Searching

- **Searching** is a process of retrieving or locating a record with a particular key value
- Searching Techniques
  - Sequential search
  - variant of sequential search- sentinel search,
  - Binary search
  - Fibonacci search

Sequential / Linear Search

- Simplest form of searching.
- Can be applied for sequential storage structures like files, arrays or linked lists.
- In this method, the searching begins from the first record. The required key value is compared with the record key. Searching continues sequentially till the record with a matching key value is found or if the table ends.
- Although not required, we usually consider the list to be unsorted when doing a sequential search, because there are other algorithms that perform better on sorted lists. Sequential search looks at elements, one at a time, from the first in the list until a match for the target is found

Pros & Cons

- ***Advantages :***
  - It is a very simple method.
  - It does not require the data to be ordered.
- ***Disadvantage :***
  - If n is very large, this method is very inefficient and slow

Analysis of Linear Search

- **Best Case :** Record found at first position itself, only 1 comparison is required.
- **Worst Case :** Record not present in the list, n+1 comparisons are required.
- **Average Case :** For all keys Ki (1 <= i < n) the total number of comparisons for successful searches are :

  $$1 + 2 + \dots + n = (n + 1) / 2 \quad = O(n)$$

- Hence the time complexity of this method is O(n).

Sentinel Linear Search

- **Linear Search :**
  - Compare the search item with the elements in the list one by one (using a loop) and stop as soon as we get the first copy of the search element in the list.
  - In Worst case when the search element does not exist in the list of size **N** then the **Simple Linear Search** will take a total of **2N+1** comparisons (**N** comparisons against every element in the search list and **N+1** comparisons to test against the end of the loop condition).
- **Sentinel Linear Search :**
  - Idea is to reduce the number of comparisons required to find an element in a list.
  - Here we replace the last element of the list with the search element itself and run a **while loop** to see if there exists any copy of the search element in the list and quit the loop as soon as we find the search element.

Algorithm

```
1.  Let X be the element to be searched
2.  Let last = A[N-1];
3.  Let A[N-1] = X;
4.  Initialize i to 0;
5.  while(A[i] != X)
    {
       i++;
    }
6.  A[N-1] = last;
7.  if( (i < N-1) || (X == array[N-1]) )
       Display Element Found at ith position;
    else
      Display Element Not Found
```

Analysis

- The **while loop** makes only one comparison in each iteration and it is sure that it will terminate since the last element of the list is the search element itself. So in the worst case (if the search element does not exists in the list) then there will be at most **N+2** comparisons (**N** comparisons in the while loop and **2** comparisons in the if condition). Which is better than (**2N+1**) comparisons as found in **Simple Linear Search**.
- both the algorithms have time complexity of **O(n)**.

Binary Search

- Efficient searching method used for linear / sequential data.
- In this search, Data has to be in the sorted order either ascending or descending.
- Sorting has to be done based on the key values.
- In this method, the required key is compared with the key of the middle record.
- If a match is found, the search terminates.
- If the key is less than the record key, the search proceeds in the left half of the table.
- If the key is greater than record key, search proceeds in the same way in the right half of the table.
- The process continues till no more partitions are possible.
- Thus every time a match is not found, the remaining table size to be searched reduces to half.

Analysis

- ***Advantage :***
  - Time complexity is O(logn) which is very efficient.
- ***Disadvantage :***
  - Data has to be in sorted manner.

- **Analysis:**

After 0 comaparisons → Remaining file size = n

After 1 comaparisons → Remaining file size = n / 2 = $n / 2^1$

After 2 comaparisons → Remaining file size = n / 4 = $n / 2^2$

After 3 comaparisons → Remaining file size = n / 8 = $n / 2^3$

……

After k comaparisons → Remaining file size = $n / 2^k$

The process terminates when no more partitions are possible i.e.

remaining file size = 1

$n / 2^k = 1$

$k = \log_2 n$

Thus, the time complexity is $O(\log_2 n)$. which is very efficient.

Fibonacci Search

- Alternate method to binary search
- Comparison-based technique that uses Fibonacci numbers to search an element in a sorted array.
- Splits the sub-file according to the Fibonacci sequence instead of splitting file by dividing it into half.
- The Fibonacci sequence is
  - ❖1, 1, 2, 3, 5, 8, 13, 21, 34, ……..
  - ❖F1 = 1 , F2 = 1 , Fi = Fi-1 + Fi-2
- Advantage is that it involves only addition and subtraction rather than the division as in binary search.
- Hence, its average performance is better than that of binary search because computer takes more time on an average for division as compared to that of addition or subtraction.

Similarities with Binary Search

- Works for sorted arrays
- A Divide and Conquer Algorithm.
- Has Log n time complexity.

Differences with Binary Search

- Fibonacci Search divides given array in unequal parts
- Binary Search uses division operator to divide range. Fibonacci Search doesn’t use /, but uses + and -. The division operator may be costly on some CPUs.
- Fibonacci Search examines relatively closer elements in subsequent steps. So when input array is big that cannot fit in CPU cache or even in RAM, Fibonacci Search can be useful.

- **Observations:**
  Below observation is used for range elimination, and hence for the O(log(n)) complexity.
- $F(n - 2) \approx (1/3)*F(n)$ and
- $F(n - 1) \approx (2/3)*F(n)$.

Technique

- Idea is it to first find the smallest Fibonacci number that is greater than or equal to length of given array.
- Let the found fibonacci number be fn.
- We use f(n-2)th Fibonacci number as index (If it is a valid index). Let (n-2)’th Fibonacci Number be i, we compare arr[i] with x, if x is same, we return i. Else if x is greater, we recur for subarray after i, else we recur for subarray before i.

Algorithm

Let arr[0..n-1] be th input array and element to be searched be x.

1. Find the smallest Fibonacci Number greater than or equal n. Let this number be fibM **(f3)**[m'th Fibonacci Number]. Let the two Fibonacci numbers preceding it be fibMm1 (f2) [(m-1)'th Fibonacci Number and fibMm2 (f1)[(m-2)'th Fibonacci Number.
2. While the array has elements to be inspected:
    a) Compare x with the last element of the range covered by fibMm2 (f1)
    b) If x matches, return index
    c) Else If x is less than the element, move the three Fibonacci variables two Fibonacci down, indicating elimination of approximately rear two-third of the remaining array.
    d) Else x is greater than the element, move the three Fibonacci variables one Fibonacci down. Reset offset to index. Together these indicate elimination of approximately front one-third of the remaining array.
3. Since there might be a single element remaining for comparison, check if fibMm1 **(f2)** is 1. If Yes, compare x with that remaining element. If match, return index.

Algorithm

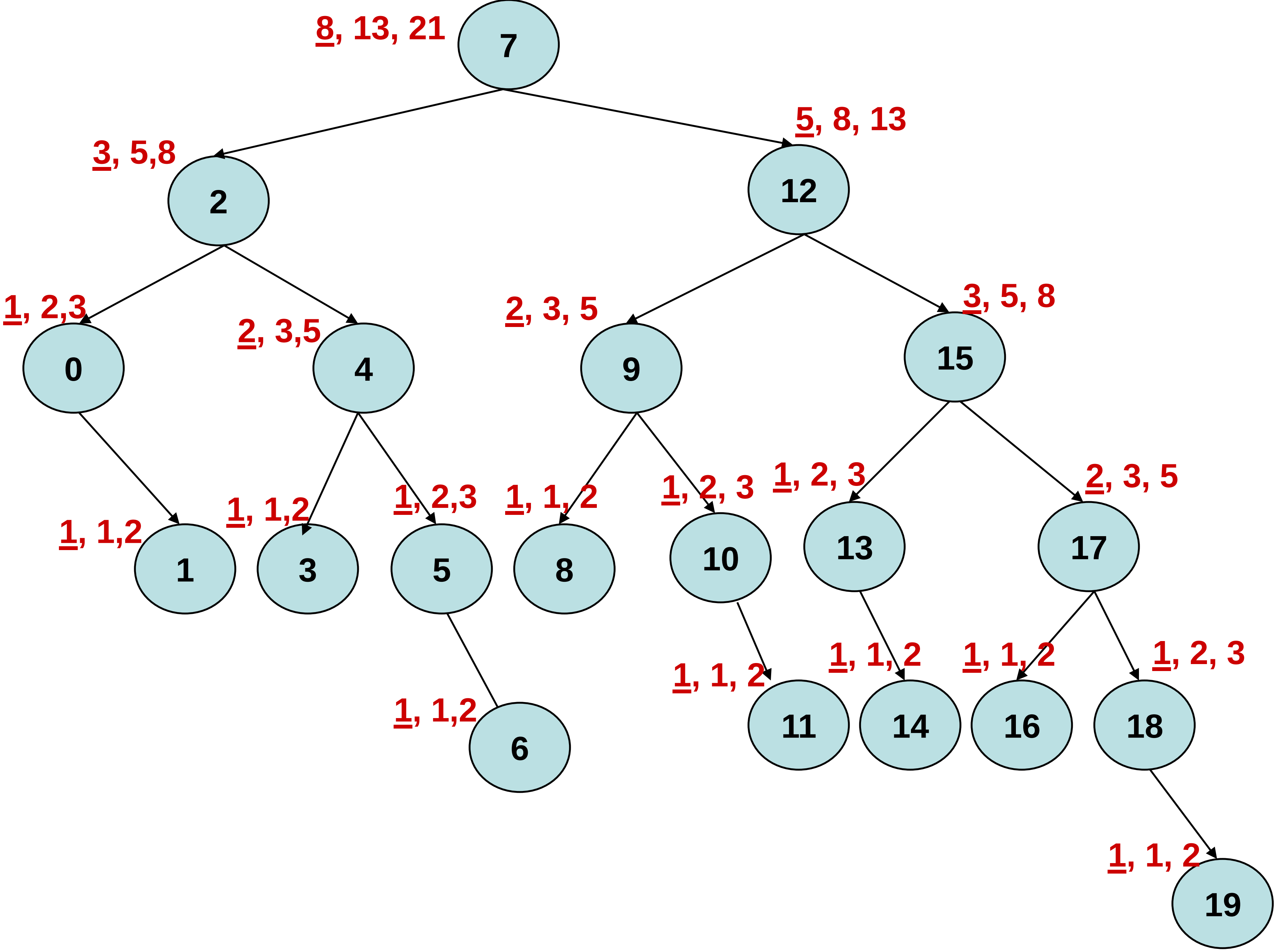
```
int Fibonacci_Search(int A[], int n, int X)
{
    int f1 = 0, f2 = 1;
    int f3 = f1 + f2;
      int i, offset = -1;
    while (f3 < n)
    {
        f1 = f2;
        f2 = f3;
        f3  = f1 + f2;
    }
    while (f3 > 1)
    {
        i = min(offset+f1, n-1);
                if(A[i] == X)
                                return i;  // Found
Else {
                if (X < A[i] )  // left subarray (66 % or 2/3 array)
                                {
                                        f3  = f1;
                                        f2 = f2 - f1;
                                        f1 = f3 - f2;
                                }
                                else                    // right subarray ( 33 % or 1/3 array)
                                {
                                        f3  = f2;
                                        f2 = f1;
                                        f1 = f3 - f2;
                                        offset = i;
                                }
                }
    }

      if(f2 == 1 && A[offset+1] == X)
                return offset+1;   // Found

    return -1; // NOT FOUND
}
```

Fibonacci Search tree

- It will be easy to understand this algorithm by constructing a binary tree as below.
- This tree is a Fibonacci tree.
- The values in the nodes show how the file will be broken up during the searching process.


8, 13, 21
7
5, 8, 13
3, 5,8
12
2
1, 2,3
2, 3,5
2, 3, 5
3, 5, 8
0
4
9
15
1, 2, 3
2, 3, 5
1, 1,2
1, 1,2
1, 2,3
1, 1, 2
1, 2, 3
1
3
5
8
10
13
17
1, 1, 2
1, 1, 2
1, 1, 2
1, 2, 3
1, 1,2
11
14
16
18
6
1, 1, 2
19

Analysis

- Time complexity for this algorithm depends on the height of binary tree having n nodes.
- Therefore it is **O(log n)** .
- Pros and Cons of Fibonacci search
  - ➢It is **O(log n)** algorithm
  - ➢Inefficient for small lists.
  - ➢Efficient than binary search for large size lists.

Time Complexity analysis:

- The worst case will occur when we have our target in the larger (2/3) fraction of the array, as we proceed finding it.
- In other words, we are eliminating the smaller (1/3) fraction of the array every time.
- We call once for n, then for(2/3) n, then for (4/9) n and henceforth.

$$fib(n) = \left[\frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^{n} \right] \sim c * 1.62^{n}$$

for $n \sim c * 1.62^{n'}$ *we make* $O(n')$ *comparisons. We, thus, need* $O(\log(n))$ *comparisons.*