CSE 105:
Data Structures and Algorithms-I
(Part 2)

Instructor

Dr Md Monirul Islam

Text Books:

- Introduction to Algorithms
  - Thomas H. Cormen and others

- Data Structures and Algorithm analysis (either one of C++ / Java Version)
  - Clifford A Shaffer

Contents:

- lists (array and linked)
- stacks, queues
- trees and tree traversals; graphs and graph representations
- Heaps and priority queue
- binary search trees;
- Graph traversals: DFS, BFS, applications of DFS and BFS;

Some Preliminaries

- **Type:** Collection of values
  - **Example:** Boolean type consists of **true** and **false**
    Integer type consists of **whole numbers**

Some Preliminaries

- **Type:** Collection of values
  - **Example:** Boolean type consists of **true** and **false**
    Integer type consists of **whole numbers**

- **Simple Type and Composite Type**
  - **Simple/Primitive/System defined Type:**
    Examples: Boolean and integer
    No sub-parts
  - **Composite/Aggregate/User defined Type:**
    Examples: structures you learned in C
    consists of multiple simple and/or composite types
    have sub-parts

Some Preliminaries

- **Type:** Collection of values
  - **Example:** Boolean type consists of **true** and **false**
    Integer type consists of **whole numbers**

- **Data Item:** a particular member of a type
  - **Examples:** *true* **is a member of** *Boolean* type
    10 **is a member of** *integer*

Some Preliminaries

- **Data Type:** *type* along with *a set of operations* to manipulate the type
  - **Example:** *Integer type* and operations like *addition, subtraction*, etc

Some Preliminaries

- **Data Type:** *type* along with *a set of operations* to manipulate the type
  - **Example:** *Integer type* and operations like *addition, subtraction*, etc

What about user defined type?

They are also defined for particular tasks.

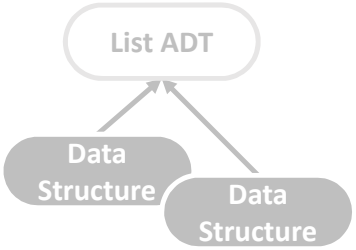
Some Preliminaries

- **Abstract Data Type (ADT):**
  - *(user defined) type* along with *a set of operations* to manipulate the type
  - Implementation details NOT specified in definition
  - Each operation specifies only inputs and/or outputs

Some Preliminaries

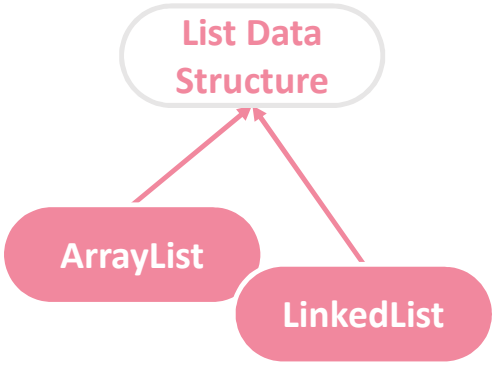
- **Data Structure:**
  - Physical implementation of an ADT
  - Each operation is implemented by a subroutine.
  - Example: In C++, a class is an implementation of an ADT

ADT / Data Structure



Does not specify

- implementation details
- even list of what



- Can be implemented by both resizable array or linked list
- specifies implementation details

List

- A list is an ordered sequence of data items known as **elements**.
  - Variable/finite size
  - Elements can be added to or removed from any position

- Examples:
  - CT marks < 19, 20, 19, 18, . . . >
  - Your home districts < ‘Dhaka’, ‘Brahman Baria’, ‘Chittagong’, . . . >
  - .
  - .
  - .

  - And many more

List

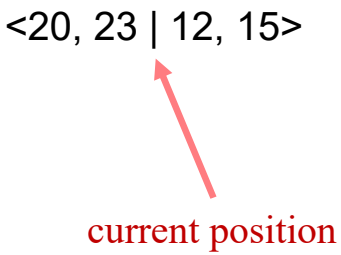
- Important concept: every list element has a position.

- Notation: $<a_0, a_1, \ldots, a_{n-1}>$
- Empty list: no elements (< >)
- Length of the list: number of elements
- head: beginning of the list
- tail: end of the list
- Element position: k >=0 is the position of element a_k.

List ADT

- What operation we want?

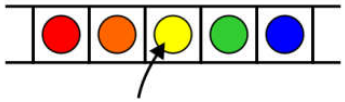
- Assume we have a current position in $<a_0, a_1, \ldots, a_{n-1}>$
- Operation will act relative to current position



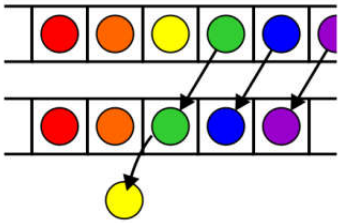
List ADT: Possible Operations

Operations at the k^{th} entry of the list include:

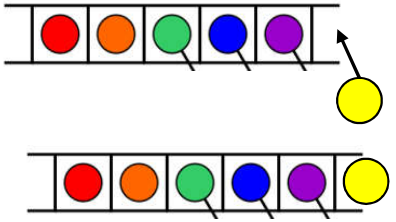
Access to the object



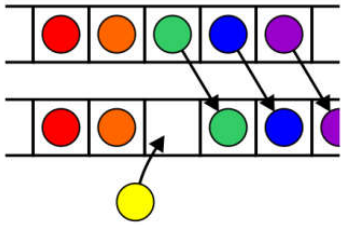
Remove an object



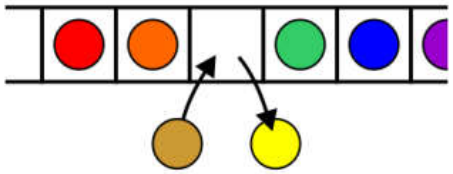
Append



Insertion of a new object

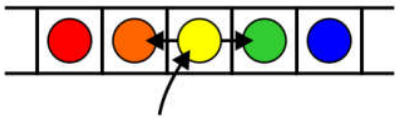


Replacement of the object



List ADT: Possible Operations

Given access to the k^{th} object, gain access to either the previous or next object



Given two abstract lists, we may want more complex operations

- Concatenate the two lists
- Determine if one is a sub-list of the other

List ADT: Possible Operations

1. **clear();**
2. **insert(item);** //insert at current position
3. **append(item);**
4. **remove();** //insert at current position
5. **moveToStart();**
6. **moveToEnd();**
7. **prev();**
8. **next();**
9. **length();**
10. **currPos();**
11. **moveToPos(int pos);**
12. **getValue();**

List ADT: Possible Operations

Function	Parameter	Return Value	After function execution
[before function execution]			<20, 23 | 12, 15>
clear()	-		<>
insert(item)	19		<20, 23 | 19, 12, 15>
append(item)	19		<20, 23 | 12, 15,19>
remove()	-	12	<20, 23 | 15>
moveToStart()	-		< | 20, 23, 12, 15>
moveToEnd()	-		<20, 23, 12 |15>
prev()	-		<20 | 23, 12, 15>
next()	-		<20, 23 , 12 | 15>
length()	-	4	<20, 23 | 12, 15>
currPos()	-	2	<20, 23 | 12, 15>
moveToPos(int pos)	1		<20 | 23, 12, 15>
getValue()	-	12	<20, 23 | 12, 15>

List ADT: Operation Explained

Assume we use class template of C++ to define a List ADT

```
template <typename E> class List { // List ADT
private:
.
.
public:
List() {} // Default constructor
virtual ~List() {} // Base destructor
// Insert an element at the current location.
// item: The element to be inserted
virtual void insert(const E& item) = 0;
// Append an element at the end of the list.
// item: The element to be appended.
virtual void append(const E& item) = 0;
.
.
}
```

Objectives:

- Make it very generic
- Hide implementation details

List ADT: Operation Explained

List: <12 | 32, 15>

L.insert(99);

Result: <12 | 99, 32, 15>

Iterate through the whole list:

```
for (L.moveToStart();  L.currPos()<L.length();    L.next()) {
        it = L.getValue();
        do_something(it);
}
```

Use of List ADT: Find an Object

```
/* return true if k is in list L,   false otherwise */
template <typename E>
boolean find(List<E> L, E k) {
  for (L.moveToStart(); L.currPos()<L.length();   L.next()  )
       E item = L.getValue();
       if (k == item) return true;          // k found
       return false;                        // k not found
}
```

Use of List ADT: Find an Object

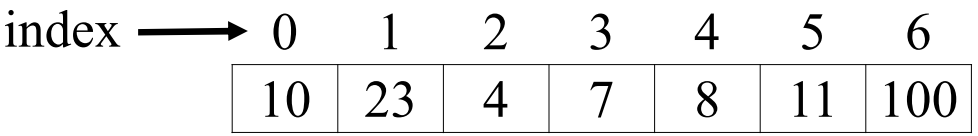
```
/* return true if k is in list L,
    false otherwise */
boolean find(List<int> L, int k) {
  for (L.moveToStart(); L.currPos()<L.length();   L.next()  )
       int item = L.getValue();
       if (k == item) return true;          // k found
       return false;                        // k not found
}
```

Different Implementations of List ADT

- Array based implementation
- Link list based implementation

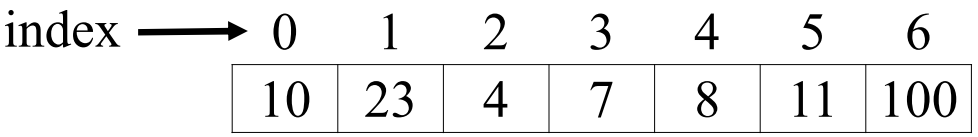
Array Data Structure

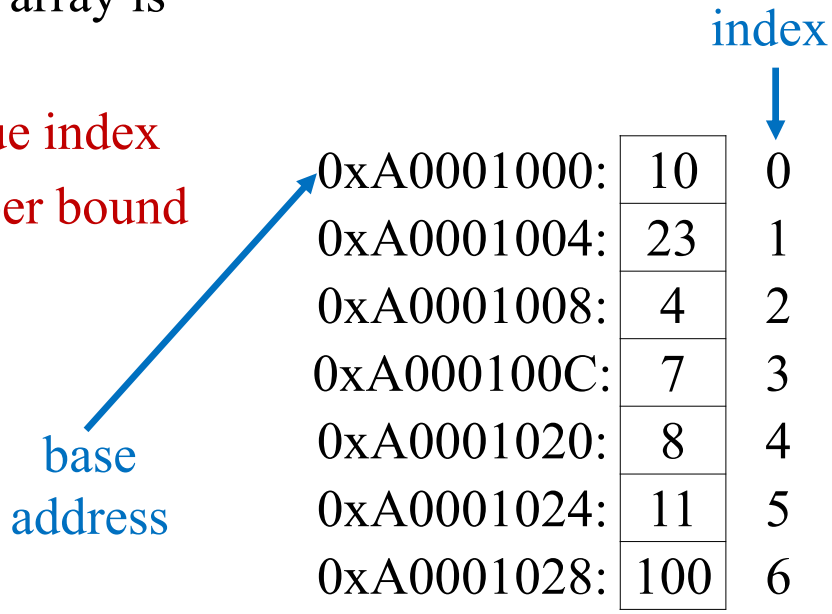
- An array is an indexed sequence of components
  - Typically, the array occupies sequential storage locations
  - The length of the array is determined when the array is created, and cannot be changed
  - Each component of the array has a fixed, unique index
    - Indices range from a lower bound to an upper bound



Array Data Structure

- An array is an indexed sequence of components
  - Typically, the array occupies sequential storage locations
  - The length of the array is determined when the array is created, and cannot be changed
  - Each component of the array has a fixed, unique index
    - Indices range from a lower bound to an upper bound

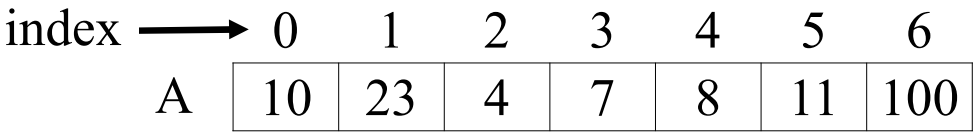




Array Data Structure

– Any component of the array can be inspected or updated by using its index

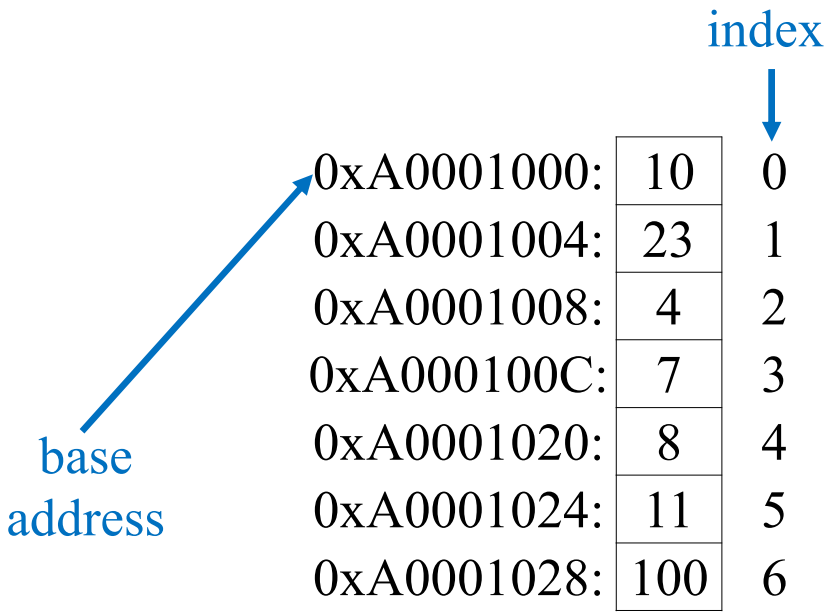
- This is an efficient operation: O(1) = constant time



Accessing A [1] or A[5] takes same time: Why?

Array Data Structure

A [1] is accessed from address 0xA0001000+4*1
A [5] is accessed from address 0xA0001000+4*5
.
.
A [*n*] is accessed from address 0xA0001000+4**n*



Accessing A [1] or A[5] takes same time: Why?

Array Data Structure

- Advantage:
  - Accessing an element by its index is very fast (constant time)

- Disadvantage:
  - All elements must be of the same type
  - Insertion into and deletion from arrays are very slow
  - The array size is fixed and can never be changed ??

Array Based Implementation (C++ template)

```
template <typename E> // Array-based list implementation
class AList : public List<E> {
private:
int maxSize; // Maximum size of list
int listSize; // Number of list items now
int curr; // Position of current element
E* listArray; // Array holding list elements
public:
AList(int size=defaultSize) { // Constructor
    //   initialize all private variables
}
~AList() { /*deallocate all memories*/ } // Destructor

// Insert "it" at current position
void insert(const E& it) {
    for(int i=listSize; i>curr; i--)          // Shift elements up to make room
            listArray[i] = listArray[i-1];
    listArray[curr] = it;
    listSize++; // Increment list size
}
.
}
```

Array Based Implementation (in C)

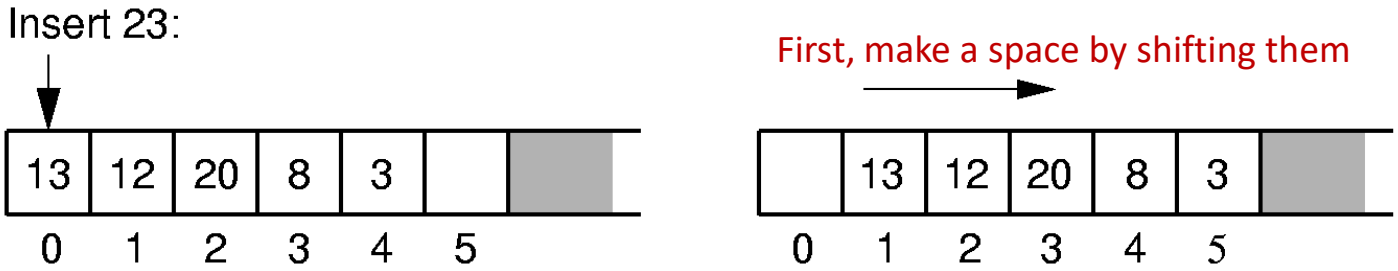
```
int maxSize; // Maximum size of list
int listSize; // Number of list items now
int curr; // Position of current element
int *listArray; // Array holding list elements

// Insert "it" at current position
void insert(int it) {
    for(int i=listSize; i>curr; i--) // Shift elements up
            listArray[i] = listArray[i-1]; // to make room
    listArray[curr] = it;
    listSize++; // Increment list size
}
```

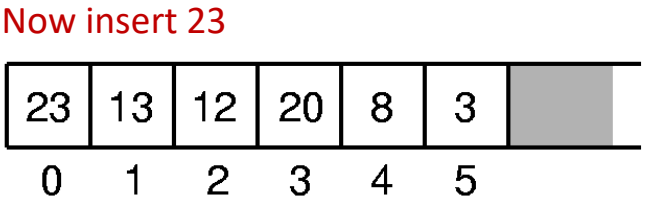
Array Based Implementation (in C)

Algorithm for insertion of a new item

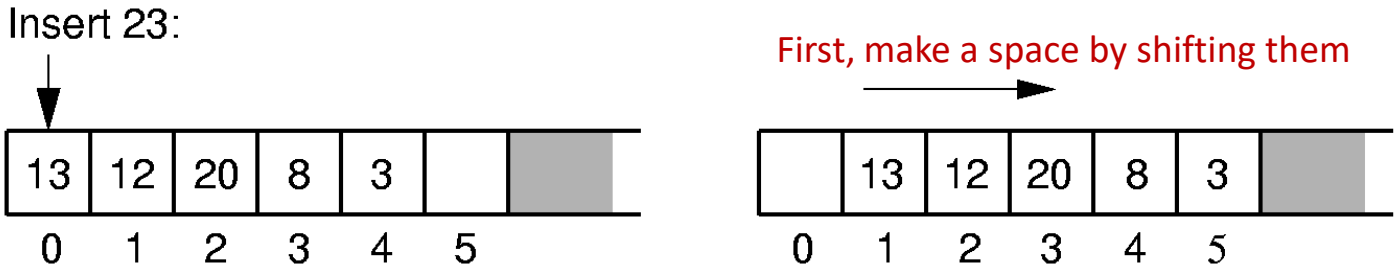
1. Move listArray[*curr*] .. . listArray[*n*] to listArray [*curr*+1] .. . listArray[*n* +1]
2. Insert new item in listArray[curr]
3. Increase listSize by 1

Array Based Implementation

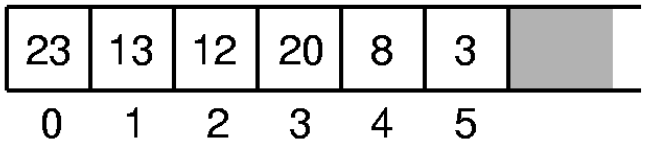




Array Based Implementation

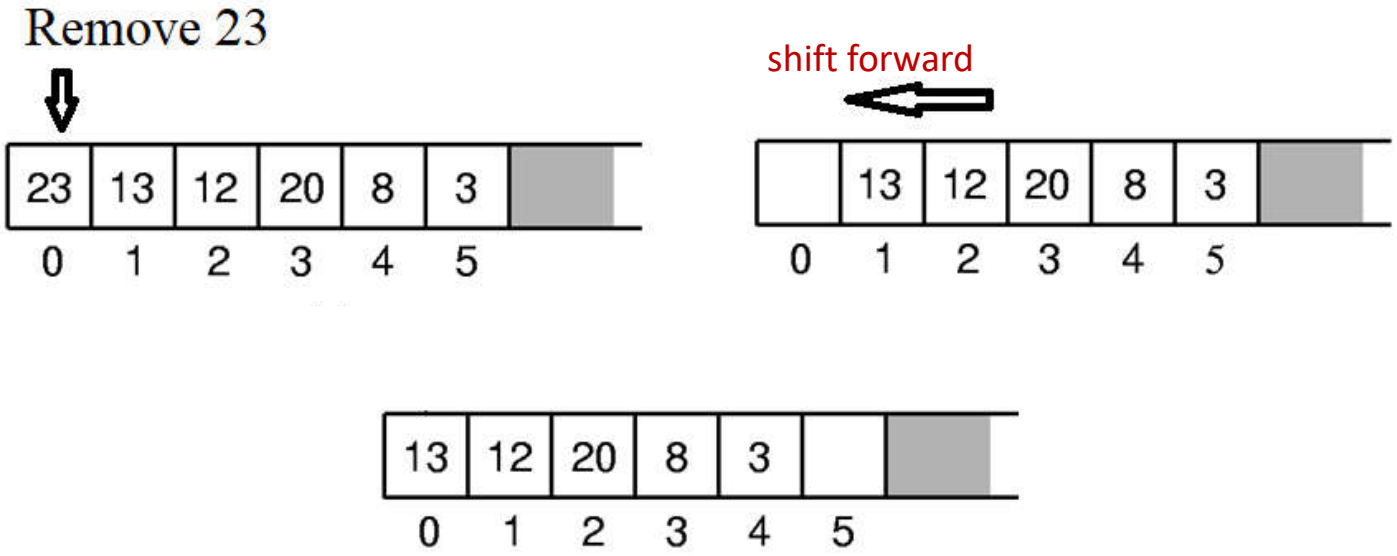


Now insert 23



Complexity?

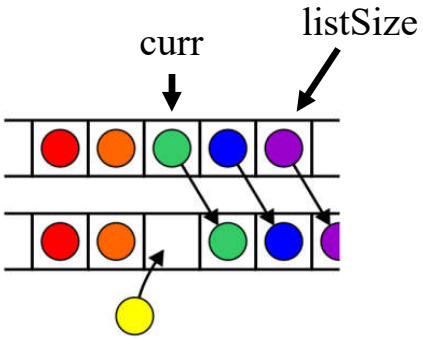
Array Based Implementation



Complexity: $O(n)$

Array Based Implementation (insertion at current position)

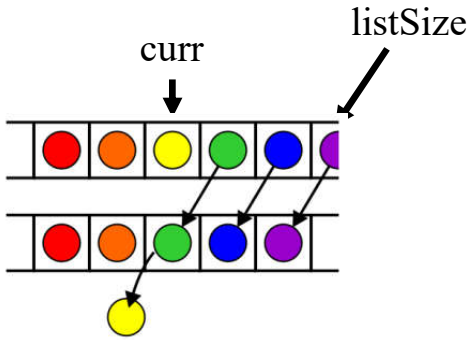
```
for(int i=listSize; i>curr; i--) // Shift elements up
        listArray[i] = listArray[i-1]; // to make room
  listArray[curr] = it;
  listSize++; // Increment list size
```



Complexity: $O(n)$

Array Based Implementation (remove from current position)

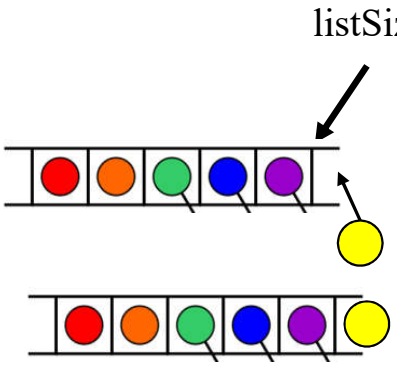
```
int it = listArray[curr];
for(int i=curr; i<listSize-1; i++)
  listArray[i] = listArray[i+1];
listSize--;
return it;
```



Complexity: $O(n)$

Array Based Implementation (Append)

```
listArray[listSize++] = it;
```



Complexity: $O(1)$

Array Based Implementation (Resizing array)

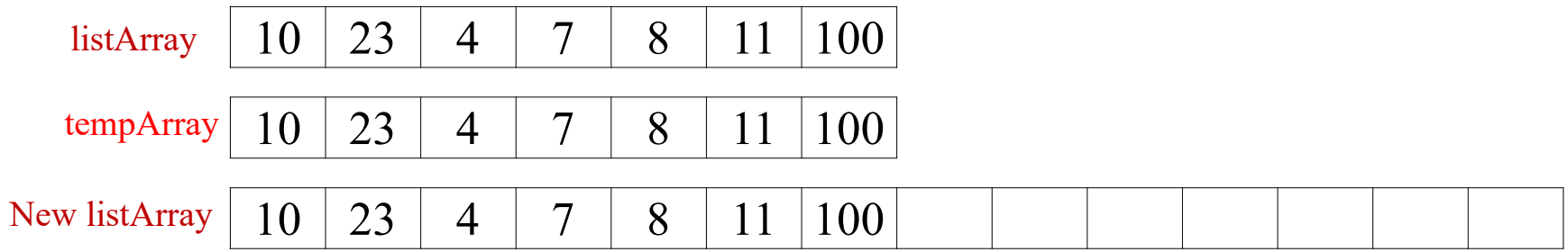
Think that multiple insertion/append makes the listArray full.

No more insertion/append is possible.

Array Based Implementation (Resizing array)

Assume both listArray and tempArray be pointers to integers

1. Allocate memory for tempArray
2. Copy all elements of listArray to tempArray
3. Free existing memory of listArray
4. Reallocate double storage for listArray
5. Copy all elements of tempArray to listArray
6. Deallocate storage of tempArray



Array Based Implementation

Operations	Complexity
`insert (front)`	linear
`insert (middle)`	linear
`remove (front)`	linear
`remove (middle)`	linear
`append`	(usually) constant
`remove (back)`	constant
`getValue`	constant
`setValue`	constant