Linear Searching, Reading

Search, also known as "find", is traversing over a data structure to find a match. A search may result in a Boolean true or false, like our AssociativeArray::containsKey, or it may result in returning the **V value** for a matching **K key**, like our AssociativeArray::operator[ ]. Or in the case of arrays, it may return the index where the match was found.

A "linear search" is one that starts at the first value in an array or linked list, and traverses it one value at a time in search of a match, using equals-equals for the compare. The compared data types either have to be defined in the C++ language (like ints and chars) or one of its libraries (like strings), or in the case of programmer-designed objects, the object has to have an operator== getter function or stand-alone function, like this:

```
struct Student
{
  string name;
  int ID;
};

bool operator==(const Student& a, const Student& b)
{
  return a.ID == b.ID;
}
```

```
struct Time
{
  int hr;
  int mn;
};

bool operator==(const Time& a, const Time& b)
{
  return 60 * a.hr + a.mn == 60 * b.hr + b.mn;
}
```

Searching is normally controlled by a for-loop, with a return or break, if and when a match is found.

```
int i;
for (i = 0; i < n; i++)
  if (a[i] == matchThis)
    break;

if (i == n) // not found
else // match at i
```

```
Node* p;
for (p = firstNode; p; p = p->next)
  if (p->value == matchThis)
    break;

if (p) // match at p
else // no match
```

Each of these loops has these in common:

1. The loop index or pointer is declared *above* the loop so that it survives below the loop.
2. The loops exit with a break instead of a return, or when they reach the end.
3. Expression of the result takes place below the loop based in the loop index or pointer value.

These constructions are the most versatile of all possible constructions, because they can be adapted to returning true or false, or a value or a dummy, or an index.

Linear Searching A Sorted Array Or Linked List

A variation is applying the search loop to a *sorted* array or linked list, because once the loop finds a value *greater* than the value to match, there's no point in continuing the search. The coding is just a bit more complicated:

```
int i;
bool found = false;
for (i = 0; i < n && !found; i++)
{
  found = a[i] == matchThis;
```

```
Node* p;
bool found = false;
for (p = firstNode; p != 0 && !found; p = p->next)
{
  found = p->value == matchThis;
  if (found || matchThis < a[i])
    break;
}

if (found)
else // not
```

```
  if (found || matchThis < p->value)
    break;
}

if (found)
else // not
```

Of course, this does put the added requirement on the ADT being compared that it support less-than comparison *in addition to* equals-equals:

```
struct Student
{
  string name;
  int ID;
};

bool operator==(const Student& a, const Student& b)
{
  return a.ID == b.ID;
}

bool operator<(const Student& a, const Student& b)
{
  return a.ID < b.ID;
}
```

```
struct Time
{
  int hr;
  int mn;
};

bool operator==(const Time& a, const Time& b)
{
  return 60 * a.hr + a.mn == 60 * b.hr + b.mn;
}

bool operator<(const Time& a, const Time& b)
{
  return 60 * a.hr + a.mn < 60 * b.hr + b.mn;
}
```

Linear Search Big Oh

Big oh is expressed in terms of whether a search was successful or not. For an unsuccessful search of an unordered array or linked list, there are n comparisons to make, so that's O(n). For an ordered array, we can expect that on average we'll traverse halfway, for n/2 compares -- still O(n), although with a lower constant multiplier.

A successful search finds the match at an average of midway through, whether it's ordered or not. That's n/2 compares, or O(n). No matter how we look at it or what we assume about the contents of the array or linked list, linear search is:

```
O(n)
```