Data Structure

Lecture 8

Dr. Ahmed Fathalla

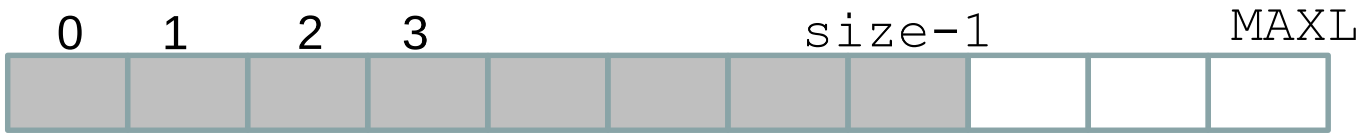
```
/*Pre: list is created.
Post:return the location if element exists, otherwise return -1.
*/

int SequentialSearch(KeyType target, List *pl){
   int current, s=ListSize(pl);
   KeyType currententry;
   for(current=0; current<s; current++){
      RetrieveItem(current, &currententry, pl);
      if( target == currententry)
         return current;
   }
   return -1;
}
```

See, how we did not call ListSize here so that we save the call time every iteration.

if `current position` trick is not used in linked implementation this statement alone is *O*(n), which is very inefficient, because every time `RetrieveList` is called the elements are traversed from the first one.

Re-write the code above in the implementation level for linked and contiguous implementations to save the time of the function calls. (compare to the book).

What about special cases in the above code?



Searching for 653

[061	087	154	170	275	426	503	509	512	612	653	677	703	765	897	908]
061	087	154	170	275	426	503	509	[512	612	653	677	703	765	897	908]
061	087	154	170	275	426	503	509	[512	612	653]	677	703	765	897	908
061	087	154	170	275	426	503	509	512	612	[653]	677	703	765	897	908

Searching for 400

[061	087	154	170	275	426	503	509	512	612	653	677	703	765	897	908]
[06	087	154	170	275	426	503]	509	512	612	653	677	703	765	897	908
061	087	154	170	[275	426	503]	509	512	612	653	677	703	765	897	908
061	087	154	170	[275]	426	503	509	512	612	653	677	703	765	897	908
061	087	154	170	275]	[426	503	509	512	612	653	677	703	765	897	908

Binary Search

Can we speed up the search time? We will assume that we will search in an ordered list.

Definition: An **ordered list** is a list in which each entry contains a key, such that the keys are in order. That is, if entry i comes before entry j in the list, then the key of entry i is less than or equal to the key of entry j.

This requires replacing the `InsertList` with `InsertOrder`. See the connection between enhancing the algorithms and designing the data structures.

Let us first write `InsertOrder`

```
/*Pre: created, not full, and ordered.
  post:  e inserted in order. if the new element has a key equal to an
         element in the list it will be inserted before it*/

void InsertOrder(ListEntry e, List *pl){
   int current, s=ListSize(pl);
   KeyType currententry;
   for(current=0; current<s; current++)
    {
      RetrieveItem(current, &currententry, pl);
      if ( e <= currententry)
            break;
    }
   InsertList(current, e, pl);
}
```

E.g., 8 will be inserted in position 3

if `current position` trick is not used in linked implementation this statement alone is *O*(n), which is very inefficient, because every time `RetrieveList` is called the elements are traversed from the first one.

Re-write the code above in the implementation level for contiguous implementations to save the time of the function calls.

What about special cases in the above code?

0	1	2	3	4			size-1			MAXLIST-1
3	5	7	9	15						

Binary Search

Searching for 653

[061	087	154	170	275	426	503	509	512	612	653	677	703	765	897	908]
061	087	154	170	275	426	503	509	[512	612	653	677	703	765	897	908]
061	087	154	170	275	426	503	509	[512	612	653]	677	703	765	897	908
061	087	154	170	275	426	503	509	512	612	[653]	677	703	765	897	908

Searching for 400

[061	087	154	170	275	426	503	509	512	612	653	677	703	765	897	908]
[06	087	154	170	275	426	503]	509	512	612	653	677	703	765	897	908
061	087	154	170	[275	426	503]	509	512	612	653	677	703	765	897	908
061	087	154	170	[275]	426	503	509	512	612	653	677	703	765	897	908
061	087	154	170	275]	[426	503	509	512	612	653	677	703	765	897	908

It looks recursive; let us try it. The *RecBinarySearch* **interface** has to be:

```
    int RecBinarySearch(KeyType, List *)
```

However, it seems from the table that we need to specify the start and the end indices. Therefore, we have to write another recursive function and call it in the above.

```
/*pre: list is ordered     Post: location returned, O.W. -1*/
int RecBinary(List *pl, KeyType k, int bottom, int top){
   int middle;
   KeyType mid_item;
   if (bottom<=top){
       middle=(bottom+top)/2;
       RetrieveItem(middle, &mid_item, pl)
       if ( k == mid_item )
          return middle;
       else if ( k < mid_item )
          return RecBinary(pl, k, bottom, middle-1);
       else
          return RecBinary(pl, k, middle+1, top);
   }
   return -1;
}
int RecBinarySearch(KeyType k, List *pl){
       return RecBinary(pl, k, 0, pl->size-1);
}
```

[061	087	154	170	275	426	503	509	512	612	653	677	703	765	897	908]
061	087	154	170	275	426	503	509	[512	612	653	677	703	765	897	908]
061	087	154	170	275	426	503	509	[512	612	653]	677	703	765	897	908

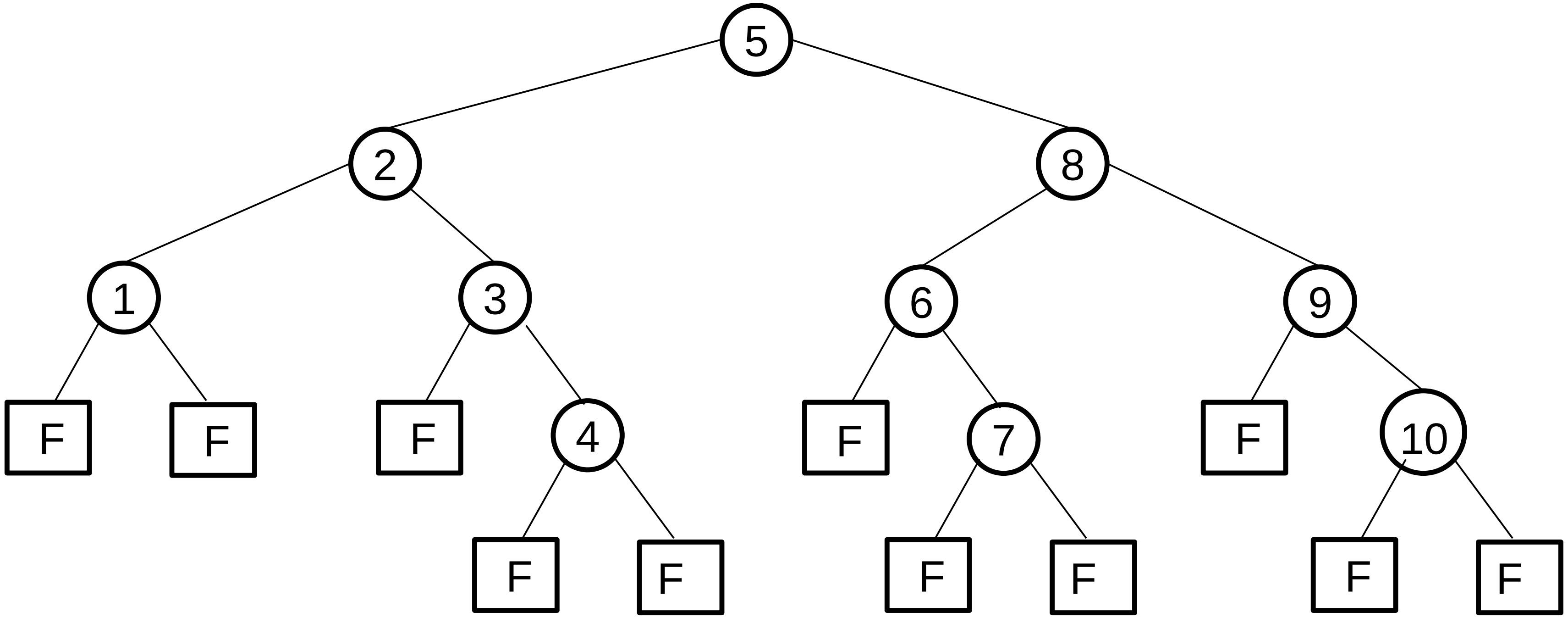
```
/*pre: list is ordered  Post: location returned, O.W. -1*/
int BinarySearch(KeyType k, List *pl){
    int middle, bottom=0, top=pl->size-1;
    KeyType mid_item;
    while(bottom<=top){
        middle=(bottom+top)/2;
        RetrieveItem(middle, &mid_item, pl)
        if ( k == mid_item )
            return middle;
        else if ( k < mid_item )
            top=middle-1;
        else
            bottom=middle+1;
    }
    return -1;
}
```

[061	087	154	170	275	426	503	509	512	612	653	677	703	765	897	908]
061	087	154	170	275	426	503	509	[512	612	653	677	703	765	897	908]
061	087	154	170	275	426	503	509	[512	612	653]	677	703	765	897	908

Another important connection between algorithms and data structures is this. Binary Search is suitable only for the contiguous implementation. However, if we have to place the data in a linked structures (linked list) and be able in the same time to fasten the search what should we do?

This will be achieved by implementing the linked list as a binary tree as we will see later. To see the connection let us see first how we analyze binary search. The idea comes from analyzing the binary search for contiguous implementation.

The analysis of Binary Search requires basic definitions and mathematical relations for trees as a mathematical structure NOT yet an ADT data structure.



1	2	3	4	5	6	7	8	9	10