Contents

Lecture 1

Testing

1.1. Test case design

- **Illegal cases:** Illegal states for data.
- **Boundary cases:** "Edges" of data ranges.
- **Typical cases:** Norma; or expected data range.
- **Special cases:** Unusual or exceptional values or states.

1.2. Linear Search

```
1  public static int search(int [] a, int target) {
2      if (a == null || a.length == 0) {
3          return NullPointerException(" illegal case");
4      }
5      int i = 0;
6      while (i < a.length && a[i] == target) {
7          i++;
8      }
9      if (i < a.length) { // found
10         return i;
11     }
12     else { // not found
13         return -1;
14     }
15 }
```

1.3. Min of three

```
1  public static int min(int a, int b, int c) {
2      if (a <= b && b <= c) {
3          return a;
4      }
5      else if (b <= a && a <= c) {
6          return b;
7      }
```

```
    else {
        return c;
    }
}
```

- **Illegal cases:** none
- **Boundary cases:** min/max values
- **Typical cases:** random int values
- **Special cases:** duplicates (let 0 = min, 1 = not min)

a	b	c
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

```
@Test public void minTestSpecialCases {
    Assert.assertEquals(1, MinOfThree.min(1, 2, 3)); // 0 1 1
    Assert.assertEquals(1, MinOfThree.min(2, 1, 3)); // 1 0 1
    Assert.assertEquals(1, MinOfThree.min(2, 3, 1)); // 1 1 0
    Assert.assertEquals(2, MinOfThree.min(2, 2, 2)); // 0 0 0
    Assert.assertEquals(2, MinOfThree.min(2, 2, 3)); // 0 0 1
    // stay tuned ...
}
```

Lecture 2

Generality

2.1. Making code more general

What we do:

1. Choose very general data type but retain type-safety.
2. Select a representation for a "list" that is less concrete and more flexible.
3. Employ general programming idioms to make code more reusable.

```
1  int search(Object[] a, Object target) {
2      if (a == null || a.length == 0) {
3          return NullPointerException(" illegal case");
4      }
5      int i = 0;
6      while (i < a.length && a[i].equals(target)) {
7          i++;
8      }
9      if (i < a.length) { // found
10         return i;
11     }
12     else { // not found
13         return -1;
14     }
15 }
```

2.1.1. Examples: equals

- String overrides equals – **OK**

```
1  public void testSearchString() {
2      String[] a = {"2", "4", "6", "8", "10"};
3      String target = "8";
4      int expected = 3;
5      int actual = search(a, target);
6      Assert.assertEquals(expected, actual);
7  }
8
```

- Integer overrides equals – **OK**

```
1  public void testSearchInteger() {
```

```
    Integer [] a = {2, 4, 6, 8, 10};
    Integer target = 8;
    int expected = 3;
    int actual = search(a, target);
    Assert.assertEquals(expected, actual);
}

```

- **NOT OK**, unless Book overrides equals

```
public void testSearchBook() {
    Book[] a = {new Book("Author 1", "Title 1", 123),
                new Book("Author 2", "Title 2", 456),
                new Book("Author 3", "Title 3", 789),
                new Book("Author 4", "Title 4", 666)};
    Book target = new Book("Author 4", "Title 4", 666);
    int expected = 3;
    int actual = search(a, target);
    Assert.assertEquals(expected, actual);
}

```

Thus, **this is the responsibility of the underlying data classes to provide an appropriate implementation of the equals method**.

2.2. `equals` Method

- **The equals method** is defined in class Object, so all classes in Java have equals.
- The default implementation is the same as == operator (i.e., aliasing).
- You may want to override equals when the class you're writing has a notion of **logical equality** that is different from the mere object identity, and a superclass has not already overridden equals to implement the desired behavior.
  - Java classes, such as String, Integer, etc. have customized, overridden implementations of equals.
  - Note that the following does NOT overrides equals, it overloads it:

```
public boolean equals(Book that) {
                    // alert: formal parameters does not
                    // match definition in Object class
    return this.title.equals(that.title);
}

```

`public boolean equals(Object obj)`
The equals method implements an equivalence relation on non-null object references.

- **Reflexive:** x.equals(x) >> true
- **Symmetric:** if x.equals(y) >> true, then y.equals(x) >> true
- **Transitive:** if x.equals(y) >> true and y.equals(z) >> true, then x.equals(z) >> true
- x.equals(**null**) >> **false**

`equals` **Recipe**

1. The formal parameter must be of type Object the to override:

```
public boolean equals(Object o) {
    // ...
}

```

2. Check for **aliasing**:

```
if (o == this) return true;

```

3. Check for **null**:

```
if (o == null) return false;

```

4. Check for **type compatibility**:

```
if (!(o instanceof Book)) return false;

```

5. Cast from Object:

```
Book b = (Book) o;

```

2.3. Type compatibility

- The reference type of object in the declaration (left of =) is its **static (compile-time) type**, and the object type in the instantination (right of =) is its **dynamic (run-time) type**.

- **Polymorphism:** We can use a reference of a superclass type to refer to an object of that type as well as any subclass type (any number of levels down the class inheritance hierarchy).
  - **The static type of reference can be the same as its dynamic type or a superclass of its dynamic type**.

2.3.1. Casting up

This is called **casting up** the class hierarchy, since the cast is towards the root:

```
ClassA ref1 = new ClassA();
ClassB ref2 = new ClassB();
ref1 = ref2; // A = B
```

2.3.2. Casting down

Consider the following:

```
TV tv = null;
if (<consition>) {
  tv = new TV();
}
else {
  tv = new StereoTV();
}...

tv.setBalance(7); // setBalance declared in StereoTV class
```

- The last statement would be incorrect if tv were to refer to a TV object, since TV does not define a setBalance method. But the statement would be correct if tv were indeed refer to a StereoTV object.
- **Although one can use a superclass reference to refer to a subclass object, one cannot use it to invoke a method that is special only to the subclass.**

If we know for sure that tv indeed refers to a StereoTV object, we may invoke the setBalance method by forcing a **cast down** the class hierarchy:

```
TV tv = new StereoTV();
((StereoTV)tv).setBalance(7);
```

However, **if the reference we are casting actually refers to the superclass type object, and not to an object of a subclass type to which it is being cast, the program will issue a runtime error**.

2.3.3. instanceof Operator

- Java provides the **instanceof operator** to determine the relationship between a reference and the type of object to which it refers at runtime.
- If obj refers to an object whose **run-time type** is either Y or a subclass of Y, it is **true**.

```
obj instanceof Y;
```

2.3.4. getClass Method

- **Returns a run-time class of an object**.
- Does **NOT** allow mixed-type equality.

```
if (obj.getClass() != this.getClass()) {
  return false;
}
```

- Always **type-safe** if the client implements equals.
- Must restrict operations to those common to all Objects (e.g., equals).
  - If the algorithm being implemented requires any operation Object can't perform, this approach generally does **NOT** work.

2.4. Comparable Interface

- Imposes a **total ordering** on the objects of each class that implements it. This ordering is referred to as the class's *natural ordering*, and the class's **compareTo method** is referred as its *natural comparison method*.
- **It is strongly recommended the natural ordering be consistent with equals.**

Note that **null** is not an instance of any class, and x.compareTo(**null**) should return **NullPointerException** even though x.equals(**null**) returns **false**.

```
public int search(Comparable[] a, Comaprable target) {
    int i = 0;
    while (i < a.length && a[i].comapreTo(target) != 0) {
        i++;
    }
    if ( i < a.length) {
        return i;
```

```
8       }
9       else {
10          return -1;
11      }
12  }
```

`int compareTo(T o)`
Compares this object with the specified object for order.

```
1  x.compareTo(y); // ret i < 0 iff x < y
2                  // ret i = 0 iff x == y
3                  // ret i > 0 iff x > y
```

2.4.1. Examples: compareTo

- This will work – String overrides compareTo.

```
1  public void testSearchString() {
2      String[] a = {"2", "4", "6", "8", "10"};
3      String target = "8";
4      int expected = 3;
5      int actual = search(a, target);
6      Assert.assertEquals(expected, actual);
7  }
8
```

- This will work – Integer overrides compareTo.

```
1  public void testSearchInteger() {
2      Integer[] a = {2, 4, 6, 8, 10};
3      Integer target = 8;
4      int expected = 3;
5      int actual = search(a, target);
6      Assert.assertEquals(expected, actual);
7  }
8
```

- This will **NOT** work, unless Book implements Comparable.

```
1  public void testSearchBook() {
2      Book[] a = {new Book("Author 1", "Title 1", 123),
3                  new Book("Author 2", "Title 2", 456),
4                  new Book("Author 3", "Title 3", 789),
5                  new Book("Author 4", "Title 4", 666)};
6      Book target = new Book("Author 4", "Title 4", 666);
7      int expected = 3;
8      int actual = search(a, target);
```

```
9       Assert.assertEquals(expected, actual);
10  }
11
```

`compareTo` Recipe

1. The class must implement the Comparable interface:

```
1  public class Book implements Comparable {
2      /...
3  }
4
```

2. Preferable **NOT** Object, stay tuned...

```
1  public int comapreTo(Object o) {
2      /...
3  }
4
```

3. Cast from Object (exceptions possible if **NOT** type-compatible):

```
1  Book b = (Book) o;
2
```

4. From most significant field to least significant field, make comparisons until order ($>, <, ==$) can be determined.

Can't guarantee type-safety: Instances of distinct, incompatible classes that implement the Comparable interface are legal arguments to search method.

2.5. Type safety

- **Type safety** refers to the extent to which typing errors are prevented.
- A Java program is considered *type safe* if there are no definite or potential type errors identified by the compiler.

2.5.1. Examples: Type safe

- **Type safe**

```
1  public class ArrayLin1 {
2      public static int search(Object[] a, Object target) {
3          int i = 0;
4              while (i < a.length && !(a[i].equals(target))) {
```

```
5              i++;
6            }
7            if (i < a.length) {
8              return i;
9            }
10           else {
11             return -1;
12           }
13     }
14     public static void main(String[] args) {
15       String[] sa = {"2", "4", "6", "8", "10"};
16       int i = search(sa, "8");
17     }
18 }
19
```

- **NOT** type safe, but **NO** run-time errors.

```
1  public class ArrayLin2 {
2      public static int search(Comparable[] a, Comparable target) {
3          int i = 0;
4          while (i < a.length && a[i].compareTo(target) != 0) {
5              i++;
6          }
7          if (i < a.length) {
8              return i;
9          }
10         else {
11             return -1;
12         }
13     }
14     public static void main(String[] args) {
15         String[] sa = {"2", "4", "6", "8", "10"};
16         int i = search(sa, "8");
17     }
18 }
19
```

- **NOT** type safe **AND** generates a run-time error

```
1  public class ArrayLin3 {
2      public static int search(Comparable[] a, Comparable target) {
3          int i = 0;
4          while (i < a.length && a[i].compareTo(target) != 0) {
5              i++;
6          }
7          if (i < a.length) {
8              return i;
9          }
```

```
10          else {
11              return -1;
12          }
13      }
14      public static void main(String[] args) {
15          String[] sa = {"2", "4", "6", "8", "10"};
16          int i = search(sa, new Double(3.14));
17      }
18  }
19
```

- **Type safe**

```
1  public class ArrayLin4 {
2      public static int search(Comparable[] a, Comparable target) {
3          int i = 0;
4          while (i < a.length && !(a[i].equals(target))) {
5              i++;
6          }
7          if (i < a.length) {
8              return i;
9          }
10         else {
11             return -1;
12         }
13     }
14     public static void main(String[] args) {
15         String[] sa = {"2", "4", "6", "8", "10"};
16         int i = search(sa, "8");
17     }
18 }
19
```

- The language is designed to guarantee that **if your entire application has been compiled without unchecked warnings, it is type safe**.
- A **unchecked warning** is issued by a Java compiler to indicate that not enough type information is available for the compiler to ensure that no unexpected type error will occur at runtime.

2.6. Generics

- Java allows a **type variable** to be used in place of a specific type name.
  - A type variable can be used to parameterize a class, interface, or method with respect to the types involved.

- This generic typing allows classes, interfaces, and methods to deal with objects of different types at runtime while maintaining compile-time type safety.

2.6.1. Parameterizing a method

```
public class ArrayLib {
    public static <T> int search(T[] a, T target) {
        int i = 0;
        while (i < a.length && !(a[i].equals(target))) {
            i++;
        }
        if (i < a.length) {
            return i;
        }
        else {
            return -1;
        }
    }
}
```

The client can supply a value for the type variable when the method is called:

```
String [] sarr = {"2", "4", "6", "8", "10"};
Integer [] intarr = {2, 4, 6, 8, 10};
//...
ArrayLin.<String>search(sarr, "8");
            // tells the compiler to associate
            // the real type String with the type variable T
```

The client can leave the type variable **unspecified**:

```
ArrayLib.search(sarr, "8"); // this ops-out of the generic type system
                            // and is poor practice
ArrayLib.<String>search(sarr, "8"); // type safe
ArrayLib.<String>search(sarr, 8); // compile-time error
ArrayLib.seqrch(sarr, 8);  // not type safe, but no compile-time error
```

2.6.2. Parameterizing a class

```
public class ArrayLib<T> {
    public int search(T[] a, T target) {
        int i = 0;
        while (i < a.length && !(a[i].equals(target))) {
            i++;
        }
        if (i < a.length) {
```

```
8               return i;
9           }
10          else {
11              return -1;
12          }
13      }
14  }
```

- Since T is a type variable, and the **ArrayLib class must be instantiated in order to bind T to a concrete type**.
  - So, the search method is *no longer static*.

2.6.3. Type associated with generics in Java

- **A type variable** is an unqualified identifier used as a type in class, interface, method, and constructor bodies.
- A class, interface, or method is **generic** if it declares one or more type variables.
  - Generic classes and interfaces are called **generic types**.
- A type variable is introduced by the declaration of a type parameter of a generic class, interface, method, or constructor.
- A generic class or interface defines a set of **parameterized types**.
  - A parameterized type is of the form $C < T_1, T_2, ..., T_n >$, where C is the name of the *generic type* and $< T_1, T_2, ..., T_n >$ is the list of *type arguments* that denote a particular parameterization of the generic type.
- **Parameterized types exist ONLY at compile-time**.
  - All type arguments are eliminated through a process known as **type erasure** and do not exist at runtime.
- A **raw type** is a generic type name without parameterization; that is, a **raw type is the erasure of the generic type at compile-time**.

```
1   public class ArrayLib2 {// Comparable is the raw type of Comparable<T>
2       public int search(Comparable[] a, Comparable target) {
3           int i = 0;
4           while (i < a.length && a[i].compareTo(target) != 0) {
5               // with the raw type, here is no way for the compiler to know
6               // if compareTo will throw a ClassCastException
7               i++;
8           }
9           if (i < a.length) {
10              return i;
11          }
12          else {
13              return -1;
```

```
        }
    }
} // -> NOT type safe

```

- A type variable can have a **bound** specified in its declaration using the extends ¡type¿ notation.

```
public class ArrayLib2<T extends Comparable> {
            // Use a generic type variable with Comparable as its upper bound
    public int search(T[] a, T target) {
            // Note: if raw type, compiler can't ensure that the types in a
            //       and target are mutually comparable
        int i = 0;
        while (i < a.length && a[i].compareTo(target) != 0) {
            // with the raw type,
            // there is no way for the compiler to know
            // if compareTo will throw a ClassCastException
            i++;
        }
        if (i < a.length) {
            return i;
        }
        else {
            return -1;
        }
    }
}

```

2.7. `Comparator` Interface

`public interface Comparator<T>`

- Imposes a total ordering on some collection of objects.
- Comparators can be passed to a sort method, such as Collections.sort or Arrays.sort, to allow precise control over the sort order.

`int compare(T o1, T o2)`
Compares two arguments for order.

```
comp.compare(x, y); // ret i < 0 iff x < y
                    // ret i = 0 iff x == y
                    // ret i > 0 iff x > y
```

The following allows the client to search through the same data using different strategies for the comparison:

```
public class ArrayLib {
    public <T> int search(T[] a, T target, Comparator<T> c) {
        int i = 0;
        while (i < a.length && c.compare(a[i], target) != 0) {
            i++;
        }
        if (i < a.length) {
            return i;
        }
        else {
            return -1;
        }
    }
}

class CompareBooksByLength implements Comparator<Book> {
    public int compare(Book b1, Book b2) {
        return b1.pages - b2.pages;
    }
}
```

2.8. Iterations

- **Iterator pattern:** Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- **Iterators** are central to generic programming because they are an interface between containers and algorithms.

2.8.1. Implementing iterators

- One of the fundamental operations on collections is to process each item by iterating through the collection using Java's **foreach statement**:

```
Stack<String> collection = new Stack<String>();
/...
for (String s : collection) {
    System.out.println(s);
}
```

- This foreach statement is shorthand for a while construct. It is essentially equivalent to the following while statement:

```
Iterator<String> i = collection.iterator();
while (i.hasNext()) {
    String s = i.next();
```

```
4       System.out. println (s);
5   }
```

To implement in any iterable collection:

1. Add the phrase implements Iterable¡T¿ to its declaration, matching the interface:

```
1  public  interface  Iterable <T> {
2       Iterator <T> iterator();
3  }
4
```

2. Add a method **iterator** to the class that returns an Iterator<T> object.
    - Note that Iterators are *generic*, so we can use our parameterized type T to allow clients to iterate through objects of whatever type is provided by our client
3. The Iterator class **MUST** include two methods:
    - **hasNext** = returns a boolean value
    - **next** = returns a generic item from the collection

Lecture 3

Algorithm Analysis

Algorithm analysis: An approach to describe the time/work or space requirements of an algorithm in terms of its input size.

3.1. Approaches to algorithm analysis

- **Empirical analysis:** Analyze running time based on observations and experiments.
- **Mathematical analysis:** Develop a cost model that includes costs for individual operations.

Two different but complementary approaches to achieving the same result: **Characterizing the running time of a program (its time complexity) as a function of a problem size**.

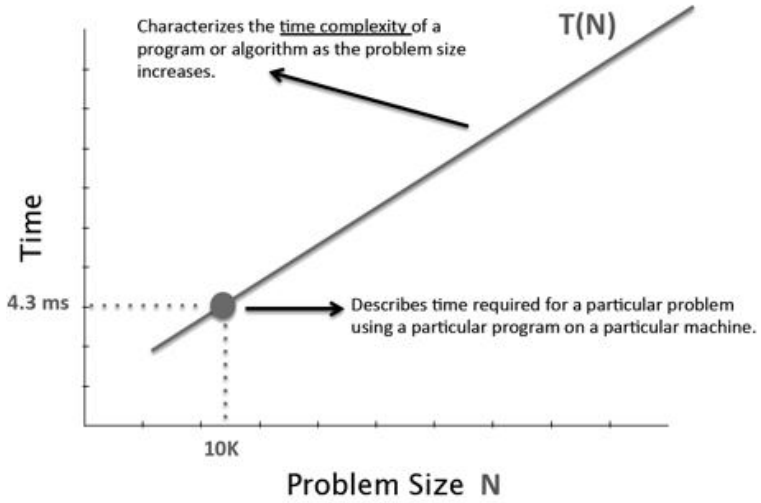


Figure 1.

3.2. Empirical analysis

The ratios approach a limiting value of 2^k:

- **The order of growth** of the running time is approximately N^k.
- **To predict running times**, multiply the last observed running time by 2^k, continuing as long as desired.

$$T(2N_i) = 2^k T(N_i) \tag{3.1}$$

N	T(N)	Ratio
N	T(N)	-
2N	T(2N)	T(2N)/T(N)
4N	T(4N)	T(4N)/T(2N)
8N	T(8N)	T(8N)/T(4N)
. . .	. . .	. . .

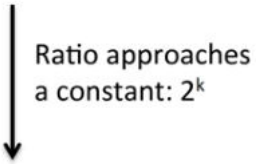


Figure 2.

3.2.1. Example: Empirical analysis

- **Hypothesis:** As N doubles, running time is increased by a factor of 4

N	Elapsed Time	Ratio
250	0.061	N/A
500	0.042	1.35
1000	0.112	2.67
2000	0.340	3.04
4000	1.298	3.82
8000	5.334	4.11
16000	21.88	4.10

- **Perdition:** For $N = 3200$, elapsed time will be 87.52 s and for $N = 64000$, elapsed time will be 350.08 s.
- **Experimentally test the hypothesis:**

N	Elapsed Time	Ratio
32000	85.763	3.92
64000	345.634	4.03

- Since we corroborated our hypothesis, we can say that the ratio will converge to $4 = 2^2 = N^2$, so running time is growing in proportion to N^2.

3.3. Mathematical analysis

Computational model:

$$\text{Total running time} = \sum c_i \times f_i \tag{3.2}$$

where

- c_i – **the cost of executing the operation** i (a property of the underlying system);
- f_i – **the frequency of execution of operation** i (a property of the algorithm).

3.3.1. Examples: Mathematical analysis

- **Abstraction decision:** Treat the cost of primitive operations as some unspecified constant.

```
int sumA(int N) {
    int sum;
    sum = N * (N + 1) / 2;
}

```

Operation	Cost (c_i)	Freq (f_i)
variable declaration	1.5 ns	1
assignment	1.7 ns	1
int addition	2.1 ns	1
int multiplication	2.5 ns	1
int division	5.4 ns	1
return statement	2.0 ns	1

  - Running time of sumA is **constant** (= same for any value of N).

- **Abstraction decision:** Focus only on core operations instead counting every single operation that is performed.

```
int sumB(int N) {
    int sum = 0;
    for (int i = 1; i <= N; i++) {
        sum = sum + 1;
    }
}

```

Operation	Cost (c_i)	Freq (f_i)
sum = sum + i	c	N

– Running time of sumB is **linear** (= increases in direct proportion to N).

- **Abstraction decision:** Focus on the highest-order terms and ignore coefficients, constants, and low-order terms.

```
1 int sumC(int N) {
2     int sum = 0;
3     for (int i = 1; i <= N; i++) {
4         for (int j = 1; j <= i; j++) {
5             sum = sum + i;
6         }
7     }
8 }
9
```

Operation	Cost (c_i)	Freq (f_i)
sum = sum + i	c	N(N+1)/2

– Running time of sumC is **quadratic** (= increases in direct proportion to N^2).

```
public int sumA(int N) {
   int sum;
   sum = N*(N+1)/2;
   return sum;
}
```

Performs a **constant** amount of work.

```
public int sumB(int N) {
  int sum = 0;
  for (int i = 1; i <= N; i++) {
     sum = sum + i;
  }
  return sum;
}
```

Performs a **linear** (~N) amount of work.

```
public int sumC(int N) {
  int sum = 0;
  for (int i = 1; i <= N; i++) {
     for (int j = 1; j <= i; j++ {
        sum = sum + 1;
     }
  }
  return sum;
}
```

Performs a **quadratic** (~N^2) amount of work.

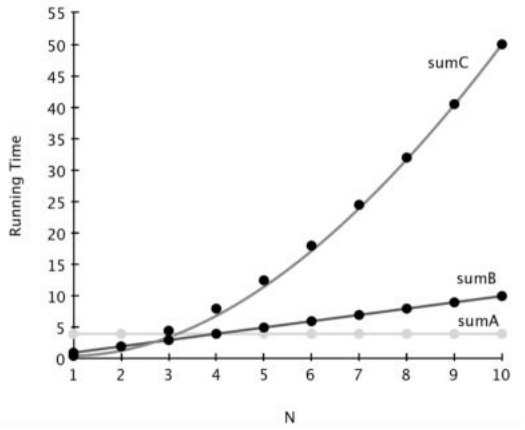


Figure 2.

We are characterizing time complexity by the running time function's **growth rate**.

3.4. Analysis of Binary Search

```
1  int binarySearch(int [] arr, int target) {
2      int left = 0;
3      int right = arr.length - 1;
4      while (left <= right) {
5          int middle = (right - left)/2;
6          if (target == arr[middle]) {
7              return middle;
8          }
9          else if (target < arr[middle]) {
10             right = middle - 1;
11         }
12         else {
13             right = middle + 1;
14         }
15     }
16     return -1;
17 }
```

Worst-case analysis: As a function of the array size N, what is the maximum number of times this statement could be executed?

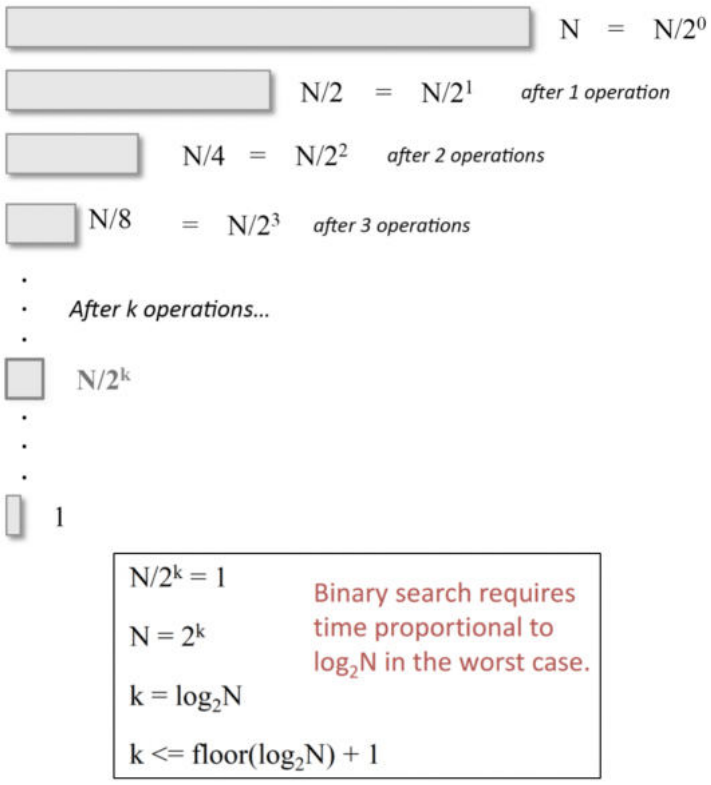


Figure 3.

3.5. Growth rate

- **The growth rate** of a time complexity function is a measure of how the amount of work an algorithm does changes as input (problem) size increases.
  - The growth function is also called its **order**.
  - A useful way of thinking about growth rate for a time complexity function $T(N)$ is to think about the change in $T(N)$ as N doubles.

3.6. Big-Oh notation

- We will describe the growth rate of an algorithm's time complexity function in terms of **big-Oh**.
  - We will make statements like this: The time complexity of this algorithm is $O(N^2)$.

- **Big-Oh:** Let $f(n)$ and $g(n)$ be functions defined on the non-negative integers. We say "$f(n)$ is $O(g(n))$" iff there exists a non-negative integer n_0 and a constant $c > 0$ such that for all integers $n \geq n_0$ we have $f(n) \leq cg(n)$.
  - **Ex.:** $f(N) = N^2 + 2N + 1$ is $O(N^2)$ since for $N \geq 1$ if $f(N) \leq 4N^2$.

Just use the fastest growing term in the function and drop any coefficients it may have.

3.6.1. Order of growth

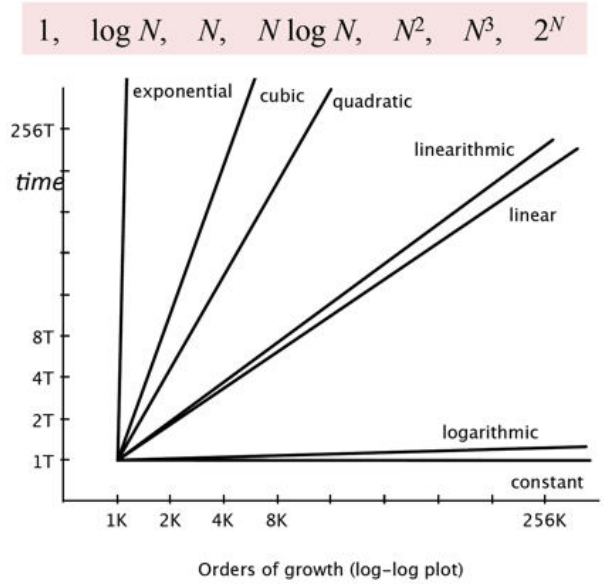


Figure 4.

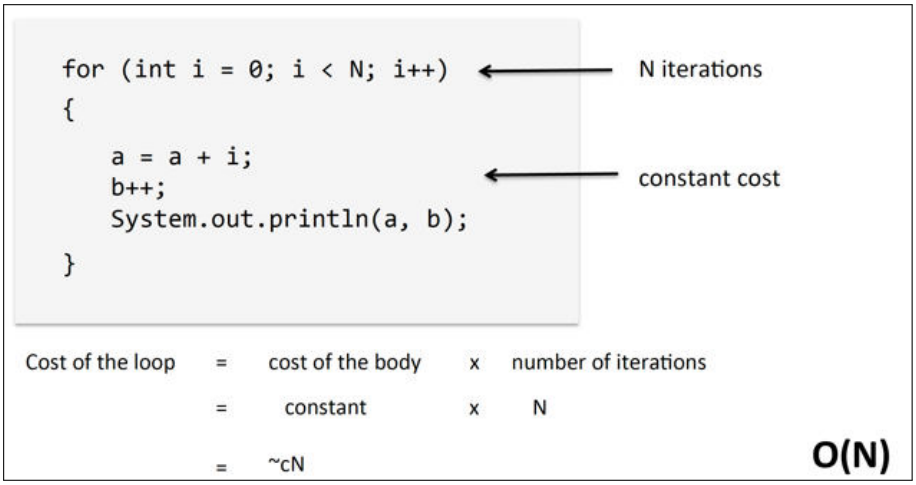
3.6.2. Big-Oh analysis

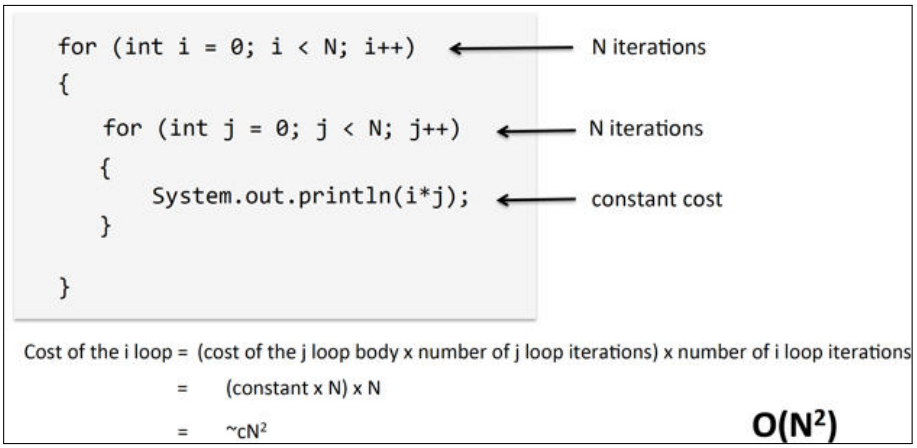
- **Big-Oh analysis** refers to analyzing an algorithm or program and expressing its running time in terms of big-Oh.
- **Types of analysis:**
  - **Best-case:** Provides a *lower* bound on cost
  - **Average-case:** Provides an expectation of cost
  - **Worst-case:** Provides an *upper* bound on cost
    - Defined by the "most difficult" input
    - Provides a guarantee on the performance

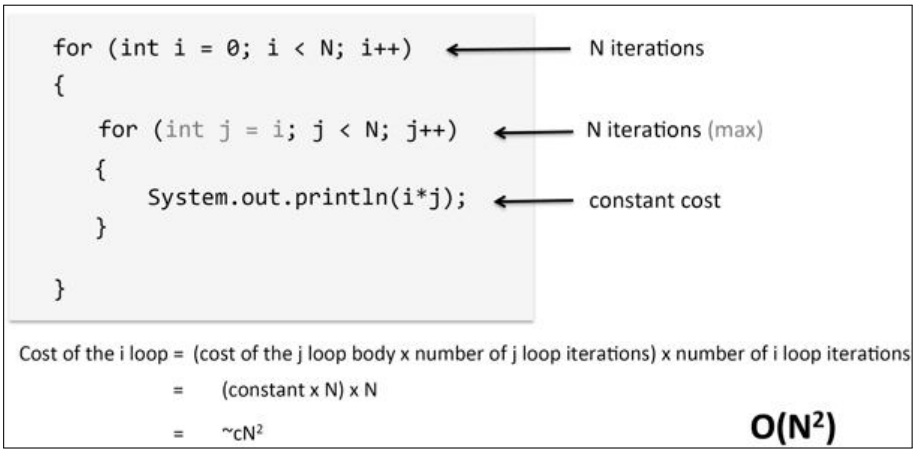
Calculating the worst-case big-Oh:

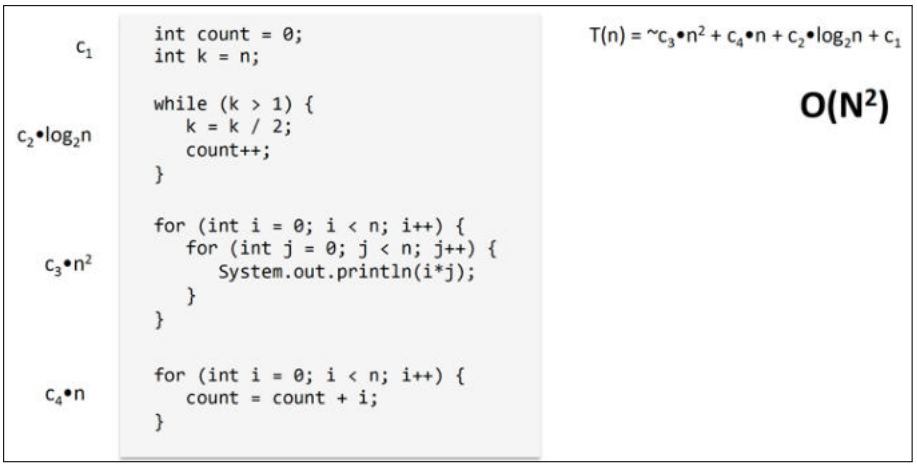
1. **All primitive operations** have *constant cost*.
2. **The cost of a sequence of statements** is the *sum of the costs of each individual statement*.
3. **The cost of a selection statement** is the *cost of the most expensive branch*.
4. **The cost of the loop** is the *cost of the body multiplied by the maximum number of iterations that the loop makes*.

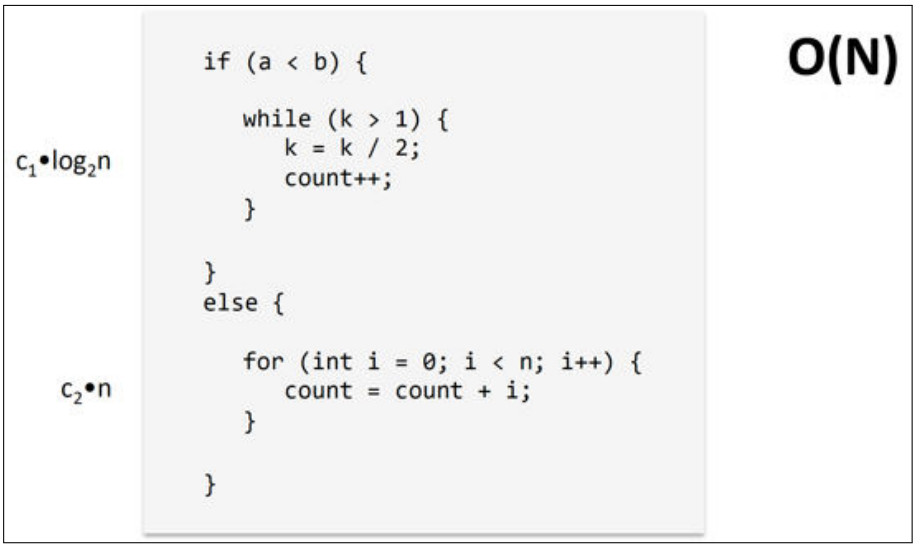
3.6.3. Examples: Big-Oh analysis




for (int i = 0; i < N; i++)
N iterations
{
for (int j = 0; j < N; j++)
N iterations
{
System.out.println(i*j);
constant cost
}
}
Cost of the i loop = (cost of the j loop body x number of j loop iterations) x number of i loop iterations
= (constant x N) x N
= ~cN²
O(N²)


for (int i = 0; i < N; i++)
N iterations
{
for (int j = i; j < N; j++)
N iterations (max)
{
System.out.println(i*j);
constant cost
}
}
Cost of the i loop = (cost of the j loop body x number of j loop iterations) x number of i loop iterations
= (constant x N) x N
= ~cN²
O(N²)


c1
int count = 0;
int k = n;
while (k > 1) {
c2•log2n
k = k / 2;
count++;
}
for (int i = 0; i < n; i++) {
c3•n²
for (int j = 0; j < n; j++) {
System.out.println(i*j);
}
}
for (int i = 0; i < n; i++) {
c4•n
count = count + i;
}
T(n) = ~c3•n² + c4•n + c2•log2n + c1
O(N²)


O(N)
$c_1 \bullet \log_2 n$
$c_2 \bullet n$
if (a < b) {
while (k > 1) {
k = k / 2;
count++;
}
}
else {
for (int i = 0; i < n; i++) {
count = count + i;
}
}

Lecture 4

Sorting

4.1. Properties of sort

- **In-place:** The list itself is rearranged and only a constant amount of extra space is required.
- **Adaptive:** Running time can be improved by taking advantage of certain input arrangements.
- **Stable:** Equal elements maintain the same relative order.

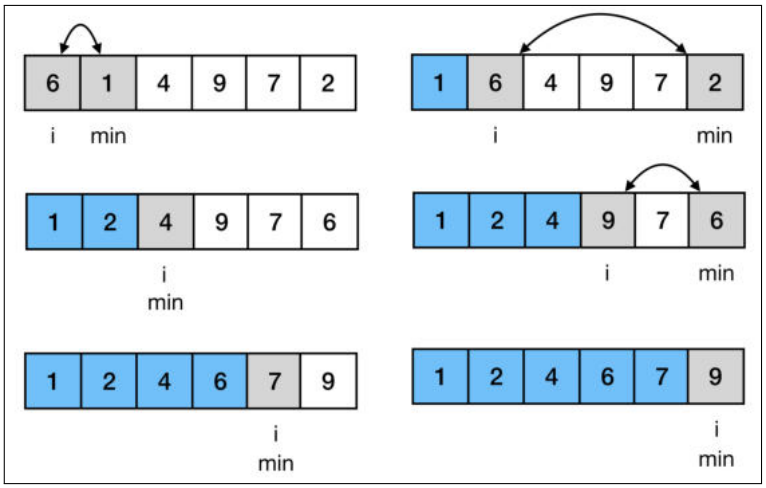
	Selection	**Insert**	**Mergesort**	**Quicksort**
In-place?	Yes	Yes	No	Yes
Adaptive?	No	Yes	No	No
Stable?	No	Yes	Yes	No
Worst-case	$O(N^2)$	$O(N^2)$	$O(N \log N)$	$O(N^2)$
Average-case	$O(N^2)$	$O(N^2)$	$O(N \log N)$	$O(N \log N)$
Best-case	$O(N^2)$	$O(N)$	$O(N \log N)$	$O(N \log N)$

4.2. Selection sort

- **Complexity:** $O(N^2)$
- **In-place**

How does Selection sort work?

1. **The list is divided into two parts, sorted part at left end and sorted part at right end.** (Initially sorted part is empty, unsorted part is the entire list.)
2. **Smallest element is selected from the unsorted array and swapped with the leftmost element and that element becomes the part of the sorted array.**
3. This procedure continues moving unsorted array boundary by one element to the right until there are no elements in the unsorted (right) array.



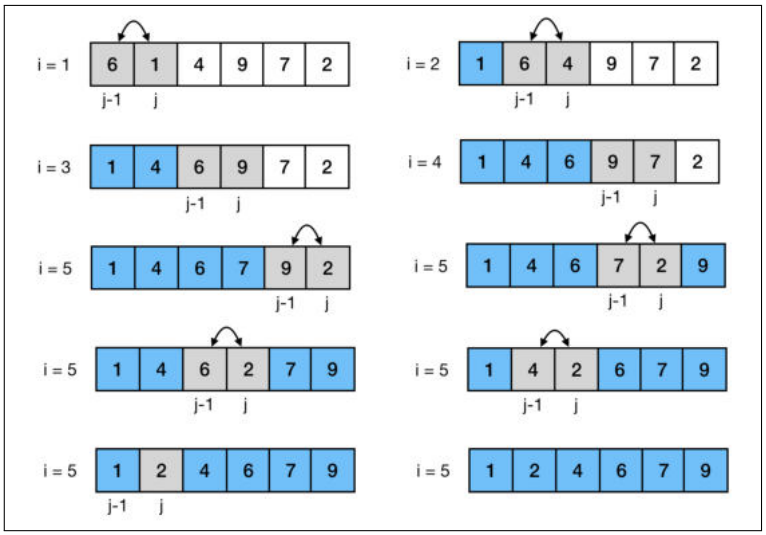

```
1  public void selectionSort (T[] a) {
2      for (int i = 0; i < a.length; i++) {
3          int min = i;
4          for (int j = i + 1; j < a.length; j++) {
5              if (less(a[j], a[min])) {
6                  min = j;
7              }
8          }
9          swap(a, i, min);
10     }
11 }
```

4.3. Insertion sort

- **Complexity:**
  - Worst/average case: $O(N^2)$
  - Best case: $O(N)$
- **In-place, adaptive, stable**

How does Insertion sort work?

1. A sub-list is maintained, which is always sorted.
2. **An element**, which is to be inserted in this sorted sub-list, **has to find its appropriate place and insert it here**.



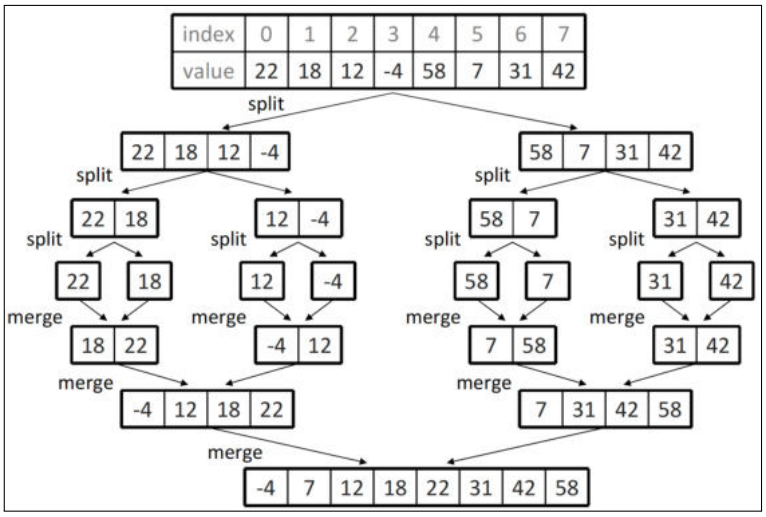

```
1  public void insertionSort (T[] a) {
2      for (int i = 0; i < a.length; i++) {
3          for (int j = i; j > 0; j--) {
4              if (less(a[j], a[j - 1])) {
5                  swap(a, j, j - 1);
6              }
7              else {
8                  break;
9              }
10         }
11     }
12 }
```

4.4. Mergesort

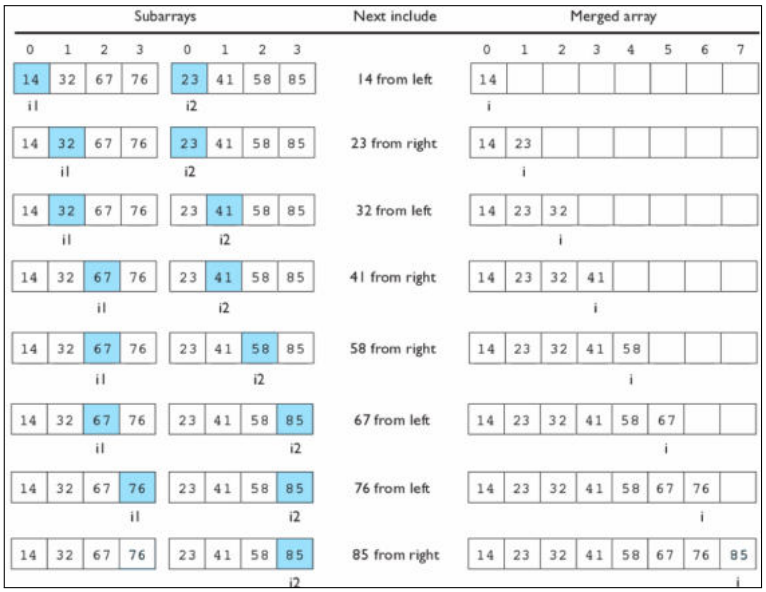
- **Complexity:** $O(N \log N)$
- **Stable**

How does Mergesort work?
Repeatedly divides the data in half, sorts each half, and combines the sorted halves into a sorted whole.

1. Divide the lost into two roughly equal parts.
2. Recursively sort the left part.
3. Recursively sort the right part.
4. Merge the two sorted halves into one sorted list.




```
1  public static void merge(int[] result, int[] left, int[] right) {
2      int i1 = 0; // index into left array
3      int i2 = 0; // index into right array
4      for (int i = 0; i < result.length; i++) {
5        if (i2 >= right.length ||
6                  (i1 < left.length && left[i1] <= right[i2])) {
7              result[i] = left[i1]; // take from left
8              i1++;
9          }
10         else {
11             result[i] = right[i2]; // take from right
12             i2++;
13         }
14     }
15 }
```



```java
public static void mergeSort(int [] a) {
    if (a.length >= 2) { // split array into two halves
        int [] left = Arrays.copyOfRange(a, 0, a.length/2);
        int [] right = Arrays.copyOfRange(a, a.length/2, a.length);

        // recursively sort the two halves
        mergeSort(left);
        mergeSort(right);

        // merge the sorted halves into a sorted whole
        merge(a, left, right);
    }
}
```

4.5. Quicksort

- **Complexity:**
  - Worst case: $O(N^2)$
  - Best/average case: $O(N \log N)$
- **In-place**

How does Quicksort work?
Comparison sort based on the **divide-and-conquer strategy**.

1. **Divide:**
   (a) pivot is in its correct position
   (b) no larger element is to the left of the pivot
   (c) no smaller element is to the right of the pivot
2. Combine

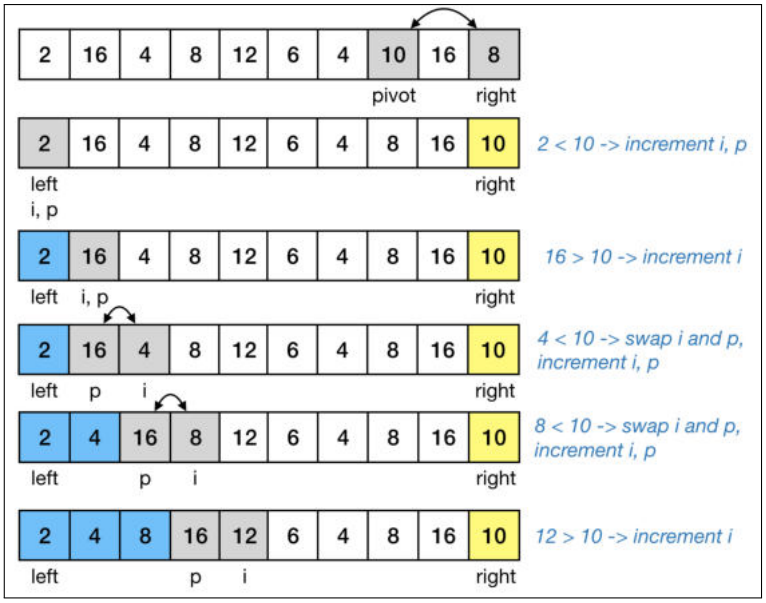
The choice of the pivot determines the size of the partitions.

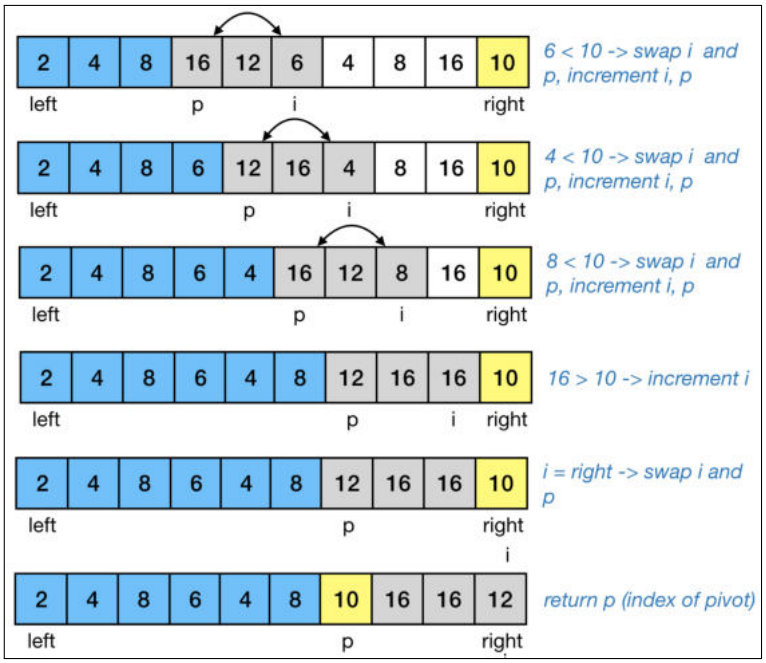
```java
public void qsort(Comparable[] a, int left, int right) {
    // Conquer
    if (right <= left) return;

    // Divide
    int j = partition(a, left, right);
    qsort(a, left, j-1);
    qsort(a, j+1, right);
}
```

The partition method takes the array and the left and right indices of the current portion of the array, and returns the resulting index of the pivot value after the partitioning:

```
1  private int partition(T[] a, int left, int right, int pivotIndex) {
2                  // pivot location is provided as parameter
3      T pivot = a[pivotIndex];
4      swap(a, pivotIndex, right); // move pivot to the end
5      int p = left; // p will become the final index of pivot
6
7      for (int i = left; i < right; i++) {
8          if (less(a[i], pivot)) {
9              swap(a, i, p);
10             p++;
11         }
12     }
13     swap(a, p, right); // move pivot to its correct location
14     return p;
15 }
```





4.5.1. Choosing a pivot value

- **Median of three:** Choose three elements of the array (usually first, middle, last) and then use the median of those three values as the pivot.
- **Shuffle once, pick first element:** Randomize the order of elements in the array once up front (before the D&C part of the sort begins) and then just pick the first element as the pivot each time.

```
1 public void shuffle(T[] a) {
2     Random rng = new Random();
3     for (int i = a.length - 1; i > 0; i--) {
4         int j = rng.nextInt(i + 1);
5         swap(a, i, j);
6     }
7 }
8
```

Lecture 5

Bag

- **Bag:** A collection of elements where there is no particular order and duplicates are allowed; this is essentially what java.util.Collection describes.
- **Refactoring:** The process of changing the structure or "factoring" of existing code without changing its behavior.

```
1  public class RandomList<T> implements RandomizedList<T> {
2      T[] elements;
3      int size; final int DEFAULT_CAPACITY = 5;
4
5      ////////////////////////
6      //// CONSTRUCTOR ////
7      ////////////////////////
8
9      RandomList() {
10         elements = (T[]) new Object[DEAFULT_CAPACITY];
11         size = 0;
12     }
13
14     ////////////////////
15     /////// ADD ////////
16     ////////////////////
17
18     public void add(T element) {
19         if (size == elements.length) {
20             resize(elements.length * 2); // O(N) -> amortized cost -> O(1)
21         }
22         elements[size++] = element;
23     }
24
25     ////////////////////
26     ///// REMOVE //////
27     ////////////////////
28
29     public T remove() {
30         // array is empty
31         if (size == 0) {
32             return null;
33         }
34         int index = search(sample()); // O(N)
35         // save for return
```

```
36          T ret = elements[index];
37          // located so remove
38          elements[index] = elements[size--];
39          elements[size] = null;
40          if (size > 0 && size < elements.length / 4) {
41              resize(elements.length / 2);
42          }
43          return ret;
44      }
45
46      ////////////////////
47      ///// RESIZE //////
48      ////////////////////
49
50      private void resize(int capacity) {
51          T[] copy = (T[]) new Object[capacity];
52          for (int i = 0; i < size; i++) {
53              copy[i] = elements[i];
54          }
55          elements = copy;
56      }
57
58      ////////////////////
59      ///// SEARCH //////
60      ////////////////////
61
62      private int search(T element) {
63          for (int i = 0; i < size; i++) {
64              if (elements[i].equals(element)) {
65                  return i;
66              }
67          }
68          return -1;
69      }
70  }
```

- **Amortized analysis:** We can amortize the cost of expanding the capacity of the array over a sequence of N calls to add; that is, when the array becomes less than 25% full, reduce the capacity by half.
  - Starting an empty bag at capacity 1 and using the dynamic resizing strategies just described allows us to maintain the following invariant: the array must be always between 25% and 100% full.
  - Thus, the amount of memory needed for the array is constant times N, that is, $O(N)$; we can guarantee that our implementation only needs a linear amount of memory.

Lecture 6

Linked Structures

6.1. Singly-linked Bag

6.1.1. Singly-linked Nodes

- **Linked list:** A sequence of elements arranged one after another, with each element connected to the next element by a "link".
- A common programming practice is to place each element together with the link to the next element, resulting in a component called **node**. Each node contains two pieces of information:
  - **Element:** A reference to the value this node stores.
  - **Link:** A reference to the next node in the chain.

```
1 private class SinglyNode<T> {
2     T data;
3     SinglyNode<T> next;
4
5     SinglyNode(T data) {
6         this.data = data;
7         next = null;
8     }
9 }
```

```
1  public class SinglyLinkedBag<T> implements Bag<T> {
2      private Node<T> front;
3      private int size;
4      public SinglyLinkedBag() {
5          front = null;
6          size = 0;
7      }
8      public int size() {
9          return size;
10     }
11     public boolean isEmpty() {
12         return size == 0;
13     }
14     // ...
15 }
```

Adding a new node at the front of the linked list

1. Create a new node.
2. Make the new node's next link refer to the first entry of the list.
3. Make the reference to the first node refer to this node.

```
public void addAtFront(T element) {
    Node<T> newnode = Node<T>(element);
    newnode.next = front;
    front  = newnode;
    size++;
}
```

Removing a node that is NOT at the front

1. Get a reference to the node to be deleted via the reference to the previous node.
2. Delink this node from the linked list.

```
public boolean remove(T element) {
    Node<T> n = new Node<T>(element);
    Node<T> prev = null; // drag behind n
    while (n != null && !(n.element.equals(element))) {
        prev = n;
        n = n.next;
    }
    if (n == null) { // not found
        return false;
    }
    else { // found
        if (n == front) { // sc: delete at the front
            front = front.next;
        }
        else {
            prev.next = n.next;
        }
        size--;
    }
    return true;
}
```

```
public boolean contains(T element) {
    Node<T> temp = front;
```

```
3       while (temp != null) {
4           if (temp.element.equals(element)) {
5               return true;
6           }
7           temp = temp.next;
8       }
9       return false;
10  }
```

6.2. Lists

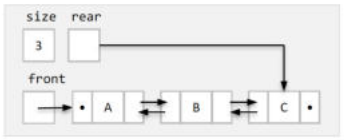
- **List:** A collection that keeps its elements in some particular order.

Array-Based

- Keep elements left-justified (anchored at 0, no gaps)
- Keep a size counter (can serve as a rear marker)

Node-Based

- Doubly-linked
- Keep both a front and rear pointer
- Keep a size counter




```
public class ArrayIndexedList<T> implements IndexedList<T> {

        private T[] elements;
        private int rear;
```

```
public class LinkedIndexedList<T> implements IndexedList<T> {

        private Node front;
        private Node rear;
        private int size;
```

6.2.1. Array List

```
1   public class ArraySet<T extends Comparable<? super T>> implements Set<T> {
2       T[] elements;
3       int size;
4
5       @SuppressWarnings("unchecked")
6       public ArraySet() {
7           elements = (T[]) new Comparable[1];
8           size = 0;
9       }
10
11      /*****************************************************************/
12      /**************************SERVICE METHODS************************/
13      /*****************************************************************/
14
15      public int size() {
16          return size;
```

```
17      }
18
19      public boolean isEmpty() {
20          return size == 0;
21      }
22
23       public boolean add(T element, int index) {
24           if (index == rear) {
25               return add(element);
26           }
27           if (! validIndex(index)) {
28               return false;
29           }
30           if ( isFull ()) {
31               resize(elements.length * 2);
32           }
33           shiftRight(index);
34          elements[index] = element;
35           rear++;
36           return true;
37       }
38
39      public boolean contains(T element) {
40          if (element == null) {
41              return false;
42          }
43          if (isEmpty()) {
44              return false;
45          }
46          int i = locate(element);
47          return elements[i].compareTo(element) == 0;
48      }
49
50      public Iterator<T> iterator() {
51          return new ArraySetIterator();
52      }
53
54      /*****************************************************************/
55      /*************************SUPPORT METHODS*************************/
56      /*****************************************************************/
57
58      private boolean isFull () {
59          return size == elements.length;
60      }
61
62      private void resize(int capacity) {
63          T[] copy = (T[]) new Object[capacity];
64          for (int i = 0; i < size; i++) {
65              copy[i] = elements[i];
```

```
66          }
67          elements = copy;
68      }
69
70      private void shiftRight(int loc) {
71          assert size < elements.length;
72          for (int i = size; i > loc; i--) {
73              elements[i] = elements[i - 1];
74          }
75          elements[loc] = null;
76      }
77
78      private void shiftLeft(int loc) {
79          for (int i = loc; i < size; i++) {
80              elements[i - 1] = elements[i];
81          }
82          elements[size - 1] = null;
83      }
84
85      /*****************************************************************/
86      /*************************NESTED CLASSES**************************/
87      /*****************************************************************/
88
89      private class ArraySetIterator implements Iterator<T> {
90          int current = 0;
91
92          public boolean hasNext() {
93              return current < size;
94          }
95
96          public T next() {
97              if (!hasNext()) {
98                  throw new NoSuchElementException();
99              }
100             return elements[current++];
101         }
102
103         public void remove() { }
104     }
105 }
```

6.2.2. Doubly-linked List

```
1  public class DoubleEndList<T> implements DoubleEndedList<T> {
2      private Node<T> head;
3      private Node<T> tail;
4      private int size;
```

```
5
6   public DoubleEndList() {
7       head = null;
8       tail = null;
9       size = 0;
10  }
11
12  /**************************************************************************/
13  /*************************SERVICE METHODS**************************/
14  /**************************************************************************/
15
16  public void addFirst(T element) {
17      if (element == null) {
18          throw new IllegalArgumentException("element is null");
19      }
20      Node<T> newnode = new Node<T>(element);
21      if (isEmpty()) { // list is empty
22          tail = newnode;
23      }
24      else {
25          head.prev = newnode;
26      }
27      newnode.next = head;
28      head = newnode;
29      size++;
30  }
31
32  public void addLast(T element) {
33      if (element == null) {
34          throw new IllegalArgumentException("element is null");
35      }
36      Node<T> newnode = new Node<T>(element);
37      if (isEmpty()) { // list is empty
38          head = newnode;
39      }
40      else {
41          tail.next = newnode;
42          newnode.prev = tail;
43      }
44      tail = newnode;
45      size++;
46  }
47
48  public T removeFirst() {
49      if (isEmpty()) { // list is empty
50          return null;
51      }
52      Node<T> oldnode = head;
53      if (head == tail) { // one entry
```

```
54            tail = null;
55        }
56        else {
57            head.next.prev = null;
58        }
59        head = head.next;
60        oldnode.next = null;
61        size--;
62        return oldnode.data;
63    }
64
65    public T removeLast() {
66        if (isEmpty()) { // list is empty
67            return null;
68        }
69        Node<T> oldnode = tail;
70        if (head == tail) { // one entry
71            head = null;
72        }
73        else {
74            tail.prev.next = null;
75        }
76        tail = tail.prev;
77        oldnode.prev = null;
78        size--;
79        return oldnode.data;
80    }
81
82    public int size() {
83        return size;
84    }
85
86    public boolean isEmpty() {
87        return size == 0;
88    }
89
90    public Iterator<T> iterator() {
91        return new LinkedIterator();
92    }
93
94    /*****************************************************************************/
95    /*************************NESTED CLASSES**************************************/
96    /*****************************************************************************/
97
98    private class Node<T> {
99        private T data;
100       private Node<T> next;
101       private Node<T> prev;
102
```

```
103         Node(T data) {
104             this.data = data;
105             next = null;
106             prev = null;
107         }
108     }
109 
110     private class LinkedIterator implements Iterator<T> {
111         private Node<T> curr = head;
112 
113         public boolean hasNext() {
114             return curr != null;
115         }
116 
117         public T next() {
118             if (!hasNext()) {
119                 throw new NoSuchElementException("no elememts left to process");
120             }
121             T ret = curr.data;
122             curr = curr.next;
123             return ret;
124         }
125 
126         public void remove() {
127             throw new UnsupportedOperationException();
128         }
129     }
130 }
```

6.2.3. Summary

	Bag				**Set**			
	Array		**Sinly-linked**		**Array**		**Singly-linked**	
	Ordered	**Unordered**	**Ordered**	**Unordered**	**Ordered**	**Unordered**	**Ordered**	**Unordered**
add	$O(N)$	$O(1)$	$O(N)$	$O(1)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$
remove	$O(N)$	$O(N)$	$OO(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$
contains	$O(\log N)$	$O(N)$	$O(N)$	$O(N)$	$O(\log N)$	$O(N)$	$O(N)$	$O(N)$

Lecture 7

Stack, Queue, BFS and DFS

7.1. Definitions

- **Queue:** A collection that maintains its elements in the *first-in, first-out* (FIFO) order.

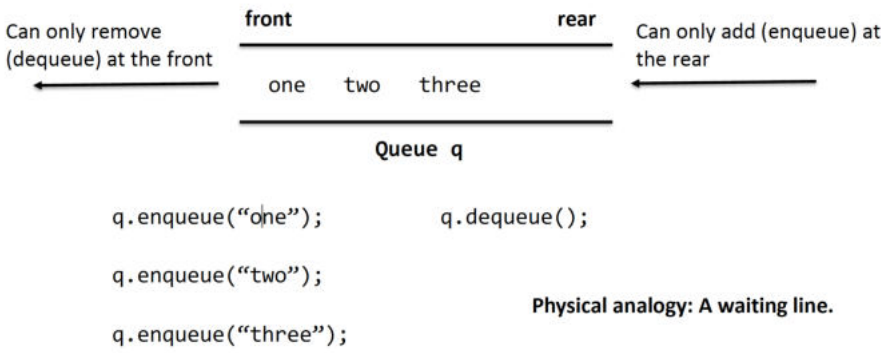


Figure 1.

- **Stack:** A collection that maintains its elements in the *last0in, first-out* (LIFO) order.

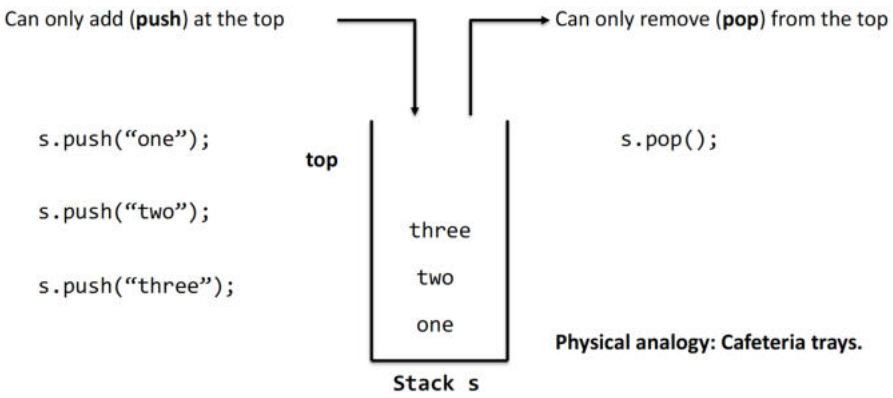


Figure 2.

7.2. Implementation alternatives

7.2.1. Array-based stack

```
1 public class ArrayStack<T> implements Stack<T> {
2     // holds the values in the stack
```

```
3       private T[] elements;
4       // index of the next insertion point
5       private int top;
6
7       @SuppressWarnings("unchecked")
8       public ArrayStack(int capacity) {
9           elements = (T[]) new Object[capacity];
10          top = 0;
11      }
12
13      public int size() {
14          return top;
15      }
16
17      public boolean isEmpty() {
18          return size() == 0;
19      }
20
21      public void push(T element) {
22          if (size() == elements.length) {
23              resize(elements.length * 2);
24          }
25          elements[top++] = element;
26      }
27
28      public T pop() {
29          if (isEmpty()) {
30              return null;
31          }
32          T topValue = elements[--top];
33          elements[top] = null;
34          return topValue;
35      }
36
37      public T peek() {
38          if (isEmpty()) {
39              return null;
40          }
41          return elements[top - 1];
42      }
43  }
```

Time complexity: $O(1)$ for all methods.

7.2.2. Node-based. implementation

```
1  public class LinkedStack<T> implements Stack<T> {
2      // reference to the top of the stack
3      private Node top;
4      // number of values in the stack
5      private int size;
6
7      /** Create an instance of the LinkedStack class. */
8      public LinkedStack() {
9          top = null;
10         size = 0;
11     }
12
13     public int size() {
14     return size;
15     }
16
17     public boolean isEmpty() {
18         return size() == 0;
19     }
20
21     public void push(T element) {
22         top = new Node(element, top);
23         size++;
24     }
25
26     public T pop() {
27         if (isEmpty()) {
28             return null;
29         }
30         T topValue = top.element;
31         top = top.next;
32         size--;
33         return topValue;
34     }
35
36     public T peek() {
37         if (isEmpty()) {
38             return null;
39         }
40         return top.element;
41     }
42 }
```

Time complexity: $O(1)$ for all methods.

7.2.3. Node-based queue

```
1  public class LinkedQueue<T> implements Queue<T> {
2      // the front node
3      private Node front;
4      // the rear node;
5      private Node rear;
6      // the number of elements in the queue
7      private int size;
8
9      /** Create an instance of the LinkedQueue class. */
10     public LinkedQueue() {
11         front = null;
12         rear = null;
13         size = 0;
14     }
15
16     public int size() {
17         return size;
18     }
19
20     public boolean isEmpty() {
21         return size() == 0;
22     }
23
24     public void enqueue(T element) {
25         Node n = new Node(element, null);
26         if (isEmpty()) {
27             rear = n;
28             front = rear;
29         }
30         else {
31             rear.next = n;
32             rear = n;
33         }
34         size++;
35     }
36
37     public T dequeue() {
38         if (isEmpty()) {
39             return null;
40         }
41         T result = front.element;
42         front = front.next;
43         if (size == 1) {
44             rear = front;
45         }
46         size--;
```

```
47          return result;
48      }
49
50      public T first() {
51          if (isEmpty()) {
52              return null;
53          }
54          return front.element;
55      }
56  }
```

Time complexity: $O(1)$ for all methods.

7.2.4. Array-based queue

- The array-management strategy of "left justified with no gaps" no longer works well for the queue collection; by anchoring one end of the queue at index 0, shifting would be required in either dequeue or enqueue.
- So, the time complexity of our implementation would be one of the following:
  - enqueue $O(1)$, dequeue $O(N)$
  - enqueue $O(N)$, dequeue $O(1)$
- We need to use a **circular array strategy** to make both enqueue and dequeue $O(1)$.
  - We keep the elements of the array contiguous, but allow them to float down toward the right and then wrap around once they hit the end.

```
1   public class ArrayQueue<T> implements Queue<T> {
2       // stores the elements in the queue
3       private T[] elements;
4       // index of the front element
5       private int front;
6       // index of the next insertion point
7       private int rear;
8       // number of elements in the queue
9       private int size;
10
11      @SuppressWarnings("unchecked")
12      public ArrayQueue(int capacity) {
13          elements = (T[]) new Object[capacity];
14          front = 0;
15          rear = 0;
16          size = 0;
17      }
```

```
18
19      public boolean isEmpty() {
20          return size() == 0;
21      }
22
23      public int size() {
24          return size;
25      }
26
27      public void enqueue(T element) {
28          if (size() == elements.length) {
29              resize(elements.length * 2);
30          }
31          elements[rear] = element;
32          rear = (rear + 1) % elements.length;
33          size++;
34      }
35
36      public T dequeue() {
37          if (isEmpty()) {
38              return null;
39          }
40          T result = elements[front];
41          elements[front] = null;
42          front = (front + 1) % elements.length;
43          size--;
44          return result;
45      }
46
47      public T first() {
48          if (isEmpty()) {
49              return null;
50          }
51          return elements[front];
52      }
53  }
```

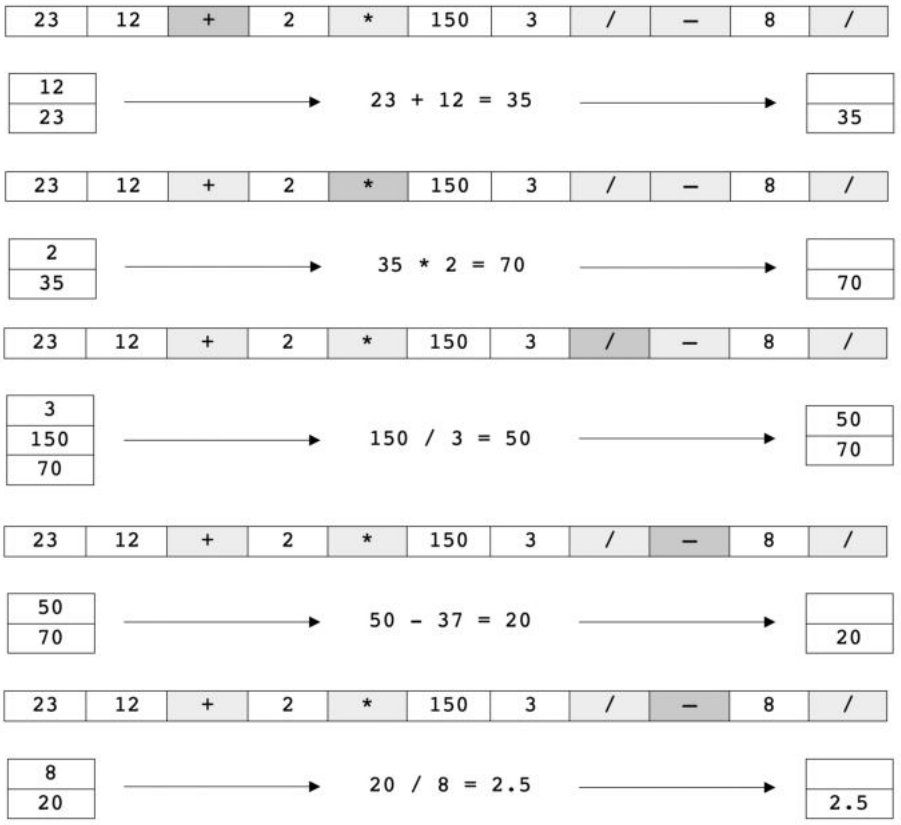
Time complexity: $O(1)$ for all methods.

7.3. Stack Machine Evaluation (SME)

7.3.1. Postfix

- Consider the following:

```
23 12 + 2 * 150 3 / - 8 /
```



Postfix (hand-and-paper approach)
The expression is scanned **left to right**:

1. When an operand is encountered, it is pushed on the stack.
2. When an operator is encountered, the top two entries on the stack are popped, the operation is performed, and the result of the operation is pushed back on the stack.
3. The evaluation terminates when the expression has been scanned, at which point the lone entry on the stack is the result.

```
1  Algorithm postfixEvaluation
2  expr: input postfix expression (from the left)
3  tok: first token of expr
4  while expr hasNext do
5      if (tok is an operator) then
6          topFirst = pop stack
7          topNext = pop stack
8          result = apply tok on topNext and topFirst // ex.: topNext - topFirst
9          push result on stack
10     else
11         val = tok str converted to int
12         push val on stack
```

```
    endif
    tok = next tok of expr
endwhile
result = pop stack
```

7.3.2. Prefix

Prefix (hand-and-paper approach)
The expression is scanned **right to left**:

1. When an operand is encountered, it is pushed on the stack.
2. When an operator is encountered, the top two entries on the stack are popped, the operation is performed, and the result of the operation is pushed back on the stack.
3. The evaluation terminates when the expression has been scanned, at which point the lone entry on the stack is the result.

```
Algorithm prefixEvaluation
expr: input prefix expression
tok: first token of expr (from the right)
while expr hasNext do
    if (tok is an operator) then
        topFirst = pop stack
        topNext = pop stack
        result = apply tok on topFirst and topNext // ex.: topFirst - topNext
        push result on stack
    else
        val = tok str converted to int
        push val on stack
    endif
    tok = next tok of expr
endwhile
result = pop stack
```

7.4. Breadth-First Search (BFS) and Depth-First Search (DFS)

- We will represent the search space with a two-dimensional array.

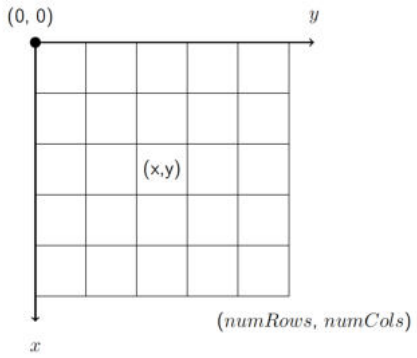


Figure 3.

- **We will just use BFS and DFS as systematic ways to explore every position in the 2d array.** So, we're treating BFS and DFS as *traversal methods* rather than search strategies.

7.4.1. BFS

Breadth-first search: Explore the immediate neighborhood *before* going deeper.

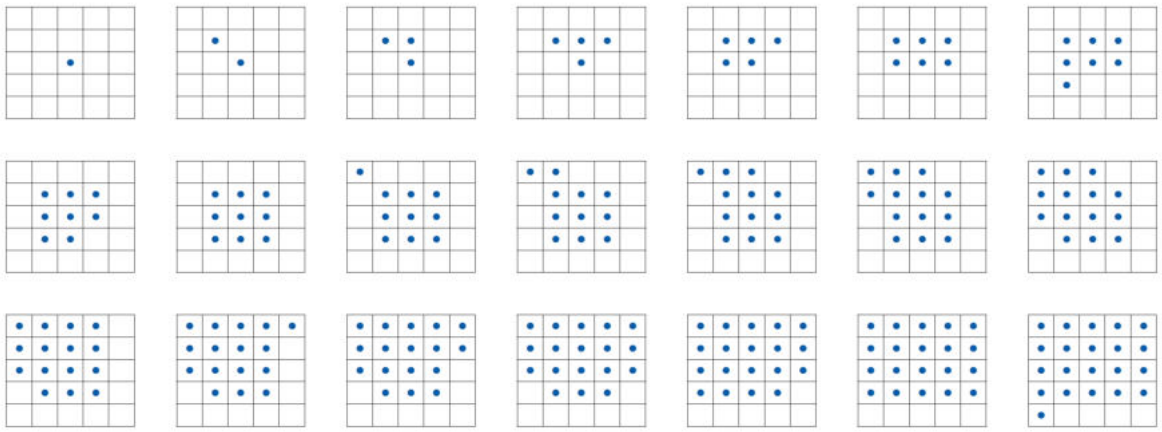


Figure 4.

- **Ex:** Breadth-first search starting at (3,2):

26	27	28	29	30
10	11	12	15	16
13	2	3	4	17
14	5	1	6	18
19	7	8	9	20
21	22	23	24	25

- **Ex:** Breadth-first search starting at (0,0):

1	2	5	10	17
3	4	6	11	18
7	8	9	12	19
13	14	15	16	20
21	22	23	24	25
26	27	28	29	30

- **Ex:** Breadth-first search starting at (5,4):

26	27	28	29	30
17	18	19	22	23
20	10	11	12	15
21	13	5	6	7
24	14	8	2	3
25	16	9	4	1

DFS

Depth-first search: Explore as deeply as possible *before* backing up.

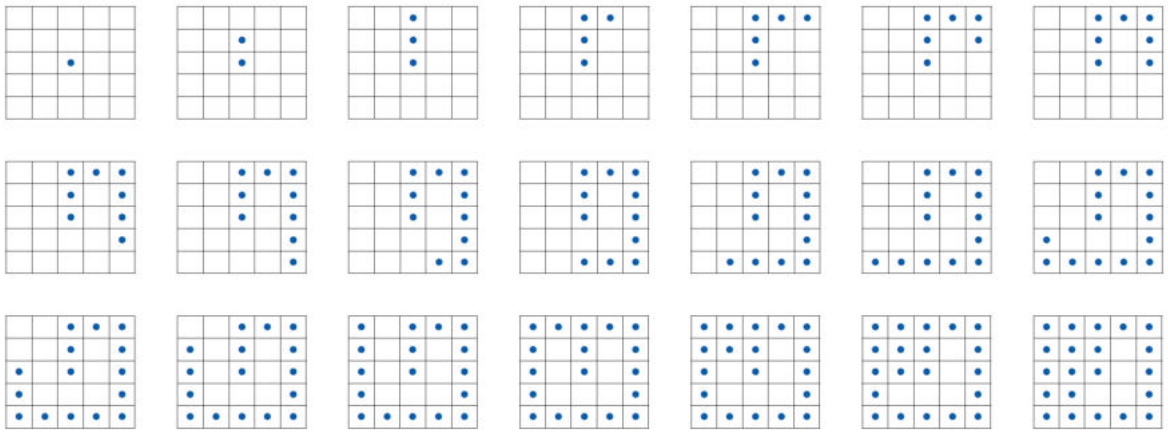


Figure 5.

- **Ex:** Depth-first search starting at (3,2):

20	19	18	17	16
21	11	12	13	15
10	30	29	28	14
9	27	1	26	23
8	25	24	2	22
7	6	5	4	3

- **Ex:** Depth-first search starting at (0,0):

1	30	28	20	19
29	2	27	24	18
26	25	3	23	17
12	22	21	4	16
11	13	15	14	5
10	9	8	7	6

- **Ex:** Depth-first search starting at (5,4):

18	19	20	14	13
17	16	15	21	12
25	24	23	22	11
26	7	8	9	10
6	27	28	30	29
5	4	3	2	1

Lecture 8

Trees

8.1. Tree Terminology

- **Order:** an integer ≥ 2 that represents the upper limit on the number of children that any node can have.
- **Path:** a sequence of nodes from one node to another, going from parent to child.
- **Path length:** number of nodes on the path.
- **Height:** the path length from a given node to its lowest leaf. *When height is applied to a tree, it refers to the height of its root.*
- **Depth:** the length of path from the root of the tree to a given node.

8.2. Types of Binary Trees

- **Full:** all the leaves are at the same depth.
- **Complete:** it is full to the next-to-last level, and the leaves on the lowest level are "left justified".

Max number of nodes at height h

$$N = 2^h - 1 \tag{8.1}$$

where h is the height if a binary tree.

Max height of a binary tree with N nodes

$$h = \lfloor \log_2 N \rfloor + 1 \tag{8.2}$$

where N is the number of nodes in a binary tree.

8.3. Traversing in a tree

- **Inorder:** recursively traverse *Left* subtree of T, *Visit* the root of T, recursively traverse *Right* subtree of T.
  - **LNR**
- **Preorder:** *Visit* the root of T, recursively traverse *Left* subtree of T, recursively traverse *Right* subtree of T.

  - **NLR**

- **Postorder:** recursively traverse *Left* subtree of T, recursively traverse *Right* subtree of T, *Visit* the root of T.

  - **LRN**

Lecture 9

Binary Search Tree

Binary search tree (BST): A binary tree in which the search property holds in every node.

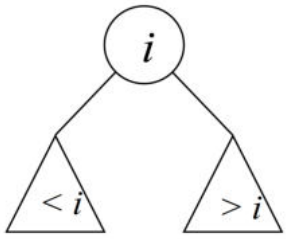


Figure 1.

9.1. Adding values

The newly inserted node always becomes a leaf node in the search tree.

9.2. Removing values

- **Case 1: Leaf**
  - Set its parent's pointer to null.
- **Case 2: A node has one child**
  1. Set its parent's pointer to its child.
  2. Set this node's pointer to null.
- **Case 3: A node has two children**
  1. Find its inorder predecessor/successor Y.
  2. Copy the entry of Y into this node.
  3. *Delete* Y.

9.3. Time Complexity

The number of comparisons to find a given value is equal to the depth of the node that contains it.

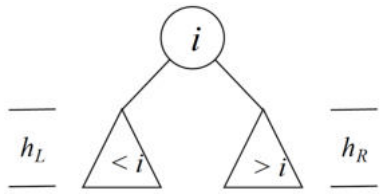
Searching, adding, and removing in a BST is $O(height)$:

- **Worst-case:** $O(N)$ (tall and narrow);
- **Best-case:** $O(\log N)$ (short and wide).

Lecture 10

AVL Tree

AVL tree: a binary search tree in which the heights of the left and the right subtree of every node differ at most by 1.



$$|h_R - h_L| \leq 1$$

Figure 1.

Every node in AVL has a **balance factor**.

$$bf_N = h_R - h_L \tag{10.1}$$

10.1. Rebalancing

- **Case 1:** bf(Root) = **2**, bf(Right) = **1**
  - Rotate left around Right.
    - If Right has a left child, it becomes the right child of Root.
- **Case 2:** bf(Root) = **−2**, bf(Left) = **−1**
  - Rotate right around Left.
    - If Left has a right child, it becomes the left child of Root.
- **Case 3:** bf(Root) = **2**, bf(Right) = **−1**
  1. Rotate right about Right's **left** child.
  2. Rotate left about Right's **left** child.
- **Case 4:** bf(Root) = **−2**, bf(Left) = **1**
  1. Rotate left about Left's **right** child.
  2. Rotate right about Right's **right** child.

10.2. Adding values

1. Add a new node to the last level so it becomes a leaf.
2. Go from this newly inserted node o the root; stop at first unbalanced node, rebalance.

At most one rebalancing operation will be required per insertion.

10.3. Removing values

1. Perform BST deletion (look up special cases).
2. Stop at first unbalanced node, rebalance.

Multiple rebalancing operations may be required per deletion, so the reverse walk must go to the root each time.

Lecture 11

Red-Black Trees

Red-black tree: A binary search tree with the following node colors.

- **Rule 1:** Each node is either red or black.
- **Rule 2:** The root is black.
- **Rule 3:** All paths from the root to an empty tree contain the same number of black nodes (**black height**).
- **Rule 4:** A red node can't have a red child.

11.1. Adding values

What color do we make a new node? **Red.** Let

- N: he first red node (from the bottom) with a red parent;
- G: Grandparent of N;
- P: Parent of N;
- A: Uncle of N;

Then,

- **Case 1: A (uncle) is red**
  - Recolor G, P, A.
- **Case 2: P is right child of G, N is left child of P**
  1. Rotate right around P.
  2. Go to Case 4.
- **Case 3: P is left child of G, N is right child of P**
  1. Rotate left around P.
  2. Go to Case 5.
- **Case 4: P is right child of G, N is right child of P**
  1. Color flip G, P.
  2. Rotate left around G.
- **Case 5: P is left child of G, N is left child of P**
  1. Color flip G, P.
  2. Rotate right around G.

11.2. Summary

	BST	**AVL/RBT**
add	$O(N)$	$O(\log N)$
remove	$O(N)$	$O(\log N)$
search	$O(N)$	$O(\log N)$

Lecture 12

2-4 Trees

Each internal node in a 4-way tree:

- $2 \leq n \leq 4$, where n is the number of children;
- contains max $n - 1$ items;
- all external nodes have the same depth;
- height is $O(\log N)$.

12.1. Adding values

New values are always added in the context of existing leaf nodes. If an insertion makes a node full, we must perform a **split** operation:

1. Take the value in the (initial) middle and send it to he root.
2. The values on the left and on he right of this value split respectively.

Lecture 13

Heap

Binary heap: a complete binary tree of height $O(\log N)$ in which each node obeys a partial order property.

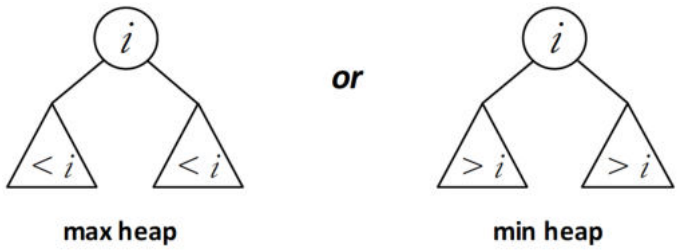


Figure 1.

13.1. Array-based implementation

1. Store the root at index 0.
2. For a node stored at index i, store its left child at index $2i + 1$ and its right child at index $2i + 2$. Thus, the parent of a node stored at index i will be at index $(i - 1)/2$.

13.2. Adding values

1. The newly inserted element becomes the leftmost leaf.
2. Swap values if necessary on the leaf-o-root path to maintain the partial order.

13.3. Removing values

1. Maintain the complete shape by replacing the root value with the value in the lowest, right-most leaf. Then delete that leaf.
2. Swap values as necessary on a root-to-leaf path to maintain the partial order.

13.4. Heapsort

- **Phase 1:** Transform an arbitrary array to a heap
    1. Transform an array onto a binary heap.
    2. "Heapify" each subtree rooted at the same level starting from the bottom.

  - *Heapify:* switch the element in the root with the greater/lesser element in its subtree (switch other elements, if necessary, o maintain partial order).

- *__Phase 2:__ Transform heal into a sorted array.

Lecture 14

Huffman's Algorithm

- Binary trees in which the leaves contain the characters to be coded.
- Interior nodes are just place-holders.
- The root of every subtree is annotated with **cumulative frequency** of all its descendant leaves.
- Character codes are generated by root to leaf traversals:
  - Left branch = 0
  - Right branch = 1

Create a single-node tree for each character and insert each of these trees in a priority queue:

```
while queue has more than one element do {
    c1 = queue.deletemin(); // first min
    c2 = queue.deletemin(); // second min
    c3 = new tree(c1, c2);  // form a new tree with
                            // c1 as a left subtree
                            // c2 as a right subtree
    queue.add(c3);
}
```

Lecture 15

Hash Tables

The Hash Function

$$h(hashcode) = hashcode\%M$$

where

- $hashcode$ = input key value converted into an int;
- M = number of elements in a table (= table.length);

Two broad categories of collision resolution:

1. **Open addressing:** find another index; note that an index contains at most one element reference.
2. **Closed addressing:** store all elements outside the hash table (linked list).

15.1. Close addressing

- Store all elements outside the hash table in linked lists. Each index in the table points to a linked list that stores all elements that hash to that index.
- The time required for add, remove, and the search methods is **proportional to the length of the collision chains**.

15.2. Open addressing

Formula to resolve the collision

$$h(hashcode) = (index + i * C)\%M$$

where

- $index$ = index where the collusion occurred;
- $i =$ of probe attempts;
- C = constant multiple (depends on the probing strategy);
- $M =$ of elements in the table (= table.length);

Linear Probing $(C = 1)$

$$h(hashcode) = (index + i)\%M$$

Quadratic Probing $(C = i)$

$$h(hashcode) = (index + i^2)\%M$$

Double Probing $(C = h_2)$

$$h(hashcode) = (index + i * h_2)\%M$$

where

$$h_2(hascode) = 1 + (hashcode\%(M - 1))$$

15.2.1. Uniform hashing

Load factor

$$\lambda = \frac{N}{m}$$

where

- N = number of elements;
- M = capacity of the table (= table.length);

Note that in an empty table: $\lambda = 0$, full table: $\lambda = 1$.

- **Simple hashing assumption:** for a hash table with M indexes and N keys, each index has to within a small constant factor N/M keys hashed to it, i.e., the size of each "collision chain" is approximately N/M.
- **Being able to attain $O(1)$ time complexity depends on the simple uniform hashing assumption.**

Lecture 16

Graphs

16.1. DFS

> Explore the graph by looking for new vertices far away from the start vertex, and examining only nearer vertices only when dead ends encountered.

```
dfs(Vertex v) {
    visit (v)
    mark v as visited
    for each w adjacent to v
        if w is not visited {
            dfs(w)
        }
}
```

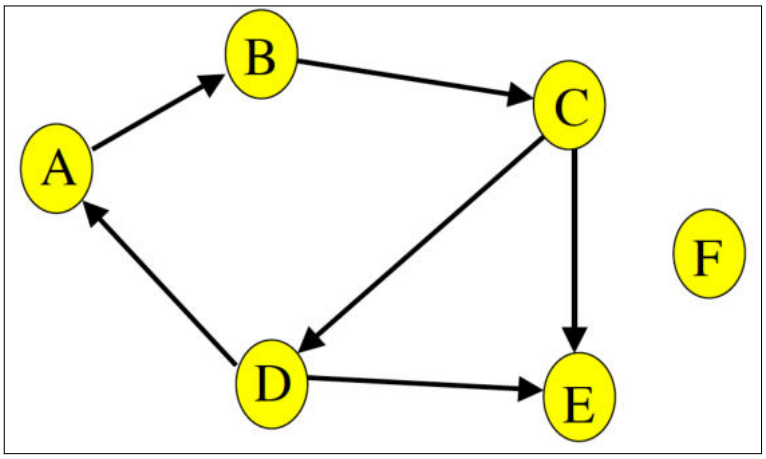
16.2. BFS

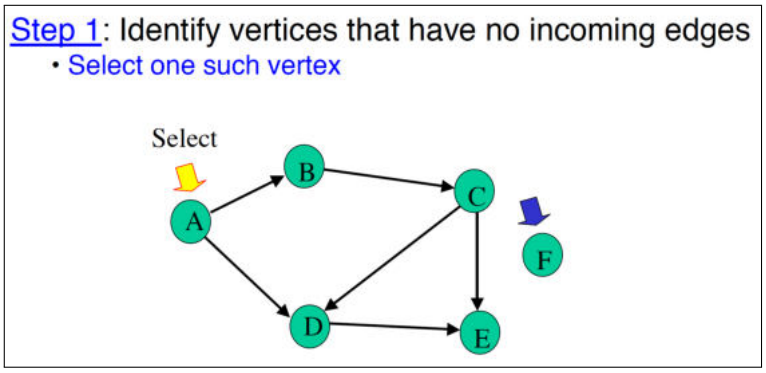
> Starts at some arbitrary vertex of a graph and explores the neighbourhooding vertices first, before moving to the next layer of neighbours.

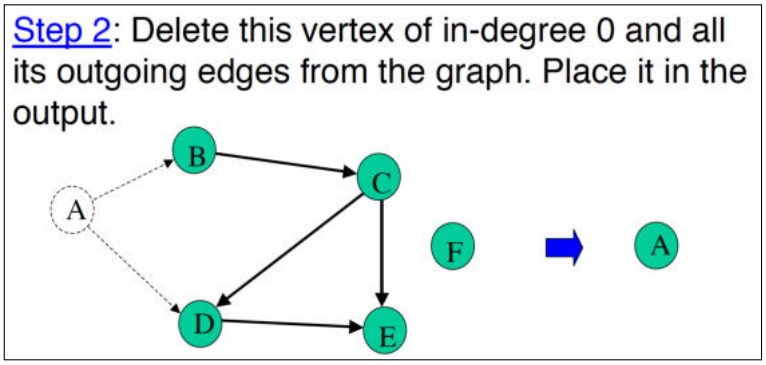
```
bfs(Vertex v) {
    visit (v)
    mark v as visited
    queue.add(v)
    while (!queue.isEmpty()) {
        w = queue.remove()
        for each p adjacent to w {
            if notVisited(p) {
                visit (p)
                mark p as visited
                queue.add(p)
            }
        }
    }
}
```

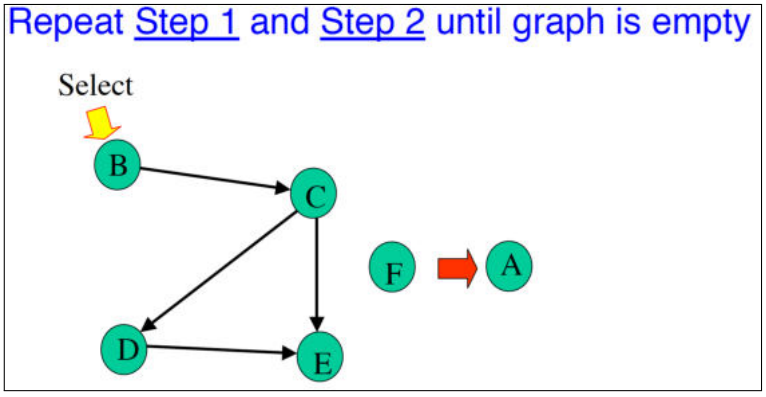
16.3. Topological Sort

Pre: A graph is directed and acyclic.





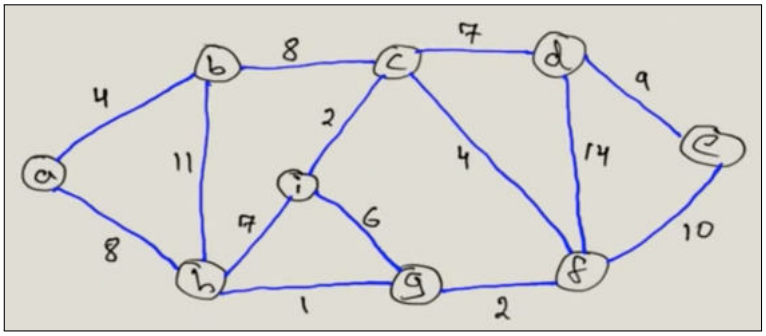


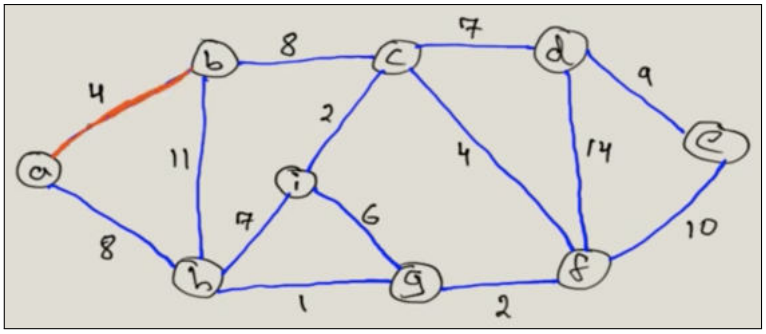


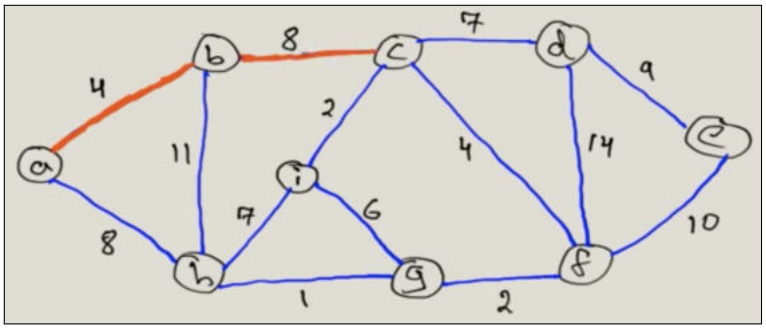
16.4. Prim's Algorithm

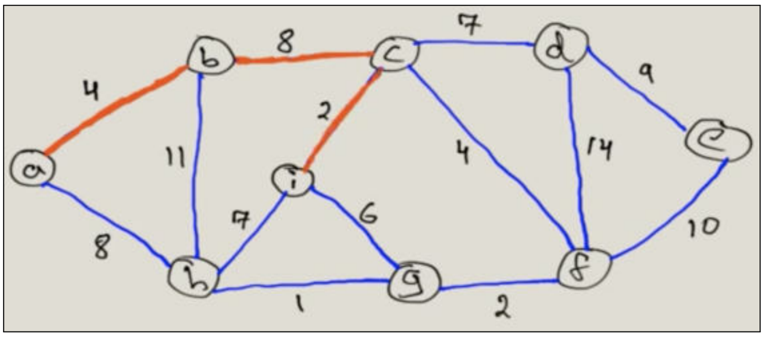
Pre: A graph is connected, weighted, and undirected.

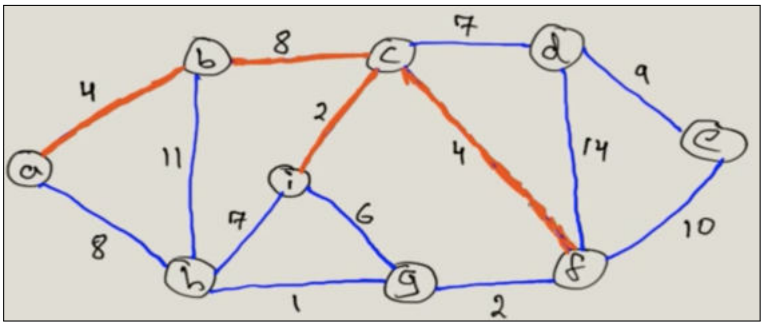
```
1 Initialize the MST as an empty graph.
2 Arbitrarily select a vertex and add it to the MST.
3 Add all the edges in the graph to a collection E.
4 While (there are vertices not in the MST) {
5     Select the edge of minimum weight that connects a vertex
6             in the MST to a vertex that is not in the MST.
7     Add this edge and new vertex to the MST.
8 }
```

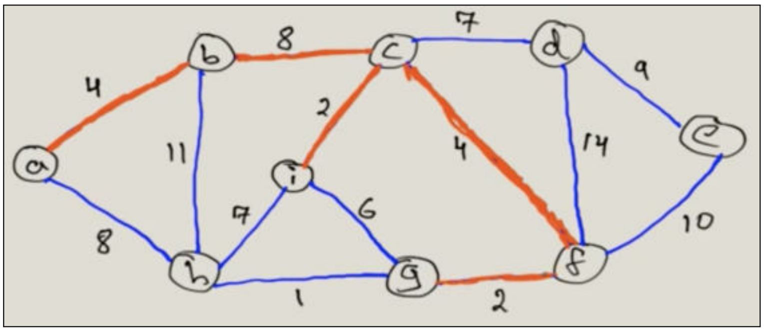


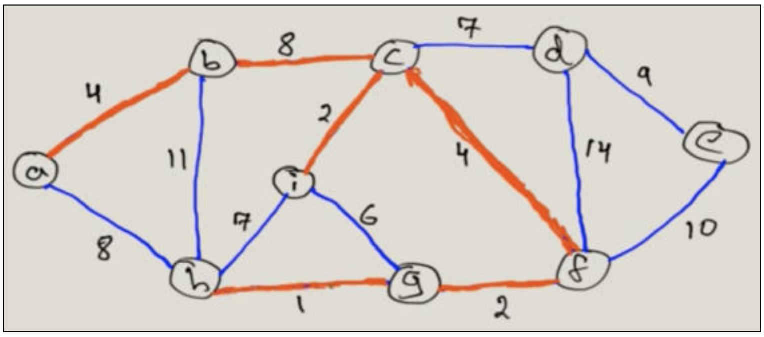


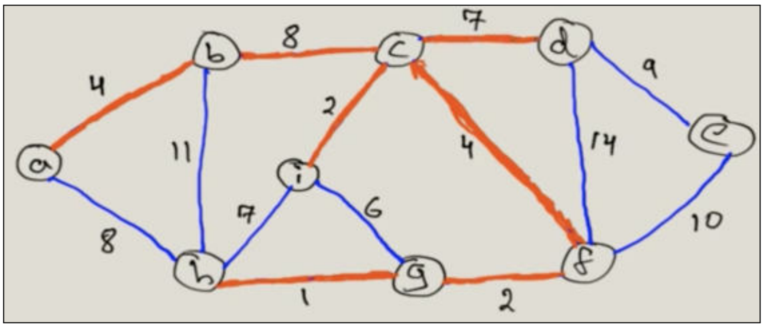


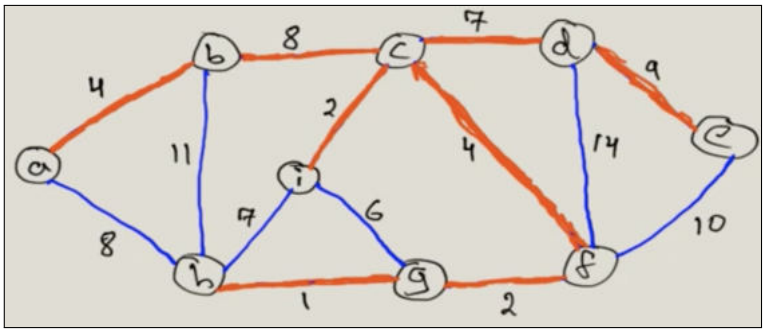

a
b
c
d
e
f
g
h
i
4
8
7
9
2
11
4
14
7
6
10
1
2


a
b
c
d
e
f
g
h
i
4
8
7
9
2
11
4
14
7
6
10
1
2


a
b
c
d
e
f
g
h
i
4
8
7
9
2
11
4
14
7
6
10
1
2


a
b
c
d
e
f
g
h
i
4
8
7
9
2
11
4
14
7
6
10
1
2

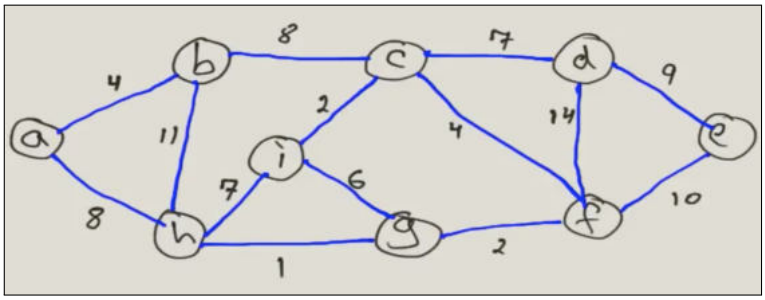

a
b
c
d
e
f
g
h
i
4
8
7
9
2
11
4
14
7
6
10
1
2

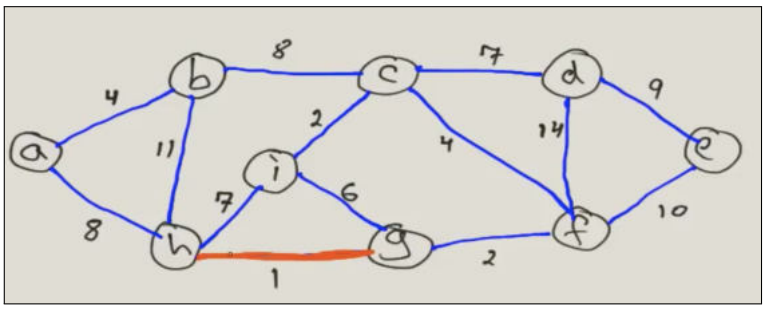


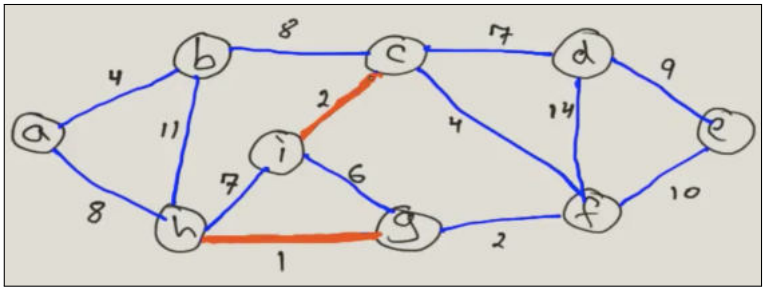
16.5. Kruskal's Algorithm

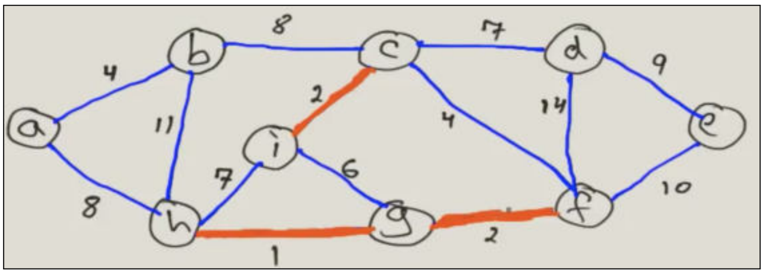
Pre: A graph is connected, weighted, and undirected.

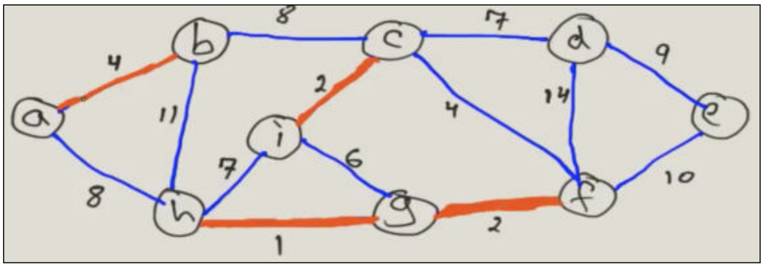
1. Select the edges in ascending order of weight.
2. Add each edge to the MST unless it would create a cycle.
3. Stop when n-1 edges have been added.

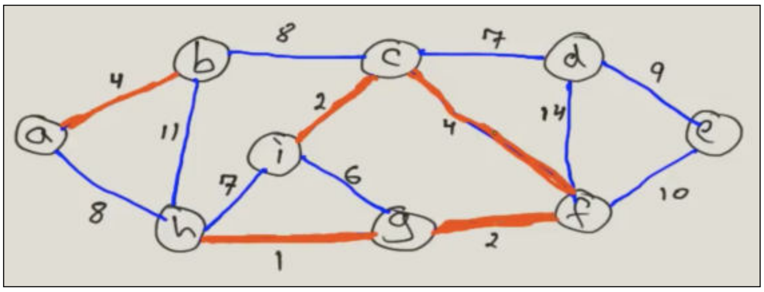


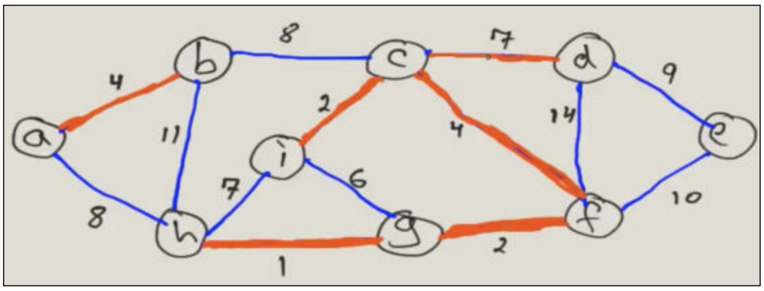


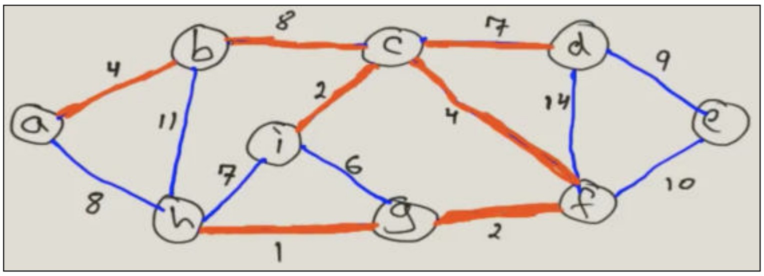


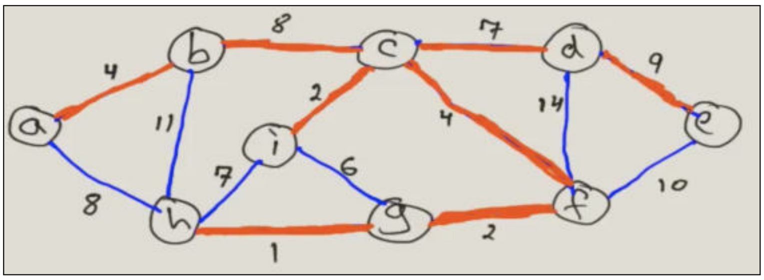

a
b
c
d
e
f
g
h
i
4
8
7
9
11
2
4
14
7
6
10
8
1
2


a
b
c
d
e
f
g
h
i
4
8
7
9
11
2
4
14
7
6
10
8
1
2


a
b
c
d
e
f
g
h
i
4
8
7
9
11
2
4
14
7
6
10
8
1
2


a
b
c
d
e
f
g
h
i
4
8
7
9
11
2
4
14
7
6
10
8
1
2

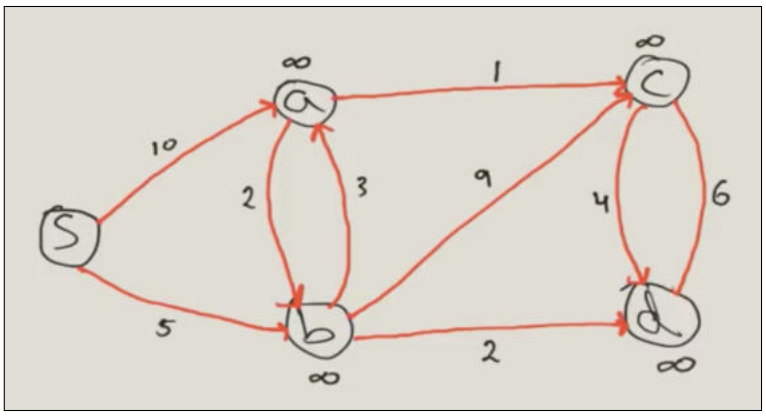

a
b
c
d
e
f
g
h
i
4
8
7
9
11
2
4
14
7
6
10
8
1
2

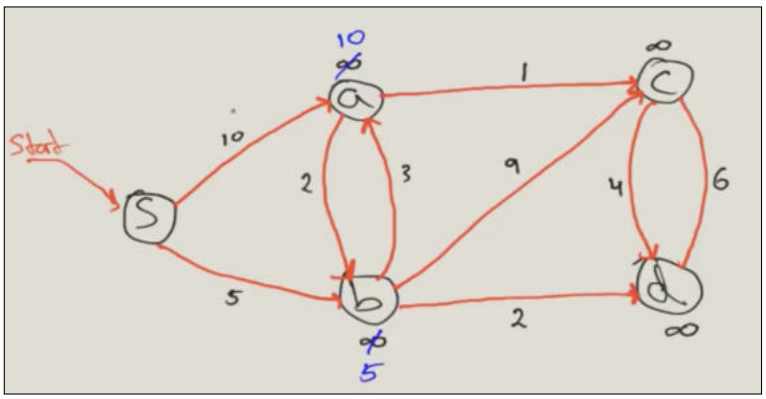

a
b
c
d
e
f
g
h
i
4
8
7
9
11
2
4
14
7
6
10
8
1
2

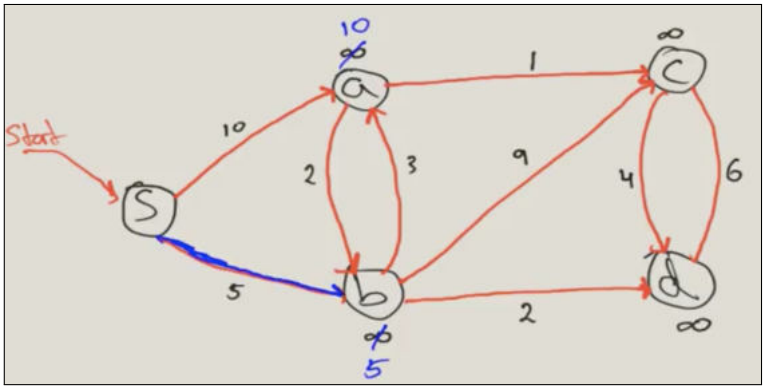
16.6. Dijkstra's Algorithm

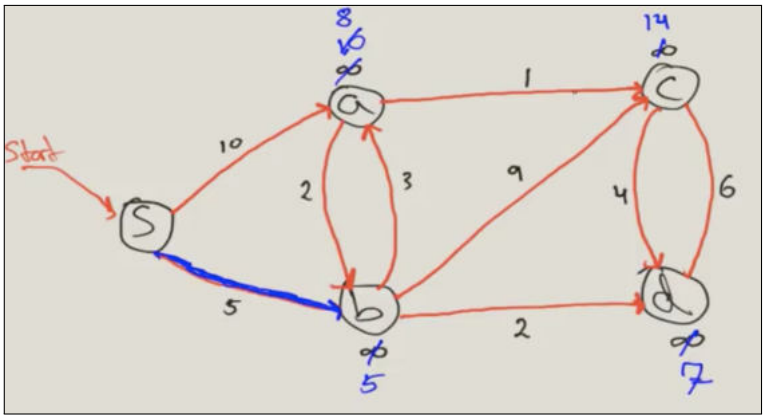
Least-cost path: a weighted, directed graph with non-negative edge weights is a path from vertex A to vertex B such that the cumulative cost of the edge weights is at least as small as the cumulative cost of any other path from A to B.

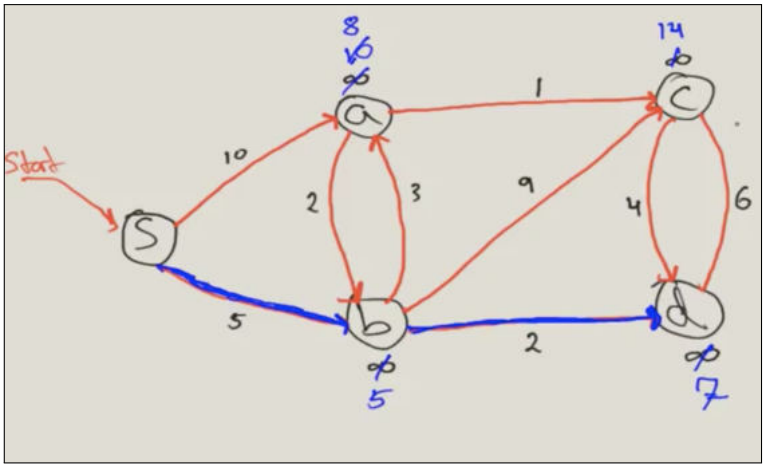
1. Initially, start node has cost 0 and all other nodes have cost ∞.
2. At each step:
   (a) Pick closest unknown vertex v
   (b) Add it to the "cloud" of known vertices
   (c) Update distances for nodes with edges from v

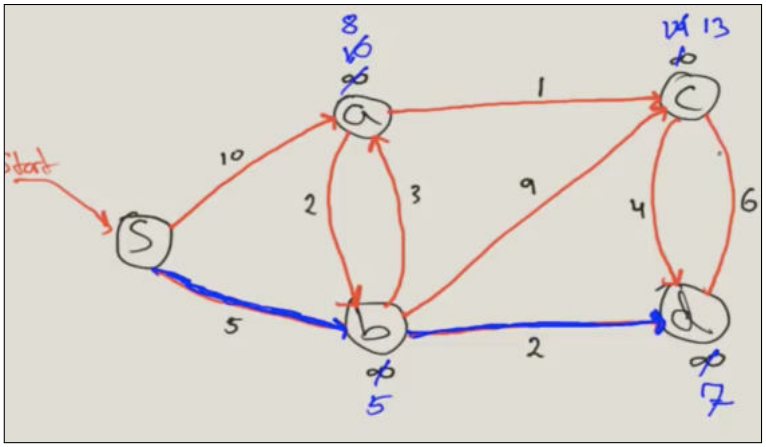


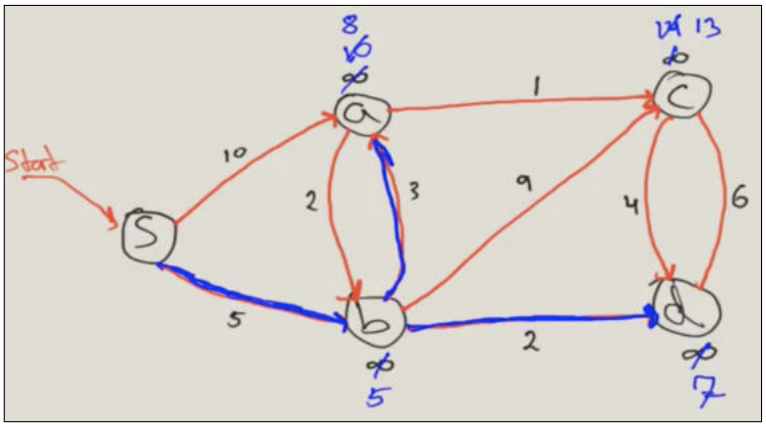


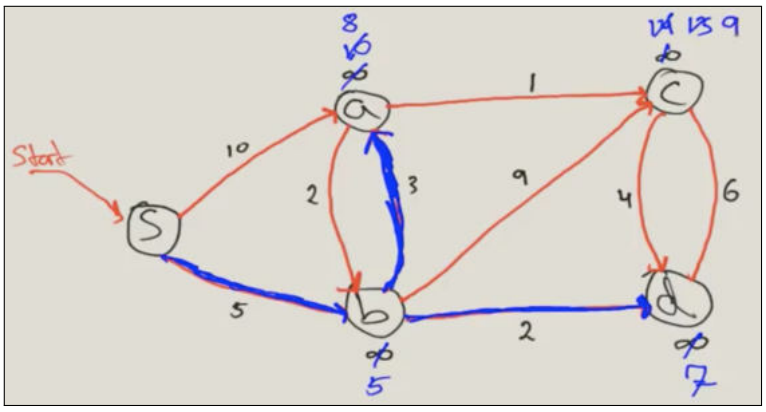


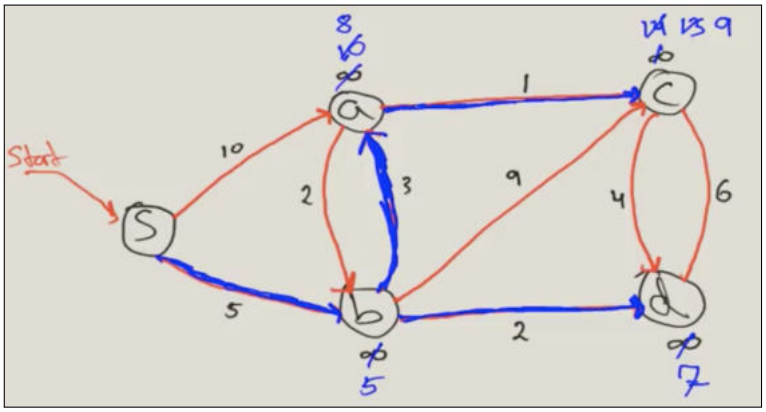

8
14
Start
10
1
2
3
9
4
6
S
a
c
5
b
2
d
5
7


8
14
Start
10
1
2
3
9
4
6
S
a
c
5
b
2
d
5
7


8
13
Start
10
1
2
3
9
4
6
S
a
c
5
b
2
d
5
7


Start
S
a
b
c
d
10
2
3
1
9
4
6
5
2
8
13
5
7


Start
S
a
b
c
d
10
2
3
1
9
4
6
5
2
8
13
9
5
7


Start
S
a
b
c
d
10
2
3
1
9
4
6
5
2
8
13
9
5
7