Week 2: List ADT—Array

CS-250 Data Structure and Algorithms

DR. Mehwish Fatima | Assist. Professor
Department of AI & DS | SEECS, NUST

Goals

- After studying this, you should be able to
  - Use the list operations to implement utility routines to do the following application-level tasks:
    - Print the list of elements
    - Create a list of elements from a file
  - Implement list operations for both unsorted lists and sorted lists:
    - Create and destroy a list
    - Determine whether the list is full
    - Insert an element
    - Retrieve an element
    - Delete an element

Abstract Data Types (ADTs)

- The basic idea of an abstract data type is to focus on what it can do (**operations**)
  - For example an abstract stack data structure could be defined by two operations:
    - push—inserts some data item into the structure, and
    - pop—extracts an item from it.

- The advantages of using the ADT approach are:
  - The implementation of ADT can change without affecting those method that use the ADT, like we can implement stack using both an array and a linked list
  - The complexity of the implementation are hidden.

List ADT

- A list is a homogeneous collection of elements, with a **linear** relationship.
  - Linear relationship
    - Each element except the first has a unique predecessor, and
    - each element except the last a unique predecessor.
  - Length
    - The number of items in a list;
    - the length can vary over time

Types of List ADT

- Unsorted list
  - A list in which data items are placed in no particular order;
  - the only relationships between data elements are the list predecessor and successor relationships

- Sorted list
  - A list that is sorted by the value in the key;
  - a semantic relationship exists among the keys of in the list items

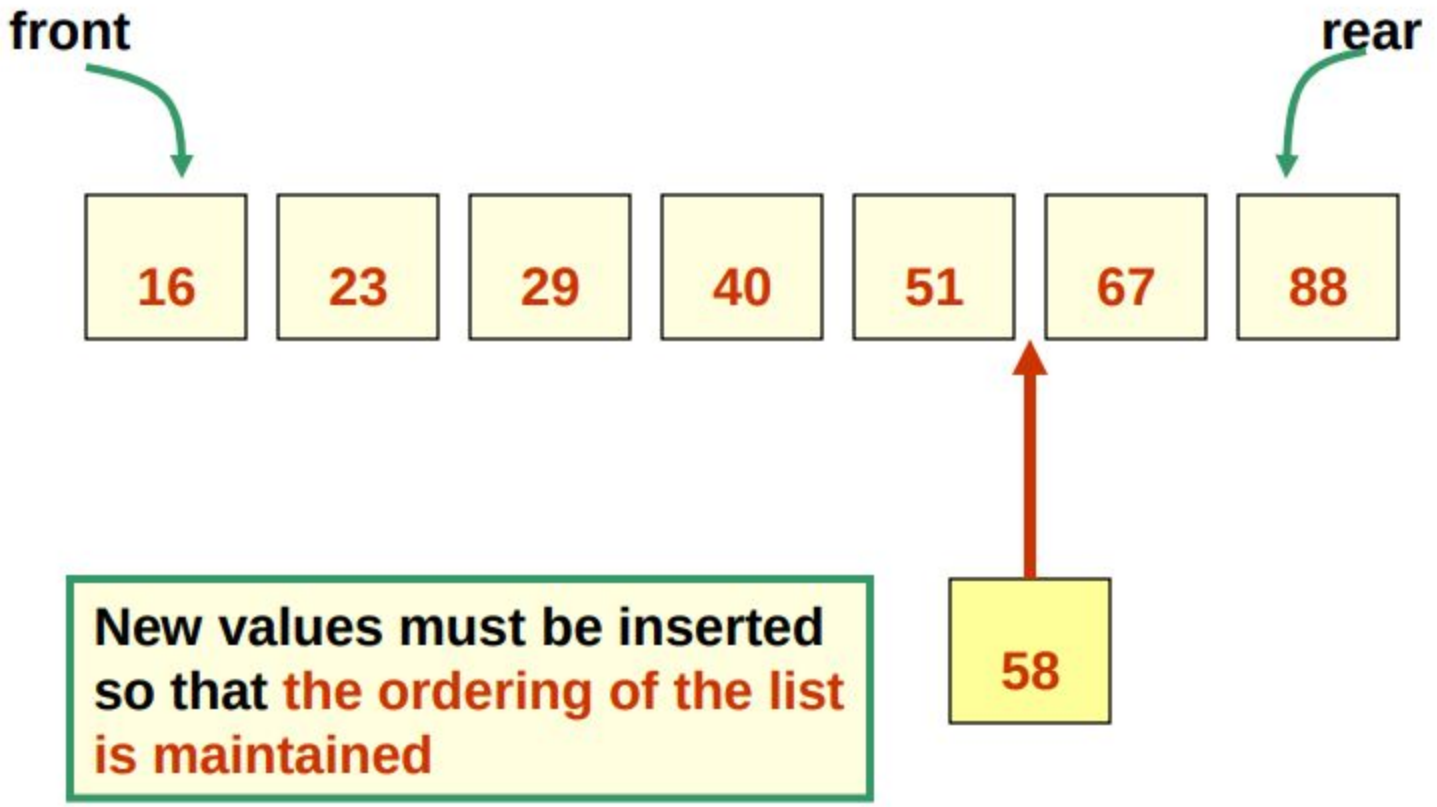
- Key
  - A member of a record (struct or class) whose value is used to determine the logical and/or physical order of the items of a list

Types of List ADT

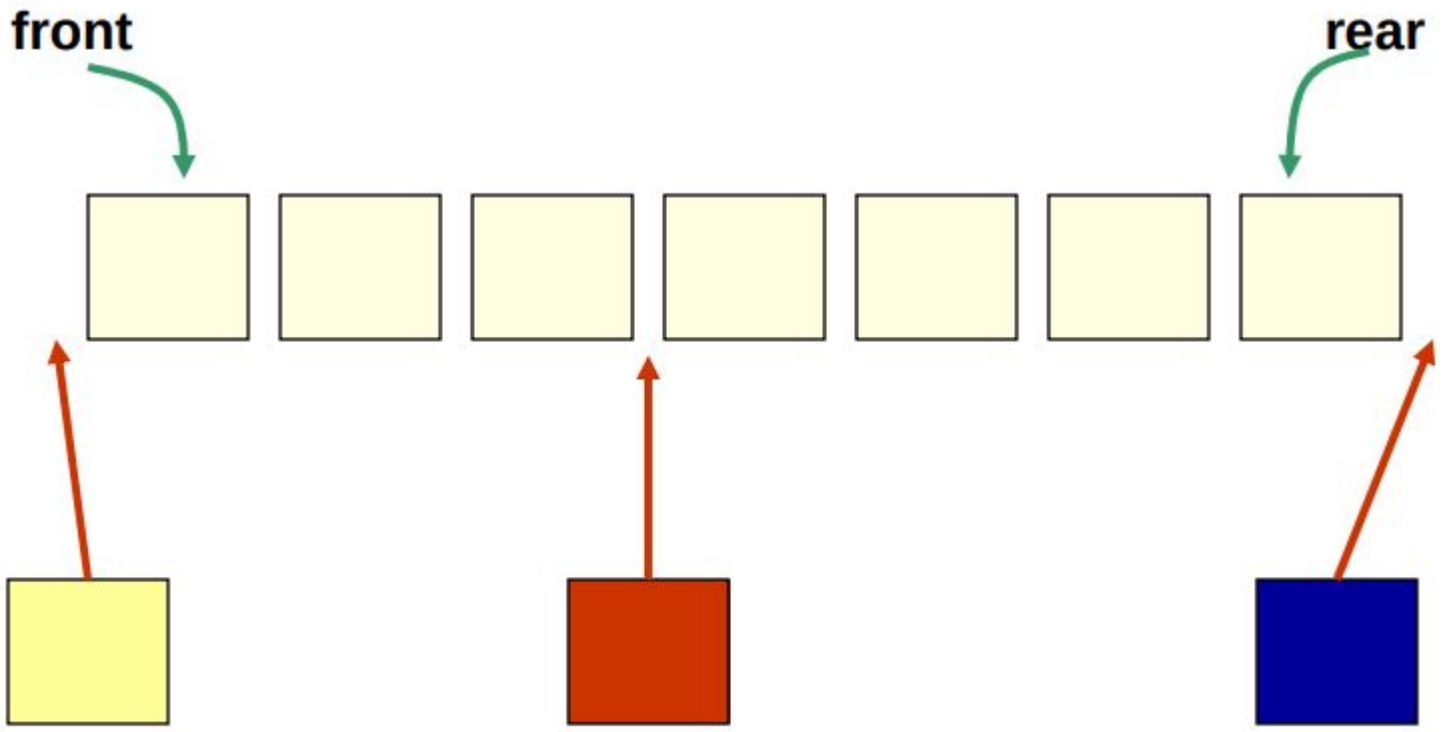
- List may be ordered in different ways
  - Lists can also be ordered by value.
  - A list of names can be ordered alphabetically.
  - A list of grades can be ordered numerically.

Index	0	1	2	3	4	5
Data	Eggs	Bread	Milk	Fruits		

Conceptual View of an Ordered List



Conceptual View of an Unordered List



New values can be inserted anywhere in the list

Implementation: Logical Level

- Programmers can provide many different operations for lists.
- For different applications
  - we can imagine all kinds of things users might need to do to a list of elements.
  - we formally define a list and develop a set of general-purpose operations for creating and manipulating lists.
  - By doing so, we build an abstract data type.

Implementation Consideration: LIST ADT

- There is a representative set of list operations where
  - ***L*** is a list of objects/elements of type **elementType**,
  - ***x*** is an object/element of that type, and
  - ***p*** is of type position.

- **Note: "position"** is another data type whose implementation will vary for different list implementations.

Operations on List ADT

Operations on List ADT

Constructors

- A constructor creates an instance of the data type. It is usually implemented with a language-level declaration.

Transformers

- Transformers are operations that change the structure in some way:
  - They may make the structure empty, put an item into the structure, or remove a specific item from the structure.
  - For our **Unsorted** List ADT, let's call these transformers
    - MakeEmpty, InsertItem, and DeleteItem.

Operations on List ADT

Transformers

- Transformers are operations that change the structure in some way:
  - **MakeEmpty** needs only the list, no other parameters.
  - **InsertItem** and **DeleteItem** need an additional parameter:
    - the item to be inserted or removed.
  - A transformer that takes two sorted lists and merges them into one sorted list or appends one list to another would be a binary transformer.

Operations on List ADT

Observers

- Observers come in several forms.
- They ask **true/false** questions about the data type
  - Is the structure empty?
  - Select or access a particular item
    - Give me a copy of the last item
  - Return a property of the structure
    - How many items are in the structure?

Operations on List ADT

Observers

- The Unsorted List ADT needs at least two observers:
  - **IsFull** returns true if the list is full;
  - **LengthIs** tells us how many items appear in the list.
  - Another useful observer **searches** the list for an item with a particular key and returns a copy of the associated information if it is found; let's call it **RetrieveItem.**
  - **min, max, average**

Operations on List ADT

Iterators

- Iterators are used with composite types to allow the user to process an entire structure, component by component.

- To give the user access to each item in sequence
  - **ResetList:** one to initialize the iteration process
    - analogous to **Reset** or Open with a file

    - **ResetList** is not an iterator itself, but rather an auxiliary operation that supports the iteration.

Operations on List ADT

Iterators

- Iterators are used with composite types to allow the user to process an entire structure, component by component.

- To give the user access to each item in sequence
  - **GetNextItem:** to return a copy of the "**next component**" each time it is called.

  - **Foreach:** sed to apply a specific operation or function to each element in the list.

List ADT: Generic Data Types

- The operations are defined but the types of the items being manipulated are not.
  - We let the user define the type of the items on the list in a class named **ItemType** and have our **Unsorted List ADT** include the class definition.

- Two of the list operations (**DeleteItem** and **RetrieveItem**) will involve the comparison of the keys of two list components
  - so does **InsertItem** if the list is sorted by key value.

List ADT: Generic Data Types

- We could require the user to name the key data member "**key**" and compare the key data members using relational operators.

- This approach isn't a very satisfactory solution for two reasons:
  - "**key**" is not always a meaningful identifier in an application program, and the keys would be limited to values of simple types.

  - needs a way to change the meaning of the relational operators

List ADT: Generic Data Types

- We let the user define a member function **ComparedTo** in the class **ItemType**.
  - This function compares two items and
  - returns LESS, GREATER, or EQUAL
  - depending on whether the key of one item comes before the key of the other item,
  - the first key comes after it,
  - or the keys of the two items are equal, respectively.

List ADT: List Size

- Our ADT needs one more piece of information from the client: the maximum number of items on the list.

- As this information varies from application to application, it is logical for the client to provide it

Item Type

Class Name: Item Type	Superclass:	Subclasses:
Responsibilities	Collaborations	
Provide		
MAX_ITEMS		
enum RelationType (LESS, GREATER, EQUAL)		
Compared To (item) returns RelationType		
. . .		

Item Type

Class Name: Unsorted Type	Superclass:	Subclasses:

Responsibilities	Collaborations
MakeEmpty( )	
IsFull returns Boolean	
LengthIs returns integer	
RetrieveItem (item, found)	Item Type
InsertItem (item)	Item Type
DeleteItem (item)	Item Type
ResetList	
GetNextItem (item)	Item Type
.	
.	
.	

Unsorted List ADT

Unsorted List ADT

Structure

- The list elements are of **ItemType**.

- 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.

Unsorted List ADT

Definitions and Operation (provided by user)

- **MAX_ITEMS**
  - A constant specifying the maximum number of items on the list
- **ItemType**
  - Class encapsulating the type of the items in the list
- **RelationType**
  - An enumeration type that consists of LESS, GREATER, EQUAL

Unsorted List ADT

RelationType ComparedTo(ItemType item)

- Function
  - Determines the ordering of two **ItemType** objects based on their keys.
- Precondition
  - Self and item have their key members initialized.
- Postcondition
  - Function value = LESS if the key of self is less than the key of item.
  - = GREATER if the key of self is greater than the key of item.
  - = EQUAL if the keys are equal.

Constructor

ItemType CreateList(MAX_ITEMS)

- Function
  - Initializes list with the given size
- Precondition
  - None
- Postcondition
  - An empty list is created and ready for use.
- Parameters
  - **MAX_ITEMS:** size of the list.

Observer

Boolean isEmpty()

- Function
  - Determines whether list is empty.
- Precondition
  - List has been initialized.
- Postcondition
  - Function value = (list is empty).
- Parameters
  - None

Observer

Boolean isFull()

- Function
  - Determines whether list is full.
- Precondition
  - List has been initialized.
- Postcondition
  - Function value = (list is full).
- Parameters
  - None

Observer

int LengthIs()

- Function
  - Determines the number of elements in list.
- Precondition
  - List has been initialized.
- Postcondition
  - Function value = number of elements in list
- Parameters
  - None

Insertion

Insert(ItemType item, int position)

- Function
  - Adds item to list at a specific position
- Precondition
  - List has been initialized.
  - position is a valid index within the list bounds (0 ≤ position ≤ length).
- Postcondition
  - item is inserted at the specified position in the list, shifting subsequent elements to the right.
  - List = Original list + NewElement
- Parameters
  - **item:** The element to insert.
  - **position:** The index at which the item is inserted.

Insertion

Append(ItemType item)

- Function
  - Adds item to list at the end
- Precondition
  - List has been initialized.
  - List is not full
- Postcondition
  - item is added to the end of the list.
  - List = Original list + NewElement
- Parameters
  - **item:** The element to add.

Deletion

Remove/ Delete(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.
  - The elements to the right are shifted left.
  - List = Original list - DeletedElement
- Parameters
  - **item:** The element to delete.

Deletion

Clear / DeleteAll()

- Function
  - Deletes all the elements
- Precondition
  - List has been initialized.
  - The list is not empty.
- Postcondition
  - The list becomes empty.
- Parameters
  - None

Access (Retrieval)

ItemType Get(int position)

- Function
  - Retrieves the value of list element whose position matches (if present).
- Precondition
  - List has been initialized.
  - position is a valid index within the list bounds (0 ≤ position < length).
- Postcondition
  - Returns the element at the specified position.
  - List is unchanged.
- Parameters
  - **position:** The index of the element to retrieve.

Access (Retrieval)

int FindItem(ItemType item)

- Function
  - Retrieves the position of 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 return position, else -1/ Null.
  - List is unchanged.
- Parameters
  - **item:** The element to find.

Update

Replace(int position, ItemType item)

- Function
  - Modify the value of the given position (if present).
- Precondition
  - List has been initialized.
  - position is a valid index within the list bounds (0 ≤ position < length).
- Postcondition
  - Replaces the element at position with item.
- Parameters
  - **position:** The index of the element to replace.
  - **item:** The new element.

Traversal & Iteration

Iterator()

- Function
  - Iterates over all the elements of the list.
- Precondition
  - The list is not empty.
- Postcondition
  - Returns an iterator to traverse the elements of the list.
- Parameters
  - None

Traversal & Iteration

ForEach(operation)

- Function
  - Applies operation to each element of the list.
- Precondition
  - The list is not empty.
- Postcondition
  - Applies a given operation to each element in the list.
  - All elements will have the operation applied sequentially.
- Parameters:
  - A function or operation that takes each element of the list as input.

Reversal

Reverse()

- Function
  - Reverse the order of the elements
- Precondition
  - The list is not empty.
- Postcondition
  - Reverses the order of elements in the list.
- Parameters
  - None

Implementation of List ADT

Implementation of List

- A list can be created by two possible ways:
  - **Static:** Array based Implementation
    - The elements are stored in contiguous cells of an array.
    - With this representation a list is easily traversed and new elements can be appended readily to the tail of the list.
    - Inserting an element into the middle of the list, however, requires shifting all following elements one place over in the array to make room for the new element.
    - Similarly, deleting any element except the last also requires shifting elements to close up the gap.

Implementation of List

- A list can be created by two possible ways:
  - **Dynamic:** Pointer based Implementation
    - This implementation of list uses pointers to link successive list elements.
    - This implementation frees us from using contiguous memory for storing a list and hence from shifting elements to make room for new elements or to close up gaps created by deleted elements.
    - However, one price we pay is extra space for pointers.
    - In this representation, a list is made up of cells, each cell consisting of an element of the list and a pointer to the next cell on the list.

Sequential/Array-based List ADT

- In this implementation list elements are stored sequentially, in adjacent slots in an array.
- Array-based list implementations are so common that many people refer to an “array” of data when they really mean a “list”.
  - Of course , this is not correct.

C++ Implementation of List ADT

```
class UnsortedType
{
public:
  UnsortedType();
  bool IsFull() const;
  int LengthIs() const;
  void RetrieveItem(ItemType& item, bool& found);
  void InsertItem(ItemType item);
  void DeleteItem(ItemType item);
  void ResetList();
  void GetNextItem(ItemType& item);
private:
  int length;
  ItemType info[MAX_ITEMS];
  int currentPos;
};
```

C++ Implementation of List ADT

```
UnsortedType::UnsortedType()
{
  length = 0;
}
```

```
int UnsortedType::LengthIs() const
{
  return length;
}
```

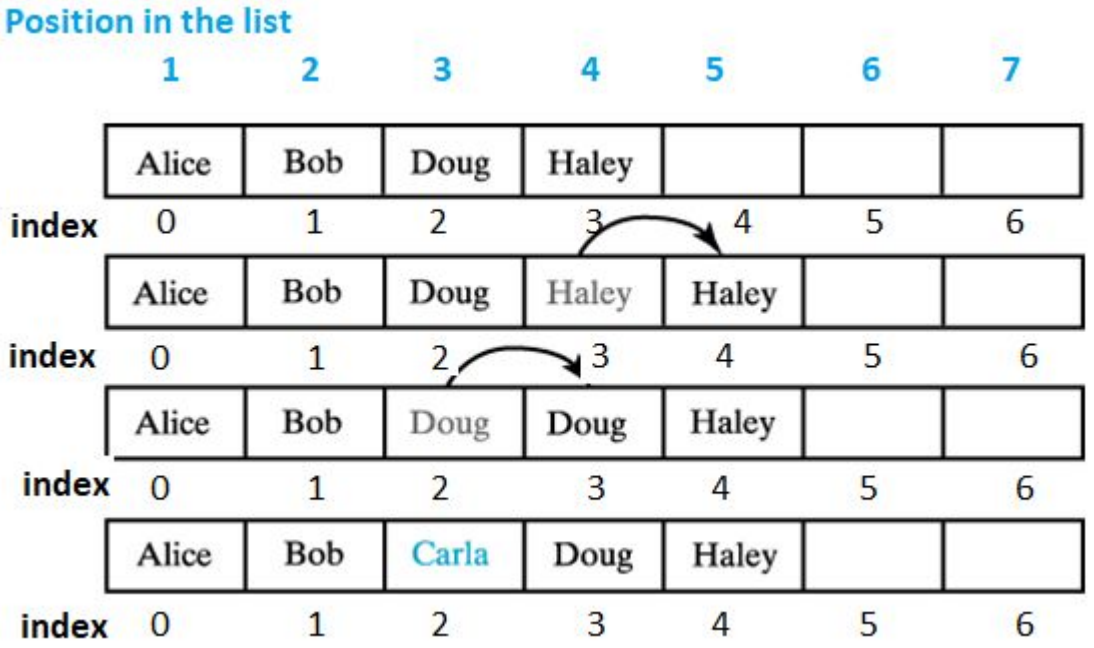
```
bool UnsortedType::IsFull() const
{
  return (length == MAX_ITEMS);
}
```

```
void UnsortedType::ResetList()
// Post: currentPos has been initialized.
{
  currentPos = -1;
}
```

Insertion Operation

- Insertion at the Tail End
  - We may add a new item at the end of the list
  - Assign new value at end
  - Increment length of list

- Insertion at a Particular Position
  - A user may insert an item at a particular position in a list.
  - Requires a utility method, **makeRoom()** to shift elements towards right.
  - Then the new value is inserted at the vacated position.



Insertion at the Tail End

```
void InsertAtEnd(int value){
    if(!isFull()){
        array[length]=value;
        length++;
    }
    else{
        cout<<"List is Full. Insertion not possible"<<endl;

    }
}
```

List of length 5

Position in the list	1	2	3	4	5	
Index	0	1	2	3	4	5
Value	8	20	33	44	48	

Search Operation

- Searches a value in a non-empty list
- If found, return the position in the list
  - Different from the index in the array
- If not found, return null or -1.

List of length 5

Position in the list	1	2	3	4	5	
Index	0	1	2	3	4	5
Value	8	20	33	44	48	

```
int Search(int value){
    if(!isEmpty()){
        int index=0;
        while(index<length){
            if(value==array[index])
                return (index+1);
            index++;
        }
    }
    else{
        cout<<"List is Empty."<<endl;}
    return -1;
}
```

Brain Teasers

- Suppose that myList is a list that contains the five entries a b c d e.
  - What does myList contain after executing myList.insertAtPosition('w', 5)?

- Starting with the original five entries, what does myList contain after executing myList.insertAtPosition('w', 6)?
  - Which one of the above operations involves elements to be shifted in the array.

0	1	2	3	4	5	6
a	b	c	d	e		

Brain Teasers

- If myList is a list of five entries, each of the following statements adds a new entry to the end of the list:
  - myList.InsertAtEnd(newEntry);
  - myList.InsertAtPosition(newEntry , 6); // 6 is position of the new item in the list.

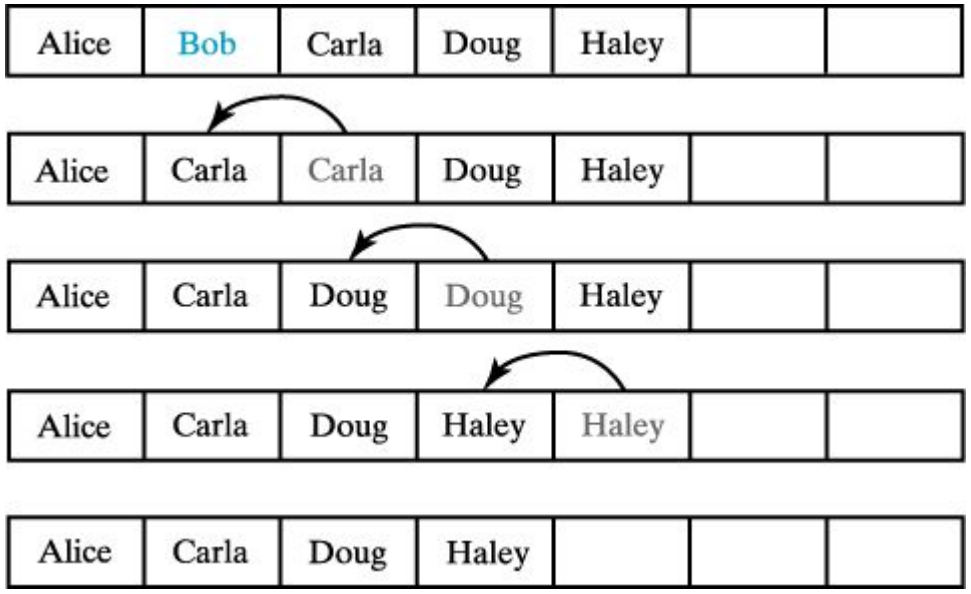
0	1	2	3	4	5	6
22	33	45	47	66		

- Which one requires fewer operations?

Deletion Operation

- Must shift existing entries to avoid gap in the array
  - Except when removing last entry

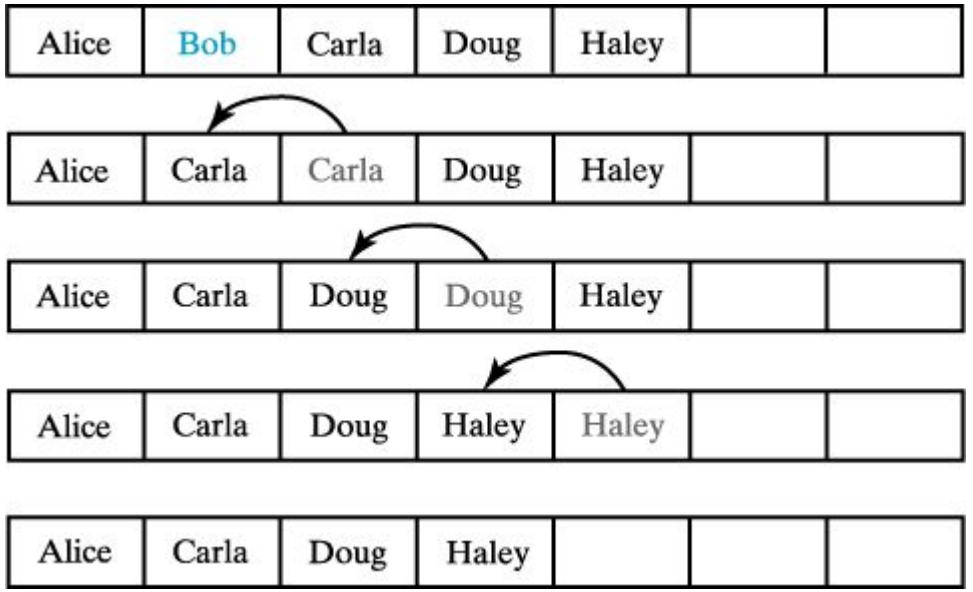
- Method must also handle error situation
  - When position specified in the remove is invalid
  - When remove() is called and the list is empty
  - Invalid call returns a null value



Removing Bob by shifting array entries.

Deletion Operation

```
void Delete(int value){
    int index= Search(value)-1;
    if(index==-1){
        cout<<"value not found. It cannot be deleted"<<endl;
        return;
    }
    FillRoom(index);
    length--;
}

  void FillRoom(int Index){
      if(Index<length-1){
          for(int x=Index; x<length-1; x++){
              array[x]=array[x+1];
          }
      }

  }
```



Removing Bob by shifting array entries.

Sorted List ADT

Sorted List ADT

Structure

- The list elements are of **ItemType**.
- The list elements have a **semantic** relationship.
- 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.

Insertion

Insert(ItemType item)

- Function
  - Adds item to list

- Precondition
  - List has been initialized.
  - List is not full
  - List is sorted

- Postcondition
  - item is in list.
  - List is still sorted.
  - List = Original list + NewElement

- Parameters
  - **item:** The element to insert.

Deletion

Remove/ Delete(ItemType item)

- Function
  - Deletes the element whose key matches item's key.
- Precondition
  - List has been initialized.
  - List is sorted
  - 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.
  - List = Original list - DeletedElement
- Parameters
  - **item:** The element to delete.

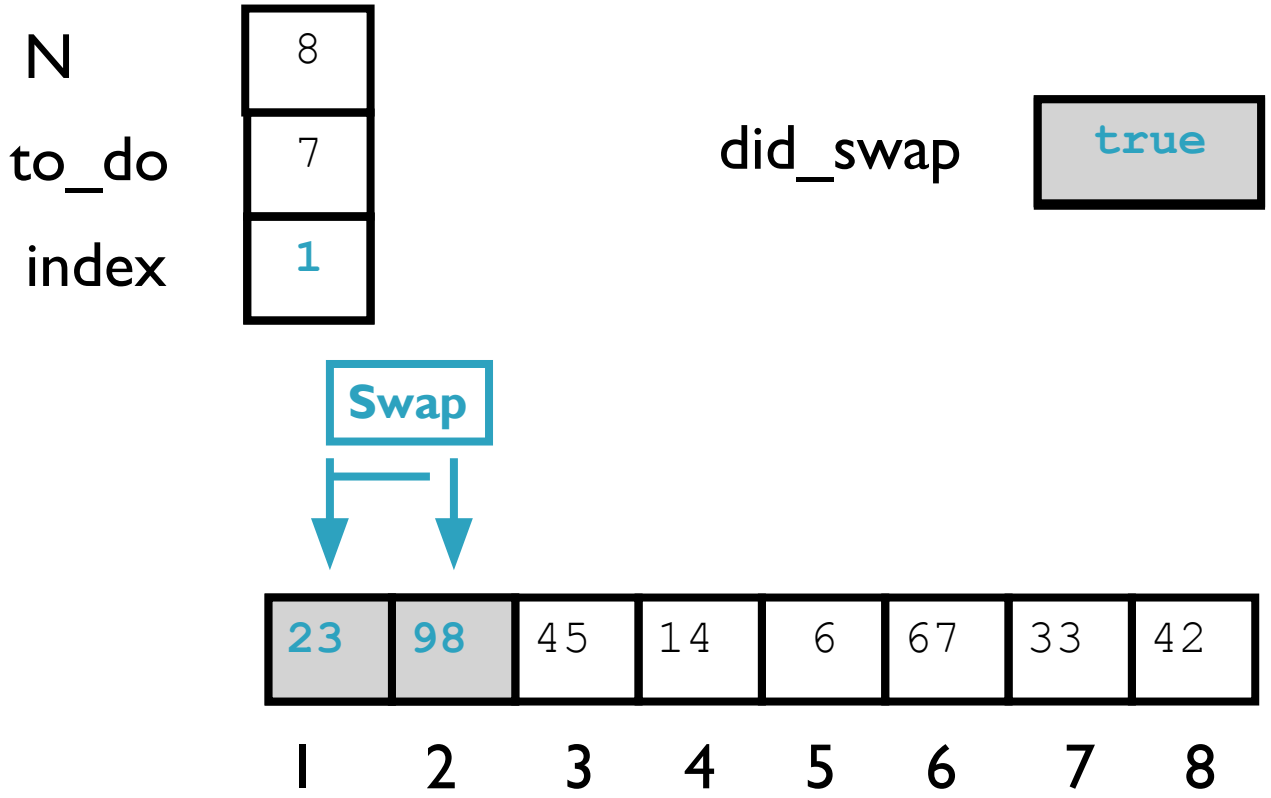
Insertion

```
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];
```

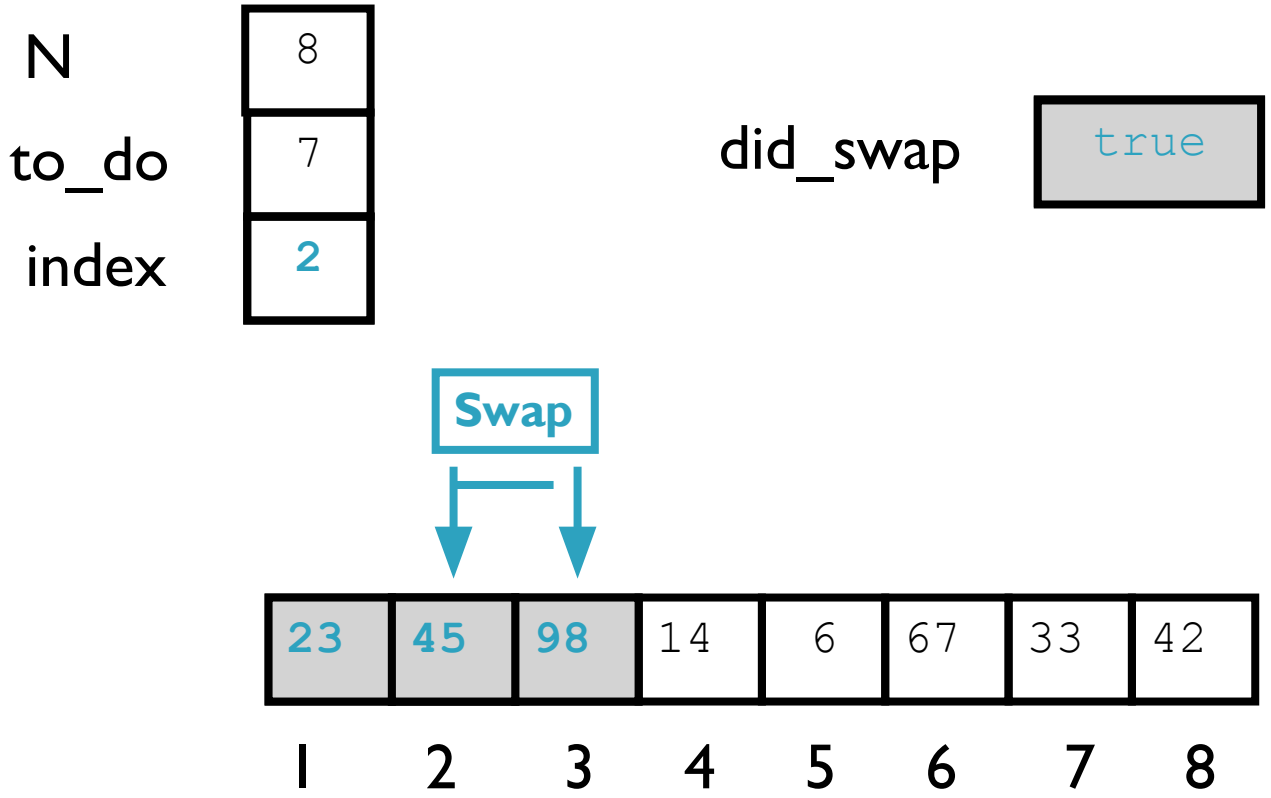
Deletion

```
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];
  length--;
}
```

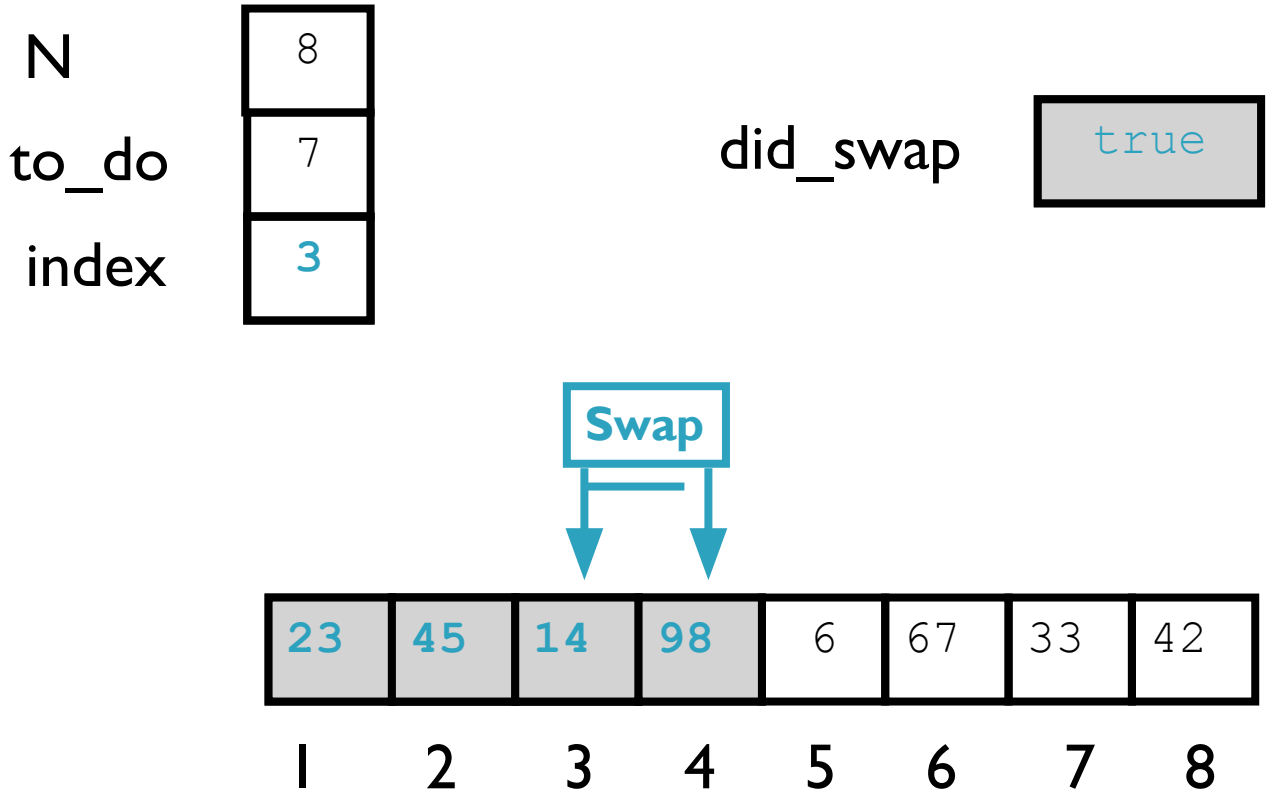
Example :1ST Iteration



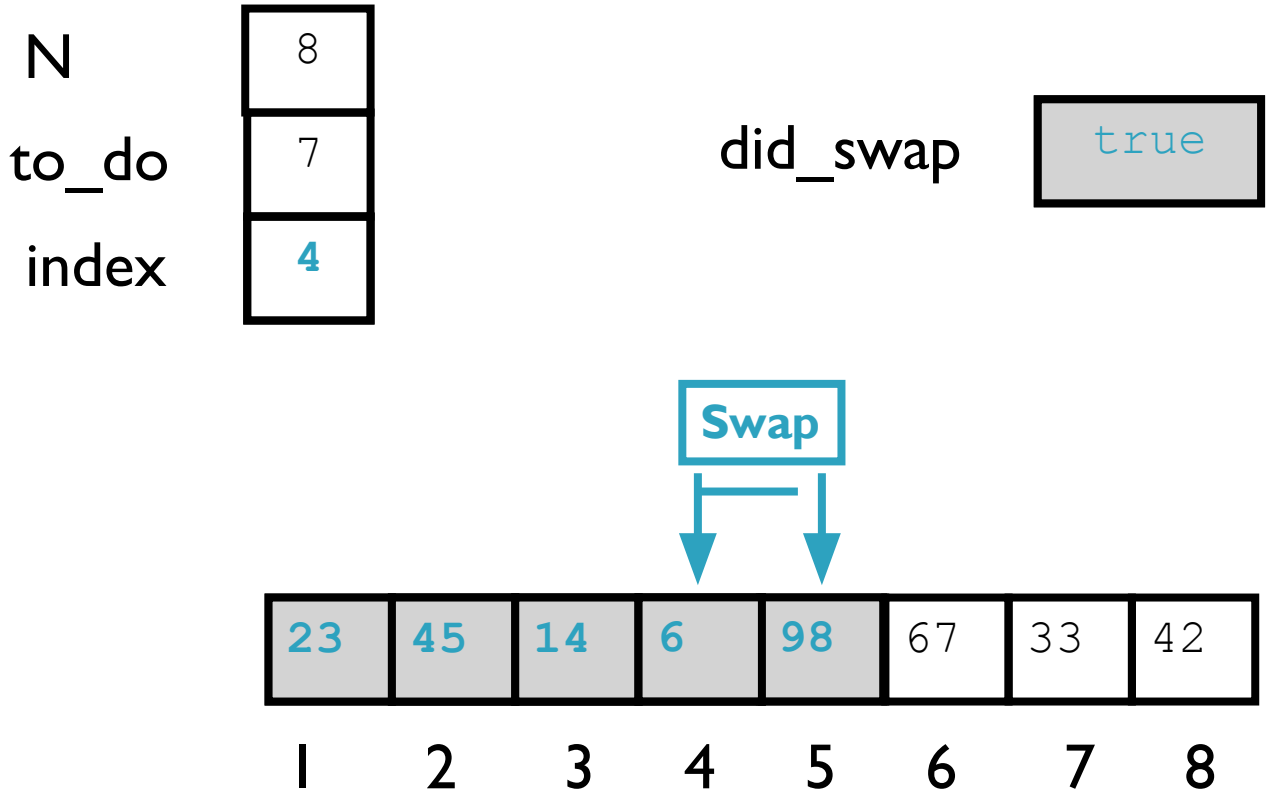
Example :1ST Iteration



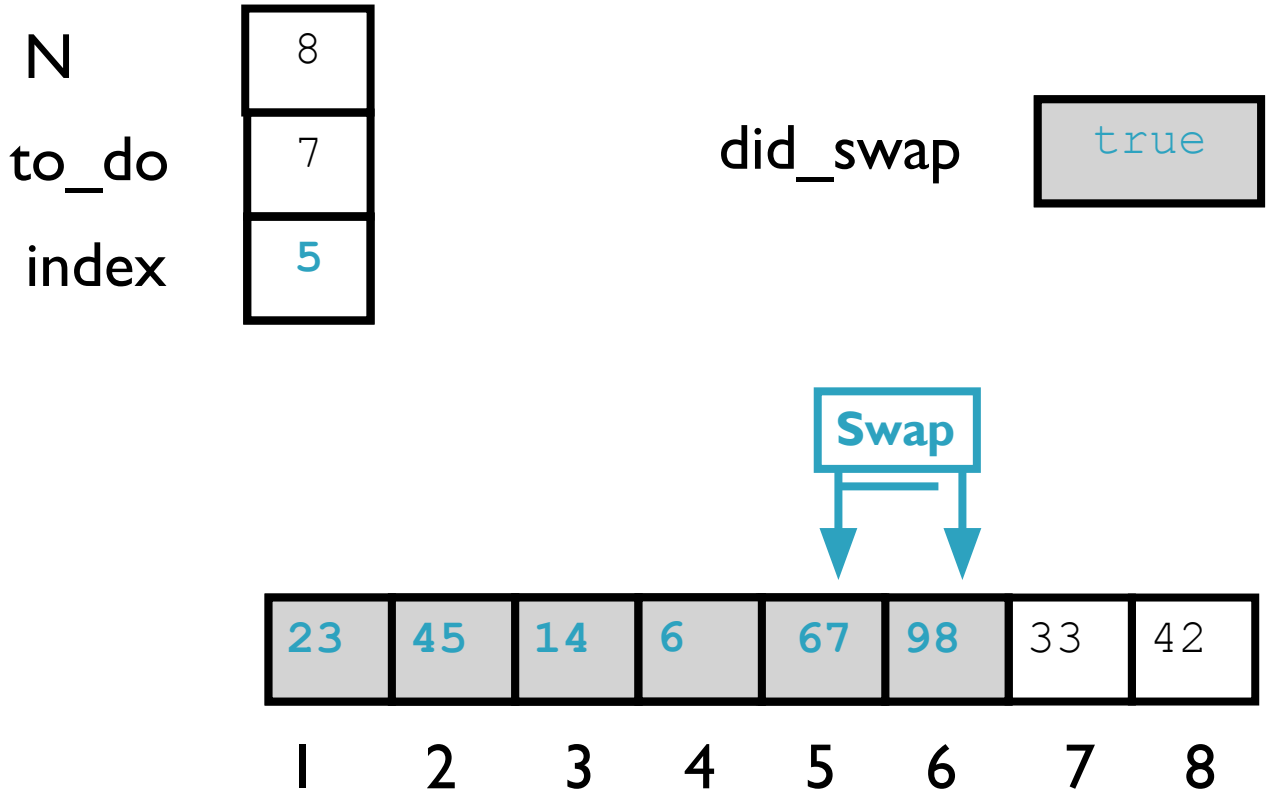
Example: 1ST Iteration



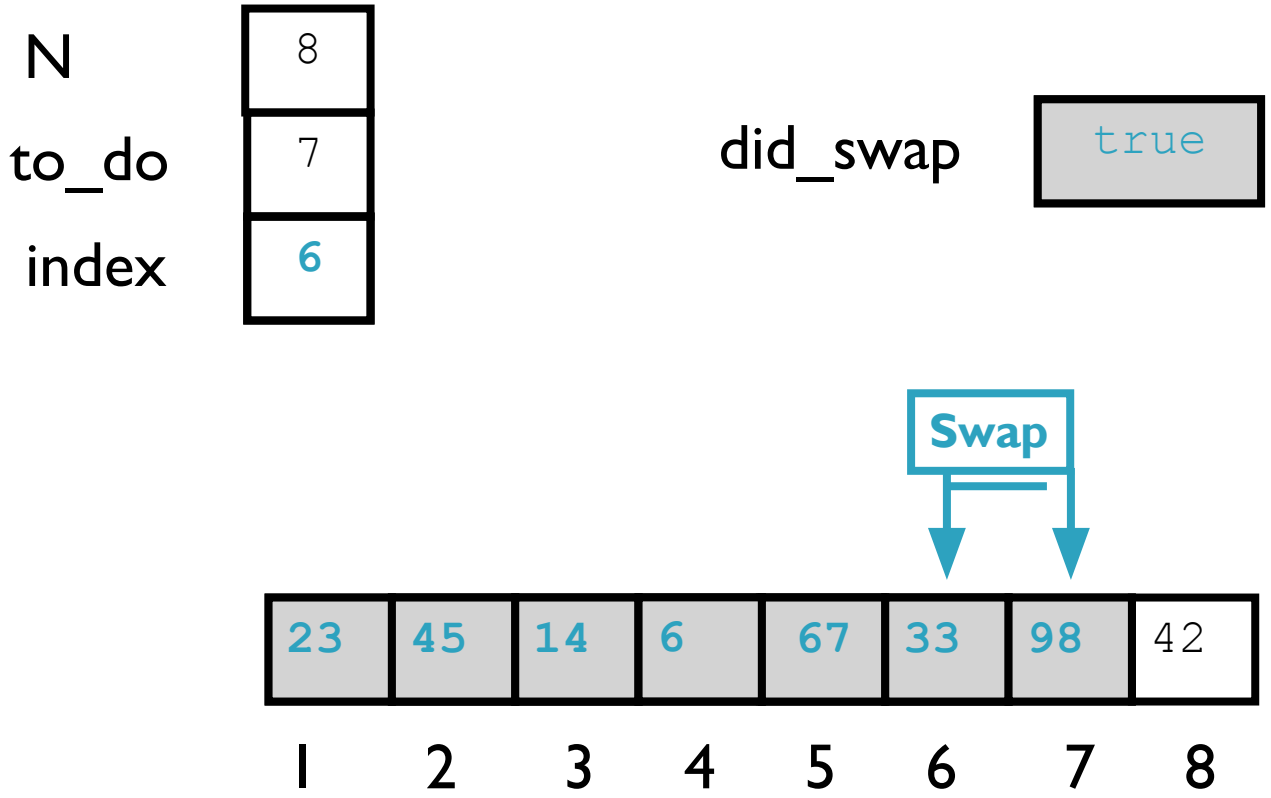
Example: 1ST Iteration



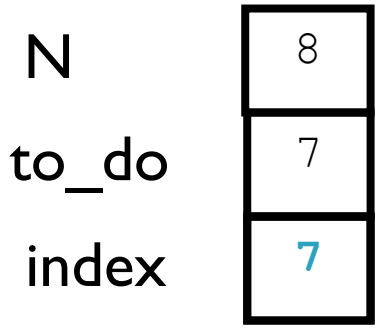
Example: 1ST Iteration



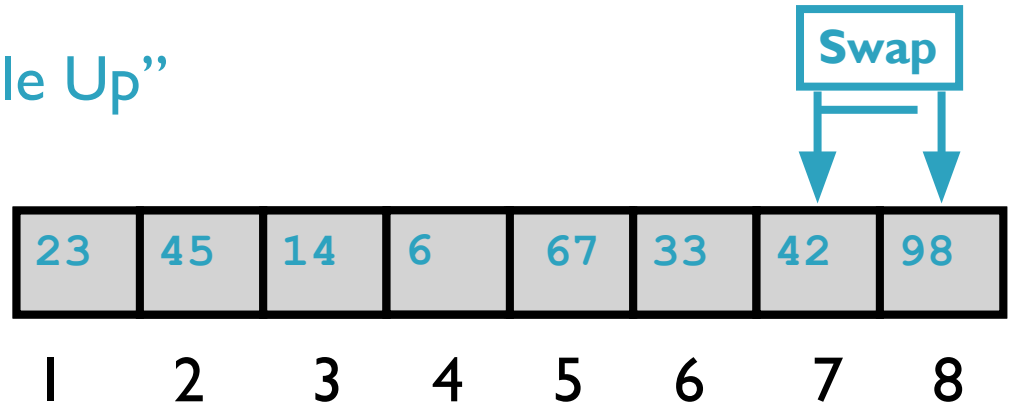
Example: 1ST Iteration



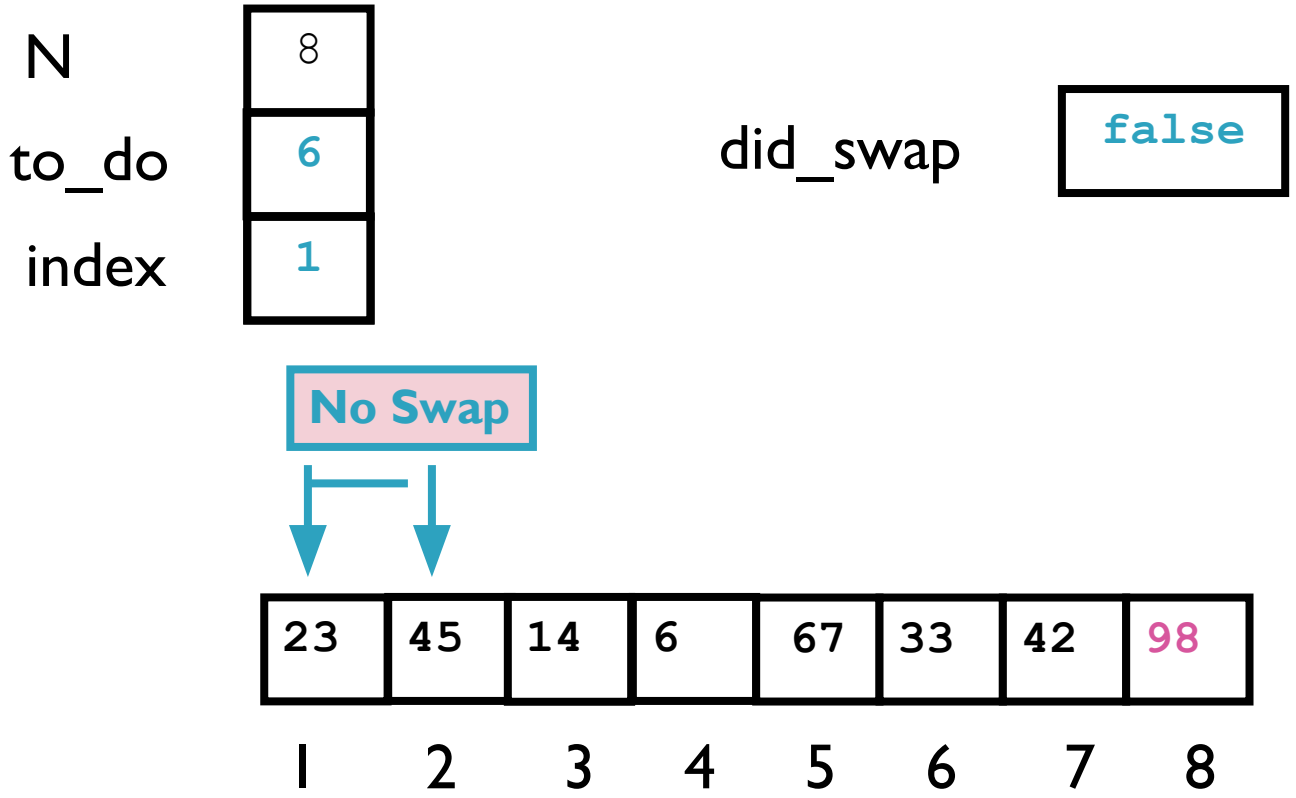
Example: 1ST Iteration



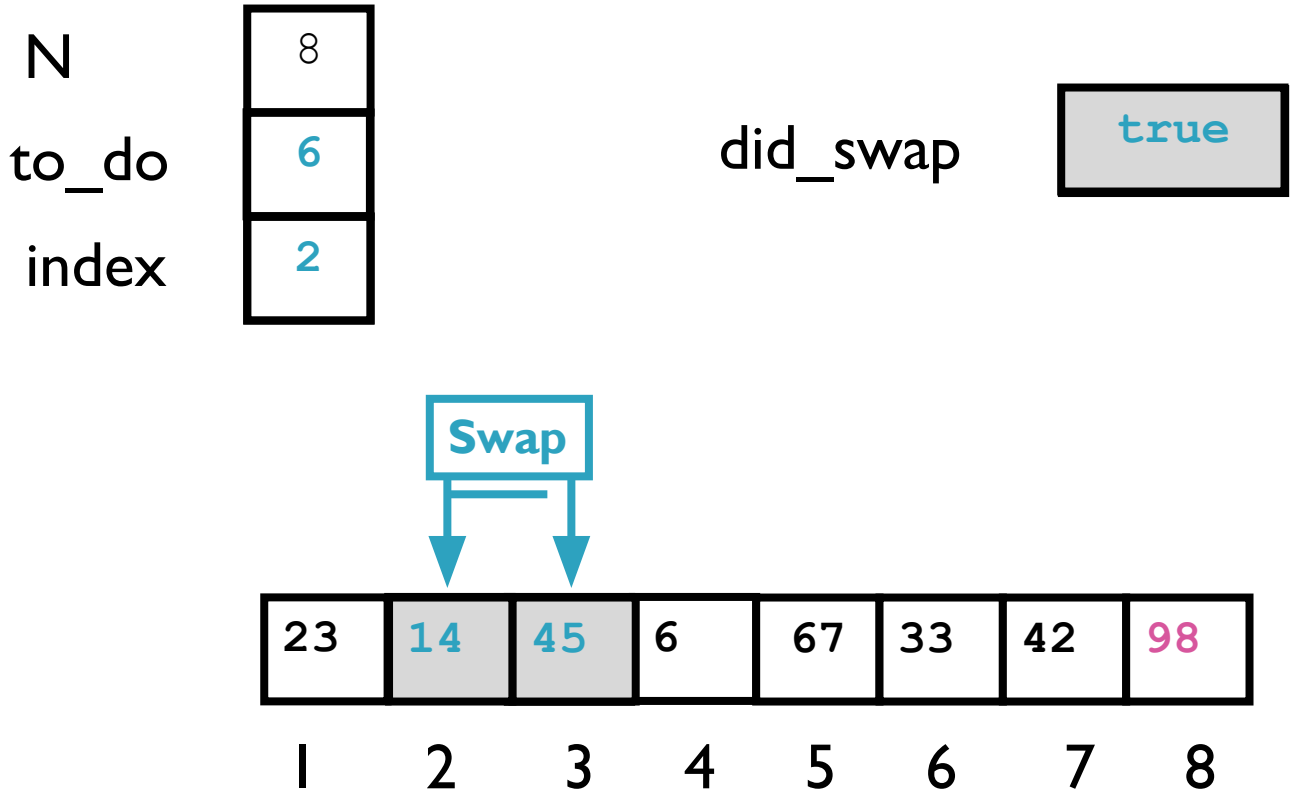
Finished first "Bubble Up"



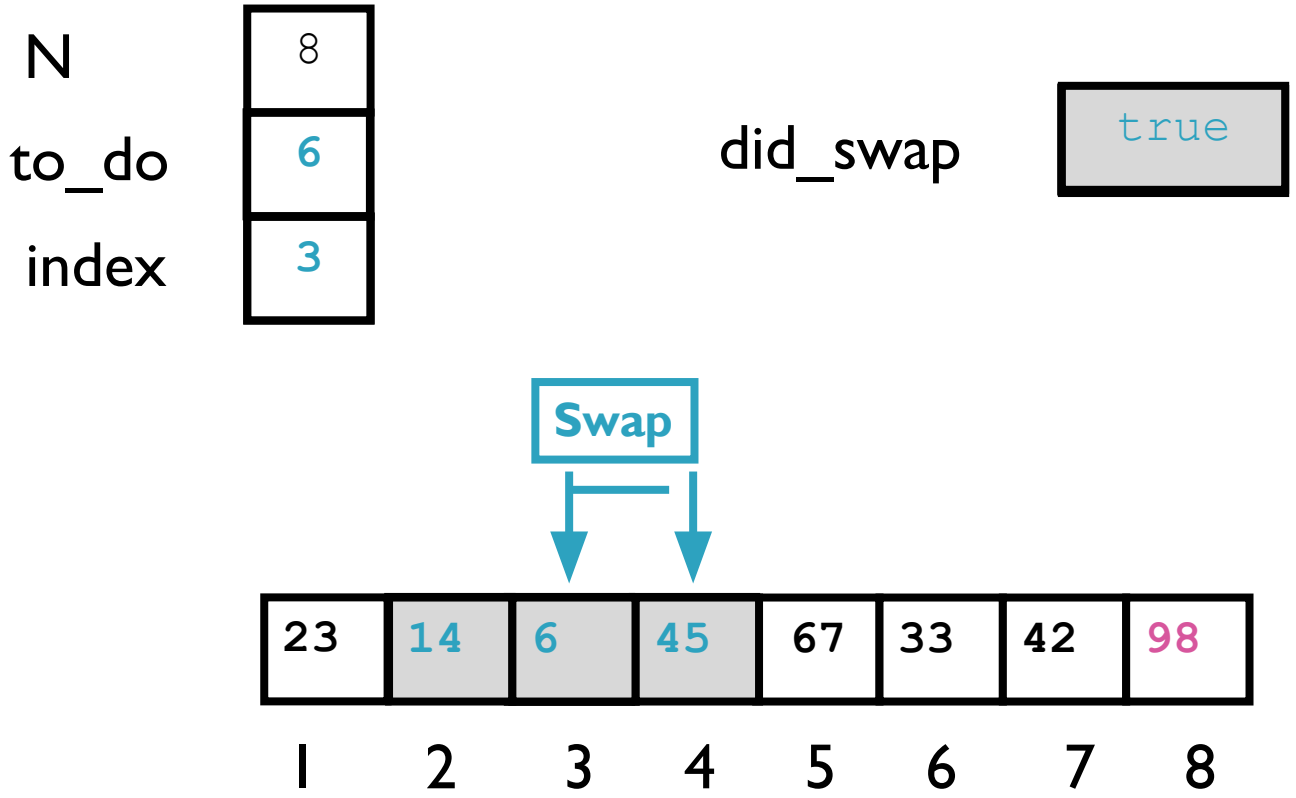
Example: 2ND Iteration



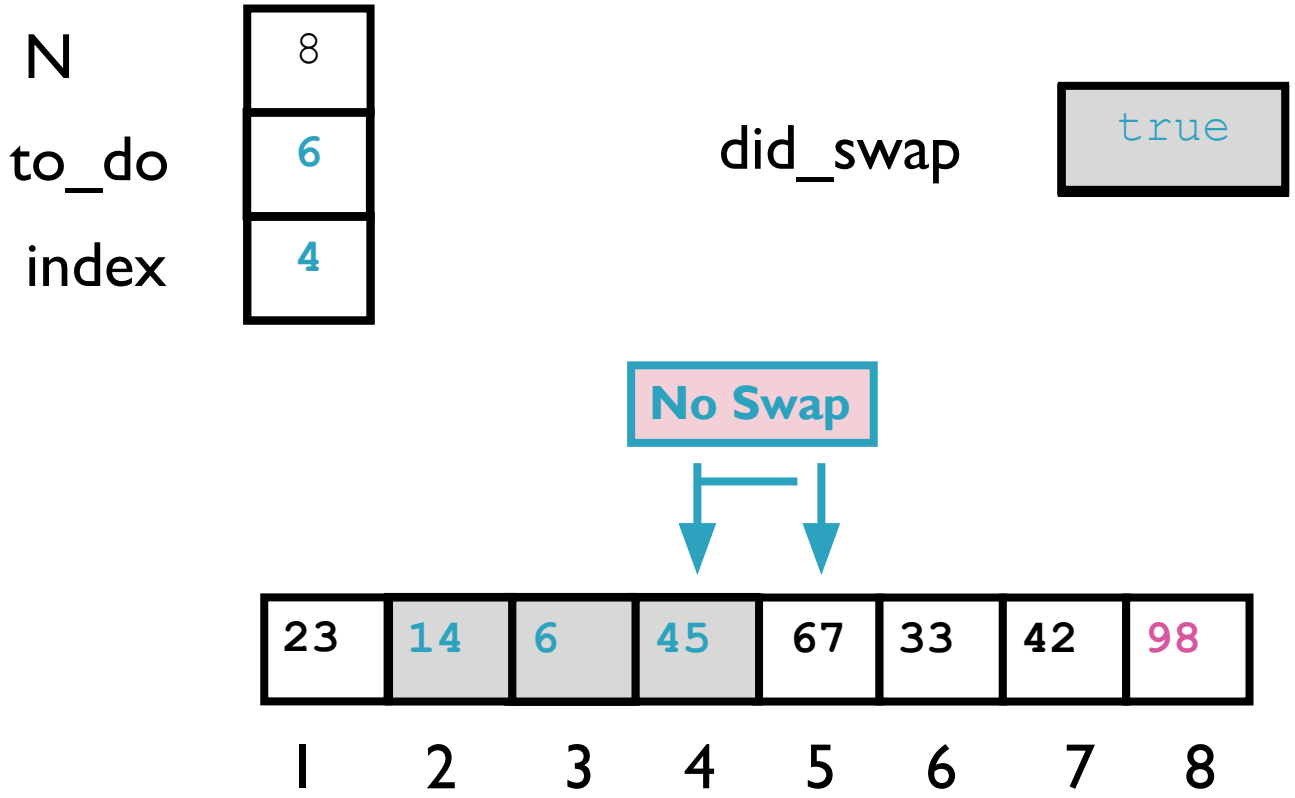
Example: 2ND Iteration



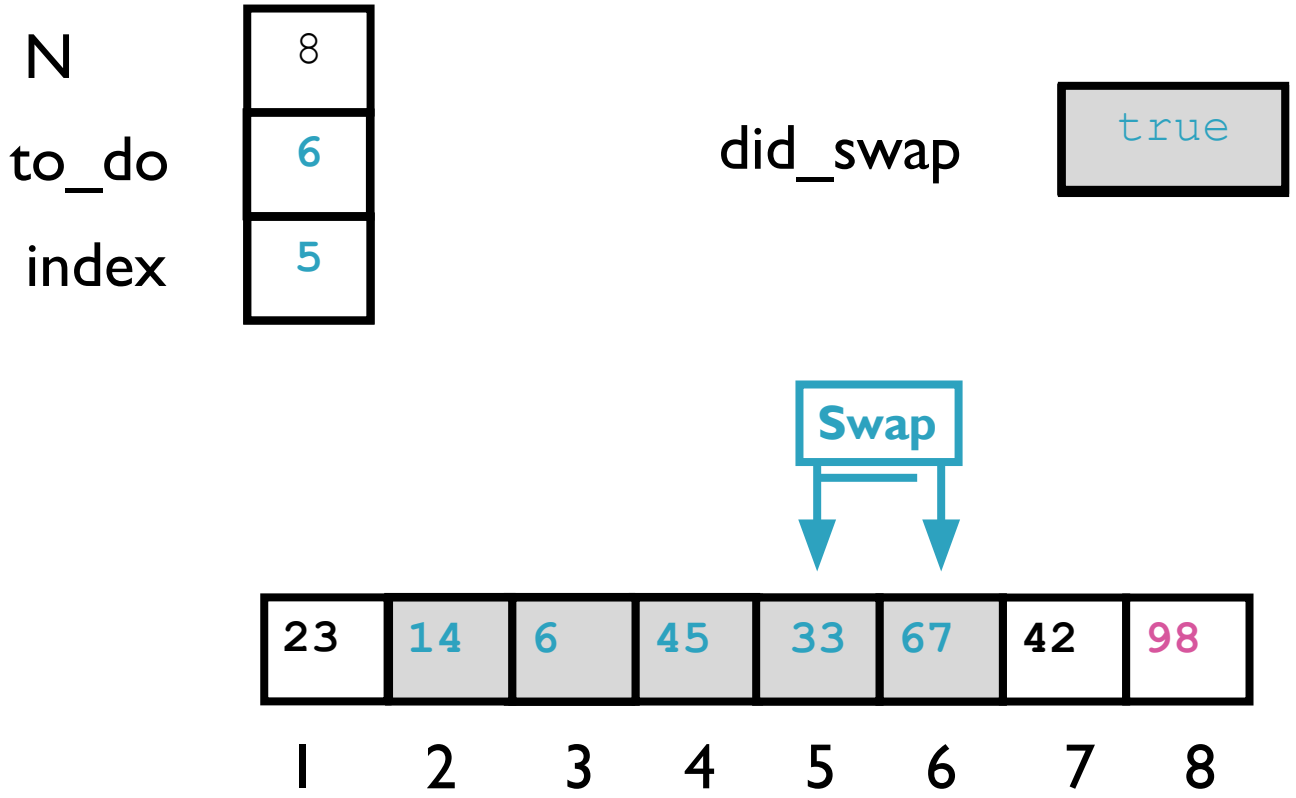
Example: 2ND Iteration



Example: 2ND Iteration

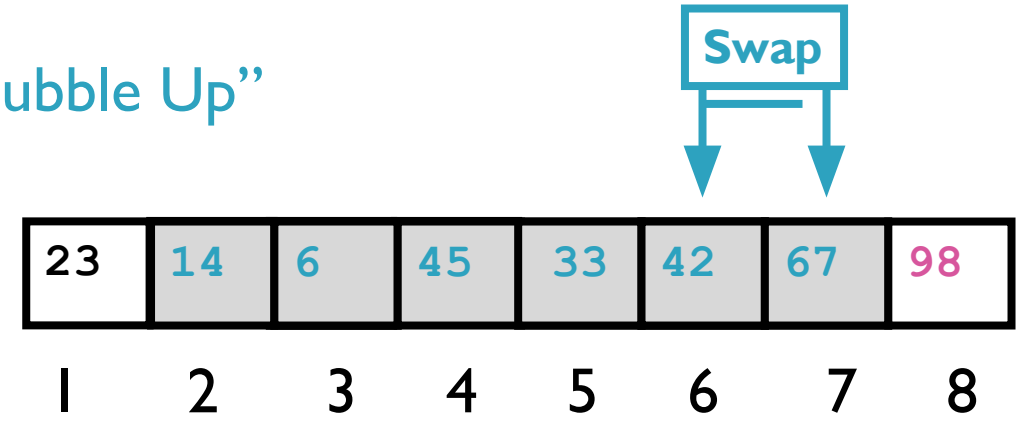


Example: 2ND Iteration

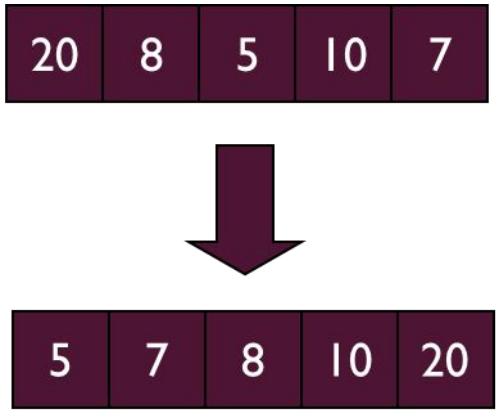


Example: 2ND Iteration

Finished second "Bubble Up"



Sorting




```
void bubbleSort() {
    for (int i = 0; i < length - 1; i++) {
        // Last i elements are already sorted
        for (int j = 0; j < length - i - 1; j++) {
            // Swap if the element found is greater than the next
  element
            if (list[j] > list[j + 1]) {
                int temp = list[j];
                list[j] = list[j + 1];
                list[j + 1] = temp;
            }}}}
```

Sorting

- “Bubble Up” algorithm will move largest value to its correct location (to the right)
- Repeat “Bubble Up” until all elements are correctly placed:
  - Maximum of length-1 times
  - Can finish early if no swapping occurs
- We reduce the number of elements we compare each time one is correctly placed

Bubble sort analysis

- What is the time complexity of this algorithm?
  - Worst case > Average case > Best case
  - Each iteration compares all the adjacent elements, swapping them if necessary

first iteration	N	+
second iteration	N-1	+
…		
last iteration	1	
Total	$N(1+N)/2 = O(N^2)$	

Searching revisited

- A question you should always ask when selecting a search algorithm is "How fast does the search have to be?"
  - The reason is that, in general, the faster the algorithm is, the more complex it is.

- **Bottom line:** you don't always need to use or should use the fastest algorithm.

- Let's explore the following search algorithms, keeping speed in mind.
  - Sequential (linear) search
  - Binary search

Linear Search

- A search traverses the collection until
  - The desired element is found
  - Or the collection is exhausted

- If the collection is **ordered**, I might not have to look at all elements
  - I can stop looking when I know the **element cannot be in the collection**.

Linear Search: Sorted vs. Unsorted List

- If the **order** was ascending alphabetical on customer's last names, how would the search for John Adams on the ordered list compare with the search on the unordered list?

- Unordered list
  - if John Adams was in the list?
  - if John Adams was not in the list?

- Ordered list
  - if John Adams was in the list?
  - if John Adams was not in the list?

Linear Search: Sorted vs. Unsorted List

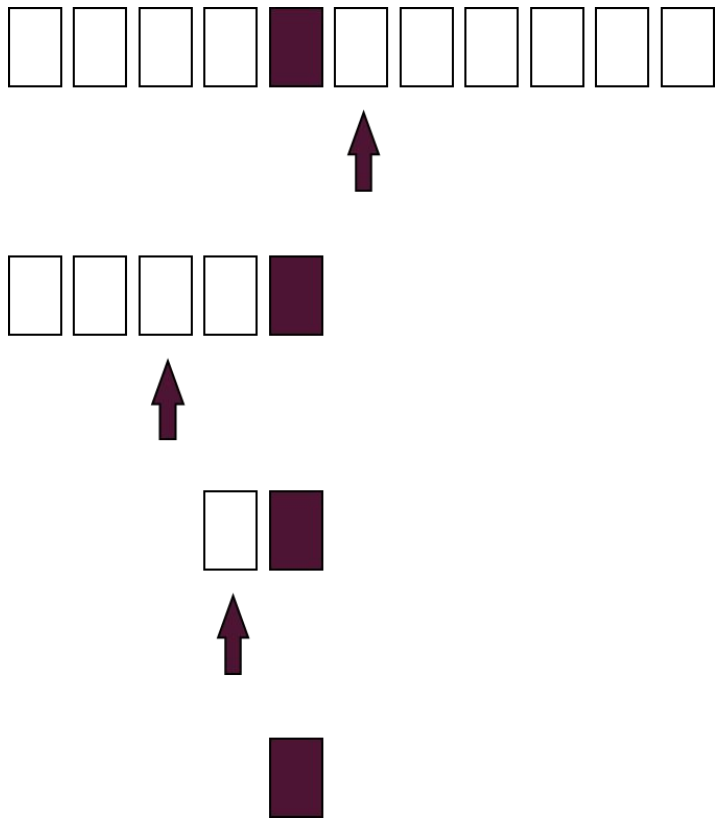
- **Observation**
  - The search is faster on an ordered list only when the item being searched for is not in the list.
  - Also, keep in mind that the list has to first be placed in order for the ordered search.

- **Conclusion**
  - The efficiency of these algorithms is roughly the same.
  - So, if we need a faster search, we need a completely different algorithm.

- How else could we search an ordered file?

BINARY SEARCH

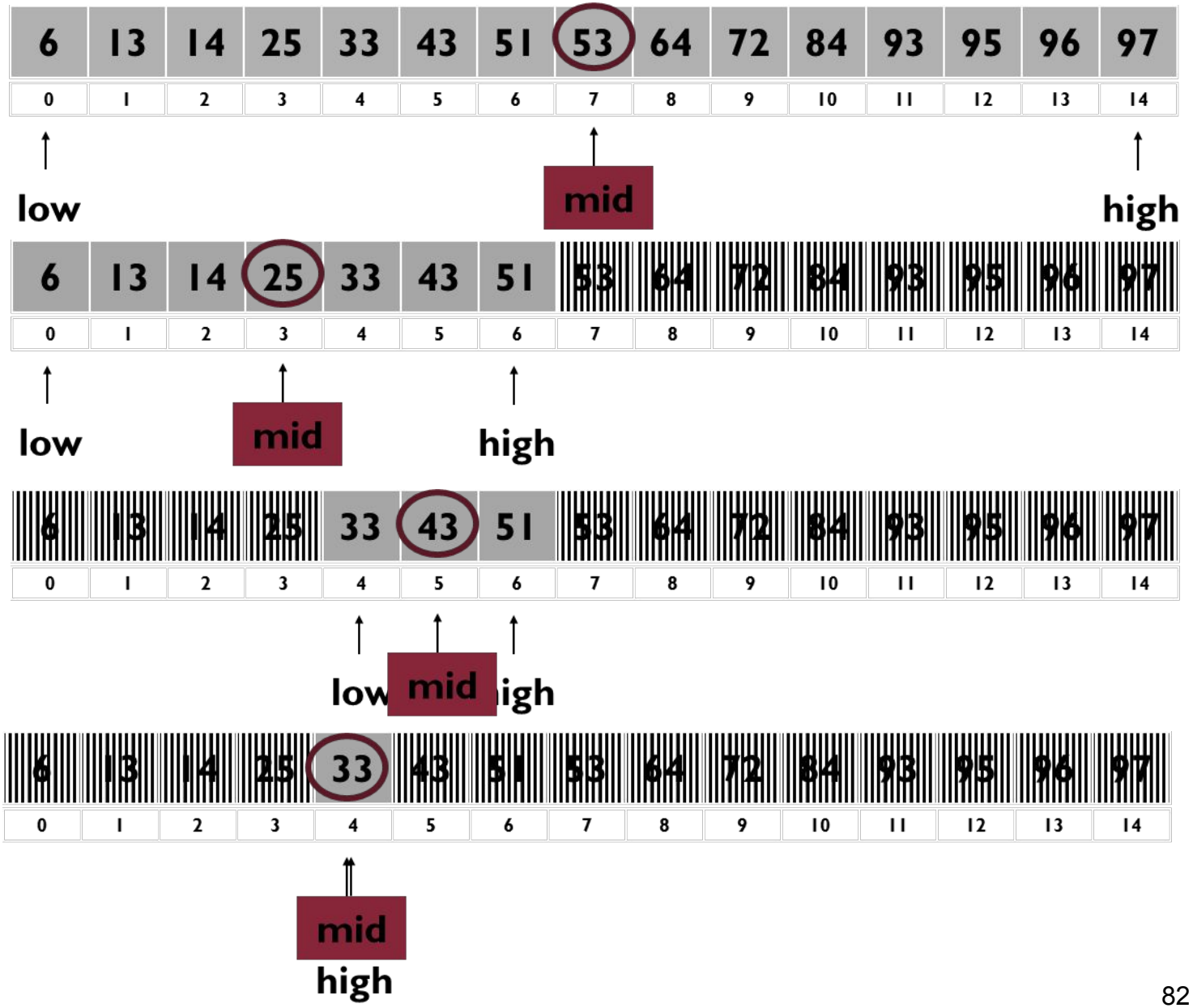
- If we have an ordered list and we know upper bound of list we can use a different strategy.
- The binary search gets its name because the algorithm continually divides the list into two parts.



Always look at the center value.
Each time you get to discard half of the remaining list.
Is this fast ?

Binary search for 33

Algorithm Invariant: maintains list[low]<=key<=list[high]



Binary Search

```
int binarySearch(int low, int high, int key) {
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == key) // Check if the key is present at mid
            return mid;
        if (arr[mid] < key)  // If key is greater, ignore the left half
            low = mid + 1;
        else     // If key is smaller, ignore the right half
            high = mid - 1;
    }
    return -1;  // Return -1 if the key is not found
}
```

How fast is binary search

- How about the worst case for a list with 32 items ?
  - 1st try - list has 16 items
  - 2nd try - list has 8 items
  - 3rd try - list has 4 items
  - 4th try - list has 2 items
  - 5th try - list has 1 item

List has 250 items
1st try - 125 items
2nd try - 63 items
3rd try - 32 items
4th try - 16 items
5th try - 8 items
6th try - 4 items
7th try - 2 items
8th try - 1 item

List has 512 items
1st try - 256 items
2nd try - 128 items
3rd try - 64 items
4th try - 32 items
5th try - 16 items
6th try - 8 items
7th try - 4 items
8th try - 2 items
9th try - 1 item

What's the pattern

- List of 11 took 4 tries
- List of 32 took 5 tries
- List of 250 took 8 tries
- List of 512 took 9 tries
  - $32 = 2^5$ and $512 = 2^9$
  - $8 < 11 < 16$ $\quad 2^3 < 11 < 2^4$
  - $128 < 250 < 256$ $\quad 2^7 < 250 < 2^8$

- How long (worst case) will it take to find an item in a list 30,000 items long?

$2^{10} = 1024$ $\quad 2^{13} = 8192$

$2^{11} = 2048$ $\quad 2^{14} = 16384$

$2^{12} = 4096$ $\quad 2^{15} = 32768$

- So, it will take only 15 tries!

Efficiency

We say that the binary search algorithm runs in **log2 n** time. (Also written as lg n) **Lg n** means the log to the base 2 of some value of n. There are no algorithms that run faster than lg n time.

$8 = 2^3$ $\quad \lg 8 = 3$ $\quad 16 = 2^4$ $\quad \lg 16 = 4$

Time complexity of LIST ADT-Array based

Example of algorithm	Complexity
Traversal	O(n)
Retrieval / Access	O(1)
Insertion At End	O(1)
Insertion At middle/Start	O(n)
Deletion At End	O(1)
Deletion At middle/Start	O(n)
Sorting (Bubble Sort)	$O(n^2)$
Linear Search	O(n)
Binary Search	$Log_2\ n$

Summary

- In this lecture we have been discussed:
  - Concept of abstract data types
  - List ADT with array based implementation
  - Fundamental operations on List ADT
  - Sorting and searching
  - Concept of Big-O notation