Data Structures and Algorithms
(ECEG 4171)

Chapter Three
Fundamental Data Structures

Ephrem A. (M.Sc.)
School of Electrical and Computer Engineeing
Chair of Computer Engineering
October 26, 2019

Abstract Data Types & Data Structures

- Oftentimes abstract data types (ADT) and data structures are used as synonymous. However, it's important to view them as follows:
  - An Abstract Data Type (ADT) is a data type whose representation is hidden from the client. It is a definition of new type, describes its properties and operations.
    - You can formally define (i.e., using mathematical logic) what an ADT is/does. *e.g.*, a Stack is a list implements a LIFO policy on additions/deletions.
  - A data structure is more concrete. Typically, it is a technique or strategy for implementing an ADT. *e.g.* Use a linked list or an array to implement a stack class.
- Some common ADTs that all trained programmers know about:
  - stack, queue, priority queue, dictionary, sequence, set
- Some common data structures used to implement those ADTS:
  - array, linked list, hash table (open, closed, circular hashing)
  - trees (binary search trees, heaps, AVL trees, 2-3 trees, tries, red/black trees, B-trees)

Data Structures Classification

- Data structures are classified into two types:
  1. Linear data structures: Elements are accessed in a sequential order but it is not compulsory to store all elements sequentially (say, Linked Lists). Examples: Linked Lists, Stacks and Queues.
  2. Non-linear data structures: Elements stored/accessed in a nonlinear order. Examples: Trees and graphs.

Abstract Data Types (ADTs)

- To specify the behavior of an ADT, we use an application programming interface (API), which is a list of constructors and instance methods (operations), with an informal description of the effect of each.
- **Example:**

public class	Counter	
	Counter(String id)	*Create a counter named id*
void	increment()	*increment the counter by one*
int	tally()	*number of increments since creation*
String	toString()	*string representation*

```
public class Flips{
  public static void main(String[] args){
    int T = Integer.parseInt(args[0]);
    Counter heads = new Counter("heads");
    Counter tails = new Counter("tails");
    for (int t = 0; t < T; t++)
      if (Math.random() > 0.5)
        heads.increment();
      else tails.increment();
    System.out.println(heads);
    System.out.println(tails);
  }
}
```

Introduction to Java Generics

- At its core the term *generics* means *parameterized types*.
- Parameterized types enable you to create classes, interfaces, and methods in which the type of data upon which they operate is specified as a parameter.
- Using generics, it is possible to create a single class that automatically works with different types of data.
- A class, interface, or method that operates on a parameterized type is called generic, as in *generic class* or *generic method*.
- Generics work only with objects and differ based on their type arguments.

A Simple Generics Example

```
class Gen<T> {
  T ob; // declare an object of
        type T
  Gen(T o) {
    ob = o;
  }
  T getob() {
    return ob;
  }
  void showType() {
    System.out.println("Type of T
          is " +
          ob.getClass().getName());
  }
}
```

```
class GenDemo {
  public static void main(String args[]) {
    Gen<Integer> iOb = new Gen<Integer>(88);
    iOb.showType();
    int v = iOb.getob();
    System.out.println("value: " + v);
    System.out.println();
    Gen<String> strOb = new Gen<String>("Generics
          Test");
    strOb.showType();
    String str = strOb.getob();
    System.out.println("value: " + str);
  }
}
```

Output

```
Type of T is java.lang.Integer
value: 88

Type of T is java.lang.String
value: Generics Test
```

Here, **T** is the name of a type parameter.

Because Gen uses a type parameter, Gen is a generic class, which is also called a *parameterized type*.

How Generics Improve Type Safety

Prior to Java SE 5, generic programming was implemented by relying heavily on Java's **Object class**, which is the universal supertype of all objects.

Example

```
public class ObjectPair {
  Object first;
  Object second;
  public ObjectPair(Object a, Object b) {
    first = a;
    second = b;
  }
  public Object getFirst() { return
       first; }
  public Object getSecond() { return
       second;}
}
```

Declaration and instantiation:

```
ObjectPair bid = new ObjectPair("ORCL",
     32.07);
```

However, the following is wrong:

```
String stock = bid.getFirst(); // compile
```

How Generics Improve Type Safety

Prior to Java SE 5, generic programming was implemented by relying heavily on Java's **Object class**, which is the universal supertype of all objects.

Example

```
public class ObjectPair {
  Object first;
  Object second;
  public ObjectPair(Object a, Object b) {
    first = a;
    second = b;
  }
  public Object getFirst() { return
       first; }
  public Object getSecond() { return
       second;}
}
```

Declaration and instantiation:

```
ObjectPair bid = new ObjectPair("ORCL",
     32.07);
```

However, the following is wrong:

```
String stock = bid.getFirst(); // compile
```

Using Generics Framework

```
public class Pair<A,B> {
  A first;
  B second;
  public Pair(A a, B b) {
    first = a;
    second = b;
  }
  public A getFirst() { return first; }
  public B getSecond() { return second;}
}
```

Declaration and instantiation:

```
Pair<String,Double> bid = new
     Pair<>("ORCL", 32.07); //Typer
     inference. OR
Pair<String, Double> bid = new
     Pair<String, Double>("ORCL",
```

Generics and Arrays

1. Code outside a generic class may wish to declare an array storing instances of the generic class with actual type parameters.

```
Pair<String,Double>[ ] holdings;
holdings = new Pair<String, Double>[25]; // illegal; compile error
holdings = new Pair[25]; // correct, but warning about unchecked cast
holdings[0] = new Pair<>("ORCL", 32.07); // valid element assignment
```

2. A generic class may wish to declare an array storing objects that belong to one of the formal parameter types.

```
public class Portfolio<T> {
  T[] data;
  public Portfolio(int capacity) {
    data = new T[capacity]; // illegal; compiler error
    data = (T[]) new Object[capacity]; // legal, but compiler warning
  }
  public T get(int index) { return data[index]; }
  public void set(int index, T element) { data[index] = element; }
}
```

Generic Methods

To define generic methods, we include a generic formal type declaration among the method modifiers.

Example: a parameterized static method that can reverse an array containing elements of any object type.

```
public class GenericDemo {
  public static <T> void reverse(T[] data) {
    int low = 0, high = data.length - 1;
    while (low < high) { // swap data[low] and data[high]
      T temp = data[low];
      data[low++] = data[high]; // post-increment of low
      data[high--] = temp; // post-decrement of high
    }
  }
}
```

Bounded Generic Types

A formal parameter type can be restricted by using the extends keyword followed by a class or interface. In that case, only a type that satisfies the stated condition is allowed to substitute for the parameter.

```
public class MyAnimalList<T extends Animal> {
```

Introduction to Nested Classes

- In Java, it is possible to define a class within another class, such classes are known as *nested classes*.
- Nested classes enable you to logically group classes that are only used in one place.
- This increases the use of encapsulation and reduce name conflicts, and create more readable and maintainable code.
- **Syntax:**

```
class OuterClass{
  ....
  class NestedClass{
    ....
  }
}
```

Nested Class

- The scope of a nested class is bounded by the scope of its enclosing class. Thus, class `NestedClass` does not exist independently of class `OuterClass`.
- A nested class has access to the members, including private members, of the class in which it is nested. However, the reverse is not true i.e. the enclosing class does not have access to the members of the nested class.
- A nested class is also a member of its enclosing class.
- As a member of its enclosing class, a nested class can be declared private, public, protected, default.
- Nested classes are divided into two categories:
  - **static nested class:** Nested classes that are declared static
  - **inner class:** A non-static nested class.

Static Nested Classes

- As with class methods and variables, a static nested class is associated with its outer class.
- And like static class methods, a static nested class cannot refer directly to instance variables or methods defined in its enclosing class.
- It can use them only through an object reference.
- They are accessed using the enclosing class name.

```
OuterClass.StaticNestedClass
```

- For example, to create an object for the static nested class, use this syntax:

```
OuterClass.StaticNestedClass nestedObject = new OuterClass.StaticNestedClass();
```

Static Nested Class Example

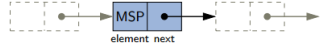
```
class OuterClass{
  static int outer_x = 10;
  int outer_y = 20;
  private static int outer_private = 30;

  static class StaticNestedClass{
    void display(){
      //can access static member of outer class
      System.out.println("outer_x = " + outer_x);

      //can access private static member of outer class
      System.out.println("outer_private = " + outer_private);

      //The following statement will give compilation error
      //as static nested class cannot directly access non-static
      //System.out.println("outer_y = " + outer_y);
    }
  }
}

public class StaticNestedClassDemo{
  public static void main(String[] args){
    //accessing a static nested class
    OuterClass.StaticNestedClass nestedObject = new
        OuterClass.StaticNestedClass();
```

Output

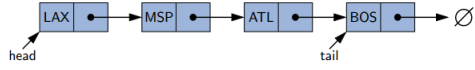
outer_x = 10
outer_private = 30

Linked Data Structures

- A linked list is a collection of nodes that collectively form a linear sequence.
- In a singly linked list (SLL), each node stores a reference to an object that is an element of the sequence, as well as a reference to the next node of the list.



- The linked list instance must keep a reference to the first node of the list, known as the head. The last node of the list is known as the tail.
- We can identify the tail as the node having null as its next reference by *traversing* through the nodes but storing explicit reference to the tail node is more efficient.

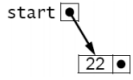


Consider the following node class

```
class Node {
  int data;
  Node next;
  Node(int data) {
    this.data = data;
  }
}
```

Notice that the Node class is now *self-referential*. Its *next* field is declared to have type Node.

```
//Constructing a linked list.
Node start = new Node(22);
```

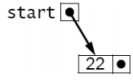


Consider the following node class

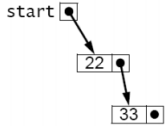
```
class Node {
  int data;
  Node next;
  Node(int data) {
    this.data = data;
  }
}
```

Notice that the Node class is now *self-referential*. Its *next* field is declared to have type Node.

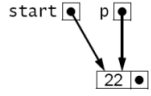
```
//Constructing a linked list.
Node start = new Node(22);
```



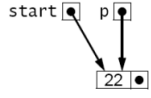

```
start.next = new Node(33);
```



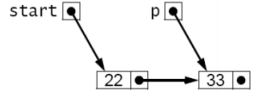
```
p = start = new Node(22);
```



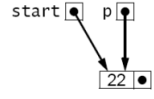
```
p = start = new Node(22);
```



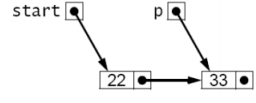

```
p = p.next = new Node(33);
```



```
p = start = new Node(22);
```




```
p = p.next = new Node(33);
```

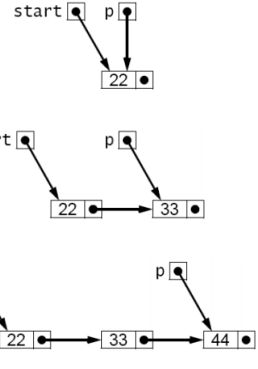



```
p = p.next = new Node(44);
```

```
p = start = new Node(22);
```

```
p = p.next = new Node(33);
```

```
p = p.next = new Node(44);
```



Traversing a Linked List

```
for (Node p = start; p != null; p = p.next) {
  System.out.println(p.data);
}
```

The output is:

```
22
33
44
```

Implementing a Singly Linked List Class

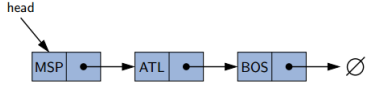
A complete implementation of a Singly Linked List class, supports the following methods:

size():	Returns the number of elements in the list.
isEmpty():	Returns true if the list is empty, and false otherwise.
first():	Returns (but does not remove) the first element in the list.
last():	Returns (but does not remove) the last element in the list.
addFirst(e):	Adds a new element to the front of the list.
addLast(e):	Adds a new element to the end of the list.
removeFirst():	Removes and returns the first element of the list.

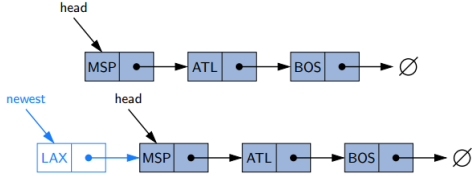
Implementing a SLL Class

```
public class SinglyLinkedList<E> {
  private static class Node<E> {
    private E element;
    private Node<E> next;
    public Node(E e, Node<E> n) {
      element = e;
      next = n;
    }
    public E getElement() { return element; }
    public Node<E> getNext() { return next; }
    public void setNext(Node<E> n) { next = n; }
  }
  private Node<E> head = null;
  private Node<E> tail = null;
  private int size = 0;
  public SinglyLinkedList() { }
  // access methods
  public int size() { return size; }
  public boolean isEmpty() { return size == 0; }
  public E first() {
    if (isEmpty()) return null;
    return head.getElement();
  }
  public E last() {
    if (isEmpty()) return null;
    return tail.getElement();
  }
}
```

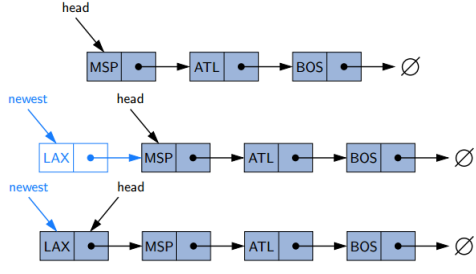
Inserting an Element at the Head of SLL



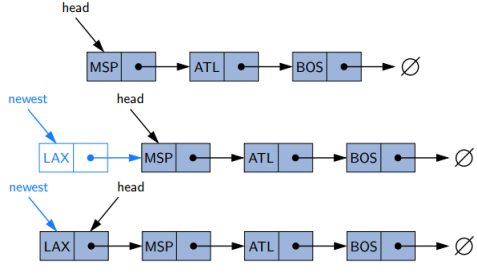
Inserting an Element at the Head of SLL



Inserting an Element at the Head of SLL

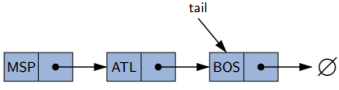


Inserting an Element at the Head of SLL

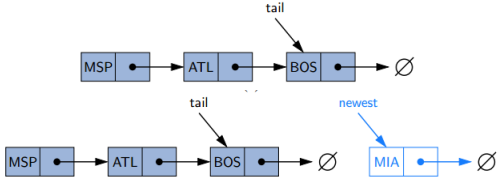



```
public void addFirst(E e) {
   head = new Node<>(e, head);
   if (size == 0)
      tail = head;
   size++;
}
```

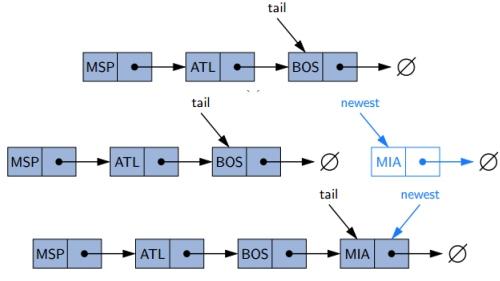
Inserting an Element at the Tail of a SLL



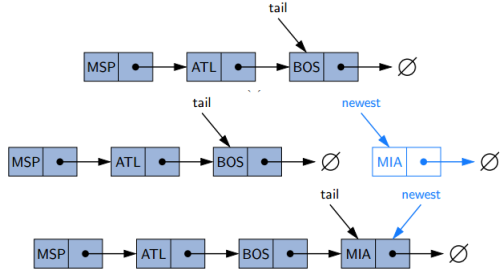
Inserting an Element at the Tail of a SLL



Inserting an Element at the Tail of a SLL

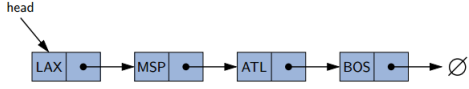


Inserting an Element at the Tail of a SLL

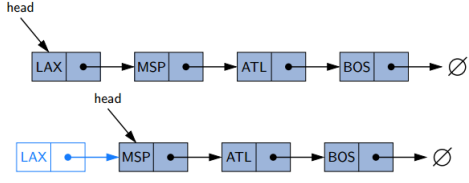



```
public void addLast(E e) {
   Node<E> newest = new Node<>(e, null);
   if (isEmpty())
      head = newest;
   else
      tail.setNext(newest);
   tail = newest;
   size++;
}
```

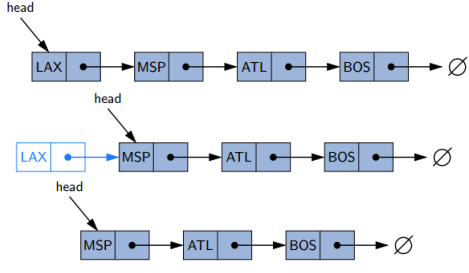
Removing an Element from a SLL



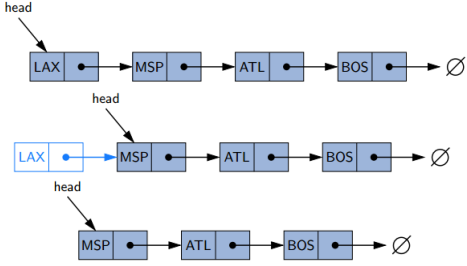
Removing an Element from a SLL



Removing an Element from a SLL



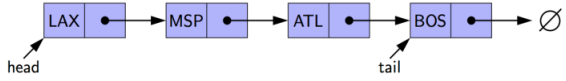
Removing an Element from a SLL




```
public E removeFirst() { // removes and returns the first element
   if (isEmpty()) return null;
   E answer = head.getElement();
   head = head.getNext();
   size--;
   if (size == 0)
      tail = null;
   return answer;
}
```

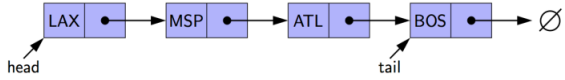
Removing at the Tail

- Removing at the tail of a singly linked list is not efficient ($O(n)$).
- There is no constant-time way to update the tail to point to the previous node.



Removing at the Tail

- Removing at the tail of a singly linked list is not efficient ($O(n)$).
- There is no constant-time way to update the tail to point to the previous node.




```
public E removeLast() { // removes and returns the last element
   if (isEmpty()) return null;
   if (size == 1){
      E answer = head.getElement();
      head = tail = null;
      size--;
      return answer;
   }
   else{
      E answer = tail.getElement();
      Node<E> temp = head;
      while(temp.getNext() != tail)
         temp = temp.getNext();
      temp.setNext(null);
      tail = temp;
      size--;
      return answer;
   }
}
```