Searching

강지훈
jhkang@cnu.ac.kr
충남대학교 컴퓨터공학과

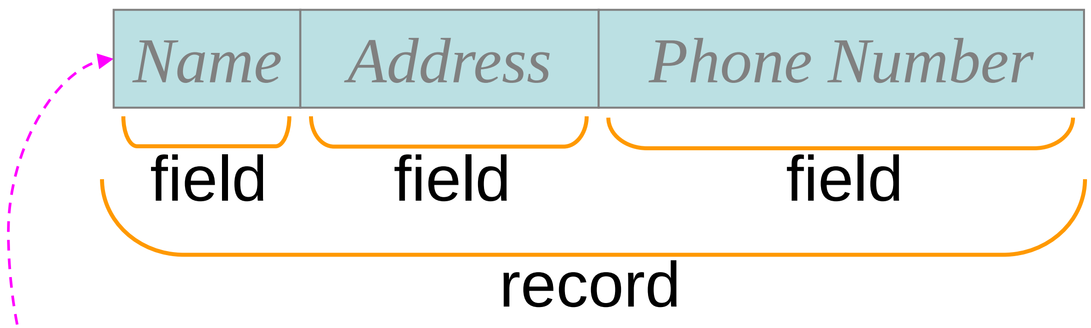
Search

Sequential Search
Binary Search
Interpolation Search

Terms

- **List** : A collection of records in Memory.
- **File** : A collection of records in External Storage.
- **Key** : The field used to distinguish among records.

  (Example) Telephone Directory File



Key: The search is performed on this '*Name*' field.

Search

- What is the SEARCH ?

  ⇒ Find the record with the given key value.

  ⇒ Find i such that

  (_elements[i].key == givenKey)

  for the given key value *givenKey*.

Sequential Search

Sequential Search Algorithm

```
public class Element {
    private int key;
    ....
}

static final int   MAX_SIZE = 1000 ; /* maximum size of list plus one */
private Element[] _elements = new Element[MAX_SIZE];

public int sequentialSearch (int givenKeyValue, int givenSize)
{
    //  Search an array "_elements[]" that has "givenSize" numbers.
    //  Return    i if (_elements[i].key = givenKey),
    //  Return -1 if (givenKey is not in the _elements[].key).
    int  i ;
    _elements[givenSize].setKey(givenKeyValue) ;
        /* a sentinel that signals the end of the list */
    for ( i=0 ; _elements[i].key() != givenKeyValue && i<givenSize ; i++) ;
    return ((i < givenSize) ? i : -1) ;
        // if (i < givenSize) then return i else return -1 ;
}
```

Analysis

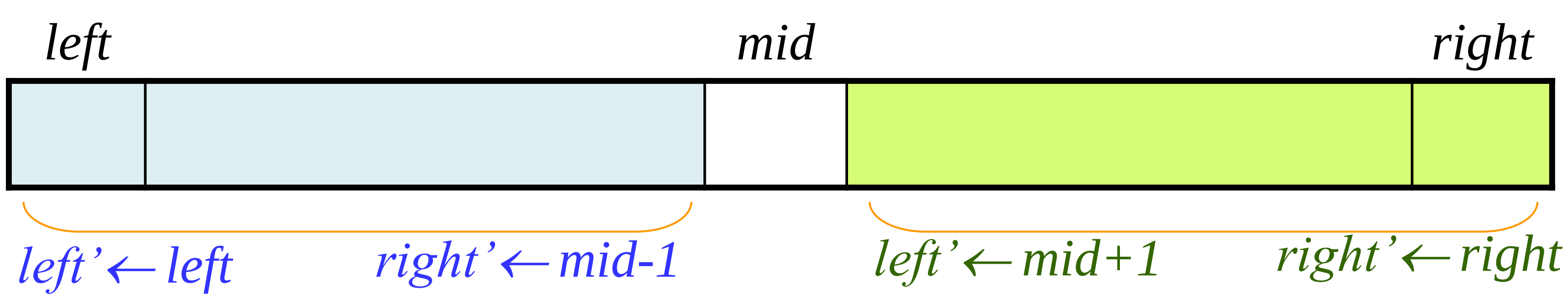
- What is the role of the sentinel ?
  - It simplifies the loop condition.
  - N: "givenSize"
  - Worst case
    - $(N+1)$ key comparisons ➔ $O(N)$
  - Average case
    - If the keys are distinct and $\_elements[i].key == givenKey$ then $(i+1)$ key comparisons are made.

$$\Rightarrow \sum_{i=0}^{N-1} \frac{i+1}{N} = \frac{1}{N}\sum_{i=1}^{N} i = \frac{1}{N} \times \frac{N(N+1)}{2} = \frac{N+1}{2} = O(N)$$

Binary Search

Basic idea

- It assumes that the searching values are already sorted in non-decreasing order.
  - $\_elements[0].key \le \_elements[1].key \le \cdots \le \_elements[n\text{-}1].key$
- If we compare the given value with the value in the middle position, we can consider only the half of the list for the next comparison.
  - Initially, $left \leftarrow 0$ and $right \leftarrow givenSize\text{-}1$.
  - $mid \leftarrow \lfloor (left + right) / 2 \rfloor$.



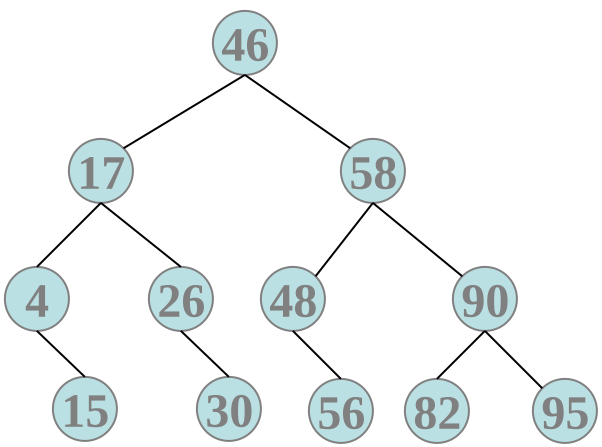
Example: Binary Search (*givenSize:12*)

- _elements[].key :{4, 15, 17, 26, 30, 46, 48, 56, 58, 82, 90, 95}

- To find 56,
  left = 0, right = 11, mid = $\lfloor (0+11)/2 \rfloor$ = 5, _elements[5].key = 46;
  left = 6, right = 11, mid = $\lfloor (6+11)/2 \rfloor$ = 8, _elements[8].key = 58;
  left = 6, right = 7, mid = $\lfloor (6+7)/2 \rfloor$ = 6, _elements[6].key = 48;
  left = 7, right = 7, mid = $\lfloor (7+7)/2 \rfloor$ = 7, _elements[7].key = 56: FOUND;

- To find 35,
  left = 0, right = 11, mid = $\lfloor (0+11)/2 \rfloor$ = 5, _elements[5].key = 46;
  left = 0, right = 4, mid = $\lfloor (0+4)/2 \rfloor$ = 2, _elements[2].key = 17;
  left = 3, right = 4, mid = $\lfloor (3+4)/2 \rfloor$ = 3, _elements[3].key = 26;
  left = 4, right = 4, mid = $\lfloor (4+4)/2 \rfloor$ = 4, _elements[4].key = 30;
  left = 5, right = 4: NOT FOUND;

- The tree is called a Decision Tree.



Implementation of Binary search algorithm

```
private int  binarySearch (int givenKey, int givenSize)
{
    /* search _elements[0],…, _elements[n-1] */
    int left = 0, right = givenSize-1, middle;
    while (left <= right) {
        middle = (left + right) / 2;
        switch (compare(_elements[middle].key(), givenKey)) {
                case -1 : left = middle + 1;
                        break;
                case 0  : return middle; /* Found */
                case 1  : right = middle - 1;
        }
    }
    return -1;
}
```

Recursive Approach for Binary Search

```
private int binarySearchRecursively (int givenKey, int left, int right) {
    if ( left <= right ) {
        int  mid = (left + right) / 2 ;
        if ( _elements[mid].key() == givenKey )
            return mid ;
        else if ( _elements[mid].key() > givenKey )
            return binarySearchRecursively (givenKey, left, mid-1);
        else if ( _elements[mid].key() < givenKey )
            return binarySearchRecursively (givenKey, mid+1, right);
    }
    return -1;
}
```

❑Comparison function

- An implementation example:

```
private int  compare ( int x, int y )
{
    /* compare x and y:
     return -1 for less than, 0 for equal, 1 for greater */
    if ( x < y )
        return -1;
    else if ( x == y )
        return  0;
    else
        return  1;
}
```

- Any type value can be implemented.

Time Complexity of Binary Search

- Worst case comparison

  Let c be the number of comparisons in the worst case.

  Then, $\lceil n / 2^c \rceil = 1$

  Roughly, $n / 2^c = 1$

  $2^c = n$

  So, $c = \log n = O(\log n)$

Interpolation Search

Basic Idea

- If we are looking for a name beginning with *w* in the telephone directory, we start the search towards the end of the directory rather than the middle.

- Use the value of *givenKey* for deciding the middle position.
  - *mid = (givenKey - _elements[left].key) / (_elements[right].key - _elements[left].key) * (right - left) + left*
  - *Initially:*
    left = 0, and right = givenSize - 1.
    *So, mid =(givenKey - _elements[0].key) / (_elements[givenSize-1].key - _elements[0].key) *(givenSize - 1)*

List Verification

List verification

- We compare lists to verify whether they are identical or to identify the differences.
  - _elements1[ ]: n records (_elememts1[0] to _elememts1[n-1])
  - _elements2[ ]: m records (_elememts2[0] to _elememts2[m-1])

- We consider the 2 cases:
  - _elememts1 and _elememts2 are UNORDERED.
  - _elememts1 and _elememts2 are ORDERED.

List verification for Unordered Lists

- Basic idea:

```
for (each record in _elememts1[])  /* n times */  {
        search _elememts2[] sequentially; /* O(m) */
}
```

- Time complexity: Totally, O(nm).

List verification for unordered lists [1]

```
private  verify1 (Element[] givenElements, int n, int m)
/* compare two unordered lists this._elements and givenElements */
{
    int  i, comparisonResult ;
    boolean[] marked = new boolean[MAX_SIZE] ;
    for ( j=0 ; j < m ; j++ ) {
        marked[j] = false ;
    }
    for ( i=0 ; i < n ; i++ ) {
        comparisionResult = SequentialSearch(givenElements, m, _elements[i].key()) ;
        if ( (comparisonResult < 0 ) {
              System.out.println( this. _elements[i].key() + " is not in givenElements.");
        }
        else {
            /* check each of the other fields from _elements[i] and givenElements[j],
             * and print out any discrepancies */
             marked[j] = true;
        }
    }
    for ( j=0 ; j < m ; j ++ ) {
        if ( !marked[j] )
             System.out.println(givenElements[j].key() + " is not in elements.");
    }
}
```

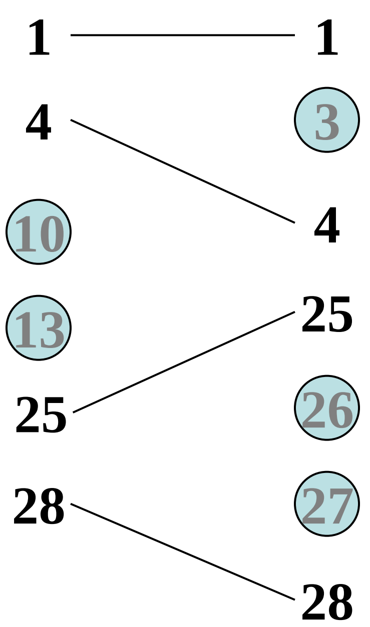
List verification for unordered lists [2]

- If we want to know only the elements with the same key:

```
private List verify1 (Element[] givenElements, int n, int m)
/* compare two unordered lists this._elements and givenElements */
{
    int  i ;
    int  comparisonResult ;
    List commonElementList = new List() ;

    for ( i=0 ; i < n ; i++ ) {
        comparisionResult =
            SequentialSearch(givenElements, m, _elements[i].key()) ;
        if ( (comparisonResult == 0 ) { // the same keys are found
             commonElementList .add(_elements[i]) ;
        }
    }
    return commonElementList ;
}
```

List verification for Ordered Lists



For sorting list1: $O(n \log n)$

list2: $O(m \log m)$

Comparisons: $O(n + m)$

Totally: $O(n \log n + m \log m + n + m)$

$= O(\max[n \log n\ ,\ m \log m])$

Algorithm for ordered lists [1]

```
private void  verify2 (Element[] givenElements, int n, int m)
/* Same task as verify1, but this._elements and givenElements are ordered
*/
{
    int     i, j ;
    sort (this._elements, n) ;
    sort (givenElements, m) ;
    i = j = 0 ;
```

Algorithm for ordered lists [2]

```
while (i < n && j < m) {
    if (_elements[i].key() < givenElements[j].key()) {
        System.out.println (_elements[i].key() + " is not in givenElements ") ;
        i++; /* ① */
    }
    else if (_elements[i].key() == givenElements [j].key()) {
        /* compare _elements[i] and givenElements [j]
         * on each of the other fields and
         * report any discrepancies */
        i++; j++; /* ② */
    }
    else {
        System.out.println(givenElements[j].key() + " is not in elements ") ;
        j++; /* ③ */
    }
} /* end of while */
for ( ; i < n; i ++)
    System.out.println (_elements[i].key() + " is not in givenElements.");
for ( ; j < m; j++ )
    System.out.println (givenElements[j].key() + " is not in elements.");
}
```

The worst case regarding the number of comparisons is when the loop variables are increased separately (① and ③), not both at the same time (②).

End of Searching