

Arrays and Linked Lists

Dr. Anirban Ghosh

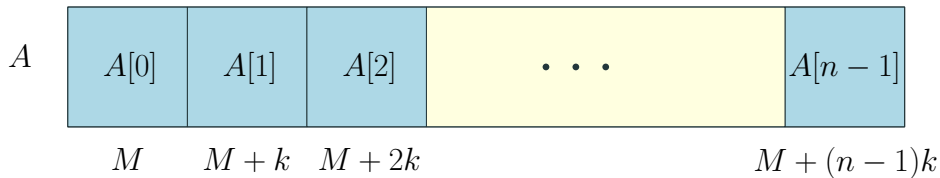
School of Computing
University of North Florida



What is an array?

Definition

An array is a **contiguous** sequence of memory locations each of which can hold items of a fixed data type



Your first data structure!

👉 **Things work a bit differently when we work with non-primitives! In this case, the cells store references as opposed to storing the objects.**

How to create an array of objects?

- **Approach 1.** declaring and defining the array at the same time.

```
Point[] anArrayOfPoints = {new Point(1.1,2.1), new Point(3.1, 4.1), new Point(5.1,6.1)};
```

- **Approach 2.** allocate the array and then create the objects.

```
Point[] anArrayOfPoints = new Point[n]; // point objects are not created yet!  
// double x = anArrayOfPoints[5].getX(); <-- cannot use this; objects are not created yet; will throw a NullPointerException  
for(int i = 0; i < anArrayOfPoints.length; i++)  
    anArrayOfPoints[i] = new Point(Math.random() * 10, Math.random() * 10);
```

Deep copy vs shallow copy

Shallow copy: source and destination point to the same entities

```
int[] sourceArrayA = {10,20,30,40,50};  
int[] destinationArrayA = sourceArrayA; // shallow-copy; just copying references; both point to the same array in the memory
```

Deep copy: source and destination point to different entities

```
int[] sourceArrayB = {50,60,70,80,90,100};  
int[] destinationArrayB = new int[sourceArrayB.length];  
  
for(int index = 0; index < sourceArrayB.length; index++) //deep-copy; copying stuff to the destination array  
    destinationArrayB[index] = sourceArrayB[index];  
  
// Another way to deep copy in Java  
int[] sourceArrayC = {11,21,31,41,51};  
int[] destinationArrayC = sourceArrayC.clone(); // deep-copy; beware! shallow-copy for non-primitives
```

Let us look at a demo
Class name. **CopyingDemo**

A few things about arrays

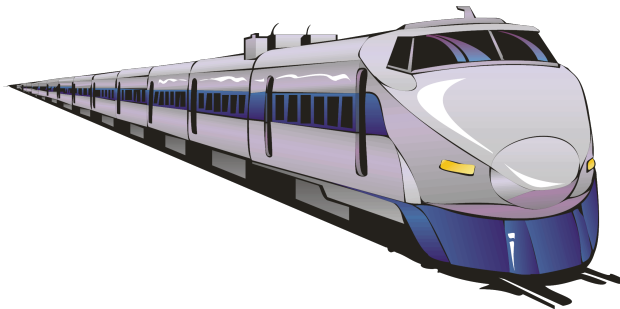
Good things

- Fast accesses for reading and writing since every location is indexed; every access takes $O(1)$ time (constant time)
- Once allocated no more memory allocation worries!
- Super-simple coding (`arr[i] = 10; a[i] = b[j] + c[k]; x = t[i]; etc.`)

Bad things

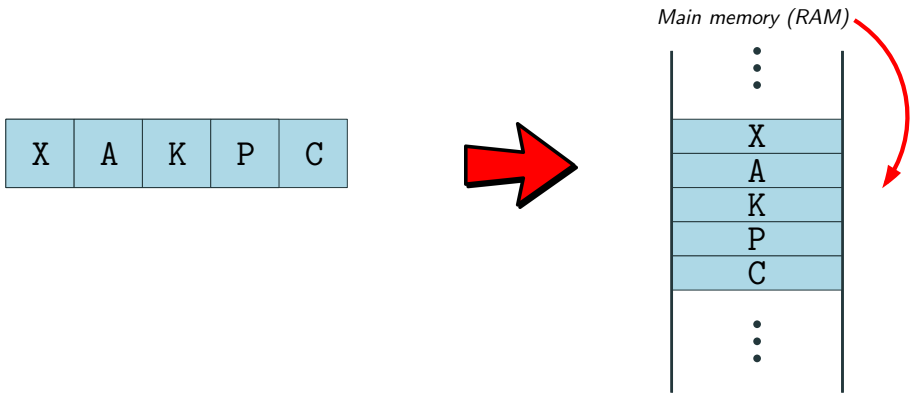
- Need to know size in advance otherwise it may run out of space for holding the incoming items; if more space is allocated in advance, space may remain unused!
- Cannot grow (arrays are static)
- Insertion/deletion can be expensive since right/left shifts are required which take $O(n)$ time in the worst case

The solution



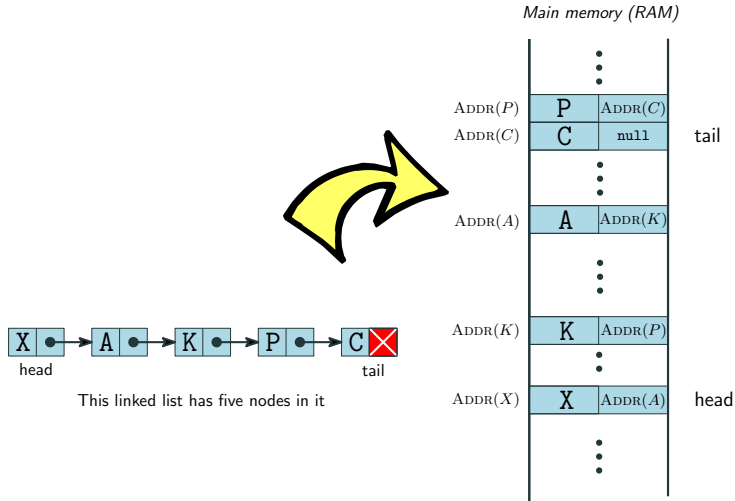
Linked Lists (Singly/Doubly/Circular)
Our first *dynamic* data structure

Layouts of arrays



Layout of arrays in the main memory (RAM)

Linked lists: what are they?



Layout of **singly** linked lists in the main memory (RAM)

Coding the example (a very naive way of course!)

```
public class ToySinglyLinkedList {  
  
    public static class Node { // a nested node  
        Character element;  
        Node next;  
  
        public Node(Character element, Node next) {  
            this.element = element;  
            this.next = next;  
        }  
  
        public void setNext(Node next) {  
            this.next = next;  
        }  
  
        public Character getElement() {  
            return element;  
        }  
    }  
}  
  
// contd. on the next slide
```

Coding the example (a very naive way of course!)

```
public class ToySinglyLinkedList {  
    // contd. from the previous slide  
    public static void main(String[] args) {  
        Node nodeX = new Node('X',null);  
        Node nodeA = new Node('A',null);    nodeX.setNext(nodeA);  
        Node nodeK = new Node('K',null);    nodeA.setNext(nodeK);  
        Node nodeP = new Node('P',null);    nodeK.setNext(nodeP);  
        Node nodeC = new Node('C',null);    nodeP.setNext(nodeC);  
  
        Node current = nodeX;  
        while( current != null ) {  
            System.out.print(current.getElement());  
  
            if(current.next != null)  
                System.out.print(" -> ");  
  
            current = current.next;  
        }  
    }  
}
```

Output

X -> A -> K -> P -> C

Good things about linked lists

- No need to worry about length in advance
- Can be grown/shrunk by manipulating the links and adding/deleting nodes (linked lists are **dynamic**)

👉 You **cannot** jump to a node directly like arrays! We need to traverse the list starting at the head using a for loop or a while loop and reach that node

👉 Coding can get a bit challenging at times.
NullPointerException is a common thing

It is time to go for generics!

**public class SinglyLinkedList<E> implements
Iterable<E>**

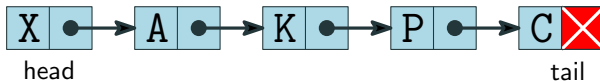
- ① **int** size() {...}
- ② **boolean** isEmpty() {...}
- ③ **E** first() {...}
- ④ **E** last() {...}
- ⑤ **void** addFirst(**E** e) {...}
- ⑥ **void** addLast(**E** e) {...}
- ⑦ **E** removeFirst() {...}
- ⑧ **boolean** addAfter(**E** predecessor, **E** incomingItem) {...}
- ⑨ **String** toString() {...}
- ⑩ **Iterator<E>** iterator() {...}

Adding a new node after a node: addAfter('K', 'T')

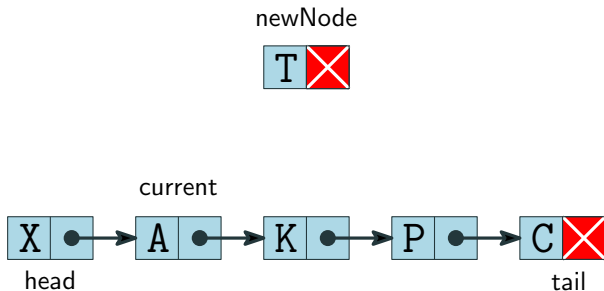
newNode



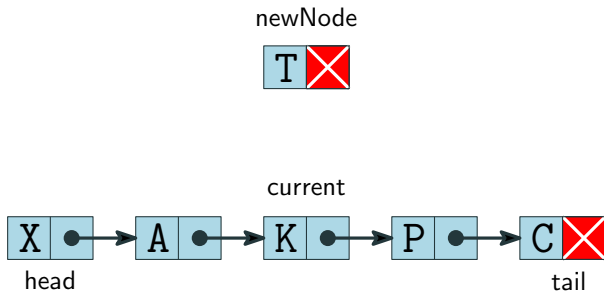
current



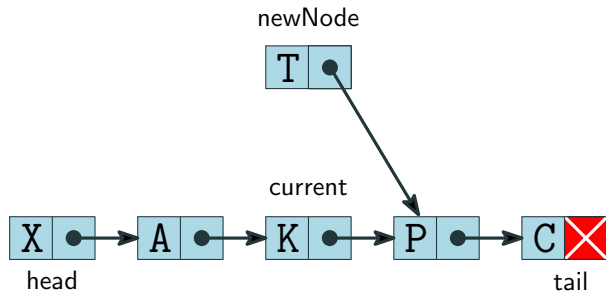
Adding a new node after a node: addAfter('K', 'T')



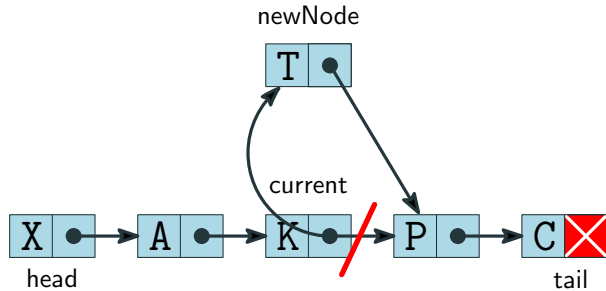
Adding a new node after a node: addAfter('K', 'T')



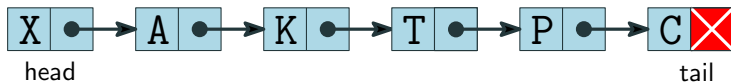
Adding a new node after a node: addAfter('K', 'T')



Adding a new node after a node: addAfter('K', 'T')



Adding a new node after a node: `addAfter('K', 'T')`



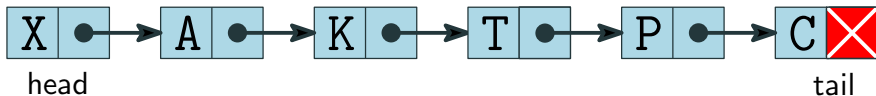
Complexity stuff

👉 Linked list traversals take $O(n)$ time

SinglyLinkedList<E>

- ① `int size() {...}`, **Time complexity:** $O(1)$
- ② `boolean isEmpty() {...}`, **Time complexity:** $O(1)$
- ③ `E first() {...}`, **Time complexity:** $O(1)$
- ④ `E last() {...}`, **Time complexity:** $O(1)$
- ⑤ `void addFirst(E e) {...}`, **Time complexity:** $O(1)$
- ⑥ `void addLast(E e) {...}`, **Time complexity:** $O(1)$
- ⑦ `E removeFirst() {...}`, **Time complexity:** $O(1)$
- ⑧ `boolean addAfter(E predecessor, E incomingItem) {...}`,
Time complexity: $O(n)$ where n is the number of nodes in the list currently
- ⑨ `String toString() {...}`, **Time complexity:** $O(n)$ where n is the number of nodes in the list currently

Limitations of singly linked lists



- Given just a reference to a node, we **cannot** efficiently delete it or add a node before it
- **Cannot** traverse from right to left just by following the links, if needed

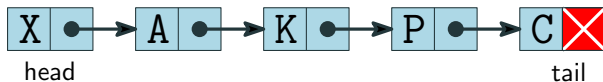
Solution

Store two links at every node: **prev** and **next**

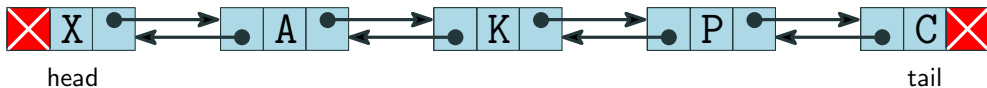
Downside?

Use of extra space at every node and possibly more complicated code

Doubly linked lists



A **singly** linked list

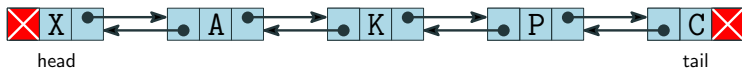


A **doubly** linked list

**public class DoublyLinkedList<E> implements
Iterable<E>**

- ① **int** size() {...}
- ② **boolean** isEmpty() {...}
- ③ **E** first() {...}
- ④ **E** last() {...}
- ⑤ **void** addBetween(**E** e, **Node<E>** predecessor, **Node<E>** successor) {...}
- ⑥ **E** remove(**Node<E>** node) {...}
- ⑦ **void** addFirst(**E** e) {...}
- ⑧ **void** addLast(**E** e) {...}
- ⑨ **E** removeFirst() {...}
- ⑩ **E** removeLast() {...}
- ⑪ **String** toString() {...}
- ⑫ **Iterator<E>** iterator() {...}

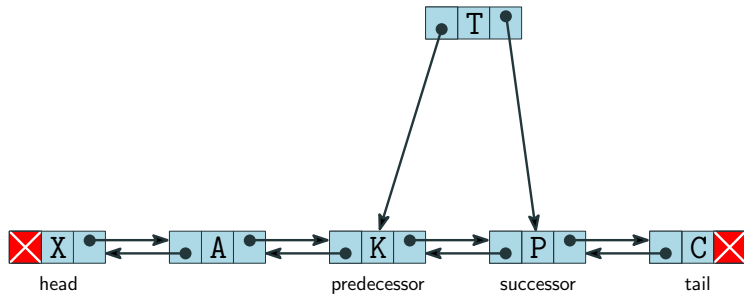
Illustrating addBetween(E e, Node<E> predecessor, Node<E> successor)



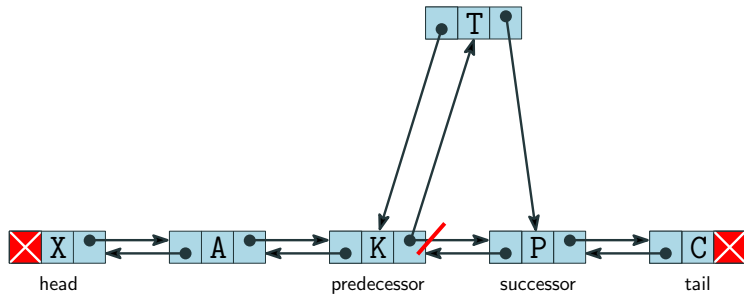
Illustrating addBetween(E e, Node<E> predecessor, Node<E> successor)



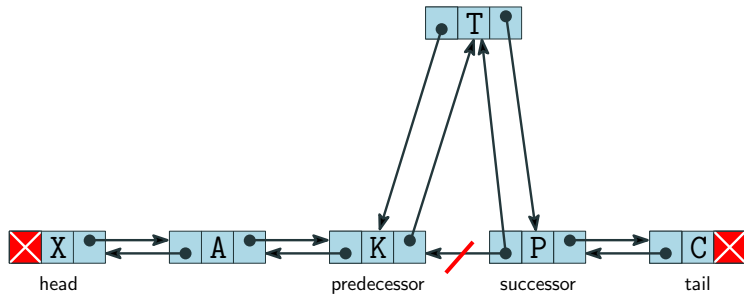
Illustrating addBetween(E e, Node<E> predecessor, Node<E> successor)



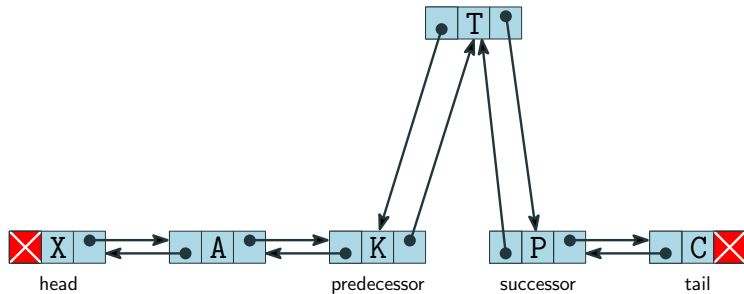
Illustrating addBetween(E e, Node<E> predecessor, Node<E> successor)



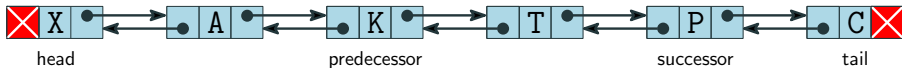
Illustrating addBetween(E e, Node<E> predecessor, Node<E> successor)



Illustrating addBetween(E e, Node<E> predecessor, Node<E> successor)



Illustrating addBetween(E e, Node<E> predecessor, Node<E> successor)



Complexity stuff

DoublyLinkedList<E>

- ① `int size() {...}`, **Time complexity:** $O(1)$
- ② `boolean isEmpty() {...}`, **Time complexity:** $O(1)$
- ③ `E first() {...}`, **Time complexity:** $O(1)$
- ④ `E last() {...}`, **Time complexity:** $O(1)$
- ⑤ `void addBetween(E e, Node<E> predecessor, Node<E> successor) {...}`
Time complexity: $O(1)$
- ⑥ `E remove(Node<E> node) {...}`, **Time complexity:** $O(1)$
- ⑦ `void addFirst(E e) {...}`, **Time complexity:** $O(1)$
- ⑧ `void addLast(E e) {...}`, **Time complexity:** $O(1)$
- ⑨ `E removeFirst() {...}`, **Time complexity:** $O(1)$
- ⑩ `E removeLast() {...}`, **Time complexity:** $O(1)$
- ⑪ `String toString() {...}`, **Time complexity:** $O(n)$ where n is the number of nodes in the list currently

For most linked list methods, traversing is required
Two popular approaches for traversing a linked list

Option A: while loop

```
Node<E> current = head;  
while( current != null ) {  
    // do something  
    current = current.next;  
}
```

Option B: for loop

```
Node<E> current = head;  
for(int pos = 0; pos < size; pos++) {  
    // do something  
    current = current.next;  
}
```

Words of caution



- In some cases, linked lists can be slow especially when you are trying to do many insertions since every time a memory allocation request has to be made
- In Java, for a large number of insertions, ArrayLists can beat singly linked lists in terms of speed
- Linked lists can take substantially more space per data element compared to arrays and ArrayLists since every node is storing at least one link (reference to another node in the same list)