Linked lists

in C

Algorithms and data structures ID1021

Johan Montelius

Spring 2026

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 C we do not have any objects as in Java but we have something similar to collect a number of values and this construct is called a `struct`. A struct is a data structure that has a number of named elements. Each element is a data structure that could be a primitive value, an array, a struct or a pointer to another data structures.

In the following assignment we will create a simple data structure that holds a value and a reference to another data structure. It's the simplest linked data structures that you can imagine, yet it will be very different from the array structures that you have been working with so far.

a linked list

The simplest linked structures is a *linked list*. A linked list will hold a sequence of *cells* but only have access to the first cell in the sequence. Each

cell in the sequence will have some property but also have a reference to the next cell in the sequence (sometimes called the tail). If this reference is a *null-pointer* the cell is the last item in the list.

A simple linked list class that holds a sequence of integer could look like follows:

```
typedef struct cell {
  int value;
  struct cell *tail;
} cell;

typedef struct linked {
  cell *first;
} linked;
```

In C we most likely would like to have a procedure that allocates a linked list on the heap and initialize the properties. Each new cell that is added to the list will also be allocated on the heap so when it is time to deallocate the structure wee need to return all cells.

```
linked *linked_create() {
  linked *new = (linked*)malloc(sizeof(linked));
  new->first = NULL;
  return new;
}

void linked_free(linked *lnk) {
  cell *nxt = lnk->first;
  while (nxt != NULL) {
    cell *tmp = nxt->tail;
    free(nxt);
    nxt = tmp;
  }
  free(lnk);
}
```

We can now add methods to for example add another integer to the beginning of the list or finding the n'th integer in the list etc. Implement the following methods:

- `void linked_add(linked *lnk, int item)` : add the item as the first cell in the list.
- `int linked_length(linked *lnk)` : return the length of the list.

- `boolean linked_find(linked *lnk, int item)` : return true or false depending on if the item can be found in the list.
- `void linked_remove(linked *lnk, int item)` : remove the item if it exists in the list (and also free the cell).

Remember that when you add a new integer you need to allocate a new cell structure on the heap. If you have not started to think about what goes on the stack and what goes on the heap it is now high time realize that C is very different from Java.

```
void linked_add(linked *lnk, int item) {
  cell *new = (cell*)malloc(sizeof(cell));
   :
   :
}
```

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

```
void linked_append(linked *a, linked *b) {
  cell *nxt = a->first;
  cell *prv = NULL;

  while(nxt != NULL) {
    prv = nxt;
    nxt = nxt->tail;
  }
  if (prv != NULL)
    :
  else
    :
  :
}
```

When you have found the last cell in the first list you set its `tail` reference to the first cell of the second list. You should also (maybe, you decide) set the second list to have an empty (`null`) reference as its first cell. We might get very confused if a list of cells appear in two different lists but it's up to you.

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` i.e. the big-O complexity.

To generate a linked lists of length n, one could do something like this:

```
linked *linked_init(int n) {
  linked *a = linked_create();
  for (int i = 0; i < n; i++) {
    linked_add(a, i);
  }
  return a;
}
```

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

compared to an array

So far you have been working with arrays that behave quite different from a linked list. How would you implement the append function if you had used arrays instead of a linked list? Explain in your own words what the implementation would look like and what this would mean for the benchmark.

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, how would you implement a stack data structure with the regular push and pop operations? Use your own words and describe the pros and cons.

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