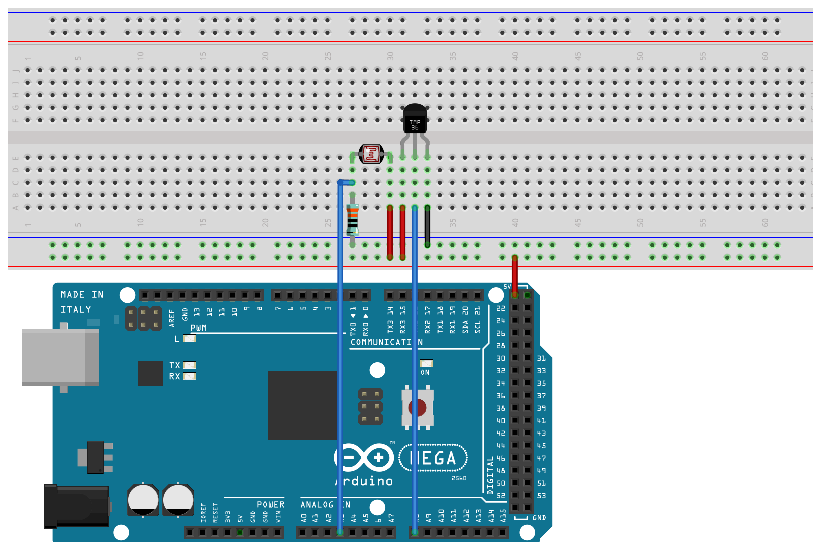
Abgabe

Geben Sie die Lösung bis **Dienstag, 23. Juni um 14:00** als PDF-Dokument per E-Mail an Ihren Tutor ab. Die Abgabeformalitäten sind die Gleichen wie auf Blatt 1. Bitte achten Sie darauf! Wir werden Abgaben mit inkorrekten Dateiformat o.Ä. nicht bewerten.

Bitte geben Sie auch zusätzlich zum PDF die Ino-Dateien zu den Aufgabenteilen ab! Dies erleichtert erheblich die Korrektur. Benennen Sie dazu die Dateien folgendermaßen: `Ping-MATRIKELNUMMER-Loesung-BLATTNUMMER-Aufgabe-AUFGABE.ino`. Also z.B.: `Ping-1234567-Loesung-7-Aufgabe-1.ino`.

Schaltung

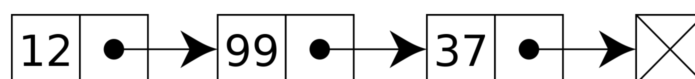
Auf diesem Blatt werden Sie zusammengesetzte Typen und abstrakte Datentypen benutzen. Bauen Sie dazu zuerst folgende Schaltung auf:



1 Messreihe

Sie sollen mal wieder eine Messreihe implementieren. Diesmal ist die Anzahl der Messpunkte nicht im Vorhinein bekannt. Um schnell neue Messpunkte aufnehmen zu können, sollen Sie daher eine verkettete Liste verwenden.

Bauen Sie daher zunächst ein `struct`, das die zu messenden Daten enthält. Erweitern Sie danach das `struct` um einen *Zeiger* auf das `struct` selbst, um somit die Messpunkte verketten zu können. Dieser Zeiger zeigt immer auf das nächste Element in der Liste - also den davor gemessenen Messpunkt. Wenn kein Nachfolger existiert, ist dieser Zeiger `NULL`. Folgene Abbildung illustriert solch eine verkettete Liste:



Schließlich brauchen Sie nur noch einen *Zeiger* auf den *Kopf* der Liste. Mithilfe des *Kopf-Zeigers* können Sie nun durch die Liste durchiterieren, indem Sie sich entlang der *Zeiger* auf das nächste Element hangeln. Zum Hinzufügen in die Liste, müssen sie nur etwas Speicher für ein neues Element reservieren, das Folgelement des aktuellen Elementes auf den alten Kopf setzen und das aktuelle Element als neuen Kopf deklarieren. Betrachten Sie dazu folgendes Beispiel:

```
struct list_node {
  int x;
  struct list_node *next;
};

struct list_node *head = NULL;

void setup() {
}

void loop() {
  struct list_node *n;

  // Einfuegen eines Elements an den Anfang der Liste
  n = malloc(sizeof(struct list_node));
  n->x = 42; // einzufuegendes Element
  n->next = head;
  head = n;

  // Gesamte Liste ausgeben
  for (n = head; n != NULL; n = n->next) {
    Serial.println(n->x);
  }
}
```

Implementieren Sie nun eine Messreihe mithilfe einer verketteten Liste:

1. Legen Sie jede Sekunde einen Messpunkt an.
2. Ein Messpunkt besteht aus Temperatur, Helligkeit und Uhrzeit.
3. Speichern Sie die Temperatur in $°C$ (siehe Blatt 6 für Umrechnung), die Helligkeit in % (lineare Abbildung von $[0, 1023] \to [0, 100]$), und die Zeit in ms seit Programmstart.
4. Speichern Sie alle Messpunkte in einer verketteten Liste.
5. **Bonus:** Wenn die Liste eine Größe von 100 Elementen erreicht, soll das älteste Element aus der Liste entfernt werden. Vergessen Sie nicht, den Speicher des Listenelements wieder freizugeben!
6. **Bonus:** Mit einer einfach verketteten Liste müssen Sie zum Löschen des letzten Elements durch die ganze Liste navigieren. Implementieren Sie eine doppelt-verkettete Liste um dieses Problem zu umgehen.

2 Binärer Suchbaum

Ein Suchbaum ist eine dynamische Datenstruktur, die eine Speicherung und effiziente Suche von Daten erlaubt. Die Größe der gesamten Datenstruktur ist im Vorhinein nicht bestimmt.

Ein Suchbaum besteht, wie eine verkettete Liste, aus Knoten, die die zu speichernde Werte enthalten. Bei einem binären Baum enthält jeder Knoten jedoch zwei *Zeiger* die auf ein *linkes* und ein *rechtes Kind* verweisen. Falls ein *Kind* nicht existiert, wird der entsprechende *Zeiger* auf `NULL` gesetzt.

Um nun effizient in diesem Baum suchen zu können, wird er nach einem *Schlüssel* geordnet. Ein *Schlüssel* ist ein Wert mit dem die *Knoten* im Baum nachgeschlagen werden können. Beinhaltet der Baum z.B. eine Menge von Studenten, so könnte man diese per Matrikelnummer nachschlagen. Somit wäre die Matrikelnummer der *Schlüssel*. Ein binärer Suchbaum wird so geordnet, dass das linke Kind eines Knotens immer einen kleineren *Schlüssel* hat als der Knoten selbst und so dass das rechte Kind einen größeren *Schlüssel* hat.

Dieses `struct` würde einen Baum aufbauen, der einfache Zahlen beinhaltet:

```
struct Node {
  int value;
  struct Node *left;
  struct Node *right;
};
```

Um einen Wert zu suchen, fängt man beim *Wurzelknoten* an. Hat der Knoten den gesuchten Wert, ist die Suche abgeschlossen. Ist der gesuchte Wert kleiner als der Wert im Knoten, wird die Suche im linken Teilbaum fortgesetzt, ansonstem im Rechten. Existieren die Teilbäume nicht (der jeweilige Zeiger ist `NULL`), wurde kein Wert gefunden. Folgende Funktion sucht *iterativ* einen Wert in einem Baum.

```
struct Node *find_node(struct Node *tree, int x) {
  while (tree != NULL) {
    if (x < tree->value) {
      tree = tree->left;
    }
    else if (x > tree->value) {
      tree = tree->right;
    }
    else { // x == tree->value;
      return tree;
    }
  }
}
```

Ihre Aufgabe ist es, den so implementierten Suchbaum zu nutzen, um eine *Studentendatenbank* zu implementieren. Studenten sollen nach ihrer Matrikelnummer gesucht werden; weitere Daten sollen im jeweiligen Knoten abgelegt werden.

1. Für jeden Studenten soll neben Matrikelnummer noch Name (80 Zeichen) und Semesteranzahl (Ganzzahl ≥ 0) gespeichert werden. Ergänzen Sie die obige Definition von `struct Node`.

2. Schreiben Sie eine Funktion `student_name()`, die für eine gegebene Matrikelnummer den Namen des Studenten zurück gibt. Existiert die Matrikelnummer nicht im Suchbaum, geben Sie stattdessen `NULL` zurück. Hinweis: Nutzen Sie `find_node()`.

3. **Bonus:** Geben Sie eine *rekursive* Implementierung von `find_node()` an, die ohne Schleifen auskommt.

4. **Bonus:** Bauen Sie die unten angegebene Funktion `insert_node()` auf Ihre Implementierung um und befüllen Sie den Baum mit Testdaten. Bauen Sie danach **sinnvolle** Testaufrufe in `loop()` ein.

5. **Bonus:** Sehen Sie ein Problem mit der unten aufgeführten Implementierung von `insert_node`?

```
void insert_node(struct Node *tree, struct Node *node) {
  if (node->value < tree->value) {
    if (tree->left == NULL) {
      tree->left = value;
    }
    else {
      insert_node(tree->left, node);
    }
  }
  else if (node->value > tree->value) {
    if (tree->right == NULL) {
      tree->right = value;
    }
    else {
      insert_node(tree->right, node);
    }
  }
  else {
    // Wert existiert schon, wird nicht ueberschrieben.
  }
}
```