Basic graph algorithms

TECHNISCHE UNIVERSITÄT DARMSTADT

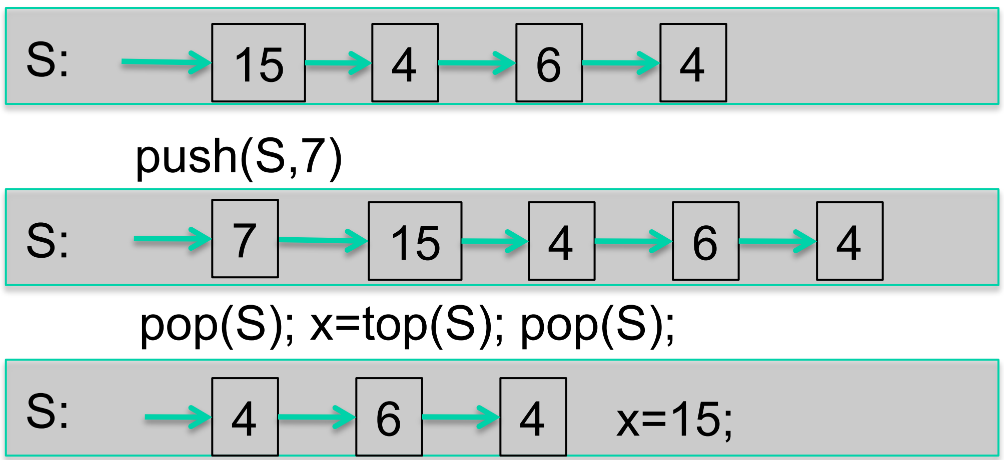
- **DFS going a different way**

  – Abstract data-types: stack und queue
    - Dynamic data container
    - Elements can be added or removed
    - Stack implements a last-in-first-out (LIFO) strategy
    - Queue implements s first-in-first-out (FIFO) strategy
    - Stack and queue can be manipulated with the help of pre-defined operations only.

Basic graph algorithms

- DFS going different way

  – Let S be a stack. Then there are 4 operations with runtime O(1):
    - empty() Is the stack empty?
    - push(S,x) Add element x to stack S
    - pop(S) remove the last added element.
    - x=top(S) Read the content of the latest added element

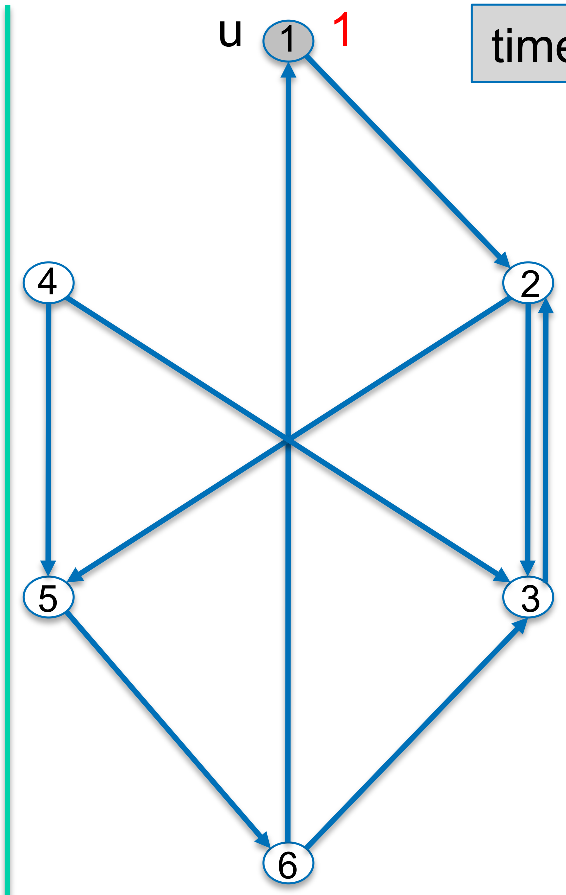


Basic graph algorithms

TECHNISCHE UNIVERSITÄT DARMSTADT

```
DFS(G)
for each node u∈V do
        color[u] := white;
time := 0; S:=∅;
for each node u∈V do
        if color[u]==white then DFS-Visit(u);
```

```
DFS-Visit(u)
push(S,u);
while not empty(S) do
    u:=top(S);
    if color[u]==white then
        color[u] := gray; time:=time+1; d[u]:=time;
        for each node v∈adj[u] do
            if color[v]==white push(S,v);
    else if color[u]==gray then
        color[u] := black; time:=time+1; f[u]:=time;
        pop(S);
    else if color[v]==black pop(S);
```



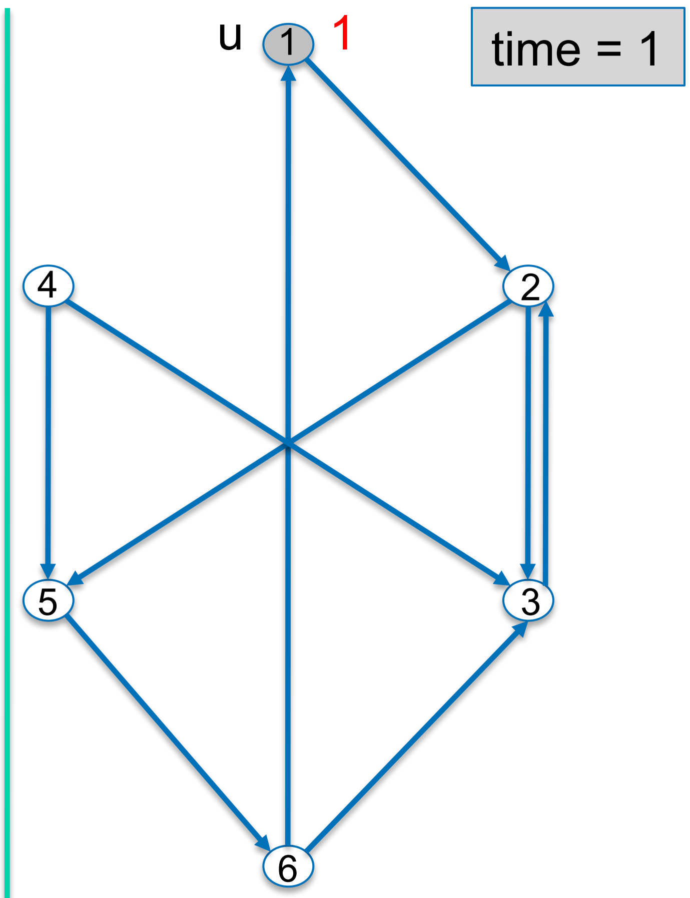
time = 1

S: 1

Basic graph algorithms

```
DFS(G)
for each node u∈V do
            color[u] := white;
time := 0; S:=∅;
for each node u∈V do
            if color[u]==white then DFS-Visit(u);
```

```
DFS-Visit(u)
push(S,u);
while not empty(S) do
      u:=top(S);
      if color[u]==white then
            color[u] := gray; time:=time+1; d[u]:=time;
            for each node v∈adj[u] do
                  if color[v]==white push(S,v);
      else if color[u]==gray then
            color[u] := black; time:=time+1; f[u]:=time;
            pop(S);
      else if color[v]==black pop(S);
```



S: 2, 1

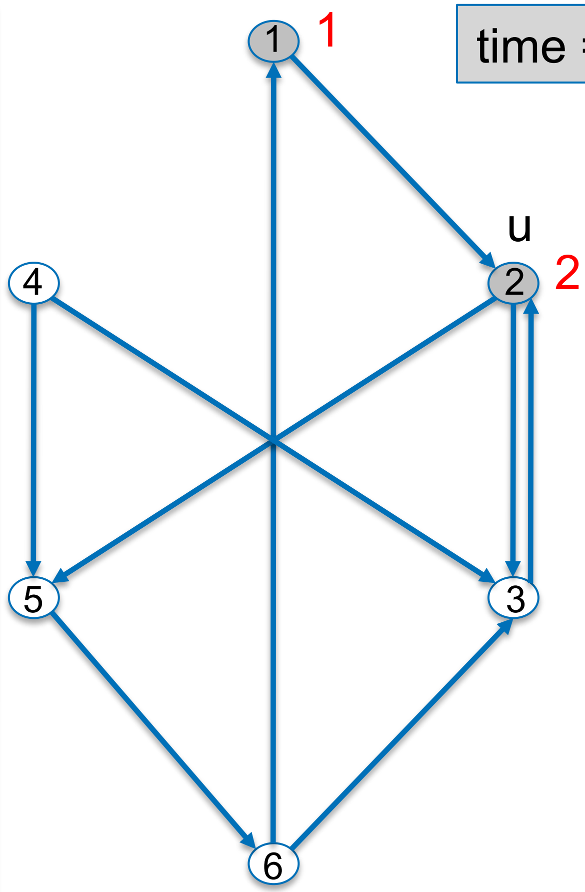
Basic graph algorithms

TECHNISCHE UNIVERSITÄT DARMSTADT

```
DFS(G)
for each node u∈V do
        color[u] := white;
time := 0; S:=∅;
for each node u∈V do
        if color[u]==white then DFS-Visit(u);
```

```
DFS-Visit(u)
push(S,u);
while not empty(S) do
    u:=top(S);
    if color[u]==white then
        color[u] := gray; time:=time+1; d[u]:=time;
        for each node v∈adj[u] do
            if color[v]==white push(S,v);
    else if color[u]==gray then
        color[u] := black; time:=time+1; f[u]:=time;
        pop(S);
    else if color[v]==black pop(S);
```

time = 2



S: 2, 1

Basic graph algorithms

TECHNISCHE UNIVERSITÄT DARMSTADT

DFS(G)

1. for each node u∈V **do**

2. color[u] := white;

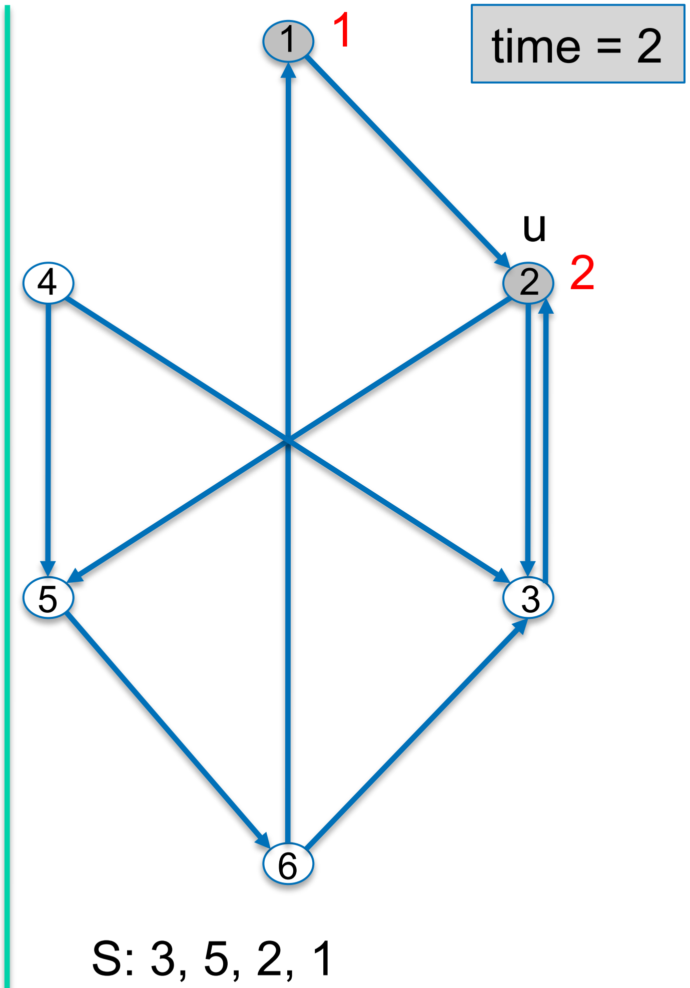
3. time := 0; S:=∅;

4. for each node u∈V **do**

5. if color[u]==white then DFS-Visit(u);

DFS-Visit(u)

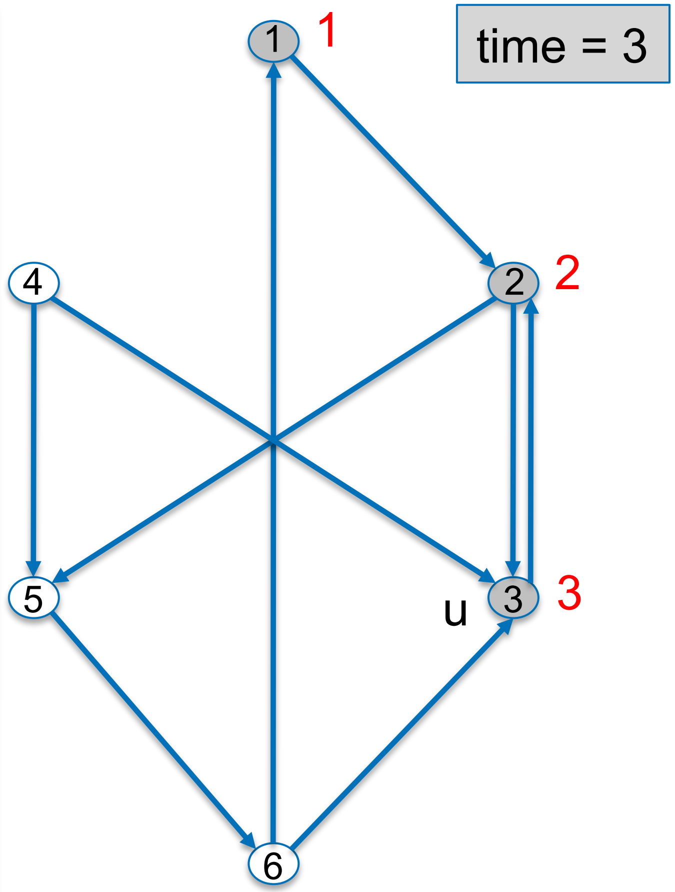
1. push(S,u);
2. while not empty(S) do
3. u:=top(S);
4. if color[u]==white then
5. color[u] := gray; time:=time+1; d[u]:=time;
6. **for each** node v∈adj[u] **do**
7. if color[v]==white push(S,v);
8. else if color[u]==gray then
9. color[u] := black; time:=time+1; f[u]:=time;
10. pop(S);
11. else if color[v]==black pop(S);



Basic graph algorithms

```
DFS(G)
for each node u∈V do
        color[u] := white;
time := 0; S:=∅;
for each node u∈V do
        if color[u]==white then DFS-Visit(u);
```

```
DFS-Visit(u)
push(S,u);
while not empty(S) do
    u:=top(S);
    if color[u]==white then
        color[u] := gray; time:=time+1; d[u]:=time;
        for each node v∈adj[u] do
            if color[v]==white push(S,v);
    else if color[u]==gray then
        color[u] := black; time:=time+1; f[u]:=time;
        pop(S);
    else if color[v]==black pop(S);
```



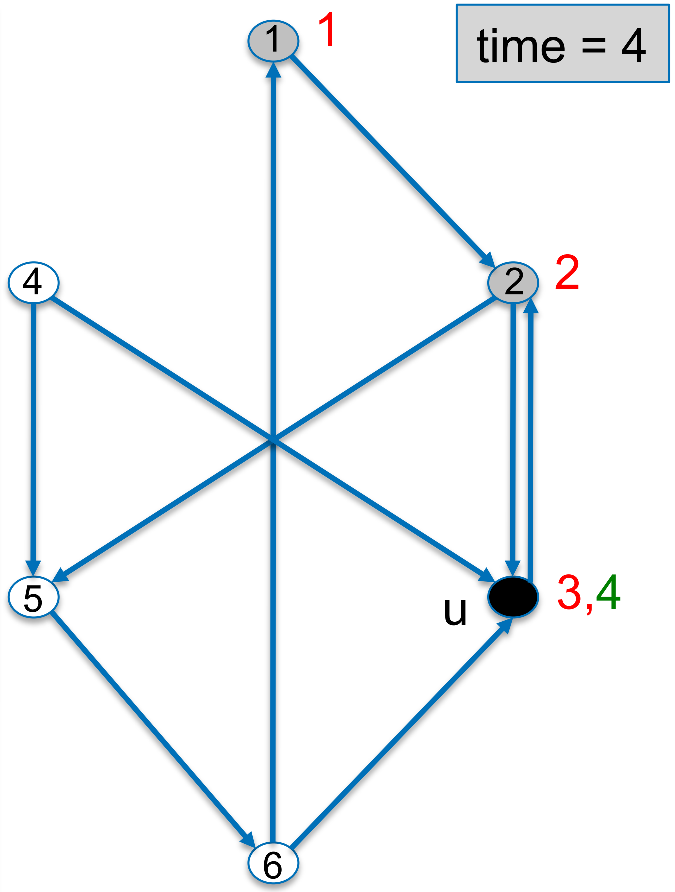
S: 3, 5, 2, 1

Basic graph algorithms


```
DFS(G)
1.  for each node u∈V do
2.          color[u] := white;
3.  time := 0; S:=∅;
4.  for each node u∈V do
5.          if color[u]==white then DFS-Visit(u);
```

```
DFS-Visit(u)
1.  push(S,u);
2.  while not empty(S) do
3.      u:=top(S);
4.      if color[u]==white then
5.          color[u] := gray; time:=time+1; d[u]:=time;
6.          for each node v∈adj[u] do
7.              if color[v]==white push(S,v);
8.      else if color[u]==gray then
9.          color[u] := black; time:=time+1; f[u]:=time;
10.         pop(S);
11.     else if color[v]==black pop(S);
```

u.s.w.



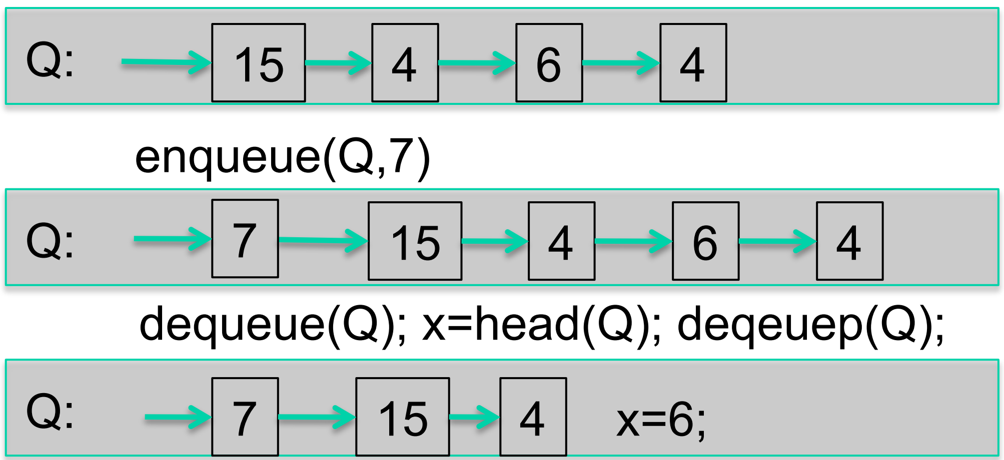
S: ~~3~~, 5, 2, 1

Basic graph algorithms

TECHNISCHE UNIVERSITÄT DARMSTADT

- BFS (Breadth-First-Search, Breitensuche)

  – Let Q be a queue. Then there are 4 different operations with runtime O(1):
    - empty() Is the Queue empty?
    - enqueue(S,x) Add the element x to Q
    - dequeue(S) Remove the earliest added element.
    - x=head(S) Read the content of the earliest added element



Basic graph algorithms

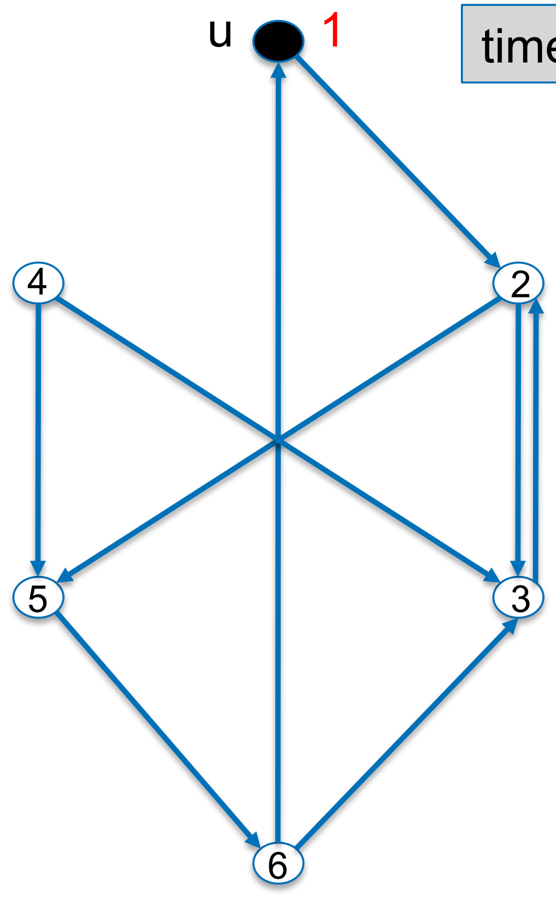
TECHNISCHE UNIVERSITÄT DARMSTADT

BFS(G)

1. for each node u∈V **do**
2. color[u] := white ; dist[u] := ∞;
3. time := 0; Q:=∅;
4. choose an arbitrary node u∈V **and do**
5. BFS-Visit(u);

BFS-Visit(u)

1. enqueue(Q,u);
2. while not empty(Q) **do**
3. u:=head(Q);
4. **if** color[u] ≠ black then
5. color[u] := black; time:=time+1; d[u]:=time;
6. for each node v∈adj[u] **do**
7. **if** color[v]==white enqueue(Q,v),color[v]:=gray
8. else if color[u]==black then
9. time:=time+1; f[u]:=time;
10. dequeue(Q);



time = 1

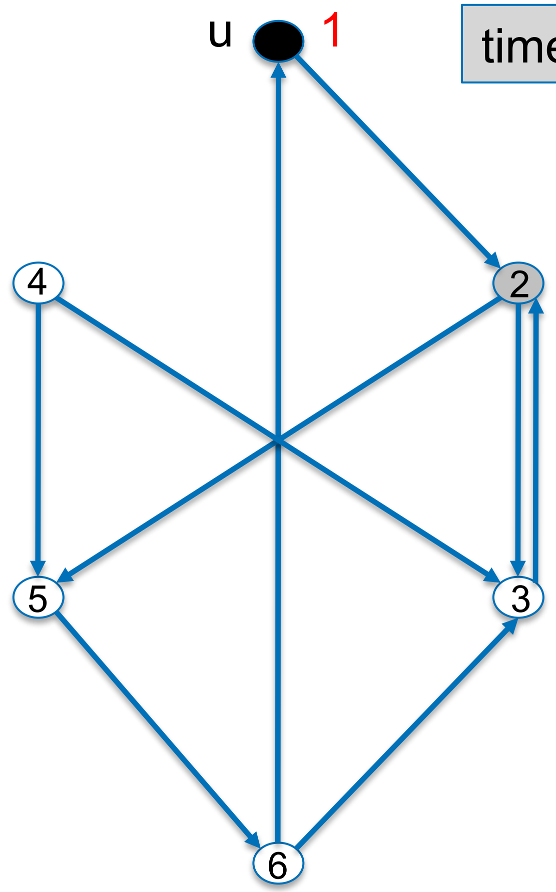
Q: 1

Basic graph algorithms

TECHNISCHE UNIVERSITÄT DARMSTADT

```
BFS(G)
for each node u∈V do
        color[u] := white ; dist[u] := ∞;
time := 0; Q:=∅;
choose an arbitrary node u∈V and do
        BFS-Visit(u);
```

```
BFS-Visit(u)
enqueue(Q,u);
while not empty(Q) do
   u:=head(Q);
   if color[u] ≠ black then
     color[u] := black; time:=time+1; d[u]:=time;
     for each node v∈adj[u] do
       if color[v]==white enqueue(Q,v),color[v]:=gray
   else if color[u]==black then
     time:=time+1; f[u]:=time;
     dequeue(Q);
```



time = 1

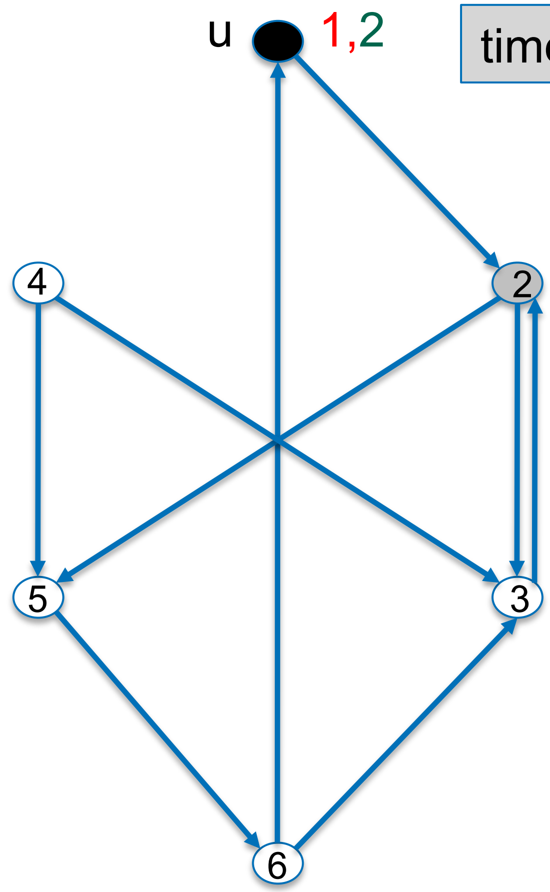
Q: 2, 1

Basic graph algorithms

BFS(G)
1. for each node u∈V **do**
2. color[u] := white ; dist[u] := ∞;
3. time := 0; Q:=∅;
4. choose an arbitrary node u∈V **and do**
5. BFS-Visit(u);

BFS-Visit(u)
1. enqueue(Q,u);
2. while not empty(Q) **do**
3. u:=head(Q);
4. **if** color[u] ≠ black then
5. color[u] := black; time:=time+1; d[u]:=time;
6. for each node v∈adj[u] **do**
7. **if** color[v]==white enqueue(Q,v),color[v]:=gray
8. else if color[u]==black then
9. time:=time+1; f[u]:=time;
10. dequeue(Q);

time = 2



Q: 2, ~~1~~

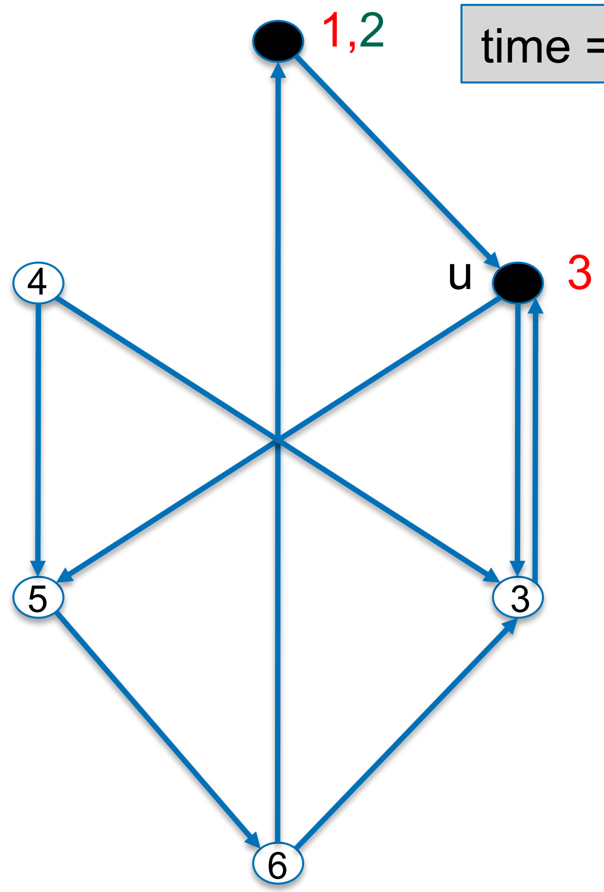
Basic graph algorithms

TECHNISCHE UNIVERSITÄT DARMSTADT

```
BFS(G)
for each node u∈V do
        color[u] := white ; dist[u] := ∞;
time := 0; Q:=∅;
choose an arbitrary node u∈V and do
        BFS-Visit(u);
```

```
BFS-Visit(u)
enqueue(Q,u);
while not empty(Q) do
   u:=head(Q);
   if color[u] ≠ black then
     color[u] := black; time:=time+1; d[u]:=time;
     for each node v∈adj[u] do
       if color[v]==white enqueue(Q,v),color[v]:=gray
   else if color[u]==black then
     time:=time+1; f[u]:=time;
     dequeue(Q);
```

time = 3



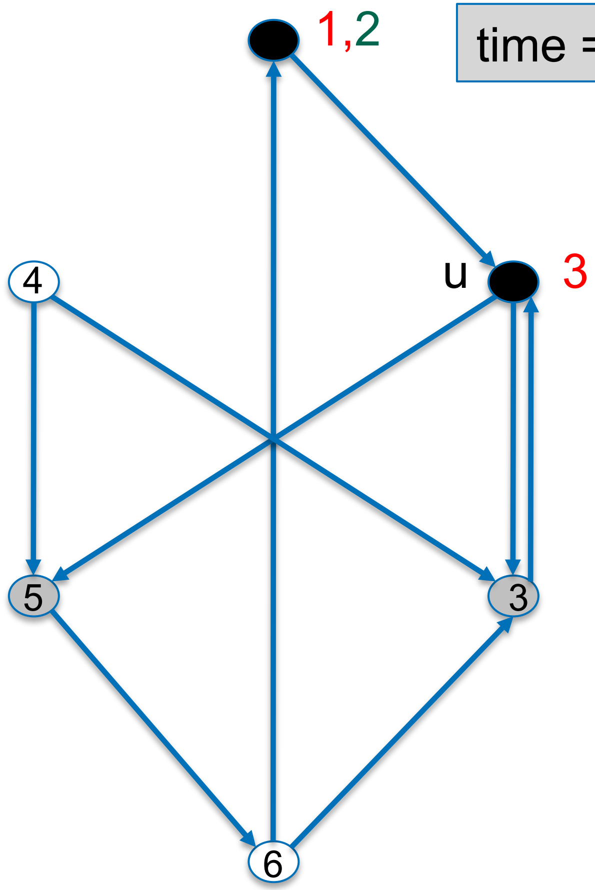
Q: 2

Basic graph algorithms

```
BFS(G)
for each node u∈V do
        color[u] := white ; dist[u] := ∞;
time := 0; Q:=∅;
choose an arbitrary node u∈V and do
        BFS-Visit(u);
```

```
BFS-Visit(u)
enqueue(Q,u);
while not empty(Q) do
   u:=head(Q);
   if color[u] ≠ black then
     color[u] := black; time:=time+1; d[u]:=time;
     for each node v∈adj[u] do
       if color[v]==white enqueue(Q,v),color[v]:=gray
   else if color[u]==black then
     time:=time+1; f[u]:=time;
     dequeue(Q);
```

time = 3



Q: 3,5,2

Basic graph algorithms

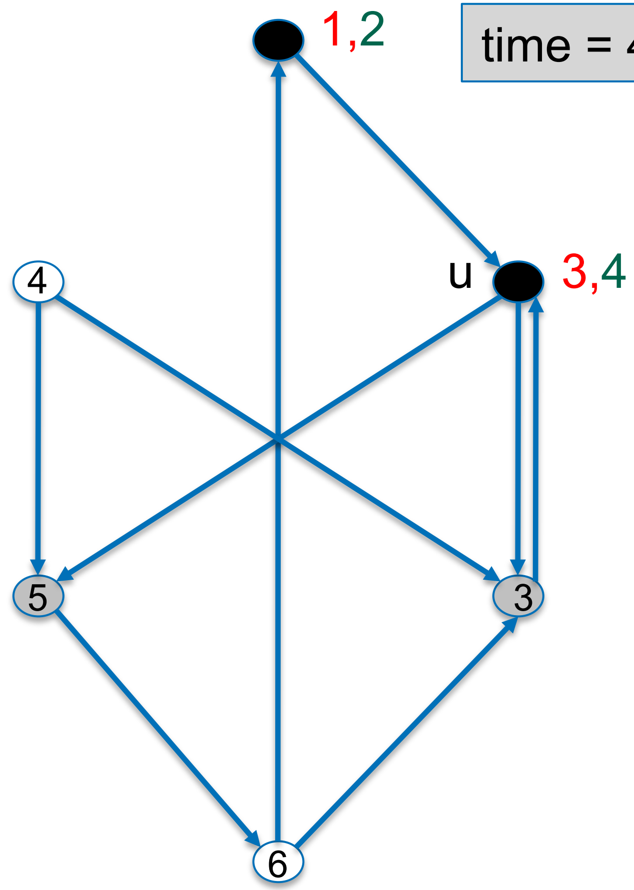
BFS(G)

1. **for each** node u∈V **do**
2. color[u] := white ; dist[u] := ∞;
3. time := 0; Q:=∅;
4. **choose an arbitrary** node u∈V **and do**
5. BFS-Visit(u);

BFS-Visit(u)

1. enqueue(Q,u);
2. **while** not empty(Q) **do**
3. u:=head(Q);
4. **if** color[u] ≠ black then
5. color[u] := black; time:=time+1; d[u]:=time;
6. **for each** node v∈adj[u] **do**
7. **if** color[v]==white enqueue(Q,v),color[v]:=gray
8. else if color[u]==black then
9. time:=time+1; f[u]:=time;
10. dequeue(Q);

time = 4



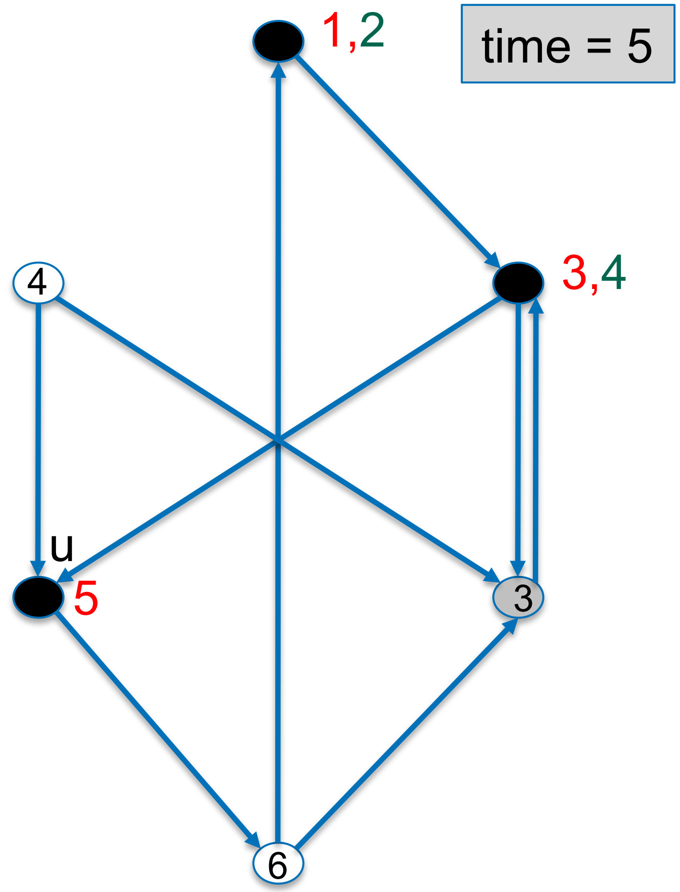
Q: 3,5,~~2~~

Basic graph algorithms

TECHNISCHE UNIVERSITÄT DARMSTADT

```
BFS(G)
for each node u∈V do
        color[u] := white ; dist[u] := ∞;
time := 0; Q:=∅;
choose an arbitrary node u∈V and do
        BFS-Visit(u);
```

```
BFS-Visit(u)
enqueue(Q,u);
while not empty(Q) do
   u:=head(Q);
   if color[u] ≠ black then
     color[u] := black; time:=time+1; d[u]:=time;
     for each node v∈adj[u] do
       if color[v]==white enqueue(Q,v),color[v]:=gray
   else if color[u]==black then
     time:=time+1; f[u]:=time;
     dequeue(Q);
```



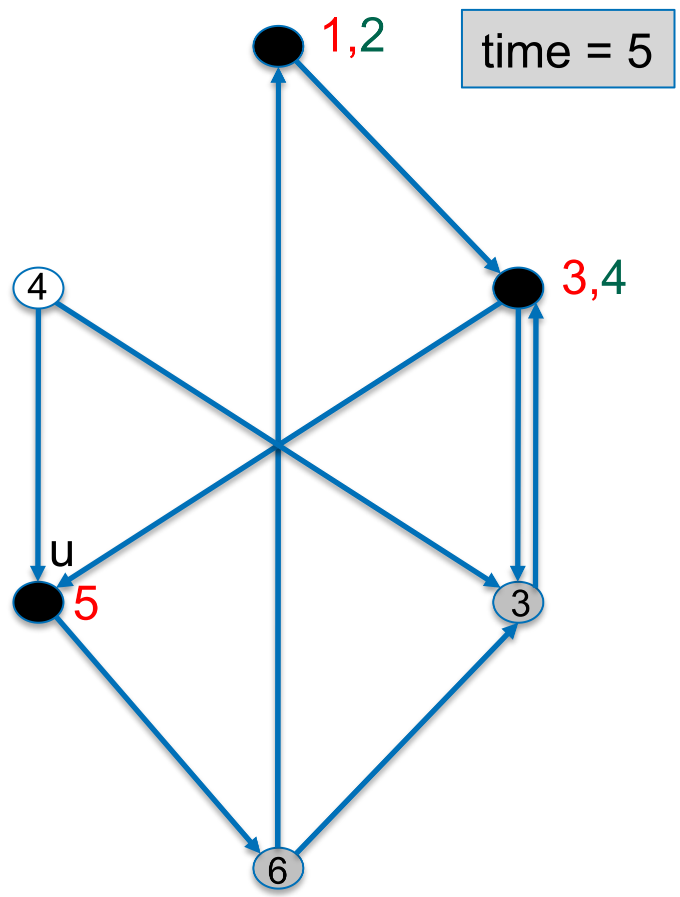
Q: 3,5

Basic graph algorithms

TECHNISCHE UNIVERSITÄT DARMSTADT

```
BFS(G)
for each node u∈V do
        color[u] := white ; dist[u] := ∞;
time := 0; Q:=∅;
choose an arbitrary node u∈V and do
        BFS-Visit(u);
```

```
BFS-Visit(u)
enqueue(Q,u);
while not empty(Q) do
   u:=head(Q);
   if color[u] ≠ black then
     color[u] := black; time:=time+1; d[u]:=time;
     for each node v∈adj[u] do
       if color[v]==white enqueue(Q,v),color[v]:=gray
   else if color[u]==black then
     time:=time+1; f[u]:=time;
     dequeue(Q);
```



Q: 6,3,5

Basic graph algorithms

TECHNISCHE UNIVERSITÄT DARMSTADT

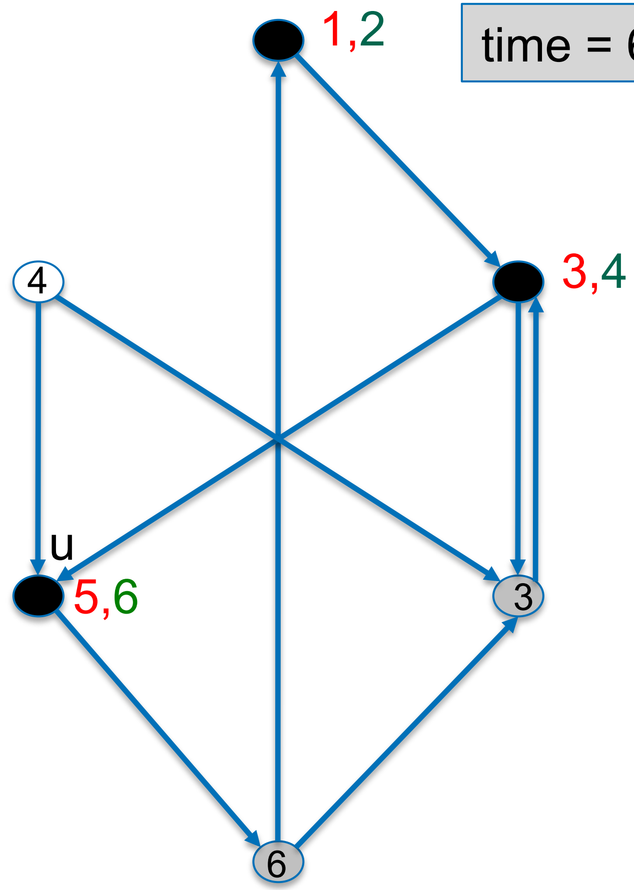
BFS(G)

1. **for each** node u∈V **do**
2. color[u] := white ; dist[u] := ∞;
3. time := 0; Q:=∅;
4. **choose an arbitrary** node u∈V **and do**
5. BFS-Visit(u);

BFS-Visit(u)

1. enqueue(Q,u);
2. **while** not empty(Q) **do**
3. u:=head(Q);
4. **if** color[u] ≠ black then
5. color[u] := black; time:=time+1; d[u]:=time;
6. **for each** node v∈adj[u] **do**
7. **if** color[v]==white enqueue(Q,v),color[v]:=gray
8. else if color[u]==black then
9. time:=time+1; f[u]:=time;
10. dequeue(Q);

time = 6



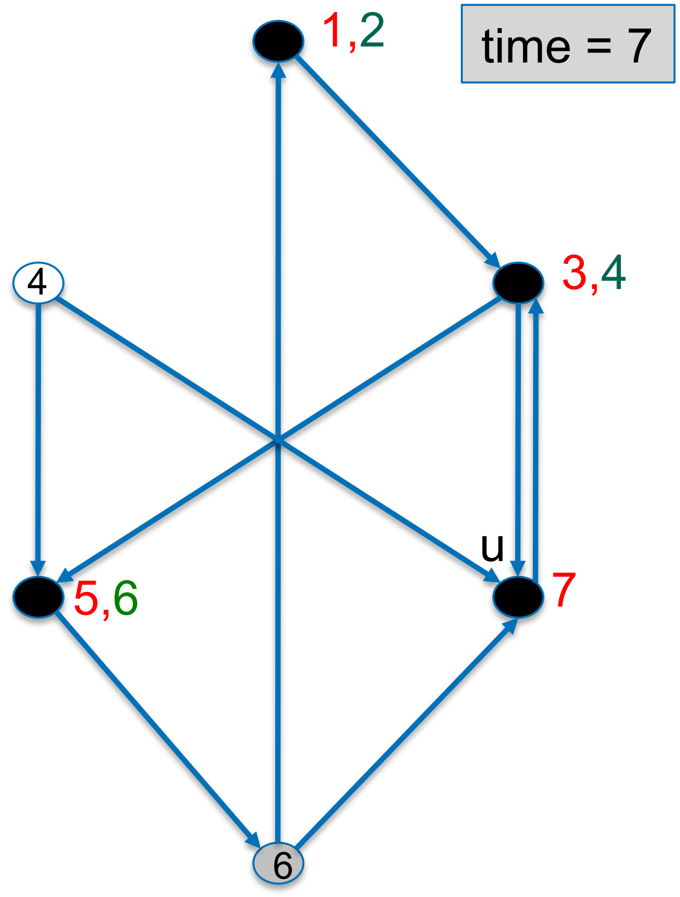
Q: 6,3,~~5~~

Basic graph algorithms

TECHNISCHE UNIVERSITÄT DARMSTADT

```
BFS(G)
for each node u∈V do
        color[u] := white ; dist[u] := ∞;
time := 0; Q:=∅;
choose an arbitrary node u∈V and do
        BFS-Visit(u);
```

```
BFS-Visit(u)
enqueue(Q,u);
while not empty(Q) do
   u:=head(Q);
   if color[u] ≠ black then
     color[u] := black; time:=time+1; d[u]:=time;
     for each node v∈adj[u] do
       if color[v]==white enqueue(Q,v),color[v]:=gray
   else if color[u]==black then
      time:=time+1; f[u]:=time;
      dequeue(Q);
```



Q: 6,3

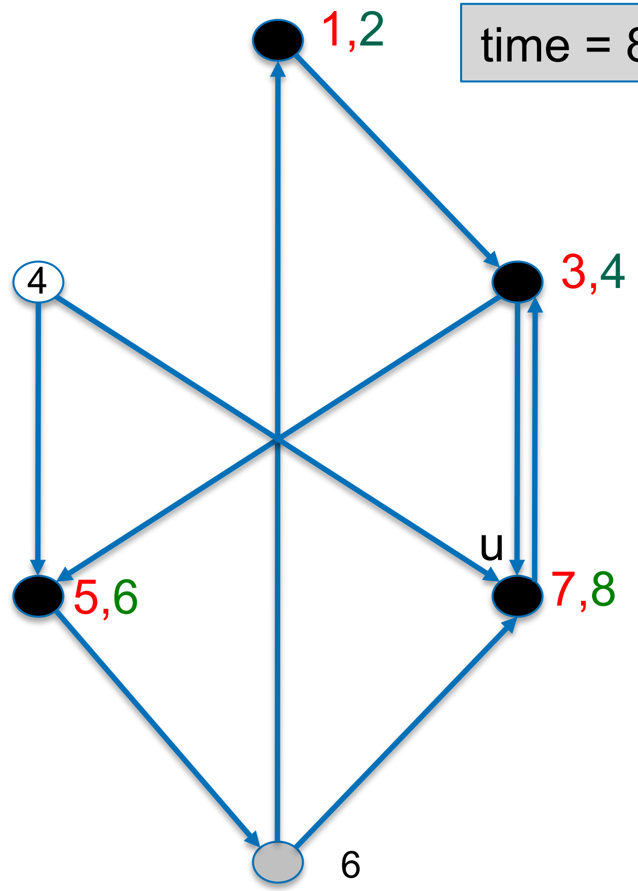
Basic graph algorithms

TECHNISCHE UNIVERSITÄT DARMSTADT

```
BFS(G)
for each node u∈V do
        color[u] := white ; dist[u] := ∞;
time := 0; Q:=∅;
choose an arbitrary node u∈V and do
        BFS-Visit(u);
```

```
BFS-Visit(u)
enqueue(Q,u);
while not empty(Q) do
  u:=head(Q);
  if color[u] ≠ black then
    color[u] := black; time:=time+1; d[u]:=time;
    for each node v∈adj[u] do
      if color[v]==white enqueue(Q,v),color[v]:=gray
  else if color[u]==black then
    time:=time+1; f[u]:=time;
    dequeue(Q);
```

time = 8



Q: 6,3

Basic graph algorithms

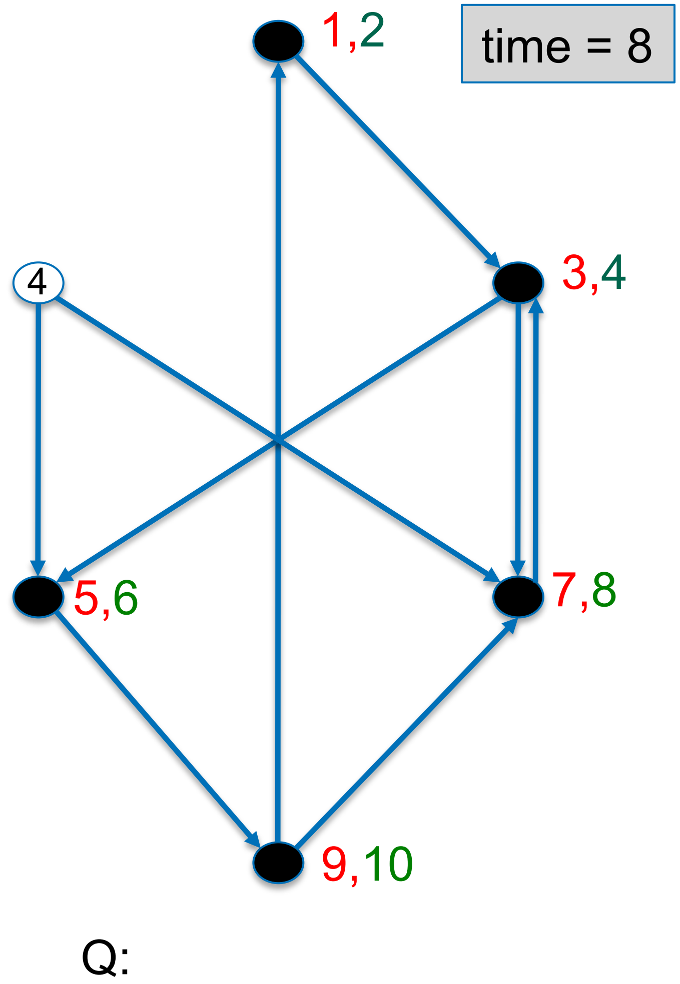
TECHNISCHE UNIVERSITÄT DARMSTADT

BFS(G)

1. for each node u∈V **do**
2. color[u] := white; dist[u] := ∞;
3. time := 0; Q:=∅;
4. choose an arbitrary node u∈V **and do**
5. BFS-Visit(u);

BFS-Visit(u)

1. enqueue(Q,u);
2. while not empty(Q) **do**
3. u:=head(Q);
4. **if** color[u] ≠ black then
5. color[u] := black; time:=time+1; d[u]:=time;
6. for each node v∈adj[u] **do**
7. **if** color[v]==white enqueue(Q,v),color[v]:=gray
8. else if color[u]==black then
9. time:=time+1; f[u]:=time;
10. dequeue(Q);



Basic graph algorithms

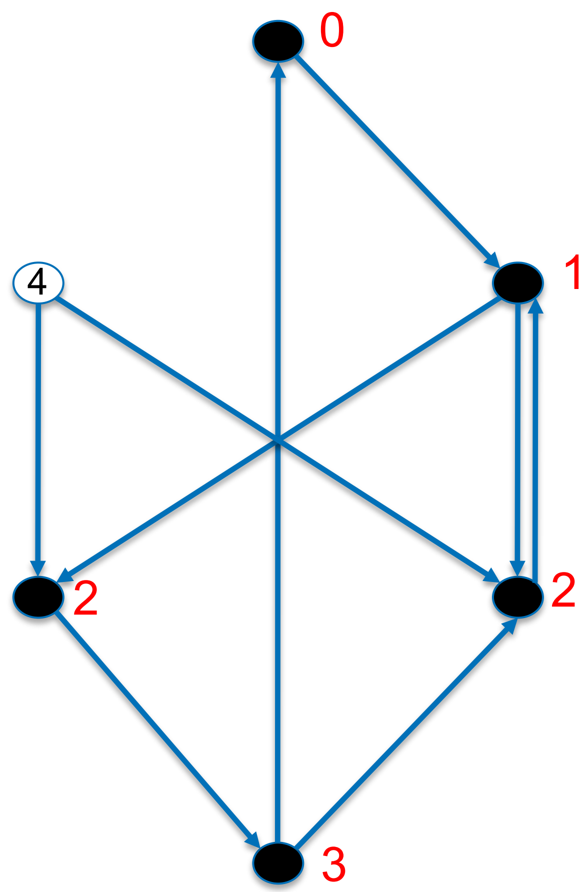
TECHNISCHE UNIVERSITÄT DARMSTADT

BFS(G)

1. for each node u∈V **do**
2. color[u] := white; dist[u] := ∞;
3. Q:=∅;
4. choose an arbitrary node u∈V **and do**
5. BFS-Visit(u);

BFS-Visit(u), *variant with dist, no timestamp*

1. enqueue(Q,u); dist(u):=0;
2. while not empty(Q) do
3. u:=head(Q);
4. for each node v∈adj[u] **do**
5. if color[v]==white
6. color[v]:=gray;
7. enqueue(Q,v);
8. dist(v):=dist(u)+1;
9. π[v]:=u;
10. color[u] := black;
11. dequeue(Q);



Q:

Basic graph algorithms

TECHNISCHE UNIVERSITÄT DARMSTADT

Breadth First Search, some properties

- BFS builds the basis of many algorithms on graphs
- E.g., an aim may be to find all nodes which can be reached from a specific source node s, in a given graph G. A node $v \in V$ is reachable from another node s, if there is a path from s to v.
- BFS computes the distance $\delta(s,v)$ of each node v from the start node s. The distance is defined as the minimal number of edges of all paths from s to v.
- The BFS examines all nodes with distance $<k$ before nodes with distance k. Therefore the name BFS.
- Runtime: $O(|V| + |E|)$

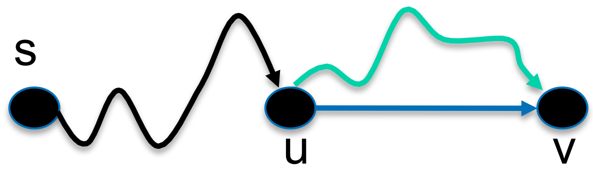
Basic graph algorithms

Breadth first Search, properties

- Let δ(s,v) be the real shortest path-length (i.e. the minmal number of edges that must be traversed in order to go from s to v) from s to v. If there is no path from s to v, let δ(s,v) = ∞.
- Claim: BFS computes the distance of s to v for all v∈V.

- Proof:
  - Lemma bfs1: Let G=(V,E) be a directed or undirected graph and let s∈V be an arbitrary node. Then it will be valid for each edge (u,v)∈E:

$\delta(s,v) \leq \delta(s,u)+1$



Proof: if u is not reachable, it is δ(s,u)=∞. Thus clear.
If u is reachable, v will be as well. The path from s to v cannot be longer than the path from s to u plus 1 edge (i.e. (u,v)). ✔

Basic graph algorithms

- Proof (cont.):
  - Lemma bfs2: Let G=(V,E) be a graph. Let BFS had been runnning on G with start node s. Then we have for each v:

    $$\text{dist}[v| \geq \delta(s,v)$$

```
BFS-Visit(u), Variante
1.  enqueue(Q,u); dist(u):=0;
2.  while not empty(Q) do
3.      u:=head(Q);
4.      for each node v∈adj[u] do
5.          if color[v]==white
6.              color[v]:=gray;
7.              enqueue(Q,v);
8.              dist(v):=dist(u)+1;
9.              π[v]:=u;
10.     color[u] := black;
11.     dequeue(Q);
```

Proof: Induction over the number of enqueue-calls in BFS-Visit

Induction start: Let s be added to Q just right now. Then BFS sets dist[s]=0 and will never change this. All other values are ∞. Additionally is valid δ(s,s)=0. ✔

Induction hypothesis: For each discovered node v∈V, it is dist[v| ≥ δ(s,v)

Inductive step: Let v have been discovered from u. Concerning IH it is: dist[u] ≥ δ(s,u). Because of line 8 (BFS-Visit) and Lemma bfs1, it is implied dist[v] = dist[u]+1 ≥ δ(s,u) + 1 ≥ δ(s,v). Thereafter, dist[v] is never changed. ✔

Basic graph algorithms

BFS-Visit(u), Variante

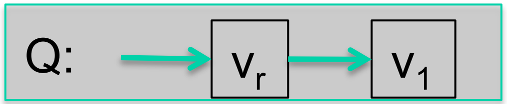
```
enqueue(Q,u); dist(u):=0;
while not empty(Q) do
    u:=head(Q);
    for each node v∈adj[u] do
        if color[v]==white
            color[v]:=gray;
            enqueue(Q,v);
            dist(v):=dist(u)+1;
            π[v]:=u;
    color[u] := black;
    dequeue(Q);
```

- Proof(cont.):
  - Lemma bfs3: Let G=(V,E) be a graph.
    Let, at some time, Q be : $Q=\langle v_1,v_2,...,v_r\rangle$
    Let v_1 be head(Q). Then, for i=1,2...,r-1:

    $dist[v_r] \leq dist[v_1]+1$ und $dist[v_i] \leq dist[v_{i+1}]$

    Proof: Induction over the number of executions of for-loop (in total)

    Induction start: After the first execution of the loop it is
    $v_1 = u = s$ und $v_r=v$. Therefore $dist[v_1]=0$ and $dist[v_r]=1$. ✔



$dist[v_r] = dist[v_1]+1$

Basic graph algorithms

- Proof (cont.):
  - Lemma bfs3: Let G=(V,E) be a graph. Let, at some time, Q be : Q=<v_1,v_2,...,v_r> Let v_1 be head(Q). Then, for i=1,2...,r-1:

    $dist[v_r] \leq dist[v_1]+1$ und $dist[v_i] \leq dist[v_{i+1}]$

BFS-Visit(u), Variante

```
enqueue(Q,u); dist(u):=0;
while not empty(Q) do
    u:=head(Q);
    for each node v∈adj[u] do
        if color[v]==white
            color[v]:=gray;
            enqueue(Q,v);
            dist(v):=dist(u)+1;
            π[v]:=u;
color[u] := black;
dequeue(Q);
```

Proof: ...

Induction hypothesis: For the first n loop executions it is valid:
$dist[v_{r(n)}] \leq dist[v_{1(n)}]+1$ and $dist[v_{i(n)}] \leq dist[v_{i(n)+1}]$
after each of the loop executions.

Basic graph algorithms

- Proof(cont.):
  - Lemma bfs3: Let G=(V,E) be a graph. Let, at some time, Q be : $Q=<v_1,v_2,...,v_r>$ Let v_1 be head(Q). Then, for i=1,2...,r-1:

    $dist[v_r] \leq dist[v_1]+1$ und $dist[v_i] \leq dist[v_{i+1}]$

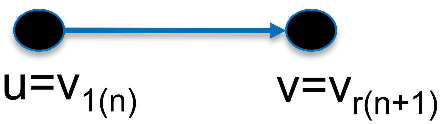
BFS-Visit(u), Variante

```
1.  enqueue(Q,u); dist(u):=0;
2.  while not empty(Q) do
3.      u:=head(Q);
4.      for each node v∈adj[u] do
5.          if color[v]==white
6.              color[v]:=gray;
7.              enqueue(Q,v);
8.              dist(v):=dist(u)+1;
9.              π[v]:=u;
10.     color[u] := black;
11.     dequeue(Q);
```

Proof: ...

Induction step: When $v_{r(n+1)}$ gets enqueued, $v_{r(n+1)}$ is successor of v_1.

Q: → $v_{r(n+1)}$ → $v_{r(n)}=v_{r(n+1)-1}$ … $v_{2(n+1)}$ → $v_{1(n+1)}=v_{1(n)}$



Some dequeue? No -> clear. Yes? -> with IH

Thus: $dist[v_{r(n+1)}] \leq dist[v_{1(n+1)}]+1$. With IH, it is also valid

$dist[v_{r(n)}] \leq dist[v_{1(n)}]+1 \leq dist[v_{1(n+1)}]+1$ (= dist[u] +1 = dist[v]) = $dist[v_{r(n+1)}]$

IH:

After a dequeue: $dist[v_{r(n+1)}] \leq dist[v_1] + 1 \leq d[v_2] + 1$ ✔

Basic graph algorithms

BFS-Visit(u), Variante

```
enqueue(Q,u); dist(u):=0;
while not empty(Q) do
    u:=head(Q);
    for each node v∈adj[u] do
        if color[v]==white
            color[v]:=gray;
            enqueue(Q,v);
            dist(v):=dist(u)+1;
            π[v]:=u;
    color[u] := black;
    dequeue(Q);
```

- Summary:
  - Lemma bfs3: Let G=(V,E) be a graph.
    Let, at some time, Q be : $Q=\langle v_1,v_2,...,v_r\rangle$
    Let v_1 be head(Q). Then, for i=1,2...,r-1:

    $\text{dist}[v_r] \leq \text{dist}[v_1]+1$ und $\text{dist}[v_i] \leq \text{dist}[v_{i+1}]$

  Proof (summary):

  Induction over number of for-loop executions in total

  Induction start: After the first loop execution it is
  $v_1 = u = s$ und $v_r=v$. Thus, $\text{dist}[v_1]=0$ and $\text{dist}[v_r]=1$. ✔

  Induction hypothesis: For the first n loop executions we have:
  $\text{dist}[v_{r(n)}] \leq \text{dist}[v_{1(n)}]+1$ und $\text{dist}[v_{i(n)}] \leq \text{dist}[v_{i(n)+1}]$
  after each loop execution.

  Induction step: When $v_{r(n+1)}$ is enqueued, $v_{r(n+1)}$ is successor of v_1.
  Thus: $\text{dist}[v_{r(n+1)}] = \text{dist}[v_{1(n)}]+1 \leq \text{dist}[v_{1(n+1)}]+1$. Concerning IH, it is also valid:
  $\text{dist}[v_{r(n)}] \leq \text{dist}[v_{1(n)}]+1 \leq \text{dist}[v_{1(n+1)}]+1$ (= dist[u] +1 = dist[v]) $= \text{dist}[v_{r(n+1)}]$

✔

TECHNISCHE UNIVERSITÄT DARMSTADT

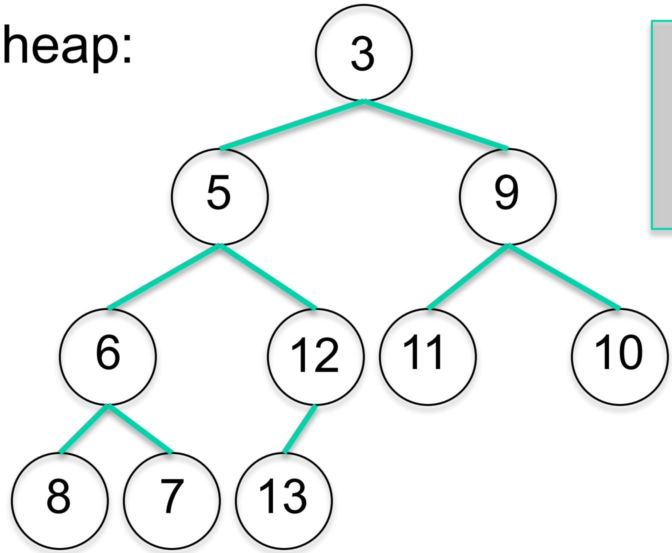
Breadth First Search, Properties

- Let $\delta(s,v)$ be the real shortest path-length (i.e. the minmal number of edges that must be traversed in order to go from s to v) from s to v. If there is no path from s to v, let $\delta(s,v) = \infty$. BFS computes the distance of s to v for all $v \in V$.

- Proof:
  1) if v is not reachable, v is never discovered and $dist[v]=\infty$.
  2) define $V_k := \{v \in V: \delta(s,v)=k\}$. Proof via induction on k.
     - Induction start: k=0: $V_0=\{s\}$ and $dist[s]=0$. ✔
     - Induction hypothesis: for all nodes with $\delta(s,v) < k$ the claim is valid.
     - IS: $k-1 \rightarrow k$: Let $v \in V_k$. With lemma bfs3 it is $dist[v_i] \leq dist[v_{i+1}]$, if v_i was enqueued
       into Q before v_{i+1}. Because of lemma bfs2 it is valid: $dist[v] \geq \delta(s,v)=k$.
       Therefore v must be enqueued into Q after all $u \in V_{k-1}$ have been enqueued in Q.
       Because $\delta(s,v)=k$, there is a path of length k from s to v and thus it exists a node $u \in V_{k-1}$, such that $(u,v) \in E$. Apply IH once more ... ✔

The heap data structure

A (min-)heap:



H: size:10

3	5	9	6	12	11	10	8	7	13
1	2	3	4	5	6	7	8	9	10

Parent(i): return $\lfloor i/2 \rfloor$
Left(i): return 2i
Right(i): return 2i+1

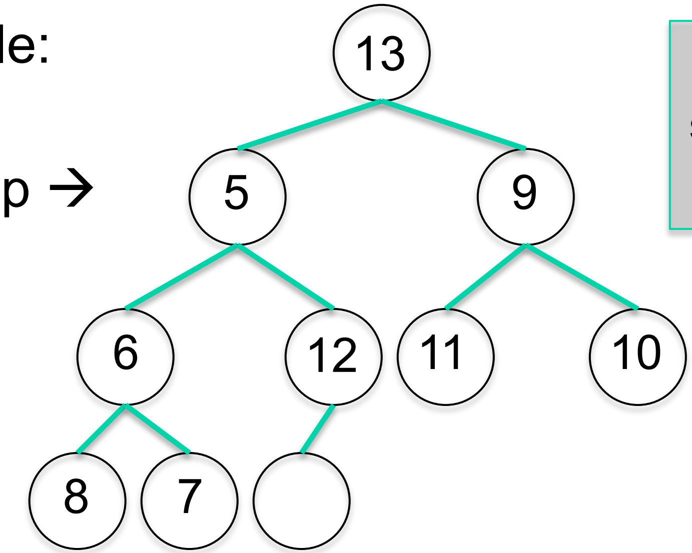
- Binary tree
- Each node corresponds to an element
- Tree becomes filled level by level
- Mostly, heap is stored in an array
- „Heap-property“: values of successors v_1,v_2 of a node v are larger then the value of v itself

The heap data structure

TECHNISCHE UNIVERSITÄT DARMSTADT

Example:

No heap →



H: size:9

13	5	9	6	12	11	10	8	7	
1	2	3	4	5	6	7	8	9	10

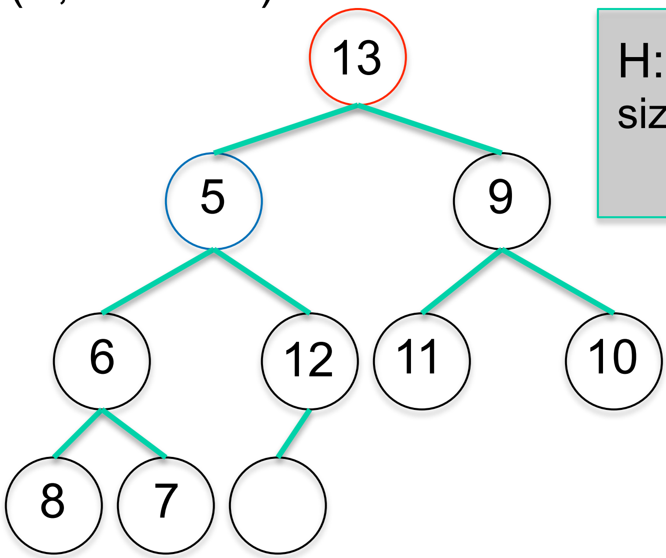
Parent(i): return $\lfloor i/2 \rfloor$
Left(i): return 2i
Right(i): return 2i+1

- Operations are
  - BuildHeap takes a set of elements and builds a heap
  - Insert adds an element
  - ExtractMin takes the smallest element out
  - Heapify reconstructs heap-property on a path from root to leaf
  - DecreaseKey(A,i,newkey) changes an element and reconstructs heap property

The heap data structure

Heapify(A,i) **// Start withi=1**

```
if Left(i) ≤ size and A[ Left(i) ] < A[i] then smallest := Left(i)
else smallest := i
if Right(i) ≤ size and A[ Right(i) ] < A[smallest] then smallest := Right(i)
if smallest ≠ i then
    exchange(A[i], A[smallest])
    Heapify(A, smallest)
```



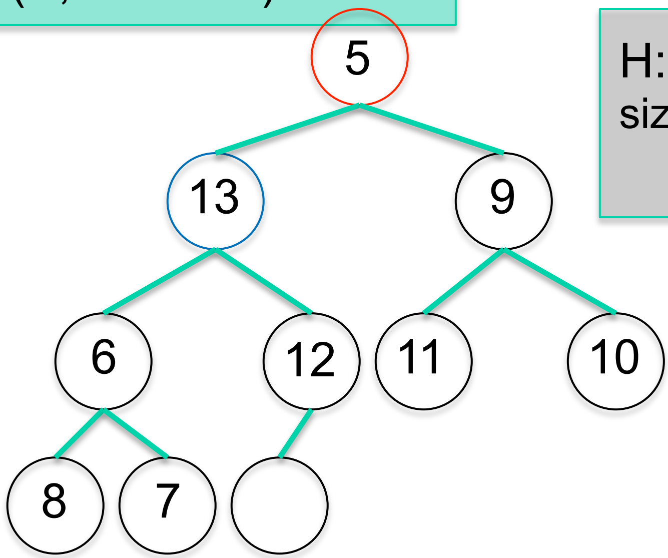
H:
size:9

13	5	9	6	12	11	10	8	7	
1	2	3	4	5	6	7	8	9	10

The heap data structure

Heapify(A,i) **// Start with i=1**

1. **if** Left(i) ≤ size **and** A[Left(i)] < A[i] **then** smallest := Left(i)
2. **else** smallest := i
3. **if** Right(i) ≤ size **and** A[Right(i)] < A[smallest] **then** smallest := Right(i)
4. **if** smallest ≠ i **then**
5. exchange(A[i], A[smallest])
6. Heapify(A, smallest)



H:

size:9

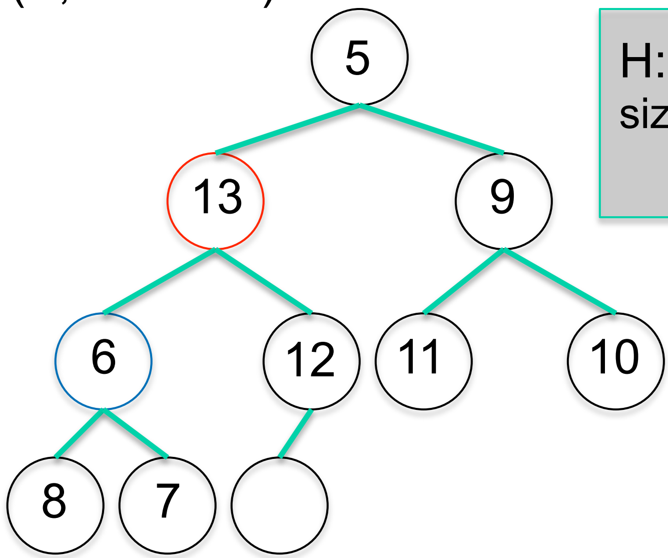
5	13	9	6	12	11	10	8	7	
1	2	3	4	5	6	7	8	9	10

The heap data structure

TECHNISCHE UNIVERSITÄT DARMSTADT

Heapify(A,i) **// with i=2**

1. **if** Left(i) ≤ size **and** A[Left(i)] < A[i] **then** smallest := Left(i)
2. **else** smallest := i
3. **if** Right(i) ≤ size **and** A[Right(i)] < A[smallest] **then** smallest := Right(i)
4. **if** smallest ≠ i **then**
5. exchange(A[i], A[smallest])
6. Heapify(A, smallest)



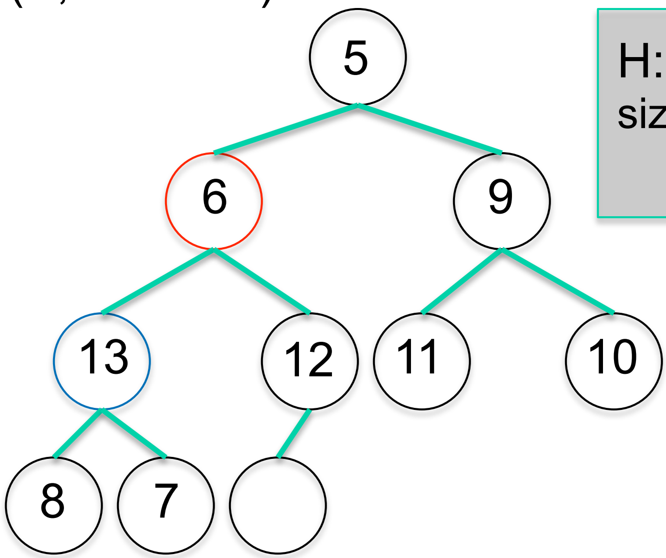
H:
size:9

5	13	9	6	12	11	10	8	7	
1	2	3	4	5	6	7	8	9	10

The heap data structure

Heapify(A,i) **// with i=2**

1. **if** Left(i) ≤ size **and** A[Left(i)] < A[i] **then** smallest := Left(i)
2. **else** smallest := i
3. **if** Right(i) ≤ size **and** A[Right(i)] < A[smallest] **then** smallest := Right(i)
4. **if** smallest ≠ i **then**
5. exchange(A[i], A[smallest])
6. Heapify(A, smallest)



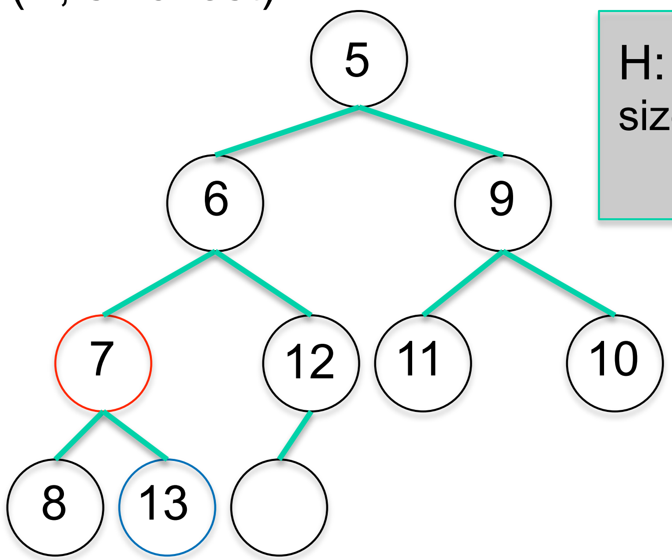
H: size:9

5	6	9	13	12	11	10	8	7	
1	2	3	4	5	6	7	8	9	10

The heap data structure

Heapify(A,i) **// with i=4**

1. **if** Left(i) ≤ size **and** A[Left(i)] < A[i] **then** smallest := Left(i)
2. **else** smallest := i
3. **if** Right(i) ≤ size **and** A[Right(i)] < A[smallest] **then** smallest := Right(i)
4. **if** smallest ≠ i **then**
5. exchange(A[i], A[smallest])
6. Heapify(A, smallest)



H:
size:9

5	6	9	7	12	11	10	8	13	
1	2	3	4	5	6	7	8	9	10

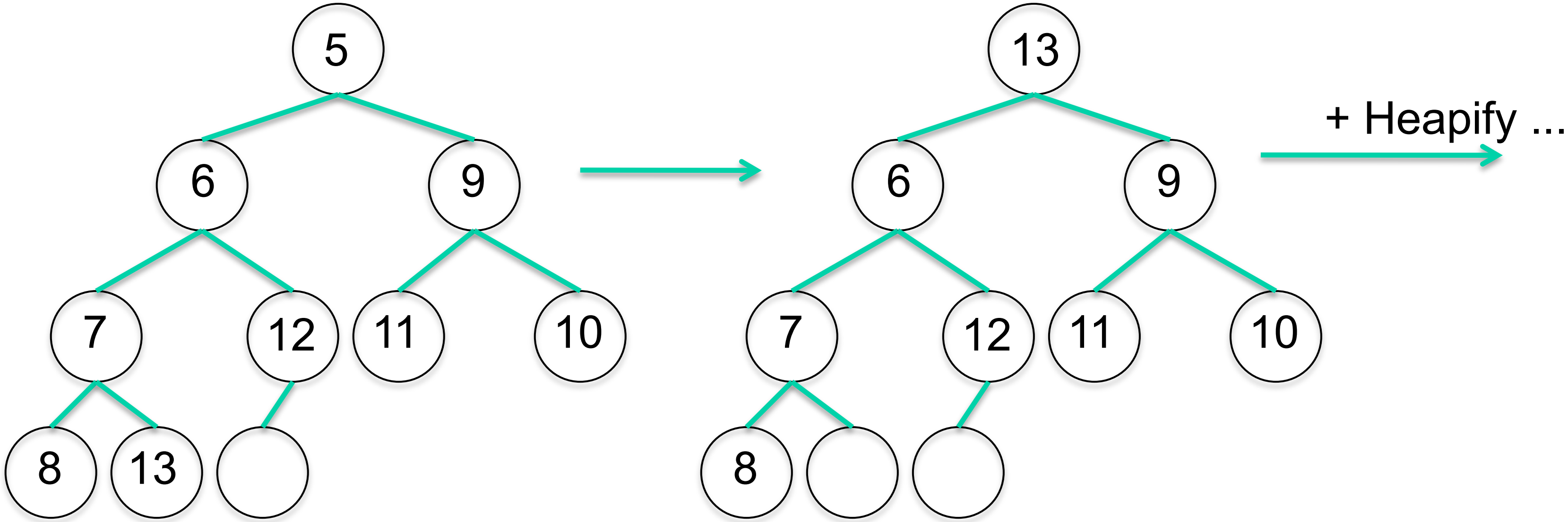
☺

The heap data structure

TECHNISCHE UNIVERSITÄT DARMSTADT

ExtractMin(A)

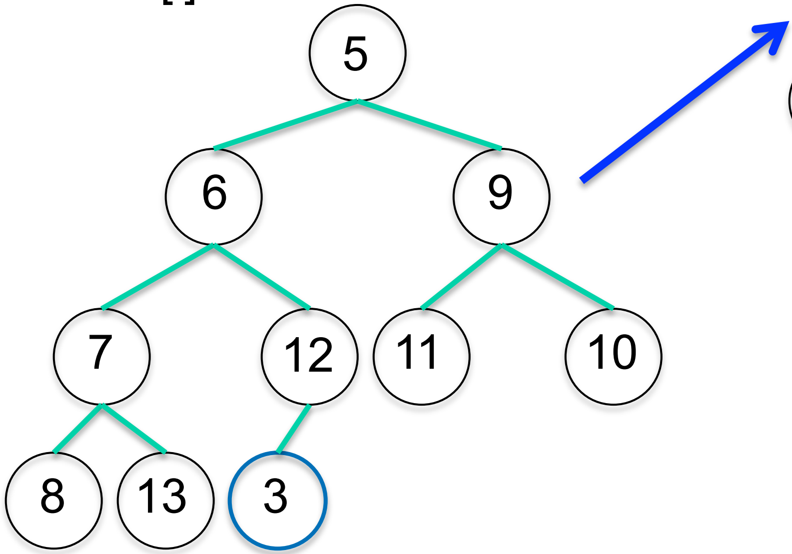
1. **take out the root element. It is the smallest.**
2. **take out the last element and put it into root**
3. **size := size -1**
4. **Heapify(A,1)**

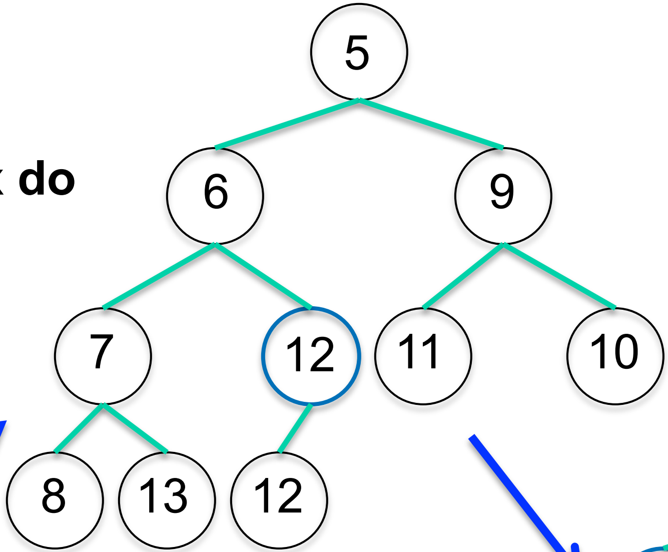


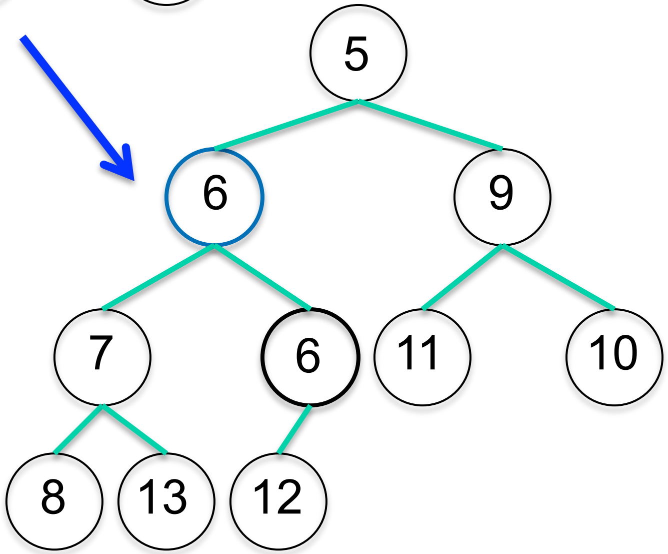
The heap data structure

Insert(A,x)

1. size := size+1
2. i := size
3. **while** i>1 **and** A[Parent(i)] > x **do**
4. A[i] := A[Parent(i)]
5. i := Parent(i)
6. A[i] := x





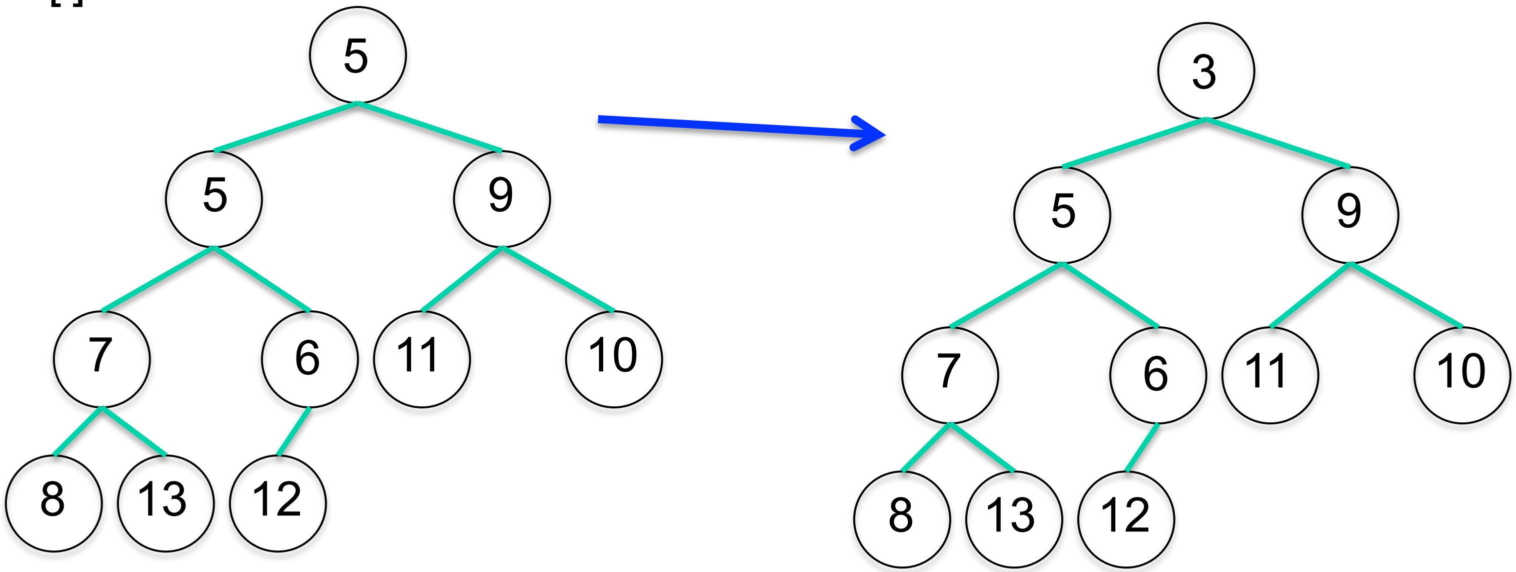


The heap data structure

Insert(A,x)

1. size := size+1
2. i := size
3. **while** i>1 **and** A[Parent(i)] > x **do**
4. A[i] := A[Parent(i)]
5. i := Parent(i)
6. A[i] := x

Correctness: if an element x of node v is copied into a successor of v, this is because the new element is smaller than x. Therefore, x will not destroy the heap property below v.



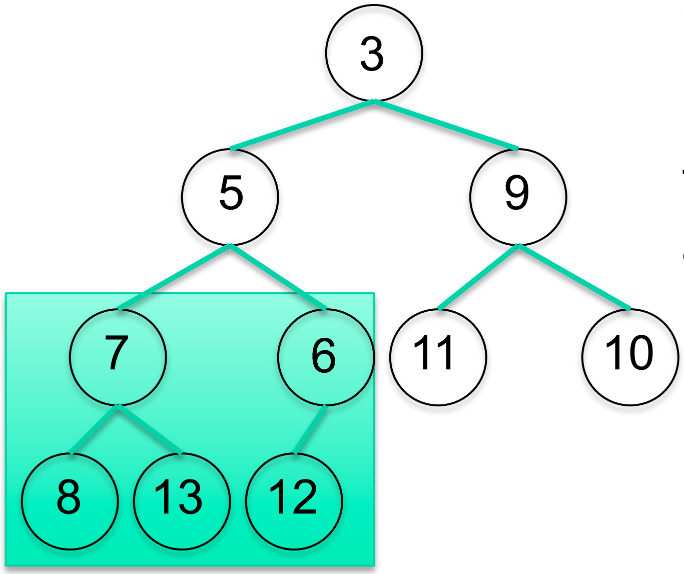
☺

The heap data structure

BuildHeap(A) // all n elements are in an unsorted array (tree)

1. size := number of elements n
2. for i := ⌊size/2⌋ downto 1 do
3.   Heapify(A,i)

Simple bound to runtime: O(n log n). More exactly: O(n) (without proof)



Correctness: Heapify makes heaps of the trees of the last two levels.
Let a new father node be created for two subheaps.
Two cases may occur:

a) The value of the new node is smaller than the son-values. Then: heap property is valid. Or:
b) Value of father is larger than one of son-values. Then: heap property is invalid locally, norwhere else. -> heapify repairs heap.

The heap data structure

TECHNISCHE UNIVERSITÄT DARMSTADT

DecreaseKey(A,i,newkey)

1. A[i] := newkey
2. **while** i>1 **and** A[Parent(i)] > A[i] **do**
3. Exchange(A[i], A[Parent(i))
4. i := Parent(i)

Correctness analogously to Insert(A,x).

☺

Shortest paths revisited

Variant of Dijkstras Algorithmus

```
Initialize(G,s)  // für alle Knoten v≠s: π[v]:=nil; dist[v]:=∞; dist[s]:=0;π[s]:=nil;
S := ∅;
A := V;
BuildHeap(A) with values dist[a] for all a∈A
while   A ≠ ∅  do
   u := ExtractMin(A)
   S := S ∪ {u};
   for each node v ∈ Adj[u] do
      if dist[v] > dist[u] + f(u,v) then
         dist[v] := dist[u] + f(u,v);
         DecreaseKey(A,v,dist[v])
         π[v] := u;
```

Runtime: $O((|E|+|V|) \cdot \log(|V|)$

With assumption: $|E| = c\,|V|$: $O((|V|) \cdot \log(|V|)$

Remark: even better: so called Fibonacci-Heaps: $O(|E| + |V| \cdot \log(|V|)$