



Testing dan Implementasi Sistem

Lukman Hakim dan Suwanto R

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 tentang perangkat lunak yang akan di uji.

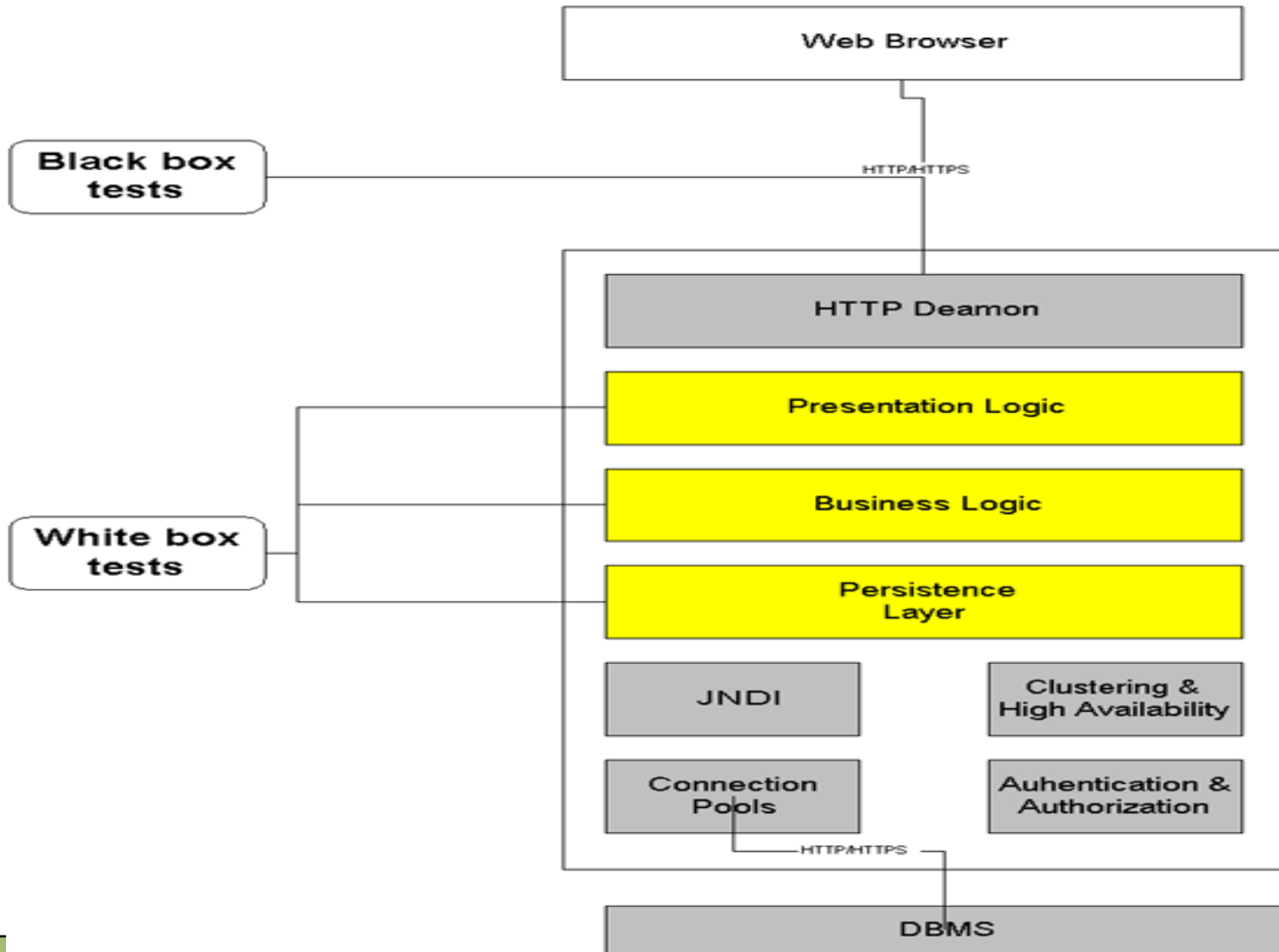
Untuk pengujian yang lengkap maka suatu perangkat lunak harus diuji dengan **white box** dan **black box testing**

White Box Testing

Dengan Menggunakan white box testing, software engineer dapat mendesain suatu *test cases* yang dapat digunakan untuk :

1. Menguji setiap jalur independent
2. Menguji keputusan logic (true atau false)
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**

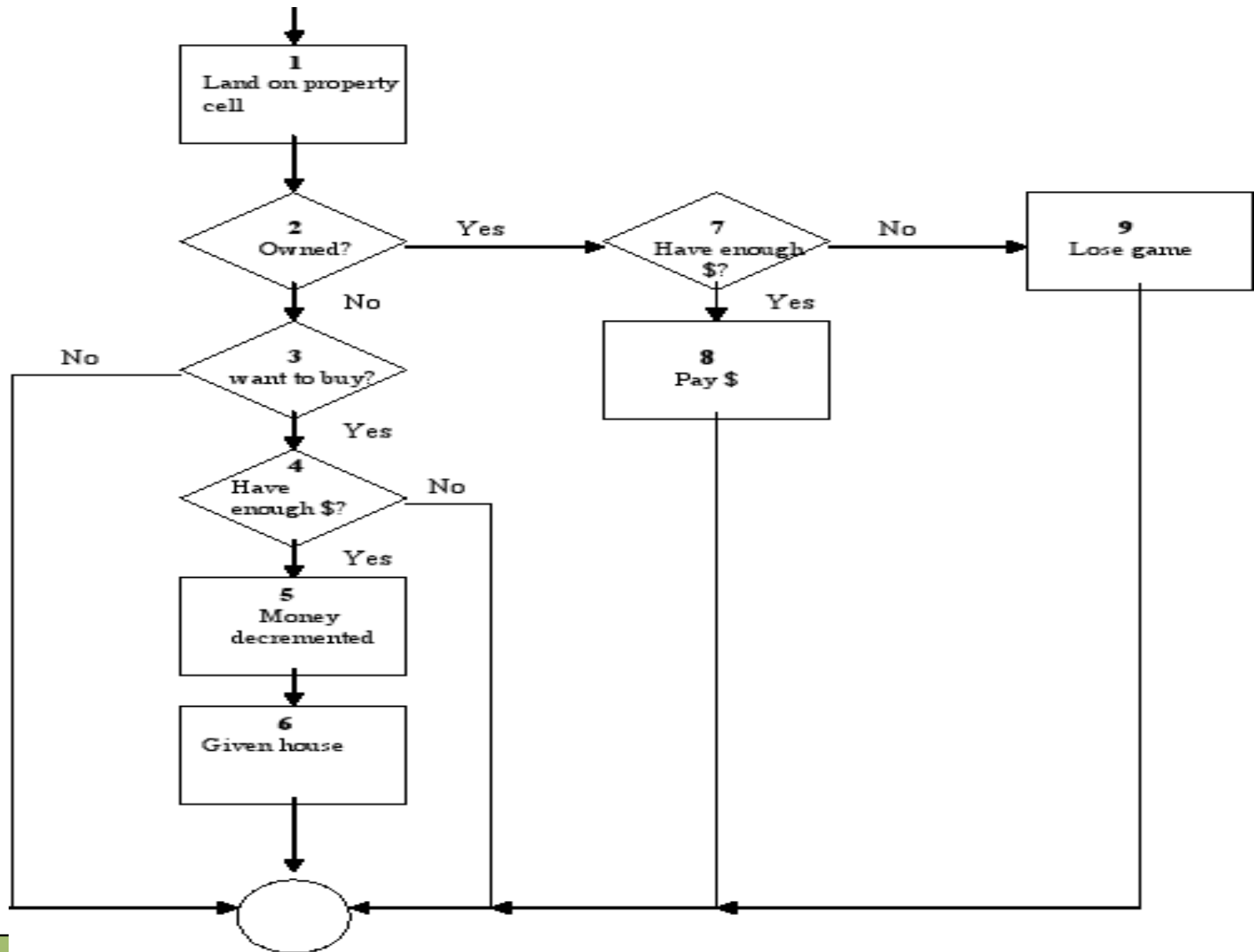
White Box Testing

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)

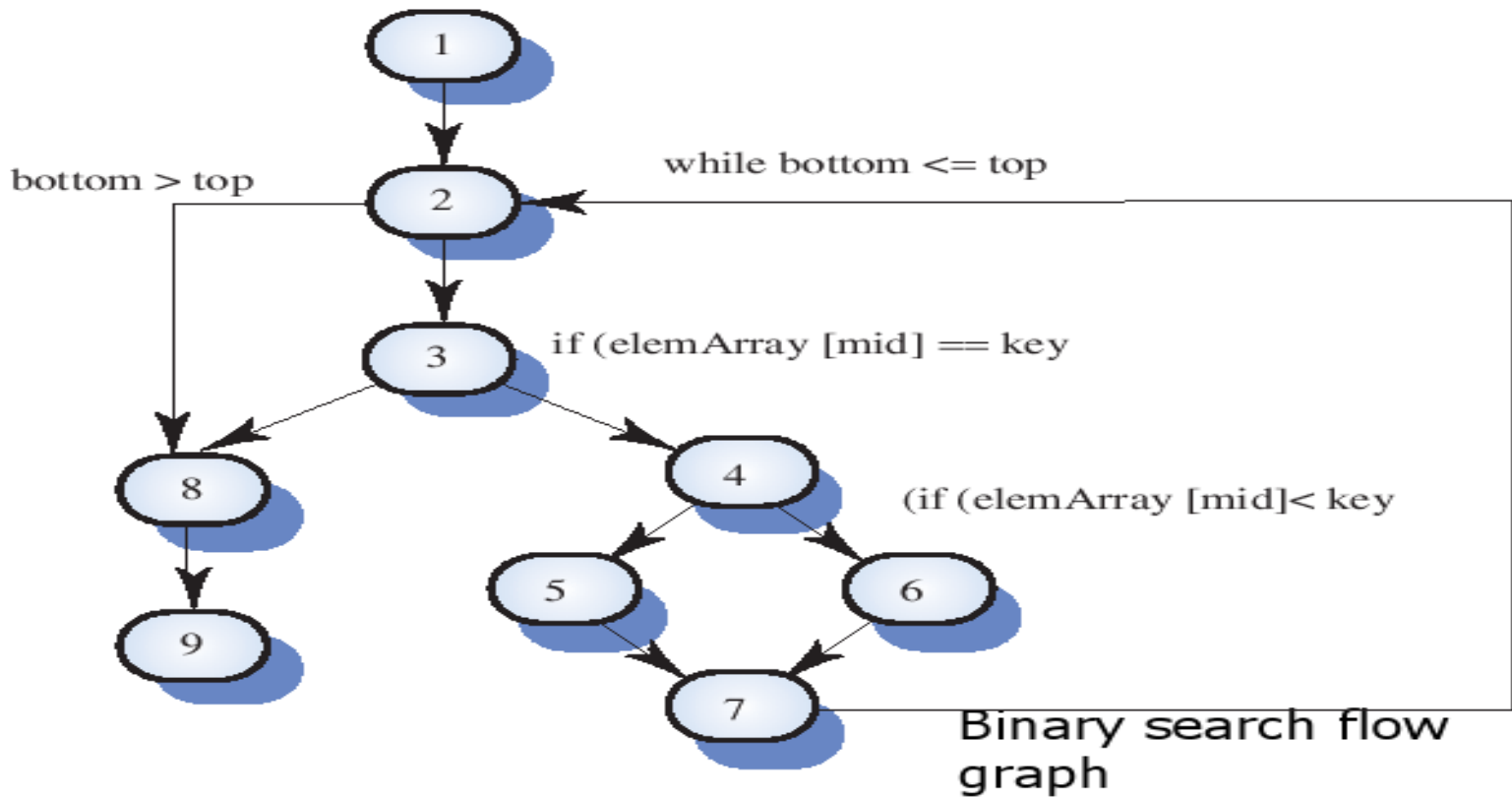
Cyclomatic Complecity : Suatu ukuran untuk menghitung kekomplekan suatu kode sumber :

CC = Jumlah edges – jumlah node + 1

atau

Jumlah Kondisi + 1

White Box Testing



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 ) {  
1 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

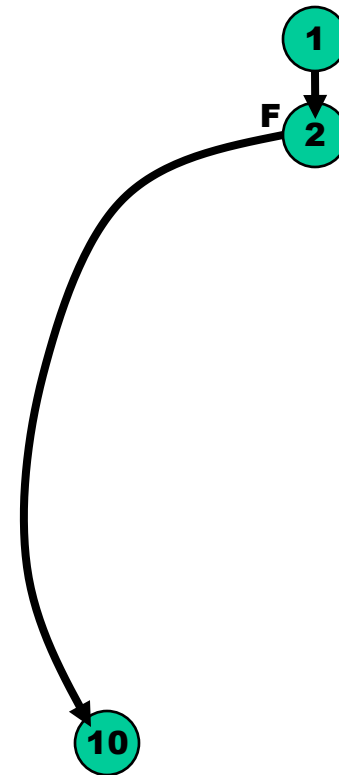
White Box Testing

```
public static int binarySearch( int key, int[] sequence ) {  
    1 int bottom = 0;  
    int top = sequence.length - 1;  
    int mid = 0;  
    int keyPosition = -1;  
  
    while( bott 2 = 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;  
}
```



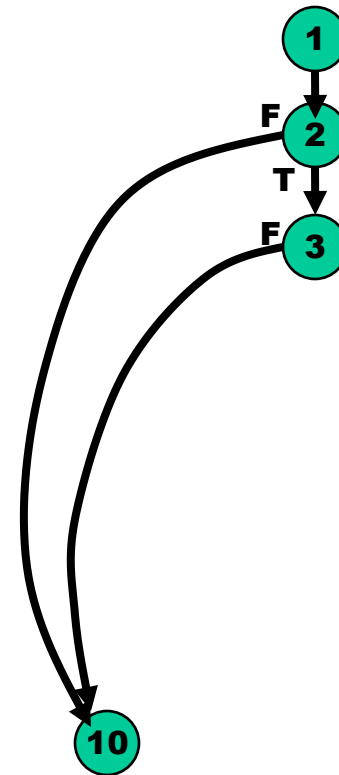
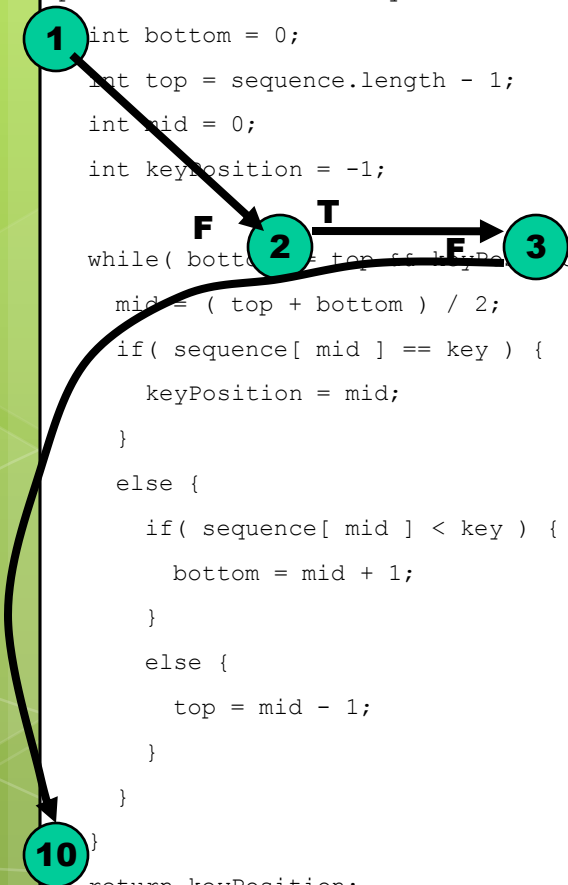
White Box Testing

```
public static int binarySearch( int key, int[] sequence ) {  
    1 int bottom = 0;  
    int top = sequence.length - 1;  
    int mid = 0;  
    int keyPosition = -1;  
  
    F 2 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;  
            }  
        }  
    }  
    10 }  
    return keyPosition;  
}
```



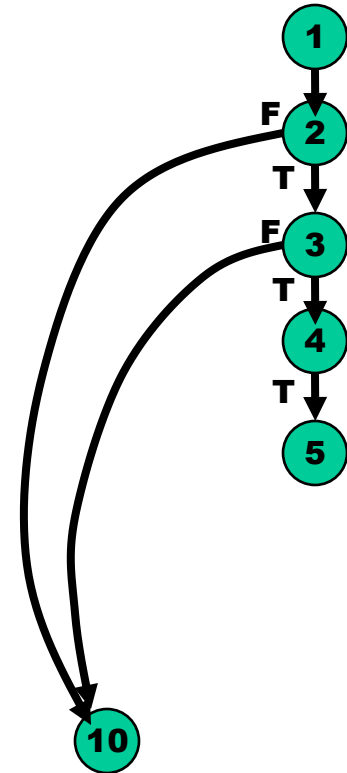
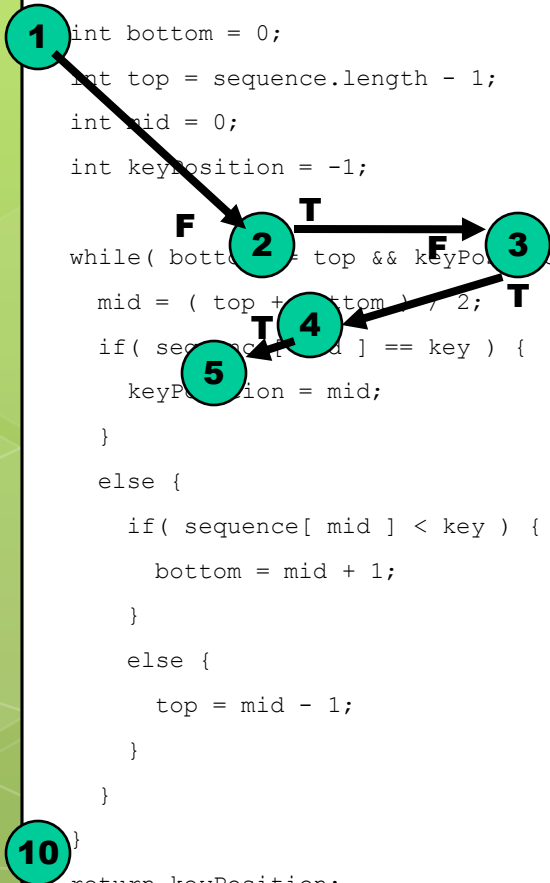
White Box Testing

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



White Box Testing

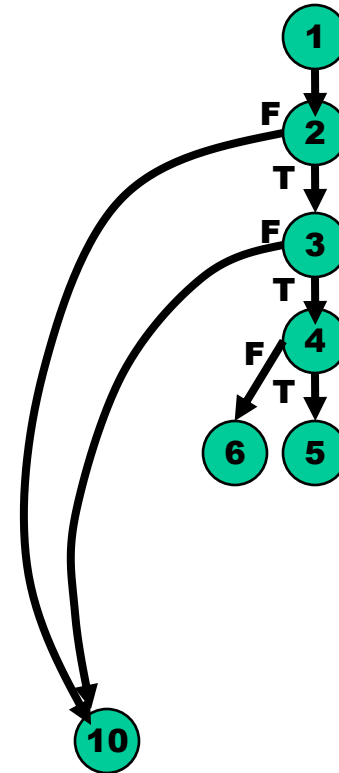
```
public static int binarySearch( int key, int[] sequence ) {  
    1 int bottom = 0;  
    int top = sequence.length - 1;  
    int mid = 0;  
    int keyPosition = -1;  
    while( bottom <= top && keyPosition == -1 ) {  
        2 mid = ( top + bottom ) / 2;  
        3 if( sequence[ mid ] == key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



White Box Testing

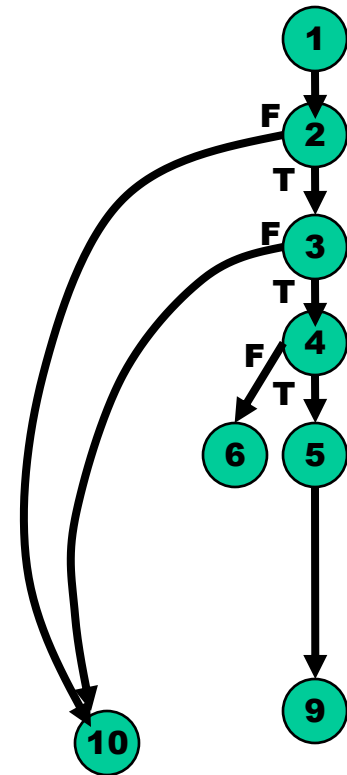
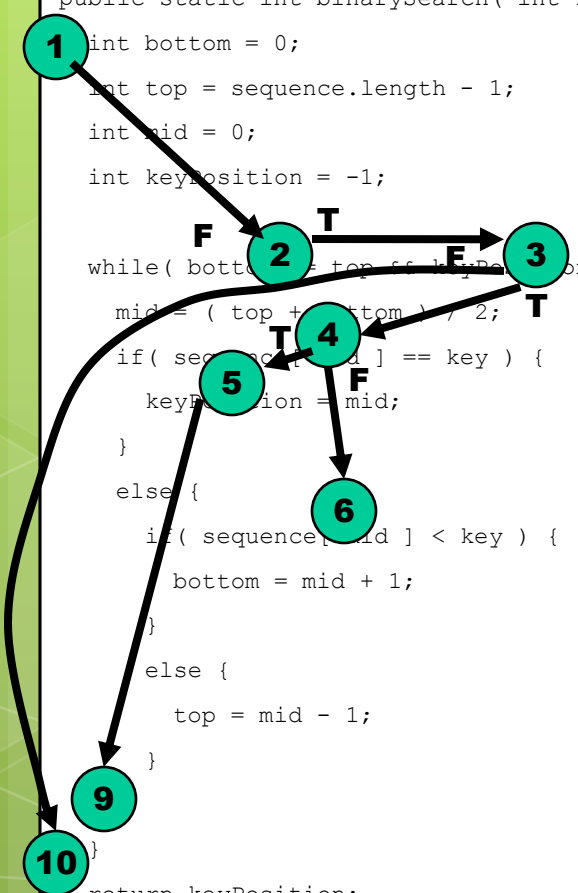
```
public static int binarySearch( int key, int[] sequence ) {  
    1 int bottom = 0;  
    int top = sequence.length - 1;  
    int mid = 0;  
    int keyPosition = -1;  
    while( bottom <= top && keyPosition == -1 ) {  
        2 mid = ( top + bottom ) / 2;  
        if( sequence[ mid ] == key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

The flowchart illustrates the execution of the binarySearch function. It starts at node 1, which is the function signature. It then proceeds to node 2, which is the first assignment statement. From node 2, there are two paths: a 'T' (True) path leading to node 3, and an 'F' (False) path leading to node 10. Node 3 is the start of a while loop. From node 3, there are two paths: a 'T' path leading to node 4, and an 'F' path leading to node 10. Node 4 is the start of an if statement. From node 4, there are two paths: a 'T' path leading to node 5, and an 'F' path leading to node 6. Node 5 is the assignment of keyPosition. From node 5, there is a path leading to node 10. Node 6 is the start of an else block. From node 6, there are two paths: a 'T' path leading to node 4, and an 'F' path leading to node 10. Node 10 is the return statement.



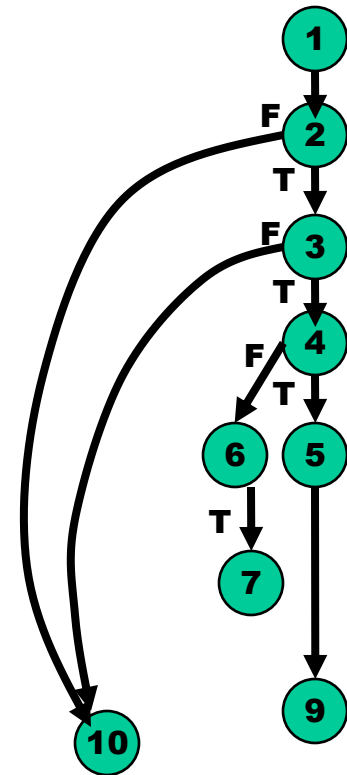
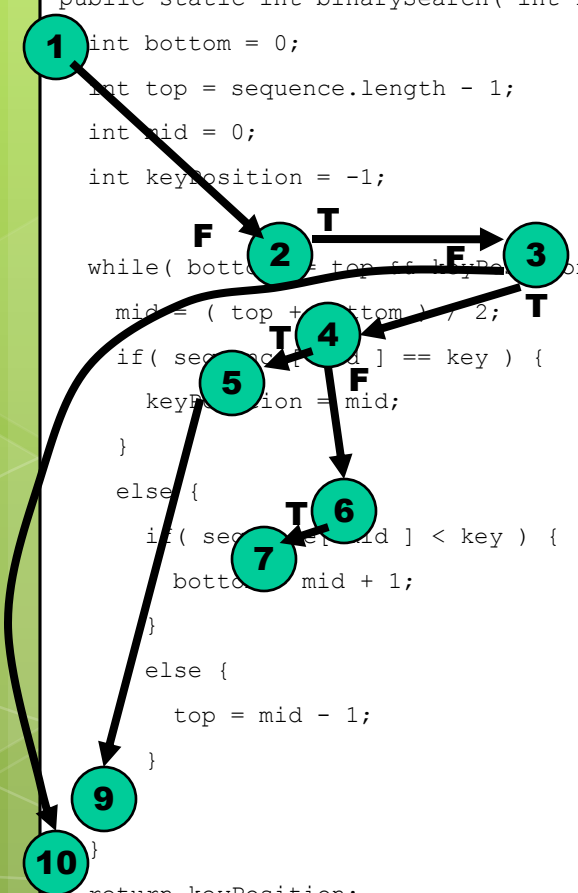
White Box Testing

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



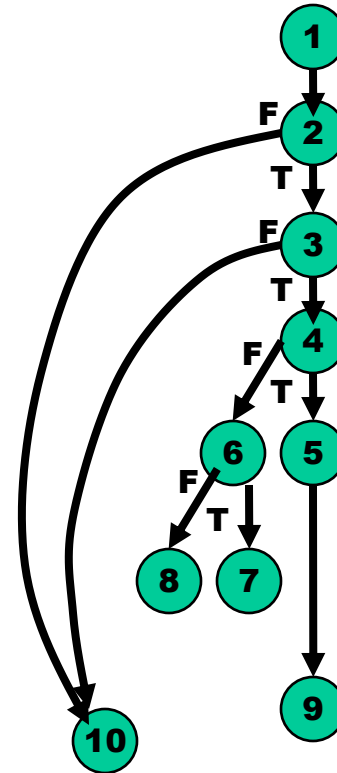
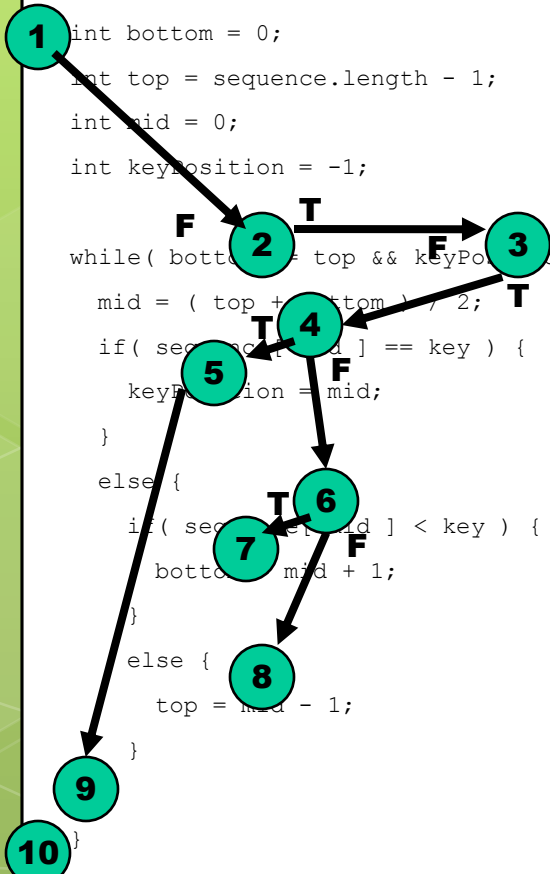
White Box Testing

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



White Box Testing

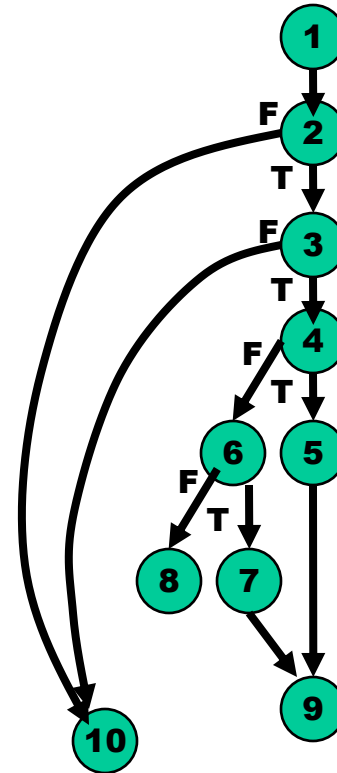
```
public static int binarySearch( int key, int[] sequence ) {  
    1 int bottom = 0;  
    int top = sequence.length - 1;  
    int mid = 0;  
    int keyPosition = -1;  
    while( bottom <= top && keyPosition == -1 ) {  
        2 mid = ( top + bottom ) / 2;  
        3 if( sequence[ mid ] == key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



White Box Testing

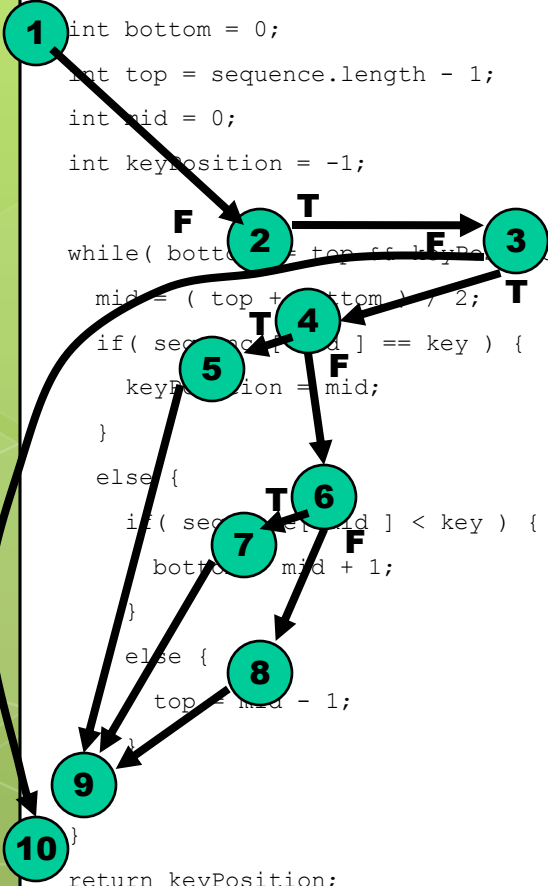
```
public static int binarySearch( int key, int[] sequence ) {  
    1 int bottom = 0;  
    int top = sequence.length - 1;  
    int mid = 0;  
    int keyPosition = -1;  
    while( bottom <= top && keyPosition == -1 ) {  
        2 mid = ( top + bottom ) / 2;  
        if( sequence[ mid ] == key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

The flowchart illustrates the execution of the binarySearch function. It starts at node 1, which is the function signature. Node 2 is the first assignment statement. Node 3 is the start of the while loop. Node 4 is the mid calculation. Node 5 is the first if statement. Node 6 is the else block. Node 7 is the if statement inside the else block. Node 8 is the else block inside the else block. Node 9 is the return statement. Node 10 is the end of the function. Decision points are labeled with 'F' for false and 'T' for true. The flow is as follows: 1 to 2, 2 to 3, 3 to 4, 4 to 5, 5 to 6, 6 to 7, 7 to 8, 8 to 9, 9 to 10. There are also feedback loops: 5 to 4, 6 to 4, 7 to 4, 8 to 4, 4 to 3, 3 to 2, 2 to 1, 3 to 10.



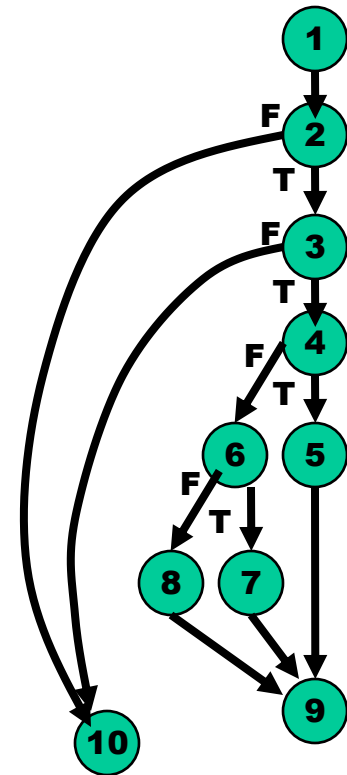
White Box Testing

```
public static int binarySearch( int key, int[] sequence ) {
```



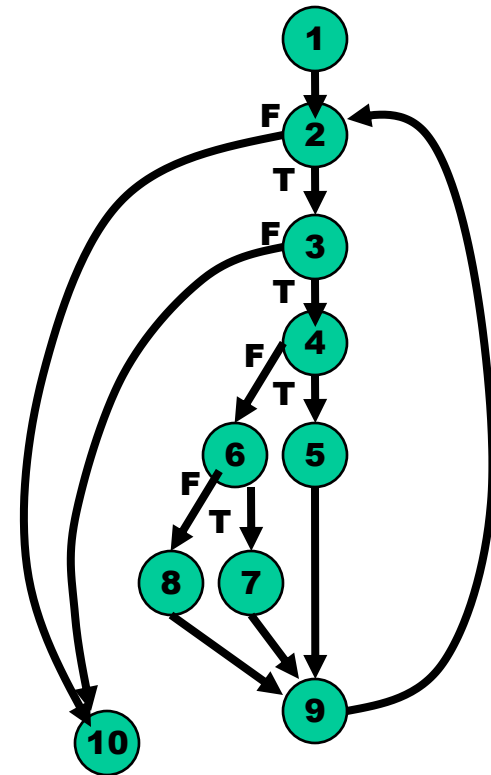
```
    int bottom = 0;
    int top = sequence.length - 1;
    int mid = 0;
    int keyPosition = -1;

    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 ) {  
    1 int bottom = 0;  
    int top = sequence.length - 1;  
    int mid = 0;  
    int keyPosition = -1;  
    2 while( bottom <= top && keyPosition == -1 ) {  
        mid = ( top + bottom ) / 2;  
        3 if( sequence[mid] == key ) {  
            keyPosition = mid;  
        }  
        else {  
            4 if( sequence[mid] < key ) {  
                bottom = mid + 1;  
            }  
            else {  
                5 top = mid - 1;  
            }  
        }  
    }  
    6 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.