Arrays in Java

We looked at the basics of Java's built-in arrays, starting from creating an array to iterating over the elements of an array, as well as at a few typical array manipulations in class today. See the link to a complete implementation for you to play with at the end of this document.

Topics, in no particular order:

Creating an array

In Java, you create a new array in the following:

```
Foo[] x = new Foo[100];
```

Now `x` refers to an array of 100 `Foo` references. Note that the array doesn't actually contain the instances of `Foo` (the objects that is), but rather hold the references to the objects. We can assign other references to this array:

```
Foo[] y = x;
```

Now `y` is just another reference to the same array, and any change made through `y` will change the array referred to by `x`.

A Java array has a single attribute called `length` that gives you the capacity of the array; `x.length` for example would produce the value 100 in this case. The capacity of an array is fixed, and once created, cannot be changed. We "resize" and array by creating a new one with higher capacity, and copying the elements to the new one. See resizing an array for details.

Iterating over the elements of an array

Iterating over the elements of an array is the same as the following: `for each element v in the array x, do` *`something`* `with v`. There are two traditional *patterns* for iterating over the elements of an array: using a `while` or a `for` loop. Let's print the elements of the array `x` using a `while` loop (here the *`doing something`* is printing the element):

```
int i = 0;
while (i < x.length) {
    System.out.println(x[i]);
    i++;
}
```

And then using `for` loop:

```
for (int i = 0; i < x.length; i++)
    System.out.println(x[i]);
```

Java has another form of the `for` loop to implement this "foreach" pattern:

```
for (Foo v : x)
    System.out.println(v);
```

This version of the `for` has one advantage: it does exactly what it says - for each element v in the array x, it prints the element v! No indexing needed at all. It also has a disadvantage: there is no way to iterate over a part of the array, which is important when the *size* is not equal to the *capacity* of the array.

Copying an array

Copying the elements of a *source* array to *destination* array is simply a matter of copying the array element by element using an iterator.

```
public static Object[] copyArray(Object[] source) {
    Object[] copy = new Object[source.length];
    for (int i = 0; i < source.length; i++)
        copy[i] = source[i];

    return copy;
}
```

For you information, the `java.util.Arrays` class provides a set of methods to do just this for you in a very efficient way. Here's how:

```
public static Object[] copyArray(Object[] source) {
    Object[] copy = java.util.Arrays.copyOf(source, source.length);
    return copy;
}
```

Resizing an array

There is the classic problem of arrays - once created, it cannot change its capacity! The only way to "resize" an array is to first create a new and larger array, and copy the existing elements to the new array. The following static method resizes `oldArray` to have a capacity of `newCapacity`, and returns a

reference to the resized array.

```
static Object[] resize(Object[] oldArray, int newCapacity) {
    Object[] newArray = new Object[newCapacity];
    for (int i = 0; i < oldArray.length; i++)
        newArray[i] = oldArray[i];
    return newArray;
}
```

We can use this method in the following way:

```
Object[] data = new Object[10];
for (int i = 0; i < 10; i++)
    data[i] = new String(String.valueOf(i));

// The array "data" is now full, so need to resize before we can
// add more elements to it.
data = resize(data, 20);
for (int i = 10; i < 20; i++)
    data[i] = new String(String.valueOf(i));
```

Reversing an array

The simplest way of reversing an array is to first copy the elements to another array in the reverse order, and then copy the elements back to the original array. This *out-of-place* method is rather inefficient, but it's simple and it works.

```
public static void reverse(Object[] array) {
    Object[] tmpArray = new Object[array.length];
    int i = 0;                          // index into array
    int j = tmpArray.length - 1;        // index into the reverse copy
    while (i < array.length) {
        tmpArray[j] = array[i];
        i++;
        j--;
    }
    // Now copy the elements in tmpArray back into the original array.
    for (int i = 0; i < array.length; i++)
        array[i] = tmpArray[i];

    // NOTE: the following DOES NOT work! Why?
    // array = tmparray;
}
```

Fortunately, there is an *in-place* method that is far more efficient!

```
public static void reverse(Object[] array) {
    int i = 0;                          // forward index into left half
    int j = array.length - 1;           // backward index into right half
    while (i < j) {
        // Exchange array[i] with array[j]
        Object tmp = array[i];
```

```
            array[i] = array[j];
            array[j] = tmp;

            i++;
            j--;
        }
    }
```

Shifing an array left

Shifting an entire array left moves each element one (or more, depending how the shift amount) position to the left. Obviously, the first element in the array will *fall off* the beginning and be lost forever. The last slot of the array before the shift (ie., the slot where where the last element was until the shift) is now unused (we can put a `null` there to signify that). The size of the array remains the same however, because the assumption is that you would something in the now-unused slot. For example, shifing the array `[5, 3, 9, 13, 2]` left by one position will result in the array `[3, 9, 13, 2, -]`. Note how the `array[0]` element with value of `5` is now lost, and there is an empty slot at the end (shown as - above).

```
public static void shiftLeft(Object array[]) {
    for (int i = 1; i < array.length; i++)
        array[i - 1] = array[i];
    array[a.length - 1] = null;         // Now empty
}
```

What would happen if this were a *circular* or *cyclic* array? See show to rotate an array left.

Shifing an array right

Shifting an entire array right moves each element one (or more, depending how the shift amount) position to the right. Obviously, the last element in the array will *fall off* the end and be lost forever. The first slot of the array before the shift (ie., the slot where where the 1st element was until the shift) is now unused (we can put a `null` there to signify that). The size of the array remains the same however, because the assumption is that you would something in the now-unused slot. For example, shifing the array `[5, 3, 9, 13, 2]` right by one position will result in the array `[-, 5, 3, 9, 13]`. Note how the `array[4]` element with value of `2` is now lost, and there is an empty slot in the beginning (shown as - above).

```
public static void shiftRight(Object array[]) {
    for (int i = array.length - 1; i > 0; i--)
        array[i] = array[i - 1];
    array[0] = null;                    // Now empty.
}
```

What would happen if this were a *circular* or *cyclic* array? See show to rotate an array right.

Inserting an element into an array

Inserting an element into any slot in an array requires that we first make *room* for it by shifting some of the elements to the right, and then insert the new element in the newly formed *gap*. The only time we

don't have to shift is when we insert in the next available empty slot in the array (the one after the last element). The insertion also assumes that there is at least one empty slot in the array, or else it must be resized as we had done earlier (or, if it's *non-resizable* by policy, then we can throw an exception). For example, inserting the value `7` in the slot with index `2` in the array `[3, 9, 12, 5]` produces the array `[3, 9, 7, 12, 5]`.

```
// Insert the given element at the given index in the non-resizable array
// with size elements.
public static void insert(Object[] array, int size, Object elem, int index) {
    if (size == array.length)
        throw new RuntimeException("no space left");
    else {
        // make a hole by shifting elements to the right.
        for (int i = index; i < size; i++)
            array[i + 1] = array[i];

        // now fill the hole/gap with the new element.
        array[index] = elem;
    }
}
```

Removing an element from an array

After removing the element at a given index, we need to *plug the hole* by shifting all the elements to its right one position to the left.

```
// Removes the element at the given index from the array with size elements.
public static void remove(Object[] array, int size, int index) {
    // Shift all elements [index+1 ... size-1] one position to the
    // left.
    for (int i = index + 1; i < size; i++)
        array[i - 1] = array[i];

    // Now nullify the unused slot at the end.
    array[size - 1] = null;
}
```

Rotating an array left

Rotating an array left is equivalent to shifting a *circular* or *cyclic* array left — the 1st element will not be lost, but rather move to the last slot. Rotating the array `[5, 3, 9, 13, 2]` left by one position will result in the array `[3, 9, 13, 2, 5]`.

```
public static void rotateLeft(Object array[]) {
    Object firstElement = array[0];
    for (int i = 1; i < array.length; i++)
        array[i - 1] = array[i];
    array[a.length - 1] = firstElement;
}
```

There is a much more elegant solution that we'll see when we discuss *circular* or *cyclic* arrays (wait till we study the Queue ADT).

Rotating an array right

Rotating an array right is equivalent to shifting a *circular* or *cyclic* array right — the last element will not be lost, but rather move to the 1st slot. Rotating the array `[5, 3, 9, 13, 2]` right by one position will result in the array `[2, 5, 3, 9, 13]`.

```
public static void rotateRight(Object array[]) {
    Object lastElement = array[array.length - 1];
    for (int i = array.length - 1; i > 0; i--)
        array[i] = array[i - 1];
    array[0] = lastElement;
}
```

There is a much more elegant solution that we'll see when we discuss *circular* or *cyclic* arrays (wait till we study the Queue ADT).

You can look at the example ArrayExamples.java class to see how these can be implemented. Run the `ArrayExamples.main()` to see the output.

Circular arrays in Java

Table of contents

Introduction

In a ``normal'' *linear* array, the first available slot is at index 0, the second one at index 1, and so on until the last one at index `arr.length - 1` (where `arr` is a reference to the array in question). We cannot start before the first slot at index 0, and we must stop at the last slot at index `arr.length - 1` — any attempt to access a slot at `index < 0` or at `index >= arr.length` will cause an `ArrayIndexOutOfBoundsException` to be thrown.

We iterate over the elements the array in the usual way by advancing a cursor from the first slot to the last. We iterate backwards in a similar way.

```
for (int i = 0; i < size; i++)
    visit(arr[i]);

for (int i = size - 1; i >= 0; i--)
    visit(arr[i]);
```

where `size ≤ arr.length` is the number of elements in the linear array.

In a *circular* array (also called *circular* buffer), the end of the array **wraps around** to the start of the array, just like in musical chairs. Simply watch the second hand of a watch go from `0` to `59`, and then wrap around to `0` again, and you see a circular array at work. You should note that we create the **perception** of a circular array — the underlying data is still kept in a linear array with a start (at index `0`) and an end (at index `arr.length - 1`). We implement the circularity by manipulating the indices such that when we advance past the end of the underlying array, the cursor wraps around to the beginning again. Similarly, when we go past the beginning going backwards, the cursor wraps around to the end of the array. Basically, we ``fake'' the concept of a circular array on top of a ``normal'' linear array.

The following shows a linear array with a capacity of 8, and with 6 elements in it. The start index is assumed to be `0`. The next available slot is at index `size` or `6`, given by `start + size = 0 + 6 = 6`. The last element is in the slot at index `size - 1` or `5`, given by `start + size - 1 = 0 + 6 - 1 = 5`.

```
  0    1    2    3    4    5    6   7
-----------------------------------------   capacity = 8
| 5 |  7 |  9 |  15 | -4 |  17 |    |   |   size     = 6
-----------------------------------------
                                 ^-- nextAvailPos = size
```

In an equivalent circular array of the same capacity and with the same elements, the first element may start at any index, not necessarily at index `0`. For example, the following circular arrays are both equivalent to each other, and to the linear array shown above.

```
  0    1    2    3    4     5    6   7
-----------------------------------------   capacity = 8
| 5 |  7 |  9 |  15 | -4 |  17 |   |   |   size     = 6
-----------------------------------------
  ^-- start                      ^-- nextAvailPos

  0    1    2    3    4     5    6   7
-----------------------------------------   capacity = 8
|   | 5  |  7 |  9  | 15 | -4  | 17|   |   size     = 6
-----------------------------------------
      ^-- start                      ^-- nextAvailPos

  0    1    2    3    4     5    6   7
-----------------------------------------   capacity = 8
| 15| -4 | 17 |     |    | 5   | 7 | 9 |   size     = 6
-----------------------------------------
 nextAvailPos --^          ^-- start
```

Back to Table of contents

Moving a cursor forward and backward

Before going to iteration, let's take a look at how to advance an cursor (in both directions) in linear and circular arrays. In a linear array, moving an cursor `i` forward and backward by unit stride are simply:

Forward

```
i++, or
i = i + 1, or
i += 1
```

Backward

```
i--, or
i = i - 1, or
i -= 1
```

With the understanding that `i < 0` and `i ≥ arr.length` are invalid indices.

In a circular array, moving moving an cursor `i` forward (or advancing the cursor) is accomplished by first advancing it normally, and then wrapping it if necessary.

```
i++;
if (i == arr.length)
    i = 0;
```

It can also be done using modular arithmetic, by using the modulus operator (`%`) in Java.

```
i++;
i = i % arr.length;
```

Or, by combining the two statements in one.

```
i = (i + 1) % arr.length;
```

Moving backwards however cannot be done using the modulus operator, so we have to wrap it explicitly, if needed.

```
i--;
if (i == -1)
    i = arr.length - 1;
```

We can of course advance the cursor by an `offset > 1`. For a linear array, we can advance a cursor, forward and backward, by a given `offset` in the following way.

Forward

```
i = i + offset
(i >= arr.length will be invalid)
```

Backward

```
i = i - offset
(i < 0 will be invalid)
```

For a circular array, no cursor position is invalid, since it simply wraps around in either direction.

Forward

```
i = i + offset
i = i % arr.length;
```

Backward

```
i = i - offset
if (i < 0)
    i = i + arr.length;
```

(please convince yourself of this before moving on ... There is also a bug in the backward direction code given above when offset is larger than the array's length).

Back to Table of contents

Iteration over the elements in a circular array

Now that we know how to move a cursor through a circular array in both directions, we can now iterate over the elements knowing the start index and the number of elements in the circular array. When iterating over the elements in a circular array, it's easier to use two different variables — one as just a counter that goes `size` (where `size` ≤ `arr.length`) times, and other is the actual cursor into the underlying array. For a circular array with the starting position at index `start`, we can iterate as follows.

Forward

```
int k = start;
for (int i = 0; i < size; i++) {
    visit(arr[k]);
    // advance k, wrapping if necessary
    k = (k + 1) % arr.length;
}
```

Backward

```
// Find the index of the last
// element in the circular array
int k = (start + size - 1) % arr.length;
for (int i = 0; i < size; i++) {
    visit(arr[k]);
    // move k backwards,
    // wrapping if necessary
    k--;
    if (k == -1)
        k = arr.length - 1;
}
```

For the reverse iteration, we need to find the index of the last element in the circular array, which can be accomplished by the following.

```
lastPos = (start + size - 1) % arr.length;
```

Similarly, the index of the next available position is the following.

```
nextAvailPos = (start + size) % arr.length;
```

Back to Table of contents

Linearizing a circular array

Linearizing a circular produces a linear array where the first element is at index `0`, and the last element is at index `size - 1`. It's basically copying the circular array into a linear one, element by element, such that `circArr[start] → linearArr[0]`, `circArr[start + 1] → linearArr[1]`, `circArr[start + 2] → linearArr[2]`, and so on. The following shows the linearized version of a circular array (the input circular array is at the top, with its linearized version at the bottom).

```
  0    1    2    3    4    5    6   7
-----------------------------------------   capacity = 8
| 15| -4 | 17 |    |    | 5   | 7 | 9 |   size     = 6
-----------------------------------------
 nextAvailPos --^          ^-- start

  0    1    2    3    4    5    6   7
-----------------------------------------   capacity = 8
| 5 |  7 |  9 |  15 | -4 |  17 |   |   |   size     = 6
-----------------------------------------
  ^-- start                    ^-- nextAvailPos
```

The returned linearized array is a copy of the circular array, with its first element at index `0`. Since we already know how to iterate, it's actually quite simple.

```
/**
 * Returns a linearized copy of the specified circular array.
 * @param circArr the circular array
 * @param start   the index of the first element in the circular array
 * @param size    the number of elements (must be ≤ circArray.length)
 * @return a linear array with the elements in the circular array
 */
public static Object[] linearize(Object[] circArr, int start, int size) {
    Object[] linearArr = new Object[size];
    int k = start;
    for (int i = 0; i < size; i++) {
        linearArr[i] = circArr[k];
        k = (k + 1) % circArr.length;
    }
    return linearArr;
}
```

Back to Table of contents

Resizing a circular array

Resizing a circular array to have a larger capacity, and maintaining the existing elements, is almost exactly the same as linearizing it! The following shows the resized version of a circular array (the input circular array is at the top, with its resized version at the bottom).

```
  0    1    2    3    4    5    6   7
-----------------------------------------   capacity = 8
| 15| -4 | 17 |    |    | 5   | 7 | 9 |   size     = 6
-----------------------------------------
 nextAvailPos --^          ^-- start

  0    1    2    3    4    5    6   7   8   9
-------------------------------------------------   capacity = 10
| 5 |  7 |  9 |  15 | -4 |  17 |   |   |   |   |   size     = 6
-------------------------------------------------
```

```
       ^-- start                      ^-- nextAvailPos
```

The returned resized array is a copy of the circular array, with its first element at index `0`. The code is shown below.

```
  /**
   * Returns a resized copy of the specified circular array.
   * @param circArr the circular array
   * @param start   the index of the first element in the circular array
   * @param size    the number of elements (must be ≤ circArray.length)
   * @param newCap  the new capacity
   * @return a resized copy with the elements in the circular array
   */
  public static Object[] resize(Object[] circArr, int start, int size,
        int newCap) {
      Object[] resizedArr = new Object[newCap];
      int k = start;
      for (int i = 0; i < size; i++) {
          resizedArr[i] = circArr[k];
          k = (k + 1) % circArr.length;
      }
      return resizedArr;
  }
```

Back to Table of contents

Inserting an element in a circular array

Let's say you want to insert a new element at a paritcular position in a circular array. In a linear array, the position is equivalent to the index where the new element would be inserted. In a circular array, the position is **not the index**, but rather it is the **offset** from the **front** element. You can compute the index into the underlying array using the by-now-familiar relation `(front + offset) % circArr.length`.

Let's see the insertion in action using the following example — we insert a new element `13` at position `3`, which results in the circular array shown at the bottom.

```
Insert element 13 in position 3, or at index (5 + 3) % 8 = 0.

       0    1    2    3    4     5    6   7
     ----------------------------------------   capacity = 8
     | 15| -4 | 17 |     |    | 5   | 7 | 9 |   size     = 6
     ----------------------------------------
       ^                        ^-- start
       ^--- this is position 3 relative to front

And the resulting circular array after insertion is show below.

       0    1    2    3    4     5    6   7
     ----------------------------------------   capacity = 8
     |13 | 15 | -4 | 17  |    | 5   | 7 | 9 |   size     = 7
     ----------------------------------------
                              ^-- start
```

Inserting into a circular array is no different that inserting into a linear array — we need to **shift** elements one position to the **right** to create a "hole" for the new element. The only difference is how we shift the elements in linear vs circular array. In either case, to shift elements, we need to know the following:

1. the **index** where to start shifting, and
2. the **number of elements** to shift.

As we noted earlier, in the case of a circular array, we are often given the **offset** from the **front** element (which is exactly what the index is for a linear array!), so we have to first compute the index where to start the shift. Once we know that, the process is exactly the same! The **number of elements** to shift is given by `size - offset` (which is exactly the same for a linear array!).

```
/**
 * Inserts an element at the given position in a circular array.
 * @param circArr the circular array
 * @param start   the index of the first element in the circular array
 * @param size    the number of elements (must be ≤ circArray.length)
 * @param elem    the new element to insert
 * @param pos     the offset of the new element relative to start
 * @return index of the new element just inserted
 */
public static int insert(Object[] circArr, int start, int size,
      Object elem, int pos) {

    // Double the capacity if full
    if (size == circArr.length)
        resize(circArr, start, size, size * 2);

    // Find the number of elements to shift
    int nShifts = size - pos;

    // Since we're shifting elements to the right, we have to start
    // from the right end of the array, and move backwards until we
    // reach the position where the new element is going to be inserted.
    // Start by figuring out the index of the last element or the next
    // available position -- either one is ok.
    int from = (start + size - 1) % circArr.length;
    int to = (from + 1) % circArr.length;
    for (int i = 0; i < nShifts; i++) {
        circArr[to] = circArr[from];

        // move to and from backwards.
        to = from;
        from = from - 1;
        if (from == -1)
            from = circArr.length;
    }

    // Now there is a hole at the given offset, so find the index of
    // the hole in the underlying array, and put the new element in it.
    int index = (start + pos) % circArr.length;
    circArr[index] = elem;

    // Return the index of the new element
    return index;
}
```

Back to Table of contents

Removing an element in a circular array

Now's let's try the opposite — **remove** an element at a given position in a circular array. The problem is exactly the opposite of insertion — we have to **plug the hole** left by removed element by shifting elements to the **left** by one position.

Let's see the removal in action using the following example — we remove the element `15` from a circular array shown at the top, which results in the circular array shown at the bottom. Since we're given the element to remove, we first need to find the **offset** relative to the **front** element, and then the underlying index. In the example below, we can iterate through the circular array starting from **start**, and see that the element `15` is at an offset `4` from the start. We

show two different ways of plugging the "hole" left by the removed element.

```
Remove element 15 in position 4, or at index (5 + 4) % 8 = 1.

       0    1    2    3    4     5    6   7
     ---------------------------------------   capacity = 8
     |13 | 15 | -4 | 17  |    | 5   | 7 | 9 |   size     = 7
     ---------------------------------------
                               ^-- start
            ^--- this is position 4 relative to front

And the resulting circular array after removal is show below.

       0    1    2    3    4     5    6   7
     ---------------------------------------   capacity = 8
     | 13| -4 | 17 |     |    | 5   | 7 | 9 |   size     = 6
     ---------------------------------------
                               ^-- start

The following is also correct - note the difference in which
elements we're shifting to "plug the hole". And note "start"
is now changed.

       0    1    2    3    4     5    6   7
     ---------------------------------------   capacity = 8
     | 9 | 13 | -4 | 17  |    |     | 5 | 7 |   size     = 6
     ---------------------------------------
                                   ^-- start
```

Just like in the case of insertion, we need to shift; just that, in the case of removal, we shift in the opposite direction to plug a hole. And to shift elements, we need to know the following:

1. the **index** where to start shifting, and
2. the **number of elements** to shift.

As we noted earlier, in the case of a circular array, we are often given the **offset** from the **front** element (which is exactly what the index is for a linear array!), so we have to first compute the index where to start the shift. Once we know that, the process is exactly the same! The **number of elements** to shift is given by `size - offset - 1` (note the `-1` at the end — it's different than the case of insertion).

We can do it one of two ways:

1. Shift all subsequent elements to the **left** by one position, which is the way we remove an element from a linear array (so we call it the *usual* way). The code to do that is shown below. This is the first way of doing it in the illustration above.

```
/**
 * Removes an element at the given position in a circular array.
 * @param circArr the circular array
 * @param start   the index of the first element in the circular array
 * @param size    the number of elements (must be ≤ circArray.length)
 * @param elem    the new element to insert
 * @param pos     the offset of the new element relative to start
 * @return the object that was removed
 */
public static Object remove(Object[] circArr, int start, int size,
      Object elem, int pos) {

    // Save a reference to the element that is to be removed, so find
    // its index first.
    int index = (start + pos) % circArr.length;
    Object removed = circArr[index];
```

```
    // Find the number of elements to shift
    int nShifts = size - pos - 1;

    // Since we're shifting elements to the left, we have to start
    // from the left end of the array where the element to be removed
    // is currently at, and move forwards until we reach the last
    // element. Start by figuring out the index of the element to be
    // removed.
    int to = index;
    int from = (to + 1) % circArr.length;
    for (int i = 0; i < nShifts; i++) {
        circArr[to] = circArr[from];

        // advance to and from forwards.
        to = from;
        from = (from + 1) % circArr.length;
    }

    // The last slot is now unused, so help GC by null'ing it.
    circArr[from] = null;

    // Return the element removed.
    return removed;
}
```

2. Shift the elements from the **front** to the given element **right** by one position, and then advance **front**. This is only possible for a circular array, since we can treat any index as the starting position. This is the second way of doing it in the illustration above.

```
/**
 * Removes an element at the given position in a circular array.
 * @param circArr the circular array
 * @param start   the index of the first element in the circular array
 * @param size    the number of elements (must be ≤ circArray.length)
 * @param elem    the new element to insert
 * @param pos     the offset of the new element relative to start
 * @return the object that was removed
 */
public static Object remove(Object[] circArr, int start, int size,
      Object elem, int pos) {

    // Save a reference to the element that is to be removed, so find
    // its index first.
    int index = (start + pos) % circArr.length;
    Object removed = circArr[index];

    // We'll shift each element from start to this index one
    // position to the right.

    // Find the number of elements to shift. Note that it's not
    // the same number as in the first method.
    int nShifts = size - pos;

    // Since we're shifting elements to the right, we have to start
    // from the right end of the array where the element to be removed
    // is currently at, and move backwards until we reach the front
    // element. Start by figuring out the index of the element to be
    // removed.
    int to = index;
    int from = to - 1;
    if (from == -1)
        from = circArray.length - 1;
    for (int i = 0; i < nShifts; i++) {
        circArr[to] = circArr[from];
```

```
            // move to and from backwards.
            to = from;
            from = from - 1;
            if (from == -1)
                from = circArray.length - 1;
        }

        // The first slot is now unused, so help GC by null'ing it.
        circArr[start] = null;

        // And advance start to point to the shifted front element.
        start = (start + 1) % circArray.length;

        // Return the element removed.
        return removed;
    }
```

Back to Table of contents

Questions to ponder

Some questions for you to ponder:

- How do you shift he elements in a cicular array? For example, how would you shift the first j elements in a circular array one place to the right? Same, but to the left? See how it's done when inserting a new element in a circular array.
- Why do we care about circular arrays? Think of one application where it was nice to have (and why).

Back to Table of contents

Linked lists in Java

Introduction

When studying Java's built-in array type, we were quite annoyed by the limitations of these arrays:

- *fixed capacity*: once created, an array cannot be resized. The only way to "resize" is to create a larger new array, copy the elements from the original array into the new one, and then change the reference to the new one. Very inefficient, but of course we can use a bit of intelligence to make it a bit better (eg., double the space each time instead of adding just the required amount, etc).
- *shifting elements in insert/remove*: since we do not allow *gaps* in the array, inserting a new element requires that we shift all the subsequent elements right to create a *hole* where the new element is placed. In the worst case, we "disturb" every slot in the array when we insert in the beginning of the array!

  Removing may also require shifting to *plug* the hole left by the removed element. If the ordering of the elements does not not matter, we can avoid shifting by replacing the removed element with the element in the last slot (and then treating the last slot as empty).

The answer to these problems is the *linked list* data structure, which is a sequence of *nodes* connected by *links*. The links allow inserting new nodes anywhere in the list or to remove an existing node from the list without having to disturb ther rest of the list (the only nodes affected are the ones adjacent to the node being inserted or removed). Since we can extend the list one node at a time, we can also resize a list until we run out of resources. However, we'll find out soon enough that what we lose in the process — random access, and space! Linked list is a sequence container that supports sequential access only, and requires additional space for at least `n` references for the links.

We maintain a linked list by referring to the first node in the list, conventionally called the *head* reference. All the other nodes can then be reached by traversing the list, starting from the *head*. An empty list is represented by setting the head reference to the null. Given a head reference to a list, how do you count the number of nodes in the list? You have to iterate over the nodes, starting from head, and count until you reach the end.

As I mentioned earlier, a linked list is a sequence of nodes. To put anything (primitive type or object reference) in the list, we must first put this "thing" in a node, and then put the node in the list. Compare this with an array – we simply put the "thing" (our primitive type or object reference) directly into the array slot, without the need for any extra scaffolding. Our `Node` class is quite simple — it contains element (an object reference if we have a list of objects), and a reference to the *next* node in the list:

```
public class Node {
    // The element that we want to put in the list
    public Object element;
    // The reference to the next node in the list
    public Node next;

    /**
     * Creates a new node with given element and next node reference.
     *
     * @param e the element to put in the node
```

```
     * @param n the next node in the list, which may be null
     */
    public Node(Object e, Node n) {
        element = e;
        next = n;
    }
}
```

This allows you a create a *singly-linked* list, and as a result, we can only move *forward* in the list by following the *next* links. To be able to move *backward* as well, we'll have to add another reference to the *previous* node, increasing the space usage even more.

Linked list manipulation

Now we look at the typical list manipulations that we're discussing in class this week. See the link to a complete implementation for you to play with at the end of this document.

Topics, in no particular order:

Creating a list of elements

To create an empty list, referenced by *head*:

```
    Node head = null;                    // empty list
```

A list with just one String element ("hello"):

```
    head = new Node("hello", null);
```

Back to list of topics | top of document

Iterating over the elements

To iterate over the elements of a list, we start at the *head* node, and then advance one node at a time. For

example, the following prints out each element (stored within the nodes) in the list.

```
Node n = head;
while (n != null) {
    System.out.println(n.element);
    n = n.next;
}
```

You can use a for loop as well of course.

```
for (Node n = head; n != null; n = n.next)
    System.out.println(n.element);
```

Back to list of topics | top of document

Counting the number of elements in a list

reference to the null. Given a head reference to a list, how can you count the number of elements in the list? You have to iterate over the nodes, starting from head, and count until you reach the end.

```
public static int count(Node head) {
    int count = 0;
    for (Node n = head; n != null; n = n.next)
        count++;

    return count;
}
```

Back to list of topics | top of document

Getting an element given the index into a list

It's useful to be able to extract an element given the index into a linked list (the index works the same way as for arrays – it goes from `0` to `size-1`). Since a linked list does not support random access, we have to scan the list to find the node at the given index, and then get the element within that node. Since we may not know the length (same as what we call size) of the list, we have to check if the index is within bounds.

```
/**
 * Returns the element at the given index in the given list.
 *
 * @param head  the reference to the head node of the list
 * @param index the index of the element to get
 * @return the element at the given index, or null if the index is out
 *         of bounds.
 */
public static Object get(Node head, int index) {
    if (index < 0)
        return null;                    // invalid index.
```

```
    int i = 0;
    for (Node n = head; n != null; n = n.next) {
        if (i == index)
            return n.element;
        else
           i++;
    }
    return null;                        // index out of bounds.
}
```

If we know the number of elements in the list (ie., the *size*), then it's much easier.

```
/**
 * Returns the element at the given index in the given list.
 *
 * @param head  the reference to the head node of the list
 * @param size  the number of elements in the list
 * @param index the index of the element to get
 * @return the element at the given index, or null if the index is out
 *         of bounds.
 */
public static Object get(Node head, int size, int index) {
    if (index < 0 || index >= size)
        return null;                    // invalid index.

    Node n = head;
    for (int i = 0; i < index; i++, n = n.next)
        ;

    return n.element;
}
```

It's typical in most applications to write a method to get the node at a given index, so that we can get the element by first getting the node, and the return the element within. We define a `nodeAt` method to return the node at the given index.

```
/**
 * Returns the node at the given index in the given list.
 *
 * @param head  the reference to the head node of the list
 * @param size  the number of elements in the list
 * @param index the index of the node to get
 * @return the node at the given index, or null if the index is out
 *         of bounds.
 */
public static Node nodeAt(Node head, int size, int index) {
    if (index < 0 || index >= size)
        return null;                    // invalid index.

    Node n = head;
    for (int i = 0; i < index; i++, n = n.next)
        ;

    return n;
}
```

With this method, it's trivial to write the `get` method.

```
/**
 * Returns the element at the given index in the given list.
 *
 * @param head  the reference to the head node of the list
 * @param size  the number of elements in the list
 * @param index the index of the element to get
 * @return the element at the given index, or null if the index is out
 *         of bounds.
 */
public static Object get(Node head, int size, int index) {
    Node node = nodeAt(head, size, index);
    if (node == null)
        return null;                    // invalid index.
    else
        return node.element;
}
```

Back to list of topics | top of document

Setting an element given the index into a list

Setting the element at the given index (also called *updating* the element) is as simple as `get` using `nodeAt` method.

```
/**
 * Sets the element to the given one at the given index in the given list.
 *
 * @param head  the reference to the head node of the list
 * @param size  the number of elements in the list
 * @param index the index of the element to update
 * @param elem  the element to update the value with
 * @return the old element at index if valid, or null if the index is out
 *         of bounds.
 */
public static Object set(Node head, int size, int index, Object elem) {
    Node node = nodeAt(head, size, index);
    if (node == null)
        return null;                    // invalid index.
    else {
        Object oldElem = node.element;
        node.element = elem;
        return oldElem;
    }
}
```

Back to list of topics | top of document

Searching for an element in a list

Searching for an element in a list can be done by sequentially searching through the list. There are two typical variants: return the index of the given element (`indexOf`), or return true if the element exists

(`contains`). Both of these require that we walk through the list and check for the existence of the given element.

```
/**
 * Returns the index of the element in the list.
 *
 * @param head  the reference to the head node of the list
 * @param size  the number of elements in the list
 * @param elem  the element to find the index of
 * @return the index of the element is found, or -1 otherwise
 */
public static int indexOf(Node head, int size, Object elem) {
    int i = 0;
    for (Node n = head; n != null; n = n.next, i++)
        if (n.element.equals(elem))
            return i;

    return -1;                          // Not found
}
```

The other variant returns true if the element is found, or false otherwise.

```
/**
 * Returns true if the element exists in the list.
 *
 * @param head  the reference to the head node of the list
 * @param size  the number of elements in the list
 * @param elem  the element to find the index of
 * @return true if the element is found, or false otherwise
 */
public static boolean contains(Node head, int size, Object elem) {
    int i = 0;
    for (Node n = head; n != null; n = n.next, i++)
        if (n.element.equals(elem))
            return true;

    return false;                               // Not found
}
```

You should note that we can easily use one of the two methods to define the other instead of writing both! We can rewrite `contains` using `indexOf` or vice-versa quite trivally.

```
/**
 * Returns true if the element exists in the list.
 *
 * @param head  the reference to the head node of the list
 * @param size  the number of elements in the list
 * @param elem  the element to find the index of
 * @return true if the element is found, or false otherwise
 */
public static boolean contains(Node head, int size, Object elem) {
    int index = indexOf(head, size, elem);
    if (index >= 0)
        return true;
    else
```

```
        return false;

    // The "if ... else" block can also be written as:
    // return (index >= 0) ? true : false;
    // or even simpler as
    // return index >= 0;
}
```

Back to list of topics | top of document

Inserting an element into a list

There are three places in a list where we can insert a new element: in the beginning ("head" changes), in the middle, and at the end. To insert a new node in the list, you need the reference to the *predecessor* to link in the new node. There is one "special" case however — inserting a new node in the beginning of the list, because the *head* node does not have a predecessor. Inserting in the beginning has an important *side-effect* — it changes the head reference! Inserting in the middle or at the end are the same — we first find the predecessor node, and link in the new node with the given element.

```
/**
 * Inserts the new element at the given index into the list.
 *
 * @param head  the reference to the head node of the list
 * @param size  the number of elements in the list
 * @param index the index of the newly inserted element
 * @param elem  the new element to insert at the given index
 * @return the reference to the head node, which will change when
 *         inserting in the beginning.
 * @exception IndexOutOfBoundsException if the index is out-of-bounds.
 */
public static Node insert(Node head, int size, Object elem, int index) {
    // Check for invalid index first. Should be between 0 and size (note:
    // note size-1, since we can insert it *after* the tail node (tail
    // node has an index of size-1).
    if (index < 0 || index > size)
        throw new IndexOutOfBoundsExceptionException();

    // Create the new node to hold the element in the list
    Node newNode = new Node(elem, null);

    // Remember the special case -- in the beginning of the list, when
    // the head reference changes.
    if (index == 0) {
        newNode.next = head;
        head = newNode;
    } else {
        // get the predecessor node first
        Node pred = nodeAt(index - 1);
        newNode.next = pred.next;
        pred.next = newNode;
    }
    return head;
}
```

If there was also a *tail* reference, in addition to *head*, *appending* to the list is very fast — it avoids having

to scan the entire list until the *tail* node (the predecessor in this case) is found.

Back to list of topics | top of document

Removing an element from a list

Removing an element from the list is done by removing the node that contains the element. If we only know the element that we want to remove, then we have to sequentially search the list to find the node which contains the element (if it exists in the list that is); or else, we may already have a reference to the node which contains the element to remove; or, we have an index of the element in the list. Just like inserting a new node in a list, removing requires that you have the reference to the predecessor node. And just like in insertion, removing the 1st node in the list is a "special" case — it does not have a predecessor, and removing it has the side-effect of changing head!

```
/**
 * Removes the element at the given index in the list.
 *
 * @param head  the reference to the head node of the list
 * @param size  the number of elements in the list
 * @param index the index of the element to remove
 * @return the reference to the head node, which will change when
 *         removing the element at the beginning.
 * @exception IndexOutOfBoundsException if the index is out-of-bounds.
 */
public static Node remove(Node head, int size, int index) {
    // Check for invalid index first. Should be between 0 and size-1.
    if (index < 0 || index >= size)
        throw new IndexOutOfBoundsException();

    // Reference to the removed node.
    Node removedNode = null;

    // Remember the special case -- from the beginning of the list, when
    // the head reference changes.
    if (index == 0) {
        removedNode = head;
        head = head.next;
    } else {
        // get the predecessor node first
        Node pred = nodeAt(index - 1);
        removedNode = pred.next;
        pred.next = removedNode.next;
    }

    // Help the GC
    removedNode.element = null;
    removedNode.next = null;

    return head;
}
```

Back to list of topics | top of document

Copying a list

Copying the elements of a *source* list to *destination* list is simply a matter of iterating over the elements of the source list, and inserting these elements at the end of the destination list.

```
/**
 * Copy the source list and return the reference to the copy.
 *
 * @param source the head reference of the source list
 * @return reference to the head of the copy
 */
public static Node copyList(Node source) {
    Node copyHead = null;
    Node copyTail = null;
    for (Node n = source; n != null; n = n.next) {
        Node newNode = new Node(n.element, null);
        if (copyHead == null) {
            // First node is special case - head and tail are the same
            copyHead = newNode;
            copyTail = copyHead;
        } else {
            copyTail.next = newNode;
            copyTail = copyTail.next;   // same as: copyTail = newNode;
        }
    }
    return copyHead;
}
```

Back to list of topics | top of document

Reversing a list

Since a linked list does not support random access, it is difficult to reverse a list *in-place* without changing the head reference. Instead we'll create a new list with its own head reference, and copy the elements in the reverse order. This method does not modify the original list, so we can call it an *out-of-place* method.

```
/**
 * Reverses the list and returns the reference to the head node.
 *
 * @param list the list to reverse
 * @param size the number of elements in the list
 * @return reference to the head of the reversed list
 */
public static Node reverse(Node head, int size) {
    Node newHead = null;

    // We iterate over the nodes in the original list, and add a copy
    // of each node to the *beginning* of the new list, which reverses
    // the order.
    for (Node n = head; n != null; n = n.next) {
        Node newNode = new Node(n.element, null);

        // Add the node's copy to the beginning of the reversed list.
        newNode.next = newHead;
        newHead = newNode;
    }
```

```
    return newHead;
}
```

The problem with this approach is that it creates a copy of the whole list, just in the reverse order. We would like an *in-place* approach instead — re-order the links instead of copying the nodes! That would of course change the original list, and would have a new head reference (which is the reference to the tail node in the original list).

```
/**
 * Reverses the list and returns the reference to the head node.
 *
 * @param list the list to reverse
 * @param size the number of elements in the list
 * @return reference to the head of the reversed list
 */
public static Node reverse(Node head, int size) {
    Node newHead = null;

    // We iterate over the nodes in the original list, and add each
    // node to the *beginning* of the new list, which reverses the
    // order. Have to be careful when we iterate -- have to save the
    // reference to the *next* node before changing the next link.
    Node n = head;
    while (n != null) {
        Node nextNode = n.next;          // need to do this!

        // now change the link so that node 'n' is placed in the
        // beginning of the list
        n.next = newHead;
        newHead = n;

        // Can't do n = n.next because the link is changed. That's why
        // we saved the reference to the next node earlier, so we can
        // use that now.
        n = nextNode;
    }

    return newHead;
}
```

Back to list of topics | top of document

Rotating a list left

Rotating a list left is much simpler than rotating an array — the 2nd node becomes the new *head*, and the 1st becomes the new *tail* node. We don't have to actually *move* the elements, it's just a matter of re-arranging a few links!

```
/**
 * Rotates the given list left by one element
 *
 * @param head the list to rotate left
 * @param size the number of elements in the list
```

```
 * @return reference to the head of the rotated list
 */
public static Node rotateLeft(Node head, int size) {
    // Bug alert - does it work for empty lists?
    Node oldHead = head;
    head = head.next;

    // Now append the old head to the end of this list. Find the tail,
    // and add the old head after the tail.
    Node tail = head;
    while (tail.next != null)
        tail = tail.next;

    // Now add after tail
    tail.next = oldHead;
    oldHead.next = null;

    return head;
}
```

We can also use *circular* or *cyclic* lists.

Back to list of topics | top of document

Rotating a list right

Rotating a list right is almost the same as rotating left — the current *tail* becomes the new *head*.

```
/**
 * Rotates the given list right by one element
 *
 * @param head the list to rotate right
 * @param size the number of elements in the list
 * @return reference to the head of the rotated list
 */
public static Node rotateRight(Node head, int size) {
    // Bug alert - does it work for empty lists?

    // Need to find tail and it's predecessor (do you see why?)
    Node p = null;
    Node q = head;
    while (q.next != null) {
        p = tail;
        q = q.next;
    }

    // Now q points to the tail node, and p points to it's predecessor
    // First make tail the new head
    q.next = head;
    head = q;

    // make p the tail
    p.next = null;

    return head;
}
```

We can also use *circular* or *cyclic* lists.

Back to list of topics | top of document

You can look at the example LinkedList.java class to see how these can be implemented. Run the `LinkedList.main()` to see the output.

Dummy Headed Doubly Linked Circular lists (DHDLC) in Java

Table of contents

Introduction

We have so far focused on the simply *head-referenced singly-linked linear* lists, and now we study a much more versatile variation that is often used in practice — *dummy head-referenced doubly linked circular* lists — which is in fact the current implementation of the `java.util.LinkedList` class in JDK. It uses a *dummy* to avoid special cases of inserting into an empty list, or removing the the last node from a list of unit size, and it uses double links to allow for iterating in both directions. The cost of course is the extra space needed to hold the dummy node (minimal cost), and the extra *previous* link in addition the usual *next* link for each node.

We augment the `Node` class that we have used so far to add the extra `prev` link.

```
public class Node {
    // The element that we want to put in the list
    public Object element;
    // The reference to the next node in the list
    public Node next;
    // The reference to the previous node in the list
    public Node prev;

    /**
     * Creates a new node with given element and next/prev node references.
     *
     * @param e the element to put in the node
     * @param n the next node in the list, which may be null
     * @param p the previous node in the list, which may be null
     */
    public Node(Object e, Node n, Node p) {
        element = e;
        next = n;
        prev = p;
    }
}
```

This allows you a create a *doubly-linked* list, and as a result, we can move *forward* and *backward* in the list by following the *next* and *prev* links.

Back to Table of contents

Creating a list of elements

To create an empty list, referenced by a dummy *head*:

```
// Create the dummy node with "null" element, and null links
Node head = new Node(null, null, null);
// And then make it circular
head.next = head.prev = head;
```

Now, let's add a single String element ("hello") after the dummy head:

```
  Node n = new Node("hello", null, null);
  n.next = head.next;
  n.prev = head;
  head.next = n;
  n.next.prev = n;
```

You should note that the reference to the *tail* node is simply `head.prev`, which gives us ability to *append* to the list in constant time (without having to iterate to find the tail reference, or having to maintain a separate tail reference).

Back to Table of contents

Iterating over the elements

To iterate over the elements of a list, we start *after* the *dummy head* node, and then advance one node at a time until we reach the *dummy head* node again. For example, the following prints out each element (stored within the nodes) in the list.

```
Node n = head.next;
while (n != head) {
    System.out.println(n.element);
    n = n.next;
}
```

You can use a for loop as well of course.

```
for (Node n = head.next; n != head; n = n.next)
    System.out.println(n.element);
```

Thanks to the *previous* links, we can iterate backwards as well. We start from the tail node, which is `head.prev`, and move backwards.

```
Node n = head.prev;             // n now refers to the tail node
while (n != head) {
    System.out.println(n.element);
```

```
    n = n.prev;
}
```

You can use a for loop as well of course.

```
for (Node n = head.prev; n != head; n = n.prev)
    System.out.println(n.element);
```

Back to Table of contents

Inserting an element into a list

There are three places in a list where we can insert a new element: in the beginning ("head" does not change since we're using a dummy), in the middle, and at the end. To insert a new node in the list, you need the reference to the *predecessor* to link in the new node. Unlike for a singly-linked linear list, there is no "special" case here, since there is always a valid *predecessor* node available, thanks to the dummy head. Let's write a static method that inserts a new element *after* the specified node in a list.

```
/**
 * Inserts the new element after the specified node in the list.
 *
 * @param p    the node after which the new element is to be added
 * @param elem the new element to insert after the specified node
 * @return the reference to the newly added node
 */
public static Node insertAfter(Node p, Object elem) {
    // Create the new node to hold the element in the list
    Node n = new Node(elem, null, null);

    // Now add it after the predecessor `p'
    Node q = p.next;            // q will refer to the next node
    n.next = q;
    n.prev = p;
    p.next = n;
    q.prev = n;

    // Done -- no special cases whatsoever!
    return n;
}
```

We can now use it to insert a new element after any existing node in the list. For example, the following adds the String element "Foo" in the beginning of the list:

```
// Insert the String element "Foo" after the dummy head.
Node n = insertAfter(head, "Foo");
```

And the following appends the String element "Foo" to the end of the list:

```
// Appends the String element "Foo" after the tail node.
Node n = insertAfter(head.prev, "Foo"); // head.prev is the tail
```

Back to Table of contents

Removing an element from a list

Removing an element from the list is done by removing the node that contains the element. If we only know the element that we want to remove, then we have to sequentially search the list to find the node which contains the element (if it exists in the list that is); or else, we may already have a reference to the node which contains the element to remove; or, we have an index of the element in the list. Just like inserting a new node in a list, removing requires that you have the reference to the predecessor node. Since we're using a doubly-linked list, finding a predecessor of a node `n`is trivial — it's `n.prev`. And thanks to the dummy head, there is no "special" case here as well. Let's write a static method that takes a reference to a node to remove from the specified list.

```
/**
 * Removes the specified node from the list.
 *
 * @param n the specified node to remove
 */
public static void removeNode(Node n) {
    // References to the predecessor and successor
    Node p = n.prev;
    Node q = n.next;

    // Remove node n.
    p.next = q;
    q.prev = p;

    // Unlike 'n' from the list.
    n.next = n.prev = null;

    // Help GC
    n.element = null;
}
```

That's it! We can remove the first node in the list as follows:

```
removeNode(head.next);
```

And the last/tail node in the list as follows:

```
removeNode(head.prev);
```

If we have only a reference to the element to remove, we have to first find the node, and then remove it using `removeNode`. Let's write that method as well.

```
/**
 * Removes the specified element from the list.
 *
 * @param head the reference to the dummy head of the list
```

```
     * @param elem the specified element to remove from the list
     * @return true if the element was removed, or false otherwise
     */
    public static void remove(Node head, Object elem) {
        // Sequentially scan to find the node. You should probably write a
        // a findNode method to return the Node that contains a given element.
        Node n = head.next;
        while (n != head) {
            if (n.element.equals(elem))
                break;
            n = n.next;
        }
        if (n == head)
            return false;           // did not find it!
        else {
            removeNode(n);
            return true;
        }
    }
```

Back to Table of contents

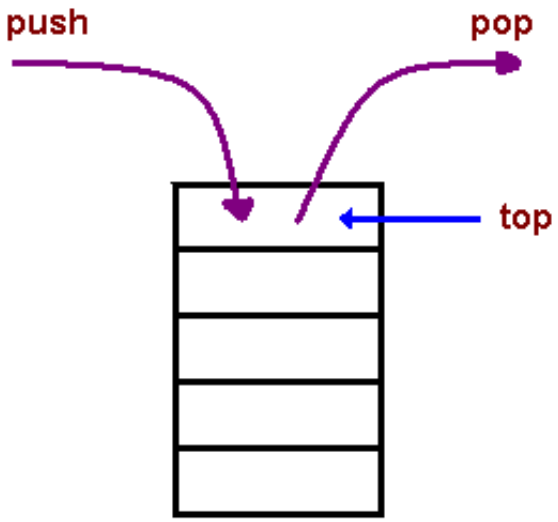
Stacks

Table of contents

Introduction

A Stack is a restricted ordered sequence in which we can only add to and remove from one end — the **top** of the stack. Imagine stacking a set of books on top of each other — you can **push** a new book on **top** of the stack, and you can **pop** the book that is currently on the top of the stack. You are not, strictly speaking, allowed to add to the middle of the stack, nor are you allowed to remove a book from the middle of the stack. The only book that can be taken out of the stack is the **most recently added** one; a stack is thus a "last in, first out" (LIFO) data structure.

We use stacks everyday — from finding our way out of a maze, to evaluating postfix expressions, "undoing" the edits in a word-processor, and to implementing recursion in programming language runtime environments.



Three basic stack operations are:

- **push(obj)**: adds `obj` to the top of the stack ("overflow" error if the stack has fixed capacity, and is full)
- **pop**: removes and returns the item from the top of the stack ("underflow" error if the stack is empty)
- **peek**: returns the item that is on the top of the stack, but does not remove it ("underflow" error if the stack is empty)

The following shows the various operations on a stack.

```
Java statement                    resulting stack
-----------------------------------------------
Stack s = new Stack();
                                   ----- (empty stack)



S.push(7);
                                  | 7 |  <-- top
                                  -----

S.push(2);
                                  | 2 |  <-- top
                                  | 7 |
                                  -----

S.push(73);
                                  |73 |  <-- top
                                  | 2 |
                                  | 7 |
                                  -----

S.pop();
                                  | 2 |  <-- top
                                  | 7 |
                                  -----
S.pop();
                                  | 7 |  <-- top
                                  -----
S.pop();
                                  -----  (empty stack)

S.pop();
                                  ERROR "stack underflow"
```

Back to Table of contents

Stack applications

- **Reverse**: The simplest application of a stack is to reverse a word. You push a given word to stack – letter by letter – and then pop letters from the stack. Here's the trivial algorithm to print a word in reverse:

```
begin with an empty stack and an input stream.
while there is more characters to read, do:
   read the next input character;
   push it onto the stack;
end while;
while the stack is not empty, do:
   c = pop the stack;
   print c;
end while;
```

- **Undo**: Another application is an "undo" mechanism in text editors; this operation is accomplished by keeping all text changes in a stack. Popping the stack is equivalent to "undoing" the last action. A very similar one is going back pages in a browser using the *back* button; this is accomplished by keeping the pages visited in a stack. Clicking on the *back* button is equivalent to going back to the most-recently visited page prior to the current one.
- **Expression evaluation**: When an arithmetic expression is presented in the *postfix* form, you can use a stack to evaluate it to get the final value. For example: the expression `3 + 5 * 9` (which is in the usual *infix* form) can be written as `3 5 9 * +` in the *postfix*. More interestingly, postfix form removes all parentheses and thus all implicit precedence rules; for example, the infix expression `((3 + 2) * 4) / (5 - 1)` is written as the postfix `3 2 + 4 * 5 1 - /`. You can now design a calculator for expressions in postfix form using a stack.

  The algorithm may be written as the following:

```
begin with an empty stack and an input stream (for the expression).
while there is more input to read, do:
   read the next input symbol;
   if it's an operand,
      then push it onto the stack;
   if it's an operator
      then pop two operands from the stack;
           perform the operation on the operands;
           push the result;
end while;
// the answer of the expression is waiting for you in the stack:
pop the answer;
```

  Let's apply this algorithm to to evaluate the postfix expression `3 2 + 4 * 5 1 - /` using a stack.

```
 Stack    Expression

|   |
 ---        3 2 + 4 * 5 1 - /
(empty)

| 3 |       2 + 4 * 5 1 - /
 ---

| 2 |
| 3 |       + 4 * 5 1 - /
 ---

| 5 |       4 * 5 1 - /
 ---

| 4 |
| 5 |       * 5 1 - /
 ---

| 20|       5 1 - /
 ---

| 5 |
| 20|       1 - /
 ---
```

```
| 1 |
| 5 |
| 20|       - /
 ---

| 4 |
| 20|       /
 ---

| 5 |      (finished. the result is on top of the stack)
 ---
```

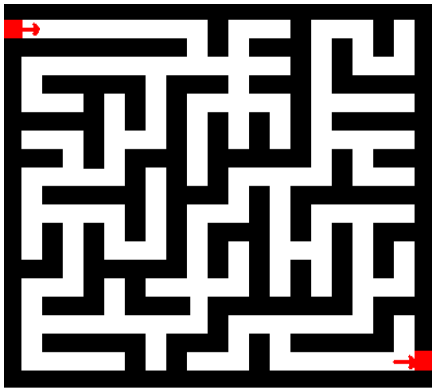
- **Parentheses matching**: We often have a expressions involving "()[]{}" that requires that the different types parentheses are *balanced*. For example, the following are properly balanced:
  - `(a (b + c) + d)`
  - `[ (a b) (c d) ]`
  - `( [a {x y} b] )`

  But the following are not:
  - `(a (b + c) + d`
  - `[ (a b] (c d) )`
  - `( [a {x y) b] )`

  The algorithm may be written as the following:

```
begin with an empty stack and an input stream (for the expression).
while there is more input to read, do:
   read the next input character;
   if it's an opening parenthesis/brace/bracket ("(" or "{" or "[")
      then push it onto the stack;
   if it's a closing parenthesis/brace/bracket (")" or "}" or "]")
      then pop the opening symbol from stack;
           compare the closing with opening symbol;
           if it matches
              then continue with next input character;
           if it does not match
              then return false;
end while;
// all matched, so return true
return true;
```

- **Backtracking**: This is a process when you need to access the most recent data element in a series of elements. Think of a labyrinth or maze - how do you find a way from an entrance to an exit?



Once you reach a dead end, you must backtrack. But backtrack to where? to the previous choice point. Therefore, at each choice point you store on a stack all possible choices. Then backtracking simply means popping a next choice from the stack.

- **Language processing**:
  - space for parameters and local variables is created internally using a stack (*activation records*).
  - compiler's syntax check for matching braces is implemented by using stack.
  - support for recursion

Back to Table of contents

Stack implementation

In the standard library of classes, the data type stack is an *adapter* class, meaning that a stack is built on top of other data structures. The underlying structure for a stack could be an array, a vector, an ArrayList, a linked list, or any other sequence (`Collection` class in JDK). Regardless of the type of the underlying data structure, a `Stack` must implement the same functionality. This is achieved by providing an interface:

```
public interface Stack {
    // The number of items on the stack
    int size();
    // Returns true if the stack is empty
    boolean isEmpty();
    // Pushes the new item on the stack, throwing the
    // StackOverflowException if the stack is at maximum capacity. It
    // does not throw an exception for an "unbounded" stack, which
    // dynamically adjusts capacity as needed.
    void push(Object o) throws StackOverflowException;
    // Pops the item on the top of the stack, throwing the
    // StackUnderflowException if the stack is empty.
    Object pop() throws StackUnderflowException;
    // Peeks at the item on the top of the stack, throwing
    // StackUnderflowException if the stack is empty.
    Object peek() throws StackUnderflowException;
    // Returns a textual representation of items on the stack, in the
    // format "[ x y z ]", where x and z are items on top and bottom
    // of the stack respectively.
    String toString();
    // Returns an array with items on the stack, with the item on top
    // of the stack in the first slot, and bottom in the last slot.
```

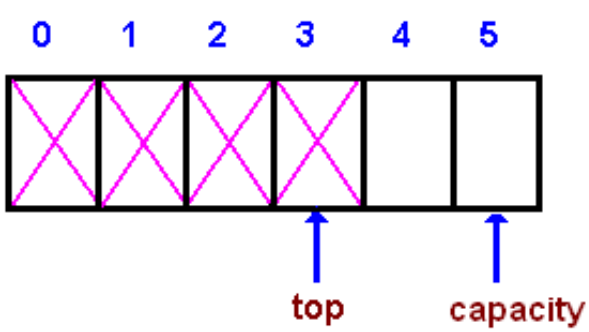
```
    Object[] toArray();
    // Searches for the given item on the stack, returning the
    // offset from top of the stack if item is found, or -1 otherwise.
    int search(Object o);
}
```

One requirement of a `Stack` implementation is that the `push` and `pop` operations run in *constant time*, that is, the time taken for stack operation is independent of how big or small the stack is.

Back to Table of contents

Array based implementation

In an array-based implementation we add the new item being pushed at the end of the array, and consequently pop the item from the end of the array as well. This way, there is no need to shift the array elements. The top of the stack is always the last used slot in the array, so we can use `size-1` to refer to the top of the stack element instead of having to keep a separate field. The *top* of the stack is not defined for an empty stack.



In a *bounded* or fixed-size stack abstraction, the capacity stays unchanged, therefore when *top* reaches *capacity*, the stack object throws an exception.

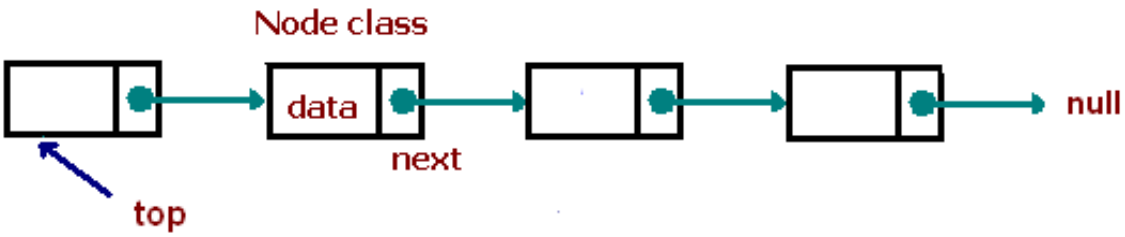
In an *unbounded* or dynamic stack abstraction when *top* reaches *capacity*, we double up the stack size. The following shows a partial array-based implementation of an *unbounded* stack.

```
public class ArrayStack implements Stack {
    private Object[] data;      // The array container
    private int      size;      // The number of items in the stack

    // Default initial capacity, which may then dynamically change
    private static final int DEF_INIT_CAPACITY = 100;

    public ArrayStack() {
        data = new Object[DEF_INIT_CAPACITY];
        size = 0;
    }
}
```

Back to Table of contents

Linked list based implementation

Linked List-based implementation provides the best (from the efficiency point of view) dynamic stack implementation.



The following shows a partial head-referenced singly-linked based implementation of an *unbounded* stack. In an singly-linked list-based implementation we add the new item being pushed at the beginning of the array (why?), and consequently pop the item from the beginning of the list as well.

```
public class ListStack implements Stack {
    private Node head;          // Reference to the top of the stack
    private int  size;          // The number of items in the stack

    public ListStack() {
        head = null;
        size = 0;
    }
}
```

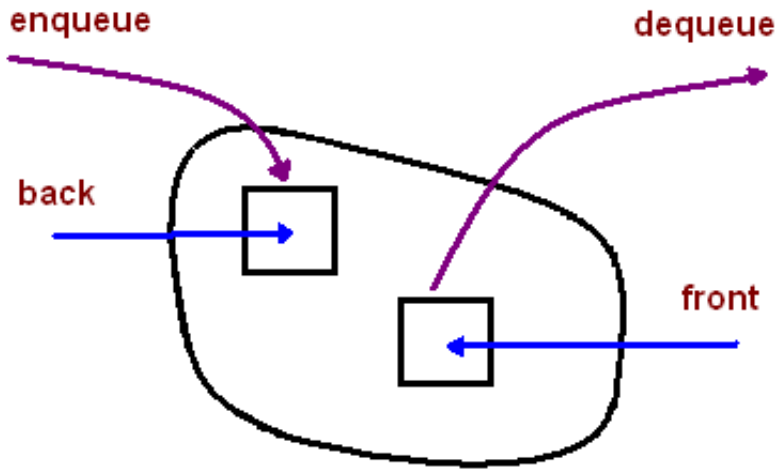
Back to Table of contents

Queues

Table of contents

Introduction

After the stack, the next interesting ADT is the queue. Just like a stack, a queue is also a restricted ordered sequence, with the difference that we can only add to one end — the **back** or **rear**, and remove from the other end — the **front** of the queue. You stand in a queue to get on a bus, where the earlier you arrive, the earlier you get on the bus; you stand in a queue to get service at a bank, where the teller services the front of the queue next. You are not, strictly speaking, allowed to add to the middle of the queue, nor are you allowed to remove an item from the middle of the queue. The only item that can be taken out of the queue is the one that was added the **earliest**; a queue is thus a "first in, first out" (FIFO) data structure. The following figure shows the FIFO nature of a queue.



Three basic queue operations are:

- **`enqueue(obj)`**: adds **`obj`** to the back of the queue ("overflow" error if the queue has fixed capacity, and is full)
- **`dequeue`**: removes and returns the item at the front of the queue ("underflow" error if the queue is empty)
- **`peek`**: returns the item that is at the front of the queue, but does not remove it ("underflow" error if the queue is empty)

Back to Table of contents

Queue applications

- **Palindrome**: A simple application is to check whether a string (or any sequence) is a palindrome or not. A *palindrome* is a sequence of symbols that reads the same backward or forward. Examples include:
  - radar
  - 31413
  - Able was I ere I saw Elba
  - GCATTACG
  - A man, a plan, a canal - Panama

  (Sometimes we ignore case. Sometimes we ignore spaces. Sometimes we ignore non-alphabetic characters.)

  How can we tell if a sequence of characters is a palindrome? We can use a stack and a queue. For each character in the sequence, we push it on the stack and place it on the back of the queue. When we have finished reading the sequence, we pop the stack and remove the front item from the queue, checking to see if the characters match. If so, we repeat until the stack and queue are empty (in which case the sequence is a palindrome) or until the popped and dequeued characters differ (in which case the sequence is not a palindrome).

  As an example, consider the word "radar":

```
|r|<-- top of stack
|a|
|d|                -----
|a|                radar
|r|                -----
---                ^   ^
                back   front
```

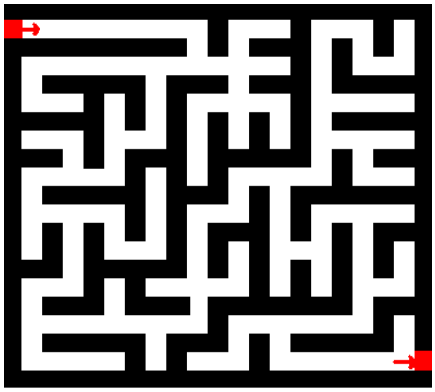
  The algorithm may be written as the following:

```
begin with an empty queue, an empty stack, and an input string;
for each character in the input string, do:
   if the character is a letter
      then enqueue it into the queue;
           push it onto the stack;
end for;
while the queue is not empty, do:
   c1 = dequeue the queue;
   c2 = pop the queue;
   if c1 <> c2
      then return false;
end while;
// All the characters must have matched, so must be a palindrome
return true;
```

  Note that you need to skip over the *non-letters*, and ignore the case as well. You can use the static methods `Character.isLetter` and `Character.toLowerCase` do accomplish that quite trivially.

- **Finding your way out of a maze**: Remember how we used a stack to find our way out of a stack

(using a technique known as **backtracking**)? We can also use a queue, which changes the sequence of steps we take, but the end result is the same.



Here is the algorithm to do just that.

```
Create an empty queue that holds the places you need to visit;
Put the starting point in this queue;
While the queue is not empty, do:
   Remove a place from the queue;
   If it is the finish point
       then we are done;
   ElseIf you have already visited this square
       then ignore it;
   Else
       mark the place as "visited";
       add all the neighbors of this place to your queue:
end for;
// If you eventually reach an empty queue and have not found the exit,
// there is no solution.
```

Back to Table of contents

Queue implementation

In the standard library of classes, the data type queue is an *adapter* class, meaning that a queue is built on top of other data structures. The underlying structure for a queue could be an array, a vector, an ArrayList, a linked list, or any other sequence (`Collection` class in JDK). Regardless of the type of the underlying data structure, a `Queue` must implement the same functionality. This is achieved by providing an interface:

```
public interface Queue {
    // The number of items in the queue
    int size();
    // Returns true if the queue is empty
    boolean isEmpty();
    // Adds the new item on the back of the queue, throwing the
    // QueueOverflowException if the queue is at maximum capacity. It
    // does not throw an exception for an "unbounded" queue, which
    // dynamically adjusts capacity as needed.
    void enqueue(Object o) throws QueueOverflowException;
    // Removes the item in the front of the queue, throwing the
    // QueueUnderflowException if the queue is empty.
```

```
    Object pop() throws QueueUnderflowException;
    // Peeks at the item in the front of the queue, throwing
    // QueueUnderflowException if the queue is empty.
    Object peek() throws QueueUnderflowException;
    // Returns a textual representation of items in the queue, in the
    // format "[ x y z ]", where x and z are items in the front and
    // back of the queue respectively.
    String toString();
    // Returns an array with items in the queue, with the item in the
    // front of the queue in the first slot, and back in the last slot.
    Object[] toArray();
    // Searches for the given item in the queue, returning the
    // offset from the front of the queue if item is found, or -1
    // otherwise.
    int search(Object o);
}
```

One requirement of a `Queue` implementation is that the `enqueue` and `dequeue` operations run in *constant time*, that is, the time taken for queue operation is independent of how big or small the queue is.

Back to Table of contents

Array based implementation

In an array-based implementation we add the new item being enqueued at the end of the array (constant cost), and consequently dequeue the item in the front the queue from the beginning of the array (linear cost, since we have to shift all the item to fill the hole). Or, we add the new item being enqueued in the beginning of the array (linear cost for shifting), and consequently dequeue the item in the front the queue from the end of the array (constant cost). Neither is a *win-win* situation, so we have to accept the loss in either enqueue or dequeue.

Let's choose the 1st option, in which the front item is always at index `0`, and where we shift after each `dequeue` operation. In this scheme, the front of the queue is always at index `0`, and rear one is at `size - 1`, where `size` denotes the number of items in the queue. A partial implementation is shown below.

```
public class LinearArrayQueue implements Queue {
    private Object[] data;      // The array container
    private int      size;      // The number of items in the queue

    // Default initial capacity, which may then dynamically change
    private static final int DEF_INIT_CAPACITY = 100;

    public LinearArrayQueue() {
        data = new Object[DEF_INIT_CAPACITY];
        size = 0;
    }
}
```

The following show a queue through a sequence of queue operations using a circular array. Note that we don't need to keep a separate `rear` variable since it's equal to `size - 1`.

```
        0    1    2    3    4     5
```

```
     ----------------------------------------
     |   |    |    |    |   |    |   |   |             size = 0
     ----------------------------------------
     (front, back undefined for an empty queue)

Let's enqueue the values 5 7 9 15 -4 and 17. The result is:

       0    1    2    3    4     5
     ----------------------------------------
     | 5 |  7 |  9 |  15 | -4 |  17 |   |   |          size = 6
     ----------------------------------------
       ^-- front                  ^-- rear

Dequeuing an item results in the following:

       0    1    2    3    4     5
     ----------------------------------------
     |  7 |  9 |  15 | -4 |  17 |   |   |   |          size = 5
     ----------------------------------------
       ^-- front             ^-- rear

Dequeuing again:

       0    1    2    3    4     5
     ----------------------------------------
     | 9 | 15 | -4 |  17 |    |     |   |   |          size = 4
     ----------------------------------------
       ^-- front      ^-- rear

And so on.
```

If we choose not to shift, but we may seem to have run out of space at the end, even if multiple dequeue operations have created space in the beginning of the array. Again, we have to shift the items to create space at the end, where we lose again.

The solution is to use a **circular** array instead of a **linear** array. See the notes on circular array for details on how it works, and why it provides a *win-win* situation when implementing a queue.

For a queue using a circular array as the container, the only complications are that we have to keep track of the index of the front item (which is now not necessarily at index `0`), and we have to **wrap** around the underlying array container when we get to the end of it. If **front** is the index of the front item, the index of the rear item is given by `(front + size - 1) % capacity`, where `capacity` is the length of the built-in array where we keep the actual queue. As in the case with the **linear** array, we add new items to the **rear** of the circular array, and remove old items from the **front** of the circular array. The index of the next available slot is then `(front + size) % capacity`.

In a *bounded* or fixed-size queue abstraction, the capacity stays unchanged, therefore when *rear* reaches *capacity*, the queue object throws an exception.

In an *unbounded* or dynamic queue abstraction when *rear* reaches *capacity*, we double up the queue size.

The following shows a partial linear array-based implementation of an *unbounded* queue.

```
public class ArrayQueue implements Queue {
    private Object[] data;      // The array container
```

```
    private int      front;     // The index of the front item
    private int      size;      // The number of items in the queue

    // Default initial capacity, which may then dynamically change
    private static final int DEF_INIT_CAPACITY = 100;

    public ArrayQueue() {
        data = new Object[DEF_INIT_CAPACITY];
        front = 0;
        size = 0;
    }

    // The rest of the implementation goes here...
}
```

The following show a queue through a sequence of queue operations using a circular array. Note that we don't need to keep a separate `rear` variable since it's equal to `(front + size - 1) % data.length`. And the next available position is simply `(front + size) % data.length`.

```
       0    1    2   3   4   5    6    7   8   9
     ---------------------------------------------
     |    |    |    |     |    |     |   |   |    |      size = 0
     ---------------------------------------------
     (front, rear undefined for an empty queue)

If we enqueue the elements 10, 20, 30, 40 and 50, we get:

       0    1    2   3   4   5    6    7   8   9
     ---------------------------------------------
     | 10 | 20 | 30 | 40  | 50 |     |   |   |    |      size = 5
     ---------------------------------------------
       ^-- front             ^-- rear

Now if we dequeue one element, we get:

       0    1    2   3   4   5    6    7   8   9
     ---------------------------------------------
     |    | 20 | 30 | 40  | 50 |     |   |   |    |      size = 4
     ---------------------------------------------
            ^-- front      ^-- rear

Now if we dequeue all four remaining elements, we get:

       0    1    2   3   4   5    6    7   8   9
     ---------------------------------------------
     |    |    |    |     |    |     |   |   |    |      size = 0
     ---------------------------------------------
     (front, rear undefined for an empty queue)

Now let's enqueue elements 5, 7, 9, 15, -4 and 17.

       0    1    2   3   4   5    6    7   8   9
     ---------------------------------------------
     | -4 | 17 |    |     |    |  5  | 7 | 9 | 15 |      size = 6
     ---------------------------------------------
           ^--rear              ^--front
```

To iterate over the items, we start at the front element and go to the rear, but unlike in the case of linear array, we may have to "wrap around" when we get to the end. It's often easiest to use two different indices to iterate through a cyclic array. For example:

```
int j = front;
for (int i = 0; i < size; i++) {
    visit(data[j]);
    j = (j + 1) % data.length;
}
```

Note how we're using `i` to count the number of times we need to iterate, and `j` to actually index into the underlying array. The variable `data` is a reference to the underlying array, which means that capacity is `data.length`, and the variable variable `size ≤ data.length` is the number of items in the queue.

If we wanted to iterate backwards (ie., from the rear end of the queue to the front), then:

```
  int j = (front + size - 1) % data.length;  // index of item in back
  for (int i = size-1; i >= 0; i--) {
      visit(data[j]);
      j--;
      if (j == -1)
          j = data.length - 1;
  }
```

Note that we can't use modulus operation when iterating backwards, but must explicitly set the index to the point to the last slot in the array container once we "fall off" the front.

Resizing a cyclic array means that we have to do the following:

1. Allocate a new array of the desired capacity
2. Copy the elements from the old cyclic array container to the new one
3. Reset the front index to `0`.

Here's an example: let's start with the following queue:

```
       0    1    2   3   4   5    6    7   8   9
     ---------------------------------------------
     | -4 | 17 |    |     |    |  5  | 7 | 9 | 15 |      size = 6
     ---------------------------------------------  capacity = 10
           ^--rear              ^--front

Now let's resize it to have a capacity of 11.

Here's the WRONG final answer (WHY?):

       0    1    2   3   4   5    6    7   8   9   10
     ------------------------------------------------
     | -4 | 17 |    |     |    |  5  | 7 | 9 | 15 |   |      size = 6
     ------------------------------------------------  capacity = 11
           ^--rear              ^--front

Here's one possible correct solution:
```

```
  0    1    2   3   4   5    6    7   8   9   10
-------------------------------------------------
| 5  | 7  | 9  | 15  | -4 |  17 |   |   |    |   |      size = 6
-------------------------------------------------  capacity = 11
  ^-- front                  ^--rear
```

is not really that tricky - it's actually quite easy if you use two different indices. One index goes from `front` to `front + size - 1`, and other one from `0` to `size - 1`. Again, see the notes on Circular Array available from Moodle course site to see how it all works.

The moral of the story is this: we use a circular array to avoid shifting items when we dequeue. The cost is that we have to maintain the position or index of the item in the front of the queue; give the number of items, we can then compute the position of the other items, and the position of the next available slot.

Back to Table of contents

Linked list based implementation

Linked List-based implementation provides the best (from the efficiency point of view) dynamic queue implementation.

The following shows a partial head-referenced singly-linked based implementation of an *unbounded* queue. In an singly-linked list-based implementation we add the new item being added at the end of the array, using a tail reference to append at constant cost (why?), and consequently remove the front item from the beginning of the list, again at constant cost.

```
public class ListQueue implements Queue {
    private Node head;          // Reference to the front of the queue
    private int  size;          // The number of items in the queue

    public ListQueue() {
        head = null;
        size = 0;
    }

    // The rest of the implementation goes here...
}
```

Back to Table of contents

Priority Queues

Table of contents

Introduction

You need to deposit a check at your bank, and so you go to the bank and there you see a single counter that says "Check Deposit". You of course pick that counter and stand at the back of the queue. You move slowly forward as the persons in front of you are serviced in a FIFO manner, until it's your turn. Since this is a FIFO queue, the earlier you arrive (your "priority"), the earlier you will receive service.

Your friend has to do the same, but at her bank, which is different than yours. Each customer gets a token with a number on it, and then waits in a nice Customer Service lounge with comfortable chairs until his or her number is called, which happens when one of the customer service agents is free. Note that there is no real queue in which people are standing, but there is an underlying queue which determintes who gets serviced next, the order being determined by the token number. The earlier you arrive, the smaller is the number on your token and the earlier you get serviced.

The differences between these two scenarios are just superficial - both systems behave exactly in the same way. In the first case, the "priority" is determined by your arrival order; in the second case, the "priority" is determined by a secondary "key" -- the token number -- which in turn depends on your arrival order. Both follow the FIFO principle. You can use a standard FIFO queue to model both the problems in exactly the same way. Also, note that once the priority is fixed, you cannot change that (let's assume that nobody else can cut in, or bribe in, while you're not looking).

Now, imagine that you want a system where the priority may depend on factors other than just the arrival time, and moreover, the priority may change at any time. In a queue of people waiting for a bus, a person with a small child infant may get higher priority and get on the bus before someone who may have gotten into the queue earlier. Or, let's say someone with much lower priority may pay a fee to "buy" higher priority, and jump ahead in line. This is actually quite different from a standard FIFO queue, in which the priority solely depends on the arrival time, and does not change until being removed from the queue. This is called a Priority Queue (PQ) because there is a priority that is not necessarily just the arrival order, and that it may change over time.

Note that a FIFO Queue is nothing but a Priority Queue with the priority or key directly proportional (or equal) to the arrival time; a LIFO Stack is also a Priority Queue with the property that the priority or key is inversely proportional to the arrival time.

As an example of what a priority queue is, imagine a sequence of integers that we're adding to a PQ, using

the value of the integer as the "key". After adding all the integers, let's remove these one by one from the PQ. There are two possibilities:

1. If a lower key represents higher priority, then the smallest integers would be removed first, followed by the smallest of the remaining ones, and so on. We will get the integers back in the non-decreasing order.
2. If a lower key represents lower priority, then the largest integers would be removed first, followed by the largest of the remaining ones, and so on. We will get the integers back in the non-increasing order.

We've just "discovered" another sorting algorithm - using a priority queue!

The interface looks just like a FIFO or LIFO, just that we explicitly specify a "priority" for each item, and we add an additional operation that allows us to change the priority of an item already in the PQ. The most basic operations are "add", "remove" and changePriority, and in addition we can have pretty much all the other operations that we had seen for Stack and Queue ADTs.

```
public interface PQueue {
    void add(Object item, Object key);
    Object remove();
    void changePriority(Object item, Object newKey);
}
```

Obviously, the "priority" Object must be Comparable, since we must be able to compare two different priorities together.

Here's an example of how to sort an array of integers using a priority queue. Assume that lower values for the key means that the object has higher priority, so will be removed earlier.

```
  /**
   * Sorts the array of integers.
   *
   * @param a the array of Integer objects to sort.
   */
  void sort(Integer[] a) {
      // Create an empty priority queue.
      PQ pq = new PQ();

      // Add all the integers in `a' to the PQ, using the integer value as
      // the key.
      for (int i = 0; i < a.length; i++)
        pq.add(a[i], a[i]);

      // Now extract the ones with the minimum key (hence highest priority)
      // from the PQ, and add it to the original array in sorted order.
      for (int i = 0; i < a.length; i++)
        a[i] = (Integer) PQ.remove();
  }
```

This will sort the array in non-decreasing order. How can you sort it in the non-increasing order? Two choices:

a. Make the priority queue understand that lower key means lower priority;

b. Sort first, and then reverse the array; or,
c. Add the Integer objects from the back.

```
for (int i = a.length-1; i >= 0; i--)
    a[i] = (Integer) PQ.remove();
```

Back to Table of contents

Implementation

The easiest implementation would use an array as the underlying container. The question is whether we would use an ordered/sorted or unordered/unsorted array (by the "key" of item) or not. Let's assume that the lower the key, the higher the priority (same as the non-decreasing order sorting example above).

Back to Table of contents

Unsorted array

If we use unsorted array, we don't care what the key is when adding the item, and simply add it to the end of the array (in the next available position in the array that is). Hence adding an item is independent of the number of items in the PQ, and takes constant time. Removing the next item however requires that we search for the lowest key value, and the remove it. Removing then requires searching at most `n` items in the PQ for the lowest key, and then may have to shift at most `n - 1` items to the left to fill up the gap left by the item removed. Even if we improved the search for the lowest key to $\log_2(n)$ using binary search, we still have to shift at most `n - 1` items to the left, so we don't gain much.

Note that removal is exactly the same as "Selection Sort"!

Worst case performance, given a PQ with `n` items in it

- **Adding**: 1 (or some other constant independent of `n`)
- **Removing**: `n` comparisons + `n - 1` shifts if we use linear search; or $\log_2(n)$ comparisons + `n - 1` shifts if we use binary search

Back to Table of contents

Sorted array

If we store the items in an array sorted by the key value (in non- increasing order), then the item with the smallest key is always in the last-used slot of the array, which removal a constant-time operation. Adding an item however takes much more time - we first have to find the proper insertion point, shift all the item from that point on right to create a gap, and then put the item in that gap. Searching for the insertion point (from the rear-end) requires at most `n` comparisons, following by at most `n` right-shifts to create a gap for the new item, and then a constant-time operation to put the item in the gap. We could modify binary search to find the insertion point in at most $\log_2(n)$ comparisons, but since we may still need at most `n` shifts, which doesn't help us much in the overall sense.

Note that adding looks is exactly the same as "Insertion Sort"!

Worst case performance, given a PQ with `n` items in it

- **Adding**: `n` comparisons + `n` shifts + 1 (or some other constant independent of `n`)
- **Removing**: 1 (or some other constant independent of `n`)

Back to Table of contents

Heap

This is the fun one! See Ch. 9 (9.1 - 9.6) for details.

Back to Table of contents

Basic searching and sorting

Table of contents

Introduction

We sort because we want to search faster, and that's all there is to it. If you didn't need to find something quickly, why would you bother storing it in any order in the first place! We'll start by looking at the very basic search techniques, and look at the basic sorting techniques after that.

Searching

We'll consider two different types of search in the basic category: sequential search through a sequence, and binary search through a sorted random-access sequence.

Sequential search

The easiest way to search for an item in a sequence is to start at the beginning, look at each item to see if there's a match. We stop if we find the item, or if we go past the end of the sequence, which means that the item does not exist in the sequence. This is the so-called **sequential search**, which requires **`n` comparisons in the worst-case for an n-element sequence**, which is rather slow for real use. However, it does have advantages: (a) it does not require that the container support random access (so a list would do), and (b) it does not require that the data be sorted.

The following shows how to sequentially search for a `key` in an array `data`, where the first `size` elements are used (where `size ≤ data.length`). If all the slots in the array are used, then you can simply substitute `data.length` for `size` (because `size == data.length` in that case).

```
/**
 * Searches for the given key in the array of size elements.
 *
 * @param data the array with the keys
 * @param size the number of keys in the array (≤ data.length)
 * @param key  the key to search for in the array
 * @return the position of the key if found, or -1 otherwise.
 */
```

```
public static int seqSearch(Object[] data, int size, Object key) {
    for (int i = 0; i < size; i++)
        if (key.equals(data[i]))
            return i;           // return index of item if found

    return -1;                  // return sentinel if not found
}
```

Similarly, the following shows how to sequentially search for a `key` in a linked list, where `list` is a reference to the first or head node.

```
/**
 * Searches for the given key in the list.
 *
 * @param head reference to the head node in the list
 * @param key  the key to search for in the array
 * @return the position of the key if found, or -1 otherwise.
 */
public static int seqSearch(Node head, Object key) {
    Node n = head;
    for (int i = 0; n != null; i++, n = n.next)
        if (key.equals(n.element))
            return i;           // return index of item if found

    return -1;                  // return sentinel if not found
}
```

Back to Table of contents

Binary search

If the data in the sequence is already **sorted**, and the sequence supports **random access** (if it's an array that is), then we can significantly improve the search peformance by using **binary search**. We're already quite familiar with this search technique — we use something similar when we look up a word in a dictionary, or a person in a phonebook. We see if the item is in the middle of the array. If so, we've found it. If not, we check if the item is larger or smaller. If it's larger, then we look at the right half of the array (to the right of the middle element), or else we look at the left half (to the left of the middle element). We continue dividing up the array, until either we find the item, or we run out of data to search (ie., the item wasn't there in the first place).

The following shows how to search for a `key` in an array `data` of sorted elements, where the first `size` elements are used (where `size ≤ data.length`). If all the slots in the array are used, then you can simply substitute `data.length` for `size` (because `size == data.length` in that case).

```
/**
 * Searches for the given key in the array of size elements.
 * Pre-condition: data must be sorted in non-decreasing order.
 *
 * @param data the array with the keys
 * @param size the number of keys in the array (≤ data.length)
 * @param key  the key to search for in the array
 * @return the position of the key if found, or -1 otherwise.
 */
```

```
public static int binSearch(Object[] data, int size, Object key) {
    int l = 0;
    int r = size - 1;
    while (l <= r) {
        int mid = (l + r)/2;
        if (key.equals(data[mid]))
            return mid;             // return index of item if found
        else if (((Comparable) key).compareTo(data[mid]) > 0)
            l = mid + 1;
        else
            r = mid - 1;
    }
    return -1;                      // return sentinel if not found
}
```

The question now is how fast (or slow) is binary search? One way to answer this question is find out the maximum number of comparisons one must make to find an item in an sorted array of `n` elements. The maximum number of comparisons happen in the "worst-case" scenario, in which we either find the item when the array size is just 1, or if it's not there at all. The following table shows the number of comparisons at each step.

Step	Array size	# of comparisons
	$n = n/2^0$	1
1	$n/2 = n/2^1$	1
2	$n/4 = n/2^2$	1
3	$n/8 = n/2^3$	1
.	.	1
.	.	1
.	.	1
k	$= n/2^k$	1
.	.	1
.	.	1
.	.	1
?	1	1

We start with an array size of `n`, and make a single comparison (with the item in the middle of the array). Then we continue with either the left half or the the right half (actually it's slightly less than `n/2`, but we can ignore such details when `n` is very large), and at each step make a single comparison. In the worst-case, we eventually end up with an array of just one item (`n == 1`), which is either the item ("found"), or not ("not found"). To find the maximum number of comparisons, we simply need to add up the 3rd column, but for that we need to know tall the column is, that is the number of steps! So, if we can find out the number of steps it takes for `n` to go down to one when we divide by 2 at each step, we can find out the maximum number of comparisons.

After `k` steps, we have an array size of $n/2^k$. So, if $n/2^k == 1$, `k` is number of steps we're looking for.

$$
\begin{aligned}
n/2^k &= 1 \\
2^k &= n \\
k &= \log_2(n)
\end{aligned}
$$

So, we have $k = \log_2(n)$ steps, and at each step, we make 1 comparison. So the total number of comparisons is then $k * 1 = \log_2(n) * 1 = \log_2(n)$.

In summary, in the worst-case, the number of comparisons made by **sequential search is n**, and by **binary search is $\log_2(n)$**.

It's summarized in the table below.

```
Worst-case performance:

    search algorithm | max # of comparisons
    -------------------------------------
       sequential    |    n
       binary        |    log2(n)
    -------------------------------------
```

Back to Table of contents

Sorting

We will look at 3 very basic sorting algorithms - bubble, selection and insertion sorts. Bubble is simply too silly to bother about, so let's take a look at selection first.

Back to Table of contents

Bubble sort

We're skipping bubble ... but you can read up on it on Wikipedia.

Back to Table of contents

Selection sort

The idea is very simple — we select the minimum element in the sequence, and move it to the first position, then select the second minimum element and move it to the second position, and so on until we've placed all the elements in its right place. Selecting the minimum is trivial — just iterate through the elements picking out the minimum. Selecting the second minimum is also trivial — **after** placing the minimum in the first position, we select the minimum of the rest of the sequence (that is, from the second position onwards). After placing the second minimum in the second position, we find the third minimum by find the minimum in the rest of the sequence, and so on.

At any given time, the array is partitioned into two areas — the left part is sorted, and the right part is being processed. And at each step, we move the partition one step to the right, until the entire array has been processed.

The following shows the sequence of steps in sorting the sequence `< 17 3 9 21 2 7 5 >`. In the example below, the "|" symbol shows where the partition is at each step.

```
    Input: 17 3 9 21 2 7 5
    n = 7

    Step 1: | 17 3 9 21 2 7 5   << minimum is 2, exchange with 17
    Step 2: 2 | 3 9 21 17 7 5   << minimum is 3, no exchange needed
    Step 3: 2 3 | 9 21 17 7 5   << minimum is 5, exchange with 9
    Step 4: 2 3 5 | 21 17 7 9   << minimum is 7, exchange with 21
    Step 5: 2 3 5 7 | 17 21 9   << minimum is 9, exchange with 17
    Step 6: 2 3 5 7 9 | 21 17   << minimum is 17, exchange with 21
          : 2 3 5 7 9 17 | 21   << STOP
```

At each step, we only consider the array to the right of the partition, since the part to the left is already sorted, and does not need to be dealt with. Also, note that we iterate `n-1` times, because in the last step, there is just a single item left (21 in this case), which is obviously the minimum.

```
/**
 * Sort the specified array using selection sort.
 *
 * @param data the array containing the keys to sort
 */
public static void selectionSort(Object[] data) {
    for (int i = 0; i < data.length - 1; i++) {
        // Find the index of the minimum item, starting at `i'.
        int minIndex = i;
        for (int j = i + 1; j < data.length; j++) {
            if (((Comparable) data[j]).compareTo(data[minIndex]) < 0)
                minIndex = j;
        // Exchange with the first item (at `i'), but only if different
        if (i != minIndex) {
            Object tmp = data[i];
            data[i] = data[minIndex];
            data[minIndex] = tmp;
        }
    }
}
```

Few things to note:

1. If only the first `size` (where `size ≤ data.length`) elements are used in the `data` array, then the iteration should stop at `size` instead of `data.length`.
2. We only need to advance through the sequence, and do not need random access. This means that we can perform selection sort on a list just as easily.

The following algorithm shows how to sort a singly-linked list using selection sort.

```
/**
 * Sort the specified list using selection sort.
 *
 * @param head reference to the first node of the list to sort
 */
public static void selectionSort(Node head) {
    // Empty list or list with a single element is already sorted.
    if (head == null || head.next == null)
        return;
```

```
    for (Node n = head; n.next != null; n = n.next) {
        // Find the node with the minimum item, starting at `n'.
        Node minNode = n;
        for (Node p = n.next; p != null; p = p.next) {
            if (((Comparable) p.element).compareTo(minNode.element) < 0)
                minNode = p;
        // Exchange with the first item (within `n'), but only if different
        if (n != minNode) {
            Object tmp = n.element;
            n.element = minNode.element;
            minNode.element = tmp;
        }
    }
}
```

How many comparisons do we make in sorting a sequence of `n` items using this algorithm? At each step, we have to find the minimum item to the right the partition.

```
  Step   |  Array size |  Max # of comparisons
  ----------------------------------------
    1    |   n         |        n - 1
    2    |   n - 1     |        n - 2
    3    |   n - 2     |        n - 3
    .    |     .       |          .
    .    |     .       |          .
    .    |     .       |          .
 n - 1   |     2       |          1
  ----------------------------------------
```

The total number of comparisons in the worst case is simply the sum of the 3rd column, which is an arithmetic series: $1 + 2 + 3 + \ldots + (n - 1) = n(n-1)/2 \sim n^2$.

Does it help if the data is already sorted? No. Regardless of the input, selection sort will always make $\sim n^2$ comparisons, so the best- and worst- cases are the same. Selection is **not adaptive**, which is one of its weaknesses. See Discussion for more on this issue.

Back to Table of contents

Insertion sort

Insertion sort also works by partitioning the array into sorted and being processed parts, but works very differently than selection sort. At each step, it takes the next "key" in the array, and moves it left until it is in its rightful place, and moves the partition one step to the right until the whole array is sorted. Since an one-element array is already sorted, we start by moving the second element (at index 1).

The following shows the sequence of steps in sorting the sequence ‹ 17 3 9 21 2 7 5 ›. In the example below, the "|" symbol shows where the partition is at each step.

```
    Input: 17 3 9 21 2 7 5
    n = 7

    Step 1: 17 | 3 9 21 2 7 5   << move 3 to the left
    Step 2: 3 17 | 9 21 2 7 5   << move 9 to the left
```

```
    Step 3: 3 9 17 | 21 2 7 5   << move 21 to the left
    Step 4: 3 9 17 21 | 2 7 5   << move 2 to the left
    Step 5: 2 3 9 17 21 | 7 5   << move 7 to the left
    Step 6: 2 3 7 9 17 21 | 5   << move 5 to the left
          : 2 3 5 7 9 17 21 5 | << STOP
```

At each step, we essential insert the key to the right of the partition into the sorted partition by scanning left. In the best case, it stays in its place (see when we moved 21 to the left); in the worst-case, it moves all the way to the first position (see when we moved 2 to the left). Also, note that we iterate `n-1` times.

```
/**
 * Sort the specified array using insertion sort.
 *
 * @param data the array containing the keys to sort
 */
public static void insertionSort(Object[] data) {
    // i denotes where the partition is
    for (int i = 1; i < data.length; i++) {
        // the key is to the right of the partition
        Object key = data[i];
        int j = i - 1;      // use j to scan left to insert key
        while (j >= 0 && ((Comparable) key).compareTo(data[j]) < 0) {
            // shift item right to make room
            data[j + 1] = data[j];
            j--;
        }
        // Found the position where key can be inserted
        data[j + 1] = key;
    }
}
```

Few things to note:

1. If only the first `size` (where `size ≤ data.length`) elements are used in the `data` array, then the iteration should stop at `size` instead of `data.length`.
2. The outer loop advances through the sequence, so we don't need random access. The inner loop also advances, but in the reverse direction. While we don't need random access, we do need to move in both directions, which means we at least need a doubly-linked list.
3. Insertion sort does not require that all the keys are already in place, like selection sort does (since it needs to scan to find the minimum element). In fact, insertion sort is very good at sorting data that arrives one element at a time, which is a typical scenario on *network algorithms*. This is a significant advantage of insertion sort over other comparable sorting algorithms of similar performance.

How many comparisons do we make in sorting a sequence of `n` items using this algorithm? At each step, we have to find the rightful place for the key by comparing it with item to the left of the partition. In the worst case, each key moves all the to the first position (which would happen if the input sorted, but in the reverse order), shifting all the sorted elements by one position to the right.

```
Step   | Sorted part size |  Max # of comparisons | Max # of shifts
-------------------------------------------------------------------
  1    |   1              |        1              |       1
  2    |   2              |        2              |       2
```

```
  3    |   3             |        3             |       3
  .    |   .             |        .             |       .
  .    |   .             |        .             |       .
  .    |   .             |        .             |       .
n - 1  |  n-1            |      n - 1           |     n - 1
-------------------------------------------------------------------
```

The total number of comparisons is simply the sum of the 3rd column, which is an arithmetic series: `1 + 2 + 3 + ... + (n - 1) = n(n-1)/2 ~ n^2`. The total number of shifts is the same: `n(n - 1)/2 ~ n^2`.

Does it help if the data is already sorted? YES! We make a single comparison at each step, since the key does not move at all. The total number of comparisons then is just `n-1`, instead of `~n^2`. This is the **best-case** performance for insertion sort. What if the input is **nearly sorted**? It also significantly improves the performance of an insertion sort over the worst-case, which makes it an **adaptive** sort. See Discussion for more on this issue.

In the **average case** however, if the input is randomly distributed, each new key is likely to travel only about halfway, which leads to slightly better performance. It's still `~n^2` however, so it's the same order as the worst-case.

```
Worst-case performance:

    sorting algorithm | max # of comparisons
    --------------------------------------
       selection      |    n(n-1)/2 ~ n^2
       insertion      |    n(n-1)/2 ~ n^2
    --------------------------------------

    sorting algorithm | max # of exchanges/shifts
    -------------------------------------------
       selection      |    n(n-1)/2 ~ n^2
       insertion      |    n(n-1)/2 ~ n^2
    -------------------------------------------

Best-case performance:

    sorting algorithm | max # of comparisons
    --------------------------------------
       selection      |    n(n-1)/2 ~ n^2    << doesn't change
       insertion      |    n - 1             << when input is sorted
    --------------------------------------

    sorting algorithm | max # of exchanges/shifts
    -------------------------------------------
       selection      |        0
       insertion      |        0
    -------------------------------------------
```

Back to Table of contents

Discussion

Given that both selection and insertion sorts take $\sim n^2$ comparisons in the worst-case, which one would you use? Let's ignore the best-case scenario for now, since that's really not a useful metric in real-life, other than its use in (often false) advertising. Let's look at the relative strengths of these sorting algorithms.

- To use using selection sort, you must have **all** the data already available in a sequence, since we have to find the minimum value to put in the first slot, the 2^{nd} to put in the second slot, and so on. As the partition moves to the right, the left (sorted) part is never touched again. For insertion sort, each new key is inserted into the left part in its rightful place, so while the left part is always sorted, it gains new keys as sorting progresses. The significant advantage of this approach is that insertion sort can deal with single keys arriving one at a time, and maintains a sorted sequence at any time by inserting the new key in its rightful place. The analogy we used in class was the difference between download a mp4 to view (have **all** the data available before you are able to play the video), or to stream a video from the network and view it real-time (see each frame **as it comes**, provided you have the bandwidth of course). This is one reason for choosing insertion sort.
- You should have noticed that selection sort performance does not improve even if the input data is already sorted, but insertion sort does take advantage of that. This makes insertion sort **adaptive**, which is a desirable property in any algorithm. For a **nearly sorted** input, insertion sort performs significantly better than selection sort.
- In the algorithm outlined above for selection sort, we exchange the minimum with the current element, which may move the current element far away from its current position. This make **selection sort**, as it's done in these notes, **unstable**. One can make selection sort stable by inserting the minimum element in the first position (**before** the current element that is) instead of exchanging it with the current element. This is fairly expensive for arrays, but can be made cheap for lists. **Insertion sort** is **stable** if we use < relation to compare the new key with the elements in the left part.
- Neither sorts require **random access**, so can be done on a list. However, insertion sort does require **bi-directional** access, which means that the list must be **doubly-linked**.

Except for the case where the sequence is a singly-linked list, insertion sort tends to perform (often much) better than selection sort. Its adaptive nature also helps its performance when the input is already partially or nearly sorted.

Back to Table of contents

Key-indexed searching and sorting

Table of contents

Introduction

If the keys used in searching are integers only, we can use key-indexed searching to search in constant time! Compare that to the best one we've seen so far — $\log_2(n)$ in the case of binary search.

Let's say we have an array A of non-negative (why?) integers that we want to search for very quickly. If we think of implementing a set or a map (also called a dictionary), we can do the following: Use the values in A (the keys) as indices into another array B, which must have a capacity of one greater than the largest key in A. (Why?)

Let's say we're given an array `A = [5 1 7 2 8 4]`, and we want to quickly search for some `key` in `A`. We first create another array `B` which has a capacity of one larger than largest element in `A`, which is `8 + 1 = 9`. Then we populate `B` with the elements in `A` as indices into `B`. This is what we call the **setup** phase below, shown below.

```
      0   1   2   3   4   5
     -------------------------
A =  | 5 | 1 | 7 | 2 | 8 | 4 |
     -------------------------

Create B, initially with all 0's in it.

      0   1   2   3   4   5   6   7   8
     -----------------------------------
B =  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
     -----------------------------------

Populate B, by using elements of A as indices into B.

      0   1   2   3   4   5   6   7   8
     -----------------------------------
B =  | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
     -----------------------------------

For each element of A, the corresponding slot in B is set to 1.
```

Searching for any `key` can be done by simply indexing into `B` using the `key` as an index — if the slot is used (that is, the value is `1`), then the `key` exists in `A`; otherwise, it does not. Searching for `8` in `A`, we see that `B[8] == 1`, so `8` exists in `A`; searching for `6` in `A`, we see that `B[6] == 0`, so `6` does not exist in `A`; and so on.

Extending this to support non-negative integers is trivial, which we discuss at the end in Extensions.

We can also use this technique to sort in linear time! Compare that to the best we've seen so far — n^2 for bubble/selection/insertion sorts. An added restriction on the input is that the keys must be distinct (why?) in addition to being non-negative.

Let's say we have to sort the a `A = [5 1 7 2 8 4]` that we used in our search example. The **setup** phase is exactly the same — we create `B` and then populate it using the elements of `A` as indices into `B`. Now we can walk through `B` and output the elements that are set to `1`, and we get the sorted output.

```
Populated B, by using elements of A as indices into B.

      0   1   2   3   4   5   6   7   8
     -----------------------------------
B =  | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
     -----------------------------------

Iterate through B, and for each element that is 1, append the value of
the index to A.

      0   1   2   3   4   5
     -------------------------
A =  | 1 | 2 | 4 | 5 | 7 | 8 |
     -------------------------
```

Extending it to support non-negative integers is the same as in the case of search. Extending this to support non-distinct integers is possible, but only under certain conditions, which we discuss at the end in Extensions.

Back to Table of contents

Searching

INPUT: array of `n` non-negative distinct integers `A[0 ... n-1]`
INPUT: an non-negative integer `key` to search for
OUTPUT: True if the key exists, or false otherwise. The actual search must be done in constant time, even if building the intermediate data structure takes longer (linear for example).

Setup:

1. Find the maximum value `k` in the input, and create a temporary array B that has a capacity of `k + 1` (why the +1?), initializing the elements to 0 (automatically done in Java).

```
int k = A[0];
for (int i = 1; i < A.length; i++)
     if (A[i] > k)
```

```
            k = A[i];

int[] B = new int[k + 1];
```

2. Iterate through the keys in `A`, and using each element in `A` (key) as the index into `B`, set it to `1` (or any non-zero value).

```
for (int i = 0; i < A.length; i++)
    B[A[i]] = 1;
```

Search:

3. Once the array `B` is ready, search for a key takes constant time. If the key is outside the valid range, it's trivially false; otherwise, if the element indexed by the key is `1` it's true.

```
if (key < 0 && key >= B.length)
    return false;
else
    return B[key] != 0;
```

Search performance

For an input of `n` elements, building `B` takes `n` assignment operations, so the running time is a **linear function** of `n`. The actual search operation, after `B` is populated, is done in **constant time**! First of all, note that the algorithm is **out-of-place**, which means we have to consider the space overhead as well as running time. The extra space required is the capacity of `B`, which is equal to the maximum value in the input (`k` in step 1). If `k` is roughly of the order as `n`, the overhead is justified; however, if `k` is much larger than `n`, then the space overhead may not be justified. The space overhead is completely unjustified if your input is `[5, 13, 2, 10^8, 9]` — for just `5` elements, you'll need a temporary array `B` with a capacity of `~10^8`!

Back to Table of contents

Sorting

We can also sort in linear time! Compare that to the n^2 sorting algorithms we've seen so far in this course. We'll see much better sorting algorithms later on, but still nothing as good as linear time.

INPUT: array of `n` non-negative distinct integers `A[0 ... n-1]`
OUTPUT: elements of A are in sorted in non-decreasing order.

Setup: Use steps 1 and 2 from the searching case to build the array B.

1. Same as step 1 in the search case above.
2. Same as step 2 in the search case above.

Sort:

3. Once the array B is ready, iterate through B and fill in A.

```
int i = 0;
for (int k = 0; k < B.length; j++) {
   if (B[k] != 0) {
       a[i++] = k;
   }
}
```

Back to Table of contents

Sort performance

Building `B` takes at most `n` operations (assignments in this case). Filling in `A` using `B` takes at most `k` assignments, where `k` is the maximum value in the input. So the running time is proportional to `n + k`, which means the running time depends on not just the size of the input (`n`), but also the magnitude of the input (`k`) — the first of its kind we've seen in this course. If `k` is roughly of the same order as `n`, then we have a **linear** sorting algorithm. However, if `k` is much larger than `n`, it may not be worth using this algorithm.

For the space overhead, the same argument as in the search case applies here as well.

Back to Table of contents

Extensions

How do we handle the case of negative keys? And, what about non-distinct keys? We discuss these two extensions below.

Back to Table of contents

Negative keys

We can easily accommodate negative keys by "shifting" the input such that the elements form valid array indices; the easiest way is to "shift" the input by the absolute of the minimum input value such that all the values are now positive (and thus valid array indices).

When searching for a key, we must first shift the key before using it as an index to see if it exists or not.

When sorting, we must shift the keys back when filling in `A` using `B`.

Back to Table of contents

Non-distinct keys

This is not a problem for searching, since we only need to know if a key is in the input or not, not how many times it occurs in the input. However, for sorting, the output will contain only one copy of each key,

which means that the output will not be correct if the input contains non-distinct keys.

Let's change the setup (`B`) and sorting slightly to accommodate non distinct keys:

Setup:

1. Step 1 is unchanged.
2. Instead of just setting the elements in `B` to `0` (does not exist) or `1` (exists), we count the number of occurrences of the key (`>= 0`).

```
for (int i = 0; i < A.length; i++)
    B[A[i]] = B[A[i]] + 1;
```

Sorting:

3. Once the array `B` is ready, iterate through `B` and fill in `A`, keeping track of the number of copies of each key.

```
int i = 0;
for (int k = 0; k < B.length; j++) {
    while (B[k] > 0) {
        a[i++] = k;
        B[k] = B[k] - 1;
    }
}
```

Now we can sort an array of non-distinct integer keys. Will the output be **stable**?

While this works for input consisting of just integer keys, it does not work when we have key/value pairs. The multiple values corresponding to the same key will get overwritten by one of the values, so you get the wrong answer. Consider sorting the following key/value pairs: `[{9,"a"}, {5,"x"}, {2,"n"}, {5,"y"}, {9,"t"}]`. The array B as used here will not suffice, since it can just hold the key, not the value. If we change `B` to hold key/value pair, it will still hold a single value. We can of course extend `B` to be an array of lists, where the list keeps the multiple values for each key. The values must be sorted as well — either maintained as a sorted list, or sorted when filling `A`. Of course, the data structure just got a whole lot more complicated, and beginning to feel like a lot of work! This is the same **Adjacency List** data structure we used for representing a **graph** (coming in a few weeks) by the way. We will also use it for **hashtables**, also coming in a few weeks.

Back to Table of contents

Recursion

Table of contents

Introduction

Recursion is a way of **thinking about** problems, and a **method for solving** problems. The basic idea behind recursion is the following: it's a method that solves a problem by solving *smaller* (in size) versions of the same problem by breaking it down in smaller subproblems. Recursion is very closely related to **mathematical induction**.

We'll start with recursive definitions, which will lay the groundwork for recursive programming. We'll then look at a few prototypical examples of using recursion to solve problems. We'll finish with looking at problems and issues with recursion.

Back to Table of contents

Recursive definitions

We'll start **thinking recursively** with a few recursive definitions:

1. A list of numbers
2. Factorial
3. Fibonacci numbers

List of numbers

Consider the (*non-empty*) list of numbers `‹24, 88, 30, 37›`. How would we *define* such as list? The first one that comes to mind, at least at this point into this semester, would probably be the following:

```
A list is one or more numbers, separated by commas.
```

The **recursive** way of thinking would produce the following definition:

```
A list is either a: number
             or, a: number comma list
```

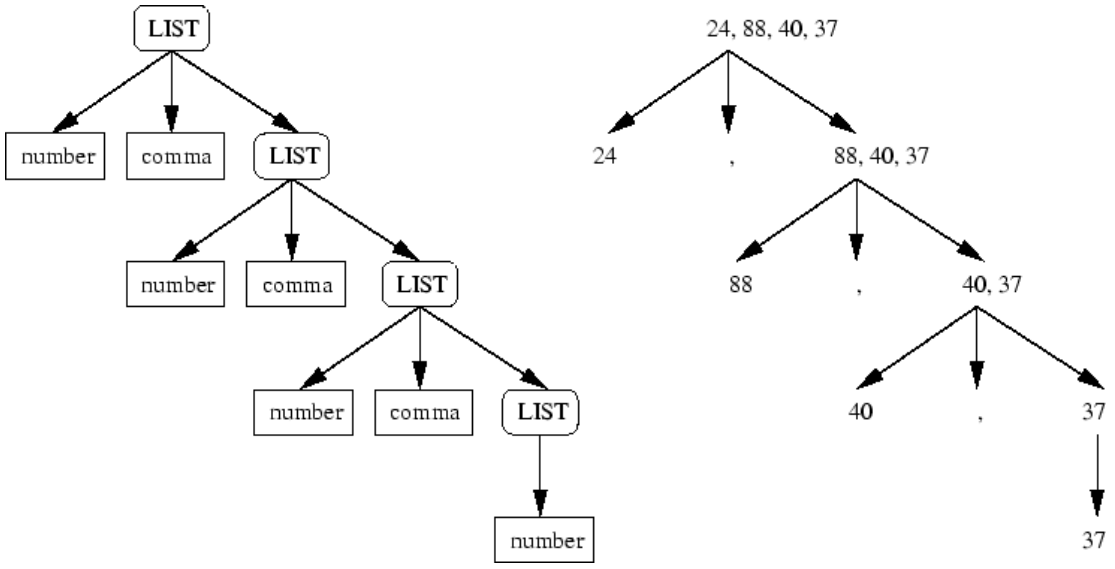
That is, a *list* is either a single number, or a number followed by a comma followed by a *list*. Two things to note here:

1. We're using *list* to define a *list*, which is to say that the concept of a *list* is used to define itself!
2. There is a **recursive** (the 2nd) part in the definition, and a **non-recursive** (the 1st) part in the definition, which is called the **base case** or the **stopping condition**. Without the base case, the recursion would never stop, and we would end up with *infinite recursion*!

With this **recursive definition** of a *list*, we can now ask whether ‹ 24 › is a *list* or not. It matches the 1st condition, which says that it's a single number, so it is indeed a *list*. What about ‹ 24,88 ›? Here we have a number 24 followed by a comma followed ‹ 88 ›; since we now know that ‹ 88 › is indeed a *list*, we can rephrase it as such: we have a number 24 followed by a comma followed by a *list* ‹ 88 ›, so it matches the 2nd definition, and so it is indeed a list. If we try a bit longer list of numbers ‹ 24,88,30,37 ›, we see it does indeed match our definition. Note how the recursive part of the *list* definition is used several times, terminating with the non-recursive part (the *base condition*, which stops the recursion).

```
number comma     list
  24     ,    88, 30, 37

            number comma   list
              88     ,    30, 37

                        number comma  list
                           30    ,    37

                                      number
                                       37
```

The following shows a graphical view of tracing this recursive definition of a *list* using the sequence ‹ 24,88,30,37 ›. This is often referred to as the **recursion tree** diagram.



Factorial

The *factorial* of a non-negative integer n is defined to be the product of all positive integers less than or equal to n. For example, 5! = 5 × 4 × 3 × 2 × 1 = 120. See Wikipedia's entry on Factorial for

more info.

```
1! = 1
2! = 2 × 1
3! = 3 × 2 × 1
4! = 4 × 3 × 2 × 1
5! = 5 × 4 × 3 × 2 × 1
...
and so on.
```

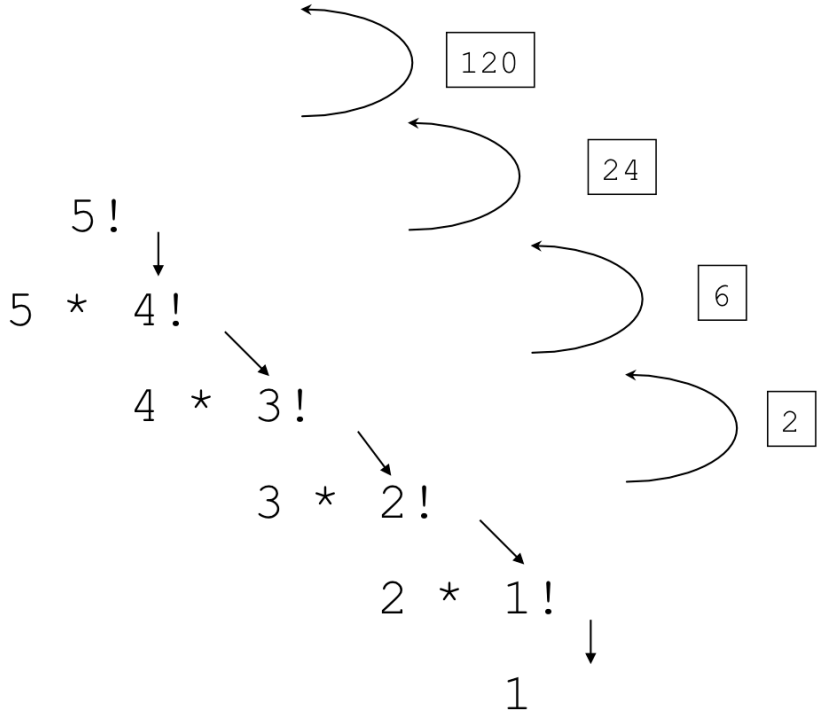
We can easily evaluate `n!` for any valid value of `n` by multiplying the values iteratively. However, there is a much more interesting recursive definition quite easily seen from the *factorial* expressions: `5!` is nothing other than `5 × 4!`. If we know `4!`, we can trivially compute `5!`. `4!` on the other hand is `4 × 3!`, and so on until we have `n! = n × (n - 1)!`, with `1! = 1` as the base case. The mathematicians have however added the special case of `0! = 1` to make easier (yes, it does, believe it or not). For the purposes of this discussion, we'll use both `0! = 1` and `1! = 1` as the two base cases (we can always add `2! = 2` as another base, as long as we keep the necessary one — `0!`).

```
    -
    | 0              if n = 1
    | 1              if n = 1
n! =|
    | n × (n - 1)!    if n > 1
    -
```

Now we can expand `5!` recursively, stopping at the base condition.

```
5! = 4 × 4!
         4 × 3!
             3 × 2!
                 2 × 1!
                     1
```

The recursion tree for `5!` shows the values as well.



Fibonacci numbers

The Fibonacci numbers are ‹ `0, 1, 1, 2, 3, 5, 8, 13, ...` › (some define it without the leading `0`, which is ok too). If we pay closer attention, we see that each number, except for the first two, is nothing but the sum of the previous two numbers. We can easily compute this iteratively, but let's stick with the recursive method. We already have the **recursive** part of the **recursive definition**, so all we need is the **non-recursive** part or the **base case**. The mathematicians have decided that the first two numbers in the sequence are `0` and `1`, which give us the base cases (notice the two base cases). See [Wikipedia's entry on Fibonacci number] for more info.

Now we can write the recursive definition for any Fibonacci number `n ≥ 0`.

```
          -
          | 0                          if n = 0
          | 1                          if n = 1
fib(n) = |
          | fib(n-1) + fib(n-2)        n ≥ 2
          -
```

We can now compute `fib(5)` using this definition.

```
                    fib(5)
                     +
                  /       \
                /           \
            fib(4)          fib(3)
              +               +
           /     \         /     \
         /         \     /         \
      fib(3)     fib(2) fib(2)    fib(1)
       +          +      +
```

```
      /    \        /   \      /  \
     /      \      /     \    /    \
    f        f    f       f   f     f
    i        i    i       i   i     i
    b        b    b       b   b     b
   (2)      (1)  (1)     (0) (1)   (0)
    +
   / \
  ....
```

Before moving on, you should note how many times we're computing Fibonacci of 3 (`fib(3)` above), and Fibonacci of 2 (`fib(2)` above) and so on. This **redundancy in computation** leads to gross inefficiency, but something we can easily address to **Memoization**, which is the next topic we study.

Back to Table of contents

Recursive programming

A **recursive function** is one that calls itself, since it needs to solve the same problem, but on a smaller-sized input. In essence, a recursive function is a function that is defined in terms of itself.

Let's start with computing the factorial. We already have the recursive definition, so the question is how do we convert that a recursive method in Java that actually computes a factorial of a given non-negative integer. Here's the recursive definition again, along with the Java code. This is an example of *functional recursion*.

```
    -                                   static int fact(int n) {
    | 1              if n = 0             if (n == 0 || n == 1)
    | 1              if n = 1                 return 1;
n! =|                                       else
    | n × (n - 1)!   if n > 1                return n * fact(n - 1);
    -                                   }
```

The first thing to note is how similar the code in `recursiveSum` is to the recursive definition. Once you have formulated the recursive definition, the rest is usually quite trivial. This is indeed one of the greatest advantages of recursive programming.

Let's now look an example of *structural recursion*, one that uses a **recursive data structure**: we want to compute the sum of the list of numbers `a = ‹3 8 2 1 13 4›`. The "easiest" (at least in Java, but certainly not in say Lisp or Scheme) way to solve it **iteratively**, as shown below.

```
static int iterativeSum(LinkedList list) {
    int sum = 0;
    Node n = list.head;
    while (n != null) {
        sum = sum + n.item;
        n = n.next;
    }
    return sum;
}
```

Now since we already know that a list of numbers has a recursive definition, meaning that it's a recursive data structure, any problem defined on it must also have a simple recursive solution. Thinking recursively, we note the following:

1. The sum of the numbers in a list is nothing but the sum of the first number **plus** the sum of the **rest of the list**. The problem of summing the **rest of the array** is the same problem as the original, except that it's smaller by one element! Ah, recursion at play.
2. The shortest list has a single number, which has the sum equal the number itself. Now we have a base case as well. Note that if we allowed our list of numbers to be empty (we said non-empty when defining it), then the base case will need to be adjusted as well: the sum of an empty list is simply `0`.

Note the key difference between the iterative and recursive approaches: **for each number in the list** vs **the rest of the list**.

Now we can write the **recursive definition** of this problem, and using that, write the recursive method:

```
      -
      | k                k is the only element
sum = |
      | k + sum(n.next)  otherwise
      -


static int recursiveSum(LinkedList list) {
    if (list.size() == 1)
        return list.head.item;
    else {
        return list.head.item + recursiveSum(list.next);
    }
}
```

Again, note how similar the code in `recursiveSum` is to the recursive definition.

Our familiar *linked list* is a **recursive data structure** since it's defined in terms of itself. The following is a recursive definition of a `linked list`:

```
A linked list is either empty, or it is a node followed by another
linked list.
```

We've of course seen this already when discussing recursive definition — this is almost exactly the same as our example of list of numbers, with only base case differing — we allow a linked list to be empty.

Back to Table of contents

Examples

We look at the following examples:

1. Length of a String
2. Length of a linked list
3. Sequential search in a sequence

4. Binary search in a sorted array
5. Finding the maximum in a sequence (linear version)
6. Finding the maximum in an array (binary version)
7. Selection sort
8. Insertion sort
9. Sum of integers 1 through N
10. Exponentiation – a^n

Length of a String

A character string is a recursive structure: a string is either empty, or a character followed by the rest of the string. This implies that recursion is a *natural* way to solve string related problems. Let's take the case of computing the length of a string; the length of "abc" is 1 longer than the length of the rest of the string, which is "bc"; the length of "bc" is 1 longer than the length of "c"; and the length of "c" is 1 longer than the length of "", which is the empty string (and hence has a length of 0). We now have our recursive case and a base case! An example of what is known as *structural recursion*.

Since we're using Java's String class to represent a character string, The length of a Java String object has the following recursive formulation: the length is 1 **plus** the length of the **rest of the String**. In Java, the `String` provides a `substr` method to extract the rest of the String: `s.substring(1)` produces the substring from index 1 onwards. The recursion stops when the String is empty, which gives us the base case.

```
          -
          | 0             if string s is empty
len(s) = |
          | 1 + len(rest) otherwise
          -


static int strLength(String s) {
    if (s.equals(""))
        return 0;
    else
        return 1 + strLength(s.substring(1));
}
```

Length of a linked list

A linked list is also a recursive structure: a list is either empty, or a node followed by the rest of the list. We can also compute the length of a list recursively as follows: the the length of a list is 1 longer than the rest of the list! The empty list has a length of 0, which is our base case.

```
          -
          | 0             if l is an empty list
len(l) = |
          | 1 + len(rest) otherwise
          -


static int listLength(List l) {
    if (l == null)
```

```
        return 0;
    else
        return 1 + listLength(l.next);
}
```

Sequential search in a sequence

How would you find something in a list? Well, look at the first node and check if the `key` is in that node. If so, done. Otherwise, check the **rest of the list** for the given `key`. If you search an empty list for any `key`, the answer is false, so that's our base case. This is almost exactly the same, at least in form, as finding the length of a linked list, and also an example of *structural recursion*.

```
                -
                | false              if list l is empty
                | true               if l.item = k
contains(l,k) = |
                | contains(l.next,k) otherwise
                -


static boolean contains(List l, int k) {
    if (l == null)
        return false;
    else if (l.item == k)
        return true;
    else
        return contains(l.next, k);
}
```

What if the sequence is an array? How do we deal with the **rest of the array** part then? We can handle it in two ways:

1. We can create a new array that contains a copy of the rest of the array, and pass that instead to the next level of recursion. This is of course very inefficient in Java, and to be avoided at all costs. In programming languages which provide efficient **array slicing**, this would be the way to go.
2. We can maintain a *left* index, along with a reference to the array, that is used to indicate where the beginning of the array is. Initially, `left = 0`, meaning that the array begins at the expected index of 0. Eventually, a value of `l == a.length - 1` means that the rest of the array is simply the last element, and then `l == a.length` means that it's an 0-sized array. For Java, this is by far the **preferred** method.

```
                  -
                  | false                    if l ≥ a.length
                  | true                     if a[l] = k
contains(a,l,k) = |
                  | return contains(a,l+1,k) otherwise
                  -


static boolean contains(int[] a, int l, int k) {
    if (l ≥ a.length)
        return false;
    else if (a[l] == k)
```

```
        return true;
    else
        return contains(a, l + 1, k);
}
```

We start the search with **`contains(a, 0, key)`**, and then at each step, the rest of the array is given by advancing the left boundary `l`.

Instead of just a **yes/no** answer, what if we wanted the **position** of the key in the array? We can simply return **l** instead of **true** as the position if found, or use a sentinel **-1** instead of **false** if not.

Binary search in a sorted array

Given the abysmal performance of sequential search, we obviously want to use binary search whenever possible. Of course, the pre-conditions must be met first:

1. The sequence must support random access (an array that is)
2. The data must be sorted

Now we can formulate a recursive version of binary search of a `key` in an array `data` between the positions `l` and `r` (inclusive):

```
if the array is empty (if l > r that is):
  return false
else:
  Find the position of the middle element: mid = (l + r)/2
  If key == data[mid], then return true
  If key > data[mid], the search the right half data[mid+1..r]
  If key < data[mid], the search the left half data[l..mid-1]
end if
```

Now we can write the recursive definition, and translate that to Java.

```
                    -
                    | false                  if l > r
                    | true                   if k = a[mid]
contains(a,l,r,k) = |
                    | contains(a,mid+1,r,k)  if k > a[mid]
                    | contains(a,l,mid-1,k)  if k < a[mid]
                    -


static boolean contains(int[] a, int l, int r, int k) {
    if (l > r)
        return false;
    else {
        int mid = (l + r)/2;
        if (k == a[mid])
            return true;
        else if (k > a[mid])
            return contains(a, mid + 1, r, k);
        else
            return contains(a, l, mid - 1, k);
```

```
    }
}
```

We start the search with **`contains(a, 0, a.length - 1, key)`**, and then at each step, the rest of the array is given by one half of the array – left or right, depending on the comparison of the key with the middle element.

Instead of just a **yes/no** answer, what if we wanted the **position** of the key in the array? We can simply return **mid** as the position instead of **true** if found, or use a sentinel **-1** instead of **false** if not.

Finding the maximum in a sequence (linear version)

Given a *sequence* of keys, our task is to find the **maximum key** in the sequence. This is of course trivially done iteratively (for a non-empty sequence): take the 1st one as maximum, and then iterate from the 2nd to the end, exchanging the current with the maximum if the current is larger than the maximum.

Formulating this recursively: the maximum key in a sequence is the larger of the following two:

1. the 1st key in the sequence
2. the **maximum** key in the **rest** of the sequence

Once we have (recursively) computed the maximum key in the rest of the sequence, we just have compare the 1st key with that, and we have our answer! The base is also trivial (for a non-empty sequence): the maximum key in a single-element sequence is the element itself.

Since the **rest of the sequence** does not need random access, we can easily do this for a linked list or an array. Let's write it for a list first.

```
                _
                | list.item                            if list.next = null
findMax(list) = |
                | max(list.item, findMax(list.next)) otherwise
                 -

static int findMax(Node l) {
    if (l.next == null)
        return l.item;
    else {
        int maxRest = findMax(l.next);
        return Math.max(l.item, maxRest);
    }
}
```

We find the maximum with **`findMax(head)`** (where `head` is the reference to the first node of the list), and then at each step, the rest of the array is given by advancing the head reference `head`.

What if the sequence is an array? Well, then we use the same technique we've used before — use a left (and optionally right) boundary to *window* into the array.

```
                _
                | a[l]                          if l =  r
```

```
                  | a[l]                          if l = r
findMax(a,l,r) = |
                  | max(a[l],findMax(a,l+1,r)) otherwise
                  -


static int findMax(int[] a, int l, int r) {
    if (l = r)
        return a[l];
    else {
        int maxRest = findMax(a, l + 1, r);
        return Math.max(a[l], maxRest);
    }
}
```

We find the maximum with **`findMax(a, 0, key)`**, and then at each step, the rest of the array is given by advancing the left boundary `l`.

Finding the maximum in an array (binary version)

If our sequence is an array, we can also find the maximum by formulating the following recursive definition: the maximum key in an array is the larger of the following two:

1. the maximum key in the **left half** of the array
2. the maximum key in the **right half** of the array

This will not produce an efficient version for linked lists, for the same reason binary search is not efficient for linked lists.

```
                  -
                  | a[l]                          if l =  r
findMax(a,l,r) = |
                  | max(findMax(a,l,mid),
                  |     findMax(a,mid+1,r)) otherwise
                  -


static int findMax(int[] a, int l, int r) {
    if (l = r)
        return a[l];
    else {
        int mid = (l + r)/2;
        int maxLeftHalf = findMax(a, l, mid);
        int maxRightHalf = findMax(a, mid + 1, r);
        return Math.max(maxLeftHalf, maxRightHalf);
    }
}
```

We find the maximum with **`findMax(a, 0, key)`**, and then at each step, the array is divided into two halves. If you draw the recursion tree, you will see why I refer to it as the *binary version*.

Selection sort

How about sorting a sequence recursively? We know from our iterative discussion of selection sort that it does not require random access, so we'll look at recursive versions for both linked lists and arrays.

does not require random access, so we'll look at recursive versions for both linked lists and arrays.

The basic idea behind selection sort is the following: put the 1st minimum in the 1st position, the 2nd minimum in the 2nd position, the 3rd minimum in the 3rd position, and so on until each key is placed in its position according to its *rank*. To come up with a recursive formulation, the following observation is the key:

> Once the 1st minimum in the 1st position, it will never change its position. Now all we have to do is to sort the **rest of the sequence** (from 2nd position onwards), and we'll have a sorted sequence.

Now we can write the recursive definition for a linked list, and translate it to Java.

```
                    -
                    | done                 if list = null or list.next = null
selectSort(list) = |
                    | exchange the minimum key with list.item,
                    |    and sort the rest of the list
                    |    with selectSort(list.next)
                    -

static void selectSort(Node l) {
    if (l == null || l.next == null)
        return;
    else {
        Node minNode = findMinNode(l);
        swap(l, minNode);
        selectSort(l.next);
    }
}
```

We sort a list headed by `head` by calling `selectSort(head)`. Note that we're not find the minimum key, but rather the node that **contains** the minimum key since we need to exchange the left key with the minimum one. We can write that iterative of course, but a recursive one is simply more fun. This is of course almost identical to finding the maximum in a sequence, with two differences: we're find the minimum, and we're return the node that contains the minimum, not the actual minimum key.

```
                     -
                     | list.item                    if list.next = null
findMinNode(list) = |
                     | get the node that contains the minimum in
                     |   the rest with findMinNode(list.next)), and
                     |   return either list.item or the other one
                     |   depending on which one has the smaller key
                     -

static Node findMinNode(Node l) {
    if (l.next == null)
        return l;
    else {
        Node minNode = l;
        Node minNodeRest = findMinNode(l.next);
        if (minNodeRest.item < l.item)
            minNode = minNodeRest;
```

```
        return minNode;
    }
}
```

If the sequence is an array, then we have to use the left (and optionally right) boundary to window into the array.

```
                    -
                    | done                 if l >= r
selectSort(a,l,r) = |
                    | exchange the minimum key with a[l],
                    |    and sort the rest of the array
                    |    with selectSort(a,l+1,r)
                    -

static void selectSort(int[] a, int l, int r) {
    if (l >= r)
        return;
    else {
        Node minIndex = findMinIndex(a, l, r);
        swap(a, l, minIndex);
        selectSort(a, l + 1, r);
    }
}
```

We sort an array `a` by calling `selectSort(a, 0, a.length - 1)`. And we can write `findMinIndex` recursively of course, which is again almost exactly the same as either finding the maximum in a sequence, or finding the maximum in an array — we'll use the binary method just for illustration.

```
                      -
                      | a[l]                    if l = r
findMinIndex(a,l,r) = |
                      | get the index of minimum in the rest with
                      |   findMinIndex(a,l+1,r)), and return either
                      |   l or the other one depending on which one
                      |   has the smaller key
                      -

static Node findMinIndex(int[] a, int l, int r) {
    if (l == r)
        return l;
    else {
        int mid = (l + r)/2;
        Node minIndexLeft = findMinIndex(a, l, mid);
        Node minIndexRight = findMinIndex(a, mid + 1, r);
        int minIndex = minIndexLeft;
        if (a[minIndexRight] < a[minIndexLeft])
            minIndex = minIndexRight;

        return minIndex;
    }
}
```

Insertion sort

Insertion works by inserting each new key in a sorted array so that it is placed in its rightful position, which now extends the sorted array by the new key. In the beginning, there is a single key in the array, which by definition is sorted. Then the second key arrives, which is then inserted into the already sorted array (of one key at this point), and now the sorted array has two keys. Then the third key arrives, which is inserted into the already sorted array, creating a sorted array of 3 keys. And so on. Iteratively, it's a fairly simple operation. The question is how can we formulate this recursively. The following observation is the key to this recursive formulation:

```
Given an array of n keys, sort the first n-1 keys and then
insert the n^th key in the sorted array such that all
n keys are now sorted.
```

Note how the recursive part comes first, and then the n^{th} key is inserted (iteratively) into the sorted array. Unlike recursive selection sort, we're going from **right to left** in the recursive version of insertion sort.

Note that we don't need random access, but need to be able to iterate in both directions (reverse direction to insert the new key in the sorted partition). So, if we're sorting a linked list using the insertion sort algorithm, the list must be doubly-linked.

If sorting an array, we can formulate the solution as follows:

```
                    -
                    | done                if l >= r
insertSort(a,l,r) = |
                    | recursively sort the first n-1 keys using
                    |    insertSort(a,l,r-1), and then insert the
                    |    n^th key (index r in
                    |    case) such that the result is sorted.
                    -

static void insertSort(int[] a, int l, int r) {
    if (l >= r)
        return;
    else {
        insertSort(a, l, r - 1);

        // Now insert the r'th key by moving leftwards
        Object key = a[r];      // save to avoid being overwritten
        int j = r - 1;

        // Shift left, making room for the key
        while (j >= 0 && key < data[j]) {
            data[j + 1] = data[j];
            j--;
        }
        // Now insert the key in the right position
        data[j + 1] = key;
    }
}
```

We sort an array `a` by calling `insertSort(a, 0, a.length - 1)`.

Sum of integers 1 through N

Computing the sum of the 1st N integers has a trivial iterative solution. It is however instructive to see the recursive formulation. This is an example of *functional recursion*.

The sum of the 1st `N` integers is `N` **plus** the sum of the 1st `N-1` integers. The base case is when we sum only the first integer, namely when `N = 1`.

```
      -                                static int sumN(int n) {
      | 1               n = 1              if (n == 1)
sum = |                                        return 1;
      | n + sum(n - 1)  n > 1              else
      -                                        return n + sumN(n - 1);
                                       }
```

Exponentiation – a^n

This is another example of *functional recursion*. To compute a^n, we can iteratively multiply `a` `n` times, and that's that. Thinking recursively, $a^n = a \times a^{n-1}$, and $a^{n-1} = a \times a^{n-2}$, and so on. The recursion stops when the exponent $n = 0$, since by definition $a^0 = 1$.

```
     -                                 static int exp(int a, int n) {
     | 1           n = 0                   if (n == 0)
a^n = |                                        return 1;
     | a × a n-1   n > 0                   else
-                                              return a * exp(a, n - 1);
                                       }
```

As it turns out, there is actually a much more efficient recursive formulation for the exponentiation of a number. We start by noting that $2^8 = 2^4 \times 2^4$, and that $2^7 = 2^3 \times 2^3 \times 2$. We can generalize that with the following recursive definition, and its implementation.

```
     -
     | 1                          n = 0
a^n = |
     | a^(n/2) × a^(n/2)            n is even
     | a^((n-1)/2) × a^((n-1)/2) × a  n is odd
     -


static int exp(int a, int n) {
    if (n == 0)
        return 1;
    else if (n % 2 == 0)
        return exp(a, n/2) * exp(a, n/2);
    else
        return exp(a, (n - 1)/2) * exp(a, (n - 1)/2) * a;
}
```

Ok, but why would we care about this formulation over the more familiar one? If we solve for the running time, both solutions take the same time, so what is the benefit of this approach? Notice how we're computing the following expressions **twice**:

1. `exp(a, n/2)`
2. `exp(a, (n - 1)/2)`

Why not compute it once, and then use the **result** twice (or as many times as needed)? We can, and as we'll find out, that'll give us a huge boost when we compute the running time of this algorithm. Here's the modified version.

```
static int exp(int a, int n) {
    if (n == 0)
        return 1;
    else if (n % 2 == 0) {
        int tmp = exp(a, n/2);
        return tmp * tmp;
    } else {
        int tmp = exp(a, (n - 1)/2);
        return tmp * tmp * a;
    }
}
```

All we're doing is removing the **redundancy** in computations by saving the **intermediate** results in temporary variables. This is a simple case of a technique known as **Memoization**, which is the next topic we study. Remember that we've already seen such redundancy in recursive computation — when computing the Fibonacci numbers.

Back to Table of contents

Issues/problems to watch out for

Inefficient recursion

The recursive solution for Fibonacci numbers outlined in these notes shows massive **redundancy**, leading to very inefficient computation. The 1st recursive solution for exponentiation also shows how redundancy shows up in recursive programs. There are ways to avoid computing the same value more than once by *caching* the intermediate results, either using **Memoization** (a *top-down technique* — see next lecture), or **Dynamic Programming** (a *bottom-up technique* — survive this semester to enjoy it in the next one).

Space for activation frames

Each recursive method call requires that it's **activation record** be put on the system *call stack*. As the depth of recursion gets larger and larger, it puts pressure on the system stack, and the stack may potentially run out of space.

Infinite recursion

Ever forgot a base case? Or miss one of the base cases? You end up with infinite recursion, since there nothing stopping your recursion! Whenever you see a **Stack Overflow** error, check your recursion!

Back to Table of contents

Discussion

Back to Table of contents

Memoization

Table of contents

Introduction

To develop a recursive algorithm or solution, we first have to define the problem (and the recursive definition, often the hard part), and then implement it (often the easy part). This is called the **top-down** solution, since the recursion open up from the top until it reaches the base case(s) at the bottom.

Once we have a recursive algorithm, the next step is to see if there are **redundancies** in the computation — that is, if the same values are being computed multiple times. If so, we can benefit from **memoizing** the recursion. And in that case, we can create a memoized version and see what savings we get in terms of the running time.

Recursion has certain overhead cost, that may be minimized by transforming the memoized recursion into an **iterative** solution. And, finally, we see if there are further improvements that we can make to improve the time and space complexity.

The steps are as follows:

1. write down the recursion,
2. implement the recursive solution,
3. memoize it,
4. transform into an iterative solution, and finally
5. make further improvements.

Back to Table of contents

Example using the Fibonacci sequence

To see this in action, let's take Fibonacci numbers as an example. Fortunately for us, the mathematicians have already defined the problem for us – the Fibonacci numbers are $\langle\ 0, 1, 1, 2, 3, 5, 8, 13, \ldots\ \rangle$ (some define it without the leading 0, which is ok too). Each number, except for the first two, is nothing but the sum of the previous two numbers. The first two are by definition 0 and 1. These two facts give us the recursive definition to compute the n^{th} Fibonacci number for some $n \geq 0$. This process, by the way, can be quite complex (and exciting, as you'll see next semester in Algorithms), and that's why we've chosen this easy example.

Let's go through the 5 steps below.

Step 1: Write or formulate the recursive definition of the n^{th} Fibonacci number (defined only for n

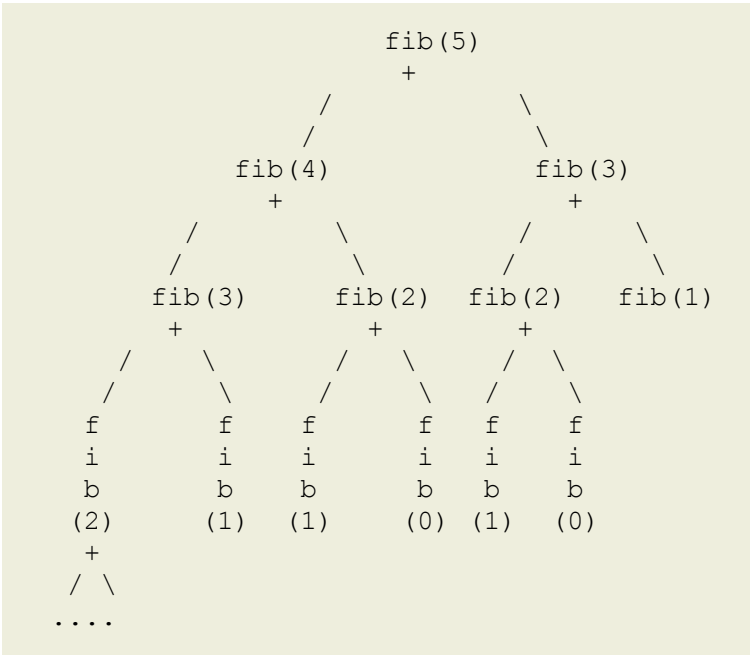
≥ 0).

```
          -
          | n                          if n < 2
fib(n) = |
          | fib(n-1) + fib(n-2)        n ≥ 2
          -
```

Step 2: Write the recursive implementation. This usually follows directly from the recursive definition.

```
/**
 * Returns the n'th Fibonacci number.
 * @return the n'th Fibonacci number.
 * @throws IllegalArgumentException if n < 0.
 */
static int fib(int n) {
    if (n < 2)                  // handle base conditions.
        return n;
    else                        // handle recursive cases.
        return fib(n - 1) + fib(n - 2);
}
```

Step 3: Memoize the recursion
Would this recursion benefit from memoization? Well, let's see by "unrolling" the recursion `fib(5)` a few levels:



Now you should notice something very interesting — we're computing the same fibonacci number quite a few times. We're computing `fib(2)` **3** times and `fib(3)` **2** times. Is there any reason why we couldn't simply save the result after computing it the first time, and re-using it each time it's needed afterward? This is the primary motivation for memoization – to avoid re-computing overlapping subproblems. In this example, `fib(2)` and `fib(3)` are **overlapping subproblems** that occur independently in different contexts.

Memoization certainly looks like a good candidate for this particular recursion, so we'll go ahead and memoize it. Of course, the first question is how we save the results of these overlapping subproblems, that we want to re-use later on.

The basic idea is very simple — before computing the i^{th} fibonacci number, first check if that has already been solved; if so, just look up the answer and return; if not, compute it and save it before returning the value. We can modify our `fib` method accordingly (using some pseudocode for now). Since fibonacci is a function of `1` integer parameter, the easiest is to use a `1`-dimensional array or table with a capacity of `n + 1` (we need to store `fib(0) ... fib(n)`, which requires `n + 1` slots) to store the intermediate results.

What is the cost of memoizing a recursion? It's the space needed to store the intermediate results. Unless there are overlapping subproblems, memoizing a recursion will not buy you anything at all, and in fact, cost you more space for nothing!

Remember this — memoization trades space for time.

Let's try out our first memoized version. (`M_fib` below stands for "memoized fibonacci").

```
static int M_fib(int n) {
    // Assume that we have some "global" array (also called a table)
    // F with n+1 capacity. Why can F not be a local variable?
    if (n < 2)
        return n;
    else {
        // Compute (and save) if it's not already computed.
        if (F[n] is empty)                // <<<< NOTE PSEUDOCODE!
            F[n] = M_fib(n - 1) + M_fib(n - 2);

        // Now just return the computed (and saved) value.
        return F[n];
    }
}
```

This is all the we need to avoid redundant computations of overlapping subproblems. The first time `fib(3)` is called, it'll compute the value and save the answer in `F[3]`, and then subsequent calls would simply return `F[3]` without doing any work at all!

There are a few details we've left out at this point:

1. Where would we create this "table" to store the results?
2. How can we initialize each element of `F` to indicate that the value has not been computed/saved yet? (Note the "is empty" in the code above).

Let's take these one at a time.

1. The "fib" method is a function of a single parameter — n, so if we wanted to save the intermediate results, all we need is an array that goes from `0 ... n` (i.e., of `n + 1` capacity). For more complex problems, we'll need much more sophisticated containers (the parameter may not be simple integers, which can be used to index into an array for example). Again, we've picked an easy example, but the basic process is the same for any other problem.

   Since the local variables within a method are created afresh each time the method is called, `F` cannot

be a local array. We can either use an instance variable within an object, or create an array in the caller of fib(4), and then pass the array to fib (in which case we'll have to modify fib to have another parameter of `type int[]`).

2. We need to use a sentinel which will indicate that the value has not been computed. Since it's array of `int`s, we can't use `null` (which is the sentinel used to indicate the absence of an object). However, we know that the factorial of any number is a non-negative integer, so we can use any negative number as the sentinel. Let's choose `-1`.

So, let's have an array `F` of `n+1` capacity that holds all the values of the intermediate results we need to compute the n^{th} Fibonacci number. We can have a **wrapper** method, and which creates this array or table, initializes the table and passes it onto `M_fib` as a parameter.

First, the wrapper method called `fib`, which basically sets up the table for `M_fib`, and calls it on the user's behalf.

```
static int fib(int n) {
    // Create and initialize the table.
    int[] F = new int[n + 1];
    for (int i = 0; i <= n; i++)
        F[i] = -1;                          // No solution saved in F[i] yet.

    // Now we can call M-Fib with this extra parameter "F".
    return M_fib(n, F);
}

static int M_fib(int n, int[] F) {
    // The table "F" is being passed as a parameter. The alternative is
    // to use a global variable.
    if (n < 2)
        return n;
    else {
        // Compute (and save) if it's not already computed.
        if (F[n] == -1)
            F[n] = M_fib(n - 1) + M_fib(n - 2);

        // Now just return the computed (and saved) value.
        return F[n];
    }
}
```

To compute the 5^{th} fibonacci number, we simply call `fib(5)`, which in turn calls `M_fib(5, F)` to compute and return the value.

Now that we have a memoized fibonacci, the next question is to see if we can improve the space overhead of memoization.

Step 4: Convert the recursion to iteration – the bottom-up solution.

To compute the 5^{th} fibonacci number, we wait for 4^{th} and 3^{th}, which in turn wait for 2^{nd}, and so on until the base cases of `n = 0` and `n = 1` return the values which move up the recursion stack. Other than the `n = 0` and `n = 1` base cases, the first value that is actually computed and saved is `n = 2`, and then `n = 3`, and then `n = 4` and finally `n = 5`. Then why not simply compute the solutions for `n = 2, 3, 4, 5` by iterating (using the base cases of course), and fill in the table from left to right? This is called the

bottom-up solution since the recursion tree starts at the bottom (the base cases) and works its way up to the top of the tree (the initial call). The bottom-up technique is more popularly known as **dynamic programming**, a topic that we will spend quite a bit of time on next semester!

```
static int fib(int n) {
    int[] F = new int[n + 1];   // The table to store computed values.

    F[0] = 0;                   // The base case for n = 0.
    F[1] = 1;                   // The base case for n = 1.

    for (int i = 2; i <= n; i++) {
        F[i] = F[i - 1] + F[i - 2];
    }

    return F[n];
}
```

You should convince yourself that this is indeed a solution to the problem, only using iteration instead of memoized recursion. Also, that it solves each subproblem (e.g., `fib(3)` and `fib(2)`) **exactly once**, and **re-uses the saved answer**.

This one avoids the overhead of recursion by using iteration, so tends to run much faster.

Can we improve this any further?

Step 5: improving the space-requirement in the bottom-up version

The n^{th} Fibonacci number depends only on the `(n - 1)` and `(n - 2)` Fibonacci numbers. However, we are storing ALL the intermediate results from `2 ... n - 1` Fibonacci numbers before computing the n^{th} one. What if we simply store the last two? In that case, instead of having a `n + 1` array, we need just **two** instance variables (or an array with `2` elements). Here's what the answer may look like.

```
static int fib(int n) {
    int f_2 = 0;                // The n-2 value.
    int f_1 = 1;                // The n-1 value.

    // The result - initialize to n (why? So that it works when n is 0 or 1).
    int f = n;

    for (int i = 2; i <= n; i++) {
        f = f_1 + f_2;
        // Now update the f_1 and f_2 for the next iteration (if any).
        f_2 = f_1;
        f_1 = f;
    }
    return f;
}
```

Back to Table of contents

Summary

We have just gone through the 5 steps of taking a recursive problem, writing a recursive solution, memoizing that recursion, transforming that into an iterative solution and then squeezing some more performance out of it. There are many questions left to be asked of course:

- Given a problem, how do we develop a recursive solution for it?
- What is the performance of the recursive solution?
- Given a recursive solution, would it benefit from memoization? In other words, are there overlapping subproblems?
- How would you transform a recursive solution into an iterative one?

Back to Table of contents

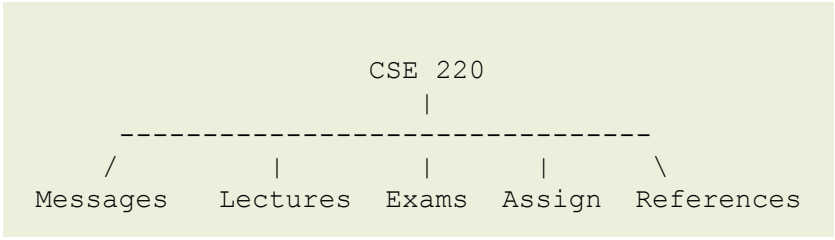
Trees

Table of contents

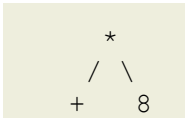
Introduction

Before defining what a *tree* is, let's take a look at examples of different types of trees we use everyday.

- Files and directories on a computer (starting from '/' on Unix-like systems, or "C:\" on Windows); structure of the CSE 220 website.



- positions in an organizations (starting from say the managing director or the CEO).
- **Arithmetic expression trees** — remember how we used postfix expression to implement a simple stack-based calculator? We can most naturally convert a prefix expression to a tree, and implement a calculator. For example, the infix expression `(5 + 3) * 8` is `* + 5 3 8` in prefix form, which can be represented as the following (expression) tree.



```
  / \
 5   3
```

- the structure of the contents in a book - the book has chapters, chapters have sections, and so on.
- specialized trees - binary search trees, heap ordered trees, and many many others.

A **tree** is made up of **nodes**, with **edges** or **links** connecting the nodes. Defined recursively:

```
A tree is either an empty tree, or a node connected to a set of
subtrees.
```

Back to Table of contents

Types of trees

The textbook discusses the following types of trees:

Free trees

A free tree (or just "a tree") is a set of nodes/vertices, with links/edges connecting pairs of distinct nodes. The defining property is that each pair of nodes must have exactly one simple path (simple path means that the nodes in a path are distinct, except perhaps the first and the last). The other way to say this is that each node is connected to every other node, and there are no cycles. This is the most general type of trees.

If there are none or multiple paths (cycles), then it's a **graph**! **Forests** are disjoint set of trees.

Rooted trees

A Rooted tree has a **distinguished node** that acts as the **root** (much like the **head** of a linked list), where it all starts. There is exactly one simple path between the root and every other node in the tree ("connected"), and there is no direction implied in these paths. Some definitions/concepts:

- **above**, **below**, **ancestor** and **descendant**: a node y is considered "below" node x if there is a path from root to y that touches x, and x is "above" y consequently. Y is a descendant of x (y below x) or x is an ancestor of y (x above x).
- **sibling**, **parent**, **children**, **grandparent**, **grandchildren**: the sibling nodes share the same parent. The immediate ancestor is called the parent, and immediate descendants children; grandparent and grandchildren are one step away in either direction. A node may have many children and many siblings, but only ONE parent.
- **leaves/terminal** nodes, **non-terminal** nodes: nodes with no children are called leaves or terminal nodes (because it's the end of the road for the links!). Similarly, nodes with at least one child are called non-terminal nodes. In trees representing recursion, the recursion is represented by the non-terminal nodes, and the non-recursive part by the terminal nodes.

Ordered trees

An ordered trees is a rooted tree in which the children of a node are ordered in a particular way. For example, the chapters in a book (ordered from earlier to later), the children in a family tree (may be ordered according to age), the order of the arguments to matrix multiplication (in expression trees), etc.

M-ary trees

An M-ary tree is an ordered tree in which each node must have **exactly** M children, which are also M-ary trees. For these trees, we use special nodes called **external** nodes (essentially *sentinel* nodes), which have no children. These external nodes representing leaves are essentially dummy nodes to satisfy the M-ary property. The **internal** nodes are those that have at least one child that is not "external". The "leaf" in an M-ary tree is then an internal node with M children all of which are external.

Binary trees

A binary tree is the simplest (and most widely used) M-ary tree with M=2. We'll spend lots of time on binary trees.

Looking at the "is-a" hierarchy -

```
Binary/M-ary tree -> ordered tree -> rooted tree -> (free) tree
```

Back to Table of contents

Recursive definitions and representations

Binary trees

Let's start with the recursive definition of a **binary tree**.

```
A binary is either: an external node (ie., an empty tree)
                or: or a an internal node connected to an ordered pair of
                    binary trees, called the left and right subtrees.
```

Given this recursive definition of a binary tree, the question now is how to represent it. One obvious way to represent the tree structure in a node is to use references to the children subtrees, and using a "sentinel" to represent an external node or the empty tree. In Java for example, we can use the null as the sentinel for an external. When using linked representation, we can maintain two different references - one to the left subtree and one to the right subtree. We can of course also use an array of two references, with the 1st

element referring to the left and the 2nd element referring to the right subtree respectively. The first choice seems more natural.

```
public class Node {
    Object item;              // the key within the node
    Node left;                // reference to the left child
    Node right;               // reference to the right child

    public Node(Object i, Node l, Node r) {
        item = i; left = l; right = r;
    }
}
```

With this representation, one can move to the left subtree with a reference assignment such as `n = n.left`, or to the right subtree with `n = n.right`.

We may also need to move "up" the tree, for which it may be convenient to also maintain a **parent** reference in each node. The root node does not have a parent, so we'll use the null as a sentinel. We'll see a need for that later on when we discuss **binary search trees**. Adding the **parent** reference to the `Node` class above, we get the following:

```
public class Node {
    Object item;              // the key within the node
    Node left;                // reference to the left child
    Node right;               // reference to the right child
    Node parent;              // reference to the parent node

    public Node(Object i, Node l, Node r, Node p) {
        item = i; left = l; right = r, parent = p;
    }
}
```

This is the so-called **linked** representation of a binary tree, where we use references to nodes as the links. There is also an **array** representation of trees, in which each node is represented by a specific index into an array. The idea is very simple: the root node is kept at index 1, and it's left and right children in indices 2 and 3. In general, any node at index k have children at indices 2k and 2k+1, and parent at floor(k/2). If 2k+1 is > N, where N is the number of internal nodes, then the node does not have a right child; if 2k > N, then it does not have a left child either. There is potentially a lot of wasted space unless the tree is "almost complete" - a binary tree that has all the levels filled, except possibly for the last level which is filled from the left. Much more on "almost complete" binary trees and array representation when we study "heap ordered" trees later this semester.

M-ary trees

The recursive definition of a **M-ary** tree is almost the same as the binary tree (remember that a binary tree is just an M-ary tree with M=2):

```
An M-ary tree is either: an external node (empty tree)
                     or: a node connected to an ordered sequence of M
                         trees, all of which are M-ary trees.
```

The representation is typically linked, using either explicit references to the children (left, middle and right for a ternary tree), or using an array of references for the children. If M > 3, it's probably easier to use an array.

```
public class Node {
    Object item;              // the key within the node
    Node[] children;          // references to the left-to-right children

    public Node(Object i, Node[] c) {
        item = i;
        children = new Node[c.length];
        System.arraycopy(c, 0, children, 0, c.length);
    }
}
```

Ordered trees

A node (called the root) connected to an ordered sequence of disjoint trees. The difference between this and an M-ary tree is that a rooted tree can have nodes with any number of children. Since the number of children varies from node to node, it may be better to use a linked list to represent the children in a particular node.

Rooted trees

Same as an ordered rooted tree, just that there no ordering in the children, so there are many representations of the same tree (eg., permuting the order of the children does not change the tree).

Free trees

Since free trees do not have a distinguished node that acts as the root, it is typically best to represent this type of tree using a graph representation. We'll see more about this when we study graphs.

Back to Table of contents

Graphs

Since we're talking about more and more general types of trees, good to introduce graphs. We've already seen graphs when discussing the Internet (or the FriendsNet assignment, which looks at friendships among people).

A graph is a set of nodes or vertices, with a set of edges that connect pairs of distinct nodes (with at most one edge connecting a pair of nodes).

A tree is nothing more than a graph with specific properties, so each tree is a graph. Is every graph a tree? Before answering that, let's define a few terms that we'll be using everywhere from now on.

path
: a path is sequence of edges starting from a source vertex leading to a target vertex.

simple path
: a path is *simple* if no vertex appears more than once in the path. In a tree, every path is simple.

cycle
: a cycle is a simple path that has the same first and last vertex. A tree cannot have a cycle by definition, since there must be a single simple path between every pair of nodes.

connected-ness
: if there is a path between every pair of vertices, then it is connected. A tree is connected by definition.

A graph G with N vertices is a tree if any of the following are satisfied (in other words, the following are all equivalent):

1. There are N-1 edges and no cycles.
2. There are N-1 edges and G is connected.
3. Exactly one path connecting each pair of vertices.
4. G is connected, but does not remain connected if any edge is removed.

Back to Table of contents

Mathematical properties of binary trees

1. A binary tree with N internal nodes contains N+1 external nodes.

   Proof: We prove this by induction. For the case of an empty tree, there is 1 external node, so this holds for N = 0. For N > 0, a binary tree with N internal nodes has a left subtree with k internal nodes, and a right subtree with N-1-k internal nodes, for some k between 0 and N-1 (the root is an internal node). By the inductive hypothesis, the left subtree has k+1 external nodes and the right subtree has N-1-k+1 = N-k external nodes, resulting in a total of k+1+N-k=N+1 external nodes.

2. A binary tree with N internal nodes has 2N internal links/edges, N-1 to internal nodes and N+1 to external nodes.

   Proof: Each internal node, except for the root, contributes one edge to its parent, which means that there are N-1 links to internal nodes. The N+1 external nodes also contribute one edge each, so there are N+1 links to external nodes, bringing the total number of links to N-1+N+1 = 2N.

3. Levels in a tree: the level of a node in a tree is one higher than the level of its parent, with the root at level 0. For a rooted tree, you can easily picture this by having horizontal lines that partition the nodes such that nodes at the same level are in the same partition. The level is also called the "depth" of a node. It's not defined for an empty tree.

   For a node `n', it's level is defined recursively in terms of its parent:

```
            -
            | 0,                    if n is the root (parent(n) == null)
level(n) = |
            | 1 + level(parent(n)), otherwise
            -
```

4. height of a tree: The height of a tree is the maximum of the levels of the tree's nodes. This however leads to an inefficient implementation, since we have to compute the levels for all the nodes, and

take the maximum value.

Alternatively, the height of a tree is the maximum height of the root's subtrees plus 1 (for the link to the root itself). An empty tree has a height of -1 (which means that a tree with a single node, ie., just the root, has a height of 0).

For a tree rooted at `n', it's height is defined recursively in terms the heights of its subtrees:

```
             -
             | -1,                    if tree is empty (n == null)
height(n) = |
             | 1 + max(height(n.left), height(n.right)), otherwise
             -
```

Note that this guarantees that the height of a tree with a single node is 0.

5. The height of a binary tree with N internal nodes is at least lg N and at most N-1.

   Proof: The worst case (the maximum height) is when each node at all the levels except the last one has one internal and one external children. The leaf node at the last level has both external children. The height is obviously N-1 in this case.

   The best case (the minimum height) is when each node at all but perhaps the last level has exactly two internal children; the nodes at the last level may have a mix of internal and external children. Level 0 has 1 (or 2^0) node, level 1 has 2^1 nodes, level 2 has 2^2 nodes, and so on until level h-1 which has 2^{h-1} nodes. The last level, level h, may have just one internal node or up to 2^h internal nodes. Given that the tree has N+1 external nodes, we have the following:

```
    2^h < N + 1 <= 2^{h+1}

    h < lg(N+1) and h >= lg(N+1) - 1
so, h = floor(lg(N))
```

   Another way to look at it is if you consider the following question: what is the shortest possible tree with N internal nodes? The maximum number of internal nodes in a binary tree of height h (without increasing its height) occurs when all the leaves are completely full, and adding another node would increase the height to h+1. That indeed is the shortest tree possible! Instead of keeping the last level full, we can decrease the number of nodes so that there is a single node at the last level, still keeping the height to be h.

```
N_max = 2^0 + 2^1 + 2^2 + ... + 2^{h-1} + 2^h = 2^{h+1} - 1
N_min = 2^0 + 2^1 + 2^2 + ....+ 2^{h-1} + 1   = 2^h - 1 + 1 = 2^h

N <= N_max
N <= 2^{h+1} - 1
N+1 <= 2^{h+1}
lg(N+1) <= h+1
h >= lg(N+1) - 1

N >= N_min
N >= 2^h
lg(N) >= h
```

```
lg(N) >= h
h <= lg(N)

** Note: a quick review of sums - arithmetic, geometric, etc.
```

Back to Table of contents

Tree traversal

Traversal in trees is analogous to "iteration" in sequences, in which each node is "visited" exactly once. Unlike in a sequence, there is no "natural" order in which to proceed to the next element, but there are a few conventional way of doing it. The difference is the choice is advancing to the next node in the tree, so there are different possible "visitation" orders or linearizations.

Depth order: The depth order traversals recursively visit the children of a node until reaching an external nodes, and then backing up and visiting the next children, until all the children are exhausted. There are 3 common ones:

```
  pre-order traversal  : visit self, then left and then right (slr)
  in-order traversal   : visit left, then self, then right (lsr)
  post-order traversal : visit left, then right, and then self (lrs)
```

The pre-order traversal is commonly used when searching for a key within a node in a tree. The pre-order traversal is commonly known as **depth-first** when dealing with graphs.

The depth-order traversals are naturally recursive, and most easily implemented as such. However, these can also be implemented iteratively using stacks (which makes sense given the connection between recursion and stacks). The following shows a recursive implementation of pre-order traversal of a binary tree.

```
public static preOrderVisit(Node n) {
    // Done if tree is empty
    if (n == null)
        return;
    else {
        // Visit self, and then visit left and right subtrees recursively
        visit(n);
        preOrderVisit(n.left);
        preOrderVisit(n.right);
    }
}
```

We start traversing a tree by calling `preOrderVisit(root)` where `root` is a reference to the root node of the binary tree.

We can get the other two depth-order traversals – in-order and post-order – by simply re-arranging the orders of the method calls.

```
public static inOrderVisit(Node n) {
    // Done if tree is empty
    if (n == null)
```

```
        return;
    else {
        // Visit left subtree, visit self, and then visit right subtree
        preOrderVisit(n.left);
        visit(n);
        preOrderVisit(n.right);
    }
}

public static posOrderVisit(Node n) {
    // Done if tree is empty
    if (n == null)
        return;
    else {
        // Visit left subtree and right subtrees, and then visit self
        preOrderVisit(n.left);
        preOrderVisit(n.right);
        visit(n);
    }
}
```

Level order: Starting from level 0 (only one node - the root), visit the nodes in level 1 (perhaps in left-to-right order), and then level 2 and so on. Can be easily implemented (iteratively) using a queue. In fact, if you replace the stack used in (iterative) pre-order traversal with a queue, you'll get level order traversal. The level order is commonly known as **breadth-first** traversal when dealing with graphs.

Back to Table of contents

Some recursive tree algorithms/methods

Since tree is an inherently recursive data structure, the operations on a tree are often easily described by recursion as opposed to by iteration.

We've already seen some of the examples earlier (height, and levels or depth), so let's take a look a few more.

1. Counting the nodes: the number of nodes in a binary tree is the sum of the number of nodes in the two subtrees, plus one (for the root of the tree).

   For a tree rooted at `n', it's size (i.e., the number of nodes) is defined recursively in terms the sizes of its subtrees:

```
          -
          | 0,                     if tree is empty (n == null)
 size(n) = |
          | 1 + size(n.left) + size(n.right)          otherwise
          -

 public static int size(Node n) {
     if (n == null)
         return 0;
     else
         return 1 + size(n.left) + size(n.right);
 }
```

```
}
```

2. Copying a binary tree: To copy a tree, you can copy the root node, and then attach a copy of the left subtree to the copy's left and a copy of it's right subtree to the copy's right. Copying an empty tree returns an empty tree of course.

   For a tree rooted at `n', it's copy is defined recursively in terms the copies of its subtrees:

```
          -
          | null,                     if tree is empty (n == null)
copy(n) = |
          | c = make a copy of node `n'
          |  (i.e., create new node with `n''s element)
          | c.left = copy(n.left)
          | c.right = copy(n.right)   otherwise
          -

public static Node copy(Node n) {
    if (n == null)
        return null;
    else {
        Node copy = new Node(n.element, copy(n.left), copy(n.right));
        return copy;
    }
}
```

3. Comparing if two trees are the same (structure and values): If both trees are empty, they are the same; otherwise, if the two root's have equal elements, AND if the left and right subtrees are the same, then the trees are the same.

   For trees rooted at `n1' and `n2', their "same-ness" or equality is is defined recursively in terms the equality of the respective subtrees:

```
          -
          | true                      if both n1 and n2 are empty
          | false                     if n1 is empty, but n2 is not
same(n) = |
          | true                      if nodes n1 and n2 have equal
          |                           elements, AND if
          |                           same(n1.left, n2.left) AND
          |                           same(n1.right, n2.right)
          |
          | false                     otherwise
          -

public static boolean same(Node n1, Node n2) {
    if (n1 == null || n2 == null) {
        if (n1 == null && n2 == null)
            return true;
        else
            return false;
    } else {
        return n1.element.equals(n2.element)
               && same(n1.left, n2.left)
               && same(n1.right, n2.right);
    }
```

```
}
```

4. Finding the leftmost node of a binary tree: the leftmost node is the one you get by taking a left each time starting from the root until there is more left.

```
public static Node leftmost(Node n) {
    if (n.left == null)
        return n;
    else
        return leftmost(n.left);
}
```

You can get the rightmost by simply replacing "left" with "right" above.

Back to Table of contents

Discussion

Back to Table of contents

Binary Search Trees

Table of contents

Introduction

See Sedgewick sections 5.4-5.6 for general discussion on trees, and 5.7 for recursive binary-tree algorithms. In these notes, we're going to focus on binary search trees, which are covered in Sedgewick section 12.6.

Binary search trees are binary trees with the additional properties:

for any internal node:

- the key in the left child must be less than the node's key, and
- the key in the right child must be larger than the node's key;

and this must hold recursively.

A consequence of this "must hold recursively" is the following:

for any internal node:

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

It is an excellent container for a Dictionary or Map ADT, especially if the tree is somewhat "balanced" (we'll talk more about this later), supporting fast insert, lookup and remove. In addition, it also supports fast minimum, maximum, successor, and predecessor operations. Note that a dictionary implemented with sorted arrays provide at least as fast operations except for in the cases of insert and remove.

Our binary tree has a simple Node class:

```
public class Node {
    public Object element;
    public Node left;
    public Node right;
    public Node parent;

    public Node (Object e) {
        this(e, null, null, null);
    }
```

```
    public Node (Object e, Node l, Node r, Node p) {
        element = e;
        left = l;
        right = r;
        parent = p;
    }
}
```

Note that we maintain a `parent` reference in each node, which may be needed to *speed up* certain operations, such as finding the successor or the predecessor of a key in a binary search tree.

Back to Table of contents

Operations on binary search trees

Some simple operations:

1. The **smallest key** is the **leftmost node** starting from the root of the tree. If you have a tree rooted at reference "root", calling `findLeftMost(root)` will return a reference to the leftmost node, and the key within the node is the smallest key in the tree.

```
Iteratively:

/**
 * Returns the leftmost node of this subtree rooted at "n".
 * @param n the root of this subtree.
 * @return the leftmost node.
 */
static Node findLeftMost(Node n) {
    while (n.left != null)
        n = n.left;
    return n;
}

Recursively:

static Node findLeftMost(Node n) {
    if (n.left == null)
        return n;
    else
        return findLeftMost(n.left);
}
```

2. The **largest key** is in the **rightmost node** starting from root of the tree. Just swap "left" with "right" in `findLeftMost` and you're done.
3. The **successor** of a key in node in a binary search tree is the key that is immediately larger this this key. Since the successor must be larger, it must belong in the right subtree. Specifically, it must be the **smallest key in the right subtree**! So, a partial implementation is the following:

```
/*
 * Returns the successor node of the given node.
 * @param n the node whose successor is sought.
 * @return the successor node, null if given node contains the largest
```

```
 *         key.
 */
public static Node findSuccessor(Node n) {
    return findRightMost(n.right);
}
```

However, this ONLY works if the given node has a right subtree. What do we do in the case where there is no right subtree? In the case where a node does not have a right subtree, the successor must be "above" somewhere. We move up the tree, and the first parent/grandparent node where we "turn right" is the successor. If we move to the root's parent, we get null, which is the case for the largest key in the tree.

```
public static Node findSuccessor(Node n) {
    if (n.right != null) {
        // If n has a right subtree, then the successor node is the
        // leftmost node in the right subtree.
        Node p = findLeftMost(n.right);
        return p;
    } else {
        // If n does not have a right subtree, the successor node is
        // an ancestor node, but only if it's to the "right" of n. The
        // successor to the largest node (which is the rightmost node
        // in the tree) is the parent of root, which is null.
        Node q = n;
        Node p = q.parent;
        while (p != null && p.right == q) {
            q = p;
            p = p.parent;
        }
        return p;
    }
}
```

This works for all cases.

4. Finding the predecessor works the same way, except that you swap "left" for "right" in the successor case.
5. How do we find a key in a binary search tree? Easy, we start at root, looking for the key in a node - if found, we're done; otherwise, we move left or right depending on the key value.

Iteratively:

```
/**
 * Finds the node containing the given key in tree rooted at "n".
 * @param key the key to find in a tree node.
 * @param n the root of this subtree.
 * @return the node if found, or null otherwise.
 */
static Node findNode(Object key, Node n) {
    while (n != null) {
        if (key.equals(n.element))
            return n;
        else if (((Comparable)key).compareTo(n.element) > 0)
            n = n.right;
        else
            n = n.left;
```

```
    }
    return n;
}
```

Recursively:

```
/**
 * Finds the node containing the given key in tree rooted at "n".
 * @param key the key to find in a tree node.
 * @param n the root of this subtree.
 * @return the node if found, or null otherwise.
 */
static Node findNode(Object key, Node n) {
    if (n == null)
        return null;
    else if (key.equals(n.element))
        return n;
    else if (((Comparable)key).compareTo(n.element) > 0)
        return findNode(key, n.right);
    else
        return findNode(key, n.left);
}
```

6. How would you get the keys in sorted manner? Two ways - you can visit the nodes "in-order" starting at the root; or, you can start at the leftmost node, and iterate by visiting the successor at each step. The first way has a "natural" recursive implementation, while the second way has a "natural" iterative implementation.

   First way - recursively (iteratively is not easy at all - just try by using a stack to emulate recursion; see Sedgewick for details):

```
/**
 * Prints the keys in sorted order using in-order traversal.
 * @param n the root of this subtree.
 */
public static void printSorted(Node n) {
    if (n == null)
        return;
    else {
        printSorted(n.left);
        System.out.println(n.element);
        printSorted(n.right);
    }
}
```

   Second way - iteratively:

```
/**
 * Prints the keys in sorted order using iteration.
 * @param n the root of this subtree.
 */
public static void printSorted(Node n) {
    n = findLeftMost(n);
    while (n != null) {
        System.out.println(n.element);
        n = findSuccessor(n);
```

```
        n = findSuccessor(n);
    }
}
```

Second way also works recursively (but of course!), but needs a "helper" method (see the two methods below - the first one is the helper one):

```
/**
 * Prints the keys in sorted order using iteration.
 * @param n the root of this subtree.
 */
public static void printSorted(Node n) {
    n = findLeftMost(n);
    printSortedRec(n);
}

private static void printSortedRec(Node n) {
    if (n == null)
        return;
    else {
        System.out.println(n.element);
        printSortedRec(findSuccessor(n));
    }
}
```

7. Ok, now we need to start insert and remove operations. Let's start with insertion. Remember that an insertion into a binary search tree ALWAYS replaces an external node with a new internal node containing the key that is being inserted.

   First, create a new Node object with the key in it. Now, if the tree is empty, then the new node becomes the root! Simple.

   Otherwise, we use lock-step method to move down the tree to find the external node that we'll replace (keeping a parent reference as we go down), and then add the new node in the appropriate place.

```
/**
 * Inserts a key in this binary search tree rooted at "root". Let's
 * assume distinct keys only, so we'll throw an exception if the key
 * already exists.
 *
 * @param key the key to insert.
 * @param root the root of this tree.
 * @return reference to the root, which may have changed.
 * @exception DuplicateKeyException if the key already exists.
 */
public static Node insert(Object key, Node root) {
    // The tree may be empty, in which case we set root to the new
    // node containing the given element.
    if (root == null) {
        root = new Node(key);
    } else {
        // Otherwise, tree is not empty, so we find the *parent* node
        // to which we'll attach the new node as a child. We use two
        // references in lock-step until one reaches an external node.
        Node p = null; // the parent node.
```

```
        Node n = root;
        while (n != null) {
            int compare = ((Comparable) key).compareTo(n.element);
            if (compare == 0) {
                throw new DuplicateKeyException();
            } else if (compare > 0) {
                p = n;
                n = n.right;
            } else {
                p = n;
                n = n.left;
            }
        }

        // At this point, n points to the external node which will be
        // replaced by the new node, and p points to the parent of n.
        // So we just have to attach the new node to either left or
        // the right child of p, depending on the relative values of
        // p.element and element.

        Node newNode = new Node(key);
        if (((Comparable) key).compareTo(p.element) > 0)
            p.right = newNode;
        else
            p.left = newNode;

        // p will become newNode's parent.
        newNode.parent = p;
    }
    return root;
}
```

There is a **much easier** and **much more elegant** recursive algorithm, but takes a bit of getting used to. See Sedgewick Sec 12.6 for details.

How would we handle duplicate keys? We have to use the relations < and >= or <= and > for the left and right subtrees. How we handle it in the insert method depends on the problem on how to handle it. If we're using binary search tree to implement a Set ADT (remember that we used a sorted array to implement a Set earlier), we may choose to ignore the duplicates. In the case of a phone book application, there is a value associated with a key, so for a duplicate key, it may simply update the value. Like I said, depends on the problem.

8. Ah, now we look at removing a key. Removing a key involves finding the node that contains the key, and removing that node from the binary search tree, ensuring that the resulting tree remains a binary search tree.

   Sedgewick uses a beautiful recursive algorithm to do this, but that may be slightly tricky to understand. So I'll cover the iterative one as well.

   Let's look at the 3 cases we had discussed in class:

   1. Leaf node with no children - removing this is trivial, as we simply have to the set the appropriate child reference in the parent to null.
   2. Node with 1 child - also simple, as we add a "bypass" from its parent to its only child. Have to keep track of left and right, etc, but that can easily be done by comparing the keys.
   3. Node with both children - here's the tough one. Removing such a node with destroy the tree's

properties. There is one thing to note - since it has both its children, its predecessor and successor are both "below" it. The predecessor is the rightmost node in its left subtree, so it must not have more than 1 child; likewise, the successor has *at most* 1 child. Removing the predecessor or the successor then falls under cases 1 or 2, since it will have either 0 or 1 child, which we already know how to do. How does that help us here? We simply replace the key in the node to be removed with its successor (or predecessor), and then remove the successor (or predecessor). Basically, we're reducing the problem of removing a node with two children to the one of removing a node with 0 or 1 child.

9. Rotating a tree around a given node. I'll leave this one for you to study from the textbook (Sec. 12.8).

Back to Table of contents

Performance of binary search trees

Now the big question - what's the **maximum number of comparisons** we need to make to find a key in a binary search tree? Well, in the worst case, we start from the root, work our way down to either a leaf node (success) or an external node (failure). Both cases, the maximum number of comparisons is 1 more than the tree's height (remember that a tree with a single node has a height of 0). We know that the maximum and minimum heights of a binary tree:

```
    log_2(N+1) <= h <= N - 1

Which means that the maximum number of comparisons, equal to 1+h, can be
computed easily as well: if C is the number of comparisons:

    C <= h + 1                  (in the worst case, it's equal to h + 1)
       = N - 1 + 1 = N

    max # of comparisons = N

just like linear search! Not good at all. However, if the tree were nicely
balanced (see the text for definition), we have a "short" tree. If we now
use the minimum height, the number of comparisons is:

    C >= log_2(N+1) + 1
```

so at least approximately log_2(N) comparisons! Note that binary search (on sorted arrays) provides a guarantee of log_2(N) comparisons in the WORST case, and for a binary search tree there is actually a range from log_2(N) to h.

This should tell you that we really really want a balanced tree! But how? We'll do that next semester.

Back to Table of contents

Discussion

Back to Table of contents

Graphs

Table of contents

Introduction

A graph is a set of nodes or vertices, with a set of edges that connect pairs of distinct nodes (with at most one edge connecting a pair of nodes). Formally, graph `G` is a pair `(V,E)`, where `V` is a finite set (set of vertices) and `E` is a finite set of pairs from `V` (set of edges). We will often denote `n = |V|, m = |E|`.

Graphs are everywhere!

Transportation networks
: How should you design the highway network in a country? What is the quickest way to drive from Badda to Naraynganj?

Communication networks
: How to send a network packet from Bracu intranet to Yahoo mail server?

Information networks
: Is the World wide a directed or undirected network?

Social networks
: Facebook, Myspace, Flickr, ...

Dependency networks
: What courses must you take before you can take CSE-423?

Mazes
: Is there way out of a maze? If so, what is a path from entrance to exit?

Often a complex system can be reduced to a simple graph keeping only the essential information needed to solve the problem at hand. For example, the labyrinth or maze shown in Fig. 1 can be reduced to a much simpler graph that has all the information needed to answer questions about the maze. In the graph representation, a vertex represents a *location* in the maze, and an edge represents connection between two adjacent locations. We can describe this graph as:

```
G = (V,E)
  = ({A,B,C,D,E,F,G,H,I,J,K,L},
     {(A,B),(A,C),(A,D),(B,E),(B,F),(C,F),(D,G),(D,H),(E,I),(E,J),
      (G,H),(I,J),(K,L)})
```

Using the graph representation, we can ask a question typical of a maze: Is there a path from `A` to `J`? If so, what is the path, that is, what is a possible sequence of steps that we may take from `A` to get to `J`?

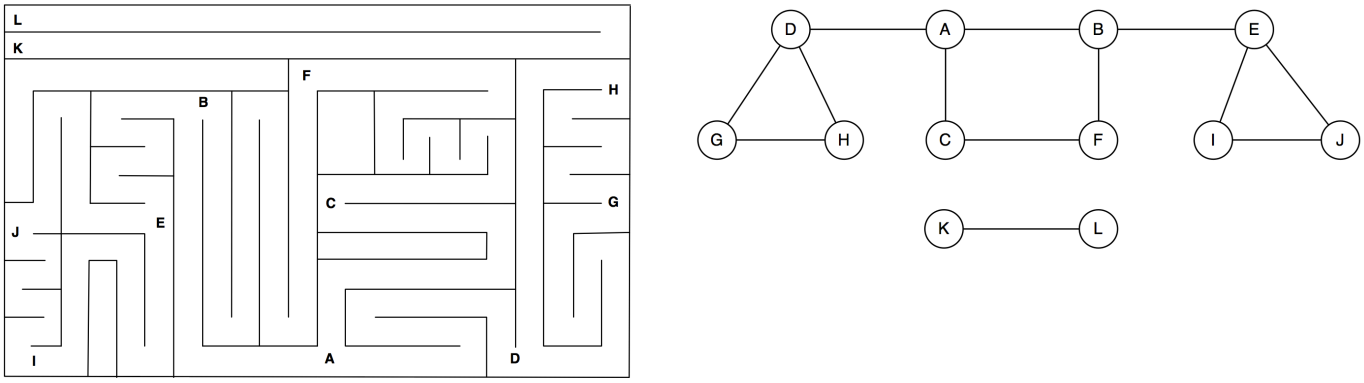


Figure 1: A maze (left) and it's graph representation (right)

Before we can start discussing how to answer such questions of a graph, let's first look at the different types of graphs, and the various terminology that we need to be familiar with.

Back to Table of contents

Graph terminology

Directed and undirected graphs

A graph is **directed** if `E` consists of **ordered** pairs. Conversely, a graph is **undirected** if `E` consists of **unordered** pairs. The graph in Fig. 1 is undirected since all edges $\in$ `E` are undirected. For example, the edge `(A,B)` $\equiv$ `(B,A)`, so is written out just once. Any undirected graph can be turned into a directed one by replacing each undirected edge `(x,y)` $\in$ `E` with two directed edges `(x,y)` and `(y,x)` (which doubles the number of edges obviously). Fig. 2 shows a directed (left) and an undirected (right) graph.

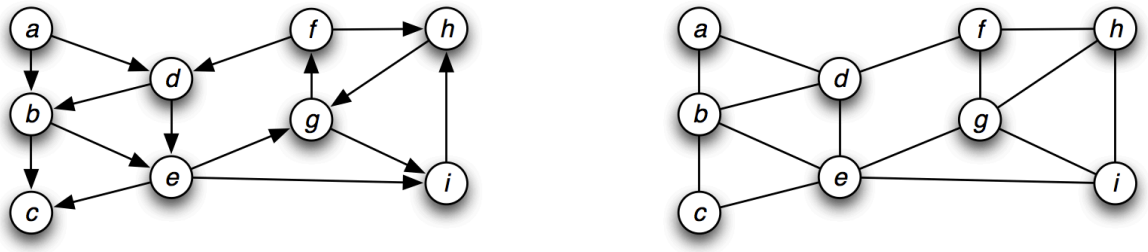


Figure 2: A directed (left) and an undirected graph (right)

If `(u,v)` $\in$ `E`, then vertices `u` and `v` are **adjacent**.

Directed Acyclic

If a directed graph has no cycles, then it's called *directed acyclic graph* or **dag** for short. Dags are a very important subclass of graphs, and we'll be seeing them in many different applications such as dependency graphs (such as the graph that describes your course pre-requisites!).

Weighted and unweighted graphs

We can assign weight function to the edges: `w(e)` is a weight of edge `e` $\in$ `E`. The graph which has such function assigned is called **weighted**. Fig. 3 shows a weighted graph.

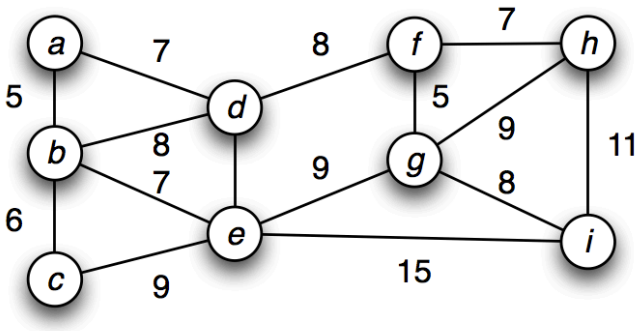


Figure 3: A weighted graph

A graph that does not have a weight function assigned to any of its edges is an **unweighted** graph. An unweighted graph is equivalent to a weighted one in which each edge is assigned the unit weight, such that `∀ e ∈ E, w(e) = 1`.

Other graph terms

A few other terms that we'll be using everywhere from now on are given below.

Degree
: Degree of a vertex `v` is the number of vertices `u` for which `(u,v) ∈ E` or `(v,u) ∈ E` (denote `deg(v)`). The number of **incoming edges** to a vertex `v` is called **in-degree** of the vertex (denote `indeg(v)`). The number of **outgoing edges** from a vertex is called **out-degree** (denote `outdeg(v)`). For an undirected graph, `deg(v) = indeg(v) = outdeg(v)`.

path
: a path is sequence of edges starting from a source vertex leading to a target vertex.

simple path
: a path is *simple* if no vertex appears more than once in the path. In a tree, every path is simple.

cycle
: a cycle is a simple path that has the same first and last vertex. A tree cannot have a cycle by definition, since there must be a single simple path between every pair of nodes.

connected-ness
: if there is a path between every pair of vertices, then it is **connected**. A tree is connected by definition. A graph has one or more **connected components**. Fig. 4 shows a graph with 2 connected components.

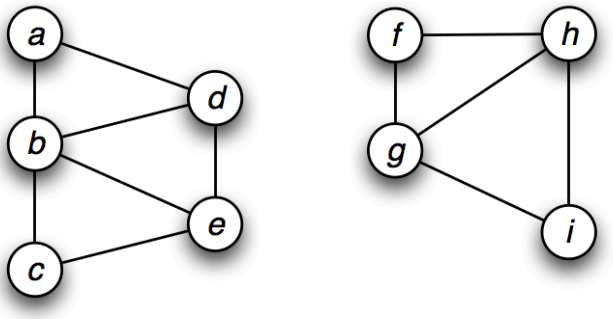


Figure 4: Graph with 2 connected components

Back to Table of contents

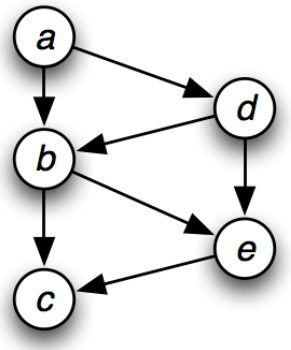
Data structures and representations

- Adjacency Matrix
- Adjacency List
- Comparison of the representations

Adjacency Matrix

The graph is represented as a $n \times n$ matrix $A = (a_{i,j})$, where $a_{i,j} = 1$ if $(v_i, v_j) \in E$, or 0 otherwise.

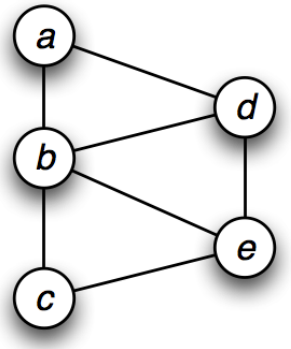
Fig. 5 shows a directed graph and it's adjacency matrix representation.



	a	b	c	d	e
a	0	1	0	1	0
b	0	0	1	0	1
c	0	0	0	0	0
d	0	1	0	0	1
e	0	0	1	0	0

Figure 5: A directed graph and it's adjacency matrix representation

Fig. 6 shows an undirected graph and it's adjacency matrix representation.



	a	b	c	d	e
a	0	1	0	1	0
b	1	0	1	1	1
c	0	1	0	0	1
d	1	1	0	0	1
e	0	1	1	1	0

Figure 6: An undirected graph and it's adjacency matrix representation

Adjacency List

This represents the graph as an array of lists, where each array slot represents a vertex, and the list represents its adjacent vertices (respecting directionality in case of a directed graph).

Fig. 7 shows a directed graph and it's adjacency list representation.

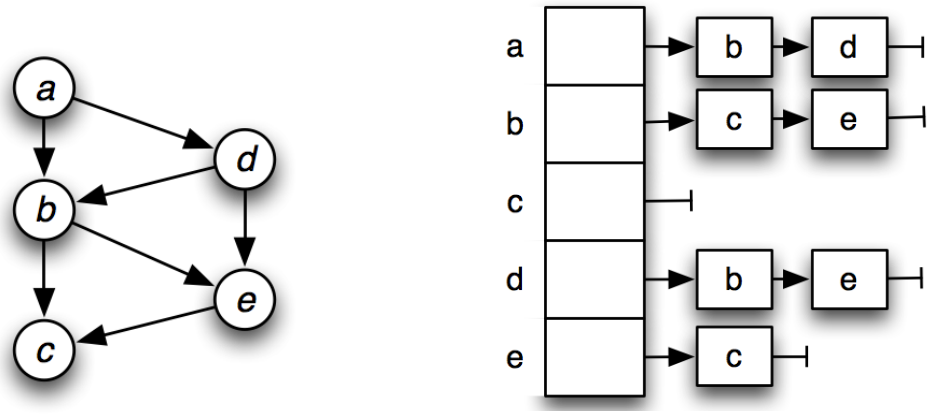


Figure 7: A directed graph and it's adjacency list representation

Fig. 8 shows an undirected graph and it's adjacency list representation.

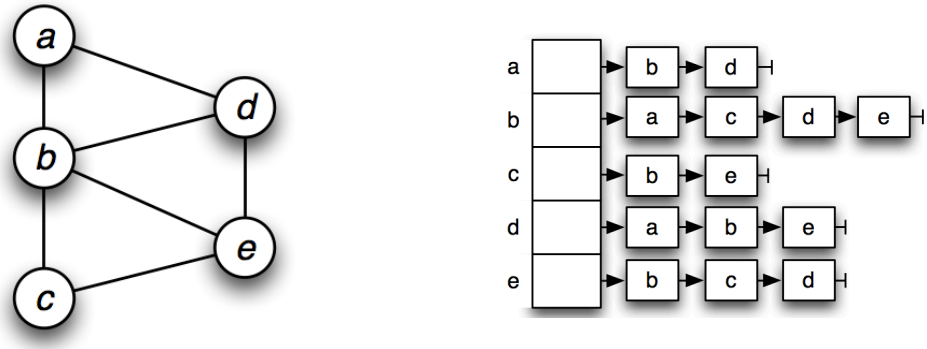


Figure 8: An undirected graph and it's adjacency list representation

Comparison of the representations

Table 1 shows the performances for the given operations using the two different representations of a graph.

Operation	Directed graph	Undirected graph
Is `(u,v) ∈ E`?	`≈ constant`	`≈ outdeg(u)`
List edges outgoing from `u`	`≈ n`	`≈ outdeg(u)`
Space requirement	`n × n`	`≈ m + n`

Table 1: Comparison of adjacency matrix and list representations

For **dense** graphs (graphs for which the number of edges $m \approx n^2$), the space requirement is the same for both, so the matrix representation wins. In general however, most graphs are **sparse** (graphs for which $m << n^2$), so the adjacency list representation is more appropriate.

Back to Table of contents

Tree Graphs

A graph `G` with `n` vertices is a **tree** if any of the following are satisfied (in other words, the following are all equivalent):

1. There are `n-1` edges and no cycles.
2. There are `n-1` edges and G is connected.

3. Exactly one path connecting each pair of vertices.
4. G is connected, but does not remain connected if any edge is removed.

Back to Table of contents

Graph traversal

The graph traversal problem is stated as follows:

> Given a graph `G = (V,E)`, and a *distinguished vertex* `s`, how do you visit each vertex `v ∈ V` **exactly** once.

The two traversals we will study are **breadth-first** and **depth-first** traversals.

Breadth-first traversal

In breadth-first traversal, we start with the source vertex and *fan* out from there. It finds the distance (in terms of number of edges or hops) of each node from the source vertex. Fig. 9 shows the breadth-first tree of the graph shown in Fig. 1, using **`A`** as the source vertex. The vertices are annotated with the shortest distance from the source vertex, and the arrows show the ancestor of each vertex after the traversal. The solid lines represent the tree edges, and the dashed lines represent the non-tree edges, which are back edges in the case of this undirected graph. The figure on the right shows the BFS tree in a more *tree-like* fashion.

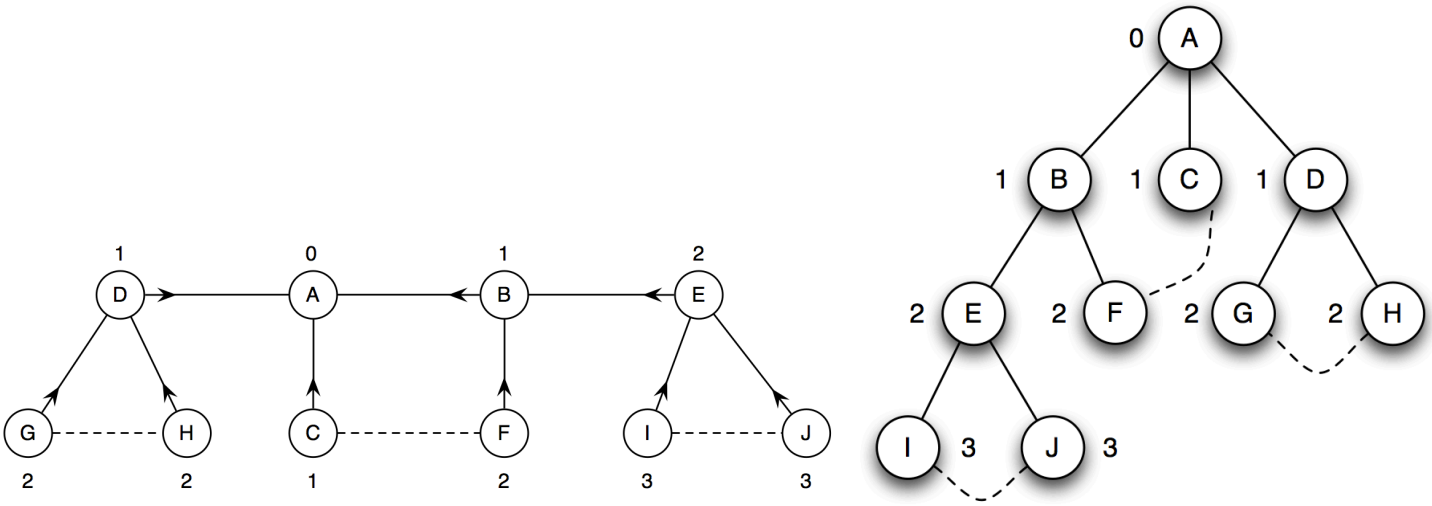


Figure 9: Breadth-first search tree of graph in Fig. 1

Since we've only performed a single BFS traversal starting at `A`, the resulting BFS tree only contains the vertices that are in the same connected component. Vertices `K` and `L` are not **reachable** from `A`, so are not visited in this traversal. If we wanted to visit **all** the vertices, then we would need to perform a BFS traversal starting at **each** vertex `v ∈ V` in the graph. After each BFS traversal starting at any vertex, all the vertices in its connected component are marked as *visited*; a BFS traversal that starts at a vertex marked *visited* returns immediately, so there is not much loss of performance.

Depth-first traversal

In depth-first traversal, we start with a source vertex and recursively visit each adjacent vertex, backtracking only when all the

choices are exhausted. Fig. 10 shows the depth-first tree of the graph shown in Fig. 1, using `A` as the source vertex. The vertices are annotated with the discovery and finish times of each vertex. The solid lines represent the tree edges, and the dashed lines represent the non-tree edges, which are back edges in the case of this undirected graph. The figure on the right shows the DFS tree in a more *tree-like* fashion.

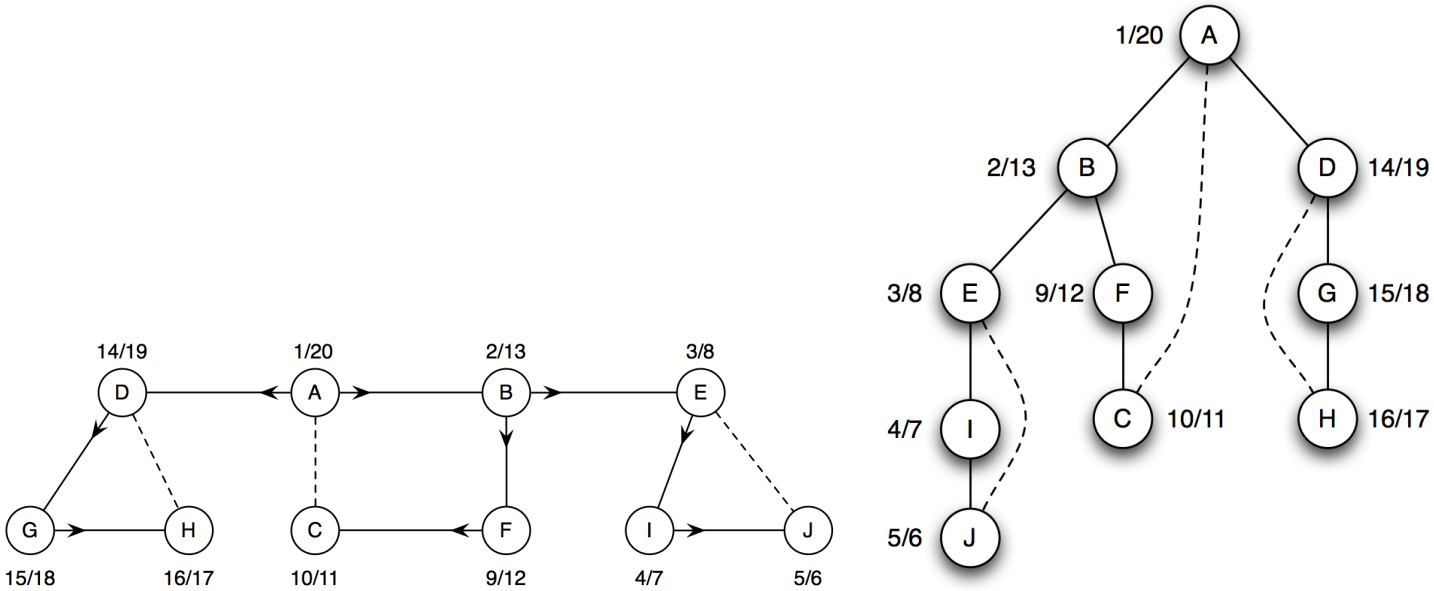


Figure 10: Depth-first search tree of graph in Fig. 1

Why do we care about these **discovery** and **finish** times for each vertex? We can use these numbers to **topologically sort** the graph. More (and much more!) on that next semester …

Since we've only performed a single BFS/DFS traversal starting at `A`, the resulting BFS/DFS tree only contains the vertices that are in the same connected component. Vertices `K` and `L` are not **reachable** from `A`, so are not visited in this traversal. If we wanted to visit **all** the vertices, then we would need to perform a BFS/DFS traversal starting at **each** vertex `v ∈ V` in the graph. After each BFS/DFS traversal starting at any vertex, all the vertices in its connected component are marked as *visited*; a DFS traversal that starts at a vertex marked *visited* returns immediately, so there is not much loss of performance.

Back to Table of contents

Some questions to ask of a graph

Here are a few typical questions to ask of a graph, and all of these can be answered using either BFS or DFS or both.

1. Starting from a given *source* vertex, can I reach a *target* vertex? This is the **s-t connectivity** problem.

   Solution: Start a BFS or DFS starting at the *source* vertex, and if at any time during the traversal, we reach the *target* vertex, we know that the *target* is reachable from *source*. Remember that BFS and DFS visit all the vertices in the same connected component as the source vertex, which are all the vertices reachable from it.

2. Starting from a given *source* vertex, how can I visit all the other vertices reachable from the source?

   Solution: See the previous solution.

3. Is the graph *connected*? If not, how many connected components does it have (and what are these)?

   Solution: The first part can be answered by simply performing a BFS or DFS starting at any vertex. If, after the traversal is done, all the vertices have been visited, then the graph is connected.

   For the second part, we initialize the number of components to be `0`, and perform a BFS or DFS starting at each vertex `v ∈ V` in the graph. Every time we encounter a starting vertex that is yet unvisited (means it's either the first time we're performing a BFS/DFS, or it's not in the same component with another vertex that was used as a starting vertex), we increment the number of connected components.

4. Is there cycle in this graph? The presence of one or more **back edges** in the BFS or DFS-tree represent a cycle. Can be answered with either BFS or DFS traversal/search.

   Solution: If, during a BFS or DFS traversal, we encounter **back edges**, the graph has at least one cycle. For an undirected graph, **all** non-tree edges are back edges; for directed graphs, there are 3 types of non-tree edges: (1) back, (2) forward, and (3) cross. Only a back edge represents a cycle.

5. Starting from a given *source* vertex, what is the shortest distance (in terms of the number of edges) of each vertex from the source vertex?

   Solution: Perform a BFS, and the **discovery time** annotation of each vertex reachable from the *source* vertex show the shortest distance from the source. Note that this **cannot** be answered with DFS traversal.

Back to Table of contents

Discussion

Back to Table of contents

Hashing and Hashtables

Table of contents

Introduction

A hash table is a data structure used for a *Dictionary/Map* ADT that provides constant time insertion (even in the worst case) and near-constant time search (in the average case).

To understand how hashing and hashtable works, let's review the *key-indexed search* method, which gives us a constant-time search for **integer-only** keys. The setup requires an intermediate or an auxiliary array of length equal to the maximum key value (the space cost). To set up the auxiliary array we do the following: for each key in the input sequence, use the key as the index into the auxiliary array, and increment the value in that slot (or just set it to be non-zero if you're not concerned about sorting). To search for a key, we use the key as the index into the auxiliary array, and see if the value is non-zero (exists) or zero (does not exist).

The basic requirement of the key-indexed search is that the keys must be integers (and only integers, since the keys are used as indices into an array). The limitations are the following:

1. **`k` must not be very large compared to `n`**: If the range `k` is much larger than `n`, then the space cost may be too high. For example, if the input sequence is `‹5, 3, 7, 10^6›`, then we'll need an 10^6 element auxiliary array, which is obviously ridiculously too large for a 4-key input sequence.
2. **keys must be non-negative**: Since the keys must be valid indices, the keys must be ≥ 0. There is a trivial solution to this problem of course: shift all keys by the most negative key such that the smallest key becomes 0 (Java arrays use 0-based indexing). *Solved*.
3. **only keys, no values associated with keys**: If there are keys that have associated values, we lose the value if there are multiple occurrences of the same key with different values (we only count the number of times a key occurs, not the values associated with each occurrence of the key). The solution is to use an array of "buckets", where each bucket contains the `(key,value)` pairs for a particular key. We can implement the buckets using arrays (the auxiliary array is then an array of arrays), or using linked lists or chains (the auxiliary array is then array of lists or chains). The second method is often preferable. *Solved*.

We still have two problems:

1. the basic requirement that the keys be integers only, and

2. the limitation that k >> n causes the auxiliary array to be too large for practical use.

Hashing and hashtables solve both of these problems quite well, but in the process we lose the ability to sort. Let's start with the limitation that k must be reasonably small first, and then we'll deal with the non-integer keys.

Back to Table of contents

Basic Principle

The basic idea behind hashing is to use a "hash" function to map a key into an index (which must be a valid index into the auxiliary array known as the hashtable). Since we want to minimize the length of the hashtable, we need to perform some form of compression to map a very large key to a small valid index, which is a key step in hashing.

Back to Table of contents

Compression

If we could compress the keys into a smaller indices, then we could use a smaller auxiliary array. The idea is trivially simple - use a circular array. The simplest compression scheme is to use the modulus (%) operator to compress the keys so that the resulting index is a valid index into the auxiliary array. The hash function takes a key, computes the index using the modulus operator. If `m` is the number of slots in the auxiliary array:

```
hash(key) = key % m
```

which guarantees that the hash value is between 0 and m-1. Take the following sequence for example: `‹5, 3, 7, 10^6›`, If we use m = 11 (a prime number. why? see Issues later on), we can build the following hash table:

```
  0    1    2   3   4   5   6   7   8   9   10
-----------------------------------------------
|  | 10^6 |   | 3 |   | 5 |   | 7 |   |   |   |
-----------------------------------------------
```

Note: `hash(10^6)` returns 1.

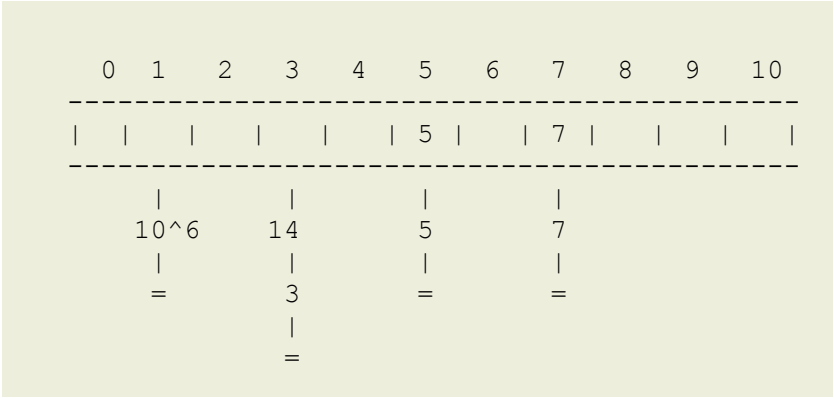
Now searching for a key can be done at constant time, and using very little space cost (for the auxiliary array).

Let's now add 14 to the sequence:

```
hash(14) = 14 % 11 = 3
```

ooops. There is already a key in that slot, and we have what's called a COLLISION. We've already seen something similar in key-indexed searching when we had non-distinct keys with (key,value) pairs, and

there we used array of buckets or chains to resolve the collision. We'll do the same here. Each array slot is now a linked list (probably using a dummy head reference). I'm only showing the non-empty chains below.



Why is 14 first in the list? (hint: where is it easy to add to a linked list?)

This particular implementation of a hashtable is called **separate chaining**. We've already seen this type of "arrays of lists" before, so there's really nothing new in terms of data structure here! We will look at other implementations later on in this document.

Back to Table of contents

Non-integer keys

Of course, we still have the basic requirement that the keys be integers. We now extend our hash function to map any arbitrary object to a large integer first, and then use compression to map it to a valid index. For example, if we have keys that are strings, our hash function would first somehow map each string to an integer, and then use compression to make it fit into the hashtable. See Chapter 14 of the textbook for details. Java makes it easy by providing a `hashCode()` method for each object (which returns a large integer, which we then need to compress to fit into our auxiliary array), so our hash function becomes:

```
public static int hash(Object o, int m) {
    int slot = o.hashCode() % m;
    return slot;
}
```

One thing to note that is that `hashCode()` may return a negative number (search google to see why), so we'll have to modify our code a bit:

```
public static int hash(Object o, int m) {
    int slot = o.hashCode() % m;
    if (slot < 0)
        slot = m + slot;
    return slot;
}
```

Back to Table of contents

Types of hashtables

Two common types of hashtables, each uses a different technique for resolving collisions:

1. Separate chaining: implementing the "buckets" using linked lists.
2. Open addressing: in case of collision, "probes" to see the next empty location where to put the key. Works well, but deleting a key is fairly complicated.

Back to Table of contents

Issues

1. How big should a hashtable be? Well, it should be bigger than "n". The idea is that the **load factor** n/m should be fairly small so that each chain is sufficiently short to make the search constant time. If the load factor becomes too high, you'll have to create a larger hash table and re-hash all the existing keys into this larger hash table.
2. Why a prime number? To avoid having patterns in the input sequence that produces collisions. For example, with m = 10, and the input sequence of < 0, 10, 20, 30, 40, 500, 1000, 23500 > produces 7 collisions with 8 keys!
3. Why do we need to rehash when resizing a hashtable? Because the compression uses the length of the hashtable, the indices into the older table won't match the indices into the new table.
4. There is no quick way to get the keys back in sorted order from a hashtable. Contrast this with a dictionary implemention using binary search tree, where a simple in-order traversal produces sorted keys as output. No choice but to extract the keys (using iteration), and then use an external sorting algorithm to get the keys in sorted order.

   Likewise, there is no quick way to tell what a successor or predecessor of a key in the table is (useful if you're looking for the keys closest to some key).

Back to Table of contents

Data Structures and implementation

For the separate chaining, we use the same data structure we've used for Adjacency List for Graphs. For open addressing/probing, we use a simple array.

Back to Table of contents

Discussion

Back to Table of contents