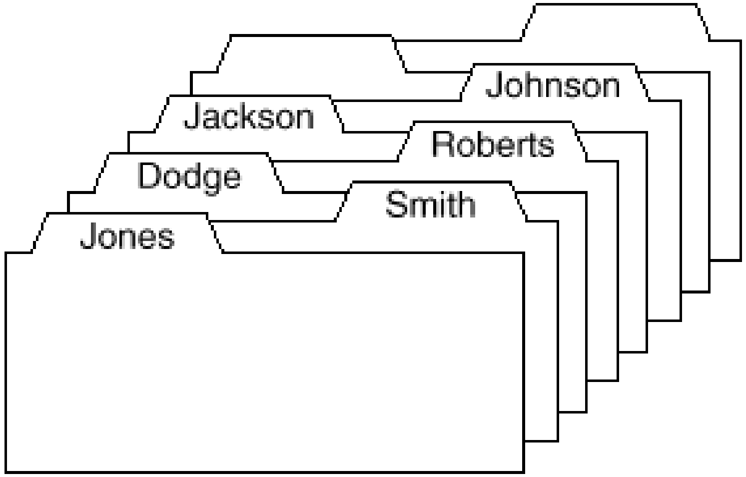
Design and Analysis of Algorithms

Lecture 6: Searching Algorithms analysis and design

Material is from Chapter 6: Kruse’s book

Search



To analyze the behavior of an algorithm that makes comparison of keys, we shall use the count of comparisons of those keys as our measure of running time.

Sequential search

```
int SequentialSearch(List list, KeyType target)
{
        int location;
        for (location = 0; location < list.count; location++)
        if (EQ(list.entry[location].key, target))
          return location;
        return -1;
}
```

The number of comparisons of keys done in sequential search of a list of length n is

- Unsuccessful search: n comparisons.
- Successful search, best case: 1 comparison.
- Successful search, worst case: n comparisons.
- Successful search, average case: $\frac{1}{2}(n+1)$ comparisons.

Binary Search

- The method date back at least to 1946, but the first version free of errors and unnecessary restrictions seems to appeared only in 1962!
- One study showed that about 90% of professional programmers fail to code binary search correctly, even after working on it one full hour.

Idea

- Start with an ordered list
- When searching an ordered list
  - first compare the target to the key in the center of the list.
  - If it is smaller, restrict the search to the left half;
  - otherwise restrict the search to the right half, and repeat.
  - In this way, at each step we reduce the length of the list to be searched by half.

Binary 1 Search - Recursive

```
int RecBinary1(List list, KeyType target, int bottom, int top)
{
            int middle = -1;
            if (bottom < top)
             { /* The list has size greater than 1. */
                   middle = (top + bottom) / 2;
                  if (GT(target, list.entry[middle].key))
                       /* Reduce to the top half of the list.*/
                      middle = RecBinary1(list, target, middle+1, top);
                  else
                       /* Reduce to the bottom half of the list.*/
                      middle = RecBinary1(list, target, bottom, middle);
              }
             else
                  if (bottom == top)
                  {/* The list has exactly 1 entry. */
                        if (EQ(target, list.entry[top].key))
                       middle = top;
                   }

           return middle;
}
```

Binary 1 Search - Iterative

```
int Binary1Search(List list, KeyType target)
{
        int bottom, middle, top; /* Initialize bounds to encompass entire list*/
        top = list.count – 1;
        bottom = 0;
        while (top>bottom) /* Check terminating condition */
         {
            middle = (top + bottom)/2;
            if(GT(target, list.entry[middle].key))
               bottom = middle + 1; /* Reduce to the top half of the list */
            else
               top = middle; /* Reduce to the bottom half of the list */
         }
         if (top==-1)
            return -1;  /* Search for an empty list always fails */

         if(EQ(list.entry[top].key, target))
            return top;
         else
            return -1;
}
```

Upgrade Binary Search with Equality Check

- Binary1 may make many unnecessary iterations because it may fail to recognize that middle is the actual target!

Binary 2 Search - Recursive

```
int RecBinary2(List list, KeyTypetarget, int bottom, int top)
{
    int middle = -1;
    if (bottom <= top)
    {
        middle = (top + bottom) / 2;
        if (LT(target, list.entry[middle].key))
            /* Reduce to the bottom half.*/
            middle = RecBinary2(list,target,bottom,middle-1);
        else
          if (GT(target, list.entry[middle].key)
            /* Reduce to the top half.*/
             middle = RecBinary2(list, target, middle + 1, top);
    }
    return middle;
}
```

Binary 2 Search - Iterative

```
Int Binary2Search(List list, KeyType target)
{
          int bottom, middle, top; /* Initialize bounds to encompass entire list*/
          top = list.count – 1;
          bottom = 0;
          while (top>=bottom) /* Check terminating condition */
           {
               middle = (top + bottom)/2;
               if(EQ(target, list.entry[middle].key))
                   return middle;
               else
                   if (LT(target, list.entry[middle].key))
                      top = middle - 1; /* Reduce to the bottom half of the list */
                  else
                      bottom = middle + 1; /* Reduce to the bottom half of the list */

           }
return -1;
}
```

Binary 1 vs 2

- Which version is more efficient?
- Vote

Comparison Tree

- **Def:** The *comparison tree of an* algorithm is obtained by tracing the action of the algorithm, representing each comparison of keys by a *vertex of the tree* (which we draw as a circle).

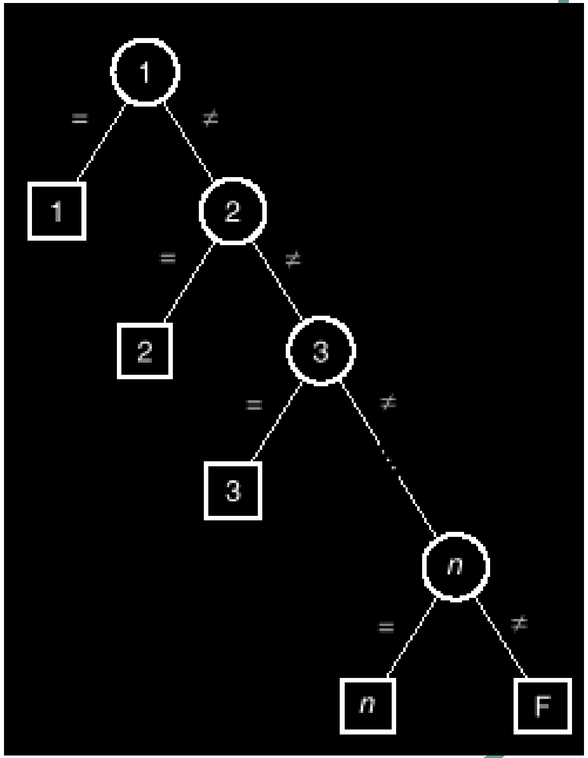
- **Intuitive:** The comparison tree represents all possible scenarios if search of n entries would be conducted.

- **Def:** Height of the tree is the number of vertices in the longest path that occurs

- **Def:** Children of vertex v are vertices immediately below a vertex v

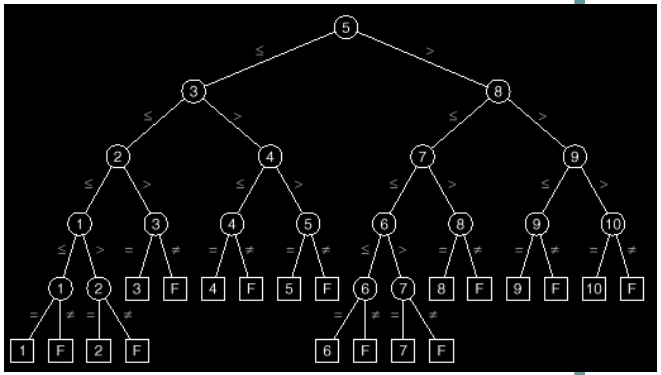
- **Def:** Parent of vertex b is the vertex immediately above the vertex v

Example of the comparison tree for the sequential search

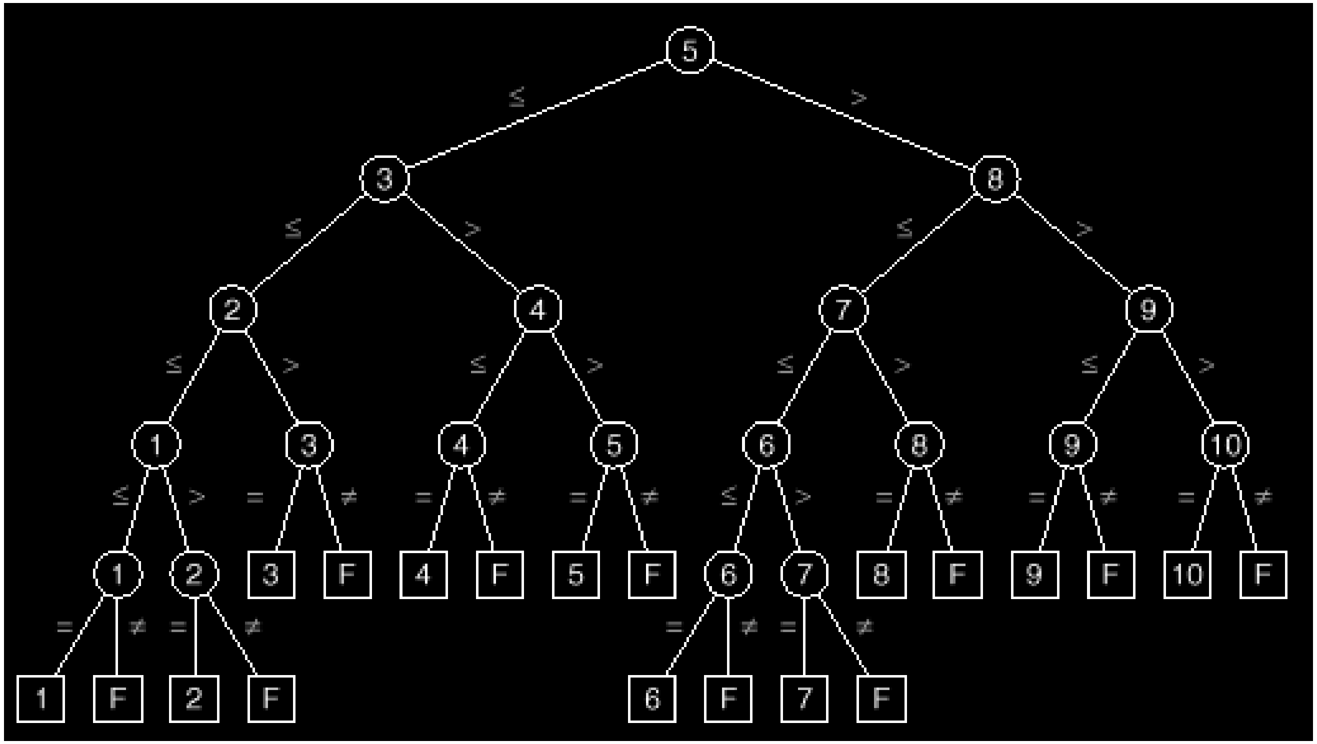


Definitions:

- Def: **<u>External path</u>** the sum of the number of branches traversed in going from the root once to every leaf in the tree.
- Def: **<u>Internal path</u>** length is the sum of the number of branches from the root to the vertex over for all vertices in the tree that are not leaves.

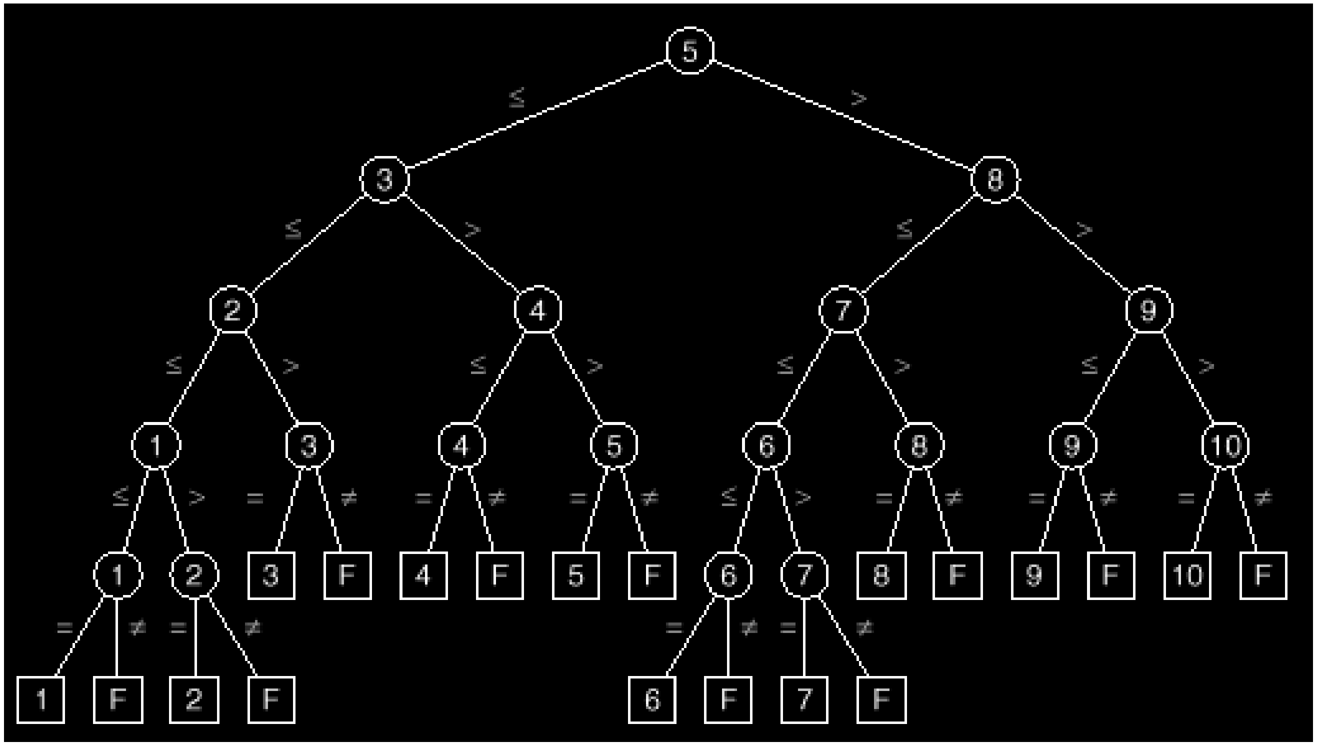


Comparison Tree for Binary1Search – Iterative solution (with 10 keys)



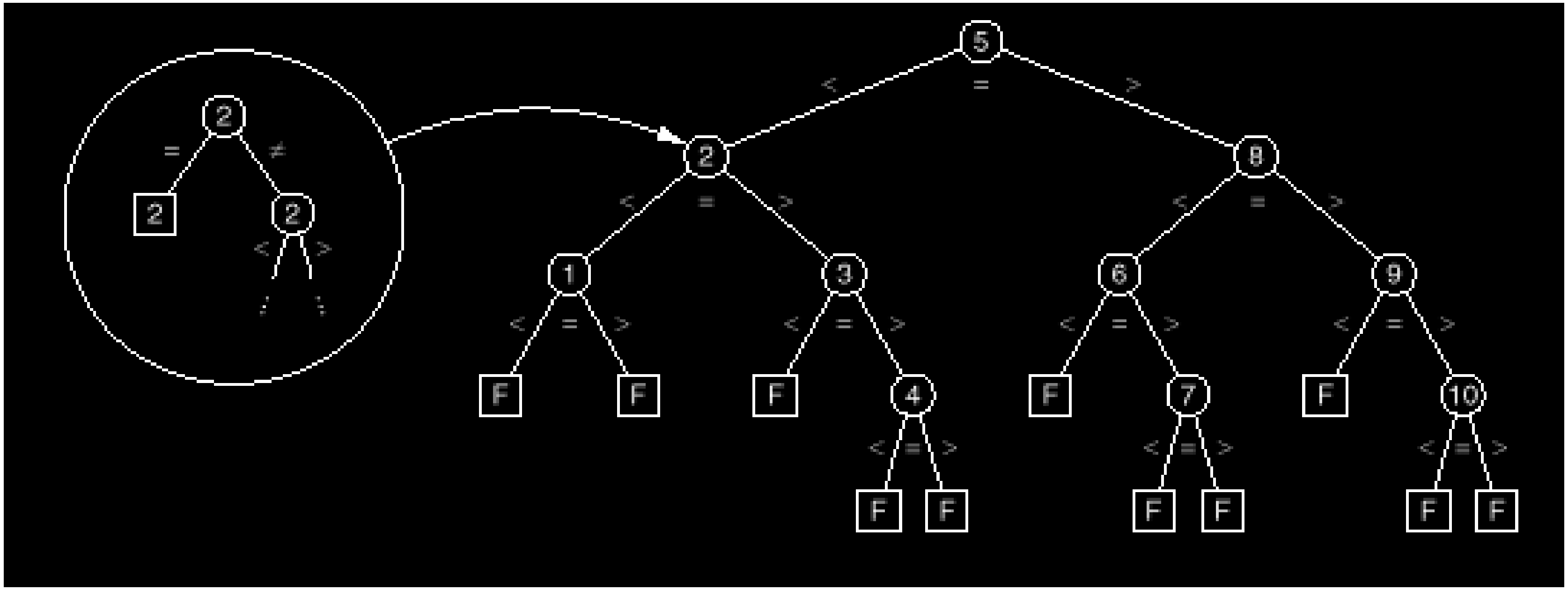
Each branch represents 1 comparison
Biggest height measured in branches represents worst case running time

Comparison Tree for Binary1Search – Iterative solution (with 10 keys)



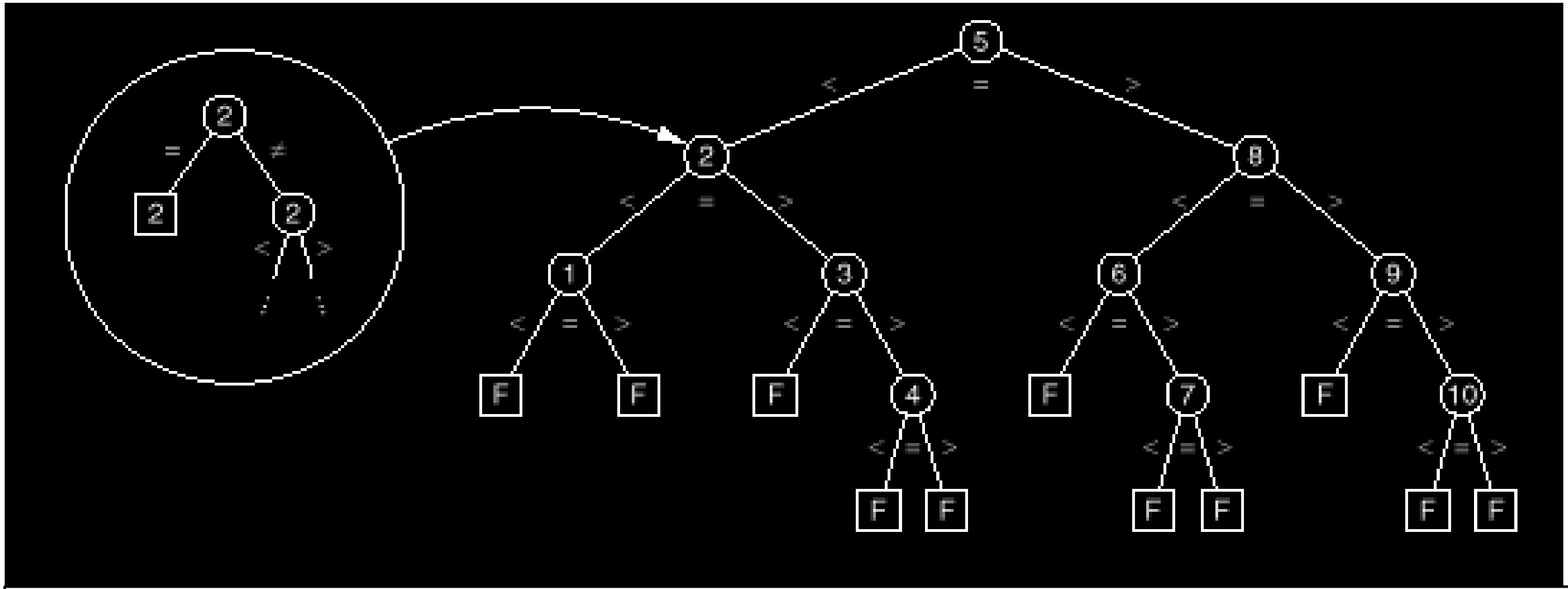
Successful search: (4x5)+(6x4)+(4x5)+(6x4)=88;   44/10 = 4.4
Unsuccessful search: same

Comparison Tree for Binary2Search – Iterative solution (with 10 keys)



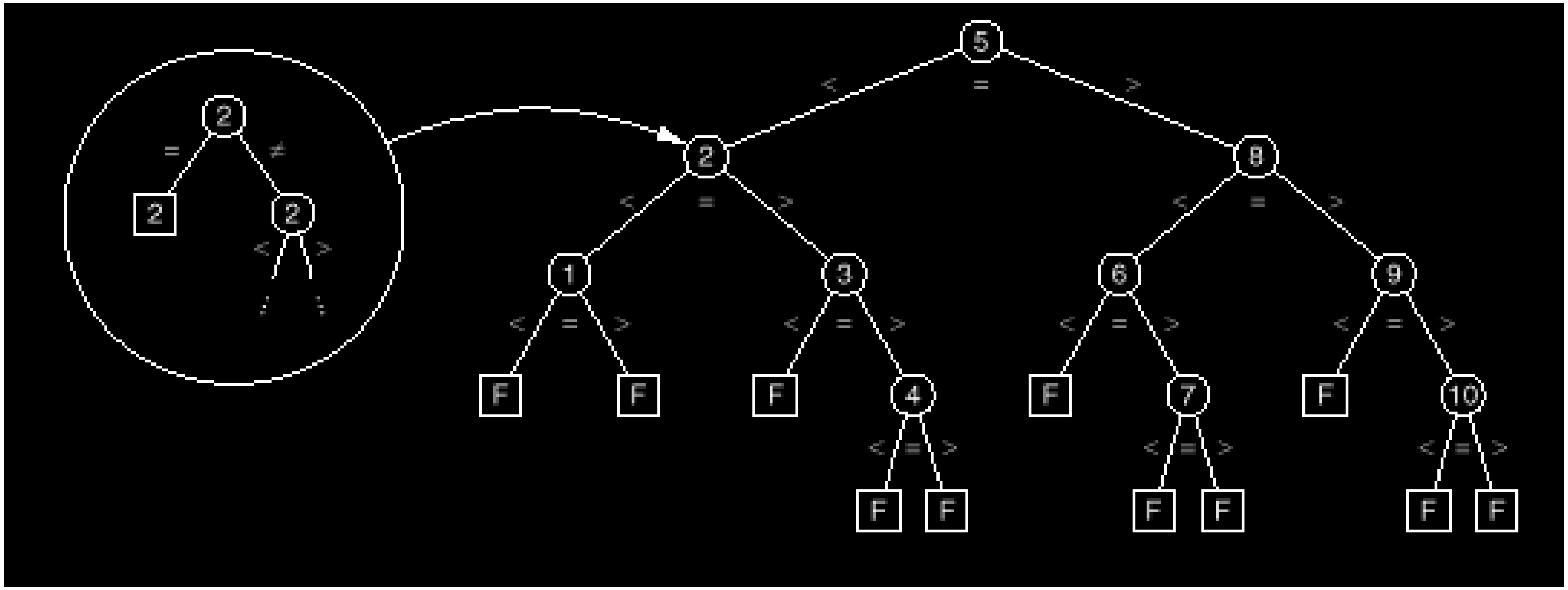
Each node, except for the last successful one, represents 2 comparisons.
Biggest height measured in branched represents worst running time.

Comparison Tree for Binary2Search – Iterative solution (with 10 keys)



Successful search:
number of branches traversed 0+1+2+2+3+1+2+3+2+3=19
number of vertices one more than number of branches, thus for 10 numbers
the average of vertices traversed (19/10)+1 the amount of comparisons 2x((19/10)+1)
one less comparison is done one target is found, there fore
average number of comparisons done 2x((19/10)+1)-1=4.8

Comparison Tree for Binary2Search – Iterative solution (with 10 keys)



Unsuccessful search:

external path length (5x3)+(6x4) = 39

Number unsuccessful tries is n+1: 11

average number of comparisons: 2x39/11 = 7.1