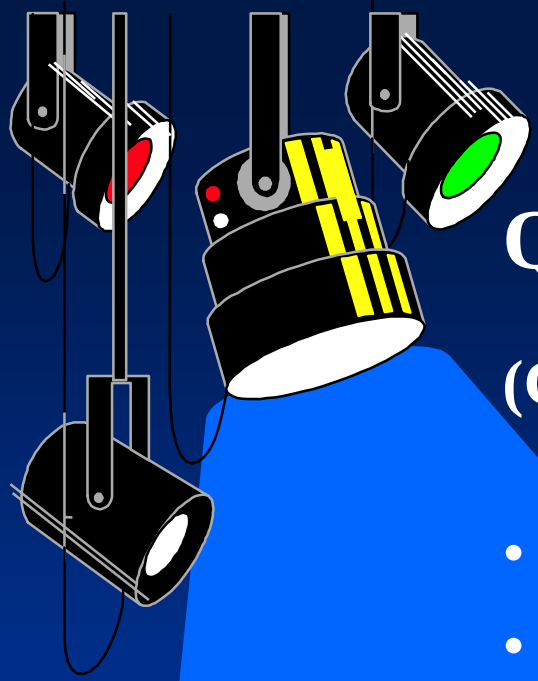
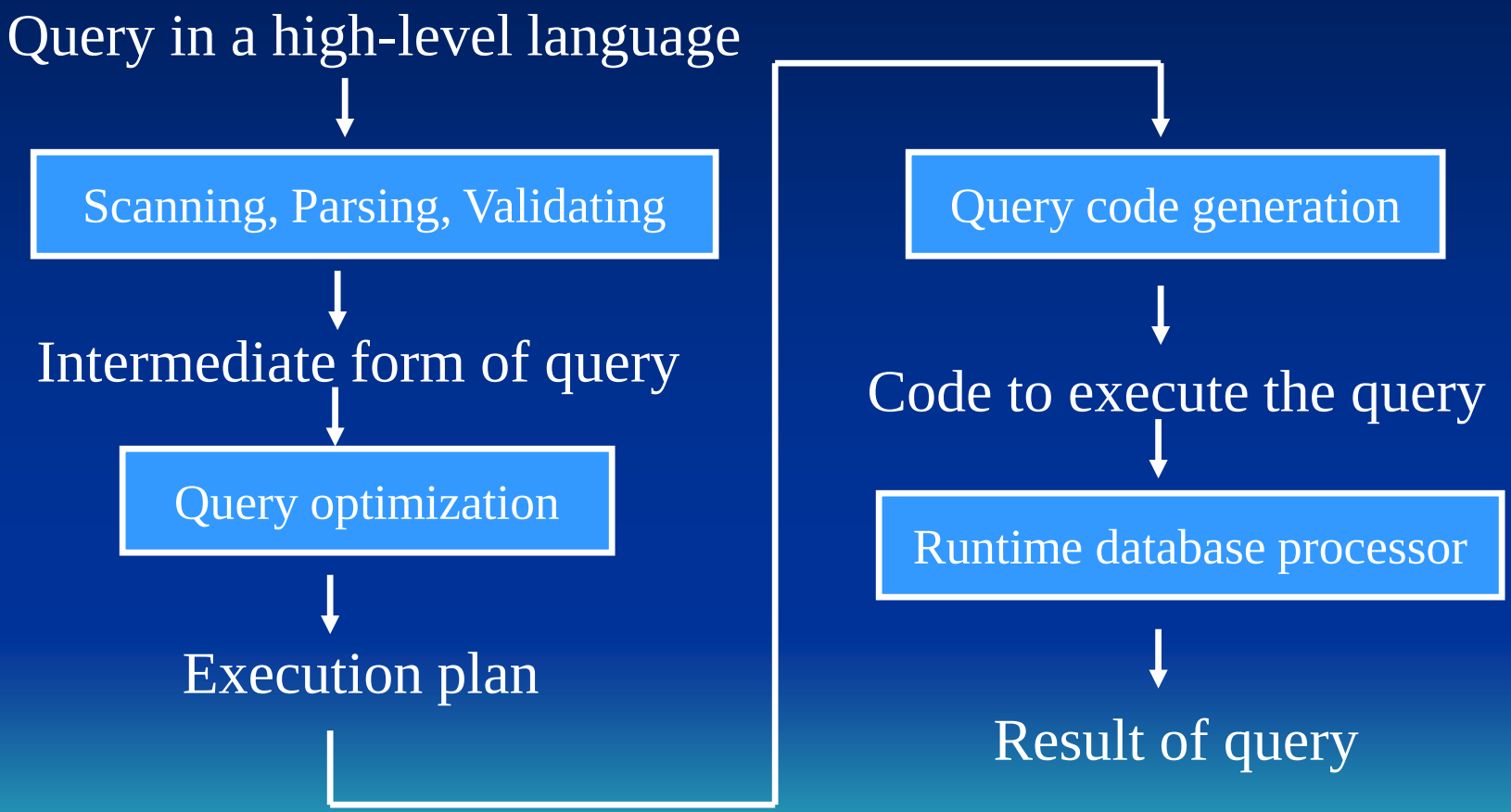


Query Processing and Optimization

(Ch. 18, 3rd – Ch. 15, 4th ed. 5th ed., Ch. 19, 6th ed., Ch. 18, 19, 7th ed.)

- Processing a high-level query
- Translating SQL queries into relational algebra
- Basic algorithms
  - Sorting: internal sorting and external sorting
  - Implementing the SELECT operation
  - Implementing the JOIN operation
  - Implementing the Project operation
  - Other operations
- Heuristics for query optimization

- **Steps of processing a high-level query**



- **Translating SQL queries into relational algebra**
  - decompose an SQL query into query blocks
    query block - SELECT-FROM-WHERE clause

Example:
```
SELECT  LNAME, FNAME
FROM    EMPLOYEE
WHERE   SALARY > (SELECT    MAX(SALARY)
                  FROM      EMPLOEE
                  WHERE     DNO = 5);
```

```
SELECT   MAX(SALARY)
FROM     EMPLOYEE
WHERE    DNO = 5
```

inner block

```
SELECT   LNAME, FNAME
FROM     EMPLOYEE
WHERE    SALARY > c
```

outer block

- **Translating SQL queries into relational algebra**
  - translate query blocks into relational algebra expressions

```
SELECT   MAX(SALARY)
FROM     EMPLOYEE
WHERE    DNO = 5
```

⇒ $\mathscr{F}_{\text{MAX SALARY}}(\sigma_{\text{DNO=5}}(\text{EMPLOYEE}))$

```
SELECT   LNAME, FNAME
FROM     EMPLOYEE
WHERE    SALARY > c
```

⇒ $\pi_{\text{LNAME FNAME}}(\sigma_{\text{SALARY>C}}(\text{EMPLOYEE}))$

- **Basic algorithms**
  - sorting: internal sorting and external sorting
  - algorithm for SELECT operation
  - algorithm for JOIN operation
  - algorithm for PROJECT operation
  - algorithm for SET operations
  - implementing AGGREGATE operation
  - implementing OUTER JOIN

- **Basic algorithms**
  - internal sorting - sorting in main memory:

    sort a series of integers,

    sort a series of keys

    sort a series of records
  - different sorting methods:

    simple sorting

    bubble sorting

    merge sorting

    quick sorting

    heap sorting

- **Basic algorithms**
  - different internal sorting methods:

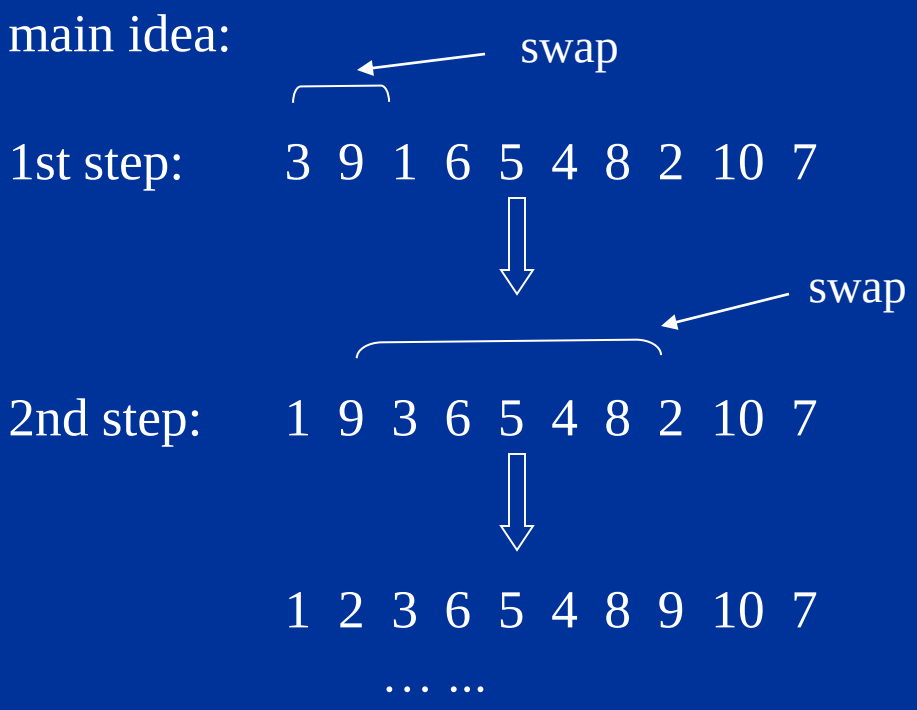
    sorting numbers

Input *n* numbers. Sort them such that the numbers are ordered increasingly.

3 9 1 6 5 4 8 2 10 7

1 2 3 4 5 9 7 8 9 10

- **Basic algorithms**
  - A simple sorting algorithm



- **Basic algorithms**
  - A simple sorting algorithm

    Algorithm

    Input: an array A containing n integers.
    Output: sorted array.
    1. $i := 2$;
    2. Find the least element c from $A(i)$ to $A(n)$;
    3. If c is less than $A(i - 1)$, exchange $A(i - 1)$ and c;
    4. $i := i + 1$; goto step (2).

    Time complexity: $O(n^2)$

    $(n – 1) + (n - 2) + \ldots + 2 + 1 = n(n - 1)/2$

Heapsort

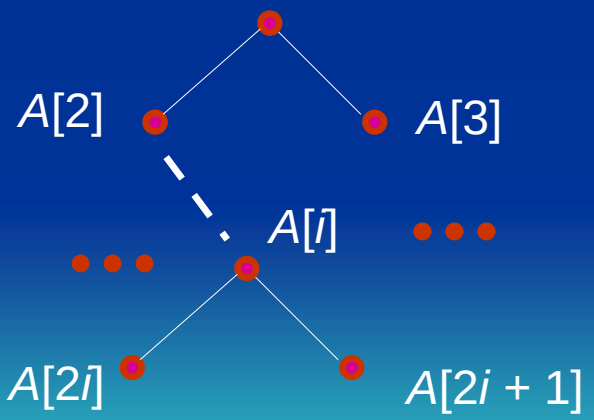
- What is a heap?
- MaxHeap and Maintenance of MaxHeaps
  - *MaxHeapify*
  - *BuildMaxHeap*
- Heapsort
  - Algorithm
  - Heapsort analysis

Heapsort

- Combines the better attributes of merge sort and insertion sort.
  - Like merge sort, but unlike insertion sort, running time is $O(n \lg n)$.
  - Like insertion sort, but unlike merge sort, sorts in place.
- Introduces an algorithm design technique
  - Create data structure (*heap*) to manage information during the execution of an algorithm.
- The *heap* has other applications beside sorting.
  - Priority Queues

Data Structure Binary Heap

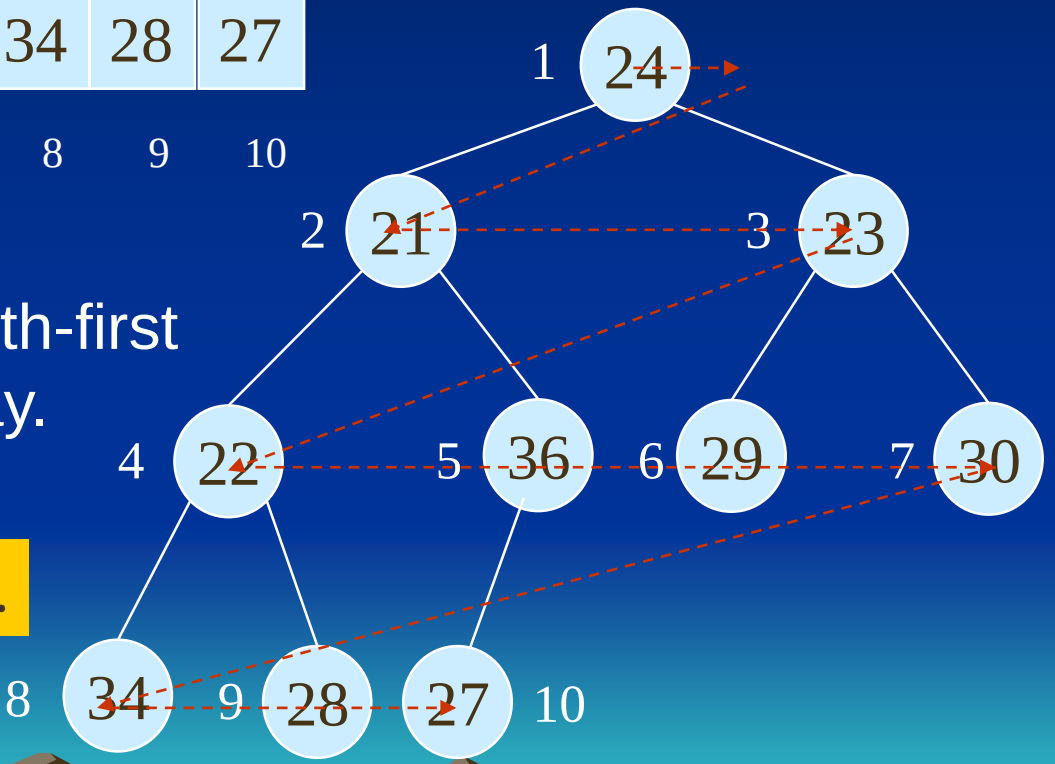
- Array viewed as a nearly complete binary tree.
  - Physically – linear array.
  - Logically – binary tree, filled on all levels (except lowest.)
- Map from array elements to tree nodes and vice versa
  - Root – $A[1]$, Left[Root] – $A[2]$, Right[Root] – $A[3]$
  - Left[i] – $A[2i]$
  - Right[i] – $A[2i+1]$
  - Parent[i] – $A[\lfloor i/2 \rfloor]$



Data Structure Binary Heap

- length[*A*] – number of elements in array *A.*
- heap-size[*A*] – number of elements in heap stored in *A.*
  - heap-size[*A*] ≤ length[*A*]

24	21	23	22	36	29	30	34	28	27
1	2	3	4	5	6	7	8	9	10

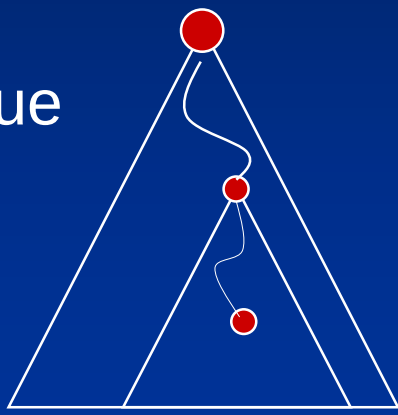


Searching the tree in breadth-first fashion, we will get the array.

Last row filled from left to right.

Heap Property (Max and Min)

- Max-Heap
  - For every node excluding the root, the value stored in that node is at most that of its parent: $A[parent[i]] \geq A[i]$
- Largest element is stored at the root.
- In any subtree, no values are larger than the value stored at the subtree's root.



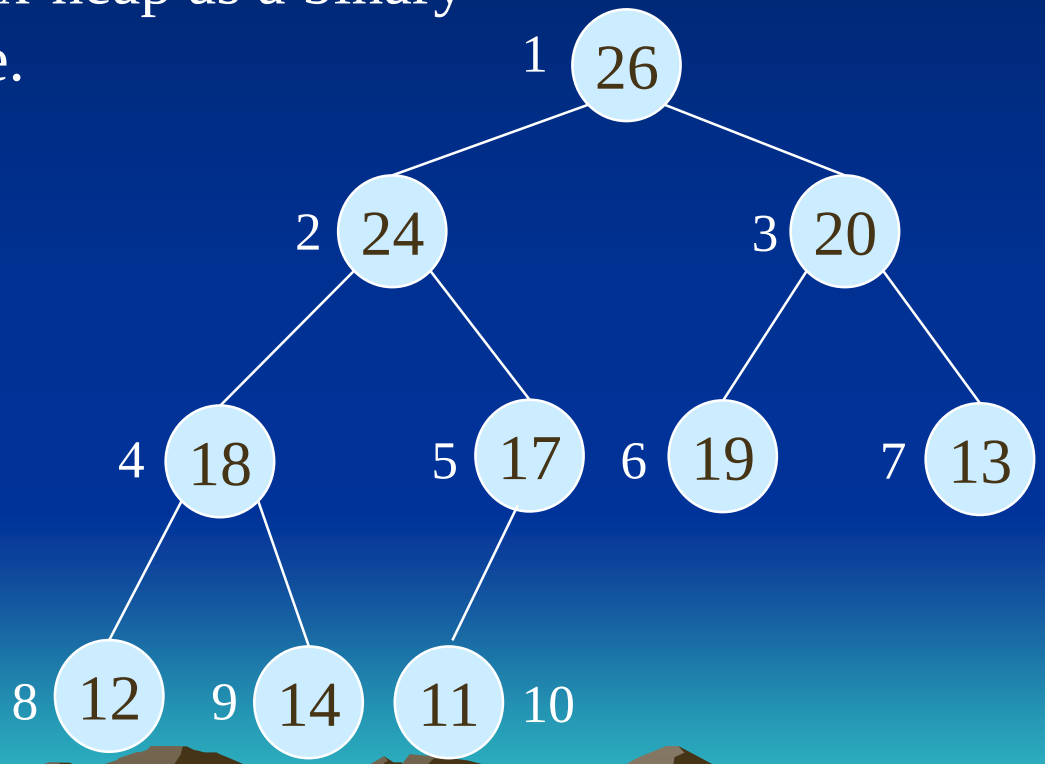
- Min-Heap
  - For every node excluding the root, the value stored in that node is at least that of its parent: $A[parent[i]] \leq A[i]$
- Smallest element is stored at the root.
- In any subtree, no values are smaller than the value stored at the subtree's root

Heaps – Example

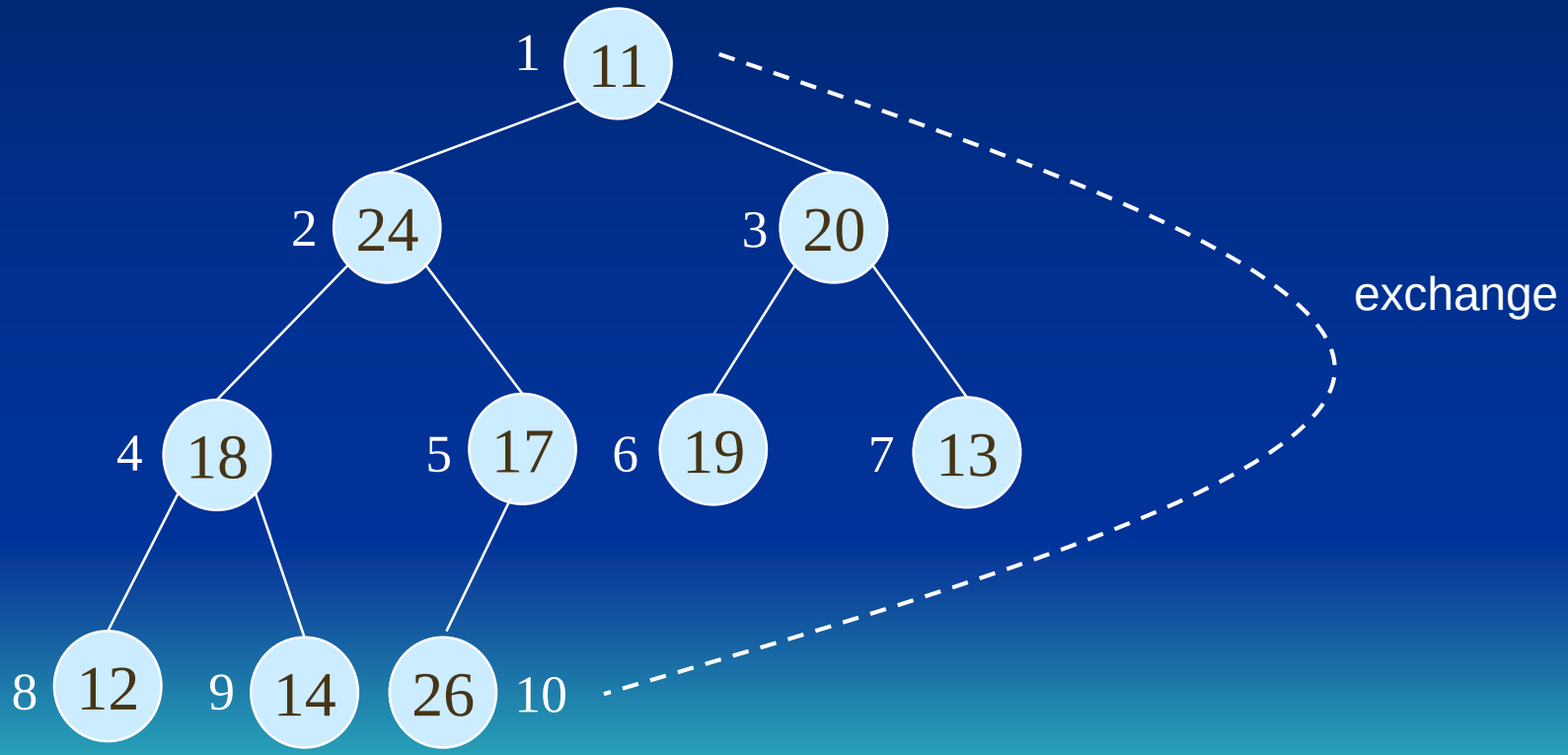
26	24	20	18	17	19	13	12	14	11
1	2	3	4	5	6	7	8	9	10

Max-heap as an array.

Max-heap as a binary tree.



11	24	20	18	17	19	13	12	14	26
1	2	3	4	5	6	7	8	9	10



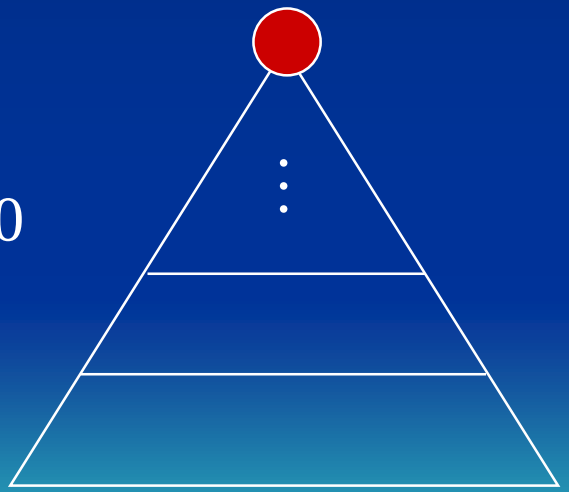
Heaps in Sorting

- Use max-heaps for sorting.
- The array representation of a max-heap is not sorted.
- Steps in sorting
  - (i) Convert the given array of size *n* to a max-heap (*BuildMaxHeap*)
  - (ii) Swap the first and last elements of the array.
    - Now, the largest element is in the last position – where it belongs.
    - That leaves *n* – 1 elements to be placed in their appropriate locations in the sequence.
    - However, the array of first *n* – 1 elements is no longer a max-heap.
    - Float the element at the root down one of its subtrees so that the array remains a max-heap (*MaxHeapify*)
    - Repeat step (ii) until the array is sorted.

Heap Characteristics

- *Height* $= \lfloor \lg n \rfloor$
- No. of *leaves* $= \lceil n/2 \rceil$
- No. of nodes of height $h \leq \lceil n/2h+1 \rceil$

height(*a leaf*) = 0



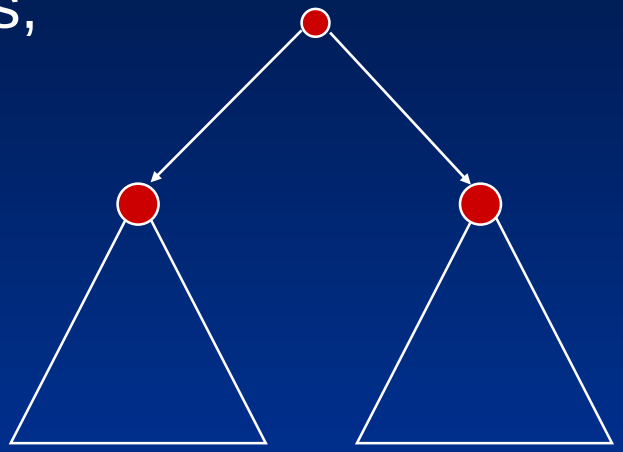
$\leq \lceil n/2^{2+1} \rceil$

$\leq \lceil n/2^{1+1} \rceil$

$\leq \lceil n/2^{0+1} \rceil$

MaxHeapify

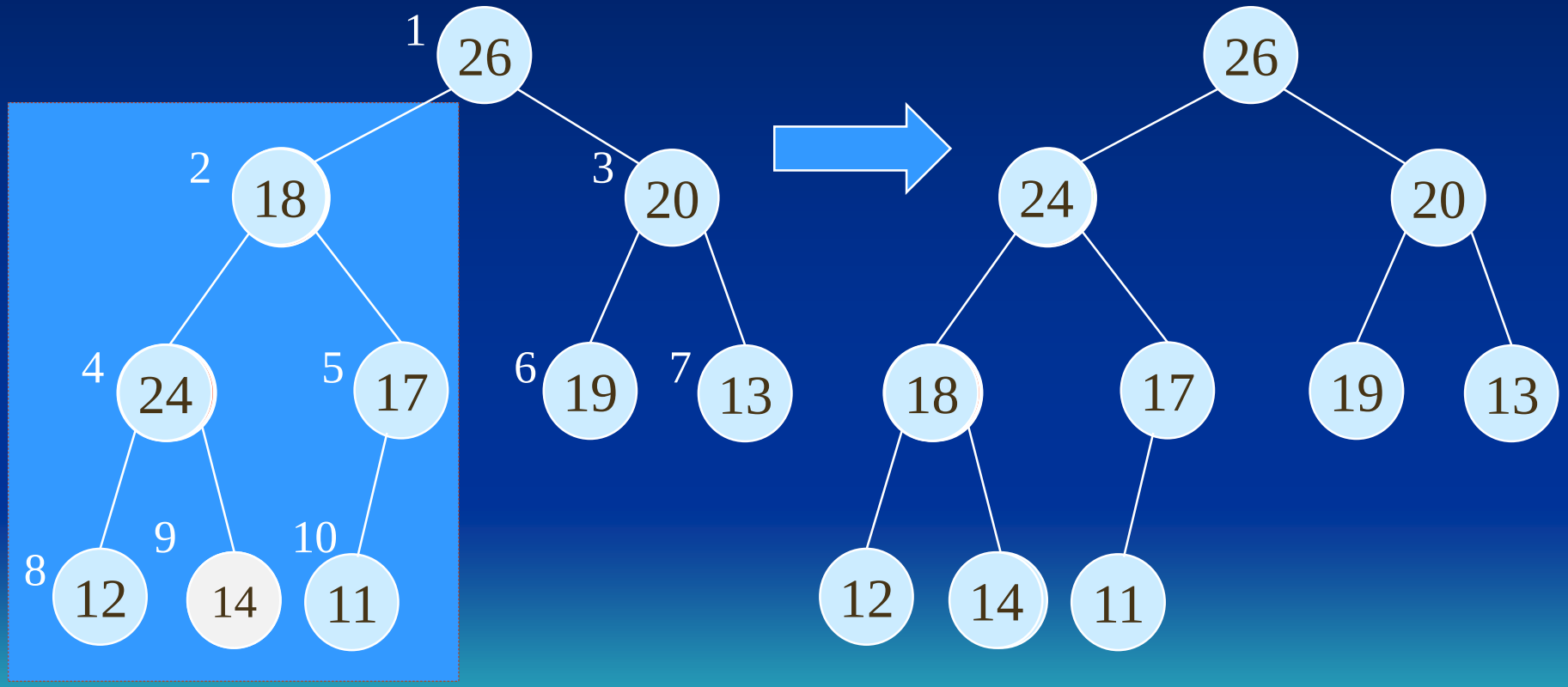
- Suppose two subtrees are max-heaps, but the root violates the max-heap property.

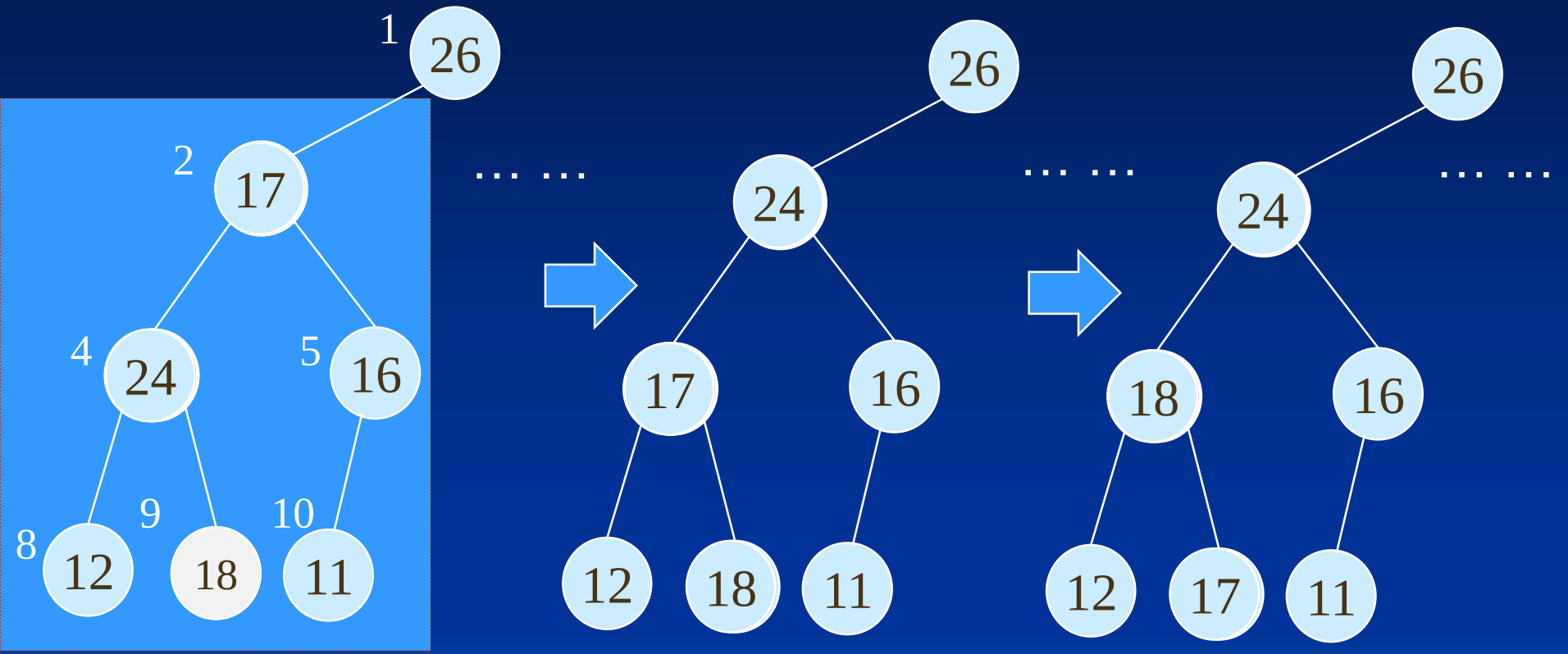


- Fix the offending node by exchanging the value at the node with the larger of the values at its children.
  - May lead to the subtree at the child not being a max heap.
- Recursively fix the children until all of them satisfy the max-heap property.

MaxHeapify – Example

MaxHeapify(*A*, 2)



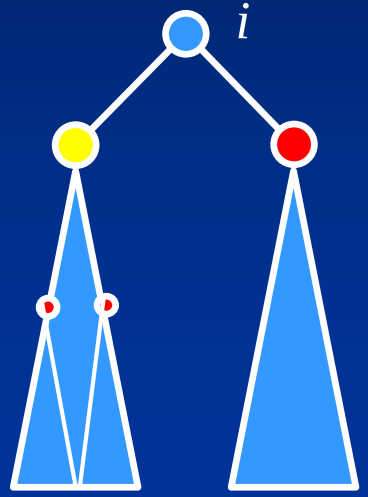

1
26
2
17
4
24
5
16
8
12
9
18
10
11
26
24
17
16
12
18
11
26
24
18
16
12
17
11

Procedure MaxHeapify

MaxHeapify(*A*, *i*)

1. $l \leftarrow$ left(i) (* $A[l]$ is the left child of $A[i]$.*)
2. $r \leftarrow$ right(i)
3. **if** $l \leq$ *heap-size*$[A]$ and $A[l] > A[i]$
4. **then** *largest* $\leftarrow l$
5. **else** *largest* $\leftarrow i$
6. **if** $r \leq$ *heap-size*$[A]$ **and** $A[r] > A[$*largest*$]$
7. **then** *largest* $\leftarrow r$
8. **if** *largest* $\neq i$
9. **then** exchange $A[i] \leftrightarrow A[$*largest*$]$
10. *MaxHeapify*(*A*, *largest*)

Assumption:
Left(i) and Right(i) are max-heaps.



A[*largest*] must be the largest among *A*[*i*], *A*[*l*] and *A*[*r*].

Running Time for MaxHeapify

- *MaxHeapify*(*A*, *i*)
- 1. $l \leftarrow$ left(*i*)
- 2. $r \leftarrow$ right(*i*)
- 3. **if** $l \leq$ *heap-size*[*A*] and *A*[*l*] > *A*[*i*]
- 4. **then** *largest* $\leftarrow l$
- 5. **else** *largest* $\leftarrow i$
- 6. **if** $r \leq$ *heap-size*[*A*] **and** *A*[*r*] > *A*[*largest*]
- 7. **then** *largest* $\leftarrow r$
- 8. **if** *largest* $\neq i$
- 9. **then** exchange *A*[*i*] $\leftrightarrow$ *A*[*largest*]
- 10. *MaxHeapify*(*A*, *largest*)

Time to fix node *i* and its children = $\Theta(1)$

PLUS

Time to fix the subtree rooted at one of *i*'s children = *T*(size of subree at *largest*)

Running Time for MaxHeapify(A, n)

- $T(n)$ = T(size of subree at *largest*) + $\Theta(1)$
- size of subree at *largest* $\leq 2n/3$ (worst case occurs when the last row of tree is exactly half full)
- $T(n) \leq T(2n/3) + \Theta(1) \Rightarrow T(n) = O(\lg n)$
- Alternately, MaxHeapify takes $O(h)$ where h is the height of the node where MaxHeapify is applied

Building a Max-heap

- Use *MaxHeapify* to convert an array *A* into a max-heap.
- How?
- Call MaxHeapify on each element in a bottom-up manner.

BuildMaxHeap(*A*)

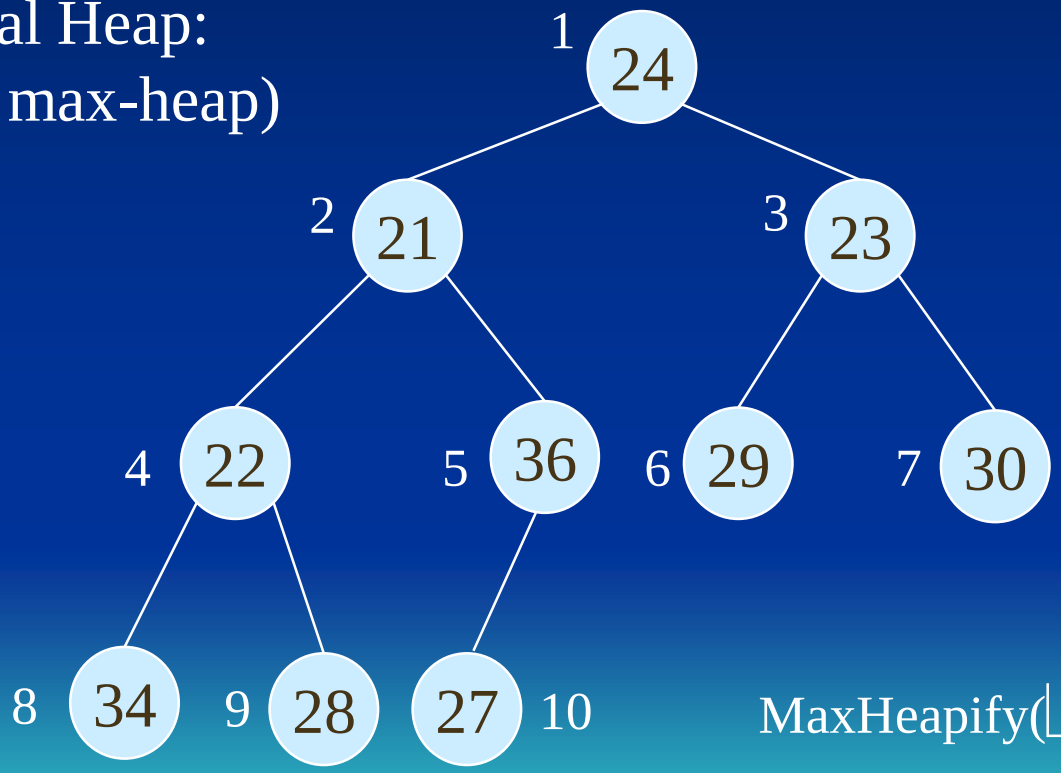
1. *heap-size*[*A*] ← *length*[*A*]
2. **for** *i* ← ⌊*length*[*A*]/2⌋ **downto** 1 (**A*[*length*[*A*]/2⌋ + 1],
3. **do** *MaxHeapify*(*A*, *i*) *A*[*length*[*A*]/2⌋ + 2],
   … are leaf nodes.*)

BuildMaxHeap – Example

Input Array:

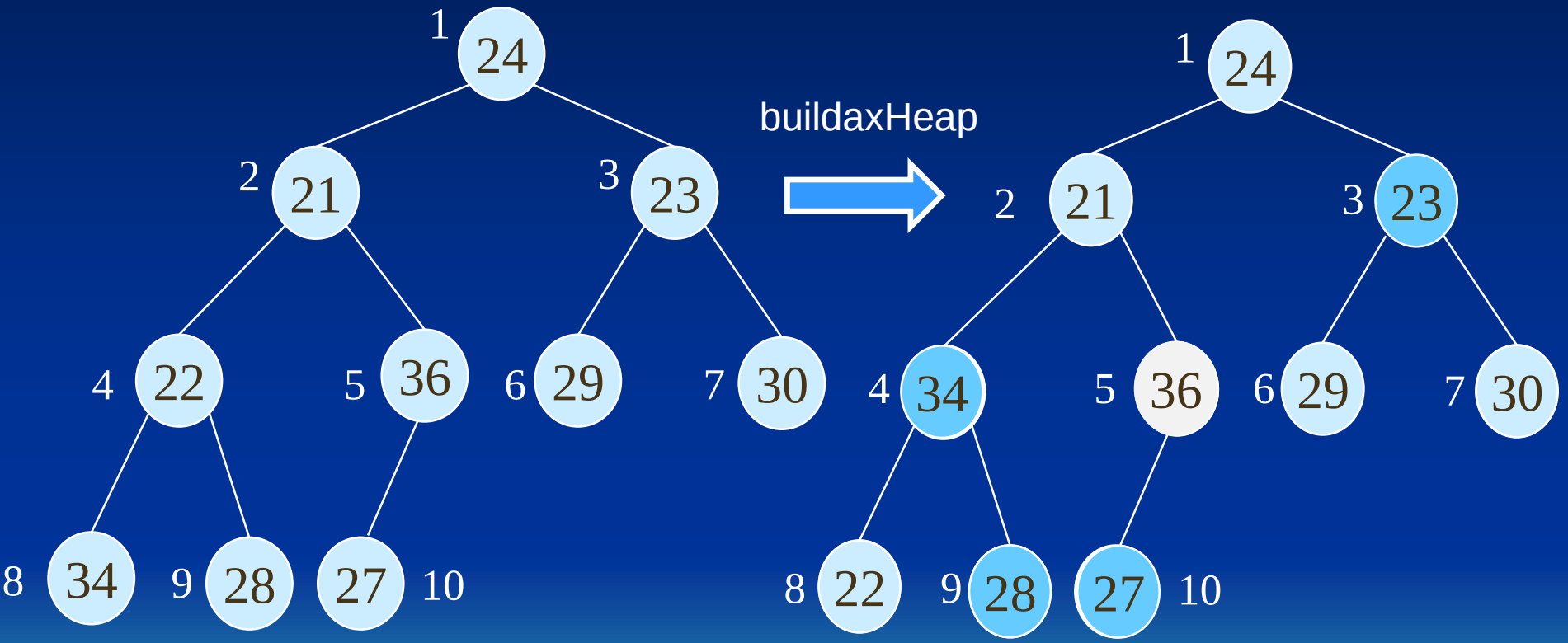
24	21	23	22	36	29	30	34	28	27

Initial Heap:
(not max-heap)



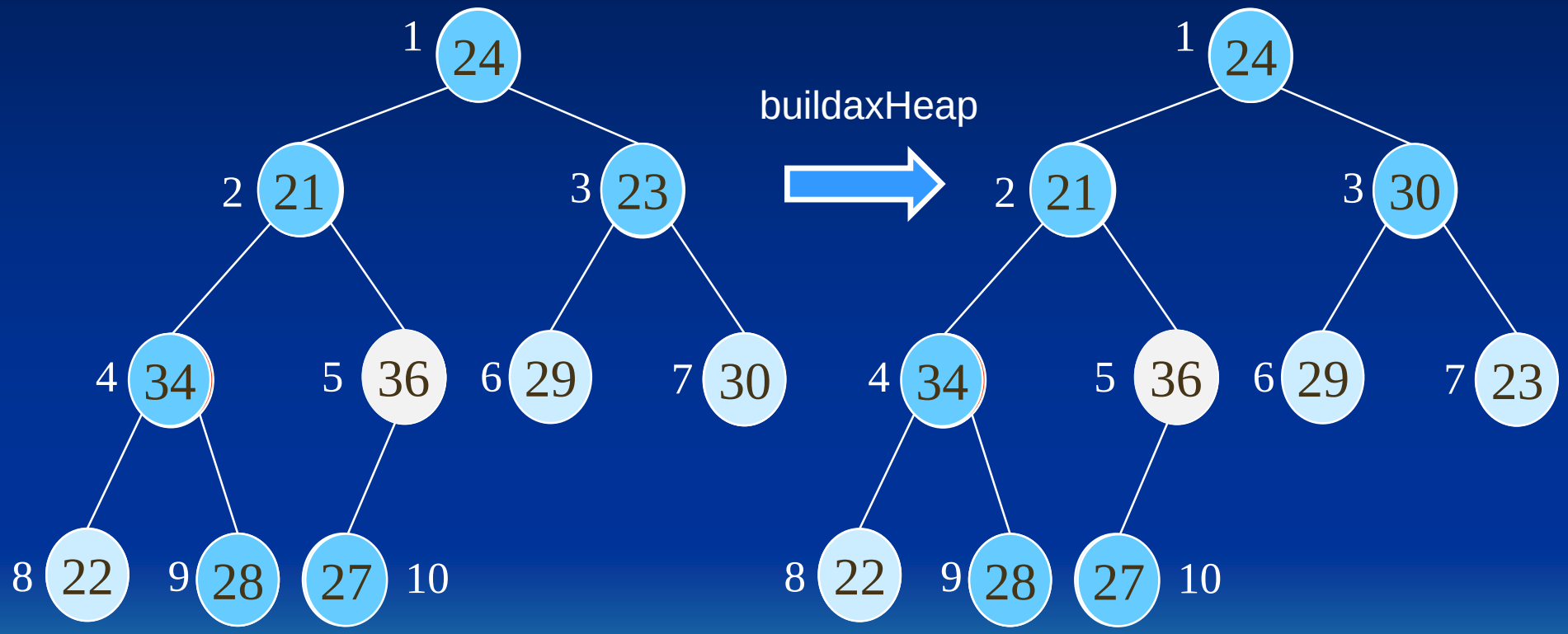
MaxHeapify(⌊10/2⌋ = 5):

BuildMaxHeap – Example



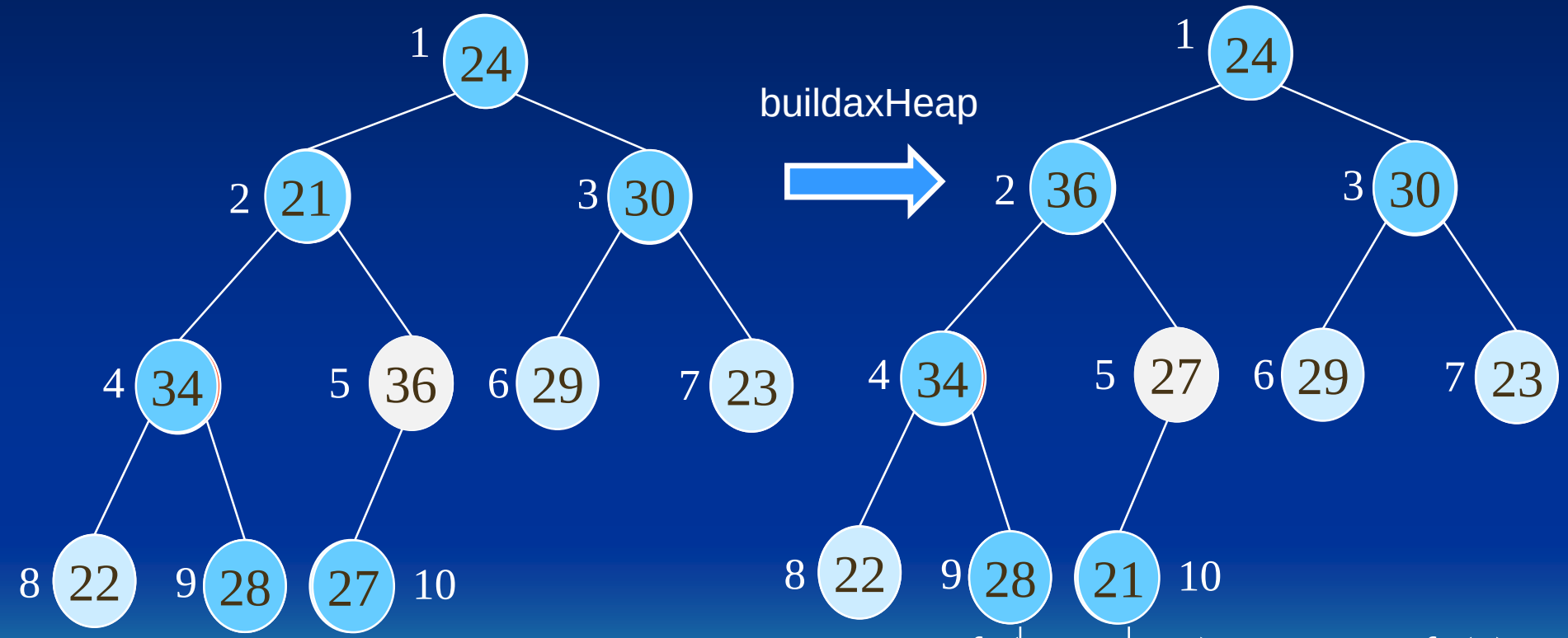
MaxHeapify(⌊10/2⌋ = 5), MaxHeapify(4)

BuildMaxHeap – Example



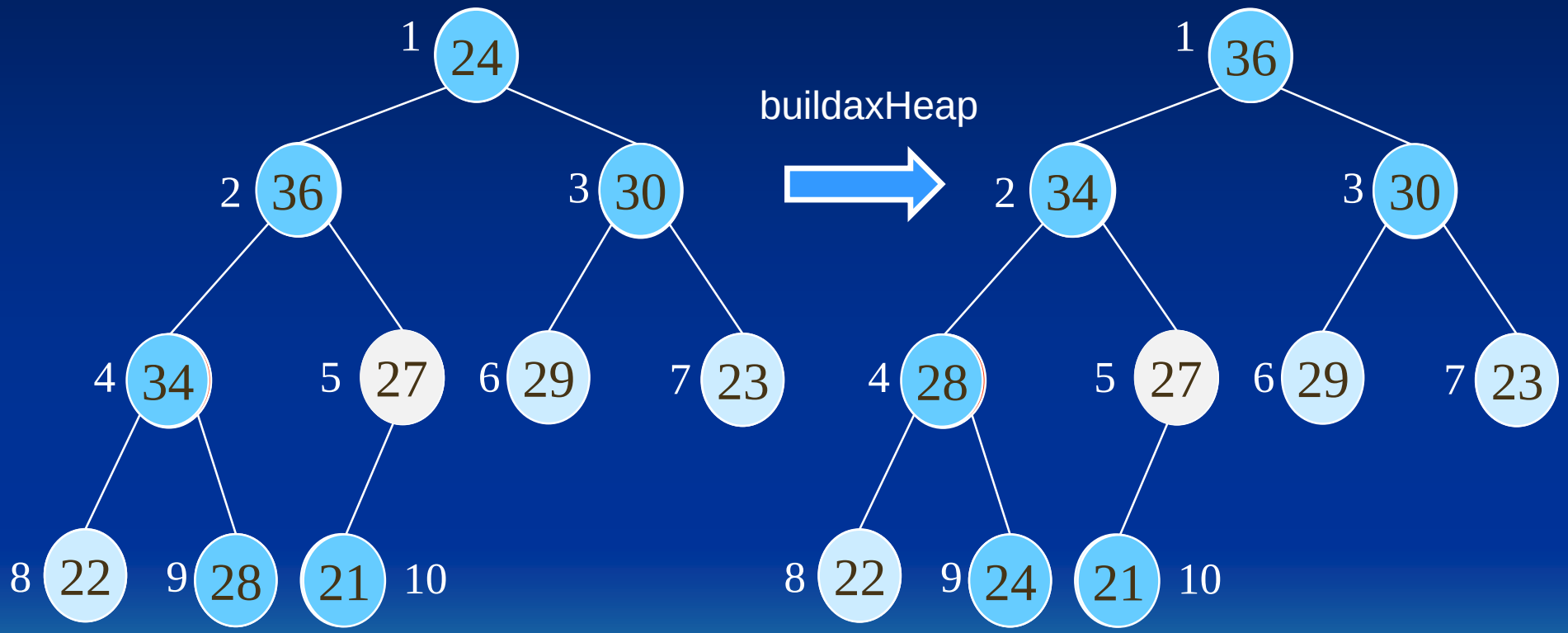
MaxHeapify(⌊10/2⌋ = 5), MaxHeapify(4), MaxHeapify(3)

BuildMaxHeap – Example



MaxHeapify(⌊10/2⌋ = 5), MaxHeapify(4), MaxHeapify(3), MaxHeapify(2)

BuildMaxHeap – Example



MaxHeapify(⌊10/2⌋ = 5), MaxHeapify(4), MaxHeapify(3), MaxHeapify(2), MaxHeapify(1)

Heapsort

- Sort by maintaining the as yet unsorted elements as a max-heap.
- Start by building a max-heap on all elements in *A.*
  - Call *BuildMaxHeap*(*A*)
  - Maximum element is in the root, *A*[1].
- Move the maximum element to its correct final position.
  - Exchange *A*[1] with *A*[*n*].
- Discard *A*[*n*] – it is now sorted.
  - Decrement heap-size[*A*].
- Restore the max-heap property on *A*[1..*n*–1].
  - Call *MaxHeapify*(*A*, 1).
- Repeat until heap-size[*A*] is reduced to 2.

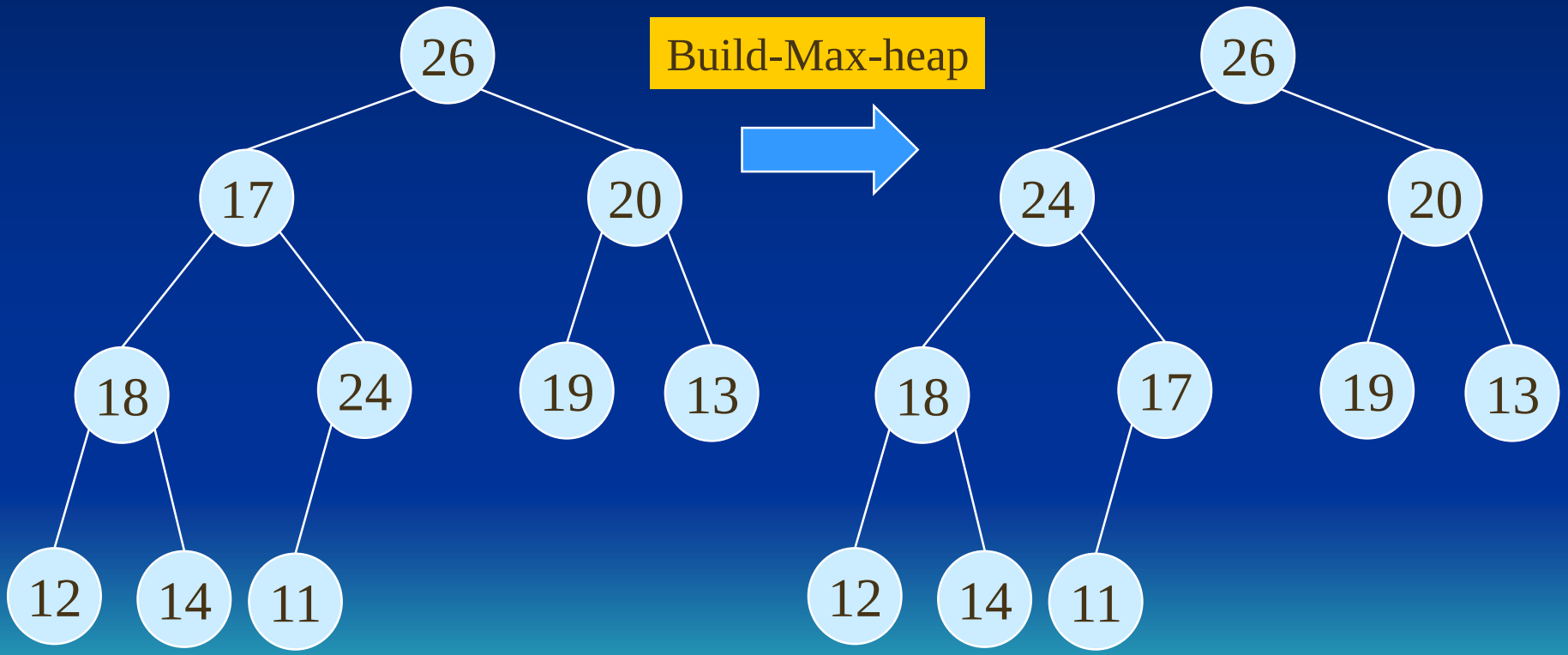
Heapsort(A)

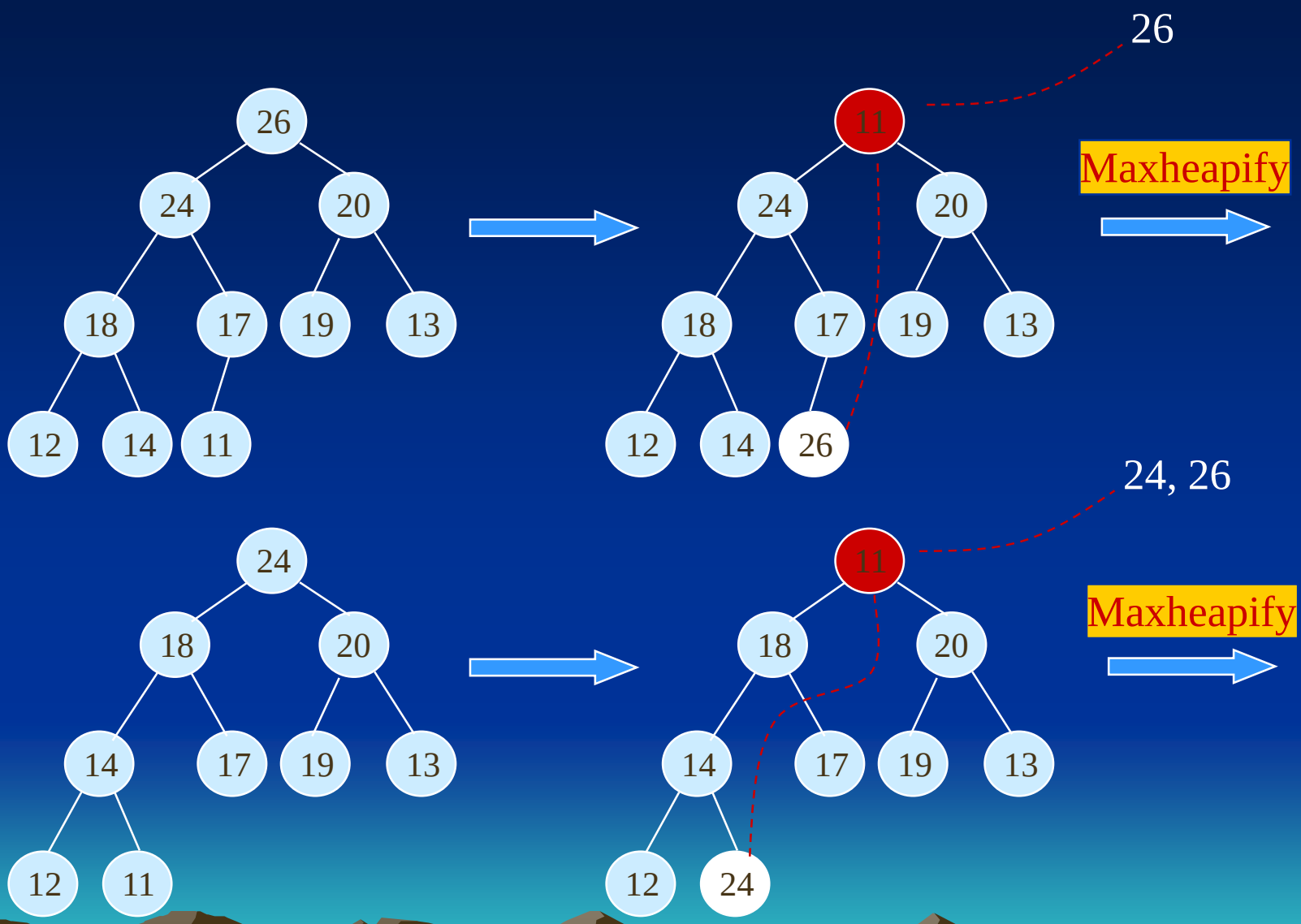
- HeapSort(A)
- 1. BuildMaxHeap(A)
- 2. for i ← length[A] downto 2
- 3. do exchange A[1] ↔ A[i]
- 4. heap-size[A] ← heap-size[A] - 1
- 5 MaxHeapify(A, 1)

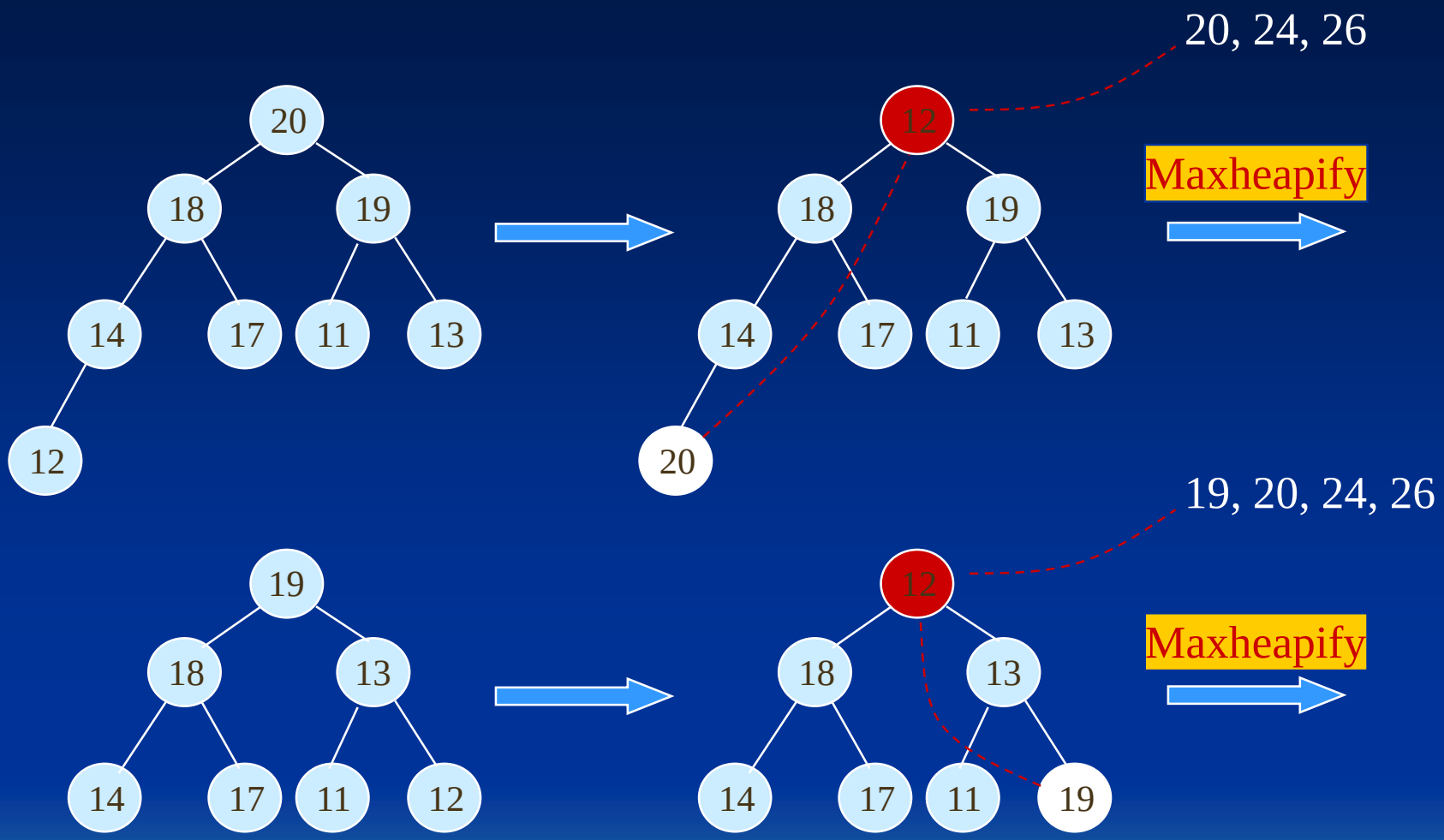
Time complexity: $O(n \cdot \log_2 n)$

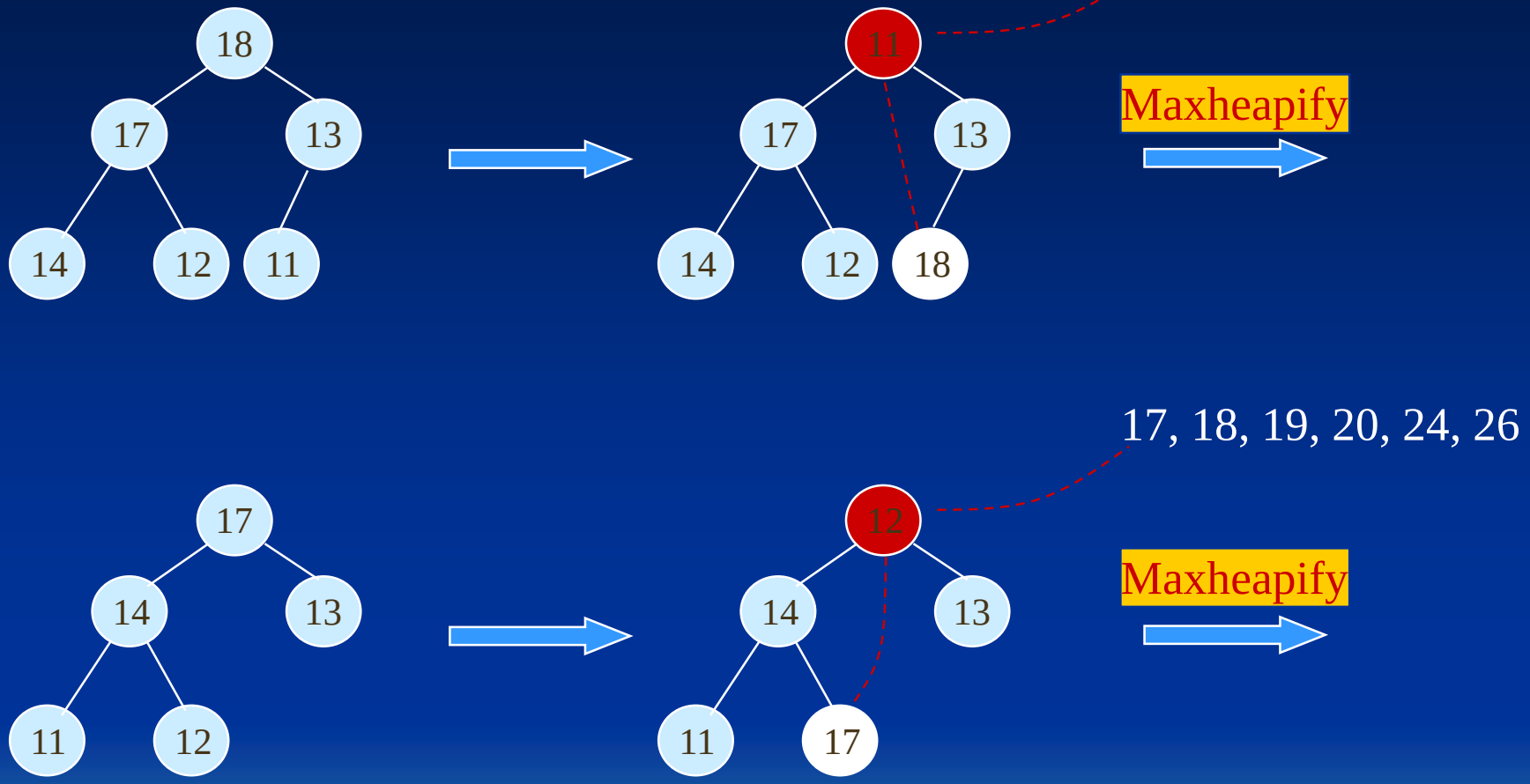
Heapsort – Example

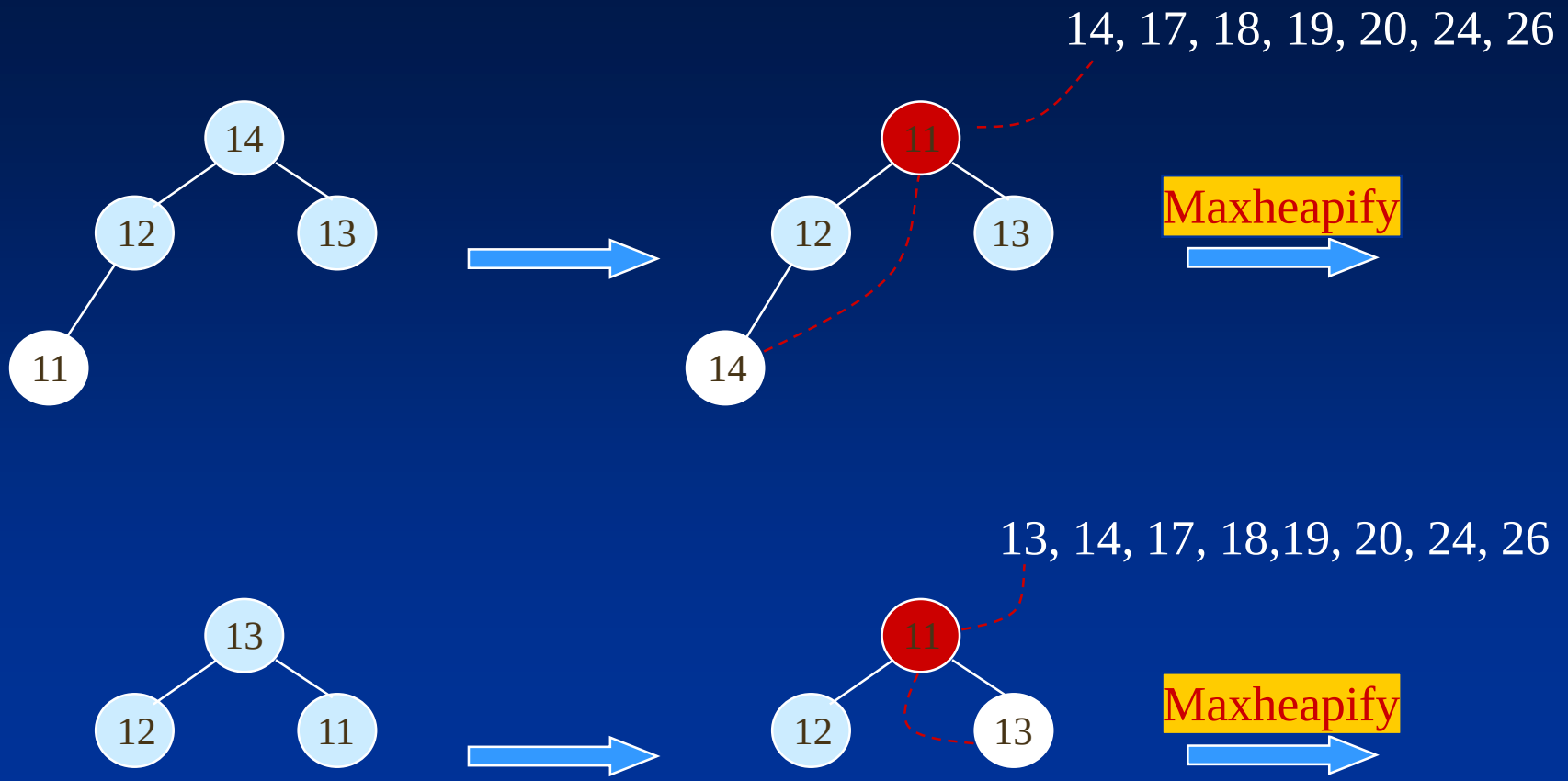
26	17	20	18	24	19	13	12	14	11
1	2	3	4	5	6	7	8	9	10




26
26
Maxheapify
24, 26
Maxheapify

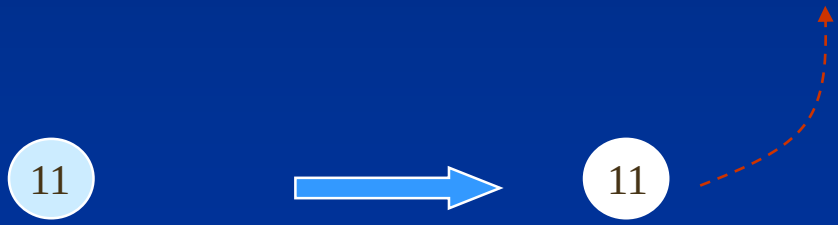

20, 24, 26
20
18
19
14
17
11
13
12
12
18
19
14
17
11
13
20
Maxheapify
19, 20, 24, 26
19
18
13
14
17
11
12
12
18
13
14
17
11
19
Maxheapify


18, 19, 20, 24, 26
18
17
13
14
12
11
11
17
13
14
12
18
Maxheapify
17, 18, 19, 20, 24, 26
17
14
13
11
12
12
14
13
11
17
Maxheapify


14, 17, 18, 19, 20, 24, 26
14
12
13
11
11
12
13
14
Maxheapify
13, 14, 17, 18,19, 20, 24, 26
13
12
11
11
12
13
Maxheapify

12, 13, 14, 17, 18, 19, 20, 24, 26

11, 12, 13, 14, 17, 18,19, 20, 24, 26



Algorithm Analysis

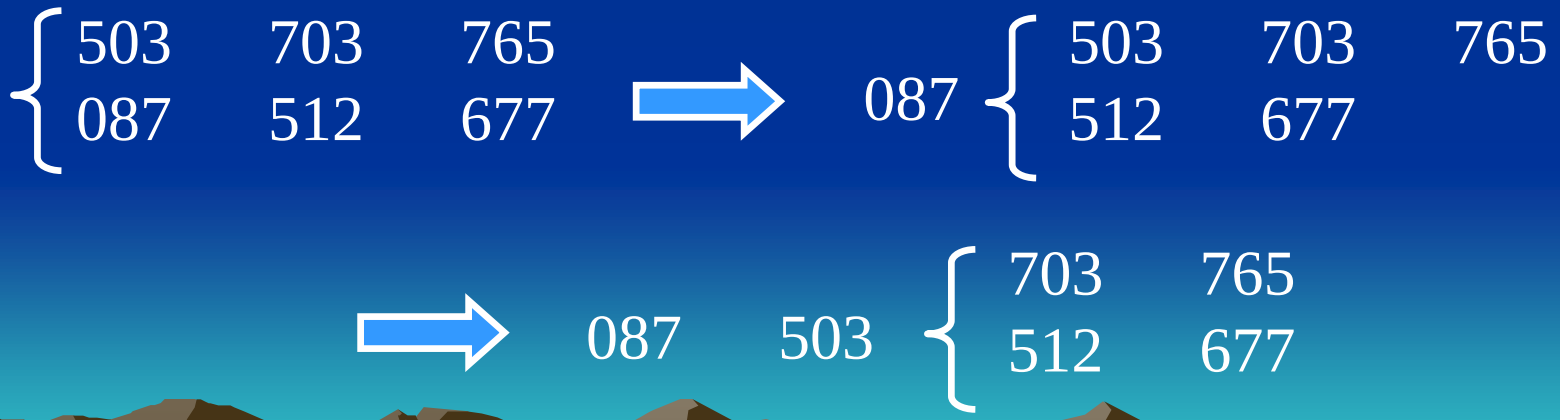
HeapSort(*A*)

1. Build-Max-Heap(*A*)
2. **for** *i* ← *length*[*A*] **downto** 2
3. **do** exchange *A*[1] ↔ *A*[*i*]
4. *heap-size*[*A*] ← *heap-size*[*A*] – 1
5. *MaxHeapify*(*A*, 1)

- In-place
- BuildMaxHeap takes $O(n)$ calls and each of the $n-1$ calls to MaxHeapify takes time $O(\lg n)$.
- Therefore, $T(n) = O(n \lg n)$

- **Basic algorithms**
  - Sorting by merging

    Merging means the combination of two or more ordered sequence into a single sequence. For example, can merge two sequences: 503, 703, 765 and 087, 512, 677 to obtain a sequence: 087, 503, 512, 677, 703, 765. A simple way to accomplish this is to compare the two smallest items, output the smallest, and then repeat the same process.



- **Basic algorithms**
  - Merging algorithm

Algorithm Merge(s1, s2)

Input: two sequences: s1 - $x1 \le x2 \ldots \le x_m$ and s2 - $y1 \le y2 \ldots \le y_n$
Output: a sorted sequence: $z1 \le z2 \ldots \le z_{m+n}$.

1.[initialize] $i := 1, j := 1, k := 1$;
2.[find smaller] if $x_i \le y_j$ goto step 3, otherwise goto step 5;
3.[output x_i] $z_k := x_i$, $k := k+1$, $i := i+1$. If $i \le m$, goto step 2;
4.[transmit $y_j \le \ldots \le y_n$] $z_k, \ldots, z_{m+n} := y_j, \ldots, y_n$. Terminate the algorithm;
5.[output y_j] $z_k := y_j$, $k := k+1$, $j := j+1$. If $j \le n$, goto step 2;
6.[transmit $x_i \le \ldots \le x_m$] $z_k, \ldots, z_{m+n} := x_i, \ldots, x_m$. Terminate the algorithm;

- **Basic algorithms**
  - Merge-sorting

Algorithm *Merge-sorting*(s)

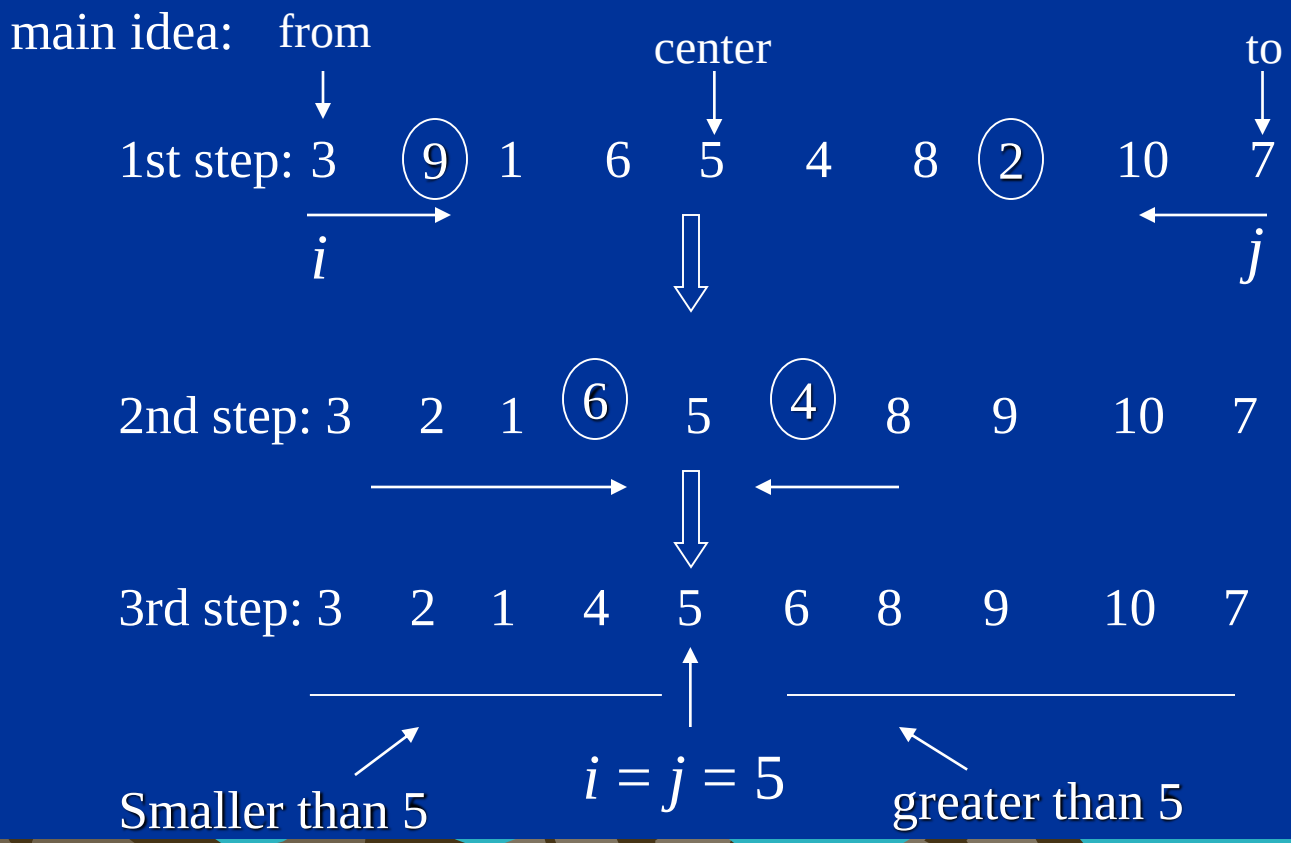
Input: a sequences s = < x_1, ..., x_m>

Output: a sorted sequence.

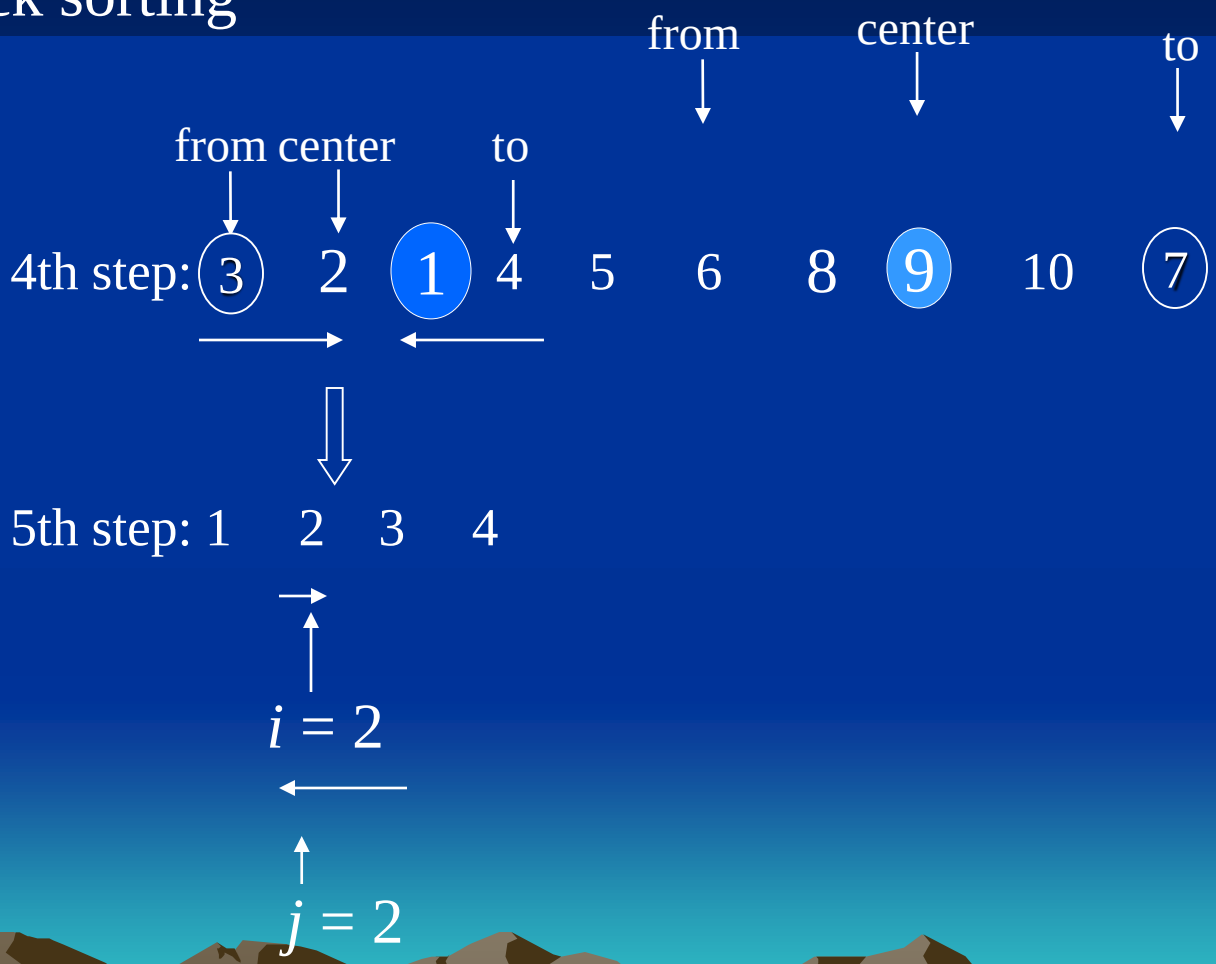
1. If |s| = 1, then return s;
2. $k := \lceil m/2 \rceil$;
3. s1 := Merge-sorting(x_1, ..., x_k);
4. s2 := Merge-sorting(x_{k+1}, ..., x_m);
5. return(Merge(s1, s2));

- **Basic algorithms**
  - quick sorting

Pivot is 5. It will be stored in a temporary variable.

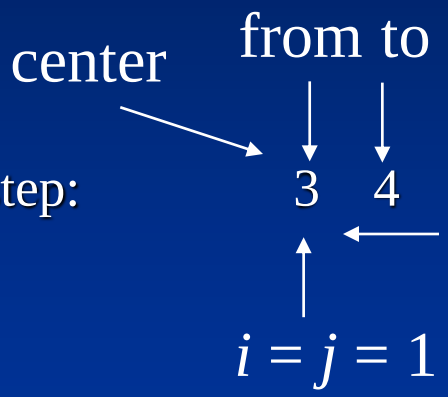


- **Basic algorithms**
  - quick sorting



6th step: 1

The sequence contains only one element, no sorting.



The center element is 3.

8th step: 4

The sequence contains only one element, no sorting.

1 2 3 4 5

6 8 9 10 7

6 7 8 9 10

- **Basic algorithms**

  - quick sorting

    main idea:

    Algorithm *quick_sort*(from, center, to)

    Input: from - pointer to the starting position of array *A*

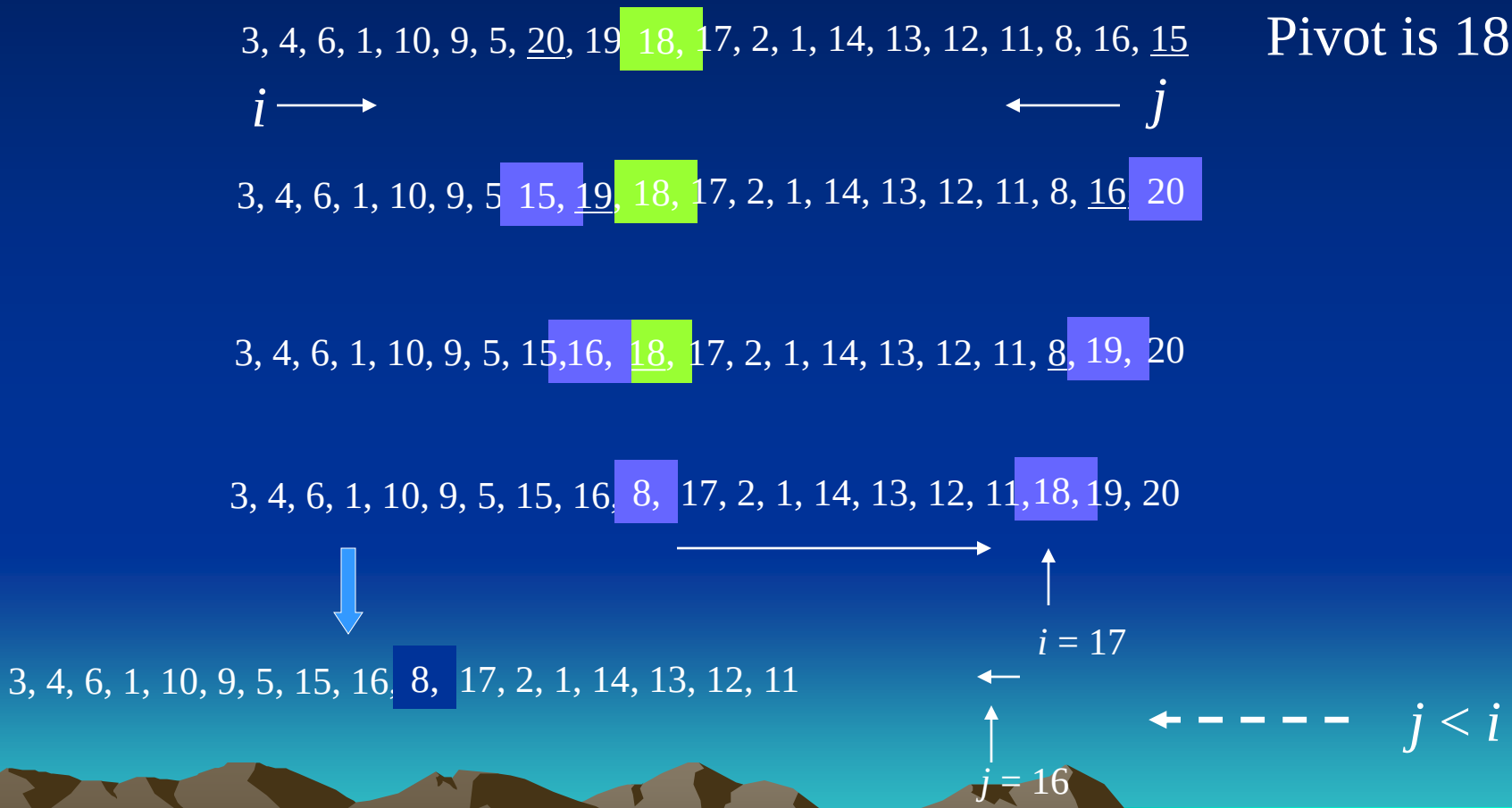
    center - pointer to the middle position of array *A*

    to - pointer to the end position of array *A*

    Output: sorted array: *A*'

    0. i := from; j := to; if i = j, return;
    1. Find the first element *a* = *A*(i) larger than or equal to *A*(center) from *A*(from) to *A*(to); (i is used to scan from left to right.)
    2. Find the first element *b* = *A*(j) smaller than or equal to *A*(center) from *A*(to) to *A*(from); (j is used to scan from right to left.)
    3. If i < j then exchange *a* and *b;*
    4. Repeat step from 1 to 3 until j ≤ i;
    5. If from < j then recursive call *quick_sort*(from,(from + j)/2, j);
    6. If i < to then recursive call *quick_sort*(i, (i+ to)/2, to);

- **Basic algorithms**
  - quick sorting



- **Basic algorithms**
  - External sorting method:

    Sort records in a large file stored on disk that does not fit entirely in main memory.

    sort-merge sorting:

    1. Divide the file into small files - called *runs*;
    2. Sort phase:

       Sort runs in the *buffer space* in main memory;
    3. Merge phase:

       Merge the sorted runs in turn.

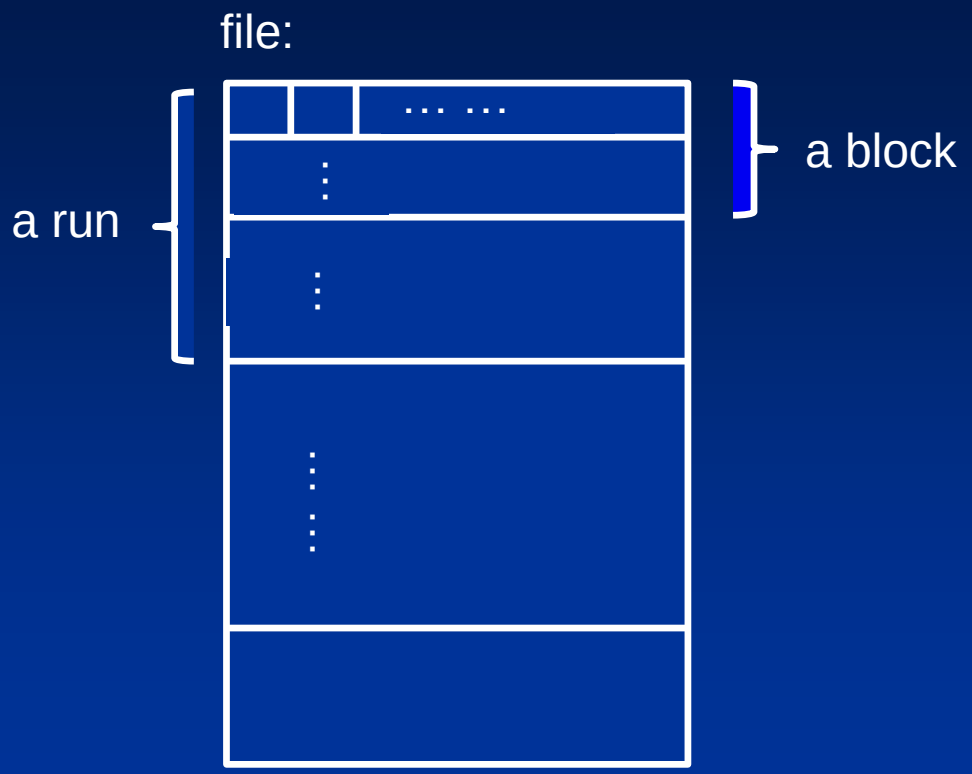
- **Basic algorithms**
  - External sorting method:

    Several parameters:

    b - number of file blocks

    n_R - number of initial runs

    n_B - available buffer space



$$n_R = \lceil b/n_B \rceil$$

Example: b = 80 blocks, n_B = 5 blocks

n_R = 16 initial runs

d_M - number of runs that can be merged together in each pass

- **Basic algorithms**
  - External sorting method:

set $i \leftarrow 1$;

$j \leftarrow$ b; /*size of the file in blocks*/

$k \leftarrow n_B$; /*size of buffer in blocks*/

$m \leftarrow \lceil j/k \rceil$; /*number of runs*/

/*sort phase*/

while ($i \leq m$)

do {read next k blocks of the file into the buffer or if there are less than k blocks remaining then read in the remaining blocks;

sort the records in the buffer and write as a temporary subfile;

$i \leftarrow i + 1$;

}

- **Basic algorithms**
  - External sorting method:

/*merge phase: merge subfiles until only 1 remains*/

set $i \leftarrow 1$;

$p \leftarrow \lceil \log_{k-1} m \rceil$; /*$p$ is the number of passes for the merging phase*/

$j \leftarrow m$; /*m is number of runs*/

while ($i \leq p$) do

{ $n \leftarrow 1$;

$q \leftarrow \lceil j/(k-1) \rceil$; /*$q$ is the number of subfiles to write in this pass*/

while ($n \leq q$) do

{ read next k-1 subfiles or remaining subfiles (from previous pass) one block at a time;

merge these subfiles and write the result as a new subfile;

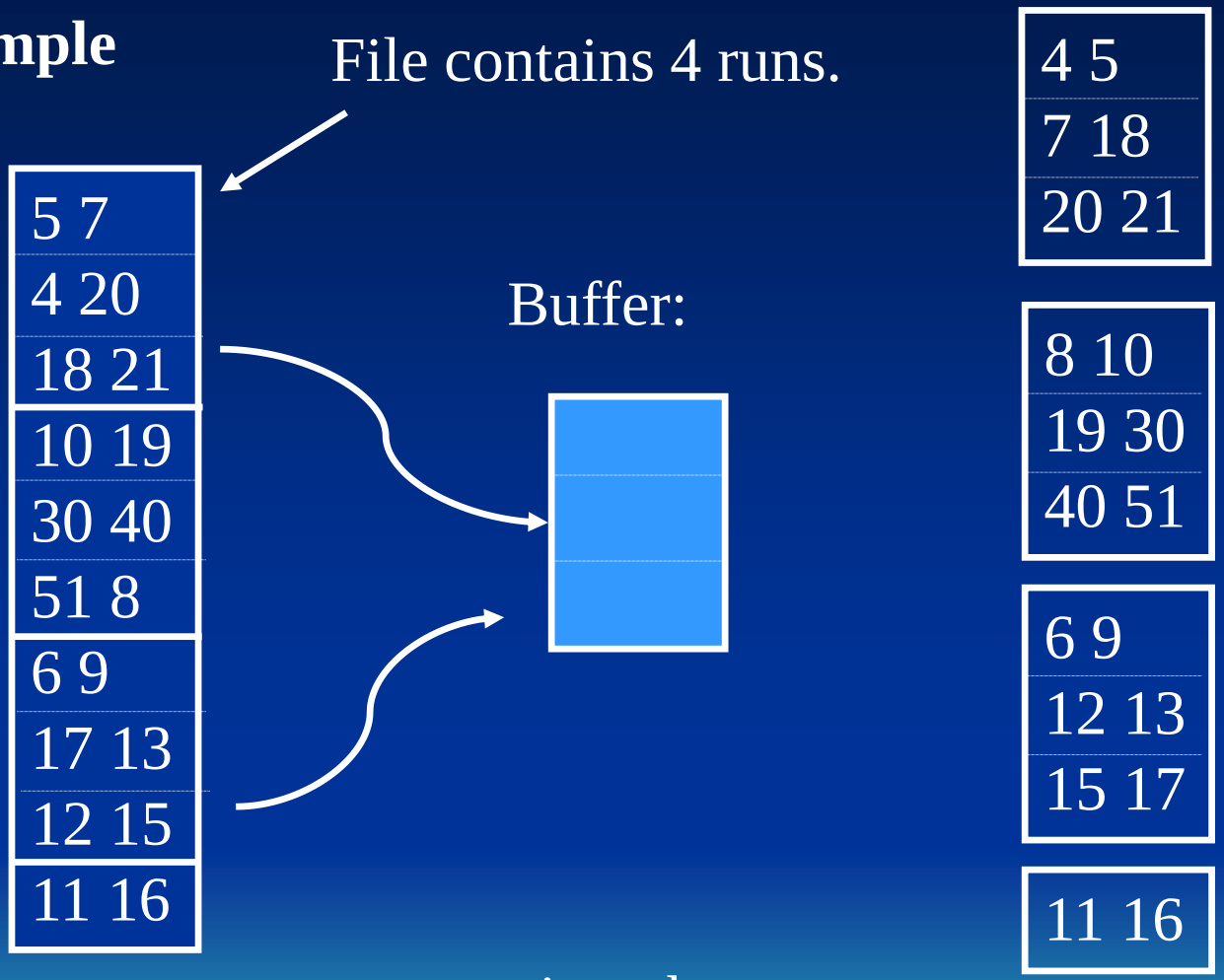
$n \leftarrow n + 1$;

}

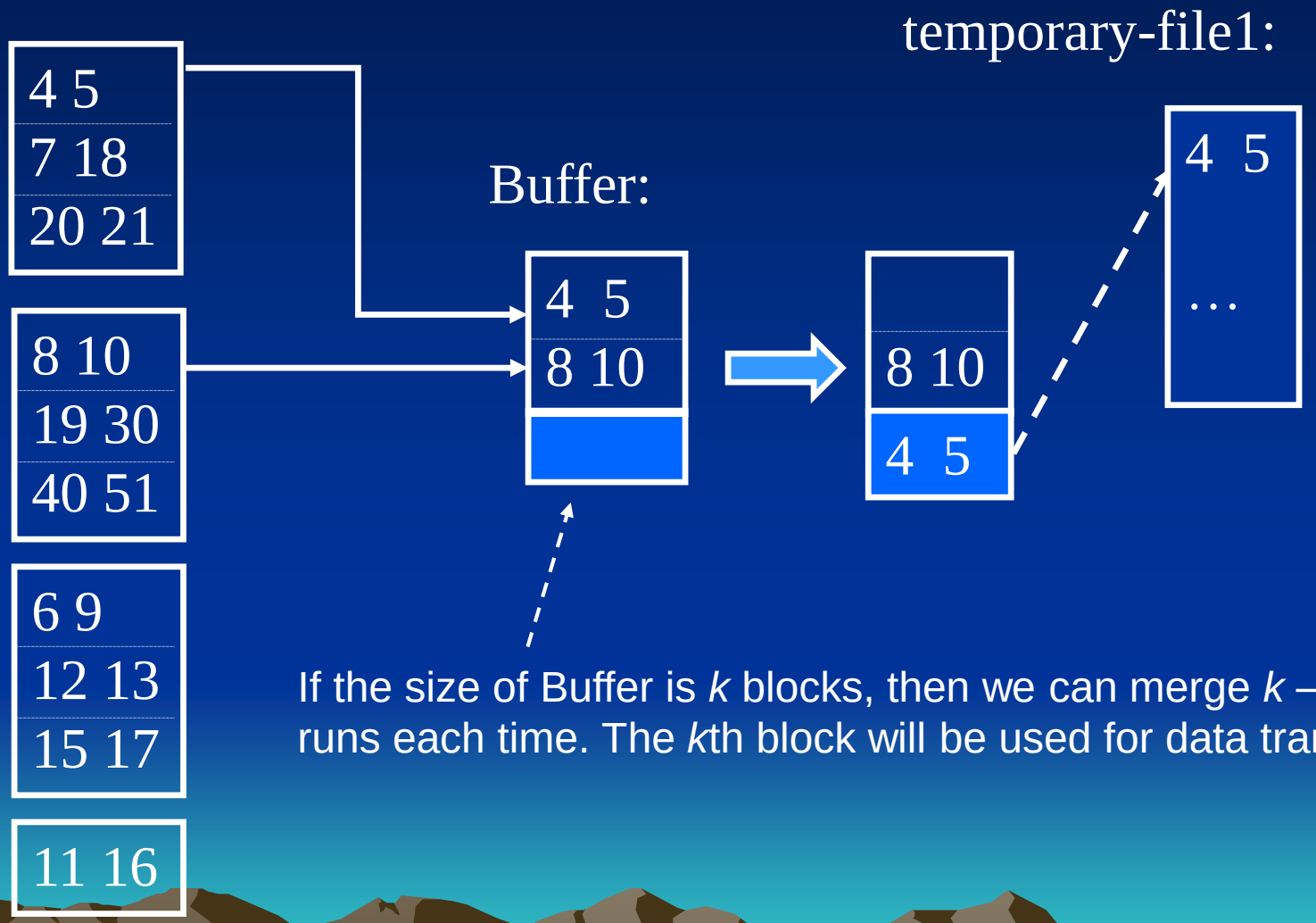
$j \leftarrow q$; $i \leftarrow i + 1$; }

}

- **Example**

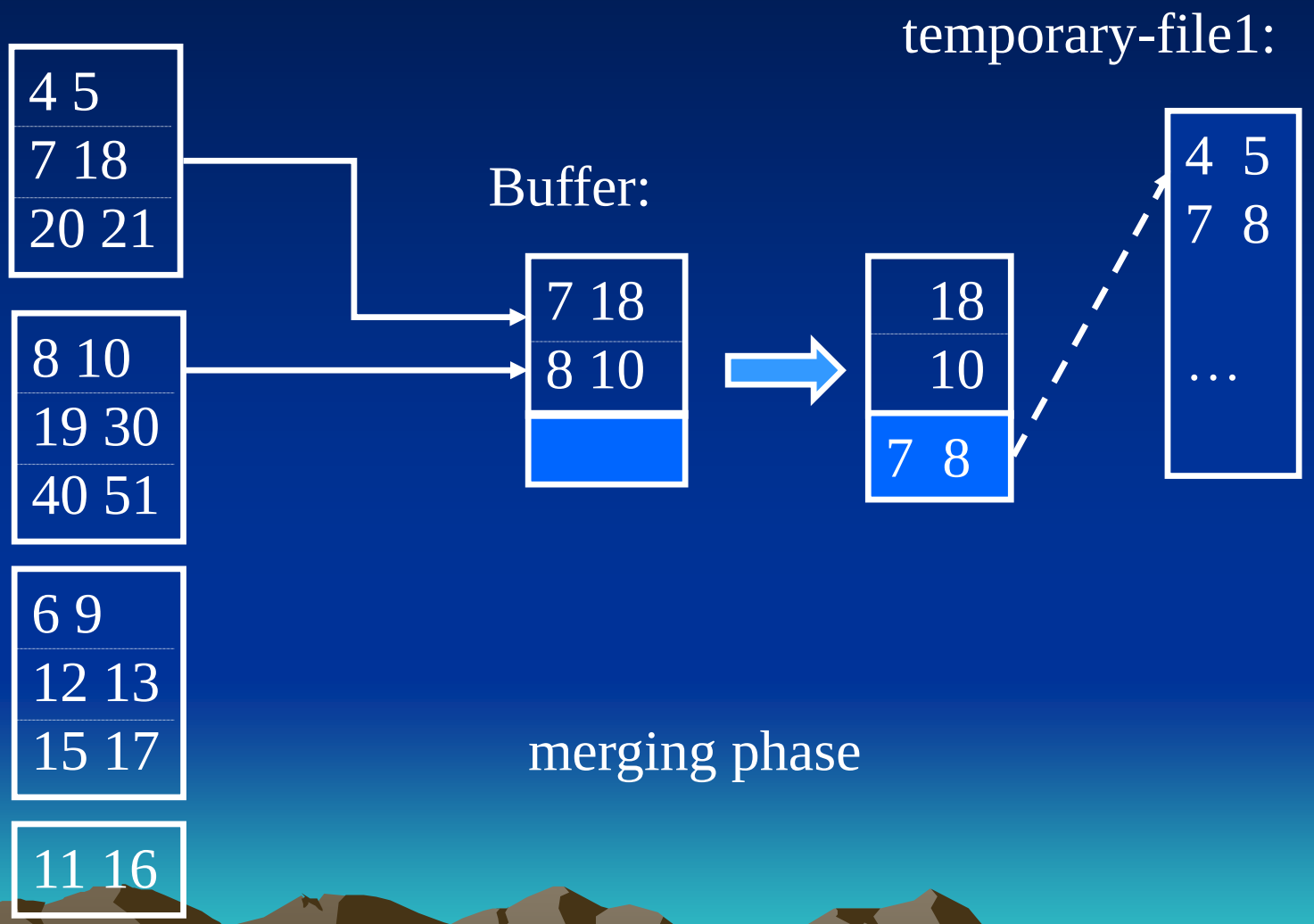

File contains 4 runs.
5 7
4 20
18 21
10 19
30 40
51 8
6 9
17 13
12 15
11 16
Buffer:
4 5
7 18
20 21
8 10
19 30
40 51
6 9
12 13
15 17
11 16
sorting phase

- **Merging phase: first pass**

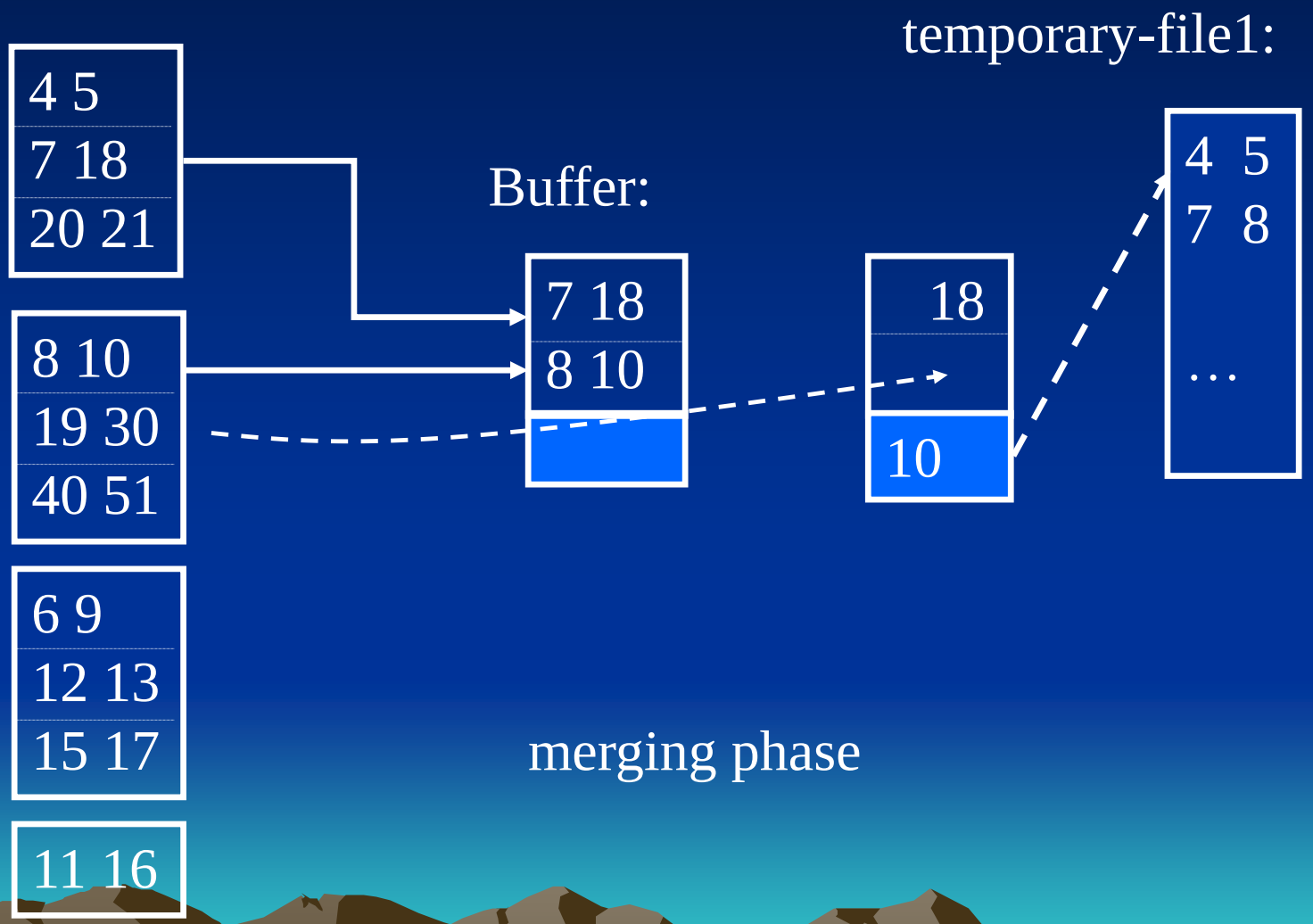


If the size of Buffer is *k* blocks, then we can merge *k* – 1 runs each time. The *k*th block will be used for data transfer.

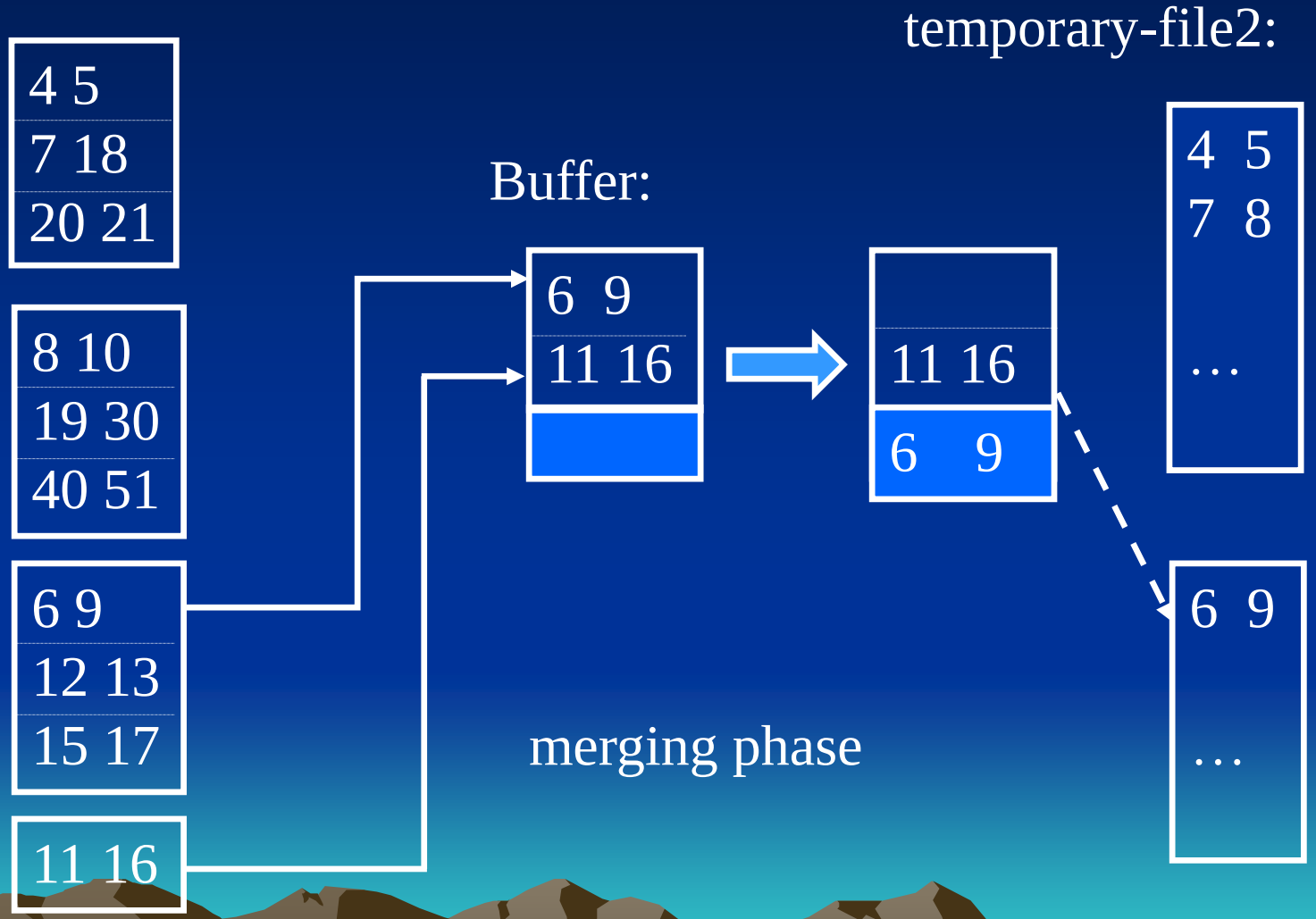
- **Example**


temporary-file1:
4 5
7 18
20 21
Buffer:
8 10
19 30
40 51
6 9
12 13
15 17
11 16
7 18
8 10
18
10
7 8
4 5
7 8
…
merging phase

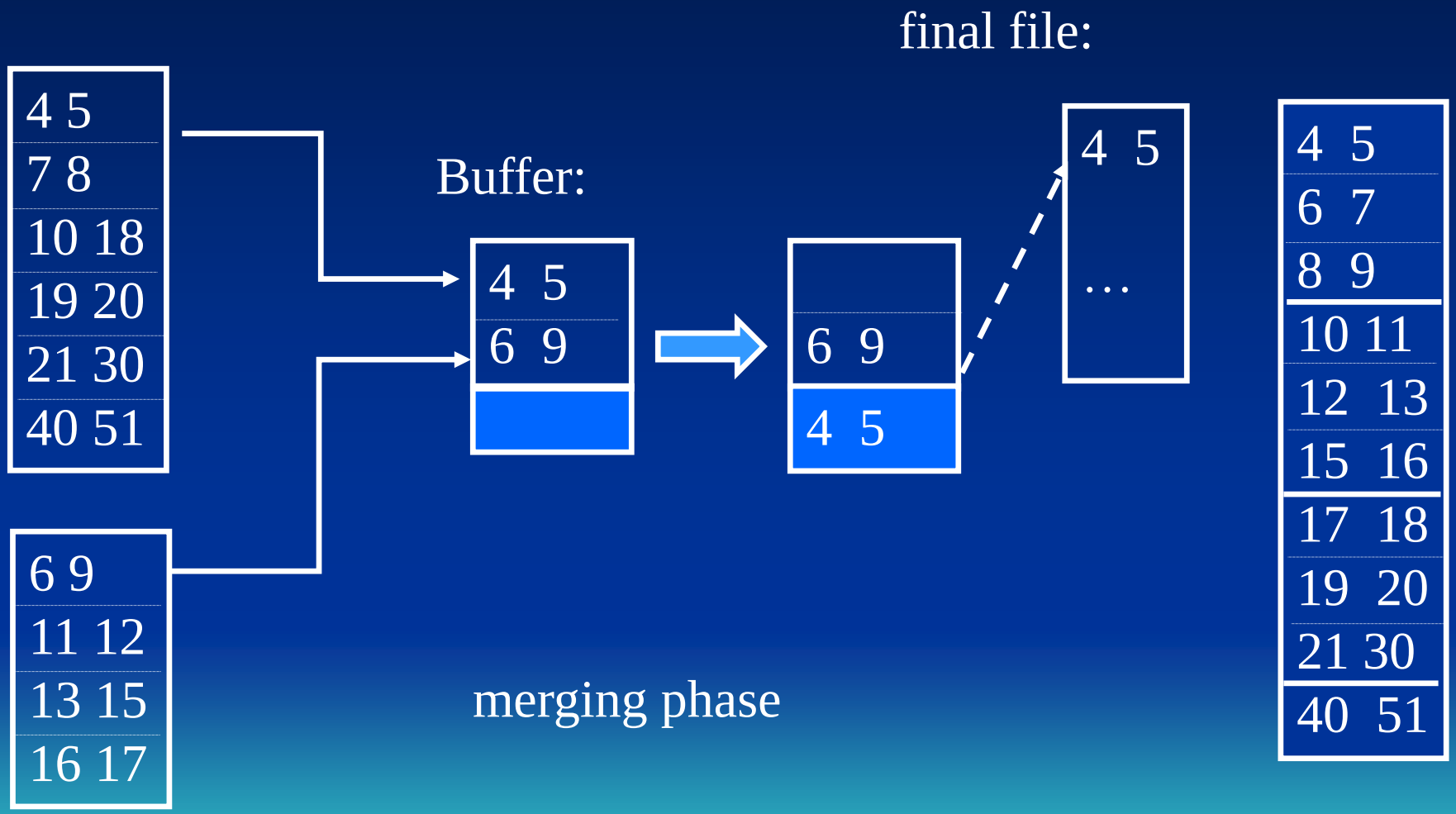
- **Example**

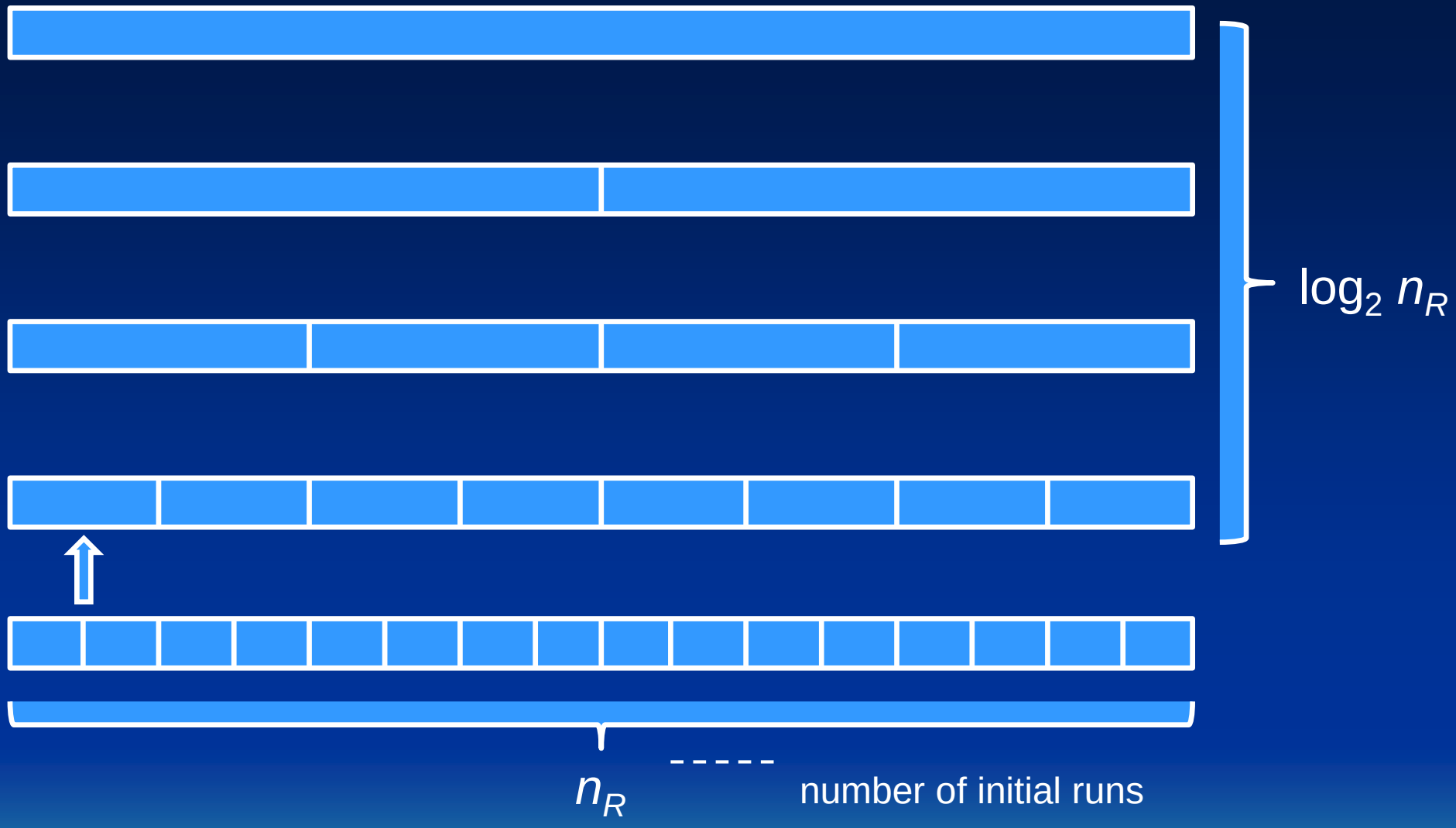

temporary-file1:
4 5
7 18
20 21
Buffer:
8 10
19 30
40 51
6 9
12 13
15 17
11 16
7 18
8 10
18
10
4 5
7 8
…
merging phase

- **Example**


temporary-file2:
4 5
7 18
20 21
8 10
19 30
40 51
6 9
12 13
15 17
11 16
Buffer:
6 9
11 16
11 16
6 9
4 5
7 8
…
6 9
…
merging phase

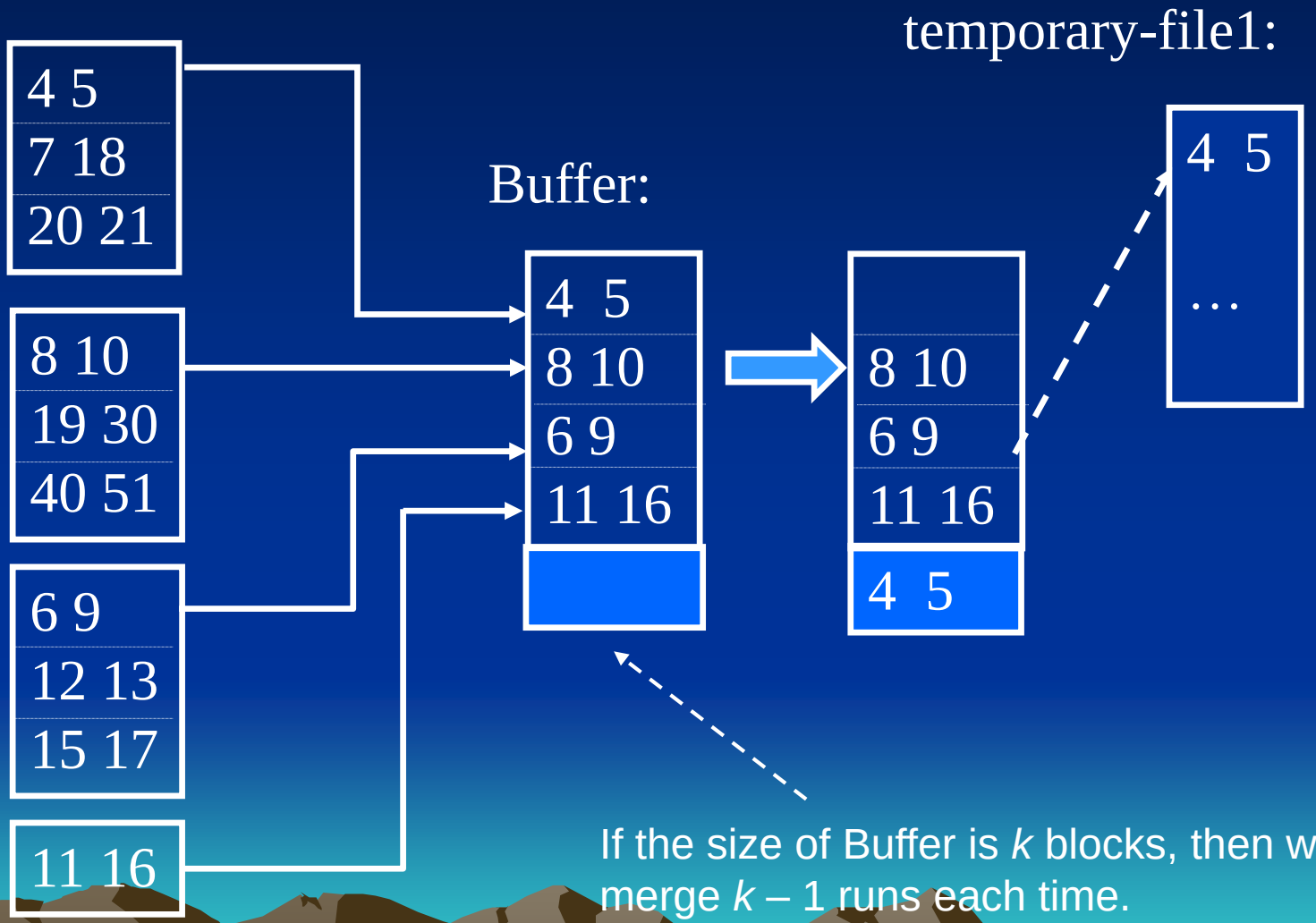
- **Merging phase: second pass**



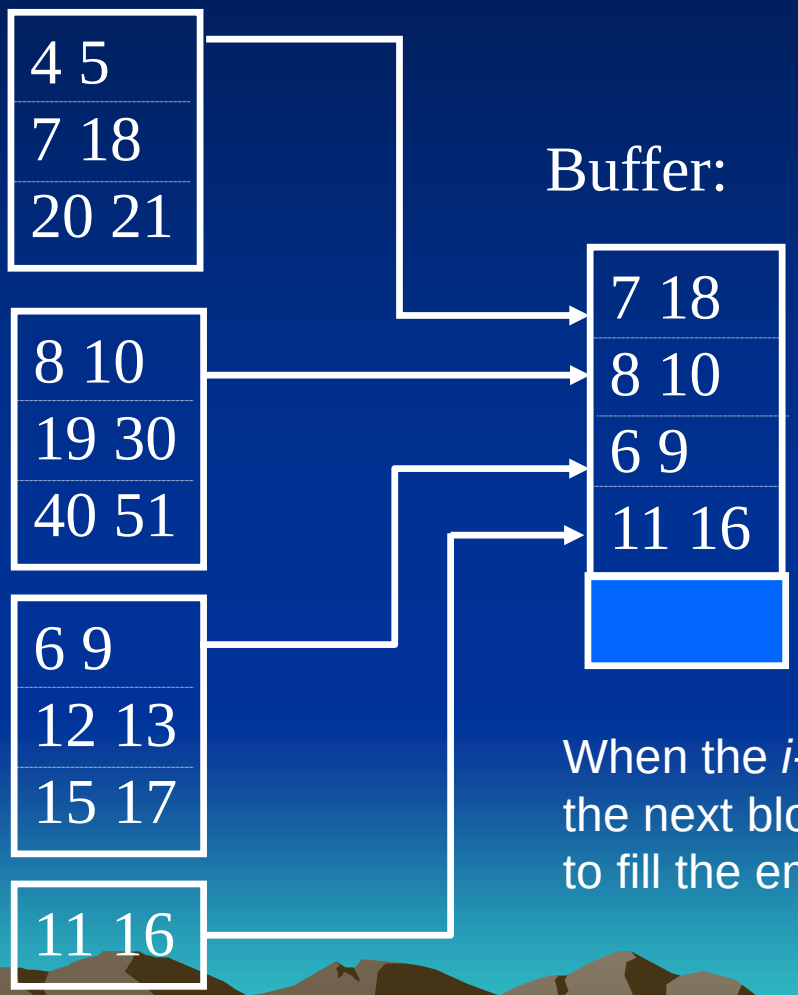


In general, $\log_{k-1} n_R$ passes should be conducted if the buffer size is k.

Comments on merging process:



Comments on merging process:



temporary-file1:

When the *i*-th block in the buffer becomes empty, the next block in the *i*-th run will be taken to fill the empty block.

- **Basic algorithms**
  - SELECT operation

    $\sigma_{\text{<search condition>}}(R)$

  Example:

  (op1): $\sigma_{ssn='123456789'}(EMPLOYEE)$

  (op2): $\sigma_{DNUMBER>5}(DEPARTMENT)$

  (op3): $\sigma_{DNO=3}(EMPLOYEE)$

  (op4): $\sigma_{DNO=1 \wedge SALARY>70000 \wedge SEX='F'}(EMPLOYEE)$

  (op5): $\sigma_{ESSN='123456789' \wedge PNO=10}(WORKS\_ON)$

  (op6): $\sigma_{superSSN='123456789' \wedge DNO=5}(EMPLOYEE)$

- **Basic algorithms**
  - Search method for simple selection
    - file scan

      linear search (brute force)

      binary search
    - index scan

      using a primary index (or hash key)

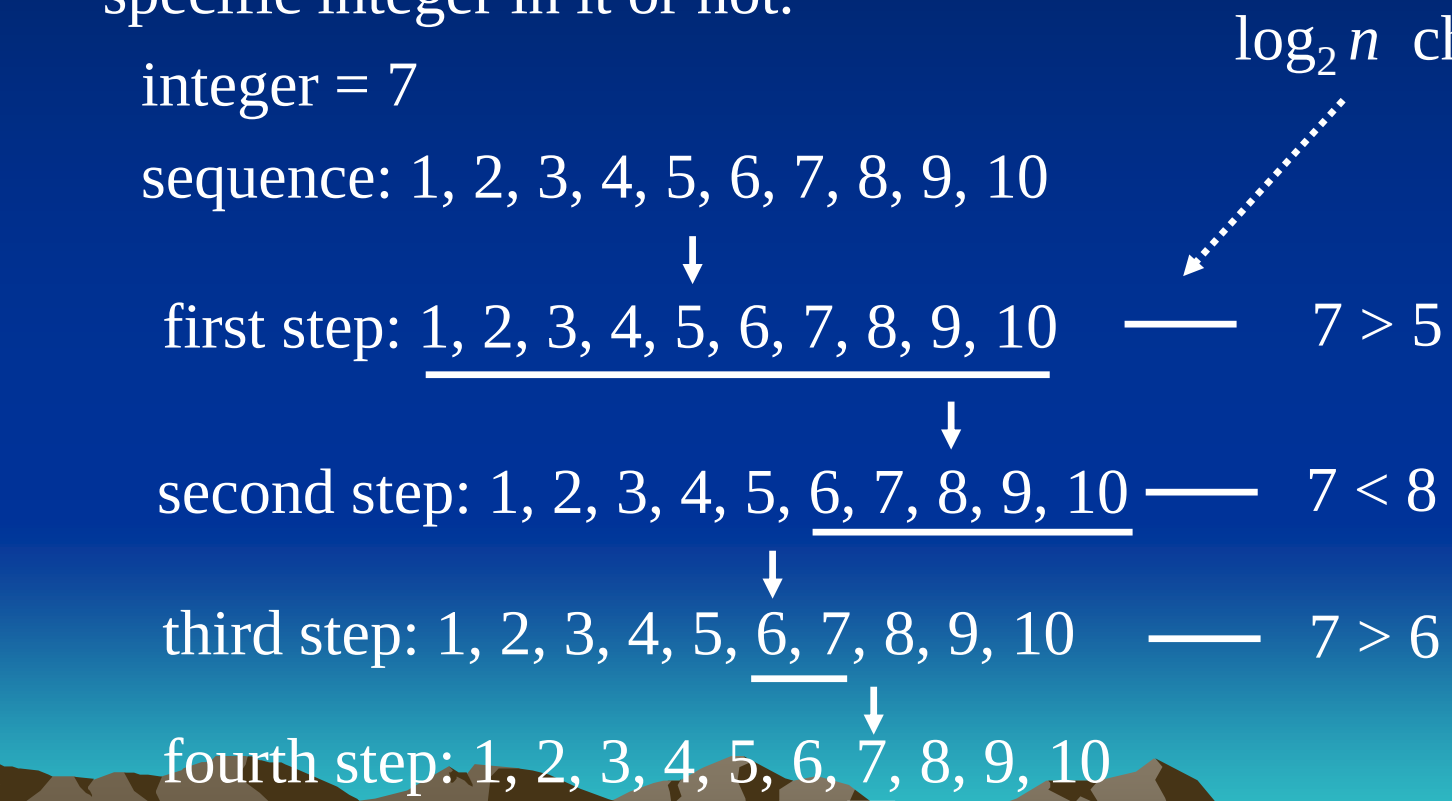
      using a primary index to retrieve multiple records

      using a clustering index to retrieve multiple records

      using a multiple level index to retrieve multiple records

- **Basic algorithms**
  - Binary search

    Search a sorted sequence of integers to see whether a specific integer in it or not.

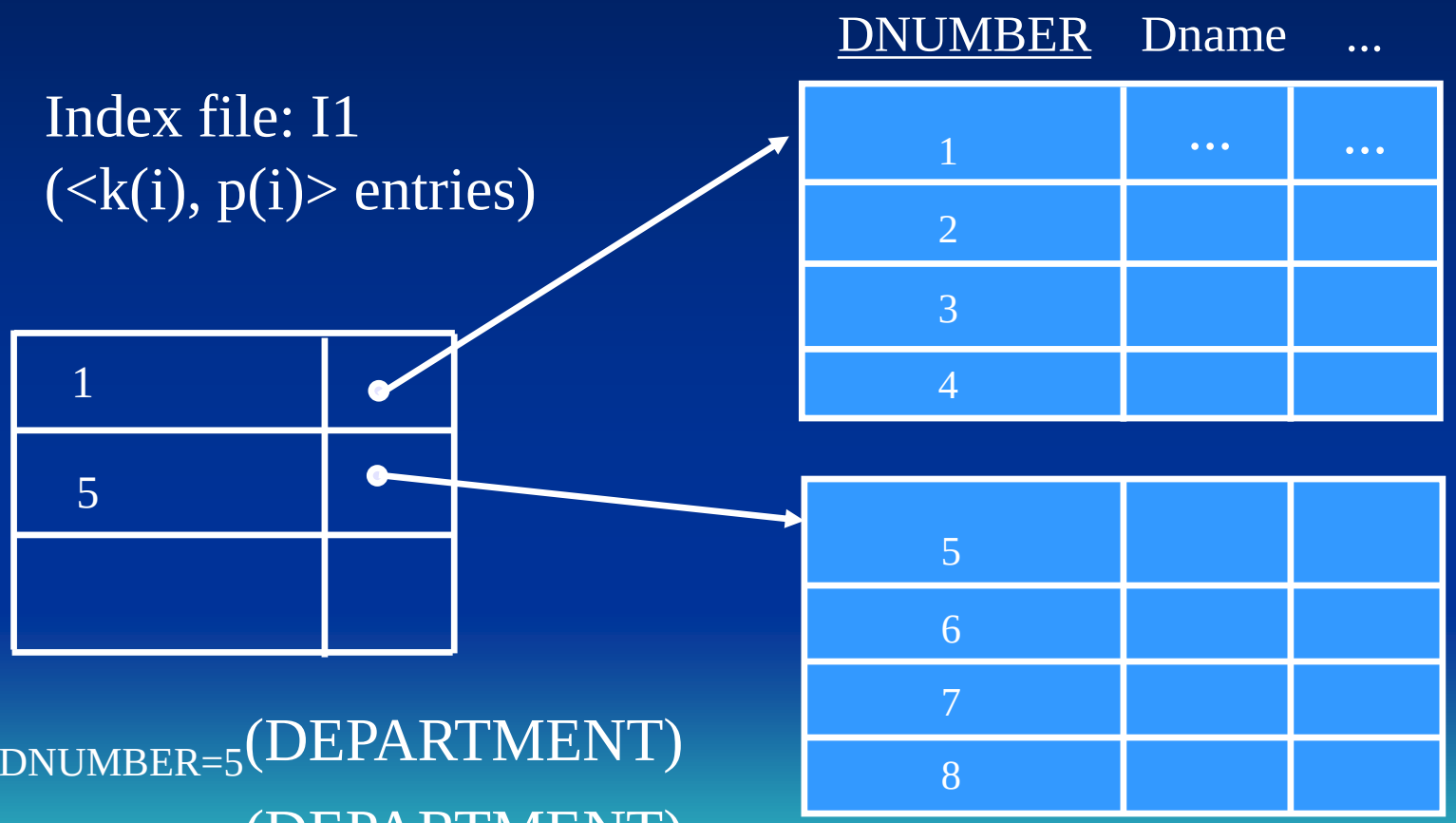


- **Basic algorithms**
  - Using a primary index to retrieve multiple records
    If the selection condition is >, >=, <, <= on a key field with a primary index, use the index to find the record satisfying the corresponding equality condition (DNUMBER = 5, in $\sigma_{DNUMBER>5}$(DEPARTMENT)), then retrieve all subsequent records in the ordered file.

- **Basic algorithms**
  - using a primary index

Primary index:

Data file: DEPARTMENT



$\sigma_{DNUMBER=5}(DEPARTMENT)$

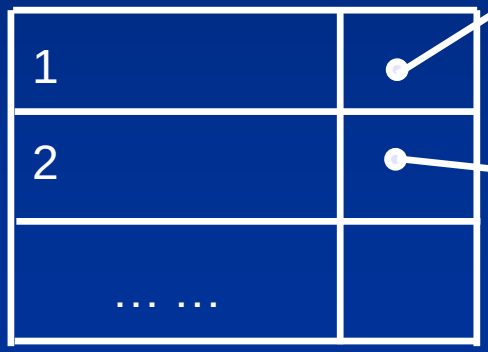
$\sigma_{DNUMBER>5}(DEPARTMENT)$

Basic algorithms

- Using a clustering index to retrieve multiple records
  If the selection condition involves an equality comparison on a non-key attribute with a clustering index (for example, DNO = 2 in $\sigma_{DNO=2}$ (EMPLOYEE)), use the index to retrieve all the records satisfying the condition.

Clustering index:

Index file: I1
(<k(i), p(i)> entries)



Data file: Employee

SSN	Dno	… …
123456789	1	…
123455555	1	
223456789	1	
234447891	1	
234567891	1	
345678912	2	
345897642	2	
456789123	3	

… …

$\sigma_{DNO=2}$ (EMPLOYEE)

- **Basic algorithms**
  - Searching methods for complex selection

    Conjunctive selection using an individual index

    If an attribute involved in any single simple condition in the conjunctive has an access path that permits one of the methods discussed above, use that condition to retrieve the records and then check whether each retrieved record satisfies the remaining simple conditions in the conjunctive condition.

$$\sigma_{DNO=1 \wedge salary > 50000}(EMPLOYEE)$$

- **Basic algorithms**
  - Searching methods for complex selection

    Conjunctive selection using a composite index

    If two or more attributes are involved in equality conditions in the conjunctive condition and a composite index (or hash structure) exists on the combined fields - for example, if an index has been created on the composite key (SSN value and PNO value) of the WORKS_ON file - we can use the index directly.

$$\sigma_{SSN=123456789 \wedge PNO = 3}(WORKS\_ON)$$

Primary index:

Data file: Works_on

SSN	Pno	hours
123456789	1	…
123456789	2	
123456789	3	
234567891	1	
234567891	2	
345678912	2	
345678912	3	
456789123	1	
…	…	

Index file: I1
(<k(i), p(i)> entries)

123456789, 1	
234567891, 2	
… …	

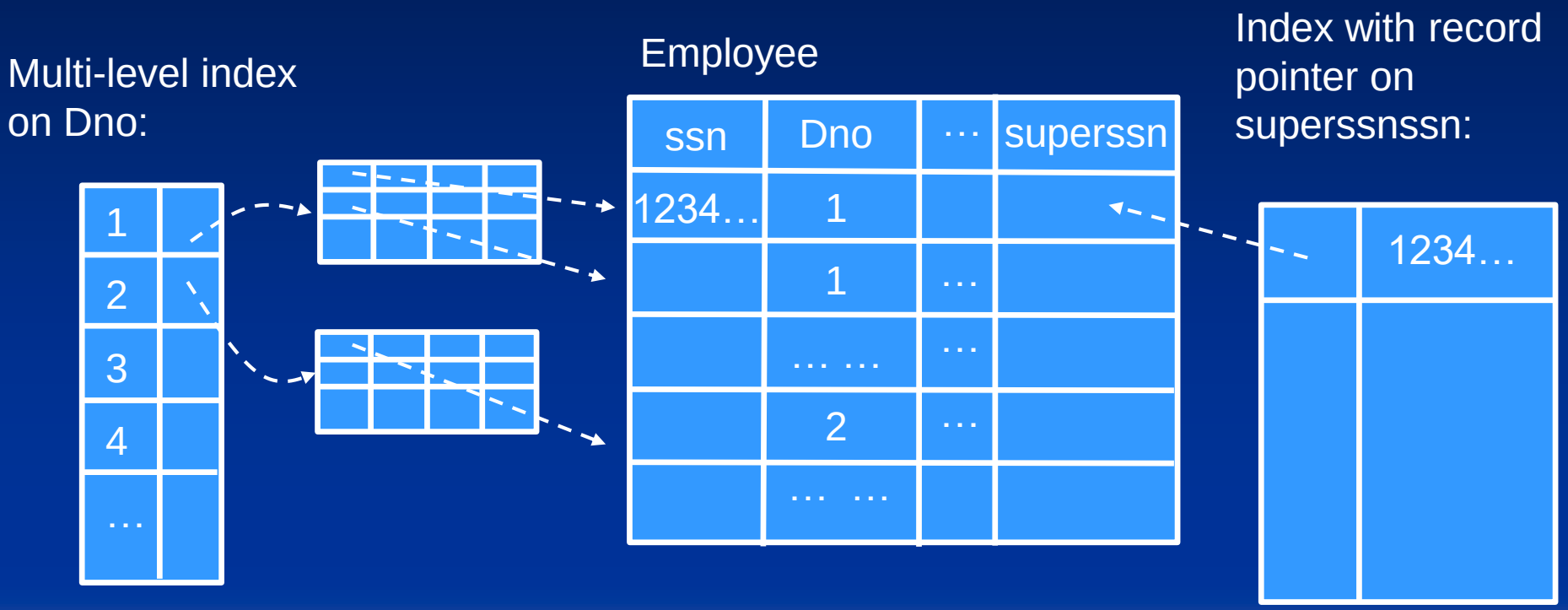
$\sigma_{SSN=123456789 \wedge PNO = 3}(WORKS\_ON)$

- **Basic algorithms**
  - Searching methods for complex selection

    Conjunctive selection by intersection of record pointers

    - Secondary indexes (indexes on any *nonordering* field of a file, which is not a key) are available on more than one of the fields
    - The indexes include record pointers (rather than block pointers)
    - Each index can be used to retrieve the set of record pointers that satisfy the individual condition.
    - The intersection of these sets of records pointers gives the record pointers that satisfy the conjunctive condition.

$\delta_{superssn='123456789' \text{ and } Dno = 1}$ (Employee)



Result = $\{s_1, s_2, \ldots, s_i\} \cap \{t_1, t_2, \ldots, t_j\}$

All employees in Dno =1.

All employees supervised by 1234….

- **Basic algorithms**
  - JOIN operation (two-way join)

$$R \underset{A=B}{\bowtie} S$$

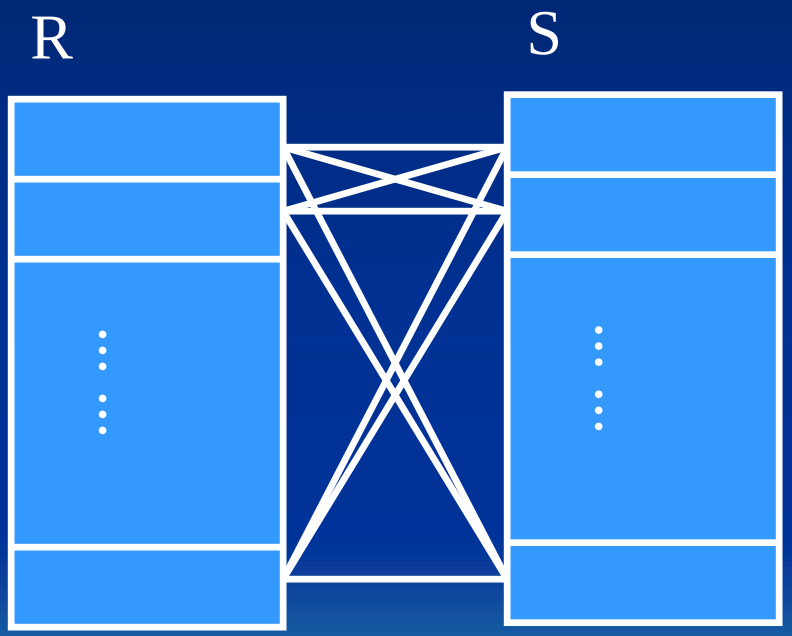
Example:

(OP6): EMPLOYEE $\underset{\text{DNO=DNUMBER}}{\bowtie}$ DEPARTMENT

(OP7): DEPARTMENT $\underset{\text{MGRSSN=SSN}}{\bowtie}$ EMPLOYEE

- **Basic algorithms**
  - Methods for implementing JOINs

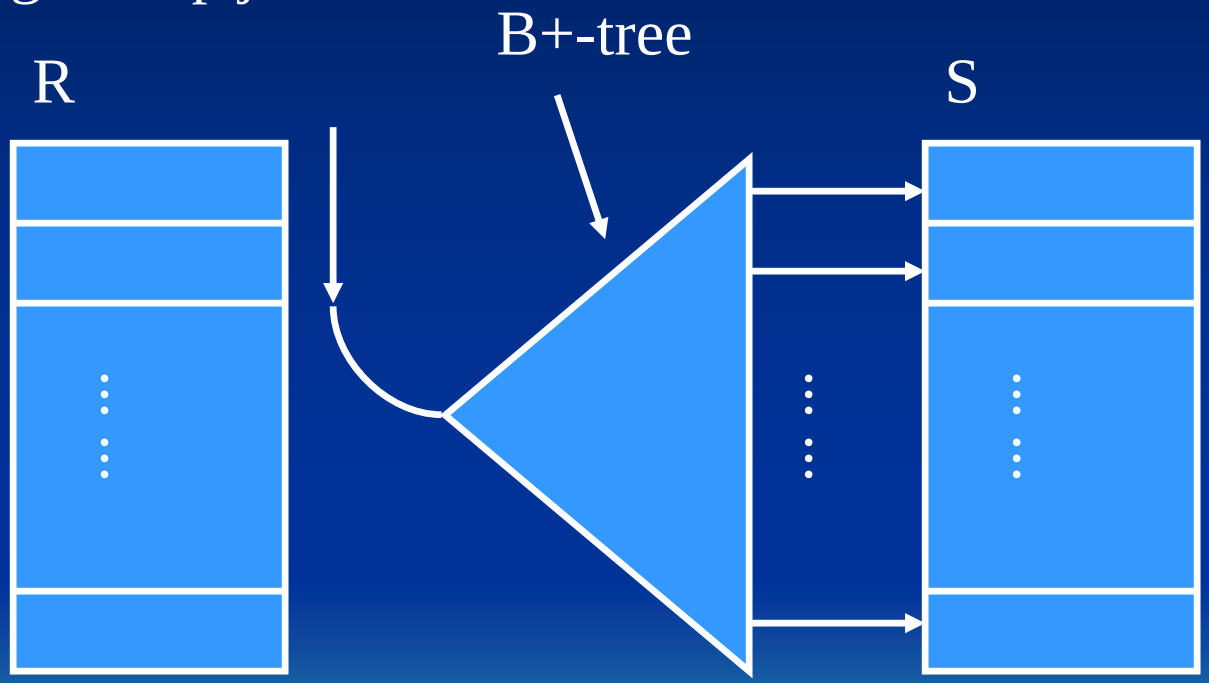
    Nested-loop join:



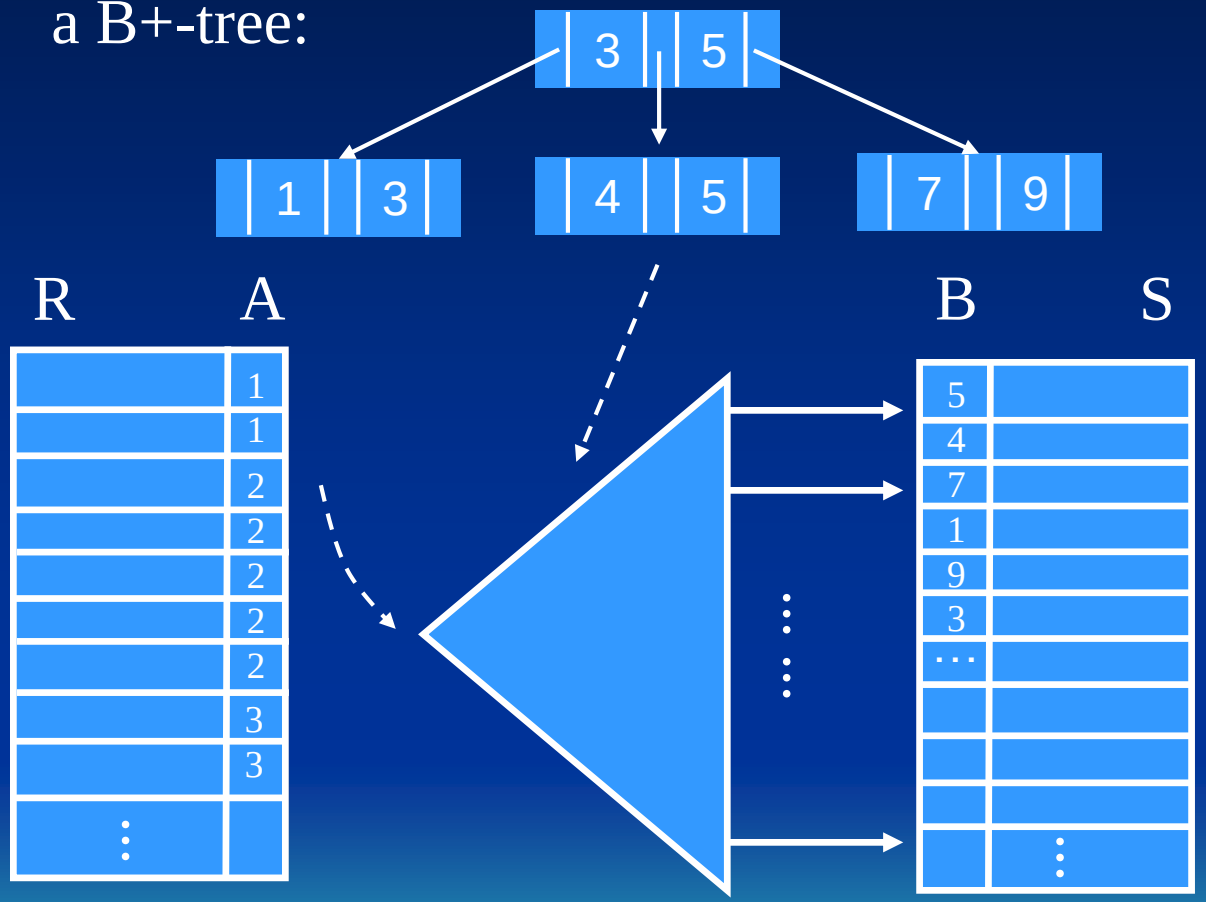
Running time: O(*nm*)

- **Basic algorithms**
  - Methods for implementing JOINs

    Single-loop join:

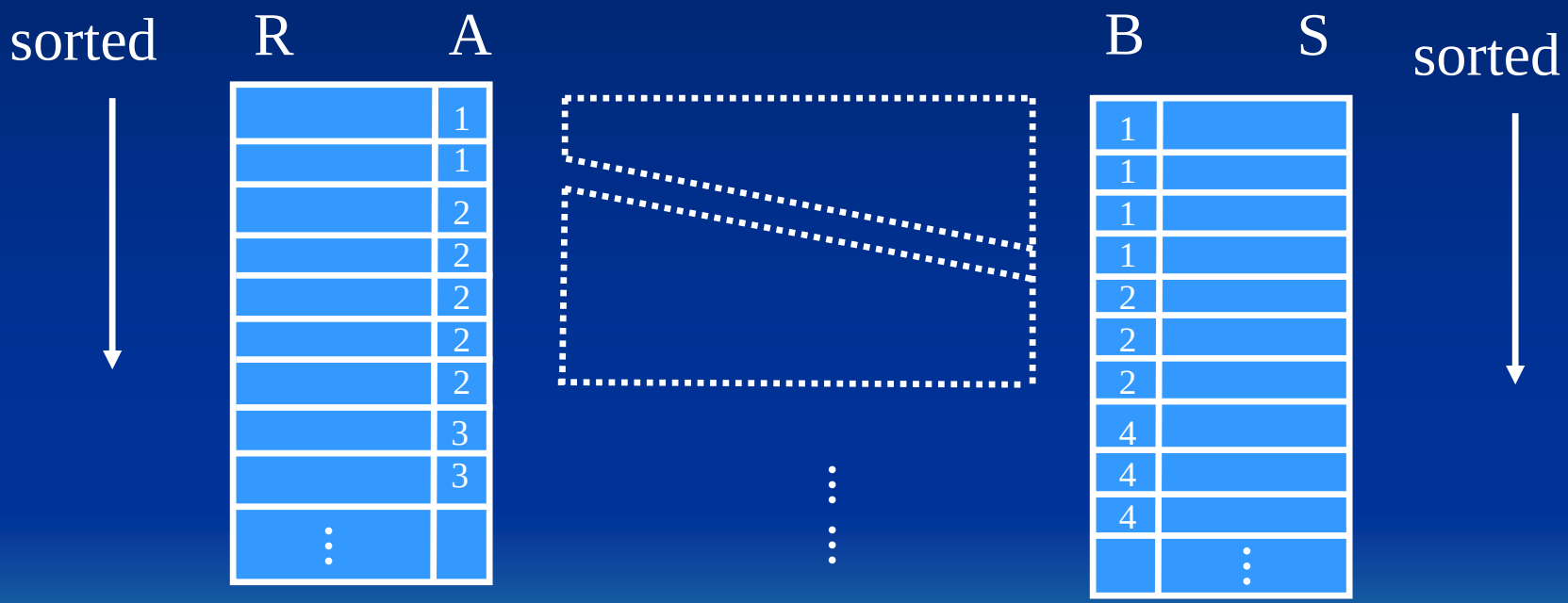


a B+-tree:



- **Basic algorithms**
  - Methods for implementing JOINs

    Sort-merge join:



Sort-merge join:

sort the tuples in R on attribute A /*assume R has n tuples*/

sort the tuples in S on attribute B /*assume S has m tuples*/

set $i \leftarrow 1$; $j \leftarrow 1$;

while ($i \leq$ n) and ($j \leq$ m)

do {if R(i)[A] > S(j)[B] then set $j \leftarrow j$ +1

else R(i)[A] < S(j)[B] then set $i \leftarrow i$ +1

else {/* R(i)[A] = S(j)[B], so we output a matched tuple*/

output the combination tuple < R(i), S(j)> to T;

/*output other tuples that matches R(i), if any*/

Sort-merge join:

set $l \leftarrow j + 1$;
while ($l \leq$ m) and (R(i)[A] = S(l)[B])
do {output the combined tuple < R(i), S(j)> to T;
set $l \leftarrow l + 1$;}
/*output other tuples that matches S(j), if any*/
set $k \leftarrow i + 1$;
while ($k \leq$ n) and (R(k)[A] = S(j)[B])
do {output the combined tuple < R(k), S(j)> to T;
set $k \leftarrow k + 1$;
}
set $i \leftarrow k$, $j \leftarrow l$;}
}

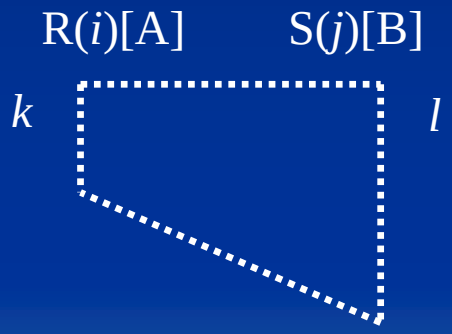
This part is wrong!
(in 3rd edition)

Sort-merge join:

set $l \leftarrow j$ +1;
while ($l \leq$ m) and (R(i)[A] = S(l)[B])
do {output the combined tuple < R(i), S(l)> to T;
    set $l \leftarrow l$ +1;}
/*output other tuples that matches S(j), if any*/
set $k \leftarrow i$ +1;
while ($k \leq$ n) and (R(k)[A] = S(j)[B])
do {output the combined tuple <R(k), S(j)> to T;
    set $k \leftarrow k$ +1;
  }
set $i \leftarrow i + 1$, $j \leftarrow j + 1$;}
}

This part is correct!
(in 4th edition)

set $i \leftarrow 1$; $j \leftarrow 1$;
while ($i \leq$ n) and ($j \leq$ m)
do {if R(i)[A] > S(j)[B] then set $j \leftarrow j$ +1
else R(i)[A] < S(j)[B] then set $i \leftarrow i$ +1
else {/* R(i)[A] = S(j)[B], so we output a matched tuple*/
set $k \leftarrow i$;
while ($k \leq$ n) and (R(k)[A] = S(j)[B])
do {set $l \leftarrow j$;
while ($l \leq$ m) and (R(k)[A] = S(l)[B])
do {output < R(k), S(l)> to T; $l \leftarrow l$ + 1; }
set $k \leftarrow k$ +1;}
set $i \leftarrow k$, $j \leftarrow l$;
}
}



discussed in classes

- **Basic algorithms**
  - PROJECT operation

$$\pi_{<\text{Attribute list}>}(R)$$

Example:

$$\pi_{\text{FNAME, LNAME, SEX}}(\text{EMPLOYEE})$$

Algorithm:

1. Construct a table according to <Attribute list> of R.
2. Do the duplication elimination.

- **Basic algorithms**

  - PROJECT operation

For each tuple t in R, create a tuple t[<Attribute list>] in T′

/*T′ contains the projection result before duplication elimination*/

if <Attribute list> includes a key of R then T ← T′

else { sort the tuples in T′;

set $i \leftarrow 1$, $j \leftarrow 2$;

while $i \leq n$

do { output the tuple T′[i] to T;

while T′[i] = T′[j] and $j \leq n$ do $j \leftarrow j + 1$;

$i \leftarrow j$; $j \leftarrow j + 1$;

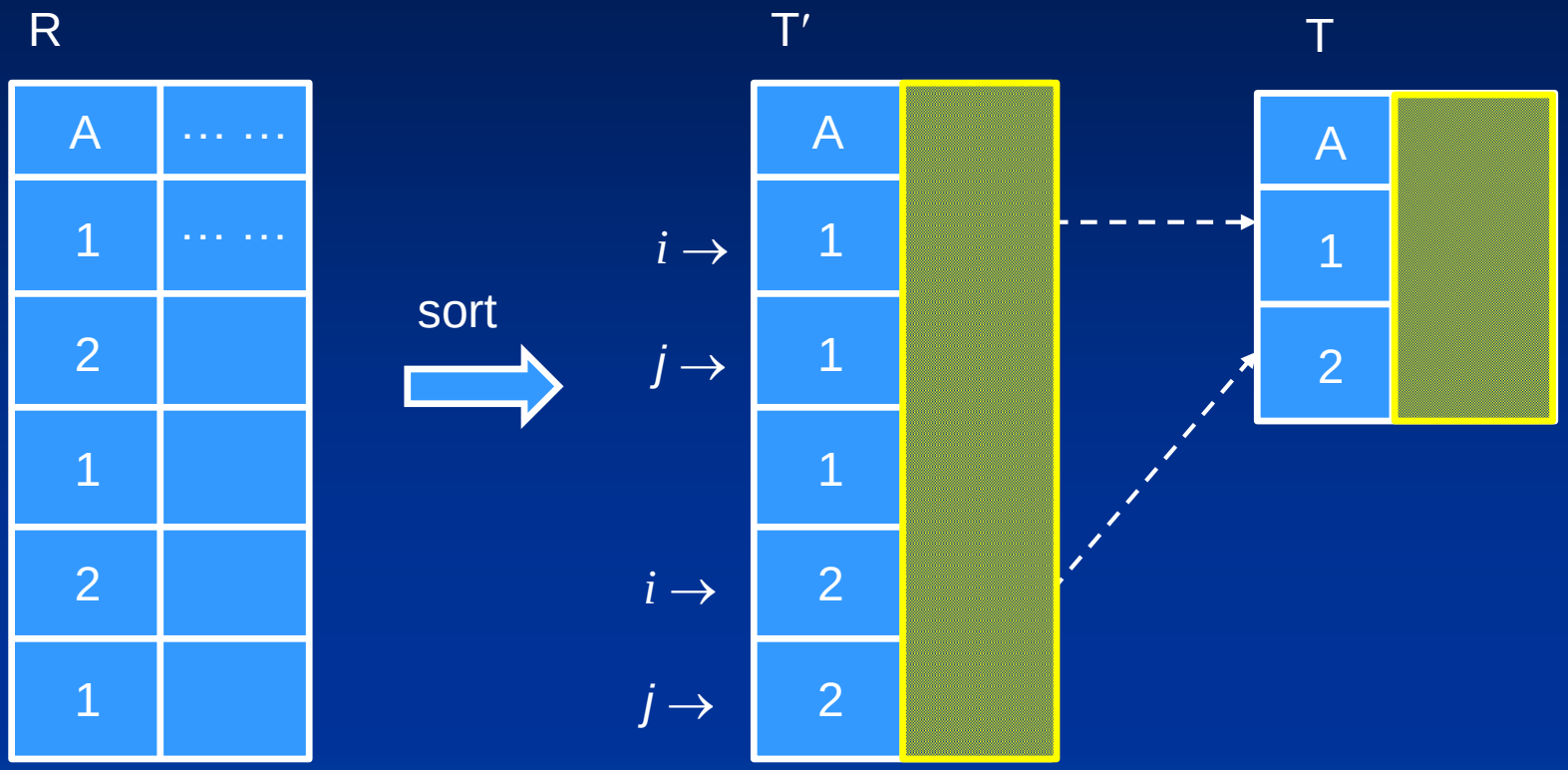
}

}

table construction

duplication elimination

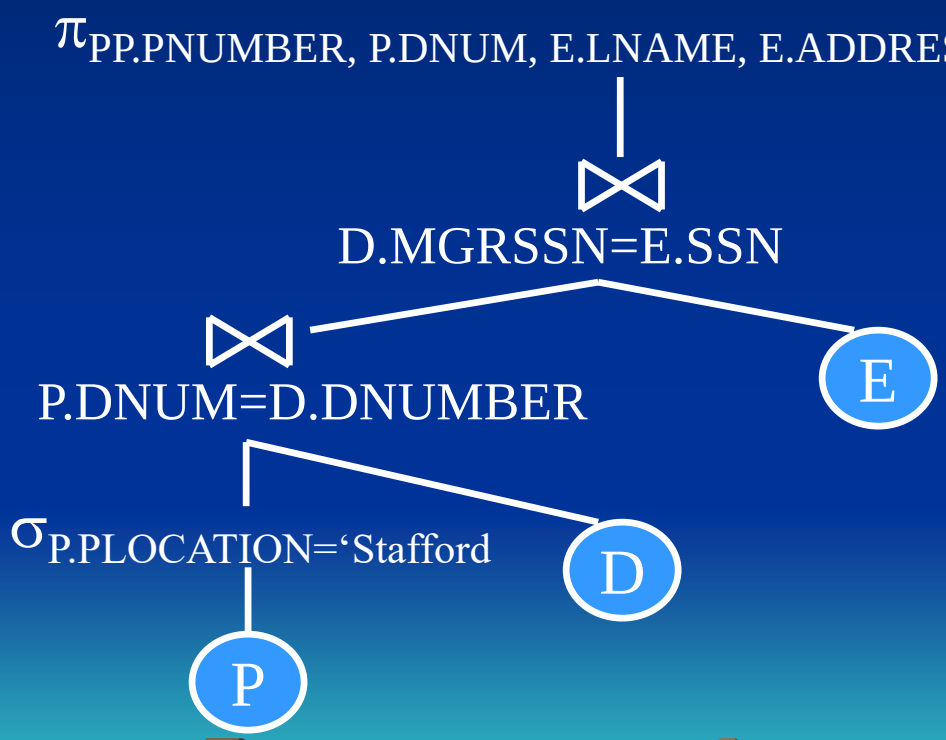
$\pi_A(R)$:



- **Heuristics for query optimization**
  - Query trees and query graphs
  - Heuristic optimization of query trees
  - General transformation rules for relational algebra operations
  - Outline of a heuristic algebraic optimization algorithm

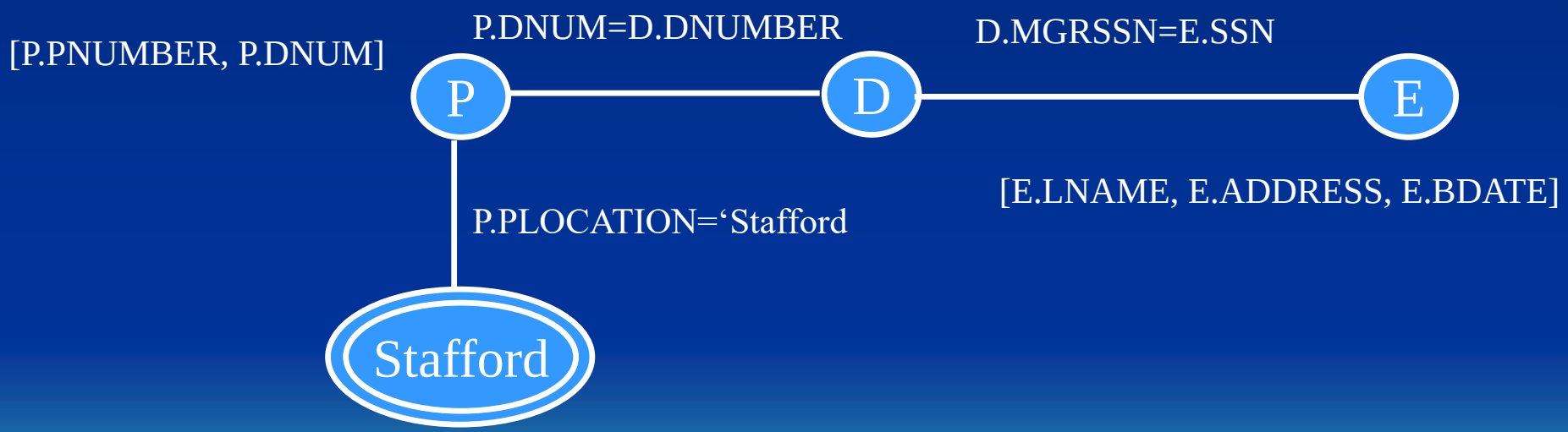
- Query trees

$\pi_{PNUMBER, DNUM, LNAME, ADDRESS, BDATE}((\sigma_{PLOCATION='Stafford}(PROJECT)) \bowtie_{DNUM=DNUMBER} (DEPARTMENT)) \bowtie_{MGRSSN=SSN} (EMPLOYEE))$



- Query graph

$$\pi_{\text{PNUMBER, DNUM, LNAME, ADDRESS, BDATE}}((\sigma_{\text{PLOCATION='Stafford}}(\text{PROJECT})) \bowtie_{\text{DNUM=DNUMBER}} (\text{DEPARTMENT})) \bowtie_{\text{MGRSSN=SSN}} (\text{EMPLOYEE}))$$

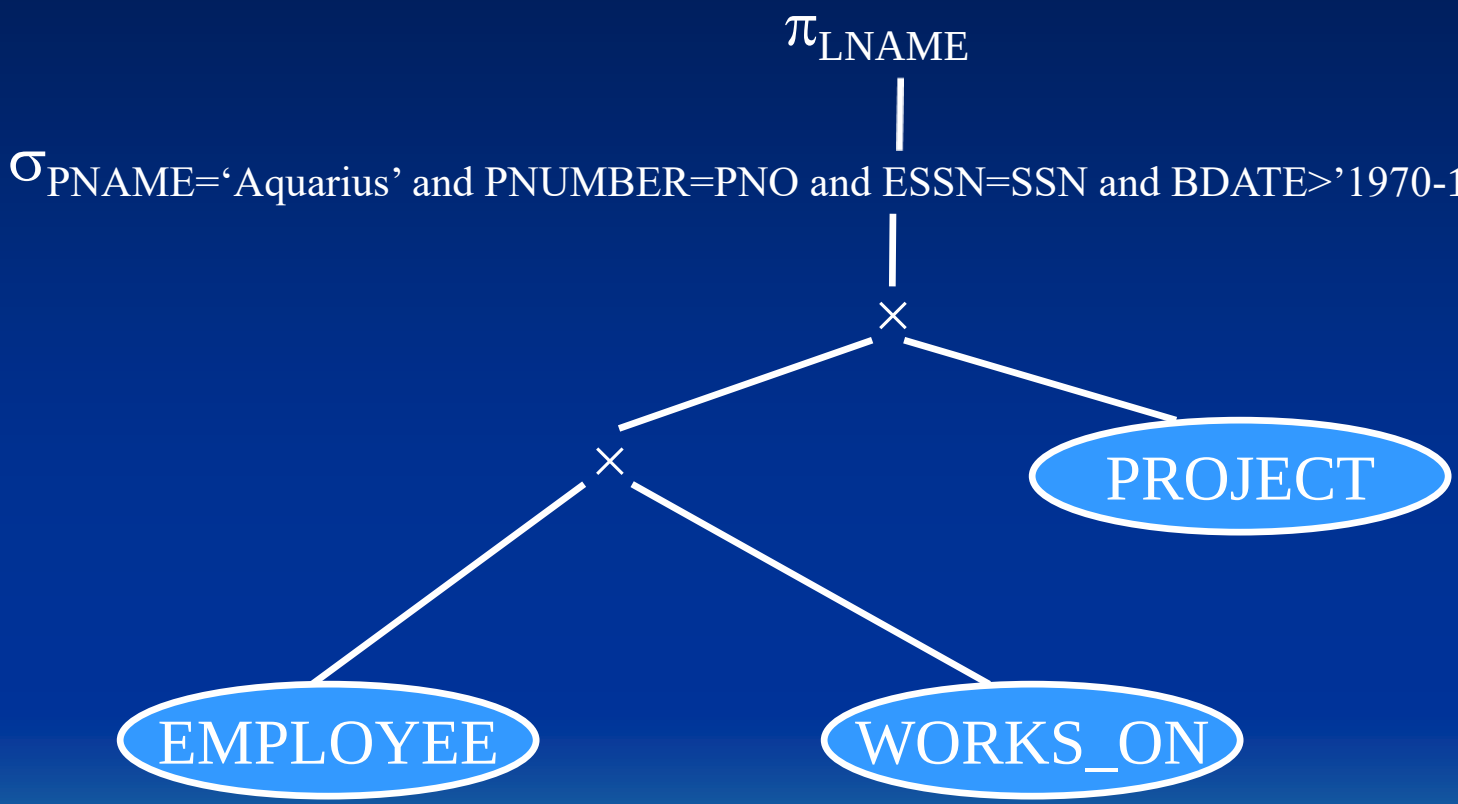


- **Heuristic optimization of query trees**
  - Generate an initial query tree for a query
  - Using the rules for equivalence to transform the query tree in such a way that a transformed tree is more efficient than the previous one.

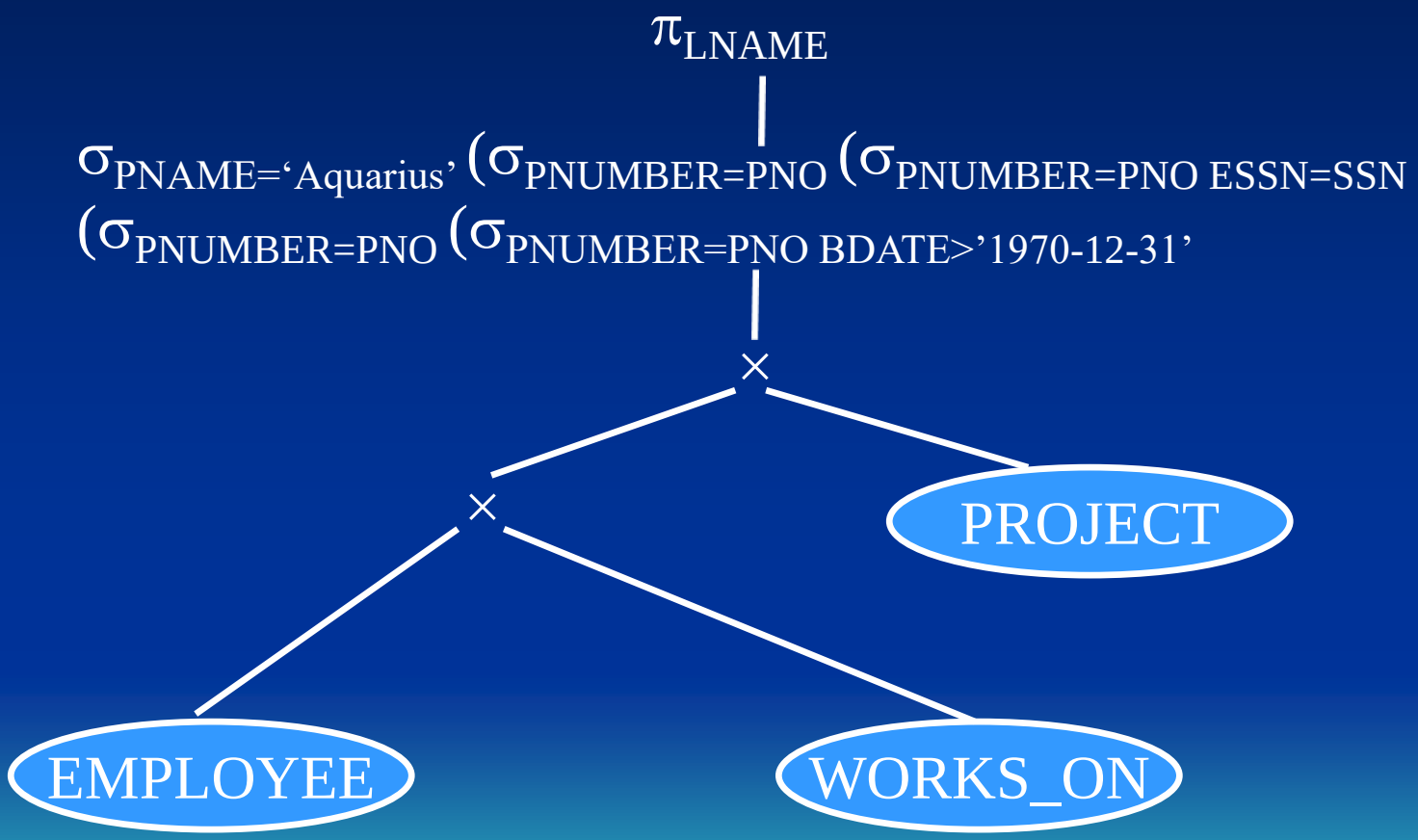
Example:

```
Q:  SELECT  LNAME
    FROM    EMPLOYEE, WORKS_ON, PROJECT
    WHERE   PNAME='Aquarius' and PNUMBER=PNO
                and   ESSN =SSN
                and   BDATE>'1970-12-31'
```

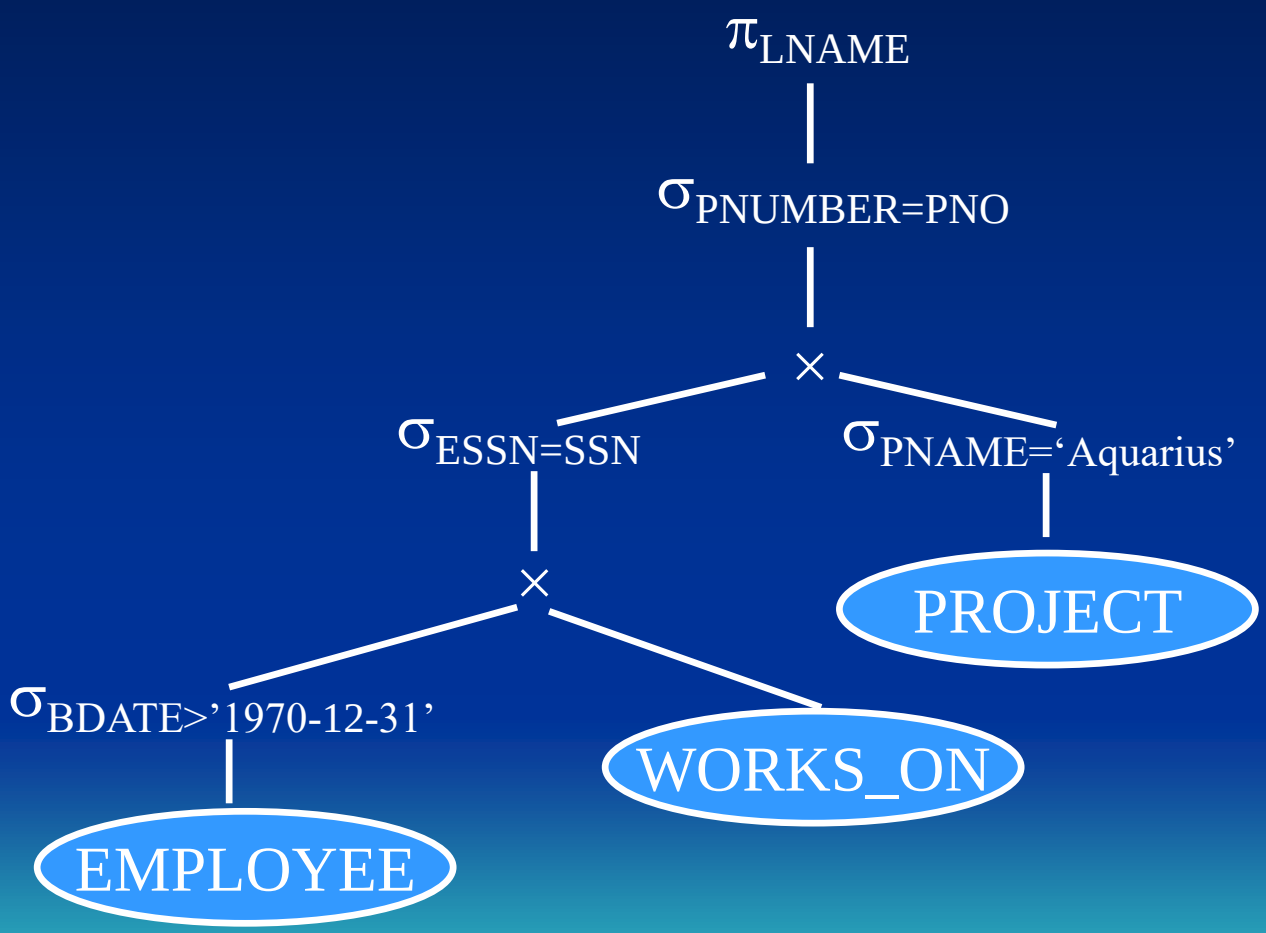
Initial query tree:



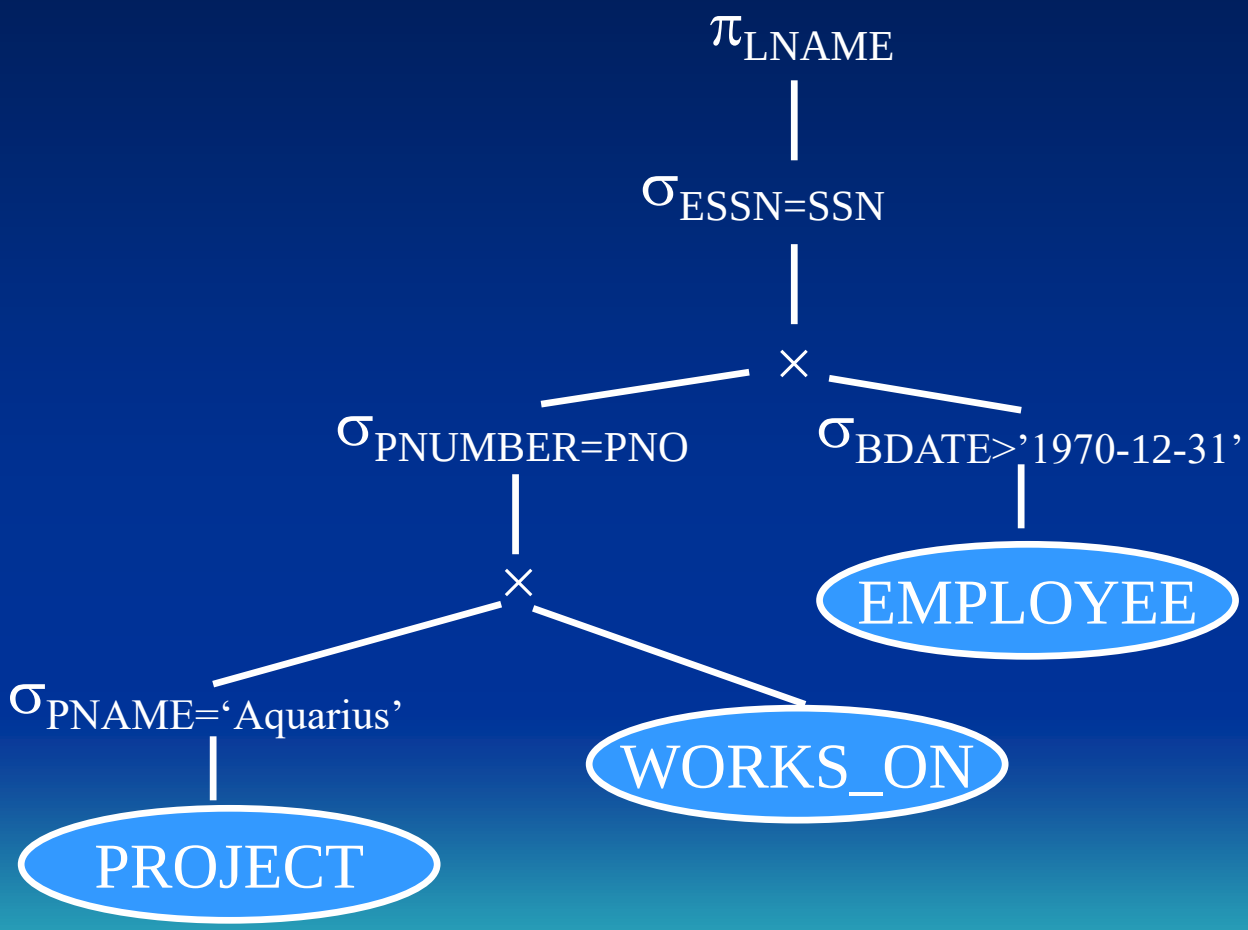
First transformation:



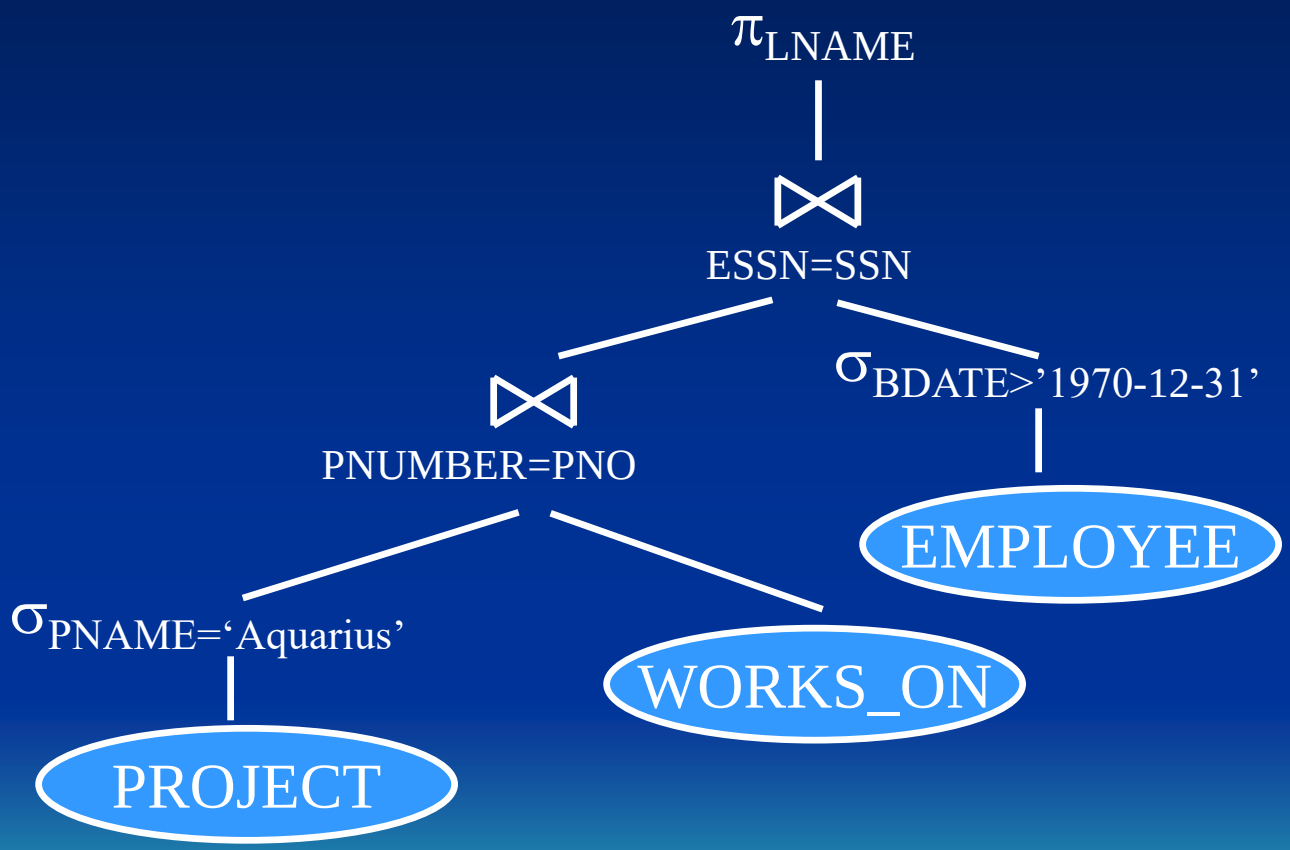
Second transformation:



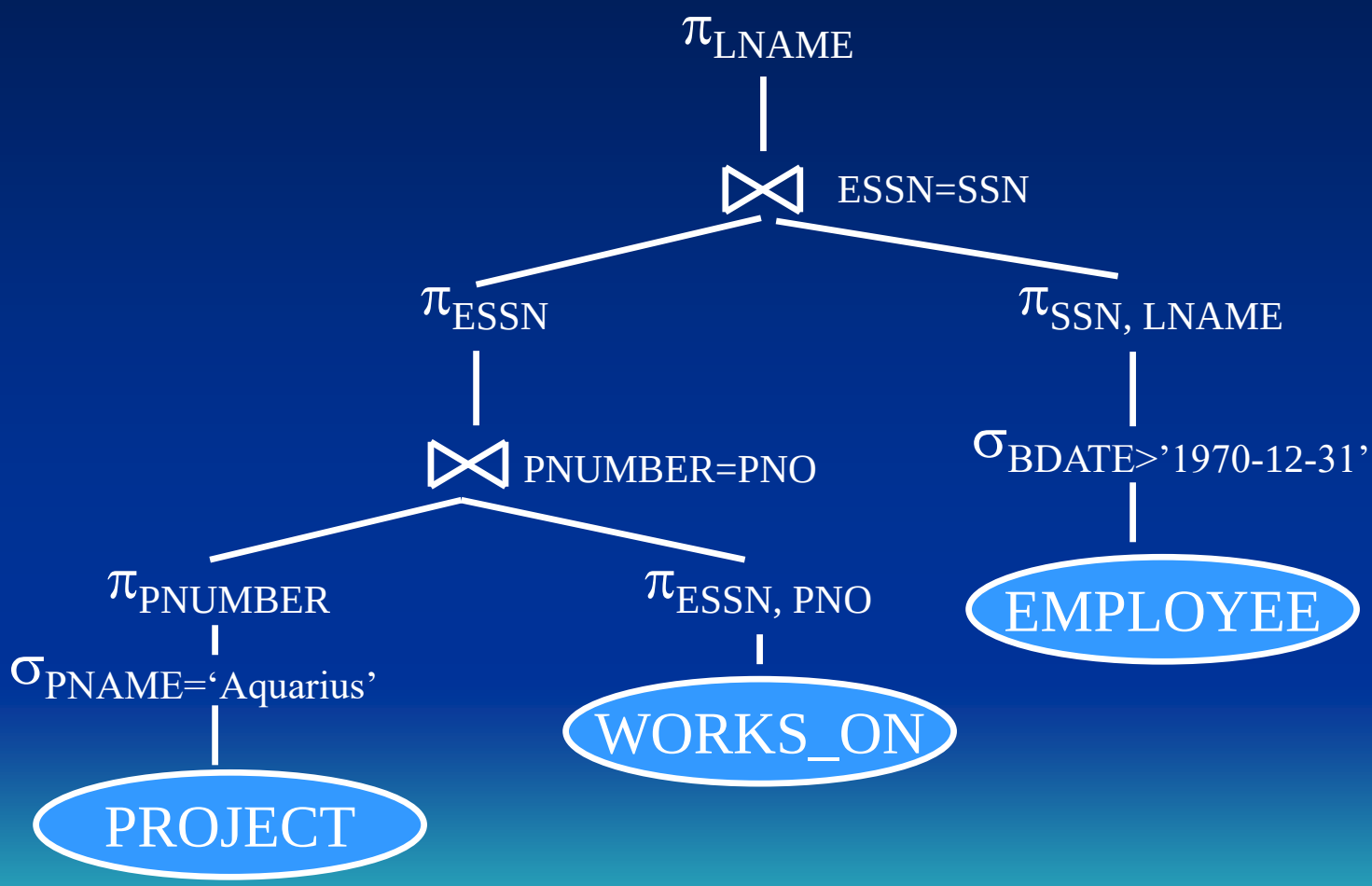
Third transformation:



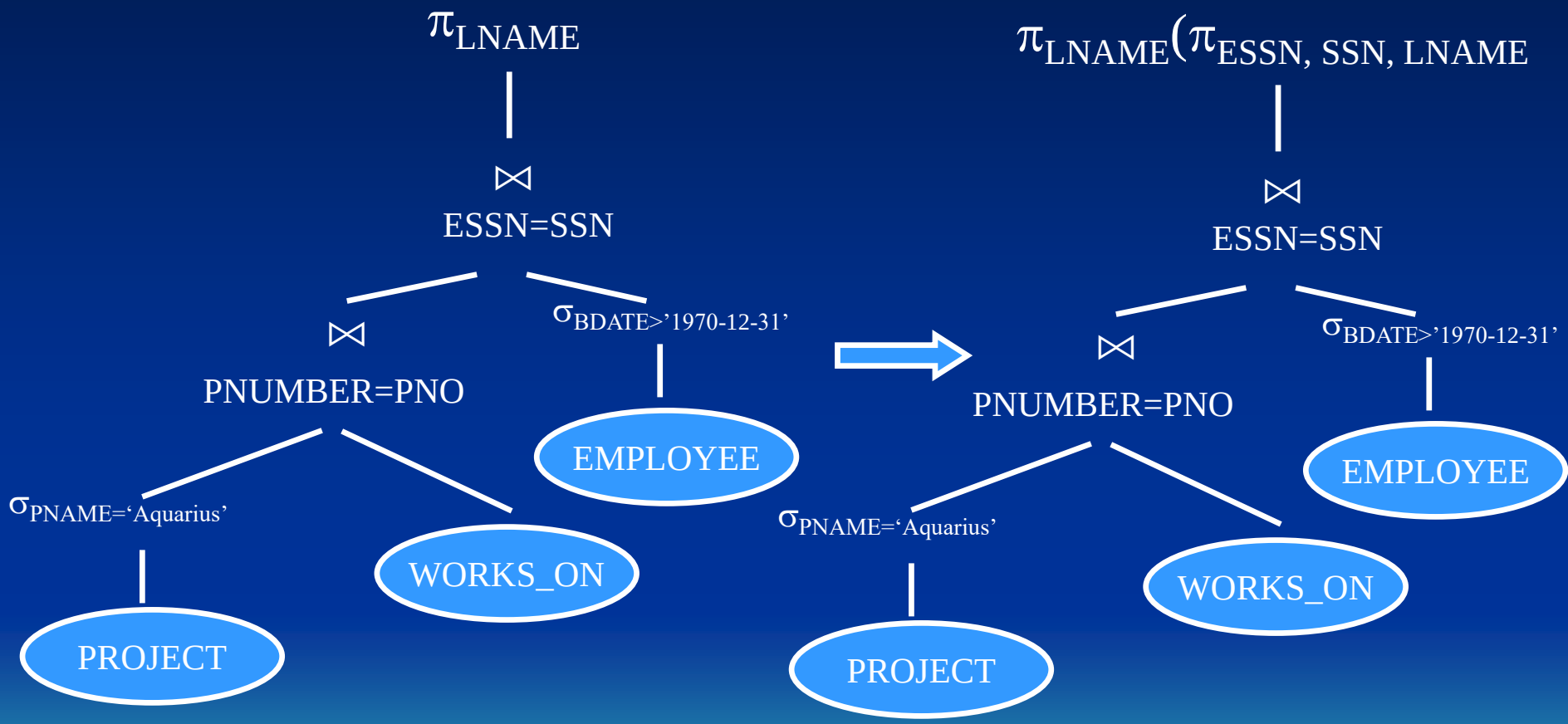
Fourth transformation:



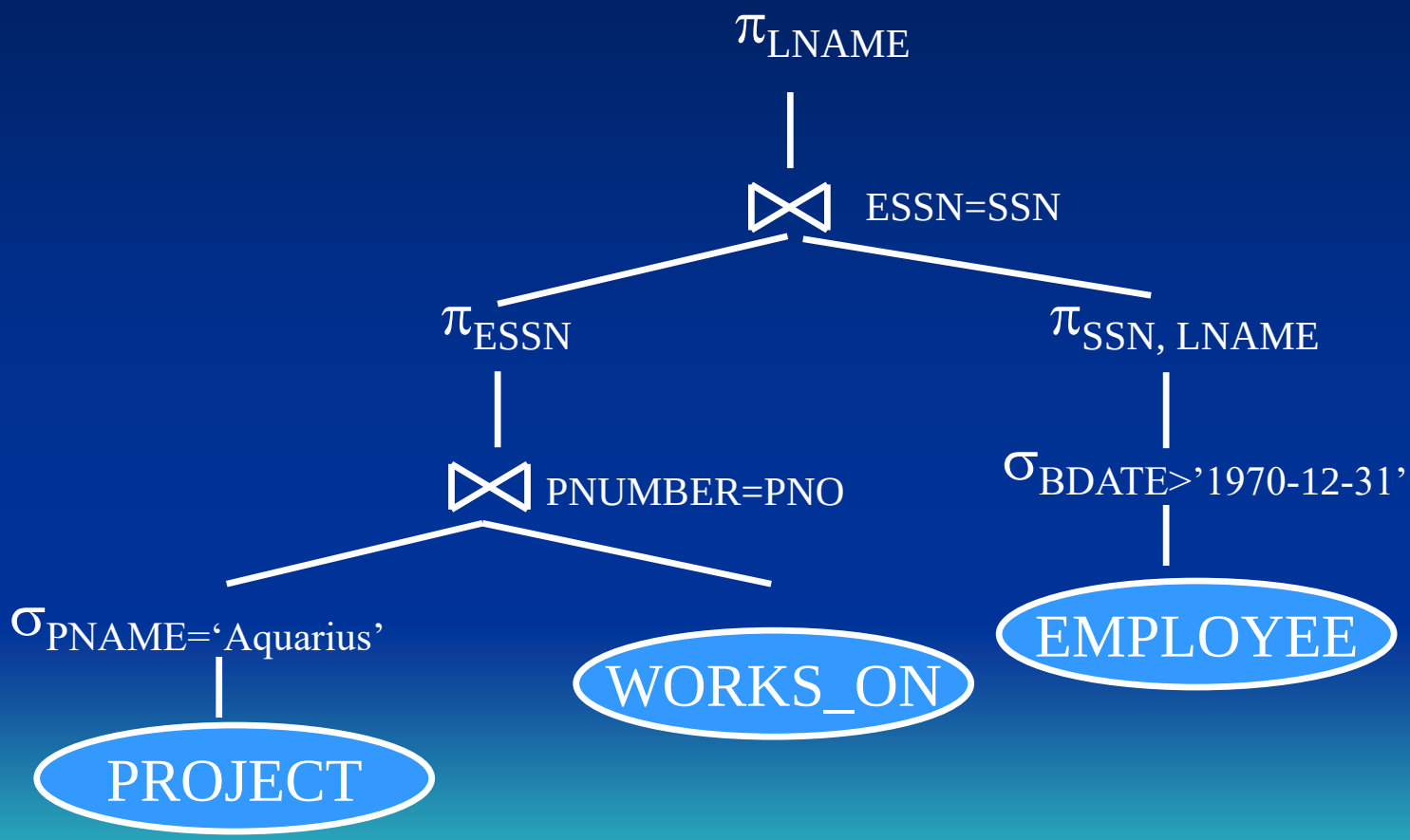
Fifth transformation:



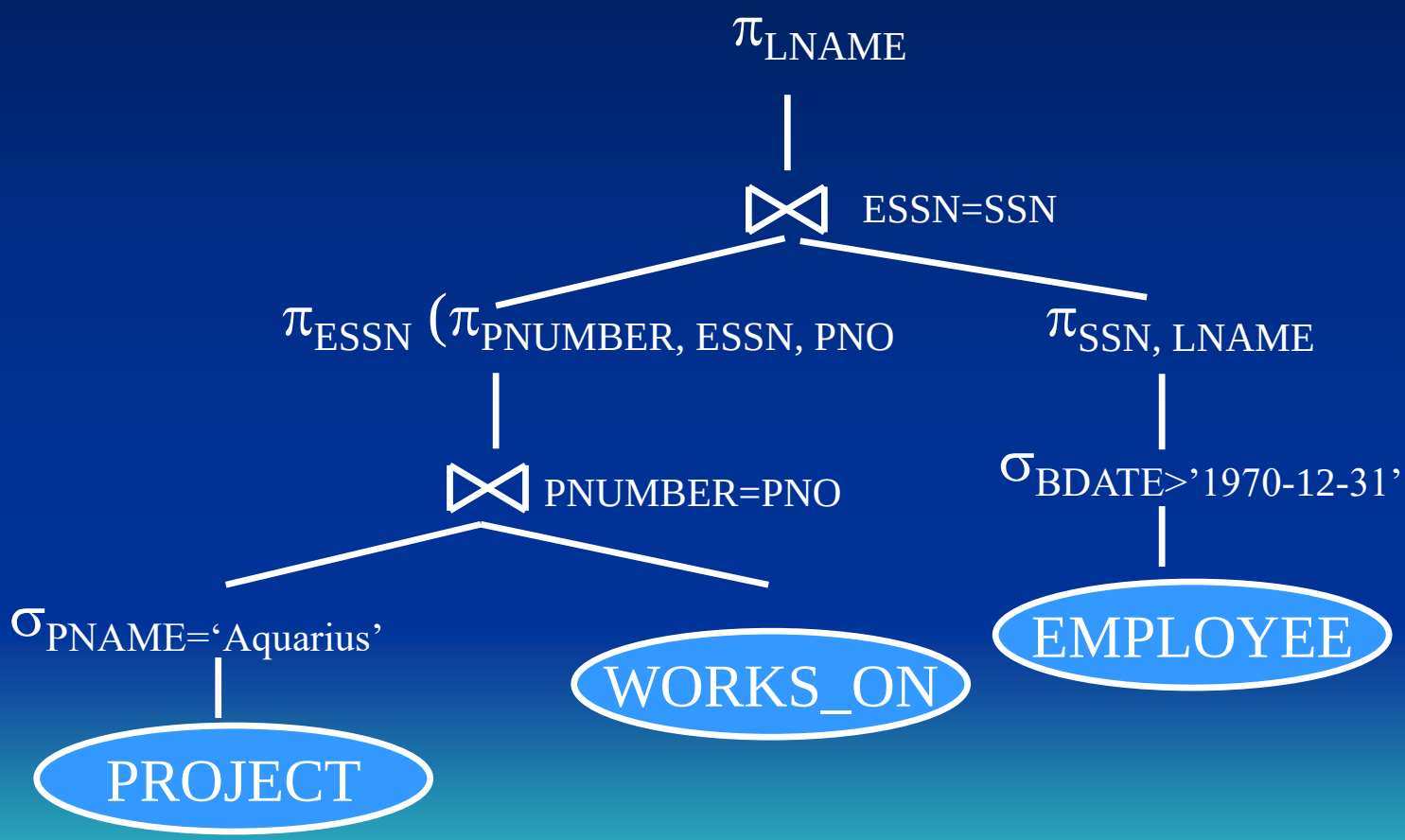
More on the fifth transformation:



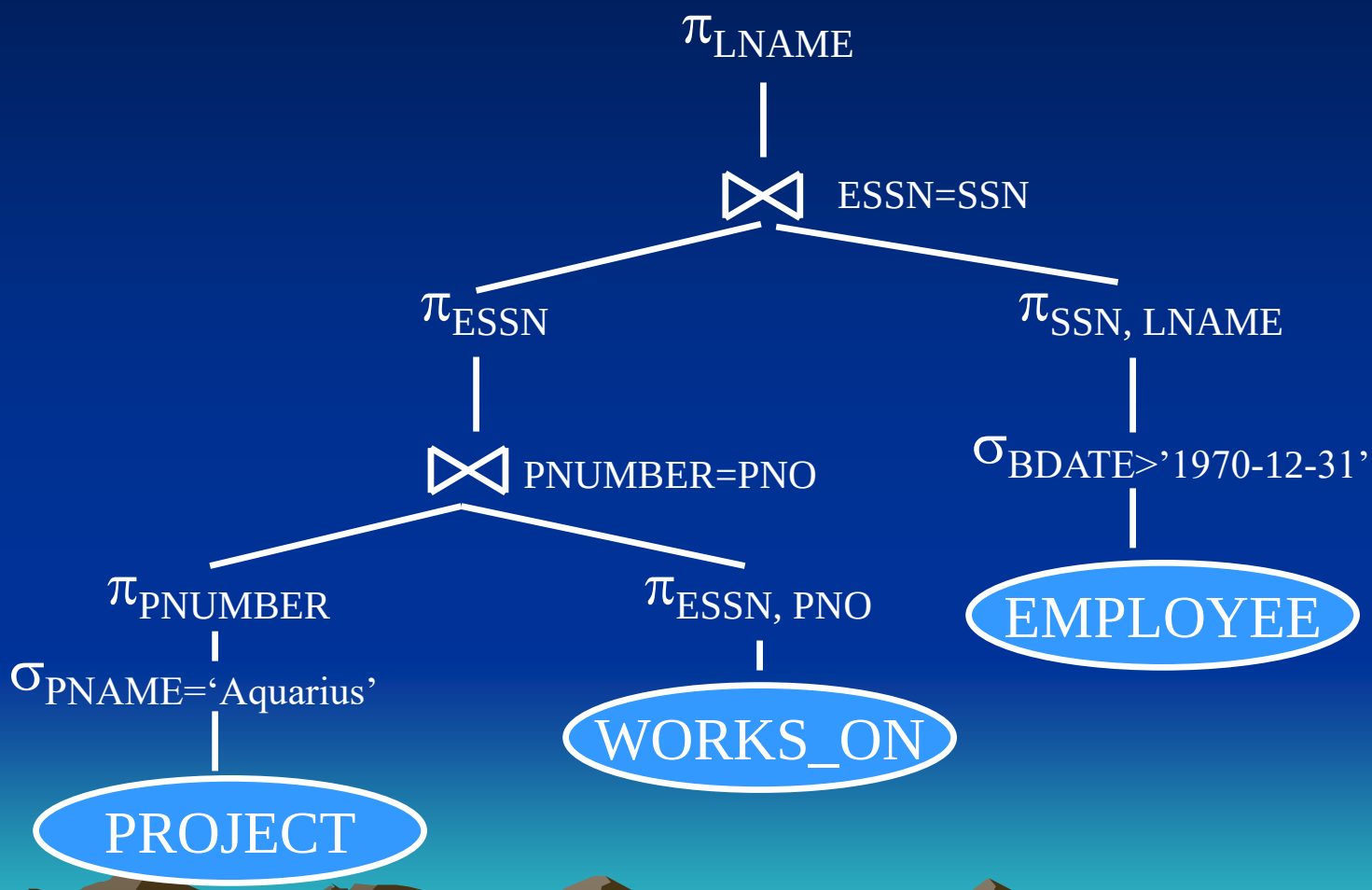
More on the fifth transformation:



More on the fifth transformation:



More on the fifth transformation:



- **General transformation rules for relational algebra operations** (altogether 12 rules)

1. Cascade of σ: A conjunctive selection condition can be broken into a cascade (i.e., a sequence) of individual σ operations:

$$\sigma_{c1 \text{ and } c2 \text{ and } \ldots \text{ and } cn}(R) \equiv \sigma_{c1}(\sigma_{c2}(\ldots(\sigma_{cn}(R))\ldots))$$

$$\sigma_{\text{ssn} = \text{‘123456789’and salary} > 30{,}000}(\text{Employee})$$

$$\sigma_{\text{ssn} = \text{‘123456789’}}\ (\sigma_{\text{salary} > 30{,}000}\ (\text{Employee}))$$

2. Commutativity of σ: The σ operation is commutative:

$$\sigma_{c1}(\sigma_{c2}(R)) \equiv \sigma_{c2}(\sigma_{c1}(R))$$

$$\sigma_{\text{ssn = '123456789'}}(\sigma_{\text{salary > 30,000}}(\text{Employee}))$$

$$\sigma_{\text{salary >30,000}}(\sigma_{\text{ssn = '123456789'}}(\text{Employee}))$$

3. Cascade of π: In a cascade (sequence) of π operations, all but the last one can be ignored:

$$\pi_{\text{list1}}(\pi_{\text{list2}}(\ldots(\pi_{\text{list}n}(R))\ldots)) \equiv \pi_{\text{list1}}(R),$$

where $\text{list}_1 \subseteq \text{list}_2 \subseteq \ldots \subseteq \text{list}_n$.

$$\pi_{\text{LNAME}}(\pi_{\text{LNAME, FNAME}}(\text{Employee}))$$

$$\pi_{\text{LNAME}}(\text{Employee})$$

- **General transformation rules for relational algebra operations** (altogether 12 rules)
  4. Commuting σ with π: If the selection condition c involves only those attributes A1, …, A*n* in the projection list, the two operations can be commuted:

$$\pi_{A1, \ldots, An}(\sigma_c(R)) \equiv \sigma_c(\pi_{A1, \ldots, An}(R))$$

$$\pi_{\text{FNAME, LNAME, salary}}(\sigma_{\text{salary} > 30{,}000}(\text{Employee}))$$

$$\sigma_{\text{salary} > 30{,}000}(\pi_{\text{FNAME, LNAME, salary}}(\text{Employee}))$$

$$\pi_{\text{FNAME, LNAME}} (\sigma_{\text{LNAME = 'Green' and salary > 30,000}}(\text{Employee}))$$

?

$$\sigma_{\text{LNAME = 'Green' and salary > 30,000}} (\pi_{\text{FNAME, LNAME}} (\text{Employee}))$$

5. Commutativity of $\bowtie$ (and $\times$): The $\bowtie$ operation is commutative, as is the $\times$ operation:

$$R \bowtie_c S \equiv S \bowtie_c R$$

$$R \times S \equiv S \times R$$

- **General transformation rules for relational algebra operations** (altogether 12 rules)

  6. Commuting σ with $\bowtie$ (or $\times$): If all the attributes in the selection condition c involves only the attributes of one of the relations being joined - say, R - the two operations can be commuted as follows:

$$\sigma_c(R \bowtie S) \equiv \sigma_c(R) \bowtie S$$

If c is of the form: c1 and c2, and c1 involves only the attributes of R and c2 involves only the attributes of S, then:

$$\sigma_c(R \bowtie S) \equiv (\sigma_{c1}(R)) \bowtie (\sigma_{c2}(S))$$

$$\sigma_{LNAME = \text{'Greenwich'}}(Employee \underset{ssn = essn}{\bowtie} Works\_on)$$

$$\sigma_{LNAME = \text{'Greenwich'}}(Employee) \underset{ssn = essn}{\bowtie} Works\_on$$

$$\sigma_{LNAME = \text{'Greenwich' and } Pno = 1}(Employee \underset{ssn = essn}{\bowtie} Works\_on)$$

$$\sigma_{LNAME = \text{'Greenwich'}}(Employee) \underset{ssn = essn}{\bowtie} \sigma_{Pno = 1}(Works\_on)$$

- **General transformation rules for relational algebra operations** (altogether 12 rules)
  7. Commuting π with $\bowtie$ (or $\times$): Suppose that the projection list is L = {A1, …, A*n*, B1, …, B*m*}, where A1, …, A*n* in R and B1, …, B*m* in S. If the attributes in the join condition c are involved in L, we have

$$\pi_{L}(R \bowtie_{C} S) \equiv (\pi_{A1, \ldots, An}(R)) \bowtie_{C} (\pi_{B1, \ldots, Bm}(S))$$

$$\pi_{ssn,\ essn,\ hours}(Employee \underset{ssn = essn}{\bowtie} Works\_on)$$

$$\pi_{ssn}(Employee) \underset{ssn = essn}{\bowtie} \pi_{essn,\ hours}(Works\_on)$$

8. Commutativity of set operations: The set operation "$\cup$" and "$\cap$" are commutative, but "-" is not.

9. Associativity of $\bowtie$, $\times$, $\cup$ and $\cap$: These four operations are individually associative; i.e., if θ stands for any one of these four operations, we have:

$$(R \ \theta \ S) \ \theta \ T \equiv R \ \theta \ (S \ \theta \ T)$$

$R \cup S \equiv S \cup R$
$R \cap S \equiv S \cap R$
But $R - S \neq S - R$

R and S have the same structure.

$(R \bowtie S) \bowtie T \equiv R \bowtie (S \bowtie T)$
$(R \times S) \times T \equiv R \times (S \times T)$
$(R \cup S) \cup T \equiv R \cup (S \cup T)$
$(R \cap S) \cap T \equiv R \cap (S \cap T)$

$R \bowtie S \bowtie T \equiv R \bowtie S \bowtie T$

- **General transformation rules for relational algebra operations** (altogether 12 rules)

10. Commuting σ with set operations: The σ operation commutes with "∪", "∩" and "-". If θ stands for any one of these three operations, we have:

$$\sigma_c(R \ \theta \ S) \equiv \sigma_c(R) \ \theta \ \sigma_c(S)$$

$$\sigma_c(R \cup S) \equiv \sigma_c(R) \cup \sigma_c(S)$$
$$\sigma_c(R \cap S) \equiv \sigma_c(R) \cap \sigma_c(S)$$
$$\sigma_c(R - S) \equiv \sigma_c(R) - \sigma_c(S)$$

11. The π operation commutes with $\cup$:

$$\pi_L(R \cup S) \equiv (\pi_L(R)) \cup (\pi_L(S))$$

$$\pi_L(R \cap S) \neq (\pi_L(R)) \cap (\pi_L(S))?$$

R

A	B
2	3
1	3

S

A	B
1	3
2	1

$\pi_A(R \cap S) =$

A
1

$\pi_A(R) \cap \pi_A(S) =$

A
1
2

$$\pi_L(R - S) \neq \pi_L(R) - \pi_L(S)?$$

12. Converting a (×, σ) sequence into ⋈: If the condition c of a σ that follows a × corresponds to a join condition, convert then (×, σ) sequence into ⋈ as follows:

$$\sigma_c(R \times S) \equiv R \bowtie_c S$$

- **General transformation rules for relational algebra operations** (other rules for transformation)

DeMorgan's rule:

NOT (c1 AND c2) ≡ (NOT c1) OR (NOT c2)

NOT (c1 OR c2) ≡ (NOT c1) AND (NOT c2)

C1	C2	Not (C1 and C2)
0	0	1
0	1	1
1	0	1
1	1	0

≡

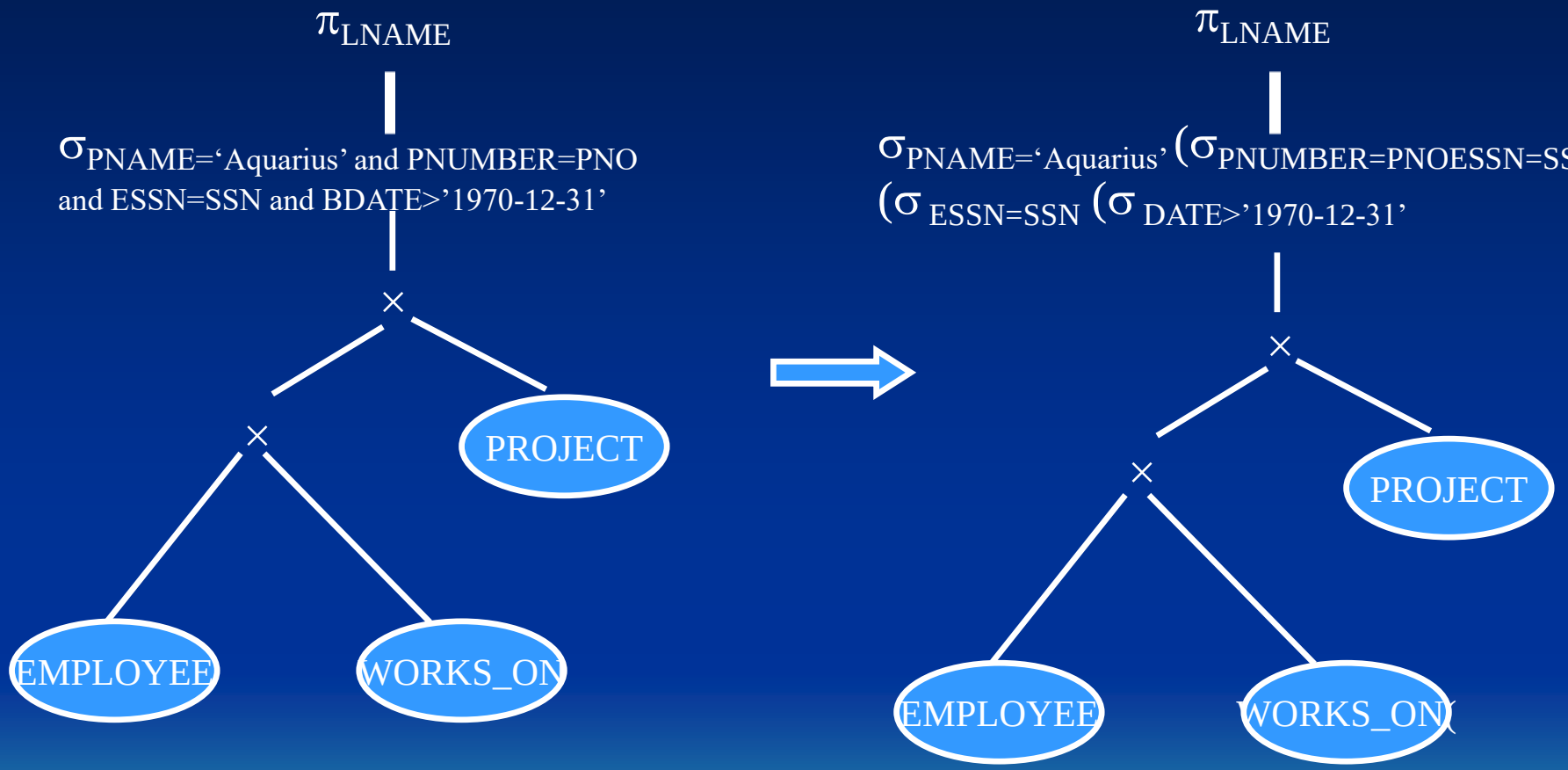
C1	C2	(Not C1) or (notC2)
0	0	1
0	1	1
1	0	1
1	1	0

- **Outline of a heuristic algebraic optimization algorithm**

1. Using Rule 1, break up any SELECT operations with conjunctive conditions into a cascade of SELECT operations. This permits a greatest degree of freedom in moving SELECT operations down different branches of the tree.

Rule 1:

$$\sigma_{c1 \text{ and } c2 \text{ and } \ldots \text{ And } cn}(R) \equiv \sigma_{c1}(\sigma_{c2}(\ldots(\sigma_{cn}(R))\ldots))$$


πLNAME
σPNAME='Aquarius' and PNUMBER=PNO and ESSN=SSN and BDATE>'1970-12-31'
×
×
PROJECT
EMPLOYEE
WORKS_ON
πLNAME
σPNAME='Aquarius' (σPNUMBER=PNOESSN=SSN (σ ESSN=SSN (σ DATE>'1970-12-31'
×
×
PROJECT
EMPLOYEE
WORKS_ON

- **Outline of a heuristic algebraic optimization algorithm**

2. Using Rules 2, 4, 6 and 10 concerning the commutativity of SELECT with other operations, move each SELECT operation as far down the query tree as is permitted by the attributes involved in the SELECT condition.

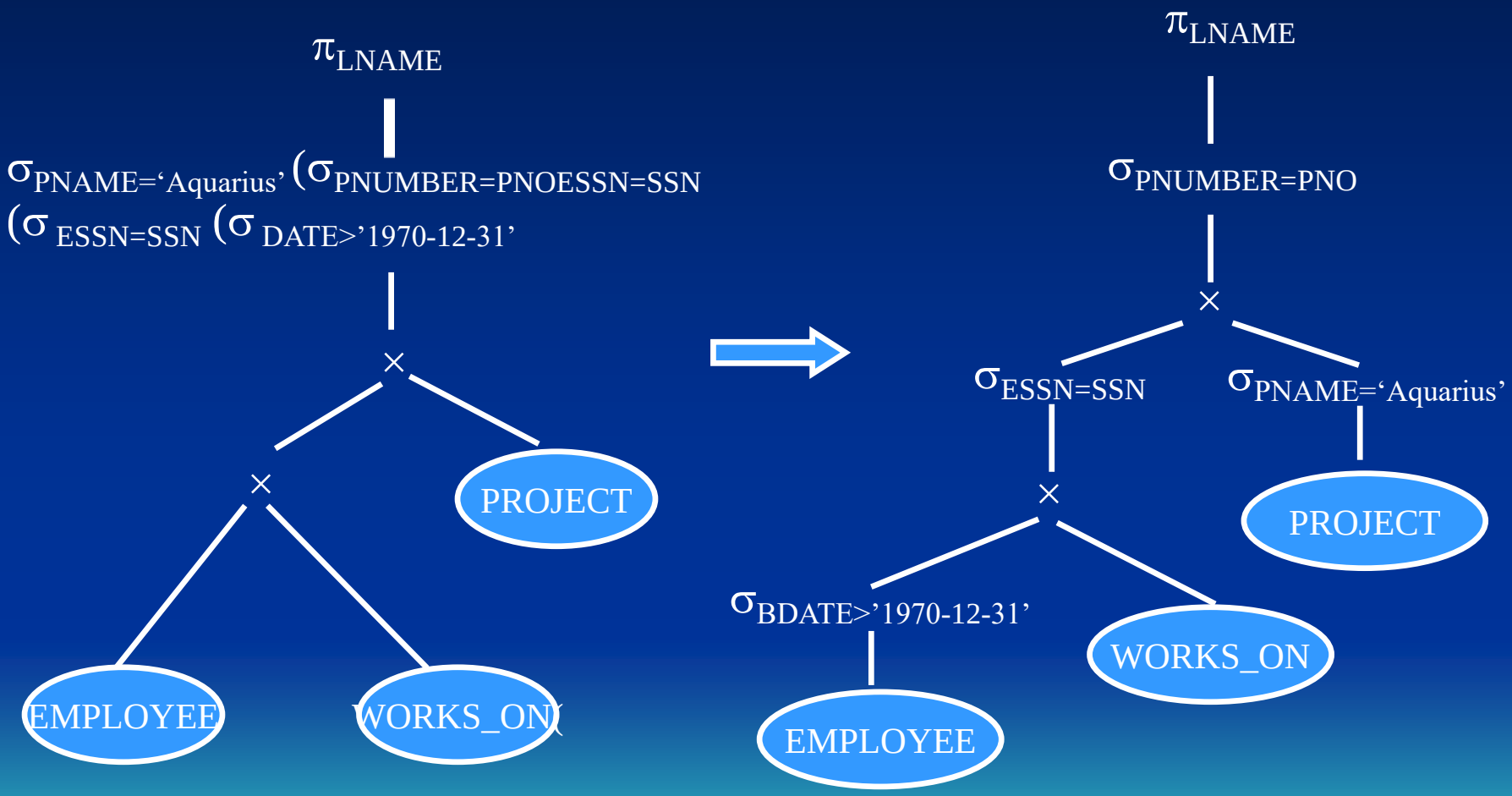
Rule 2: $\sigma_{c1}(\sigma_{c2}(R)) \equiv \sigma_{c2}(\sigma_{c1}(R))$

Rule 4: $\sigma_{c}(\pi_{A1, \ldots, An}(R)) \equiv \pi_{A1, \ldots, An}(\sigma_{c}(R))$

Rule 6: $\sigma_{c}(R \bowtie S) \equiv (\sigma_{c}(R)) \bowtie S$

$\sigma_{c}(R \bowtie S) \equiv (\sigma_{c1}(R)) \bowtie (\sigma_{c2}(S))$

Rule 10: $\sigma_{c}(R\ \theta\ S) \equiv \sigma_{c}(R)\ \theta\ \sigma_{c}(S)$ θ: $\cup$, $\cap$ or -.


πLNAME
σPNAME='Aquarius' (σPNUMBER=PNOESSN=SSN
(σ ESSN=SSN (σ DATE>'1970-12-31'
×
×
PROJECT
EMPLOYEE
WORKS_ON
πLNAME
σPNUMBER=PNO
×
σESSN=SSN
σPNAME='Aquarius'
×
PROJECT
σBDATE>'1970-12-31'
WORKS_ON
EMPLOYEE

- **Outline of a heuristic algebraic optimization algorithm**

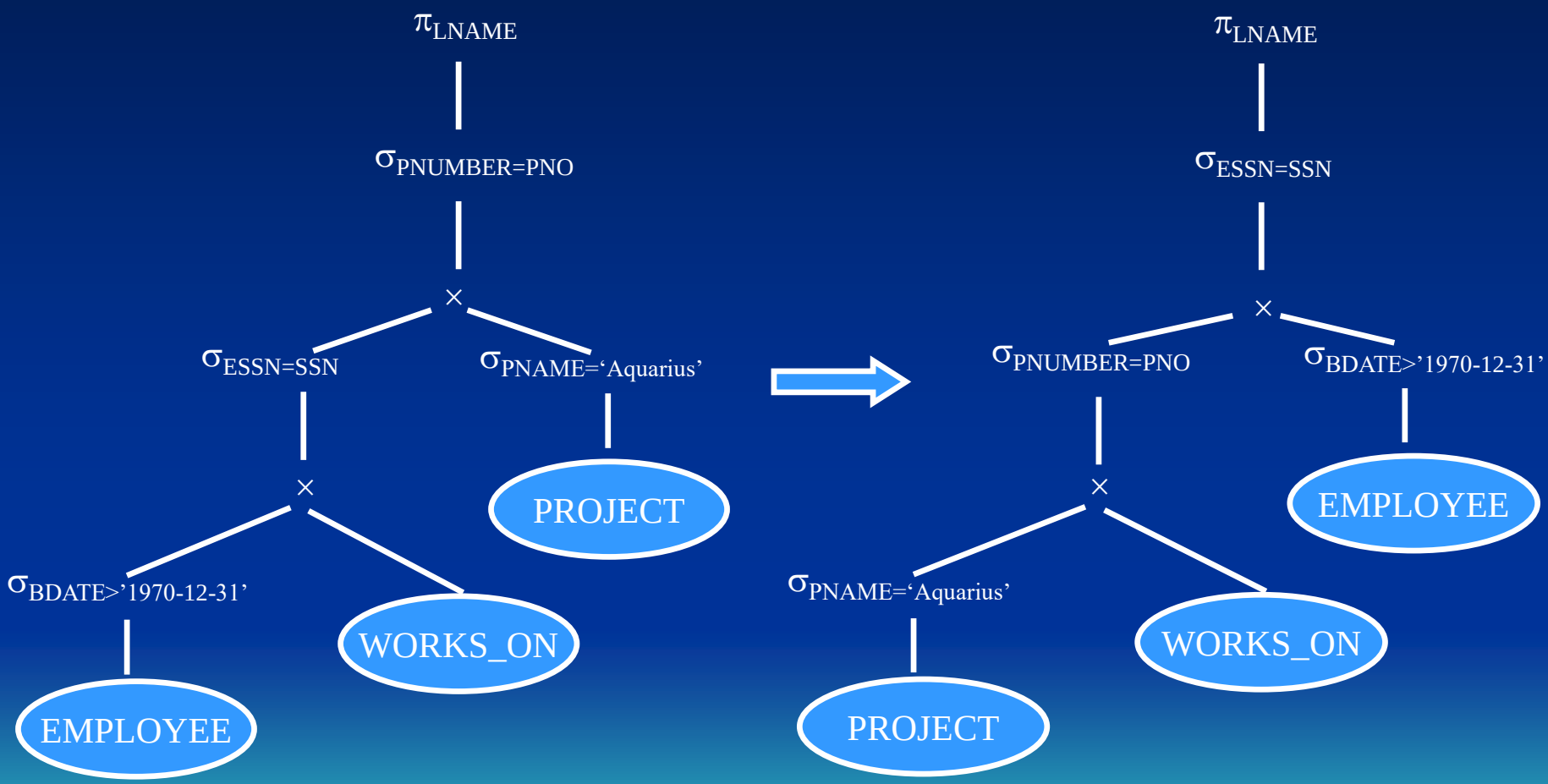
3. Using Rules 5 and 9 concerning commutativity and associativity of binary operations, rearrange the leaf nodes of the tree such that the most restrictive SELECT operations are executed first.

Rule 5: $R \bowtie_c S \equiv S \bowtie_c R$

$R \times S \equiv S \times R$

Rule 9: $(R\ \theta\ S)\ \theta\ T \equiv R\ \theta\ (S\ \theta\ T)$

θ: $\cup$, $\cap$, $\bowtie$, $\times$

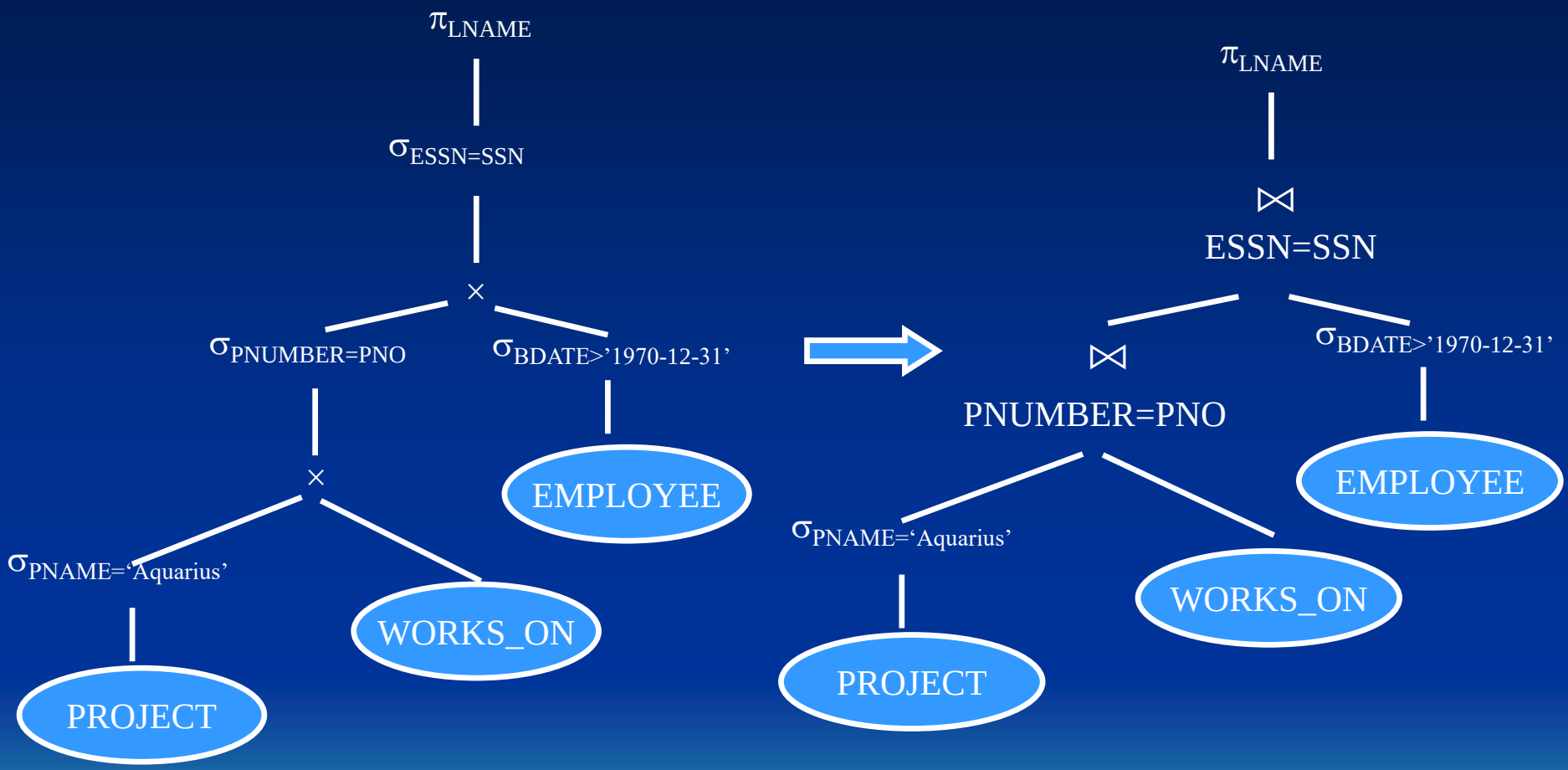

πLNAME
σPNUMBER=PNO
×
σESSN=SSN
σPNAME='Aquarius'
×
PROJECT
σBDATE>'1970-12-31'
WORKS_ON
EMPLOYEE
πLNAME
σESSN=SSN
×
σPNUMBER=PNO
σBDATE>'1970-12-31'
×
EMPLOYEE
σPNAME='Aquarius'
WORKS_ON
PROJECT

- **Outline of a heuristic algebraic optimization algorithm**

4. Using Rule 12, combine a CARTESIAN PRODUCT operation with the subsequent SELECT operation in the tree into a JOIN operation, if the condition represents a join condition.

Rule 12:

$$\sigma_c(R \times S) \equiv R \bowtie_c S$$


πLNAME
σESSN=SSN
×
σPNUMBER=PNO
σBDATE>'1970-12-31'
×
EMPLOYEE
σPNAME='Aquarius'
WORKS_ON
PROJECT
πLNAME
⋈
ESSN=SSN
⋈
σBDATE>'1970-12-31'
PNUMBER=PNO
EMPLOYEE
σPNAME='Aquarius'
WORKS_ON
PROJECT

- **Outline of a heuristic algebraic optimization algorithm**

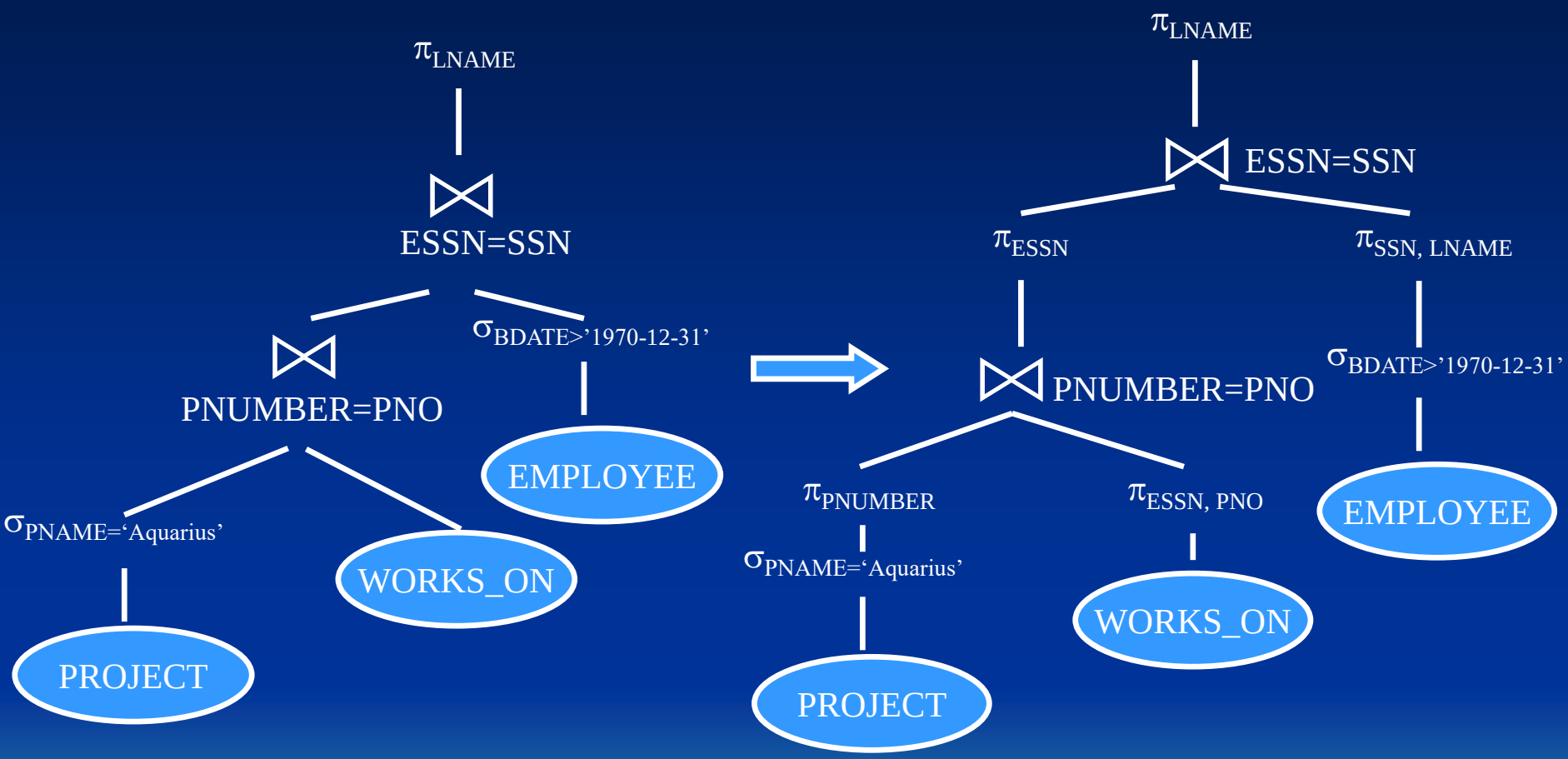
5. Using Rules 3, 4, 7 and 11 concerning the cascading of PROJECT and the commuting of PROJECT with other operations, break down and move lists of projection attributes down the tree as far as possible by creating new PROJECT operations as needed.

Rule 3: $\pi_{list1}(\pi_{list2}(\ldots(\pi_{listn}(R))\ldots)) \equiv \pi_{list1}(R)$
where $list1 \subseteq list2 \subseteq \ldots \subseteq listn$.

Rule 4: $\pi_{A1, \ldots, An}(\sigma_c(R) \equiv \sigma_c(\pi_{A1, \ldots, An}(R))$

Rule 7: $\pi_L(R \bowtie_C S) \equiv (\pi_{A1, \ldots, An}(R)) \bowtie_C (\pi_{B1, \ldots, Bm}(S))$

Rule 11: $\pi_L(R \cup S) \equiv (\pi_L(R)) \cup (\pi_L(S))$


πLNAME
ESSN=SSN
PNUMBER=PNO
σBDATE>'1970-12-31'
σPNAME='Aquarius'
EMPLOYEE
WORKS_ON
PROJECT
πLNAME
ESSN=SSN
πESSN
πSSN, LNAME
PNUMBER=PNO
σBDATE>'1970-12-31'
πPNUMBER
πESSN, PNO
σPNAME='Aquarius'
EMPLOYEE
WORKS_ON
PROJECT

- **Outline of a heuristic algebraic optimization algorithm**

  6. Identify subtrees that represent groups of operations that can be executed by a single algorithm.