

# Introduction: Abstract Data Types

---

## Contents

- [Good Programs Use Abstraction](#)
  - [Abstract Data Types](#)
- 

## Good Programs Use Abstraction

What makes a program *good*?

1. It works (as specified!).
2. It is easy to understand and modify.
3. It is reasonably efficient.

One way to help achieve (2) (which helps with (1)) is to use **abstract data types**, or **ADTs**. The idea of an ADT is to separate the notions of **specification** (what kind of thing we're working with and what operations can be performed on it) and **implementation** (how the thing and its operations are actually implemented).

The benefits of using ADTs include:

- Code is easier to understand (e.g., it is easier to see "high-level" steps being performed, not obscured by low-level code).
- Implementations of ADTs can be changed (e.g., for efficiency) without requiring changes to the program that uses the ADTs.
- ADTs can be reused in future programs.

Fortunately for us, object-oriented programming languages (like Java) make it easy for programmers to use ADTs: each ADT corresponds to a **class** (or **Java interface** - more on this later) and the operations on the ADT are the class/interface's **public methods**. The user, or client, of the ADT only needs to know about the method **interfaces** (the names of the methods, the types of the parameters, what the methods do, and what, if any, values they return), not the actual implementation (how the methods are implemented, the private data members, private methods, etc.).

## Abstract Data Types

There are two parts to each ADT:

1. The **public** or **external** part, which consists of:
  - the conceptual picture (the user's view of what the object looks like, how the structure is organized)
  - the conceptual operations (what the user can do to the ADT)
2. The **private** or **internal** part, which consists of:
  - the representation (how the structure is actually stored)
  - the implementation of the operations (the actual code)

In general, there are many possible operations that could be defined for each ADT; however, they often fall into these categories:

1. initialize
2. add data
3. access data
4. remove data

In this class, we will study a number of different abstract data types, different ways to implement them, and different ways to use them. Our first ADT (coming up in the next set of notes) is the List.

# Lists

## Contents

- [The List ADT](#)
  - [Test Yourself #1](#)
- [Java Interfaces](#)
- [Lists vs. Arrays](#)
  - [Test Yourself #2](#)
- [The Java API and Lists](#)
- [Implementing the List ADT](#)
  - [Implementing the add methods](#)
  - [Test Yourself #3](#)
  - [Implementing the constructor](#)
- [Iterators](#)
  - [What are iterators?](#)
  - [Implementing iterators](#)
- [Testing](#)

## The List ADT

Our first ADT is the List: an ordered collection of items of some element type *E*. Note that this doesn't mean that the objects are in *sorted* order, it just means that each object has a *position* in the List, starting with position zero.

Recall that when we think about an ADT, we think about both the external and internal views. The external view includes the "conceptual picture" and the set of "conceptual operations". The conceptual picture of a List is something like this:

```
item 0
item 1
item 2
. . .
item n
```

and one reasonable set of operations is:

Operation	Description
void add(E item)	add item to the end of the List
void add(int pos, E item)	add item at position pos in the List, moving the items originally in positions pos through size()-1 one place to the right to make room (error if pos is less than 0 or greater than size())
boolean contains(E item)	return <b>true</b> iff item is in the List (i.e., there is an item x in the List such that x.equals(item))
int size()	return the number of items in the List
boolean isEmpty()	return <b>true</b> iff the List is empty
E get(int pos)	return the item at position pos in the List (error if pos is less than 0 or greater than or equal to size())
E remove(int pos)	remove and return the item at position pos in the List, moving the items originally in positions pos+1 through size()-1 one place to the left to fill in the gap (error if pos is less than 0 or greater than or equal to size())

Many other operations are possible; when designing an ADT, you should try to provide enough operations to make the ADT useful in many contexts, but not so many that it gets confusing. It is not always easy to achieve this goal; it will sometimes be necessary to add operations in order for a new application to use an existing ADT.

### TEST YOURSELF #1

**Question 1:** What other operations on Lists might be useful? Define them by writing descriptions like those in the table above.

**Question 2:** Note that the second add method (the one that adds an item at a given position) can be called with a position that is equal to size(), but for the get and remove methods, the position has to be *less* than size(). Why?

**Question 3:** Another useful abstract data type is called a Map. A Map stores unique "key" values with associated information. For example, you can think of a dictionary as a Map, where the keys are the words, and the associated information is the definitions.

What are some other examples of Maps that you use?

What are the useful operations on Maps? Define them by writing descriptions like those in the table above.

## Java Interfaces

The List ADT operations given in the table above describe the **public interface** of the List ADT, that is, the information the someone would need to know in order to be able to use a List (and keep in mind that the user of a List does not - and should not - need to know any details about the **private implementation**).

Now let's consider how we can create an ADT using Java. Java provides a mechanism for separating public interface from private implementation: interfaces. A **Java interface** (i.e., an interface as defined by the Java language) is very closely tied to the concept of a public interface. A Java interface is a reference type (like a class) that contains only (public) method signatures and class constants. Java interfaces are defined very similarly to the way classes are defined. A ListADT interface with the List ADT operations we've described would be defined as follows:

```
public interface ListADT<E> {
    void add(E item);
    void add(int pos, E item);
    boolean contains(E item);
    int size();
    boolean isEmpty();
    E get(int pos);
    E remove(int pos);
}
```

Everything contained in a Java interface is public so you do not need to include the `public` keyword in the method signatures (although you can if you want). Notice that, unlike a class, an interface does not contain any method implementations (i.e., the bodies of the methods). The method implementations are provided by a Java class that **implements** the interface. To implement an interface named `InterfaceName`, a Java class must do two things:

1. include `"implements InterfaceName"` after the name of the class at the beginning of the class's declaration, and
2. for each method signature given in the interface `InterfaceName`, define a public method with the exact same signature

For example, the following class implements the ListADT interface (although it is a *trivial* implementation):

```
public class ListImplementation<E> implements ListADT<E> {
    private E myItem;

    public ListImplementation() {
        myItem = null;
    }

    public void add(E item) {
        myItem = item;
    }

    public void add(int pos, E item) {
        myItem = item;
    }

    public boolean contains(E item) {
        return false;
    }

    public int size() {
        return 0;
    }

    public boolean isEmpty() {
        return false;
    }

    public E get(int pos) {
        return null;
    }

    public E remove(int pos) {
        return null;
    }
}
```

## Lists vs. Arrays

In some ways, a List is similar to an array: both Lists and arrays are ordered collections of objects and in both cases you can add or access items at a particular position (and in both cases we consider the first position to be position zero). You can also find out how many items are in a List (using its `size` method), and how large an array is (using its `length` field).

The main advantage of a List compared to an array is that, whereas the size of an array is fixed when it is created (e.g., `int[] A = new int[10]` creates an array of integers of size 10 and you cannot store more than 10 integers in that array), the size of a List can change: the size increases by one each time a new item is added (using either version of the `add` method) and the size decreases by one each time an item is removed (using the `remove` method).

For example, suppose we have an `ArrayList` class that implements the ListADT interface. Here's some code that reads strings from a file called `data.txt`

and stores them in a ListADT named words, initialized to be an ArrayList of Strings:

```
ListADT<String> words = new ArrayList<String>();
File infile = new File("data.txt");
Scanner sc = new Scanner(infile);
while (sc.hasNext()) {
    String str = sc.next();
    words.add(str);
}
```

If we wanted to store the strings in an array, we'd have to know how many strings there were so that we could create an array large enough to hold all of them.

One disadvantage of a List compared to an array is that whereas you can create an array of any size and then you can fill in any element in that array, a new List always has size zero and you can never add an object at a position greater than the size. For example, the following code is fine:

```
String[] strList = new String[10];
strList[5] = "hello"; !
```

but this code will cause a runtime exception:

```
ListADT<String> strList = new ArrayList<String>();
strList.add(5, "hello"); // error! can only add at position 0
```

Another (small) disadvantage of a List compared to an array is that you can declare an array to hold any type of item, including primitive types (e.g., int, char), but a List only holds Objects. Thus the following causes a compile-time error:

```
List<int> numbers = new ArrayList<int>();

% javac Test.java
Test.java:9: unexpected type
found   : int
required: reference
    List<int> numbers = new ArrayList<int>();
    ^
```

Fortunately, for each primitive type, there is a corresponding "box" class (also sometimes called a **wrapper class**). For example, an Integer is an object that contains one int value. Java has a feature called **auto-boxing** that automatically converts between int and Integer as appropriate. For example,

```
ListADT<Integer> numbers = new ArrayList<Integer>();

// store 10 integer values in list numbers
for (int k = 0; k < 10; k++) {
    numbers.add(k); // short for numbers.add(new Integer(k));
}

// get the values back and put them in an array
int[] array = new int[10];
for (int k = 0; k < 10; k++) {
    array[k] = numbers.get(k); // short for array[k] = numbers.get(k).intValue();
}
```

(Older versions of Java required that your code do the boxing up of a primitive value into the corresponding wrapper class explicitly; for example, here's the code that does the same thing as the example above using explicit boxing and unboxing:

```
ListADT<Integer> numbers = new ArrayList<Integer>();

// store 10 integer values in list numbers
for (int k = 0; k < 10; k++) {
    numbers.add(new Integer(k));
}

// get the values back and put them in an array
int[] array = new int[10];
for (int k = 0; k < 10; k++) {
    array[k] = numbers.get(k).intValue();
}
```

It's useful to know this since you may encounter code written using earlier versions of Java.)

---

## TEST YOURSELF #2

**Question 1:** Assume that variable words is a List containing k strings, for some integer k greater than or equal to zero. Write code that changes words to contain 2\*k strings by adding a new copy of each string right after the old copy. For example, if words is like this before your code executes:

"happy", "birthday", "to", "you"

then after your code executes words should be like this:

"happy", "happy", "birthday", "birthday", "to", "to", "you", "you"

**Question 2:** Again, assume that variable `words` is a List containing zero or more strings. Write code that removes from `words` all copies of the string "hello". Be sure that your code works when `words` has more than one "hello" in a row.

[solution](#)

## Implementing the List ADT

Now let's consider the "private" or "internal" part of a List ADT. We will consider two ways to implement the `ListADT` interface: using an array and using a linked list (the former is covered in this set of notes, the latter in another set of notes). Here's an outline of an implementation of the `ListADT` interface that uses an array to store the items. We will also simplify the implementation to consider only lists of `Objects`, not lists of specific classes such as `String`, `Integer`, etc. We will call our class `SimpleArrayList`. The bodies of the methods are not filled in yet.

```
public class SimpleArrayList implements ListADT<Object> {

    // *** fields ***
    private Object[] items; // the items in the List
    private int numItems;   // the number of items in the List

    // *** constructor ***
    public SimpleArrayList() { ... }

    /** required ListADT methods */

    // add items
    public void add(Object item) { ... }
    public void add(int pos, Object item) { ... }

    // remove items
    public Object remove(int pos) { ... }

    // get items
    public Object get (int pos) { ... }

    // other methods
    public boolean contains (Object item) { ... }
    public int size() { ... }
    public boolean isEmpty() { ... }
}
```

Note that the public methods provide the "external" view of the List ADT. Looking only at the signatures of the public methods of the `ListADT` interface (the method names, return types, and parameters), plus the descriptions of what they do (e.g., as provided in the [table above](#)), a programmer should be able to write code that uses `ListADTs` and `ArrayLists`. It is *not* necessary for a client of the `SimpleArrayList` class to see how the `SimpleArrayList` methods are actually implemented. The private fields and the bodies of the methods of the `SimpleArrayList` class provide the "internal" view - the actual implementation - of the List ADT.

## Implementing the add methods

Now let's think about how to implement some of the `SimpleArrayList` methods. First we'll consider the `add` method that has just one parameter (the object to be added to the List). That method adds the object to the end of the List. Here is a conceptual picture of what `add` does:

```
BEFORE:  item 0    item 1    item 2    ...    item n

AFTER:   item 0    item 1    item 2    ...    item n    NEW ITEM
```

Now let's think about the actual code. First, note that the array that stores the items in the List (the `items` array) may be full. If it is, we'll need to:

1. get a new, larger array, and
2. copy the items from the old array to the new array.

Then we can add the new item to the end of the List.

Note that we'll also need to deal with a full array when we implement the other `add` method (the one that adds a new item in a given position). Therefore, it makes sense to implement handling that case (the two steps listed above) in a separate method, `expandArray`, that can be called from both `add` methods. Since handling a full array is part of the *implementation* of the `SimpleArrayList` class (it is not an operation that users of the class need to know about), the `expandArray` method should be a private method.

Here's the code for the first `add` method and for the `expandArray` method.

```
/** *****
// add
//
// Given: Object item
//
// Do:    Add item to the end of the List
//
```

```

// Implementation:
//   If the array is full, replace it with a new, larger array;
//   store the new item after the last item
//   and increment the count of the number of items in the List.
//*****
public void add(Object item) {
    // if array is full, get new array of double size,
    // and copy items from old array to new array
    if (items.length == numItems) {
        expandArray();
    }

    // add new item; update numItems
    items[numItems] = item;
    numItems++;
}

//*****
// expandArray
//
// Do:
//   o Get a new array of twice the current size.
//   o Copy the items from the old array to the new array.
//   o Make the new array be this List's "items" array.
//*****
private void expandArray() {
    Object[] newArray = new Object[numItems*2];
    for (int k = 0; k < numItems; k++) {
        newArray[k] = items[k];
    }
    items = newArray;
}

```

In general, when you write code it is a good idea to think about special cases. For example, does `add` work when the List is empty? When there is just one item? When there is more than one item? You should think through these cases (perhaps drawing pictures to illustrate what the List looks like before the call to `add` and how the call to `add` affects the List). You can then decide if the code works as is or if some modifications are needed.

Now let's think about implementing the second version of the `add` method (the one that adds an item at a specified position in the List). An important difference between this version and the one we already implemented is that for this version a bad value for the `pos` parameter is considered an error. In general, if a method detects an error that it doesn't know how to handle, it should throw an exception. (Note that exceptions should *not* be used for other purposes like exiting a loop.) More information about exceptions is provided in a separate set of notes.

So the first thing our `add` method should do is check whether parameter `pos` is in range and if not, throw an `IndexOutOfBoundsException`. If `pos` is OK, we must check whether the `items` array is full and if so, we must call `expandArray`. Then we must move the items in positions `pos` through `numItems - 1` over one place to the right to make room for the new item. Finally, we can insert the new item at position `pos` and increment `numItems`. Here is the code:

```

//*****
// add
//
// Given: int pos, Object item
//
// Do:   Add item to the List in position pos (moving items over to the right
//       to make room).
//
// Exceptions:
//   Throw IndexOutOfBoundsException if pos<0 or pos>numItems
//
// Implementation:
//   1. check for bad pos
//   2. if the array is full, replace it with a new, larger array
//   3. move items over to the right
//   4. store the new item in position pos
//   5. increment the count of the number of items in the List
//*****
public void add(int pos, Object item) {
    // check for bad pos and for full array
    if (pos < 0 || pos > numItems) {
        throw new IndexOutOfBoundsException();
    }
    if (items.length == numItems) {
        expandArray();
    }

    // move items over and insert new item
    for (int k = numItems; k > pos; k--) {
        items[k] = items[k-1];
    }
    items[pos] = item;
    numItems++;
}

```

## TEST YOURSELF #3

**Question 1:** Write the `remove` and `get` methods.

[solution](#)

### Implementing the constructor

Now let's think about the constructor; it should initialize the fields so that the `List` starts out empty. Clearly, `numItems` should be set to zero. How about the `items` field? It could be set to `null`, but that would mean another special case in the `add` methods. A better idea would be to initialize `items` to refer to an array with some initial size, perhaps specified using a class constant (i.e., a static `final` field), so that the initial size could be easily changed.

Below is the code for the constructor (including the declaration of the class constant for the initial size). This code uses 10 as the initial size. In practice, the appropriate initial size will probably depend on the context in which the `ArrayList` class is used. The advantage of a larger initial size is that more `add` operations can be performed before it is necessary to expand the array (which requires copying all items). The disadvantage is that if the initial array is never filled, then memory is wasted. The requirements for memory usage and runtime performance of the application that uses the `SimpleArrayList` class, as well as the expected sizes of the `Lists` that it uses, should be used to determine the appropriate initial size.

```
private static final int INITSIZE = 10;

//*****
// SimpleArrayList constructor
//
// initialize the List to be empty
//*****
public SimpleArrayList() {
    numItems = 0;
    items = new Object[INITSIZE];
}
```

### The Java API and Lists

The Java API contains many interfaces, including interfaces that are very similar to many of the ADTs we will be talking about. For example, the `Java.util` package provides a `List` interface (with many more methods than the ones given in our `ListADT` interface), as well as a number of classes that implement that interface, including the `ArrayList` class and the `LinkedList` class. The `SimpleArrayList` implementation we just developed is meant to give you insight into a way that Java's `ArrayList` class might be implemented.

### Iterators

#### What are iterators?

When a client uses an abstract data type that represents a collection of items (as the `List` interface does), they often need a way to **iterate** through the collection, i.e., to access each of the items in turn. Our `get` method can be used to support this operation. Given a `List L`, we can iterate through the items in the `List` as follows:

```
for (int k = 0; k < L.size(); k++) {
    Object ob = L.get(k);
    ... do something to ob here ...
}
```

However, a more standard way to iterate through a collection of items is to use an `Iterator`, which is an interface defined in `java.util`. Every Java class that implements the `Iterable` interface (which includes classes that implement the subinterface `Collection`) provides an `iterator` method that returns an `Iterator` for that collection.

The way to think about an `Iterator` is that it is a finger that points to each item in the collection in turn. When an `Iterator` is first created, it is pointing to the first item; a `next` method is provided that lets you get the item pointed to (also advancing the pointer to point to the next item) and a `hasNext` method lets you ask whether you've run out of items. For example, assume that we have added an `iterator` method (that returns an `Iterator`) to our `ArrayList` class and that we have the following list of words:

apple      pear      banana      strawberry

If we create an `Iterator` for the `List`, we can picture it as follows, pointing to the first item in the `List`:

apple      pear      banana      strawberry  
^  
|

Now if we call `next` we get back the word "apple" and the picture changes to:

apple      pear      banana      strawberry  
          ^  
          |

After two more calls to `next` (returning "pear" and "banana") we'll have:

```
apple    pear    banana    strawberry
                                     ^
                                     |
```

A call to `hasNext` now returns `true` (because there's still one more item we haven't accessed yet). A call to `next` returns "strawberry" and our picture looks like this:

```
apple    pear    banana    strawberry
                                     ^
                                     |
```

The iterator has fallen off the end of the List. Now a call to `hasNext` returns `false` and if we call `next`, we'll get a `NoSuchElementException`.

The Java interface `List<E>` has a method `iterator()` that returns an object of type `Iterator<E>`. You can use it to iterate through the elements of a list with a `while` loop like this:

```
List<String> words = new ArrayList<String>();
... // add some words to the list
Iterator<String> itr = words.iterator();
while (itr.hasNext()) {
    String nextWord = itr.next();
    ... // do something with nextWord
}
```

or with a `for` loop like this (note that the **increment** part of the loop header is empty, since the call to the iterator's `next` method inside the loop causes it to advance to the next item):

```
List<String> words = new ArrayList<String>();
... // add some words to the list
for (Iterator<String> itr = words.iterator(); itr.hasNext(); ) {
    String nextWord = itr.next();
    ... // do something with nextWord
}
```

Java also provides **extended for-loops** (sometimes also referred to as **for-each loops**), which are another way to iterate through lists. The code looks like this:

```
List<String> words = new ArrayList<String>();
... // add some words to the list
for (String nextWord : words) {
    ... // do something with nextWord
}
```

In this context, the colon ':' is pronounced "in", so the `for` statement would be read out loud as "for each `String nextWord` in `words`, do ...".

## Implementing iterators

The easiest way to implement an iterator for the `SimpleArrayList` class is to define a new class (e.g., called `SimpleArrayListIterator`) with two fields:

1. the List that's being iterated over, and
2. the index of the current item (the item the "finger" is currently pointing to).

We also define a new iterator method for the `SimpleArrayList` class; that method calls the `SimpleArrayListIterator` class's constructor, passing the `SimpleArrayList` itself:

```
/**
 * iterator
 */
// return an iterator for this List
/**
 * public Iterator<Object> iterator() {
 *     return new SimpleArrayListIterator(this);
 * }
 */
```

The `Iterator` interface is defined in `java.util`, so you need to include:

```
import java.util.*; // or import java.util.Iterator;
```

at the beginning of `SimpleArrayList.java` and `SimpleArrayListIterator.java` as well as in any other file that contains code that uses `Iterators`.

The `SimpleArrayListIterator` class is defined to implement Java's `Iterator` interface and therefore must implement three methods: `hasNext` and `next` (both discussed above) plus an optional `remove` method. If you choose not to implement the `remove` method, you still have to define it, but it should simply throw an `UnsupportedOperationException`. If you choose to implement the `remove` method, then it should remove from the `SimpleArrayList` the last (i.e., most recent) item returned by the iterator's `next` method or it should throw an `IllegalStateException` if the `next` method hasn't yet been called.

Here is code that defines the `SimpleArrayListIterator` class (note that we have chosen not to implement the `remove` operation):



```

import java.util.Iterator;
import java.util.NoSuchElementException;

public class SimpleArrayListIterator implements Iterator<Object> {

    // *** fields ***
    private SimpleArrayList list; // the list we're iterating over
    private int curPos;           // the position of the next item

    // *** constructor ***
    public SimpleArrayListIterator(SimpleArrayList list) {
        this.list = list;
        curPos = 0;
    }

    // *** methods ***

    public boolean hasNext() {
        return curPos < list.size();
    }

    public Object next() {
        if (!hasNext()) {
            throw new NoSuchElementException();
        }
        Object result = list.get(curPos);
        curPos++;
        return result;
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}

```

## Testing

One of the most important (but unfortunately most difficult) parts of programming is testing. A program that doesn't work as specified is **not** a good program and in some contexts can even be dangerous and/or cause significant financial loss.

There are two general approaches to testing: **black-box** testing and **white-box** testing. For black-box testing, the testers know nothing about how the code is actually implemented (or if you're doing the testing yourself, you pretend that you know nothing about it). Tests are written based on what the code is supposed to do and the tester thinks about issues like:

- testing all of the operations in the interface
- testing a wide range of input values, especially including "boundary" cases
- testing both "legal" (expected) inputs as well as unexpected ones

So if you were to do black-box testing of the `ArrayList` class, you should be sure to:

- call all of the `ArrayList` methods
- try each operation on an `ArrayList` with no items, with exactly one item, and with many items
- include calls that should cause exceptions (e.g., adding an item at position -1, removing an item at a position 1 past the end of the List, and adding an item at a position more than 1 past the end of the List).

For white-box testing, the testers have access to the code and the usual goal is to make sure that every line of code executes at least once. So for example, if you were to do white-box testing of the `ArrayList` class, you should be sure to write a test that causes the array to become full so that the `expandArray` method is tested (something a black-box tester may not test, since they don't even know that `ArrayLists` are implemented using arrays).

You will be a much better programmer if you learn to be a good tester. Therefore, testing will be an important factor in the grades you receive for the programming projects in this class.

# Java Exceptions

---

## Contents

- [Error Handling](#)
  - [Exceptions](#)
    - [How to catch exceptions](#)
    - [Checked and unchecked exceptions](#)
    - [Exception hierarchy](#)
    - [Choices when calling a method that may throw an exception](#)
    - [Test Yourself #1](#)
    - [Defining an exception](#)
    - [Throwing an exception](#)
    - [Test Yourself #2](#)
  - [Summary](#)
- 

## Error Handling

Runtime errors can be divided into low-level errors that involve violating constraints, such as:

- dereference of a null pointer
- out-of-bounds array access
- divide by zero
- attempt to open a non-existent file for reading
- bad cast (e.g., casting an `Object` that is actually a `Boolean` to `Integer`)

and higher-level, logical errors, such as violations of a method's preconditions:

- a call to an `Iterator`'s "next" method when there are no more items
- a call to a `Stack`'s "pop" method for an empty stack
- a call to "factorial" with a negative number

Logical errors can lead to low-level errors if they are not detected. Often, it is better to detect them (to provide better feedback).

Errors can arise due to:

- User error (for example, providing a bad file name or a poorly formatted input file): A good program should be written to anticipate these situations and should deal with them. For example, given a bad file name, an interactive program could print an error message and prompt for a new name.
- Programmer error (i.e., a buggy program): These errors should be detected as early as possible to provide good feedback. For some programs it may be desirable to do some recovery after detecting this kind of error, for example, writing out current data.

Note that recovery is often not possible at the point of the error (because the error may occur inside some utility method that doesn't know anything about the overall program or what error recovery should involve). Therefore, it is desirable to "pass the error up" to a level that can deal with it.

There are several possible ways to handle errors:

- Write an error message and quit. This doesn't provide any recovery.
- Return a special value to indicate that an error occurred. This is the usual approach for C functions (which often return 0 or -1 to signal an error).

However:

- It doesn't work if the method also returns a value on normal completion and all values are possible (i.e., there is no special value that can be used to signal an error).
- It requires that the calling code check for an error. This can reduce the efficiency of the code and is often omitted by programmers out of laziness or carelessness.
- It can sometimes make the code more clumsy. For example, if method `g` might return an error code, one would have to write something like:

```
ret = g(x);  
if (ret == ERROR_CODE) { ... }  
else f(ret);
```

instead of just:

```
f(g(x));
```

- Use a reference parameter or a global variable to hold an error code. This solves the first problem of the previous approach, but not the second or third ones.
- Use exceptions. This seems to be the method of choice for modern programming languages.

## Exceptions

### Idea:

When an error is detected, an exception is **thrown**. That is, the code that caused the error stops executing immediately and control is transferred to the **catch clause** for that type of exception of the first enclosing **try block** that has such a clause. The **try** block might be in the current method (the one that caused the error) or it might be in some method that called the current method (i.e., if the current method is not prepared to handle the exception, it is "passed up" the call chain). If no currently active method is prepared to catch the exception, the exception ends up passed up the call chain from `main` to the Java virtual machine. At that point, an error message is printed and the program stops.

Exceptions can be built-in (actually, defined in one of Java's standard libraries) or user-defined. Here are some examples of built-in exceptions with links to their documentation:

- [ArithmeticException](#) (e.g., divide by zero)
- [ClassCastException](#) (e.g., attempt to cast a `String` object to `Integer`)
- [IndexOutOfBoundsException](#)
- [NullPointerException](#)
- [FileNotFoundException](#) (e.g., attempt to open a non-existent file for reading)

## How to Catch Exceptions

Exceptions are caught using **try blocks**:

```
try {
    // statements that might cause exceptions
    // possibly including method calls
} catch ( ExceptionType1 id1 ) {
    // statements to handle this type of exception
} catch ( ExceptionType2 id2 ) {
    // statements to handle this type of exception
    .
    .
    .
} finally {
    // statements to execute every time this try block executes
}
```

### Notes:

1. Each **catch** clause specifies the type of one exception and provides a name for it (similar to the way a method header specifies the type and name of a parameter). Java exceptions are objects, so the statements in a **catch** clause can refer to the thrown exception object using the specified name.
2. The **finally** clause is optional; a **finally** clause is usually included if it is necessary to do some clean-up (e.g., closing opened files).
3. In general, there can be one or more **catch** clauses. If there *is* a **finally** clause, there can be zero **catch** clauses.

## Example

Here is a program that tries to open a file for reading. The name of the file is given by the first command-line argument .

```
public class Test {
    public static void main(String[] args) {
        Scanner fileIn;
        File inputFile;

        try {
            inputFile = new File(args[0]);
            fileIn = new Scanner(inputFile); // may throw FileNotFoundException
        } catch (FileNotFoundException ex) {
            System.out.println("file " + args[0] + " not found");
        }
    }
}
```

### Notes:

1. The program really should make sure there is a command-line argument before attempting to use `args[0]`.
2. Also, it probably makes more sense to put the **try** block in a loop, so that if the file is not found the user can be asked to enter a new file name and a new attempt to open the file can be made.
3. As is, if the user runs the program with a bad file name `foo`, the message "file foo not found" will be printed and the program will halt.
4. If there were no **try** block and the program were run with a bad file name `foo`, a more complicated message, something like this:

```

java.io.FileNotFoundException: foo
    at java.io.FileInputStream ...
    at ...
    at Test.main ...

```

would be printed. (Actually, if there were no `try/catch` for the `FileNotFoundException`, the program wouldn't compile because it fails to list that exception as one that might be thrown. We'll come back to that issue later.)

## Checked and unchecked exceptions

Every exception is either a **checked** exception or an **unchecked** exception. If a method includes code that could cause a *checked* exception to be thrown, then:

- the exception must be declared in the method header using a **throws clause**, or
- the code that might cause the exception to be thrown must be inside a `try` block with a `catch` clause for that exception.

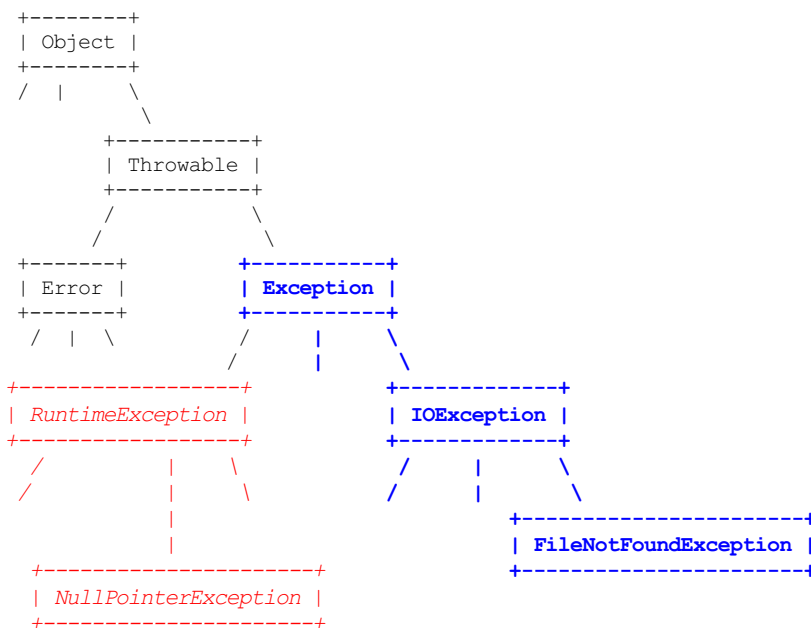
So, in general, you must always include some code that acknowledges the possibility of a checked exception being thrown. If you don't, you will get an error when you try to compile your code. (The compiler does not check to see if you have included code that acknowledges the possibility of an *unchecked* exception being thrown.)

## Exception hierarchy

How can you determine if an exception is checked or unchecked? The answer lies in the exception hierarchy:

- If an exception is a subclass of `RuntimeException`, then it is *unchecked*.
- If an exception is a subclass of `Exception` but not a subclass of `RuntimeException`, then it is *checked*.

A portion of the Java class hierarchy is shown below. Checked exceptions are in **bold**, unchecked exceptions are in *italics*.



### Note that:

- Most of the built-in exceptions (e.g., `NullPointerException`, `IndexOutOfBoundsException`) are *unchecked*.
- I/O exceptions (e.g., `FileNotFoundException`) are *checked*.
- User-defined exceptions should usually be checked, so they should be subclasses of `Exception`.

## Choices when calling a method that may throw an exception

If you are writing code that calls a method that might throw an exception, your code can do one of three things:

1. Catch and handle the exception.
2. Catch the exception, then re-throw it or throw another exception.
3. Ignore the exception (let it "pass up" the call chain).

As mentioned above, if your code might cause a *checked* exception to be thrown; i.e.,

- your code throws a checked exception, or
- your code ignores a checked exception that might be thrown by a called method

then your method must include a `throws` clause listing all such exceptions. For example:

```

public static void main(String[] args) throws FileNotFoundException, EOFException

```

```
{
    // an uncaught FileNotFoundException or EOFException may be thrown here
}
```

Only uncaught **checked** exceptions need to be listed in a method's `throws` clause. Unchecked exceptions can be caught in a `try` block, but if not, they need not be listed in the method's `throws` clause.

## TEST YOURSELF #1

Consider the following program (assume that comments are replaced with actual code that works as specified):

```
public class TestExceptions {

    static void e() {
        // might cause any of the following unchecked exceptions to be thrown:
        // Ex1, Ex2, Ex3, Ex4
    }

    static void d() {
        try {
            e();
        } catch (Ex1 ex) {
            System.out.println("d caught Ex1");
        }
    }

    static void c() {
        try {
            d();
        } catch (Ex2 ex) {
            System.out.println("c caught Ex2");
            // now cause exception Ex1 to be thrown
        }
    }

    static void b() {
        try {
            c();
        } catch (Ex1 ex) {
            System.out.println("b caught Ex1");
        } catch (Ex3 ex) {
            System.out.println("b caught Ex3");
        }
    }

    static void a() {
        try {
            b();
        } catch (Ex4 ex) {
            System.out.println("a caught Ex4");
            // now cause exception Ex1 to be thrown
        } catch (Ex1 ex) {
            System.out.println("a caught Ex1");
        }
    }

    public static void main(String[] args) {
        a();
    }
}
```

Assume that this program is run four times. The first time, method `e` throws exception `Ex1`, the second time, it throws exception `Ex2`, etc. For each of the four runs, say what is printed and whether any uncaught exception is thrown.

[solution](#)

## Defining an exception

Java exceptions are **objects**. We can define a new kind of exception by defining an instantiable class. This new class **must** be a subclass of `Throwable`; as discussed above, they are usually subclasses of `Exception` (so that they are checked). The simplest way to define a new exception is shown in this example:

```
public class EmptyStackException extends Exception { }
```

There is no need to provide any methods or fields; the class can have an empty body as shown above.

Many (if not most) of the exceptions defined in the Java API provide a constructor that takes a string as an argument (in addition to a default constructor). The purpose of the argument is to allow a detailed error message to be given when the exception object is created. You can access this message by calling the `getMessage()` method on the exception object. Providing this option in the exceptions you write is only slightly more complicated:

```

public class EmptyStackException extends Exception {
    public EmptyStackException() {
        super();
    }

    public EmptyStackException(String message) {
        super(message);
    }
}

```

## Throwing an exception

At the point where an error is detected, an exception is thrown using a **throw** statement:

```

public class Stack {
    ...
    public Object pop() throws EmptyStackException {
        if (empty())
            throw new EmptyStackException();
        ...
    }
}

```

Note that because exceptions are objects, so you cannot simply throw "EmptyStackException" -- you must use "new" to create an exception object. Also, since the pop method might throw the (checked) exception EmptyStackException, that exception must be listed in pop's throws clause.

---

## TEST YOURSELF #2

**Question 1:** Assume that method `f` might throw *checked* exceptions `Ex1`, `Ex2`, or `Ex3`. Complete method `g`, outlined below, so that:

- If the call to `f` causes `Ex1` to be thrown, `g` will catch that exception and print "Ex1 caught".
- If the call to `f` causes `Ex2` to be thrown, `g` will catch that exception, print "Ex2 caught" and then will *throw* an `Ex1` exception.

```

static void g() throws ... {
    try {
        f();
    } catch ( ... ) {
        ...
    } ...
}

```

**Question 2:** Consider the following method.

```

static void f(int k, int[] A, String S) {
    int j = 1 / k;
    int len = A.length + 1;
    char c;

    try {
        c = S.charAt(0);
        if (k == 10) j = A[3];
    } catch (ArrayIndexOutOfBoundsException ex) {
        System.out.println("array error");
        throw new InternalError();
    } catch (ArithmeticException ex) {
        System.out.println("arithmetic error");
    } catch (NullPointerException ex) {
        System.out.println("null ptr");
    } finally {
        System.out.println("in finally clause");
    }
    System.out.println("after try block");
}

```

**Part A:** Assume that variable `x` is an array of `int` that has been initialized to be of length 3. For each of the following calls to method `f`, say what (if anything) is printed by `f` and what, if any, uncaught exceptions are thrown by `f`.

- `f(0, x, "hi")`
- `f(10, x, "")`
- `f(10, x, "bye")`
- `f(10, x, null)`

**Part B:** Why doesn't `f` need to have a `throws` clause that lists the uncaught exceptions that it might throw?

[solution](#)

---

## Summary

- Code that *detects* errors often does not know how to handle them. Therefore, we need a way to "pass errors up". The best approach is to use **exceptions**.
- Java provides both built-in and user-defined exceptions.
- Exceptions are caught using a **try block**:

```
try {
    // statements (including method calls) that might cause exceptions
} catch ( ExceptionType1 id1 ) {
    // code to handle this first type of exception
} catch ( ExceptionType2 id2 ) {
    // code to handle this second type of exception
    .
    .
    .
} finally {
    // code that will execute whenever this try block executes
}
```

- Exceptions are thrown using a **throw statement**.
- If an exception is thrown in code that is not inside a `try` block, or is in a `try` block with no `catch` clause for the thrown exception, the exception is "passed up" the call stack.
- Some exceptions are checked and some are unchecked. If a method might throw one or more checked exceptions, they must be listed in the method's **throws** clause.
- Exceptions are objects; they are defined as *classes* (the class name is the name of the exception) that extend the `Exception` class.

# Implementing Lists Using Linked-Lists

## Contents

- [Introduction](#)
- [Java Types](#)
  - [Test Yourself #1](#)
- [Intro to Linked Lists](#)
  - [Test Yourself #2](#)
- [Linked List Operations](#)
  - [Adding a node](#)
    - [Test Yourself #3](#)
  - [Removing a node](#)
  - [Using a header node](#)
- [The LinkedList Class](#)
  - [add \(to end of list\)](#)
    - [Test Yourself #4](#)
  - [add \(at a given position\)](#)
    - [Test Yourself #5](#)
  - [The LinkedList constructor](#)
    - [Test Yourself #6](#)
- [Comparison: Lists via Arrays versus via Linked Lists](#)
  - [Test Yourself #7](#)
- [Linked List Variations](#)
  - [Doubly linked lists](#)
  - [Circular linked lists](#)
    - [Test Yourself #8](#)
    - [Test Yourself #9](#)
  - [Comparisons](#)

## Introduction



The previous set of notes discussed how to implement the ListADT interface using an array to store the items in the list. Here we discuss how to implement the ListADT interface using a **linked list** to store the items. However, before talking about linked lists, we will review the difference between primitive and non-primitive types in Java.

## Java Types


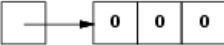
Java has two "categories" of types:

1. **primitive** types: short, int, long, float, double, boolean, char, and byte
2. **reference** types: arrays and classes

When you declare a variable with a primitive type, you get enough space to hold a value of that type. Here's some code involving a primitive type and the corresponding conceptual picture:

CODE	CONCEPTUAL PICTURE
<code>int k;</code>	<code>k:</code> 
<code>k = 5;</code>	<code>k:</code> 

When you declare a variable with a reference type, you get space for a **reference** (or **pointer**) to that type, not for the type itself. You must use "new" to get space for the type itself. This is illustrated below.

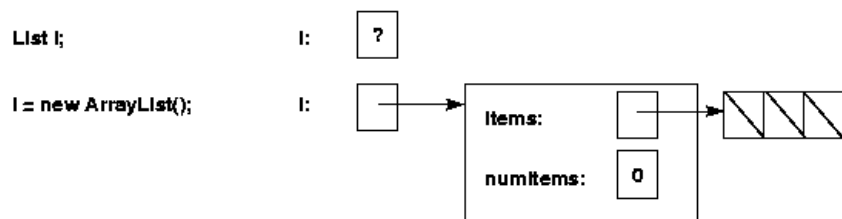
CODE	CONCEPTUAL PICTURE
<code>int[] a;</code>	<code>a:</code> 
<code>a = new int[3];</code>	<code>a:</code> 

Remember that class objects are also reference types. For example, if you declare a variable of type `List`, you only get space for a pointer to a list; no actual list exists until you use "new". This is illustrated below, assuming the array implementation of lists and assuming that the `ArrauList` constructor initializes the `items` array to be of size 3. Note that because it is an array of `Objects`, each array element is (automatically) initialized to null (shown using a diagonal line in the



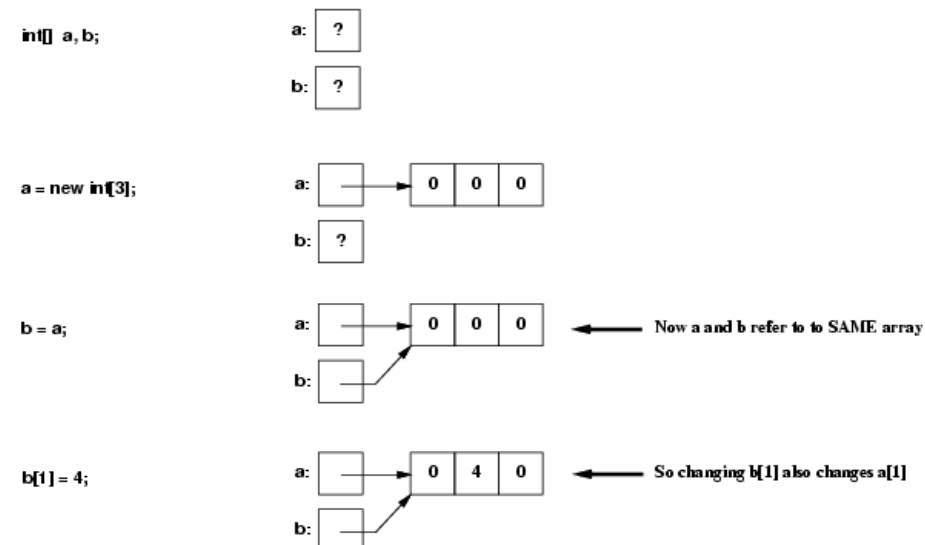
picture).

CODE CONCEPTUAL PICTURE



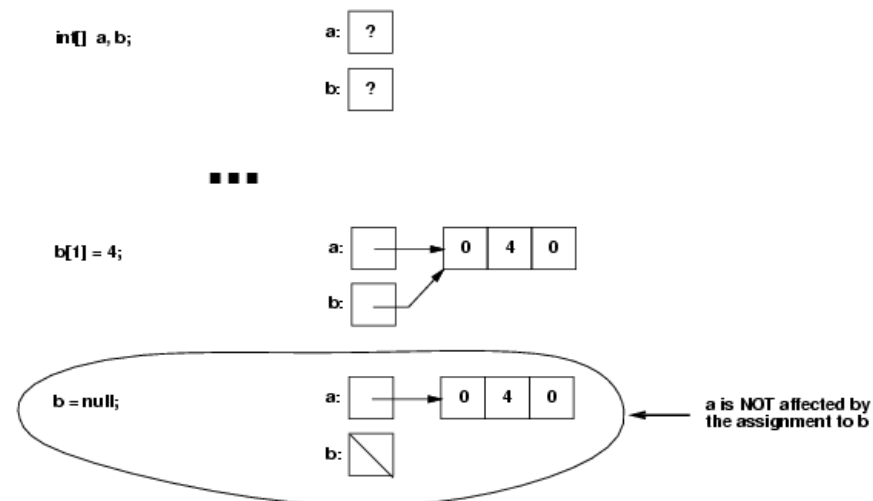
An important consequence of the fact that non-primitive types are really pointers is that assigning from one variable to another can cause **aliasing** (two different names refer to the same object). For example:

CODE CONCEPTUAL PICTURE



Note that in this example, the assignment to `b[1]` changed not only that value, but also the value in `a[1]` (because `a` and `b` were pointing to the *same* array)! However, an assignment to `b` itself (not to an element of the array pointed to by `b`) has no effect on `a`:

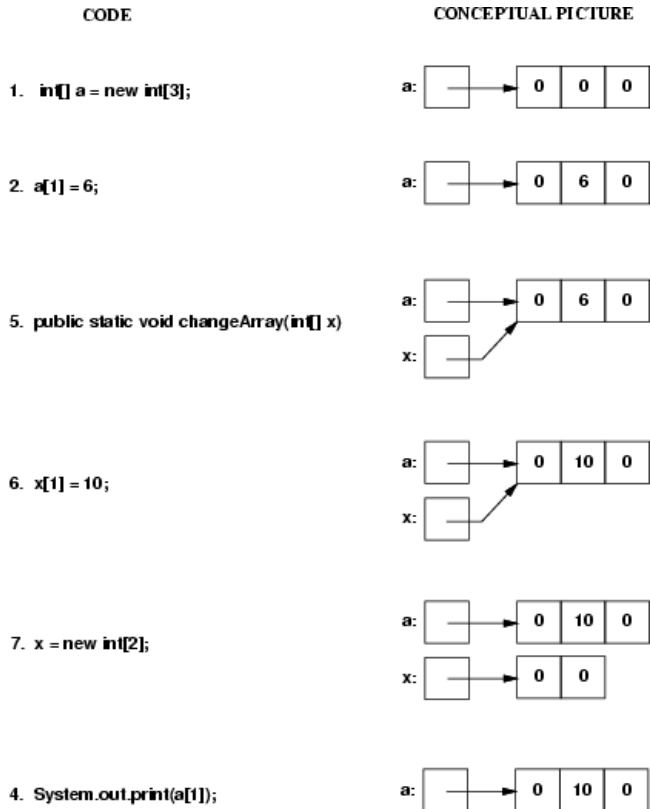
CODE CONCEPTUAL PICTURE



A similar situation arises when a non-primitive value is passed as an argument to a method. For example, consider the following 4 statements and the definition of method `changeArray`:

```
1. int[] a = new int[3];  
2. a[1] = 6;  
3. changeArray(a);  
4. System.out.print(a[1]);  
  
5. public static void changeArray(int[] x) {  
6.     x[1] = 10;  
7.     x = new int[2];  
8. }
```

The picture below illustrates what happens when this code executes.



Note that the method call causes A and X to be aliases (they both point to the same array). Therefore, the assignment `X[1] = 10` changes both `X[1]` and `A[1]`. However, the assignment `X = new int[2]` only changes X, not A, so when the call to `changeArray` finishes, the value of `A[1]` is still 10, not 0.

## TEST YOURSELF #1

For each line of the code shown below, draw the corresponding conceptual picture.

```
int [] x = new int[2];
x[0] = 1;
int y = new int[3];
y[0] = 2;
y = x;
y[0] = 3;
```

[solution](#)

## Intro to Linked Lists

Here's a conceptual picture of a linked list containing N items, pointed to by a variable named l:



Note that a linked list consists of one or more **nodes**. Each node contains some data (in this example, item 1, item 2, etc.) and a pointer. For each node other than the last one, the pointer points to the next node in the list. For the last node, the pointer is null (indicated in the example using a diagonal line). To implement linked lists in Java, we will define a `ListNode` class, to be used to represent the individual nodes of the list.

```
public class ListNode<E> {
    /** fields */
    private E data;
    private ListNode<E> next;

    /** constructors */
    // 2 constructors
    public ListNode(E d) {
        this(d, null);
    }

    public ListNode(E d, ListNode n) {
        data = d;
        next = n;
    }
}
```

```

/** methods */
// access to fields
public E getData() {
    return data;
}

public ListNode<E> getNext() {
    return next;
}

// modify fields
public void setData(E d) {
    data = d;
}


public void setNext(ListNode<E> n) {
    next = n;
}
}

```

Note that the `next` field of a `ListNode<E>` is itself of type `ListNode<E>`. That works because in Java, every non-primitive type is really a **pointer**; so a `ListNode<E>` object is really a pointer that is either null or points to a piece of storage (allocated at runtime) that consists of two fields named `data` and `next`.

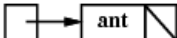
To understand this better, consider writing code to create a linked list of `Strings` with two nodes, containing "ant" and "bat", respectively, pointed to by a variable named `l`. First we need to declare variable `l`; here's the declaration together with a picture showing what we have so far:

CODE                      CONCEPTUAL PICTURE

`ListNode<String> l;`                      `l`: 

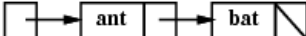
To make `l` point to the first node of the list, we need to use `new` to allocate space for that node. We want its `data` field to contain "ant" and (for now) we don't care about its `next` field, so we'll use the 1-argument `ListNode` constructor (which sets the `next` field to null):

CODE                      CONCEPTUAL PICTURE

`l = new ListNode<String>("ant");` `l`: 

To add the second node to the end of the list we need to create the new node (with "bat" in its `data` field and null in its `next` field) and we need to set the `next` field of the first node to point to the new one:

CODE                      CONCEPTUAL PICTURE

`l.setNext(new ListNode<String>("bat"));` `l`: 

## TEST YOURSELF #2

Assume that the list shown above (with nodes "ant" and "bat") has been created.

**Question 1:** Write code to change the contents of the second node's `data` field from "bat" to "cat".

**Question 2:** Write code to insert a new node with "rat" in its `data` field *between* the two existing nodes.

[solution](#)

## Linked List Operations

Before thinking about how to implement the `ListADT` interface using linked lists, let's consider some basic operations on linked lists:

- Adding a node after a given node in the list.
- Removing a given node from the list.

### Adding a node

Assume that we are given:

1. `n`, (a pointer to) a node in a list (i.e., `n` is a `ListNode`), and
2. `newdat`, the data to be stored in a new node

and that the goal is to add a new node containing `newdat` immediately after `n`. To do this we must perform the following steps:

Step 1: create the new node using the given data

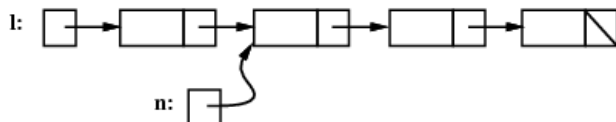
Step 2: "link it in":

- make the new node's `next` field point to whatever `n`'s `next` field was pointing to
- make `n`'s `next` field point to the new node.

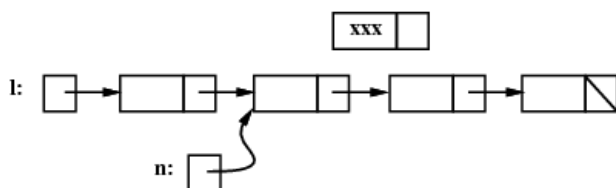
Here's the conceptual picture:

ADDING A NODE WITH `newdat = xxx`

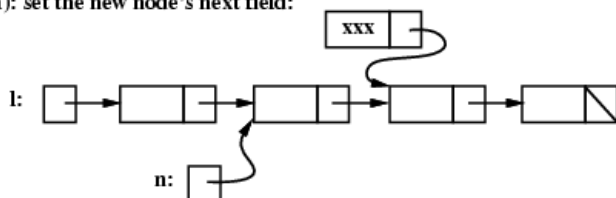
Here's the original list, with `n` pointing to one of its nodes:



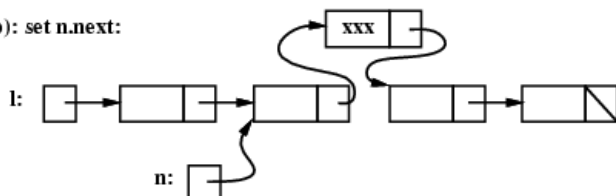
Step 1: create a new node containing `xxx`:



Step 2(a): set the new node's `next` field:



Step 2(b): set `n.next`:



And here's the code:

```
Listnode<String> tmp = new Listnode<String>(newdat); // Step 1
tmp.setNext( n.getNext() ); // Step 2(a)
n.setNext( tmp ); // Step 2(b)
```

Note that it is vital to first copy the value of `n`'s `next` field into `tmp`'s `next` field (step 2(a)) before setting `n`'s `next` field to point to the new node (step 2(b)). If we set `n`'s `next` field first, we would lose our only pointer to the rest of the list after node `n`!

Also note that, in order to follow the steps shown in the picture above, we needed to use variable `tmp` to create the new node (in the picture, step 1 shows the new node just "floating" there, but that isn't possible -- we need to have some variable point to it so that we can set its `next` field and so that we can set `n`'s `next` field to point to it). However, we could in fact accomplish steps 1 and 2 with a single statement that creates the new node, fills in its data and `next` fields, and sets `n`'s `next` field to point to the new node! Here is that amazing statement:

```
n.setNext( new Listnode<String>(newdat, n.getNext()) ); // steps 1, 2(a), and 2(b)
```

### TEST YOURSELF #3

Draw pictures like the ones given above, to illustrate what happens when node `n` is the *last* node in the list. Does the statement

```
n.setNext( new Listnode<String>(newdat, n.getNext()) );
```

still work correctly?

[solution](#)

Now consider the worst-case running time for this add operation. Whether we use the single statement or the list of three statements, we are really doing the same thing:

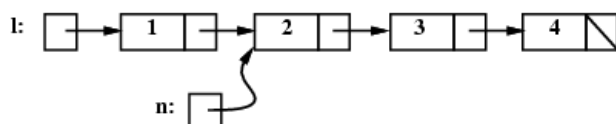
1. Using `new` to allocate space for a new node (start step 1).
2. Initializing the new node's `data` and `next` fields (finish step 1 + step 2(a)).
3. Changing the value of `n`'s `next` field (step 2(b)).

We will assume that storage allocation via `new` takes constant time. Setting the values of the three fields also takes constant time, so the whole operation is a constant-time ( $O(1)$ ) operation. In particular, the time required to add a new node immediately after a given node is independent of the number of nodes already in the list.

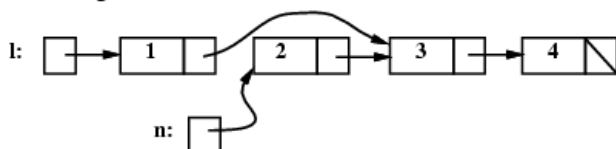
## Removing a node

To remove a given node `n` from a linked list, we need to change the `next` field of the node that comes immediately *before* `n` in the list to point to whatever `n`'s `next` field was pointing to. Here's the conceptual picture:

Before removing node `n`:



After removing node `n`:



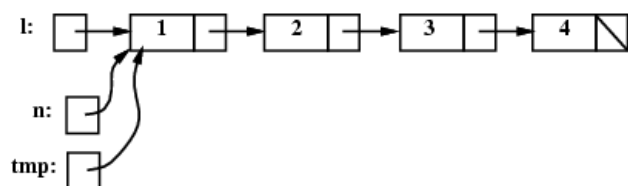
Note that the fact that `n`'s `next` field is still pointing to a node in the list doesn't matter -- `n` has been removed from the list, because it cannot be reached from `l`. It should be clear that in order to implement the remove operation, we first need to have a pointer to the node *before* node `n` (because that node's `next` field has to be changed). The only way to get to that node is to start at the beginning of the list. We want to keep moving along the list as long as the current node's `next` field is *not* pointing to node `n`. Here's the appropriate code:

```

ListNode<String> tmp = l;
while (tmp.getNext() != n) { // find the node before n
    tmp = tmp.getNext();
}
  
```

Note that this kind of code (moving along a list until some condition holds) is very common. For example, similar code would be used to implement a lookup operation on a linked list (an operation that determines whether there is a node in the list that contains a given piece of data).

Note also that there is one case when the code given above will not work. When `n` is the very first node in the list, the picture is like this:



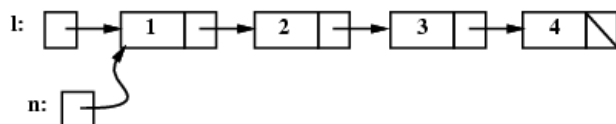
In this case, the test `(tmp.getNext() != n)` will always be false and eventually we will "fall off the end" of the list (i.e., `tmp` will become null and we will get a runtime error when we try to dereference a null pointer). We will take care of that case in a minute; first, assuming that `n` is not the first node in the list, here's the code that removes `n` from the list:

```

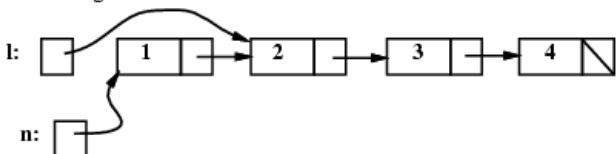
ListNode<String> tmp = l;
while (tmp.getNext() != n) { // find the node before n
    tmp = tmp.getNext();
}
tmp.setNext( n.getNext() ); // remove n from the linked list
  
```

How can we test whether `n` is the first node in the list and what should we do in that case? If `n` is the first node, then `l` will be pointing to it, so we can test whether `l == n`. The following before and after pictures illustrate removing node `n` when it is the first node in the list:

Before removing node n:



After removing node n:



Here's the complete code for removing node *n* from a linked list, including the special case when *n* is the first node in the list:

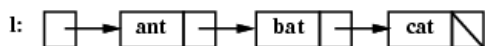
```
if (l == n) {
    // special case: n is the first node in the list
    l = n.getNext();
} else {
    // general case: find the node before n, then "unlink" n
    Listnode<String> tmp = l;
    while (tmp.getNext() != n) { // find the node before n
        tmp = tmp.getNext();
    }
    tmp.setNext(n.getNext());
}
```

What is the worst-case running time for this remove operation? If node *n* is the first node in the list, then we simply change one field (*l*'s *next* field). However, in the general case, we must traverse the list to find the node before *n*, and in the worst case (when *n* is the *last* node in the list), this requires time proportional to the number of nodes in the list. Once the node before *n* is found, the remove operation involves just one assignment (to the *next* field of that node), which takes constant time. So the worst-case time running time for this operation on a list with *N* nodes is  $O(N)$ .

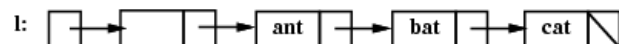
## Using a header node

There is an alternative to writing special-case code to handle removing the first node in a list. That alternative is to use a **header** node: a dummy node at the front of the list that is there only to reduce the need for special-case code in the linked-list operations. For example, the picture below shows how the list "ant", "bat", "cat", would be represented using a linked list without and with a header node:

Without a header node:



With a header node:



Note that if your linked lists do include a header node, there is no need for the special case code given above for the remove operation; node *n* can *never* be the first node in the list, so there is no need to check for that case. Similarly, having a header node can simplify the code that adds a node before a given node *n*.

Note that if you do decide to use a header node, you must remember to initialize an empty list to contain one (dummy) node, you must remember not to include the header node in the count of "real" nodes in the list (e.g., if you implement a *size* operation), and you must remember to ignore the header node in operations like *contains*.

## The LinkedList Class

Now let's consider how to implement our `ListADT` interface using linked list instead of arrays; i.e., how to implement a scaled-down `LinkedList` class. Remember, we only want to change the **implementation** (the "internal" part of the `List` abstract data type), not the **interface** (the "external" part of the abstract data type). That means that the signatures of the public methods for the `LinkedList` class will be the same as the ones for the `SimpleArrayList` class (and the `ListADT` interface) and the descriptions of what those methods do will also be the same. The only things that will change are how the list is represented and how the methods are implemented.

Look back at the [definition](#) of the `SimpleArrayList` class in the second set of notes and think about which fields and/or methods need to be changed before reading any further.

Clearly, the type of the `items` field needs to change, since the items will no longer be stored in an array; instead, the items will be stored in a linked list. Since having a header node simplifies the `add` and `remove` operations, we will assume that the linked list has a header node. Here's new declaration for the `items` field:

```
private Listnode<E> items; // pointer to the header node of the list of items
```

Given this declaration for the `items` field, let's think again about the three List methods that were discussed assuming the array implementation:

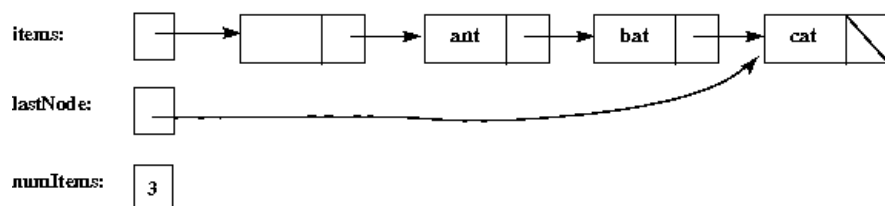
1. The version of `add` that adds to the end of the list.
2. The version of `add` that adds to a given position in the list.
3. The constructor.

### add (to end of list)

Recall that the first version of method `add` adds a given value to the end of the list. We have already discussed how to add a new node to a linked list following a given node. The only question is how best to handle adding a new node at the end of the list. A straightforward approach would be to traverse the list, looking for the last node (i.e., use a variable `tmp` as was done above in the code that looked for the node before node `n`). Once the last node is found, the new node can be inserted immediately after it.

The disadvantage of this approach is that it requires  $O(N)$  time to add a node to the end of a list with  $N$  items. An alternative is to add a `lastNode` field (often called a **tail pointer**) to the `LinkedList` class and to implement the methods that modify the linked list so that `lastNode` always points to the last node in the linked list (which will be the header node if the list is empty). There is more opportunity for error (since several methods will need to ensure that the `lastNode` field is kept up to date), but the use of the `lastNode` field will mean that the worst-case running time for this version of `add` is always  $O(1)$  and that will be important for applications that frequently add to the end of a list (which is often a common operation).

Here's a picture of the "ant, bat, cat" list, when the implementation includes a `lastNode` pointer:



---

### TEST YOURSELF #4

Write the "add at the end" method (assuming that the `LinkedList` class includes both a header node and a `lastNode` field).

[solution](#)

---

### add (at a given position)

As discussed above for the "add to the end" method, we already know how to add a node to a linked list after a given node. So to add a node at position `pos`, we just need to find the previous node in the list. Since we're assuming that our `LinkedList` class is implemented with a header node, there will always be such a node (i.e., we don't need any special-case code when we're asked to add a node at the beginning of the list).

---

### TEST YOURSELF #5

Write the "add at a given position" method (assuming that the `LinkedList` class includes both a header node and a `lastNode` field).

[solution](#)

---

### The `LinkedList` constructor

The `LinkedList` constructor needs to initialize the three fields:

1. `Listnode<E> items` (the pointer to the header node)
2. `Listnode<E> lastNode` (the pointer to the last node in the list)
3. `int numItems`

so that the list is empty. An empty list is one that has just a header node, pointed to by both `items` and `lastNode`. As for the array implementation, `numItems` should be set to zero.

---

### TEST YOURSELF #6

Write the constructor.

[solution](#)

---

## Comparison: Lists via Arrays versus via Linked Lists

When comparing the List implementations using linked lists and using arrays, we should consider:

- space requirements
- time requirements
- ease of implementation

In terms of space, each implementations has its advantages and disadvantages:

- In the linked-list implementation, one pointer must be stored for every item in the list, while the array stores only the items themselves.
- On the other hand, the space used for a linked list is always proportional to the number of items in the list. This is not necessarily true for the array implementation as described: if a lot of items are added to a list and then removed, the size of the array can be arbitrarily greater than the number of items in the list. However, we could fix this problem by modifying the `remove` operation to shrink the array when it becomes too empty.

In terms of time:

- Adding an item to the end of a list is  $O(1)$  for the array implementation if we can use a "shadow" array (as discussed in class) to avoid the  $O(N)$  cost of calling method `expandArray`. It is also  $O(1)$  for the linked-list implementation as long as we have a `lastNode` field.
- Adding an item at a given position requires  $O(N)$  worst-case time for the array implementation, because existing items need to be moved. The operation is also  $O(N)$  in the worst case for the linked-list implementation, because we have to find the node currently in that position. So this operation is worst-case  $O(N)$  for both implementations. However, it is worth noting that for the array implementation adding closer to the beginning of the list is worst and adding toward the end of the list is best, while it is the other way around for the linked-list implementation.
- The `get` operation is  $O(1)$  for the array implementation and worst-case  $O(N)$  for the linked-list implementation.

In terms of ease of implementation, straightforward implementations of both the array and linked-list versions seem reasonably easy. However, the methods for the linked-list version seem to require more special cases.

---

## TEST YOURSELF #7

Assume that lists are implemented using linked lists with header nodes and pointers to the last node in the list. How much time is required (using Big-O notation) to remove the first item from a list? to remove the last item from a list? How do these times compare to the times required for the same operations when the list is implemented using an array?

[solution](#)

---

## Linked List Variations

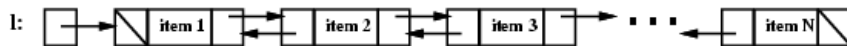
There are several variations on the basic idea of linked lists. Here we will discuss two of them:

1. doubly linked lists
2. circular linked lists

### Doubly linked lists

Recall that, given (only) a pointer to a node  $n$  in a linked list with  $N$  nodes, removing node  $n$  takes time  $O(N)$  in the worst case, because it is necessary to traverse the list looking for the node just before  $n$ . One way to fix this problem is to require **two** pointers: a pointer to the node to be removed and also a pointer to the node just before that one.

Another way to fix the problem is to use a **doubly linked** list. Here's the conceptual picture:



Each node in a doubly linked list contains **three** fields: the data and two pointers. One pointer points to the previous node in the list, and the other pointer points to the next node in the list. The previous pointer of the first node and the next pointer of the last node are both null. Here's the Java class definition for a doubly linked list node:

```
public class DbListNode<E> {
    /** fields */
    private DbListNode<E> prev;
    private E data;
    private DbListNode<E> next;

    /** constructors */
    // 3 constructors
    public DbListNode() {
        this(null, null, null);
    }

    public DbListNode(E d) {
        this(null, d, null);
    }
}
```



```

public Dbllistnode(Dbllistnode<E> p, E d, Dbllistnode<E> n) {
    prev = p;
    data = d;
    next = n;
}

/** methods */
// access to fields
public E getData() {
    return data;
}

public Dbllistnode<E> getNext() {
    return next;
}

public Dbllistnode<E> getPrev() {
    return prev;
}

// modify fields
public void setData(E d) {
    data = d;
}

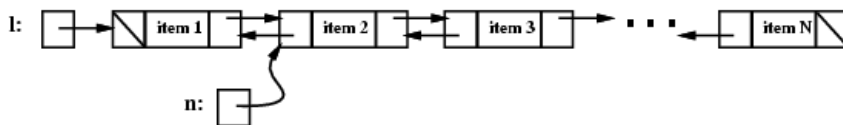
public void setNext(Dbllistnode<E> n) {
    next = n;
}

public void setPrev(Dbllistnode<E> p) {
    prev = p;
}
}

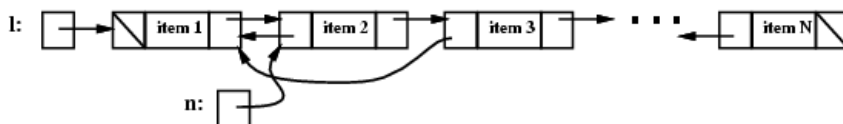
```

To remove a given node *n* from a doubly linked list, we need to change the *prev* field of the node to its right and we need to change the *next* field of the node to its left, as illustrated below.

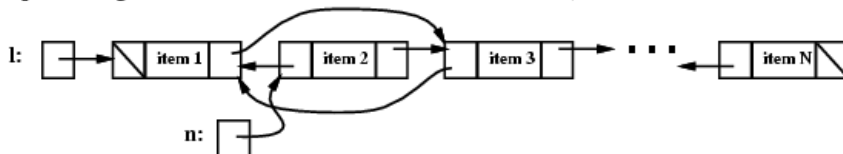
**Original list, with a pointer to a node to be removed:**



**Step 1: Change the prev field of the node to the right of node n:**



**Step 2: Change the next field of the node to the left of node n (n is now removed from the list):**



Here's the code for removing node *n*:

```

// Step 1: change the prev field of the node after n
Dbllistnode<E> tmp = n.getNext();
tmp.setPrev(n.getPrev());

// Step 2: change the next field of the node before n
tmp = n.getPrev();
tmp.setNext(n.getNext());

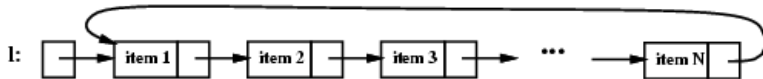
```

Unfortunately, this code doesn't work (causes an attempt to dereference a null pointer) if *n* is either the first or the last node in the list. We can add code to test for these special cases, or we can use a **circular**, doubly linked list, as discussed below.

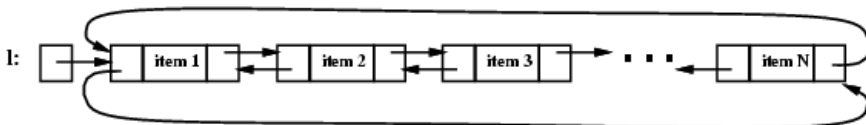
## Circular linked lists

Both singly and doubly linked lists can be made circular. Here are the conceptual pictures:

Circular, singly linked list:



Circular, doubly linked list:



The class definitions are the same as for the non-circular versions. The difference is that, instead of being null, the `next` field of the last node points to the first node and (for doubly linked circular lists) the `prev` field of the first node points to the last node.

The code given above for removing node `n` from a doubly linked list will work correctly *except* when node `n` is the *first* node in the list. In that case, the variable `l` that points to the first node in the list needs to be updated, so special-case code will always be needed unless the list includes a header node.

Another issue that you must address if you use a circular linked list is that if you're not careful, you may end up going round and round in circles! For example, what happens if you try to search for a particular value `val` using code like this:

```
Listnode<E> tmp = l;
while (tmp != null && !tmp.getData().equals(val)) {
    tmp = tmp.getNext();
}
```

and the value is not in the list? You will have an infinite loop!

---

### TEST YOURSELF #8

Write the correct code to search for value `val` in a circular linked list pointed to by `l`. (Assume that the list contains no null data values.)

[solution](#)

---

With circular lists, you don't need pointers to both ends of the list; a pointer to one end suffices. With singly linked circular lists, it is most convenient to use only a pointer to the *last* node. With the structure it is easy to write code that does each of the following three operations in  $O(1)$  (constant) time: `addFirst`, `addLast`, `removeFirst`

---

### TEST YOURSELF #9

Write the code for these operations. Assume you're adding these operations to a class named `CircularLinkedList`, which has a `last` reference to the last node in the circular singly linked list.

[solution](#)

---

## Comparison of Linked List Variations

The major disadvantage of doubly linked lists (over singly linked lists) is that they require more space (every node has two pointer fields instead of one). Also, the code to manipulate doubly linked lists needs to maintain the `prev` fields as well as the `next` fields; the more fields that have to be maintained, the more chance there is for errors.

The major advantage of doubly linked lists is that they make some operations (like the removal of a given node, or a right-to-left traversal of the list) more efficient.

The major advantage of circular lists (over non-circular lists) is that they eliminate some special-case code for some operations. Also, some applications lead naturally to circular list representations. For example, a computer network might best be modeled using a circular list.

# Complexity and Big-O Notation

---

## Contents

- [Introduction](#)
    - [Test Yourself #1](#)
    - [Test Yourself #2](#)
  - [Big-O Notation](#)
  - [How to Determine Complexities](#)
    - [Test Yourself #3](#)
    - [Test Yourself #4](#)
  - [Best-case and Average-case Complexity](#)
  - [When do Constants Matter?](#)
- 

## Introduction

An important question is: How efficient is an algorithm or piece of code? Efficiency covers lots of resources, including:

- CPU (time) usage
- memory usage
- disk usage
- network usage

All are important but we will mostly talk about CPU time in 367. Other classes will discuss other resources (e.g., disk usage may be an important topic in a database class).

Be careful to differentiate between:

1. **Performance**: how much time/memory/disk/... is actually used when a program is run. This depends on the machine, compiler, etc. as well as the code.
2. **Complexity**: how do the resource requirements of a program or algorithm scale, i.e., what happens as the size of the problem being solved gets larger.

Complexity affects performance but not the other way around.

The time required by a method is proportional to the number of "basic operations" that it performs. Here are some examples of basic operations:

- one arithmetic operation (e.g., +, \*).
- one assignment
- one test (e.g., `x == 0`)
- one read
- one write (of a primitive type)

Some methods perform the same number of operations every time they are called. For example, the `size` method of an `ArrayList` always performs just one operation: `return numItems`; so the number of operations is independent of the size of the list. We say that methods like this (that always perform a fixed number of basic operations) require **constant time**.

Other methods may perform different numbers of operations, depending on the value of a parameter or a field. For example, for the array implementation of the List, the `remove` method has to move over all of the items that were to the right of the item that was removed (to fill in the gap). The number of moves depends both on the position of the removed item and the number of items in the list. We call the important factors (the parameters and/or fields whose values affect the number of operations performed) the **problem size** or the **input size**.

When we consider the complexity of a method, we don't really care about the *exact* number of operations that are performed; instead, we care about how the number of operations relates to the problem size. If the problem size doubles, does the number of operations stay the same? double? increase in some other way? For constant-time methods like the `size` method, doubling the problem size does not affect the number of operations (which stays the same).

Furthermore, we are usually interested in the **worst case**: what is the *most* operations that might be performed for a given problem size (other cases -- best case and average case -- are discussed [below](#)). For example, as discussed above, the `remove` method has to move all of the items that come after the removed item one place to the left in the array. In the worst case, *all* of the items in the array must be moved. Therefore, in the worst case, the time for `remove` is proportional to the number of items in the list and we say that the worst-case time for `remove` is **linear** in the number of items in the list. For a linear-time method, if the problem size doubles, the number of operations also doubles.

---

## TEST YOURSELF #1

Assume that lists are implemented using an array. For each of the following List methods, say whether (in the worst case) the number of operations is independent of the size of the list (is a constant-time method) or is proportional to the size of the list (is a linear-time method):

- the constructor
- add (to the end of the list)

- add (at a given position in the list)
- isEmpty
- contains
- get

[solution](#)

Constant and linear times are not the only possibilities. For example, consider method `createList`:

```
ListADT<Object> createList(int n) {
    ListADT<Object> numbers = new SimpleArrayList();
    for (int k = 1; k <= n; k++)
        numbers.add(0, new Integer(k));
    return numbers;
}
```

Note that, for a given  $N$ , the for-loop above is equivalent to:

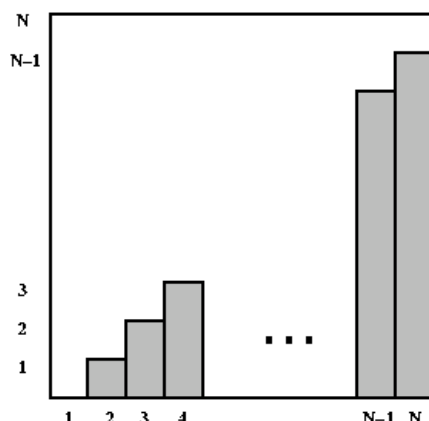
```
numbers.add(0, new Integer(1));
numbers.add(0, new Integer(2));
numbers.add(0, new Integer(3));
...
numbers.add(0, new Integer(n));
```

If we assume that the initial array is large enough to hold  $N$  items, then the number of operations for each call to `add` is proportional to the number of items in the list when `add` is called (because it has to move every item already in the array one place to the right to make room for the new item at position 0). For the  $N$  calls shown above, the list lengths are: 0, 1, 2, ...,  $N-1$ . So what is the total time for all  $N$  calls? It is proportional to  $0 + 1 + 2 + \dots + N-1$ .

Recall that we don't care about the exact time, just how the time depends on the problem size. For method `createList`, the **problem size** is the value of  $N$  (because the number of operations will be different for different values of  $N$ ). It is clear that the time for the  $N$  calls (and therefore the time for method `createList`) is **not** independent of  $N$  (so `createList` is not a constant-time method). Is it proportional to  $N$  (linear in  $N$ )? That would mean that doubling  $N$  would double the number of operations performed by `createList`. Here's a table showing the value of  $0+1+2+\dots+(N-1)$  for some different values of  $N$ :

$N$	$0+1+2+\dots+(N-1)$
4	6
8	28
16	120

Clearly, the value of the sum does more than double when the value of  $N$  doubles, so `createList` is not linear in  $N$ . In the following graph, the bars represent the lengths of the list (0, 1, 2, ...,  $N-1$ ) for each of the  $N$  calls.



The value of the sum  $(0+1+2+\dots+(N-1))$  is the sum of the areas of the individual bars. You can see that the bars fill about half of the square. The whole square is an  $N$ -by- $N$  square, so its area is  $N^2$ ; therefore, the sum of the areas of the bars is about  $N^2/2$ . In other words, the time for method `createList` is proportional to the **square** of the problem size; if the problem size doubles, the number of operations will quadruple. We say that the worst-case time for `createList` is **quadratic** in the problem size.

## TEST YOURSELF #2

Consider the following three algorithms for determining whether anyone in the room has the same birthday as you.

- **Algorithm 1:** You say your birthday and ask whether anyone in the room has the same birthday. If anyone does have the same birthday, they answer yes.
- **Algorithm 2:** You tell the first person your birthday and ask if they have the same birthday; if they say no, you tell the second person your birthday and

ask whether they have the same birthday; etc., for each person in the room.

- **Algorithm 3:** You only ask questions of person 1, who only asks questions of person 2, who only asks questions of person 3, etc. You tell person 1 your birthday and ask if they have the same birthday; if they say no, you ask them to find out about person 2. Person 1 asks person 2 and tells you the answer. If it is no, you ask person 1 to find out about person 3. Person 1 asks person 2 to find out about person 3, etc.

**Question 1:** For each algorithm, what is the factor that can affect the number of questions asked (the *problem size*)?

**Question 2:** In the worst case, how many questions will be asked for each of the three algorithms?

**Question 3:** For each algorithm, say whether it is constant, linear, or quadratic in the problem size in the worst case.

[solution](#)

## Big-O Notation

We express complexity using **big-O notation**. For a problem of size  $N$ :

- a constant-time method is "order 1":  $O(1)$
- a linear-time method is "order  $N$ ":  $O(N)$
- a quadratic-time method is "order  $N$  squared":  $O(N^2)$

Note that the big-O expressions do not have constants or low-order terms. This is because, when  $N$  gets large enough, constants and low-order terms don't matter (a constant-time method will be faster than a linear-time method, which will be faster than a quadratic-time method). See [below](#) for an example.

### Formal definition:

A function  $T(N)$  is  $O(F(N))$  if for some constant  $c$  and for all values of  $N$  greater than some value  $n_0$ :

$$T(N) \leq c * F(N)$$

The idea is that  $T(N)$  is the *exact* complexity of a method or algorithm as a function of the problem size  $N$  and that  $F(N)$  is an *upper-bound* on that complexity (i.e., the actual time/space or whatever for a problem of size  $N$  will be no worse than  $F(N)$ ). In practice, we want the smallest  $F(N)$  -- the *least* upper bound on the actual complexity.

For example, consider  $T(N) = 3 * N^2 + 5$ . We can show that  $T(N)$  is  $O(N^2)$  by choosing  $c = 4$  and  $n_0 = 2$ . This is because for all values of  $N$  greater than 2:

$$3 * N^2 + 5 \leq 4 * N^2$$

$T(N)$  is *not*  $O(N)$  because whatever constant  $c$  and value  $n_0$  you choose, I can always find a value of  $N$  greater than  $n_0$  so that  $3 * N^2 + 5$  is greater than  $c * N$ .

## How to Determine Complexities

In general, how can you determine the running time of a piece of code? The answer is that it depends on what kinds of statements are used.

### Sequence of statements

```
statement 1;
statement 2;
...
statement k;
```

(Note: this is code that really is exactly  $k$  statements; this is *not* an unrolled loop like the  $N$  calls to add shown above.) The total time is found by adding the times for all statements:

$$\text{total time} = \text{time}(\text{statement 1}) + \text{time}(\text{statement 2}) + \dots + \text{time}(\text{statement k})$$

If each statement is "simple" (only involves basic operations) then the time for each statement is constant and the total time is also constant:  $O(1)$ . In the following examples, assume the statements are simple unless noted otherwise.

### if-then-else statements

```
if (condition) {
    sequence of statements 1
}
else {
    sequence of statements 2
}
```

Here, either sequence 1 will execute, or sequence 2 will execute. Therefore, the worst-case time is the slowest of the two possibilities:  $\max(\text{time}(\text{sequence 1}), \text{time}(\text{sequence 2}))$ . For example, if sequence 1 is  $O(N)$  and sequence 2 is  $O(1)$  the worst-case time for the whole if-then-else statement would be  $O(N)$ .

### for loops

```
for (i = 0; i < N; i++) {
    sequence of statements
}
```

The loop executes  $N$  times, so the sequence of statements also executes  $N$  times. Since we assume the statements are  $O(1)$ , the total time for the for loop is  $N * O(1)$ , which is  $O(N)$  overall.

## Nested loops

First we'll consider loops where the number of iterations of the inner loop is independent of the value of the outer loop's index. For example:

```
for (i = 0; i < N; i++) {
    for (j = 0; j < M; j++) {
        sequence of statements
    }
}
```

The outer loop executes  $N$  times. Every time the outer loop executes, the inner loop executes  $M$  times. As a result, the statements in the inner loop execute a total of  $N * M$  times. Thus, the complexity is  $O(N * M)$ . In a common special case where the stopping condition of the inner loop is  $j < N$  instead of  $j < M$  (i.e., the inner loop also executes  $N$  times), the total complexity for the two loops is  $O(N^2)$ .

Now let's consider nested loops where the number of iterations of the inner loop depends on the value of the outer loop's index. For example:

```
for (i = 0; i < N; i++) {
    for (j = i+1; j < N; j++) {
        sequence of statements
    }
}
```

Now we can't just multiply the number of iterations of the outer loop times the number of iterations of the inner loop, because the inner loop has a different number of iterations each time. So let's think about how many iterations that inner loop has. That information is given in the following table:

Value of i	Number of iterations of inner loop
0	N
1	N-1
2	N-2
...	...
N-2	2
N-1	1

So we can see that the total number of times the sequence of statements executes is:  $N + N-1 + N-2 + \dots + 3 + 2 + 1$ . We've seen that formula before: the total is  $O(N^2)$ .

## TEST YOURSELF #3

What is the worst-case complexity of each of the following code fragments?

- Two loops in a row:

```
for (i = 0; i < N; i++) {
    sequence of statements
}
for (j = 0; j < M; j++) {
    sequence of statements
}
```

How would the complexity change if the second loop went to  $N$  instead of  $M$ ?

- A nested loop followed by a non-nested loop:

```
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
        sequence of statements
    }
}
for (k = 0; k < N; k++) {
    sequence of statements
}
```

- A nested loop in which the number of times the inner loop executes depends on the value of the outer loop index:

```

for (i = 0; i < N; i++) {
    for (j = N; j > i; j--) {
        sequence of statements
    }
}

```

[solution](#)

### Statements with method calls:

When a statement involves a method call, the complexity of the statement includes the complexity of the method call. Assume that you know that method  $f$  takes constant time, and that method  $g$  takes time proportional to (linear in) the value of its parameter  $k$ . Then the statements below have the time complexities indicated.

```

f(k); // O(1)
g(k); // O(k)

```

When a loop is involved, the same rule applies. For example:

```

for (j = 0; j < N; j++)
    g(N);

```

has complexity ( $N^2$ ). The loop executes  $N$  times and each method call  $g(N)$  has complexity  $O(N)$ .

## TEST YOURSELF #4

For each of the following loops with a method call, determine the overall complexity. As above, assume that method  $f$  takes constant time, and that method  $g$  takes time linear in the value of its parameter.

1. 

```
for (j = 0; j < N; j++)
    f(j);
```
2. 

```
for (j = 0; j < N; j++)
    g(j);
```
3. 

```
for (j = 0; j < N; j++)
    g(k);
```

[solution](#)

## Best-case and Average-case Complexity

Some methods may require different amounts of time on different calls, even when the problem size is the same for both calls. For example, consider the add method that adds an item to the end of the list. In the worst case (the array is full), that method requires time proportional to the number of items in the list (because it has to copy all of them into the new, larger array). However, when the array is not full, add will only have to copy one value into the array, so in that case its time is independent of the length of the list, i.e., constant time.

In general, we may want to consider the **best-case** and **average-case** time requirements of a method as well as its worst-case time requirements. Which is considered the most important will depend on several factors. For example, if a method is part of a time-critical system like one that controls an airplane, the worst-case times are probably the most important (if the plane is flying towards a mountain and the controlling program can't make the next course correction until it has performed a computation, then the best-case and average-case times for that computation are not relevant -- the computation needs to be guaranteed to be fast enough to finish before the plane hits the mountain).

On the other hand, if occasionally waiting a long time for an answer is merely inconvenient (as opposed to life-threatening), it may be better to use an algorithm with a slow worst-case time and a fast average-case time, rather than one with so-so times in both the average and worst cases.

Note that calculating the average-case time for a method can be tricky. You need to consider all possible values for the important factors and whether they will be distributed evenly.

## When do Constants Matter?

Recall that when we use big-O notation, we drop constants and low-order terms. This is because when the problem size gets sufficiently large, those terms don't matter. However, this means that two algorithms can have the **same** big-O time complexity even though one is always faster than the other. For example, suppose algorithm 1 requires  $N^2$  time and algorithm 2 requires  $10 * N^2 + N$  time. For both algorithms, the time complexity is  $O(N^2)$ , but algorithm 1 will always be faster than algorithm 2. In this case, the constants and low-order terms **do** matter in terms of which algorithm is actually faster.

However, it is important to note that constants do **not** matter in terms of the question of how an algorithm "scales" (i.e., how does the algorithm's time change when the problem size doubles). Although an algorithm that requires  $N^2$  time will always be faster than an algorithm that requires  $10 * N^2$  time, for **both** algorithms, if the problem size doubles, the actual time will quadruple.

When two algorithms have **different** big-O time complexity, the constants and low-order terms only matter when the problem size is **small**. For example, even if

there are large constants involved, a linear-time algorithm will always eventually be faster than a quadratic-time algorithm. This is illustrated in the following table, which shows the value of  $100 \cdot N$  (a time that is linear in  $N$ ) and the value of  $N^2/100$  (a time that is quadratic in  $N$ ) for some values of  $N$ . For values of  $N$  less than  $10^4$ , the quadratic time is smaller than the linear time. However, for all values of  $N$  greater than  $10^4$ , the linear time is smaller.

<b>N</b>	<b><math>100 \cdot N</math></b>	<b><math>N^2/100</math></b>
$10^2$	$10^4$	$10^2$
$10^3$	$10^5$	$10^4$
$10^4$	$10^6$	$10^6$
$10^5$	$10^7$	$10^8$
$10^6$	$10^8$	$10^{10}$
$10^7$	$10^9$	$10^{12}$



## Stacks and Queues

## Contents

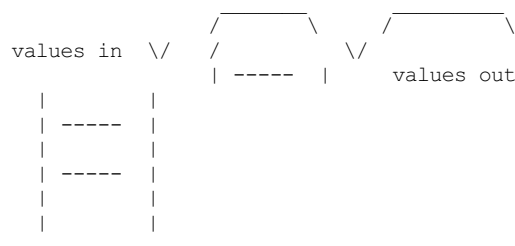
- Introduction
  - Stacks
  - Queues
- Implementing Stacks
  - Array Implementation
    - Test Yourself #1
    - Test Yourself #2
    - Test Yourself #3
  - Linked-list Implementation
    - Test Yourself #4
    - Test Yourself #5
- Implementing Queues
  - Array Implementation
  - Linked-list Implementation
- Comparison of Array and Linked-List Implementations
- Applications of Stacks and Queues
- Test Yourself #6

## Introduction

Both Stacks and Queues are like Lists (ordered collections of items), but with more restricted operations. They can both be implemented either using an array or using a linked list to hold the actual items.

## Stacks

The conceptual picture of a Stack ADT is something like this:



Think of a stack of newspapers or trays in a cafeteria. The only item that can be taken out (or even seen) is the *most recently added* (or **top**) item; a Stack is a **Last-In-First-Out (LIFO)** abstract data type.

Here are the Stack ADT operations:

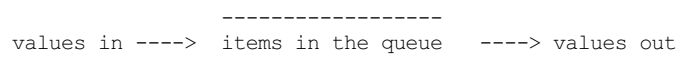
Operation	Description
boolean isEmpty()	return true iff the Stack is empty
void push(E ob)	add ob to the top of the Stack
E pop()	remove and return the item from the top of the Stack (error if the Stack is empty)
E peek()	return the item that is on the top of the Stack, but do not remove it (error if the Stack is empty)

In Java we create the `StackADT` interface as:

```
public interface StackADT<E> {
    boolean isEmpty();
    void push(E ob);
    E pop() throws EmptyStackException;
    E peek() throws EmptyStackException;
}
```

## Queues

The conceptual picture of a Queue ADT is something like this:





Think of people standing in line. A Queue is a **First-In-First-Out (FIFO)** abstract data type. Items can only be added at the **rear** of the queue and the only item that can be removed is the one at the **front** of the queue.

Here are the Queue ADT operations:

Operation	Description
boolean isEmpty()	return true iff the Queue is empty
void enqueue(E ob)	add ob to the rear of the Queue
E dequeue()	remove and return the item from the front of the Queue (error if the Queue is empty)

In Java we create the QueueADT interface as:

```
public interface QueueADT<E> {
    boolean isEmpty();
    void enqueue(E ob);
    E dequeue() throws EmptyQueueException;
}
```

## Implementing Stacks

The Stack ADT is very similar to the List ADT; therefore, their implementations are also quite similar.

### Array Implementation

Below is the definition of the ArrayStack class, using an array to store the items in the stack; note that we include a static final variable INITSIZE, to be used by the ArrayStack constructor as the initial size of the array (the same thing was done for the ArrayList class).

```
public class ArrayStack<E> implements StackADT<E> {
    // *** fields ***
    private static final int INITSIZE = 10; // initial array size
    private E[] items; // the items in the stack
    private int numItems; // the number of items in the stack

    // *** constructor ***
    public ArrayStack() { ... }

    // *** required StackADT methods ***

    // add items
    public void push(E ob) { ... }

    // remove items
    public E pop() throws EmptyStackException { ... }

    // other methods
    public E peek() throws EmptyStackException { ... }
    public boolean isEmpty() { ... }
}
```

## TEST YOURSELF #1

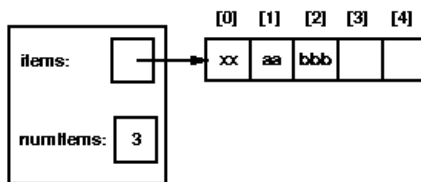
Write the ArrayStack constructor.

[solution](#)

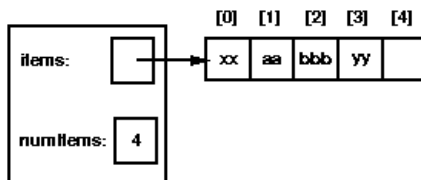
The push method is like the version of the List add method that adds an object to the end of the list (because items are always pushed onto the **top** of the stack). Note that it is up to us as the designers of the ArrayStack class to decide which end of the array corresponds to the top of the stack. We could choose always to add items at the beginning of the array or always to add items at the end of the array. However, it is clearly not a good idea to add items at the beginning of the array since that requires moving all existing items; i.e., that choice would make push be  $O(N)$  (where  $N$  is the number of items in the stack). If we add items at the end of the array, then the time for push depends on how we handle expanding the array. The naive implementation makes push  $O(1)$  when the array is not full,  $O(N)$  when it is full, and  $O(1)$  on average. If we use the "shadow array" trick, then push is always  $O(1)$ .

Here are before and after pictures, illustrating the effects of a call to push:

BEFORE CALLING `push`



AFTER CALLING `push(yy)`

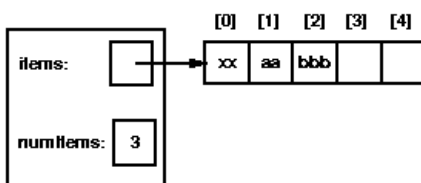


And here's the code for the `push` method:

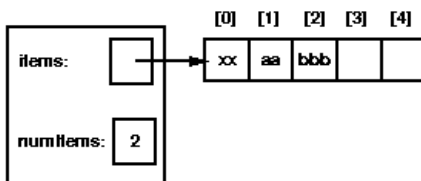
```
public void push(E ob) {  
    if (items.length == numItems) {  
        expandArray();  
    }  
    items[numItems] = ob;  
    numItems++;  
}
```

The `pop` method needs to remove the top-of-stack item and return it, as illustrated below.

BEFORE CALLING `pop`



AFTER CALLING `pop`  
(the value `bbb` is returned)



Note that, in the picture, the value "bbb" is still in `items[2]`; however, that value is no longer in the stack because `numItems` is 2 (which means that `items[1]` is the last item in the stack).

---

## TEST YOURSELF #2

Complete the `pop` method, using the following header

```
public E pop() throws EmptyStackException {  
  
}
```

[solution](#)

---

The `peek` method is very similar to the `pop` method, except that it only returns the top-of-stack value without changing the stack. The `isEmpty` method simply

returns true iff `numItems` is zero.

### TEST YOURSELF #3

Fill in the following table, using Big-O notation to give the worst and average-case times for each of the `ArrayStack` methods for a stack of size  $N$ .

Operation	Worst-case Time	Average-case Time
constructor		
isEmpty		
push		
pop		
peek		

[solution](#)

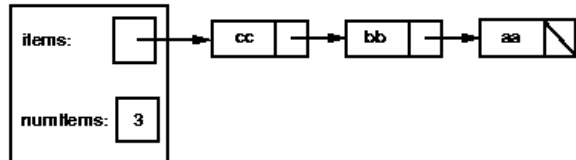
## Linked-list Implementation

To implement a stack using a linked list, we must first define the `Listnode` class. The [Listnode definition](#) is the same one we used for the linked-list implementation of the `LinkedList` class.

The signatures of the methods of the `StackADT` interface are independent of whether the stack is implemented using an array or using a linked list; to implement the `StackADT` using a linked list, we'll change the name of the class implementing the stack and the type of the `items` field:

```
public class LLStack<E> implements StackADT<E> {
    private Listnode<E> items; // pointer to the linked list of items in the stack
    ....
}
```

As discussed above, an important property of stacks is that items are only pushed and popped at one end (the top of the stack). If we implement a stack using a linked list, we can choose which end of the list corresponds to the top of the stack. It is easiest and most efficient to add and remove items at the front of a linked list, therefore, we will choose the front of the list as the top of the stack (i.e., the `items` field will be a pointer to the node that contains the top-of-stack item). Below is a picture of a stack represented using a linked list; in this case, items have been pushed in alphabetical order, so "cc" is at the top of the stack:



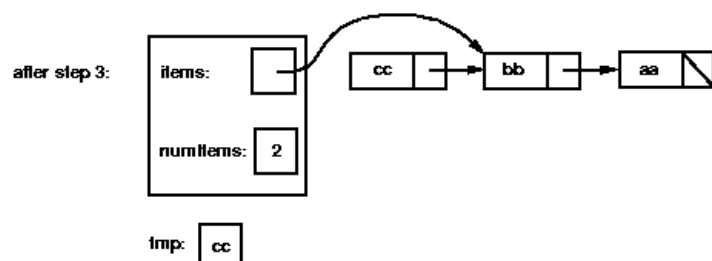
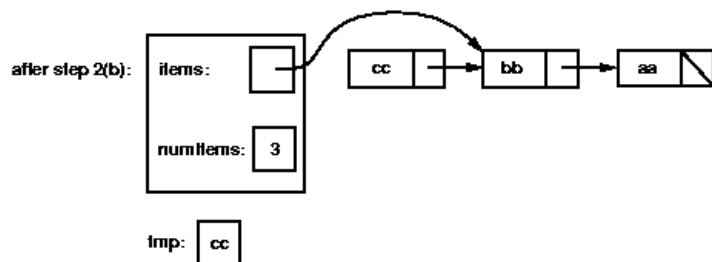
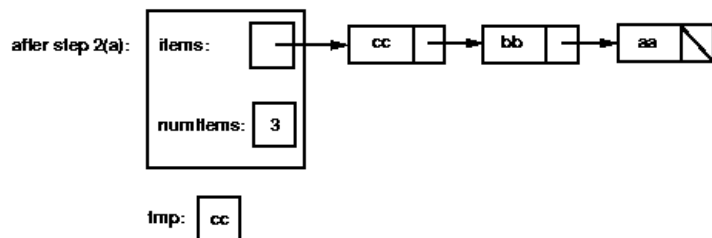
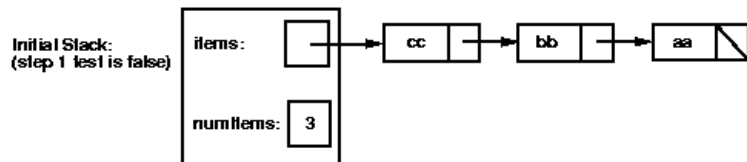
Notice that, in the picture, the top of stack is to the left (at the front of the list), while for the array implementation, the top of stack was to the right (at the end of the array).

Let's consider how to write the `pop` method. It will need to perform the following steps:

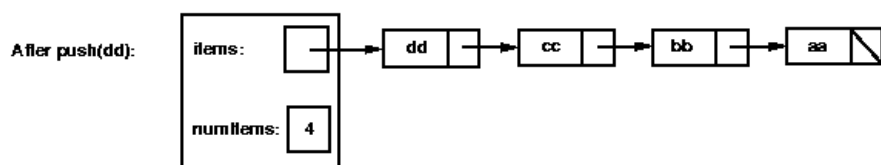
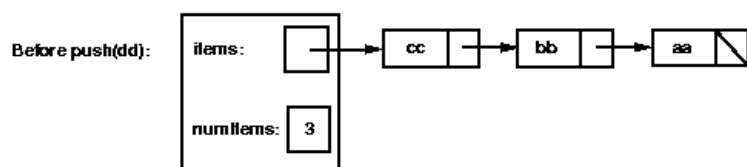
1. Check whether the stack is empty; if so, throw an `EmptyStackException`.
2. Remove the first node from the linked list by setting `items = items.getNext()`.
3. Decrement `numItems`.
4. Return the value that was in the first node in the list.

Note that by the time we get to the last step (returning the top-of-stack value), the first node has already been removed from the list, so we need to save its value in order to return it (we'll call that step 2(a)). Here's the code and an illustration of what happens when `pop` is called for a stack containing "cc", "bb", "aa" (with "cc" at the top).

```
public E pop() throws EmptyStackException {
    if (isEmpty())
        throw new EmptyStackException(); // step 1
    E tmp = items.getData();               // step 2(a)
    items = items.getNext();               // step 2(b)
    numItems--;                            // step 3
    return tmp;                            // step 4
}
```



Now let's consider the `push` method. Here are before and after pictures, illustrating the effect of a call to `push` when the stack is implemented using a linked list:



The steps that need to be performed are:

1. Create a new node whose data field contains the object to be pushed and whose next field contains a pointer to the first node in the list (or null if the list is empty). Note that the value for the next field of the new node is the value in the `LLStack`'s `items` field.
2. Change `items` to point to the new node.
3. Increment `numItems`.

#### TEST YOURSELF #4

Complete the `push` method, using the following header.

```
public void push(E ob) {
}
```

The remaining methods (the constructor, peek, and empty) are quite straightforward. You should be able to implement them without any major problems.

### TEST YOURSELF #5

Fill in the following table, using Big-O notation to give the worst-case times for each of the `LLStack` methods for a stack of size  $N$ , assuming a linked-list implementation. Look back at the table you filled in for the array implementation. How do the times compare? What are the advantages and disadvantages of using an array vs using a linked list to implement the Stack ADT?

Operation	Worst-case Time
constructor	
isEmpty	
push	
pop	
peek	

## Implementing Queues

The main difference between a stack and a queue is that a stack is only accessed from the top, while a queue is accessed from both ends (from the rear for adding items, and from the front for removing items). This makes both the array and the linked-list implementation of a queue more complicated than the corresponding stack implementations.

### Array Implementation

Let's first consider a Queue implementation that is very similar to our (array-based) List implementation. Here's the class definition:

```
public class ArrayQueue<E> implements QueueADT<E> {
    // *** fields ***
    private static final int INITSIZE = 10; // initial array size
    private E[] items; // the items in the queue
    private int numItems; // the number of items in the queue

    /*** constructor ***/
    public ArrayQueue() { ... }

    /*** required QueueADT methods ***/

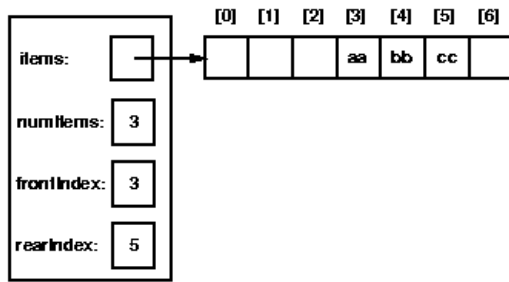
    // add items
    public void enqueue(E ob) { ... }

    // remove items
    public E dequeue() throws EmptyQueueException { ... }

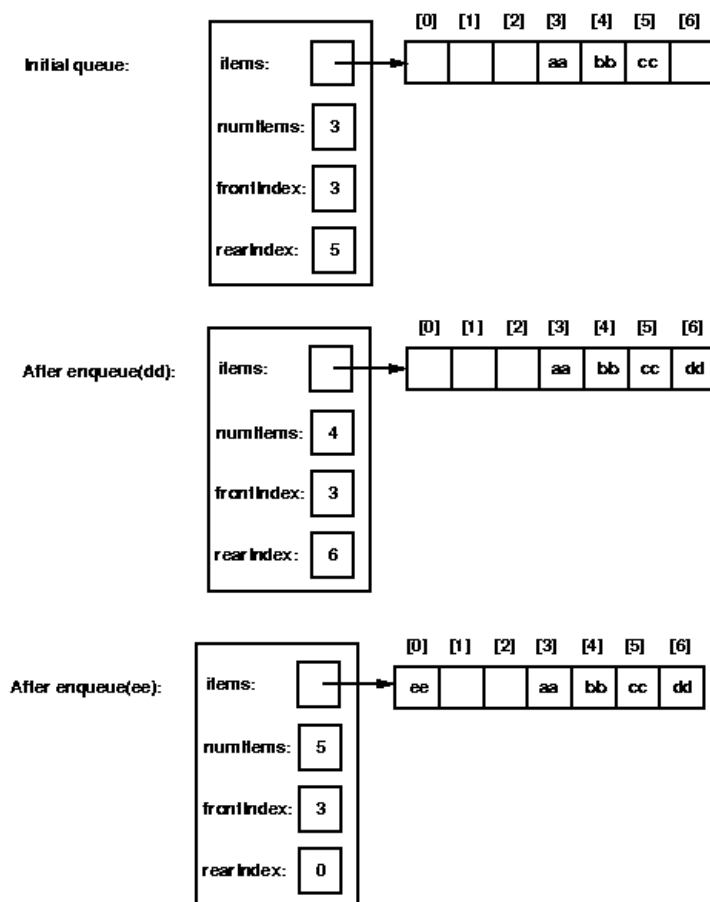
    // other methods
    public boolean empty() { ... }
}
```

We could implement enqueue by adding the new item at the end of the array and implement dequeue by saving the first item in the array, moving all other items one place to the left, and returning the saved value. The problem with this approach is that, although the enqueue operation is efficient, the dequeue operation is not -- it requires time proportional to the number of items in the queue.

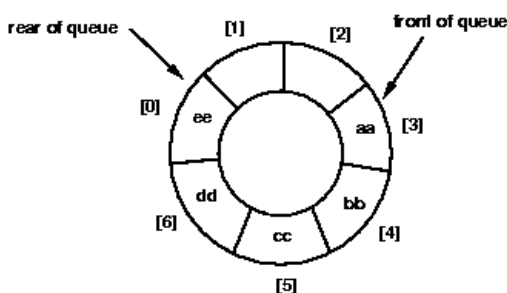
To make both enqueue and dequeue efficient, we need the following insight: There is no reason to force the front of the queue always to be in `items[0]`, we can let it "move up" as items are dequeued. To do this, we need to keep track of the indexes of the items at the front and rear of the queue (so we need to add two new fields to the `ArrayQueue` class, `frontIndex` and `rearIndex`, both of type `int`). To illustrate this idea, here is a picture of a queue after some enqueue and dequeue operations have been performed:



Now think about what should happen to this queue if we enqueue two more items: "dd" and "ee". Clearly "dd" should be stored in `items[6]`. Then what? We could increase the size of the array and put "ee" in `items[7]`, but that would lead to wasted space -- we would never reuse `items[0]`, `items[1]`, or `items[2]`. In general, the items in the queue would keep "sliding" to the right in the array, causing more and more wasted space at the beginning of the array. A better approach is to let the rear index "wrap around" (in this case, from 6 to 0) as long as there is empty space in the front of the array. Similarly, if after enqueueing "dd" and "ee" we dequeue four items (so that only "ee" is left in the queue), the front index will have to wrap around from 6 to 0. Here's a picture of what happens when we enqueue "dd" and "ee":



Conceptually, the array is a **circular** array. It may be easier to visualize it as a circle. For example, the array for the final queue shown above could be thought of as:



We still need to think about what should happen if the array is full; we'll consider that case in a minute. Here's the code for the `enqueue` method, with the "full array" case still to be filled in:

```
public void enqueue(E ob) {
    // check for full array and expand if necessary
    if (items.length == numItems) {
        // code missing here
    }

    // use auxiliary method to increment rear index with wraparound
    rearIndex = incrementIndex(rearIndex);

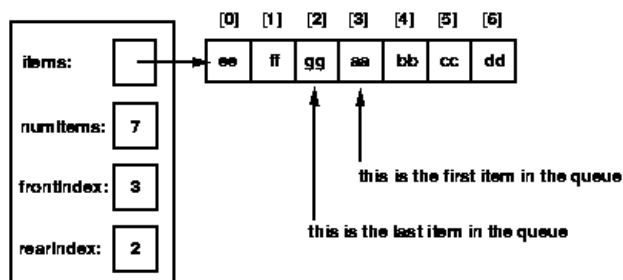
    // insert new item at rear of queue
    items[rearIndex] = ob;
    numItems++;
}

private int incrementIndex(int index) {
    if (index == items.length-1)
        return 0;
    else
        return index + 1;
}
```

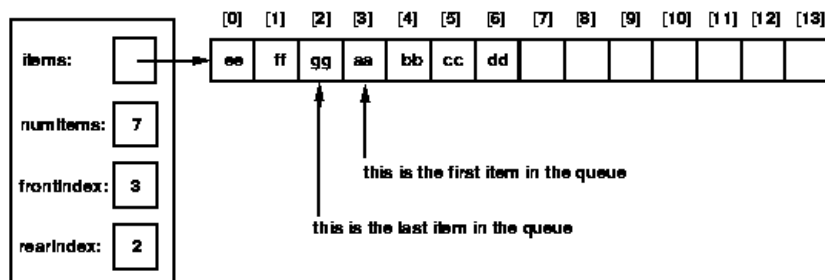
Note that instead of using `incrementIndex` we could use the mod operator (`%`), and write: `rearIndex = (rearIndex + 1) % items.length`. However, the mod operator is quite slow and it is easy to get that expression wrong, so we will use the auxiliary method (with a check for the "wrap-around" case) instead.

To see why we can't simply use `expandArray` when the array is full, consider the picture shown below.

#### INITIAL (FULL) QUEUE:



#### AFTER CALLING `expandArray`:



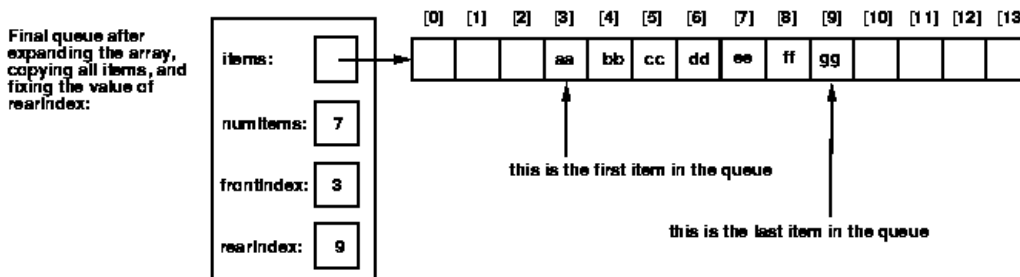
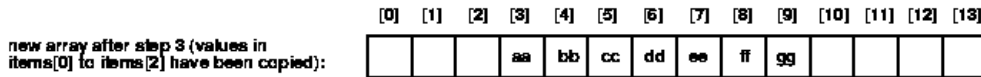
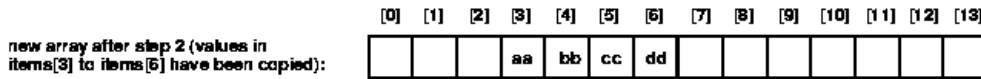
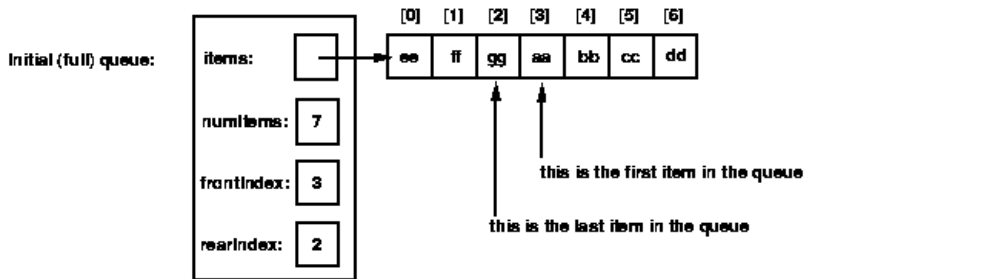
After calling `expandArray`, the last item in the queue is still right before the first item-- there is still no place to put the new item (and there is a big gap in the middle of the queue, from `items[7]` to `items[13]`). The problem is that `expandArray` copies the values in the old array into the *same* positions in the new array. This does not work for the queue implementation; we need to move the "wrapped-around" values to come after the non-wrapped-around values in the new array.

The steps that need to be carried out when the array is full are:

1. Allocate a new array of twice the size.
2. Copy the values in the range `items[frontIndex]` to `items[items.length-1]` into the new array (starting at position `frontIndex` in the new array).
3. Copy the values in the range `items[0]` to `items[rearIndex]` into the new array (starting at position `items.length` in the new array). Note: if the front of the queue was in `items[0]`, then all of the values were copied by step 2, so this step is not needed.
4. Set `items` to point to the new array.
5. Fix the value of `rearIndex`.

Here's an illustration:





And here's the final code for enqueue:

```
public void enqueue(E ob) {
    // check for full array and expand if necessary
    if (items.length == numItems) {
        E[] tmp = (E[]) (new Object[items.length*2]);
        System.arraycopy(items, frontIndex, tmp, frontIndex,
            items.length-frontIndex);
        if (frontIndex != 0) {
            System.arraycopy(items, 0, tmp, items.length, frontIndex);
        }
        items = tmp;
        rearIndex = frontIndex + numItems - 1;
    }

    // use auxiliary method to increment rear index with wraparound
    rearIndex = incrementIndex(rearIndex);

    // insert new item at rear of queue
    items[rearIndex] = ob;
    numItems++;
}
```

The dequeue method will also use method incrementIndex to add one to frontIndex (with wrap-around) before returning the value that was at the front of the queue.

The other ArrayQueue method, isEmpty, is the same as for the ArrayStack class -- it just uses the value of the numItems field.

## Linked-list Implementation

The first decision in planning the linked-list implementation of the Queue ADT is which end of the list will correspond to the front of the queue. Recall that items need to be added to the rear of the queue and removed from the front of the queue. Therefore, we should make our choice based on whether it is easier to add/remove a node from the front/end of a linked list.

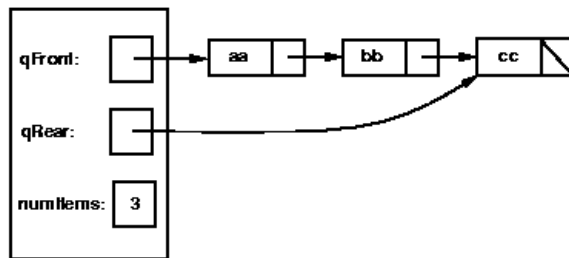
If we keep pointers to both the first and last nodes of the list, we can add a node at either end in constant time. However, while we can remove the first node in the list in constant time, removing the last node requires first locating the *previous* node, which takes time proportional to the length of the list. Therefore, we should choose to make the end of the list be the rear of the queue and the front of the list be the front of the queue.

The class definition is the similar to the array implementation,:

```
public class LLQueue<E> implements QueueADT<E> {
    // *** fields ***
```

```
private Listnode<E> qFront; // pointer to the front of the queue
                           // (the first node in the list)
private Listnode<E> qRear; // pointer to the rear of the queue
                           // (the last node in the list)
private int numItems;      // the number of items in the queue
```

Here's a picture of a queue with three items, aa, bb, cc, with aa at the front of the queue:



You should be able to write all of the `LLQueue` methods using the code you wrote for the linked-list implementation of the List ADT as a guide.

## Comparison of Array and Linked-List Implementations

The advantages and disadvantages of the two implementations are essentially the same as the advantages and disadvantages in the case of the List ADT:

- In the linked-list implementation, one pointer must be stored for every item in the stack/queue, while the array stores only the items themselves.
- On the other hand, the space used for a linked list is always proportional to the number of items in the list. This is not necessarily true for the array implementation as described: if a lot of items are added to a stack/queue and then removed, the size of the array can be arbitrarily greater than the number of items in the stack/queue. However, we could fix this problem by modifying the pop/dequeue operations to shrink the array when it becomes too empty.
- For the array implementation, the worst-case times for the push and enqueue methods are  $O(N)$  for the naive implementation, for a stack/queue with  $N$  items (to allocate a new array and copy the values); using the "shadow array" trick, those two operations are  $O(1)$ . For the linked-list implementation, push and enqueue are always  $O(1)$ .

## Applications of Stacks and Queues

Stacks are used to manage methods at runtime (when a method is called, its parameters and local variables are pushed onto a stack; when the method returns, the values are popped from the stack). Many parsing algorithms (used by compilers to determine whether a program is syntactically correct) involve the use of stacks. Stacks can be used to evaluate arithmetic expressions (e.g., by a simple calculator program) and they are also useful for some operations on [graphs](#), a data structure we will learn about later in the semester.

Queues are useful for many simulations and are also used for some operations on graphs and trees.

### TEST YOURSELF #6

Complete method `reverseQ`, whose header is given below. Method `reverseQ` should use a Stack to reverse the order of the items in its Queue parameter.

```
public static<E> void reverseQ(QueueADT<E> q) {
    // precondition: q contains x1 x2 ... xN (with x1 at the front)
    // postcondition: q contains xN ... x2 x1 (with xN at the front)
}
```

[solution](#)

# Recursion

---

## Contents

- [Introduction](#)
    - [Test Yourself #1](#)
  - [How Recursion Really Works](#)
    - [Test Yourself #2](#)
  - [Recursion vs Iteration](#)
    - [Factorial](#)
    - [Fibonacci](#)
    - [Test Yourself #3](#)
  - [Recursive Data Structures](#)
    - [Test Yourself #4](#)
  - [Analyzing Runtime for Recursive Methods](#)
    - [Informal Reasoning](#)
    - [Using Recurrence Equations](#)
      - [Test Yourself #5](#)
  - [Using Mathematical Induction to Prove the Correctness of Recursive Code](#)
  - [Summary](#)
- 

## Introduction

Recursion is:

- A way of thinking about problems.
- A method for solving problems.
- Related to mathematical induction.

A method is **recursive** if it can call itself, either directly:

```
void f() {  
    ... f() ...  
}
```

or indirectly:

```
void f() {  
    ... g() ...  
}  
  
void g() {  
    ... f() ...  
}
```

You might wonder about the following issues:

Q: Does using recursion usually make your code *faster*?

A: No.

Q: Does using recursion usually use *less memory*?

A: No.

Q: Then *why* use recursion?

A: It sometimes makes your code much *simpler*!

One way to think about recursion:

- When a recursive call is made, the method *clones itself*, making new copies of:
  - the code,
  - the local variables (with their initial values),
  - the parameters
- Each copy of the code includes a marker indicating the current position. When a recursive call is made, the marker in the old copy of the code is just after the call; the marker in the "cloned" copy is at the beginning of the method.
- When the method returns, *that* clone goes away, but the previous ones are still there and know what to execute next because their current position in the code was saved (indicated by the marker).

Here's an example of a simple recursive method:

```
void printInt( int k ) {
```

```

1.    if (k == 0) {
2.        return;
3.    }
4.    System.out.println( k );
5.    printInt( k - 1 );
6.    System.out.println( "all done" );
    }

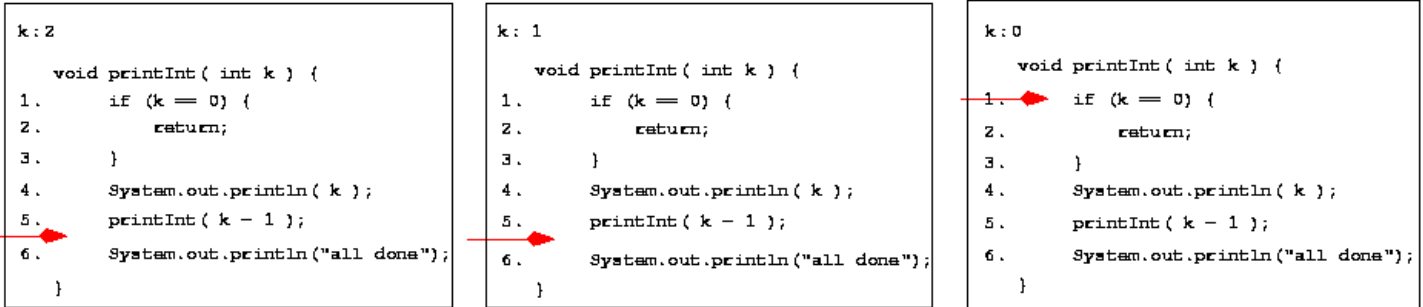
```

If the call `printInt(2)` is made, three "clones" are created, as illustrated below:

Clone 1

Clone 2

Clone 3



Output

2

1

so far:

The original call causes 2 to be output, and then a recursive call is made, creating a clone with  $k = 1$ . That clone executes line 1 (the `if` condition is false), line 4 (prints 1), and line 5 (makes another recursive call, creating a clone with  $k = 0$ ). That clone just returns (goes away) because the `if` condition is true. The previous clone executes line 6 (the line after its "marker") then returns, and similarly for the original clone.

Now let's think about what we have to do to make sure that recursive methods work correctly. First, consider the following recursive method:

```

void badPrint(int k) {
    System.out.println(k);
    badPrint(k + 1);
}

```

Note that a runtime error will occur when the call `badPrint(2)` is made (in particular, an error message like `"java.lang.StackOverflowError"` will be printed, and the program will stop). This is because there is no code that prevents the recursive call from being made again and again and .... and eventually the program runs out of memory (to store all the clones). This is an example of an **infinite recursion**. It inspires:

### \*\*\* RECURSION RULE #1 \*\*\*

Every recursive method must have a **base case** -- a condition under which no recursive call is made -- to prevent infinite recursion.

Here's another example; this version does have a base case, but the call `badPrint2(2)` will still cause an infinite recursion:

```

void badPrint2(int k) {
    if (k < 0) {
        return;
    }
    System.out.println(k);
    badPrint2(k + 1);
}

```

This inspires:

### \*\*\* RECURSION RULE #2 \*\*\*

Every recursive method must **make progress** toward the base case to prevent infinite recursion.

## TEST YOURSELF #1

Consider the method `printInt`, repeated below.

```

void printInt(int k) {
    if (k == 0) {
        return;
    }
    System.out.println(k);
    printInt(k - 1);
}

```

Does it obey recursion rules 1 and 2? Are there calls that will lead to an infinite recursion? If yes, how could it be fixed?

[solution](#)

# How Recursion Really Works

This is how method calls (recursive and non-recursive) really work:

- At runtime, a stack of **activation records** (ARs) is maintained: one AR for each active method, where "active" means: has been called, has not yet returned. This stack is also referred to as the **call stack**.
- Each AR includes space for:
  - the method's parameters,
  - the method's local variables,
  - the return address -- where (in the code) to start executing after the method returns.
- When a method is called, its AR is pushed onto the stack. The return address in that AR is the place in the code just after the call (so the return address is the "marker" for the previous "clone").
- When a method is about to return, the return address in its AR is saved, its AR is popped from the stack, and control is transferred to the place in the code referred to by the return address.

Example (no recursion):

```
1. void printChar (char c) {  
2.     System.out.print(c);  
3. }  
4.  
5. void main ( ... ) {  
6.     char ch = 'a';  
7.     printChar(ch);  
8.     ch = 'b';  
9.     printChar(ch);  
10. }
```

The runtime stack of activation records is shown below, first as it would be just before the first call to `printChar` (just after line 6) and then as it would be while `printChar` is executing (lines 1 - 3). Note that the return address stored in `printChar`'s AR is the place in `main`'s code where execution will resume when `printChar` returns.

## STACK BEFORE 1ST CALL TO `printChar`

**main's AR:**

<b>ch: 'a'</b>
<b>return addr: system</b>

## STACK WHEN `printChar` HAS BEEN CALLED

**printChar's AR:**

<b>c: 'a'</b>
<b>return addr: line 8</b>

**main's AR:**

<b>ch: 'a'</b>
<b>return addr: system</b>

When the first call to `printChar` returns, the top activation record is popped from the stack and the `main` method begins executing again at line 8. After executing the assignment at line 8, a second call to `printChar` is made (line 9), as illustrated below:

## STACK BEFORE 2ND CALL TO printChar

main's AR: 

ch: 'b'
return addr: system

## STACK WHEN printChar HAS BEEN CALLED

printChar's AR: 

c: 'b'
return addr: line 10

main's AR: 

ch: 'b'
return addr: system

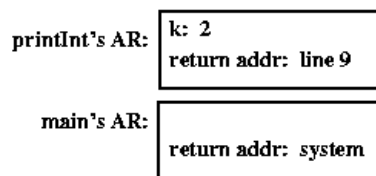
The two calls to `printChar` return to different places in `main` because different return addresses are stored in the ARs that are pushed onto the stack when the calls are made.

Now let's consider recursive calls:

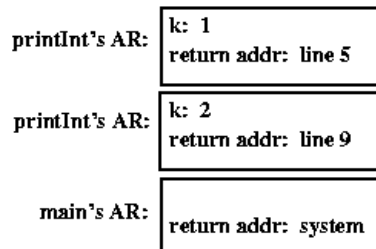
```
1. void printInt( int k ) {  
2.     if (k <= 0) return;  
3.     System.out.println( k );  
4.     printInt( k - 1 );  
5. }  
6.  
7. void main( ... ) {  
8.     printInt( 2 );  
9. }
```

The following pictures illustrate the runtime stack as this program executes.

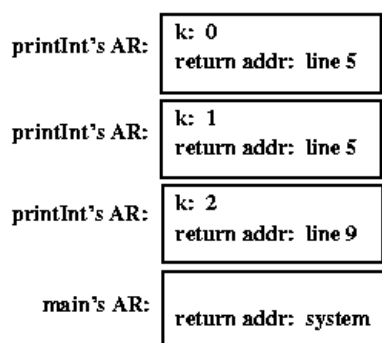
### STACK AFTER CALL TO printInt FROM main



### STACK AFTER 1ST RECURSIVE CALL



### STACK AFTER 2ND RECURSIVE CALL



Note that all of the recursive calls have the same return address -- after a recursive call returns, the previous call to `printInt` starts executing again at line 5. In this case, there is nothing more to do at that point, so the method would, in turn, return.

Note also that each call to `printInt` causes the current value of `k` to be printed, so the output of the program is: 2 1

Now consider a slightly different version of the `printInt` method:

```
1. void printInt(int k) {  
2.     if (k <= 0) return;  
3.     printInt(k - 1);  
4.     System.out.println(k);  
5. }
```

Now what is printed as a result of the call `printInt(2)`? Because the print statement comes *after* the recursive call, it is not executed until the recursive call finishes (i.e., `printInt`'s activation record will have line 4 -- the print statement -- as its return address, so that line will be executed only after the recursive call finishes). In this case, that means that the output is: 1 2 (instead of 2 1, as it was when the print statement came before the recursive call). This leads to the following insight:

### \*\*\* UNDERSTANDING RECURSION \*\*\*

Sometimes a method has more to do following a recursive call; it gets done only *after* the recursive call (and all calls it makes) are finished.

---

### TEST YOURSELF #2

```
void printTwoInts(int k) {  
    if (k == 0) {  
        return;  
    }  
    System.out.println("From before recursion: " + k);  
    printTwoInts(k - 1);  
    System.out.println("From after recursion: " + k);  
}
```

What is printed as a result of the call `printTwoInts(3)`?

[solution](#)

## Recursion vs Iteration

Now let's think about when it is a good idea to use recursion and why. In many cases there will be a choice: many methods can be written either with or without using recursion.

Q: Is the recursive version usually faster?

A: No -- it's usually slower (due to the overhead of maintaining the stack)

Q: Does the recursive version usually use less memory?

A: No -- it usually uses *more* memory (for the stack).

Q: Then *why* use recursion?

A: Sometimes it is much simpler to write the recursive version.

Here are a few examples where recursion makes things a little bit clearer, though in the second case, the efficiency problem makes it a bad choice.

## Factorial

Factorial can be defined as follows:

- factorial of 0 is 1
- factorial of N (for  $N > 0$ ) is  $N * (N-1) * \dots * 3 * 2 * 1$

or:

- factorial of 0 is 1
- factorial of N (for  $N > 0$ ) is  $N * \text{factorial}(N-1)$

(Note that factorial is undefined for negative numbers.)

The first definition leads to an **iterative** version of factorial:

```
int factorial(int N) {
    if (N == 0) {
        return 1;
    }
    int tmp = N;
    for (int k = N-1; k >= 1; k--) {
        tmp = tmp * k;
    }
    return (tmp);
}
```

The second definition leads to a **recursive** version:

```
int factorial(int N) {
    if (N == 0) {
        return 1;
    } else {
        return (N * factorial(N-1));
    }
}
```

The recursive version is

- a little shorter,
- a little clearer (closer to the mathematical definition),
- a little slower

Because the recursive version causes an activation record to be pushed onto the runtime stack for every call, it is also more limited than the iterative version (it will fail, with a "stack overflow" error), for large values of N.

---

## TEST YOURSELF #3

**Question 1:** Draw the runtime stack, showing the activation records that would be pushed as a result of the call `factorial(3)` (using the recursive version of factorial). Just show the value of N in each AR (don't worry about the return address). Also indicate the value that is *returned* as the result of each call.

**Question 2:** Recall that (mathematically) factorial is undefined for a negative number. What happens as a result of the call `factorial(-1)` for each of the two versions (iterative and recursive)? What do you think really should happen when the call `factorial(-1)` is made? How could you write the method to make that happen?

[solution](#)



## Fibonacci

Fibonacci can be defined as follows:

- fibonacci of 1 or 2 is 1
- fibonacci of  $N$  (for  $N > 2$ ) is fibonacci of  $(N-1)$  + fibonacci of  $(N-2)$

Fibonacci can be programmed either using iteration or using recursion. Here are the two versions (iterative first):

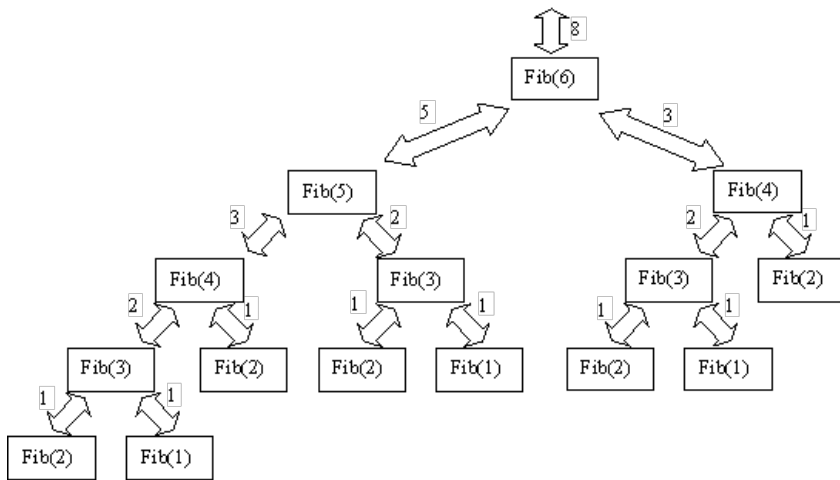
```
// iterative version
int fib( int N ) {
    int k1, k2, k3;
    k1 = k2 = k3 = 1;
    for (int j = 3; j <= N; j++) {
        k3 = k1 + k2;
        k1 = k2;
        k2 = k3;
    }
    return k3;
}

// recursive version
int fib( int N ) {
    if ((N == 1) || (N == 2)) {
        return 1;
    } else {
        return (fib( N-1 ) + fib( N-2 ));
    }
}
```

For fibonacci, the recursive version is:

- shorter,
- clearer,
- but *much* slower

Here's a picture of a **trace** of the recursive version of fibonacci, for the initial call `fib(6)`:



Note: one reason the recursive version is so slow is that it is repeating computations. For example, `fib(4)` is computed twice and `fib(3)` is computed 3 times.

Given that the recursive version of fibonacci is slower than the iterative version, is there a good reason for using it? The answer may be yes: because the recursive solution is so much simpler, it is likely to take much less time to write, debug, and maintain. If those costs (the cost for programming time) are more important than the cost of having a slow program, then the advantages of using the recursive solution outweigh the disadvantage and you should use it! If the speed of the final code is of vital importance, then you should not use the recursive fibonacci.

## Recursive Data Structures

So far, we've been doing all our recursion based on the value of some int parameter; e.g., for factorial, we start with some value  $N$  and each recursive call passes a smaller value  $(N - 1)$  until we get down to 0. A more common way to use recursion in programming is to base the recursion on a **recursive data structure**. A data structure is recursive if you can define it in terms of itself. For example, we can give a recursive definition of a string:

A string is either empty, or it is a character followed by a string.

We can give a similar, recursive definition of a linked list:

A linked list is either empty, or it is a listnode followed by a linked list.

Below are two methods that use recursive data structures; the first one prints each character of its `String` parameter and the second prints each value in its linked list parameter.

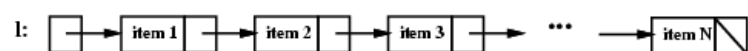
### Example 1: Use a `String` parameter

```
printStr( String S ) {
    if (S.equals("")) {
        return;
    }
    System.out.print( S.substring(0, 1) + " " );
    printStr( S.substring(1) );
}
```

### Example 2: Use a linked list parameter

```
printList( ListNode<E> l ) {
    if (l == null) {
        return;
    }
    System.out.println(l.getData());
    printList(l.getNext());
}
```

Now suppose we want to print the values in a linked list *backwards*; i.e., if the list looks like this:



we want to print:

```
itemN
itemN-1
...
item2
item1
```

It is easy to do this using recursion; all we have to do is swap the order of the print statement and the recursive call in the Example 2 code given above. Here's the new version:

```
printList(ListNode<E> llist) {
    if (llist == null) {
        return;
    }
    printList(llist.getNext());
    System.out.println(llist.getData());
}
```

In order to print the list backwards *without* using recursion, you'd have to use some auxiliary data structure. For example, you could traverse the list from left to right, pushing each value into a stack, then pop the stack, printing each value. However, it's much simpler just to use recursion!

---

## TEST YOURSELF #4

**Question 1:** Write a recursive method called `sum` with the following header:

```
public static int sum(ListNode<Integer> node)
```

Assume that `node` points to the first node in a linked list of `Integers`. The method `sum` should return the sum of the values in the list.

**Question 2:** Write a recursive method called `vowels` with the following header:

```
public static String vowels(String str)
```

The method `vowels` should return a `String` containing just the vowels (a, e, i, o, u) in `str`, in the order in which they occur in `str`. For example, `vowels("hooligan")` should return "ooia".

[solution](#)

---

## Analyzing Runtime for Recursive Methods

There are two ways to determine how much time is required for a call to a recursive method:

1. Informal Reasoning
2. Using Recurrence Equations

### Informal Reasoning

To reason about the time required for a call to a recursive method, you must figure out how many times total the method will be called and how much work (in terms of the problem size,  $N$ ) is done on each call in addition to the recursive calls. The total time is the product of those two values.

For example, consider recursive method `printInt`:

```
public static void printInt(int k) {
    if (k == 0) {
        return;
    }
    System.out.println( k );
    printInt( k - 1 );
}
```

It is easy to see that if `printInt` is first called with the value  $N$ , then a total of  $N+1$  calls will be made (for  $N$ ,  $N-1$ ,  $N-2$ , ...  $0$ ). Each time `printInt` is called, a constant amount of work is done (to check and print the value of  $k$ ) in addition to the recursive call. Therefore, the total time is  $O(N)$ .

## Using Recurrence Equations

A more formal way to analyze the running time of a recursive method is to use **recurrence equations**. The recurrence equations for a method include an equation for the base case (or several equations if there are several base cases) and an equation for the recursive case.

For example, for the `printInt` method above, the recurrence equations are:

$$T(0) = 1$$
$$T(N) = 1 + T(N-1)$$

The first equation says that a problem of size 0 (i.e., when parameter  $k = 0$ ) requires a constant amount of work (test  $k$  and return). The second equation says that a problem of size  $N$  (i.e., when parameter  $k$  is not 0) requires a constant amount of work (test and print  $k$ ) plus the time required for solving a problem of size  $N-1$ .

In general, the recurrence equation for the recursive case will be of the following form:

$$T(N) = \text{_____} + \text{_____} * T(\text{_____})$$

The first blank is the **amount of time outside the recursive call(s)**. In the `printInt` method above, that is constant, so we fill in the first blank with a 1.

The second blank is the **number of recursive calls**. Not the total number of recursive calls that will be made, just the number made here. In the `printInt` method above, there is just one recursive call, so we fill in the second blank with a 1.

The third blank is the **problem size passed to the recursive call**. It better be less than the current problem size, or the code will not make progress toward the base case! In our example, the problem size (the value of  $k$ ) decreases by one, so we fill in the third blank with  $N-1$ .

The final equation for the recursive case is

$$T(N) = 1 + 1 * T(N-1)$$

Which is the same as the equation given above (just with an explicit "times 1"). Now consider a different version of `printInt`:

```
public static void printInt(int k) {
    if (k == 0) {
        return;
    }
    printInt(k/2);
    for (int j = 0; j < k; j++) {
        System.out.println(j);
    }
    printInt(k/2);
}
```

For this version, the amount of time outside of the recursive calls is proportional to the problem size, there are **two** recursive calls, and the problem size passed to each of those calls is half the original problem size. Therefore, the recurrence equation for this new version of `printInt` is

$$T(N) = N + 2 * T(N/2)$$

Now let's consider how to solve recurrence equations. To find the worst-case time for a problem of size  $N$ , we need to find a **closed-form solution** to these equations, i.e., a solution for  $T(N)$  that doesn't involve any  $T(\dots)$  on the right-hand side. To do that we perform three steps:

1. Expand: find the times required for a problem of size 1, 2, 3, etc.
2. Look for a pattern and guess a solution for  $T(N)$ .
3. Plug in your solution and see if you got it right.

Below are those three steps for the original `printInt` equations, which were:

$$T(0) = 1$$
$$T(N) = 1 + T(N-1)$$

## Step 1: Expand

$$T(0) = 1$$

$$T(1) = 1 + T(0) = 1 + 1 = 2$$

$$T(2) = 1 + T(1) = 1 + 2 = 3$$

$$T(3) = 1 + T(2) = 1 + 3 = 4$$

## Step 2: Look for a pattern and guess a solution

We'll guess that  $T(N) = N+1$

## Step 3: Verify the guessed solution

$$T(N) = 1 + T(N-1) \quad // \text{ given}$$

$$N+1 \stackrel{?}{=} 1 + ((N-1)+1) \quad // \text{ replace } T(\dots) \text{ on both sides with the guessed solution } (N+1) \text{ in which each } N \text{ is replaced by } \dots$$

$$N+1 \stackrel{?}{=} 1 + N - 1 + 1 \quad // \text{ simplify}$$

$$N+1 = 1 + N \quad // \text{ yes the two sides are equal!}$$

---

## TEST YOURSELF #5

Suppose we have the following recurrence equations:

$$T(1) = 1$$

$$T(N) = 1 + 2 * T(N/2)$$

Do the steps necessary to solve those equations. First, fill in the following table:

N	T(N)
1	
2	
4	
8	

Next, look for a pattern and guess a solution.

Finally, verify your solution using the recursive equation.

[solution](#)

---

## Using Mathematical Induction to Prove the Correctness of Recursive Code

If you like math, you may enjoy thinking about how to use mathematical induction to prove that recursive code is correct. If not, feel free to skip this section.

When using induction to prove a theorem, you need to show:

1. that the base case (usually  $n=0$  or  $n=1$ ) is true
2. that case  $k$  implies case  $k+1$

It is sometimes straightforward to use induction to prove that recursive code is correct. Let's consider how to do that for the recursive version of factorial. First we need to prove the base case:  $\text{factorial}(0) = 0!$  (which, by definition is 1). The correctness of the factorial method for  $N=0$  is obvious from the code: when  $N=0$  it returns 1.

Now we need to show that if  $\text{factorial}(k) = k!$ , then  $\text{factorial}(k+1) = (k+1)!$ .

Looking at the code we see that for  $N \neq 0$ ,  $\text{factorial}(N) = (N) * \text{factorial}(N-1)$ . So  $\text{factorial}(k+1) = (k+1) * \text{factorial}(k)$ .

By assumption,  $\text{factorial}(k) = k!$ , so  $\text{factorial}(k+1) = (k+1) * (k!)$ .

By definition,  $k! = k * (k-1) * (k-2) * \dots * 3 * 2 * 1$ . So  $(k+1) * (k!) = (k+1) * k * (k-1) * (k-2) * \dots * 3 * 2 * 1$ , which is (by definition)  $(k+1)!$ , and the proof is done.

Note that we've shown that the factorial method is correct for all values of  $N$  greater than or equal to zero (because our base case was  $N=0$ ). We haven't shown anything about factorial for negative numbers.

## Summary

- Use recursion for *clarity* and (sometimes) for a reduction in the time needed to write and debug code, not for space savings or speed of execution.
- Remember that every recursive method must have a **base case** (rule #1).
- Also remember that every recursive method must *make progress* towards its base case (rule #2).
- Sometimes a recursive method has *more to do* following a recursive call. It gets done only *after* the recursive call (and all calls it makes) finishes.
- Use **informal reasoning** or **recurrence equations** to determine how much time a recursive call will require.
- Recursion is often simple and elegant, can be efficient, and tends to be underutilized. Consider using recursion when solving a problem!

# Searching

---

## Contents

- [Introduction](#)
  - [Searching in an Array](#)
  - [The Comparable Interface](#)
- 

## Introduction

Searching is a very common problem, both in life and in computer applications. It is so important that many data structures and algorithms have been designed specifically to make searching as easy and efficient as possible.

In this set of notes we will review the three techniques for searching for a value in an array. Other sets of notes will present several different data structures designed to support efficient searches.

## Searching in an Array

Recall that there are two basic approaches to searching for a given value `val` in an array: **sequential search** and **binary search**.

**Sequential search** involves looking at each value in turn (i.e., start with the value in `array[0]`, then `array[1]`, etc.). The algorithm quits and returns true if the current value is `val`; it quits and returns false if it has looked at all of the values in the array without finding `val`.

If the values are in **sorted** order, then the algorithm can sometimes quit and return false without having to look at all of the values in the array; in particular, if the current value is **greater** than `val` then there is no point in looking at the rest of the values in the array (`val` is definitely not there).

The worst-case time for a sequential search in an array of size  $N$  is always  $O(N)$ , even when the array is sorted (though the average-case time should be better for a sorted array than for a non-sorted array).

When the values are in sorted order, a better approach than sequential search is to use **binary search**. The algorithm for binary search starts by looking at the middle item `mid`. If `mid` is the value `val` that we're searching for, the algorithm quits and returns true. Otherwise, it uses the relative ordering of `mid` and `val` to eliminate half of the array (if `val` is less than `mid`, then it can't be to the right of `mid` in the array; similarly, if it is greater than `mid`, it can't be to the left of `mid`). Once half of the array has been eliminated, the algorithm starts again by looking at the middle item in the remaining half. It quits when it finds `val` or when the entire array has been eliminated.

The worst-case time for binary search in an array of size  $N$  is proportional to  $\log_2 N$  (pronounced "log base 2 of  $N$ "). The log base 2 of  $N$  is the number of times  $N$  can be divided in half before there is nothing left. Similarly, the log base 3 of  $N$  would be the number of times  $N$  can be divided in thirds before there is nothing left. Using Big-O notation, log base anything of  $N$  is written  $O(\log N)$ , since the difference between "log base 2 of  $N$ " and "log base some other number of  $N$ " is just a constant factor.

## The Comparable Interface

As discussed above, binary search only works if the values in the array are in sorted order. Values can be arranged in sorted order only if, given two values  $a$  and  $b$ , we can answer the question: Is  $a$  less than  $b$ ? That question can be answered for all numeric values (e.g., `int`, `float`, `double`); we can compare two numeric variables  $x$  and  $y$  using the "less-than" operator: `if (x < y) ...`

What about Java Objects? Since `Integers` and `Doubles` represent numeric values, it makes sense to be able to compare them; similarly, there is a natural ordering on strings, so it makes sense to be able to compare two `String` objects, too. For `Integers` and `Doubles`, you could convert them to the corresponding `int` / `double` values and use the less-than operator, but Java gives you an easier option (one that also works for `Strings`): `Integer`s, `Double`s, and `String`s all implement the `Comparable` interface, which means that they have a `compareTo` method. The `compareTo` method allows you to compare two `Integer`s, two `Double`s, or two `String`s to see if one is less than, equal to, or greater than the other. The `compareTo` method returns a negative value to mean "less than", it returns zero to mean "equal to", and it returns a positive value to mean "greater than".

For example, if variables `S1` and `S2` are both `Strings`, then `S1.compareTo(S2)` returns an `int` value that is:

- negative if `S1` is less than `S2`;
- zero if `S1` is equal to `S2`;
- positive if `S1` is greater than `S2`.

When you define a new class you should think about whether it makes sense to ask whether one instance of the class is less than another instance. If the answer is yes, you should define a `compareTo` method and you should make your class implement the `Comparable` interface. If you do that, it will be possible to pass instances of your class to methods that require `Comparable` parameters (for example, Java provides some sorting methods that require `Comparable` parameters).

To make your class implement the `Comparable` interface you must:

1. include `implements Comparable<C>` after the class name in the file that defines the class, and
2. define a `compareTo` method with the following signature:

```
int compareTo(C other)
```

For example, we could define a `Name` class with two fields for the first and last names. It makes sense to consider one `Name` smaller than another if the first one would come first in the phonebook (i.e., its last name comes first in alphabetical order, or the last names are the same and its first name comes first in alphabetical order). Here's a (partial) definition of the `Name` class, including just the fields and the `compareTo` method:

```
public class Name implements Comparable<Name> {
    private String firstName, lastName;

    public int compareTo(Name other) {
        int tmp = lastName.compareTo(other.lastName);
        if (tmp != 0) {
            return tmp;
        }
        return firstName.compareTo(other.firstName);
    }
}
```

# Introduction to Trees

## Contents

- [Introduction: Trees and Binary Trees](#)
- [Representing Trees](#)
  - [Test Yourself #1](#)
- [Tree Traversals](#)
  - [Pre-order](#)
  - [Post-order](#)
  - [Level-order](#)
  - [Test Yourself #2](#)
  - [In-order](#)
  - [Test Yourself #3](#)

## Introduction

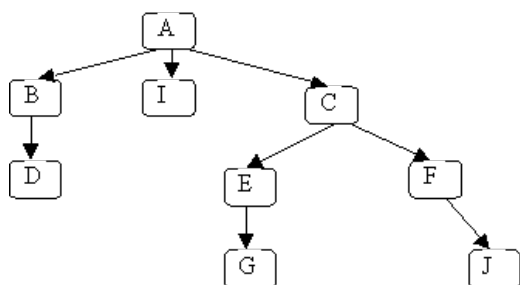
Lists, stacks, and queues are all **linear** structures: in all three data structures, one item follows another. Trees will be our first non-linear structure:

- More than one item can follow another.
- The number of items that follow can vary from one item to another.

Trees have many uses:

- representing family genealogies
- as the underlying structure in decision-making algorithms
- to represent priority queues (a special kind of tree called a **heap**)
- to provide fast access to information in a database (a special kind of tree called a **b-tree**)

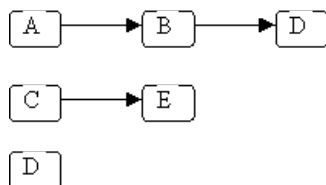
Here is the conceptual picture of a tree (of letters):



- each letter represents one **node**
- the arrows from one node to another are called **edges**
- the topmost node (with no incoming edges) is the **root** (node A)
- the bottom nodes (with no outgoing edges) are the **leaves** (nodes D, I, G & J)

So a (computer science) tree is kind of like an upside-down real tree.

A **path** in a tree is a sequence of (zero or more) connected nodes; for example, here are 3 of the paths in the tree shown above:



The **length** of a path is the number of nodes in the path, e.g.:

<u>Path</u>	<u>Length</u>
A → B → D	3
C → E	2
D	1

The **height** of a tree is the length of the longest path from the root to a leaf; for the above example, the height is 4 (because the longest path from the root to a



leaf is  $A \rightarrow C \rightarrow E \rightarrow G$ , or  $A \rightarrow C \rightarrow E \rightarrow J$ ). An empty tree has height = 0.

The **depth** of a node is the length of the path from the root to that node; for the above example:

- the depth of J is 4
- the depth of D is 3
- the depth of A is 1

Given two connected nodes like this:



Node A is called the **parent**, and node B is called the **child**.

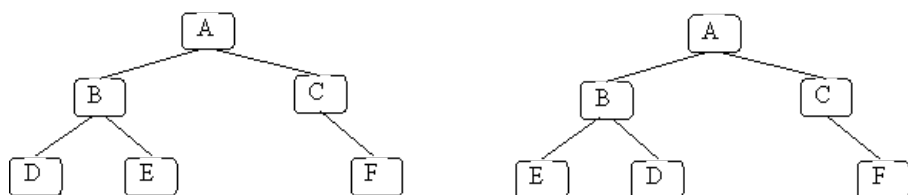
A **subtree** of a given node includes one of its children and all of that child's **descendants**. The descendants of a node N are all nodes reachable from N (N's children, its children's children, etc.). In the original example, node A has three subtrees:

1. B, D
2. I
3. C, E, F, G, J

An important special kind of tree is the **binary** tree. In a binary tree:

- Each node has 0, 1, or 2 children.
- Each child is either a left child or a right child.

Here are two examples of binary trees that are different:



The two trees are different because the children of node B are different: in the first tree, B's left child is D and its right child is E; in the second tree, B's left child is E and its right child is D. Also note that lines are used instead of arrows. We sometimes do this because it is clear that the edge goes from the higher node to the lower node.

## Representing Trees

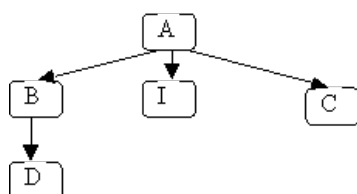
Since a binary-tree node never has more than two children, a node can be represented using a class with 3 fields: one for the data in the node plus two child pointers:

```
class BinaryTreeNode<T> {  
    // *** fields ***  
    private T data;  
    private BinaryTreeNode<T> leftChild;  
    private BinaryTreeNode<T> rightChild;  
    ...  
}
```

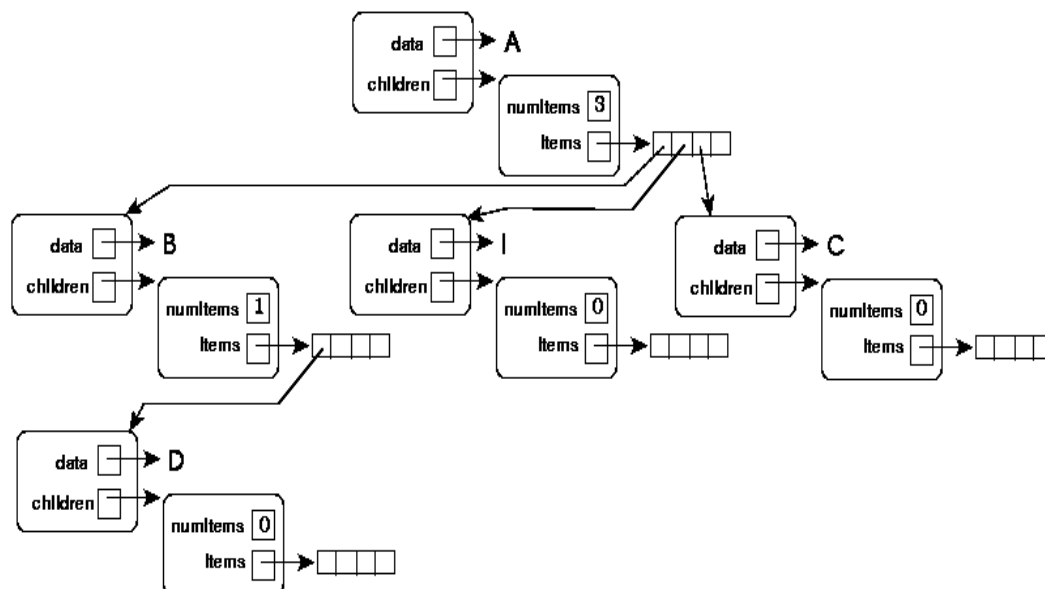
However, since a general-tree node can have an arbitrary number of children, a fixed number of child-pointer fields won't work. Instead, we can use a List to keep all of the child pointers:

```
class Treenode<T> {  
    // *** fields ***  
    private T data;  
    private ListADT<Treenode<T>> children;  
    ...  
}
```

As we know, a list can be represented using either an array or a linked-list. For example, consider this general tree (a simplified version of the original example):



For the array representation of the List (where the array has an initial size of 4) we would have:



### TEST YOURSELF #1

Draw a similar picture of the tree when the List fields are implemented using linked lists.

[solution](#)

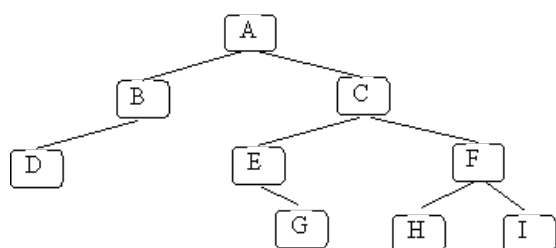
## Tree Traversals

It is often useful to iterate through the nodes in a tree:

- to print all values
- to determine if there is a node with some property
- to make a copy

When we iterate through a List, we started with the first tree node and visited each node in turn. Since each node is visited, the best possible complexity is  $O(N)$  for a tree with  $N$  nodes. All of our traversal methods will achieve this complexity.

For trees, there are many different orders in which we might visit the nodes. There are three common traversal orders for general trees and one more for binary trees: pre-order, post-order, level-order, and in-order, all described below. We will use the following tree to illustrate each traversal:



### Pre-order

A pre-order traversal can be defined (recursively) as follows:

1. **visit the root**
2. perform a pre-order traversal of the first subtree of the root
3. perform a pre-order traversal of the second subtree of the root
4. etc., for all the subtrees of the root

If we use a pre-order traversal on the example tree given above and we print the letter in each node when we visit that node, the following will be printed: A B D C E G F H I.

### Post-order

A post-order traversal is similar to a pre-order traversal, except that the root of each subtree is visited **last** rather than first:

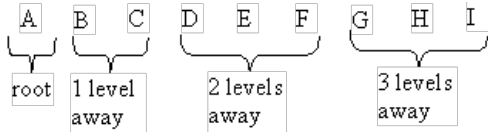
1. perform a postorder traversal of the first subtree of the root
2. perform a postorder traversal of the second subtree of the root

3. etc., for all the subtrees of the root
4. **visit the root**

If we use a post-order traversal on the example tree given above and we print the letter in each node when we visit that node, the following will be printed: D B G E H I F C A.

## Level-order

The idea of a level-order traversal is to visit the root, then visit all nodes "1 level away" (depth 2) from the root (left to right), then all nodes "2 levels away" (depth 3) from the root, etc. For the example tree, the goal is to visit the nodes in the following order:



A level-order traversal requires using a queue (rather than a recursive algorithm, which implicitly uses a stack). Here's how to print the data in a tree in level order, using a queue `Q`, and using an iterator to access the children of each node (we assume that the root node is called `root` and that the `Treenode` class provides a `getChildren` method):

```
Q.enqueue(root)
while (!Q.empty()) {
    Treenode<T> n = Q.dequeue();
    System.out.print(n.getData());
    ListADT<Treenode<T>> kids = n.getChildren();
    Iterator<Treenode<T>> it = kids.iterator();
    while (it.hasNext()) {
        Q.enqueue(it.next());
    }
}
```

## TEST YOURSELF #2

Draw pictures of `Q` as it would be each time around the outer while loop in the code given above for the [example tree](#) given above.

[solution](#)

## In-order

An in-order traversal involves visiting the root "in between" visiting its left and right subtrees. Therefore, an in-order traversal only makes sense for binary trees. The (recursive) definition is:

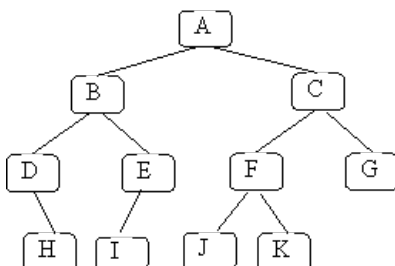
1. perform an in-order traversal of the left subtree of the root
2. **visit the root**
3. perform an in-order traversal of the right subtree of the root

If we print the letters in the nodes of our example tree using an in-order traversal, the following will be printed: D B A E G C H F I

The primary difference between the pre-order, post-order, and in-order traversals is where the node is visited in relation to the recursive calls; i.e., before, after, or in-between.

## TEST YOURSELF #3

What is printed when the following tree is visited using (a) a pre-order traversal, (b) a post-order traversal, (c) a level-order traversal, and (d) an in-order traversal?



[solution](#)

# Binary Search Trees

## Contents

- [Introduction](#)
  - [Test Yourself #1](#)
- [Implementing Binary Search Trees](#)
  - [The lookup method](#)
  - [The insert method](#)
    - [Test Yourself #2](#)
  - [The delete method](#)
    - [Test Yourself #3](#)
- [Maps and Sets](#)
- [Summary](#)

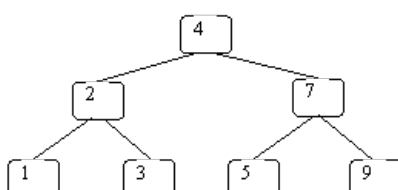
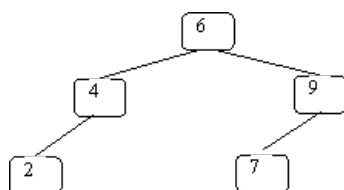
## Introduction

An important special kind of binary tree is the **binary search tree (BST)**. In a BST, each node stores some information including a unique **key value** and perhaps some associated data. A binary tree is a BST iff, for every node  $n$ , in the tree:

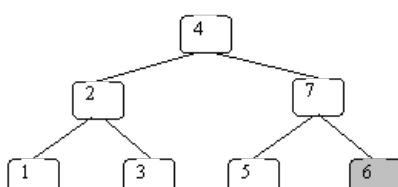
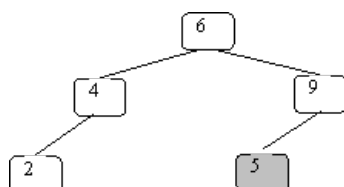
- All keys in  $n$ 's left subtree are less than the key in  $n$ , and
- all keys in  $n$ 's right subtree are greater than the key in  $n$ .

Note: if duplicate keys are allowed, then nodes with values that are equal to the key in node  $n$  can be either in  $n$ 's left subtree or in its right subtree (but not both). In these notes, we will assume that duplicates are not allowed.

Here are some BSTs in which each node just stores an integer key:

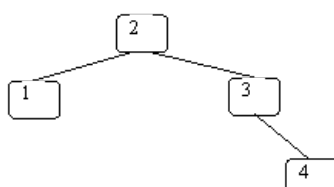
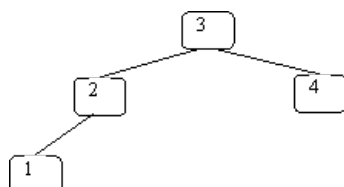


These are not BSTs:



In the left one 5 is not greater than 6. In the right one 6 is not greater than 7.

Note that more than one BST can be used to store the same set of key values. For example, both of the following are BSTs that store the same set of integer keys:

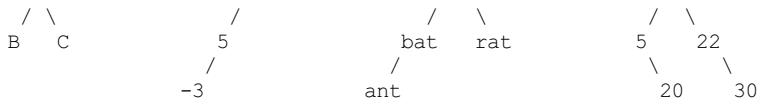


The reason binary-search trees are important is that the following operations can be implemented efficiently using a BST:

- insert a key value
- determine whether a key value is in the tree
- remove a key value from the tree
- print all of the key values in sorted order

## TEST YOURSELF #1

**Question 1:** Which of the following binary trees are BSTs? If a tree is *not* a BST, say why.



**Question 2:** Using which kind of traversal (pre-order, post-order, in-order, or level-order) visits the nodes of a BST in sorted order?

[solution](#)

## Implementing BSTs

To implement a binary search tree, we will use two classes: one for the individual tree nodes, and one for the BST itself. The following class definitions assume that the BST will store only key values, no associated data.

```

class BSTNode<K> {
    // *** fields ***
    private K key;
    private BSTNode<K> left, right;

    // *** constructor ***
    public BSTNode(K key, BSTNode<K> left, BSTNode<K> right) {
        this.key = key;
        this.left = left;
        this.right = right;
    }

    // *** methods ***

    // accessors (access to fields)
    public K getKey() { return key; }
    public BSTNode<K> getLeft() { return left; }
    public BSTNode<K> getRight() { return right; }

    // mutators (change fields)
    public void setKey(K newK) { key = newK; }
    public void setLeft(BSTNode<K> newL) { left = newL; }
    public void setRight(BSTNode<K> newR) { right = newR; }
}

public class BST<K extends Comparable<K>> {
    // *** fields ***
    private BSTNode<K> root; // ptr to the root of the BST

    // *** constructor ***
    public BST() { root = null; }

    // *** methods ***

    public void insert(K key) throws DuplicateException { ... }
    // add key to this BST; error if it is already there

    public void delete(K key) { ... }
    // remove the node containing key from this BST if it is there;
    // otherwise, do nothing

    public boolean lookup(K key) { ... }
    // if key is in this BST, return true; otherwise, return false

    public void print(PrintStream p) { ... }
    // print the values in this BST in sorted order (to p)
}

```

The type parameter `K` is the type of the key. Because most of the BST operations require comparing key values, we declare that `K` extends `Comparable<K>`, meaning that type `K` must implement a method

```
int compareTo(K other)
```

To implement a BST that stores some data with each key, we would use the following class definitions (changes are in red):

```

class BSTNode<K, V> {
    // *** fields ***
    private K key;
    private V value;
    private BSTNode<K, V> left, right;

    // *** constructor ***
    public BSTNode(K key, V value,
                  BSTNode<K, V> left, BSTNode<K, V> right) {
        this.key = key;
        this.value = value;
    }
}

```

```

        this.left = left;
        this.right = right;
    }

    // *** methods ***

    ...
    public V getValue() { return value; }
    public void setValue(V newV) { value = newV; }
    ...
}

public class BST<K extends Comparable<K>, V> {
    // *** fields ***
    private BSTNode<K, V> root; // ptr to the root of the BST

    // *** constructor ***
    public BST() { root = null; }

    // *** methods ***

    public void insert(K key, V value) throws DuplicateException {...}
    // add key and associated value to this BST;
    // error if key is already there

    public void delete(K key) {...}
    // remove the node containing key from this BST if it is there;
    // otherwise, do nothing

    public V lookup(K key) {...}
    // if key is in this BST, return its associated value; otherwise, return null

    public void print(PrintStream p) {...}
    // print the values in this BST in sorted order (to p)
}

```

From now on, we will assume that BSTs only store key values, *not* associated data. We will also assume that null is not a valid key value (i.e., if someone tries to insert or lookup a null value, that should cause an exception).

## The lookup method

In general, to determine whether a given value is in the BST, we will start at the root of the tree and determine whether the value we are looking for:

- is in the root
- might be in the root's left subtree
- might be in the root's right subtree

There are actually *two* base cases:

1. The tree is empty; return false.
2. The value is in the root node; return true.

If neither base case holds, a recursive lookup is done on the appropriate subtree. Since all values less than the root's value are in the left subtree and all values greater than the root's value are in the right subtree, there is no point in looking in *both* subtrees: if the value we're looking for is less than the value in the root, it can only be in the left subtree (and if it is greater than the value in the root, it can only be in the right subtree).

The code for the lookup method uses an auxiliary, recursive method with the same name (i.e., the lookup method is overloaded):

```

public boolean lookup(K key) {
    return lookup(root, key);
}

private static boolean lookup(BSTNode<K> n, K key) {
    if (n == null) {
        return false;
    }

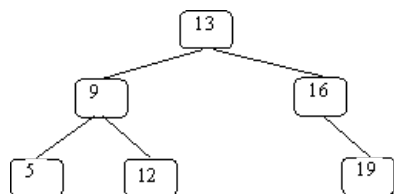
    if (n.getKey().equals(key)) {
        return true;
    }

    if (key.compareTo(n.getKey()) < 0) {
        // key < this node's key; look in left subtree
        return lookup(n.getLeft(), key);
    }

    else {
        // key > this node's key; look in right subtree
        return lookup(n.getRight(), key);
    }
}

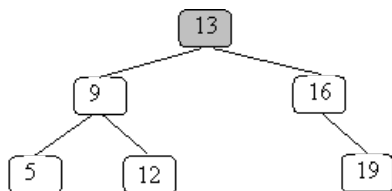
```

Let's illustrate what happens using the following BST:

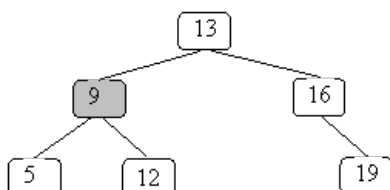


and searching for 12:

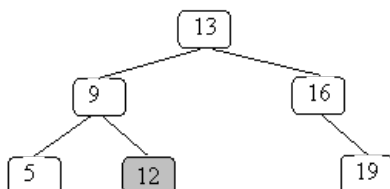
$12 < 13$  so go to left subtree



$12 > 9$  so go to right subtree

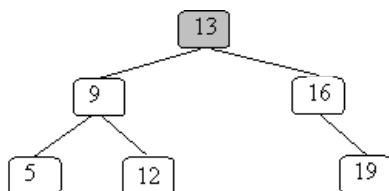


found!

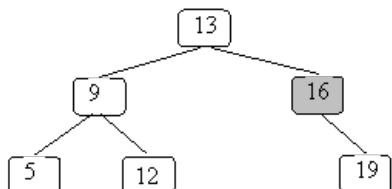


What if we search for 15:

$15 > 13$  so go to right subtree

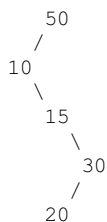


$15 < 16$  so go to left subtree. It does not exist so search fails and it returns null



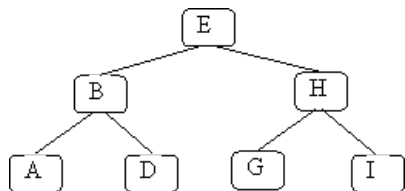
How much time does it take to search for a value in a BST? Note that lookup always follows a path from the root down towards a leaf. In the worst case, it goes all the way to a leaf. Therefore, the worst-case time is proportional to the length of the longest path from the root to a leaf (the height of the tree).

In general, we'd like to know how much time is required for lookup as a function of the number of values stored in the tree. In other words, what is the relationship between the number of nodes in a BST and the height of the tree? This depends on the "shape" of the tree. In the worst case, all nodes have just one child, and the tree is essentially a linked list. For example:



This tree has 5 nodes and also has height = 5. Searching for values in the range 16-19 and 21-29 will require following the path from the root down to the leaf (the node containing the value 20), i.e., will require time proportional to the number of nodes in the tree.

In the best case, all nodes have 2 children and all leaves are at the same depth, for example:



This tree has 7 nodes and height = 3. In general, a tree like this (a **full** tree) will have height approximately  $\log_2(N)$ , where N is the number of nodes in the tree. The value  $\log_2(N)$  is (roughly) the number of times you can divide N by two before you get to zero. For example:

```

7/2 = 3      // divide by 2 once
3/2 = 1      // divide by 2 a second time
1/2 = 0      // divide by 2 a third time, the result is zero so quit
  
```

So  $\log_2(7)$  is approximately equal to 3.

The reason we use  $\log_2$  (rather than say  $\log_3$ ) is because every non-leaf node in a full BST has **two** children. The number of nodes in each of the root's subtrees is (approximately) 1/2 of the nodes in the whole tree, so the length of a path from the root to a leaf will be the same as the number of times we can divide N (the total number of nodes) by 2.

However, when we use big-O notation, we just say that the height of a full tree with N nodes is  $O(\log N)$  -- we drop the "2" subscript, because  $\log_2(N)$  is proportional to  $\log_k(N)$  for any constant k, i.e., for any constants B and k and any value N:

$$\log_B(N) = \log_k(N) / \log_k(B)$$

and with big-O notation we always ignore constant factors.

To summarize: the worst-case time required to do a lookup in a BST is  $O(\text{height of tree})$ . In the worst case (a "linear" tree) this is  $O(N)$ , where N is the number of nodes in the tree. In the best case (a "full" tree) this is  $O(\log N)$ .

## The insert method

Where should a new item go in a BST? The answer is easy: it needs to go where you would have found it using lookup! If you don't put it there then you won't find it later.

The code for insert is given below. Note that:

- We assume that duplicates are not allowed (an attempt to insert a duplicate value causes an exception).
- The public insert method uses an auxiliary recursive "helper" method to do the actual insertion.
- The node containing the new value is always inserted as a **leaf** in the BST.
- The public insert method returns `void`, but the helper method returns a `BSTNode`. It does this to handle the case when the node passed to it is null. In general, the helper method is passed a pointer to a possibly empty tree. Its responsibility is to add the indicated key to that tree and return a pointer to the root of the modified tree. If the original tree is empty, the result is a one-node tree. Otherwise, the result is a pointer to the same node that was passed as an argument.

```

public void insert(K key) throws DuplicateException {
    root = insert(root, key);
}
  
```

```

//JIM CORRECTED 4/8/13
private static BSTNode<K> insert(BSTNode<K> n, K key) throws DuplicateException {
    if (n == null) {
        return new BSTNode<K>(key, null, null);
    }

    if (n.getKey().equals(key)) {
        throw new DuplicateException();
    }

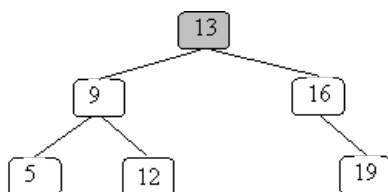
    if (key.compareTo(n.getKey()) < 0) {
        // add key to the left subtree
        n.setLeft( insert(n.getLeft(), key) );
        return n;
    }

    else {
        // add key to the right subtree
        n.setRight( insert(n.getRight(), key) );
        return n;
    }
}
  
```

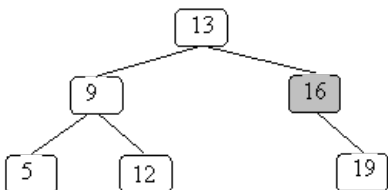
Here are pictures illustrating what happens when we insert the value 15 into the example tree used above.



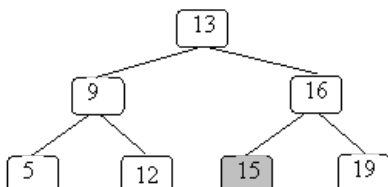
15 > 13 so go to right subtree



15 < 16 and no left subtree



So insert 15 as left child



It is easy to see that the complexity for insert is the same as for lookup: in the worst case, a path is followed all the way to a leaf.

## TEST YOURSELF #2

As mentioned above, the order in which values are inserted determines what BST is built (inserting the same values in different orders can result in different final BSTs). Draw the BST that results from inserting the values 1 to 7 in each of the following orders (reading from left to right):

1. 5 3 7 6 2 1 4
2. 1 2 3 4 5 6 7
3. 4 3 5 2 6 1 7

[solution](#)

## The delete method

As you would expect, deleting an item involves a search to locate the node that contains the value to be deleted. Here is an outline of the code for the delete method.

```
public void delete(K key) {
    root = delete(root, key);
}

private static BSTNode<K> delete(BSTNode<K> n, K key) {
    if (n == null) {
        return null;
    }

    if (key.equals(n.getKey())) {
        // n is the node to be removed
        // code must be added here
    }

    else if (key.compareTo(n.getKey()) < 0) {
        n.setLeft( delete(n.getLeft(), key) );
        return n;
    }

    else {
        n.setRight( delete(n.getRight(), key) );
        return n;
    }
}
```

There are several things to note about this code:

- As for the lookup and insert methods, the BST delete method uses an auxiliary, overloaded delete method to do the actual work.
- If  $k$  is not in the tree, then eventually the auxiliary method will be called with  $n == \text{null}$ . That is not considered an error; the tree is simply unchanged in that case.
- The auxiliary delete method returns a value (a pointer to the updated tree). The reason for this is explained below.

If the search for the node containing the value to be deleted succeeds, there are three cases to deal with:

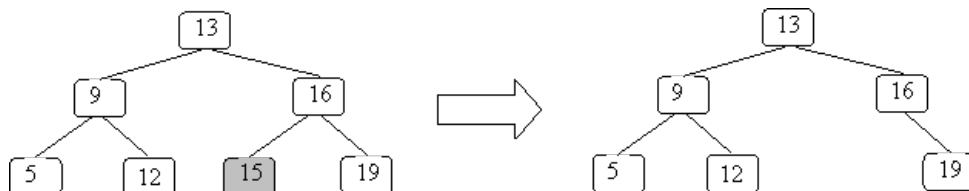
1. The node to delete is a leaf (has no children).
2. The node to delete has one child.
3. The node to delete has two children.

When the node to delete is a leaf, we want to remove it from the BST by setting the appropriate child pointer of its parent to null (or by setting root to null if the node to be deleted is the root and it has no children). Note that the call to delete was one of the following:

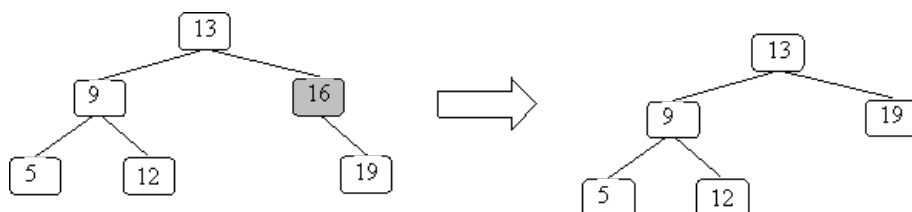
- `root = delete(root, key);`
- `n.setLeft( delete(n.getLeft(), key) );`
- `n.setRight( delete(n.getRight(), key) );`

So in all three cases, the right thing happens if the delete method just returns null.

Here's what happens when the node containing the value 15 is removed from the example BST:



When the node to delete has one child, we can simply replace that node with its child by returning a pointer to that child. As an example, let's delete 16 from the BST just formed:



Here's the code for delete, handling the two cases we've discussed so far (the new code is shown in red):

```
private static BSTNode<K> delete(BSTNode<K> n, K key) {
    if (n == null) {
        return null;
    }

    if (key.equals(n.getKey())) {
        // n is the node to be removed
        if (n.getLeft() == null && n.getRight() == null) {
            return null;
        }
        if (n.getLeft() == null) {
            return n.getRight();
        }
        if (n.getRight() == null) {
            return n.getLeft();
        }

        // if we get here, then n has 2 children
        // code still needs to be added here...
    }

    else if (key.compareTo(n.getKey()) < 0) {
        n.setLeft( delete(n.getLeft(), key) );
        return n;
    }

    else {
        n.setRight( delete(n.getRight(), key) );
        return n;
    }
}
```

The hard case is when the node to delete has two children. We'll call the node to delete *n*. We can't replace node *n* with one of its children, because what would we do with the other child? Instead, we will replace the key in node *n* with the key value *v* from another node lower down in the tree, then (recursively) delete value *v*.

The question is what value can we use to replace *n*'s key? We have to choose that value so that the tree is still a BST, i.e., so that all of the values in *n*'s left subtree are less than the value in *n*, and all of the values in *n*'s right subtree are greater than the value in *n*. There are two possibilities that work: the **largest** value in *n*'s left subtree or the **smallest** value in *n*'s right subtree. We'll arbitrarily decide to use the smallest value in the right subtree.

To find that value, we just follow a path in the right subtree, always going to the **left** child, since smaller values are in left subtrees. Once the value is found, we copy it into node *n*, then we recursively delete that value from *n*'s right subtree. Here's the final version of the delete method:

```

private static BSTNode<K> delete(BSTNode<K> n, K key) {
    if (n == null) {
        return null;
    }

    if (key.equals(n.getKey())) {
        // n is the node to be removed
        if (n.getLeft() == null && n.getRight() == null) {
            return null;
        }
        if (n.getLeft() == null) {
            return n.getRight();
        }
        if (n.getRight() == null) {
            return n.getLeft();
        }

        // if we get here, then n has 2 children
        K smallVal = smallest(n.getRight());
        n.setKey(smallVal);
        n.setRight( delete(n.getRight(), smallVal) );
        return n;
    }

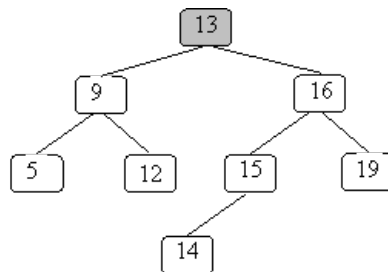
    else if (key.compareTo(n.getKey()) < 0) {
        n.setLeft( delete(n.getLeft(), key) );
        return n;
    }

    else {
        n.setRight( delete(n.getRight(), key) );
        return n;
    }
}

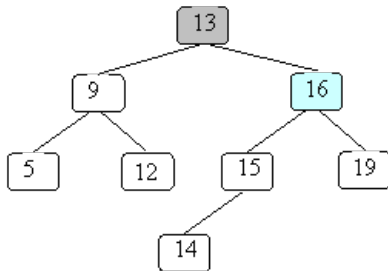
```

Below is a slightly different example BST; let's see what happens when we delete 13 from that tree.

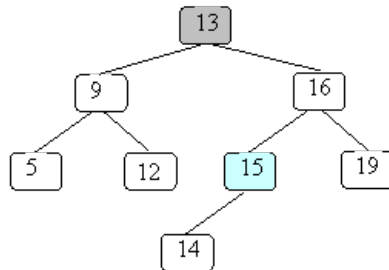
Original BST with 13 located



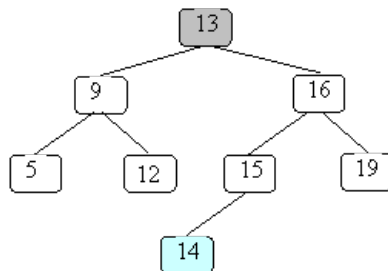
Step into right subtree.



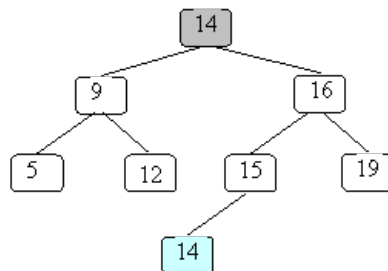
Go to left child.



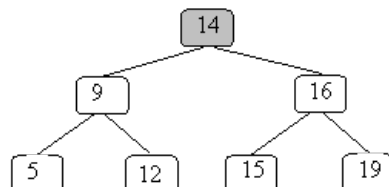
Continue to left child. This is last one.



Replace node to delete with far left child of right subtree.



Remove far left child of right subtree.



### TEST YOURSELF #3

Write the auxiliary method `smallest` used by the delete method given above. The header for `smallest` is:

```
private static K smallest(BSTNode<K> n)
// precondition: n is not null
// postcondition: return the smallest value in the subtree rooted at n
```

[solution](#)

What is the complexity of the BST delete method?

If the node to be deleted has zero or one child, then the delete method will "follow a path" from the root to that node. So the worst-case time is proportional to

the height of the tree (just like for lookup and insert).

If the node to be deleted has two children, the following steps are performed:

1. Find the node to be deleted (follow the path from the root to that node).
2. Find the smallest value  $v$  in the right subtree (continue down the path toward a leaf).
3. Recursively delete value  $v$  (follow the same path followed in step 2).

So in the worst case, a path from the root to a leaf is followed twice. Since we don't care about constant factors, the time is still proportional to the height of the tree.

## Maps and Sets

The Java standard library has built into it an industrial-strength version of binary search trees, so if you are programming in Java and need this functionality, you would be better off using the library version rather than writing your own. The class that most closely matches the outline above, in which the nodes contain only keys and no other data, is called [TreeSet](#). Class `TreeSet` is an implementation of the [Set](#) interface. (There is another implementation, called [HashSet](#), that we will study later in this course.) Here's an example of how you might use a `Set` to implement a simple spell-checker.

```
Set<String> dictionary = new TreeSet<String>();
Set<String> misspelled = new TreeSet<String>();
// Create a set of "good" words.
while (...) {
    String goodWord = ...;
    dictionary.add(goodWord);
}

// Look up various other words
while (...) {
    String word = ...;
    if (! dictionary.contains(word)) {
        misspelled.add(word);
    }
}

// Print a list

if (misspelled.size() == 0) {
    System.out.println("No misspelled words!");
} else {
    System.out.println("Misspelled words are:");

    for (String word : misspelled) {
        System.out.println("    " + word);
    }
}
```

This example used a set of `String`. You could also have a set of `Integer`, a set of `Float`, or a set of any other type of object, so long as the type implements `Comparable`. For example,

```
public class Employee implements Comparable<Employee> {
    private int employeeNumber;
    private String firstName;
    private String lastName;
    private float salary;

    public int compareTo(Employee that) {
        return this.employeeNumber - that.employeeNumber;
    }
    // ... various other methods
}

public class HumanResources {
    private Set<Employee> staff;

    public void hire(Employee recruit) {
        staff.add(recruit);
    }
    // ... etc.
}
```

If you want to associate data with each key, use interface [Map](#) and the corresponding class [TreeMap](#). A `Map<K, V>` associates a value of type `V` with each key of type `K`. For example, if you want to quickly look up an `Employee` given his employee number, you should use a `Map` rather than a `Set` to keep track of employees.

```
// the following data member and methods are inside a class definition
private Map<Integer, Employee> staff = new TreeMap<Integer, Employee>();

public Employee getEmployee(int number) {
    return staff.get(number);
}
```

```
public void addEmployee(int number, Employee emp) {
    staff.put(number, emp);
}
```

As another example, here is a complete program that counts the number of occurrences of each word in a document.

```
import java.util.*;
import java.io.*;

public class CountWords {
    public static void main(String[] args) throws Exception {
        if (args.length != 1) {
            System.err.println("usage: java CountWords file_name");
            return;
        }

        Map<String, Integer> wordCount = new TreeMap<String, Integer>();
        Scanner in = new Scanner(new File(args[0]));
        in.useDelimiter("\\W+");
        while (in.hasNext()) {
            String word = in.next();
            Integer count = wordCount.get(word);
            if (count == null) {
                wordCount.put(word, 1);
            } else {
                wordCount.put(word, count + 1);
            }
        }
        for (String word : wordCount.keySet()) {
            System.out.println(word + " " + count.get(word));
        }
    }
} // CountWords
```

(The statement `in.useDelimiter("\\W+");` tells the `Scanner` that words are delimited by non-word characters. Without it, the program would look for "words" separated by spaces, considering "details" and "details.)" to be (different) words. See the documentation for [Scanner](#) and [Pattern](#) for more details.)

The value type `V` can be any class or interface. The key type `K` can be any class or interface that implements `Comparable`, for example,

```
class MyKey implements Comparable<MyKey> {
    // ...
    public int compareTo(MyKey other) {
        // return a value < 0 if this key is less than other,
        // == 0 if this key is equal to other, and
        // > 0 if this key is greater than other, and
    }
}
```

The method `put(key, value)` returns the value previously associated with `key` if any or `null` if `key` is a new key. The method `get(key)` returns the value associated with `key` or `null` if there is no such value. Both keys and values can be `null`. If your program stores `null` values in the map, you should use [containsKey\(key\)](#) to check whether a particular key is present.

`Map` has many other useful methods. Of particular note are [size\(\)](#), [remove\(key\)](#), [clear\(\)](#), and [keySet\(\)](#). The `keySet()` method returns a `Set` containing all the keys currently in the map. The `CountWords` example uses it to list the words in the document.

## Summary

A binary search tree can be used to store any objects that implement the `Comparable` interface (i.e., that define the `compareTo` method). A BST can also be used to store `Comparable` objects plus some associated data. The advantage of using a binary search tree (instead of, say, a linked list) is that, if the tree is reasonably balanced (shaped more like a "full" tree than like a "linear" tree), the insert, lookup, and delete operations can all be implemented to run in  $O(\log N)$  time, where  $N$  is the number of stored items. For a linked list, although insert can be implemented to run in  $O(1)$  time, lookup and delete take  $O(N)$  time in the worst case.

Logarithmic time is generally *much* faster than linear time. For example, for  $N = 1,000,000$ :  $\log_2 N = 20$ .

Of course, it is important to remember that for a "linear" tree (one in which every node has one child), the worst-case times for insert, lookup, and delete will be  $O(N)$ .

# Red-Black Trees

## Contents

- [Introduction](#)
- [Red-Black Tree Operations](#)
  - [insert](#)
  - [delete](#)
- [Summary](#)

## Introduction

Recall that, for binary search trees, although the average-case times for the lookup, insert, and delete methods are all  $O(\log N)$ , where  $N$  is the number of nodes in the tree, the worst-case time is  $O(N)$ . We can *guarantee*  $O(\log N)$  time for all three methods by using a **balanced** tree -- a tree that always has height  $O(\log N)$  -- instead of a binary search tree.

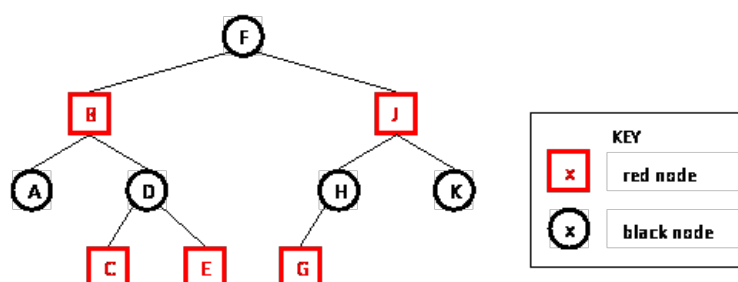
A number of different balanced trees have been defined, including **AVL trees**, **2-4 trees**, and **B trees**. You might learn about the first two in an algorithms class and the third in a database class. Here we will look at yet another kind of balanced tree called a **red-black tree**.

The important idea behind all of these trees is that the insert and delete operations may *restructure* the tree to keep it balanced. So lookup, insert, and delete will always be logarithmic in the number of nodes but insert and delete may be more complicated than for binary search trees.

A **red-black tree** is a binary search tree in which

- each node has a color (red or black) associated with it (in addition to its key and left and right children)
- the following 3 properties hold:
  1. (**root property**) The root of the red-black tree is black
  2. (**red property**) The children of a red node are black.
  3. (**black property**) For each node with at least one null child, the number of black nodes on the path from the root to the null child is the same.

An example of a red-black tree is shown below:



## Operations on a Red-Black Tree

As with the binary search tree, we will want to be able to perform the following operations on red-black trees:

- insert a key value (insert)
- determine whether a key value is in the tree (lookup)
- remove key value from the tree (delete)
- print all of the key values in sorted order (print)

Because a red-black tree *is* a binary search tree and operations that don't change the structure of a tree won't affect whether the tree satisfies the red-black tree properties, the lookup and print operations are identical to lookup and print for binary search trees.

### The insert operation

The goal of the insert operation is to insert key  $K$  into tree  $T$ , maintaining  $T$ 's red-black tree properties. A special case is required for an empty tree. If  $T$  is empty, replace it with a single **black** node containing  $K$ . This ensures that the root property is satisfied.

If  $T$  is a non-empty tree, then we do the following:

1. use the BST insert algorithm to add  $K$  to the tree
2. color the node containing  $K$  **red**
3. restore red-black tree properties (if necessary)

Recall that the BST insert algorithm always adds a leaf node. Because we are dealing with a non-empty red-black tree, adding a leaf node will not affect  $T$ 's

satisfaction of the root property. Moreover, adding a **red** leaf node will not affect T's satisfaction of the black property. However, adding a red leaf node may affect T's satisfaction of the red property, so we will need to check if that is the case and, if so, fix it (step 3). In fixing a red property violation, we will need to make sure that we don't end up with a tree that violates the root or black properties.

For step 3 for inserting into a non-empty tree, what we need to do will depend on the color of K's parent. Let P be K's parent. We need to consider two cases:

### Case 1: K's parent P is **black**

If K's parent P is black, then the addition of K did not result in the red property being violated, so there's nothing more to do.

### Case 2: K's parent P is **red**

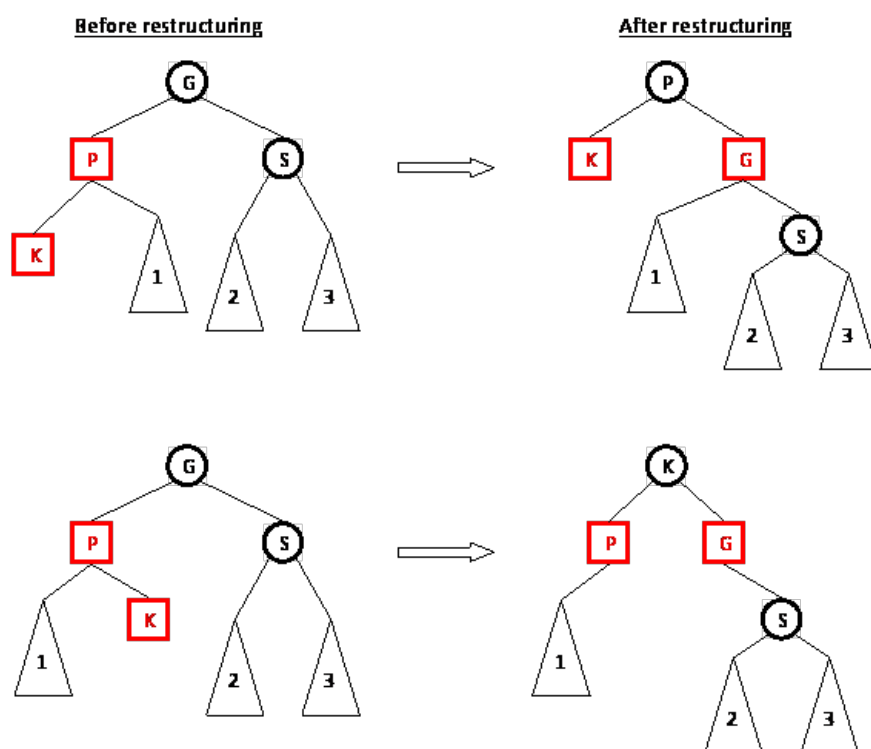
If K's parent P is red, then P now has a red child, which violates the red property. Note that P's parent, G, (K's grandparent) must be black (why?). In order to handle this **double-red** situation, we will need to consider the color of G's other child, that is, P's sibling, S. (Note that S might be null, i.e., G only has one child and that child is P.) We have two cases:

#### Case 2a: P's sibling S is **black** or null

If P's sibling S is black or null, then we will do a trinode **restructuring** of K (the newly added node), P (K's parent), and G (K's grandparent). To do a restructuring, we first put K, P, and G in order; let's call this order A, B, and C. We then make B the parent of A and C, color B black, and color A and C red. We also need to make sure that any subtrees of P and S (if S is not null) end up in the appropriate place once the restructuring is done.

There are four possibilities for the relative ordering of K, P, and G. The way a restructuring is done for each of the possibilities is shown below.

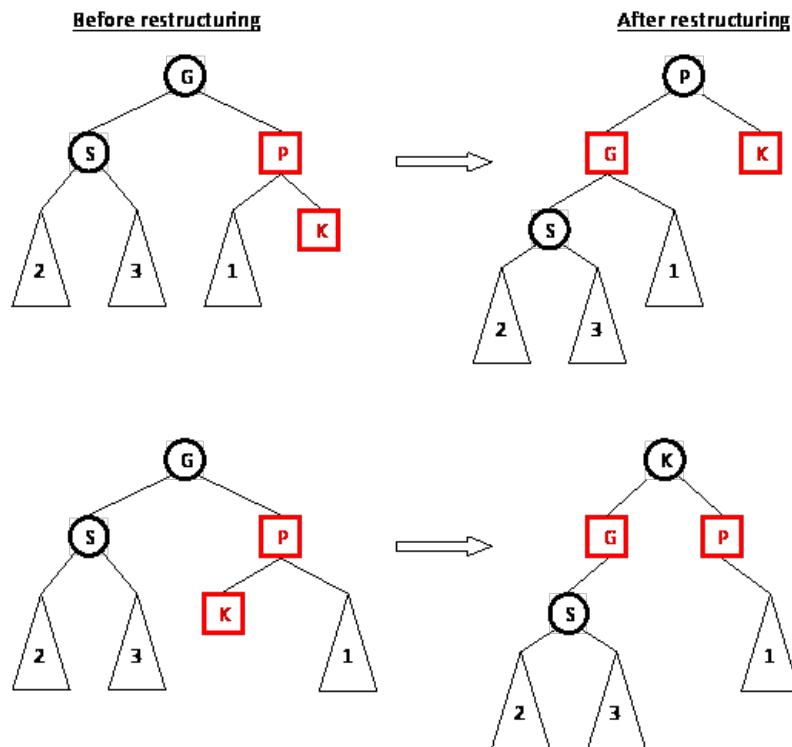
The first two possibilities are:



If S is null, then in the pictures above, S (and its subtrees labelled 2 and 3) would be replaced with nothing (i.e., null).

The second two possibilities are mirror-images of the first two possibilities:

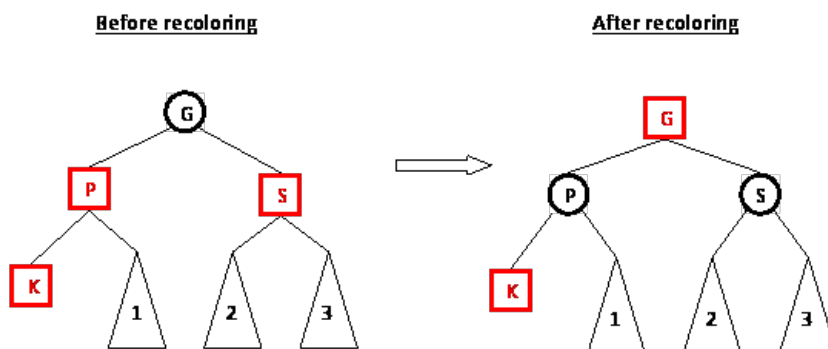




Once a restructuring is done, the double-red situation has been handled and there's nothing more to do (you should convince yourself, by looking at the diagrams above, that restructuring will not result in a violation of the black property).

#### Case 2b: P's sibling S is red

If P's sibling S is red, then we will do a **recoloring** of P, S, and G: the color of P and S is changed to black and the color of G is changed to red (unless G is the root, in which case we leave G black to preserve the root property).



Recoloring does not affect the black property of a tree: the number of black nodes on any path that goes through P and G is unchanged when P and G switch colors (similarly for S and G). But, the recoloring *may* have introduced a double-red situation between G and G's parent. If that is the case, then we recursively handle the double-red situation starting at G and G's parent (instead of K and K's parent).

An example of adding several values to a red-black tree will be presented in lecture.

#### What is the time complexity for insert?

Inserting a key into a non-empty tree has three steps. In the first step, the BST insert operation is performed. The BST insert operation is  $O(\text{height of tree})$  which is  $O(\log N)$  because a red-black tree is balanced. The second step is to color the new node red. This step is  $O(1)$  since it just requires setting the value of one node's color field. In the third step, we restore any violated red-black properties.

Restructuring is  $O(1)$  since it involves changing at most five pointers to tree nodes. Once a restructuring is done, the insert algorithm is done, so at most 1 restructuring is done in step 3. So, in the worst-case, the restructuring that is done during insert is  $O(1)$ .

Changing the colors of nodes during recoloring is  $O(1)$ . However, we might then need to handle a double-red situation further up the path from the added node to the root. In the worst-case, we end up fixing a double-red situation along the entire path from the added node to the root. So, in the worst-case, the recoloring that is done during insert is  $O(\log N)$  (= time for one recoloring \* max number of recolorings done =  $O(1) * O(\log N)$ ).

Thus, the third step (restoration of red-black properties) is  $O(\log N)$  and the total time for insert is  $O(\log N)$ .

## The delete operation

The delete operation is similar in feel to the insert operation, but more complicated. You will not be responsible for knowing the details of how to delete a key from a red-black tree.

## Summary

Balanced search trees have a height that is always  $O(\log N)$ . One consequence of this is that lookup, insert, and delete on a balanced search tree can be done in  $O(\log N)$  worst-case time. In contrast, binary search trees have a worst-case height of  $O(N)$  and lookup, insert, and delete are  $O(N)$  in the worst-case. Red-black trees are just one example of a balanced search tree.

Red-black trees are binary search trees that store one additional piece of information in each node (the node's color) and satisfy three properties. These properties deal with the way nodes can be colored (the root property and the red property) and the number of black nodes along paths from the root node to a null child pointer (the black property). Although it is not intuitively obvious, the algorithms for restoring these properties after a node has been added or removed result in a tree that stays balanced.

While the lookup method is identical for binary search trees and red-black trees, the insert and delete methods are more complicated for red-black trees. The insert method initially performs the same insert algorithm as is done in binary search trees and then must perform steps to restore the red-black tree properties through restructuring and recoloring. The delete method for a red-black tree is more complicated than the insert method and is not included in these notes.

# Red-Black Trees

## Contents

- [Introduction](#)
- [Red-Black Tree Operations](#)
  - [insert](#)
  - [delete](#)
- [Summary](#)

## Introduction

Recall that, for binary search trees, although the average-case times for the lookup, insert, and delete methods are all  $O(\log N)$ , where  $N$  is the number of nodes in the tree, the worst-case time is  $O(N)$ . We can *guarantee*  $O(\log N)$  time for all three methods by using a **balanced** tree -- a tree that always has height  $O(\log N)$  -- instead of a binary search tree.

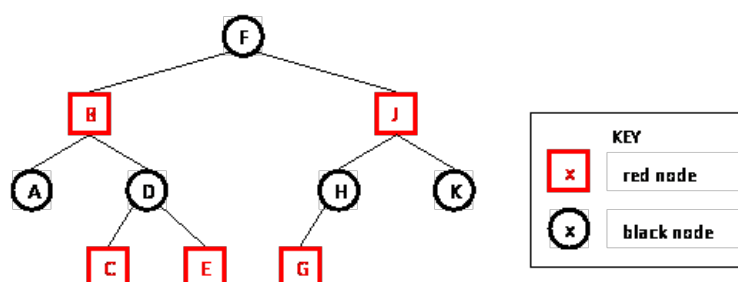
A number of different balanced trees have been defined, including **AVL trees**, **2-4 trees**, and **B trees**. You might learn about the first two in an algorithms class and the third in a database class. Here we will look at yet another kind of balanced tree called a **red-black tree**.

The important idea behind all of these trees is that the insert and delete operations may *restructure* the tree to keep it balanced. So lookup, insert, and delete will always be logarithmic in the number of nodes but insert and delete may be more complicated than for binary search trees.

A **red-black tree** is a binary search tree in which

- each node has a color (red or black) associated with it (in addition to its key and left and right children)
- the following 3 properties hold:
  1. (**root property**) The root of the red-black tree is black
  2. (**red property**) The children of a red node are black.
  3. (**black property**) For each node with at least one null child, the number of black nodes on the path from the root to the null child is the same.

An example of a red-black tree is shown below:



## Operations on a Red-Black Tree

As with the binary search tree, we will want to be able to perform the following operations on red-black trees:

- insert a key value (insert)
- determine whether a key value is in the tree (lookup)
- remove key value from the tree (delete)
- print all of the key values in sorted order (print)

Because a red-black tree *is* a binary search tree and operations that don't change the structure of a tree won't affect whether the tree satisfies the red-black tree properties, the lookup and print operations are identical to lookup and print for binary search trees.

### The insert operation

The goal of the insert operation is to insert key  $K$  into tree  $T$ , maintaining  $T$ 's red-black tree properties. A special case is required for an empty tree. If  $T$  is empty, replace it with a single **black** node containing  $K$ . This ensures that the root property is satisfied.

If  $T$  is a non-empty tree, then we do the following:

1. use the BST insert algorithm to add  $K$  to the tree
2. color the node containing  $K$  **red**
3. restore red-black tree properties (if necessary)

Recall that the BST insert algorithm always adds a leaf node. Because we are dealing with a non-empty red-black tree, adding a leaf node will not affect  $T$ 's

satisfaction of the root property. Moreover, adding a **red** leaf node will not affect T's satisfaction of the black property. However, adding a red leaf node may affect T's satisfaction of the red property, so we will need to check if that is the case and, if so, fix it (step 3). In fixing a red property violation, we will need to make sure that we don't end up with a tree that violates the root or black properties.

For step 3 for inserting into a non-empty tree, what we need to do will depend on the color of K's parent. Let P be K's parent. We need to consider two cases:

### Case 1: K's parent P is **black**

If K's parent P is black, then the addition of K did not result in the red property being violated, so there's nothing more to do.

### Case 2: K's parent P is **red**

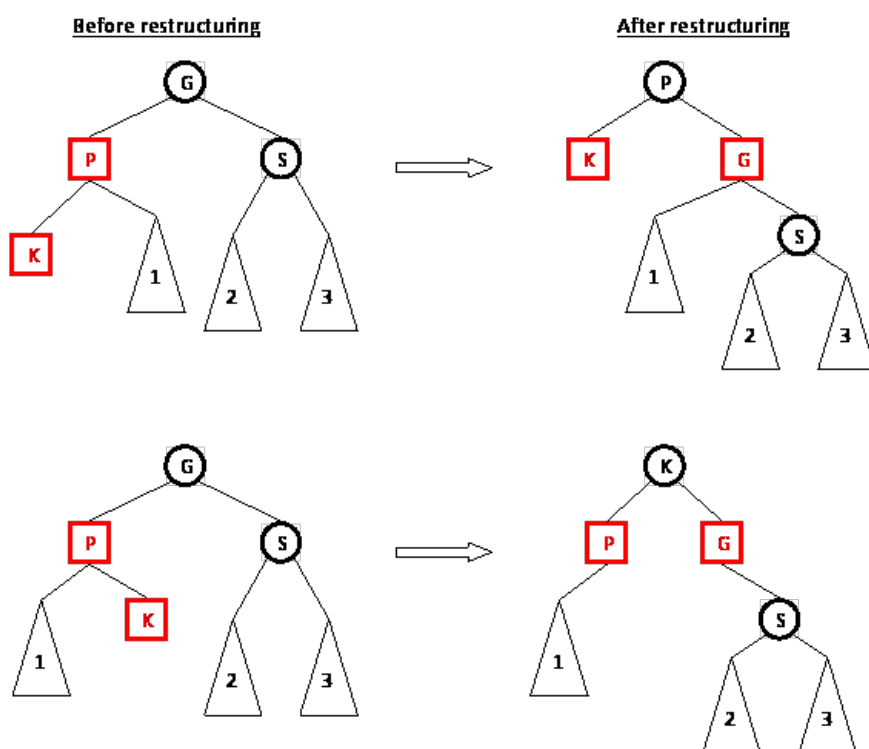
If K's parent P is red, then P now has a red child, which violates the red property. Note that P's parent, G, (K's grandparent) must be black (why?). In order to handle this **double-red** situation, we will need to consider the color of G's other child, that is, P's sibling, S. (Note that S might be null, i.e., G only has one child and that child is P.) We have two cases:

#### Case 2a: P's sibling S is **black** or null

If P's sibling S is black or null, then we will do a trinode **restructuring** of K (the newly added node), P (K's parent), and G (K's grandparent). To do a restructuring, we first put K, P, and G in order; let's call this order A, B, and C. We then make B the parent of A and C, color B black, and color A and C red. We also need to make sure that any subtrees of P and S (if S is not null) end up in the appropriate place once the restructuring is done.

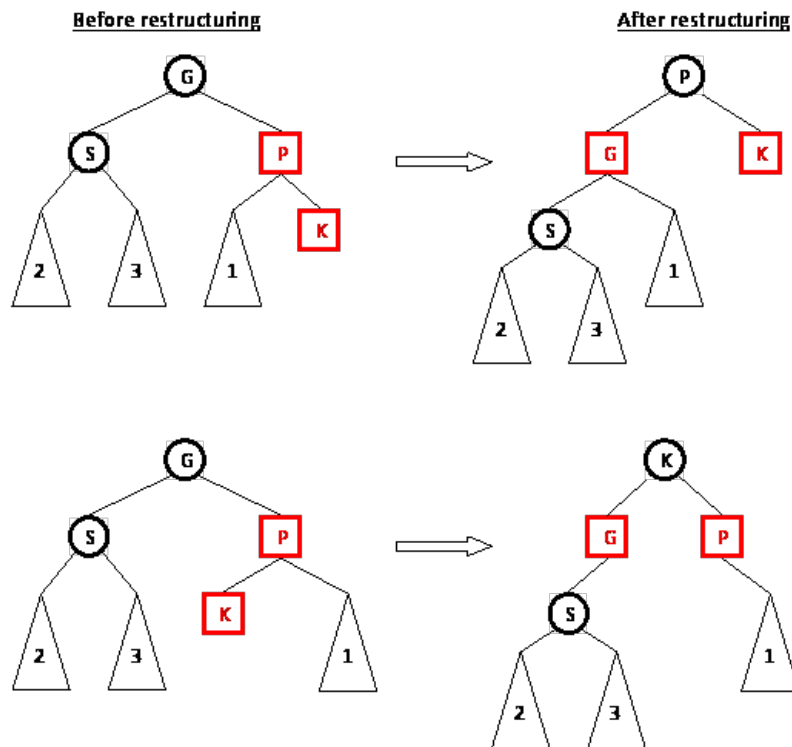
There are four possibilities for the relative ordering of K, P, and G. The way a restructuring is done for each of the possibilities is shown below.

The first two possibilities are:



If S is null, then in the pictures above, S (and its subtrees labelled 2 and 3) would be replaced with nothing (i.e., null).

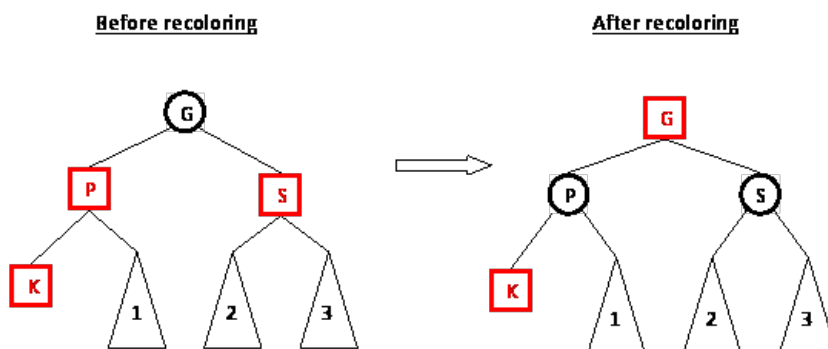
The second two possibilities are mirror-images of the first two possibilities:



Once a restructuring is done, the double-red situation has been handled and there's nothing more to do (you should convince yourself, by looking at the diagrams above, that restructuring will not result in a violation of the black property).

#### Case 2b: P's sibling S is red

If P's sibling S is red, then we will do a **recoloring** of P, S, and G: the color of P and S is changed to black and the color of G is changed to red (unless G is the root, in which case we leave G black to preserve the root property).



Recoloring does not affect the black property of a tree: the number of black nodes on any path that goes through P and G is unchanged when P and G switch colors (similarly for S and G). But, the recoloring *may* have introduced a double-red situation between G and G's parent. If that is the case, then we recursively handle the double-red situation starting at G and G's parent (instead of K and K's parent).

An example of adding several values to a red-black tree will be presented in lecture.

#### What is the time complexity for insert?

Inserting a key into a non-empty tree has three steps. In the first step, the BST insert operation is performed. The BST insert operation is  $O(\text{height of tree})$  which is  $O(\log N)$  because a red-black tree is balanced. The second step is to color the new node red. This step is  $O(1)$  since it just requires setting the value of one node's color field. In the third step, we restore any violated red-black properties.

Restructuring is  $O(1)$  since it involves changing at most five pointers to tree nodes. Once a restructuring is done, the insert algorithm is done, so at most 1 restructuring is done in step 3. So, in the worst-case, the restructuring that is done during insert is  $O(1)$ .

Changing the colors of nodes during recoloring is  $O(1)$ . However, we might then need to handle a double-red situation further up the path from the added node to the root. In the worst-case, we end up fixing a double-red situation along the entire path from the added node to the root. So, in the worst-case, the recoloring that is done during insert is  $O(\log N)$  (= time for one recoloring \* max number of recolorings done =  $O(1) * O(\log N)$ ).

Thus, the third step (restoration of red-black properties) is  $O(\log N)$  and the total time for insert is  $O(\log N)$ .

## The delete operation

The delete operation is similar in feel to the insert operation, but more complicated. You will not be responsible for knowing the details of how to delete a key from a red-black tree.

## Summary

Balanced search trees have a height that is always  $O(\log N)$ . One consequence of this is that lookup, insert, and delete on a balanced search tree can be done in  $O(\log N)$  worst-case time. In contrast, binary search trees have a worst-case height of  $O(N)$  and lookup, insert, and delete are  $O(N)$  in the worst-case. Red-black trees are just one example of a balanced search tree.

Red-black trees are binary search trees that store one additional piece of information in each node (the node's color) and satisfy three properties. These properties deal with the way nodes can be colored (the root property and the red property) and the number of black nodes along paths from the root node to a null child pointer (the black property). Although it is not intuitively obvious, the algorithms for restoring these properties after a node has been added or removed result in a tree that stays balanced.

While the lookup method is identical for binary search trees and red-black trees, the insert and delete methods are more complicated for red-black trees. The insert method initially performs the same insert algorithm as is done in binary search trees and then must perform steps to restore the red-black tree properties through restructuring and recoloring. The delete method for a red-black tree is more complicated than the insert method and is not included in these notes.

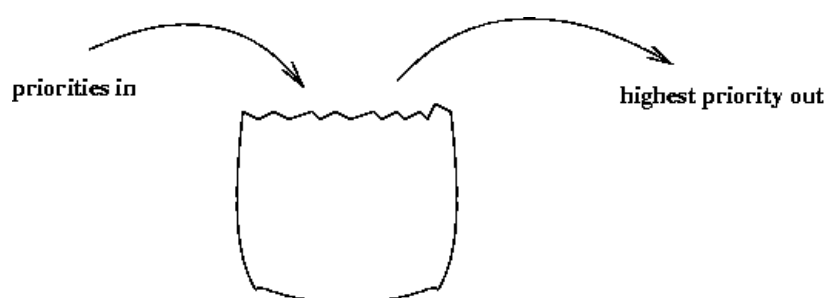
# Priority Queues

## Contents

- [Introduction](#)
  - [Test Yourself #1](#)
- [Heaps](#)
- [Implementing priority queues using heaps](#)
  - [Implementing insert](#)
    - [Test Yourself #2](#)
  - [Implementing removeMax](#)
    - [Test Yourself #3](#)
  - [Complexity](#)
- [Summary](#)

## Introduction

Here is a conceptual picture of a Priority Queue:



Think of a priority queue as a kind of bag that holds priorities. You can put one in and you can take out the current **highest** priority. (Priorities can be any Comparable values; in our examples, we'll just use numbers.)

A Priority Queue is different from a "normal" Queue, because instead of being a "first-in-first-out" data structure, values come out in order of priority. A Priority Queue might be used, for example, to handle the jobs sent to the Computer Science Department's printer: jobs sent by the department chair should be printed first, then jobs sent by professors, then those sent by graduate students, and finally those sent by undergraduates. The values put into the priority queue would be the priority of the sender (e.g., using 4 for the chair, 3 for professors, 2 for grad students, and 1 for undergrads) and the associated information would be the document to print. Each time the printer is free, the job with the highest priority would be removed from the print queue and printed. (Note that it is OK to have multiple jobs with the same priority; if there is more than one job with the same **highest** priority when the printer is free, then any one of them can be selected.)

The operations that need to be provided for a Priority Queue are shown in the following table, assuming that just priorities (no associated information) are to be stored in a Priority Queue.

Operation	Description
boolean isEmpty()	return true iff the PriorityQueue is empty
void insert(Comparable p)	add priority p to the PriorityQueue
Comparable removeMax()	remove and return the highest priority from the PriorityQueue (error if the PriorityQueue is empty)
Comparable getMax()	return the highest priority from the PriorityQueue, but do not remove it (error if the PriorityQueue is empty)

A Priority Queue can be implemented using many of the data structures that we've already studied (an array, a linked list, or a binary search tree). However, those data structures do not provide the most efficient operations. To make all of the operations very efficient, we'll use a new data structure called a **heap**.

### TEST YOURSELF #1

Consider implementing a priority queue using an array, a linked list, or a BST. For each, describe how each of the priority queue operations (as well as a constructor) would be implemented and what the worst-case time would be.

[solution](#)

## Heaps

A **heap** is a binary tree (in which each node contains a Comparable key value), with two special properties:

The **ORDER** property:

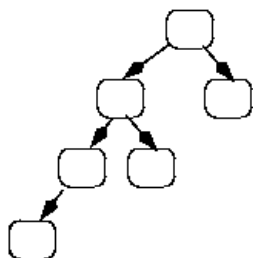
For every node  $N$ , the value in  $N$  is *greater than or equal to* the values in its children (and thus is also greater than or equal to all of the values in its subtrees).

The **SHAPE** property:

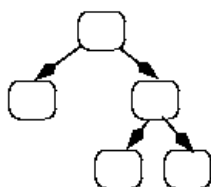
1. All leaves are either at depth  $d$  or  $d-1$  (for some value  $d$ ).
2. All of the leaves at depth  $d-1$  are to the *right* of the leaves at depth  $d$ .
3. (a) There is at most 1 node with just 1 child. (b) That child is the *left* child of its parent, and (c) it is the *rightmost* leaf at depth  $d$ .

(Note that the shape property is the definition of what it means for a binary tree to be **complete**.)

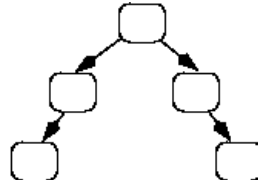
Here are some binary trees, some of which violate the *shape* properties and some of which respect those properties:



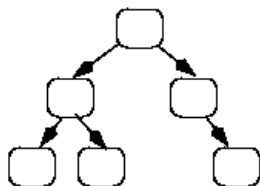
NO: violates shape property 1



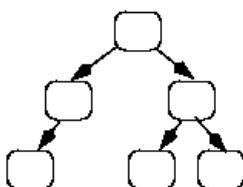
NO: violates shape property 2



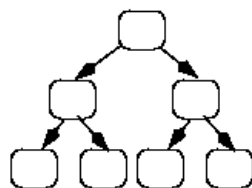
NO: violates shape property 3(a)



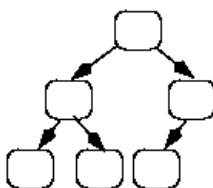
NO: violates shape property 3(b)



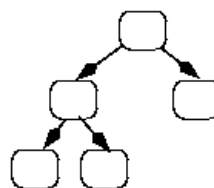
NO: violates shape property 3(c)



YES!

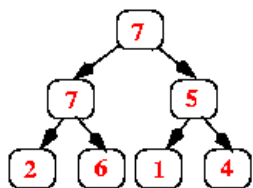


YES!

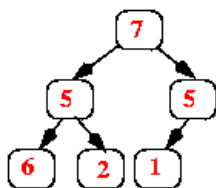


YES!

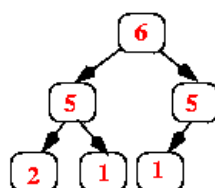
And here are some more trees; they all have the *shape* property but some violate the *order* property:



YES!



NO (6 is > 5)



YES!

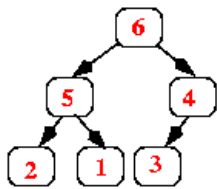
## Implementing priority queues using heaps

Now let's consider how to implement priority queues using a heap. The standard approach is to use an *array* (or an *ArrayList*), starting at position 1 (instead of 0), where each item in the array corresponds to one node in the heap as follows:

- The root of the heap is always in `array[1]`.
- Its *left* child is in `array[2]`.
- Its *right* child is in `array[3]`.
- In general, if a node is in `array[k]`, then its left child is in `array[k*2]` and its right child is in `array[k*2 + 1]`.
- If a node is in `array[k]`, then its *parent* is in `array[k/2]` (using integer division, so that if  $k$  is odd, then the result is truncated, e.g.,  $3/2 = 1$ ).

Here's an example, showing both the conceptual heap (the binary tree) and its array representation:





[0]	[1]	[2]	[3]	[4]	[5]	[6]
	6	5	4	2	1	3

Note that the heap's *shape* property guarantees that there are never any "holes" in the array.

The operations that determine if a heap is empty is quite straightforward; below we discuss the insert and removeMax operations.

## Implementing insert

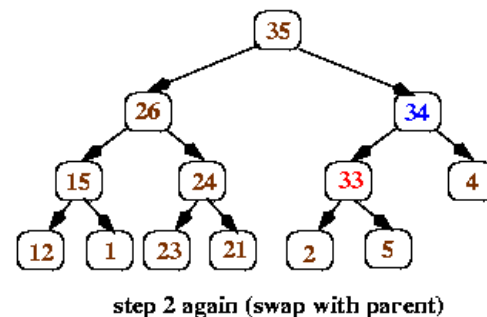
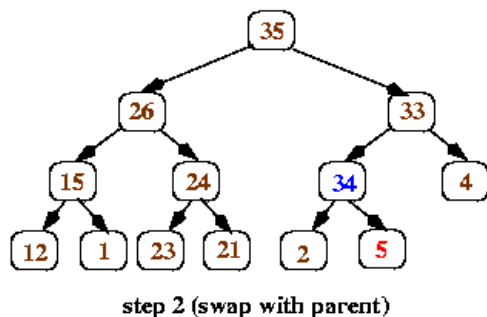
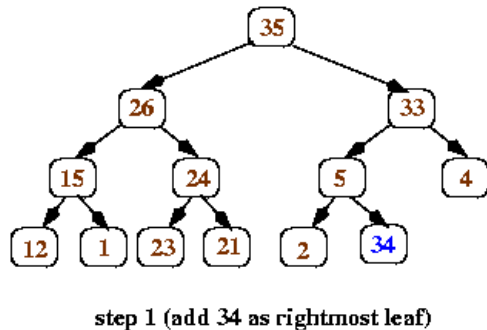
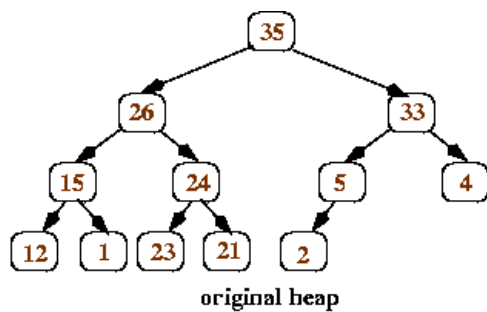
When a new value is inserted into a heap, we need to:

- add the value so that the heap still has the *order* and *shape* properties, and
- do it efficiently!

The way to achieve these goals is as follows:

1. Add the new value at the *end* of the array; that corresponds to adding it as a new rightmost leaf in the tree (or, if the tree was a **full** binary tree, i.e., all leaves were at the *same* depth  $d$ , then that corresponds to adding a new leaf at depth  $d+1$ ).
2. Step 1 above ensures that the heap still has the *shape* property; however, it may not have the *order* property. We can check that by comparing the new value to the value in its parent. If the parent is smaller, we swap the values, and we continue this check-and-swap procedure up the tree until we find that the order property holds, or we get to the root.

Here's a series of pictures to illustrate inserting the value 34 into a heap:



All done!

## TEST YOURSELF #2

Insert the values 6, 40, and 28 into the tree shown above (after 34 has been inserted).

[solution](#)

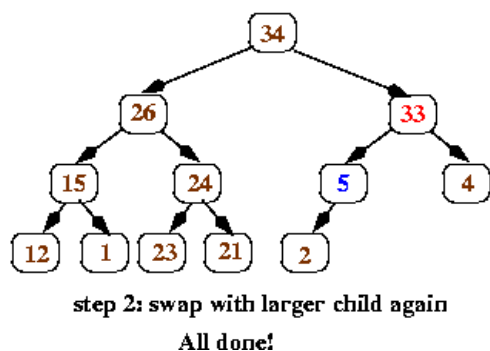
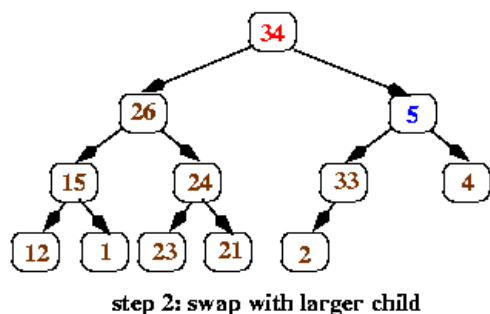
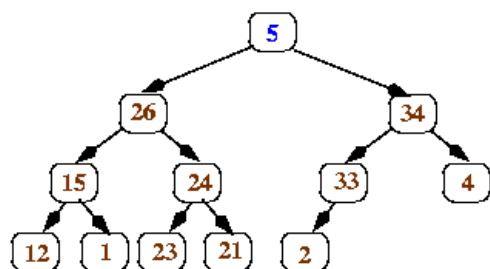
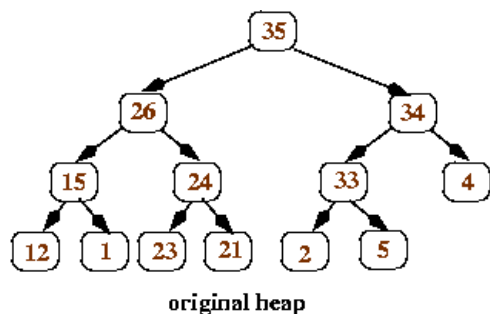
## Implementing removeMax

Because heaps have the *order* property, the largest value is always at the root. Therefore, the `getMax` operation will always return the root value and the `removeMax` operation will always remove and return the root value; the question then is how to replace the root node so that the heap still has the *order* and *shape* properties.

The answer is to use the following algorithm:

1. Replace the *value* in the root with the *value* at the end of the array (which corresponds to the heap's rightmost leaf at depth  $d$ ). Remove that leaf from the tree.
2. Now work your way down the tree, swapping values to restore the order property: each time, if the value in the current node is less than one of its children, then swap its value with the *larger* child (that ensures that the new root value is larger than both of its children).

Here's a series of pictures to illustrate the removeMax operation applied to the heap shown above.



---

### TEST YOURSELF #3

Perform 3 more removeMax operations using the example tree.

[solution](#)

---

### Complexity

For the insert operation, we start by adding a value to the end of the array (constant time, assuming the array doesn't have to be expanded); then we swap values up the tree until the *order* property has been restored. In the worst case, we follow a path all the way from a leaf to the root (i.e., the work we do is proportional to the height of the tree). Because a heap is a *balanced* binary tree, the height of the tree is  $O(\log N)$ , where  $N$  is the number of values stored in the tree.

The removeMax operation is similar: in the worst case, we follow a path down the tree from the root to a leaf. Again, the worst-case time is  $O(\log N)$ .

### Summary

A Priority Queue is an abstract data type that stores priorities (Comparable values) and perhaps associated information. A priority queue supports inserting new

priorities and removing/returning the highest priority. When a priority queue is implemented using a **heap**, the worst-case times for both insert and removeMax are logarithmic in the number of values in the priority queue.

# Hashing

## Contents

- [Introduction](#)
  - [Test Yourself #1](#)
- [Lookup, Insert, and Delete](#)
- [Choosing the Hashtable Size](#)
- [Choosing the Hash Function](#)
  - [Test Yourself #2](#)
- [Summary](#)

## Introduction

Recall that for a balanced tree (e.g., a red-black tree), the insert, lookup, and delete operations can be performed in time logarithmic in the number of values stored in the tree. Can we do better? Yes! We can use a technique called **hashing** that is logarithmic in the worst case, but has expected time  $O(1)$ !

The basic idea is to store values (unique keys plus perhaps some associated data) in an array, computing each key's position in the array as a function of its value. This takes advantage of the fact that we can do a subscripting operation on an array in constant time.

For example, suppose we want to store information about 50 employees, each of whom has a unique ID number. The ID numbers start with 100 and the highest ID number for any employee is 200 (i.e., there are a total of 101 possible ID numbers, all in the range 100 to 200). In this case, we can use an array of size 101 and we can store the information about the employee with ID number  $k$  in `array[k-100]`. The insert, lookup, and delete operations will all be constant time. The only disadvantage is that some of the array entries will be empty (i.e., will contain null to indicate that no information is stored there), so some space will be wasted.

Before we go on, here is some terminology:

- The array is called the **hashtable**.
- We will refer to the size of the array as **TABLE\_SIZE**.
- The function that maps a key value to the array index in which that key (and its associated data) will be stored is called the **hash function**. For this example, the key is the employee's ID number, and the hash function is:  $\text{hash}(k) = k - 100$ .

Now, think about the problem of storing information about *students* based on their ID numbers. The problem is that student ID numbers are in a **large** range (student IDs have 10 digits, so there are  $10^{10}$  possible values). In this case, it is probably not practical to use an array with one place for each possible value.

The solution is to use a "reasonable" sized array (more about this later) and to use a hash function that maps ID numbers to places in that array. If we can find a hash function that, given only a small set of ID numbers, is likely to map each ID number to a different place in the array, then we still have fast lookup, insert, and delete operations without requiring a huge array!

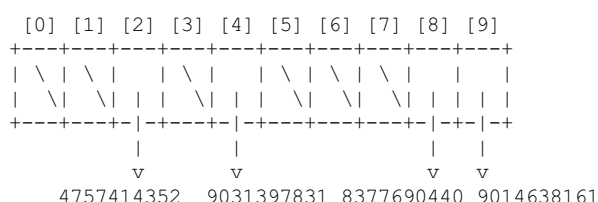
Here's an example: Suppose we decide to use an array of size 10 and we use the hash function:

$$\text{hash}(\text{ID}) = \text{sum of digits in ID mod } 10$$

Here are some ID numbers, the sums of their digits, and the array indexes to which they hash:

ID	Sum of digits	Array index (sum of digits mod 10)
9014638161	39	9
9103287648	48	8
4757414352	42	2
8377690440	48	8
9031397831	44	4

Note that we have a problem: both the second and the fourth ID have the **same** hash value (8). This is called a **collision**. How can we store both keys in `array[8]`? The answer is that we can make the array an array of linked lists, or an array of search trees, so that in case of collisions (if multiple keys have the same hash value), we can store multiple keys in the same place in the array. Assuming that we use linked lists, here's what the hashtable looks like after the 5 ID numbers given above have been inserted:



## TEST YOURSELF #1

Consider storing the names: George, Amy, Alan, and Sandy in a hashtable of size 10, using the hash function:

$\text{hash}(\text{name}) = \text{sum of characters mod } 10$

where  $a = A = 1$ ,  $b = B = 2$ , etc. Draw the hashtable that would be produced.

[solution](#)

Two important questions are:

- How should we choose the size of the hashtable (the array)?
- How should we choose the hash function?

The answers will be discussed below. First, let's assume that the hashtable size is `TABLE_SIZE` and that we have a hash function, and let's consider what the lookup, insert, and delete operations will do.

## Lookup, Insert, and Delete

To look up a key  $k$  in a hashtable, all you have to do is compute  $k$ 's hash value ( $v = \text{hash}(k)$ ), then see if  $k$  is in `array[v]`. As mentioned above, the array will contain linked lists, or possibly search trees. In either case, you should already know how to look for  $k$ . The time for lookup will be proportional to the time for the hash function, plus the time to look for  $k$  in the data structure in `array[v]`. In the best case, when at most one key hashes to each location in the table, lookup in `array[v]` will be  $O(1)$ . In the worst case, *all* of the keys will hash to the same place. In that case, if linked lists are used, the worst-case time for lookup will be  $O(N)$ , where  $N$  is the number of values stored in the hashtable. If a balanced search tree is used, the time will be  $O(\log N)$ .

Inserting a key  $k$  in a hashtable is similar to looking it up: first,  $v = \text{hash}(k)$  is computed, then  $k$  is added to `array[v]`. If linked lists are used,  $k$  should be added at the front of the list (since that can be done in constant time). The time for insert is similar to the time for lookup: the sum of the time for the hash function and the time to insert  $k$  into `array[v]`. However, if linked lists are used, the time to insert  $k$  into the array will always be  $O(1)$  rather than  $O(N)$  in the worst case.

To delete a key  $k$  from a hashtable,  $v = \text{hash}(k)$  is computed, then  $k$  is deleted from the linked-list / search tree in `array[v]`. The worst-case time is the same as for lookup, since the value has to be found before it can be deleted.

## Choosing the Hashtable Size

The best size to choose for the hashtable will depend on the expected number of values that will be stored and how important space consumption is (there will be a trade-off between the amount of space used and the number of keys that hash to the same array index). It is reasonable to use a table that is a bit larger than the expected number of items (say 1.25 times the expected number). If the number of items to be stored is not known, then you can always plan to expand the hashtable whenever it gets too full.

## Choosing the Hash Function

The important issues to consider when choosing a hash function are:

- The computation of the hash function should be *efficient* (i.e., should not take too long).
- The hash function should spread the key values as evenly as possible (i.e., should map different keys to different locations in the hashtable).

Since the result of the hash function will be used as an array index, it must be in the range 0 to `TABLE_SIZE-1`. Therefore, it is reasonable for the hash function to convert the key to an integer  $n$  and to return  $n \bmod \text{TABLE\_SIZE}$ .

If the keys *are* integers, with well distributed values (i.e., modding them with `TABLE_SIZE` is likely to produce results evenly distributed from 0 to `TABLE_SIZE-1`), then the hash function can just return  $\text{key} \bmod \text{TABLE\_SIZE}$ . However, if the keys are not well distributed or if they are strings (or some other non-integer type), then we must convert them to well-distributed integer values (before modding them by `TABLE_SIZE`).

Let's assume that the keys are strings. Most programming languages provide a way to convert individual characters to integers. In Java, you can just cast a `char` to an `int`; for example, to convert the first character in `String S` to an `int`, use:

```
int x = (int) (S.charAt(0));
```

Once we know how to convert individual characters to integers, we can convert a whole string to an integer in a number of ways. First, though, let's think about *which* characters in the string we want to use. Remember that we want our hash function to be *efficient* and to map different keys to different locations in the hashtable. Unfortunately, there tends to be tension between those two goals; for example, a hash function that only uses the first and last characters in a key will be faster to compute than a hash function that uses all of the characters in the key, but it will also be more likely to map different keys to the same hashtable location.

A reasonable solution is to compromise; e.g., to use  $N$  characters of the key for some reasonable value of  $N$  ("reasonable" will depend on how long you expect your keys to be and how fast you want your hash function to be). For keys that have more than  $N$  characters, there is also a question of which characters to choose. If the keys are likely to differ only at one end or the other (e.g., they are strings like "value1", "value2", etc., that differ only in their last characters), then this decision could make a big difference in how well the hash function "spreads out" the keys in the hashtable.

To simplify our discussion, let's assume that we know the keys won't be too long, so our hash function can simply use all of the characters in the keys.

Here are some ways to combine the integer values for the individual characters to compute a single integer  $n$  (remember that the hash function will return  $n \bmod \text{TABLE\_SIZE}$ ):

- Add the integer values of all of the characters. This method is not very good if the keys are short and the table size is large because summing the integer values of the characters may not ever produce large indexes, so many spaces in the hashtable will never be used and other spaces will have to store many keys. Furthermore, it will hash all **permutations** to the same value. For example, `hash("able")` will be the same as `hash("bale")`.
- Add the integer values of the characters, but multiply intermediate results by some constant to make the values larger. For example, suppose the key is "hello" and the integer values of the letters are:  $a = 1$ ,  $b = 2$ ,  $c = 3$ , etc. The values for the characters in "hello" are: 8, 5, 12, 12, 15. We can add the values, multiplying each intermediate result by 13:

```
(8+5) * 13 = 169
(169+12) * 13 = 2353
(2353+12) * 13 = 30745
(30745+15) * 13 = 399880
```

This technique gives you a wider range of values than just adding all of the characters. However, you would have to be prepared to handle overflow. (You would probably want to test the value of the sum so far, and if it is too large, divide by some constant, making it smaller to prevent overflow.)

- Multiply individual characters by different constants and then add. For example, we could multiply the value of the first character by 8, the second character by 4, and the third character by 3. Then start again: multiply the fourth character by 8, the fifth by 4, and the sixth by 3, etc., and finally add up all of the products. This is less likely to lead to overflow than the previous technique and (unlike the first technique) will usually map permutations to different values.

---

## TEST YOURSELF #2

Consider hashing keys that are strings containing only lower-case letters. The hash function that will be used computes the **product** of the integer values of the characters in a key, using the scheme:  $a = 0$ ,  $b = 1$ ,  $c = 2$ , etc. Why is this scheme not as good as using:  $a = 1$ ,  $b = 2$ , etc.?

[solution](#)

---

Although it is a good idea to understand the issues involved in choosing a hash function, if you program in Java, you can simply use the `hashCode` method that is supplied for every `Object` (including `Strings`). The method returns an integer  $j$ , and you can just use  $j \bmod \text{TABLE\_SIZE}$  as your hash function. Of course, if your keys are members of a class that you define, you will need to implement the `hashCode` method for that class yourself.

## Summary

- Given a fixed table size and a hash function that distributes keys evenly in the range 0 to  $\text{TABLE\_SIZE}-1$ , the expected times for insert, lookup, and delete are all  $O(1)$ , as long as the number of keys in the table is less than  $\text{TABLE\_SIZE}$ .
- Using balanced trees as array elements, the worst-case times for insert, lookup, and delete are all  $O(\log N)$ , where  $N$  is the number of keys stored in the table.
- Using linked lists as array elements, the worst-case times are  $O(1)$  for insert and  $O(N)$  for lookup and delete.
- A **disadvantage** of hashtables (compared to binary-search trees and red-black trees) is that it is not easy to implement a print method that prints all values in sorted order.

# Graphs

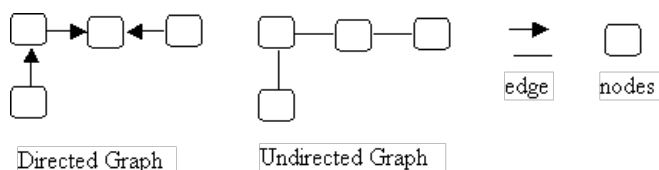
## Contents

- [Introduction](#)
- [Terminology](#)
- [Some special kinds of graphs](#)
- [Test Yourself #1](#)
- [Uses for Graphs](#)
- [Representing Graphs](#)
- [Test Yourself #2](#)
- [Graph Operations](#)
  - [Depth-First Search](#)
    - [Test Yourself #3](#)
    - [Uses for Depth-First Search](#)
    - [Test Yourself #4](#)
  - [Breadth-First Search](#)
  - [Dijkstra's Algorithm](#)
- [Summary](#)

## Introduction

**Graphs** are a generalization of trees. Like trees, graphs have **nodes** and **edges**. (The nodes are sometimes called **vertices** and the edges are sometimes called **arcs**.) However, graphs are more general than trees: in a graph, a node can have *any number* of incoming edges (in a tree, the root node cannot have any incoming edges and the other nodes can only have one incoming edge). Every tree is a graph, but not every graph is a tree.

There are two kinds of graphs, **directed** and **undirected**:



Note that in a directed graph, the edges are arrows (are directed from one node to another) while in the undirected graph the edges are plain lines (they have no direction). In a directed graph, you can only go from node to node following the direction of the arrows, while in an undirected graph, you can go either way along an edge. This means that in a directed graph it is possible to reach a "dead end" (to get to a node from which you cannot leave).

## Terminology

Here are two example graphs (one directed and one undirected) and the terminology to describe them.



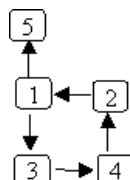
In the directed graph, there is an edge from node 2 to node 1; therefore:

- The two nodes are **adjacent** (they are **neighbors**).
- Node 2 is a **predecessor** of node 1.
- Node 1 is a **successor** of node 2.
- The **source** of the edge is node 2 and the **target** of the edge is node 1.

In the undirected graph, there is an edge between node 1 and node 3; therefore:

- Nodes 1 and 3 are **adjacent** (they are neighbors).

Now consider the following (directed) graph:

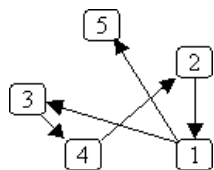


In this graph, there is a **path** from node 2 to node 5:  $2 \rightarrow 1 \rightarrow 5$ . There is a path from node 1 to node 2:  $1 \rightarrow 3 \rightarrow 4 \rightarrow 2$ . There is also a path from node 1 back to



itself:  $1 \rightarrow 3 \rightarrow 4 \rightarrow 2 \rightarrow 1$ . The first two paths are **acyclic** paths: no node is repeated; the last path is a **cyclic** path because node 1 occurs twice.

Note that the layout of the graph is arbitrary -- the important thing is which nodes are connected to which other nodes. So, for example, the following graph is the **same** as the one given above, it's just been drawn differently:

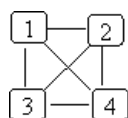


Also note that an edge can connect a node to itself, for example:



## Some special kinds of graphs

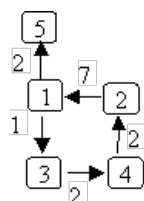
- A directed graph that has **no** cyclic paths (that contains no cycles) is called a **DAG** (a Directed Acyclic Graph).
- An undirected graph that has an edge between every pair of nodes is called a **complete** graph. Here's an example:



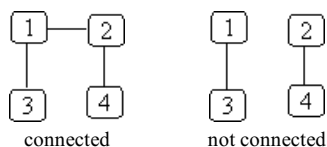
A directed graph can also be a complete graph; in that case, there must be an edge from every node to every other node.

- A graph that has values associated with its edges is called a **weighted** graph. The graph can be either directed or undirected. The weights can represent things like:
  - The cost of traversing the edge.
  - The length of the edge.
  - The time needed to traverse the edge.

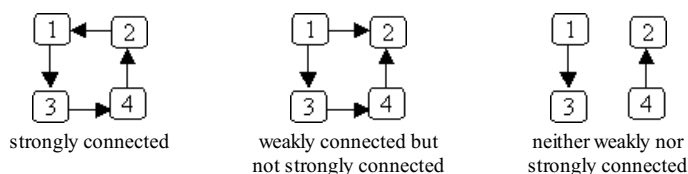
Here's an example of a weighted, directed graph:



- An **undirected** graph is **connected** if there is a path from every node to every other node. For example:



- A **directed** graph is **strongly connected** if there is a path from every node to every other node. A **directed** graph is **weakly connected** if, treating all edges as being **undirected**, there is a path from every node to every other node. For example:

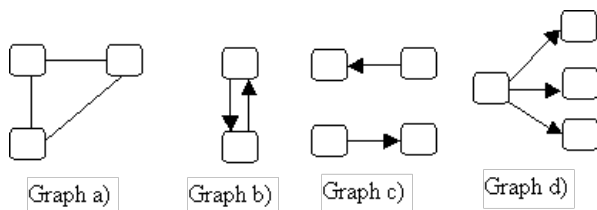


## TEST YOURSELF #1

For each of the following graphs, say whether it is:

- connected, strongly connected, weakly connected, or not connected
- complete or not complete

If the graph is a directed graph, also say whether it is cyclic or acyclic.

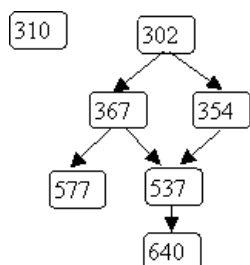


[solution](#)

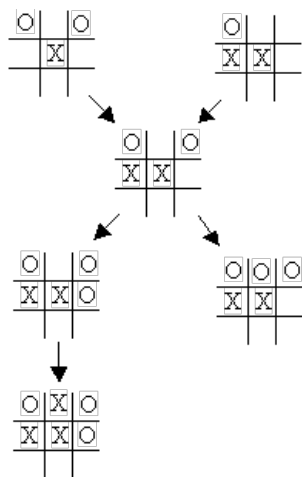
## Uses for Graphs

In general, the nodes of a graph represent objects and the edges represent relationships. Here are some examples:

- Flights between cities. The nodes represent the cities and there is an edge  $j \rightarrow k$  iff there is a flight from city  $j$  to city  $k$ . This graph could be a weighted graph, using the weights to represent the distance, the flight time, or the cost.
- Interdependent tasks to be done. The nodes represent the tasks and there is an edge  $j \rightarrow k$  iff task  $j$  must be completed before task  $k$ . For example, we can use a graph to represent what must be done to finish a CS major, with the nodes representing the courses to be taken, and the edges representing prerequisites. Here's a partial example:



- Flow charts (also known as control-flow graphs). The nodes represent the statements and conditions in a program and the edges represent the flow of control.
- State transition diagrams. The nodes represent states and the edges represent legal moves from state to state. For example, we could use a graph to represent legal moves in a game of tic-tac-toe. Here's a small part of that graph:



The reason graphs are good representations in cases like those described above is that there are many standard graph algorithms (operations on graphs) that can be used to answer useful questions like:

- What is the cheapest way to fly from Madison to Saskatoon?
- Which CS classes must I take before I can take CS640?
- Is it possible for variable  $k$  to be used before being initialized?
- Can I win a game of tic-tac-toe starting from the current position?

## Representing Graphs

In a tree, all nodes can be reached from the root node, so a tree can be represented using two classes: a `Treenode` class (used to represent each individual node) and a `Tree` class that contains a pointer to the root node. Some graphs have a similar property, i.e., there is a special "root" node from which all other nodes are reachable (control-flow graphs often have this property). In that case, a graph can also be represented using a `Graphnode` class for the individual

nodes and a `Graph` class that contains a pointer to the root node. However, if there is no root node, then the `Graph` class needs to use some other data structure to keep track of the nodes in the graph. There are many possibilities: an array, a List, or a Set of `Graphnodes` could be used.

The `Graphnodes` will contain whatever data is stored in a node (e.g., the name of a city, the name of a CS class, the statement represented by a control-flow graph node). The nodes will also contain pointers to their successors (stored, e.g., in an array, a List, or a Set).

Here's one reasonable pair of (incomplete) class definitions for directed graphs, using `ArrayLists` to store the nodes in the graph and the successors of each node:

```
class Graphnode<T> {
    // *** fields ***
    private T data;
    private List<Graphnode<T>> successors = new ArrayList<Graphnode<T>>();

    // *** methods ***
    ...
}

public class Graph {
    // *** fields ***
    private List<Graphnode<T>> nodes = new ArrayList<Graphnode<T>>();

    // *** methods ***
    ...
}
```

---

## TEST YOURSELF #2

Suppose we have a **weighted** graph (one in which each edge has an associated value). How could the class definitions given above be extended to store the edge weights?

[solution](#)

---

## Graph Operations

As discussed above, graphs are often a good representation for problems involving objects and their relationships because there are standard graph operations that can be used to answer useful questions about those relationships. Here we discuss two such operations: **depth-first search** and **breadth-first search** and some of their applications.

Both depth-first and breadth-first search are "orderly" ways to traverse the nodes and edges of a graph that are reachable from some starting node. The main difference between depth-first and breadth-first search is the order in which nodes are visited. Of course, since in general not all nodes are reachable from all other nodes, the choice of the starting node determines which nodes and edges will be traversed (either by depth-first or breadth-first search).

### Depth-first Search

**Depth-first search** can be used to answer many questions about a graph:

- Is it connected?
- Is there a path from node  $j$  to node  $k$ ?
- Does it contain a cycle?
- What nodes are reachable from node  $j$ ?
- Can the nodes be ordered so that for every node  $j$ ,  $j$  comes before all of its successors in the ordering?

The basic idea of a depth-first search is to start at some node  $n$ , and then to follow an edge out of  $n$ , then another edge out, etc., getting as far away from  $n$  as possible before visiting any more of  $n$ 's successors. To prevent infinite loops in graphs with cycles, we must keep track of which nodes have been visited. Here is the basic algorithm for a depth-first search from node  $n$ , starting with all nodes marked "unvisited":

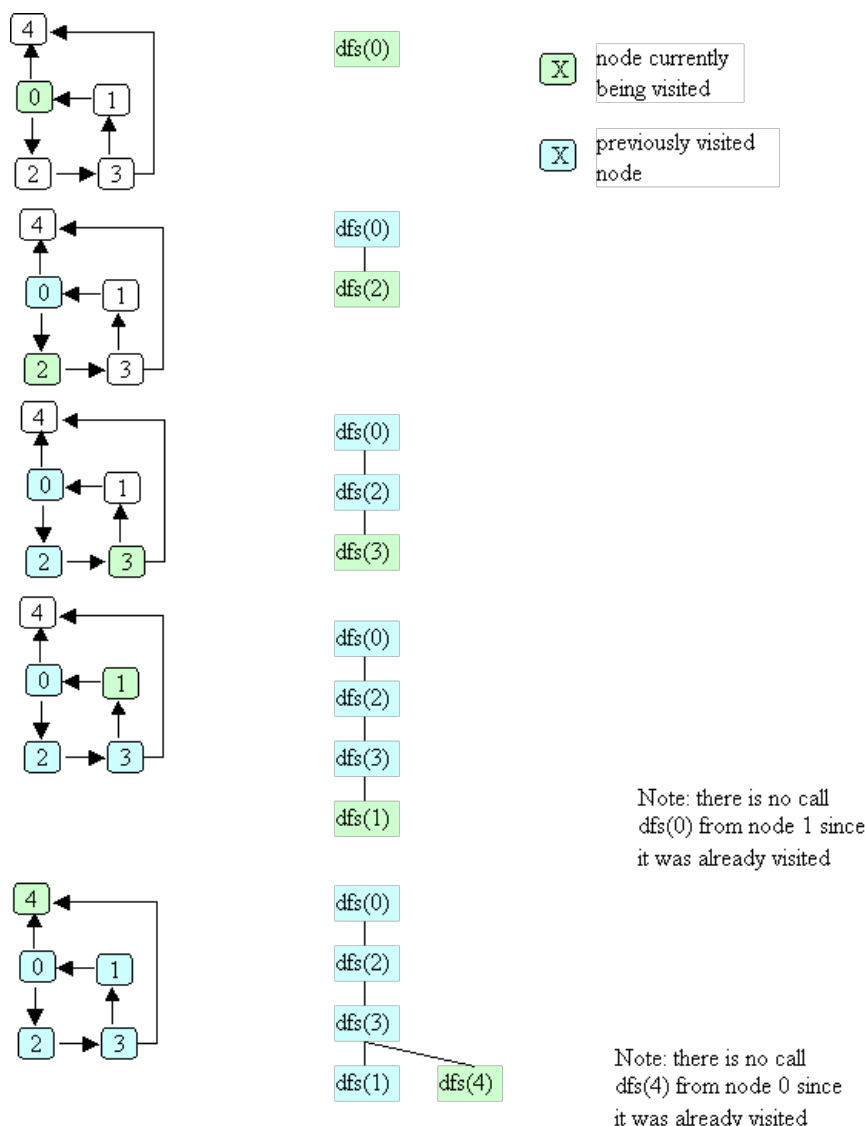
1. mark  $n$  "visited"
2. recursively do a depth-first search from each of  $n$ 's unvisited successors

Information about which nodes have been visited can be kept in the nodes themselves (e.g., using a boolean field) or, if the nodes are numbered from 1 to  $N$ , the "visited" information can be stored in an auxiliary array of booleans of size  $N$ . Below is code for depth-first search, assuming that visited information is in a node field named `visited`, that each node's successors are in a List named `successors`, and that the `Graphnode` class provides the usual `get/set` methods to access its fields. Note that this basic depth-first search doesn't actually do anything except mark nodes as having been visited. We'll see in the next section how to use variations on this code to do useful things.

```
public static void dfs (Graphnode<T> n) {
    n.setVisited(true);
    for (Graphnode<T> m : n.getSuccessors()) {
        if (! m.getVisited()) {
            dfs(m);
        }
    }
}
```

}

Here's a picture that illustrates the `dfs` method. In this example, node numbers are used to denote the nodes themselves (i.e., the call `dfs(0)` really means that the `dfs` method is called with a pointer to the node labeled 0). Two different colors are used to indicate the node currently being visited and the previously visited node.



Note that in the example illustrated above, the order in which the nodes are visited is: 0, 2, 3, 1, 4. Another possible order (if node 4 were the first successor of node 0) is: 0, 4, 2, 3, 1.

To analyze the time required for depth-first search, note that one call is made to `dfs` for each node that is reachable from the start node. Each call looks at all successors of the current node, so the time is  $O(\text{\# reachable nodes} + \text{total \# of outgoing edges from those nodes})$ . In the worst case, this is *all* nodes and *all* edges, so the worst-case time is  $O(N + E)$ , where  $N$  is the number of nodes in the graph and  $E$  is the number of edges in the graph.

### TEST YOURSELF #3

Assume that you start with all nodes "unvisited" and you do a depth-first search. Write a `(Graph)` method that sets all nodes back to "unvisited".

[solution](#)

### Uses for Depth-First Search

Recall that at the beginning of this section we said that depth-first search can be used to answers questions about a graph such as:

1. Is it connected?
2. Is there a path from node  $j$  to node  $k$ ?
3. Does it contain a cycle?
4. What nodes are reachable from node  $j$ ?
5. Can the nodes be ordered so that for every node  $j$ ,  $j$  comes before all of its successors in the ordering?

Questions 2, 3 and 5 are discussed; the others are left as exercises.

## Path Detection

The first question we will consider is: is there a path from node  $j$  to node  $k$ ? This question might be useful, for example:

- when the graph represents airline routes and we want to ask "Can I fly from Madison to London (maybe w/ some connections)?", or
- when the graph represent CS course prerequisites and we want to ask "Is CS367 a (transitive) prerequisite for CS640?"

To answer the question, do the following:

- step 1: mark all nodes "not visited"
- step 2: `dfs(j)`
- step 3: there is a path from  $j$  to  $k$  iff  $k$  is marked "visited"

## Cycle Detection

There are two variations that might be interesting:

1. Does a graph contain a cycle?
2. Is there a cyclic path starting from node  $j$ ?

Consider the example given above to illustrate depth-first search. There *is* a cycle in that graph starting from node 0. Is there something that happens during the depth-first search that indicates the presence of that cycle? Note that during `dfs(1)`, 0 is a successor of 1 but is already visited. But that isn't quite enough to say that there's a cycle, because during `dfs(3)`, node 4 is a successor of 3 that has already been visited, but there is *no* cycle starting from node 4.

What's the difference? The answer is that when node 0 is considered as a successor of node 1, the call `dfs(0)` is still "active" (i.e., its activation record is still on the call stack); however, when node 4 is considered as a successor of node 3, the call `dfs(4)` has already finished. How can we tell the difference? The answer is to keep track of when a node is "in progress" (as well as whether it has been visited or not). We can do this by using a `mark` field with three possible values:

1. UNVISITED
2. IN\_PROGRESS
3. DONE

instead of the boolean `visited` field we've been using. Initially, all nodes are marked UNVISITED. When the `dfs` method is first called for node  $n$ , it is marked IN\_PROGRESS. Once all of its successors have been processed, it is marked DONE. There is a cyclic path reachable from node  $n$  iff some node's successor is found to be marked IN\_PROGRESS during `dfs(n)`.

Here's the code for cycle detection:

```
public boolean hasCycle(Graphnode<T> n) {
    n.setMark(IN_PROGRESS);
    for (Graphnode<T> m : n.getSuccessors()) {
        if (m.getMark() == IN_PROGRESS) {
            return true;
        }
        if (m.getMark() != DONE) {
            if (hasCycle(m)) {
                return true;
            }
        }
    }
    n.setMark(DONE);
    return false;
}
```

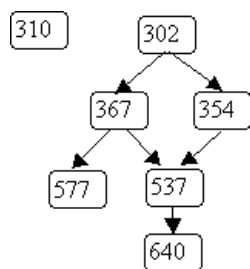
Note that if we want to know whether a graph contains a cycle anywhere (not just one that is reachable from node  $n$ ) we might have to call `hasCycle` at the "top-level" more than once. Here's a pseudo-code version of a method of the `Graph` class that returns true iff there is a cycle somewhere in the graph:

```
public boolean graphHasCycle() {
    mark all nodes unvisited
    for each node k in the graph {
        if (node k is marked unvisited) {
            if (hasCycle(k)) {
                return true;
            }
        }
    }
    return false;
}
```

## Topological Numbering

Think again about the graph that represents course prerequisites. As long as there are no cycles in the graph there is at least one order in which to take courses, such that all prereqs are satisfied, i.e., so that for every course, all prerequisites are taken before the course itself is taken. (Note that it is reasonable to assume that there are no cycles in a graph that represents course prerequisites because a cycle would mean that a course was a prerequisite for itself!)

Topological numbering can be used to find the order in which to take the classes (so that all prereqs are satisfied first). The goal is to assign numbers to nodes so that for every edge  $j \rightarrow k$ , the number assigned to  $j$  is less than the number assigned to  $k$ . A topological numbering of the prerequisites graph would tell you one legal order in which to take the CS courses. For example:



Two legal topological orderings:

1: 310	1: 302
2: 302	2: 367
3: 367	3: 354
4: 577	4: 537
5: 354	5: 640
6: 537	6: 310
7: 640	7: 577

To find a topological numbering, we use a variation of depth-first search. The intuition is as follows: as long as there are no cycles in the graph, there must be at least one node with no outgoing edges:

- The **last** number ( $N$ ) can be given to any such node (310, 577, or 640 in our example).
- Once all of a node's successors have numbers, the node itself can get the next smallest number.

These 2 situations correspond to the point in method `hasCycle` where node  $n$  is marked `DONE` (when it has no more unvisited successors). We just need to keep track of the current number. Below is a method that, given a node  $n$  and a number  $num$ , assigns topological numbers to all unvisited nodes reachable from  $n$ , starting with  $num$  and working down. Note that before calling this method for the first time, all nodes should be marked `UNVISITED` and that the initial call should pass  $N$  (the number of nodes in the graph) as the 2nd parameter.

```

public int topNum(Graphnode<T> n, int num) throws CycleException {
    n.setMark(IN_PROGRESS);
    for (Graphnode<T> m : n.getSuccessors()) {

        if (m.getMark() == IN_PROGRESS) {
            // no topological ordering for a cyclic graph!
            throw new CycleException();
        }
        if (m.getMark() != DONE) {
            num = topNum(m, num);
        }
    }

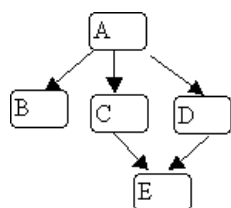
    // here when n has no more successors
    n.setMark(DONE);
    n.setNumber(num);
    return num - 1;
}

```

As was the case for cycle detection, we might need several "top-level" calls to number *all* nodes in a graph.

## TEST YOURSELF #4

**Question 1:** Give two different topological orderings for the following graph.



**Question 2:** The `topNum` method given above only assigns numbers to the nodes reachable from node  $n$ . Write pseudo code for method `numberGraph`, similar to the code given for method `graphHasCycle` above, that assigns topological numbers to *all* nodes in a graph.

**Question 3:** Write a `Graph` method `isConnected` that returns true if and only if the graph is connected. Assume that every node has a list of its predecessors as well as a list of its successors.

[solution](#)

## Breadth-first Search

**Breadth-first search** provides another "orderly" way to visit (part of) a graph. The basic idea is to visit all nodes at the same distance from the start node before visiting farther-away nodes. Like depth-first search, breadth-first search can be used to find all nodes reachable from the start node. It can also be used to find the shortest path between two nodes in an unweighted graph.

Breadth-first search uses a **queue** rather than recursion (which actually uses a stack); the queue holds "nodes to be visited". If the graph is a tree, breadth-first

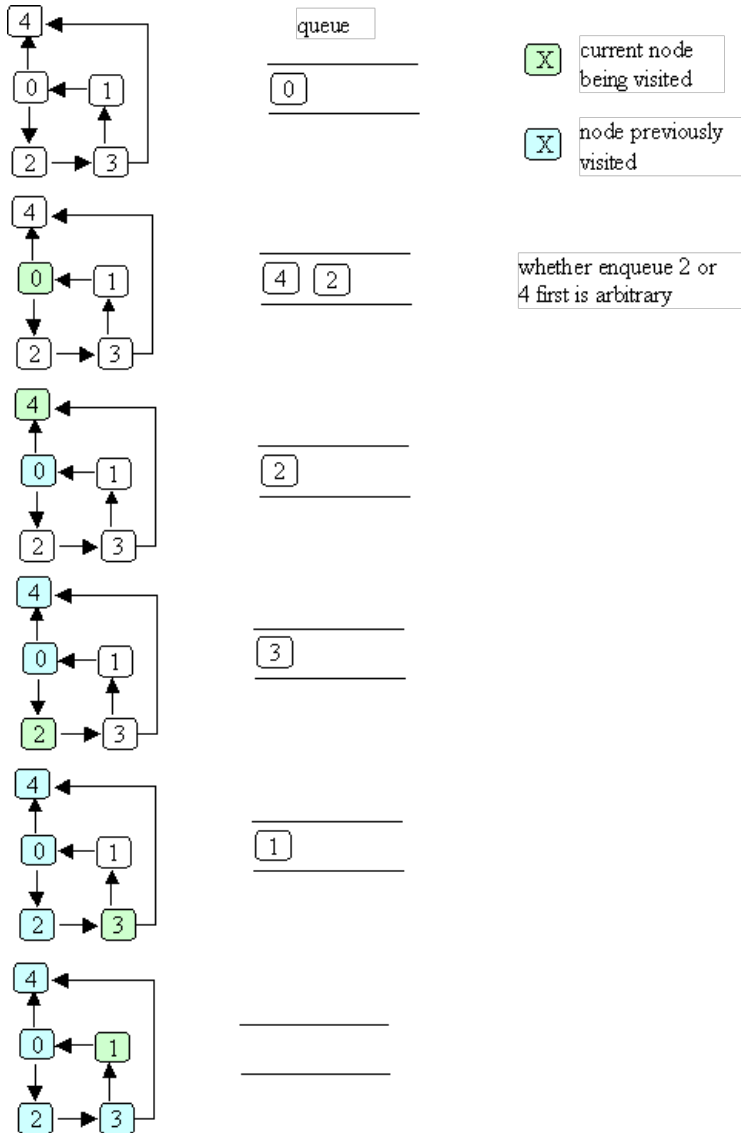
search gives you a level-order traversal. Here's the pseudo code:

```
public void bfs(Graphnode<T> n) {
    Queue<Graphnode> queue = new Queue<Graphnode>();

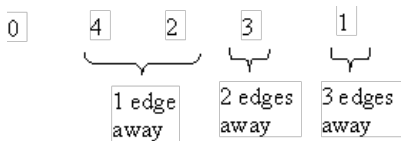
    n.setVisited(true);
    queue.enqueue( n );
    while (!queue.isEmpty()) {
        Graphnode<T> current = queue.dequeue();
        for (Graphnode<T> k : current.getSuccessors()) {

            if (!k.getVisited()){
                k.setVisited(true);
                queue.enqueue(k);
            } // end if k not visited
        } // end for every successor k
    } // end while queue not empty
}
```

Here's the same example graph we used for depth-first search and an illustration of breadth-first search starting with node 0:



The order in which nodes are "visited" as a result of `bfs(0)` is:



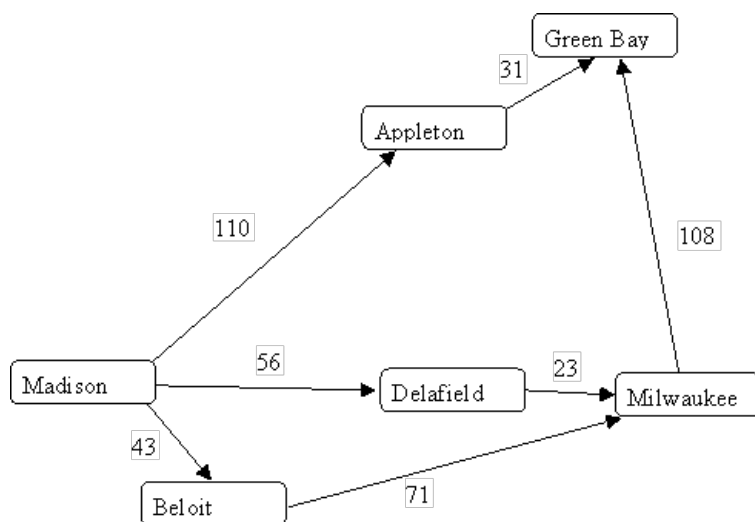
As with depth-first search, all nodes marked `VISITED` are reachable from the start node, but nodes are visited in a different order than they would be using depth-first search.

We can use a variation of `bfs` to find the shortest distance (the length of the shortest path) to each reachable node:

- add a distance field to each node

- when `bfs` is called with node `n`, set `n`'s distance to zero
- when a node `k` is about to be enqueued, set `k`'s distance to the distance of the current node (the one that was just dequeued) + 1

This technique only works in **unweighted** graphs (i.e., in graphs in which all edges are assumed to have length 1). An interesting problem is how to find shortest paths in a weighted graph; i.e., given a "start" node `n`, to find, for each other node `m`, the path from `n` to `m` for which the sum of the weights on the edges is minimal (assuming that no edge has a negative weight). For example, in the following graph, nodes represent cities, edges represent highways, and the weights on the edges represent distances (the length of the highway between the two cities). Breadth-first search can only tell you which route from Madison to Green Bay goes through the fewest other cities; it cannot tell you which route is the shortest.



## Dijkstra's algorithm

A clever algorithm that *can* be used to solve this problem (to find shortest paths in a weighted graph with non-negative edge weights) has been defined by Edsger Dijkstra (and so is called "**Dijkstra's algorithm**"). The worst-case running time of the algorithm is  $O(E \log N)$ , where  $E$  is the number of edges and  $N$  is the number of nodes. The details of the algorithm are covered in lecture (and may be expanded upon here in this reading - check back later for more details).

## Summary

- A **graph** is a set of **nodes** and a set of **edges**.
- There are two kinds of graphs: **directed** and **undirected**.
- High-level operations include:
  - **depth-first search**, which can be done on the *entire* graph (e.g., to find cycles or to produce a topological ordering) or on part of a graph (e.g., to determine which nodes are reachable from a given node)
  - **breadth-first search**, which can also be used to determine reachability and can be used to find shortest paths in unweighted graphs
  - **Dijkstra's algorithm**, which finds shortest paths in weighted graphs.



# Sorting

## Contents

- [Introduction](#)
- [Selection Sort](#)
  - [Test Yourself #1](#)
- [Insertion Sort](#)
  - [Test Yourself #2](#)
- [Merge Sort](#)
  - [Test Yourself #3](#)
  - [Test Yourself #4](#)
- [Quick Sort](#)
  - [Test Yourself #5](#)
- [Heap Sort](#)
- [Radix Sort](#)
- [Sorting Summary](#)

## Introduction

Consider sorting the values in an array  $A$  of size  $N$ . Most sorting algorithms involve what are called **comparison sorts**, i.e., they work by comparing values. Comparison sorts can never have a worst-case running time less than  $O(N \log N)$ . Simple comparison sorts are usually  $O(N^2)$ ; the more clever ones are  $O(N \log N)$ .

Three interesting issues to consider when thinking about different sorting algorithms are:

- Does an algorithm always take its worst-case time?
- What happens on an already-sorted array?
- How much space (other than the space for the array itself) is required?

We will discuss four comparison-sort algorithms:

1. selection sort
2. insertion sort
3. merge sort
4. quick sort

Selection sort and insertion sort have worst-case time  $O(N^2)$ . Quick sort is also  $O(N^2)$  in the worst case, but its expected time is  $O(N \log N)$ . Merge sort is  $O(N \log N)$  in the worst case.

## Selection Sort

The idea behind selection sort is:

1. Find the smallest value in  $A$ ; put it in  $A[0]$ .
2. Find the second smallest value in  $A$ ; put it in  $A[1]$ .
3. etc.

The approach is as follows:

- Use an outer loop from 0 to  $N-1$  (the loop index,  $k$ , tells which position in  $A$  to fill next).
- Each time around, use a nested loop (from  $k+1$  to  $N-1$ ) to find the smallest value (and its index) in the unsorted part of the array.
- Swap that value with  $A[k]$ .

Note that after  $i$  iterations,  $A[0]$  through  $A[i-1]$  contain their final values (so after  $N$  iterations,  $A[0]$  through  $A[N-1]$  contain their final values and we're done!)

Here's the code for selection sort:

```
public static <E extends Comparable<E>> void selectionSort(E[] A) {
    int j, k, minIndex;
    E min;
    int N = A.length;

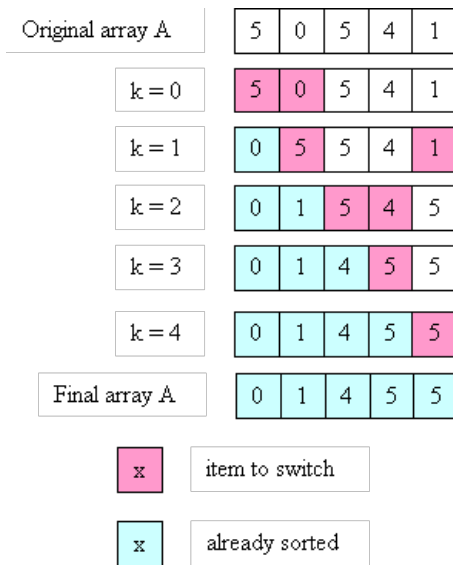
    for (k = 0; k < N; k++) {
        min = A[k];
        minIndex = k;
        for (j = k+1; j < N; j++) {
            if (A[j].compareTo(min) < 0) {
                min = A[j];
                minIndex = j;
            }
        }
        swap(A, k, minIndex);
    }
}
```

```

    }
}
A[minIndex] = A[k];
A[k] = min;
}

```

and here's a picture illustrating how selection sort works:



What is the time complexity of selection sort? Note that the inner loop executes a different number of times each time around the outer loop, so we can't just multiply  $N * (\text{time for inner loop})$ . However, we can notice that:

- 1st iteration of outer loop: inner executes  $N - 1$  times
- 2nd iteration of outer loop: inner executes  $N - 2$  times
- ...
- Nth iteration of outer loop: inner executes 0 times

This is our old favorite sum:

$$N-1 + N-2 + \dots + 3 + 2 + 1 + 0$$

which we know is  $O(N^2)$ .

What if the array is already sorted when selection sort is called? It is still  $O(N^2)$ ; the two loops still execute the same number of times, regardless of whether the array is sorted or not.

## TEST YOURSELF #1

It is not necessary for the outer loop to go all the way from 0 to  $N-1$ . Describe a small change to the code that avoids a small amount of unnecessary work.

Where else might unnecessary work be done using the current code? (Hint: think about what happens when the array is already sorted initially.) How could the code be changed to avoid that unnecessary work? Is it a good idea to make that change?

[solution](#)

## Insertion Sort

The idea behind insertion sort is:

1. Put the first 2 items in correct relative order.
2. Insert the 3rd item in the correct place relative to the first 2.
3. Insert the 4th item in the correct place relative to the first 3.
4. etc.

As with selection sort, a nested loop is used; however, a different invariant holds: after the  $i$ -th time around the outer loop, the items in  $A[0]$  through  $A[i-1]$  are in order relative to each other (but are not necessarily in their final places). Also, note that in order to insert an item into its place in the (relatively) sorted part of the array, it is necessary to move some values to the right to make room.

Here's the code:

```

public static <E extends Comparable<E>> void insertionSort(E[] A) {
    int k, j;

```

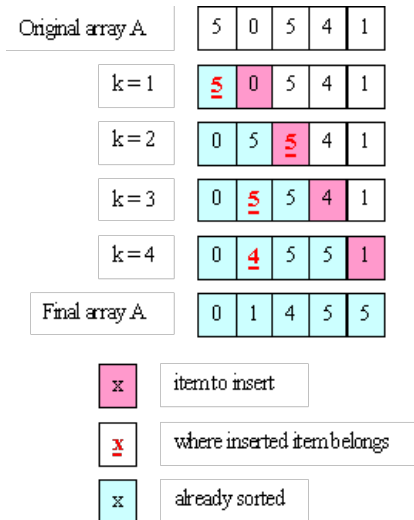
```

E tmp;
int N = A.length;

for (k = 1; k < N; k++) {
    tmp = A[k];
    j = k - 1;
    while ((j >= 0) && (A[j].compareTo(tmp) > 0)) {
        A[j+1] = A[j]; // move one value over one place to the right
        j--;
    }
    A[j+1] = tmp;      // insert kth value in correct place relative
                      // to previous values
}
}

```

Here's a picture illustrating how insertion sort works on the same array used above for selection sort:



What is the time complexity of insertion sort? Again, the inner loop can execute a different number of times for every iteration of the outer loop. In the *worst* case:

- 1st iteration of outer loop: inner executes 1 time
- 2nd iteration of outer loop: inner executes 2 times
- 3rd iteration of outer loop: inner executes 3 times
- ...
- N-1st iteration of outer loop: inner executes N-1 times

So we get:

$$1 + 2 + \dots + N-1$$

which is still  $O(N^2)$ .

## TEST YOURSELF #2

**Question 1:** What is the running time for insertion sort when:

1. the array is already sorted in ascending order?
2. the array is already sorted in descending order?

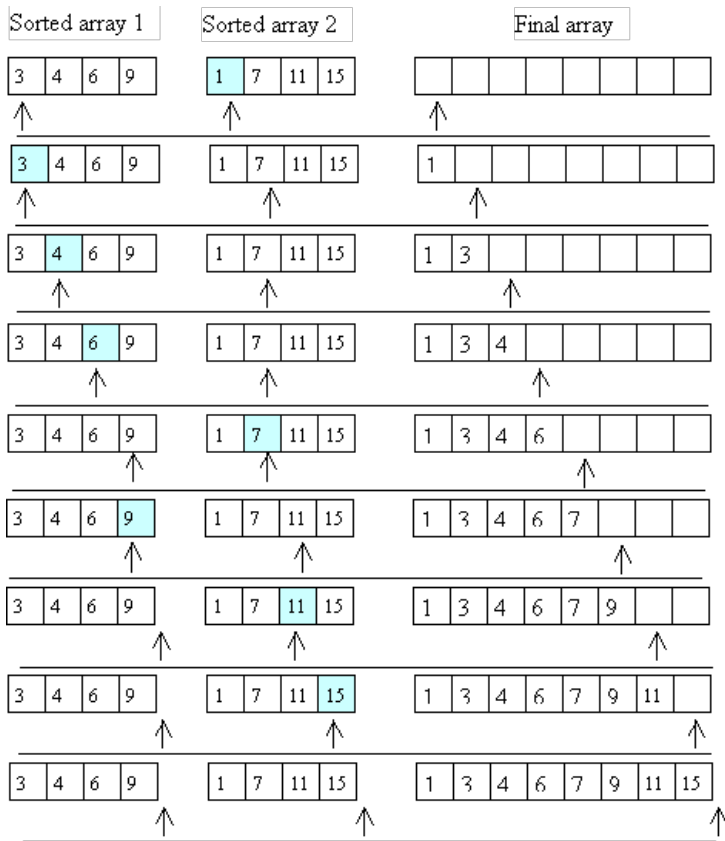
**Question 2:** On each iteration of its outer loop, insertion sort finds the correct place to insert the next item, relative to the ones that are already in sorted order. It does this by searching back through those items, one at a time. Would insertion sort be speeded up if instead it used binary search to find the correct place to insert the next item?

[solution](#)

## Merge Sort

As mentioned above, merge sort takes time  $O(N \log N)$ , which is quite a bit better than the two  $O(N^2)$  sorts described above (for example, when  $N=1,000,000$ ,  $N^2=1,000,000,000,000$ , and  $N \log_2 N = 20,000,000$ ; i.e.,  $N^2$  is 50,000 times larger than  $N \log N$ !).

The key insight behind merge sort is that it is possible to *merge* two sorted arrays, each containing  $N/2$  items to form one sorted array containing  $N$  items in time  $O(N)$ . To do this merge, you just step through the two arrays, always choosing the smaller of the two values to put into the final array (and only advancing in the array from which you took the smaller value). Here's a picture illustrating this merge process:



Now the question is, how do we get the two sorted arrays of size  $N/2$ ? The answer is to use recursion; to sort an array of length  $N$ :

1. Divide the array into two halves.
2. Recursively, sort the left half.
3. Recursively, sort the right half.
4. Merge the two sorted halves.

The base case for the recursion is when the array to be sorted is of length 1 -- then it is already sorted, so there is nothing to do. Note that the merge step (step 4) needs to use an auxiliary array (to avoid overwriting its values). The sorted values are then copied back from the auxiliary array to the original array.

An outline of the code for merge sort is given below. It uses an auxiliary method with extra parameters that tell what part of array A each recursive call is responsible for sorting.

```
public static <E extends Comparable<E>> void mergeSort(E[] A) {
    mergeAux(A, 0, A.length - 1); // call the aux. function to do all the work
}

private static <E extends Comparable<E>> void mergeAux(E[] A, int low, int high) {
    // base case
    if (low == high) return;

    // recursive case

    // Step 1: Find the middle of the array (conceptually, divide it in half)
    int mid = (low + high) / 2;

    // Steps 2 and 3: Sort the 2 halves of A
    mergeAux(A, low, mid);
    mergeAux(A, mid+1, high);

    // Step 4: Merge sorted halves into an auxiliary array
    E[] tmp = (E[]) (new Comparable[high-low+1]);
    int left = low; // index into left half
    int right = mid+1; // index into right half
    int pos = 0; // index into tmp

    while ((left <= mid) && (right <= high)) {
        // choose the smaller of the two values "pointed to" by left, right
        // copy that value into tmp[pos]
        // increment either left or right as appropriate
        // increment pos
        ...
    }

    // when one of the two sorted halves has "run out" of values, but
    // there are still some in the other half, copy all the remaining
    // values to tmp
}
```

```

// Note: only 1 of the next 2 loops will actually execute
while (left <= mid) { ... }
while (right <= high) { ... }

// all values are in tmp; copy them back into A
arraycopy(tmp, 0, A, low, tmp.length);
}

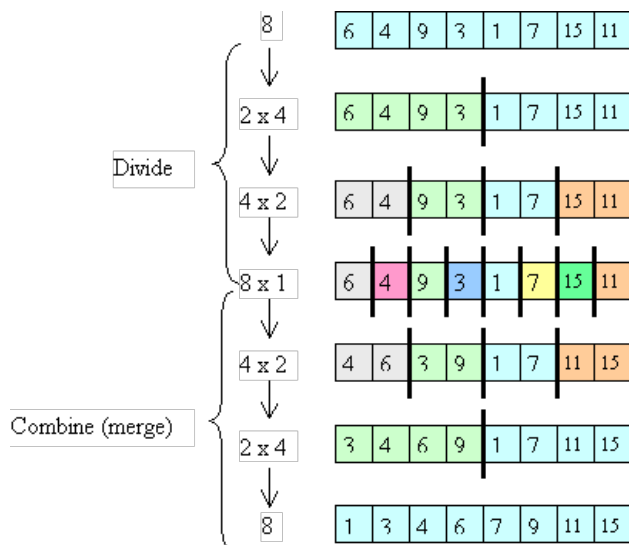
```

### TEST YOURSELF #3

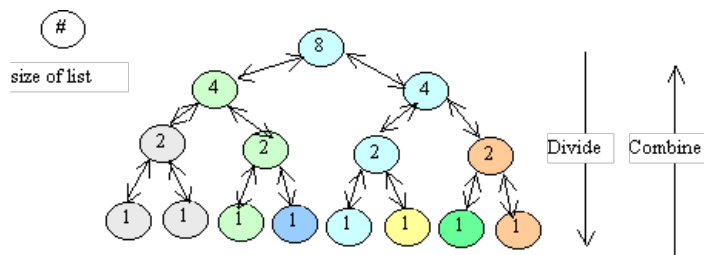
Fill in the missing code in the `mergeSort` method.

[solution](#)

Algorithms like merge sort -- that work by dividing the problem in two, solving the smaller versions, and then combining the solutions -- are called **divide-and-conquer** algorithms. Below is a picture illustrating the divide-and-conquer aspect of merge sort using a new example array. The picture shows the problem being divided up into smaller and smaller pieces (first an array of size 8, then two halves each of size 4, etc.). Then it shows the "combine" steps: the solved problems of half size are merged to form solutions to the larger problem. (Note that the picture illustrates the conceptual ideas -- in an actual execution, the small problems would be solved one after the other, not in parallel. Also, the picture doesn't illustrate the use of auxiliary arrays during the merge steps.)



To determine the time for merge sort, it is helpful to visualize the calls made to `mergeAux` as shown below (each node represents one call and is labeled with the size of the array to be sorted by that call):



The height of this tree is  $O(\log N)$ . The total work done at each "level" of the tree (i.e., the work done by `mergeAux` excluding the recursive calls) is  $O(N)$ :

- Step 1 (finding the middle index) is  $O(1)$ , and this step is performed once in each call, i.e., a total of once at the top level, twice at the second level, etc., down to a total of  $N/2$  times at the second-to-last level (it is not performed at all at the very last level, because there the base case applies, and `mergeAux` just returns). So for any one level, the total amount of work for Step 1 is at most  $O(N)$ .
- For each individual call, Step 4 (merging the sorted half-arrays) takes time proportional to the size of the part of the array to be sorted by that call. So for a whole level, the time is proportional to the sum of the sizes at that level. This sum is always  $N$ .

Therefore, the time for merge sort involves  $O(N)$  work done at each "level" of the tree that represents the recursive calls. Since there are  $O(\log N)$  levels, the total worst-case time is  $O(N \log N)$ .

### TEST YOURSELF #4

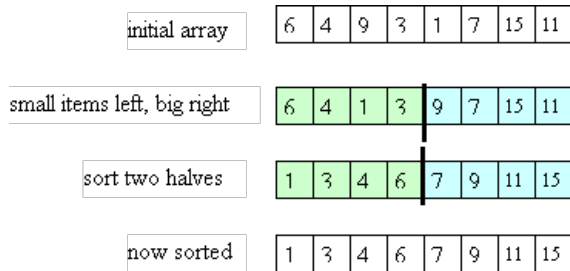
What happens when the array is already sorted (what is the running time for merge sort in that case)?

[solution](#)

## Quick Sort

Quick sort (like merge sort) is a divide-and-conquer algorithm: it works by creating two problems of half size, solving them recursively, then combining the solutions to the small problems to get a solution to the original problem. However, quick sort does more work than merge sort in the "divide" part and is thus able to avoid doing any work at all in the "combine" part!

The idea is to start by **partitioning** the array: putting all small values in the left half and putting all large values in the right half. Then the two halves are (recursively) sorted. Once that's done, there's no need for a "combine" step: the whole array will be sorted! Here's a picture that illustrates these ideas:



The key question is how to do the partitioning? Ideally, we'd like to put exactly half of the values in the left part of the array and the other half in the right part, i.e., we'd like to put all values less than the **median** value in the left and all values greater than the median value in the right. However, that requires first computing the median value (which is too expensive). Instead, we pick one value to be the **pivot** and we put all values less than the pivot to its left and all values greater than the pivot to its right (the pivot itself is then in its final place). Copies of the pivot value can go either to its left or to its right.

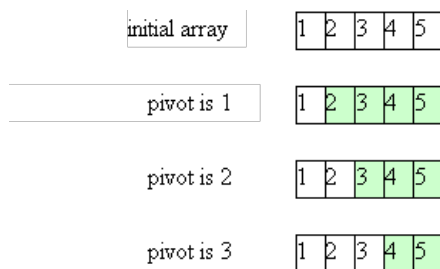
Here's the algorithm outline:

1. Choose a pivot value.
2. Partition the array (put all value less than the pivot in the left part of the array, then the pivot itself, then all values greater than the pivot). Copies of the pivot value can go in either part of the array.
3. Recursively, sort the values less (or equal to) than the pivot.
4. Recursively, sort the values greater than (or equal to) the pivot.

Note that, as for merge sort, we need an auxiliary method with two extra parameters -- low and high indexes to indicate which part of the array to sort. Also, although we could "recurse" all the way down to a single item, in practice, it is better to switch to a sort like insertion sort when the number of items to be sorted is small (e.g., 20).

Now let's consider how to choose the pivot item. (Remember our goal is to choose it so that the "left part" and "right part" of the array have about the same number of items -- otherwise we'll get a bad runtime.).

An easy thing to do is to use the first value --  $A[\text{low}]$  -- as the pivot. However, if  $A$  is already sorted this will lead to the worst possible runtime, as illustrated below:



In this case, after partitioning, the left part of the array is empty and the right part contains all values except the pivot. This will cause  $O(N)$  recursive calls to be made (to sort from 0 to  $N-1$ , then from 1 to  $N-1$ , then from 2 to  $N-1$ , etc.). Therefore, the total time will be  $O(N^2)$ .

Another option is to use a random-number generator to choose a random item as the pivot. This is OK if you have a good, fast random-number generator.

A simple and effective technique is the "**median-of-three**": choose the median of the values in  $A[\text{low}]$ ,  $A[\text{high}]$ , and  $A[(\text{low}+\text{high})/2]$ . Note that this requires that there be at least 3 items in the array, which is consistent with the note above about using insertion sort when the piece of the array to be sorted gets small.

Once we've chosen the pivot, we need to do the partitioning. (The following assumes that the size of the piece of the array to be sorted is greater than 3.) The basic idea is to use two "pointers" (indexes), left and right. They start at opposite ends of the array and move toward each other until left "points" to an item that is greater than the pivot (so it doesn't belong in the left part of the array) and right "points" to an item that is smaller than the pivot. Those two "out-of-place" items are swapped and we repeat this process until left and right cross:

1. Choose the pivot (using the "median-of-three" technique); also, put the smallest of the 3 values in  $A[\text{low}]$ , put the largest of the 3 values in  $A[\text{high}]$ , and swap the pivot with the value in  $A[\text{high}-1]$ . (Putting the smallest value in  $A[\text{low}]$  prevents right from falling off the end of the array in the following steps.)
2. Initialize: left =  $\text{low}+1$ ; right =  $\text{high}-2$
3. Use a loop with the condition:

while (left <= right)

The loop invariant is:

all items in A[low] to A[left-1] are  $\leq$  the pivot  
all items in A[right+1] to A[high] are  $\geq$  the pivot

Each time around the loop:

left is incremented until it "points" to a value  $>$  the pivot  
right is decremented until it "points" to a value  $<$  the pivot  
if left and right have not crossed each other,  
then swap the items they "point" to.

#### 4. Put the pivot into its final place.

Here's the actual code for the partitioning step (the reason for returning a value will be clear when we look at the code for quick sort itself):

```
private static <E extends Comparable<E>> int partition(E[] A, int low, int high) {
    // precondition: A.length > 3

    E pivot = medianOfThree(A, low, high); // this does step 1
    int left = low+1; right = high-2;
    while ( left <= right ) {
        while (A[left].compareTo(pivot) < 0) left++;
        while (A[right].compareTo(pivot) > 0) right--;
        if (left <= right) {
            swap(A, left, right);
            left++;
            right--;
        }
    }
    swap(A, left, high-1); // step 4
    return right;
}
```

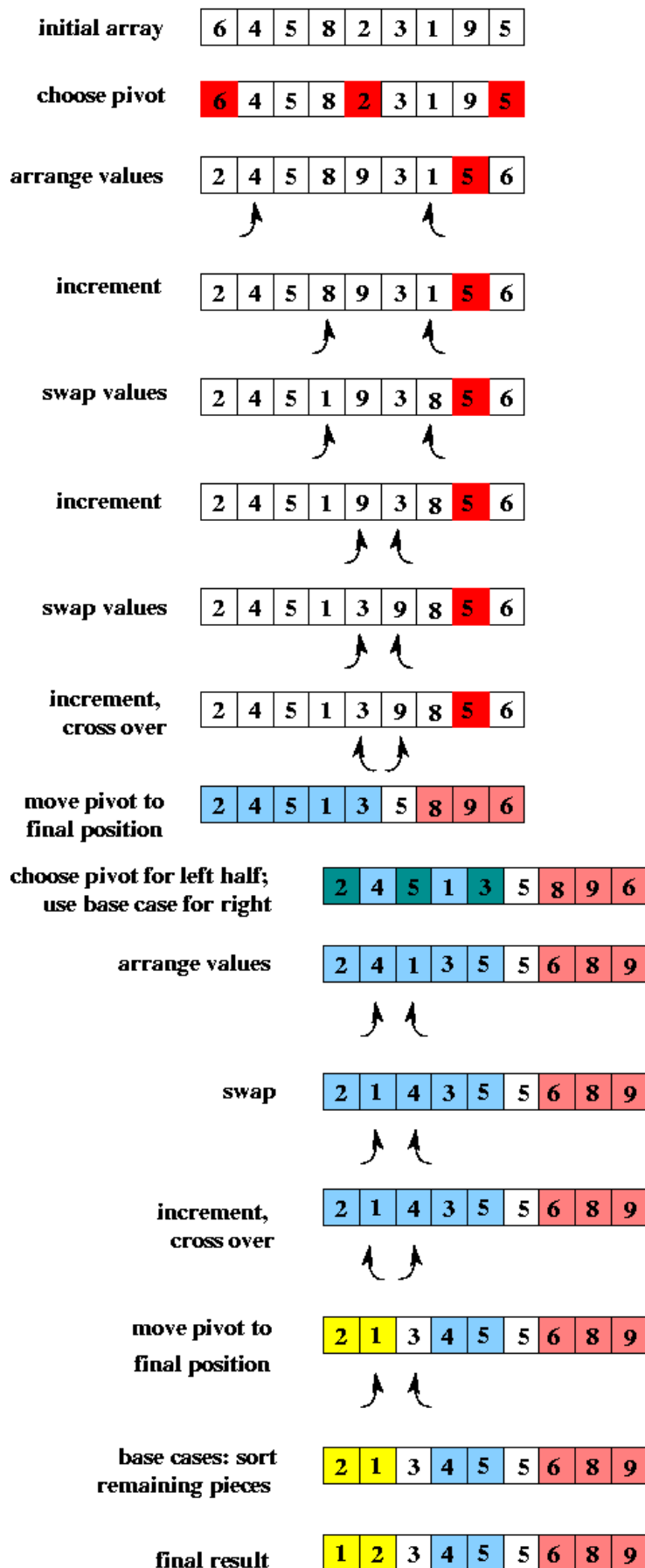
After partitioning, the pivot is in A[right+1], which is its final place; the final task is to sort the values to the left of the pivot and to sort the values to the right of the pivot. Here's the code for quick sort (so that we can illustrate the algorithm, we use insertion sort only when the part of the array to be sorted has less than 4 items, rather than when it has less than 20 items):

```
public static <E extends Comparable<E>> void quickSort(E[] A) {
    quickAux(A, 0, A.length-1);
}

private static <E extends Comparable<E>> void quickAux(E[] A, int low, int high) {
    if (high-low < 4) insertionSort(A, low, high);
    else {
        int right = partition(A, low, high);
        quickAux(A, low, right);
        quickAux(A, right+2, high);
    }
}
```

Note: If the array might contain a lot of duplicate values, then it is important to handle copies of the pivot value efficiently. In particular, it is not a good idea to put all values strictly less than the pivot into the left part of the array and all values greater than or equal to the pivot into the right part of the array. The code given above for partitioning handles duplicates correctly.

Here's a picture illustrating quick sort:



What is the time for quick sort?

- If the pivot is always the median value, then the calls form a balanced binary tree (like they do for merge sort).
- In the worst case (the pivot is the smallest or largest value) the calls form a "linear" tree.
- In any case, the total work done at each level of the call tree is  $O(N)$  for partitioning.

So the total time is:

- worst-case:  $O(N^2)$



- in practice:  $O(N \log N)$

Note that quick sort's worst-case time is worse than merge sort's. However, an advantage of quick sort is that it does not require extra storage, as merge sort does.

## TEST YOURSELF #5

What happens when the array is already sorted (what is the running time for quick sort in that case, assuming that the "median-of-three" method is used to choose the pivot)?

[solution](#)

## Heap Sort

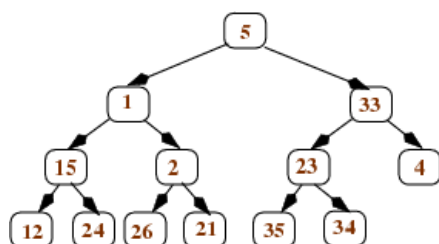
As we know from studying priority queues, we can use a heap to perform operations insert and remove max in time  $O(\log N)$ , where  $N$  is the number of values in the heap. Therefore, we could sort an array of  $N$  items as follows:

1. Insert each item into an initially empty heap.
2. Fill in the array, right-to-left as follows:

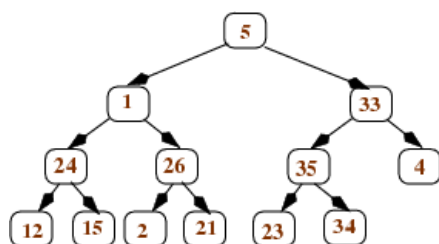
while the heap is not empty: do one removeMax operation and put the returned value into the next position of the array

It takes  $O(N \log N)$  time to add  $N$  items to an initially empty heap, and it takes  $O(N \log N)$  time to do  $N$  removeMax operations. Therefore, the total time to sort the array this way is  $O(N \log N)$ . We can do slightly better as well as avoiding the need for a separate data structure (for the heap) by using a special heapify operation, that starts with an unordered array of  $N$  items, and turns it into a heap (containing the same items) in time  $O(N)$ . Once the array is a heap, we can still fill in the array right-to-left without using any auxiliary storage: each removeMax operation frees one more space at the end of the array, so we can simply swap the value in the root with the value in the last leaf, and we will have put that root value in its final place. (Of course, we wouldn't use our usual trick of leaving position zero empty, but that was just to make the presentation simpler: if the root of the heap is in position 0, we just need to subtract 1 when computing the indexes of a node's children or of its parent in the tree).

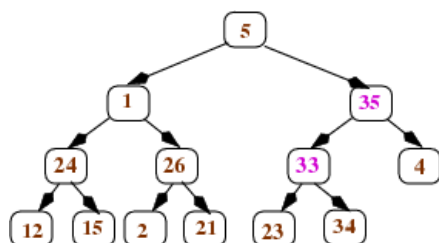
It is easiest to understand the heapify operation in terms of the tree represented by the array, but it can actually be performed on the array itself. The idea is to work bottom-up in the tree, turning each subtree into a heap. We will illustrate the process using the tree shown below.



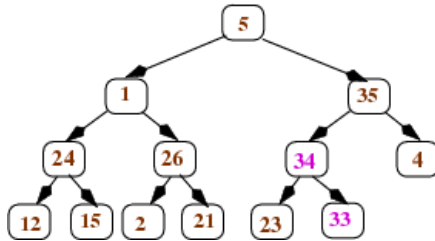
We start with the nodes that are parents of leaves. For each such node  $n$ , we compare  $n$ 's value to the values in its children, and swap  $n$ 's value with its larger child if necessary. After doing this, each node that is the parent of leaves is now the root of a (small) heap. Here's what happens to the tree shown above after we heapify all nodes that are parents of leaves:



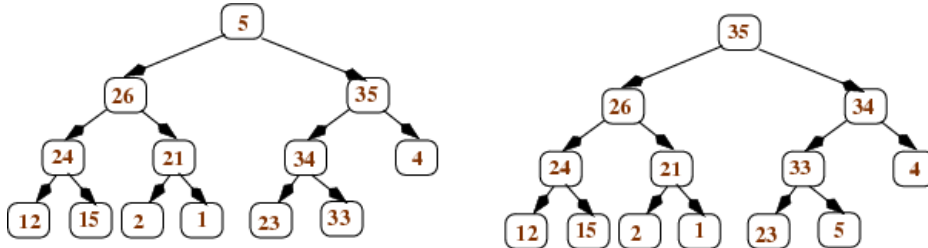
Then we continue to work our way up the tree: After applying heapify to all nodes at depth  $d$ , we apply it to the nodes at depth  $d-1$ . Each such node  $n$  will have 2 children, and those children will be the roots of heaps. If  $n$ 's value is greater than the values in its children, we're done:  $n$  is the root of a heap. If not, we swap  $n$ 's value with its larger child, and continue to swap down as needed (just like we do for the removeMax operation after moving the last leaf to the root). This happens in our example tree when we apply heapify to the right child of the root (the node containing the value 33). We swap that value with the larger child (35) to get this tree (with the changed values in pink):



And then we swap the 33 with its larger child (34):



Once we apply heapify to the root, we're done: the whole tree is a heap. Here are the trees that result from applying heapify first to the left child of the root and then to the root itself:



It should be clear that, in the worst case, each heapify operation takes time proportional to the height of the node to which it is applied. Therefore, the total time for turning an array of size  $N$  into a heap is the sum of the heights of all nodes:

$$\text{total time} = \text{sum from } i=1 \text{ to } N \text{ of height}(i)$$

where the  $i$ 's are the nodes of the tree. This is equivalent to

$$\text{sum from } k=1 \text{ to } \log N \text{ of } k * (N/2^k)$$

Here,  $k$  is  $\text{height}(i)$  and  $(N/2^k)$  is the maximum number of nodes at that height. For example,  $k$  is 1 for all leaves and there are at most  $N/2$  leaves;  $k$  is  $\log N$  for the root and there is at most 1 root. We can convert as follows:

$$= \text{sum from } k=1 \text{ to } \log N \text{ of } k * (N/2^k) = N * \text{sum from } k=1 \text{ to } \log N \text{ of } (k/2^k)$$

The final sum is bounded by a constant, so the whole thing is just  $O(N)$ ! One way to get some intuition for why it is just  $O(N)$  is to consider that most nodes have small heights: half of the nodes (the leaves) have height 1; another half of the remaining nodes have height 2, and so on.

The total time for heap sort is still  $O(N \log N)$ , because it takes time  $O(N \log N)$  to perform the  $N$  removeMax operations, but using heapify does speed up the sort somewhat.

## Radix Sort

So far, the sorts we have been considering have all been **comparison sorts** (which can never have a worst-case running time less than  $O(N \log N)$ ). We will now consider a sort that is **not** a comparison sort: radix sort. Radix sort is useful when the values to be sorted are fairly short sequences of comparable values. For example, radix sort can be used to sort numbers (sequences of digits) or strings (sequences of characters). The time for radix sort is  $O((N + \text{range}) * \text{len})$ , where

- $N$  is the number of sequences to be sorted
- $\text{range}$  is the number of values each item in the sequence could have
- $\text{len}$  is the (maximum) length of the sequences.

For example, sorting 100 4-digit integers would take time  $(100 + 10) * 4$ , and sorting 1000 10-character words (where each word contains only lower-case letters) would take time  $(1000 + 26) * 10$ .

There are a number of variations on radix sort. The one presented here uses an auxiliary array of queues and works by processing each sequence right-to-left (least significant "digit" to most significant). On each pass, the values are taken from the original array and stored in a queue in the auxiliary array based on the value of the current "digit". Then the queues are dequeued back into the original array, ready for the next pass.

On the first pass, the position in the auxiliary array is determined by looking at the rightmost "digit" and each sequence is put into the queue in that position of the array. For example, suppose we want to sort the following original array:

[132, 355, 104, 327, 111, 285, 391, 543, 123, 535]

We would use an auxiliary array of size 10 (since each digit can range in value from 0 to 9). After the first pass, the array of queues would look like this (where the array is shown vertically, indicated by position numbers, and the fronts of the queues are to the left):

```
0:
1: 111, 391
2: 132
```

```
3: 543, 123
4: 104
5: 355, 285, 535
6:
7: 327
8:
9:
```

Then we would make one pass through the auxiliary array, putting all values back into the original array:

```
[111, 391, 132, 543, 123, 104, 355, 285, 535, 327]
```

On the next pass, the position in the auxiliary array is determined by looking at the second-to-right "digit":

```
0: 104
1: 111
2: 123, 327
3: 132, 535
4: 543
5: 355
6:
7:
8: 285
9: 391
```

Values are again put back into the original array:

```
[104, 111, 123, 327, 132, 535, 543, 355, 285, 391]
```

and then the final pass chooses array position based on the leftmost digit (since we are sorting sequences of digits of length 3):

```
0:
1: 104, 111, 123, 132
2: 285
3: 327, 355, 391
4:
5: 535, 543
6:
7:
8:
9:
```

Putting the values back, we have a sorted array:

```
[104, 111, 123, 132, 285, 327, 355, 391, 535, 543]
```

Each pass of radix sort takes time  $O(N)$  to put the values being sorted into the correct queue, and time  $O(N + range)$  to put the values back into the original array. The number of passes is equal to the number of "digits" in each value, so the total time is  $O((N + range) * len)$ . This is often better than an  $O(N \log N)$  sort. For example,  $\log_2 1,000,000$  is about 20, so if we want to sort 1,000,000 numbers in the range 0 to 1,000,000, we have

- $(N + range) * len = (1,000,000 + 10) * 7 = 7,000,070$
- $N \log N = 1,000,000 * 20 = 20,000,000$

## Sorting Summary

### Selection Sort:

- $N$  passes  
on pass  $k$ : find the  $k$ -th smallest item, put it in its final place
- always  $O(N^2)$

### Insertion Sort:

- $N$  passes  
on pass  $k$ : insert the  $k$ -th item into its proper position relative to the items to its left
- worst-case  $O(N^2)$
- given an already-sorted array:  $O(N)$

### Merge Sort:

- recursively sort the first  $N/2$  items  
recursively sort the last  $N/2$  items  
merge (using an auxiliary array)
- always  $O(N \log N)$

### Quick Sort:

- choose a pivot value  
partition the array:

left part has items  $\leq$  pivot  
right part has items  $\geq$  pivot

recursively sort the left part  
recursively sort the right part

- worst-case  $O(N^2)$
- expected  $O(N \log N)$

### Heap Sort:

- use heapify to convert the unsorted array into a heap, then do N removeMax operations. Each operation frees one more space at the end of the array; put the returned max value into that space.
- always  $O(N \log N)$

### Radix Sort:

- make  $len$  passes through the N sequences to be sorted, right-to-left  
on each pass, put the values into the queue in position p of the auxiliary array, where p is the value of the current "digit"  
then put the values back from the auxiliary array into the original array
- no comparisons of values are done (i.e., radix sort is not a comparison sort).
- always  $O(N + range) * len$

## CS 367 Links

### Main

[Home](#)  
[Syllabus](#)

### Course Work

[Grading](#)  
[Exams](#)  
[Programs](#)  
[Homeworks](#)  
[Submission](#)

### Resources

[Piazza](#)  
[CS Computer Labs](#)  
[Remote Access](#)  
[New to Linux?](#)

[Learn@UW](#)

[CSL](#)  
[UW CS Department](#)

### Guides

[CS 302 Style](#)  
[CS 302 Commenting](#)

### Eclipse

[Eclipse Home](#)  
[CS 302 Tutorial](#)  
[CS 302 Debugging Lab](#)  
[CS 302 Download Info](#)

### Java 8

[Java SE](#)  
[API Specs](#)  
[CS 302 Download Info](#)

# CS 367: Introduction to Data Structures

## SYLLABUS

[Current Week](#) | [Midterm 1](#) | [Midterm 2](#) | [Final](#)

### NOTE:

This syllabus will be updated as the semester progresses - check it regularly.

### Week 1: ADTs, Interfaces, Java Objects and Generics, Array-based Lists

*Readings:* [Introduction](#), [Lists](#)

*Outlines:* [Tuesday](#), [Thursday](#)

### Week 2: Array-based Lists (cont.), Iterators

*Readings:* [Lists](#)

*Outlines:* [Tuesday](#), [Thursday](#)

*Code:* [ListADT.html](#), [ListADT.java](#)

*Java:* [List interface](#), [ArrayList](#), [Iterable interface](#), [Iterator interface](#)

*Programs:* [p1](#) assigned 1/29

*Homeworks:* [h1](#) assigned 1/30

### Week 3: Exceptions Review, Primitives vs. References Review

*Readings:* [Exceptions](#)

*Outlines:* [Tuesday](#), [Thursday](#)

*Homeworks:* [h1](#) due 10:00 pm on Friday, 2/6; [h2](#) assigned 2/6

### Week 4: Linked Lists

*Readings:* [Linked Lists](#)

*Outlines:* [Tuesday](#), [Thursday](#)

*Homeworks:* [h2](#) due 10:00 pm on Friday, 2/13; [h3](#) assigned 2/13

*Programs:* [p1](#) due 10:00 pm on Sunday, 2/15

### Week 5: Linked Lists (cont.), Complexity

*Readings:* [Linked Lists](#), [Complexity](#)

*Outlines:* [Tuesday](#), [Thursday](#)

*Homeworks:* [h3](#) due 10:00 pm on Friday, 2/20; [h4](#) assigned 2/20

*Programs:* [p2](#) assigned 2/15

### Week 6: Complexity (cont.), Stacks and Queues

*Readings:* [Complexity](#), [Stacks-and-Queues](#)

*Outlines:* [Tuesday](#), [Thursday](#)

*Homeworks:* [h4](#) due 10:00 pm on Friday, 2/27

### Week 7: Recursion

*Readings:* [Recursion](#)

*Outlines:* [Tuesday](#), [Thursday](#)

**Exam: Midterm 1**, Tuesday, March 3rd, 5:00 PM to 7:00 PM

*Homeworks:* [h5](#) assigned 3/6

*Programs:* [p2](#) due 10:00 pm on Sunday, 3/8

### Week 8: Recursion (cont.), Search

*Readings:* [Recursion](#), [Searching](#)

*Outline:* [Tuesday](#)

*Homeworks:* [h5](#) due 10:00 pm on Friday, 3/13; [h6](#) assigned 3/13

### Week 9: General Trees, Binary Trees, Binary Search Trees

*Readings:* [Trees](#), [Binary-Search-Trees](#)

### Week 10: Binary Search Trees (cont.), Balanced Search Trees, Red-Black Trees

*Readings:* [Binary-Search-Trees](#), [Red-Black-Trees](#)

### Spring Break 3/28 - 4/5

### Week 11: Priority Queues, Heaps, Hashing

*Readings:* [Priority Queues](#), [Hashing](#)

### Week 12: Hashing (cont.)

*Readings:* [Hashing](#)

**Exam: Midterm 2**, Tuesday, April 14th, 5:00 PM to 7:00 PM

**Week 13: Graphs**  
*Readings:* [Graphs](#)

**Week 14: Graphs (cont.)**  
*Readings:* [Graphs](#)

**Week 15: Sorting**  
*Readings:* [Sorting](#)

**Week 16: Exam: Final**, Wednesday, May 13th, 5:05 PM to 7:05 PM

**Last updated:** 3/9/2015 ©2008-2015 Jim Skrentny (cgi by Dalibor Zelený)