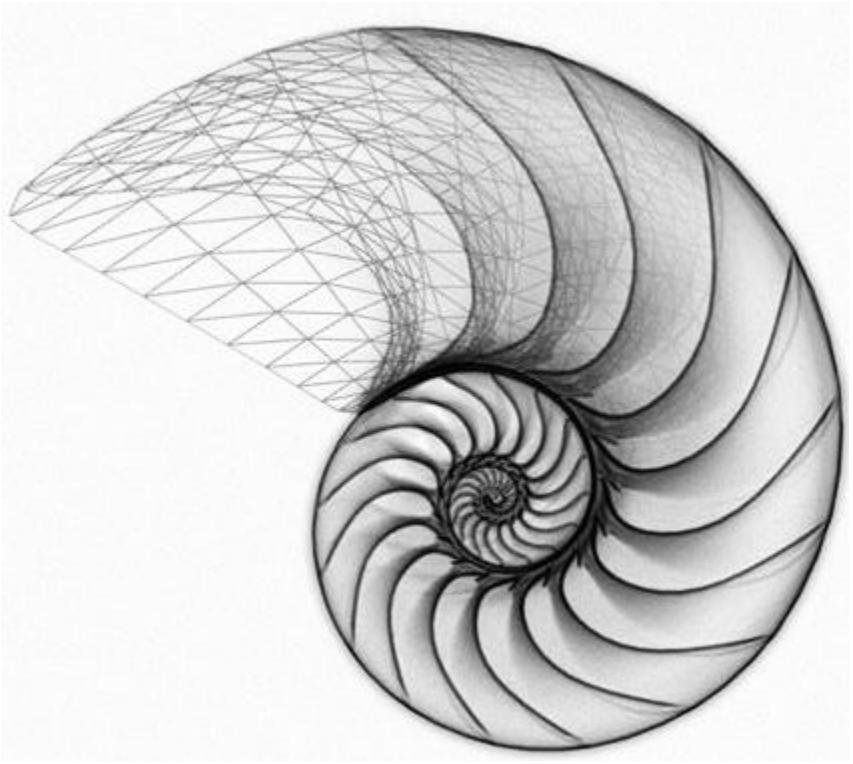


Lecture 05

Abstract Data Type - Unsorted List and Sorted List (Array-based Implementation)

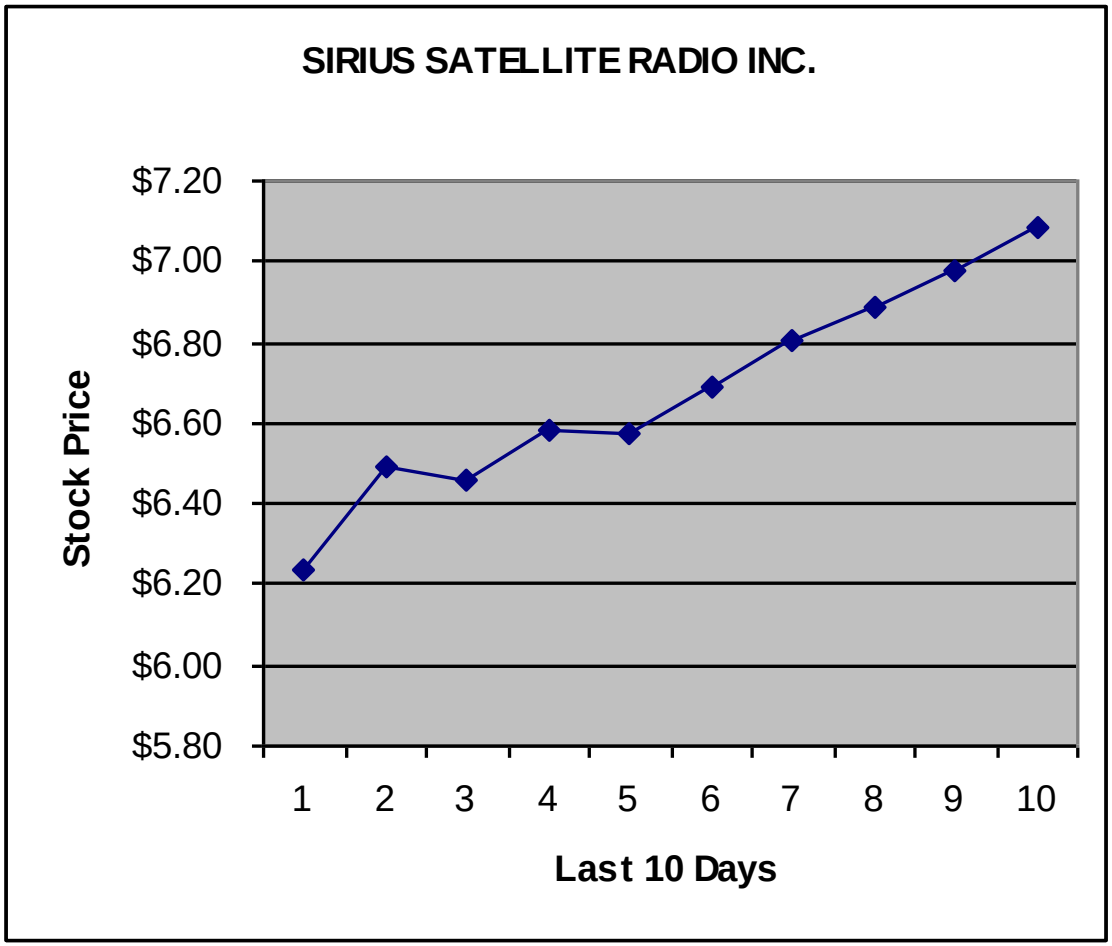
CSE225: Data Structures and Algorithms

Data vs. Information

Data

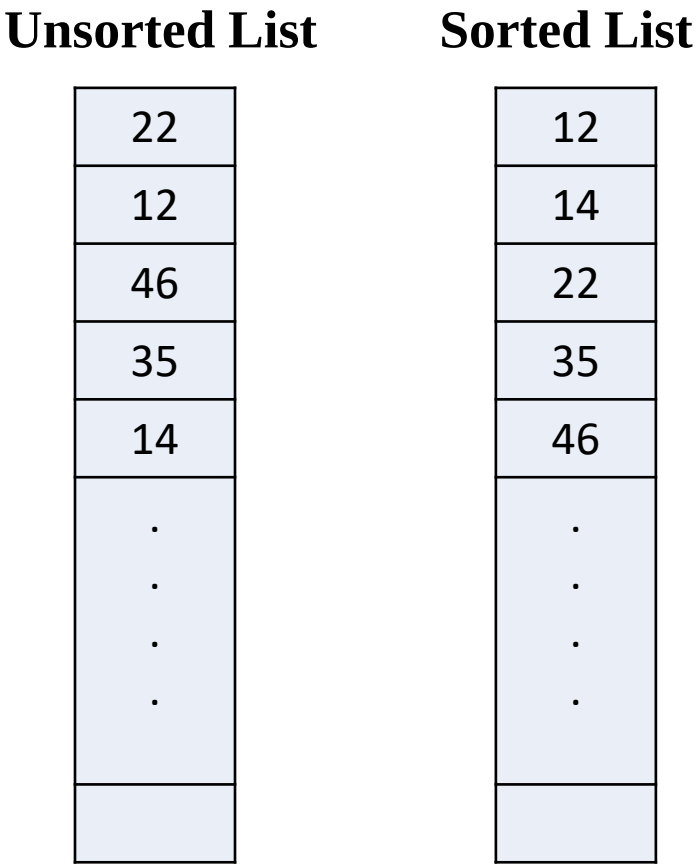
- 6.34
- 6.45
- 6.39
- 6.62
- 6.57
- 6.64
- 6.71
- 6.82
- 7.12
- 7.06

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
46
35
14
12
14
22
35
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 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 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)

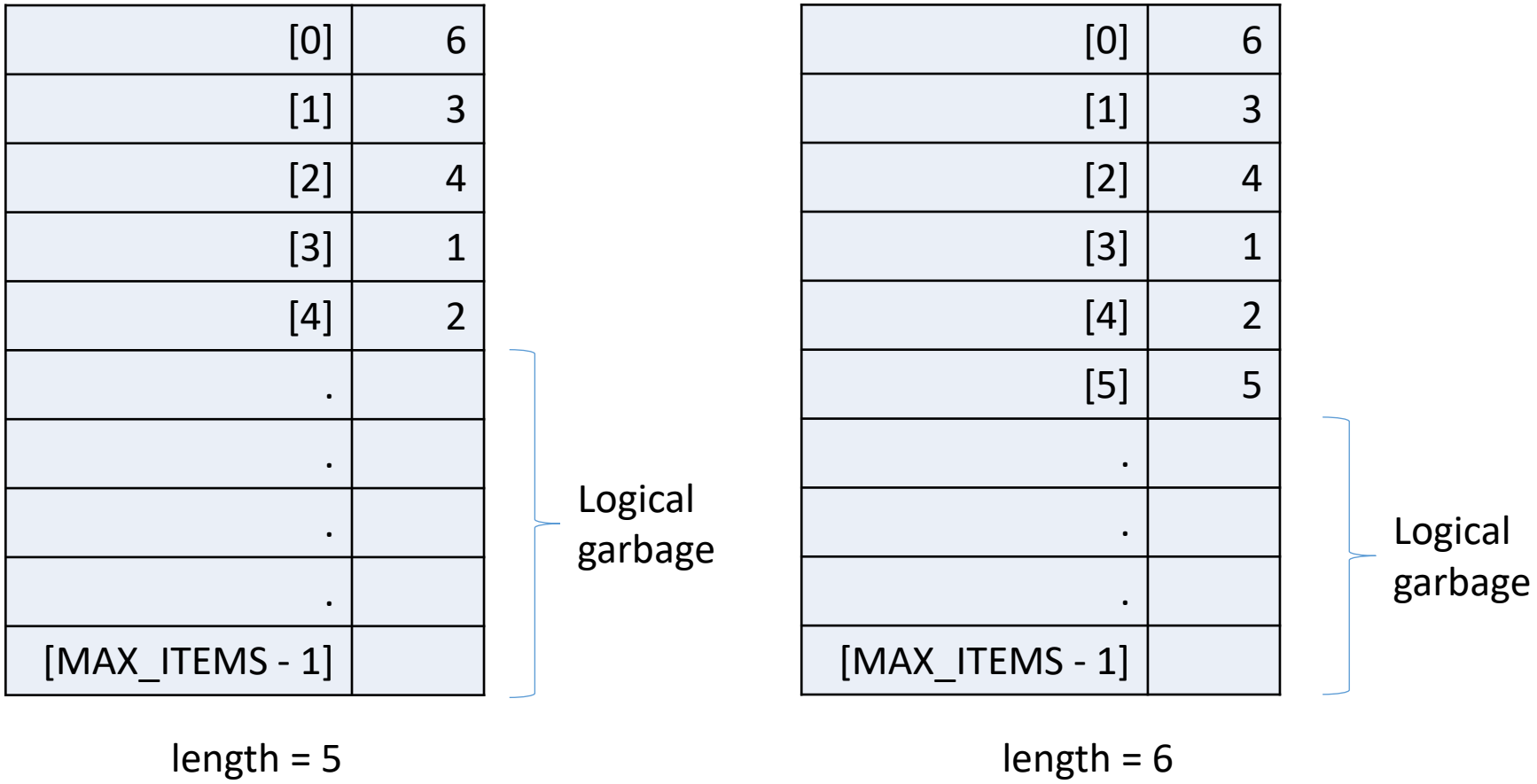
```
template <class ItemType>
void SortedType<ItemType>::ResetList()
{
    currentPos = -1;
}
```

O(1)

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

O(1)

Inserting an Item into Unsorted List



Insert 5

unsortedtype.cpp

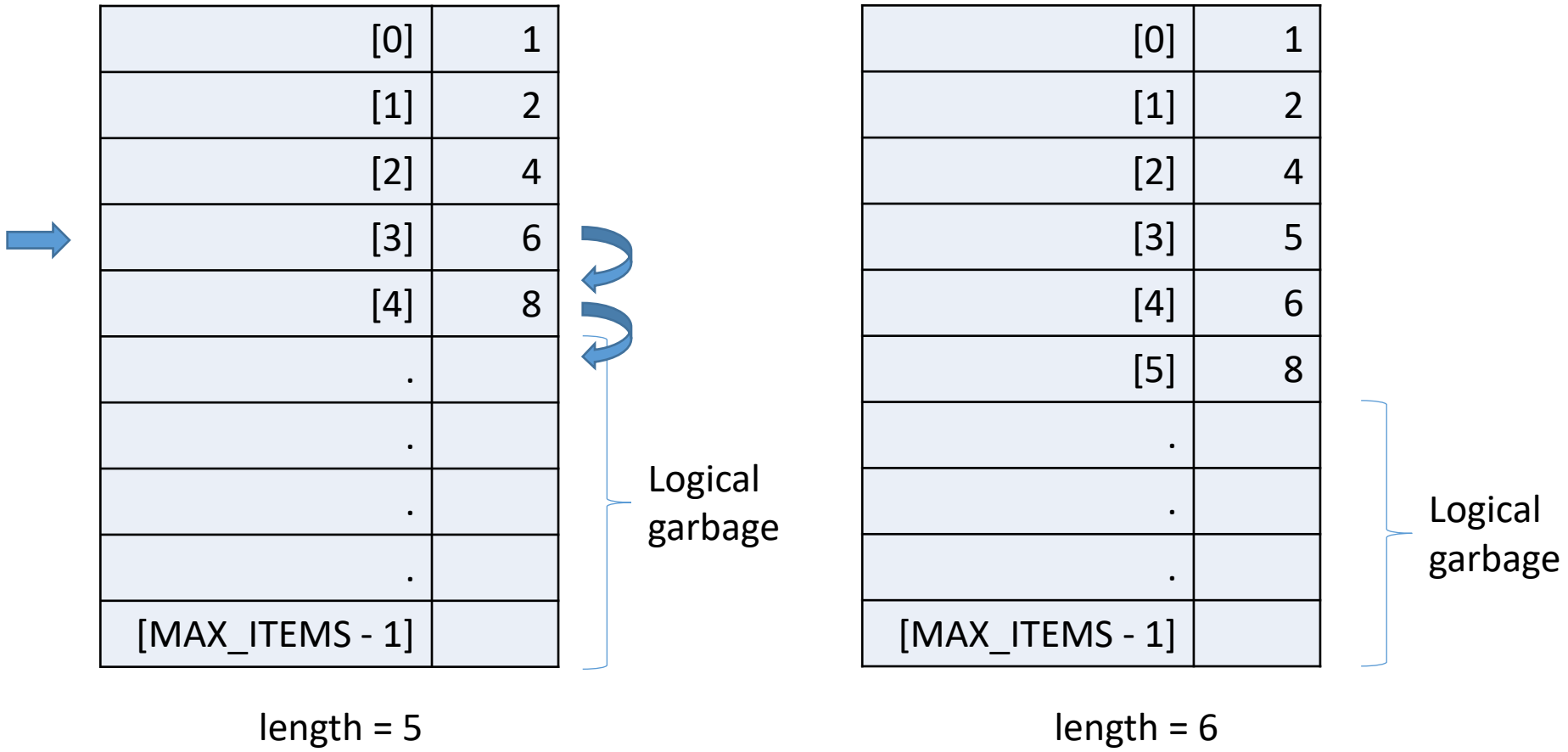
```
template <class ItemType>
void UnsortedType<ItemType>::InsertItem(ItemType item)
{
    info[length] = item;
    length++;
}
```

unsortedtype.cpp

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

O(1)

Inserting an Item into Sorted List



Insert 5

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++;
}
```

sortedtype.cpp

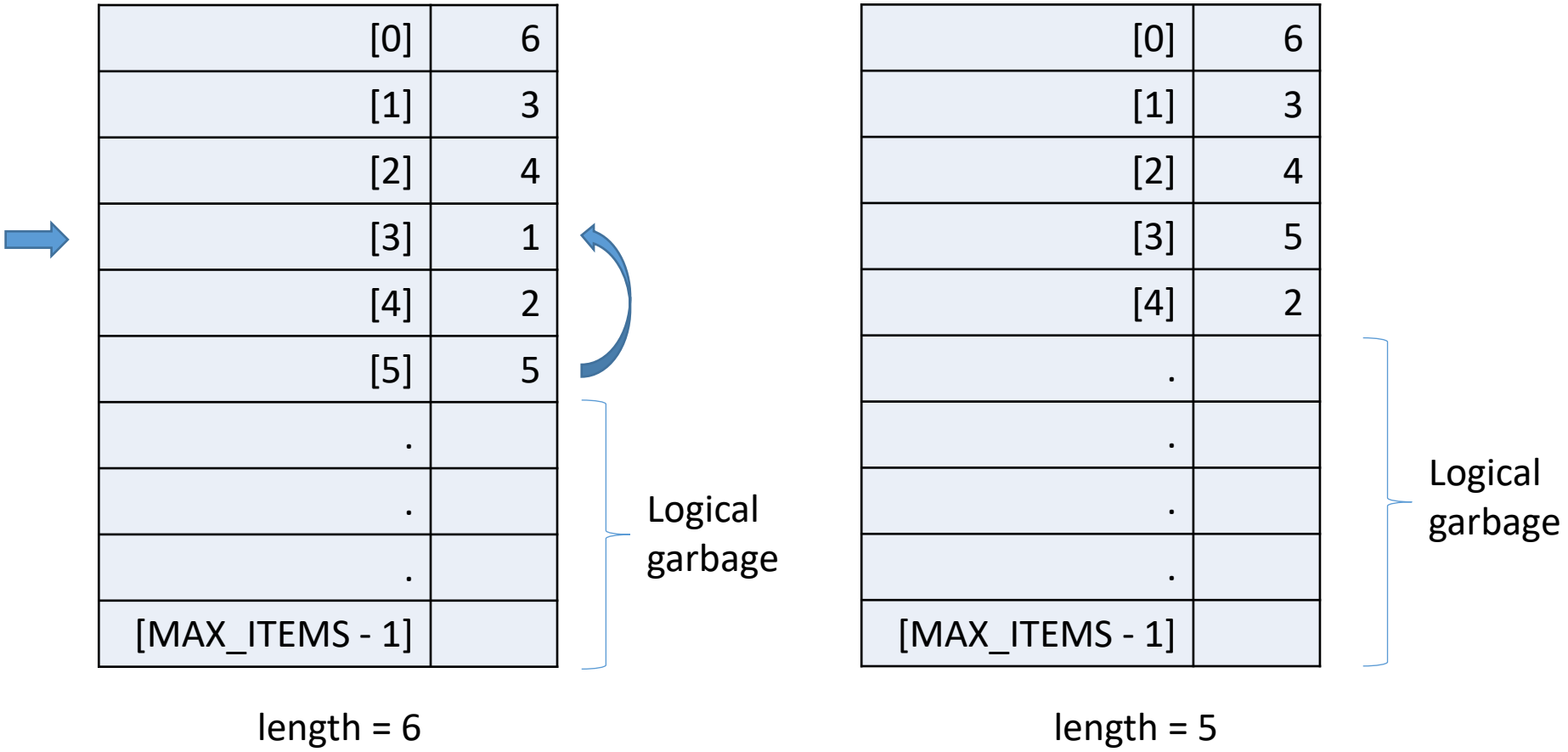
```
template <class ItemType>
void SortedType<ItemType>::InsertItem(ItemType item)
{
    int location = 0;
    bool moreToSearch = (location < length);

    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 Unsorted List



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

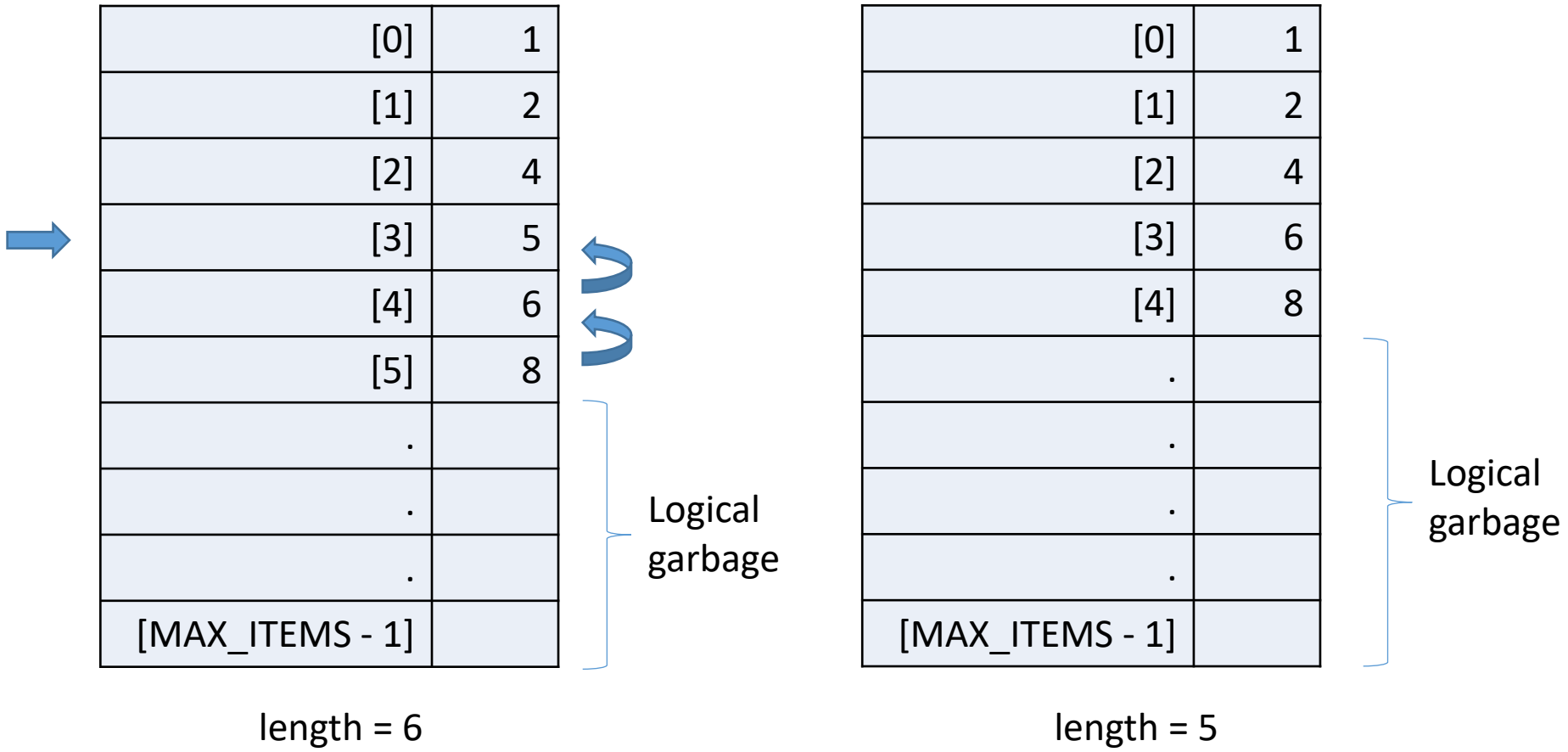
```
template <class ItemType>
void UnsortedType<ItemType>::DeleteItem(ItemType item)
{
    int location = 0;
    while (item != info[location])
        location++;
    info[location] = info[length - 1];
    length--;
}
```

O(N)

O(1)

O(N)

Deleting an Item from Sorted List



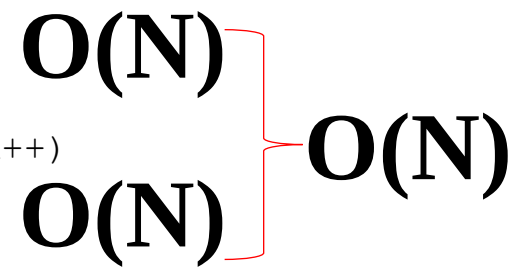
Delete 5

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--;
}
```



Retrieving an Item from Unsorted List

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

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);
        }
    }
}
```

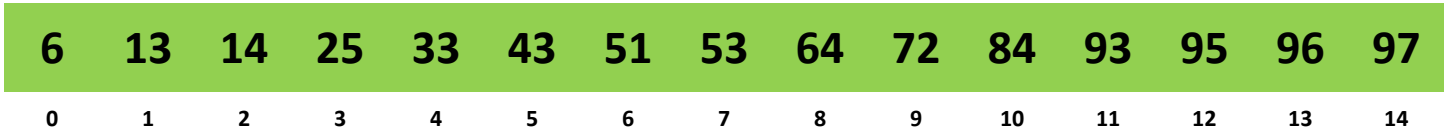
unsortedtype.cpp

```
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)

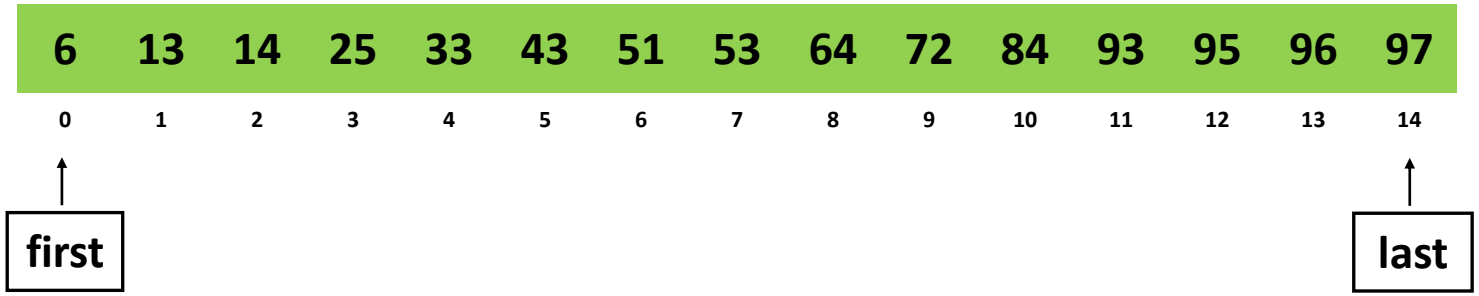
Retrieving an Item from Sorted List

- Find **84**



Retrieving an Item from Sorted List

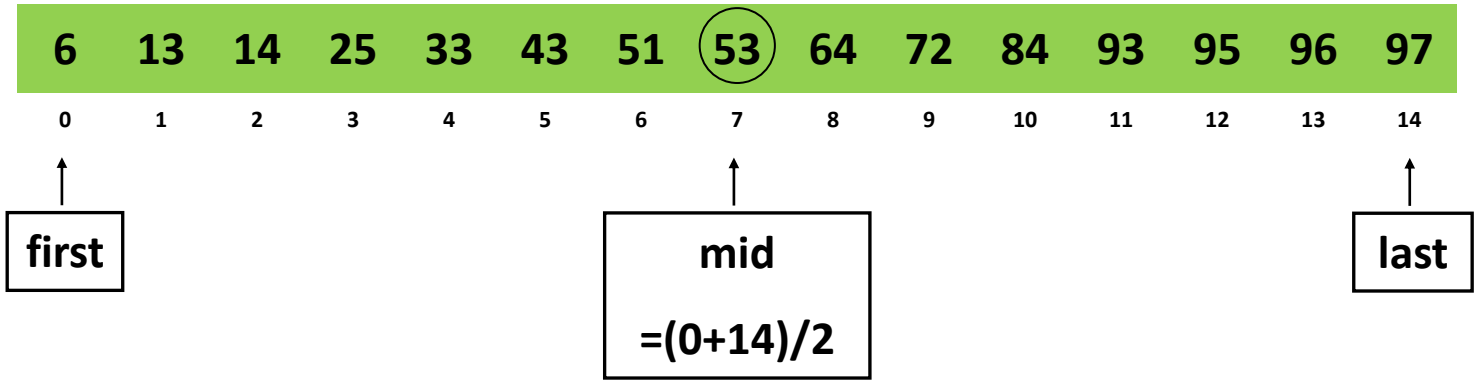
- Find **84**



- **Step 1**

Retrieving an Item from Sorted List

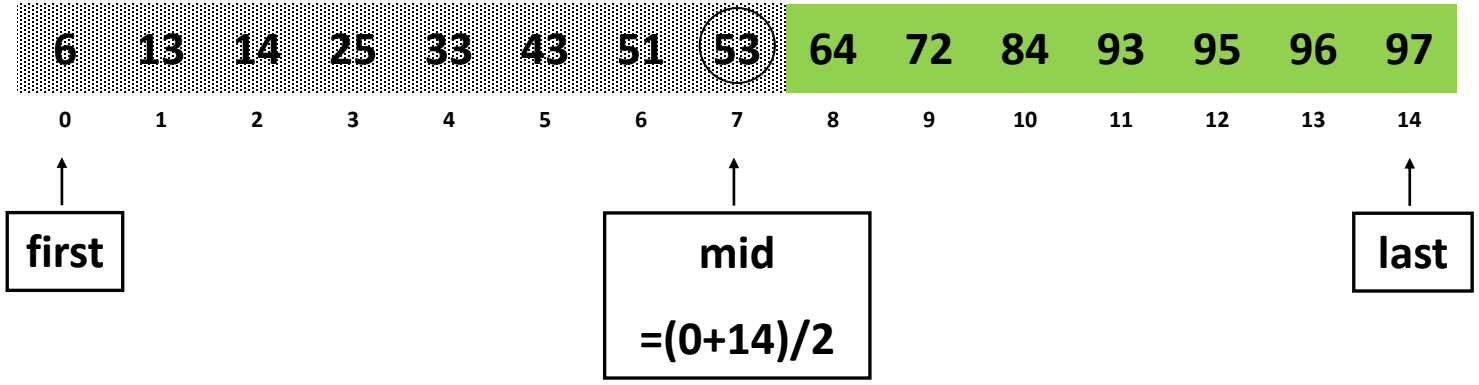
- Find **84**



- **Step 1**

Retrieving an Item from Sorted List

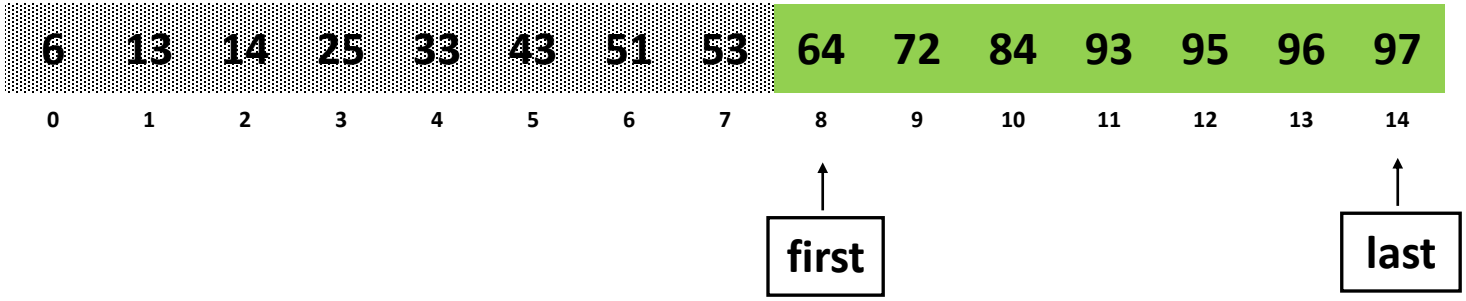
- Find **84**



- **Step 1**

Retrieving an Item from Sorted List

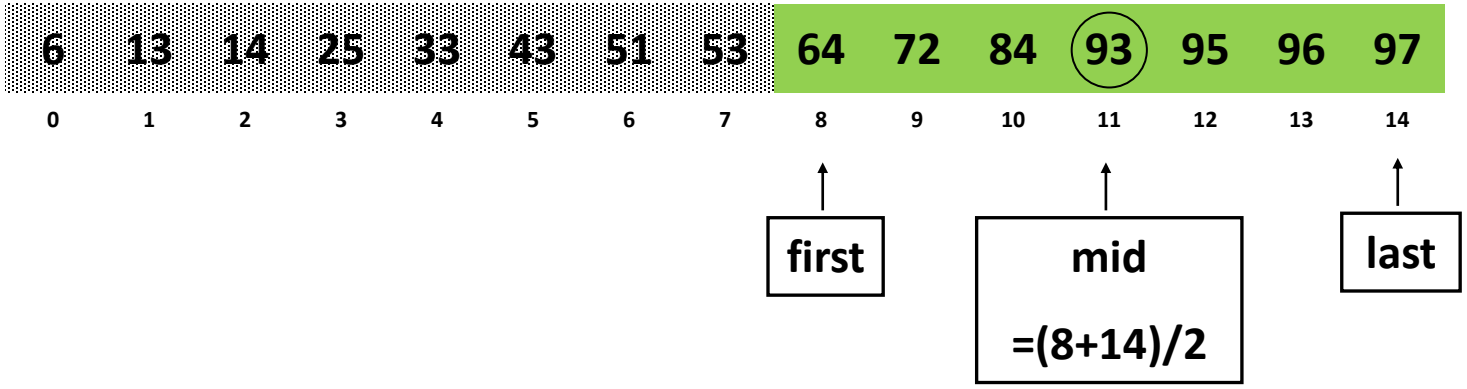
- Find **84**



- **Step 2**

Retrieving an Item from Sorted List

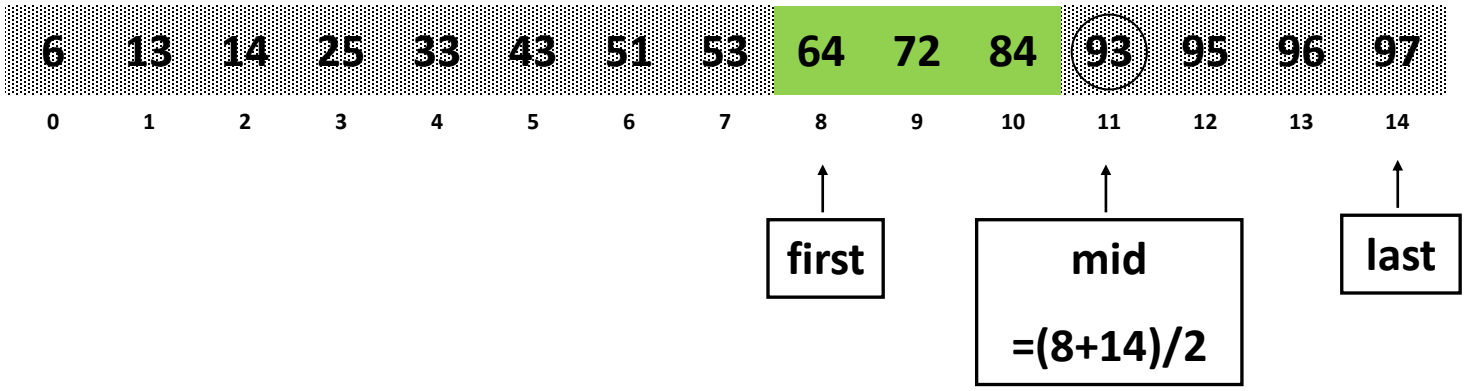
- Find **84**



- **Step 2**

Retrieving an Item from Sorted List

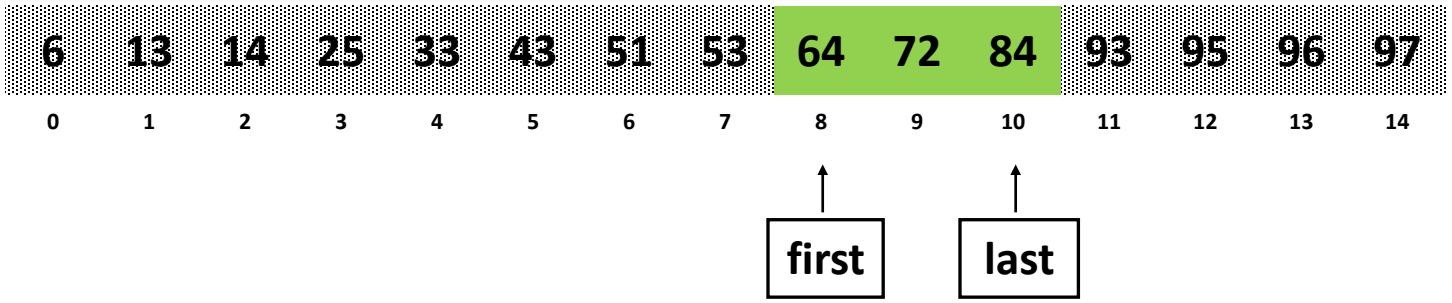
- Find **84**



- **Step 2**

Retrieving an Item from Sorted List

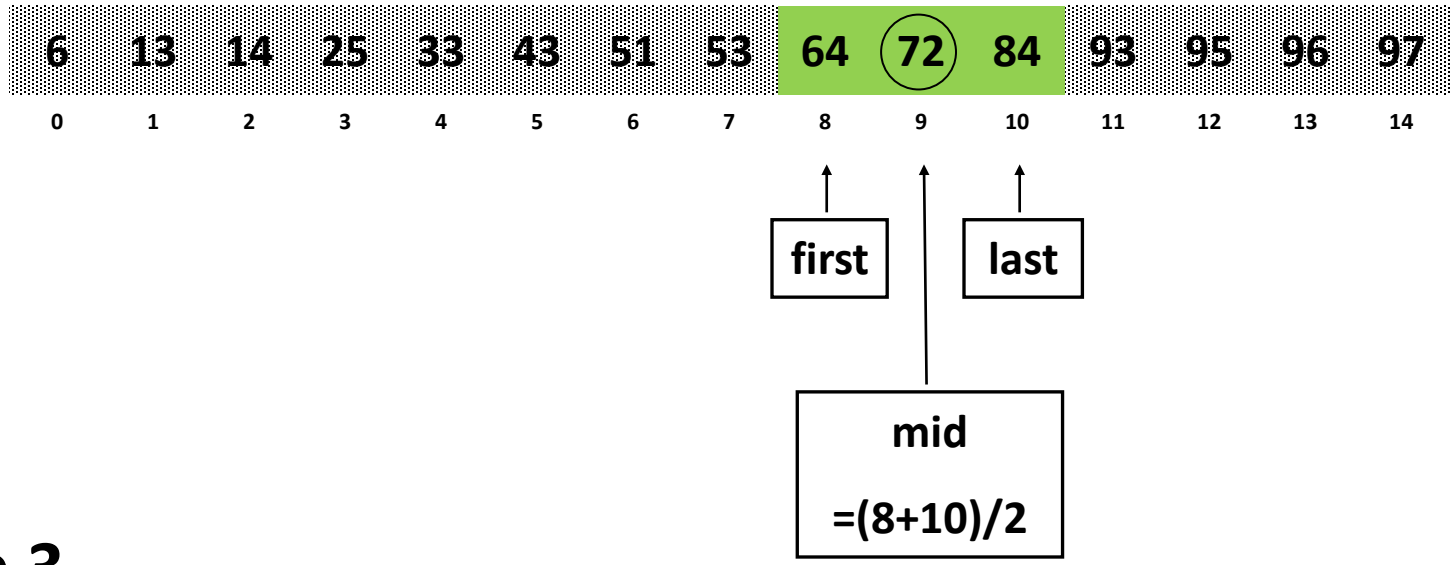
- Find **84**



- **Step 3**

Retrieving an Item from Sorted List

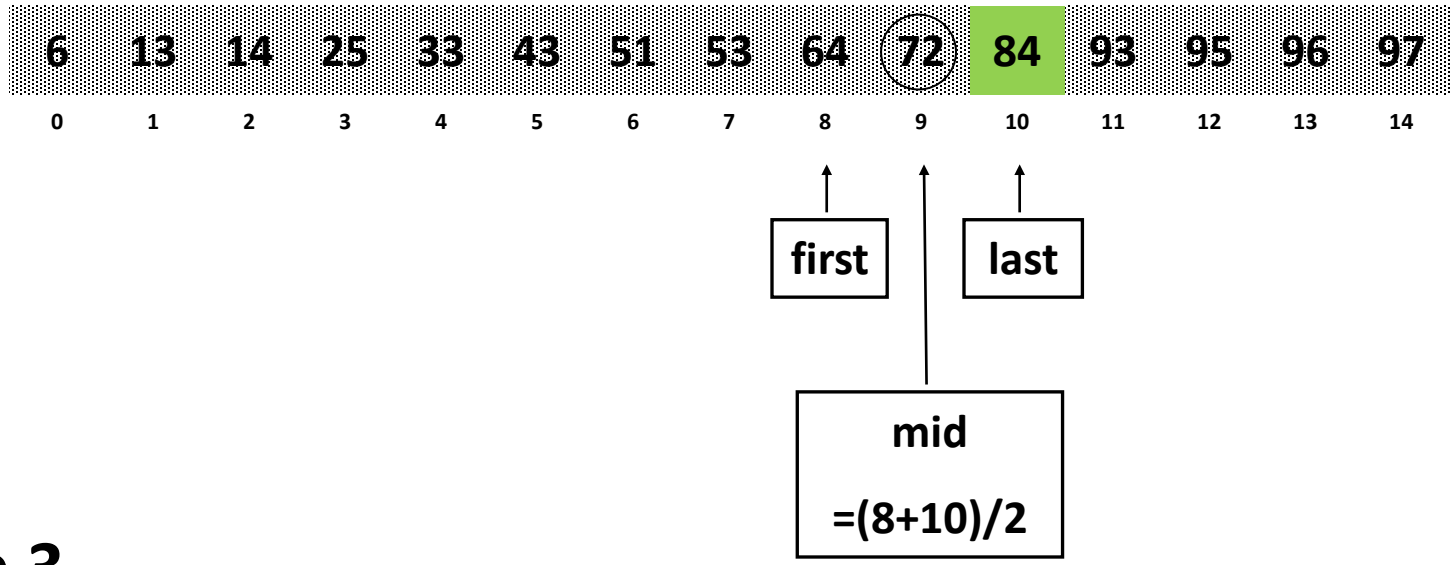
- Find **84**



- **Step 3**

Retrieving an Item from Sorted List

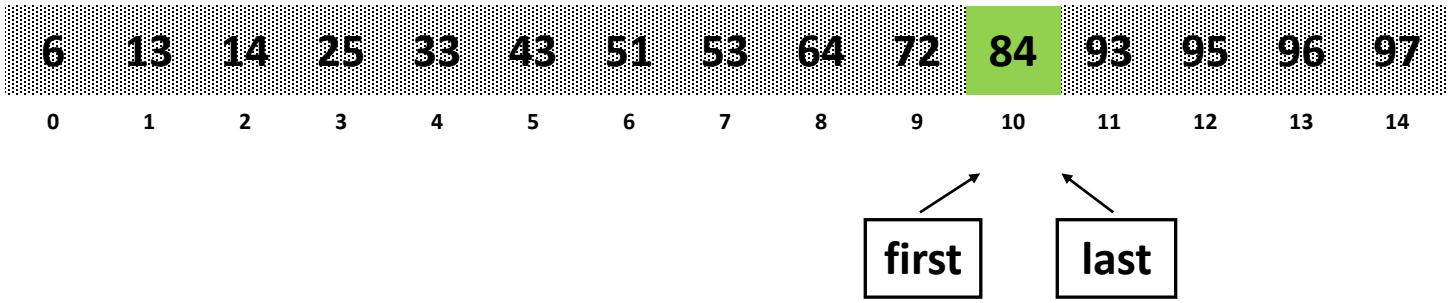
- Find **84**



- **Step 3**

Retrieving an Item from Sorted List

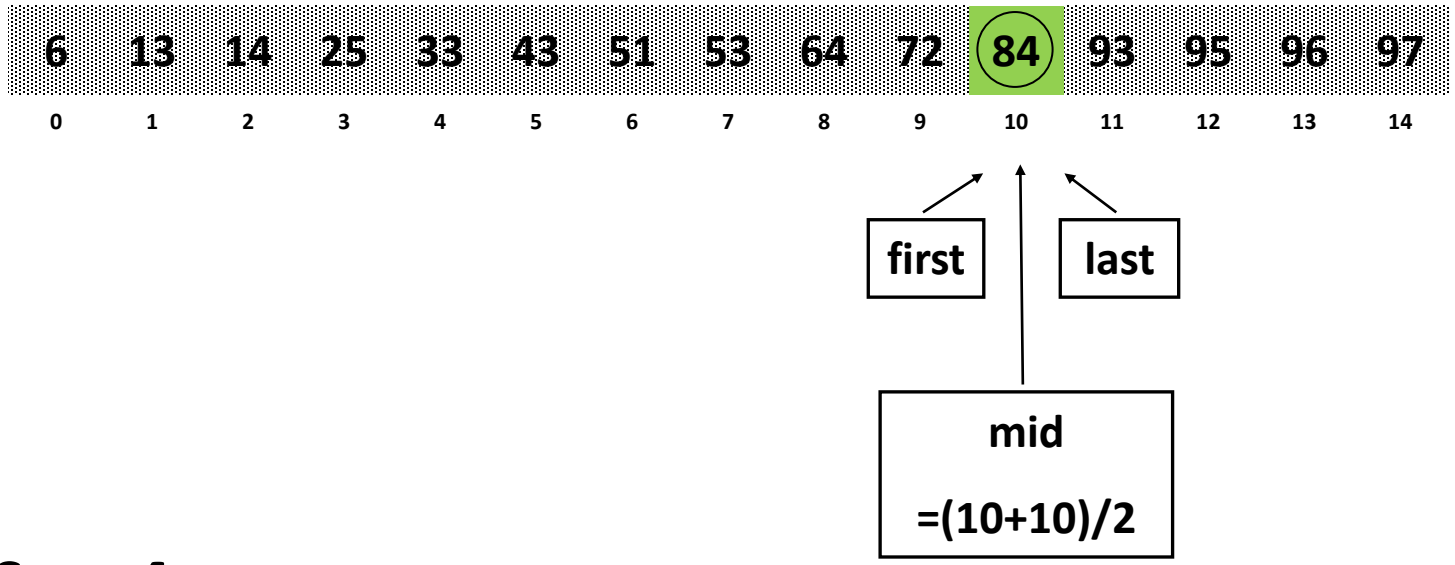
- Find **84**



- **Step 4**

Retrieving an Item from Sorted List

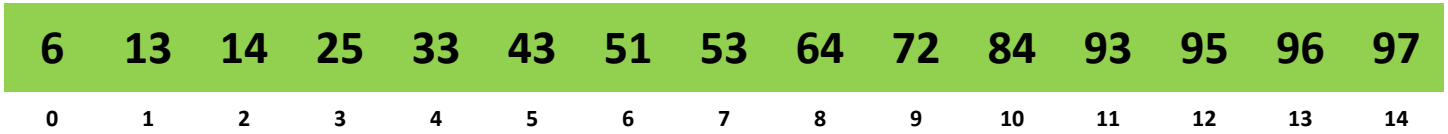
- Find **84**



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

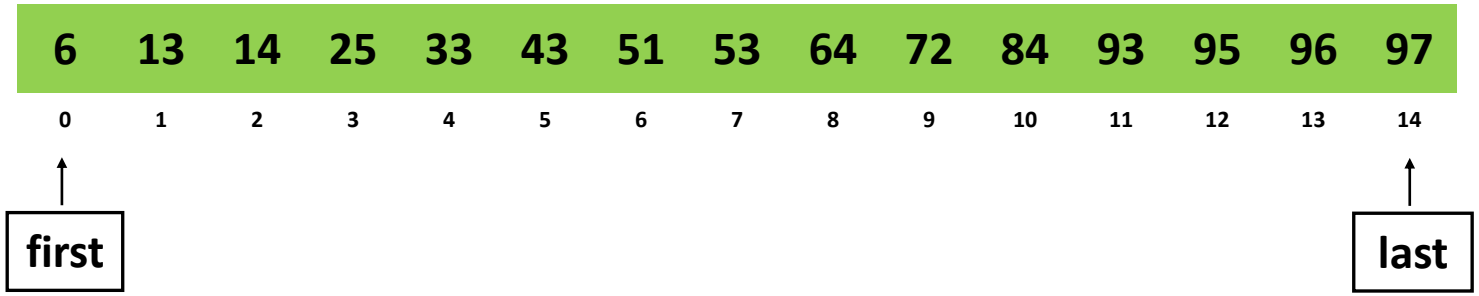
Retrieving an Item from Sorted List

- Find **73**



Retrieving an Item from Sorted List

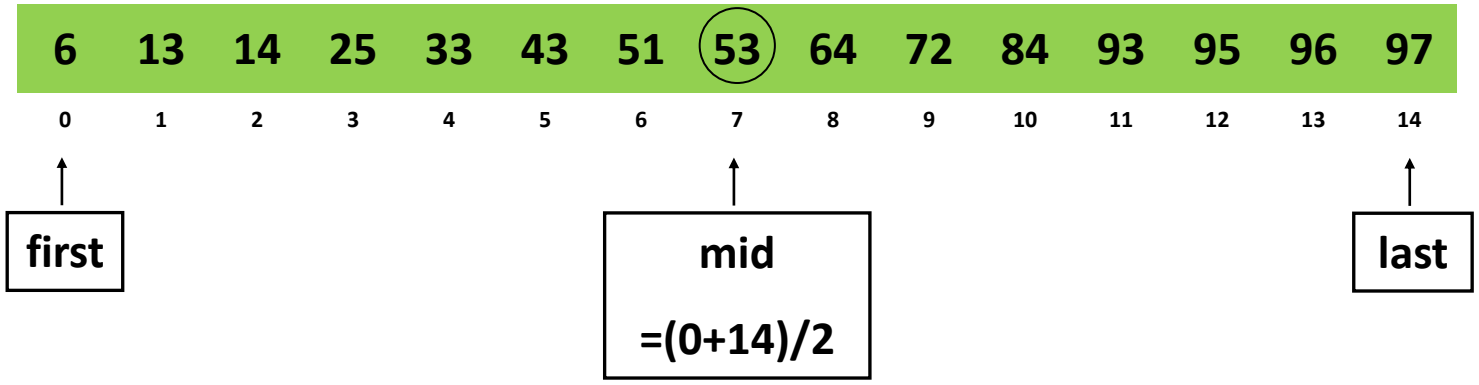
- Find **73**



- **Step 1**

Retrieving an Item from Sorted List

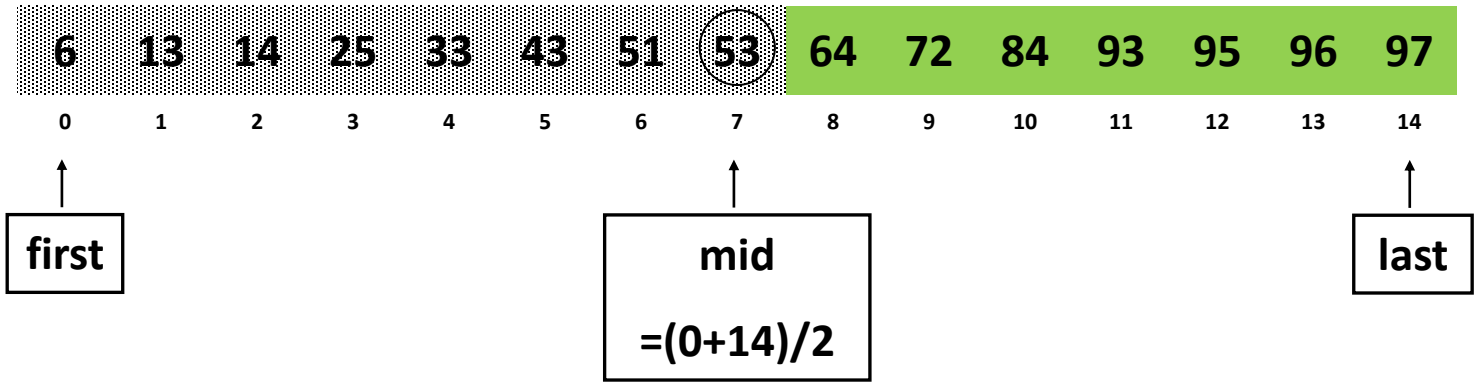
- Find **73**



- **Step 1**

Retrieving an Item from Sorted List

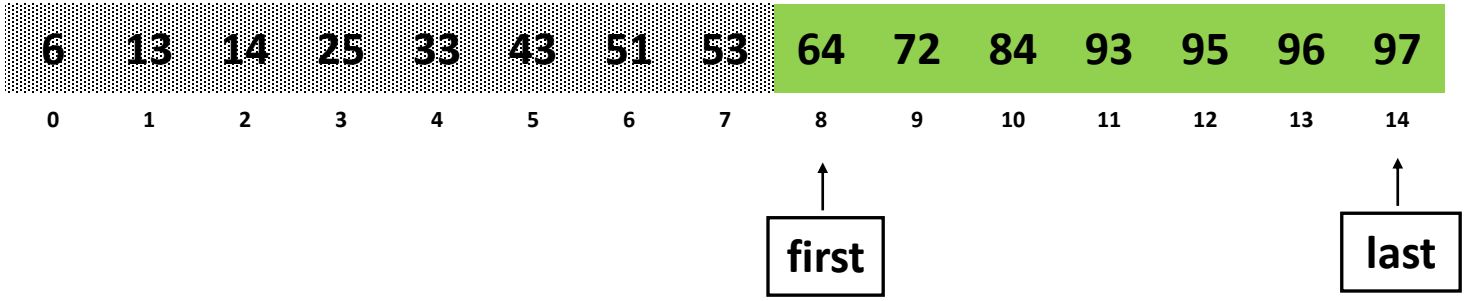
- Find **73**



- **Step 1**

Retrieving an Item from Sorted List

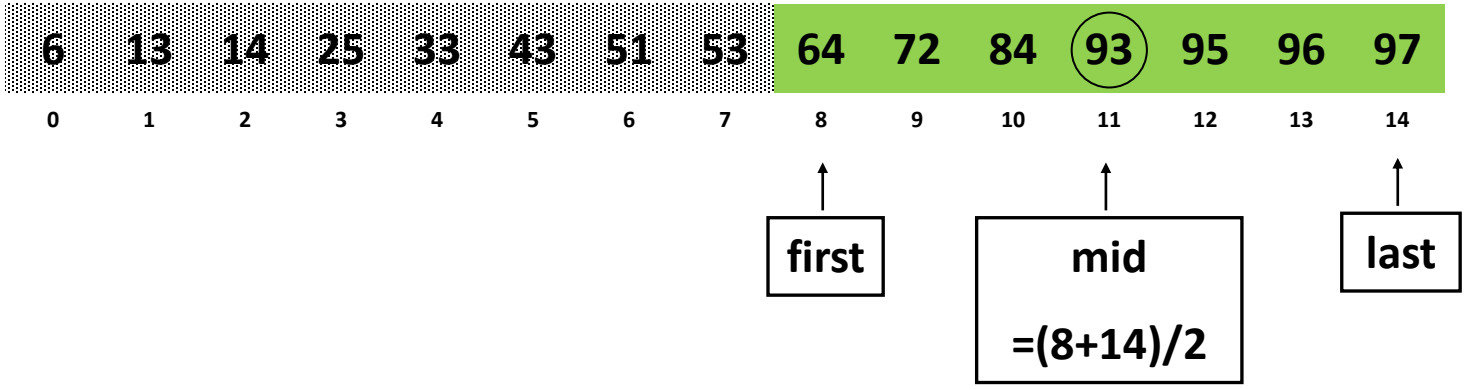
- Find **73**



- **Step 2**

Retrieving an Item from Sorted List

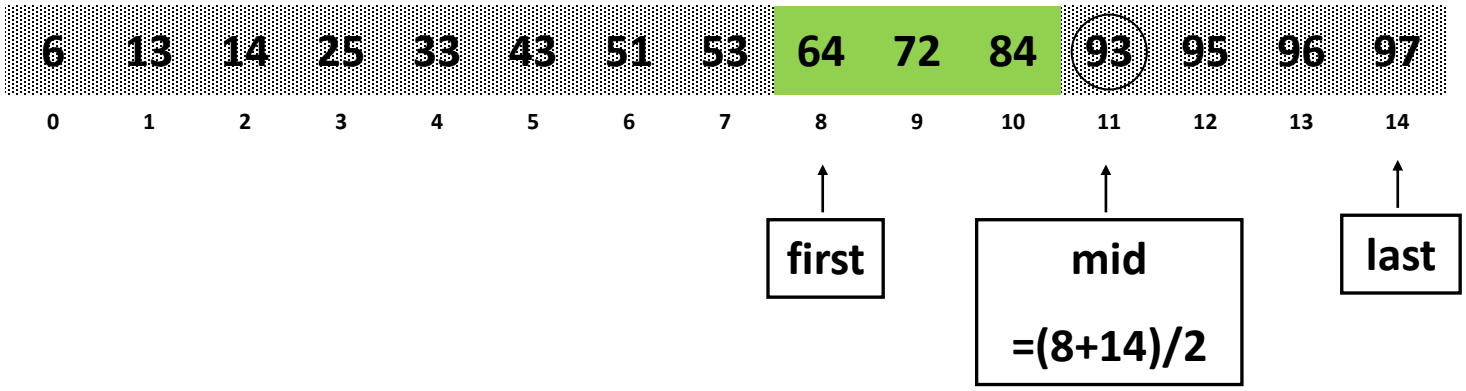
- Find **73**



- **Step 2**

Retrieving an Item from Sorted List

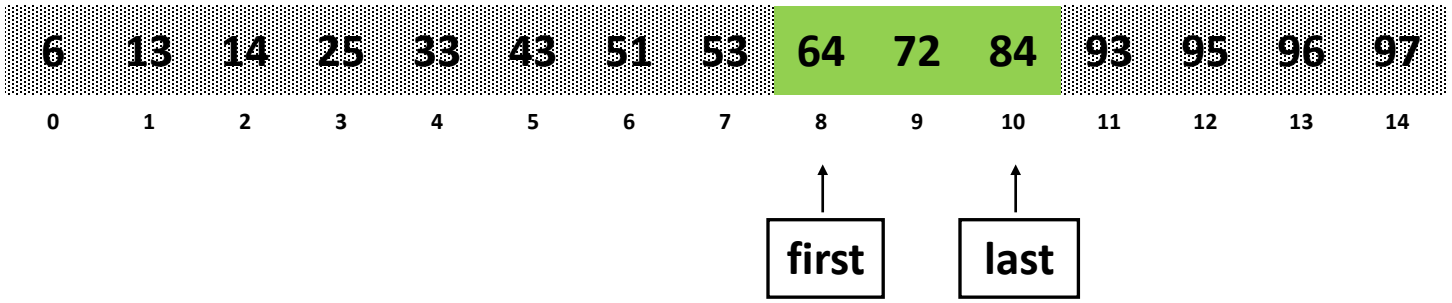
- Find **73**



- **Step 2**

Retrieving an Item from Sorted List

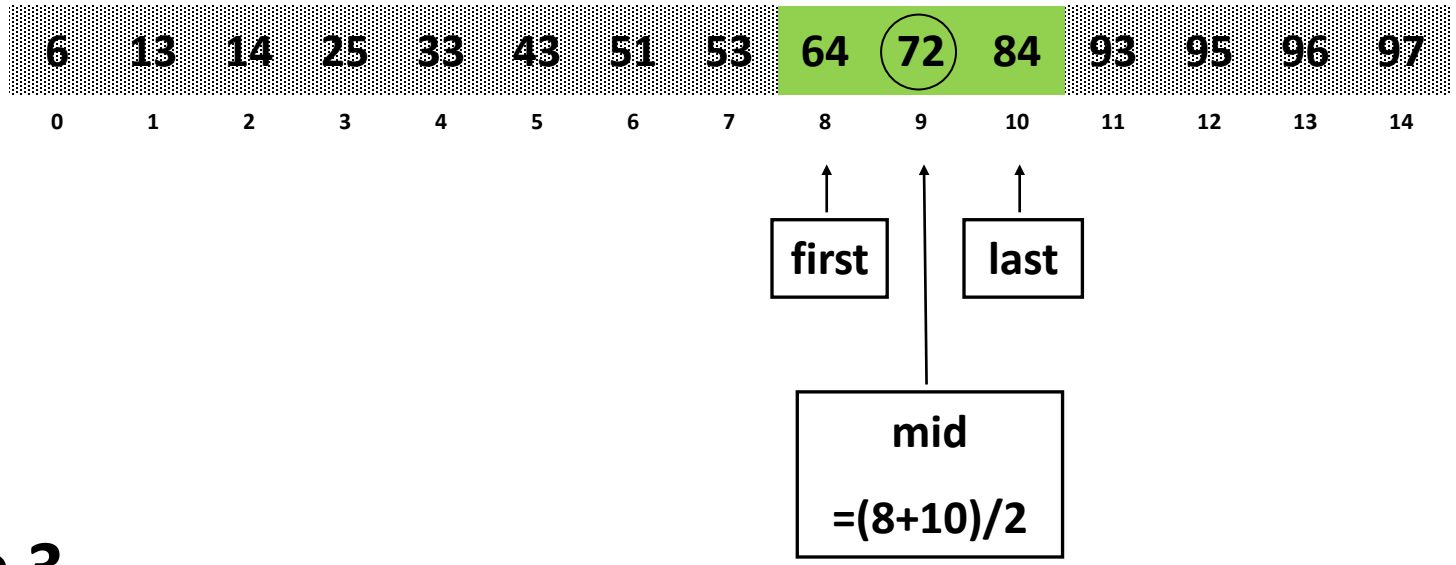
- Find **73**



- **Step 3**

Retrieving an Item from Sorted List

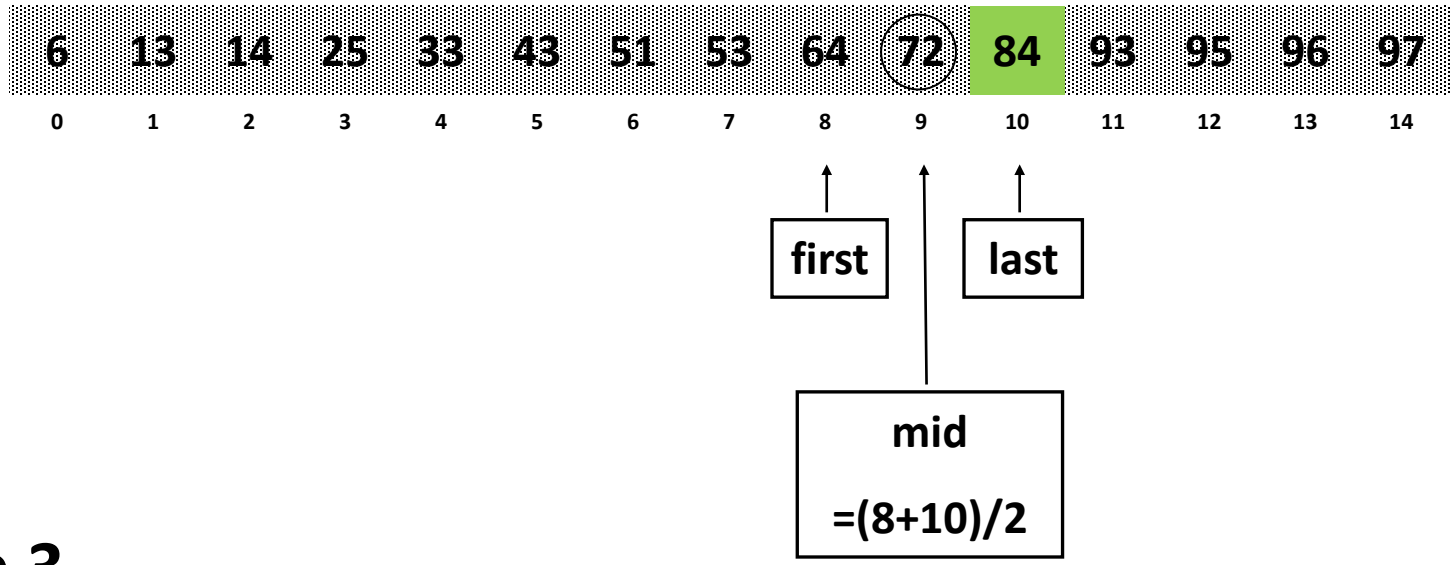
- Find **73**



- **Step 3**

Retrieving an Item from Sorted List

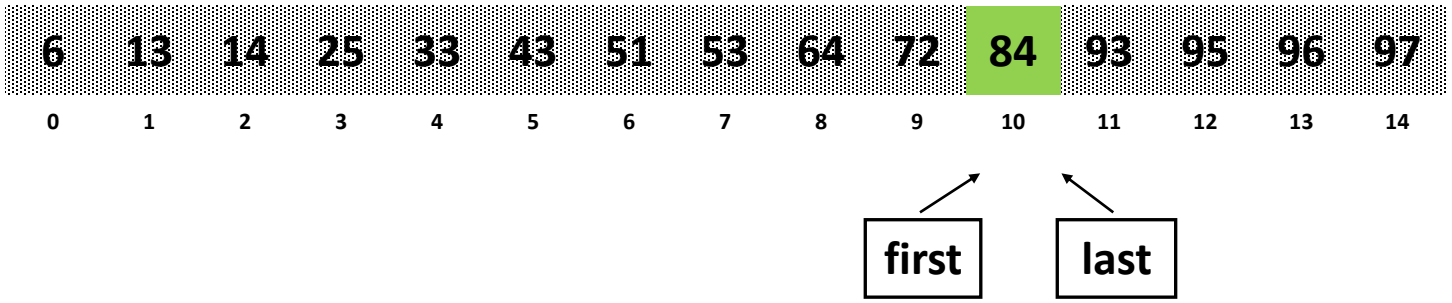
- Find **73**



- **Step 3**

Retrieving an Item from Sorted List

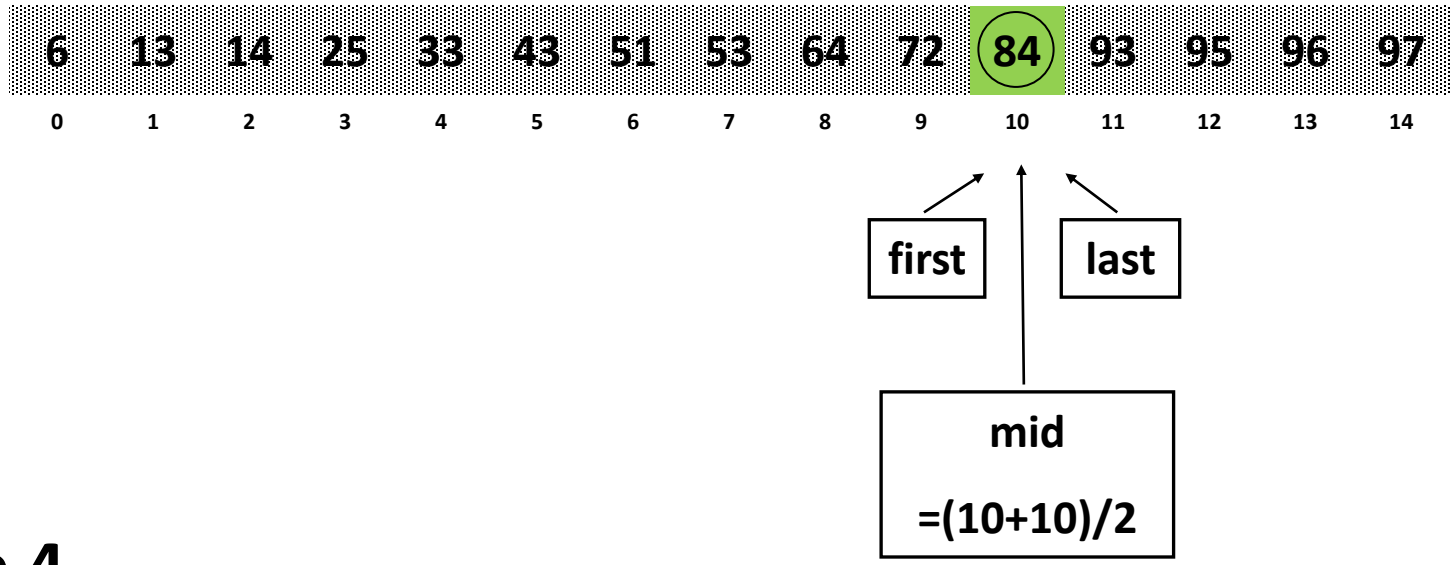
- Find **73**



- **Step 4**

Retrieving an Item from Sorted List

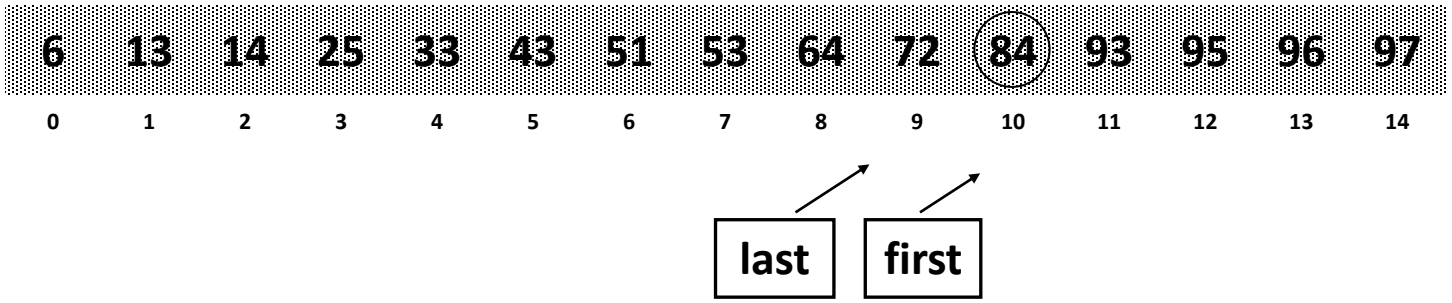
- Find **73**



- **Step 4**

Retrieving an Item from Sorted List

- Find **73**



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

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

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,

$$x = \log N$$

sortedtype.cpp

```
template <class ItemType>
void SortedType<ItemType>::RetrieveItem(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];
        }
    }
}
```

sortedtype.cpp

```
template <class ItemType>
void SortedType<ItemType>::RetrieveItem(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];
        }
    }
}
```

O(logN)