Dedicated
to
My Grandmother
Late Smt Gauri Devi Agarwal

Contents

CHAPTER

1

Introduction to Algorithms

1.1 Introduction

A common man thinks that a computer can do anything and everything and it is very difficult to make people understand that it is not really the computer but the man behind computer who does the whole thing.

In the modern world man feels just by entering what he wants to search into the computers he can get information as desired by him. He believes that, the computer does this. A common man rarely understands that a man made procedure called **search** has done the entire job and the only support provided by the computer is the execution speed and organized storage of information.

In the above instance, a designer of the information system should know what one frequently searches for. He should make a structured organization of all those details to store in memory of the computer. Based on the requirement, the right information is brought out. This is accomplished through a set of instructions created by the designer of the information system to search the right information matching the requirement of the user. This set of instructions is termed as **program**.

1.2 Why Study Algorithms ?

As the speed of processors increases, performance is frequently said to be less central than the other software quality characteristics (*e.g.*, security, extensibility, re-usability, etc.) However, large problem sizes are common place in the area of computational science; which makes performance a very important factor. This is because longer computation time, to name a few,

mean slower results, less through research, and higher cost of computation (if buying CPU hours from an external party). The study of algorithms, therefore, gives us a language to express performance as a function of problem size.

1.3 Definitions

The name 'Algorithm' is given after the name of **Abu Ja'far Muhammad ibn Musa Al-Khwarizmi**, Ninth Century, is defined as follows :

- An algorithm is a set of rules for carrying out calculation either by hand or on a machine.
- An algorithm is a sequence of computational steps that transform the input into the output.
- An algorithm is a sequence of operations performed on data that have to be organized in data structures.
- A finite set of instruction that specify a sequence of operations to be carried out in order to solve a specific problem or class of problems is called an algorithm.
- An algorithm is an abstraction of a program to be executed on a physical machine (model of computation).

al-jabr means "restoring" which is the process of moving a subtracted quantity to the other side of an equation; al-muqabala is "comparing" and refers to subtracting equal quantities from both sides of an equation.

An algorithm must have the following properties :

- **Finiteness.** Algorithm must complete after a finite number of instructions have been executed.
- **Absence of ambiguity.** Each step must be clearly defined, having only one interpretation.
- **Definition of sequence.** Each step must have a unique defined preceding and succeeding step. The first step (start step) and last step (halt step) must be clearly noted.
- **Input/output.** Number and types of required inputs and results must be specified.
- **Feasibility.** It must be possible to perform each instruction.

Let us try to present the scenario of a man brushing his own teeth as an algorithm as follows :

Step 1. Take the brush
Step 2. Put the paste on it
Step 3. Start brushing
Step 4. Clean
Step 5. Wash
Step 6. Stop

If one goes through these 6 steps without being aware of the statement of the problem, he could possibly feel that this is the algorithm for cleaning a toilet. This is because of several ambiguities while comprehending every step. The step 1 may imply toothbrush, paintbrush, toilet brush etc. Such an ambiguity doesn't an instruction an algorithmic step. Thus every step should be made unambiguous. An unambiguous step is called '**definite instruction**'. Even if the step 2 is rewritten as apply the toothpaste, to eliminate ambiguities yet the conflicts such as, where to apply the toothpaste and where is the source of the toothpaste, need to be resolved. Hence, the act of

applying the toothpaste is not mentioned. Although unambiguous, such unrealizable steps can't be included as algorithmic instruction as they are not effective.

The definiteness and effectiveness of an instruction implies the successful termination of that instruction. However the above two may not be sufficient to guarantee the termination of the algorithm. Therefore, while designing an algorithm care should be taken to provide a proper termination for algorithm.

Characteristics of an Algorithm

Every algorithm should have the following *five* characteristic features :

1. Input 2. Output 3. Definiteness 4. Effectiveness 5. Termination

Characteristics (1) and (2) require that an algorithm produces one or more outputs and have zero or more inputs that are externally supplied.

According to characteristic (3) each operation must be definite meaning that it must be perfectly clear what should be done. Instructions such as "compute x/o" or "subtract 7 or 6 to x" are not permitted because it is not clear which of the two possibilities should be done or what the result is. That is definiteness means each instruction is clear and unambiguous. Characteristic (4) requires that each operation be *effective* that is each step must be such that it can be done by a person using pencil and paper in a finite amount of time. The (5) characteristic for algorithm is that it terminates after a finite number of operations. A related consideration is that the time for termination should be reasonably short.

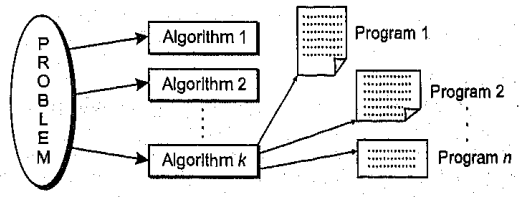
Therefore, an **algorithm can be defined as a sequence of definite and effective instructions, while terminates with the production of correct output from the given input.**

In other words, viewed little more formally, an algorithm is a step-by-step formalization of a mapping function to map input set onto an output set.

1.4 Algorithms vs. Programs

In computational theory, we distinguish between an algorithm and a program. Program does not have to satisfy the finiteness condition. For example, we can think of an operating system that continues in a "wait" loop until more jobs are entered. Such a program does not terminate unless the system crashes. Since our programs always terminate, we use "algorithm" and "program" interchangeably.

Given a problem to solve, the **design phase** produces an algorithm and the **implementation phase** then produces a program that expresses the designed algorithm. So, the concrete expression of an algorithm in a particular programming language is called a program.



Example :

Problem *Finding the largest value number among n numbers.*
Input The value of *n* and *n* numbers.
Output The largest value number.

Steps
1. Let the value of the first be the largest value denoted by BIG
2. Let R denote the number of remaining number. $R = n - 1$
3. If $R! = 0$ then it is implied that the list is still not exhausted. Therefore look the next number called NEW.
4. Now R becomes $R - 1$
5. If NEW is greater than BIG then replace BIG by the value of NEW
6. Repeat steps 3 to 5 until R becomes zero.
7. Print BIG
8. Stop

End of algorithm

1.5 Algorithm Design Techniques

For a given problem, there are many ways to design algorithms for it, (*e.g.*, Insertion sort is an incremental approach). The following is a list of several popular design approaches.

- Divide-and-Conquer (D and C)
- Greedy Approach
- Dynamic Programming
- Branch-and-Bound
- Randomized Algorithms
- Backtracking Algorithms

Divide and Conquer

- Divide the original problem into a set of sub problems
- Solve every sub problem individually, recursively
- Combine the solutions of the sub problems (top level) into a solution of the whole original problem.

Greedy approach

Greedy algorithms seek to optimize a function by making choices (greedy criterion) which are the best locally but do not look at the global problem. The result is a good solution but not necessarily the best one. The greedy algorithm does not always guarantee the optimal solution however it generally produces solutions that are very close in value to the optimal.

Dynamic Programming

Dynamic programming is a technique for efficiently computing recurrences by storing partial results. It is a method of solving problems exhibiting the properties of overlapping subproblems and optimal substructure that takes much less time than naive methods.

Branch-and-Bound

In a branch and bound algorithm a given sub problem, which cannot be bounded, has to be divided into at least two new restricted sub problems. Branch and bound algorithms are methods for global optimization in nonconvex problems. Branch and bound algorithms can be (and often are) slow, however, in the worst case they require effort that grows exponentially with problem size, but in some cases we are lucky, and the methods converge with much less effort.

Randomized Algorithms

A randomized algorithm is defined as an algorithm that is allowed to access a source of independent, unbiased random bits, and it is then allowed to use these random bits to influence its computation.

Backtracking Algorithms

Backtracking algorithms try each possibility until they find the right one. It is a depth-first search of the set of possible solutions. During the search, if an alternative doesn't work, the search backtracks to the choice point, the place which presented different alternatives, and tries the next alterative. When the alternatives are exhausted, the search returns to the previous choice point and try the next alternative there. It there are no more choice points, the search fails.

1.6 Algorithm Classification

Algorithms that use a similar problem-solving approach can be grouped together. This classification is neither exhaustive nor disjoint. Algorithm types we will consider include :

- Recursive algorithms
- Backtracking algorithms
- Divide and conquer algorithms
- Dynamic programming algorithms
- Greedy algorithms
- Branch and bound algorithms
- Brute force algorithms
- Randomized algorithms

1.6.1 Recursive Algorithms

A recursive algorithm :

- Solves the base cases directly
- Recurs with a simpler sub problem
- Does some extra work to convert the solution to the simpler sub problem into a solution to the given problem.
- *Examples* :
  - To count the number of elements in a list.
  - To test if a value occurs in a list.

1.6.2 Backtracking Algorithms

Backtracking algorithms are based on a depth-first recursive search. A backtracking algorithm :

- Tests to see if a solution has been found, and if so, returns it ; otherwise
- For each choice that can be made at this point,
  1. Make that choice 2. Recur 3. If the recursion returns a solution, return it
- If no choices remain, return failure

Example, To color a map with no more than four colors :

```
Color (Country n) :
If all countries have been colored (n > number of countries) return success ; otherwise,
    For each color c four colors,
        If country n is not adjacent to country that has been colored c
            Color country n with color c
            recursively color country n + 1
    If successful, return success
    If loop exists, return failure
```

1.6.3 Divide and Conquer

A divide and conquer algorithm consists of *two* parts :

1. Divide the problem into smaller sub problems of the same type, and solve these sub problems recursively.
2. Combine the solutions to the sub problems into a solution to the original problem.

Traditionally, an algorithm is only called "divide and conquer" if it contains at least two recursive calls. *Example*, Quick sort, Merge sort.

1.6.4 Dynamic Programming Algorithms

A dynamic programming algorithm remembers past results and uses them to find new results. Dynamic programming is generally used for optimization problems. In dynamic programming multiple solutions exist, need to find the "best" one. It "requires substructure" and "overlapping sub problems". Optimal substructure means optimal solution contains solutions to sub problems. Overlapping sub problems means solutions to sub problems can be stored and reused in a bottom-up fashion.

This differs from Divide and Conquer, where sub problems generally need not overlap.

1.6.5 Greedy Algorithm

An optimization problem is one in which we want to find, not just a solution, but the best solution. A "greedy algorithm" sometimes works well for optimization problems. A greedy algorithm works in phases :

At each phase :

- We take the best we can get right now, without regard for future consequences.
- We hope that by choosing a *local* optimum at each step, we will end up at a *global* optimum

1.6.6 Branch and Bound Algorithms

Branch and bound algorithms are generally used for optimization problem. As the algorithm progresses, a tree of subproblems is formed. The original problem is considered the "root problem".

A method is used to construct an upper and lower bound for a given problem.

- At each node, apply the bounding methods
- If the bounds match, it is deemed a feasible solution to that particular sub problem.
- If bounds do not match, partition the problem represented by that node, and make the two sub problems into children nodes.
- Continue, using the best known feasible solution to trim sections of the tree, until all nodes have been solved or trimmed.

1.6.7 Brute Force Algorithm

A brute force algorithm simply tries *all* possibilities until a satisfactory solution is found. Such an algorithm can be :

- **Optimizing.** Find the best solution. This may require finding all solution, or if a value for the best solution is known, it may stop when any best solution is found. Example : Finding the best path for a travelling salesman.
- **Satisfying.** Stop as soon as a solution is found that is *good enough*. Example, finding a travelling salesman path that is within 10% of optimal.

1.6.8 Randomized Algorithms

A randomized algorithm uses a random number at least once during the computation to make a decision

- *Example*, In Quick sort, using a random number to choose a pivot
- *Example*, Trying to factor a large number by choosing random numbers as possible divisors.

1.7 Algorithm Analysis

The analysis of an algorithm provides background information that gives us a general idea of how long an algorithm will take for a given problem set. For each algorithm considered, we will come up with an estimate of how long it will take to solve a problem that has a set of N input values. So, for example, we might determine how many comparisons a sorting algorithm does to put a list of N values into ascending order, or we might determine how many arithmetic operations it takes to multiply two matrices of size $N * N$. There are a number of algorithms that will solve a problem. Studying the analysis of algorithms gives us the tools to choose between algorithms.

The purpose of analysis of algorithms is not to give a formula that will tell us exactly how many seconds or computer cycles a particular algorithm will take. This is not useful information because we would then need to talk about the type of computer, whether it has one or many users at a time, what processor it has, how fast its clock is, whether it has a complex or reduced instruction set processor chip, and how well the compiler optimizes the executable code.

All of those will have an impact on how fast a program for an algorithm will run. To talk about analysis in those terms would mean that by moving a program to a faster computer, the algorithm would become better because it now completes its job faster. That's not true, so, we do our analysis without regard to any specific computer.

In the case of a small or simple routine it might be possible to count the exact number of operations performed as a function of N. Most of the time, however, this will not be useful. In fact, we will see that the difference between an algorithm that does $N + 5$ operations and one that does $N + 250$ operations becomes meaningless as N gets very large.

Another reason we do not try to count every operation that is performed by an algorithm is that we could fine-tune an algorithm extensively but not really make much of a difference in its overall performance. For instance, let's say that we have an algorithm that counts the number of different characters in a file. An algorithm for that might look like the following :

```
for all 256 characters do
    assign zero to the counter
end for
while there are more characters in the file do
    get the next character
    increment the counter for this character by one
end while
```

When we look at this algorithm, we see that there are 256 passes for the initialization loop. If there are N characters in the input file, there are N passes for the second loop. So the question becomes what do we count ? In a for loop, we have the initialization of the loop variable and then

for each pass of the loop, a check that the loop variable is within the bounds, the execution of the loop, and the increment of the loop variable. This means that the initialization loop does a set of 257 assignments (1 for the loop variable and 256 for the counters), 256 increments of the loop variable, and 257 checks that this variable is within the loop bounds (the extra one is when the loop stops). For the second loop, we will need to do check of the condition $N+1$ times (the +1 is for the last check when the file is empty), and we will increment N counters. The total number of operations is

◂ Increment $N+256$ ◂ Assignments 257 ◂ Conditional checks $N+258$

So, if we have 500 characters in the file, the algorithm will do a total of 1771 operations, of which 770 are associated with the initialization (43%). Now consider what happens as the value of N gets large. If we have a file with 50,000 characters, the algorithm will do a total of 100,771 operations, of which there are still only 770 associated with the initialization (less than 1% of the total work). The number of initialization operations has not changed, but they become a much smaller percentage of the total as N increases. Let's look at this way. Computer organization information shows that copying large blocks of data is as quick as an assignment. We could initialize the first 16 counters to zero and then copy this block 15 times to fill in the rest of the counters. This would mean a reduction in the initialization pass down to 33 conditional checks, 33 assignments, and 31 increments. This reduces the initialization operation to 97 from 770, a saving of 87%. When we consider this relative to the work of processing the file of 50,000 characters, we have saved less than 0.7%(100,098 vs. 100,771). Notice we could save even more time if we did all of these initializations without loops, because only 31 pure assignments would be needed, but this would only save an additional 0.7%. It's not worth the effort. We see that the importance of the initialization is small relative to the overall execution of this algorithm. In analysis terms, the cost of the initialization becomes meaningless as the number of input values increases. The earliest work in analysis of algorithms determined the computability of an algorithm on a Turing machine. The analysis would count the number of times that the transition function needed to be applied to solve the problem.

An analysis of the space needs of an algorithm would count how many cells of a Turing machine tape would be needed to solve the problem. The sort of analysis is a valid determination of the relative speed of two algorithms, but it is also time-consuming and difficult. To do this sort of analysis, you would first need to determine the process used by the transition functions of the Turing machine that carries out the algorithm. Then you would need to determine how long it executes–a very tedious process. An equally valid way to analyze an algorithm, and the one we will use, is to consider the algorithm as it is written in a higher-level language. This language can be Pascal, C, C++, Java, or a general pseudocode. The specifics don't really matter as long as the language can express the major control structures common to algorithms. This means that any language that has a looping mechanism, like a for or while, and a selection mechanism, like an if, case, or switch, will serve our needs. Because we will be concerned with just one algorithm at a time, we will rarely write more than a single function or code fragment, and so the power of many of the languages mentioned will not even come into play. For this reason, a generic pseudocode will be used in this book. Some languages use short-circuit evaluation when determining the value of a Boolean expression. This means that in the expression A and B, the term B will only be evaluated if A is true, because if A is false, the result will be false no matter what B is. Likewise, for A or B, B will not be evaluated if A is true. As we will see, counting a compound expression as one or two comparisons will not be significant. So, once we are past the basics in this chapter, we will not worry about short-circuited evaluations.

Algorithm analysis requires a set of rules to determine how operations are to be counted. There is no generally accepted set of rules for algorithm analysis. In some cases, an exact count of operations is desired ; in other cases, a general approximation is sufficient.

The rules presented that follow are typical of those intended to produce an exact count of operations.

Exact Analysis Rules

1. We assume an arbitrary time unit.
2. Execution of one of the following operations takes time 1 (one) :
   (*a*) Assignment operations (*b*) Single I/O operations
   (*c*) Single boolean operations, numeric comparisons
   (*d*) Single arithmetic operations (*e*) Function return
   (*f*) Array index operations, pointer dereferences
3. Running time of a selection statement (if, switch) is the time for the condition evaluation + the maximum of the running times for the individual clauses in the selection.
4. Loop execution time is the sum, over the number the loop is executed, of the body time + time for the loop check and update operations, + time for the loop setup. (Always assume that the loop executes the maximum number of iterations possible.)
5. Running time of a function call is 1 for setup + the time for any parameter calculations + the time required for the execution of the function body.

In other case, the analysis of an algorithm is to predict the resources that the algorithm requires, and such analysis is based on individual computational models. In the following several popular computational models are listed.

◂ **RAM.** time and space (traditional serial computers)

◂ **PRAM.** parallel time, number of processors, and read-and-write restrictions (SIMD type of parallel computers)

◂ **Message Passing Model.** communication cost (number of message), and computational cost (usually the cost for local computation is ignored) (Distributed computing, peer-to-peer networking, MIMD type of Machine)

◂ **Turing Machine.** time and space (abstract theoretical machine)

The analysis of an algorithm is to evaluate the performance of the algorithm based on the given models and metrics.

◂ **Input size**

◂ **Running time (worst-case and average-case) :** The running time of an algorithm on a particular input is the number of primitive operations or steps executed. Unless otherwise specified, we shall concentrate on finding only the worst case running time.

◂ **Order of growth.** To simplify the analysis of algorithms, we are interested in the growth rate of the running time, *i.e.*, we only consider the leading terms of a time formula. *e.g.*, the leading term is n^2 in the expression $n^2+100n+50000$.

Most algorithms are designed to work with inputs of arbitrary length. Usually the efficiency or complexity of an algorithm is stated as a function relating the input length to the number of steps (time complexity) or storage locations.

Algorithm analysis is an important part of a broader computational complexity theory, which provides theoretical estimates for the resources needed by any algorithm which solves a given computational problem. These estimates provide an insight into reasonable directions of search of efficient algorithms.

In theoretical analysis of algorithms it is common to estimate their complexity in asymptotic sense, *i.e.*, to estimate the complexity function for reasonably large length of input. Big O notation, omega notation and theta notation are used to this end. For instance, binary search is said to run an amount of steps proportional to a logarithm, or in $O(\log(n))$, colloquially "in logarithmic time". **Usually asymptotic estimates are used because different implementations of the same algorithm may differ in efficiency.** However the efficiencies of any two "reasonable" implementations of a given algorithm are related by a constant multiplicative factor called hidden constant.

Exact (not asymptotic) measures of efficiency can sometimes be computed but they usually require certain assumptions concerning the particular implementation of the algorithm, called **model of computation.** A model of computation may be defined in terms of an abstract computer, *e.g.*, Turing machine, and/or by postulating that certain operations are executed in unit time. For example, if the sorted set to which we apply binary search N elements, and we can guarantee that a single binary lookup can be done in unit time, then at most $\log_2 N+1$ time units are needed to return an answer.

Exact measures of efficiency are useful to the people who actually implement and use algorithms, because they are more precise and thus enable them to know how much time they can expect to spend in execution. To some people (*e.g.*, game programmers), a hidden constant can make all the difference between success and failure.

Time efficiency estimates depend on what we define to be a step. For the analysis to make sense, the time required to perform as step must be guaranteed to be bounded above by a constant. One must be careful here ; for instance, some analysis count and addition of two numbers as a step. This assumption may not be warranted in certain contexts. For example, if the numbers involved in a computation may be arbitrarily large, addition no longer can be assumed to require constant time (compare the time we need to add two 2-digit integers and two 1000-digits integers using a pen and paper).

1.8 Formal and Informal Algorithm Analysis

There are a number of standard criteria which can be used to evaluate any algorithm, which can be divided into formal and informal techniques. A formal criterion is one which yields a result which can be expressed in a logical or arithmetic statement, an informal criterion is one which yields a less precisely stated result.

The criteria, which are about to be introduced, can be used to evaluate two or more different possible algorithms in order to choose which one should actually be implemented for a particular requirement. The techniques can also be used when a proposed optimization of an existing implementation is considered. Some of the most useful informal criteria which can be used include the following.

Effectiveness is the most fundamental criterion and is concerned with an algorithm fulfilling its specification. As it is sometimes difficult to determine if an algorithm is always effective for all possible sets of input data, there are formal techniques which can be used in an attempt to determine this.

Termination is strictly an attribute of effectiveness but is sufficiently for it to be considered as a distinct criterion. For an algorithm to be useful it must at some stage terminate and it is again not always possible to be certain of this for all possible sets of input data.

Generality. The definition of any algorithm should specify the input data which it will require and the output data which it will produce. An algorithm which will accept a greater range of input data and/or produce a greater range of output data is more general and likely to be a more useful algorithm, as it may be usable in a wider range of situations without requiring amendments.

Efficiency is sometimes taken as the most important criterion and is not always adequately specified. Possible sub-criteria include speed of execution, amount of main or secondary storage required or amount of complexity of maintenance. The basic rule for considering efficiency is that it is not possible to optimize an algorithm against any efficiency sub-criterion until it has first been shown to be effective.

Elegance. This is the most subjective aspect of informal algorithm analysis. An elegant algorithm is one which is both simple and ingenious; of these two factors simplicity is far more important than ingenuity.

1.9 How to Calculate Running Time of an Algorithm ?

We can calculate the running time of an algorithm reliably by running the implementation of the algorithm on a computer. Alternatively we can calculate the running time by using a technique called algorithm analysis. We can estimate an algorithm's performance by counting the number of basic operations required by the algorithm to process an input of a certain size.

Basic Operation. The time to complete a basic operation does not depend on the particular values of its operands. So it takes a constant amount of time.

Examples, Arithmetic operation (addition, subtraction, multiplication, division), Boolean operation (AND, OR, NOT), Comparison operation, Module operation, Branch operation etc.

Input Size. It is the number of input processed by the algorithm.

Example, For sorting algorithm the input size is measured by the number of records to be sorted.

Growth Rate. The growth rate of an algorithm is the rate at which the running time (cost) of the algorithm grows as the size of the input grows. The growth rate has a tremendous effect on the resources consumed by the algorithm.

1.10 Loop Invariants

This is a justification technique. We use loop Invariants to help us understand why an algorithm is correct. To prove some statement S about a loop is correct, define S in terms of a series of smaller statements $S_0, S_1, \ldots S_k$ where,

(*i*) The initial claim, so is true before the loop begins.

(*ii*) If S_{i-1} is true before iteration i begins, then one can show that S_i will be true after iteration i is over.

(*iii*) The final statement S_k, implies the statement S that we wish to justify as being true.

Example. *Consider the following search problem :*

Input : *A sequence $A = \langle a_1, a_2 \ldots, a_n \rangle$ and a value of v.*

Output : *An index i such that $v = A[i]$ or the special value Nil if v does not appear in A.*

Write pseudocode for the linear search, which scans through the sequence, looking for v. Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulfills the three conditions.

Solution. Below is a sample algorithm to solve this linear search problem :

```
1.  i ←1
2.  while (i <=n) do
3.      If (v == A[i])
4.          return i
5.      else i <i + 1
6.  return Nil
```

We now show that either the code returns with the index of v when $v \in A[1..n]$; or the following loop invariant holds at the beginning of the while loop :

$$v \notin A[1..i-1]$$

At the beginning of the very first loop, $i = 1$. There is no way for the code to return, but the invariant is true, since $A[1..0]$ is empty.

Assume that the code has yet to return, and the loop invariant holds at the beginning of the loop when $i = i_0$ *i.e.*, $v \notin A[1..i_0 - 1]$. Once in the loop, if $v = A[i] (= A[i_0])$, then line 4 returns with the value of i_0 ; otherwise line 5 increments i to $i_0 + 1$, which is the value of i in the next loop. Hence, either the code returns with the index of v ; or at the beginning of the next loop, we have $v \notin A[1..i_0]$, *i.e.*, $v \notin A[1..i-1]$. Thus, the maintenance case holds.

If the code does not return for any $i \leq n$, the code continues till $i = n+1$ when the invariant still holds at the beginning of that loop. Thus, $v \notin A[1..n]$ since $i = n+1$.

Exercise

1. Why do we study algorithms ?
2. What is the main difference between algorithms and programs ?
3. Define the main characteristics of an algorithm.
4. Define different design approaches of an algorithm.
5. What do you mean by analysis of an algorithm ?
6. How would you calculate running time of an algorithm ?

CHAPTER 2

Growth of Functions

The key idea that supports most of the theory of algorithms is the method of quantifying the execution time of an algorithm. Execution time depends on many parameters which are independent of the particular algorithm : machine clock rate, quality of code produced by compiler, whether or not the computer is multi-programmed etc.

The execution time of an algorithm is a function of the values of its input parameters. If we know this function (at least, to within a scalar multiple determined by the machine parameters) then we can predict the time the algorithm will require without running it. (We might not wish to run it until we knew that it would complete in a reasonable time). Unfortunately this execution time function measure is generally very complicated, too complicated to know exactly. But now a number of observations come to the release.

The first is that is very often possible to get meaningful predictive results by concentrating on one parameter alone : the total size of the input (rather than the accurate values of each input parameter). The second is that execution time functions are often the sum of a large number of terms and many of these terms make an insignificant contribution to function values : in mathematical terms, it is asymptotic form of the function that matters rather than the precise form. The final observation is that, in practice, we wish only to know either an upper bound on the execution time (to be assured that it will perform acceptably) or a lower bound (possibly to encourage us to develop a better algorithm). To sum up these observations : we measure the execution time of an algorithm as a function $T(n)$ of a single quantity n (the size of the entire input).

2.1 Complexity of Algorithms

It is very convenient to classify algorithms based on the relative amount of time or relative amount of space they require and specify the growth of time/space requirements as a function of the input size.

1. **Time Complexity.** Running time of the program as a function of the size of input.

2. **Space Complexity.** Most of what we will be discussing is going to be how efficient various algorithms are in terms of time, but some forms of analysis could be done based on how much space an algorithm needs to complete its task. This space complexity analysis was critical in the early days of computing when storage space on a computer (both internal and external) was limited. When considering space complexity, algorithms are divided into those that need extra space to do their work and those that work in place. It was not unusual for programmers to choose an algorithm that was slower just because it worked in place, because there was not enough extra memory for a faster algorithm.

Computer memory was at a premium ; so another form of space analysis would examine all of the data being stored to see if there were more efficient ways to store it. For example, suppose we are storing a real number that has only one place of precision after the decimal point and ranges between – 10 and +10. If we store this as a real number, most computers will use between 4 and 8 bytes of memory, but if we first multiply the value by 10, we can then store this as an integer between – 100 and +100. This needs only 1 byte, a saving of 3 to 7 bytes. A program that stores 1000 of these values can save 3000 to 7000 bytes. When you consider that computers as recently as the early 1980s might have only had 65,536 bytes of memory, these savings are significant. It is this need to save space on these computers along with the longevity of working computer programs that lead to all of the Y2K bug problems. When you have a program that works with a lot of dates, you use half the space for the year by storing it as 99 instead of 1999. Also, people writing programs in the 1980s and earlier never really expected their programs to still be in use in 2000.

Looking at software that is on the market today, it is easy to see that space analysis is not being done. Programs, even simple ones, regularly quote space needs in a number of megabytes. Software companies seem to feel that making their software space efficient is not a consideration because customers who don't have enough computer memory can just go out and buy another 32 megabytes (or more) of memory to run the program or a bigger hard disk to store it. This attitude drives computers into obsolescence long before they really are obsolete.

A recent change to this is the popularity of personal digital assistants (PDAs). These small handheld devices typically have between 2 and 8 megabytes for both their data and software. In this case, developing small programs that store data compactly is not only important, it is critical.

2.2 RAM Machine

In computing time complexity, one good approach is to count primitive operations. This approach of simply counting primitive operations gives rise to a computational model called the Random Access Machine (RAM). Primitive operations are simply :

- Basic computations performed by an algorithm
- Identifiable in pseudo code

- Largely independent from the programming language
- Exact definition not important (we will see why later)
- Assumed to take a constant amount of time in the RAM model

Some examples of primitive operations are :

- Assigning a value to a variable
- Calling a method
- Performing an arithmetic operation
- Comparing two numbers
- Indexing into an array
- Following an object reference
- Returning from a method

RAM model views a computer simply as a CPU connected to a bank of memory cells. Each memory cell stores a word, which can be a number, a character string, or an address *i.e.*, the value of a base type. The term "random access" refers to the ability of the CPU to access an arbitary memory cell with one primitive operation. To keep the model simple, we do not place any specific limits on the size of numbers that can be stored in words of memory. We assume the CPU in the RAM model can perform any primitive operation in a constant number of steps, which do not depend on the size of the input. Thus, an accurate bound on the number of primitive operations an algorithm performs corresponds directly to the running time of that algorithm in the RAM model. By inspecting the pseudo code, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size.

2.3 Best, Worst and Average - Case Complexity

Using the random-access machine (RAM) model of computation, we can count how many steps our algorithm will take on any given input instance by simply executing it on the given input. In the RAM model, instructions are executed one after another, with no concurrent operations. However, to really understand how good or bad an algorithm is, we must know how it works over *all* instances.

To understand the notions of the best, worst and average-case complexity, one must think about running an algorithm on all possible instances of data that can be fed to it. The best, worst and average cases of a given algorithm express what the resource usage is *at least, at most* and *on average*, respectively. Usually the resource being considered is running time, but it could also be memory or other resource. The *worst-case complexity* of the algorithm is the function defined by the maximum number of steps taken on any instance of size *n*. The *best-case complexity* of the algorithm is the function defined by the minimum number of steps taken on any instance of size *n*. Finally, the *average-case complexity* of the algorithm is the function defined by the average number of steps taken on any instance of size *n*.

Worst-case Running Time. The behaviour of the algorithm with respect to the worst possible case of the input instance. The worst-case running time of an algorithm is an upper bound on the running time for any input. Knowing it gives us a guarantee that the algorithm will never take any longer time.

Average case Running Time. The expected behaviour when the input is randomly drawn from a given distribution. The average-case running time of an algorithm is an estimate of the running time for an "average" input. Computation of a average-case running time

entails knowing all possible input sequences, the probability distribution of occurrence of these sequences, and the running times for the individual sequences.
Often it is assumed that all inputs of a given size are equally likely.

Amortized Running Time. Here the time required to perform a sequence of (related) operations is averaged over all the operations performed. Amortized analysis can be used to show that the average cost of an operation is small, if one averages over a sequence of operations, even though a simple operation might be expensive. Amortized analysis guarantees the average performance of each operation in the worst case.

Worst-case analysis and average-case analysis

Worst case performance analysis and average case performance analysis have similarities, but usually require different tools and approaches in practice. Determining what average input means is difficult, and often that average input has properties which make it difficult to characterise mathematically (consider, for instance, algorithms that are designed to operate on strings of text). Similarly, even when a sensible description of a particular "average case" is possible, they tend to result in more difficult to analyse equations.

Worst case analysis has similar problems, typically it is impossible to determine the exact worst case scenario. Instead, a scenario is considered which is at least as bad as the worst case. For example, when analysing an algorithm, it may be possible to find the longest possible path through the algorithm (by considering maximum number of loops, for instance) even if it is not possible to determine the exact input that could generate this. Indeed, such an it is input may not exist. This leads to a safe analysis (the worst case is never underestimated), but which is *pessimistic*, since no input might require this path.

Alternatively, a scenario which is thought to be close to (but not necessarily worse than) the real worst case may be considered. This may lead to an *optimistic* result, meaning that the analysis may actually underestimate the true worst case.

In some situations it may be necessary to use a pessimistic analysis in order to guarantee safety. Often, however, a pessimistic analysis may be too pessimistic, so an analysis that gets closer to the real value but may be optimistic (perhaps with some known low probability of failure) can be a much more practical approach.

When analyzing algorithms which often take a small time to complete, but periodically require a much larger time, amortized analysis can be used to determine the worst case running time over a (possible infinite) series of operations. This amortized worst case cost can be much closer to the average case cost, while still providing a guaranteed upper limit on the running time.

The other thing that has to be decided when making these considerations is whether what is of interest is the *average* performance of a program, or whether it is important to guarantee that even in the worst case, performance obeys certain rules. In many every-day applications, the average case will be the more important one, and saving time there may be more important than guaranteeing good behaviour in the worst case. On the other hand, when considering time-critical problems (think of a program that keeps track of the planes in a certain sector), it may be totally unacceptable for the software to take very long if the worst comes to the worst.

Again, algorithms often trade efficiency of the average case versus efficiency of the worst case *i.e.*, the most efficient program on average might have a particularly bad worst case. We have seen concrete examples for these when we considered algorithms for searching.

For example,

- Consider the problem of finding the minimum in a list of elements.
  Worst case = $O(n)$; Average case = $O(n)$
- Quick sort
  Worst case = $O(n^2)$; Average case = $O(n \log n)$
- Merge sort, Heap sort
  Worst case = $O(n^2)$; Average case = $O(n^2)$
- Binary search tree : Search for an element
  Worst case = $O(n)$; Average case = $O(\log n)$

2.4 Growth Rate of Functions

Resources for an algorithm are usually expressed as a function of input. Often this function is messy and difficult to work. To study function growth easily, we reduce the function down to the important part.

Let $f(n) = an^2 + bn + c$.

In this function, the n^2 term dominates the function, that is when n gets sufficiently large, the other terms bare factor into the result.

Dominate terms are what we are interested in to reduce a function, in this we ignore all constants and coefficients and look at the highest order term in relation to n.

2.5 Asymptotic Analysis

Asymptotic means a line that tends to converge to a curve, which may or may not eventually touch the curve. It is a line that stays within bounds.

Asymptotic notation is a shorthand way to write down and talk about 'fastest possible' and 'slowest possible' running times for an algorithm, using high and low bounds on speed. These are also referred to as 'best case' and 'worst case' scenarios respectively.

2.5.1 Why are Asymptotic Notations Important ?

1. They give a simple characterization of an algorithm's efficiency.
2. They allow the comparison of the performances of various algorithms.

2.5.2 Asymptotic Notations

1. Big-oh notation. Big-oh is the formal method of expressing the upper bound of an algorithm's running time. It is the measure of the longest amount of time it could possibly take for the algorithm to complete. More formally, for non-negative functions, $f(n)$ and $g(n)$, if there exists an integer n_0 and a constant $c > 0$ such that for all integers $n > n_0$.

$$f(n) \leq cg(n)$$

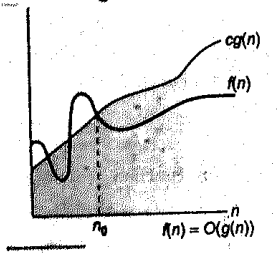


Figure 2.1

Then $f(n)$ is Big-oh of $g(n)$. This is denoted as

"$f(n) \in O(g(n))$"

i.e., the set of functions which, as n gets large, grow no faster than a constant times $f(n)$.

2. Big-omega notation. For non-negative functions, $f(n)$ and $g(n)$, if there exists an integer n_0 and a constant $c > 0$ such that for all integers $n > n_0$, $f(n) \geq cg(n)$ then $f(n)$ is big omega of $g(n)$. This is denoted as

"$f(n) \in \Omega(g(n))$"

This is almost the same definition as Big-oh, except that "$f(n) \geq (g(n))$", this makes $g(n)$ a lower bound function instead of an upper bound function. It describes the best that can happen for a given data size.

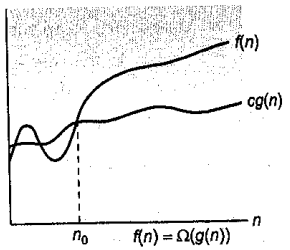


Figure 2.2

3. Theta notation. The lower and upper bound for the function 'f' is provided by the theta notation (θ). For non-negative functions $f(n)$ and $g(n)$, if there exists an integer n_0 and positive constants c_1 and c_2 *i.e.,* $c_1 > 0$ and $c_2 > 0$ such that for all integers $n > n_0$, $c_1 g(n) \leq f(n) \leq c_2 g(n)$ then $f(n)$ is theta of $g(n)$. This is denoted as "$f(n) \in \theta(g)$" we mean "f is order g".

Theorem

If $f(n) = a_m n^m + a_{m-1} n^{m-1} + \ldots a_1 n + a_0$ *and* $a_m > 0$, *then* $f(n) = O(n^m)$

Proof.
$$f(n) \leq \sum_{k=0}^{m} |a_k| n^k$$
$$\leq n^m \sum_{k=0}^{m} |a_k| n^{k-m} \leq n^m \sum_{k=0}^{m} |a_k| \text{ for } n \geq 1$$

So $f(n) = O(n^m)$.

Properties. Let $f(n)$ and $g(n)$ be such that $\lim_{n \to \infty} \frac{f(n)}{g(n)}$ exists, then

(*i*) Function $f \in O(g)$ *i.e.,* $f(n) \in O(g(n))$ if

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = c < \infty,$$ also including the case in which limit is 0.

(*ii*) Function $f(n) \in \Omega(g(n))$ if

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} > 0,$$ including the case in which limit is ∞.

(*iii*) Function $f(n) \in \theta(g(n))$ if

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = c$$ for some constant c such that $0 < c < \infty$.

4. Little-oh notation (o). Asymptotic upper bound provided by O-notation may not be asymptotically tight. So o-notation is used to denote an upper bound that is asymptotically tight.

$o(g(n)) = \{ f(n)$: for any +ve constant $c > 0$, if a constant $n_0 > 0$ such that $0 \leq f(n) < cg(n) \ \forall \ n \geq n_0 \}$

The function $f(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity, *i.e.,*

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

For example, $2n = o(n^2)$

$2n^2 \neq o(n^2)$

5. Little-Omega Notation. As o-notation is to O-notation, we have ω-notation as Ω-notation. Little-omega (ω) is used to denote an lower bound that is asymptotically tight,

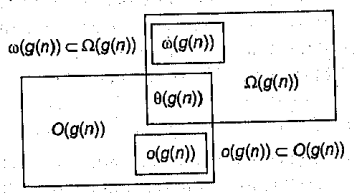
$\omega(g(n)) = \{ f(n)$: for any +ve constant $c > 0$, if a constant $n_0 > 0$ such that $0 \leq cg(n) < f(n) \}$

Here $\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$.

Thus asymptotic notation gives us a way to express their relationship.

- If $f(n)$ is $O(g(n))$ *i.e.,* $f(n)$ grows no faster than $g(n)$
- If $f(n)$ is $o(g(n))$ *i.e.,* $f(n)$ grows slower than $g(n)$
- If $f(n)$ is $\omega(g(n))$ *i.e.,* $f(n)$ grows faster than $g(n)$.
- If $f(n)$ is $\Omega(g(n))$ *i.e.,* $f(n)$ grows no slower than $g(n)$.
- If $f(n)$ is $\theta(g(n))$ *i.e.,* $f(n)$ and $g(n)$ grow at the same rate.

Relationships between O, o, θ, Ω, ω notations



θ is a subset of Ω and of O.

(*i*) $\omega(g)$ union $\theta(g)$ is a subset of $\Omega(g)$

i.e., $\omega(g) \cup \theta(g) \subset \Omega(g)$

(*ii*) $o(g)$ union $\theta(g)$ is a subset of $O(g)$.

i.e., $o(g) \cup \theta(g) \subset O(g)$

(*iii*) $\theta(g)$ is the intersection of $O(g)$ and $\Omega(g)$.

i.e., $\theta(g) = \Omega(g) \cap O(g)$

In other words

$$f(n)=\theta(g(n)) \text{ iff } f(n)=o(g(n)) \text{ and } f(n)=\Omega(g(n))$$

Figure 2.2A shows how the various measures of complexity compare with one another. The horizontal axis represents the size of the problem–for example, the number of records to process in a search algorithm. The vertical axis represents the computational effort required by algorithms of each class. This is not indicative of the running time or the CPU cycles consumed ; it merely gives an indication of how the computational resources will increase as the size of the problem to be solved increases.

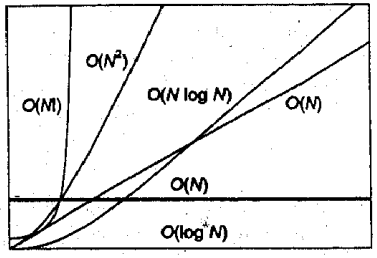


Figure 2.2A

Referring back at the list, you may have noticed that none of the orders contain constants. That is, if an algorithm's expected runtime performance is proportional to N, $2\times N$, $3\times N$, or even $100\times N$, in all cases the complexity is defined as being $O(N)$. This may seem a little strange at first–surely $2\times N$ is better than $100\times N$–but as mentioned earlier, the aim is not to determine the exact number of operations but rather to provide a means of comparing different algorithms for relative efficiency. In other words, an algorithm that runs in $O(N)$ time will generally outperform another algorithm that runs in $O(N^2)$. Moreover, when dealing with large values of N, constants make less of a difference : As a ratio of the overall size, the difference between 1,000,000,000 and 20,000,000,000 is almost insignificant even though one is actually 20 times bigger.

Of course, at some point you will want to compare the actual performance of different algorithms, especially if one takes 20 minutes to run and the other 3 hours, even if both are $O(N)$. The thing to remember, however, is that it's usually much easier to halve the time of an algorithm that is $O(N)$ than it is to change an algorithm that's inherently $O(N^2)$ to one that is $O(N)$.

Why not use θ notation all the time ?

At the first glance, it would appear that θ is all we really want or need-after all, it is the right answer. Where the order of $g(n)$ is exactly $f(n)$. That is, if we know the real order of $f(n)$, then we know $\theta(g(n))$.

Unfortunately, it is not so simple – there are functions that do not have a θ at all. For example, consider a function $f(n)$ that is $O(n)$ when n is even and $O(1)$ if n is odd. Therefore

$$f(n)=O(n) \quad \text{but} \quad f(n)=\Omega(1).$$

Helpful Hints

1. Not every pair of functions is comparable.
2. It may be easier to test for $o(g)$ and $\omega(g)$. Try these first and then try O, Ω, and θ.
3. Sometimes we can deduce several relationships from the knowledge of only one. For example, if a function is $o(g)$ it is also $O(g)$, but never $\theta(g)$, $\Omega(g)$ or $\omega(g)$.
4. When is doubt, graph the functions.

Example 1. *Prove* $n!=o(n^n)$

Solution. We know $n!=n(n-1)(n-2)\ldots2\cdot1$

$$=n^n\left[1-\frac{1}{n}\right]\ldots\frac{2}{n}\cdot\frac{1}{n}$$

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=\lim_{n\to\infty}\frac{n^n\left[1-\frac{1}{n}\right]\ldots\frac{2}{n}\cdot\frac{1}{n}}{n^n}=0$$

$\therefore\ n!=o(n^n)$.

Example 2. *Prove* $n!=w(2^n)$

Solution. $\lim_{n\to\infty}\frac{f(n)}{g(n)}=\lim_{n\to\infty}\frac{n^n\left[1-\frac{1}{n}\right]\left[1-\frac{2}{n}\right]\ldots\frac{2}{n}\cdot\frac{1}{n}}{2^n}=\frac{\infty}{2^\infty}=\infty.$

Hence $n!=w(2^n)$.

Example 3. *Prove that* $f(n)=\log_2 n$ *is* $O(n^\alpha)$ *for any* $\alpha>0$.

Solution. By definition $f(n)\in O(n)$ if

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=0$$

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=\lim_{n\to\infty}\frac{\log_2 n}{n^\alpha}=\lim_{n\to\infty}\frac{\lg n}{\lg 2\cdot n^\alpha}$$

i.e.,

$$\frac{f(n)}{g(n)}=\frac{\lg n}{\lg 2\cdot n^\alpha}$$

$$\frac{f'(n)}{g'(n)}=\frac{\frac{1}{n}}{\lg 2\cdot\alpha n^{\alpha-1}}=\frac{n}{n\alpha\lg 2n^\alpha}=\frac{1}{\alpha\lg 2\cdot n^\alpha}.$$

By L's Hospital Rule,

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=\lim_{n\to\infty}\frac{f'(n)}{g'(n)}=\lim_{n\to\infty}\frac{1}{\alpha\lg 2\cdot n^\alpha}=\frac{1}{\infty}=0.$$

$\therefore\ f(n)\in O(n^\alpha)$.

2.5.3 Asymptotic Notation Properties

1. Reflexivity
2. Symmetry
3. Transpose Symmetry
4. Transitivity

1. Reflexivity

$$f(n)=\theta(f(n))$$
$$f(n)=O(f(n))$$
$$f(n)=\Omega(f(n))$$

2. Symmetry

$f(n)=\theta(f(n))$ if and only if $g(n)=\theta(f(n))$

3. Transpose Symmetry

$f(n)=O(f(n))$ if and only if $g(n)=\Omega(f(n))$
$f(n)=o(g(n))$ if and only if $g(n)=\omega(f(n))$

4. Transitivity

$f(n)=\theta(g(n))$ and $g(n)=\theta(h(n))$ imply $f(n)=\theta(h(n))$
$f(n)=O(g(n))$ and $g(n)=O(h(n))$ imply $f(n)=O(h(n))$
$f(n)=\Omega(g(n))$ and $g(n)=\Omega(h(n))$ imply $f(n)=\Omega(h(n))$
$f(n)=o(g(n))$ and $g(n)=o(h(n))$ imply $f(n)=o(h(n))$
$f(n)=\omega(g(n))$ and $g(n)=\omega(h(n))$ imply $f(n)=\omega(h(n))$

Manipulating asymptotic notations :

1. $cO(f(n))=O(f(n))$
2. $O(O(f(n)))=O(f(n))$
3. $O(f(n)g(n))=f(n)O(g(n))$
4. $O(f(n))O(g(n))=O(f(n)g(n))$
5. $O(f(n)+g(n))=O(\max(f(n), g(n)))$
6. $\sum_{i=1}^{n} O(f(i))=O\left(\sum_{i=1}^{n} f(i)\right)$

2.5.4 Asymptotic Notations in equations

(*i*) $n=O(n^2)$
It generally means $n \in$ in the set of $O(n^2)$

(*ii*) $2n^2+6n+5=2n^2+\theta(n)$
It means, we do not care about details of $\theta(n)$
It means $\theta(n)$ is some function $f(n) \in \theta(n)$
$f(n)$ is an anonymous function.

(*iii*) If asymptotic notation appears on the left
i.e., $2n^2+\theta(n)=\theta(n^2)$
We can always choose a right hand side to satisfy equality.

(*iv*) $2n^2+16n+2=2n^2+\theta(n)=\theta(n^2)$

2.6 Analysing Algorithm Control Structures

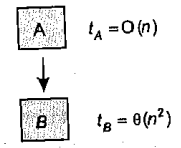
The analysis of an algorithm is calculated by considering its individual instructions. The individual instructions are calculated and then according to the control structures, we combine these times. Some algorithm control structures are :

1. Sequencing
2. If-then-else
3. "for" loop
4. "while" loop
5. Recursion

1. Sequencing

Suppose our algorithm consists of two parts A and B. A takes time t_A and B takes t_B time for computation. The total computaion "t_A+t_B" is according to the sequencing rule. According to maximum rule this computation time is (max (t_A, t_B)).

Example. Suppose $t_A=O(n)$ and $t_B=\theta(n^2)$. Calculate the total computation time



Computation time $=t_A+t_B$
$=(\max(t_A, t_B))$
$=(\max(O(n), \theta(n^2))=\theta(n^2)$

2. If-then-else

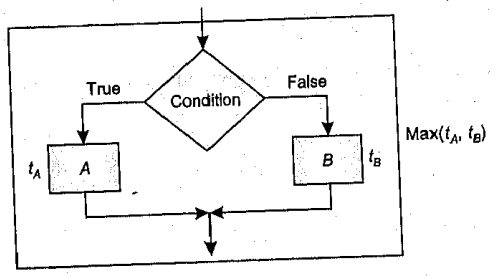


Figure 2.3

The total computation time is according to the conditional rule - "if-then-else". According to the maximum rule this computation time is max $(t_A\ t_B)$.

Example, Suppose $t_A = O(n^2)$ and $t_B = \theta(n^2)$. Calculate the total computation time for the following :

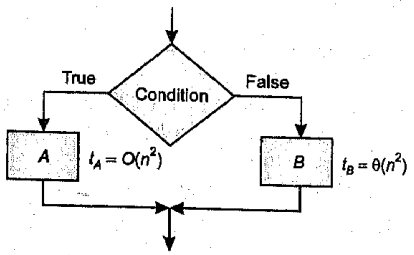


Figure 2.4

Total computation $= \max(t_A, t_B)$

$= \max(O(n^2), \theta(n^2)) = \theta(n^2)$

3. "For" loop

Consider the following loop.

```
for i ← 1 to m.
{
   P(i)
}
```

If the computation time t_i for $P(i)$ various as a function of "i", then total computation time for the loop is given not by a multiplication but by a sum. *i.e.,*

```
for i ← 1 to m.
{
   P(i)
}
```

takes $\sum_{i=1}^{m} t_i$ time. *i.e.,* $\sum_{i=1}^{m} \theta(1) = \theta \sum_{i=1}^{m} 1 = \theta(m)$.

If the algorithm consists of nested "for" loops then the total computation time is

```
for i ← 1 to m.
{
   for j ← 1 to m.
   {
      P(ij)
   }
}
```

$\sum_{i=1}^{m} \sum_{j=1}^{m} t_{ij}$

Example. *Consider following 'for' loops, calculate the total computation time for the following*

```
for i ← 2 to m − 1.
{
   for j ← 3 to i.
   {
      sum ← sum + A [i] [j]
   }
}
```

Solution. The total computation time is

$$\sum_{i=2}^{m-1} \sum_{j=3}^{i} t_{ij} = \sum_{i=2}^{m-1} \sum_{j=3}^{i} \theta(1)$$

$$= \sum_{i=2}^{m-1} \theta(i)$$

$$= \theta\left(\sum_{i=2}^{m-1} i\right) = \theta(m^2/2 + \theta(m))$$

$$= \theta(m^2).$$

4. "while" loop

In this, there is no obvious method which determines how many times we shall have to repeat the loop. The simple technique for analysing the loops is to firstly determine function of variables involved whose value decreases each time around. Secondly for terminating the loop, it is necessary that this value must be a positive integer. By keeping the track of how many times the value of function decreases, one can obtain the number of repetition of the loop. The other approach for analysing "while" loops is to treat them as recursive algorithms.

Example. *The running time of algorithm array Max for computing the maximum element in an array of n integers is $O(n)$.*

Solution. array Max (A, n).

```
1.  current max ← A[0].
2.  for i ← 1 to n − 1
3.      do if current max < A[i]
4.          then current max ← A[i]
5.  return current max.
```

The number of primitive operations $t(n)$ executed by this algorithm is atleast

$$2 + 1 + n + 4(n-1) + 1 = 5n$$

and at most $\quad 2 + 1 + n + 6(n-1) + 1 = 7n - 2.$

The best case $t(n)=5n$ occurs when $A[o]$ is the maximum element. The worst case $t(n)=7n-2$ occurs when the elements are sorted in increasing order.

We may therefore apply the big-oh definition with $c=7$ and $n_0=1$ and conclude that running time of this is $O(n)$.

Example. *Let $f(n)$ and $g(n)$ be asymptotically non-negative functions. Using the basic definition of θ-notation prove that*

$$max(f(n), g(n))=\theta(f(n)+g(n))$$

Solution. This is true if there exists $c_1, c_2, n_0>0$ such that

$$0\le c_1(f(n)+g(n))\le \max(f(n), g(n))\le c_2(f(n)+g(n)) \quad \text{for all } n\ge n_0.$$

◄ Choose for instance $c_1=\frac{1}{2}$ ⇒ two situations.

1. $f(n)>g(n) \Rightarrow \frac{1}{2}(f(n)+g(n))<\frac{1}{2}(f(n)+f(n))=f(n)=\max(f(n),g(n))$

2. $f(n)<g(n) \Rightarrow \frac{1}{2}(f(n)+g(n))<\frac{1}{2}(g(n)+g(n))=g(n)=\max(f(n),g(n))$

◄ Choose for instance $c_2=1$

$$\max(f(n), g(n))\le 1(f(n)+g(n))=f(n)+g(n)$$

$$\max(f(n), g(n))\le f(n)+g(n)$$

So, choose $c_1=\frac{1}{2}$, $c_2=1$ and the statement holds.

Actually, any $c_1\le\frac{1}{2}$ and $c_2\ge\frac{1}{2}$ works.

Example. *"Explain why the statement "The running time of algorithm A is at least $O(n^2)$", is meaningless".*

Solution. "The running time of algorithm. A is atleast $O(n^2)$" is like saying that $T(n)$ is asymptotically greater than or equal to all those functions that are $O(n^2)$.

O-notation is meant to indicate the upper bound.

Therefore, $O(n^2)$ by itself means "at most $c\cdot n^2$. So the statement "The running time of algorithm A is at least at most $c\cdot n^2$ does not make sense.

Example. *Prove that $(n+a)^b=\theta(n^b)$, $b>0$*

Solution. It means, we have to show that

$$c_1 n^b\le(n+a)^b\le c_2 n^b.$$

$$(n+a)^b=(n+a)(n+a)(n+a)\ldots(n+a) \text{ } b \text{ times}$$

$$=n^b+k_1 n^{b-1}+k_2 n^{b-2}+\ldots+k_{b-1}n^1+k_b$$

$$c_1 n^b\le n^b+k_1 n^{b-1}+k_2 n^{b-2}+\ldots+k_{b-1}n^1+k_b\le c_2 n^b.$$

Now, $$c_1 n^b\le n^b+k_1 n^{b-1}+k_2 n^{b-2}+\ldots+k_{b-1}n^1+k_b$$

$$c_1\le 1+\frac{k_1}{n}+\frac{k_2}{n^2}+\ldots+\frac{k_b}{n^b}.$$

It is possible to find c_1 small enough such that for particular values of $k_1, k_2, \ldots k_{b-1}, k_b$ even if values are negative for $n\ge n_0$.

i.e., $$(n+a)^b=\Omega(n^b) \qquad \ldots(1)$$

Now, $$n^b+k_1 n^{b-1}+k_2 n^{b-2}+\ldots k_{b-1}n^1+k_b\le n^b+k_1 n^b+k_2 n^b+\ldots k_{b-1}n^b+k_b n^b=kn^b$$

i.e., $$n^b+k_1 n^{b-1}+k_2 n^{b-2}+\ldots+k_b\le c_2 n^b \quad \forall\ n\ge n_0.$$

$$(n+a)^b=O(n^b) \qquad \ldots(2)$$

From (1) and (2), we get

$$(n+a)^b=\theta(n^b)$$

Example. *"Is $2^{n+1}=O(2^n)$? Is $2^{2n}=O(2^n)$?"*

Solution. Is $2^{n+1}=O(2^n)$ *i.e.,* $2^{n+1}\le c\cdot 2^n$ for any value of c and all $n\ge n_0$.

Yes, choose $c\ge 2$ and the statement is true.

Hence $2^{n+1}=O(2^n)$ is true.

Now is $2^{2n}=O(2^n)$ *i.e.,* $2^{2n}\le c\cdot 2^n$ for any c and all $n\ge n_0$.

We now if $\lim\limits_{n\to\infty}\frac{f(n)}{g(n)}=$ constant, then $f(n)=O(g(n))$

In this case the ratio

$$\frac{2^{2n}}{c\cdot 2^n}=\frac{(2^n)^2}{c\cdot 2^n}=\frac{2^n}{c}$$

which grows unbounded with n. This $c\cdot 2^n$ can never be an upper bound to 2^{2n}, regardless how big we choose c.

Example. *Let $p(n)=\sum\limits_{i=0}^{d}a_i n^i$ where $a_d>0$ be a degree-d polynomial in n and let k be a constant. Use the definitions of the asymptotic notations to prove the following properties :*

(a) If $k\ge d$, then $p(n)=O(n^k)$

(b) If $k=d$, then $p(n)=\theta(n^k)$.

Solution. $p(n)=\sum\limits_{i=0}^{d}a_i n^i$ where $a_d>0$.

(*a*) Show that if $k \geq d$, then $p(n) = O(n^k)$.

This is true if and only if

$$0 \leq \sum_{i=0}^{d} a_i n^i \leq c \cdot n^k \text{ for all } n > n_0 \text{ and some constant } c.$$

Divide with n^k we get $0 \leq \sum_{i=0}^{d} a_i n^{i-k} \leq c$.

Since $k \geq d$

all $i-k \leq 0$ *i.e.,* all n^{i-k} will be less than 1.

So, choose $c = \sum_{i=0}^{d} a_i$ the statement is true.

(*b*) Show that if $k = d$, then $p(n) = \theta(n^k)$.

This is true if and only if

$$0 \leq c_1 \cdot n^k \leq \sum_{i=0}^{d} a_i n^i \leq c_2 \cdot n^k$$

Divide by n^k, we get $0 \leq c_1 \leq \sum_{i=0}^{d} a_i n^{i-k} \leq c_2$

Since $k = d$,

Thus $0 \leq c_1 \leq a_0 n^{0-k} + a_1 n^{1-k} + \ldots + a_k n^{k-k} \leq c_2$

i.e., $0 \leq c_1 \leq a_0 n^{-k} + a_1 n^{1-k} + \ldots + a_k n^0 \leq c_2$

Here, we choose $c_1 = a_d$ and $c_2 = \sum_{i=0}^{d} a_i$, the statement is true.

Example. *Prove that for any two functions $f(n)$ and $g(n)$, we have*

$f(n) = \theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Solution. We need to prove $f(n) = O(g(n))$ and $f(n) = \Omega(g(n)) \leftrightarrow f(n) = \theta(g(n))$

To prove one implication,

$f(n) = O(g(n))$ *i.e.,* $f(n) \leq c_0 g(n)$, $n \geq n_0$

$f(n) = \Omega(g(n))$ *i.e.,* $f(n) \geq c_1 g(n)$, $n \geq n_1$

which means $c_1 g(n) \leq f(n) \leq c_0 g(n)$ $n \geq \max(n_0, n_1)$

Now, to prove the other side of the implication.

$f(n) = \theta(g(n))$ means $c_0 g(n) \leq f(n) \leq c_1 g(n)$, $n \geq n_0$

Hence, $f(n) \leq c_0 g(n)$, $n \geq n_0$ means $f(n) = O(g(n))$

$f(n) \geq c_1 g(n)$, $n \geq n_0$ means $f(n) = \Omega(g(n))$

Both sides of the implication were proved true. Hence

$f(n) = O(g(n))$ and $f(n) = \Omega(g(n)) \leftrightarrow f(n) = \theta(g(n))$

Example. *Prove that $o(g(n)) \cap \omega(g(n))$ is the empty set.*

Solution. Proof by contradiction assume $f(n) = o(g(n))$ and $f(n) = \omega(g(n))$

According to definition of little-*o*

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 \quad \ldots(1)$$

and definition of little-omega says $\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$ $\ldots(2)$

which is clearly impossible for any function $f(n)$.

Hence $f(n)$ cannot be both *o* and ω.

Hence $o(g(n)) \cap \omega(g(n))$ is the empty set.

Example. *Prove the equation $\lg(n!) = \theta(n \lg n)$. Also prove that $n! = \omega(2^n)$ and $n! = o(n^n)$.*

Solution. (*a*) $\lg(n!) = \lg[n(n-1)(n-2)\ldots 1] \leq \lg[n \times n \times n \ldots n]$

$= \lg(n^n) = n \lg n$.

Thus $\lg(n!) \leq c_1 n \lg n$ for $c_1 \geq 1$ and $n \geq 0$.

This proves $\lg(n!) = O(n \lg n)$

To prove that $\lg(n!) = \Omega(n \lg n)$

We use Striling's approximation :

$$n! = \sqrt{2\pi n}\left[\frac{n}{e}\right]^n \left[1 + \theta\left(\frac{1}{n}\right)\right]$$

where *e* is the base of the natural logarithm gives us a tighter upper bound and a lower bound as well.

Now, prove $n! \geq cn^n$.

$$n! \geq \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \frac{c}{n}\right)$$

$$= \sqrt{2\pi}\left(\frac{n^{n+0.5}}{e^n}\right)\left(1 + \frac{c}{n}\right) = \sqrt{2\pi}\left(\frac{n^{n+0.5}}{e^n}\right) + \sqrt{2\pi}\left(\frac{cn^{n-0.5}}{e^n}\right)$$

$\geq 2\pi n^n$ for $c > 0$, $n \geq 1$

Hence $n! \geq n^n$ for $c > 0$, $n \geq 1$.

If we take log of both sides, the inequality will still be true because log is monotonic function.

$\log(n!) \geq \log(n^n) = n \log n$ for $c > 0$, $n \geq 1$

Hence $\lg(n!) = \Omega(n \lg n)$

Since, we proved $\lg(n!) = \Omega(n \lg n)$ and $\lg(n!) = O(n \lg n)$

$\lg(n!) = \theta(n \lg n)$.

Now prove $n! = \omega(2^n)$

$$n*(n-1)*(n-2)...1 > c*2*2*2...2$$

Divide both sides by 2^n.

$$\frac{n}{2}*\frac{(n-1)}{2}*\frac{(n-2)}{2}\cdots\frac{1}{2} > c$$

Each element on the left side (except ½) is greater than 1.

Hence if $c > \frac{1}{2}$, the inequality holds for $n > 0$, therefore

$$n! = \omega(2^n)$$

Now prove $n! = o(n^n)$

$$n*(n-1)*...2*1 < c*n*n...n$$

Inequality holds for $n > 1$ and $c > 1$.

Example. *Find two non-negative functions $f(n)$ and $g(n)$ such that neither $f(n) = O(g(n))$ nor $g(n) = O(f(n))$.*

Solution. Consider $f(n) = 1 + \cos(n)$ and $g(n) = 1 + \sin(n)$.

Therefore, both $f(n)$ and $g(n)$ are periodic and take values in [0...2]

But when $f(n) = 0$, $g(n) = 2$ and when $g(n) = 0$, $f(n) = 2$.

Therefore, we cannot find a positive constant c, such that $f(n) \le c \cdot g(n)$ for large n, because $g(n)$ always comes back to 0 when $f(n)$ is 2.

Therefore, $f(n)$ cannot be $O(g(n))$. The same argument works for the other case.

Example. *"Let $f(n)$ and $g(n)$ be asymptotically positive functions. Prove or disprove each of the following conjectures".*

(a) $f(n) = O(g(n))$ implies $g(n) = O(f(n))$

(b) $f(n) + g(n) = \theta(\min(f(n), g(n)))$

(c) $f(n) = O(g(n))$ implies $\lg(f(n)) = O(\lg(g(n)))$ where $\lg(g(n)) \ge 1$ and $f(n) \ge 1$ for all sufficiently large n.

(d) $f(n) = O(g(n))$ implies $2^{f(n)} = O(2^{g(n)})$

(e) $f(n) = O((f(n))^2)$

(f) $f(n) = O(g(n))$ implies $g(n) = \Omega(f(n))$

(g) $f(n) = \theta\left(f\left(\frac{n}{2}\right)\right)$

(h) $f(n) + o(f(n)) = \theta(f(n))$

Solution. (a) No, $f(n) = O(g(n))$ does not imply $g(n) = O(f(n))$

Clearly, $n = O(n^2)$ but $n^2 \ne O(n)$

(b) No, $f(n) + g(n)$ is not $\theta(\min(f(n), g(n)))$

e.g., if $f(n) = n$

$g(n) = 1$ then $f(n) + g(n) = n + 1 \ne \theta(\min(n, 1))$

i.e., $n + 1 \ne \theta(1)$

other example is if $f(n) = n^{17}$ $g(n) = n^2$

then $n^{17} + n^2 \ne \theta(n^2)$.

(c) $f(n) = O(g(n)) \Leftrightarrow 0 \le f(n) \le c \cdot g(n)$ for some $c > 0$ and all $n > n_0$.

$\Rightarrow \lg(f(n)) \le \lg(c \cdot g(n))$

$\Rightarrow \lg(f(n)) \le \lg c + \lg(g(n))$

$\le c_2 \lg(g(n))$ for some $c_2 > 1$

i.e., $\lg(f(n)) = O(\lg(g(n)))$

This only holds if $\lg(g(n))$ is not approaching 0 as n grows.

(d) No, $f(n) = O(g(n))$ does not imply $2^{f(n)} = O(2^{g(n)})$

If $f(n) = 2n$ and $g(n) = n$, then

$2^{2n} = 4^n \ne O(2^n)$

(e) If $f(n) < 1$ for large n, then

$(f(n))^2 < f(n)$

and the upper band will not hold otherwise $f(n) > 1$ and the statement is trivally true.

e.g., suppose $f(n) = \frac{1}{n} \ne O\left(\frac{1}{n^2}\right)$

(f) $f(n) = O(g(n)) \Leftrightarrow 0 \le f(n) \le c_1 g(n)$ for some $c_1 > 0$ and all $n > n_0$

$$\Leftrightarrow 0 \le \left(\frac{1}{c_1}\right) f(n) \le g(n)$$

$$\Leftrightarrow g(n) = \Omega(f(n))$$

(g) No, let $f(n) = 4^n$

$4^n \ne \theta(4^{n/2} = 2^n)$

(h) Show that $f(n) + o(f(n)) = \theta(f(n))$

$$o(f(n)) = \{g(n) : 0 \le g(n) < c \cdot f(n), \ \forall c > 0, \ \forall n \ge n_0\}$$

At most $f(n) + o(f(n))$ can be $f(n) + c f(n) = (1 + c) f(n)$.

At least $f(n) + o(f(n))$ can be $f(n) + 0 = f(n)$.

Thus $0 < f(n) < f(n) + o(f(n)) < (1 + c) f(n)$ which is the same as saying

$f(n) + o(f(n)) = \theta(f(n))$

2.7 Logarithms and Exponents

One of the interesting and sometimes even surprising aspects of the analysis of data structures and algorithms is the ubiquitous presence of logarithms and exponents, where we say

$$\log_b a = c \text{ if } a = b^c$$

i.e., An logarithm is simply an inverse exponential function. Exponential functions are functions that grow at a distressingly fast rate, as anyone who however tried to pay off a mortgage or bank loan understands.

Thus, inverse exponential functions, *i.e.*, logarithms grow refreshingly slowly. If we have an algorithm that runs in $O(\log n)$ time, take it and run. This will be blindingly fast even on very large problem instances. Binary search is an example of an algorithm that takes $O(\log n)$ time. The power of binary search and logarithms is one of the most fundamental ideas in the analysis of algorithms. Two mathematical properties of logarithms are important to understand :

1. The base of the logarithm has relatively little impact on the growth rate. Compare the following *three* values :

- $\log_2 (1{,}000{,}000) = 19.9316$
- $\log_3 (1{,}000{,}000) = 12.5754$
- $\log_{100} (1{,}000{,}000) = 3$

A big change in the base of the logarithm produces relatively little difference in the value of the log.

This is a consequence of the formula for changing the base of the logarithm :

$$\log_a b = \frac{\log_c b}{\log_c a}$$

Changing the base of the log from a to c involves multiplying or dividing by $\log_c a$. This will be lost to the big oh-notation whenever a and c are constants as is typical. Thus we are usually justified in ignoring the base of the logarithm, when analyzing algorithms.

2. Logarithms cut any function down to size. The growth rate of the logarithm of any polynomial function is $O(\lg n)$.

This follows because

$$\log_a n^b = b \log_a n.$$

The power of binary search on a wide range of problems is a consequence of this observation.

For example, note that doing a binary search on a sorted array of n^2 things requires only twice as many comparisons as a binary search on n things. Thus, logarithms efficiently cut any function down to size.

Example. *For each function $f(n)$ and time t in the following table, determine the largest size n of a problem that can be solved in time t, assuming that the algorithm to solve the problem takes $f(n)$ microseconds.*

	1 second	*1 minute*	*1 hour*	*1 day*	*1 month*	*1 year*	*1 century*
$\lg n$							
$\sqrt{n}$							
n							
$n \lg n$							
n^2							
n^3							
2^n							
$n!$							

Solution.

	1 second	*1 minute*	*1 hour*	*1 day*	*1 month*	*1 year*	*1 century*
$\lg n$	2^{10^6}	$2^{6 \cdot 10^7}$	$2^{36 \cdot 10^8}$	$2^{864 \cdot 10^8}$	$2^{2592 \cdot 10^9}$	$2^{94608 \cdot 10^{10}}$	$2^{94608 \cdot 10^{12}}$
$\sqrt{n}$	10^{12}	$36 \cdot 10^{14}$	$1296 \cdot 10^{16}$	$746496 \cdot 10^{16}$	$6718464 \cdot 10^{18}$	$8950673664 \cdot 10^{20}$	$8950673664 \cdot 10^{24}$
n	10^6	$6 \cdot 10^7$	$36 \cdot 10^8$	$864 \cdot 10^8$	$2592 \cdot 10^9$	$94608 \cdot 10^{10}$	$94608 \cdot 10^{12}$
$n \lg n$	62746	2801417	??	??	??	??	??
n^2	10^3	24494897	$6 \cdot 10^4$	293938	1609968	30758413	307584134
n^3	10^2	391	1532	4420	13736	98169	455661
2^n	19	25	31	36	41	49	56
$n!$	9	11	12	13	15	17	18

We assume that all months are 30 days and all years are 365.

Exercise

1. Find the Big-oh(O) notation for the following functions.
   (a) $f(n) = 6993$
   (b) $f(n) = 6n^2 + 135$
   (c) $f(n) = 7*n^2 + 8*n + 39$
   (d) $f(n) = n^4 + 35n^2 + 84$
2. Find the Big theta (θ) and Big omega (Ω) notation for the following :
   (a) $f(n) = 14*7 + 83$
   (b) $f(n) = 83n^3 + 84n$
   (c) $f(n) = n^2 + n$
   (d) $f(n) = 13n + 8$
3. Prove that the following are the loose bounds.
   (a) $17n^3 + 8n^2 = O(n^4)$
   (b) $8n + 16 = O(n^2)$
   (c) $8n^3 + 6n^2 = \Omega(n^2)$
   (d) $8n + 3 = \Omega(1)$
4. Arrange the following growth rates in the increasing order.
   $O(n^4)$, $O(1)$, $O(n^3)$, $O(n)$, $O(n\log n)$, $O(n^2 \log n)$, $\Omega(n^{0.5})$, $\Omega(n^2 \lg n)$, $\theta(n^2)$, $\theta(n^{1.5})$, $\theta(n \lg n)$.
5. Obtain the running time for the following "for" loops.

(a)
```
for (i ← 1 to k)
{
    for (i ← 1 to k²)
    {   for (i ← 1 to k³)
        {
            p ← p * p ;
        }
    }
}
```

(b)
```
for (i ← 2 to k − 1)
{
    for (j ← 3 to k − 2)
        {
            A ← A + 2 ;
        }
}
```

CHAPTER 3

Recurrences

3.1 Introduction

A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs. To solve a recurrence relation means to obtain a function defined on the natural numbers that satisfies the recurrence. For example the worst-case running time $T(n)$ of the MERGE-SORT procedure is described by the recurrence.

$$T(n) = \begin{cases} \theta(1) & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + \theta(n) & \text{if } n > 1 \end{cases}$$

There are three methods for solving this : **Substitution method**, we guess a bound and then use mathematical induction to prove our guess correct. The **iteration method** converts the recurrence into a summation and then relies on techniques for bounding summations to solve the recurrence and the **master method** provides bounds for recurrences of the form.

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \quad \text{where } a \geq 1,\ b > 1 \quad \text{and} \quad f(n) \text{ is a given function.}$$

When we state and solve recurrences, we often omit floors, ceilings and boundary conditions.

3.2 The Substitution Method

It involves guessing the form of the solution and then using mathematical induction to find the constants and show that the solution works. This method is powerful, but it can be applied only in cases when it is easy to guess the form of the answer. The substitution method can be used to establish either upper or lower bounds on a recurrence.

Example. *Consider the recurrence* $T(n)=T\left(\left\lfloor\frac{n}{2}\right\rfloor\right)+1$

we have to show that it is asymptotically bound by $O(\log n)$.

Solution. For $T(n)=O(\log n)$

we have to show that for some constant c.

$$T(n)\le c\log n.$$

Put this in the given recurrence equation.

$$T(n)\le c\log\left(\left\lfloor\frac{n}{2}\right\rfloor\right)+1$$

$$\le c\log\left(\frac{n}{2}\right)+1=c\log n-c\log_2 2+1$$

$$\le c\log n \text{ for } c\ge 1$$

Thus $T(n)=O(\log n)$

Example. *Consider the recurrence* $T(n)=2T\left(\left\lfloor\frac{n}{2}\right\rfloor+16\right)+n$.

We have to show that it is asymptotically bound by $O(n\log n)$.

Solution. For $T(n)=O(n\log n)$ we have to show that for some constant c.

$$T(n)\le cn\log n.$$

Put this in the given recurrence equation.

$$T(n)\le 2\left[c\left(\left\lfloor\frac{n}{2}\right\rfloor+16\right)\log\left(\left\lfloor\frac{n}{2}\right\rfloor+16\right)\right]+n$$

$$=cn\log\left(\frac{n}{2}\right)+32+n=cn\log n-cn\log 2+32+n$$

$$=cn\log n-cn+32+n=cn\log n-(c-1)n+32$$

$$\le cn\log n \quad (\text{for } c\ge 1)$$

Thus $T(n)=O(n\log n)$

Example. *Consider the recurrence*

$$T(n)=\begin{cases}1 & n=1\\ 2T\left(\left\lfloor\frac{n}{2}\right\rfloor\right)+n & n>1\end{cases}$$

Find an asymptotic bound on T.

Solution. We guess the solution is $O(n\lg n)$. Thus for a constant 'c'

$$T(n)\le cn\log n.$$

Put this in the given recurrence equation.

Now $T(n)\le 2c\left\lfloor\frac{n}{2}\right\rfloor\log\left\lfloor\frac{n}{2}\right\rfloor+n$

$$\le cn\log n-cn\lg 2+n$$

$$=cn\log n-n(c\lg 2-1)$$

$$\le cn\log n \ \forall\ c\ge 1.$$

By mathematical induction, we require to show that our solution holds for the boundary conditions.

$$T(1)\le c\cdot 1\log 1=0.$$

Thus for any value of c, this will not hold, so by asymptotic definition we need to prove

$$T(n)\le cn\lg n \quad \text{for all } n\ge n_0.$$

Now $T(2)\le c\cdot 2\log 2$ $T(3)\le c\cdot 3\log 3$

Thus the above relation holds for $c\ge 2$.

Hence our guess $T(n)=O(n\lg n)$ is true.

For most of the recurrences we shall examine, it is straightforward to extend boundary conditions to make the inductive assumption work for small n.

3.2.1 Making a good guess

1. If a recurrence is similar to one we have seen before, then guessing a similar solution is reasonable. For example,
   $T(n)=2T\left(\left\lfloor\frac{n}{2}\right\rfloor+17\right)+n$ which looks difficult because of added "17" in the argument to T, but this additional term cannot substantially affect the solution to the recurrence. Because, when n is large, the difference between $T\left(\left\lfloor\frac{n}{2}\right\rfloor\right)$ and $T\left(\left\lfloor\frac{n}{2}\right\rfloor+17\right)$ is not large and both cut n nearly evenly in half. Hence, we make the guess that $T(n)=O(n\lg n)$.
2. Another way to make a good guess is to prove loose upper and lower bounds on the recurrence and then reduce the range of uncertainly.
3. There are times when we can correctly guess at an asymptotic bound on the solution of a recurrence, but somehow the math doesn't seem to work out in the induction. When we hit such a situation revising the guess by subtracting a lower order term often permits the math to go through.
4. **Changing Variables.** Sometimes, a little algebraic manipulation can make an unknown recurrence similar to one we have seem before.

Example. *Consider the following recurrence*

$T(n)=2T(\sqrt{n})+\log n$. *Solve it by changing variable.*

Solution. $T(n)=2T(\sqrt{n})+\log n$

Suppose $m=\log_2 n \Rightarrow n=2^m$

$\therefore \quad n^{1/2}=2^{m/2} \Rightarrow \sqrt{n}=2^{m/2}$

Put the values, we get

$$T(2^m)=2T(2^{m/2})+m$$

Again consider,

$$S(m)=T(2^m)$$

We have $S(m)=2S\left(\frac{m}{2}\right)+m$

We know this recurrence has the solution.

$$S(m)=O(m\log m)$$

Substitute the values of m, we get

$$T(n)=S(m)=O(m\log m)=O(\lg n \lg\lg n)$$

Example. *Solve the recurrence :* $T(n)=2T(\sqrt{n})+1$ *By making a change of variables.*

Solution. $T(n)=2T(\sqrt{n})+1$

Suppose $m=\log n \Rightarrow n=2^m \Rightarrow \sqrt{n}=2^{m/2}$

Thus $T(2^m)=2T(2^{m/2})+1$

$$S(m)=T(2^m)$$

We have $S(m)=2S\left(\frac{m}{2}\right)+1$

We know the solution to above recurrence is

$$S(m)=O(m)$$

Substituting for m, we get

$$T(n)=S(m)=O(m)=O(\lg n)$$

3.3 The Iteration Method

In iteration method the basic idea is to expend the recurrence and express it as a summation of terms dependent only on 'n' and the initial conditions.

Example. *Consider the recurrence :* $T(n)=3T\left(\left\lfloor\frac{n}{4}\right\rfloor\right)+n$

Solution. We iterate it as follows :

$$T(n)=n+3T\left(\left\lfloor\frac{n}{4}\right\rfloor\right)$$

$$=n+3\left(\left\lfloor\frac{n}{4}\right\rfloor+3T\left(\left\lfloor\frac{n}{16}\right\rfloor\right)\right)$$

$$=n+3\left(\left\lfloor\frac{n}{4}\right\rfloor+3\left(\left\lfloor\frac{n}{16}\right\rfloor+3T\left(\left\lfloor\frac{n}{64}\right\rfloor\right)\right)\right)$$

$$=n+3\left\lfloor\frac{n}{4}\right\rfloor+9\left\lfloor\frac{n}{16}\right\rfloor+27T\left(\left\lfloor\frac{n}{64}\right\rfloor\right)$$

$$\le n+\frac{3n}{4}+\frac{9n}{16}+\ldots 3^iT\left(\frac{n}{4^i}\right)$$

The series terminates when $\frac{n}{4^i}=1 \Rightarrow n=4^i$ or $i=\log_4 n$.

$$T(n)\le n+\frac{3n}{4}+\frac{9n}{16}+\frac{27n}{64}+\ldots+3^{\log_4 n}T(1)$$

$$\le n+\frac{3n}{4}+\frac{9n}{16}+\frac{27n}{64}+\ldots+3^{\log_4 n}\theta(1)$$

$$\le n\sum_{i=0}^{\infty}\left(\frac{3}{4}\right)^i+\theta(n^{\log_4 3}) \text{ as } 3^{\log_4 n}=n^{\log_4 3}$$

$$\le n\cdot\frac{1}{1-\frac{3}{4}}+o(n) \text{ as } \log_4 3<1 \textit{ i.e., } \theta(n^{\log_4 3})=o(n)$$

$$=4n+o(n)=O(n)$$

Example. *Consider the recurrence*

$$T(n)=T(n-1)+1 \textit{ and } T(1)=\theta(1). \textit{ Solve it}$$

Solution. $T(n)=T(n-1)+1$

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

$$=(T(n-3)+1)+1+1$$

$$=T(n-4)+4=T(n-5)+1+4$$

$$=T(n-5)+5$$

$$=T(n-k)+k$$

where $k=n-1$

i.e., $T(n-k)=T(1)=\theta(1)$

i.e., $T(n)=\theta(1)+(n-1)=1+n-1=n=\theta(n)$.

Example. $T(n)=T\left(\frac{n}{3}\right)+n^{4/3}$. *Solve this recurrence by iteration method.*

Solution. $T(n)=T\left(\frac{n}{3}\right)+n^{4/3}=n^{4/3}+T\left(\frac{n}{3}\right)$

$$=n^{4/3}+\left(\frac{n}{3}\right)^{4/3}+T\left(\frac{n}{3^2}\right)$$

$$\therefore \quad T(n) = n^{4/3} + \left(\frac{n}{3}\right)^{4/3} + \left(\frac{n}{3^2}\right)^{4/3} + \ldots \left(\frac{n}{3^k}\right)^{4/3}$$

Let $k = \log_3 n$ when $\frac{n}{3^k} = 1$ *i.e.*, $3^k = n$.

$$\therefore \quad T(n) = n^{4/3} + n^{4/3}\left(\frac{1}{3^{4/3}}\right) + n^{4/3}\left(\frac{1}{3^{4/3}}\right)^2 + \ldots n^{\frac{4}{3}}\left(\frac{1}{3^{4/3}}\right)^k$$

$$= n^{4/3} \sum_{i=0}^{k} \left(\frac{1}{3^{4/3}}\right)^i$$

$$\leq n^{4/3} \sum_{i=0}^{\infty} \left(\frac{1}{3^{4/3}}\right)^i \leq n^{4/3} \frac{1}{1 - \frac{1}{3^{4/3}}} = O(n^{4/3}).$$

Example. Solve $T(n) = T(n-1) + \frac{1}{n}$.

Solution. By iteration method.

$$T(n) = \frac{1}{n} + T(n-1)$$

$$= \frac{1}{n} + \frac{1}{n-1} + T(n-2)$$

$$= \frac{1}{n} + \frac{1}{n-1} + \frac{1}{n-2} + T(n-3)$$

$$= \frac{1}{n} + \frac{1}{n-1} + \frac{1}{n-2} + \ldots + T(1)$$

$$= \sum_{i=0}^{n-2} \frac{1}{n-i} + T(1) \leq \sum_{i=0}^{\infty} \frac{1}{n-i} + \theta(1)$$

Let $n - i = x$

$-di = dx$

Thus, it can be transformed into an integral.

$$\sum_{i=0}^{n} \frac{1}{n-i} = -\int_{n}^{0} \frac{dx}{x} = \log n.$$

Thus $T(n) = \theta(\log n) + \theta(1) = \theta(\log n)$

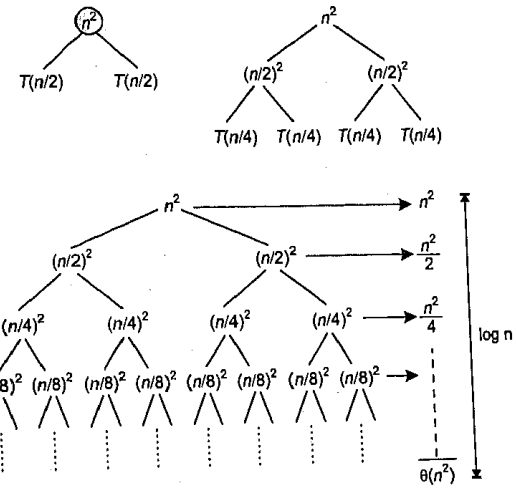
3.3.1 Recursion Tree

Recursion tree method is a pictorial representation of an iteration method, which is in the form of a tree, where at each level nodes are expanded. In general, we consider second term in recurrence as root. It is useful when divide and conquer algorithm is used.

Example. *Consider* $T(n) = 2T\left(\frac{n}{2}\right) + n^2$.

We have to obtain the asymptotic bound using recursion tree method.

Solution. The recursion tree for the above recurrence is



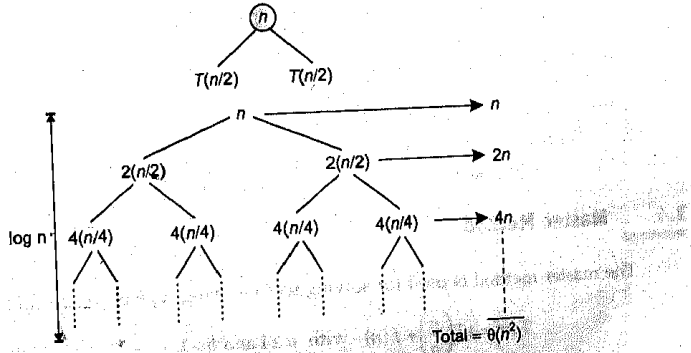
Hence solution is $\theta(n^2)$, because the values decrease geometrically, the total is at most a constant factor more than largest (first) term, and hence the solution is $\theta(n^2)$.

Example. *Consider the following recurrence*

$$T(n) = 4T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n$$

Obtain the asymptotic bound using recursion tree method.

Solution. The recursion tree for the above recurrence



We have $n+2n+4n+\ldots \log_2 n$ times

$$=n(1+2+4+\ldots \log_2 n \text{ times})$$

$$=n\frac{(2^{\log_2 n}-1)}{(2-1)}=\frac{n(n-1)}{1}=n^2-n=\theta(n^2)$$

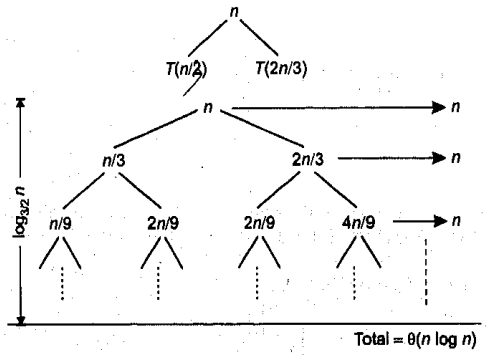
$$\therefore \quad T(n)=\theta(n^2)$$

Example. *Consider the following recurrence*

$$T(n)=T\left(\frac{n}{3}\right)+T\left(\frac{2n}{3}\right)+n$$

Obtain the asymptotic bound using recursion tree method.

Solution. The given recurrence has the following recursion tree.



When we add the values across the levels of the recursion tree, we get a value of n for every level. The longest path from the root to a leaf is

$$n \to \frac{2}{3}n \to \left(\frac{2}{3}\right)^2 n \to \ldots 1$$

Since $\left(\frac{2}{3}\right)^i n=1$ when $i=\log_{3/2} n$.

Thus the height of the tree is $\log_{3/2} n$.

$$\therefore \quad T(n)=n+n+n+\ldots+\log_{3/2} n \text{ times.}$$

$$=\theta(n\log n)$$

3.4 Master Method

The master method is used for solving the following type of recurrence.

$$T(n)=aT\left(\frac{n}{b}\right)+f(n) \quad \text{with } a\geq 1 \text{ and } b>1$$

In this, problem is divided into 'a' subproblems, each of size $\frac{n}{b}$ where a and b are positive constants. The cost of dividing the problem and combining the results of the subproblems is described by the function $f(n)$.

The master method depends on the following theorem.

Master Theorem

Let $T(n)$ be defined on the non-negative integers by the recurrence.

$T(n)=aT\left(\frac{n}{b}\right)+f(n)$ where $a\geq 1$ and $b>1$ be constants and $f(n)$ be a function and $\frac{n}{b}$ can be interpreted as $\left\lfloor\frac{n}{b}\right\rfloor$ or $\left\lceil\frac{n}{b}\right\rceil$.

Then $T(n)$ can be bound asymptotically as follows :

1. If $f(n)=O(n^{\log_b a-\epsilon})$ for some constant $\epsilon>0$, then $T(n)=\theta(n^{\log_b a})$
2. If $f(n)=\theta(n^{\log_b a})$, then $T(n)=\theta(n^{\log_b a}\lg n)$
3. If $f(n)=\Omega(n^{\log_b a+\epsilon})$ for some constant $\epsilon>0$ and if a $f\left(\frac{n}{b}\right)\leq cf(n)$ for some $c<1$ and all sufficiently large n, then $T(n)=\theta(f(n))$, a $f\left(\frac{n}{b}\right)\leq cf(n)$ is called the "regularity" condition.

Note *It is important to note that the three cases do not cover all the possibilities for $f(n)$. There is a gap between cases 1 and 2. When $f(n)$ is smaller than $n^{\log_b a}$ but not polynomially smaller. Similarly there is a gap between cases 2 and 3 when $f(n)$ is larger than $n^{\log_b a}$ but not polynomially larger. If the function $f(n)$ falls into one of these gaps, or if the regularity condition in case 3 fails to hold, the master method cannot be used to solve the recurrence.*

Example. *Solve the recurrence using master method.*

$$T(n)=9T\left(\frac{n}{3}\right)+n$$

Solution. Compare, the given recurrence with the $T(n)=aT\left(\frac{n}{b}\right)+f(n)$

We get $a=9$, $b=3$, $f(n)=n$.

Now, $n^{\log_b a}=n^{\log_b a}=n^2=\theta(n^2)$

$$f(n)=n=n^{\log_3 9-1}=n^{2-1}$$

Hence we can apply case 1 and the solution is $\theta(n^{\log_3 9})=\theta(n^2)$.

Example. *Solve $T(n)=T\left(\frac{2n}{3}\right)+1$ by master method.*

Solution. Here $a=1$, $b=\frac{3}{2}$, $f(n)=1$.

$n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$

$f(n) = n^{\log_b a}$, hence case 2 applies. Thus solution is

$T(n) = \theta(n^{\log_b a} \lg n) = \theta(\lg n)$.

Example. *Solve the recurrence* $T(n) = 3T\left(\frac{n}{4}\right) + n \lg n$ *by master method.*

Solution. Here $a = 3,\ b = 4\ f(n) = n \lg n$.

$$n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$$

$$f(n) = \Omega(n^{\log_4 3 + \epsilon}) \quad \text{where } \epsilon = 0.2$$

Case 3 applies, now for regularity condition *i.e.*,

$$af\left(\frac{n}{b}\right) \le c f(n)$$

$$3\left(\frac{n}{4}\right)\log\left(\frac{n}{4}\right) \le \left(\frac{3}{4}\right) n \lg n = c f(n) \text{ for } c = \frac{3}{4}.$$

Therefore, the solution is

$$T(n) = \theta(n \lg n)$$

Example. *Solve* $T(n) = 16T\left(\frac{n}{4}\right) + n^3$

Solution. By master theorem

$$a = 16 \quad b = 4 \quad f(n) = n^3$$

$$n^{\log_b a} = n^{\log_4 16} = n^2$$

$$f(n) = n^3 = n^{\log_b a + \epsilon} = n^{2+1} \quad \therefore \epsilon = 1$$

Now regularity condition *i.e.*,

$$16 f\left(\frac{n^3}{4^3}\right) \le c f(n^3)$$

$16\frac{n^3}{64} \le cn^3$ for $c = \frac{1}{4}$. So regularity condition holds.

Hence by case 3, the solution is

$$T(n) = \theta(f(n)) = \theta(n^3)$$

3.4.1 Proof of the Master theorem

The proof is in two parts. The first part analyzes the recurrence $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ under the simplifying assumption that $T(n)$ is defined only on the exact power of $b > 1$ *i.e.*, for $n = 1, b, b^2, b^3$ ..... The second part shows how the analysis can be extended to all positive integers n.

Proof for exact powers

The first part of the proof of the master theorem analysis the recurrence $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ under the assumption that n is an exact power of $b > 1$, where b need not be an integer. The analysis is broken into three lemmas.

The first lemma reduces the problem into the problem of evaluating an expression that contains a summation. The second determines bounds on this summation and the third lemma puts the first two together to prove a version of the master theorem in which n is an exact power of b.

Lemma 1

Let $a \ge 1$ and $b > 1$ be constants and let $f(n)$ be a non-negative function defined on exact powers of b. Then define $T(n)$ on exact powers of b by the recurrence.

$$T(n) = \begin{cases} \theta(1) & \text{if } n = 1 \\ aT\left(\frac{n}{b}\right) + f(n) & \text{if } n = b^i \end{cases}$$

where i is a positive integer, then $T(n) = \theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f\left(\frac{n}{b^j}\right)$

Proof. $T(n) = f(n) + aT\left(\frac{n}{b}\right)$

$$= f(n) + af\left(\frac{a}{b}\right) + a^2 T\left(\frac{n}{b^2}\right)$$

$$= f(n) + af\left(\frac{n}{b}\right) + a^2 f\left(\frac{n}{b^2}\right) + \ldots a^i T\left(\frac{n}{b^i}\right)$$

$\because \left(\frac{n}{b^i} = 1 \Rightarrow n = b^i \Rightarrow i = \log_b n\right)$

$$\therefore \quad T(n) = f(n) + af\left(\frac{n}{b}\right) + a^2 f\left(\frac{n}{b^2}\right) + \ldots + a^{\log_b n} T(1)$$

$$= f(n) + af\left(\frac{n}{b}\right) + a^2 f\left(\frac{n}{b^2}\right) + \ldots + a^{\log_b n - 1} T\left(\frac{n}{b^{\log_b n - 1}}\right) + \theta(n^{\log_b a})$$

$(\because a^{\log_b n} = n^{\log_b a})$

$$T(n) = \sum_{j=0}^{\log_b n - 1} a^j f\left(\frac{n}{b^j}\right) + \theta(n^{\log_b a})$$

which completes the proof.

Lemma 2

Let $a \ge 1$ and $b > 1$ be constants and let $f(n)$ be a non-negative function defined on exact powers of b. A function $g(n)$ defined over exact powers of b by

$$g(n) = \sum_{j=0}^{\log_b n - 1} a^j f\left(\frac{n}{b^j}\right) \qquad \ldots(A)$$

It can be bounded asymptotically for exact powers of b, as follows :

1. If $f(n)=O(n^{\log_b a-\epsilon})$ for some constant $\epsilon>0$ then $g(n)=O(n^{\log_b a})$
2. If $f(n)=\theta(n^{\log_b a})$ then $g(n)=\theta(n^{\log_b a}\lg n)$
3. If $af\left(\frac{n}{b}\right)\le c\,f(n)$ for some constant $c<1$ and all $n\ge b$ then $g(n)=\theta(f(n))$

Proof.

Case 1

We have $f(n)=O(n^{\log_b a-\epsilon})$ implies that

$$f\left(\frac{n}{b^j}\right)=O\left(\left(\frac{n}{b^j}\right)^{\log_b a-\epsilon}\right)$$

Put this in equation (A), we get

$$g(n)=O\left(\sum_{j=0}^{\log_b n-1} a^j\left(\frac{n}{b^j}\right)^{\log_b a-\epsilon}\right)$$

Now
$$\sum_{j=0}^{\log_b n-1} a^j\left(\frac{n}{b^j}\right)^{\log_b a-\epsilon}=n^{\log_b a-\epsilon}\sum_{j=0}^{\log_b n-1}\left(\frac{ab^\epsilon}{b^{\log_b a}}\right)^j$$

$$=n^{\log_b a-\epsilon}\sum_{j=0}^{\log_b n-1}(b^\epsilon)^j$$

$$=n^{\log_b a-\epsilon}\left(\frac{b^{\epsilon\log_b n}-1}{b^\epsilon-1}\right)$$

$$=n^{\log_b a-\epsilon}\left(\frac{n^\epsilon\log_b b-1}{b^\epsilon-1}\right)$$

$$=n^{\log_b a-\epsilon}\left(\frac{n^\epsilon-1}{b^\epsilon-1}\right)$$ where b and ϵ are constants.

Thus, expression reduces to

$$n^{\log_b a-\epsilon}O(n^\epsilon)=O(n^{\log_b a})$$

$$\therefore\quad g(n)=O(n^{\log_b a})$$

For case 2

We have $f(n)=\theta(n^{\log_b a})$

So $$f\left(\frac{n}{b^j}\right)=\theta\left(\left(\frac{n}{b^j}\right)^{\log_b a}\right)$$

Put this value in equation (A), we get

$$g(n)=\theta\left(\sum_{j=0}^{\log_b n-1} a^j\left(\frac{n}{b^j}\right)^{\log_b a}\right)$$

We bound the summation with in the θ as in case 1, but this time we do not obtain a geometric series. Instead, we discover that every term of the smmation is the same.

$$\sum_{j=0}^{\log_b n-1} a^j\left(\frac{n}{b^j}\right)^{\log_b a}=n^{\log_b a}\sum_{j=0}^{\log_b n-1}\left(\frac{a}{b^{\log_b a}}\right)^j$$

$$=n^{\log_b a}\sum_{j=0}^{\log_b n-1}(1)^j=n^{\log_b a}\log_b n$$

Put this, we get $g(n)=\theta(n^{\log_b a}\log_b n)$

$$g(n)=\theta(n^{\log_b a}\lg n)$$

Case 3

Since $f(n)$ appears in the definition of $g(n)$ and all terms of $g(n)$ are non-negative, we can conclude that $g(n)=\Omega(f(n))$ for exact powers of b. Under the assumption that $af\left(\frac{n}{b}\right)\le c\,f(n)$ for some constant $c<1$ and $n\ge b$, we have

$$a^j f\left(\frac{n}{b^j}\right)\le c^j f(n)$$

Put this in equation (A) and simplifying we get a geometric series but this series has decreasing terms.

$$g(n)\le\sum_{j=0}^{\log_b n-1} a^j f\left(\frac{n}{b^j}\right)$$

$$\le\sum_{j=0}^{\log_b n-1} c^j f(n)\le f(n)\sum_{j=0}^{\infty}c^j$$

$$=f(n)\left(\frac{1}{1-c}\right)=O(f(n))$$

Since c is a constant. Thus we can conclude that $g(n)=\theta(f(n))$ for exact powers of b which completes the proof.

We can now prove the case in which n is an exact power of b.

Lemma 3

Let $a\ge1$ and $b>1$ be constants and let $f(n)$ be a non-negative function defined on exact powers of b. Define $T(n)$ on exact powers of b by the recurrence.

$$T(n)=\begin{cases}\theta(1) & \text{if } n=1\\ aT\left(\frac{n}{b}\right)+f(n) & \text{if } n=b^i\end{cases}$$

where i is a positive integer. Then $T(n)$ can be bounded asymptotically for exact powers of b as follows :

1. If $f(n)=O(n^{\log_b a-\epsilon})$ for some constant $\epsilon>0$ then $T(n)=\theta(n^{\log_b a})$

2. If $f(n)=\theta(n^{\log_b a})$ then $T(n)=\theta(n^{\log_b a}\lg n)$
3. If $f(n)=\Omega(n^{\log_b a+\epsilon})$ for some constant $\epsilon>0$ and if $af\left(\frac{n}{b}\right)\le cf(n)$, for some constant $c<1$ and all sufficiently large 'n' then $T(n)=\theta(f(n))$

Proof. We use the bounds in lemma 2 to evaluate the summation

For **case 1**, we have

$$T(n)=\theta(n^{\log_b a})+O(n^{\log_b a})$$
$$=\theta(n^{\log_b a})$$

and for **case 2**

$$T(n)=\theta(n^{\log_b a})+\theta(n^{\log_b a}\lg n)$$
$$=\theta(n^{\log_b a}\lg n)$$

For **case 3**, the condition $af\left(\frac{n}{b}\right)\le cf(n)$ implies

$$f(n)=\Omega(n^{\log_b a+\epsilon}) \text{ consequently}$$
$$T(n)=\theta(n^{\log_b a})+\theta(f(n))$$
$$=\theta(f(n))$$

Floor and Ceilings

To complete the proof of the master theorem, we must now extend our analysis to the situation in which floors and ceilings are used in the master recurrence. So that recurrence is defined for all integers.

Obtaining a lower bound on

$$T(n)=aT\left(\left\lceil\frac{n}{b}\right\rceil\right)+f(n) \qquad ...(1)$$

and an upper bound on

$$T(n)=aT\left(\left\lfloor\frac{n}{b}\right\rfloor\right)+f(n) \qquad ...(2)$$

and $\left\lfloor\frac{n}{b}\right\rfloor\le\frac{n}{b}$ and $\left\lceil\frac{n}{b}\right\rceil\ge\frac{n}{b}$.

To iterate the recurrence (1), we obtain a sequences of recursive invocations on the arguments

n

$\lceil n/b\rceil$

$\lceil\lceil n/b\rceil/b\rceil$

$\lceil\lceil\lceil n/b\rceil/b\rceil/b\rceil$

the ith element in the sequence by n_i is defined as

$$n_i=\begin{cases} n & \text{if } i=0\\ \lceil n_{i-1}/b\rceil & \text{if } i>0\end{cases}$$

The first aim is to determine the number of iterations k such that n_k is a constant. Using the inequality $\lceil x\rceil\le x+1$, we get

$$n_0\le n;$$
$$n_1\le\frac{n}{b}+1$$
$$n_2\le\frac{n}{b^2}+\frac{1}{b}+1;$$
$$n_3\le\frac{n}{b^3}+\frac{1}{b^2}+\frac{1}{b}+1$$

In general $n_i\le\frac{n}{b^i}+\sum_{j=0}^{i-1}\frac{1}{b^j}$

$$\le\frac{n}{b^i}+\frac{b}{b-1}$$

and thus, when $i=\lfloor\log_b n\rfloor$, we obtain $n_i\le b+\frac{b}{b-1}=O(1)$

We can now iterate recurrence, obtaining

$$T(n)=f(n_0)+aT(n_1)$$
$$=f(n_0)+af(n_1)+a^2T(n_2)$$
$$\le f(n_0)+af(n_1)+a^2f(n_2)+\ldots+a^{\lfloor\log_b n\rfloor-1}f\left(n_{\lfloor\log_b n\rfloor-1}\right)+a^{\lfloor\log_b n\rfloor}T\left(n_{\lfloor\log_b n\rfloor}\right)$$
$$=\theta(n^{\log_b a})+\sum_{j=0}^{\lfloor\log_b n\rfloor-1}a^jf(n_j)$$

Now $g(n)=\sum_{j=0}^{\lfloor\log_b n\rfloor-1}a^jf(n_j)$ ...(A)

We can evalute this summation like lemma 2. Starting with case 3, if $af\left(\left\lceil\frac{n}{b}\right\rceil\right)\le cf(n)$ for $n>b+\frac{b}{b-1}$ where $c<1$ is a constant then it follows that $a^jf(n_j)\le c^jf(n)$. Therefore, the sum in equation (A) can be evaluated just as in lemma 2.

For case 2, we have $f(n)=\theta(n^{\log_b a})$. If we can show that

$$f(n_j)=O\left(\frac{n^{\log_b a}}{a^j}\right)=O\left(\left(\frac{n}{b^j}\right)^{\log_b a}\right)$$, then the proof is same.

Observe that $j\le\lfloor\log_b n\rfloor$ implies $\frac{b^j}{n}\le1$ Then bound $f(n)=O(n^{\log_b a})$ implies that there exists a constant $c>0$ such that for sufficiently large n_j.

$$f(n_j) \le c\left(\frac{n}{b^j}+\frac{b}{b-1}\right)^{\log_b a}$$

$$= c\left(\frac{n^{\log_b a}}{a^j}\right)\left(1+\left(\frac{b^j}{n}\cdot\frac{b}{b-1}\right)\right)^{\log_b a}$$

$$\le c\left(\frac{n^{\log_b a}}{a^j}\right)\left(1+\frac{b}{b-1}\right)^{\log_b a}$$

$$\le O\left(\frac{n^{\log_b a}}{a^j}\right)$$

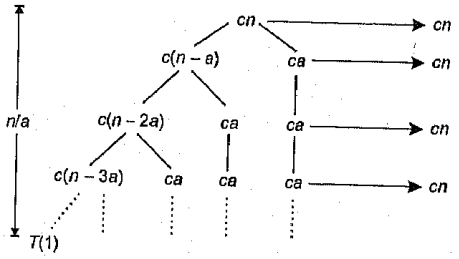
Since $c\left(1+\frac{b}{b-1}\right)^{\log_b a}$ is constant.

Thus case 2 is proved. The proof of case 1 is almost identical. The key is to prove the bound $f(n_j) = O(n^{\log_b a-\epsilon})$ which is similar to the corresponding proof of case 2.

Example. *Use a recursion tree to give an asymptotically tight solution to the recurrence*

$$T(n) = T(n-a)+T(a)+cn \text{ where } a \ge 1 \text{ and } c > 0 \text{ are constants.}$$

Solution.



Suppose the tree ends when $n-ka=1$

$$k = (n-1)/a \approx \frac{n}{a} \text{ levels.}$$

Total sum is $T(n) = \left(\frac{n}{a}\right)\cdot cn = O(n^2)$

Now verify that $T(n) = O(n^2)$ *i.e.*, $T(n) < dn^2$

Assume that $T(k) < dk^2 \quad \forall\, k < n$.

Now

$$T(n) = T(n-a)+T(a)+cn$$
$$< d(n-a)^2 + d(a^2)+cn$$
$$= dn^2 - 2adn + 2da^2 + cn$$
$$= dn^2 - (2adn - 2da^2 - cn)$$

For $T(n) < dn^2$

$$2adn - 2da^2 - cn > 0$$

e.g., $$d = \frac{(c+1)}{2a}$$

we get, $$T(n) < dn^2 - (n - ac - a)$$

$\therefore$ $T(n) < dn^2$ for large value of n.

i.e., $T(n) = O(n^2)$.

Example. *"The recurrence $T(n) = 7T\left(\frac{n}{2}\right)+n^2$ describes the running time of an algorithm A. A competing algorithm A' has a running time of $T'(n) = aT'\left(\frac{n}{4}\right)+n^2$. What is the largest integer value for 'a' such that A' is asymptotically faster than A?"*

Solution. $T(n) = 7T\left(\frac{n}{2}\right)+n^2$

The master method gives us $a=7,\ b=2\ \ f(n)=n^2$

$$n^{\log_b a} = n^{\log_2 7} \approx n^{2.8}$$

It is the first case because $f(n) = n^2 = O(n^{\log 7-\epsilon})$

where $\epsilon = 0.8$ which gives $T(n) = \theta(n^{\log 7})$

The other recurrence $T'(n) = aT'\left(\frac{n}{4}\right)+n^2$ is a bit more difficult to analyse because when 'a' is unknown it is not so easy to say which of the three cases applies in the master method.

But $f(n)$ is same in both algorithms which leads us to try the first case. $\log_4 a$ must be log7, which happens when $a = 48$.

In other words, A' is asymptotically faster than A as long as $a < 48$. The other cases in the master method do not apply for $a > 48$.

Hence A' is asymptotically faster than A up to $a = 48$.

Example. *Give asymptotic upper and lower bounds for $T(n)$ in each of the following recurrences. Assume that $T(n)$ is constant for $n \le 2$. Make your bounds as tight as possible, and justify your answer.*

(*a*) $T(n) = 2T\left(\frac{n}{2}\right)+n^3$

(*b*) $T(n) = T\left(\frac{9n}{10}\right)+n$

(*c*) $T(n) = 16T\left(\frac{n}{4}\right)+n^2$

(*d*) $T(n) = 7T\left(\frac{n}{3}\right)+n^2$

(*e*) $T(n) = 7T\left(\frac{n}{2}\right)+n^2$

(f) $T(n)=2T\left(\frac{n}{4}\right)+\sqrt{n}$

(g) $T(n)=T(n-1)+n$

(h) $T(n)=T(\sqrt{n})+1$

Solution. (a) $T(n)=2T\left(\frac{n}{2}\right)+n^3$

Here $a=2 \quad b=2 \quad f(n)=n^3$

$n^{\log_b a}=n$

$f(n)=\Omega(n^{1+\epsilon})$ for $\epsilon=2$.

$2f\left(\frac{n}{2}\right)=cf(n)$ if $c=\frac{1}{4}$

Thus from case 3 $T(n)=\theta(n^3)$

(b) $T(n)=T\left(\frac{9n}{10}\right)+n$

$a=1 \quad b=\frac{10}{9} \quad f(n)=n$

$n^{\log_b a}=n^0 \quad \therefore \quad f(n)=\Omega(n^{0+\epsilon})$ with $\epsilon=1$

$f\left(\frac{9n}{10}\right)\le cf(n)$ for large values of n if $\frac{9}{10}<c<1$

Thus $T(n)=\theta(n)$

(c) $T(n)=16T\left(\frac{n}{4}\right)+n^2$

$a=16 \quad b=4 \quad f(n)=n^2$

$n^{\log_b a}=n^2$ and $f(n)=n^{\log_b a}$

$\therefore$ By case 2. $T(n)=\theta(n^2\log n)$

(d) $T(n)=7T\left(\frac{n}{3}\right)+n^2$

$a=7 \quad b=3 \quad f(n)=n^2$

$n^{\log_b a}=n^{\log_3 7}$ we know $1<\log_3 7<2$

$f(n)=\Omega(n^{\log_3 7+\epsilon})$ where $\epsilon=2-\log_3 7$

$\therefore \quad T(n)=\theta(n^2)$

(e) $T(n)=7T\left(\frac{n}{2}\right)+n^2$

$a=7 \quad b=2 \quad f(n)=n^2$

$n^{\log_b a}=n^{\log 7}\approx n^{2.8} \quad f(n)=O(n^{\log 7-\epsilon})$ where $\epsilon\approx 0.8$

$\therefore \quad T(n)=\theta(n^{\log 7})$

(f) $T(n)=2T\left(\frac{n}{4}\right)+\sqrt{n}$

$a=2 \quad b=4 \quad f(n)=\sqrt{n}$

$n^{\log_b a}=\sqrt{n} \quad f(n)=n^{\log_b a}$

$\therefore \quad T(n)=\theta(\sqrt{n}\lg n)$

(g) $T(n)=T(n-1)+n$

In this case the master method does not apply, but by recussion tree, we obtain $T(n)=\theta(n^2)$

(h) $T(n)=T(\sqrt{n})+1$

In this case the master method does not apply but by recursion tree, we obtain $T(n)=\theta(\lg\lg n)$.

Example. *Solve* $T(n)=2T\left(\frac{n}{2}\right)+\frac{n}{\lg n}$.

Solution. $T(n)=2T\left(\frac{n}{2}\right)+\frac{n}{n\lg n}$

$\le 2T\left(\frac{n}{2}\right)-n\lg n.$

$T(n)=2T\left(\frac{n}{2}\right)-\theta(n\lg n)$

This recurrence cannot be solved by master method, so using the iteration method, we get

$$2T\left(\frac{n}{2}\right)=\sum_{i=0}^{k-1}2^i 2^{k-i}\log_2(2^{k-i})$$

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

As $k=\log n$.

$=O(n\log^2 n)$

So $T(n)=O(n\log^2 n)-\theta(n\lg n)$

$=O(n\log^2 n)$

Because the term $O(n\log^2 n)$ is greater than $\theta(n\lg n)$.

Example. *Solve the recurrence*

$T(1)=1$

$T(n)=3T\left(\frac{n}{2}\right)+2n^{1.5}$

Solution. We have $T(n)=3T\left(\frac{n}{2}\right)+2n^{1.5}$

Divide by 2, we get

$$\frac{1}{2}T(n)=\frac{3}{2}T\left(\frac{n}{2}\right)+n^{1.5}$$

Let $\frac{T(n)}{2} = S(n)$

$$S(n) = 3S\left(\frac{n}{2}\right) + n^{1.5} \quad ...(A)$$

and $$S(1) = \frac{1}{2} = 0.5$$

Solve the recurrence equation (3) with the help of master theorem

Here $a = 3,\ b = 2\ \ f(n) = n^{1.5}$

$$n^{\log_b a} = n^{\log_2 3} \approx n^{1.59}$$

$$f(n) = n^{\log_b a - \epsilon} = (n^{1.59-0.9}) \text{ i.e., } \epsilon = .09$$

By case 1 : $S(n) = \theta(n^{1.59}) = O(n^{\log_2 3})$

Hence $T(n) = 2\ S(n) = O(n^{\log_2 3})$

$\therefore$ $T(n) = O(n^{\log_2 3})$

Exercise

1. Consider the recurrence

$$T_n = 3T_{n-1} + 15$$
$$T_0 = 0$$

Guess the solution, and prove it by induction.

2. Consider the recurrence

$$T(n) = 14T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n^2$$

Find the asymptotic bound.

3. Using master theorem find the asymptotic bound for the following recurrences :

(a) $T(n) = 2T\left(\frac{n}{2}\right) + n^3$ (b) $T(n) = 5T\left(\frac{n}{4}\right) + n^2$

(c) $T(n) = 7T\left(\frac{n}{2}\right) + O(n^2)$ (d) $T(n) = 2T\left(\frac{n}{2}\right) + O(n^2)$

(e) $T(n) = 2T\left(\frac{n}{2}\right) + O(\sqrt{n})$ (f) $T(n) = 4T\left(\frac{n}{4}\right) + O(n^2)$

4. Can the master method be applied to the recurrence. $T(n) = 4T\left(\frac{n}{3}\right) + n^2 \lg n$? Why or why not ? **Give** an asymptotic upper bound for this recurrence.

5. Use a recursion tree to give an asymptotically tight solution to the recurrence.

$$T(n) = T(an) + T((1-\alpha)n) + cn$$

where α is a constant in the range $0 < \alpha < 1$ and $c < 0$ is also a constant.

CHAPTER

4

Analysis of Simple Sorting Algorithms

One of the fundamental problems of computer science is ordering a list of items. There's a surplus of solutions to this problem, known as **sorting algorithms**. Some sorting algorithms are simple, such as the bubble sort. Others, such as the quick sort, are extremely complicated, but produce lightning-fast results.

The common sorting algorithms can be divided into two classes by the complexity of their algorithms. Algorithmic complexity is generally written in a form known as Big-O notation, where the 'O' represents the complexity of the algorithm and a value n represents the size of the set the algorithm is run against.

For example, $O(n)$ means that an algorithm has a linear complexity. In other words, it takes ten times longer to operate on a set of 100 items than it does on a set of 10 items (10*10 = 100). If the complexity was $O(n^2)$ (quadratic complexity), then it would take 100 times longer to operate on a set of 100 items than it does on a set of 10 items.

The two classes of sorting algorithms are $O(n^2)$, which includes the bubble, insertion, selection, and shell sorts ; and $O(n \log n)$ which includes the heap, merge, and quick sorts.

In addition to algorithmic complexity, the speed of the various sorts can be compared with empirical data. Since the speed of a sort can vary greatly depending on what data set it sorts, accurate empirical results require several runs of the sort be made and the results averaged together.

Here, we are going to look at three simple sorting techniques : *Bubble Sort, Selection Sort,* and *Insertion Sort.*

4.1 Bubble Sort

Bubble sort, also known as exchange sort, is a simple sorting algorithm. It works by repeatedly stepping through the list to be sorted, comparing two items at a time and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which means the list is sorted. The algorithm gets its name from the way smaller elements "bubble" to the top (*i.e.*, the beginning) of the list via the swaps. Because it only uses comparisons to operate on elements, it is a **comparison sort**. This is the easiest comparison sort to implement.

Compare each element (except the last one) with its neighbour to the right

- If they are out of order, swap them
- This puts the largest element at the very end
- The last element is now in the correct and final place

Compare each element (except the last *two*) with its neighbour to the right

- If they are out of order, swap them
- This puts the second largest element next to last
- The last two elements are now in their correct and final places

Compare each element (except the last *three*) with its neighbour to the right. Continue as above until you have no unsorted elements on the left.

The bubble sort is generally considered to the most inefficient sorting algorithm in common usage. Under best-case conditions (the list is already sorted), the bubble sort can approach a constant $O(n)$ level of complexity. General-case is an extremely bad $O(n^2)$.

```
Bubble Sort (A)
1.   for i ← 1 to length [A]
2.       for j ← length [A] downto i + 1
3.           if A[j] < A[j − 1]
4.               exchange (A[j], A[j − 1])
```

The outer loop is executed $n-1$ times. Each time the outer loop is executed, the inner loop is executed. Inner loop executes $n-1$ times at first, linearly dropping to just once. On average, inner loop executes about $n/2$ times for each execution of the outer loop. In the inner loop, the comparison is always done (constant time), the swap might be done (also constant time). Thus result is $n * n/2 * k$, that is $O(n^2)$.

Example. *Illustrate the operation of Bubble sort on the array* $A = \langle 5,2,1,4,3,7,6 \rangle$.

Solution. $A = \langle 5,2,1,4,3,7,6 \rangle$

Here length $[A] = 7$

$i = 1$ to 7 and $j = 7$ to 2

$i = 1, \quad j = 7$

$A[7] = 6$ and $A[6] = 7$. So $A[7] < A[6]$

Now exchange $(A[7], A[6])$

i.e., $A[\,] =$

5	2	1	4	3	6	7

Now, $i = 1$, $j = 6$ then $A[6] = 6$

$A[5] = 3$ and $A[5] < A[6]$

Now, $i = 1$, $j = 5$ then $A[5] = 3$

$A[4] = 4$ and $A[5] < A[4]$

So, exchange $(A[5], A[4])$

and $A[\,] =$

5	2	1	3	4	6	7

Now, $i = 1$, $j = 4$ then $A[4] = 3$

$A[3] = 1$ and $A[4] > A[3]$

Now, $i = 1$, $j = 3$ then $A[3] = 1$

$A[2] = 2$ and $A[3] < A[2]$

So, exchange $(A[3], A[2])$

then $A[\,] =$

5	1	2	3	4	6	7

Now, $i = 1$, $j = 2$ then $A[2] = 1$

$A[1] = 5$ and $A[2] < A[1]$

So, exchange $(A[2], A[1])$

then $A[\,] =$

1	5	2	3	4	6	7

Now, $i = 1$, $j = 7$ then $A[7] = 7$

$A[6] = 6$ and $A[7] > A[6]$. No exchange

Similarly, $i = 2$, $j = 6,5,4$. No change

then $i = 2$, $j = 3$

$A[3] = 2$

$A[2] = 5$ and $A[3] < A[2]$

So, exchange $(A[3], A[2])$

nd $A[\,] =$

1	2	5	3	4	6	7

Now, $i = 3$, $j = 7,6,5$ No change

ien $i = 3$, $j = 4$

$A[4] = 3$

$A[3] = 5$ and $A[4] < A[3]$

So, exchange $(A[4], A[3])$

en $A[\,] =$

1	2	3	5	4	6	7

Now $i=4$, $j=7,6$ No change

Now $i=4$, $j=5$ then $A[5]=4$

$A[4]=5$ and $A[5]<A[4]$

So exchange ($A[5]$, $A[4])$

and $A[\,]=$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | is the sorted array.

Example. (*a*) *Let A' denote the output of the procedure. To prove that it is correct, we need to prove that it terminates and* $A'[1]\le A'[2]\le \ldots \le A'[n]$.

What else must be proved to show that it actually sorts ?

(*b*) *State explicitly a loop invariant for the for loop in lines 2-4, and prove that it holds for the loop.*

(*c*) *Using the termination condition proved in the last part, state a loop invariant for the for loop in lines 1-4 that will allow you to prove the property as stated in the above part a.*

(*d*) *What is the worst-case running time for bubble sort ? How does it compare to that of the insertion sort ?*

Solution. (*a*) We need to show that the elements of A' form a permutation of the elements of A, *i.e.*, they only come from A.

(*b*) The loop invariant may be stated as follows :

At the start of each iteration of the for loop of lines 2 – 4,

$$A[j]=\min\{A[k] \mid j\le k\le n\}$$

and the sub array $A[j..n]$ is a permutation of the values that were in $A[j..n]$ at the time that the loop started.

Now the proof :

- Initially, $j=n$, and the sub array $A[j..n]$ consists of single elements $A[n]$. Thus, the loop invariant trivially holds.
- For the maintenance part, consider an iteration for a given value of $j=j_0$, by the loop invariant, $A[j_0]$ is the smallest value in $A[j_0..n]$. Lines 3-4 exchange $A[j_0]$ and $A[j_0-1]$ if $A[j_0]$ is less than $A[j_0-1]$, and so $A[j_0-1]$ will be the smallest value in $A[j_0-1..n]$ afterward.

  Since the only change to the subarray $A[j_0-1..n]$ is the possible exchange, and the subarray $A[j_0..n]$ is a permutation of the values that were in $A[j_0..n]$ at the time that the loop started, we see that $A[j_0-1..n]$ is a permutation of the values that were in $A[j_0-1..n]$ at the time that the loop started. Decrementing j to j_0-1 for the next iteration maintains the invariant.
- Finally, the loop terminates when j reaches i. By the statement of the loop invariant,

  $$A[i]=\min\{A[k] \mid i\le k\le n\}$$

  and $A[i..n]$ is a permutation of the values that were in $A[i..n]$ at the time that the loop started.

(*c*) We can have the following loop invariant for the loop in lines 1–4.

At the start of each iteration of the this for loop, the subarray $A[1..i-1]$ consists of the $i-1$ smallest values originally in $A[1..n]$, in sorted order, and $A[i..n]$ consists of the $n-i+1$ remaining values originally in $A[1..n]$.

Now the proof :

- Before the first iteration of the loop, $i=1$. The subarray $A[1..i-1]$ is empty, and so the loop invariant trivially holds.
- Consider an iteration for a given value of $i=i_0$. By the loop invariant, $A[1..i_0-1]$ consists to the i_0 smallest values in $A[1..n]$, in sorted order.

  The above part *b* showed that after executing the for loop of lines 2 –4, $A[i_0]$ is the smallest value in $A[i_0..n]$, and so $A[1..i_0]$ is now the i_0 smallest values originally in $A[1..n]$, in sorted order. Morever, since the for loop of lines 2 –4 permutes $A[i_0..n]$, the subarray $A[i_0+1..n]$ consists of the $n-i_0$ remaining values originally in $A[1..n]$. Increment of i to i_0+1 makes the loop invariant holds at the beginning of the next loop.
- Finally, the for loop of lines 1–4 terminates when $i=n+1$, so that $i-1=n$. By the statement of the loop invariant, $A[1..i-1]$ is the entire array $A[1..n]$, and it consists of the original array $A[1..n]$, in sorted order.

(*d*) The running time depends on the number of iteration of the for loop of lines 2-4. For a given value of i, this loop makes $n-i$ iterations, and $i\in[1,n]$

Thus the total number of iterations is

$$B(n)=\sum i=1^{n}(n-i)=\frac{1}{2}n(n-1).$$

4.2 Selection Sort

The idea of selection sort is rather simple : we repeatedly find the next largest (or smallest) element in the array and move it to its final position in the sorted array. Assume that we wish to sort the array in increasing order, *i.e.*, the smallest element at the beginning of the array and the largest element at the end. We begin by selecting the largest element and moving it to the highest index position. We can do this by swapping the element at the highest index and the largest element. We then reduce the effective size of the array by the one element and repeat the process on the smaller sub array. The process stops when the effective size of the array becomes 1 (an array of 1 element is already sorted). Thus, the selection sort works by selecting the smallest unsorted item remaining in the list, and then swapping it with the item in the next position to be filled. The selection sort has a complexity of $O(n^2)$.

```
Selection Sort (A)
1.  n ← length [A]
2.      for j ← 1 to n − 1
3.          smallest ← j
4.          for i ← j + 1 to n
5.              if A[i] < A [smallest]
6.                  then smallest ← i
7.          exchange (A[j], A [smallest])
```

Selection sort is very easy to analyze since none of the loops depends on the data in the array. Selecting the lowest elements requires scanning all n elements (this takes $n-1$ comparisons) and then swapping it into the first position. Finding the next lowest element requires scanning the remaining $n-1$ elements and so on, for a total of $(n-1)+(n-2)+\ldots+2+1=\Theta(n^2)$ comparisons. Each of these scans requires one swap for a total of $n-1$ swaps (the final element is already in place). Thus, the comparisons dominate the running time, which is $\Theta(n^2)$.

Example. *Sort the following array using selection sort* : $A[\,]=\langle 5,2,1,4,3\rangle$.

Solution.

	1	2	3	4	5
A[ ] =	5	2	1	4	3

Here $n=5$

For $j=1$ to 4.

$j=1$, smallest = 1

For $i=2$ to 5

$i=2$, smallest = 1

$A[2]=2$ $A[1]=5$ $A[2]<A[1]$

then smallest = 2

Now $i=3$, smallest = 2

$A[3]=1$ $A[2]=2$ $A[3]<A[2]$

then smallest = 3

Now $i=4$ smallest = 3

$A[4]=4$

$A[3]=1$ $A[4]>A[3]$. No change

Now $i=5$, smallest = 3

$A[5]=3$

$A[3]=1$ $A[5]>A[3]$. So, No change

then

exchange ($A[1]$, A[smallest])

i.e., exchange (5, 1)

Now

A[ ] =	1	2	5	4	3

Now $j=2$, smallest = 2

For $i=3$ to 5

Now $i=3$, smallest = 2

$A[3]=5$

$A[2]=2$ $A[3]>A[2]$ No change

Now $i=4$, smallest = 2. No change

$i=5$, smallest = 2. No change

Now $j=3$, smallest = 3

For $i=4$ to 5

$i=4$ smallest = 3

$A[4]=4$

$A[3]=5$ $A[4]<A[3]$ then smallest = 4

Now $i=5$ smallest = 4

$A[5]=3$

$A[4]=4$ $A[5]<A[4]$ then smallest = 5

Now exchange [$A[3]$, $A[5]$]

then

A[ ] =	1	2	3	4	5

Now $j=4$, smallest = 4, $i=5$

$A[5]=5$

$A[4]=4$ $A[5]>A[4]$. No change

Hence sorted array is

1	2	3	4	5

Example. *Write a pseudocode for the Selection sort algorithm. What loop invariant does it maintain ? Why does it need to run for only the first $n-1$ elements, rather than for all n elements ? Give the best and worst-case analysis of this sort.*

Solution. The pseudocode for selection sort is

```
Selection sort (A)
1.  n ← length [A]
2.      for j ← 1 to n −1
3.          smallest ← j
4.              for i ← j+1 to n
5.                  if A[i]<A [smallest]
6.                      then smallest ← i
7.              exchange (A[j], A [smallest])
```

The algorithm maintains the loop invariant that at the start of each iteration of the for loop between line 2 and line 7, the sub array $A[1..j-1]$ consists of the $j-1$ smallest elements in the array $A[1..n]$, and this sub array is in sorted order.

Morever, after the first $n-1$ elements, according to the previous invariant, the sub array $A[1..n-1]$ contains the smallest $n-1$ elements, sorted. Hence, element $A[n]$ must be the largest element.

Regarding the running time of the algorithm, it is easy to see that, for all cases, we have to do the following amount of work :

$$S(n)=\sum_{j=1}^{n-1}\sum_{i=j+1}^{n}=\sum j=1^{n-1}(n-j)=n(n-1)-\sum_{j=1}(n-1)$$

$$=n(n-1)-\frac{1}{2}n(n-1)=\frac{1}{2}n(n-1)=\Theta(n^2)$$

It yields a 60% performance improvement over the bubble sort, but the insertion sort is over twice as fast as the bubble sort and is just as easy to implement as the selection sort. In short, there really isn't any reason to use the selection sort – use the insertion sort instead.

4.3 Insertion Sort

The insertion sort works just like its name suggests – it inserts each item into its proper place in the final list. The simplest implementation of this requires two list structures – the source list and the list into which sorted items are inserted. To save memory, most implementations use an in-place sort that works by moving the current item pass the already sorted items and repeatedly swapping it with the preceding item until it is in place.

Like the bubble sort, the insertion sort has a complexity of $O(n^2)$. Although it has the same complexity, the insertion sort is a little over twice as efficient as the bubble sort. It has various advantages :

- It is simple to implement
- It is efficient on small data sets

- It is efficient on data sets which are already substantially sorted : it runs in $O(n+d)$ time, where d is the number of inversions
- It is more efficient in practice than most other simple $O(n^2)$ algorithms such as selection sort or bubble sort : the average time is $\frac{n^2}{4}$ and it is linear in the best case.
- It is stable (does not change the relative order of elements with equal keys)
- It is In-place (only requires a constant amount $O(1)$ of extra memory space)
- It is an online algorithm, in that it can sort a list as it receives it.

Every iteration of an insertion sort removes an element from the input data, inserting it at the correct position in the already sorted list, until no elements are left in the input. The choice of which element to remove from the input is arbitrary and can be made using almost any choice algorithm. Sorting is typically done in-place. The resulting array after k iterations contains the first k entries of the input array and is sorted. In each step, the first remaining entry of the input is removed, inserted into the result at the right position, thus extending the result.

```
Insertion-Sort (A)
1.  for j ← 2 to length [A]
2.      do key ← A[j]
3.          ▷Insert A[j] into the sorted sequence A[1..j −1]
4.          i ← j −1
5.          while i > 0 and A[i] > key
6.              do A[i +1] ← A[i]
7.                  i ← i −1
8.          A[i +1] ← key
```

Example. *Illustrate the operation of INSERTION SORT on the array* $A=\langle 2,13,5,18,14\rangle$.

Solution.

	1	2	3	4	5
$A[\,]=$	2	13	5	18	14

For $j=2$ to 5

$j=2$, key = $A[2]$

key =13

$i=2-1=1$, $i=1$

while $i>0$ and $A[1]>13$

condition false, so no change.

Now $j=3$, key $= A[3]=5$

$i=3-1=2$

$i=2$, key $=5$

while $i>0$ and $A[2]>$ key

condition is true

So $A[2+1] \leftarrow A[2]$

$A[3] \leftarrow A[2]$

i.e.,

2		13	18	14

and $i=2-1=1$, $i=1$

while $1>0$ and $A[1]>$ key

condition **false**. So no change

then $A[1+1] \leftarrow$ key

$A[2] \leftarrow 5$

That is

2	5	13	18	14

For $j=4$

key $= A[4]$

key $= 18$, $i=3$

Now, while $3>0$ and $A[3]>18$

condition is **false**. No change

Similarly, $j=5$

key $= A[5]$

So key $=14$, $i=4$

Now, while $4>0$ and $A[4]>14$

condition is **true**

So $A[5]=18$ and $i=4-1=3$

Now, while $3>0$ and $A[3]>14$

condition is **false**.

So $A[3+1]=A[4]=14$

and the sorted array is

$A[\,]=$

2	5	13	14	18

Analysis of Insertion Sort

The time taken by INSERTION - SORT procedure depends on the input. We start by presenting the INSERTION - SORT procedure with the time "cost" of each statement and the number of times each statement is executed.

	Insertion-Sort (A)	*Cost*	*times*
1.	for $j \leftarrow 2$ to length [A]	c_1	n
2.	do key $\leftarrow A[j]$	c_2	$n-1$
3.	▷ insert $A[j]$ into the sorted sequence $A[i..j-1]$	0	$n-1$
4.	$i \leftarrow j \leftarrow 1$	c_4	$n-1$
5.	while $i>0$ and $A[i]>$ key	c_5	$\sum_{j=2}^{n} t_j$
6.	do $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7.	$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8.	$A[i+1] \leftarrow$ key	c_8	$n-1$

The running time of the algorithm is the sum of running times for each statements executed.

To compute $T(n)$, the running time, we sum the product of the cost and times columns.

$$T(n)=c_1 n+c_2(n-1)+c_4(n-1)+c_5\sum_{j=2}^{n} t_j+c_6\sum_{j=2}^{n}(t_j-1)+c_7\sum_{j=2}^{n}(t_j-1)+c_8(n-1)$$

Best case occurs if the array is already sorted. For each $j=2,3,\ldots n$, we then find that $A[i] \leq$ key in line 5 when $i=j-1$.

Thus $t_j=1$ for $j=2,3,\ldots n$

$$T(n)=c_1 n+c_2(n-1)+c_4(n-1)+c_5(n-1)+c_8(n-1)$$

$$=(c_1+c_2+c_4+c_5+c_8)n-(c_2+c_4+c_5+c_8)$$

$$=an+b, \text{ where } a \text{ and } b \text{ are constants.}$$

$$=\text{linear function of } n=O(n).$$

In worst case, the array is in reverse order. We must compare each element $A[j]$ with each element in the entire sorted, subarray $A[1\ldots j-1]$ and $t_j=j$ for $j=2,3,\ldots n$.

$$T(n) = c_1 n + c_2 (n-1) + c_4 (n-1) + c_5 \sum_{j=2}^{n} j + c_6 \sum_{j=2}^{n} (j-1) + c_7 \sum_{j=2}^{n} (j-1) + c_8 (n-1)$$

$$= c_1 n + c_2 (n-1) + c_4 (n-1) + c_5 \left(\frac{n(n+1)}{2} - 1 \right) + c_6 \left(\frac{n(n-1)}{2} \right) + c_7 \left(\frac{n(n-1)}{2} \right) + c_8 (n-1)$$

$$\left(\because \sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 \text{ and } \sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2} \right)$$

$$= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2} \right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8 \right) n - (c_2 + c_4 + c_5 + c_8)$$

$= an^2 + bn + c$ for a, b and c are constant.

= quadratic function of n.

$= O(n^2)$.

We use loop invariant to help us understand why an algorithm is correct. We must show *three* things about a loop invariant :

1. Initialization
2. Maintenance
3. Termination

- **Initialization.** It is true prior to the first iteration of the loop.
- **Maintenance.** If it is true before an iteration of the loop, it remains true before the next iteration.
- **Termination.** When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

The third property is the most important one, since we are using the loop invariant to show correctness.

Example. *Insertion sort can be expressed as a recursive procedure : recursively sort $A[1..n-1]$, then insert $A[n]$ into the sorted array. Write a recurrence for the running time of this recursive version.*

Solution. Since it takes $\theta(n)$ time in the worst case to insert $A[n]$ into the sorted array of $A[1..n-1]$, let $T(n)$ be the total time it takes to insertion sort a list with n elements, we get the recurrence

$$T(n) = \theta(1) \text{ if } n = 1 \text{ and}$$
$$T(n) = T(n-1) + \theta \text{ otherwise.}$$

The solution is this recurrence is that $T(n) = \theta(n^2)$.

Example. *Can we use a binary search to replace the linear search the insertion sort algorithm to improve the worst-case running to $\theta(n \log n)$?*

Solution. The modified code, which uses a procedure similar to the binary search to find out the appropriate position, k in the line 4 ; then move all the items k and $j-1$ one position to the right in lines 5–6.

```
Insertion-Sort-with-Binary Search (A)
1.  for j ← 2 to length [A]
2.      do key ← A[j]
3.  // Insert A[j] into the sorted A[1.. j−1]
4.  k ← Binary Search for Position (key)
5.  for m ← j down to k + 1
6.      A[m] ← A[m − 1]
7.  A[k] ← key
```

In the above code, we use the binary search to look for the insertion position of key, which takes $\theta(\log j)$ to complete. Since we work with the worst case, when the input is a reversibly sorted list, it always comes back with the position of 1. Then, lines 5–6 always takes $j-1$ movements, plus the two made in line 2 and 7, for each j, $j+1$ movements have to be made. Hence, the total time is the following :

$$T(n) = \sum_{j=2}^{n} [\log j + (j+1)]$$
$$= \theta(n \log n) + \theta(n^2)$$
$$= \theta(n^2)$$

Example. *Write the pseudocode of Insertion Sort as a recursive procedure.*

Solution.

```
Insertion - Sort (A, p, r)
1.  if p < r
2.      then INSERTION - SORT (A, p, r − 1)
3.           INSERT (A[p, ... r − 1], A[r] )
```

where INSERT ($A[p..r]$, key) inserts key into the sorted array $A[p..r]$.

The pseucode for INSERT is shown below and it consists of one iteration of the non-recursive INSERTION-SORT.

```
INSERT (A [p..r], key )
1.  i ← r − 1
2.      while i > p − 1 and A[i] > key
3.          do A[i + 1] ← A[i]
4.             i ← 1−1
5.  A[i + 1] ← key
```

Exercise

1. Rewrite the Bubble sort, Selection sort and Insertion sort procedure which sorts the given list of elements in a non-increasing order.
2. Illustrate the operation of INSERTION SORT on the array $A = \langle 41,31,69,16,38,62 \rangle$
3. Illustrate the operation of Selection sort and Bubble sort on the following array :
   (*a*) $\langle 10,2,13,15,19,2,18 \rangle$
   (*b*) $\langle 31,41,59,26,41,48,101,99,78 \rangle$
   (*c*) $\langle 70,80,40,50,60,35,85,2 \rangle$
   (*d*) $\langle 1,3,5,8,9,10 \rangle$
   (*e*) $\langle 1,1,1,1,1,1 \rangle$
4. Design an algorithm to find the sum of smallest $\log_2 n$ elements in an unsorted array of n distinct elements.
5. Analyse the Insertion sort in worst case.

CHAPTER

5

Merge Sort

5.1 Introduction

This algorithm was invented by John von Neumann in 1945. It closely follows the divide-and-conquer paradigm.

Conceptually, it works as follows :

1. **Divide.** Divide the unsorted list into two sub lists of about half the size
2. **Conquer.** Sort each of the two sub lists recursively until we have list sizes of length 1, in which case the list itself is returned
3. **Combine.** Merge the two-sorted sub lists back into one sorted list.

We note that the recursion "bottoms out". When the sequence to be sorted has length 1, in which case there is no work to be done, and since every sequence of length 1 is already in sorted order. The key operation of the merge sort algorithm is the merging of two sorted sequences in the "combine" step. To perform the merging, we use an auxiliary procedure MERGE (A, p, q, r), where A is an array and p, q, and r are indices numbering elements of the array such that $p \le q < r$. The procedure assumes that the sub arrays $A[p..q]$ and $A[q+1..r]$ are in sorted order. It *merges* them to form a single sorted sub array that replaces the current sub array $A[p..r]$.

```
MERGE (A, p, q, r)

1.  n1 ← q − p + 1
2.  n2 ← r − q
3.  create arrays L[1.. n1 + 1] and R[1.. n2 + 1]
4.  for i ← 1 to n1
5.      do L[i] ← A[p + i − 1]
6.  for j ← 1 to n2
7.      do R[j] ← A[q + j]
8.  L[n1 + 1] ← ∞
9.  R[n2 + 1] ← ∞
10. i ← 1
11. j ← 1
12. for k ← p to r
13.     do if L[i] ≤ R[j]
14.         then A[k] ← L[i]
15.              i ← i + 1
16.         else A[k] ← R[j]
17.              j ← j + 1
```

It is easy to imagine a MERGE procedure that takes time $\theta(n)$, where $n = r - p + 1$ is the number of elements being merged.

We can now use the MERGE procedure as a subroutine in the merge sort algorithm. The procedure MERGE-SORT (A, p, r) sorts the elements in the sub array $A[p..r]$. If $p \le r$, the sub array has at most one element and is therefore already sorted. Otherwise, the divide step simply computes an index q that partitions $A[p..r]$ into two sub arrays: $A[p..q]$, containing $\lceil n/2 \rceil$ elements, and $A[q+1..r]$, containing $\lfloor n/2 \rfloor$ elements.

```
MERGE-SORT (A, p, r)

1.  if p < r
2.      then q ← ⌊(p + r) / 2⌋
3.           MERGE-SORT (A, p, q)
4.           MERGE-SORT (A, q + 1, r)
5.           MERGE (A, p, q, r)
```

To sort the entire sequence $A = (A[1], A[2], \ldots A[n])$, we call MERGE-SORT $(A, 1, length[A])$, where once again $length[A] = n$. If we look at the operation of the procedure bottom-up when n is a power of two, the algorithm consists of merging pairs of 1-item sequences to form sorted sequences of length 2, merging pairs of sequences of length 2 to form sorted sequences of length 4, and so on, until two sequences of length $n/2$ are merged to form the final sorted sequence of length n. Figure shows this process in the array $(5,2,4,6,1,3,2,6)$.

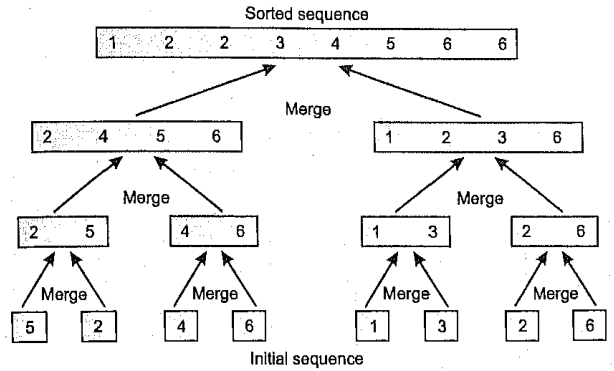


Figure 5.1

Example of Merge Sort

0 MS	85	24	63	45	17	31	96	50
1 Div	85	24	63	45	17	31	96	50
1 MS	85	24	63	45				
2 Div	85	24	63	45				
2 MS	85	24						
3 Div	85	24						
3 MS	85							
3 MS		24						
3 Merge	24	85						
2 MS			63	45				
3 Div			63	45				
3 MS			63					
3 MS				45				
3 Merge			45	63				
2 Merge	24	45	63	85				

0 MS	85	24	63	45	17	31	96	50
1 Div	85	24	63	45	17	31	96	50
1 MS	85	24	63	45	17	31	96	50
2 Merge	24	45	63	85	17	31	96	50
1 MS	24	45	63	85	17	31	96	50
2 Div	24	45	63	85	17	31	96	50
2 MS	24	45	63	85	17	31	96	50
3 Div	24	45	63	85	17	31	96	50
3 MS	24	45	63	85	17	31	96	50
3 MS	24	45	63	85	17	31	96	50
3 Merge	24	45	63	85	17	31	96	50
2 MS	24	45	63	85	17	31	96	50
3 Div	24	45	63	85	17	31	96	50
3 MS	24	45	63	85	17	31	96	50
3 MS	24	45	63	85	17	31	96	50
3 Merge	24	45	63	85	17	31	50	96
2 Merge	24	45	63	85	17	31	50	96
1 Merge	17	24	31	45	50	63	85	96

5.2 Analysis of Merge Sort

Although the pseudocode for MERGE-SORT works correctly when the number of elements is not even, our recurrence-based analysis is simplified if we assume that the original problem size is a power of two. Each divide step then yields two subsequence of size exactly $n/2$. This assumption does not affect the order of growth of the solution to the recurrence.

Merge sort on just one element takes constant time. When we have $n > 1$ elements, we break down the running time as follows.

- **Divide.** The divide step just computes the middle of the sub array, which takes constant time. Thus, $D(n) = \theta(1)$.
- **Conquer.** We recursively solve two sub problems, each of size $n/2$, which contributes $2T(n/2)$ to the running time.
- **Combine.** We have already noted that the MERGE procedure on an n-element sub array takes time $\theta(n)$, so $C(n) = \theta(n)$.

When we add the functions $D(n)$ and $C(n)$ for the merge sort analysis, we are adding a function that is $\theta(n)$ and a function that is $\theta(1)$. This sum is a linear function of n, that is, $\theta(n)$. Adding it to the $2T(n/2)$ term from the "conquer" step gives the recurrence for the worst-case running time $T(n)$ of merge sort :

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

By Master Theorem , we shall show that $T(n)$ is $\theta(n \lg n)$, where $\lg n$ stands for $\log_2 n$. For large enough inputs, merge sort, with its $\theta(n \lg n)$ running time, outperforms insertion sort, whose running time is $\theta(n^2)$, in the worst case

Merge sort's merge operation is useful in online sorting, where the list to be sorted is received a piece at a time, instead of all at the beginning. In this application, we sort each new piece that is received using any sorting algorithm, and then merge it into our sorted list so far using the merge operation. However, this approach can be expensive in time and space if the received pieces are small compared to the sorted list – a better approach in this case is to store the list in a self-balancing binary search tree and add elements to it as they are received.

Although heap sort has the same time bounds as merge sort, it requires only $\theta(1)$ auxiliary space instead of merge sort's $\theta(n)$, and is consequently often faster in practical implementations. Quick sort, however, is considered by many to be the fastest general-purpose sort algorithm although there are proofs to show that merge sort is indeed the fastest sorting algorithm out of the three. Its average-case complexity is $O(n \log n)$ even with perfect input and given optimal pivots quick sort is not as fast as merge sort and it is quadratic in the worst case. On the plus side, merge sort is a stable sort, parallelizes better, and is more efficient at handling slow-to-access sequential media. Merge sort is often the best choice for sorting a linked list: in this situation it is relatively easy to implement a merge sort in such a way that it requires only $\theta(1)$ extra space, and the slow random-access performance of a linked list makes some other algorithms (such as Quick sort) perform poorly, and others (such as heap sort) completely impossible.

As of Perl 5.8, merge sort is its default-sorting algorithm (it was Quick sort in previous versions of Perl). In Java, the Arrays sort( ) methods use merge sort or a tuned Quick sort depending on the data types and for implementation efficiency switch to insertion sort when fewer than seven array elements are being sorted.

Merge sort incorporates *two* main ideas to get better its runtime:

1. A small list will take fewer steps to sort than a large list.
2. Fewer steps are required to construct a sorted list from two sorted lists than two unsorted lists. For example, you only have to traverse each list once if they're already sorted (see the merge function below for an example implementation).

5.3 Insertion Sort and Merge Sort

Suppose we are comparing implementation of insertion sort and merge sort on the same machine. For inputs of size n, insertion sort runs in $8n^2$ steps, while merge sort runs in $64\, n \log n$ steps.

For $n \le 43$, $8n^2 \le 64 n \log n$ and insertion sort beats merge sort.

Thus for $n \le 43$ insertion sort beats merge sort.

Although merge sort runs in $\theta(n \lg n)$ worst case time and insertion sort runs in $\Theta(n^2)$ worst-case time, the constant factors in insertion sort make it faster for small n. Therefore, it makes sense to use insertion sort within merge sort when subproblems become sufficiently small. Consider a modification of merge sort in which subarrays of size k or less (for some k) are not divided further, but sorted explicitly with Insertion sort.

```
MERGE-SORT (A, p, r)
  if p < r − k + 1
      then q ← ⌊(p + r)/2⌋
           MERGE-SORT (A, p, q)
           MERGE-SORT (A, q + 1, r)
           MERGE (A, p, q, r)
      else INSERTION-SORT (A[p..r])
```

(i) The total time spent on all calls to Insertion sort is in the worst-case $\theta(nk)$.

Proof. If $n \le k$, then this modified MERGE-SORT will just call. INSERTION-SORT on the whole array.

Therefore, the running time will be $\theta(n^2) = \theta(n \cdot n) = O(nk)$

So let us assume that $n > k$.

First note that the length of subarray $A[p..r]$ is $l = r - p + 1$. So the condition $p < r - k + 1$ is the same as $l > k$. Therefore, any subarray of length l on which we call INSERTION-SORT has to satisfy $l \le k$ (this is actually in the given of the algorithm). Moreover, any subarray $A[p..r]$ of length l on which we call INSERTION-SORT has to satisfy $\frac{k}{2} \le l$. The reason for this is the following : if INSERTION-SORT is called on $A[p..r]$, then $A[p..r]$ must be the first or second half of another subarray $A[p'..r']$ that was divided (we assumed that $n > k$ so $A[p..r]$ cannot be $A[1..n]$). Thus, $A[p'..r']$ has length $l' > k$. Therefore, since $l \ge \left\lfloor \frac{l'}{2} \right\rfloor$ and $l' > k$, then $l \ge \frac{k}{2}$.

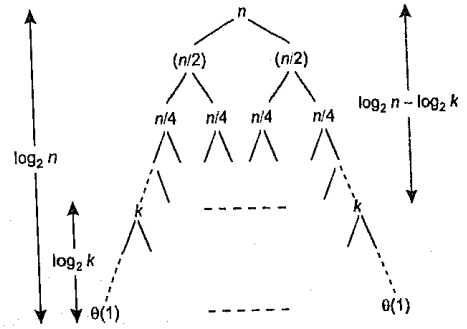
The running time of INSERTION-SORT on $A[p..r]$ is $\Theta(l^2)$, where $l = r - p + 1$ is the length of $A[p..r]$. But since $\frac{k}{2} \le l \le k$, then $l = \Theta(k)$ and the running time of INSERTION-SORT on $A[p..r]$ is $\Theta(k^2)$.

Now let us see how many such sub arrays (on which we call INSERTION-SORT) we have. Since all the sub arrays are disjoint (*i.e.*, their indices do not overlap) and for every one of them $\frac{k}{2} \le l \le k$, then the number of these sub arrays, call it m, satisfies $\frac{n}{k} \le m \le \frac{2n}{k}$. Therefore $m = \Theta(n/k)$.

The total time spent on INSERTION-SORT is therefore $\Theta(k^2) \cdot \Theta(n/k) = \Theta(nk)$.

(ii) Show that the total time spent on merging is in the worst-case $\theta(n \log(n/k))$.

Proof. The merging proceeds in the same way as with the basic version of MERGE-SORT except that it stops whenever the subarray reaches a length $l \le k$ (instead of 1). Therefore, we need to determine the number of levels in the recursive tree. With the basic version of MERGE-SORT, we argued that the number of levels is $\Theta(\log n)$ because we divide n by 2 with every level until we reach 1. With this modification, we divide n by 2 with every level until we reach k (or less). Therefore the number of levels is $\theta(\log n) - \theta(\log k)$, which is the number of levels originally minus the number of levels that we would have had if we continued beyond k. But $\theta(\log n) - \theta(\log k) = \theta(\log(n/k))$. So the total time that we spend on merging is $\Theta(n \log(n/k))$ since we spend $\Theta(n)$ time in every level as before.



Therefore, this modified merge sort runs in $\theta(nk) + \theta(n \log(n/k))$ (sorting + merging).

Thus, the total running time is $\theta(nk + n \log(n/k))$.

(iii) What is the largest asymptotic (θ-notation) value of k as a function of n for which the modified algorithm has the same asymptotic running time as standard merge sort ? How should k be chosen in practice ?

Proof. Obviously, if k grows faster than $\log n$ asymptotically, then we would get something worse than $\Theta(n \log n)$ because of the $\Theta(nk)$ term. Therefore, we know that $k = O(\log n)$. So lets pick $k = \Theta(\log n)$ and see if this works. If $k = \Theta(\log n)$ then the running time of the modified MERGE-SORT will be

$$\Theta\left(n \log n + n \log \frac{n}{\log n}\right) = \Theta(n \log n + n \log n - n \log n(\log n)) = \Theta(n \log n).$$

So if $k = \Theta(\log n)$, the running time of the modified algorithm has the same asymptotic running time as the standard one. In practice, k should be the maximum input size on which Insertion sort is faster then Merge sort.

Example. *Rewrite the MERGE procedure so that it does not use sentinels, instead stopping once either array has had all its items copied back to A.*

Solution. The code could be the following :

```
MERGE (A, p, q, r)
1.   n1 ← q − p + 1
```

```
2.    n2 ← r − q
3.    create arrays L[1..n] and R[1..n2]
4.    for i ←1 to n1
5.        do L[i] ←A[p + i − 1]
6.    for j ←1 to n2
7.        do R[j] ←A[q + i]
8.    i ←1
9.    j ←1
10.   for k ←p to r
11.       do if (i <= n1)
12.           if (j <= n2)
13.           then if (L[i] <= R[j])
14.                then A[k] ←L[i]
15.                     i ←i + 1
16.                else A[k] ← R[j])
17.                     j ← j + 1
18.           else A[k] ←L[i]
19.                i <i + 1
20.       else if (j <= n2)
21.           then A[k] ← R[j]
22.                j ← j + 1
```

The idea is that once a list, either L or R is completed, everything in the order list is copied back.

Exercise

1. Consider following list of elements as 50, 40, 20, 70, 15, 35, 20, 60
   Sort the above list using Merge sort.
2. Among, Merge sort, Insertion sort and Bubble sort which sorting techniques is the best in the worst case. Support your argument with an example and analysis.
3. Analyse the Merge sort algorithm. Argue on its best case, average case and worst case time complexity.
4. Sort the list 70, 80, 40, 50, 60, 12, 35, 95, 10. Using Merge sort.
5. Show that the merge sort associated with an execution of merge-sort on a sequence of size n has height $\lceil \log n \rceil$.
6. Show that merging two sorted sequences S_1 and S_2 takes $O(n_1 + n_2)$ time, where n_1 is the size of S_1 and n_2 is the size of S_2.
7. Show that the running time of the merge-sort algorithm on the n-element sequence is $O(n \log n)$, even when n is not a power of 2.
8. Let S be a sequence of n elements on which a total order relation is defined. An inversion in S is a pair of elements x and y such that x appears before y in S but $x > y$. Describe an algorithm running in $O(n \log n)$ time for determining the number of inversions in S.

CHAPTER 6

Heap Sort

6.1 Binary Heap

The binary heap data structure is an array that can be viewed as a complete binary tree. Each node of the binary tree corresponds to an element of the array. The array is completely filled on all levels except possibly lowest.

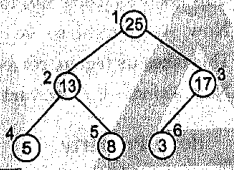


Figure 6.1

We represent heaps in level order, going from left to right. The array corresponding to the heap in Fig. 6.1 above is

	1	2	3	4	5	6
A	25	13	17	5	8	3

If an array A contains key values of nodes in a heap, length $[A]$ is the total number of elements.

heap-size $[A]$ = length $[A]$ = number of elements.

The root of the tree $A[1]$ and given index i of a node the indices of its parent, left child and right child can be computed.

```
PARENT (i)
    return floor (i / 2)
LEFT (i)
    return 2i
RIGHT (i)
    return 2i + 1
```

Let us try these out on a heap to make sure we believe they are correct, take this heap

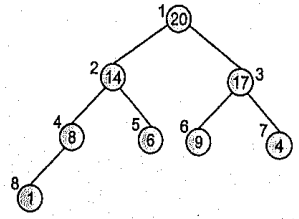


Figure 6.2

which is represented by the array

1	2	3	4	5	6	7	8	9
20	14	17	8	6	9	4	1	

The index of the 20 is 1

To find the index of the left child, we calculate $1*2 = 2$

This takes us (correctly) to the 14.

Now, we go right, so we calculate $2*2+1 = 5$.

This takes us (again, correctly) to the 6.

Now, 4's index is 7, we want to go to the parent, so we calculate $7/2 = 3$ which takes us to the 17.

6.2 Heap Property

In a Max-heap, for every node i other than the root, the value of a node is greater than or equal (at most) to the value of its parent

$$A[\text{PARENT}(i)] \geq A[i]$$

Thus, the largest element in a heap is stored at the root. Following is an example of MAX-HEAP.

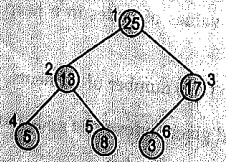


Figure 6.3

and a Min-Heap is organized in the opposite way *i.e.*, the min-heap property is that for every node i other than the root is

$$A[\text{PARENT}(i)] \leq A[i]$$

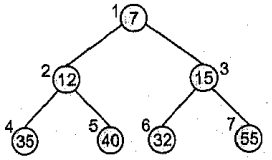


Figure 6.4

By the definition of a heap, all the tree levels are completely filled except possibly for the lowest level which is filled from the left up to a point.

Following is not a heap, because it only has the heap property – it is not a complete binary tree. Recall that to be complete, a binary tree has to fill up all of its levels with the possible exception of the last one, which must be filled in from the left side.

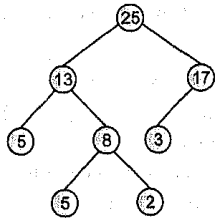


Figure 6.5

6.3 Height of a Heap

Height of a node

We define the height of a node in a tree to be a number of edges on the longest simple downward path from a node to a leaf.

Height of a tree

The number of edges on a simple downward path from a root to a leaf.

Note that the height of a tree with n node is $\lfloor \lg n \rfloor$ which is $\Theta(\lg n)$. This implies that an n-element heap has height $\lfloor \lg n \rfloor$.

In order to show this let the height of n-element heap be h. From the bounds obtained on maximum and minimum number of elements in a heap, we get

$$2^h \leq n \leq 2^{h+1} - 1$$

Where n is the number of elements in a heap.

$$2^h \leq n \leq 2^{h+1}$$

Taking logarithms to the base 2

$$h \le \lg n \le h+1$$

It follows that $h = \lfloor \lg n \rfloor$

Clearly, a heap of height h has the **minimum number** of elements when it has just one node at the lowest level. The levels above the lowest level form a complete binary tree of $h-1$ and $2^h - 1$ nodes.

Hence, the minimum number of nodes possible in a heap of height h is 2^h.

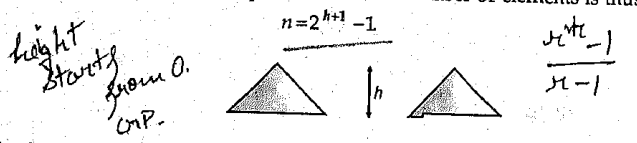
Clearly, a heap of height h, has the **maximum number** of elements when its lowest level is completely filled. In this case the heap is a complete binary tree of height h and hence has $2^{h+1} - 1$ nodes.

We know from above that largest element resides in root, $A[1]$. The natural question to ask is **where in a heap might the smallest element resides** ? Consider any path from root of the tree to a leaf. Because of the heap property, as we follow that path, the elements are either decreasing or staying the same. If it happens to be the case all elements in the heap are distinct, then the above implies that the smallest is in a leaf of the tree. It could also be that an entire sub tree of the heap is the smallest element or indeed that there is only one element in the heap, which is the smallest element, so the smallest element is everywhere. Note that anything below the smallest element must equal the smallest element.

Example. *What are the minimum and maximum numbers of elements in a heap of height h ?*

Solution. Let n is the number of elements in a heap where the height is fixed to h.

The largest value of n occurs when the heap is shaped as the left Fig. below. We cannot pack any more elements into such a heap. The maximum number of elements is thus



In the other situation when there is only one element on the last row, we can't have any fewer elements in such a heap without reducing the height h. The minimum number of elements is thus $n = 2^{h+1-1} - 1 + 1 = 2^h$.

In conclusion, $2^h \le n \le 2^{h+1} - 1$

Example. *Show that an n-element heap has height* $\lfloor \lg n \rfloor$.

Solution. From above, we can derive the following constraints by taking the logarithm of both sides :

$$h \le \lg n \le \lg(2^{h+1} - 1)$$

Since $\lg(2^{h+1} - 1) < \lg(2^{h+1}) = h+1$, we can write

$$h \le \lg n \le h+1$$

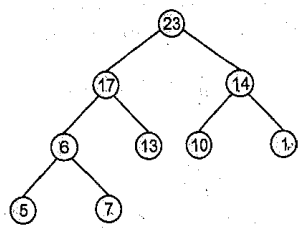
which is the same as saying $h = \lfloor \lg n \rfloor$.

Example. *Where in a max-heap might the smallest element reside, assuming that all elements are distinct ?*

Solution. The smallest element must reside in one of the leaf nodes.

Example. *Is the sequence* $\langle 23,17,14,6,13,10,1,5,7,12 \rangle$ *a max heap ?*

Solution. No, the sequence doesn't fulfill the heap-property since the element 7 is below element 6.



6.4 Heapify

Maintaining the heap property

Heapify is a procedure for manipulating heap data structures. It is given an array A and index i into the array. The subtree rooted at the children of $A[i]$ are heap but node $A[i]$ itself may possibly violate the heap property *i.e.*, $A[i] < A[2i]$ or $A[i] < A[2i+1]$. The procedure 'Heapify' manipulates the tree rooted at $A[i]$ so it becomes a heap. In other words, 'Heapify' is let the value at $A[i]$ "float down" in a heap so that subtree rooted at index i becomes a heap.

Outline of procedure heapify

Heapify picks the largest child key and compares it to the parent key. If parent key is larger then heapify quits, otherwise it swaps the parent key with the largest child key. So that the parent now becomes larger than its children. It is important to note that swap may destroy the heap property of the subtree rooted at the largest child node. If this is the case, heapify calls itself again using largest child node as the new root.

```
MAX-HEAPIFY (A, i)
1.   l ← left [i]
2.   r ← right [i]
3.   if l ≤ heap-size [A] and A[l] > A[i]
4.      then largest ← l
5.      else largest ← i
6.   if r ≤ heap-size [A] and A[r] > A [largest]
7.      then largest ← r
8.   if largest ≠ i
9.      then exchange A[i] ↔ A [largest]
10.        MAX-HEAPIFY (A, largest)
```

Analysis

If we put a value at root that is less than every value in the left and right subtree, then 'Heapify' will be called recursively until leaf is reached. To make recursive calls traverse the longest path to a leaf, choose value that makes 'Heapify' always recurs on the left child. It follows the left branch when left child is greater than or equal to the right child, so putting 0 at the root and 1 at all other nodes, for example, will accomplish this task. With such values 'Heapify' will be called h times, where h is the heap height so its running time will be $\theta(h)$ (since each call does $\theta(1)$ work), which is $\theta(\lg n)$. Since we have a case in which Heapify's running time $\theta(\lg n)$, its worst-case running time is $\Omega(\lg n)$.

Example : Suppose we have a complete binary tree.

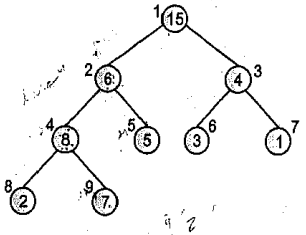


Figure 6.6

In this complete binary tree, the subtrees of 6 are not max-heap. So we call MAX-HEAPIFY (A, 2)

$l = 4$,

$r = 5$: and $i = 2$

Here heap-size $(A) = 9$

$l \leq$ heap-size and $A[l] = 8$ and $A[i] = 6$

So, $A[l] > A[i]$

then largest $= 4$

$r \leq$ heap-size $[A]$ and $A[r] = 5$

$A[r] < A$[largest]

Here, largest $\neq i$

then $A[2] = 8$ and $A[4] = 6$ after exchanging $A[i]$ and A [largest]

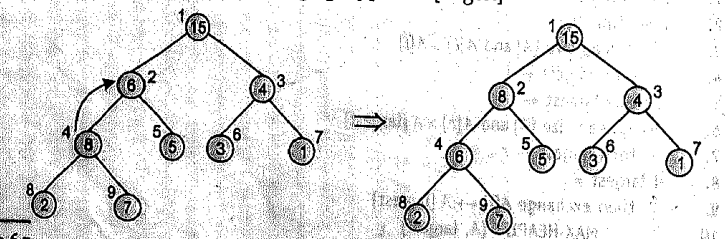


Figure 6.7

Now, call MAX-HEAPIFY $(A, 4)$

$l = 8$

$r = 9$ and $A[l] = 2$

$A[r] = 7$

$A[i] = 6$

$l \leq$ heap-size $[A]$ and $A[l] < A[i]$ so largest $= i$

$r \leq$ heap-size $[A]$ and $A[r] > A$ [largest] so largest $= r$

largest $\neq i$ then exchange A [largest] $\leftrightarrow A[i]$

Thus the final tree is

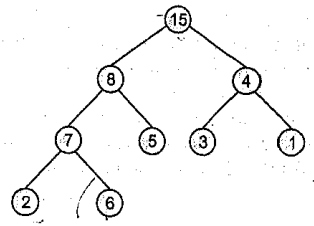


Figure 6.8

Example. *What is the effect of calling MAX-HEAPIFY (A, i) when the element $A[i]$ is larger than its children ?*

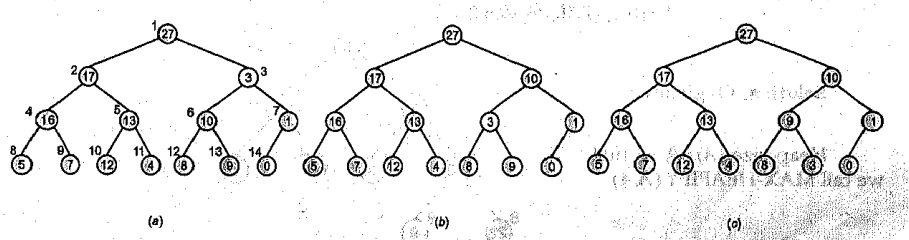
Solution. Nothing happens.

Example. *What is the effect of calling MAX-HEAPIFY (A, i) for $I >$ heap-size $[A]/2$?*

Solution. Nothing happens (l and $r >$ heap-size $[A]$).

Example. *Using Fig. illustrate the operation of MAX-HEAPIFY $(A,3)$ on the array* $A = \langle 27,13,3,16,13,10,1,5,7,12,4,8,9,0 \rangle$

Solution. The following happens when MAX-HEAPIFY $(A,3)$ is called.



(a) (b) (c)

6.5 Building a Heap

We can use the procedure 'Heapify' in a bottom-up fashion to convert an array $A[1..n]$ into a heap.

```
BUILD-MAX-HEAP (A)
1.   heap-size (A) ← length [A]
2.   For i ← floor(length [A]/2) down to 1 do
3.      MAX-HEAPIFY (A, i)
```

We can build a heap from an unordered array in linear time.

6.6 Heap Sort Algorithm

The heap sort combines the best of both merge sort and insertion sort. Like merge sort, the worst case time of heap sort is $O(n \log n)$ and like insertion sort, heap sort sorts in-place. The heap sort algorithm starts by using procedure MAX-BUILD-HEAP to build a heap on the input array $A[1..n]$. Since the maximum element of the array stored at the root $A[1]$, it can be put into its correct final position by exchanging it with $A[n]$ (the last element in A). If we now discard node n from the heap then the remaining elements can be made into heap. Note that the new element at the root may violate the heap property. All that is needed to restore the heap property.

```
HEAP-SORT(A)
1.   BUILD-MAX-HEAP (A)
2.   for i ← length [A] down to 2
3.      do exchange A[1] ↔ A[i]
4.         heap-size [A] ← heap-size [A] –1
5.         MAX-HEAPIFY (A, 1)
```

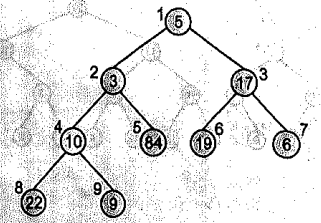
Analysis

We have seen that the running time of 'Build-heap' is $O(n)$. The heap-sort algorithm makes a call to 'Build-heap' for creating a (max) heap, which will take $O(n)$ time and each of the $(n-1)$-calls to Max-heapify to fix up the new heap (which is created after exchanging the root and by decreasing the heap size). We know 'MAX-HEAPIFY' takes time $O(\lg n)$.

Thus the total running time for the heap-sort is $O(n \lg n)$.

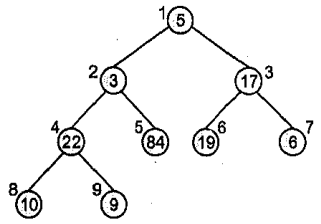
Example. *Using Fig. illustrate the operation of BUILD-MAX-HEAP on the array* $A = \langle 5,3,17,10,84,19,6,22,9 \rangle$.

Solution. Originally :

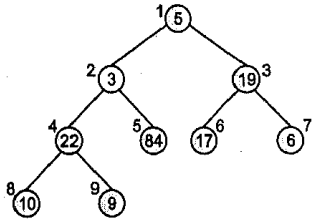


Heap-size $(A) = 9$, so first we call MAX-HEAPIFY $(A, 4)$

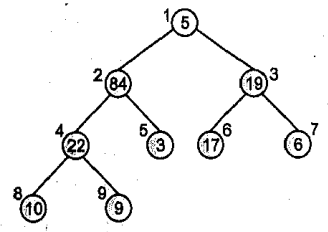
After MAX-HEAPIFY $(A, 4)$



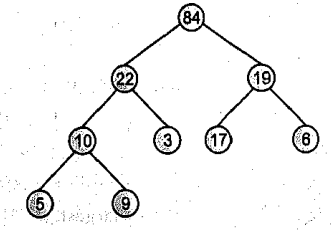
After MAX-HEAPIFY $(A, 3)$



After MAX-HEAPIFY $(A, 2)$

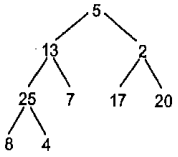


After MAX-HEAPIFY $(A, 1)$



Example. *Illustrate the operation of HEAP-SORT on the array* $A=\langle 5,13,2,25,7,17,20,8,4\rangle$

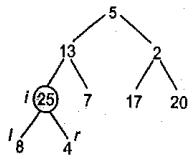
Solution. Originally,



First we call BUILD-MAX-HEAP

heap-size $(A) = 9$

So, $i = 4$ to 1. Call Max-Heapify (A, i).

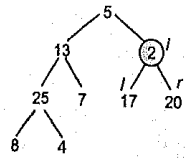


i.e., first we call MAX-HEAPIFY $(A,4)$

$A[l]=8 \quad A[i]=25 \quad A[r]=4$

$A[i] > A[l]$

$A[i] > A[r]$



Now we call MAX-Heapify $(A,3)$.

$A[i]=2 \quad A[l]=17 \quad A[r]=20$

$A[l] > A[i]$

$\therefore$ largest $= 6$

$A[r] > A[\text{largest}]$

$20 > 17$

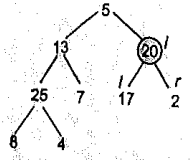
$\therefore$ largest $= 7$

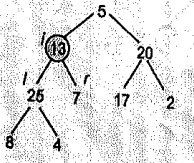
largest $\neq i$

$\therefore$ $A[l] \leftrightarrow A[\text{largest}]$ *i.e.*, now

$A[i] > A[l]$

$A[i] > A[r]$





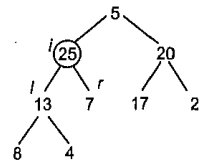
Now, we call MAX-HEAPIFY $(A, 2)$, *i.e.*,

$A[i] < A[l]$

So, largest $= 4$

$A[\text{largest}] > A[r]$

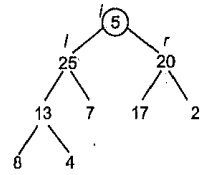
$\therefore$ $i \neq$ largest, so $A[i] \leftrightarrow A[\text{largest}]$



Now,

$A[i] > A[l]$

$A[i] > A[r]$



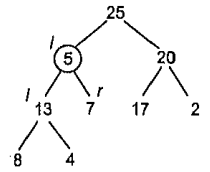
We call MAX-HEAPIFY $(A,1)$.

$A[i] < A[l]$

largest $= 2$

$A[\text{largest}] > A[r]$ and largest $\neq i$

$A[i] \leftrightarrow A[\text{largest}]$

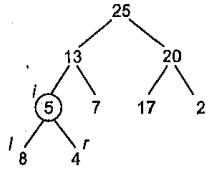


largest $= 2$, so $i = 2$

$A[i] < A[l]$ then largest $= 4$

and $A[\text{largest}] > A[r]$, largest $\neq i$

$A[i] \leftrightarrow A[\text{largest}]$



Now,

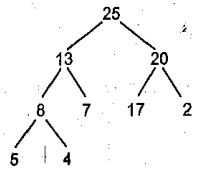
$A[i] < A[l]$

largest $= 8 \quad A[\text{largest}] > A[r]$

largest $\neq i$

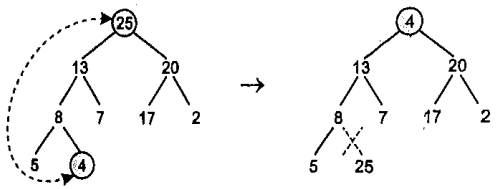
$A[\text{largest}] \leftrightarrow A[i]$

So, this is the final tree after BUILD-MAX-HEAP.

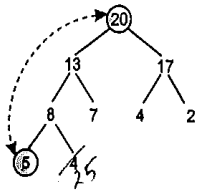


Now, $i = 9$ down to 2 exchange $A[i] \leftrightarrow A[i]$ and size = size -1 and call MAX-HEAPIFY $(A,1)$ each time.

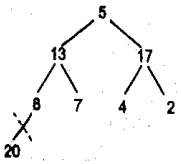
Exchanging $A[1] \leftrightarrow A[9]$, we get



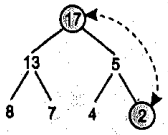
Now call MAX-HEAPIFY (A,1), we get



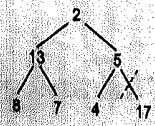
Now, exchange $A[1]$ and $A[8]$ and size = size $-1 = 8-1 = 7$



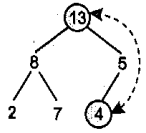
Again, call MAX-HEAPIFY (A, 1), we get



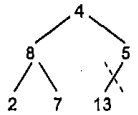
Exchange $A[1]$ and $A[7]$ and size = size $-1 = 7-1 = 6$.

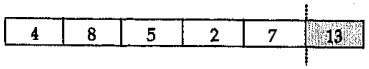


Again, call MAX-HEAPIFY (A, 1), we get

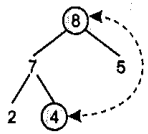


Exchange $A[1]$ and $A[6]$ and now size $= 6-1 = 5$.

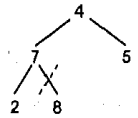


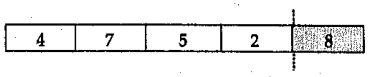


Again call MAX-HEAPIFY (A,1), we get

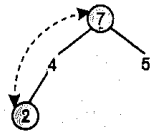


Exchange $A[1]$ and $A[5]$ and now size $= 5-1 = 4$.

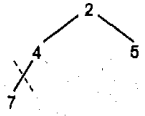




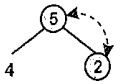
Again, call MAX-HEAPIFY (A, 1), we get



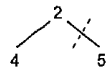
Exchange $A[1]$ and $A[4]$ and size $= 4-1 = 3$.

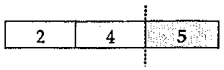


Call MAX-HEAPIFY $(A,1)$, we get



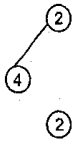
Exchange $A[1]$ and $A[3]$ and size $=3-1=2$.

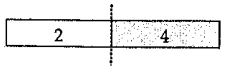




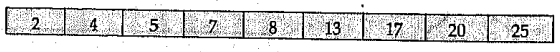
Call, MAX-HEAPIFY $(A,1)$, we get

Exchange $A[1]$ and $A[2]$ and size $=2-1=1$.





Thus the sorted array.



Example. *What is the running time of heap sort on an array A of length n that is already sorted in increasing order ? What about decreasing order ?*

Solution.

```
HEAP-SORT (A)
1.    BUILD-MAX-HEAP (A)
2.    for i ← length [A] down to 2
3.        do exchange A[1] ↔ A[i]
4.           heap-size [A] ← heap-size [A] − 1
5.           MAX-HEAPIFY (A, 1)
```

- **Increasing order.** Line 1 takes $O(n)$ time (worst case) while line 5 takes $O(\lg n)$ time since MAX-HEAPIFY must do $\lg n$ exchange. All together,

$$T(n) = O(n) + nO(\lg n) = O(n \lg n).$$

- **Decreasing order.** Line 1 results in $O(n/2)$ calls to MAX-HEAPIFY without any work *i.e.*, $O(n/2)$ in total. Line 5 costs just as much as in increasing order so in total we get

$$T(n) = O(n/2) + nO(\lg n) = O(n \lg n)$$

6.7 Heap-Insert

The procedure MAX-HEAP-INSERT implements the INSERT operation. It takes the key of new element as the input into max-heap A. The procedure first expands the max-heap by adding to the tree a new leaf whose key is $-\infty$. Then it calls HEAP-INCREASE-KEY to set the key of this new node to its correct value and maintain the max-heap property.

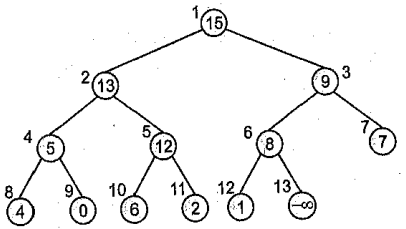
```
MAX-HEAP-INSERT (A, key)
1.    heap-size [A] ← heap-size [A] + 1
2.    A [heap-size [A] ] ← −∞
3.    HEAP-INCREASE-KEY (A, heap-size [A], key)

HEAP-INCREASE-KEY (A, i,. key)
1.    if key < A[i]
2.        then error "new key is smaller than current key".
3.    A[i] ← key
4.    while i > 1 and A [PARENT (i) < A[i] ]
5.        do exchange A[i] ↔ A [PARENT (i)]
6.           i ← PARENT (i)
```

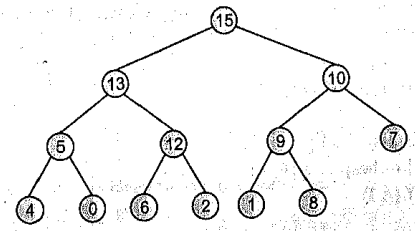
The running time of MAX-HEAP-INSERT on an n-element heap is $O(\lg n)$.

Example. *"Illustrate the operation of MAX-HEAP-INSERT $(A,10)$ on the heap $A = \langle 15,13,9,5,12,8,7,4,0,6,2,1 \rangle$.*

Solution. The Figures show the heap before and after the call HEAP-INCREASE-KEY $(A$, heap-size $[A]$, 10) which is called MAX-HEAP-INSERT $(A,10)$.



Increase heap size *i.e.*, $12+1=13$ and $A[13]=-\infty$



Final Heap

Example. *"The procedure BUILD-MAX-HEAP in section 6.5 can be implemented by repeatedly using MAX-HEAP-INSERT to insert the elements into the heap. Consider the following :*

```
BUILD-MAX-HEAP' (A)
1. heap-size [A] ← 1
2. for i ← 2 to length [A]
3.          do MAX-HEAP-INSERT (A, A[i])
```

(a) Do the procedures BUILD-MAX-HEAP and BUILD-MAX-HEAP' always create the same heap when run on the same input array ? Prove that they do, or provide a counterexample.

(b) Show that in the worst case, BUILD-MAX-HEAP' requires $\Theta(n \lg n)$ time to build an n-element heap."

Solution. The original algorithm to build a max-heap is

```
BUILD-MAX-HEAP (A)
1. heap-size [A] ← length [A]
2. for i ← ⌊length [A] / 2⌋ down to 1
3.      do MAX-HEAPIFY (A, i).
```

(a) No, they do not always create the same heap.

e.g., $\langle 0,1,2,3,4,5 \rangle$.

(b) The worst case occurs when MAX-HEAP-INSERT must let the element float all the way up to the root. This scenario can be provoked every time by presenting an array of decreasing values $\langle n, n-1, n-2, \ldots 1 \rangle$

Then, $T(n) = \theta(n \lg n)$.

6.8 Heap-Delete

HEAP-DELETE (A, i) is the procedure, which deletes the item in node i from heap A. HEAP-DELETE runs in $O(\lg n)$ time for an n-element max-heap.

```
HEAP-DELETE (A, i)
1. A[i] ← A [heap-size [A]]
2. heap-size [A] ← heap-size [A] – 1
3. MAX-HEAPIFY (A, i)
```

6.9 Heap-Extract-Max

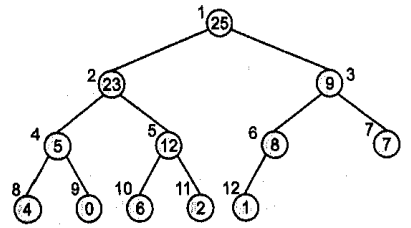
This operation removes and returns the element having largest key value from the set.

```
HEAP-EXTRACT-MAX (A)
1. IF heap-size [A] < 1
2.      then error "heap underflow"
3. max ← A[1]
4. A[1] ← A [heap-size [A]]
5. heap-size [A] ← heap-size [A] –1
6. MAX-HEAPIFY (A, 1)
7. return max.
```


Example. *Illustrate HEAP-EXTRACT-MAX on the heap.*

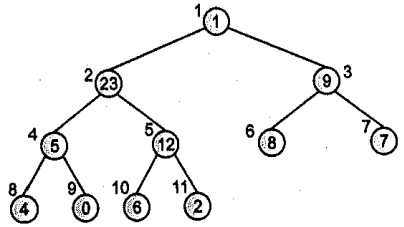
$A = \langle 25,23,9,5,12,8,7,4,0,6,2,1 \rangle$

Solution.

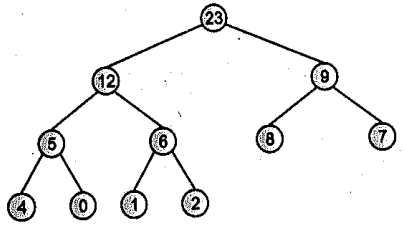


$\max \leftarrow A[1]$

Thus $\max = 25$ and $A[1] \leftarrow A[12]$, Now heap size $[A] = 12 - 1 = 11$



Now, call MAX-HEAPIFY $(A, 1)$ and readjust the heap and the final heap is as follows :



Exercise

1. Create a Min-Heap and Max-Heap for the following list

   (a) $L = \langle G, F, D, C, B, A, H, I, J, K \rangle$

(b) $L = \langle 20, 10, 1, 5, 4, 80, 60, 30 \rangle$

(c) $L = \langle 5, 15, 8, 3, 9, 65 \rangle$

(d) $L = \langle A, B, G, H, D, X \rangle$

2. Suppose the elements in an array are (starting at index 1) $\langle 25, 19, 15, 5, 12, 4, 13, 3, 7, 10 \rangle$
   Does this array represent a heap ? Justify your answer.
3. Sort the following array using Heap-Sort techniques.
   $\langle 5, 8, 3, 9, 2, 10, 1, 45, 32 \rangle$
4. Suppose the array to be sorted (into alphabetical order) by Heap-sort initially contains the following sequence of letters :
   ALGORITHMS
   show how they would be arranged in the array after the heap construction phase. (Build-Heap).
   How many key comparisons are done to construct the heap with these keys ?
5. Illustrate the performance of the heap-sort algorithm on the following input sequence ;
   $\langle 2, 5, 16, 4, 10, 23, 39, 18, 26, 15 \rangle$

CHAPTER 7

Quick Sort

7.1 Introduction

The basic version of quick sort algorithm was invented by C. A. R. Hoare in 1960 and formally introduced quick sort in 1962. It is used on the principle of divide-and-conquer. Quick sort is an algorithm of choice in many situations because it is not difficult to implement. It is a good "general purpose" sort and it consumes relatively fewer resources during execution.

Advantages

- It is in-place since it uses only a small auxiliary stack.
- It requires only $n \log (n)$ time to sort n items.
- It has an extremely short inner loop.
- This algorithm has been subjected to a thorough mathematical analysis ; a very precise statement can be made about performance issues.

Disadvantages

- It is recursive. Especially if recursion is not available, the implementation is extremely complicated.
- It requires quadratic (*i.e.*, n^2) time in the worst-case.
- It is fragile *i.e.*, a simple mistake in the implementation can go unnoticed and cause it to perform badly.

Quick sort works by partitioning a given array $A[p..r]$ into two non-empty sub arrays $A[p..q]$ and $A[q+1..r]$ such that every key in $A[p..q]$ is less than or equal to every key in $A[q+1..r]$. Then the two sub arrays are sorted by recursive calls to Quick sort. The exact position of the partition depends on the given array and index q is computed as a part of the partitioning procedure.

```
QUICK-SORT (A, p, r)
1.  If p < r then
2.      q ←PARTITION (A, p, r)
3.      QUICK SORT (A, p, q − 1)
4.      QUICK SORT (A, q + 1, r)
```

Note that to sort entire array, the initial call is Quick Sort (*A*, 1, length [A])

As a first step, Quick Sort chooses as pivot one of the items in the array to be sorted. Then array is partitioned on either side of the pivot. Elements that are less than or equal to pivot will move toward the left and elements that are greater than or equal to pivot will move toward the right.

7.2 Partitioning the Array

Partitioning procedure rearranges the sub arrays in-place.

```
PARTITION (A, p, r)
1.  x ← A[r]
2.  i ← p − 1
3.  for j ← p to r − 1
4.      do if A[j] ≤ x
5.          then i ← i + 1
6.              exchange A[i] ↔ A[j]
7.  exchange A[i + 1] ↔ A[r]
8.  return i + 1
```

Partition selects the first key, $A[p]$ as a pivot key about which the array will be partitioned :

- Keys $\le A[p]$ will be moved towards the left .
- Keys $\ge A[p]$ will be moved towards the right.

The running time of the partition procedure is $\theta(n)$ where $n = r - p + 1$ which is the number of keys in the array.

Another argument that running time of PARTITION on a sub array of size $\theta(n)$ is as follows: Pointer i and pointer j start at each end and move towards each other, conveying somewhere in the middle. The total number of times that i can be incremented and j can be decremented is therefore $O(n)$. Associated with each increment or decrement there are $O(1)$ comparisons and swaps. Hence, the total time is $O(n)$.

Example. "Using Fig. as a model, illustrate the operation of PARTITION on the array

$A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21 \rangle$".

13	19	9	5	12	8	7	4	11	2	6	21
13	19	9	5	12	8	7	4	11	2	6	21
13	19	9	5	12	8	7	4	11	2	6	21
13	19	9	5	12	8	7	4	11	2	6	21
13	19	9	5	12	8	7	4	11	2	6	21
13	19	9	5	12	8	7	4	11	2	6	21
13	19	9	5	12	8	7	4	11	2	6	21
13	19	9	5	12	8	7	4	11	2	6	21
13	19	9	5	12	8	7	4	11	2	6	21
13	19	9	5	12	8	7	4	11	2	6	21
13	19	9	5	12	8	7	4	11	2	6	21
13	19	9	5	12	8	7	4	11	2	6	21
13	19	9	5	12	8	7	4	11	2	6	21

Example. *What value of q does PARTITION return when all elements in the array $A[p..r]$ have the same value?*

Solution. PARTITION will return r, identical elements are the worst-case for QUICK SORT since PARTITION will make the worst split : i and j both move to the right while elements are swapped with themselves.

Example. *Give a brief argument that the running time of PARTITION on a sub array of size n is $\theta(n)$.*

Solution. By examining the code we can see that

- Each element $A[I]$ is composed with x exactly once ;
- Each element $A[I]$ is swapped at most once.

Taken together, the running time must therefore be $\theta(n)$.

7.3 Performance of Quick Sort

The running time of quick sort depends on whether partition is balanced or unbalanced, which in turn depends on which elements of an array to be sorted are used for partitioning.

A very good partition splits an array up into two equal sized arrays. A bad partition, on other hand, splits an array up into two arrays of very different sizes. The worst partition puts only one element in one array and all other elements in the other array. If the partitioning is balanced, the Quick sort runs asymptotically as fast as merge sort. On the other hand, if partitioning is unbalanced, the Quick sort runs asymptotically as slow as insertion sort.

Best Case

The best thing that could happen in Quick sort would be that each partitioning stage divides the array exactly in half. In other words, the best to be a median of the keys in $A[p..r]$ every time procedure 'Partition' is called. The procedure 'Partition' always splits the array to be sorted into two equal sized arrays. If the procedure 'Partition' produces two regions of size $n/2$, the recurrence relation is then

$$T(n) = T(n/2) + T(n/2) + \theta(n)$$
$$= 2T(n/2) + \theta(n)$$

And from case 2 of Master theorem

$$T(n) = \theta(n \lg n)$$

Worst case Partitioning

The worst-case occurs if given array $A[1..n]$ is already sorted. The PARTITION (A, p, r) call always returns p so successive calls to partition will split arrays of length n, $n-1$, $n-2$,...,2 and running time proportional to $n+(n-1)+(n-2)+...+2 = [(n+2)(n-1)/2] = \theta(n^2)$. The worst-case also occurs if $A[1..n]$ starts out in reverse order.

Example. *Show that the running time of QUICK-SORT is $\theta(n^2)$ when the array A contains distinct elements and is sorted in decreasing order.*

Solution. We have thus an array whose elements are sorted in decreasing order, for instance $\langle 8,6,4,1 \rangle$.

```
QUICK-SORT (A, p, r)
1.  if p < r then
2.      q ← PARTITION (A, p, r)
3.      QUICK-SORT (A, p, q − 1)
4.      QUICK-SORT (A, q + 1, r)
```

Since the pivot element is the leftmost element, PARTITION will make a lousy partitioning. Line 2 takes $\theta(n)$ time and sets $q \leftarrow p$. Line 3 takes $T(1)$ time. Line 4 takes $T(n-1)$ time.

In total, we get the recurrence

$$T(n) = T(n-1) + \theta(n)$$

Thus, $T(n) = \theta(n^2)$, by solving with recursion tree.

Example. *"Banks often record transaction on an account in order of the times of the transactions, but many people like to receive their bank statements with checks listed in order by check number. People usually write check in order by check number, and merchants usually cash them with reasonable dispatch. The problem of converting time-of-transaction ordering to check-number ordering is therefore the problem of sorting almost-sorted input. Argue that the procedure INSERTION-SORT would tend to beat the procedure QUICK-SORT on this problem."*

Solution. "Almost sorted" is almost the worst case for QUICK-SORT that is $\theta(n^2)$ and "Almost sorted" is almost the best case for INSERTION-SORT *i.e.*, $\theta(n)$.

Thus, in this problem, insertion-sort beats the Quick-sort.

Example. *"Show that quick sort's best-case running time is $\Omega(n \lg n)$.*

Solution. Best case : $T(n) = \min_{1 \le q \le n-1} (T(q) + T(n-q)) + \theta(n)$

Guess that $T(n) \ge cn \lg n$ for some constant c

$$T(n) \ge \min_{1 \le q \le n-1} (cq \lg q + c(n-q) \lg (n-q)) + \theta(n)$$

$$= c \min_{1 \le q \le n-1} (q \lg q + (n-q) \lg (n-q)) + \theta(n)$$

Now, what is the minimum value of the function

$$f(q) = q \lg q + (n-q) \lg (n-q) ?$$

For $q = 1$

$$f(q) = (n-1) \lg (n-1)$$

For $q = n-1$

$$f(q) = (n-1) \lg (n-1)$$

For $q = \frac{n}{2}$

$$f(q) = n \lg \left(\frac{n}{2}\right) < n \lg n.$$

Thus, the minimum appears somewhere between.

$$q = 1 \text{ and } q = n-1.$$

Now, $$f'(q) = \frac{1}{q} q + \lg q + (n-q) \frac{1}{(n-q)} (-1) + (-1) \lg (n-q)$$

$$= 1 + \lg (q-1) - \lg (n-q)$$

$$= \lg q - \lg (n-q)$$

For minimum

$$f'(q)=0$$

i.e., $\lg q-\lg(n-q)=0$

which gives $q=\frac{n}{2}$

Put $q=\frac{n}{2}$ into the original equation.

$$T(n)\geq c\left(n\lg\left(\frac{n}{2}\right)\right)+\theta(n)$$

$$=cn\lg n-cn\lg 2+\theta(n)$$

$$\geq cn\lg n$$

if we choose the constant c so that the constant in $\theta(n)>c\lg 2$.

Thus, quick-sort's best-case execution time is $\Omega(n\lg n)$.

Example. *"The running time of QUICK SORT can be improved in practice by taking advantage of the fast running time of insertion sort when its input is "nearly" sorted. When quick sort is called on a sub array with fewer than k elements, let it simply return without sorting the sub array.*

After the top-level call to quick sort return, run insertion sort on the entire array to finish the sorting process. Argue that this sorting algorithm runs in $O(nk+n\lg(n/k))$ expected time. How should k be picked, both in theory and in practice ?"

Solution. QUICK SORT without sorting sub arrays with less than k elements

$$T(n)=1 \text{ if } n<k$$

$$T(n)=2T(n/2)+n \text{ otherwise}$$

Expand the recurrence a few times.

$$T(n)=2T(n/2)+n$$
$$=n+2T(n/2)$$
$$=n+2(2T(n/2)+n/2)$$
$$=4T(n/4)+2n$$
$$=8T(n/8)+3n$$
$$\vdots$$

This goes on x times until $\frac{n}{2}x=k$ *i.e.,* $x=\lg\left(\frac{n}{k}\right)$

For the QUICK-SORT-part we get $T(n)=O(n\lg(n/k))$. The sorting is finished with an insertion sort where the number of elements is n elements, though divided up into blocks of k. It is important to notice that the elements do not move between blocks. Thus, we have n/k blocks that each can be sorted in $O(k^2)$ (worst-case), which gives $O(nk)$ time for the whole array.

The total becomes $O(nk)+O(n\lg(n/k))$

$$=O(nk+n\lg(n/k))$$

One should, of course pick k as to minimize the running time. For a certain n one can study the minimum of the function $f(n)=nk+n\lg(n/k)$.

In practice, one would run a series of bench marks to conclude the best k for various inputs. It suggests $k=9$.

Example. *"Professors Howard, Fine and Howard have proposed the following "elegant" sorting algorithm.*

```
STOOGE-SORT (A, i, j)
1.  if A[i] > A[j]
2.      then exchange A[i] ↔ A[j]
3.  if i + 1 ≥ j
4.      then return
5.  k ← ⌊(j − i + 1) / 3⌋          ▷ Round down
6.  STOOGE-SORT (A, i, j − k)      ▷ First two-thirds
7.  STOOGE-SORT (A, i + k, j)      ▷ Last two-thirds.
8.  STOOGE-SORT (A, i, j − k)      ▷ First two-thirds again.
```

(*a*) *Give a recurrence for the worst-case running time of STOOGE-SORT and a tight asymptotic (θ-notation) bound on the worst-case running time.*

(*b*) *Compare the worst-case running time of STOOGE-SORT with that of insertion sort, merge sort, heap sort, and quick sort. Do the professors deserve tenure ?"*

Solution. (*a*) The recurrence for STOOGE-SORT :

$$T(1)=\theta(1)$$

$$T(n)=3T(2n/3)+\theta(1) \text{ for } n>1$$

Apply master method.

$$f(n)=1 \quad a=3 \quad b=3/2 \quad \Rightarrow \quad n^{\log_b a}=n^{\log_{3/2}3}$$

$$f(n)=1=O(n^{\log_{3/2}3-\epsilon}) \quad \text{for} \quad \epsilon=\log_{3/2}3.$$

which gives $T(n)=\theta(n^{\log_{3/2}3})$

$$\approx\theta(n^{2.7})$$

(*b*) Worst-case for a few sorting algorithms :

Insertion sort	:	$\theta(n^2)$
Merge sort	:	$\theta(n\lg n)$
Heap sort	:	$\theta(n\lg n)$
Quick sort	:	$\theta(n^2)$
Stooge sort	:	$\theta(n^{2.7})$

So, no tenure.

7.4 Versions of Quick Sort

Version 1. *Hoare's Algorithm*

```
QUICK SORT
    if p < r
    then q ← PARTITION (A, p, r)
        QUICK SORT (A, p, q)
        QUICK SORT (A, q + 1, r)
```

```
PARTITION (A, p, r)
1.  x ← A[p]
2.  i ← p − 1
3.  j ← r + 1
4.  while TRUE
5.      do repeat j ← j − 1
6.          until A[j] ≤ x
7.      repeat i ← i + 1
8.          until A[i] ≥ x
9.  if i < j
10.     then exchange A[i] ↔ A[j]
11. else return j
```

Technicalities :

- Pivot must be $A[p]$, not $A[r]$, if pivot is $A[r]$ and $A[r]$ contains the largest element in the array, then quick sort will loop forever.
- **Problem.** This version of partition can lead to one of the partitions containing 1 element.
  - This can happen when the smallest element is in $A[p]$.
  - When array is originally in sorted order, this leads to the worst case.

Version 2. *Lomuto's Algorithm*

```
QUICK SORT (A, p, r)
1.  if p < r
2.  then q ← PARTITION (A, p, r)
3.      QUICK SORT (A, p, q − 1)
4.      QUICK SORT (A, q + 1, r)
```



```
PARTITION (A, p, r)
1.  x ← A[r]
2.  i ← p − 1
3.  for j ← p to r − 1
4.      do if A[j] ≤ x
5.          then i ← i + 1
6.              exchange A[i] ↔ A[j]
7.  exchange A[i + 1] ↔ A[r]
8.  return i + 1
```

Version 3. Randomized Quick Sort

In the randomized version of Quick sort we impose a distribution on input. This does not improve the worst-case running time independent of the input ordering.

In this version we choose a random key for the pivot. Assume that procedure Random (a, b) returns a random integer in the range $[a, b)$; there are $b-a+1$ integers in the range and procedure is equally likely to return one of them. The new partition procedure, simply implemented the swap before actually partitioning.

```
RANDOMIZED-PARTITION (A, p, r)
1.  i ← RANDOM (p, r)
2.  exchange A[p] ↔ A[i]
3.  return PARTITION (A, p, r)
```

Now randomized quick sort call the above procedure in place of PARTITION

```
RANDOMIZED-QUICK SORT (A, p, r)
1.  if p < r
2.      then q ← RANDOMIZED-PARTITION (A, p ,r)
3.          RANDOMIZED-QUICK SORT (A, p, q)
4.          RANDOMIZED-QUICK SORT (A, q + 1, r)
```

Like other randomized algorithms, RANDOMIZED-QUICK SORT has the property that no particular input elicits its worst-case behavior; the behavior of algorithm only depends on the random-number generator. Even intentionally, we cannot produce a bad input for RANDOMIZED - QUICK SORT unless we can predict generator will produce next.

Version 4. MEDIAN-OF-3 PARTITION

```
Median-of-3-Partition (A, p, r)
1.  i ← RANDOM (p, r)
2.  j ← RANDOM (p, r)
3.  k ← RANDOM (p, r)
```

```
4.  if (A[i] ≤ A[j] and A[j] ≤ A[k]) or (A[k] ≤ A[j] and A[j] ≤ A[i])
5.      then l ← j
6.  else if (A[j] ≤ A[i] and A[i] ≤ A[k]) or (A[k] ≤ A[i] and A[i] ≤ A[j])
7.      then l ← i
8.  else if (A[i] ≤ A[k] and A[k] ≤ A[j]) or (A[j] ≤ A[k] and A[k] ≤ A[i])
9.      then l ← k
10. exchange A[r] ↔ A[l]
11. return partition (A, p, r)
```

One common approach in median-of-3 method is choose the pivot as the median (middle element) of a set of 3 elements randomly selected. The median-of-3 partition algorithm makes it very improbable that the worst case partitioning will occur in every recursive call of quick sort. Thus this algorithm gives very good results in all but a few isolated cases.

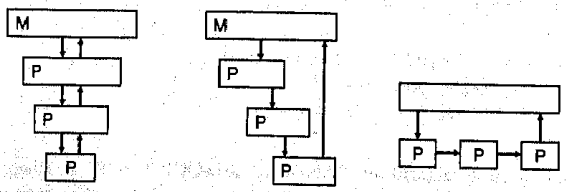
Stability

Quick Sort is not stable.

Tail Recursion

In the simple stack implementation of recursion, the local variables of the function will be pushed onto the stack as the recursive call is initiated. When the recursive call terminates, these local variables will be poped from the stack and thereby restored to their formal values.

But doing so is pointless if the very last action of a function is to make recursive call to itself as function now terminates and just restored local variables are immediately discarded.



(a) Recursion (b) Tail Recursion (c) Iteration

Thus, the special case when a recursive call is the last executed statement of the function, is called **tail recursion.**

A method used with Quick sort, which avoids the second recursive call, by using an iterative control structure is called tail recursion. A function is tail recursive if it either returns a value without making a recursive call, or returns directly the result of a recursive call. All possible branches of the function must satisfy one of these conditions.

If a function calls itself as the last thing it does then this call could be replaced by a "goto" to the beginning of the function after setting up the arguments correctly. For Dijkstra-aware programmers, this goto can be replaced by a while loop

This technique also works for calls to other functions although it is harder to use with some programming languages. Some compilers do this optimization automatically at higher levels of optimization.

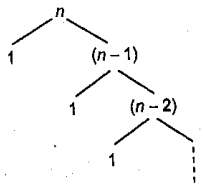
Example. *"Why do we analyze the average-case performance of a randomized algorithm and not its worst-case performance ?"*

Solution. Because the worst-case for a randomized algorithm is the same as the worst-case for the (non-randomized) original algorithm, which is rather a meaningless case to study.

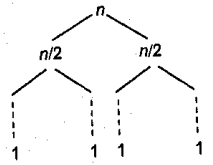
Example. *"During the running of the procedure RANDOMIZED-QUICK-SORT, how many calls are made in the random-number generator RANDOM in the worst case ? How about in the best case ? Give your answer in terms of the θ-notation".*

Solution. RANDOM is called the same number of times as RANDOMIZED-PARTITION.

◂| In the worst case this happens $\theta(n)$ times, the height of the following recursion tree.



◂| In the best case it happens $\theta(\lg n)$ times.



Exercise

1. Show how quick sort the following sequence of keys.
   (a) 2, 3, 18, 17, 5, 1

(*b*) 1, 1, 1, 1, 1, 1, 1

(*c*) 5, 5, 8, 9, 3, 4, 4, 3, 2

(*d*) 1, 3, 5, 7, 9, 12, 15

2. What is the running time of QUICK SORT when all elements of array A have the same value ?
3. Suppose that the splits at every level of quick sort are in the proportion $1-\alpha$ to α, where $0 < \alpha \leq \frac{1}{2}$ is a constant.

   Show that the minimum depth of a leaf in the recursion tree is approximately $-\lg n / \lg \alpha$ and the maximum depth is approximately $-\lg n / \lg(1-\alpha)$.
4. Show that $q^2 + (n-q-1)^2$ achieves a maximum over $q = 0,1,2,\ldots n-1$ when $q = 0$ or $q = n-1$.
5. Show that RANDOMIZED-QUICK SORT's expected running time is $\Omega(n \lg n)$.
6. Show that randomized quick sort runs in $O(n \lg n)$ time with probability $1-\frac{1}{n^2}$.

CHAPTER

8

Sorting in Linear Time

Merge sort, quick sort and heap sort algorithms share an interesting property: the sorted order they determine is based only on comparisons between the input elements. We call such sorting algorithms **comparison sorts**. Any comparison sort must make $\Omega(n \lg n)$ comparisons in the worst case to sort a sequence of n elements.

Heap Sort	$n \log n$
Insertion Sort	n^2
Quick Sort	n^2
Merge Sort	$n \log n$
Randomized Quick Sort	$n \log n$ expected

Figure 8.1

There are sorting algorithms that run faster than $O(n \lg n)$ time but they require special assumptions about the input sequence to be sort. Examples of sorting algorithms that run in linear time are counting sort, radix sort and bucket sort. Counting sort and radix sort assume that the

input consists of integers in a small range. Whereas, bucket sort assumes that a random process that distributes elements uniformly over the interval generates the input. Needless to say, these algorithms use operations other than comparisons to determine the sorted order.

8.1 Stability

We say that a sorting algorithm is *stable* if, when two records have the same key, they stay in their original order. This property will be important for extending bucket sort to an algorithm that works well when k is large. But first, which of the algorithms we've seen is stable ?

- Bucket sort ? Yes. We add items to the lists A[i] in order, and concatenating them preserves that order.
- Heap sort ? No. The act of placing objects into a heap (and heapifying them) destroys any initial ordering they might have.
- Merge sort ? Maybe. It depends on how we divide lists into two, and on how we merge them. For instance if we divide by choosing every other element to go into each list, it is unlikely to be stable. If we divide by splitting a list at its midpoint, and break ties when merging in favor of the first list, then the algorithm can be stable.
- Quick sort ? Again, maybe. It depends on how we do the partition step.

Any comparison-sorting algorithm can be made stable by modifying the comparisons to break ties according to the original positions of the objects, but only some algorithms are automatically stable.

8.2 Counting Sort

Counting sort assumes that each of the n input elements is an integer in the range 1 to k, for some integer k. When $k = O(n)$, the sort runs in $O(n)$ time. The basic idea of counting sort is to determine, for each input element x, the number of elements less than x. This information can be used to place element x directly into its position in the output array. For example, if there are 15 elements less than x, then x belongs in output position 16. This scheme must be modified slightly to handle the situation in which several elements have the same value, since we don't want to put them all in the same position. In the code for counting sort, we assume that the input is an array $A[1..n]$, and thus $length[A] = n$. We require two other arrays: the array $B[1..n]$ holds the sorted output, and the array $C[1..k]$ provides temporary working storage, where k is the highest number in array A.

```
COUNTING-SORT (A, B, k)
1.  for i ← 0 to k
2.      do C[i] ← 0
3.  for j ← 1 to length[A]
4.      do C[A[j]] ← C[A[j]] + 1
5.  /* C[i] now contains the number of elements equal to i. */
```



```
6.  for i ← 1 to k
7.      do C[i] ← C[i] + C[i − 1]
8.  /* C[i] now contains the number of elements less than or equal to i. */
9.  for j ← length[A] down to 1
10.     do B[C[A[j]]] ← A[j]
11.        C[A[j]] ← C[A[j]] − 1
```

An important property of counting sort is that it is ***stable:*** numbers with the same value appear in the output array in the same order as they do in the input array. That is, number appears first in the input array appears first in the output array. Of course, the property of stability is important only when satellite data are carried around with the element being sorted.

Example. *Suppose that the* **for** *loop in line 9 of the COUNTING-SORT procedure is rewritten :*

9 **for** $j \leftarrow 1$ **to** *length* $[A]$

Show that the algorithm still works properly. Is the modified algorithm stable ?

Solution. COUNTING-SORT will work correctly no matter what order A is processed in, however it is not stable. The modification to the for loop actually causes numbers with the same value to appear in reverse order in the output. Running a few examples we can see this.

Example. *Illustrate the operation of COUNTING-SORT on the array*

$$A = \langle 6,0,2,0,1,3,4,6,1,3,2 \rangle.$$

Solution.

	1	2	3	4	5	6	7	8	9	10	11
A	6	0	2	0	1	3	4	6	1	3	2

Here $k = 6$ (highest number in A)

For $i = 0$ to 6

$c[i] = 0$

i.e.,

	0	1	2	3	4	5	6
c	0	0	0	0	0	0	0

For $j \leftarrow 1$ to 11

	0	1	2	3	4	5	6
c	2	2	2	2	1	0	2

For $i = 1$ to 6.

	0	1	2	3	4	5	6
c	2	4	6	8	9	9	11

j	$A[j]$	$C[A[j]]$	$B[C[A[j]]] \leftarrow A[j]$	$C[A[j]] \leftarrow C[A[j]]-1$
11	2	6	$B[6] \leftarrow 2$	$C[2] \leftarrow 5$
10	3	8	$B[8] \leftarrow 3$	$C[3] \leftarrow 7$
9	1	4	$B[4] \leftarrow 1$	$C[1] \leftarrow 3$
8	6	11	$B[11] \leftarrow 6$	$C[6] \leftarrow 10$
7	4	9	$B[9] \leftarrow 4$	$C[4] \leftarrow 8$
6	3	7	$B[7] \leftarrow 3$	$C[3] \leftarrow 6$
5	1	3	$B[3] \leftarrow 1$	$C[1] \leftarrow 2$
4	0	2	$B[2] \leftarrow 0$	$C[0] \leftarrow 1$
3	2	5	$B[5] \leftarrow 2$	$C[2] \leftarrow 4$
2	0	1	$B[1] \leftarrow 0$	$C[0] \leftarrow 0$
1	6	10	$B[10] \leftarrow 6$	$C[6] \leftarrow 9$

	1	2	3	4	5	6	7	8	9	10	11
B	0	0	1	1	2	2	3	3	4	6	6

Thus, the final sorted array is B

8.3 Radix Sort

Radix-Sort is a sorting algorithm that is useful when there is a constant 'd' such that all the keys are d digit numbers. To execute Radix-Sort, for p=1 toward 'd' sort the numbers with respect to the pth digit from the right using any linear-time stable sort.

In a typical computer, which is a sequential random-access machine, radix sort is sometimes used to sort records of information that are keyed by multiple fields. For example, we might wish to sort dates by *three* keys: year, month, and day. We could run a sorting algorithm with a comparison function that, given two dates, compares years, and if there is a tie, compares months, and if another tie occurs, compares days. Alternatively, we could sort the information three times with a stable sort: first on day, next on month, and finally on year.

The code for radix sort is straightforward. The following procedure assumes that each element in the n-element array A has d digits, where digit 1 is the lowest-order digit and digit d is the highest-order digit.

RADIX-SORT (A, d)
1. for $i \leftarrow 1$ to d
2. do use a stable sort to sort array A on digit i

Since a linear-time sorting algorithm is used 'd' times and d is a constant, the running time of Radix-Sort is linear. When each digit is in the range 1 to k, and k is not too large, counting sort is the obvious choice. Each pass over n d-digit numbers then takes time $\theta(n+k)$. There are d passes, so the total time for radix sort is $\theta(dn+kd)$. When d is constant and $k=O(n)$, radix sort runs in linear time.

Example : The first column is the input. The remaining columns show the list after successive sorts on increasingly significant digit positions. The vertical arrows indicate the digit position sorted on to produce each list from the previous one

329		720		720		329
457		355		329		355
657		436		436		436
839	⇒	457	⇒	839	⇒	457
436		657		355		657
720		329		457		720
355		839		657		839
		↑		↑		↑

8.4 Bucket Sort

Bucket sort runs in linear time on the average. Like counting sort, bucket sort is fast because it assumes something about the input. Whereas counting sort assumes that the input consists of integers in a small range, bucket sort assumes that the input is generated by a random process that distributes elements uniformly over the interval $U=[0,1)$.

To sort n input numbers, Bucket-Sort
1. partitions U into n non-overlapping intervals, called buckets,
2. puts each input number into its bucket,
3. sorts each bucket using a simple algorithm, *e.g.* Insertion-Sort, and then,
4. concatenates the sorted lists.

The idea of bucket sort is to divide the interval [0, 1) into n equal-sized subintervals, or **buckets**, and then distribute the n input numbers into the buckets. Since the inputs are uniformly distributed over [0, 1), we don't expect many numbers to fall into each bucket. To produce the output, we simply sort the numbers in each bucket and then go through the buckets in order, listing the elements in each.

Bucket sort assumes that the input is an n-element array A and that each element $A[i]$ in the array satisfies $0 \le A[i] < 1$. The code requires an auxiliary array $B[0..n-1]$ of linked lists (buckets) and assumes that there is a mechanism for maintaining such lists.

BUCKET-SORT (A)
1. $n \leftarrow length\ [A]$
2. for $i \leftarrow 1$ to n
3. do insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$
4. for $i \leftarrow 0$ to $n-1$
5. do sort list $B[i]$ with insertion sort
6. concatenate the lists $B[0]$, $B[1]$,...$B[n-1]$ together in order

To see that this algorithm works, consider two elements $A[i]$ and $A[j]$. If these elements fall in the same bucket, they appear in the proper relative order in the output sequence because

insertion sort sorts their bucket. Suppose they fall into different buckets. Let these buckets be $B[i']$ and $B[j']$, respectively, and assume without loss of generality that $i' < j'$. When the lists of B are concatenated in line 6, elements of bucket $B[i']$ come before elements of $B[j']$, and thus $A[i]$ precedes $A[j]$ in the output sequence. Hence, we must show that $A[i] \le A[j]$. Assuming the contrary, we have

$$\begin{aligned} i' &= \lfloor nA[i] \rfloor \\ &\ge \lfloor nA[j] \rfloor \\ &= j', \end{aligned}$$

which is a contradiction, since $i' < j'$. Thus, bucket sort works.

To analyze the running time, observe that all lines except line 5 take $O(n)$ time in the worst case. The total time to examine all buckets in line 5 is $O(n)$, and so the only interesting part of the analysis is the time taken by the insertion sorts in line 5.

Since insertion sort runs in quadratic time, the expected time to sort the elements in bucket is

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

The total expected time to sort all the elements in all the buckets is therefore linear.

Despite of linear time usually these algorithms are not very desirable from practical point of view. Firstly, the efficiency of linear-time algorithms depend on the keys randomly ordered. If this condition is not satisfied, the result is the degrading in performance. Secondly, these algorithms require extra space proportional to the size of the array being sorted, so if we are dealing with large file, extra array becomes a real liability. Thirdly, the "inner-loop" of these algorithms contain quite a few instructions, so even though they are linear, they would not be as faster than quick sort.

Example. *Show how to improve the worst-case running time of bucket sort to $O(n \lg n)$.*

Solution. Simply replace the insertion sort used to sort the linked lists with some worst case $O(n \lg n)$ sorting algorithm, *e.g.* merge sort. The sorting then takes time:

$$\sum_{i=0}^{n-1} O(n_i \lg n_i) \le \sum_{i=0}^{n-1} O(n_i \lg n) = O(\lg n) \sum_{i=0}^{n-1} O(n_i) = O(n \lg n)$$

The total time of bucket sort is thus $O(n \lg n)$.

Example. Show how to sort n integers in the range 1 to n^2 in $O(n)$ time.

Solution. The number of digits used to represent an n^2 different numbers in a k-ary number system is $d = \log_k (n^2)$. Thus considering the n^2 numbers as radix n numbers gives us that :

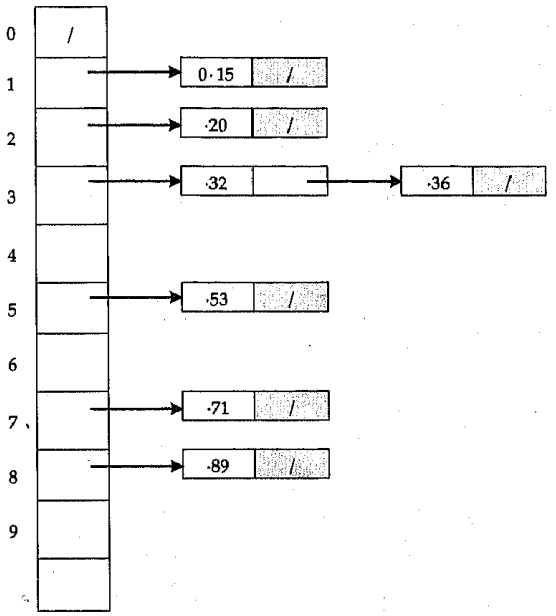
$$d = \log_n (n^2) = 2 \log_n (n) = 2$$

Radix sort will then have a running time of $\theta(d(n+k)) = \theta(2(n+n)) = \theta(n)$.

Example . *Illustrate the operation of BUCKET-SORT on the array*

$$A = \langle 0.36, 0.15, 0.20, 0.89, 0.53, 0.71, 0.32 \rangle$$

Solution.



Now, Sorted array is

0	1	2	3	4	5	6
0.15	0.20	0.32	0.36	0.53	0.71	0.89

Exercise

1. Prove that counting sort is stable ?
2. What is a stable sorting algorithm ? Which of the sorting algorithm we have seen are stable and which are unstable ? Give the names.
3. Illustrate the operation of COUNTING SORT on the array

$$A = \langle 6, 14, 3, 25, 2, 10, 20, 3, 7, 6 \rangle$$

4. Suppose we radix sort n words of b bits each. Each could be viewed as having $\left(\frac{b}{r}\right)$ digits of r bits each, for some integer r.
   Now fill in the boxes :
   (*a*) Each pass of radix sort takes ☐ time.
   (*b*) At min r = ☐
   (*c*) $T(n,b)$ = ☐
5. What is time complexity of COUNTING-SORT ?
   Sort 1 9 3 3 4 5 6 7 7 8 by counting sort.
6. Modify the bucket sort algorithm such that the key of each record has value 0 or 1.

CHAPTER

9

Medians and Order Statistics

Let A be a set containing n distinct orderable elements : The ith order statistic is the number in A that is larger than exactly $i-1$ elements of A. We will consider some special cases of the order statistics problem :

- minimum = 1st order statistic
- maximum = nth order statistic
- median(s) = $\lceil (n+1)/2 \rceil$ and $\lfloor (n+1)/2 \rfloor$

9.1 Selection Problem

The ***selection problem*** can be specified formally as follows :

Input: a set A of n (distinct) numbers and a number i, with $1 \le i \le n$

Output: The element $x \in A$ that is larger than exactly $i-1$ other elements of A.

The selection problem can be solved in $O(n \lg n)$ time, since we can sort the numbers using heap sort or merge sort and then simply index the ith element in the output array.

9.2 Finding Minimum (or Maximum)

```
MINIMUM(A)
1.  lowest ← A[1]
2.  for i ← 2 to n do
3.      lowest ← min( lowest, A[i] )
```

Running Time :

- just scan input array
- exactly $n-1$ comparisons

9.3 Finding Minimum & Maximum Simultaneously

In some applications, we must find both the minimum and the maximum of a set of n elements. For example, a graphics program may need to scale a set of (x,y) data to fit onto a rectangular display screen or other graphical output device. To do so, the program must first determine the minimum and maximum of each coordinate.

It is not too difficult to devise an algorithm that can find both the minimum and the maximum of n elements using the asymptotically optimal $\Omega(n)$ number of comparisons.

Plan A : Find the minimum and maximum separately using $n-1$ comparisons for each $=2n-2$ comparisons.

Plan B : Process elements in pairs. Compare pairs of elements from the input, first with each other, and then compare the smaller to the current min and the larger to the current max, changing current values of max and/or min if necessary. Simultaneous computation of max and min can be done in $3(n-3)/2$ steps.

```
MAX-AND-MIN (A, n)
1.  max ← A[n]; min ← A[n]
2.  for i ← 1 to ⌊n / 2⌋ do
3.      if A[2i − 1] ≥ max then
4.      {   if A[2i − 1] > max then
5.              max ← A[2i − 1]
6.          if A[2i] < min then
7.              min ← A[2i] }
8.      else {  if A[2i] > max then
9.                  max ← A[2i]
10.             if A[2i − 1] < min then
11.                 min ← A[2i − 1]  }
12. return max and min
```

9.4 Selection of *i*th-order Statistic in Linear Time

Randomized-Select returns the ith smallest element of A. RANDOMIZED-SELECT uses the procedure RANDOMIZED-PARTITION .Thus, like RANDOMIZED-QUICKSORT, it is a randomized algorithm, since its behavior is determined in part by the output of a random-number generator. The following code for RANDOMIZED-SELECT returns the ith smallest element of the array $A[p..r]$.

```
RANDOMIZED-SELECT (A, p, r, i)
1.  if p = r
2.      then return A[p]
3.  q ← RANDOMIZED-PARTITION(A, p, r)
```



```
4.  k ← q − p + 1
5.  if i < k
6.      then return RANDOMIZED-SELECT (A, p, q, i)
7.      else return RANDOMIZED-SELECT (A, q + 1, r, i − k)
```

Randomized-Partition first swaps $A[r]$ with a random element of A and then proceeds as in the Partition subroutine of Quick sort.

```
RANDOMIZED-PARTITION (A, p, r)
1.  j ← Random(p, r)
2.  A[r] ↔ A[j]
3.  return Partition (A, p, r)
```

After RANDOMIZED-PARTITION is executed in line 3 of the RANDOMIZED-SELECT algorithm, the array $A[p..r]$ is partitioned into two nonempty subarrays $A[p..q]$ and $A[q+1..r]$ such that each element of $A[p..q]$ is less than each element of $A[q+1..r]$. Line 4 of the algorithm computes the number k of elements in the subarray $A[p..q]$. The algorithm now determines in which of the two subarrays $A[p..q]$ and $A[q+1..r]$ the ith smallest element lies. If $i<k$, then the desired element lies on the low side of the partition, and it is recursively selected from the subarray in line 6. If $i>k$, however, then the desired element lies on the high side of the partition. Since we already know k values that are smaller than the ith smallest element of $A[p..r]$ – namely, the elements of $A[p..q]$ – the desired element is the $(i-k)$th smallest element of $A[q+1..r]$, which is found recursively in line 7.

The worst-case running time for RANDOMIZED-SELECT is $\theta(n^2)$, even to find the minimum, because we could be extremely unlucky and always partition around the largest remaining element. The algorithm works well in the average case, though, and because it is randomized, no particular input elicits the worst-case behavior.

Example. *"Suppose we use RANDOMIZED-SELECT to select the minimum element of the array $A=\langle 3,2,9,0,7,5,4,8,6,1\rangle$. Describe a sequence of partitions that results in a worst-case performance of RANDOMIZED-SELECT."*

Solution. Here is a sequence of partitions that results in a worst–case performance of RANDOMIZED-SELECT :

1	3 2 9 0 7 5 4 8 6 **1**	Choose 9 as a pivot element.
2	3 2 1 0 7 5 4 8 6 **9**	Choose 8 as a pivot element.
3	3 2 1 0 7 5 4 6 **8**	Choose 7 as a pivot element.
4	3 2 1 0 6 5 4 **7**	Choose 6 as a pivot element.
5	3 2 1 0 4 5 **6**	Choose 5 as a pivot element.
6	3 2 1 0 4 **5**	Choose 4 as a pivot element.
7	3 2 1 0 **4**	Choose 3 as a pivot element.
8	0 2 1 **3**	Choose 2 as a pivot element.
9	0 1 **2**	Choose 1 as a pivot element.
10	0 **1**	0 was the minimum element.

9.5 Worst-Case Linear-Time Order Statistics

We now examine a selection algorithm whose running time is $O(n)$ in the worst case. Like RANDOMIZED-SELECT, the algorithm SELECT finds the desired element by recursively partitioning the input array. The idea behind the algorithm, however, is to *guarantee* a good split when the array is partitioned. SELECT uses the deterministic partitioning algorithm PARTITION from quick sort modified to take the element to partition around as an input parameter.

SELECT (i, n)

1. Divide the n elements into groups of 5. Find the median of each 5-element group by rote.
2. Recursively SELECT the median x of the $\lfloor n/5 \rfloor$ group medians to be the pivot.
3. Partition around the pivot x. Let k = rank (x)
4. if $i = k$
   then return x
   elseif $i < k$
   then recursively SELECT the ith smallest element in the lower part
   else
   recursively SELECT the $(i-k)$th smallest element in the upper part

Example. *"In the algorithm SELECT, the input elements are divided into groups of 5. Will the algorithm work in linear time if they are divided into groups of 7 ? Argue that SELECT does not run in linear time if groups of 3 are used."*

Solution.

◂ **Groups of 7.**

The number of elements greater than $x \geq 4\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{7}\right\rceil \right\rceil\right) \geq \frac{4n}{14} - 8$

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq d \\ T\left(\left\lceil \frac{n}{7}\right\rceil\right) + T\left(\frac{10n}{14} + 8\right) + O(n) & \text{if } n > d \end{cases}$$

Suppose $T(n) \leq cn$ for some c and all $n \leq d$

$$\begin{aligned} T(n) &\leq c\left\lceil \frac{n}{7}\right\rceil + c\left(\frac{10n}{14} + 8\right) + O(n) \\ &\leq c\frac{n}{7} + c + \frac{10cn}{14} + 8c + O(n) \\ &\leq \frac{12cn}{14} + 9c + O(n) \\ &\leq cn \end{aligned}$$

if $cn - \frac{12cn}{14} - 9c = c\left(n - \frac{12n}{14} - 9\right) = c\left(\frac{n}{7} - 9\right)$ dominates the constant in $O(n)$ then $T(n) \leq cn$, which happens when $n > 63$, since we can choose c freely.

◂ **Groups of 3.**

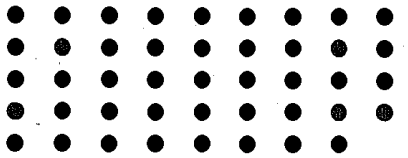
The number of elements greater than $x \geq 2\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{3}\right\rceil \right\rceil - 2\right) \geq \frac{2n}{6} - 4 = \frac{n}{3} - 4$.

Suppose $T(n) \leq cn$ for some c and all $n \leq d$.

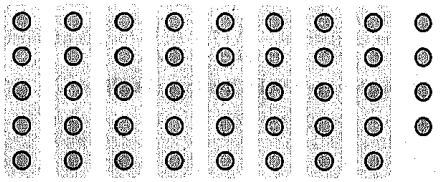
$$\begin{aligned} T(n) &\leq c\left\lceil \frac{n}{3}\right\rceil + c\left(\frac{2n}{3} + 4\right) + O(n) \\ &\leq \frac{cn}{3} + c + \frac{2cn}{3} + 4c + O(n) \\ &\leq cn + 5c + O(n) \not\leq cn \end{aligned}$$

In other words, 3 is too small.

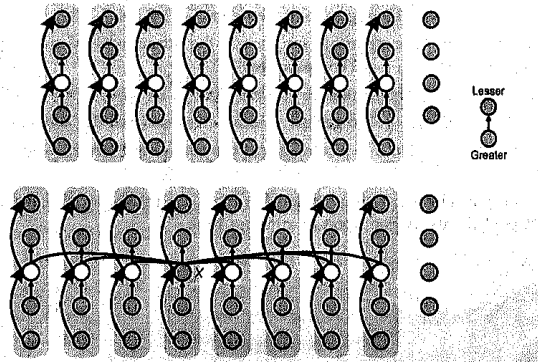
9.6 Choosing the Pivot



1. Divide the n elements into groups of 5. Find the median of each 5-element group by rote.



2. Recursively select the median x of the $\lfloor n/5 \rfloor$ group medians to be the pivot.



At least half the group medians are $\leq x$, which is at least $\lfloor \lfloor n/5 \rfloor /2 \rfloor = \lfloor n/10 \rfloor$ group medians.

- Therefore, at least $3\lfloor n/10 \rfloor$ elements are $\leq x$
- Similarly, at least $3\lfloor n/10 \rfloor$ elements are $\geq x$.

We can now develop a recurrence for the worst-case running time $T(n)$ of the algorithm SELECT. Steps 1, 2, and 4 take $O(n)$ time. (Step 2 consists of $O(n)$ calls of insertion sort on sets of size $O(1)$.) Step 3 takes time $T(\lceil n/5 \rceil)$, and step 5 takes time at most $T(7n/10+6)$, assuming that T is monotonically increasing. Note that $7n/10+6 < n$ for $n>20$ and that any input of 80 or fewer elements requires $O(1)$ time. We can therefore obtain the recurrence

$$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}+6\right) + O(n)$$

We show that the running time is linear by substitution. Assume that $T(n) \leq cn$ for some constant c and all $n>0$. Substituting this inductive hypothesis into the right-hand side of the recurrence yields

$$T(n) \leq c\frac{n}{5} + c\left(\frac{7n}{10}+6\right) + O(n)$$

$$\leq \frac{cn}{5} + c + \frac{7cn}{10} + 6c + O(n)$$

$$= \frac{9cn}{10} + 7c + O(n) = cn + \left(-\frac{cn}{10} + 7c + an\right)$$ which is at most cn if $\left(-\frac{cn}{10} + 7c + an\right) \leq 0$

The worst-case running time of SELECT is therefore linear *i.e.*, $O(n)$.

Exercise

1. Find the MEDIAN in the following set S of elements 1 8 3 3 5 4 7 7 6 8.
2. Design a divide-and-conquer algorithm for finding the minimum and the maximum element of n numbers using no more than $3n/2$ comparisons.
3. Show how to compute weighted median of n elements in $O(n \lg n)$ worst case time using sorting.
4. Write an algorithm MEDIAN (S) to get the median element from the sequence S of n elements.
5. Can you obtain less than $\lceil 3n/2 \rceil - 2$ comparison for obtaining both maximum and minimum of n number.
6. Devise an algorithm which can simultaneously obtain the minimum and maximum element from the given set of elements. Argue upon its running time.

CHAPTER 10

Dictionaries and Hash Tables

10.1 Dictionaries

A computer dictionary is similar to an ordinary dictionary because both are used to look things up. The main idea is that users can assign keys to elements and then use those keys later to look up or remove elements. So an abstract data type that support the operations insert, delete and search is called **dictionary**.

A dictionary stores key-element pairs (k, e) which we call items where k is the key and e is the element. We distinguish *two* types of dictionaries :

1. Unordered dictionary
2. Ordered dictionary

In either case, we use a key as an identifier that is assigned by an application or user to an associated element.

As an ADT, a dictionary D supports the following methods :

1. **Find element** (k) : If D contains an item with key equal to k then return the element of such item else return No-such-key.
2. **Insert item** (k, e) : Insert an item with element e and key k into dictionary D.
3. **Remove element** (k) : Remove from D an item with key equal to k and return its element. If D has no such item, then return the No-such-key.

10.2 Log Files

A simple way of realizing a dictionary D uses an unsorted sequence S which is implemented using a vector or list to store the key-element pairs. Such an implementation is called **log files**. The space required for a log file is $\theta(n)$ because both vector and linked list data structures can maintain their memory usage to be proportional to their size. The primary applications of a log file are situations where we wish to store small amounts of data or data that is unlikely to change much over time. We also refer to the log file implementation of D as an unordered sequence implementation.

In ordered dictionary, we wish to perform the usual dictionary operations and also maintain an order relation for the keys in our dictionary. We can use a comparator to provide the order relation among keys. Such an ordering helps us to efficiently implement the dictionary ADT. In addition, an ordered dictionary also supports the following methods :

Closest Key Before (k) : Return the key of the item with largest key less than or equal to k.

Closest Elem Before (k) : Return the element for the item with largest key less than or equal to k.

Closest Key After (k) : Return the key of the item with smallest key greater than or equal to k.

Closest Elem After (k) : Return the element for the item with smallest key greater than or equal to k.

Each of these methods returns the special No-such-key object if no item in the dictionary satisfies the query.

Thus, a dictionary is a data structure that supports the operations of Search, Insert, Delete.

There are *two* important variants of dictionary :

- **A static dictionary** is a restricted version that supports only the SEARCH operation. The goal here is to come up with a compact representation of the set while supporting very fast look-ups.
- When the elements of the set come from a totally ordered set, a dictionary on a total order supports MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR operations in addition to the three standard dictionary operations.

10.3 Hash Tables

Hash tables support one of the most efficient types of searching : ***hashing***. Fundamentally, a hash table consists of an array in which data is accessed via a special index called a *key*. The primary idea behind a hash table is to establish a mapping between the set of all possible keys and positions in the array using a *hash function*. A hash function accepts a key and returns its *hash coding*, or *hash value*. Keys vary in type, but hash codings are always integers. Since both computing a hash value and indexing into an array can be performed in constant time, the beauty of hashing is that we can use it to perform constant time searches. When a hash function can guarantee that no two keys will generate the same hash coding, the resulting hash table is said to be *directly addressed*.

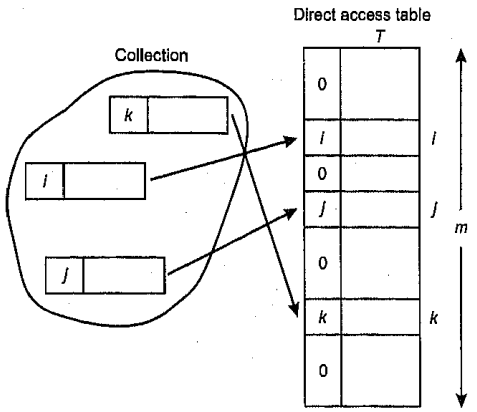


Figure 10.1

This is ideal, but direct addressing is rarely possible in practice.

Typically, the number of entries in a hash table is small relative to the universe of possible keys. Consequently, most hash functions map some keys to the same position in the table. When two keys map to the same position, they *collide*. A good hash function minimizes collisions, but we must still be prepared to deal with them.

10.3.1 Applications of Hash Tables

Some applications of hash tables are :

1. **Database systems.** Specifically, those that require efficient random access. Generally, database systems try to optimize between two types of access methods : sequential and random. Hash tables are an important part of efficient random access because they provide a way to locate data in a constant amount of time.

2. **Symbol tables.** The tables used by compilers to maintain information about symbols from a program. Compilers access information about symbols frequently. Therefore, it is important that symbol tables be implemented very efficiently.

3. **Tagged buffers.** A mechanism for storing and retrieving data in a machine-independent manner. Each data member resides at a fixed offset in the buffer. A hash table is stored in the buffer so that the location of each tagged member can be ascertained quickly. One use of a tagged buffer is sending structured data across a network to a machine whose byte ordering and structure alignment may not be the same as the original host's. The buffer handles these concerns as the data is stored and extracted member-by-member.

4. **Data dictionaries.** Data structures that support adding, deleting, and searching for data. Although the operations of a hash table and a data dictionary are similar, other data structures may be used to implement data dictionaries. Using a hash table is particularly efficient.

5. Associative arrays. Most commonly used in languages that do not support structured types. Associative arrays consist of data arranged so that the nth element of one array corresponds to the nth element of another. Associative arrays are useful for indexing a logical grouping of data by several key fields. A hash table helps to key into each array efficiently.

10.4 Hashing

Hashing is a widely used class of data structures that support the operations of insert, delete and search on a dynamic set of keys S. The keys are drawn from a universe $U(|U| >> |S|)$, which for our purposes will be a set of positive integers. These keys are mapped into a hash table using a hash function; the size of the hash table is usually within a constant factor of the number of elements in the current set S.

There are two main methods used to implement hashing : hashing with chaining and hashing with open addressing.

10.4.1 Hashing with Chaining

In hashing with chaining, the elements in S are stored in a hash table $T[0..m-1]$ of size m, where m is somewhat larger than n, the size of S. The hash table is said to have m slots. Associated with the hashing scheme is a hash function h which is a mapping from U to $\{0,..,m-1\}$. Each key $k \in S$ is stored in location $T[h(k)]$, and we say that key k is hashed into slot $h(k)$. If more than one key in S hashes into the same slot, then we have a **collision.**

In such a case, all keys that hash into the same slot are placed in a linked list associated with that slot; this linked list is called the **chain at that slot.** The **load factor** of a hash table is defined to be $\alpha = n/m$; it represents the average number of keys per slot. We typically operate in the range $m = \theta(n)$, so α is usually a constant (usually $\alpha < 1$). If a large number of insertions or deletions destroy this property, a more suitable value of m is chosen and the keys are re-hashed into a new table with a new hash function.

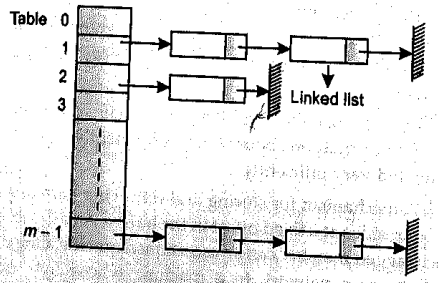


Figure 10.2

In simple uniform hashing we assume that we have a 'good' hash function h such that any element in U is equally likely to hash into any of the m slots in T, independent of the other keys in the table. We also assume that $h(k)$ can be computed in constant time.

In hashing with chaining, inserts and deletes can be performed in constant time with a suitable linked list representation for the chains. In the worst case, a search operation can take as long as n steps if all keys hash into the same slot. However, if we assume simple uniform hashing then the expected time for a search operation is easily shown to be $\theta(\alpha)$, which is a constant under the normal operation condition of $m = \theta(n)$. The main drawback with this method is the requirement that the scheme implements simple uniform hashing.

Analysis of hashing with chaining

Given a hash table T with m slots that stores n elements, we define the *load factor* α for T as n/m, that is, the average number of elements stored in a chain. The worst-case time for searching is thus $\theta(n)$ plus the time to compute the hash function—no better than if we used one linked list for all the elements. Clearly, hash tables are not used for their worst-case performance.

The average performance of hashing depends on how well the hash function h distributes the set of keys to be stored among the m slots, on the average.

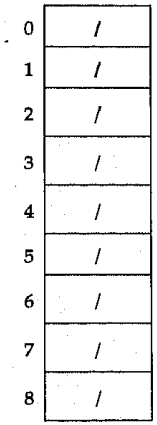
Theorem

In a hash table in which collisions are resolved by chaining, an unsuccessful search takes time $\theta(1+\alpha)$, on the average, under the assumption of simple uniform hashing.

Proof : Under the assumption of simple uniform hashing, any key k is equally likely to hash to any of the m slots. The average time to search unsuccessfully for a key k is thus the average time to search to the end of one of the m lists. The average length of such a list is the load factor $\alpha = n/m$. Thus, the expected number of elements examined in an unsuccessful search is α, and the total time required (including the time for computing $h(k)$) is $\theta(1+\alpha)$.

Example. *Let us consider the insertion of elements 5, 28, 19, 15, 20, 33, 12, 17, 10 into a chained-hash table. Let us suppose the hash table has 9 slots and the hash function be $h(k) = k \bmod 9$.*

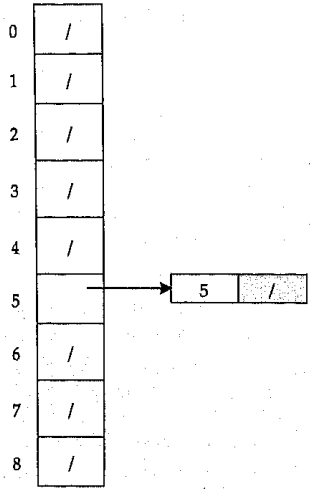
Solution. The initial state of the chained-hash table



Insert 5 :

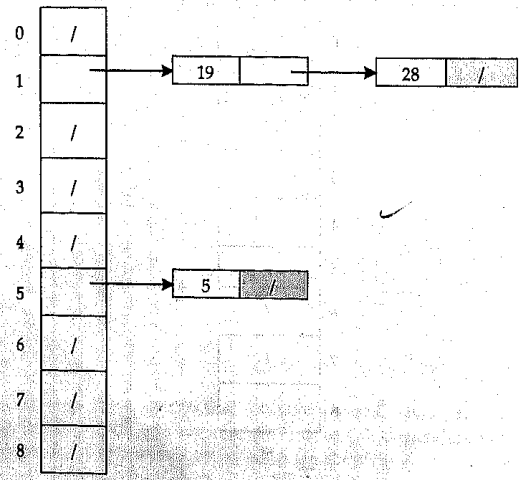
$h(5) = 5 \bmod 9 = 5$

Create a linked-list for T[5] and store value 5 in it.



Similarly insert 28. $h(28) = 28 \bmod 9 = 1$. Create a link-list for $T[1]$ and store value 28 in it.

Now insert 19 $h(19) = 19 \bmod 9 = 1$. Insert value 19 in the slot $T[1]$ in the beginning of the link-list.



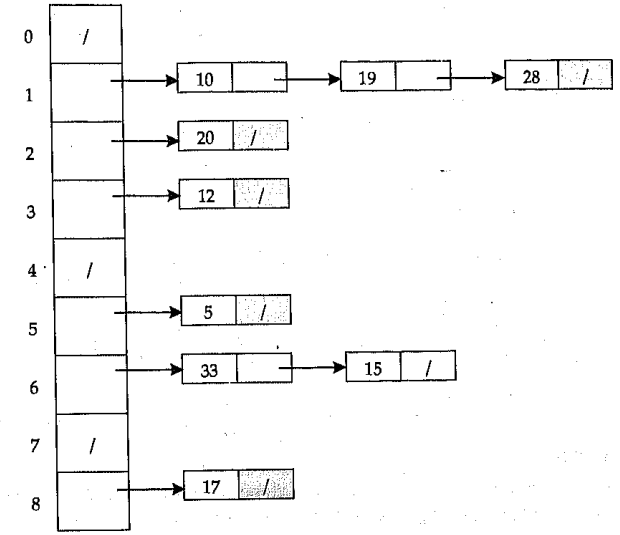
Now insert 15. $h(15) = 15 \bmod 9 = 6$. Create a link list for $T[6]$ and store value 15 in it. Similarly insert 20, $h(20) = 20 \bmod 9 = 2$ in $T[2]$. Insert 33, $h(33) = 33 \bmod 9 = 6$, in the beginning of the linked list for $T[6]$. Then,

Insert 12 *i.e.*, $h(12) = 12 \bmod 9 = 3$ in $T[3]$

Insert 17 *i.e.*, $h(17) = 17 \bmod 9 = 8$ in $T[8]$

Insert 10 *i.e.*, $h(10) = 10 \bmod 9 = 1$ in $T[1]$.

Thus the chained-hash-table after inserting key 10 is



10.5 Hash Functions

Hash Function is a function which, when applied to the key, produces an integer which can be used as an address in a hash table. The intent is that elements will be relatively randomly and uniformly distributed. **Perfect Hash Function** is a function which, when applied to all the members of the set of items to be stored in a hash table, produces a unique set of integers within some suitable range. Such function produces no collisions. **Good Hash Function** minimizes collisions by spreading the elements uniformly throughout the array.

There is no magic formula for the creation of the hash function. It can be any mathematical transformation that produces a relatively random and unique distribution of values within the address space of the storage.

Characteristics of a Good Hash Function

There are *four* main characteristics of a good hash function :

- The hash value is fully determined by the data being hashed.

- The hash function uses all the input data.
- The hash function "uniformly" distributes the data across the entire set of possible hash values.
- The hash function generates very different hash values for similar strings.

Let's examine why each of these is important:

Rule 1 If something else besides the input data is used to determine the hash, then the hash value is not as dependent upon the input data, thus allowing for a worse distribution of the hash values.

Rule 2 If the hash function doesn't use all the input data, then slight variations to the input data would cause an inappropriate number of similar hash values resulting in too many collisions.

Rule 3 If the hash function does not uniformly distribute the data across the entire set of possible hash values, a large number of collisions will result, cutting down on the efficiency of the hash table.

Rule 4 In real world applications, many data sets contain very similar data elements. We would like these data elements to still be distributable over a hash table.

However, finding a perfect hash function that works for a given data set can be extremely time consuming and very often it is just impossible. Therefore we must live with collisions and learn how to handle them.

Some common Hash functions are:

1. Division method

In the division method for creating hash functions, we map a key k into one of m slots by taking the remainder of k divided by m That is, the hash function is

$$h(k) = k \bmod m \quad \text{or} \quad h(k) = k \bmod m+1$$

For example, if the hash table has size $m=12$ and the key is $k=100$, then $h(k)=4$. Since it requires only a single division operation, hashing by division is quite fast.

When using the division method, we usually avoid certain values of m For example, m should not be a power of 2, since if $m=2^p$, then $h(k)$ is just the p lowest-order bits of k. Powers of 10 should be avoided if the application deals with decimal number of keys.

2. Multiplication method

The **multiplication method** for creating hash functions operates in two steps. First, we multiply the key k by a constant A in the range $0 < A < 1$ and extract the fractional part of kA. Then, we multiply this value by m and take the floor of the result. In short, the hash function is

$$h(k) = \lfloor m(kA \bmod 1) \rfloor,$$

where "$kA \bmod 1$" means the fractional part of kA, that is, $kA - \lfloor kA \rfloor$.

This method is not affected too much by the choice of m (which is the size of the hash table), but the value of A will impact the effectiveness of this method. 'Knuth' has suggested in his study that the following value of A is likely to work reasonably

$$A \approx \frac{(\sqrt{5}-1)}{2} = 0.6180339887 \ldots \text{ is a good choice.}$$

10.6 Universal Hashing

The main idea behind universal hashing is to select the hash function at random at run time from a carefully designed class of functions. This approach guarantees good average-case performance, no matter what keys are provided as input. Poor performance occurs only if the compiler chooses a random hash function that causes the set of identifiers to hash poorly, but the probability of this occurring is small and is the same for any set of identifiers of the same size.

Let H be a finite collection of hash functions that map a given universe U of keys into the range $\{0,1,\ldots m-1\}$. Such a collection is said to be **universal** if for each pair of distinct keys $x, y, \in U$, the number of hash functions $h \in H$ for which $h(x) = h(y)$ is precisely $\frac{|H|}{m}$. In other words, with a hash function randomly chosen from H, the chance of a collision between x and y when $x \neq y$ is exactly $1/m$, which is exactly the chance of a collision if $h(x)$ and $h(y)$ are randomly chosen from the set $\{0,1,\ldots,m-1\}$.

10.7 Hashing With Open Addressing

In *open addressing*, all elements are stored in the hash table itself. That is, each table entry contains either an element of the dynamic set or NIL. When searching for an element, we systematically examine table slots until the desired element is found or it is clear that the element is not in the table. Thus, in open addressing, the load factor α can never exceed 1.

The advantage of open addressing is that it avoids pointers altogether. Instead of following pointers, we *compute* the sequence of slots to be examined. The extra memory freed by not storing pointers provides the hash table with a larger number of slots for the same amount of memory, potentially yielding fewer collisions and faster retrieval.

The process of examining the locations in the hash table is called 'Probing'

To perform insertion using open addressing, we successively examine, or *probe*, the hash table until we find an empty slot in which to put the key.

```
HASH-INSERT (T, k)
1.  i ← 0
2.  repeat j ← h(k,i)
3.      if T[j] = NIL
4.          then T[j] ← k
5.          return j
6.      else i ← i + 1
7.  until i = m
8.  error "hash table overflow"
```

The algorithm for searching for key k probes the same sequence of slots that the insertion algorithm examined when key k was inserted. Therefore, the search can terminate (unsuccessfully) when it finds an empty slot, since k would have been inserted there and not later in its probe sequence. (Note that this argument assumes that keys are not deleted from the hash table.) The procedure HASH-SEARCH takes as input a hash table T and a key k, returning j if slot j is found to contain key k, or NIL if key k is not present in table T.

```
HASH-SEARCH(T, k)
1.  i ← 0
2.  repeat j ← h(k,i)
3.      if T[j] = j
4.          then return j
5.              i ← i + 1
6.  until T[j] = NIL or i = m
7.  return NIL
```

Deletion from an open-address hash table is difficult. When we delete a key from slot i, we cannot simply mark that slot as empty by storing NIL in it. Doing so might make it impossible to retrieve any key k during whose insertion we had probed slot i and found it occupied. One solution is to mark the slot by storing in it the special value DELETED instead of NIL. We would then modify the procedure HASH-SEARCH so that it keeps on looking when it sees the value DELETED, while HASH-INSERT would treat such a slot as if it were empty so that a new key can be inserted. When we do this, though, the search times are no longer dependent on the load factor α, and for this reason chaining is more commonly selected as a collision resolution technique when keys must be deleted.

Three techniques are commonly used to compute the probe sequences required for open addressing: **linear probing, quadratic probing, and double hashing.**

10.7.1 Linear Probing

Given an ordinary hash function $h' : U \{0,1,..,m-1\}$, the method of **linear probing** uses the hash function

$$h(k, i) = (h'(k) + i) \bmod m$$

where 'm' is the size of the hash table and $h'(k) = k \bmod m$ (basic hash function). For $i = 0,1,...m-1$. Given key k, the first slot probed is $T[h'(k)]$. We next probe slot $T[h'(k)+1]$, and so on up to slot $T[m-1]$. Then we wrap around to slots $T[0]$, $T[1]$,..., until we finally probe slot $T[h'(k)-1]$. Since the initial probe position determines the entire probe sequence, only m distinct probe sequences are used with linear probing.

Example. *Consider inserting the keys 26, 37, 59, 76, 65, 86 into a hash-table of size $m=11$ using linear probing, consider the primary hash function is $h'(k) = k \bmod m$.*

Solution. Initial state of the hash table

	0	1	2	3	4	5	6	7	8	9	10
T	/	/	/	/	/	/	/	/	/	/	/

1. Insert 26. We know $h(k,i) = [h'(k) + i] \bmod m$

Now $h(26,0) = [26 \bmod 11 + 0] \bmod 11$

$= (4+0) \bmod 11 = 4 \bmod 11 = 4$

Since $T[4]$ is free, insert key 26 at this place.

2. Insert 37. Now $h(37,0) = [37 \bmod 11 + 0] \bmod 11$

$= [4+0] \bmod 11 = 4$

Since $T[4]$ is not free, the next probe sequence is computed as

$h[37,1] = [37 \bmod 11 + 1] \bmod 11$

$= [4+1] \bmod 11 = 5 \bmod 11 = 5$

$T[5]$ is free, insert key 37 at this place.

3. Insert 59. Now $h(59,0) = [59 \bmod 11 + 0] \bmod 11$

$= [4+0] \bmod 11 = 4$

$T[4]$ is not free, so the next probe sequence is computed as

$h[59,1] = [59 \bmod 11 + 1] \bmod 11$

$= 5$

$T[5]$ is also not free, so the next probe sequence is computed.

$h(59,2) = (59 \bmod 11 + 2) \bmod 11$

$= 6 \bmod 11 = 6$

$T[6]$ is free. Insert key 59 at this place.

4. Insert 76. $h(76,0) = (76 \bmod 11 + 0) \bmod 11$

$= (10+0) \bmod 11 = 10$

$T[10]$ is free, insert key at this place.

5. Insert 65. $h(65,0) = (65 \bmod 11 + 0) \bmod 11$

$= (10+0) \bmod 11 = 10$

$T[10]$ is occupied, the next probe sequence is computed as

$h(65,1) = (65 \bmod 11 + 1) \bmod 11$

$= (10+1) \bmod 11 = 11 \bmod 11 = 0$

$T[0]$ is free, insert key 65 at this place.

6. Insert 86. $h(86,0) = (86 \bmod 11 + 0) \bmod 11$

$= 9 \bmod 11 = 9$

$T[9]$ is free, insert key 86 at this place.

Thus,

	0	1	2	3	4	5	6	7	8	9	10
T	65	/	/	/	26	37	59	/	/	86	76

One main disadvantage of linear probing is that records tend to 'Cluster' i.e. appear next to one another, when the load factor is greater than 50%. Such clustering substantially increases the average search time for a record. Two techniques to minimize clustering are as follows :

10.7.2 Quadratic Probing

Suppose a record R with key k has the hash address $H(k)=h$ then instead of searching the locations with addresses h, $h+1$, $h+2$,... We linearly search the locations with addresses

$$h,\ h+1,\ h+4,\ h+9,\ldots,h+i^2,\ldots$$

Quadratic probing uses a hash function of the form

$$h(k,i)=(h'(k)+c_1 i+c_2 i^2)\bmod m$$

where (as in linear probing) h' is an auxiliary hash function, c_1 and $c_2 \neq 0$ are auxiliary constants, and $i=0,1,\ldots,m-1$ The initial position probed is $T[h'(k)]$; later positions probed are offset by amounts that depend in a quadratic manner on the probe number i. This method works much better than linear probing, but to make full use of the hash table, the values of c_1, c_2 and m are constrained.

Example. *Consider inserting the keys 76, 26, 37, 59, 21, 65 into a hash table of size $m=11$ using quadratic probing with $c_1=1$ and $c_2=3$. Further consider that the primary hash function is $h'(k)=k \bmod m$*

Solution. For quadratic probing, we have

$$h(k,i)=[k \bmod m+c_1 i+c_2 i^2]\bmod m$$

0	1	2	3	4	5	6	7	8	9	10
/	/	/	/	/	/	/	/	/	/	/

This is the initial state of the hash table.

Here $c_1=1$ $c_2=3$

$$h(k,i)=[k \bmod m+i+3i^2]\bmod m$$

1. Insert 76.

$$h(76,0)=(76 \bmod 11+0+3\times 0)\bmod 11$$
$$=(10+0+0)\bmod 11=10$$

$T[10]$ is free, insert the key 76 at this place.

2. Insert 26.

$$h(26,0)=(26 \bmod 11+0+3\times 0)\bmod 11$$
$$=(4+0+0)\bmod 11=4.$$

$T[4]$ is free, insert the key 26 at this place.

3. Insert 37.

$$h(37,0)=(37 \bmod 11+0+3\times 0)\bmod 11$$
$$=(4+0+0)\bmod 11=4$$

$T[4]$ is not free, so next probe sequence is computed as

$$h(37,1)=(37 \bmod 11+1+3\times 1^2)\bmod 11$$
$$=(4+1+3)\bmod 11$$
$$=8 \bmod 11=8$$

$T[8]$ is free. Insert the key 37 at this place.

4. Insert 59.

$$h(59,0)=(59 \bmod 11+0+3\times 0)\bmod 11$$
$$=(4+0+0)\bmod 11=4 \bmod 11=4$$

$T[4]$ is not free, so next probe sequence is computed as

$$h(59,1)=(59 \bmod 11+1+3\times 1^2)\bmod 11$$
$$=(4+1+3)\bmod 11=8 \bmod 11=8$$

$T[8]$ is also not free, so the next probe sequence is computed as

$$h(59,2)=(59 \bmod 11+2+3\times 2^2)\bmod 11$$
$$=(4+2+12)\bmod 11=18 \bmod 11=7$$

$T[7]$ is free, insert key 59 at this place.

5. Insert 21.

$$h(21,0)=(21 \bmod 11+0+3\times 0)\bmod 11$$
$$=(10+0+0)\bmod 11=10 \bmod 11=10$$

$T[10]$ is not free, the next probe sequence is computed as

$$h(21,1)=(21 \bmod 11+1+3\times 1^2)\bmod 11$$
$$=(10+1+3)\bmod 11=14 \bmod 11=3$$

$T[3]$ is free, so insert key 21 at this place.

6. Insert 65.

$$h(65,0)=(65 \bmod 11+0+3\times 0)\bmod 11$$
$$=(10+0+0)\bmod 11=10 \bmod 11=10$$

Since $T[10]$ is not free, the next probe sequence is computed as

$$h(65,1)=(65 \bmod 11+1+3\times 1^2)\bmod 11$$
$$=(10+1+3)\bmod 11=14 \bmod 11=3$$

$T[3]$ is not free, so the next probe sequence is computed as

$$h(65,2)=(65 \bmod 11+2+3\times 2^2)\bmod 11$$
$$=(10+2+12)\bmod 11=24 \bmod 11=2$$

$T[2]$ is free, insert the key 65 at this place.

Thus, after inserting all keys, the hash table is

0	1	2	3	4	5	6	7	8	9	10
/	/	65	21	26	/	/	59	37	/	76

10.7.3 Double hashing

Double hashing is one of the best methods available for open addressing because the permutations produced have many of the characteristics of randomly chosen permutations.

Double hashing uses a hash function of the form

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m$$

where h_1 and h_2 are auxiliary hash functions and m is the size of the hash table.

$h_1(k) = k \bmod m$ or $h_2(k) = k \bmod m'$. Here m' is slightly less than m (say $m-1$ or $m-2$).

The initial position probed is $T[h_1(k)]$; successive probe positions are offset from previous positions by the amount $h_2(k)$ modulo m. Thus, unlike the case of linear or quadratic probing, the probe sequence here depends in two ways upon the key k, since the initial probe position, the offset, or both, may vary.

Double hashing represents an improvement over linear or quadratic probing in that $\theta(n^2)$ probe sequences are used, rather than $\theta(m)$, since each possible $(h_1(k), h_2(k))$ pair yields a distinct probe sequence, and as we vary the key, the initial probe position $h_1(k)$ and the offset $h_2(k)$ may vary independently. As a result, the performance of double hashing appears to be very close to the performance of the "ideal" scheme of uniform hashing.

Example. *Consider inserting the keys 76, 26, 37, 59, 21, 65 into a hash table of size $m=11$ using double hashing. Consider that the auxiliary hash functions are $h_1(k) = k \bmod 11$ and $h_2(k) = k \bmod 9$.*

Solution. Initial state of hash table is

	0	1	2	3	4	5	6	7	8	9	10
T	/	/	/	/	/	/	/	/	/	/	/

1. Insert 76.

$h_1(76) = 76 \bmod 11 = 10$

$h_2(76) = 76 \bmod 9 = 4$

$h(76,0) = (10 + 0 \times 4) \bmod 11$

$= 10 \bmod 11 = 10$

$T[10]$ is free, so insert key 76 at this place.

2. Insert 26.

$h_1(26) = 26 \bmod 11 = 4$

$h_2(26) = 26 \bmod 9 = 8$

$h(26,0) = (4 + 0 \times 8) \bmod 11$

$= 4 \bmod 11 = 4$

$T[4]$ is free, so insert key 26 at this place.

3. Insert 37.

$h_1(37) = 37 \bmod 11 = 4$

$h_2(37) = 37 \bmod 9 = 1$

$h(37,0) = (4 + 0 \times 1) \bmod 11 = 4 \bmod 11 = 4$

$T[4]$ is not free, the next probe sequence is computed as

$h(37,1) = (4 + 1 \times 1) \bmod 11 = 5 \bmod 11 = 5$

$T[5]$ is free, so insert key 37 at this place.

4. Insert 59.

$h_1(59) = 59 \bmod 11 = 4$

$h_2(59) = 59 \bmod 9 = 5$

$h(59,0) = (4 + 0 \times 5) \bmod 11 = 4 \bmod 11 = 4$

Since, $T[4]$ is not free, the next probe sequence is computed as

$h(59,1) = (4 + 1 \times 5) \bmod 11 = 9 \bmod 11 = 9$

$T[9]$ is free, so insert key 59 at this place.

5. Insert 21.

$h_1(21) = 21 \bmod 11 = 10$

$h_2(21) = 21 \bmod 9 = 3$

$h(21,0) = (10 + 0 \times 3) \bmod 11 = 10 \bmod 11 = 10$

$T[10]$ is not free, the next probe sequence is computed as

$h(21,1) = (10 + 1 \times 3) \bmod 11 = 13 \bmod 11 = 2$

$T[2]$ is free, so insert key 21 at this place.

6. Insert 65.

$h_1(65) = 65 \bmod 11 = 10$

$h_2(65) = 65 \bmod 9 = 2$

$h(65,0) = (10 + 0 \times 2) \bmod 11 = 10 \bmod 11 = 10$

$T[10]$ is not free, the next probe sequence is computed as

$h(65,1) = (10 + 1 \times 2) \bmod 11 = 12 \bmod 11 = 1$

$T[1]$ is free, so insert key 65 at this place.

Thus after insertion of all keys the final hash table is

0	1	2	3	4	5	6	7	8	9	10
/	65	21	/	26	37	/	/	/	59	76

10.8 Rehashing

If at any stage the hash table becomes nearly full, the running time for the operations will start taking too much time, insert operation may fail than in such situation, the best possible solution is as follows :

1. Create a new hash table double in size.
2. Scan the original hash table, compute new hash value and insert into the new hash table.
3. Free the memory occupied by the original hash table.

Example. *Suppose we wish to search a linked list of length n, where each element contains a key k along with a hash value h(k). Each key is a long character string. How might we take advantage of the hash values when searching the list for an element with a given key?*

Solution. Each key is a long character thus to compare keys, at every node we need to perform a string comparison operation which is very time consuming. Instead we generate a hash value for the key (*i.e.*, generate a numeric value for each string) we are searching for and comparing hash values $h(k)$ along the length of the list, which turns out to be numeric values and the comparison is faster.

Example. *Consider inserting the keys 10, 22, 31, 4, 15, 28, 17, 88, and 59 into a hash table of length* $m=11$ *using open addressing with the primary hash function* $h'(k)=k \bmod m$. *Illustrate the result of inserting these keys using linear probing, using quadratic probing with* $c_1=1$ *and* $c_2=3$, *and using double hashing with* $h_2(k)=1+(k \bmod (m-1))$.

Solution. Using Linear probing the final state of the hash table would be :

0	22
1	88
2	/
3	/
4	4
5	15
6	28
7	17
8	59
9	31
10	10

Using Quadratic probing, with $c_1=1$, $c_2=3$, the final state of the hash table would be $h(k,i)=(h'(k)+c_1*i+c_2*i^2) \bmod m$ where $m=11$ and $h'(k)=k \bmod m$

0	22
1	88
2	/
3	17
4	4
5	/
6	28
7	59
8	15
9	31
10	10

Using double hashing the final state of the hash table would be:

0	22
1	/
2	59
3	17
4	4
5	15
6	28
7	88
8	/
9	31
10	10

Example. *Consider an open-address hash table with uniform hashing. Give upper bounds on the expected number of probes in an unsuccessful search and on the expected number of probes in a successful search when the load factor is ¾ and when it is 7/8.*

Solution. Given an open address hash table with load factor $\alpha=n/m<1$, the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$, assuming uniform hashing.

$\alpha=\frac{3}{4}$, then the upper bound on the number of probes $=\frac{1}{\left(1-\frac{3}{4}\right)}=4$ **probes.**

$\alpha=\frac{7}{8}$, then the upper bound on the number of probes $=\frac{1}{\left(1-\frac{7}{8}\right)}=8$ **probes.**

Given an open address hash table with load factor $\alpha = n/m < 1$, the expected number of probes in a successful search is at most $(1/\alpha)\ln(1/(1-\alpha))$, assuming uniform hashing and assuming that each key in the table is equally likely to be searched for.

$$\alpha = \frac{3}{4}\left(\frac{1}{3/4}\right)\ln\frac{1}{1-\frac{3}{4}} = 1.85 \text{ probes}$$

$$\alpha = \frac{7}{8}\left(\frac{1}{.875}\right)\ln\frac{1}{1-.875} = 2.38 \text{ probes}$$

Example. *Professor Marley hypothesizes that substantial performance gains can be obtained if we modify the chaining scheme so that each list is kept in sorted order. How does the professor's modification affect the running time for successful searches, unsuccessful searches, insertions, and deletions ?*

Solution. Consider keeping the chaining lists in sorted order. Searching will still take time proportional to the length of the list and therefore the running times are the same. The only difference is the insertions which now also take time proportional to the length of the list.

Exercise

1. Write an algorithm to search a record in chained method.
2. For data 46, 78, 6, 2, 43, 27, 62, 4 give the result of inserting the values into a hash table when using open addressing with quadratic collision resolution ($c_1 = c_2 = 1$) and
   Hash $(k) = k\%17$ $(M = 17)$.
3. What is Hashing ? Explain why we require the table size to be a prime number in double hashing.
4. Let the table S and 12 memory locations and suppose records A, B, C, D, E, X, Y and Z are to be inserted with following hash address.

Record	A	B	C	D	E	X	Y	Z
	4	8	2	12	4	12	5	1

   Find successful and unsuccessful linear probes.
5. Consider inserting the keys 20, 12, 31, 4, 25, 28, 27, 38, 69 into a hash table of length $m = 11$ using open addressing with the primary hash function $h_1(k) = k \bmod m$.
   Illustrate the result of inserting keys using double hashing with $h_2(k) = 1 + (k \bmod (m-1))$

CHAPTER 11

Elementary Data Structures

11.1 Introduction

Abstraction refers to the treatment of problems whilst ignoring associated concrete details such as computer hardware or implementation details. One simple example of this abstraction is provided by the pre-defined type integer to be found in many programming languages. It is possible to discuss the properties of integers and how to use them without necessarily having to know how they are represented at machine level. Another example of abstraction is provided by the high-level program, which represents, as it were, the solution to a problem on an abstract machine. The high-level programming language provides a convenient notation for problem analysis and solution. The details of the physical machine on which the program may be executed have been abstracted away.

It is this process of abstraction that makes programming feasible.

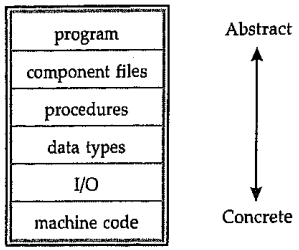


Figure 11.1

High-level programming languages make a selection of predefined atomic data types available. An atomic data type consists of a single component. These *primitive* types commonly include *integer, real or floating point, character,* and *boolean or logical.* In addition to these, programming languages offer a selection of predefined structured data types. A structured data type consists of a number of components associated together to form one logical whole. Simple examples are *arrays* and *array lists.* An array consists of a number of elements, which will be taken to be of the same basic type and in which the elements can be distinguished by means of an index. It should be appreciated that there is a considerable degree of abstraction here, which disguises the mapping between the indices and machine addresses. This abstraction is all the greater in the case of two-dimensional arrays since the basic machine memory is *linear.* An array list consists of a number of objects, which need not necessarily be of the same basic type.

Programming languages also provide a means of building new and more complex structures. The obvious example in Java is the *class,* which is a means of specifying objects, which comprise a set of variables, together with methods to do operations on those variables. An object displays *representational abstraction,* in as much as the implementation of the variables may be hidden from the outside, and *procedural abstraction,* in that the implementation of the methods is hidden. We will say lots more about this. In the mean time it is important to recognize that each data type has an associated set of operations by means of which the data values may be manipulated.

Data structures store a given set of data according to certain rules. These rules help in organizing data. Arrays and structures are built in data structures in most of the programming languages. Data can be stored in many other ways using other type of data structures. Thus, we can define data structure as a collection of data elements whose organization is characterized by accessing functions that are used to store and retrieve individual data elements. Data structures are represented at the logical level using a tool called **Abstract data type (ADT)** and actually implemented using algorithms.

11.2 Abstract Data Type

An ***abstract data type*** *(ADT)* consists of a data type together with a set of operations, which define how the type may be manipulated. This specification is stated in an implementation-independent manner.

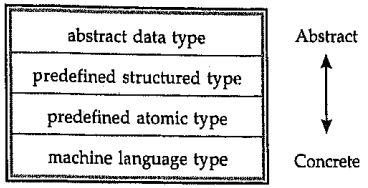


Figure 11.2

Abstract data types exist conceptually and concentrate on the (essential) mathematical properties of the data type ignoring implementation constraints and details. *Virtual* data types exist on a virtual processor such as a programming language. *Physical* data types exist on a physical processor such as the machine level of a computer. Abstract data types are implemented with virtual data types. Virtual data types are translated into physical data types. The advantages offered by abstract data types include :

1. modularity
2. precise specifications
3. information hiding
4. simplicity
5. integrity
6. implementation independence

While defining an ADT, we are not concerned with time and space efficiency or any other implementation details of the data structure. ADT is just a useful guideline to use and implement the data type.

An ADT has *two* parts :

1. Value definition
2. Operation definition

Value definition is again divided into *two* parts :

1. Definition clause
2. Condition clause

As the name suggests the **definition clause** states the contents of the data type and **condition clause** defines any condition that applies to the data type. **Definition clause** is mandatory while **condition clause** is optional.

In operational definition, there are *three* parts :

- Function
- Precondition
- Post condition

The function clause defines the role of the operation. If we consider the addition operation in integers, the function clause will state that two integers can be added using this function. In general, precondition specifies any restrictions that must be satisfied before the operation can be applied. This clause is optional. If we consider the division operation on integers then the

precondition will state that the divisor should not be zero. Precondition specifies any condition that may apply as a pre-requisite for the operation definition. Post condition specifies what the operation does *i.e.* it specifies the state after the operation is performed

The following figure 11.3 lists some typical abstract data types :

Vector	Direct access to each element and Fixed size.
List	*Unbounded.* Fast access to first element and thereafter sequential access.
Stack	*LIFO structure.* Insertions and deletions from one end only.
Queue	*FIFO structure.* Insertions and deletions from the ends.
Tree	Fast insertions and access. Hierarchical structures.
Hash table	Fast access.
Set	Fast unions and intersections. Elements not maintained in order.
Bag	As with sets but allows multiple occurrences of elements.
Dictionary	Values associated with key.
Graph	Collection of nodes and arcs. Unordered
File	External storage for persistent data. Slow access.

Figure 11.3

11.3 Stack

A stack is a temporary abstract data type and data structure based on the principle of **Last In First Out** (LIFO). Stacks are used extensively at every level of a modern computer system. Stack is a container of nodes and has two basic operations : **push** and **pop**. Push adds a given node to the top of the stack leaving previous nodes below. Pop removes and returns the current top node of the stack.

We can implement a stack of at most n elements with an array $S[1..n]$. The top $[S]$ indexes the most recently inserted element. The stack consists of elements S [1...top [S]], where $S[1]$ is the bottom element of the stack and S [top[S]] is the element at the top. When top[S]=0 the stack contains no elements and is empty. The stack can be tested for emptiness by the operation STACK-EMPTY. If an empty stack is popped, we say the stack **underflows**. If top[S]exceeds n, the stack **overflows**

The Stack operations can be implemented with following procedures

```
STACK-EMPTY (S)
1.  if top[S]=0
2.      then return TRUE
3.      else return FALSE
```



```
PUSH (S, x)
1.  top [S]← top[S]+1
2.  S [top [S]]←x

POP (S)
1.  if STACK-EMPTY(S)
2.      then error "underflow"
3.      else top[S] ← top[S] – 1
4.           return S [top [S] + 1]
```

Each of the three stack operations takes $O(1)$ time.

11.3.1 Applications of Stacks

(a) Polish Notation

For most common arithmetic operations, the operator symbol is placed between its two operands. For example,

$$A+BC-DE*FG/H$$

This is called infix notation. With this notation, we must distinguish between $(A+B)*C$ and $A+(B*C)$ by using either parentheses or some operator-precedence convention such as the usual precedence levels. The order of the operators and operands in an arithmetic expression does not uniquely determine the order in which the operations are to be performed.

Polish notation, named after the Polish mathematician **Jan Lukasiewicz**, refers to the notation in which the operator symbol is placed before its two operands. For example,

$$+AB-CD*EF/GH$$

We translate, step by step, the following infix expressions into Polish notation using brackets [ ] to indicate a partial translation:

$$(A+B)*C=[+AB]*C=*+ABC$$

$$A+(B*C)=A+[*BC]=+A*BC$$

$$(A+B)/(C-D)=[+AB]/[-CD]=/+AB-CD$$

The fundamental property of Polish notation is that the order in which the operations are to be performed is completely determined by the positions of the operators and operands in the expression. Accordingly, one never needs parentheses when writing expressions in Polish notation.

Reverse Polish notation refers to the analogous notation in which the operator symbol is placed after its two operands :

$$AB+CDEF*GH/$$

Again, one never needs parentheses to determine the order of the operations in any arithmetic expression written in reverse Polish notation. This notation is frequently called postfix (or suffix) notation.

The computer usually evaluates an arithmetic expression written in infix notation in two steps. First, it converts the expression to postfix notation, and then it evaluates the postfix expression. In each step, the stack is the main tool that is used to accomplish the given task.

(b) Evaluation of a Postfix Expression

Suppose P is an arithmetic expression written in postfix notation. The following algorithm, which uses a STACK to hold operands, evaluates P.

Algorithm

This algorithm finds the VALUE of an arithmetic expression P written in postfix notation.

1. Add a right parenthesis ")" at the end of P. [This acts as a sentinel.]
2. Scan P from left to right and repeat Steps 3 and 4 for each element of P until the sentinel ")" is encountered.
3. If an operand is encountered, put it on STACK.
4. If an operator x is encountered, then :
   a. Remove the two top elements of STACK, where A is the top element and B is the next-top element.
   b. Evaluate B.A.
   c. Place the result of (b) back on STACK.
   [End of If structure.]

   [End of Step 2 loop.]
5. Set VALUE equal to the top element to STACK.
6. Exit.

We note that, when Step 5 is executed, there should be only one number on STACK.

(c) Transforming Infix Expressions into Postfix Expressions

Let Q be an arithmetic expression written in infix notation. Besides operands and operators, Q may also contain left and right parentheses. We assume that the operators in Q consist only of exponentiations ($\cdot$), multiplications ($*$), divisions ($/$), additions ($+$) and subtractions ($-$), and that they have the usual three levels of precedence as given above.

We also assume that operators on the same level, including exponentiation, are performed from left to right unless otherwise indicated by parentheses. (This is not standard, since expressions may contain unary operators and some languages perform the exponentiation from right to left. However, these assumptions simplify our algorithm.)

The following algorithm transforms the infix expression Q into its equivalent postfix expression P. The algorithm uses a stack to temporarily hold operators and left parentheses. The

postfix expression P will be constructed from left to right using the operands from Q and the operators, which are removed from STACK. We begin by pushing a left parenthesis onto STACK and adding a right parenthesis at the end of Q. The algorithm is completed when STACK is empty.

Algorithm

Suppose Q is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression P.

POLISH (Q, P)

1. Push "(" onto STACK, and add ")" to the end of Q.
2. Scan Q from left to right and repeat Steps 3 to 6 for each element of Q until the STACK is empty :
3. If an operand is encountered, add it to P.
4. If a left parenthesis is encountered, push it onto STACK.
5. If an operator . is encountered, then:
   (a) Repeatedly pop from STACK and add to P each operator (on the top of STACK) which has the same precedence as or higher precedence than.
   (b) Add. To STACK.
   [End of If structure.]
6. If a right parenthesis is encountered, then:
   (a) Repeatedly pop from STACK and add to P each operator (on the top of STACK) until a left parenthesis is encountered.
   (b) Remove the left parenthesis. [Do not add the left parenthesis to P.]
   [End of If structure.]
   [End of Step 2 loop.]
7. Exit

11.4 Queue

A queue is a linear list in which items may be added only at one end and items may be removed-only at the other end. The name "queue" likely comes from the everyday use of the term. Consider a queue of people waiting at a bus stop. Each new person who comes takes his or her place at the end of the line, and when the bus comes, the people at the front of the line board first. Clearly, the first person in the line is the first person to leave. Thus queues are also called **first-in first-out** (FIFO) lists. Another example of a queue is a batch of jobs waiting to be processed, assuming no job has higher priority than the others. A queue is a linear list of elements in which deletions can take place only at one end, called the head (front), and insertions can take place only at the other end, called the tail (rear). The terms "front" and "rear" are used in describing a linear list only when it is implemented as a queue. Queues are also called first-in first-out (FIFO) lists, since the first element in a queue will be the first element out of the queue. In other words, the order in which elements enter a queue is the order in which they leave. This contrasts with stacks, which are last-in first-out (LIFO) lists.

We call the INSERT operation on a queue ENQUEUE and we call the delete operation DEQUEUE. In ENQUEUE and DEQUEUE procedure the error checking for underflow and overflow has been omitted.

```
ENQUEUE (Q, x)
1.  Q [tail[Q]] ← x
2.  if tail [Q] = length [Q]
3.      then tail [Q] ← 1
4.      else tail[Q]←tail [Q] + 1
```

```
DEQUEUE (Q)
1.  x ←Q [ head[Q] ]
2.  if head[Q] = length[Q]
3.      then head[Q] ← 1
4.      else head[Q] ← tail[Q] + 1
5.  return x
```

Both ENQUEUE and DEQUEUE operation takes O (1) time.

Example. *Explain how to implement two stacks in one array A[1..n] in such a way that neither stack overflows unless the total number of elements in both stacks together is n. The PUSH and POP operations should run in O(1) time.*

Solution. Two stacks can be implemented in a single array without overflows occurring if they grow from each end and towards the middle.

Example. *Show how to implement a queue using two stacks. Analyze the running time of the queue operations.*

Solution. Implement a queue using two stacks. Denote the two stacks $S1$ and $S2$. The ENQUEUE operation is simply implemented as a push on $S1$. The dequeue operation is implemented as a pop on $S2$. If $S2$ is empty, successively pop $S1$ and push $S2$. This reverses the order of $S1$ onto $S2$. The worst-case running time is $O(n)$.

```
Q-Insert (x)
1.  if (top = max)
2.      then error "over flow"
3.  else S1[top+1]← x
```

```
Q-delete ( )
1.  if (top = 0)
2.      then error "under flow"
3.      flag ← 0
4.  else t ← 0
5.  while (top < 1)
6.  { x ← POP(S1)
7.      PUSH(S2, x)
8.      t ← t + 1
9.      top ← top - 1
10. }
11. y ← POP(S2)
12. top ← top-1
13. while (t ≠ 0)
14. { x ← POP(S2)
15.     PUSH (S1, x)
16.     top ← top +1
17. }
18. return y
```


Example. *Show how to implement a stack using two queues. Analyze the running time of the stack operations.*

Solution. To implement the stack using two queues $Q1$ and $Q2$, we can simply ENQUEUE elements into $Q1$ whenever a push call is made. This takes $O(1)$ time. For POP calls we can DEQUEUE all elements of $Q1$ and ENQUEUE them into $Q2$ except for the last element which we set aside in a temp variable. We then return the elements to $Q1$ by DEQUEUE from $Q2$ and ENQUEUE into $Q1$. The last element that we set aside earlier is then return as the result of the POP. Thus POP takes $O(n)$ time.

11.4.1 Types of Queues

1. DEQUES

A deque (pronounced either "deck" or "dequeue") is a linear list in which elements can be added or removed at either end but not in the middle. The term deque is a contraction of the name **double-ended queue**.

There are various ways of representing a deque in a computer. Here, we will assume our deque is maintained by a circular array DEQUE with pointers LEFT and RIGHT, which point to the two ends of the deque. We assume that the elements extend from the left end to the right end in the array. The term "circular" comes from the fact that we assume that DEQUE [1] comes after DEQUE [N] in the array. The condition LEFT = NULL will be used to indicate that a deque is empty.

There are *two* kinds of a deque

- **Input-restricted deque** is a deque, which allows insertions at only one end of the list but allows deletions at both ends of the list;
- **Output-restricted deque** is a deque, which allows deletions at only one end of the list but allows insertions at both ends of the list.

2. PRIORITY QUEUES

A priority queue is a collection of elements such that each element has been assigned a priority and such that the order in which elements are deleted and processed comes from the following rules :

1. An element of higher priority is processed before any element of lower priority.
2. Two elements with the same priority are processed according to the order in which they were added to the queue.

A prototype of a priority queue is a timesharing system: programs of high priority are processed first, and programs with the same priority form a standard queue.

11.5 Linked-List

Linked lists and arrays are similar since they both store collections of data. The terminology is that arrays and linked lists store "elements" on behalf of "client" code. The specific type of element is not important since essentially the same structure works to store elements of any type. Linked lists have their own strengths and weaknesses, but they happen to be strong where arrays are weak. The array's features all follow from its strategy of allocating the memory for all its elements in one block of memory. Linked lists use an entirely different strategy. **Linked lists allocate memory for each element separately and only when necessary.**

A **linked list**, or **one-way list**, is a linear collection of data elements, called **nodes**, where the linear order is given by means of **pointers**. That is, each node is divided into two parts: the first part contains the information of the element, and the second part, called the link field or next pointer field, contains the address of the next node in the list.

11.5.1 Searching a node in a linked-list

The procedure LIST-SEARCH (L, k) finds the first element with key k in the list L by a simple linear search, returning a pointer to this element. If no object with key k appears in the list, then NIL is returned.

```
LIST-SEARCH (L, x)
1.  x ← head [L]
2.  while x ≠ NIL and key[x] ≠ k
3.      do x ← next[x]
4.  return x
```

To search a list of n objects, the LIST-SEARCH procedure takes $\theta(n)$ time in the worst case, since it may have to search the entire list.

11.5.2 Inserting into a linked-list

LIST-INSERT procedure inserts x onto the front of the linked list.

```
LIST-INSERT (L, x)
1.  next[x] ← head[L]
2.  if head[L] ≠ NIL
3.      then prev[head[L]] ← x
4.  head[L] ← x
5.  prev[x] ← NIL
```

The running time of LIST-INSERT procedure on a list of n elements is $O(1)$.

11.5.3 Deleting from a linked-list

The procedure LIST-DELETE removes an element x from a linked list L If we wish to delete an element with a given key, we must first call LIST- SEARCH to retrieve a pointer to the element. LIST-DELETE runs in $O(1)$ time , but if we wish to delete an element with a given key, $\theta(n)$ time is required in the worst case because we must first call LIST-SEARCH.

```
LIST-DELETE (L, x )
1.  if prev[x] ≠ NIL
2.      then next[prev[x] ] ← next[x]
3.      else head[L] ← next[x]
4.  if next[x] ≠ NIL
5.  then prev [next[x] ] ← prev [x]
```

11.6 Binary Tree

We represent each node of a tree by an object. As with linked lists, we assume that each node contains a key field. The remaining fields of interest are pointers to other nodes, and they vary according to the type of tree. We use the fields p, *left*, and *right* to store pointers to the parent, left child, and right child of each node in a binary tree T. If $p[x] = \text{NIL}$, then x is the root. If node x has no left child, then $left[x] = \text{NIL}$, and similarly for the right child. The root of the entire tree T is pointed to by the attribute $root[T]$. If $root[T] = \text{NIL}$, then the tree is empty.

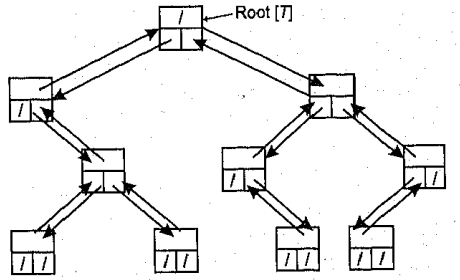


Figure 11.4 The representation of a binary tree T. Each node x has the fields $p[x]$ (top), left[x] (lower left), and right[x] (lower right). The key fields are not shown.

Exercise

1. Show how to implement a queue using two stacks. Analyze the running time of the queue operations.

2. A binary tree has 9 nodes. The Inorder and Preorder traversal of tree yield the following sequence of nodes :

   INORDER : E A C K F H D B G

   PREORDER : F A E K C D H G B

   Draw the tree.

3. Write an algorithm for insertion and deletion of elements for a queue. Use a boolean variable to distinguish between a queue being empty or full.
4. Write a C program that will split a circularly linked list into two circularly linked lists.
5. Convert the following infix expressions to the prefix expressions. Show all steps :

   (*i*) $A \hat{} B * C - D + E / F / (G + H)$

   (*ii*) $(A + B) * (C \hat{} (D - E) + F) - G$
6. Write a "C" program using stack to check whether a string is palindrome or not. Do not define empty, push, pop functions.

CHAPTER

12

Binary Search Tree

A binary search tree is organized in a binary tree. Such a tree can be represented by a linked data structure in which each node is an object. In addition to a *key* field, each node contains fields *left*, *right*, and *p* that point to the nodes corresponding to its left child, its right child, and its parent, respectively. If a child or the parent is missing, the appropriate field contains the value NIL. The root node is the only node in the tree whose parent field is NIL. The keys in a binary search tree are always stored in such a way as to satisfy the **binary-search-tree property**.

12.1 Binary-Search-Tree Property

Let x be a node in a binary search tree.

- If y is a node in the left sub tree of x, then key $[y] \leq$ key $[x]$.
- If y is a node in the right sub tree of x, then key $[x] \leq$ key $[y]$.

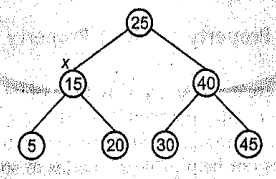


Figure 12.1

In this tree key $[x] = 15$

- If y is a node in the left subtree of x then key $[y] = 5$.

  i.e., key $[y] \leq$ key $[x]$

and

- If y is a node in the right subtree of x then key $[y] = 20$.

  i.e., key $[x] \leq$ key $[y]$

The binary-search-tree property allows us to print out all the keys in a binary search tree in sorted order by a simple recursive algorithm, called an *inorder tree walk*. This algorithm derives its name from the fact that the key of the root of a sub tree is printed between the values in its left sub tree and those in its right sub tree. (Similarly, a *preorder tree walk* prints the root before the values in either sub tree, and a *post order tree walk* prints the root after the values in its sub trees.)

```
INORDER-TREE-WALK (x)
(During this type of walk, we visit the root of a sub tree between the left sub tree visit
and right sub tree visit.)
1.  if x ≠ NIL
2.      then INORDER-TREE-WALK (left[x])
3.           print key[x]
4.           INORDER-TREE-WALK (right[x])
```

It takes $\theta(n)$ time to walk a tree of n nodes. Note that the Binary Search Tree property allows us to print out all the elements in the Binary Search Tree in sorted order.

```
PREORDER-TREE-WALK (x)
(In which we visit the root node before the nodes in either sub tree)
1.  if x ≠ NIL
2.      then print key[x]
3.           PREORDER-TREE-WALK (left[x])
4.           PREORDER-TREE-WALK (right[x])
```

```
POSTORDER-TREE-WALK (x)
(In which we visit the root node after the nodes in its sub trees)
1.  if x ≠ NIL
2.      then POSTORDER-TREE-WALK (left[x])
3.           POSTORDER-TREE-WALK (right[x])
4.           print key[x]
```

It takes $O(n)$ time to walk (inorder, preorder and postorder) a tree of n nodes

12.2 Binary-Search-Tree Property vs Heap Property

In a heap, a nodes key is greater than equal to both of its children's keys. In binary search tree, a node's key is greater than or equal to its child's key but less than or equal to right child's key. Furthermore, this applies to entire sub tree in the binary search tree case. It is very important to note that the heap property does not help print the nodes in sorted order because this property does not tell us in which sub tree the next item is. If the heap property could be used to print the keys in sorted order in $O(n)$ time, this would contradict our known lower bound on comparison sorting.

The last statement implies that since sorting n elements takes $\Omega(n \lg n)$ time in the worst case in the comparison model, any comparison-based algorithm for constructing a Binary Search Tree from arbitrary list n elements takes $\Omega(n \lg n)$ time in the worst case.

We can show the validity of this argument (in case you are thinking of beating $\Omega(n \lg n)$ bound) as follows : let $c(n)$ be the worst-case running time for constructing a binary tree of a set of n elements. Given an n-node BST, the INORDER walk in the tree outputs the keys in sorted order (shown above). Since the worst-case running time of any computation based sorting algorithm is $\Omega(n \lg n)$, we have

$$c(n) + O(n) = \Omega(n \lg n)$$

Therefore, $$c(n) = \Omega(n \lg n).$$

12.3 Querying a Binary Search Tree

The most common operation performed on a binary search tree is searching for a key stored in the tree. Besides the SEARCH operation, binary search trees can support such queries as MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR. These operations run in $O(h)$ time where h is the height of the tree i.e., h is the number of links root node to the deepest node.

12.3.1 Searching

The TREE-SEARCH (x, k) algorithm searches the tree root at x for a node whose key value equals k. It returns a pointer to the node if it exists otherwise NIL

```
TREE-SEARCH (x, k)
1.  if x = NIL or k = key[x]
2.      then return x
3.  if k < key[x]
4.      then return TREE-SEARCH (left[x], k)
5.      else return TREE-SEARCH (right[x], k)
```

Clearly, this algorithm runs in $O(h)$ time where h is the height of the tree. The iterative version of above algorithm is very easy to implement

```
ITERATIVE-TREE-SEARCH (x, k)
1.  while x ≠ NIL and k ≠ key[x]
2.      do if k < key[x]
3.           then x ← left[x]
4.           else x ← right[x]
5.  return x
```

12.3.2 Minimum and Maximum

An element in a binary search tree whose key is a minimum can always be found by following *left* child pointers from the root until a NIL is encountered. The following procedure returns a pointer to the minimum element in the sub tree rooted at a given node x.

```
TREE-MINIMUM (x)
1.  while left[x] ≠ NIL
2.      do x ← left[x]
3.  return x
```

The pseudocode for TREE-MAXIMUM is symmetric.

```
TREE-MAXIMUM (x)
1.  while right[x] ≠ NIL
2.      do x ← right[x]
3.  return x
```

Both of these procedures run in $O(h)$ time on a tree of height h, since they trace paths downward in the tree.

12.3.3 Successor and predecessor

Given a node in a binary search tree, it is sometimes important to be able to find its successor in the sorted order determined by an inorder tree walk. If all keys are distinct, the successor of a node x is the node with the smallest key greater than $key[x]$. The structure of a binary search tree allows us to determine the successor of a node without ever comparing keys. The following procedure returns the successor of a node x in a binary search tree if it exists, and NIL if x has the largest key in the tree.

```
TREE SUCCESSOR (x)
1.  if right[x] ≠ NIL
2.      then return TREE-MINIMUM(right[x])
3.  y ← p[x]
4.  while y ≠ NIL and x = right[y]
5.      do x ← y
6.         y ← p[y]
7.  return y
```

The code for TREE-SUCCESSOR is broken into two cases. If the right sub tree of node x is nonempty, then the successor of x is just the left-most node in the right sub tree, which is found in line 2 by calling TREE-MINIMUM ($right$ [x]). On the other hand, if the right sub tree of node x is empty and x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor of x. To find y, we simply go up the tree from x until we encounter a node that is the left child of its parent; lines 3-7 of TREE-SUCCESSOR accomplish this.

The running time of TREE-SUCCESSOR on a tree of height h is $O(h)$, since we either follow a path up the tree or follow a path down the tree. The procedure TREE-PREDECESSOR, which is symmetric to TREE-SUCCESSOR, also runs in time $O(h)$.

12.3.4 Insertion and deletion

Insertion

To insert a new value v into a binary search tree T, we use the procedure TREE-INSERT. The procedure is passed a node z for which $key[z] = v$, $left[z] = NIL$, and $right[z] = NIL$. It modifies T and some of the fields of z in such a way that z is inserted into an appropriate position in the tree.


```
TREE-INSERT (T, z)
1.  y ← NIL
2.  x ← root[T]
3.  while x ≠ NIL
4.      do y ← x
5.         if key[z] < key[x]
6.            then x ← left[x]
7.            else x ← right[x]
8.  p[z] ← y
9.  if y = NIL
10.    then root[T] ← z
11.    else if key[z] < key[y]
12.            then left[y] ← z
13.            else right[y] ← z
```

Like the procedures TREE-SEARCH and ITERATIVE-TREE-SEARCH, TREE-INSERT begins at the root of the tree and traces a path downward. The pointer x traces the path, and the pointer y is maintained as the parent of x. After initialization, the while loop in lines 3-7 causes these two pointers to move down the tree, going left or right depending on the comparison of $key[z]$ with $key[x]$, until x is set to NIL. This NIL occupies the position where we wish to place the input item z. Lines 8-13 set the pointers that cause z to be inserted.

Like the other primitive operations on search trees, the procedure TREE-INSERT runs in $O(h)$ time on a tree of height h.

Sorting

We can sort a given set of n numbers by first building a binary search tree containing these numbers by using TREE-INSERT (x) procedure repeatedly to insert the numbers one by one and then printing the numbers by an inorder tree walk. In best-case running time printing takes $O(n)$ time and n insertion cost $O(\lg n)$ each (tree is balanced, half the insertions are at depth $\lg(n)-1$). This gives the best-case running time $O(n \lg n)$. In worst-case running time printing still takes $O(n)$ time and n insertion costing $O(n)$ each (tree is a single chain of nodes) is $O(n^2)$. The n insertion cost $1, 2, 3, \ldots n$, which is arithmetic sequence so it is $n^2/2$.

Deletion

When deleting a node from a tree it is important that any relationships, implicit in the tree, be maintained. The deletion of nodes from a binary search tree will be considered. Three distinct cases can be identified:

1. ***Nodes with no children.*** This case is trivial. Simply set the parent's pointer to the node to be deleted to nil and delete the node.

2. ***Nodes with one child.*** This case is also easy to deal with. The single child of the node to be deleted becomes the child of its grandparent through the pointer that currently points to its parent, which is to be deleted. Examining the relationship between the grandparent node and its child, which is to be deleted, can see that this preserves the tree relationships. If the node to be deleted is a

left child then it is less than its parent. Whether or not the child of this node is left or right does not matter since by its place in the tree as the child of a left child it must be less than its grandparent and so can replace its parent in that position.

3. *Nodes with two children.* This case is a little more difficult since two sub trees need to be re-attached and both must maintain the same relation to the parent of the node being deleted and to each other. This can only be achieved if a new node is found to replace the deleted node as the root of these two sub trees. Whatever node is to become parent to these two hanging sub trees it must have the property that it is greater than any node in the left sub tree and less than any node in the right sub tree.

If a binary search tree is examined, it can be seen that there are just two possible nodes which can satisfy these conditions: the rightmost node in the left sub tree and the leftmost node in the right sub tree, hence one of these must replace the node to be deleted.

Inorder to find this node, move to the left and then keep searching right until there is a nil pointer. The node with the nil pointer is the node to be moved. The same process can be done by moving to the right of the node to be deleted and search left until a nil pointer is found.

Having come to this conclusion the best way to achieve this without wholesale reassignment of pointers is to copy the contents of the node doing the replacing into the node to be deleted. This effectively deletes the specified node and replaces it with an appropriate root for its two sub trees. All that remains is to delete the extra copy of the replacing node. This is going to be one of the other two trivial cases and so is easily handled.

The code for TREE-DELETE organizes these three cases a little differently.

```
TREE-DELETE (T, z)
1.  if left[z] = NIL or right[z] = NIL
2.      then y ← z
3.      else y ← TREE-SUCCESSOR(z)
4.  if left[y] ≠ NIL
5.      then x ← left[y]
6.      else x ← right[y]
7.  if x ≠ NIL
8.      then p[x] ← p[y]
9.  if p[y] = NIL
10.     then root[T] ← x
11.     else if y = left[p[y]]
12.         then left[p[y]] ← x
13.         else right[p[y]] ← x
14. if y ≠ z
15.     then key[z] ← key[y]
16.     if y has other fields, copy them, too.
17. return y
```

The procedure runs in $O(h)$ time on a tree of height h.

Example. *What is the difference between the binary-search-tree property and the heap property? Can the heap property be used to print out the keys of an n-node tree in sorted order in $O(n)$ time ? Explain how or why not ?*

Solution. The definitions clearly differ. If the heap property allowed the elements to be printed in sorted order in time $O(n)$, we would have an $O(n)$ time comparison sorting algorithm since BUILD-HEAP takes $O(n)$ time. This, however, is impossible since we know $\Omega(n \lg n)$ is a lower bound for sorting.

Example. *Show that if a node in a binary search tree has two children then its successor has no left child and its predecessor has no right child.*

Solution. Let v is a node with two children. The nodes that immediately precede v must be in the left sub tree and the nodes that immediately follow v must be in the right sub tree. Thus the successor s must be in the right sub tree and s will be the next node from v in an inorder walk. Therefore s cannot have a left child since this child since this would come before s in the inorder walk. Similarly, the predecessor has no right child.

Example. *Show that the inorder walk of a n-node binary search tree implemented with a call to TREE-MINIMUN followed by $n-1$ calls to TREE-SUCCESSOR takes $O(n)$ time.*

Solution. Consider the algorithm at any given node during the algorithm. The algorithm will never go to the left sub tree and going up will therefore cause it to never return to this same node. Hence the algorithm only traverses each edge at most twice and therefore the running time is $O(n)$.

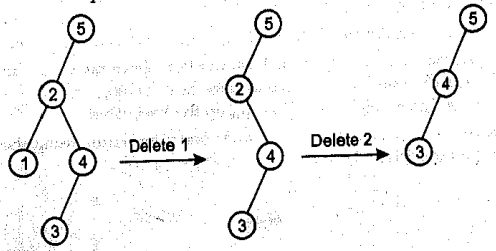
Example. *Give a recursive version of the TREE-INSERT procedure.*

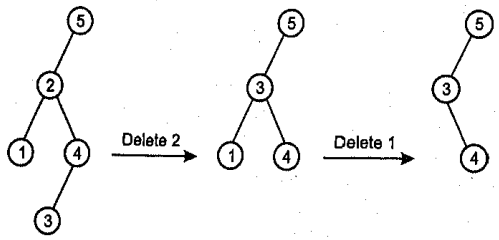
Solution.

```
TREE-INSERT(z, k)
1.  if z = NIL
2.      then key[z] ← k
3.      left[z] ← NIL
4.      right[z] ← NIL
5.  else if k < key[z]
6.      then TREE-INSERT(left[z], k)
7.      else TREE-INSERT(right[z], k)
```

Example. *Is the operation of deletion "commutative" in the sense that deleting x and then y from a binary search tree leaves the same tree as deleting y and then x ? Argue why it is or give a counter example.*

Solution. The deletion operation is not commutative. A counterexample is shown in the Fig.





Exercise

1. Suppose that we have numbers between 1 and 1000 in a binary search tree and want to search for the number 363. Which of the following sequences could not be the sequence of nodes examined ?
   (*a*) 2, 252, 401, 398, 330, 344, 397, 363. (*b*) 924, 220, 911, 244, 898, 258, 362, 363.
   (*c*) 925, 202, 911, 240, 912, 245, 363. (*d*) 2, 399, 387, 219, 266, 382, 381, 278, 363.
   (*e*) 935, 278, 347, 621, 299, 392, 358, 363.
2. A random binary search tree having n nodes, where $n = 2^k - 1$ for some positive integer k, is to be reorganized into a perfectly balanced binary search tree. Outline an efficient algorithm for this task. The algorithm should not use any memory locations other than those already used by the random binary search tree.
3. Consider three keys, k_1, k_2, k_3 such that $k_1 < k_2 < k_3$. A binary search tree is constructed with these three keys. Depending on the order in which the keys are inserted, five different binary search trees are possible. Write down the five binary search trees.
4. Given a set of 31 names, each containing 10 uppercase alphabets, we wish to set up a data structure that would lead to efficient average case performance of successful and unsuccessful searches on this set. It is known that not more than 5 names start with the same alphabet. Also, assume that successful and unsuccessful searches are equally likely and that in the event of a successful search, it is equally likely that any of the 31 names was searched for. The two likely choices for the data structure are :
   - A closed hash table with 130 locations where each location can accommodate one name.
   - A full binary search tree of height 4, where each of the 31 nodes contains a name and lexicographic ordering is used to set up the BST.
5. Answer the following questions :
   (*a*) What is the hashing function you would use for the closed hash table ?
   (*b*) Assume that the computation of the hash function above takes the same time as comparing two given names. Now, compute the average case running time of a search operation using the hash table. Ignore the time for setting up the hash table.
   (*c*) Compute the average case running time of a search operation using the BST. Ignore the time for setting up the BST.

CHAPTER 13

AVL Tree

Height Balanced Tree

13.1 Introduction

A height balanced tree is one in which the difference in the height of the two subtrees for any node is less than or equal to some specified amount. One type of height-balanced tree is the **AVL-tree** named after its originators : *Adel'son-Vel'ski* and *Landis*.

In this tree, the height difference may be no more than 1. In fact for an AVL tree it will never be greater than $1.44 \lg (n+2)$. Because of this, the AVL tree provides a method to ensure that tree searches always stay close to the theoretical minimum of $O(\lg n)$ and never degenerate to $O(n)$.

Definition

- *An empty binary tree B is an AVL tree.*
- If B is the non-empty binary tree with B_L and B_R are its left and right subtrees then B is an AVL tree if and only if
  (*i*) B_L and B_R are AVL trees, and
  (*ii*) $|h_L - h_R| \leq 1$, where h_L and h_R are the heights of B_L and B_R subtrees, respectively.

Exercise

1. Insert items with the following (in the given order) into an initially empty splay tree and draw the tree after each operation :

   0, 2, 4, 6, 8, 11, 13, 15, 18, 20, 25

2. What does a splay tree look like if keys are accessed in increasing order ?

3. Describe how to implement operation JOIN (S_1, S_2) which returns a splay tree for the UNION of the elements in the splay trees. S_1 and S_2 under the condition that for all keys x in S_1 and all keys y in S_2 we have $x < y$.

   Splay trees S_1 and S_2 are destroyed by this operation.

CHAPTER 15

Red-Black Trees

15.1 Introduction

A **red-black tree** is a type of self-balancing binary search tree.It was invented in 1972 by Rudolf Bayer who called them "**symmetric binary B-trees**".

A red-black tree is a binary tree where each node has color as an extra attribute, either red or black. By constraining the coloring of the nodes it is ensured that the longest path from the root to a leaf is no longer than twice the length of the shortest path. This means that the tree is balanced. A red-black tree must satisfy these ***properties*** :

1. The root is always black.
2. A nil is considered to be black. This means that every non-NIL node has two children.
3. Black Children Rule : The children of each red node are black.
4. Black Height Rule : For each node v, there exists an integer bh (v) such that each downward path from v to a nil has exactly bh (v) black real (*i.e.* non-nil) nodes. Call this quantity the **black height** of v. We define the black height of an RB tree to be the black height of its root.

A tree T is an **almost red-black tree** (ARB tree) if the root is red, but other conditions above hold.

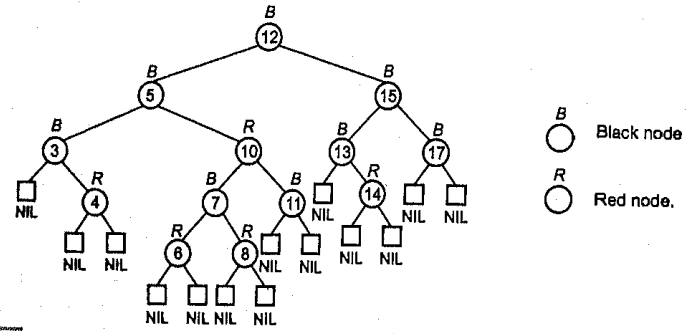


Figure 15.1

Lemma

Let T be an RB tree having some n internal nodes. Then the height of T is at most 2 lg (n+1).

Proof. We first show that the sub tree rooted at any node x contains at least $2^{bh(x)}-1$ internal nodes. We prove this by induction.

If the height of x is 0 then x must be leaf (NIL), and the sub tree rooted at x contains at least $2^{bh(x)}-1=2^0-1=0$ internal nodes.

For inductive step, consider a node x that has positive height and is an internal node with two children. Each child has a black height of either $bh(x)$ or $bh(x)-1$, depending on whether its color is red or black, respectively. Since the height of a child of x is less than the height of x itself. So, we can apply the inductive hypothesis to conclude that each child has at least $2^{bh(x)-1}-1$ internal nodes.

Thus, the sub tree rooted at x contains at least $(2^{bh(x)-1}-1)+(2^{bh(x)-1}-1)+1=2^{bh(x)}-1$ internal nodes.

Let h be the height of T. Let $v_0, v_1, \ldots, v_{h+1}$ be an arbitrary length h downward path from the root to a nil, where v_0 is the root, v_h is the leaf, and v_{h+1} is the nil. v_0 is black and v_{h+1} is black. The number of red nodes among $v_1, \ldots, v_h$ is maximized when for all odd i v_i is red. So, the number of red nodes is at most $h/2$. This means that the number of black ones among $v_1, \ldots, v_h$ is at least $h-h/2$. Thus, $bh(T) \geq h/2$. If x is the root of the tree then $bh(x)$ must be at least $h/2$, thus

$$n \geq 2^{h/2}-1$$

Moving the 1 to the left hand side and taking logarithms on both sides, we get

$$\lg(n+1) \geq h/2$$

By solving this we have $h \leq 2\lg(n+1)$

15.2 Operations on RB Trees

The search-tree operations TREE-INSERT and TREE-DELETE, when run on a red-black tree with n keys, take $O(\lg n)$ time. Because they modify the tree, the result may violate the red-black properties. To restore these properties, we must change the colours of some of the nodes in the tree and also change the pointer structure. We will study *two* operations, *insertion* and *deletion*. The two operations make use of two operations, Left-Rotate and Right-Rotate.

15.2.1 Rotations

Restructuring operations on red-black trees (and many other kinds of trees) can often be expressed more clearly in terms of the **rotation** operation, diagrammed below.

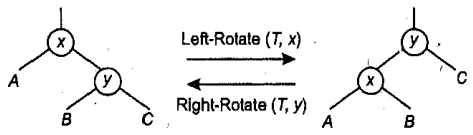
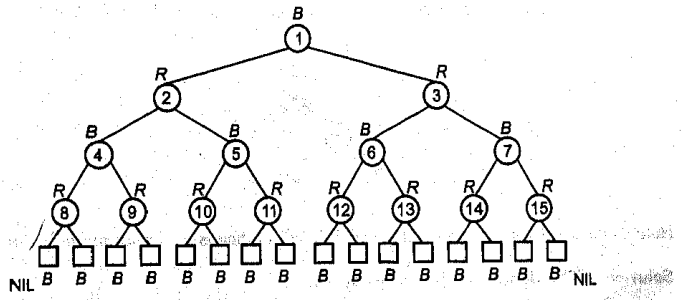


Figure 15.2

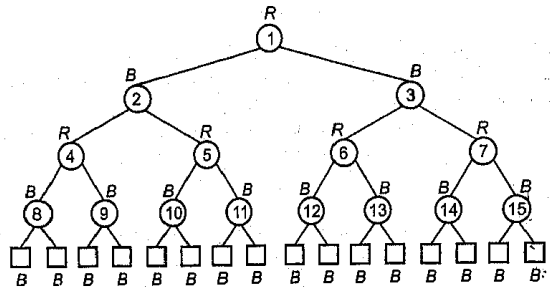
Clearly the order $\langle AxByC \rangle$ is preserved by the rotation operation. Therefore, if we start with a BST, and only restructure using rotations, then we will still have a BST i.e. rotations do not break the BST-property.

Example. *Draw the complete binary tree of height 3 on the keys {1, 2, 3, ..., 15}. Add the NIL leaves and colour the nodes in three different ways such that the black-heights of the resulting trees are: 2, 3 and 4.*

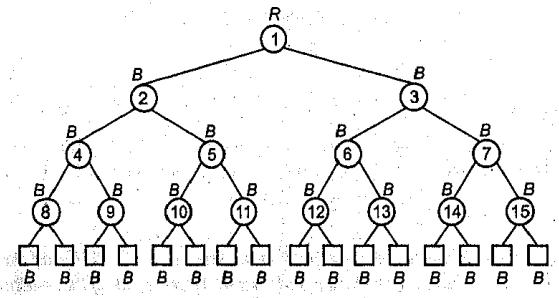
Solution.



Tree with black-height-2

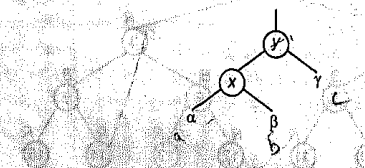


Tree with black-height-3



Tree with black-height-4

Example. *Let a, b, c be arbitrary nodes in subtrees α, β and γ respectively in the tree.*



How do the depths of a, b and c change when a left rotation is performed on node x in the figure?

Solution. The depth of a increases by +1.
The depth of b remains the same.
The depth of c changes by −1.

15.2.2 Insertion

- Insert the new node the way it is done in binary search trees
- Color the node red
- If an inconsistency arises for the red-black tree, fix the tree according to the type of discrepancy.

A discrepancy can result from a parent and a child both having a red color. This type of discrepancy is determined by the location of the node with respect to its grand parent, and the color of the sibling of the parent.

Discrepancies, in which the sibling is red, are fixed by changes in color. Discrepancies, in which the siblings are black, are fixed through AVL-like rotations. Changes in color may propagate the problem up toward the root. On the other hand, at most one rotation is sufficient for fixing a discrepancy.

```
RB-INSERT (T, z)
1.  y ← nil[T]
2.  x ← root[T]
3.  while x ≠ nil[T]
4.      do y ← x
5.          if key[z] < key[x]
6.          then x ← left[x]
7.          else x ← right[x]
8.  p[z] ← y
9.  if y = nil[T]
10.     then root[T] ← z
11.     else if key[z] < key[y]
12.         then left[y] ← z
13.         else right[y] ← z
14. left[z] ← nil[T]
15. right[z] ← nil[T]
16. color[z] ← RED
17. RB-INSERT-FIXUP (T, z)
```

After the insert new node, now the question arises which color do we give the new node. Coloring this new node into black may violate the black-height condition and coloring this new node into red may violate coloring condition *i.e.*, root is black and red node has no red children.

We know the black-height violations are hard. So we color the node red. After this if there is any color violation then we have to correct them by RB-INSERT-FIXUP procedure.

```
RB-INSERT-FIXUP(T, z)
1.  while color[p[z]] = RED
2.      do if p[z] = left[p[p[z]]]
3.          then y ← right[p[p[z]]]
4.              if color[y] = RED
```

```
5.              then color[p[z]] ← BLACK                ▷ Case 1
6.                   color[y] ← BLACK                   ▷ Case 1
7.                   color[p[p[z]]] ← RED               ▷ Case 1
8.                   z ← p[p[z]]                        ▷ Case 1
9.              else if z = right[p[z]]
10.                  then z ← p[z]                      ▷ Case 2
11.                       LEFT-ROTATE (T, z)            ▷ Case 2
12.                  color[p[z]] ← BLACK                ▷ Case 3
13.                  color[p[p[z]]] ← RED               ▷ Case 3
14.                  RIGHT-ROTATE (T, p[p[z]])          ▷ Case 3
15.         else (same as then clause)
                 with "right" and "left" exhanged)
16.  color[root[T]] ← BLACK
```

There are actually six cases to consider in the **while** loop, but three of them are symmetric to the other three, depending on whether x's parent $p[x]$ is a left child or a right child of x's grandparent $p[p[x]]$, which is determined in line 4. We have given the code only for the situation in which $p[x]$ is a left child. We have made the important assumption that the root of the tree is black—a property we guarantee in line 18 each time we terminate—so that $p[x]$ is not the root and $p[p[x]]$ exists.

Case 1 is distinguished from cases 2 and 3 by the color of x's parent's sibling, or "uncle." Line 5 makes y point to x's uncle *right* $[p[p[x]]]$, and a test is made in line 6. If y is red, then case 1 is executed. Otherwise, control passes to cases 2 and 3. In all three cases, x's grandparent $p[p[x]]$ is black, since its parent $p[x]$ is red, and property 3 is violated only between x and $p[x]$.

Case 1 is executed when both $p[x]$ and y are red. Since $p[p[x]]$ is black, we can color both $p[x]$ and y black, thereby fixing the problem of x and $p[x]$ both being red, and color $p[p[x]]$ red, thereby maintaining property 4. The only problem that might arise is that $p[p[x]]$ might have a red parent; hence, we must repeat the **while** loop with $p[p[x]]$ as the new node x.

In cases 2 and 3, the color of x's uncle y is black. The two cases are distinguished by whether x is a right or left child of $p[x]$. In case 2, node x is a right child of its parent. We immediately use a left rotation to transform the situation into case 3, in which node x is a left child. Because both x and $p[x]$ are red, the rotation affects neither the black-height of nodes nor property 4. Whether we enter case 3 directly or through case 2, x's uncle y is black, since otherwise we would have executed case 1. We execute some color changes and a right rotation, which preserve property 4, and then, since we no longer have two red nodes in a row, we are done. The body of the **while** loop is not executed another time, since $p[x]$ is now black.

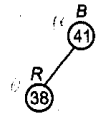
Since the height of a red-black tree on n nodes is $O(\lg n)$, the call to TREE-INSERT takes $O(\lg n)$ time. The **while** loop only repeats if case 1 is executed, and then the pointer x moves up the tree. The total number of times the **while** loop can be executed is therefore $O(\lg n)$. Thus, RB-INSERT takes a total of $O(\lg n)$ time. Interestingly, it never performs more than two rotations, since the **while** loop terminates if case 2 or case 3 is executed.

Example. *Show the red-black trees that result after successively inserting the keys 41, 38, 31, 12, 19, 8 into an initially empty red-black tree.*

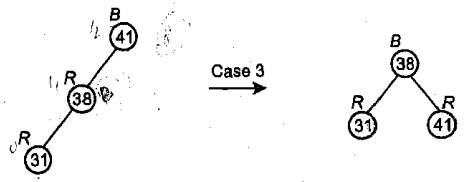
Solution.

◂ Insert 41

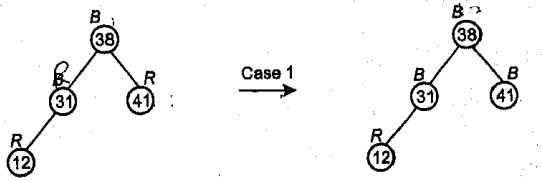
◂ Insert 38



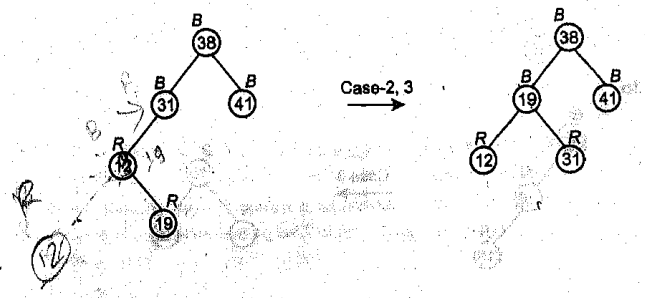
◂ Insert 31



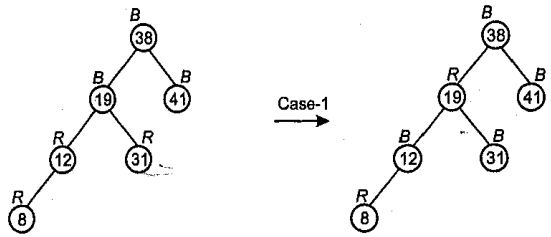
◂ Insert 12



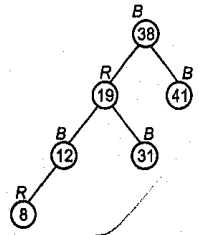
◂ Insert 19



◀| Insert 8



Thus final tree is

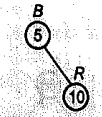


Example. *Show the red-black trees that result after successively inserting the keys 5, 10, 15, 25, 20 and 30 into an initially empty red-black tree.*

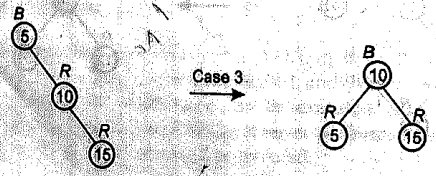
Solution.

◀| Insert 5

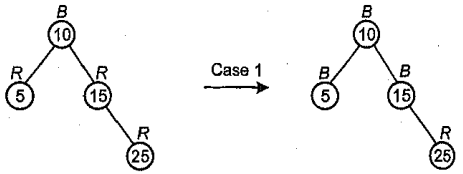
◀| Insert 10



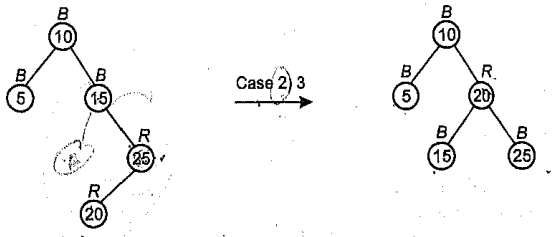
◀| Insert 15



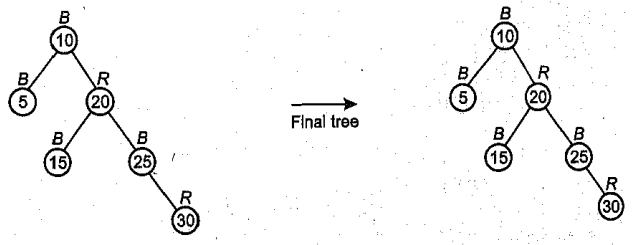
◀| Insert 25



◀| Insert 20



◀| Insert 30



15.2.3 Deletion

First, search for an element to be deleted :

◀| If the element to be deleted is in a node with only left child, swap this node with the one containing the largest element in the left sub tree. (This node has no right child).

◀| If the element to be deleted is in a node with only right child, swap this node with the one containing the smallest element in the right sub tree (This node has no left child).

◀| If the element to be deleted is in a node with both a left child and a right child, then swap in any of the above two ways. While swapping, swap only the keys but not the colors.

- The item to be deleted is now in a node having only a left child or only a right child. Replace this node with its sole child. This may violate red constraint or black constraint. Violation of red constraint can be easily fixed.
- If the deleted node is black, the black constraint is violated. The removal of a black node y causes any path that contained y to have one fewer black node.
- *Two* cases arise :
  1. The replacing node is red, in which case we merely color it black to make up for the loss of one black node.
  2. The replacing node is black.

The procedure RB-DELETE is a minor modification of the TREE-DELETE procedure. After splicing out a node, it calls an auxiliary procedure RB-DELETE-FIXUP that changes colors and performs rotations to restore the red-black properties.

RB-DELETE (T, z)

```
1.  if left[z] = nil[T] or right[z] = nil[T]
2.      then y ← z
3.      else y ← TREE-SUCCESSOR(z)
4.  if left[y] ≠ nil[T]
5.      then x ← left[y]
6.      else x ← right[y]
7.  p[x] ← p[y]
8.  if p[y] = nil[T]
9.      then root[T] ← x
10.     else if y = left[p[y]]
11.         then left[p[y]] ← x
12.         else right[p[y]] ← x
13. if y ≠ z
14.     then key[z] ← key[y]
15.         copy y's satellite data into z
16. if color[y] = BLACK
17.     then RB-DELETE-FIXUP (T,x)
18. return y
```

There are three differences between the procedures TREE-DELETE and RB-DELETE. First, all references to NIL in TREE-DELETE have been replaced by references to the sentinel $nil[T]$ in RB-DELETE. Second, the test for whether x is NIL in line 7 of TREE-DELETE has been removed, and the assignment $p[x] \leftarrow p[y]$ is performed unconditionally in line 7 of RB-DELETE. Thus, if x is the sentinel $nil[T]$, its parent pointer points to the parent of the spliced-out node y. Third, a call to RB-DELETE-FIXUP is made in lines 16-17 if y is black. If y is red, the red-black properties still hold when y is spliced out, since no black-heights in the tree have changed and no red nodes have been made adjacent. The node x passed to RB-DELETE-FIXUP is the node that was y's sole child before y was spliced out if y had a non-NIL child, or the sentinel $nil[T]$ if y had no children. In the latter case,

the unconditional assignment in line 7 guarantees that x's parent is now the node that was previously y's parent, whether x is a key-bearing internal node or the sentinel $nil[T]$.

We can now look at how the procedure RB-DELETE-FIXUP restores the red-black properties to the search tree.

RB-DELETE-FIXUP (T, x)

```
1.  while x ≠ root[T] and color[x] = BLACK
2.      do if x = left[p[x]]
3.          then w ← right[p[x]]
4.              if color[w] = RED
5.                  then color[w] ← BLACK                          ▷ Case 1
6.                      color[p[x]] ← RED                          ▷ Case 1
7.                      LEFT-ROTATE (T, p[x])                      ▷ Case 1
8.                      w ← right[p[x]]                            ▷ Case 1
9.              if color[left[w]] = BLACK and color[right[w]] = BLACK
10.                 then color[w] ← RED                            ▷ Case 2
11.                     x ← p[x]                                   ▷ Case 2
12.                 else if color[right[w]] = BLACK
13.                     then color[left[w]] ← BLACK                ▷ Case 3
14.                         color[w] ← RED                         ▷ Case 3
15.                         RIGHT-ROTATE (T, w)                    ▷ Case 3
16.                         w ← right[p[x]]                        ▷ Case 3
17.                     color[w] ← color[p[x]]                     ▷ Case 4
18.                     color[p[x]] ← BLACK                        ▷ Case 4
19.                     color[right[w]] ← BLACK                    ▷ Case 4
20.                     LEFT-ROTATE (T, P[x])                      ▷ Case 4
21.                     x ← root[T]                                ▷ Case 4
22.         else (same as then clause with "right" and "left" exchanged)
23. color[x] ← BLACK
```

Since the height of a red-black tree of n nodes is $O(\lg n)$, the total cost of the procedure without the call to RB-DELETE-FIXUP takes $O(\lg n)$ time. Within RB-DELETE-FIXUP, cases 1, 3, and 4 each terminate after performing a constant number of color changes and at most three rotations. Case 2 is the only case in which the while loop can be repeated, and then the pointer x moves up the tree at most $O(\lg n)$ times and no rotations are performed. Thus, the procedure RB-DELETE-FIXUP takes $O(\lg n)$ time and performs at most three rotations, and the overall time for RB-DELETE is therefore also $O(\lg n)$.

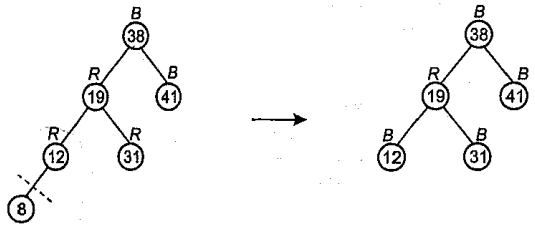
15.3 Elementary Properties of Red-Black Tree

1. Black-height of an red-black tree is the black height of its root. This equals the number of black edges to reach a leaf.
2. If Red-black tree has black-height 'bh' :
   (a) then it has atleast $2^{bh} - 1$ internal black nodes.
   (b) it has atleast $4^{bh} - 1$ internal nodes.

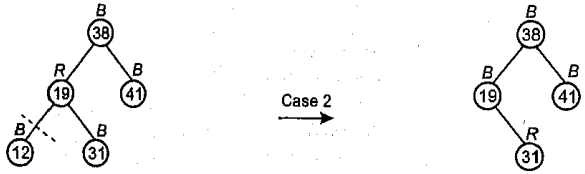
Example. *In previous exercise, we found the red-black tree that results from successively inserting the keys 41, 38, 31, 12, 19, 8 into an initially empty tree. Now show the red-black trees that result from the successive deletion of the keys in the order 8, 12, 19, 31, 38, 41.*

Solution.

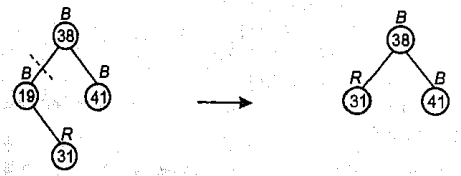
◂ Delete 8



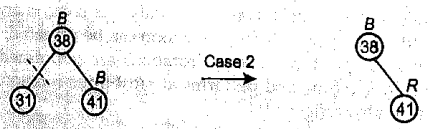
◂ Delete 12



◂ Delete 19



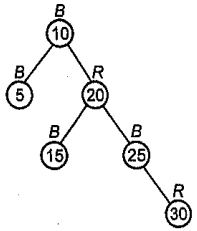
◂ Delete 31



◂ Delete 38

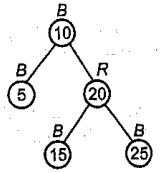
◂ Delete 41 No Tree.

Example. *Show the red-black tree that results from successively deleting the keys 30, 25, 20, 15, 10 and 5 from the final tree.*

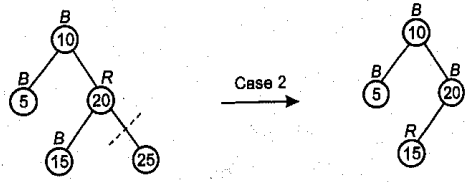


Solution.

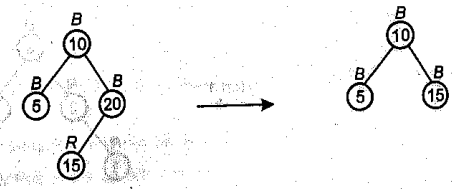
◂ Delete 30



◂ Delete 25

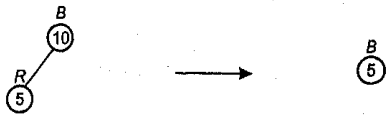


◂ Delete 20



◂ Delete 15

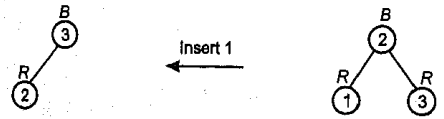
◂ Delete 10



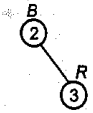
◂ Delete 5. No tree.

Example. *Suppose that a node x is inserted into a red-black tree with RB-INSERT and then immediately deleted with RB-DELETE. Is the resulting red-black tree the same as the initial red-black tree? Justify your anwer?*

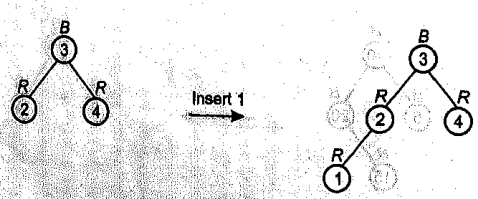
Solution. Inserting and immediately deleting need not yield the same tree. Here is an example that alters the structure and one that changes the color.



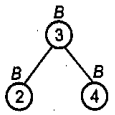
Now delete 1, we get



Again,



Now, delete 1, we get



Example. *Suppose that the root of a red-black tree is red. If we make it black, does the tree remain a red black tree?*

Solution. Coloring the root node black can obviously not violate any property.

Example. *Show that the longest simple path from a node x in a red black tree to a descendant leaf has length at most twice that of the shortest simple path from node x to a descendant leaf?*

Solution. By property 4 the longest and shortest path must contain the same number of black nodes. By property 3 every other nodes in the longest path must be black and therefore the length is at most twice that of the shortest path.

Example. *What is the largest possible number of internal nodes in a red black tree with black height k? What is the smallest possible number?*

Solution. Consider a red-black tree with black-height k. If every node is black, the total number of internal nodes is $2k-1$. If only every other node is black, we can construct a tree with $2^{2k}-1$ node.

Example. *In line 2 of RB-INSERT, we set the color of the newly inserted node x to red. Notice that if we had chosen to set x's color to black, then property 3 of a red-black tree would not be violated. Why didn't we choose to set x's color to black*

Solution. If we choose to set the color of a newly inserted node to black then property 3 is not violated but clearly property 4 is violated.

Exercise

1. Show how an AVL tree can be colored as a red-black tree.
2. Draw the red-black tree resulting from inserting the numbers 5, 16, 22, 45, 2, 10, 18, 30, 50, 12, 1 in this order.
3. A subtree of a red-black tree is red-black. Right or wrong, if wrong, provide counter example.
4. Draw a red-black that is not an AVL-tre.

5. Explain how to use a red-black tree to sort n comparable entries in $O(n \lg n)$ time in the worst case.
6. If the common black depth of a red-black tree is B what is the minimum number of black nodes.
7. Show the result of inserting 2, 1, 4, 5, 9, 3, 6, 7 in this order into an empty RB-tree. Do the same for a AVL tree.
8. You are adding an entry to a RB-tree. You are at a node whose children are both red, and the right child of the left child is also red. What steps do you take ? What information do you pass to the next level up ?
9. You are removing an entry from a RB-tree. All paths through the left child of the current node are short by one black node. The right child is red. What steps do you take ? Will you need to do anything at the next level up ?
10. After an addition in a red-black tree, what is the maximum number of rotations required to rebalance the tree ?
11. After a remove in RB tree, what is the maximum number of rotations required to rebalance the tree ?
12. What is the maximum height of a RB tree with 1,000,000 values ?

CHAPTER

16

Augmenting Data Structures

By augmenting already existing data structures one can build new data structures. Augmenting a data structure is not always straightforward, however, since the added information must be updated and maintained by the ordinary operations on the data structure.

16.1 Augmenting a red-black tree

For each node x, add a new field size(x), the number of non-NIL nodes in the sub tree rooted at x. Now with the size information, we can fast compute the dynamic order statistics and the rank, the position in the linear order.

An ***order-statistic tree*** T is simply a red-black tree with additional information stored in each node. Besides the usual red-black tree fields $key[x]$, $color[x]$, $p[x]$, $left[x]$, and $right[x]$ in a node x, we have another field $size[x]$. This field contains the number of (internal) nodes in the subtree rooted at x (including x itself), that is, the size of the subtree.

If we define *size*[NIL] to be 0, then we have the identity

$$size[x] = size[left[x]] + size[right[x]] + 1$$

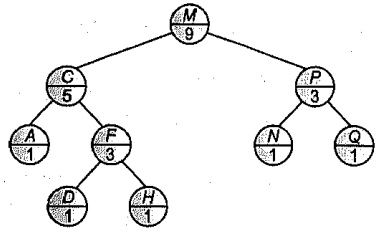


Figure 16.1

16.2 Retrieving an element with a given rank

The procedure OS-SELECT(x, i) returns a pointer to the node containing the ith smallest key in the sub tree rooted at x. To find the ith smallest key in an order-statistic tree T, we call OS-SELECT(*root*[T], i).

```
OS-SELECT(x,i)
1.  r ← size[left[x]] +1
2.  if i = r
3.      then return x
4.  elseif i < r
5.      then return OS-SELECT(left[x], i)
6.  else return OS-SELECT(right[x], i − r)
```

The value of *size*[*left*[x]] is the number of nodes that come before x in an inorder tree walk of the subtree rooted at x. Thus, *size*[*left*[x]]+1 is the rank of x within the subtree rooted at x.

In line 1 of OS-SELECT, we compute r, the rank of node x within the subtree rooted at x. If $i = r$, then node x is the ith smallest element, so we return x in line 3. If $i < r$, then the ith smallest element is in x's left sub tree, so we recurse on *left*[x] in line 5. If $i > r$, then the ith smallest element is in x's right subtree. Since there are r elements in the subtree rooted at x that come before x's right subtree in an inorder tree walk, the ith smallest element in the subtree rooted at x is the $(i-r)$th smallest element in the subtree rooted at *right*[x]. This element is determined recursively in line 6.

Running time O(height)

To see how OS-SELECT operates, consider an example

OS-SELECT (root, 5)

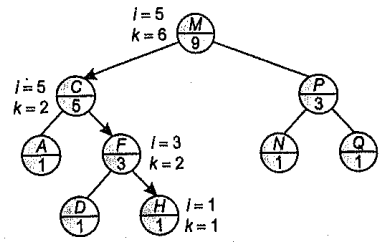


Figure 16.2

16.3 Determining the rank of an element

Given a pointer to a node x in an order-statistic tree T, the procedure OS-RANK returns the position of x in the linear order determined by an inorder tree walk of T.

```
OS-RANK (T, x)
1.  r ← size[left[x]] +1
2.  y ← x
3.  while y ≠ root[T]
4.      do if y = right[p[y]]
5.          then r ← r + size[left[p[y]]] +1
6.          y ← p[y]
7.  return r
```

Since each iteration of the **while** loop takes $O(1)$ time, and y goes up one level in the tree with each iteration, the running time of OS-RANK is at worst proportional to the height of the tree: $O(\lg n)$ on an n-node order-statistic tree.

16.4 Data Structure Maintenance

Example. *Why not keep the ranks themselves in the nodes instead of sub tree sizes ?*

Solution. They are hard to maintain when the tree is modified.

Given the *size* field in each node, OS-SELECT and OS-RANK can quickly compute order-statistic information. But unless these fields can be efficiently maintained by the basic modifying operations on red-black trees. We shall now show that sub tree sizes can be maintained for both **insertion** and **deletion** without affecting the asymptotic running times of either operation.

We know that insertion into a red-black tree consists of *two* phases. The first phase goes down the tree from the root, inserting the new node as a child of an existing node. The second phase goes up the tree, changing colors and ultimately performing rotations to maintain the red-black properties.

To maintain the subtree sizes in the first phase, we simply increment *size*[x] for each node x on the path traversed from the root down toward the leaves. The new node added gets a *size* of 1. Since there are $O(\lg n)$ nodes on the traversed path, the additional cost of maintaining the *size* fields is $O(\lg n)$.

In the second phase, the only structural changes to the underlying red-black tree are caused by rotations, of which there are at most two. Moreover, a rotation is a local operation: it invalidates only the two *size* fields in the nodes incident on the link around which the rotation is performed. Referring to the code for LEFT-ROTATE (T, x) in, we add the following lines :

```
13 size[y] ← size[x]
14 size[x] ← size[left[x]] + size[right[x]] +1
```

Thus, the size has to be changed for only one node :

- the left-child of the rotated node in the case of right rotation and
- the right-child of the rotated node in the case of left rotation.

Since at most two rotations are performed during insertion into a red-black tree, only $O(1)$ additional time is spent updating *size* fields in the second phase. Thus, the total time for insertion into an n-node order-statistic tree is $O(\lg n)$-asymptotically the same as for an ordinary red-black tree.

Deletion from a red-black tree also consists of two phases: the first operates on the underlying search tree, and the second causes at most three rotations and otherwise performs no structural changes. The first phase splices out one node y. To update the sub tree sizes, we simply traverse a path from node y up to the root, decrementing the *size* field of each node on the path. Since this path has length $O(\lg n)$ in an n-node red-black tree, the additional time spent maintaining *size* fields in the first phase is $O(\lg n)$. The $O(1)$ rotations in the second phase of deletion can be handled in the same manner as for insertion.

Thus, both insertion and deletion, including the maintenance of the *size* fields, take $O(\lg n)$ time for an n-node order-statistic tree.

16.5 An Augmentation Strategy

16.5.1 How to augment a data structure

Augmenting a data structure can be broken into *four* steps :

1. choosing an underlying data structure,
2. determining what kind of additional information should be maintained in the underlying data structure,
3. verify that the additional information can be maintained during the execution of each basic modifying operation of the underlying data structure, and
4. developing new operations.

We followed these steps in to design our order-statistic trees. For step 1, we chose red-black trees as the underlying data structure. A clue to the suitability of red-black trees comes from their efficient support of other dynamic-set operations on a total order, such as MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR.

For step 2, we provided the *size* fields, which in each node x stores the size of the subtree rooted at x. Generally, the additional information makes operations more efficient. For example, we could have implemented OS-SELECT and OS-RANK using just the keys stored in the tree, but they would not have run in $O(\lg n)$ time.

For step 3, we ensured that insertion and deletion could maintain the *size* fields while still running in $O(\lg n)$ time. Ideally, a small number of changes to the data structure should suffice to maintain the additional information. For example, if we simply stored in each node its rank in the tree, the OS-SELECT and OS-RANK procedures would run quickly, but inserting a new minimum element would cause a change to this information in every node of the tree. When we store subtree sizes instead, inserting a new element causes information to change in only $O(\lg n)$ nodes.

For step 4, we developed the operations OS-SELECT and OS-RANK. After all, the need for new operations is why we bother to augment a data structure in the first place. Occasionally, rather than developing new operations, we use the additional information to expedite existing ones.

16.6 Interval trees

For an interval $i = [l, t]$, call l the low end and t the high end of i.

The trichotomy of intervals

For every pair of intervals i and j, exactly one of the following conditions holds :

1. i and j overlap
2. high[i] < low[j], *i.e.*, j is to the right of i
3. high[j] < low[i], *i.e.*, j is to the left of i

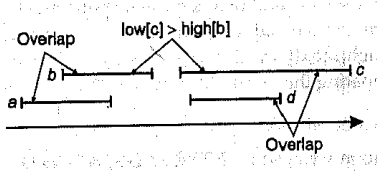


Figure 16.3

An *interval tree* is a red-black tree that maintains a dynamic set of elements, with each element x containing an interval *int*[x]. Interval trees support the following operations.

- INTERVAL-INSERT(T, x) adds the element x, whose *int* field is assumed to contain an interval, to the interval tree T.
- INTERVAL-DELETE(T, x) removes the element x from the interval tree T.
- INTERVAL-SEARCH(T, i) returns a pointer to an element x in the interval tree T such that *int*[x] overlaps interval i, or NIL if no such element is in the set.

We know that insertion into a red-black tree consists of *two* phases. The first phase goes down the tree from the root, inserting the new node as a child of an existing node. The second phase goes up the tree, changing colors and ultimately performing rotations to maintain the red-black properties.

To maintain the subtree sizes in the first phase, we simply increment $size[x]$ for each node x on the path traversed from the root down toward the leaves. The new node added gets a *size* of **1.** Since there are $O(\lg n)$ nodes on the traversed path, the additional cost of maintaining the *size* fields is $O(\lg n)$.

In the second phase, the only structural changes to the underlying red-black tree are caused by rotations, of which there are at most two. Moreover, a rotation is a local operation: it invalidates only the two *size* fields in the nodes incident on the link around which the rotation is performed. Referring to the code for LEFT-ROTATE (T, x) in, we add the following lines :

```
13 size[y] ← size[x]
14 size[x] ← size[left[x]] + size[right[x]] +1
```

Thus, the size has to be changed for only one node :

- the left-child of the rotated node in the case of right rotation and
- the right-child of the rotated node in the case of left rotation.

Since at most two rotations are performed during insertion into a red-black tree, only $O(1)$ additional time is spent updating *size* fields in the second phase. Thus, the total time for insertion into an n-node order-statistic tree is $O(\lg n)$-asymptotically the same as for an ordinary red-black tree.

Deletion from a red-black tree also consists of two phases: the first operates on the underlying search tree, and the second causes at most three rotations and otherwise performs no structural changes. The first phase splices out one node y. To update the sub tree sizes, we simply traverse a path from node y up to the root, decrementing the *size* field of each node on the path. Since this path has length $O(\lg n)$ in an n-node red-black tree, the additional time spent maintaining *size* fields in the first phase is $O(\lg n)$. The $O(1)$ rotations in the second phase of deletion can be handled in the same manner as for insertion.

Thus, both insertion and deletion, including the maintenance of the *size* fields, take $O(\lg n)$ time for an n-node order-statistic tree.

16.5 An Augmentation Strategy

16.5.1 How to augment a data structure

Augmenting a data structure can be broken into *four* steps :

1. choosing an **underlying data structure,**
2. determining **what kind of additional information** should be maintained in the underlying data structure,
3. **verify that the additional information can be maintained** during the execution of each basic modifying operation of the underlying data structure, and
4. developing **new operations.**

We followed these steps in to design our order-statistic trees. For step 1, we chose red-black trees as the underlying data structure. A clue to the suitability of red-black trees comes from their efficient support of other dynamic-set operations on a total order, such as MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR.

For step 2, we provided the *size* fields, which in each node x stores the size of the subtree rooted at x. Generally, the additional information makes operations more efficient. For example, we could have implemented OS-SELECT and OS-RANK using just the keys stored in the tree, but they would not have run in $O(\lg n)$ time.

For step 3, we ensured that insertion and deletion could maintain the *size* fields while still running in $O(\lg n)$ time. Ideally, a small number of changes to the data structure should suffice to maintain the additional information. For example, if we simply stored in each node its rank in the tree, the OS-SELECT and OS-RANK procedures would run quickly, but inserting a new minimum element would cause a change to this information in every node of the tree. When we store subtree sizes instead, inserting a new element causes information to change in only $O(\lg n)$ nodes.

For step 4, we developed the operations OS-SELECT and OS-RANK. After all, the need for new operations is why we bother to augment a data structure in the first place. Occasionally, rather than developing new operations, we use the additional information to expedite existing ones.

16.6 Interval trees

For an interval $i = [l, t]$, call l the **low end** and t the **high end** of i.

The **trichotomy** of intervals

For every pair of intervals i and j, exactly one of the following conditions holds :

1. i and j overlap
2. high[i] < low[j], *i.e.*, j is to the right of i
3. high[j] < low[i], *i.e.*, j is to the left of i

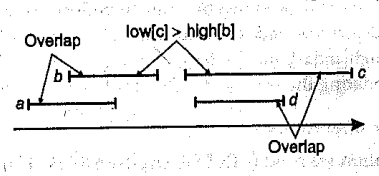


Figure 16.3

An ***interval tree*** is a red-black tree that maintains a dynamic set of elements, with each element x containing an interval $int[x]$. Interval trees support the following operations.

- **INTERVAL-INSERT**(T, x) adds the element x, whose *int* field is assumed to contain an interval, to the interval tree T.
- **INTERVAL-DELETE**(T, x) removes the element x from the interval tree T.
- **INTERVAL-SEARCH**(T, i) returns a pointer to an element x in the interval tree T such that $int[x]$ overlaps interval i, or NIL if no such element is in the set.

Step 1 Underlying data structure

We choose a red-black tree in which each node x contains an interval $int[x]$ and the key of x is the low endpoint, $low[int[x]]$, of the interval. Thus, an inorder tree walk of the data structure lists the intervals in sorted order by low endpoint.

Step 2 Additional information

In addition to the intervals themselves, each node x contains a value $m = \max[x]$, which is the maximum value of any interval endpoint stored in the sub tree rooted at x. Since any interval's high endpoint is at least as large as its low endpoint, $max[x]$ is the maximum value of all right endpoints in the sub tree rooted at x.

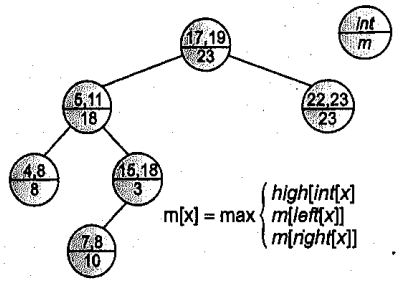


Figure 16.4

Step 3 Maintaining the information

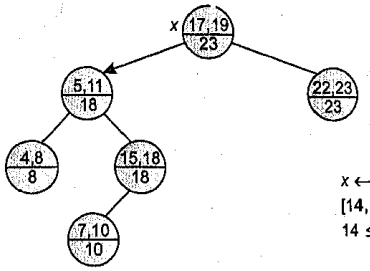
We must verify that insertion and deletion can be performed in $O(\lg n)$ time on an interval tree of n nodes. We can determine $max[x]$ given interval $int[x]$ and the max values of node x's children : max[x] = max(high[int[x]], max[left[x]], max[right[x]]).Thus, insertion and deletion run in $O(\lg n)$ time. In fact, updating the max fields after a rotation can be accomplished in $O(1)$ time

Step 4 Developing new operations

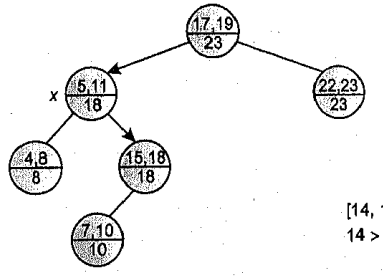
The only new operation we need is **INTERVAL-SEARCH (T, i)**, which finds an interval in tree T that overlaps interval i. If there is no interval that overlaps i in the tree, NIL is returned.

```
INTERVAL-SEARCH (i)
1.  x ← root[T]
2.  while x ≠ NIL and i does not overlap int[x]
3.      do if left[x] ≠ NIL and max[left[x]] ≥ low[i]
4.      then x ← left[x]
5.      else x ← right[x]
6.  return x
```

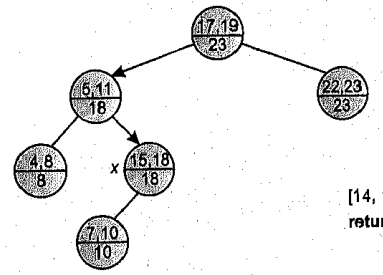

Example: INTERVAL-SEARCH ([14, 16])



$x \leftarrow root$
[14, 16] and [17, 19] don't overlap
$14 \le 18 \Rightarrow x \leftarrow left[x]$



[14, 16] and [5, 11] don't overlap
$14 > 18 \Rightarrow x \leftarrow right[x]$



[14, 16] and [15, 18] overlap
return [15, 18]

Exercise

1. Augment the following data structures.

   (a) **Link list**, where now each node maintains the number of preceding nodes.

(*b*) **Binary tree**, where now each node maintains, its own level.

(*c*) **Directed graph**, where now each node maintains its degree (in-degree and out-degree)

(*d*) **Undirected graph**, where now each node maintains its degree.

2. Show how to use an ordered-statistic tree to count the number of inversions in an array of size n in time $O(n \lg n)$.

3. Write a non-recursive version of OS-SELECT.

4. Write pseudo code for LEFT-ROTATE and RIGHT-ROTATE that operations on node in an interval tree.

5. Show how the depths of nodes in a red-black tree be efficiently maintained as fields in the nodes of the tree ?

6. Suppose each element 'e' in a red-black tree has some positive value $v[e]$ stored in it. Can we augment the nodes in the red-black tree so that each node maintains the product of values in its subtree without affecting the asymptotic running times of any standard tree operations ?

7. What are the basic steps in augmenting ? Why is augmenting of a data structure done ? Show how the dynamic set queries MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR can each be supported in $O(1)$ worst case time on an augmented order statistic tree.

CHAPTER

17

B-trees

17.1 Introduction

B-tree is a tree data structure that keeps data sorted and allows insertions and deletions in logarithmic amortized time. *B*-trees are balanced search trees designed to work well on magnetic disks or other direct-access secondary storage devices. It is most commonly used in databases and file systems. The B-tree's creators, **Rudolf Bayer** and **Ed McCreight**, have not explained what, if anything, the *B* stands for. The most common belief is that *B* stands for ***balanced***, as all the leaf nodes are at he same level in the tree.

In *B*-trees, internal nodes can have a variable number of child nodes within some pre-defined range. When data is inserted or removed from a node, its number of child nodes changes. In order to maintain the pre-defined range, internal nodes may be joined or split. Because a range of child nodes is permitted, *B*-trees do not need re-balancing as frequently as other self-balancing search trees, but may waste some space, since nodes are not entirely full. The lower and upper bounds on the number of child nodes are typically fixed for a particular implementation. For example, in a 2-3 *B*-tree (often simply 2-3 tree), each internal node may have only 2 or 3 child nodes.

A *B*-tree is kept balanced by requiring that all leaf nodes are at the same depth. This depth will increase slowly as elements are added to the tree, but an increase in the overall depth is infrequent, and results in all leaf nodes being one more hop further removed from the root.

Difference from Red Black Tree

B-trees differ form red-black trees in that *B*-tree nodes may have many children, from a handful to thousands. That is, the "branching factor" of a *B*-tree can be quite large, although it is usually determined by characteristics of the disk unit used. *B*-trees are similar to red-black trees in that every n-node *B*-tree has height $O(\lg n)$, although the height of a *B*-tree can be considerably less than that of a red-black tree because its branching factor can be much larger, Therefore, *B*-trees can also be used to implement many dynamic-set operations in time $O(\lg n)$. *B*-trees are similar to red-black trees but they are better at minimizing disk *I*/*O* operations. Many database systems use *B*-trees, to store information.

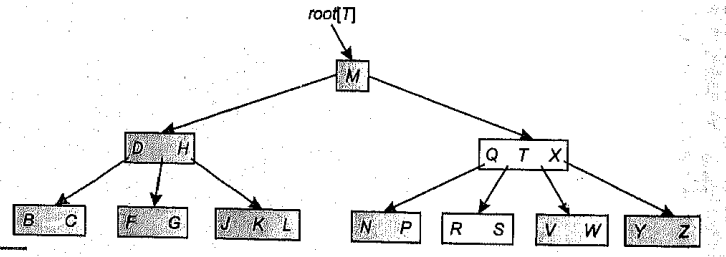


Figure 17.1

B-trees generalize binary search trees in a natural manner. If an internal *B*-tree node x contains $n[x]$ keys, then x has $n[x]+1$ children. The keys in node x are used as dividing points separating the range of keys handled by x into $n[x]+1$ sub ranges, each handled by one child of x. When searching for a key in a *B*-tree, we make an $(n[x]+1)$-way decision based on comparisons with the $n[x]$ keys stored at node x. Then structure of leaf nodes differs from that of internal nodes.

In a typical *B*-tree application, the amount of data handled is so large that all the data do not fit into main memory at once. The *B*-tree algorithms copy selected from the disk into main memory as needed and writes back onto disk, the pages that have changed. *B*-tree is designed so that only a constant number of pages are in main memory at any time; thus, the size of main memory does not limit the size of *B*-trees that can be handled.

Let x be a pointer to an object. If the object is currently in the computer's main memory, then we can refer to the fields of the object as usual ; *key*[x], for example, If the object referred to by x resides on disk, however, then we must perform the operation. DISK-READ (x) to read object x into main memory before we can refer to its fields. (We assume that if x is already in main memory, then DISK-READ (x) requires no disk accesses ; it is a "no-op.") Similarly, the operation DISK-WRITE (x) is used to save any changes that have been made to the fields of object x. That is, the typical pattern for working with an object is as follows :

```
x ← a pointer to some object
DISK-READ(x)
operations that access and/or modify the fields of x;
DISK-WRITE(x)                    // omitted if no fields of x were changed.
other operations that access but do not modify fields of x.
```


Since in most systems the running time of a *B*-tree algorithm is determined mainly by the number of DISK-READ and DISK-WRITE operations it performs, it is sensible to use these operations efficiently by having them read or write as much information as possible. Thus, a *B*-tree node is usually as large as a whole disk page. **The number of children a *B*-tree node can have is therefore limited by the size of a disk page.**

For a large *B*-tree stored on a disk, branching factors between 50 and 2000 are often used, depending on the size of a key relative to the size of a page. A large branching factor dramatically reduces both the height of the tree and the number of disk accesses required to find any key.

17.2 Definition of *B*-tree

A *B*-tree is a rooted tree (whose root is root [T]) having the following properties :

1. Every node x has the following fields :
   (*a*) $n[x]$, the number of keys currently stored in node x
   (*b*) the $n[x]$ keys themselves, stored in non-decreasing order, so that
   $$key_1[x] \le key_2[x] \le \ldots \le key_{n[x]}[x]$$
   (*c*) leaf[x], a boolean value is TRUE if x is a leaf and FALSE if x is an internal node.
2. Each internal node x also contains $n[x]+1$ pointers $c_1[x], c_2[x], \ldots c_{n[x]+1}[x]$ to its children. Leaf nodes have no children, so their c_i fields are undefined.
3. The keys $key_i[x]$ separate the range of keys stored in each subtree : if k_i is any key stored in the subtree with root $c_i[x]$, then
   $$k_1 \le key_1[x] \le k_2 \le key_2[x] \le \ldots \le key_{n[x]}[x] \le key_{n[x]+1}$$
4. All leaves have the same depth, which is the tree's height h.
5. There are lower and upper bounds on the number of keys a node can contain. These bounds are expressed in terms of a fixed integer $t \ge 2$ called the minimum degree of the *B*-tree :
   (*a*) Every node other than the root must have at least $t-1$ keys. Every internal node other than the root has at least t children. If the tree is nonempty, the root must have at least one key.
   (*b*) Every node can contain at most $2t-1$ keys. Therefore, an internal node can have at most $2t$ children. We say that a node is full if it contains exactly $2t-1$ keys.

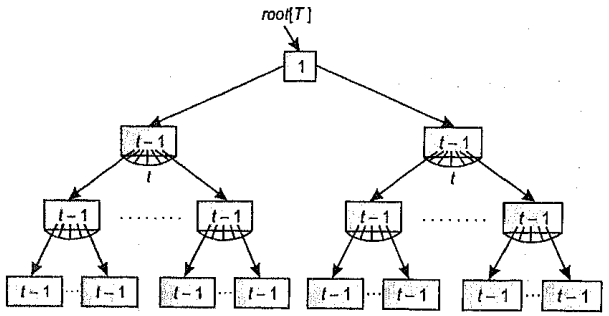
The simplest *B*-tree occurs when $t=2$. Every internal node then has either 2, 3, or 4 children, and we have a 2-3-4 tree. In practice, however, much larger values of t are used, typically adjusted to block size of hard disks for efficient *I*/*O*.

Theorem

If $n \ge 1$, then for any n-keys B-tree T of height h and minimum degree $t \ge 2$, then

$$h \le \log_t \frac{n+1}{2}$$

Proof. The root contains atleast one key. All other nodes contain atleast $t-1$ keys. There are atleast 2 nodes at depth 1, at least $2t$ nodes at depth 2, at least $2t^{i-1}$ nodes at depth i and $2t^{h-1}$ nodes at depth h.



Depth	Number of nodes
0	1
1	2
2	$2t$
3	$2t^2$

Figure 17.2 Figure shows the B-tree of height = 3

So we write

$$n \geq 1+(t-1)(2+2t+2t^2+2t^3+..+2t^{h-1})$$

$$=1+2(t-1)(1+t+t^2+t^3...+t^{h-1})$$

$$=1+2(t-1)\left(\frac{(t^h-1)}{(t-1)}\right)$$

$$=2t^h-1$$

we get,

$$t^h \leq (n+1)/2$$

Taking base-t logarithms of both sides, we get $h \leq \log_t (n+1)/2$.

17.3 Searching for a Key k in a B-tree

1. If x = nil, then k does not exist.
2. Compute the smallest i such that the ith key at x is greater than or equal to k.
3. If the ith key is equal to k, then the search is done.
4. Otherwise, set x to the ith child.

```
B-TREE-SEARCH (x, k)
1.  i ← 1
2.  while i ≤ n[x] and k > key_i[x]
3.      do i ← i + 1
4.  if i ≤ n[x] and k = key_i[x]
```



```
5.      then return (x, i)
6.  if leaf[x]
7.      then return NIL
8.      else DISK-READ (c_i[x])
9.           return B-TREE-SEARCH (c_i[x], k)
```

In B-TREE-SEARCH procedure, the nodes encountered during the recursion from a path down- ward from the root of the tree. The number of disk pages accessed by B-TREE-SEARCH is therefore $\theta(h)=\theta(\log_t n)$, where h is height of the tree and n is the number of keys in the tree. Since $n[x]<2t$, time taken by the while loop of lines 2-3 within each node is $O(t)$ and the total CPU time is $O(\text{th}) = O(t \log_t n)$

17.4 Creating an Empty B-Tree

To create an empty B-Tree a space for x is created and x is made the leaf node and then it is being made the root of the tree. ALLOCATE-NODE procedure allocates one disk page to be used as a new node in $O(1)$ time. We assume that it requires no DISK-READ, because there is no useful information stored on the disk for that node.

```
B-TREE-CREATE(T)
1.  x ← ALLOCATE-NODE( )
2.  leaf[x] ← TRUE
3.  n[x] ← 0
4.  DISK-WRITE(x)
5.  root[T] ← x
```

7.5 How do we search for a predecessor ?

```
B-TREE-PREDECESSOR (T, x, i)
1.  ▷ Find a pred. of key_i[x] in T
2.  if i ≥ 2 then
3.      if c_i[x] = nil then return key_{i-1}[x]
4.  ▷ if i ≥ 2 and x is a leaf
5.  ▷ return the (i − 1)st key
6.      else {
7.              ▷ if i ≥ 2 and x is not a leaf
8.              ▷ find the rightmost key in the i-th child
9.              y ← c_i[x]
10.             repeat
11.                 z ← c_{n[y]+1}
12.                 if z ≠ nil then y ← z
13.             until z = nil
14.             return key_{n[y]}[y]
15.         }
```

```
16.     else {
17.             ▷ Find y and j ≥ 1 such that
18.             ▷ x is the left most key in c_j[y]
19.                 while y ≠ root[T] and c_1[p[y]] = y do
20.                 y ← p[y]
21.             j ← 1
22.             while c_j[p[y]] ≠ y do j ← j + 1
23.             if j = 1 then return "No Predecessor"
24.             return key_{j-1}[p[y]]
25.         }
```

17.6 Inserting a Key into a *B*-tree

Suppose that a key k needs to be inserted in the sub tree rooted at y in a B-tree T. Before inserting the key we make sure that is room for insertion, that is, not all the nodes in the sub tree are full. Since visiting all the nodes in the sub tree is very costly, we will make sure only that y is not full. If y is a leaf, insert the key. If not, find a child in which the key should go to and then make a recursive call with y set to the child.

B-TREE-INSERT (T, k)

```
1.  r ← root[T]
2.  if n[r] = 2t − 1
3.      then s ← ALLOCATE-NODE()
4.           root[T] ← s
5.           leaf[s] ← FALSE
6.           n[s] ← 0
7.           c_1[s] ← r
8.           B-TREE-SPLIT-CHILD (s, 1, r)
9.           B-TREE-INERT-NONFULL (s, k)
10.     else B-TREE-INSERT-NONFULL (r, k)
```

In B-TREE-INSERT line 3-9 handle the case in which the root node r is full : the root is split and a new node (having two children) becomes the root. Splitting the root is the only way to increase the height of the B-tree. **Unlike a binary search tree, a B-tree increases in height at the top instead of at the bottom.** B-TREE-SPLIT-CHILD introduce an operation that splits a full node y having $2t-1$ keys around its median key $key_t[y]$ into two nodes having $t-1$ keys each. The median key moves up into y's parent to identify the dividing point between the two new trees. But if y's

parent is full, it must be split before the new key can be inserted. Fig. shows this process. It takes as input a nonfull internal node x and a node y such that $y = c_i[x]$ is a full child of x. The procedure then splits this child in two and adjusts x so that it has an additional child. The tree thus grows in height by one, splitting is the only means by which tree grows.

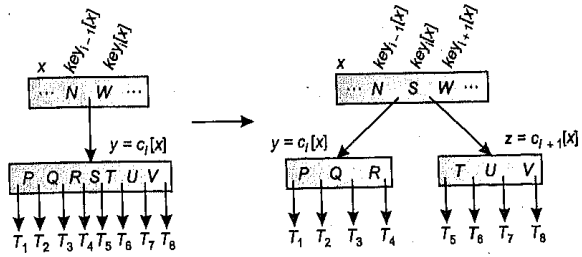


Figure 17.3

B-TREE-SPLIT-CHILD (x, i, y)

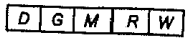
```
1.  z ← ALLOCATE-NODE()
2.  leaf[z] ← leaf[y]
3.  n[z] ← t − 1
4.  for j ← 1 to t − 1
5.      do key_j[z] ← key_{j+t}[y]
6.  if not leaf[y]
7.      then for j ← 1 to t
8.           do c_j[z] ← c_{j+t}[y]
9.  n[y] ← t − 1
10. for j ← n[x] + 1 down to i + 1
11.     do c_{j+1}[x] ← c_j[x]
12. c_{i+1}[x] ← z
13. for j ← n[x] down to i
14.     do key_{j+1}[x] ← key_j[x]
15. key_i[x] ← key_t[y]
16. n[x] ← n[x] + 1
17. DISK-WRITE(y)
18. DISK-WRITE(z)
19. DISK-WRITE(x)
```

The insertion procedure finishes by calling B-TREE-INSERT-NONFULL to perform the insertion of the key in the tree rooted at the nonfull root node.

B-TREE-INSERT-NONFULL (x, k)

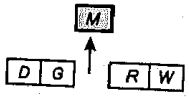
```
1.  i ← n[x]
2.  if leaf[x]
3.      then while i ≥ 1 and k < key_i[x]
4.              do key_{i+1}[x] ← key_i[x]
5.                 i ← i − 1
6.          key_{i+1}[x] ← k
7.          n[x] ← n[x] + 1
8.          DISK-WRITE(x)
9.      else while i ≥ 1 and k < key_i[x]
10.     do i ← i − 1
11.         i ← i + 1
12.         DISK-READ (c_i[x])
13.         if n[c_i[x]] = 2t − 1
14.         then B-TREE-SPILIT-CHILD (x, i, c_i[x])
15.             if k > key_i[x]
16.             then i ← i + 1
17.     B-TREE-INSERT-NONFULL (c_i[x], k)
```

Example : *After inserting D, G, M, R, and W into a B-Tree with minimum degree 3, 2 to 5 values per node :*

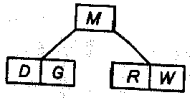


$t = 3$

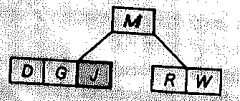
◂ To insert '*J*' : Node is full thus before insert node *j*.



The root node must be split. Move the middle value up and create two children.

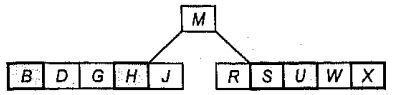


Set the child pointers.

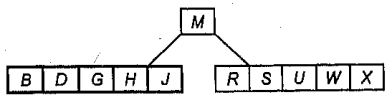


Place '*J*' in the appropriate node.

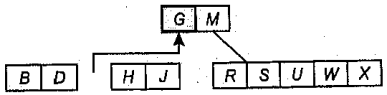
After inserting *B, H, S, U,* and *X* :



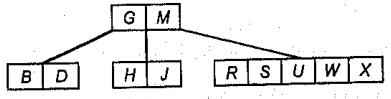
◂ To Insert '*A*' :



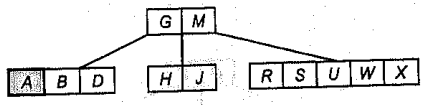
The node where '*A*' must go, is full, so it must be split before inserting node *A*.



Move the middle value, *G*, up into its parent and create 2 children.

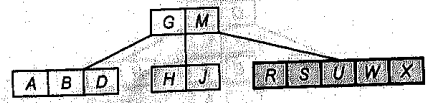


Set the children pointers.

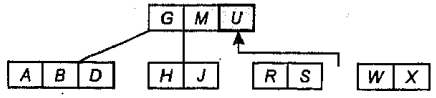


Place '*A*' in the appropriate node.

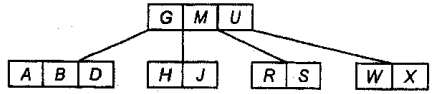
◂ To insert '*T*' :



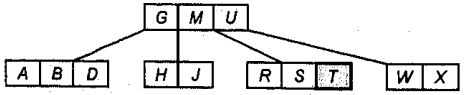
The node where 'T' goes is full, so it must be split.



Move the middle value, U, up to its parent and create two children.



Set the child pointers.



Place 'T' in the appropriate node.

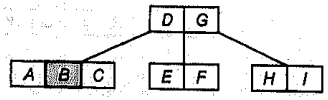
Deleting a Key from a *B*-tree

The *B*-tree delete algorithm has several different cases :

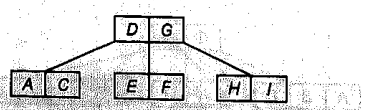
When deleting key *k* from a node *x* in a *B*-tree with $t = 3$:

Case 1. If *x* is a leaf node, then the key can just be removed.

Example : **Delete 'B'**



'B' is in a leaf node, so it can just be removed.



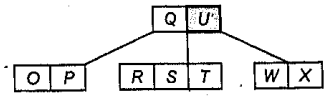
Case 2. If *x* is an internal node,

(*a*) If the key's left child has atleast *t* keys, then its largest value can be moved up to replace *k*.

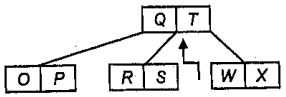
(*b*) If the key's right child has atleast *t* keys, then its smallest value can be moved up to replace *k*.

(*c*) If neither child has atleast *t* keys, then the two must be merged into one and *k* must be removed.

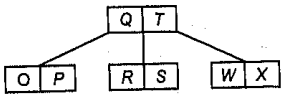
Example : **Delete 'U'**



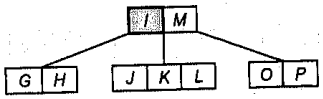
'U' is in an Internal Node.



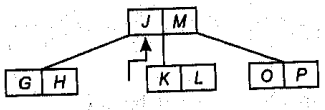
U's left child has *t* keys, so the largest value, 'T', can be moved up to replace 'U' by case 2*a*.



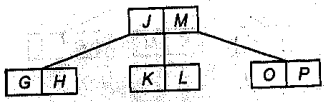
Example : **Delete 'I'**



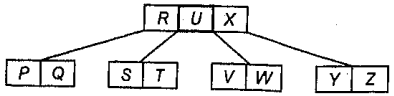
'I' is in an Internal Node.



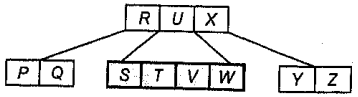
I's right child has *t* keys, so the smallest value, 'J', can be moved up to replace 'I' by case 2*b*.



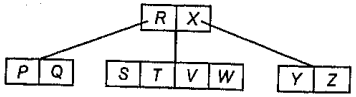
Example : **Delete 'U'**



'U's children both have $t-1$ keys.



Merge the two children into one node.



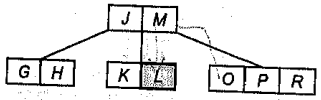
Remove 'U' and set the child pointer to the new node.

Case 3. If x has $t-1$ keys

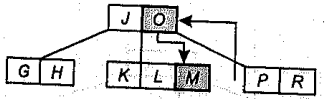
(*a*) If x has a sibling with at least t keys, move x's parent key into x, and move the appropriate extreme from x's sibling into the open slot in the parent node. Then delete the desired value.

(*b*) If x's siblings also have $t-1$ keys, merge x with one of its siblings by bringing down the parent to be the median value. Then delete the desire value.

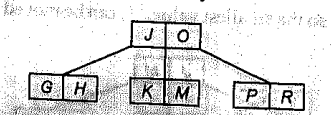
Example : **Delete 'L'**



'L' is in a node with $t-1$ keys.

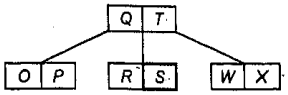


An immediate sibling has t keys, so move x parent key, 'M', into x. Move the sibling's appropriate extreme, 'O', into the parent node. All by case 3*a*.

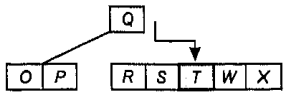


Now delete 'L' by case 1.

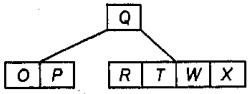
Example : **Delete 'S'**



'S' is in a node with $t-1$ keys, and its siblings also only have $t-1$ keys.

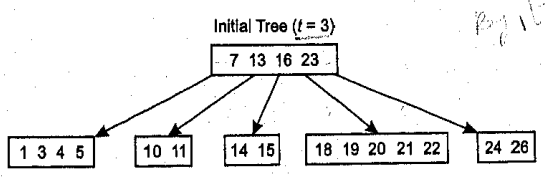


Choose one of the siblings and merge it with x by moving down the parent key to be the median for the new node, by case 3*b*

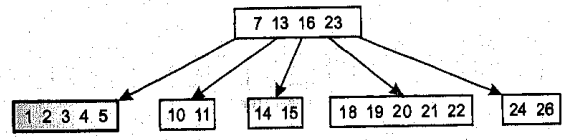


'S' can now be deleted by case 1.

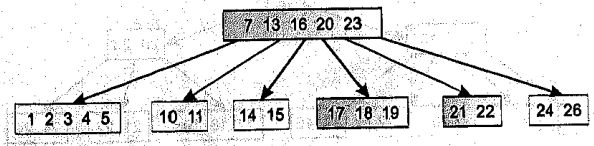
Example : *Inserting a Key in a B-tree*

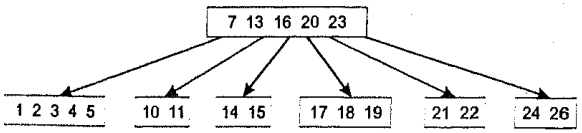


◂| Insert 2



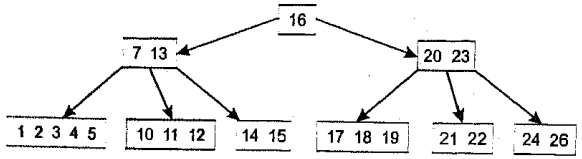
◂| Insert 17



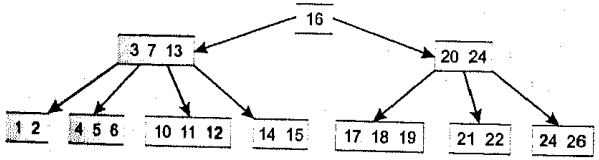


Initial tree ($t = 3$)

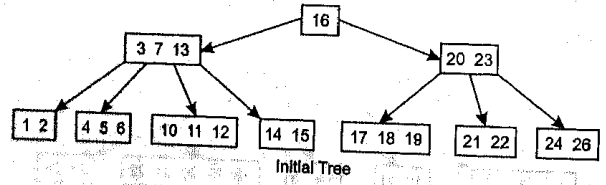
◂ Insert 12



◂ Insert 6

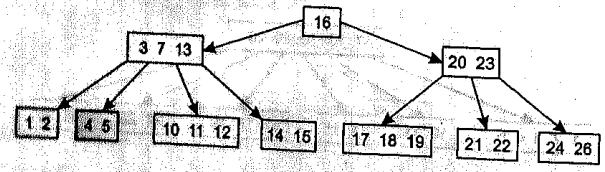


Example : *Deleting a Key in a B-tree*

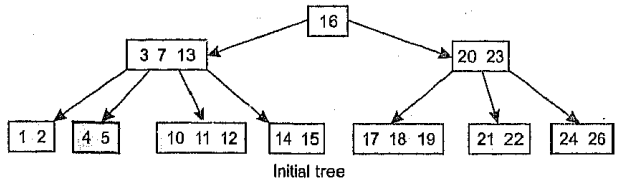


Initial Tree

◂ 6 deleted

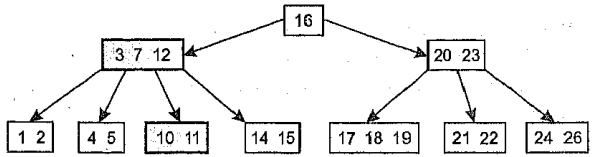


The first and simple case involves deleting the key from the leaf. $t-1$ keys remain.

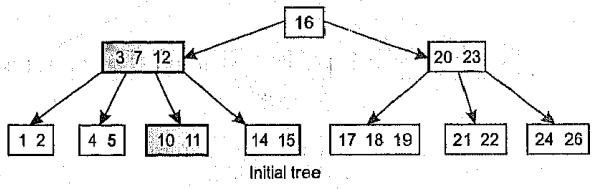


Initial tree

◂ 13 deleted

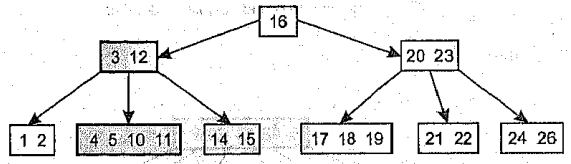


Case 2*a* is illustrated. The predecessor of 13, which lies in the preceding child of x, is moved up and takes 13*s* position. The preceding child had a key to spare in this case



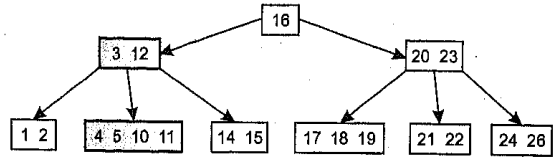
Initial tree

◂ 7 deleted



Here, both the preceding and successor children have $t-1$ keys, the minimum allowed. 7 is initially pushed down and between the children nodes to form one leaf, and is subsequently removed from the leaf.

◄| 4 deleted : Initial Tree and key 4 to be deleted in this tree.

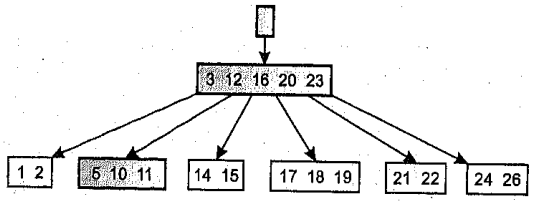


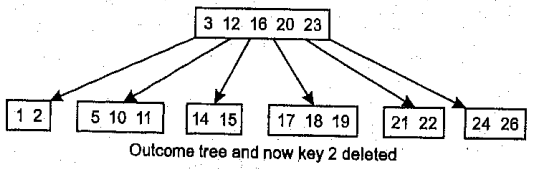
Recursion cannot descend to node 3, 12 because it has $t-1$ keys.

In case the two leaves to the left and right had more than $t-1$, 3, 12 could take one and 3 would be moved down.

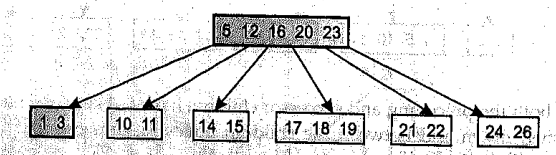
Also, the sibling of 3, 12 has also $t-1$ keys, so it is not possible to move the root to the left and take the leftmost key from the sibling to be the new root.

Therefore the root has to be pushed down merging its own children, so that 4 can be safely deleted from the leaf.





◄| 2 deleted



In this case 1, 2 has $t-1$ keys, but the sibling to the right has t. Recursion moves 5 to fill 3s position, 5 is moved to the appropriate leaf, and deleted from there.

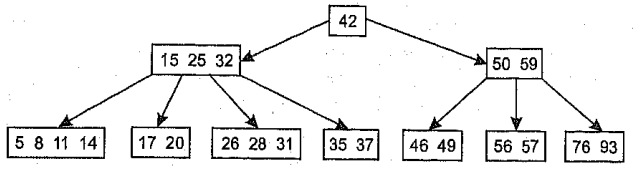
Example. *Why don't we allow a minimum degree of $t=1$.*

Solution. If we have minimum degree of t and it is not a root node, then its parent node must have $t-1$ *i.e.*, $1-1=0$ keys, but it is not possible because we cannot have a node with no keys in it so it is not possible thus $t \geq 2$.

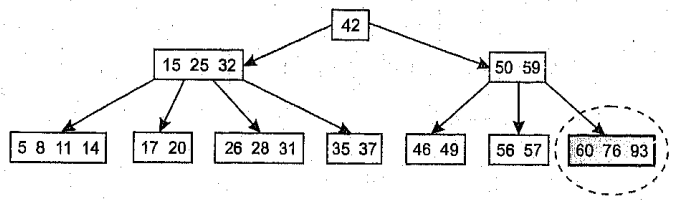
Example. *Given the following 5-way B-tree of integer data (i.e., each node can have atmost 5 children and 4 integer keys). Draw the B-Tree that results when we do each of the operations given. For each operation performed, start from the original tree i.e., these operations are not done in sequence. Circle each final answer.*

(i) Insert 60 *(ii) Remove 59*

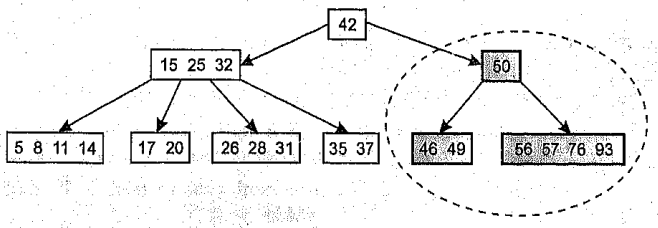
Start from B-Tree shown below



Solution. *(i)* **Insert 60.**

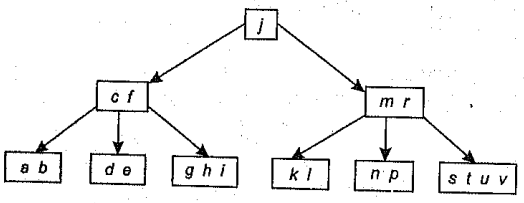


(ii) **Remove 59**



Exercise

1. Write procedures of operations :
   (*i*) *B*-TREE-SEARCH
   (*ii*) *B*-TREE-CREATE
   (*iii*) *B*-TREE-INSERT.
2. Show the results of inserting the keys *F, Q, P, K, A, L, R, M, N, X, Y, D, Z, E, H, T, V, W, C* in order to an empty *B*-Tree. Draw the configurations of the tree just before node splits. Also draw the final configurations.
3. Create *B*-Tree of order 5 from the following lists of data items.
   30, 20, 35, 95, 15, 60, 55, 25, 5, 65, 70, 10, 40, 50, 80, 45.
4. Create a *B*-Tree for the following list of elements $L = \langle 80,40,60,20,10,30,70,50,90,110 \rangle$ given minimization factor $t = 3$, minimum degree = 2 and maximum degree = 5.
5. Show all legal *B*-trees of degree 2 that represent {1, 2, 3, 4, 5}.
6. Prove that maximum number of keys in a *B*-tree is $2t - 1$ if t is the minimum degree.
7. Define In-order, Pre-order and Post-order traversing of *B*-tree.
8. Draw all *B*-trees of order 5 that can be constructed from the keys {1, 2, 3, 4, 5, 6, 7, 8}.
9. Perform the deletion operation in the following *B*-tree.



delete *b, m, n, e, c* in sequence.

CHAPTER 18

Binomial Heaps

A **binomial heap** is a data structure similar to binary heap but also supporting the operation of merging two heaps quickly. This is achieved by using a special tree structure. It is important as an implementation of the **mergeable heap** abstract data type (also called meldable heap), which is a priority queue supporting merge operation.

18.1 Mergeable Heaps

A data structure known as **mergeable heaps**, which support the following *five* operations :

- **MAKE-HEAP()** creates and returns a new heap containing no elements.
- **INSERT**(H, x) inserts node x, whose *key* field has already been filled in, into heap H.
- **MINIMUM**(H) returns a pointer to the node in heap H whose key is minimum.
- **EXTRACT-MIN**(H) deletes the node from heap H whose key is minimum, returning a pointer to the node.
- **UNION**(H_1, H_2) creates and returns a new heap that contains all the nodes of heaps H_1 and H_2. Heaps H_1 and H_2 are "destroyed" by this operation.

In addition, the data structures also support the following *two* operations :

- **DECREASE-KEY** (H, x, k) assigns to node x within heap H the new key value k, which is assumed to be no greater than its current key value.
- **DELETE** (H, x) deletes node x from heap H.

18.2 Binomial Trees and Binomial Heaps

A binomial heap is a collection of binomial trees. A *binomial tree* B_k is an ordered tree defined recursively.

1. The binomial tree B_0 consists of a single node.
2. For $k \geq 1$, the binomial tree B_k consists of two binomial trees B_{k-1} that are *linked* together: the root of one is the leftmost child of the root of the other.

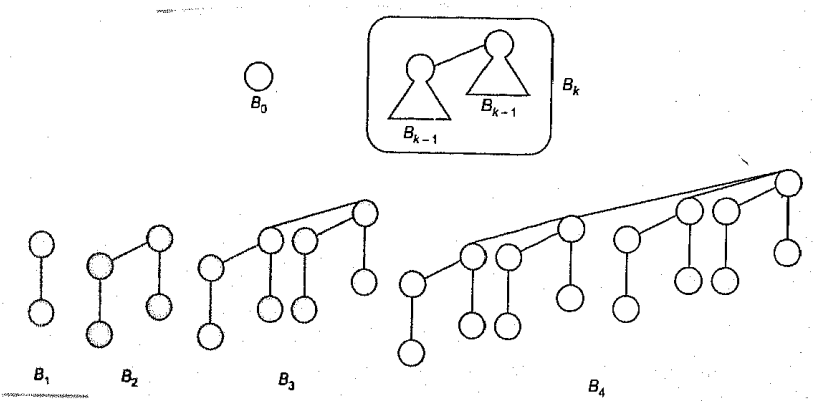


Figure 18.1

18.2.1 Properties of Binomial Trees

For the binomial tree B_k,

1. There are 2^k nodes,
2. the height of the tree is k,
3. there are exactly nodes $\binom{k}{i}$ at depth i for $i=0,1,\ldots,k$, and
4. the root has degree k, which is greater than that of any other node; moreover if the children of the root are numbered from left to right by $k-1, k-2,\ldots,0$, *child i* is the root of a subtree B_i.

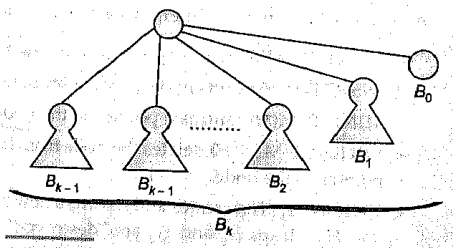


Figure 18.2

Proof : The proof is by induction on k. For each property, the basis is the binomial tree B_0. Verifying that each property holds for B_0 is trivial.

For the inductive step, we assume that this holds for B_{k-1}.

1. Binomial tree B_k consists of two copies of B_{k-1}, so B_k has $2^{k-1}+2^{k-1}=2^k$ nodes.
2. Because the two copies of B_{k-1} are linked to form B_k, the maximum depth of a node in B_k is one greater than the maximum depth in B_{k-1}. By the inductive hypothesis, this maximum depth is $(k-1)+1=k$.
3. Let $D(k, i)$ be the number of nodes at depth i of binomial tree B_k. Since B_k is composed of two copies of B_{k-1} linked together, a node at depth i in B_{k-1} appears in B_k once at depth i and once at depth $i+1$. In other words, the number of nodes at depth i in B_k is the number of nodes at depth i in B_{k-1} plus the number of nodes at depth $i-1$ in B_{k-1}. Thus,
$$D(k,i)=D(k-1,i)+D(k-1,i-1)=\binom{k-1}{i}+\binom{k-1}{i-1}=\binom{k}{i}$$
4. The only node with greater degree in B_k than in B_{k-1}, is the root, which has one more child than in B_{k-1}. Since the root of B_{k-1} has degree k-1, the root of B_k has degree k. Now by the inductive hypothesis, and as Figure 18.2 shows, from left to right, the children of the root of B_{k-1} are roots of $B_{k-2}, B_{k-3},\ldots, B_0$. When B_{k-1} is linked to B_{k-1}, therefore, the children of the resulting root are roots of $B_{k-1}, B_{k-2},\ldots, B_0$.

The term "**binomial tree**" comes from property 3 since the terms $\binom{k}{i}$ are the binomial coefficients.

i.e., ${}^kC_i$

The maximum degree of any node in an n-node binomial tree is lg n.

18.3 Binomial Heaps

A **binomial heap** H is a set of binomial trees that satisfies the following **binomial-heap properties :**

1. Each binomial tree in H is *heap-ordered:* the key of a node is greater than or equal to the key of its parent.
2. There is at most one binomial tree in H whose root has a given degree.

The first property ensures that the root of each binomial tree contains the smallest key in the tree, which applies to the entire heap. The second property implies that a binomial heap with n elements consists of at most $\lfloor \lg n \rfloor + 1$ binomial trees. In fact, the number of elements n uniquely determines the number and orders of these trees : **each binomial tree corresponds to digit one in the binary representation of number n.**

For example number 13 is 1101 in binary, $2^3+2^2+2^0$, and thus a binomial heap with 13 elements will consist of three binomial trees of orders 3, 2, and 0.

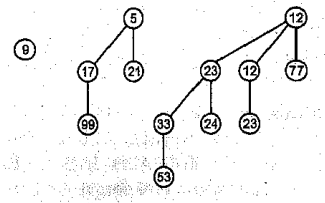


Figure 18.3

18.3.1 Representing Binomial Heaps

Each node has a *key* field and any other satellite information required by the application. In addition, each node x contains pointers $p[x]$ to its parent, *child* $[x]$ to its leftmost child, and *sibling*$[x]$ *to the sibling of* x immediately to its right. If node x is a root, then $p[x] = \text{NIL}$. If node x has no children, then *child*$[x] = \text{NIL}$, and if x is the rightmost child of its parent, then *sibling*$[x] = \text{NIL}$. Each node x also contains the field *degree*$[x]$, which is the number of children of x.

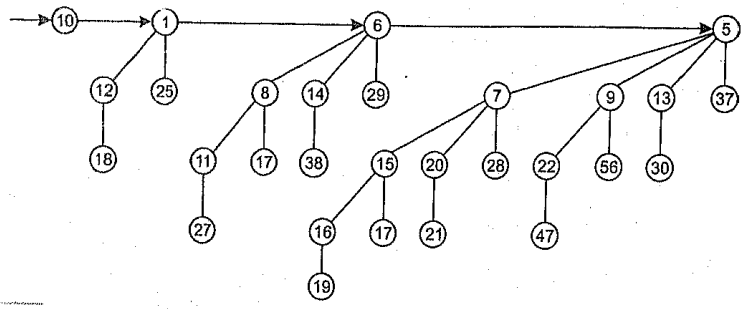


Figure 18.4

The roots of the binomial trees within a binomial heap are organized in a linked list, which we refer to as the *root list*. The degrees of the roots strictly increase as we traverse the root list. By the second binomial-heap property, in an n-node binomial heap the degrees of the roots are a subset of $\{0, 1, \ldots, \lfloor \lg n \rfloor\}$. If x is a root, then *sibling*$[x]$ points to the next root in the root list. (As usual, *sibling*$[x] = \text{NIL}$ *if* x is the last root in the root list.)

Thus, each node contains

1. a field **key** for its key,
2. a field **degree** for the number of children,
3. a pointer **child**, which points to the leftmost-child,
4. a pointer **sibling**, which points to the right-sibling, and
5. pointer **p**, which points to the parent.

A given binomial heap H is accessed by the field *head*$[H]$, which is simply a pointer to the first root in the root list of H. If binomial heap H has no elements, then *head*$[H] = \text{NIL}$.

18.3.2 Operations on Binomial Heaps

1. creation of a new heap,
2. search for the minimum key,
3. uniting two binomial heaps,
4. insertion of a node,
5. removal of the root of a tree,
6. decreasing a key, and
7. removal of a node.

18.3.2A Creating a New Binomial Heap

To make an empty binomial heap, the MAKE-BINOMIAL-HEAP procedure simply allocates and returns an object H, where *head*$[H] = \text{NIL}$. The running time is $\theta(1)$.

18.3.2B Searching the minimum key

The procedure BINOMIAL-HEAP-MINIMUM returns a pointer to the node with the minimum key in an n-node binomial heap H. This implementation assumes that there are no keys with value ∞.

```
BINOMIAL-HEAP-MINIMUM (H)
1.  y ← NIL
2.  x ← head[H]
3.  min ← ∞
4.  while x ≠ NIL
5.      do if key[x] < min
6.           then min ← key[x]
7.                y ← x
8.      x ← sibling[x]
9.  return y
```

Since a binomial heap is heap-ordered, the minimum key must reside in a root node. The BINOMIAL-HEAP-MINIMUM procedure checks all roots, which number at most $\lfloor \lg n \rfloor + 1$, saving the current minimum in *min* and a pointer to the current minimum in y. Because there are at most $\lfloor \lg n \rfloor + 1$ roots to check, the running time of BINOMIAL-HEAP-MINIMUM is $O(\lg n)$.

18.3.2C Uniting two binomial heaps

The BINOMIAL-HEAP-UNION procedure repeatedly links binomial trees whose roots have the same degree. The following procedure links the B_{k-1} tree rooted at node y to the B_{k-1} tree rooted at node z; that is, it makes z the parent of y. Node z thus becomes the root of a B_k tree.

```
BINOMIAL-LINK (y, z)
1.  p[y] ← z
2.  sibling[y] ← child[z]
3.  child[z] ← y
4.  degree[z] ← degree[z] + 1
```

The BINOMIAL-LINK procedure makes node y the new head of the linked list of node z's children in $O(1)$ time. It works because the left-child, right-sibling representation of each binomial tree matches the ordering property of the tree: in a B_k tree, the **leftmost child of the root is the root of a B_{k-1} tree.**

The following procedure unites binomial heaps H_1 and H_2, returning the resulting heap. It destroys the representations of H_1 and H_2 in the process. Besides BINOMIAL-LINK, the procedure uses an auxiliary procedure BINOMIAL-HEAP-MERGE that merges the root lists of H_1 and H_2 into a single linked list that is sorted by degree into monotonically increasing order.

BINOMIAL-HEAP-UNION($H1, H2$)

```
1.  H ← MAKE-BINOMIAL-HEAP( )
2.  head[H] ← BINOMIAL-HEAP-MERGE(H1, H2)
3.  free the objects H1 and H2 but not the lists they point to
4.  if head[H] = NIL
5.      then return H
6.  prev – x ← NIL
7.  x ← head[H]
8.  next–x ← sibling[x]
9.  while next – x ≠ NIL
10.     do if (degree[x] ≠ degree[next – x]) or
              (sibling[next – x] ≠ NIL and degree[sibling[next – x]] = degree[x])
11.         then prev – x ← x                          // Cases 1 and 2
12.              x ← next – x                          // Cases 1 and 2
13.     else if key[x] ≤ key[next – x]
14.             then sibling[x] ← sibling[next – x]    // Case 3
15.                  BINOMIAL-LINK (next – x, x)       // Case 3
16.             else if prev – x = NIL                 // Case 4
17.                  then head[H] ← next – x           // Case 4
18.                  else sibling[prev – x] ← next – x // Case 4
19.                  BINOMIAL-LINK(x, next – x)        // Case 4
20.                  x ← next – x                      // Case 4
21.     next – x ← sibling[x]
22. return H
```

The BINOMIAL-HEAP-UNION procedure has two phases. The first phase, performed by the call of BINOMIAL-HEAP-MERGE, merges the root lists of binomial heaps H_1 and H_2 into a single linked list H that is sorted by degree into monotonically increasing order. There might be as many as two roots (but no more) of each degree, however, so the second phase links roots of equal degree until at most one root remains of each degree. Because the linked list H is sorted by degree, we can perform all the link operations quickly.

Binomial-Heap-Merge ($H1, H2$)

```
1.  a ← head[H1]
2.  b ← head[H2]
3.  head [H] ← Min-Degree(a, b)
4.  if head[H] = NIL
5.      return
6.  if head[H] = b
7.      then b ← a
8.  a ← head[H]
9.  while b ≠ NIL
```



```
10.     do if sibling[a] = NIL
11.         then sibling[a] ← b
12.              return
13.         else if degree[sibling[a]] < degree[b]
14.              then a ← sibling[a]
15.              else c ← sibling[b]
16.                   sibling[b] ← sibling[a]
17.                   sibling[a] ← b
18.                   a ← sibling[a]
19.                   b ← c
```

The running time of BINOMIAL-HEAP-UNION is $O(\lg n)$, where n is the total number of nodes in binomial heaps H_1 and H_2.

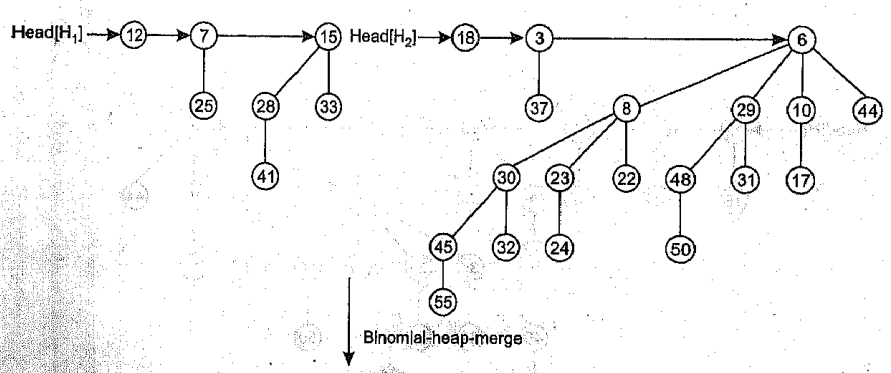
Inserting a node

The following procedure inserts node x into binomial heap H

BINOMIAL-HEAP-INSERT (H, x)

```
1.  H' ← MAKE-BINOMIAL-HEAP( )
2.  p[x] ← NIL
3.  child[x] ← NIL
4.  sibling[x] ← NIL
5.  degree[x] ← 0
6.  head[H'] ← x
7.  H ← BINOMIAL-HEAP-UNION(H,H')
```

Example of Binomial Heap-Union

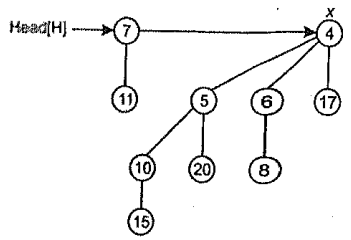


degree[x] = degree[next – x]

degree[x] ≠ degree[sibling(next – x)] and sibling(next – x) = NIL

key[x] < key(next – x)

So, case 3 apply, we get



Now, next – x = NIL, so this is the final binomial heap.

Exercise

1. How many entries does a binomial tree of rank n have ?
2. What conditions does a binomial tree satisfy ?
3. How do you join two binomial trees ? When is this a legitimate operation for binomial trees ?
4. A binomial heap has four binomial trees. Their degrees are 0, 1, 2 and 4. After you add an entry how many binomial trees will the heap have ? What are the degrees of the trees ?
5. Describe how you would use a binomial heap as an intermediate data structure in order to sort a list. Derive a tight asymptotic bound for the worst-case running time of your algorithm.
6. Can binomial tree be empty ? Why ?
7. What operations can be done efficiently on a binomial heap ? What is the advantage over a standard heap based on an almost complete binary tree ?
8. Prove that binomial tree B_k contains 2^k nodes, of which $\binom{k}{i}$ are at depth i, $0 \le i \le k$.
9. Prove that binomial heap containing n items comprises at most $\lfloor \lg n \rfloor$ binomial trees, the largest of which is $2^{\lfloor \lg n \rfloor}$ items.
10. Discuss the relationship between inserting into a binomial heap and incrementing a binary number and the relationship between uniting two binomial heaps and adding two binary numbers.

CHAPTER 19

Fibonacci Heaps

19.1 Introduction

Fibonacci heaps are linked lists of heap-ordered trees (children's keys are at least those of their parents) with the following characteristics :

1. The trees are not necessarily binomial.
2. Siblings are bi-directionally linked.
3. There is a pointer min [H] to the root with the minimum key.
4. The root degrees are not unique.
5. A special attribute $n[H]$ maintains the total number of nodes.
6. Each node has an additional Boolean label mark, indicating whether has lost a child since the last time it was made a child of another node.

Like a binomial heap, a Fibonacci heap is a collection of trees. Fibonacci heaps, in fact, are loosely based on binomial heaps. If neither DECREASE-KEY nor DELETE is ever invoked on a Fibonacci heap, each tree in the heap is like a binomial tree. **Fibonacci heaps differ from binomial heaps, however, in that they have a more relaxed structure, allowing for improved asymptotic time bounds.**

Fibonacci heaps offer a good example of a data structure designed with amortized analysis in mind.

Like binomial heaps, Fibonacci heaps are not designed to give efficient support to the operation SEARCH; operations that refer to a given node, therefore require a pointer to that node as part of their input.

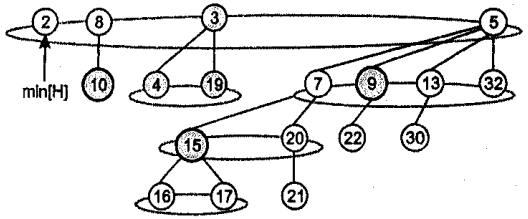


Figure 19.1

19.2 Structure of Fibonacci Heaps

Like a binomial heap, a ***Fibonacci heap*** is a collection of heap-ordered trees. The trees in a Fibonacci heap are not constrained to be binomial trees. Fig. 19.1 shows an example of a Fibonacci heap.

Unlike trees within binomial heaps, which are ordered, trees within Fibonacci heaps are rooted but unordered. Each node x contains a pointer $p[x]$ to its parent and a pointer $child[x]$ to any one of its children. The children of x are linked together in a circular, doubly linked list, which we call the *child list* of x. Each child y in a child list has pointers $left[y]$ and $right[y]$ that point to y's left and right siblings, respectively. If node y is an only child, then $left[y] = right[y] = y$. The order in which siblings appear in a child list is arbitrary.

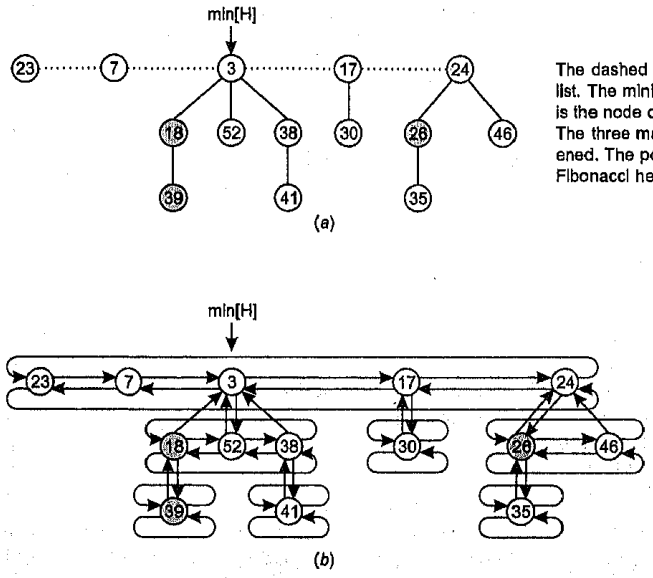
Two other fields in each node will be of use. The number of children in the child list of node x is stored in $degree[x]$. The boolean-valued field $mark[x]$ indicates whether node x has lost a child since the last time x was made the child of another node. Newly created nodes are unmarked, and a node x becomes unmarked whenever it is made the child of another node

19.2.1 Advantages of Circular, doubly linked lists use in Fibonacci heaps

Circular, doubly linked lists have two advantages for use in Fibonacci heaps. *First*, we can remove a node from a circular, doubly linked list in $O(1)$ time. *Second*, given two such lists, we can concatenate them (or "merge" them together) into one circular, doubly linked list in $O(1)$ time.

A given Fibonacci heap H is accessed by a pointer $min[H]$ to the root of the tree containing a minimum key; this node is called the **minimum node** of the Fibonacci heap. If a Fibonacci heap H is empty, then $min[H] = \text{NIL}$.

The roots of all the trees in a Fibonacci heap are linked together using their *left* and *right* pointers into a circular, doubly linked list called the **root list** of the Fibonacci heap. The pointer $min[H]$ thus points to the node in the root list whose key is minimum. The order of the trees within a root list is arbitrary. $n[H]$ is the number of nodes currently in H.



The dashed line indicates the root list. The minimum node of the heap is the node containing the key 3. The three marked nodes are blackened. The potential of this particular Fibonacci heap is 5 + 2.3 = 11.

Figure 19.2 A Fibonacci heap consisting of five heap-ordered trees and 14 nodes. (*a*) Simple representation. (b) A more complete representation showing pointers p (up arrows), child (down arrows), and left and right (sideways arrows)

19.3 Potential function

To analyze the performance of Fibonacci heap operations we use the potential method. For a given Fibonacci heap H, we indicate by $t(H)$ the number of trees in the root list of H and by $m(H)$ the number of marked nodes in H. The potential of Fibonacci heap H is then defined by

$$\Phi(H) = t(H) + 2m(H)$$

For example, the potential of the Fibonacci heap shown in Fig. 19.2 is $5 + 2.3 = 11$. We assume that a Fibonacci heap application begins with no heaps. The initial potential, therefore, is 0. The potential function of a set of fibonacci heaps is the sum of individual potential function of fibonacci heaps.

19.4 Operations

19.4.1 Creating a new Fibonacci heap

To make an empty Fibonacci heap, the MAKE-FIB-HEAP procedure allocates and returns the Fibonacci heap object H, where $n[H]=0$ and $min[H]=\text{NIL}$; there are no trees in H. Because $t(H)=0$ and $m(H)=0$, the potential of the empty Fibonacci heap is, therefore, $\Phi(H)=0$. The amortized cost of MAKE-FIB-HEAP is thus equal to its $O(1)$ actual cost.

19.4.2 Inserting a Node

The following procedure inserts node x into Fibonacci heap H, assuming that the node has already been allocated and that $key[x]$ has already been filled in.

```
FIB-HEAP-INSERT (H, x)
1.  degree[x] ← 0
2.  p[x] ← NIL
3.  child[x] ← NIL
4.  left[x] ← x
5.  right[x] ← x
6.  mark[x] ← FALSE
7.  concatenate the root list containing x with root list H
8.  if min[H] = NIL or key[x] < key[min[H]]
9.        then min[H] ← x
10. n[H] ← n[H] + 1
```

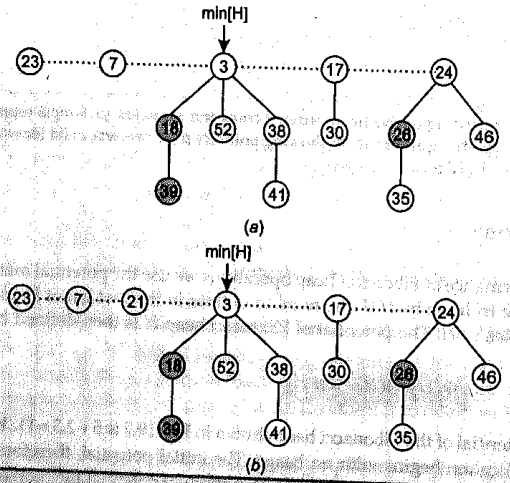


Figure 19.3 Inserting a node into a Fibonacci heap. (a) A Fibonacci heap H. (b) Fibonacci heap H after the node with key 21 has been inserted.

Unlike the BINOMIAL-HEAP-INSERT procedure, FIB-HEAP-INSERT makes no attempt to consolidate the trees within the Fibonacci heap. If k consecutive FIB-HEAP-INSERT operations occur, then k single-node trees are added to the root list.

To determine the amortized cost of FIB-HEAP-INSERT, let H be the input Fibonacci heap and H' be the resulting Fibonacci heap. Then, $t(H)=t(H)+1$ and $m(H')=m(H)$, and the increase in potential is

$$(t(H)+1)+2m(H))-(t(H)+2m(H))=1.$$

Since the actual cost is $O(1)$, the amortized cost is $O(1)+1=O(1)$.

19.4.3 Finding the Minimum Node

The minimum node of a Fibonacci heap H is always the root node given by the pointer $min[H]$, so we can find the minimum node in $O(1)$ actual time. Because the potential of H does not change, the amortized cost of this operation is equal to its $O(1)$ actual cost.

19.4.4 Uniting two Fibonacci Heaps

The following procedure unites Fibonacci heaps H_1 and H_2, destroying H_1 and H_2 in the process.

```
FIB-HEAP-UNION (H1, H2)
1.  H ← MAKE-FIB-HEAP()
2.  min[H] ← min[H1]
3.  concatenate the root list of H2 with the root list of H
4.  if (min[H1] = NIL) or (min[H2] ≠ NIL and min[H2] < min[H1])
5.        then min[H] ← min[H2]
6.  n[H] ← n[H1] + n[H2]
7.  free the objects H1 and H2
8.  return H
```

Lines 1-3 concatenate the root lists of H_1 and H_2 into a new root list H. Lines 2, 4, and 5 set the minimum node of H, and line 6 sets $n[H]$ to the total number of nodes. The Fibonacci heap objects H_1 and H_2 are freed in line 7, and line 8 returns the resulting Fibonacci heap H. As in the FIB-HEAP-INSERT procedure, no consolidation of trees occurs.

The change in potential is

$$\begin{aligned}&\Phi(H)-(\Phi(H_1)+\Phi(H_2))\\&=(t(H)+2m(H))-((t(H_1)+2m(H_1))+(t(H_2)+2m(H_2)))\\&=0\end{aligned}$$

because $t(H)=t(H_1)+t(H_2)$ and $m(H)=m(H_1)+m(H_2)$. The amortized cost of FIB-HEAP-UNION is therefore equal to its $O(1)$ actual cost.

19.4.5 Extracting the Minimum Node

The following pseudocode extracts the minimum node. The code assumes for convenience that when a node is removed from a linked list, pointers remaining in the list are updated, but pointers in the extracted node are left unchanged. It also uses the auxiliary procedure CONSOLIDATE

FIB-HEAP-EXTRACT-MIN (*H*)

```
1.  z ← min[H]
2.  if z ≠ NIL
3.      then for each child x of z
4.               do add x to the root list of H
5.               p[x] ←NIL
6.           remove z from the root list of H
7.           if z = right[z]
8.               then min[H] ← NIL
9.               else min[H] ← right[z]
10.                   CONSOLIDATE(H)
11.              n[H] ← n[H]−1
12. return z
```

FIB-HEAP-EXTRACT-MIN works by first making a root out of each of the minimum node's children and removing the minimum node from the root list. It then consolidates the root list by linking roots of equal degree until at most one root remains of each degree.

We start in line 1 by saving a pointer z to the minimum node; this pointer is returned at the end. If $z = NIL$, then Fibonacci heap H is already empty. Otherwise, as in the BINOMIAL-HEAP-EXTRACT-MIN procedure, we delete node z from H by making all of z's children roots of H in lines 3-5 (putting them into the root list) and removing z from the root list in line 6. If $z = right[z]$ after line 6, then z was the only node on the root list and it had no children, so all that remains is to make the Fibonacci heap empty in line 8 before returning z. Otherwise, we set the pointer $min[H]$ into the root list to point to a node other than z (in this case, $right[z]$).

The next step, in which we reduce the number of trees in the Fibonacci heap, is ***consolidating*** the root list of H; this is performed by the call CONSOLIDATE(H). Consolidating the root list consists of repeatedly executing the following steps until every root in the root list has a distinct *degree* value.

1. Find two roots x and y in the root list with the same degree, where $key[x] \le key[y]$.
2. ***Link*** y to x: remove y from the root list, and make y a child of x. This operation is performed by the FIB-HEAP-LINK procedure. The field $degree[x]$ is incremented, and the mark on y, if any, is cleared.

The procedure CONSOLIDATE uses an auxiliary array $A[0..D(n[H])]$; if $A[i] = y$, then y is currently a root with $degree[y] = i$.

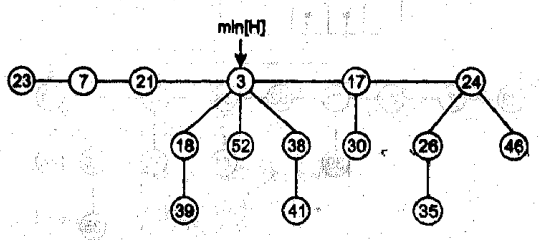
CONSOLIDATE (*H*)

```
1.  for i ← 0 to D(n[H])
2.      do A[i] ← NIL
3.  for each node w in the root list of H
4.      do x ← w
5.         d ← degree[x]
6.         while A[d] ≠ NIL
7.               do y ← A[d]
8.                  if key[x] > key[y]
9.                     then exchange x ↔ y
10.                    FIB-HEAP-LINK(H, y, x)
11.                    A[d] ← NIL
12.                    d ← d + 1
13.                A[d] ← x
14. min[H] ← NIL
15. for i ← 0 to D(n[H])
16.     do if A[i] ≠ NIL
17.        then add A[i] to the root list of H
18.             if min[H] = NIL or key[A[i]] < key[min[H]]
19.                then min[H] ← A[i]
```

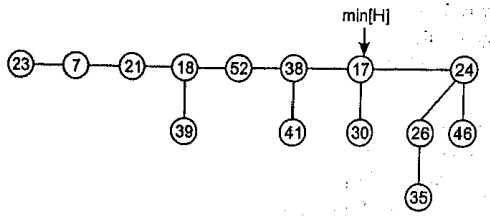
FIB-HEAP-LINK (*H*, *y*, *x*)

```
1.  remove y from the root list of H
2.  make y a child of x, incrementing degree[x]
3.  mark[y] ← FALSE
```

Example. *Show the Fibonacci heap that results from calling FIB-HEAP-EXTRACT-MIN on the Fibonacci heap shown in the figure.*



Solution. In first step the minimum node z *i.e.*, 3 is removed from the root list and its children are added to the root list.

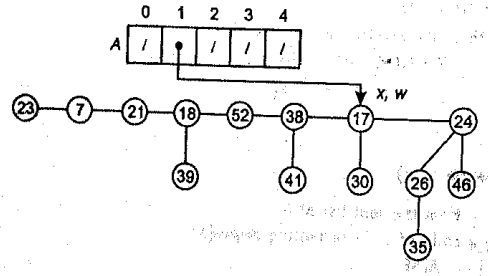


Now, min $[H] \leftarrow$ right [z]

Now, call consolidate (H), for this we use an auxiliary array $A[0..D(n[H])]$ and making each entry NIL. Then root list is processed by starting at the node pointed to by min$[H]$.

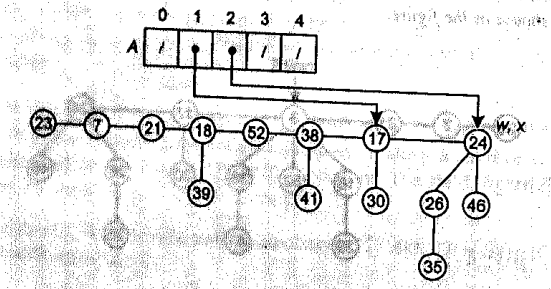
degree $[x] = 1$ $\quad \therefore \quad d = 1$

$A[d] =$ NIL

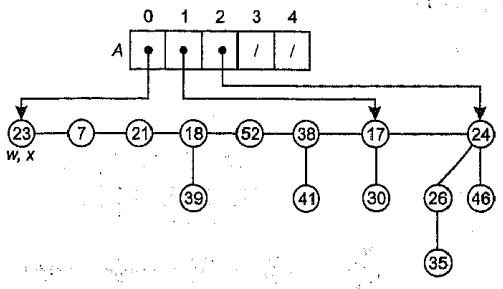


Now, w is the right node *i.e.*,

degree$[x] = 2$



degree$[x] = 0$



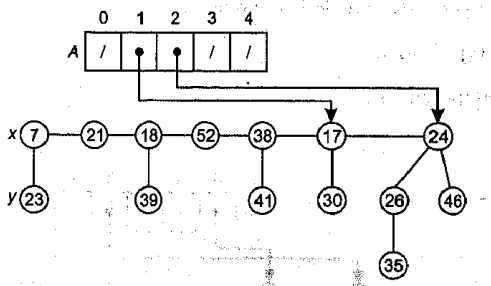
Now, w is the right node of node 23 *i.e.*, node 7.

degree$[x] = 0$

and $A[0] \neq$ NIL

then $y = A[0]$ *i.e.*, node 23.

and key$[x]$ <key$[y]$. Now call FIB-HEAP-LINK procedure *i.e.*, remove y from the root list and make y a child of x and degree$[x]$ is incremented and the mark on y, if any, is cleared *i.e.*,



But $A[1] \neq$ NIL thus $y = A[1]$

here, key$[x]$ < key$[y]$

so call FIB-HEAP-LINK and make y a child of x and degree of x is incremented

i.e., degree$[x] = 1+1 = 2$ so $d = 2$

But $A[d] \neq$ NIL, then again $y = A[d]$ *i.e.*, node 24.

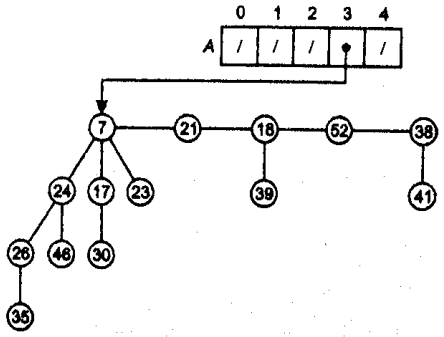
key$[x]$ < key$[y]$

Now call FIB-HEAP-LINK procedure and degree[x] is incremented *i.e.*,

now, degree[x] = 2 + 1 = 3

d = 3

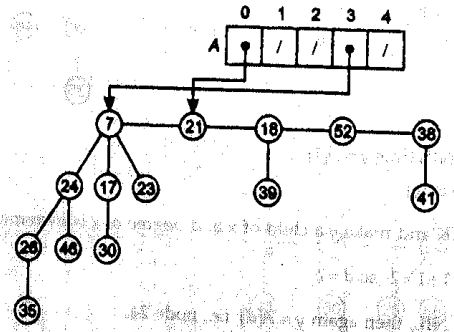
A[3] = NIL



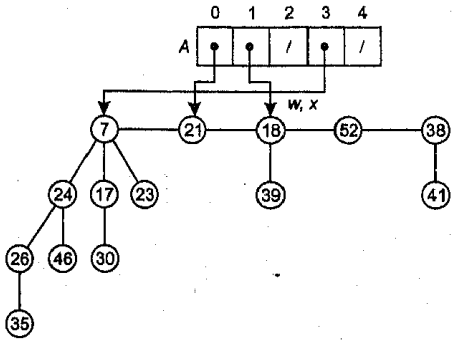
Now, w is the right node of node 7 *i.e.*, node 21.

degree [x] = 0 so d = 0

and A[0] = NIL, so



Now, w is the node 18 and degree [x] = 1 and A[1] = NIL, so



Now, w is the node 52 and node 52 has degree 0 *i.e.*, d = 0

$A[d] \neq$ NIL $y \leftarrow A[d]$

key[x] > key[y] so exchange $x \leftrightarrow y$.

Now x is node 21.

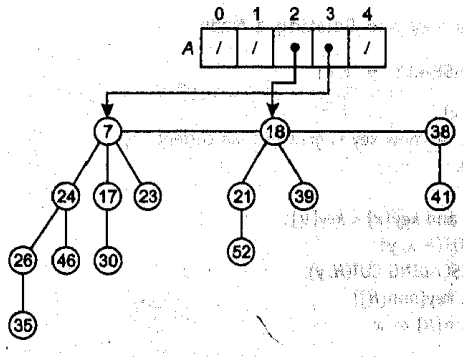
call FIB-HEAP-LINK

$A[d] \neq$ NIL again. $y \leftarrow A[d]$

key[x] > key[y] so exchange $x \leftrightarrow y$.

Now x is 18 and d = 1 + 1 = 2.

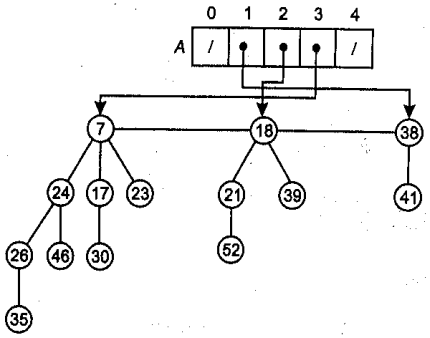
$A[d]$ = NIL



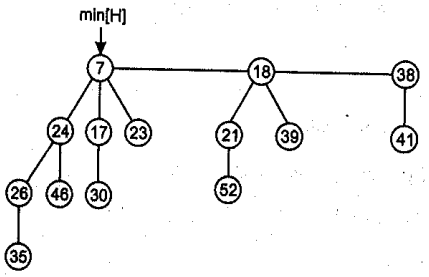
Now, w is the node 38

degree $[x] = 1$

and $A[1] = \text{NIL}$



So, final heap is



19.4.6 Decreasing a Key and Deleting a Node

FIB-HEAP-DECREASE-KEY (H, x, k)

1. if $k > key[x]$
2. then error "new key is greater than current key"
3. $key[x] \leftarrow k$
4. $y \leftarrow p[x]$
5. if $y \neq$ NIL and $key[x] < key[y]$
6. then CUT(H, x, y)
7. CASCADING-CUT(H, y)
8. if $key[x] < key[\min[H]]$
9. then $\min[H] \leftarrow x$

CUT (H, x, y)

1. remove x from the child list of y, decrementing $degree[y]$
2. add x to the root list of H
3. $p[x] \leftarrow$ NIL
4. $mark[x] \leftarrow$ FALSE

CASCADING-CUT (H, y)

1. $z \leftarrow p[y]$
2. if $z \neq$ NIL
3. then if $mark[y] =$ FALSE
4. then $mark[y] \leftarrow$ TRUE
5. else CUT(H, y, z)
6. CASCADING-CUT(H, z)

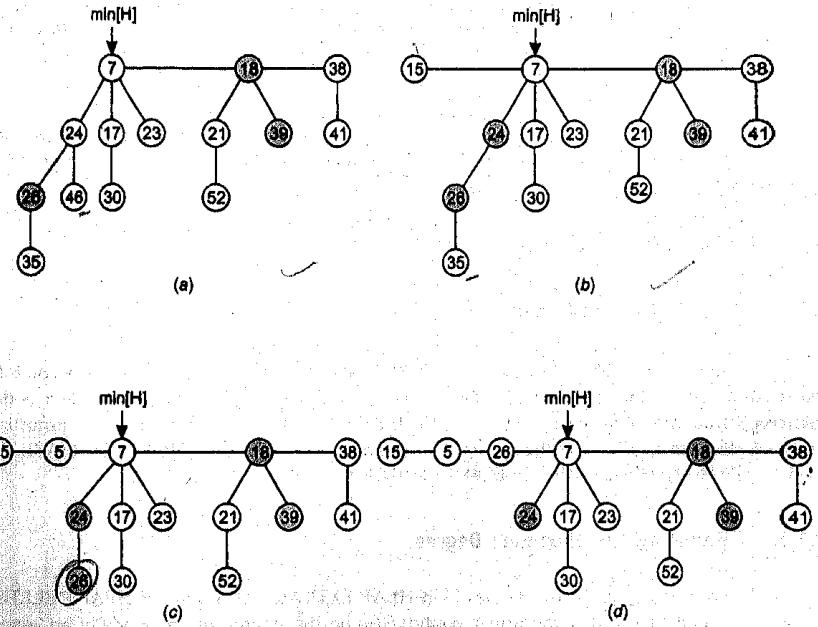


Figure 19.4

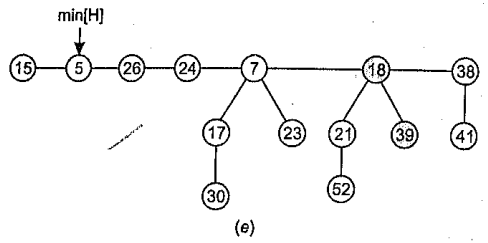


Figure. 19.4 Two calls of FIB-HEAP-DECREASE-KEY. (*a*) The initial Fibonacci heap. (*b*) The node with key 46 has its key decreased to 15. The node becomes a root, and its parent (with key 24), which had reviously been unmarked, becomes marked. (*c*)-(*e*) The node with key 35 has its key decreased to 5. In part (*c*), the node, now with key 5, becomes a root. Its parent, with key 26, is marked, so a cascading cut occurs. The node with key 26 is cut from its parent and made an unmarked root in (*d*). Another cascading cut occurs, since the node with key 24 is marked as well. This node is cut from its parent and made an unmarked root in part (*e*). The cascading cuts stop at this point, since the node with key 7 is a root. (Even if this node were not a root, the cascading cuts would stop, since it is unmarked.) The result of the FIB-HEAP-DECREASE-KEY operation is shown in part (*e*), with min[*H*] pointing to the new minimum node.

Deleting a Node

It is easy to delete a node from an n-node Fibonacci heap in $O(D(n))$ amortized time, as is done by the following pseudocode. We assume that there is no key value of $-\infty$ currently in the Fibonacci heap.

FIB-HEAP-DELETE (H, x)

1. FIB-HEAP-DECREASE-KEY($H, x, -\infty$)
2. FIB-HEAP-EXTRACT-MIN(H)

FIB-HEAP-DELETE is analogous to BINOMIAL-HEAP-DELETE. It makes x become the minimum node in the Fibonacci heap by giving it a uniquely small key of $-\infty$. Node x is then removed from the Fibonacci heap by the FIB-HEAP-EXTRACT-MIN procedure. The amortized time of FIB-HEAP-DELETE is the sum of the $O(1)$ amortized time of FIB-HEAP-DECREASE-KEY and the $O(D(n))$ amortized time of FIB-HEAP-EXTRACT-MIN.

19.5 Bounding the Maximum Degree

To prove that the amortized time of FIB-HEAP-EXTRACT-MIN and FIB-HEAP-DELETE is $O(\lg n)$, we must show that the upper bound $D(n)$ on the degree of any node of an n-node Fibonacci heap is $O(\lg n)$. The maximum degree in a fibonacci heap of n-nodes is indicated by $D(n)$.

Lemma 1

Let x be any node in a Fibonacci heap, and suppose that degree $[x]=k$. Let $y_1, y_2, \ldots, y_k$ denote the children of x in the order in which they were linked to x, from the earliest to the latest. Then, degree$[y_1] \geq 0$ and degree $[y_i] \geq i-2$ for $i=2,3,\ldots,k$.

Proof. It is obvious that for first child of x, y_1, degree $[y_1] \geq 0$.

For $i \geq 2$, we note that when y_1 was linked to x, all of $y_1, y_2, \ldots, y_{i-1}$ were children of x, so we must have had *degree*$[x] \geq i-1$. Node y_i is linked to x only if *degree*$[x]$ = *degree*$[y_i]$, so we must have also had *degree*$[y_i] \geq i-1$ at that time. Since then, node y_i has lost at most one child, otherwise it would have been cut from x if it had lost two children. Thus *degree* $[y_i] \geq i-2$.

The above lemma is proved.

Recall that the nth Fibonacci sequence is defined by the recurrence

$$F_k = \begin{cases} 0 & \text{if } k=0 \\ 1 & \text{if } k=1 \\ F_{k-1}+F_{k-2} & \text{if } k \geq 2 \end{cases}$$

The following lemma gives another way to express F_k.

Lemma 2

For all integers $k \geq 0$,

$$F_{k+2} = 1+\sum_{i=0}^{k} F_i$$

Proof. The proof is by induction on k. When $k=0$,

$$1+\sum_{i=0}^{k} F_i = 1+F_0$$

$$= 1+0 = 1 = F_2$$

We now assume the inductive hypothesis that $F_{k+1} = 1+\sum_{i=0}^{k-1} F_i$, and we have

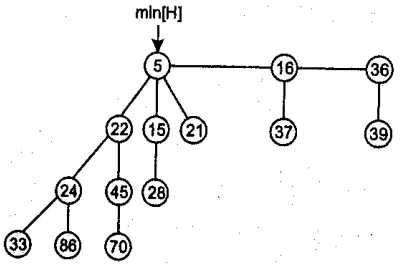
$$F_{k+2} = F_k + F_{k+1}$$

$$= F_k + \left[1+\sum_{i=0}^{k-1} F_i\right] = 1+\sum_{i=0}^{k} F_i$$

The above lemma is proved.

Exercise

1. Prove that, for a node x in a Fibonacci Heap, if c_i is the ith youngest child of x, then rank of c_i is at least $i-2$.
2. Prove that the rank of any node in a Fibonacci heap is $O(\lg n)$.
3. Suppose that a root x in a Fibonacci heap is marked. Explain how x came to be a market root. Argue that it doesn't matter to analysis that x is marked, even though it is not a root that was first linked to another node and then lost one child.
4. Show that if only the mergeable-heap operations are supported, the maximum degree $D(n)$ in an n-node Fibonacci heap is at most $\lfloor \lg n \rfloor$.
5. Suppose we generalize the cascading-cut rule to cut a node x from its parent as soon as it loses its kth child, for some integer constant k. For what values of k is $D(n) = O(\lg n)$?
6. Create a Fibonacci-heap for the following list $\langle 20,10,5,30,35,55,25,45,36,32 \rangle$.
   After creation, change the value of 45 to 23.
7. Decrease the key 45 to 34 in a Fibonacci heap shown in the Fig. Also find the resultant fibonacci heap after decreasing the key.



CHAPTER 20

Data Structures for Disjoint Sets

20.1 Introduction

An important application of the tree is the representation of sets, where "n" distinct elements are needed to be grouped into a number of disjoint sets.

A ***disjoint-set data structure*** maintains a collection $S = \{S_1, S_2, \ldots, S_k\}$ of disjoint dynamic sets. Each set is identified by a *representative*, which is some member of the set. If we have two sets S_x and S_y, $x \neq y$, such that $S_x = \langle 3,4,5,6,7 \rangle$ and $S_y = \langle 1,2 \rangle$ then these sets are called disjoint sets as there is no element which is common in both sets.

20.2 Disjoint-Set Operations

In the dynamic-set implementations, an object represents each element of a set. Letting x denote an object, we wish to support the following operations.

- **MAKE-SET(x)** creates a new set whose only member (and thus representative) is pointed to by x. Since the sets are disjoint, we require that x not already be in a set.