Data Structure

Sorted List (ADT)

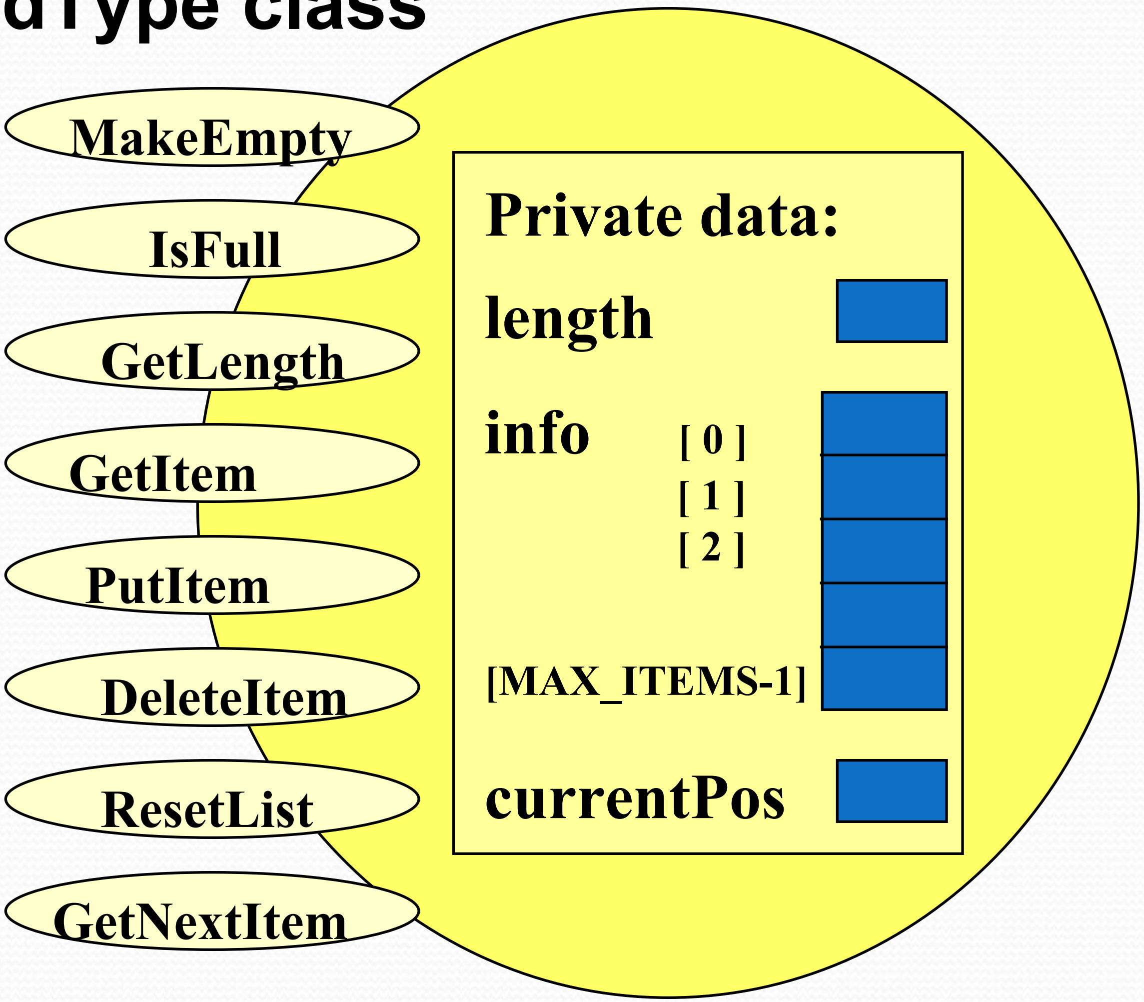
Lecture-07

Object-oriented vs. Top Down

- Object-oriented design
  - focuses on the data objects that are to be transformed
    - resulting in a hierarchy of objects
  - nouns are the primary focus
    - **nouns** in objects;
    - **verbs** become operations
- Top-down design
  - focus on the process of transforming the input into the output,
    - resulting in a hierarchy of tasks
  - verbs are the primary focus

Sorted Type Class Interface Diagram

SortedType class



Specification of SortedType

Structure:	**The list has a special property called the *current position* - the position of the last element accessed by GetNextItem during an iteration through the list. Only ResetList and GetNextItem affect the current position.**
Operations (provided by Unsorted List ADT):	
MakeEmpty	
Function	Initializes list to empty state.
Precondition	
Postcondition	List is empty.
Boolean IsFull	
Function	Determines whether list is full.

Specification of SortedType

int GetLength	
Function	Determines the number of elements in list.
Precondition	List has been initialized.
Postcondition	Returns the number of elements in list.
ItemType GetItem (ItemType item, Boolean &found)	
Function	Retrieves list element whose key matches item's key (if present).
Precondition	List has been initialized. Key member of item is initialized.
Postcondition	If there is an element some Item whose key matches item's key, then found = true and item is a copy of someItem; otherwise found = false and item is unchanged. **List is unchanged.**
PutItem (ItemType item)	

Specification of SortedType

DeleteItem (ItemType item)	
Function	Deletes the element whose key matches item's key.
Precondition	List has been initialized. Key member of item is initialized. One and only one element in list has a key matching item's key.
Post-condition	No element in list has a key matching item's key. **List is still sorted.**
ResetList	
Function	Initializes current position for an iteration through the list.
Precondition	List has been initialized.
Post-condition	Current position is prior to first element in list.
ItemType GetNextItem ()	

Member functions

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

- **PutItem**
- **DeleteItem**

InsertItem algorithm for SortedList ADT

- **Find proper location for the new element in the sorted list.**

- **Create space for the new element by moving down all the list elements that will follow it.**

- **Put the new element in the list.**

- **Increment length.**

```
template<class ItemType>
class sortedtype
{
private:
       int length;
        ItemType info[MAX_ITEM];
      int currentPos;
 public:
       sortedtype();
       void MakeEmpty();
       void InsertItem(ItemType);
       void DeleteItem(ItemType);
       bool isFull();
       int LengthIs();
   void RetriveItem(ItemType&, bool&);
       void ResetList();
       void GetNextItem(ItemType&);
       ~sortedtype();
};
```

Constructor

```
template <class ItemType>
sortedtype<ItemType>::sortedtype()
{
   length = 0;
   currentPos = - 1;
}
```

```
template <class ItemType>
void sortedtype<ItemType>::MakeEmpty()
{
   length=0;
}
```

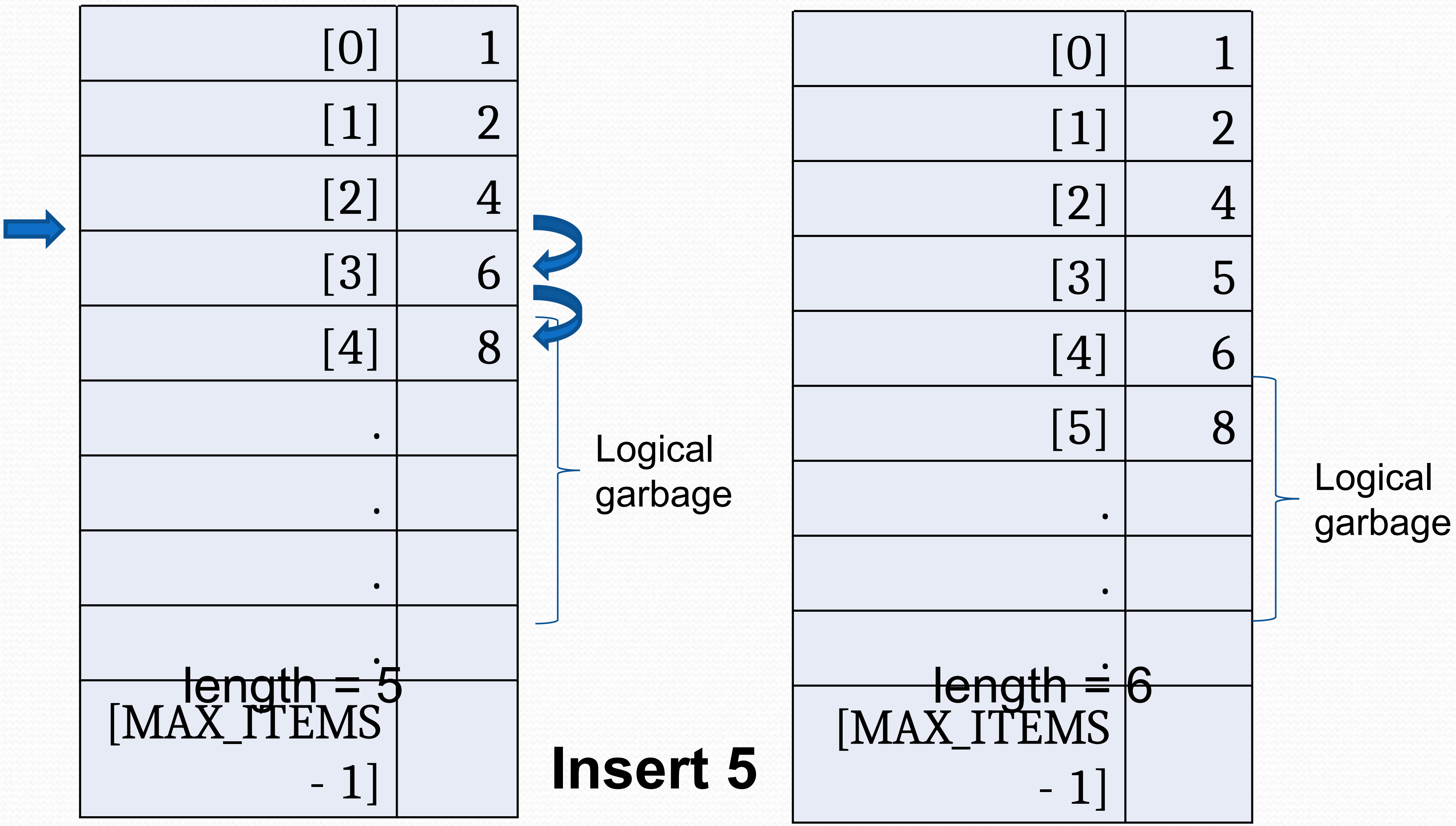
```
template <class ItemType>
bool sortedtype<ItemType>::isFull()
{
   return(length==MAX_ITEM);
}
```

```
template <class ItemType>
int sortedtype<ItemType>::LengthIs()
{
   return length;
}
```

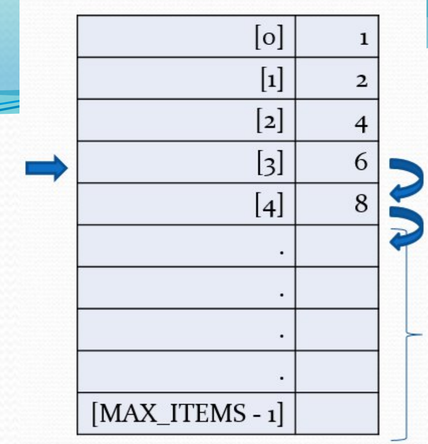
```
template <class ItemType>
void sortedtype<ItemType>::ResetList()
{
   currentPos=-1;
}
```

```
template <class ItemType>
void sortedtype<ItemType>::GetNextItem(ItemType& item)
{
currentPos++;
item=info[currentPos];
}
```

Inserting an Item into Sorted List



sortedtype.cpp



length = 5

Insert 5

```
template <class ItemType>
void SortedType<ItemType>::InsertItem(ItemType item)
{
     int location = 0;
     bool moreToSearch = (location < length);

     while (moreToSearch)
// This will identify the location where the item will be stored
     {
          if(item > info[location])
          {
               location++;
               moreToSearch = (location < length);
          }
          else if(item < info[location])
               moreToSearch = false;
     }
     for (int index = length; index > location; index--)
          info[index] = info[index - 1];
     info[location] = item;
     length++;
}
```

sortedtype.cpp

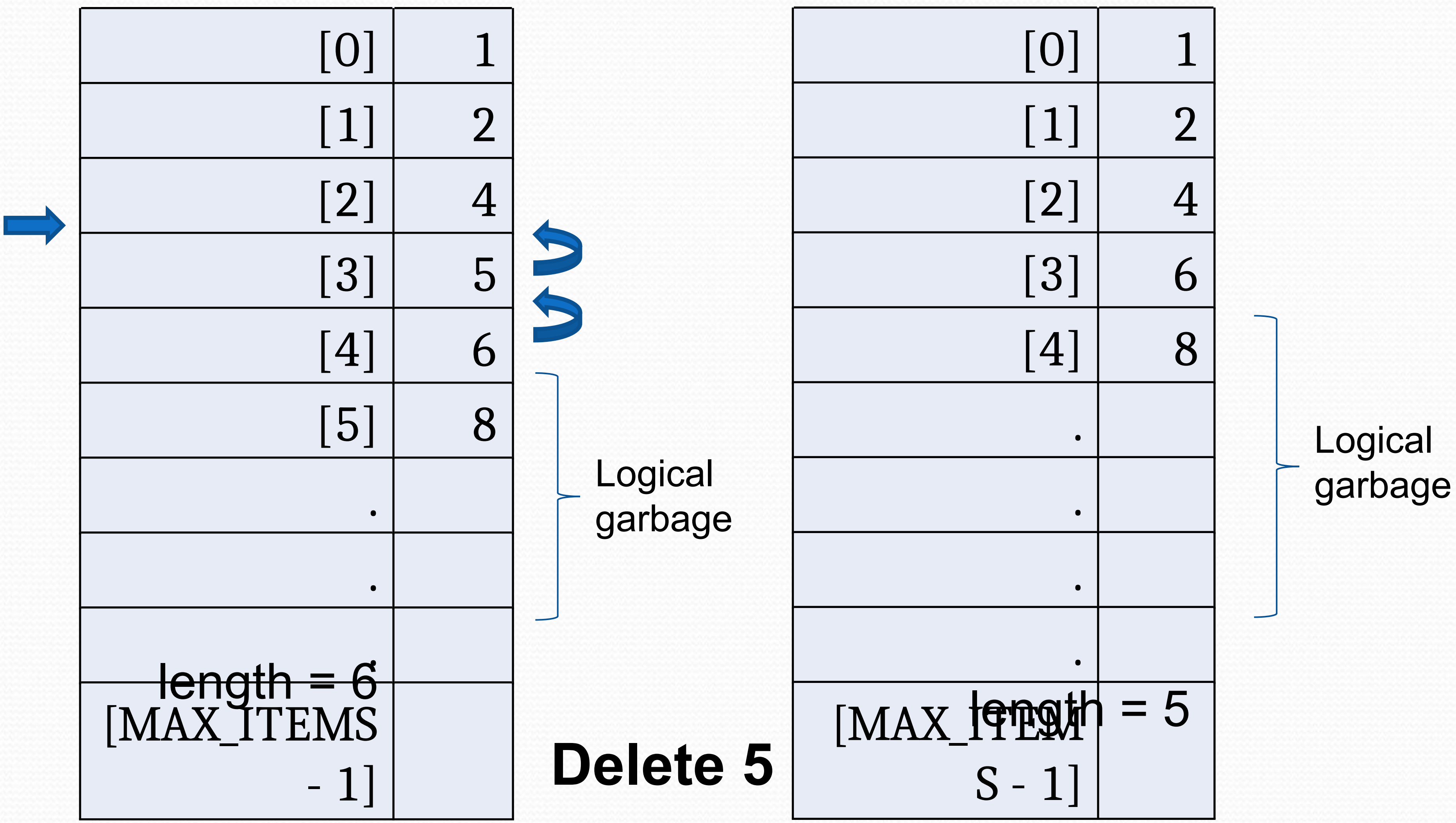
```
template <class ItemType>
void SortedType<ItemType>::InsertItem(ItemType item)
{
    int location = 0;
    bool moreToSearch = (location < length);

    while (moreToSearch)
    {
        if(item > info[location])
        {
            location++;
            moreToSearch = (location < length);
        }
        else if(item < info[location])
            moreToSearch = false;
    }
    for (int index = length; index > location; index--)
        info[index] = info[index - 1];
    info[location] = item;
    length++;
}
```

O(N)

O(N)

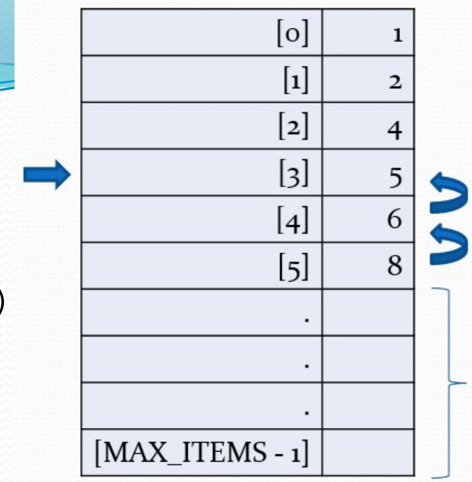
O(N)

Deleting an Item from Sorted List



sortedtype.cpp

```
template <class ItemType>
void SortedType<ItemType>::DeleteItem(ItemType item)
{
     int location = 0;

     while (item != info[location])
          location++;


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



length = 6

Delete 5

sortedtype.cpp

```
template <class ItemType>
void SortedType<ItemType>::DeleteItem(ItemType item)
{
    int location = 0;

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

O(N)

O(N)

O(N)

```
void  SortedType :: DeleteItem ( ItemType  item )
{
  int location = 0;
  //   find location of element to be deleted
  while ( item.ComparedTo ( info[location] )!=  EQUAL )
       location++;
  // move up elements that follow deleted item in sorted list
  for ( int  index = location + 1 ; index  <  length; index++ )
       info [ index - 1 ] = info [ index ];
   length--;
}
```

DeleteItem algorithm for SortedList ADT

- Find the location of the element to be deleted from the sorted list.
- Eliminate space occupied by the item by moving up all the list elements that follow it.
- Decrement length.

```
template <class ItemType>
void sortedtype<ItemType>::RetriveItem(ItemType& item, bool& found)
{
   int midPoint, first = 0, last = length - 1;
   bool moreToSearch = (first <= last);
   found = false;
   while (moreToSearch && !found)
   {
      midPoint = (first + last) / 2;
      if(item < info[midPoint])
      {
         last = midPoint - 1;
         moreToSearch = (first <= last);
      }
      else if(item > info[midPoint])
      {
         first = midPoint + 1;
         moreToSearch = (first <= last);
      }
      else
      {
         found = true;
         item = info[midPoint];
      }
   } }
```

Retrieving an Item from Sorted List

- Find **84**

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Retrieving an Item from Sorted List

- Find **84**

6	**13**	**14**	**25**	**33**	**43**	**51**	**53**	**64**	**72**	**84**	**93**	**95**	**96**	**97**
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

first **last**

- **Step 1**

Retrieving an Item from Sorted List

- Find **84**

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

first

mid =(0+14)/2

last

- **Step 1**

Retrieving an Item from Sorted List

- Find **84**

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

first

mid =(0+14)/2

last

- **Step 1**

Retrieving an Item from Sorted List

- Find **84**

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

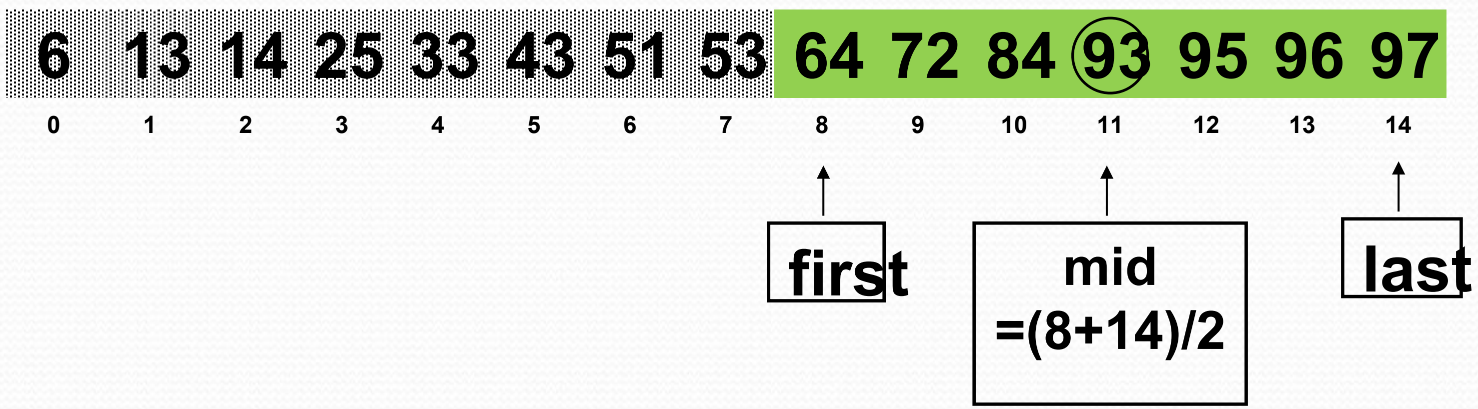
first

last

- **Step 2**

Retrieving an Item from Sorted List

- Find **84**



- **Step 2**

Retrieving an Item from Sorted List

- Find **84**

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

first

mid =(8+14)/2

last

- **Step 2**

Retrieving an Item from Sorted List

- Find **84**

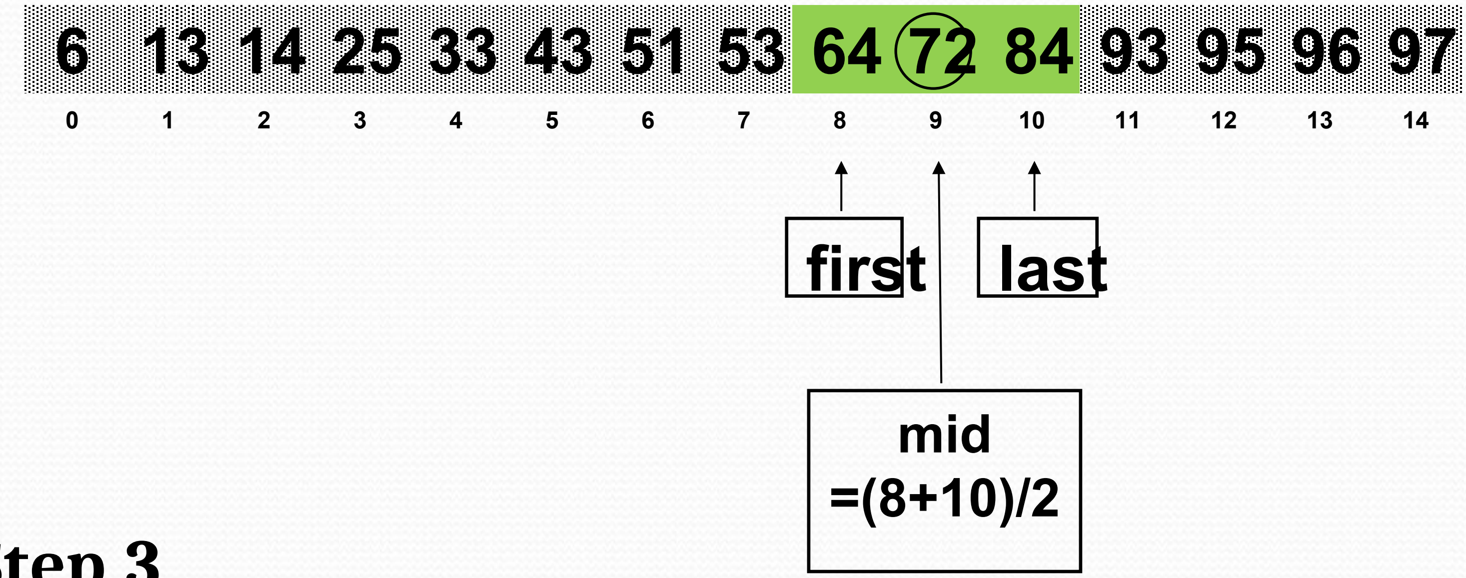
6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

first last

- **Step 3**

Retrieving an Item from Sorted List

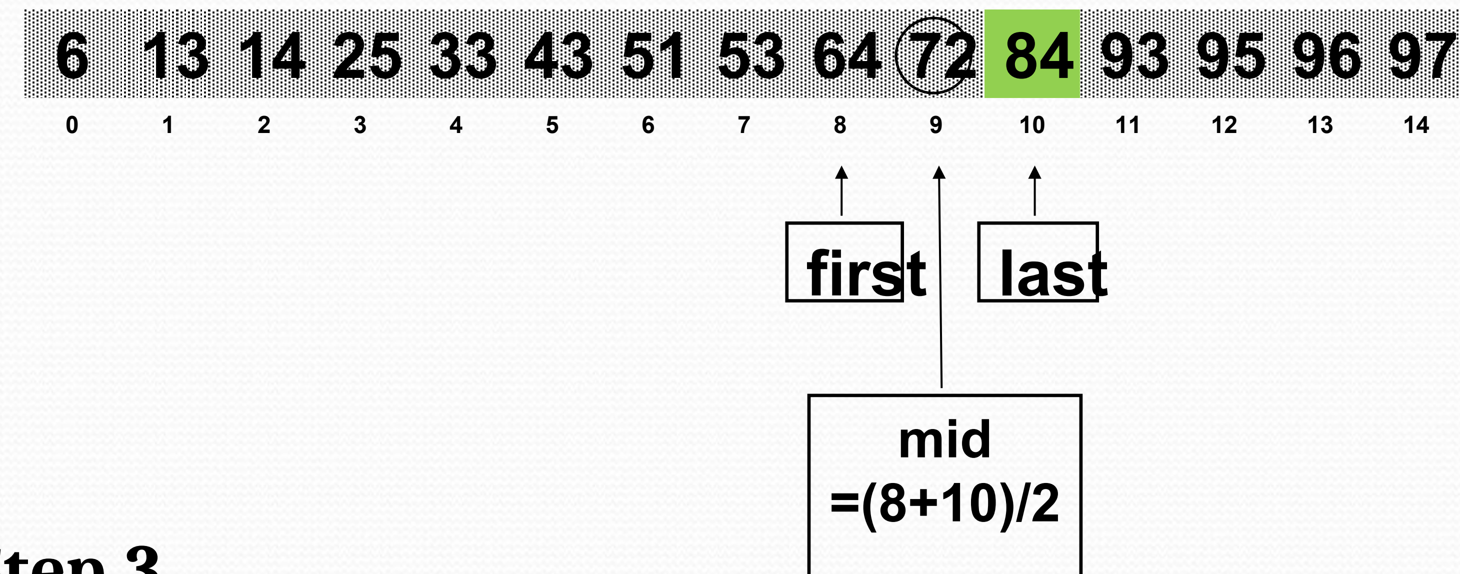
- Find **84**



- **Step 3**

Retrieving an Item from Sorted List

- Find **84**



- **Step 3**

Retrieving an Item from Sorted List

- Find **84**

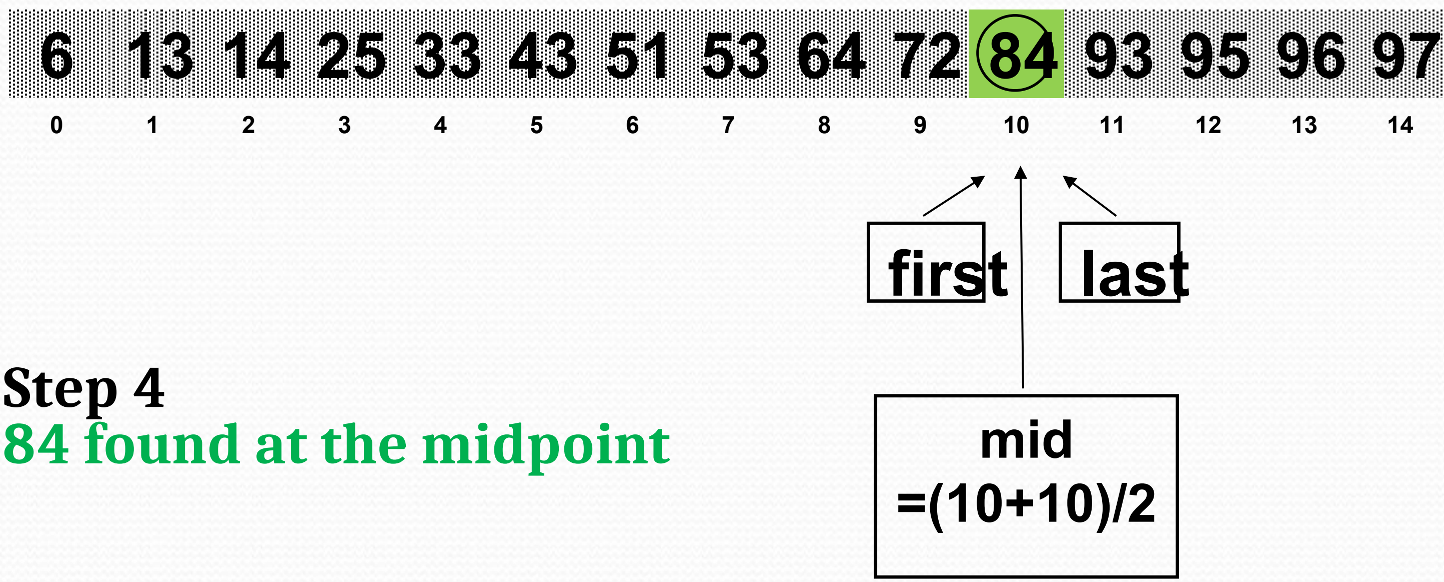
6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

first last

- **Step 4**

Retrieving an Item from Sorted List

- Find **84**



- **Step 4**
- **84 found at the midpoint**

Retrieving an Item from Sorted List

- Find **73**

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Retrieving an Item from Sorted List

- Find **73**

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

first

last

- **Step 1**

Retrieving an Item from Sorted List

- Find **73**

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

first

mid =(0+14)/2

last

- **Step 1**

Retrieving an Item from Sorted List

- Find **73**

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

first

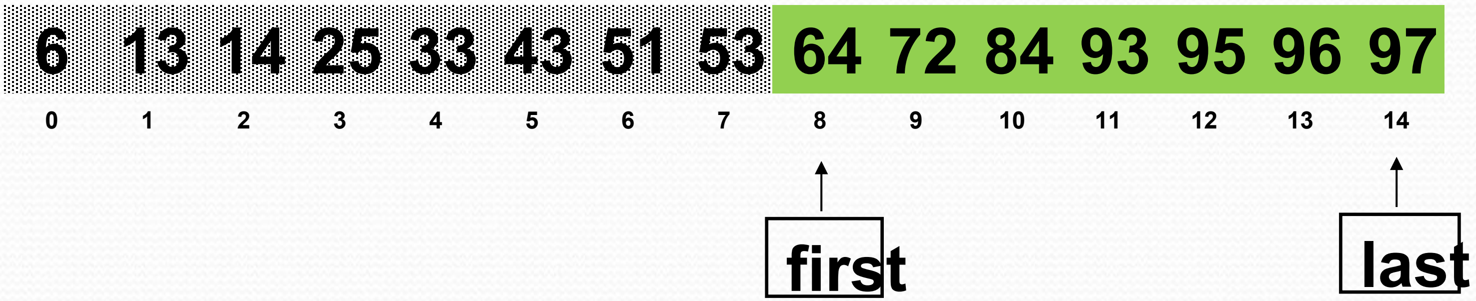
mid
=(0+14)/2

last

- **Step 1**

Retrieving an Item from Sorted List

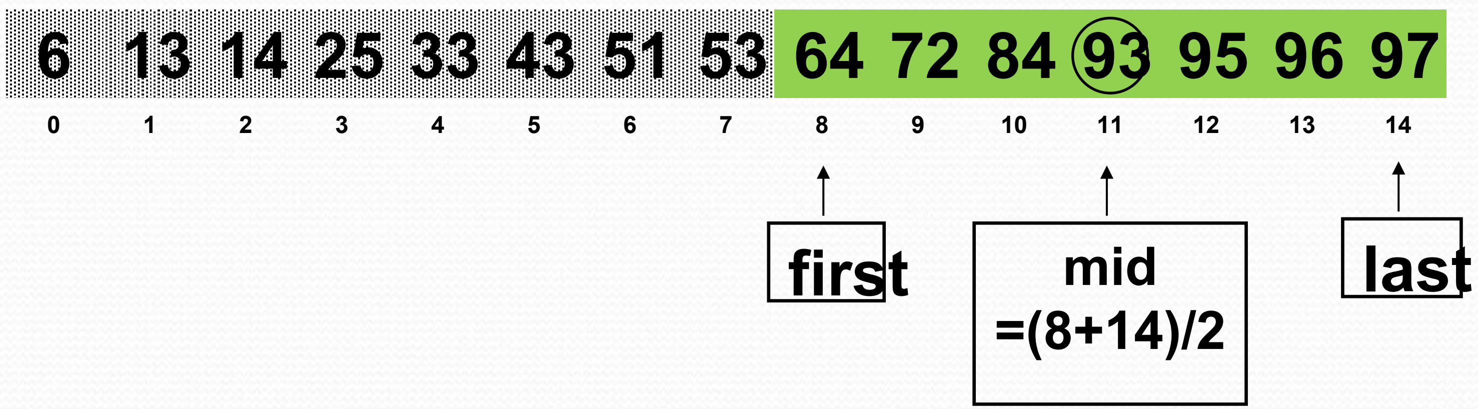
- Find **73**



- **Step 2**

Retrieving an Item from Sorted List

- Find **73**



- **Step 2**

Retrieving an Item from Sorted List

- Find **73**

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

first

mid =(8+14)/2

last

- **Step 2**

Retrieving an Item from Sorted List

- Find **73**

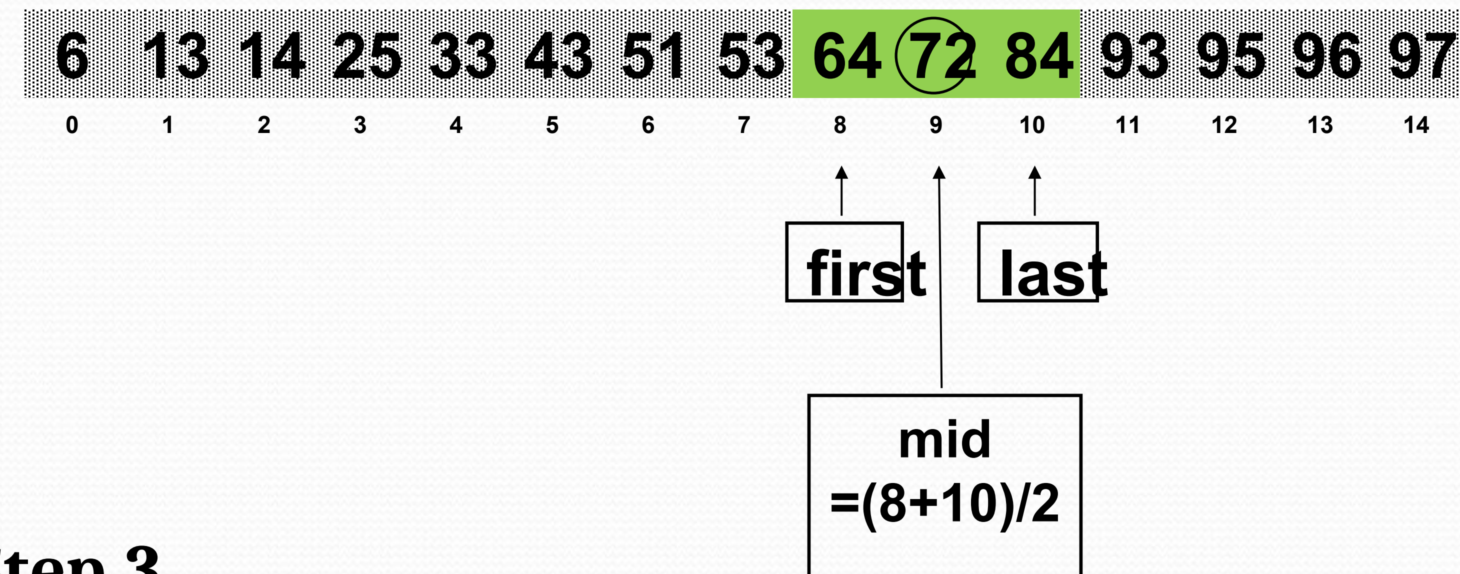
6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

first last

- **Step 3**

Retrieving an Item from Sorted List

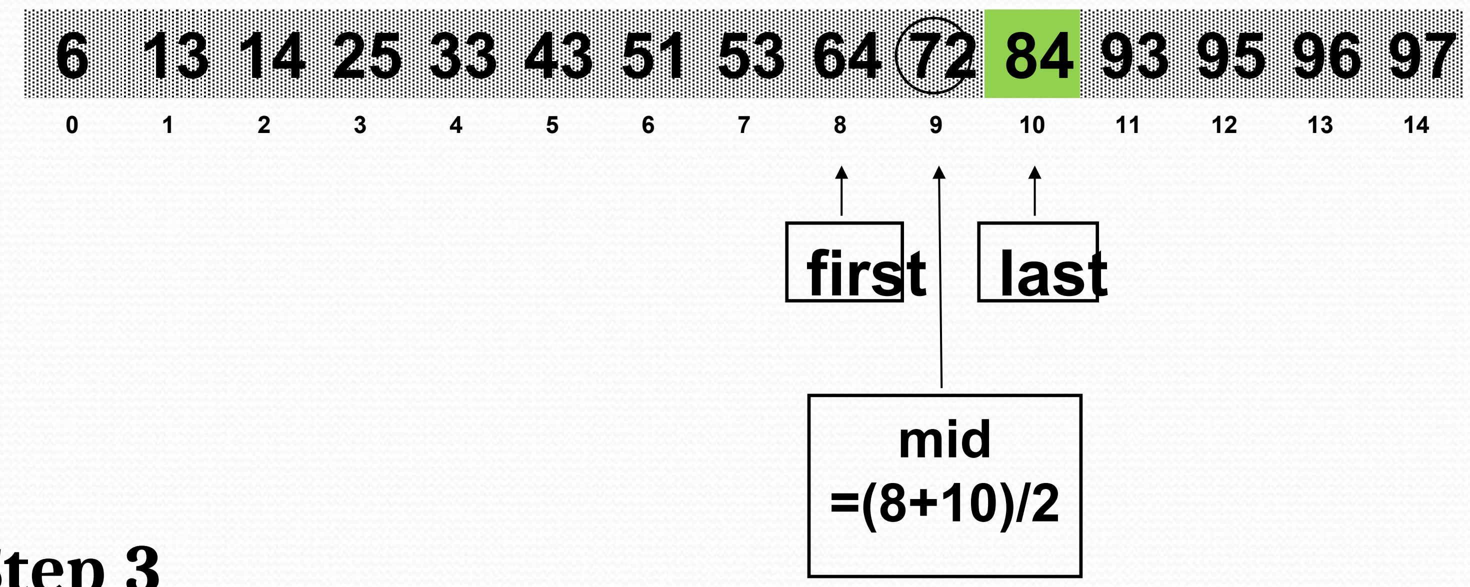
- Find **73**



- **Step 3**

Retrieving an Item from Sorted List

- Find **73**



- **Step 3**

Retrieving an Item from Sorted List

- Find **73**

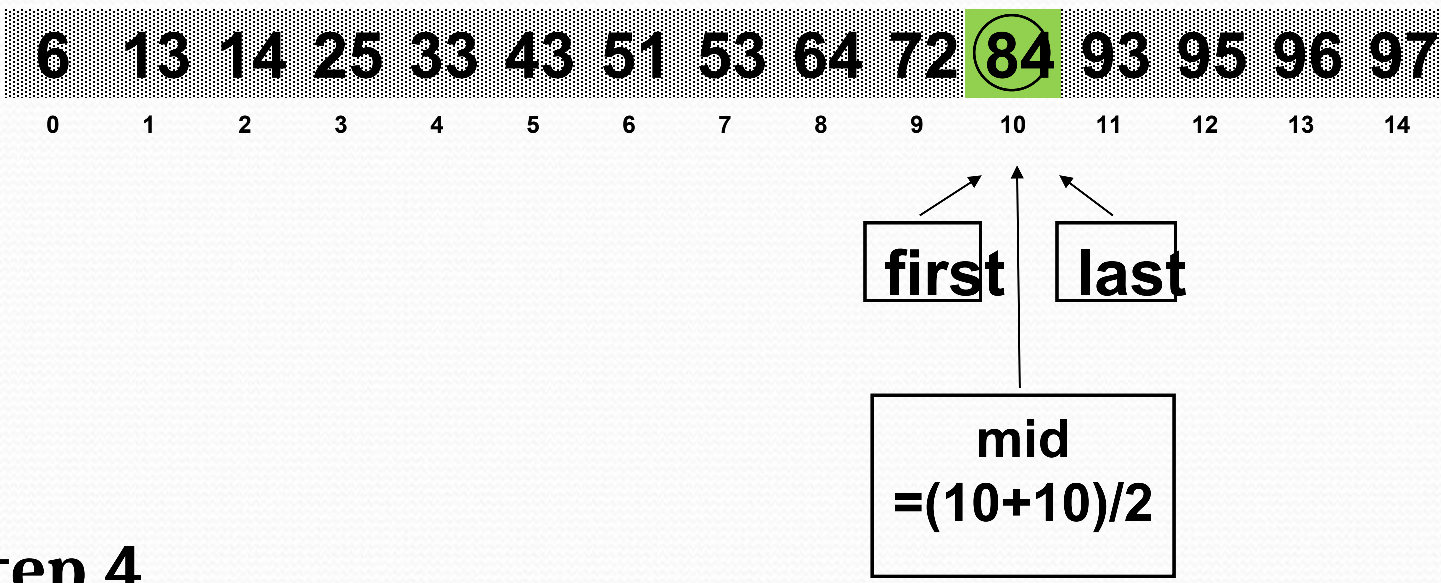
6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

first last

- **Step 4**

Retrieving an Item from Sorted List

- Find **73**



- **Step 4**

Retrieving an Item from Sorted List

- Find **73**

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

last first

- **Step 5**
- **last < first (indicates the absence of the item)**

Binary Seach in a Sorted List

- **Examines the element in the middle of the array.**
  Is it the sought item?
  If so, stop searching.
  Is the middle element too small?
  Then start looking in second half of array.
  Is the middle element too large?
  Then begin looking in first half of the array.
- **Repeat the process in the half of the list that should be examined next.**
- **Stop when item is found, or when there is nowhere else to look and item has not been found.**

```
ItemType  SortedType::GetItem ( ItemType  item,   bool&  found )
//  Pre: Key member of item is initialized.
//  Post: If found, item's key matches an element's key in the list
// and a copy of that element is returned; otherwise,
//  original item is returned.
{  int  midPoint;
   int  first =  0;
    int last  = length - 1;

   bool moreToSearch  =  ( first  <=  last );
    found = false;

   while ( moreToSearch  &&  !found )
   {     midPoint  =  ( first + last ) / 2 ;   // INDEX OF MIDDLE ELEMENT
         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

15	26	38	57	62	78	84	91	108	119
info[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
first				midPoint					last

LESS — last = midPoint - 1

15	26	38	57	~~62~~	~~78~~	~~84~~	~~91~~	~~108~~	~~119~~
info[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
first	midPoint		last						

GREATER — first = midPoint + 1

Trace continued

item = 45

~~15~~	~~26~~	38	57	~~62~~	~~78~~	~~84~~	~~91~~	~~108~~	~~119~~
info[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
		first, midPoint	last						

GREATER → first = midPoint + 1

~~15~~	~~26~~	~~38~~	57	~~62~~	~~78~~	~~84~~	~~91~~	~~108~~	~~119~~
info[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
			first, midPoint, last						

LESS → last = midPoint - 1

Trace concludes

item = 45

info[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
15	26	38	57	62	78	84	91	108	119
		last	first						

first > last $\Rightarrow$ **found = false**

```
ItemType SortedType::GetItem ( ItemType  item,   bool&  found )
//  ASSUMES info ARRAY SORTED IN ASCENDING ORDER
{  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      :    last = midPoint - 1;
                                    moreToSearch = ( first <= last );
                                     break;
               case  GREATER   :   first = midPoint + 1;
                                    moreToSearch = ( first <= last );
                                     break;
               case  EQUAL     :   found = true ;
                                     item = info[ midPoint ];
                                     break;
          }
    }return item;
}
```

Allocation of memory

STATIC ALLOCATION

Static allocation is the allocation of memory space at compile time.

DYNAMIC ALLOCATION

Dynamic allocation is the allocation of memory space at run time by using operator new.

3 Kinds of Program Data

- **STATIC DATA**: **memory allocation exists throughout execution of program.**
  ```
  static long SeedValue;
  ```

- **AUTOMATIC DATA**: **automatically created at function entry, resides in activation frame of the function, and is destroyed when returning from function.**

- **DYNAMIC DATA**: **explicitly allocated and deallocated during program execution by C++ instructions written by programmer using unary operators `new` and `delete`**

Arrays created at run time

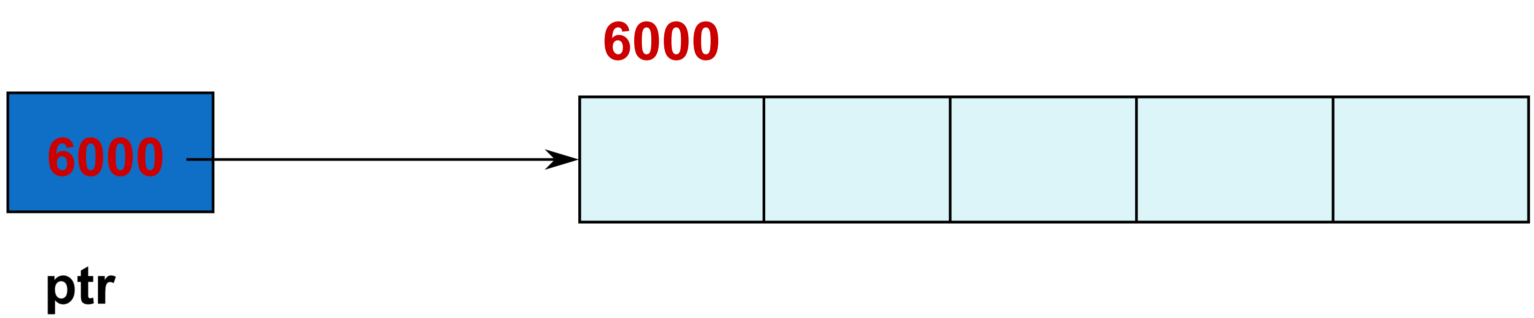
If memory is available in an area called the free store (or heap), operator new allocates memory for the object or array and returns the address of (pointer to) the memory allocated.

Otherwise, the `NULL` pointer 0 is returned.

The dynamically allocated object exists until the delete operator destroys it.

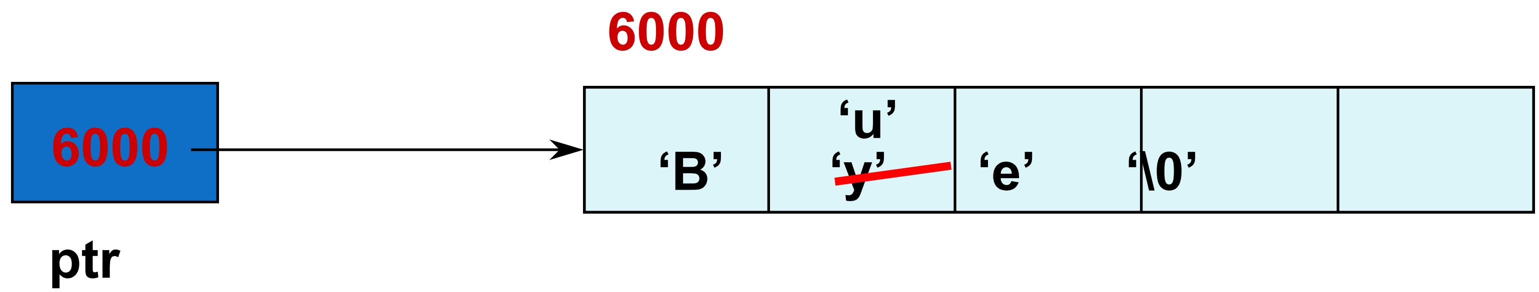
Dynamic Array Allocation

```
char  *ptr;            // ptr is a pointer variable that
                       //  can hold the address of a char

ptr  =  new  char[ 5 ];
              // dynamically, during run time, allocates
              // memory for 5 characters and places into
// the contents of ptr their beginning address
```

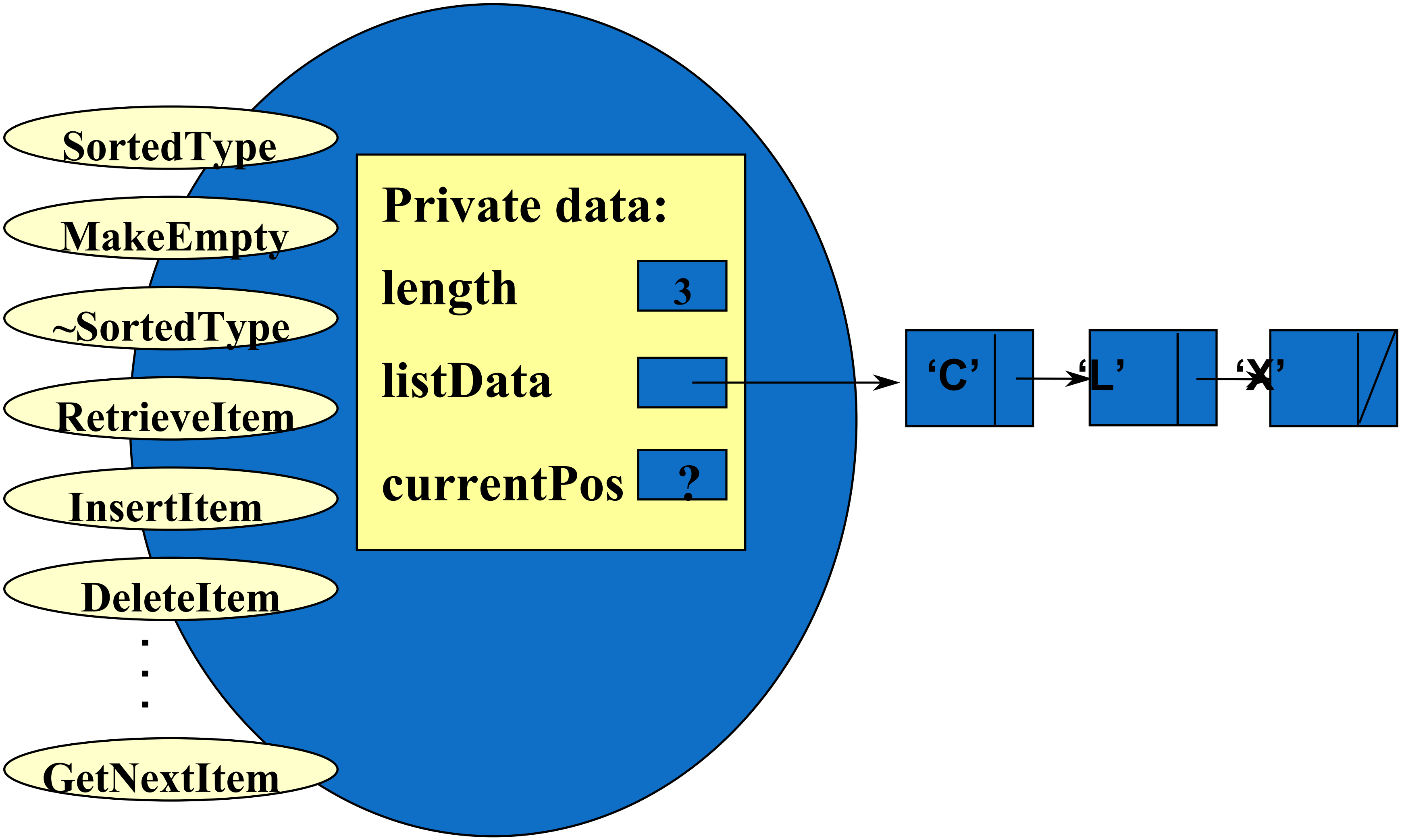


Dynamic Array Allocation

```
char  *ptr ;

ptr  =  new  char[ 5 ];

strcpy( ptr, “Bye” );

ptr[ 1 ] = ‘u’; // a pointer can be subscripted

std::cout  << ptr[ 2] ;
```



class SortedType<char>



InsertItem algorithm for Sorted Linked List

- **Find proper position for the new element in the sorted list using two pointers predLoc and location, where predLoc trails behind location.**
- **Obtain a node for insertion and place item in it.**
- **Insert the node by adjusting pointers.**
- **Increment length.**

Why is a destructor needed?

When a local list variable goes out of scope, the memory space for data member listPtr is deallocated.

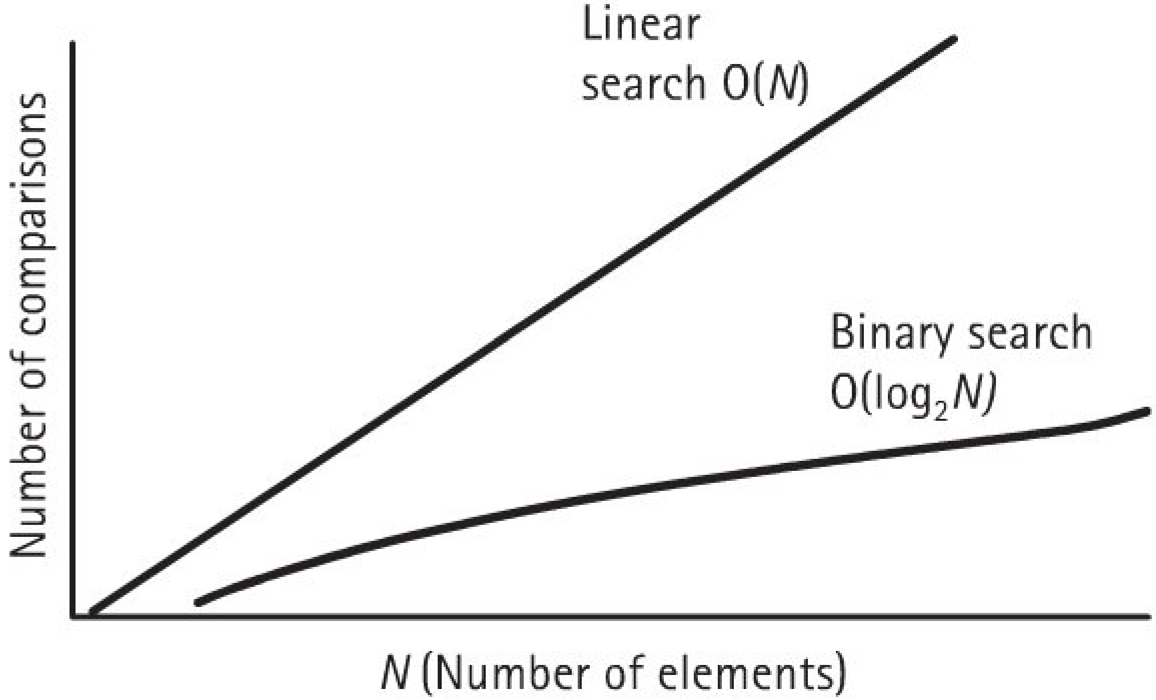
But the nodes to which listPtr points are not deallocated.

A class destructor is used to deallocate the dynamic memory pointed to by the data member.

Implementing the Destructor

```
SortedType::~SortedType()
// Post: List is empty; all items have
// been deallocated.
{
  NodeType* tempPtr;
  while (listData != NULL)
  {
    tempPtr = listData;
    listData = listData->next;
    delete tempPtr;
  }
}
```

How do the SortedList implementations compare?



SortedList implementations comparison

Array Implementation		Linked Implementation
O(1)	class constructor	O(1)
O(1)	**MakeEmpty**	**O(*N*)**
O(1)	IsFull	O(1)
O(1)	GetLength	O(1)
O(1)	ResetList	O(1)
O(1)	GetNextItem	O(1)
O(*N*)*	GetItem	O(*N*)
	PutItem	
O(*N*)*	Find	O(*N*)
O(*N*)	**Put**	**O(1)**
O(*N*)	Combined	O(*N*)
	DeleteItem	
O(*N*)*	Find	O(*N*)
O(*N*)	**Delete**	**O(1)**
O(*N*)	Combined	O(*N*)

*$O(\log_2 N)$ if a binary search is used.

List implementations comparison

Function	Unsorted List ADT Array-based	Unsorted List ADT Linked	Sorted List ADT Array-based	Sorted List ADT Linked
Class constructor	O(1)	O(1)	O(1)	O(1)
MakeEmpty	O(1)	O(*N*)	O(1)	O(*N*)
IsFull	O(1)	O(1)	O(1)	O(1)
GetLength	O(1)	O(1)	O(1)	O(1)
ResetList	O(1)	O(1)	O(1)	O(1)
GetNextItem	O(1)	O(1)	O(1)	O(1)
GetItem	O(*N*)	O(*N*)	linear search O(*N*) binary search O($\log_2 N$)	O(*N*)
PutItem				
Find	O(1)	O(1)	O(*N*)	O(*N*)
Put	O(1)	O(1)	O(*N*)	O(1)
Combined	O(1)	O(1)	O(*N*)	O(*N*)
DeleteItem				
Find	O(*N*)	O(*N*)	O(*N*)	O(*N*)
Delete	O(1)	O(1)	O(*N*)	O(1)
Combined	O(*N*)	O(*N*)	O(*N*)	O(*N*)