Lists

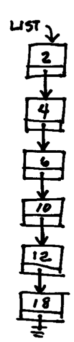
- List - an ordered collection of data items. All locations are available for insertion and deletion. Order is application dependent.
- No limit on number of items.
- No limit on nature of items.

Lists

- The implementation will place limits.
- The ordering here is arbitrary
- Usually A,B,C... or 1,2,3..., etc.
- The client code is responsible for selecting the insert/delete point.

Lists: Alternative Definition

- a set of ordered pairs: a location and a value at that location
- E.g. (5,12), (2,4), (4,10), (1,2), (6,18), (3,6)



Lists: ADT

- Standard List operations
  - Insert
  - Delete
- Data: pointer to start of list; length(?)
- Optional**:** List_Empty
  - List_Copy
  - Display_List
  - Search_List

ADT List Example

ADT List

Data:
Allocate space initialized to blank or empty to hold the data values in the list. A reference initialized to "empty" is needed to the first item on the list. A length parameter initialized to zero is optional.

Methods:
ListEmpty

Input:	*None*
Precondition:	*List has been initialized.*
Process:	*Examines the list to see if there is content*
Postcondition:	*List is unaltered.*
Output:	Returns TRUE if the List *is* empty, otherwise returns FALSE.

ListLength

Input:	*None*
Precondition:	*List has been initialized.*
Process:	*Verifies the number of items in the list*
Postcondition:	*List is unaltered*
Output:	Returns the number of items that are currently in the list.

ADT List Example

ADT List Methods (continued):

ListInsert

Input:	*NewItem is the value to be inserted at position NewPosition*
Precondition:	*List has been initialized. 1 ≤ NewPosition ≤ ListLength+1*
Process:	*If NewPosition ≤ ListLengtb(),* items are *shifted* as *follows: The item* at *Newposition moves to NewPosition+1,* the *item* at *NewPosition+1 moves to NewPosition+2,* and so *on.*
Postcondition:	If insertion is successful. NewXtem is at position NewPosition in the list, other items are renumbered accordingly. Length of list is increased by 1
Output:	*Success indicates whether* the *insertion* was *successful.*

ListDelete

Input:	Position indicates where the deletion should occur.
Precondition:	*List has been initialized.* 1 *<- position ≤ ListLength().*
Process:	*If Position < ListLength(); items are shifted* as *follows: the item at Position+1 moves to Position, the* item at *Position+2 moves to Position+l, and* so *on.*
Postcondition:	*The size of the list is reduced by 1. The positonal location of list entries beyond the point of deletion are renumbered.*
Output:	*The value of the deleted item is returned to the user. Success indicates whether* the *deletion* was *successful.*

ADT List Example

What other methods would be desirable?
This is the point to think about it.

How do we locate insertion or deletion positions?

7

ADT Sorted List Example

ADT SortedList

Data:
Allocate space initialized to blank or empty to hold the data values in the list. A reference initialized to "empty" is needed to the first item on the list. A length parameter initialized to zero is optional.

Methods:
SortedListEmpty

Input:	*None*
Precondition:	*List has been initialized.*
Process:	*Examines the list to see if there is content*
Postcondition:	*List is unaltered.*
Output:	Returns TRUE if the List *is* empty, otherwise returns FALSE.

SortedListLength

Input:	*None*
Precondition:	*List has been initialized.*
Process:	*Verifies the number of items in the list*
Postcondition:	*List is unaltered*
Output:	Returns the number of items that are currently in the list.

8

ADT List Example

ADT SortedList Methods (continued):

SortedListInsert

Input:	*NewItem is the value to be inserted at position NewPosition*
Precondition:	*List has been initialized. 1 ≤ NewPosition ≤ ListLength+1*
Process:	*If NewPosition ≤ ListLengtb(),* items are *moved* as *follows: The item* at *Newposition moves to NewPosition+1,* the *item* at *NewPosition+1 moves to NewPosition+2,* and so *on.*
Postcondition:	If insertion is successful. NewXtem is at position NewPosition in the list, other items are renumbered accordingly. Length of list is increased by 1
Output:	*Success indicates whether* the *insertion* was *successful.*

SortedListDelete

Input:	Position indicates where the deletion should occur.
Precondition:	*List has been initialized.* 1 *<- position ≤ ListLength().*
Process:	*If Position < ListLength(); items are moved* as *follows: the item at Position+1 moves to Position, the* item at *Position+2 moves to Position+l, and* so *on.*
Postcondition:	*The size of the list is reduced by 1. The positonal location of list entries beyond the point of deletion are renumbered.*
Output:	*The value of the deleted item is returned to the user. Success indicates whether* the *deletion* was *successful.*

9

ADT SortedList Example

What other methods would be desirable?
This is the point to think about it.

Are the options affected by the fact the list is maintained in a sorted ordering?

ADT List Example

Another way to think about pre and postconditons:

Axioms for the ADT List
1. (L.CreateList()).ListLength() = 0
2. (L.ListInsert(I,X)).ListLength() = L.ListLength() + 1
3. (L.ListDelete(I)).ListLength() = L.ListLength() - 1
4. (L.CreateList ()).ListEmpty() = TRUE
5. (L.ListInsert(I,Item)).ListEmpty() = FALSE
6. (L.ListEmpty()).ListDelete(I) = error
7. (L.ListInsert(I,X)).ListDelete(I) = L
8. (L.CreateList()).ListRetrieve(I) = error
9. (L.ListInsert(I,X)).List.Retrieve(I) = X
10. L.ListRetrieve(I) = (L.Listinsert(I,X)).ListRetrieve(I+l)
11. L.ListRetrieve(I+1) =(L.ListDelete(I)).ListRetrieve(I)

ADT: Interface Example

public interface List
ublic void ListClass() //constructor
public boolean ListEmpty()
public int ListSize()
public void ListCopy (ListNode List)
//a copy constructor...takes existing list **List**
//and makes a copy of it to initiate a new list
//may require modifying **interface** or ADT
public Datatype ListDelete(**int** Rank)
public void ListInsert (Datatype Item, **int** Rank)
} //end List Interface

ADT: Interface Example

Later when writing code…

public class ArrayList **implements** List

OR

public class ListClass **implements** List

Lists: Array Implementation

☹Limits size (static)

😐Requires list to be homogenous

☺Random access - Exploit!

- Lists are stored in an array in an obvious fashion.
- Inserting requires shifting to make room, e.g. insert B.
  - On average, **O**(n/2)

Lists: Array Implementation

- Deleting requires shifting to close gap in list, e.g. delete E
  - On average, **O**(n/2)
- If list contains > k items then can exploit random access to find k^{th} value in list at location k-1 of array
- Simple, <u>standard</u> Implementation

Lists: Array Implementation

- Size = 5
- Array[size - 1]
  is end of List

0	A
1	C
2	D
3	E
4	F
5	
6	
7	
8	
9	
10	

Lists: Alternate Array Implementation

- No more shifting when
  deleting
- Mark the deleted spot
  - **O**(1)
- Use flag value -
  Can you guarantee
  that data will never
  look like flag value??

0	A
1	ZZZZ
2	D
3	E
4	ZZZZ
5	G
6	ZZZZ
7	I
8	J
9	ZZZZ
10	ZZZZ

Lists: Alternate Array Implementation

- OR use flag field
- Requires 1+ more bit per item on list
- When inserting, shift only from insertion point to marked deleted slot.
- Shifting will be reduced.

<< **O**(n/2) **J3??**

0	1	A
1	1	B
2	1	D
3	1	E
4	0	F
5	1	G
6	~~0~~ 1	~~H~~ G2
7	1	I
8	1	J
9	~~0~~ 1	~~K~~ J2
10	1	L

Lists: Alternate Array Implementation

- How complicated do you make it?
- How important is it to avoid false overflow?
- Item k on the list is no longer at location k-1. You cannot exploit random access.
- If more deletions than insertions, then list could evolve to have many deleted items. Then, waste time looking at deleted items when processing list.
- Inefficient.

Lists: Alternate Array Implementation

0	0	A
1	1	B
2	0	C
3	0	D
4	0	E
5	1	F
6	1	G
7	0	H
8	0	I
9	0	J
10	1	K
11	0	L
12	0	M
13	1	N
14	0	O
15	0	P

List contains:

B, F, G, K,N

Lists: Alternate Array Implementation

- List can be reorganized at time system is not being used
- Lengthy reorganization time will not matter.
- Garbage collection is a standard operating system process.
- Reference: Horowitz and Sahni

Lists: Linked implementation

☺No size limits (dynamic allocation)
😐Requires list to be homogenous
☹Sequential access

LIST Code

A simple Linked implementation

```
class ListNode {
   DataType Data;           //any appropriate type
   ListNode  Next;
}                    //Default constructor is ListNode()
                      //You could define methods like
                     //GetData and SetData if desired
```



```
public class ListClass implements List{
   private ListNode List;
   private int size;      //optional size parameter

public void ListClass() {                //constructor
   List = null;
   size = 0;                                  //if used;
}

public boolean ListEmpty() {
      return (List == null);       //or examine size
}
```

```
public int ListSize() {
   return size;
}

public void ListCopy (ListNode List) {
   //a copy constructor...takes existing list List
   //and makes a copy of it to initiate a new list
   //may require modifying interface or ADT
   ...exercise for the student...
}
```



```
// 2 private functions to simplify working with list
private boolean ValidNode (int Rank) {
   return ((Rank >= 1) &&  (Rank <= Size ))
}                  //Checks that Rank is within extent of List

private ListNode PtrTo (int Rank) {
   if ValidNode (Rank) {
      ListNode Here = List  //Set temp ptr to head of list
      for ( int i = 1, i < Rank, i++)
         Here = Here.next;
      return Here;
   }
   else return null;                         //invalid request
}   //end PtrTo
```



```
public Datatype ListDelete(int Rank) {
   ListNode Temp;
   if !(ValidNode(Rank)) "error"                  //do error handling
   else {                                          //Valid Request
      if (Rank == 1) {                           //Delete at head of list
         Temp = List;                          //Grab node for deletion
               List = Temp.Next;                     //Update list ptr
         }
      else {                                       //generic deletion
         ListNode After = PtrTo (Rank-1);          //Ref to prev node
         Temp = After.Next;                     //Grab node for deletion
            After.Next = Temp.Next;              //Connect head &  tail
        }
     Size = Size -1;                               //Update size
     Temp.Next = null;                  //disconnect node from list
   return Temp.Data;                        //Return deleted value
     }   //end Valid Request
}   //end ListDelete
```

```
public void ListInsert ( Datatype Item, int Rank ) {
//assumes error checking on Rank done

    ListNode Temp = new ListNode;              //allocate space
    Temp.Data = Item;                          //stuff in information
    Size = Size + 1;                           //Update size
  if (Rank == 1) {                             //Insert at head of list
     Temp.Next = List ;                        //Connect new node
     List = Temp;                              //Update list ref
  }
  else {                                       //generic insertion
       ListNode After = PtrTo (Rank-1);        //Get ref to prev node
       Temp.Next = After.Next;                 //Connect to tail
       After.Next = Temp;                      //Connect head to node
  }
}   //end ListInsert
} //end ListClass
```