Abstract Data Types

Table of Contents

- Data Structures
- Language Defined Data Type
- Abstract Data Types
- Info Hiding / Encapsulation
- Sorted Lists: Array of Records
- Array of Records: Manipulations
- Array of Records: Insert
- Autonomous List Interface Notation
- Autonomous List: Item Interface
- Autonomous List: List Interface
- Autonomous List: Pointer Implementation
- Autonomous List: Array of Records Implementation
- High-Level Implementation: Insert
- High-Level Implementation: Remove
- Alternative FreeNode Management
- Generic Sorted List ADT specification

Data Type

- a collection of related data elements plus accessing (manipulation) operations used to retrieve elements.

Types

- Built-In (Language Defined)
  - Array, Structures (Records)
- Programmer Defined, Abstract Data Types (ADT)
  - Lists, Stacks, Queues

Views

- Abstract (Logical)
  - organization viewed by the user
    - (black box view)
- Implementation (Physical)
  - coding methods used to represent the data and the operations
    - (open box view)
- Application
  - usage in a particular program
    - (variant box view)
Example

- Two-Dimensional Array
  
  † Logical View: table, matrix
  † Application View: maze, surface points
  † Physical View
    Stored sequentially (implies logical to physical mapping)
    Index Limits (L1 .. U1, L2 .. U2)
    Length = (U1 - L1 + 1) * (U2 - L2 + 1)

Accessing

- Column Major: all elements in a column are stored in sequence
  FORTRAN - Column Major

- Row Major: all elements in a row are stored in sequence
  "C" , PASCAL - Row Major

Row Major Accessing:

- Location of [i][j] element (Row Major)
  \[ \beta + (U2-L2+1) \times (i-L1) + (j-L2) \]
  \[ \beta = \text{base address of array} \]
  \( (U2 - L2 + 1) = \text{Size of Row} \)
  \( (i - L1) = \text{number of rows to skip} \)
  \( (j - L2) = \text{number of columns to skip} \)

Logical view (1,1 1,2 1,3 1,4 2,1 2,2 2,3 2,4 3,1 3,2 3,3 3,4)

Physical (row-major) compiler programmer linear view (1,1 1,2 1,3 1,4 2,1 2,2 2,3 2,4 3,1 3,2 3,3 3,4)

Physical (column-major) compiler programmer linear view (1,1 2,1 3,1 2,2 3,2 1,3 2,3 3,3 1,4 2,4 3,4)
Abstract Data Types

**ADTs**

**Abstract Data Types**

- New data type defined by programmer
- Includes:
  † Set of Data Objects
  † Set of Abstract Operations

**Data Abstraction**

- Design of abstract data objects and operations upon those objects.
- Abstraction in programming separates a data type’s logical properties from its implementation.

**Information Hiding**

- Used in the design of functions and new data types.
- Each component should hide as much information as possible from the users of the component (function).
- Implementation details are hidden from the user by providing access through a well-defined communication interface.

**Encapsulation**

- the bundling of data and actions in such a way that the logical properties of the data and actions are separated from the implementation details [Dale].
- access to an ADT is **restricted** to a specified set of supplied operations
- Implies:
  † User has no "Need to Know"
  † User may not directly manipulate data elements
- Advantages
  † Changes to the underlying operators or representations does not affect code in a client of the data type
  † Extends programming languages

The distinction between these terms is not well recognized. Some authors do not distinguish between information hiding and encapsulation, while others assign the opposite definitions given here.
What is the difference between Information Hiding and Encapsulation?

**Information Hiding**
- A question of *program design*
- In many cases: Language Supported (functions, procedures)

**Encapsulation**
- A question of *language design*
- "...an abstraction is effectively encapsulated only when the language prohibits access to information hidden within the abstraction."
- Ada packages
- C++ classes
  † “C” modules

**Info Hiding Example: Strings**

String Data Type provides interface to the user program. Operation implementation details are hidden.

String: string
  substring
  concat
  delete
  insert

User Program

Available Pool
- Linked List of Free Nodes
- Deletion performed by `AllocateNewNode()`
- Insertion performed by `FreeNode()`
- No ordering
- Operations must be efficient
  - Head of List — Inserts/Deletes
  - Link field contains index to next available node.
  - Item fields contain 'leftover' values which are ignored

**Sorted Lists: Array of Records**

Array of Records (structures)

Implementation Instance (nodes structure array)

```
<table>
<thead>
<tr>
<th>Item</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>? 9</td>
</tr>
<tr>
<td>1:</td>
<td>G 5</td>
</tr>
<tr>
<td>2:</td>
<td>C 7</td>
</tr>
<tr>
<td>3:</td>
<td>X -1</td>
</tr>
<tr>
<td>4:</td>
<td>? -1</td>
</tr>
<tr>
<td>5:</td>
<td>K 8</td>
</tr>
<tr>
<td>6:</td>
<td>? 0</td>
</tr>
<tr>
<td>7:</td>
<td>E 1</td>
</tr>
<tr>
<td>8:</td>
<td>R 3</td>
</tr>
<tr>
<td>9:</td>
<td>? 4</td>
</tr>
</tbody>
</table>
```

Initialization

List: (2) C ⇒ (7) E ⇒ (1) G ⇒ (5) K ⇒ (8) R ⇒ (3) X ⇒ (-1) null
Avail: (6) ⇒ (0) ⇒ (9) ⇒ (4) ⇒ (-1) null
Array of Records: Manipulations

Results of `insert(List, “A”);`

```
Item  Link
0:  ?   9
1:  G   5
2:  C   7
3:  X   -1
4:  ?   -1

List:
(6) A ⇒ (2) C ⇒ (7) E ⇒ (1) G ⇒ (5) K ⇒ (8) R ⇒ (3) X ⇒ (-1) null
```

Avail:
```
(0) ⇒ (9) ⇒ (4) ⇒ (-1) null
```

Results of `remove(List, “X”);`

```
Item  Link
0:  ?   9
1:  G   5
2:  C   7
3:  X   0
4:  ?   -1

List:
(6) A ⇒ (2) C ⇒ (7) E ⇒ (1) G ⇒ (5) K ⇒ (8) R ⇒ (-1) null
```

Avail:
```
(3) ⇒ (0) ⇒ (9) ⇒ (4) ⇒ (-1) null
```

Array of Records: Insert

Sorted (ascending) List Insertion Function

```cpp
bool insert( nodePtr& list, infoType elem) {
    nodePtr prevPtr, currPtr, newPtr;
    newPtr = AllocateNewNode( ); // make new node
    if (newPtr == null)
        return false; // heap is empty
    nodes[newPtr].element = elem;
    nodes[newPtr].link = null;
    prevPtr = null;
    currPtr = list;
    while ((currPtr != null) && 
         (greaterThan(elem, nodes[currPtr].element)) {
        prevPtr = currPtr;
        currPtr = nodes[currPtr].link;
    }
    if (prevPtr == null) { // insert at head or
        nodes[newPtr].link = list; // to empty list
        list = newPtr;
    } else {
        nodes[prevPtr].link = newPtr; // insert in middle
        nodes[newPtr].link = currPtr; // or at tail
    }
    return true; // successful insertion
}
```

Logic is identical to `insert( )` for pointer representation.
Syntax for access to underlying structure has changed.
**Autonomous List Interface Notation**

**Representation Independent Notation**

- Implements low-level generic single linked-list operations
- Achieves independence between list usage levels (information hiding)
  - Changes to list representation or low-level list Fn implementation do not affect High Level list FNs, (e.g., insert, remove, etc.)
  - Changes to list representation or low-level list Fn implementation do not affect user’s code utilizing either low-level or high-level list functions
- Allows sorted, unsorted and other high-level list variations to be implemented independent of underlying representation

**Low Level List FNs**

- Setlink(np1, np2);
- GetLink(np1);
- SetItem(np1, item);
- GetItem(np1, item);
- AllocateNewNode(np1);
- FreeNode(np1);

**ADT List Levels**

**List Applications**

**High Level List FNs**

**Low Level List FNs**

**User Code**

**Autonomous List: Item Interface**

**ItemInterface.h (user supplied)**

- Code file: ItemInterface.cpp
- Linked with ListInterface files

```c
typedef struct {
  . . .
} ItemType;
void AssignItem(ItemType& , ItemType );
bool equalTo(ItemType , ItemType );
bool lessThan(ItemType , ItemType );
bool greaterThan(ItemType , ItemType );
```

**Equality operations**

- User must implement in ItemInterface.cpp and supply through ItemInterface.h

**Content Isolation**

- Item Interface files isolates the list item content information from the list operation implementation.
- Allows List Operation code to be easily reused to implement lists with different types of items.
Autonomous List: List Interface

ListInterface.h (Pointers)

```c
#include "ItemInterface.h"
define null NULL
typedef ItemType ListItem;
typedef struct NodeTag {
    ListItem Item;
    struct NodeTag *Link;
} Node;
typedef Node *NodePointer;
void SetLink( NodePointer, NodePointer );
NodePointer GetLink( NodePointer );
void SetItem( NodePointer , ListItem );
void GetItem( NodePointer , ListItem & );
void AllocateNewNode( NodePointer & );
void FreeNode( NodePointer );
void InitializationForLists( void );
```

ListInterface.h (Array of Records)

```c
#include "ItemInterface.h"
define null -1
typedef int NodePointer;
typedef ItemType ListItem;
typedef struct {
    ListItem Item;
    NodePointer Link;
} Node;
void SetLink( NodePointer , NodePointer );
NodePointer GetLink( NodePointer );
void SetItem( NodePointer , ListItem );
void GetItem( NodePointer , ListItem & );
void AllocateNewNode( NodePointer & );
void FreeNode( NodePointer );
void InitializationForLists( void );
```

Autonomous List: Implementation 1

ListImplementation.cpp (Pointers)

```c
#include <stdio.h>
#include "ItemInterface.h" // ItemType & assignment
#include "ListInterface.h"

void SetLink ( NodePointer N, NodePointer L)
{ N ->Link = L; }

NodePointer GetLink (NodePointer N)
{ return (N ->Link); }

void SetItem ( NodePointer N , ListItem A ) ;
{ AssignItem ( N->Item , A ); }

void GetItem ( NodePointer N, ListItem&  A) ;
{ AssignItem ( A , N->Item ); }

void AllocateNewNode (NodePointer &N ) ;
{ N = new Node; }

void FreeNode ( NodePointer &N ) ;
{ delete N; } // dangling reference if node 
  // pointer not passed by reference

void InitializationForLists ( void ) ;
{ // no initialization for pointer implementation }
```

The line:
```
#include "ItemInterface.h" // ItemType & assignment
```
must precede the line:
```
#include "ListInterface.h"
```
since the typedef ItemType from “ItemInterface.h” is used in “ListInterface.h” and ItemInterface.h is not included in ListInterface.h
ListImplementation.cpp (Array of Records)

```c
#include <stdio.h>
#include "ItemInterface.h" // ItemType & assignment
#include "ListInterface.h"
#define MINPOINTER 0
#define MAXPOINTER 100
NodePointer Avail;
Node Listmemory[MAXPOINTER];

void SetLink ( NodePointer N, NodePointer L)
{ Listmemory[N ].Link = L; }

NodePointer GetLink (NodePointer N)
{ return (Listmemory[N ].Link); }

void SetItem ( NodePointer N , ListItem A )
{ AssignItem ( Listmemory[N ].Item , A ); }

void GetItem ( NodePointer N, ListItem& A)
{ AssignItem ( A , Listmemory[N ].Item ); }

void InitializationFor Lists ( void )
{ NodePointer N;
  for  ( N=MINPOINTER, N<MAXPOINTER-1, N++)
    SetLink ( N , N + 1 );
  SetLink ( MAXPOINTER - 1 , null );
  Avail = MINPOINTER ;
}

void AllocateNewNode (NodePointer & N ) {
  N = Avail ;
  if (Avail != null) {
    Avail =
      GetLink( Avail );
    SetLink( N , null );
  }
}

void FreeNode   ( NodePointer N ) {
    SetLink (N , Avail );
    Avail = N;
}
```

High-Level Implementation: Insert

```c
bool insert( NodePointer& list, ListItem elem) {
    NodePointer prevPtr, currPtr, newPtr;
    ListItem tmp;
    bool slotFound = false;
    AllocateNewNode( newPtr );
    if (newPtr == null)
        return false; // no space available
    SetItem(newPtr, elem);
    SetLink(newPtr, null);
    prevPtr = null;
    currPtr = list;
    while ( (currPtr != null) && ( !slotFound ) ){     GetItem(currPtr, tmp);     if greaterThan(elem, tmp) {
        prevPtr = currPtr;
        currPtr = GetLink(currPtr);
    }
    else
        slotFound = true;
    }
    if (prevPtr == null) {        //insert at head or
    SetLink(newPtr, list);     //       empty list
    list = newPtr;
    }
    else {
        SetLink(prevPtr, newPtr);   //insert in middle
        SetLink(newPtr, currPtr);  //or at tail
    }
    return true;  // successful insertion
}
```
High-Level Implementation: Remove

Remove( ) implemented using low-level list FNs

```c
bool remove( NodePointer& list, ListItem delElem) {
    NodePointer ptr = list, delPtr;
    ListItem tmp;
    bool elemFound = false;
    if ( list == null )
        return false;                    // removal failure
    GetItem(list , tmp)
    if (equalTo(delElem, tmp) {
        list = Getlink( list );          // delete head
        FreeNode ( ptr );
    } else {
        if (Getlink(ptr) == null) return false;
        // perform 1-element look-ahead search
        while( (GetLink(GetLink(ptr)) != null) &&
            (!elemFound ) ) {
            GetItem(GetLink(ptr) , tmp);
            if (equalTo(delElem, tmp)
                elemFound = true;
            else
                ptr = GetLink(ptr);
        } // remove middle or tail node
        GetItem(GetLink(ptr) , tmp));
        if (equalTo(delElem, tmp) ) {
            delPtr = GetLink(ptr);
            Setlink(ptr, GetLink(GetLink(ptr))
            FreeNode ( delPtr );
        } else              //end of list && delElem !found
            return false; // removal failure
    }
    return true; // successful removal
}
```

Alternative FreeNode Management

- **Program specific heap management**
  - Programs can aid the default C++ routines that manage the application heap.
  - Requires maintaining a free node list for each different type of memory cell (node) used in a program.

- **In ListInterface.cc (Pointer Implementation)**

  ```c
  NodePointer freeList;

  void AllocateNewNode (NodePointer &N ) {
    if ( freeList != NULL ) {
      N = freeList;
      freeList = freeList -> Link;
      N ->Link = NULL;
    } else
      N = new Node;
  }

  void FreeNode ( NodePointer &N ) {
    N->Link = freeList;
    freeList = N;
    N  = NULL;
  }

  void InitializationForLists ( void ) {
    freeList = NULL;
  }
  ```

- Removed node is never returned to the system heap.
- Saves heap reallocation steps — can result in substantial runtime performance improvement if node creation and deletion is frequent.
Sorted List ADT Specification

Structure:
- list elements are of ItemType
- specific list property: current position, changed only by ResetList() and GetNextItem()

Definitions provided by client:
- constant MAX_ITEMS
- ItemType specification: data type encapsulating list items
- RelationType specification: enum type consisting of EQUAL, GREATER, and EQUAL
- member function for comparison:
  RelationType ComparedTo(ItemType item);

Function: determines ordering of two objects based on keys
Pre: self and item objects have key members initialized
Post: function value =
  LESS if key of self is less than key of item
  EQUAL if keys of self and item are equal
  GREATER if key of self is greater than key of item

Definitions provided Sorted List ADT:
MakeEmpty()
Function initializes list to empty state
Pre: none
Post: list is empty

More Definitions provided Sorted List ADT:

bool IsFull()
Function determines whether list is full
Pre: list has been initialized
Post: function value = (list is full)

int LengthIs()
Function determines number of elements in list
Pre: list has been initialized
Post: function value = number of elements in list

RetrieveItem(ItemType& item, bool& found)
Function retrieves list element whose key matches item’s key, if present
Pre: list has been initialized; key member of item is initialized
Post: if there is an element, someItem, whose key value matches item’s key, then found = true and item is a copy of someItem; otherwise, found = false and item is unchanged; list is always unchanged.

InsertItem(ItemType item)
Function adds item to list
Pre: list has been initialized; list is not full; item is not in list; list is sorted by key using ComparedTo()
Post: item is in list; list is still sorted
## Sorted List ADT Specification

**More Definitions provided Sorted List ADT:**

**DeleteItem(ItemType item)**

Function: deletes the element whose key matches item’s key  
Pre: list has been initialized; key member of item is initialized; list is sorted by key using function ComparedTo( ); exactly one item in list has a key matching item’s key  
Post: no item in list has a key matching item’s key; list is still sorted

**ResetList( )**

Function: initializes current position for an iteration of the list  
Pre: list has been initialized  
Post: current position is prior to first element in list

**GetNextItem(ItemType& item)**

Function: gets the next item in list  
Pre: list has been initialized; current position is defined; element at current position is not last in list  
Post: current position is updated to next position; item is a copy of element at current position

**Question:** which of the operations in the unsorted list ADT are mutators? observers? constructors?

Note that while Dale provides this ADT for a class-based implementation, a procedural one is just as natural.