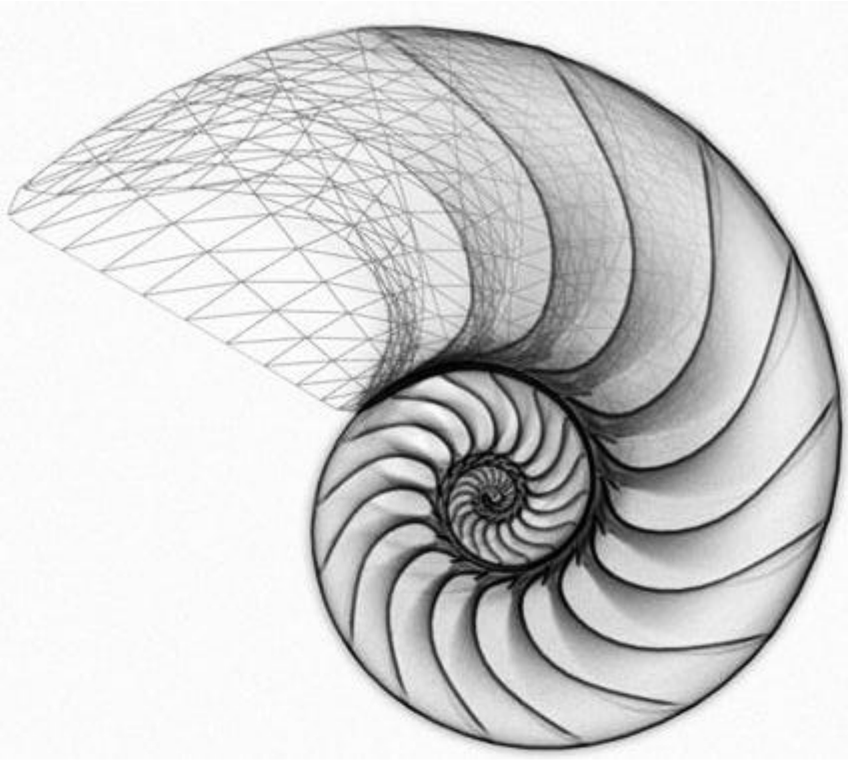


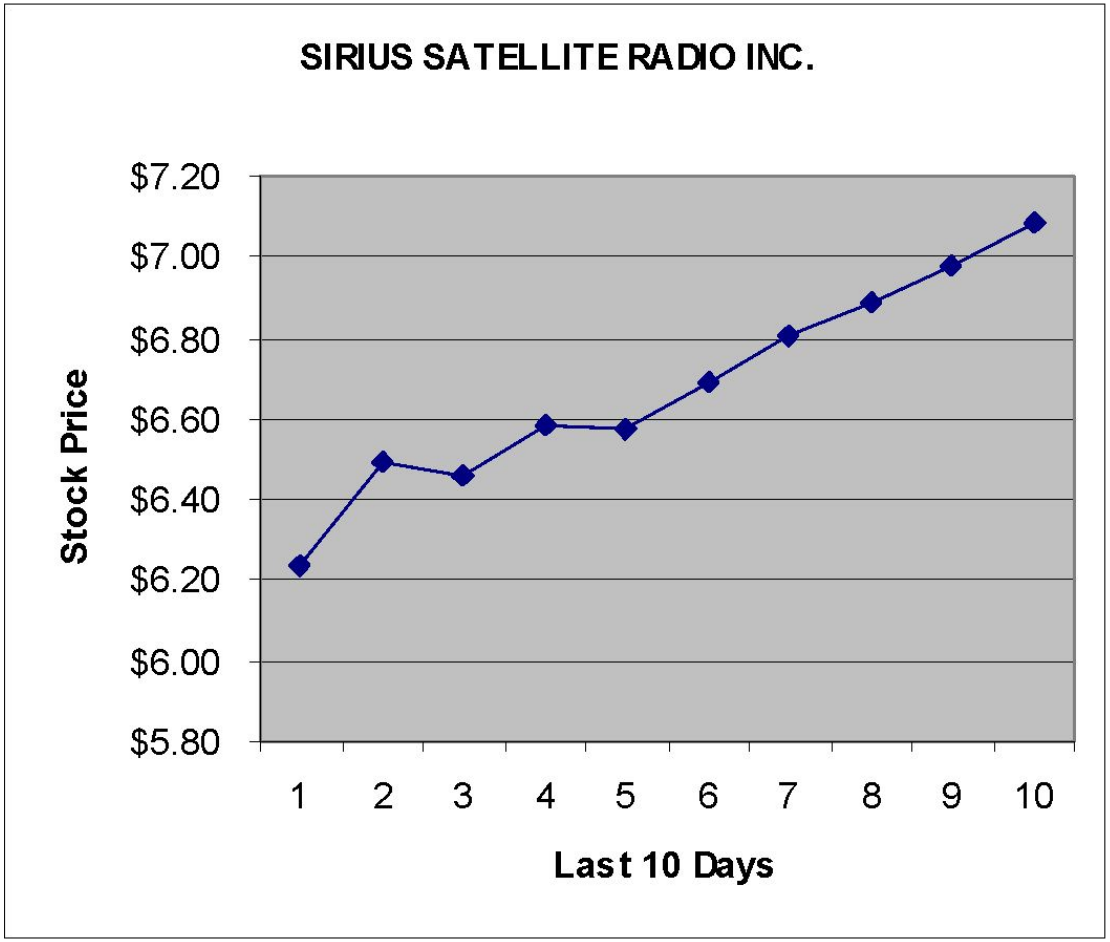
Lecture 08

Data vs. Information

Data

- 6.34
- 6.45
- 6.39
- 6.62
- 6.57
- 6.64
- 6.71
- 6.82
- 7.12
- 7.06

Information



Lists

- **Unsorted list:**
  - A list in which data items are placed in no particular order.
- **Sorted List:**
  - A list in which data items are placed in a particular order.
  - Key: a member of the class whose value is used to determine the order of the items in the list.

Unsorted List	Sorted List
22	12
12	14
46	22
35	35
14	46
. . . .	. . . .

Sorted List

ID	Name	Address
22	Jack Black	120 S. Virginia Street
45	Simon Graham	6762 St Petersburg
59	Susan O'Neal	1807 Glenwood, Palm Bay
66	David peterson	1207 E. Georgetown

Key

Specification of UnsortedType

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.
Precondition	List has been initialized.
Postcondition	Returns true if list is full and false otherwise.

Specification of UnsortedType

int LengthIs	
Function	Determines the number of elements in list.
Precondition	List has been initialized.
Postcondition	Returns the number of elements in list.
RetrieveItem (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 someItem 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.
InsertItem (ItemType item)	
Function	Adds item to list.
Precondition	List has been initialized. List is not full. item is not in list.
Postcondition	item is in list.

Specification of UnsortedType

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.
Postcondition	No element in list has a key matching item's key.
ResetList	
Function	Initializes current position for an iteration through the list.
Precondition	List has been initialized.
Postcondition	Current position is prior to first element in list.
GetNextItem (ItemType& item)	
Function	Gets the next element in list.
Precondition	List has been initialized. Current position is defined. Element at current position is not last in list.
Postcondition	Current position is updated to next position. item is a copy of element at current position.

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 Sorted List ADT):	
MakeEmpty	
Function	Initializes list to empty state.
Precondition	
Postcondition	List is empty.
Boolean IsFull	
Function	Determines whether list is full.
Precondition	List has been initialized.
Postcondition	Returns true if list is full and false otherwise.

Specification of SortedType

int LengthIs	
Function	Determines the number of elements in list.
Precondition	List has been initialized.
Postcondition	Returns the number of elements in list.
RetrieveItem (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 someItem 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.
InsertItem (ItemType item)	
Function	Adds item to list.
Precondition	List has been initialized. List is not full. item is not in list.
Postcondition	item is in list. List is still sorted.

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.
Postcondition	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.
Postcondition	Current position is prior to first element in list.
GetNextItem (ItemType& item)	
Function	Gets the next element in list.
Precondition	List has been initialized. Current position is defined. Element at current position is not last in list.
Postcondition	Current position is updated to next position. item is a copy of element at current position.

unsortedtype.h

```
#ifndef UNSORTEDTYPE_H_INCLUDED
#define UNSORTEDTYPE_H_INCLUDED

const int MAX_ITEMS = 5;

template <class ItemType>
class UnsortedType
{
    public :
        UnsortedType();
        void MakeEmpty();
        bool IsFull();
        int LengthIs();
        void InsertItem(ItemType);
        void DeleteItem(ItemType);
        void RetrieveItem(ItemType&, bool&);
        void ResetList();
        void GetNextItem(ItemType&);
    private:
        int length;
        ItemType info[MAX_ITEMS];
        int currentPos;
};
#endif // UNSORTEDTYPE_H_INCLUDED
```

sortedtype.h

```
#ifndef SORTEDTYPE_H_INCLUDED
#define SORTEDTYPE_H_INCLUDED

const int MAX_ITEMS = 5;

template <class ItemType>
class SortedType
{
    public :
        SortedType();
        void MakeEmpty();
        bool IsFull();
        int LengthIs();
        void InsertItem(ItemType);
        void DeleteItem(ItemType);
        void RetrieveItem(ItemType&, bool&);
        void ResetList();
        void GetNextItem(ItemType&);
    private:
        int length;
        ItemType info[MAX_ITEMS];
        int currentPos;
};
#endif // SORTEDTYPE_H_INCLUDED
```

unsortedtype.cpp

```
#include “unsortedType.h"

template <class ItemType>

UnsortedType<ItemType>::UnsortedType()

{

    length = 0;

    currentPos = -1;

}

template <class ItemType>

void UnsortedType<ItemType>:: MakeEmpty()

{

    length = 0;

}

template <class ItemType>

bool UnsortedType<ItemType>:: IsFull()

{

    return (length == MAX_ITEMS);

}
```

```
template <class ItemType>

int UnsortedType<ItemType>::LengthIs()

{

    return length;

}

template <class ItemType>

void UnsortedType<ItemType>::ResetList()

{

    currentPos = -1;

}

template <class ItemType>

void UnsortedType<ItemType>::
GetNextItem(ItemType& item)

{

    currentPos++;

    item = info [currentPos] ;

}
```

unsortedtype.cpp

```
#include “unsortedType.h"
template <class ItemType>
UnsortedType<ItemType>::UnsortedType()
{
    length = 0;
    currentPos = -1;
}
```

O(1)

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

O(1)

```
template <class ItemType>
bool UnsortedType<ItemType>::IsFull()
{
    return (length == MAX_ITEMS);
}
```

O(1)

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

O(1)

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

O(1)

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

O(1)

sortedtype.cpp

```
#include "sortedtype.h"
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_ITEMS);
}
```

```
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];
}
```

sortedtype.cpp

```
#include "sortedtype.h"
template <class ItemType>
SortedType<ItemType>::SortedType()
{
    length = 0;
    currentPos = -1;
}
```

O(1)

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

O(1)

```
template <class ItemType>
bool SortedType<ItemType>::IsFull()
{
    return (length == MAX_ITEMS);
}
```

O(1)

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

O(1)

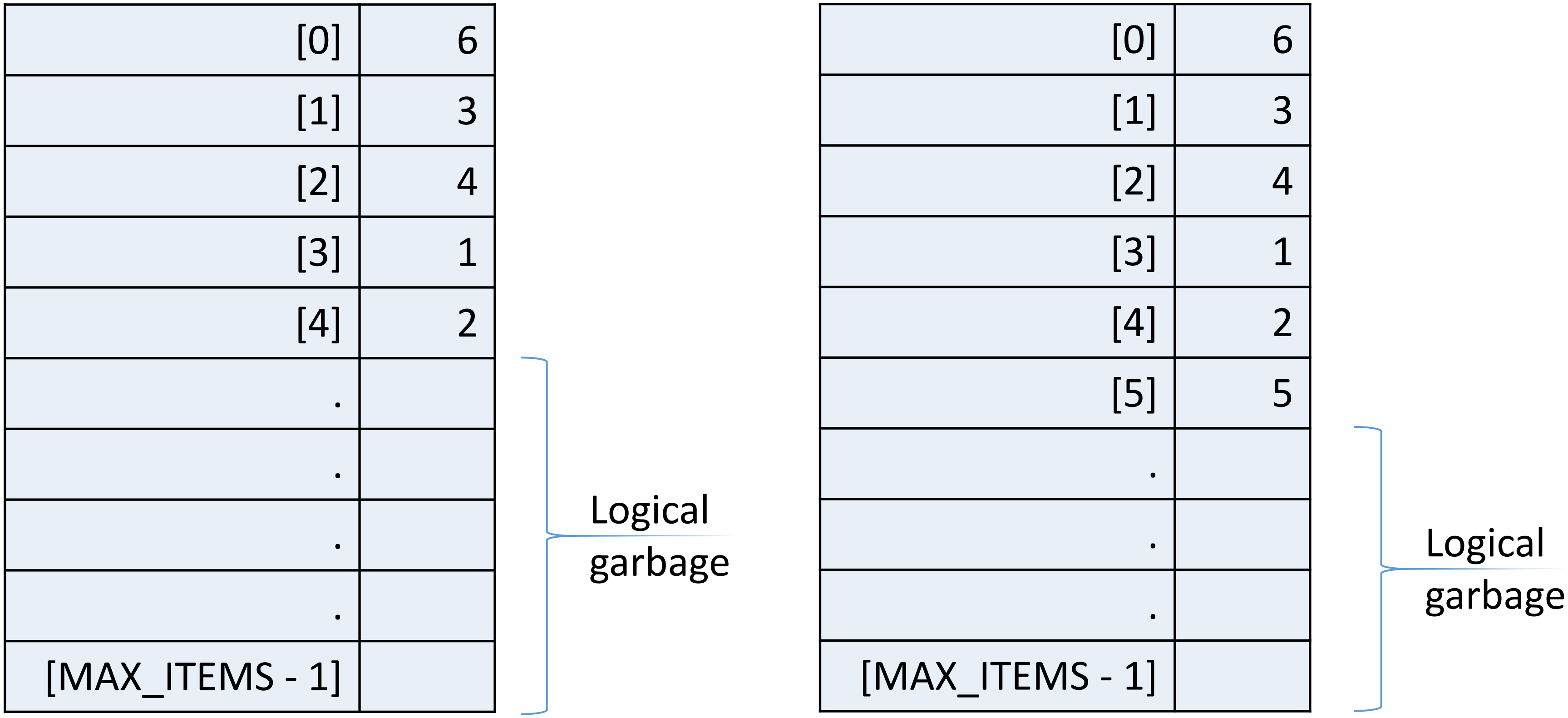
```
template <class ItemType>
void SortedType<ItemType>::ResetList()
{
    currentPos = -1;
}
```

O(1)

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

O(1)

Inserting an Item into Unsorted List



[0]	6
[1]	3
[2]	4
[3]	1
[4]	2
.	
.	
.	
.	
[MAX_ITEMS - 1]	

length = 5

[0]	6
[1]	3
[2]	4
[3]	1
[4]	2
[5]	5
.	
.	
.	
[MAX_ITEMS - 1]	

length = 6

Insert 5

unsortedtype.cpp

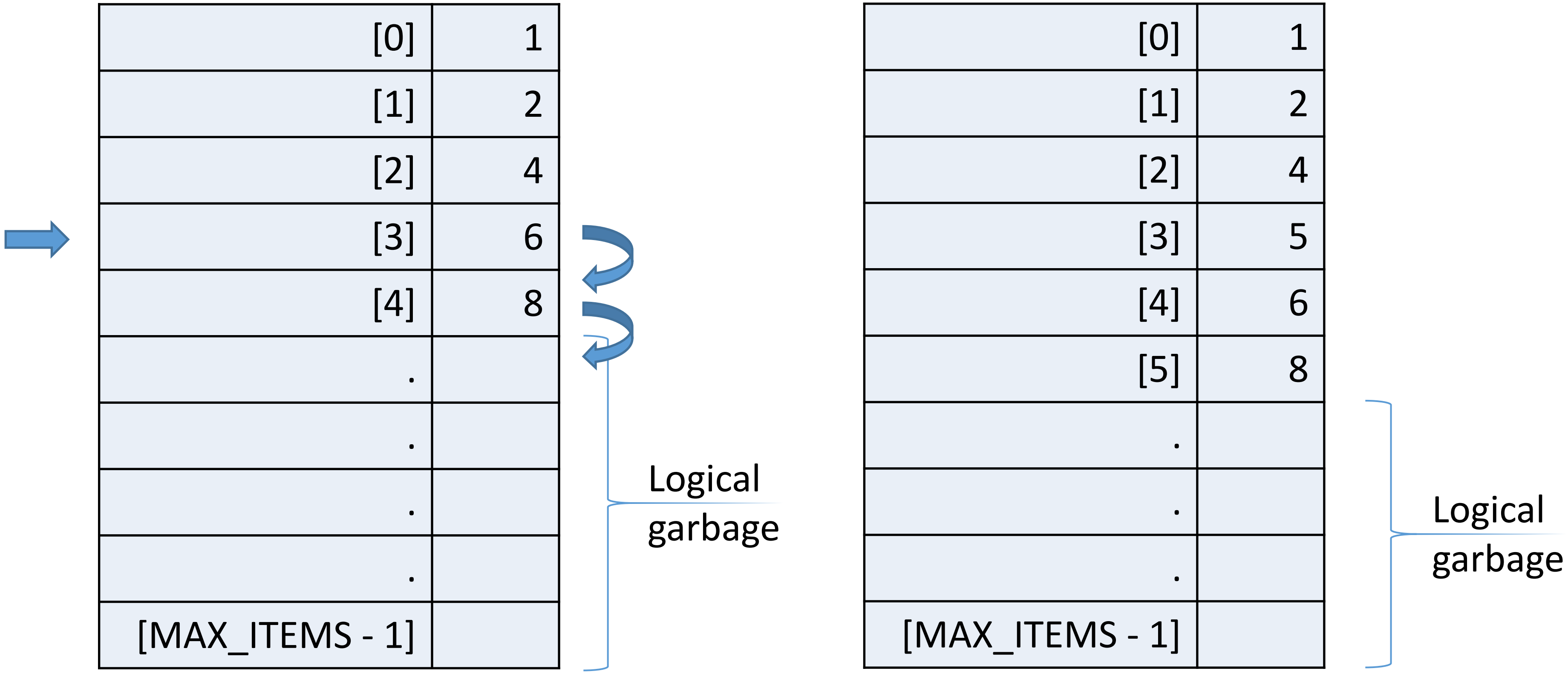
```
template <class ItemType>
void UnsortedType<ItemType>::InsertItem(ItemType item)
{
    info[length] = item;
    length++;
}
```

unsortedtype.cpp

```
template <class ItemType>
void UnsortedType<ItemType>::InsertItem(ItemType item)
{
    info[length] = item;
    length++;
}
```

O(1)

Inserting an Item into Sorted List



[0]	1
[1]	2
[2]	4
[3]	6
[4]	8
.	
.	
.	
.	
[MAX_ITEMS - 1]	

Logical garbage

length = 5

[0]	1
[1]	2
[2]	4
[3]	5
[4]	6
[5]	8
.	
.	
.	
[MAX_ITEMS - 1]	

Logical garbage

length = 6

Insert 5

sortedtype.cpp

```
template <class ItemType>
void SortedType<ItemType>::InsertItem(ItemType item)
{
    int location = 0;
    //find the location to insert the item
    while (location < length)
    {
         if(item > info[location])
       location++;
    else
            break;
    }
    //shift all elements at indexes >= location one cell right
    for (int index = length-1; index >= location; index--)
        info[index+1] = info[index];
    //insert item at index location
    info[location] = item;
    //update length
    length++;
}
```

sortedtype.cpp

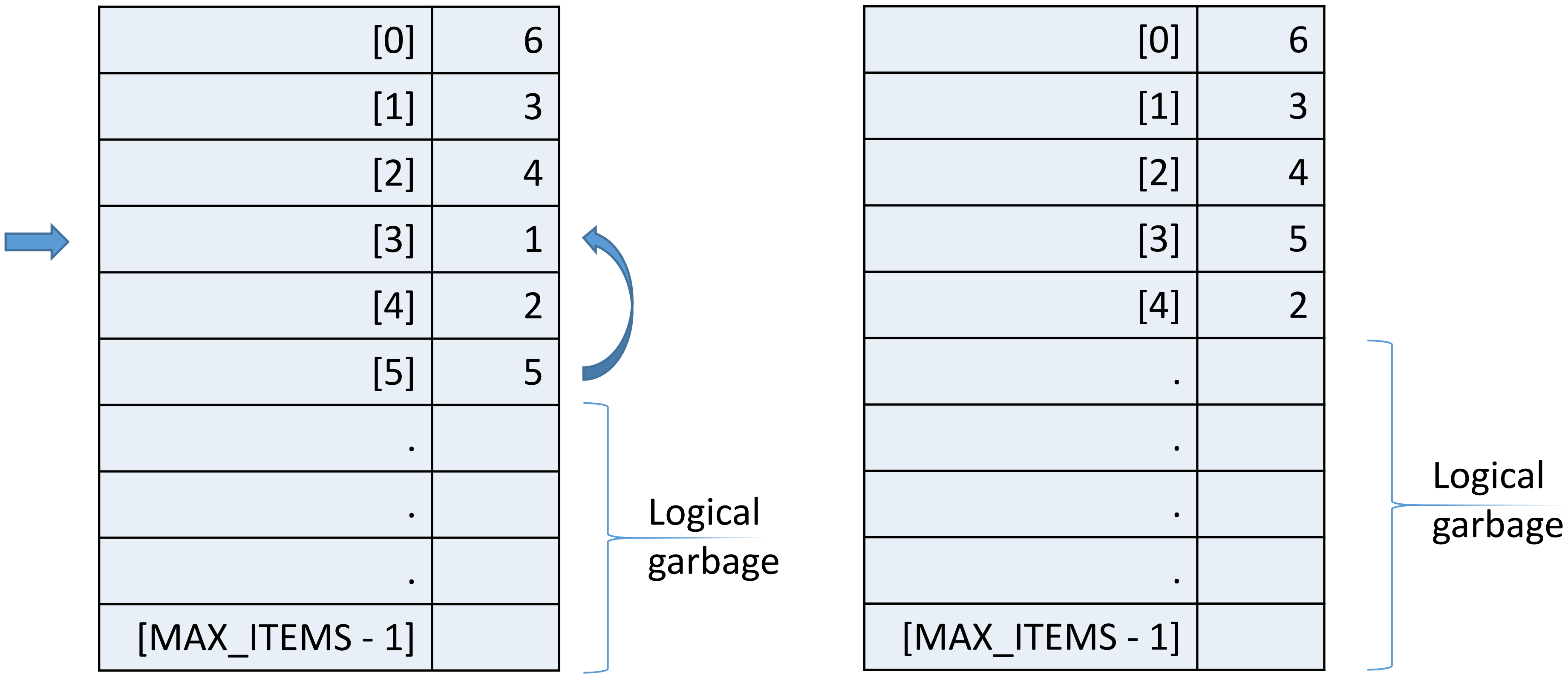
```
template <class ItemType>
void SortedType<ItemType>::InsertItem(ItemType item)
{
    int location = 0;
    //find the location to insert the item
    while (location < length)
    {
         if(item > info[location])
       location++;
    else
            break;
    }
    //shift all elements at indexes >= location one cell right
    for (int index = length-1; index >= location; index--)
        info[index+1] = info[index];
    //insert item at index location
    info[location] = item;
    //update length
    length++;
}
```

O(N)

O(N)

O(N)

Deleting an Item from Unsorted List



[0]	6
[1]	3
[2]	4
[3]	1
[4]	2
[5]	5
.	
.	
.	
[MAX_ITEMS - 1]	

Logical garbage

length = 6

[0]	6
[1]	3
[2]	4
[3]	5
[4]	2
.	
.	
.	
.	
[MAX_ITEMS - 1]	

Logical garbage

length = 5

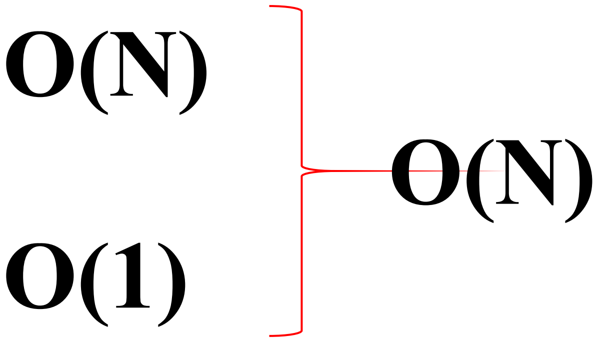
Delete 1

unsortedtype.cpp

```
template <class ItemType>
void UnsortedType<ItemType>::DeleteItem(ItemType item)
{
    int location = 0;
    while (item != info[location])
        location++;
    info[location] = info[length - 1];
    length--;
}
```

unsortedtype.cpp

```
template <class ItemType>
void UnsortedType<ItemType>::DeleteItem(ItemType item)
{
    int location = 0;
    while (item != info[location])
        location++;
    info[location] = info[length - 1];
    length--;
}
```

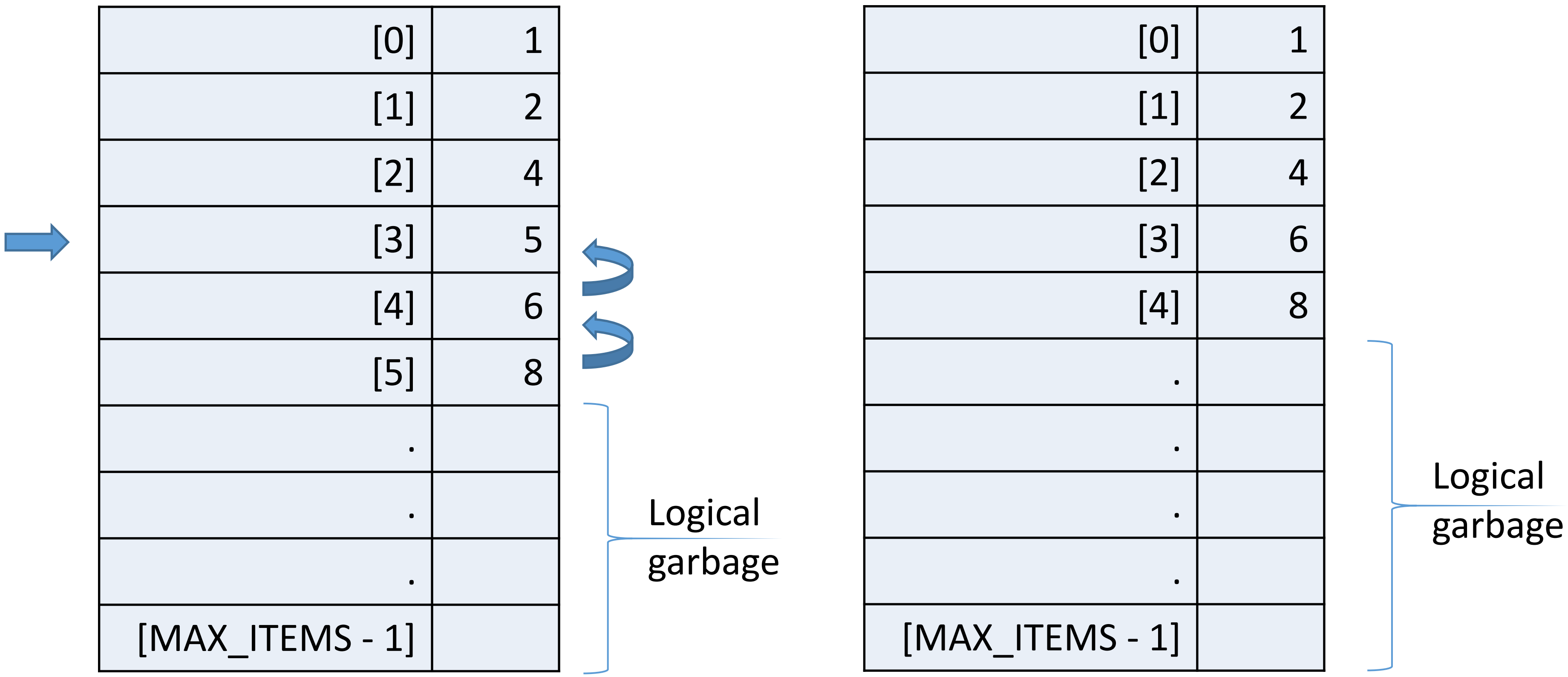


O(N)

O(1)

O(N)

Deleting an Item from Sorted List



[0]	1
[1]	2
[2]	4
[3]	5
[4]	6
[5]	8
.	
.	
.	
[MAX_ITEMS - 1]	

Logical garbage

length = 6

[0]	1
[1]	2
[2]	4
[3]	6
[4]	8
.	
.	
.	
.	
[MAX_ITEMS - 1]	

Logical garbage

length = 5

Delete 5

sortedtype.cpp

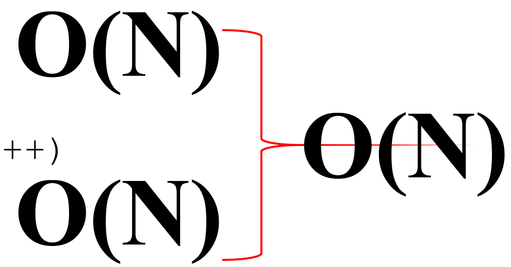
```
template <class ItemType>
void SortedType<ItemType>::DeleteItem(ItemType item)
{
    int location = 0;

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

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)

Sorting Algorithm

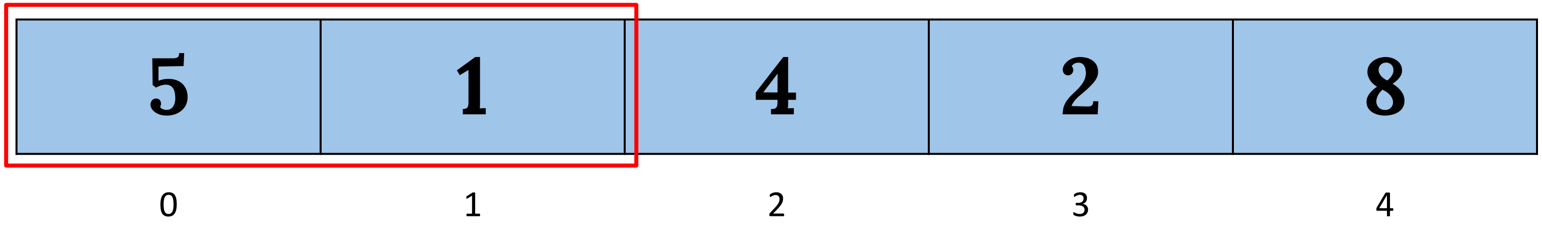
- **Bubble Sort**

  Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order.

Bubble Sort

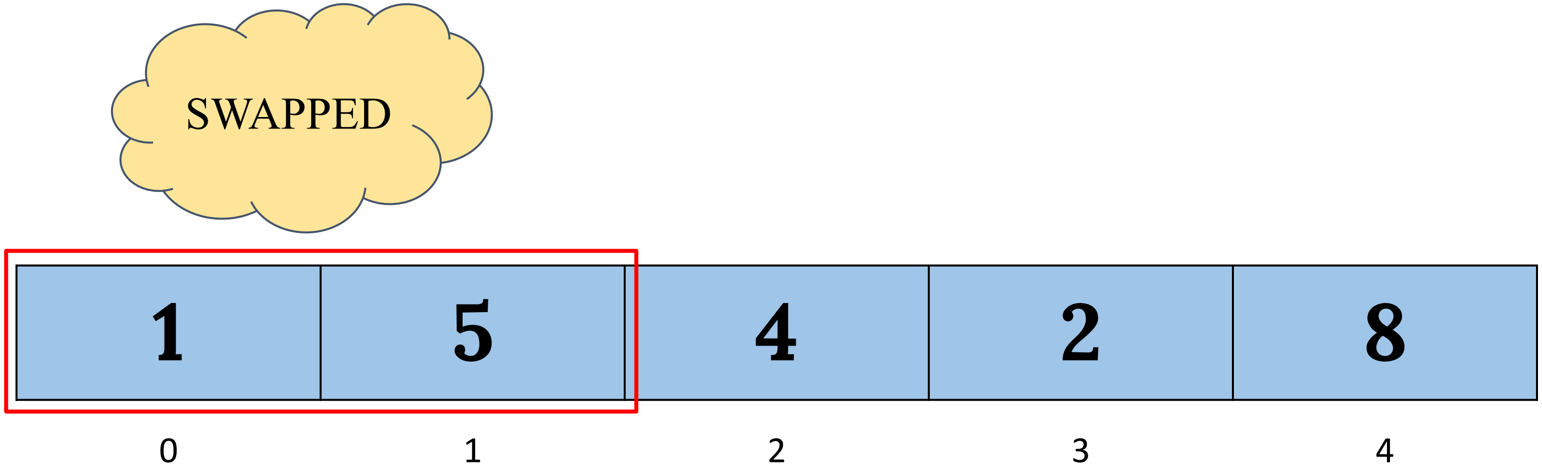
5	**1**	**4**	**2**	**8**
0	1	2	3	4

Bubble Sort



5	**1**	**4**	**2**	**8**
0	1	2	3	4

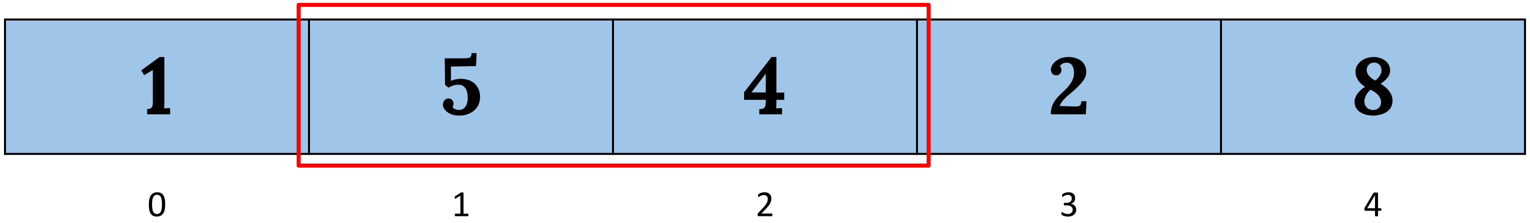
Bubble Sort



SWAPPED

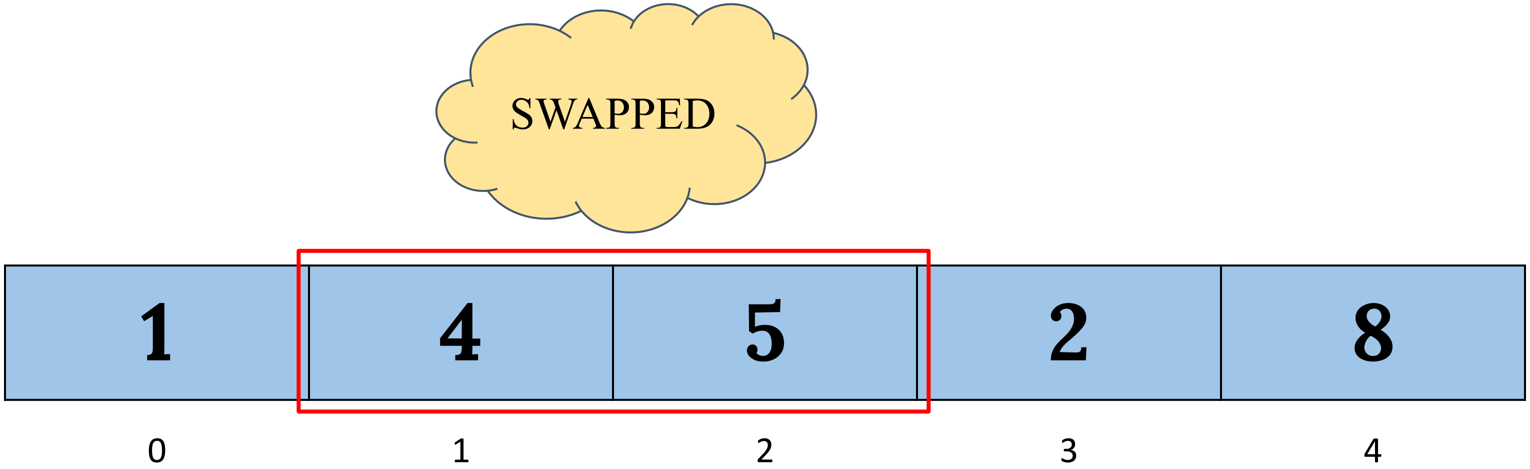
1	5	4	2	8
0	1	2	3	4

Bubble Sort



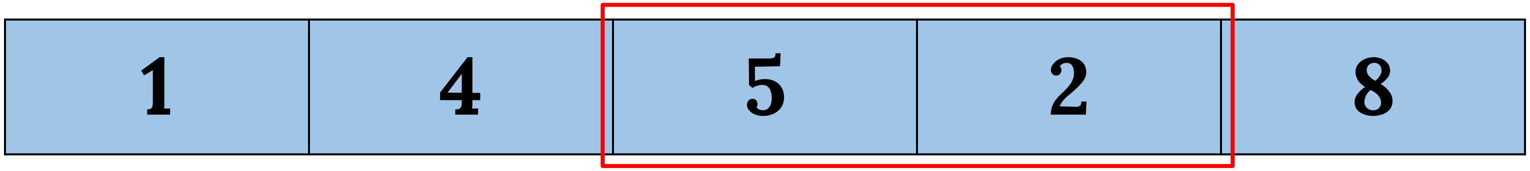
1	5	4	2	8
0	1	2	3	4

Bubble Sort



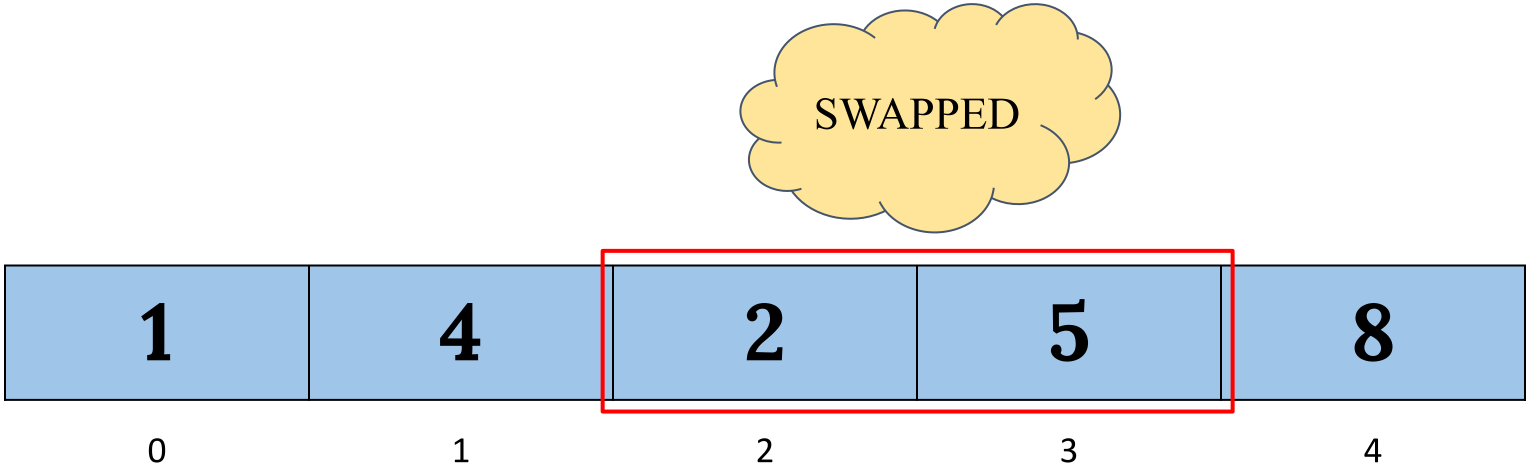
1	**4**	**5**	**2**	**8**
0	1	2	3	4

Bubble Sort



1	4	5	2	8
0	1	2	3	4

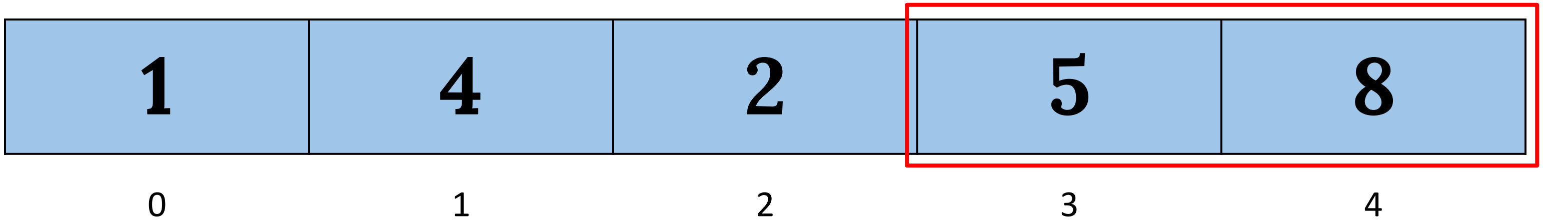
Bubble Sort



SWAPPED

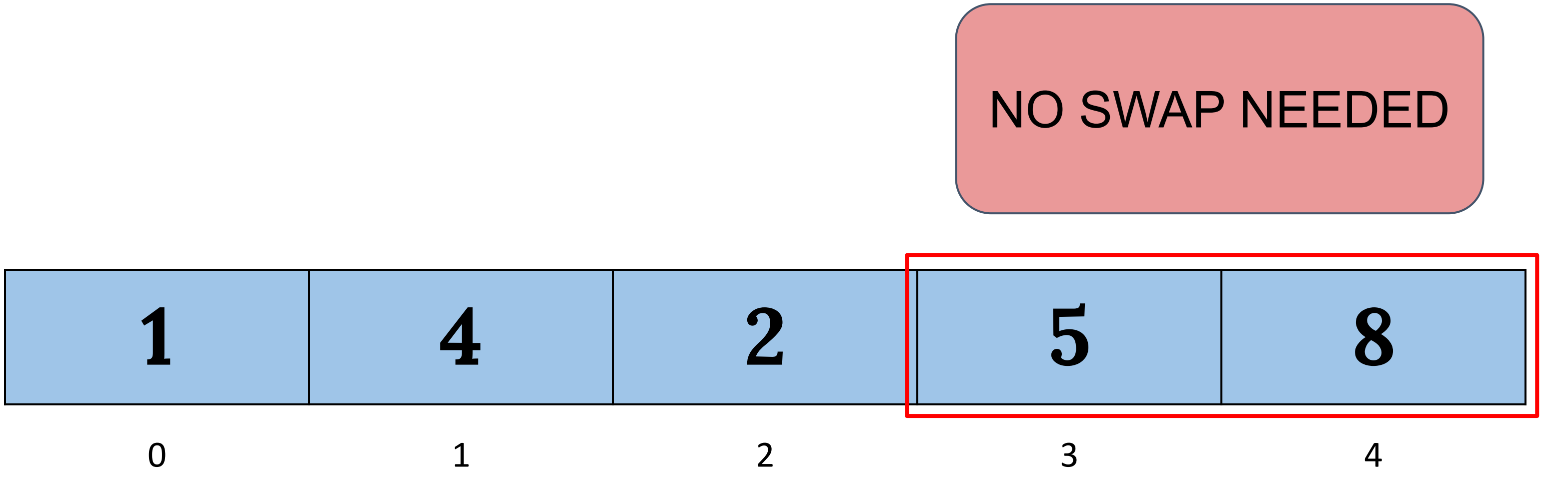
1	**4**	**2**	**5**	**8**
0	1	2	3	4

Bubble Sort



1	4	2	5	8
0	1	2	3	4

Bubble Sort



NO SWAP NEEDED

1	**4**	**2**	**5**	**8**
0	1	2	3	4

Bubble Sort

First Pass

5	**1**	**4**	**2**	**8**
0	1	2	3	4

1	**5**	**4**	**2**	**8**
0	1	2	3	4

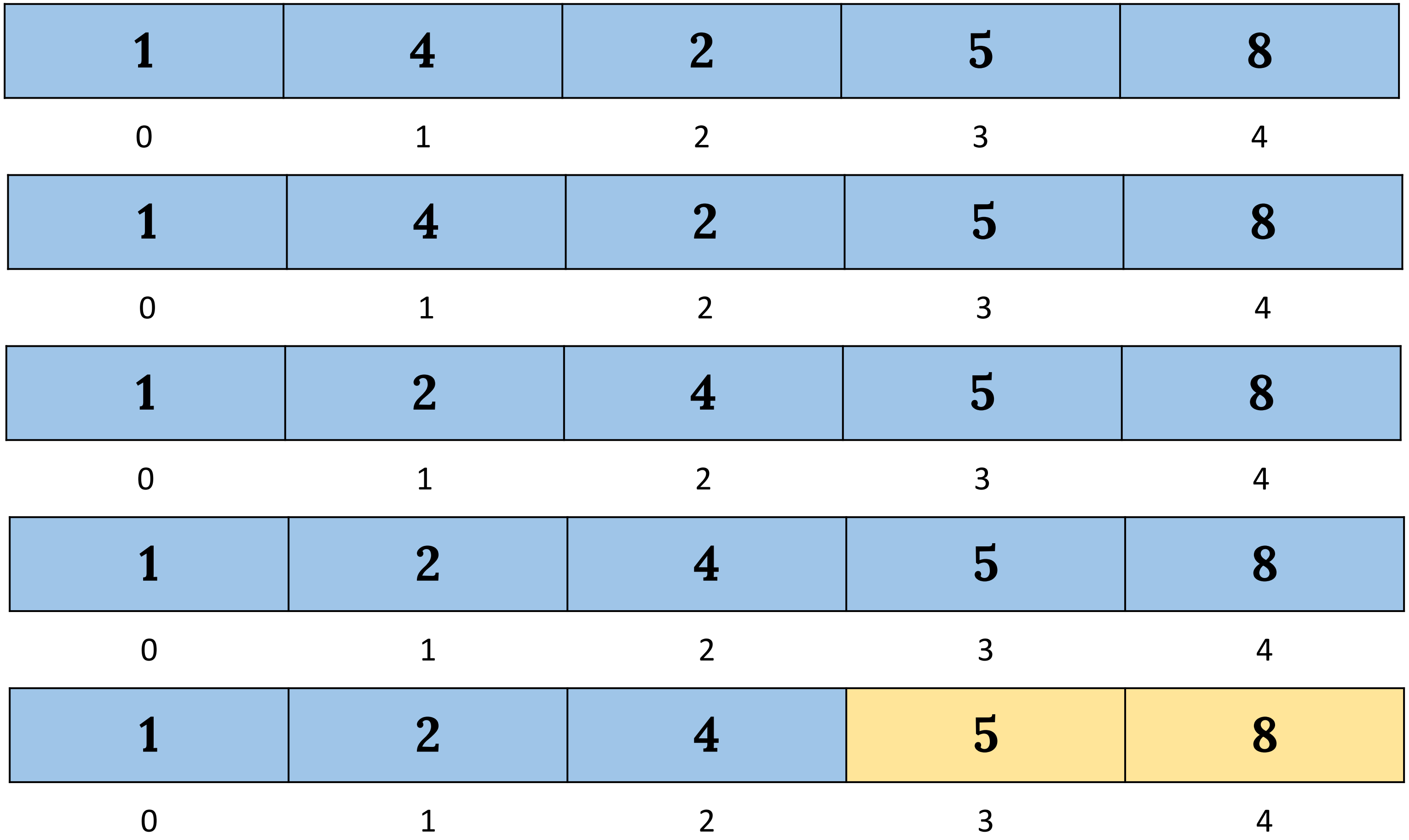
1	**4**	**5**	**2**	**8**
0	1	2	3	4

1	**4**	**2**	**5**	**8**
0	1	2	3	4

1	**4**	**2**	**5**	**8**
0	1	2	3	4

Bubble Sort

Second Pass



1	4	2	5	8
0	1	2	3	4

1	4	2	5	8
0	1	2	3	4

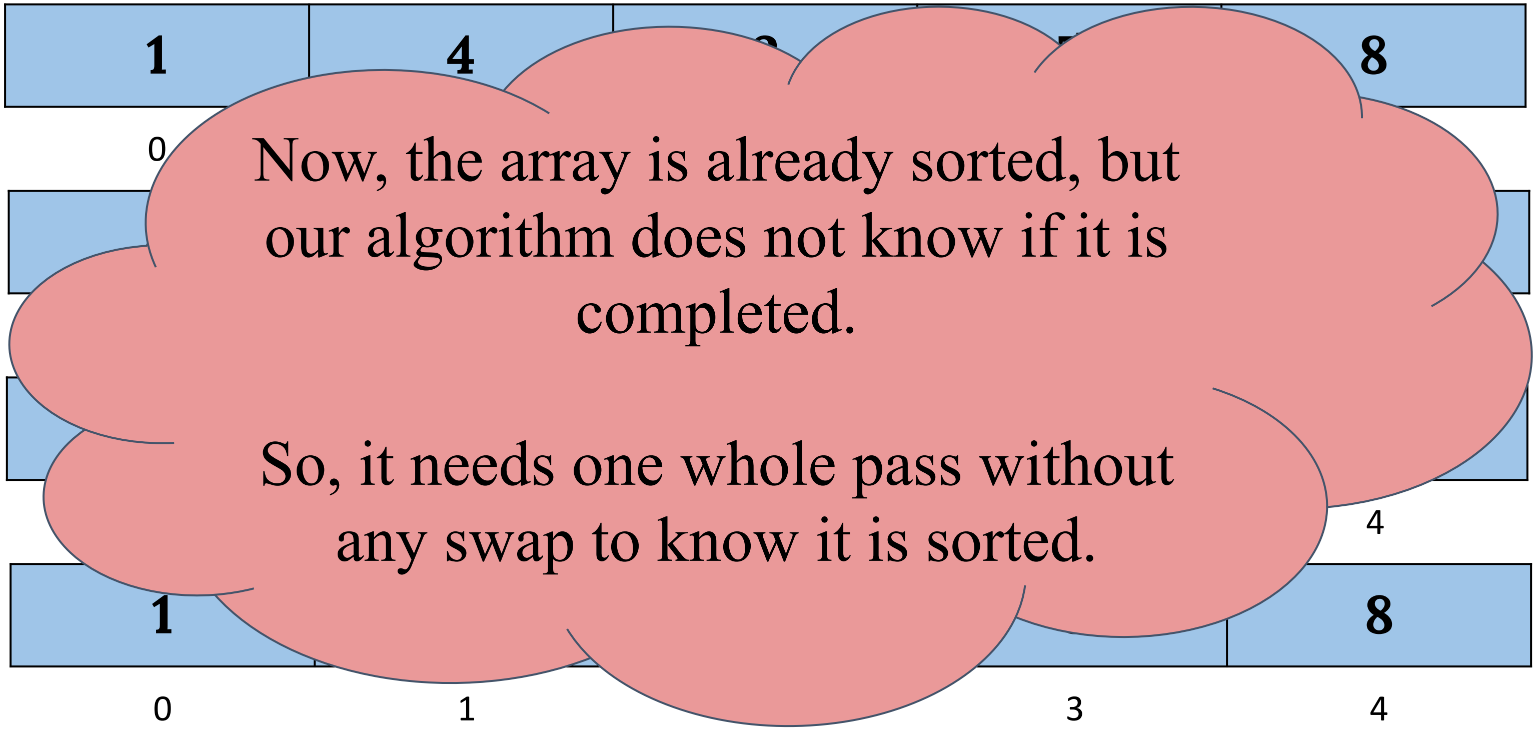
1	2	4	5	8
0	1	2	3	4

1	2	4	5	8
0	1	2	3	4

1	2	4	5	8
0	1	2	3	4

Bubble Sort

Second Pass



Now, the array is already sorted, but our algorithm does not know if it is completed.

So, it needs one whole pass without any swap to know it is sorted.

1	2	4	5	8
0	1	2	3	4

Bubble Sort

Third Pass

1	2	4	5	8
0	1	2	3	4

1	2	4	5	8
0	1	2	3	4

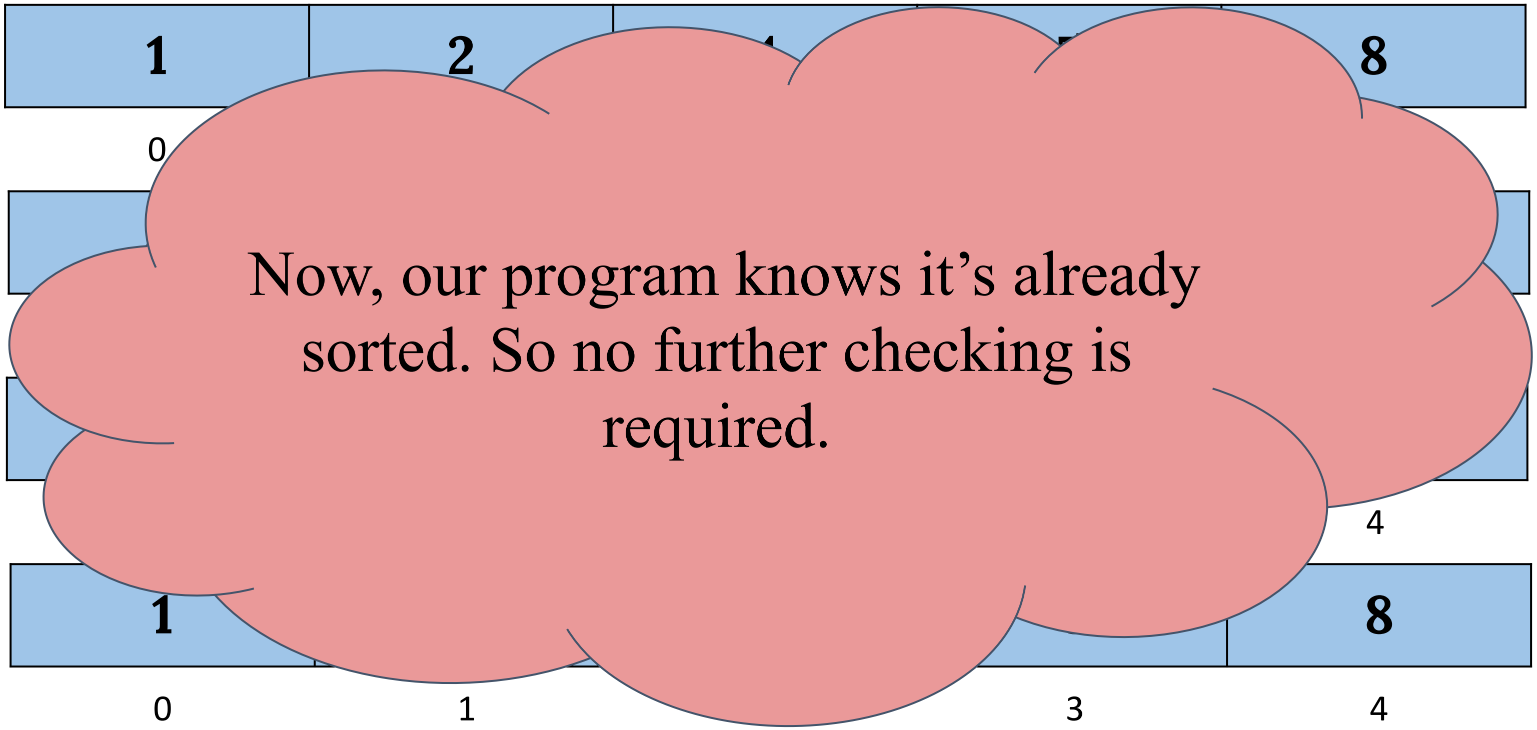
1	2	4	5	8
0	1	2	3	4

1	2	4	5	8
0	1	2	3	4

1	2	4	5	8
0	1	2	3	4

Bubble Sort

Third Pass



Now, our program knows it's already sorted. So no further checking is required.

1	**2**	**4**	**5**	**8**
0	1	2	3	4

Bubble Sort

```
void bubbleSort(int array[], int size)
{
    for (int step = 0; step < size; ++step)
    {
         for (int i = 0; i < size - step; ++i)
         {
              if (array[i] > array[i + 1]) {
                   int temp = array[i];
                   array[i] = array[i + 1];
                   array[i + 1] = temp;
               }
          }
     }
}
```

Bubble Sort

```
void bubbleSort(int array[], int size)
{
    for (int step = 0; step < size; ++step)
    {
        for (int i = 0; i < size - step; ++i)
        {
            if (array[i] > array[i + 1]) {
                int temp = array[i];
                array[i] = array[i + 1];
                array[i + 1] = temp;
            }
        }
    }
}
```

$O(N^2)$

Retrieving an Item from Unsorted List

- Visit each element in the list, one by one, until the item is found.

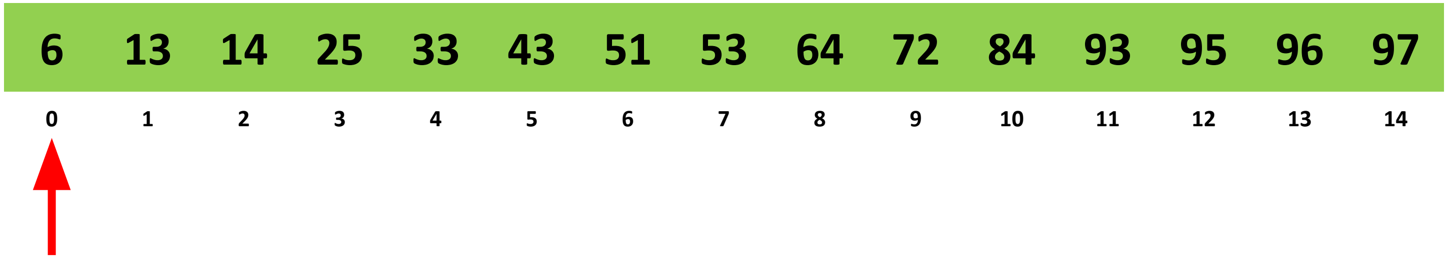
Retrieving an Item from Unsorted List

- Find **51**

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 Unsorted List

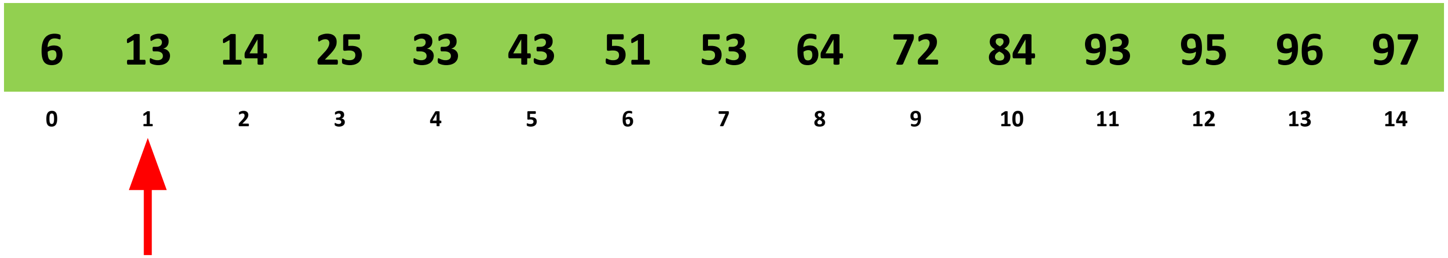
- Find **51**



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 Unsorted List

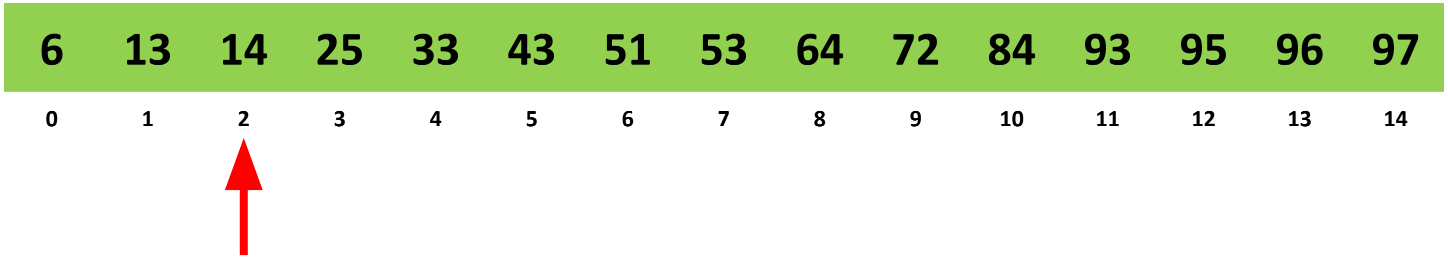
- Find **51**



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 Unsorted List

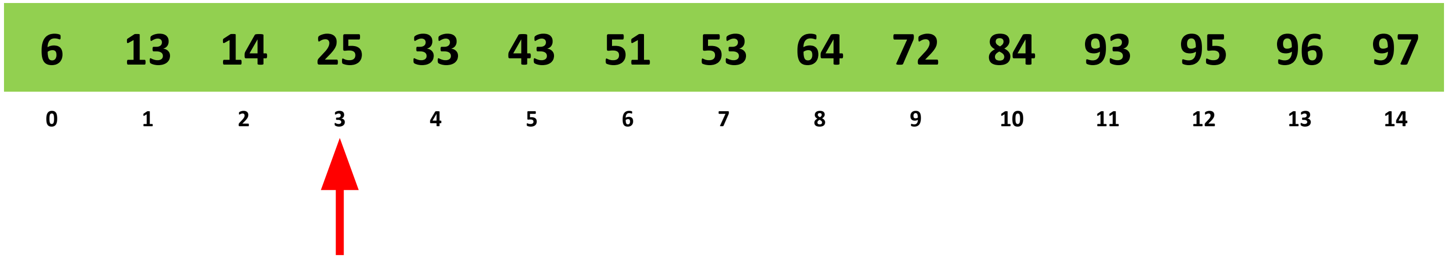
- Find **51**



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 Unsorted List

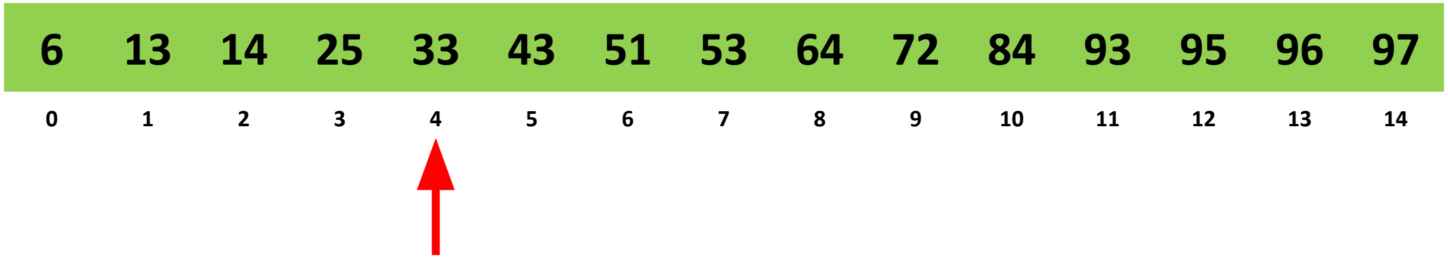
- Find **51**



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 Unsorted List

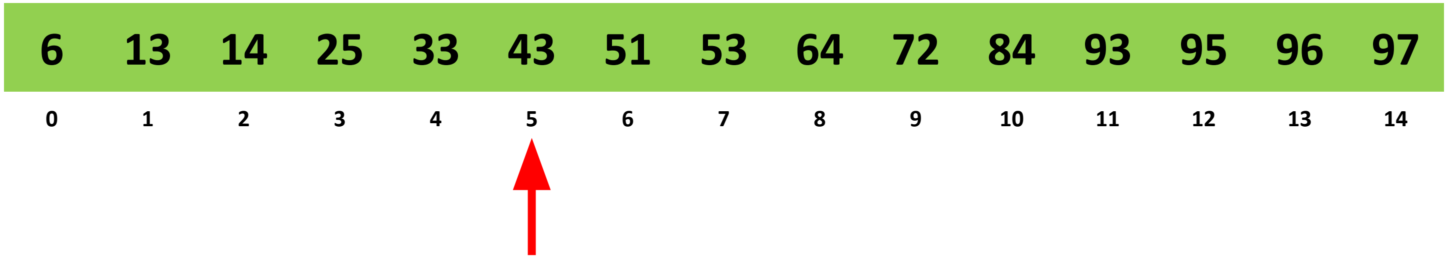
- Find **51**



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 Unsorted List

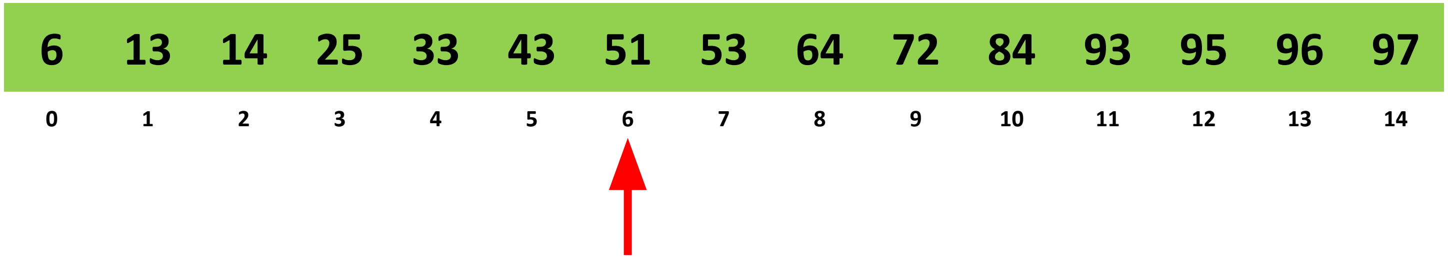
- Find **51**



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 Unsorted List

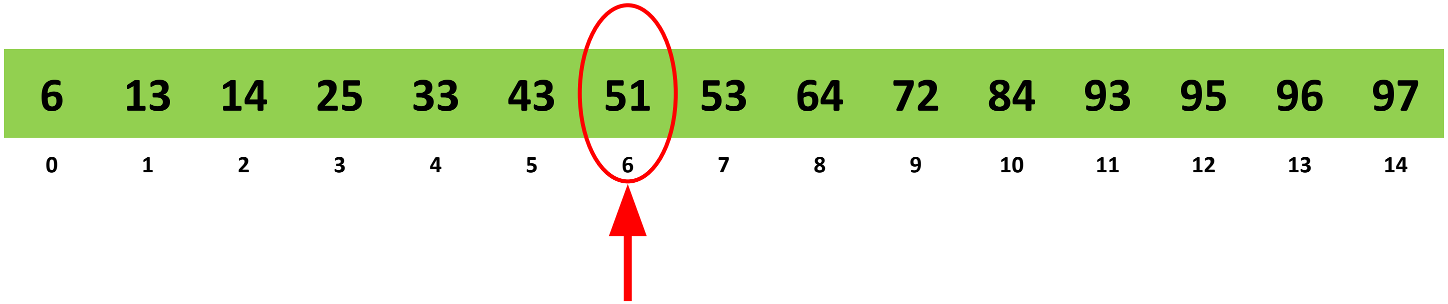
- Find **51**



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 Unsorted List

- Find **51**



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**

Found 51 at index position 6

unsortedtype.cpp

```
template <class ItemType>
void UnsortedType<ItemType>::RetrieveItem(ItemType& item, bool &found)
{
    int location = 0;
    bool moreToSearch = (location < length);
    found = false;
    while (moreToSearch && !found)
    {
        if(item == info[location])
        {
            found = true;
            item = info[location];
        }
        else
        {
            location++;
            moreToSearch = (location < length);
        }
    }
}
```

unsortedtype.cpp

```
template <class ItemType>
void UnsortedType<ItemType>::RetrieveItem(ItemType& item, bool &found)
{
    int location = 0;
    bool moreToSearch = (location < length);
    found = false;
    while (moreToSearch && !found)
    {
        if(item == info[location])
        {
            found = true;
            item = info[location];
        }
        else
        {
            location++;
            moreToSearch = (location < length);
        }
    }
}
```

O(N)

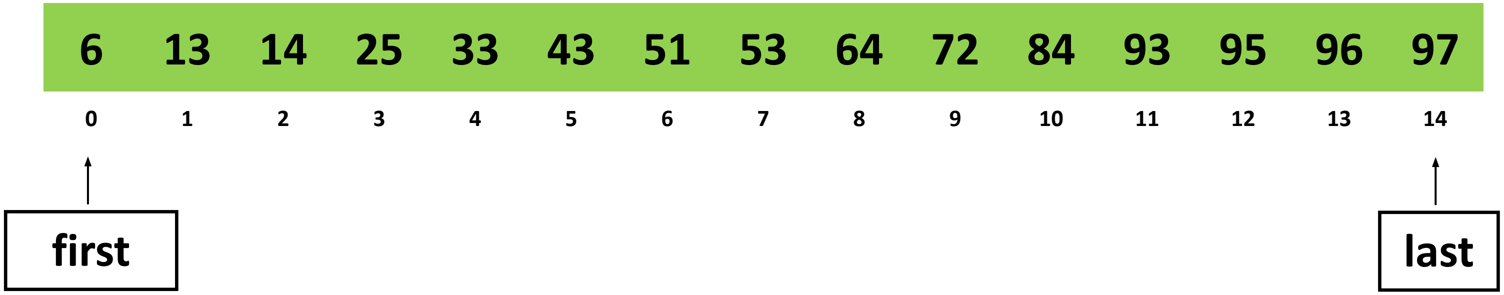
Retrieving an Item from Sorted List

- 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**

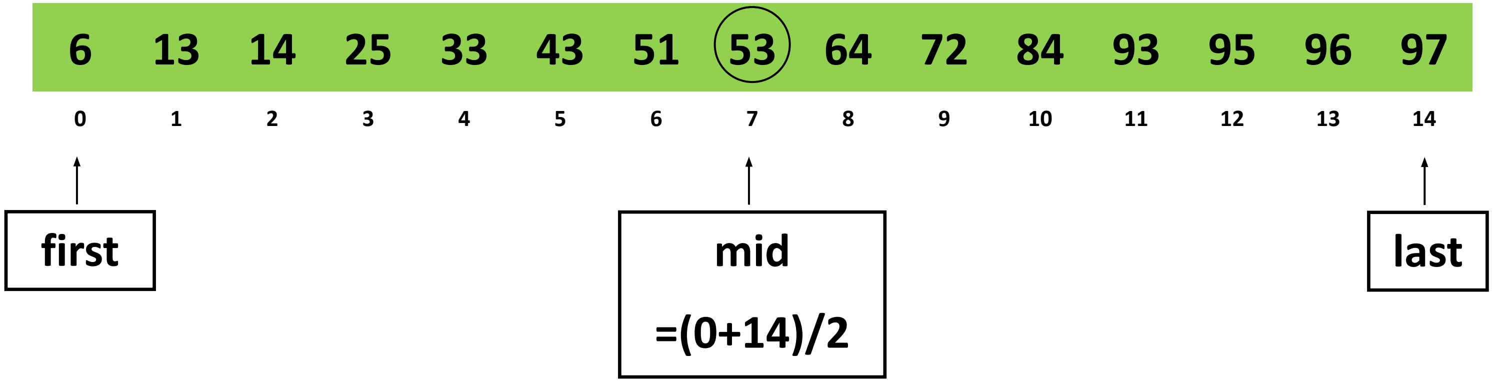


6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

- **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

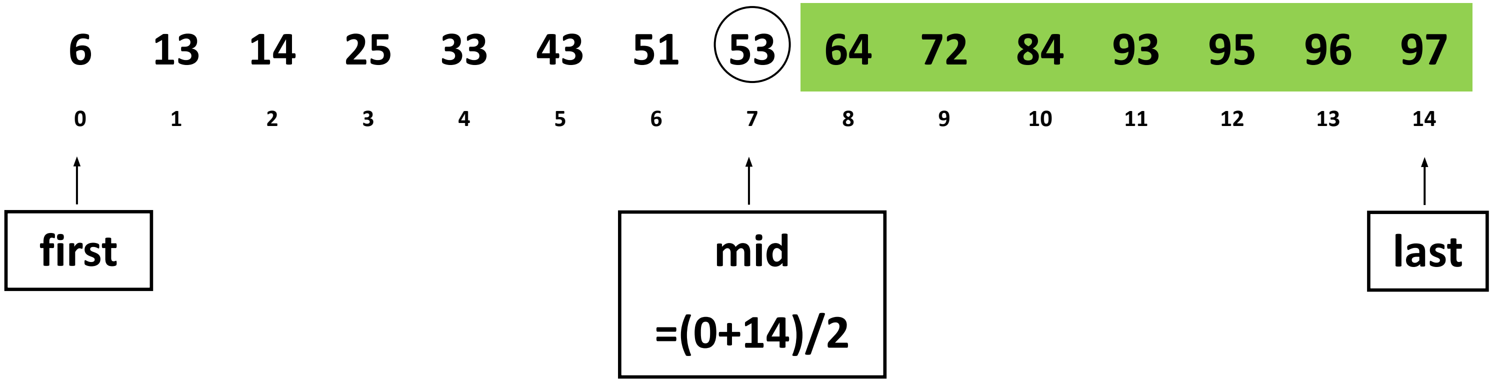
=(0+14)/2

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

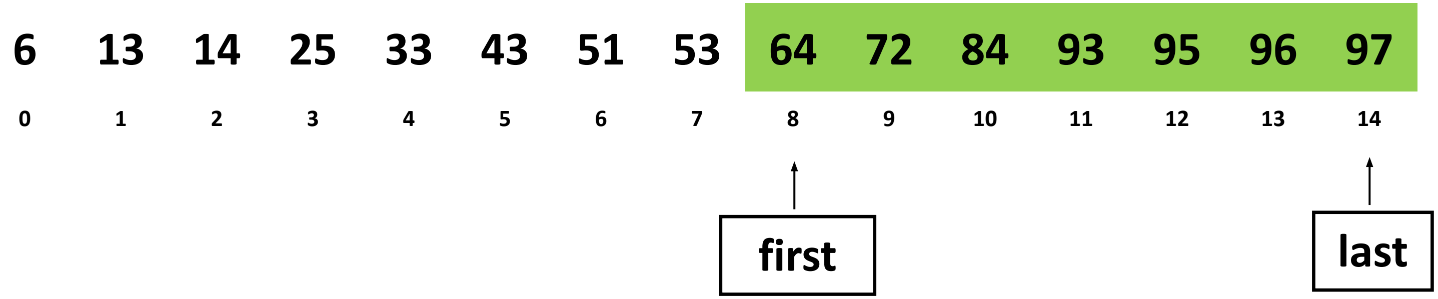
=(0+14)/2

last

- **Step 1**

Retrieving an Item from Sorted List

- Find **84**

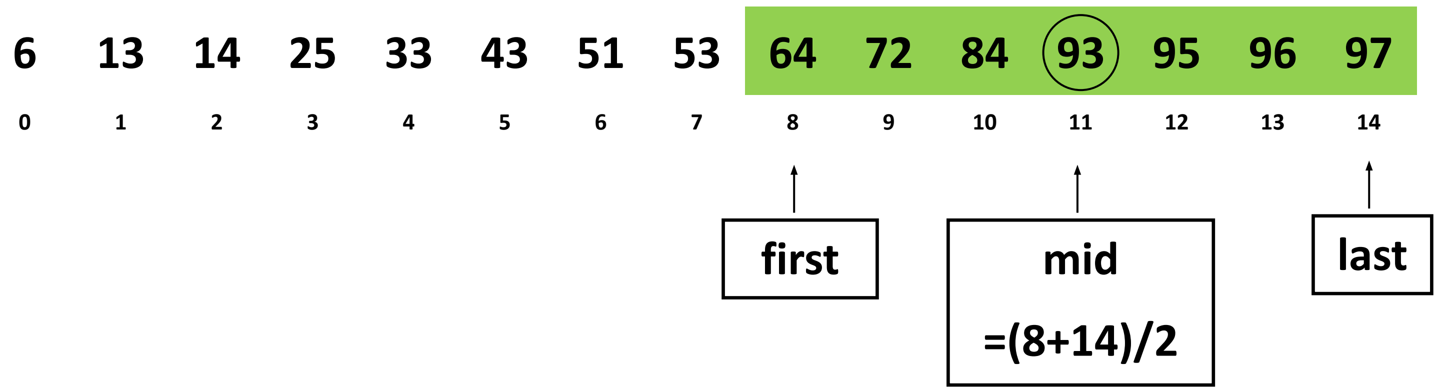


6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

- **Step 2**

Retrieving an Item from Sorted List

- Find **84**

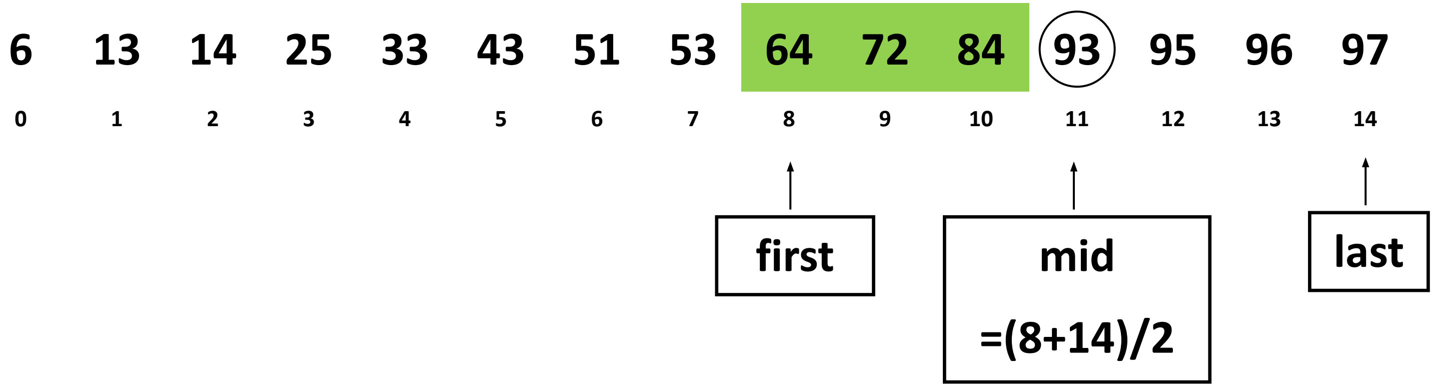


6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

- **Step 2**

Retrieving an Item from Sorted List

- Find **84**

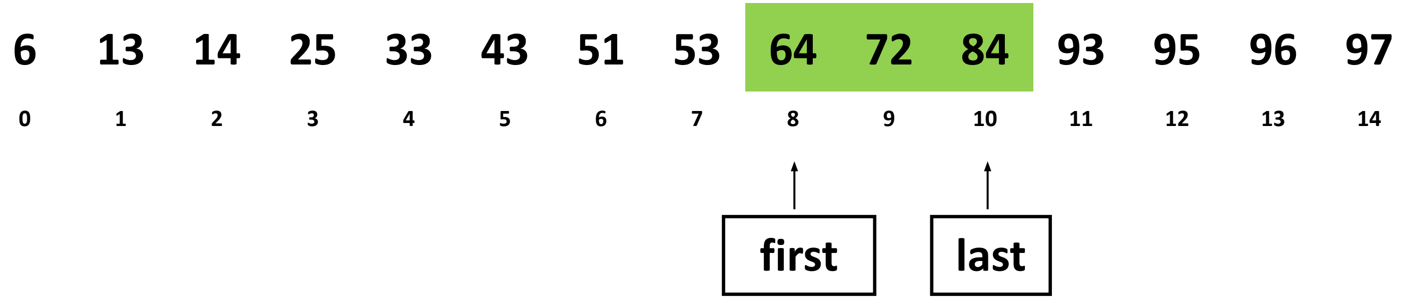


6	**13**	**14**	**25**	**33**	**43**	**51**	**53**	**64**	**72**	**84**	**93**	**95**	**96**	**97**
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

- **Step 2**

Retrieving an Item from Sorted List

- Find **84**

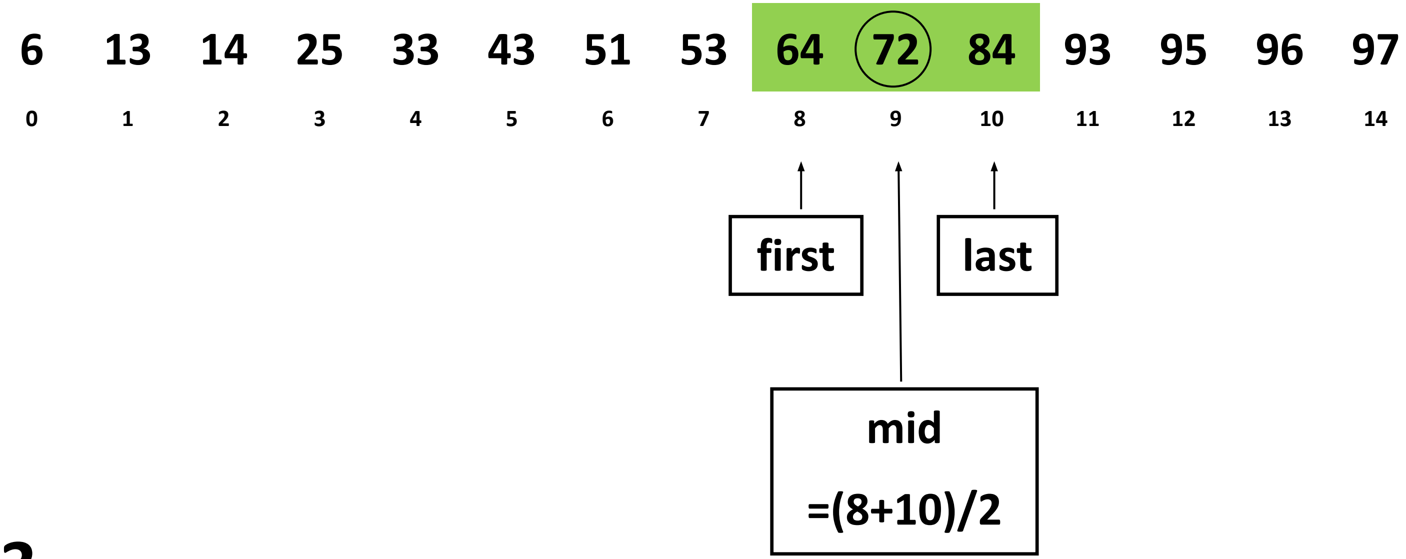


6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

- **Step 3**

Retrieving an Item from Sorted List

- Find **84**



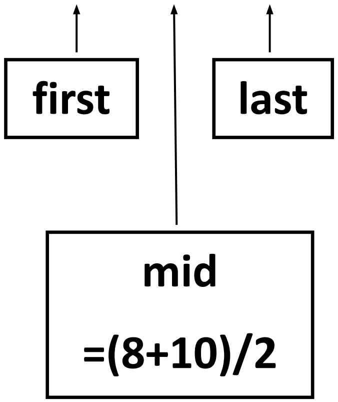
6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

- **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

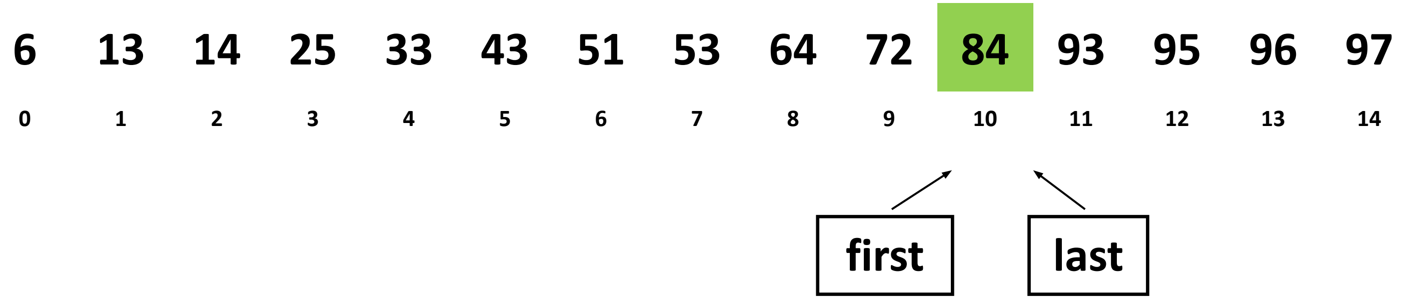
mid

=(8+10)/2

- **Step 3**

Retrieving an Item from Sorted List

- Find **84**



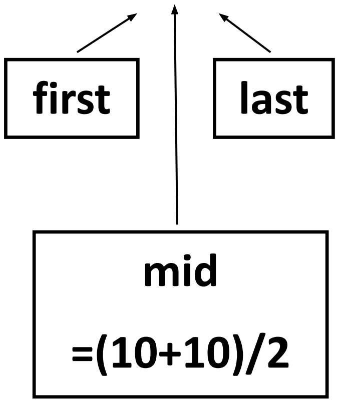
6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

- **Step 4**

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**



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

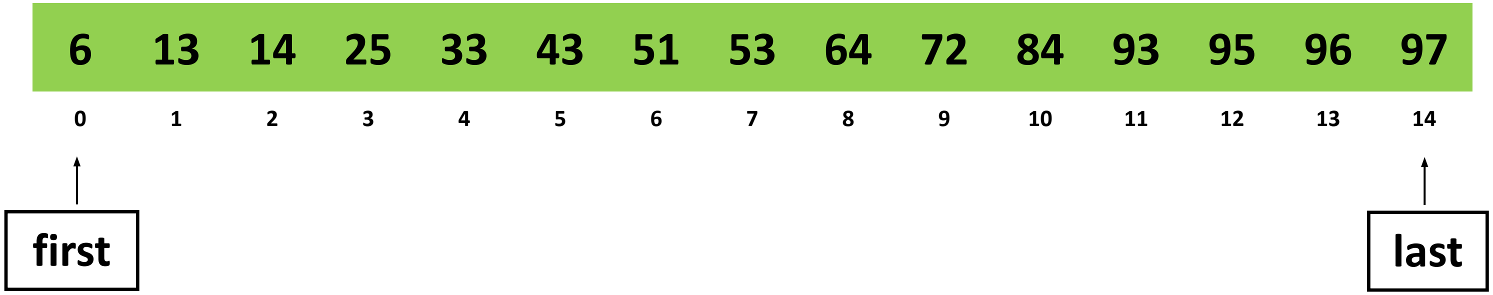
Retrieving an Item from Sorted List

- 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**

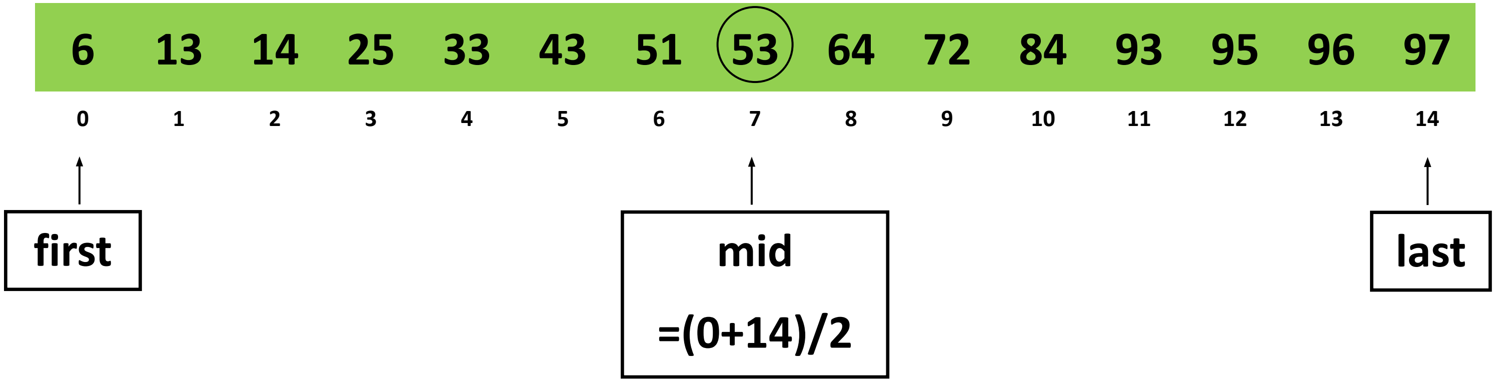


6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

- **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

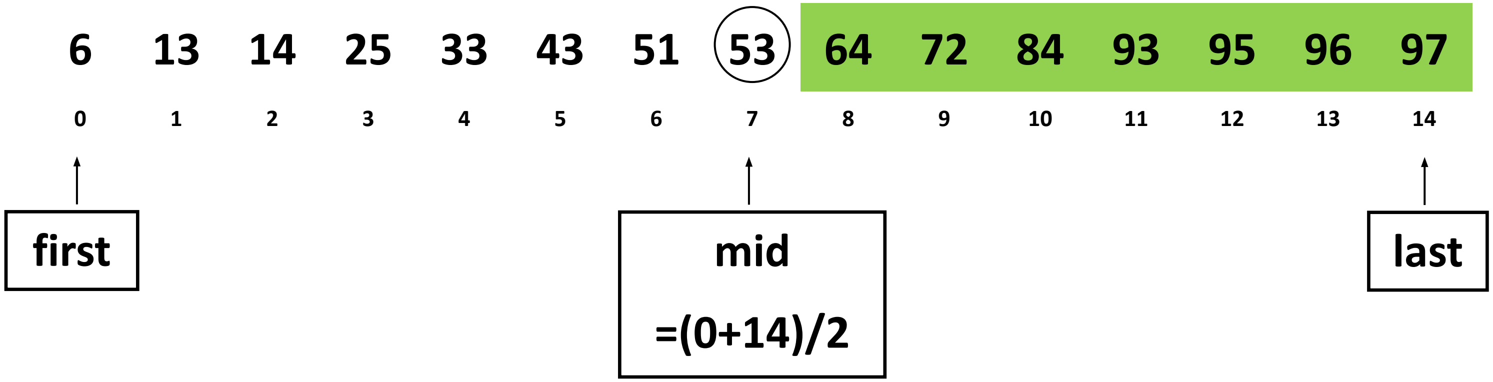
=(0+14)/2

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

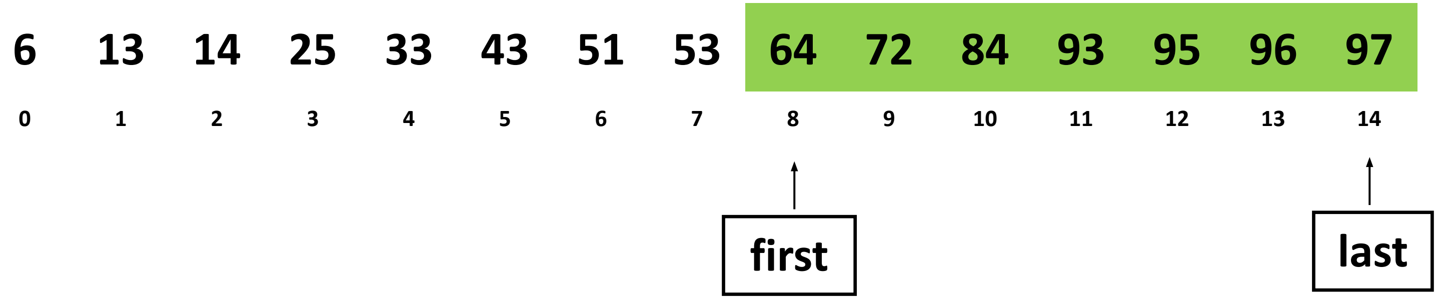
=(0+14)/2

last

- **Step 1**

Retrieving an Item from Sorted List

- Find **73**



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**

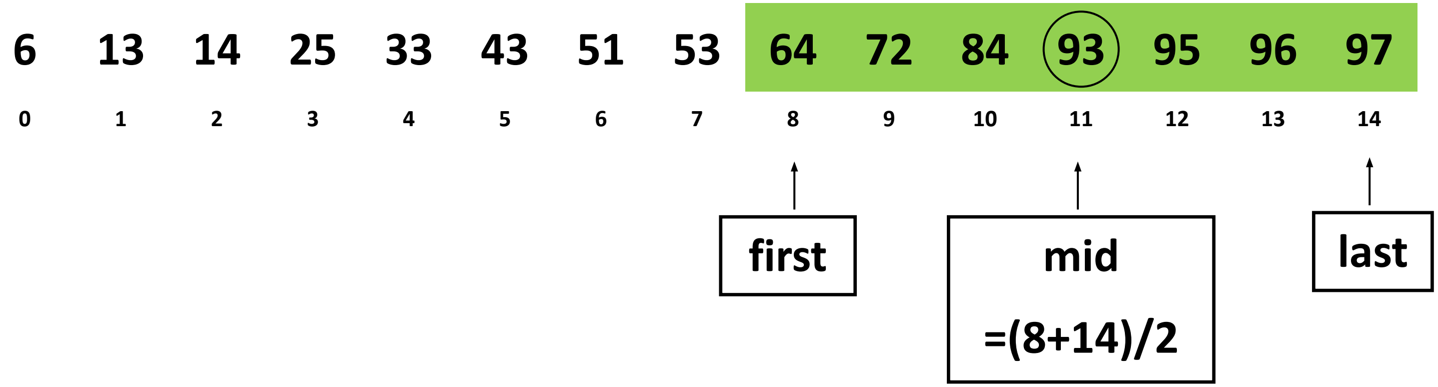
first

last

- **Step 2**

Retrieving an Item from Sorted List

- Find **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

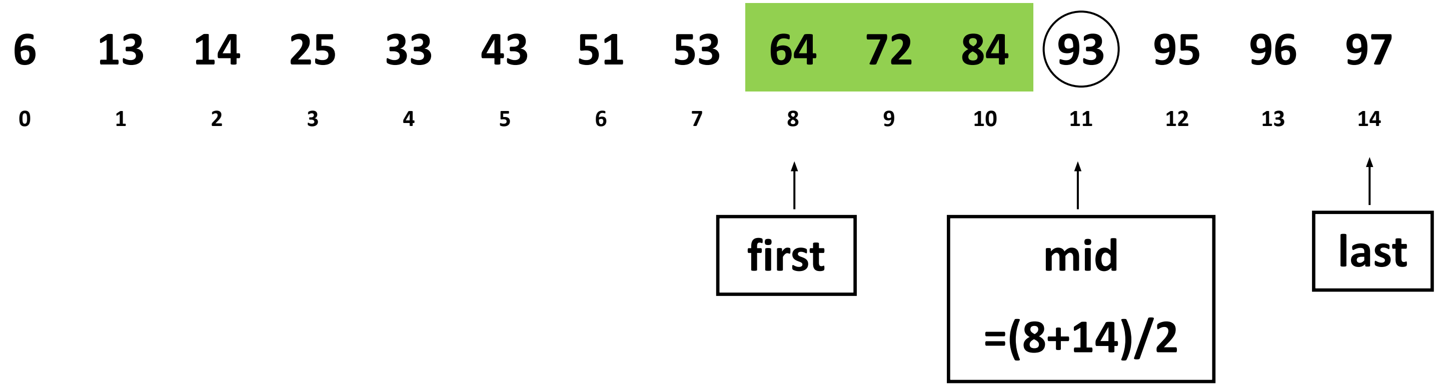
=(8+14)/2

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

mid

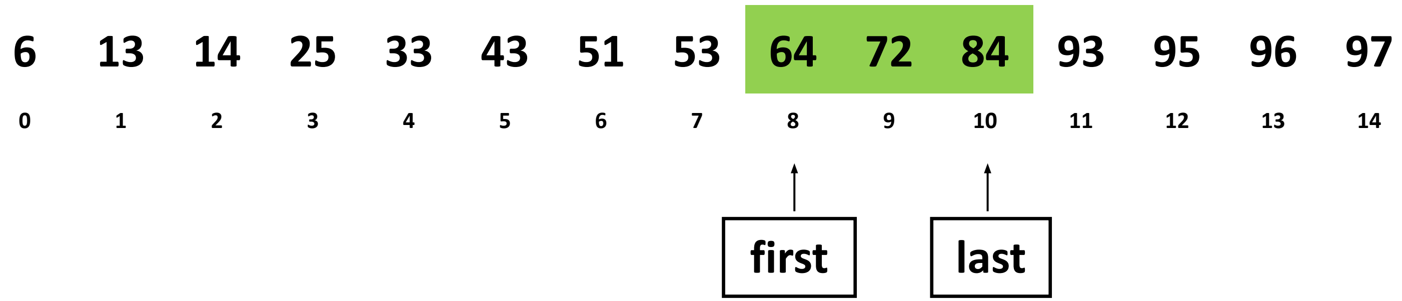
=(8+14)/2

last

- **Step 2**

Retrieving an Item from Sorted List

- Find **73**

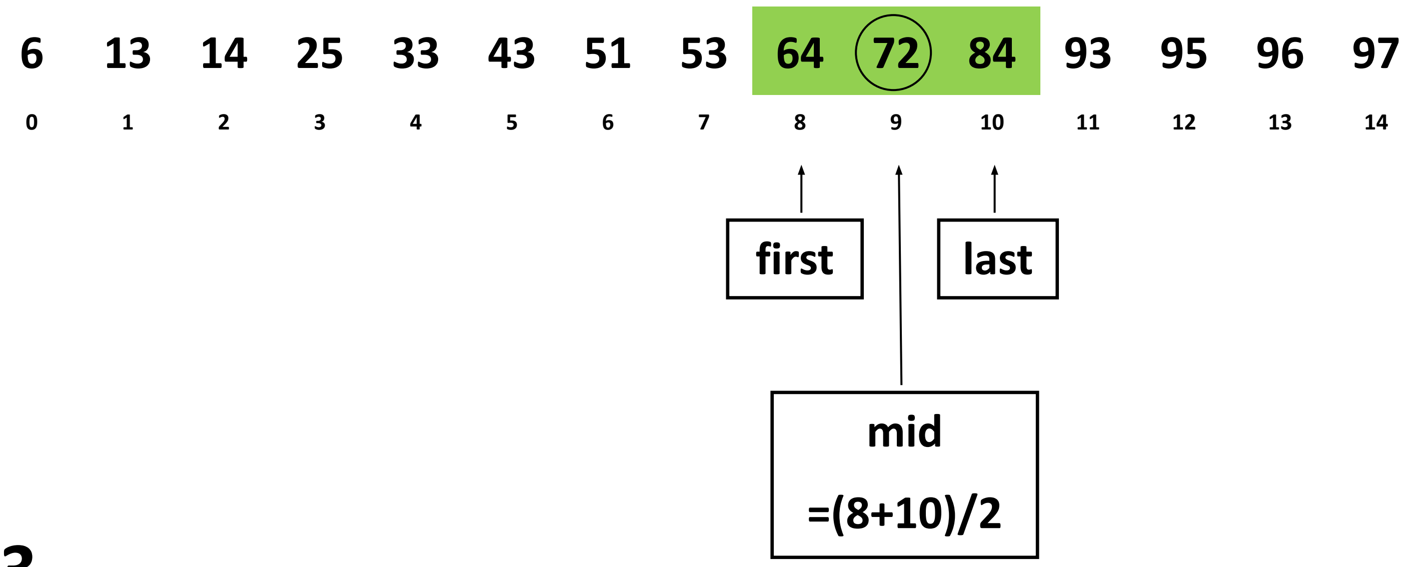


6	**13**	**14**	**25**	**33**	**43**	**51**	**53**	**64**	**72**	**84**	**93**	**95**	**96**	**97**
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

- **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

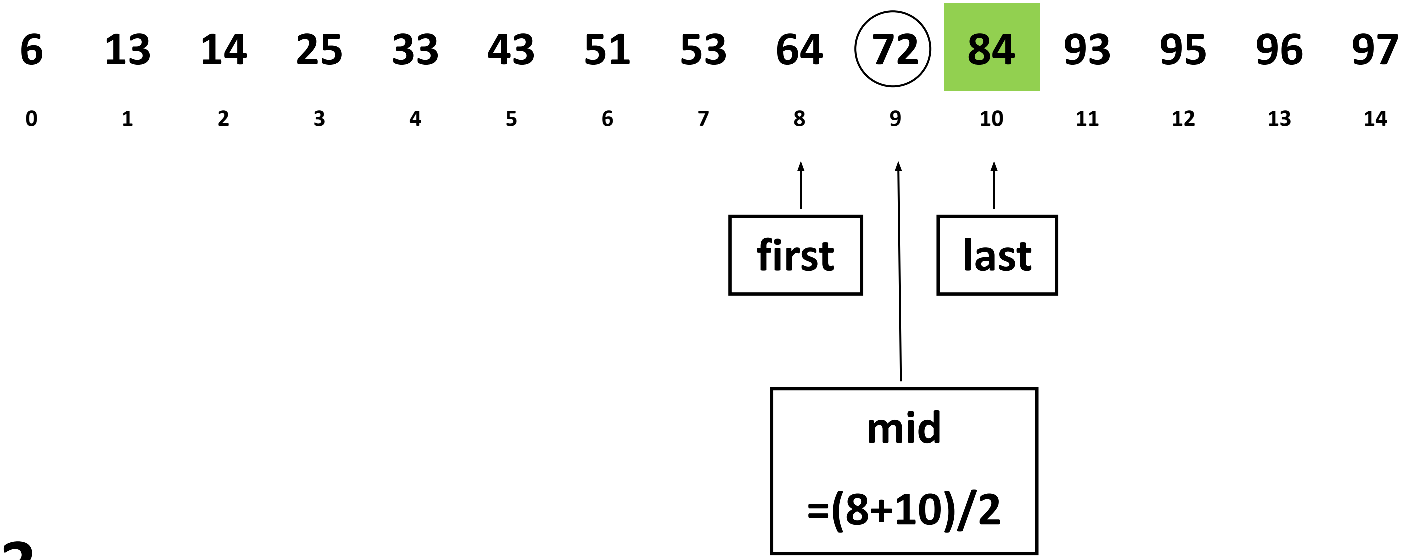
mid

=(8+10)/2

- **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

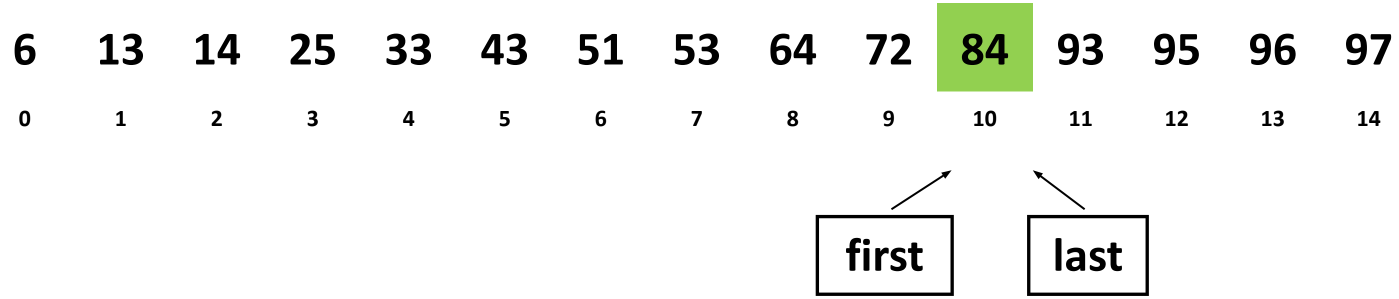
mid

=(8+10)/2

- **Step 3**

Retrieving an Item from Sorted List

- Find **73**

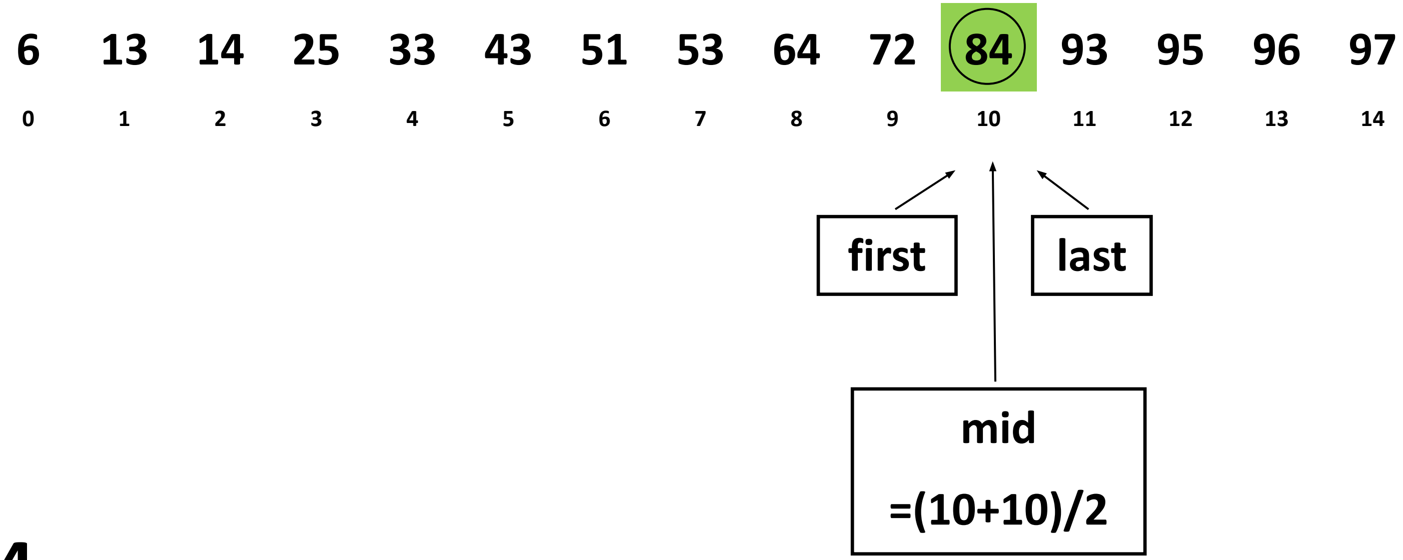


6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

- **Step 4**

Retrieving an Item from Sorted List

- Find **73**

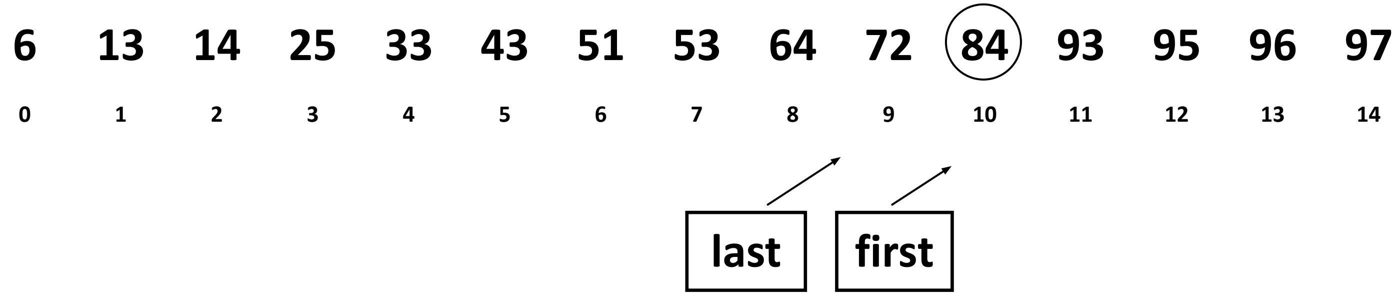


6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

- **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

- **Step 5**
- **Item != info[mid] (indicates the absence of the item)**

Retrieving an Item from Sorted List

- What is the number of steps required?
- How many times can you divide N by 2 until you have 1?

Array size	expressed as 2^a
N	$2^{(x)}$
N/2	$2^{(x-1)}$
N/4	$2^{(x-2)}$
N/8	$2^{(x-3)}$
.	.
.	.
.	.
.	.
4	2^2
2	2^1
1	2^0

Retrieving an Item from Sorted List

- What is the number of steps required?

Array size	expressed as 2^a
N	$2^{(x)}$
N/2	$2^{(x-1)}$
N/4	$2^{(x-2)}$
N/8	$2^{(x-3)}$
.	.
.	.
.	.
.	.
4	2^2
2	2^1
1	2^0

$$2^x \sim N$$

Or,

$$x = \log_2 N$$

Or simply,

$$x = \lg N$$

sortedtype.cpp

```
template <class ItemType>
void SortedType<ItemType>::RetrieveItem(ItemType& item, bool& found)
{
    int mid, first = 0, last = length - 1;
    while (first <= last)
    {
        mid = first + (last-first)/2;

        // Check if item is present at mid
        if (info[mid] == item){
            found = true;
      return;
        }

        // If item greater, ignore left half
        if (info[mid] < item)
            first = mid + 1;

        // If item is smaller, ignore right half
        else last = mid - 1;
    }

    // if we reach here, then element was
    // not present
    found = false;

}//end function
```

sortedtype.cpp

```
template <class ItemType>
void SortedType<ItemType>::RetrieveItem(ItemType& item, bool& found)
{
    int mid, first = 0, last = length - 1;
    while (first <= last)
    {
        mid = first + (last-first)/2;

        // Check if item is present at mid
        if (info[mid] == item){
            found = true;
      return;
        }

        // If item greater, ignore left half
        if (info[mid] < item)
            first = mid + 1;

        // If item is smaller, ignore right half
        else last = mid - 1;
    }

    // if we reach here, then element was
    // not present
    found = false;

}//end function
```

O(lg N)

RetrieveItem (recursive)

```
template <class ItemType>
void SortedType<ItemType>::RetrieveItemRec(ItemType& item, bool& found, int l,
int r)
{
   if (r >= l)
   {
        int mid = l + (r - l)/2;

        // If the element is present at the middle
        // itself
        if (info[mid] == item) found = true;

        // If element is smaller than mid, then
        // it can only be present in left subarray
        if (arr[mid] > x)
            return RetrieveItemRec (item, found, l, mid-1);

        // Else the element can only be present
        // in right subarray
        return RetrieveItemRec (item, found, mid+1, r);
   }

   // We reach here when element is not
   // present in array
   found = false;

}//end function
```

O(lg N)