

# Chapter 29 – Unordered Lists

## 1. What is a List?

Lists are common and are easy to recognize. We add items to a grocery list. We make a list of things that need to be repaired or serviced around the house.

Basically, a list is a sequential collection of elements of similar type. Elements of a list may be accessed sequentially, that is one at a time. Lists are among the most often-used data structures in computer programs.

Lists can be represented in various ways. The text files from previous chapters are examples of lists.

When a list is needed in memory, it is often stored in an array. This brings up an interesting condition where the length of a list is distinct from the length of the array that contains it. Of course, the length of the list cannot be longer than the length of the array.

You have probably realized by now that you have been working with lists since files were introduced. This chapter exists to review and formalize the discussion of lists in arrays and to include the analysis of worst case growth in time and space of list algorithms and data structures.

## 2. Inserting into an Unordered List of Unknown Length

Lists are typically populated by interactive user input, input from some device or from a file. If the list is stored in an array and there is no intrinsic order to the elements, each element added to the list is added to the back of the list, but only if there is room in the array or if the array is dynamic.

### Static Array

Here is an example that simply stops loading if the array is too small to contain the list:

```
const int max = 1000;
double array[max];
int listlen = 0;
ifstream infile;
infile.open("data.dat", ios::in);
infile >> array[listlen];
while (infile && listlen < max) {
    listlen++;
    infile >> array[listlen];
}
infile.close();
```

After the while loop terminates, *array* contains *listlen* number of elements of the list. All activity on the list must be within the indices 0 to (*listlen*-1).

**Analyses on Time:** Assuming that the array is large enough to hold the list and that the list is  $n$  elements in length, the most frequently performed line in the algorithm,

```
while (listlen && listlen < max)
```

will be performed  $n+1$  times. This gives a worst case growth in time of  $O(n)$ .

**Analyses on Space:** Since the array is static and does not grow, the growth rate of the data structure in space is  $O(1)$ .

### Dynamic Array

The following is an example of loading the elements of a list into a dynamic array.

```
int max = 100;
double * array;
array = new double[max];
double * temp;
int listlen = 0;
ifstream infile;
infile.open("data.dat", ios::in);

infile >> array[listlen];
while (infile) {
    listlen++;
    if (listlen >= max) {
        temp = new double[max+100];
        for (int x=0; x<max; x++) temp[x] = array[x];
        delete [ ] array;
        array = temp;
        max+=100;
    }
    infile >> array[listlen];
}
infile.close();
```

As in the previous example, when the while loop terminates *array* contains *listlen* number of elements of the list and all activity on the list must be within the indices 0 to (*listlen*-1).

*Note: In later chapters, we will see how to store lists as dynamic lists rather than in arrays. Length of the list will not be an issue when using dynamic lists.*

**Analysis on Time:** If the number of elements in the list is big enough, the most frequently performed statements will be the nested loop *for (int x=0; x<max; x++)*, which will have to traverse the entire length of the array each time *listlen* meets or exceeds *max*. Since the loop the *for* loop is nested in is performed  $n$  times, the *for* loop will be started ( $n/100$ ) times. If  $n$  is big enough, even dividing by 100 will not significantly affect the growth in time of this portion of the algorithm. In the worst case, the *for* loop

itself will take  $(n+1)$  repetitions to transfer all of the elements of array into temp. This yields an analysis of  $n * (n + 1) = n^2 + n$ . Taking the most effective term (the largest power term), this yields a worst case growth in time of  $O(n^2)$ .

**Analysis on Space:** The array *array* will achieve a worst case size of  $(n+100)$ . The array temp will achieve a worst case size of  $n$ . Adding the two results gives  $2n+100$ . Taking the term of the highest power yields a worst case growth in space for the algorithm of  $O(n)$ .

### 3. $N^2$ Searching of an Unordered List

Having searched an unordered list in the past, you know that in some cases you must search until the end of the list regardless of the value being found or not and in other cases you may stop searching when the correct value is found. For example, when the item being searched for is not in the list or when searching for the min/max value, every element in the list must be checked.

The classic example definition of searching an unordered list is as followings. Notice that the algorithm returns  $-1$  (an impossible index) if the element is not found.

```
int search(elementtype array[ ], int listlen, elementtype item) {
    index = 0;
    while (index < listlen && item != array[index]) index++;
    if (index < listlen)
        return index;
    else
        return -1;
}
```

If the function *search* returns a value equivalent to *listlen*, the value *item* was not found. If the function returns a value less than *listlen*, the value returned is the index giving the location of the value in the array equal to *item*.

A slightly more efficient way to write this search is to assign the item being searched for to the element at index *listlen*. This allows the first test the Boolean expression of the while loop to be dropped, just relying on the second to stop the loop in any event. This improves performance because only one simple Boolean expression need be evaluated each pass through the loop.

```
int search(elementtype array[ ], int listlen, elementtype item) {
    index = 0;
    array[listlen] = item;
    while (item != array[index]) index++;
    if (index < listlen)
        return index;
    else
        return -1;
}
```

**Analysis on Time:** It is easy to see that the most frequently performed statement in the search is the *while* statement. If the item is not found, the *while* statement will be executed  $(n+1)$  times. However, this does not mean that a search of an unordered list yields a worst case growth in time of  $O(n)$ . The number of times that the search algorithm is initiated must be considered. In other words, if the number of times that a search of an unordered list is called is represented by  $m$ , the worst case growth in time of a search of an unordered list could be expressed as  $O(nm)$ . But, even this is not final. If  $n$  and  $m$  are big enough, the difference between  $n$  and  $m$  becomes minor enough for  $n$  and  $m$  to be considered equal. In this case both values can then be represented by the same variable,  $n$ . These gives an actual worst case growth in time analysis of searching an unordered list of  $O(n^2)$ .

#### **4. Deleting from an Unordered List**

Deleting from an unordered list stored in an array is accomplished in the following steps:

1. find the index of the element to be deleted
2. if the element to be deleted is found, starting from the index of the element to be deleted + 1, copy each element from its index to its index-1
3. subtract 1 from the list length variable

The following is an example of deleting from an unordered list:

```
void deleteFromList(elementtype array[ ], int & listlen, int item) {
    int index = search(array, listlen, item);
    if (index < listlen) {
        for (int x=index+1; x<listlen; x++)
            array[x-1]=array[x];
        listlen--;
    }
}
```

Note that *listlen* is passed by reference in this example.

#### **Exercise 29.1**

Analyze the algorithm in `deleteFromList` for worst case growth in time.

### **Programming Assignment 29.1**

The following program simulates the temperatures (highs and lows) of each day in a year by exaggerating the output of the sine function.

```
#include <iostream>
#include <fstream>
#include <stdlib.h>
#include <conio.h>
#include <time.h>
#include <cmath>
using namespace std;

int main() {
    double sinout,hi,low;
    srand((unsigned)time(NULL));
    ofstream outfile;
    outfile.open("data.dat", ios::out);
    for (double x=0.0; x<6.28; x+=(6.28/360.0)) {
        sinout = sin(x) * 50.0 + 50.0;
        hi = sinout + 10 + (rand() % 10);
        low = sinout - 10 + (rand() % 10);
        cout << hi << " " << low << endl;
        outfile << hi << " " << low << endl;
        cout.flush();
        getch();
    }
    outfile.close();
    return 0;
}
```

Implement and run the above program. Write a program to analyze the data in the file created by the above program for the

- minimum low for the year
- minimum high for the year
- maximum low for the year
- maximum high for the year
- average temperature for each day
- average low for the year
- average high for the year
- average temperature for the year

**Programming Assignment 29.2**

Write a program that accepts numeric grades and stores them in an array. When the user enters a grade of -1, no more grades will be entered. The program should then find the lowest grade and delete it from the list. Having done this, the remaining grades and the average of the remaining grades should be output.

**Programming Assignment 29.3**

Include the public function `deleteCard()` in a class *phonebook* of programming exercise 26.2.

```
void deleteCard( ); // calls findcards to display all cards with a given last name with
                   // the index of each card; user selects an index for a card, then
                   // the card is deleted or the user selects cancel
```

Alter the class *phonebook* and the main of 26.2 to allow the user to chose to delete items in the list.