Abstract Data Type

- A data type for which:
  - only the properties of the data and the operations to be performed on the data are specific,
  - how the data will be represented or how the operations will be implemented is unspecified.
- An ADT may be implemented using various specific data types or data structures, in many ways and in many programming languages.
- Examples:
  - Stacks and Queues (implemented using arrays+LL)
  - string class (not sure how it’s implemented)

The Abstract List Data Type

- A List is an ordered collection of items of some type T:
  - each element has a position in the list
  - duplicate elements are allowed
- List is not a C++ data type. It is conceptual. It can be implemented in various ways
- We have implemented it using a linked list (NumberList).
- Now we are going to use an array to implement the list.

Common List operations

- Basic operations over a list:
  - create a new, empty list
  - append a value to the end of the list
  - insert a value within the list
  - delete a value (remove it from the list)
  - display the values in the list
  - delete/destroy the list (if it was dynamically allocated)
Declaring the List data type

- We will be defining a class called NumberList to represent a List data type.
  - ours will store values of type double, using an array.
- The class will implement the basic operations over lists on the previous slide.
- In the private section of the class we will:
  - define an array of double to store the elements in the list.
  - define a count variable that keeps track of how many elements are currently in the list.

NumberList class declaration

```cpp
class NumberList {
private:
  static const int SIZE = 100;
  double array[SIZE];
  int count;
public:
  NumberList(); // creates an empty list
  ~NumberList(); // not needed, no dynamic allocation
  void appendNode(double);
  void insertNode(double);
  void deleteNode(double);
  void displayList();
};
```

- This has the same public interface as it does when using linked lists.

Operation: Create the empty list

- Constructor: sets up empty list

```cpp
#include "NumberList.h"
NumberList::NumberList()
{
  count = 0;
}
```

Operation: append value to end of list

- appendNode: adds new value to end of list
- Algorithm:
  Make sure the list isn’t full.
  Put new element in array at position count.
  Increment count.

```cpp
void NumberList::appendNode(double num) {
  if (count < SIZE) {
    array[count] = num;
    count++;
  } else
    cout << "Error: cannot append value, list is full"
    << endl;
  //maybe we should add isFull/isEmpty?
}
```
Operation: **display** the list

- Use a for loop
- Stop at count, not SIZE

```cpp
void NumberList::displayList() {
    for (int i=0; i<count; i++) {
        cout << array[i] << " ";
    }
    cout << endl;
}
```

**delete** a node from the list

- deleteNode: removes a given value from list
- We need to shift elements over to fill the gap.

Deleting 13 from the list

```
1 4 7 13 17 25
```

Count: 6

```
1 4 7 17 25 25
```

Count: 5

**deleteNode** code

```cpp
void NumberList::deleteNode(double num) {
    int i=0;
    while (i<count && array[i]!=num) {
        i++;
    }
    if (i<count) {  //found at i
        count--;
        //shift left to close gap
        while (i<count) {
            array[i] = array[i+1];
            i++;
        }
    }
}
```

**insert** a value into a list

- Inserts a new value into the middle of a list.
- We'll assume the list is sorted, and insert before first number greater than this value.
- We need to shift elements over to produce a gap.

Inserting 15 into the list

```
1 4 7 13 17 25
```

Count: 6

```
1 4 7 13 15 17 25
```

Count: 7
void NumberList::insertNode(double num) {
    // keep the list sorted
    int i = 0;
    while (i<count && array[i]<num) {
        i++;
    }
    count++;
    // shift right to open up a spot in the array
    int j = count-1;
    while (j>i) {
        array[j] = array[j-1];
        j--;
    }
    array[i] = num;
}

int main() {
    // set up the list
    NumberList list;
    list.appendNode(2.5);
    list.appendNode(7.9);
    list.appendNode(12.6);
    list.displayList();
    list.insertNode(8.5);
    list.displayList();
    list.insertNode(1.5);
    list.displayList();
    list.insertNode(21.5);
    list.displayList();
    // continued on next slide
}

cout << "remove 7.9: " << endl;
list.deleteNode(7.9);
list.displayList();
cout << "remove 8.9: " << endl;
list.deleteNode(8.9);
list.displayList();
cout << "remove 2.5: " << endl;
list.deleteNode(2.5);
list.displayList();
cout << "remove 12.6: " << endl;
list.deleteNode(12.6);
list.displayList();
}

linked lists vs arrays: space issues

- Linked list is never full (if there’s more memory)
  - For arrays we need to predict the largest possible size.
- The amount of memory used to store the linked list version is always proportional to the number of elements in the list (it grows+shrinks)
  - For arrays, the amount of memory used is often much more than is required by the actual elements in the list.
- Arrays do not require extra storage for links
  - linked lists are impractical for lists of characters or booleans (pointer value is bigger than data value).
linked lists vs arrays: time issues

- When a value is inserted into or deleted from a linked list, none of the other nodes have to be moved.
  - Array elements must be shifted to make room or close a gap.

- Arrays allow random access to elements: array[i]
  - for arrays this is pointer arithmetic
  - linked lists must be traversed to get to i'th element.