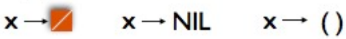
Dynamic Data Structures

Static vs Dynamic Data Structures

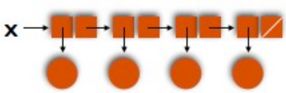
- memory to store all values of a static data structures is allocated at the same time
- e.g., all 100 slots of an array are allocated together when theArray = new Object[100];
- dynamic data structures are allocated incrementally as each new value is added

Linked Lists

- the empty linked list is called NIL



- each node of a linked list has a head with a value in it, and a tail with the rest of the linked list in it



- an element is added to a linked list (e.g. at the front) by pushing a new node onto it

Immutable Data Structures

- upon allocating memory for a data structure (in Java, with "new"), instance variables are initialized with code in the constructor
- an immutable data structure is one whose fields cannot be changed after initialization
- can be enforced in Java with the "final" attribute

Pros and Cons of Immutable Data

- pro — less prone to buggy code
- pro — nodes can be shared among different data objects
- con — can require less efficient algorithms than mutable data structures

```
public class List {
    private final Object data;
    private final List next;
    public static final List NIL = new List(null, null);

    private List (Object d, List n) {
        data = d;
        next = n;
    }
.
.
.
}
```

```java
public class List {
.
.
.
    /* factory methods for creating or extending Lists */

    public static List list (Object d) {
        return new List(d, List.NIL);
    }

    public List push (Object d) {
        return new List(d, this);
    }
.
.
.
}
```

```java
public class List {
.
.
.
    /* accessors */

    public Object head () {
        return data;
    }

    public List tail () {
        return next;
    }

    /* isEmpty test */

    public boolean isEmpty () {
        return this == List.NIL;
    }
.
.
.
}
```

```java
public class List {
.
.
.
    /* x.length() returns the number of elements in x.
       base case: an empty list has zero elements.
       otherwise: a list has one more element than its tail.
    */

    public int length () {
        return this.isEmpty() ?
            0 :
            1 + this.tail().length();
    }
.
.
.
}
```

```
public class List {
.
.
.
    /* x.find(y) returns List.NIL if y is not an element of x
       and returns the sublist of x whose head is y if y is
       an element of x.
       base case: return the list itself if it is empty or
                  if its head equals the object.
       otherwise: find the object in the tail.
    */

    public List find (Object d) {
        return this.isEmpty() || d.equals(this.head()) ?
            this :
            this.tail().find(d);
    }
.
.
.
}
```

```
public class List {
.
.
.
    /* this.append(that) returns a new List with all elements
       of this followed by all elements of that.
       base case: appending that onto the empty list is
                  just that.
       otherwise: append that onto the tail of this
                  and then push the head of this onto the
                  result.
    */

    public List append (List that) {
        return this.isEmpty() ?
            that :
            this.tail().append(that).push(this.head());
    }
.
.
.
}
```

```
public class List {
.
.
.
    /* x.reverse() returns a list with all elements of in
       reverse order.
       base case: reversing an empty list is the empty list.
       otherwise: reverse the tail, and then append a one-
                  element list containing the head onto the
                  end.
    */

    public List reverse () {
        return this.isEmpty() ?
            this :
         this.tail().reverse().append(List.list(this.head()));
    }
.
.
.
}
```

```
public class List {
.
.
.
    /* the nth element of a list (indexed from zero)
       base case: if n is zero, return the head.
       otherwise: return the n-1'st element of the tail.
    */

    public Object nth (int n) {
        return this.isEmpty() || n < 0 ? null :
               n == 0 ? this.head() :
               this.tail().nth(n - 1);
    }
.
.
.
}
```

```
public class List {
.
.
.
    /* x.delete(y) returns a new list with all elements of
       x except those that equal y.
       base case: an empty list has no elements, so return
                  the empty list.
       otherwise: if the head is equal to y, return the result
                  of deleting y from the tail.
       otherwise: return the result of deleting y from the
                  tail, but then push the head onto that
                  result.
    */
    public List delete (Object d) {
        return this.isEmpty() ? this :
               d.equals(this.head()) ? this.tail().delete(d) :
               this.tail().delete(d).push(this.head());
    }
.
.
.
}
```

```
public class List {
.
.
.
    /* x.insert(y) assumes that x is a list of Comparables
       in order and returns a new list with all the elements
       of x in addition to the new element y, inserted in the
       correct position.
    */

    public List insert (Comparable d) {
        return this.isEmpty() ||
               d.compareTo(this.head()) < 0 ?
            this.push(d) :
            this.tail().insert(d).push(this.head());
    }
.
.
.
}
```

```
public class List {
.
.
.
    public String toString () {
        return this.isEmpty() ?
            "()" :
            "(" + this.head().toString()
                + (this.tail().isEmpty() ? "" : " ")
                + this.tail().toString().substring(1);
    }
.
.
.
}
```

```
public class List {
.
.
.
    public static List parseIntList (String s) {
      int openBracket = s.indexOf('(');
      int closeBracket = s.indexOf(')');
      if (openBracket!=0 || closeBracket!=s.length() - 1)
        throw new IllegalArgumentException(s);

      String[] intStrings =
        s.substring(openBracket + 1, closeBracket).split(" ");
      List result = List.NIL;

      for (int i = intStrings.length - 1; i >= 0; i--)
        result = result.push(Integer.parseInt(intStrings[i]));

      return result;
    }
.
.
.
}
```