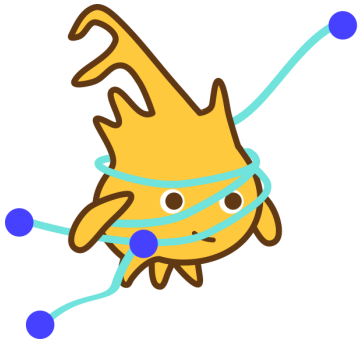
Dieses Beispiel baut auf den Kapiteln
Blockpalette, **First-class Objekte** und **Datentypen** auf!

Of nodes and edges



Link zum Projekt: http://tiny.cc/al6ubz

In diesem Beispiel soll ein Framework aufgezeigt werden, mit dessen Hilfe es möglich ist, Graphen und ihre Algorithmen direkt in Snap! zu implementieren. Es wird wenig Vorwissen benötigt; dennoch wäre es gut, wenn die Zielgruppe bereits erste Erfahrungen mit blockbasierten Sprachen, z. B. in Form von Scratch gesammelt hat.
Wir verwenden neue Bedienkonzepte, die blockbasierte Sprachen uns ermöglichen - wie etwa **Tinkering** und **Direct Drive**.

Überblick über die verwendeten Symbole in diesem Handout:

	Aufgabe
	Bonusaufgabe für Fortgeschrittene, bzw. Benutzer mit Vorerfahrung
	Hinweis
	Tinkering: hier gibt es keine falschen Antworten!
	Gespräch: Reflektion, Diskussion, oder Austausch mit dem Banknachbar
	Tipp: Hinweise, bzw. Ideen für die Umsetzung im Unterricht

Diese Einheit zeigt ein Framework mit dessen Hilfe die Datenstruktur Graph visualisiert und somit besonders gewinnbringend thematisiert werden kann. Die besondere Stärke des Frameworks liegt darin, dass alle Funktionalitäten direkt in Snap! implementiert wurden, d.h. jeder Block ist einsehbar.

Tipp: Das ist besonders deswegen gut, weil schnelle Schülerinnen und Schüler Erweiterungen und Verbesserungen am Framework selbst durchführen können. Eine solche Binnendifferenzierung ermöglicht es, Schülerinnen und Schüler unterschiedlichster Kompetenzstufen anzusprechen.

Was bietet uns das Framework?
Es ermöglicht die Erstellung von Graphen und Netzwerken in Snap!. Knoten werden platziert und dabei automatisch von A - I abhängig von ihrer Anzahl benannt. Knoten können danach dann durch Mausklick verbunden werden. Dabei wird eine Kante vom Start- zum Zielknoten erzeugt, mit einer Längenangabe versehen, und ein entsprechender Verweis im Attribut “Nachbarn” der jeweiligen Knoten gesetzt. Die Längenangaben/Gewichtungen können mithilfe der Pfeiltasten angezeigt und versteckt werden.
Wurden Knoten miteinander verbunden, so bewirkt ein Mouseover eine farbliche Hervorhebung der Nachbarn **(visueller “getter”).** Knoten verfügen über viele vorgefertigte Blöcke. Dazu gehört beispielsweise ein Funktionsblock, der zurückliefert, welche Kanten der Knoten berührt, ein Block, der zurückliefert, welche Kante die kürzeste davon ist, und ein Block, der den nächstgelegenen Nachbarn (direkt als Objekt!) zurückliefert.
Schließlich können Knoten mithilfe eines entsprechenden Blocks direkt über ihren Bezeichner adressiert werden. Damit ist es ein Leichtes, Graphenalgorithmen und -funktionen innerhalb des Frameworks umzusetzen.
Das Ziel dieser Einheit ist demnach eine Abdeckung der typischen Lehrplaninhalte einer Einheit zur Datenstruktur Graph. Als Beispiel hier ein Auszug aus dem bayerischen LehrplanPLUS:

Lehrplaninhalte zu den Kompetenzen:

- *Eigenschaften von Graphen: gerichtet, ungerichtet, zusammenhängend, unzusammenhängend, bewertet (gewichtet), unbewertet, mit Zyklen, zyklenfrei, Erreichbarkeit von Knoten*
- *Adjazenzmatrix, zweidimensionales Feld*

Als letzten (hier nicht gelisteten) Punkt ist die Implementierung von Graphenalgorithmen wie der Breiten- oder Tiefensuche angegeben. Beide Suchalgorithmen werden wir innerhalb dieser Einheit implementieren, doch zunächst bietet es sich an, einige wichtige Begrifflichkeiten anhand konkreter Beispiele zu diskutieren.

Aufgabe: Öffnen Sie das Framework und klicken Sie auf die grüne Flagge. Platzieren Sie einige Knoten und verbinden Sie diese, um

- einen **zusammenhängenden**
- einen **unzusammenhängenden**
- einen **zyklenfreien**
- und einen **zyklischen** Graphen zu erzeugen.

Hinweis*: Das Framework muss nach jeder Graphenerzeugung durch den roten Knopf gestoppt und neu gestartet werden!*

Tipp: Im Objekt **node** finden Sie den Block 

. Dieser Block kann für Eingabewerte 1-5 **Beispielgraphen** erzeugen. Der Block eignet sich somit für die **Generierung von Aufgaben** für Schülerinnen und Schüler, beispielsweise *"verbindet den Graphen von Beispielformation 1 so, dass ein Zyklus der Länge 5 entsteht!"*

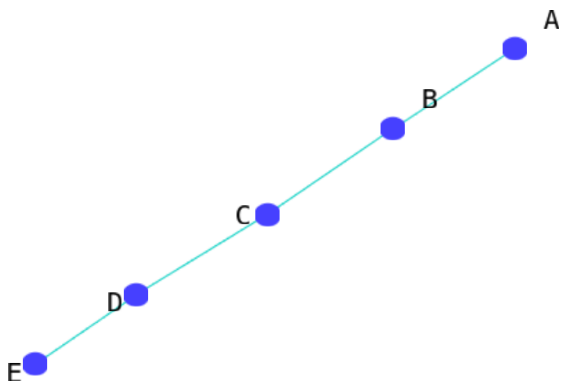
Tipp: Es empfiehlt sich, die Begriffe den Schülerinnen und Schülern anhand dieser konkreten Beispiele zu erläutern. Außerdem ist es gewinnbringend, den die Schülerinnen und Schüler jeweils die Aufgabe zu geben, eigene Graphen für jede der Arten zu erstellen.
Beachten Sie, dass das Framework **nur ungerichtete** Graphen erlaubt. Zum Vergleich, bzw. zur Verdeutlichung kann ein im Framework erzeugter Graph auf der Tafel skizziert werden und zu einem gerichteten Graphen gemacht werden. Graphen können mithilfe der Pfeiltasten zwischen **gewichtet** und **ungewichtet** geschalten werden. **Zu Demonstrationszwecken ist es außerdem möglich, die Kanten zu verstecken, beispielsweise um verschiedene Verbindungsmöglichkeiten zwischen Knoten vorzuführen.**

Tipp: Der Block 

mit Eingabewert **5** erzeugt einen Graphen, der sich besonders gut dafür eignet, als **Binärbaum** verbunden zu werden! Anhand dieses Graphen, bzw. Baumes können Bäume als Datenstruktur thematisiert werden. Auch die **Tiefen-**, bzw. **Breitensuche**, die wir im weiteren Verlauf dieser Einheit implementieren, wird bei Binärbäumen besonders deutlich.

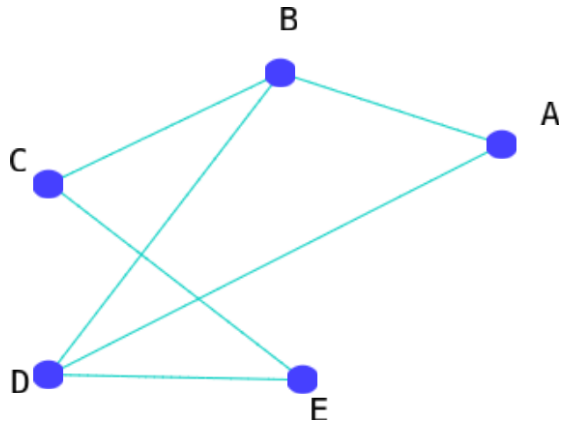
Auch **Aufgaben zu Adjazenzmatrizen** lassen sich mithilfe des Frameworks leicht erstellen:

Aufgabe: Erstellen Sie für die folgenden vorgegebenen Graphen jeweils eine Adjazenzmatrix!



a) **Adjazenzmatrix**

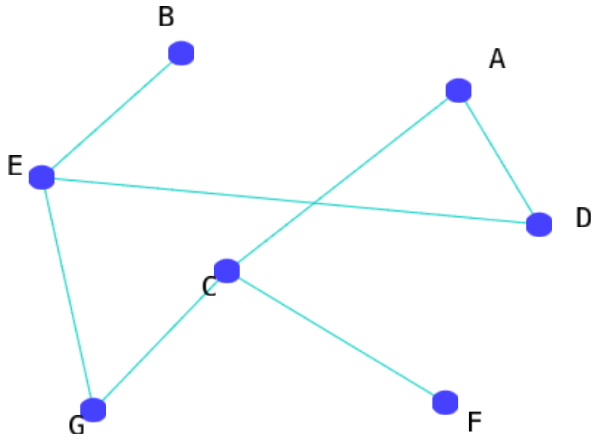
	A	B	C	D	E
A	0	1			
B	1	0			
C			0		
D				0	
E					0



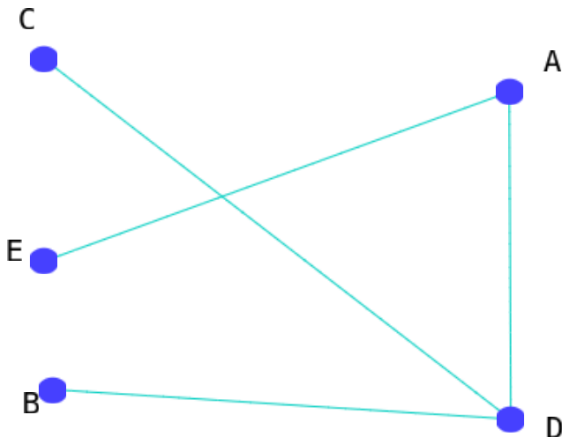
b) **Adjazenzmatrix**

	A	B	C	D	E
A	0	1			
B	1	0			
C			0		
D				0	
E					0

c)



d)



Zum Abschluss soll in dieser Einheit gezeigt werden, wie sich die Graphalgorithmen der Breiten- und Tiefensuche innerhalb des Frameworks implementiert werden können.

Vor dieser Einheit werden typischerweise Stapel und Listen thematisiert. Beide Datenstrukturen finden hier Verwendung, da sie den zentralen Unterschied zwischen Tiefen- und Breitensuche ausmachen.

Tipp: Es empfiehlt sich, zunächst beide Datenstrukturen, d.h. Listen und Stapel zu wiederholen, da Letzterer in dieser Einheit in Snap! implementiert werden soll. Es bietet sich an, hierfür Beispiele aus dem echten Leben zu verwenden:
Liste: Warteschlange beim Bäcker; wer zuerst ankommt wird zuerst bedient; **first-in-first-out-Prinzip**
Stapel: Pfannkuchen auf dem Teller; sie werden der Zubereitungsreihenfolge nach auf den Teller gestapelt und in entgegengesetzter Reihenfolge gegessen; **last-in-first-out-Prinzip**

Zunächst wenden wir uns der Liste zu. Wichtige Operationen auf Listen sind das **Hinzufügen** und das **Entfernen** von Elementen. Snap! gibt uns diese Operationen, sowie die gesamte Datenstruktur einer Liste bereits **nativ** vor:

Mithilfe dieser Bausteine und den vorgegebenen Blöcken des Frameworks können wir nun einen Block zur Breitensuche implementieren.

! Das tun wir im Objekt **"examples"!** Die Objekte **pen, line** und **label** sind für uns **uninteressant.**

Im Objekt **node** ist lediglich der 

-Block **für uns relevant.**

Breitensuche

Aufgabe: Erzeugen Sie einen **neuen Block**. Der Block soll:

- ein Befehlsblock sein
- einen passenden (sprechenden) Namen tragen
- **drei** Eingabeparameter haben: einen Startknoten, eine Liste der noch zu besuchenden Knoten, und eine Liste der bisher besuchten Knoten
- einer passenden Kategorie angehören (beispielsweise Listen)

Mögliche Lösung: 

Nun bearbeiten wir den **Inhalt des Blocks.** Die Bilder sollen dabei keine Musterlösung darstellen, sondern eine mögliche Implementierung des angegebenen Pseudocodes.

Wir orientieren uns an folgendem Pseudocode, den wir schrittweise entwickeln:

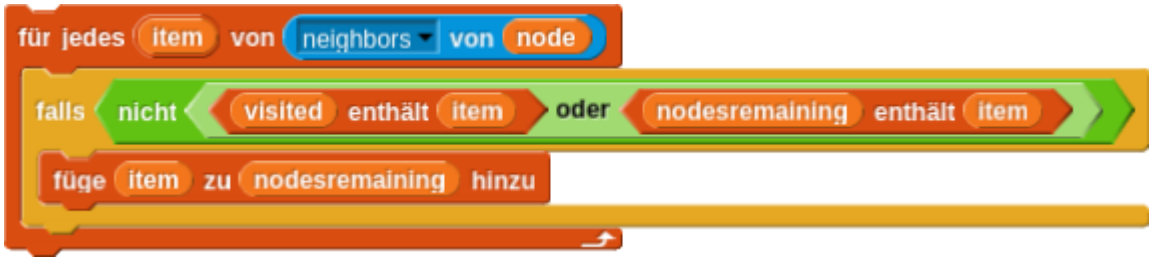
Breitensuche (Startknoten, Knoten_rest, Knoten_besucht):

Der aktuelle Knoten soll zu den besuchten Knoten hinzugefügt werden. In diesem Schritt soll er auch gleich farblich etwas hervorgehoben werden, damit wir auf der Leinwand leichter beobachten können, an welchem Knoten der Algorithmus gerade arbeitet.

füge Startknoten zu Knoten_besucht hinzu
hebe Startknoten farblich hervor

Die noch zu besuchenden Knoten werden in einer Liste verwaltet. Zu dieser Liste sollen nun die Nachbarknoten des übergebenen Startknotens hinzugefügt werden - aber natürlich nur dann, falls sie nicht bereits besucht wurden **oder** nicht bereits Teil dieser Liste sind.

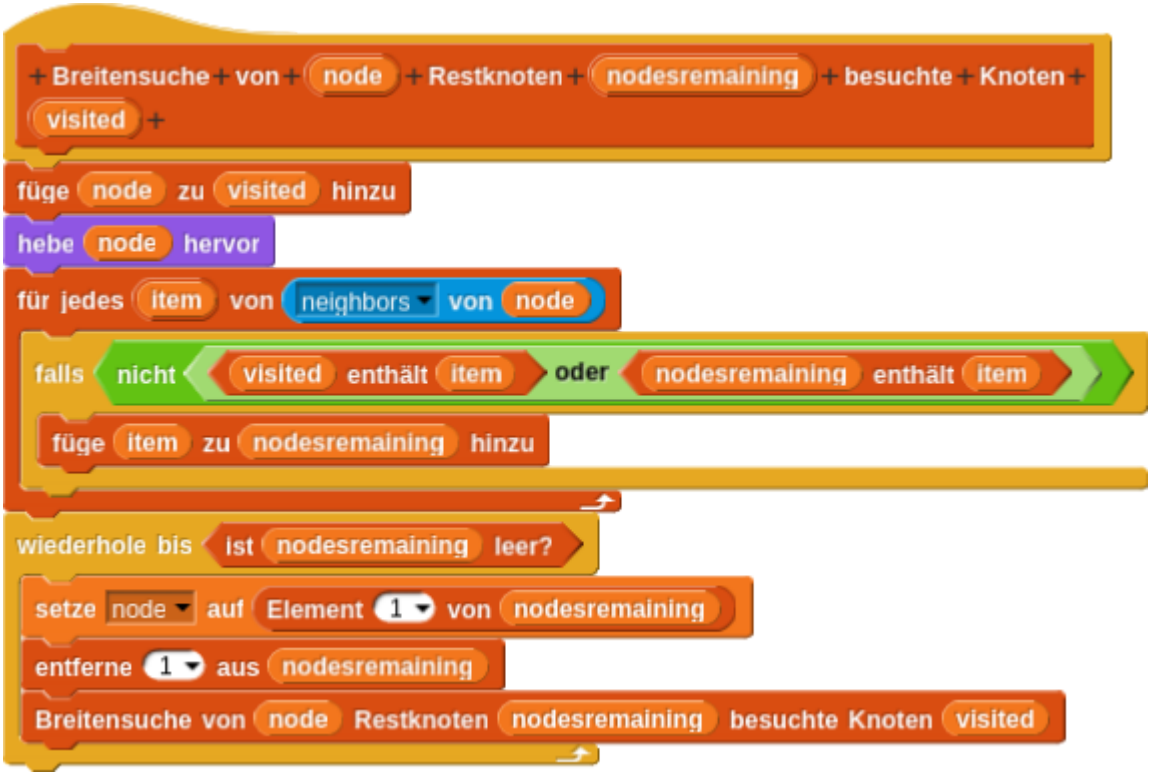
für jeden Nachbar von Startknoten:
wenn nicht Nachbar in Knoten_rest oder Nachbar in Knoten_besucht:
füge Nachbar zu Knoten_rest hinzu



Damit sind wir bereits fast am Ende. Im letzten Schritt soll das **erste** Element der Restknotenliste zum neuen Startknoten werden (wir erinnern uns: Listen arbeiten nach dem **first-in-first-out-Prinzip!**). Dann soll dieses Element aus der Liste der Restknoten entfernt werden, und schließlich der gesamte Algorithmus rekursiv mit den neuen Parametern aufgerufen werden. Das wiederholen wir solange, bis es keine Knoten mehr zu besuchen gibt.

wiederhole bis Knoten_rest leer:
setze Startknoten auf Element 1 von Knoten_rest
entferne Element 1 aus Knoten_rest
Breitensuche (Startknoten, Knoten_rest, Knoten_besucht)

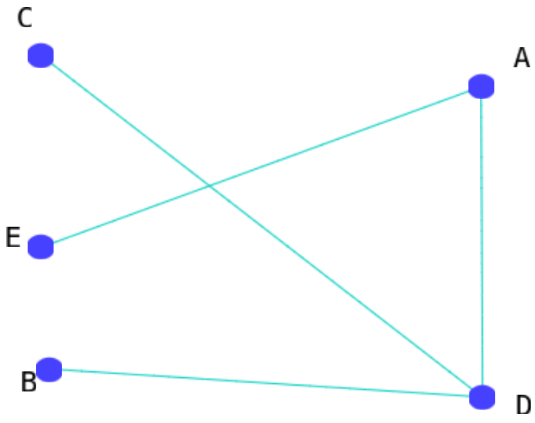
Der gesamte Block sieht somit folgendermaßen aus:



Es ist hilfreich, sich kurz vor Augen zu führen, wie sich die Parameter zwischen den Aufrufen verändert haben.

Die Eingabeparameter sind beim ersten Durchlauf, also beim ersten Aufruf des Blocks noch trivial. Als Startknoten wählen wir hier **A**, die Liste der noch zu besuchenden Knoten ist **leer**, und **dasselbe** gilt für die Liste der bereits besuchten Knoten:

Wir gehen von folgendem Beispielgraphen aus:



Am Ende des ersten Durchlaufs des Blocks ist der neue Startknoten z. B. **D.**

Die Liste der Restknoten beinhaltet **B** und **E.**

Die Liste der besuchten Knoten beinhaltet **A.**

	**Tinkering:** Testen Sie Ihren Block mit verschiedenen Eingaben!

Als besonders gutes Beispiel, um die Breitensuche zu testen und zu visualisieren, eignet sich der Graph, der vom Block 

erzeugt wird.
Wir erinnern uns: diesen Block finden wir im Objekt “node”.

	**Tinkering:** Erzeugen sie mithilfe des  -Blocks einen Binärbaum und testen Sie ihre Breitensuche an diesem. Vergleichen Sie erwartetes Verhalten und tatsächliches Verhalten. Ist die Reihenfolge, in der die Knoten besucht werden, korrekt?

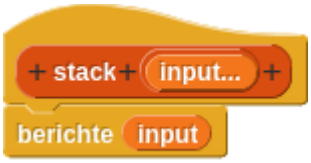
Tiefensuche

Im nächsten Schritt wollen wir uns nun der Tiefensuche zuwenden. Der zentrale Unterschied zwischen der Tiefensuche und der Breitensuche liegt darin, welcher Knoten als nächstes besucht wird. Während die Breitensuche zur Verwaltung der Restknoten eine Warteschlange/Liste verwendet, stützt sich die Tiefensuche auf einen **Stapel.**

Wir wiederholen zunächst den Stapel, indem wir ihn in Snap! implementieren.

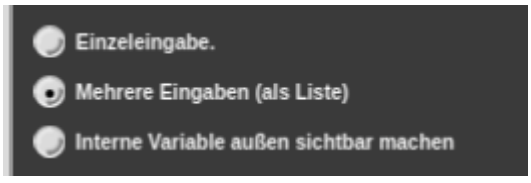
	**Aufgabe:** Implementieren Sie die folgenden Blöcke in Snap!. Achten Sie dabei besonders auf: ● korrekte **Blockformen**, bzw. **Arten** (“Puzzle-förmig”, vs. “runde Kanten”) ● sinnvolle, sprechende Benennungen der Variablen, bzw. Eingabeparameter

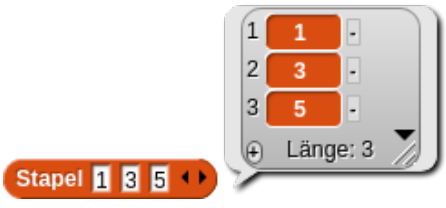
Zunächst benötigen wir einen Block, der einen leeren Stapel erzeugt und ihn zurückliefert.



Der Block ist denkbar simpel. Es gibt jedoch eine Feinheit zu beachten.

Für den **input** wurde hier folgende Option gesetzt:



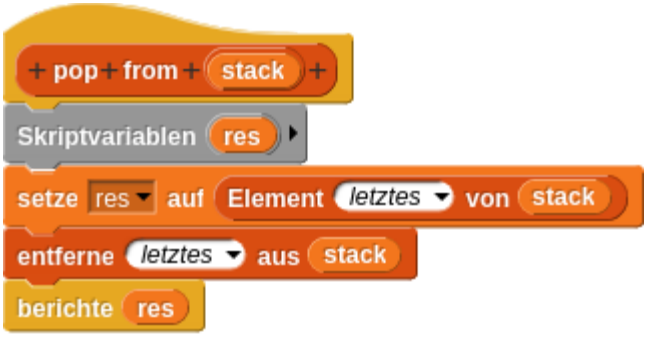


Anders als bei Listen gibt uns Snap! leider keine native Unterstützung für Stapel. Daher müssen wir auch die drei wichtigen **Stapeloperationen** selbst implementieren.

1) Peek (manchmal auch als **top** bezeichnet) liefert das oberste Element des Stapels zurück. Da ein Stapel nach dem **last-in-first-out-Prinzip** arbeitet, müssen wir somit das **letzte** Element zurückliefern.

2) Push fügt dem Stapel ein neues Element hinzu. Elemente werden wie bei einer **Liste** stets **am Ende** eines Stapels angefügt.

3) Pop liefert uns das oberste Element des Stapels zurück. Der Unterschied zu **peek** besteht darin, dass **pop** das Element auch aus dem Stapel **entfernt**!



🛠	**Tinkering:** Testen Sie Ihren Stapel mit verschiedenen Eingaben und Operationen!

Nun sind wir in der Lage, eine Tiefensuche in Snap! zu implementieren und anhand diverser Graphen zu visualisieren.

Aufgabe: Erzeugen Sie einen **neuen Block**. Der Block soll:

- ein Befehlsblock sein
- einen passenden (sprechenden) Namen tragen
- **drei** Eingabeparameter haben: einen Startknoten, **einen Stapel** der noch zu besuchenden Knoten, und eine Liste der bisher besuchten Knoten
- einer passenden Kategorie angehören (beispielsweise Listen)

Mögliche Lösung: 

Nun bearbeiten wir den **Inhalt des Blocks.** Die Bilder sollen dabei keine Musterlösung darstellen, sondern eine mögliche Implementierung des angegebenen Pseudocodes.

Wir orientieren uns an folgendem Pseudocode, den wir schrittweise entwickeln:

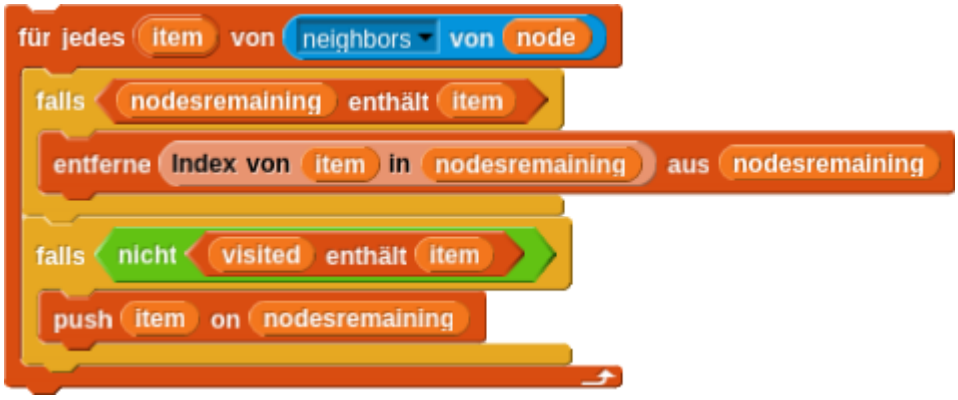
Tiefensuche (Startknoten, Knoten_rest, Knoten_besucht):

Der aktuelle Knoten soll zu den besuchten Knoten hinzugefügt werden. In diesem Schritt soll er auch gleich farblich etwas hervorgehoben werden, damit wir auf der Leinwand leichter beobachten können, an welchem Knoten der Algorithmus gerade arbeitet.

füge Startknoten zu Knoten_besucht hinzu
hebe Startknoten farblich hervor

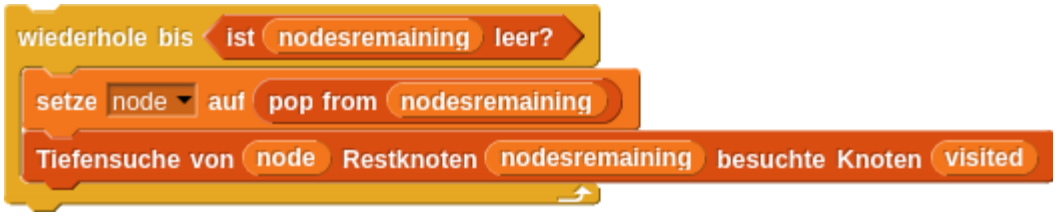
Die noch zu besuchenden Knoten werden in **einem Stapel** verwaltet. Hier zeigt sich der zentrale Unterschied zur **Breitensuche**. Die Nachbarn des aktuellen Knotens sollen auf den Stapel gelegt werden - aber nur, falls sie noch nicht bereits besucht wurden. Außerdem muss vorher überprüft werden, ob der Nachbar bereits in Knoten_rest liegt; **ist das der Fall, so muss er entfernt werden und neu auf den Stapel gelegt werden.**

für jeden Nachbar von Startknoten:
falls der Nachbar bereits in Knoten_rest liegt:
entferne ihn aus Knoten_rest
falls der Nachbar noch nicht besucht wurde:
lege den Knoten auf den Stapel Knoten_rest

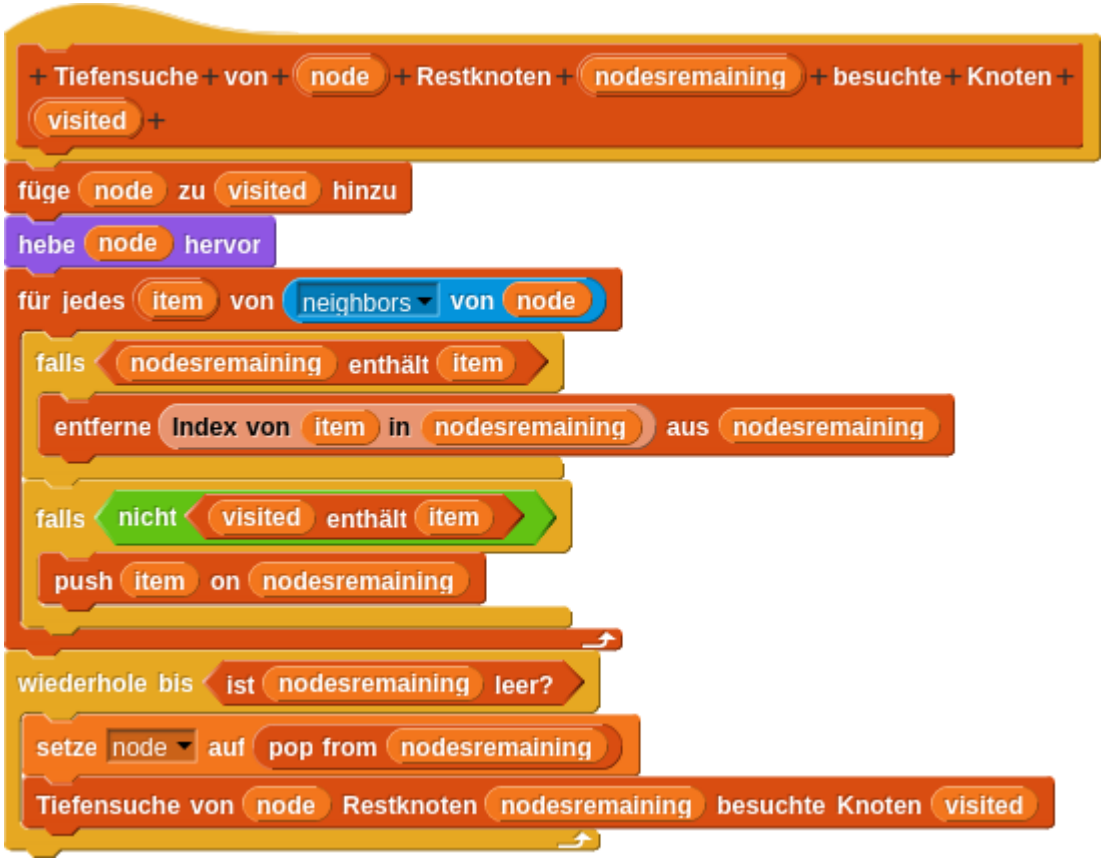


Damit sind wir bereits fast am Ende. Im letzten Schritt soll das **letzte** Element des **Restknotenstapels** zum neuen Startknoten werden (wir erinnern uns: Stapel arbeiten nach dem **last-in-first-out-Prinzip!**). Dann soll schließlich der gesamte Algorithmus rekursiv mit den neuen Parametern aufgerufen werden. Das wiederholen wir solange, bis es keine Knoten mehr zu besuchen gibt.

wiederhole bis Knoten_rest leer:
setze Startknoten auf pop(Knoten_rest)
Tiefensuche (Startknoten, Knoten_rest, Knoten_besucht)



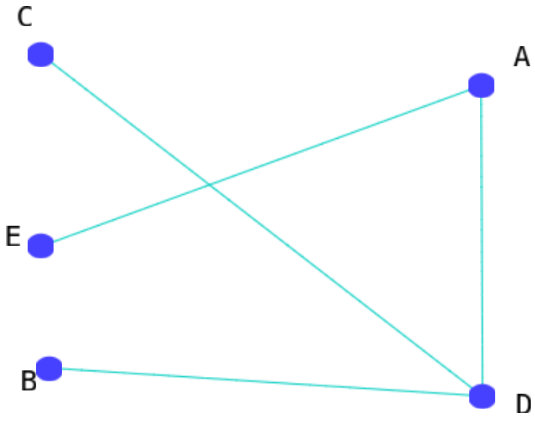
Der gesamte Block sieht somit folgendermaßen aus:



Wieder ist es hilfreich, sich kurz vor Augen zu führen, wie sich die Parameter zwischen den Aufrufen verändert haben.

Die Eingabeparameter sind beim ersten Durchlauf, also beim ersten Aufruf des Blocks noch trivial. Als Startknoten wählen wir hier **A**, die Liste der noch zu besuchenden Knoten ist **leer**, und **dasselbe** gilt für die Liste der bereits besuchten Knoten:

Wir gehen von folgendem Beispielgraphen aus:



Am Ende des ersten Durchlaufs des Blocks ist der neue Startknoten z. B. **D.**

Die Liste der Restknoten beinhaltet **B** und **C.**

Die Liste der besuchten Knoten beinhaltet **A.**

	Tinkering: Testen Sie Ihren Block mit verschiedenen Eingaben!

Als besonders gutes Beispiel, um die Tiefensuche zu testen und zu visualisieren, eignet sich der Graph, der vom Block 

erzeugt wird.
Wir erinnern uns: diesen Block finden wir im Objekt “node”.

	Tinkering: Erzeugen sie mithilfe des  -Blocks einen Binärbaum und testen Sie ihre Tiefensuche an diesem. Vergleichen Sie erwartetes Verhalten und tatsächliches Verhalten. Ist die Reihenfolge, in der die Knoten besucht werden, korrekt?

Für einen Blick in das **fertige Projekt** inklusive aller **Musterlösungen**, klicken Sie hier.
(http://tiny.cc/yy7ubz)

Reflektion

Was haben wir in diesem Abschnitt konzeptionell getan?
Wir haben ein Framework kennen gelernt, in dem es möglich ist, rekursive Datenstrukturen wie Graphen und Bäume in Snap! zu visualisieren und zu verwenden. Mithilfe dieses Frameworks ist es möglich, Eigenschaften und Begrifflichkeiten von Graphen zu erläutern und zu diskutieren.

Desweiteren haben wir jeweils einen Block zur Breitensuche und zur Tiefensuche implementiert. Für letzteres war es notwendig, zunächst eine eigene Datenstruktur, den **Stapel**, in Snap! zu erstellen.

Die einzige Einschränkung des Frameworks ist, dass (bislang) **nur ungerichtete Graphen** umsetzbar sind. Abgesehen von dieser Beschränkung ist jeder erdenkliche Schulinhalt mithilfe dieses Frameworks abdeckbar.

Bonusaufgabe:
Implementieren Sie den **Algorithmus von Dijkstra** mithilfe des Frameworks!