Abstract Programming – an overview.

In the DSA (Data Structures & Algorithms) course, the start point for abstract programming is the sequence (as an example). By abstract programming I mean that much of the implementation detail is hidden. This usually means the details of whether **arrays** or **structures and pointers** have been used for the implementation.

Sequence Definition

Sequence	::= **Head Tail** | empty	// (::= means is defined as)
Head	::= element	
Tail	::= **Sequence**	// recursive definition

What does this imply?

1. A sequence is either **empty** or **non-empty**
2. A **non-empty** sequence is composed of a **head** and a **tail**
3. To **deconstruct** the sequence, 2 functions are required
   a. **Head(sequence S)**
   b. **Tail(sequence S)**
4. To **reconstruct** the sequence, a further function **cons(head(S), tail(S))** is required where S is a sequence

This in turn determines how the code will be written – e.g. add an element

```
static sequence be_add_val(sequence S, int v)  {

  return is_empty(S)                ? create_e(v)

   :    v < get_value(head(S))      ? cons(create_e(v), S)

   :                                  cons(head(S), be_add_val( tail(S),v) );

}
```

create_e(int v) creates a new element from a value

The abstract form of the above example is

```
1. Check for an empty sequence     // is_empty(S)
2. Process the Head                // non-empty non-recursive
3. Process the Tail                // non-empty recursive
```

This provides an abstract programming pattern for the remaining operations on the sequence.

Cardinality (size)

```
static int be_size_seq(sequence S)

{ return is_empty(S) ? 0 : 1 + be_size_seq(tail(S)); }
```

OR using if/then/else

```
static int be_size_seq(sequence S)

{ if (is_empty(S)) return 0 else return 1 + be_size_seq(tail(S)); }
```

Non-empty implies that the sequence contains **at least** one element

Remove

```
static sequence be_rem_val(sequence S, int v) {

  return is_empty(S)               ? S

   :    v == get_value(head(S))    ? tail(S)

   :                                 cons(head(S), be_rem_val(tail(S), v));

}
```

Garbage collection is assumed!

Find

Here there is a **choice**. Find can either return **true or false** OR if the element is found, **a reference** to the element otherwise a null reference (however this is implemented – the reference is an abstract type)

```
static sequence be_find_val(sequence S, int v) {
return (is_empty(S) || (v==get_value(head(S)))) ? S : be_find_val(tail(S), v);
}
```

This is a **shortened version** of the pattern – the "standard version" is

```
static sequence be_find_val(sequence S, int v) {
return      is_empty(S)              ? S                  // S is NULLREF (not found)
       :    v==get_value(head(S))    ? S                  // found
       :                               be_find_val(tail(S), v);
}
```

The Boolean version is

```
static int be_find_val(sequence S, int v) {
return      is_empty(S)              ? 0                       // false
       :    v==get_value(head(S))    ? 1                       // true
       :                               be_find_val(tail(S), v);
}
```

NOTE in all of these functions, **sequence** is an (abstract) reference to the sequence.

The **abstract implementation** then becomes

```
#define NULLREF NULL                    // the abstract null reference (pointer)
```

```
typedef struct seqelem * sequence;      // abstract reference to an element
typedef struct seqelem {                // element – structure + pointer
        int             value;
        sequence        tail;           // abstract sequence reference
        } seqelem;
```

This may be further abstracted by defining an **abstract type** for the value

```
typedef int   valuetype                 // here the type is an integer
```

giving

```
typedef struct seqelem * sequence;      // abstract reference to an element
typedef struct seqelem {                // element – structure + pointer
        valuetype       value;          // abstract value type
        sequence        tail;           // abstract sequence reference
        } seqelem;
```

In the rest of the code only NULLREF, valuetype and sequence are used.

S==NULL, pointers (→), and value type int should NOT be used

This allows the value type to be easily redefined to another type by changing the typedef to `typedef xxx valuetype` – where **xxx** is any defined type.

The next step is to define the get/set functions on the sequence element. These will be implementation dependent in the code BUT the get/set functions represent the **abstraction** (hiding) of the implementation details.

There are 2 attributes (i) **value** and (ii) a **reference** to a sequence

The get/set functions are

```
static valuetype    get_value(sequence E)                   { return E->value;       }
static sequence     get_tail (sequence E)                   { return E->tail;        }
static sequence     set_value(sequence E, valuetype v)      { E->value = v; return E; }
static sequence     set_tail (sequence E, sequence t)       { E->tail  = t; return E; }
```

Finally there is one more implementation dependent function

sequence create_e(valuetype v)   // create an element from a value

```
static sequence create_e(valuetype v)
{
  return set_tail(set_value(malloc(sizeof(sequence)), v), NULLREF);
}
```

Here, the **malloc** reveals that the implementation is based on structures and pointers. For an array based implementation, the create_e function would get the next free element in the array.

After this there is a further abstraction based on the definition of the sequence

The de-construction functions	**Head**(sequence S), **Tail**(sequence S)
The re-construction function	**cons**(sequence **Head**, sequence **Tail**)

```
static sequence Head(sequence S)                    { return S; }
static sequence Tail(sequence S)                    { return get_tail(S); }
static sequence cons(sequence H, sequence T)        { return set_tail(H, T); }
```

and finally define the is-empty function

```
static int    is_empty(sequence S)                  { return S == NULLREF; }
```

After this point only these functions should be used to manipulate the sequence. See the code for the sequence operations (add, rem, find) above.

By defining and naming these functions, the code is self-documenting.

Head, **Tail** and **cons** show how the sequence is de- and re-constructed in the operations.

Signs of **non-abstract programming** are NOT using the abstract functions and revealing the implementation details. For example add

```
static sequence be_add_val(sequence S, int v)  {
  return is_empty(S)              ? create_e(v)
   :    v < get_value(head(S))    ? cons(create_e(v), S)
   :                                cons(head(S), be_add_val( tail(S),v) );
}
```

Might be written as

```
static sequence be_add_val(sequence S, int v)  {
  return S==NULL                  ? create_e(v)
   :    v < get_value(S)          ? cons(create_e(v), S)
   :                                cons(S, be_add_val( S→tail, v) );
}
```

Where **S==NULL, S** and **S→tail** reveal the implementation details (structure + pointer) and the function is **less well documented** in terms of the **ABSTRACT DEFINITION** of the sequence and possibly less well **understood** by the programmer!!!

If you are using the second form (**non-abstract**) in your labs please change the code to the **abstract** form.

This is a **mental exercise** in abstract programming since both the sequence and the binary tree are particularly good examples of **how the definition determines and constrains the code**.

While the student may think (right now) that this is perhaps a “strange” way of programming, it comes from the functional programming paradigm. The above function written in the programming language **Haskell** becomes

```
bAdd v [ ]              = v: [ ]            // [ ] is the empty sequence
bAdd v [x:xs]                               // :    is cons
| v < x                 = v : [x:xs]        // x    is head
| otherwise             = x : bAdd v xs     // xs   is tail
```

Where the sequence is defined as a **cons(Head, Tail)** in a very shortened form

[x : xs] where x is the **head**, xs is the **tail**, : is the **cons** function and [ ] the **empty** list. [x : xs] can be read as “a list is a head ‘**cons**:ed’ to a tail”.

Signs of abstract programming

1. The implementation details (array/structure + pointer) are hidden (abstracted) by
   a. Defining an abstract **NULLREF**
   b. Defining an **abstract value type**
   c. Defining **get/set functions** for each attribute
   d. Defining a **create element** function
   e. Defining an **is_empty** function
   f. Defining **de-construction** functions for the ADT
      i. For a **sequence** these are **Head(S)**, **Tail(S)**
      ii. For a **binary tree** these are **LC(T)**, **node(T)**, **RC(T)**
   g. Defining a **re-construction** function for the ADT
      i. For a **sequence** this is **cons(Head(S), Tail(S))**
      ii. For a **binary tree** this is **cons(LC(T), node(T), RC(T))**
2. Functions are **short** and **perform one operation only**
3. The code is **self-documenting** and reflects the structure of the ADT
4. The code follows a **pattern** determined by the structure of the ADT
5. The code is **easy to understand** by a person who understands ADTs
6. The code is **elegant**!

These ideas represent a new style of programming which is perhaps new (and strange) for you BUT it is part of this course and the understanding of ADTs.

ADT Binary Tree / Binary Search Tree / AVL tree

The next stage is to apply the above ideas to the BT (which is part of the tree lab (BST/AVL-tree) of the course). So how do we start?

1. Look at the **definition** of the BT
2. Identify the **code pattern** from the definition
3. **Apply this pattern** to the functions which implement the operations

The Definition

```
BT    ::=   LC  N  RC  | empty          // non-empty or empty
N     ::=   element                     // N is the node / root
LC    ::=   BT                          // LC is the Left Child
RC    ::=   BT                          // RC is the Right Child
```

The Pattern

1. The **empty** case
2. The **node / root** case — // non-empty, non-recursive
3. The **LC** case — // non-empty, **recursive**
4. The **RC** case — // non-empty, **recursive**

However in this case it is sometimes more advantageous to slightly change the pattern to

1. The **empty** case
2. The **LC** case — // non-empty, **recursive**
3. The **RC** case — // non-empty, **recursive**
4. The **node / root** case — // non-empty, non-recursive

The **start definitions** and functions then become

```
#define NULLREF   NULL                          // abstract null reference

typedef struct treenode * treeref;              // abstract tree reference

typedef struct treenode {                       // tree element (node)
      int           value;
      int           height;
      treeref       LC;                         // abstract tree reference
      treeref       RC;                         // abstract tree reference
      } treenode;

static treeref T    = (treeref) NULLREF;               // define a tree (empty)

static int        is_empty(treeref T)          { return T == NULLREF;  }


static int        get_value(treeref T)         { return T->value;        }
static int        get_height(treeref T)        { return T->height;       }
static treeref    get_LC(treeref T)            { return T->LC;           }
static treeref    get_RC(treeref T)            { return T->RC;           }

static treeref    set_value(treeref T,  int v)     { T->value  = v; return T;  }
static treeref    set_height(treeref T, int h)     { T->height = h; return T; }
static treeref    set_LC(treeref T, treeref L)     { T->LC     = L; return T;   }
static treeref    set_RC(treeref T, treeref R)     { T->RC     = R; return T;   }
static treeref create_node(int v)
{
  return set_RC(
            set_LC(
              set_height(
                set_value(malloc(sizeof(treenode)), v),
              0),
            NULLREF),
          NULLREF);
}
```

```
static treeref node(treeref T)          { return T; }
static treeref LC(treeref T)            { return get_LC(T); }
static treeref RC(treeref T)            { return get_RC(T); }
```

```
static treeref cons(treeref LC, treeref N, treeref RC) {
  return  set_LC(set_RC(N, RC), LC);
   }
```

LC(T), node(T) and **RC(T)** are the **deconstruction** functions for the tree

cons(LC, node, RC) is the **reconstruction** function for the tree

The **operations** on the tree are then **implemented** according to the **pattern** and **abstractions** defined above. Add, find, cardinality and remove.

```
static treeref b_add(treeref T, int v)          // no duplicate values
{
 return is_empty(T)                ? create_node(v)
   : v < get_value(node(T))        ? cons(b_add(LC(T), v), node(T), RC(T))
   : v > get_value(node(T))        ? cons(LC(T), node(T), b_add(RC(T), v))
   :                                 T;
}
```

```
static int b_findb(treeref T, int v)
{
  return is_empty(T)                ? 0                     // false
        : v <  get_value(node(T))  ? b_findb(LC(T), v)
        : v >  get_value(node(T))  ? b_findb(RC(T), v)
        :                             1;                    // true
}
```

```
static int b_size(treeref T)
{
  return is_empty(T) ? 0 : 1 + b_size(LC(T)) + b_size(RC(T));
  }
```

```
static treeref b_rem(treeref T, int v)
{
   return is_empty(T)            ? T
   : v <  get_value(node(T))     ? cons(b_rem(LC(T), v), node(T), RC(T))
   : v >  get_value(node(T))     ? cons(LC(T), node(T), b_rem(RC(T), v))
   :                               removeAtRoot(T);
}
```

Note that **b_rem** maintains the **pattern** defined above. Another sign of abstract programming is to separate functionality and to divide problems up into smaller sub-problems. Note the new sub-goal is to implement **removeAtRoot**.

There are 4 cases:

```
static  treeref removeAtRoot(treeref T) {
 return  is_LeafNode(T)          ? NULLREF          // no LC, no RC
     :     is_empty(LC(T))       ? RC(T)            // RC only
     :     is_empty(RC(T))       ? LC(T)            // LC only
     :                             twoChild(T);     // LC and RC
  }
```

So the next sub-goal is to write **twoChild**() (and **is_LeafNode**() – used to document the code).

In the case where the root has both a LC and a RC then there are 2 possibilities

1. Replace the value at the root by the maximum value in the LC
2. Replace the value at the root by the minimum value in the RC

The minimum/maximum values must be removed from the RC/LC.

One way of proceeding is --- twoChild(T) thus requires a decision

1. height(LC(T)) > height(RC(T)) ➔ use maximum value in LC
2. otherwise ➔ use minimum value in RC

Then **cons(LC, node, RC)** may then be used to return the resultant tree

This in turn might lead to further help functions which will be mirror images.

Again the **definition** determines the **pattern** and thus how the code is written.

In summary

Signs of abstract programming

1. The implementation details (array/structure + pointer) are hidden (abstracted) by
   a. Defining an abstract **NULLREF**
   b. Defining an **abstract value type**
   c. Defining **get/set functions** for each attribute
   d. Defining a **create element** function
   e. Defining an **is_empty** function
   f. Defining **de-construction** functions for the ADT
      i. For a sequence these are **Head(S)**, **Tail(S)**
      ii. For a binary tree these are **LC(T)**, **node(T)**, **RC(T)**
   g. Defining a **re-construction** function for the ADT
      i. For a sequence this is **cons(Head(S), Tail(S))**
      ii. For a binary tree this is **cons(LC(T), node(T), RC(T))**
2. Functions are **short** and **perform one operation only**
3. The code is **self-documenting** and reflects the structure of the ADT
4. The code follows a **pattern** determined by the structure of the ADT
5. The code is **easy to understand** by a person who understands ADTs
6. The code is **elegant**!

These ideas represent a new style of programming which is perhaps new (and strange) for you BUT it is part of this course and the understanding of ADTs.