

Data Structures (Chapter 6 and Part IV)

There are several well-known data structures that every software engineer uses. The most basic ones are: the **container** data structures: the List, the Stack, and the Queue. Containers hold objects of any type and provide standard operations for inserting and removing objects from the container. More interesting data structures are Trees (various kinds), Graphs, and Hash Tables.

There are two design aspects to every data structure:

1. the interface part
2. the implementation part

The interface defines an **abstract data type** (ADT), which provides function declarations for all the operations that the data structure supports; for example, creation, assignment, insert element, remove element, and destruction.

The choice of implementation can and should be made independent of the interface. There can be more than one implementation choice for an interface. The implementation defines how the ADT operations are implemented in terms of the representation that is chosen for the data structure.

As we have already seen, C++ provides language features that can be used to realize a data structure by defining a separate interface and implementation classes, and then tie them together. C++ provides the programmer with templates, inheritance, and virtual functions as language features that allow us to define and construct data structures in a flexible and reusable manner.

Lists

There are severe disadvantages of the fixed size list if it turns out you need a bigger one:

1. An array cannot be extended dynamically, you have to allocate a new array of the appropriate size and copy the old array to the new array:

```
int vector[10];
...
int* new_vector = new int[100];
for (int i = 0; i < 10; i++)
    new_vector[i] = vector[i];
```

Alternatively, you could have allocated the original vector as having 100 elements, in anticipation of the future need for more than 10 elements.

2. If you want to insert/remove an element to/from a fixed position in the list, then you must move elements already in the list to make room. On average, you probably copy half the elements. The worst case is that you insert into position 1, and have to move all the elements. Copying elements can result in longer running times for a program if insert/remove operations are frequent, especially when you consider the cost of calling operator= on some object that does a deep copy (e.g., the String object).

```
String names[100];
...
insert_nth(String names[], int n, String x)
{
    for (int i = 99; i > n; i--)
        names[i] = names[i-1]; // calls String::operator=(const String& s)
    names[n] = x;
}
```

Lists (Chapters 6 & 16)

The most common data structure is the list. There are two general types:

1. the array or vector
2. the linked list

The array has a fixed size, which works fine when the “thing” that are to be put into the list are fixed in number.

```
int vector[10]; // an array of 10 integers
String names[100]; // an array of 100 names
```

The advantages of the array is that storage for it is allocated at compile-time, it is simple to access using an integer index, and we can easily *iterate* through an array with a for loop:

```
for (int i = 0; i < 10; i++)
    vector[i] = 0;
for (int j = 99; j >= 0; j--)
    names[j] = "";
```

Iteration is the process of stepping through a list one element at a time. We can do forward iteration, or backward iteration, by simply arithmetic on the array index. We can also “randomly” access an element of the array in constant time, as long as we use a valid index value that does not exceed the bounds of the array (note: C++ does not perform bounds checking when accessing an array).

```
vector[0] = 99;
vector[10] = -1; // "silent" ERROR, access beyond the end of the vector
```

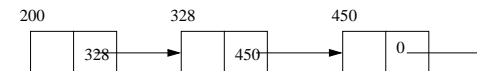
Linked Lists

The linked list is a flexible alternative to the array or vector, with the extra “cost” or more storage usage per list element, called a **list node**, because of the need to keep a pointer to the next node in the list.

Each node is dynamically allocated from the heap, so the node has a unique address. A linked list is formed by having each node contain a “next” pointer that has the address of the next node in the list.

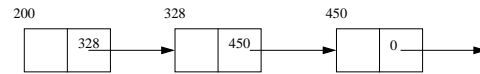
The last node in the list has a next pointer with the value NULL, indicating the end of the list.

Question: How do you insert/remove a node from the list?

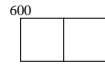


Linked List Node Insertion/Removal

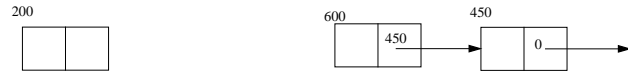
If a new node is allocated from the heap, and is to be inserted as the 3rd node in the list, what operations are required on the existing list? What about removing a node from the list? Are there any special cases to consider? What is the “cost” of insertion/removal?



Insert node 600



Delete node 328



List Interface

A list interface should define a container for any type T, with a well-defined set of operations for inserting and removing elements from the container. When designing the list interface, try to think of the typical usage of a list container, and provide operations that will allow such usage.

```
template <class T> class List {
public:
    List();
    List(const List<T>& l);
    ~List()

    List<T>& operator=(const List<T>& l);

    void insert_front(const T& elem);
    T* remove_front();

    void insert_back(const T& elem);
    T* remove_back();

    void insert_nth(int n, const T& elem);
    T* remove_nth(int n);

    void reverse();

    void merge(const List<T>* l);
    ...
};
```

What about an operation to sort the elements in a list? What do you need to know in order to sort the elements in a list?

Linked List Iteration

How do we iterate over a linked list?

1. We need a pointer to the first node.
2. We terminate the iteration when the next node is NULL.
3. We step through the list by following the next pointer.

```
T* list; // pointer to start of the list
for (T* p = list; p != NULL; p = p->next)
```

Question: Why doesn't pointer arithmetic work in this case?

```
T* list;
for (T* p = list; p != NULL; p++)
    ...
```

Can you think of an object oriented way to iterate over a list? I.e., and Iterator interface independent of the list implementation. What if you defined an Iterator class, that had methods for returning the first node in a list, checking for the end of the list, and returning a pointer to the next node.

```
template<class T> class Iterator {
public:
    Iterator(List<T>* list);

    T* begin();
    T* end();
    T* next();
};

Iterator<T> i = list;
for (T* p = i.begin(); p != i.end(); p = i.next())
    ...
```

A List Implementation

We have a choice to make: do we implement the list as a fixed size array or a linked list?

The answer depends on how the list will be used. The most generally useful kind of list is a linked list, since it is not constrained by having a fixed number of elements, but there is some extra cost associated with the overhead of maintaining pointers, and traversing the list by following the “next” pointer.

The first thing we need is to define the representation of a list node, and a pointer to the first node in the list.

```
template<class T> class List
public:
    ...
private:
    struct Node {
        T* elem;
        Node* next;

        Node() : elem(NULL), next(NULL) {}
        Node(const T& e, Node* n = NULL) : next (n) { elem = new T(e); }
    };

    Node* head;
```

Given this choice of a representation for the linked list, we next have to implement all the operations.

Programming Assignment #3: Implement a List template and all the methods defined in the interface. You should have a List.h header file defining the interface and a List.C (List.cpp) class defining the implementation. Also define an Iterator class for the List<T> class, so that it can be used to iterate over a linked list.