Computer Science 2530
March 23, 2020

Happy Monday, March 23.

With this lecture we are beginning to look at data structures beyond the basic ones of arrays and structures. This lecture covers the material in the notes for 3/23, pages **30A** to **30D**, on **abstract data types**, **lists** and **linked lists**.

1. Encapsulations (30A)

One of the guiding principles of software design is the need to make software easy to modify. That involves creating *encapsulations*, so that modifications that are made inside an encapsulation do not propagate out to the rest of the program. Encapsulations that we have seen include the following.

- **Named constants.** By writing

  ```
  const int maxNameLength = 50;
  ```

  you have encapsulated a piece of information, that the maximum number of characters that are allowed in a name is 50. The rest of the program refers to maxNameLength, not to 50. Changing the maximum name length involves changing just one line of code.

- **Functions.** Definition

  ```
  double distance(double x1, double y1, double x2, double y2)
  {
    double deltaX = x2 - x1;
    double deltaY = y2 - y1;
    return sqrt(deltaX * deltaX + deltaY * deltaY);
  }
  ```

  encapsulates an algorithm to compute the distance between points ($x1$, $y1$) and ($x2$, $y2$) in the plane. No other part of the program needs to know how to compute distances. If you write the distance formula directly in several places, chances are good you will make an error in some of those places, obviously not a good idea. Also, if you need to change how 'distance' works, you only need to change the body of this one function.

- **Modules.** A module is a collection of definitions, possibly including type definitions, function definitions and constant definitions. The module has an *interface* that tells exactly what this module exports to other modules. Any change that does not affect the interface can remain entirely within a single module.

Abstract data types: interface

Now we look at a special kind of module, called an *abstract data type*, or ADT for short. The interface of an ADT contains

- a type T;
- a collection of functions that work on or are related to objects of type T;
- possibly constants of type T.

For example, for assignment 4, you created a module that is an ADT. Its interface contains

- a type, **ER**;
- functions **newER**, **destroyER**, **equivalent** and **combine**.

Other modules can use type ER and the functions in the interface, and that is all.

It is conventional to refer to an ADT by the name of the type that it defines. So your solution to assignment 4 is called **ER**.

Abstract data types: implementation

The *implementation* of an ADT T consists of

- a choice of representation of the information, which is given as the details of the definition of type T;
- definitions of the functions that are part of the interface, as computer code that performs them.

For example, the interface of ADT **ER** contains the following.

- The first part of the implementation is the idea that an equivalence relation is represented by an array of integers. If equivalence relation R is represented by array r, then we think of $r[x]$ as the *boss* of x.

  We could have used a representation where $r[x]$ is the leader of x. Initially, that seems more natural. But doing a 'combine' operationscan potentially change the leaders of many different values, and that makes it expensive to do. The boss idea has the advantage that, to combine two equivalence classes, you only need to make one change to the array of bosses.

  As a rule, selecting a way to represent information is the most crucial step in implementing an abstract data type. Everything will follow from that.

- The second part of the implementation is a collection of (C++) definitions of the functions that are described in the interface. But an essential part of that is a definition of another function, **leader**, that converts from boss information to leader information. The leader function is private to the ER module.

The representation of type T is never part of the interface of ADT T. There must be nothing in your solution to assignment 5, **mst.cpp**, that makes direct use of the fact that an object of type **ER** is an array of integers. Instead, **mst.cpp** just uses **ER**.

2. Conceptual lists (30B)

A *list* is a sequence of values. For simplicity, we will restrict attention to lists of integers.

There is a convenient *conceptual notation* for writing lists: write the list's members in square brackets, separated by commas. Here are some lists in that notation.

- [1, 2, 3, 4, 5, 6]
- [2, 2, 3, 3, 4, 4]
- [2, 4, 6]
- [1, 9, 5, 6]
- [5, 2]

- [3]
- [ ]

List [3] is a list that contains exactly one number, 3. List [ ] is called the *empty list*; it has no members.

Important. Conceptual list notation is not part of C++. It is a notation that we use to help understand lists.

Head and tail

Functions **head** and **tail** are part of the list ADT interface that allow you to get the parts of a list.

- **head**(L) is the first value in list L. For example,
  - head([2, 4, 6, 8]) = 2
  - head([3, 3, 3]) = 3
  - head([5, 3, 1]) = 5
  - head([4,7]) = 4
  - head([4]) = 4
  - head([ ]) is undefined. The empty list has no head.
- **tail**(L) is the list that you get from list L by removing the first value. For example,
  - tail([2, 4, 6, 8]) = [4, 6, 8]
  - tail([3, 3, 3]) = [3, 3]
  - tail([5, 3, 1]) = [3, 1]
  - tail([4,7]) = [7]
  - tail([4]) = [ ]
  - tail([ ]) is undefined. The empty list has no tail.

You can use head and tail to get any member of a list. For example,

$$\begin{aligned}
\text{head}([2,4,6,8]) &= 2 \\
\text{head}(\text{tail}([2,4,6,8])) &= \text{head}([4,6,8]) \\
&= 4 \\
\text{head}(\text{tail}(\text{tail}([2,4,6,8]))) &= \text{head}(\text{tail}([4,6,8])) \\
&= \text{head}([6,8]) \\
&= 6
\end{aligned}$$

The : operator

Head and tail allow us to get the members of a list that has already been built. Clearly, we also need a way to build up a list. For that, we use conceptual binary operator **:**. There is no **:** operator in C++, but we can use it for thinking about lists.

Definition of $h\mathbin{:}t$
$h\mathbin{:}t$ is the list whose head is h and whose tail is t.

For example,

- $2\mathbin{:}[4,\ 6] = [2,\ 4,\ 6]$
- $5\mathbin{:}[1,\ 2,\ 5] = [5,\ 1,\ 2,\ 5]$
- $3\mathbin{:}[3] = [3,\ 3]$
- $2\mathbin{:}[4] = [2,\ 4]$
- $8\mathbin{:}[\,] = [8]$.

Notice that the left-hand operand of **:** is *always* an integer and the right-hand operand of **:** is *always* a list.
By convention, operator **:** is done from right to left. So

$$\begin{aligned} 1\mathbin{:}2\mathbin{:}[3] &= 1\mathbin{:}(2\mathbin{:}[3]) \\ &= 1\mathbin{:}[2,3] \\ &= [1,2,3] \end{aligned}$$

Notice that

$$\begin{aligned} 6\mathbin{:}[\,] &= [6] \\ 4\mathbin{:}6\mathbin{:}[\,] &= [4,6] \\ 2\mathbin{:}4\mathbin{:}6\mathbin{:}[\,] &= [2,4,6] \end{aligned}$$

Notice the [ ] at the end.

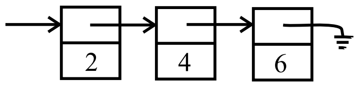
Do exercises

Read section **30B**.

Do all of the exercises at the end of page **30B**.

3. Linked lists (30C)

We need a way to represent lists, and that leads us to the idea of a *linked list*. Here is a picture of a linked list that represents conceptual list [2, 4, 6].



Each rectangle is a structure of type **ListCell**. Notice the null pointer at the end.

```
  struct ListCell
  {
    ListCell* tail;
    int       head;

  ListCell(int h, ListCell* t)
  {
    head = h;
    tail = t;
  }
};
```

A list is actually a pointer to a list cell. Type definition

```
  typedef ListCell* List;
```

says that List means the same thing as ListCell*.

Do exercises

Read page **30C**. Do the exercises at the bottom of the page.

4. Implementation of Linked Lists

Additional components of the list ADT interface

C++ does not allow us to define a binary operator called **:**. It also does not allow conceptual list notation. Here are some additions to the list ADT that get around those issues.

- **isEmpty**(L) is true if list L is empty. For example,

  - isEmpty([2, 4, 6, 8]) = false
  - isEmpty([ ]) = true

- **emptyList** is an empty list. For example,

  ```
  List e = emptyList;
  ```

  creates variable e and sets it to an empty list.

- **cons(**h**,** t**)** is C++ notation for $h:t$. For example,

  ```
  List four    = cons(4, emptyList);
  List twofour = cons(2, cons(4, emptyList));
  ```

  makes four be list [4] and makes twofour be list [2, 4].

Implementation of the list functions and constant

Once we have decided on a representation (linked lists), the function definitions fall out without much thought.

```
const ListCell* emptyList = NULL;

int head(const ListCell* L)
{
  return L->head;
}

ListCell* tail(ListCell* L)
{
  return L->tail;
}

bool isEmpty(const ListCell* L)
{
  return L == emptyList;
}

ListCell* cons(int h, ListCell* t)
{
  return new ListCell(h, t);
}
```

Do exercises

Read page **30D**. Do the exercises at the bottom of the page.