CS2040C Data Structures and Algorithms

List ADT and its implementations

Outline

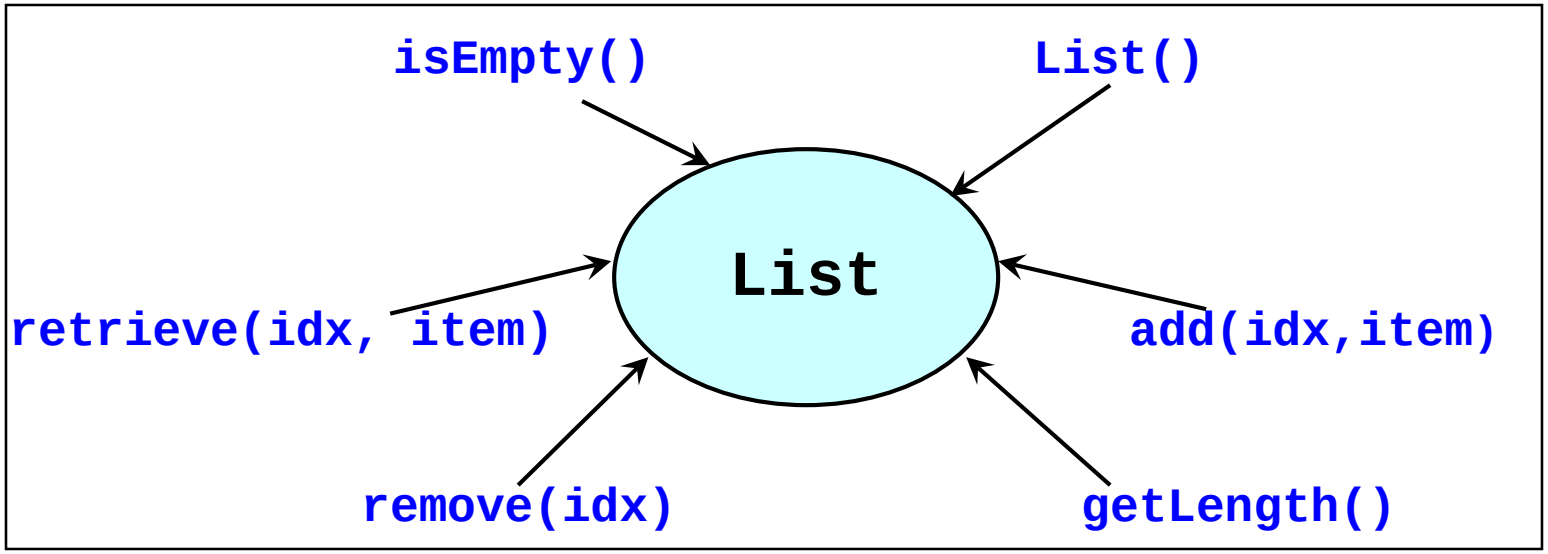
- List ADT
  - Specification
- Implementations for List ADT
  - Array Based
  - Linked List Based
    - Variations of Linked List
- Linked List in STL

List ADT

Is your name on the guest list?

List ADT

- A sequence of items where positional order matters $<a_1, a_2, \ldots, a_{n-1}, a_n>$
- 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

```
template <typename T>
class List {
    public:
        List();

        bool isEmpty() const;
        int getLength() const;

        void insert(int index, const T& newItem)
            throw (SimpleException);

        void remove(int index)
            throw (SimpleException);

        void retrieve(int index, T& dataItem) const
            throw (SimpleException);

    private:
        //Implementation dependent
        // See subsequent implementation slides
};  // end List class
```

A template class

Use the *SimpleException* class in previous lecture

Some design decisions

- Why template class?
  - List can be used on a wide range of data types
  - e.g. integer list, string list, even BankAcct list!
- Why exception?
  - Force the user of List class to handle wrong usage
- Why split the template class?
  - splitting the header + implementation highlights the **specification** and the **implementation** of an ADT

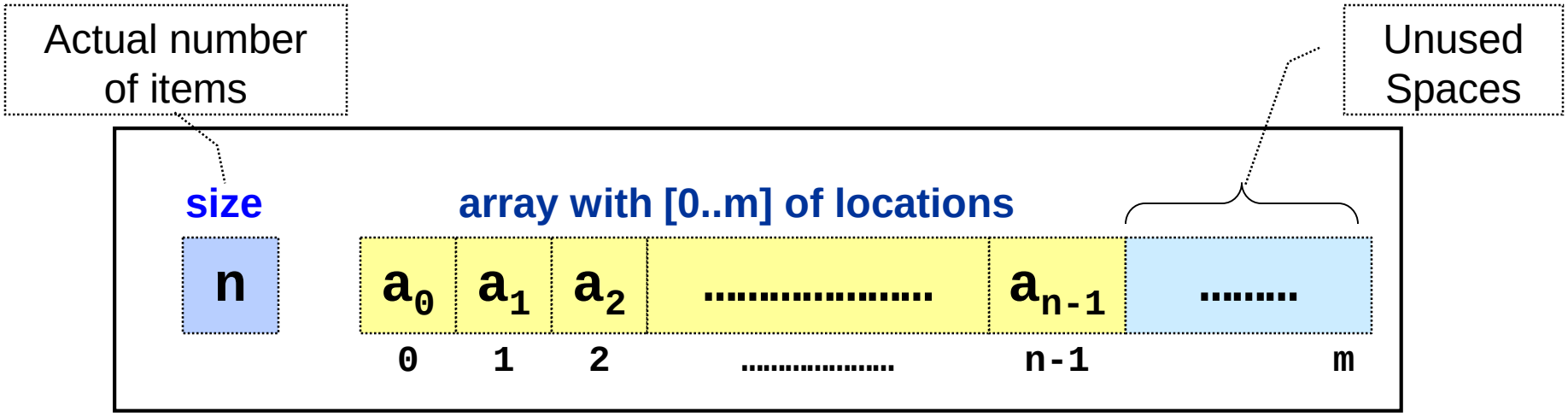
    (Need to include only the .cpp file in the user program)
- Alternatively, when you code a template class, you can combine the specification and implementation into a single .h file (inclusion model)

  (user only has to include the .h file)

List ADT using Array

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

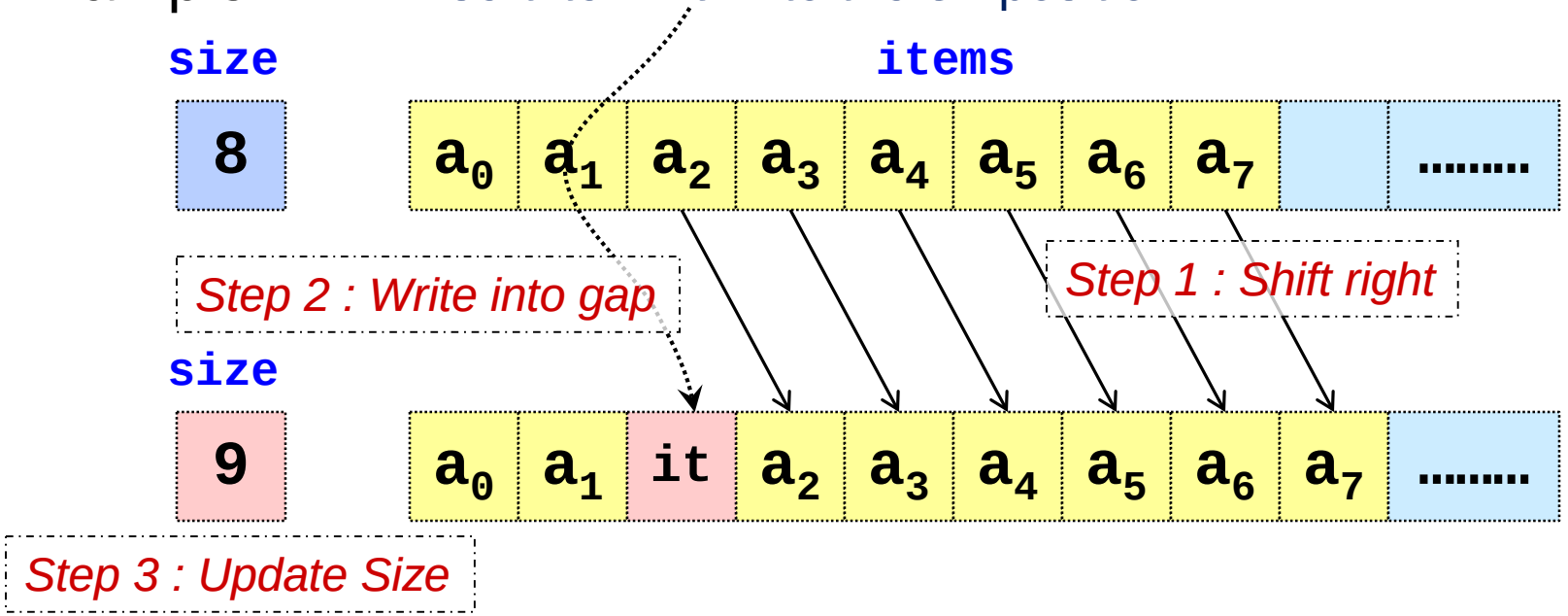


Internal of the **`list`** ADT, Array Version

Insertion: Using Array

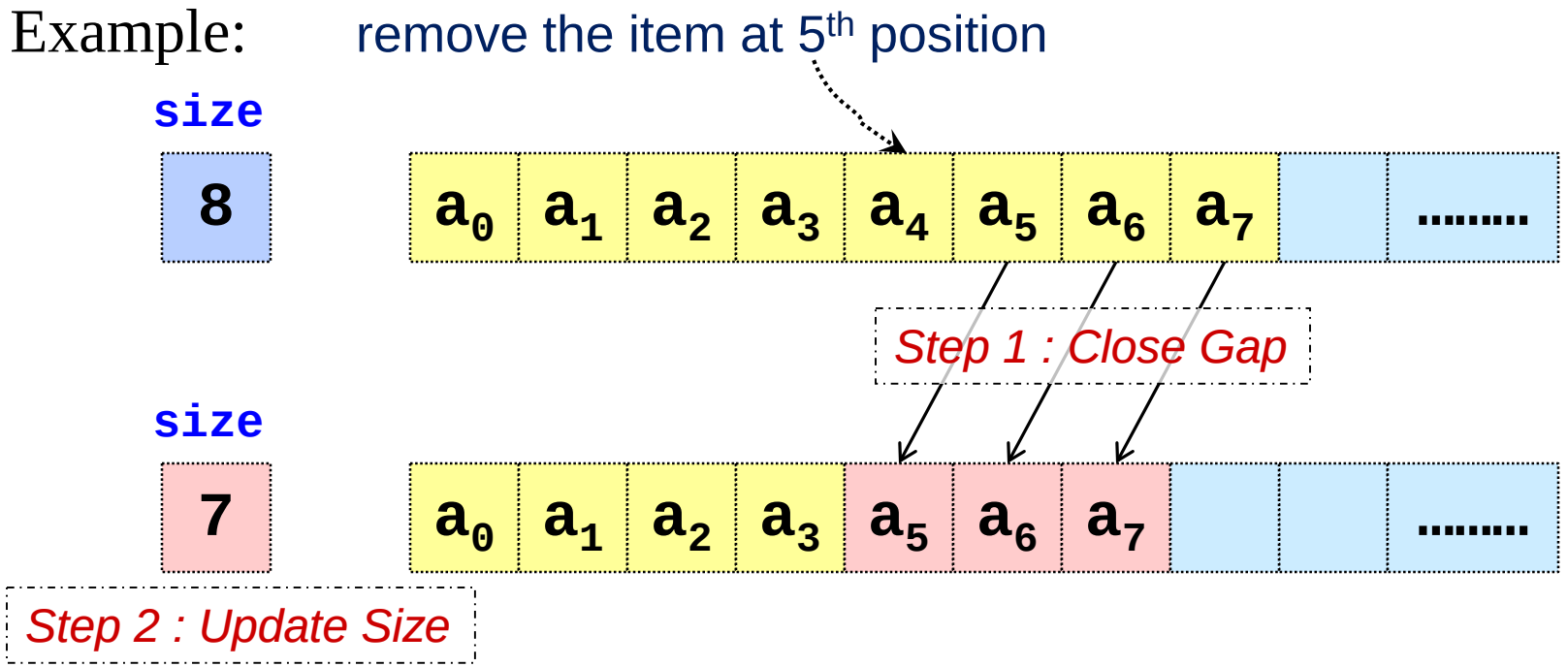
- **Simplest Case:** Insert to the end of array
- Other Insertions:
  - Some items in the list need to be shifted
  - **Worst case:** Inserting at the head of array

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



Deletion: Using Array

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



List ADT (Array): C++ Specification

```
const int MAX_LIST = 50;

template <typename T>
class List {
public:
    List();
    bool isEmpty() const;
    int getLength() const;
    void insert(int index, const T& newItem)
         throw (SimpleException);

    void remove(int index)
         throw (SimpleException;

    void retrieve(int index, T& dataItem) const
         throw (SimpleException);

private:

   T _items[MAX_LIST];

   int _size;

};  // end List class
```

Maximum number of items = 50

Methods from slide 5. No change.

Items stored in an array

Number of items

ListA.h

Implement List ADT (Array): 1/4

```
#include "ListA.h"

template <typename T>
List<T>::List()
{
    _size = 0;
}

template <typename T>
bool List<T>::isEmpty() const
{
    return _size == 0;
}

template <typename T>
int List<T>::getLength() const
{
    return _size;
}
```

ListA.cpp

This syntax indicates that **isEmpty()** method belongs to the template class **List<T>**

Implement List ADT (Array): 2/4

```
template <typename T>
void List<T>::retrieve(int userIdx, T& dataItem) const
   throw (SimpleException)
{
    int index = userIdx - 1;

    if ((index >= 0) && (index < _size) )
        dataItem = _items[index];

    else
        throw SimpleException("Bad Index!");
}
```

User counts from 1 to N, but array indices range from 0 to N-1. Need to convert.

Index is within range

Out of range index

ListA.cpp

- Exception message should be more meaningful:
  - e.g. “Bad index in **`retrieve()`** method”
  - A shorter message is used to conserve slide space

Implement List ADT (Array): 3/4

```
template <typename T>
void List<T>::insert( int userIdx, const T& newItem )
   throw (SimpleException)
{
    int index = userIdx - 1;

    if (_size >= MAX_LIST)
        throw SimpleException("List is full in insert()!");

    if ( (index>=0) && (index < _size + 1) ) {

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

        _items[index] = newItem;

        ++_size;

    } else
        throw SimpleException("Bad Index in insert()!");
}
```

Shift item(s) to the right

Insert new item

Increase size

ListA.cpp

Implement List ADT (Array): 4/4

```
template <typename T>
void List<T>::remove( int userIdx )
    throw (SimpleException)
{
    int index = userIdx - 1;

    if ((index >= 0) && (index < _size)){

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

        --_size;

    } else
        throw SimpleException("Bad Index in remove()!");
}
```

Shift item(s) to the left

decrease size

ListA.cpp

List ADT: Sample User Program 1/2

```
#include <iostream>
#include "ListA.cpp"

int main()
{
    List<int> intList;
    int rItem;

    try {
        intList.insert(1, 333);
        intList.insert(1, 111);
        intList.insert(3, 777);
        intList.insert(3, 555);

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

A list of integers

Prepare to catch exceptions

Several insertions to try out the **`insert()`** method

List ADT: Sample User Program 2/2

```
        //continue from previous slide
        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;

    } catch (SimpleException sExcpt) {
        cout << sExcpt.getMessage() << endl;
    }

}
```

Several deletions to try out the **`remove()`** method

- Not a very exciting program ☺
  - intended to test the ADT implementation

Efficiency (time) of Array Implementation

- **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

Efficiency (space) of Array Implementation

- 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*

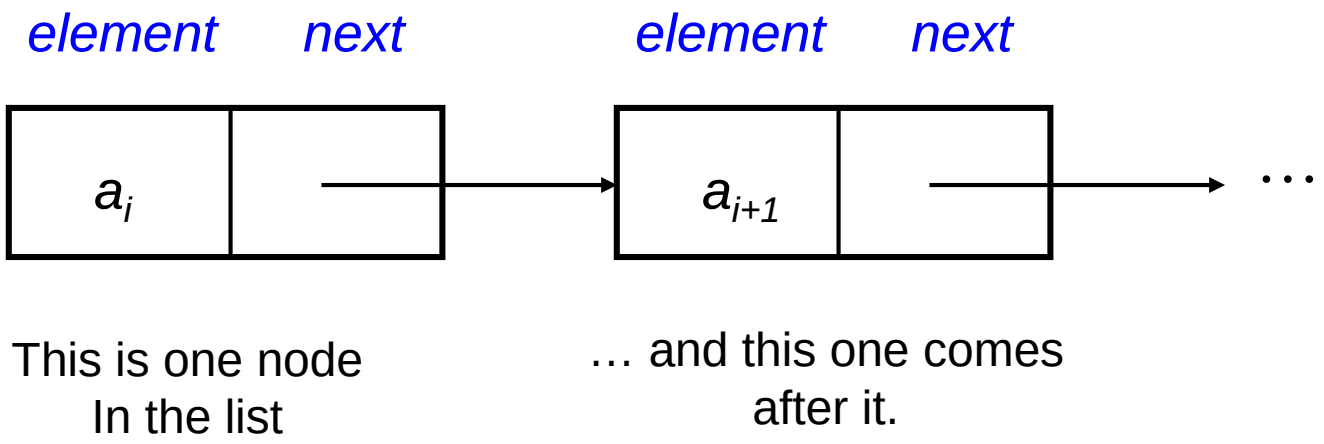
Observations about Array

- 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 using Linked List

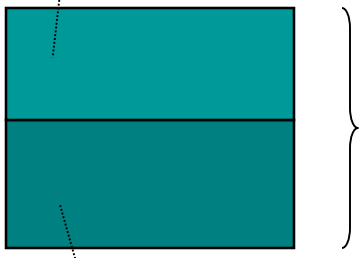
Implement List ADT using Linked List

- 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 element;
   ListNode *next;

};
```



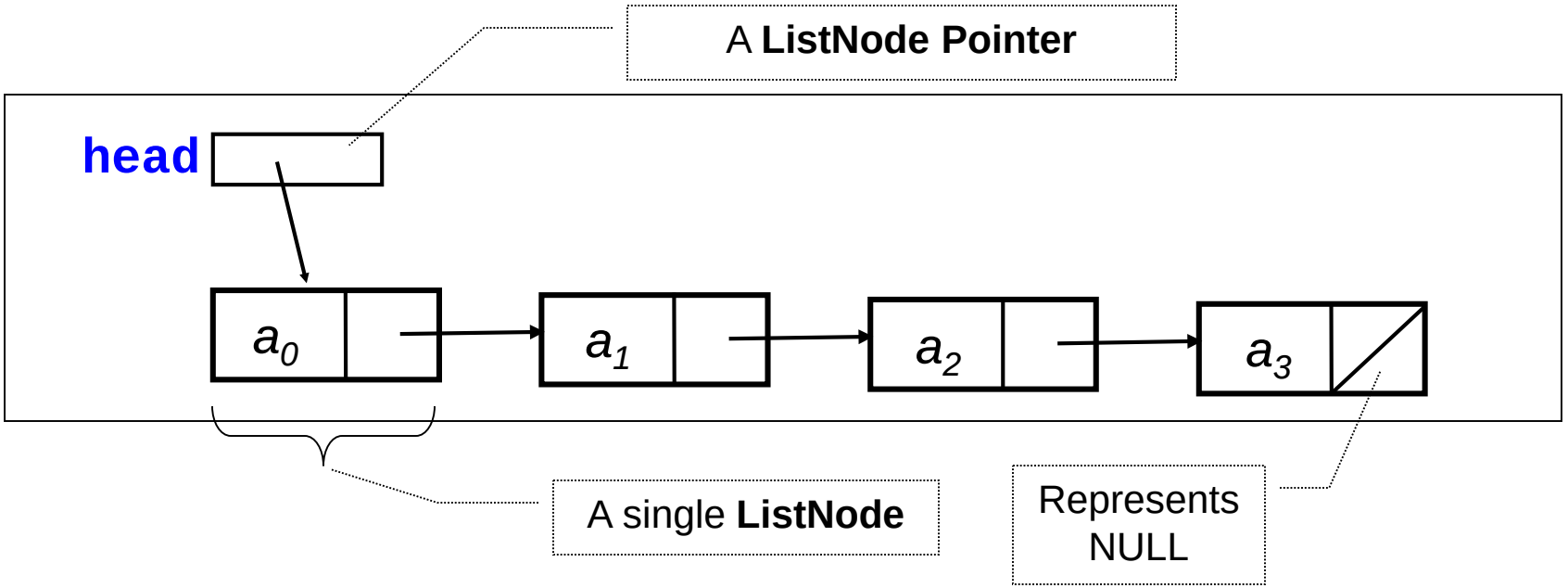
Store a single integer in this example

A pointer to another structure with the same layout

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

An example of a Linked List

- List of four items < a_0, a_1, a_2, a_3 >

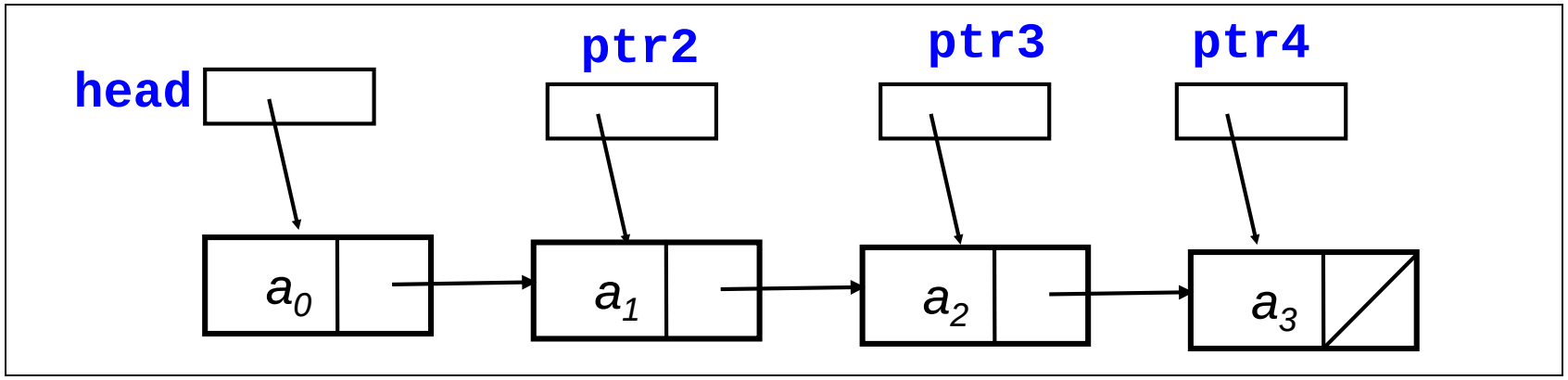


- We need:
  - **head** pointer to indicate the first node
  - **NULL** in the next pointer field of last node

Building a Linked List: An Example

- The previous list can be built with the following code

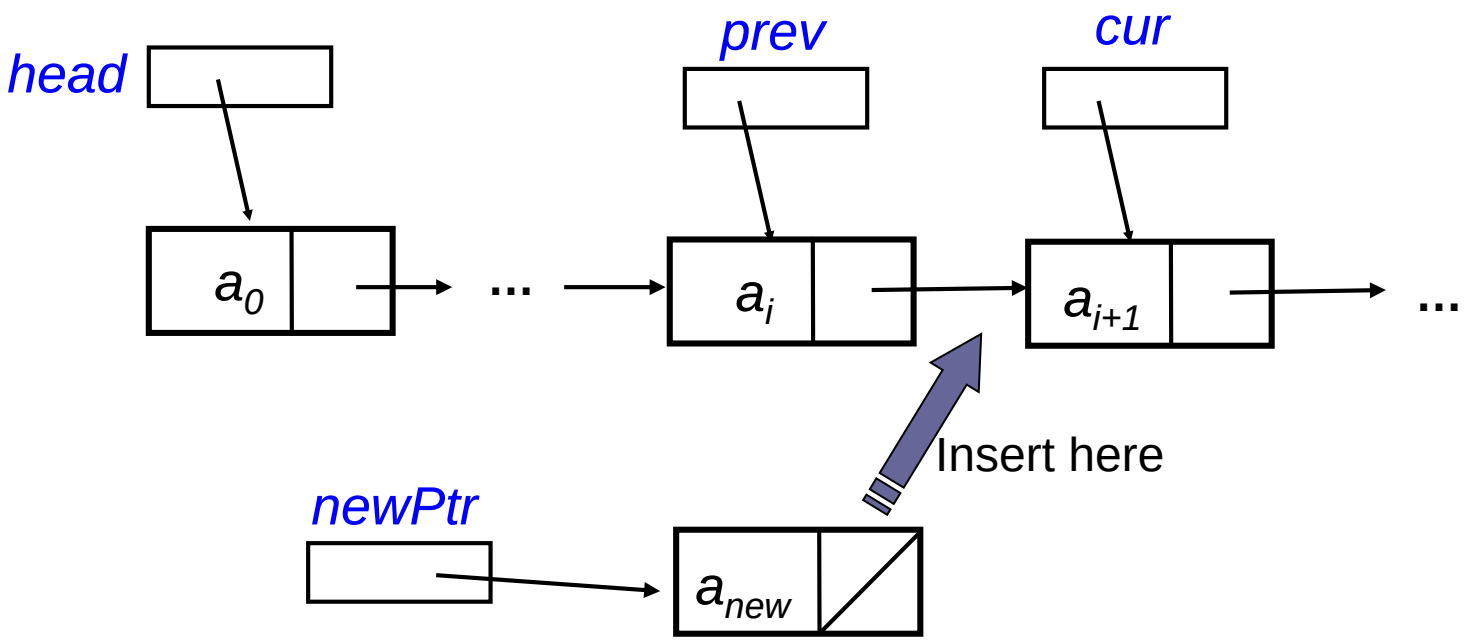
```
ListNode* ptr4 = new ListNode;
ptr4 ->element = a3;
ptr4 -> next = NULL;

ListNode* ptr3 = new ListNode;
ptr3 ->element = a2;
ptr3 -> next = ptr4;

ListNode* ptr2 = new ListNode;
ptr2 ->element = a1;
ptr2 -> next = ptr3;

ListNode* head = new ListNode;
head ->element = a0;
head -> next = ptr2;
```



- Do we need **ptr2**, **ptr3**, **ptr4 after** the list is built?

Insertion: Using Linked List

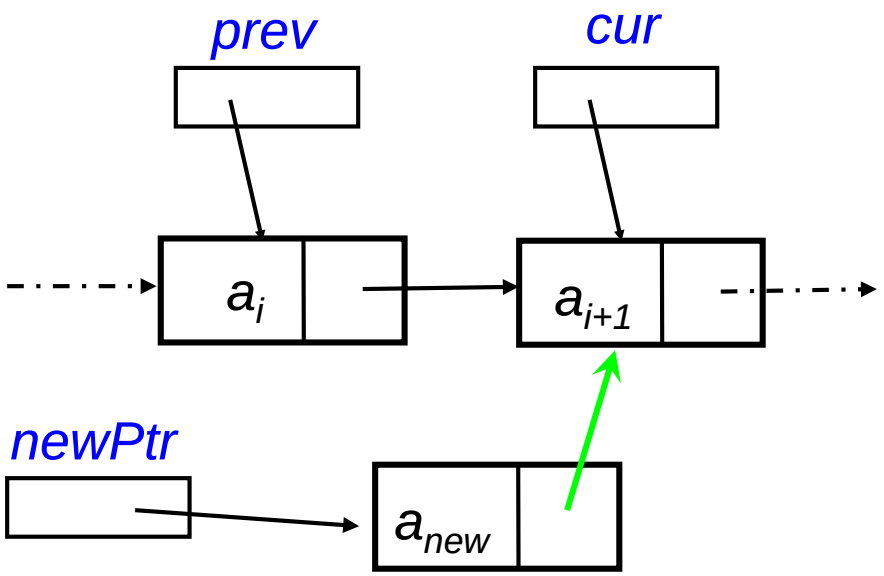
- Assume we have the following:
  - **`newPtr`** pointer:
    - Pointing to the new node to be inserted
  - **`prev`**, **`cur`** pointers:
    - Pointing to two consecutive nodes respectively
    - The new node is to be inserted in between



Insertion: Using Linked List

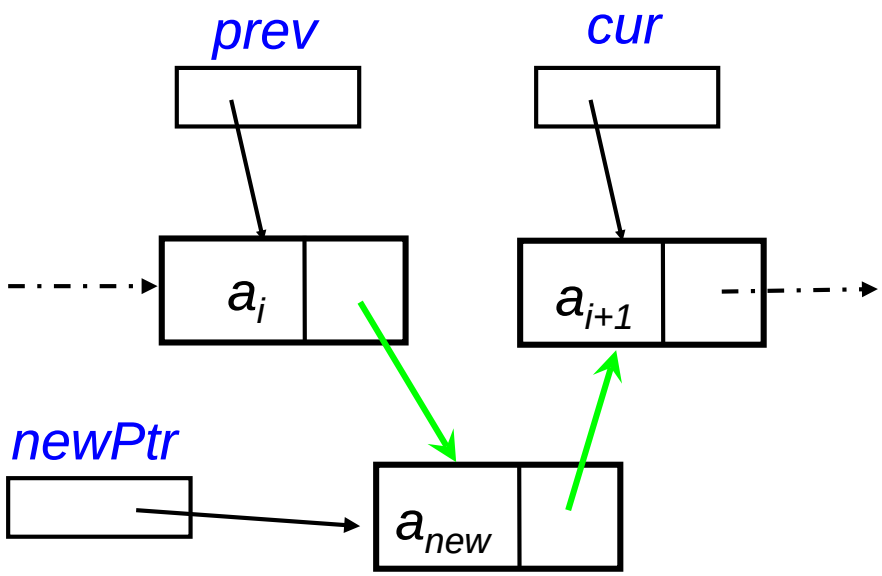
Step 1:

```
newPtr->next = cur;
```
OR
```
newPtr->next = prev->next;
```



Step 2:

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



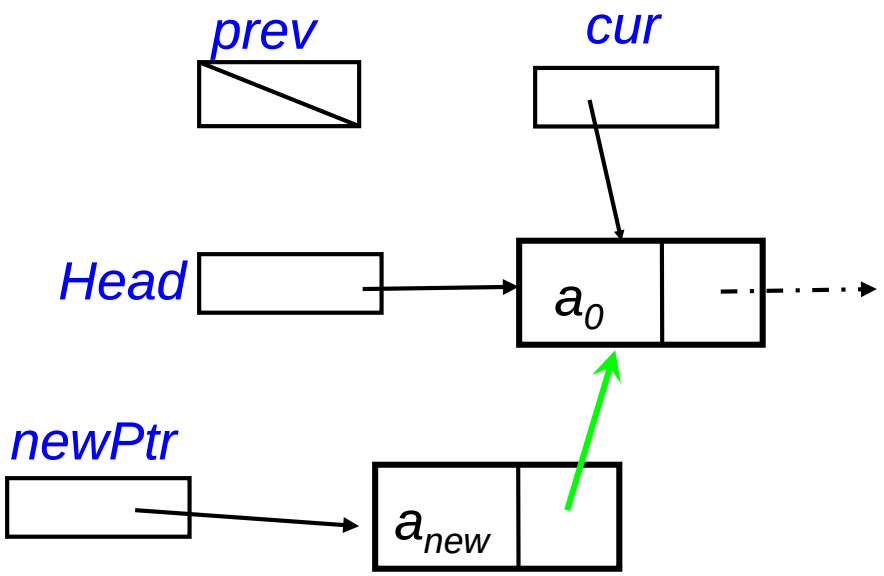
Question: Can we do insertion without the `cur` pointer?

Insertion at Head: Using Linked List

Step 1:

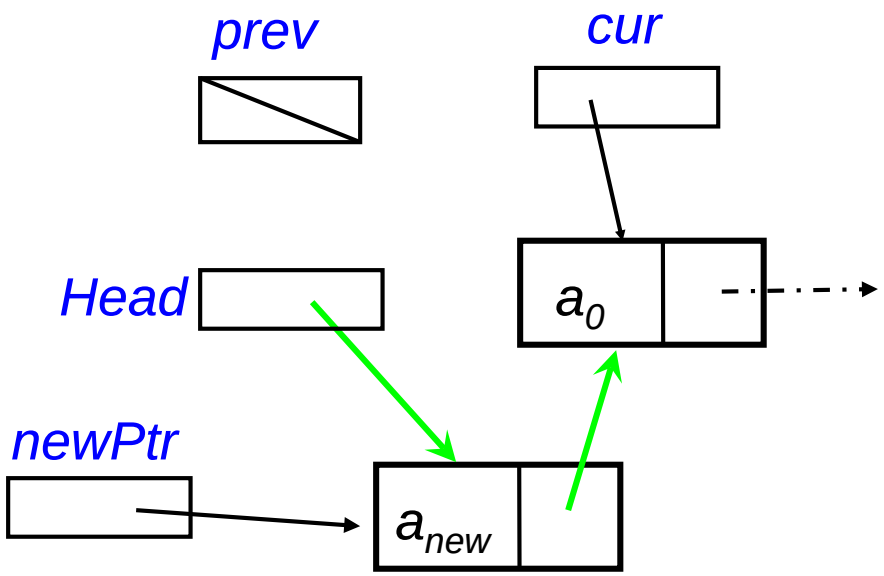
```
newPtr->next = cur;
```
OR
```
newPtr->next = head;
```

Why?



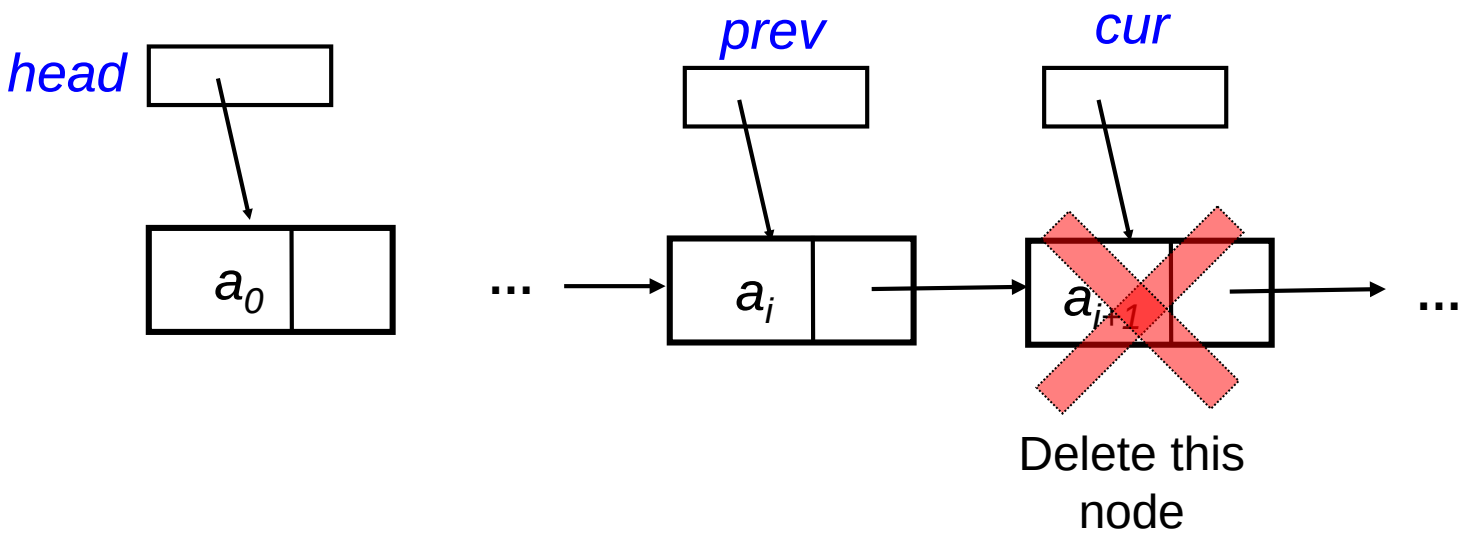
Step 2:

```
head = newPtr;
```



Deletion: Using Linked List

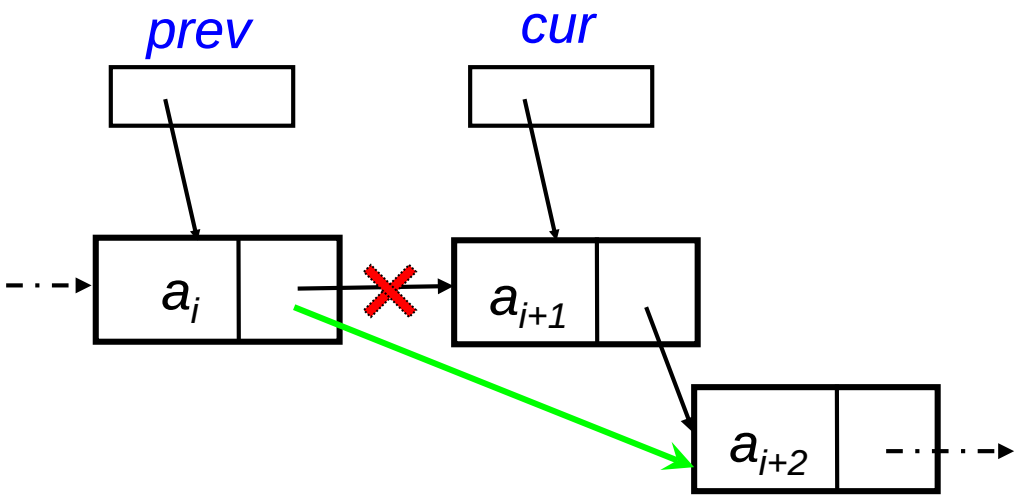
- Assume we have the following:
  - prev, cur pointers:
    - Pointing to two consecutive nodes respectively
    - cur points to the node to be deleted



Deletion: Using Linked List

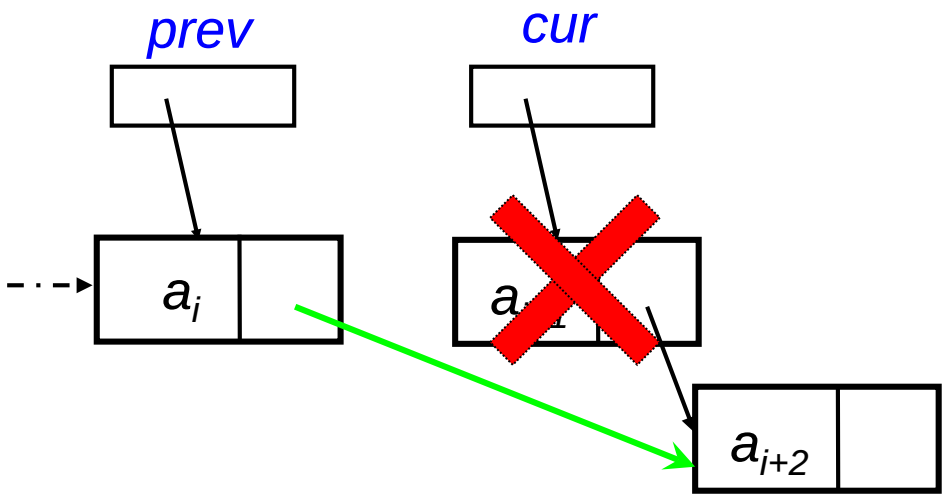
Step 1:

```
prev->next = cur->next;
```



Step 2:

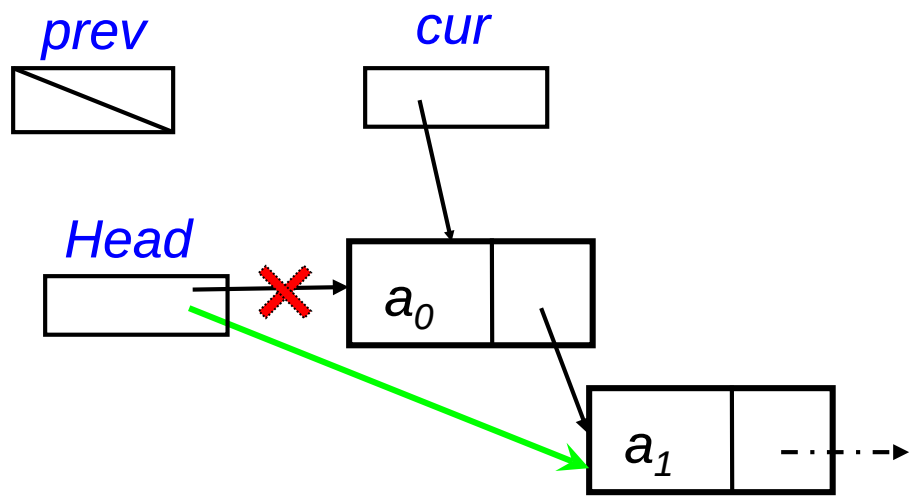
```
delete cur;
cur = NULL;
```



Deletion at Head: Using Linked List

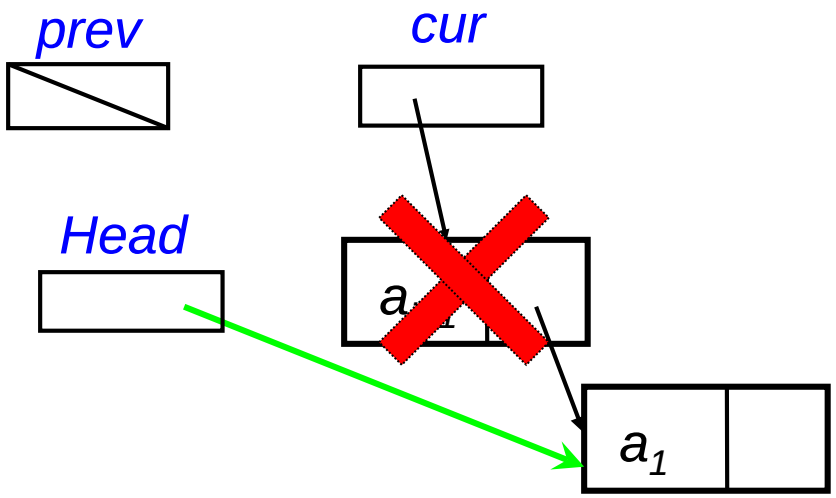
Step 1:

```
head = cur->next
OR
head = head->next;
```



Step 2:

```
delete cur;
cur = NULL;
```



Setting up **`prev`** and **`cur`** pointers

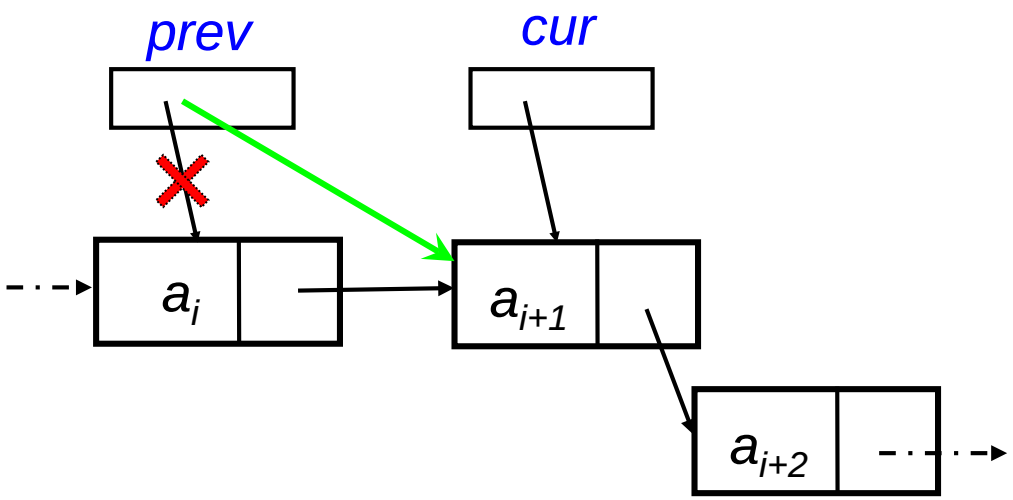
- We need a method to move along the list (traverse the list)
  - Set up **`prev`** and **`cur`** **pointers**
  - Stop when **`cur`** points to the target node
- Target node can be indicated by:
  - **Index** : Stop at Indexth node
  - **Value:** Stop at node with a particular value
- ***Pseudo-code*** for both versions will be discussed
- We will use the ***Index method*** for performing the traversal for implementation List ADT:
  - Index are provided for **`insert()`**, **`remove()`**, **`retrieve()`**

Traversing the Linked List

- To move forward one node

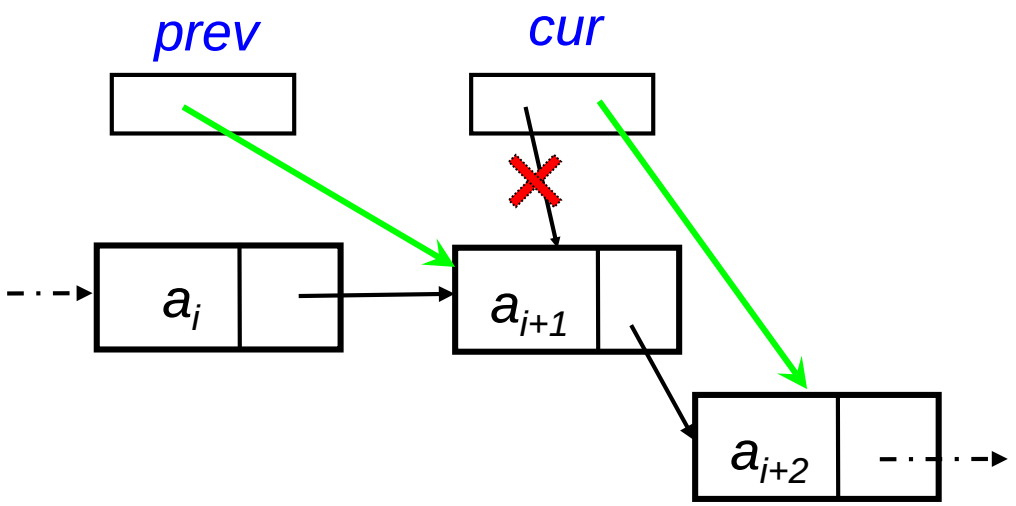
Step 1:

```
prev = cur;
```



Step 2:

```
cur = cur->next;
```



Traversing the Linked List using Index

```
ListNode* find(int index) const
/* Purpose: Return pointer to Index'th node in list */
/* Note: Not actual C++ code */
{
   ListNode *prev, *cur;

   prev = NULL;
   cur = Head of List;

   make sure 1 <= index <= length of list

   for (I = 1; I < index; I++){
        prev = cur;
        cur = next;
   }

   return cur;
}
```

- Question:
  - Do we need **`prev`** in this case?

Traversing the Linked List using Value

```
ListNode* findValue( int value ) const
/* Purpose: Return pointer to a node with value indicated in
   list */
/* Note: Not actual C++ code */

{
   ListNode *prev, *cur;

   prev = NULL;
   cur = Head of List;

   while (not found AND cur is not NULL){
        if ( cur->element == value ){
            found = true;
        } else {
            prev = cur;
            cur = next;
        }
   }

   return cur;
}
```

List ADT (Linked List): C++ Specification 1/2

```
template <typename T>
class List {
    public:
        List();
        ~List();

        bool isEmpty() const;
       int getLength() const;

       //The three main operations of List ADT
       void insert(int index, const T& newItem)
           throw (SimpleException);

       void remove(int index)
           throw (SimpleException);

       void retrieve(int index, T& dataItem) const
           throw (SimpleException);

   //... more declarations on next slide ...
```

Destructor declared

Methods from slide 5. No change.

ListP.h

List ADT (Linked List): C++ Specification 2/2

```
   //... ... continued from previous page

private:
    struct ListNode {
        T item;          //note the "T"
         ListNode *next;
    };

    int _size;

    ListNode*_head;

    ListNode* find(int index) const;


};  // end List class
```

Structure declaration can be private

Number of items

Pointer to the linked list

To locate a node given the index
This is also an example of private method.

ListP.h

Implement List ADT (Linked List): 1/8

```
template <typename T>
List<T>::List()
{   _size = 0;
    _head = NULL;
}

template <typename T>
List<T>::~List()
{
    while (!isEmpty())
        remove(1);
}

template <typename T>
bool List<T>::isEmpty() const
{    return _size == 0;      }

template <typename T>
int List<T>::getLength() const
{    return _size;           }
```

We need a destructor to return each node to the system.

ListP.cpp

Implement List ADT (Linked List): 2/8

```
template <typename T>
typename List<T>::ListNode* List<T>::find(int index) const
{
    if ( (index < 1) || (index > getLength()) )
        return NULL;

    else  // count from the beginning of the list.
    {
        ListNode*cur = _head;
        for (int skip = 1; skip < index; ++skip)
            cur = cur->next;
        return cur;
    }  // end if

}
```

ListP.cpp

- The next slide explains the syntax used above

Implement List ADT (Linked List): 3/8

```
template <typename T>
typename List<T>::ListNode* List<T>::find(int index) const
{
    //.. Body Not Shown
}
```

This is the return type of ***find()*** method. **List<T>::** is needed because **ListNode** is a **private declaration** in template class **List**.

typename is required to inform the compiler that **List<T>::ListNode** is a datatype

Normal syntax to indicate ***find()*** is a method in template class ***List<T>***

- So, the above simply means:

```
ListNode* find(int index) const
{    ...
```

Implement List ADT (Linked List): 4/8

```
template <typename T>
void List<T>::retrieve(int userIdx, T& dataItem) const
    throw (SimpleException)
{
   if ( (userIdx < 1) || (userIdx > getLength()) )
      throw SimpleException("Bad Index in retrieve()");

   else  {  // get pointer to node, then data in node

      ListNode *cur = find(userIdx);
      dataItem = cur->item;

   }
}
```

Use the **find()** method

ListP.cpp

Implement List ADT (Linked List): 5/8

```
template <typename T>
void List<T>::insert(int userIdx, const T& newItem)
   throw (SimpleException)
{
    int newLength = getLength() + 1;

    if ( (userIdx < 1) || (userIdx > newLength) )
        throw SimpleException("Bad Index in insert()");

    else {

        ListNode *newPtr = new ListNode;
        _size = newLength;
        newPtr->item = newItem;

    //Continue on next slide
```

Allocate a new node and initialize the value

ListP.cpp

Implement List ADT (Linked List): 6/8

```
    //continued from previous slide

    // attach new node to list
        if (userIdx == 1) {

            newPtr->next = _head;
            _head = newPtr;

        } else {

            ListNode *prev = find(userIdx-1);

            newPtr->next = prev->next;
            prev->next = newPtr;
        }

    } //end if
}
```

Special Case: Insert at head. See Slide 28

General Case: Other insertions. See Slide 27

ListP.cpp

Implement List ADT (Linked List): 7/8

```
template <typename T>
void List<T>::remove(int userIdx)
    throw (SimpleException)
{

    ListNode *cur;

    if ( (userIdx < 1) || (userIdx > getLength()) )
        throw SimpleException("Bad index in remove()");

    else {
        --_size;

        /* continue on next slide */
```

Reduce size

ListP.cpp

Implement List ADT (Linked List): 8/8

```
        //Continued from previous slide
         if (userIdx == 1) {

             cur = _head;  // save pointer to node
             _head = _head->next;

         } else {
             ListNode *prev = find(userIdx - 1);

             cur = prev->next;  // save pointer to node
             prev->next = cur->next;
         }  // end if

         cur->next = NULL;
         delete cur;
         cur = NULL;

    }  // end if
}
```

Special Case: Delete at head. See slide 31.

General Case: Other deletions. See slide 30.

Free memory space pointed by **cur** pointer

ListP.cpp

Linked List Variations

Variations on Linked List

- The linked list implementation shown is known as **singly linked list**
  - Each node has one pointer (a single link)
- Many other variations
  - Doubly Linked List
  - Circular Linked List
  - Dummy Head Node
  - Tailed Linked List
  - Etc
- Some variations can be combined:
  - Circular Doubly-Linked List
  - Circular Linked List with Dummy Head Node
  - etc

Doubly Linked List

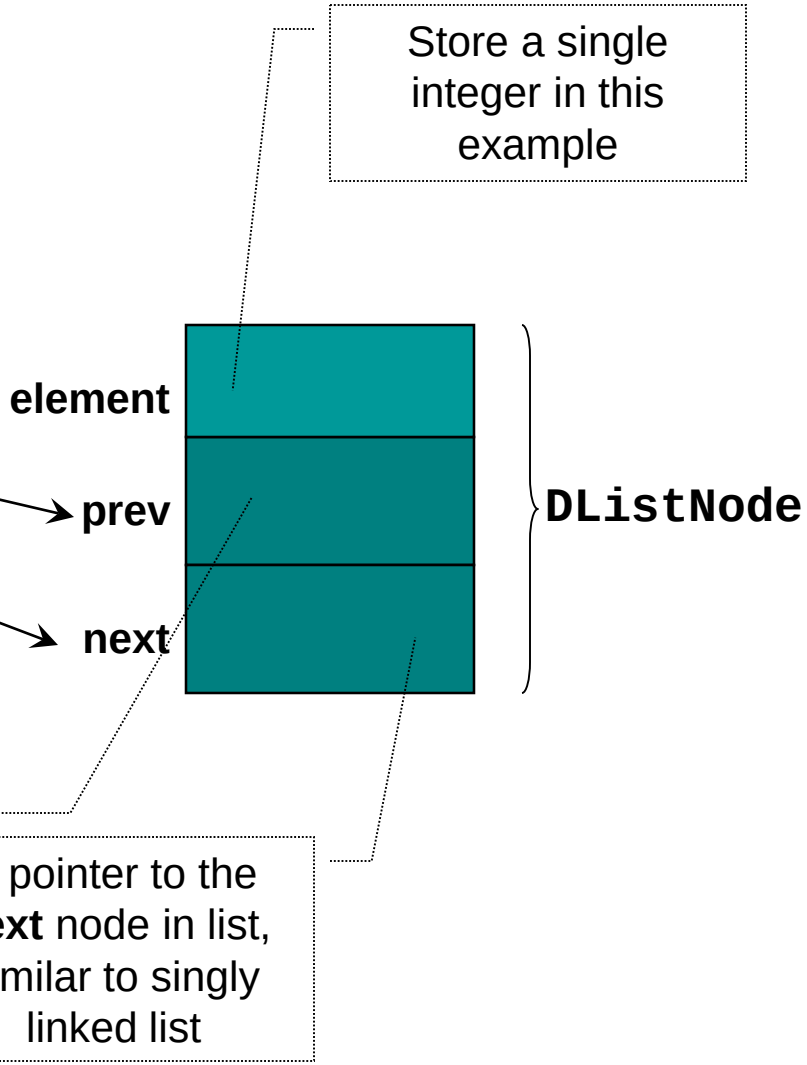
Two is better than one

Doubly Linked List: Motivation

- Singly Linked List only facilitates movement in one direction (from head to end of list)
  - Can get to next node in list easily
  - Cannot go to the previous node
  - The last node takes the longest time to reach
- Doubly Linked List facilitates movement in both directions
  - Can get to next node in list
  - Can get to previous node in list
  - Simplifies most of the methods

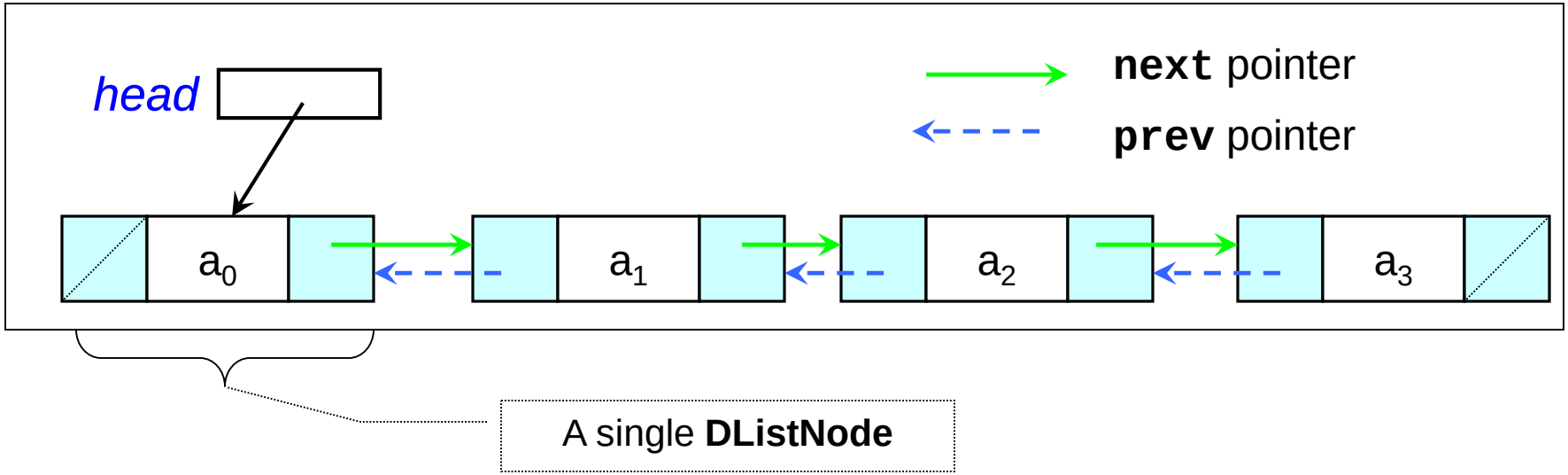
A single node in Doubly Linked List

```
struct DListNode {

   int element;
   DListNode *prev;
   DListNode *next;

};
```



An example of Doubly Linked List

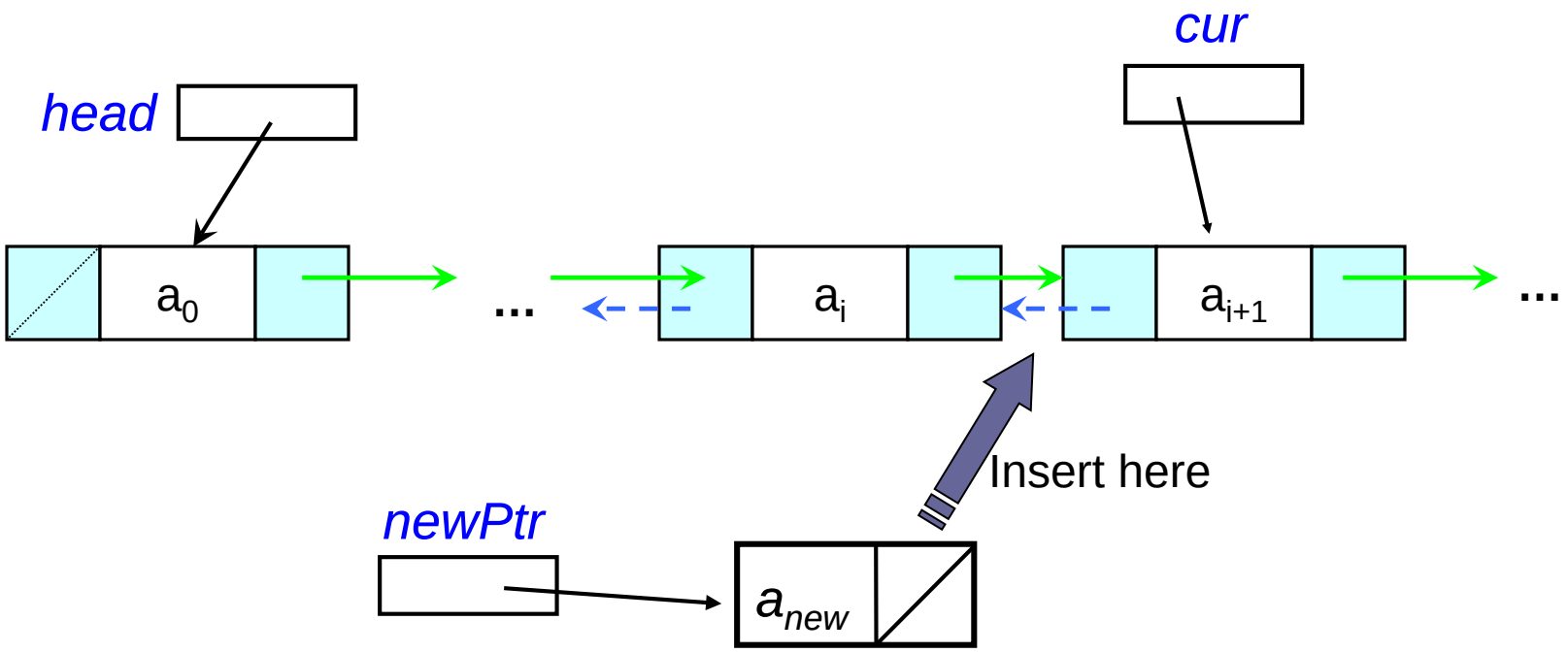
- List of four items < a_0, a_1, a_2, a_3 >



- We need:
  - **head** pointer to indicate the first node
  - **NULL** in the prev pointer field of first node
  - **NULL** in the next pointer field of last node

Insertion: Using Doubly Linked List

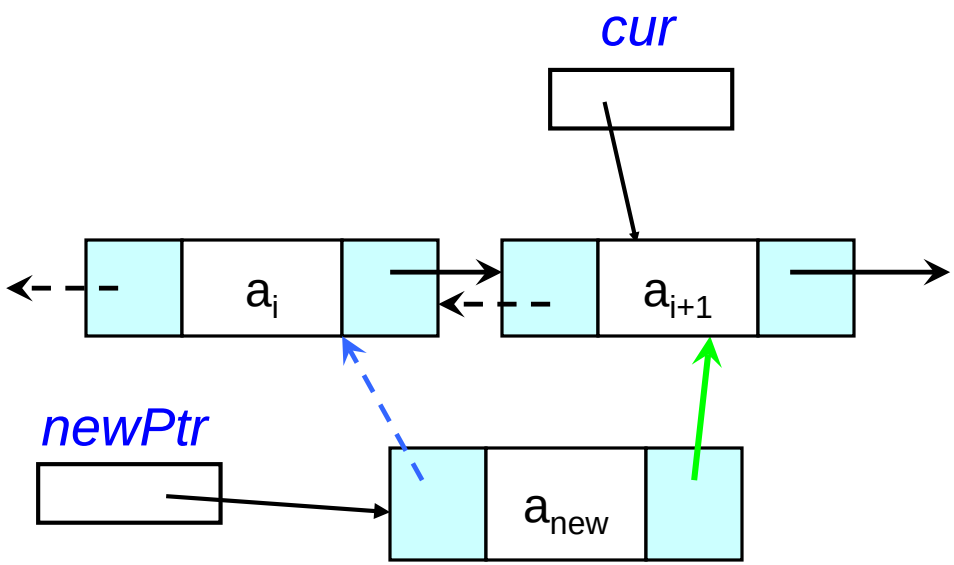
- Assume we have the following:
  - **`newPtr`** pointer:
    - Pointing to the new node to be inserted
  - **`cur`** pointers:
    - The new node is to be inserted **before** this node



Insertion: Using Doubly Linked List

Step 1:

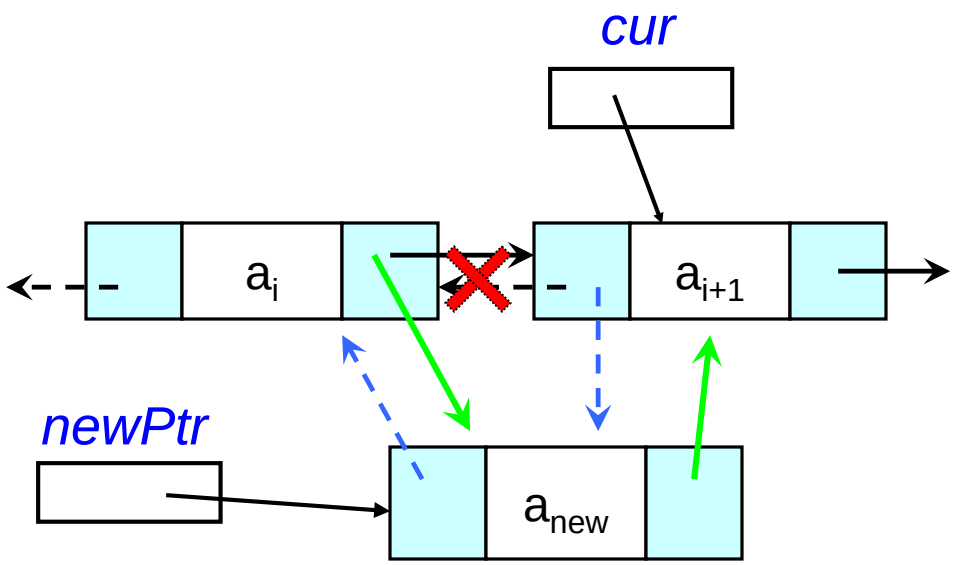
```
newPtr->next = cur;
newPtr->prev = cur->prev;
```



Step 2:

```
cur->prev->next = newPtr;
cur->prev = newPtr;
```

Any other alternatives?

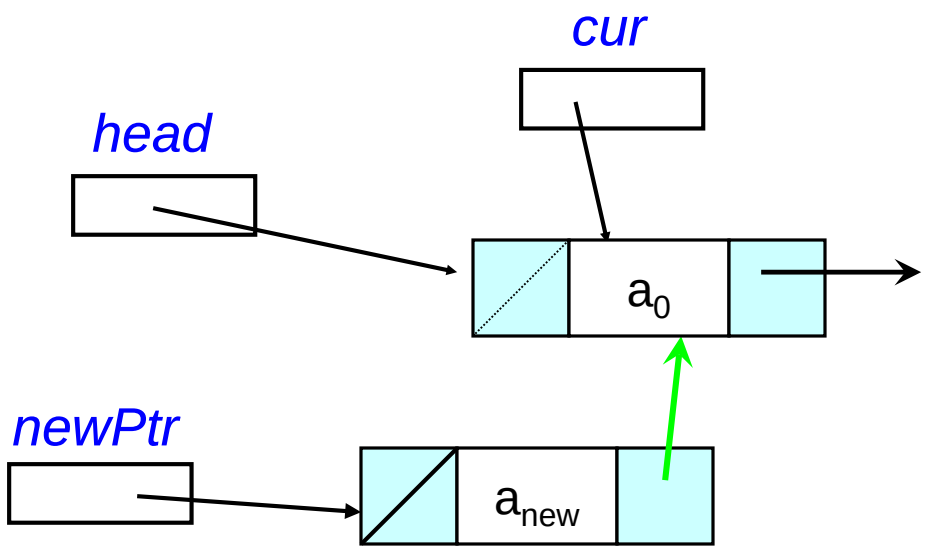


Insertion at Head: Using Doubly Linked List

Step 1:

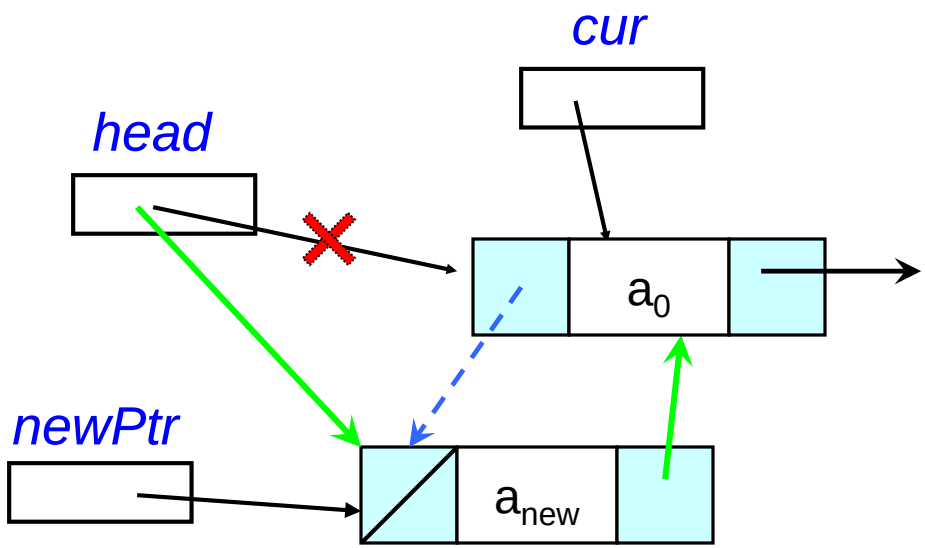
```
newPtr->next = cur;
newPtr->prev = NULL;
```

Question:
How do you know cur == head?



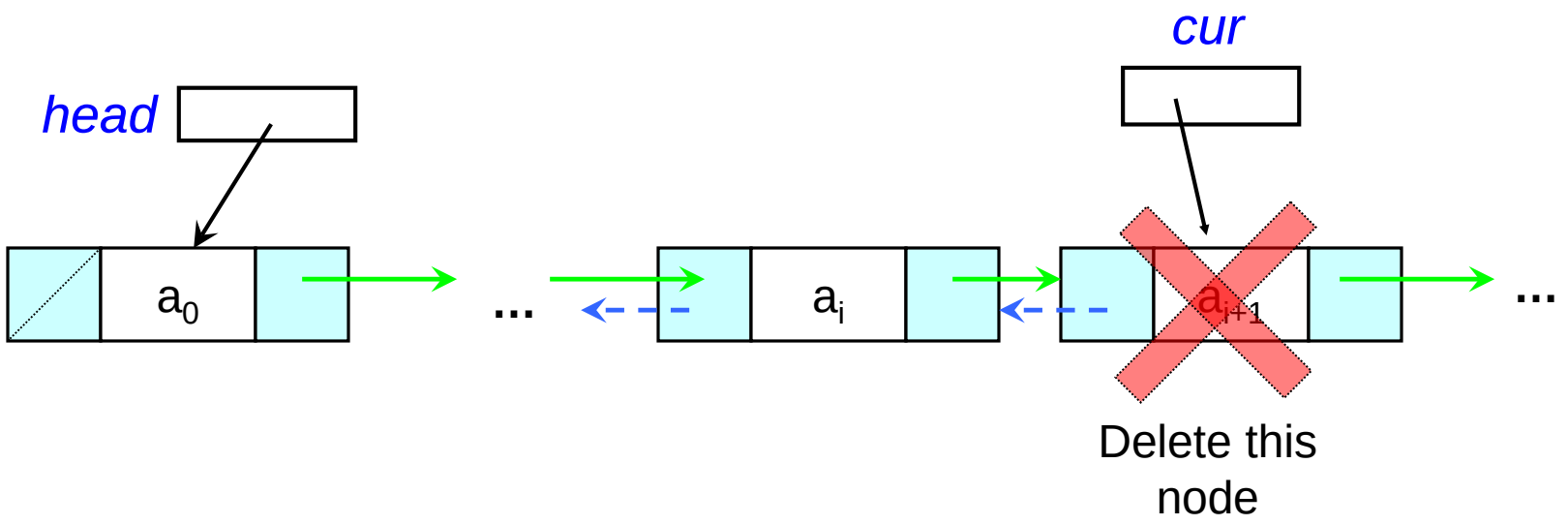
Step 2:

```
cur->prev = newPtr;
head = newPtr;
```



Deletion: Using Doubly Linked List

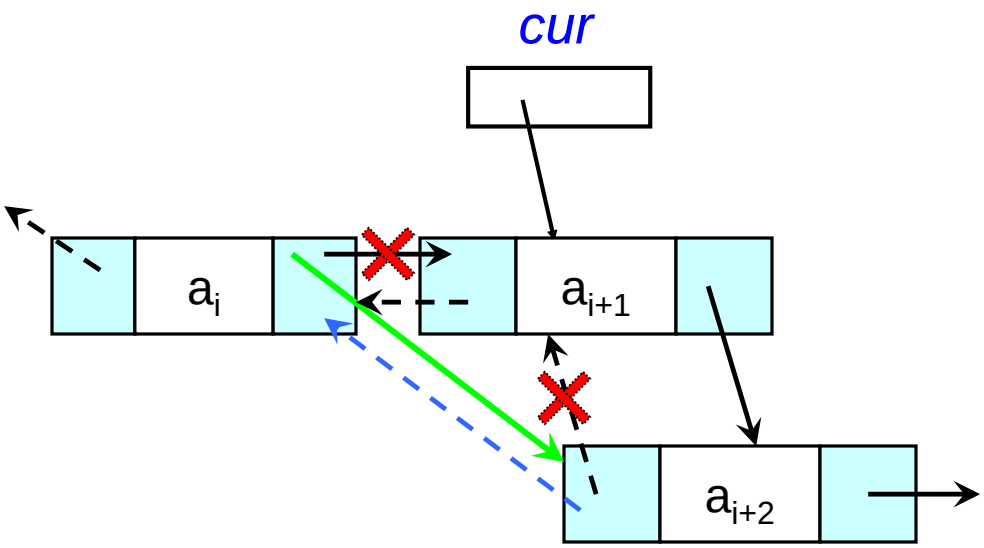
- Assume we have the following:
  - **`cur`** pointer:
    - Points to the node to be deleted



Deletion: Using Doubly Linked List

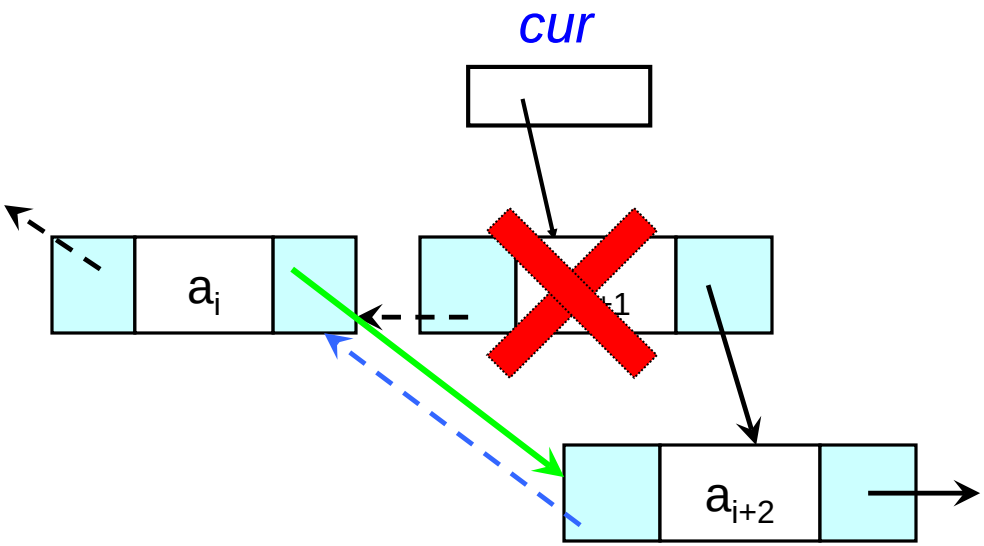
Step 1:

```
cur->prev->next = cur->next;
cur->next->prev = cur->prev;
```



Step 2:

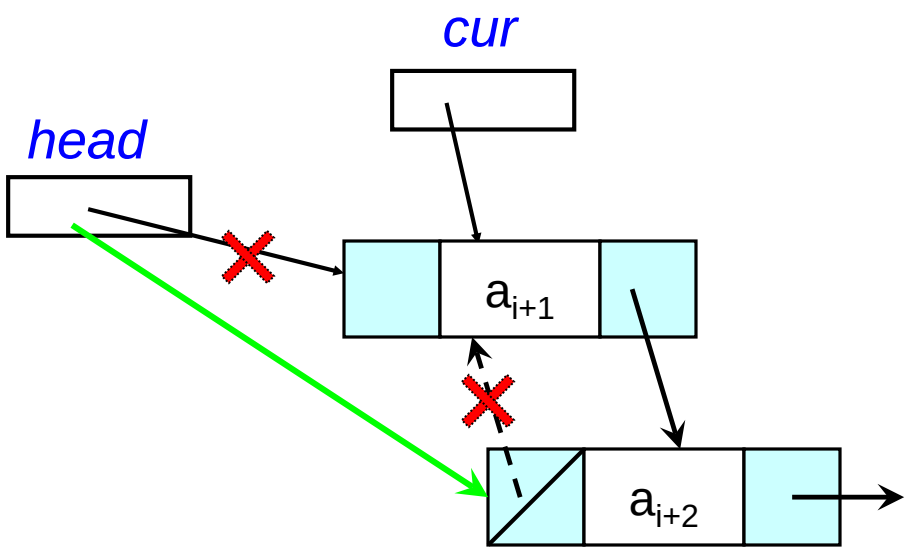
```
delete cur;
cur = NULL;
```



Deletion at head: Using Doubly Linked List

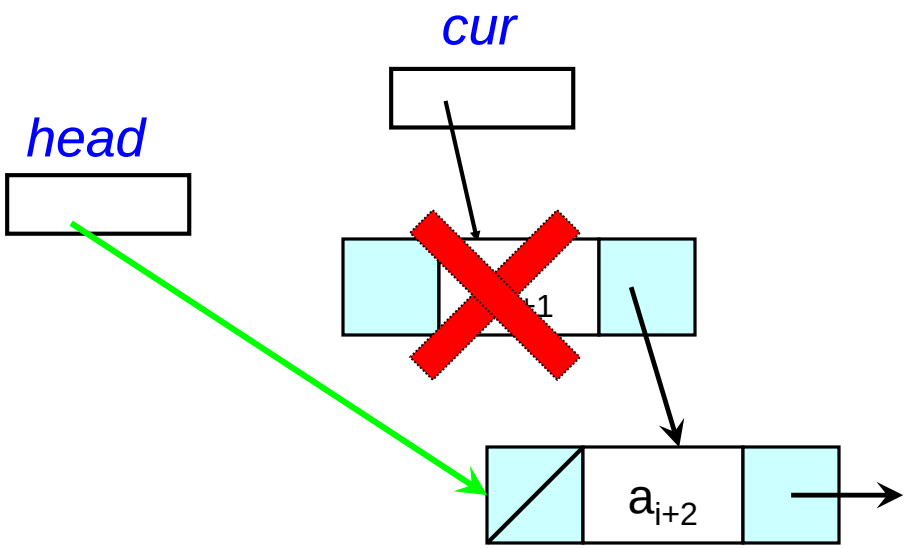
Step 1:

```
head = cur->next;
cur->next->prev = NULL;
```



Step 2:

```
delete cur;
cur = NULL;
```



List ADT (Doubly Linked List): C++ Specification

```
//ListDLL.h: List ADT using Doubly Linked List

template <typename T>
class List {
public:
   /* Similar to List ADT using Linked List */
   ... ... ...
   /* other methods not shown */

private:
   struct DListNode {
       T item;
       DListNode *prev;
       DListNode *next;
   };

   int size;

   DListNode *head;

   DListNode* find(int index) const;
};  // end List class
```

New Structure

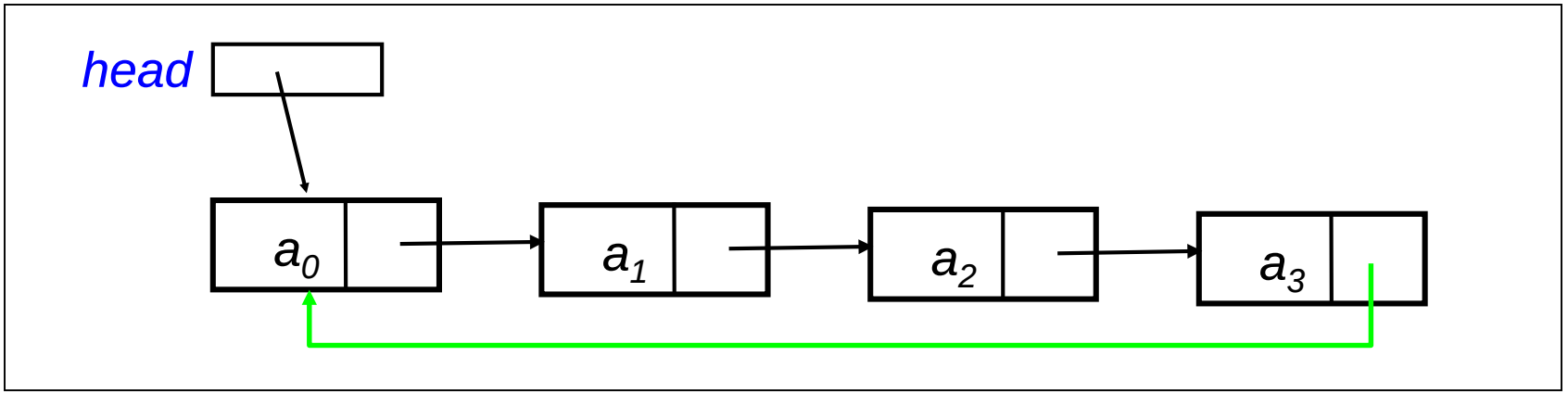
Slight change in type

Circular Linked List

Go round and round

Circular Linked List

- Just a simple addition:
  - The last node in singly linked list points back to the first node



- There is no need to change the `ListNode` structure at all
- There is no NULL anywhere in the list

Circular Linked List: Motivation

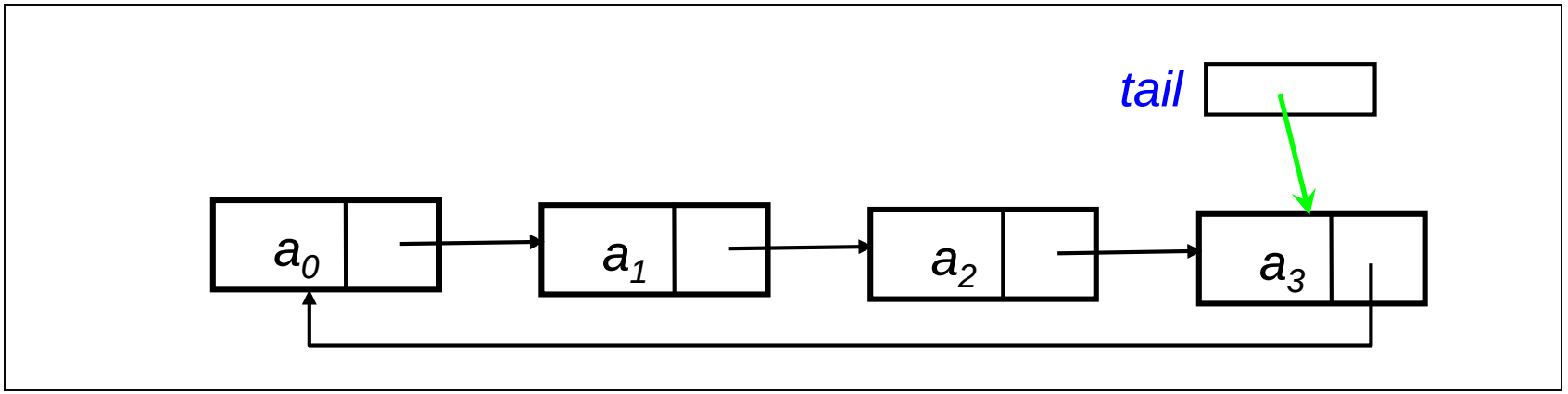
- Useful when we need to repeatedly go through the list
  - "Real World" example: Suppose your TA needs to repeatedly go through the name list until every student has attempted
- How do we know we have passed through the list once (i.e. visited every node)?

```
cur = head
do {
     visit the node cur points to
     cur = cur->next;
} while (cur != head);
```

Simple solution as long as the list is not empty

Circular Linked List: Motivation

- Even more useful if we keep track of the **tail** of the list instead of the head:



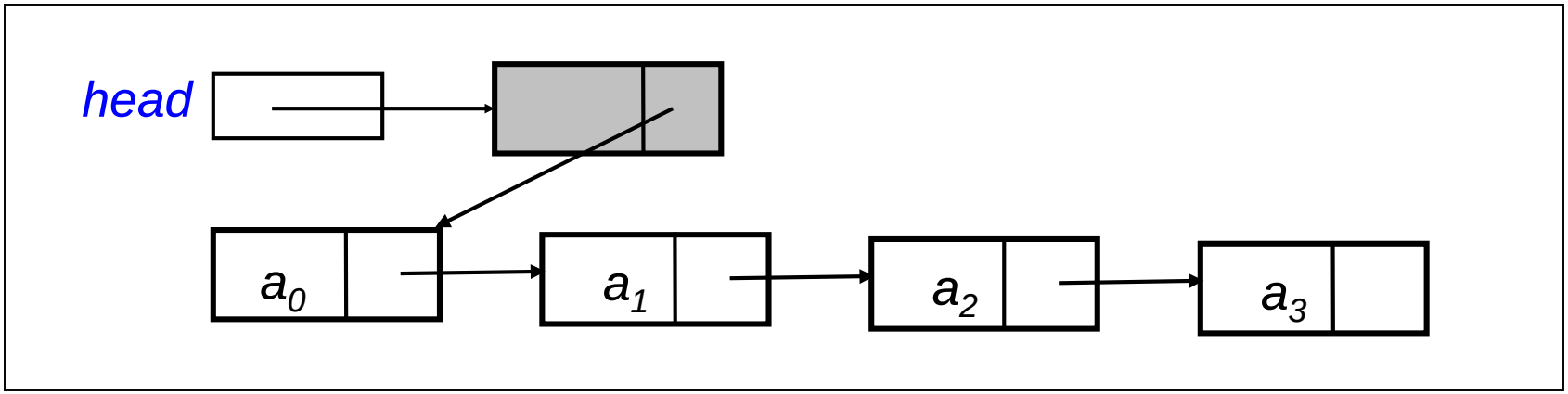
- Can access both the tail and head easily:
  - … how do we get the head?

Dummy Head Node

There is a dummy at the head!!

Dummy Head Node

- There is an extra node at the beginning of the list:
  - It is **not used** to store real element, hence the name **dummy**
  - Simplify the **`insert()`, `remove()`** such that there is no more special case



Linked List in STL

You mean I don’t need to code?!?!

Linked List in STL

- Linked List has a standard implementation in STL
- Header File:

  `#include <list>`
- Basic Idea:
  - Uses **`iterator`** to refer to locations in the linked list
  - Provide a standard set of manipulation methods
    - **`insert(), erase()`**etc for adding/removing items
    - **`front(), back()`** etc for accessing items
    - **`begin(), end()`** etc to give access to well defined iterators
    - Refer to reference text or website for more details

STL List: Simple Example

```
#include <list>
...
int main()
{
   list<double> dl;

   dl.insert( dl.begin(), 3.14 );
   dl.insert( dl.begin(), 1.23 );

   list<double>::iterator li = dl.begin();

   cout << *li << endl;
   li++;
   cout << *li << endl;

   return 0;
}
```

dl is a linked list of double values

insert at head

insert infront of 3.14

Use iterator to access items

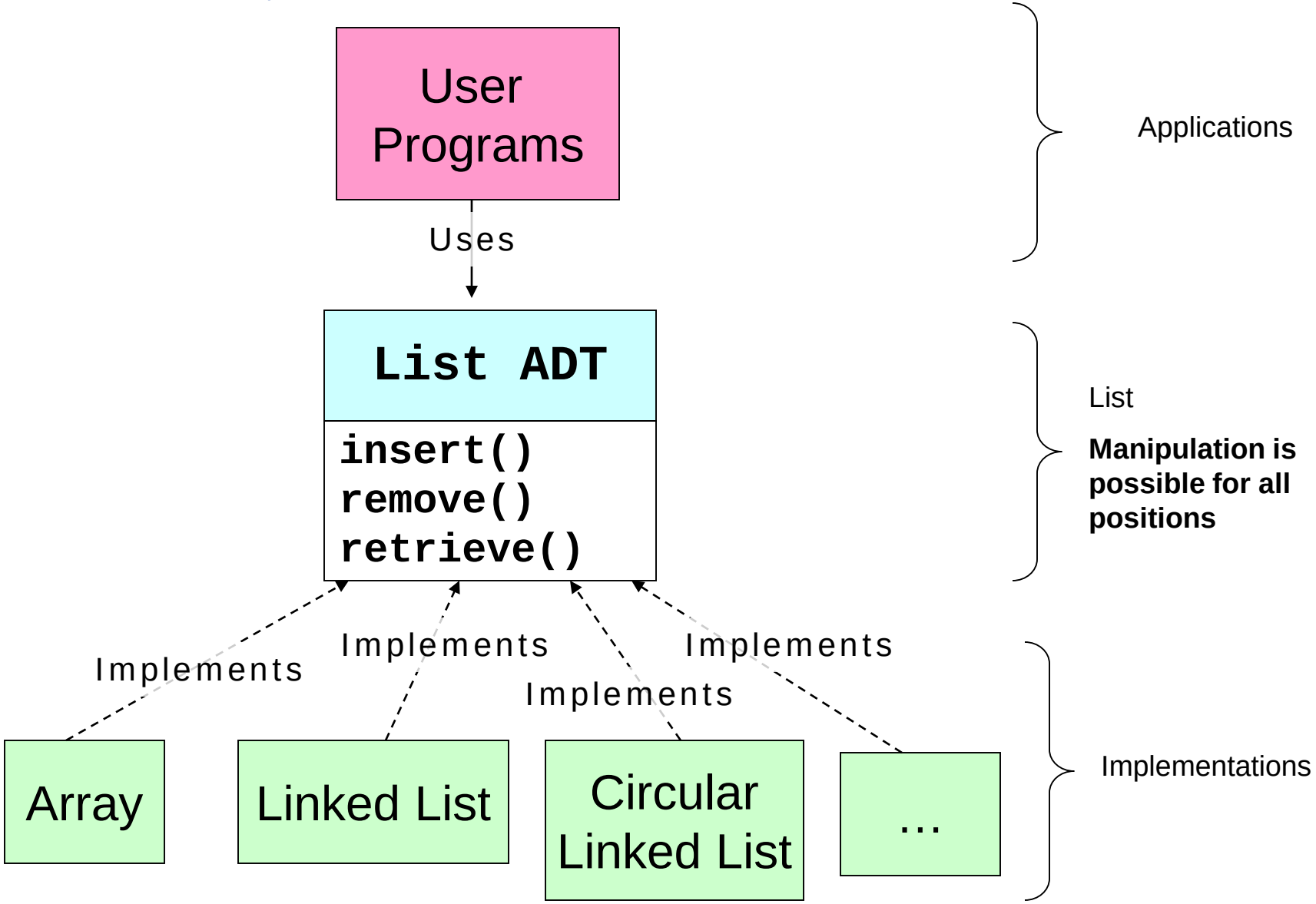
Output:

```
1.23
3.14
```

References

- Carrano's Book
  - Chapter 3
    - List ADT and array based implementation
  - Chapter 4
    - Linked List and STL list

Summary


User Programs
Uses
List ADT
insert()
remove()
retrieve()
Implements
Implements
Implements
Implements
Array
Linked List
Circular 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
    - Variations of Linked List
- STL List