DSA: Programming

“Cheap Tricks” for the
(abstract) programmer!

Programming

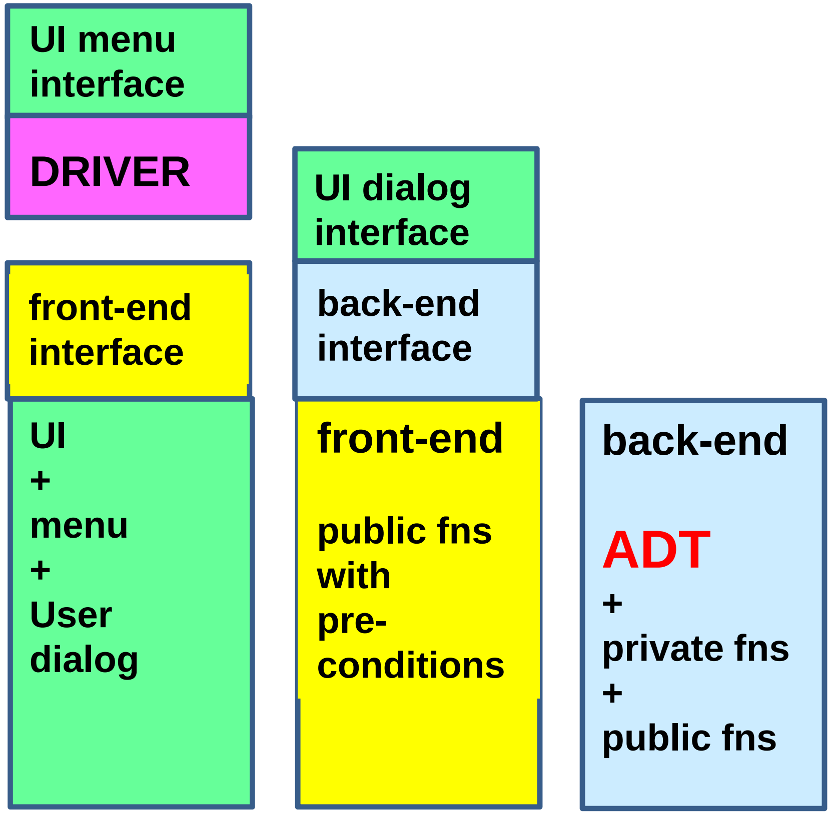
- Towards a more **abstract style**
- **Functions** should be **simple**
  - **A small number of steps & 1 operation only**
  - **10-20 lines is a “big” function**
- Rules of Thumb (guidelines)
  - A start point for your own development
- Learn **the art of re-factoring**

UI Front-end Back-end model

A rapid prototyping testbed

UI (User Interface) + Front-end / Back-end Model

- “cheap” prototyping
- may be used
  - interactively
    - for demonstrations
  - with scripts
    - for test runs
  - with Linux scripts
    - build & test system



UI menu interface

DRIVER

front-end interface

UI + menu + User dialog

UI dialog interface

back-end interface

front-end

public fns with pre-conditions

back-end

ADT

+ private fns + public fns

The Sequence lab exercise

- The back-end represents an **empty list**
- Step 1: implement display for the empty list case – print “**list is empty**” count 0
- Step 2: implement the **navigation functions***
- Step 3: Add a value to the (sorted) list
-
```
listref create_element(valtype v) {        //create & init new element
      newref = make_new_element();         //allows for a counter or function
      set_value(newref, v);
      set_prev(newref, NULLREF);
      set_next(newref, NULLREF);
      return newref;
      }
```

* iterative version only

The Sequence lab exercise

- Step 4: implement display for the non-empty list case – **list (4) is: 1 2 5 8**
- Step 5: implement find
- Step 6: implement remove
- Note the similarities in add/find/remove
- Step 7: implement add_, find_, rem_pos
- Step 8: test stack & queue modes

Points to note

- **A similar development sequence applies for the tree and graph labs**
    - **empty + count (0) + display (empty) + add + count (non 0) + display (non-empty) + find + remove**
- The implementation is hidden in
    - **set / get / create_element functions**
    - **type valtype, listref & NULLREF**
- the remaining functions use these abstractions
- arrays ➔ struct & pointers requires little change

C examples

- See the C workshop pages

http://www.cs.kau.se/cs/education/courses/C_workshop/

- Check out “recipes” for example
- Check out the example programs

“Abstract” programming

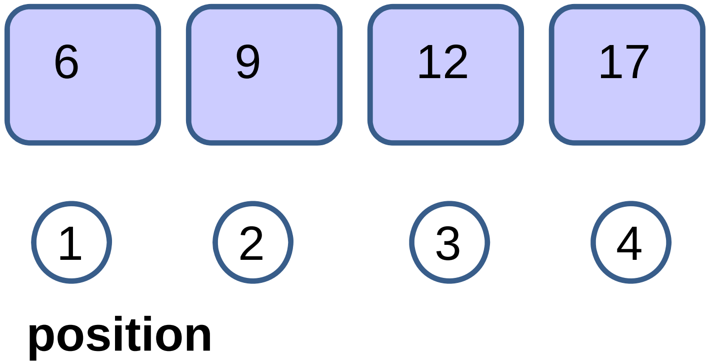
Abstracting (“hiding”) the implementation details

Array or structure + pointers

The abstract sequence (list)

- **what the user sees**
  - **6, 9, 12, 17**
- **properties**
  - **There are n elements**
  - **Every element has a position & value**
  - **Every element except the last has a successor**
  - **Every element except the first has a predecessor**
  - **The sequence may be empty (size == 0)**

ADT Visualisation

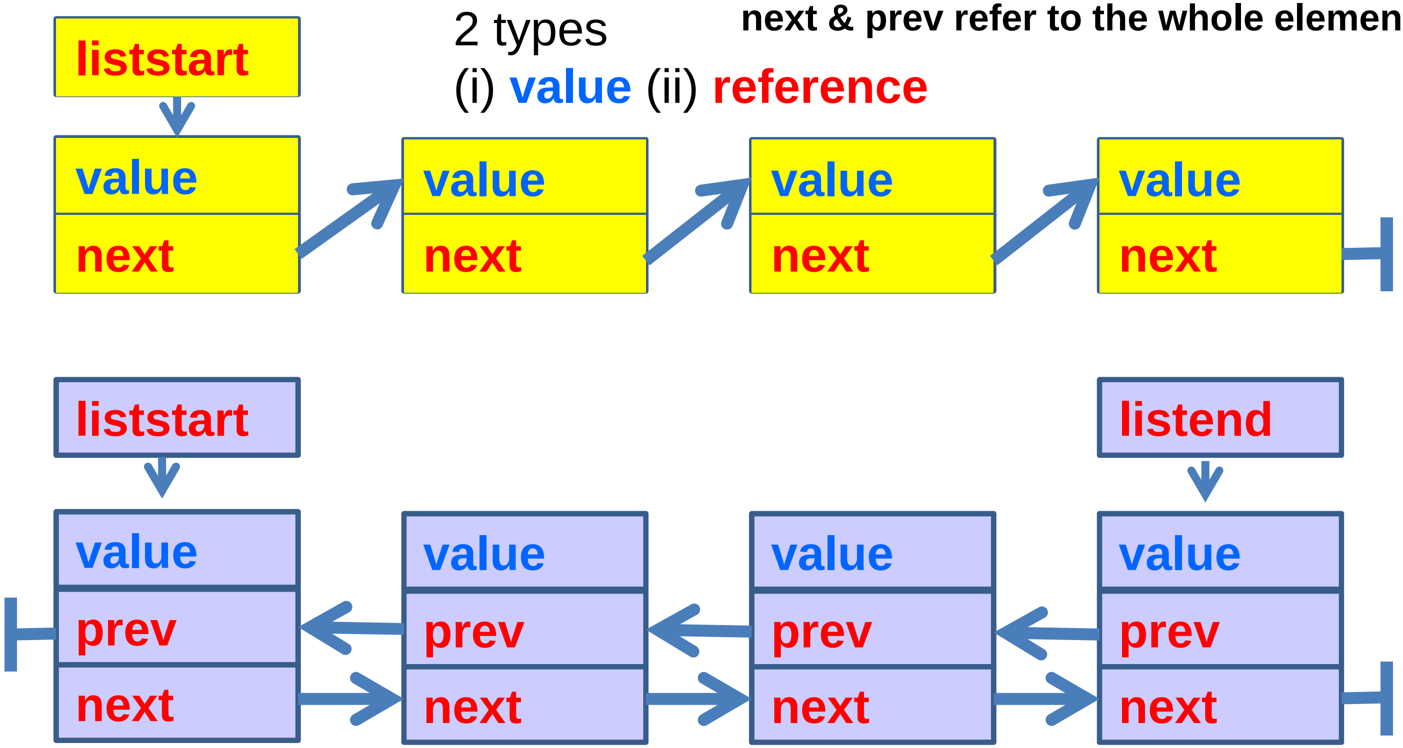


The “abstract” implementation(s)

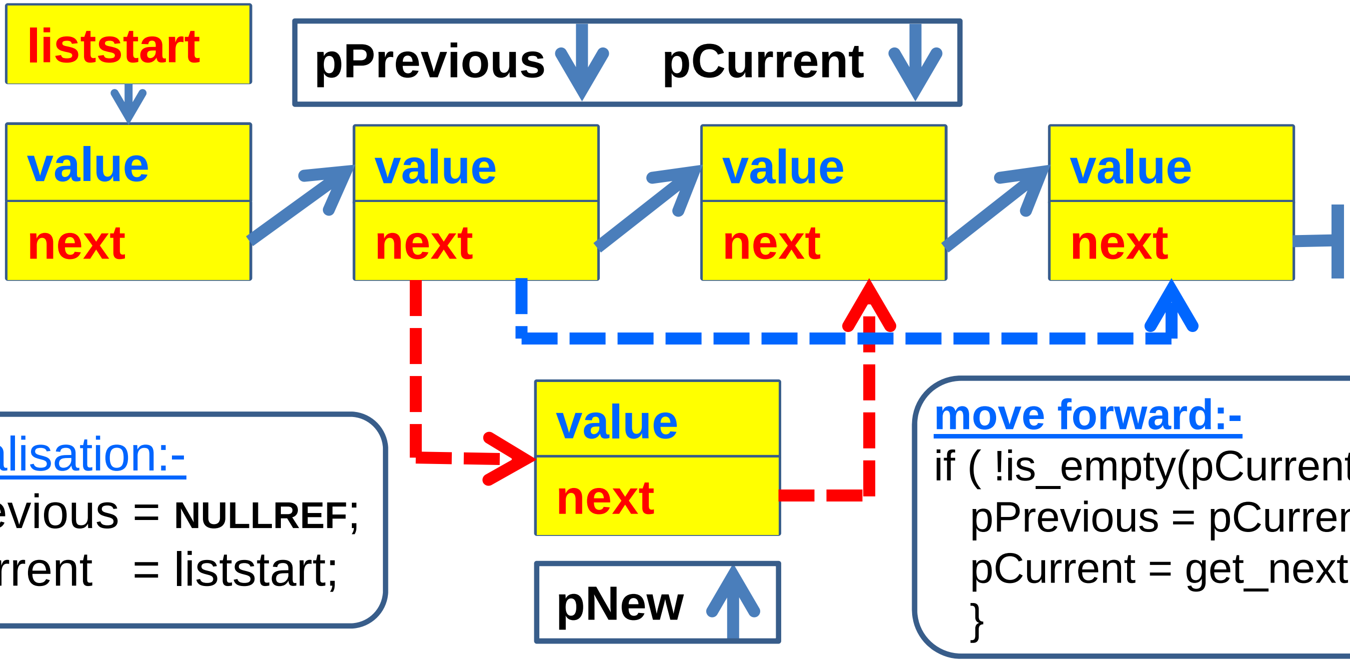
2 types

(i) value (ii) reference

next & prev refer to the whole element



The role of pPrevious, pCurrent, pNew



Initialisation:-

```
pPrevious = NULLREF;
pCurrent   = liststart;
```

move forward:-

```
if ( !is_empty(pCurrent)  ) {
   pPrevious = pCurrent;
   pCurrent = get_next(pCurrent) ;
   }
```

(pPrevious, pCurrent) move as a pair along the list (used in add /find/ remove)

pNew is inserted between pPrevious and pCurrent (used in add)

The “real” implementation(s)

- array
- pointer + structure

NULLREF is a “constant” (macro); typedef (re)defines a type in C

```
#define NULLREF -1
typedef   int    valtype;
typedef   int    listref;
```

```
valtype    value[LSIZE];
listref     next[LSIZE];
listref     prev[LSIZE];
```

```
#define NULLREF NULL
typedef   int    valtype;
typedef struct listelem * listref;
```

```
typedef struct listelem
{    valtype     value;
     listref     next;
     listref     prev;
} listelem;
```

Entity attributes

- For each attribute there should be a corresponding get/set function
- E.g.
  - 3 attributes A, B, C ➔
    - **get_A(), get_B(), get_C()**
    - **set_A(), set_B(), set_C()**

Implementation details

- These are "hidden" (a wrapper!) in the **get/set** functions & **create element** function
  - Array implementation
    - **Atype get_A(Reftype Ref)** **{ return A[Ref]; }**
    - **void set_A(Reftype Ref, Atype v)** **{ A[Ref] = v; }**
  - Structure/pointer implementation
    - **Atype get_A(Reftype Ref)** **{ return Ref→A; }**
    - **void set_A(Reftype Ref, Atype v)** **{ Ref→A = v; }**
- All other functions use these **get/set functions** and the **NULLREF** constant (**-1** (array) / **NULL** (ptr+str)) plus

```
int is_empty(listref ref) { return ref == NULLREF; }
```

Sequences / lists

- These are better viewed as variable length structures – **use while - not for**
  - ➔ `while (!empty(L)) { /* process element */ }`
  - ○ elements may be added / removed at will
  - ○ There MUST be an end-of-list marker

- **for loops are NOT recommended for sequences/lists** **(for is OK in other contexts – repeat n times; n known)**

Signs of “**non-abstract**” programming

- **The following appear outside of get/set & is_empty(x)**
- **Array implementation**
  - **return a[i] or a[i] = v → use get / set**
  - **if (ref == NULLREF) { … } → if (is_empty(ref)) …**
  - **if (ref == -1) { … } → if (is_empty(ref)) …**
- **Pointer & structure implementation**
  - **return ref→a or ref→a = v → use get / set**
  - **if (ref == NULLREF) { … } → if (is_empty(ref)) …**
  - **if (ref == null) { … } → if (is_empty(ref)) …**
- **Local declarations of**
  - **int index; (array) / listelem * ref; (ptr+str) – use listref**

Clichés

Programming patterns

Clichés – programming patterns

- In many contexts the **same or similar patterns repeat themselves**
- In **re-factoring** - recognise these patterns and perhaps make them into functions
- In **coding** you have a “library” of (abstract) patterns (in your head) which you can use
- The following pages discuss some of these patterns

List operations

Iterative - clichés

Navigation functions

- Navigation functions (using pPrevious & pCurrent)

```
void get_Seq_first()   { pPrevious = NULLREF; pCurrent = liststart; }
int   is_Seq_empty()  { return is_empty(pCurrent); }
void get_Seq_next()   {
   if ( !is_Seq_empty() ) {               // ➔ pCurrent != NULLREF
      pPrevious = pCurrent;
      pCurrent   = get_next(pCurrent);
      }
   }
```

pPrevious & pCurrent have been hidden (abstracted away)

- Navigation (iteration) through the list

```
get_Seq_first();
while ( !is_Seq_empty() ) { /* process element */ get_Seq_next();   }
```

Handling sequences / lists

- **Iterative method:-**
  - **add "pNew" between pPrevious & pCurrent**

```
void be_add_val(valtype val) {

get_Seq_first();                          // navigate to correct position
  while (!is_Seq_empty() && (val > get_Element_value()) )   get_Seq_next();

 link_in(create_element(val));            // add the new element
  }

valtype    get_Element_value() { return get_value(pCurrent); }
```

“Simplicity”

- The add function has 2 parts
  1. Find position (using value or position)
  2. Create and Link in element

- be_add_val and be_add_pos can share create/link
- be_rem_val & + be_rem_pos could share “unlink element”

Handling sequences / lists

- **Iterative method:- find & remove**

```
listref be_find_val( valtype val)  {
 get_Seq_first();
 while (!is_Seq_empty() && (val != get_Element_value()) )  get_Seq_next();
 return get_Current_ref();
 }
```

get_Current_ref() returns the value of pCurrent – which is a **reference**

pCurrent is **NULLREF** (not found) or **refers to an element** (found)

```
void be_rem_val   (valtype val)    { unlink( be_find_val(val) );   }
void be_rem_pos (postype pos)  { unlink( be_find_pos(pos) ); }
```

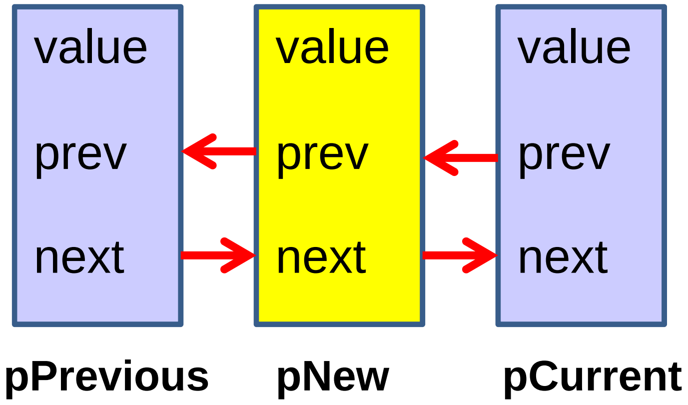
The Add operation

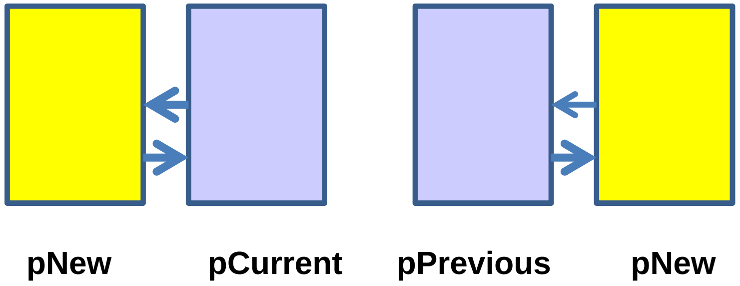
Use pictures!

Cliché: **pNew** is added between
pPrevious and **pCurrent**

link_in() - pPrevious / pNew / pCurrent

- **Draw a picture**
- **Add in the middle**
  - **set_prev(pNew, pPrevious)**
  - **set_next(pNew, pCurrent)**
  - **set_next(pPrevious, pNew)**
  - **set_prev(pCurrent,   pNew)**
- **Add at the beginning pPrevious==NULLREF**
  - **íf is_empty(pPrevious) liststart=pNew**
    else set_next(pPrevious, pNew)
- **Add at the end pCurrent==NULLREF**
  - **íf is_empty(pCurrent) listend=pNew**
    else set_prev(pCurrent, pNew)
- **Final code**
  - **set_prev(pNew, pPrevious)**
  - **set_next(pNew, pCurrent)**
  - **íf is_empty(pPrevious) liststart=pNew**
    else set_next(pPrevious, pNew)
  - **íf is_empty(pCurrent) listend=pNew**
    else set_prev(pCurrent, pNew)





The “link_in” function

```
void link_in( pNew ) {                        // singly linked list
    set_next(pNew, pCurrent);
   íf (is_empty(pPrevious)) liststart = pNew   else set_next(pPrevious, pNew);
}
void link_in( pNew ) {                        // doubly linked list
   set_prev(pNew, pPrevious);
   set_next(pNew, pCurrent);
   íf (is_empty(pPrevious)) liststart = pNew   else set_next(pPrevious, pNew);
   íf (is_empty(pCurrent))    listend  = pNew   else set_prev(pCurrent, pNew);
}
```

- pNew is always defined and is thus non-null – but check just the same! TO DO!
- pPrevious is null on insertion at the beginning - check required
- pCurrent is null on insertion at the end - check required

List operations

Recursive - clichés

Handling sequences / lists

- **Recursive** method:- traverse list

```
void RT(listref L) {
  if (!is_empty(L) ) { process( head(L) ); RT( tail(L) ); }
   return;
   }
```

Handling sequences / lists

- **Recursive** method:- add

```
static listref be_add_val(listref L, valtype v)
{
 return is_empty(L)                ?   create_e(v)
        :   v < get_value(head(L)) ?   cons(create_e(v), L)
        :                              cons(head(L), be_add_val(tail(L),v));
}
```

Pattern:-

The **empty** case	- non-recursive – **stop condition**
The **non-empty** case	- non-recursive – the **head**
The **recursive** case	- the **tail**

if versus a functional style

```
static listref be_add_val(listref L, valtype v)
{
   if is_empty(L)                          return create_e(v);
   if v < getval(head(L))                  return cons(create_e(v), L);
                                           return cons(head(L),  be_add_val(tail(L) ,v));
}
===========================================================
static listref be_add_val(listref L, valtype v)
{
  return is_empty(L)                 ?   create_e(v)
         :   v < get_value(head(L)) ?   cons(create_e(v), L)
         :                              cons(head(L), be_add_val(tail(L) ,v));
}
```

Reusing code: add_val ➔ add_pos

```
static listref be_add_val(listref L, valtype v)
{
  return is_empty(L)              ?  create_e(v)
       :  v < get_value(head(L)) ?  cons(create_e(v), L)
       :                            cons(head(L), be_add_val(tail(L) ,v));
}
==========================================================================
static listref be_add_pos(listref L, valtype v,  postype pos  )
{
  return is_empty(L)              ?  create_e(v)
       :  pos == 1                ?  cons(create_e(v), L)
       :                            cons(head(L), be_add_pos(tail(L) ,v, pos-1));
}
```

Reusing code: add_pos ➔ rem_pos

```
static listref be_add_pos(listref L, valtype v, postype pos)
{
  return is_empty(L)               ?  create_e(v)
         :  pos ==  1              ?  cons(create_e(v), L)
         :                            cons(head(L), be_add_pos(tail(L), v, pos-1));
}

=======================================================================
static listref be_rem_pos(listref L,  postype pos)
{
  return is_empty(L)               ?  L
         : pos == 1                ?  tail(L)
         :                            cons(head(L), be_rem_pos(tail(L), pos-1));
}
```

Handling sequences / lists

- Recursive method:- find value

```
static listref be_find_val(listref L , valtype v) {
  return (is_empty(L) || (v == get_value(head(L)))) ? L : be_find_val(tail(L), v);
}
```

This is a shortened version of the pattern

```
static listref be_find_val(listref L , valtype v) {
  return is_empty(L)                     ? L
       :   v == get_value(head(L)))) ? L
       :                                    be_find_val(tail(L), v);
}
```

Handling sequences / lists

- **Recursive** method:- remove value

```
static listref b_rem_val(listref L , valtype v) {

   return is_empty(L)                    ? L
          : v == get_value(head(L)) ? tail(L)
          :                                 cons(head(L), b_rem_val(tail(L), v));
}
```

Programming clichés

- The above are clichés
- **Re-factoring** would spot these patterns and optimise the code
- Similar arguments apply to other sequential structures:- files / tables

```
get_first_element();
while (!EOF) { process element(); get_next_element(); }
```

True and false

Different views

Use of “true” “false”

```
if (A == B) then return true;
else return false;
if (A == B) then return 1;
else return 0;
```

```
➔ return (A == B);
```

I even see this in programming textbooks!!!

What is the point?

The question is does A == B?

The answer is
yes/no
true/false

Boolean expression!

And…

```
listref be_find_val(valtype v)  {                              // ref to element (if found)
   get_Seq_first();
   while (!is_Seq_empty() && (v != get_element_value()) )  get_Seq_next();
   return   get_Current_ref();
   }

int be_find_val(valtype  v)  {                                 // Boolean version (T|F)
   get_Seq_first();
   while (!is_Seq_empty() && (v != get_element_value()) )  get_Seq_next();
   return   !is_empty( get_Current_ref() );
   }
```

OR – using the version **listref   be_find_val(valtype v) { … }** above

```
int be_is_member( valtype v) { return  !is_empty( be_find_val(v) );  }
```

Readability

Using space effectively

Readability – use 2 dimensions!!!

```
static void      set_value (  listref R,   valtype v  )     { list[R]   = v;     }
static void      set_next  (  listref R,   listref n  )     { next[R]   = n;     }
static void      set_prev  (  listref R,   listref p  )     { prev[R]   = p;     }

static valtype   get_value (  listref R )                   { return list[R];    }
static listref   get_next  (  listref R )                   { return next[R];  }
static listref   get_prev  (  listref R )                   { return prev[R];  }
```

Readability – use 2 dimensions!!!

```
static listref be_add_val(listref L , valtype v)
{
  return is_empty(L) ? create_e(v)
         :  v < get_value(head(L)) ? cons(create_e(v), L)
         :  cons(head(L), be_add_val(tail(L), v));
}
```

```
static listref be_add_val(listref L , valtype v)
{
  return is_empty(L)                  ?   create_e(v)
         :   v < get_value(head(L))   ?   cons(create_e(v), L)
         :                                cons(head(L), be_add_val(tail(L), v));
}
```

cons adds an element at the head of the list

Readability – use 2 dimensions!!!

```
static listref be_add_val(listref L , valtype v)
{
  return is_empty(L)                  ?   create_e(v)
        :   v < get_value(head(L))    ?   cons(create_e(v), L)
        :                                 cons(head(L), be_add_val(tail(L), v));
}
```

N.B. create_e(v) **is the same as** cons(create_e(v), L) **if L == empty**

To Haskell ☺ **simpler & simpler!**

```
bAdd v [ ]                      = v: [ ]
bAdd v [x:xs]
  | v < x                       = v : [x:xs]
  | otherwise                   = x : bAdd v xs
```

Readability – use 2 dimensions!!!

```
static treeref create_node(valtype v)
{
  return set_RC(
                  set_LC(
                          set_height(
                                       set_value(malloc(sizeof(treenode)), v),
                          0),
                  NULLREF),
           NULLREF);
}
```

OR

```
return set_RC( set_LC( set_height( set_value( malloc(sizeof(treenode)), v), 0), NULLREF), NULLREF);
```

Re-factoring re-visited

Now look at your own code…

…time to refactor!!!

As a training exercise

- **Reflect** on what has been said in the previous slides
- **NOW** go back and look at your code
- What could be **improved?**
  - i.e. made shorter
  - i.e. made more efficient
  - i.e. made more readable

Customers and consultants

Let’s play at reality!

The customer and consultant game

- **The customer wants…**
  - To see a demo of the product
  - A commentary on the different options
  - To believe that all is going well ;-)
- **The customer does NOT want…**
  - To hear your problems – good news only!
  - To see your trace and debug info

You need therefore to

- Keep a **working demonstration** at all times – you never know when the customer will want a demonstration
- Keep a **development version**
  - The words “**under development**” hide a multitude of sins
  - The current problems **will** disappear tomorrow!!!

Testing

Test at the limits!

Test in the middle!

Test at the limits

- If the prompt says
  - >enter position [1..5]:
  - **the test sequence will be 0, 1, 5, 6, 3**
- Programming with preconditions is a useful technique here
  - Test the user input BEFORE calling the backend function(s)

Test in the middle

- Case 3 above for example
- It is often easier to understand the “middle” case (e.g. insert between 2 existing values in a list)
- Reality however, demands the following order
  - Add to an **empty** list
  - Add at the **beginning** of the list (limit case)
  - Add at the **end** of the list (limit case)
  - Add between 2 elements in the list (middle case)

Testing & development sequence

- **Empty list**
  - Write & test **is_empty**
  - Write & test **cardinality** (zero)
  - Write & test **display** – empty list message
- **Non-empty list**
  - Write & test **add_val** + **display**
  - Write & test **find_val**
  - Write & test **rem_val**
  - Write & test **add_pos**, **find_pos**, **rem_pos**

Debugging

The Development Version!

Debugging

Je te crois pas!	**Montre-moi!**
I don’t believe you!	**Show me!**
Jag tror inte på dig!	**Visa mig!**

Keep in mind…

- Festina lente
  - Make haste slowly!
  - sakta, sakta!
- The 30 minute rule!
- The sleep on it rule!
- Talk to another person (a dialog)
- **Develop stepwise**

Cheap Tricks

- Use **print statements** to trace/diagnose
- Use a **debug switch**
  - #define DEBUG 1 // debug on
  - #define DEBUG 0 // debug off
  - if (DEBUG) printf(“\n message…”);
- Use a “**hidden debug switch**” via the menu to toggle debug on/off
- Switch off debug for demonstrations!

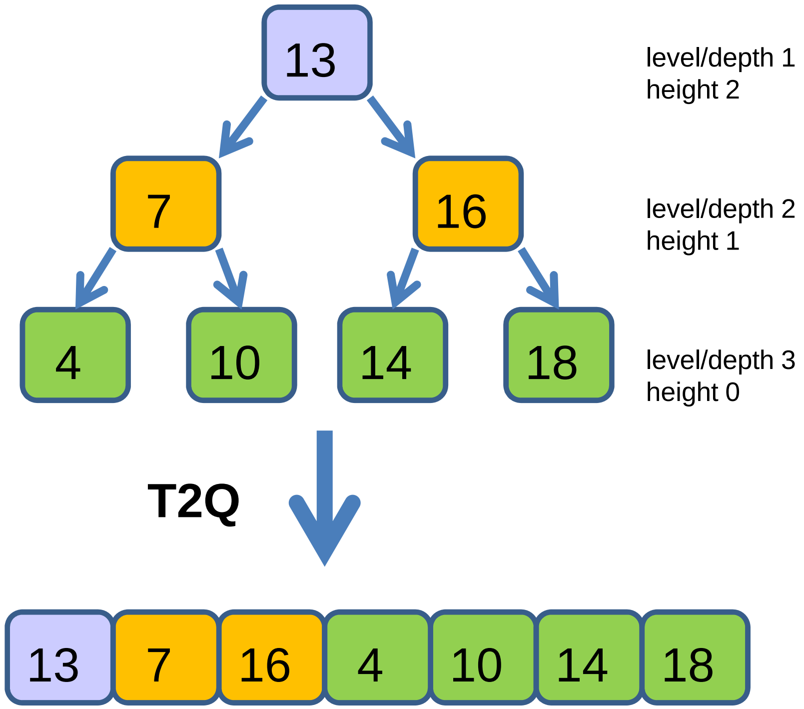
Problem Solving

“Play” with the problem before coding!

Tools: paper & pen!

Example: 2D tree display

- Draw an example:
- Height of the tree is 2
- The tree has 3 levels: 1,2,3
- The maximum number of nodes **at each level** is
    - **2^{L-1} → 1, 2, 4** (L=1, 2, 3)
- The maximum number of nodes in a tree of depth D
    - **$2^{D}-1$ → 1, 3, 7** (D=1,2,3)
- T2Q gives a breadth first order



Final Comments

Food for thought

Comments

- I do **not** want to tell you **how to** program
- I want to **suggest** how you **might** program
- The UI + frontend + backend is a framework

- I want you to think of ADTs + operations
- Implement these operations **stepwise**
- Use a “mental toolbox” i.e. ADTs

Comments

- This is a **start point** for you to develop your own style of programming
- Develop an **automatic style**
- Work on **problem solving**
- Develop your own **thinking process**
- Study code examples & alternatives

Algorithms

Understanding versus Code

Algorithms

- Use **visual representations**
  - Set, sequence, tree, graph
- Develop an **intuitive understanding**
- **Interpret** code (not translate)
- Practise, practise, practise!!!
- Remember
  - **This is new – it takes time!**