University of Nantes
Master 2 in Bioinformatics

Pedagogical Support

Annotated Exercises

Christine Sinoquet

FOREWORD

The solutions to the following exercises were provided by students. They are presented as such, then the erroneous sections are annotated. Afterwards, the different steps of improvement are provided, until a final correct algorithm is obtained.

CONVENTIONS

Annotations in red indicate an error.
The corresponding modifications are shown in blue.
Annotations in green indicate a possible further improvement or optimization.
The corresponding modifications are shown in blue.

TABLE OF CONTENTS

```
typedef struct {
        int val;
        struct TCell* next;
} TCell;
typedef  TCell* TPointer
```

Access to position k in a chained list
Iterative version in pseudo-code

```
/*********************************************************/
// student version
accessPos (input   head : TPointer,
           input   k  : integer,
           output  ptrCell : TPointer,
           output  found : boolean){

 found ← false
 ptrCell ← head

 i ← 0
 while ( i < k && ptrCell != NULL){
   ptrCell ← ptrCell→next
   i       ← i+1
 }

 if (i == k){
  found ← true
 } else {
  found ← false
 }
}

/*********************************************************/

PROCEDURE accessPos (input   head : TPointer,
                     input   k : integer,
                     output  ptrCell : TPointer,
                     output  found : boolean){
 local variables:
 i : integer

 found ← false
 ptrCell ← head

 i ← 0 i ← 1
 while ( (i < k) && and (ptrCell != NULL)){
   ptrCell ← ptrCell→next
   i       ← i+1 i++
 }

 if (i == k ptrCell != NULL){
  found ← true
 } else {
  found ← false
 }
}

/*********************************************************/
```

```
PROCEDURE accessPos (input   head : TPointer,
                     input   k : integer,
                     output  ptrCell : TPointer,
                     output  found : boolean){
 local variables:
 i : integer

 found ← false
 ptrCell ← head

 i ← 1
 while ( (i < k) and (ptrCell != NULL)){
   ptrCell ← ptrCell→next
   i++
 }

 if (ptrCell != NULL){
  found ← true
 }
}

/*********************************************************/
```

```
PROCEDURE accessPos (input   head : TPointer,
                     input   k : integer,
                     output  ptrCell : TPointer,
                     output  found : boolean){
 local variables:
 i : integer

 found ← false
 ptrCell ← head

 i ← 1
 while ( (i < k) and (ptrCell != NULL)){
   ptrCell ← ptrCell→next
   i++
 }

 if (ptrCell != NULL){
  found ← true
 }
}

/*********************************************************/
```

```
PROCEDURE accessPos (input   head : TPointer,
                     input   k : integer,
                     output  ptrCell : TPointer,
                     output  found : boolean){
 local variables:
 i : integer

 ptrCell ← head
 i       ← 1
 while ( (i < k) and (ptrCell != NULL)){
   ptrCell ← ptrCell→next
   i++
 }

  found ←  (ptrCell != NULL)
}

/*********************************************************/
```

Translation into C

```
void accessPos (TPointer       head,
                int            k,
                TPointer*      adr_ ptrCell,
                boolean*       adr_found){
 int i ;

 *adr_ptrCell = head ;
 i          = 1 ;
 while ( (i < k) && ( *adr_ptrCell != NULL)){
    *adr_ptrCell =  *adr_ptrCell→next ;
   i++ ;
 }

  *adr_found =  ( *adr_ptrCell != NULL) ;
}

/*********************************************************/

void main(){
 TPointer    head_list = create_chained_list(...) ; // not developed here
 int         k         = 4 ;
 TPointer    physical_address ;
 boolean     found ;

  accessPos(head_list, k, &physical_address, &found) ;
}

/*********************************************************/
```

Translation into C++

```
void accessPos (TPointer       head,
                int            k,
                TPointer  & ptrCell,
                boolean   & found){
 int i ;

 ptrCell = head ;
 i       = 1 ;
 while ( (i < k) && ( ptrCell != nullptr)){
   ptrCell =  ptrCell→next ;
   i++ ;
 }

  found =  ( ptrCell != nullptr) ;
}

/*********************************************************/

void main(){
 TPointer    head_list = create_chained_list(...) ; // not developed here
 int         k         = 4 ;
 TPointer    physical_address ;
 boolean     found ;

  accessPos(head_list, k,  physical_address,  found) ;
}

/*********************************************************/
/*********************************************************/
/*********************************************************/
```

Access to position k in a chained list
Iterative version in pseudo-code, version 2

```
/*******************************************************/

// student version
PROCEDURE accessPos (input    head : TPointer,
                     input    k : integer,
                     output   ptrCell : TPointer,
                     output   found : boolean){
 ptrCell ← head
 found   ← false
 while ( (ptrCell != NULL) and (not found)){
   k ← k-1
   if (k==1){
     found ← true
   } else {
    ptrCell ← ptrCell→next
  }
 }
}

/*******************************************************/

PROCEDURE accessPos (input    head : TPointer,
                     input    k : integer,
                     output   ptrCell : TPointer,
                     output   found : boolean){
 ptrCell ← head
 found   ← false
 while ( (ptrCell != NULL) and (not found)){
    k ← k-1 // Since paramater k is passed by value, the decrement of k will have no effect outside
            // the procedure.
     if (k==0){
       found ← true
     } else {
      ptrCell ← ptrCell→next
    }
 }
}

/*******************************************************/
/*******************************************************/
/*******************************************************/
```

● Copy of a chained list
Iterative version in pseudo-code
Version 1

```
/*********************************************************/

// student version
PROCEDURE clone( input   head1 : TPointer ,
                 output head2 : TPointer){

 p1 ← head1
 while (p1!= NULL){
  p2 ← new Pointer
  p2→ val ← p1→ val
  p1 ← p1→next
  p_sauv→next ← p2
 }
}

/*********************************************************/

PROCEDURE clone( input   head1 : TPointer ,
                 output head2 : TPointer){

 local variables:
 p1, p2, p2_prec : TPointer

 p1 ← head1 ; p2_prec ← NULL;
 while (p1!= NULL){
  p2 ← new Pointer p2 ← allocate_memory()
  p2→ val ← p1→ val
  p1 ← p1→next
  p_sauv→next ← p2     if (p2_prec == NULL){
                         head2 ← p2
                       } else {
                         p2_prec→next ← p2
                       }

  p2_prec ← p2
 }

 if (head1 == NULL){
  head2 ← NULL
 } else {
  p2→next ← NULL
 }
}

/*********************************************************/
```

```
PROCEDURE clone( input   head1 : TPointer ,
                 output head2 : TPointer){

 local variables:
 p1, p2, p2_prec : TPointer

  p1 ← head1 ; p2_prec ← NULL;
  while (p1!= NULL){
   p2 ← allocate_memory()
   p2→ val ← p1→ val

   p1 ← p1→next

   if (p2_prec == NULL){
     head2 ← p2
   } else {
    p2_prec→next ← p2
   }
   p2_prec ← p2
  }

  if (head1 == NULL){
   head2 ← NULL
  } else {
   p2→next ← NULL
  }
}

/*********************************************************/
```

```
PROCEDURE clone( input   head1 : TPointer ,
                 output head2 : TPointer){

 local variables:
 p1, p2, p2_prec : TPointer

  p1 ← head1 ; p2_prec ← NULL;
  while (p1!= NULL){
   p2 ← allocate_memory()
   p2→ val ← p1→ val

   p1 ← p1→next

   if (p2_prec == NULL){
     head2 ← p2
   } else {
    p2_prec→next ← p2
   }
   p2_prec ← p2
  }

  if (head1 == NULL){
   head2 ← NULL
  } else {
   p2→next ← NULL
  }
}

/*********************************************************/
```

```
PROCEDURE clone( input   head1 : TPointer ,
                 output head2 : TPointer){

 local variables:
 p1, p2, p2_prec : TPointer

  p1 ← head1 ; p2_prec ← NULL;
  while (p1!= NULL){
   p2 ← allocate_memory()
   p2→ val ← p1→ val

   p1 ← p1→next

   if (p2_prec == NULL){
     head2 ← p2
   } else {
    p2_prec→next ← p2
   }
   p2_prec ← p2
  }

  if (head1 == NULL){head2 ← NULL ; return}
  p2→next ← NULL
}

/*********************************************************/
/*********************************************************/
/*********************************************************/
```

Copy of a chained list
Iterative version in C++, version 2

```
/*********************************************************/

// student version
void copy(TPointer     head1,
                TPointer & head2){
  boolean first = true ;
  TPointer prec ;
  TPointer p2 ;
  while(head1!= nullptr){
    p2 = new TCell ;
    if (first) {
     head2 = p2 ;
    } else {
      prec→next = p2 ;
    }
    p2→val = head1→val ;
    head1 = head1→next ;
    prec = p2 ;
    first = false ;
  }

  if (head1 == nullptr){
    head2 =  nullptr;}
  else  {
   p2→next = nullptr ;}
}
```

```
void copy(TPointer     head1,
                TPointer & head2){
  boolean first = true ;
  TPointer prec ;
  TPointer p2 ;
  while(head1!= nullptr){
    p2 = new TCell ;
    if (first) {
     head2 = p2 ;
    } else {
      prec→next = p2 ;
    }
    p2→val = head1→val ;
    head1 = head1→next ;
    prec = p2 ;
    first = false ;
  }

  if (head1 == nullptr){
    head2 =  nullptr;}
  else  {
   p2→next = nullptr ;}
}

/*************************************************************/
```

```
void copy(TPointer     head1,
          TPointer & head2){
  boolean first = true ;
  TPointer prec ;
  TPointer p2 ;
  while(head1!= nullptr){
    p2 = new TCell ;
    if (first) {
     head2 = p2 ;
     first = false ;
    } else {
      prec→next = p2 ;
    }
    p2→val = head1→val ;
    head1 = head1→next ;
    prec = p2 ;
  }

  if (head1 == nullptr){
    head2 =  nullptr;
  } else  {
   p2→next = nullptr ;
 }
}

/*********************************************************/
```

```
void copy(TPointer     head1,
          TPointer & head2){
  boolean first = true ;
  TPointer prec2 ;
  TPointer  p1 = head1; // Even if parameter head1 is passed by value, as is therefore not
                        // impacted by modifications within procedure copy, it is not wise
                        // to use head1 to traverse list head1. A local variable (p1)
                        // should be used instead.
  TPointer p2 ;
  while(p1 != nullptr){
    p2 = new TCell ;
    if (first) {
     head2 = p2 ;
     first = false ;
    } else {
      prec2→next = p2 ;
    }
    p2→val = p1→val ;
    p1 = p1→next ;
    prec2 = p2 ;
  }

  if (head1 == nullptr){
    head2 =  nullptr;
  } else  {
   p2→next = nullptr ;
 }
}

/*********************************************************/
/*********************************************************/
/*********************************************************/
```

Copy of a chained list
Iterative version in C++, version 3

```
**********************************************************/

// student version
void clone_bis(TPointer     head1,
                    TPointer & head2){
  if (head1 == nullptr){
    head2 = nullptr ;
  } else {

   p1   = head1 ;
   p2 = (char *) malloc(sizeof (char)) ;

   while (p1!= nullptr){
    p2→val = p1→val ;
    p1 = p1→next ;
    if (p1!= nullptr){
     p_clone→next = (char*) malloc(sizeof(char)) ;
     p2→next = p_clone
    }
   }
  }
}

/**********************************************************/
```

```
void clone_bis(TPointer     head1,
               TPointer & head2){
 TPointer  p1,p2 ;

 if (head1 == nullptr){
  head2 = nullptr ;
 } else {

  // The list head1 contains at least one element.
  p1   = head1 ;
 p2 = (char *) malloc(sizeof (char)) ; p2 = new TCell ;

  while (p1!= nullptr){
   p2→val = p1→val ;
   p1 = p1→next ;
   if (p1!= nullptr){
    p_clone→next = (char*) malloc(sizeof(char)) ;  p_clone = new TCell ;
    p2→next = p_clone ;
   p2 = p_clone ;
   } else {
   p2→next = nullptr ;
   }
  } // end while
 } // end else
}

/*********************************************************/
```

```
void clone_bis(TPointer     head1,
               TPointer & head2){
 TPointer  p1,p2 ;

 if (head1 == nullptr){
  head2 = nullptr ;
 } else {

  // The list head1 contains at least one element.
  p1   = head1 ;
  p2   = new TCell ;

  while (p1!= nullptr){
   p2→val = p1→val ;
   p1 = p1→next ;
   if (p1!= nullptr){
    p_clone = new TCell ;
    p2→next = p_clone ;
   p2 = p_clone ;
   } else {
   p2→next = nullptr ;
   }
  } // end while
 } // end else
}

/*********************************************************/
```

```
void clone_bis(TPointer     head1,
               TPointer & head2){
 TPointer  p1,p2 ;

 if (head1 == nullptr){head2 = nullptr ; return;}

 // The list head1 contains at least one element.
 p1   = head1 ;
 p2   = new TCell ;

 while (p1!= nullptr){
  p2→val = p1→val ;
  p1 = p1→next ;
  if (p1!= nullptr){
   p_clone = new TCell ;
   p2→next = p_clone ;
   p2 = p_clone ;
  } else {
   p2→next = nullptr ;
  }
 } // end while

}

/*************************************************************/
/*************************************************************/
/*************************************************************/
```

Insertion at position k in a chained list
Iterative version in C++, version 1

```
/*********************************************************/

// student version
void insertPos(TPointer      head,
               int           info,
               int           k,
               bool        & possible){
  TPointer ptr_after_k ;
  TPointer ptr_cell ;

  ptr_cell = head ;

  int pos = 1 ;
  possible = false ;
  k = k-1 ;

  while (pos!= k & ptr_cell→next != nullptr){
    ptr_cell = ptr_cell→next ;
    pos++ ;
  }

  if (pos == k){
    ptr_after_k = ptr_cell→next ;
    TPointer novel_p ;
    novel_p→val = info ;
    novel_p→next = ptr_cell_after_k ;
    ptr_cell→next = novel_p ;
  }

  if ((k+1) == 1){
   add_front(head,info) ;
  }
}

/*********************************************************/

void main(){
  TPointer list = create_chained_list() ; // not developed here
  int k = 4 ;
  bool is_inserted ;
  int value_to_insert ;
  insertPos(list,  value_to_insert,  k,  is_inserted) ;
}

/*********************************************************/
```

```
void insertPos(TPointer    & head,
               int           info,
               int           k,
               bool        & possible){
 TPointer ptr_after_k ;  TPointer ptr_k ;
 TPointer ptr_cell ;

 ptr_cell = head ;

 int pos = 1 ;
 possible = false ;
 k = k-1 ; // confusing

 // test is missing to handle        if (head == nullptr){
 // the case of  the                   if (k==1){add_front(head,info) ; possible = true ;}
 // empty cell                         return ;
                                      }

 int pos = 1 ;
 while (pos!= k & ptr_cell →next != nullptr){  while ((pos < k-1) && (ptr_cell != nullptr)){
   ptr_cell = ptr_cell→next ;
   pos++ ;
 }

 if (pos == k){    if (pos == k-1){
   possible = true ;
   ptr_k = ptr_cell→next ;        // ptr_k may be equal to nullptr, which indicates
                                  // that cells 1 to k-1 exit, but cell k does not. In this case,
                                  // the insertion is possible and will take place at the end of the list.
  TPointer novel_p;    TPointer novel_p = new TCell;
   novel_p→val = info ;
   novel_p→next = ptr_k ; // possibly equal to nullptr
   ptr_cell→next = novel_p ;
 }

 if ((k+1) == 1){    if (k == 1){ // Insertion in a front of a non empty list.
   add_front(head,info) ;          // The previous ‘’while” and ‘’if” instructions where short circuited.
   possible = true ;
 }
}

/*********************************************************/

void main(){
 TPointer list = create_chained_list() ; // not developed here
 int          k = 4 ;
 bool         is_inserted ;
 int          value_to_insert = 18;
 insertPos(list,  value_to_insert,  k,  is_inserted) ;
}

/*********************************************************/
```

```
void insertPos(TPointer    & head,
               int           info,
               int           k,
               bool        & possible){
 TPointer ptr_k ;
 TPointer ptr_cell = head ;

 possible = false ;

 if (head == nullptr){
  if (k==1){add_front(head,info) ; possible = true ;}
  return ;
 }

 int pos = 1 ;
 while ((pos < k-1) && (ptr_cell != nullptr)){
  ptr_cell = ptr_cell→next ;
  pos++ ;
 }

  if (pos == k-1){
  possible = true ;
  ptr_k = ptr_cell→next ;        // ptr_k may be equal to nullptr, which indicates
                                 // that cells 1 to k-1 exit, but cell k does not. In this case,
                                 // the insertion is possible and will take place at the end of the list.
  TPointer novel_p = new TCell;
  novel_p→val = info ;
  novel_p→next = ptr_k ; // possibly equal to nullptr
  ptr_cell→next = novel_p ;
 }

 if (k == 1){                // Insertion in a front of a non empty list.
  add_front(head,info) ; // The previous ‘’while’’ and ‘’if’’ instructions where short circuited.
  possible = true ;
 }

}

/*********************************************************/

void main(){
 TPointer list = create_chained_list() ; // not developed here
 int          k = 4 ;
 bool         is_inserted ;
 int          value_to_insert = 18 ;
 insertPos(list,  value_to_insert,  k,  is_inserted) ;
}

/*********************************************************/
/*********************************************************/
/*********************************************************/
```

Insertion at position k in a chained list
Iterative version in C++, version 2

```
/*********************************************************/

// student version

void addPos(TPointer & head,
                  int              k,
                  int              info,
                  bool         & possible){
 TPointer  p = head ;
 TPointer prec ;
  bool       front_insertion = true ;
 possible          = false ;
 while ((p!= nullptr) && (not possible)){
   k = k-1 ;
   if (k == 0){
     possible = true ;
   } else {
    front_insertion = false ;
    prec = p ;
    p      = p→next ;
  }
 }

  if (possible){
    add_front(p,info) ;
    if (front_insertion){
      head = p ;
    } else {
      prec→next = p ;
    }
  }

  if (k-1 == 0){
    possible = true ;
    addQueue(prec,info) ;
  }

}

/*********************************************************/
```

```
void addPos(TPointer & head,
                int             k,
                int             info,
                bool        & possible){
 TPointer  p = head ;
 TPointer prec ;
  bool       front_insertion = true ;
 possible                        = false ;
 while ((p!= nullptr) && (not possible)){
  k = k-1 ;
  if (k == 0){
    possible = true ;
  } else {
   front_insertion = false ;
   prec = p ;
   p      = p→next ;
  }
 }

  if (possible){ // possible could only be set to true if the list was not empty
    add_front(p,info) ;
    if (front_insertion){
      head = p ;
    } else {
      prec→next = p ;
    }
   return ;
  }

 // case of the empty list
 if (k-1 == 0){   if (head == nullptr){ // The above while instruction was short circuited.
   if (k == 1) {                         //  It is only possible to insert an element in an empty list at
     possible = true ;                   // the first position.
     add_front(head,info) ;
   }
   addQueue(prec,info) ;
 }

}

/*********************************************************/
```

```
void addPos(TPointer & head,
                int             k,
                int             info,
                bool        & possible){
 TPointer  p = head ;
 TPointer prec ;
  bool       front_insertion = true ;
 possible                    = false ;
 while ((p!= nullptr) && (not possible)){
  k = k-1 ;
  if (k == 0){
   possible = true ;
  } else {
   front_insertion = false ;
   prec = p ;
   p      = p→next ;
  }
 }

  if (possible){ // possible could only be set to true if the list was not empty
   add_front(p,info) ;
   if (front_insertion){
    head = p ; add_front(head,info) ;
   } else {
    prec →next = p ; add_front(rec_next,info) ;
   }
   return ;
  }

 // case of the empty list
  if (head == nullptr){        // The above while instruction was short circuited.
   if (k == 1) {               //  It is only possible to insert an element in an empty list at
    possible = true ;          // the first position.
    add_front(head,info) ;
   }
  }

}

/*********************************************************/
```

```
void addPos(TPointer & head,
                int             k,
                int             info,
                bool        & possible){
 TPointer  p = head ;
 TPointer prec ;
  bool       front_insertion = true ;
 possible                        = false ;
 while ((p!= nullptr) && (not possible)){
   k = k-1 ;
   if (k == 0){
     possible = true ;
   } else {
    front_insertion = false ;
    prec = p ;
    p      = p→next ;
  }
 }

  if (possible){ // possible could only be set to true if the list was not empty
    if (front_insertion){
      add_front(head,info) ;
    } else {
      add_front(rec_next,info) ;
    }
    return ;
  }

  // case of the empty list
  if (head == nullptr){          // The above while instruction was short circuited.
     if (k == 1) {               //  It is only possible to insert an element in an empty list at
       possible = true ;         // the first position.
       add_front(head,info) ;
   }
  }

}

/*********************************************************/
/*********************************************************/
/*********************************************************/
```

Insertion at position k in a chained list
Iterative version in C++, version 3

```
/*********************************************************/

// student version

void insertPos(TPointer & head,
               int          k,
               int          info,
               bool       & possible){

 TPointer  p ;
  TPointer prec ;

  p = head ;
  prec = nullptr ;
  int cnt = 1 ;
   possible = false ;
   while( (p!=nullptr) && (cnt <k)){
     prec = p ; p = p→next ;
   }

  if (k==1){
    add_front(head,info) ;
    possible = true ;
    return ;
  }

TPointer p_new = new TCell ;
 p_new→ val = info ;
 p_new→ next = p ;
 prec→next = p_new ;
 possible = true ;
 return ;
}

/*********************************************************/
```

```
void insertPos(TPointer & head,
               int          k,
               int          info,
               bool       & possible){

 TPointer  p ;
  TPointer prec ;

   possible = false ;

 if (k==1){
    add_front(head,info) ;
    possible = true ;
    return ;
  }

  p = head ;
 prec = nullptr ;
  int cnt = 1 ;
 possible = false ;
   while( (p!=nullptr) && (cnt <k)){
     prec = p ; p = p→next ; cnt++ ;
   }
  // postcondition :
  // p == nullptr  and cpt < k : The non empty list is too short.
  // p == nullptr and cpt ==k : Insertion at the end of the non-empty list.
  // p != nullptr and cpt == k : Insertion in the middle of the non-empty list.

 if (k==1){
    add_front(head,info) ;
    possible = true ;
    return ;
 }

 if ( (p==nullptr) && (cnt <k){return ;} // The non empty list is too short.

TPointer p_new = new TCell ;
 p_new→ val = info ;
 p_new→ next = p ;
 prec→next = p_new ;
 possible = true ;
 return ;
}

/*********************************************************/
```

```
void insertPos(TPointer & head,
               int          k,
               int          info,
               bool       & possible){

 TPointer  p ;
  TPointer prec ;

   possible = false ;

   if (k==1){
    add_front(head,info) ;
    possible = true ;
    return ;
  }

  p = head ;
  int cnt = 1 ;
   while( (p!=nullptr) && (cnt <k)){
     prec = p ; p = p→next ; cnt++ ;
    }
    // postcondition :
  // p == nullptr  and cpt < k : The non empty list is too short.
  // p == nullptr and cpt ==k : Insertion at the end of the non-empty list.
  // p != nullptr and cpt == k : Insertion in the middle of the non-empty list.

 if ( (p==nullptr) && (cnt <k){return ;} // The non empty list is too short.

 add_front(prec→next,info) ;
 possible = true ;
}

/*********************************************************/
```

```
void insertPos(TPointer & head,
               int          k,
               int          info,
               bool       & possible){

 TPointer  p ;
  TPointer prec ;

   possible = false ;

   if (k==1){
    add_front(head,info) ;
    possible = true ;
    return ;
  }

  p = head ;
  int cnt = 1 ;
   while( (p!=nullptr) && (cnt <k)){
     prec = p ; p = p→next ; cnt++ ;
    }
    // postcondition :
  // p == nullptr  and cpt < k : The non empty list is too short.
  // p == nullptr and cpt ==k : Insertion at the end of the non-empty list.
  // p != nullptr and cpt == k : Insertion in the middle of the non-empty list.

 if ( (p==nullptr) && (cnt <k){return ;} // The non empty list is too short.

 add_front(prec→next,info) ;
 possible = true ;
}

/*********************************************************/
/*********************************************************/
/*********************************************************/
```

Insertion at queue position in a chained list
Iterative version in C++, version 1

```
/*********************************************************/

// student version

void insertQueue(TPointer & head,
                 int        info){

 TPointer p = head ;
 if (p!= nullptr){
   while(p→next!= nullptr){
     p = p→next ;
   }
 }

 TPointer p_new = new TCell ;
 p_new→val = info ;
 p_new→next = nullptr ;
 p→next = p_new ;

}

/*********************************************************/
```

```
void insertQueue(TPointer & head,
                 int        info){

 TPointer   p    = head ;
  TPointer prec = nullptr ;
 if (p!= nullptr){                    while(p !=  nullptr){
   while(p→next!= nullptr){             prec = p ; p = p→next ;
     p = p→next ;                     }
   }
 }

 if (prec == nullptr){
   add_front(head, info) ;
 } else {

  TPointer p_new = new TCell ;
  p_new→val = info ;
  p_new→next = nullptr ;
  p→next = p_new ;
}

}

/*********************************************************/
```

```
void insertQueue(TPointer & head,
                 int        info){

 TPointer   p     = head ;
  TPointer prec = nullptr ;
   while(p !=  nullptr){
       prec = p ; p = p→next ;
   }

 if (prec == nullptr){
   add_front(head, info) ;
 } else {
  add_front(prec→next,info) ;
}

}

/*********************************************************/
```

```
void insertQueue(TPointer & head,
                 int        info){

 TPointer   p     = head ;
  TPointer prec = nullptr ;
   while(p !=  nullptr){
       prec = p ; p = p→next ;
   }

 if (prec == nullptr){
   add_front(head, info) ;
 } else {
  add_front(prec→next,info) ;
}

}

/*********************************************************/
/*********************************************************/
/*********************************************************/
```

Reversal of a chained list
Iterative version in C++, version 1

```
/*********************************************************/

// student version

void reverse (TPointer       head1,
                    TPointer  & head2){

 TPointer p1 = head1 ;
  head2 = nullptr ;
  if (p1!= nullptr){
    while(p1→next !=  nullptr){
      add_front(head2,p1→val) ;
    }
  }

}
/*********************************************************/
```

```
void reverse (TPointer       head1,
                    TPointer  & head2){

 TPointer p1 = head1 ;
  head2 = nullptr ;
 if (p1!= nullptr){
    while(p1→next !=  nullptr){       while (p1!= nullptr){
      add_front(head2,p1→val) ;
    }
 }

}
/*********************************************************/
```

```
void reverse (TPointer       head1,
                  TPointer  & head2){

 TPointer p1 = head1 ;
  head2 = nullptr ;
   while (p1!= nullptr){
     add_front(head2,p1→val) ;
   }

}
/*********************************************************/

void reverse (TPointer       head1,
                  TPointer  & head2){

 TPointer p1 = head1 ;
  head2 = nullptr ;
   while (p1!= nullptr){
     add_front(head2,p1→val) ;
   }

}

/*********************************************************/
/*********************************************************/
/*********************************************************/
```

Retrieval of an information in a binary tree, with display of the values encountered along the path down to the information if existing
Recursive version in pseudo-code, version 1

```
/*********************************************************/

// student version

PROCEDURE find_elem_and_display_path (input          n : TBinTree,
                                      input          e : integer,
                                      input/output   list : TPointer,
                                      input/output   found : boolean){

 if  (n!= null && ! found){

   add_queue(list,n→val)
   if (n→val == e){
     display_list(list)
     found ← true
   } else {
      find_elem_and_display_path(n→lc, e, list, found)
      find_elem_and_display_path(n→rc, e, list, found)
      pop_queue(list)
  }

 }
}

/*********************************************************/

void main(){
  root         : TBinTree
  path         : TPointer
  found        : boolean
  searched_val : integer

  root ← create_binary_tree() // not developed here

  found ← false ; path ← null ; searched_val = 48
  find_elem_and_display_path (root, searched_val, path, found)
}

/*********************************************************/
```

```
PROCEDURE find_elem_and_display_path (input            n : TBinTree,
                                      input            e : integer,
                                      input/output     list : TPointer,
                                      input/output     found : boolean){

if (n == null){
} else {

~~if (n!= null && ! found){~~   if (n!= null){

  add_queue(list,n→val)
  if (n→val == e){
    display_list(list)
    found = true
  } else {
     find_elem_and_display_path(n→lc, e, list, found)
     if ( ! found){ find_elem_and_display_path(n→rc, e, list, found)}
     if ( ! found){pop_queue(list)}
  }
 }
}
/*********************************************************/

void main(){
  root            : TBinTree
  path            : TPointer
  found           : boolean
  searched_val : integer

  root ← create_binary_tree() // not developed here

  found ← false ; path ← null ; searched_val = 48
  find_elem_and_display_path (root, searched_val, path, found)
}

/*********************************************************/
```

```
PROCEDURE find_elem_and_display_path (input          n : TBinTree,
                                      input          e : integer,
                                      input/output   list : TPointer,
                                      input/output   found : boolean){

if (n == null){
} else {        if (n!= null){
  add_queue(list,n→val)
  if (n→val == e){
    display_list(list)
    found = true
  } else {
     find_elem_and_display_path(n→lc, e, list, found)
     if ( ! found){ find_elem_and_display_path(n→rc, e, list, found)}
     if ( ! found){pop_queue(list)}
  }
 }
}
/*********************************************************/

void main(){
  root          : TBinTree
  path          : TPointer
  found         : boolean
  searched_val : integer

  root ←  create_binary_tree() // not developed here

  found ← false ; path ← null ; searched_val = 48
  find_elem_and_display_path (root, searched_val, path, found)
}

/*********************************************************/
```

```
PROCEDURE find_elem_and_display_path (input            n : TBinTree,
                                      input            e : integer,
                                      input/output     list : TPointer,
                                      input/output     found : boolean){

  if (n!= null){
   add_queue(list,n→val)
   if (n→val == e){
     display_list(list)
     found = true
   } else {
      find_elem_and_display_path(n→lc, e, list, found)
      if ( ! found){
       find_elem_and_display_path(n→rc, e, list, found)
       if ( ! found){pop_queue(list)}
      }
  }
 }
}
/*********************************************************/

void main(){
  root            : TBinTree
  path            : TPointer
  found           : boolean
  searched_val : integer

  root ← create_binary_tree() // not developed here

  found ← false ; path ← null ; searched_val = 48
  find_elem_and_display_path (root, searched_val, path, found)
}

/*********************************************************/
```

```
PROCEDURE find_elem_and_display_path (input          n : TBinTree,
                                      input          e : integer,
                                      input/output   list : TPointer,
                                      input/output   found : boolean){

  if (n!= null){
   add_queue(list,n→val)
   if (n→val == e){
     display_list(list)
     found = true
   } else {
      find_elem_and_display_path(n→lc, e, list, found)
      if ( ! found){
       find_elem_and_display_path(n→rc, e, list, found)
       if ( ! found){pop_queue(list)}
      }
  }
 }
}

/*********************************************************/

void main(){
  root          : TBinTree
  path          : TPointer
  found         : boolean
  searched_val : integer

  root ← create_binary_tree() // not developed here

  found ← false ; path ← null ; searched_val = 48
  find_elem_and_display_path (root, searched_val, path, found)
}

/*********************************************************/
/*********************************************************/
/*********************************************************/
```

Retrieval of an information in a binary tree, with display of the values encountered along the path down to the information if existing
Recursive version in pseudo-code, version 2

```
/*********************************************************/
// student version

PROCEDURE find_elem_and_display_path_bis (input          n : TBinTree,
                                          input          e : integer,
                                          input/output   list : TPointer,
                                          input/output   found : boolean){

  if ((n→val == e) && (n!=null)){
    add_front(list, n→val)
    found = true
  } else {
     find_elem_and_display_path_bis(n→lc, e, list, found)
     if ( ! found){find_elem_and_display_path_bis(n→rc, e, list, found)}
     if ( found){add_front(list, n→val)}
  }
}

/*********************************************************/

void main(){
  root          : TbinTree
  path          : TPointer
  found         : boolean
  searched_val : integer

  root ←  create_binary_tree() // not developed here

  found ← false ; path ← null ; searched_val = 48
  find_elem_and_display_path_bis (root, searched_val, path, found)
  if (path!= null) {display(path)}
}

/*********************************************************/
```

```
PROCEDURE find_elem_and_display_path_bis (input          n : TBinTree,
                                          input          e : integer,
                                          input/output   list : TPointer,
                                          input/output   found : boolean){

  if ((n→val == e) && (n!=null)){      if ((n!=null) && (n→val == e)){ // left to right evaluation,
                                                                       //  precondition for
                                                                       //  test n→val == e :
                                                                       //  n!= null
                                  // Besides, the else section below applies to (n!=null and n→val!= e)
                                  // not to the negation of the test ((n!=null) && (n→val == e))
    add_front(list, n→val)
    found = true
  } else {
     find_elem_and_display_path_bis(n→lc, e, list, found)
     if ( ! found){find_elem_and_display_path_bis(n→rc, e, list, found)}
     if ( found){add_front(list, n→val)}
   }
}

/*********************************************************/

void main(){
  root          : TbinTree
  path          : TPointer
  found         : boolean
  searched_val : integer

  root ← create_binary_tree() // not developed here

  found ← false ; path ← null ; searched_val = 48
  find_elem_and_display_path_bis (root, searched_val, path, found)
  if (path!= null) {display(path)}
}

/*********************************************************/
```

```
PROCEDURE find_elem_and_display_path_bis (input          n : TBinTree,
                                         input          e : integer,
                                         input/output   list : TPointer,
                                         input/output   found : boolean){

if (n!=null){

  if (n→val == e)){
   add_front(list, n→val)
   found = true
  } else {
    find_elem_and_display_path_bis(n→lc, e, list, found)
    if ( ! found){find_elem_and_display_path_bis(n→rc, e, list, found)}
    if ( found){add_front(list, n→val)}
   }

 }
}

/*********************************************************/

void main(){
  root          : TbinTree
  path          : TPointer
  found         : boolean
  searched_val : integer

  root ← create_binary_tree() // not developed here

  found ← false ; path ← null ; searched_val = 48
  find_elem_and_display_path_bis (root, searched_val, path, found)
  if (path!= null) {display(path)}
}

/*********************************************************/
```

```
PROCEDURE find_elem_and_display_path_bis (input            n : TBinTree,
                                         input            e : integer,
                                         input/output     list : TPointer,
                                         input/output     found : boolean){

if (n!=null){

   if (n→val == e)){
     add_front(list, n→val)
     found = true
   } else {
      find_elem_and_display_path_bis(n→lc, e, list, found)
      if ( ! found){find_elem_and_display_path_bis(n→rc, e, list, found)}
      if ( found){add_front(list, n→val)}
    }

 }
}

/*********************************************************/

void main(){
  root            : TbinTree
  path            : TPointer
  found           : boolean
  searched_val : integer

  root ← create_binary_tree() // not developed here

  found ← false ; path ← null ; searched_val = 48
  find_elem_and_display_path_bis (root, searched_val, path, found)
  if (path!= null) {display(path)}
}

/*********************************************************/
```