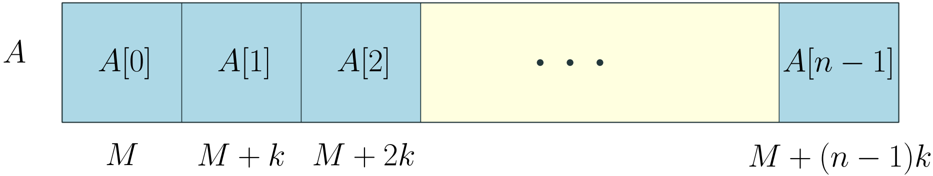
Arrays and Linked Lists

Dr. Anirban Ghosh

School of Computing
University of North Florida

Definition

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



M is the memory address of $A[0]$ and k is the space occupied by an array cell

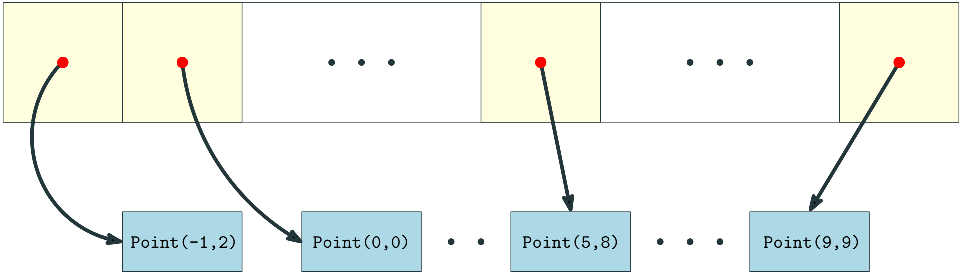
Your first data structure!

Arrays of primitives vs Arrays of objects

An array of **primitives**

10	-13	$\cdots$	5	$\cdots$	11

An array of **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

```java
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

```java
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`

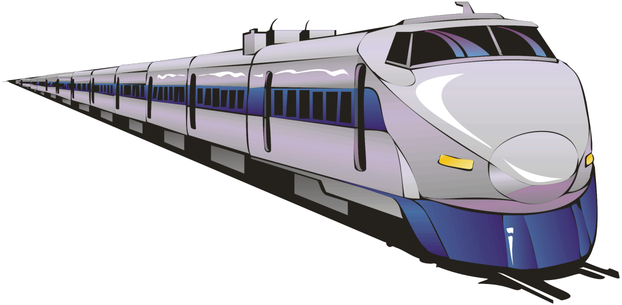
A few things about arrays

Good things

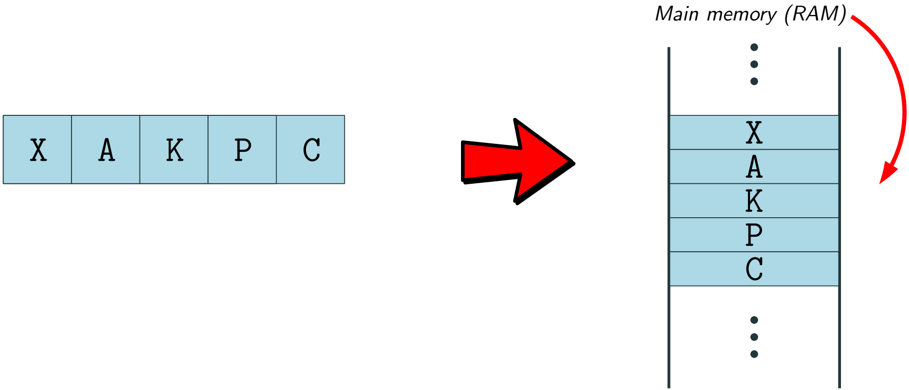
- Fast accesses for reading and writing since every location is indexed; every access takes $O(1)$ time (constant time)
- Once an array is allocated no more memory allocation worries!
- Super-simple coding (`arr[i] = 10; a[i] = b[j] + c[k]; x = t[i]; etc.`)

Bad things

- Need to know array-size in advance otherwise, an array may run out of space for holding the incoming items; on the other hand, if too much of extra space is allocated in advance, space may remain unused (*bad memory utilization*)
- Cannot grow (arrays are **static**)
- Insertions/deletions can be expensive since right/left shifts are required which take $O(n)$ time in the worst case

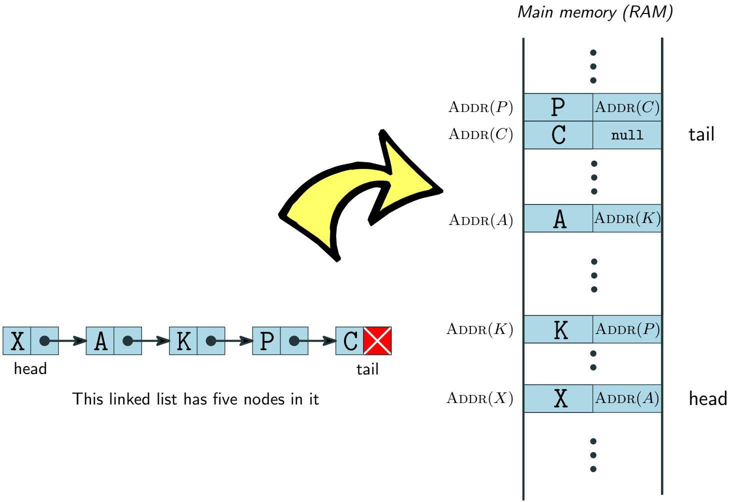


Linked Lists (Singly/Doubly/Circular)
Our first dynamic data structure



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!)

```java
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!)

```java
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

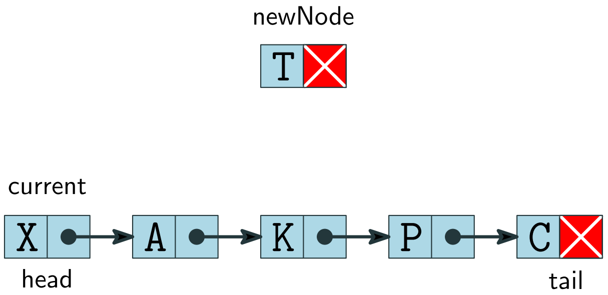
☞ Coding can get a bit challenging at times. **`NullPointerException`** is a common thing

It is time to go for generics!

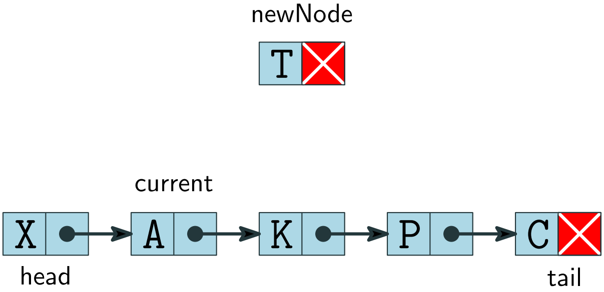
```
public class SinglyLinkedList<E> implements
Iterable<E>
```

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() {...}`

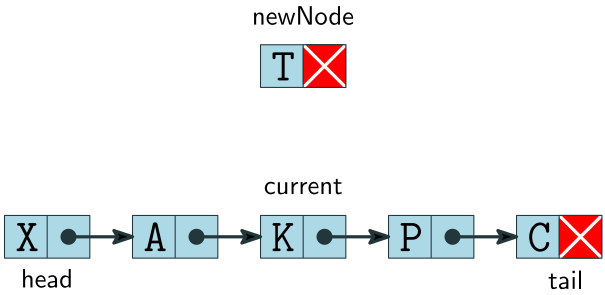
Adding a new node after a node: `addAfter('K','T')`



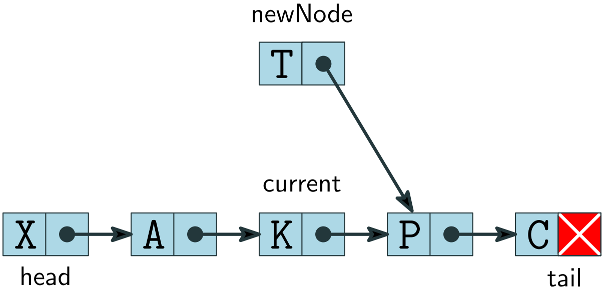
Adding a new node after a node: `addAfter('K','T')`



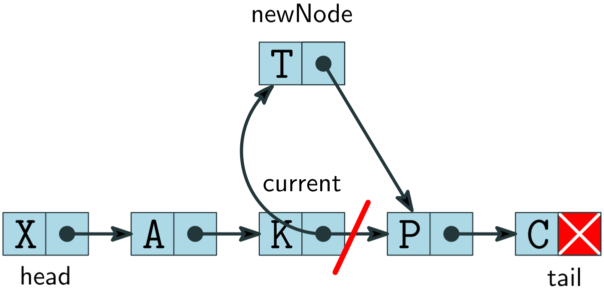
Adding a new node after a node: `addAfter('K','T')`



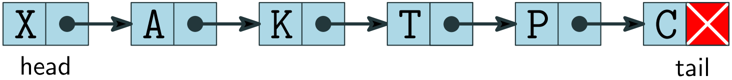
Adding a new node after a node: `addAfter('K','T')`



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



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



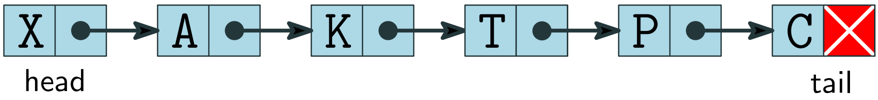
Complexity stuff

☞ 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



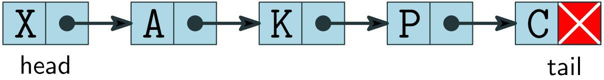
- Given just a reference to a node, we **cannot** efficiently delete it or add a node before it
- **Cannot** traverse from right to left just by following the links, if needed

Solution

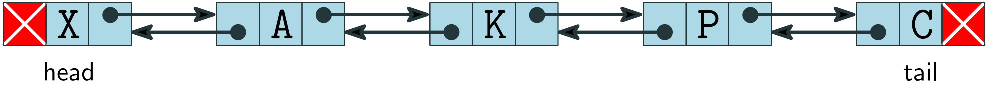
Store two links at every node: **prev** and **next**

Downside?

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



A **singly** linked list



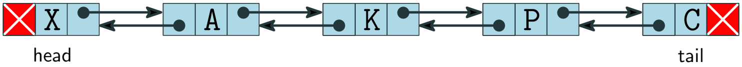
A **doubly** linked list

Methods

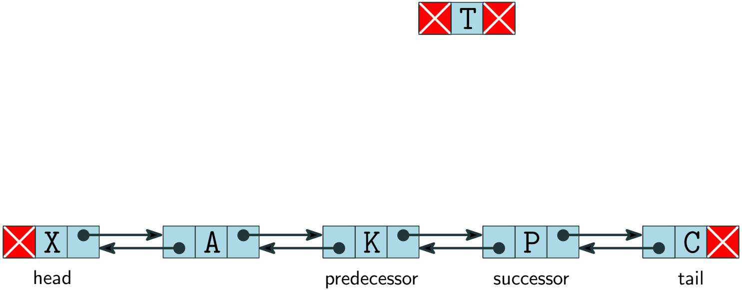
```
public class DoublyLinkedList<E> implements
                Iterable<E>
```

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() {...}`

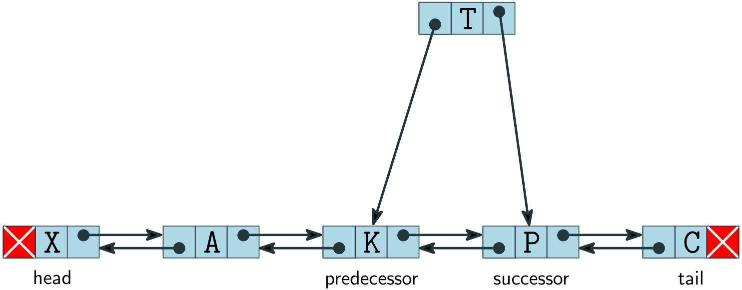
Illustrating `addBetween(E e, Node<E> predecessor, Node<E> successor)`



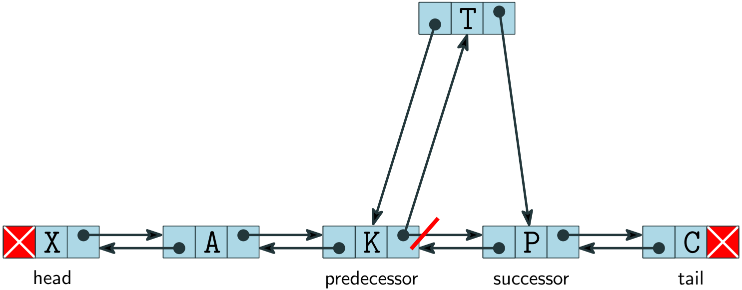
Illustrating `addBetween(E e, Node<E> predecessor, Node<E> successor)`



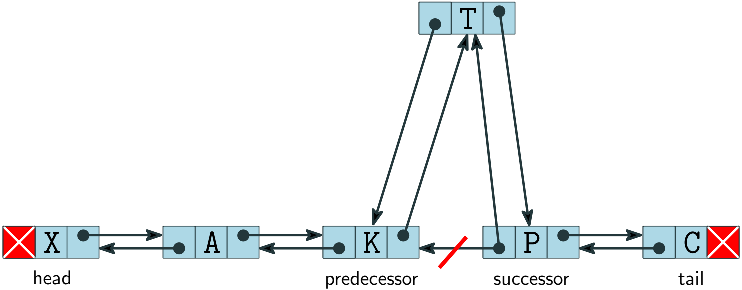
Illustrating `addBetween(E e, Node<E> predecessor, Node<E> successor)`



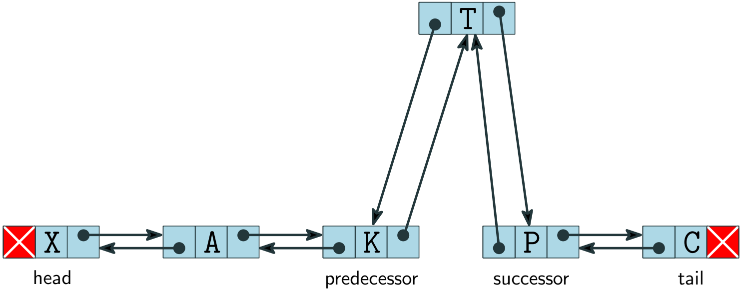
Illustrating `addBetween(E e, Node<E> predecessor, Node<E> successor)`



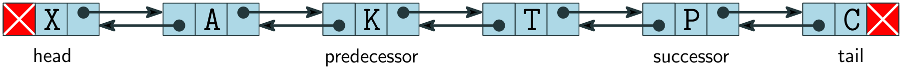
Illustrating `addBetween(E e, Node<E> predecessor, Node<E> successor)`



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



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



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;
}
```

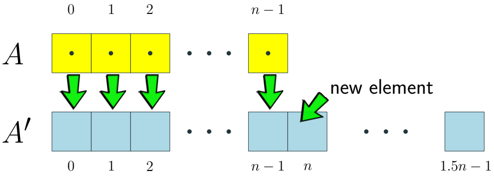
- Linked lists can take substantially more space per data element compared to arrays and `ArrayLists` since every node is storing at least one link (reference to another node in the same list)
- Linked lists are slower than `ArrayLists`, especially when trying to execute many insertions at a go; every time an insertion is made, a memory allocation request has to be made for the new node. Further, more primitive operations need to be executed in the case of a linked list.

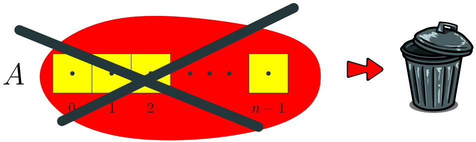
SLLs vs ArrayLists

```java
Random generator = new Random();

SinglyLinkedList<Integer> LL = new SinglyLinkedList<>();

int n = 1000000;

long startA = System.currentTimeMillis();
for(int i = 0; i < n; i++) {
   Integer randomInteger = generator.nextInt();
   LL.addLast(randomInteger);
}
long timeTakenA = System.currentTimeMillis() - startA;
System.out.println("Time taken by SLL: " + timeTakenA + " ms" ); // printed 162 ms in some run

ArrayList<Integer> AL = new ArrayList<>();

long startB = System.currentTimeMillis();
for(int i = 0; i < n; i++) {
   Integer randomInteger = generator.nextInt();
   AL.add(randomInteger);
}
long timeTakenB = System.currentTimeMillis() - startB;
System.out.println("Time taken by AL: " + timeTakenB + " ms" ); // printed 34 ms in the same run
```

So what is so special about `ArrayLists`?

- They are **dynamic** arrays (can insert/delete elements) and are maintained using plain arrays
- If there is no space for inserting a new element, a new array A' is created whose length is 1.5 times that of the current array A; the elements in A are copied to A' and then the new incoming element is inserted at the first available spot from the left. Finally, A is deleted and A' becomes the new A.





- It means that we seldom need to allocate new memory since often empty cells are available for insertion at the end of the array.
- `ArrayLists` are thus fast and we also get indexing support!

How many times new array allocations are needed?

n **(upto)**	**Array reallocations required**
10	1
100	7
$1,000$	13
$10,000$	19
$100,000$	24
$1,000,000$	30
$10,000,000$	36

Initial array size is assume to be 10

☞ Even for inserting up to $10M$ elements, at most 35 reallocations are needed!

☞ See the class `COP3530ArrayList` for an implementation

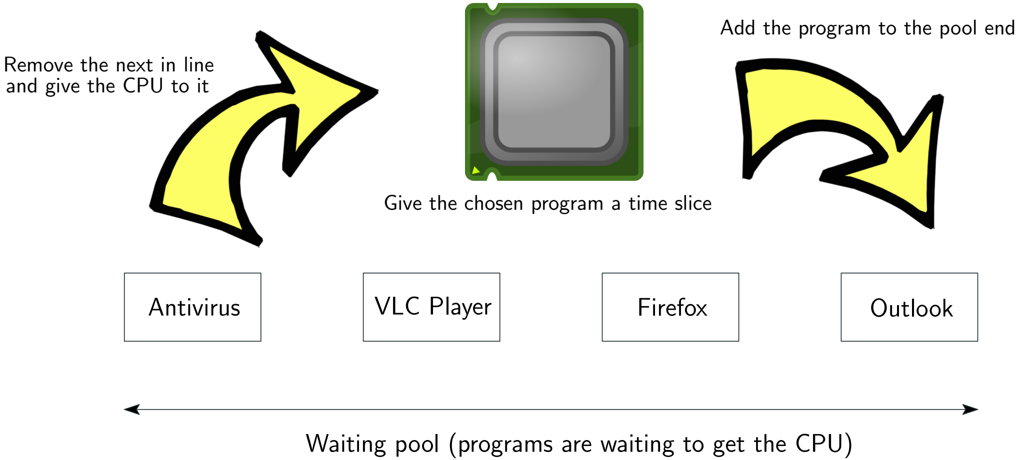
Why is the `StringBuilder` class so much faster than `String` class?

- **`StringBuilder`** is implemented as **dynamic array** and its objects are **mutable**
- In this case, **mutable** means if space is available, new items can be inserted
- Recall that dynamic arrays are very fast when we need to append a large number of times; that's why `StringBuilder` beats `String` when it comes to a appending a large number of characters
- On the other hand, **`String`** objects are **immutable** and every time a new append is executed, the existing `String` object is discarded and a new one is created *without increasing the new array capacity by a multiplicative factor*. Of course, the old content is transferred to the new object. Therefore, for n appends, approximately, n reallocations are needed.
- Such a large number of reallocations (approximately, n) drastically slows down `String` appends in practice!

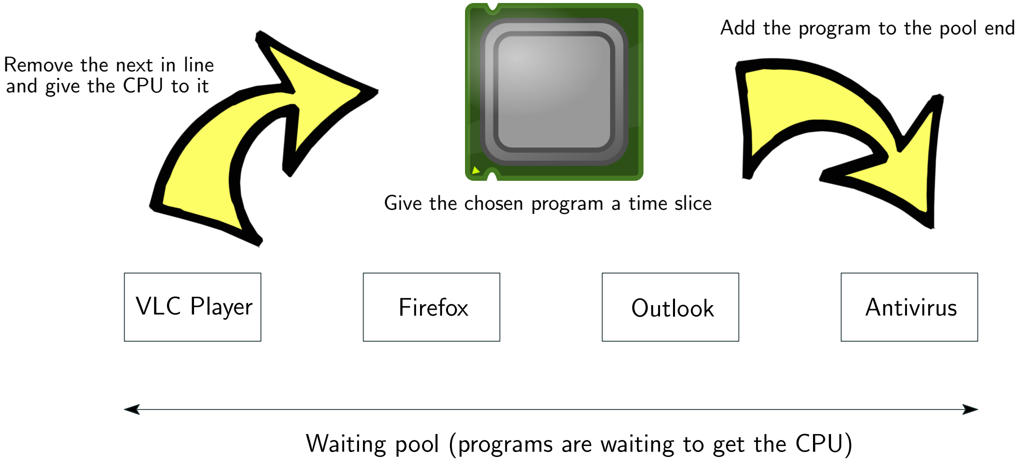
Maintaining a cyclic order

- In some applications we need the following two features:
  1. Lists should be **dynamic**
  2. Lists should have a **cyclic order**: well-defined neighboring relationships but no fixed beginning or end
- Example: maintain a cyclic list of processes (programs in execution) which need CPU access in the cyclic order (this is known as **round-robin scheduling** in the operating system literature)

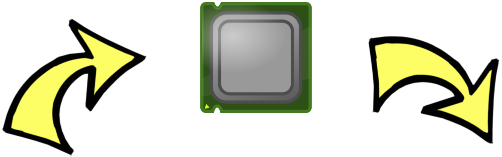
Example



Remove the next in line
and give the CPU to it
Give the chosen program a time slice
Add the program to the pool end
Antivirus
VLC Player
Firefox
Outlook
Waiting pool (programs are waiting to get the CPU)

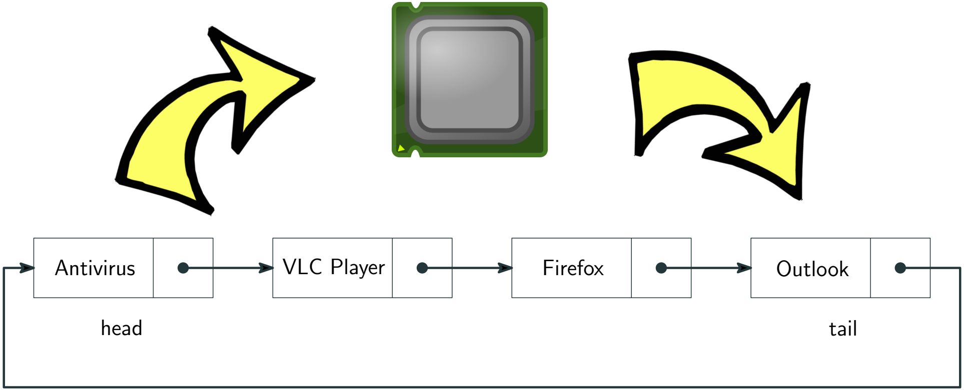

Remove the next in line
and give the CPU to it
Give the chosen program a time slice
Add the program to the pool end
VLC Player
Firefox
Outlook
Antivirus
Waiting pool (programs are waiting to get the CPU)

Idea! make the tail point to the head

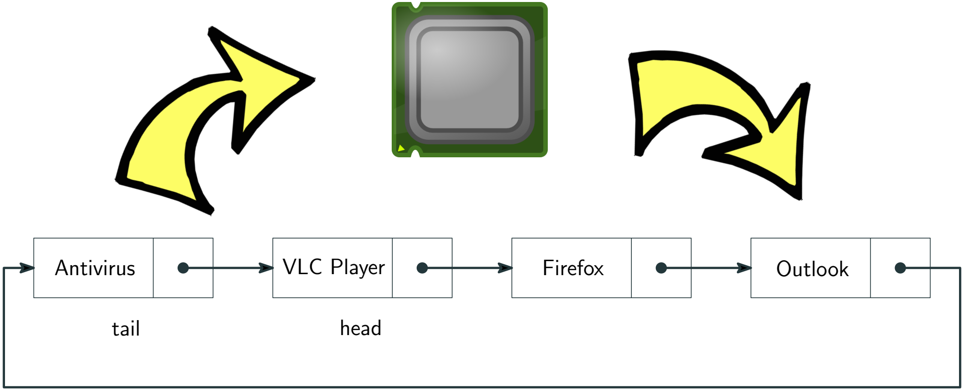


Antivirus	•	→	VLC Player	•	→	Firefox	•	→	Outlook	•
head									tail	

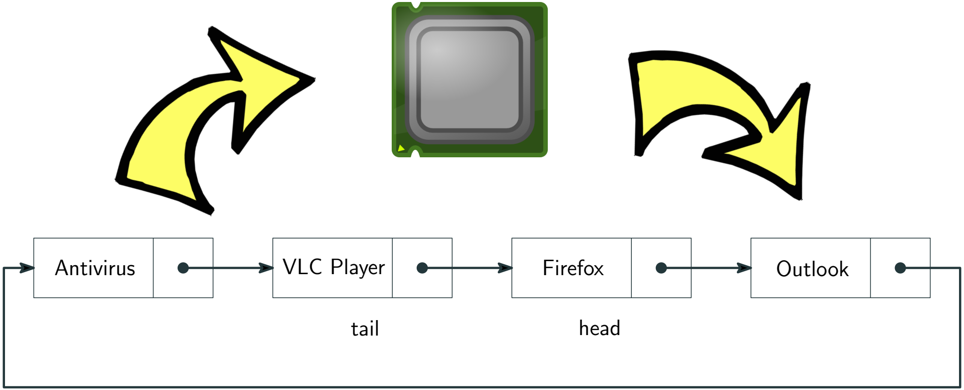
Circular linked list



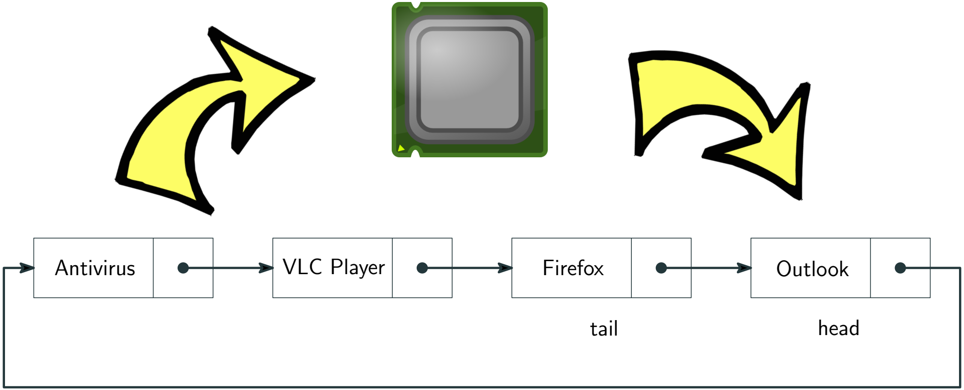
Circular linked list



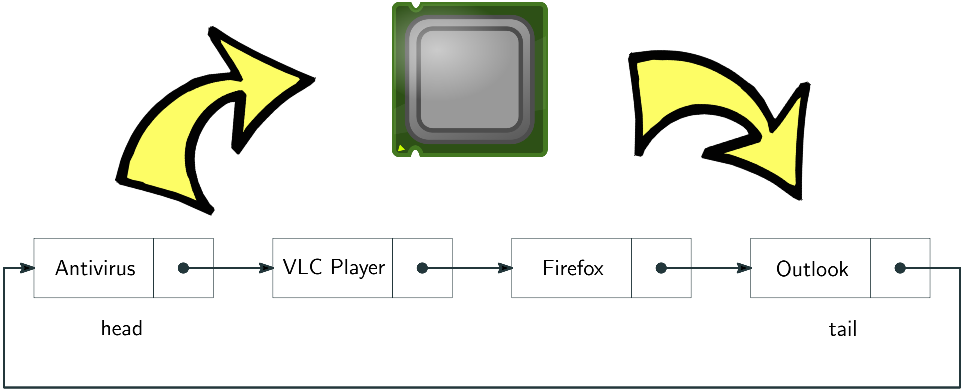
Circular linked list



Circular linked list



Circular linked list



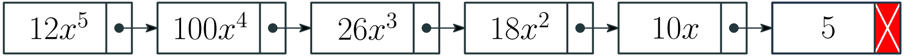
Methods

public class CircularLinkedList<E> implements Iterable<E>

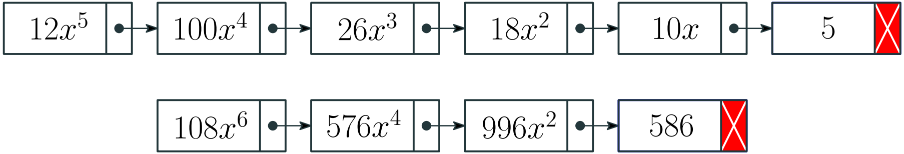
1. `int size() {...}`, **Time complexity**: $O(1)$
2. `boolean isEmpty() {...}`, **Time complexity**: $O(1)$
3. `E first() {...}`, **Time complexity**: $O(1)$
4. `E last() {...}`, **Time complexity**: $O(1)$
5. `void rotate() {...}`, **Time complexity**: $O(1)$
6. `void addFirst(E e) {...}`, **Time complexity**: $O(1)$
7. `void addLast(E e) {...}`, **Time complexity**: $O(1)$
8. `E removeFirst() {...}`, **Time complexity**: $O(1)$
9. `Iterator<E> iterator() {...}`

Representing polynomials using linked lists

- Polynomials can be effectively represented using linked lists where every node holds a polynomial term. The terms are **reversely sorted** based on their exponents. For example, the polynomial $p(x) = 12x^5 + 100x^4 + 26x^3 + 18x^2 + 10x + 5$ is represented as:



- Now, we will see how to add two polynomials represented using two different linked-lists. Example:



- Let the two input polynomials be denoted by $p(x)$ and $q(x)$. The objective is to compute a linked-list representation of $r(x) = p(x) + q(x)$, when the two linked-list representations of $p(x)$ and $q(x)$ are supplied.
- If $q(x) = 0$, return $p(x)$ since in this case, $r(x) = p(x) + q(x) = p(x) + 0 = p(x)$. Similarly, if $p(x) = 0$, return $q(x)$.

Algorithm

Computing $r(x) = p(x) + q(x)$

```
nextTermFromP = p.head.term, nextTermFromQ = q.head.term;
while( there are un-scanned nodes both in p and q) {
    if( nextTermFromP.exponent == nextTermFromQ.exponent  ) {
      Add nextTermFromP and nextTermFromQ and append a new node with the result term
      in the linked-list r. Update nextTermFromP and nextTermFromQ using the next two
      terms from the polynomials p and q, respectively;
    }
    else if( nextTermFromP.exponent > nextTermFromQ.exponent ) {
         Append a new node with the term nextTermFromP in the linked-list r
         and update nextTermFromP using the next term from p;
    }
    else {
         Append a new node with the term nextTermFromQ in the linked-list r
         and update nextTermFromQ using the next term from q;
    }
}
If there are un-scanned nodes from p, send those to r (using a loop);
If there are un-scanned nodes from q, send those to r (using a loop);
```

Efficiency

- Assume that there are m nodes in $p(x)$'s linked-list and n in $q(x)$'s linked-list
- Total work done by the algorithm is proportional to the number of nodes in both the linked-lists. To see this observe that we visit every node in both the lists exactly once.
- The `addLast()` method is invoked no more than $n+m$ times and each such call takes $O(1)$ time.
- Every iteration of the while-loop takes $O(1)$ time. So, the total time taken by the while-loop is $O(m+n)$ since it iterates at most $n+m$ times.
- Line 17: runs in $O(m)$ time and line 18: in $O(n)$ time.
- Therefore, the runtime of the add algorithm is:

$$O(m+n)+O(m)+O(n)=O(2m+2n)=O(2(m+n))=O(m+n)$$

See the class `Polynomial` to see how polynomial addition algorithm can be implemented efficiently

The `Cloneable` interface

- Cloning an array is not difficult since we can simply use the `clone()` method on it (does deep-copy for **primitives** and shallow-copy for **non-primitives**)
- What if we want to clone an object of an user-defined class X? We need to make the class X cloneable so the `clone()` method can be invoke on it.

```java
// Reference: https://docs.oracle.com/en/java/javase/17/docs//api/java.base/java/lang/Cloneable.html
public class Polynomial implements Cloneable, Iterable<Polynomial.PolynomialTerm> {
   // other variables and methods

    public Polynomial clone() throws CloneNotSupportedException {
       var clonePolynomial = (Polynomial) super.clone();

       clonePolynomial.L = new SinglyLinkedList<>();
       for( var term : L ) // create identical terms in the clone
          clonePolynomial.L.addLast( new PolynomialTerm(term.coefficient, term.exponent) );
       return clonePolynomial;
    }
}
```

- Inside the `clone()` method, ensure you set up every instance variable of the clone object appropriately; otherwise, the cloned object will not be a replica!

Recommended

Make the `SinglyLinkedList` and `DoublyLinkedList` classes cloneable

Chapter 5 from

`https://opendsa-server.cs.vt.edu/OpenDSA/Books/CS3/html/`