

Searching

CIS*2520 Summer 2006

Searching

- Information retrieval is central to many computer applications, e.g.
 - given a name, provide a telephone number
 - given an account number, provide list of recent transactions
 - etc.
- In general:
 - given some piece of data (or many)
 - find a record containing other information associated with this data

*CIS*2520 - Summer 2006*

D. McCaughan

Searching

- Note:
 - in general we will assume that for any given key there is at most one record associated with it (this is not always the case, but is a good place to start)
- Searching for keys to locate records can be the most time-consuming operation for a program
 - efficiency is an issue
 - we attempt to optimize this by
 - appropriate organization of records
 - good search techniques

*CIS*2520 - Summer 2006*

D. McCaughan

Terminology

- External Searching
 - records are stored on disk or tape
 - target of search lies on external storage
- Internal Searching
 - records to be searched are entirely stored in memory
- There are specific techniques that can be used to exploit the organization of these two types
 - we will only concern ourselves with internal sorting in this course

*CIS*2520 - Summer 2006*

D. McCaughan

Terminology (cont.)

- **KEY value**
 - the data being used as the basis for record association
 - each record has a key value
- **TARGET value**
 - the KEY for which we are searching
 - there is only one TARGET of a given search
 - direct searching of an array is a bit of an exception to this as there is no real control over what has been put in an array, so duplicate keys can easily be present

CIS*2520 - Summer 2006

D. McCaughan

Conventions

- A conceptual picture of data for which searching is desired:
 - there exists some “record” type which contains a **key** and the **data** associated with this key
 - we wish to consider these concepts completely abstractly
 - avoid specifics of data structures or implementation
 - note that it is common to search pre-existing data structures, in which case the “key” field might well be some existing element of the data structure

```
typedef struct Record_t
{
    KeyType *key; /* some type for the key */
    DataType *data; /* some type for the data */
    ... /* e.g. could include next/prev if a linked list */
}

KeyType *getKey(Record *);
DataType *getData(Record *);
```

CIS*2520 - Summer 2006

D. McCaughan

Conventions (cont.)

- e.g. a list

```
typedef struct List_t
{
    int item_count;
    Record items[]; /* a generic record - could be anything */
    ...
} List;

typedef struct List_t
{
    int item_count;
    Record *head;
    Record *current;
    ...
} List;
```

CIS*2520 - Summer 2006

D. McCaughan

Conventions (cont.)

- Any search function we write conceptually has 3 parameters:
 - the TARGET key
 - the list to be searched (whatever its representation)
 - in an OO language this will typically be just the instance invoking the method
 - a result parameter
 - the data associated with the target if it is located, which we will generically refer to as being of type DataType
 - NOTE: if we are searching a simple array, an alternative is to return the index where the target was found

CIS*2520 - Summer 2006

D. McCaughan

Conventions (cont.)

- Implementation of these concepts is language dependent
 - C/Pascal/C++
 - can have function receive a pointer/reference to the data to be filled in, and return 0/1 or true/false to indicate success or failure
 - use return value (what about failure? null?)
 - A language such as Java does not support pass by reference, or the notion of passing a reference to a reference (result parameters are not possible)
 - solutions?
 - declare a class to hold the result and pass an instance of that
 - use return value (what about failure? null? throw exception?)

CIS*2520 - Summer 2006

D. McCaughan

Conventions (cont.)

- We need a way to compare key values during our search
 - this is obvious for built-in types (<, <=, !=, >=, >)
 - we may need user-defined comparisons for user-defined data types
 - (e.g. lexicographic search over non-string data)
 - we will simply assume we have defined appropriate `isEqual`, `lessThan` and `greaterThan` functions for the key values in the list
 - this permits the description of our search algorithms without regard to actual implementation in so far as that is possible
 - note that for ADTs this becomes generally true
 - where (module-wise) would such a function be implemented?
 - stand-alone? macros? operation on Record?

CIS*2520 - Summer 2006

D. McCaughan

Conventions (cont.)

- For example, if KeyType is an integer
 - note that where the key is a more complicated type, it might be passed as a pointer in-keeping with how we have treated abstract types to this point

```
int isEqual(int a, int b)
{
    return(a == b);
}

int lessThan(int a, int b)
{
    return(a < b);
}

int greaterThan(int a, int b)
{
    return(a > b);
}
```

CIS*2520 - Summer 2006

D. McCaughan

Sequential Search

- Linear Search
- Simplest kind of search
 - begin at one end of the list and scan it until the key is found or the end is reached
- Note: this lends itself easily to both contiguous and linked implementations
- Assumption: this list is sorted
 - is this necessary?

CIS*2520 - Summer 2006

D. McCaughan

Sequential Search: Contiguous

```
DataType *sequentialSearch(List *l, KeyType *target)
{
    int location = 0;

    while((lessThan(getKey(l->items[location]), target)
        && (location < l->item_count))
        location++;

    if (isEqual(target, getKey(l->items[location])))
        return(getData(l->items[location]));
    else
        return(null);
}

/* how to modify this to handle an unsorted list? */
/* why not use a for loop? */
```

CIS*2520 - Summer 2006

D. McCaughan

Sequential Search: Linked

```
DataType *sequentialSearch(List *l, KeyType *target)
{
    Record *location;

    location = l->head;
    while ((location != null)
        && (lessThan(getKey(location), target))
        location = getNext(location);

    if ((location == null)
        || (!isEqual(getKey(location), target))
        return(null);
    else
        return(getData(location));
}

/* how to modify this to handle an unsorted list? */
```

CIS*2520 - Summer 2006

D. McCaughan

Binary Search

- Sequential search, in the worst case, must examine all elements in the list to determine success or failure
 - running time directly proportional to the number of elements in the list
- Consider finding “Thomas Z. Smith” in the phone book by sequential search
 - is there a better way?
- If the list is sorted, we can use a binary search algorithm to locate keys very efficiently, even for large lists

CIS*2520 - Summer 2006

D. McCaughan

Binary Search (cont.)

- General idea:
 - compare the target to the key in the middle of the list, and then restrict our search to either the first half or second half depending on whether our target is smaller or larger than the middle key
 - continue this process with the sub-list, now only half the size of the original
- Size of the list to be searched is reduced by a factor of 2 at each step
- Restrictions:
 - key values must be of some ordinal type
 - list must be in order

CIS*2520 - Summer 2006

D. McCaughan

Binary Search (cont.)

- Note: well suited to contiguous implementations, but poor for linked representation (no random access)
- Caution:
 - a simple idea, but difficult to implement correctly
 - 90% of professional programmers fail to code binary search correctly in a short period of time
 - Jon Bentley, "Programming Pearls: Writing Correct Programs", *Communications of the ACM* 26 (1983), pp. 1040--1045

CIS*2520 - Summer 2006

D. McCaughan

Implementing Binary Search

- Use 2 indices, *first* and *last*, to enclose the part of the list in which we are looking for the target key
 - at each iteration we reduce this interval by half
- How do we know this works?
 - loop invariant: the target key, if present, will be found between the indices *first* and *last*, inclusive
 - established initially by setting *first* = 1 and *last* = item_count
 - the loop must stop when *last* <= *first*
 - i.e. when remaining part of list contains at most 1 item
 - loop will terminate
 - sub-list size (*last* - *first* + 1) must strictly decrease at each iteration

CIS*2520 - Summer 2006

D. McCaughan

Binary Search Implementation

```
DataType *binarySearch(List *l, KeyType *target)
{
    int first = 0, last = l->item_count - 1, middle;

    while (last > first)
    {
        middle = (last + first) / 2; /* note: integer division */

        if (lessThan(getKey(l->items[middle]), target)
            first = middle+1; /* reduce to top half of list */
        else
            last = middle; /* reduce to bottom half of list */
    }

    if (last < 0)
        return(null); /* search of empty list always fails */
    else if (isEqual(getKey(l->items[last]), target)
        return(getData(l->items[last]));
    else
        return(null);
}
```

CIS*2520 - Summer 2006

D. McCaughan

Notes Regarding Binary Search

- Verify for yourself that our preconditions hold
- 2 Points
 - *repeatedly reducing the size of a problem by half at each iteration is frighteningly efficient*
 - a list of 1 000 000 entries requires only 20 comparisons to either find, or fail to find, any target key
 - *we do not detect equality in the provided method*
 - it is possible that extra work is being done after we find the target (as the list shrinks around it)
 - however, adding a test for equality at each iteration increases the amount of work we do at each iteration --- it is not clear that testing for equality saves us anything
 - it turns out that on average, testing for equality will do slightly more work than just letting it run to the end

CIS*2520 - Summer 2006

D. McCaughan