Chapter 4

ADT Sorted List

Outline

- Sorted List
- Array-based Implementation
- Linked Implementation
- Comparison

ADT Sorted List

Remember the difference between an unsorted list and a sorted list?

Remember the definition of a key?

If the list is a sorted list

of names, what would be the key?

of bank balances, what would be the key?

of grades, what would be the key?

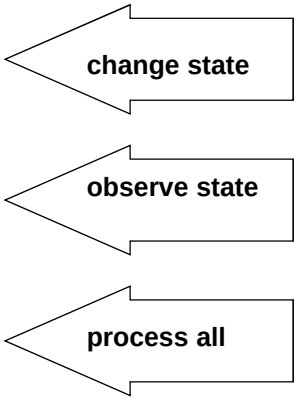
ADT Sorted List

- **Sorted list**
  - A list that is sorted by the value in the key; there is a semantic relationship among the keys of the items in the list
- **Key**
  - The attributes that are used to determine the logical order of the list

ADT Sorted List Operations

Logical Level

- **Transformers**
  - **MakeEmpty**
  - **InsertItem**
  - **DeleteItem**
- **Observers**
  - **IsFull**
  - **GetLength**
  - **RetrieveItem**
- **Iterators**
  - **ResetList**
  - **GetNextItem**

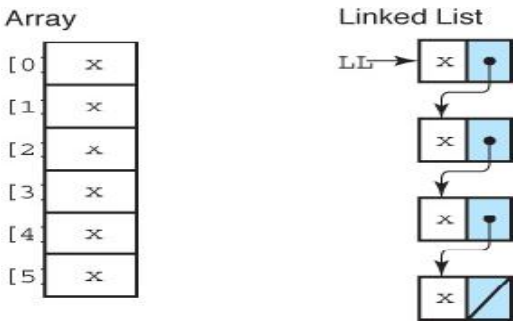


ADT Sorted List

- Client is responsible for error checking.
  - Precondition of each function enforces it.
  - Provide clients the tools with which to check for the conditions.
    - Observers
- Client provide the generic data type class **`ItemType`**

ADT Sorted List

- **Implementation Level**

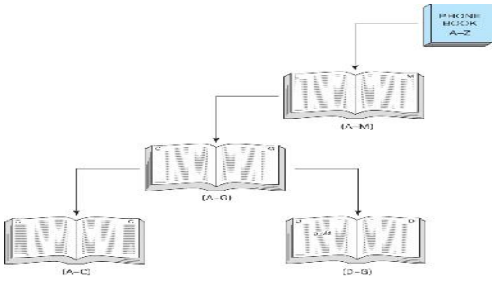


Two implementations

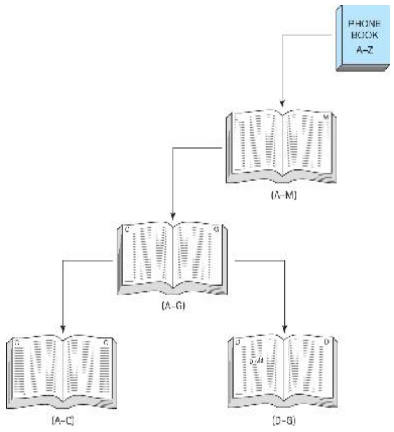
Array Implementation

- *Can we improve searching in a sorted list?*

With the Unsorted List ADT we examined each list element beginning with **`info[ 0 ],`** until we found a matching key or we examined all the elements



Binary Search Algorithm



Binary Search Algorithm

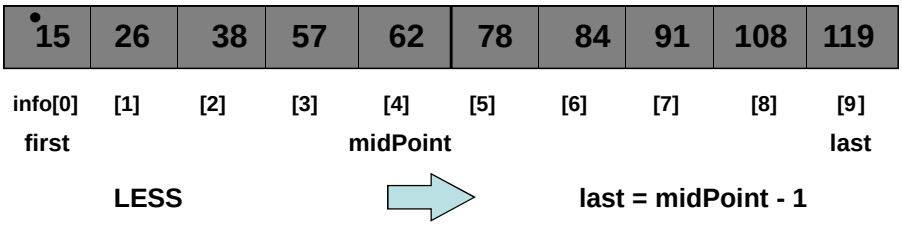
- Examine the element in the middle of the array
  - Match item?
    - Stop searching
  - Middle element too small?
    - Search second half of array
  - Middle element too large?
    - Search first half of array Is it the sought item?
- Repeat the process in half that should be examined next
- Stop when item is found or when there is nowhere else to look

Binary Search Algorithm

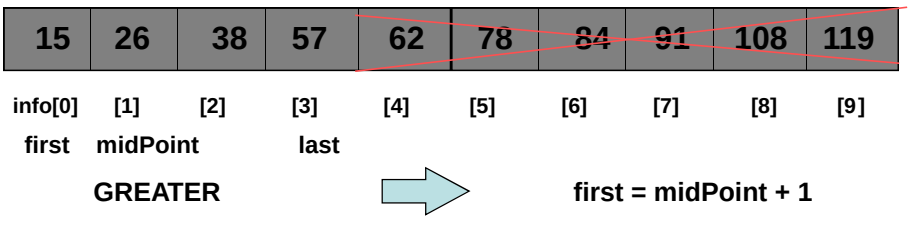
```
// Pre: Key member of item is initialized.
// Post: If found, item's key matches an element's key and a copy
//   of element has been stored in item; otherwise, item is unchanged.
void SortedType::RetrieveItem(ItemType& item, bool&found)
{
   int midPoint;
   int first  =  0;
   int last  = length - 1
   bool  moreToSearch  =  ( first  <=  last );
   found = false;
   while ( moreToSearch  &&  !found )
   {
      midPoint = ( first + last ) / 2;
      switch ( item.ComparedTo(info[midPoint]) )
      {
         case LESS     :       . . .  // LOOK IN FIRST HALF NEXT
         case GREATER  :       . . .  // LOOK IN SECOND HALF NEXT
         case  EQUAL   :       . . .  // ITEM HAS BEEN FOUND
      }
   }
}
```


Trace of Binary Search

item = 45



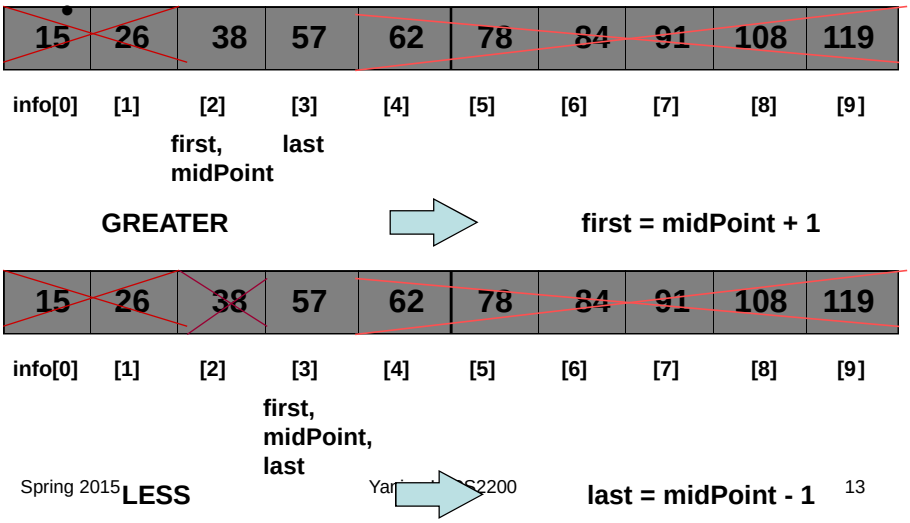
LESS → **last = midPoint - 1**



GREATER → **first = midPoint + 1**

Trace continued

item = 45



ADT Sorted List

- *Which member function specifications and implementations must change to ensure that any instance of the Sorted List ADT remains sorted at all times?*
  - **InsertItem**
  - **DeleteItem**

What about the other transformer ***`MakeEmtpy`*** *?*

Array-Based Implementation

What do you have to do to insert **Clair** *into the following list?*

1. **Anne**
2. **Betty**
3. **Mary**
4. **Susan**

Array-Based Implementation

1. Find proper location for the new element in the sorted list
2. Create space for the new element by **moving down** all the list elements that will follow it
3. Put the new element in the list
4. Increment length

```
void SortedType::InsertItem(ItemType item)
{
   bool moreToSearch;
   int location = 0;
   moreToSearch = (location < length);
   while (moreToSearch)
   {
      switch(item.ComparedTo(info[location]))
      {
         case LESS : moreToSearch = false; break;
         case GREATER : location++;
                        moreToSearch = (location < length);
                        break;
      }
   }
   for ( int index = length; index > location; index--)
        info[index] = info[index-1];
   info[location] = item;
   length++;
}
```

Find the first element which is larger than item.


```
void SortedType::DeleteItem(ItemType item)
{
   int location = 0;
   while(item.ComparedTo(info[location])!= EQUAL)
       location++;

   for ( int index = location +1; index < length; index++)
        info[index-1] = info[index];
   info[location] = item;
   length--;
}
```

Big-O of Binary Search Algorithm

- How many time we could split a list of N elements?

$$2^{\log_2 N}=N$$

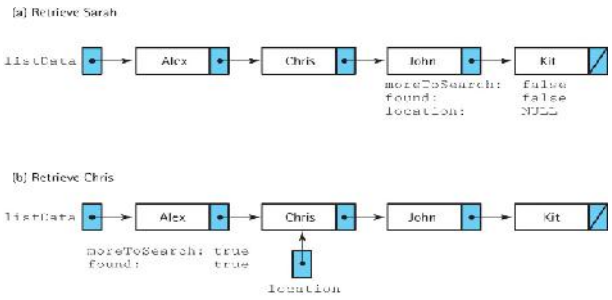
- The time complexity of Binary Search Algorithm : $O(\log_2 N)$

Array-Based Big-O Comparison

Operations	Unsorted List	Sorted List
RetrieveItem	O(n)	O(n) Linear Search $O(\log_2 n)$ binary Search
InsertItem		
Find	O(1)	O(n) search
Put	O(1)	O(n) moving down
Combined	**O(1)**	**O(n)**
DeleteItem		
Find	O(n)	O(n) search
remove	O(1) swap	O(n) moving up
Combined	**O(n)**	**O(n)**

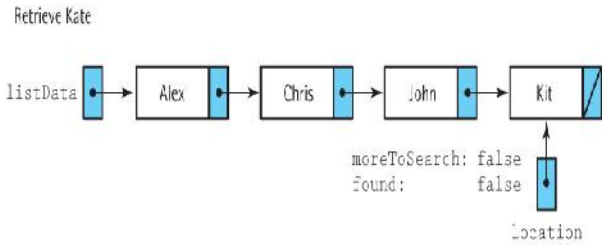
Linked Implementation

•*How about searching of linked implementation?*



Linked Implementation

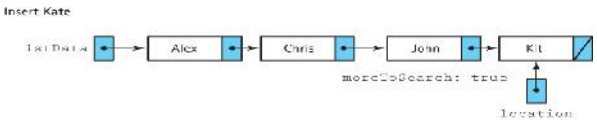
What about passing spot where item would be if there?



Linked Implementation

- *Is Inserting as easy? Let's see*

```
Set location to listData
Set moreToSearch to (location != NULL)
while  moreToSearch
    switch  (item.ComparedTo(location->info))
      case GREATER    :
                Set location to location->next
                Set moreToSearch to (location != NULL)
      case LESS          :  Set moreToSearch to false
```



See the problem ?

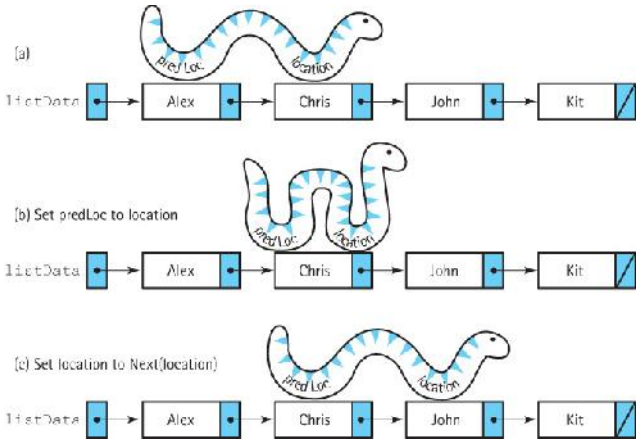
Problem of One Direction Pointer

- Use two pointers to hold the information of current locations.
  - In UnsortedType, we hold current node and one node ahead when deleting an item.

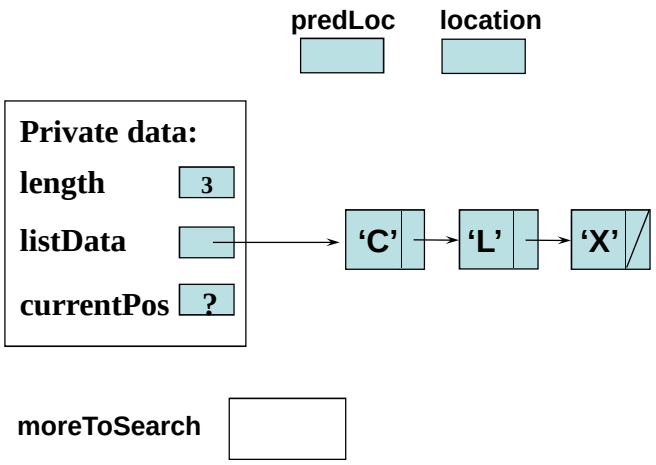
    `(location->next)->info`
  - In SortedType, we hold current node and one node behind when inserting an item.
  - Why there is a difference?

Linked Implementation

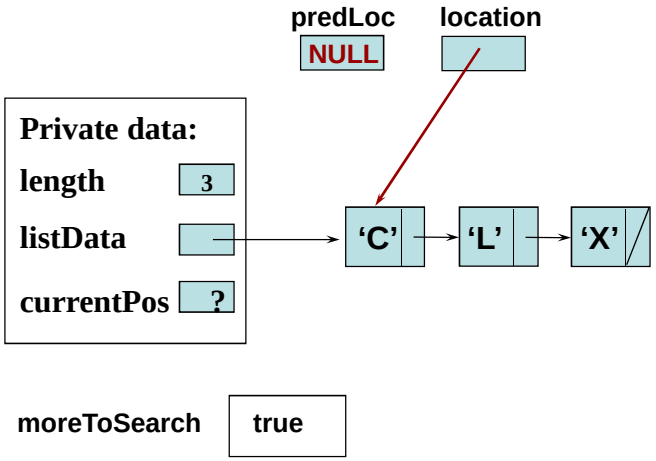
We need a trailing pointer



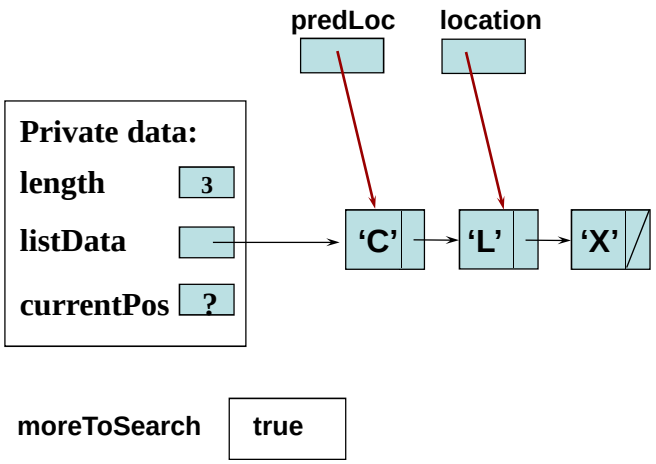
Inserting ‘S’ into a Sorted List



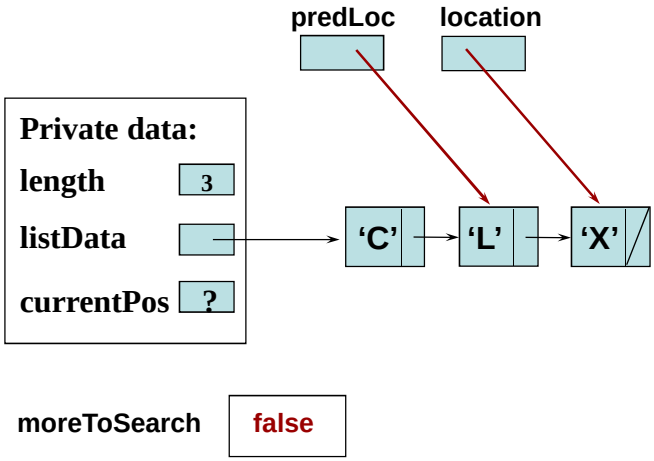
Finding proper position for ‘S’



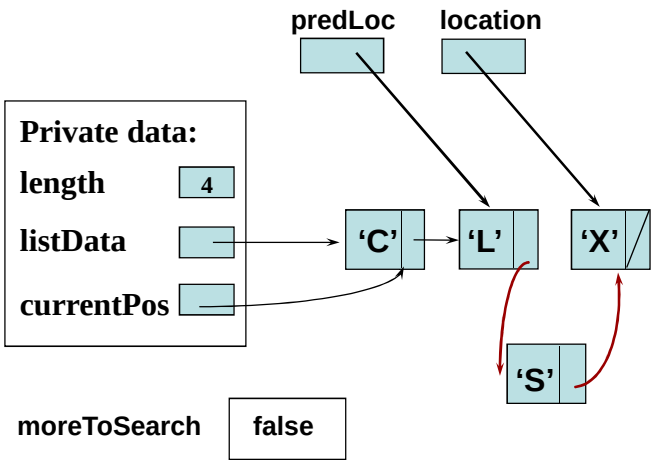
Finding proper position for ‘S’



Finding proper position for ‘S’



Inserting ‘S’ into Proper Position



Linked Implementation

- *Does DeleteItem have to be changed?*

Big-O Comparison

Which array-based operations are O(1)?

Which linked operations are O(1)?

Which array-based operations are O(N)?

Which linked operations are O(N)?

Can you say which implementation is better?

Which sorted operations differ in Big-O from unsorted operations?

Linked-List Big-O Comparison

Operations	Unsorted List	Sorted List
RetrieveItem	O(n)	O(n) Linear Search
InsertItem		
Find	O(1)	O(n) search
Break/Connect	O(1)	O(1)
Combined	**O(1)**	**O(n)**
DeleteItem		
Find	O(n)	O(n) search
Break/Connect	O(1)	O(1)
Combined	**O(n)**	**O(n)**

Sorted List Big-O Comparison

Operations	Array-based	Linked Implementation
RetrieveItem	O(n) Linear Search $O(\log_2 n)$ binary Search	O(n) Linear Search
InsertItem		
Find	O(n) search	O(n) search
Insert	O(n) moving down	O(1) break/connect
Combined	**O(n)**	**O(n)**
DeleteItem		
Find	O(n) search	O(n) search
Remove	O(n) moving up	O(1) break/connect
Combined	**O(n)**	**O(n)**

Reference

- Reproduced from C++ Plus Data Structures, 4th edition by Nell Dale**.**
- **Reproduced by permission of Jones and Bartlett Publishers International.**