
Chapter 7

Data Structure and Algorithm

1 Data Structure

For processing data in a computer, it is necessary to give instructions with a program. At that time, if the way to handle the data is predetermined according to the process, the efficiency is increased. This section explains data structure, which is the format of recording data.

1 - 1 Array

Array is a data structure that is used when a set of data using the same format is handled in a consolidated manner. Every a single data item (i.e., element) is distinguished with a number (a **subscript** or an **index**).

[Notation system of array elements]

- Index may start from 0 or 1. This textbook uses the notation that starts from 1 in consideration of the ease of understanding.
- Index can be enclosed with (), [], and so on. This textbook uses ().

Example 1: Without using an array, make the score of three subjects 0.

English	Japanese	Mathematics
---------	----------	-------------

<Process>

- 1) 0 → English
- 2) 0 → Japanese
- 3) 0 → Mathematics

Example 2: Using an array, make the score of three subjects 0.

Score	Score(1)	Score(2)	Score(3)
-------	----------	----------	----------

<Process>

- 1) 0 → Score(1)
- 2) 0 → Score(2)
- 3) 0 → Score(3)

The advantage of using an array is that when the same process is performed on multiple data items, the process can be described by just changing the value of the index. For example, in the process of example 2, by representing the index with variable *I*, it can be described as follows:

- Repeat the following process while variable *I* is changed from 1 through 3.
- 0 → Score(*I*)

In this manner, when a data structure is used, the processing sequence can easily be

represented. In this example, there may not seem a big difference because there are fewer process targets (i.e., three subjects). However, for larger volumes of data (e.g., 20 subjects), the user may understand how efficient it is.

(1) One-dimensional array

One-dimensional array is a data structure that handles a set of data in the same format by arranging it in a row. “Array” generally refers to a one-dimensional array.

Example: A one-dimensional array that records the results (i.e., scores) of exams in four subjects

	1	2	3	4
Score	Score(1)	Score(2)	Score(3)	Score(4)

(2) Multidimensional array

Multidimensional array is a data structure that handles array elements with multiple relations. For example, a two-dimensional array handles elements with two relations, namely, horizontal and vertical. Therefore, it requires multiple indexes for specifying elements. (The number of indexes is the same as the number of dimensions.)

Example: A two-dimensional array that records the results (i.e., scores) of three persons who took exams in four subjects

Score	1	2	3	4	
1	Score(1,1)	Score(1,2)	Score(1,3)	Score(1,4)	: 1st person
2	Score(2,1)	Score(2,2)	Score(2,3)	Score(2,4)	: 2nd person
3	Score(3,1)	Score(3,2)	Score(3,3)	Score(3,4)	: 3rd person

(3) Structured array

Structured array is a data structure where array elements are a set of data of different formats. It is also called **record array** because it has the same concept as records that are stored in a file.

Example: A structured array that records examinee’s number, name, and total score of three examinees

	Exam_number	Name	Total_score	
1	Exam_number(1)	Name (1)	Total_score (1)	: 1st examinee
2	Exam_number(2)	Name (2)	Total_score (2)	: 2nd examinee
3	Exam_number(3)	Name (3)	Total_score (3)	: 3rd examinee

A general array is stored with the fixed number of elements in contiguous areas of memory (main memory). This is called **static array**. However, some programming languages support **dynamic array** where the number of array elements can be varied during the process.

In addition, an array is an index based data structure that can be accessed directly, and therefore, it is also used as a **hash table** where storage location (i.e., index) is determined from the data to be stored. As the methods of determining storage location (i.e., index), **division method**, **superposition method**, and **radix conversion method** can be applied, which are used when the record address is calculated from the record key of a direct organization file. (A dedicated **hash function** can also be used.) Here, it is also necessary to consider the way of handling synonyms.

1 - 2 List

List (linked list) is a data structure that decides the arrangement of data with a pointer (i.e., position (or address) where element is recorded).

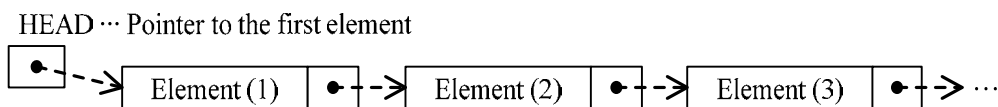


Figure 7-1 Image of list

A list can also be represented by using an array. In this case, indexes that indicates the following elements are recorded in the pointers. Figure 7-2 shows an example of a list where an array is used.

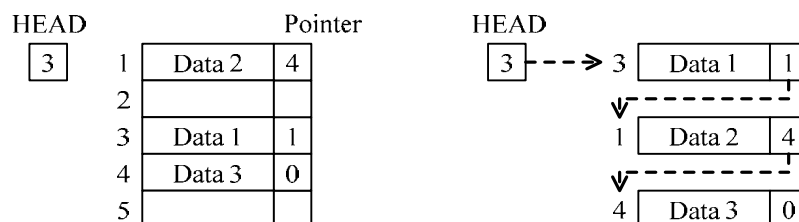


Figure 7-2 Example of a list that uses an array

Since an array is a data structure where the recording order of data has significance, insertion and deletion of data become manipulation of the entire array and are not efficient. By contrast,

in the case of a list where the recording order of data has no significance, insertion and deletion of data can be easily performed by just replacing the pointer.

For example, for inserting data X that is newly added in element (5) between data 1 and data 2 in the list shown in Figure 7-2, the pointer of data 1 (i.e., data that is located just before the data to be inserted) and data X (i.e., data to be inserted) only need to be replaced.

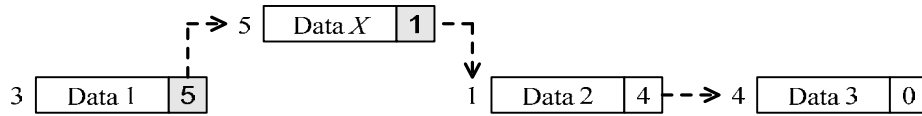


Figure 7-3 Example of inserting data in the list

On the other hand, for deleting data 2 from the list shown in Figure 7-2, the pointer of data 1 (i.e., data that is located just before the data to be deleted) can simply be replaced as shown in Figure 7-4. However, while data 2 is logically deleted from the list, data itself remains behind. Therefore, it is necessary to physically erase the data.

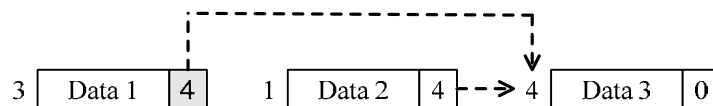
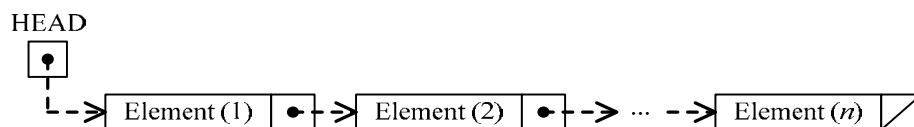


Figure 7-4 Example of deleting data from the list

(1) Singly-linked list

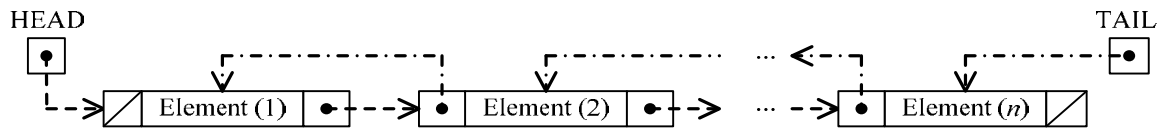
Singly-linked list is a list that can trace data in only one direction. It is also referred to as a **linear list**, a single linked list, or a unidirectional list. Since each element just has a pointer that indicates the recording position of the next element, the amount of data is less. However, its applications are limited because data cannot be traced in the reverse direction. In the pointer of last element of the list, value (e.g., NULL, 0) that indicates end is recorded.



(2) Doubly-linked list

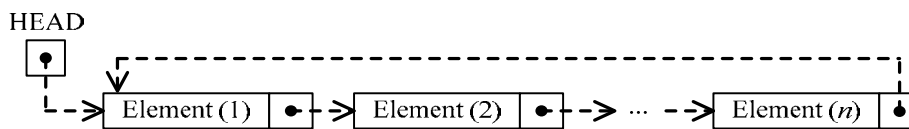
Doubly-linked list is a list where data can be tracked in both directions because it has a forward pointer that indicates the recording position of the next element and a backward pointer that indicates the recording position of the previous element. It is also called a two-way list or a bidirectional list. Since elements can be tracked in both directions, it offers wider usage of data. However, the amount of data becomes large because more pointers are

required, and processing (i.e., replacement of pointers) is also complex.



(3) Circular list

Circular list is a list where all elements are linked in the form of a ring. When a circular list is implemented with a singly-linked list, the pointer to the last element points at the first element.



A general list can be dynamically recorded in the storage area as required. Therefore, this data structure can be easily used even when the number of data items to be handled cannot be predicted. However, elements can only be traced in a sequential access, and therefore it is not suitable when some specific elements are to be processed most of the time. Even in singly-linked lists and circular lists, when a process is mostly performed on the last element of the list (for example, adding data in the last of the list), the tail pointer is used just like in a doubly-linked list. However, it is not possible to trace before the last element, and therefore, it does not result in any significant improvement of efficiency. (Efficiency hardly changes when the last element is deleted from the list.)

1 - 3 Stack and Queue

Stack and **queue** are data structures that model the methods of using data. They are also called **problem resolution data structures**, and they are implemented by using an array.

1-3-1 Stack

Stack is an **LIFO (Last-In First-Out)** data structure where data that is recorded last is extracted first. You can simply imagine a data structure where data is stacked in the box, and data is extracted from the box. Since data is stacked and extracted from the same side, data that is stored most recently is extracted first. Storing data in the stack is called **PUSH**, and extracting data from the stack is called **POP**.

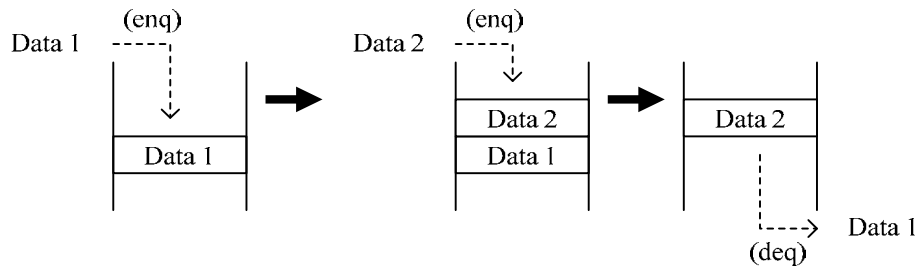
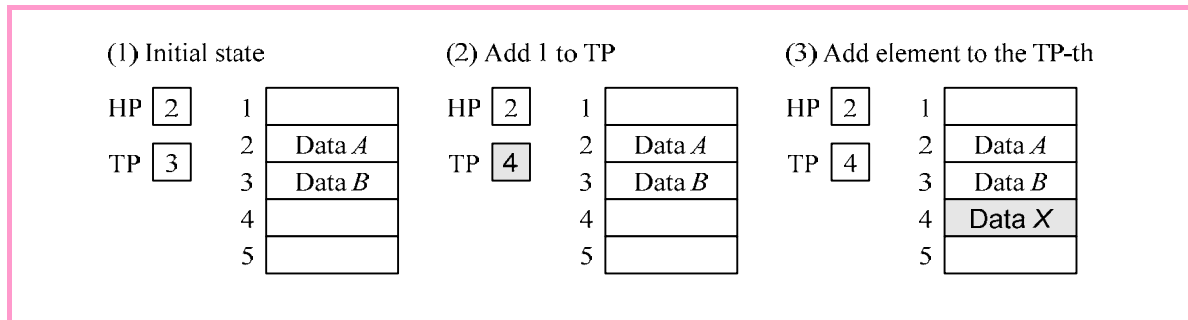


Figure 7-6 Image of queue

For implementing a queue, an HP (Head Pointer) that points at the first element and a TP (Tail Pointer) that points at the last element are used. When data is stored, it is stored at the position that is obtained by updating the value (e.g., generally +1) of pointer TP. In contrast, when data is extracted, the value of pointer HP is updated (e.g., generally +1) after data is extracted from the position of pointer HP. The process (i.e., enq) of storing data in a queue is shown as follows:



When a queue is represented with an array, both pointers (HP and TP) keep increasing by one at a time. Therefore, an infinite number of array elements is required. So, in the actual process, when the value of a pointer becomes larger than the number of array elements, the concept of returning this value to 1 is used. With this concept, array elements are used in circulation, and it becomes possible to implement a queue with an array of the finite number of elements.

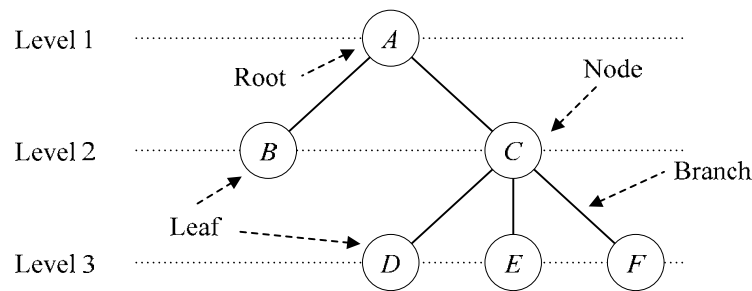


Figure 7-7 Image of circular use of array

1 - 4 Tree Structure

Tree structure is a data structure for representing a one-to-many hierarchical structure where there are multiple children under one parent.

[Constituent elements of tree structure]



- **Node** : This is an individual element (i.e., data) that is shown as a circle.
- **Root** : This is the highest-level node. A tree structure has only one root.
- **Leaf** : This is the node that does not have any lower-level node.
- **Branch** : This is a line that connects to each node (including root, leaf).
- **Level** : This is the depth of hierarchy of a tree structure.

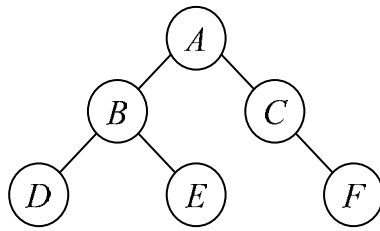
For implementing a tree structure with an array, the relation between parent node and child node is represented with a pointer (i.e., index). For example, if the tree structure shown in the figure above is represented with an array by using a pointer to parent node (parent PT), the following table is obtained. Here, Parent PT of root is 0 (not connected).

	1	2	3	4	5	6
Node value	A	B	C	D	E	F
Parent PT	0	1	1	3	3	3

On the other hand, when a relation is represented with child nodes, it is difficult to handle if the number of child nodes is not decided. Therefore, a **binary tree** is used where the number of child nodes is restricted to two nodes or less. In some cases, a tree structure that has n child nodes where the number of child nodes is more than two is distinguished as **n -ary tree** (also known as multi-way tree or multi-branch tree).

For representing a binary tree by using an array, a left pointer (left TP) that connects to the left child node and a right pointer (right PT) that connects to the right child node are used.

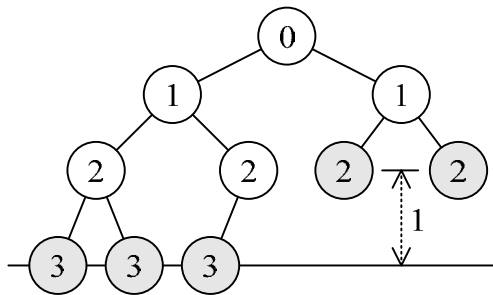
[Array representation of binary tree]



	Left PT	Node value	Right PT
1	2	A	3
2	4	B	5
3	0	C	6
4	0	D	0
5	0	E	0
6	0	F	0

In a binary tree where all final level leaves are left aligned, if level from root up to all leaves is equal or different by one level at most, it is specifically called a **complete binary tree**.

- Complete binary tree



- Tree that is not a complete binary tree

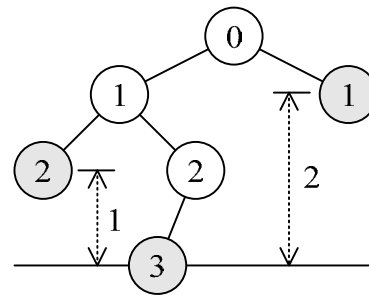


Figure 7-8 Examples of complete binary tree and tree that is not a complete binary tree

NOTE: There is another definition of a complete binary tree; in other words, a complete binary tree (sometimes called a perfect binary tree) is a full binary tree in which all leaves are at the same depth.

In addition, a method of extracting node values (i.e., data) from a tree structure is called **tree traversal**. There are the following types of tree traversals.

- **Breadth-first search**

Node values are extracted at one level at a time from left to right and from root towards leaves.

- **Depth-first search**

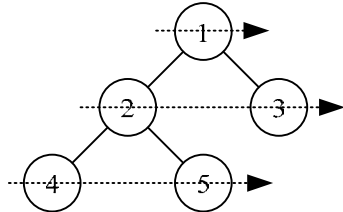
Node values are extracted in order while the periphery is tracked from the root and from left subtree to right subtree.

- **Pre-order**: This extracts node values in the sequence of “node → left subtree →

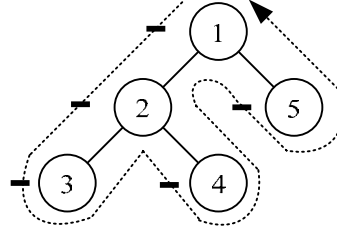
right subtree” (pre-order traversal).

- **In-order:** This extracts node values in the sequence of “left subtree → node → right subtree” (in-order traversal).
- **Post-order:** This extracts node values in the sequence of “left subtree → right subtree → node” (post-order traversal).

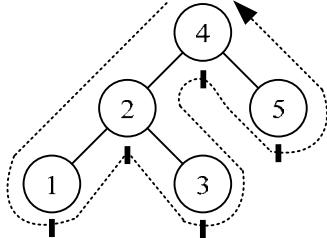
(1) Breadth-first search



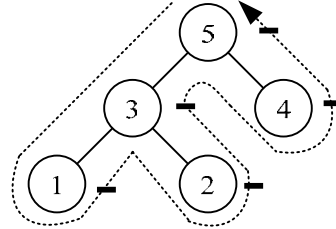
(2) Depth-first search (pre-order)



(3) Depth-first search (in-order)



(4) Depth-first search (post-order)



1-4-1 Practical Use of Binary Tree

(1) Notation of expressions with binary trees

An arithmetic expression can be represented with a binary tree in accordance with the following rules. When an arithmetic expression is represented with a binary tree, extracting node values with depth-first search (i.e., pre-order traversal) gives an arithmetic expression in the **Polish notation**, and extracting node values with depth-first search (i.e., post-order traversal) gives an arithmetic expression in the **reverse Polish notation**.

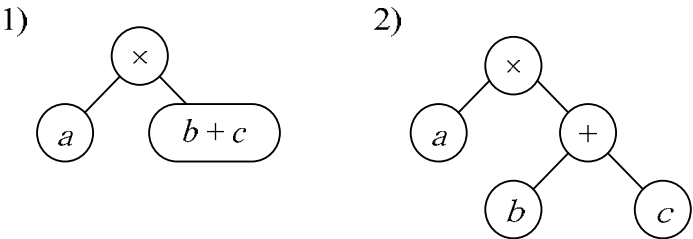
Rule 1: Let the arithmetic operator be the node, and both sides (e.g., variable, expression) of the arithmetic operator be the left child and the right child.

Rule 2: When there are multiple operators, keep the arithmetic operator of the arithmetic operation to be performed later at the higher level.

Rule 3: () is only used for changing the order of arithmetic operations, and it is not described in the node.

Example: Represent the expression “ $a \times (b + c)$ ” in the reverse Polish notation.

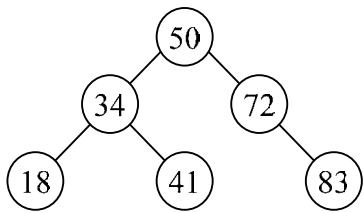
- (1) Represent the expression with binary tree.
- 1) Create a binary tree where the node is arithmetic operator “ \times ” which is executed later.
 - 2) Create a binary tree where “ $b+c$ ” is the right subtree; in other words, arithmetic operator “ $+$ ” is the node, and then, “ b ” and “ c ” are the left child and the right child respectively.



- (2) Extract node values with depth-first search (i.e., post-order traversal).
Reverse Polish notation: $a\ b\ c\ +\ \times$

(2) Binary search tree

Binary search tree is a binary tree where the relation “left subtree $<$ node $<$ right subtree” holds true in all nodes. As its name suggests, binary search tree is a tree structure that is suitable for searching data. (This kind of tree structure is referred to as **search tree**.)



Left subtree	Node	Right subtree
18, 34, 41	50	72, 83
18	34	41
—	72	83

Note: In all nodes,
Left subtree $<$ Node $<$ Right subtree

Figure 7-9 Example of binary search tree

[Data searching procedure in binary search tree]

- 1) Let the root be the first search node.
- 2) Repeat the following process until search nodes are exhausted or the target value is found:
 - Compare node value of the search node with target value.
 - “Node value $>$ Target value”: Let the left child node be the next search node.
 - “Node value $<$ Target value”: Let the right child node be the next search node.

(3) Heap

Heap is a complete binary tree where a constant magnitude relation holds true between the parent node and the corresponding child nodes. Magnitude relation of only the child and parent is constant. There is no constant magnitude relation between sister nodes. (Like a binary search tree or heap, a tree structure with a constant order relation between nodes is called an **ordered tree**).

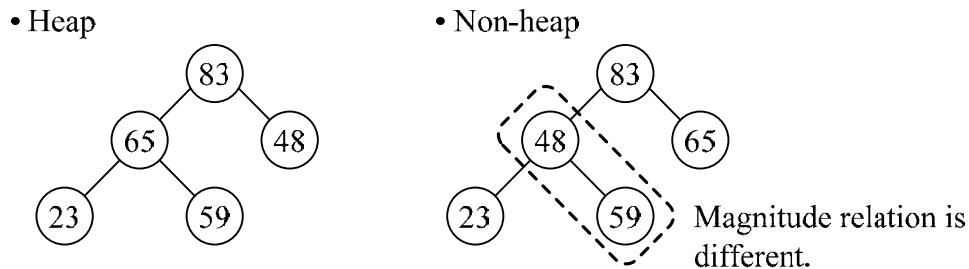


Figure 7-10 Examples of a heap and a non-heap

A heap is **reorganized** for maintaining a complete binary tree structure. In addition, in the root of a heap, the maximum value (or minimum value) of all nodes is recorded. On the basis of this property, it is possible to sort the data by extracting the data of the root in order. (This sorting method is called **heap sort**.)

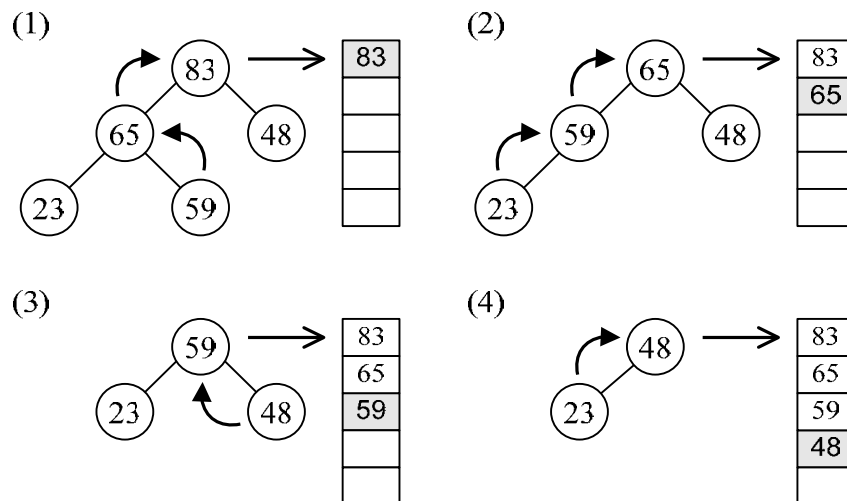


Figure 7-11 Image of heap sort

1-4-2 Balanced Tree

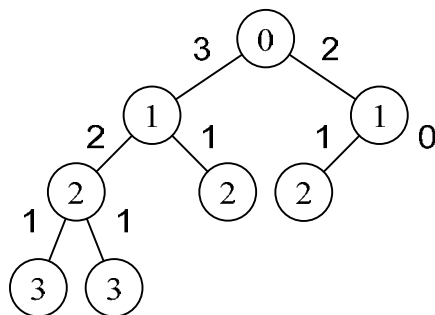
Balanced tree is a tree structure that is **reorganized** to prevent any decline in access efficiency because the level of only a particular leaf becomes deep by adding or deleting data. (A heap is also a type of balanced tree.)

The following two tree structures are the representative balanced trees.

(1) AVL tree

AVL tree is a binary tree where difference in depth between right and left leaves in each node is zero or one.

• AVL tree



• Non-AVL tree

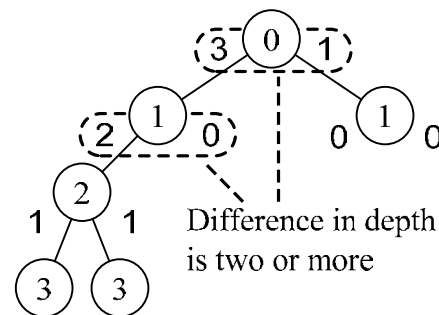


Figure 7-12 Examples of an AVL tree and a non-AVL tree

(2) B tree

B tree is a balanced tree where the concept of a binary tree is developed into n-ary tree. B tree is a data structure that has the following features.

- All leaves are at the same level.
- Each node except the root has n or more and $2n$ or fewer elements.
- In each node (including the root), the number of child node is “number of elements + 1”.
- Each element in the node is sorted.

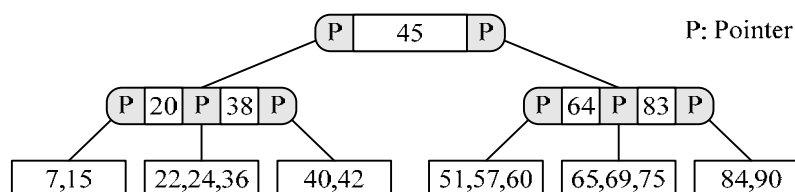


Figure 7-13 Image of B tree

2 Basic Algorithm

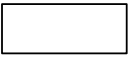
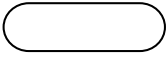
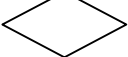
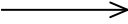
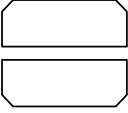
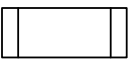
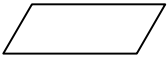
An algorithm is a sequence of solving a problem or a sequence that is used when a process is performed on a computer. An algorithm should have as much processing efficiency as possible, and it should be easy to understand. This section explains how to write flowcharts that describe algorithms, and also explains typical algorithms.

2 - 1 Flowchart

Flowchart is a commonly used notational convention of algorithms. Its main advantage is that it is visually easy to understand because the processing sequence is represented as a combination of symbols.

2-1-1 Main Symbols

The following are the main symbols that are used in flowcharts.

Symbol	Symbol name	Meaning
	Process	This represents various processes (e.g., substitution, calculation).
	Terminal	This represents start/end of flowchart. In some cases, a name is added to the terminal symbol that indicates the start.
	Decision	This branches the process according to conditions.
	Flowline	This connects each symbol and represents execution order.
	Loop limit	This represents the start/end of the iterative process (i.e., loop). It can also be described in the form of "Variable name: Initial value, Increment, Final value".
	Predefined process	This calls a predefined process. (It runs a flowchart that starts with the terminal symbol of identical names.)
	Input/Output	This represents input, output of data (it is also substituted for a process symbol in some cases).

[Structured chart]

Structured charts that are used for representing algorithms include **PAD**, **NS chart**, and so on.

2-1-2 Three Basic Structures

Three basic structures are structure units that are used in **structured theorem** for creating easy-to-understand algorithms that have excellent processing efficiency. (In the structured theorem, if the program has one entry and one exit, it can be represented as a combination of three basic structure units.) When a program is created in consideration of the processing sequence based on the structured theorem, this programming technique is called **structured programming**.

(1) Sequence

“**Sequence**” is a structure that runs the process in sequence from top to bottom. It is the most basic structure, and other structure units are also combined in “Sequence”.

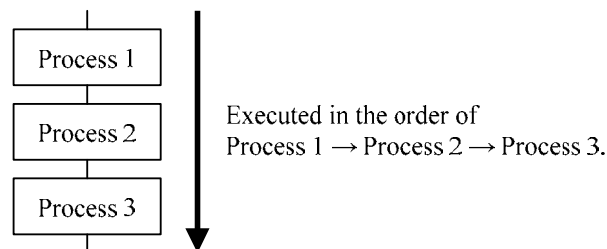


Figure 7-14 Sequence

(2) Selection (Decision)

“**Selection (Decision)**” is a structure that branches the process according to conditions. There is “**two-way selection (IF-THEN-ELSE selection)**” which branches the process into two according to conditions, and there is “**multiple selection (CASE selection)**” which branches the process into multiple (usually three or more) paths. (Generally, “Selection” mostly means “IF-THEN-ELSE selection”.)

“Two-way selection” executes one of the processes depending on whether the condition is true or false. On the other hand, “CASE selection” executes the process that corresponds to the condition that holds true among multiple conditions. However, in both cases, processes are not simultaneously executed in parallel.

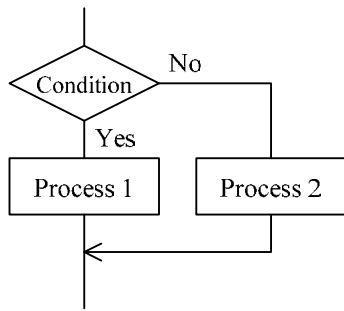


Figure 7-15 IF-THEN-ELSE selection

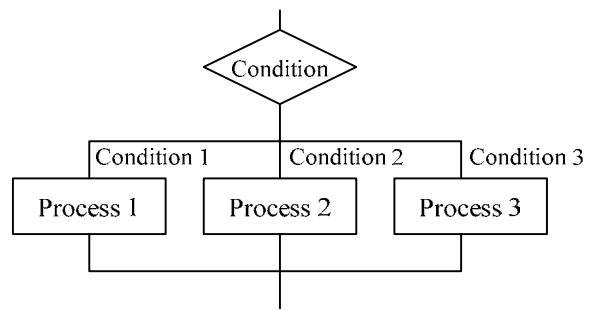


Figure 7-16 CASE selection

(3) Iteration (Loop)

“**Iteration (Loop)**” is a structure that repeats the process while the condition holds true (or until the condition holds true). In the conditions that are used in “iteration,” the **continuing condition** continues the iteration process when the condition holds true, and the **terminating condition** terminates the iteration process as soon as the condition becomes true.

In “Iteration”, there is “**pre-test iteration (DO-WHILE iteration)**” which determines the condition before the iteration process, and there is “**post-test iteration (REPEAT-UNTIL iteration)**” which determines the condition after the iteration process. “DO-WHILE iteration” may not execute the iteration process even once. However, “REPEAT-UNTIL iteration” executes the iteration process at least once.

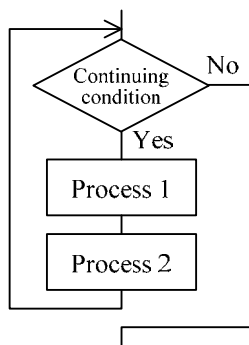


Figure 7-17 DO-WHILE iteration

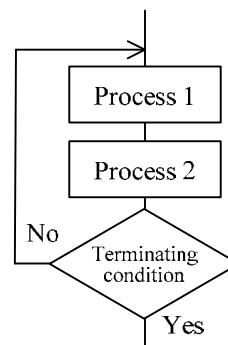


Figure 7-18 REPEAT-UNTIL iteration

In loop limit symbol, the terminating condition is basically used as the condition of “iteration”. Moreover, other than the terminating condition, changes in the value of a variable can be described in the form “Variable name: Initial value, Increment, Final value”.

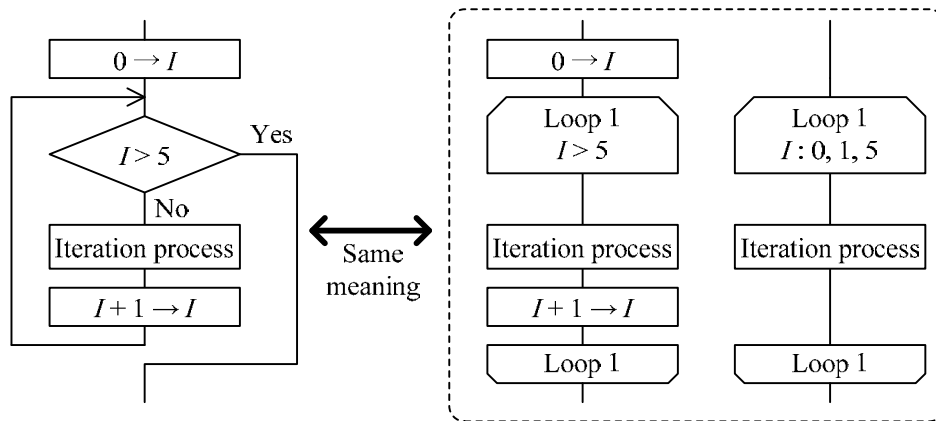


Figure 7-19 Iteration by using loop limit symbol

Three basic structures can be used in any combination. (“Iteration” can be put into the “Selection” process, and “Selection” or “Iteration” can be put into the “Iteration” process.) However, it is better to avoid the **nested structure** as much as possible where there is “Selection” in the “Selection” process.

2 - 2 Data Search Process

The process of finding the target data from the recorded data is called **search**. Representative search algorithms are **linear search** and **binary search**.

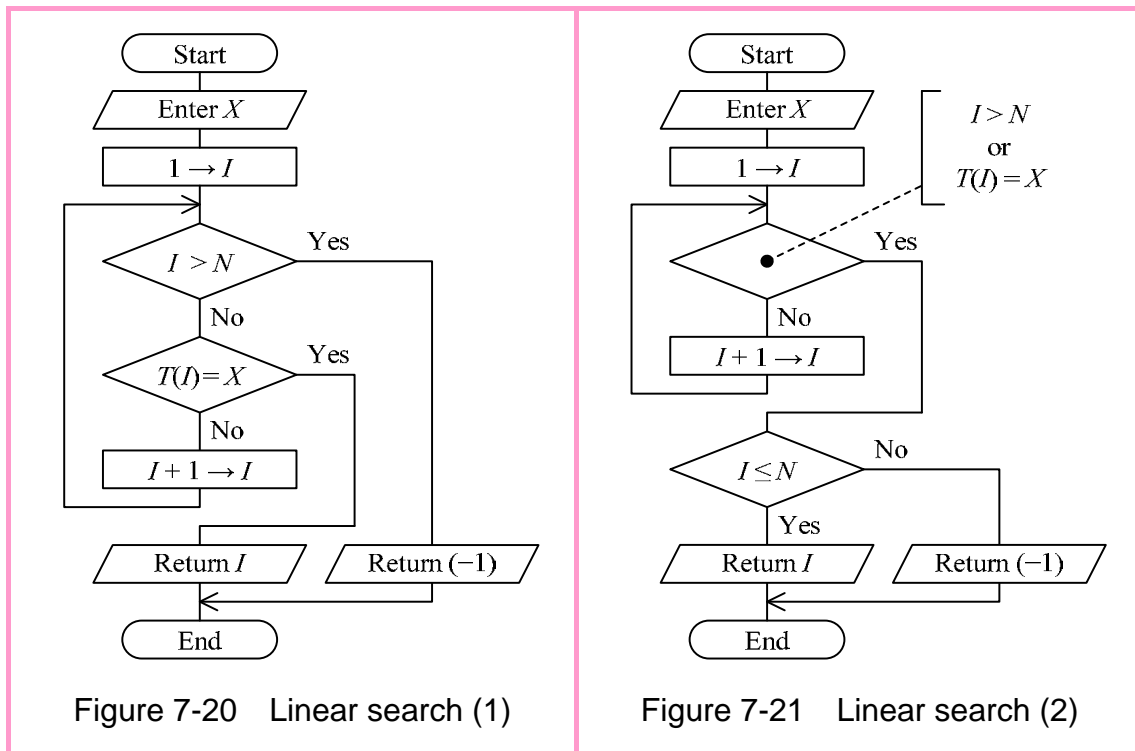
2-2-1 Linear Search

Linear search (sequential search) is the algorithm that searches for the target value in sequence from the first element.

A linear search algorithm that searches for the input target value X from array T is considered. The array T has N elements where data is recorded and returns the position (i.e., index) where the target value X is recorded. When an element that matches with the target value X does not exist, “-1” is returned instead of the index.



In linear search, comparison is made in sequence from the first element $T(1)$ to $T(N)$, in order to determine whether the element matches with the target value X or not. Therefore, in Figure 7-20, the algorithm searches for the element for which “ $T(I)=X$ ” while index I is changed in sequence from 1 to N . (Condition “ $I > N$ ” holds true when there is no element in $T(1)$ through $T(N)$ that matches with X .) However, Figure 7-20 is not a combination of basic structures: that is, exit of iteration is at two places. Therefore, this algorithm is rewritten in Figure 7-21.



In Figure 7-20 and Figure 7-21, two conditions of “ $I > N$ ” and “ $T(I) = X$ ” are compared for every one round of iteration. There is a concept called **sentinel method** that reduces this comparison of conditions (i.e., decision process) and improves process efficiency. In the sentinel method, prior to searching, in the last of all elements, the target value is stored as “sentry”. If “sentry” is stored, the target value is always found. Therefore, it is sufficient to just keep the ending condition of iteration as “ $T(I) = X$ ” where the target value is found”.

Figure 7-22 shows the algorithm of the sentry method that uses the following array T .

	1	2	3	...	N	$N+1$	
T	35	67	21	...	54		... $T(N+1)$: Element where sentry is stored

In addition, linear search can also be used in the list. When a list is used, elements are searched for in sequence while the list is traced by using a pointer from the first element.

In Figure 7-23, an algorithm searches for the entered target value X from the data that is recorded in the list using the following array, and returns the position (i.e., index) where the data is recorded.

HEAD		D	PT	
	1			<ul style="list-style-type: none"> HEAD: Pointer (i.e., index) to the head element D: Data to be searched PT: Pointer (i.e., index) to the next element (0 when there is no next element)
	2			
	3			
	⋮	⋮	⋮	
	N			

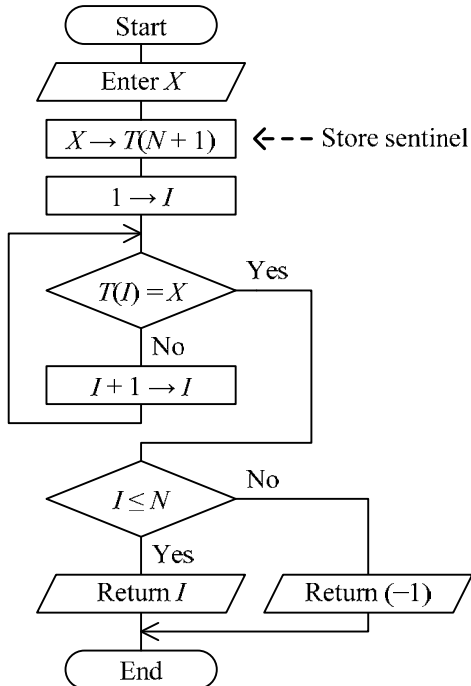


Figure 7-22 Linear search
(sentinel method)

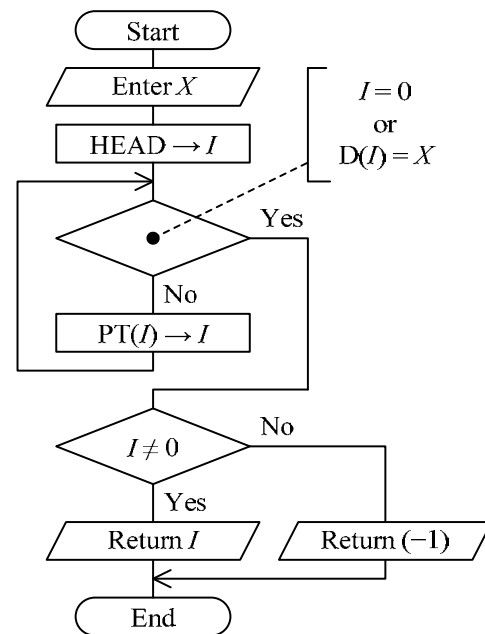


Figure 7-23 Linear search (list)

2-2-2 Binary Search

Binary search is a search algorithm that can only be used when data is recorded in ascending (or descending) order. This algorithm compares the target value and the data that is located in the center of the search range, and narrows down the search range on the basis of the magnitude relation.

A binary search algorithm that searches for the entered target value X from array T that has nine elements is considered. In the array T , data is recorded in ascending order. The binary search algorithm returns the position (i.e., index) where the data is recorded. When an element that matches with the target value X does not exist, “-1” is returned instead of the index.

	1	2	3	4	5	6	7	8	9	
T	12	28	34	45	57	60	71	83	96	X <input type="text"/>

When 34 is entered as target value X , the search procedure is as follows:

- 1) Specify the entire array as the initial search range. Store a smaller index of the search range in L and a larger index in H .

	L								H	
T	12	28	34	45	57	60	71	83	96	

- 2) Compare element $T(5)$ that is located in the center ($M=5$) of a search range with the target value X .

→ Since $T(M) > X$, X is not located after $T(M)$. (Move H to one position before M .)

	L			H	M				
T	12	28	34	45	57	60	71	83	96

- 3) Compare element $T(2)$ that is located in the center ($M=2$) of the search range with the target value X .

→ Since $T(M) < X$, X is not located before $T(M)$. (Move L to one position after M .)

		M	L	H					
T	12	28	34	45	57	60	71	83	96

- 4) Compare element $T(3)$ that is located in the center ($M=3$) of the search range with the target value X .

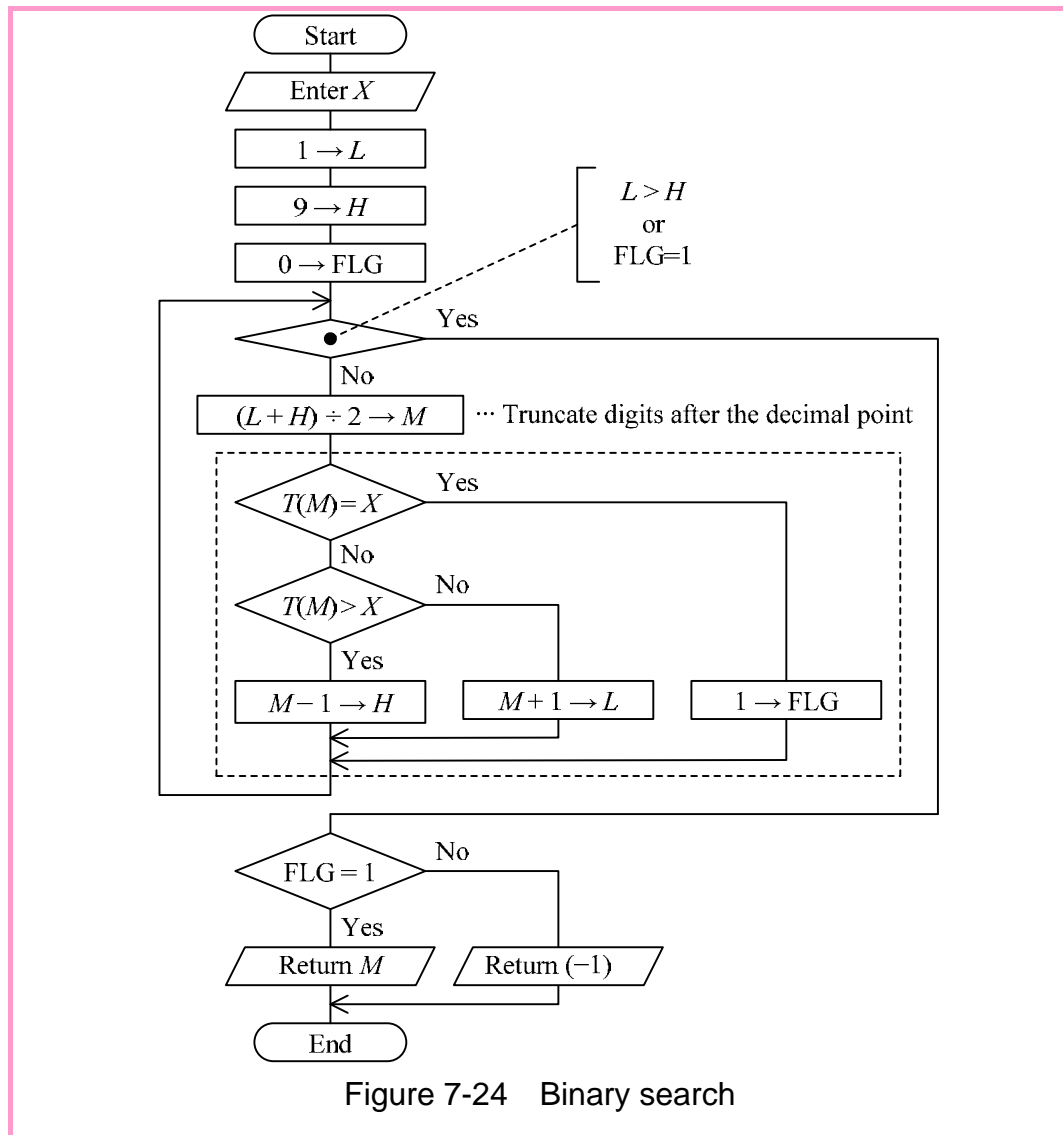
→ Since $T(M) = X$, M is returned and the search process is terminated.

			L, M	H					
T	12	28	34	45	57	60	71	83	96

Notes: 1. Index M of the element that is located in the center of the search range is determined as " $(L+H) \div 2$ ". Here, if the calculation result is a decimal fraction, truncate the digits after the decimal point.

2. If the search range is exhausted in process of repeating the process (that is, when $L > H$), it means there is no element that matches with the target value X .

A flowchart based on this concept is shown in Figure 7-24. In this flowchart, the state during the search is checked with the value of variable FLG (0: Not found, 1: Found). (A variable that represents the state is called a **flag**.) The nested structure part that is covered with a dotted line can be rewritten with CASE selection.



A search that uses a **binary search tree** can also be called a type of binary search. In a binary search tree, “Left subtree < Node < Right subtree” holds true in all nodes. Therefore, the next node to be searched for is decided according to the magnitude relation between the node value and the target value.

[Search using binary search tree (procedure when 41 is the target value)]

	Left PT	Node value	Right PT
1	2	50	3
2	4	34	5
3	0	72	6
4	0	18	0
5	0	41	0
6	0	83	0

- 1) Store index (i.e., 1) of root in I
- 2) Node value(I) > Target value
 \Rightarrow Left PT(I) $\rightarrow I$
- 3) Node value(I) < Target value
 \Rightarrow Right PT(I) $\rightarrow I$
- 4) Node value(I) = Target value
 \Rightarrow Return I and end the search

2-2-3 Hash Search Algorithm

Hash search algorithm is a search algorithm that uses **hash table** which determines the storage location (i.e., index) from the data to be stored. From the target value, the storage location of data is determined by calculation (**hash function**). Therefore, it can basically find the target value with one round of searching. The main point is the process when **synonym** has occurred.

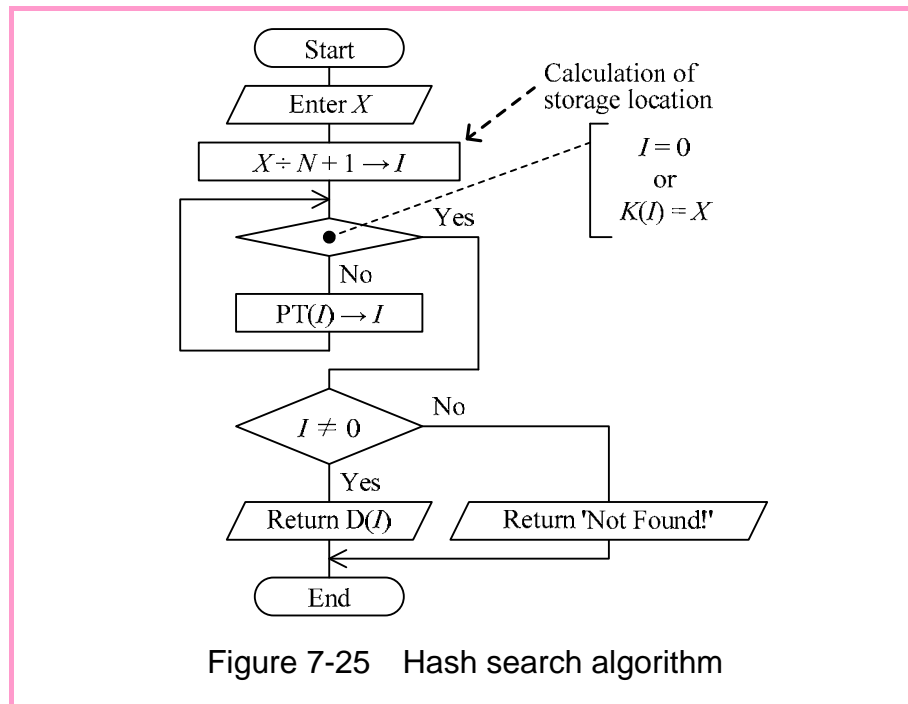
An algorithm is considered. Here, the algorithm searches for the key that matches with the entered target value X from the recorded data in the following hash table (i.e., array) and returns the data that corresponds to this key. When a key that matches with the target value X does not exist, the message “Not Found!” is returned.

	K	D	PT
1			
2			
3			
\vdots	\vdots	\vdots	\vdots
N			
\vdots	\vdots	\vdots	\vdots
M			

- K : Key that decides the storage location (i.e., index)
 $\text{Storage location (i.e., index)} = K \div N + 1$
- D : Data that corresponds to the key
- PT : Pointer (i.e., index) to synonym records
 (0 when there is no synonym record)

} Synonym area (stores synonym records)

In this hash table, **chain method** is used, which reserves the synonym area to store synonym records, and connects them with pointers. Therefore, when the key of the storage location that is determined from the target value does not match with the target value, synonym records are traced with the help of pointers. Figure 7-25 shows this flowchart.



2-2-4 Computational Complexity

Computational complexity is one of the indicators for evaluating algorithms. There is **time complexity** which shows the increase in processing time according to the amount of data to be processed, and **space complexity** which shows the increase in the use of memory space. The notation system of computational complexity includes **big O notation** which shows the upper limit of computational complexity of an algorithm with O (i.e., order). In the big O notation, if computational complexity of the part that has the maximum impact on the overall is proportional to data number N , it is represented as $O(N)$. If it is proportional to the square of N , it is represented as $O(N^2)$.

The computational complexity (e.g., time complexity) for three search algorithms is compared. In a search algorithm, a comparison process (decision process) is repeated until the target value is found. Therefore, the number of comparisons (i.e., number of searches) decides the processing time of the algorithm (or the implemented program).

The following is the number of comparisons made in each search algorithm for N data items that are covered in the search.

	Minimum number of comparisons	Maximum number of comparisons	Average number of comparisons
Linear search	1 time	N times	" $N \div 2$ " times
Binary search	1 time	" $\lceil \log_2 N \rceil + 1$ " times	$\lceil \log_2 N \rceil$ times
Hash search	1 time	" $1 + \alpha$ " times	1 time (See note below.)

- $\lceil x \rceil$ that is referred to as Gauss symbol shows means the greatest integer that is less

than or equal to x .

- $\log_2 N$ shows x for which $2^x = N$ (e.g., $\log_2 16 = 4$).

Note: This is the case the storage location (i.e., hash value) is approximated with uniform distribution, and in addition, the probability of occurrence of a synonym is so small that it can be ignored.

Example: When the number of data has increased from 100 to 200, by how many times does the average number of comparisons increase for a linear search and a binary search, respectively?

- Increase in the average number of comparisons of linear search
= Average number of comparisons of 200 data items
– Average number of comparisons of 100 data items
= $(200 \div 2)$ times – $(100 \div 2)$ times = 100 times – 50 times = 50 times
- Increase in the average number of comparisons of binary search
= Average number of comparisons of 200 data items
– Average number of comparisons of 100 data items
= $\lceil \log_2 200 \rceil$ times – $\lceil \log_2 100 \rceil$ times = 7 times – 6 times = 1 time

Therefore, the computational complexity of linear search is $O(N)$, computational complexity of binary search is $O(\log_2 N)$, and the computation complexity of hash search is $O(1)$.

The computational complexity of big O notation is an indicator (or rough guide), and it may not always match with the actual processing time. For example, when the target value is not found, the maximum number of comparisons is made. Therefore, the average search time varies depending on the probability of search success and search failure. If an improvement is made so that the search ends as soon as it is known that the target value does not exist by using the data recorded in ascending order (or descending order) in linear search, the number of comparisons can be reduced in the event of search failure. However, the order does not change.

2 - 3 Data Sorting Process

The process of rearranging the recorded data in ascending (or descending) order of the specified items is called **sorting**. Sorting is broadly classified into two.

- **Internal sorting**

Internal sorting is performed when the data to be sorted is recorded in main memory. Selection sort, bubble sort, insertion sort, quick sort, heap sort, shaker sort, shell

sort

- **External sorting**

External sorting is performed when the data to be sorted is recorded in auxiliary storage.

Merge sort

2-3-1 Selection Sort

Selection sort is a sorting algorithm that decides one element at a time in sequence from the first element.

An algorithm of the selection sort is considered. The algorithm sorts array S with N elements where data is recorded in ascending order. For simplifying the question, the following array S with 5 elements (i.e., $N=5$) is used.

	1	2	3	4	5	
S	28	84	73	16	51	N 5 ... Number of elements

The following is the procedure that sorts array S in ascending order with the selection sort.

- 1) Select the minimum value from all elements, and let it be the first element $S(1)$.

S	16	84	73	28	51
-----	----	----	----	----	----

- 2) Select the minimum value from the remaining elements, and let it be the 2nd element $S(2)$.

S	16	28	73	84	51
-----	----	----	----	----	----

- 3) Select the minimum value from the remaining elements, and let it be the 3rd element $S(3)$.

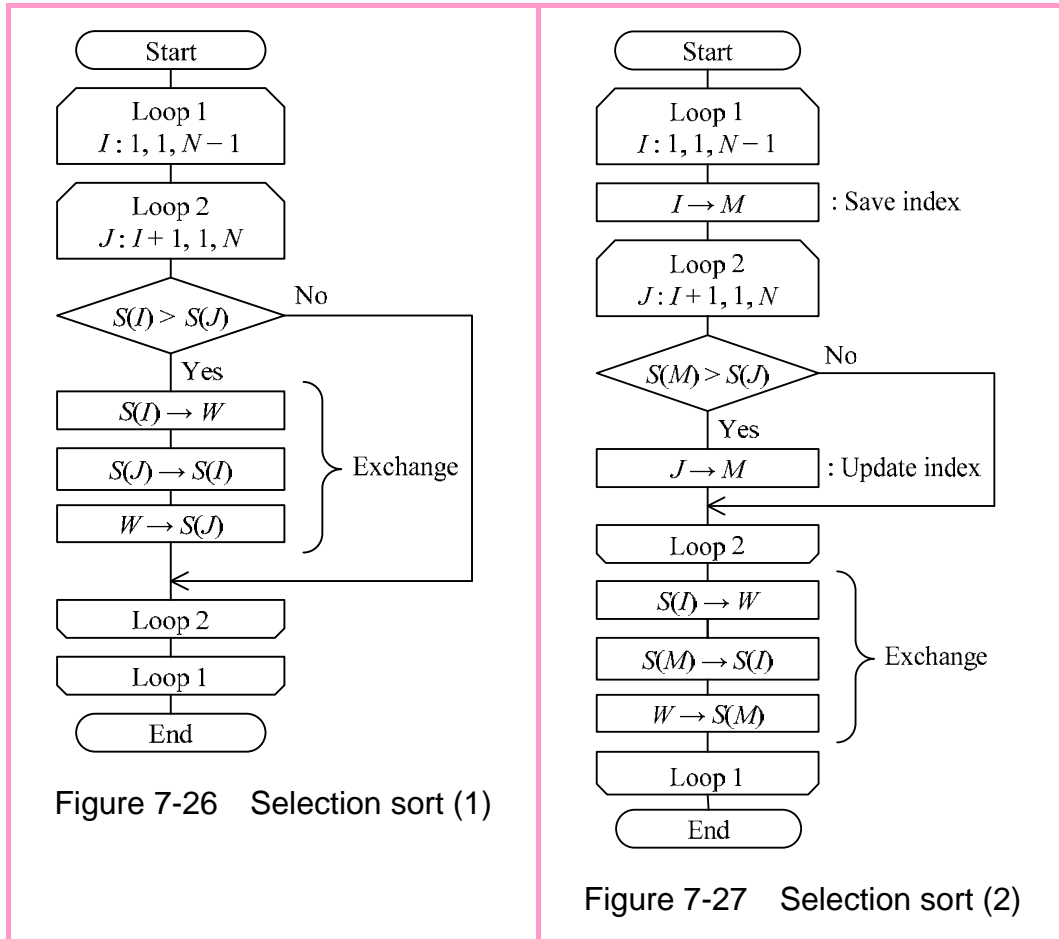
S	16	28	51	84	73
-----	----	----	----	----	----

- 4) Select the minimum value from the remaining elements, and let it be the 4th element $S(4)$.

→ The 5th element $S(5)$ is automatically decided. (Sorting ends.)

S	16	28	51	73	84
-----	----	----	----	----	----

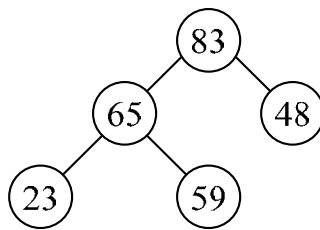
Figure 7-26 and Figure 7-27 show the flowcharts that are prepared on the basis of this concept. In Figure 7-26, elements are exchanged each time comparison is made. However, in Figure 7-27, exchange is performed only once after deciding the element to be exchanged at the completion of comparison. (The procedure that is mentioned above corresponds to Figure 7-27.) In all methods, for deciding the elements to be stored in $S(I)$, comparison is repeated with $S(I+1)$ through $S(N)$.



[Heap sort]

Heap sort is a sorting algorithm that is an improved selection sort. It represents the data to be sorted with a heap, and then sequentially extracts the data of root. (Refer to p.414 for the details of heap and heap sort.)

In the array representation of a heap, the array of the root is placed at the 1st position. The left child node of the I -th element is placed in $(2 \times I)$ -th position, while the right child node is placed at $(2 \times I + 1)$ -th position. By using this relation, a heap is reorganized while a parent node and a child node are compared and exchanged.



	1	2	3	4	5	6
H	83	65	48	23	59	
	↓ Extract root (83) and reorganize.					
H	65	59	48	23		
	↓ Extract root (65) and reorganize.					
H	59	23	48			

2-3-2 Bubble Sort

Bubble sort is a sorting algorithm that compares adjacent elements. If the magnitude relation is reverse, it exchanges the elements to correct the relation. This operation is performed in sequence from the first element. The name “bubble sort” was chosen because the action of gradually moving the maximum value (or the minimum value) in the rear direction resembles the movement of bubbles that are generated in water.

By using array S ($N=5$) that is used in selection sort, the algorithm of bubble sort is considered. Here, this algorithm sorts array S in ascending order where array S has N elements where the data is recorded.

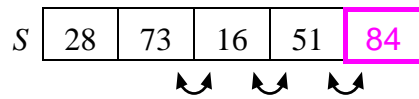
	1	2	3	4	5	
S	28	84	73	16	51	

N	5	... Number of elements
-----	---	------------------------

The procedure of sorting array S in ascending order with bubble sort is as shown below. The process of repeating comparison/exchange from the first element up to the last element is called one round of path.

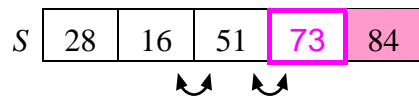
1) Make the 1st comparison/exchange in order from the first element.

→ Decide the 5th element $S(5)$.



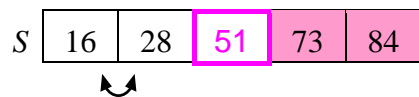
2) Make the 2nd comparison/exchange in order from the first element.

→ Decide the 4th element $S(4)$.



3) Make the 3rd comparison/exchange in order from the first element.

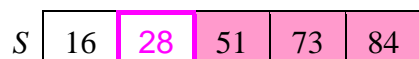
→ Decide the 3rd element $S(3)$.



4) Make the 4th comparison/exchange in order from the first element.

→ Decide the 2nd element $S(2)$.

→ The 1st element $S(1)$ is automatically decided. (Sorting ends.)



In this explanation, last elements are decided one by one and are removed from the scope of sorting. The path is repeated until positions of all elements are decided. In Figure 7-28, this algorithm is implemented by reducing N elements one by one for each round of a path.

On the other hand, if exchange does not occur even once in one round of a path, it means that all elements are arranged in the correct order and sorting ends. Figure 7-29 shows the method that uses the value of variable FLG (0: initial state, 1: exchange occurred) and terminates the process if exchange does not occur.

When efficiency is important, these two methods can also be used in combination. Please think about what kind of flowchart it turns into.

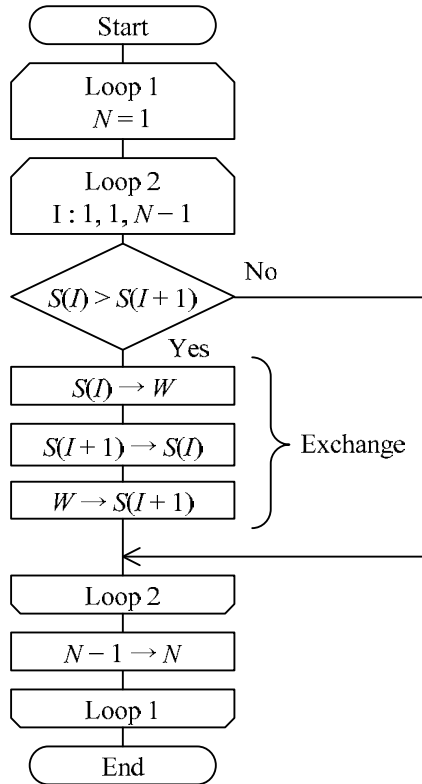


Figure 7-28 Bubble sort (1)

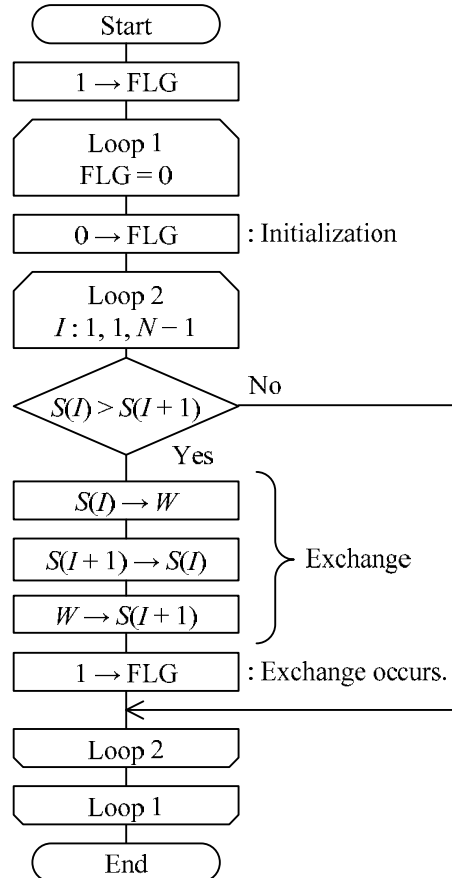
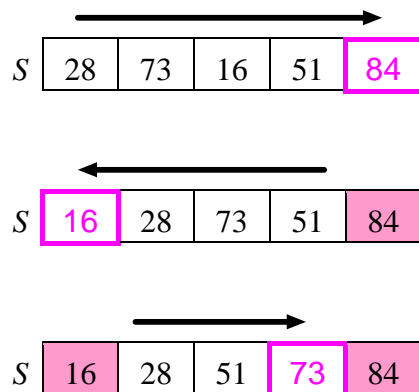


Figure 7-29 Bubble sort (2)

[Shaker sort]

Shaker sort is a sorting method that is obtained by improving bubble sort. Bubble sort compares/exchanges adjacent elements in order from the smaller element number, while shaker sort compares/exchanges adjacent elements in order from the smaller element number and then compares/exchanges in order from the larger element number. Since elements move alternately from a smaller number to a larger number and a larger number to a smaller number, it is called shaker sort. However, efficiency does not improve that much.



2-3-3 Insertion Sort

Insertion sort is a sorting algorithm that divides the data into the sorted part and elements before sorting, and then inserts data so that the order of the sorted part is not disturbed.

By using array S ($N=5$) that is used in selection sort, the algorithm of insertion sort is considered. This algorithm sorts array S in ascending order where array S has N elements where data is recorded.



The procedure of sorting array S in ascending order with insertion sort is as follows:

- 1) Assume that the 1st element has already been sorted.

S

28

84	73	16	51
----	----	----	----

- 2) In the sorted part (1-1), insert “84” from the elements before sorting.

S

28	84
----	----

73	16	51
----	----	----

- 3) In the sorted part (1-2), insert “73” from the elements before sorting.

S

28	73	84
----	----	----

16	51
----	----

- 4) In the sorted part (1-3), insert “16” from the elements before sorting.

S

16	28	73	84
----	----	----	----

51

- 5) In the sorted part (1-4), insert “51” from the elements before sorting.

→ There is no element before sorting. (Sorting ends.)

S

16	28	51	73	84
----	----	----	----	----

In insertion sort, for inserting data in the sorted part in ascending order, comparison is made in order from the last element of the sorted part, and exchange is repeated until a value smaller than the inserted data is found or no comparison element is left (or exchange is performed up to the first element). In step 5) of the above example, the process of inserting element “51” is shown as follows:

- * In the sorted part (1-4), insert “51” from the elements before sorting.

S

16	28	73	84	51
----	----	----	----	----

- 1) Since “84 > 51”, exchange two elements.

S

16	28	73	51	84
----	----	----	----	----

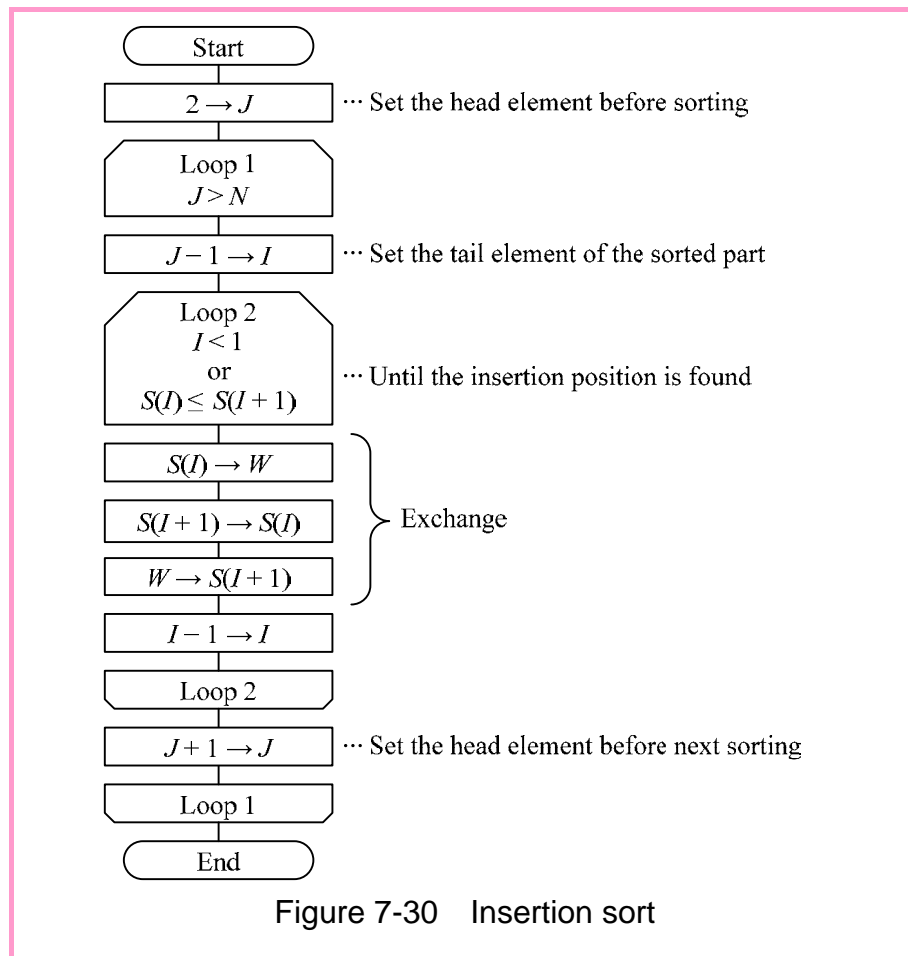
- 2) Since “73 > 51”, exchange two elements.

S

16	28	51	73	84
----	----	----	----	----

- 3) Since “28 < 51”, the insertion position is decided.

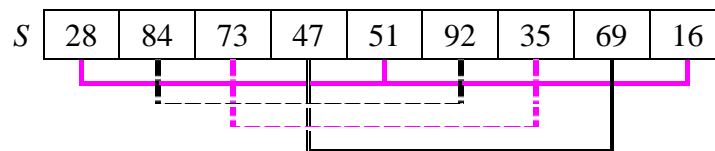
Figure 7-30 shows the flowchart that is prepared on the basis of this concept. In Figure 7-30, the first element (i.e., data to be inserted) before sorting is managed with variable J .



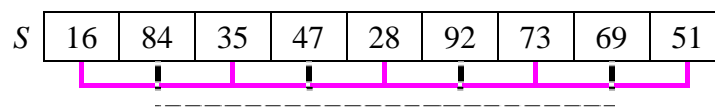
[Shell sort]

Shell sort is a sorting technique that is an improved insertion sort. Data to be sorted is extracted at regular intervals (i.e., gaps), and the data is sorted by using insertion sort. The gap size is gradually reduced, and the process is repeated until it becomes 1. It is a very efficient sorting technique where no large movement of data is required.

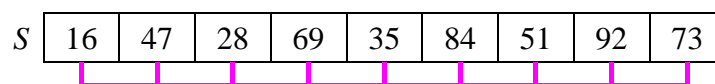
1) Gap = 4



2) Gap = 2



3) Gap = 1

**2-3-4 Quick Sort**

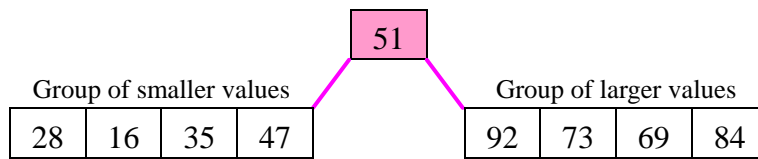
Quick sort is a high-speed sorting algorithm. The reference value is selected from the data group to be sorted. Next, data is divided into the group that has values smaller than the reference value and the group that has values larger than the reference value. After that, the reference value is selected for each group, and they are divided in a similar manner. It is a **recursive algorithm** that repeatedly calls itself until the number of elements in each group becomes 1.

By using array S ($N=9$) below, the algorithm of quick sort is considered.

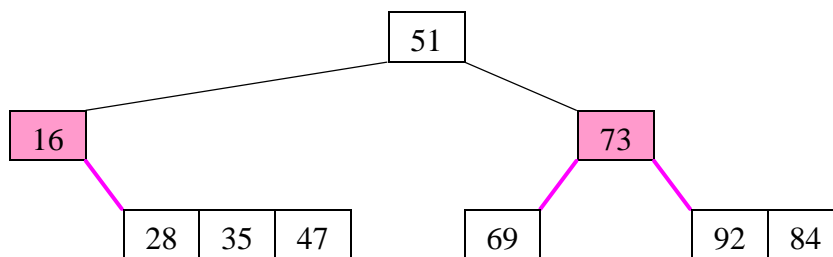
	1	2	3	4	5	6	7	8	9	
S	28	84	73	47	51	92	35	69	16	N 9

The following is the image of a procedure of sorting array S in ascending order with quick sort. The element in the middle of the target data is used as the reference value.

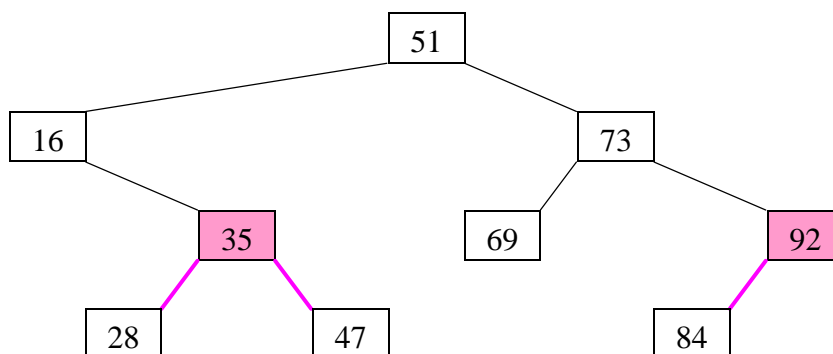
- 1) Divide $S(1)$ through $S(9)$ into the group that has values smaller than reference value $S(5)$ and the group that has values larger than reference value $S(5)$.



- 2) In the group of smaller values $\{S(1)$ through $S(4)\}$ and the group of larger values $\{S(6)$ through $S(9)\}$, decide the reference value for each group and split the groups.



- 3) In groups $\{S(2)$ through $S(4), S(8)$ through $S(9)\}$ where there are more than one elements, decide the reference value for each group and split the groups.



- 4) Sorting is complete when all groups have only one element.

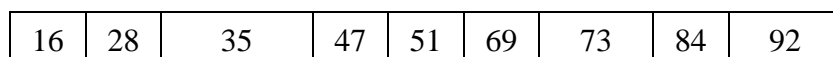


Figure 7-31 shows the flowchart that is prepared on the basis of this concept. In Figure 7-31, quick sort process is recursively called with “QSORT (index of leftmost element, index of rightmost element)”.

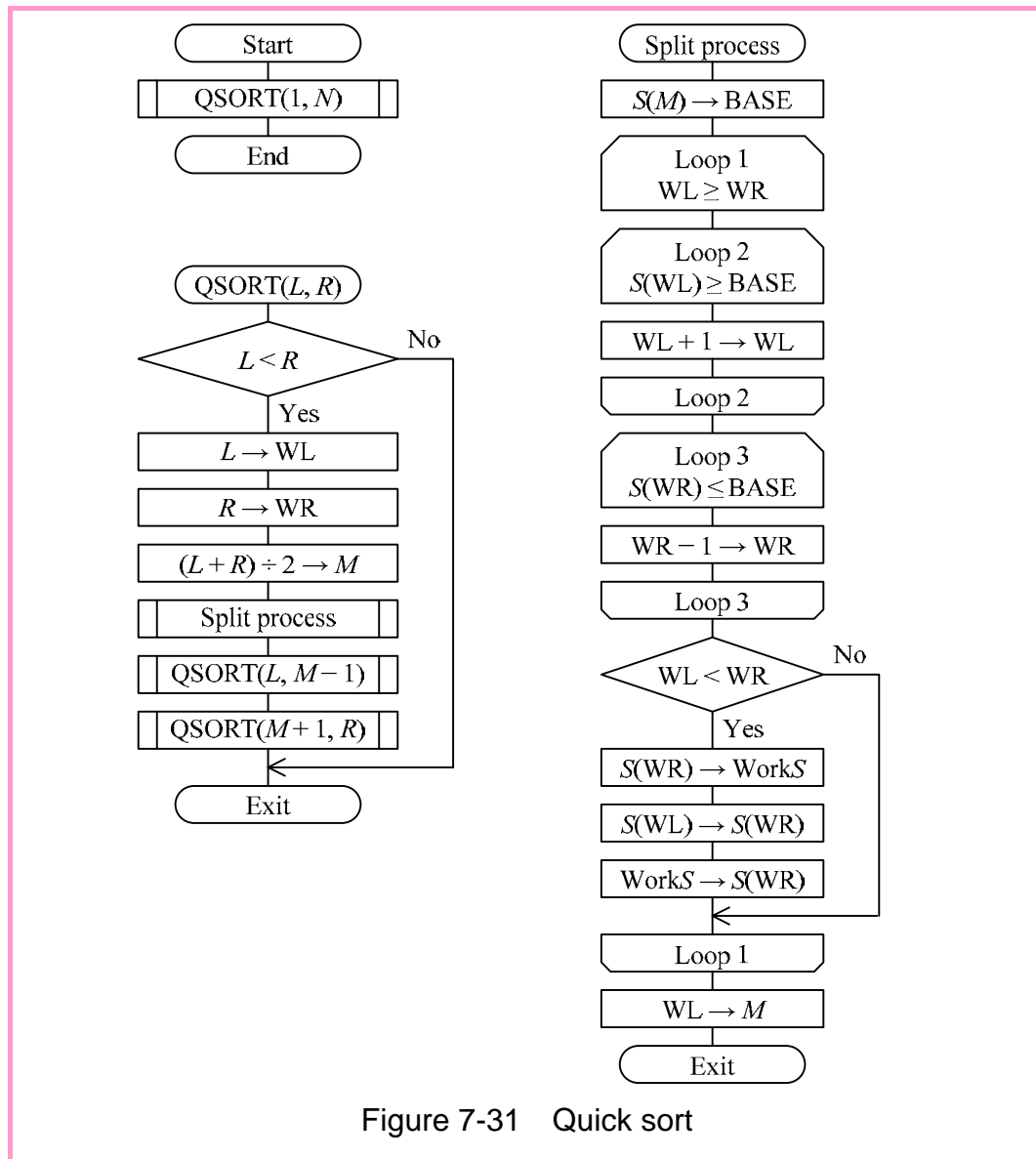
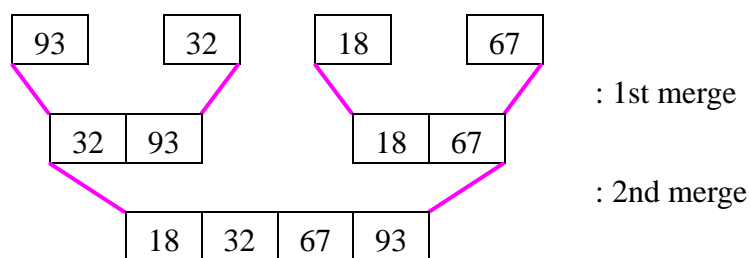


Figure 7-31 Quick sort

[Merge sort]

Merge sort is a sorting algorithm that uses **merge**, which merges two sorted data items into one without disrupting the order. This algorithm recursively performs a series of processes of splitting and merging data.



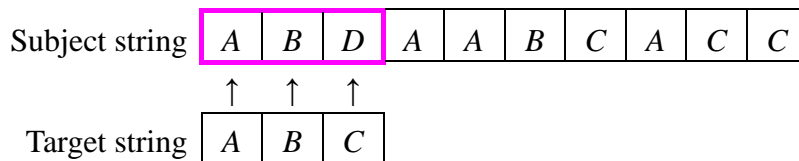
2 - 4 Other Algorithms

2-4-1 Character String Processing

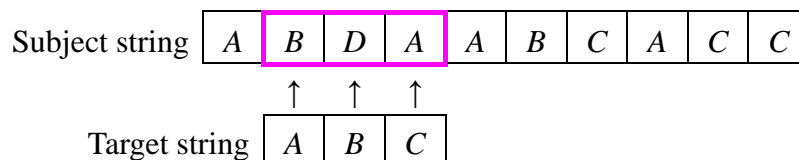
Character string processing is an algorithm that uses characters as the elements of an array. **String pattern matching** is the main string processing that searches for the target character string from the character strings that are recorded in the array. The basic concept is the same as when numerical values are recorded in the elements of an array. However, since the target value is a character string, it must be confirmed whether multiple elements simultaneously match or not.

The easiest concept of string pattern matching is the method of matching a substring (i.e., string that has the same number of characters as the target string) in array elements while one character is shifted from the first at a time.

- 1) Compare the target string with the substring that is located at the first in array elements.

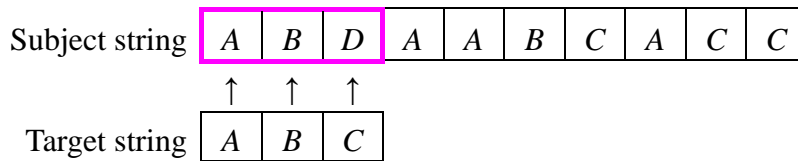


- 2) When the substring does not match, compare after the starting position is shifted by one character.

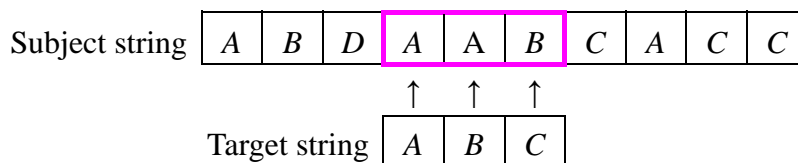


While this procedure is correct, efficiency is not very good. Therefore, the following **Boyer-Moore method** was proposed as an effective matching technique of character strings.

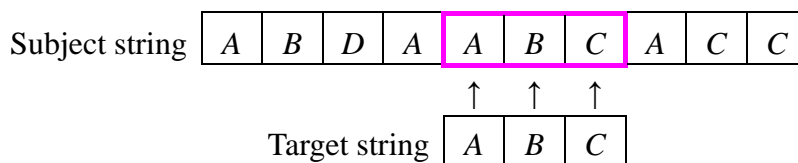
- 1) Compare the target string with the substring that is located at the first in array elements.



- 2) When the tail character of the substring does not exist in the target string, make a comparison after the starting position is shifted by the number of characters in the target string.



- 3) When the tail character of the substring exists in the target string, make a comparison by shifting the starting position so that the position of the tail character is matched with the position of the corresponding character in the target string.



Other than this string pattern matching, the string process includes compression/decompression algorithms of strings, such as the **run length method**.

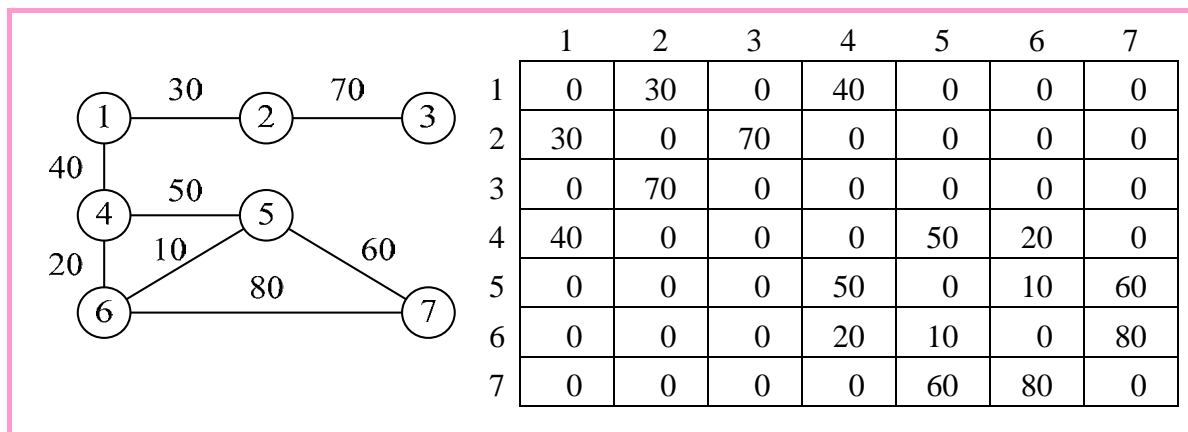
2-4-2 Graph Processing

Graph processing is an algorithm that uses a graph. **Graph** is a pictorial figure that is made of nodes and branches (or edges). (A tree structure is also a kind of graph.)

Graphs include **directed graph** where branches have a direction like a state transition diagram, and **undirected graph** where branches have no direction like a tree structure or a network diagram.

A graph can be represented with a matrix, array, list, and so on. In algorithms, mostly an **adjacency matrix** (i.e., array) is used that shows the adjacent (or linked) status of nodes. The following is an example of an adjacency matrix that shows an undirected graph (i.e., network diagram). In this example, cost (e.g., distance) of branches that connect nodes is recorded to

show that there is a branch. (0 if there is no branch.)



Representative graph processing is **route searching** that searches for the route from the origin point (i.e., node) up to the target point (i.e., node). The following are the two main methods of route searching.

- **Depth-first search**

Depth-first search searches one path until the end (i.e., dead-end or target point). When it is found that the route is not the solution (e.g., ①→②→③), it returns to the earlier point and searches for a different route. In this manner, it is a **backtrack method (backtracking)**.

- **Breadth-first search**

Breadth-first search searches the route in a hierarchical manner. It repeats the process of determining the node at one layer below adjacent to each node until the target node is found.

However, in these two methods, the route that was found may not be always the shortest route (i.e., route with minimum cost: ①→④→⑥→⑤→⑦). Therefore, as the method of **shortest route search** (or shortest path search), the **Dijkstra method** was proposed so that the lowest cost node can be finalized one by one in order.

[General procedure of Dijkstra method]

- 1) At the node adjacent to origin node, finalize the lowest cost node X.
- 2) Including the adjacent nodes to the finalized node X, finalize one lowest cost node among them.
- 3) Repeat the process of step 2) until the lowest cost of all nodes is finalized.
=> Decide the lowest cost (i.e., route) up to the target node.

2-4-3 Numerical Processing

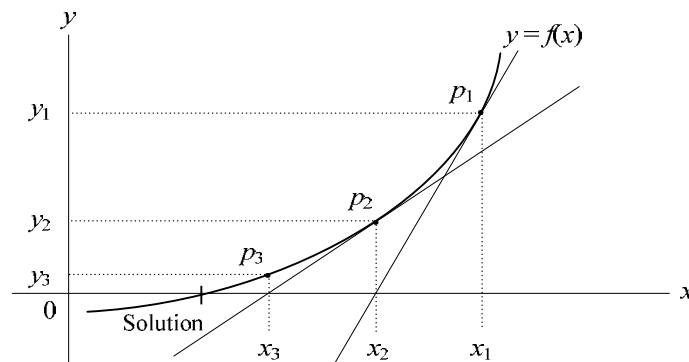
Numerical processing is an algorithm that uses numerical calculation for determining an approximate solution.

(1) Newton's method

When the approximate solution of n order equation is known where n is large, **Newton's method** determines the approximate value of a real solution while this approximate solution is modified.

[Algorithm of Newton's method]

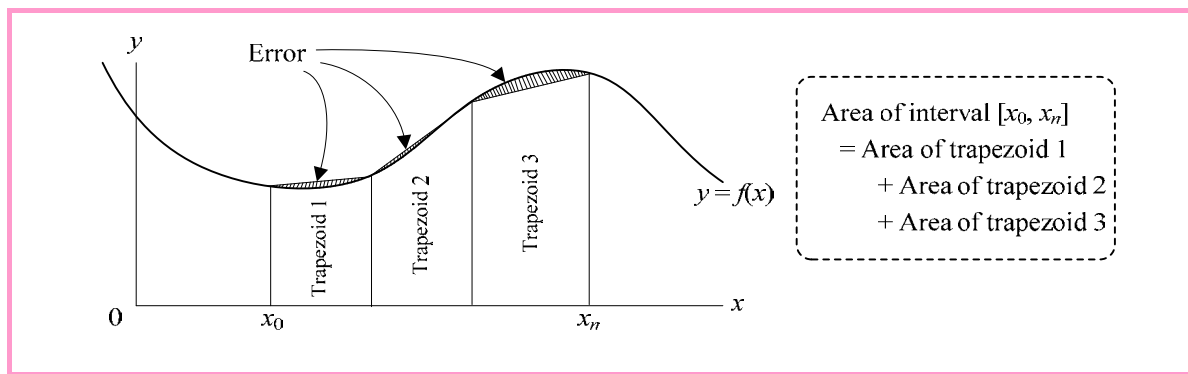
- 1) Let tangent of $y=f(x)$ and point of intersection with x axis in point $p_1(x_1, y_1)$ be x_2 .
- 2) Let tangent of $y=f(x)$ and point of intersection with x axis in point $p_2(x_2, y_2)$ be x_3 .
- 3) Similarly, determine x_4, x_5, \dots , and when the difference between x_n and x_{n+1} converges below a certain value, let x_{n+1} be the solution.



Newton's method cannot be used unless function $f(x)$ is differentiable, because it is necessary to determine the tangent of function $f(x)$. In addition, x coordinate of one point on the curve can be assigned as the initial value. However, a solution may not converge depending on how the initial value is assigned: that is, approximate value of solution might not be obtained.

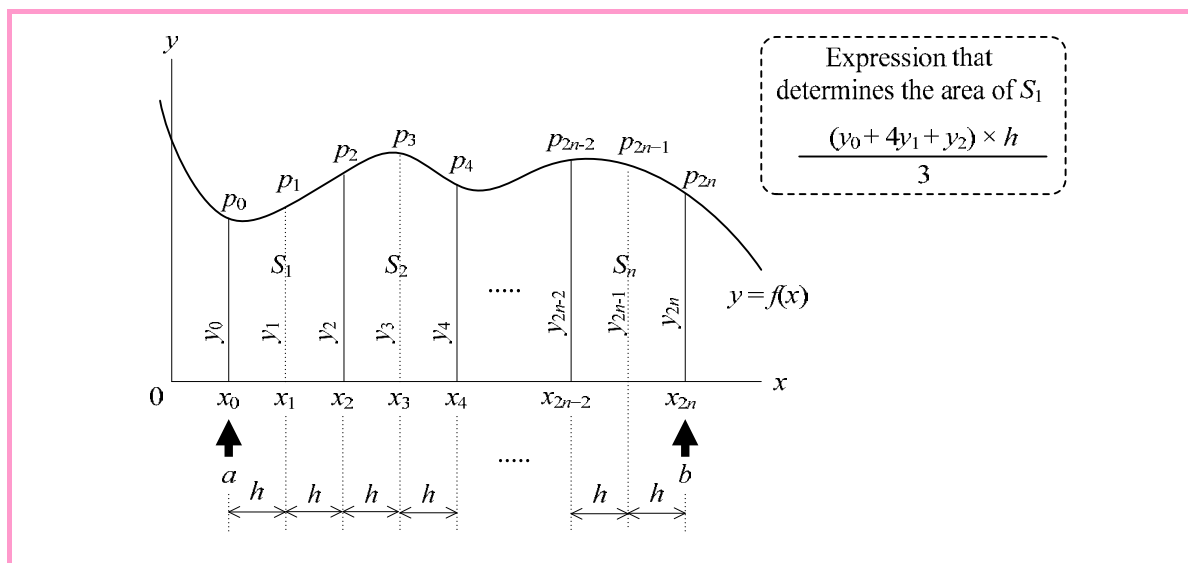
(2) Simpson's method

Trapezoidal rule is one of the methods of calculating the approximate value of an area between the x -axis and the curve over the given interval. In the trapezoidal rule, part of the area to be calculated is split into trapezoids, and the total is determined.



However, the trapezoidal rule approximates a curve with a straight line. Therefore, it suffers from the disadvantage that errors become large. The only way of reducing the error is to increase the number of trapezoids to be split. However, if the number of trapezoids to be split is large, the algorithm becomes complex and computational complexity increases.

For this reason, in **Simpson's method**, when the area after splitting is determined, the portion that corresponds to the side of $y=f(x)$ is approximated with a parabola. In specific terms, the area of S_1 portion that is split as shown below is determined with the expression by considering a parabola that passes through point p_0 , p_1 , and p_2 instead of connecting p_0 and p_2 with a straight line.



In the trapezoidal rule, the portion of the area to be determined is divided into n equal parts. However, the characteristic of Simpson's method is that it is divided into $2n$ (i.e., even number) equal parts.

2-4-4 File Processing

File processing is an algorithm that uses files where records are stored. In the basic file processing, a series of flows like “reading records from the input file”, “processing (e.g., calculation process, editing process) input records”, and “writing records in the output file” is repeated until no record is left in the input file.

- **Sorting process/merging process**

Sorting process/merging process is a process of sorting records recorded in a file and merging two sorted files into one file without disturbing the sorting order. One of the methods implements this with internal processing after the contents of records are sorted in an array, and another method uses a service program or instructions (e.g., SORT instruction, MERGE instruction) of programming languages.

- **Control break process**

When the records of a file are sorted with key items, the control break process handles the records as one group until the value of the key items change. It is used in the process (i.e., group total process) that determines the total for each group.

2 - 5 Algorithm Design

Algorithm design refers to conducting problem analysis and creating an algorithm.

In problem analysis, type and data structure of input data/output data, relation between data items are defined, and complex process conditions are summarized in a **decision table**, or other method.

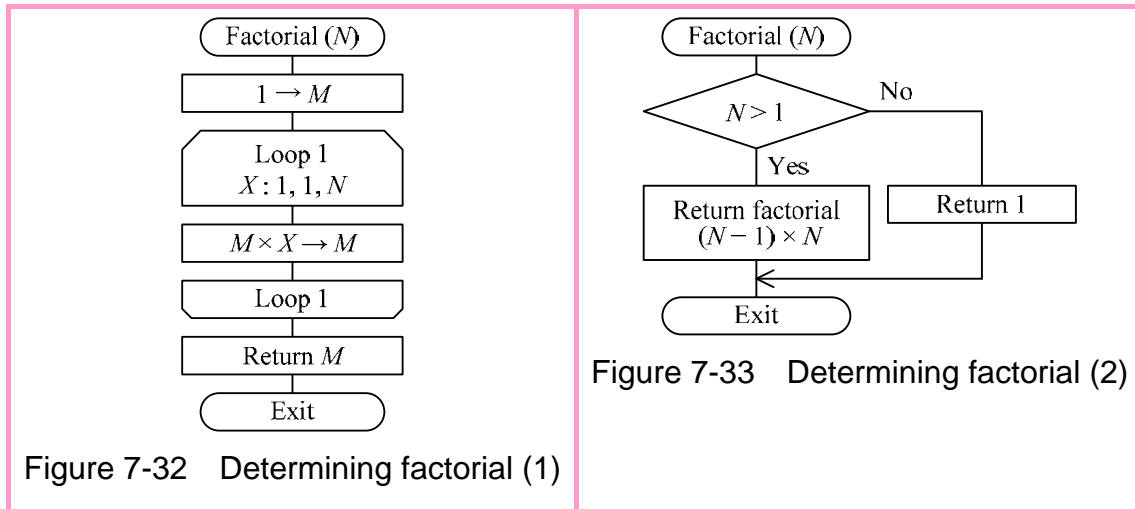
- **Decision table**

A decision table summarizes an action or process according to conditions in a table form.

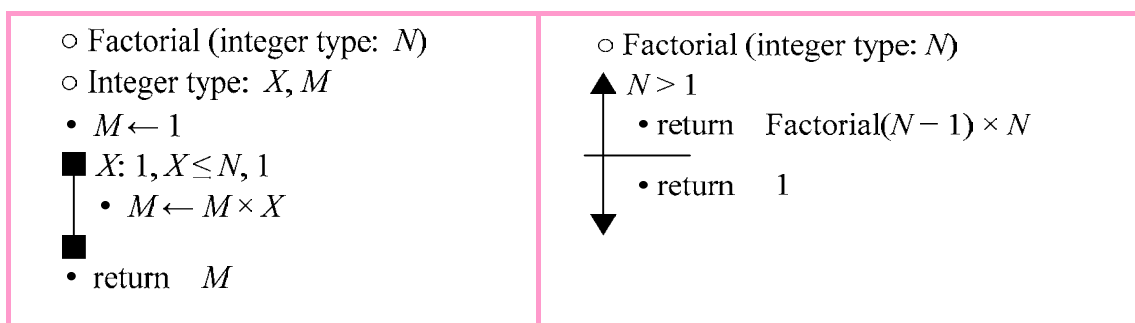
Condition title field	Condition input field (Y/N/-)	It is raining	Y	N	N
		Probability of rain is 30% or more	-	Y	N
Action title field	Action input field (X/-)	Carry long umbrella	X	-	-
		Carry folding umbrella	-	X	-
		Do not carry umbrella	-	-	X

For creating an algorithm, when the problem is large and complex, the **divide-and-conquer approach** is also used that splits the problems into smaller problems, and then, each of the smaller problems is solved. When the divide-and-conquer approach is used, the algorithm

becomes **recursive** in most of the cases (e.g., quick sort). For example, when the algorithm in Figure 7-32, which determines factorial of 1 through N ($N > 2$), is prepared on the basis of the concept of the divide-and-conquer approach (with a focus on what to do with respect to N), it becomes recursive as shown in Figure 7-33.



In addition, the algorithm in the figure above can also be represented by using **pseudo-language**.



This file is a part of the book:

Original Japanese edition published by Infotech Serve Inc.

ITワールド

(ISBN978-4-906859-06-1)

Copyright © 2013 by Infotech Serve Inc.

Translation rights arranged with Infotech Serve Inc.

Translation copyright © 2015 by Information-technology Promotion Agency, Japan

Information-technology Promotion Agency, Japan

Center Office 16F, Bunkyo Green Court, 2-28-8, Hon-Komagome, Bunkyo-ku, Tokyo,
113-6591 JAPAN
