Programming & Data Structure

CS 11002

Partha Bhowmick

http://cse.iitkgp.ac.in/~pb

CSE Department
IIT Kharagpur

Spring 2012-2013

Searching

Largest Element (1)

Write a recursive C function that takes an `int` array and the number of elements present in the array as parameters, and returns the largest element of the array.

Iterative Function

```
int largestElement(int a[], int n) {
    int i, largest = a[0];
    for(i=1; i<n; ++i)
        if(a[i] > largest) largest = a[i];
    return largest;}
```

Largest Element

Inductive Def ⇒ Recursive Function

$$l(a[0 \cdots n-1]) = \begin{cases} a[0] & \text{if } n = 1, \\ max(a[0],\ l(a[1 \cdots n-1])) & \\ & \text{if } n > 1. \end{cases}$$

```
int largestElement(int a[], int n) {
    int temp;
    if(n == 1) return a[0];
    temp = largestElement(a+1, n-1);
    return MAX(a[0], temp);
} // largestFR.c
```

Reversal (1)

Consider the problem of reversing a string or the digits of a number.
Ex: reverse(274) = 472; reverse(palin) = nilap.

Iterative Function

```
int reverse(int n){
    int rev = 0;
    while(n){
        rev=10*rev+n%10; n /= 10;}
    return rev;
} // reverseIntF.c
```

Sequential / Linear Search (1)

Write a function that takes n data stored in an array of type `int` and an integer as the *key*. The function sequentially searches the array for the *key*. If the *key* is present among the data in the array, then it returns the corresponding array *index*; otherwise it returns -1.

Example

`a` =

8	3	11	5	7

key = 5: The function should return 3.

key = 6: The function should return -1.

Sequential / Linear Search (2)

Iterative C Function

```
#define NOTFOUND -1
int seqSearch(int a[], int n, int key){
    int i;
    for(i=0; i<n; ++i)
        if(a[i] == key) break;
    if(i == n) return NOTFOUND;
    return i;
} // seqSearchF.c
```

Sequential / Linear Search (3)

Runtime

Best case: `key` = `a[0]` and the `for loop` is executed only once.

Worst case: When `key` = `a[n - 1]` or there is no match. The number of times the loop is executed is `n`, the number of data.

Sequential / Linear Search (4)

Recursive C Function

```
#define NOTFOUND -1
int seqSearch(int a[],int n,int key){
    int temp;
    if(a[0] == key) return 0;
    if(n == 1) return NOTFOUND;
    temp = seqSearch(a+1, n-1, key);
    if(temp == NOTFOUND) return NOTFOUND;
    return temp + 1;
} // seqSearchFR.c
```

Sequential / Linear Search (5)

Runtime

Best case: `key` = `a[0]` and there is only one call.

Worst case: When `key` = `a[n − 1]` or there is no match. The number of times the function is called is `n`, the number of data.

Space Usage: The number of stack frames (activation records) used by the recursive function is proportional to `n` in the worst case. On the contrary, the iterative function uses constant amount of extra space.

Binary Search

Search for a *key* can be made more efficient if the elements stored in the array are *sorted* in *ascending order* (or descending order).

Binary Search (2)

$key = 21$

0	1	2	3	4	5	6	7	8	9
2	6	7	9	10	13	21	23	24	28
2	6	7	9	10	13	21	23	24	28
2	6	7	9	10	13	21	23	24	28
2	6	7	9	10	13	21	23	24	28

lo	hi	mid
0	9	4
4	9	6
4	6	5
6	=	6

⇑

end of `while` `loop`

Binary Search

Iterative C Function

```
int binarySearch(int a[], int lo,
                 int hi, int key) {
  while(lo != hi) {
    int mid = (lo+hi)/2;
    if(key <= a[mid]) hi = mid;
    else lo = mid+1; }
  if(key == a[low]) return low;
  return -1;
} // binarySearchF.c
```

Binary Search

Runtime

The `while loop` is executed at most $\log_2 n + 1 = O(\log n)$ times.
For a large value of n, this gives a definite advantage over *sequential search* that executes the loop n times on a *'bad' data set*.
But in case of binary search, sorted data is required.

Binary Search

Inductive Definition

$$\mathrm{bS}(a,l,h,k) = \begin{cases} l & \text{if } l = h \;\&\; a[l] = k, \\ -1 & \text{if } l = h \;\&\; a[l] \neq k, \\ bS(a,l,m,k) & \text{if } l < h \;\&\; k \leqslant a[m] \\ bS(a,m+1,h,k) & \text{if } l < h \;\&\; k > a[m] \end{cases}$$

where l is the *low index*, h is the *high index*, k is the *key* to search, and $m = \frac{l+h}{2}$.

Binary Search

Recursive C Function

```
int binarySearch(int a[], int l, int h, int key){
  if(l == h) {
    if(a[l] == key) return l;
    else return -1;}
  if(l < h) {
    int m = (l+h)/2;
    if(a[m] >= key)
      return binarySearch(data, l, m, key);
    else return binarySearch(data, m+1, h, key);}
} // binarySearchR.c
```

Binary Search

Runtime

For n elements, let T_n be the runtime of recursive binary search. Then

$$T_n = \begin{cases} c_0 & \text{if } n = 1 \\ T_{n/2} + c_1 & \text{if } n > 1, \end{cases}$$

where c_0 and c_1 are two constants.
Using the *iterative method*, we have
$T_{n/2} = T_{n/2^2} + c_1$, $T_{n/2^2} = T_{n/2^3} + c_1$, ..., $T_1 = c_0$.
Hence, $T_n = c_0 + c_1 \log_2 n = O(\log_2 n)$.