Script generated by TTT

Title: groh: profile1 (01.07.2016)

Date: Fri Jul 01 09:14:29 CEST 2016

Duration: 72:22 min

Pages: 42

Listen / Array – Laufzeit-Vergleich

Operation	DoppeltVkttListe	EinfVkttListe	Dynamisches Array	
[.]	O(n)	O(n)	O(1)	
get(.)	O(n)	O(n)	O(1)	
set(.,.)	O(n)	O(n)	O(1)	
size()	O(1)*	O(1)*	O(1)	*nicht wenn splice mit mehreren involvierten Listen zur Verfügung steht, dann O(n) (siehe Mehlhorn Sanders 3.1.1. ganz am Ende)
first()	O(1)	O(1)	O(1)	
last()	O(1)	O(1)	O(1)	
insertAfter(.,.)	O(1)	O(1)	O(n)	
insertBefore(.,.)	O(1)	O(n)	O(n)	
remove(.)	O(1)	O(1)*	O(n)	*nur removeAfter
pushBack(.)	O(1)	O(1)	O(1)*	* nur amortisiert
pushFront(.)	O(1)	O(1)	O(n)	* nur amortisiert
popBack(.)	O(1)	O(n)	O(1)*	* nur amortisiert
popFront(.)	O(1)	O(1)	O(n)	* nur amortisiert
concat(.,.)	O(1)	O(1)	O(n)	
splice(.,.,.)	O(1)	O(1)	O(n)	
findNext(.,.)	O(n)	O(n)	O(n)*	* Cache-effizient

60

Listen / Array – Laufzeit-Vergleich

Operation	DoppeltVkttListe	EinfVkttListe	Dynamisches Array	
[.]	O(n)	O(n)	O(1)	
get(.)	O(n)	O(n)	O(1)	
set(.,.)	O(n)	O(n)	O(1)	
size()	O(1)*	O(1)*	O(1)	*nicht wenn splice mit mehreren involvierten Listen zur Verfügung steht, dann O(n) (siehe Mehlhorn Sanders 3.1.1. ganz am Ende)
first()	O(1)	O(1)	O(1)	
last()	O(1)	O(1)	O(1)	
insertAfter(.,.)	O(1)	O(1)	O(n)	
insertBefore(.,.)	O(1)	O(n)	O(n)	
remove(.)	O(1)	O(1)*	O(n)	*nur removeAfter
pushBack(.)	O(1)	O(1)	O(1)*	* nur amortisiert
pushFront(.)	O(1)	O(1)	O(n)	* nur amortisiert
popBack(.)	O(1)	O(n)	O(1)*	* nur amortisiert
popFront(.)	O(1)	O(1)	O(n)	* nur amortisiert
concat(.,.)	O(1)	O(1)	O(n)	
splice(.,.,.)	O(1)	O(1)	O(n)	
findNext(.,.)	O(n)	O(n)	O(n)*	* Cache-effizient

60

Listen / Array – Laufzeit-Vergleich

Operation	DoppeltVkttListe	EinfVkttListe	Dynamisches Array	
[.]	O(n)	O(n)	O(1)	
get(.)	O(n)	O(n)	O(1)	
set(.,.)	O(n)	O(n)	O(1)	
size()	O(1)*	O(1)*	O(1)	*nicht wenn splice mit mehreren involvierten Listen zur Verfügung steht, dann O(n) (siehe Mehlhorn Sanders 3.1.1. ganz am Ende)
first()	O(1)	O(1)	O(1)	
last()	O(1)	O(1)	O(1)	
insertAfter(.,.)	O(1)	O(1)	O(n)	
insertBefore(.,.)	O(1)	O(n)	O(n)	
remove(.)	O(1)	O(1)*	O(n)	*nur removeAfter
pushBack(.)	O(1)	O(1)	O(1)*	* nur amortisiert
pushFront(.)	O(1)	O(1)	O(n)	* nur amortisiert
popBack(.)	O(1)	O(n)	O(1)*	* nur amortisiert
popFront(.)	O(1)	O(1)	O(n)	* nur amortisiert
concat(.,.)	O(1)	O(1)	O(n)	
splice(.,.,.)	O(1)	O(1)	O(n)	
findNext(.,.)	O(n)	O(n)	O(n)*	* Cache-effizient

60

Listen / Array – Laufzeit-Vergleich

Operation	DoppeltVkttListe	EinfVkttListe	Dynamisches Array	
[.]	O(n)	O(n)	O(1)	
get(.)	O(n)	O(n)	O(1)	
set(.,.)	O(n)	O(n)	O(1)	
size()	O(1)*	O(1)*	O(1)	*nicht wenn splice mit mehreren involvierten Listen zur Verfügung steht, dann O(n) (siehe Mehlhorn Sanders 3.1.1. ganz am Ende)
first()	O(1)	O(1)	O(1)	
last()	O(1)	O(1)	O(1)	
insertAfter(.,.)	O(1)	O(1)	O(n)	
insertBefore(.,.)	O(1)	O(n)	O(n)	
remove(.)	O(1)	O(1)*	O(n)	*nur removeAfter
pushBack(.)	O(1)	O(1)	O(1)*	* nur amortisiert
pushFront(.)	O(1)	O(1)	O(n)	* nur amortisiert
popBack(.)	O(1)	O(n)	O(1)*	* nur amortisiert
popFront(.)	O(1)	O(1)	O(n)	* nur amortisiert
concat(.,.)	O(1)	O(1)	O(n)	
splice(.,.,.)	O(1)	O(1)	O(n)	
findNext(.,.)	O(n)	O(n)	O(n)*	* Cache-effizient

Listen / Array – Laufzeit-Vergleich

Operation	DoppeltVkttListe	EinfVkttListe	Dynamisches Array	
[.]	O(n)	O(n)	O(1)	
get(.)	O(n)	O(n)	O(1)	
set(.,.)	O(n)	O(n)	O(1)	
size()	O(1)*	O(1)*	O(1)	*nicht wenn splice mit mehreren involvierten Listen zur Verfügung steht, dann O(n) (siehe Mehlhorn Sanders 3.1.1. ganz am Ende)
first()	O(1)	O(1)	O(1)	
last()	O(1)	O(1)	O(1)	
insertAfter(.,.)	O(1)	O(1)	O(n)	
insertBefore(.,.)	O(1)	O(n)	O(n)	
remove(.)	O(1)	O(1)*	O(n)	*nur removeAfter
pushBack(.)	O(1)	O(1)	O(1)*	* nur amortisiert
pushFront(.)	O(1)	O(1)	O(n)	* nur amortisiert
popBack(.)	O(1)	O(n)	O(1)*	* nur amortisiert
popFront(.)	O(1)	O(1)	O(n)	* nur amortisiert
concat(.,.)	O(1)	O(1)	O(n)	
splice(.,.,.)	O(1)	O(1)	O(n)	
findNext(.,.)	O(n)	O(n)	O(n)*	* Cache-effizient

Listen / Array – Laufzeit-Vergleich

Operation	DoppeltVkttListe	EinfVkttListe	Dynamisches Array	
[.]	O(n)	O(n)	O(1)	
get(.)	O(n)	O(n)	O(1)	
set(.,.)	O(n)	O(n)	O(1)	
size()	O(1)*	O(1)*	O(1)	*nicht wenn splice mit mehreren involvierten Listen zur Verfügung steht, dann O(n) (siehe Mehlhorn Sanders 3.1.1. ganz am Ende)
first()	O(1)	O(1)	O(1)	
last()	O(1)	O(1)	O(1)	
insertAfter(.,.)	O(1)	O(1)	O(n)	
insertBefore(.,.)	O(1)	O(n)	O(n)	
remove(.)	O(1)	O(1)*	O(n)	*nur removeAfter
pushBack(.)	O(1)	O(1)	O(1)*	* nur amortisiert
pushFront(.)	O(1)	O(1)	O(n)	* nur amortisiert
popBack(.)	O(1)	O(n)	O(1)*	* nur amortisiert
popFront(.)	O(1)	O(1)	O(n)	* nur amortisiert
concat(.,.)	O(1)	O(1)	O(n)	
splice(.,.,.)	O(1)	O(1)	O(n)	
findNext(.,.)	O(n)	O(n)	O(n)*	* Cache-effizient

Stack & Queue

Stack-Methoden

- pushBack (bzw. push) : Neues Element vorne einfügen
- last (bzw. top): Liefert vorderstes Element
- popBack (bzw. pop): Vorderstes Element löschen

Queue-Methoden

- pushback (bzw. enqueue): Neues Element hinten einfügen
- last (bzw. top): Liefert vorderstes Element
- popFront: Vorderstes Element löschen

} dequeue

---

beide lassen sich mit verketteten Listen realisieren

Stack & Queue

Stack-Methoden

- pushBack (bzw. push) : Neues Element vorne einfügen
- last (bzw. top): Liefert vorderstes Element
- popBack (bzw. pop): Vorderstes Element löschen

Queue-Methoden

- pushback (bzw. enqueue): Neues Element hinten einfügen
- last (bzw. top): Liefert vorderstes Element
- popFront: Vorderstes Element löschen

} dequeue

---

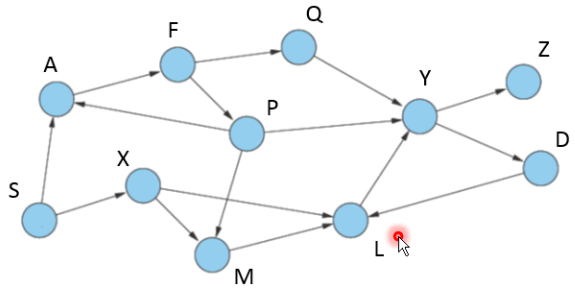
beide lassen sich mit verketteten Listen realisieren

Anwendung für Queue: Breadth First Traversierung eines Graphen

gegeben: Graph $G(V, E)$, (d.h. alle Kantengewichte == 1)

Aufgabe: Besuche alle Knoten (Mögliche Zwecke

- um etwas zu suchen oder
- auf jedem Knoten Operationen auszuführen
- oder SSSP (Single Source Shortest Path) Problem zu lösen



$V = \{S, A, X, F, Q, P, M, Y, L, Z, D\}$

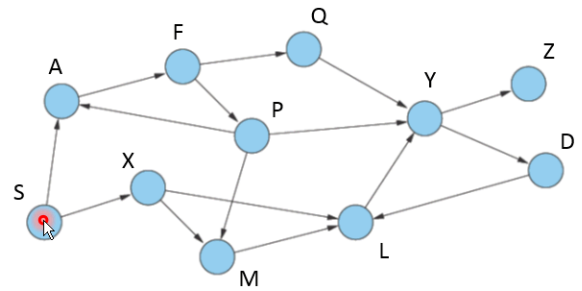
E = $\{(S, A), (A, F), (F, P), (P, A), (P, Y), (S, X), (X, M), (P, M), \ldots, (Y, D)\}$

Anwendung für Queue: Breadth First Traversierung eines Graphen

gegeben: Graph $G(V, E)$, (d.h. alle Kantengewichte == 1)

Aufgabe: Besuche alle Knoten (Mögliche Zwecke

- um etwas zu suchen oder
- auf jedem Knoten Operationen auszuführen
- oder SSSP (Single Source Shortest Path) Problem zu lösen

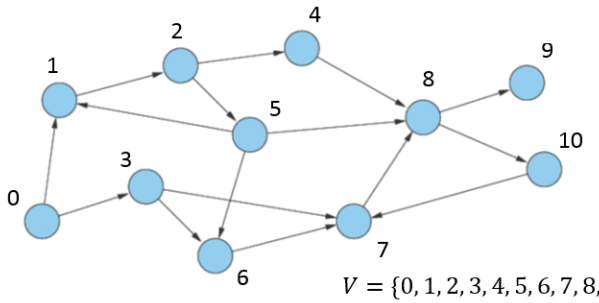


$V = \{S, A, X, F, Q, P, M, Y, L, Z, D\}$

E = $\{(S, A), (A, F), (F, P), (P, A), (P, Y), (S, X), (X, M), (P, M), \ldots, (Y, D)\}$

Anwendung für Queue: Breadth First Traversierung eines Graphen

- Zur Vereinfachung: Integer-Codierung der Knoten:



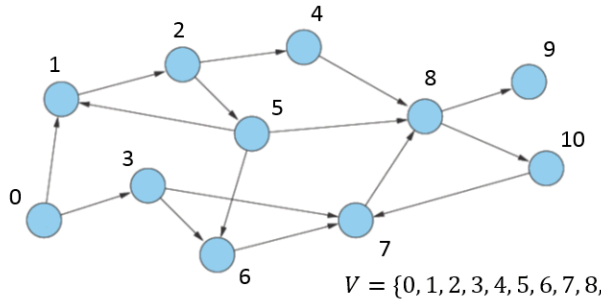
$V = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$

E = $\{(0, 1), (1, 2), (2, 5), \ldots, (8, 10)\}$

- Annahme: Es existiere Klasse `Graph`, die entsprechende Methoden zum Speichern und Zugreifen von Elementen des Graphen bereitstellt.
- Annahme: Es existiere Klasse `Queue<Integer>`, die eine Queue von Integers (←→ Knoten) implementiert.

Anwendung für Queue: Breadth First Traversierung eines Graphen

- Zur Vereinfachung: Integer-Codierung der Knoten:



$V = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$

$E = \{(0, 1), (1, 2), (2, 5), \ldots, (8, 10)\}$

- Annahme: Es existiere Klasse `Graph`, die entsprechende Methoden zum Speichern und Zugreifen von Elementen des Graphen bereitstellt.
- Annahme: Es existiere Klasse `Queue<Integer>`, die eine Queue von Integers (←→ Knoten) implementiert.

Anwendung für Queue: Breadth First Traversierung eines Graphen

```
public void bfs(Graph G, int sourceNode) {
    boolean[] marked = new boolean[G.sizeOfV()];
       //marked[v]: v has been visited
    int[] distTo = new int[G. sizeOfV()];
       //dist[v]: (current) distance to v
    int[] parentOf = new int[G. sizeOfV()];
       //parentOf[v]: (current) parent node on shortest path to v
    Queue<Integer> queue = new Queue<Integer>();
    for (int v=0; v<G.sizeOfV(); v++)
       distTo[v] = INFINITY;
    distTo[sourceNode] = 0;
    marked[sourceNode] = true;
    q.enqueue(s);

    while (!queue.isEmpty()) {
        int v = queue.dequeue();
        int[] nodesAdjacentToV = G.nodesAdjacentTo(v);
        for (int w=0; w<nodesAdjacentToV.length; w++) {
            if (!marked[w]) {
                parentOf[w] = v;
                distTo[w] = distTo[v] + 1;
                marked[w] = true;
                queue.enqueue(w);
            }
        }
    }
}
```


Anwendung für Queue: Breadth First Traversierung eines Graphen

```
public void bfs(Graph G, int sourceNode) {
    boolean[] marked = new boolean[G.sizeOfV()];
       //marked[v]: v has been visited
    int[] distTo = new int[G. sizeOfV()];
       //dist[v]: (current) distance to v
    int[] parentOf = new int[G. sizeOfV()];
       //parentOf[v]: (current) parent node on shortest path to v
    Queue<Integer> queue = new Queue<Integer>();
    for (int v=0; v<G.sizeOfV(); v++)
       distTo[v] = INFINITY;
    distTo[sourceNode] = 0;
    marked[sourceNode] = true;
    q.enqueue(s);

    while (!queue.isEmpty()) {
        int v = queue.dequeue();
        int[] nodesAdjacentToV = G.nodesAdjacentTo(v);
        for (int w=0; w<nodesAdjacentToV.length; w++) {
            if (!marked[w]) {
                parentOf[w] = v;
                distTo[w] = distTo[v] + 1;
                marked[w] = true;
                queue.enqueue(w);
            }
        }
    }
}
```


Anwendung für Queue: Breadth First Traversierung eines Graphen

```
public void bfs(Graph G, int sourceNode) {
    boolean[] marked = new boolean[G.sizeOfV()];
       //marked[v]: v has been visited
    int[] distTo = new int[G. sizeOfV()];
       //dist[v]: (current) distance to v
    int[] parentOf = new int[G. sizeOfV()];
       //parentOf[v]: (current) parent node on shortest path to v
    Queue<Integer> queue = new Queue<Integer>();
    for (int v=0; v<G.sizeOfV(); v++)
       distTo[v] = INFINITY;
    distTo[sourceNode] = 0;
    marked[sourceNode] = true;
    q.enqueue(s);

    while (!queue.isEmpty()) {
        int v = queue.dequeue();
        int[] nodesAdjacentToV = G.nodesAdjacentTo(v);
        for (int w=0; w<nodesAdjacentToV.length; w++) {
            if (!marked[w]) {
                parentOf[w] = v;
                distTo[w] = distTo[v] + 1;
                marked[w] = true;
                queue.enqueue(w);
            }
        }
    }
}
```

Anwendung für Queue: Breadth First Traversierung eines Graphen

```
public void bfs(Graph G, int sourceNode) {
    boolean[] marked = new boolean[G.sizeOfV()];
       //marked[v]: v has been visited
    int[] distTo = new int[G. sizeOfV()];
       //dist[v]: (current) distance to v
    int[] parentOf = new int[G. sizeOfV()];
       //parentOf[v]: (current) parent node on shortest path to v
    Queue<Integer> queue = new Queue<Integer>();
    for (int v=0; v<G.sizeOfV(); v++)
       distTo[v] = INFINITY;
    distTo[sourceNode] = 0;
    marked[sourceNode] = true;
    q.enqueue(s);

    while (!queue.isEmpty()) {
        int v = queue.dequeue();
        int[] nodesAdjacentToV = G.nodesAdjacentTo(v);
        for (int w=0; w<nodesAdjacentToV.length; w++) {
            if (!marked[w]) {
                parentOf[w] = v;
                distTo[w] = distTo[v] + 1;
                marked[w] = true;
                queue.enqueue(w);
            }
        }
    }
}
```


Anwendung für Queue: Breadth First Traversierung eines Graphen

```
public void bfs(Graph G, int sourceNode) {
    boolean[] marked = new boolean[G.sizeOfV()];
       //marked[v]: v has been visited
    int[] distTo = new int[G. sizeOfV()];
       //dist[v]: (current) distance to v
    int[] parentOf = new int[G. sizeOfV()];
       //parentOf[v]: (current) parent node on shortest path to v
    Queue<Integer> queue = new Queue<Integer>();
    for (int v=0; v<G.sizeOfV(); v++)
       distTo[v] = INFINITY;
    distTo[sourceNode] = 0;
    marked[sourceNode] = true;
    q.enqueue(s);

    while (!queue.isEmpty()) {
        int v = queue.dequeue();
        int[] nodesAdjacentToV = G.nodesAdjacentTo(v);
        for (int w=0; w<nodesAdjacentToV.length; w++) {
            if (!marked[w]) {
                parentOf[w] = v;
                distTo[w] = distTo[v] + 1;
                marked[w] = true;
                queue.enqueue(w);
            }
        }
    }
}
```


Anwendung für Queue: Breadth First Traversierung eines Graphen

```
public void bfs(Graph G, int sourceNode) {
    boolean[] marked = new boolean[G.sizeOfV()];
       //marked[v]: v has been visited
    int[] distTo = new int[G. sizeOfV()];
       //dist[v]: (current) distance to v
    int[] parentOf = new int[G. sizeOfV()];
       //parentOf[v]: (current) parent node on shortest path to v
    Queue<Integer> queue = new Queue<Integer>();
    for (int v=0; v<G.sizeOfV(); v++)
       distTo[v] = INFINITY;
    distTo[sourceNode] = 0;
    marked[sourceNode] = true;
    q.enqueue(s);

    while (!queue.isEmpty()) {
        int v = queue.dequeue();
        int[] nodesAdjacentToV = G.nodesAdjacentTo(v);
        for (int w=0; w<nodesAdjacentToV.length; w++) {
            if (!marked[w]) {
                parentOf[w] = v;
                distTo[w] = distTo[v] + 1;
                marked[w] = true;
                queue.enqueue(w);
            }
        }
    }
}
```


Anwendung für Queue: Breadth First Traversierung eines Graphen

```
public void bfs(Graph G, int sourceNode) {
    boolean[] marked = new boolean[G.sizeOfV()];
       //marked[v]: v has been visited
    int[] distTo = new int[G. sizeOfV()];
       //dist[v]: (current) distance to v
    int[] parentOf = new int[G. sizeOfV()];
       //parentOf[v]: (current) parent node on shortest path to v
    Queue<Integer> queue = new Queue<Integer>();
    for (int v=0; v<G.sizeOfV(); v++)
       distTo[v] = INFINITY;
    distTo[sourceNode] = 0;
    marked[sourceNode] = true;
    q.enqueue(s);

    while (!queue.isEmpty()) {
        int v = queue.dequeue();
        int[] nodesAdjacentToV = G.nodesAdjacentTo(v);
        for (int w=0; w<nodesAdjacentToV.length; w++) {
            if (!marked[w]) {
                parentOf[w] = v;
                distTo[w] = distTo[v] + 1;
                marked[w] = true;
                queue.enqueue(w);
            }
        }
    }
}
```

Anwendung für Queue: Breadth First Traversierung eines Graphen

```
public void bfs(Graph G, int sourceNode) {
    boolean[] marked = new boolean[G.sizeOfV()];
       //marked[v]: v has been visited
    int[] distTo = new int[G. sizeOfV()];
       //dist[v]: (current) distance to v
    int[] parentOf = new int[G. sizeOfV()];
       //parentOf[v]: (current) parent node on shortest path to v
    Queue<Integer> queue = new Queue<Integer>();
    for (int v=0; v<G.sizeOfV(); v++)
       distTo[v] = INFINITY;
    distTo[sourceNode] = 0;
    marked[sourceNode] = true;
    q.enqueue(s);

    while (!queue.isEmpty()) {
        int v = queue.dequeue();
        int[] nodesAdjacentToV = G.nodesAdjacentTo(v);
        for (int w=0; w<nodesAdjacentToV.length; w++) {
            if (!marked[w]) {
                parentOf[w] = v;
                distTo[w] = distTo[v] + 1;
                marked[w] = true;
                queue.enqueue(w);
            }
        }
    }
}
```


Anwendung für Queue: Breadth First Traversierung eines Graphen

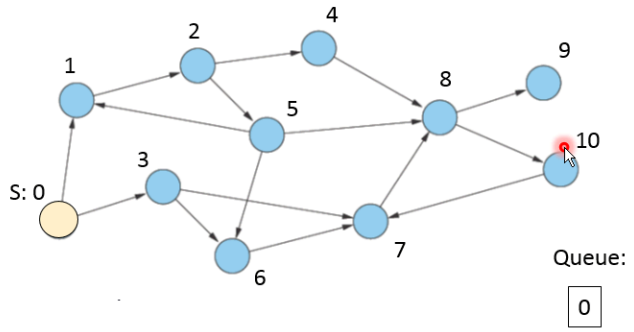
```
public void bfs(Graph G, int sourceNode) {
    boolean[] marked = new boolean[G.sizeOfV()];
       //marked[v]: v has been visited
    int[] distTo = new int[G. sizeOfV()];
       //dist[v]: (current) distance to v
    int[] parentOf = new int[G. sizeOfV()];
       //parentOf[v]: (current) parent node on shortest path to v
    Queue<Integer> queue = new Queue<Integer>();
    for (int v=0; v<G.sizeOfV(); v++)
       distTo[v] = INFINITY;
    distTo[sourceNode] = 0;
    marked[sourceNode] = true;
    q.enqueue(s);

    while (!queue.isEmpty()) {
        int v = queue.dequeue();
        int[] nodesAdjacentToV = G.nodesAdjacentTo(v);
        for (int w=0; w<nodesAdjacentToV.length; w++) {
            if (!marked[w]) {
                parentOf[w] = v;
                distTo[w] = distTo[v] + 1;
                marked[w] = true;
                queue.enqueue(w);
            }
        }
    }
}
```

64

Anwendung für Queue: Breadth First Traversierung eines Graphen

```
while (!queue.isEmpty()) {
    int v = queue.dequeue();
    int[] nodesAdjacentToV = G.nodesAdjacentTo(v);
    for (int w=0; w<nodesAdjacentToV.length; w++) {
        if (!marked[w]) {
            parentOf[w] = v;
            distTo[w] = distTo[v] + 1;
            marked[w] = true;
            queue.enqueue(w);
        }
    }
}
```



marked

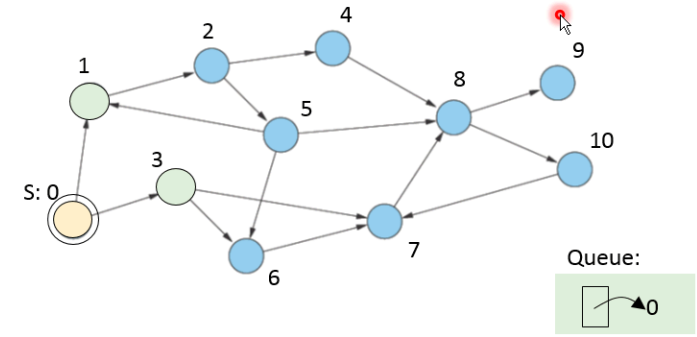
	distTo	parentOf
0	0	
1	∞	
2	∞	
3	∞	
4	∞	
5	∞	
6	∞	
7	∞	
8	∞	
9	∞	
10	∞	

Anwendung für Queue: Breadth First Traversierung eines Graphen

```
while (!queue.isEmpty()) {
    int v = queue.dequeue();
    int[] nodesAdjacentToV = G.nodesAdjacentTo(v);
    for (int w=0; w<nodesAdjacentToV.length; w++) {
        if (!marked[w]) {
            parentOf[w] = v;
            distTo[w] = distTo[v] + 1;
            marked[w] = true;
            queue.enqueue(w);
        }
    }
}
```

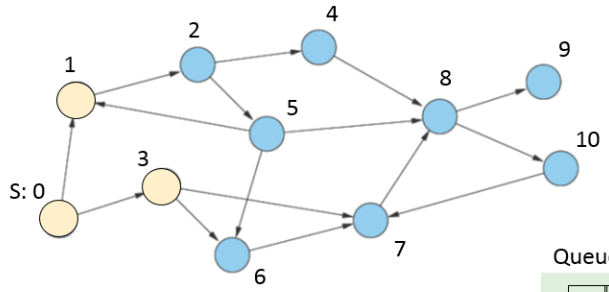
marked

	distTo	parentOf
0	0	
1	∞	
2	∞	
3	∞	
4	∞	
5	∞	
6	∞	
7	∞	
8	∞	
9	∞	
10	∞	



Anwendung für Queue: Breadth First Traversierung eines Graphen

```
while (!queue.isEmpty()) {
    int v = queue.dequeue();
    int[] nodesAdjacentToV = G.nodesAdjacentTo(v);
    for (int w=0; w<nodesAdjacentToV.length; w++) {
        if (!marked[w]) {
            parentOf[w] = v;
            distTo[w] = distTo[v] + 1;
            marked[w] = true;
            queue.enqueue(w);
        }
    }
}
```



Queue:

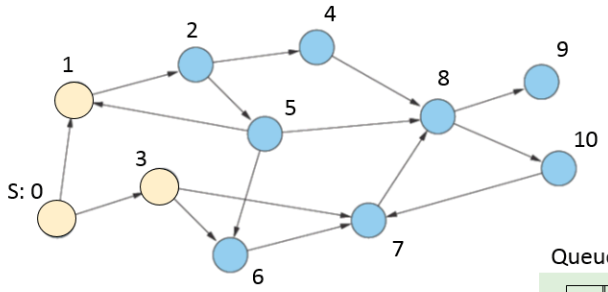
1	3

marked

	distTo	parentOf
0	0	
1	1	0
2	∞	
3	1	0
4	∞	
5	∞	
6	∞	
7	∞	
8	∞	
9	∞	
10	∞	

Anwendung für Queue: Breadth First Traversierung eines Graphen


```
while (!queue.isEmpty()) {
    int v = queue.dequeue();
    int[] nodesAdjacentToV = G.nodesAdjacentTo(v);
    for (int w=0; w<nodesAdjacentToV.length; w++) {
        if (!marked[w]) {
            parentOf[w] = v;
            distTo[w] = distTo[v] + 1;
            marked[w] = true;
            queue.enqueue(w);
        }
    }
}
```

Queue:

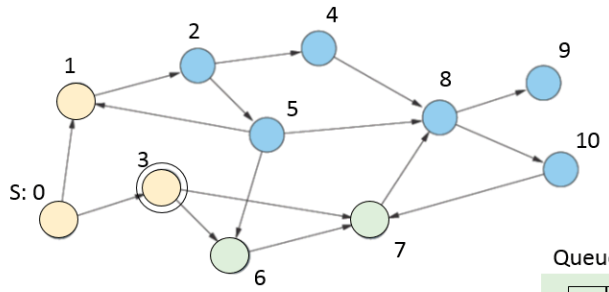
1	3

marked

	distTo	parentOf
0	0	
1	1	0
2	∞	
3	1	0
4	∞	
5	∞	
6	∞	
7	∞	
8	∞	
9	∞	
10	∞	

Anwendung für Queue: Breadth First Traversierung eines Graphen


```
while (!queue.isEmpty()) {
    int v = queue.dequeue();
    int[] nodesAdjacentToV = G.nodesAdjacentTo(v);
    for (int w=0; w<nodesAdjacentToV.length; w++) {
        if (!marked[w]) {
            parentOf[w] = v;
            distTo[w] = distTo[v] + 1;
            marked[w] = true;
            queue.enqueue(w);
        }
    }
}
```

Queue:

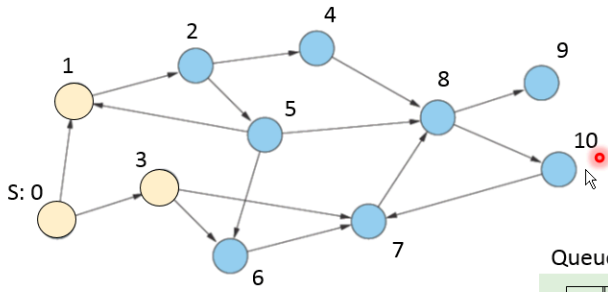
1		3

marked

	distTo	parentOf
0	0	
1	1	0
2	∞	
3	1	0
4	∞	
5	∞	
6	∞	
7	∞	
8	∞	
9	∞	
10	∞	

Anwendung für Queue: Breadth First Traversierung eines Graphen


```
while (!queue.isEmpty()) {
    int v = queue.dequeue();
    int[] nodesAdjacentToV = G.nodesAdjacentTo(v);
    for (int w=0; w<nodesAdjacentToV.length; w++) {
        if (!marked[w]) {
            parentOf[w] = v;
            distTo[w] = distTo[v] + 1;
            marked[w] = true;
            queue.enqueue(w);
        }
    }
}
```

Queue:

1	3

marked

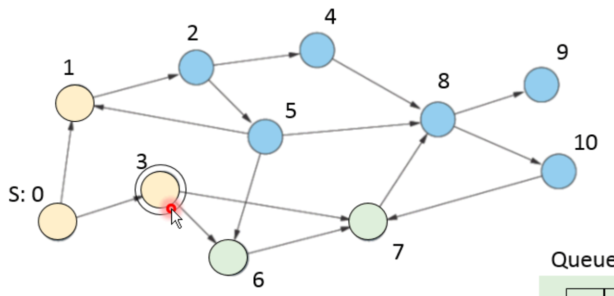
	distTo	parentOf
0	0	
1	1	0
2	∞	
3	1	0
4	∞	
5	∞	
6	∞	
7	∞	
8	∞	
9	∞	
10	∞	

Anwendung für Queue: Breadth First Traversierung eines Graphen

```
while (!queue.isEmpty()) {
    int v = queue.dequeue();
    int[] nodesAdjacentToV = G.nodesAdjacentTo(v);
    for (int w=0; w<nodesAdjacentToV.length; w++) {
        if (!marked[w]) {
            parentOf[w] = v;
            distTo[w] = distTo[v] + 1;
            marked[w] = true;
            queue.enqueue(w);
        }
    }
}
```

marked

	distTo	parentOf
0	0	
1	1	0
2	∞	
3	1	0
4	∞	
5	∞	
6	∞	
7	∞	
8	∞	
9	∞	
10	∞	



Queue:

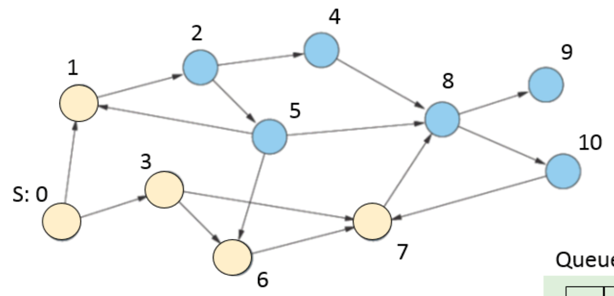
1		3

Anwendung für Queue: Breadth First Traversierung eines Graphen

```
while (!queue.isEmpty()) {
    int v = queue.dequeue();
    int[] nodesAdjacentToV = G.nodesAdjacentTo(v);
    for (int w=0; w<nodesAdjacentToV.length; w++) {
        if (!marked[w]) {
            parentOf[w] = v;
            distTo[w] = distTo[v] + 1;
            marked[w] = true;
            queue.enqueue(w);
        }
    }
}
```

marked

	distTo	parentOf
0	0	
1	1	0
2	∞	
3	1	0
4	∞	
5	∞	
6	2	3
7	2	3
8	∞	
9	∞	
10	∞	



Queue:

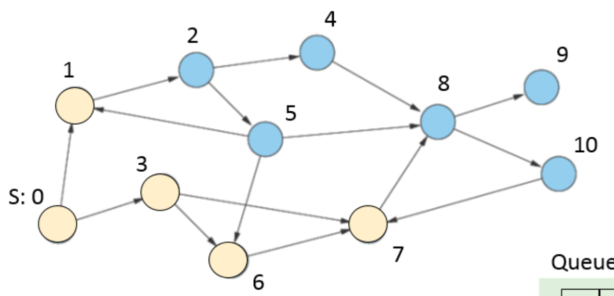
6	7	1

Anwendung für Queue: Breadth First Traversierung eines Graphen

```
while (!queue.isEmpty()) {
    int v = queue.dequeue();
    int[] nodesAdjacentToV = G.nodesAdjacentTo(v);
    for (int w=0; w<nodesAdjacentToV.length; w++) {
        if (!marked[w]) {
            parentOf[w] = v;
            distTo[w] = distTo[v] + 1;
            marked[w] = true;
            queue.enqueue(w);
        }
    }
}
```

marked

	distTo	parentOf
0	0	
1	1	0
2	∞	
3	1	0
4	∞	
5	∞	
6	2	3
7	2	3
8	∞	
9	∞	
10	∞	



Queue:

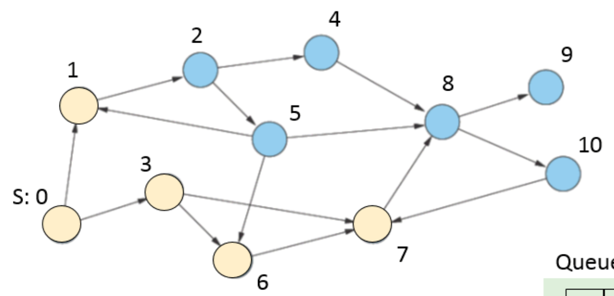
6	7	1

Anwendung für Queue: Breadth First Traversierung eines Graphen

```
while (!queue.isEmpty()) {
    int v = queue.dequeue();
    int[] nodesAdjacentToV = G.nodesAdjacentTo(v);
    for (int w=0; w<nodesAdjacentToV.length; w++) {
        if (!marked[w]) {
            parentOf[w] = v;
            distTo[w] = distTo[v] + 1;
            marked[w] = true;
            queue.enqueue(w);
        }
    }
}
```

marked

	distTo	parentOf
0	0	
1	1	0
2	∞	
3	1	0
4	∞	
5	∞	
6	2	3
7	2	3
8	∞	
9	∞	
10	∞	

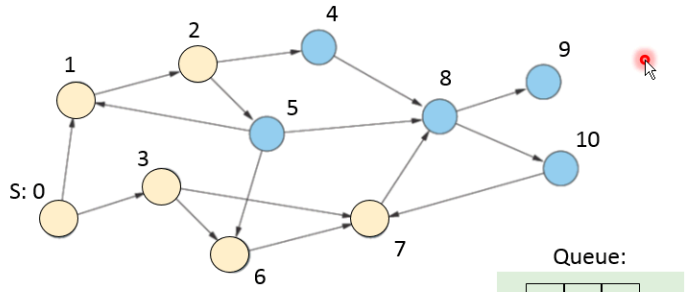


Queue:

6	7	1

Anwendung für Queue: Breadth First Traversierung eines Graphen

```
while (!queue.isEmpty()) {
    int v = queue.dequeue();
    int[] nodesAdjacentToV = G.nodesAdjacentTo(v);
    for (int w=0; w<nodesAdjacentToV.length; w++) {
        if (!marked[w]) {
            parentOf[w] = v;
            distTo[w] = distTo[v] + 1;
            marked[w] = true;
            queue.enqueue(w);
        }
    }
}
```



Queue:

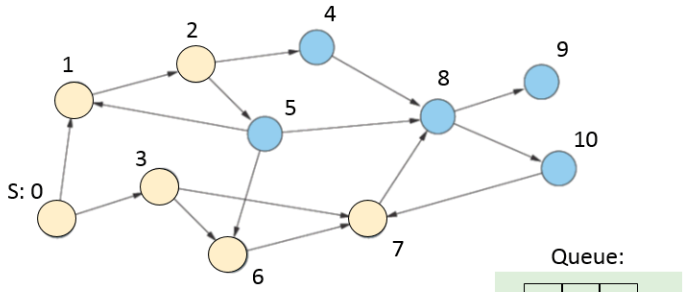
2	6	7

marked

	distTo	parentOf
0	0	
1	1	0
2	2	1
3	1	0
4	∞	
5	∞	
6	2	3
7	2	3
8	∞	
9	∞	
10	∞	

Anwendung für Queue: Breadth First Traversierung eines Graphen


```
while (!queue.isEmpty()) {
    int v = queue.dequeue();
    int[] nodesAdjacentToV = G.nodesAdjacentTo(v);
    for (int w=0; w<nodesAdjacentToV.length; w++) {
        if (!marked[w]) {
            parentOf[w] = v;
            distTo[w] = distTo[v] + 1;
            marked[w] = true;
            queue.enqueue(w);
        }
    }
}
```

Queue:

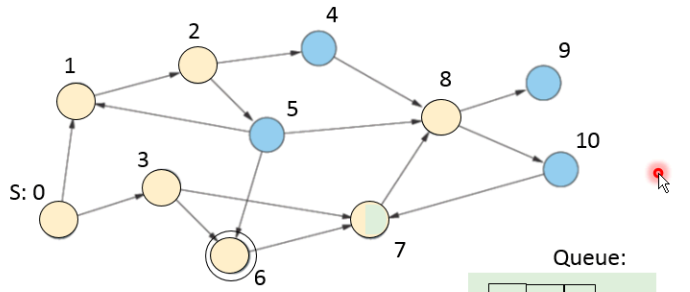
2	6	7

marked

	distTo	parentOf
0	0	
1	1	0
2	2	1
3	1	0
4	∞	
5	∞	
6	2	3
7	2	3
8	∞	
9	∞	
10	∞	

Anwendung für Queue: Breadth First Traversierung eines Graphen


```
while (!queue.isEmpty()) {
    int v = queue.dequeue();
    int[] nodesAdjacentToV = G.nodesAdjacentTo(v);
    for (int w=0; w<nodesAdjacentToV.length; w++) {
        if (!marked[w]) {
            parentOf[w] = v;
            distTo[w] = distTo[v] + 1;
            marked[w] = true;
            queue.enqueue(w);
        }
    }
}
```

Queue:

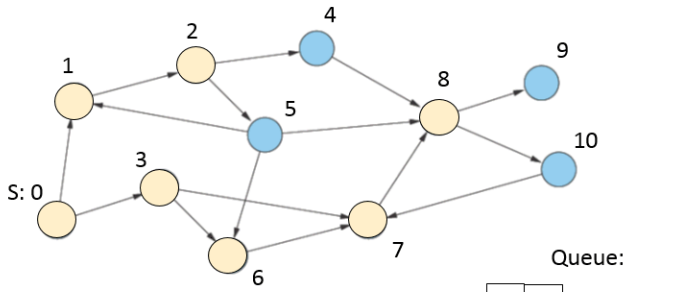
8	2	→ 6

marked

	distTo	parentOf
0	0	
1	1	0
2	2	1
3	1	0
4	∞	
5	∞	
6	2	3
7	2	3
8	3	7
9	∞	
10	∞	

Anwendung für Queue: Breadth First Traversierung eines Graphen


```
while (!queue.isEmpty()) {
    int v = queue.dequeue();
    int[] nodesAdjacentToV = G.nodesAdjacentTo(v);
    for (int w=0; w<nodesAdjacentToV.length; w++) {
        if (!marked[w]) {
            parentOf[w] = v;
            distTo[w] = distTo[v] + 1;
            marked[w] = true;
            queue.enqueue(w);
        }
    }
}
```

Queue:

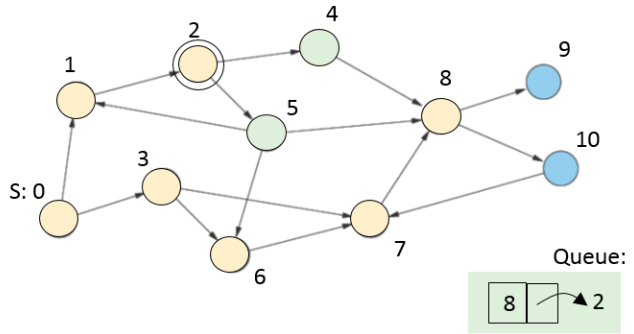
8	2

marked

	distTo	parentOf
0	0	
1	1	0
2	2	1
3	1	0
4	∞	
5	∞	
6	2	3
7	2	3
8	3	7
9	∞	
10	∞	

Anwendung für Queue: Breadth First Traversierung eines Graphen

```
while (!queue.isEmpty()) {
    int v = queue.dequeue();
    int[] nodesAdjacentToV = G.nodesAdjacentTo(v);
    for (int w=0; w<nodesAdjacentToV.length; w++) {
        if (!marked[w]) {
            parentOf[w] = v;
            distTo[w] = distTo[v] + 1;
            marked[w] = true;
            queue.enqueue(w);
        }
    }
}
```

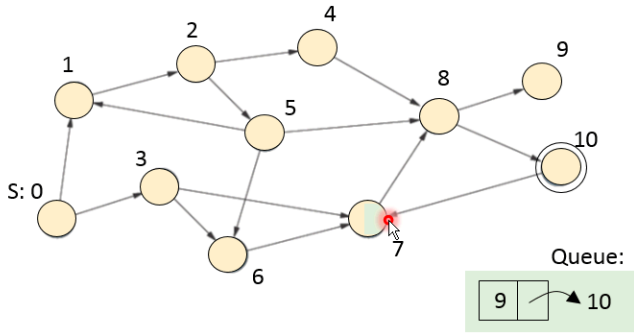


marked

	distTo	parentOf
0	0	
1	1	0
2	2	1
3	1	0
4	∞	
5	∞	
6	2	3
7	2	3
8	3	7
9	∞	
10	∞	

Anwendung für Queue: Breadth First Traversierung eines Graphen

```
while (!queue.isEmpty()) {
    int v = queue.dequeue();
    int[] nodesAdjacentToV = G.nodesAdjacentTo(v);
    for (int w=0; w<nodesAdjacentToV.length; w++) {
        if (!marked[w]) {
            parentOf[w] = v;
            distTo[w] = distTo[v] + 1;
            marked[w] = true;
            queue.enqueue(w);
        }
    }
}
```

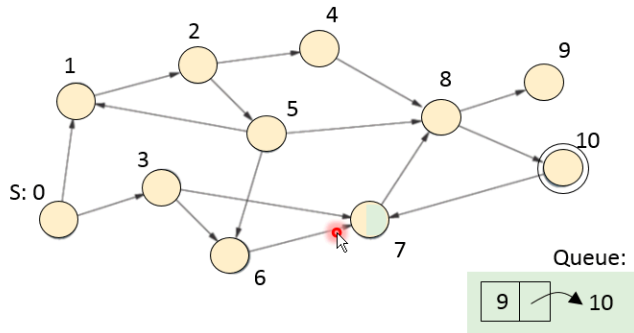


marked

	distTo	parentOf
0	0	
1	1	0
2	2	1
3	1	0
4	3	2
5	3	2
6	2	3
7	2	3
8	3	7
9	4	8
10	4	8

Anwendung für Queue: Breadth First Traversierung eines Graphen

```
while (!queue.isEmpty()) {
    int v = queue.dequeue();
    int[] nodesAdjacentToV = G.nodesAdjacentTo(v);
    for (int w=0; w<nodesAdjacentToV.length; w++) {
        if (!marked[w]) {
            parentOf[w] = v;
            distTo[w] = distTo[v] + 1;
            marked[w] = true;
            queue.enqueue(w);
        }
    }
}
```

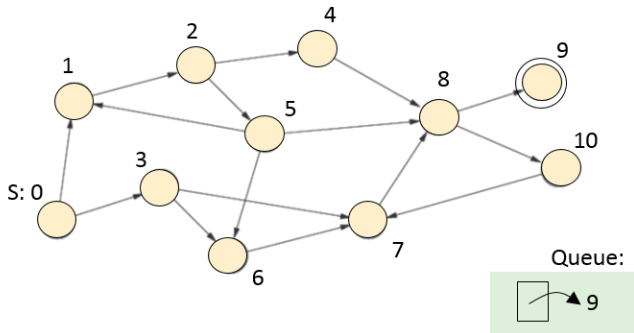


marked

	distTo	parentOf
0	0	
1	1	0
2	2	1
3	1	0
4	3	2
5	3	2
6	2	3
7	2	3
8	3	7
9	4	8
10	4	8

Anwendung für Queue: Breadth First Traversierung eines Graphen

```
while (!queue.isEmpty()) {
    int v = queue.dequeue();
    int[] nodesAdjacentToV = G.nodesAdjacentTo(v);
    for (int w=0; w<nodesAdjacentToV.length; w++) {
        if (!marked[w]) {
            parentOf[w] = v;
            distTo[w] = distTo[v] + 1;
            marked[w] = true;
            queue.enqueue(w);
        }
    }
}
```

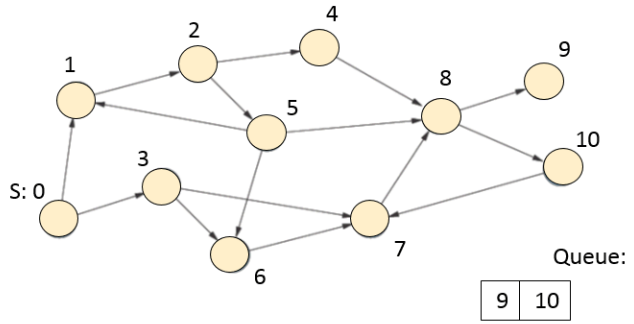


marked

	distTo	parentOf
0	0	
1	1	0
2	2	1
3	1	0
4	3	2
5	3	2
6	2	3
7	2	3
8	3	7
9	4	8
10	4	8

Anwendung für Queue: Breadth First Traversierung eines Graphen

```
while (!queue.isEmpty()) {
    int v = queue.dequeue();
    int[] nodesAdjacentToV = G.nodesAdjacentTo(v);
    for (int w=0; w<nodesAdjacentToV.length; w++) {
        if (!marked[w]) {
            parentOf[w] = v;
            distTo[w] = distTo[v] + 1;
            marked[w] = true;
            queue.enqueue(w);
        }
    }
}
```



Queue:

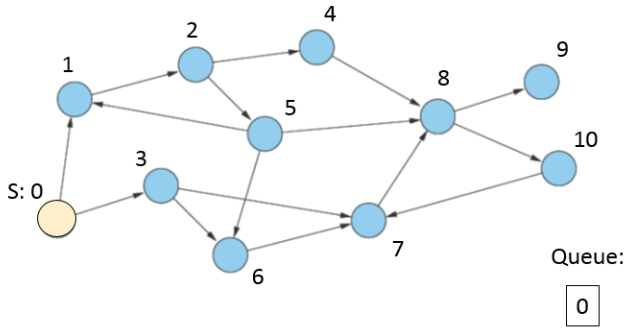
9	10

marked

	distTo	parentOf
0	0	
1	1	0
2	2	1
3	1	0
4	3	2
5	3	2
6	2	3
7	2	3
8	3	7
9	4	8
10	4	8

Anwendung für Queue: Breadth First Traversierung eines Graphen

```
while (!queue.isEmpty()) {
    int v = queue.dequeue();
    int[] nodesAdjacentToV = G.nodesAdjacentTo(v);
    for (int w=0; w<nodesAdjacentToV.length; w++) {
        if (!marked[w]) {
            parentOf[w] = v;
            distTo[w] = distTo[v] + 1;
            marked[w] = true;
            queue.enqueue(w);
        }
    }
}
```



Queue:

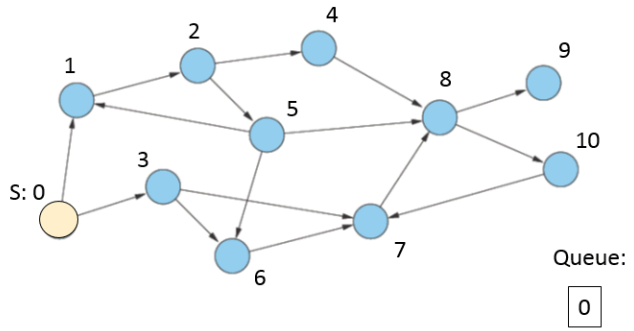
0

marked

	distTo	parentOf
0	0	
1	∞	
2	∞	
3	∞	
4	∞	
5	∞	
6	∞	
7	∞	
8	∞	
9	∞	
10	∞	

Anwendung für Queue: Breadth First Traversierung eines Graphen

```
while (!queue.isEmpty()) {
    int v = queue.dequeue();
    int[] nodesAdjacentToV = G.nodesAdjacentTo(v);
    for (int w=0; w<nodesAdjacentToV.length; w++) {
        if (!marked[w]) {
            parentOf[w] = v;
            distTo[w] = distTo[v] + 1;
            marked[w] = true;
            queue.enqueue(w);
        }
    }
}
```



Queue:

0

marked

	distTo	parentOf
0	0	
1	∞	
2	∞	
3	∞	
4	∞	
5	∞	
6	∞	
7	∞	
8	∞	
9	∞	
10	∞	