# Linked lists

Algorithms and data structures ID1021

Johan Montelius

Fall term 2022

## Introduction

So far you have only been working with primitive data structures and arrays. More complicated structures are better described as structures that are linked to each other using *references*, also called *pointers* or *links*. You have probably used this method in a regular Java program where one object could have a property that is referring to another object. The reference is one-way so the object that has the property of course knows that it is referring to another object but the other object is unaware.

As an example you can create a class that describes a person. The person will of course have a name, an address etc but it could also have a father and a mother. These properties could then be references to other objects rather than strings with the name of the parents. A person could of course also have an array of children where each child is a reference to another person object.

In this assignment we will look at simple linked structures and find out their properties rather than actually represent anything. Think about it as your first assignment but now using linked structures.

## a linked list

The simplest linked structures is a *linked list*. Each element in the list will have some properties but it will always have a reference to the next element in the list (sometimes called the tail). If this reference is a *null-pointer* the item is the last item in the list.

A simple linked list class that holds an integer could look like follows:

```
public class LinkedLints {
  int head;
  LinkedList tail;
```

```
    public LinkedList(int item, LinkedList list) {
      head = item;
      tail = list;
    }

    :
    :
  }
```

The basic methods that we provide are returning the first item in the list, i.e. the head element in the list, and returning the tail of the list.

```
  public int head() {
    return this.head;
  }

  public Linkedlist tail() {
    return this.tail;
  }
```

Another method that we can provide is to *append* a list to the end of a list. We do this by moving to the last element in the linked list and making it point to the argument list.

```
  public void append(LinkedList b) {
    LinkedList nxt = this;
    while (nxt.tail != null) {
      nxt = nxt.tail;
    }
    nxt.tail = b;
  }
```

## benchmarks

Your first task is to set up a benchmark that gives us an idea of the running time of the append operation. You should vary the size of the first linked list (`a`) and append it to a fixed size linked list (`b`). We're not interested in the exact run time but only how the run time changes with growing length of list `a`.

You should then switch the benchmark around so that you have the length of `a` fixed and increase the length of `b`. Explain you findings, why does it look like it does?

Your second task is to benchmark a linked list against the equivalent operation using and array. If we have two arrays the append operation would be to first allocate a new array that is large enough to hold all items and then copy the values from both arrays to the new array. The time to do this should then of course include allocating the new array. How does the array append operation vary with the size of the first and the second array?

When we compare these two data structures it is interesting to see how the cost of allocation differs. Change your benchmark of the linked list so that it shows the cost of building a list of $n$ items. How does the cost of building the list compares to the time to allocate an array of the same size?

## a stack

In one of your previous assignments you implemented a dynamic stack that would change size as you pushed and popped items. Now using your implementation of a linked list, implement a stack data structure with the regular push and pop operations.

Without doing any measurements, describe the difference in execution time for the array implementation and the linked list implementation.