Autonomous List Interface Notation

Procedural List ADT Implementation

Representation Independent Notation

NodePointer list

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

Low Level List FNs

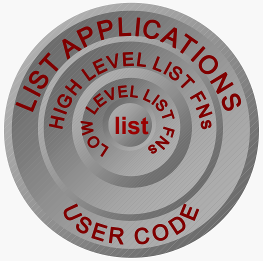
set link of node np1 = np2
return link of node nptr
store item in node nptr
set item = item in node nptr
get a new node
delete a node

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

ADT List Levels



Allows sorted, unsorted and other high-level list variations to be implemented independent of underlying representation

Autonomous List: Item Interface

ItemInterface.h *(user supplied)*

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

```
typedef struct {
    .   .   .
} ItemType;

void AssignItem(ItemType& , ItemType );
```

Equality operations

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

```
bool equalTo(ItemType , ItemType );

bool lessThan(ItemType , ItemType );

bool greaterThan(ItemType , ItemType );
```

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.

Semi-generic list operation code.

unions, generic pointers.

Autonomous List: List Interface

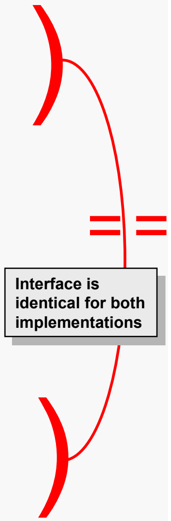
ListInterface.h (pointers)

```
#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 );
```

ListItem is an alias for ItemType

ListInterface.h (Array of Records)

```
#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 );
```



Interface is identical for both implementations

Autonomous List: Implementation 1

ListImplementation.cpp (Pointers)

```
#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 InitializationFor Lists ( void ) ;
   {// no initialization for pointer implementation }
```

The statement:

```
#include “ItemInterface.h” // ItemType &assignment
```

must precede the statement:

```
#include “ListInterface.h”
```

since the typedef ItemType from “ItemInterface.h” is used in “ListInterface.h” and ItemInterface.h is not included in ListInterface.h

Autonomous List: Implementation 2

ListImplementation.cpp (Array of Records)

```
#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 );
   }
}
```

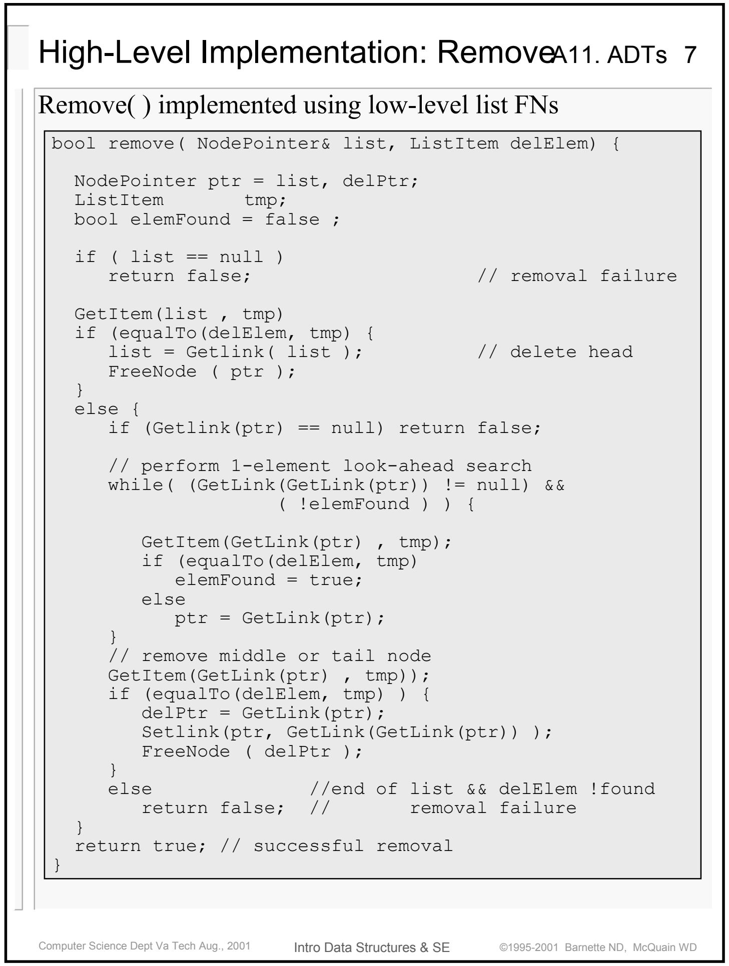
```
void FreeNode
   ( NodePointer N ){
  SetLink (N , Avail );
  Avail = N;
}
```

High-Level Implementation: Insert

Insert( ) implemented using low-level list FNs

```
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
}
```

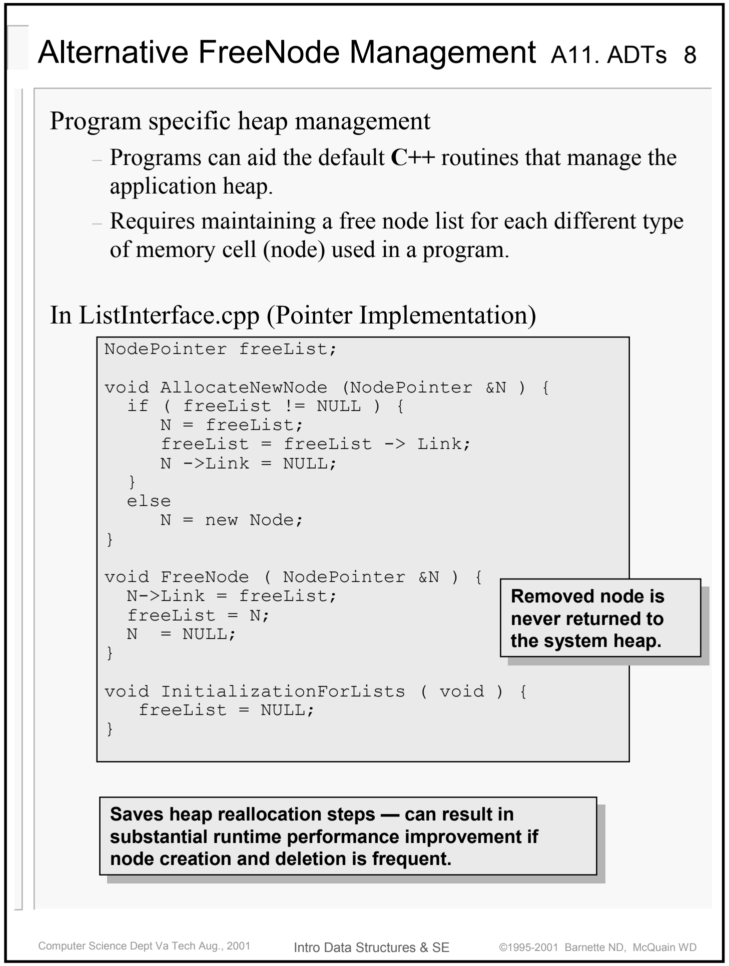
Avoids Boolean short-circuiting



High-Level Implementation: Remove

Remove( ) implemented using low-level list FNs

```
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.cpp (Pointer Implementation)

```
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.