CPE231RC — Algorithms — Lecture6

Space and Time Trade-Offs

Dr. Prapong Prechaprapranwong

Computer Engineering

King Mongkut's University of Technology Thonburi

Space and time trade-offs

algorithm trades increased space usage with decreased time

- **space** refers to the data storage in memory
- **time** refers to the time consumed in operation

Input enhancement approach

- counting method for sorting
  - Comparison-counting sort
  - Distribution-counting sort
- Input enhancement in string matching
  - Horspool’s algorithm
  - Boyer-Moore algorithm

\\

preprocessing , preconditioning

Store additional info by input preprocessing to accelerate solving the problem afterward

Prestructuring approach

Create access structure for faster/flexible data access.

- Hashing refer to CPE111
- B-tree

Sorting by Counting

for each element of a list to be sorted, the total number of elements smaller that element and recorded the results in a table. These numbers will indicate the positions of the elements in the sorted list → **comparison counting sort**

Array *A*[0..5]		62	31	84	96	19	47
Initially	*Count* []	0	0	0	0	0	0
After pass *i* = 0	*Count* []	3	0	1	1	0	0
After pass *i* = 1	*Count* []		1	2	2	0	1
After pass *i* = 2	*Count* []			4	3	0	1
After pass *i* = 3	*Count* []				5	0	1
After pass *i* = 4	*Count* []					0	2
Final state	*Count* []	3	1	4	5	0	2
Array *S*[0..5]		19	31	47	62	84	96

there are 3 numbers that smaller than 62

the numbers that larger than 62 plus 1

ALGORITHM *ComparisonCountingSort*$(A[0..n-1])$

//Sorts an array by comparison counting
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $S[0..n-1]$ of A's elements sorted in nondecreasing order
for $i \leftarrow 0$ **to** $n-1$ **do** $Count[i] \leftarrow 0$
for $i \leftarrow 0$ **to** $n-2$ **do**
  for $j \leftarrow i+1$ **to** $n-1$ **do**
    if $A[i] < A[j]$
      $Count[j] \leftarrow Count[j] + 1$
    else $Count[i] \leftarrow Count[i] + 1$
for $i \leftarrow 0$ **to** $n-1$ **do** $S[Count[i]] \leftarrow A[i]$
return S

$O(n^2)$

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i) = \frac{n(n-1)}{2}.$$

Practice I) use comparison-counting sort to sort this sequence of numbers (20 minutes)

A[0..7]	16	27	15	23	64	93	25	11

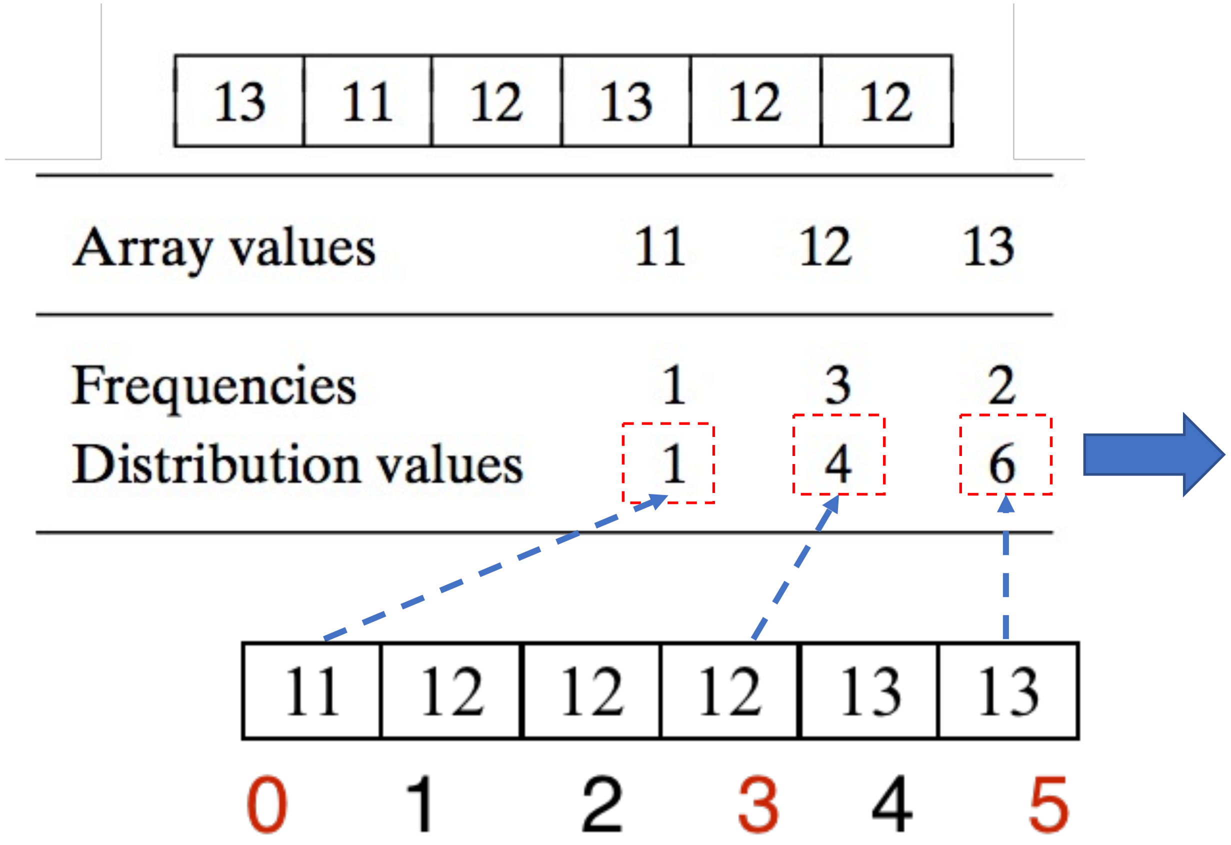
Count [ ]

initial	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
i = 0								
i = 1								
i = 2								
i = 3								
i = 4								
i = 5								
i = 6								
Final								

S[0..7]								

Sorting by Counting

Sorting a list which limited positive integer eq. 1-5. Sorting can be done by taking advantage of this feature with accumulate the sum of frequencies of these numbers → **distribution counting sort**



Tell ending indices of different value in sorted output.

- 11 at 0
- 12 at 3, 2, 1
- 13 at 5, 4

13	11	12	13	12	12

Array values	11	12	13
Frequencies	1	3	2
Distribution values	1	4	6

1.create the unique set of numbers

2. count the frequencies of all numbers

3. accumulate the frequencies

4. sort the numbers

	$D[0..2]$		
$A[5] = 12$	1	**4**	6
$A[4] = 12$	1	**3**	6
$A[3] = 13$	1	2	**6**
$A[2] = 12$	1	**2**	5
$A[1] = 11$	**1**	1	5
$A[0] = 13$	0	1	**5**

$S[0..5]$					
			12		
		12			
					13
	12				
11					
				13	

ALGORITHM *DistributionCountingSort*$(A[0..n-1], l, u)$

//Sorts an array of integers from a limited range by distribution counting
//Input: An array $A[0..n-1]$ of integers between l and u $(l \le u)$
//Output: Array $S[0..n-1]$ of A's elements sorted in nondecreasing order
for $j \leftarrow 0$ **to** $u - l$ **do** $D[j] \leftarrow 0$ //initialize frequencies
for $i \leftarrow 0$ **to** $n - 1$ **do** $D[A[i] - l] \leftarrow D[A[i] - l] + 1$ //compute frequencies
for $j \leftarrow 1$ **to** $u - l$ **do** $D[j] \leftarrow D[j - 1] + D[j]$ //reuse for distribution
for $i \leftarrow n - 1$ **downto** 0 **do**
    $j \leftarrow A[i] - l$
    $S[D[j] - 1] \leftarrow A[i]$
    $D[j] \leftarrow D[j] - 1$
return S

O(n+k)
n = no. of elements
k = no. of unique elements

Practice II) use distribution-counting sort to sort this sequence of numbers (20 minutes)

	1	2	3	4
Frequencies				
Distribution value				

D[0..3]

A[9] = 3				

A[0..9]

1	**4**	**2**	**1**	**3**	**2**	**1**	**3**	**2**	**3**

S[0..9]

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

Input Enhancement in String Matching

String Matching Problem

- Determine if a given pattern string (m characters) is in a text (n characters, $n \geq m$)
- Worst-case efficiency **O(*nm*)** by brute force

```
N O B O D Y _ N O T I C E D _ H I M
N O T
  N O T
    N O T
      N O T
        N O T
          N O T
            N O T
              N O T
```

Algorithm BruteForceStringMatching

Input: Pattern $P = p_1 p_2 p_3 \cdots p_m$ and Text $T = t_1 t_2 t_3 \cdots t_n$, $n \geq m$
Output: Index of the 1st character in the text that starts a matching substring, and -1 for the unsuccessful search

for $i := 1$ to $n - m + 1$ **do**
    $j \leftarrow 1$
    while $j \leq m$ and $p_j == t_{i+j}$ **do**
        $j \leftarrow j + 1$
    if $j == m + 1$ **return** i
return -1

Horspool's Algorithm

Searching for the pattern BARBER in some text from right to left:

$s_0 \quad \ldots \quad c \quad \ldots \quad s_{n-1}$

B A R B E R

In general, there are 4 possibilities can occur:

Case 1: There are no character *c* in the pattern. e.g., *c* is letter S

$s_0 \quad \ldots \quad$ S $\quad \ldots \quad s_{n-1}$

$\neq$

B A R B E R can shift the pattern by its entire length (*m*)

B A R B E R

Case 2: There are character *c* in the pattern but not the last one. e.g., *c* is letter B

s_0 ... B ... s_{n-1}

$\not\parallel$

B A R B E R

Shift the pattern to align the rightmost occurrence of *c* in the pattern *(m – (j+1))*

B A R B E R

Case 3: Character *c* matches the last position of the pattern and shows up only once in the pattern. e.g., *c* is letter R

s_0 ... M E R ... s_{n-1}

$\not\parallel$ $\parallel$ $\parallel$

L E A D E R

Shift the pattern by its entire length (*m*) <u>same as case 1</u>

L E A D E R

Case 4: Character c matches the last position and shows up many times in the pattern. e.g., c is letter R

s_0 . . . A R . . . s_{n-1}

$\nparallel$ $\parallel$

R E O R D E R

R E O R D E R

Shift the pattern to align the rightmost occurrence of *c* in the pattern *(m – (j+1))* <u>same as case 2</u>

It will be inefficient to check the character *c* with the pattern every time. The idea of input enhancement is performed by **precompute** shift size and **store** them in a table. e.g., If *c* = 'S', shift by 6 positions (Case 1), If *c* = 'B', shift by 2 positions (Case 2)

character *c*	A	B	C	D	E	F	. . .	R	. . .	Z	_
shift *t*(*c*)	4	2	6	6	1	6	6	3	6	6	6

$$t(c) = \begin{cases} \text{the pattern's length } m, \\ \text{if } c \text{ is not among the first } m-1 \text{ characters of the pattern;} \\ \\ \text{the distance from the rightmost } c \text{ among the first } m-1 \text{ characters} \\ \text{of the pattern to its last character, otherwise.} \end{cases} \quad \textbf{(7.1)}$$

ALGORITHM *ShiftTable*$(P[0..m-1])$

//Fills the shift table used by Horspool's and Boyer-Moore algorithms
//Input: Pattern $P[0..m-1]$ and an alphabet of possible characters
//Output: $Table[0..size-1]$ indexed by the alphabet's characters and
// filled with shift sizes computed by formula (7.1)
for $i \leftarrow 0$ **to** $size - 1$ **do** $Table[i] \leftarrow m$
for $j \leftarrow 0$ **to** $m - 2$ **do** $Table[P[j]] \leftarrow m - 1 - j$
return *Table*

Example: Search the pattern “BARBER” with Horspool’s algorithm

BARBER, m = 6

j

Table['B'] = 6-1-$\boxed{0}$ =5
Table['A'] = 6-1-$\boxed{1}$ = 4
Table['R'] = 6-1-$\boxed{2}$ = 3
Table['B'] = 6-1-$\boxed{3}$ = 2 (update)
Table['E'] = 6-1-$\boxed{4}$ = 1
Table['R'] = 6-1-$\boxed{5}$ = 0 (don’t use)

Otherwise = 6

character c	A	B	C	D	E	F	. . .	R	. . .	Z	_
shift $t(c)$	4	2	6	6	1	6	6	3	6	6	6

```
J I M _ S A W _ M E _ I N _ A _ B A R B E R S H O P
B A R B E R           B A R B E R ≠
B A R B E R               B A R B E R
          B A R B E R           B A R B E R
```

```
ALGORITHM  HorspoolMatching(P[0..m − 1], T[0..n − 1])
    //Implements Horspool's algorithm for string matching
    //Input: Pattern P[0..m − 1] and text T[0..n − 1]
    //Output: The index of the left end of the first matching substring
    //        or −1 if there are no matches
    ShiftTable(P[0..m − 1])          //generate Table of shifts
    i ← m − 1                        //position of the pattern's right end
    while i ≤ n − 1 do
        k ← 0                        //number of matched characters
        while k ≤ m − 1 and P[m − 1 − k] = T[i − k] do
            k ← k + 1
        if k = m
            return i − m + 1
        else i ← i + Table[T[i]]
    return −1
```

Boyer-Moore's Algorithm

like Horspool's: right to left
additional: **bad-symbol shift** and **good-suffix shift**
k = No. of matched characters

Bad-Symbol shift: guide by the character *c* caused a mismatch with the pattern. if *c* is not in the pattern, shift the pattern to just pass this *c* in the text

the bad symbol shift-size d_1 = max $\{t_1(c) - k, 1\}$

- $t_1(c)$ is the same shift-size as Horspool's
- k is the no. of matched characters

text ➔	s_0		…	c	s_{i-k+1}	…	s_i	… s_{n-1}
				$\neq$	$\|\|$		$\|\|$	
pattern ➔		p_0	…	p_{m-k-1}	p_{m-k}	…	p_{m-1}	
		p_0	…	p_{m-k-1}	p_{m-k}	…	p_{m-1}	

Example: Search the pattern BARBER in text using Bad-symbol shift

match the last two characters before failing on letter S (k = 2)

s_0 ... S E R ... s_{n-1}

≠ || ||

B A R B E R

Boyer-Moore's → B A R B E R shift by $t_1(S) - k = 6 - 2 = 4$

Horspool's → B A R B E R shift by $t_1(R) = 3$

character c	A	B	C	D	E	F	. . .	R	. . .	Z	_
shift $t(c)$	4	2	6	6	1	6	6	3	6	6	6

Example II: Search the pattern BARBER in text using Bad-symbol shift

match the last two characters before failing on letter A (k = 2)

s_0 … A E R … s_{n-1}

$\neq$ || ||

B A R B E R

Boyer-Moore's → B A R B E R shift by $t_1(A) - k = 4 - 2 = 2$

Horspool's → B A R B E R shift by $t_1(R) = 3$

character c	A	B	C	D	E	F	. . .	R	. . .	Z	_
shift $t(c)$	4	2	6	6	1	6	6	3	6	6	6

The good-suffix shift: guide by a successful match of the last $k > 0$ characters of the pattern

the good-suffix shift-size d_2 varied by k (no. of matched characters)

for example:

k	pattern	d_2
1	A B C B A B	2
2	A B C B A B	4
3	A B C B A B	4
4	A B C B A B	4
5	A B C B A B	4

note that for k = 3,4,5, d_2 is not 6 because there is the rightmost pattern "AB"

Example: Search the pattern ABCBAB in text using Good-Suffix shift

match the last three characters before failing on letter c (k = 3)

s_0 … c B A B C B A B … s_{n-1}

≠ || || ||

A B C B A B

A B C B A B

k = 3 shift by 4

shift-size of Boyer –Moore's algorithm

$$d = \begin{cases} d_1 & if\ k = 0 \\ \max(d_1, d_2\,) & if\ k > 0 \end{cases} \qquad \text{where } d_1 = \max\{t_1(c) - k, 1\}$$

Example: Boyer-Moore's algorithm, search the pattern BAOBAB in text

The bad-symbol table

c	A	B	C	D	…	O	…	Z	_
$t_1(c\,)$	1	2	6	6	6	3	6	6	6

The good-suffix table

k	pattern	d_2
1	B A O B A B	2
2	B A O B A B	5
3	B A O B A B	5
4	B A O B A B	5
5	B A O B A B	5

B E S S _ K N E W _ A B O U T _ B A O B A B S

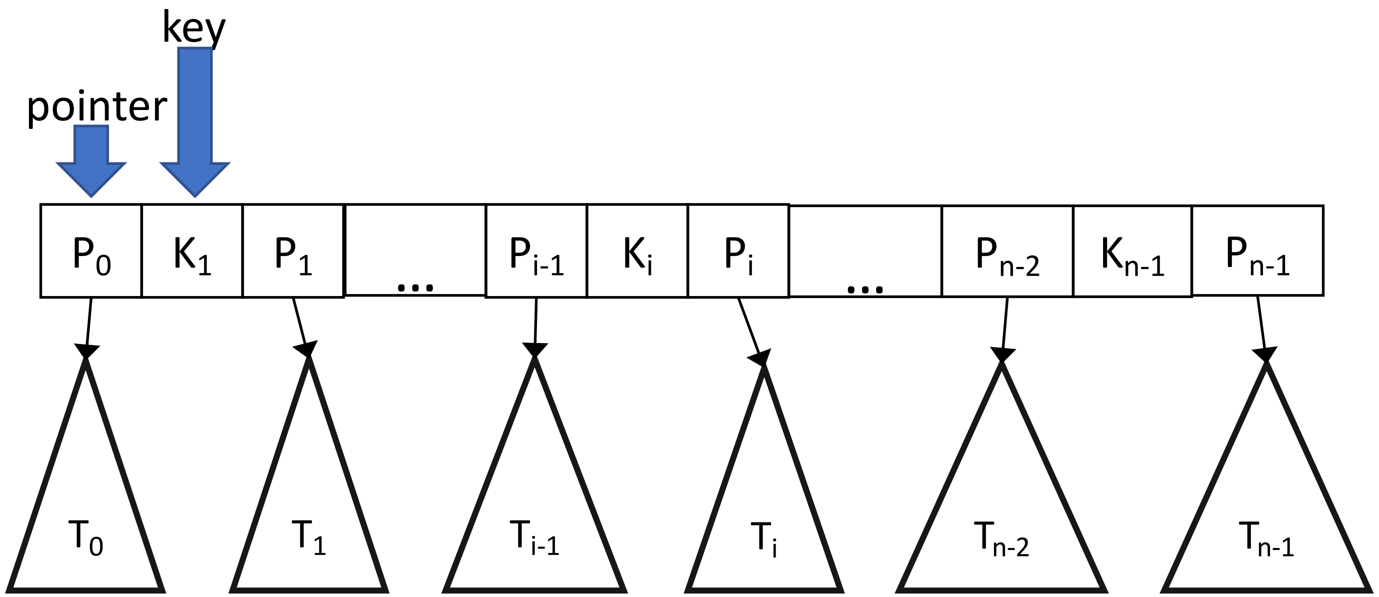
$d_1 = t_1(K) - 0 = 6$

$d_1 = t_1(\_) - 2 = 4$
$d_2 = 5$
$d = \max\{4,5\} = 5$

$d_1 = t_1(\_) - 1 = 5$
$d_2 = 2$
$d = \max\{5,2\} = 5$

B-Trees and B$^+$-Tree

B-Trees extend the idea of the 2-3 trees by permitting more than one key in the same node of a search tree and all data records (or record keys) are stored at the leaves,



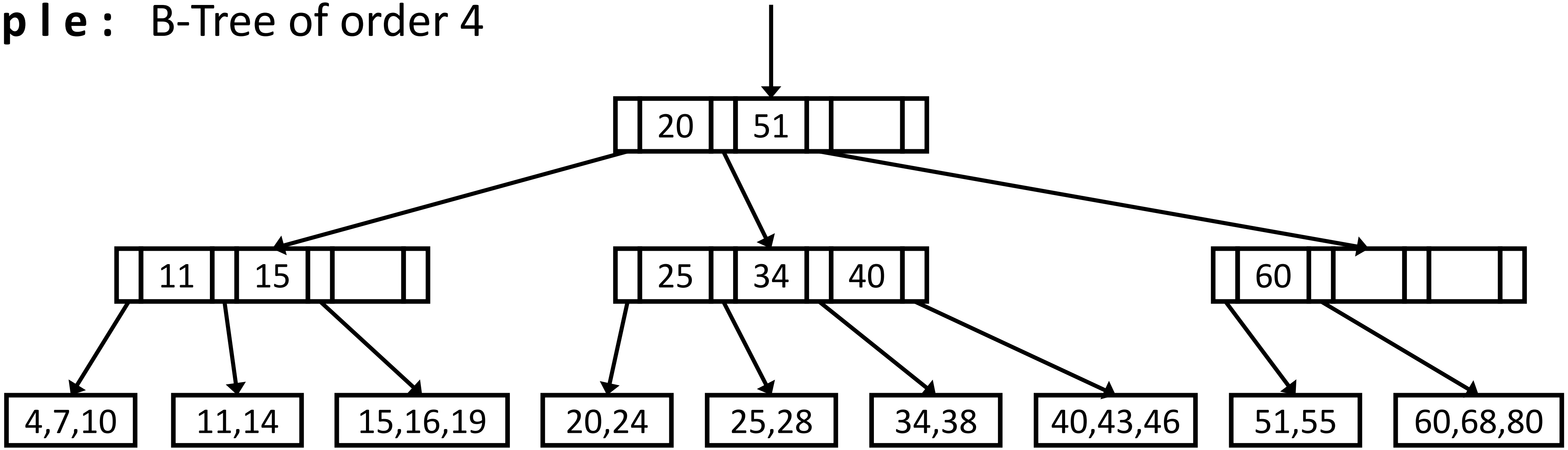
each parent node contains *n-1* ordered keys(K_1 < … < K_{n-1}) interposed with *n* pointers (P_0 … P_{n-1}) to the node’s children (T_0 .. T_{n-1})

- keys in subtree $T_0 < K_1$
- all the keys in subtree $T_1 \geq K_1$ and $< K_2$ with K_1 = the smallest key in T_1
- the last subtree $T_{n-1} \geq K_{n-1}$ with K_{n-1} = the smallest key in T_{n-1}

B-Tree of order $\underline{m} \geq 2$

- the root is either a leaf or has between 2 and <u>m</u> children
- Each node, except for the root and the leaves, has between $\lceil m/2 \rceil$ and m children (and hence between $\lceil m/2 \rceil - 1$ and <u>*m*-1</u> keys)
- the tree is perfectly balanced. i.e., all its leaves are at the same level

Example: B-Tree of order 4



- order of 4 means each node has between 2 and 4 children
- the height *h* of the B-Tree of order *m* with *n* nodes → $h \leq \left\lfloor log_{\lceil \frac{m}{2} \rceil} \frac{n+1}{4} \right\rfloor + 1$
- searching in a B-Tree is a O(log *n*)

order *m*	50	100	250
h's upper bound	6	5	4