Computer Programming

Implementing an abstract datatype. Linked lists and queues

Marius Minea

marius@cs.upt.ro

4 January 2016

Libraries and abstract datatypes

Use of (standard) library so far:

we know a *function prototype* (declaration), e.g.

```
FILE *fopen(const char *fname, const char *mode);
```

declaration is included from *header file* `#include <stdio.h>`

we do *not know or need the source* code for `fopen`

only the *object (binary) code* which is part of the library

last compile stage *links* program with the library

Program is *independent* of underlying details

(Unix/Windows? file system type?)

implementation of library function can *change*

(new compiler version, bug fix, new file system)

as long as *interface* (function prototype) stays the same

Abstract datatypes

An abstract datatype is a mathematical model for datastructures
defined by the operations applicable to them (*functions*)
and the constraints among them (*axioms*)
without exposing details about the implementation.

ADTs *separate interface from implementation*
the interface provides the *abstraction*
the implementation is *encapsulated* (hidden)

ADTs allow changeable and interchangeable implementations
client program relies only on interface, is not affected

`FILE` is an abstract datatype in the standard C library
don't know implementation detail
can only access with given functions (`fopen`, `fgets`, `fread`, etc.)

Lists as abstract data types

An ADT list L with elementtype E is usually defined by:

$nil : () \rightarrow L$	empty list constructor
	can also be constant rather than function
$isempty : L \rightarrow Bool$	is empty ?
$cons : E \times L \rightarrow L$	list constructor
$head : L \rightarrow E$	head of list
$tail : L \rightarrow L$	tail of list

and the *axioms*

$$head(cons(e, l)) = e \quad \text{and} \quad tail(cons(e, l)) = l$$

Example ADT for integer list

```c
#ifndef _INTLIST_H
#define _INTLIST_H

typedef struct ilst *intlist_t;

intlist_t empty(void);
int isempty(intlist_t lst);
int head(intlist_t lst);
intlist_t tail(intlist_t lst);
intlist_t cons(int el, intlist_t tl);

// for freeing memory only: splits first element from tail,
// if elp non-NULL, store value of head there
intlist_t decons(intlist_t lst, int *elp);

#endif
```

Hiding / exposing the representation

Implementation is hidden if only a *pointer* to the data is exposed:
  incomplete structure type: `typedef struct ilst *intlist_t`
  or even a `void *` (only implementation knows what it points to)

Declaration of structure should be hidden in `.c` file
not exposed in `.h` file (which is included by all clients)

```
struct ilst {
  intlist_t nxt;
  int el;
};
```

If library client has this structure, can use internal representation
(no longer an ADT)

Implementing the list ADT

```c
#include <stdlib.h> // for NULL and malloc
#include "intlist.h" // ensures .h and .c consistent

struct ilst {
  intlist_t nxt;
  int el;
};

intlist_t empty(void) { return NULL; }

int isempty(intlist_t lst) { return lst == NULL; }

int head(intlist_t lst) { return lst->el; }

intlist_t tail(intlist_t lst) { return lst->nxt; }
```

Implementing the list ADT (cont'd)

```
intlist_t cons(int el, intlist_t tl)
{
  intlist_t p = malloc(sizeof(struct ilst));
  if (!p) return NULL; // could report some error
  p->el = el;
  p->nxt = tl;
  return p;
}

 // returns tail, assignes *elp with head, deletes cell
intlist_t decons(intlist_t lst, int *elp)
{
  if (elp) *elp = lst->el;
  intlist_t tl = lst->nxt;
  free(lst); // just first cell, keeps rest
  return tl;
}
```

Can we do lists of arbitrary types?

C does not have polymorphism or parametric types
⇒ cannot declare, e.g., list of *arbitrary type*

Could do: `typedef int elemtype;` (or even a `#define`)
and have everything else use `elemtype`

But need to *recompile* everything when changing `elemtype`
binary code differs even for assignment/parameter passing
due to varying element size; even more so for addition, etc.)

If instead of values we store *pointers* to values,
we can have just one implementation (list of `void *`)
must separately allocate memory for elements
program logic must know element type (info not in the list)

Example: list reversal in-place

Assume: we know declaration

```
struct ilst {
  intlist_t nxt;
  int el;
};
```

Two pointers, splitting list:

one to part of list already reversed (initially `NULL`)
one to rest of list to be reversed (initially full list)

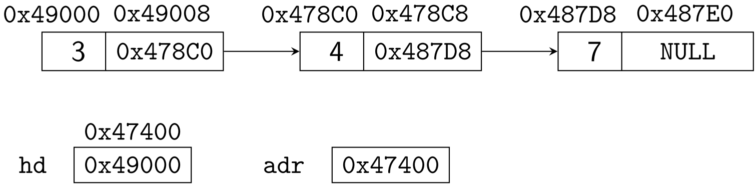
```
intlist_t rev2(intlist_t rest, intlist_t done) {
  if (isempty(rest)) return done;
  intlist_t nxt = rest->nxt; // rest to be reversed
  rest->nxt = done; // link first cell to done part
  return rev2(nxt, rest); // tail-recursive, becomes loop
}
intlist_t rev(intlist_t lst) { return rev2(lst, empty()); }
```

Traversing linked list with address of pointer

When inserting/deleting into a linked list (e.g. *ordered* list),
must change link in cell *prior* to the one inserted/deleted

- keep *address* of pointer to be changed (address of link field)
- better than with address of previous element (may not exist)

In picture, top row denotes *addresses* of individual fields




```
intlist_t hd = cons(3, cons(4, cons(7, NULL)));
intlist_t *adr = &hd;
adr = &(*adr)->nxt; // advance to next element
```

Implementing a queue ADT

Queue: first-in, first-out (FIFO): insert/remove at different ends

```
#ifndef _QUEUE_H
#define _QUEUE_H

typedef struct q *queue_t;

queue_t q_new(void);
int q_isempty(queue_t q);
int q_get(queue_t q);
queue_t q_put(queue_t q, int el);
void q_del(queue_t q);
void q_print(queue_t q);

#endif
```

Implementing a queue

To uniformly handle case when queue becomes empty/grows again we use a *dummy* cell (flag); actual first cell is *after* the dummy cell

```
typedef struct e { // cell for element, with pointer to next
  struct e *nxt;
  int el;
} elem_t;

struct q {
  elem_t *hd;   // dummy; actual first cell is next
  elem_t *last; // last cell (or dummy if empty)
};

queue_t q_new(void) {
  queue_t q = malloc(sizeof(struct q));
  elem_t *p = malloc(sizeof(elem_t)); // dummy cell
  p->nxt = NULL;        // no actual element
  q->hd = q->last = p; // initially both dummy cell
  return q;
}
```