CS2040 Data Structures and Algorithms

Lecture Note #5

List ADT – Array Lists and Linked Lists

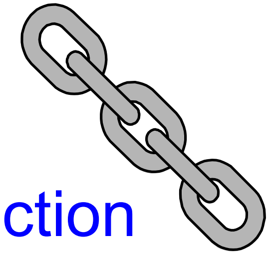
Outline

1. Use of a List (Motivation)
   - List ADT
2. List ADT Implementation via Array
   - Adding and removing elements in an array
   - Time and space efficiency
3. List ADT Implementation via Linked Lists
   - Linked list approach
   - ListNode class: forming a linked list with ListNode
   - BasicLinkedList
4. More Linked Lists
   - EnhancedLinkedList, TailedLinkedList
5. Other Variants
   - CircularLinkedList, DoublyLinkedList
6. Java API: LinkedList class
7. Summary

1 Use of a List

Motivation

Motivation



- **List** is one of the most basic types of data collection
  - For example, list of groceries, list of modules, list of friends, etc.
  - In general, we keep items of the same type (class) in one list
- Typical Operations on a data collection
  - Add data
  - Remove data
  - Query data
  - The details of the operations vary from application to application. The overall theme is the management of data

ADT of a List (1/3)

- A list ADT is a dynamic linear data structure
  - A collection of data items, accessible one after another starting from the beginning (head) of the list
- Examples of List ADT operations:
  - Create an empty list
  - Determine whether a list is empty
  - Determine number of items in the list
  - Add an item at a given position
  - Remove an item at a position
  - Remove all items
  - Read an item from the list at a position
  - …

ADT of a List (2/3)

ListInterface.java

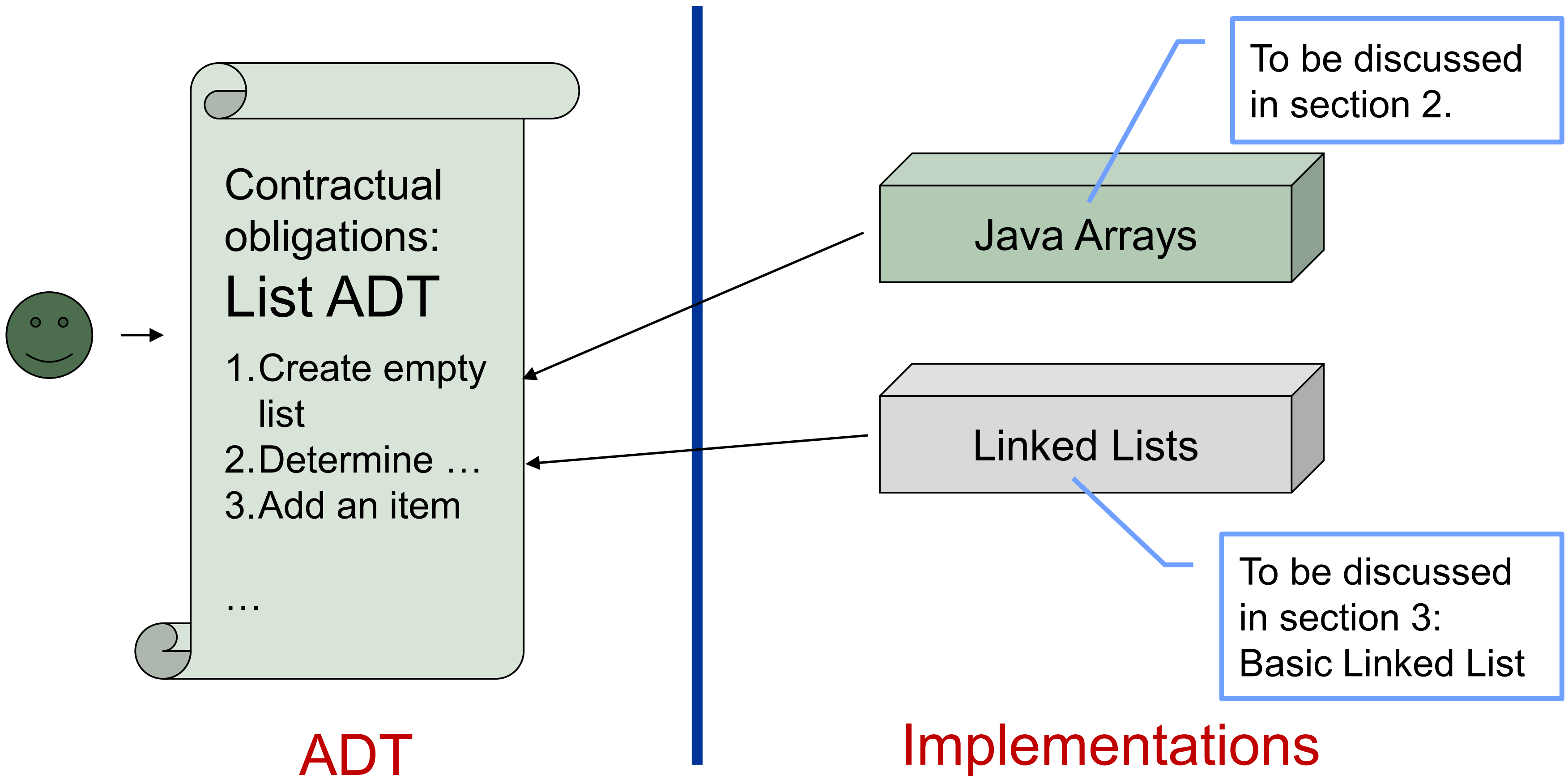
```
import java.util.*;

// list interface for a list of integers
// Note: 1st item at index 0 & last item at index N-1
//       (where N is number of items in the list)
public interface ListInterface {
  public boolean isEmpty();
  public int     size();
  public int     indexOf(int item);
  public boolean contains(int item);
  public int     getItemAtIndex(int index);
  public int     getFirst();
  public int     getLast();
  public void    addAtIndex(int index, int item);
  public void    addFront(int item);
  public void    addBack(int item);
  public int     removeAtIndex(int index);
  public int     removeFront();
  public int     removeBack();
  public void    print();
}
```

ADT of a List (2/3)

- The **ListInterface** defines the operations (methods) we would like to have in a List ADT
- The operations shown here are just a small sample. An actual List ADT usually contains more operations.
- Here we assume that the List ADT only contains integer elements
- Using indices to access the elements in the list, 1^{st} element is at index 0 and last element is at index N-1 (where N is the number of elements in the list)

ADT of a List (3/3)

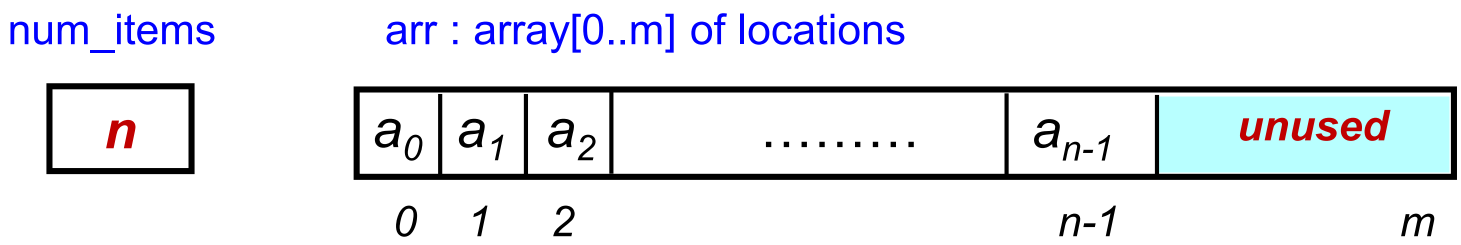
- We will examine 2 implementations of list ADT, both using the **ListInterface** shown in the previous slide



2 List Implementation via Array

List Implementation: Array

- This is a straight-forward approach
  - Use Java array of a sequence of *n* items

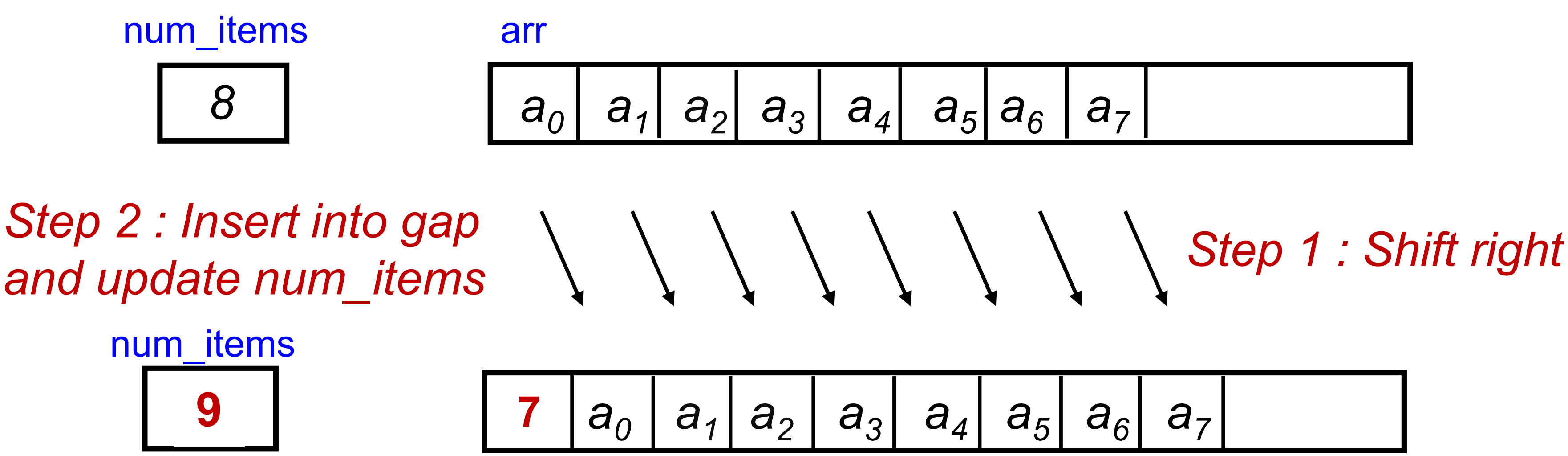


  - Since it is an array, items occupy a contiguous block of memory

Basic operations: Insert an item

- To insert an item at position/index i
    1. Shift last item to item at index i to the right by 1 to create a “gap”
    2. Insert the new item in the “gap” created

Example: inserting 7 at index 0



Basic operations: Insert an item

- What happens if the array is already filled when inserting an item?
  1. Enlarge it by creating a new array (usually doubling the size of original array) and copy original array over
  2. Insert new item as per normal

- This makes the array a dynamic array (can be re-sized) instead of a static array (fixed size)

Insert method with array resizing

ListUsingArray.java

```
import java.util.*;

class ListUsingArray implements ListInterface {
  public int capacity = 1000; // size of the array
  public int num_items;       // number of items in the array
  public int[] arr = new int[capacity];
  …

  // helper non-interface methods

  public void insert(int index, int item) {
    if (num_items+1 > capacity) // array is full, enlarge it
      enlargeArr();
    for (int i=num_items-1; i >= index; i--) // create gap
      arr[i+1] = arr[i];
    arr[index] = item; // insert item in gap
    num_items++;
  }
}
```

Insert method with array resizing

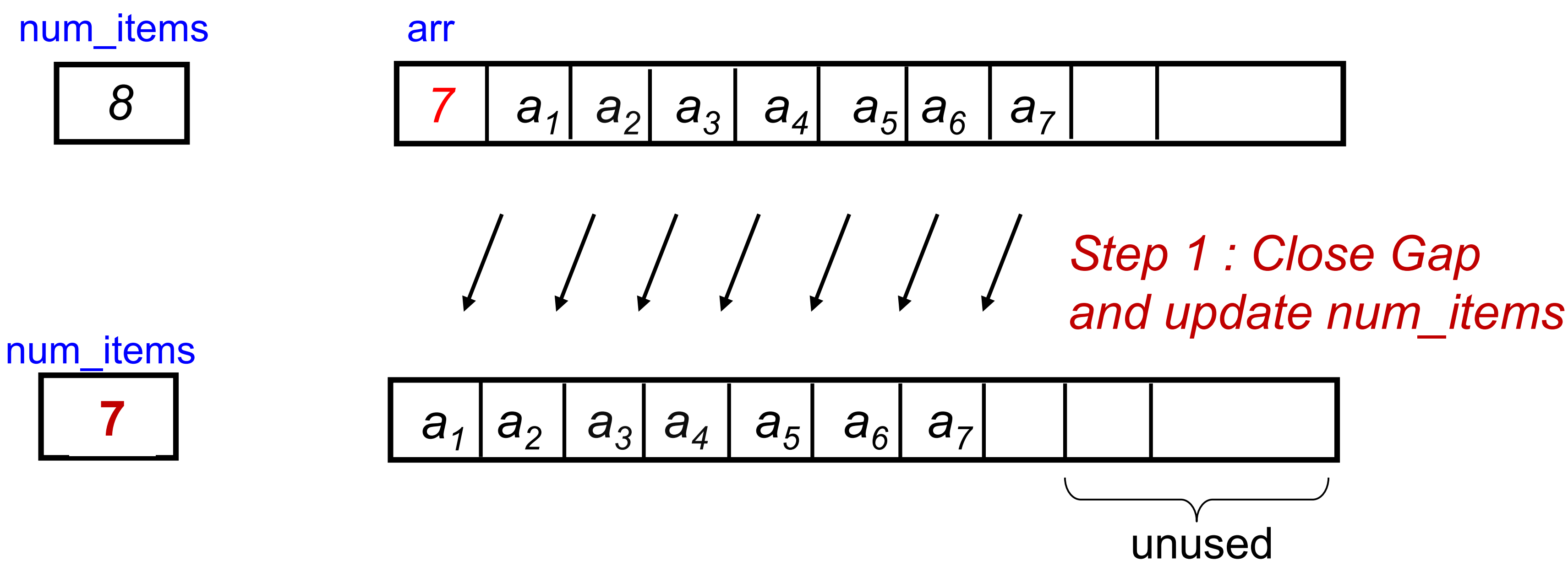
ListUsingArray.java

```
  public void enlargeArr() {
    int newSize = capacity * 2; // double the size
    int[] temp = new int[newSize];

    if (temp == null) { // not enough memory
      System.out.println("run out of memory!");
      System.exit(1);
    }
    // copy the original array to the new array
    for (int j=0; j < num_items; j++)
      temp[j] = arr[j];
    arr = temp; // point arr to the new array
    capacity = newSize;
  }
}
```

Basic operations: Remove an item

- To remove/delete an item at position/index i
  1. Shift current items from index i+1 and onwards to the left by 1 to delete the item and close the gap

Example: remove first item



Remove method

ListUsingArray.java

```
import java.util.*;

class ListUsingArray implements ListInterface {
  …

  // remove the item at index and return it
  public int remove(int index) {
    int item = arr[index];

    // shift item from index+1 onwards to the left to close the gap
    for (int i=index+1; i < num_items; i++)
      arr[i-1] = arr[i];
    num_items--;

    return item;
  }
}
```

List operations using array implementation

- Useful operations
  - empty() – return true if num_items is 0, false otherwise
  - size() – return num_items

- Operations to check if item exist in the list
  - indexOf(int item) – scan through the array and return the index of item if it is found otherwise return -1
  - contains(int item) – if indexOf(item) == -1 return false else return true

List operations using array implementation

- Operations to retrieve/access item at index i in the list
  - getItemAtIndex(int i) – if i within the bounds of the array return arr[i]
  - getFirst() – return getItemAtIndex(0)
  - getLast() – return getItemAtIndex(num_items-1)

- Operations to add item at index i in the list

  **(i = num_items means add as last item in the list)*
  - addAtIndex(int i, int item) – if 0 <= i <= num_items call insert(i,item)
  - addFront(int item) – call addAtIndex(0,item)
  - addBack(int item) – call addAtIndex(num_items,item)

List operations using array implementation

- Operations to remove an item at index i in the list and return it
  - removeItemAtIndex(int i) – if i is within the bounds of the array return remove(i) otherwise do nothing
  - removeFront() – return removeItemAtIndex(0)
  - removeBack() – return removeItemAtIndex(num_items-1)

- Refer to *ListUsingArray.java* for the full program

Testing Array Implementation of List

TestListUsingArray.java

```
import java.util.*;

public class TestListUsingArray {
  public static void main(String [] args) {
    ListUsingArray list = new ListUsingArray();
    list.addFront(1);
    list.addFront(2);
    list.addFront(3);
    list.addBack(4);
    list.addAtIndex(2,5);
    list.print();

    System.out.println("Testing removal");
    list.removeFront();
    list.removeBack();
    list.removeAtIndex(1);
    list.print();

    if (list.contains(1))
      list.addFront(6);
    list.print();
  }
}
```

```
List is: 3, 2, 5, 1, 4.
Testing removal
List is: 2, 1.
List is: 6, 2, 1.
```

Analysis of Array Impl^n of List

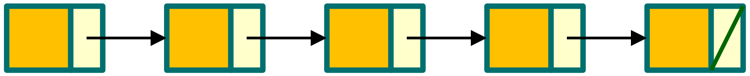
- Time complexity of the different list operations
  - Retrieval: *getItemAtIndex(int i), getFirst(), getLast()*
    - O(1) – indexing into an array is constant time due to random access memory of the computer
  - Insertion: *addItemAtIndex(int i, int item), addFront(), addBack()*
    - Best case = O(1) – if adding at the back and no need to enlarge array
    - Worst case = O(n) – if adding to the front due to shifting all item to the right

      or need to enlarge the array so have to perform copying of all n items to new array
    - Average case = O(n) – on average need to shift ½(n) items to the right
  - Deletion: *removeItemAtIndex(int i), removeFront(), removeBack()*
    - Best case = O(1) – if removing from the back
    - Worst case = O(n) – if removing from the front due to shifting all items to the left
    - Average case = O(n) – on average need to shift ½(n) items to the left

Analysis of Array Impl^n of List

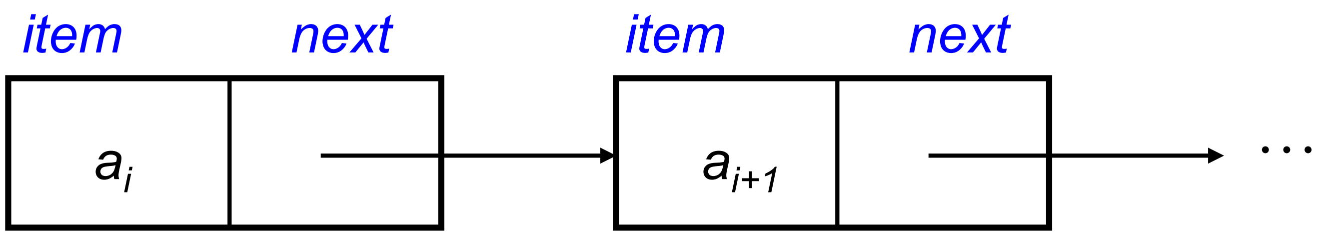
- What about the Space Complexity?
  - In the best case, we use exactly n space for n items
  - In the worst case, we use 2n space for n+1 item where n is multiple of 2
  - Either way, the space complexity is O(n)

3 List Implementation via Linked List



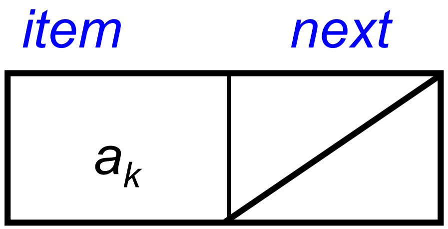
3.1 Linked List Approach

- **Idea**
  - Each item in the list is stored in a *node*, which also contains a next pointer that references/points to the node to its right (its neighbour)
  - Order the nodes by associating each with its neighbour(s)
  - Allow elements in the list to occupy *non-contiguous* memory



This is one node of the collection…

… and this one comes after it in the collection (most likely not occupying contiguous memory that is next to the previous node).



Next pointer of this node is "null", i.e. it has no next neighbour.

3.3 ListNode – contain integer element

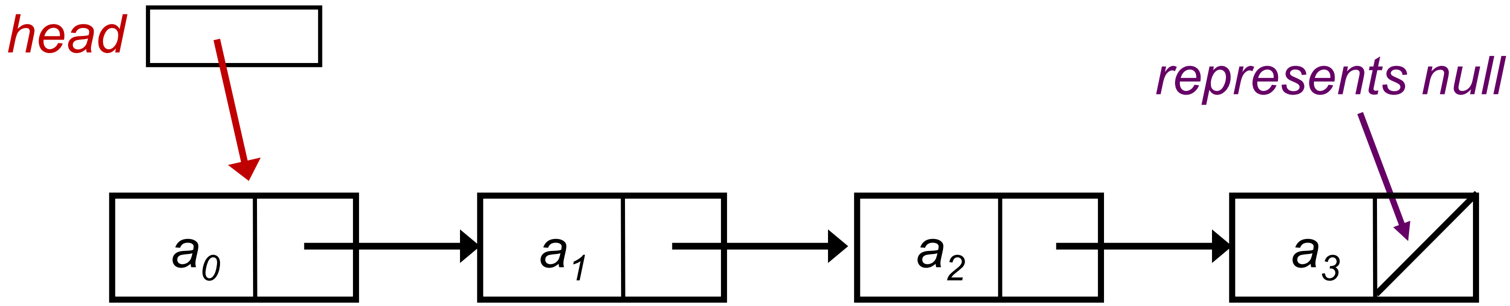
ListNode.java

```
class ListNode {
  /* attributes */
  public int item;
  public ListNode next;

  /* constructors */
  public ListNode(int val) { this(val, null); }

  public ListNode(int val, ListNode n) {
    item = val;
    next = n;
  }

  /* get the next ListNode */
  public ListNode getNext() { return next; }

  /* get the item of the ListNode */
  public int getItem() { return item; }

  /* set the item of the ListNode */
  public int setItem(int val) { item = val; }

  /* set the next reference */
  public void setNext(ListNode n) { next = n };
}
```

Mark this slide – You may need to refer to it later when we study the different variants of linked list.

Forming a Linked List

- For a sequence of 4 items < a_0, a_1, a_2, a_3 >



We need a *head* to indicate where the first node is.
From the *head* we can get to the rest.

- Create a BasicLinkedList class that contains a head attribute and implements the list operations

3.5 Basic Linked List

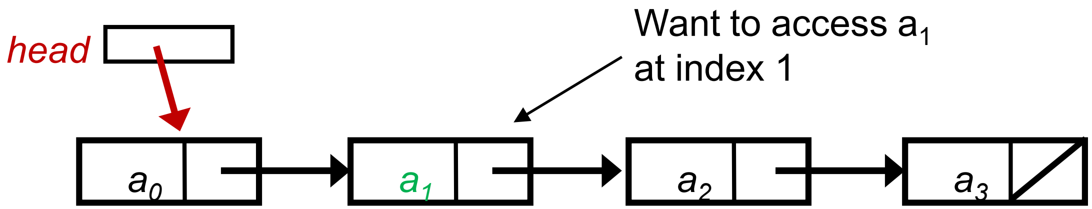
- Using ListNode to define BasicLinkedList

BasicLinkedList.java

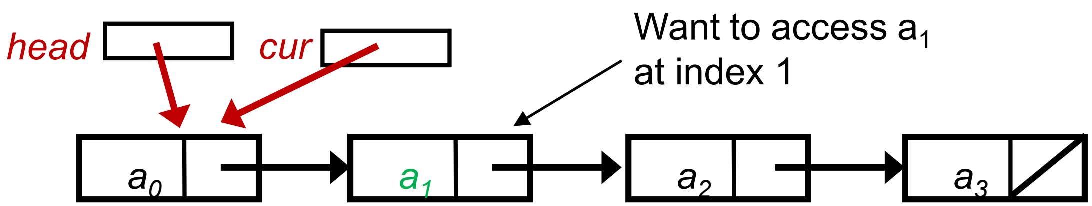
```
import java.util.*;

class BasicLinkedList implements ListInterface {
  public ListNode head; // points to the 1st node in the list,
                        // initialized to null
  public int num_nodes;

  …
}
```

Basic Operations: Accessing items in the LL

- To access an item at index i in the LL we need to have a reference pointing to its node (cannot directly access the node like an array)

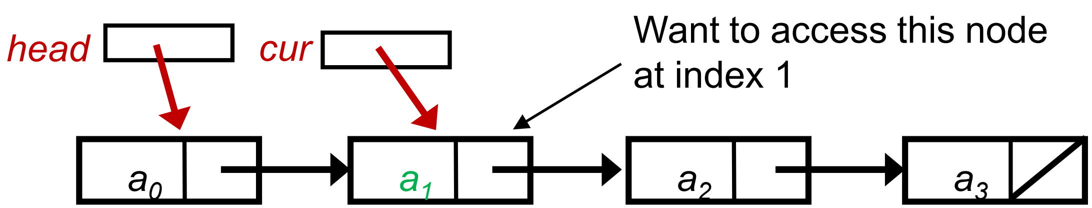


- Have a reference cur that start from head




```
ListNode cur = head;
```

- And “moves” towards node containing a_1



*To access nodes further away, simply use a loop

```
cur = cur.getNext();
```

Accessing an item: getItemAtIndex(int i)

BasicLinkedList.java

```
import java.util.*;

class BasicLinkedList implements ListInterface {
  …

  public int getItemAtIndex(int index) {
    int counter = 0;
    int item = 0;

    if (index < 0 || index > size()-1) {
      System.out.println("invalid index");
      System.exit(1);
    }
    for (ListNode cur = head; cur != null; cur = cur.getNext(),
          counter++) {
      if (counter == index) {
        item = cur.getItem();
        break;
      }
    }
    return item;
  }
}
```

We have an equivalent method **ListNode getNodeAtIndex(int index)** which returns the node instead of item in the node. Refer to BasicLinkedList.java

Start from head

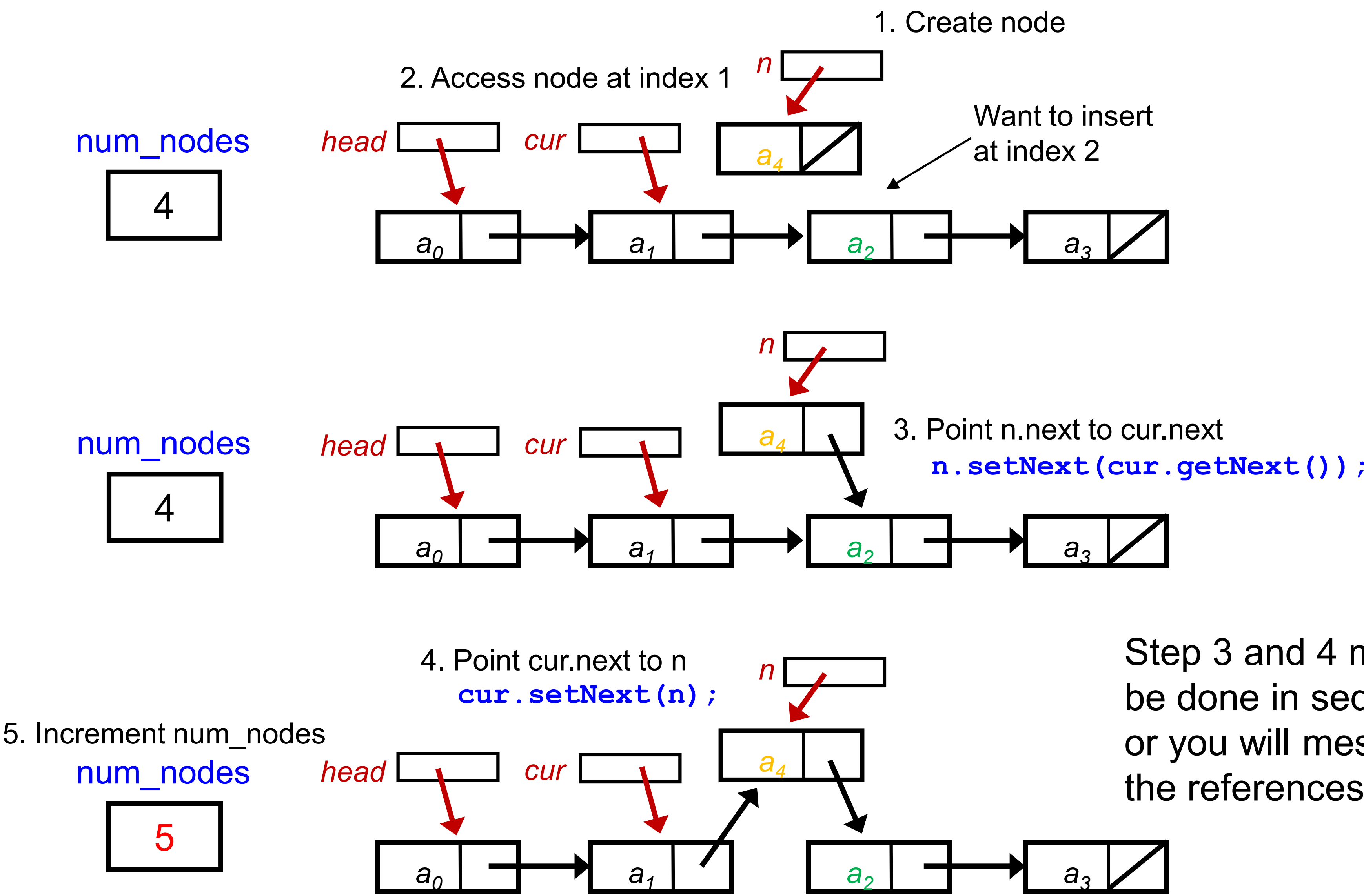
While not moved past last node in the list

Move through the list

Basic Operations: Insert an item

- Insert an item at index i in the linked list
  1. Create a new node n containing the item
  2. Access the node cur at index i-1
  3. Point next reference of n to neighbor of cur
  4. Point next reference of cur to n
  5. Increment number of nodes

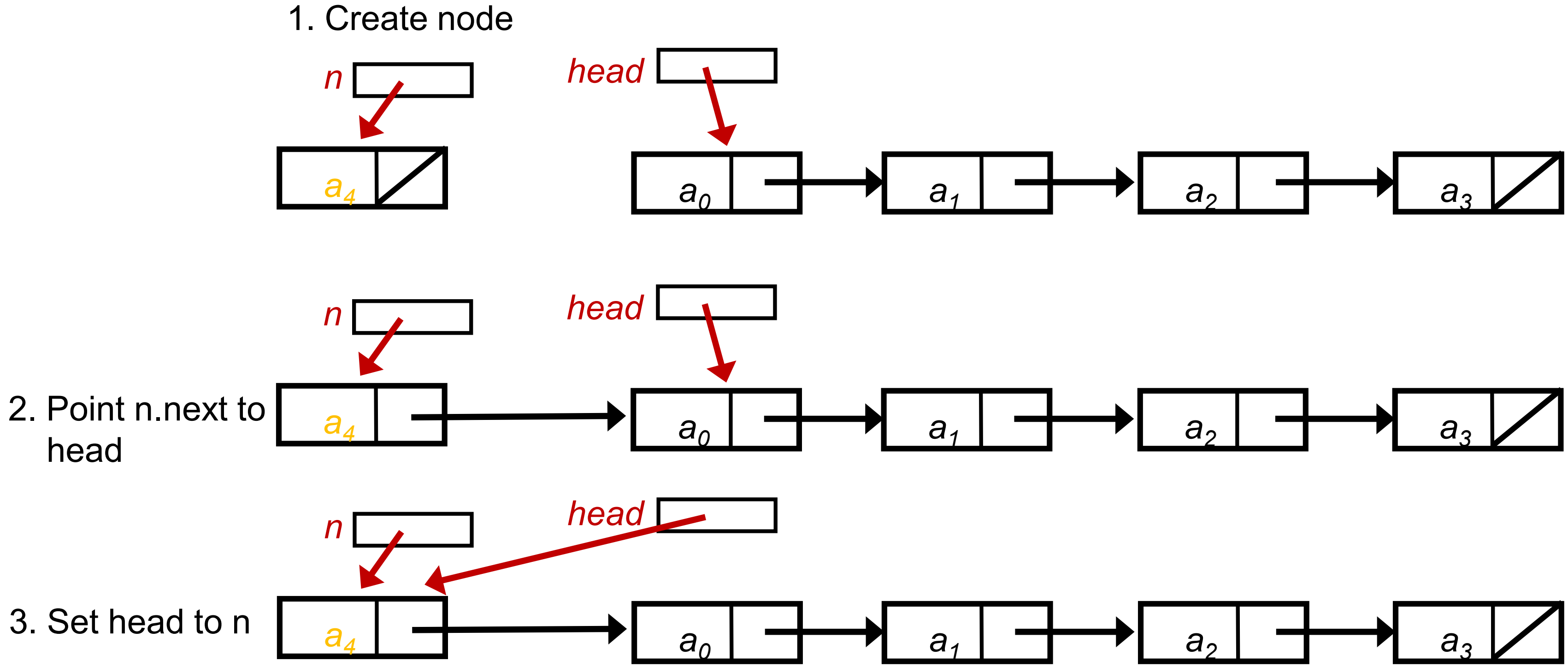
Basic Operations: Insert an item



Step 3 and 4 must be done in sequence or you will mess up the references!

Insert an item – Special cases?

1. When adding to the front of the linked list (i.e index 0) → In this case cur == null (no node before index 0)
   1. Create new node n
   2. Point next of n to head
   3. Set head to n && increment num_nodes (*not shown due to space constraint*)



Insert an item – Special cases?

2. When the list is empty
   - New item is added to the front of the list therefore this is same case as 1.

3. When adding to back of the list (i.e index num_nodes)
   - No need special handling as cur will be pointing to the last node (which is at index num_nodes-1)

Insert an item: insert(ListNode cur, ListNode n)

- Code for inserting a new node n at index i given reference cur to node at index i-1

BasicLinkedList.java

```
import java.util.*;

class BasicLinkedList implements ListInterface {
  …

  public void insert(ListNode cur, ListNode n) {
    if (cur == null) { // insert in front of list
      n.setNext(head);
      head = n; // update head
    }
    else { // insert anywhere else
      n.setNext(cur.getNext());
      cur.setNext(n);
    }
    num_nodes++; // important !
  }
}
```

Time complexity of insert is O(1) (no loops, only a fixed number of statements)

Implement addAtIndex using insert

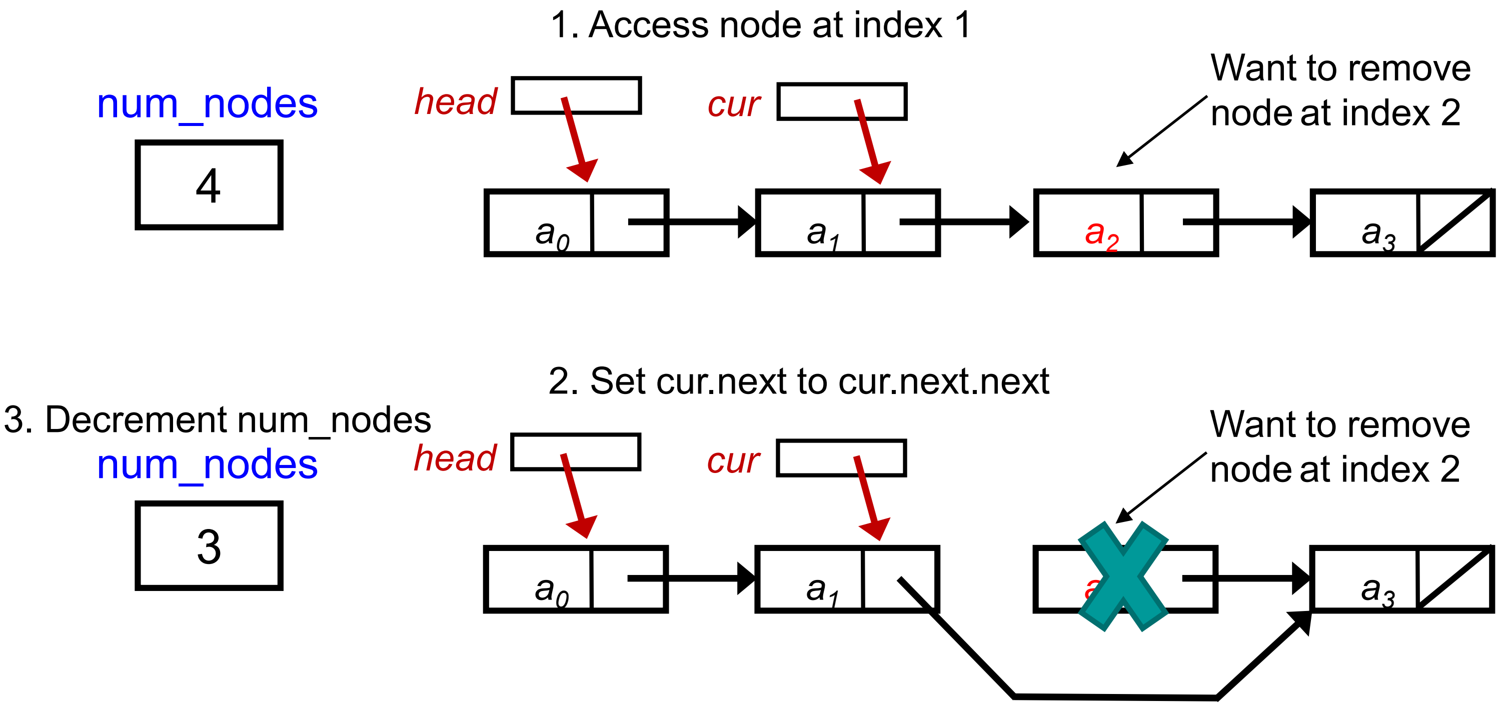
- List operation addAtIndex can now be easily implemented using insert

BasicLinkedList.java

```
public void addAtIndex(int index, int item) {
  ListNode cur;
  ListNode newNode = new ListNode(item);

  if (index >= 0 && index <= size()) {
    if (index == 0) // insert in front
      insert(null,newNode);
    else {
      cur = getNodeAtIndex(index-1); // access node at index-1
      insert(cur,newNode);
    }
  }
  else { // index out of bounds
   System.out.println("invalid index");
   System.exit(1);
  }
}
```

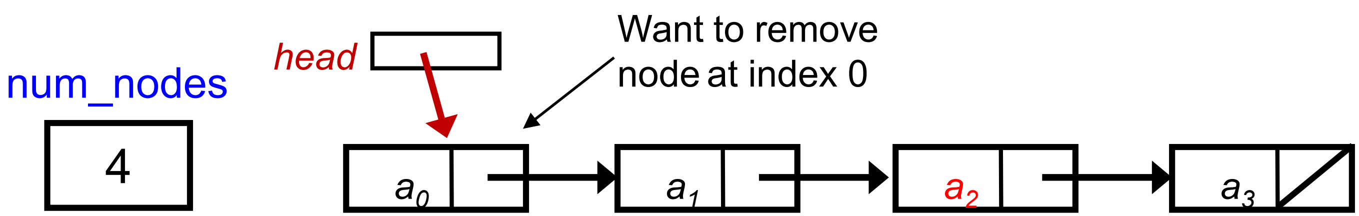
Basic Operation: Remove an item

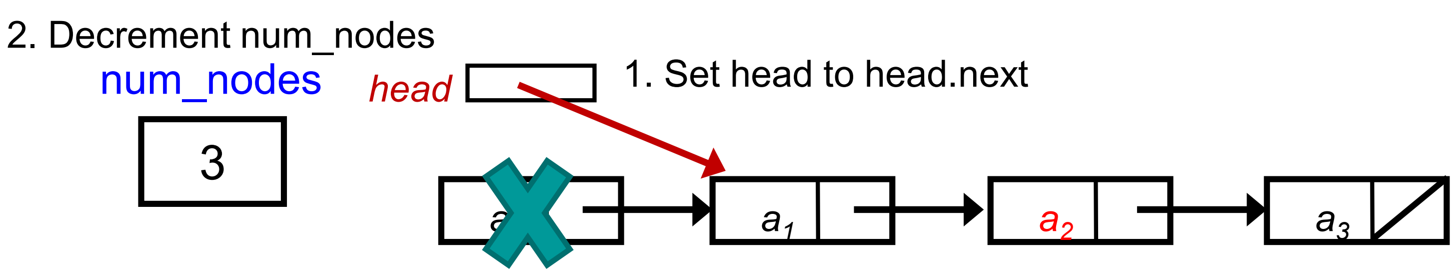
- Remove an item at index i in the linked list
  1. Access the node cur at index i-1
  2. Point next reference of cur to neighbor of neighbor of cur
  3. Decrement number of nodes



Remove an item – Special case

- When removing from the front of the linked list (i.e index 0) → In this case cur == null (no node before index 0)
  1. Point head to head.next
  2. decrement num_nodes





Remove an item: remove(ListNode cur)

- Code for removing a node at index i given reference cur to node at index i-1

BasicLinkedList.java

```
import java.util.*;

class BasicLinkedList implements ListInterface {

  public int remove(ListNode cur) {
    int value;

    if (cur == null) { // remove 1st node
      value = head.getItem();
      head = head.getNext(); // update head
    }
    else { // remove any other node
      value = cur.getNext().getItem();
      cur.setNext(cur.getNext().getNext());
    }
    num_nodes--; // important !

    return value;
  }
```

Time complexity of remove is O(1) (no loops, only a fixed number of statements)

Implement removeAtIndex using remove

- List operation removeAtIndex can now be easily implemented using remove

BasicLinkedList.java

```
public int removeAtIndex(int index) {
  ListNode cur;
  int item = 0;

  // index within bounds and list is not empty
  if (index >= 0 && index < size() && head != null) {
    if (index == 0) // remove 1st item
      item = remove(null);
    else {
      cur = getNodeAtIndex(index-1); // access node at index-1
      item = remove(cur);
    }
  }
  else { // index out of bounds
    System.out.println("invalid index or list is empty");
    System.exit(1);
  }
  return item;
}
```

3.5 Test Basic Linked List

TestBasicLinkedList.java

```
import java.util.*;

public class TestBasicLinkedList {
  public static void main(String [] args) {
    BasicLinkedList list = new BasicLinkedList();
    list.addFront(1);
    list.addFront(2);
    list.addFront(3);
    list.addBack(4);
    list.addAtIndex(2,5);
    list.print();

    System.out.println("Testing removal");
    list.removeFront();
    list.removeBack();
    list.removeAtIndex(1);
    list.print();

    if (list.contains(1))
      list.addFront(6);
    list.print();
  }
}
```

```
List is: 3, 2, 5, 1, 4.
Testing removal
List is: 2, 1.
List is: 6, 2, 1.
```

Analysis of Linked List Implementation of List

- Time complexity of the different list operations
  - Retrieval: *getItemAtIndex(int i), getFirst(), getLast()*
    - Best case = O(1) – accessing the first node, return the head
    - Worst case = O(n) – accessing the last node, since you need to move all the way to the back from the head (n moves)
    - Average case = O(n) – need to move about half way through the list to access any node on average so ½(n) iterations of the for loop
  - Insertion: *addItemAtIndex(int i, int item), addFront(), addBack()*
    - Best case = O(1) – if adding at the front (don't have to worry about enlarging the list unlike array)
    - Worst case = O(n) – if adding to the back due to having to move all the way to the back from the head (n moves)
    - Average case = O(n) – on average need to make ½(n) moves

Analysis of Linked List Implementation of List

- Deletion: *removeItemAtIndex(int i), removeFront(), removeBack()*
  - Best case = O(1) – if removing from the front
  - Worst case = O(n) – if removing from the back, again due to moving all the way to the back from the head
  - Average case = O(n) – on average need to make ½(n) moves

- What about the Space Complexity?
  - We use as much space as there are nodes in the list so exactly O(n) (plus some constant overhead to store each node which requires more space then a simple integer that is stored in an array)

When to use Array vs LL implementation of List

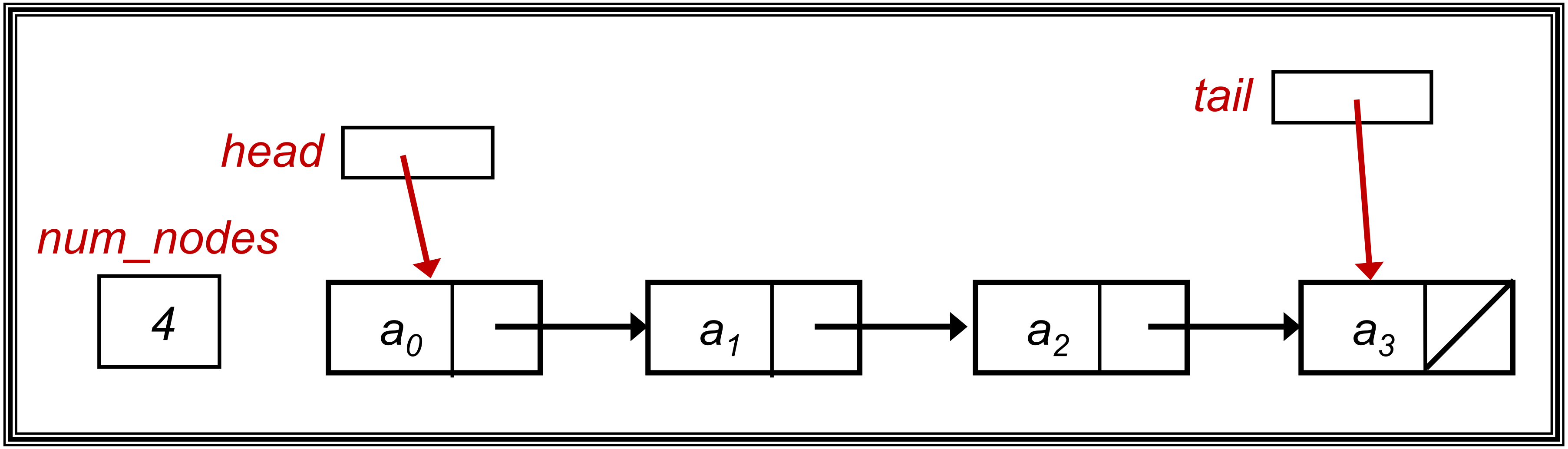
- Depends on the problem
  - Only need to add/remove to front of the list → Use LL
  - Only need to add/remove to back of the list → Use Array, especially if we know maximum num. of items in advance, just create a large enough array from start
  - If we need to add/remove anywhere in the list
    - Equal chance of adding to any index → both LL or Array is the same
    - If we need to keep adding/removing to a particular index i in the list → Use LL, maintain a reference to the node at index i-1, all insertions/deletions thereafter is only O(1) time
  - If we have few insertions/deletions but a lot of accesses → Use Array since accessing any item is O(1) time

4 More Linked Lists

Exploring variants of linked list

Tailed Linked List

- We further improve on our Basic Linked List
  - To address the issue that adding to the end is slow (need to move all the way to the back)
  - Add an extra attribute called tail that points to the last node in the list
  - Extra attributes means extra maintenance too – no free lunch!
- Difficulty: Learn to take care of ALL cases of updating...



4.1 Tailed Linked List

- We create a new class TailedLinkedList:

TailedLinkedList.java

```
import java.util.*;

public class TailedLinkedList implements ListInterface {
  …
  ListNode tail; // additional attribute

  …
  public ListNode getTail(); // method to return tail
}
```

Accessing an item: getItemAtIndex(int i)

TailedLinkedList.java

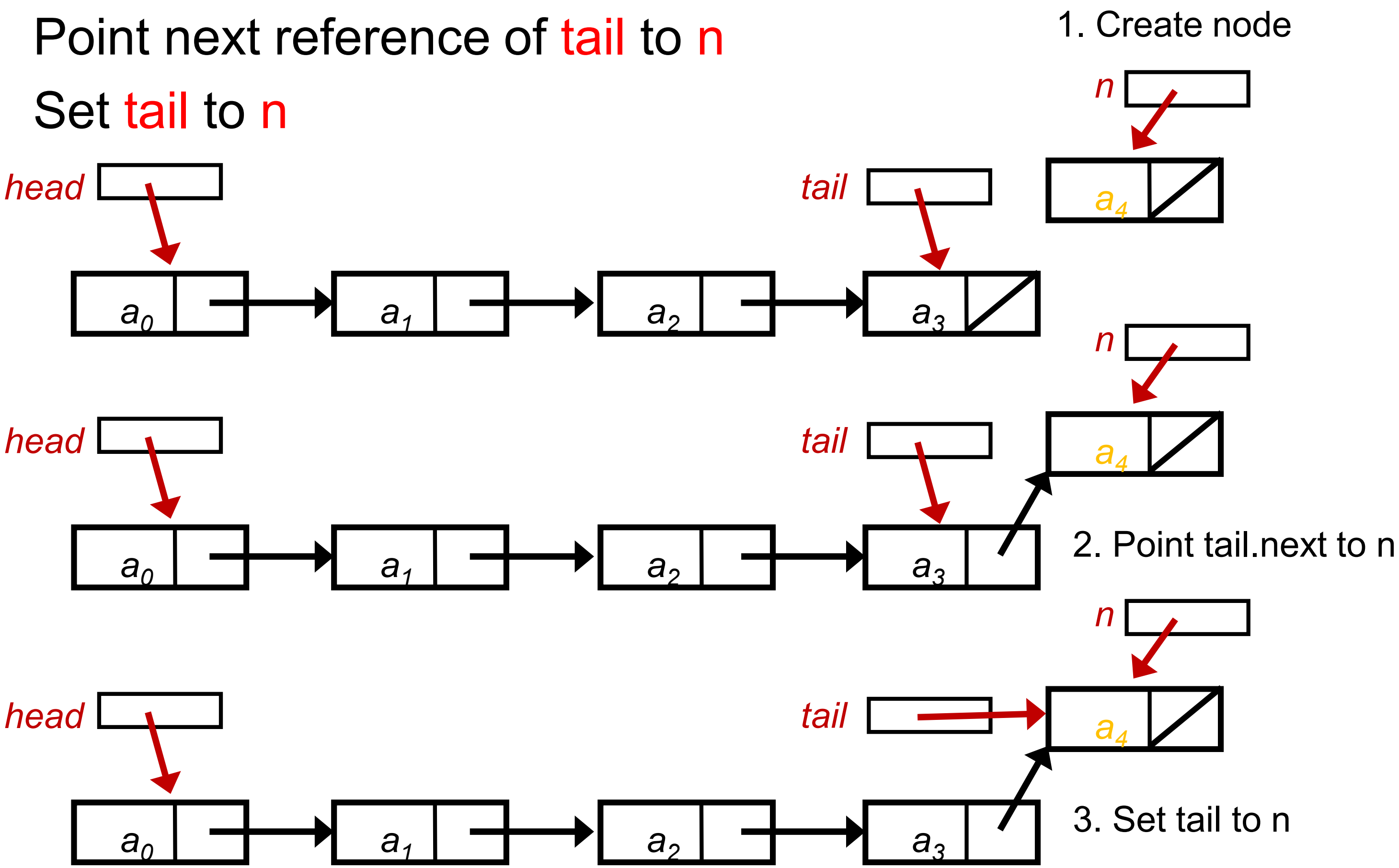
```
class TailedLinkedList implements ListInterface {

  public int getItemAtIndex(int index) {
    int counter = 0;
    int item = 0;

    if (index < 0 || index > size()-1) {
      System.out.println("invalid index");
      System.exit(1);
    }
    if (index == size()-1)
      item = tail.getItem();
    else {
      for (ListNode cur = head; cur != null; cur = cur.getNext(),
           counter++) {
        if (counter == index) {
          item = cur.getItem();
          break;
        }
      }
    }
    return item;
  }
}
```

Simply return item in tail if accessing last item, thus **getLast()** is O(1) time

Otherwise same as for BasicLinkedList

Inserting in a tailed linked list

- Inserting at index == size(), Immediately insert using the tail without having to start from head and moving to the back.
  1. Create the node n to be inserted
  2. Point next reference of tail to n
  3. Set tail to n



No Free lunch – Extra book keeping in insert()

- In the insert method need to update tail
  1. When inserting to tail (illustrated in previous slide)
  2. When inserting into empty list (tail == null)

TailedLinkedList.java

```
public void insert(ListNode cur, ListNode n) {
  if (cur == null) { // insert in front of list
    n.setNext(head);
    head = n; // update head
    if (tail == null) // update tail if list originally empty
      tail = head;
  }
  else { // insert anywhere else
    n.setNext(cur.getNext());
    cur.setNext(n);
    if (cur == tail) // update tail if insert new last item
      tail = tail.getNext();
  }
  num_nodes++;
}
```

Case 2

Case 1

Simple modification of addAtIndex

- List operation addAtIndex can now be easily modified to support fast insertion to end of the Tailed Linked List

TailedLinkedList.java

```
public void addAtIndex(int index, int item) {
  ListNode cur;
  ListNode newNode = new ListNode(item);

  if (index >= 0 && index <= size()) {
    ListNode t = new ListNode(item);

    if (index == 0) // insert in front
      insert(null,newNode);
    else if (index == size())
      insert(tail,newNode);
    else {…} // insert anywhere else
  }
  else {…} // index out of bounds
}
```

Use tail as the cur reference when inserting to the back. Don't have to move all the way to the back to insert

Removing from a tailed linked list

- Need to update tail reference whenever last item and only item is removed (NO FREE LUNCH!)

```
class TailedLinkedList implements ListInterface {

  public int remove(ListNode cur) {
    int value;

    if (cur == null) { // remove 1st node
      value = head.getItem();
      head = head.getNext(); // update head
      if (num_nodes == 1)
        tail = null;
    }
    else { // remove any other node
      value = cur.getNext().getItem();
      cur.setNext(cur.getNext().getNext());
      if (cur.getNext() == null)
        tail = cur;
    }
    num_nodes--;
    return value;
  }
```

Update tail to null if only item in list is removed

Update tail to cur if last item in list is removed

4.2 Test Tailed Linked List (8/10)

TestTailedLinkedList.java

```
import java.util.*;

public class TestTailedLinkedList {
  public static void main(String [] args) {
    TailedLinkedList list = new TailedLinkedList();
    list.addFront(1);
    list.addFront(2);
    list.addFront(3);
    list.addBack(4);
    list.addAtIndex(2,5);
    list.print();

    System.out.println("Testing removal");
    list.removeFront();
    list.removeBack();
    list.removeAtIndex(1);
    list.print();

    if (list.contains(1))
      list.addFront(6);
    list.print();
  }
}
```

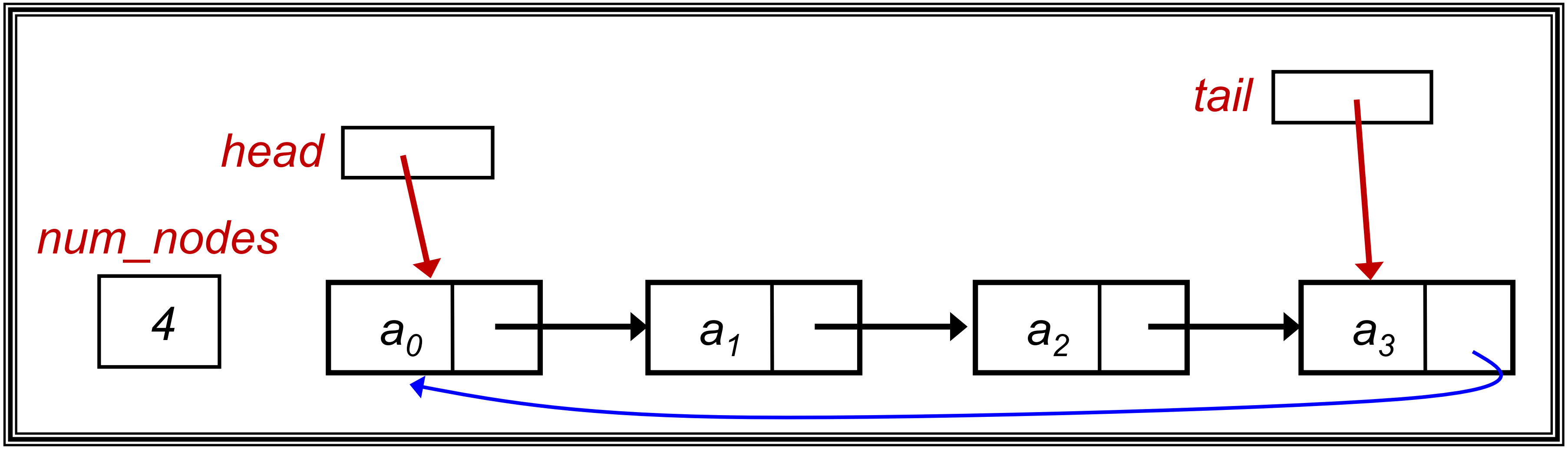
```
List is: 3, 2, 5, 1, 4.
Testing removal
List is: 2, 1.
List is: 6, 2, 1.
```

5 Other Variants

Other variants of linked lists

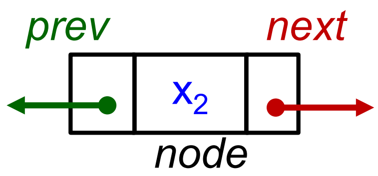
5.1 Circular Linked List

- There are many other possible enhancements of linked list
- Example: Circular Linked List
  - To allow cycling through the list repeatedly, e.g. in a **round robin system** to assign shared resource
  - Add a link from tail node of the TailedLinkedList to point back to head node
  - Difference in linking need different maintenance – no free lunch!
- Difficulty: Learn to take care of ALL cases of updating, such as inserting/deleting the first/last node in a Circular Linked List
- Explore this on your own; write a class CircularLinkedList



Doubly Linked List

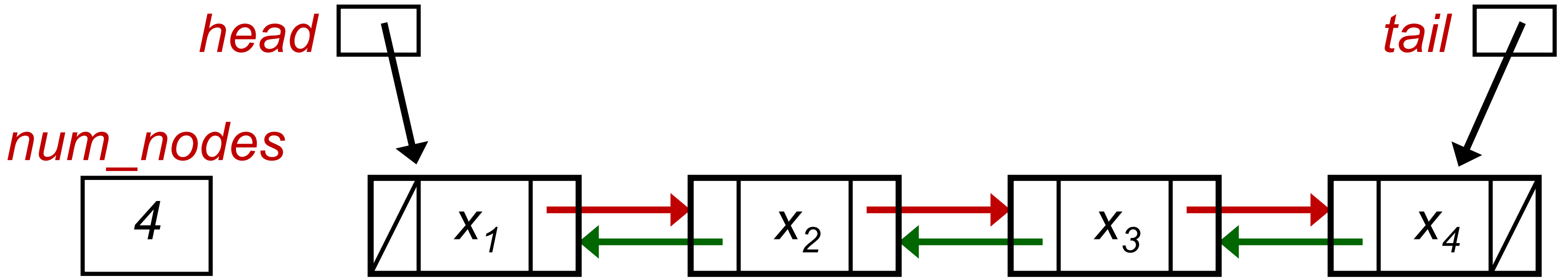
- In the preceding discussion, we have a "**next**" pointer to move forward
- Often, we need to move backward as well
- Use a "**prev**" pointer to allow backward traversal
- Once again, no free lunch – need to maintain "**prev**" in all updating methods
- Instead of ListNode class, need to create a DListNode class that includes the additional "**prev**" pointer



- Study DListNode.java for how to implement a DListNode

Doubly Linked List

- An example of a doubly linked list



Usefulness of Doubly Linked List

- Reduce moving of references
  - When adding/removing/accessing a particular index, you can start from the end (front or back) that is closer to the index
  - Does not change time complexity of the List operations but improves the constant which is still important in practice! (especially in beating time limit constraints of lab assignments …)

- Write a class DoublyLinkedList to implement the various linked list operations for a doubly linked list.

6 Java API: ArrayList class and LinkedList class

Using the LinkedList class

6 Java Classes: ArrayList and LinkedList

- These 2 are classes provided by Java library which implements the List ADT/interface
- ArrayList is the array implementation while LinkedList is the linked list implementation
- They have many more methods than what we have discussed so far of our versions of linked lists. On the other hand, we created some methods not available in the Java library class too.
- Please do not confuse this library class from our class illustrated here. In a way, we open up the Java library to show you the inside working.
- For purposes of labs or exam, you can use ArrayList or LinkedList if it is not stated that you have to use your own List implementation

6 Java Classes: ArrayList and LinkedList

- Both ArrayList and LinkedList have the same set of methods as they both implement the List interface
- Thus we can use them interchangeably
  - However note that the time complexity of some methods will differ due to the difference in the underlying data structure used (array vs linked list)
- Also both ArrayList and LinkedList classes are generic containers meaning they can contain objects of any class (but not primitive types)

Eg Using ArrayList

- Declaring an ArrayList variable/reference
  - To declare an ArrayList reference, specify the type of the object it contains. For example,

    ```
    ArrayList<Integer> list;
    ```

    list is a reference to an Arraylist containing Integer objects
- Creating an ArrayList object
  - Same as declaring an ArrayList reference need to specify the type of object it contains. For example,

    ```
    list = new ArrayList<Integer>();
    ```

    list now points to an ArrayList object that contains Integers
- Make sure the reference type and object type is the same/compatible
- LinkedList is used in the same manner

Why “reinvent the wheel”?

- In a data structures course, students are often asked to implement well-known data structures.
- A question we sometimes hear from students: “Since there is the API, why do we need to learn to write our own code to implement a data structure like linked list?”
- Writing the code allows you to gain an in-depth understanding of the data structures and their operations
- The understanding will allow you to appreciate their time complexity analysis and use the API effectively

7 Summary

- We learn to create our own data structure to implement the List ADT
  - Need to be careful with boundary cases
  - Manipulation of references (The sequence of statements is important! With the wrong sequence, the result will be wrong.)
  - Re-use of codes (Think in terms of basic operations and how more complex operations can re-use the basic ones → removeLast & removeFirst, addLast & addFirst)
  - Drawings are very helpful in understanding the cases, which then can help in knowing what can be used/manipulated

End of file