Lists

- A list is a varying-length, linear collection of homogeneous elements.
- Linear means each list element (except the first) has a unique predecessor, and each element (except the last) has a unique successor.
- In a linear structure, components can only be accessed sequentially one after the other.

Lists (cont.)

- It normally has two "special" named elements called head and tail. They point to the first and last element of the list.

Lists as ADTs

- Domain
  - A collection elements of the same type
  - An implicit list cursor in the range 1 through n+1 where n is the current length of the list.
- Operations
  - create(): create a new list. Should be done using constructors
  - Boolean IsEmpty() & Boolean IsFull(): returns true if the list is empty and full respectively
  - void InsertBeginning(v) & void InsertEnd(v): Insert the value v either in the end or the beginning of the list
  - void Delete(): item at list cursor deleted
  - print(): Print the whole list

More List Operations

- Operations
  - insertAfter(v, anotherV): insert the value v after the first occurrence of anotherV.
  - insertBefore(v, anotherV): Insert the value v before the first occurrence of anotherV.
  - search(v): search for the first occurrence of v in the list and return its position. Could be used to help the implementation of several other operations
  - deleteAll(v): Delete all occurrences of v in the list
  - reset(): List cursor is at front of list.
- We can assume that private data in a list includes the lists itself (using an array or linked list), the head, the tail and the size of the list.

Array-based class List

Private data:
- length
- data [0] [1] [2] [MAX_LENGTH-1]

- IsEmpty
- IsFull
- Length
- Insert
- Delete
- IsPresent
- Print
- SortedList
// ARRAY-BASED LIST (list.h)
const int MAX_LENGTH = 50;
typedef int ItemType;

class SortedList {
public:
    // public member functions
    SortedList();  // constructor
    bool IsEmpty() const; // returns length of list
    bool IsFull() const;
    int Length() const;
    void Insert(ItemType item);
    void Delete(ItemType item);
    bool IsPresent(ItemType item) const;
    void Print();

private:
    // private data members
    int length;  // number of values currently stored
    ItemType data[MAX_LENGTH];
    void BinSearch(ItemType item, bool& found, int& position) const;
};

How to Implement a List

- use a built-in array stored in contiguous memory locations, implementing operations Insert and Delete by moving list items around in the array, as needed
- use a linked list (to avoid excessive data movement from insertions and deletions) not necessarily stored in contiguous memory locations

List Implementation via Arrays
create()

List Implementation via Arrays
insertEnd(40)

List Implementation via Arrays
insertEnd(40)

List Implementation via Arrays
insertBeginning(22)
List Implementation Via Arrays

insertBeginning(22)
List Implementation via Arrays

**insertBeginning(22)**

```
if list is not full
for i=(tail-1) to 0
    list[i+1] = list[i]
list[0] = v
size++
tail++
```

**delete(33)**

```
if list is not full
for i=(tail-1) to 0
    list[i+1] = list[i]
list[0] = v
size++
tail++
```
How to Implement a List

- use a **built-in array** stored in contiguous memory locations, implementing operations Insert and Delete by moving list items around in the array, as needed
- use a **linked list** (to avoid excessive data movement from insertions and deletions) not necessarily stored in contiguous memory locations

Linked Lists vs. Arrays

- Lists differ from arrays because they don’t have a fixed size but like arrays can only store elements of the same type
- Main advantage over arrays is easy insertion and deletion of nodes
- A well defined list may be the basis for the implementation of several other data structures, such as queues, stacks, trees and graphs.
Self-referential data types

```java
class Node {
    private Object info;     // the “info”
    private Node next;      // the “link”
}
```

**A Linked List**

- A linked list is a list in which the order of the components is determined by an explicit link member in each node.
- The nodes are structs—each node contains a component member and also a link member that gives the location of the next node in the list.
- An external pointer (or head pointer) points to the first node in the list.

```
head 10 8 50 /
```

**Nodes can be located anywhere in memory**

The link member holds the memory address of the next node in the list.

```
Memory address
3000 10 5000 8 2000 50 NULL
```

**Declarations for a Linked List**

```c
// Type DECLARATIONS
struct NodeType {
    int info;
    NodeType* next;
};
```

```c
// Variable DECLARATIONS
NodeType* head;
NodeType* ptr;
```

**Member Selection**

- `ptr` points to a node.
- `(*ptr).info` accesses the info member.
- `ptr->info` accesses the info member.

```c
ptr = head;
while (ptr != NULL) {
    cout << (*ptr).info << endl;
    // Or, do something else with
    ptr = ptr->next;
}
```

**Traversing a Linked List**

```
ptr 3000 10 5000 8 2000 50 NULL
```

```
ptr = head;
while (ptr != NULL) {
    cout << ptr->info << endl;
    // Or, do something else with
    ptr = ptr->next;
}
Traversing a Linked List

```c
node *ptr;
ptr = head;
while (ptr != NULL) {
    cout << ptr->info;
    // Or, do something else with
    ptr = ptr->next;
}
```

//PRE: head points to a linked list
Traversing a Linked List

```cpp
//PRE: head points to a linked list
node *ptr;
ptr = head;
while (ptr != NULL) {
    cout << ptr->info;
    // Or, do something else with
    ptr = ptr->next;
}
```

Using Operator new

If memory is available in an area called the free store (or heap), operator new allocates the requested object, and returns a pointer to the memory allocated.

The dynamically allocated object exists until the delete operator destroys it.
Inserting a Node at the Front of a List

```c
int item = 6;
Node *location;
location = new NodeType;
location->info = item;
location->next = head;
head = location;
```

Diagram:
- `head` points to the new node with value 6.
- The new node is inserted at the front of the list.
Using Operator delete

- The object currently pointed to by the pointer is deallocated, and the pointer is considered undefined. The object's memory is returned to the free store.

Deleting the First Node from the List

```c
Node *tempPtr;

item = head->info;
tempPtr = head;
head = head->next;
delete tempPtr;
```

Deleting the First Node from the List

```
head    6  5  8  3
```

Deleting the First Node from the List

```
item    6
Node   *tempPtr;

item = head->info;
tempPtr = head;
head = head->next;
delete tempPtr;
```

Deleting the First Node from the List

```
item    6
Node   *tempPtr;

item = head->info;
tempPtr = head;
head = head->next;
delete tempPtr;
```

Deleting the First Node from the List

```
head    6  5  8  3
```

Deleting the First Node from the List

```
item    6
Node   *tempPtr;

item = head->info;
tempPtr = head;
head = head->next;
delete tempPtr;
```

Deleting the First Node from the List

```
item    6
Node   *tempPtr;

item = head->info;
tempPtr = head;
head = head->next;
delete tempPtr;
```

Deleting the First Node from the List

```
head    5  8  3
```

Deleting the First Node from the List

```
item    6
Node   *tempPtr;

item = head->info;
tempPtr = head;
head = head->next;
delete tempPtr;
```
class PersonList
{
    Private data:
    head
    currPtr

    Public Methods:
    ~PersonList
    IsEmpty
    Advance
    InsertAfter
    PersonList
    InsertBefore
    PersonList
    CurrentRec
    Delete

    Boolean PersonList::IsEmpty() const
    // POST: FCTVAL == ( list is empty )
    {
        return (head == NULL);
    }

    Boolean PersonList::EndOfList() const
    // POST: FCTVAL == ( list cursor is beyond end of list )
    {
        return (currPtr == NULL);
    }

    void PersonList::Advance()
    // PRE: NOT IsEmpty() && NOT EndOfList()
    // POST: List cursor has advanced to next record
    {
        currPtr = currPtr->next;
    }

    PersonRec PersonList::CurrentRec() const
    // POST: FCTVAL == record at list cursor
    {
        PersonRec rec;
        rec.name = currPtr->name;
        rec.age = currPtr->age;
        return rec;
    }
}

void PersonList::InsertAfter ( PersonRec someRec )
// PRE: Assigned (someRec) && NOT IsEmpty ()
// && NOT IsFull () && NOT EndOfList ()
// POST: someRec inserted after list cursor
// && This new node has become the current record
{
   // obtain and fill a node
   PersonNode *ptr = new PersonNode ;
   ptr->name = new char [ strlen ( someRec.name ) + 1 ] ;
   strcpy ( ptr->name, someRec.name ) ;
   ptr->age = someRec.age ;
   ptr->next = currPtr->next ;
   currPtr->next = ptr ;
   currPtr = ptr ;
}
ptr->age = someRec.age;

ptr->next = currPtr->next;

currPtr->next = ptr;

currPtr = ptr;