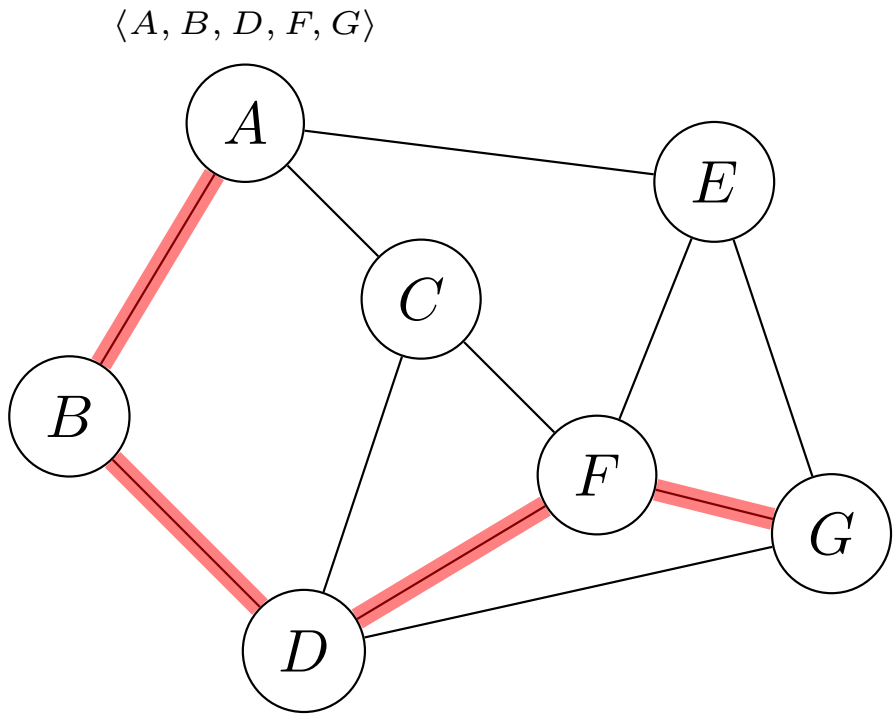
Grafer

Vandring



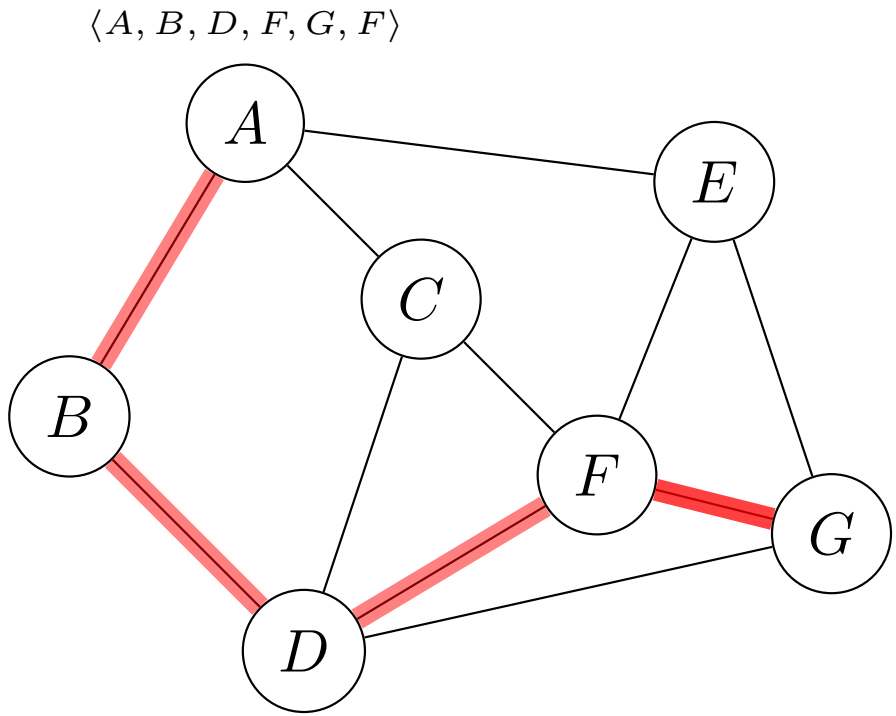
Grafer

Vandring



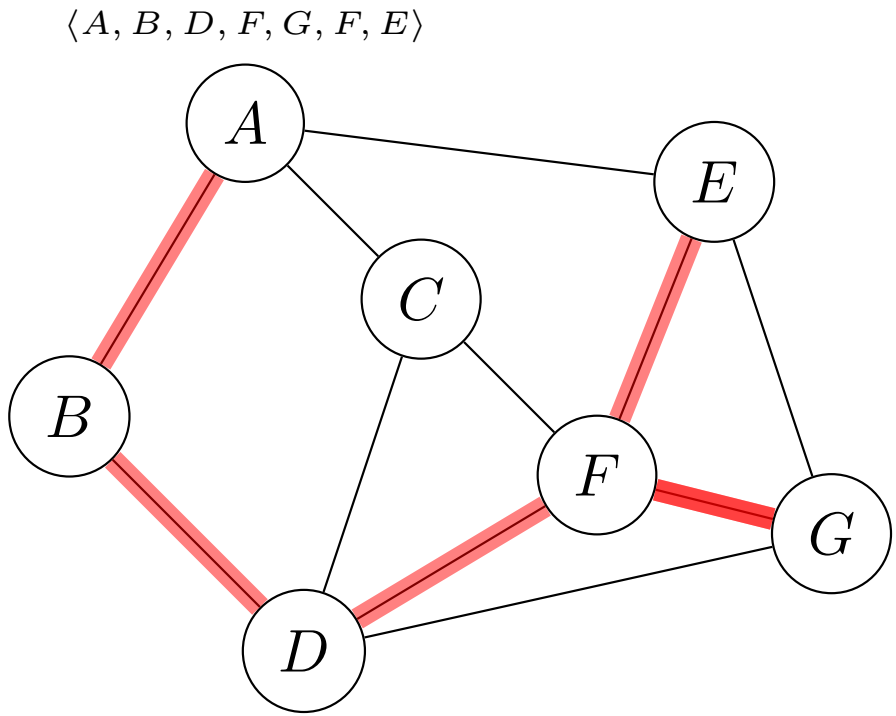
Grafer

Vandring



Grafer

Vandring



Grafer

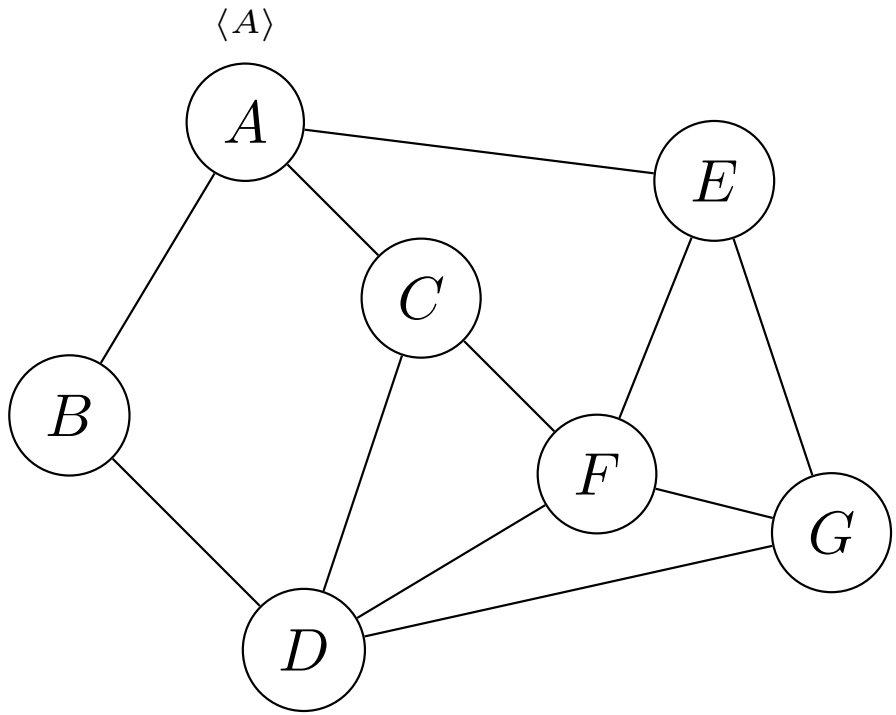
Vandring

Vandring

En *vandring* i en graf $G = (V, E)$ är en sekvens av noder $\langle v_1, v_2, \ldots, v_n \rangle$ sådana att v_i och v_{i+1} är grannar för alla $i = 1, 2, \ldots, n - 1$.

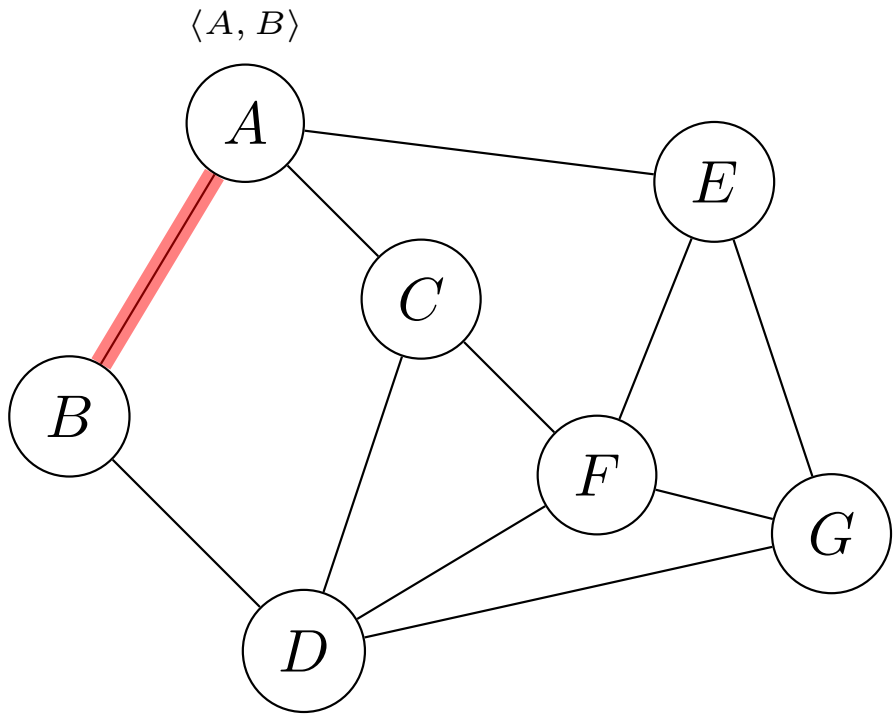
Grafer

Väg



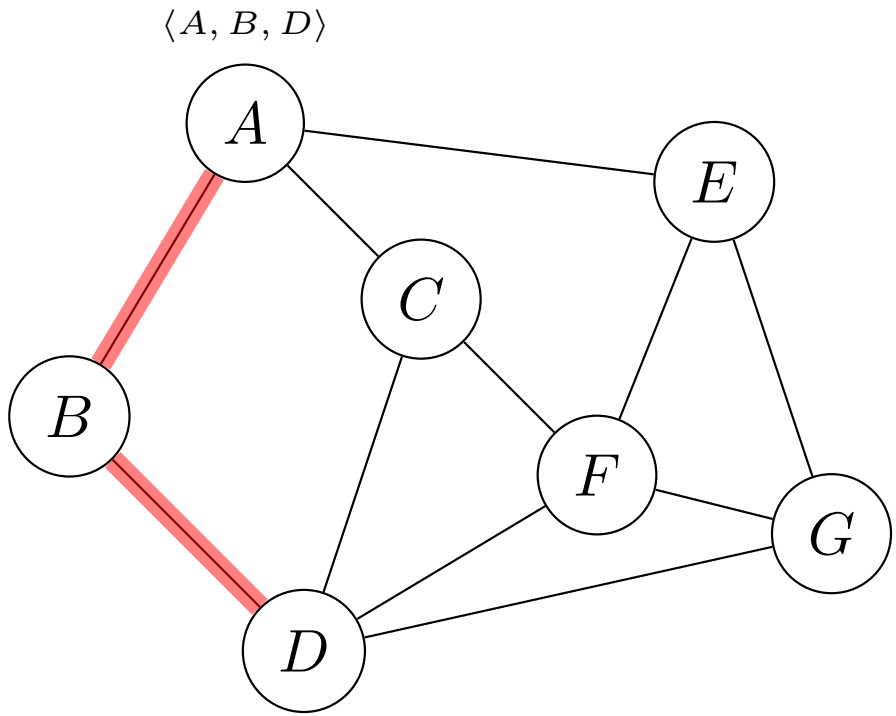
Grafer

Väg



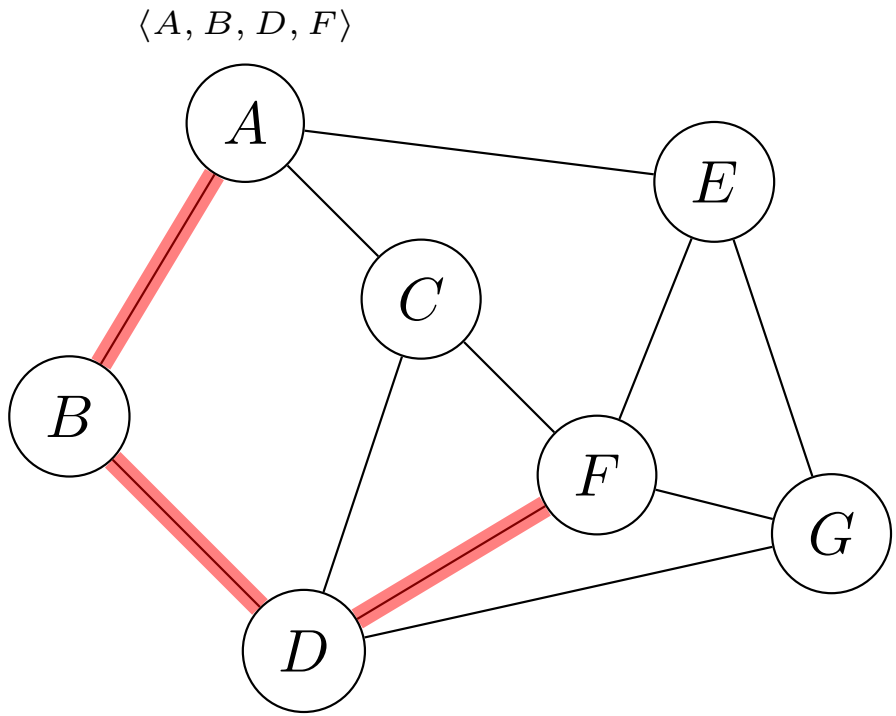
Grafer

Väg



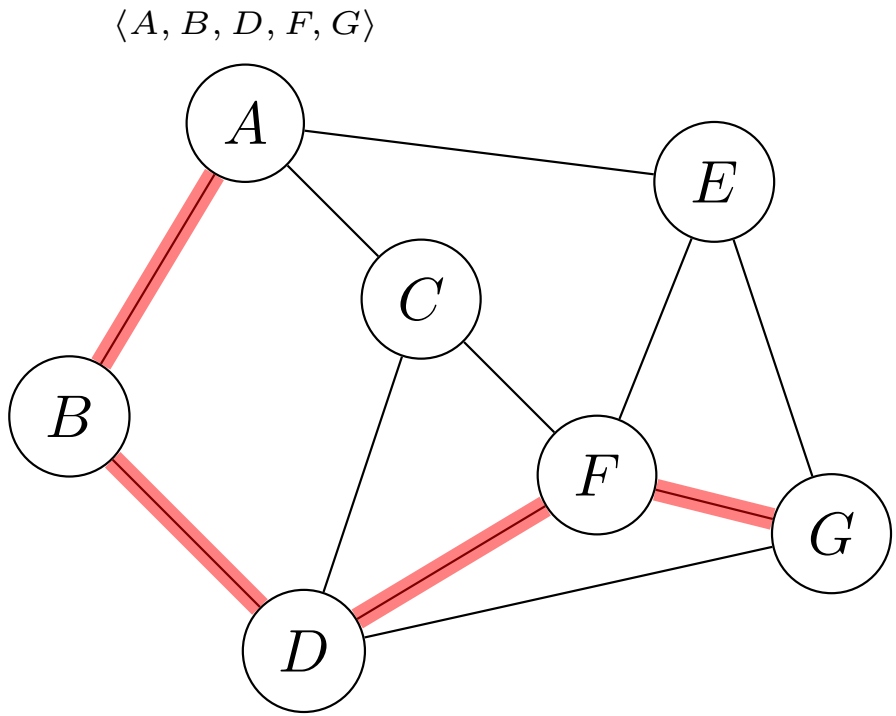
Grafer

Väg



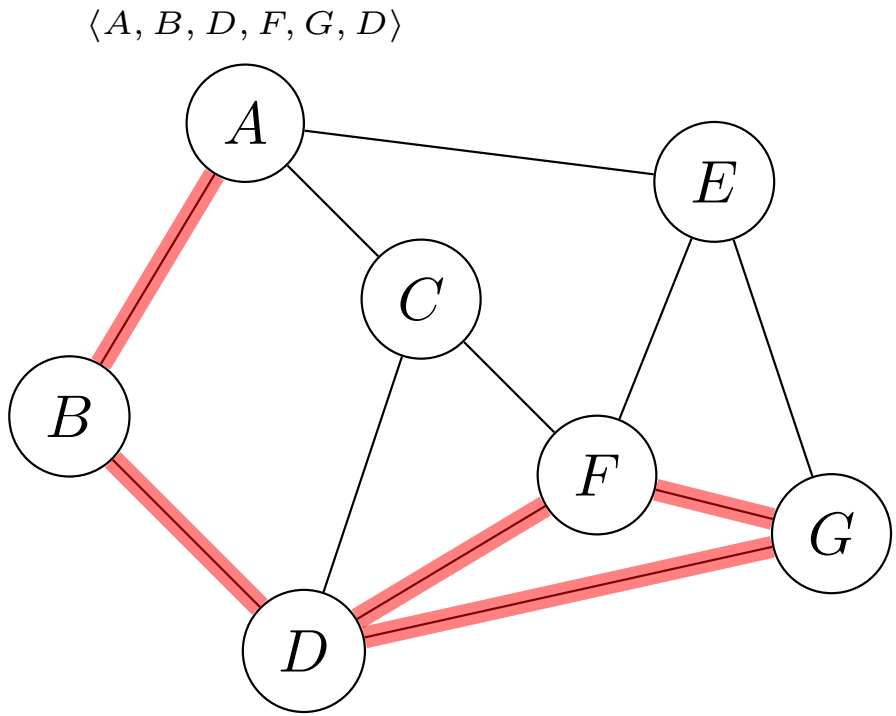
Grafer

Väg



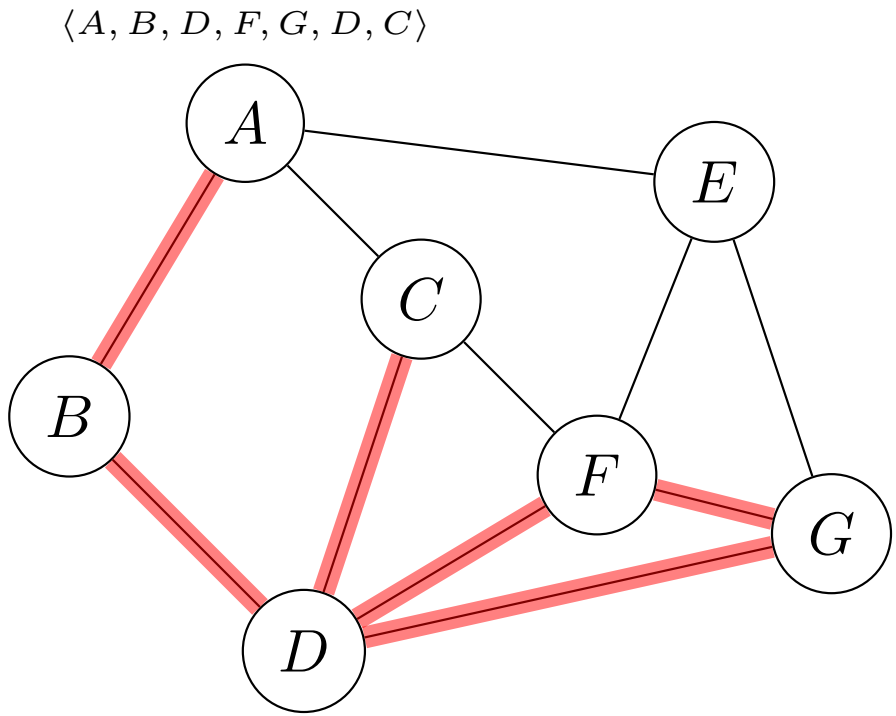
Grafer

Väg



Grafer

Väg



Grafer

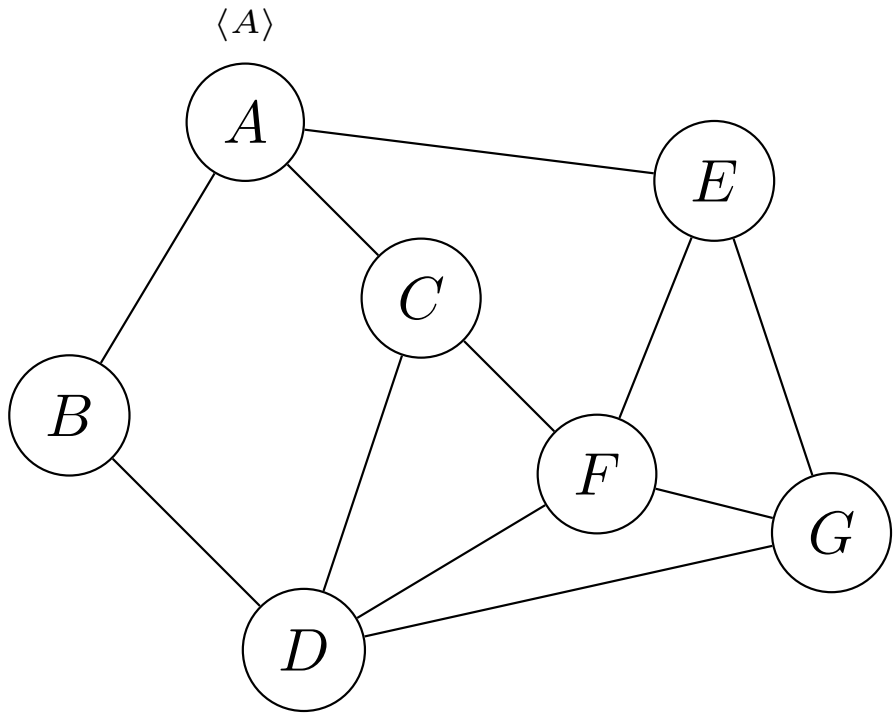
Väg

Definition: Väg

En *väg* i en graf $G = (V, E)$ är en *vandring* där bågar ej återupprepas.

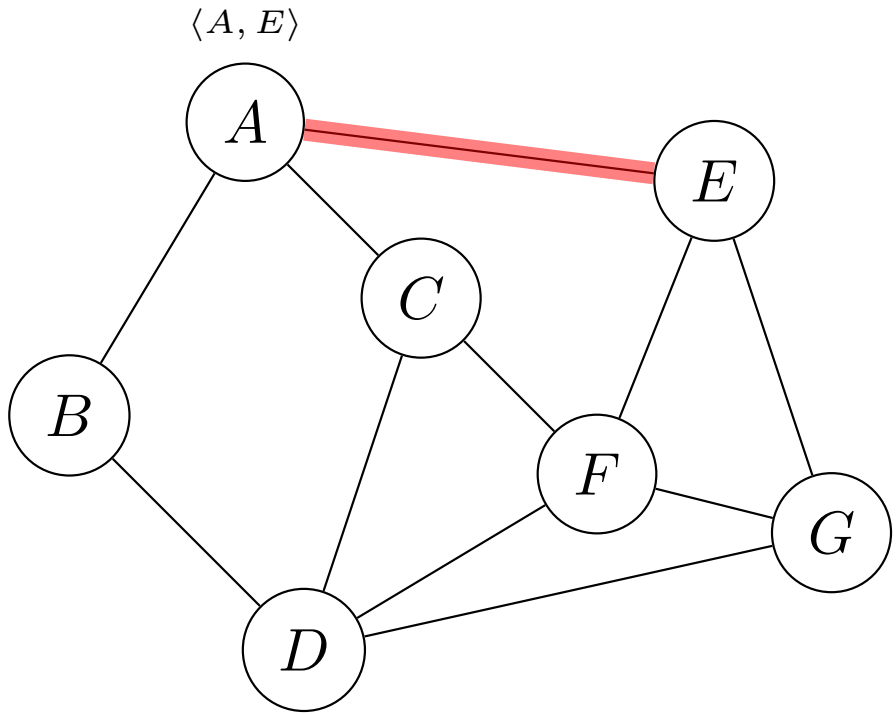
Grafer

Stig



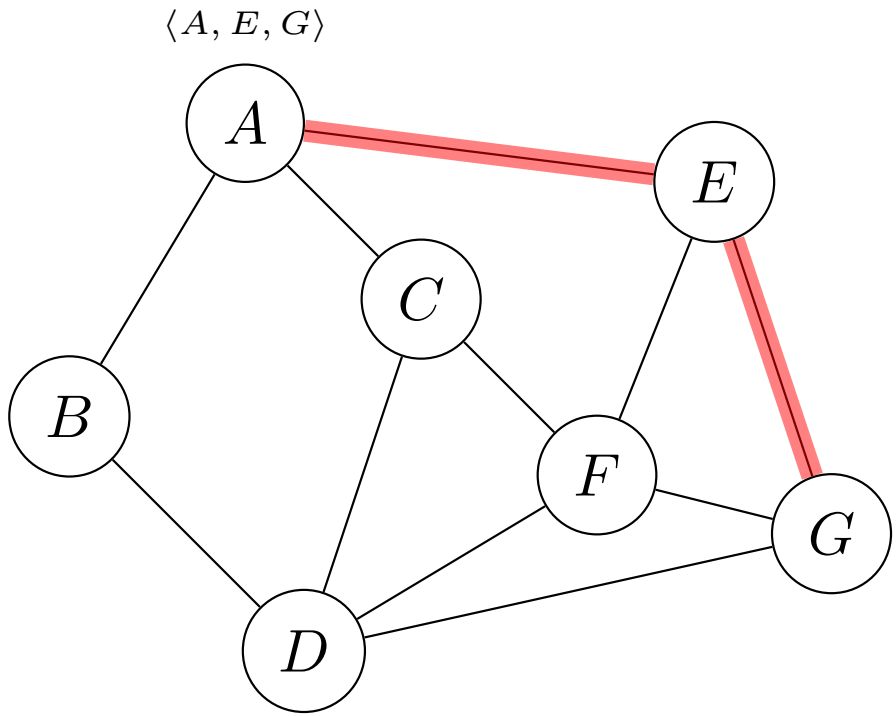
Grafer

Stig



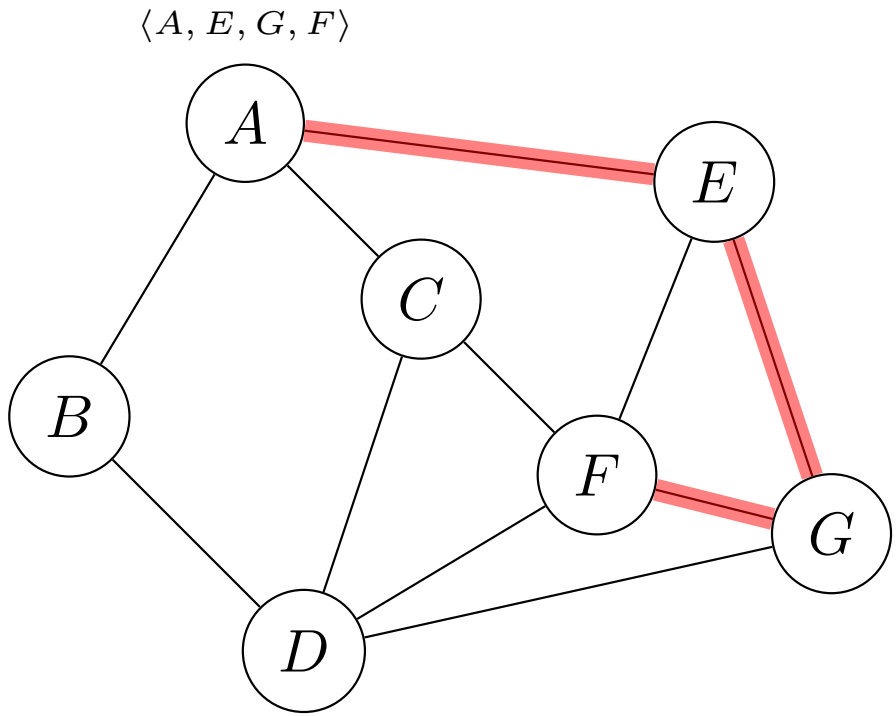
Grafer

Stig



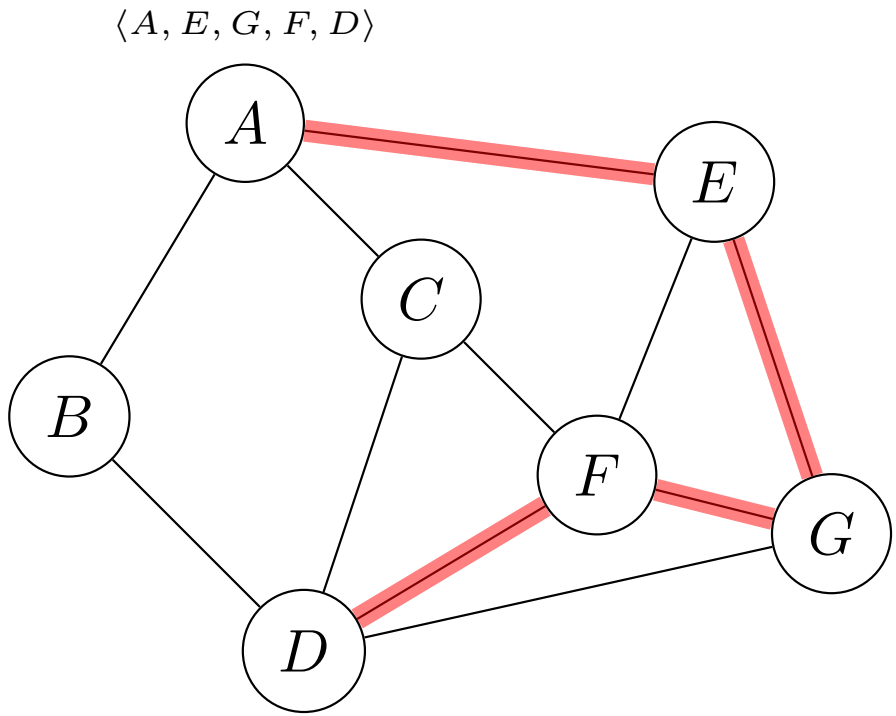
Grafer

Stig



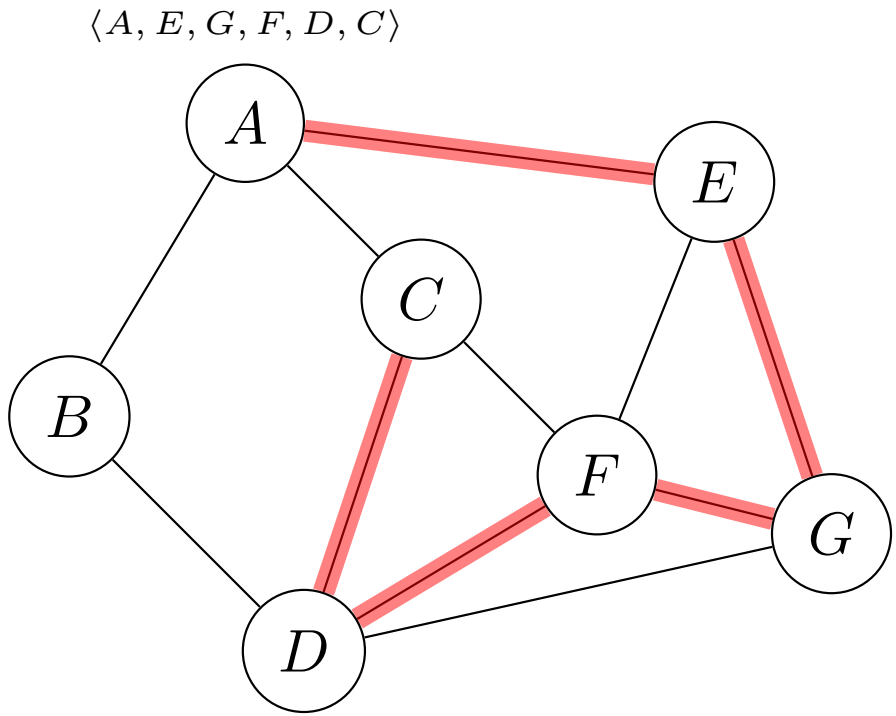
Grafer

Stig



Grafer

Stig



Grafer

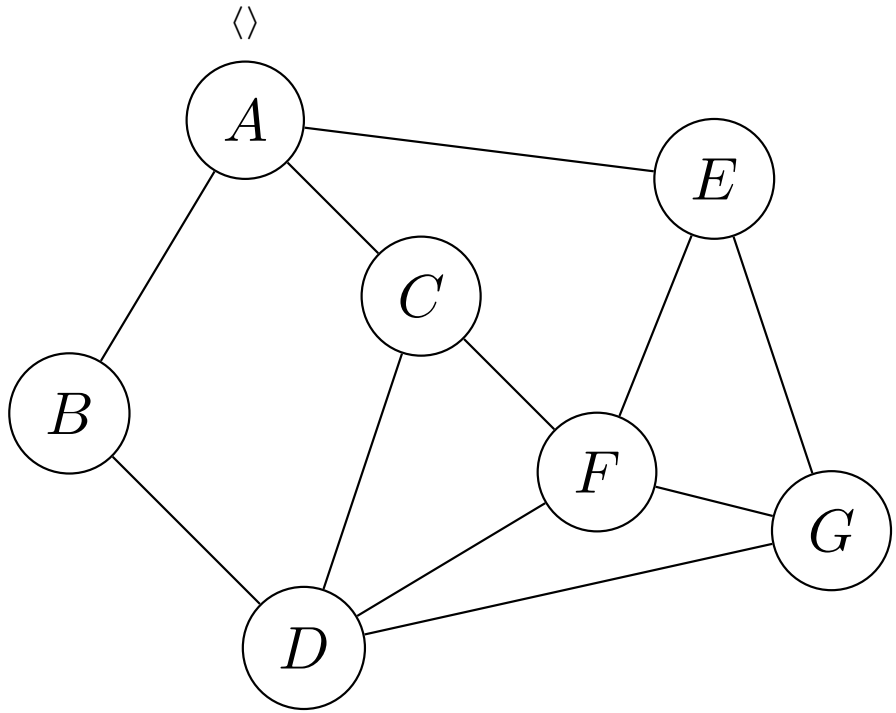
Stig

Definition: Stig

En *stig* i $G = (V, E)$ är en vandring där inga noder får återupprepas.

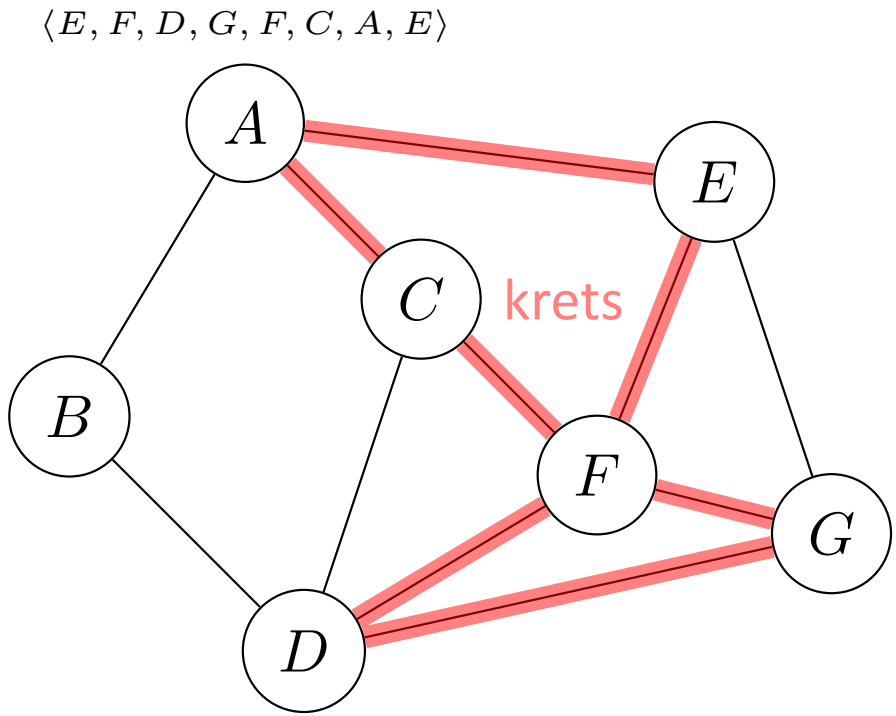
Grafer

Krets och cykel



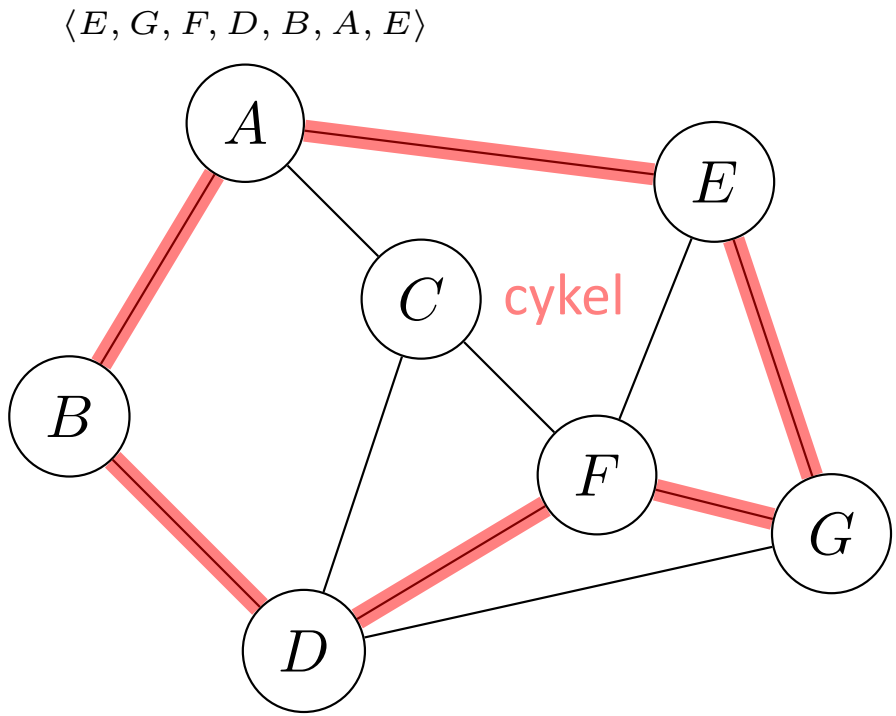
Grafer

Krets och cykel



Grafer

Krets och cykel



Grafer

Krets och cykel

Defintion: Krets och cykel

- En *krets* är en *väg*, som börjar och slutar i samma nod.
- En *cykel* är en *stig* som även inkluderar en båge från sista noden tillbaka till första noden.

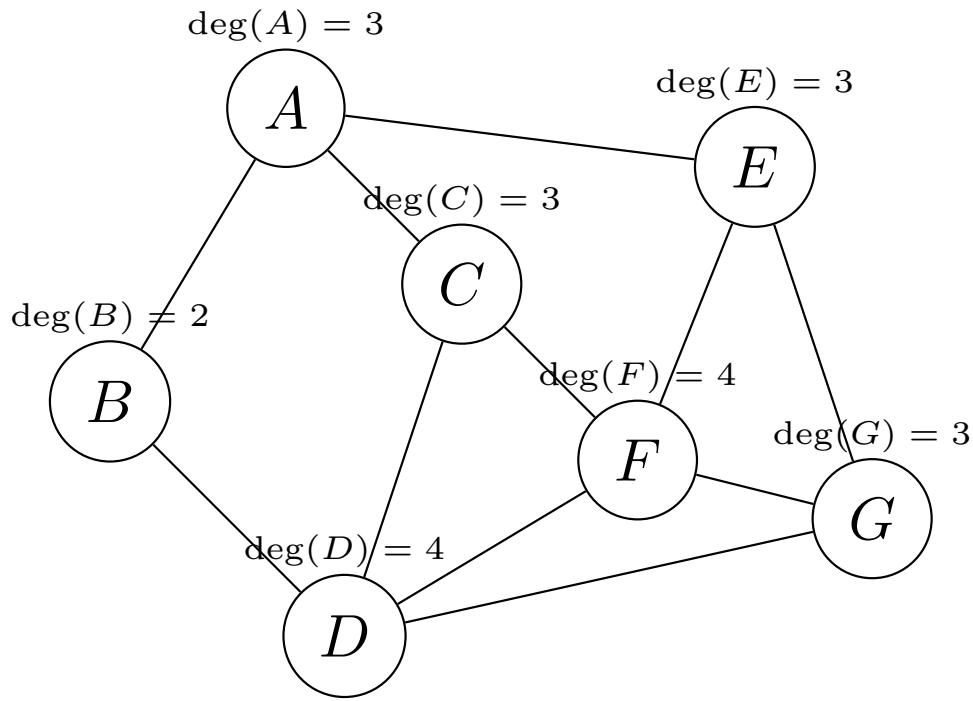
Grafer

Vandring, väg, stig, krets och cykel

- Begreppen vandring, väg och stig är nära relaterade.
- Specifikt handlar det om olika typer av sekvenser av grannar.
- Vägar är ofta det som studeras inom datavetenskap, för i många fall är det relativt enkelt att översätta algoritmer mellan att jobba med vägar och stigar.

Grafer

Nodgrad



Grafer

Nodgrad

Defintion: Nodgrad

Nodgraden $\deg(v)$ för en nod $v \in V$ i grafen $G = (V, E)$ definieras som

$$\deg(v) = |\{(v, u) \in E \mid u \in V\}|,$$

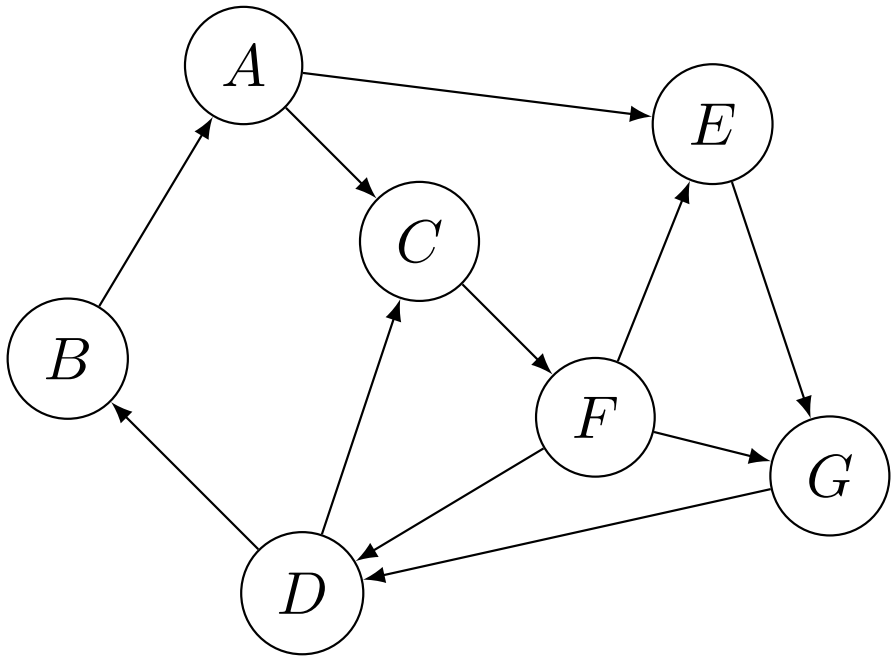
vilket motsvarar antalet bågar som har v som en ändpunkt.

1 Repetition: Grafer
2 Repetition: Riktade grafer
3 Representation av grafer
4 Grafsökning
5 Heap

LINKÖPING UNIVERSITY

Riktade grafer

En riktad graf



Riktade grafer

En riktad graf

- I föregående avsnitt diskuterades s.k. *oriktade* grafer, d.v.s. grafer där bågarna är oordnade par.
- Dessa grafer används oftast för att representera associativa eller kommutativa egenskaper mellan objekten.
- Men i många fall behöver vi representera ensidiga relationer. Detta görs med s.k. *riktade* grafer.
- I en riktad graf har bågarna en *riktning* (d.v.s. de är pilar).

Riktade grafer

Riktad graf

Definition: Riktad graf

En graf $G = (V, E)$ är *riktad* om paren $(u, v) \in E$ är ordnade, d.v.s. att $(u, v) \neq (v, u)$ givet att $u \neq v$.

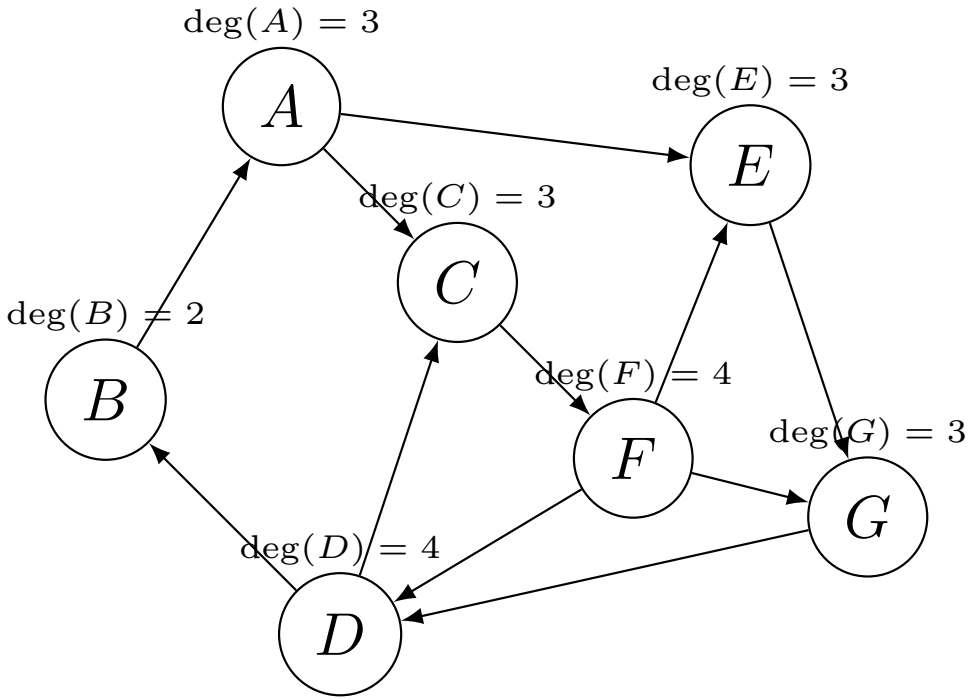
Riktade grafer

Terminologi

- I en riktad graf har bågarna två ändpunkter, en *startnod* och en *slutnod*.
- I riktade grafer har varje nod v både inkommande och utgående bågar, d.v.s. bågar som har v som slut– respektive startnod.
- Varje riktad graf har en relaterad graf som kallas grafens *transponat*. Detta är den graf där alla bågars riktning vänds.

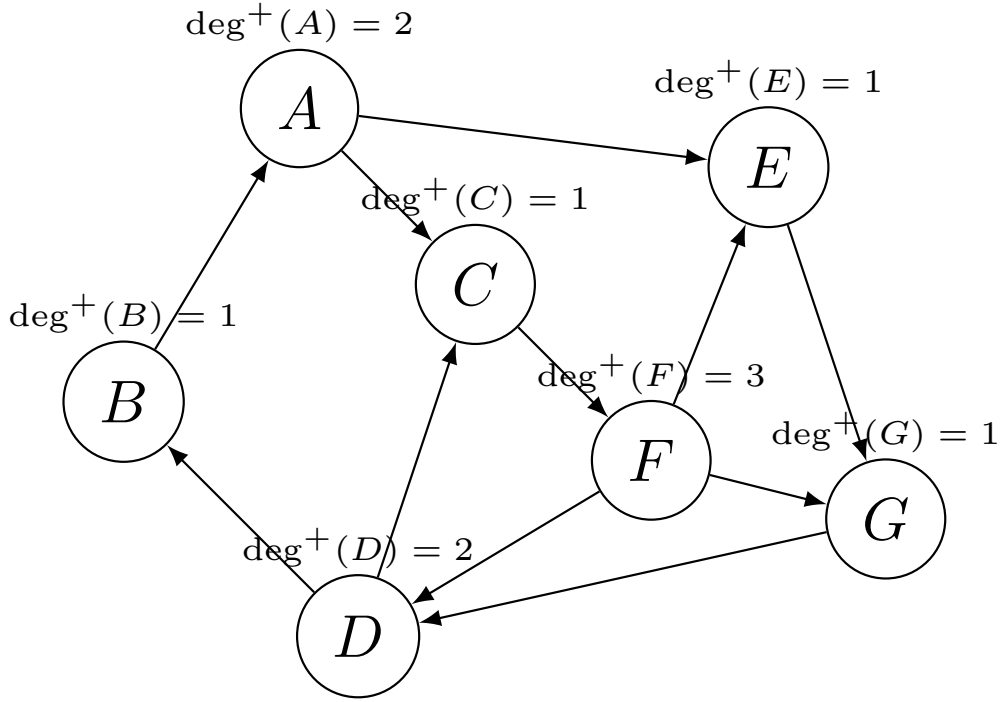
Riktade grafer

Nodgrad



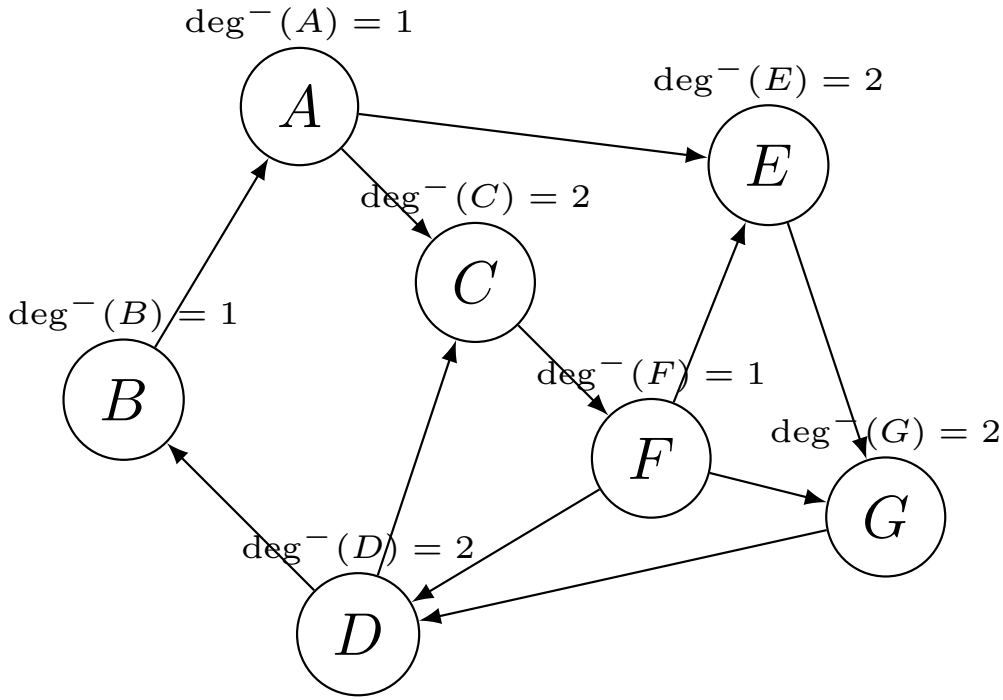
Riktade grafer

Nodgrad



Riktade grafer

Nodgrad



Riktade grafer

Nodgrad

- I riktade grafer blir begreppet nodgrad mindre tydligt.
- Detta eftersom att vi nu har två typer av bågar, inkommande och utgående, som har olika betydelse.
- P.g.a. detta definierar vi för varje nod v, inkommande nodgrad som antalet bågar som har v som slutnod, och utgående nodgrad som antalet bågar som har v som startnod.

Riktade grafer

Nodgrad

Definition: Nodgrad för riktade grafer

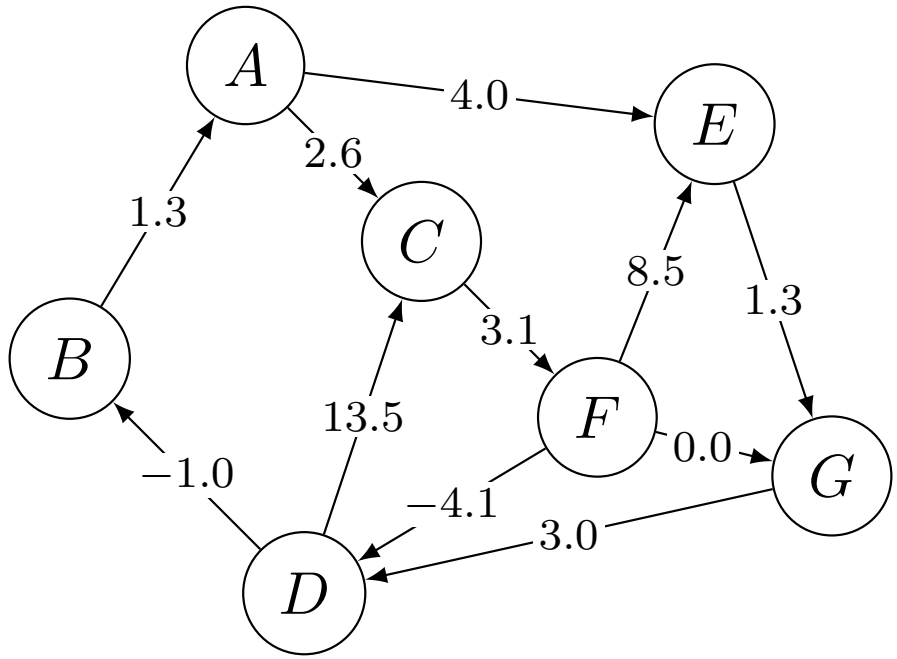
För en riktad graf $G = (V, E)$ definierar vi för varje $v \in V$:

- $\deg^+(v) = |\{s \mid (v, s) \in E\}|$ (utgående bågar)
- $\deg^-(v) = |\{u \mid (u, v) \in E\}|$ (inkommande bågar)

Notera: $\deg(v) = \deg^-(v) + \deg^+(v)$

Riktade grafer

Viktade grafer



Riktade grafer

Viktade grafer

- En viktad graf är (oftast) en riktad graf.
- Viktade grafer associerar en vikt eller kostnad för vardera båge. Vad denna vikt eller kostnad representerar beror på tillämpningsområde.
- Dessa typer av grafer är oerhört vanliga inom optimeringslära, nätverksteori och andra praktiska tillämpningsområden.

Riktade grafer

Viktade grafer

Definition: Viktade grafer

En *viktad* graf är en graf $G_w = (V, E)$ med en tillhörande avbildning $w : E \to \mathbb{R}$ som motsvarar en båges *vikt*.

1 Repetition: Grafer
2 Repetition: Riktade grafer
3 Representation av grafer
4 Grafsökning
5 Heap

LINKÖPING UNIVERSITY

Representation av grafer

Hur lagrar vi en graf?

- Som tidigare nämnt så används grafer ofta som en icke-linjär datastruktur för relationer/associationer mellan olika objekt.
- Detta innebär att vi måste kunna lagra denna data på något sätt samtidigt som vi bibehåller grafstrukturen.
- I detta kapitel presenteras ett antal olika sätt att lagra en graf m.h.a. tidigare använda datastrukturer.

Representation av grafer

Mål med representationen

- lagra noder
- lagra bågar
- lagra eventuella vikter
- insättning (noder och bågar)
- borttagning (noder och bågar)
- sökning (?)

Representation av grafer

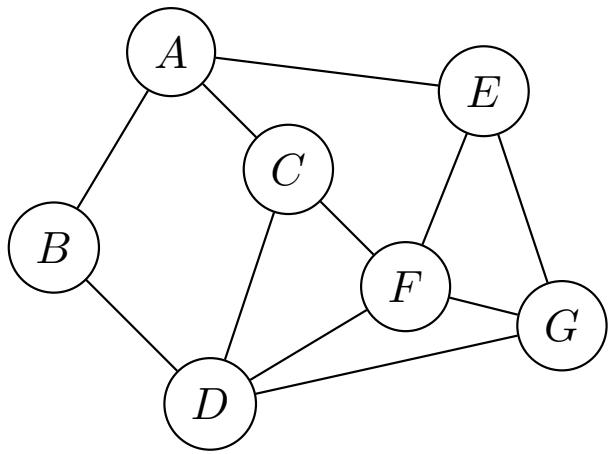
Idé #1

Ide #1

Lagra alla noder i en array och alla bågar som par i en annan array.

Representation av grafer

Linjär representation



noder:

A	B	C	D	E	F	G

bågar:

A, B	A, C	A, E	B, D	C, D	C, F	D, F	D, G	E, F	E, G	F, G

Representation av grafer

Analys av linjär representation

- Insättning av nod: $\mathcal{O}(|V|)$
- Insättning av båge: $\mathcal{O}(|E|)$
- Borttagning av nod: $\mathcal{O}(|V|)$
- Borttagning av båge: $\mathcal{O}(|E|)$
- Hitta om nod existerar: $\mathcal{O}(|V|)$
- Hitta om båge existerar: $\mathcal{O}(|E|)$
- Om allting hålls sorterat blir det istället logaritmiska tidskomplexiteter...
- Vi kan dock göra ännu bättre!

Representation av grafer

Observation

Givet $|V| = n$:

- Om oriktad graf $\Rightarrow |E| \leq \frac{n(n-1)}{2}$
- Om riktad graf $\Rightarrow |E| \leq n(n-1)$

Representation av grafer

Observation

- E definieras som par av noder.
- Det finns $n(n-1)$ unika par av noder som kan bildas.
- I.o.m. att bågar definieras som par av noder så kommer en riktad båge motsvara exakt ett av dessa unika par.
- Så antalet riktade bågar är som mest $n(n-1)$.
- I oriktade grafer är $(u, v) = (v, u)$, d.v.s. att ordningen på noderna i bågen spelar ingen roll. Då motsvarar vardera båge exakt två möjliga par, så vi får maximalt $\frac{n(n-1)}{2}$ unika oriktade bågar.

Representation av grafer

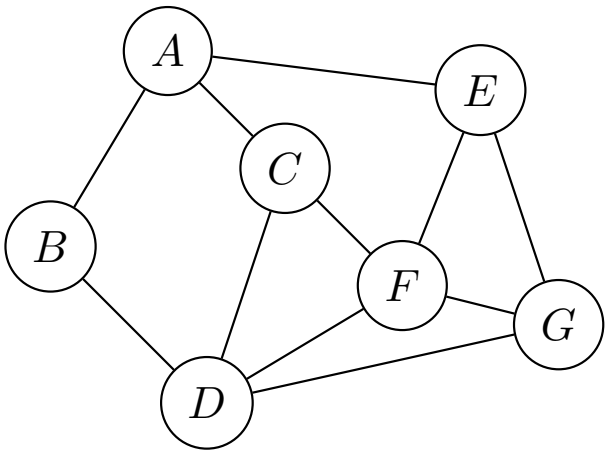
Idé #2

Idé #2

Lagra bågarna som en matris!

Representation av grafer

Grannmatris

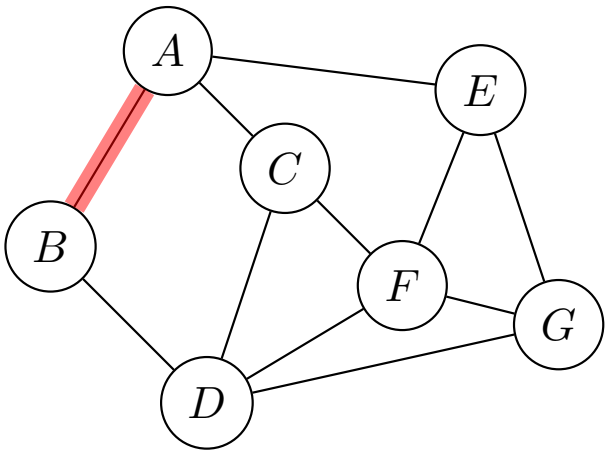


	0	1	2	3	4	5	6
noder:	A	B	C	D	E	F	G

	0	1	2	3	4	5	6
0	■						
1		■					
2			■				
3				■			
4					■		
5						■	
6							■

Representation av grafer

Grannmatris

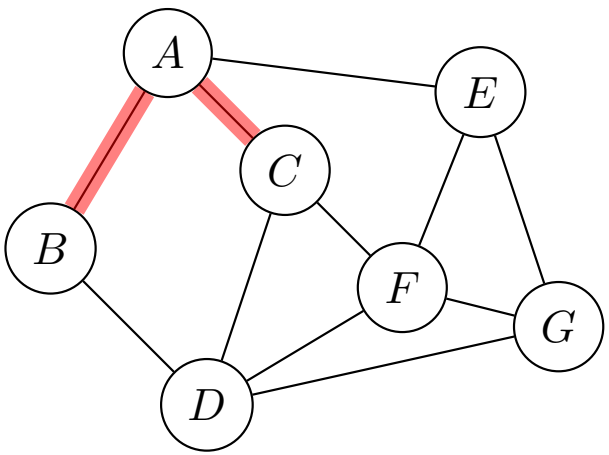


noder:	0	1	2	3	4	5	6
	A	B	C	D	E	F	G

	0	1	2	3	4	5	6
0		1					
1	1						
2							
3							
4							
5							
6							

Representation av grafer

Grannmatris

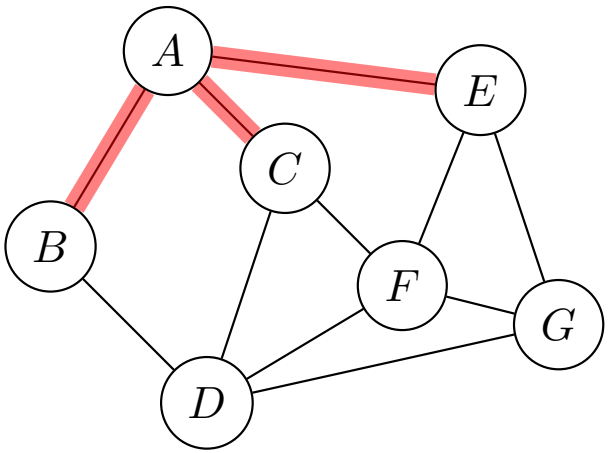


noder:	0	1	2	3	4	5	6
	A	B	C	D	E	F	G

	0	1	2	3	4	5	6
0		1	1				
1	1						
2	1						
3							
4							
5							
6							

Representation av grafer

Grannmatris



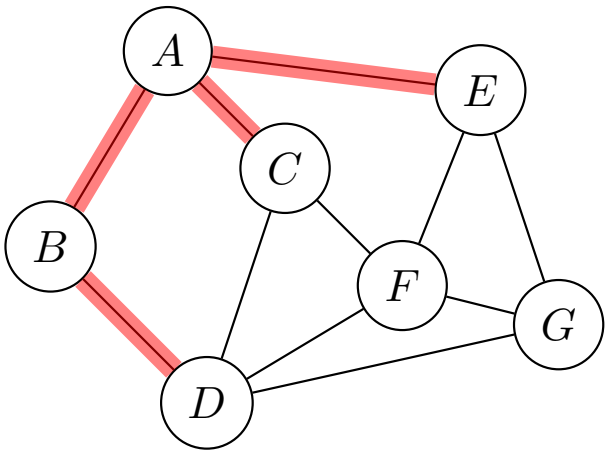
noder:

0	1	2	3	4	5	6
A	B	C	D	E	F	G

	0	1	2	3	4	5	6
0		1	1		1		
1	1						
2	1						
3							
4	1						
5							
6							

Representation av grafer

Grannmatris



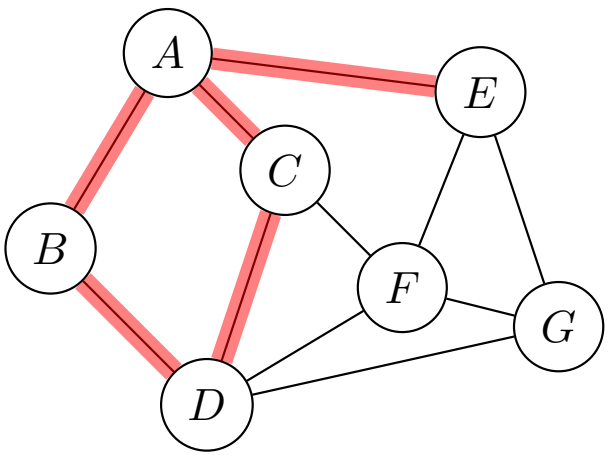
noder:

0	1	2	3	4	5	6
A	B	C	D	E	F	G

	0	1	2	3	4	5	6
0		1	1		1		
1	1			1			
2	1						
3		1					
4	1						
5							
6							

Representation av grafer

Grannmatris



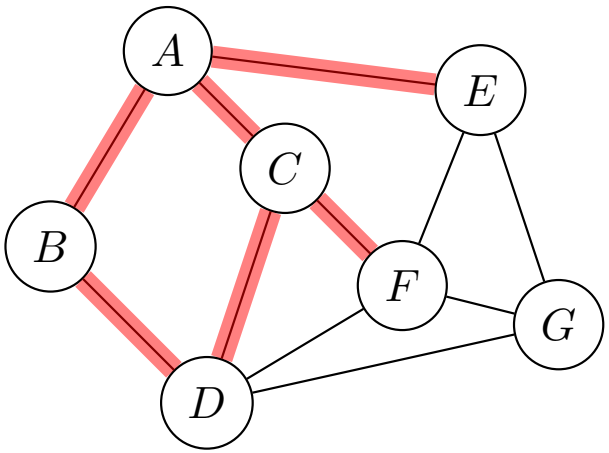
noder:

0	1	2	3	4	5	6
A	B	C	D	E	F	G

	0	1	2	3	4	5	6
0		1	1		1		
1	1			1			
2	1			1			
3		1	1				
4	1						
5							
6							

Representation av grafer

Grannmatris



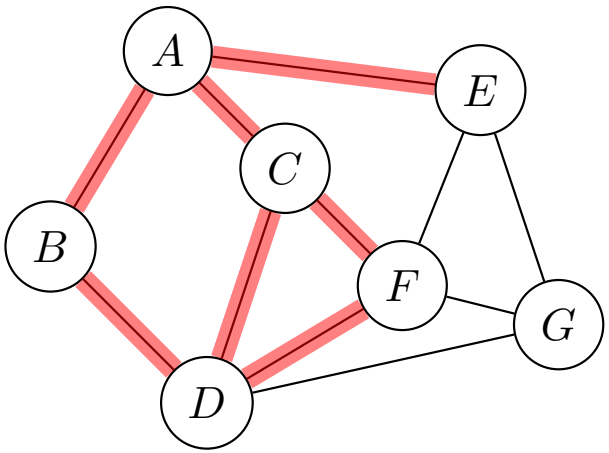
noder:

0	1	2	3	4	5	6
A	B	C	D	E	F	G

	0	1	2	3	4	5	6
0		1	1		1		
1	1			1			
2	1			1		1	
3		1	1				
4	1						
5			1				
6							

Representation av grafer

Grannmatris

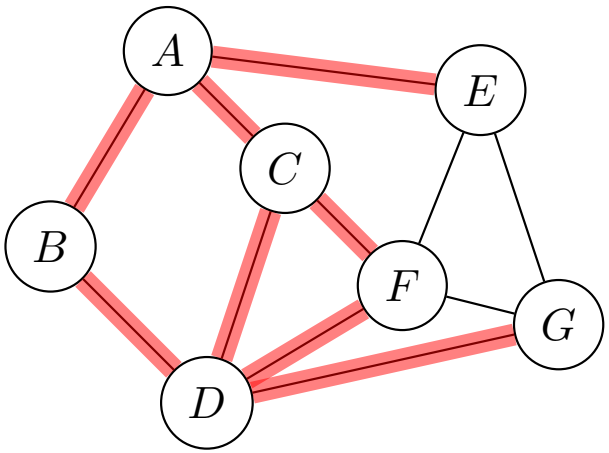


noder:	0	1	2	3	4	5	6
	A	B	C	D	E	F	G

	0	1	2	3	4	5	6
0		1	1		1		
1	1			1			
2	1			1		1	
3		1	1			1	
4	1						
5			1	1			
6							

Representation av grafer

Grannmatris

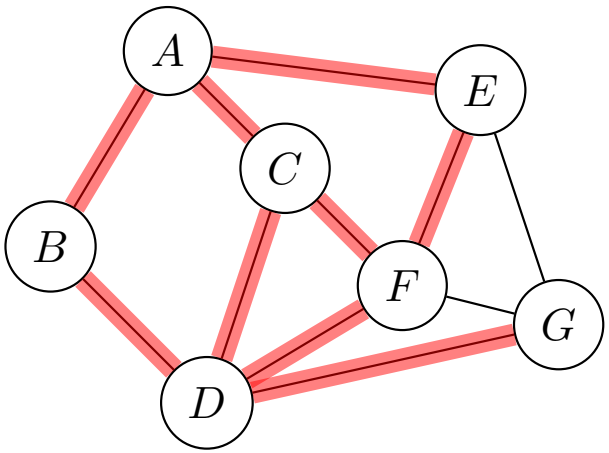


noder:	0	1	2	3	4	5	6
	A	B	C	D	E	F	G

	0	1	2	3	4	5	6
0		1	1		1		
1	1			1			
2	1			1		1	
3		1	1			1	1
4	1						
5			1	1			
6				1			

Representation av grafer

Grannmatris

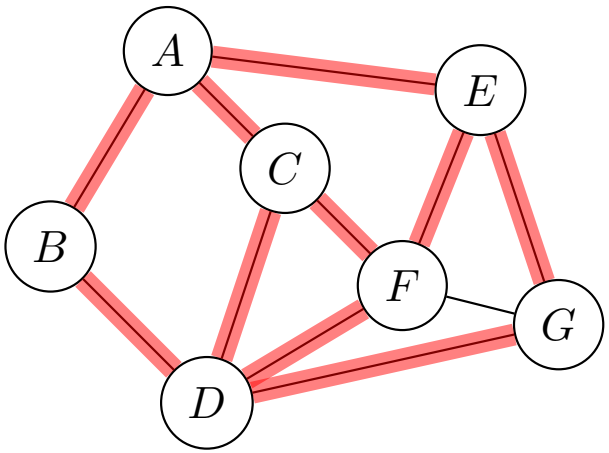


noder:	0	1	2	3	4	5	6
	A	B	C	D	E	F	G

	0	1	2	3	4	5	6
0		1	1		1		
1	1			1			
2	1			1		1	
3		1	1			1	1
4	1					1	
5			1	1	1		
6				1			

Representation av grafer

Grannmatris

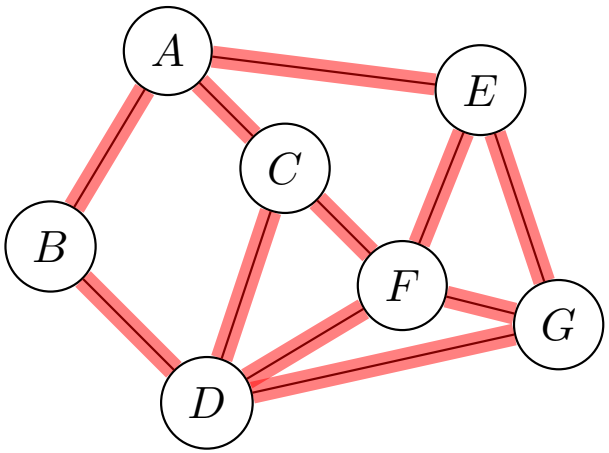


noder:	0	1	2	3	4	5	6
	A	B	C	D	E	F	G

	0	1	2	3	4	5	6
0		1	1		1		
1	1			1			
2	1			1		1	
3		1	1			1	1
4	1					1	1
5			1	1	1		
6				1	1		

Representation av grafer

Grannmatris

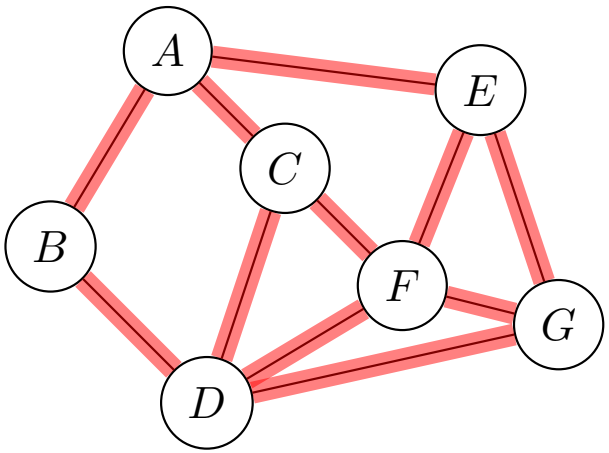


noder:	0	1	2	3	4	5	6
	A	B	C	D	E	F	G

	0	1	2	3	4	5	6
0		1	1		1		
1	1			1			
2	1			1		1	
3		1	1			1	1
4	1					1	1
5			1	1	1		1
6				1	1	1	

Representation av grafer

Grannmatris

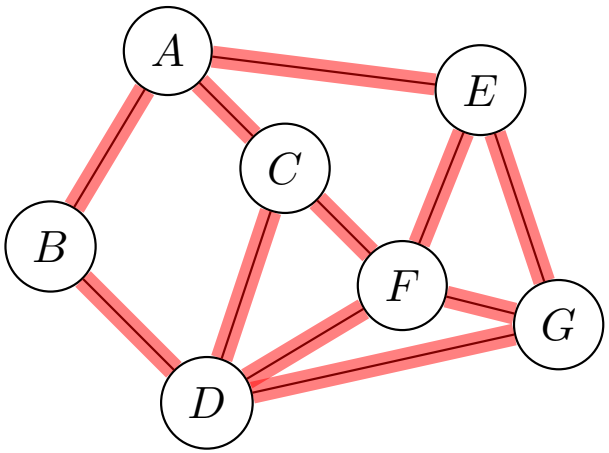


noder:	0	1	2	3	4	5	6
	A	B	C	D	E	F	G

	0	1	2	3	4	5	6
0	0	1	1	0	1	0	0
1	1	0	0	1	0	0	0
2	1	0	0	1	0	1	0
3	0	1	1	0	0	1	1
4	1	0	0	0	0	1	1
5	0	0	1	1	1	0	1
6	0	0	0	1	1	1	0

Representation av grafer

Grannmatris



noder:	0	1	2	3	4	5	6
	A	B	C	D	E	F	G

	0	1	2	3	4	5	6
0	0	1	1	0	1	0	0
1	1	0	0		0	0	0
2	1	0	0	1		1	0
3	0	1	1	0	0		1
4	1	0	0	0	0	1	
5	0	0	1	1	1	0	1
6	0	0	0	1	1	1	0

räcker för oriktade grafer

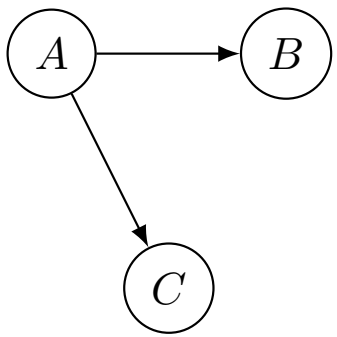
Representation av grafer

Analys av grannmatris

- Insättning av bågar: $\mathcal{O}(1)$
- Borttagning av bågar: $\mathcal{O}(1)$
- Hitta båge: $\mathcal{O}(1)$
- Insättning av nod: $\mathcal{O}\left(|V|^2\right)$
- Borttagning av nod: $\mathcal{O}\left(|V|^2\right)$
- Hitta nod (om noder är sorterade): $\mathcal{O}(\log|V|)$
- Hitta nod (om noder är osorterade): $\mathcal{O}(|V|)$
- Minnesanvändning: $\mathcal{O}\left(|V|+|V|^2\right)$

Representation av grafer

Varför $\mathcal{O}\left(|V|^2\right)$?

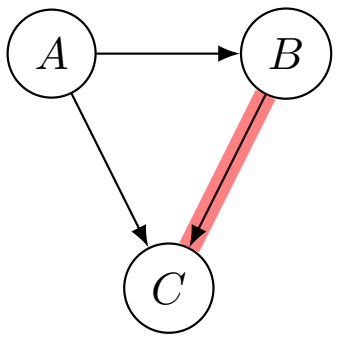


	0	1	2
noder:	A	B	C

	0	1	2
0	0	1	1
1	0	0	0
2	0	0	0

Representation av grafer

Varför $\mathcal{O}\left(|V|^2\right)$?

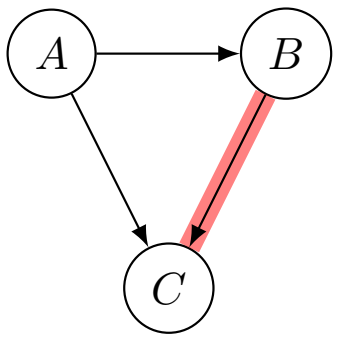


	0	1	2
noder:	A	B	C

	0	1	2
0	0	1	1
1	0	0	0
2	0	0	0

Representation av grafer

Varför $\mathcal{O}\left(|V|^2\right)$?

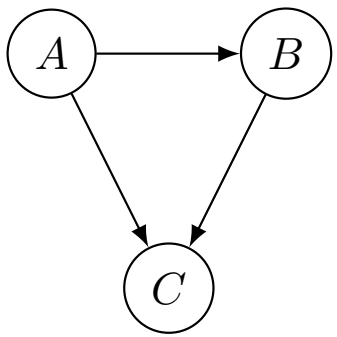


	0	1	2
noder:	A	B	C

	0	1	2
0	0	1	1
1	0	0	1
2	0	0	0

Representation av grafer

Varför $\mathcal{O}\left(|V|^2\right)$?

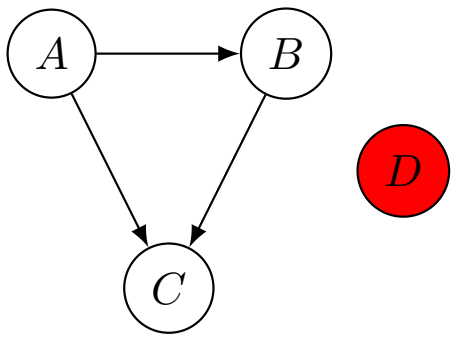


	0	1	2
noder:	A	B	C

	0	1	2
0	0	1	1
1	0	0	1
2	0	0	0

Representation av grafer

Varför $\mathcal{O}\left(|V|^2\right)$?



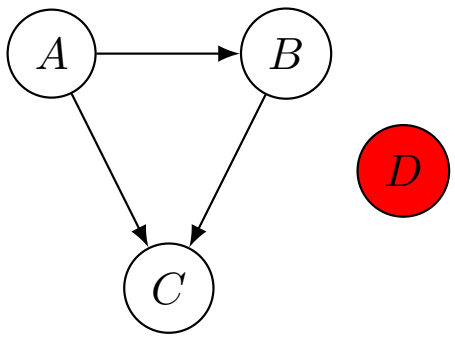
noder:

0	1	2
A	B	C

	0	1	2
0	0	1	1
1	0	0	1
2	0	0	0

Representation av grafer

Varför $\mathcal{O}\left(|V|^2\right)$?

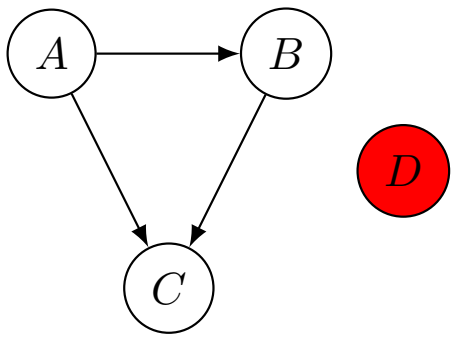


	0	1	2	3
noder:	A	B	C	D

	0	1	2
0	0	1	1
1	0	0	1
2	0	0	0

Representation av grafer

Varför $\mathcal{O}\left(|V|^2\right)$?



noder:

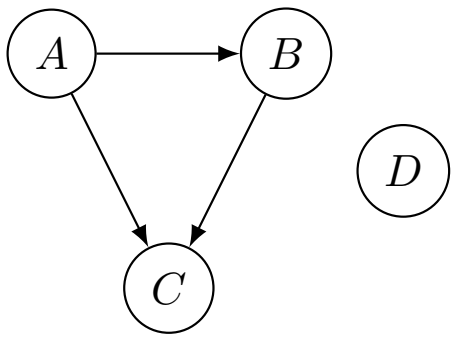
0	1	2	3
A	B	C	D

	0	1	2
0	0	1	1
1	0	0	1
2	0	0	0

onödigt!

Representation av grafer

Varför $\mathcal{O}\left(|V|^2\right)$?

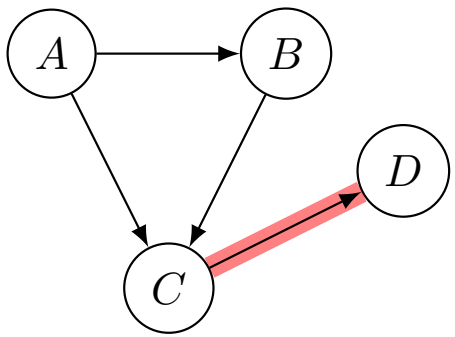


	0	1	2	3
noder:	A	B	C	D

	0	1	2	3
0	0	1	1	0
1	0	0	1	0
2	0	0	0	0
3	0	0	0	0

Representation av grafer

Varför $\mathcal{O}\left(|V|^2\right)$?

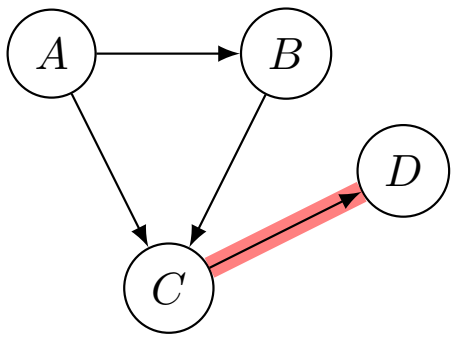


noder:	0	1	2	3
	A	B	C	D

	0	1	2	3
0	0	1	1	0
1	0	0	1	0
2	0	0	0	0
3	0	0	0	0

Representation av grafer

Varför $\mathcal{O}\left(|V|^2\right)$?



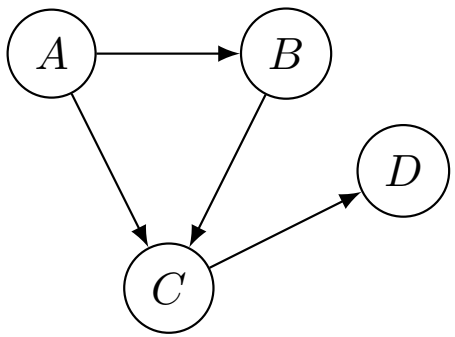
noder:

0	1	2	3
A	B	C	D

	0	1	2	3
0	0	1	1	0
1	0	0	1	0
2	0	0	0	1
3	0	0	0	0

Representation av grafer

Varför $\mathcal{O}\left(|V|^2\right)$?



noder:

0	1	2	3
A	B	C	D

	0	1	2	3
0	0	1	1	0
1	0	0	1	0
2	0	0	0	1
3	0	0	0	0

Representation av grafer

Varför $\mathcal{O}\left(|V|^2\right)$?

- Varje gång vi lägger till en ny nod så öker storleken på matrisen, och därför måste den omallokeras, vilket tar $\mathcal{O}\left(|V|^2\right)$ för den behöver kopiera hela matrisen.
- Vi kan anta att noderna är sorterade för då kan vi binärsöka fram dem (dock måste vi hitta dess sorterade plats vid insättning, men det spelar ingen roll för den dominerande termen är omallokeringen av matrisen).

Representation av grafer

Slutsatser om grannmatris

- Använder en del minne
- Väldigt snabb om bågarna ändras ofta
- Vi vill helst inte ändra noderna för det är dyrt

Representation av grafer

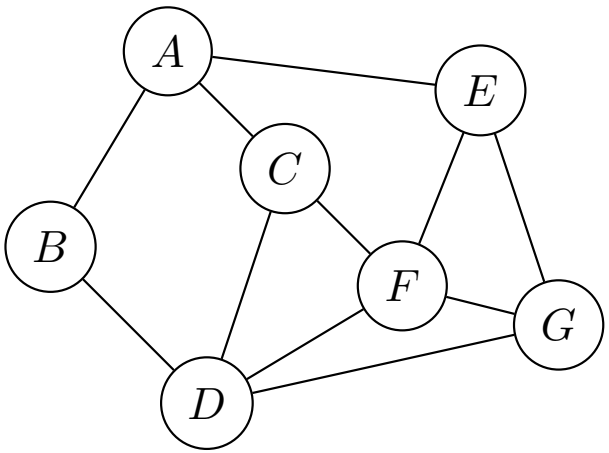
Idé #3

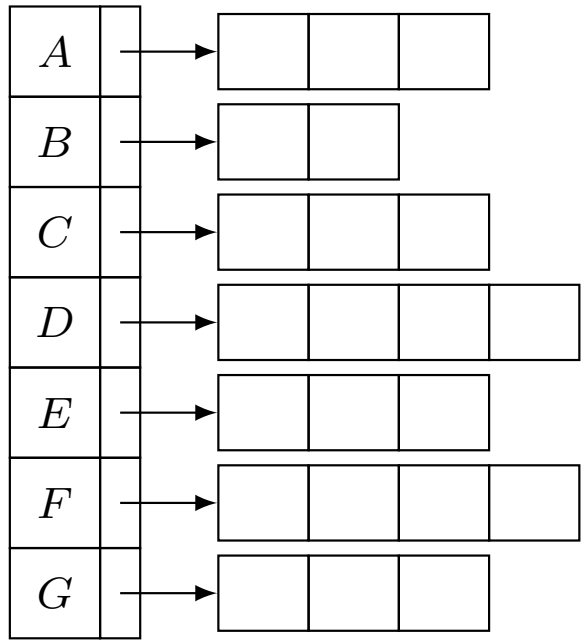
Idé #3

För varje nod, lagra vilka noder den har en båge till

Representation av grafer

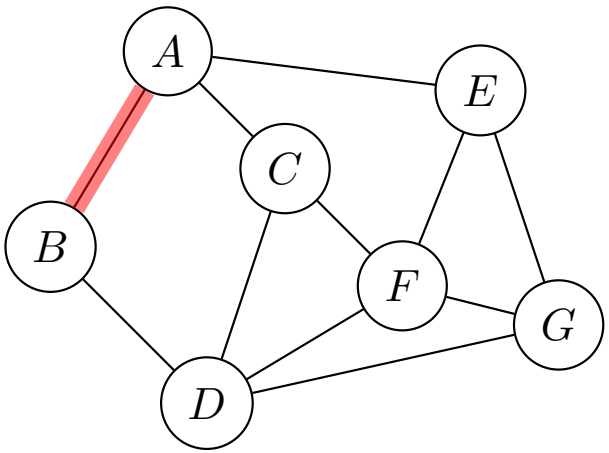
Grannlista

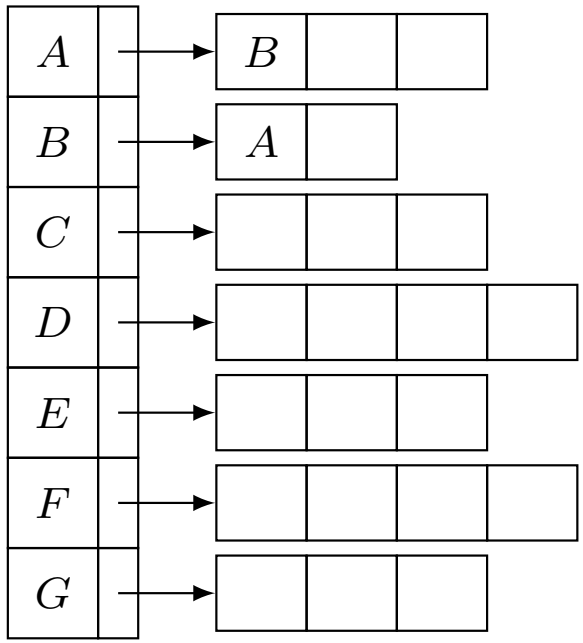




Representation av grafer

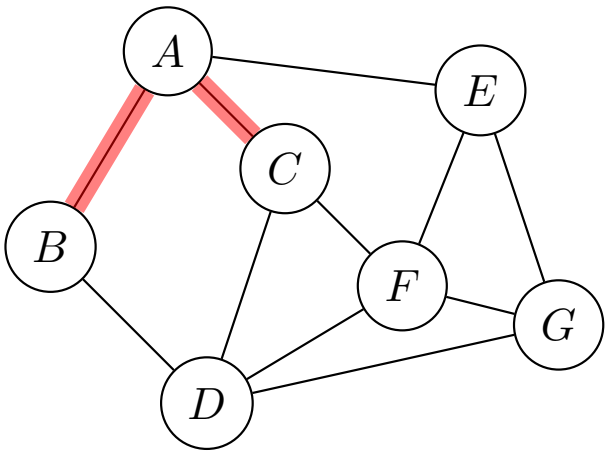
Grannlista

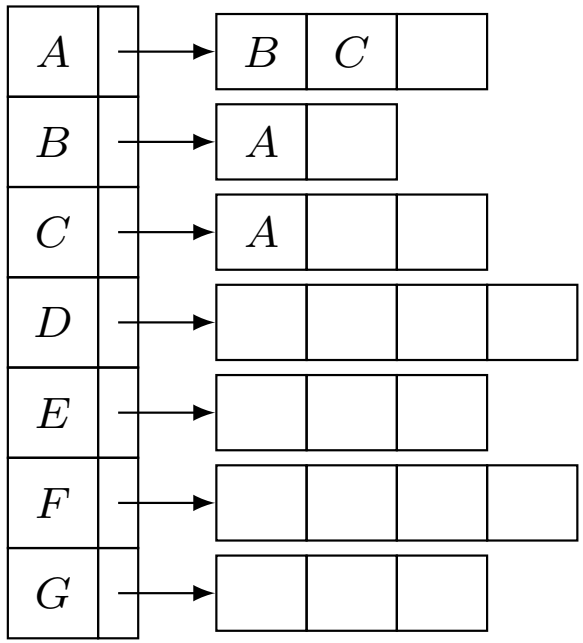




Representation av grafer

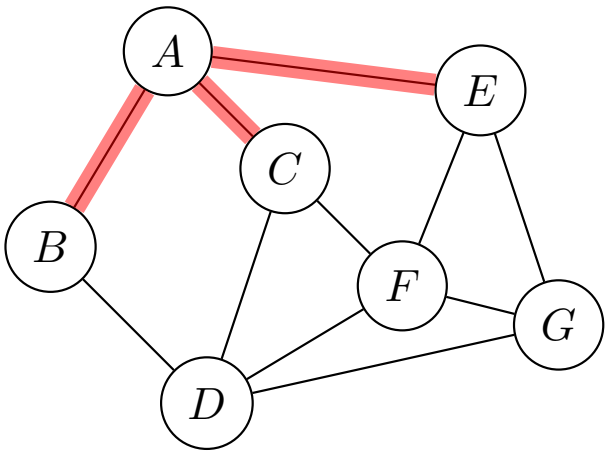
Grannlista

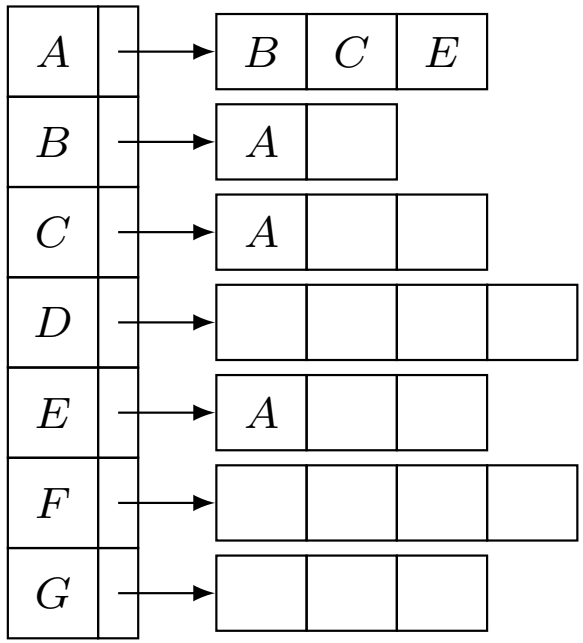




Representation av grafer

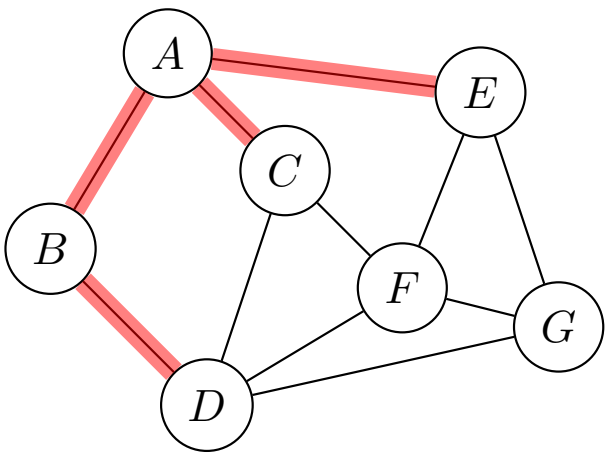
Grannlista

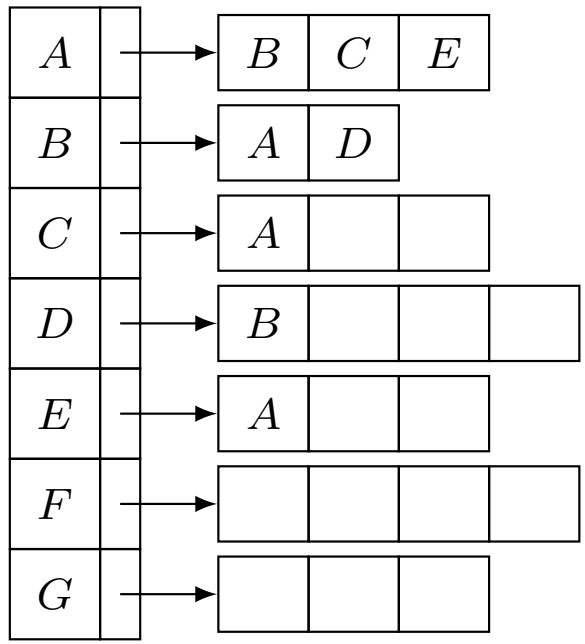




Representation av grafer

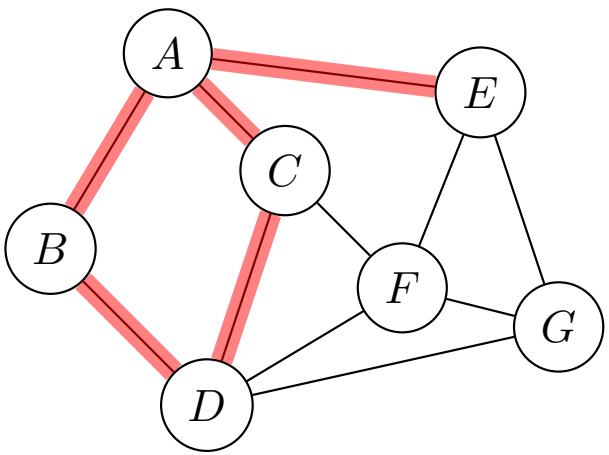
Grannlista

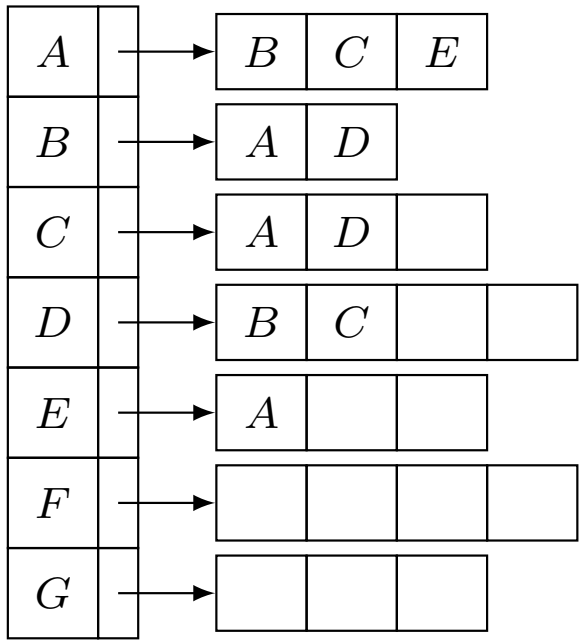




Representation av grafer

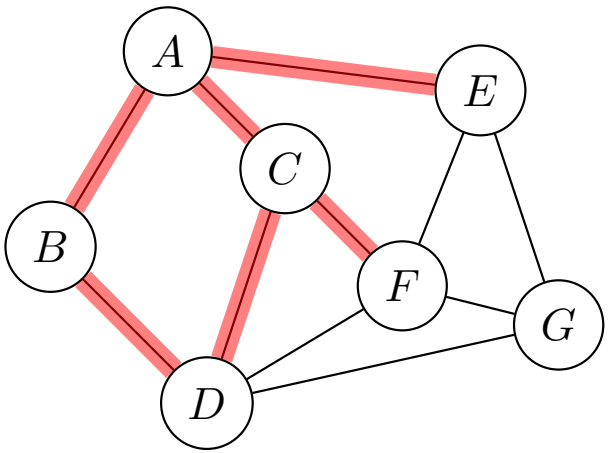
Grannlista

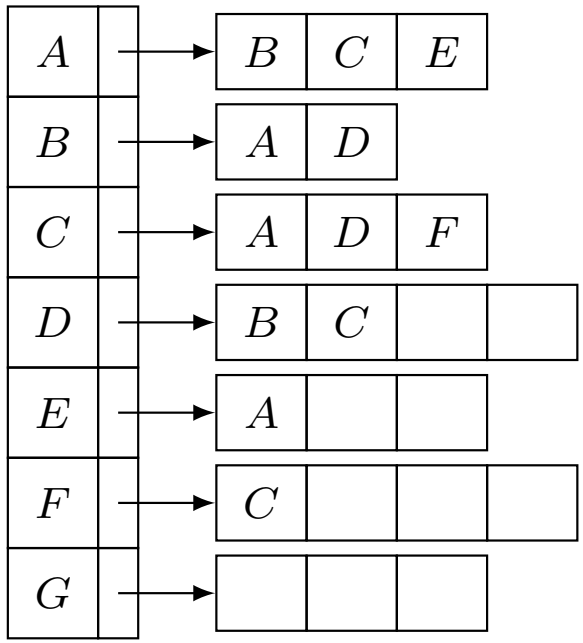




Representation av grafer

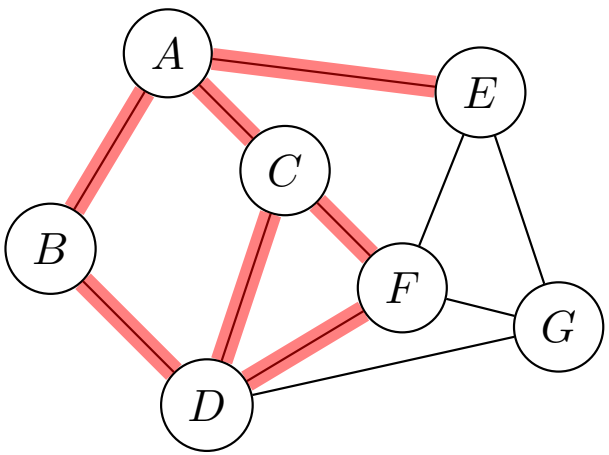
Grannlista

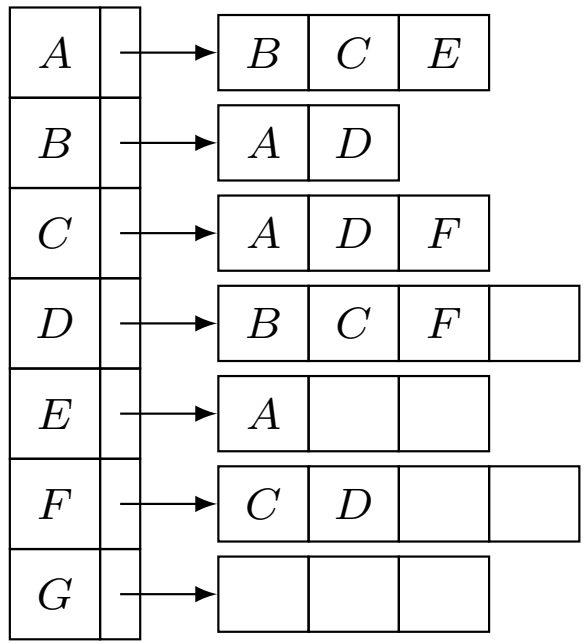




Representation av grafer

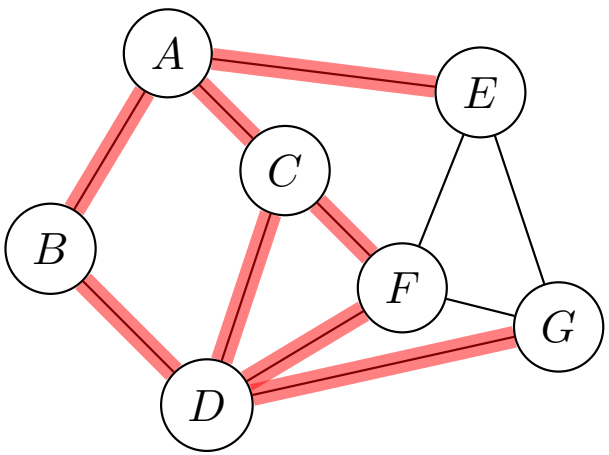
Grannlista

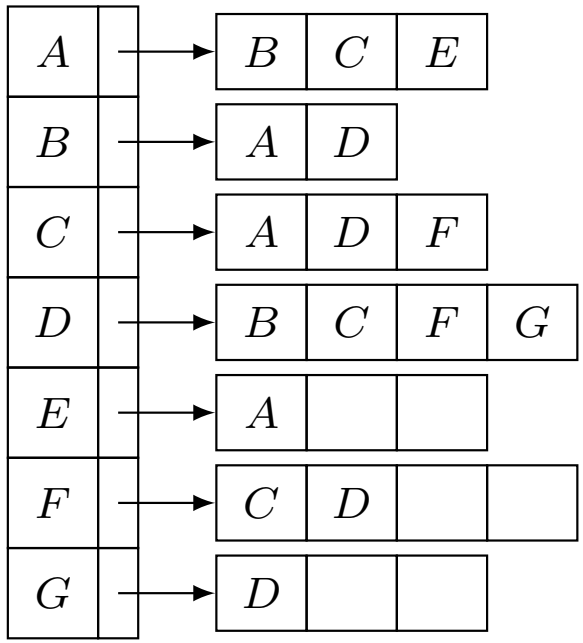




Representation av grafer

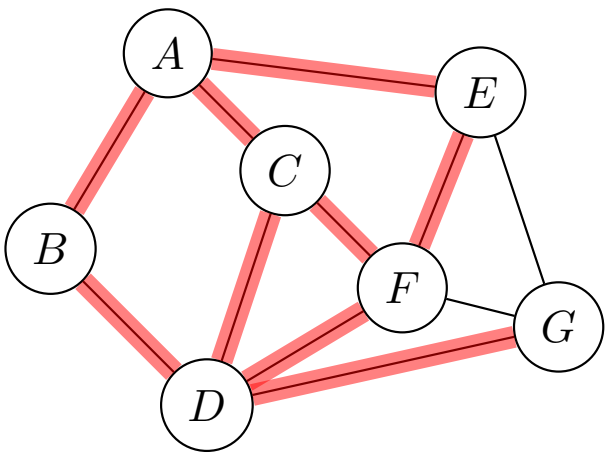
Grannlista

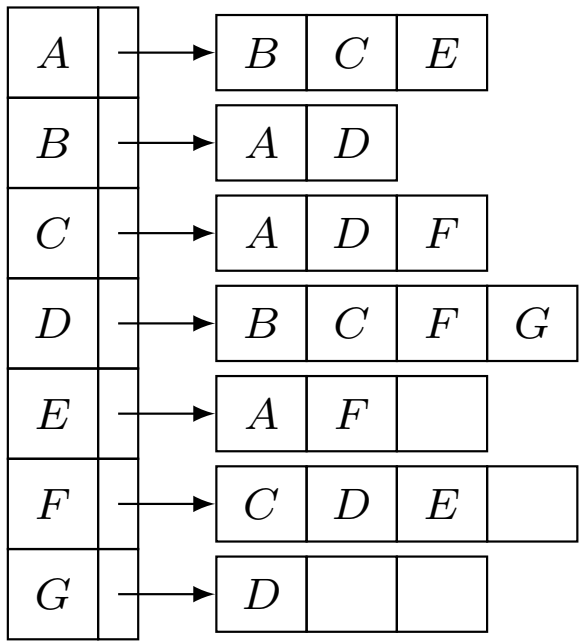




Representation av grafer

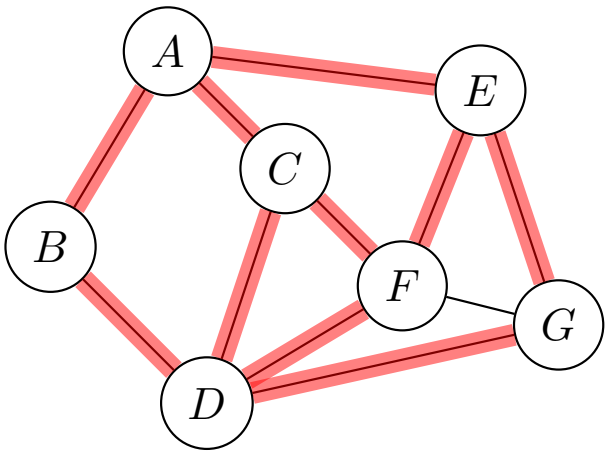
Grannlista

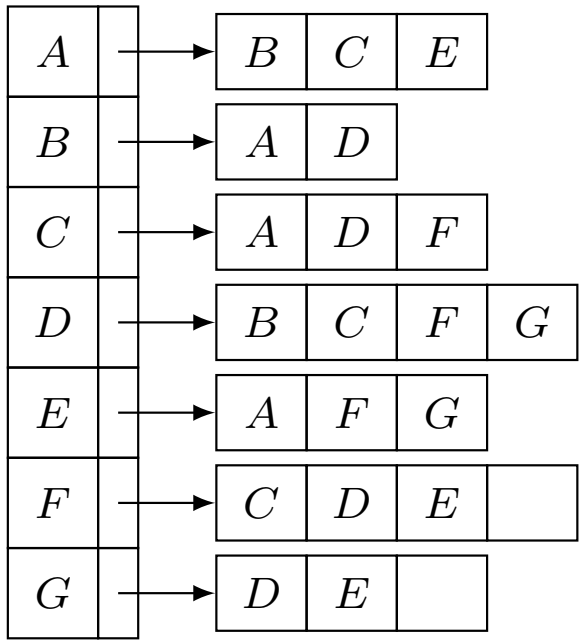




Representation av grafer

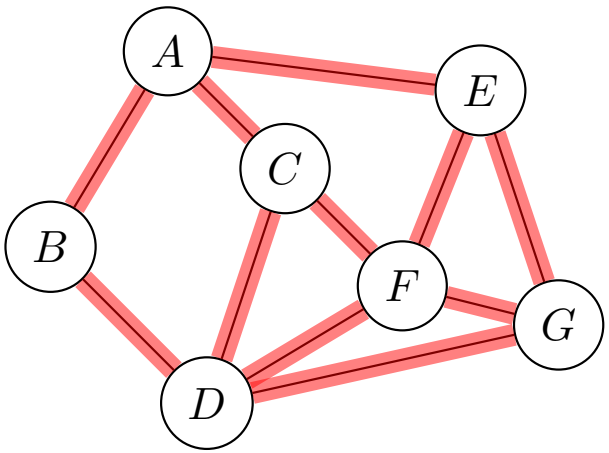
Grannlista

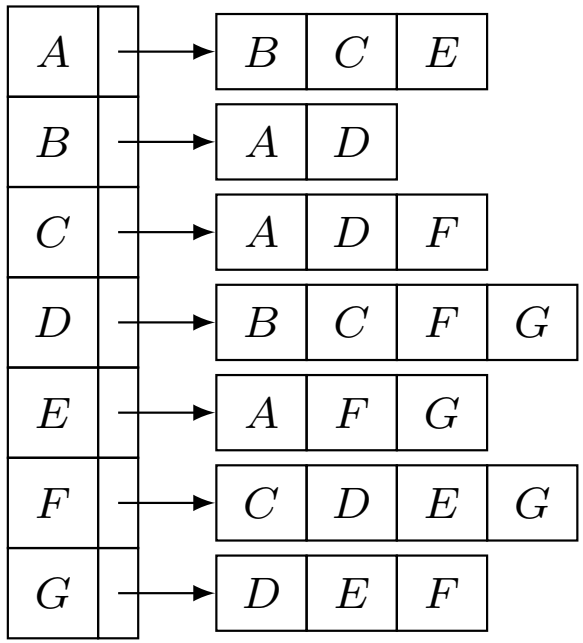




Representation av grafer

Grannlista





Representation av grafer

Analys av grannlista

- Antag hashtabell innehållandes dynamiska arrayer
- Insättning av noder och bågar amorterat $\mathcal{O}(1)$
- Borttagning av nod är $\mathcal{O}(|V|)$
- Borttagning av båge är $\mathcal{O}(\max_{v \in V} \{\deg(v)\})$
- Hitta nod är amorterat $\mathcal{O}(1)$
- Hitta båge är $\mathcal{O}(|V|)$
- Minnesanvändning: $\mathcal{O}(|V| + |E|)$

Representation av grafer

Jämförelse grannmatris och grannlista

Grannmatris

- Insättning nod: $\mathcal{O}\left(|V|^2\right)$
- Insättning båge: $\mathcal{O}\left(1\right)$
- Ta bort nod: $\mathcal{O}\left(|V|^2\right)$
- Ta bort båge: $\mathcal{O}\left(1\right)$
- Hitta nod: $\mathcal{O}\left(\log|V|\right)$
- Hitta båge: $\mathcal{O}\left(1\right)$
- Minne: $\mathcal{O}\left(|V|+|V|^2\right)$

Grannlista

- Insättning nod: $\mathcal{O}\left(1\right)$
- Insättning båge: $\mathcal{O}\left(1\right)$
- Ta bort nod: $\mathcal{O}\left(|V|\right)$
- Ta bort båge: $\mathcal{O}\left(\max_{v\in V}\left\{\deg(v)\right\}\right)$
- Hitta nod: $\mathcal{O}\left(1\right)$ amorterat
- Hitta båge: $\mathcal{O}\left(|V|\right)$
- Minne: $\mathcal{O}\left(|V|+|E|\right)$

Representation av grafer

Jämförelse grannmatris och grannlista

Grannmatris

- Insättning nod: $\mathcal{O}\left(|V|^2\right)$
- Insättning båge: $\mathcal{O}\left(1\right)$
- Ta bort nod: $\mathcal{O}\left(|V|^2\right)$
- Ta bort båge: $\mathcal{O}\left(1\right)$
- Hitta nod: $\mathcal{O}\left(\log|V|\right)$
- Hitta båge: $\mathcal{O}\left(1\right)$
- Minne: $\mathcal{O}\left(|V|+|V|^2\right)$

Grannlista

- Insättning nod: $\mathcal{O}\left(1\right)$
- Insättning båge: $\mathcal{O}\left(1\right)$
- Ta bort nod: $\mathcal{O}\left(|V|\right)$
- Ta bort båge: $\mathcal{O}\left(\max_{v\in V}\left\{\deg(v)\right\}\right)$
- Hitta nod: $\mathcal{O}\left(1\right)$ amorterat
- Hitta båge: $\mathcal{O}\left(|V|\right)$
- Minne: $\mathcal{O}\left(|V|+|E|\right)$

Ingen klockren vinnare...

Representation av grafer

Avslutande ord

- Grannmatriser och grannlistor är bra på olika saker.
- Grannmatriser är en oerhört effektiv representation i grafer med många bågar och där noderna är relativt oförändrade.
- Grannlistor är bra i glesa grafer där det är relativt få bågar, men där noderna är något mer dynamiska.
- Notera specifikt att $|V| + 1 \leq |E| \leq |V|(|V| - 1)$
- Om $|E| \in \Theta\left(|V|^2\right)$ så kommer grannlistans minnesanvändning i regel vara värre än grannmatrisens, och borttagning av bågar i grannlistan blir $\mathcal{O}\left(|V|\right)$ (ty vi har en nästan komplett graf)
- Om $|E| \in \mathcal{O}\left(|V|\right)$ så är grannlistans minnesanvändning mycket bättre än grannmatrisen, och borttagning av bågar i grannlistan blir $\mathcal{O}\left(1\right)$ (ty vi har ungefär en båge per nod, så $\deg(v) \approx 1$ för alla v)

Grafsökning

Länkad representation

```
1  class Node
2  {
3  public:
4    int data;
5    vector<Node*> edges;
6  };
```

Representation av grafer

Länkad representation

- I många fall används varken grannmatriser eller grannlistor.
- Typiskt exempel är länkade listor, trädstrukturer o.s.v. där en fullständig graf är dyrare än det simplare alternativet.
- Det finns en hel drös med grafproblem som inte kräver en fullständig representation av en graf, trots att problemet fundamentalt är en graf.
- Ett typiskt exempel på detta är om vi lagrar noderna i ett rutnät, eller om strukturen inte kräver borttagning/insättning av noder eller bågar.
- I dessa fall kan det exempelvis räcka med en länkad struktur av noder, där varje nod lagrar någon data samt alla grannoder.

Representation av grafer

Implicita grafer

```
1 vector<int> get_neighbours(int node)
2 {
3   vector<int> neighbours { };
4   // beräkna alla grannar till `node`
5   return neighbours;
6 }
```

Representation av grafer

Implicita grafer

- Ett annat typiskt exempel när en fullständig representation av grafer inte kan användas är när grafen är på tok för stor..
- Vad händer t.ex. om antalet noder och bågar är så många att det inte får plats i minnet, men vi kan för varje nod *beräkna* deterministiskt vad alla grannar är?
- Detta kallas oftast för att vi har en *implict* graf.

Representation av grafer

Hitta alla noder

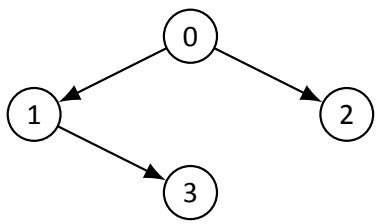
Hur kan vi hitta alla noder för dessa representationer?

1 Repetition: Grafer
2 Repetition: Riktade grafer
3 Representation av grafer
4 Grafsökning
5 Heap

LINKÖPING UNIVERSITY

Grafsökning

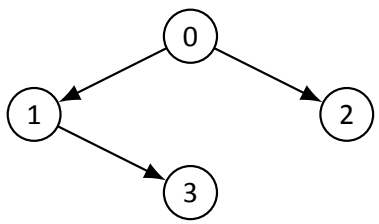
Rekursiv sökning – länkad struktur




```
1  vector<Node*> find_nodes(Node* node)
2  {
3    if (node == nullptr) return { };
4
5    vector<Node*> neighbours { };
6    for (Node* next : node->edges)
7    {
8      vector<Node*> nodes {
9        find_nodes(next);
10     };
11     for (Node* curr : nodes)
12       neighbours.push_back(curr);
13   }
14
15   return neighbours;
16 }
```

Grafsökning

Rekursiv sökning – länkad struktur




```
1  vector<Node*> find_nodes(Node* node)
2  {
3    if (node == nullptr) return { };
4
5    vector<Node*> neighbours { };
6    for (Node* next : node->edges)
7    {
8      vector<Node*> nodes {
9        find_nodes(next);
10     };
11     for (Node* curr : nodes)
12       neighbours.push_back(curr);
13   }
14
15   return neighbours;
16 }
```

Funkar!

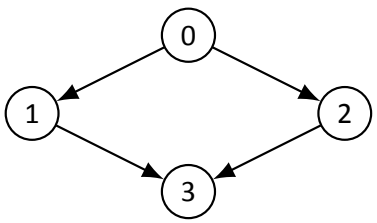
Grafsökning

Rekursiv sökning – länkad struktur

- Vi gör som tidigare och hittar alla noder rekursivt!
- I det här fallet går vi igenom alla grannar och beräknar alla noder som går att hitta från dem och lägger till dem i vår lista över noder.
- Detta funkar utmärkt för den givna grafen!

Grafsökning

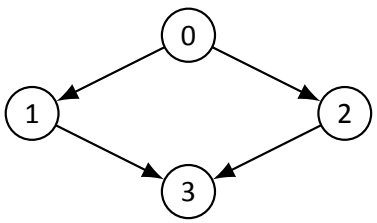
Ett problem




```
1  vector<Node*> find_nodes(Node* node)
2  {
3    if (node == nullptr) return { };
4
5    vector<Node*> nodes { node };
6    for (Node* next : node->edges)
7    {
8      vector<Node*> neighbours {
9        find_nodes(next);
10     };
11     for (Node* curr : neighbours)
12       nodes.push_back(curr);
13   }
14
15   return nodes;
16 }
```

Grafsökning

Ett problem




```
1  vector<Node*> find_nodes(Node* node)
2  {
3    if (node == nullptr) return { };
4
5    vector<Node*> nodes { node };
6    for (Node* next : node->edges)
7    {
8      vector<Node*> neighbours {
9        find_nodes(next);
10     };
11     for (Node* curr : neighbours)
12       nodes.push_back(curr);
13   }
14
15   return nodes;
16 }
```

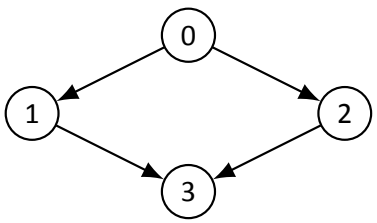
Grafsökning

Ett problem

- Ett mindre problem uppstår om vi lägger till en båge från nod 2 till nod 3, för då dyker nämligen nod 3 upp två gånger i vår lista, vilket vi vill undvika.
- För att lösa detta gör vi om vår `vector` till ett `set` (eller `unordered_set` om våra noder har en hashfunktion definierad)

Grafsökning

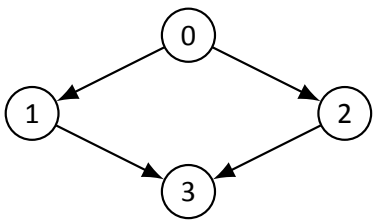
Lösning!




```
1  set<Node*> find_nodes(Node* node)
2  {
3    if (node == nullptr) return { };
4
5    set<Node*> nodes { node };
6    for (Node* next : node->edges)
7    {
8      set<Node*> neighbours {
9        find_nodes(next);
10     };
11     for (Node* curr : neighbours)
12       nodes.push_back(curr);
13   }
14
15   return nodes;
16 }
```

Grafsökning

Lösning!

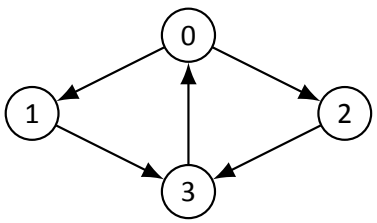



```
1  set<Node*> find_nodes(Node* node)
2  {
3    if (node == nullptr) return { };
4
5    set<Node*> nodes { node };
6    for (Node* next : node->edges)
7    {
8      set<Node*> neighbours {
9        find_nodes(next);
10     };
11     for (Node* curr : neighbours)
12       nodes.push_back(curr);
13   }
14
15   return nodes;
16 }
```

Funkar!

Grafsökning

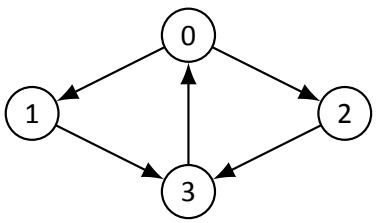
Ett nytt problem...




```
1  set<Node*> find_nodes(Node* node)
2  {
3    if (node == nullptr) return { };
4
5    set<Node*> nodes { node };
6    for (Node* next : node->edges)
7    {
8      set<Node*> neighbours {
9        find_nodes(next);
10     };
11     for (Node* curr : neighbours)
12       nodes.push_back(curr);
13   }
14
15   return nodes;
16 }
```

Grafsökning

Ett nytt problem...




```
1  set<Node*> find_nodes(Node* node)
2  {
3    if (node == nullptr) return { };
4
5    set<Node*> nodes { node };
6    for (Node* next : node->edges)
7    {
8      set<Node*> neighbours {
9        find_nodes(next);
10     };
11     for (Node* curr : neighbours)
12       nodes.push_back(curr);
13   }
14
15   return nodes;
16 }
```

Oändlig rekursion...

Grafsökning

Ett nytt problem...

- När vi lägger till en båge från nod 3 tillbaka till nod 0 kraschar vårt program för att det får slut på minne...?!
- Det som händer är att vi har introducerat en cykel i vår graf. Detta innebär att när vi söker efter grannar till nod 3 så kommer vi tillbaka till nod 0 och börjar i princip om sökningen från början.
- Detta problemet är i regel vad som gör grafsökning till en egen klass av problem.
- Lösning är dock förvånansvärt simpel...

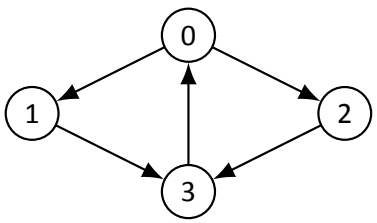
Grafsökning

Djupet-först-sökning (DFS)

- Lösningen är att vi kontrollerar huruvida en nod redan har hittats innan vi söker efter dess grannar.
- Om noden redan har hittats behöver vi inte kolla den igen så vi avbryter.
- Vi skriver om funktionen så att rekursiva steg har tillgång till behållaren av hittade noder.

Grafsökning

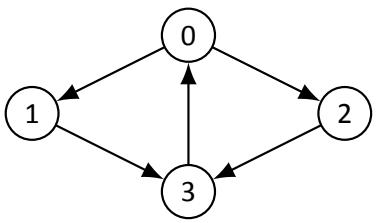
Djupet-först-sökning (DFS)




```
1  void find_nodes_help(Node* node, set<Node*>& found)
2  {
3    if (node == nullptr) return { };
4    if (found.count(node) > 0) return { };
5
6    found.insert(node);
7    for (Node* next : node->edges)
8      find_nodes_help(next, found);
9  }
10
11 set<Node*> find_nodes(Node* node)
12 {
13   set<Node*> nodes { };
14   find_nodes_help(node, nodes);
15   return nodes;
16 }
```

Grafsökning

Djupet-först-sökning (DFS)




```
1  void find_nodes_help(Node* node, set<Node*>& found)
2  {
3    if (node == nullptr) return { };
4    if (found.count(node) > 0) return { };
5
6    found.insert(node);
7    for (Node* next : node->edges)
8      find_nodes_help(next, found);
9  }
10
11 set<Node*> find_nodes(Node* node)
12 {
13   set<Node*> nodes { };
14   find_nodes_help(node, nodes);
15   return nodes;
16 }
```

Funkar!

Grafsökning

Djupet-först-sökning (DFS)

- Som tidigare diskuterat i kursen är i regel rekursiva lösningar något dyrare än iterativa varianter.
- Så vi testar att implementera DFS iterativt istället.
- Nyckelobservationen här är att vi måste hålla koll på vilka noder vi ska besöka härnäst genom att successivt lägga till nya noder när de upptäcks.
- Nästa nod vi vill besöka är i regel den senaste vi hittade (om vi vill få samma beteende som den rekursiva implementationen) så en stack låter lämpligt!

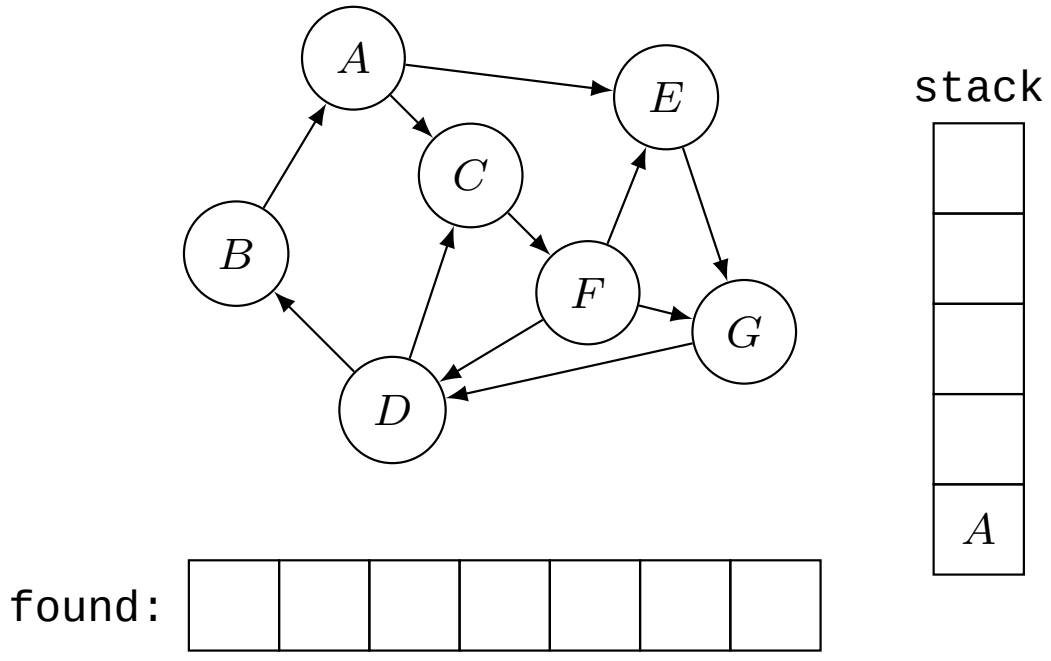
Grafsökning

Iterativ variant

```
1  set<Node*> DFS(Node* start)
2  {
3    set<Node*> found;
4    stack<Node*> stack;
5    stack.push(start);
6    while (!stack.empty())
7    {
8      Node* node { stack.top() };
9      stack.pop();
10     if (node == nullptr) continue;
11     if (found.count(node) > 0) continue;
12     found.insert(node);
13     for (Node* next : node->edges)
14       stack.push(next);
15   }
16   return found;
17 }
```

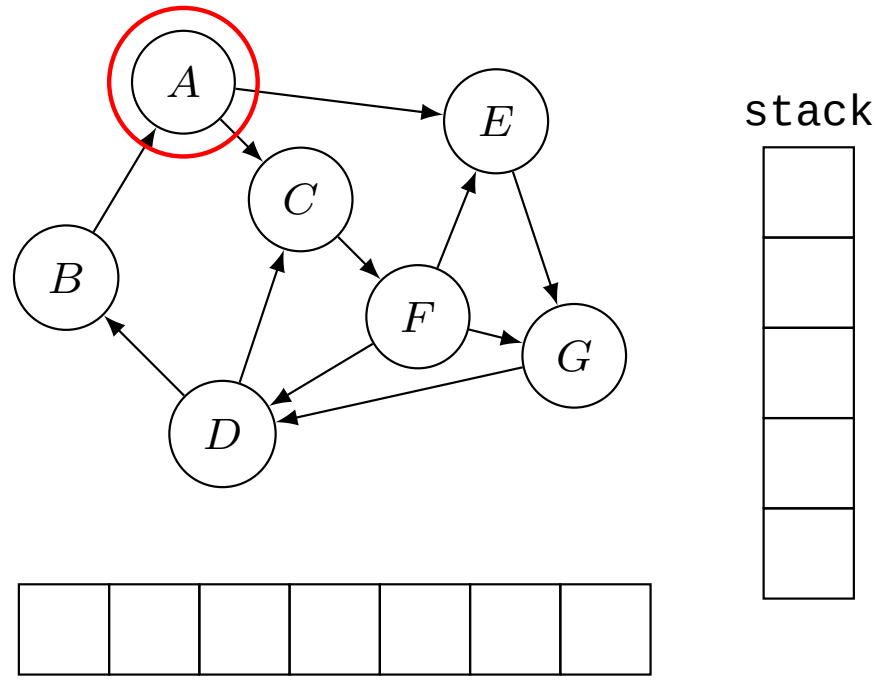
Grafsökning

djupet-först-sökning (DFS)



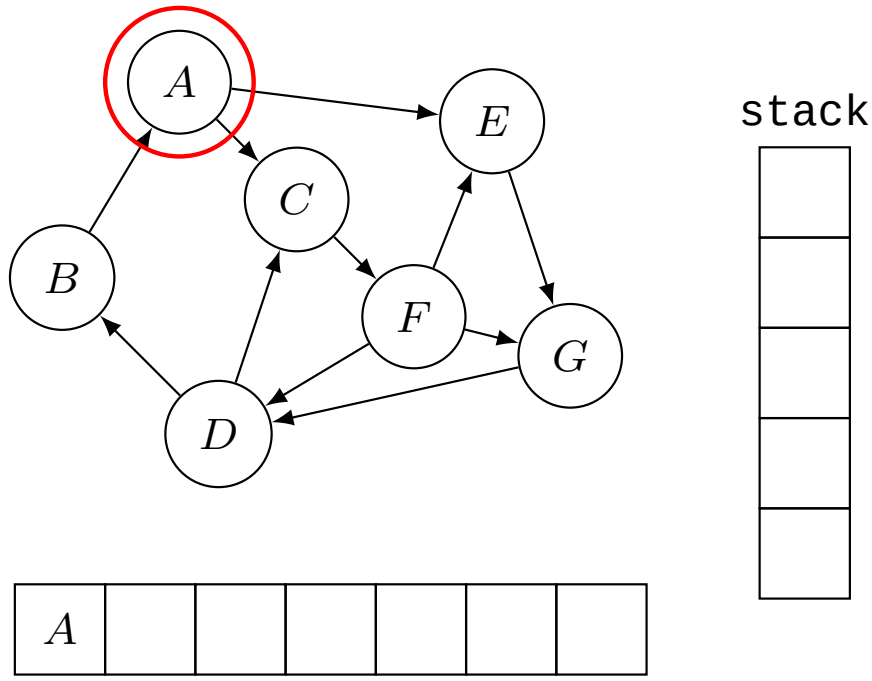
Grafsökning

djupet-först-sökning (DFS)



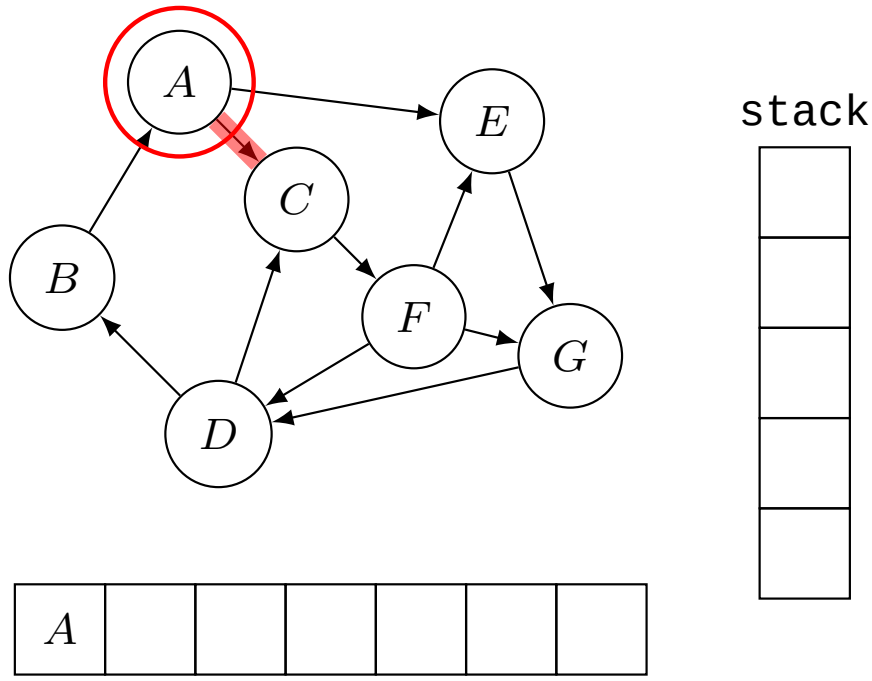
Grafsökning

djupet-först-sökning (DFS)



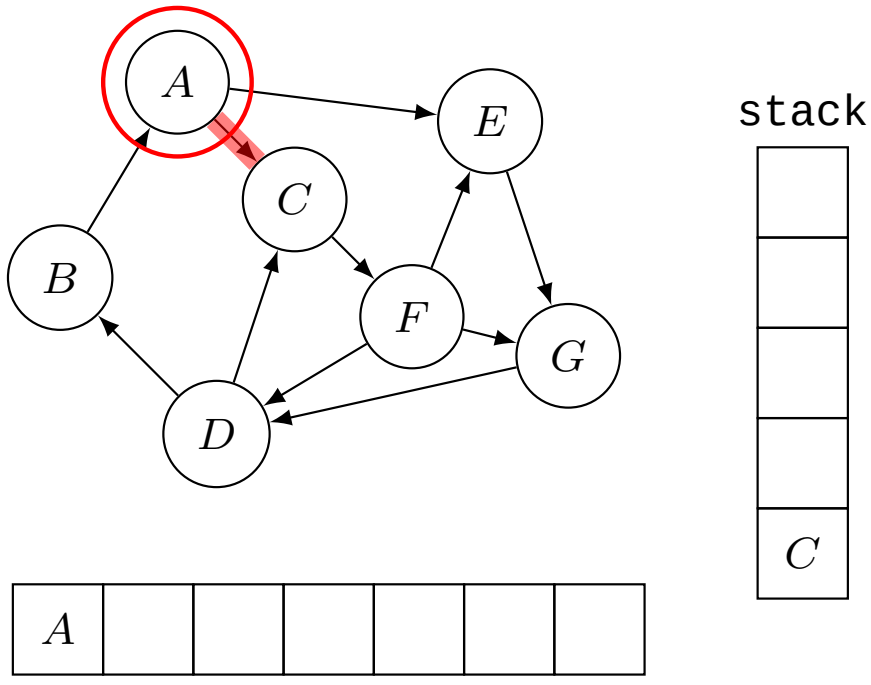
Grafsökning

djupet-först-sökning (DFS)



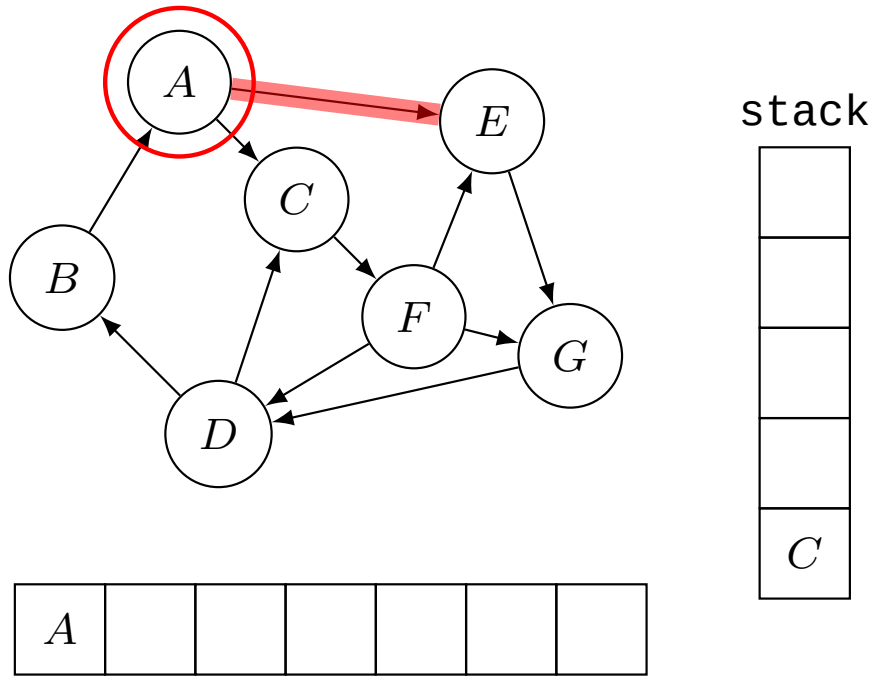
Grafsökning

djupet-först-sökning (DFS)



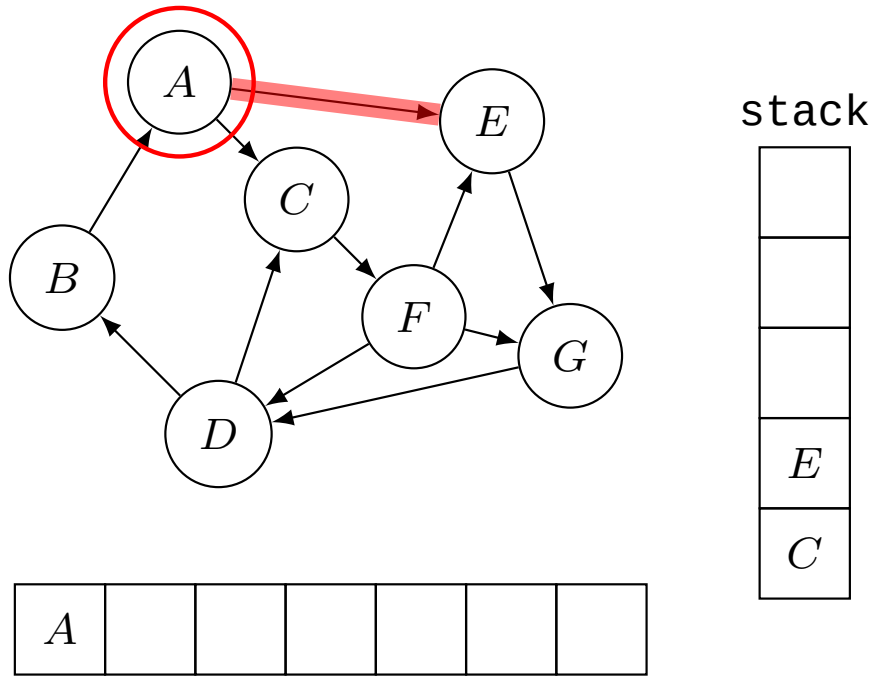
Grafsökning

djupet-först-sökning (DFS)



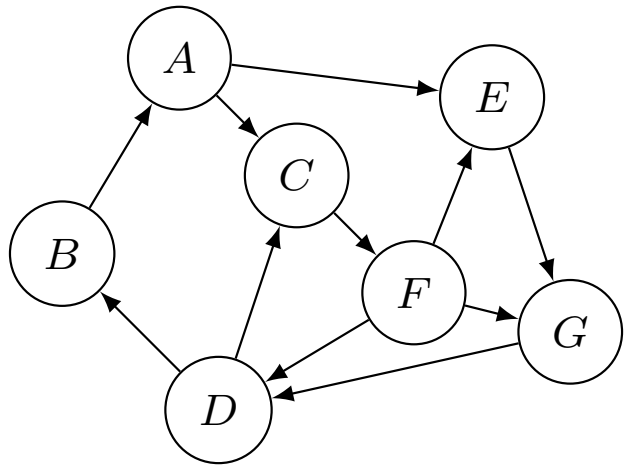
Grafsökning

djupet-först-sökning (DFS)

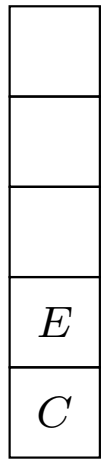


Grafsökning

djupet-först-sökning (DFS)



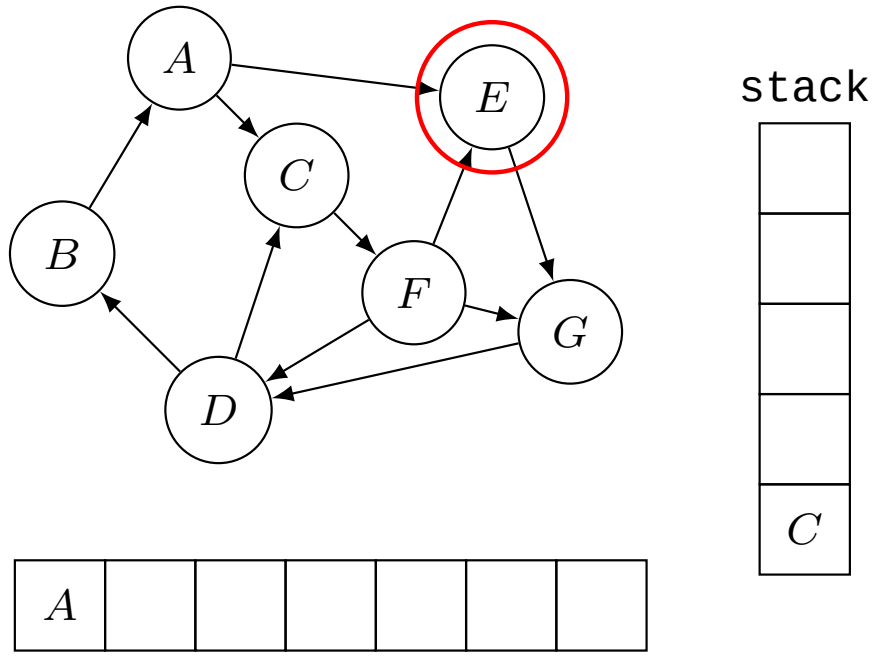
stack



found:

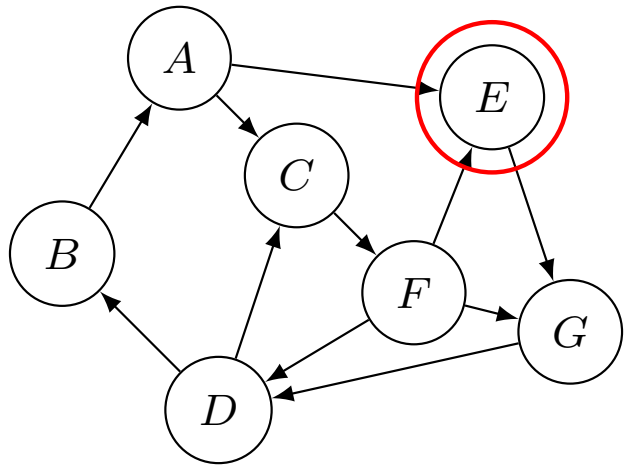
Grafsökning

djupet-först-sökning (DFS)

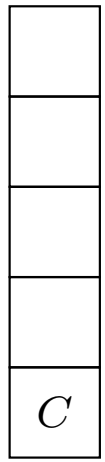


Grafsökning

djupet-först-sökning (DFS)



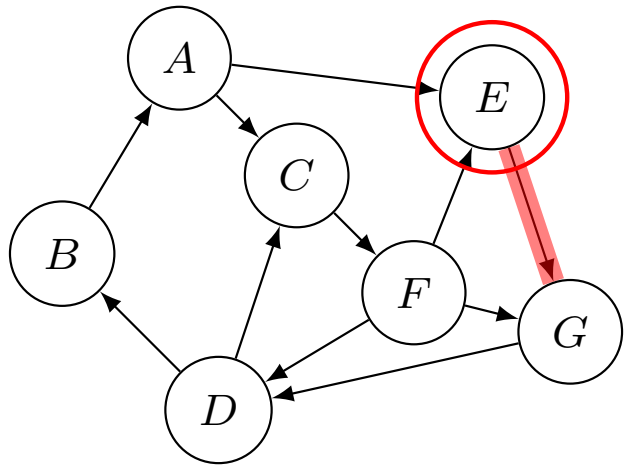
stack



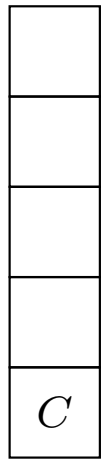
found:

Grafsökning

djupet-först-sökning (DFS)



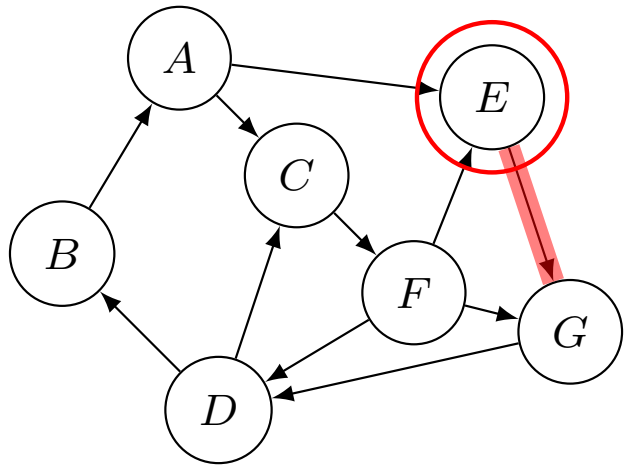
stack



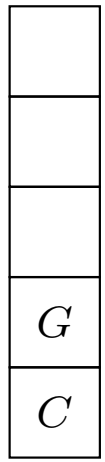
found:

Grafsökning

djupet-först-sökning (DFS)



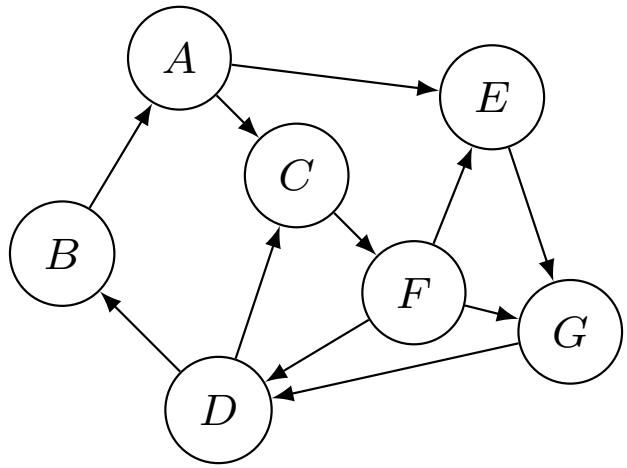
stack



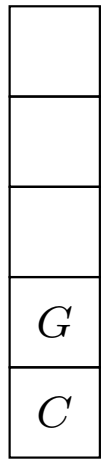
found:

Grafsökning

djupet-först-sökning (DFS)



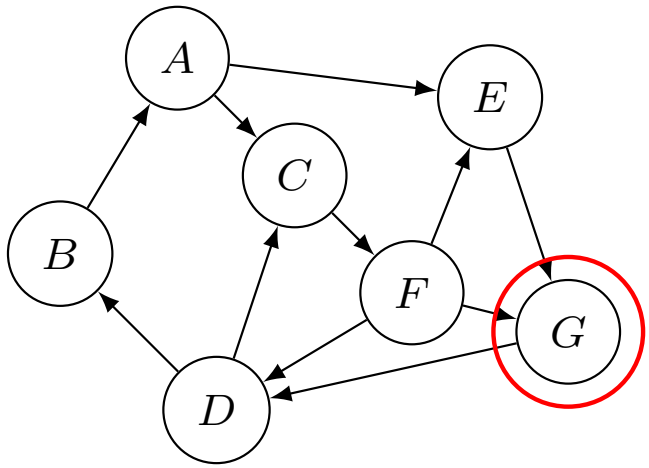
stack

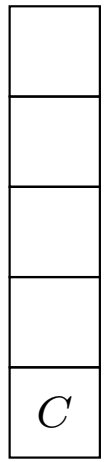


found:

Grafsökning

djupet-först-sökning (DFS)

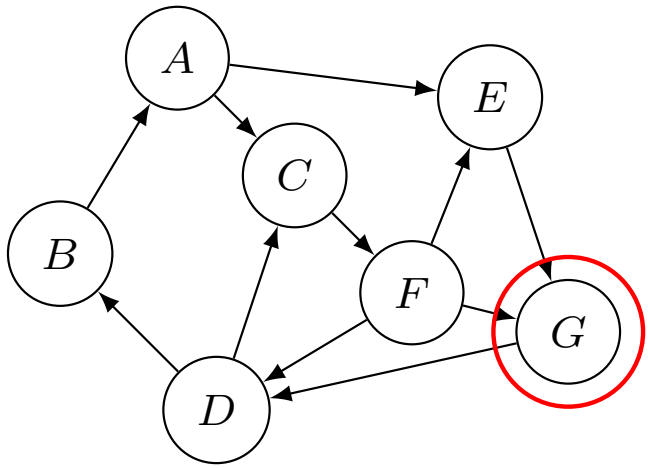




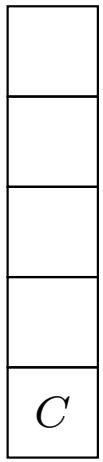
found: A E

Grafsökning

djupet-först-sökning (DFS)



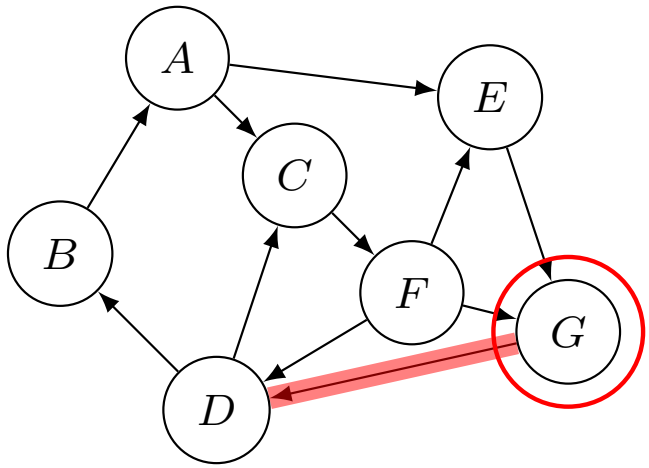
stack



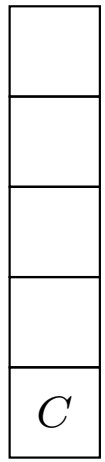
found:

Grafsökning

djupet-först-sökning (DFS)



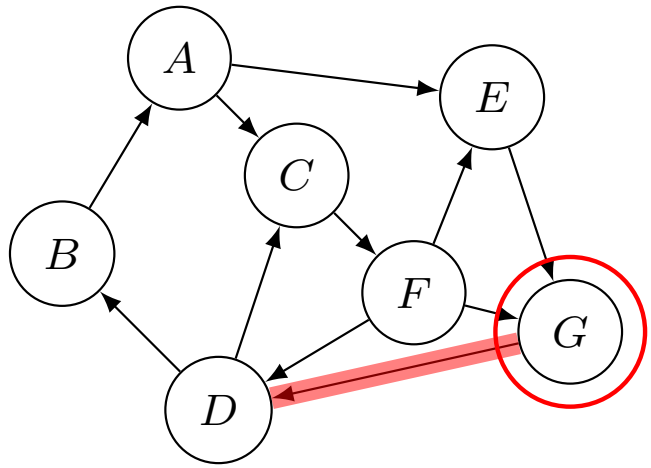
stack



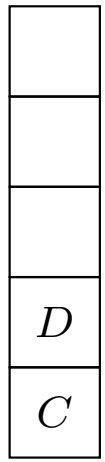
found:

Grafsökning

djupet-först-sökning (DFS)



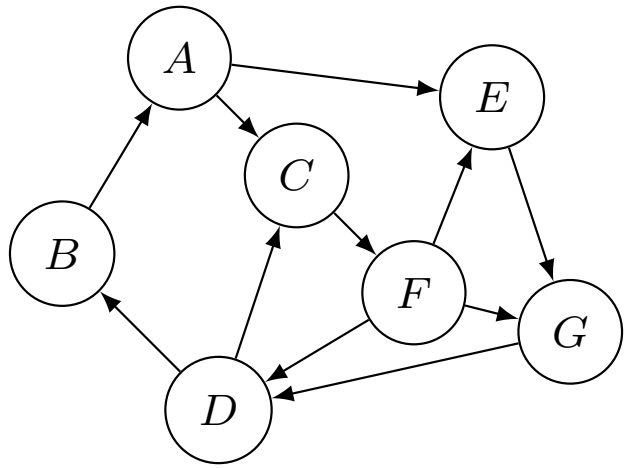
stack



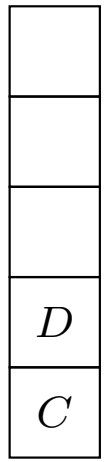
found:

Grafsökning

djupet-först-sökning (DFS)



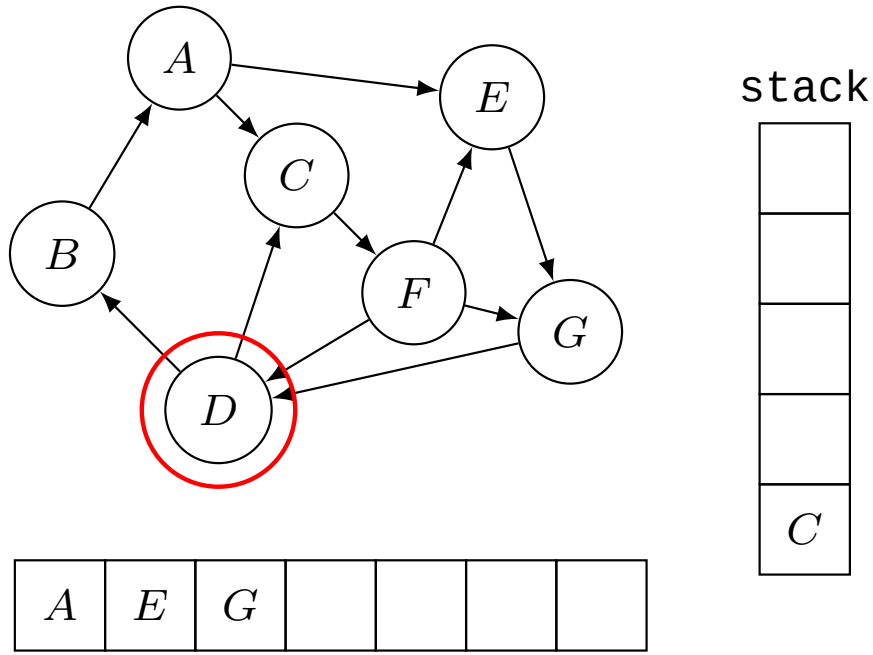
stack



found:

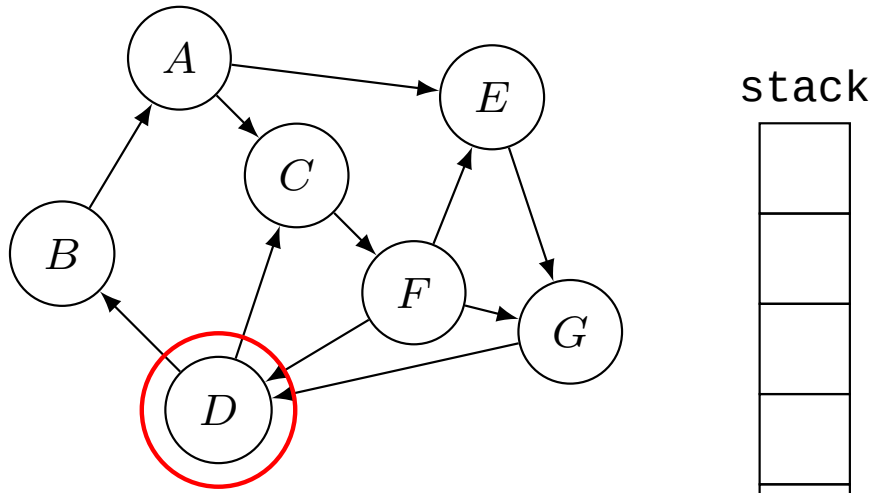
Grafsökning

djupet-först-sökning (DFS)



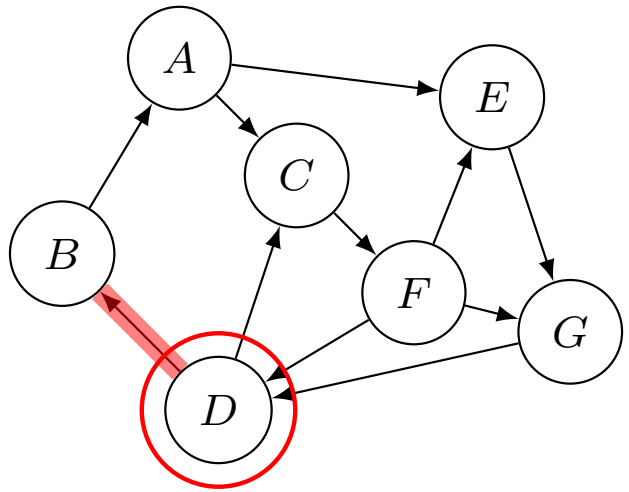
Grafsökning

djupet-först-sökning (DFS)

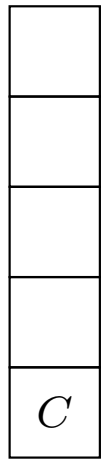


Grafsökning

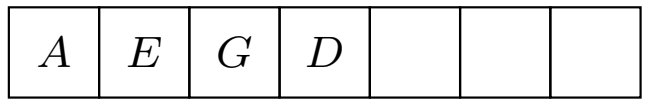
djupet-först-sökning (DFS)



stack

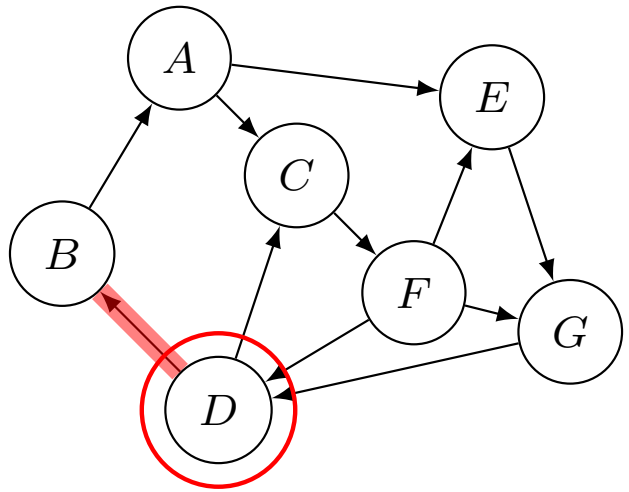


found:

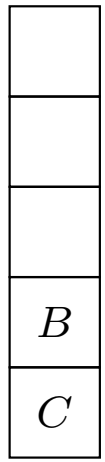


Grafsökning

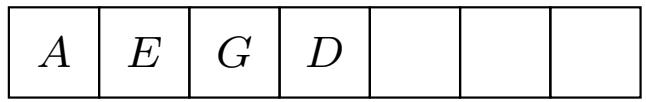
djupet-först-sökning (DFS)



stack

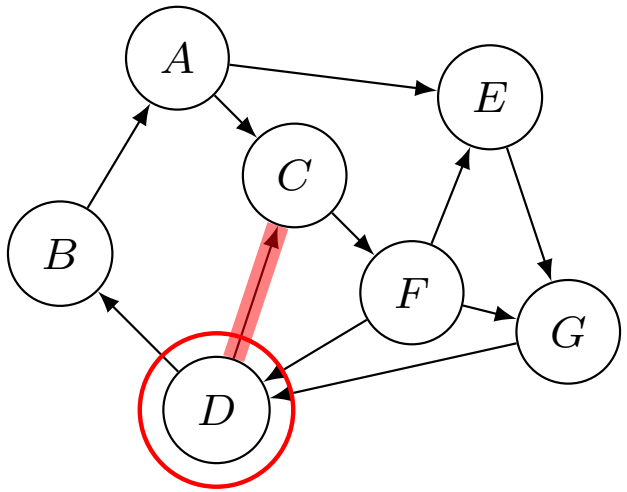


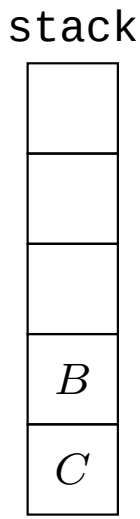
found:



Grafsökning

djupet-först-sökning (DFS)



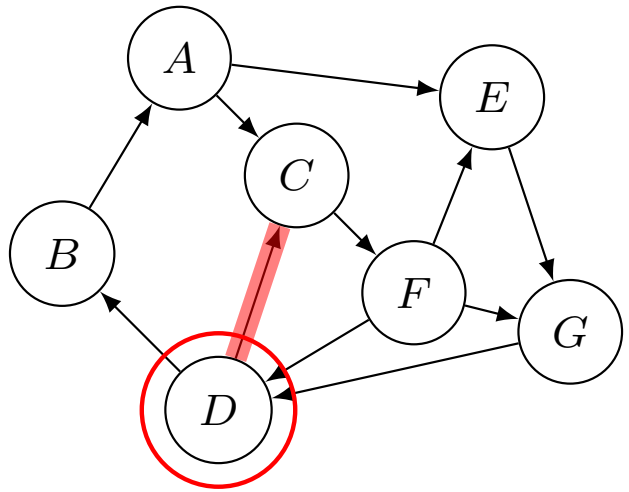


found:

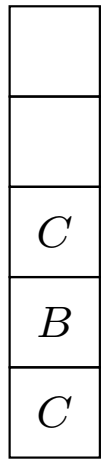
A	E	G	D			

Grafsökning

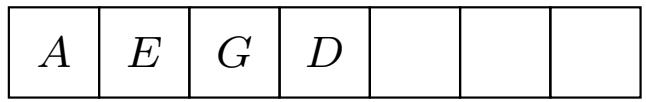
djupet-först-sökning (DFS)



stack

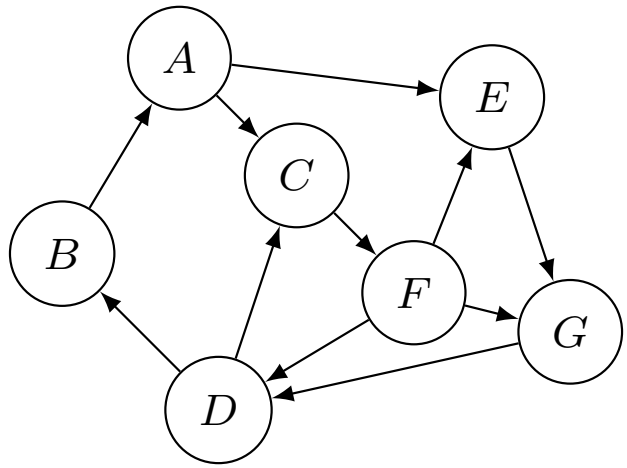


found:

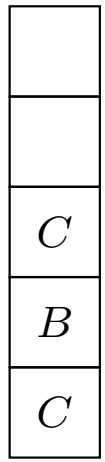


Grafsökning

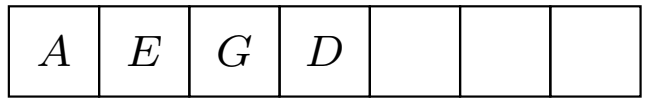
djupet-först-sökning (DFS)



stack

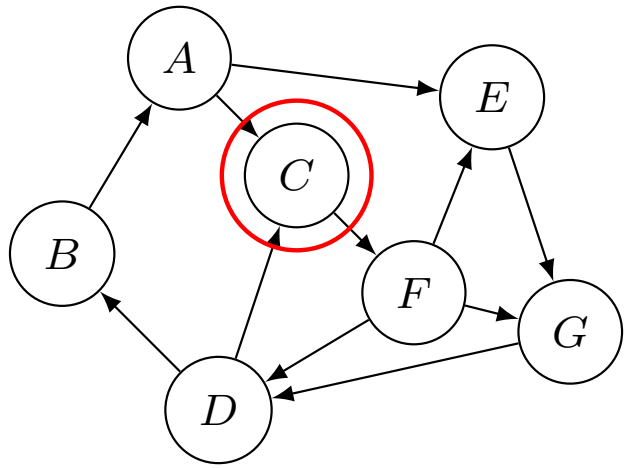


found:

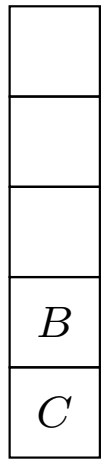


Grafsökning

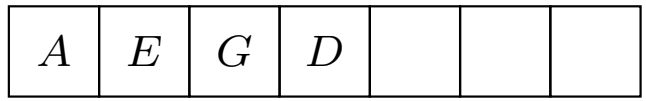
djupet-först-sökning (DFS)



stack

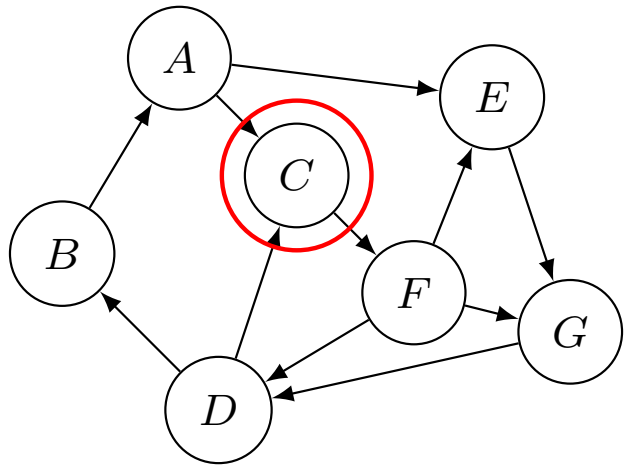


found:

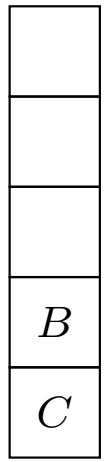


Grafsökning

djupet-först-sökning (DFS)



stack

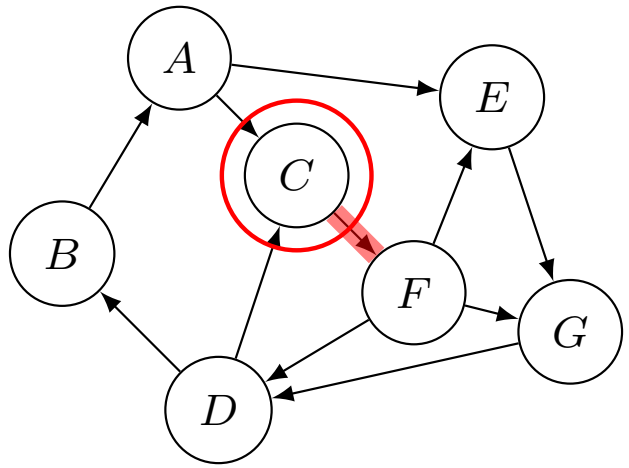


found:

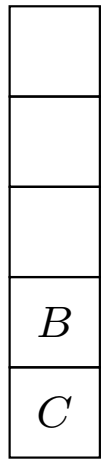
A	E	G	D	C		

Grafsökning

djupet-först-sökning (DFS)



stack

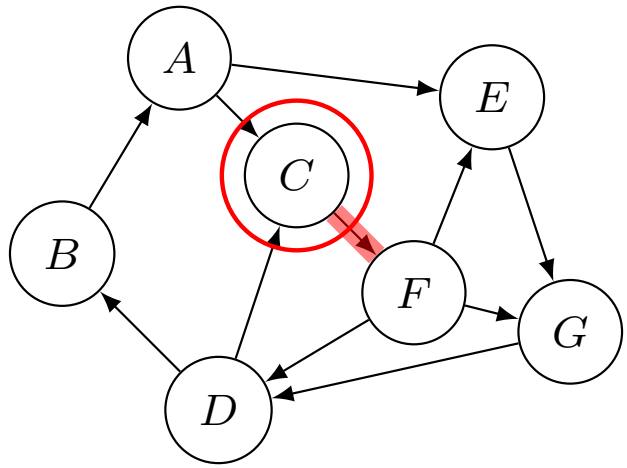


found:

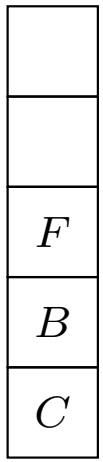
A	E	G	D	C		

Grafsökning

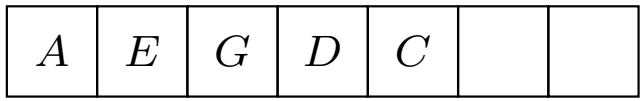
djupet-först-sökning (DFS)



stack

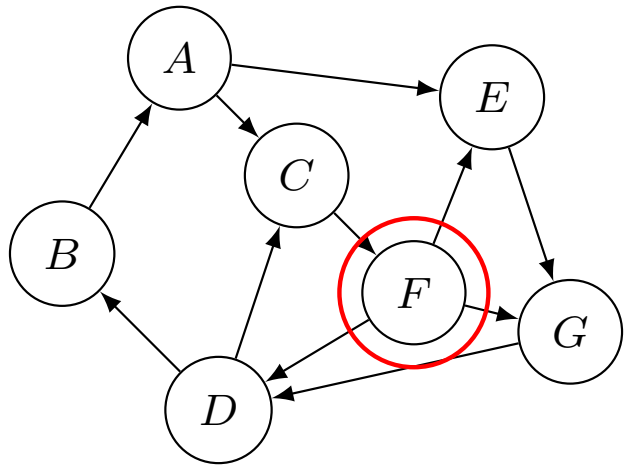


found:

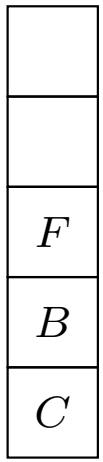


Grafsökning

djupet-först-sökning (DFS)



stack

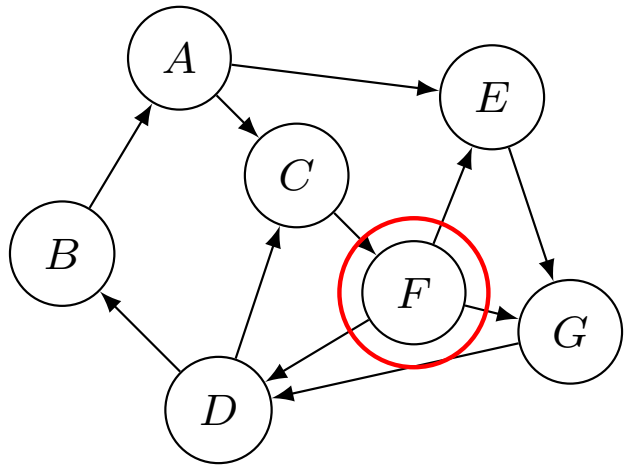


found:

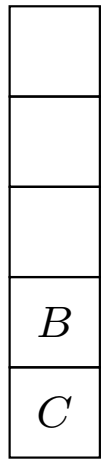
A	E	G	D	C		

Grafsökning

djupet-först-sökning (DFS)



stack

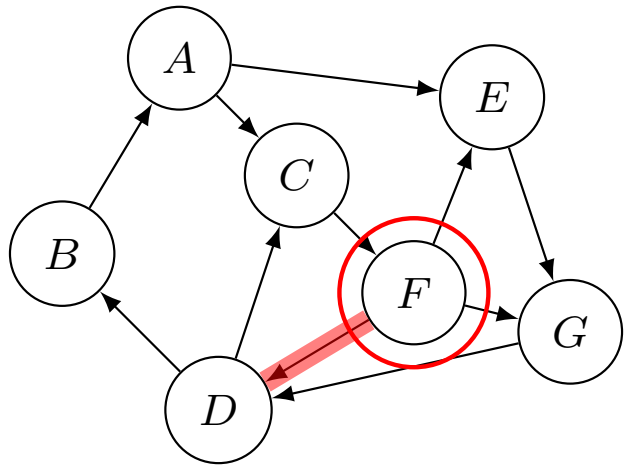


found:

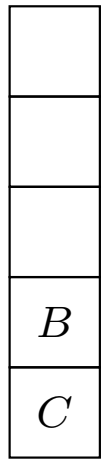
A	E	G	D	C	F	

Grafsökning

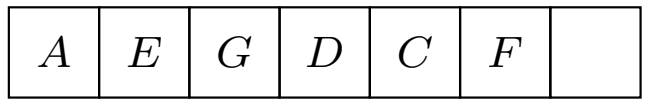
djupet-först-sökning (DFS)



stack

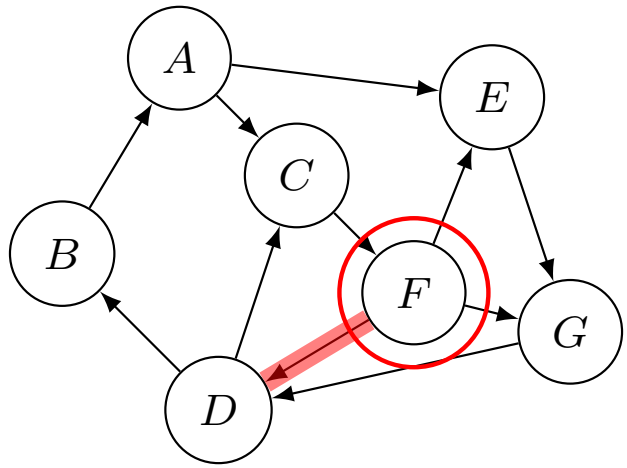


found:

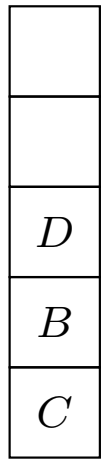


Grafsökning

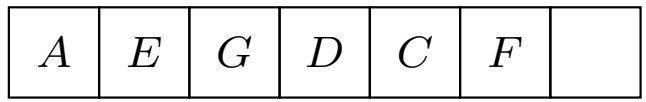
djupet-först-sökning (DFS)



stack

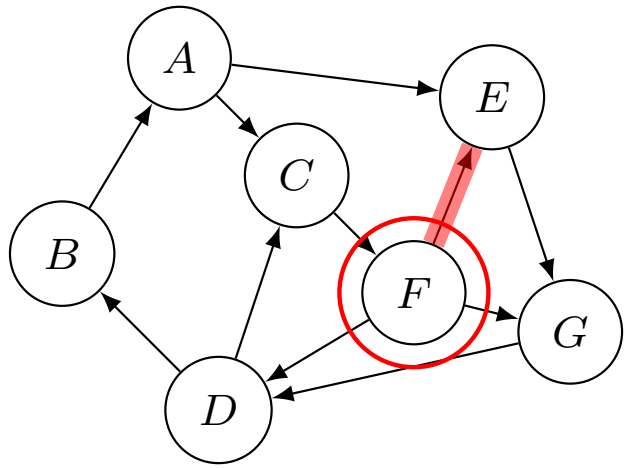


found:

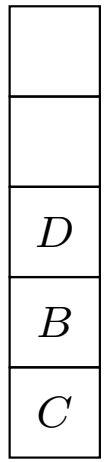


Grafsökning

djupet-först-sökning (DFS)



stack

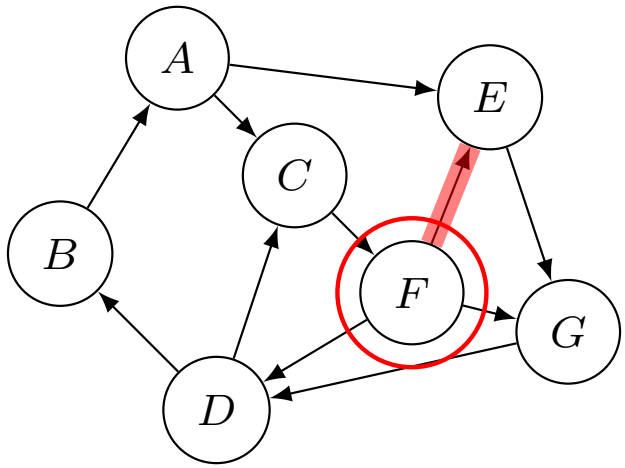


found:

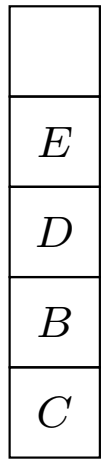
A	E	G	D	C	F	

Grafsökning

djupet-först-sökning (DFS)



stack

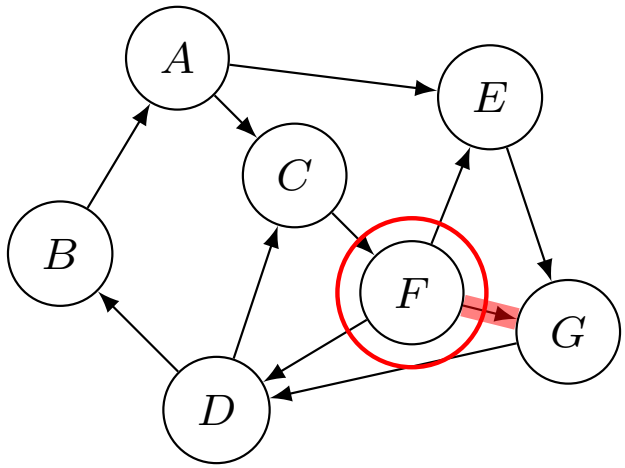


found:

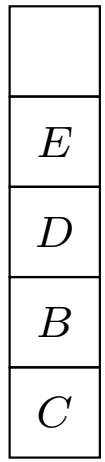
A	E	G	D	C	F	

Grafsökning

djupet-först-sökning (DFS)



stack

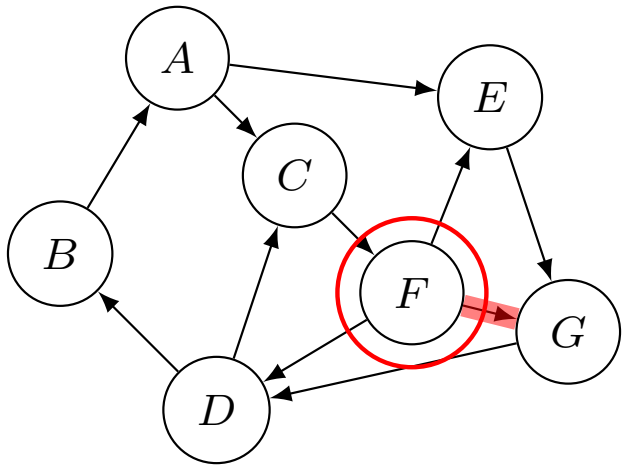


found:

A	E	G	D	C	F	

Grafsökning

djupet-först-sökning (DFS)



stack

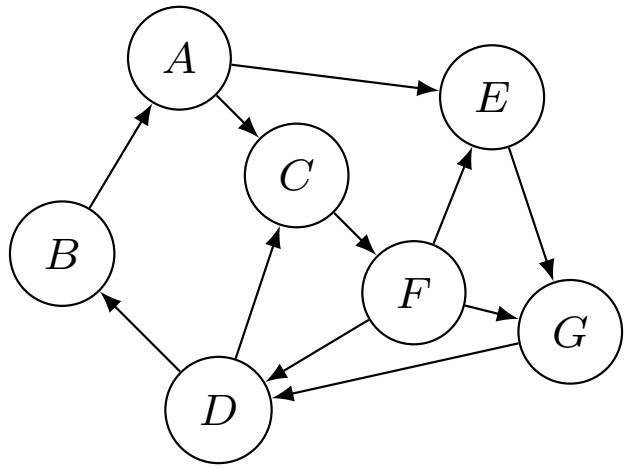
G
E
D
B
C

found:

A	E	G	D	C	F	

Grafsökning

djupet-först-sökning (DFS)



stack

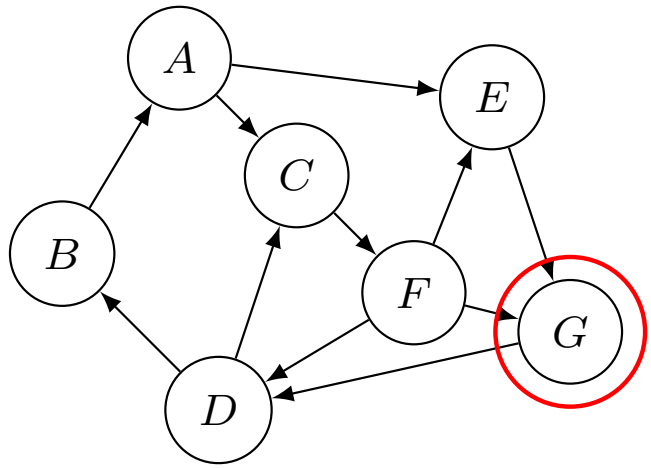
G
E
D
B
C

found:

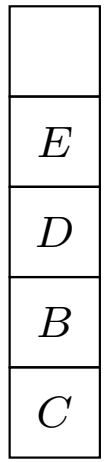
A	E	G	D	C	F	

Grafsökning

djupet-först-sökning (DFS)



stack

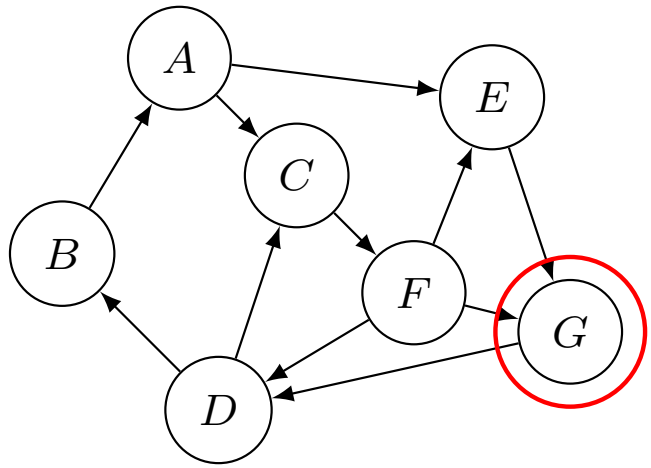


found:

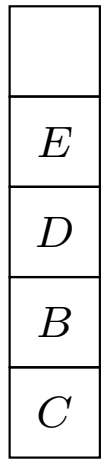
A	E	G	D	C	F	

Grafsökning

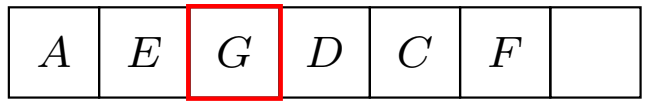
djupet-först-sökning (DFS)



stack

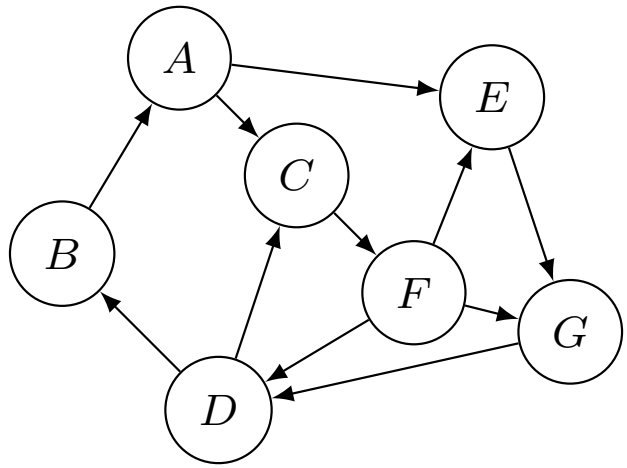


found:

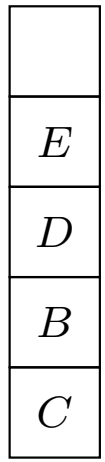


Grafsökning

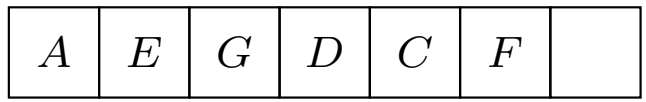
djupet-först-sökning (DFS)



stack

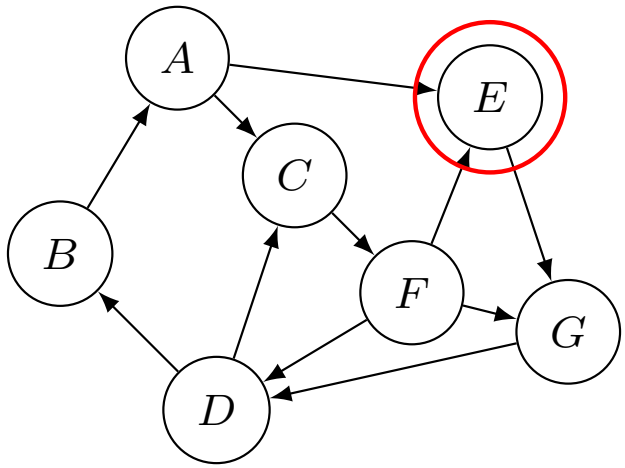


found:

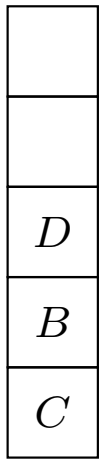


Grafsökning

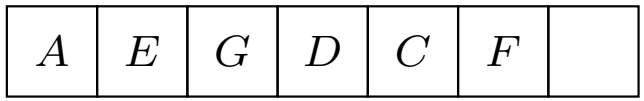
djupet-först-sökning (DFS)



stack

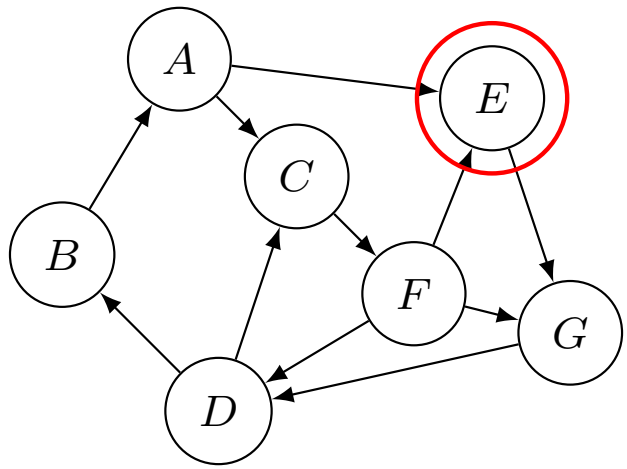


found:

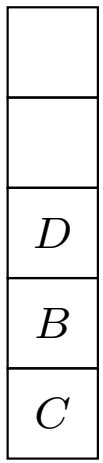


Grafsökning

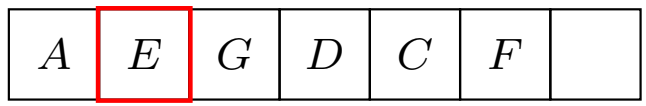
djupet-först-sökning (DFS)



stack

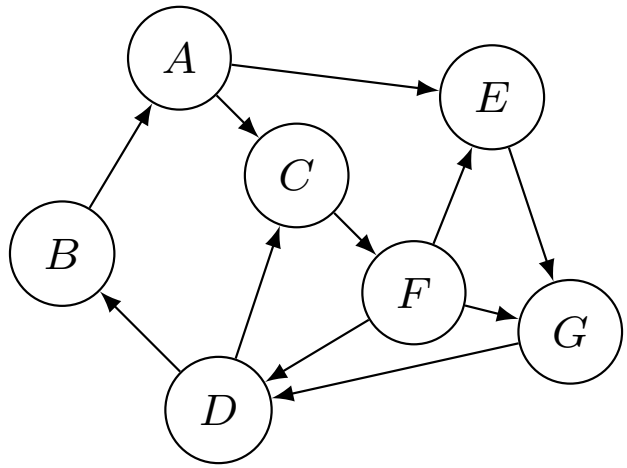


found:

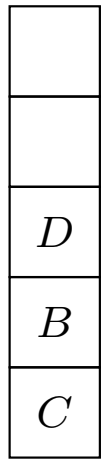


Grafsökning

djupet-först-sökning (DFS)



stack

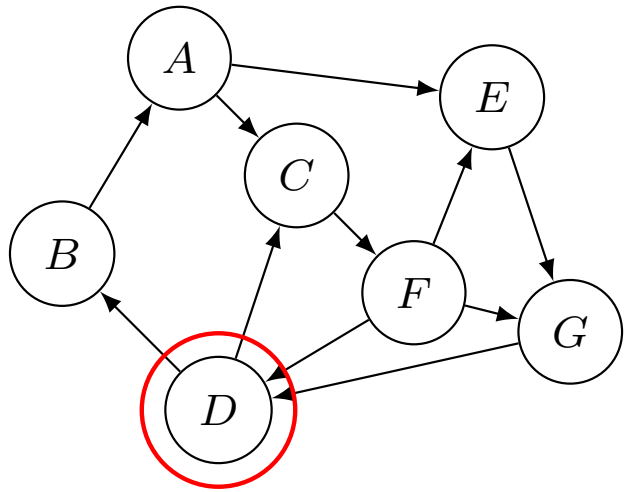


found:

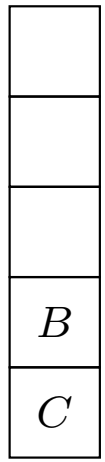
A	E	G	D	C	F	

Grafsökning

djupet-först-sökning (DFS)



stack

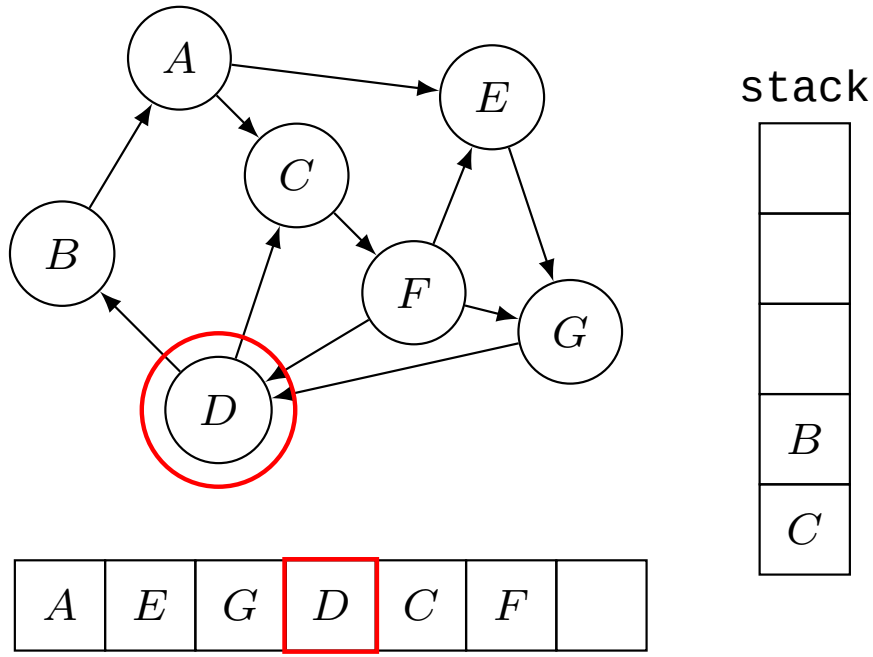


found:

A	E	G	D	C	F	

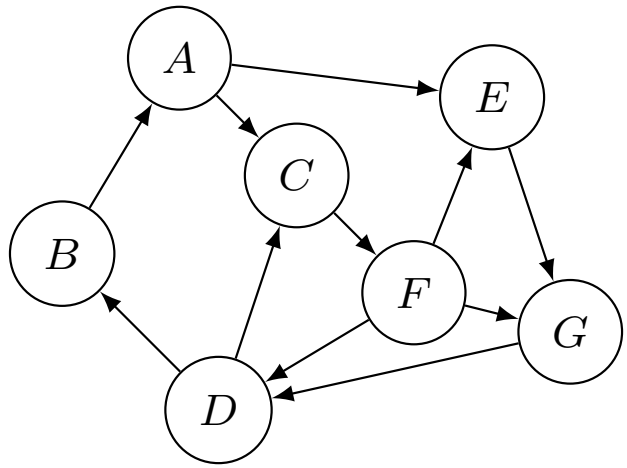
Grafsökning

djupet-först-sökning (DFS)

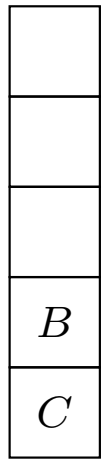


Grafsökning

djupet-först-sökning (DFS)



stack

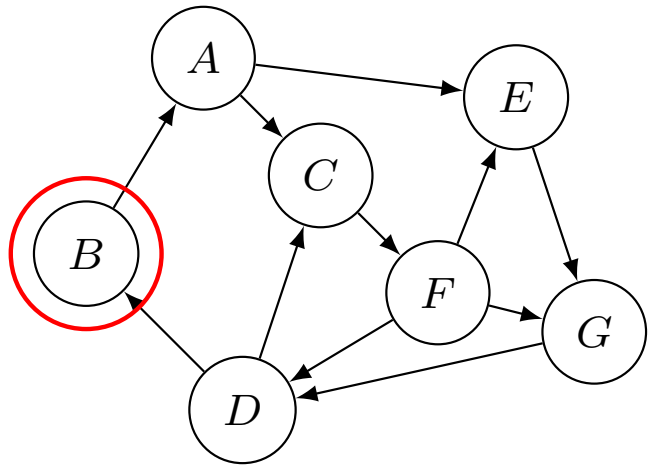


found:

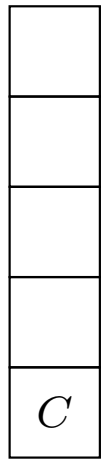
A	E	G	D	C	F	

Grafsökning

djupet-först-sökning (DFS)



stack

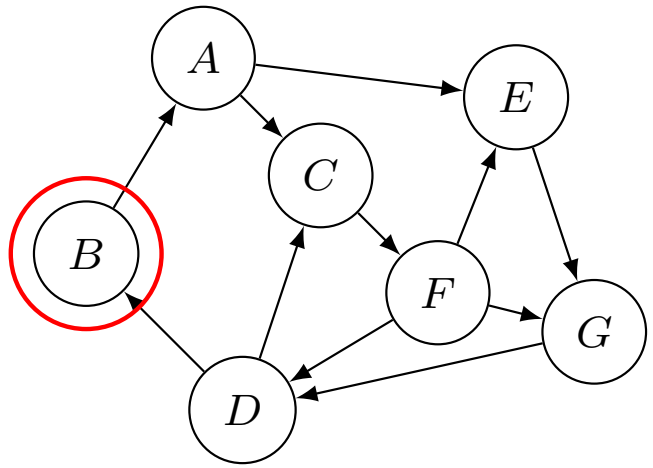


found:

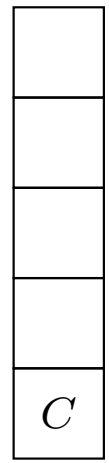
A	E	G	D	C	F	

Grafsökning

djupet-först-sökning (DFS)



stack

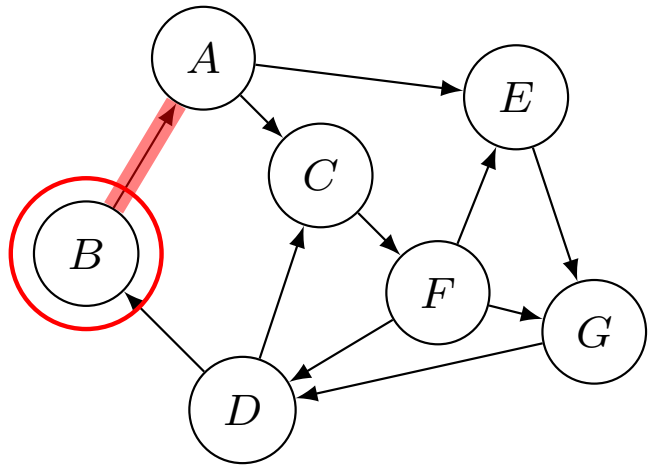


found:

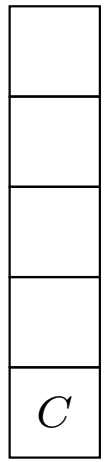
A	E	G	D	C	F	B

Grafsökning

djupet-först-sökning (DFS)



stack

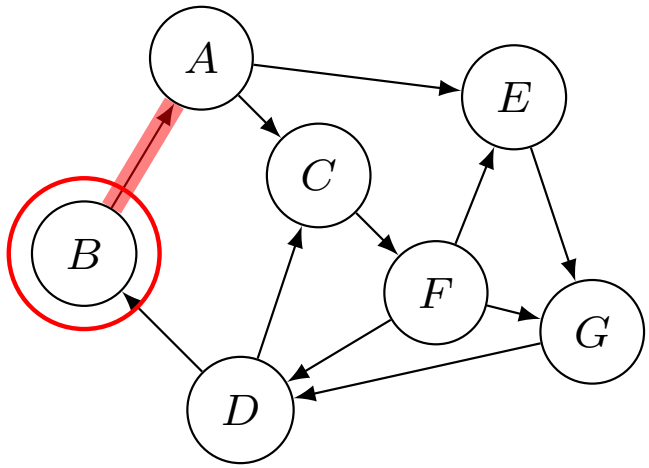


found:

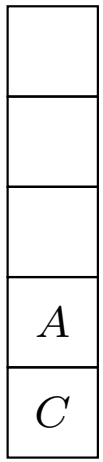
A	E	G	D	C	F	B

Grafsökning

djupet-först-sökning (DFS)



stack

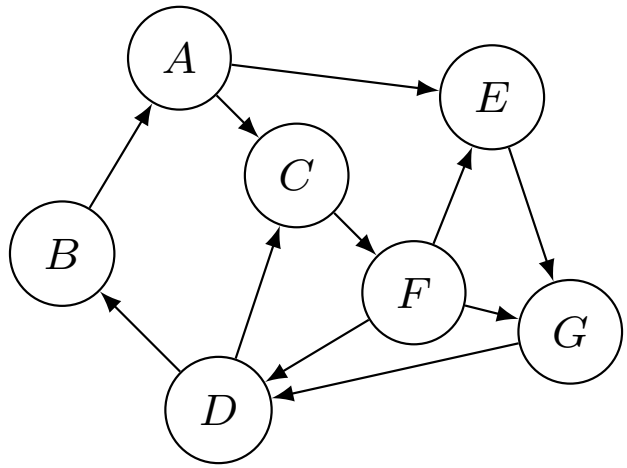


found:

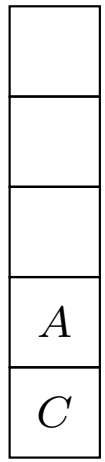
A	E	G	D	C	F	B

Grafsökning

djupet-först-sökning (DFS)



stack

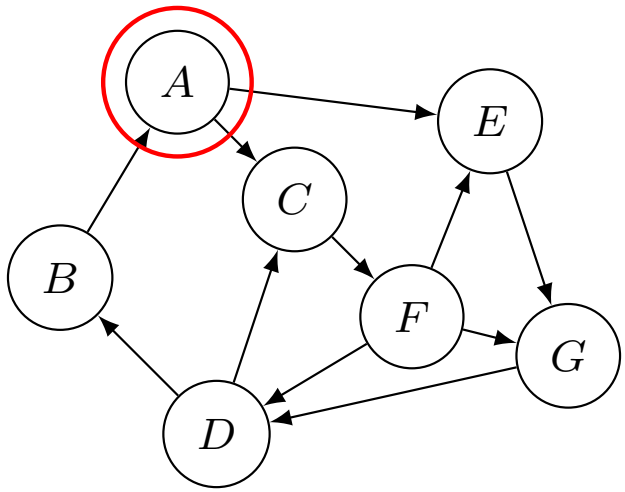


found:

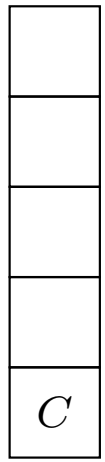
A	E	G	D	C	F	B

Grafsökning

djupet-först-sökning (DFS)



stack

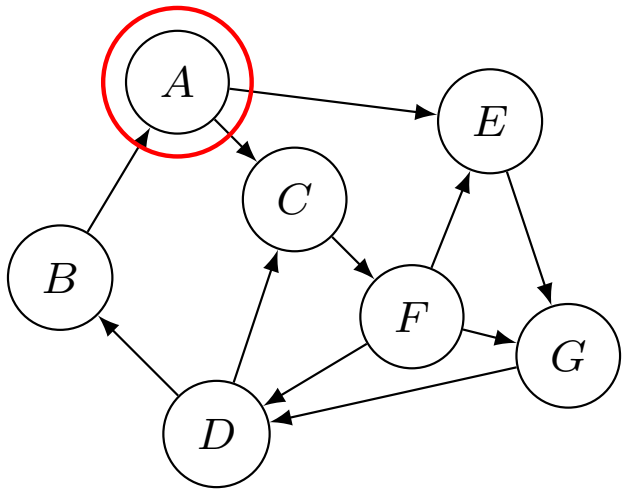


found:

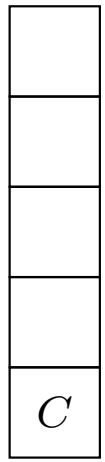
A	E	G	D	C	F	B

Grafsökning

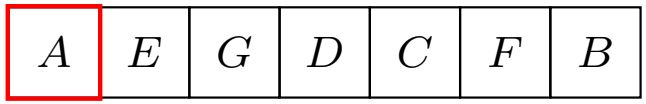
djupet-först-sökning (DFS)



stack

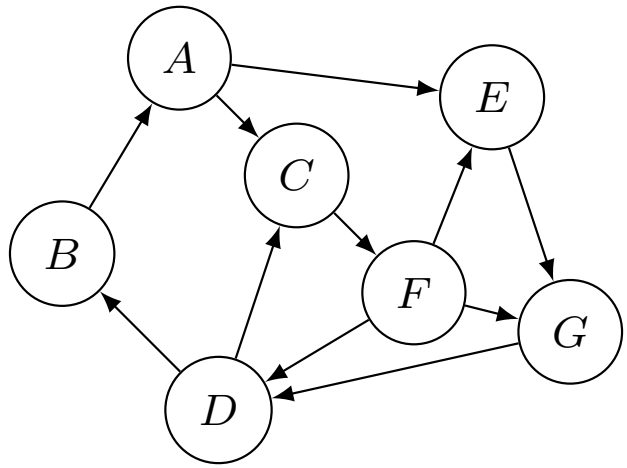


found:

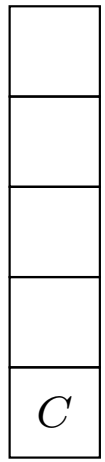


Grafsökning

djupet-först-sökning (DFS)



stack

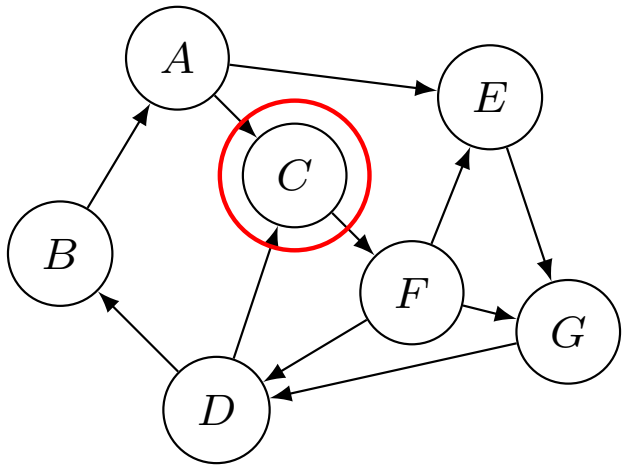


found:

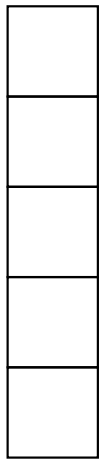
A	E	G	D	C	F	B

Grafsökning

djupet-först-sökning (DFS)



stack

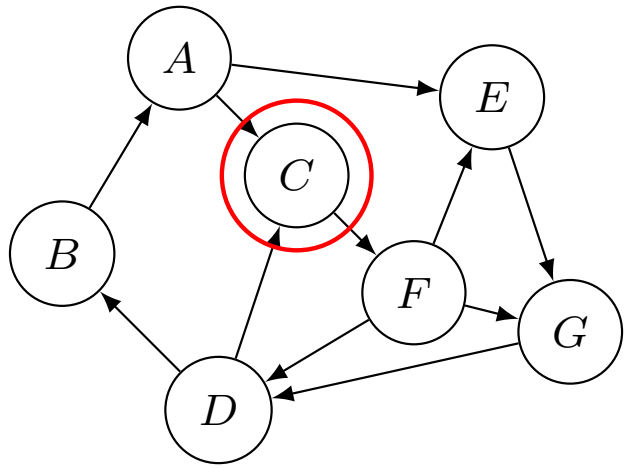


found:

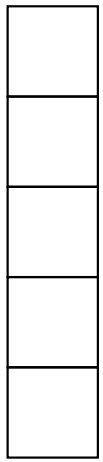
A	E	G	D	C	F	B

Grafsökning

djupet-först-sökning (DFS)



stack

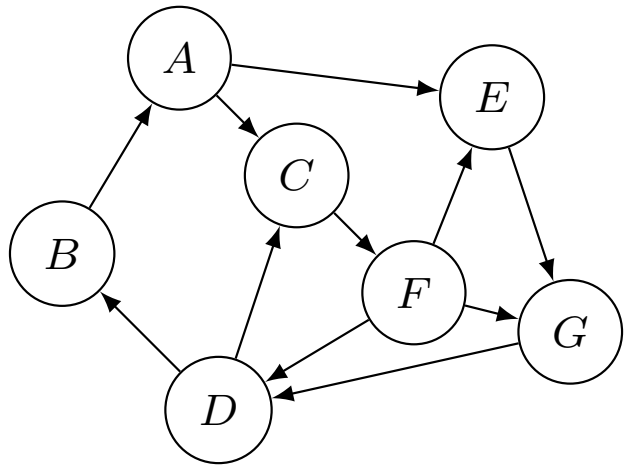


found:

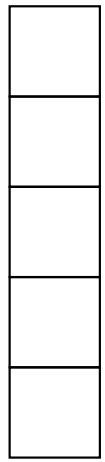
A	E	G	D	C	F	B

Grafsökning

djupet-först-sökning (DFS)



stack



found:

A	E	G	D	C	F	B

Grafsökning

Kan vi göra i någon annan ordning?

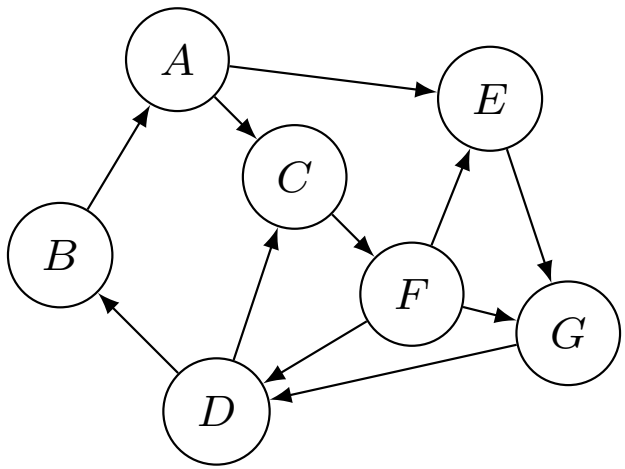
Grafsökning

Byt ut stacken mot en kö!

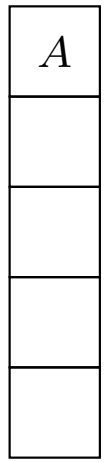
```
1  set<Node*> DFS(Node* start)
2  {
3    set<Node*> found;
4    queue<Node*> queue;
5    queue.push(start);
6    while (!queue.empty())
7    {
8      Node* node { queue.front() };
9      queue.pop();
10     if (node == nullptr) continue;
11     if (found.count(node) > 0) continue;
12     found.insert(node);
13     for (Node* next : neighbours)
14       queue.push(next);
15   }
16   return found;
17 }
```

Grafsökning

bredden-först-sökning (BFS)



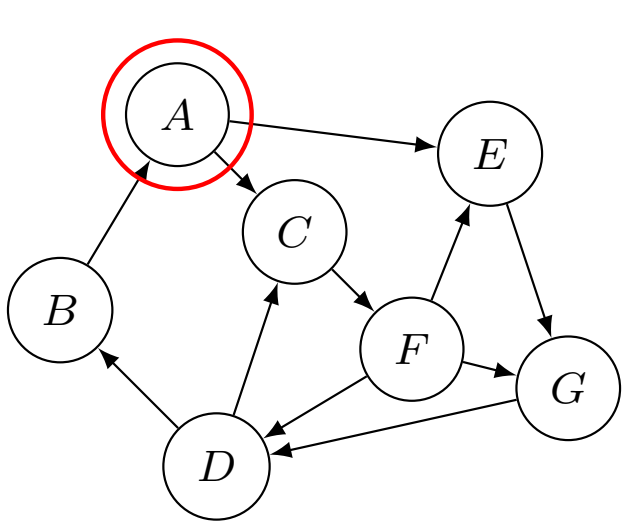
queue



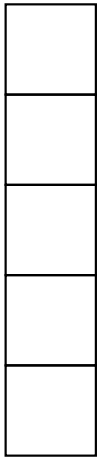
found:

Grafsökning

bredden-först-sökning (BFS)



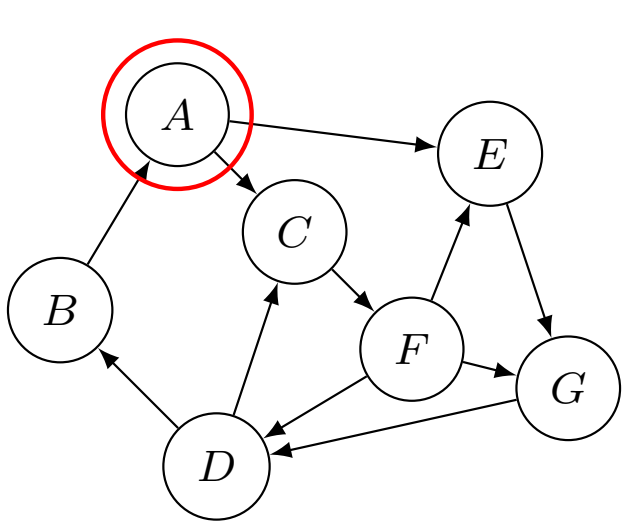
queue



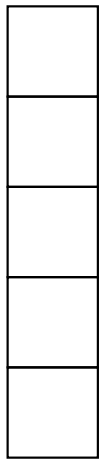
found:

Grafsökning

bredden-först-sökning (BFS)



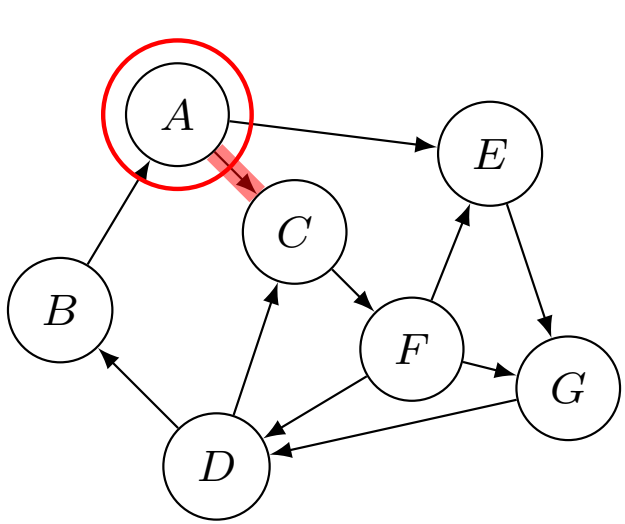
queue



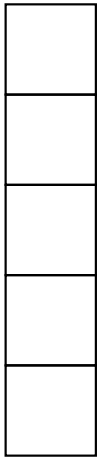
found: A

Grafsökning

bredden-först-sökning (BFS)



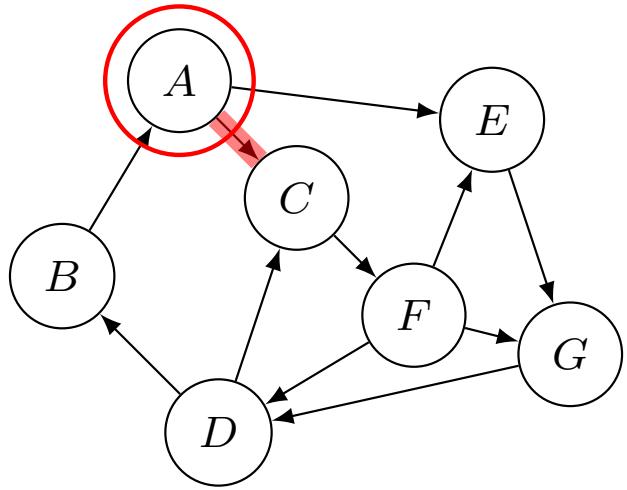
queue



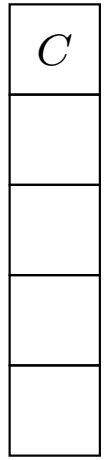
found: A

Grafsökning

bredden-först-sökning (BFS)



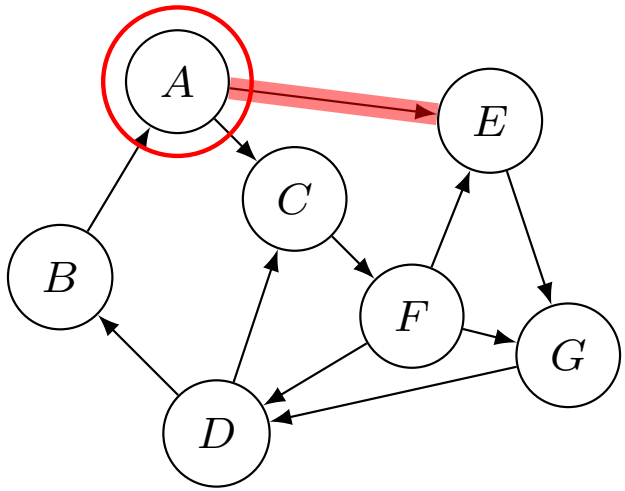
queue



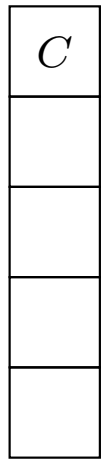
found:

Grafsökning

bredden-först-sökning (BFS)



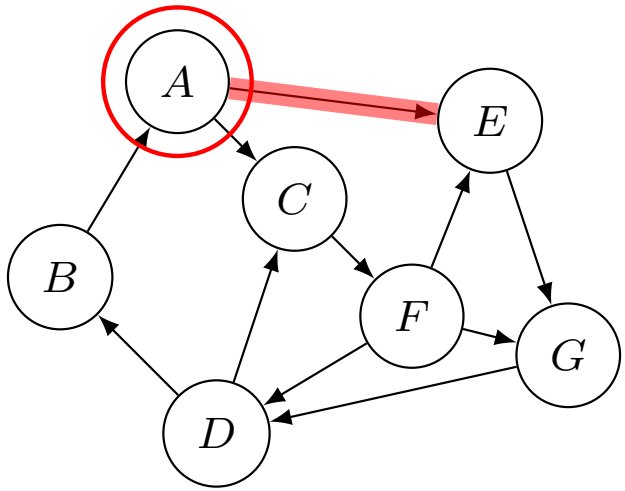
queue



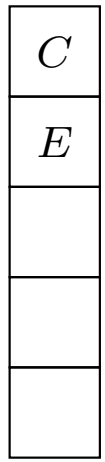
found:

Grafsökning

bredden-först-sökning (BFS)



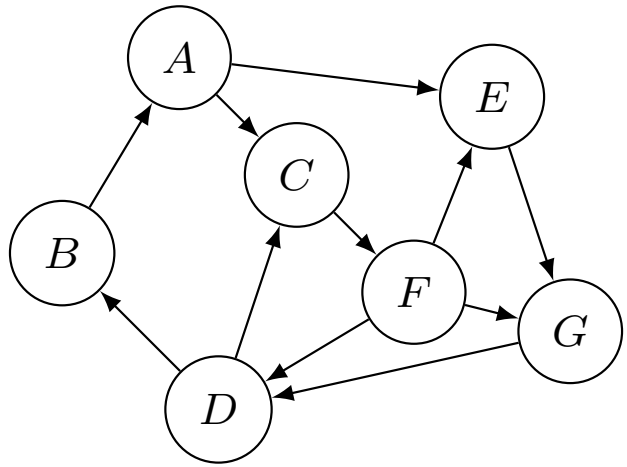
queue



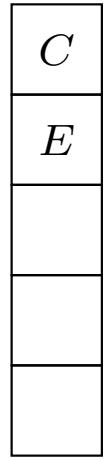
found:

Grafsökning

bredden-först-sökning (BFS)



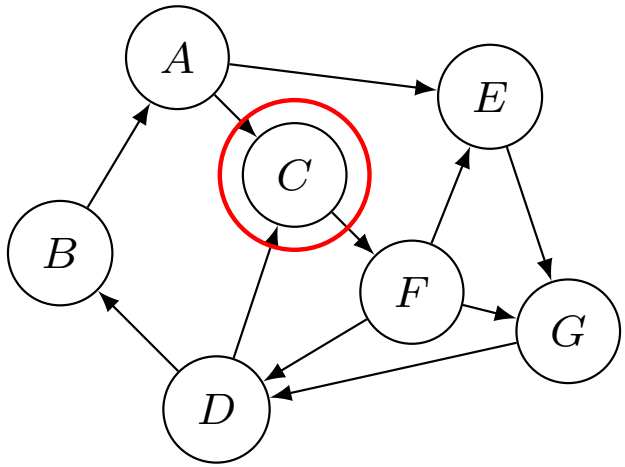
queue



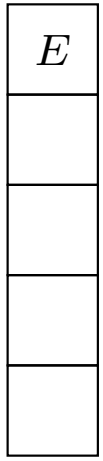
found:

Grafsökning

bredden-först-sökning (BFS)



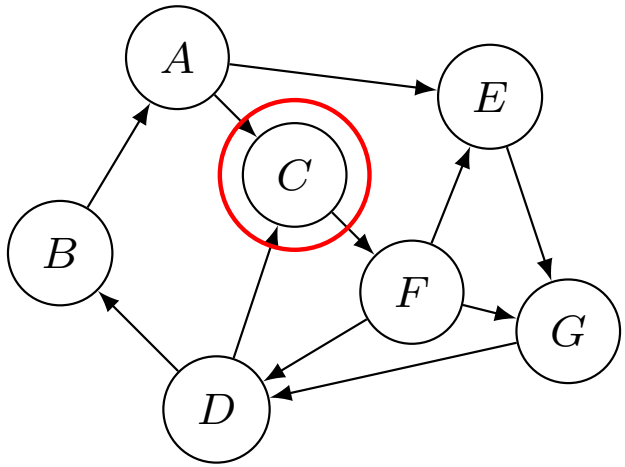
queue



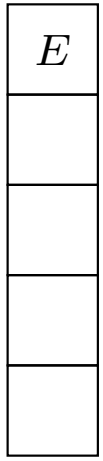
found:

Grafsökning

bredden-först-sökning (BFS)



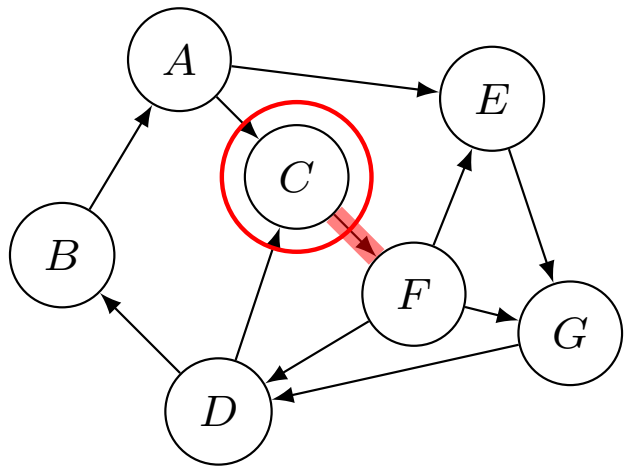
queue



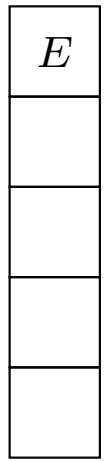
found:

Grafsökning

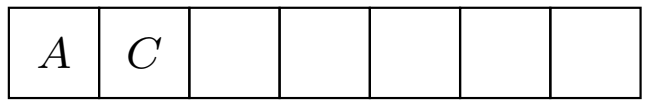
bredden-först-sökning (BFS)



queue

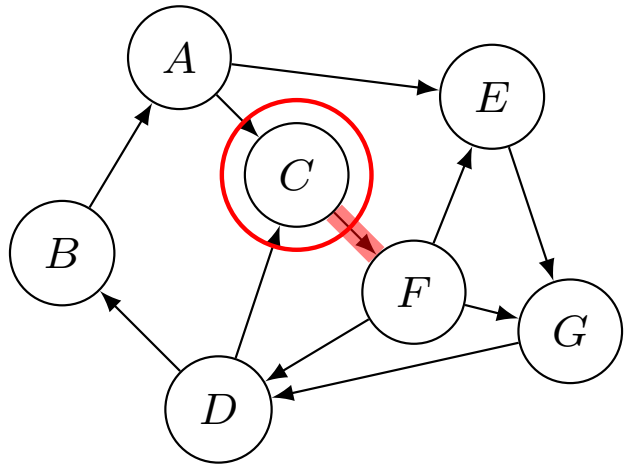


found:

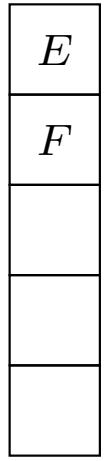


Grafsökning

bredden-först-sökning (BFS)



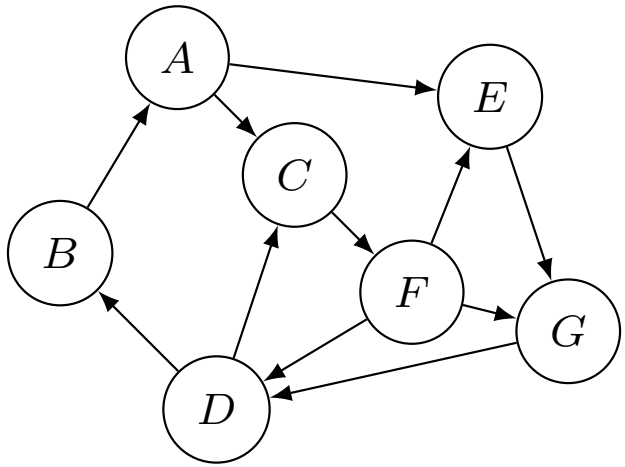
queue



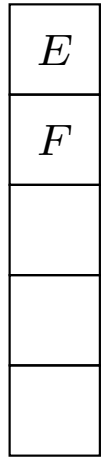
found:

Grafsökning

bredden-först-sökning (BFS)



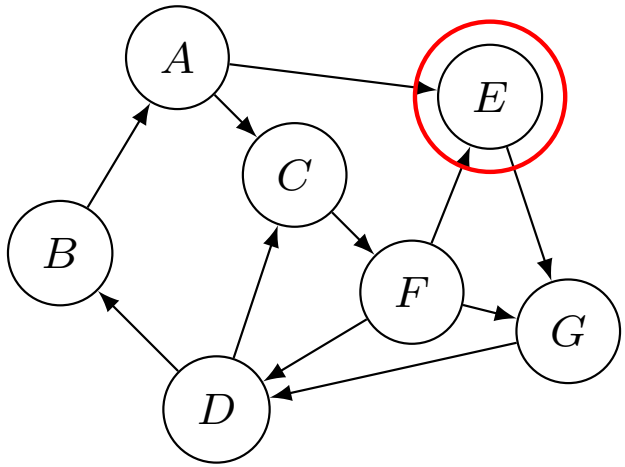
queue



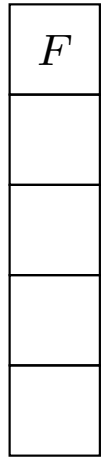
found:

Grafsökning

bredden-först-sökning (BFS)



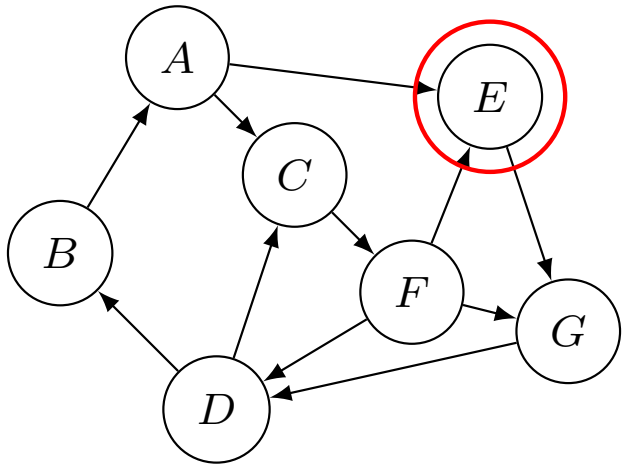
queue



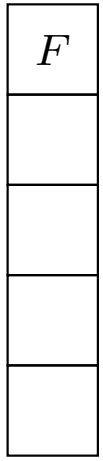
found:

Grafsökning

bredden-först-sökning (BFS)



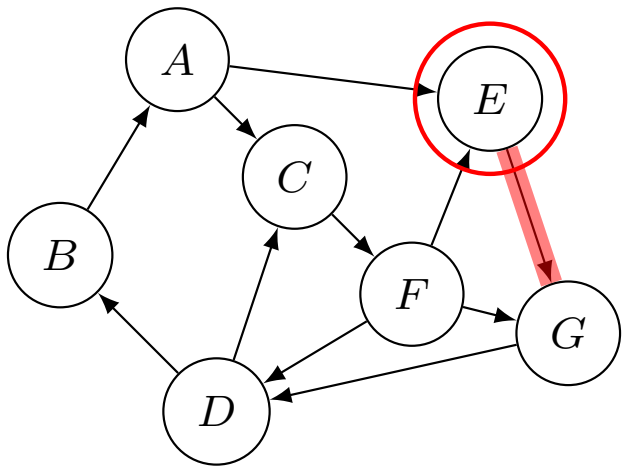
queue



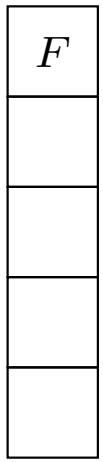
found:

Grafsökning

bredden-först-sökning (BFS)



queue

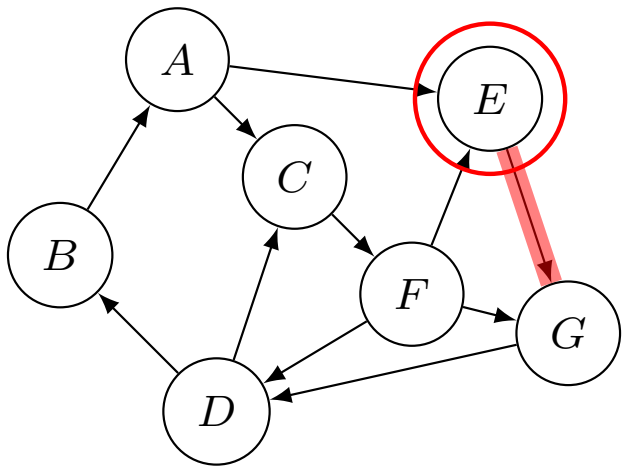


found:

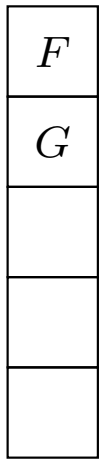
A	C	E				

Grafsökning

bredden-först-sökning (BFS)



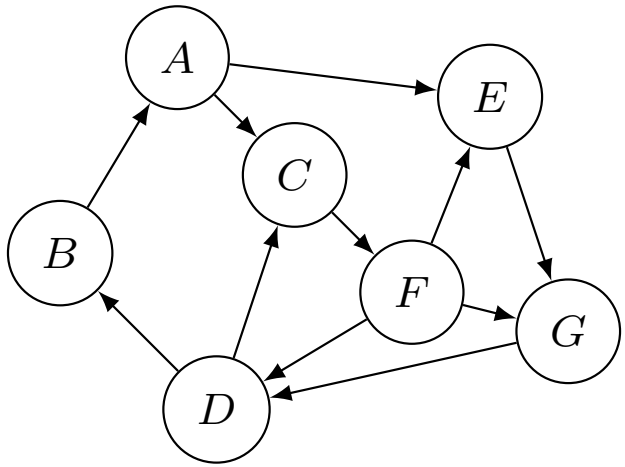
queue



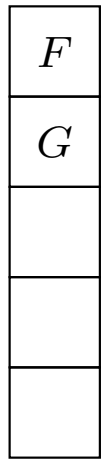
found:

Grafsökning

bredden-först-sökning (BFS)



queue

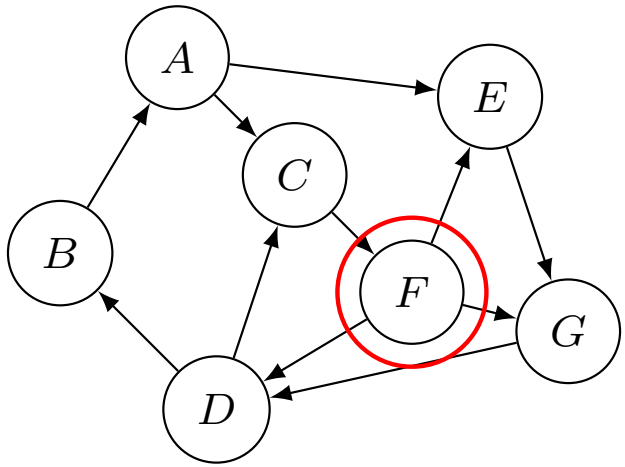


found:

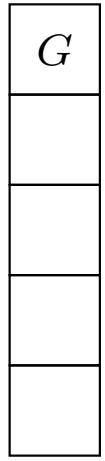
A	C	E				

Grafsökning

bredden-först-sökning (BFS)



queue

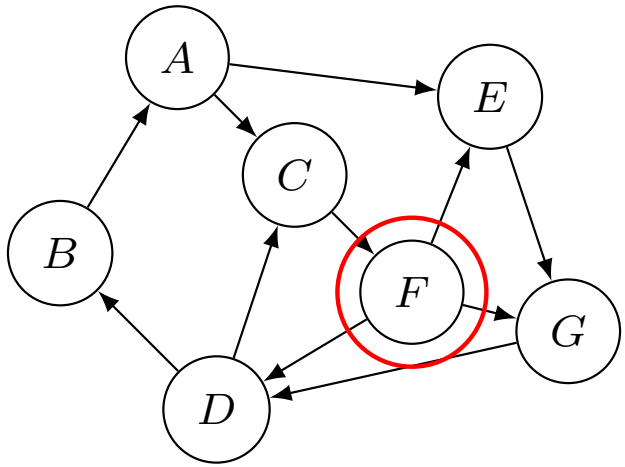


found:

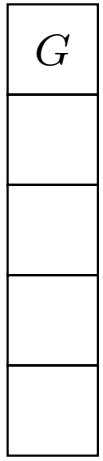
A	C	E				

Grafsökning

bredden-först-sökning (BFS)



queue

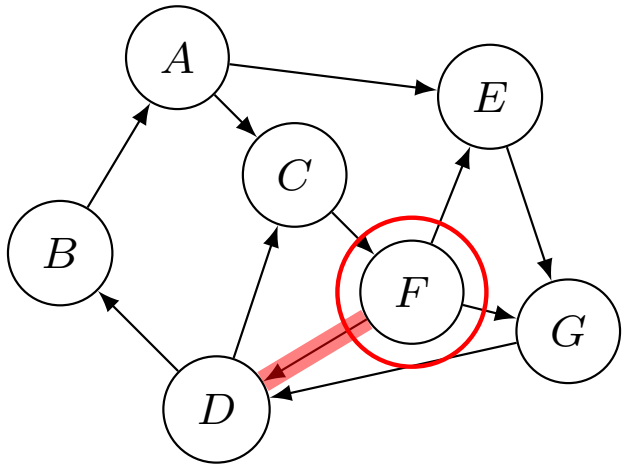


found:

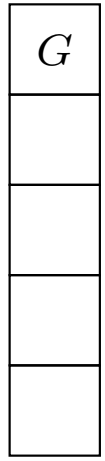
A	C	E	F			

Grafsökning

bredden-först-sökning (BFS)



queue

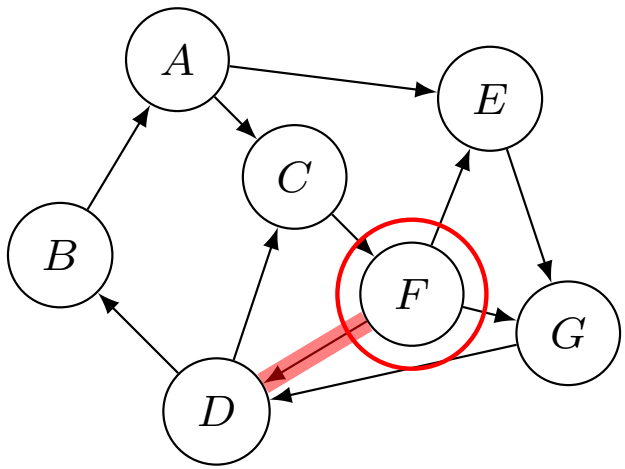


found:

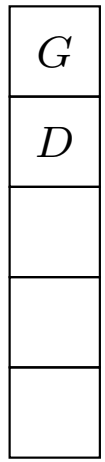
A	C	E	F			

Grafsökning

bredden-först-sökning (BFS)



queue

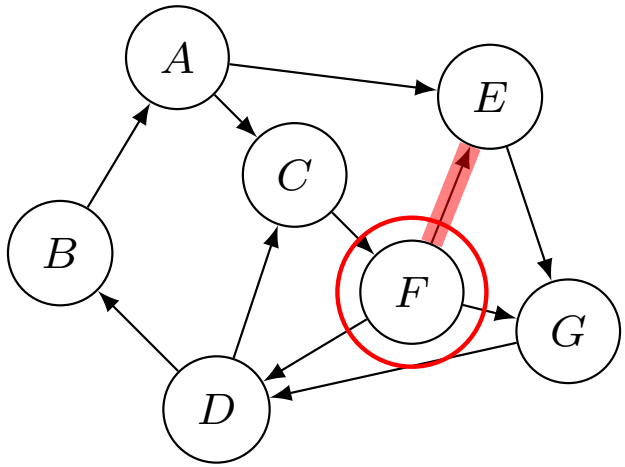


found:

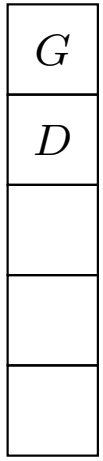
A	C	E	F			

Grafsökning

bredden-först-sökning (BFS)



queue

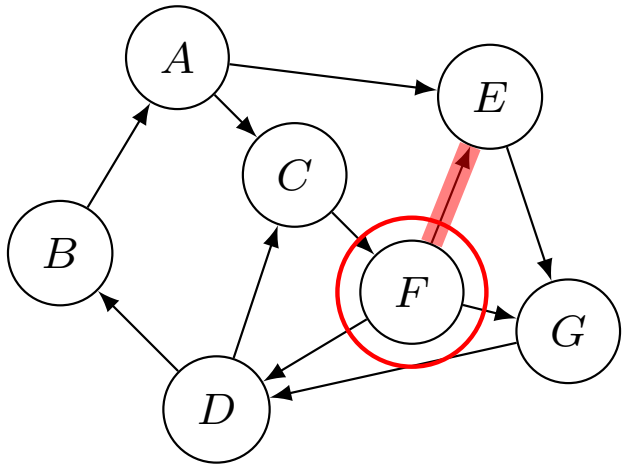


found:

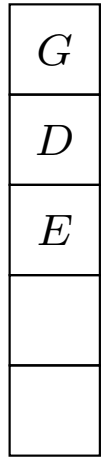
A	C	E	F			

Grafsökning

bredden-först-sökning (BFS)



queue

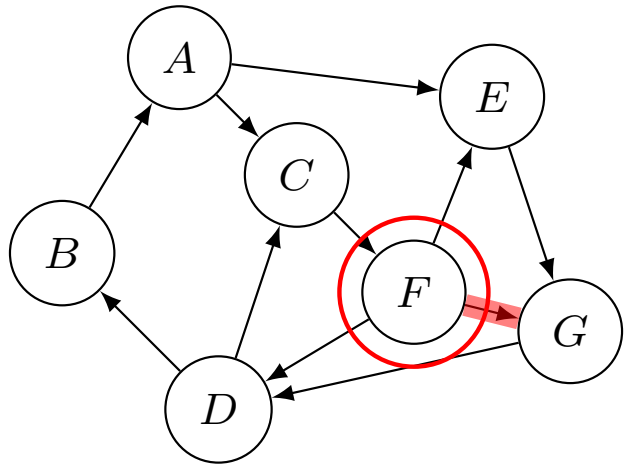


found:

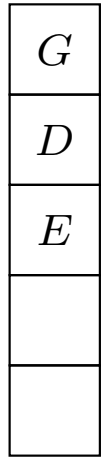
A	C	E	F			

Grafsökning

bredden-först-sökning (BFS)



queue

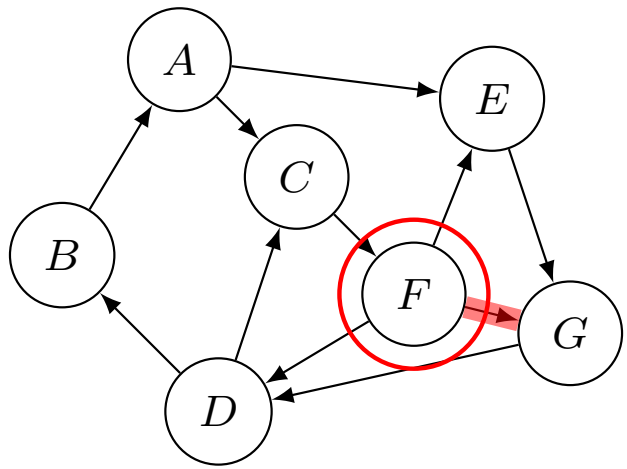


found:

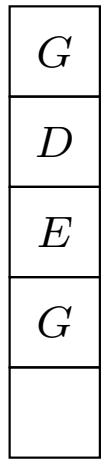
A	C	E	F			

Grafsökning

bredden-först-sökning (BFS)



queue

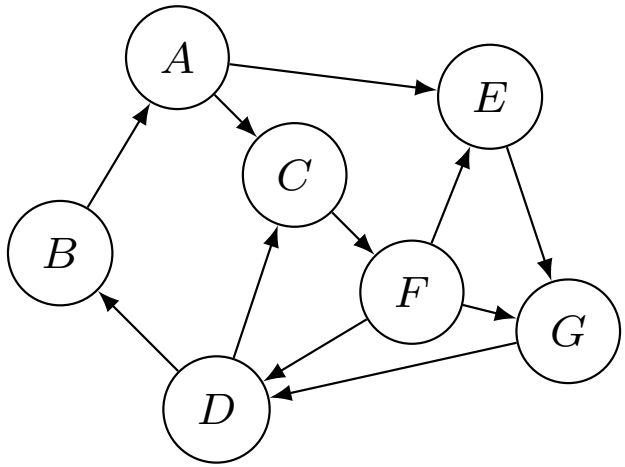


found:

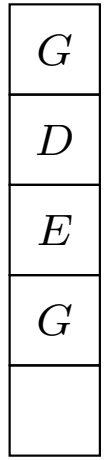
A	C	E	F			

Grafsökning

bredden-först-sökning (BFS)



queue

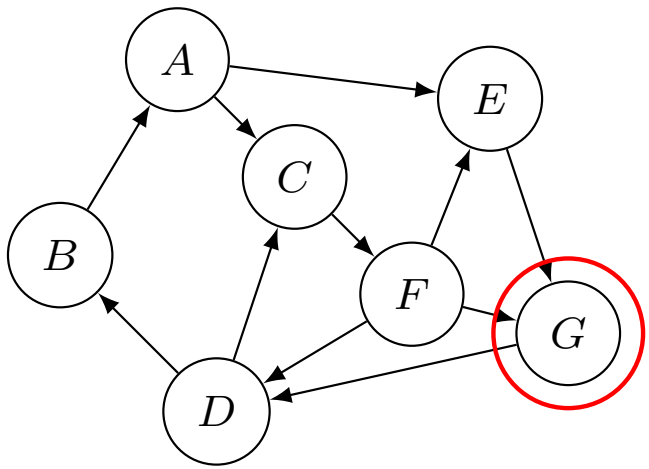


found:

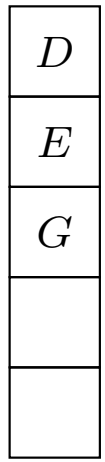
A	C	E	F			

Grafsökning

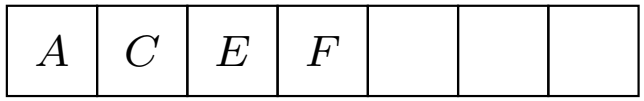
bredden-först-sökning (BFS)



queue

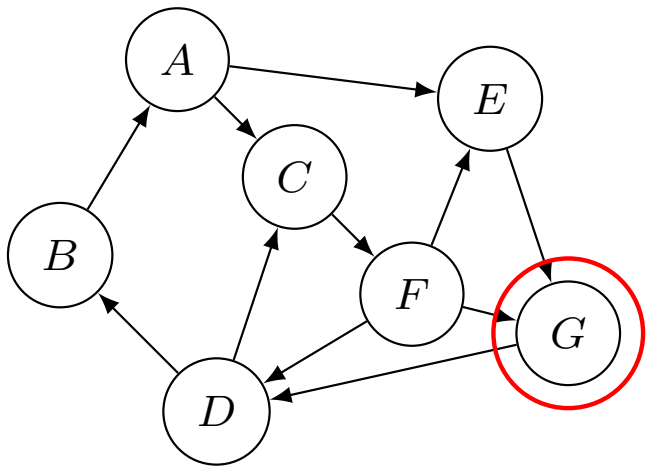


found:



Grafsökning

bredden-först-sökning (BFS)



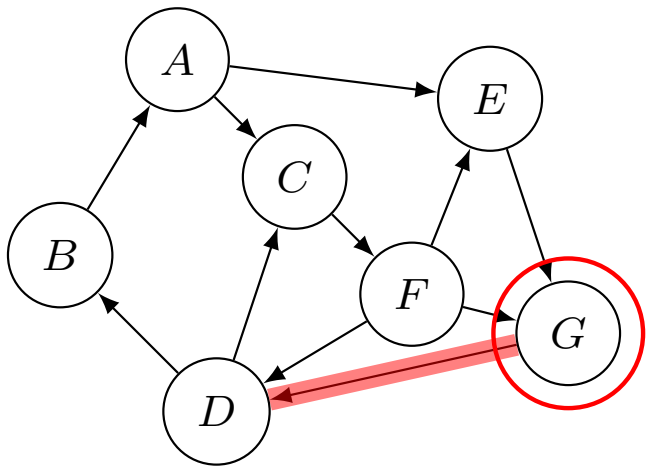
queue

found:

A	C	E	F	G		

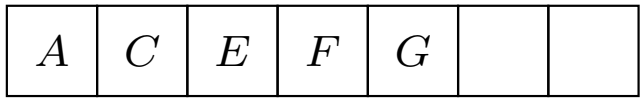
Grafsökning

bredden-först-sökning (BFS)



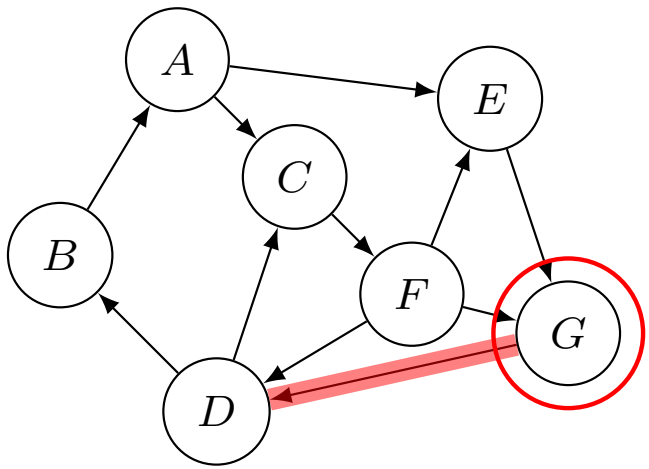
queue

found:

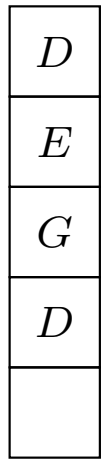


Grafsökning

bredden-först-sökning (BFS)



queue

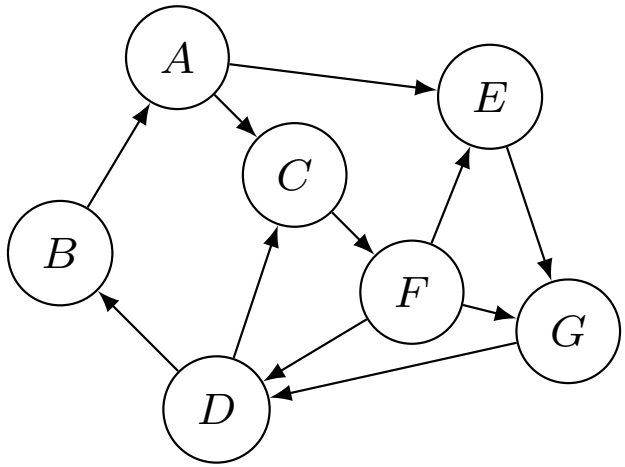


found:

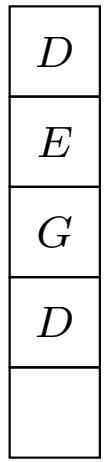
A	C	E	F	G		

Grafsökning

bredden-först-sökning (BFS)



queue

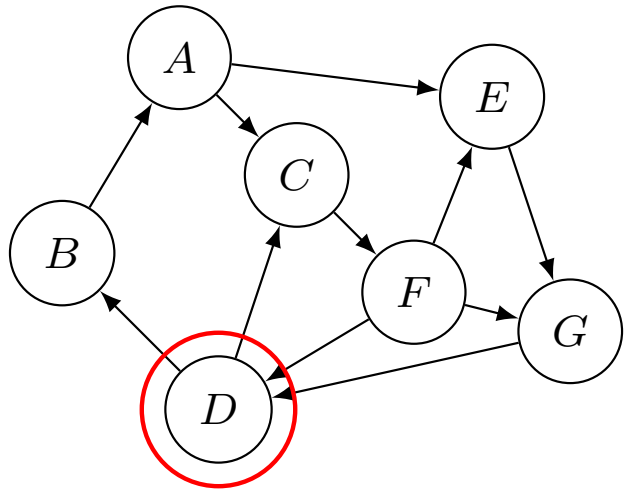


found:

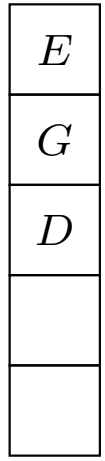
A	C	E	F	G		

Grafsökning

bredden-först-sökning (BFS)



queue

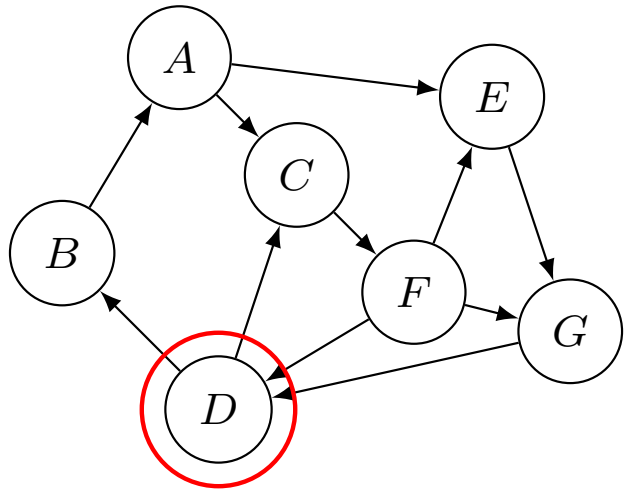


found:

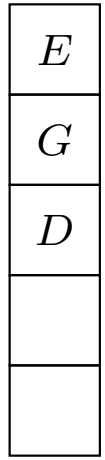
A	C	E	F	G		

Grafsökning

bredden-först-sökning (BFS)



queue

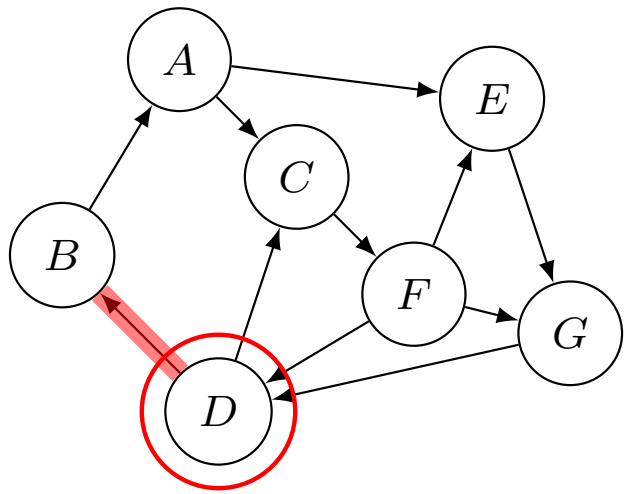


found:

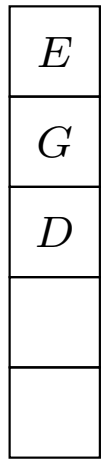
A	C	E	F	G	D	

Grafsökning

bredden-först-sökning (BFS)



queue

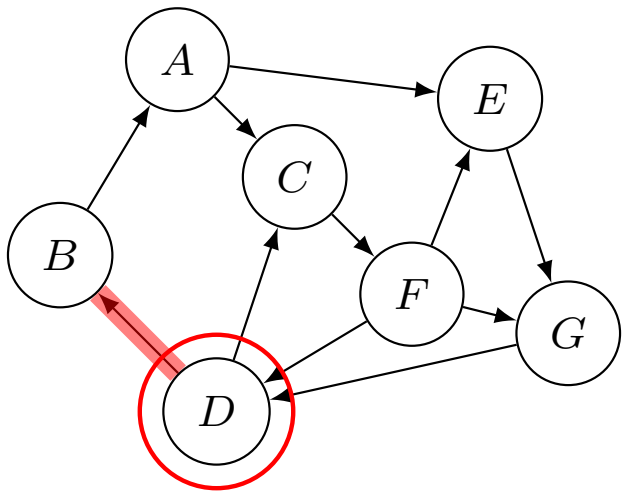


found:

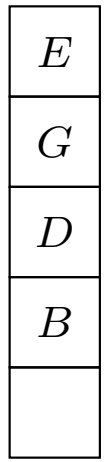
A	C	E	F	G	D	

Grafsökning

bredden-först-sökning (BFS)



queue

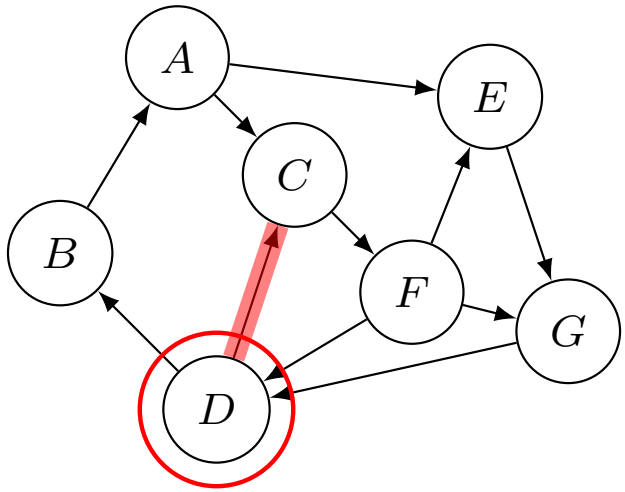


found:

A	C	E	F	G	D	

Grafsökning

bredden-först-sökning (BFS)



queue

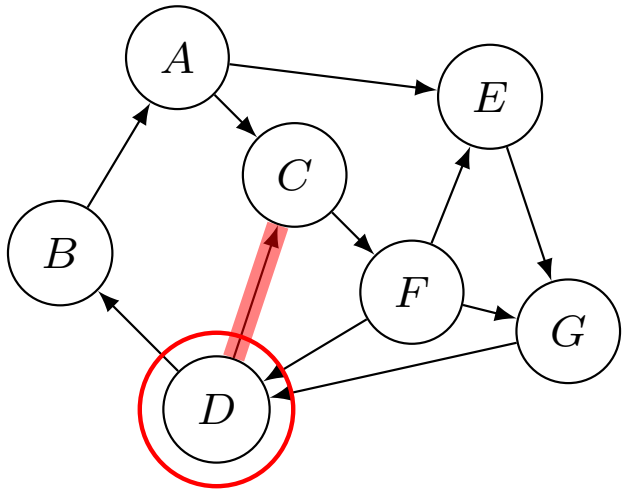
E
G
D
B

found:

A	C	E	F	G	D	

Grafsökning

bredden-först-sökning (BFS)



queue

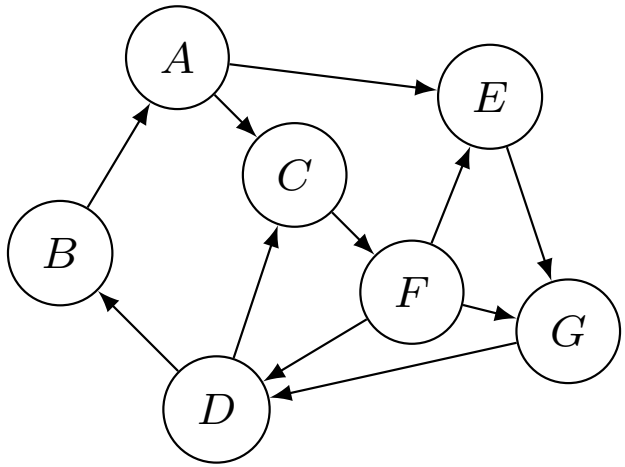
E
G
D
B
C

`found:`

A	C	E	F	G	D	

Grafsökning

bredden-först-sökning (BFS)



queue

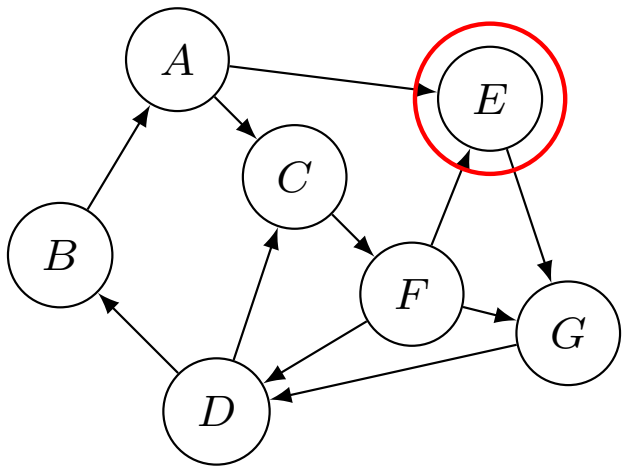
E
G
D
B
C

found:

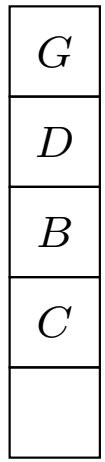
A	C	E	F	G	D	

Grafsökning

bredden-först-sökning (BFS)



queue

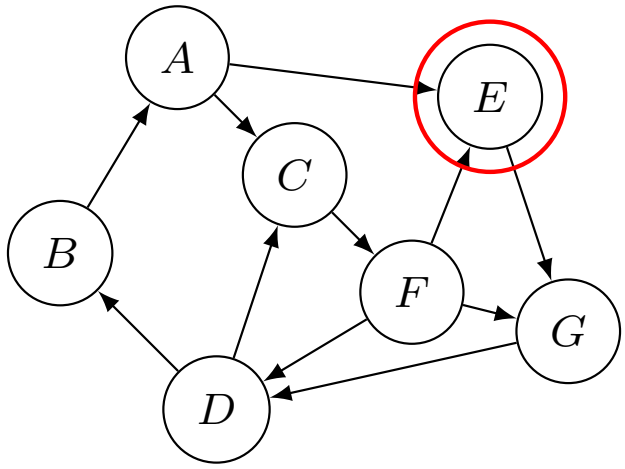


found:

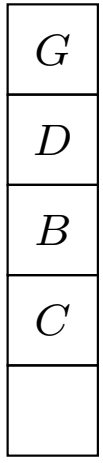
A	C	E	F	G	D	

Grafsökning

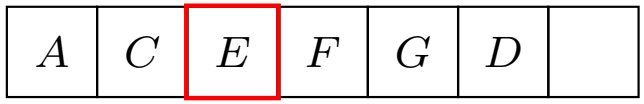
bredden-först-sökning (BFS)



queue

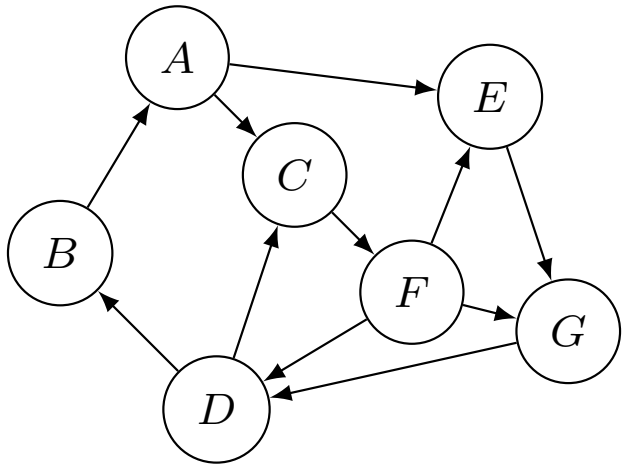


found:



Grafsökning

bredden-först-sökning (BFS)



queue

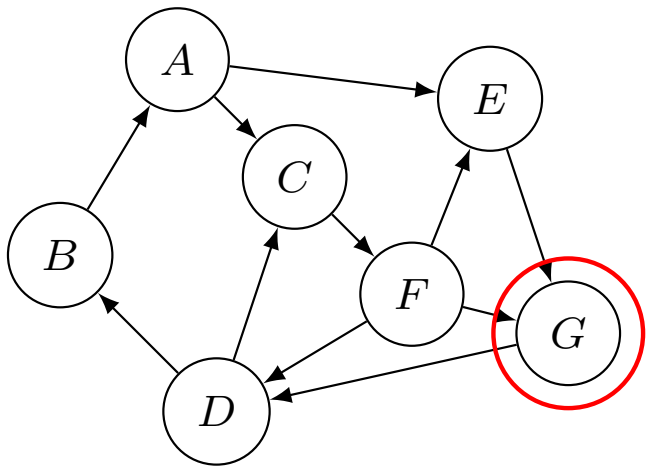
G
D
B
C

found:

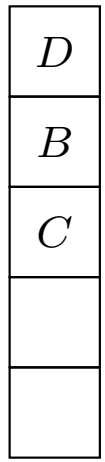
A	C	E	F	G	D	

Grafsökning

bredden-först-sökning (BFS)



queue

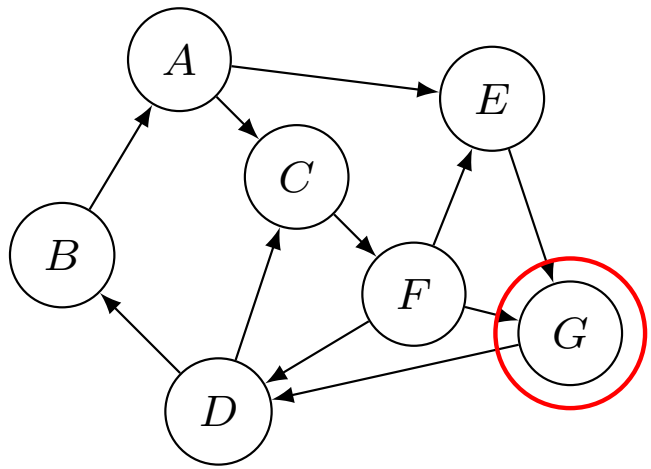


found:

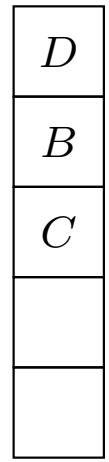
A	C	E	F	G	D	

Grafsökning

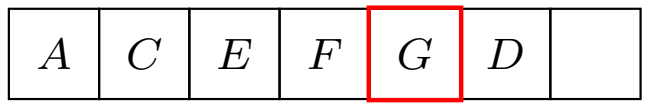
bredden-först-sökning (BFS)



queue

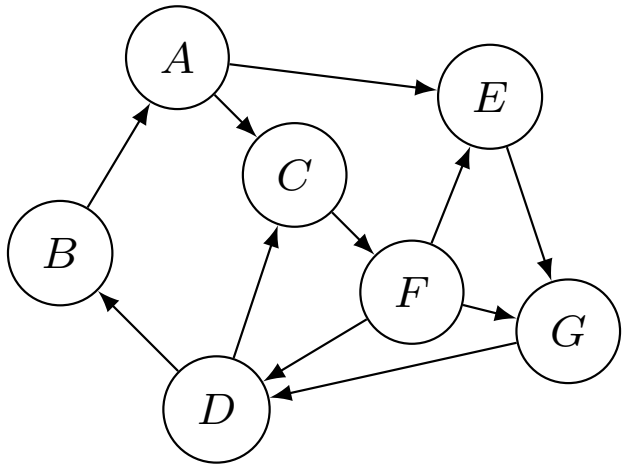


found:

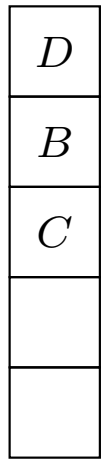


Grafsökning

bredden-först-sökning (BFS)



queue

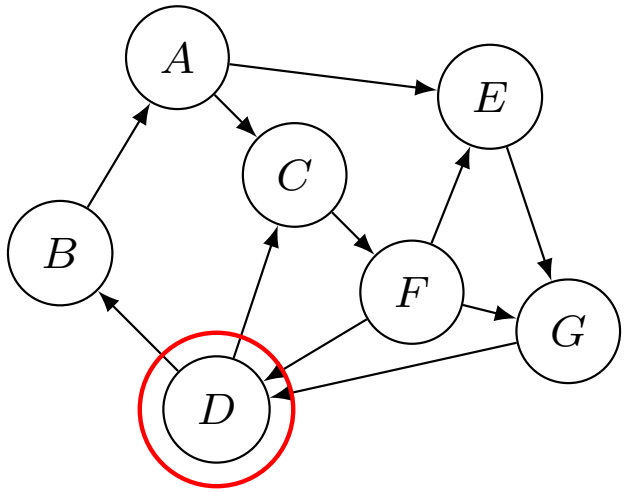


found:

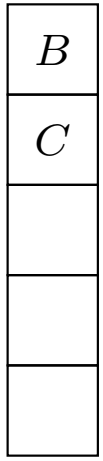
A	C	E	F	G	D	

Grafsökning

bredden-först-sökning (BFS)



queue

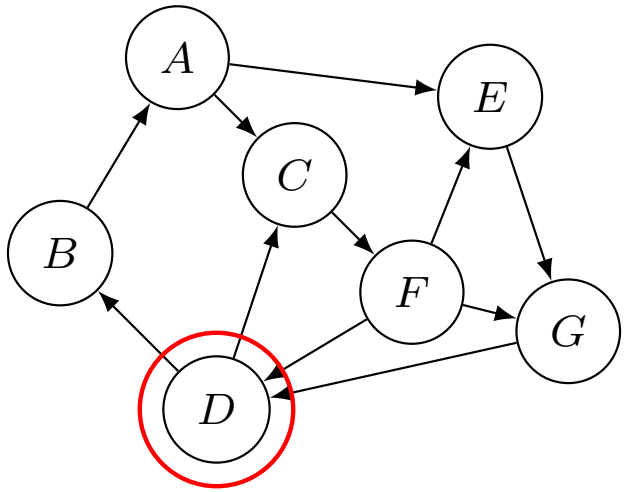


found:

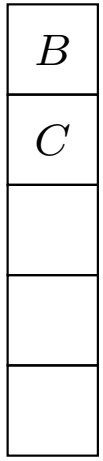
A	C	E	F	G	D	

Grafsökning

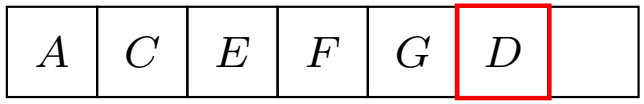
bredden-först-sökning (BFS)



queue

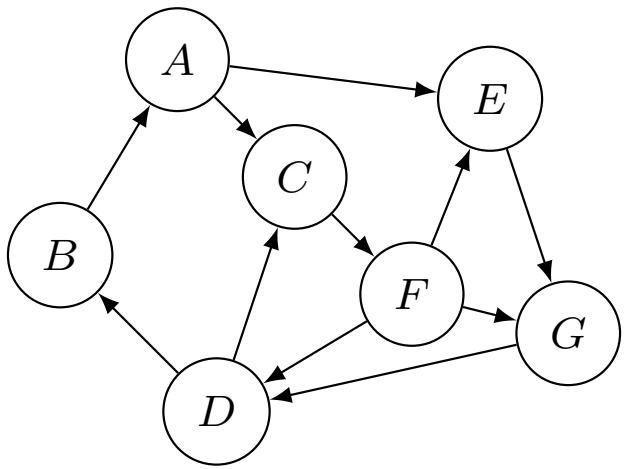


found:

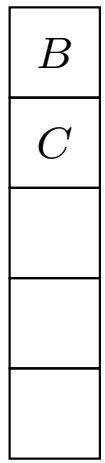


Grafsökning

bredden-först-sökning (BFS)



queue

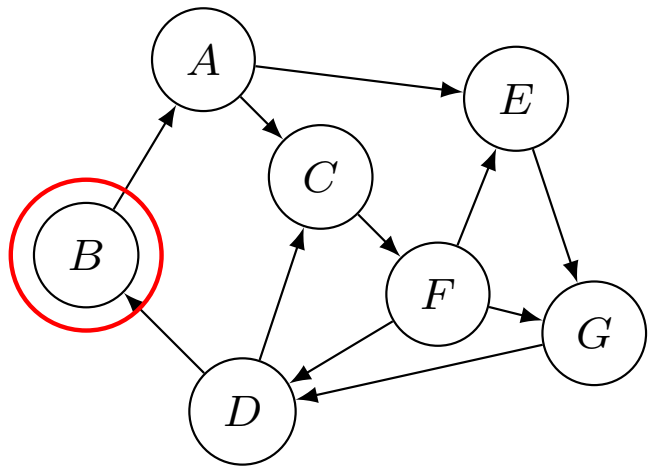


found:

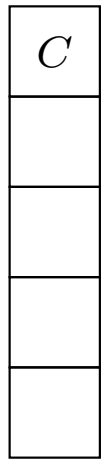
A	C	E	F	G	D	

Grafsökning

bredden-först-sökning (BFS)



queue

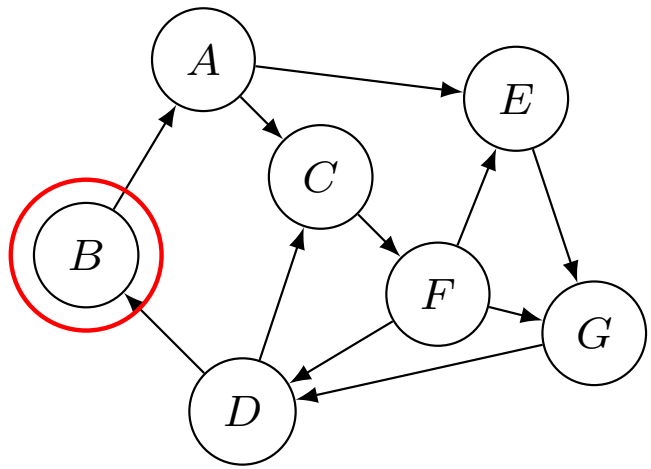


found:

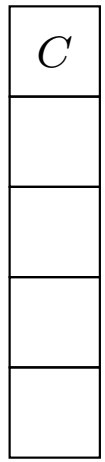
A	C	E	F	G	D	

Grafsökning

bredden-först-sökning (BFS)



queue

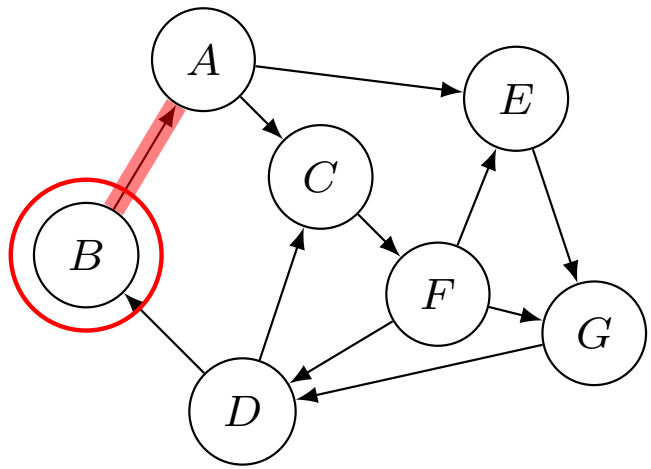


found:

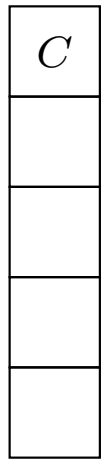
A	C	E	F	G	D	B

Grafsökning

bredden-först-sökning (BFS)



queue

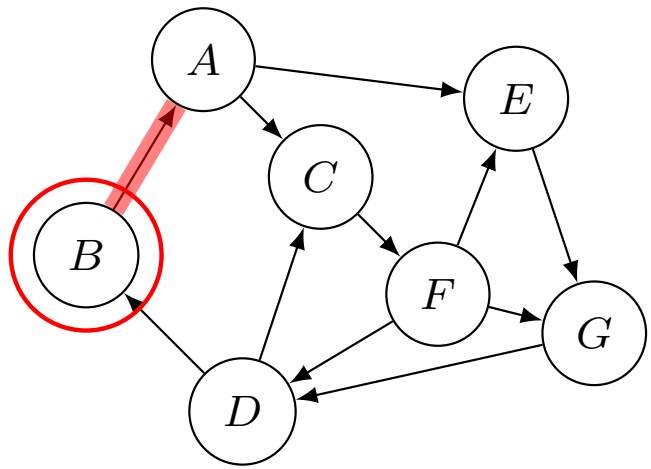


found:

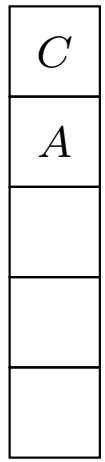
A	C	E	F	G	D	B

Grafsökning

bredden-först-sökning (BFS)



queue

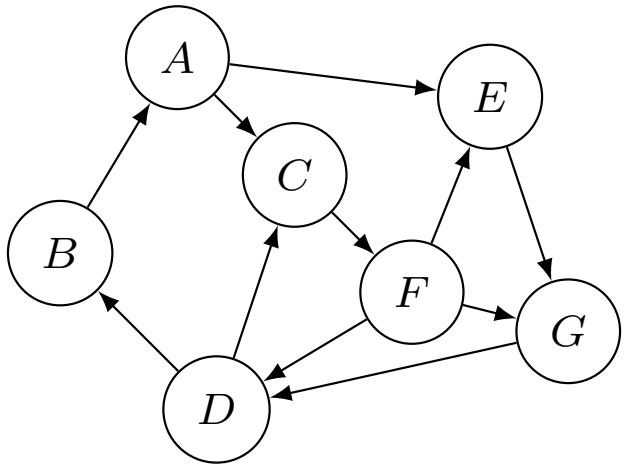


found:

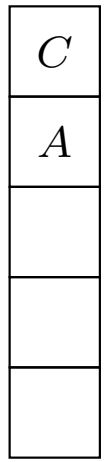
A	C	E	F	G	D	B

Grafsökning

bredden-först-sökning (BFS)



queue

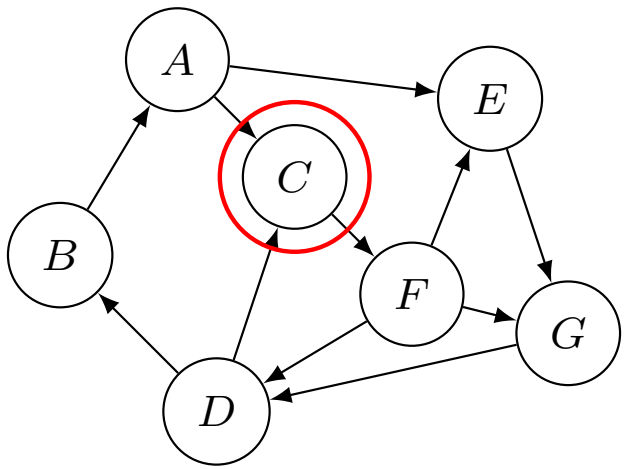


found:

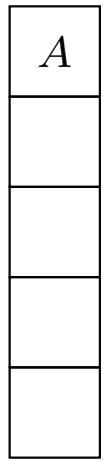
A	C	E	F	G	D	B

Grafsökning

bredden-först-sökning (BFS)



queue

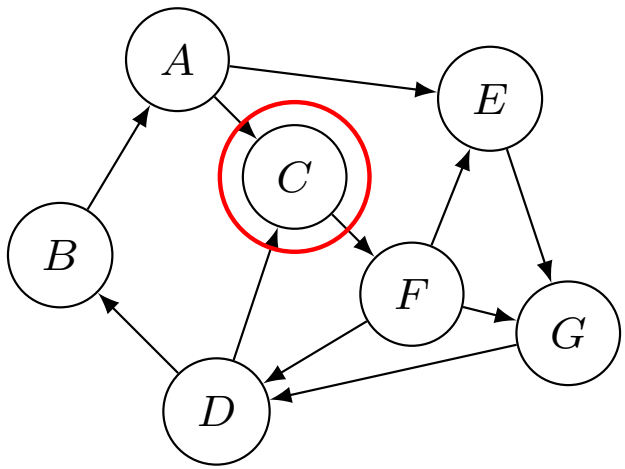


found:

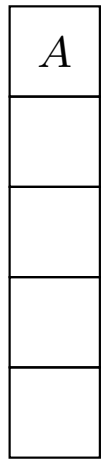
A	C	E	F	G	D	B

Grafsökning

bredden-först-sökning (BFS)



queue

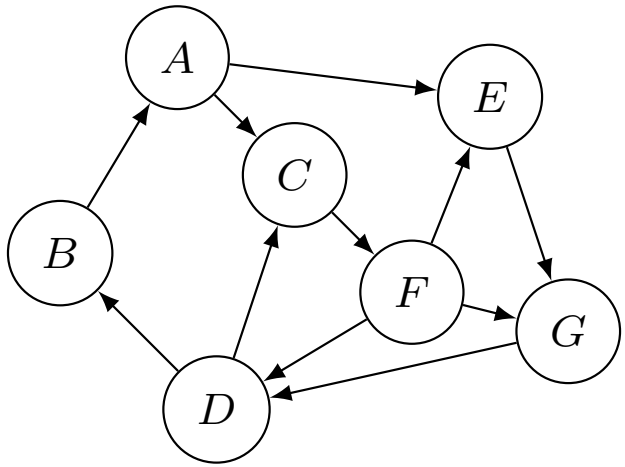


found:

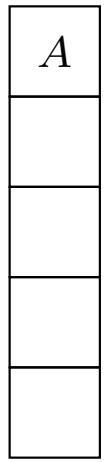
A	C	E	F	G	D	B

Grafsökning

bredden-först-sökning (BFS)



queue

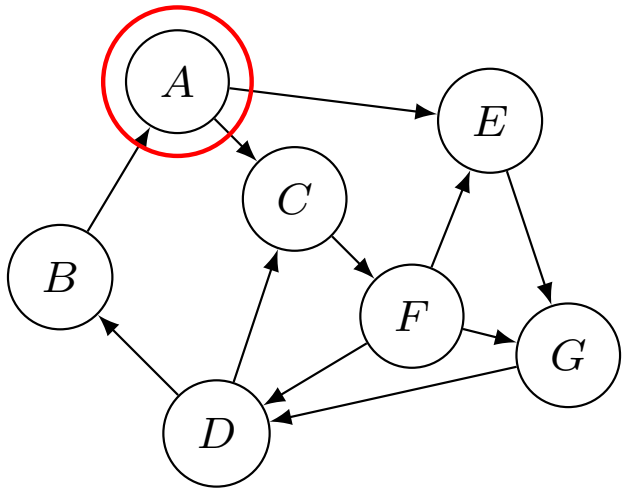


found:

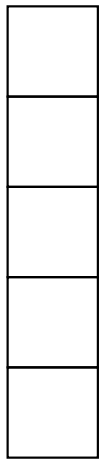
A	C	E	F	G	D	B

Grafsökning

bredden-först-sökning (BFS)



queue

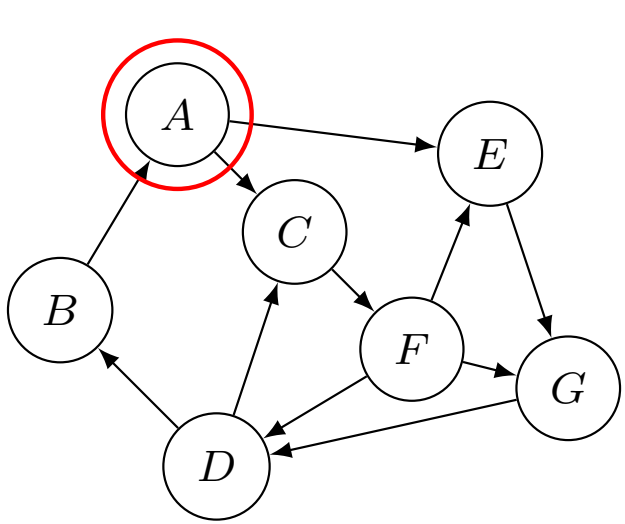


found:

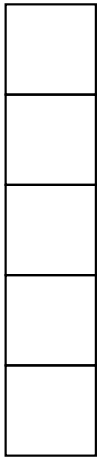
A	C	E	F	G	D	B

Grafsökning

bredden-först-sökning (BFS)



queue

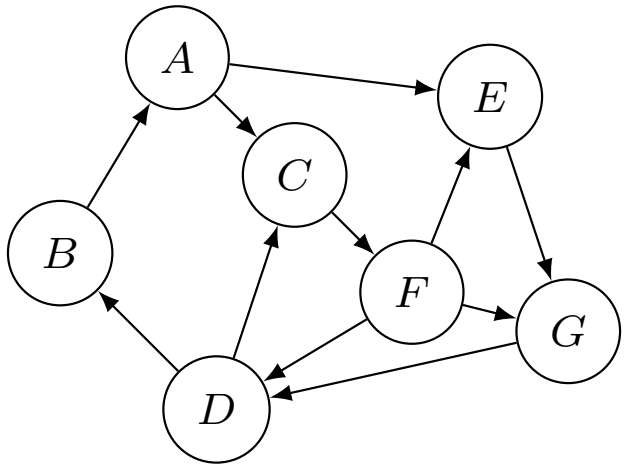


found:

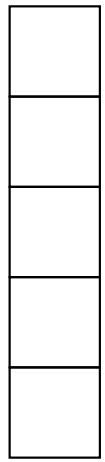
A	C	E	F	G	D	B

Grafsökning

bredden-först-sökning (BFS)



queue

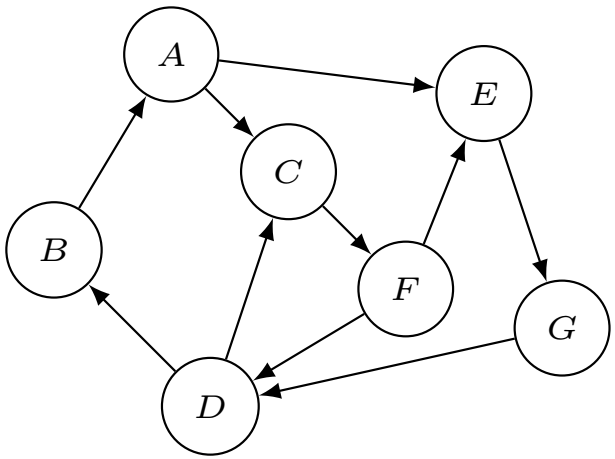


found:

A	C	E	F	G	D	B

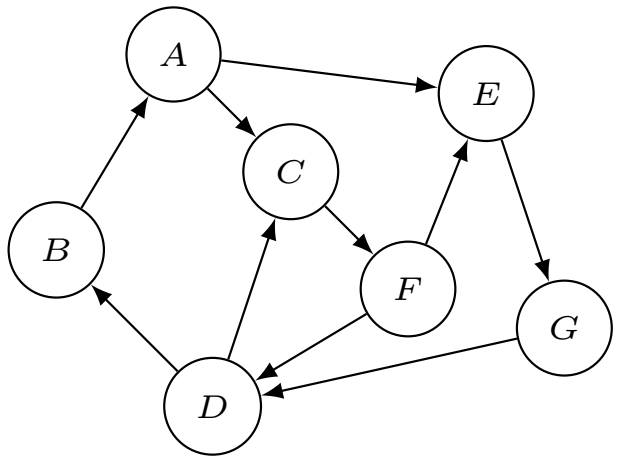
Grafsökning

Jämförelse DFS och BFS



DFS ordning

A	E	G	D	C	F	B

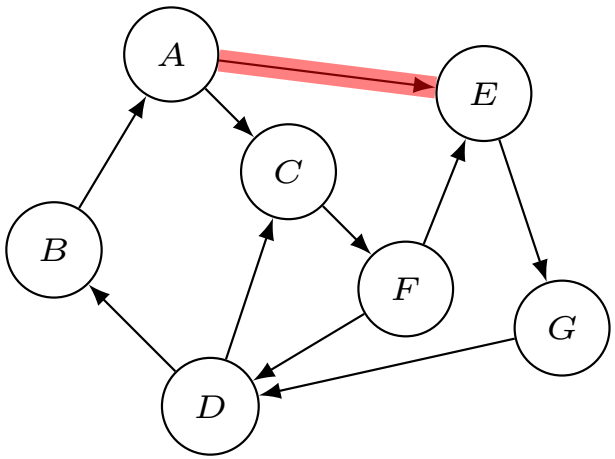


BFS ordning

A	C	E	F	G	D	B

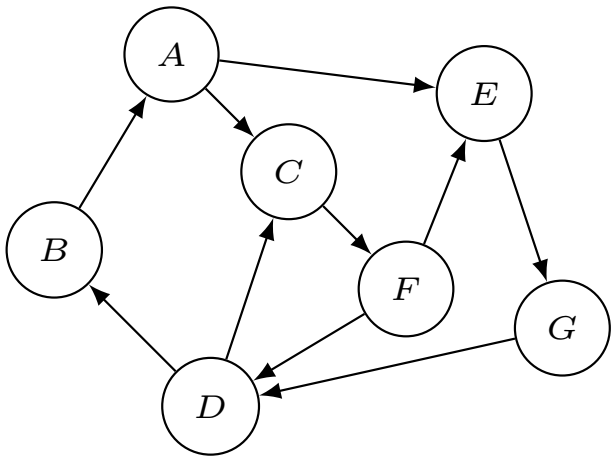
Grafsökning

Jämförelse DFS och BFS



DFS ordning

A	E	G	D	C	F	B

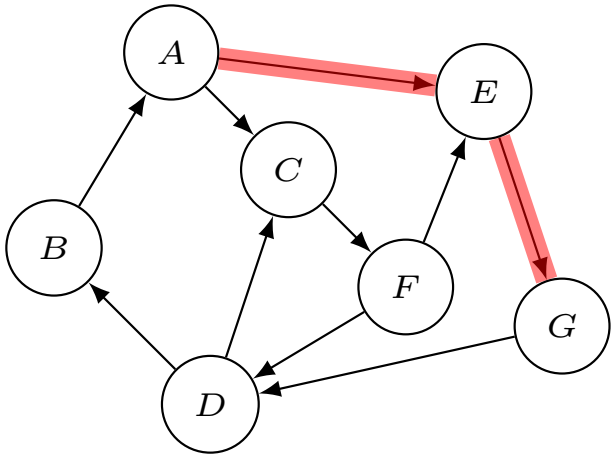


BFS ordning

A	C	E	F	G	D	B

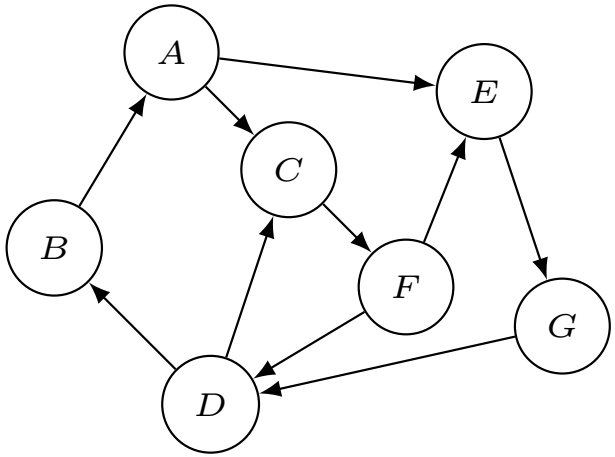
Grafsökning

Jämförelse DFS och BFS



DFS ordning

A	E	G	D	C	F	B

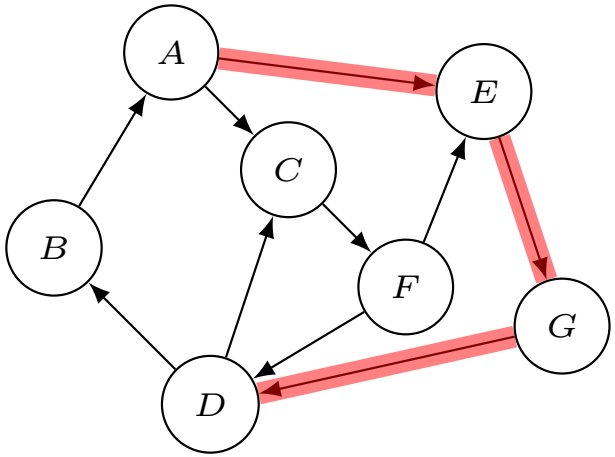


BFS ordning

A	C	E	F	G	D	B

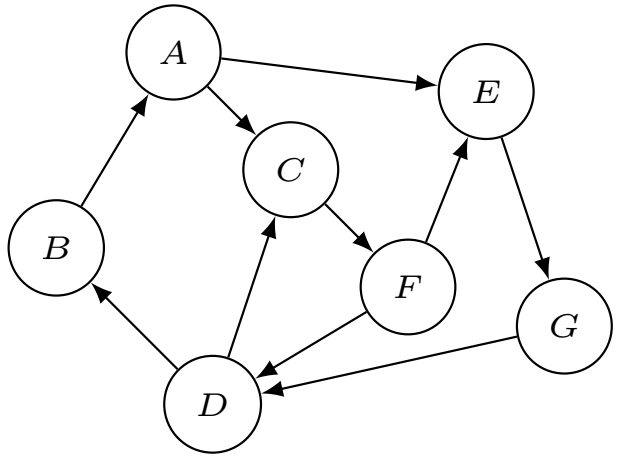
Grafsökning

Jämförelse DFS och BFS



DFS ordning

A	E	G	D	C	F	B

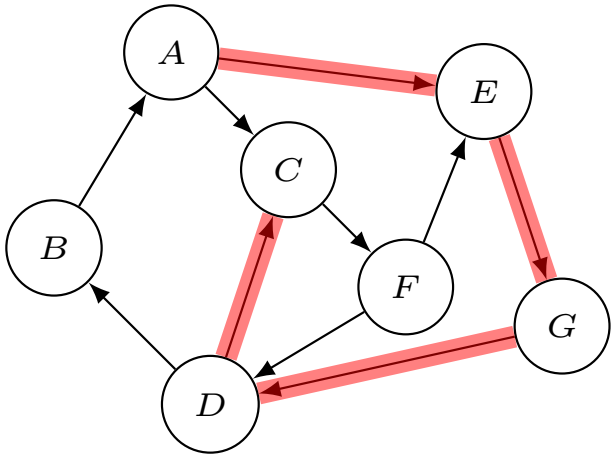


BFS ordning

A	C	E	F	G	D	B

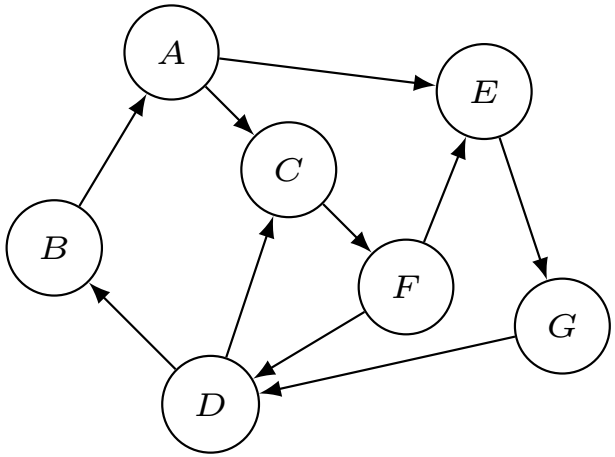
Grafsökning

Jämförelse DFS och BFS



DFS ordning

A	E	G	D	C	F	B

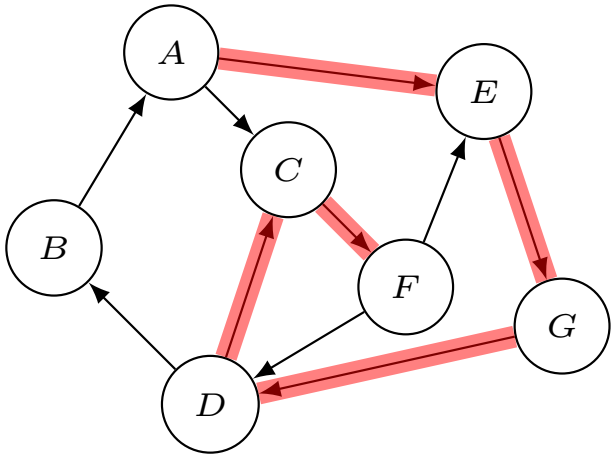


BFS ordning

A	C	E	F	G	D	B

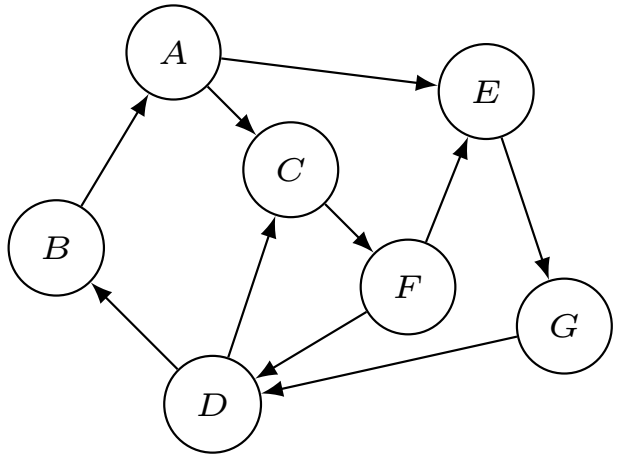
Grafsökning

Jämförelse DFS och BFS



DFS ordning

A	E	G	D	C	F	B

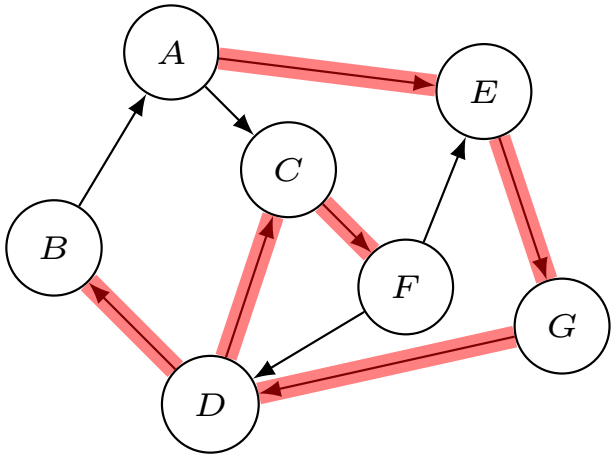


BFS ordning

A	C	E	F	G	D	B

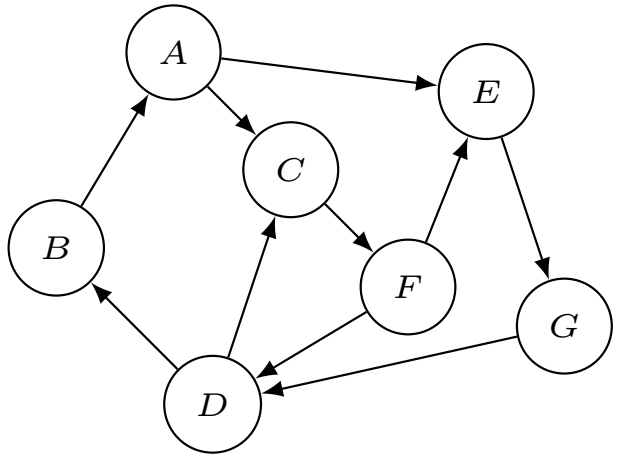
Grafsökning

Jämförelse DFS och BFS



DFS ordning

A	E	G	D	C	F	B

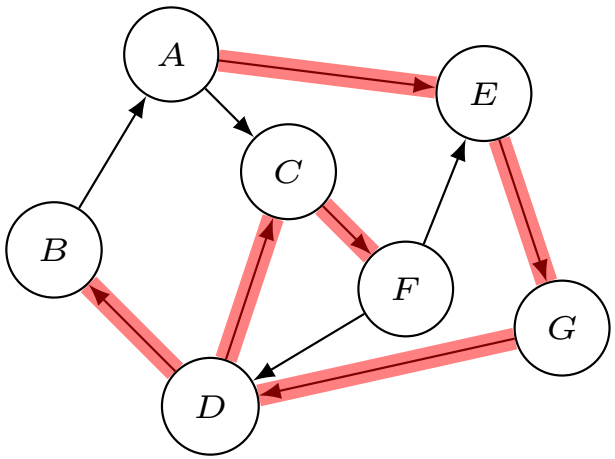


BFS ordning

A	C	E	F	G	D	B

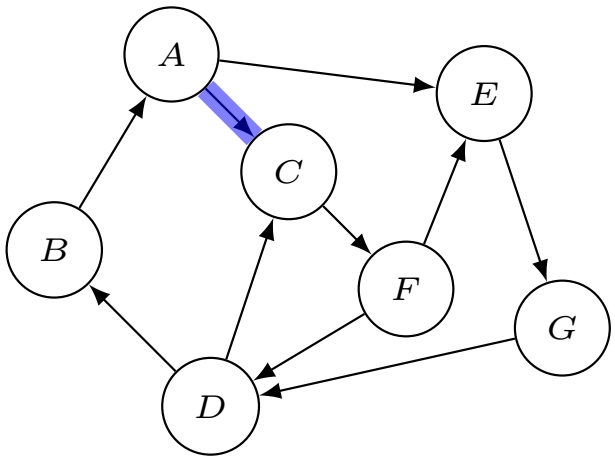
Grafsökning

Jämförelse DFS och BFS



DFS ordning

A	E	G	D	C	F	B

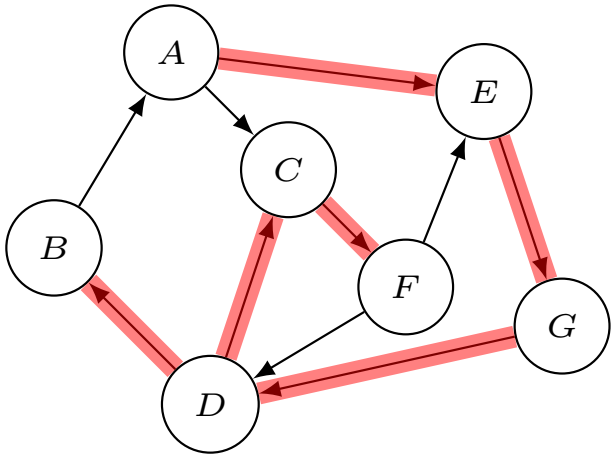


BFS ordning

A	C	E	F	G	D	B

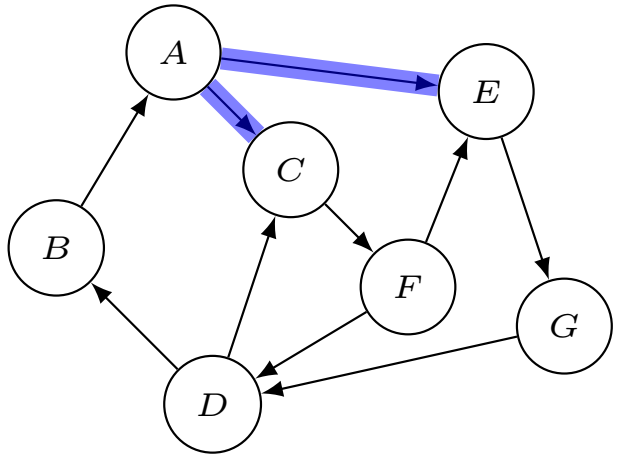
Grafsökning

Jämförelse DFS och BFS



DFS ordning

A	E	G	D	C	F	B

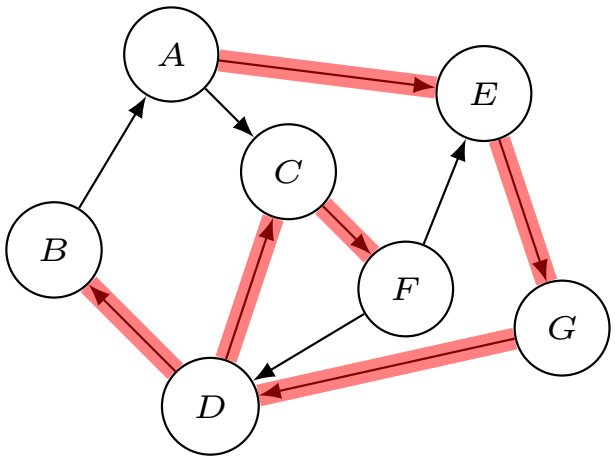


BFS ordning

A	C	E	F	G	D	B

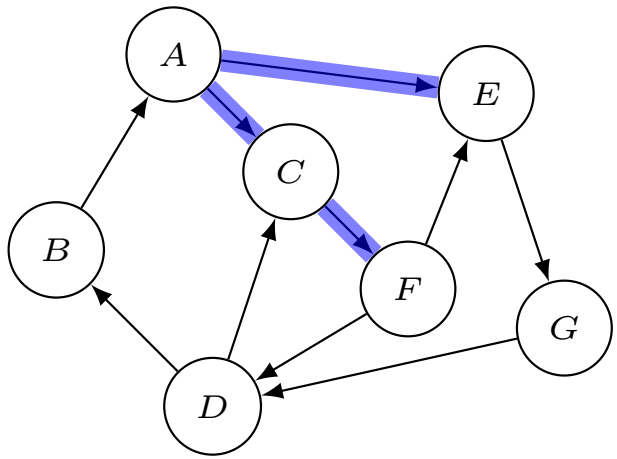
Grafsökning

Jämförelse DFS och BFS



DFS ordning

A	E	G	D	C	F	B

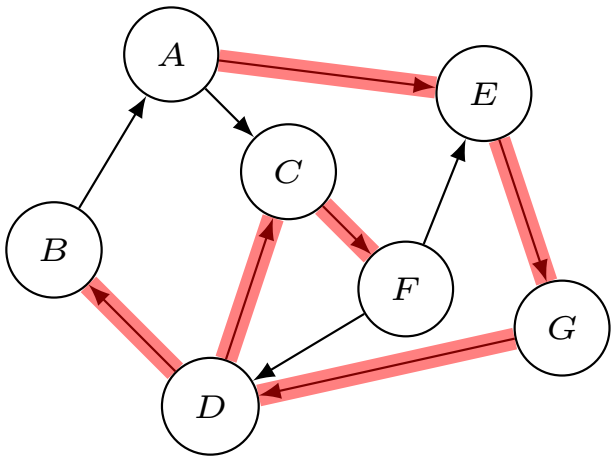


BFS ordning

A	C	E	F	G	D	B

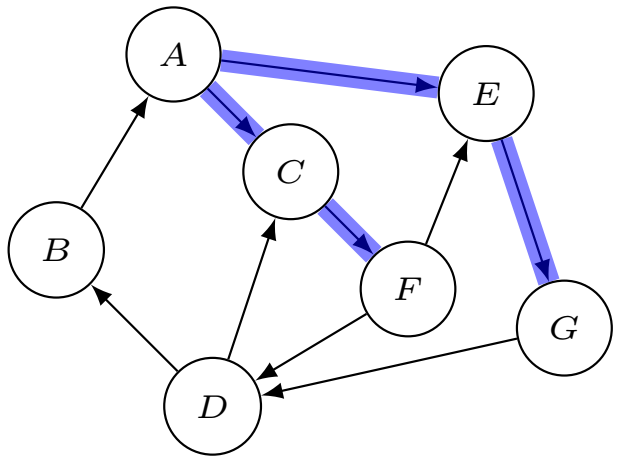
Grafsökning

Jämförelse DFS och BFS



DFS ordning

A	E	G	D	C	F	B

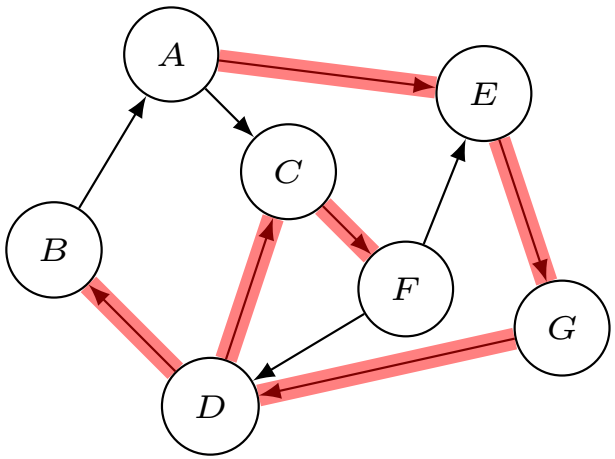


BFS ordning

A	C	E	F	G	D	B

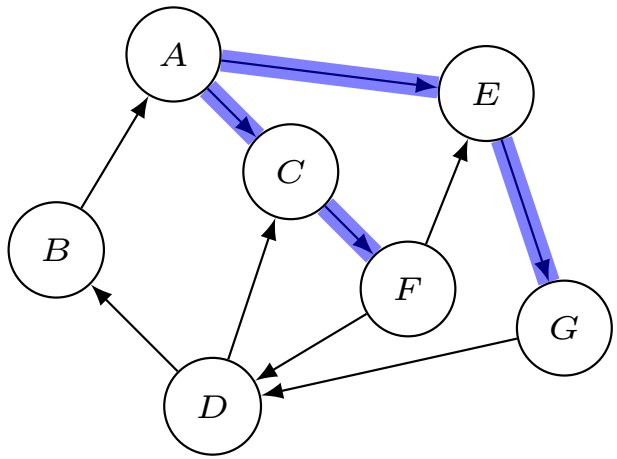
Grafsökning

Jämförelse DFS och BFS



DFS ordning

A	E	G	D	C	F	B

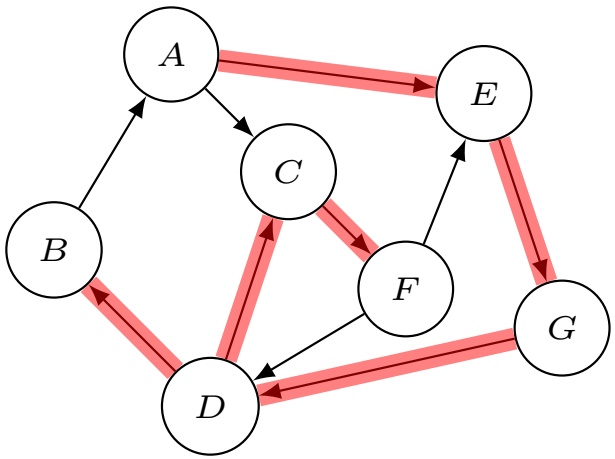


BFS ordning

A	C	E	F	G	D	B

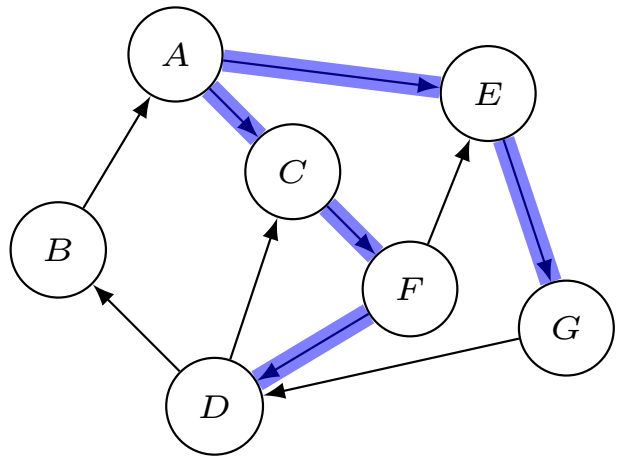
Grafsökning

Jämförelse DFS och BFS



DFS ordning

A	E	G	D	C	F	B

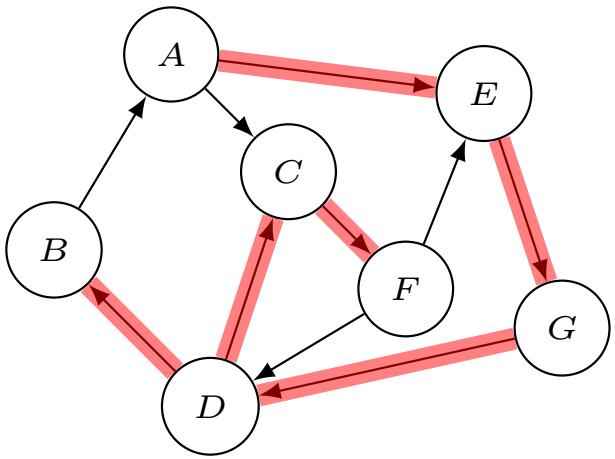


BFS ordning

A	C	E	F	G	D	B

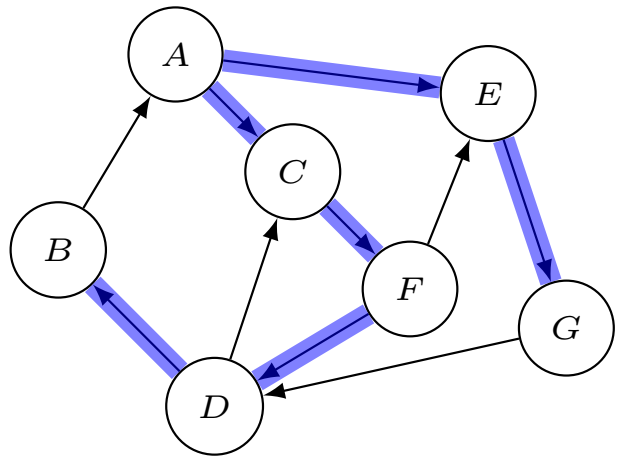
Grafsökning

Jämförelse DFS och BFS



DFS ordning

A	E	G	D	C	F	B



BFS ordning

A	C	E	F	G	D	B

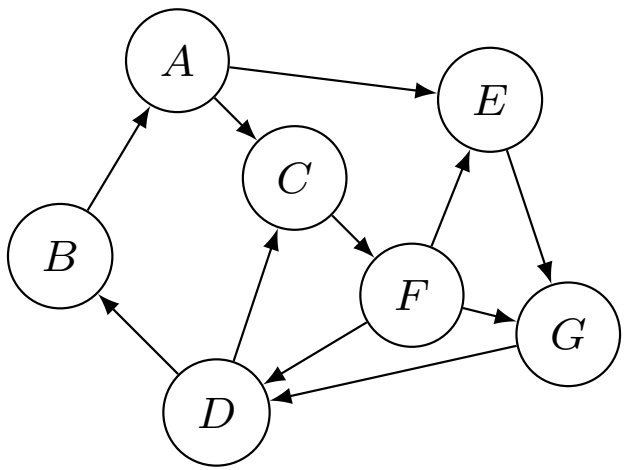
Grafsökning

Jämförelse DFS och BFS

- Både DFS och BFS hittar garanterat alla noder som är sammanhängande med startnoden.
- DFS är något enklare att implementera då det (enkelt) kan göras rekursivt.
- BFS traverserar den kortaste vägen till varje nod (bevis görs i inlämningen)
- DFS hittar en väg snabbt, men vi kan inte säga något särskilt om den vägen.
- DFS kommer tidigt hitta noder som ligger långt bort, medan BFS måste traversera lager för lager i grafen (först alla som är ett steg från startnoden, sedan alla som är två steg bort, o.s.v.).
- Dessa egenskaper gör att BFS och DFS lämpar sig för olika ändamål: BFS används i regel när en kortaste väg är viktig, men i andra fall tenderar DFS att vara mer naturligt och enklare.
- Tidskomplexiteten för BFS och DFS är samma (vad den är lämnas till inlämningen)
- På nästa slide diskuterar vi hur man använder BFS och DFS för att hitta vägar.

Grafsökning

Hitta en kortaste väg mellan A och F



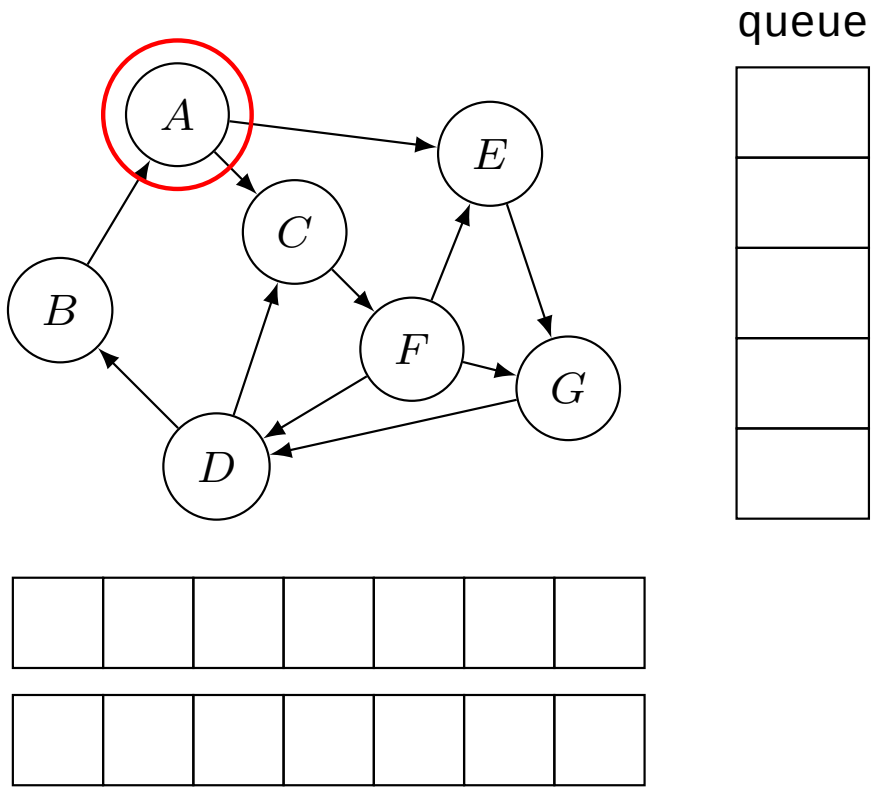
queue

$A, -$

found:							
from:							

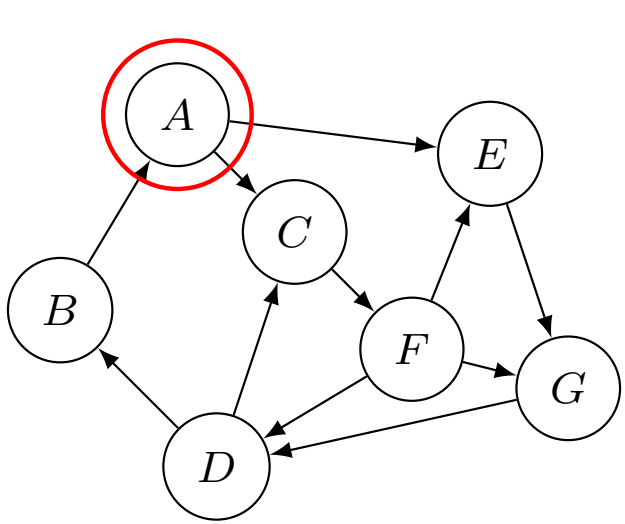
Grafsökning

Hitta en kortaste väg mellan A och F

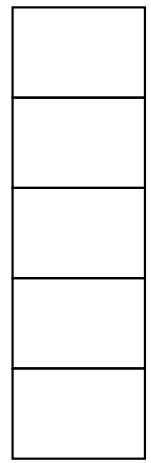


Grafsökning

Hitta en kortaste väg mellan A och F



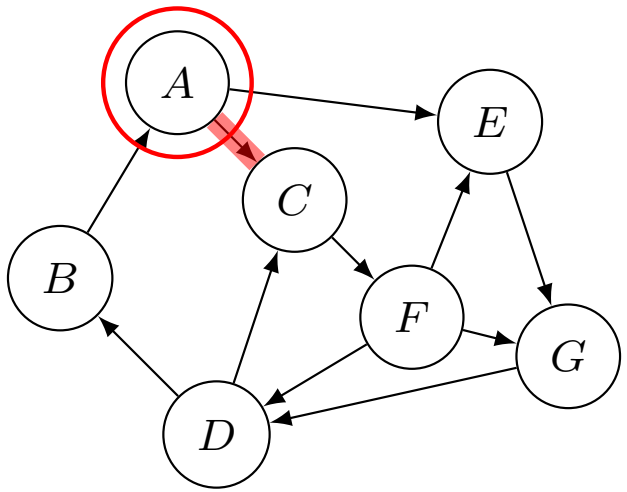
queue



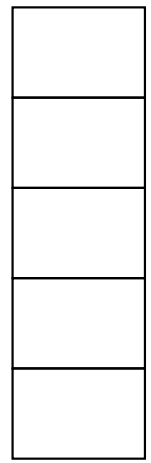
found:	A						
from:	$-$						

Grafsökning

Hitta en kortaste väg mellan A och F



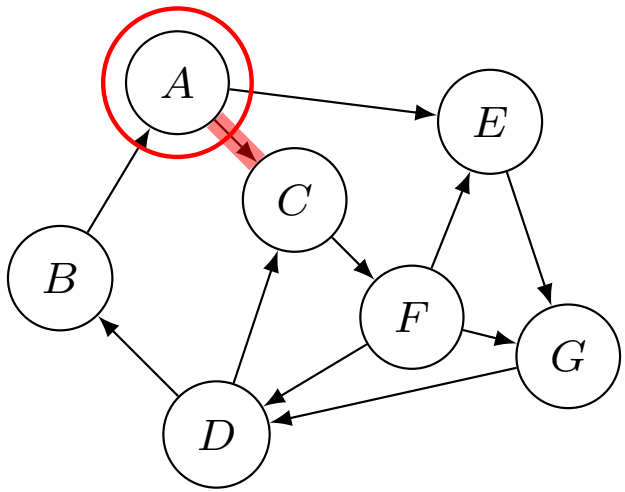
queue



found:	A						
from:	$-$						

Grafsökning

Hitta en kortaste väg mellan A och F



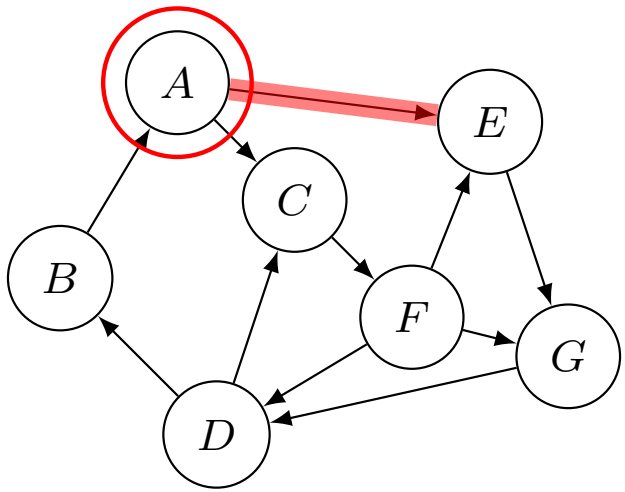
queue

C, A

found:	A						
from:	$-$						

Grafsökning

Hitta en kortaste väg mellan A och F



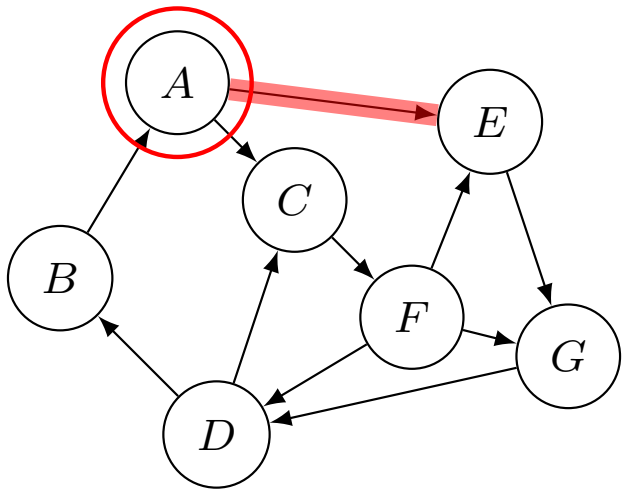
queue

C, A

found:	A						
from:	$-$						

Grafsökning

Hitta en kortaste väg mellan A och F



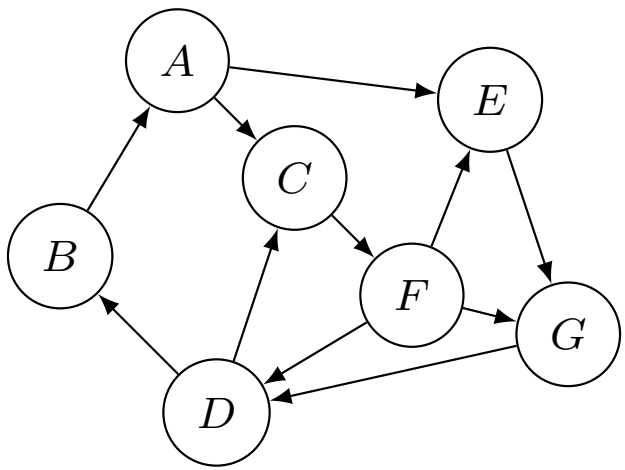
queue

C, A
E, A

found:	A						
from:	$-$						

Grafsökning

Hitta en kortaste väg mellan A och F



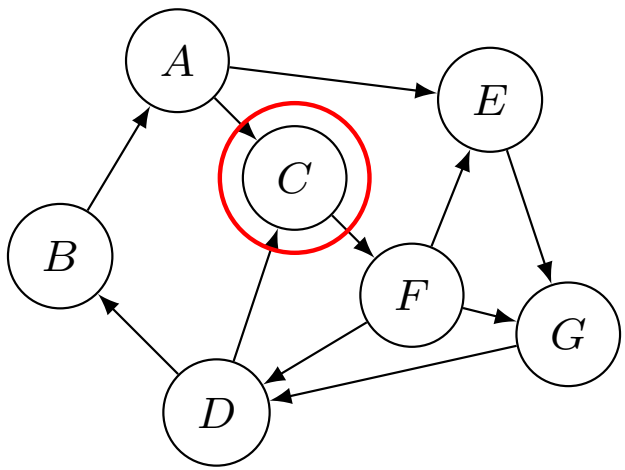
queue

C, A
E, A

found:	A						
from:	$-$						

Grafsökning

Hitta en kortaste väg mellan A och F



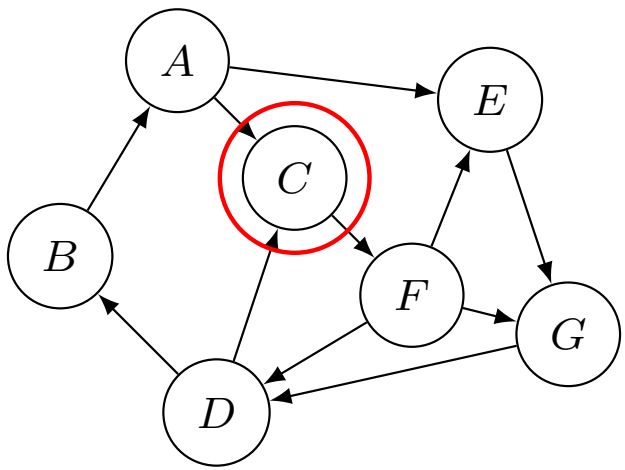
queue

E, A

found:	A						
from:	$-$						

Grafsökning

Hitta en kortaste väg mellan A och F



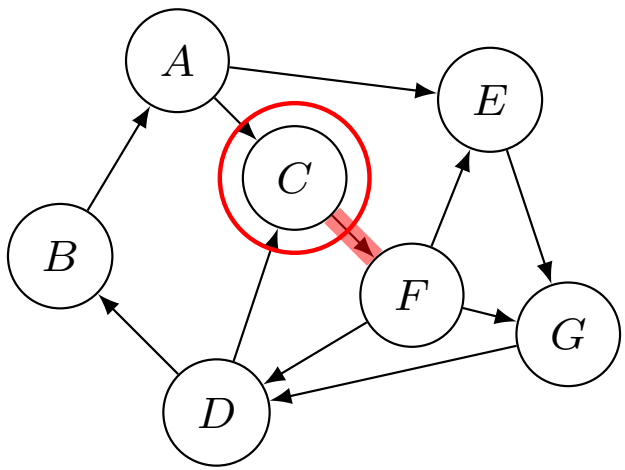
queue

E, A

found:	A	C					
from:	$-$	A					

Grafsökning

Hitta en kortaste väg mellan A och F



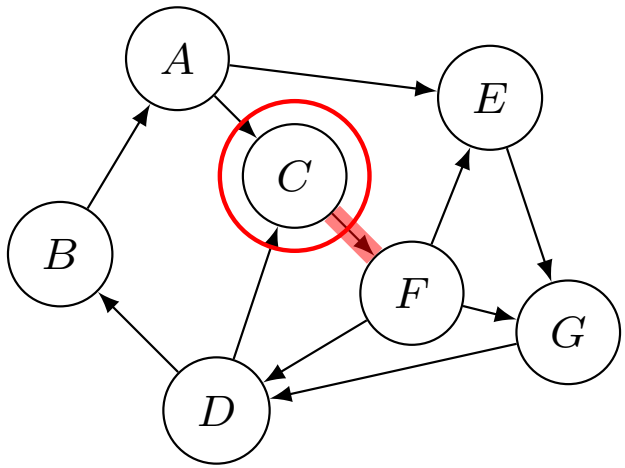
queue

E, A

found:	A	C					
from:	$-$	A					

Grafsökning

Hitta en kortaste väg mellan A och F



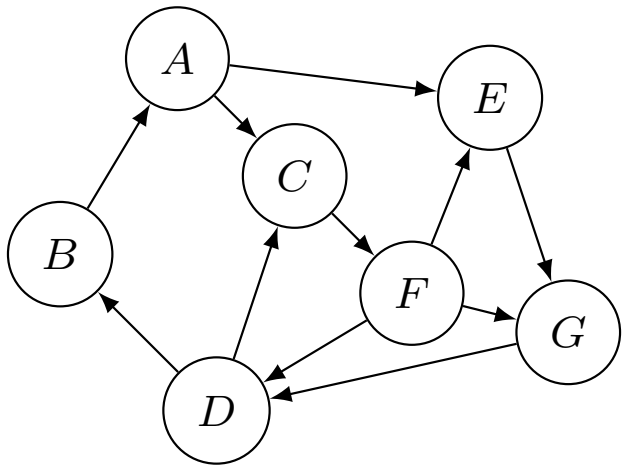
queue

E, A
F, C

found:	A	C					
from:	$-$	A					

Grafsökning

Hitta en kortaste väg mellan A och F



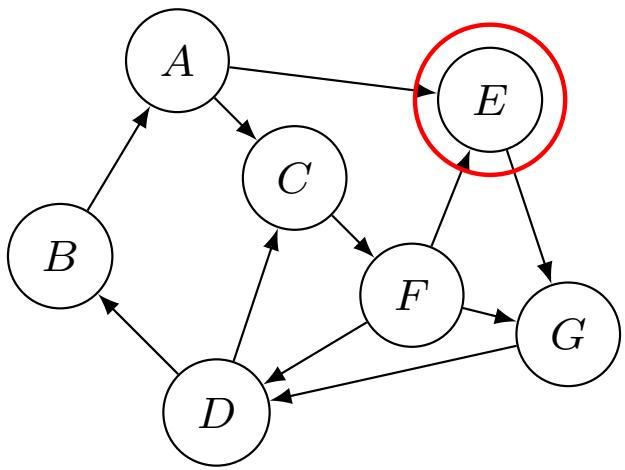
queue

E, A
F, C

found:	A	C					
from:	$-$	A					

Grafsökning

Hitta en kortaste väg mellan A och F



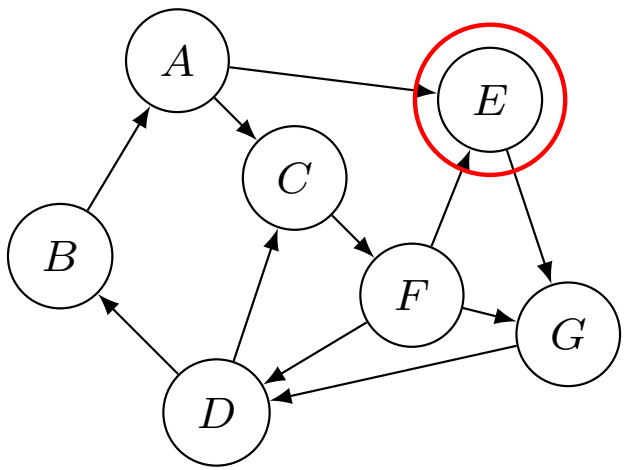
queue

F, C

found:	A	C					
from:	$-$	A					

Grafsökning

Hitta en kortaste väg mellan A och F



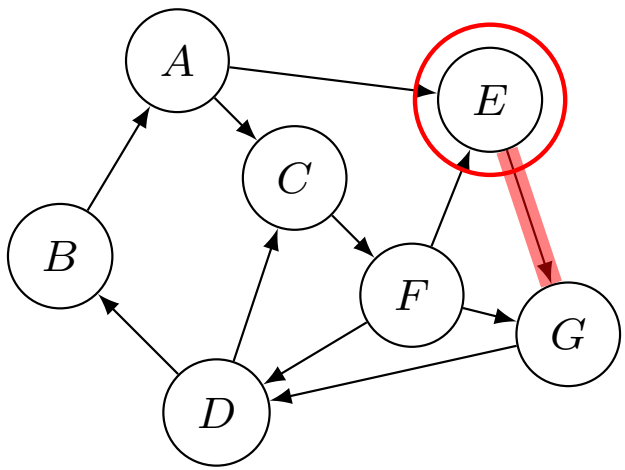
queue

F, C

found:	A	C	E				
from:	$-$	A	A				

Grafsökning

Hitta en kortaste väg mellan A och F



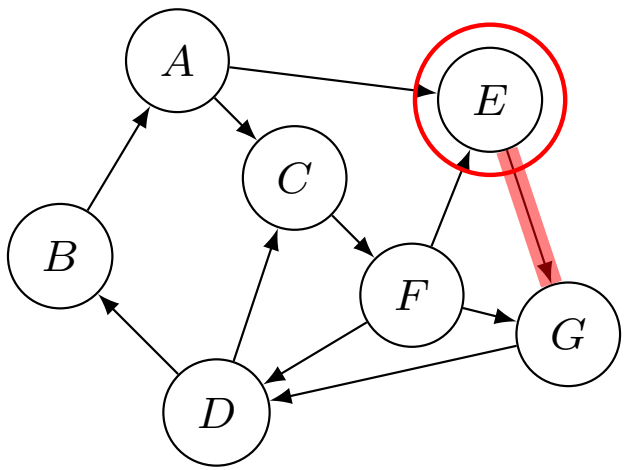
queue

F, C

found:	A	C	E				
from:	$-$	A	A				

Grafsökning

Hitta en kortaste väg mellan A och F



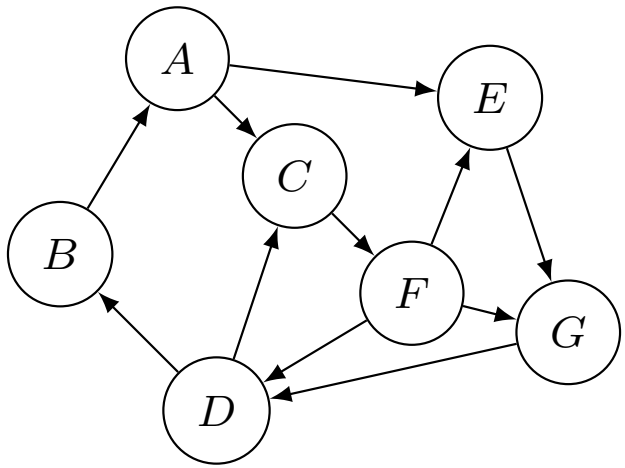
queue

F, C
G, E

found:	A	C	E				
from:	$-$	A	A				

Grafsökning

Hitta en kortaste väg mellan A och F



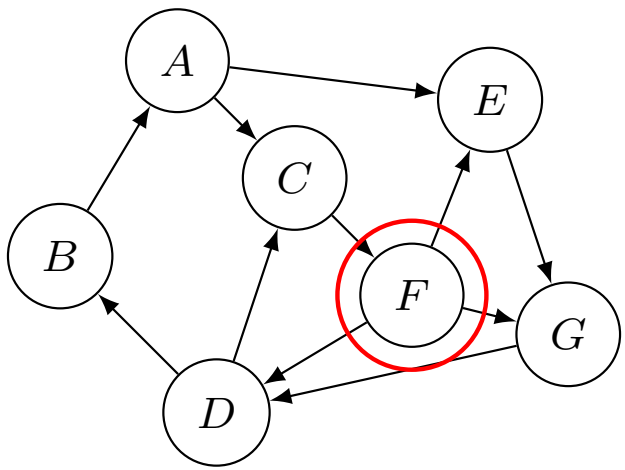
queue

F, C
G, E

found:	A	C	E				
from:	$-$	A	A				

Grafsökning

Hitta en kortaste väg mellan A och F



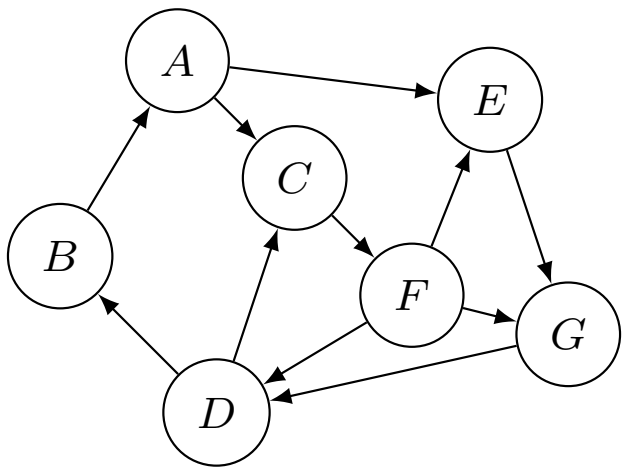
queue

G, E

found:	A	C	E				
from:	$-$	A	A				

Grafsökning

Hitta en kortaste väg mellan A och F



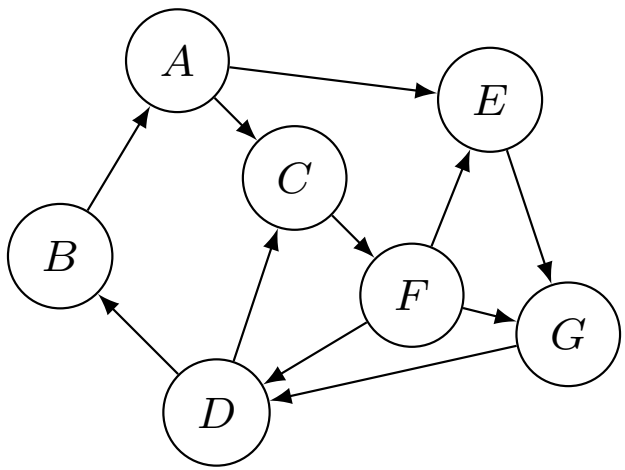
queue

G, E

found:	A	C	E	F			
from:	$-$	A	A	C			

Grafsökning

Hitta en kortaste väg mellan A och F



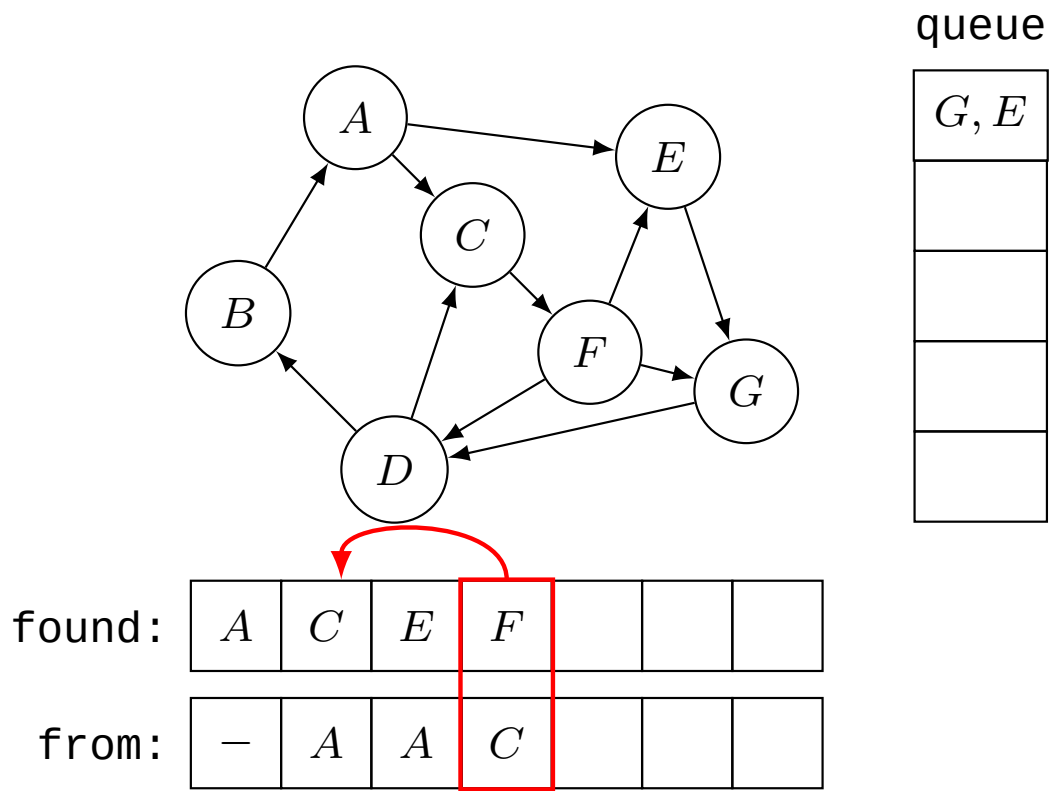
queue

G, E

found:	A	C	E	F			
from:	$-$	A	A	C			

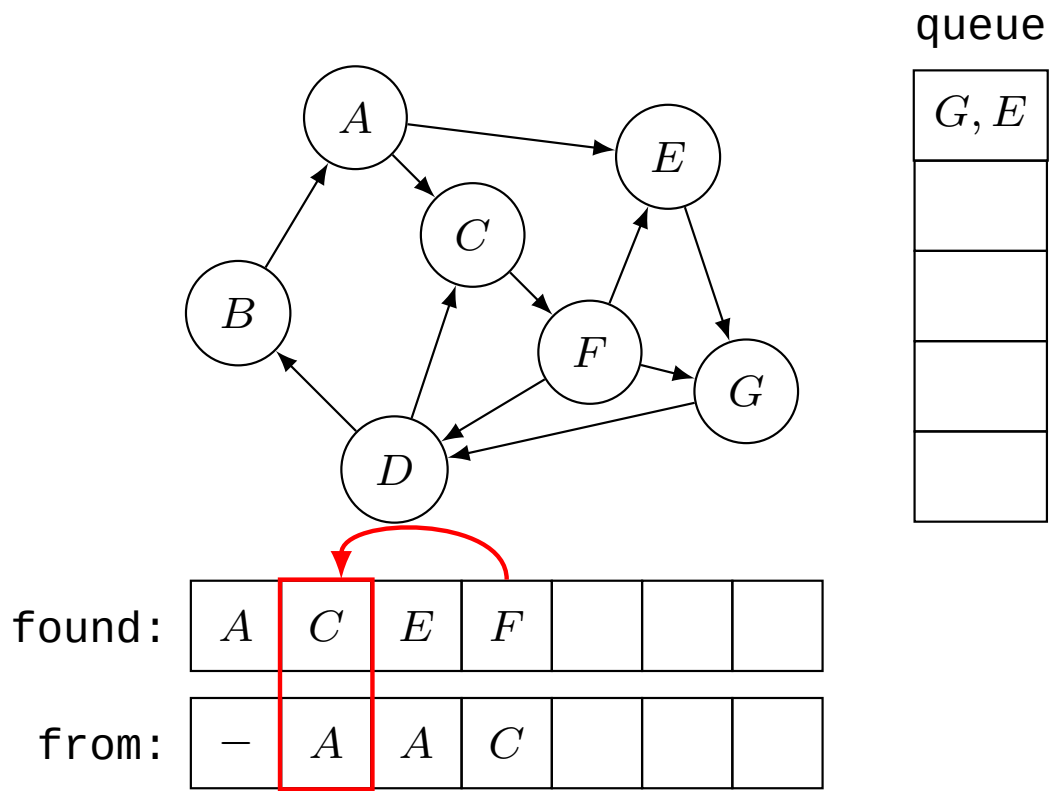
Grafsökning

Hitta en kortaste väg mellan A och F



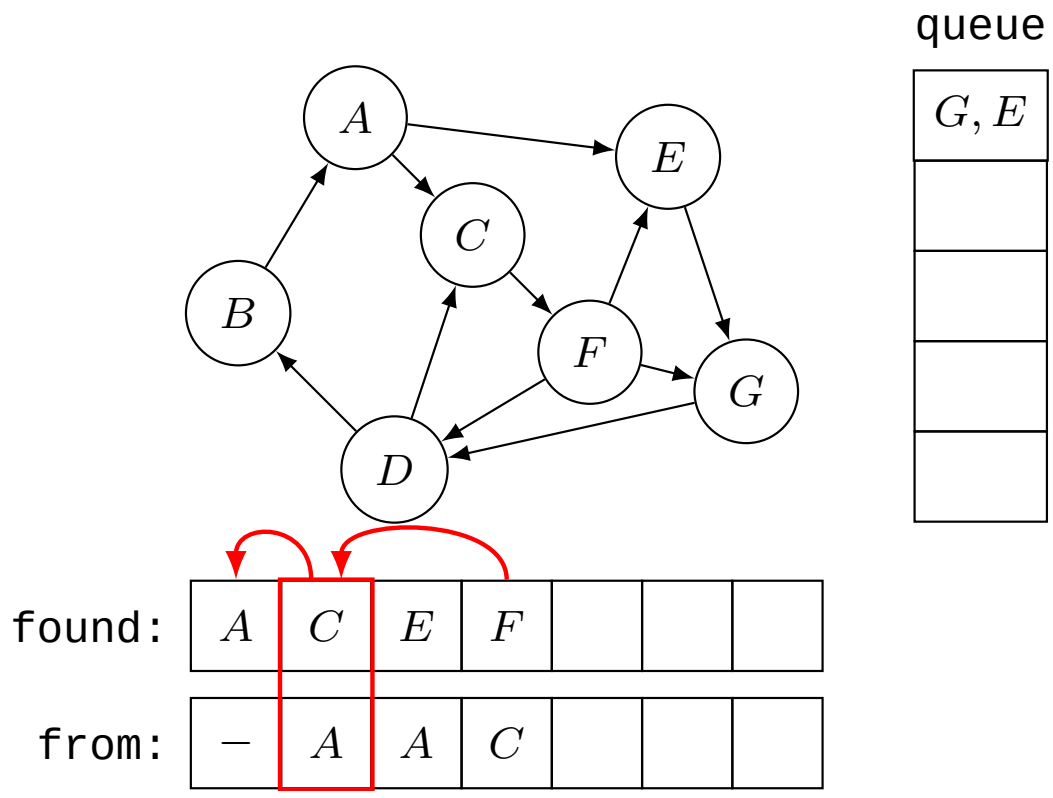
Grafsökning

Hitta en kortaste väg mellan A och F



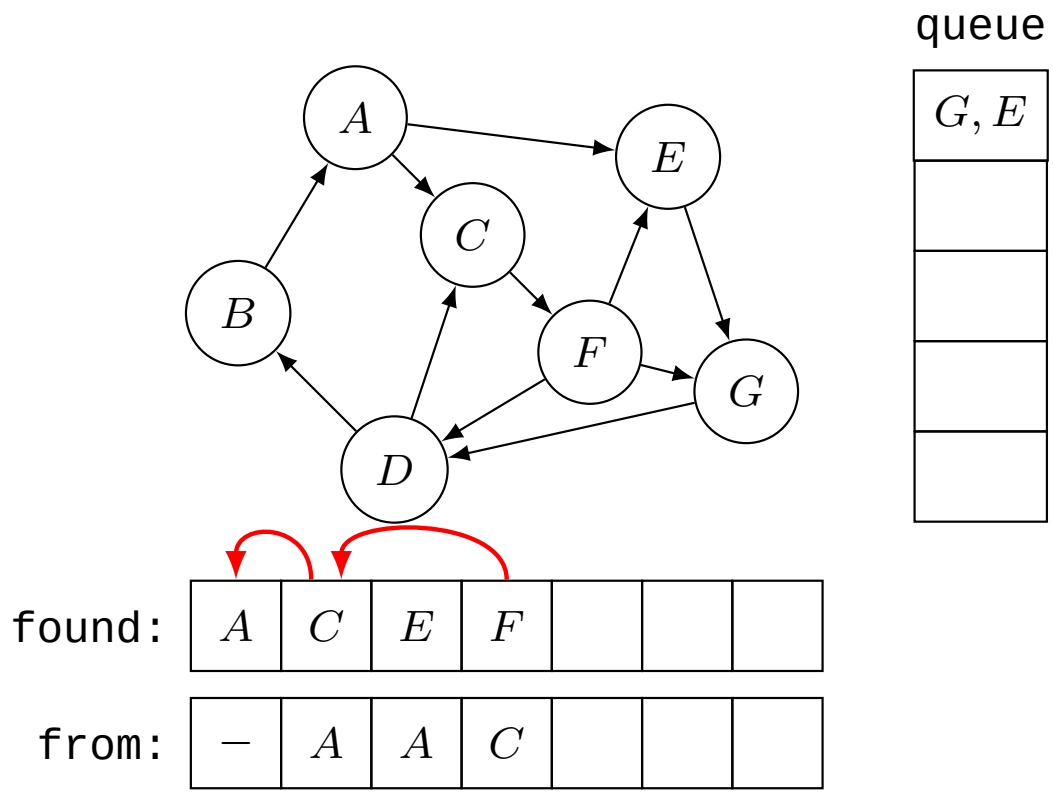
Grafsökning

Hitta en kortaste väg mellan A och F



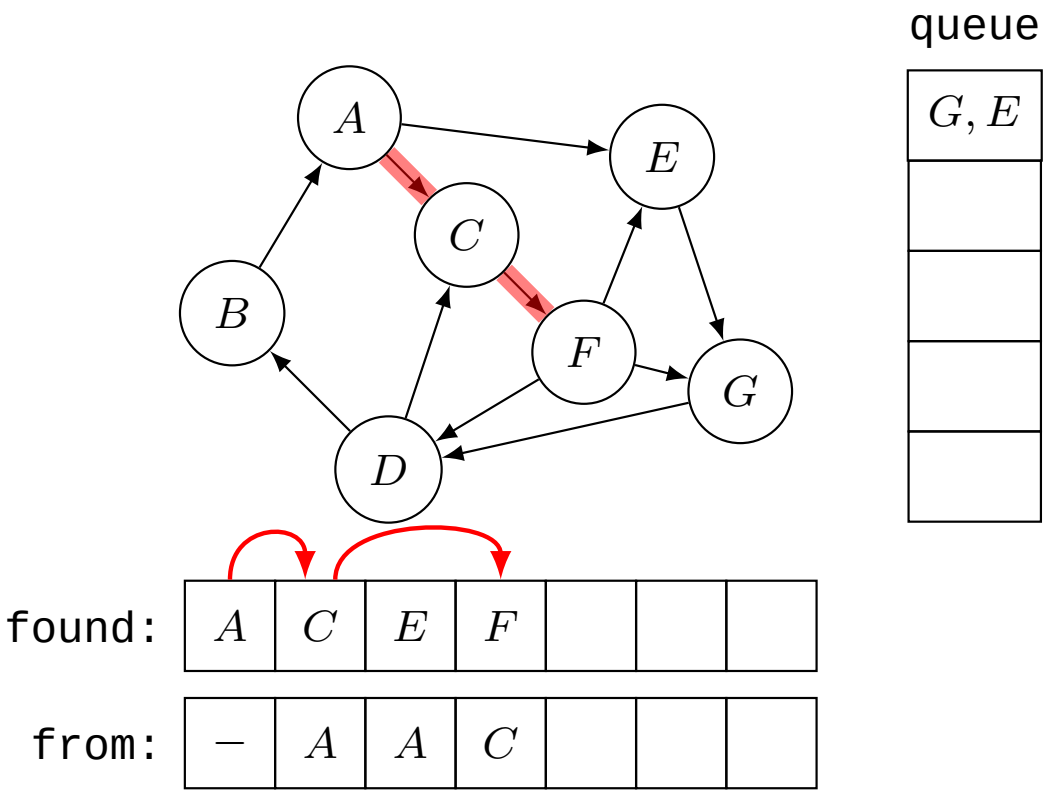
Grafsökning

Hitta en kortaste väg mellan A och F



Grafsökning

Hitta en kortaste väg mellan A och F



Grafsökning

Hitta kortaste väg mellan två noder

```
1  vector<Node*> shortest_path(Node* start, int end_id)
2  {
3    unordered_map<Node*, Node*> from;
4    queue<pair<Node*, Node*>> queue;
5    queue.push({ start, nullptr });
6    while (!queue.empty())
7    {
8      auto [curr, prev] = queue.front();
9      queue.pop();
10     if (from.count(curr) > 0) continue;
11     if (curr->id == end_id)
12       return produce_path(from, curr);
13
14     parents[curr] = prev;
15     for (Node* next : curr->edges)
16       queue.push({ next, curr });
17   }
18   return { }; // ingen väg hittade
19 }
```

Grafsökning

Hitta kortaste väg mellan två noder

```
1  vector<Node*> produce_path(unordered_map<Node*, Node*> const& from,
2                             Node* end)
3  {
4    vector<Node*> path;
5    Node* curr { end };
6    while (curr != nullptr)
7    {
8      path.push_back(curr);
9      curr = from.at(curr);
10   }
11
12   reverse(path.begin(), path.end());
13   return path;
14 }
```

Grafsökning

Billigaste väg

Definition: Billigaste väg

I en viktad graf $G_w = (V, E)$ ges *kostnaden* för en väg $P = \langle v_1, v_2, \ldots, v_n \rangle$ av

$$c(P) = \sum_{i=1}^{n-1} w(v_i, v_{i+1})$$

En billigaste väg är en väg P sådan att $c(P) \leq c(Q)$ för alla vägar Q. Dessa vägar hittas m.h.a. *Dijkstras algoritm*.

Grafsökning

Dijkstras algoritm

- Dijkstras algoritm (känd från TAOP07) är en *variant* av bredden först sökning, men istället för att söka i lager, så söker vi hela tiden längs den *hittills* billigaste vägen.
- För att göra detta måste vi likt tidgare hålla koll på mer information, specifikt hur kostnaden av den hittills billigaste vägen till varje utforskad nod.
- Detta innebär att vi varken kan använda en stack eller en kö. Istället vill vi ha en s.k. prioritetskö där vi alltid plockar ut den billigaste vägen.
- Hur implementerar vi egentligen det?

1 Repetition: Grafer
2 Repetition: Riktade grafer
3 Representation av grafer
4 Grafsökning
5 Heap

LINKÖPING UNIVERSITY

Heap

Mål med prioritetskö

- Slå upp högst prioritet: $\mathcal{O}(1)$
- Ta bort högst prioritet: Helst bättre än $\mathcal{O}(n)$
- Insättningen: Helst bättre än $\mathcal{O}(n)$
- Antag att högsta prioritet är det *minsta* värdet

Heap

Idé

Idé #1

Använd ett balanserat binärt sökträd, då är allt $\mathcal{O}(\log n)$!

Heap

Idé

- Ett balanserat binärt sökträd kan hitta (och ta bort) det minsta värdet i $\mathcal{O}(\log n)$ tid, vilket inte är fullt så bra som vad vi siktar på.
- Men insättning görs i $\mathcal{O}(\log n)$ vilket absout är bättre än $\mathcal{O}(n)$.
- Så ett balanserat binärt sökträd duger inte för våra ändamål, men vi kan göra vissa observationer för att förbättra detta...

Heap

Observation #1

Observation #1

Att hitta rotnoden i ett träd görs i $\mathcal{O}(1)$.

Heap

Idé #2

Idé #2

Behåll det balanserade binära trädet men skippa sökstrukturen. Lagra istället det minsta värdet i roten.

Heap

Idé #2

- Om vi ser till att rotnoden *alltid* innehåller det minsta värdet kan vi enkelt slå upp det i $\mathcal{O}(1)$.
- ... Men vi vill kunna ta bort det minsta värdet också.
- För att göra det måste vi dels byta plats på rotnoden och en lövnod, och sedan hitta det värde som är minst när roten är borta och flytta det till rotnoden.
- Den naiva lösningen på detta är att söka hela strukturen, vilket tar $\mathcal{O}(n)$.
- Men vi vill göra bättre...

Heap

Observation #2

Observation #2

Om vi strukturerar trädet s.a. det näst-minsta värdet också kan hittas i konstant tid så blir borttagningen mycket enklare.

Heap

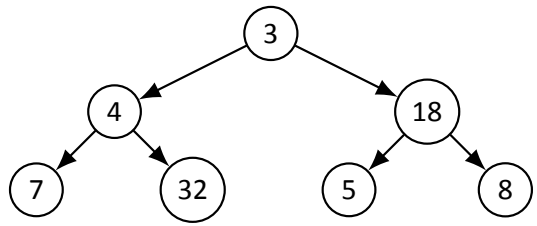
Idé #3

Idé #3

Se till att det näst-minsta värdet alltid ligger som ett barn till roten.

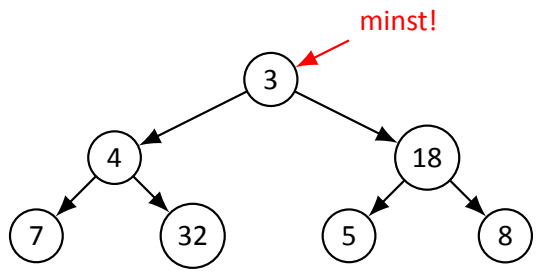
Heap

Vad vi har hittills



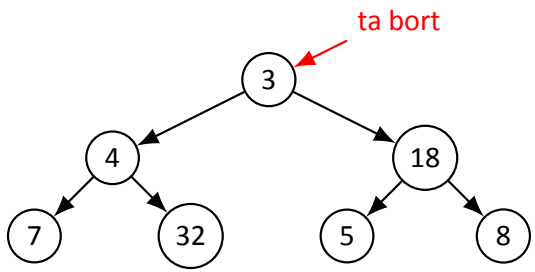
Heap

Vad vi har hittills



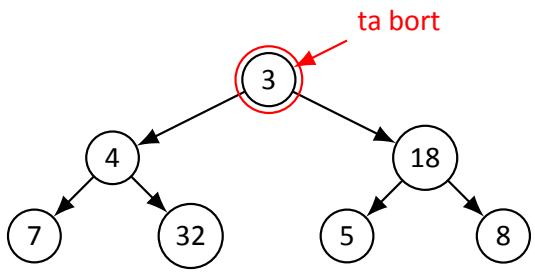
Heap

Vad vi har hittills



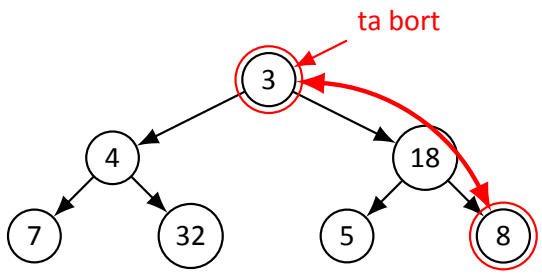
Heap

Vad vi har hittills



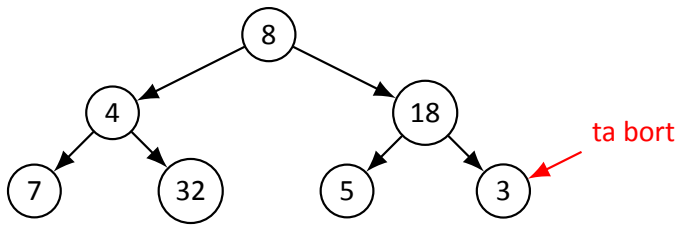
Heap

Vad vi har hittills



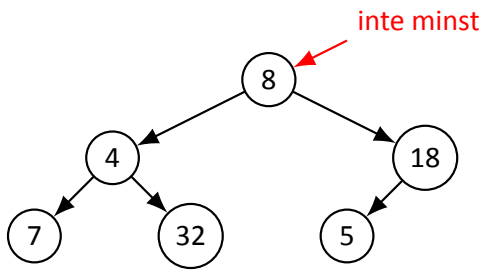
Heap

Vad vi har hittills



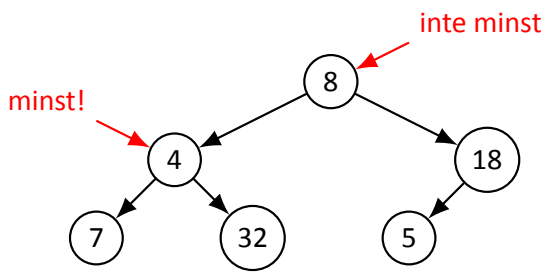
Heap

Vad vi har hittills



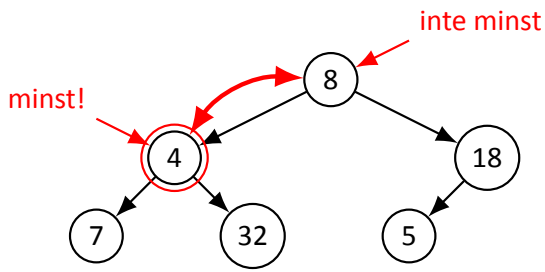
Heap

Vad vi har hittills



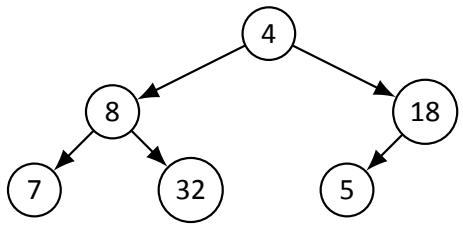
Heap

Vad vi har hittills



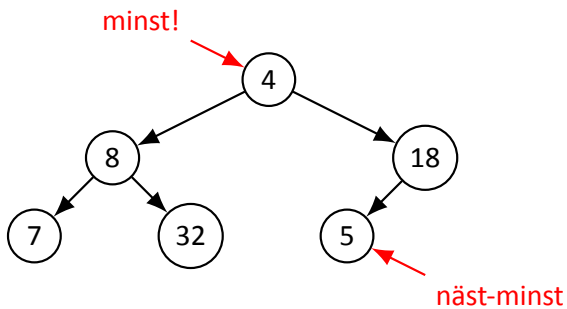
Heap

Vad vi har hittills



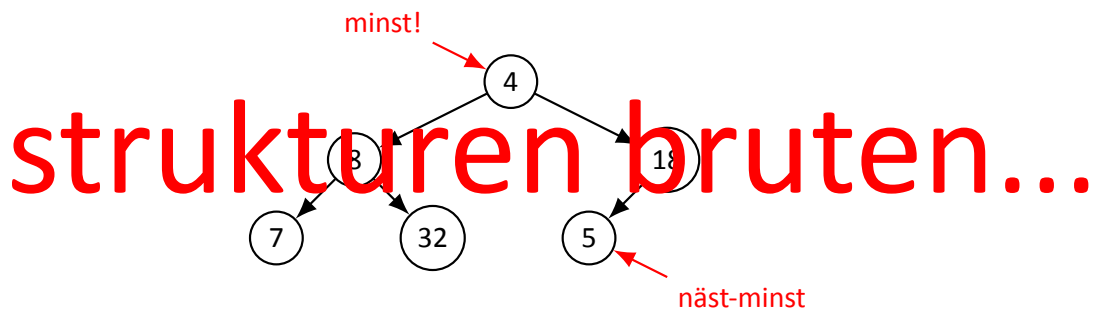
Heap

Vad vi har hittills



Heap

Vad vi har hittills



Heap

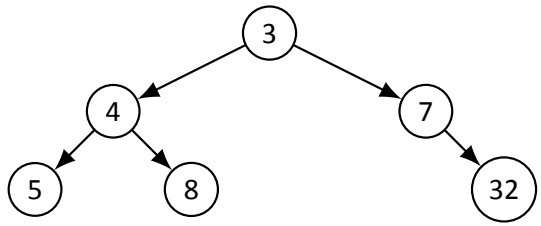
Idé #4

Idé #4

Vad händer om vi säger att alla barn ska vara *större* än sina föräldrar?

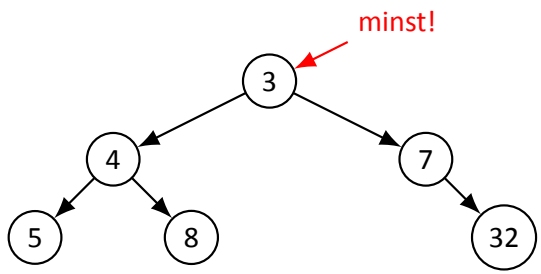
Heap

Mer struktur



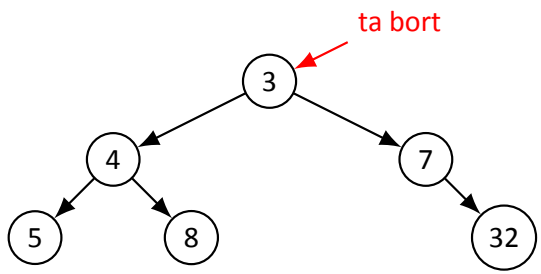
Heap

Mer struktur



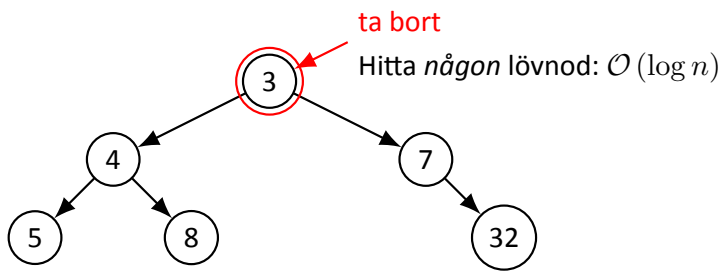
Heap

Mer struktur



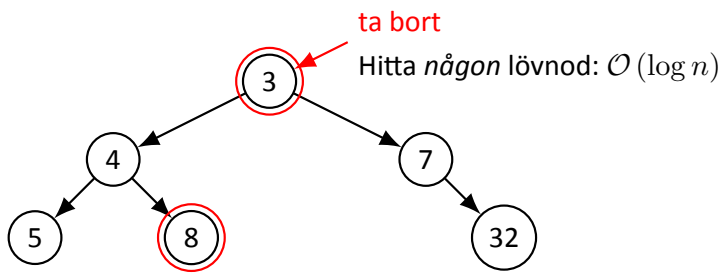
Heap

Mer struktur



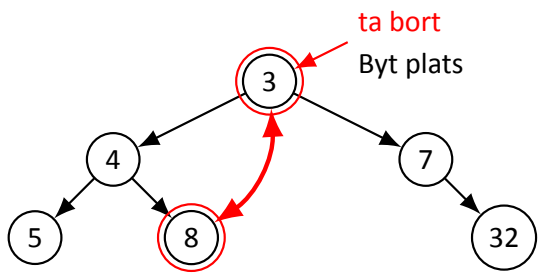
Heap

Mer struktur



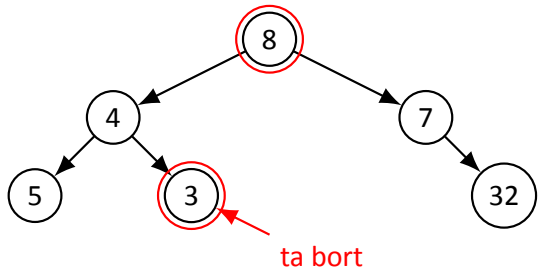
Heap

Mer struktur



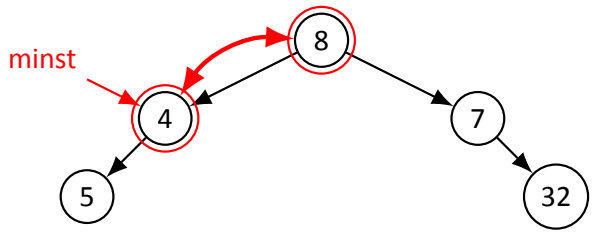
Heap

Mer struktur



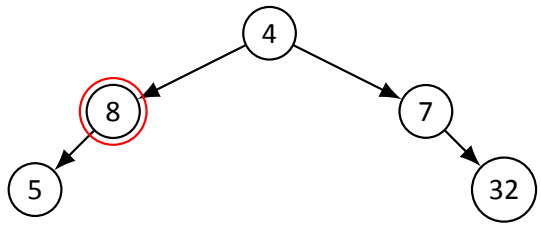
Heap

Mer struktur



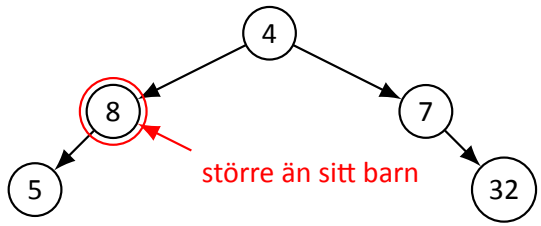
Heap

Mer struktur



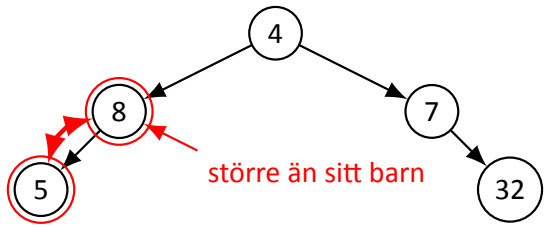
Heap

Mer struktur



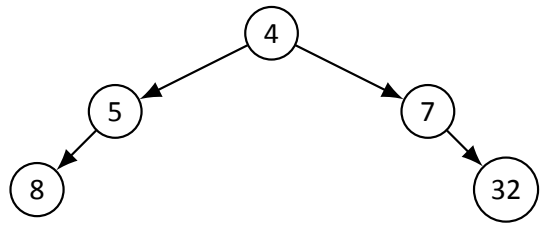
Heap

Mer struktur



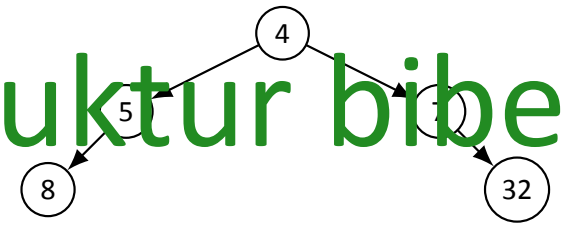
Heap

Mer struktur



Heap

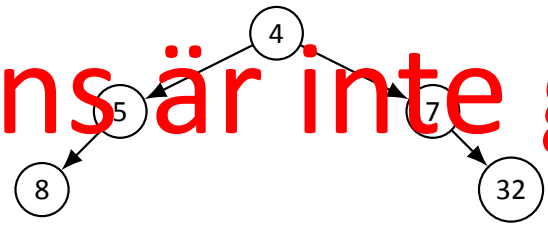
Mer struktur



Struktur bibehålls!

Heap

Mer struktur



Men balans är inte garanterat...

Heap

Mer struktur

- Idén att varje nod ska vara mindre än sina barn måste upprätthållas efter varje operation.
- Detta gör att borttagning av rotnoden blir något mer involverat.
- Vi behöver först och främst hitta en lövnod (vilken som helst), vilket tar $\mathcal{O}(\log n)$ (givet att trädet är *balanserat*)
- Sedan byter vi plats på rotnoden och den funna lövnoden.
- Efter det vill vi att det näst-minsta värdet nu ska ligga i roten, men den är lätt att hitta: den är en av barnen till rotnoden.
- Vi byter plats på dessa två, och sedan upprepar vi den här processen tills trädets struktur är återställd.
- Denna swap-kedja besöker som mest varje nivå i trädet, så $\mathcal{O}(\log n)$ (givet att trädet är balanserat)
- ... men ingenstans garanterar vi att balansen upprätthålls...

Heap

Nästan komplett träd

Definition: Komplett– och nästan komplett träd

- Ett träd $\mathcal{T}$ är *komplett* om det har exakt $2^k - 1$ noder fördelade i exakt k nivåer.
- Ett träd $\mathcal{T}$ är *nästan komplett* om alla nivår förutom sista är fulla, och den sista nivån är fylld från vänster till höger.
- **Följd:** Ett nästan komplett träd är balanserat

Heap

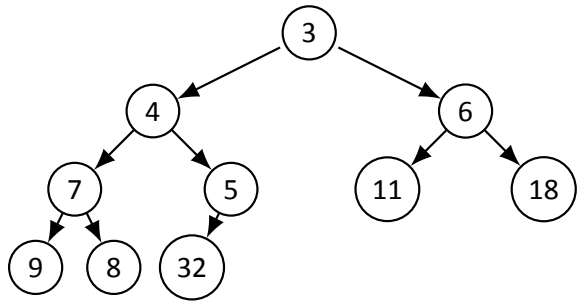
Idé #5

Idé #5

- Lagra vår prioritetskö i ett nästan komplett träd, d.v.s. fyll varje nivå från vänster till höger vid insättning.
- Vid borttagning, välj alltid den lövnod som är mest till höger på sista nivån

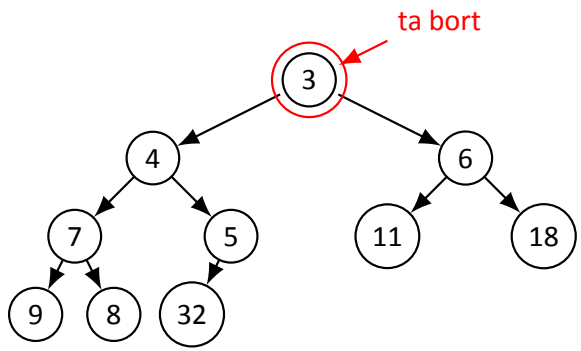
Heap

Nästan komplett träd



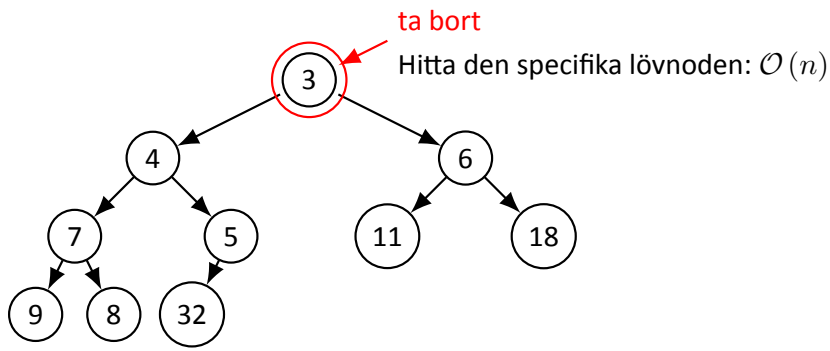
Heap

Nästan komplett träd



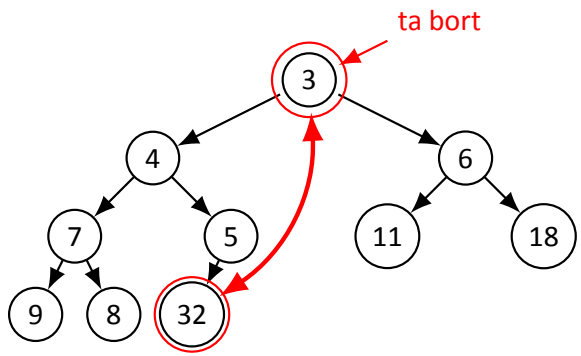
Heap

Nästan komplett träd



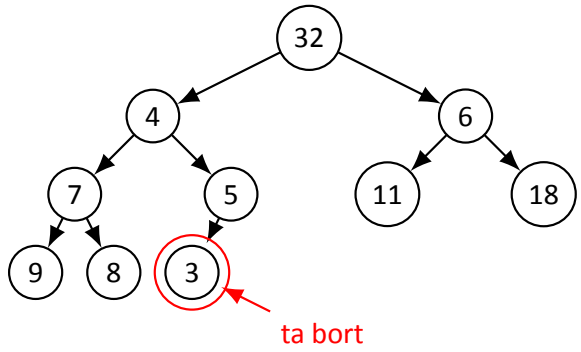
Heap

Nästan komplett träd



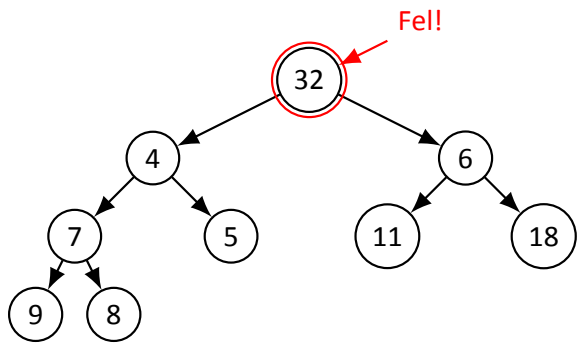
Heap

Nästan komplett träd



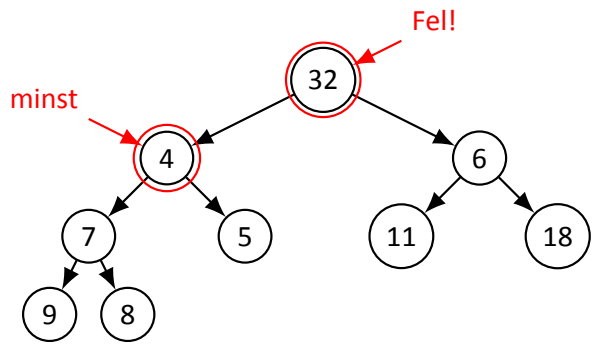
Heap

Nästan komplett träd



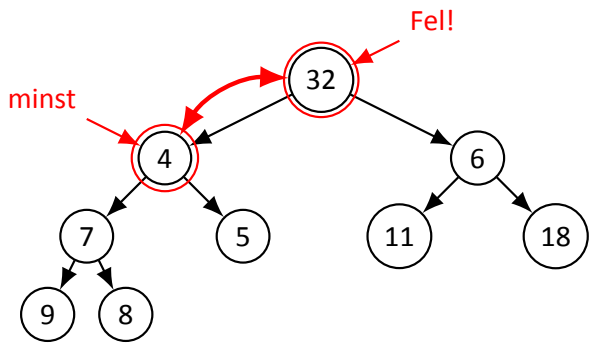
Heap

Nästan komplett träd



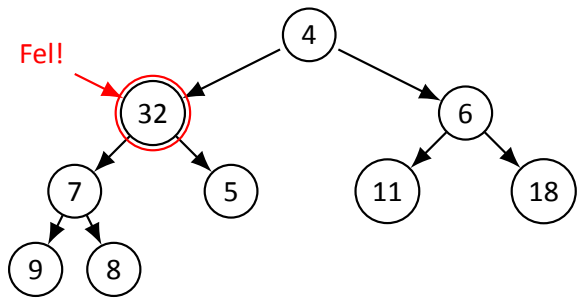
Heap

Nästan komplett träd



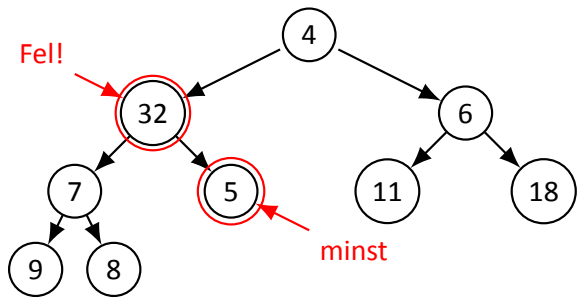
Heap

Nästan komplett träd



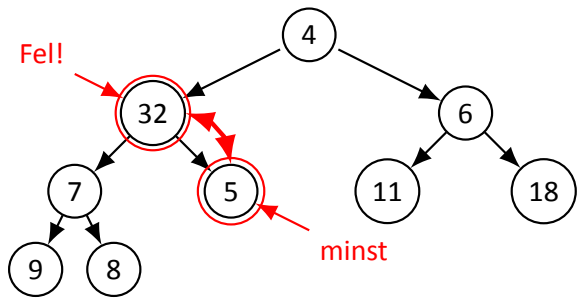
Heap

Nästan komplett träd



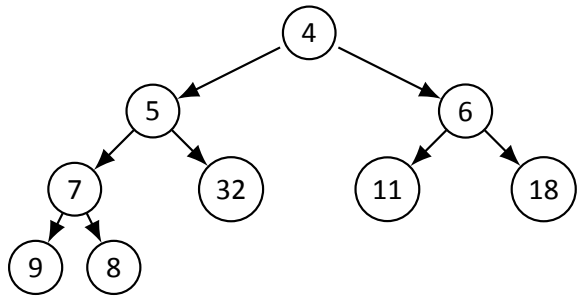
Heap

Nästan komplett träd



Heap

Nästan komplett träd



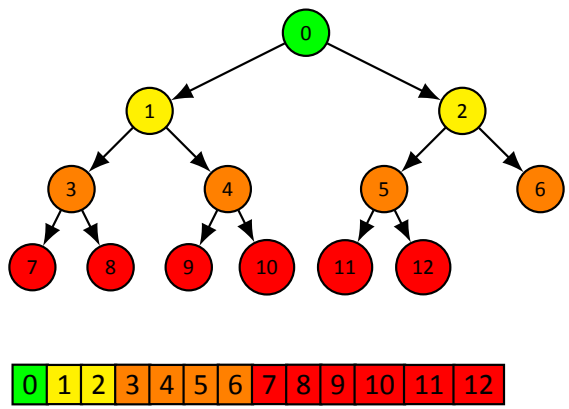
Heap

Nästan komplett träd

- I.o.m. att trädet är nästan komplett och vi alltid tar bort den mest högra lövnoden på sista nivån så kommer vi efter borttagning fortsätta ha ett komplett träd.
- P.g.a. detta kan vi garantera att trädet också alltid är balanserat.
- Detta gör att antalet nivåer alltid är $\mathcal{O}(\log n)$
- Nackdelen dock är ju att vi inte har ett smidigt sätt att hitta den mest högra noden i sista noden.
- Men detta går att lösa genom att fundera på hur ett nästan komplett träd kan lagras...

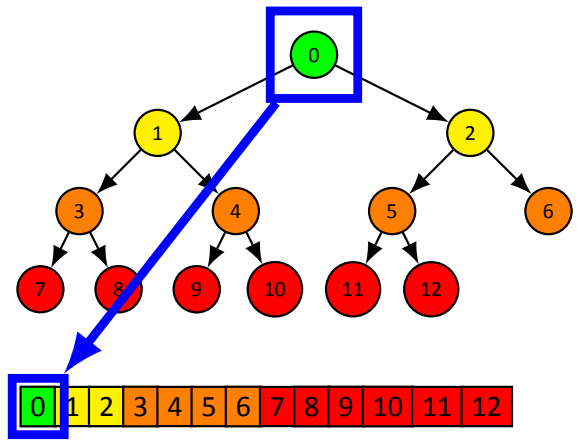
Heap

Lagra ett nästan komplett träd i en dynamisk array



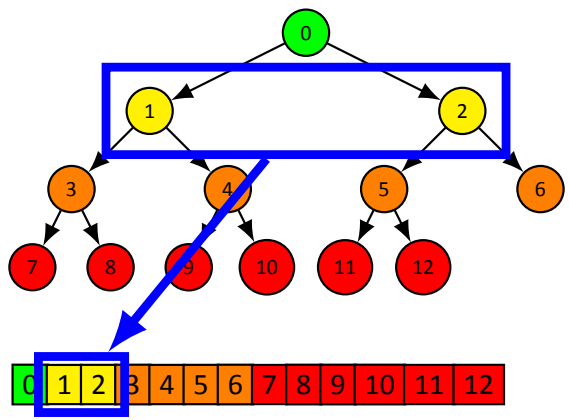
Heap

Lagra ett nästan komplett träd i en dynamisk array



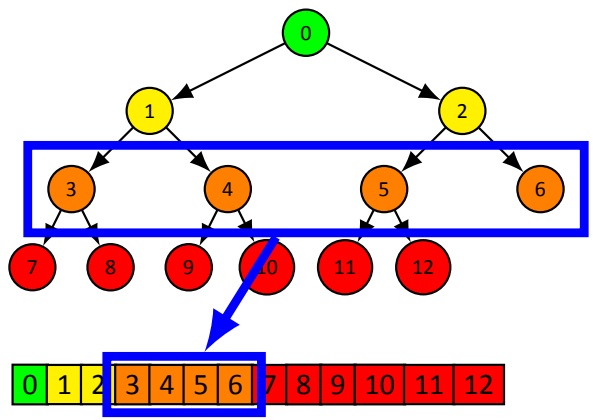
Heap

Lagra ett nästan komplett träd i en dynamisk array



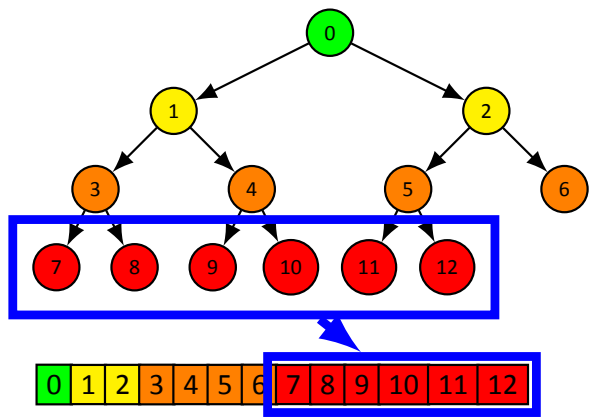
Heap

Lagra ett nästan komplett träd i en dynamisk array



Heap

Lagra ett nästan komplett träd i en dynamisk array



Heap

Lagra ett nästan komplett träd i en dynamisk array

- Det stora problemet med att lagra generella träd i en array är att det kan finnas stora luckor i trädet, d.v.s. att lövnoder kan i princip förekomma vart som helst.
- Men i ett nästan komplett träd *vet* vi att varje nivå är fylld, förutom potentiellt sista.
- Men den sista nivån fylls från vänster till höger.
- Så vad det innebär är att vi i princip kan spara varje lager i trädet i en array.
- Specifikt kan vi sedan lagra varje lager i sekvens i en dynamisk array som representerar hela grafen.

Heap

Lagra ett nästan komplett träd i en dynamisk array

Avbildning från träd till array

Vi avbildar ett nästan komplett träd till en array såhär:

- Rotnoden lagras på index 0
- Nod på index i lagrar sitt vänstra barn på index $2i + 1$
- Nod på index i lagrar sitt högra barn på index $2i + 2$
- Nod på index i har nod på index $\left\lfloor \frac{i-1}{2} \right\rfloor$ som förälder

Heap

Slutsats

- Med denna avbildning kan göra följande observationer:
- Insättning av ny lövnod längst till höger är ekvivalent med att stoppa in sist i en array $\Rightarrow \mathcal{O}(1)$ amorterat
- Borttagning av den sista lövnoden är ekvivalent med att ta bort sista elementet i en array $\Rightarrow \mathcal{O}(1)$ amorterat
- Med detta kan vi till sist färdigställa vår prioritetskö

Heap

Heaps

Definition: Heap

- En *min-heap* är ett binärt träd $\mathcal{H}$ där:
  - (i) $\mathcal{H}$ är ett *nästan komplett* träd
  - (ii) Varje nod i $\mathcal{H}$ är *mindre* än båda sina barn
- En *max-heap* är ett binärt träd $\mathcal{H}$ där:
  - (i) $\mathcal{H}$ är ett *nästan komplett* träd
  - (ii) Varje nod i $\mathcal{H}$ är *större* än båda sina barn

Heap

Heaps, egenskaper

- Hitta minsta/största: $\mathcal{O}(1)$
- Insättning: $\mathcal{O}(\log n)$
- Borttagning: $\mathcal{O}(\log n)$

Heap

Implementation

```
1  class Min_Heap
2  {
3  public:
4
5    Min_Heap() = default;
6
7    void push(int value);
8
9    int pop_min();
10   int peek_min() const;
11
12 private:
13
14   void sift_down(unsigned index);
15   void sift_up(unsigned index);
16
17   vector<int> values;
18 };
```

Heap

Implementation: Hjälpfunktioner

```
1  unsigned left(unsigned index)
2  {
3    return 2*index + 1;
4  }
5
6  unsigned right(unsigned index)
7  {
8    return 2*index + 2;
9  }
10
11 unsigned parent(unsigned index)
12 {
13   return (index - 1) / 2;
14 }
```

Heap

Implementation: sift-down

```
1  void Min_Heap::sift_down(unsigned index)
2  {
3    if (index >= values.size()) return;
4    unsigned l { left (index) }; // hitta vänsterbarn
5    unsigned r { right(index) }; // hitta högerbarn
6
7    // om vänster inte finns så finns inga barn
8    if (l >= values.size()) return;
9
10   unsigned next { l }; // hitta det minsta barnet
11   if (r < values.size() && values[r] < values[l])
12     next = r;
13
14   // om vi redan är mindre än vårt minsta barn är vi klara
15   if (values[index] < values[next]) return;
16
17   // annars byter vi plats med minsta barnet och rekurserar
18   swap(values[index], values[next]);
19   sift_down(next);
20 }
```

Heap

Implementation: sift-up

```
1  void Min_Heap::sift_up(unsigned index)
2  {
3    if (index == 0 || index >= values.size()) return;
4
5    // hitta föräldern
6    unsigned p { parent(index) };
7
8    // om heap egenskapen redan är uppfylld är vi klara
9    if (values[p] < values[index]) return;
10
11   // annars byter vi värdena och rekurserar
12   swap(values[p], values[index]);
13   sift_up(p);
14 }
```

Heap

Implementation: Insättning, borttagning och hitta

```
1  void Min_Heap::push(int value)
2  {
3    values.push_back(value); // lägg till som sista lövnod
4    sift_up(values.size() - 1); // flytta uppåt tills det är en heap igen
5  }
6
7  int Min_Heap::pop_min()
8  {
9    int min { values.front() }; // spara det minsta värdet
10   swap(values.front(), values.back()); // byt plats med sista lövnod
11   values.pop_back(); // ta bort sista lövnoden
12   sift_down(0); // flytta nedåt tills det är en heap igen
13
14 }
15
16 int Min_Heap::peek_min()
17 {
18   return values.front();
19 }
```

Heap

Avslut

- Kan man göra bättre?
- Ja, det finns t.ex. något som heter en *Fibonacci heap* där insättning blir $\mathcal{O}(1)$ amorterat.
- En Fibonacci heap utökar även funktionaliteten av prioritetskön till att kunna uppdatera prioriteten av redan befintliga nycklar, vilket den kan göra i $\mathcal{O}(1)$ amorterat.
- En Fibonacci heap är dock mer komplicerad att implementera och analysera.
- Fibonacci heaps bygger på den något simplare *Binomial heap*
- `https://www.cs.usfca.edu/~galles/visualization/FibonacciHeap.html`

www.liu.se

LINKÖPING UNIVERSITY