```cpp
//
//  BinaryChop.cpp
//  Return index of element
//
//  Created by Stewart Bracken on 12/8/13.
//  Copyright (c) 2013 Stewart Bracken. All rights reserved.
//

#include <stdio.h>
#include <vector>
#include <thread>

#include "BinaryChop.h"
#include "FunctionalVector.h"

#include "MathUtil.h"

// My progression for better binary search functions.
// Chop6 should be the fastest.

//Iterative approach. Pretty fast, not as fast as chop2.
// O(nlogn) -- if lucky search hit, can return before nlogn.
int BinaryChop::chop1(int to_find, const std::vector<int>& data){
    int len = static_cast<int>(data.size()),
        low = 0;
    if(len == 0) return NOT_FOUND;
    int i, curr_data;

    while( len > 0 ){
        i = len/2 + low;
        curr_data = data[i];
        if( curr_data == to_find ){
            return i;
        }else if( to_find > curr_data ){
            //Need to search farther down array
            low = i+1;
            len = (len-1)/2;
        }else{ //to_find < curr_data
            //Search again at a smaller index.
            len = div_ceil( len-1, 2 );
        }
    }
    return NOT_FOUND;
}
```

```
//Helper tail recursive function for chop2
static int chop2_rec(int& to_find, const std::vector<int>& data,
                     int& length, int& low){
    if(length == 0)
        return NOT_FOUND;
    int i = length/2 + low;
    int curr_data = data[i];
    if( curr_data == to_find ){
        return i;
    }else if( to_find > curr_data ){
        low = i+1;
        length = (length-1)/2;
    }else{
        length = div_ceil(length-1, 2);
    }
    return chop2_rec(to_find, data, length, low);
}

// Tail recursive chop.
// O(nlogn) -- if lucky search hit, can return before nlogn.
int BinaryChop::chop2(int to_find, const std::vector<int>& data){
    int length = static_cast<int>(data.size());
    int low = 0;
    return chop2_rec(to_find, data, length, low);
}


//Functional style array slicing binary chop
// O(nlogn) -- if lucky search hit, can return before nlogn.
int BinaryChop::chop3(int to_find, const std::vector<int>& data){
    FunctionalVector<int> fun_data(data); //copy data
    size_t i;
    int curr_data;
    while( fun_data.size() > 0 ){
        i = fun_data.size()/2;
        curr_data = fun_data[i];
        if( curr_data == to_find ){
            return static_cast<int>(fun_data.index_at(i));
        }else if( to_find > curr_data ){
            //Need to search farther down array
            fun_data.slice(i+1, (fun_data.size()-1)/2);
        }else{ //to_find < curr_data
            //Search again at a smaller index.
            fun_data.slice(0, div_ceil( static_cast<int>(fun_data.size()-1), 2) );
        }
```

```
    }
    return NOT_FOUND;
}

//thread function for doing work on data.
void chop4_thread(int to_find, const std::vector<int> &data, int low, int length,
int* result){
    int i = length/2 + low;
    int curr_data = data[i]; //thread-safe read

    if(curr_data == to_find){
        *result = i;
        return;
    }else if( to_find > curr_data ){
        low = i+1;
        length = (length-1)/2;
    }else{
        length = div_ceil(length-1, 2);
    }

    if(length == 0)
        return;

    std::thread continue_search(chop4_thread, to_find, data, low, length, result);
    continue_search.join();
}

void chop4_thread_spawn(int to_find, const std::vector<int> &data, int low, int
length, int* result){
    if(length == 0) return;
    if(to_find < data[low] || to_find > data[low+length-1]) return;
    std::thread continue_search(chop4_thread, to_find, data, low, length, result);
    continue_search.join();
}

//very slow. binary search wan't really meant to be multithreaded.
// O(nlogn) -- if lucky search hit, can return before nlogn.
int BinaryChop::chop4( int to_find, const std::vector<int> &data ){
    int result = NOT_FOUND; //The thread which finds it sets result to idx
    int half = static_cast<int>(data.size())/2;
    std::thread left(chop4_thread_spawn, to_find, data, 0, half, &result);
    std::thread right(chop4_thread_spawn, to_find, data, half,
div_ceil(static_cast<int>(data.size()), 2), &result);
    left.join();
    right.join();
```

```
    return result;
}

//Iterative approach 2. Attempting to be faster than tail recursion.
// This ends up being just as fast as tail recursion.
// Wow, it turns out I overengineered all my other solutions.
// This one is much more simple and elegant. Nice dice.
// O(nlogn) -- bound above and below nlogn due to deferred equality.
int BinaryChop::chop5( int to_find, const std::vector<int>& data ){
    int imax = static_cast<int>( data.size()-1 ),
        imin = 0,
        imid;
    if( imax < 0 ) return NOT_FOUND;

    while( imin < imax ){
        imid = (imin + imax)/2;
        if( to_find > data[imid] ){
            imin = imid+1;
        }else{ //to_find < curr_data
            imax = imid;
        }
    }
    if( imin == imax && data[imin] == to_find ){
        return imin;
    }
    return NOT_FOUND;
}

//Helper tail recursive function for chop6
static int chop6_rec( int& to_find, const std::vector<int>& data,
                      int& imin, int& imax ){
    if( imin == imax ){
        if( data[imin] == to_find ){
            return imin;
        } else {
            return NOT_FOUND;
        }
    }

    int imid = ( imin + imax )/2;

    if( to_find > data[imid] ){
        imin = imid+1;
    }else{
        imax = imid;
```

```
    }

    return chop6_rec( to_find, data, imin, imax );
}

// θ(nlogn) -- bound above and below nlogn due to deferred equality.
int BinaryChop::chop6( int to_find, const std::vector<int>& data ){
    int imax = static_cast<int>( data.size()-1 );
    int imin = 0;
    if( imax < 0 )
        return NOT_FOUND;
    return chop6_rec( to_find, data, imin, imax );
}
```