Testing dan Implementasi Sistem

Lukman Hakim dan Suwanto R

Pertemuan ke 3:White Box Testing

White Box Testing

Dikenal juga dengan nama *glass box*, *structural*, *clear box* dan *open box testing*. Merupakan teknik testing perangkat lunak yang harus mengetahui secara detail tetang perangkat lunak yang akan di uji.

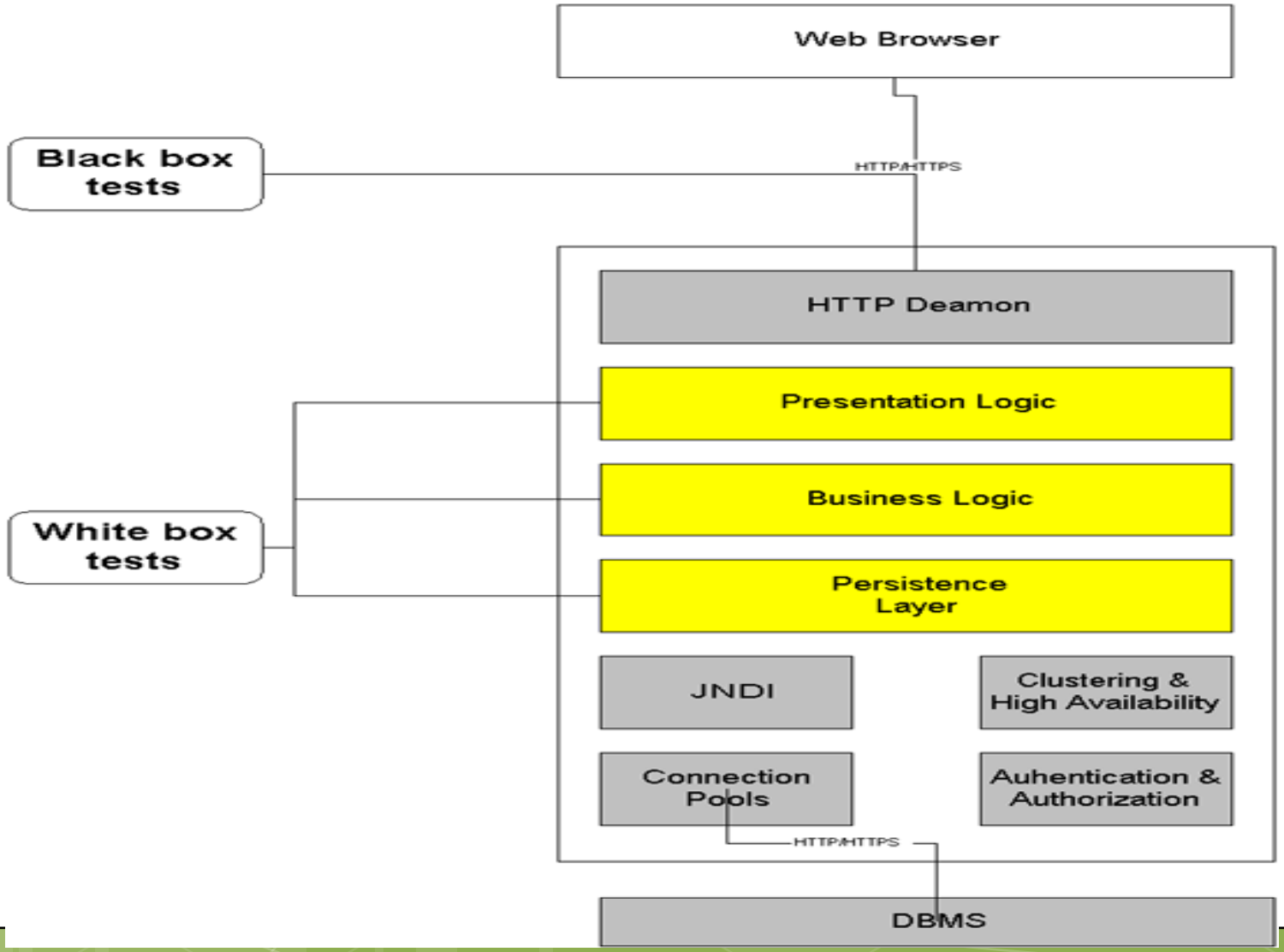
Untuk pengujian yang lengkap maka suatu perangkat lunak harus diuji dengan white box dan black box testing

White Box Testing

Dengan Menggunakan white box testing, software engineer da[at medesain suatu *test cases* yang dapat digunakan untu :

1. Menguji setiap jalur independent
2. Menguji keputusan logic (true atau falsa)
3. Menguji Loops dan batasannya
4. Menguji Data Struktur internalnya

White Box dan Black Box testing



White Box Testing

White Box Testing menggunakan 3 macam tahapam testing

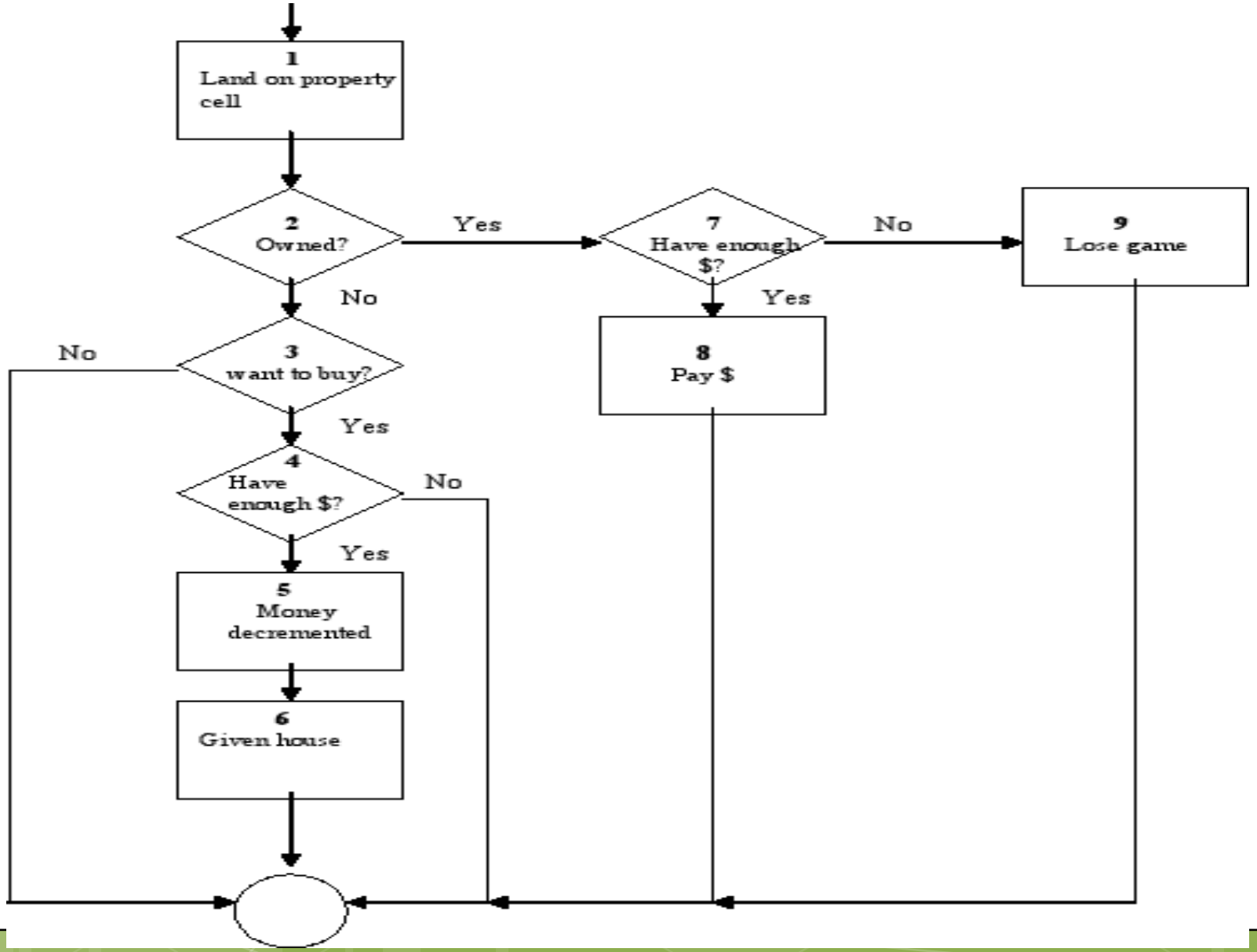
1. **Unit Testing**
2. **Integration testing**
3. **Regression Testing**

White Box Testing

Mendesain Test Case, Setiap membuat module/unit maka test case harus dilakukan : Test Awal yang dapat dilakukan adalah Analisa Code Coverage : Basic block coverage, Decision coverage, Condition coverage, Branch coverage, Loop coverage

1. Basis path testing
   - **Digunakan untuk melakukan pengujian bahwa semua jalur independent terlewati semua. Paling tidak suatu jalur minimal harus terlewati sekali.**

Basis Path Testing

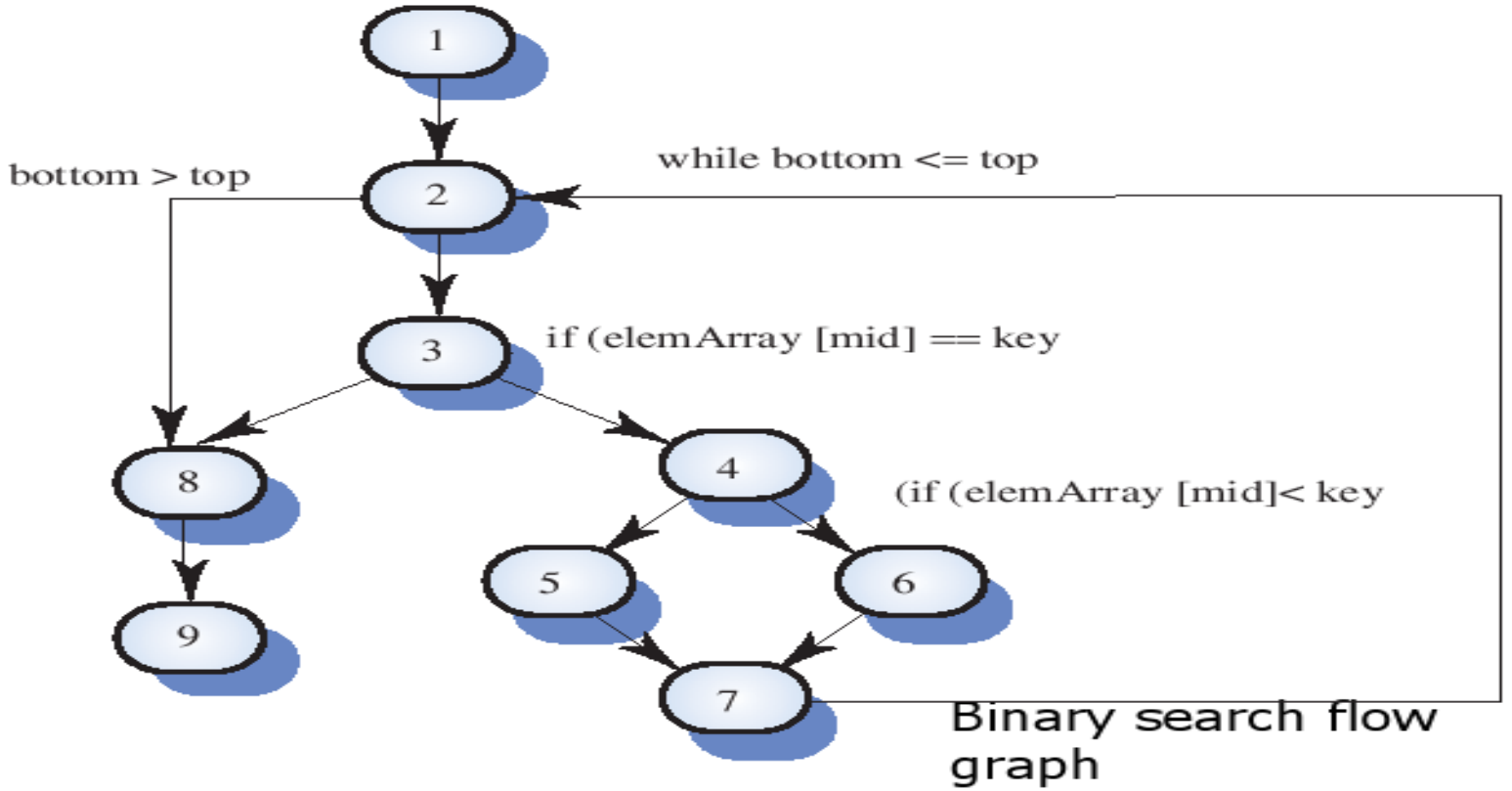


White Box Testing

Independent Path pada gambar di atas:
1. 1-2-7-8 (property owned, pay rent)
2. 1-2-7-9 (property owned, no money for rent)
3. 1-2-3-4-5-6 (buy house)
4. 1-2-3 (don't want to buy)
5. 1-2-3-4 (want to buy, don't have enough money)

Cyclomatic Complecity : Suatu ukuran untuk menghitung kekomplekan suatu kode sumber :
CC = Jumlah edges – jumlah node + 1
atau
Jumlah Kondisi + 1

White Box Testing



Binary search flow graph

White Box Testing

```
public static int binarySearch( int key, int[] sequence ) {
  int bottom = 0;
  int top = sequence.length - 1;
  int mid = 0;
  int keyPosition = -1;

  while( bottom <= top && keyPosition == -1 ) {
    mid = ( top + bottom ) / 2;
    if( sequence[ mid ] == key ) {
      keyPosition = mid;
    }
    else {
      if( sequence[ mid ] < key ) {
        bottom = mid + 1;
      }
      else {
        top = mid - 1;
      }
    }
  }
  return keyPosition;
}
```

White Box Testing

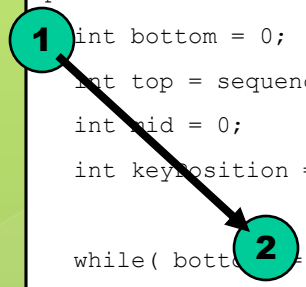
```
public static int binarySearch( int key, int[] sequence ) {
  int bottom = 0;
  int top = sequence.length - 1;
  int mid = 0;
  int keyPosition = -1;

  while( bottom <= top && keyPosition == -1 ) {
    mid = ( top + bottom ) / 2;
    if( sequence[ mid ] == key ) {
      keyPosition = mid;
    }
    else {
      if( sequence[ mid ] < key ) {
        bottom = mid + 1;
      }
      else {
        top = mid - 1;
      }
    }
  }
  return keyPosition;
}
```

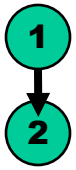
1

1

White Box Testing

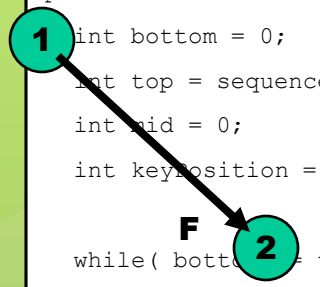
```
public static int binarySearch( int key, int[] sequence ) {
  int bottom = 0;
  int top = sequence.length - 1;
  int mid = 0;
  int keyPosition = -1;

  while( bottom <= top && keyPosition == -1 ) {
    mid = ( top + bottom ) / 2;
    if( sequence[ mid ] == key ) {
      keyPosition = mid;
    }
    else {
      if( sequence[ mid ] < key ) {
        bottom = mid + 1;
      }
      else {
        top = mid - 1;
      }
    }
  }
  return keyPosition;
}
```

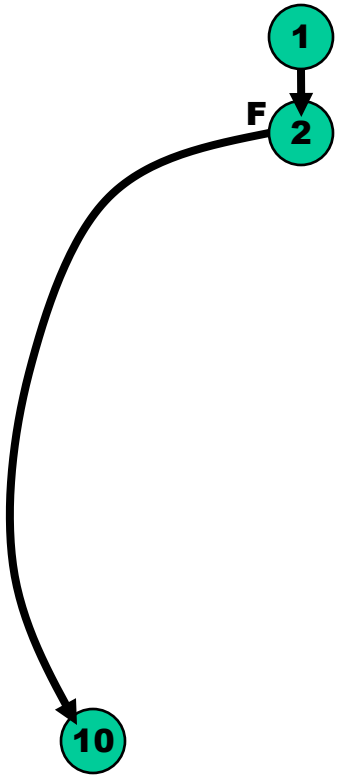




White Box Testing

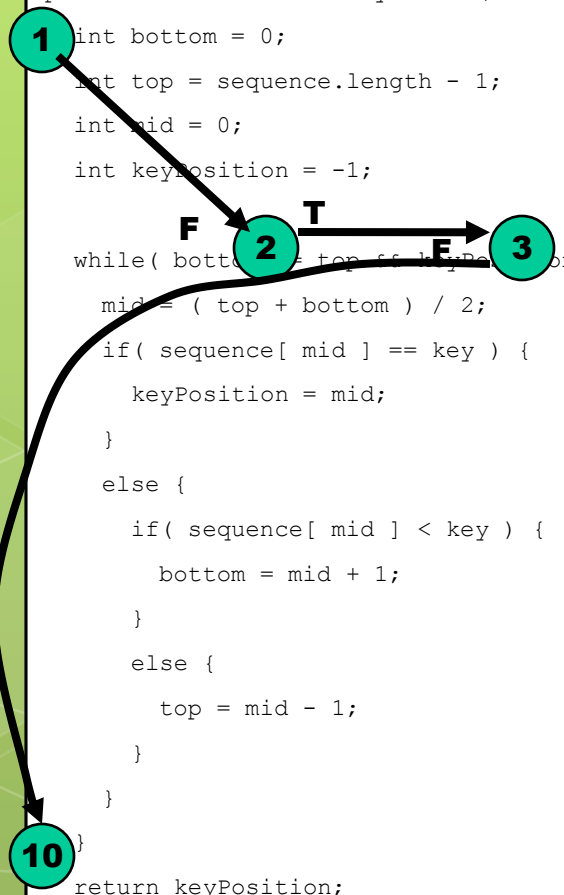
```
public static int binarySearch( int key, int[] sequence ) {
  int bottom = 0;
  int top = sequence.length - 1;
  int mid = 0;
  int keyPosition = -1;

  while( bottom <= top && keyPosition == -1 ) {
    mid = ( top + bottom ) / 2;
    if( sequence[ mid ] == key ) {
      keyPosition = mid;
    }
    else {
      if( sequence[ mid ] < key ) {
        bottom = mid + 1;
      }
      else {
        top = mid - 1;
      }
    }
  }
  return keyPosition;
}
```

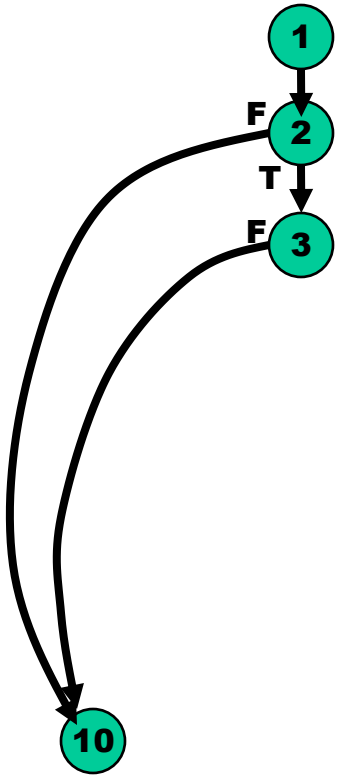




White Box Testing

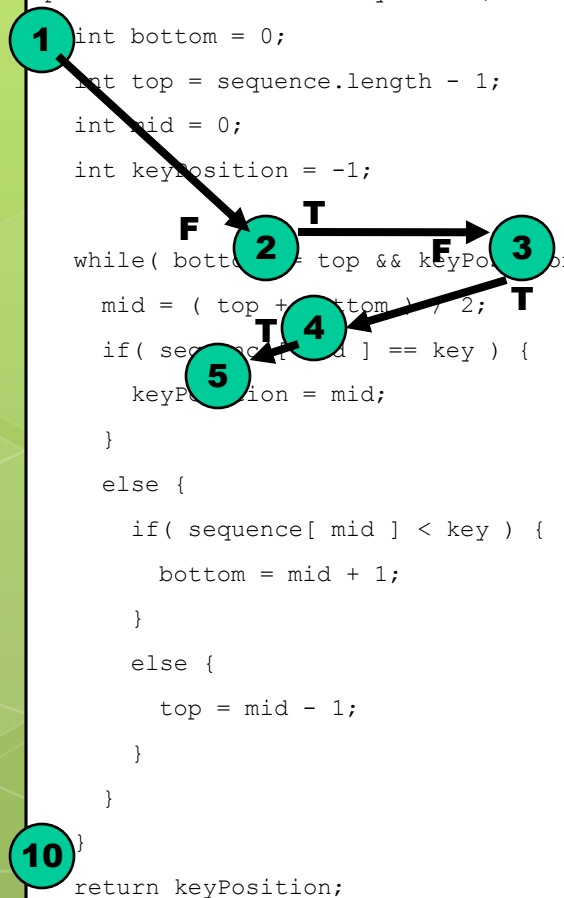
```
public static int binarySearch( int key, int[] sequence ) {
  int bottom = 0;
  int top = sequence.length - 1;
  int mid = 0;
  int keyPosition = -1;
  while( bottom <= top && keyPosition == -1 ) {
    mid = ( top + bottom ) / 2;
    if( sequence[ mid ] == key ) {
      keyPosition = mid;
    }
    else {
      if( sequence[ mid ] < key ) {
        bottom = mid + 1;
      }
      else {
        top = mid - 1;
      }
    }
  }
  return keyPosition;
}
```

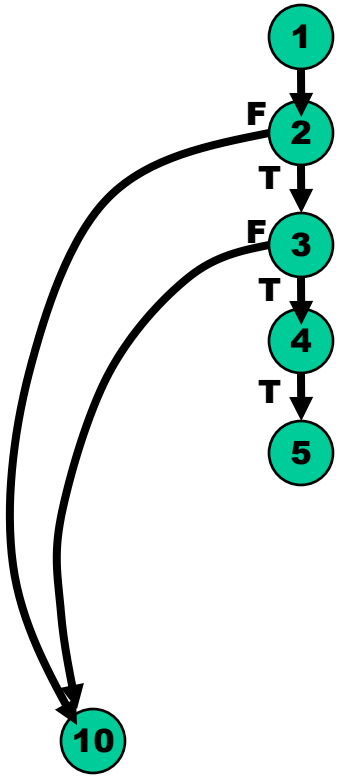




White Box Testing

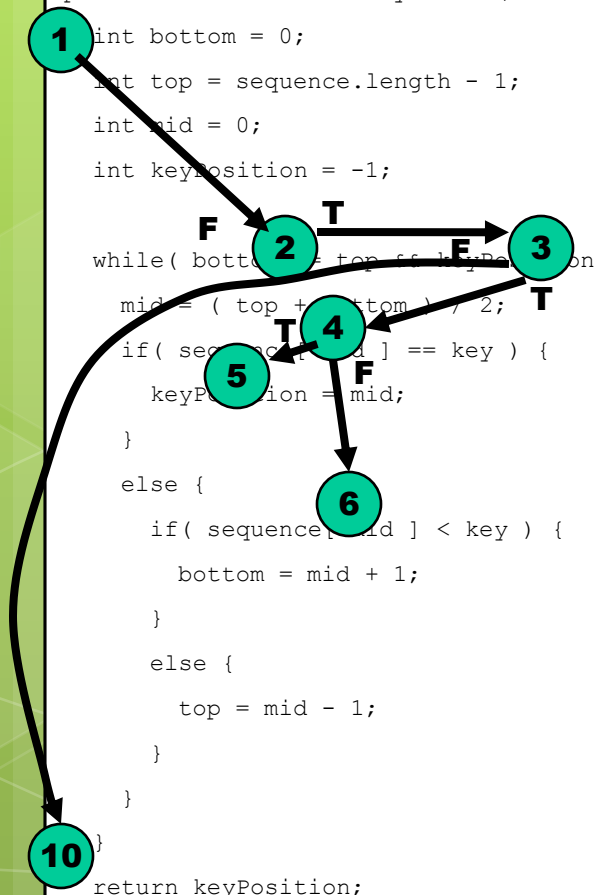
```
public static int binarySearch( int key, int[] sequence ) {
  int bottom = 0;
  int top = sequence.length - 1;
  int mid = 0;
  int keyPosition = -1;
  while( bottom <= top && keyPosition == -1 ) {
    mid = ( top + bottom ) / 2;
    if( sequence[ mid ] == key ) {
      keyPosition = mid;
    }
    else {
      if( sequence[ mid ] < key ) {
        bottom = mid + 1;
      }
      else {
        top = mid - 1;
      }
    }
  }
  return keyPosition;
}
```

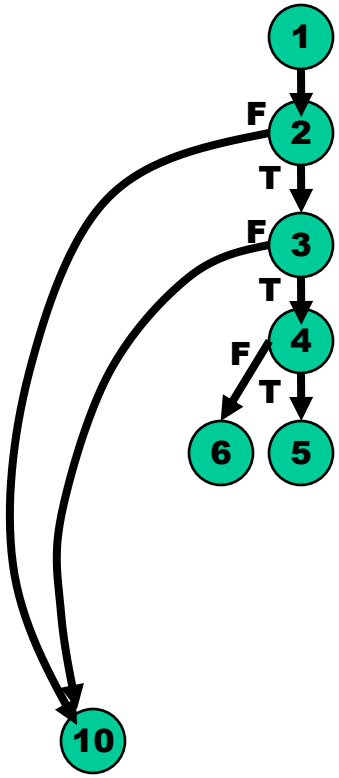




White Box Testing

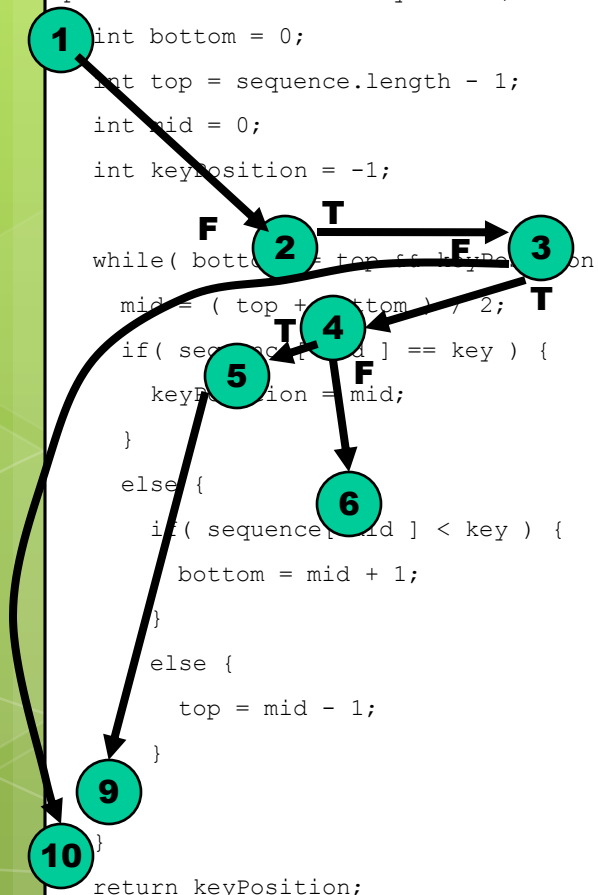
```
public static int binarySearch( int key, int[] sequence ) {
  int bottom = 0;
  int top = sequence.length - 1;
  int mid = 0;
  int keyPosition = -1;

  while( bottom <= top && keyPosition == -1 ) {
    mid = ( top + bottom ) / 2;
    if( sequence[ mid ] == key ) {
      keyPosition = mid;
    }
    else {
      if( sequence[ mid ] < key ) {
        bottom = mid + 1;
      }
      else {
        top = mid - 1;
      }
    }
  }
  return keyPosition;
}
```

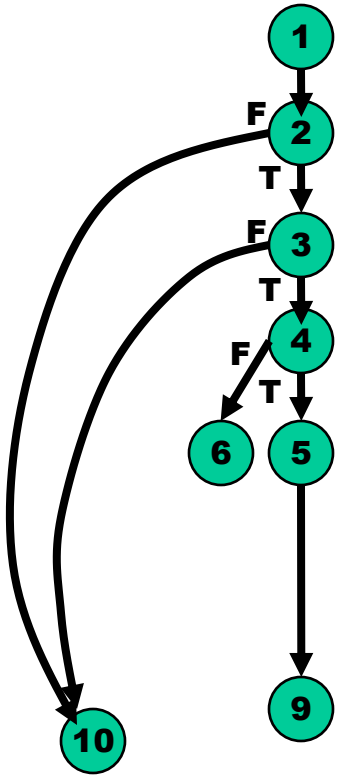




White Box Testing

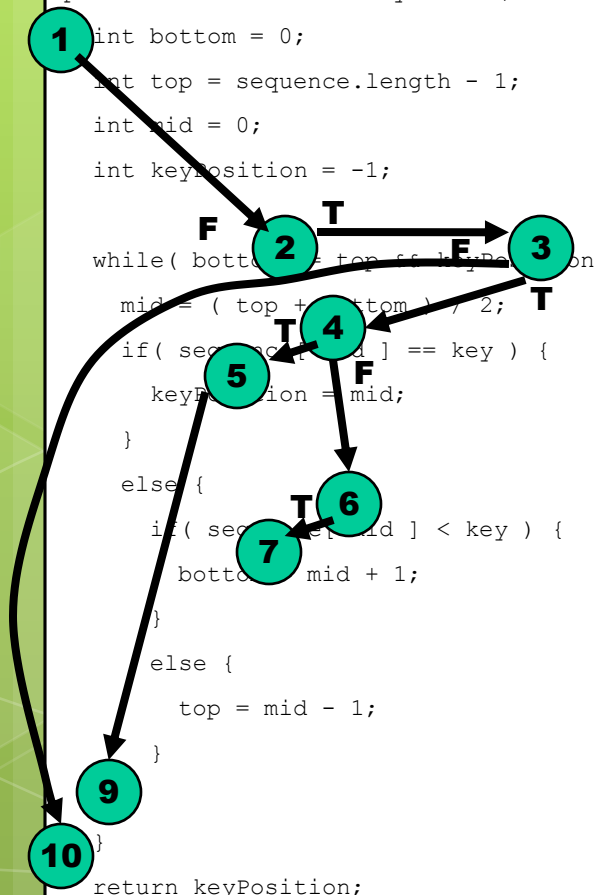
```
public static int binarySearch( int key, int[] sequence ) {
  int bottom = 0;
  int top = sequence.length - 1;
  int mid = 0;
  int keyPosition = -1;

  while( bottom <= top && keyPosition == -1 ) {
    mid = ( top + bottom ) / 2;
    if( sequence[ mid ] == key ) {
      keyPosition = mid;
    }
    else {
      if( sequence[ mid ] < key ) {
        bottom = mid + 1;
      }
      else {
        top = mid - 1;
      }
    }
  }
  return keyPosition;
}
```

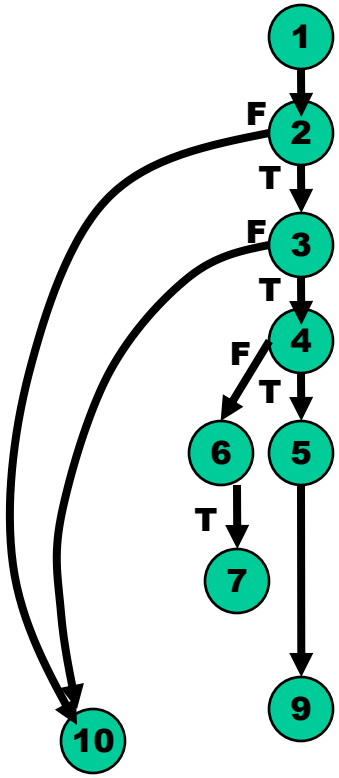




White Box Testing

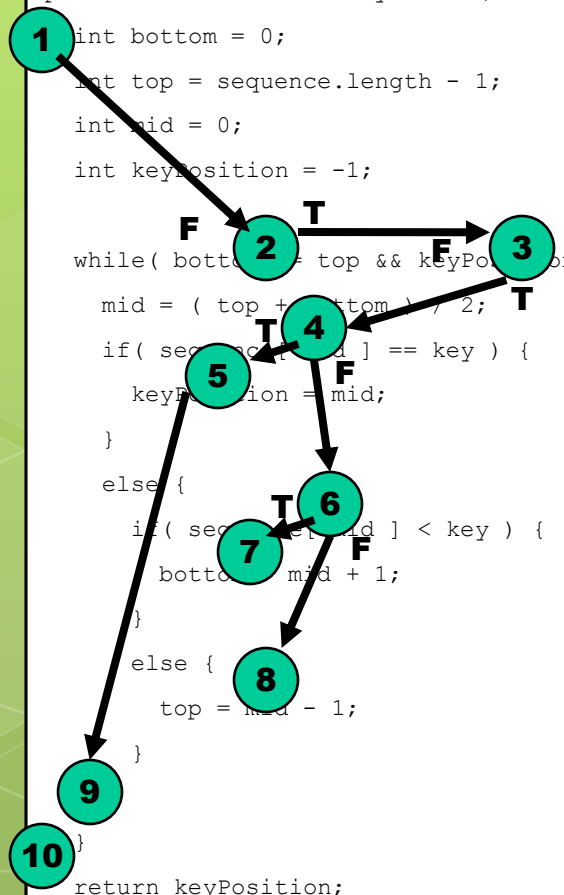
```
public static int binarySearch( int key, int[] sequence ) {
  int bottom = 0;
  int top = sequence.length - 1;
  int mid = 0;
  int keyPosition = -1;

  while( bottom <= top && keyPosition == -1 ) {
    mid = ( top + bottom ) / 2;
    if( sequence[ mid ] == key ) {
      keyPosition = mid;
    }
    else {
      if( sequence[ mid ] < key ) {
        bottom = mid + 1;
      }
      else {
        top = mid - 1;
      }
    }
  }
  return keyPosition;
}
```

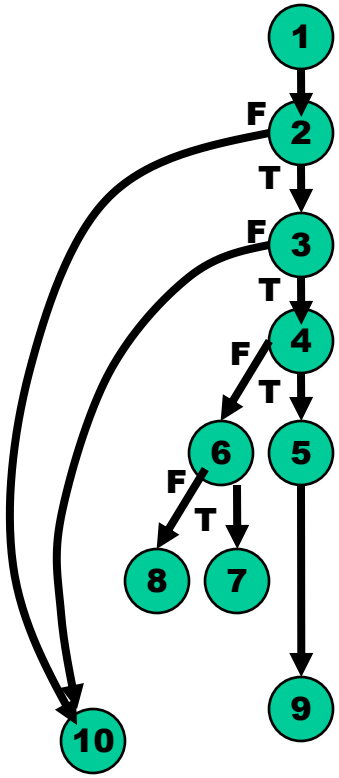




White Box Testing

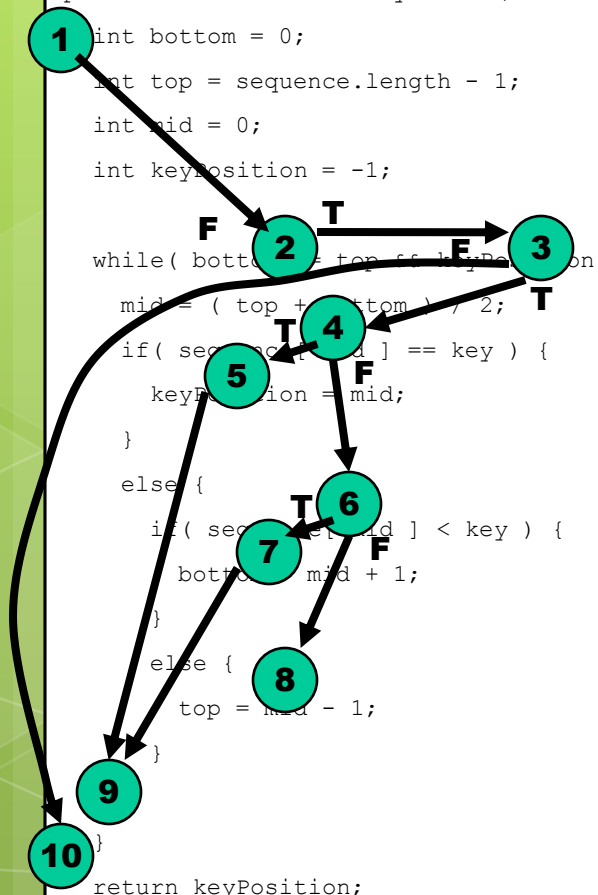
```
public static int binarySearch( int key, int[] sequence ) {
  int bottom = 0;
  int top = sequence.length - 1;
  int mid = 0;
  int keyPosition = -1;
  while( bottom <= top && keyPosition == -1 ) {
    mid = ( top + bottom ) / 2;
    if( sequence[ mid ] == key ) {
      keyPosition = mid;
    }
    else {
      if( sequence[ mid ] < key ) {
        bottom = mid + 1;
      }
      else {
        top = mid - 1;
      }
    }
  }
  return keyPosition;
}
```

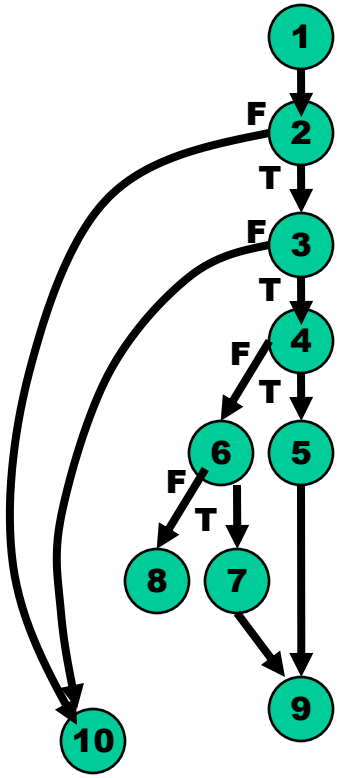




White Box Testing

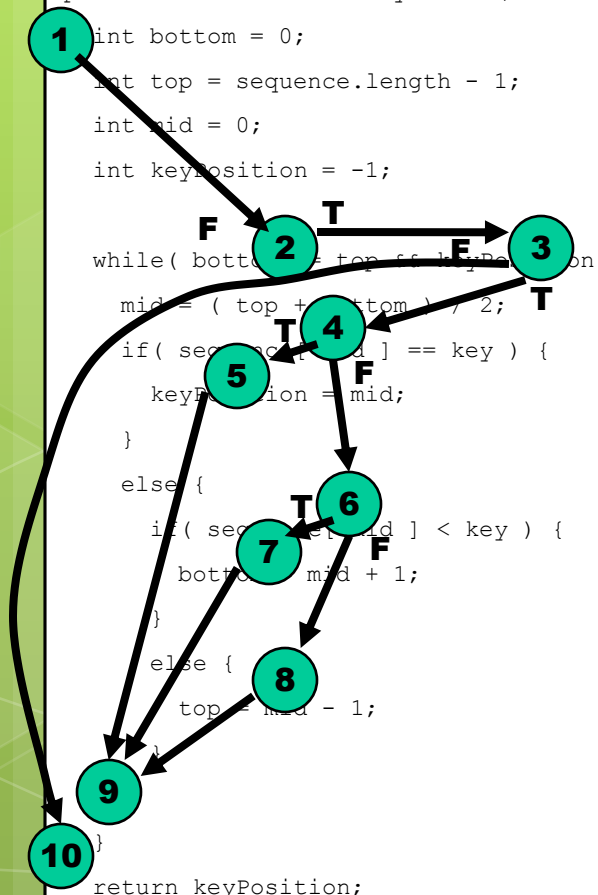
```
public static int binarySearch( int key, int[] sequence ) {
  int bottom = 0;
  int top = sequence.length - 1;
  int mid = 0;
  int keyPosition = -1;

  while( bottom <= top && keyPosition == -1 ) {
    mid = ( top + bottom ) / 2;
    if( sequence[ mid ] == key ) {
      keyPosition = mid;
    }
    else {
      if( sequence[ mid ] < key ) {
        bottom = mid + 1;
      }
      else {
        top = mid - 1;
      }
    }
  }
  return keyPosition;
}
```

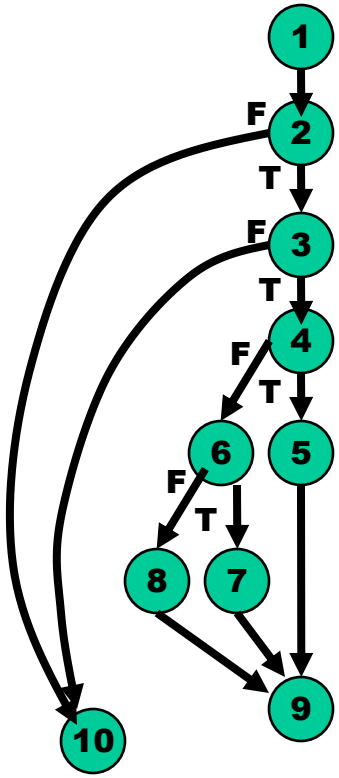




White Box Testing

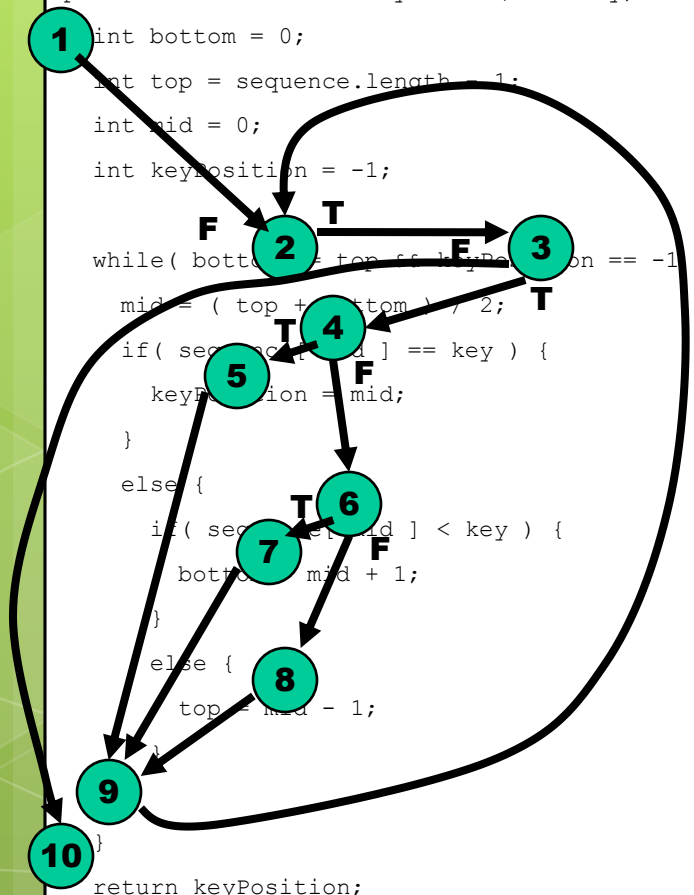
```
public static int binarySearch( int key, int[] sequence ) {
  int bottom = 0;
  int top = sequence.length - 1;
  int mid = 0;
  int keyPosition = -1;

  while( bottom <= top && keyPosition == -1 ) {
    mid = ( top + bottom ) / 2;
    if( sequence[ mid ] == key ) {
      keyPosition = mid;
    }
    else {
      if( sequence[ mid ] < key ) {
        bottom = mid + 1;
      }
      else {
        top = mid - 1;
      }
    }
  }
  return keyPosition;
}
```

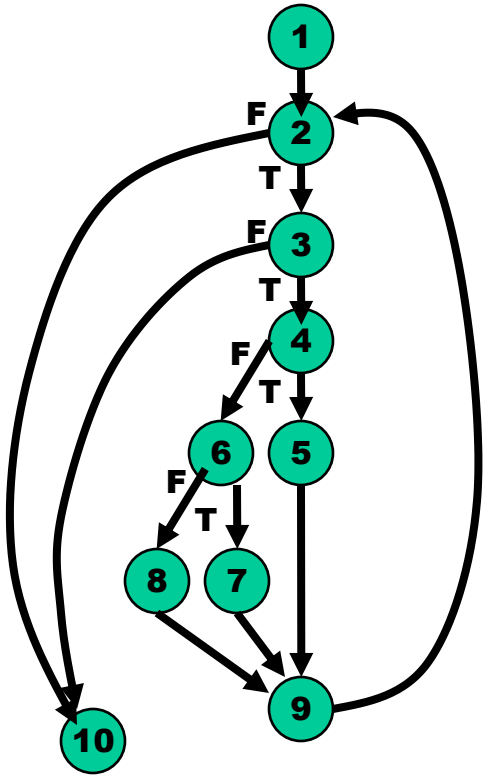




White Box Testing

```
public static int binarySearch( int key, int[] sequence ) {
  int bottom = 0;
  int top = sequence.length - 1;
  int mid = 0;
  int keyPosition = -1;

  while( bottom <= top && keyPosition == -1 ) {
    mid = ( top + bottom ) / 2;
    if( sequence[ mid ] == key ) {
      keyPosition = mid;
    }
    else {
      if( sequence[ mid ] < key ) {
        bottom = mid + 1;
      }
      else {
        top = mid - 1;
      }
    }
  }
  return keyPosition;
}
```





Test Case

- Work out the number of distinct paths.
  - **Cyclomatic Complexity**

    CC = noEdges – noNodes + 2

    CC = 13 – 10 + 2 = 5
- List the distinct paths.
  - **1, 2, 10**
  - **1, 2, 3, 10**
  - **1, 2, 3, 4, 5, 9, 2… (loop again?)**
  - **1, 2, 3, 4, 6, 7, 9, 2… (loop again?)**
  - **1, 2, 3, 4, 6, 8, 9, 2… (loop again?)**
- Figure out the conditions that cause execution of these paths.