Abstract Data Types

- Sorting Arrays
- Linked Lists and Stacks
- Container Classes
- Primitive Storage

Common computational task
- Many data types can be ordered
  - Numbers, strings, dates, records
- A sorted list can be searched more efficiently
  - E.g., binary search algorithm
- Many different sorting algorithms
  - Efficiency depends on how wrong the initial order is
  - Algorithms can be evaluated on average, best case, worst case
Selection Sort

- Select the "right" value for a position in the list
  - Find minimum in list and make it first
  - Find minimum of rest and make it second
  - In general, find minimum in a[i]...a[n-1] and swap it with a[i]

```
3 4 7 8 23 34 41 45 89
```

- For list of n values, at ith step, there are n-i-1 comparisons to find the minimum

```
SelectionSort.java
```

Insertion Sort

- Insert value at correct position
  - First value okay to start in first slot
  - Compare second to first, swap if needed
  - Compare third to second, if needed, swap and compare to first, swap if needed
  - In general, take a[i] and insert in proper position of a[0]...a[i-1] by comparing and swapping with a[i-1], then a[i-2], etc., until in correct position

```
3 4 7 8 23 34 41 45 89
```

- At ith step, there may be n-i-1 comparisons and swaps

```
InsertionSort.java
```
Arrays are an effective way to store lists of data, but the size of an array is fixed:

- Number of items to store must be known when array is created.
- If more storage is needed as execution progresses, a larger array must be created and all data copied.
- If a value is removed, a smaller array must be created and all (but one) value copied.

Order in array is not easily changed:

- Inserting value requires all later values to be copied to later positions (carefully, so values are not lost).

Many applications require lists that grow, shrink, and/or rearrange:

- Linked lists provide general abstraction for efficient, flexible storage of a dynamic list of data.

Linked Lists

- All objects in Java are references:
  - So one object can contain a link to another object.
  - A linked list is formed of objects, where each object contains a link (a reference) to the next object in the list.
  - The first object is called the head of the list.
  - The last object does not link to a next object (reference is null).
  - Each item in the list is called a node.
A Simple Linked List of Integers

- Use private inner class to encapsulate value and link
  - Enclosing class can directly access inner class data

```java
public class IntList {
    private class IntNode {
        public int value;
        public IntNode next;
        public IntNode(int n) { value = n; next = null; }
    }
    private IntNode head;
    IntList() { head = null; }
}
```

Inner class for integer value and link to next one
Reference to first element
Initialize the list as empty

Linked List Patterns

- Common pattern for traversing a linked list where the temporary variable `cur` is the current node

```java
cur = head;
while ( cur != null ) {
    ...
    cur = cur.next;
}
```

Do something with this node
Advance to next node in list

- Can also code with a `for` loop

```java
for ( Node cur = head; cur != null; cur = cur.next ) {
    ...
}
```

Start with first node
until end of list
on to next item
Linked List Patterns

- Sometimes we need to deal with an empty list as a special case

```java
    if ( head == null ) {
        . . .
    } else {
        cur = head;
        while ( cur != null ) {
            . . .
            cur = cur.next;
        }
    }
```

- Special handling for empty list
- Normal handling for non-empty list

- Find the end of a list

```java
    cur = head;
    if ( cur != null ) {
        while ( cur.next != null )
            cur = cur.next;
    }
```

- Now cur is the node at end

- Find and remove elements of list

```java
    prev = null; cur = head;
    while ( cur != null ) {
        if ( cur.matches ) {
            if (prev == null) head = cur.next;
            else prev.next = cur.next;
        } else prev = cur;
        cur = cur.next;
    }
```

- Remove node by making its previous node refer to its next

IntList.java
### Java LinkedList Class

- Implementation for linked list of any object
  - Methods to add, remove, check if an element is in list, get first, get last, get element at particular position
  - Since Java 1.5, uses generic types to avoid need for casting
  - Example: use linked list of Strings like array:
    ```java
    LinkedList<String> slist = new LinkedList<String>();
    for (int i = 0; i < slist.size(); ++i)
        . . . slist.set(i, value) . . .
        . . . slist.get(i) . . .
    ```

- LinkedList also may use a Java Iterator for looping
- API for [LinkedList](https://docs.oracle.com/en/java/javase/11/docs/api/)
- LinkedList is just one of many Java container classes
  - ArrayList, Vector, Stack, PriorityQueue, TreeSet, HashSet, ...

### Other Data Structures

#### Stack
- Last in, first out list
- Add elements to one end of list - the push operation
- Remove from same end – the pop operation
- Easy to implement with linked list
  - Push inserts at beginning
  - Pop removes from beginning

#### Queue
- First in, first out list
- Add element at tail of list – the enqueue operation
- Remove element at head of list – the dequeue operation
- Like ticket line
  - [QueueTest.java](QueueTest.java)
Primitive Storage

- A number is stored in **bits** in memory
  - Zero is 0...00, One is 0...01, Two is 0...010, etc.
- For **n** bits, there are \(2^n\) possible values
  - \(2^8 = 256\) possible values for **byte**
  - \(2^{16} = 65,536\) possible values for **short**
  - \(2^{32} = 4,294,967,296\) possible values for **int**
  - etc.
- What about sign? (positive vs. negative)
  - An unsigned short would range from 0 to 65,535
  - A signed short would range from -32,768 to +32,767
  - Half of the values are negative, half are non-negative

How should we represent **signed** numbers with bits?

- Let's take a simple example – suppose there was a numerical data type with just **four** bits.
- That would be \(2^4 = 16\) possible values, which would range from -8 to +7, i.e.,
  - \(-8\ -7\ -6\ -5\ -4\ -3\ -2\ -1\ 0\ +1\ +2\ +3\ +4\ +5\ +6\ +7\)
- The bit patterns possible with four bits are:
  - \(0000\ 0001\ 0010\ 0011\ 0100\ 0101\ 0110\ 0111\ 1000\ 1001\ 1010\ 1011\ 1100\ 1101\ 1110\ 1111\)
- How should these bit patterns correspond to the numerical values?
## Primitive Storage

- For the non-negative values 0 through 7 it is natural to use:
  \[
  0000 \ 0001 \ 0010 \ 0011 \ 0100 \ 0101 \ 0110 \ 0111
  \]

- That is, first bit is sign bit, rest is absolute value
  - Gives us two representations of zero
  - Makes arithmetic difficult – have to check sign and decide what to do
  - For example, \(-4 + 1\) should be \(-3\), but normal binary arithmetic would give us the value \(-5\)
  - This approach of sign and magnitude was used in early computers

- Another approach is One's Complement
  - Invert all the bits to get the negative value, e.g., 1110 is \(-1\)
  - Also have two representations of zero
  - But arithmetic is a little easier

## Two's Complement

- For the non-negative values 0 through 7 use:
  \[
  0000 \ 0001 \ 0010 \ 0011 \ 0100 \ 0101 \ 0110 \ 0111
  \]

- Think of the numbers as a "ring" – when we get to the highest possible value, wrap around to the lowest possible value (the "most negative")

- This is called Signed Two's Complement
  - Gives us one representation of zero
  - Makes arithmetic easy
    - \(-4 + 1\) is \(-3\)
    - \(-2 + 3\) is \(1\)
Signed Two's Complement

- In Signed Two's Complement, just do arithmetic as normal binary arithmetic, but discard extra bits
- Arithmetic may result in overflow
  - 7 + 3 in binary would be 0111 + 0011 which is 1010, i.e., -6 (since we can't store the value 10 in a signed 4 bits)
  - 7 + 7 would be 0111 + 0111 which is 1110, i.e., -2
  - 4 * 4 would be 10000 (16 in binary), discarding excess bits gives 0000, or just 0
  - 7 * 7 would be 110001 (49 in binary), discarding excess bits leaves 0001, or just 1
- Find binary representation for a negative number
  - Take absolute value and subtract from 2^n (n= number of bits)
  - Or, invert all the bits of absolute value and add one
  - Twos.java