Algorithmen und Datenstrukturen
Kapitel 8
Graphen

Frank Heitmann
heitmann@informatik.uni-hamburg.de

2. Dezember 2015

Einführung

Graphen sind eine grundlegende Datenstruktur, die in vielen Bereichen der Informatik (und auch in anderen Bereichen) Anwendung findet. Man kann ohne Einschränkung zwei Elemente einer Mengen (den Knoten) in Beziehung setzen (durch eine Kante).

Anmerkung

Erlaubt man verschiedene Kanten-'Typen', so kann man sogar verschiedene Beziehungen ausdrücken.

Definitionen

Definition

Ein *Graph* ist ein Tupel $G = (V, E)$ bestehend aus einer Menge V (auch $V(G)$) von *Knoten* oder Ecken und einer Menge E (auch $E(G)$) von *Kanten*.
Ist G ein *ungerichteter Graph*, so ist

$$E \subseteq \{\{v_1, v_2\} \mid v_1, v_2 \in V, v_1 \neq v_2\},$$

ist G ein *gerichteter Graph*, so ist

$$E \subseteq V^2.$$

Ist $|E|$ viel kleiner als $|V|^2$, so nennt man den Graphen *dünn besetzt*. Ist $|E|$ nahe an $|V|^2$, so spricht man von *dicht besetzten* Graphen.

Gewichteter Graph

Definition

Bei einem *gewichteten Graphen* ist neben dem Graph $G = (V, E)$ (gerichtete oder ungerichtet) noch eine *Gewichtsfunktion* $w : E \to \mathbb{R}^+$ gegeben, die jeder Kante $e \in E$ ihre *Kosten* $w(e)$ zuweist.

Anmerkung

Diese Definition ist in der Datenstruktur-Vorlesung nicht explizit auf den Folien gewesen, ist aber sehr wichtig. (Die meisten Graphen haben Gewichte (oder Buchstaben o.ä.) an den Kanten.)

Definitionen

Definition

- Sind je zwei Knoten von G mit einer Kante verbunden, so ist G ein *vollständiger Graph*. Bei n Knoten: K^n.
- Eine Menge paarweise nicht benachbarter Knoten nennt man *unabhängig*.
- Der *Grad* $d(v)$ eines Knotens v ist die Anzahl mit v inzidenter Kanten.
- Die Menge der *Nachbarn* eines Knotens v bezeichnet man mit $N(v)$ (hier gilt $d(v) = |N(v)|$).
- $\delta(G)$ ist der *Minimalgrad* von G, $\Delta(G)$ der *Maximalgrad*.

Definitionen

Definition

- Ein *Weg* ist ein nicht leerer Graph $P = (V, E)$ mit $V = \{x_0, x_1, \ldots, x_k\}$, $E = \{x_0x_1, x_1x_2, \ldots, x_{k-1}x_k\}$, wobei die x_i paarweise verschieden sind. x_0 und x_k sind die Enden von P, sie sind durch P verbunden. Die Anzahl der Kanten eines Weges ist seine *Länge*.
- Ist P wie oben ein Weg, so ist $P + x_kx_0$ ein Kreis (der Länge $k + 1$).
- Der Abstand zweier Knoten x und y voneinander wird mit $d(x, y)$ bezeichnet und ist die geringste Länge eines x-y-Weges.

Definitionen

Definition

- Seien $G = (V, E)$ und $G' = (V', E')$ Graphen. Gilt $V' \subseteq V$ und $E' \subseteq E$, so nennt man G' einen *Teilgraphen* von G.
- Ist $G = (V, E)$ ein Graph und $V' \subseteq V$, so nennt man den Graphen $G' = (V', E')$ mit $E' = \{\{v_1, v_2\} \in E \mid v_1, v_2 \in V'\}$ den von V' *induzierten Graphen*.

Darstellung von Graphen

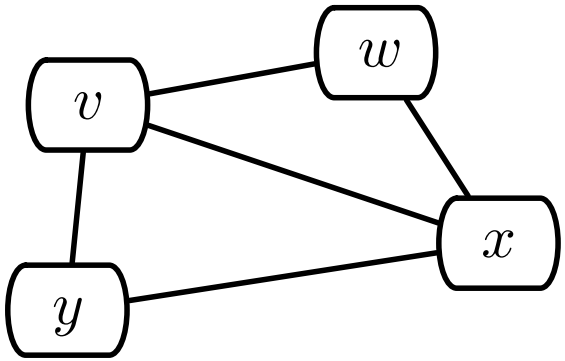
Ein Graph $G = (V, E)$ wird dargestellt indem man seine Knoten als Punkte und die Tupel oder Mengen aus E als (gerichtete) Kanten zwischen die Knoten einzeichnet.

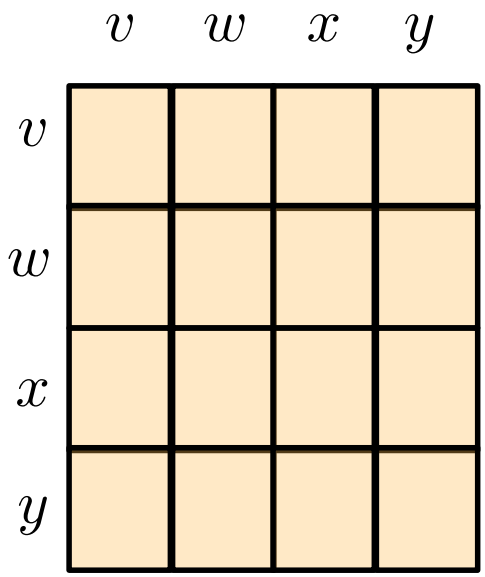
Im Computer speichert man einen Graphen meist mittels einer *Adjazenzmatrix* oder einer *Adjazenzliste*. (Man kann die Mengen V und E aber auch direkt speichern.)

Anmerkung

Bei Graphen schreibt man (und wir) oft $O(V + E)$ etc., wenn $O(|V| + |E|)$ gemeint ist. Man beacht zudem, dass dies die Komplexität bzgl. der Kenngrößen V und E ausdrückt und nicht umbedingt die Größe der Eingabe wiederspiegelt!

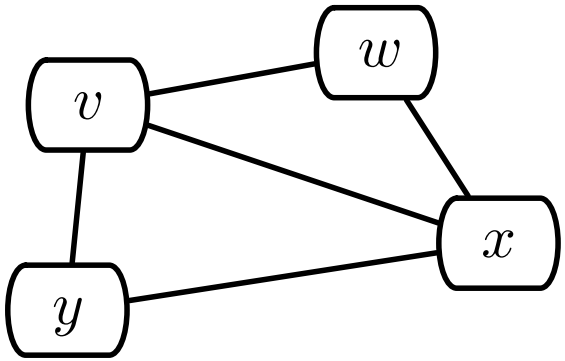
Darstellung von Graphen - Adjazenzmatrix





$$V = \{v, w, x, y\}$$
$$E = \{\{v, w\}, \{v, x\}, \{v, y\}, \{w, x\}, \{x, y\}\}$$

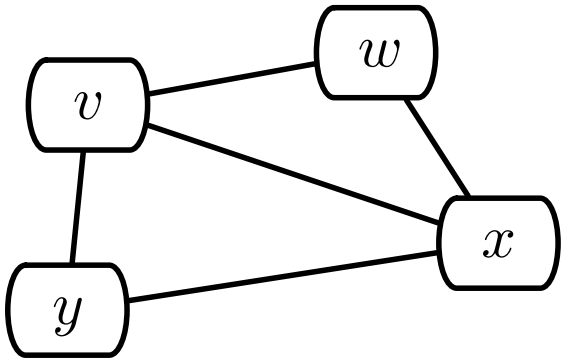
Darstellung von Graphen - Adjazenzmatrix



	v	w	x	y
v	0			
w		0		
x			0	
y				0

$$V = \{v, w, x, y\}$$
$$E = \{\{v, w\}, \{v, x\}, \{v, y\}, \{w, x\}, \{x, y\}\}$$

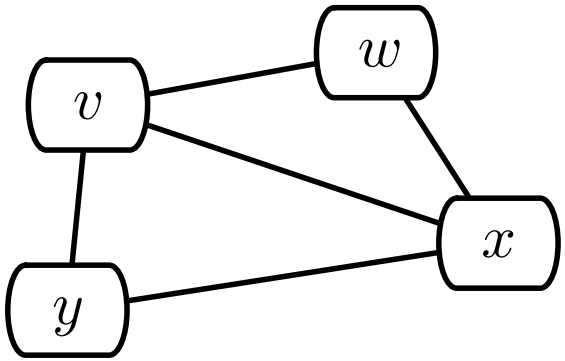
Darstellung von Graphen - Adjazenzmatrix



	v	w	x	y
v	0	1	1	1
w	1	0		
x	1		0	
y	1			0

$$V = \{v, w, x, y\}$$
$$E = \{\{v, w\}, \{v, x\}, \{v, y\}, \{w, x\}, \{x, y\}\}$$

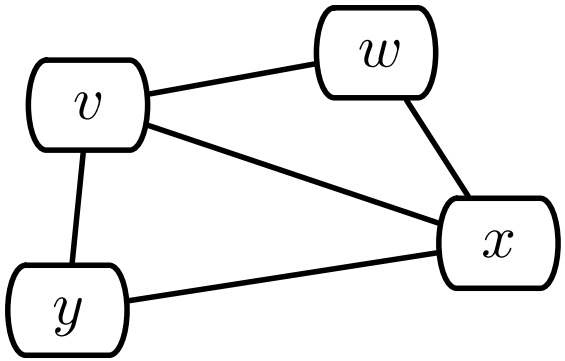
Darstellung von Graphen - Adjazenzmatrix



	v	w	x	y
v	0	1	1	1
w	1	0	1	0
x	1	1	0	1
y	1	0	1	0

$$V = \{v, w, x, y\}$$
$$E = \{\{v, w\}, \{v, x\}, \{v, y\}, \{w, x\}, \{x, y\}\}$$

Darstellung von Graphen - Adjazenzmatrix

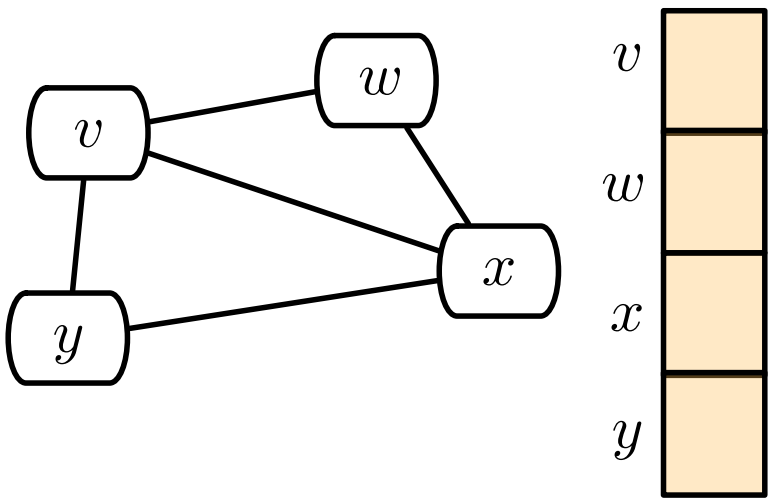


	v	w	x	y
v	0	1	1	1
w	1	0	1	0
x	1	1	0	1
y	1	0	1	0

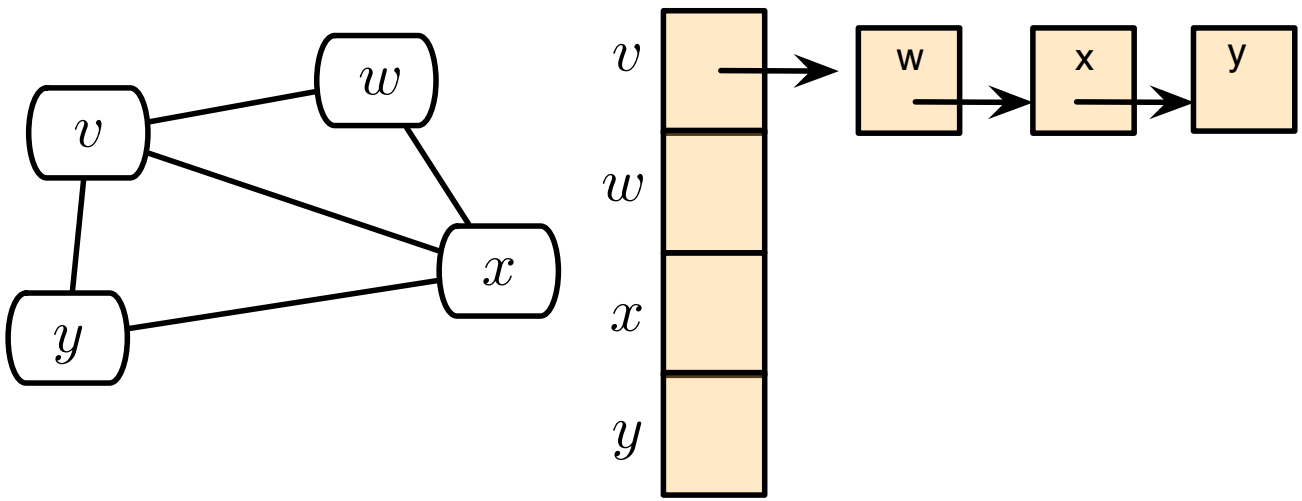
Bei einer Adjazenzmatrix hat man eine $n \times n$-Matrix, bei der an der Stelle (i, j) genau dann eine 1 steht, wenn v_i und v_j verbunden sind.

Der Speicherplatzbedarf ist in $\Theta(V^2)$ (unabhängig von der Kantenzahl).

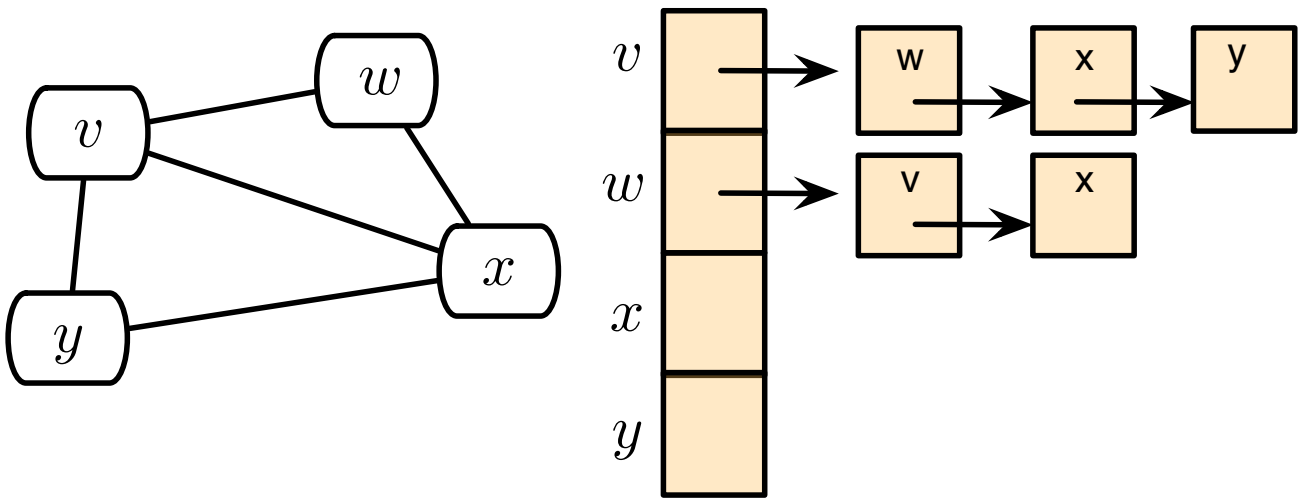
Darstellung von Graphen - Adjazenzlisten



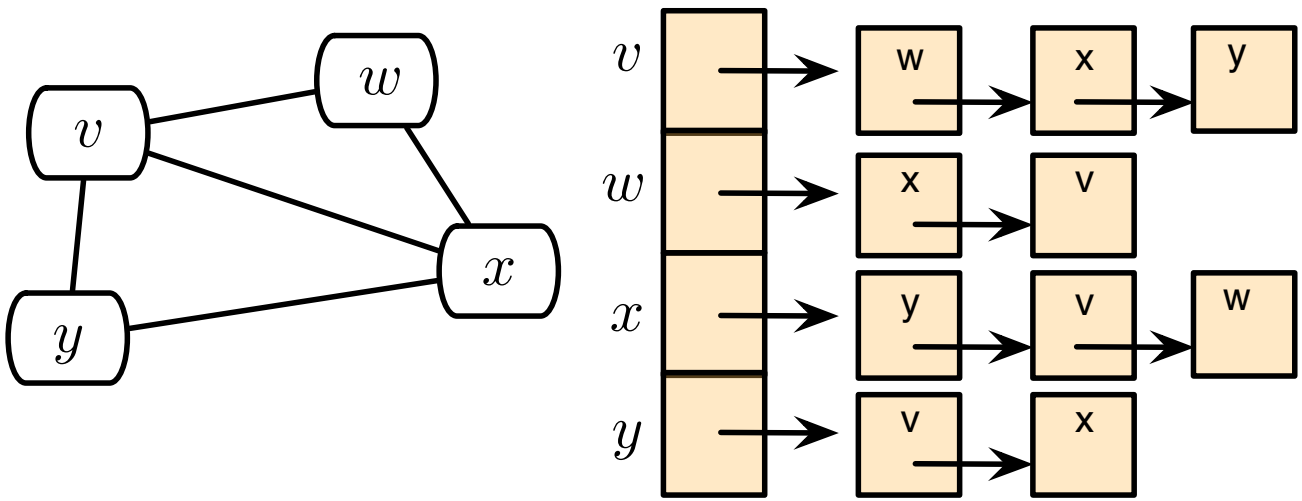
Darstellung von Graphen - Adjazenzlisten



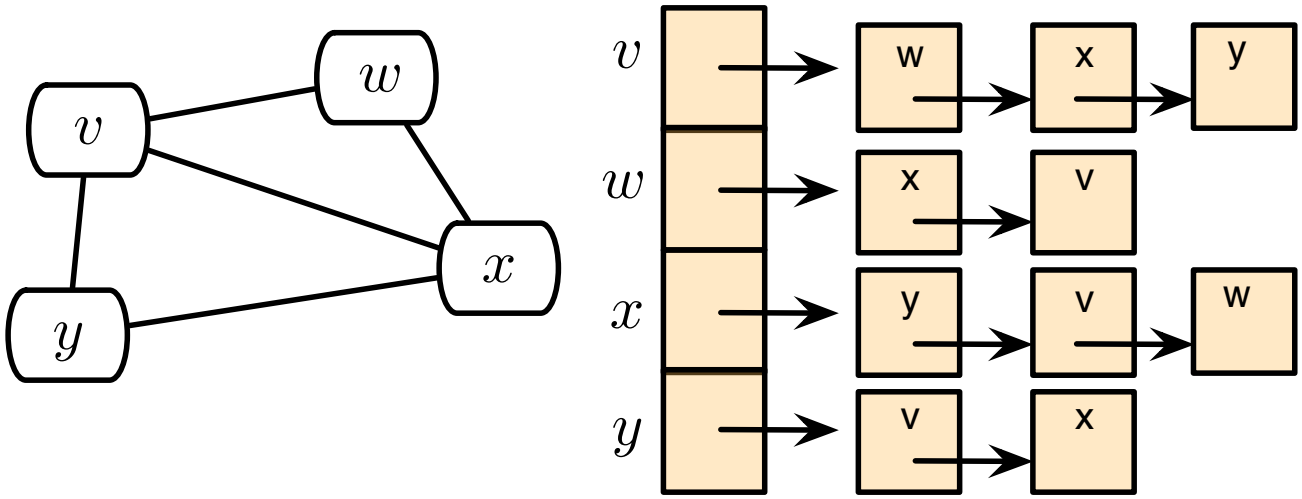
Darstellung von Graphen - Adjazenzlisten



Darstellung von Graphen - Adjazenzlisten



Darstellung von Graphen - Adjazenzlisten



Bei der Adjazenzlistendarstellung haben wir ein Array von $|V|$ Listen, für jeden Knoten eine. Die Adjazenzliste $Adj[v]$ zu einem Knoten v enthält alle Knoten, die mit v adjazent sind.

Bei einem gerichteten Graphen ist die Summe aller Adjazenzlisten $|E|$, bei einem ungerichteten Graphen $|2E|$. Der Speicherplatzbedarf ist folglich $\Theta(V + E)$.

Darstellung von Graphen - Zusammenfassung

- Adjazenzmatrix: $|V| \times |V|$-Matrix $A = (a_{ij})$ mit $a_{ij} = 1$ falls $(i, j) \in E$ und 0 sonst. Größe in $\Theta(V^2)$.
- Adjazenzliste: Liste $Adj[v]$ für jeden Knoten $v \in V$ in der die Knoten, die mit v adjazent sind gespeichert sind. Größe in $\Theta(V + E)$.
- Bei einer Adjazenzmatrix kann man schnell herausfinden, ob zwei Knoten benachbart sind oder nicht. Dafür ist es langsamer alle Knoten zu bestimmen, die mit einem Knoten benachbart sind. (Bei Adjazenzlisten genau andersherum.)
- Beide Darstellungen sind ineinander transformierbar.
- Beide Darstellungen sind leicht auf den Fall eines gewichteten Graphen anpassbar.

Die Struktur eines Graphen

Wir wollen nun etwas grundlegendes über die *Struktur* eines gegebenen Graphen erfahren. Hierzu ist es zunächst praktisch Algorithmen zu haben, die den Graphen *durchwandern*, d.h. in systematischer Weise die Kanten entlangwandern und die Knoten besuchen. Oft ist es auch nützlich einen *Spannbaum* zu ermitteln.

Definition

Ein Spannbaum ist ein Teilgraph T eines Graphen G, wobei T ein Baum ist und alle Knoten von G enthält.

Breiten- und Tiefensuche

Die *Breiten-* und *Tiefensuche* in einem Graphen erreichen (im Prinzip) beide Ziele! Es wird zwar i.A. nicht der minimale Spannbaum ermittelt, doch dazu später mehr.

Breiten- und Tiefensuche sind oft 'Urtypen' für weitere Graphalgorithmen entweder

- als wichtige Subroutine oder
- als 'Ideengeber'

Breitensuche - Die Idee

Gegeben ein Graph G und ein Startknoten s 'entdeckt' die Breitensuche alle Knoten, die von s aus erreichbar sind. Zudem wird der Abstand (in Kanten) von s aus berechnet (tatsächlich wird sogar der *kürzeste Abstand* ermittelt).

Man kann auch zusätzlich einen 'Breitensuchbaum' (mit Wurzel s) und damit die kürzesten Pfade von s zu den anderen Knoten ermitteln.

Der Algorithmus entdeckt zunächst die Knoten mit Entfernung k und dann die mit Entfernung $k + 1$, daher der Name. Er geht erst in die Breite...

Breitensuche - Die Idee

- Starte mit Knoten s in einer Queue Q.
- Wiederhole solange Q nicht leer...
  - Nimm vordersten Knoten v aus Q.
  - (Bearbeite diesen und Färbe diesen so, dass er nicht wieder besucht wird.)
  - Tue alle Nachbarn von v, die bisher nicht besucht wurden in die Queue.

Anmerkung

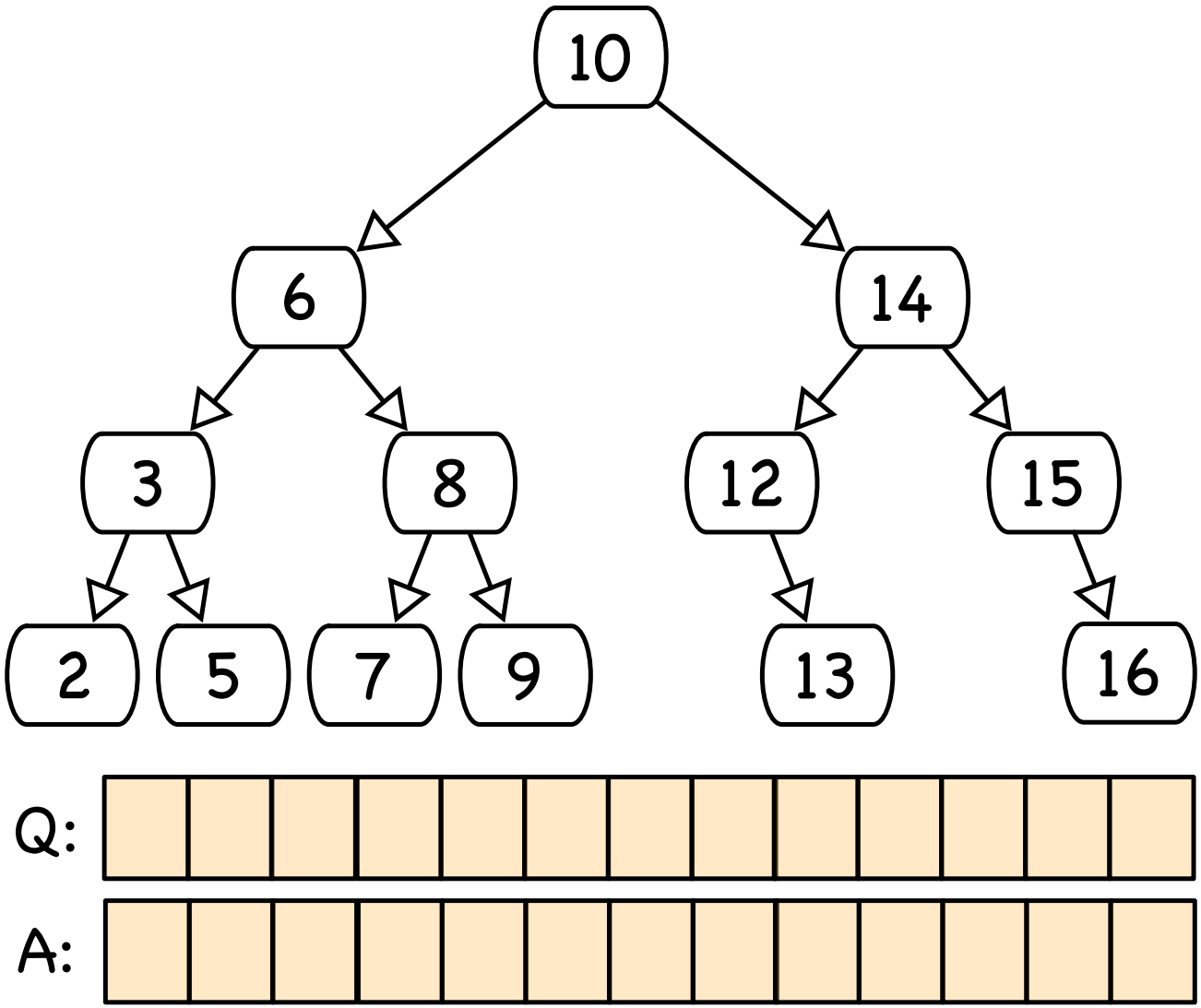
Durch die Queue wird sichergestellt, dass die Knoten 'in der Breite' besucht werden, d.h. zunächst werden die Knoten mit Entfernung k von s entdeckt und erst danach jene mit Entfernung $k + 1$.

Zur Nachbereitung

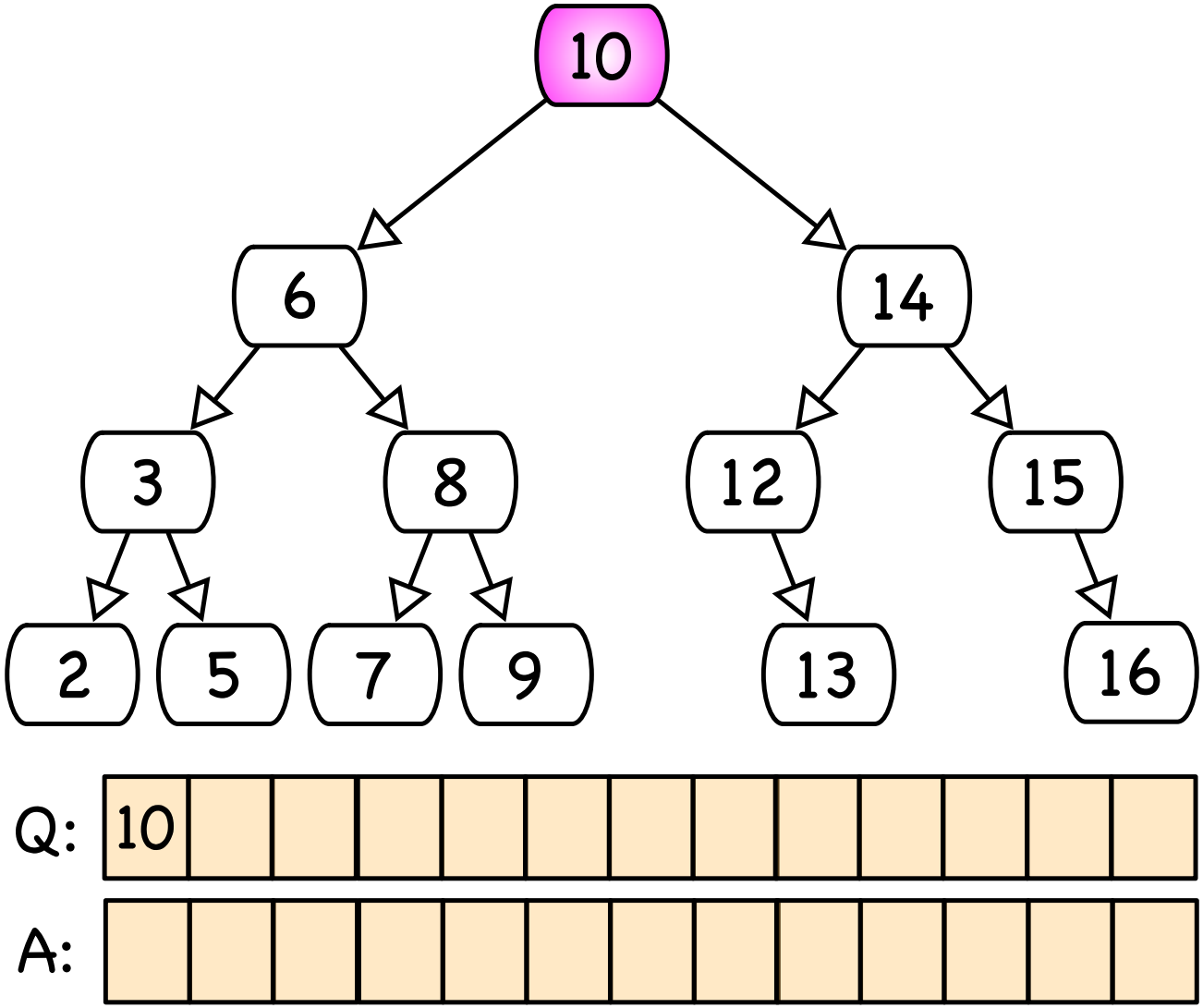
Anmerkung (zur Nachbereitung)

Im nachfolgenden Beispiel ist für einen Knoten v jeweils sein linkes Kind der erste Knoten in v's Adjazenzliste, sein rechtes Kind der zweite Knoten in der Adjazenzliste. (Dies ist wichtig, um zu verstehen in welcher Reihenfolge die Knoten in der Queue gespeichert werden.)
Für das Beispiel ist es ferner unerheblich, ob die Kanten gerichtet oder ungerichtet sind. Bei ungerichteten Kanten wäre auch stets noch der Vater in der Adjazenzliste eines Knotens. Dieser wäre jedoch bereits gefärbt und würde nicht wieder in der Queue gespeichert werden.

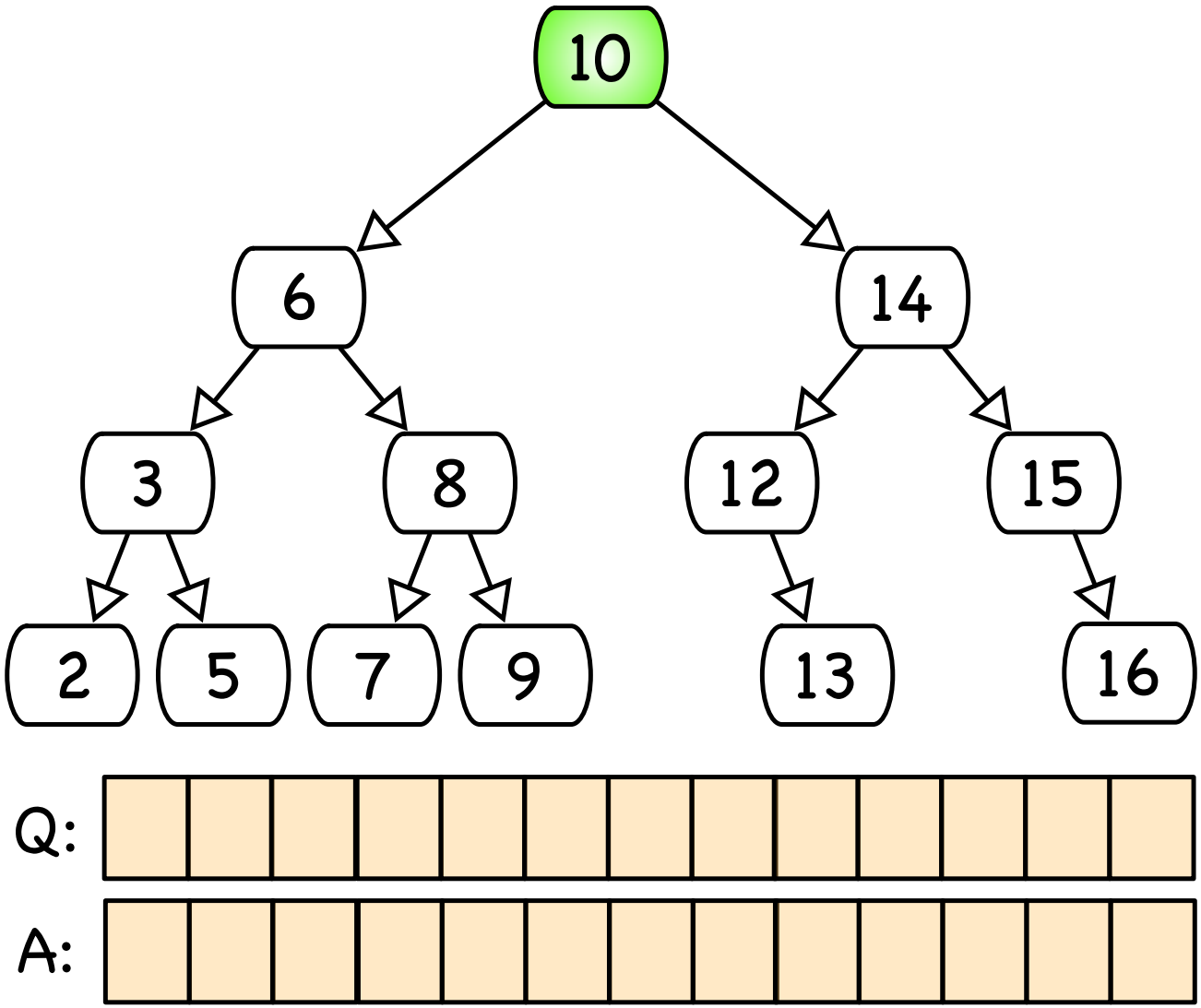
Breitensuche - im Baum



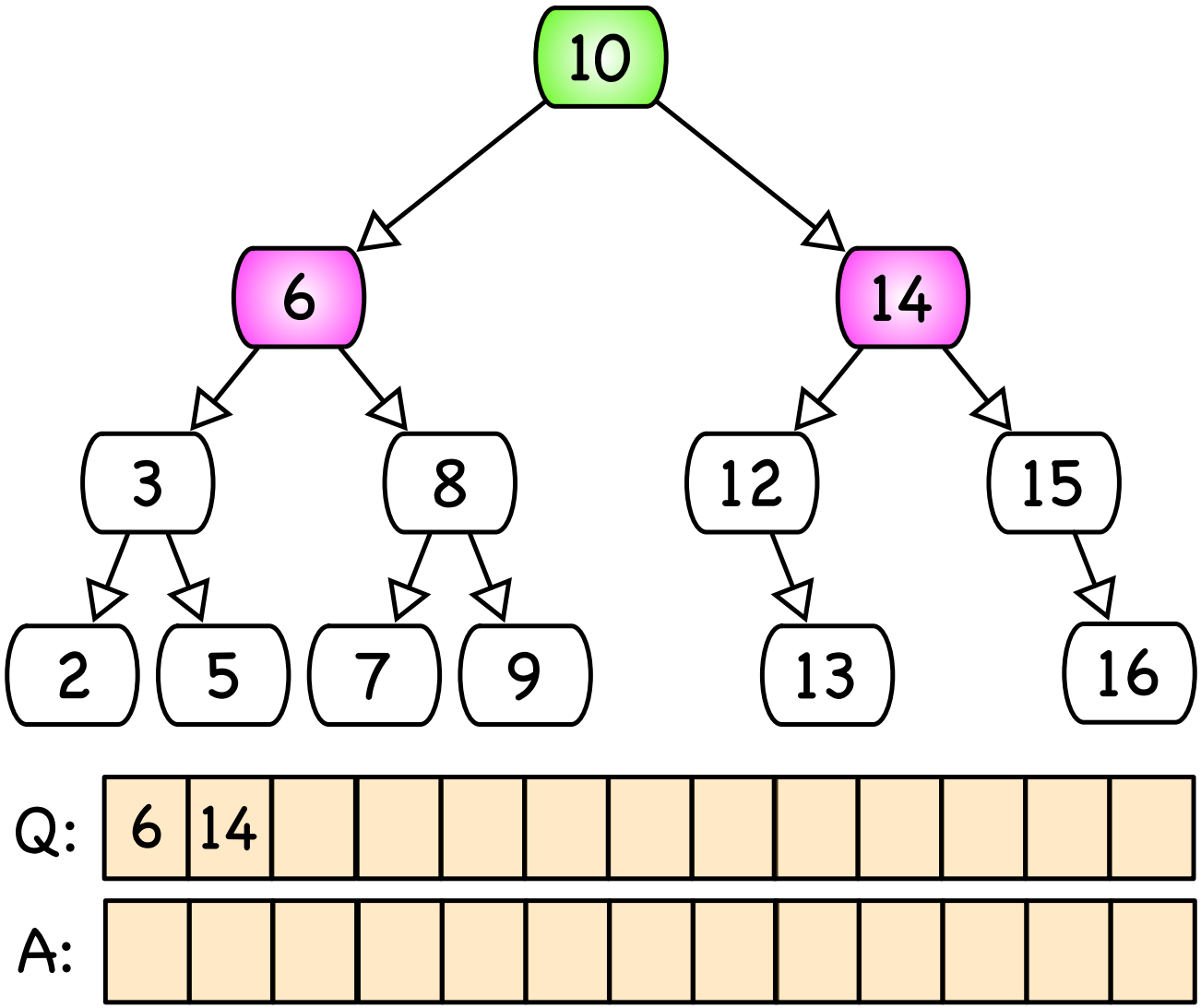
Breitensuche - im Baum



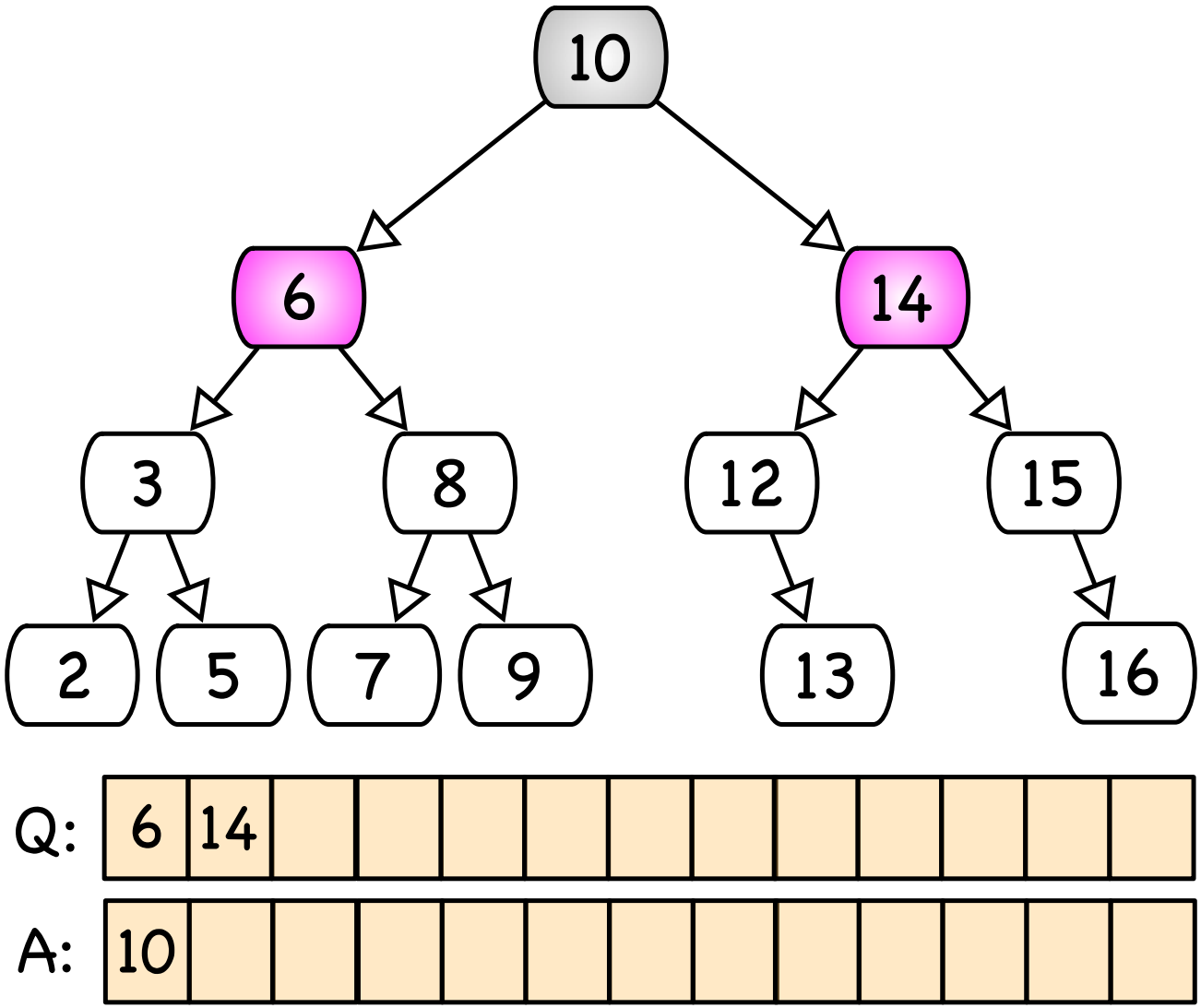
Breitensuche - im Baum



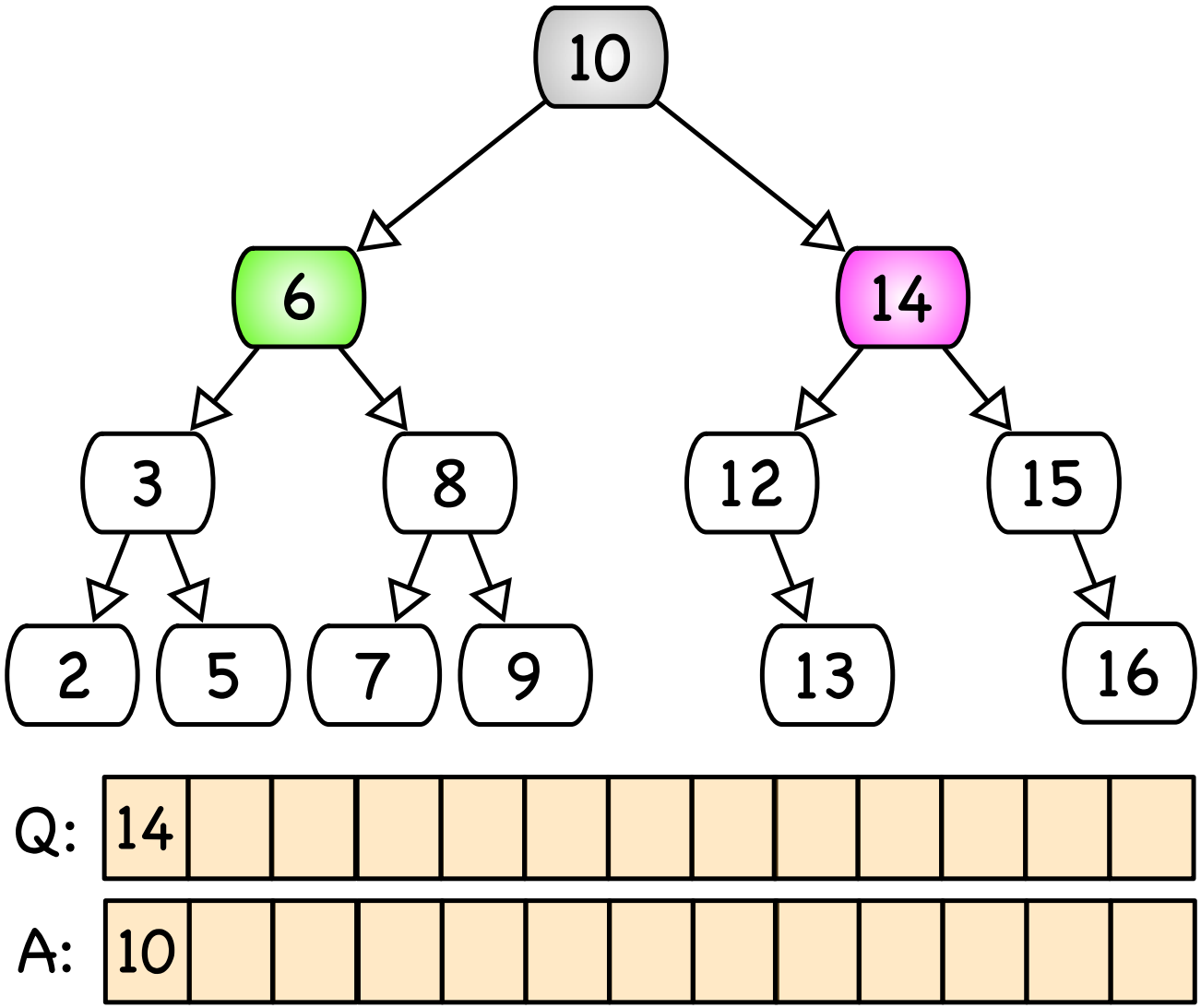
Breitensuche - im Baum



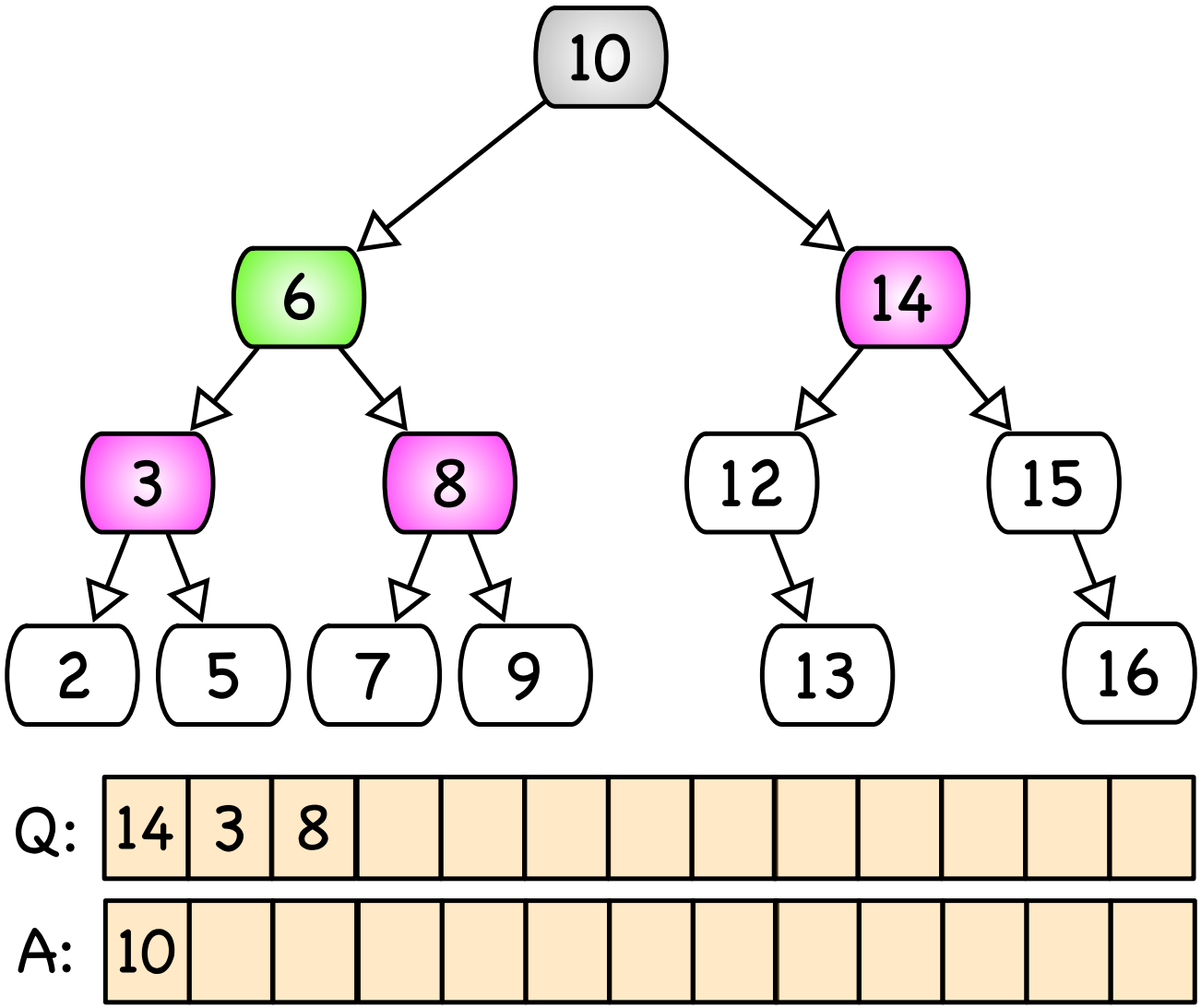
Breitensuche - im Baum



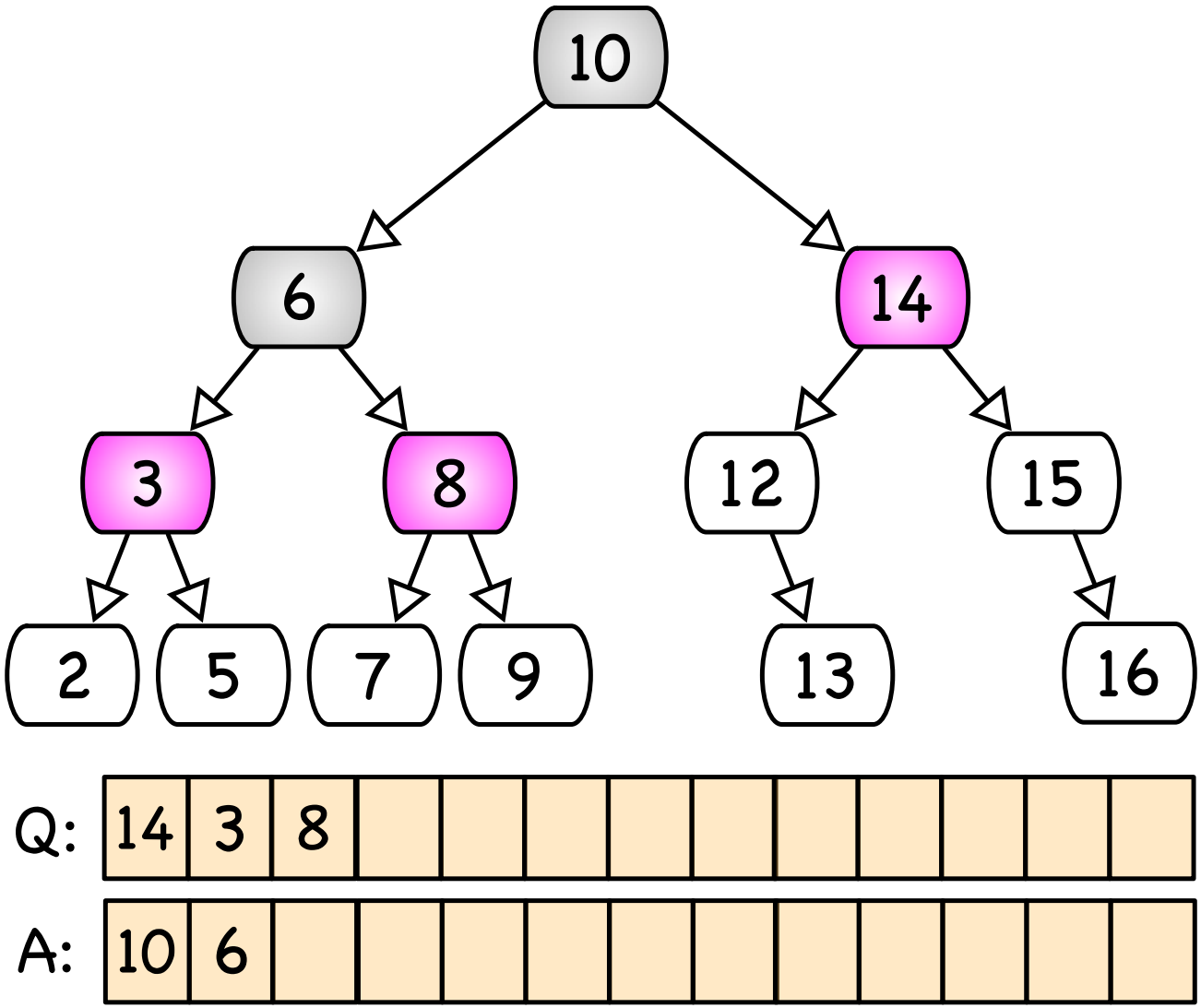
Breitensuche - im Baum



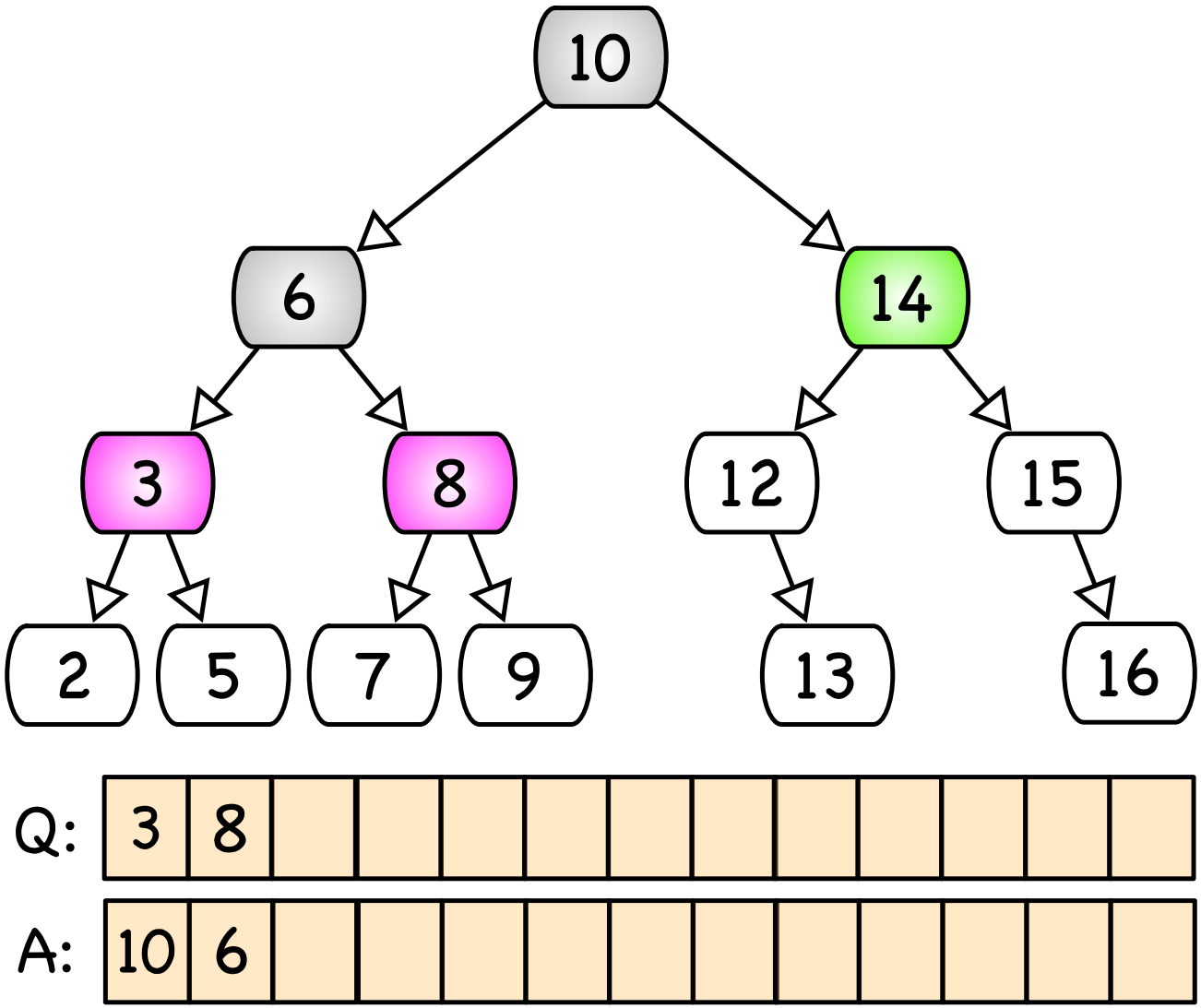
Breitensuche - im Baum



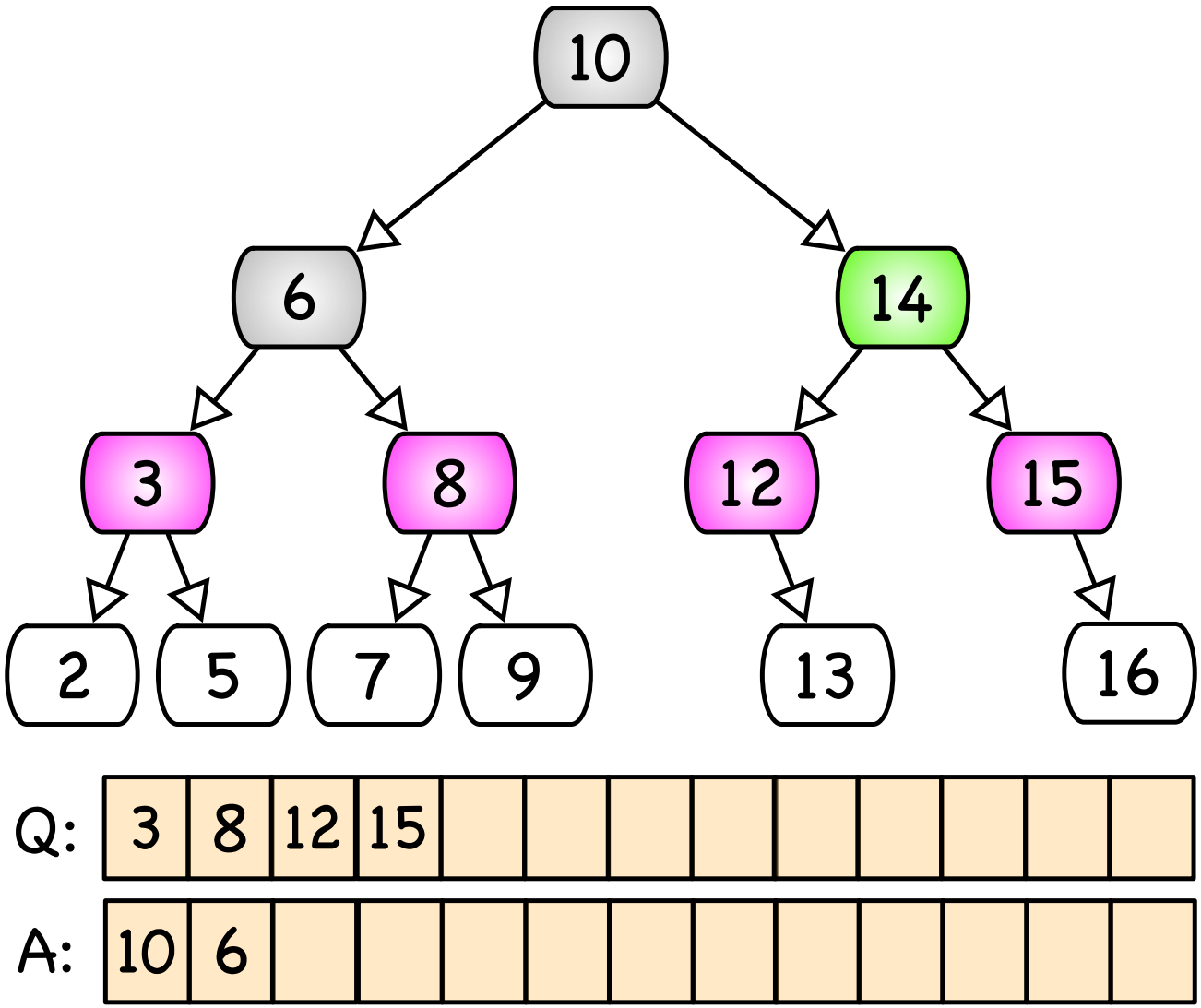
Breitensuche - im Baum



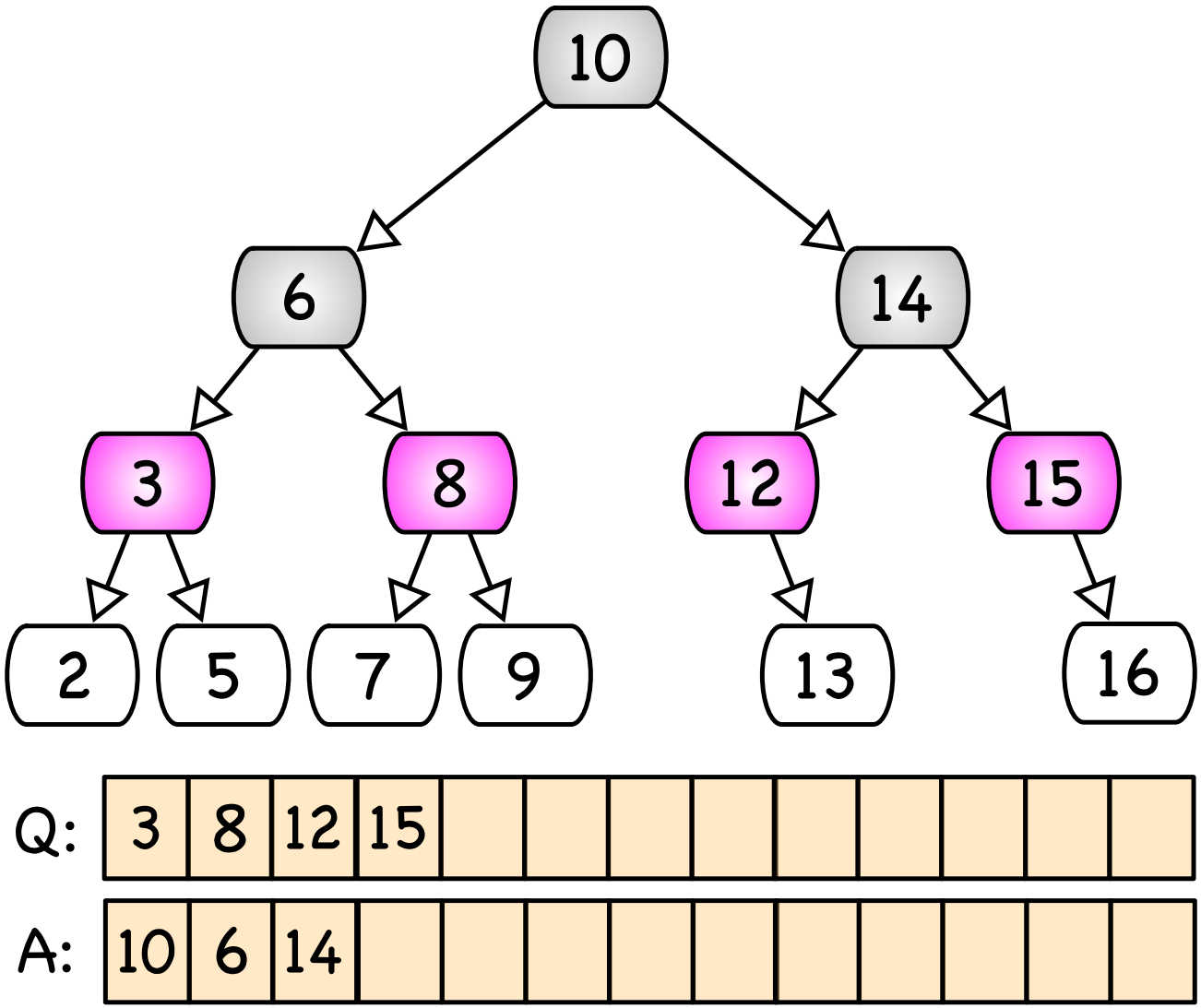
Breitensuche - im Baum



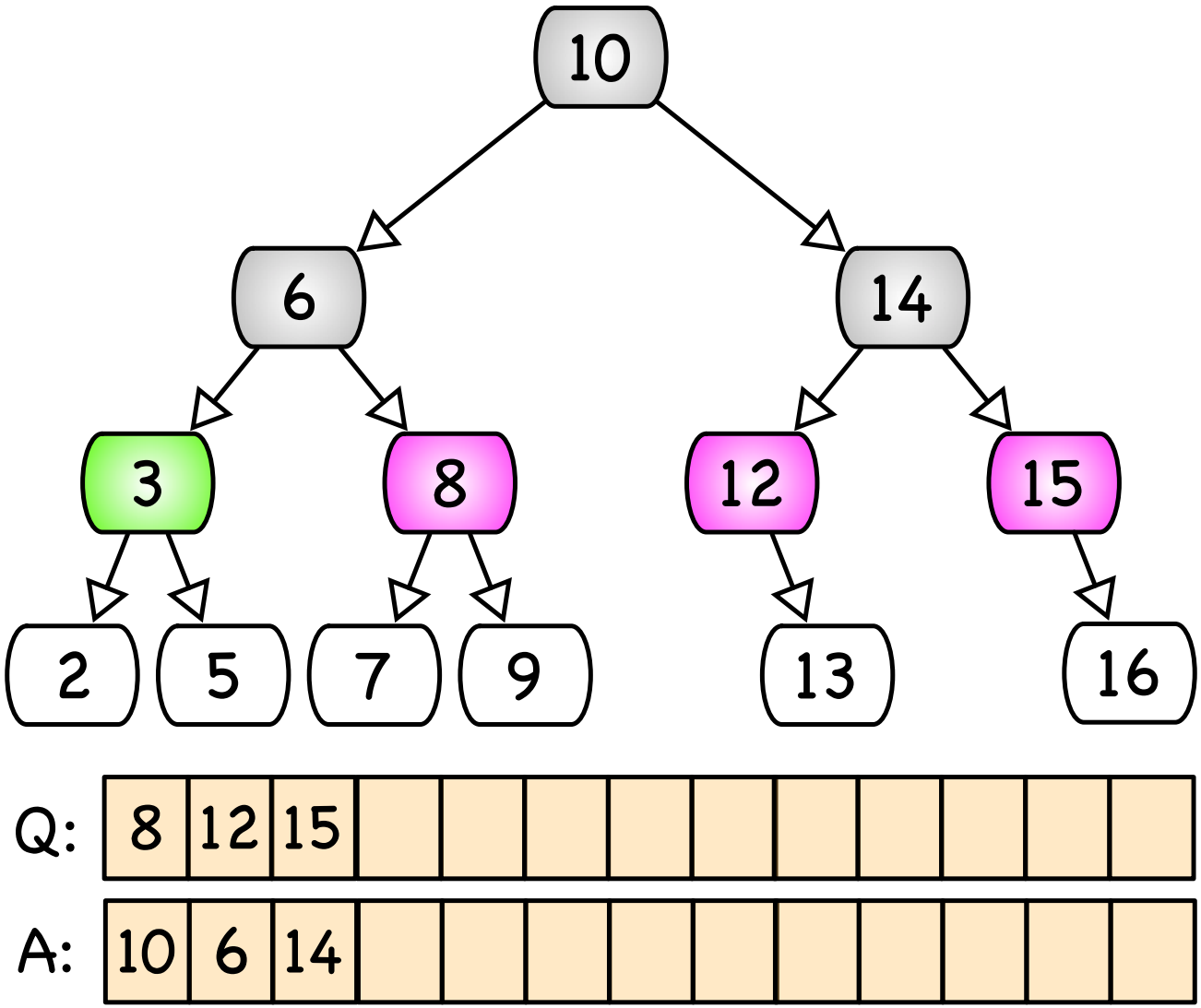
Breitensuche - im Baum



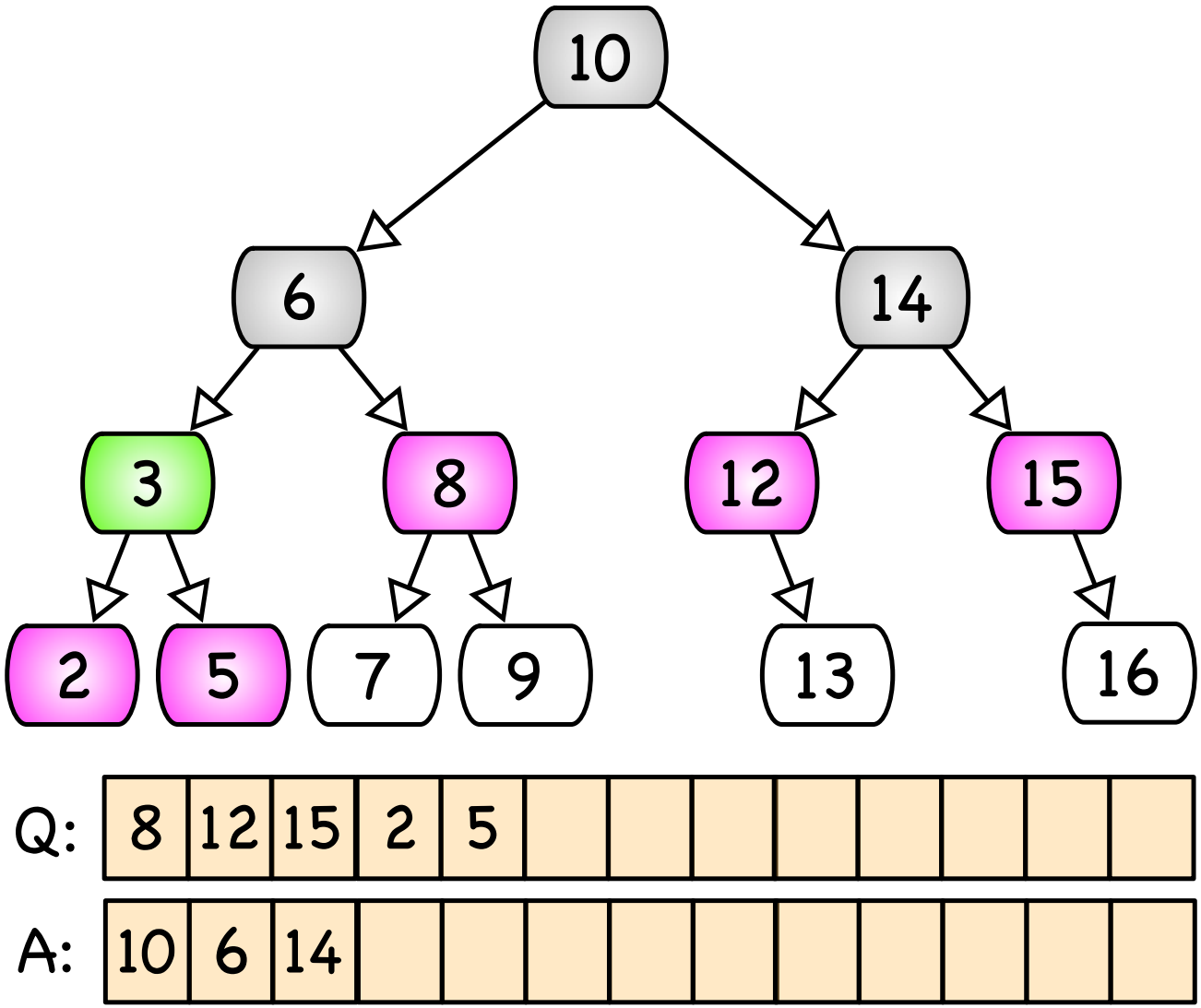
Breitensuche - im Baum



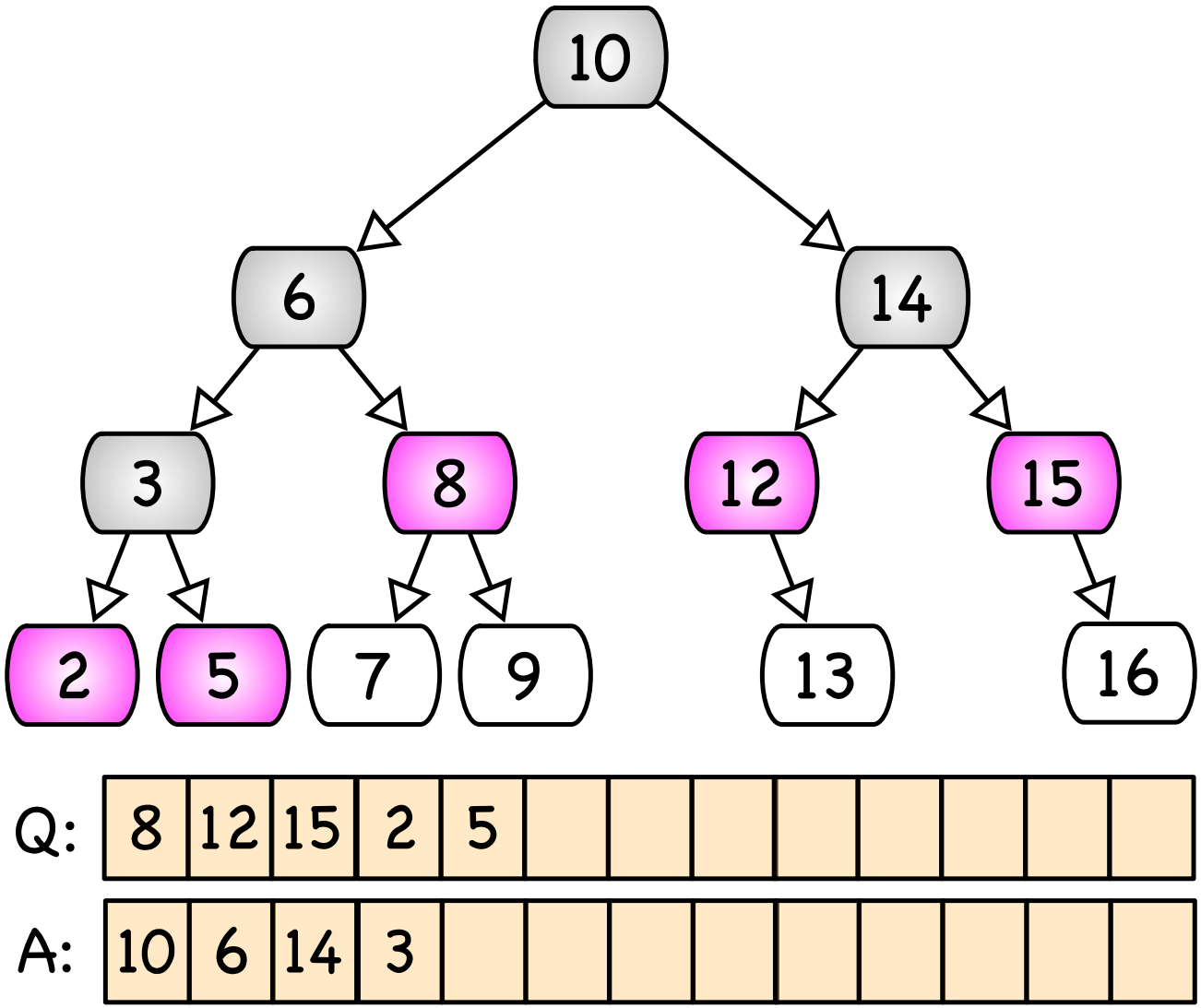
Breitensuche - im Baum



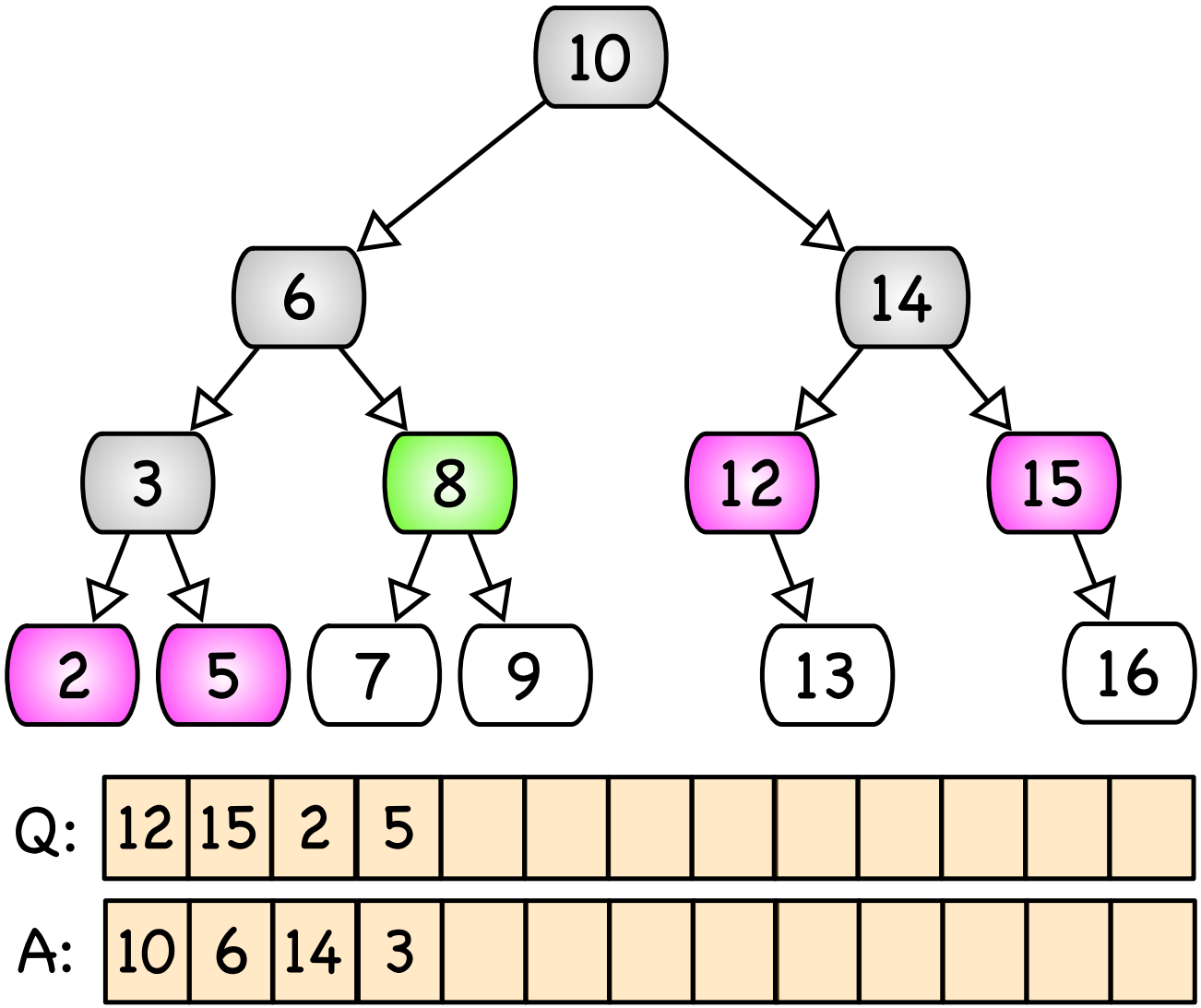
Breitensuche - im Baum



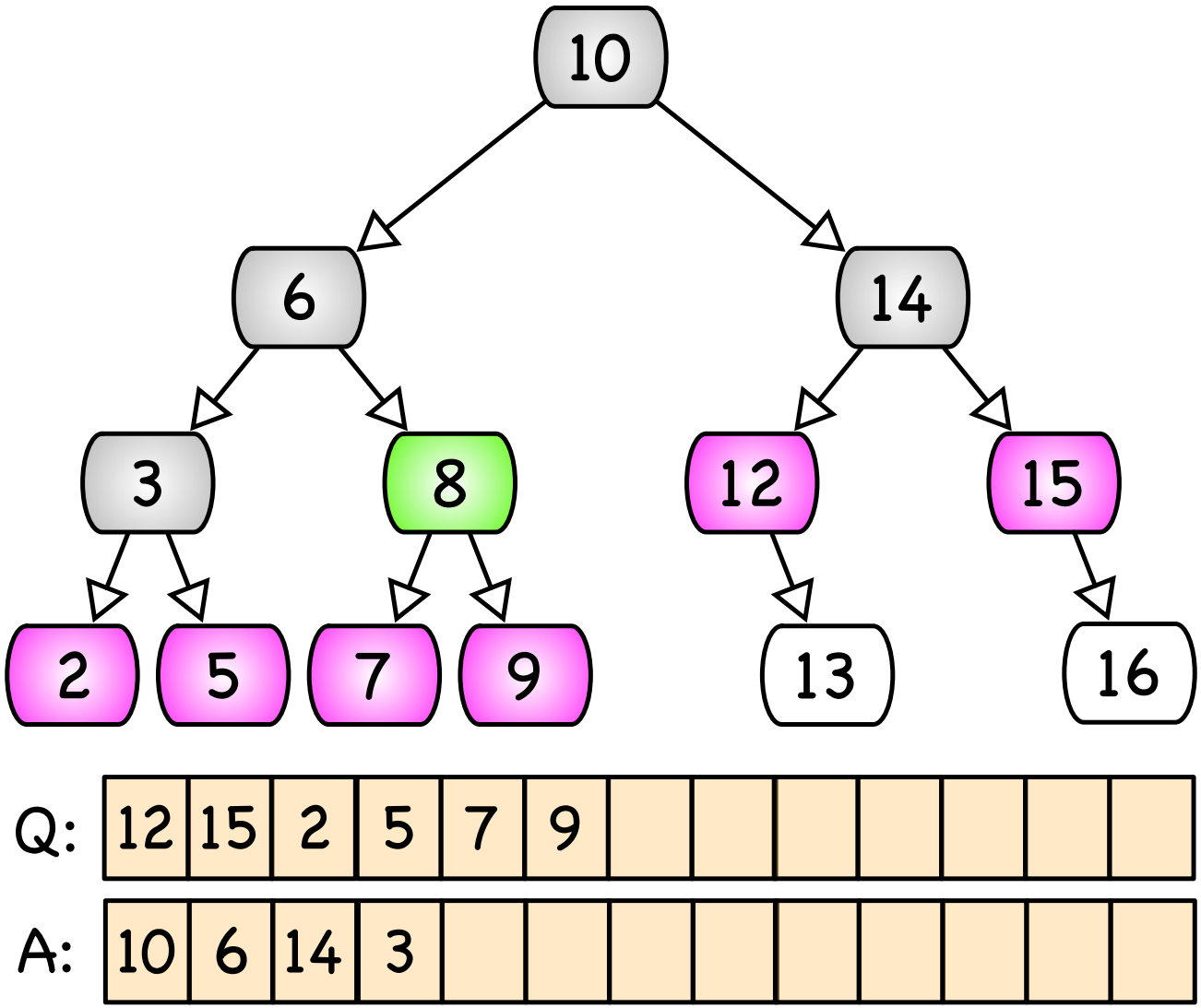
Breitensuche - im Baum



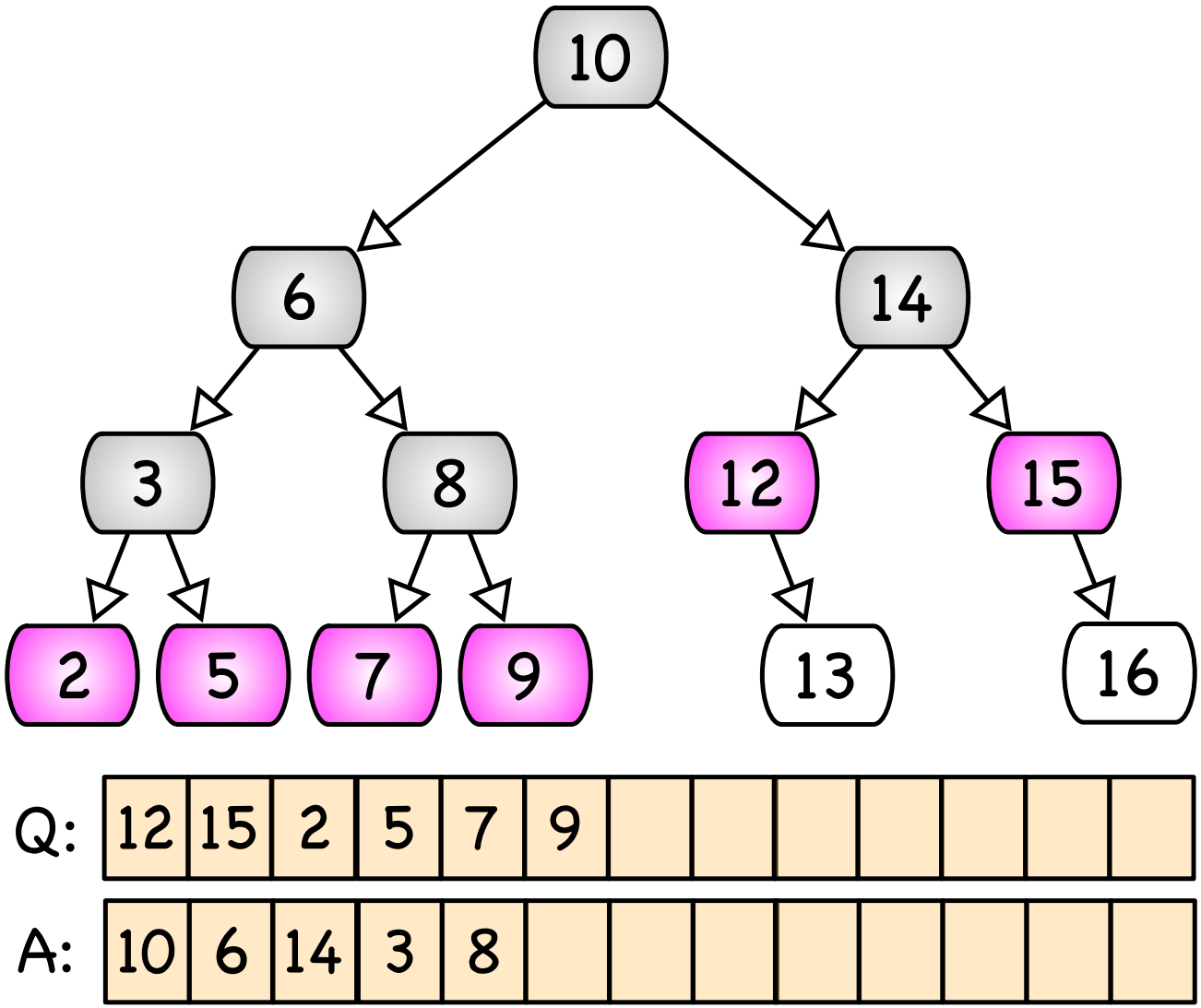
Breitensuche - im Baum



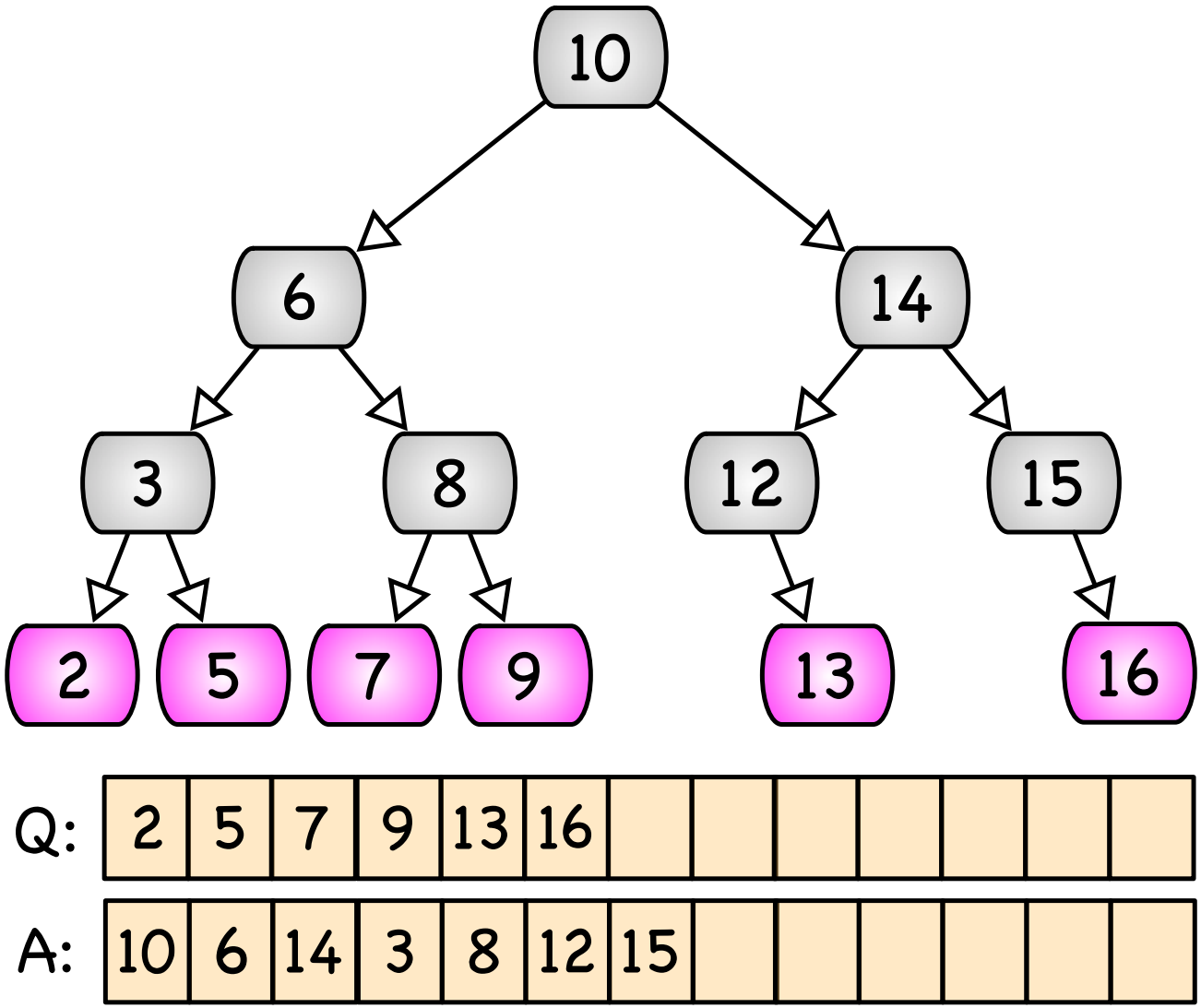
Breitensuche - im Baum



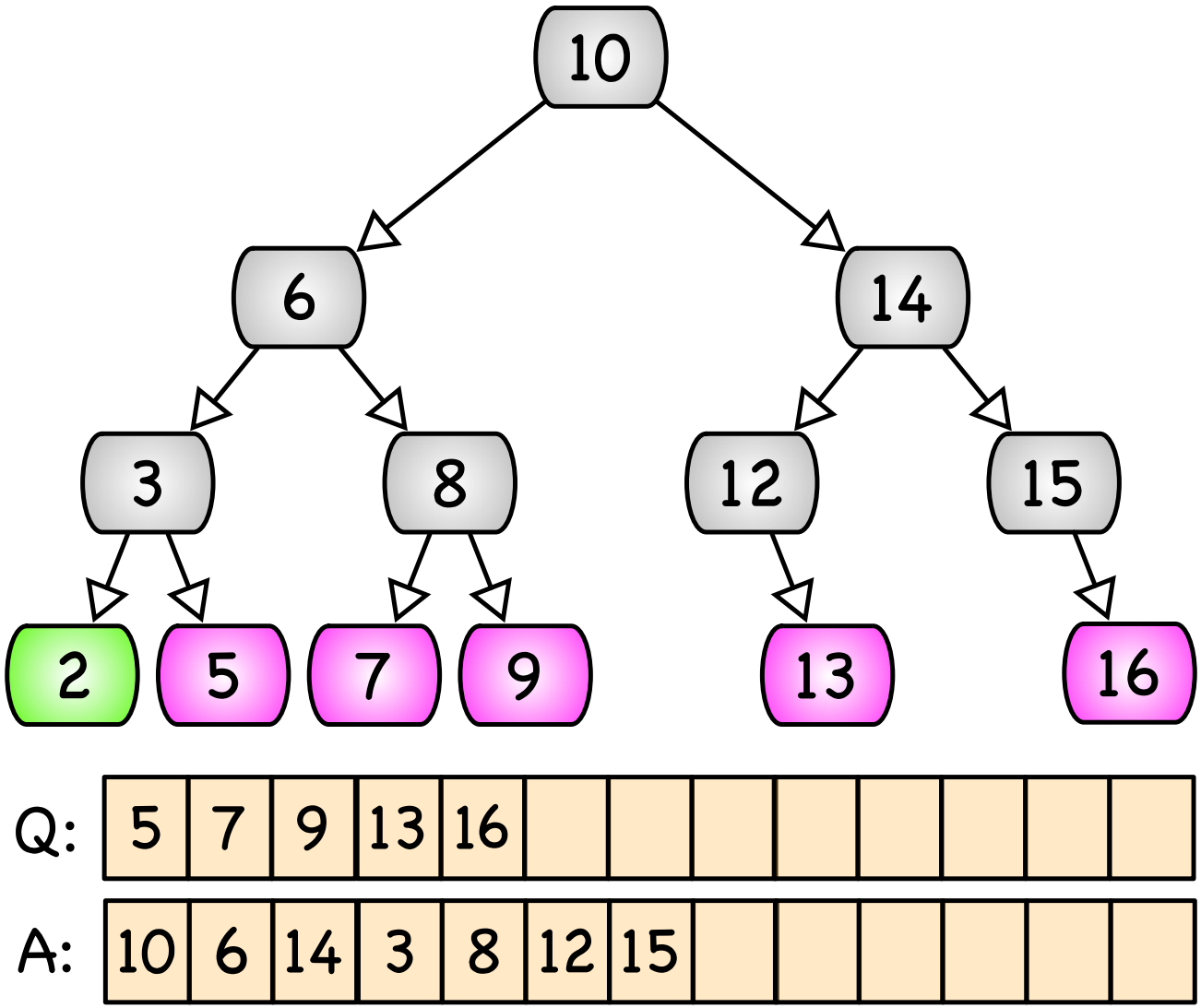
Breitensuche - im Baum



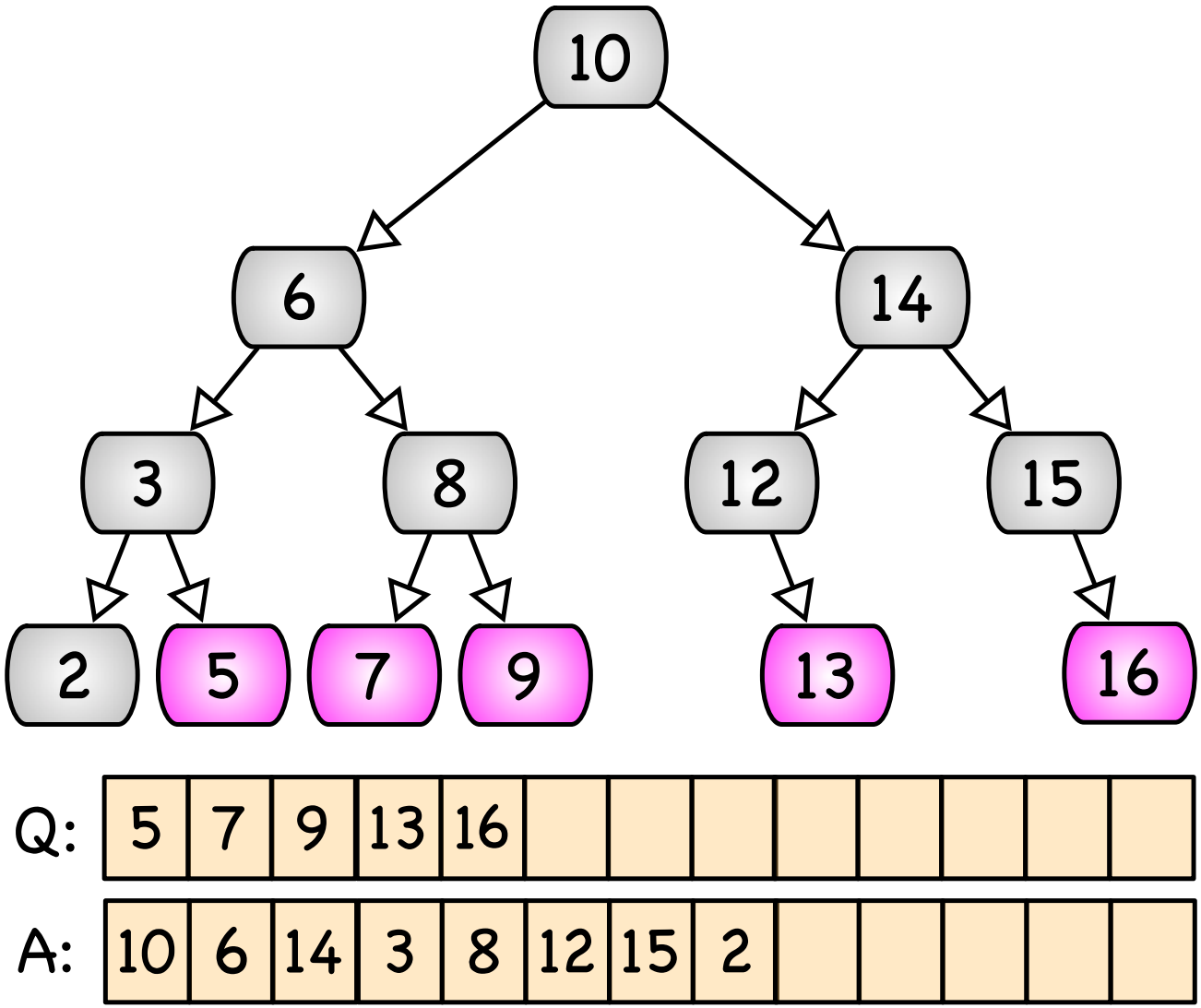
Breitensuche - im Baum



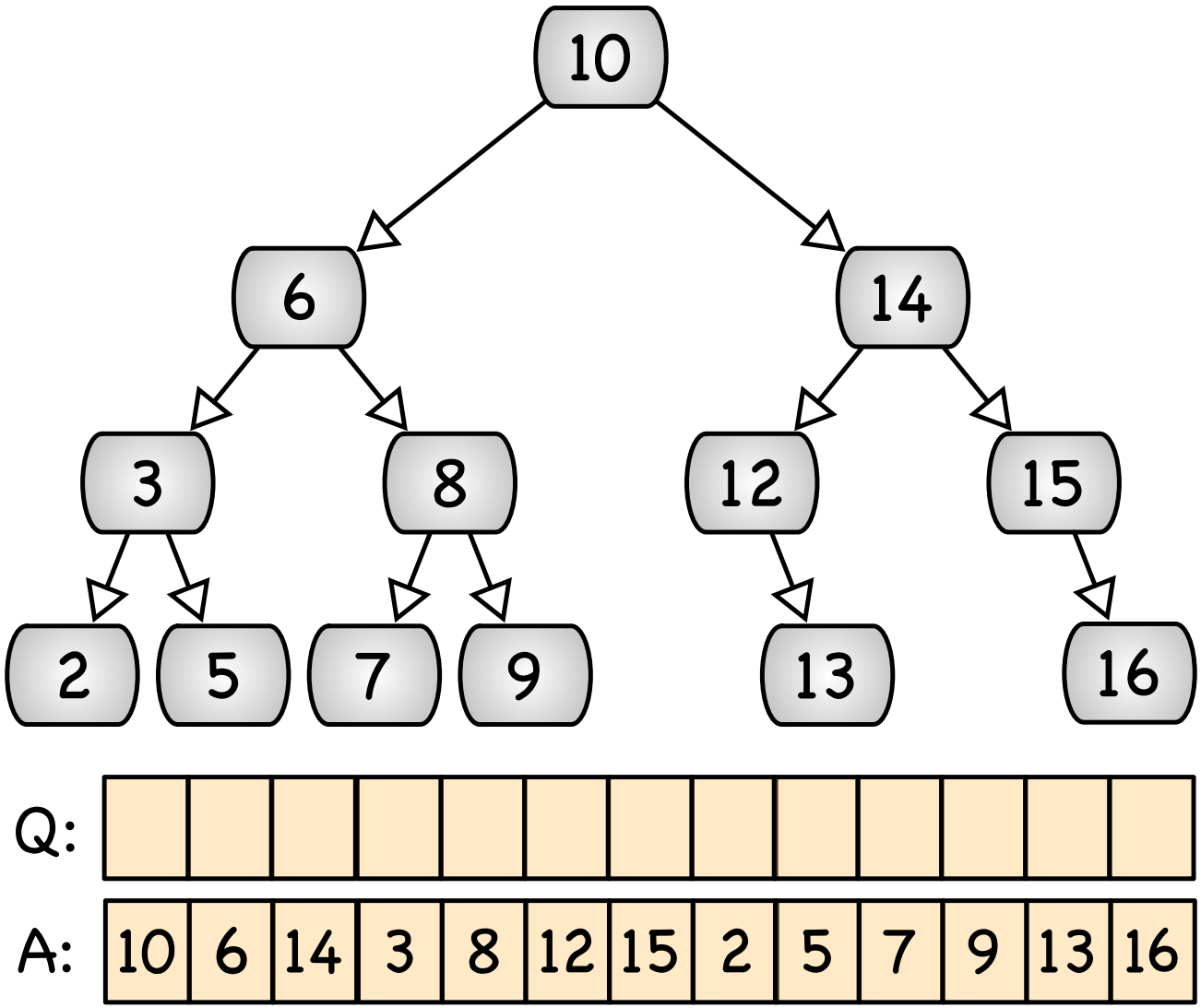
Breitensuche - im Baum



Breitensuche - im Baum



Breitensuche - im Baum



Breitensuche - Kernidee als Algorithmus

Algorithmus 1 BFS(G, s)

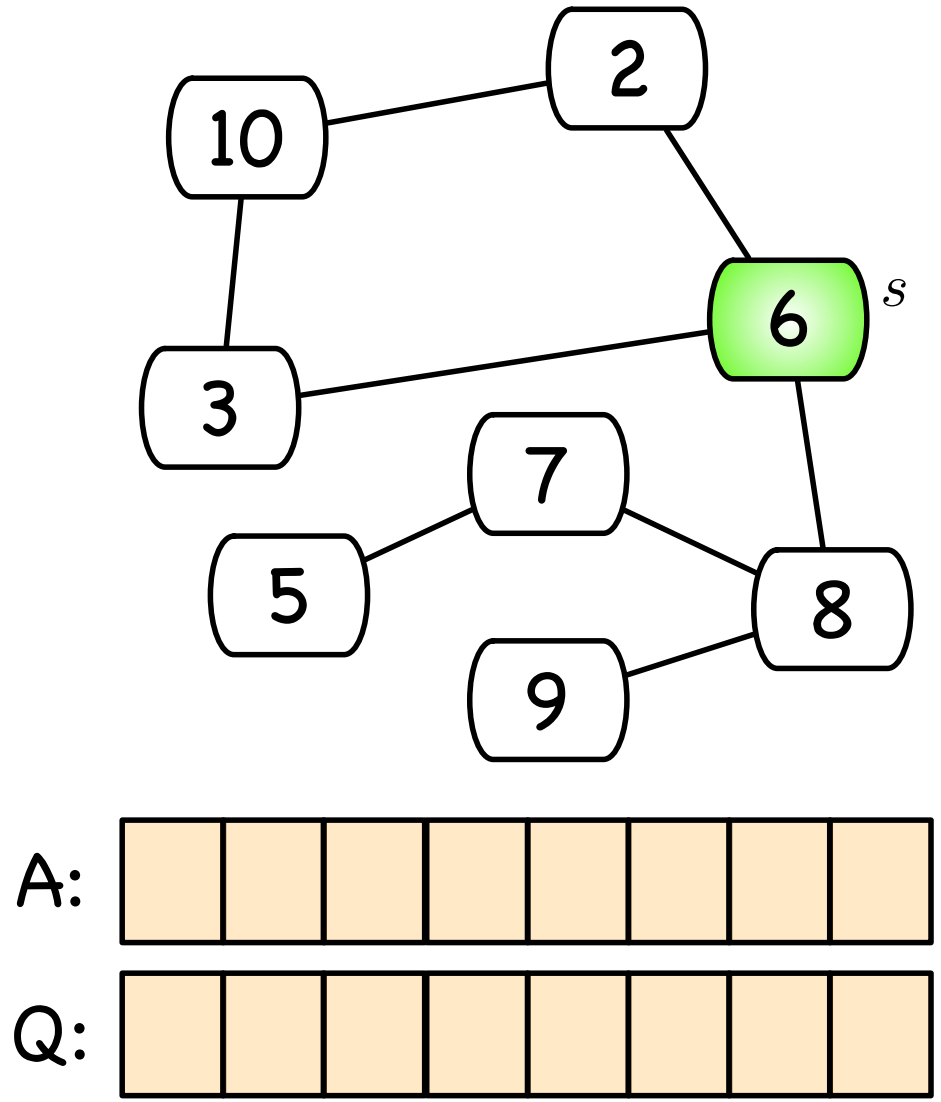
```
1:  farbe[s] = pink
2:  Q = ∅, enqueue(Q, s)
3:  while Q ≠ ∅ do
4:      u = dequeue(Q)
5:      for each v ∈ Adj[u] do
6:          if farbe[v] == weiss then
7:              farbe[v] = pink
8:              enqueue(Q, v)
9:          end if
10:     end for
11:     farbe[u] = schwarz
12: end while
```

Zur Nachbereitung

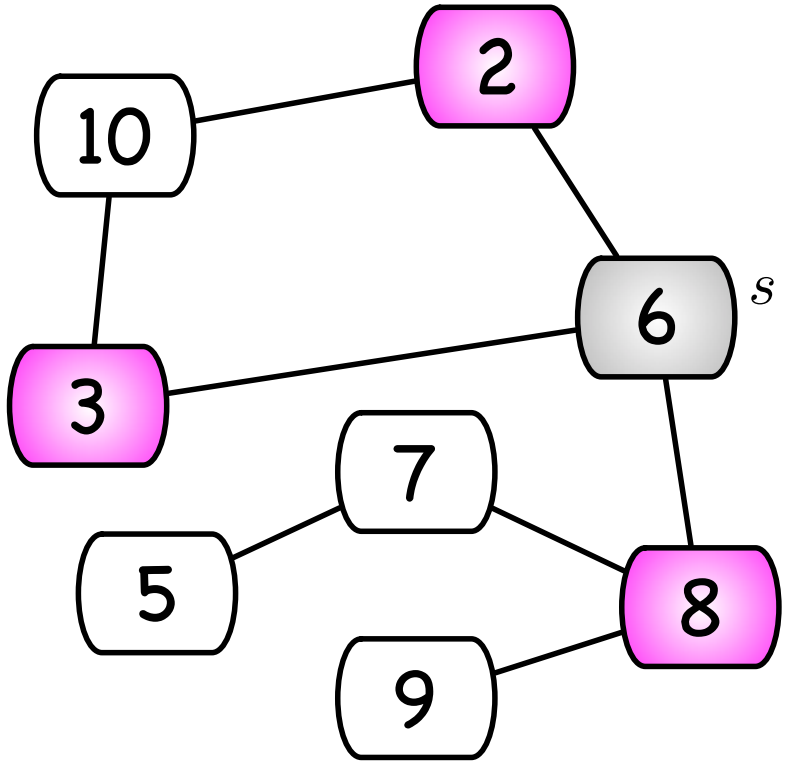
Anmerkung (zur Nachbereitung)

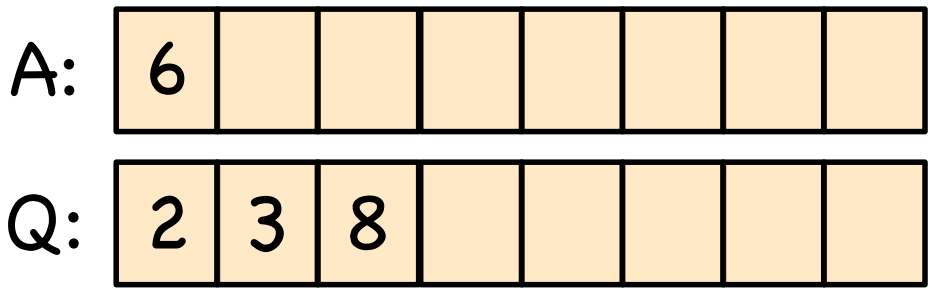
Wir verzichten für das nächste Beispiel auf die Angabe der einzelnen Adjazenzlisten. Man kann sie aus dem Beispiel ablesen. Man beachte noch, dass je nachdem wie die Knoten in der Adjazenzliste angeordnet sind, der Algorithmus verschiedene Ergebnisse liefern kann. Wären die Knoten 7 und 9 bspw. in der Adjazenzliste der 8 in umgekehrter Reihenfolge, so würde zuerst die 9 abgearbeitet werden, bevor man bei 7 und 5 weiter im Baum hinabsteigt. (Bei der Tiefensuche später ist dieser Unterschied noch merklicher.)

Breitensuche - im Graphen

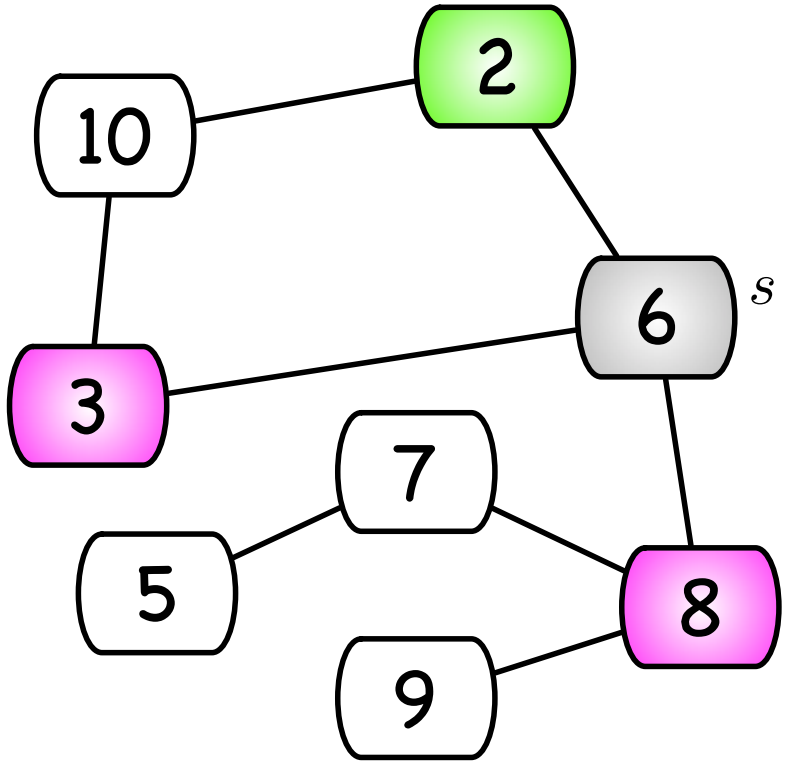


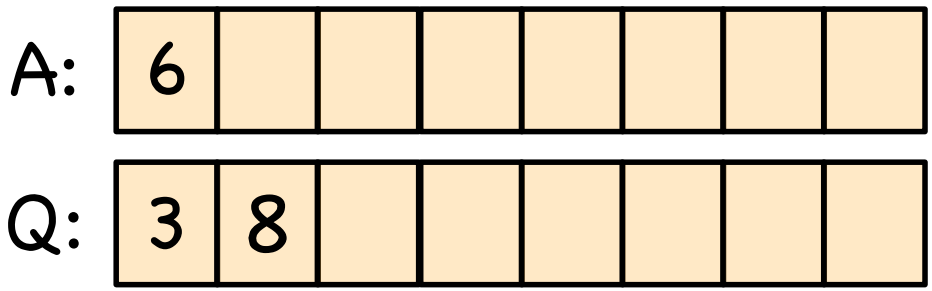
Breitensuche - im Graphen



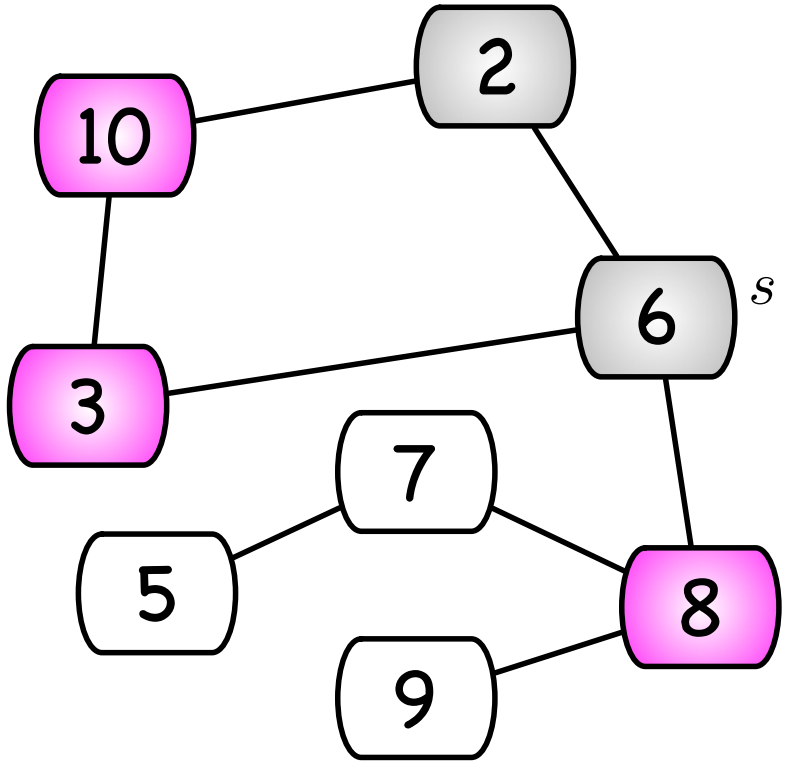


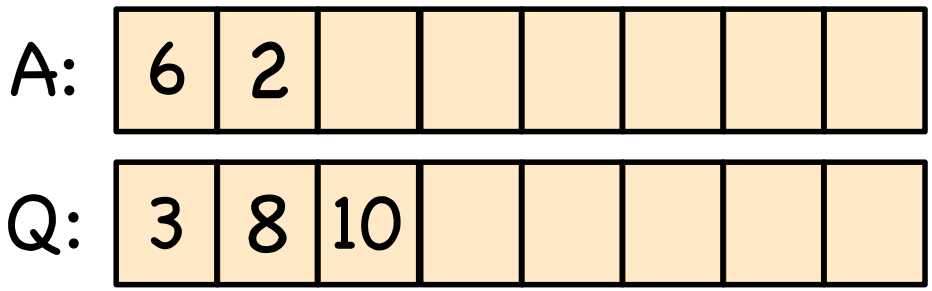
Breitensuche - im Graphen



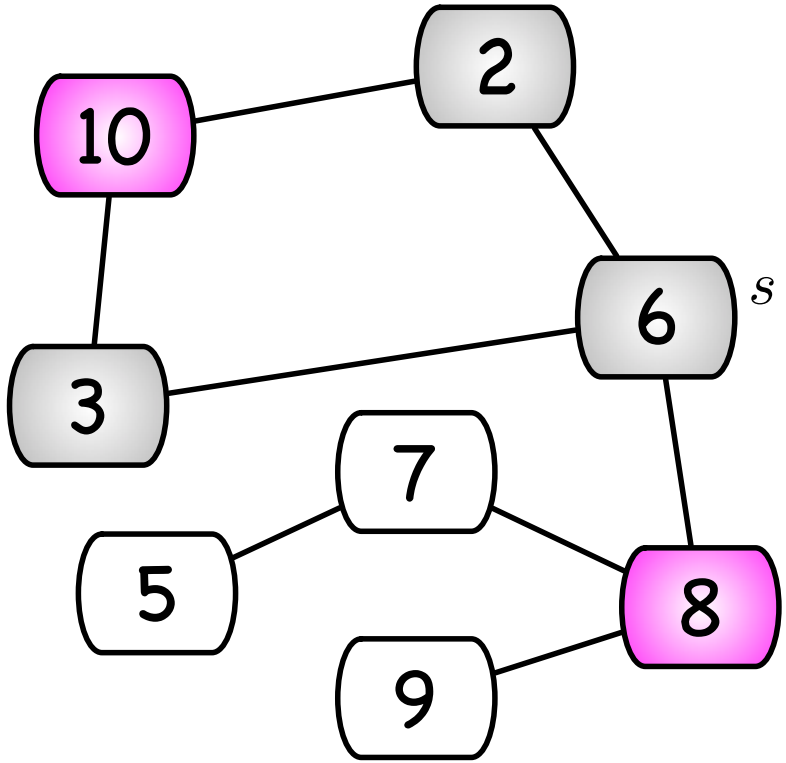


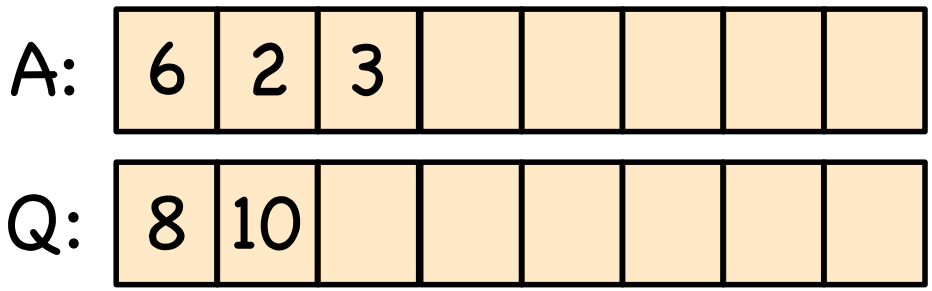
Breitensuche - im Graphen



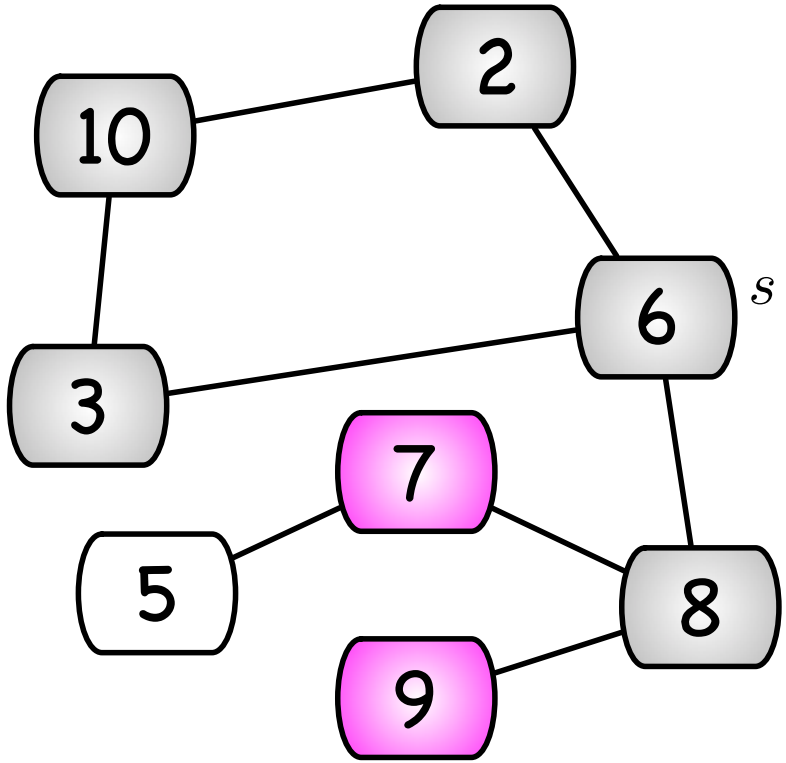


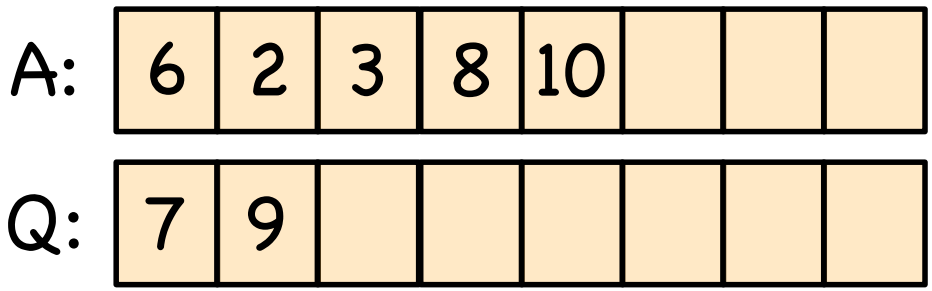
Breitensuche - im Graphen



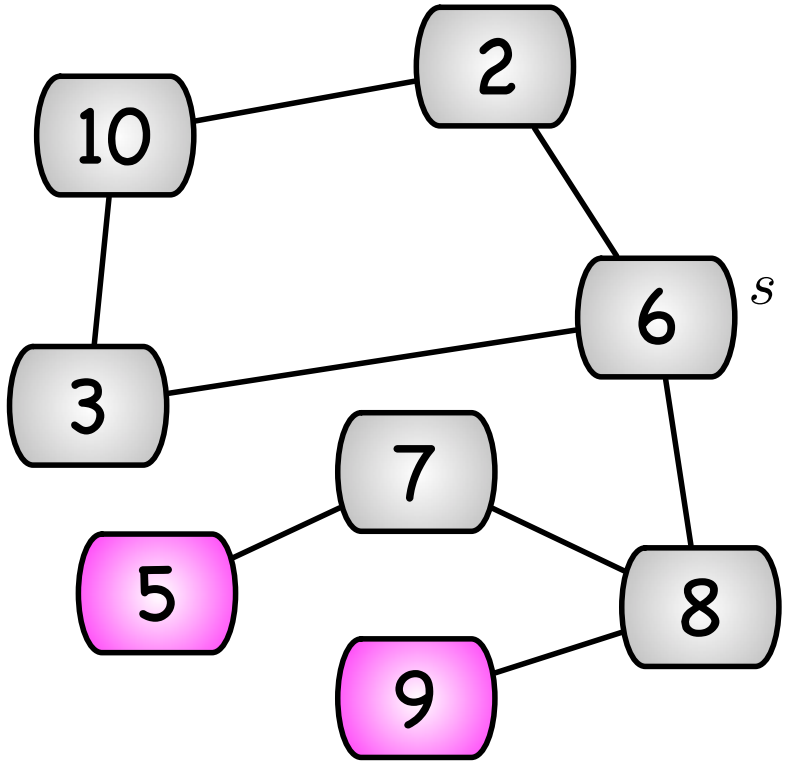


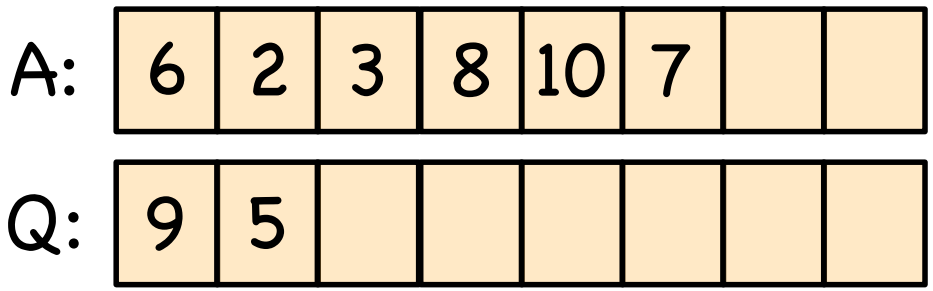
Breitensuche - im Graphen



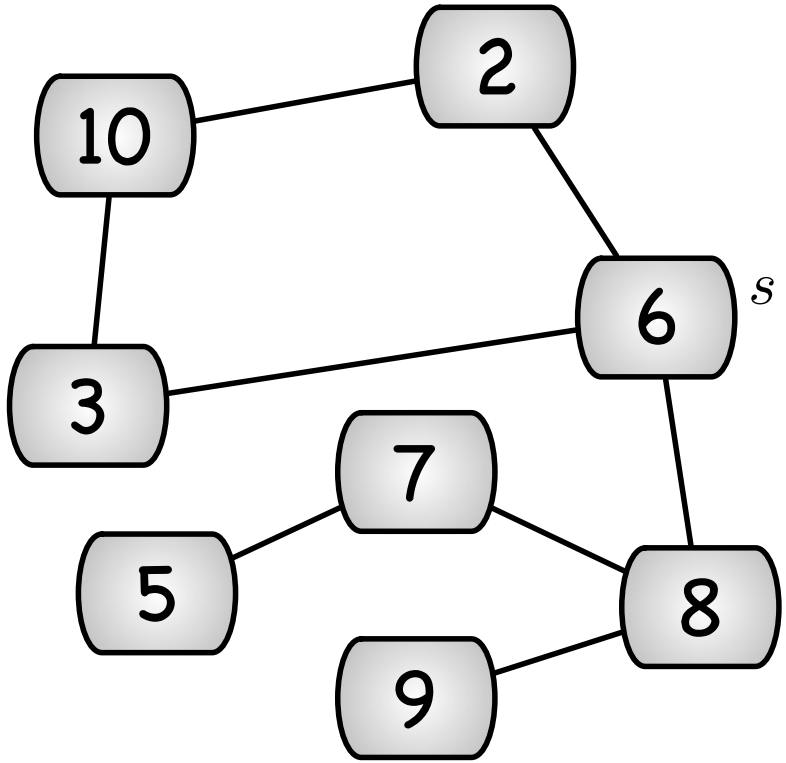


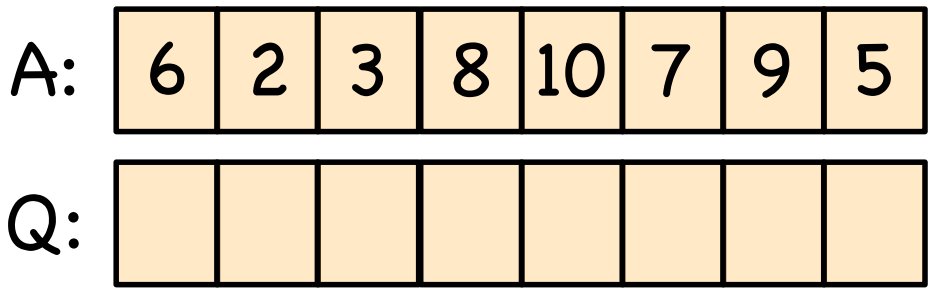
Breitensuche - im Graphen





Breitensuche - im Graphen





Breitensuche - mehr Informationen

Reichern wir die Breitensuche etwas an, so können wir mehr Informationen gewinnen, z.B. den Abstand der einzelnen Knoten von s und den Breitensuchbaum.

Der folgende Algorithmus ist wie im [Cormen].

Breitensuche - Initialisierung

Algorithmus 2 BFS(G, s) - Teil 1, Initphase

1: **for** each $u \in V(G) \setminus \{s\}$ **do**
2: farbe$[u] =$ weiss, $d[u] = \infty, \pi[u] =$ `nil`
3: **end for**
4: farbe$[s] =$ grau
5: $d[s] = 0, \pi[s] =$ `nil`
6: $Q = \emptyset$, enqueue(Q, s)

Anmerkung

Farben: weiss heißt 'noch nicht besucht', grau 'besucht, aber noch unbesuchte Nachbarn', schwarz 'besucht, alle Nachbarn entdeckt'. $d[u]$ ist die Anzahl der Schritt von s zu u, $\pi[u]$ der Vorgänger von u auf diesem Pfad.

Breitensuche - Hauptschleife

Algorithmus 3 BFS(G, s) - Teil 2, Hauptteil

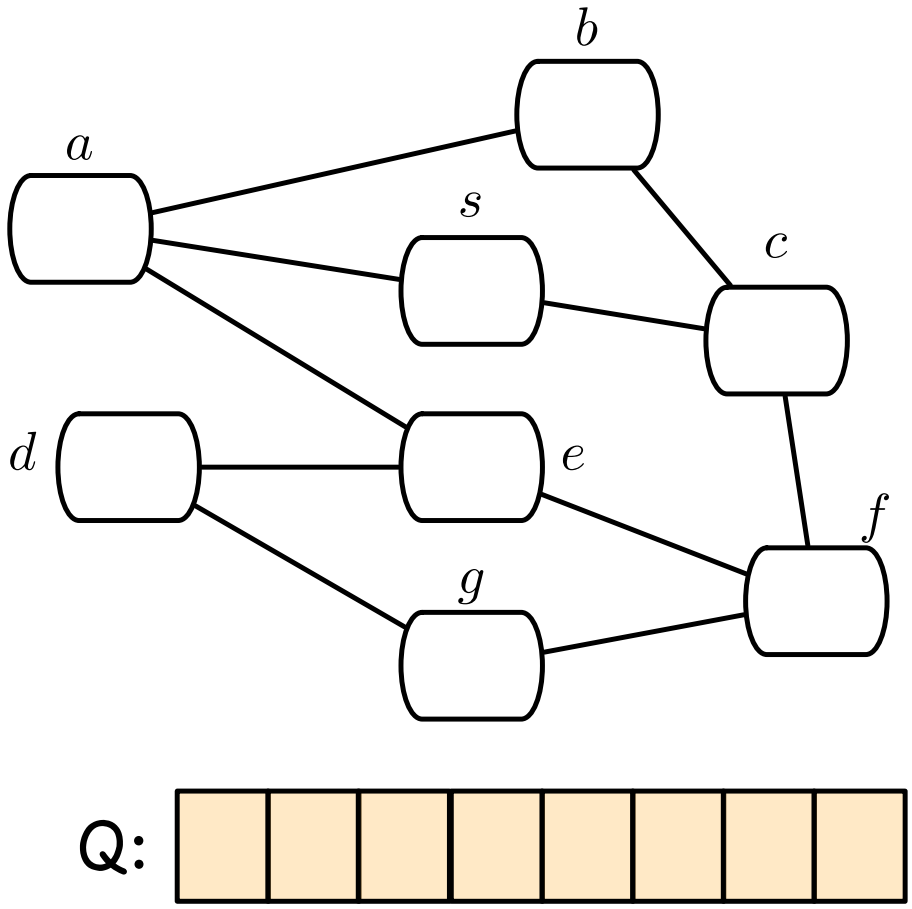
```
 1: while Q ≠ ∅ do
 2:     u = dequeue(Q)
 3:     for each v ∈ Adj[u] do
 4:         if farbe[v] == weiss then
 5:             farbe[v] = grau
 6:             d[v] = d[u] + 1, π[v] = u
 7:             enqueue(Q, v)
 8:         end if
 9:     end for
10:     farbe[u] = schwarz
11: end while
```

Zur Nachbereitung

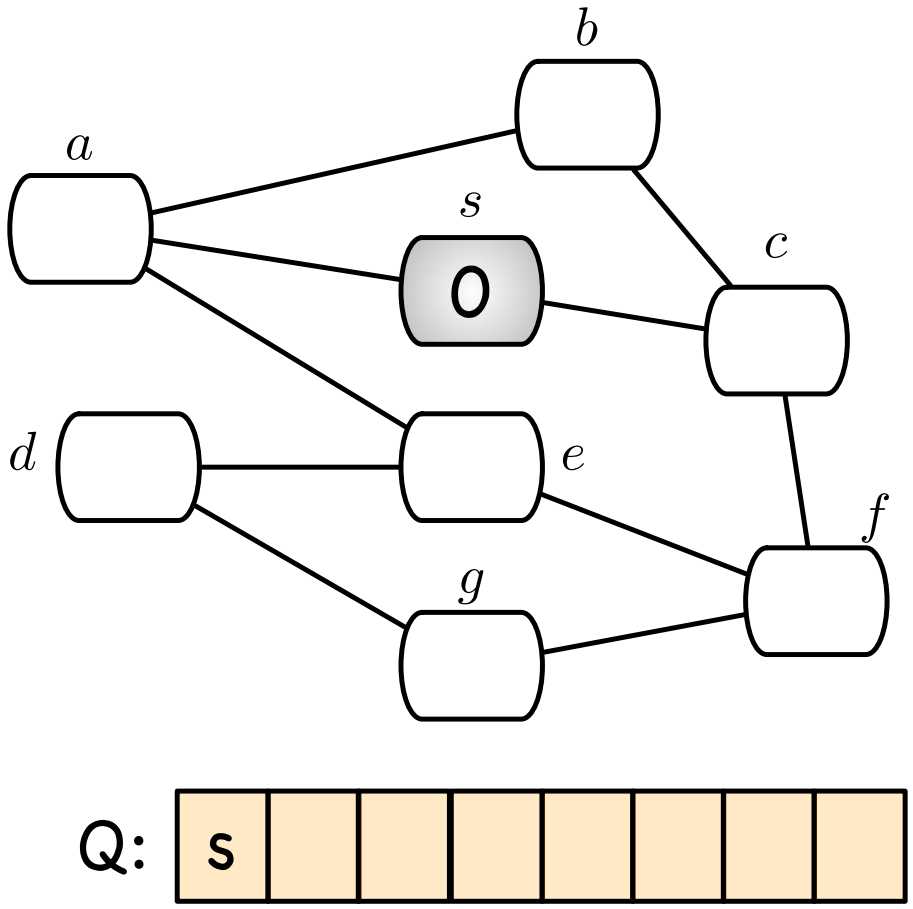
Anmerkung (zur Nachbereitung)

Für das nachfolgende Beispiel verzichten wir erneut auf die Angabe der Adjazenzlisten.
Die Zahl in einem Knoten v ist nun nicht mehr der Schlüssel o.ä., sondern der Wert von $d[v]$. Die Farbe 'grün' in den Bildern entspricht der Farbe 'schwarz' im Quellcode (d.h. ein grüner Knoten ist vollständig abgearbeitet und alle seine Kinder sind 'entdeckt' (also mindestens 'grau')).
Die dicken Kanten stellen die Baumkanten dar, die mittels der π Funktion gewonnen werden können, die jeden Knoten auf seinen Vorgänger im Baum abbildet.

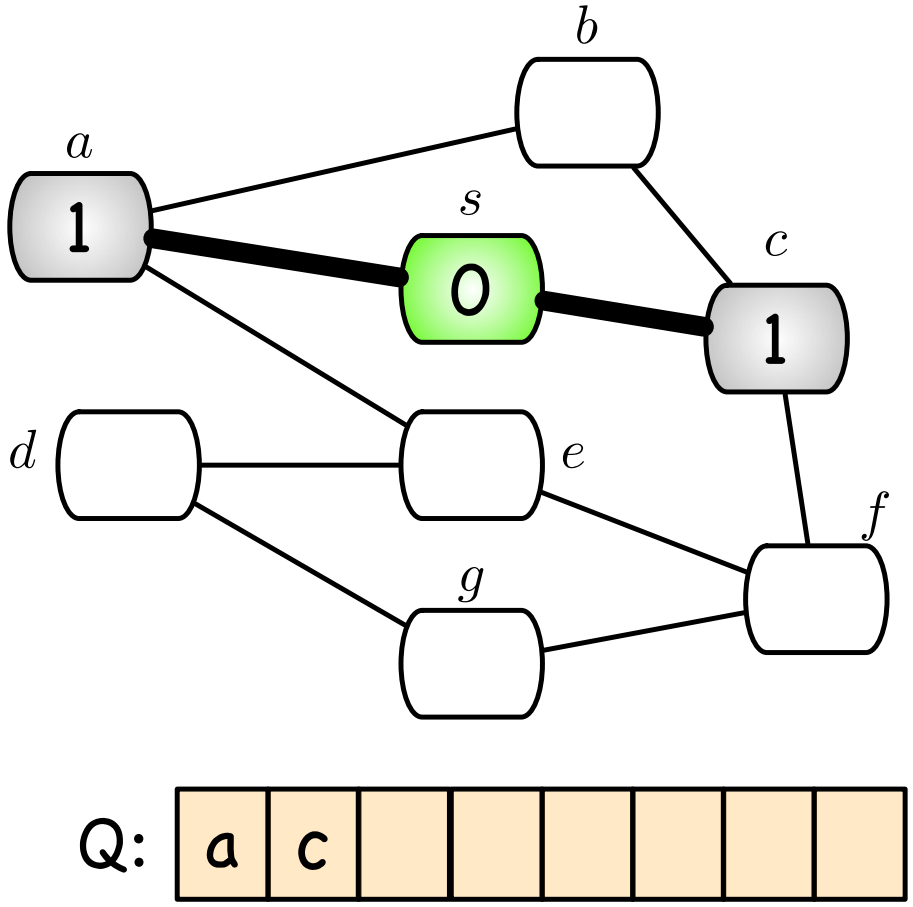
Breitensuche - Beispiel



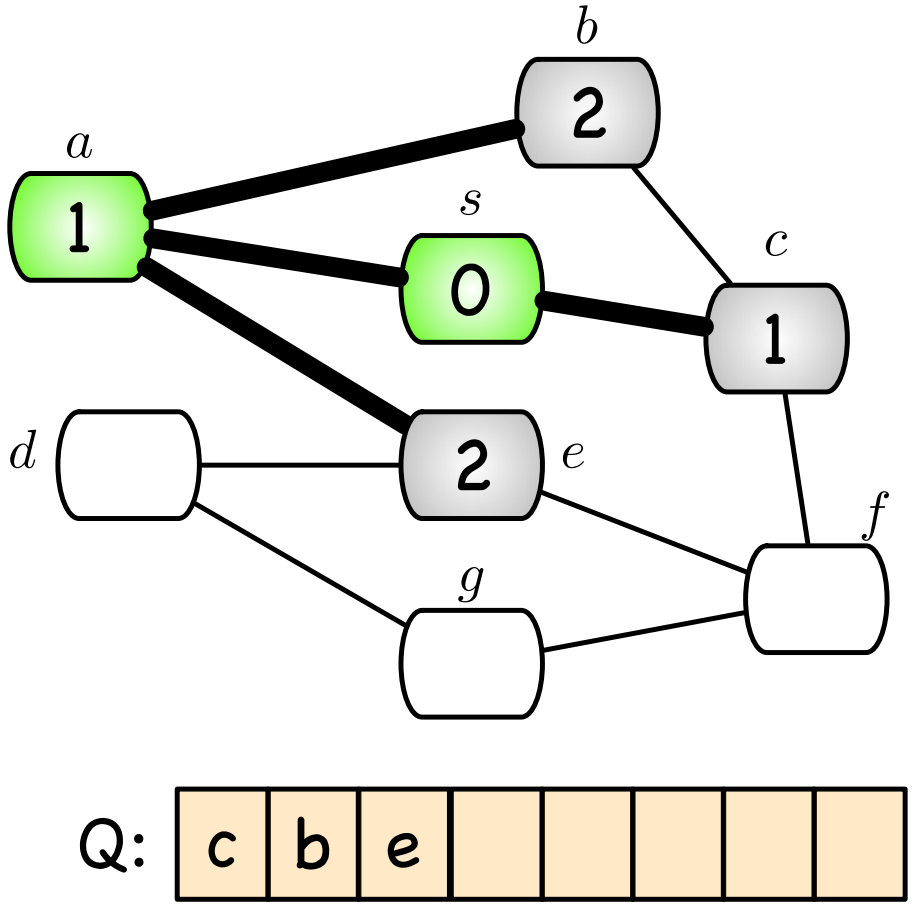
Breitensuche - Beispiel



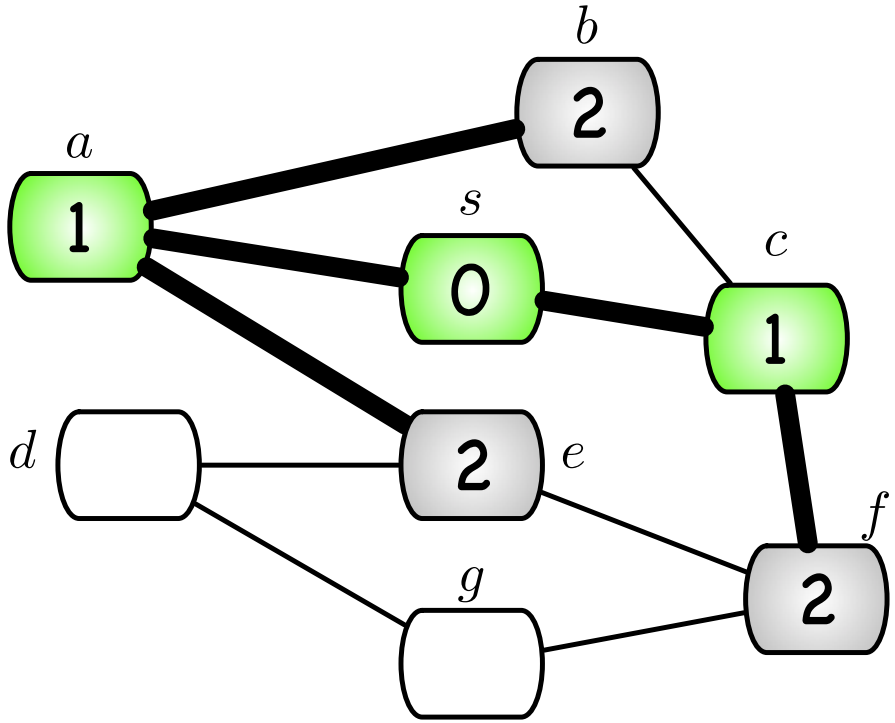
Breitensuche - Beispiel

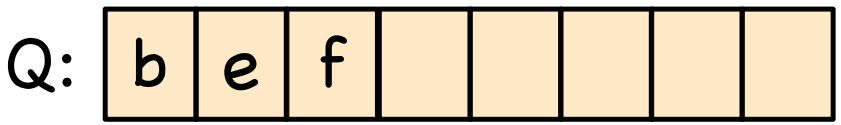


Breitensuche - Beispiel

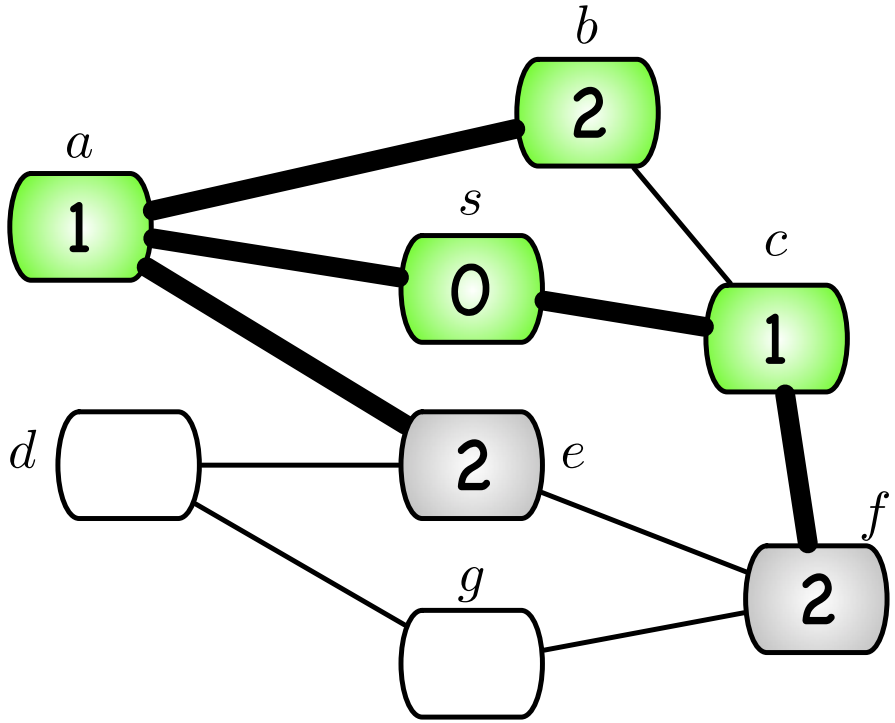


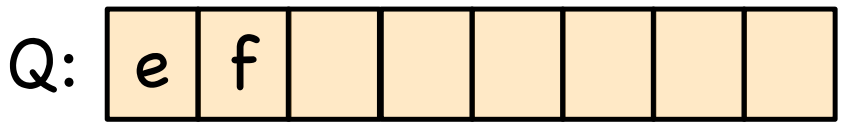
Breitensuche - Beispiel



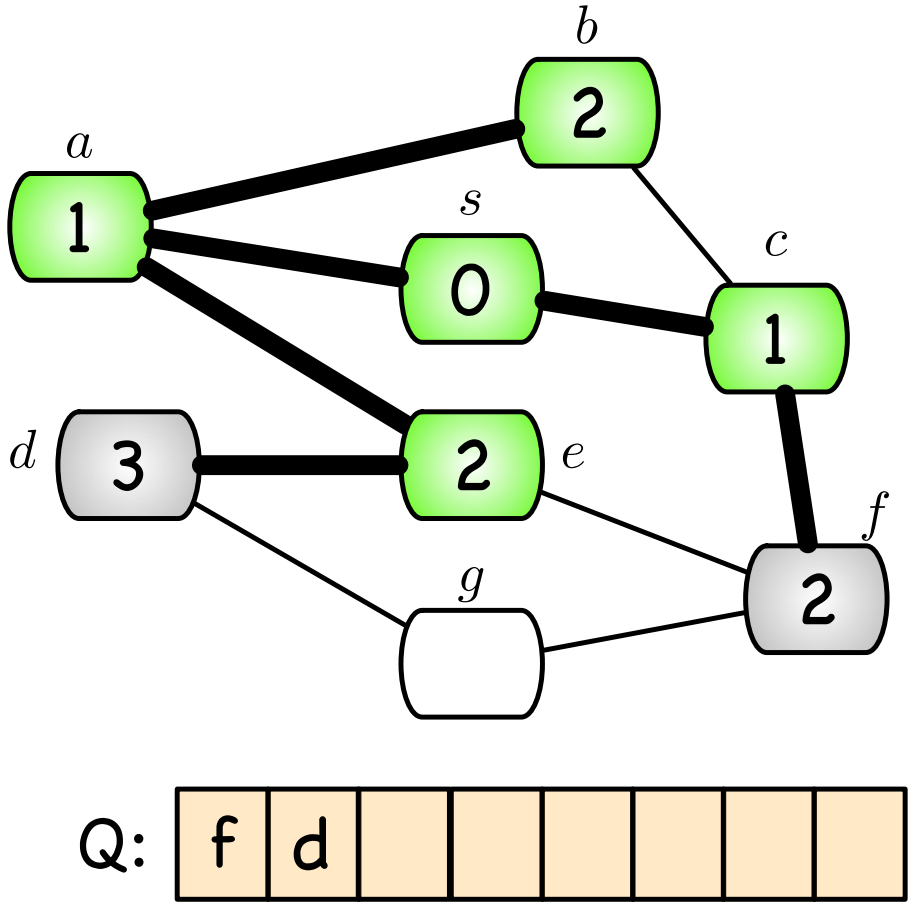


Breitensuche - Beispiel

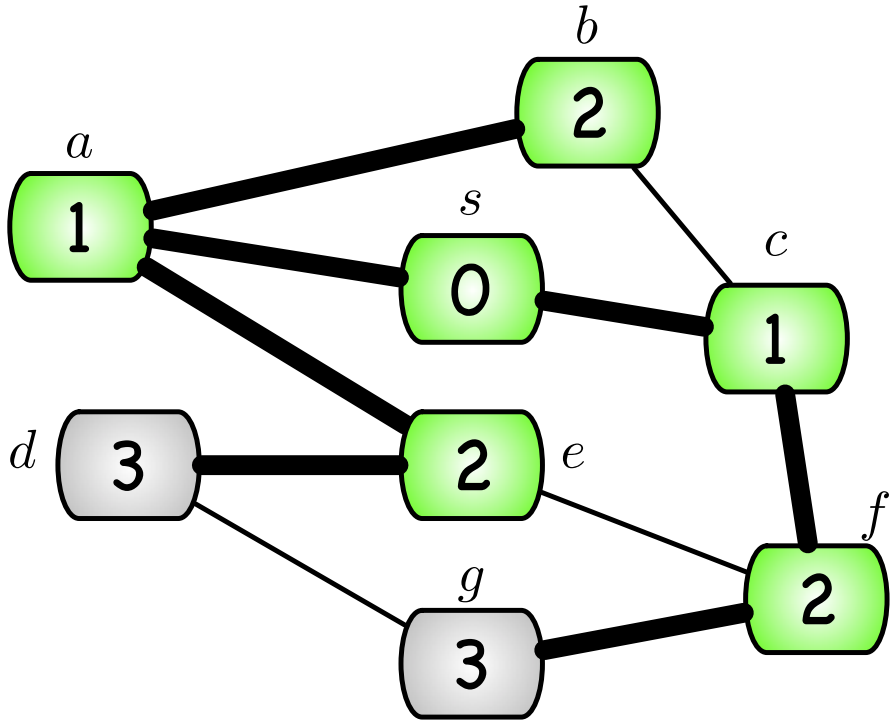


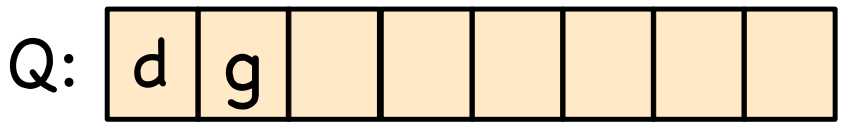


Breitensuche - Beispiel

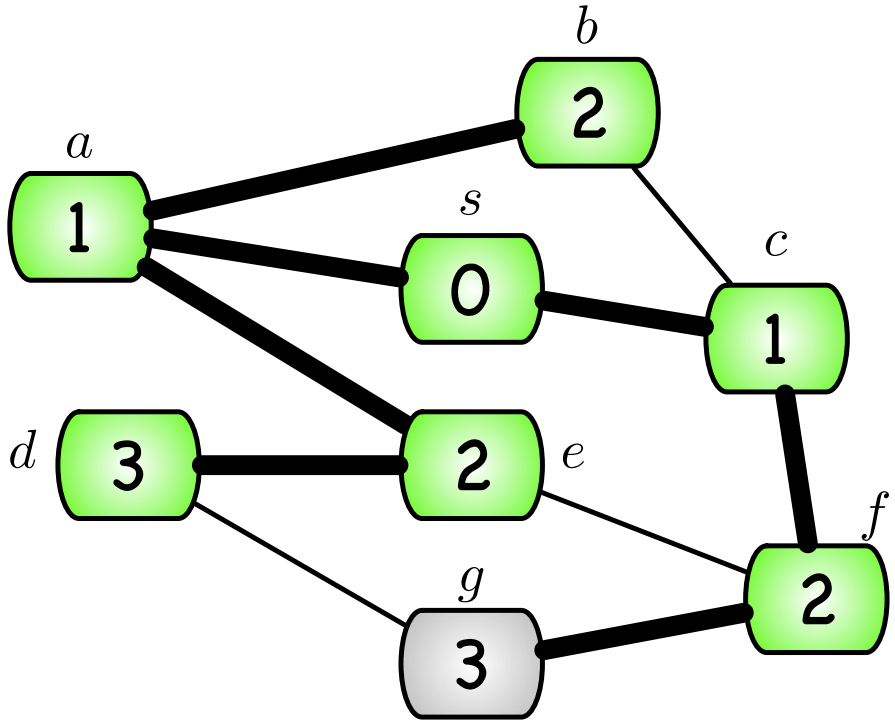


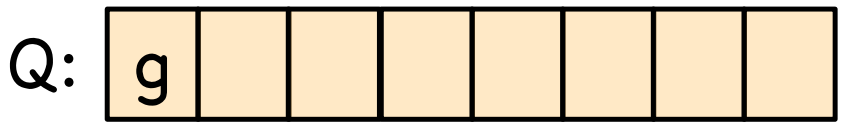
Breitensuche - Beispiel



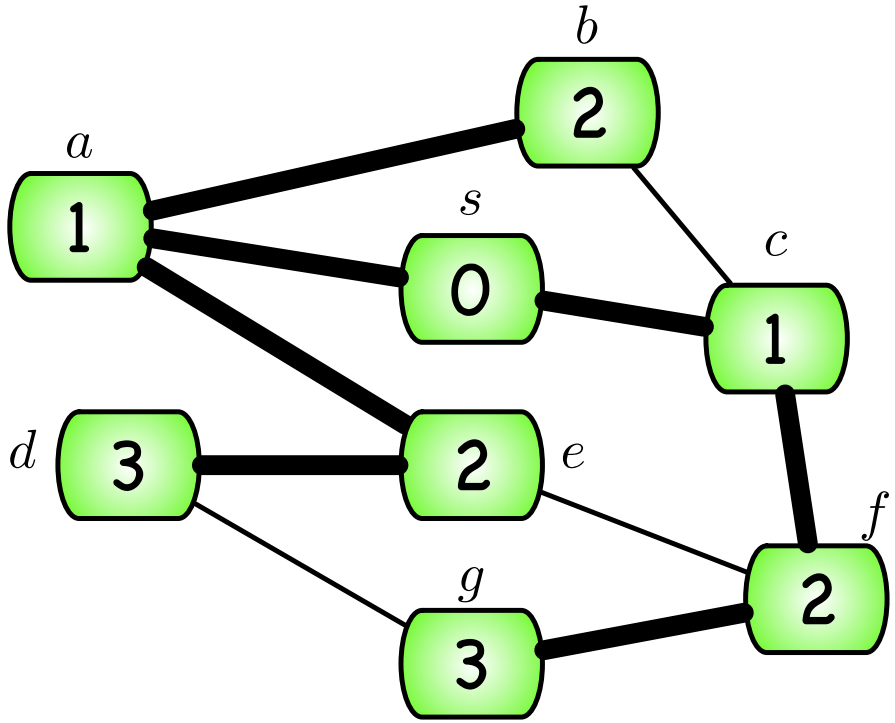


Breitensuche - Beispiel





Breitensuche - Beispiel



Q:

Analyse (Laufzeit)

- Zu Anfang ist jeder Knoten weiß. Der Test in der Schleife stellt daher sicher, dass jeder Knoten nur einmal in die Queue eingefügt und aus ihr entnommen wird.
- Für die Warteschlangenoperationen brauchen wir also $O(V)$.
- Die Adjazenzliste jedes Knotens wird nur einmal geprüft (wenn der Knoten aus der Queue entnommen wird).
- Die Summe aller Adjazenzlisten ist in $\Theta(E)$, folglich benötigt das Prüfen der Adjazenzlisten $O(E)$.
- Initialisierung geht auch in $O(V)$.
- Insgesamt ergibt sich so: $O(V + E)$, was linear in der Größe der Adjazenzliste ist.

Anmerkung

Diese Laufzeitanalyse stimmt nur, wenn der Graph durch Adjazenzlisten gegeben ist!

Analyse (Korrektheit)

Die Korrektheit kann man mittels einer Schleifeninvarianten zeigen, die besagt, dass die Queue aus der Menge der grauen Knoten besteht.

Alternativ kann man mit mehreren Lemmata zeigen, dass die ermittelten Abstände tatsächlich die kürzesten sind und ein Baum entsteht. Man erhält so (siehe [Cormen], dort ca. 3 Seiten):

Satz

Gegeben $G = (V, E)$ (gerichtet oder ungerichtet) und $s \in V$. BFS ermittelt jeden von s aus erreichbaren Knoten v. Bei Terminierung gilt $d[v] = \delta(s, v)$ für alle v. Zudem ist für jeden von s aus erreichbaren Knoten v $(v \neq s)$ einer der kürzesten Pfade von s nach v ein kürzester Pfad von s nach $\pi[v]$ gefolgt von der Kante $(\pi[v], v)$.

Breitensuche - Ergebnis

Zur Betonung:

Satz

In einem ungewichteten Graphen (bzw. einem Graph in dem jede Kante das Gewicht 1 hat) ermittelt der BFS-Algorithmus für jeden Knoten v den kürzesten Abstand zu s sowie einen kürzesten Pfad von s zu v (den man erhält indem man den $\pi[v]$ rückwärts folgt). Der Algorithmus läuft in $O(V + E)$, wenn der Graph als Adjazenzliste gegeben ist.

Wichtige Anmerkung

Beweis umbedingt mal im [Cormen] nachlesen!

Tiefensuche - Die Idee

Bei der Tiefensuche geht man einfach immer weiter noch ungeprüfte Kanten entlang, so lange dies möglich ist. Man folgt also 'einem Pfad' so lange es möglich ist. Erst wenn man einen Knoten erreicht von dem aus man nur bereits besuchte Knoten erreichen kann, geht man einen Schritt zurück.

Bleiben unentdeckte Knoten übrig wählt man diese als neue Startknoten. So kann sich (bei einem unzusammenhängenden Graphen) ein Tiefensuch*wald* ergeben. (Die Breitensuche kann man auch so anpassen, aber typischerweise werden die Algorithmen so benutzt.)

Tiefensuche - Die Idee

- Starte mit Knoten s in einem Stack S.
- Wiederhole solange S nicht leer...
  - Nimm obersten Knoten v aus S.
  - (Bearbeite diesen und Färbe diesen so, dass er nicht wieder besucht wird.)
  - Tue alle Nachbarn von v, die bisher nicht besucht wurden auf den Stack.

Anmerkung

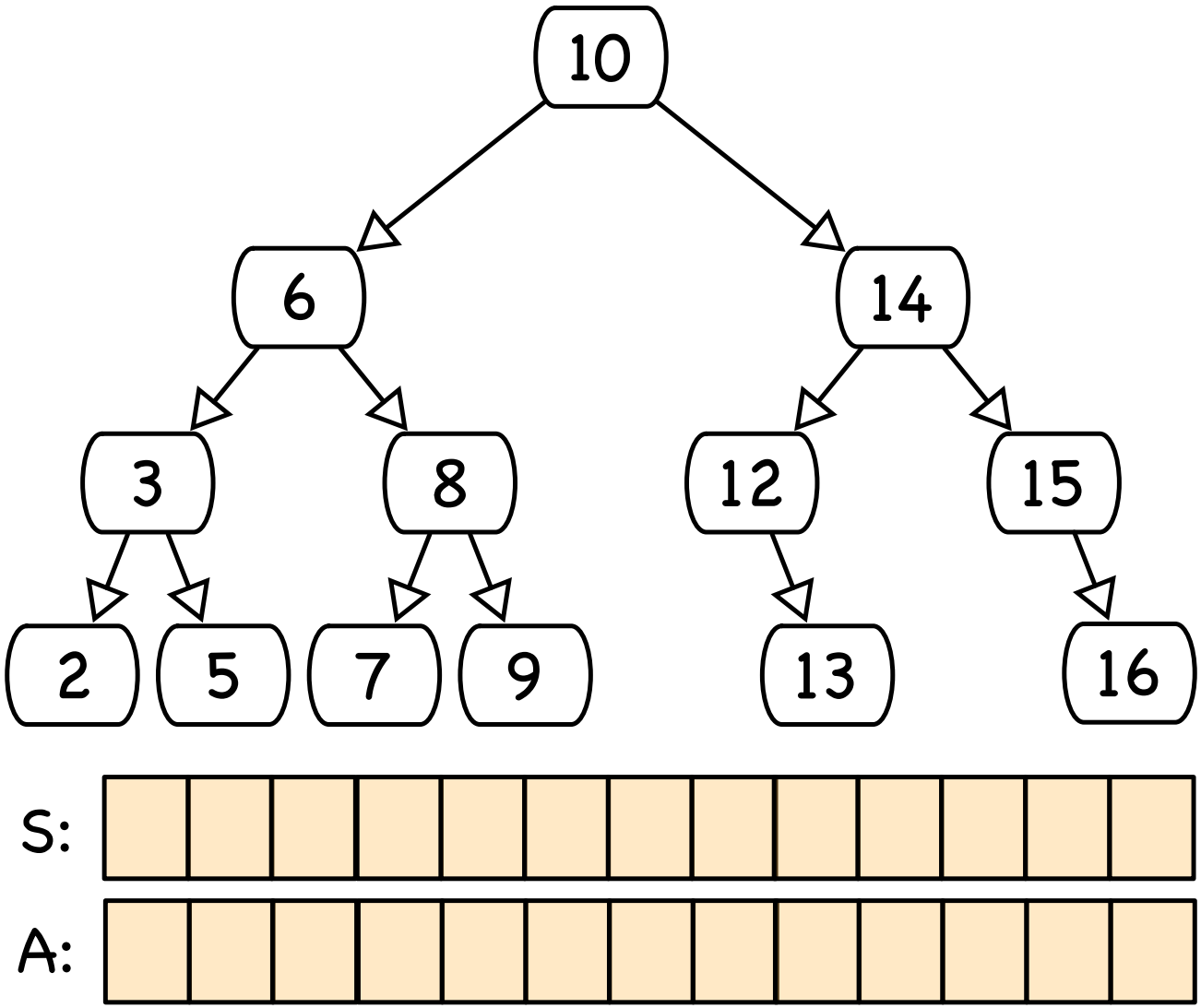
Durch den Stack wird sichergestellt, dass die Knoten 'in der Tiefe' besucht werden, d.h. dass man zunächst von einem Knoten aus immer weiter 'in die Tiefe' geht, bevor man zurück geht und eine andere 'Abzweigung' nimmt.

Zur Nachbereitung

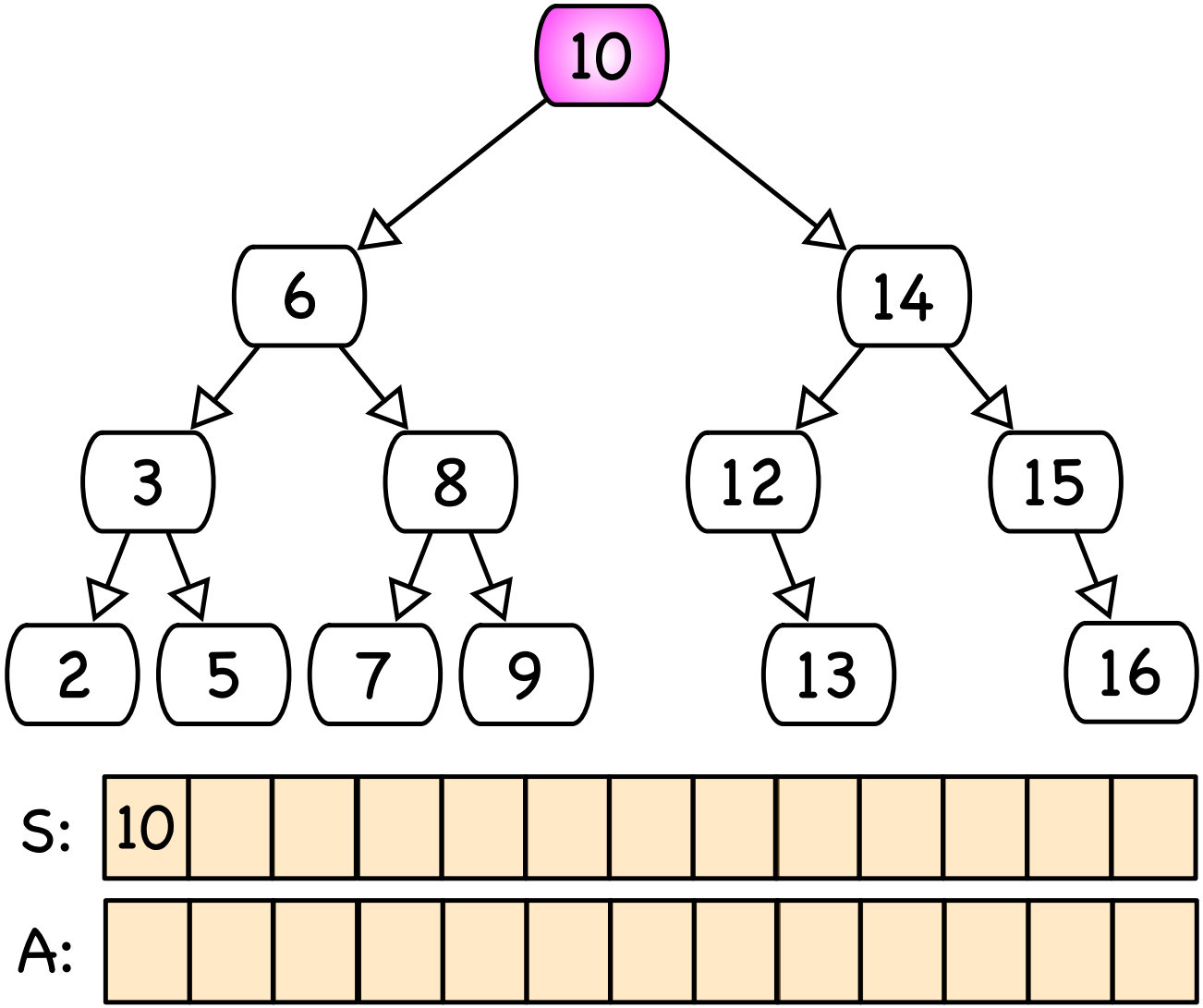
Anmerkung (zur Nachbereitung)

Wie bei der Breitensuche ist im nachfolgenden Beispiel für einen Knoten v jeweils sein linkes Kind der erste Knoten in v's Adjazenzliste, sein rechtes Kind der zweite Knoten in der Adjazenzliste. **Mit einer Ausnahme:** Beim Knoten 10 ist dies genau umgekehrt. Darum ist der Stack im vierten Bild mit $6, 14$ gefüllt.
Für das Beispiel ist es wieder unerheblich, ob die Kanten gerichtet oder ungerichtet sind. Bei ungerichteten Kanten wäre auch stets noch der Vater in der Adjazenzliste eines Knotens. Dieser wäre jedoch bereits gefärbt und würde nicht wieder im Stack gespeichert werden.

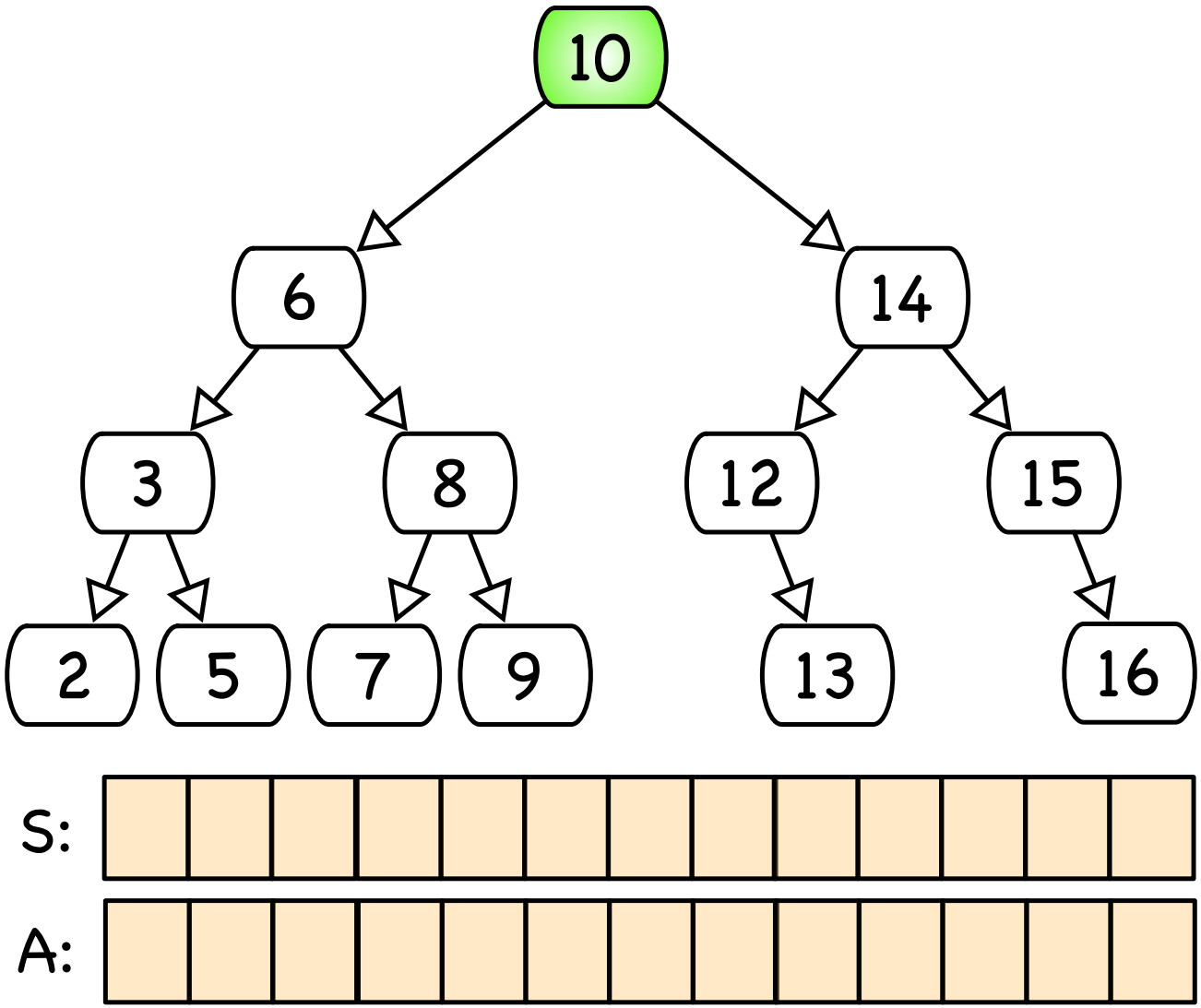
Tiefensuche - im Baum



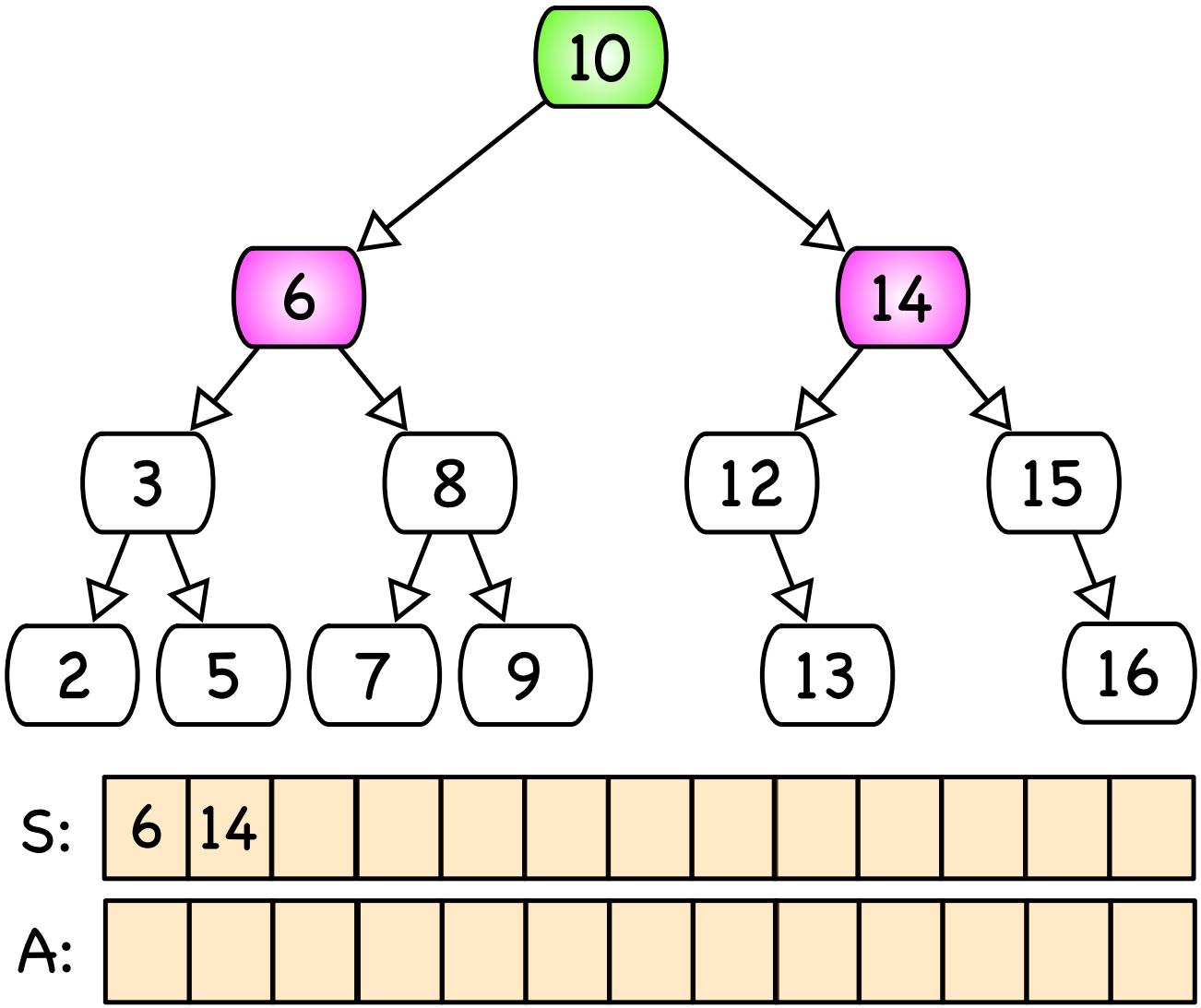
Tiefensuche - im Baum



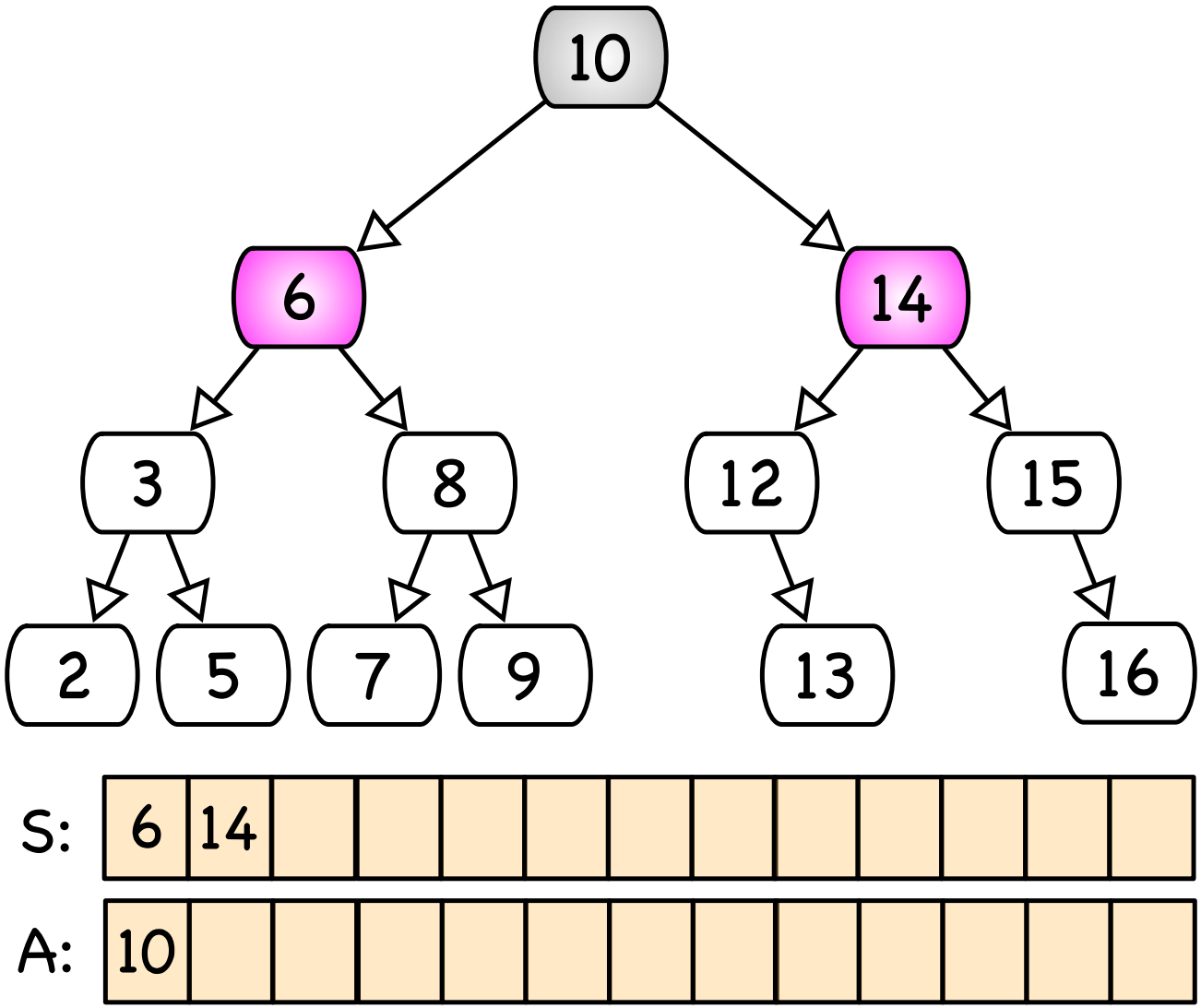
Tiefensuche - im Baum



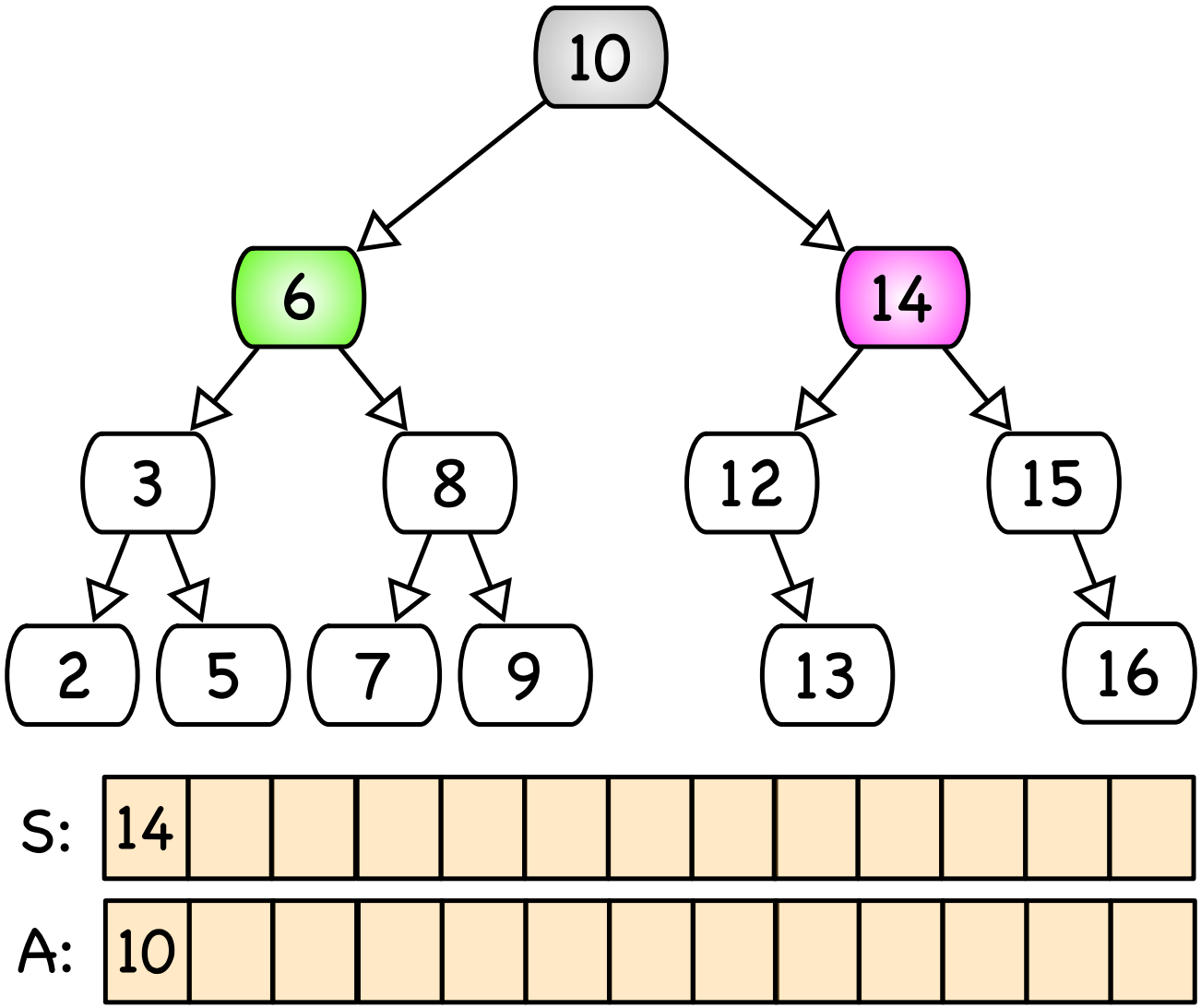
Tiefensuche - im Baum



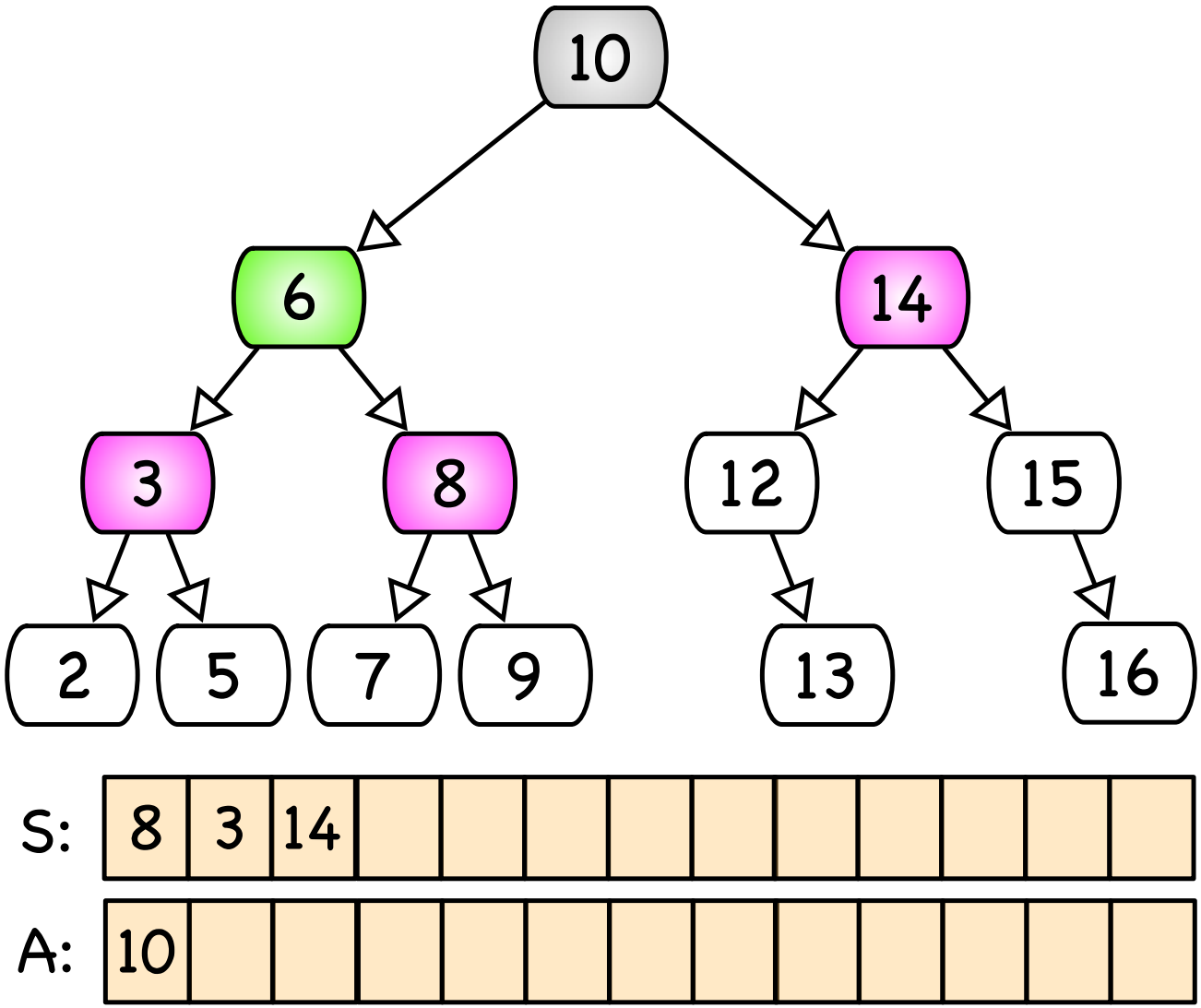
Tiefensuche - im Baum



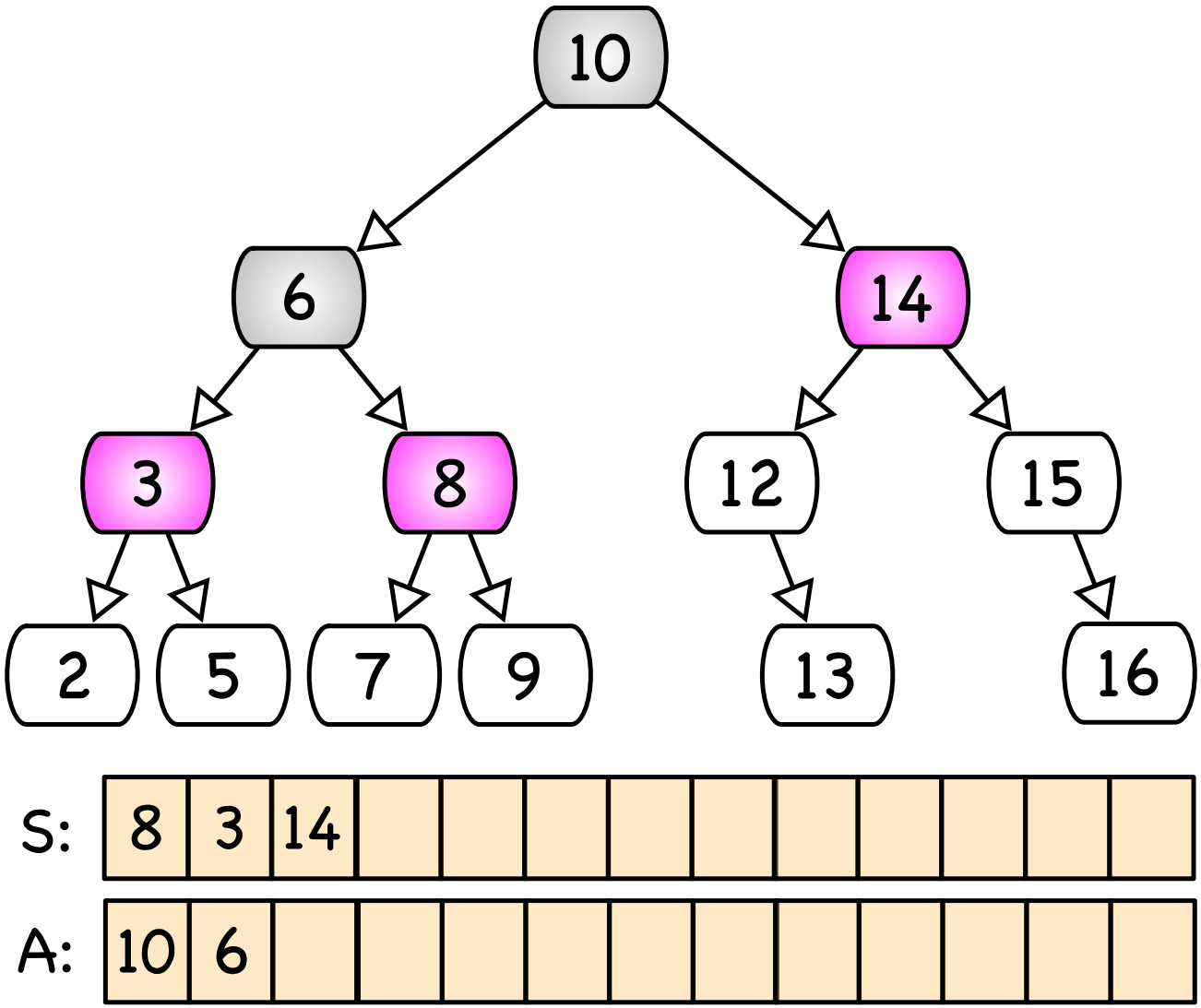
Tiefensuche - im Baum



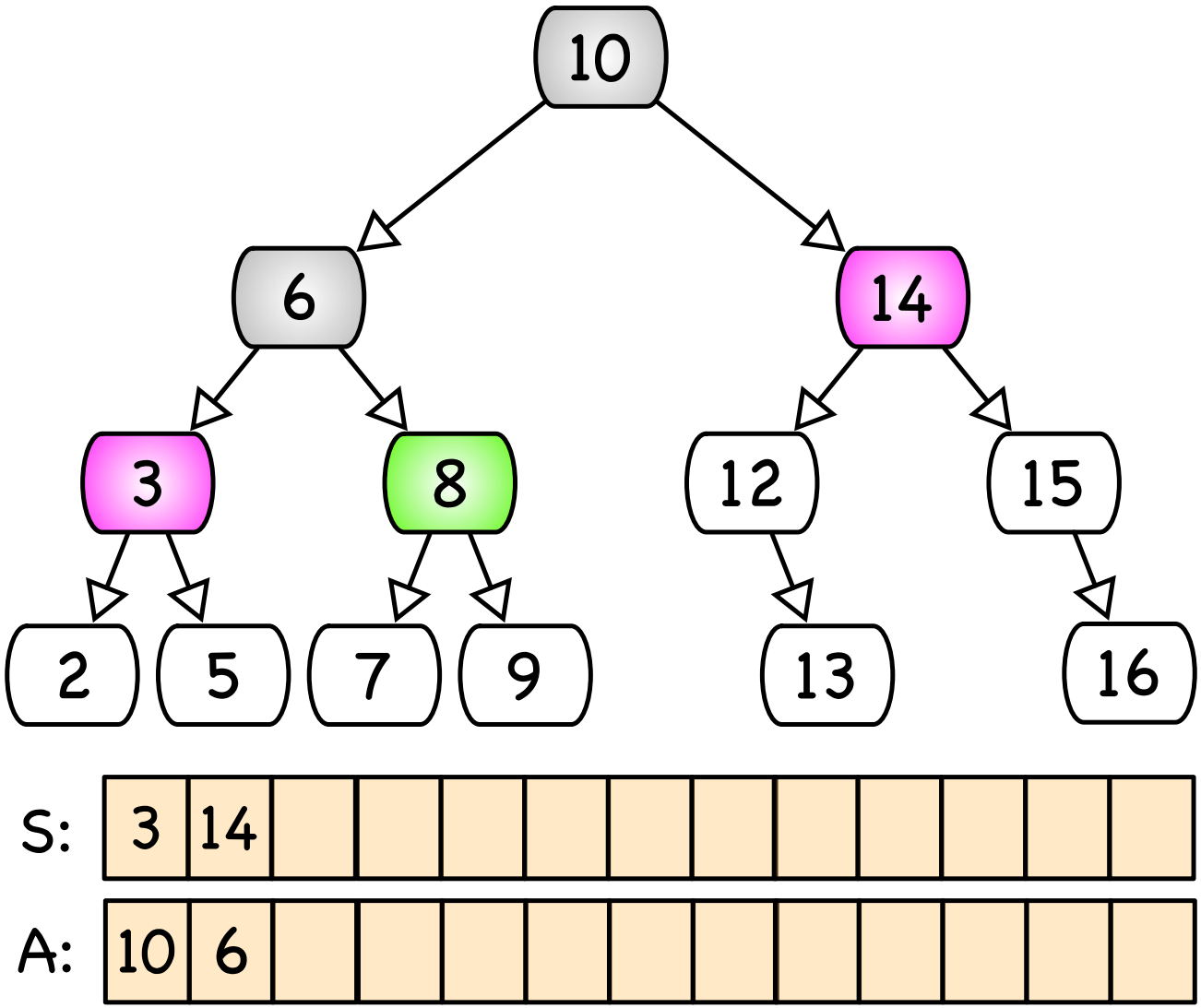
Tiefensuche - im Baum



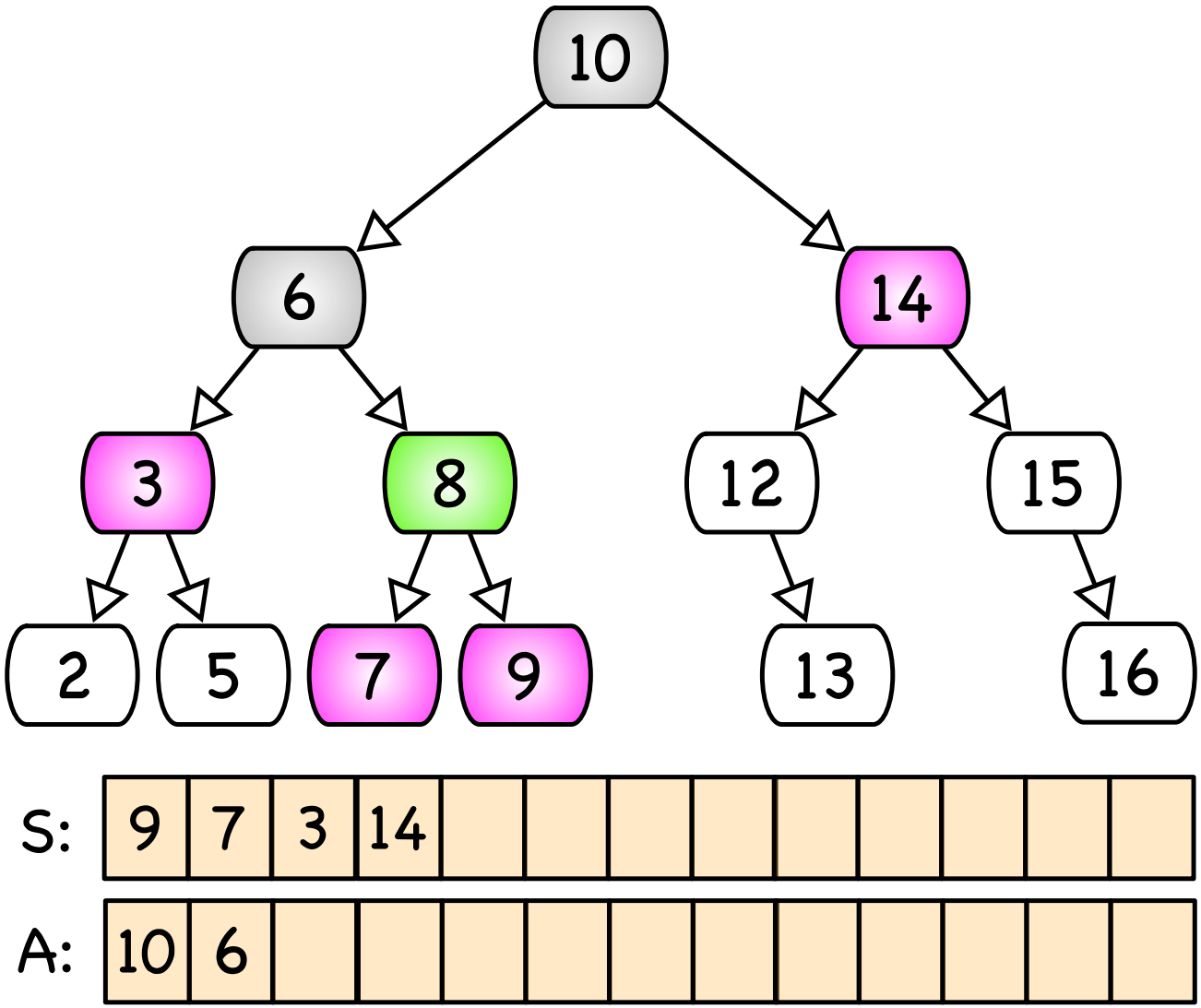
Tiefensuche - im Baum



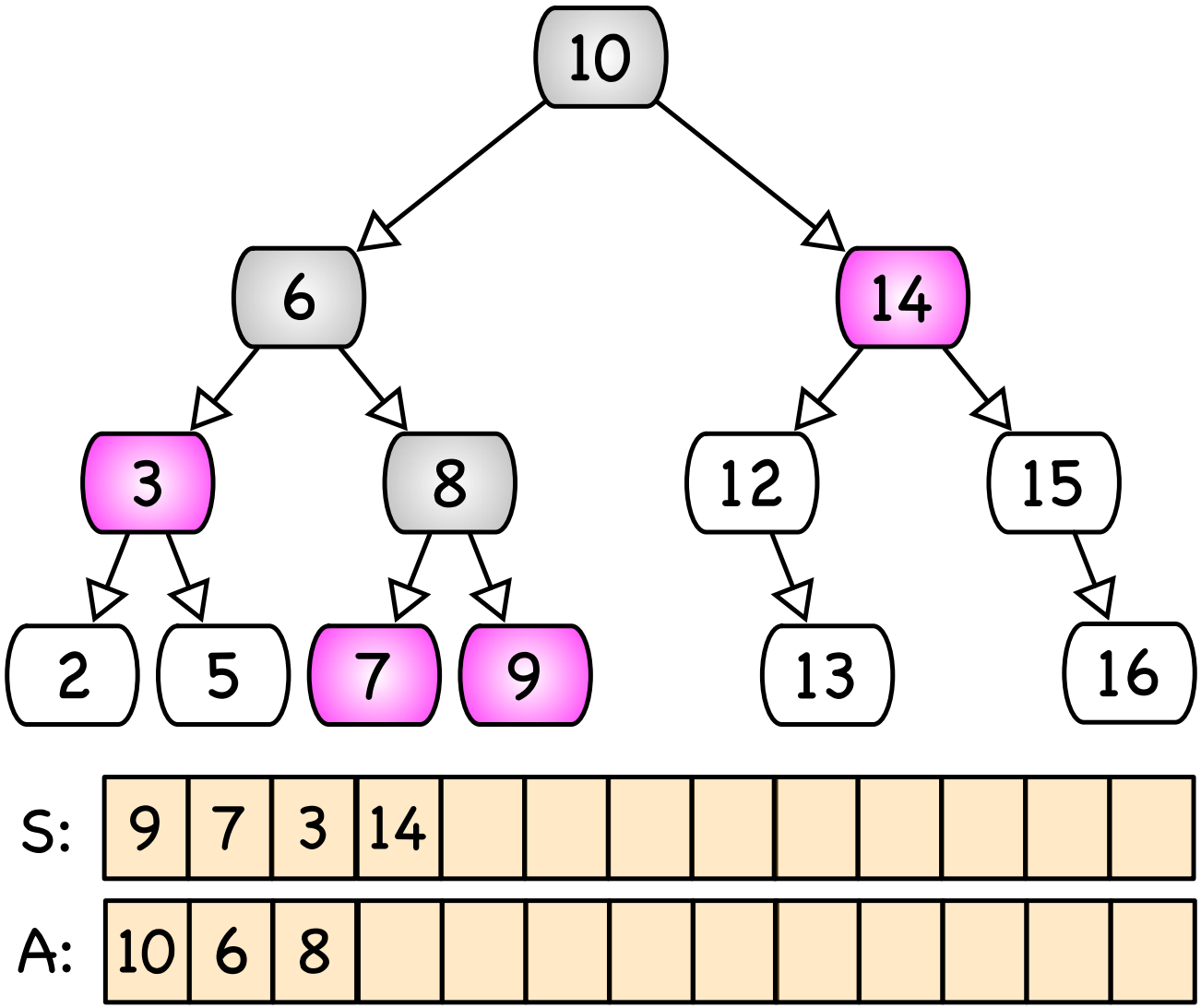
Tiefensuche - im Baum



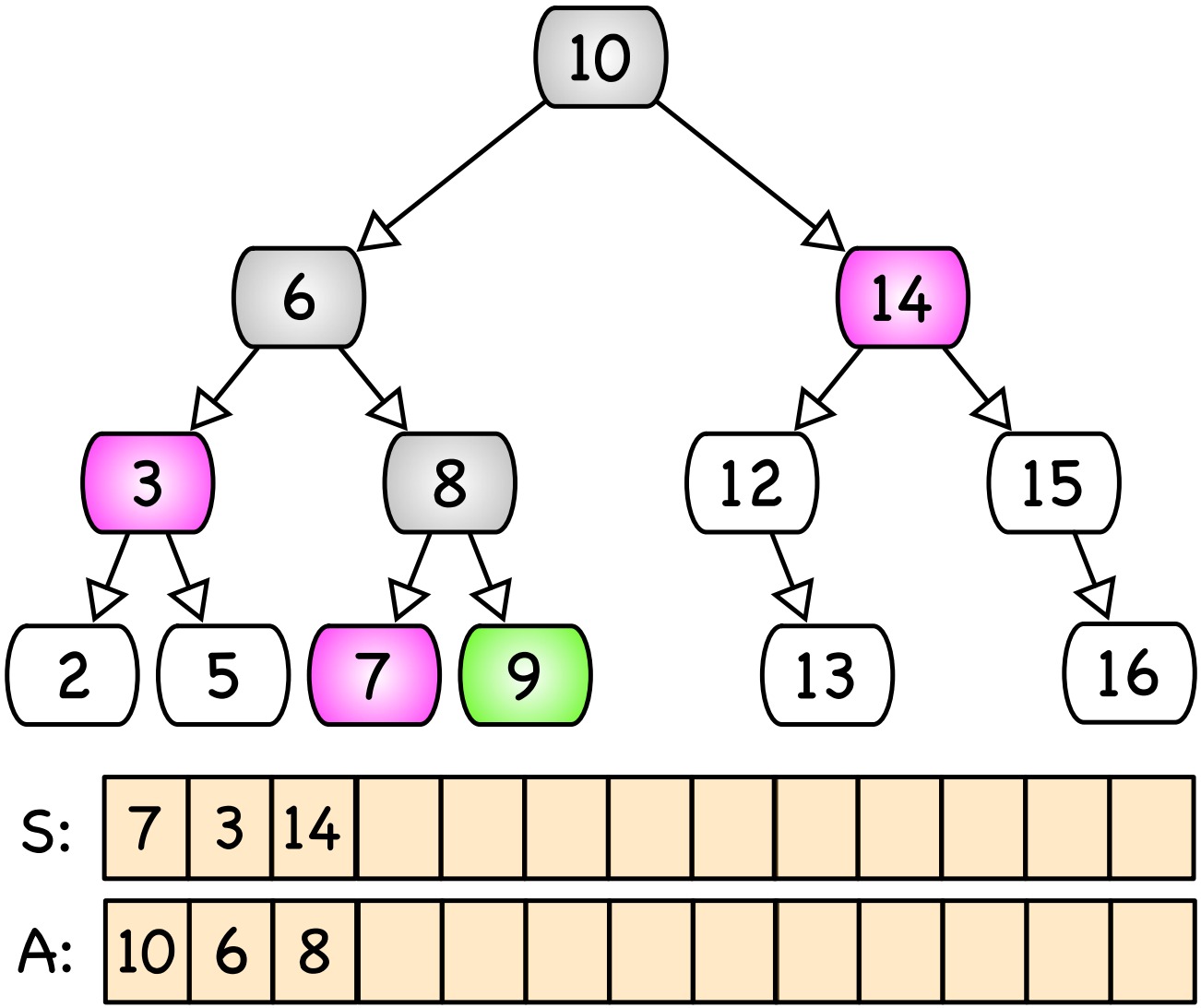
Tiefensuche - im Baum



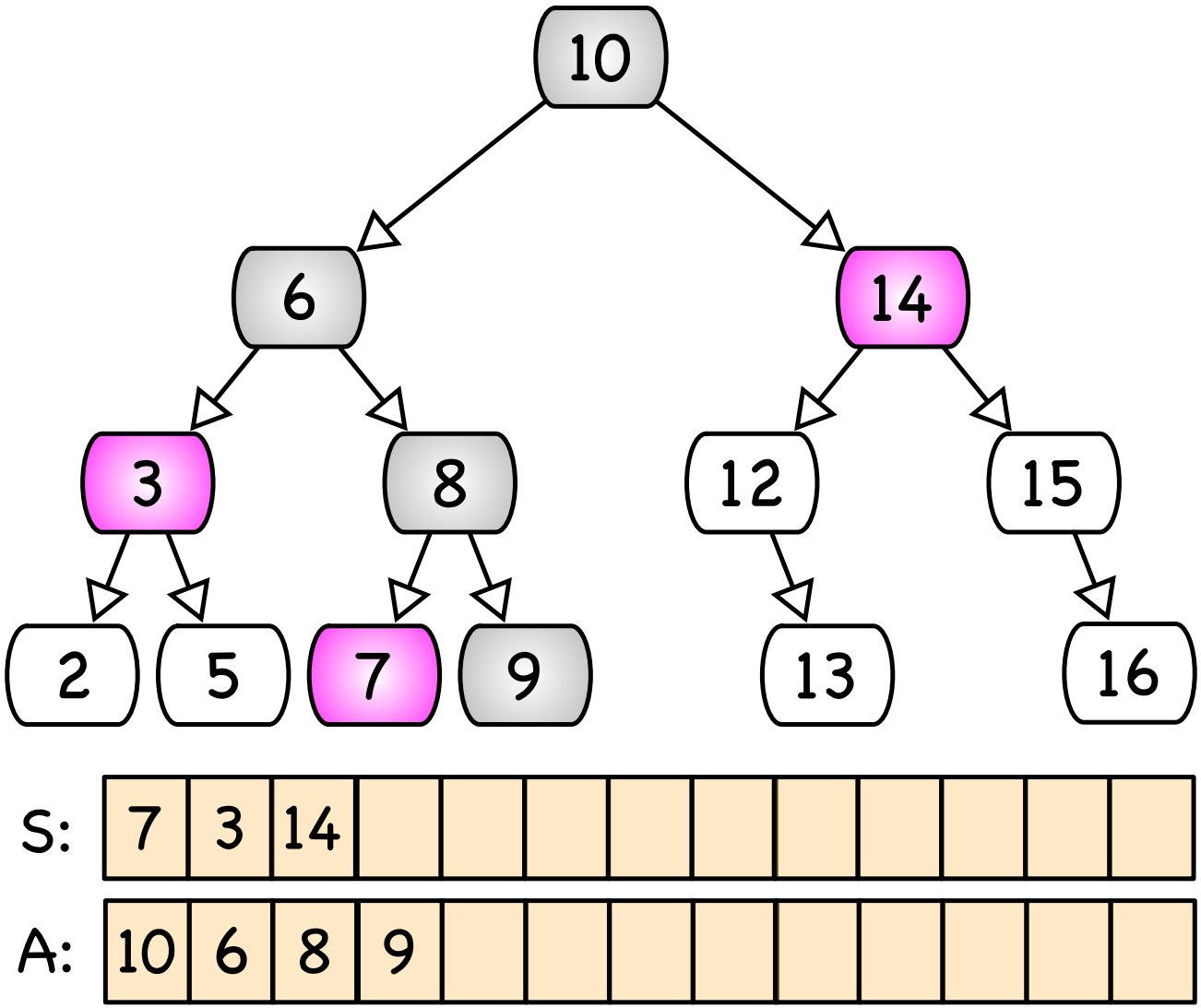
Tiefensuche - im Baum



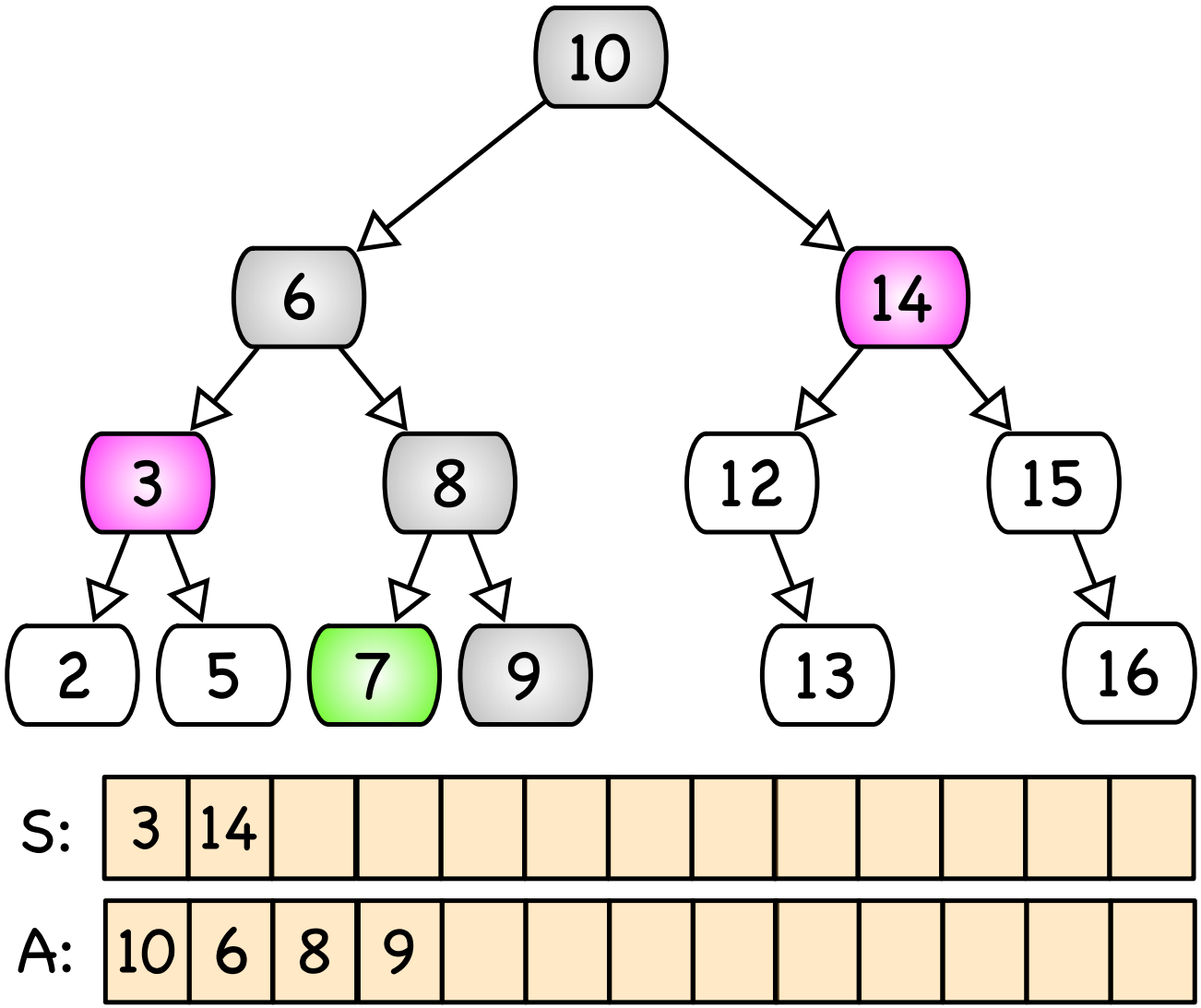
Tiefensuche - im Baum



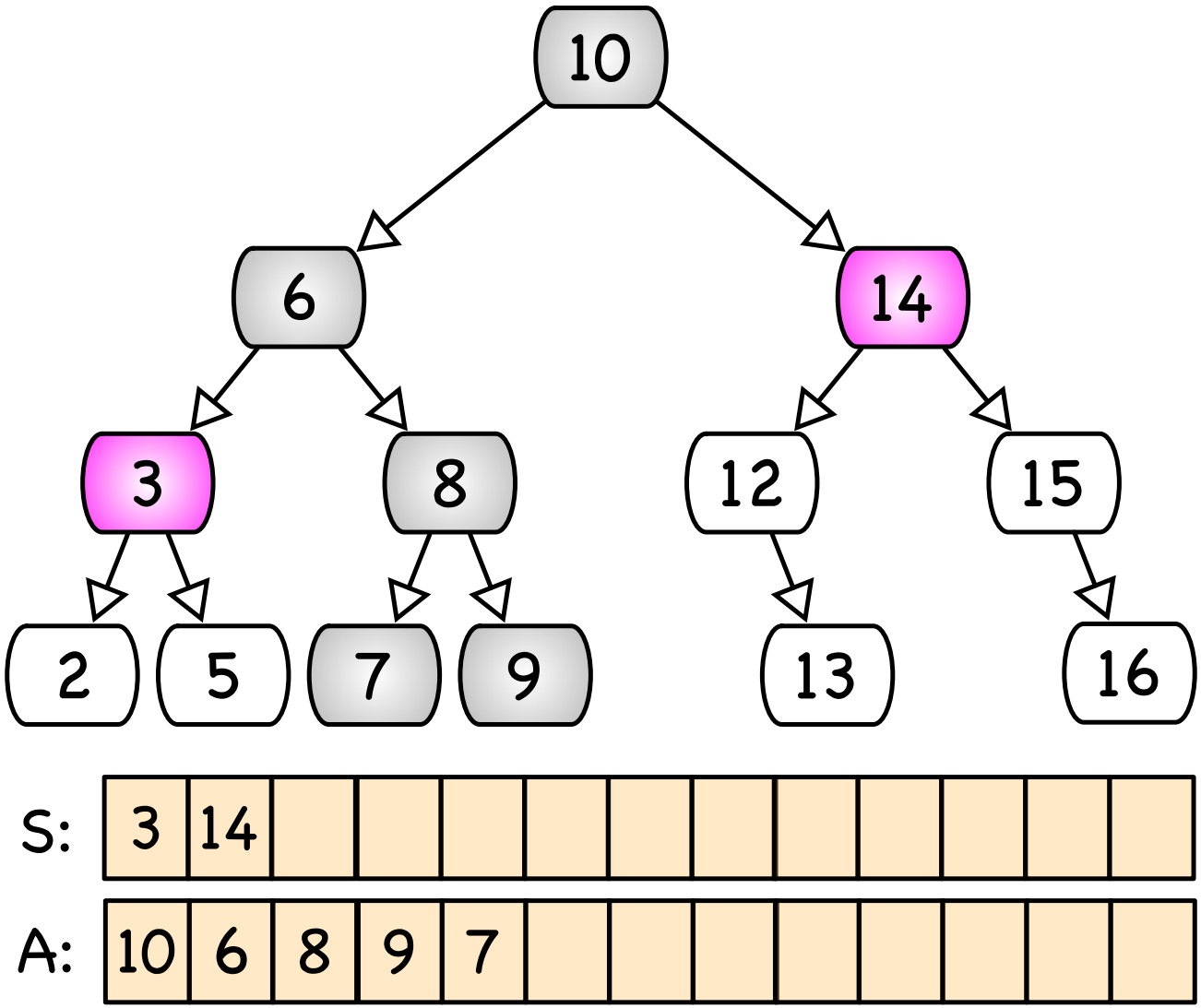
Tiefensuche - im Baum



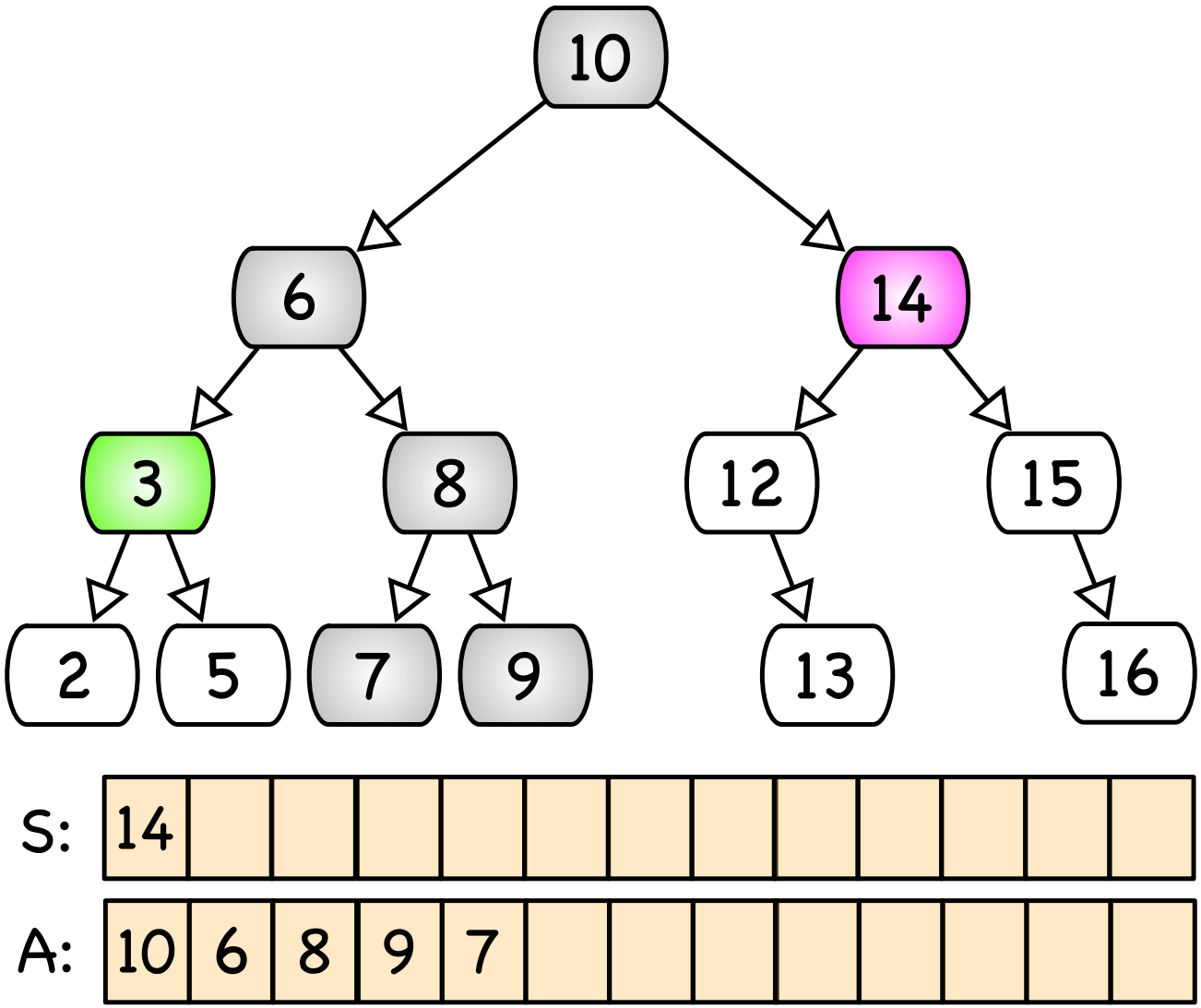
Tiefensuche - im Baum



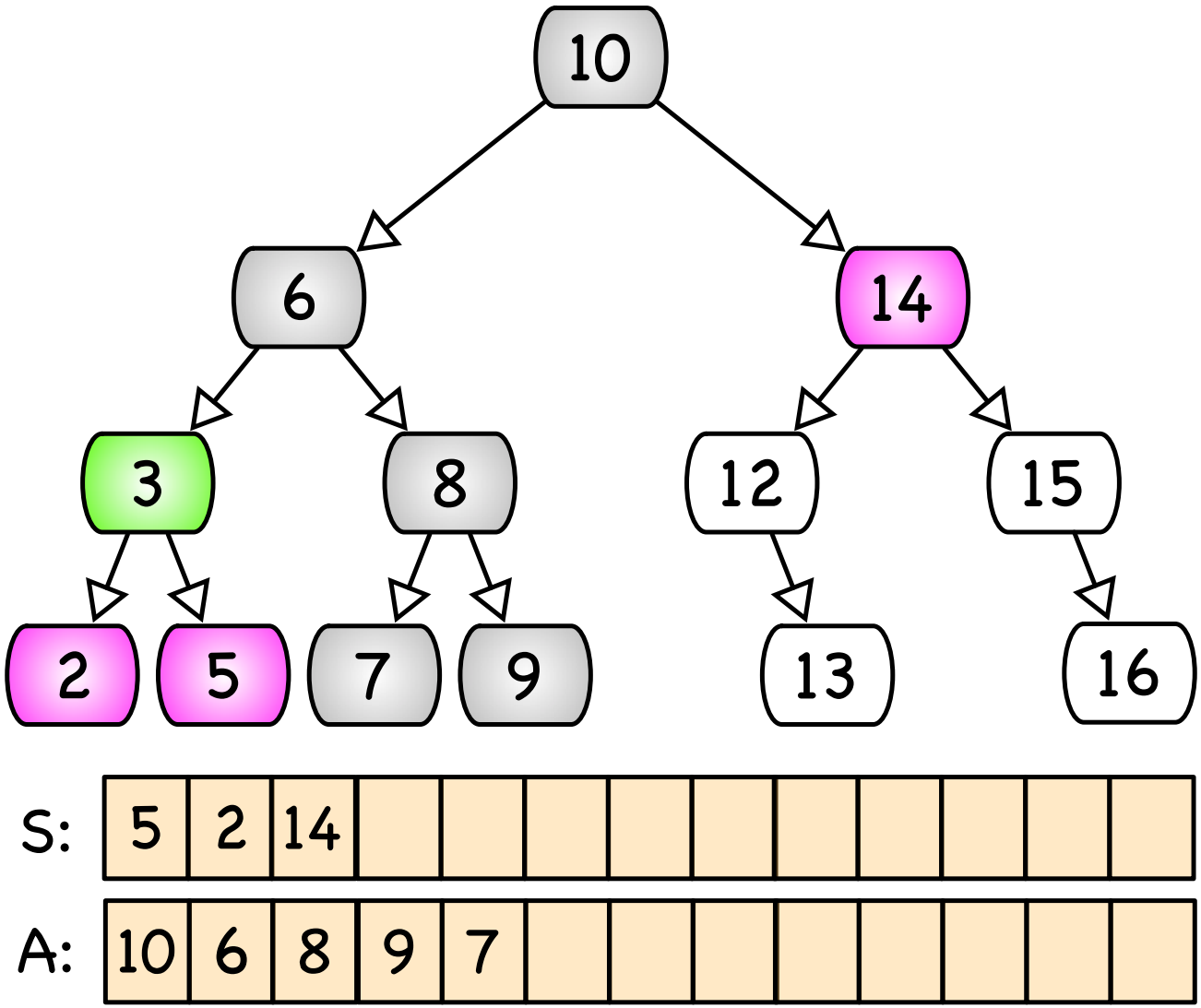
Tiefensuche - im Baum



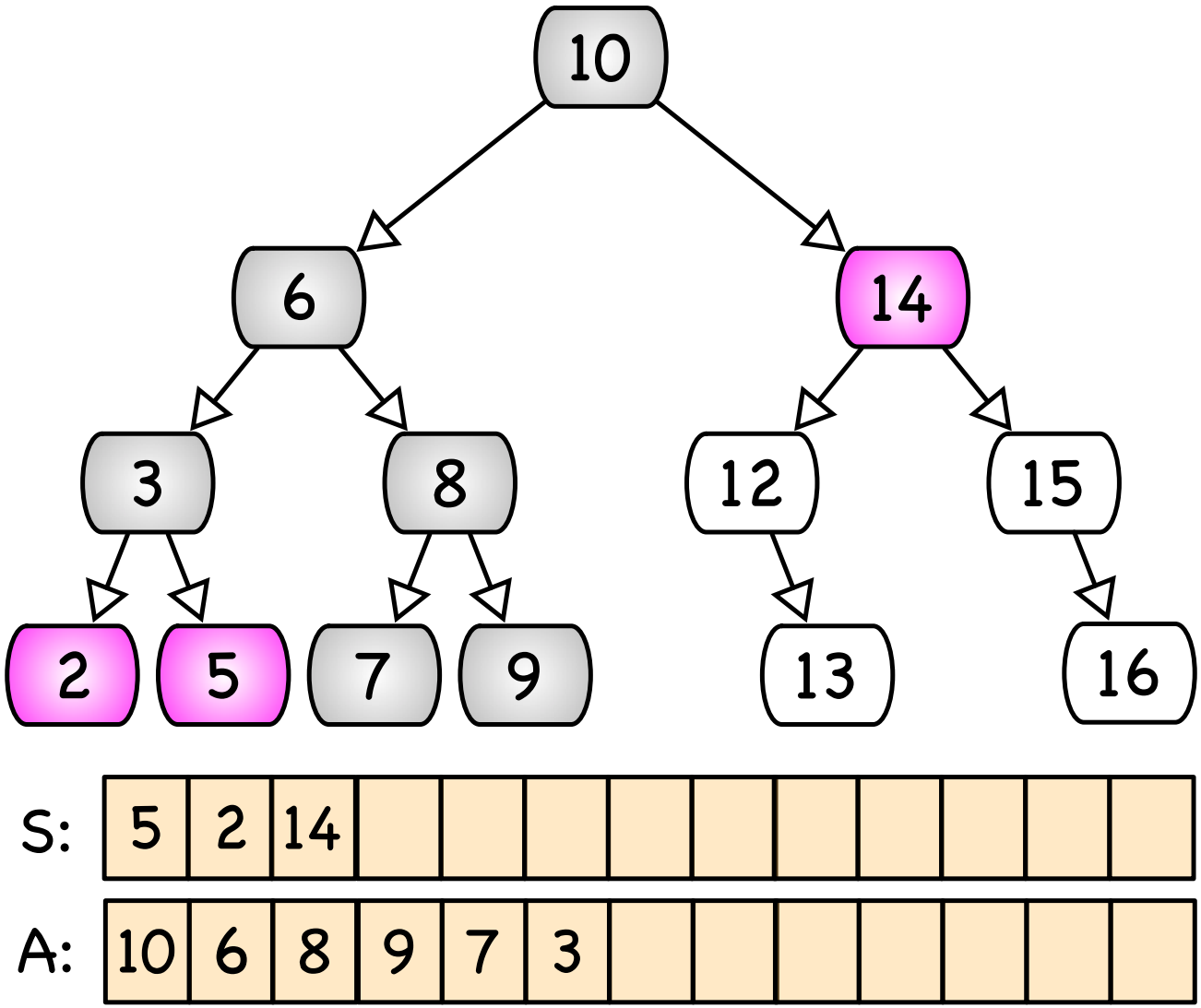
Tiefensuche - im Baum



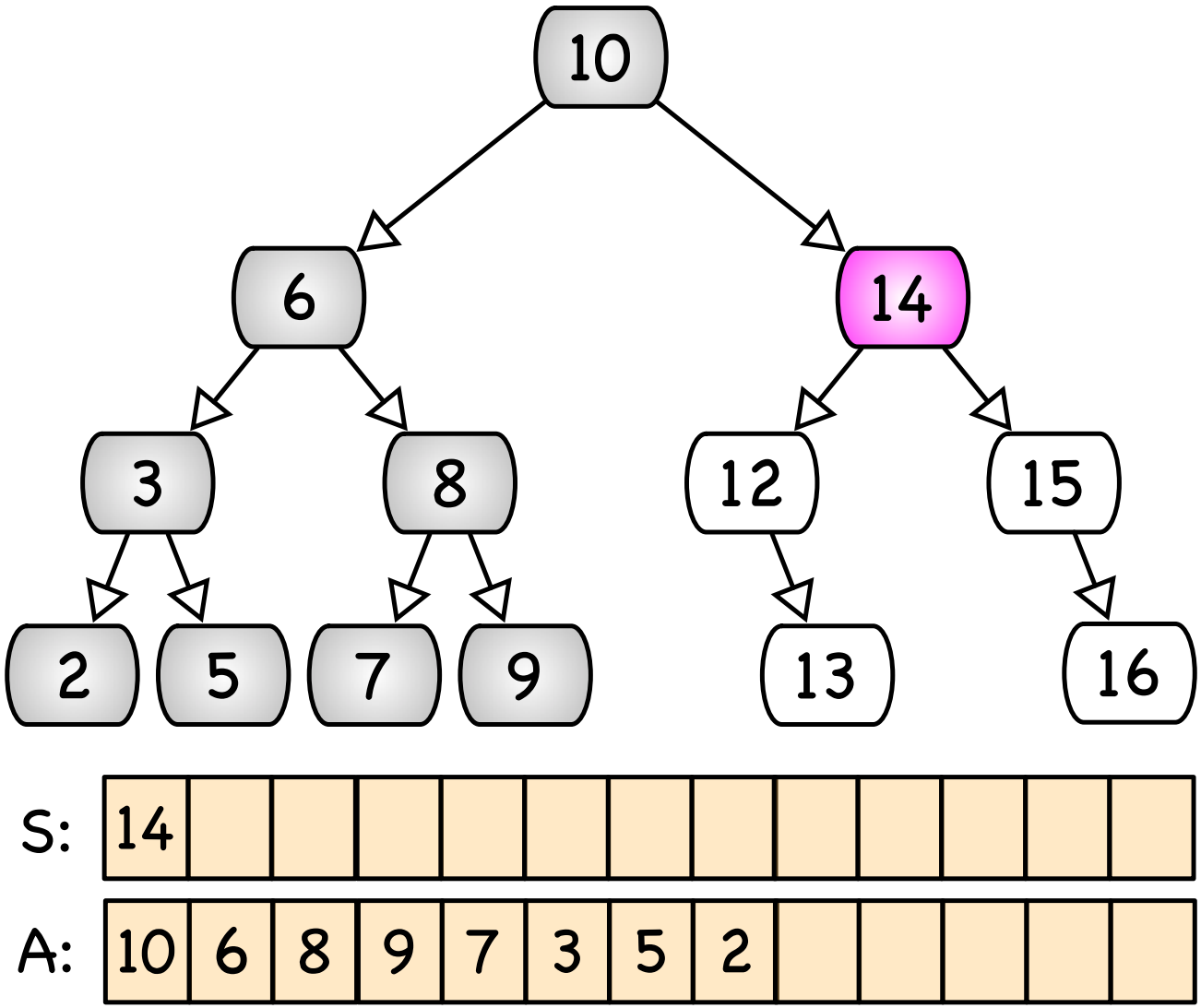
Tiefensuche - im Baum



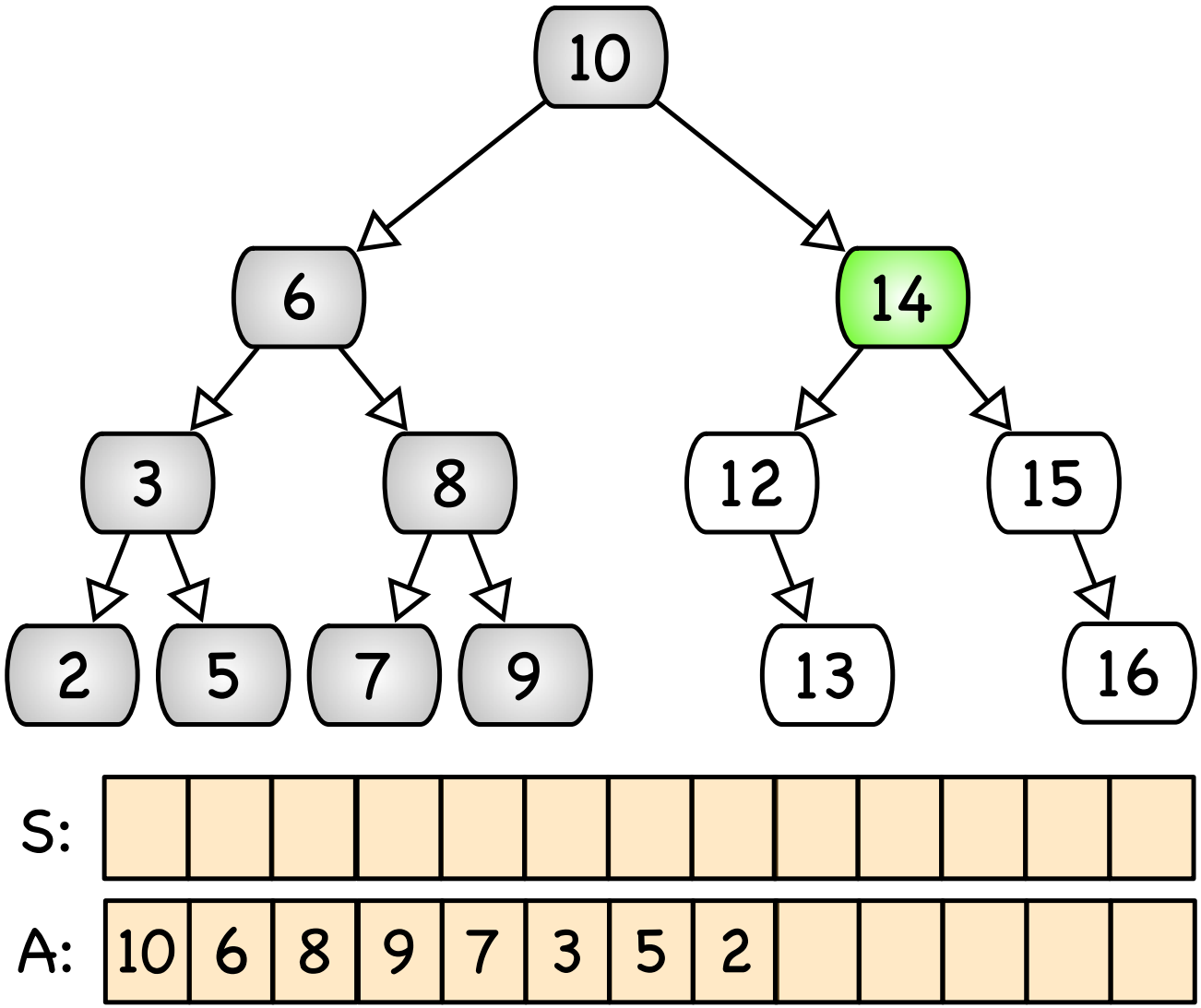
Tiefensuche - im Baum



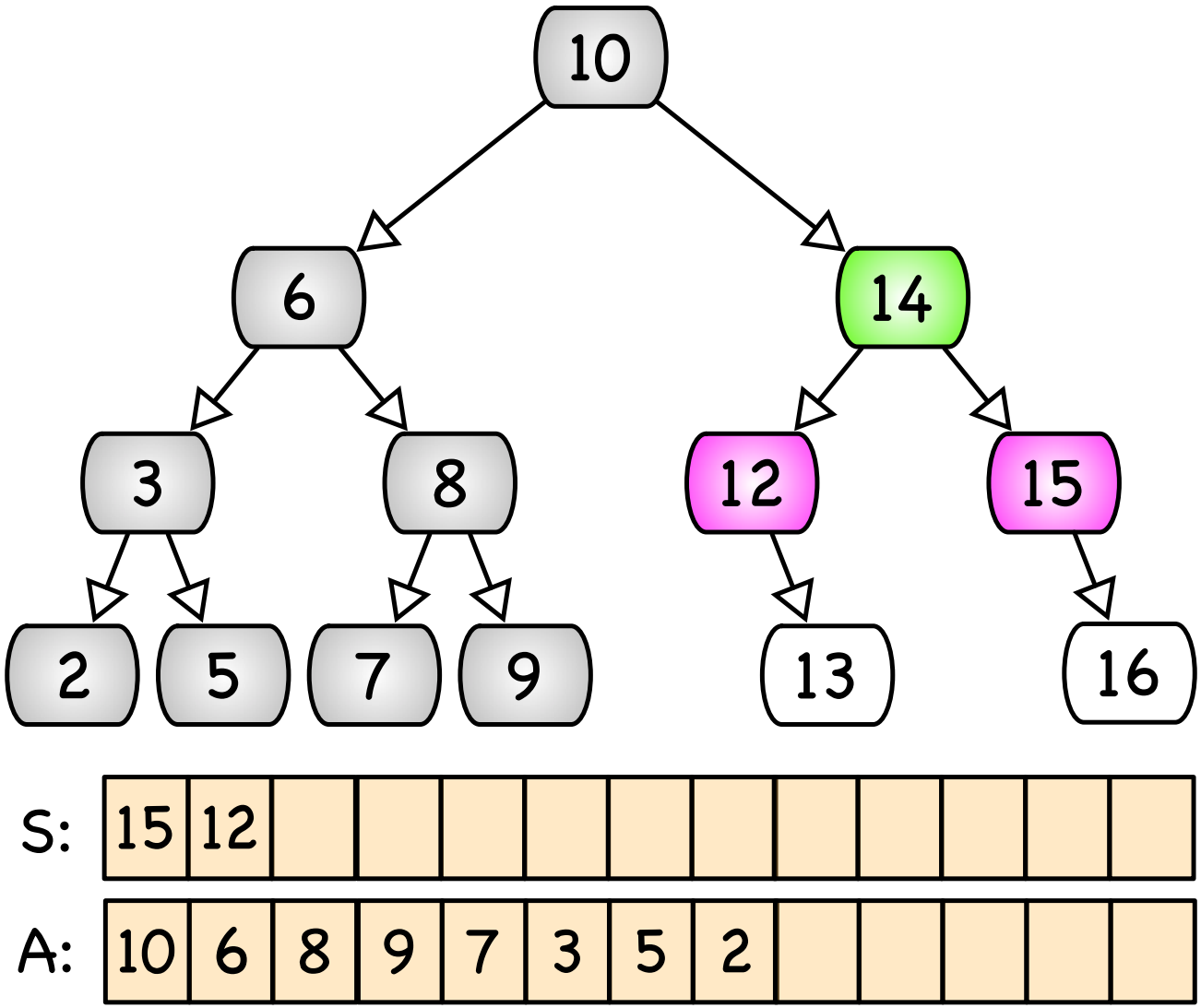
Tiefensuche - im Baum



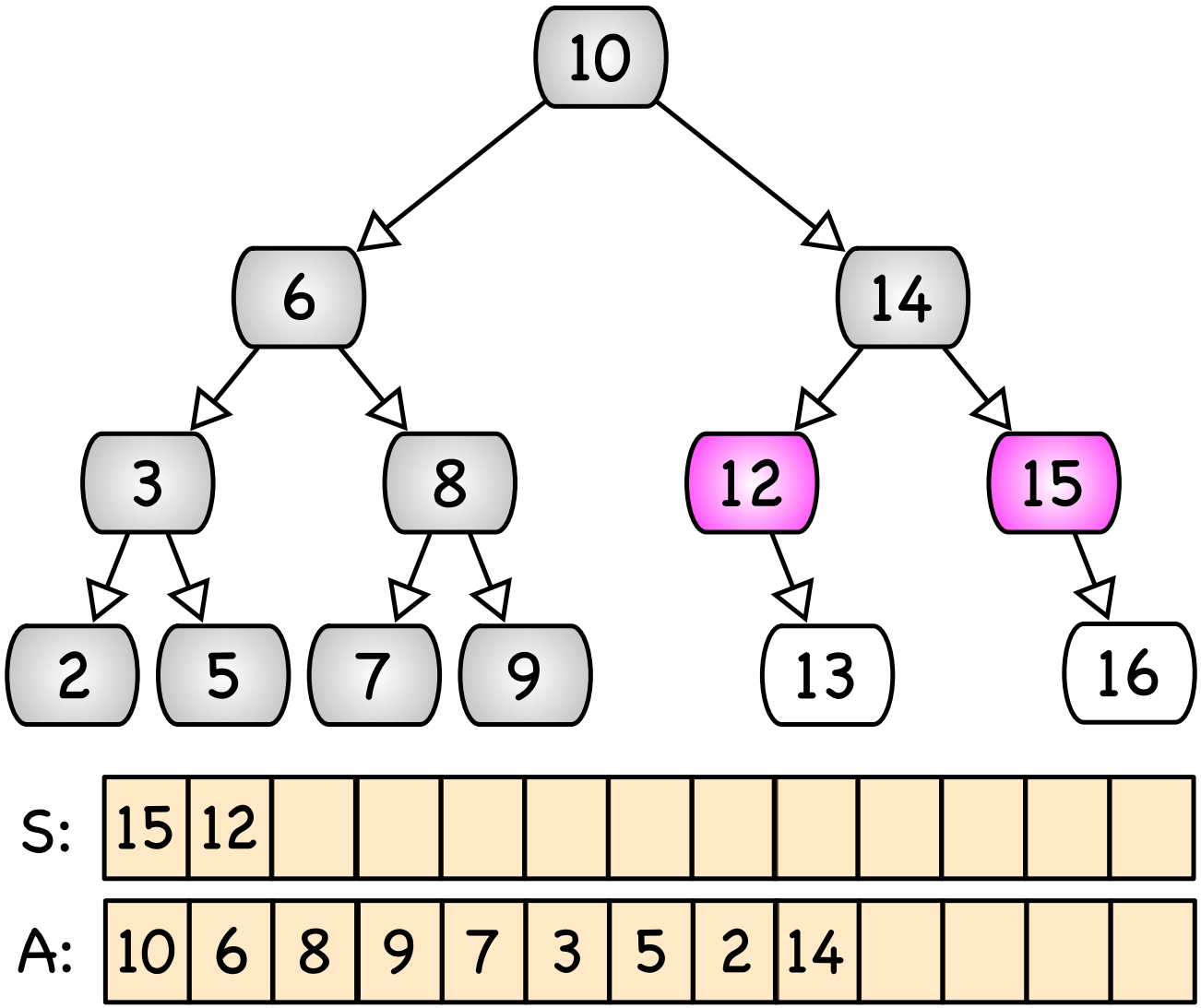
Tiefensuche - im Baum



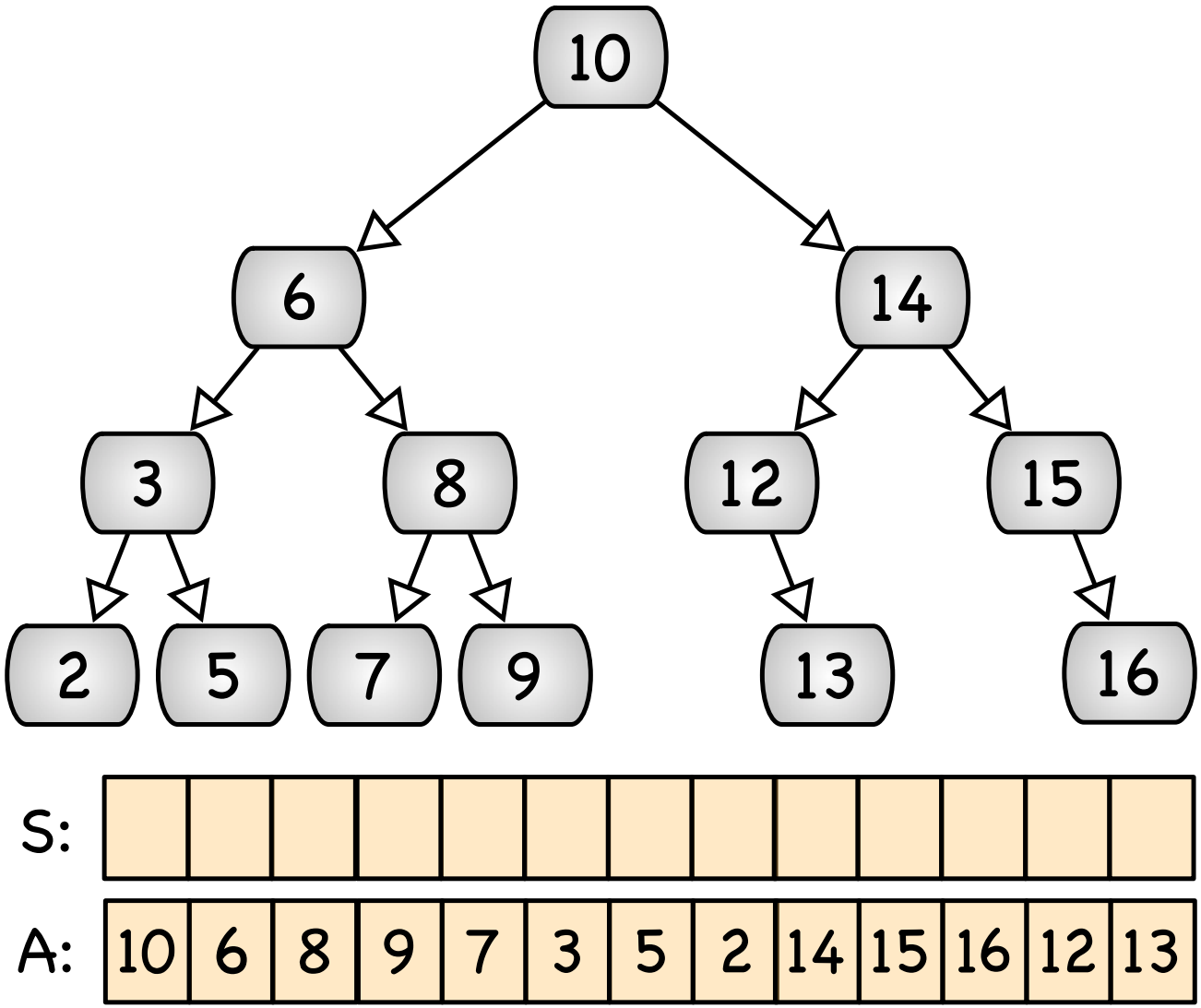
Tiefensuche - im Baum



Tiefensuche - im Baum



Tiefensuche - im Baum



Tiefensuche - Idee

Algorithmus 4 DFS(G, s)

```
1:  farbe[s] = pink
2:  S = ∅, push(S, s)
3:  while S ≠ ∅ do
4:      u = pop(S)
5:      for each v ∈ Adj[u] do
6:          if farbe[v] == weiss then
7:              farbe[v] = pink
8:              push(S, v)
9:          end if
10:     end for
11:     farbe[u] = schwarz
12: end while
```

Tiefen- vs. Breitensuche

Beobachtung

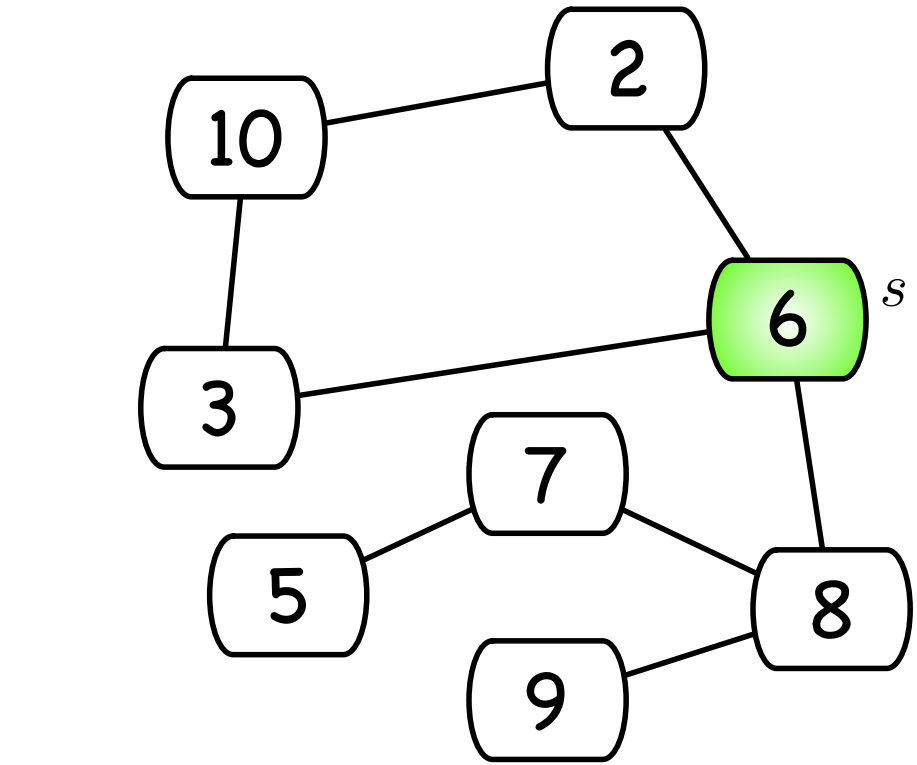
Der einzige Unterschied zur Breitensuche ist die Benutzung eines Stacks statt einer Queue!

Zur Nachbereitung

Anmerkung (zur Nachbereitung)

Wie bei der Breitensuche verzichten wir auch im nachfolgenden Beispiel auf die Angabe der Adjazenzlisten.

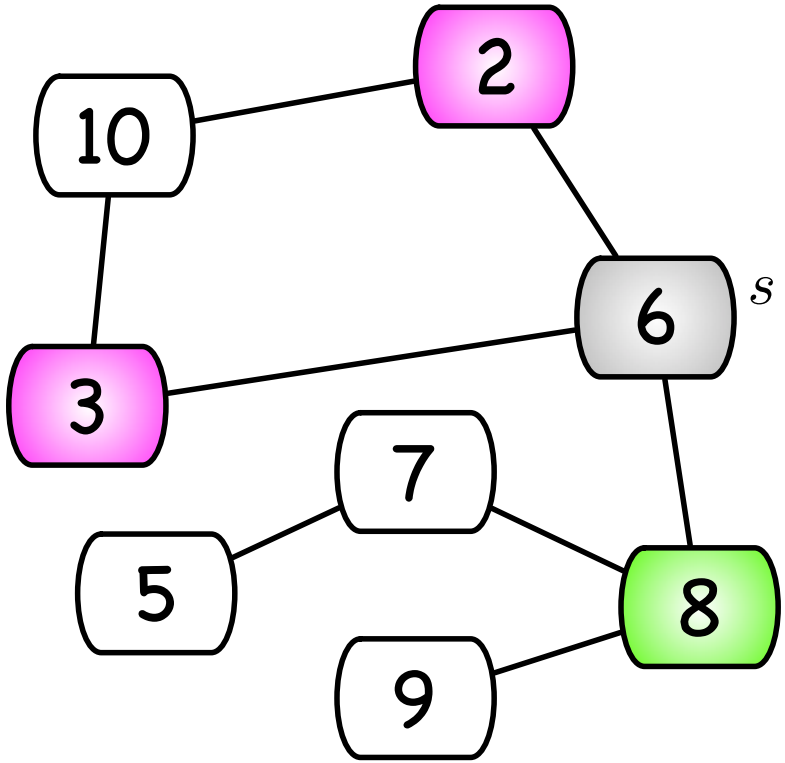
Tiefensuche - im Graphen

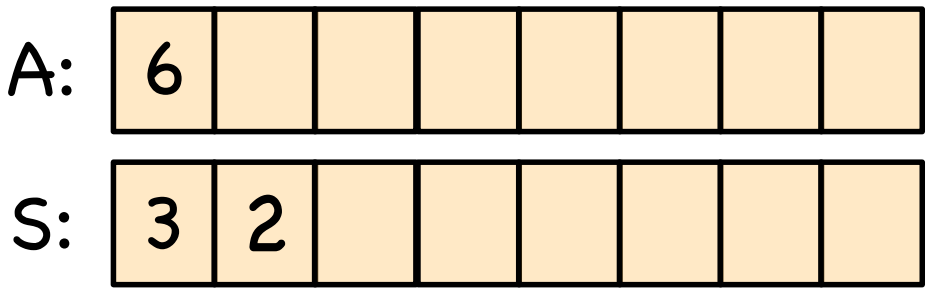


A:

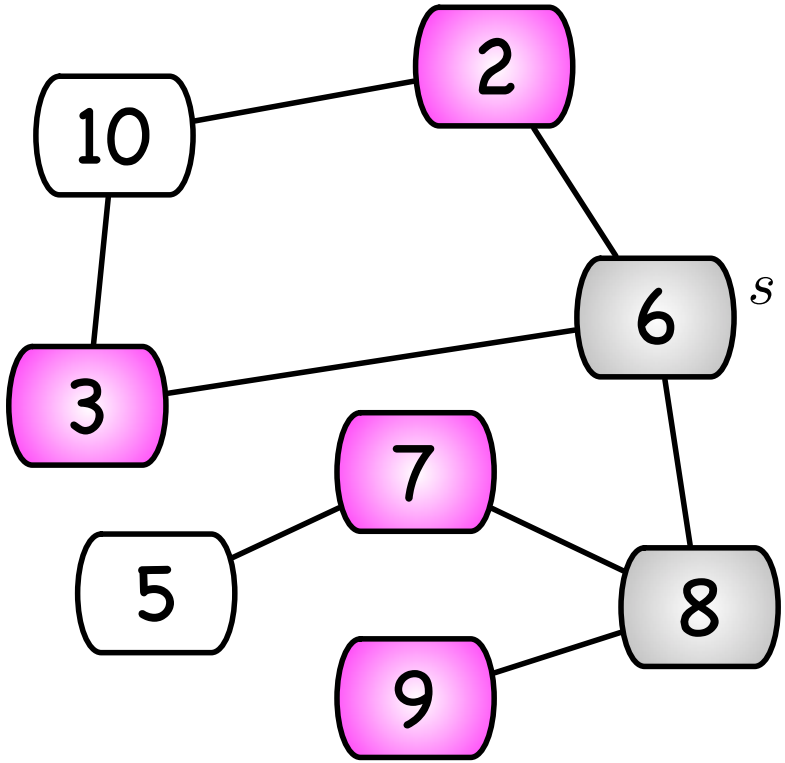
S:

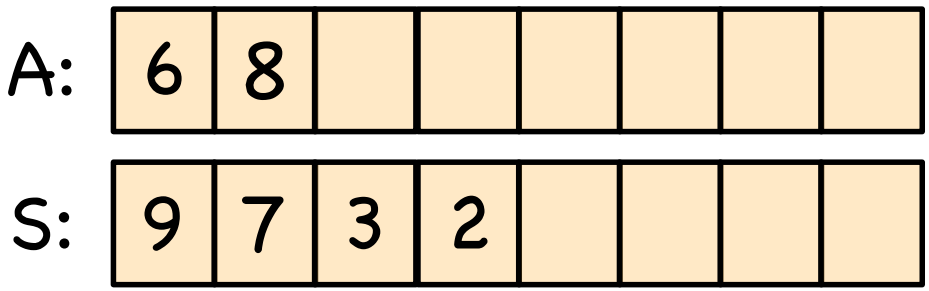
Tiefensuche - im Graphen



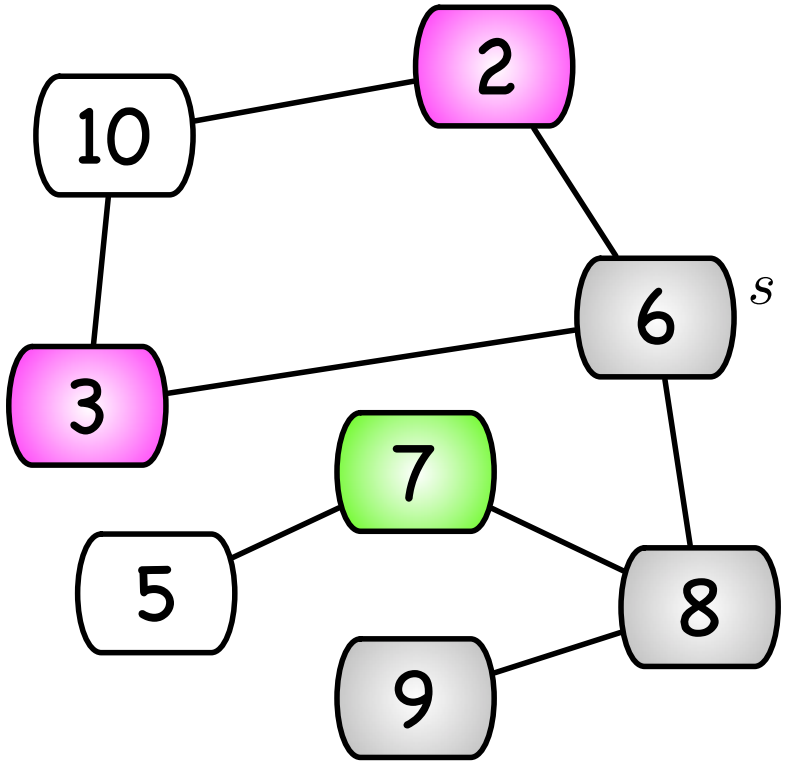


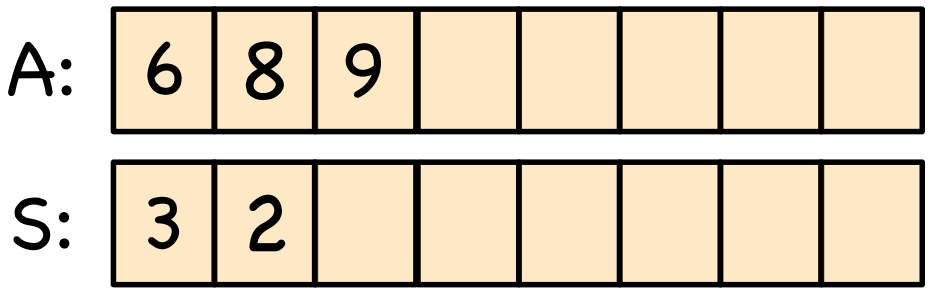
Tiefensuche - im Graphen



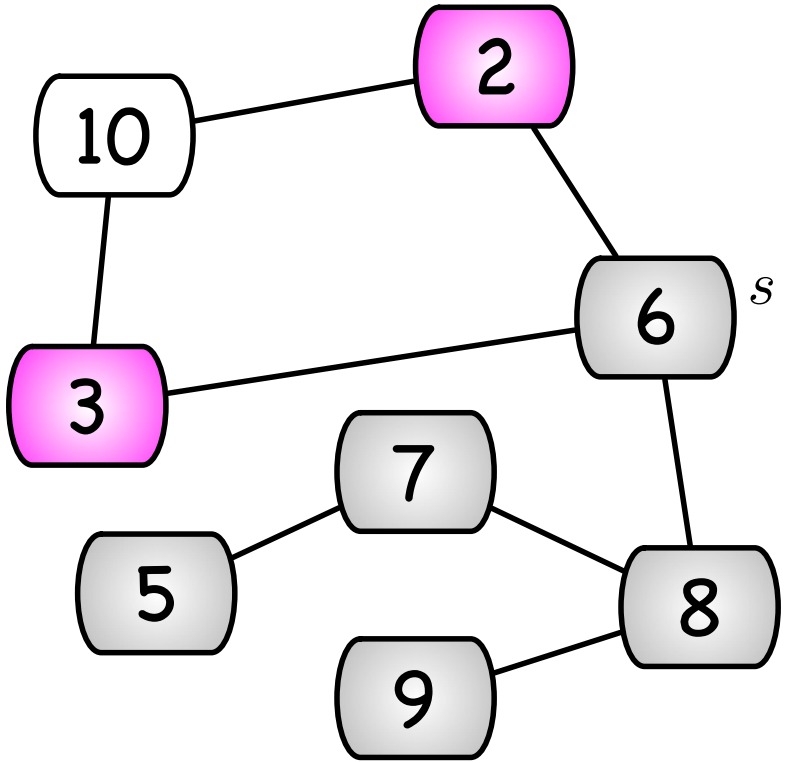


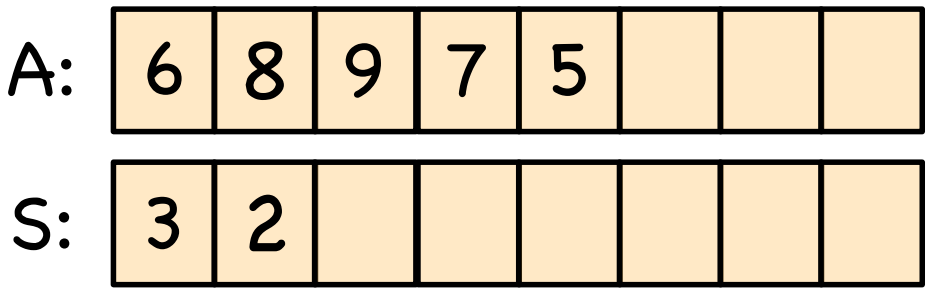
Tiefensuche - im Graphen



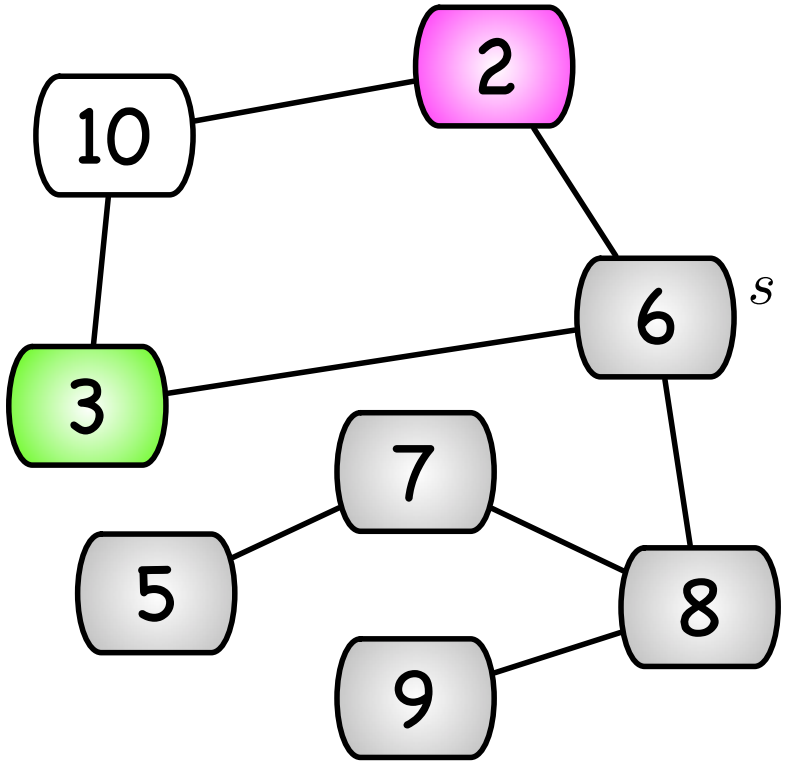


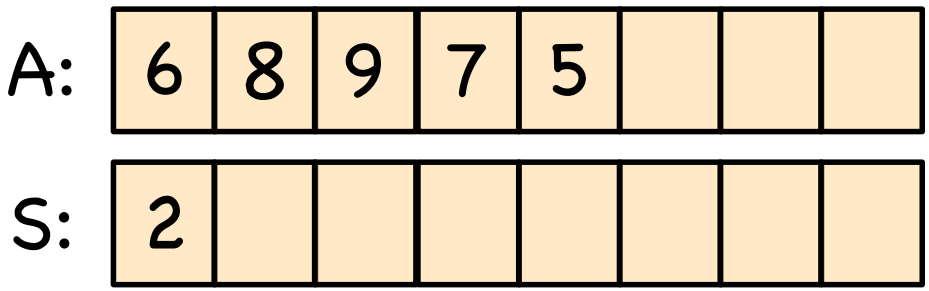
Tiefensuche - im Graphen



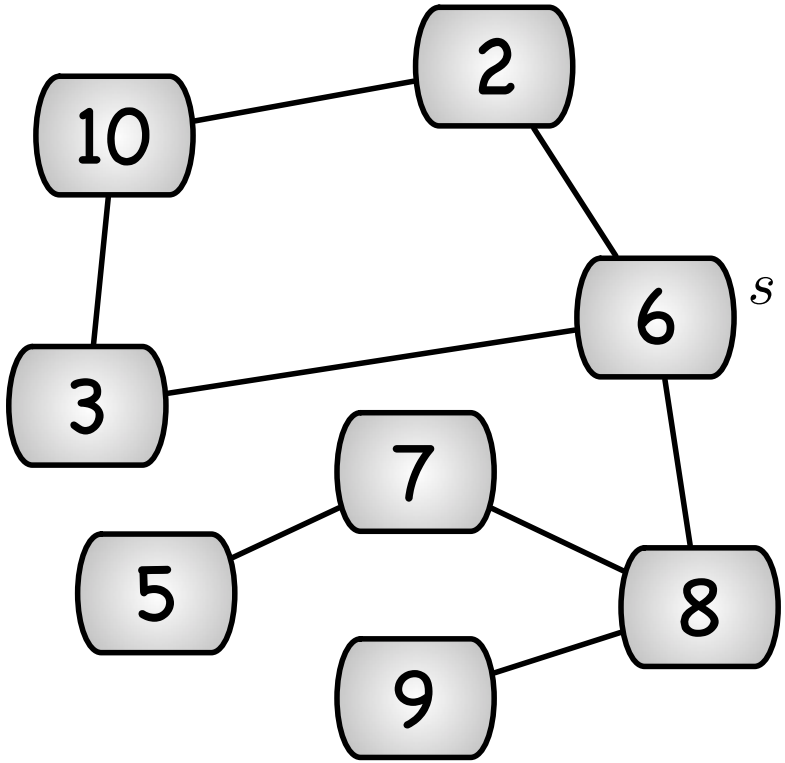


Tiefensuche - im Graphen





Tiefensuche - im Graphen



A:	6	8	9	7	5	3	10	2
S:								

Tiefensuche - mehr Informationen

Reichern wir nun die Tiefensuche etwas an, so können wir mehr Informationen gewinnen, was insb. später nützlich wird. (Der folgende Algorithmus ist wieder wie im [Cormen].)

Anmerkung

- Farben: weiss heißt 'noch nicht besucht', grau 'besucht, aber noch unbesuchte Nachbarn', schwarz 'besucht, alle Nachbarn verarbeitet'.
- $d[u]$ ist 'discovery' Zeit, $f[u]$ ist 'finished' Zeit. (Beachte: Immer wenn $d[u]$ oder $f[u]$ gesetzt werden, wird vorher die Zeit erhöht!)
- $\pi[u]$ ist der Vorgänger von u in diesem Tiefensuchbaum.

Tiefenensuche - Initialisierung

Algorithmus 5 DFS(G) - Initphase und Aufruf

```
1: for each u ∈ V(G) do
2:     farbe[u] = weiss, π[u] = nil
3: end for
4: zeit = 0
5: for each u ∈ V(G) do
6:     if farbe[u] == weiss then
7:         DFS_VISIT(u)
8:     end if
9: end for
```

Tiefensuche - Hauptroutine

Algorithmus 6 DFS_VISIT(u) - Teil 2, Hauptteil

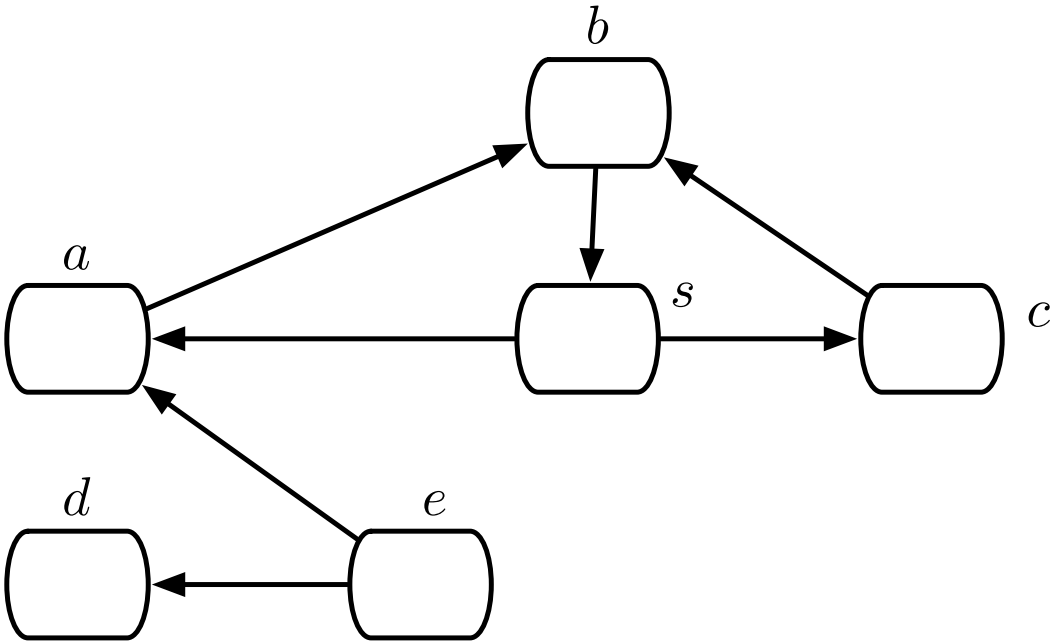
```
1:  farbe[u] = grau
2:  zeit = zeit + 1
3:  d[u] = zeit
4:  for each v ∈ Adj[u] do
5:      if farbe[v] == weiss then
6:          π[v] = u
7:          DFS_VISIT(v)
8:      end if
9:  end for
10: farbe[u] = schwarz
11: zeit = zeit + 1
12: f[u] = zeit
```

Zur Nachbereitung

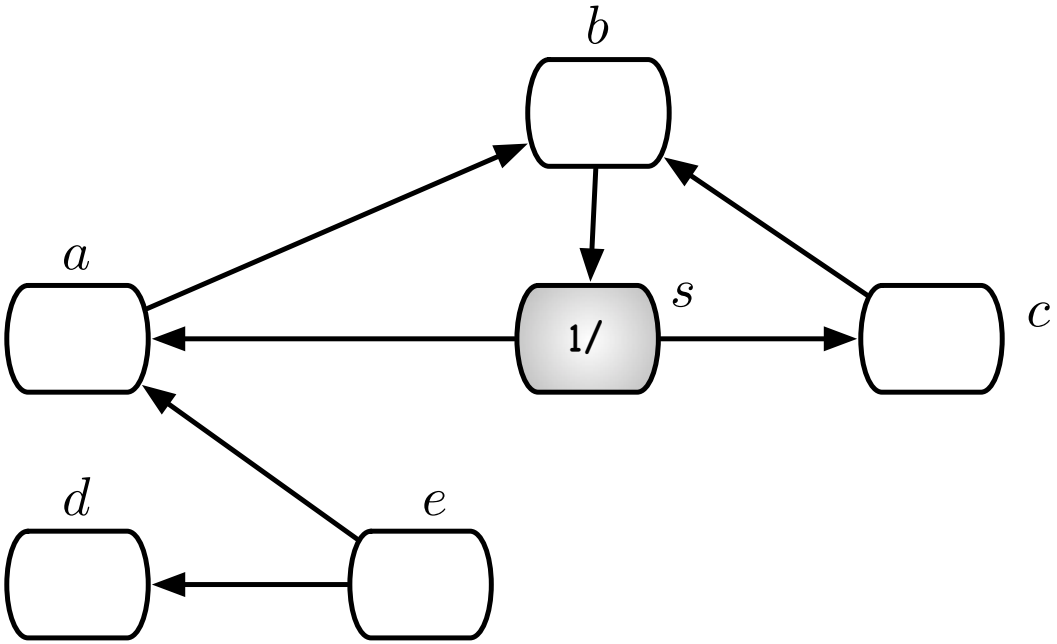
Anmerkung (zur Nachbereitung)

Für das nachfolgende Beispiel verzichten wir erneut auf die Angabe der Adjazenzlisten.
Die Zahlen x/y in einem Knoten v sind die Werte $d[v](= x)$ und $f[v](= y)$. Die Farbe 'grün' in den Bildern entspricht wieder der Farbe 'schwarz' im Quellcode, dicke Kanten stellen wieder Baumkanten dar. Die gestrichelten Kanten, sind entdecke Kanten, die als Rückwärtskanten (B) oder Querkanten (C) klassifiziert worden. Ferner gibt es Vorwärtskanten, die im Beispiel aber nicht auftreten. Eine Kante von s zu b wäre bei ansonsten gleichen Ablauf (d.h. wenn zuerst die Kante (s, a) gewählt wird) eine Vorwärtskante.

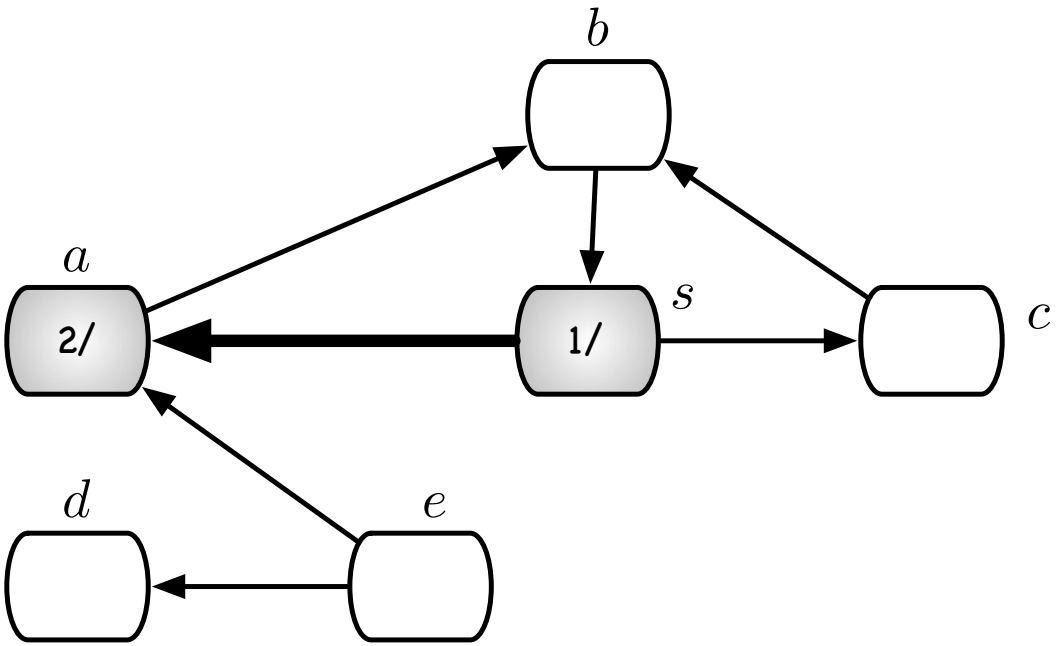
Tiefensuche - Beispiel



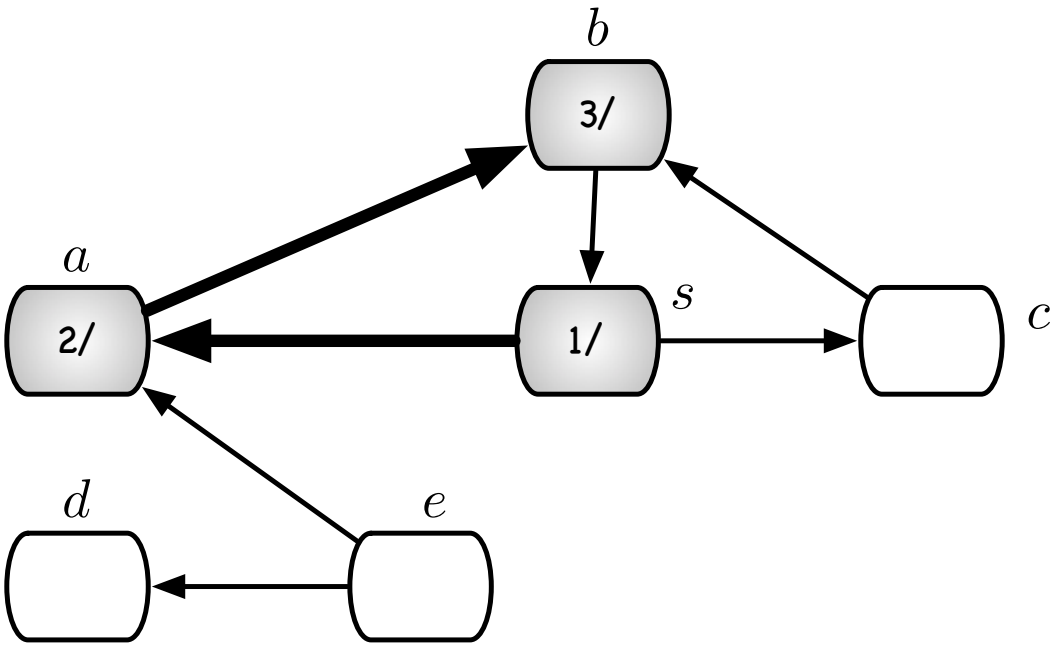
Tiefensuche - Beispiel



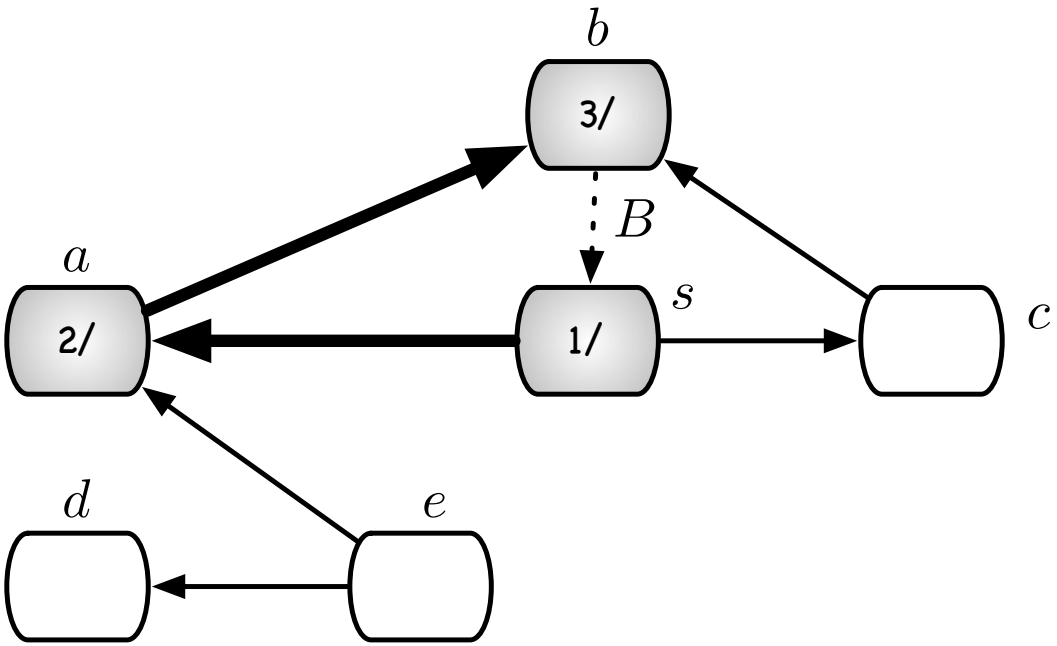
Tiefensuche - Beispiel



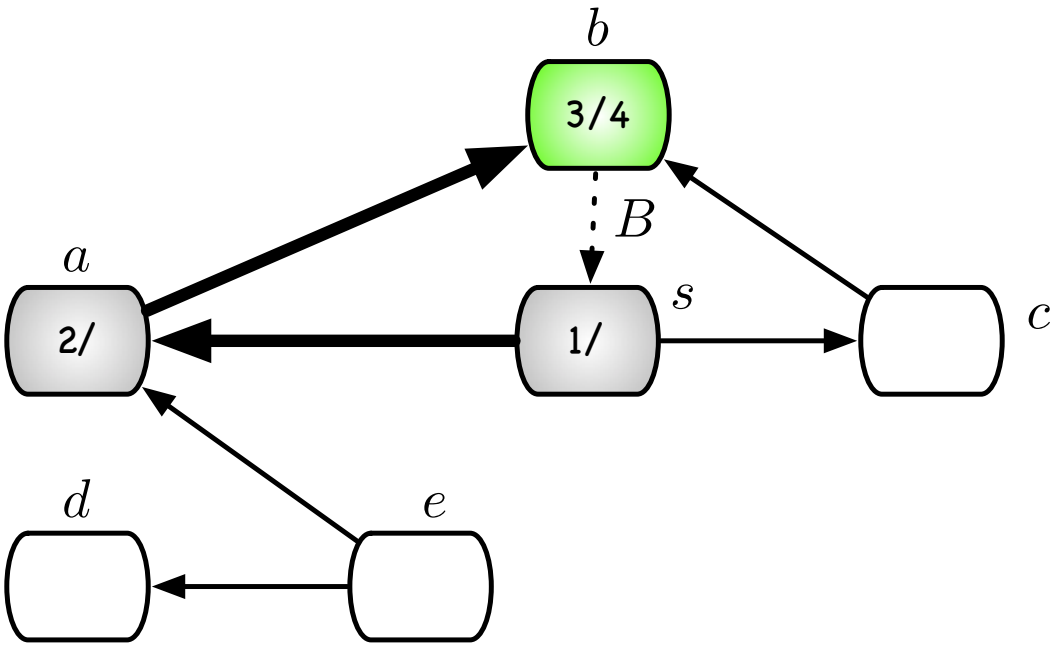
Tiefensuche - Beispiel



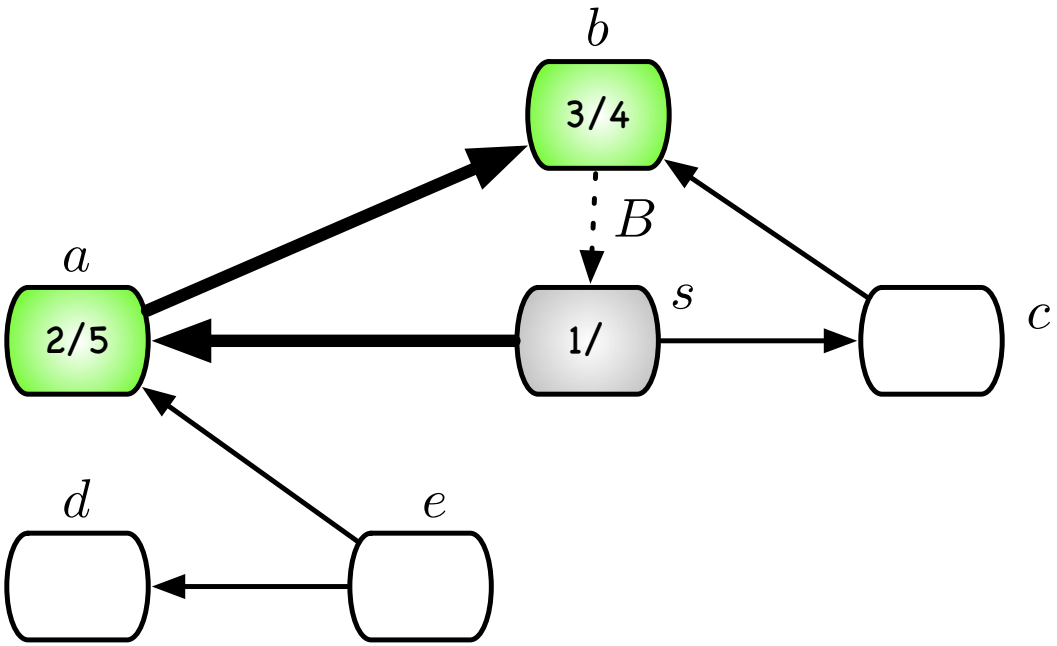
Tiefensuche - Beispiel



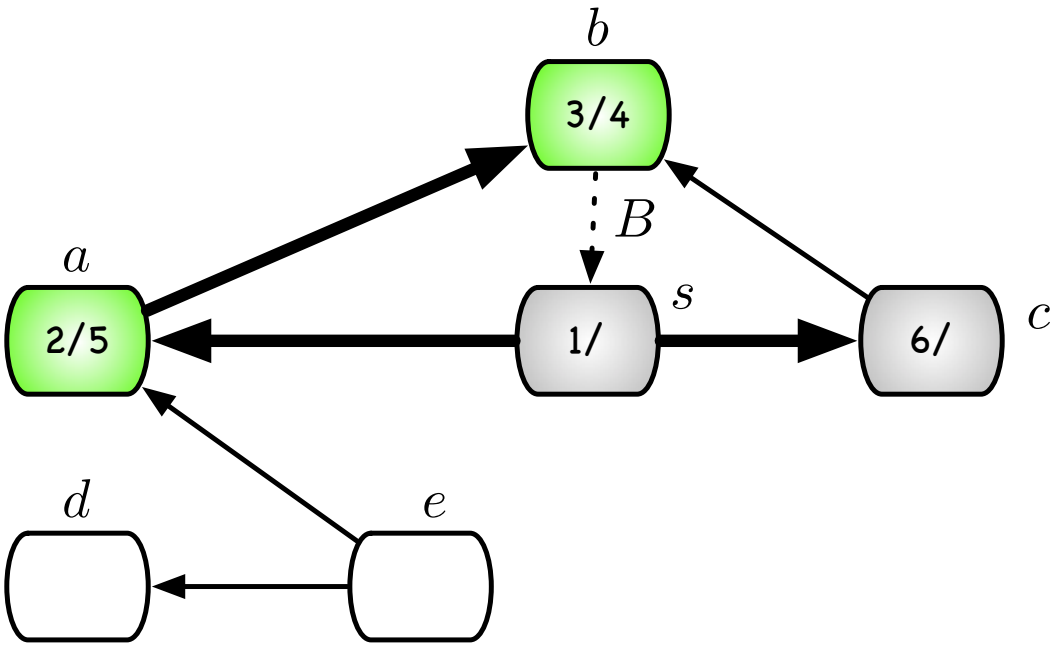
Tiefensuche - Beispiel



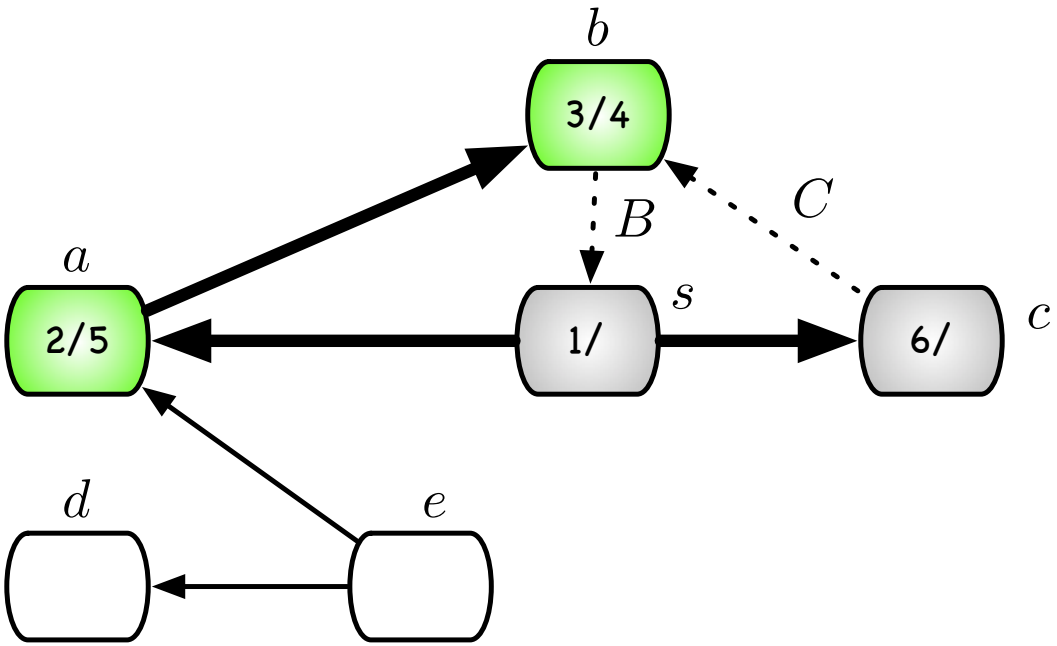
Tiefensuche - Beispiel



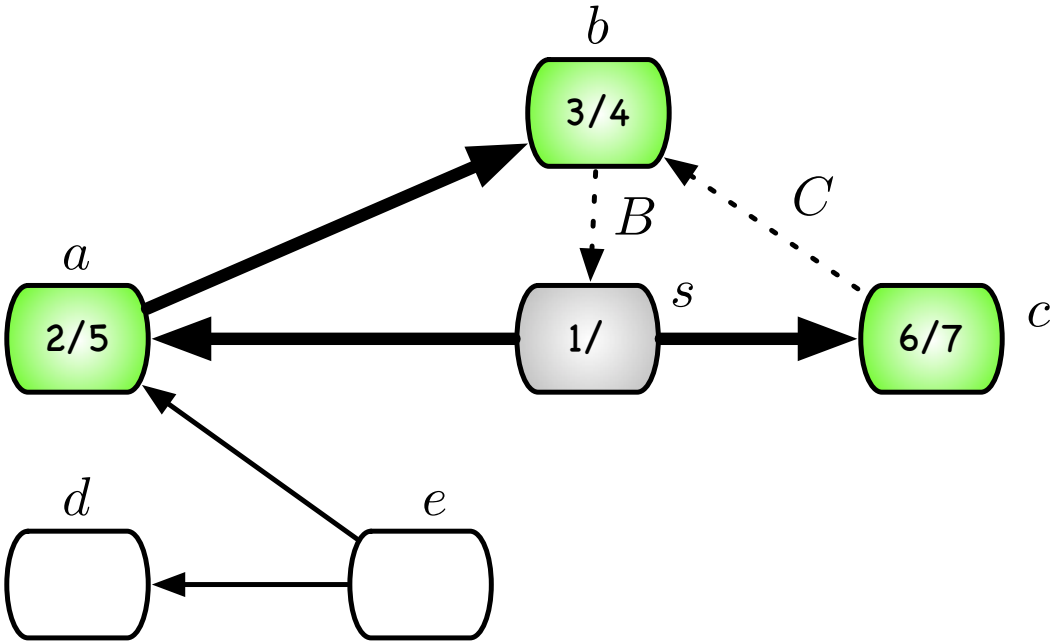
Tiefensuche - Beispiel



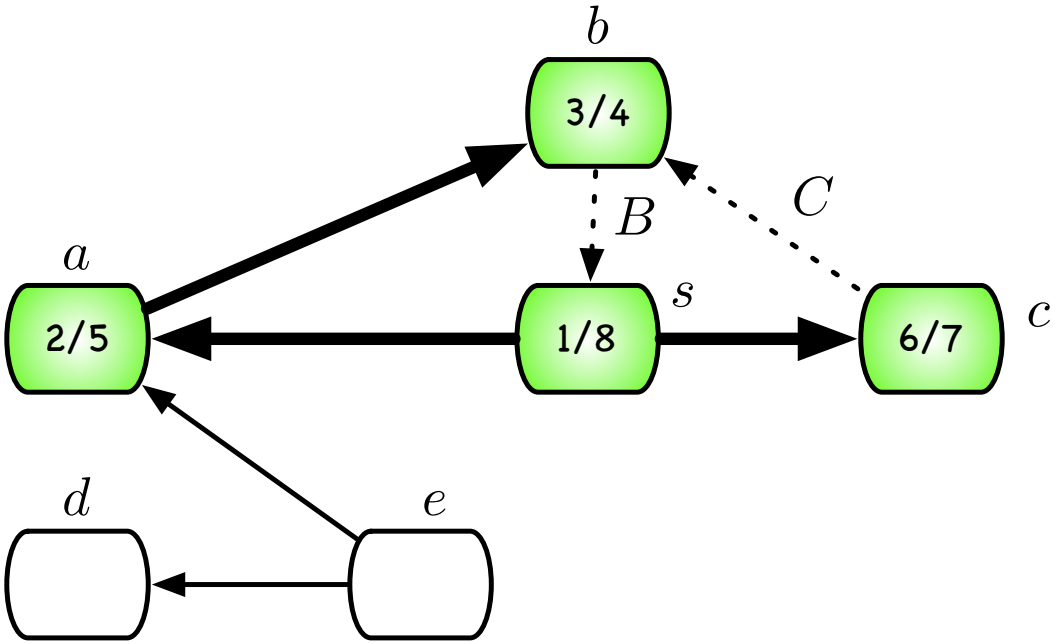
Tiefensuche - Beispiel



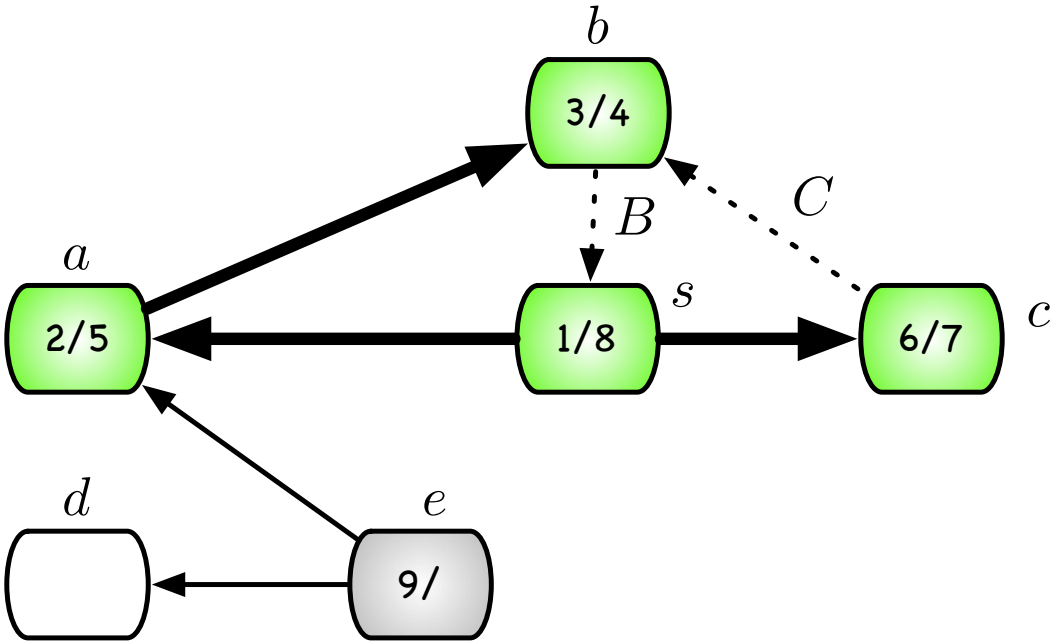
Tiefensuche - Beispiel



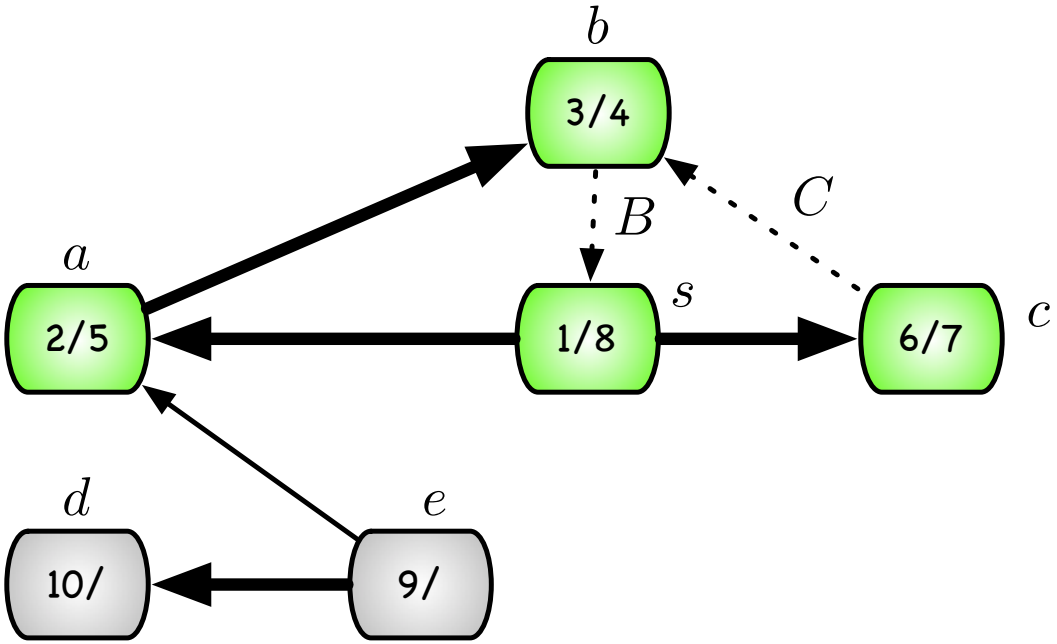
Tiefensuche - Beispiel



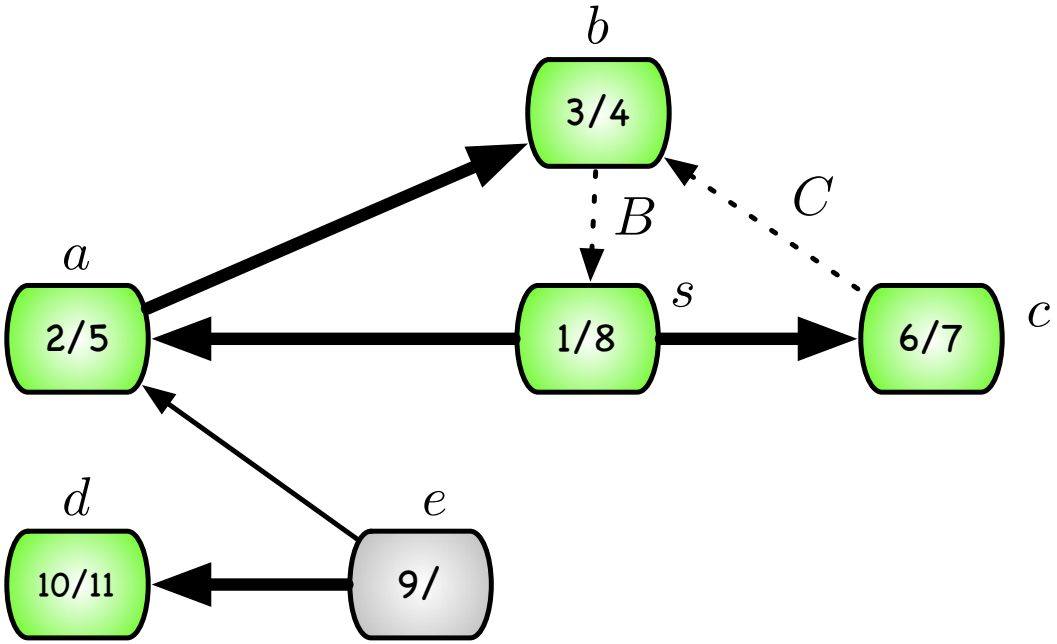
Tiefensuche - Beispiel



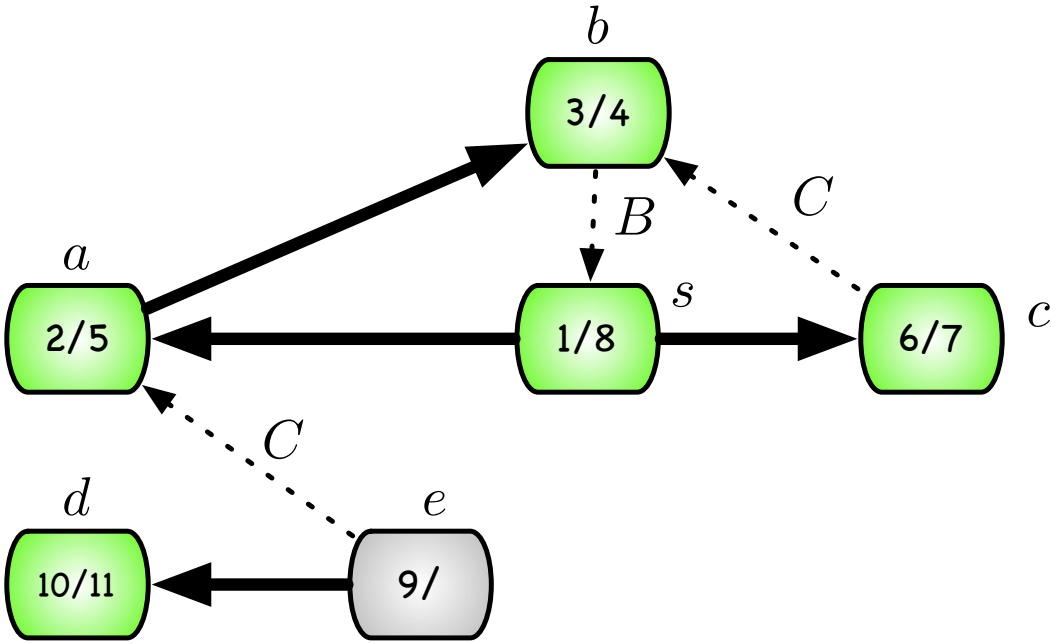
Tiefensuche - Beispiel



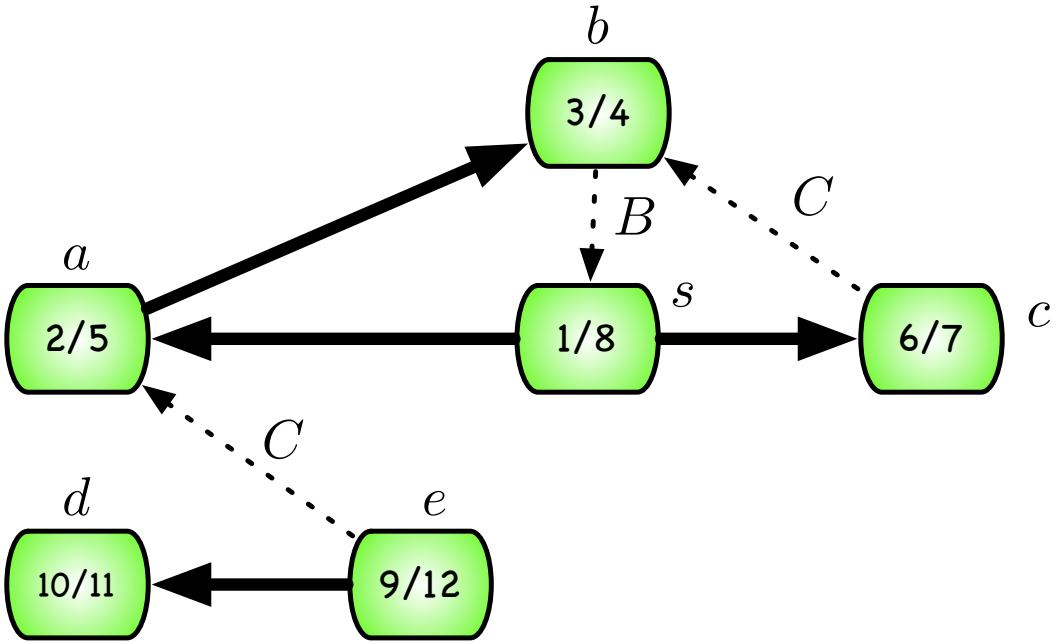
Tiefensuche - Beispiel



Tiefensuche - Beispiel



Tiefensuche - Beispiel



Tiefensuche - Besondere Kanten

Man kann bei der Tiefensuche die Kanten klassifizieren je nachdem, welche Farbe der Knoten v bei einer Kanten (u, v) hat (u ist der Knoten der gerade bearbeitet wird, v ist in $Adj[u]$):

- farbe$[v]$ = weiss: Baumkante.
- farbe$[v]$ = grau: Rückwärtskante.
- farbe$[v]$ = schwarz: Vorwärts- oder Querkante.
  - Vorwärtskante, falls $d[u] < d[v]$.
  - Querkante, falls $d[u] > d[v]$.

Anmerkung (zur Nachbereitung)

Im Beispiel eben ist dies bereits geschehen!

Tiefensuche - Besondere Kanten

Dabei ist

- Baumkante. Kanten, die zum Baum gehören. v wurde bei der Sondierung der Kante (u, v) entdeckt (erstmalig besucht).
- Rückwärtskante. Kante (u, v), die u mit einem Vorfahren v im Tiefensuchbaum verbindet.
- Vorwärtskante. (Nicht-Baum-)Kante (u, v), die u mit einem Nachfahre in v im Tiefensuchbaum verbindet.
- Querkanten. Die übrigen Kanten. Im gleichen Tiefensuchbaum, wenn der eine Knoten nicht Vorfahre des anderen ist oder zwischen zwei Knoten verschiedener Tiefensuchbäume (im Wald).

Anmerkung

Bei einem ungerichteten Graphen gibt es nur Baum- und Rückwärtskanten.

Analyse

Satz

Die Tiefensuche ist korrekt und ihre Laufzeit ist wieder in $O(V + E)$.

Breiten- und Tiefensuche - Zusammenfassung

Zusammenfassung

- Breitensuche nutzt eine Queue, Tiefensuche einen Stack. (Die Tiefensuche wie im [Cormen] kommt ohne Stack aus, arbeitet dafür aber rekursiv und mit Zeitstempeln.)
- Beide Operationen laufen in $\Theta(V + E)$, wenn der Graph mittels Adjazenzlisten gegeben ist.
- Die Breitensuche hat (meist) nur einen Startknoten. Sie wird oft verwendet, um kürzeste Abstände und den Vorgängerteilgraphen zu ermitteln.
- Die Tiefensuche ermittelt einen Tiefensuchwald (mehrere Startknoten). Sie wird oft als Unterroutine in anderen Algorithmen verwendet.

Problemstellung

Die Tiefensuche wird z.B. beim topologischen Sortieren benutzt...

Problemstellung

... oder beim Ermitteln der Zusammenhangskomponenten eines Graphen.

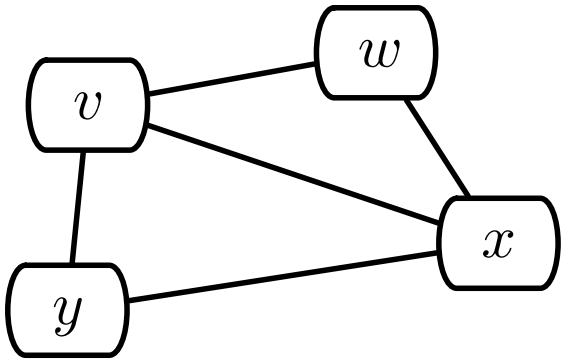
Problemstellung

Zu beidem gleich mehr...

Darstellung von Graphen - Zusammenfassung

- Adjazenzmatrix: $|V| \times |V|$-Matrix $A = (a_{ij})$ mit $a_{ij} = 1$ falls $(i, j) \in E$ und 0 sonst. Größe in $\Theta(V^2)$.
- Adjazenzliste: Liste $Adj[v]$ für jeden Knoten $v \in V$ in der die Knoten, die mit v adjazent sind gespeichert sind. Größe in $\Theta(V + E)$.
- Bei einer Adjazenzmatrix kann man schnell herausfinden, ob zwei Knoten benachbart sind oder nicht. Dafür ist es langsamer alle Knoten zu bestimmen, die mit einem Knoten benachbart sind. (Bei Adjazenzlisten genau andersherum.)
- Beide Darstellungen sind ineinander transformierbar.
- Beide Darstellungen sind leicht auf den Fall eines gewichteten Graphen anpassbar.

Darstellung von Graphen - Adjazenzmatrix

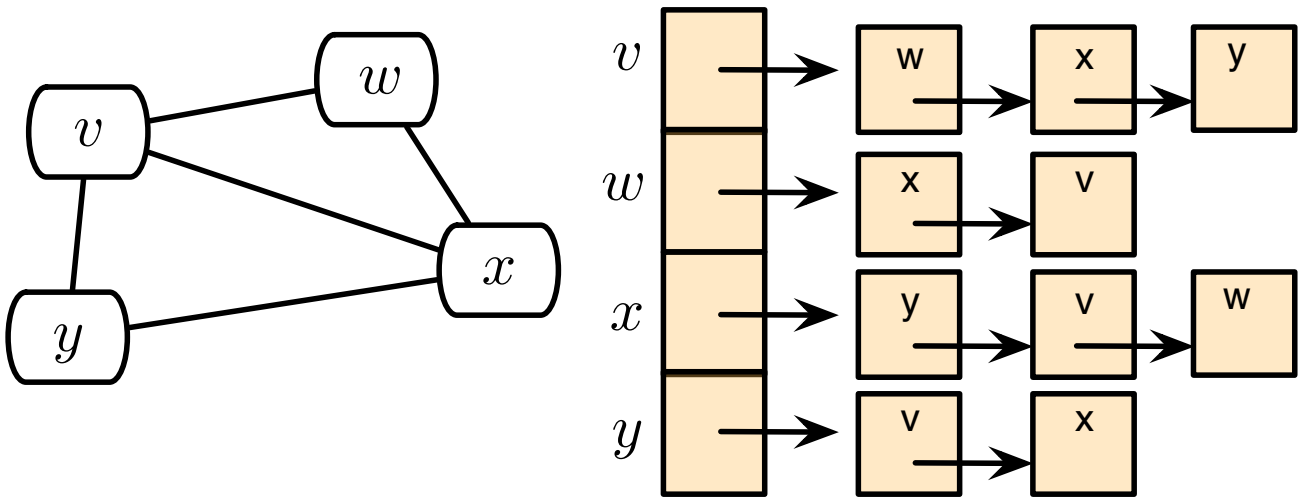


	v	w	x	y
v	0	1	1	1
w	1	0	1	0
x	1	1	0	1
y	1	0	1	0

Bei einer Adjazenzmatrix hat man eine $n \times n$-Matrix, bei der an der Stelle (i, j) genau dann eine 1 steht, wenn v_i und v_j verbunden sind.

Der Speicherplatzbedarf ist in $\Theta(V^2)$ (unabhängig von der Kantenzahl).

Darstellung von Graphen - Adjazenzlisten



Bei der Adjazenzlistendarstellung haben wir ein Array von $|V|$ Listen, für jeden Knoten eine. Die Adjazenzliste $Adj[v]$ zu einem Knoten v enthält alle Knoten, die mit v adjazent sind.

Bei einem gerichteten Graphen ist die Summe aller Adjazenzlisten $|E|$, bei einem ungerichteten Graphen $|2E|$. Der Speicherplatzbedarf ist folglich $\Theta(V + E)$.

Breitensuche - Die Idee

Gegeben ein Graph G und ein Startknoten s 'entdeckt' die Breitensuche alle Knoten, die von s aus erreichbar sind. Zudem wird der Abstand (in Kanten) von s aus berechnet (tatsächlich wird sogar der *kürzeste Abstand* ermittelt).

Man kann auch zusätzlich einen 'Breitensuchbaum' (mit Wurzel s) und damit die kürzesten Pfade von s zu den anderen Knoten ermitteln.

Der Algorithmus entdeckt zunächst die Knoten mit Entfernung k und dann die mit Entfernung $k + 1$, daher der Name. Er geht erst in die Breite...

Breitensuche - Die Idee

- Starte mit Knoten s in einer Queue Q.
- Wiederhole solange Q nicht leer...
  - Nimm vordersten Knoten v aus Q.
  - (Bearbeite diesen und Färbe diesen so, dass er nicht wieder besucht wird.)
  - Tue alle Nachbarn von v, die bisher nicht besucht wurden in die Queue.

Anmerkung

Durch die Queue wird sichergestellt, dass die Knoten 'in der Breite' besucht werden.

Tiefensuche - Die Idee

Bei der Tiefensuche geht man einfach immer weiter noch ungeprüfte Kanten entlang, so lange dies möglich ist. Mein folgt also 'einem Pfad' so lange es möglich ist. Erst wenn man einen Konten erreicht von dem aus man nur bereits besuchte Knoten erreichen kann, geht man einen Schritt zurück.

Bleiben unentdeckte Knoten übrig wählt man diese als neue Startknoten. So kann sich (bei einem unzusammenhängenden Graphen) ein Tiefensuch*wald* ergeben. (Die Breitensuche kann man auch so anpassen, aber typischerweise werden die Algorithmen so benutzt.)

Tiefensuche - Die Idee

- Starte mit Knoten s in einem Stack S.
- Wiederhole solange S nicht leer...
  - Nimm obersten Knoten v aus S.
  - (Bearbeite diesen und Färbe diesen so, dass er nicht wieder besucht wird.)
  - Tue alle Nachbarn von v, die bisher nicht besucht wurden auf den Stack.

Anmerkung

Durch den Stack wird sichergestellt, dass die Knoten 'in der Tiefe' besucht werden.

Breiten- und Tiefensuche - Erweiterungen

- Die Breitensuche kann mit 'distance'-Stempeln angereichert werden, um so die kürzesten Abstände von s zu ermitteln.
- Die Tiefensuche kann mit 'discovery' und 'finished' Zeit angereichert werden. So können die Kanten klassifiziert werden und man kann z.B. Kreise finden und noch andere Dinge tun. Wie oben kann man mit dem Vorgänger einen Tiefensuchbaum konstruieren.

Breiten- und Tiefensuche - Zusammenfassung

Zusammenfassung

- Breitensuche nutzt eine Queue, Tiefensuche einen Stack. (Die Tiefensuche wie im [Cormen] kommt ohne Stack aus, arbeitet dafür aber rekursiv und mit Zeitstempeln.)
- Beide Operationen laufen in $\Theta(V + E)$, wenn der Graph mittels Adjazenzlisten gegeben ist.
- Die Breitensuche hat (meist) nur einen Startknoten. Sie wird oft verwendet, um kürzeste Abstände und den Vorgängerteilgraphen zu ermitteln.
- Die Tiefensuche ermittelt einen Tiefensuchwald (mehrere Startknoten). Sie wird oft als Unterroutine in anderen Algorithmen verwendet.

Auf einen Blick...

- Graphen
  - Grundlagen, Adjazenzmatrix, Adjazenzliste
  - Breiten- & Tiefensuche
  - Ausblick: Anwendung der Tiefensuche
  - Ausblick: Minimale Spannbäume