ECE244

Wael Aboulsaadat

Hash table Implementation

Acknowledgment: these slides are partially based on slides by; Prof. Schmidt from Drexel U., Prof. Shewchuk from UC Berkely, Kruse & Ryba Data Structure and Program Design in C++, Prof. Savitch Problem Solving in C++ and others

Hash table -- insert

```
bool HashTable::insert( const Element &elm)
{
    if( _nCount == _nSize)
        return (false); // table is full!

    int home = hashfunc(elm.key);
    int nIndex;

    for(nIndex=home; !is_empty(nIndex); nIndex = (nIndex + 1) % _nSize )
        if( elm.key == Hashtable[nIndex].key )
                return (false); // duplicate

    HashTable[nIndex] = elm;

    _nCount++;

    return (true);

}
```

Hash table -- search

```
bool HashTable::search( const int Key, Element& elm )
{
    int nHome = hashfunc(key);

    for(int nIndex = nHome; !is_empty(nIndex%_nSize) &&
                                nIndex < (nHome + _nSize); nIndex++ )
          if(Key == HashTable[nIndex % _nSize].key )
          {
               elm = HashTable[nIndex % _nSize];
               return (true);
          }

    return (false);
}
```

Analysis of Algorithms

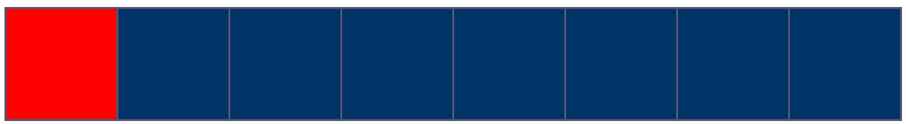
Big-Oh
Part 3

Recurrence Relations

- Can easily describe the runtime of recursive algorithms
- Can then be expressed in a closed form (not defined in terms of itself)
- Consider the linear search:

Eg. 1 - Linear Search

- **Recursively**
- **Look at an element (constant work, c), then search the remaining elements…**



- $T(n) = T(n-1) + c$
- "The cost of searching n elements is the cost of looking at 1 element, plus the cost of searching n-1 elements"

Linear Seach (cont)

Caveat:

- You need to convince yourself (and others) that the single step, examining an element, *is* done in constant time.
- Can I get to the i^{th} element in constant time, either directly, or from the $(i-1)^{th}$ element?
- Look at the code

Methods of Solving Recurrence Relations

- Substitution (aka iteration)
- Draw the recursion tree, think about it
- Guess at an upper bound, prove it by induction

Linear Search (cont.)

- **We'll "unwind" a few of these**

$T(n) = T(n-1) + c$ **(1)**

But, $T(n-1) = T(n-2) + c$, from above

Substituting back in:

$T(n) = T(n-2) + c + c$

Gathering like terms

$T(n) = T(n-2) + 2c$ **(2)**

Linear Search (cont.)

- **Keep going:**

  $T(n) = T(n-2) + 2c$

  $T(n-2) = T(n-3) + c$

  $T(n) = T(n-3) + c + 2c$

  $T(n) = T(n-3) + 3c \qquad (3)$

- **One more:**

  $T(n) = T(n-4) + 4c \qquad (4)$

Looking for Patterns

- **Note, the intermediate results are enumerated**
- **We need to pull out patterns, to write a general expression for the k^{th} unwinding**
  - This requires practice. It is a little bit art. The brain learns patterns, over time. Practice.
- **Be careful while simplifying after substitution**

Eg. 1 – list of intermediates

Result at i^{th} unwinding	i
$T(n) = T(n-1) + 1c$	1
$T(n) = T(n-2) + 2c$	2
$T(n) = T(n-3) + 3c$	3
$T(n) = T(n-4) + 4c$	4

Linear Search (cont.)

- **An expression for the kth unwinding:**

  $T(n) = T(n-k) + kc$

- **We have 2 variables, k and n, but we have a relation**
- **T(d) is constant (can be determined) for some constant d (we know the algorithm)**
- **Choose any convenient # to stop.**

Linear Search (cont.)

- **Let's decide to stop at T(0). When the list to search is empty, you're done…**
- **0 is convenient, in this example…**

Let n-k = 0 => n=k

- **Now, substitute n in everywhere for k:**

$T(n) = T(n-n) + nc$

$T(n) = T(0) + nc$

$= c_0 + cn = O(n)$

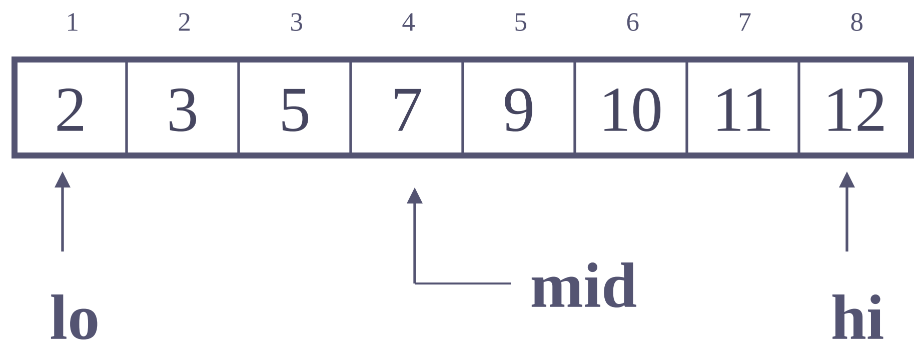
(T(0) is some constant, c_0)

Binary Search

- for an ordered array A, finds if x is in the array A[lo…hi]

BINARY-SEARCH (A, lo, hi, x)

```
if (lo > hi)
    return FALSE
mid ← ⌊(lo+hi)/2⌋
if x = A[mid]
    return TRUE
if ( x < A[mid] )
    BINARY-SEARCH (A, lo, mid-1, x)
if ( x > A[mid] )
    BINARY-SEARCH (A, mid+1, hi, x)
```



Example

- **A[8] = {1, 2, 3, 4, 5, 7, 9, 11}**
  - lo = 1 hi = 8 x = 7

1	2	3	4	5	6	7	8
1	2	3	4	5	7	9	11

mid = 4, lo = 5, hi = 8

				5	6	7	8
1	2	3	4	5	7	9	11

mid = 6, A[mid] = x Found!

Another Example

- **A[8] = {1, 2, 3, 4, 5, 7, 9, 11}**
  - lo = 1 hi = 8 x = 6

1	2	3	4	5	6	7	8
1	2	3	4	5	7	9	11

low high

mid = 4, lo = 5, hi = 8

1	2	3	4	5	7	9	11

low high

mid = 6, A[6] = 7, lo = 5, hi = 5

1	2	3	4	5	7	9	11

mid = 5, A[5] = 5, lo = 6, hi = 5

1	2	3	4	5	7	9	11

high low

NOT FOUND!

Analysis of BINARY-SEARCH

```
BINARY-SEARCH (A, lo, hi, x)
    if (lo > hi)                               ← constant time: c1
        return FALSE
    mid ← ⌊(lo+hi)/2⌋                          ← constant time: c2
    if x = A[mid]                              ← constant time: c3
        return TRUE
    if ( x < A[mid] )
        BINARY-SEARCH (A, lo, mid-1, x)        ← same problem of size n/2
    if ( x > A[mid] )
        BINARY-SEARCH (A, mid+1, hi, x)        ← same problem of size n/2
```

- $T(n) = T(n/2) + c$
  - $T(n)$ – running time for an array of size n

Binary Search (cont)

Let's do some quick substitutions:

$T(n) = T(n/2) + c$ (1)

but $T(n/2) = T(n/4) + c$, so

$T(n) = T(n/4) + c + c$

$T(n) = T(n/4) + 2c$ (2)

$T(n/4) = T(n/8) + c$

$T(n) = T(n/8) + c + 2c$

$T(n) = T(n/8) + 3c$ (3)

Binary Search (cont.)

Result at i^{th} unwinding	i
$T(n) = T(n/2) + c$	1
$T(n) = T(n/4) + 2c$	2
$T(n) = T(n/8) + 3c$	3
$T(n) = T(n/16) + 4c$	4

Binary Search (cont)

- **We need to write an expression for the k^{th} unwinding (in n & k)**
  - Must find patterns, changes, as i=1, 2, …, k
  - This can be the hard part
  - Do not get discouraged! Try something else…
  - We'll re-write those equations…
- **We will then need to relate n and k**

Binary Search (cont)

Result at i^{th} unwinding			i
$T(n)$	$= T(n/2) + c$	$= T(n/2^1) + 1c$	1
$T(n)$	$= T(n/4) + 2c$	$= T(n/2^2) + 2c$	2
$T(n)$	$= T(n/8) + 3c$	$= T(n/2^3) + 3c$	3
$T(n)$	$= T(n/16) + 4c$	$= T(n/2^4) + 4c$	4

Binary Search (cont)

- **After k unwindings:**

  $T(n) = T(n/2^k) + kc$

- **Need a convenient place to stop unwinding – need to relate k & n**
- **Let's pick $T(0) = c_0$ So,**

  $n/2^k = 0$ =>

  $n=0$

  Hmm. Easy, but not real useful…

Binary Search (cont)

- **Okay, let's consider $T(1) = c_0$**
- **So, let:**

$$n/2^k = 1 \quad \Rightarrow$$

$$n = 2^k \quad \Rightarrow$$

$$k = \log_2 n = \lg n$$

Binary Search (cont.)

- **Substituting back in (getting rid of k):**

$$T(n) = T(1) + c \lg(n)$$

$$= c_0 + c \lg(n)$$

$$= O(\lg(n))$$

Example Recurrences

- $T(n) = T(n-1) + n$ $\Theta(n^2)$
  - Recursive algorithm that loops through the input to eliminate one item
- $T(n) = T(n/2) + c$ $\Theta(\lg n)$
  - Recursive algorithm that halves the input in one step
- $T(n) = T(n/2) + n$ $\Theta(n)$
  - Recursive algorithm that halves the input but must examine every item in the input
- $T(n) = 2T(n/2) + 1$ $\Theta(n)$
  - Recursive algorithm that splits the input into 2 halves and does a constant amount of other work

Methods of Solving Recurrence Relations

→ Substitution

Draw the recursion tree, think about it

Guess at an upper bound, prove it by induction

Exam!

- **Focus on second half of semester**
- **Data Structures**
  - Linked Lists
  - BST
  - Hash table
- **Big Oh**
- **Labs**
- **Exam Samples**
- **Problems**