SE 2228-Analysis and Design of Algorithms

Algorithm Analysis-2

Iterative/Recursive Algorithms

Computing and Simplifying Running Time

- Method: Associate a "cost" with each statement and find the "total cost“ by finding the total number of times each statement is executed.
- Express running (Execution) time in terms of the size of the problem.
- Although some algorithms may have different cost values, their asymptotic complexity expression may be the same.

Example $T1(n) = c1*n$ and $T2(n) = (c1+c2)*n$

Both are $O(n)$ algorithms.

General Rules for Running Time Estimation

- Loops: The running time of a loop is at most the running time of the statements inside of that loop times the number of iterations.
- Nested Loops: Running time of a nested loop containing a statement in the inner most loop is the running time of statement multiplied by the product of the sizes of all loops.
- Consecutive Statements: Just add the running times of those consecutive statements.
- If/Else: Never more than the running time of the test plus the larger of running times of S1 and S2.

Computing Execution Time of Algorithms

- Each operation in an algorithm (or a program) has a cost.

  →Each operation takes a certain of time.

count = count + 1;

takes a certain amount of time, but it is constant

A sequence of operations:

count = count + 1; Cost: c_1

sum = sum + count; Cost: c_2

Total Cost = $c_1 + c_2$

Computing Execution Time of Algorithms

Example: Simple If-Statement

	Cost	Times
`if (n < 0)`	c1	1
`    absval = -n`	c2	1
`else`		
`    absval = n;`	c3	1

Total Cost <= c1 + max(c2,c3)

Example1: Simple Loop

Code	Cost	Times
`i = 1;`	c1	1
`sum = 0;`	c2	1
`while (i <= n) {`	c3	n+1
`   i = i + 1;`	c4	n
`   sum = sum + i;`	c5	n
`}`		

Total Cost = c1 + c2 + (n+1)*c3 + n*c4 + n*c5

➔ The time required for this algorithm is proportional to n

Computing Execution Time of Algorithms

Example-2:Nested loop

Code	Cost	Times
`i=1;`	c1	1
`sum = 0;`	c2	1
`while (i <= n) {`	c3	n+1
`    j=1;`	c4	n
`    while (j <= n) {`	c5	n*(n+1)
`        sum = sum + i;`	c6	n*n
`        j = j + 1;`	c7	n*n
`    }`		
`    i = i +1;`	c8	n
`}`		

$$T(n) = c1 + c2 + (n+1)*c3 + n*c4 + n*(n+1)*c5+n*n*c6+n*n*c7+n*c8$$

$$= (c5+c6+c7)*n^2 + (c3+c4+c5+c8)*n + (c1+c2+c3)$$

$$= a*n^2 + b*n + c$$

➔ The growth-rate function for this algorithm is **$O(n^2)$**

Asymptotic Analysis

- Using *rate of growth* as a measure to compare different functions implies comparing them asymptotically.
- If $f(x)$ is *growing faster* than $g(x)$, then $f(x)$ always eventually becomes larger than $g(x)$ in the limit (for large enough values of x).

How to express rate of growth for different algorithm types?

We consider two main categories of algorithms:

- Iterative (nonrecursive)Algorithms
- Recursive Algorithms

Iterative Algorithms

- Iterative algorithms take one step at a time towards the final destination (Solution).

→ Repeat operations until some condition becomes true or false.

General structure:

```
loop (until done)
    take steps
end loop
```

Iterative algorithms :Element Uniqueness

Problem:Check whether all the elements in a given array are distinct.

```
Unique_Elements (A)
//Input: An array A [0..n - 1]
//Output: Returns "true" if all the elements in A are distinct and "false"
otherwise.
  for i ← 0 to n - 2 do
     for j ← i + 1 to n - 1 do
        if A[i] = A [j]
        return false   //Not distinct
  return true
```

Analysis of Element Uniqueness

Worst case number of comparisons :

T(n) $=\sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$

$=\sum_{i=0}^{n-2}(n-1-(i+1)+1)$

$=\sum_{i=0}^{n-2} n-1-i$

$=\sum_{i=0}^{n-2}(n-1)- \sum_{i=0}^{n-2} i$

simplifies to:

= n(n-1)/2

$=O(n^2)$

Euclid's algorithm : Iterative Version

- A method for finding the greatest common divisor (GCD) of two integers m and n.

Example : let m=210 , n= 45 .

GCD(210,45)=15

→15 is the greatest number that divides both 210 and 45.

The algorithm is based on the following fact: for m>n

GCD (m , n)

= GCD (n, m mod n)

=GCD (n, m%n)

% is the mod operator : Returns the remainder after dividing m by n

Example : gcd(60,24) = gcd(24,60%24)

=gcd(24,12)

= gcd(12,0)

= 12

Euclid's Algorithm: Iterative Version

//Also known as Euclidean Algorithm.Non-recursive implementation)

```
long gcd ( long m, long n)
{
  long r;
  while (n != 0)
        {
            r = m % n;
            m = n;
            n  = r;
        }
   return m;
   }
```

Tracing Euclid's Algorithm

m	n	r
210	45	30
45	30	15
30	15	0
15	0	Done .

Return m=15.

GCD(210,45) =15

Time Complexity of Euclide -1

- How many times the loop iterates. This is the same question as:

  How many times can we apply the mod operation?

- Each iteration reduces the first argument of the mod operation:

  In the example we had : 210,45,30,15

- What can be the maximum number of iterations?
- Detailed analysis involves different possibilities . We are going to consider a simple approach.

Time Complexity of Euclid -2

Consider the simple fact: GCD(M,N)

If M > N, then (M % N) < M/2. (r < M/2)

→ The size of M mod N is strictly less than M/2.

- So how many times % is applied for the remainder to be zero (r = 0)?

→The number of times mod operations is performed < the number of times M can be divided by 2 .

→ This can not be more than $\log_2(M)$.

→ Worst case Time complexity of Euclid's algorithm is $O(\log_2 M)$.

- Best case : O(1),If at the start the remainder is 0. Ex: (100,50)

When the worst case occurs ?

One possibility :If N and M are two consecutive Fibonacci numbers (Check this!).

Example : Insertion Sort

- Take multiple passes over the list
- Keep already sorted part at low-end
- Find next unsorted element
- Insert it in correct place, relative to the ones already sorted
- Invariant**:** each pass increases the size of sorted portion.

Reminder : Insertion Sort Algorithm

```
INSERTION-SORT(A,n)
  for j ← 2 to n do
        key ← A[ j ]
        // Insert A[ j ] into the sorted sequence A[1 . . j -1]
          i ← j - 1
          while i > 0 and A[i] > key do
                A[i + 1] ← A[i]
                  i ← i – 1
          A[i + 1] ← key
```

Insertion Sort: How it Works ?

Insertion sort sorts the elements in place

input array

A: 5 2 4 6 1 3

at each iteration, the array is divided in two sub-arrays,the left subarray keeps growing

left sub-array right sub-array

2 5 | 4 6 1 3

sorted unsorted

Complexity of Insertion Sort

- The body is made up of two nested for loops. The outer loop is executed n – 1 times.
- The inner loop is harder to analyze because the number of times it executes depends on how many keys in positions 1 to i – 1 have a value less than the key in position i.
- In the worst case, each record must make its way to the start of the array. This would occur if the keys are reversely sorted.
- In this case, the number of comparisons will be 1, the first time through the loop, 2 the second time, and so on. Thus, the total number of comparisons will be

$$T(n) = \sum_{i=2}^{n} i = 2+3+\ldots..+(n-1) = \Theta(n^2)$$

Complexity of Insertion Sort

- Consider the best-case . This occurs when the keys begin in sorted order. In this case, every pass through the inner loop will fail immediately, and no values will be moved.
- The total number of comparisons will be the number of times the outer loop executes:

$$T(n) = \sum_{i=2}^{n} 1 = 1+1+++++1$$

$$=(n-2+1) *1 = n-1$$

$$=\Theta(n)$$

- Average case : We expect on average that half of the keys in the first i − 1 array positions will have a value greater than that of the key at position i. Thus, the average case should be about half the cost of the worst case:

$[n(n+1)/2] /2 = O(n^2/4)$, which is still $\Theta(n^2)$.

→Asymptotically, the average case is no better than the worst case

Summary : Explaining Different Cases

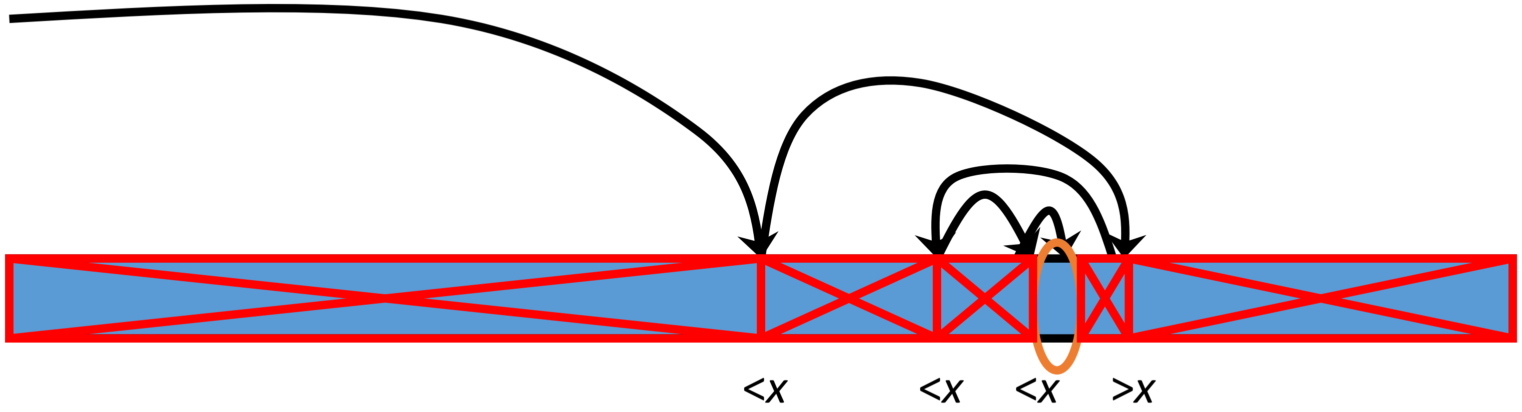
- The worst-case occurs if the array is sorted in reverse order i.e., in decreasing order:
  - We always find that $A[i]$ is greater than the key in the while-loop test. So, we must compare each element $A[j]$ with each element in the entire sorted subarray $A[1 \ldots j-1]$
- The best case occurs if the array is already sorted:
  - For each $j = 2, 3, \ldots, n$, we find that $A[i]$ less than or equal to the key when i has its initial value of $(j-1)$. In other words, when $i = j-1$, always find the key $A[i]$ upon the first time the WHILE loop is run.
- The average case is also quadratic, which makes insertion sort impractical for sorting large arrays.

Generic Insertion Sort Function: C++

```
template <class T>
void InsertionSort(T A[], int n) {
  for (int i = 1; i < n; ++i) {
       if (A[i] < A[i-1]) {
          T val = A[i];
          int j = i;
          do {  A[j] = A[j-1];
             --j;
          } while ((j > 0) && (val < A[j-1]));
          A[j] = val;
       }
  }
}
```

Binary Search(Iterative Version)

- Basic idea: On each step, look at the *middle* element of the remaining list to eliminate half of it, and quickly locate the desired element.



Reminder : Binary Search Algorithm

```
//Input A: a1, a2, …, an: distinct integers.Search key : x
binary search(A,n,x)
   i ← 1                     //left endpoint of search interval
   j ← n                     //right endpoint of search interval
   while i<j   do
   mid ← ⌊(i+j)/2⌋        //midpoint
            if x>a_mid then
                i ← mid+1
            else
                j ← mid

         if x = a_i then      //Found
             location ← i
             else location ←0  //Not found
   return location
```

Binary Search Complexity

- Best Case: match from the first comparison: O(1)
- Worst Case: divide until reach one item, or no match
- For an array of size N, it eliminates ½ (half) until 1 element remains.

  N, N/2, N/4, N/8, ..., 4, 2, 1

  - How many divisions does it take?
- Think of it from the other direction:
  - How many times do I have to multiply by 2 to reach N?

    1, 2, 4, 8, ..., N/4, N/2, N
  - Call this number of multiplications "x".

    $2^x = N$

    $x = \log_2 N$

→Binary search is in the logarithmic complexity class.

Worst case Analysis : Recurrence Relation

Item not in the array (size N)

$T(N)$ = number of comparisons with array elements

$T(1) = 1$

$$
\begin{aligned}
T(N) &= 1 + T(N / 2) \\
&= 1 + [1 + T(N / 4)] \\
&= 2 + T(N / 4) \\
&= 2 + [1 + T(N / 8)] \\
&= 3 + T(N / 8) \\
&= \ldots \\
&= k + T(N / 2^k) \qquad [1]
\end{aligned}
$$

Worst-case Analysis : Recurrence

$T(N / 2^k)$ gets smaller until the base case: $T(1)$.

Assume

$2^k = N$

$k = \log_2 N$

Replace terms with k in [1]:

$$T(N) = \log_2 N + T(N / N)$$
$$= \log_2 N + T(1)$$
$$= \log_2 N + 1$$

→ "$\log_2 N$" algorithm

Is Binary Search more efficient?

- For a list of N elements, Binary Search can execute at most $\log_2 N$ times.
- A disadvantage of Binary Search is the overhead of maintaining the list in ordered form.

<u>Number of Iterations</u>

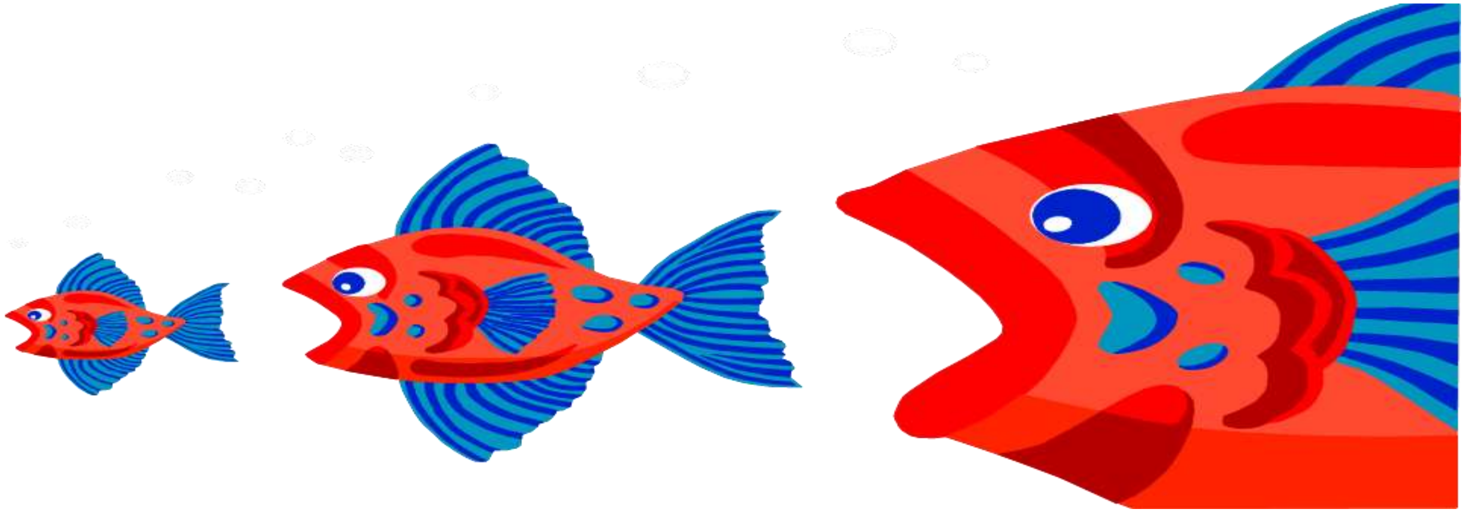
N	Linear Search(Av)	Binary Search(Worst)
10	5	3
100	50	6
1,000	500	9
1,000,000	500000	20

Binary Search: C++ Implementation

```
int binarySearch(int[] a, int target)
 {
    int min = 0;
    int max = a.length - 1;
    while (min <= max) {
        int mid = (min + max)/2;
        if (a[mid] < target)
            min = mid + 1;
         else if (a[mid] > target)
            max = mid - 1;
            else
            return mid;    // target found
         }

    return -1;     // target not found
 }
```

Recursive Algorithms



"To iterate is human, to recurse is divine!"

Recursion vs. Iteration

- Repetition
  - Iteration: explicit loop
  - Recursion: repeated function calls
- Termination
  - Iteration: loop condition fails
  - Recursion: base case recognized
- Both can have infinite loops
- Recursive solutions can be easier to understand and to describe than iterative solutions.

Example -1 : Recursive Euclid algorithm

```
//Base case n=0
long gcd( long m, long n)
   {  if ( n == 0 )
        return m
      else
        return gcd( n, m % n) }
```

Worst case complexity is the same as the iterative version : O(logn)

Example-2: Recursive *n*!

Recursive algorithm for *n*!

Input size: *n*, Basic operation: multiplication "*"

- Analysis :Let $M(n)$ be the number of multiplications needed to compute *n*! , then the recurrence can be expressed as:

$$M(0) = 0$$

$$M(n) = M(n-1) + 1 \quad \textbf{for } n > 0$$

To compute *Factorial*(*n*-1)   To multiply *Factorial*(*n*-1) by *n*

ALGORITHM $Factorial(n)$

if $n = 0$

return 1

else

return $Factorial(n-1) * n$

Analysis : Solving the Recurrence

$$M(0) = 0$$

$$M(n) = M(n-1) + 1 \ \textbf{for} \ n > 0$$

$$\Rightarrow M(n)$$

$$= M(n-2) + 2$$

$$= M(n-3) + 3$$

$$= \ldots$$

$$= M(n-n) + n$$

$$= n$$

→ The complexity of this algorithm is Ө(n)
(Both best and worst cases)

Stack Frames in Recursion

- A stack is used to remember parameters and local variables across recursive calls.
- Each recursive invocation gets its own stack frame
- A stack frame contains storage for the current call :
  - Local variables
  - Parameters
  - Return info (return address and return value)
  - Other bookkeeping info
- A new stack frame is pushed with each recursive call
- The stack frame is popped when the function returns

Stack Frames in Recursion

- When a new stack frame is created, previous stack frames are left in the stack (In memory) .
- For example,in fact(3) the recursive function is called 3 times, so there will be 3 stack frames pushed on the stack.
- Each frame stores one instance and holds different values.
- As usual, the first stack frame is at the bottom and the last frame is at the top. The order in which the statements run is denoted by the order of stack frames.

Tracing the call fact(3): Stack Frames

After original call

N = 3
if (N==0) false
return (3***fact(2)**)

After 1st call

N = 2
if (N==0) false
return (2***fact(1)**)

N = 3
if (N==0) false
return (3***fact(2)**)

After 2nd call

N = 1
if (N==0) false
return (1***fact(0)**)

N = 2
if (N==0) false
return (2***fact(1)**)

N = 3
if (N==0) false
return (3***fact(2)**)

After 3rd call

N = 0
if (N==0) true
return (1)

N = 1
if (N==0) false
return (1***fact(0)**)

N = 2
if (N==0) false
return (2***fact(1)**)

N = 3
if (N==0) false
return (3***fact(2)**)

Tracing the call fact(3): Stack Frames

After return from 3rd call	After return from 2nd call	After return from 1st call	return **6**
N = 1 if (N==0) false return (1* **1**)			
N = 2 if (N==0) false return (2***fact(1)**)	N = 2 if (N==0) false return (2* **1**)		
N = 3 if (N==0) false return (3***fact(2)**)	N = 3 if (N==0) false return (3***fact(2)**)	N = 3 if (N==0) false return (3* **2**)	

Example-3 : Recursive Fibonacci

Fibonacci sequence : 0,1, 1, 2, 3, 5, 8, 13, 21, 34, …..

```
// Recursively calculates nth Fibonacci number
long fib (int n)
 {
   if( n <= 1 )
       return n
       else
         return fib(n – 1) + fib(n – 2);
 }
```

This is a straightforward, but inefficient recursion ...

Analysis of Recursive Fibonacci Algorithm

- Let n_k denote the number of function calls made by Fib(n).

$n_0 = 1$ initial call

$n_1 = 1$ # calls

$n_2 = n_1 + n_0 + 1 = 1 + 1 + 1 = 3$

$n_3 = n_2 + n_1 + 1 = 3 + 1 + 1 = 5$

$n_4 = n_3 + n_2 + 1 = 5 + 3 + 1 = 9$

$n_5 = n_4 + n_3 + 1 = 9 + 5 + 1 = 15$

$n_6 = n_5 + n_4 + 1 = 15 + 9 + 1 = 25$

$n_7 = n_6 + n_5 + 1 = 25 + 15 + 1 = 41$

$n_8 = n_7 + n_6 + 1 = 41 + 25 + 1 = 67.$

- Note that the value at least doubles for each two consecutive indices. Example : n_4 is more than twice n_2 (9>2*3)….

→ $n_k > 2^{k/2}$ [Ex: $n_8 > 2^{8/2}$]

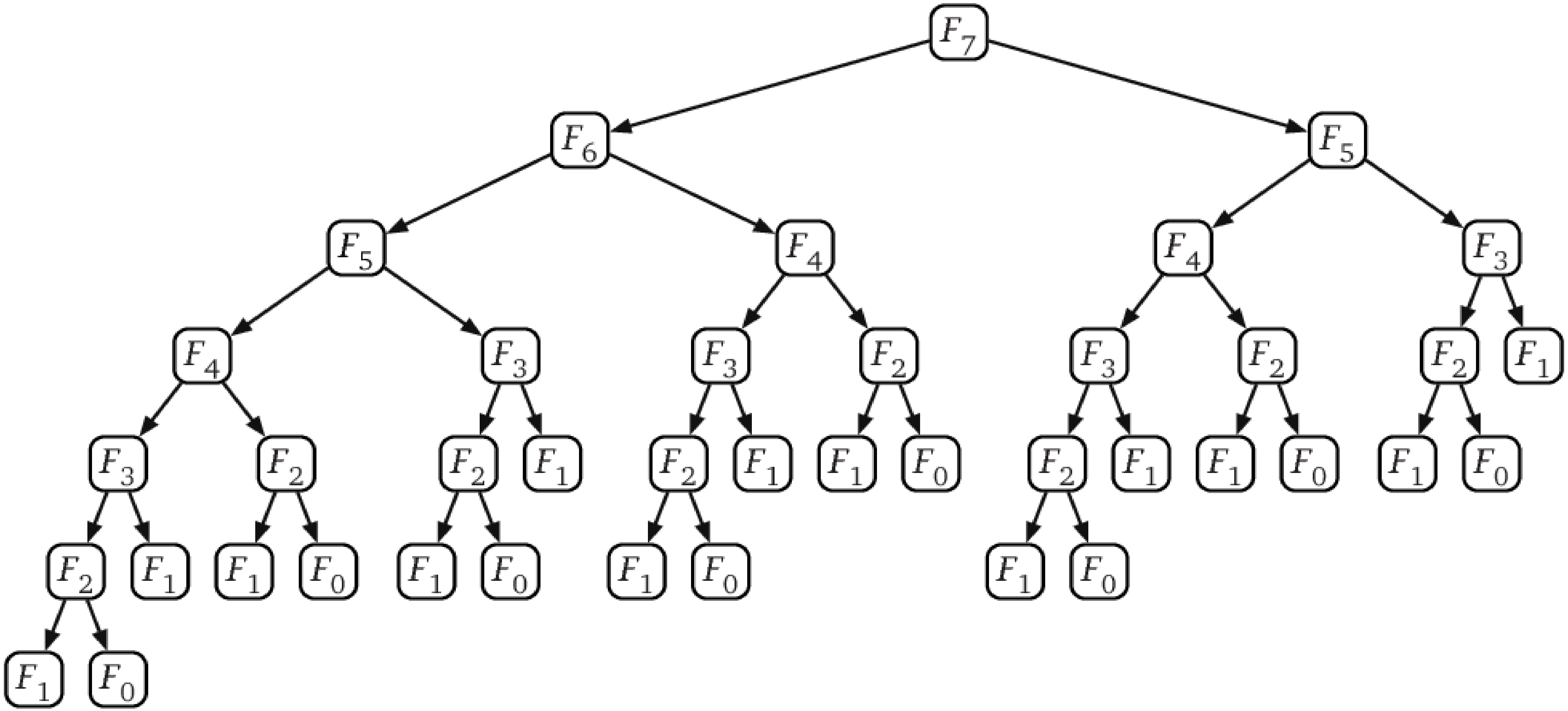
→An exponential time algorithm!

Why again ? If not convinced , See the following recursion tree!

The recursion tree for computing F7 ; arrows represent recursive calls.

Growth of the tree is very fast

Notice repeated computations of the same value : F(0),F(1),F(2),F(3)…



Example-4 Recursive Binary Search

```
RecbinSearch ( A, first, last, x)
  if ( first <= last )
    mid = (first + last) / 2;     // mid is int.
      if  x =A(mid )    // found it!
        return mid;
        else if  x < A(mid)   // must be in 1st half
                 return RecbinSearch( A, first, mid-1, x);
                else          // must be in 2nd half
                  return RecbinSearch(A, mid+1, last, x);
  return -1;  //Not found
```

- No loop! Recursive calls takes its place.

Analysis of Recursive Binary Search

- The recurrence is the same as in Iterative algorithm
  (See the analysis slides for iterative algoritm)

$T(N)$ = Number of comparisons with array elements

$T(1) = 1$

$T(N) = 1 + T(N / 2)$

…………………

$= \log_2 N + 1$

$= O(\log_2 N)$

The complexity is logarithmic.

But actual run time is bigger in recursive binary search.

Is Recursion Always Good ?

- Recursion works best when the algorithm and/or data structure that is used naturally supports recursion.
- Recursive solutions are clearer, simpler, shorter, and easier to understand than their non-recursive counterparts.
- From a practical software engineering point of view these are important benefits, reducing the cost of maintaining the software.
- However, recursive algorithms run slower than their iterative counterparts. Why?
  - When a recursive call is made, it takes time to build a stackframe for the call and push it onto the system stack. This also takes time.
  - Conversely, for every return , the stackframe must be popped from the stack and the local variables must be reset to their previous values – this also takes time.
- Therefore, if the recursion is deep, say, factorial(50) or Fibonacci(20) , we may run out of memory!

Multiway/ Mway Trees

What is a multiway tree ?

- A General non-binary tree which can be used as a search tree.
- A multiway tree differs from a binary tree in a few key ways:
  - Each node has *m* children → There are m pointers
  - Each node has *m-1* keys (Data values)
  - In nodes,keys are in ascending order :

    K1<K2<,…..,<Km-1

Why Multiway Trees ?

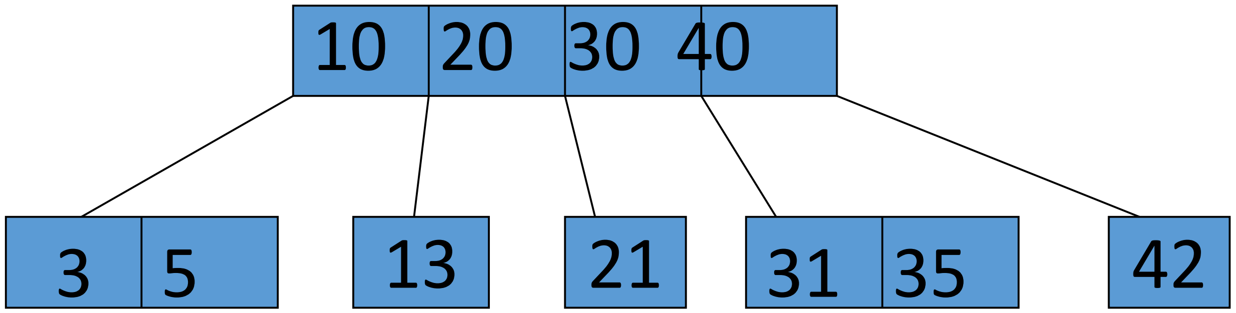
Secondary Memory vs RAM

- We generally assume that the data would be stored in primary memory or RAM.
- Suppose the data is too big to store in RAM, so needs to be stored on a hard disk.
- Access time for hard disk includes;
  - *seek time + rotation time + transfer time*
- The seek time is particularly slow as it depends on mechanical movement : disk head physically moving to the correct position.
- The seek time is in the order of milliseconds, while CPU processes are in the order of sub-microseconds (thousands of times faster).

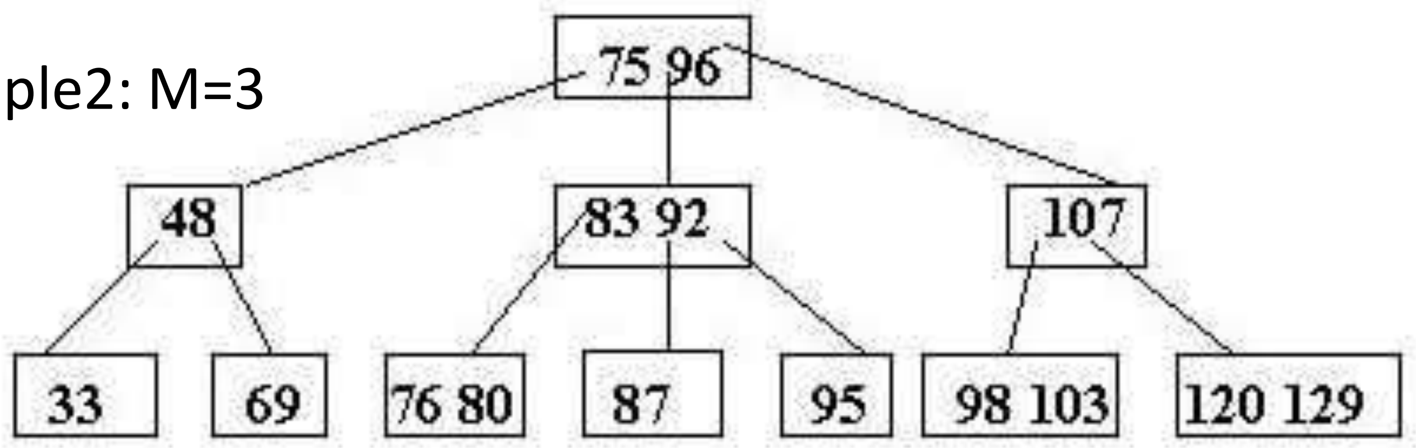
Multiway /M-ary Tree:Examples

M is called the order of tree.

Example1 : M=5→ Maximum 5 pointers,4 keys.



Example2: M=3

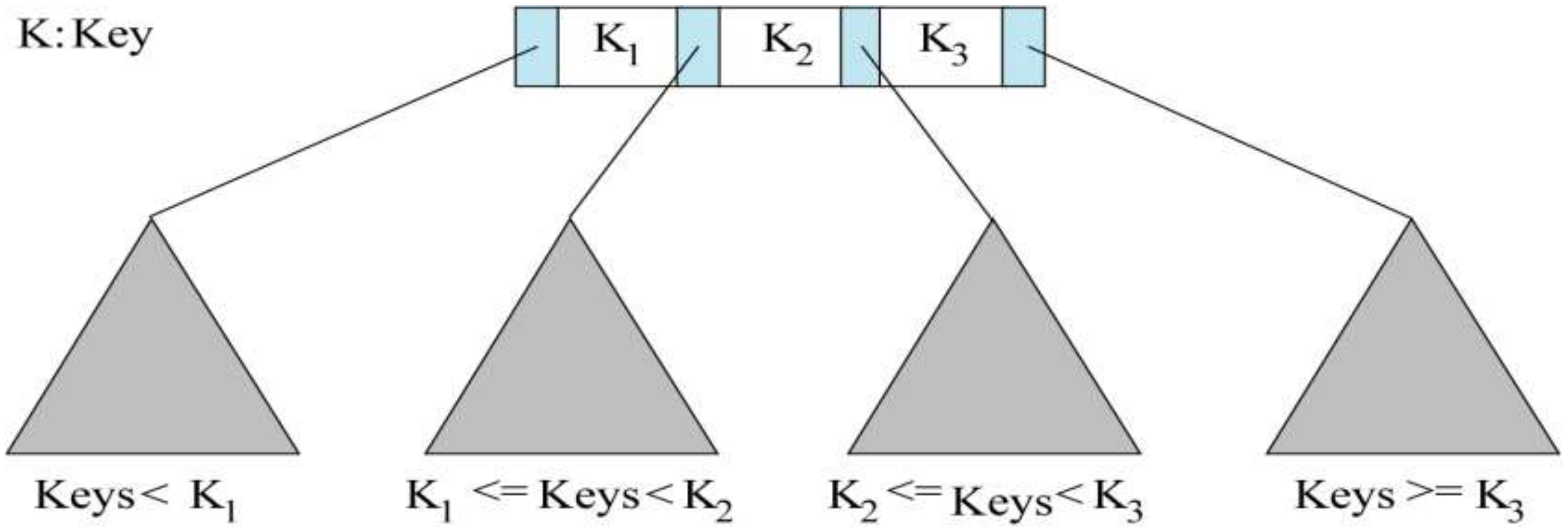


Properties of Multiway Trees

Given the nonempty multiway tree, the following properties hold:

1. Each node has 0 to m subtrees(children).
2. The keys in the first i children are smaller than the i th key.
3. The keys of the data entries are ordered in each node.

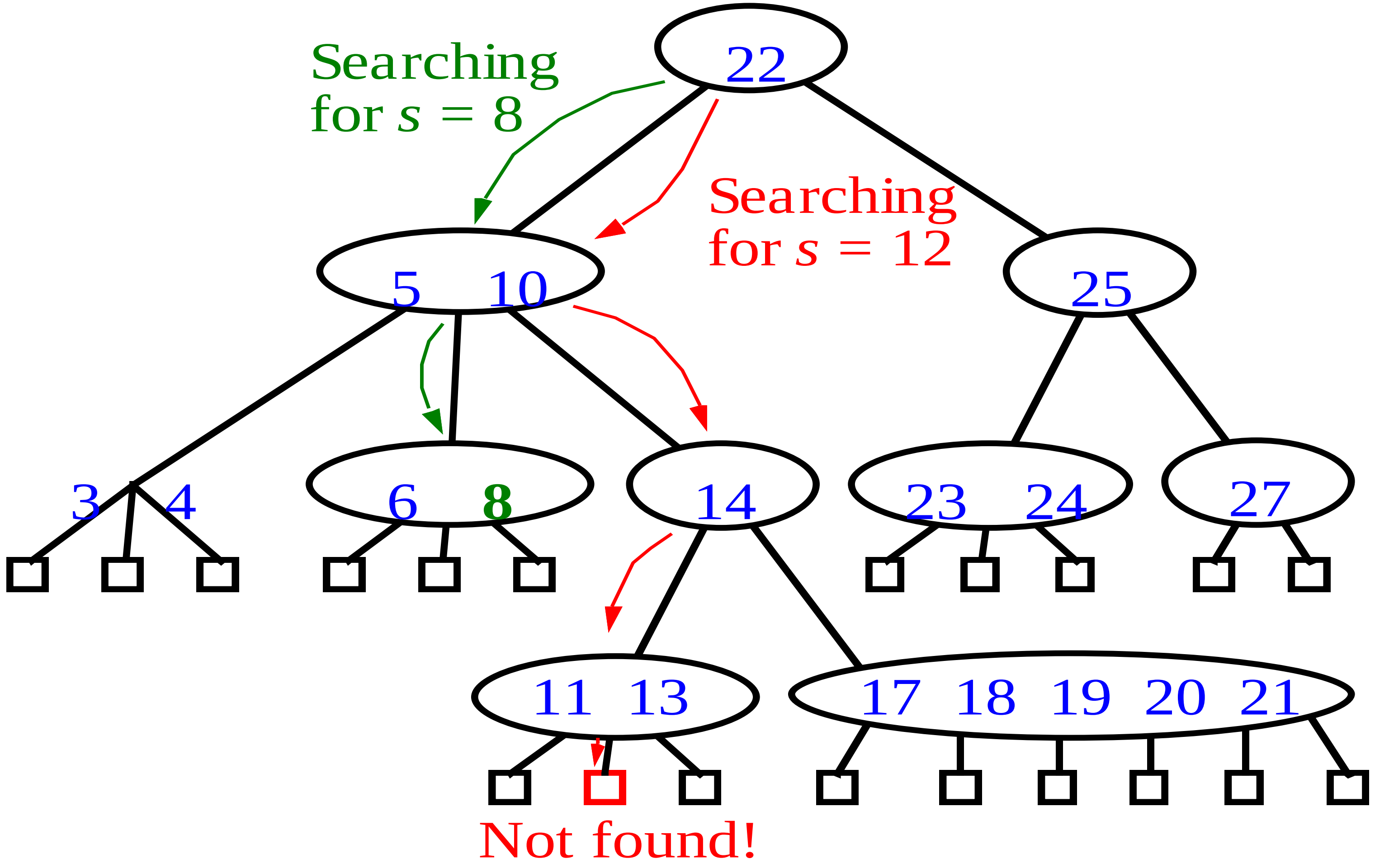
$m = 4$, K: Key



Searching an *m*-way Search Tree

- Suppose we search an *m*-way search tree *T* for the key value *x*. By searching the keys of the root, we determine *i* such that $K_i \leq x < K_{i+1}$.
  - If $x = K_i$, the search is complete.
  - If $x \neq K_i$, x must be in a subtree A_i
  - We proceed to search *x* in subtree A_i and continue the search until we find *x* or determine that *x* is not in *T*.

Search: Example-1



B-Trees and B+ Trees

- B-Trees are balanced m-ary trees with two types of nodes:
  - Leaf Nodes and Internal Nodes.
  - Node capacities and total number of nodes can be very high.
- B-trees are dynamic index structures that adjust automatically to inserts and deletes.
- B+ trees are most widely used version of dynamic B-tree index structures.We are going to consider B+ trees.

Properties of B+ Trees

- Can be used to form Generalized,dynamic multilevel indexes.
- Balanced Tree: The same height for paths from root to leaf.
- Interior nodes contain key values and leaf nodes contain key values and pointers to records in files.
- Parameter: n → Max #of pointers in a node.

Every node contains:

Key values : Ki,

Pointers : Pi

B+ tree: Parameter

- Given the parameter n (Called tree order):

  -Min # of keys in a node: ceil ((n/2)-1),

  Max # of keys in a node: n-1

  -Min # of ptrs : ceil(n/2) , Max # of ptrs: n

  Example : n=5

  #of keys in each node: 2 to 4

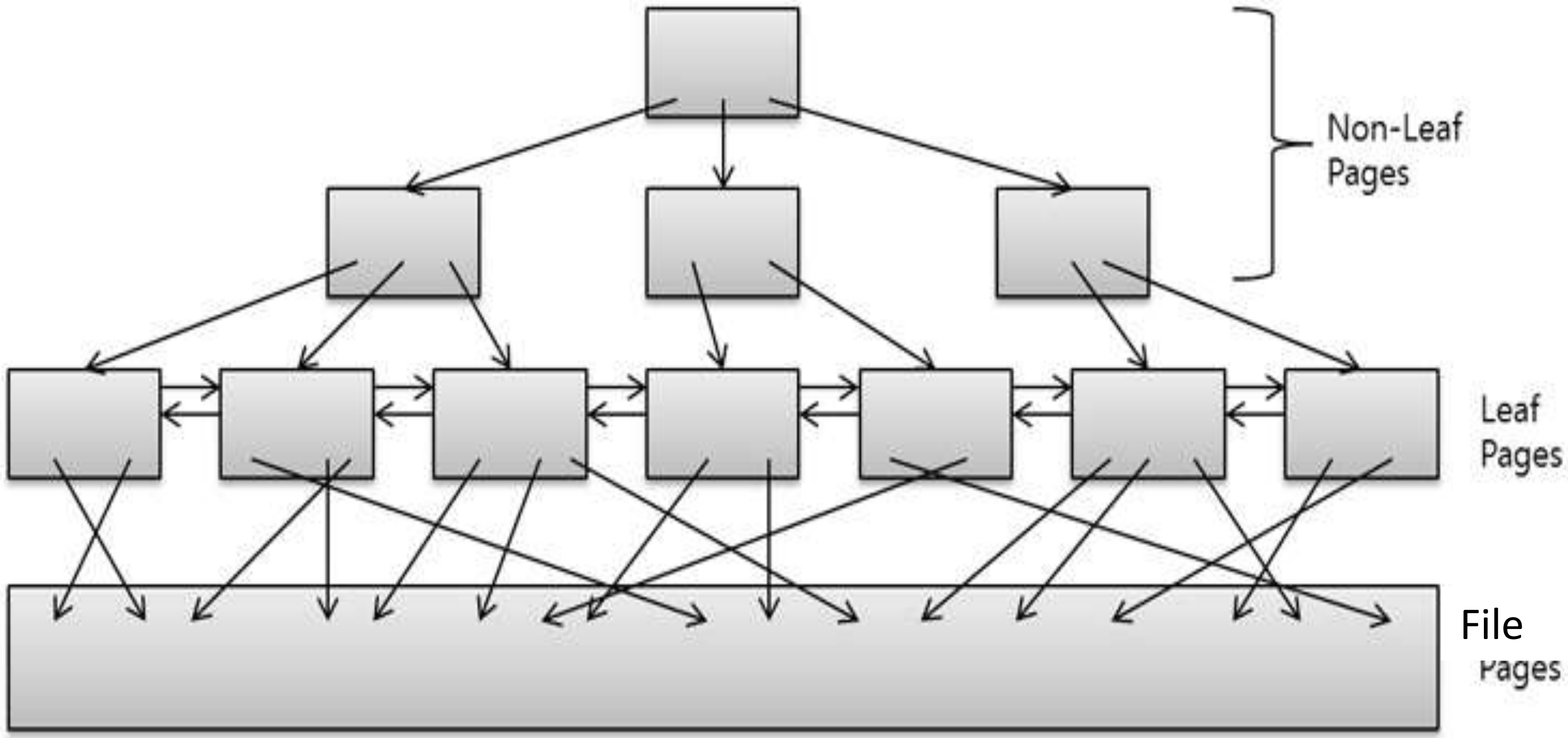
  #of pointers “ “ : 3 to 5

- Except for the root node

– Root node has at least one key and two pointers

B+ Tree Index Structure

Page→Node,block



B+Tree:Node Structures

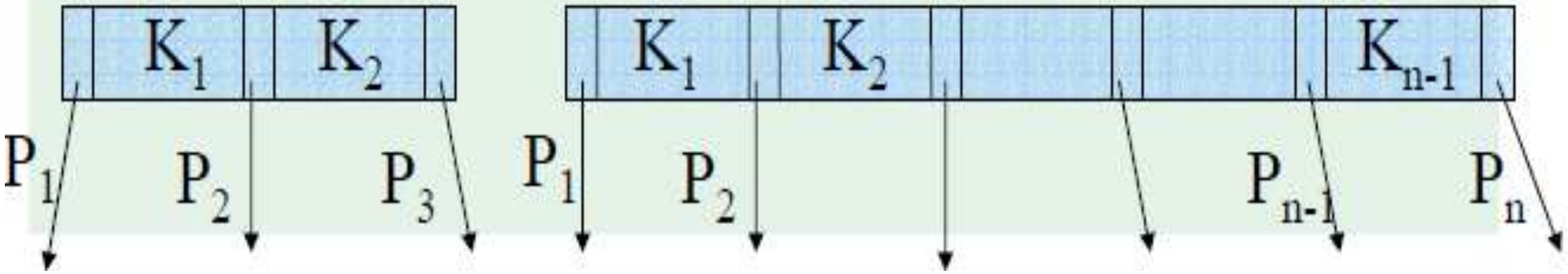
- Data pointers stored only in leaf nodes.
- Internal nodes contain only keys and tree ptrs.
- Note that in any node: K1<K2<........<Kn-1
- Leaf nodes form a linked(or Doubly linked)list.
- Leaf nodes contain keys and pointers to data file records.
- The #of levels is usually very small :

  →Short Tree, Fast searching.

B+ Tree :Internal Nodes

- B+ Tree is constructed by parameter **n**
  - Each Node (except root) has $\lceil n/2 \rceil$ to n pointers
  - Each Node (except root) has $\lceil n/2 \rceil$-1 to n-1 search-key values

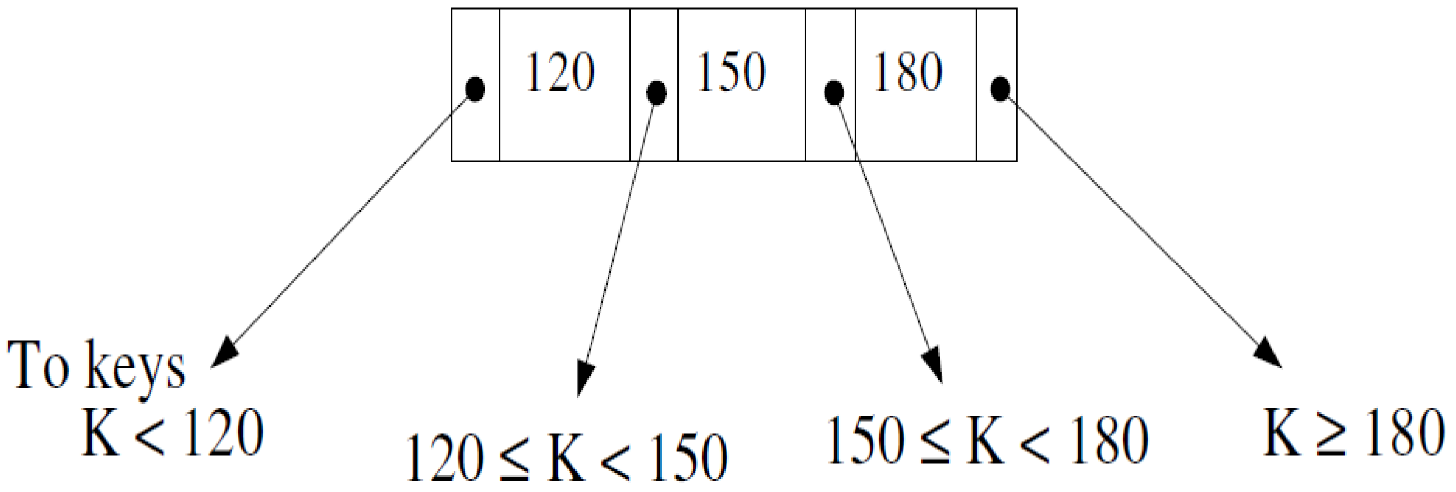
Case for n=3 General case for n



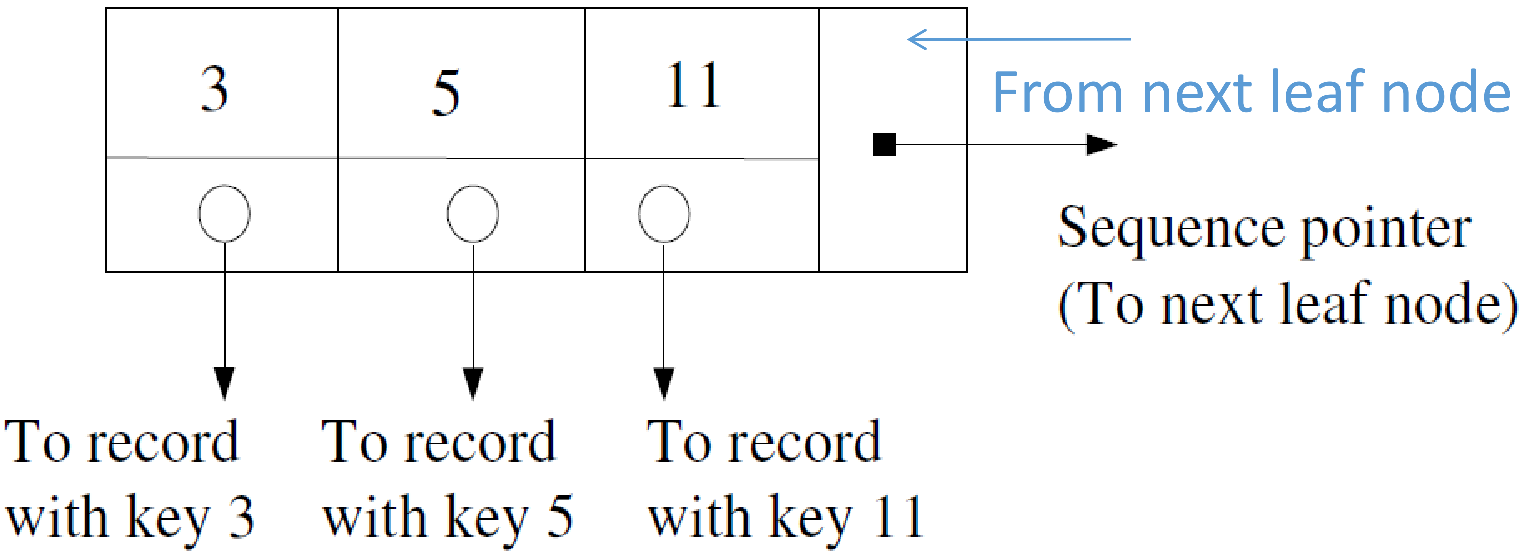
3 ptrs,2 keys

n ptrs ,n-1 keys.

B+ trees – internal node

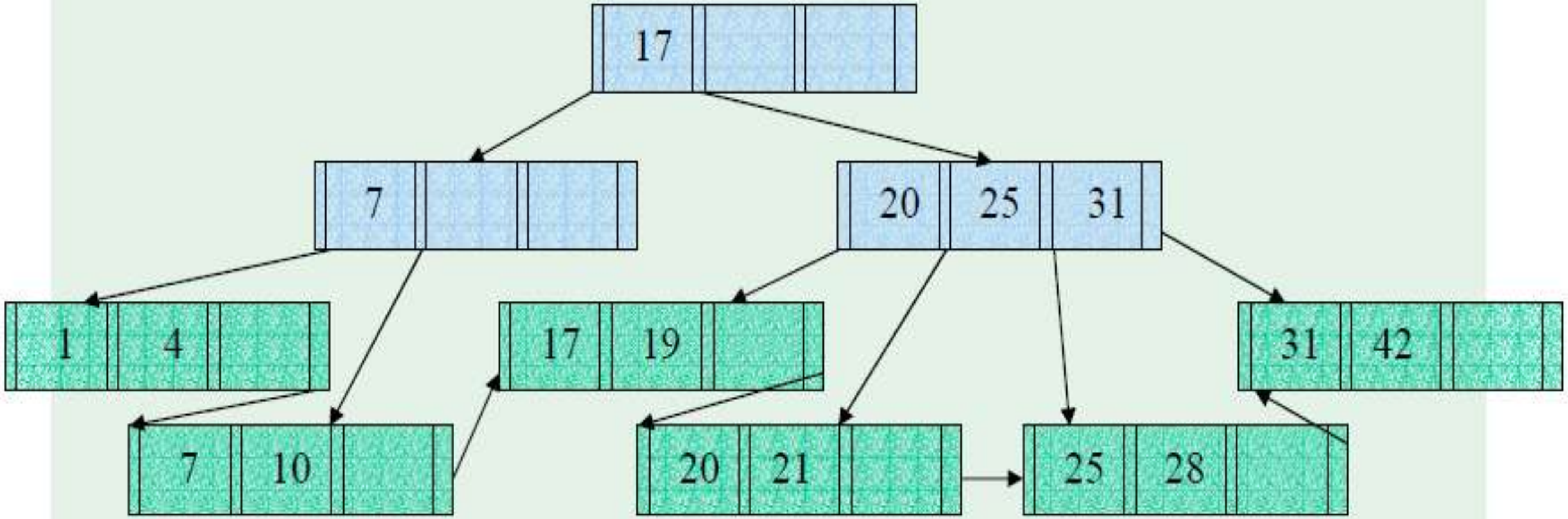


B+ tree – leaf node



Example: B+ Tree

- 1, 4, 7, 10, 17, 21, 31, 25, 19, 20, 28, 42



Using B+ trees: Index Search

- Suppose we have a data file and a B+ tree index to that file.
- How to access the data record whose key value is Kİ ?

  -Start from the root, find the position of Ki in the root

  -Follow the pointers until a leaf node.

  -The leaf node contains Kİ and ptr to the requested data record .

Searching a B-Tree

//Search is in a node , node size (fanout) = n[x] . A linear search is implemented
//We assume that, record pointers are stored in tree nodes.

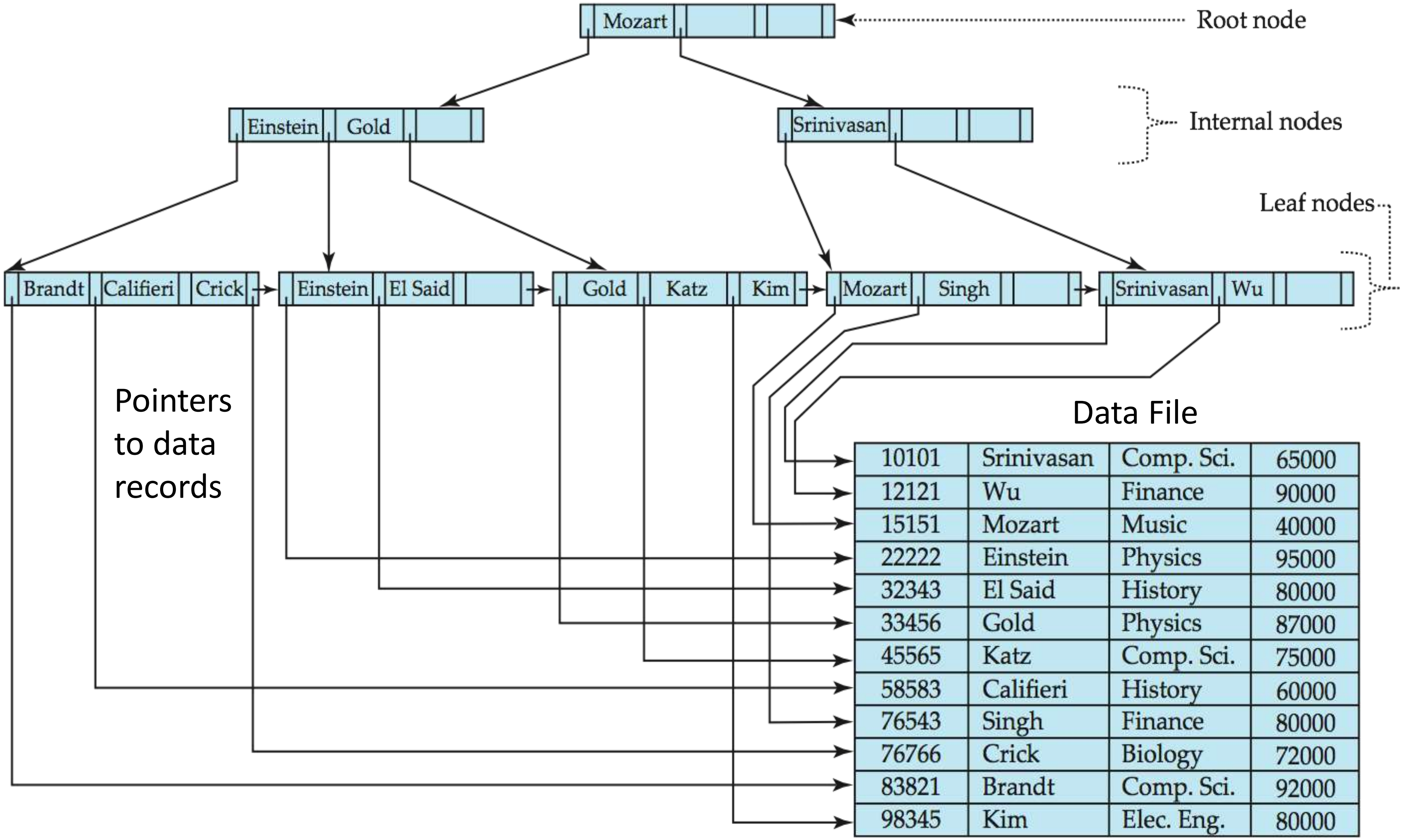
```
B-TREE-SEARCH(x, k)
i ← 1
while i ≤ n[x] and k > key_i[x]
     do i ← i + 1
if i ≤ n[x] and k = key_i[x]
   then return (x, i)
if leaf[x]
   then return NIL
   else DISK-READ(c_i[x])
        return B-TREE-SEARCH(c_i[x], k)
```

Start at the leftmost key in the node, and go to the right until you go too far.

If it is a leaf node, then you are done, as there is no leaf to inspect

Otherwise, retrieve the child node from the disk, and put it into memory

An Example of B+-Tree



10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Why B+ Tree Indexing ?

- B+ trees are fast,dynamic multilevel index structures.
- The root node is usually stored in main memory.

  →Locating a record using a B+ tree index requires only 3-4 disk acceses for even very large files!

How Efficient is the B+ indexing?

Consider an example:

Block size=4KB, Key: integer, ptr : 4B.

Node capacity= floor ((4096-4)/8) =511 [key+ptr=8B]

Find how many index keys can be stored at each level.

(For node pointers,we subtract 4 or 8 from the block capacity)

Why B+ Tree Indexing ?

- The Root : 511 keys, 512 ptrs.
- Level 1 : 512*511=261632 Keys

  512*512=262144 ptrs
- Level 2:(512*512) * 511=134.000.000 Keys !

If this is the leaf level,we are able to index 134 million records using just three levels!

How many disk accesses are needed? Three ! level1, level2, and the data block.

→Accessing any data record takes the same time.

Note : These are maximum possible values ,because we assume full node occupancy.

Time Complexity of B+Tree Operations

- Remember that all leaf nodes are at the same level(Balanced).
- Assume that the parameter is n which means all nodes must be at least half full (n/2 keys)
- Suppose we have N data records (keys) .
  1-Best case search→All Nodes are full.
  T_N= CEIL ($\log_{(n)}$ N) (Similar to $\log_2$ N)
- 2-Worst case search→All nodes are only half full.

  T_N= CEIL ($\log_{(n/2)}$ N) Example 1: let N=1.000.000 and n=100
  Best Case : $T_{1000000}$ =CEIL ($\log_{100}$ 1.000.000)
  = 3