Computer Programming

Implementing an abstract datatype. Linked lists and queues

Marius Minea

marius@cs.upt.ro

19 December 2016

Review: compilation basics

Briefly: Compiler translates *source code* to *executable code*.

First step: produce *object code*: `gcc -c file.c` → `file.o`
has binary (executable) code for all functions
contains *symbols* (names) of functions/variables *defined* in source
and *referenced* symbols (e.g. library functions) defined elsewhere

Can also produce *assembly*: `gcc -S file.c` → `file.s`
(human-readable version of executable code)

Second step: *link* object files together (and with standard library)
`gcc file1.o file2.o ...` → `a.out`
resolves (links) symbols used in one module and defined in another

More *load-time* linking done by operating system at program start
(for dynamic libraries)
one memory copy of library can be shared by many programs

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:

$nil : () \to L$	empty list constructor
	can also be constant rather than function
$isempty : L \to Bool$	is empty ?
$cons : E \times L \to L$	constructor: new list from element and rest
$head : L \to E$	first element
$tail : L \to L$	*list* with all elements after head

and the *axioms* linking these functions

$head(cons(e, ?)) = e$ and $tail(cons(?, l)) = l$

can be seen as definition of *cons*

$isempty(nil()) = true$, $isempty(cons(?, ?)) = false$

head, *tail* undefined for list which *isempty*

Example ADT for integer list

```
#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

If header file declares (exposes) only a *pointer* type to the data, implementation is *hidden*

incomplete structure type: `typedef struct ilst *intlist_t`

or a `void *` (but dangerous: no type safety)

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

```
#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, assigns *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()); }
```

Remember: pointers are for indirection

A pointer `p` allows *indirect* access to a value: `*p`:
the *value* of *variable* `p` is an *address*
we can use the *value* `*p` found at *address* `p`
(either read or write)

Useful for communicating between program parts:
have an address `p`
other functions that have `p` can change `*p`
by reading `*p` always have latest value

Analogy:
URL (address) vs. web page contents (value, may be updated)

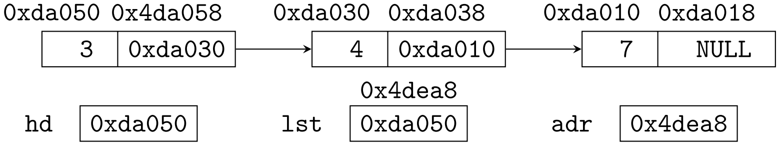
Traversing linked list with address of pointer

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)

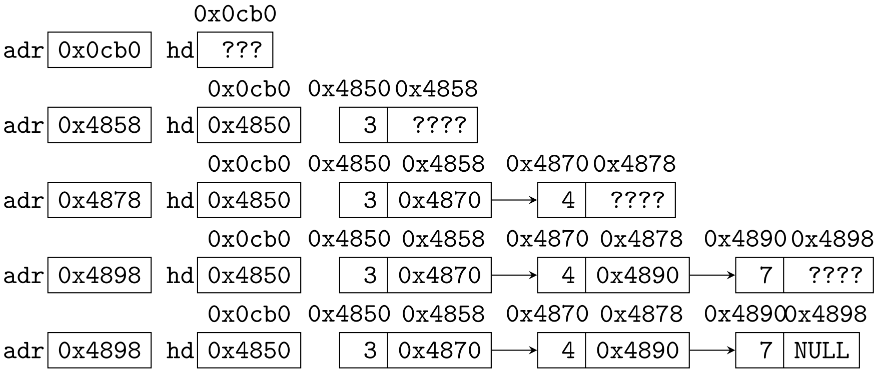
```
intlist_t hd = cons(3, cons(4, cons(7, NULL))); // in main
void trav_addr(intlist_t lst) {
  for (intlist_t *adr = &lst; *adr; adr = &(*adr)->nxt)
    printf("adr: %p, *adr: %p\n", adr, *adr);
} // might print:
adr: 0x4dea8, *adr: 0xda050
adr: 0xda058, *adr: 0xda030
adr: 0xda038, *adr: 0xda010
```

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



Creating a list using addresses of pointers

```
intlist_t rdlist(void) {     // read ints and place in list
  intlist_t hd, *adr = &hd;  // address where t<o link next cell
  for (int n; scanf("%d", &n) == 1; adr = &(*adr)->nxt)
    (*adr = malloc(sizeof(*hd)))->el = n; // malloc and set elem
  *adr = NULL; // done, set link to next cell to NULL
  return hd;   // value from first cycle or NULL above if empty
}
```



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

Use a *dummy* cell before actual first element; each `get` deletes it, next cell becomes dummy. Invariant: empty queue has `hd==last`.

```
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));
  q->hd = q->last = malloc(sizeof(elem_t)); // both dummy cell
  q->hd->nxt = NULL;                        // no actual element
  return q;
}
```