Lecture 6
List ADT

It is very pervasive

Lecture Overview

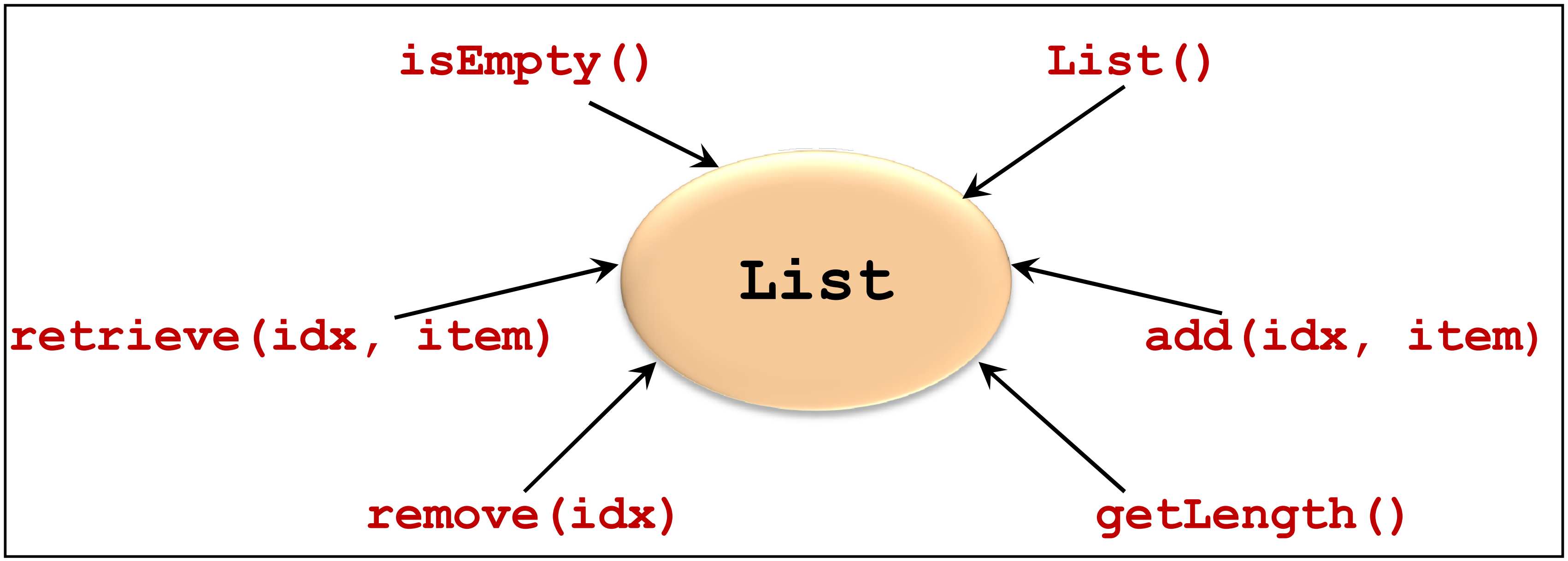
- List ADT
  - Specification

- Implementation for List ADT
  - Array Based
    - Pros and Cons

  - Linked List Based
    - Pros and Cons

List ADT

- A sequence of items where positional order matter $<a_1, a_2, \ldots, a_{n-1}, a_n>$
- Lists are very pervasive in computing
  - e.g. student list, list of events, list of appointments etc



The `list` ADT

`idx` : Position, integer
`item` : Data stored in list, can be any data type

List ADT : C++ Specification

```
// includes are not shown

class ListBase {
public:
    virtual bool isEmpty() = 0;
    virtual int getLength() = 0;

    virtual bool insert(int index, const int& newItem) = 0;
    virtual bool remove(int index) = 0;
    virtual bool retrieve(int index, int& dataItem) = 0;

    virtual string toString() = 0;
};
```

Operations to check on the state of list.

The three major operations

Operation to ease printing & debugging.

ListBase.h

Design Decisions

- This is a simplified design:
  - to reduce the "syntax burden"
  - to concentrate on the internal logic

- You are encouraged to enhance the class:
  - After you have understood the internal logic

- Possible enhancements:
  - Use **Template Class**:
    - So that list can contain item of any data type
  - Use **Inheritance + Polymorphism**:
    - Similar to the Complex Number ADT

Two Major Implementations

1. Array implementation
2. Linked list implementation (discussed soon)

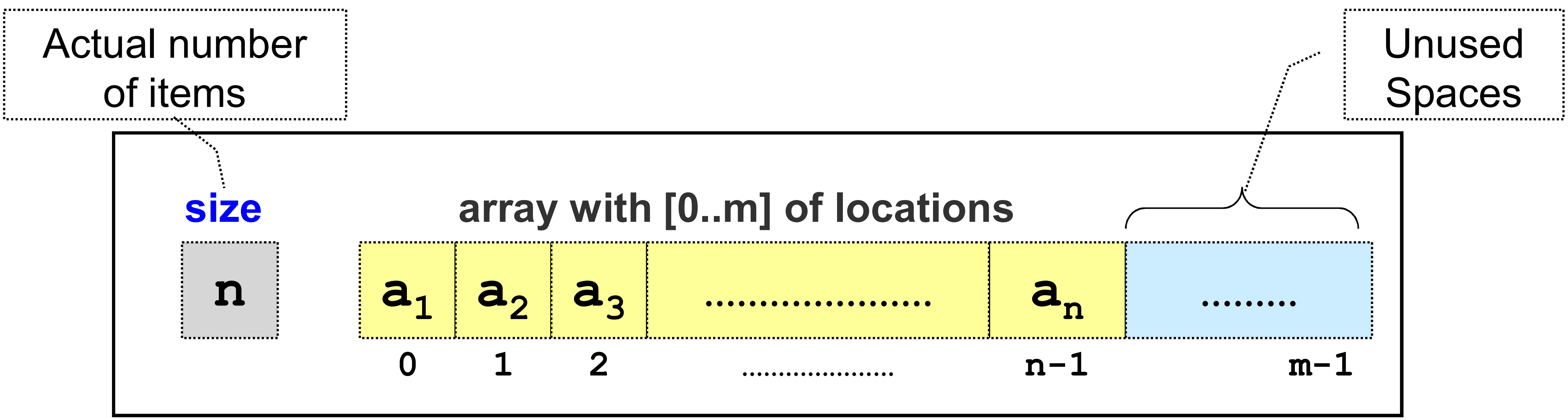
- General steps:
  1. Choose an **internal data structure**
     - e.g. Array or linked list
  2. Figure out the algorithm needed for each of the major operations in List ADT:
     - **insert**, **remove,** and **retrieve**
  3. Implement the algorithm from step (2)

List ADT – Version A

Array Implementation

Implement List ADT: Using Array

- Array is a prime candidate for implementing the ADT
  - Simple construct to handle a collection of items
- **Advantage:**
  - Very fast retrieval

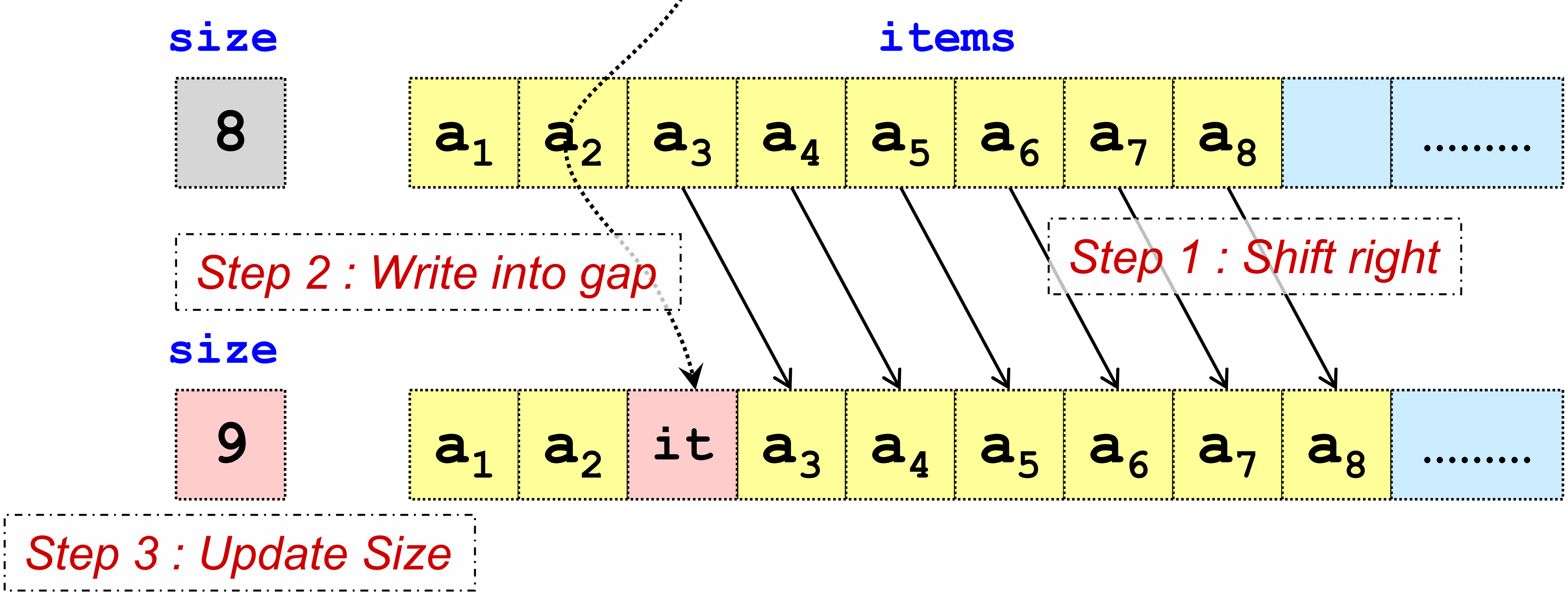


Internal of the `list` ADT, Array Version

Insertion : Using Array

- **Simplest Case:** Insert to the end of array
- Other Insertions:
  - Some items in the list needs to be shifted
  - **Worst case:** Inserting at the head of array

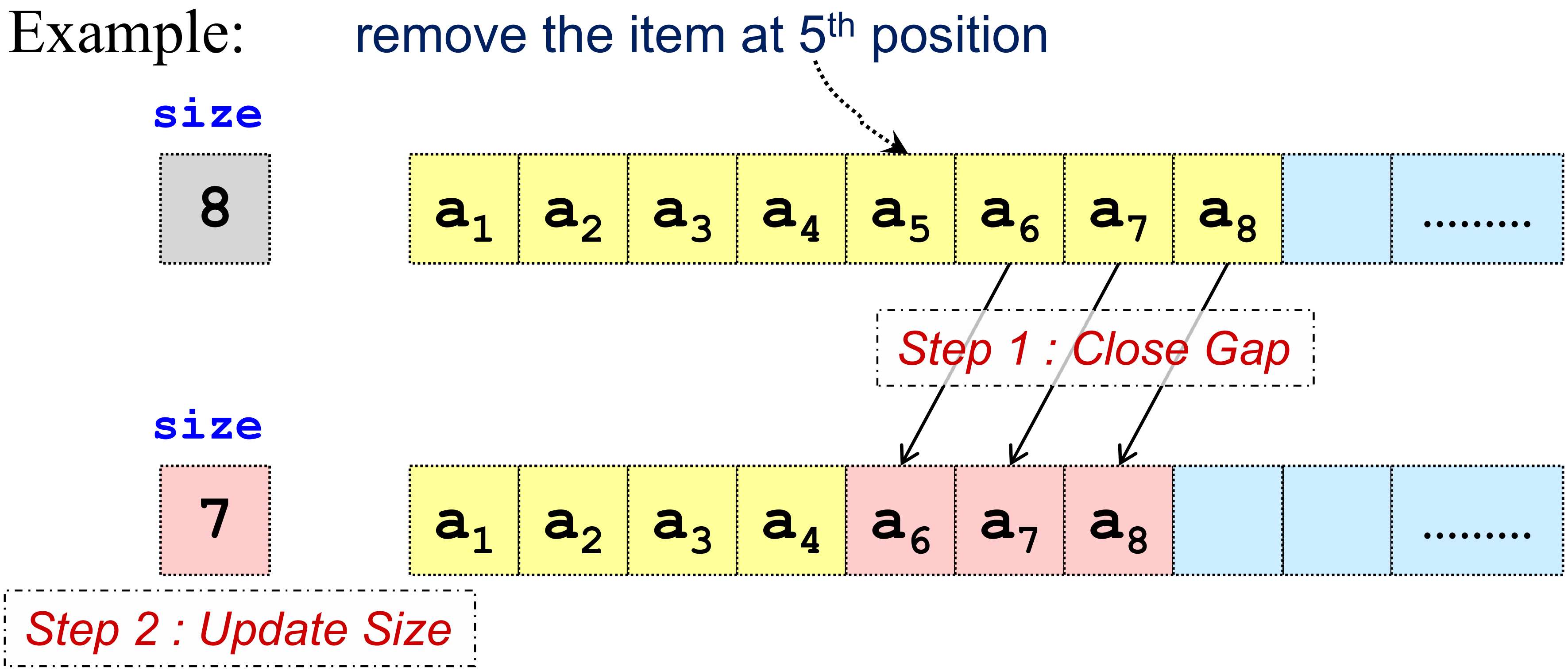
Example : Insert item "*it*" into the 3rd position



Deletion: Using Array

- **Simplest Case:** Delete item from the end of array
- Other deletions:
  - Items needs to be shifted
  - **Worst Case:** Deleting at the head of array

Example:



List Array: Specification

```
#include "ListBase.h"

const int MAX_LIST = 50;

class ListArray : public ListBase {

private:
    int _size;
    int _items[MAX_LIST];

public:
    ListArray();

    virtual bool isEmpty();
    virtual int getLength();

    virtual bool insert(int index, const int& newItem);
    virtual bool remove(int index);
    virtual bool retrieve(int index, int& dataItem);

    virtual string toString();
};
```

Items stored in an static array

ListArray.h

List Array: Implementation (1/4)

```
#include <sstream>
#include "ListArray.h"

ListArray::ListArray() {
   _size = 0;
}

bool ListArray::isEmpty() {
   return _size == 0;
}

int ListArray::getLength() {
    return _size;
}
```

ListArray.cpp (Part 1)

- ***isEmpty()*** and ***getLength()*** methods are easy to code:
  - will be omitted in later implementations

List Array: Implementation (2/4)

```
bool ListArray::insert(int userIdx, const int& newItem) {
    int index = userIdx-1;

    if (_size >= MAX_LIST)
        return false;

    if ((index < 0) || (index >= _size+1))
        return false;

    for (int pos = _size-1; pos >= index; pos--)
        _items[pos+1] = _items[pos];

    _items[index] = newItem;

    _size++;

    return true;
}
```

List index starts from 1, but array index starts from 0

Maximum capacity reached

List index out of range

Step 1. Shift items

Step 2. Write into gap

Step 3. Update Size

ListArray.cpp (Part 2)

List Array: Implementation (3/4)

```
bool ListArray::remove(int userIdx) {
    int index = userIdx-1;

    if ((index < 0) || (index >= _size))
        return false;

    for (int pos = index; pos < _size-1; pos++)
        _items[pos] = _items[pos+1];

    _size--;

    return true;
}
```

List index out of range

Step 1. Close gap

Step 2. Update size

ListArray.cpp (Part 3)

List Array: Implementation (4/4)

```
bool ListArray::retrieve(int userIdx, int& dataItem) {
    int index = userIdx-1;

    if ((index < 0) || (index >= _size))
        return false;

    dataItem = _items[index];
    return true;
}

string ListArray::toString() {
    ostringstream os;

    os << "[ ";
    for (int i = 0; i < _size; i++)
        os << _items[ i ] << " ";
    os << "]";

    return os.str();
}
```

Retrieval is simple, as array item can be accessed directly.

The result is passed back through the reference parameter

A useful method to print all items into a string with the format

[*item1 item2* … *itemN*]

ListArray.cpp (Part 4)

Using the List ADT: User Program

- Instead of an actual List ADT application, we show a program used to test the implementation of various List ADT operations

- Pay attention to **how we test** the operations:
  - For each operations:
    - Test different scenarios, basically to exercise different "decision path" in the implementation
  - For example, to test the **insert** operation:
    - Insert into an empty list
    - Insert at the first, middle and last position of the list
    - Insert with incorrect index

List ADT : Sample User Program 1/2

```
#include <iostream>
#include "ListArray.h"
using namespace std;

int main() {
    ListArray intList;
    int rItem;

    if (intList.insert(1, 333))
        cout << "Insertion successful!\n";
    else
        cout << "Insertion failed!\n";

    intList.insert(1, 111);
    intList.insert(3, 777);
    intList.insert(3, 555);
```

Using the array implementation of list

This is one way to use the operations: Check the return result for the status of the operation.

If the insertion is implemented properly, the list should contain
`[ 111 333 555 777 ]`
at this point

`ListTest.cpp` (Part 1)

List ADT : Sample User Program 2/2

```
    cout << intList.toString() << endl;

    intList.retrieve(1, rItem);
    cout << "First item is " << rItem << endl;
    intList.retrieve(intList.getLength(), rItem);
    cout << "Last item is " << rItem << endl;

    cout << "Remove test" << endl;
    intList.remove(1);
    intList.remove(2);
    intList.remove(intList.getLength());

    intList.retrieve(1, rItem);
    cout << "First item is " << rItem << endl;
    intList.retrieve(intList.getLength(), rItem);
    cout << "Last item is " << rItem << endl;

    return 0;
}
```

Test **toString**() and also confirm the content of List

Test **retrieve**() and **getLength**()

Test **removal**():

-remove 1st item
-remove item in the middle
-remove last item

ListTest.cpp (Part 2)

Array Implementation: **Efficiency (time)**

- **Retrieval:**
  - **Fast:** one access
- **Insertion:**
  - **Best case:** No shifting of elements
  - **Worst case:** Shifting of all ***N*** elements.
- **Deletion:**
  - **Best case:** No shifting of elements
  - **Worst case:** Shifting of all ***N*** elements

Array Implementation : **Efficiency (space)**

- Size of array is restricted to **`MAX_LIST`**
- **Problem:**
  - Maximum size is **not known in advance**
    - **`MAX_LIST`** is too big == unused space is wasted
    - **`MAX_LIST`** is too small == run out of space easily
- **Solution:**
  - Make **`MAX_LIST`** ***a variable***
  - When array is full:
    1. Create a larger array
    2. Move the elements from the old array to the new array
  - No more limits on size, but *space wastage and copying overhead is still a problem*

Array Implementation : **Observations**

- For **fixed-size collections**
  - Arrays are **great**

- For **variable-size collections**, where dynamic operations such as insert/delete are common
  - Array is a **poor choice** of data structure

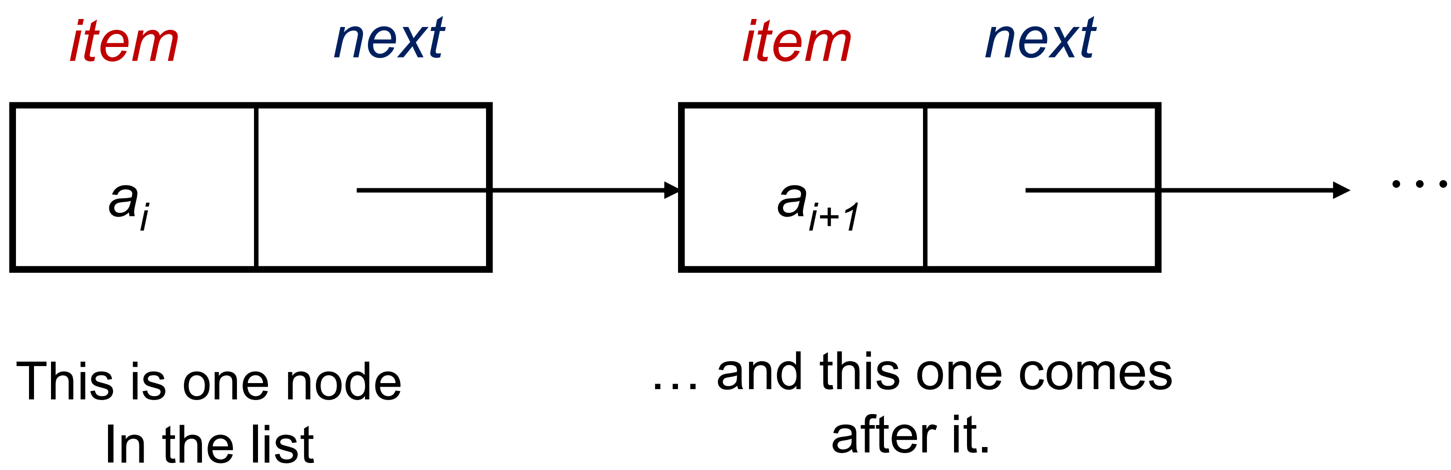
  - For such applications, ***there is a better way……***

List ADT – Version B

Linked List Implementation

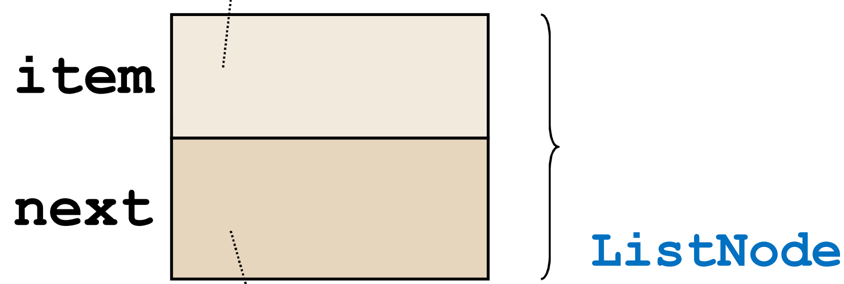
Implement List ADT using Linked List

- Pointer Based Linked List:
  - Allow elements to be **non-contiguous** in memory
  - Order the elements by associating each with its **neighbour(s)** through pointers



A single node in the Linked List

```
struct ListNode {

    int item;
    ListNode *next;

};
```



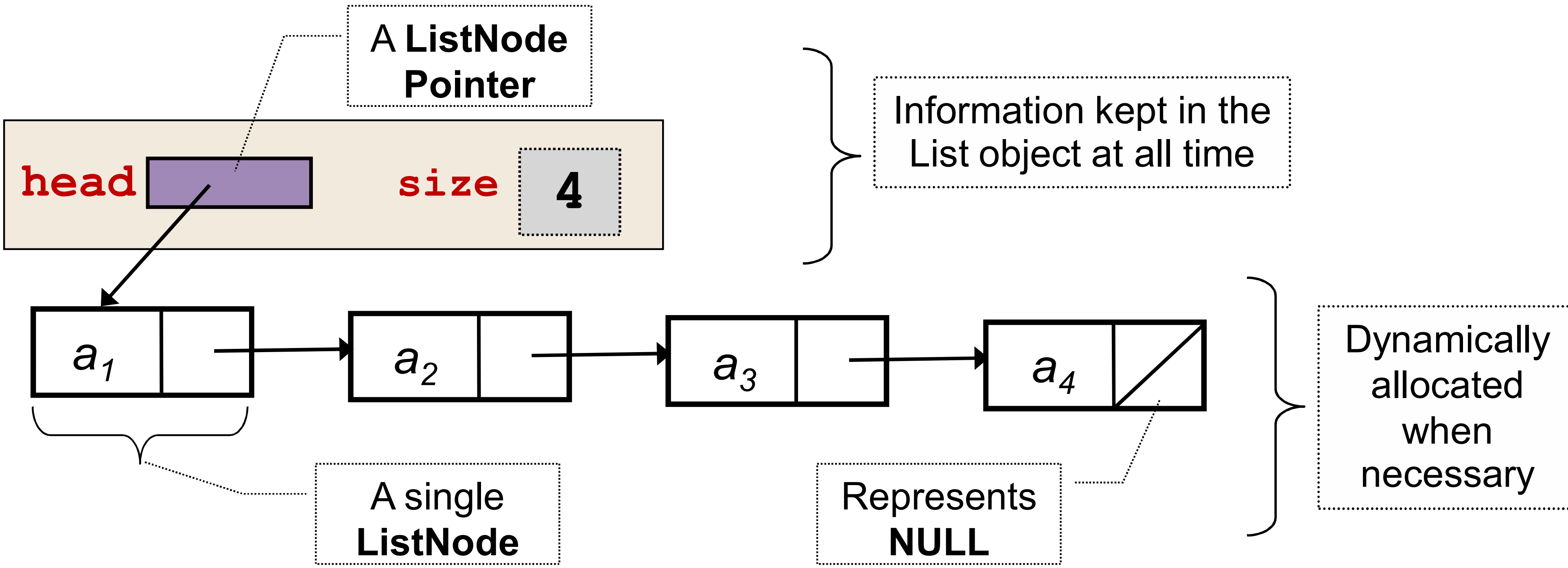
Store a single integer in this example

A pointer to another structure with the same layout

C++ Allows structure name to be used **without** the keyword **struct**

List ADT: Using **Linked List**

- List of four items < a_1, a_2, a_3, a_4 >



- We need:
  - `head` pointer to indicate the first node
    - Other nodes are accessed by "hopping" through the next pointer
  - `size` for the number of items in the linked list

Linked List Implementation: **Design**

- Linked list implementation is more complicated:
  - Need to handle a number of scenarios separately

- Let us walkthrough the insertion algorithm in detail:
  - Highlight the importance of design before coding
  - Highlight the design considerations:
    - How to consider different cases?
    - How to modularize and reuse common code?

Linked List Insertion: General

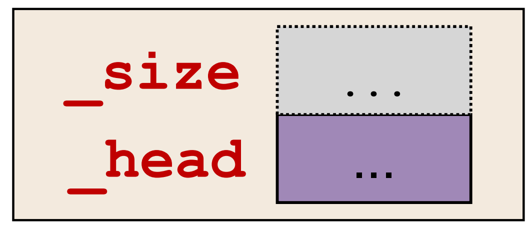
- List ADT provides the ***insert()*** method to add an item:
  - The new item itself is given
  - The index [**1**…**size+1**] of the new item is given

- Due to the nature of linked list, there are several possible scenarios:
  1. Item is added to an **empty** linked list
  2. Item is added to the **head (first item)** of the linked list
  3. Item is added to the **last position** of the linked list
  4. Item is added to the **other positions** of the linked list

Linked List Insertion: **Preliminary**

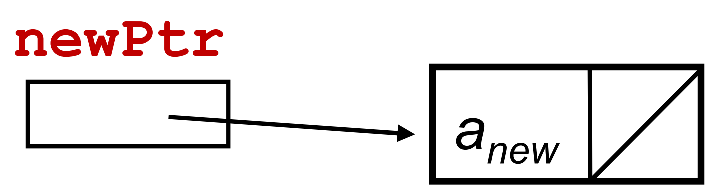
- The `List` object stores:
  - Head pointer and the current size of linked list

```
class List {
private:
    int _size;
    ListNode* _head;
    ....
};
```

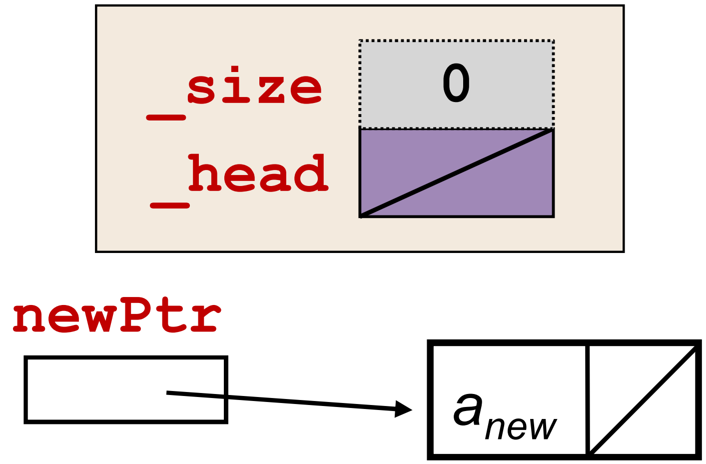


- For all valid cases, we need to construct a new linked list node to store the new item

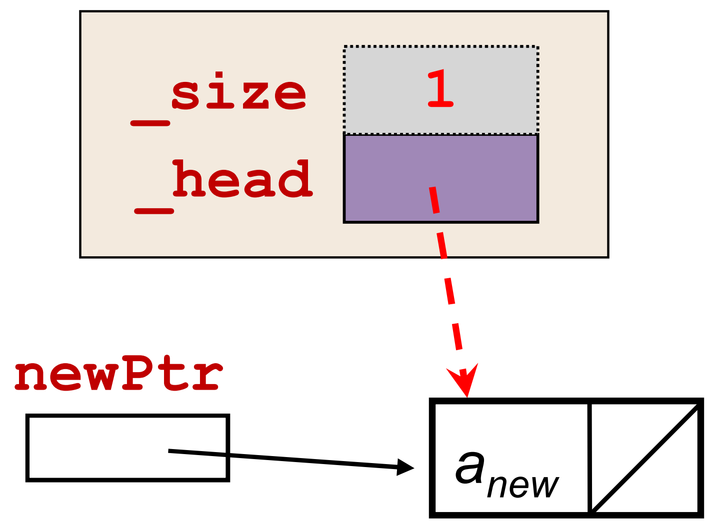
```
ListNode *newPtr;

newPtr = new ListNode;
newPtr->item = a_new;
newPtr->next = NULL;
```



Insertion Case 1: **Empty Linked List**

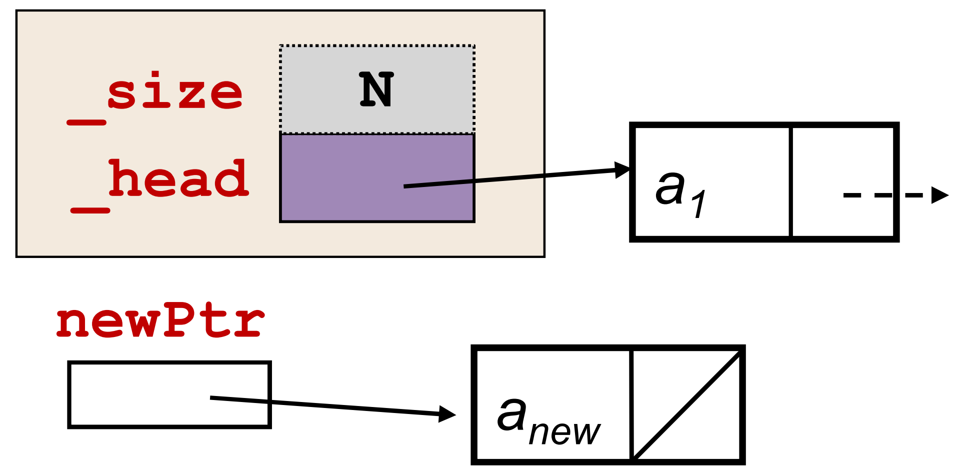



```
_size++;

_head = newPtr;
```

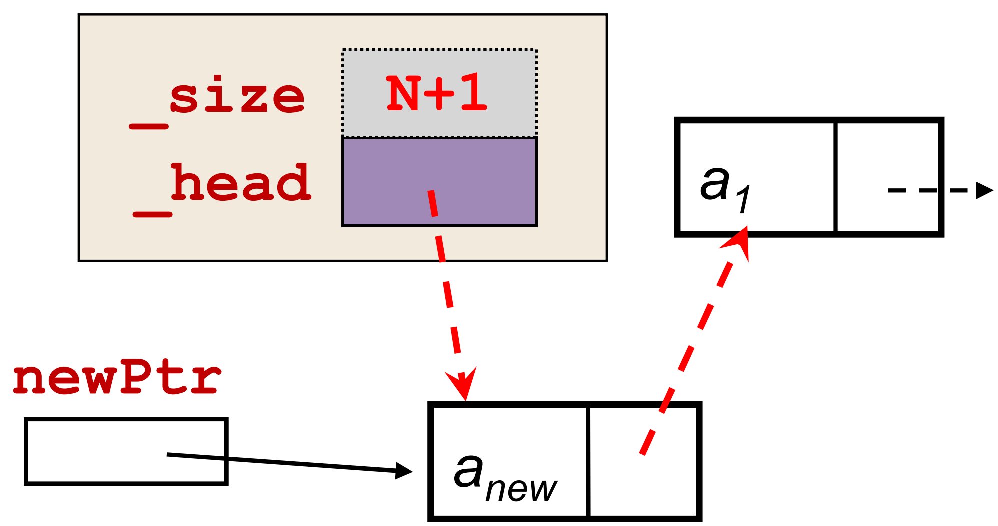


Question: is **`newPtr`** needed after this operation?

Insertion Case 2: **Head of Linked List**




```
_size++;

newPtr->next = _head;
_head = newPtr;
```

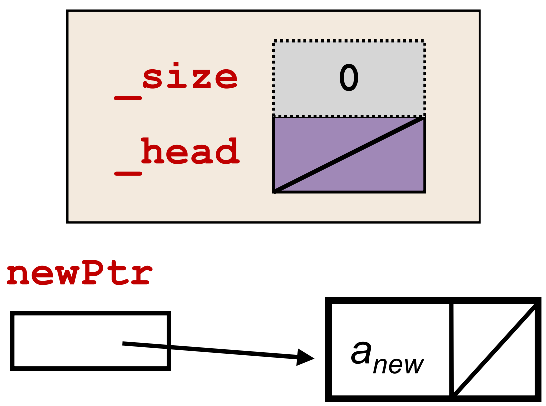


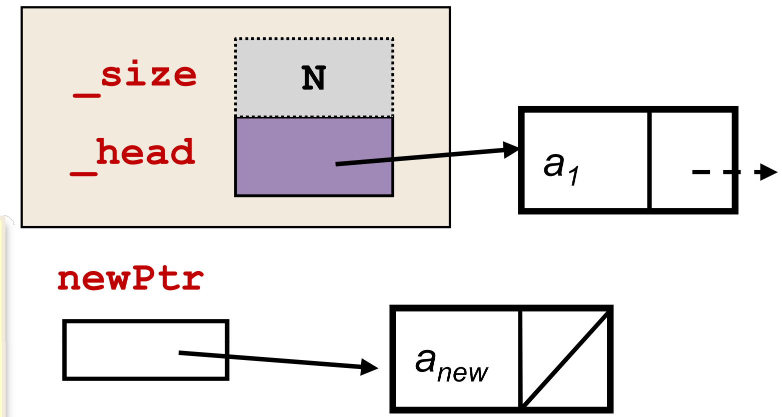
Question1: Can we reorder the last two lines of code above?

Question2: Very similar to previous case, can we combine?

Insertion Case 1 and 2: **Common Code**

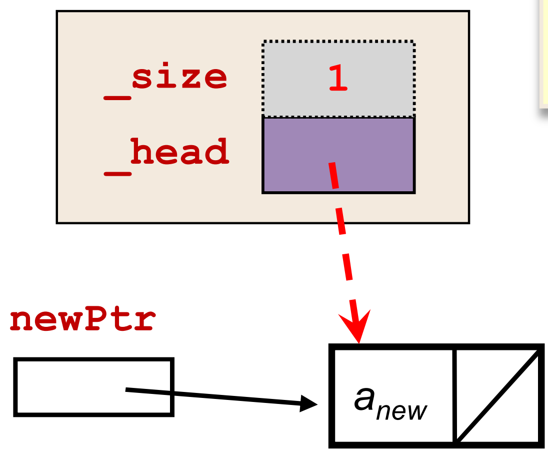
- Insert into head of linked list (possibly empty)

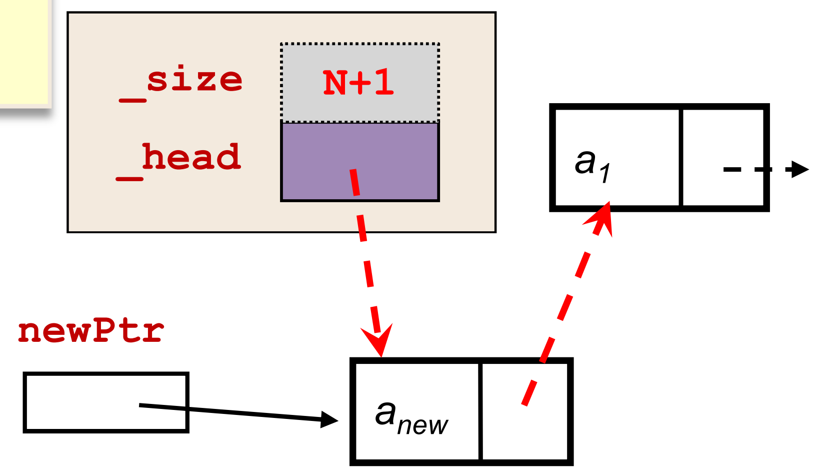




Can use the same code:

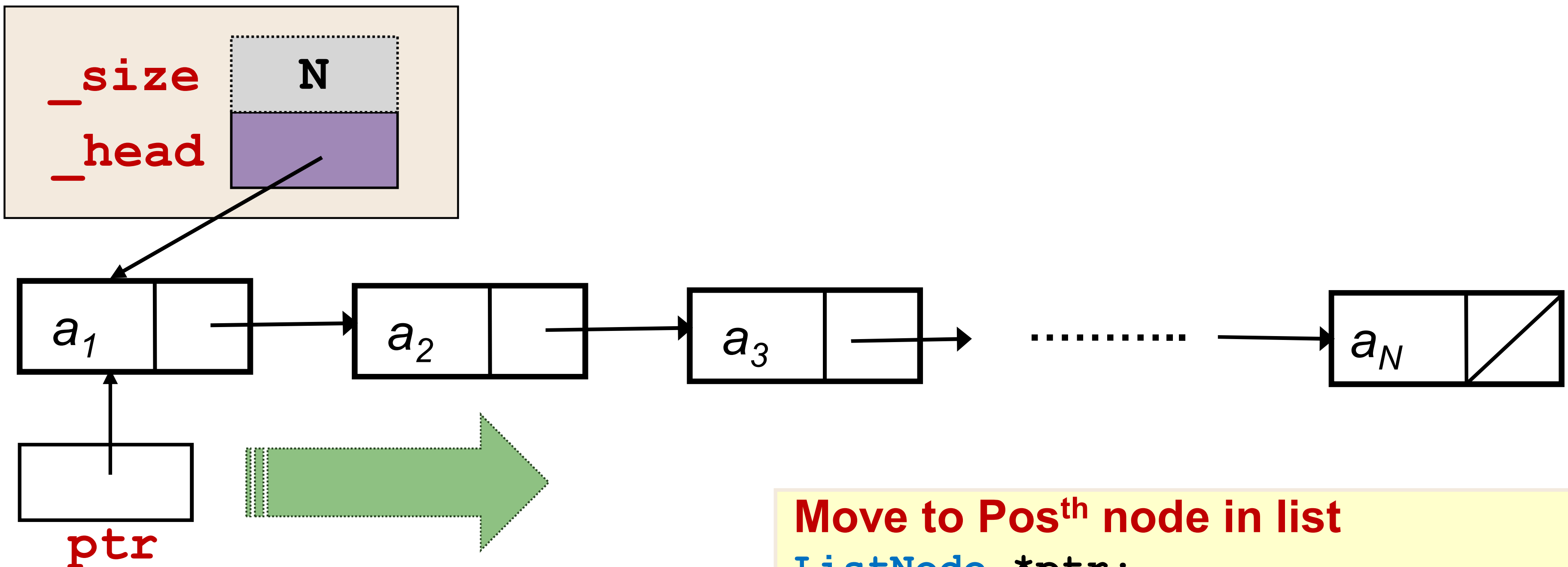
```
_size++;

newPtr->next = _head;
_head = newPtr;
```





Linked List Insertion: **Traversal**

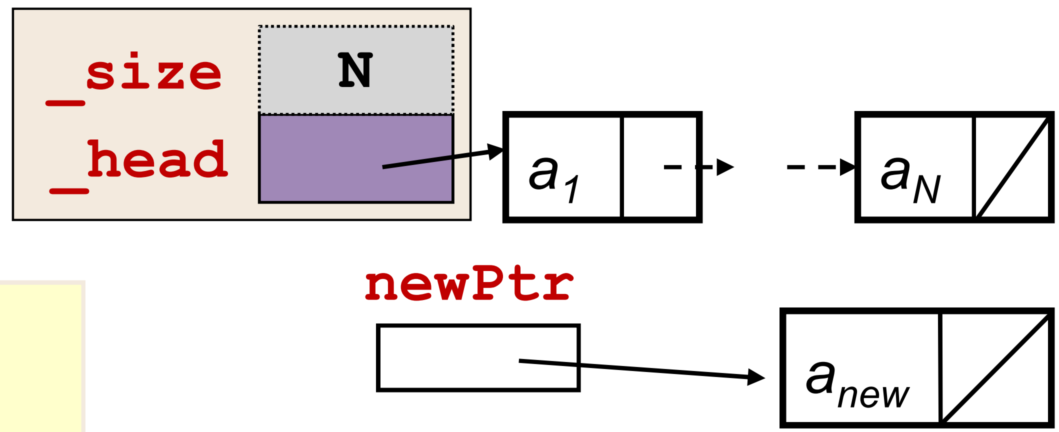
- Since we only keep the head pointer, **list traversal** is needed to reach other positions
  - Needed for insertion case 3 and 4



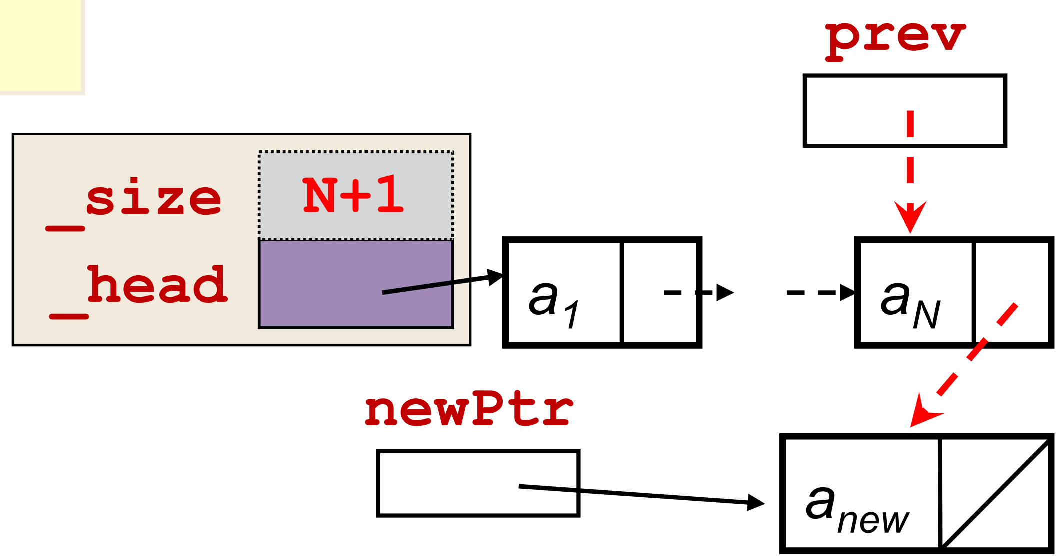
Move to Posth node in list

```
ListNode *ptr;

ptr = _head;
for (i = 1; i < Pos; i++) {
   ptr = ptr->next;
}
```

Insertion Case 3: **End of Linked List**



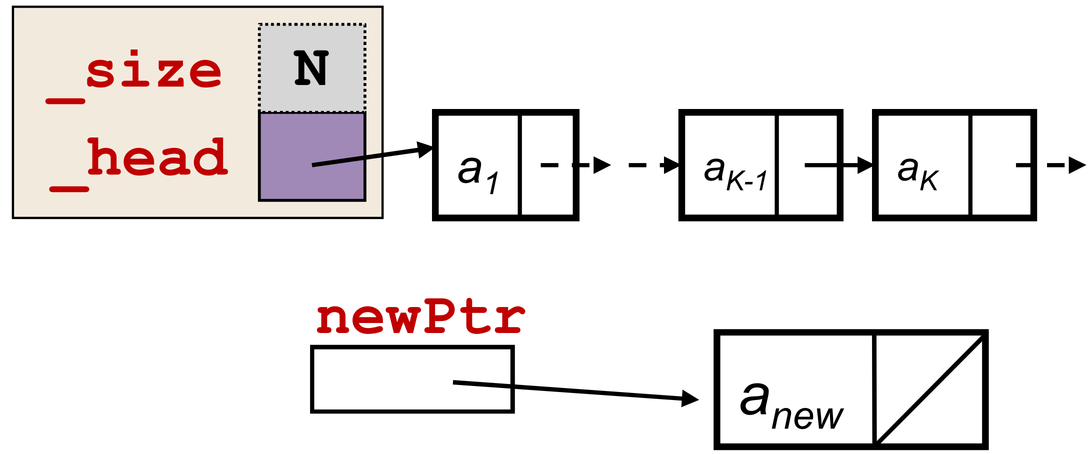

```
ListNode* prev;

prev = Traverse to Nth node

_size++;
prev->next = newPtr;
```

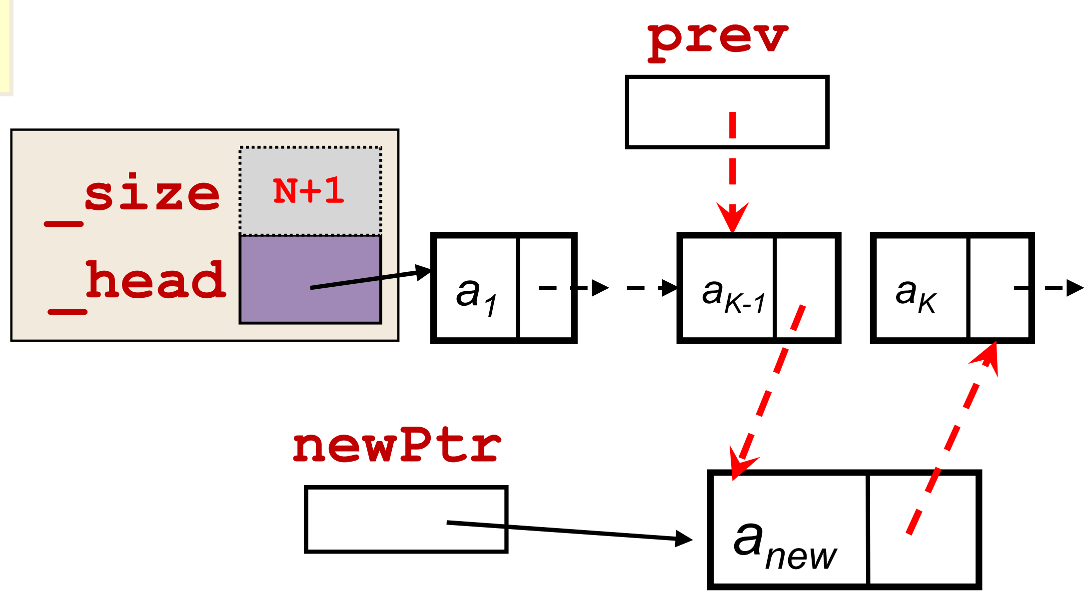


Use the list traversal code discussed to move `prev` pointer

Insertion Case 4: K^{th} Position (Middle)

```
ListNode* prev;

prev = Traverse to K-1th node

_size++;
newPtr->next = prev->next;
prev->next = newPtr;
```



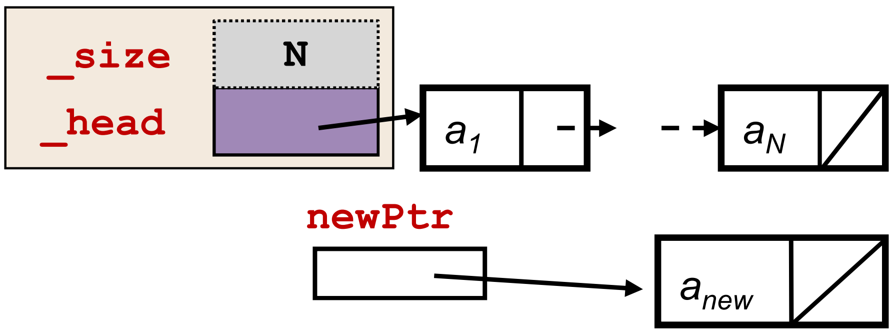


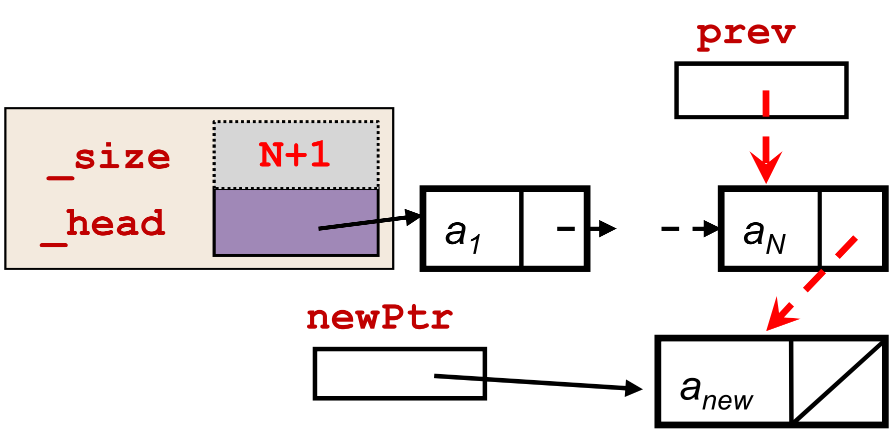
We only need to change at most two pointers for ANY insertion

Insertion Case 3 and 4: **Common Code**

- The code for case 4 happens to be a more general form of case 3:
  - Case 3 can be handled with the same code in case 4

```
ListNode* prev;

K = N+1
prev = Traverse to K-1th node

_size++;
newPtr->next = prev->next;
prev->next = newPtr;
```





Insertion Code: **Summary**

- To insert `ItemNew` into `Index`th position
  - Assume `Index` is in range **[1….size+1]**

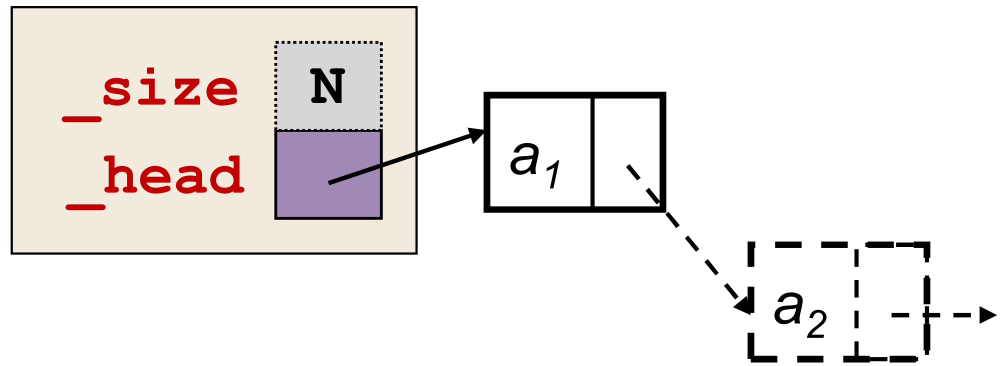
```
ListNode *newPtr, *prev;

1.newPtr = new ListNode // Create a new node
   i. item = ItemNew
   ii.next pointer = NULL
2._size increases by 1
3.If Index is 1      // Case 1 + 2
   i. newPtr->next = _head
   ii._head = newPtr
4.Else               // Case 3 + 4
   i.  prev = Traverse to Index-1th node
   ii. newPtr->next = prev->next
   iii.prev->next = newPtr
```

Linked List Deletion: **General**

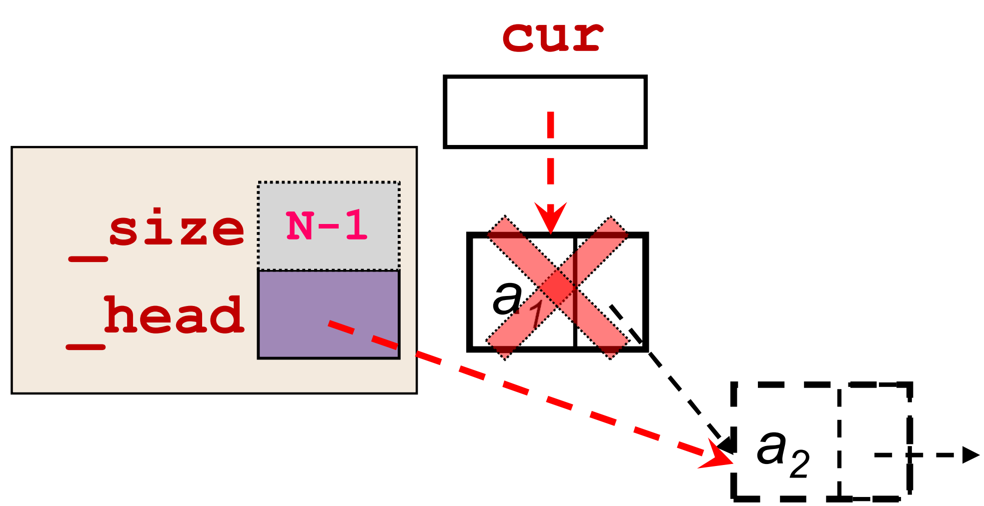
- For Linked List deletion, the cases can be simplified similar to:
  1. Deletion of head node (1^{st} Node in list)
  2. Deletion of other node (including middle or end of list)

- Try to deduce the code logic using similar approach

Deletion Case 1: **Head of Linked List**



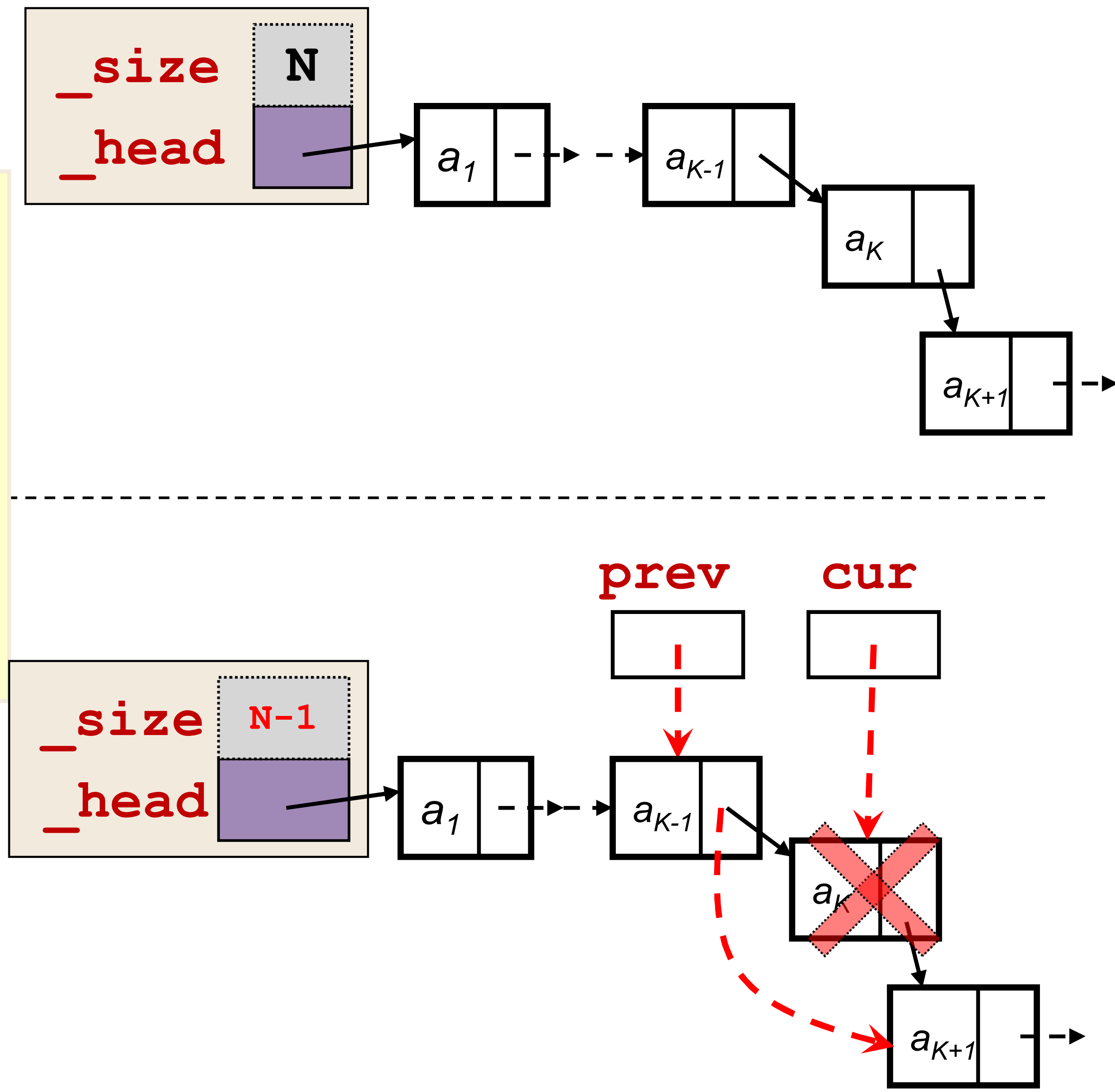

```
ListNode* cur;

_size--;

cur = _head;
_head = _head->next;

delete cur;
```



Question: What if there is only 1 node? Will the code work?

Deletion Case 2: **K^th Position (Middle)**

```
ListNode *prev, *cur;

prev = Traverse to K-1th node

_size--;
cur = prev->next;
prev->next = cur->next;

delete cur;
```



Question: What if the K^th node is the last node?

Deletion Code: **Summary**

- To delete item at `Index`th position
  - Assume `Index` is in range **[1....size]**

```
ListNode *prev, *cur;

1. _size decreases by 1
2. If Index is 1        // Case 1
   i. cur = _head
   ii. _head = _head->next;
3. Else                 // Case 2
   i.   prev = Traverse to Index-1th node
   ii.   cur = prev->next
   iii. prev->next = cur->next
4. Free memory pointed by cur pointer
```

The traversal code can be shared between insertion and deletion. Let's make it into another method

List(Linked List): Specification

```cpp
#include "ListBase.h"
class ListLL : public ListBase {
private:
    struct ListNode {
        int item;
        ListNode *next;
    };
    int _size;
    ListNode* _head;
    ListNode* traverseTo(int index);
public:
    ListLL();
    ~ListLL();

    virtual bool isEmpty();
    virtual int getLength();

    virtual bool insert(int index, const int& newItem);
    virtual bool remove(int index);
    virtual bool retrieve(int index, int& dataItem);

    virtual string toString();
};
```

The **`ListNode`** structure declaration is **hidden** from outsider

The **`traversal`** method is only used internally

ListLL.h

ListLL: Implementation (1/5)

```
ListLL::ListLL() {
    _head = NULL;
    _size = 0;
}

ListLL::~ListLL() {
    while (!isEmpty())
        remove(1);
}

string ListLL::toString() {
    ostringstream os;
    ListNode *cur;

    os << "[ ";
    for (cur = _head; cur != NULL; cur = cur->next) {
        os << cur->item << " ";
    }
    os << "]";

    return os.str();
}
```

We need a destructor to free each node as they are dynamically allocated.

Go through all nodes to print all items.

ListLL.cpp (part 1)

ListLL: Implementation (2/5)

This is the return type of *find()* method.
ListLL:: is needed because **ListNode** is a **private declaration** in class **ListLL**.

```
ListLL::ListNode* ListLL::traverseTo(int index) {
    if ((index < 1) || (index > getLength()))
        return NULL;
    else {
        ListNode *cur = _head;

        for (int skip = 1; skip < index; skip++)
            cur = cur->next;

        return cur;
    }
}
```

index out of range

Traversal code as discussed.

This method is used in both insertion and deletion

ListLL.cpp (part 2)

ListLL: Implementation (3/5)

```
bool ListLL::insert(int userIdx, const int& newItem) {
    int newLength = getLength()+1;

    if ((userIdx < 1) || (userIdx > newLength))
        return false;
    else {
        ListNode *newPtr = new ListNode;
        newPtr->item = newItem;
        newPtr->next = NULL;
        _size = newLength;

        if (userIdx == 1) {
            newPtr->next = _head;
            _head = newPtr;
        }
        else {
            ListNode *prev = traverseTo(userIdx-1);
            newPtr->next = prev->next;
            prev->next = newPtr;
        }
    }
    return true;
}
```

List index out of range

Refer to the insertion code summary. See how the steps are translated into actual code.

ListLL.cpp (part 3)

ListLL: Implementation (4/5)

```
bool ListLL::remove(int userIdx) {
    ListNode *cur;

    if ((userIdx < 1) || (userIdx > getLength()))
        return false;
    else {
        --_size;
        if ( userIdx == 1 ) {
            cur = _head;
            _head = _head->next;
        }
        else {
            ListNode *prev = traverseTo(userIdx-1);
            cur = prev->next;
            prev->next = cur->next;
        }
        delete cur;
        cur = NULL;
    }
    return true;
}
```

List index out of range

Immediately set a deleted pointer to NULL is a good programming practice.

ListLL.cpp (part 4)

ListLL: Implementation (5/5)

```
bool ListLL::retrieve(int userIdx, int& dataItem) {
    if ((userIdx < 1) || (userIdx > getLength()))
        return false;
    else {
        ListNode *cur = traverseTo(userIdx);
        dataItem = cur->item;
    }
    return true;
}
```

List index out of range

With the traversal method, retrieval is straightforward

ListLL.cpp (part 5)

- With the complete linked list implementation of List ADT:
  - We can use the same user program to try it out
- What do you think is the main difference between **array** and **linked list** implementation of List ADT?

List ADT : Sample User Program (Again!)

```
#include <iostream>
#include "ListLL.h"
using namespace std;

int main() {
    ListLL intList;
    int rItem;

    // All other usage of List ADT remain
    // unchanged..........

    // You can see how easy to change between
    // the two implementations of List ADT
```

Using the linked list implementation of list

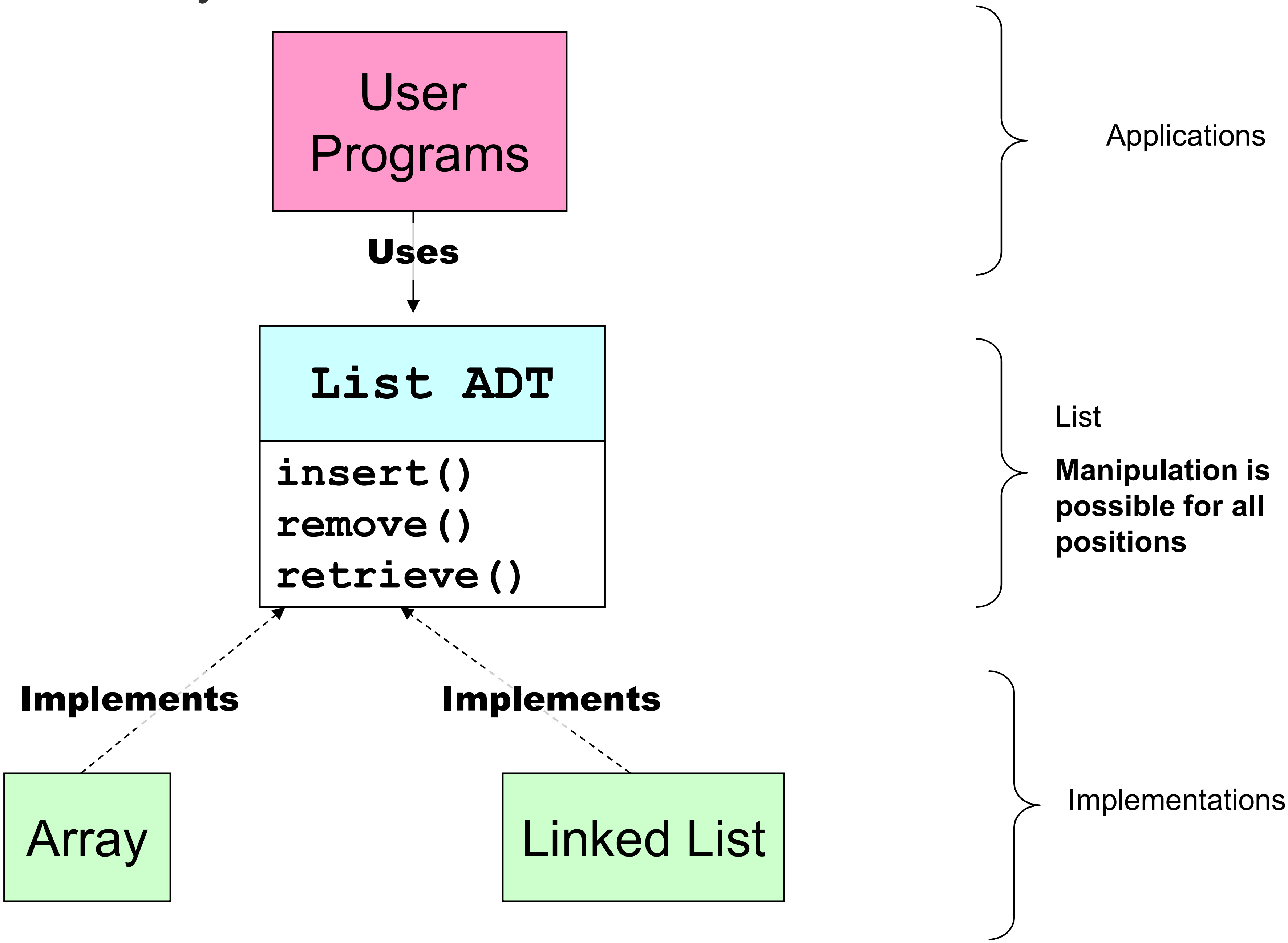
Question:

What do you think is the main difference between **ListArray** and **ListLL** implementation of List ADT?

References

- [Carrano]
  - Chapter 3
    - List ADT and array based implementation
  - Chapter 4
    - Linked List and STL list
- [Koffman & Wolfgang]
  - Chapter 4.5 to 4.8

Summary


User Programs
Uses
List ADT
insert()
remove()
retrieve()
Implements
Implements
Array
Linked List
Applications
List
Manipulation is possible for all positions
Implementations

Summary

- List ADT
  - Usage
  - Specification

- Implementation of List ADT
  - Array Based
    - Pros and Cons
  - Linked List Based
    - Pros and Cons