Syllabus for Algorithms II

1. *Lectures 1 & 2.* Introduction to algorithms. Review of what an algorithm is. Important aspects of algorithms. Expressing algorithms. Categories of algorithms. Examples of recursive algorithms.

2. *Lectures 3 & 4.* Greedy algorithms and the coin change problem. Backtracking and mazes. Divide and conquer algorithms and the lowest two problem.

3. *Lectures 5 & 6.* Branch and bound technique. Exhaustive search and tree pruning. The travelling salesman problem. Abstract data types. Arrays and lists. Linear and binary searches of a list. Singly linked lists. The storage pool.

4. *Lectures 7 & 8.* Doubly linked lists. Dynamic storage management example.

5. *Lectures 9 & 10.* Stacks and queues, with examples. Sorting. Stability. Selection sort and bubble sort.

6. *Lectures 11 & 12.* Improvements to bubble sort. Insertion sort algorithm and analysis of best, worst and average cases. Shellsort algorithm. Quicksort algorithm with analysis of best and worst cases.

7. *Lectures 13 & 14.* Non-recursive quicksort algorithm. Tree representation of quicksort. Improvements to quicksort including median-of-threes version. Mergesort algorithm.

8. *Lectures 15 & 16.* Heapsort algorithm. Search algorithms. Binary search, interpolation search. Binary search trees. Binary tree search algorithm. intersection.

9. *Lectures 17 & 18.* Insertion and deletion operations on binary search trees. Sorting binary search trees. Searching for strings. Brute force algorithm.

10. *Lectures 19 & 20.* Knuth-Morris-Pratt algorithm for string searching.

11. *Lectures 21 & 22.* Evaluation of the failure function in the KMP algorithm. Relaxed searches. Soundex codes. Review of coursework.

Assessment

Two hour College examination	80%
Two pieces of assessed coursework	20%

Useful Books

- "Fundamentals of Data Structures," Horowitz and Sahni, published by Computer Science Press, ISBN 0-914894-20X.
- "Algorithms in C," Robert Sedgewick, published by Addison Wesley, 1990. ISBN 0-201-51425-7.
- "Understanding Algorithms and Data Structures," Brunskill and Turner, published by McGraw-Hill, ISBN 0-07-709141-8, 1996.
- World-wide web site for lecture notes

  `http://www.cs.cf.ac.uk/User/David.W.Walker/ALGII/`

What is an algorithm?

An algorithm is a finite set of instructions for accomplishing a particular task.

- The word *algorithm* is named after an Iranian mathematician Al-Khawarizmi.
- Technically an algorithm must accomplish a task in a finite number of steps, but the term “algorithm” is also often used to refer to any sequence of actions, which may not necessarily terminate.
- An algorithm may have zero or more inputs.
- An algorithm must produce at least one output.

Examples of algorithms not related to computers include

- recipes for preparing food
- instructions for building something

What about instructions for getting form one place to another? Is this an algorithm?

Algorithmic Issues

Some important issues when designing and studying an algorithm are

- How does the algorithm work?
- How accurate is the algorithm?
- How robust is the algorithm? Is it sufficiently accurate for all possible inputs?
- How can the algorithm be implemented on a computer? What programming language is most suitable?
- What resources does the algorithm require?

Algorithmic analysis seeks to predict what resources an algorithm needs.

- computational time,
- memory,
- I/O,
- communication bandwidth

Expressing an algorithm

- We might use a natural language, such as English, French, etc, to describe a simple algorithm.
- Natural languages have difficulty in clearly expressing complex algorithms.
- We could just pick a language such as C, Pascal, or Fortran to express an algorithm, but this requires us to worry about the syntax of the language rather than the features of the algorithm.
- A good compromise is to express algorithms in *pseudocode*. This allows us to express algorithms clearly without being bothered with the precise details of a programming language.
- At times it may also be useful to describe an algorithm graphically using a flow chart.

Averaging Example

Suppose we want to find the average of a set of n numbers.

- The input to the algorithm is the set of numbers.
- The output is the average of the numbers

In English we can express the algorithm as follows:

> *"To find the average, add the numbers together and divide the result by n."*

Flowchart for Averaging

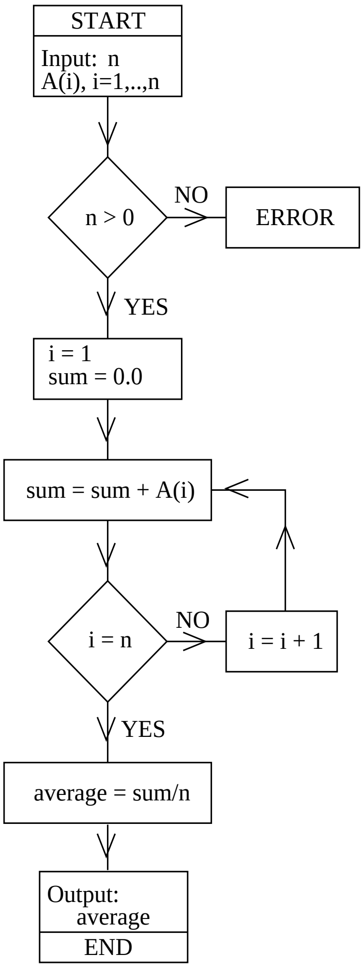


Figure 1: Flowchart for averaging example.

The flowchart in Figure 1 does not correctly represent the averaging algorithm. Can you see why?

SPARKS Pseudocode

In this course we shall use a form of pseudocode called SPARKS for describing algorithms. SPARKS is described in section 1.2 of Horowitz and Sahni.

- *Assignment*

$$\text{variable} \leftarrow \text{expression}$$

- *Expressions* may be arithmetic, boolean, or character type.
- *Boolean expressions* can only have one of two values: **true** or **false**. They may involve the logical operator **and**, **or**, and **not**. The following relational operators are also provided;

$$<, \leq, =, \neq, \geq, >$$

- There are *statements* for assignment, conditional testing, iteration, input/output, etc. Several such statements can be combined on a single line if they are separated by a semi-colon. For example,

$$i \leftarrow 1; \, sum \leftarrow 0.0$$

Conditional Statements

A conditional statement has one of the following forms;

if *cond* **then** S_1

or,

if *cond* **then**
    S_1
else
    S_2

- *cond* is a boolean expression.
- S_1 and S_2 are arbitrary groups of SPARKS statements.
- If S_1 or S_2 contains more than one statement these must be enclosed in square brackets. For example,

if $i = 0$ **then** [
    $r \leftarrow 10.0$
    $t \leftarrow r/3.0$]
else [
    $r \leftarrow 5.0$
    $t \leftarrow r/2.0$]

Conditional Flowcharts

The two forms of the SPARKS conditional statement can be represented using flowcharts as shown in Figure 2.

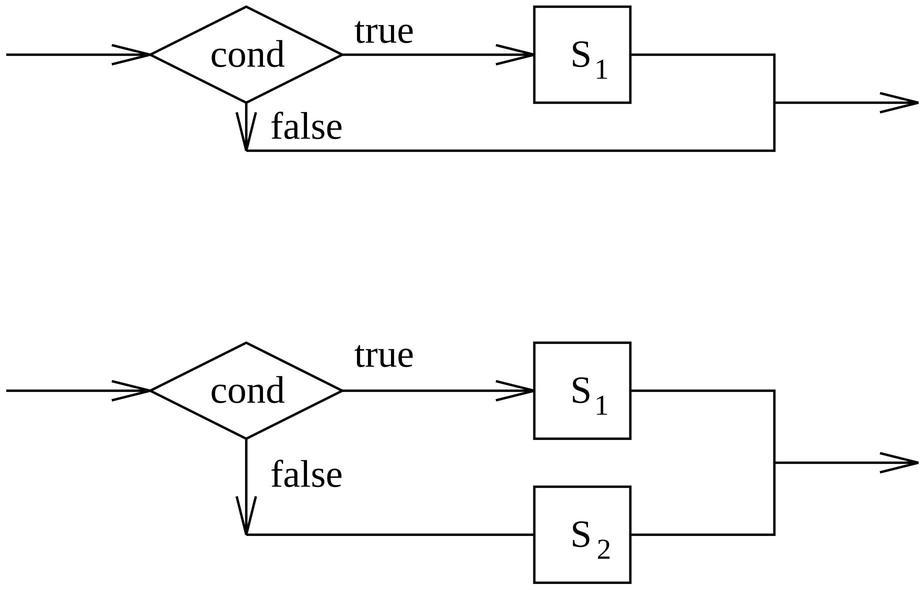


Figure 2: Flowcharts for SPARKS conditional statements.

Iteration Statements

SPARKS provides a number of ways of accomplishing iteration,

- The **while** statement

while *cond* **do**
  S
end

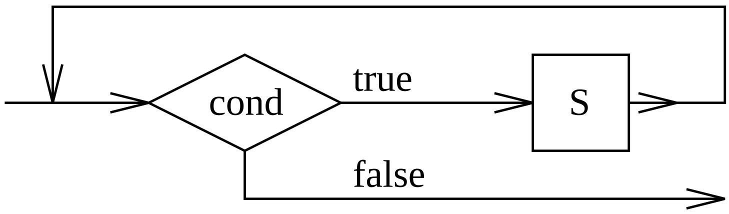


Figure 3: Flowchart for **while** statement.

- The **repeat-until** statement

repeat
  S
until *cond*

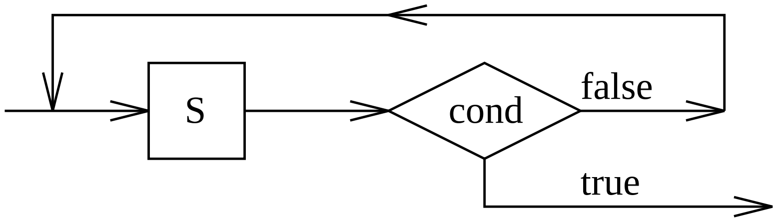


Figure 4: Flowchart for **repeat-until** statement.

- The **for** statement

for *variable* $\leftarrow$ *start* **to** *finish* **by** *increment* **do**
  S
end

The for Statement

The **for** statement

for *variable* $\leftarrow$ *start* **to** *finish* **by** *increment* **do**
    S
end

can be expressed in terms of a **while** statement as follows.

$$
\begin{array}{l}
\mathit{variable} \leftarrow \mathit{start} \\
\mathit{fin} \leftarrow \mathit{finish} \\
\mathit{incr} \leftarrow \mathit{increment} \\
\textbf{while } (\mathit{variable} - \mathit{fin}) * \mathit{incr} \leq 0 \textbf{ do} \\
\quad S \\
\quad \mathit{variable} \leftarrow \mathit{variable} + \mathit{incr} \\
\textbf{end}
\end{array}
$$

In the **for** statement, the "**by** *increment*" clause is optional, and if it is omitted *increment* is taken as 1.

The loop-forever Statement

The **loop-forever** statement is usually used in conjunction with the **exit** command, which transfers control to the first statement after the innermost loop that contains it. For example,

loop
    S_1
    if *cond* **then exit**
    S_2
forever

This would execute as follows

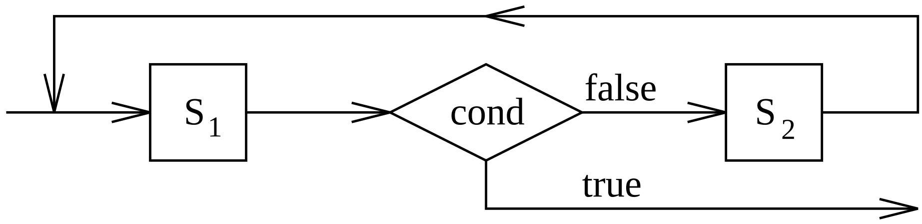


Figure 5: Example of the use of the **loop-forever** statement.

A loop may also be exited using the "**goto** *label*" statement which transfers control to the line with the specified label.

The case Statement

The **case** statement allows one to choose between several alternatives, without the need for multiple **if-then-else** statements.

case
  : *cond 1* : S_1
  : *cond 2* : S_2
  etc.
  : *cond n* : S_n
  : **else** : S_{n+1}
end

where the S_i, $1 \le i \le n+1$, are groups of SPARKS statements. The **else** clause is optional.

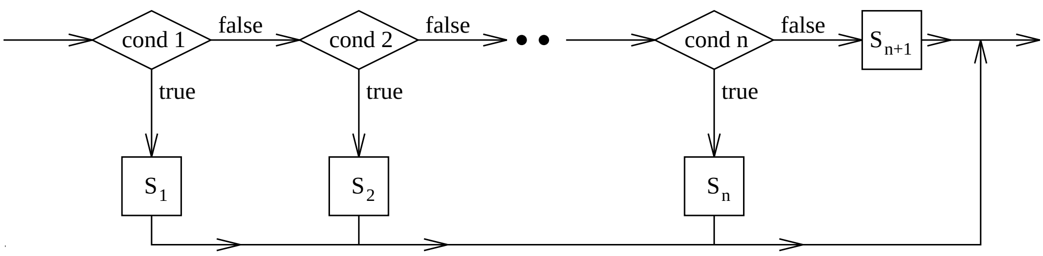


Figure 6: Flowchart for **case** statement.

Procedures

- In SPARKS, algorithms are expressed in terms of procedures. Procedures have the following structure;

procedure *NAME (parameter list)*
  S
end

- A procedure can be used as a function to return a value

return *(expr)*

- If "expr" is omitted a return is made to the calling procedure.
- A procedure is called as follows,

call *NAME (parameter list)*

- A procedure may call itself. This is referred to as *recursion*.

Further Details

- Input and output are done using **read** and **print**.

  read *(argument list)*, **print** *(argument list)*

  where the arguments may be variables or quoted strings.

- The **stop** command halts execution.

- Comments are enclosed in double slashes.

  //This is a comment //

Pseudocode for Averaging Example

Now let's return to the algorithm for averaging a set of n numbers. the SPARKS pseudocode may be written as follows.

```
procedure average (A, n, error)
   error ← 0
   if n ≤ 0 then [
      error ← −1
      return ]
   sum ← 0.0
   for i ← 1 to n do
      sum ← sum + A(i)
   end
   return (sum/n)
end
```

- The variable *error* is returned in the argument list to indicate whether the input value of n was valid.
- This procedure returns the average as a function.

Classification of Algorithms

Algorithms can be divided into a number of broad classes, such as

- Recursive algorithms
- Iterative algorithms
- Greedy algorithms
- Backtracking algorithms
- Divide-and-conquer algorithms

We shall now examine examples of these types of algorithms.

Recursive Algorithms

Consider the factorial function

$$f(n) = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1.$$

- The input is a positive integer, n.
- The output is the product of all the non-negative integers up to and including n. This is called "factorial n", and is often written as $n!$

We can write

$$f(n) = n \times f(n-1)$$

so in English we can express the algorithm as follows

> "Find $f(n)$ by multiplying $f(n-1)$ by n."

This is an example of a *recursive algorithm*. To find $f(n)$ we must first find $f(n-1)$. Recursive algorithms are often simple to express. The SPARKS pseudocode can be written as

procedure *factorial (n)*
    if $n = 1$
        return *(1)*
    else
        return *(n∗factorial(n − 1))*
end

We say that that recursion *terminates* when $n = 1$. This is handled using a conditional statement.

Fibonacci Numbers

The factorial function provides a rather trivial example of recursion. We now consider another example of recursion, namely the numbers generated by the following recurrence

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

for $n \geq 2$, with $f(0) = 0$, and $f(1) = 1$. These are known as Fibonacci numbers.

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$$

We can use SPARKS pseudocode to describe the algorithm.

procedure fibonacci (n)
    if $n = 0$ **then return** (0)
    if $n = 1$ **then return** (1)
    return (*fibonacci*$(n-1)$ + *fibonacci*$(n-2)$)
end

This is a more interesting example of a recursive function because the *fibonacci* function appears twice on the righthand side.

Recursion Tree

The computations performed in evaluating *fibonacci(n)* can be represented by a tree

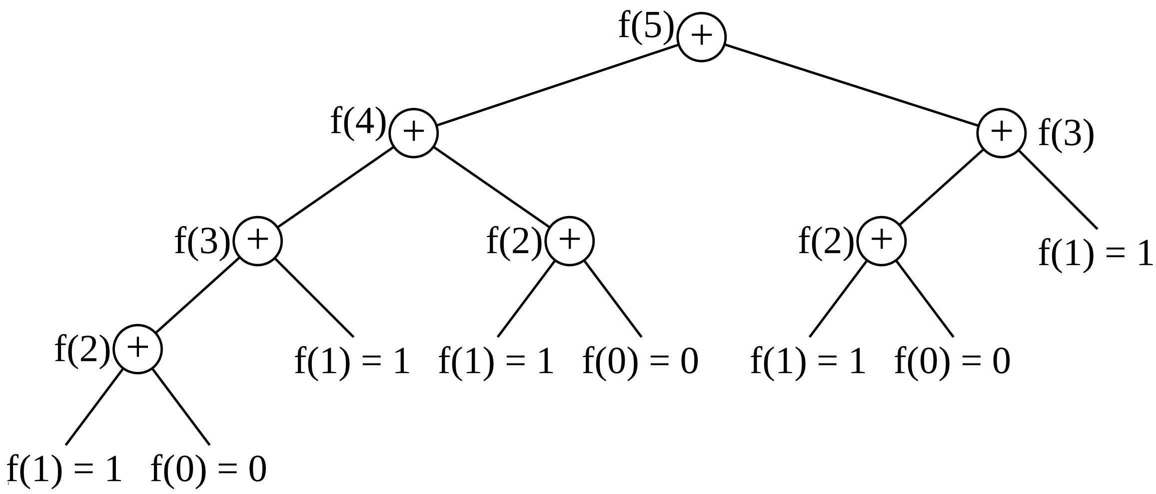


Figure 7: Representation of *fibonacci* function with a tree for $n = 5$.

We can view the algorithm from the bottom-up using this recursion tree. Values flow up the tree and get added at the nodes.

An Iterative Fibonacci Algorithm

- Use of recursion often results in a very compact algorithm.
- A recursive algorithm can also be written in an iterative form. The iterative form of the Fibonacci algorithm is as follows.

procedure fibonacci (n)
   if $n = 0$ **then return** (0)
   if $n = 1$ **then return** (1)
   $f2 \leftarrow 0;\ f1 \leftarrow 1$
   for $i \leftarrow 2$ **to** n **do**
      $f \leftarrow f1 + f2$
      $f2 \leftarrow f1$
      $f1 \leftarrow f$
   end
   return *(f)*
end

- Most compilers will produce code that runs the iterative code faster than the recursive code.
- The iterative version is easier to analyse.

Analysis of Fibonacci Algorithm

The standard way to analyse the *computational complexity* of an algorithm is to count the number of operations performed. If the number of each type of operation is known an estimate of the computing time of the algorithm can be obtained by multiplying the operation counts by the time for each operation and adding.
In the iterative version of the Fibonacci algorithm there are

- 2 conditional statements outside the **for** loop
- 2 assignment statements outside the **for** loop
- 1 addition in each pass through the loop
- 1 **goto** command in each pass through the loop
- 2 assignment statements in each pass through the loop
- the loop counter i must be incremented and tested in each pass through the loop, and upon loop termination.

If we count entering and exiting the procedure as one operation, then the total number of operations is

$$6 + 4(n - 1) + n = 5n + 2$$

- For large values of n we can ignore the 2. We often say the computing time is $O(n)$ (pronounced "big oh of n").

The Big-Oh Notation

In general, saying that the computing time is $O(g(n))$ means that for large enough n it is no more than a constant times $g(n)$.

- $O(n)$ is called *linear*.
- $O(n^2)$ is called quadratic.
- $O(\log n)$ is called logarithmic.
- $O(a^{bn})$, where a and b are constants, is called exponential.

What do you think $O(1)$ means?

Analysis of the Recursive Algorithm

If the recursive algorithm was performed naively the number of operations would be the number of additions in the recursion tree. This would perform many operations twice.

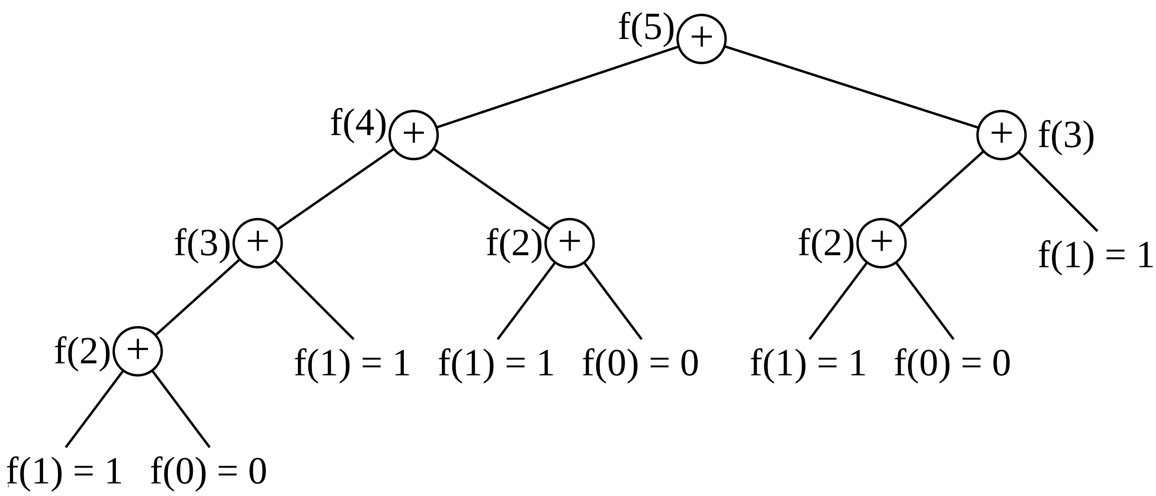


Figure 8: Representation of *fibonacci* function with a tree for $n = 5$.

If b_n is the number of additions (also called *interior nodes*) in the recursion tree for finding $f(n)$, then

$$b_n = b_{n-1} + b_{n-2} + 1$$

for $n \geq 2$ and $b_0 = b_1 = 0$. This has solution $b_n = f(n) - 1$.
It can be shown for large n that $f(n) \propto a^n$ where a is a constant. So the naive recursive algorithm would have computing time $O(a^n)$. This is called *exponential*.

Greedy Algorithms

See chapter 10 of Brunskill and Turner.

- Greedy methods are often used in optimisation problems.
- Greedy methods always choose the best option currently available.
- A greedy method may, or may not, lead to an optimal solution. If an optimal solution is reached we say the method is a *greedy algorithm*.

As an example consider the *coin change problem* in which we want to find the minimum number of coins needed to make up a particular amount of change.

Coin Change Example

Using the 7 British coins we can make up the sum of 46 pence in several ways. Some examples are shown in Figure 9.

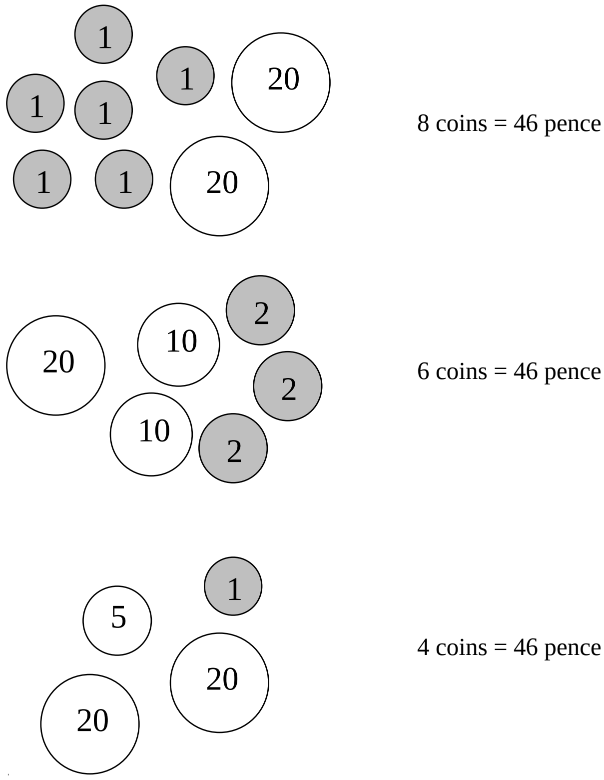


Figure 9: Some ways of making up 46p

The last way shown in Figure 9 uses 4 coins and is optimal.

Brute Strength Approach

- A *brute strength* approach would consider every possible combination of coins that adds up to 46 pence and choose the best.
- One way to do this would be to observe that we'll never need more than n coins (all pennies) to make n pence. Then we consider all possible combinations of n coins.
- There are 7^n such combinations which is very large for even fairly small values of n.
- The naive brute strength approach has computing time $O(7^n)$, i.e. it is exponential.
- We can represent the brute strength approach with a tree.

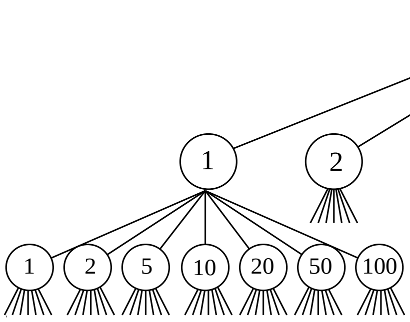


Figure 10: Schematic of tree for brute strength method for making 3p.

A Greedy Coin Change Algorithm

- The brute strength approach takes too long.
- The greedy algorithm at each stage simply takes the largest coin that is less than the remaining change required. Thus the algorithm proceeds in the following steps.

1. Choose 20p (the largest coin $\leq$ 46p). This leaves 26p.
2. Choose 20p (the largest coin $\leq$ 26p). This leaves 6p.
3. Choose 5p (the largest coin $\leq$ 6p). This leaves 1p.
4. Choose 1p.

This gives us a solution, but how do we *prove* that it is optimal, i.e. that we cannot make 46p with less than 4 coins. More generally, can we prove that the greedy method always gives an optimal solution to the coin change problem using British coins?

Leapfrogging

We say that a "leapfrog" occurs in a coin set

- if the combined value of a number of lower denominations is greater than that of a given coin,
- and if the removal of one of these lower denomination coins reduces their combined value below that of the given coin.

For example, we can leapfrog 5p because $3 \times 2p > 5p$ and $2 \times 2p < 5p$.

In any coin set in which leapfrogging is impossible the greedy method gives an optimal solution to the coin change problem.

WHY?

Because if we choose a smaller coin when we could have chosen a larger one, a subset of these smaller coins must add up to the larger one. Hence a non-greedy approach is suboptimal.

Leapfrogging can occur in the British coin set in two ways.

$$\begin{aligned} 2+2+2 &> 5 \\ 20+20+20 &> 50 \end{aligned}$$

However, we can always replace 2p+2p+2p by 5p+1p, and 20p+20p+20p by 50p+10p. So, in fact, the greedy method is an optimal algorithm for the coin change problem *if the British coin set is used.*

Another Example

Before decimalisation in 1971, British coins had the following values,

- 1d (penny)
- 3d
- 6d
- 12d (one shilling)
- 24d (two shillings)
- 30d (one half crown)

The greedy algorithm does not always work optimally for this coin set.
Try to make 48d.

Pseudocode for the Coin Change Problem

```
procedure change (n, c, value)
   declare value(1:c)
   left ← n
   coins ← 0
   choice ← c
   while choice > 0 and left > 0 do
      if value(choice) ≤ left then [
         coins ← coins + 1
         left ← left − value(choice)
         print ("Choosing coin of value ",value(choice)) ]
      else
         choice ← choice − 1
   end
   return (coins)
end
```

Here

- n is the amount of change to make
- c is the number of coins in the coin set
- *value* is an array containing the values of the coins.

Backtracking

The general idea behind *backtracking* methods is to evolve a solution in a series of steps, at each of which we have an attempted partial solution. If the partial solution at some stage is known to be a part of the final solution then we go back, or backtrack, to some previous partial solution and try to evolve to the final solution from there.
A good example is finding a path through a maze.

- We travel through the maze. At each intersection we choose one of the paths.
- When we get to a deadend we backtrack to the last intersection, and choose a path that has not yet been tried.
- If all the paths at that intersection have been tried already without success, then we backtrack to the preceding intersection, and choose an untried path there.

If we just choose paths at random then this backtracking method is just a type of brute-strength method.

Labelling Maze Intersections

- We can specify the direction taken at an intersection as North, South, East, or West. There are either 2 or 3 choices at each intersection.
- For simplicity we don't allow diagonal moves.
- We can label the intersections with integers as shown in Figure 11.

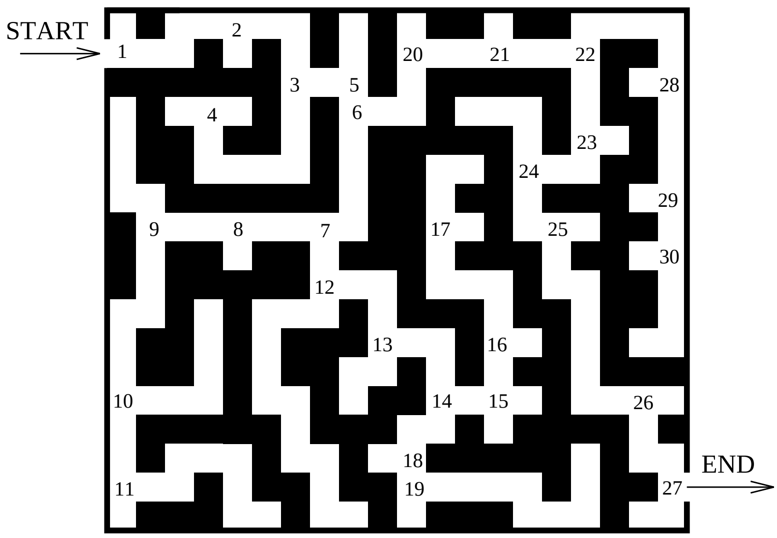


Figure 11: Maze with labelled intersections.

Paths Through the Maze

Suppose we initially follow the following path through the maze.

1. East at intersection 1.
2. East at intersection2.
3. South at intersection 3.
4. West at intersection 4.

This leads to a deadend as shown in Figure 12.

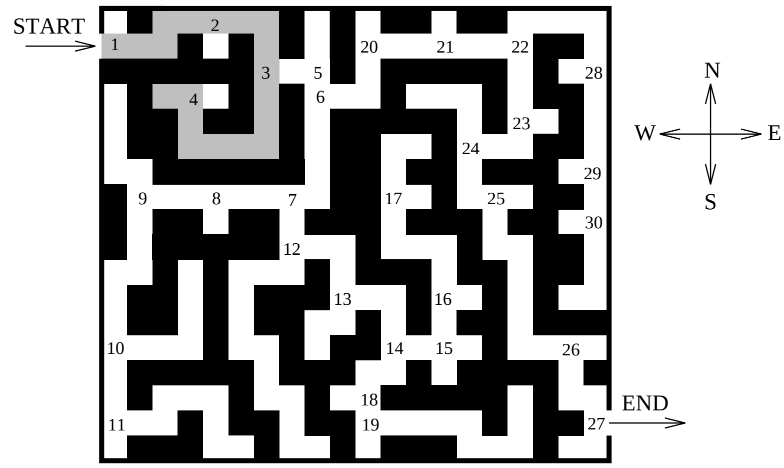


Figure 12: A first path through the maze.

Paths Through the Maze

To generate a second trial path through the maze we proceed as follows.

- Backtrack to intersection 4.
- Take the East turn. This also deadends as shown in Figure 13.

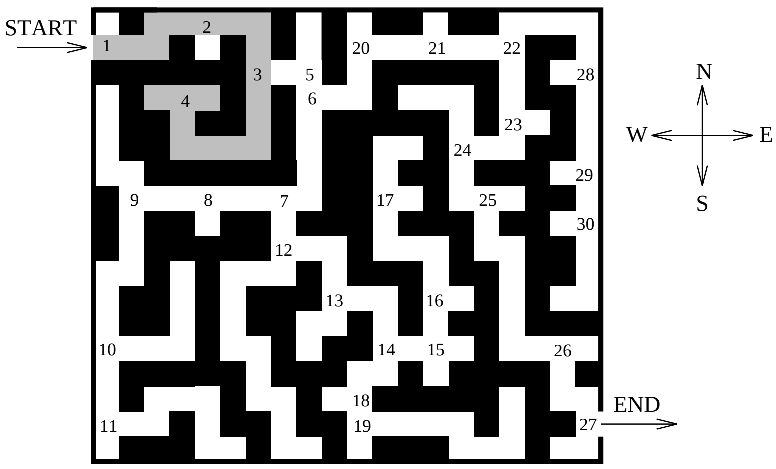


Figure 13: A second path through the maze.

Paths Through the Maze

To generate a third trial path through the maze we proceed as follows.

- Backtrack to intersection 3.
- Take the East turn to intersection 5.
- Take the North turn. This also deadends as shown in Figure 14.

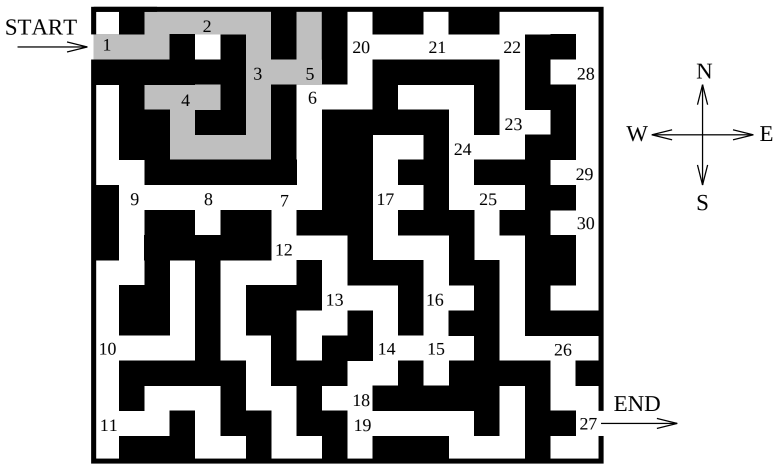


Figure 14: A third path through the maze.

Points to Note

Two important issues in designing a backtracking algorithm for the maze problem are

- What sort a data structure should be used to describe the maze. Possibilities include
  - A 2-D array of 0's and 1's, where 1's indicate wall segments, and 0's path segments.
  - A tree.
- We need to keep track of which choices lead to deadends so we don't take those paths again. In real life we might use a piece of string to mark where we have been

Tree Representation of a Maze

A maze can be represented by a tree as shown in Figure 15. Thus, the maze problem is actually a form of the problem of searching for an item in a tree.

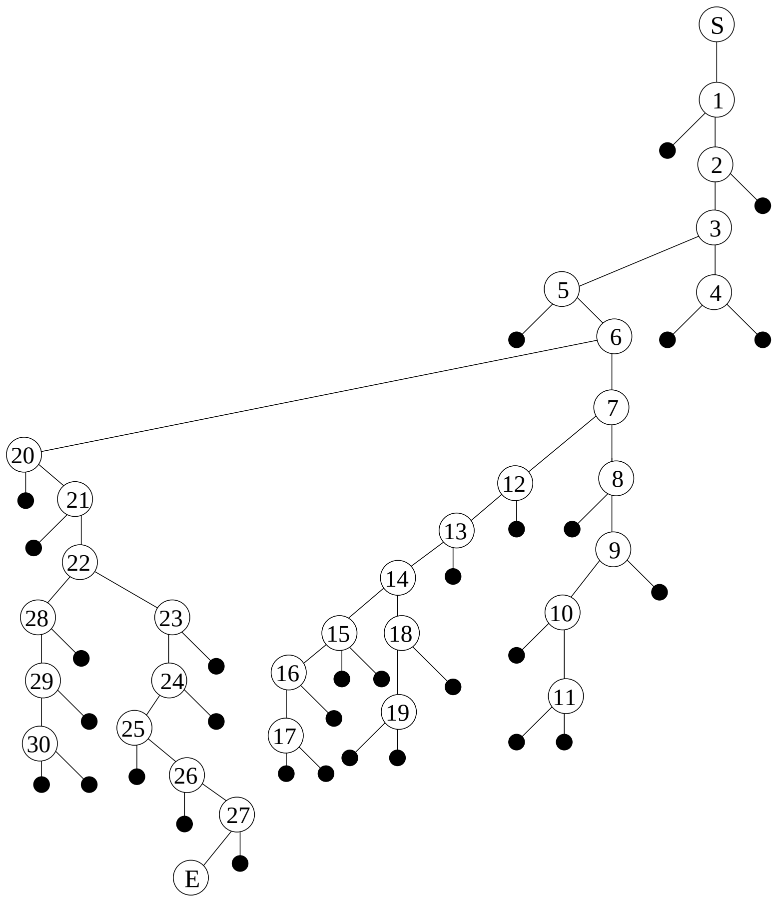


Figure 15: The tree representation of a maze.

Divide And Conquer Algorithms

The divide-and-conquer approach solves a problem by

- dividing the data into parts,
- finding subsolutions for each of the parts,
- constructing the final answer from the subsolutions.

Often, but not always, the divide-and-conquer approach results in a recursive algorithm.

Lowest Two Problem

As an example of the divide-and-conquer approach consider the *lowest two problem*:

- Given an array of n values find, in order, the lowest two values.

For simplicity we assume that n is at least 2, and is an even number.
Thus, if the array is

$$2, 4, 6, 6, 3, 7$$

then the lowest two is 2 followed by 3. Similarly, if the array is

$$3, 1, 1, 4$$

the lowest two is 1 followed by 1.
We shall consider several algorithms to show how the search for a good algorithm might progress.

Obvious Approaches

Sorting

One rather obvious approach is to sort the numbers into ascending order and then to solve the lowest two problem by just taking the first two numbers in the sorted list.
The best sorting algorithm (as we shall see later in this course) has computing time $O(n \log n)$. Can we do better than this?

Successive lowest

Another approach is to

- First find the lowest value
- Then find the second lowest value

Since it takes $n - 1$ comparisons to find the lowest value, we might expect an algorithm that finds the lowest two to take about $2n$ operations.

Successive Lowest Algorithm

The pseudocode for finding first the lowest value and then the second lowest value is as follows.

procedure *lowest_two (n, a)*
  declare *a(1:n)*
  //find lowest value and its position //
  lowest1 $\leftarrow$ *a(1)*
  position $\leftarrow$ *1*
  for $i \leftarrow 2$ **to** n **do**
    if *a(i)* $<$ *lowest1* **then** [
      lowest1 $\leftarrow$ *a(i)*
      position $\leftarrow$ *i*]
  end
  //find second lowest //
  if *position* $\neq$ *1* **then**
    lowest2 $\leftarrow$ *a(1)*
  else
    lowest2 $\leftarrow$ *a(2)*
  for $i \leftarrow 2$ **to** n **do**
    if $i \neq$ *position* **and** *a(i)* $<$ *lowest2* **then**
      lowest2 $\leftarrow$ *a(i)*
  end
  print *("Lowest values are ", lowest1, lowest2)*
end

This requires almost $3n$ comparisons – approximately n to find *lowest1* and $2n$ to find *lowest2*.
Can we do better?

A Better Algorithm

We would like to reduce the number of comparisons needed to find *lowest2*. We can do this by moving lowest value to the first location in the array after the first **for** loop.

```
procedure lowest_two (n, a)
   declare a(1:n)
   //find lowest value and its position //
   lowest1 ← a(1)
   position ← 1
   for i ← 2 to n do
      if a(i) < lowest1 then [
         lowest1 ← a(i)
         position ← i ]
   end
   //put lowest1 in first location //
   temp ← a(1)
   a(1) ← lowest1
   a(position) ← temp
   //find second lowest //
   lowest2 ← a(2)
   for i ← 3 to n do
      if a(i) < lowest2 then
         lowest2 ← a(i)
   end
   print ("Lowest values are ", lowest1, lowest2)
end
```

This requires $2n - 3$ comparisons, which is better than the previous algorithm. Can we do even better?

A Tracking Algorithm

Another approach is to continuously keep track of the lowest two values found so far, instead of first finding the lowest value and then the next lowest value.
In this approach we start by setting *lowest1* and *lowest2* to the first two values in the array. Then, as we step through the rest of the array, the value $a(i)$ of each new element gives rise to three possible cases.

1. $a(i)$ is less than *lowest2*, and less than *lowest1*. In this case, set *lowest2* = *lowest1*, and *lowest1* = $a(i)$.

2. $a(i)$ is less than *lowest2*, but not less than *lowest1*. In this case, set *lowest2* = $a(i)$, and leave *lowest1* unchanged.

3. $a(i)$ is not less than *lowest2*. Leave *lowest2* and *lowest1* unchanged.

Code for Tracking Algorithm

```
procedure lowest_two (n, a)
  declare a(1:n)
  //initialise the lowest two //
  if a(1) < a(2) then [
    lowest1 ← a(1)
    lowest2 ← a(2) ]
  else [
    lowest1 ← a(2)
    lowest2 ← a(1) ]
  for i ← 3 to n do
    if a(i) < lowest2 then [
      if a(i) < lowest1 then [
        lowest2 ← lowest1
        lowest1 ← a(i) ]
      else
        lowest2 ← a(i)
    ]
  end
  print ("Lowest values are ", lowest1, lowest2)
end
```

- In each loop we always perform the first comparison – first gives $n - 2$ comparisons.
- Some of the time we also perform the inner comparison.
- In the worst case we perform $2n - 3$ comparisons, but on average we should do better than this.

Is this the best algorithm?

Divide And Conquer Approach

So far we have just looked at iterative algorithms to solve the lowest two problem. Now we shall consider divide and conquer approaches.
The first question is

How should the data be divided?

- We could just divide the data into two halves, find the lowest value in each half, and use these as the lowest two values. This is quick as finding the lowest value in each half takes $n/2$ comparisons, but it is incorrect as the two lowest values may both lie in the same half of the data.

- We could divide the data into two halves, find the lowest two values in each half, and then merge the two resulting pairs to give the lowest two values for the whole array. This will give a correct result, but is no faster than the earlier algorithm.

- We could divide the data into two halves, sort each half in ascending order, and then find the lowest two values for the whole array by examining the first two values in each sorted subarray. Each sort requires about $(n/2)\log(n/2)$ comparisons, so this method takes at least $n \log n - n$ comparisons.

- A different approach to dividing the data is to divide it into $n/2$ pairs

A Pair-Based Algorithm

The pair-based divide and conquer algorithm proceeds as follows.

- *Sort phase*

  Sort each pair into order. This requires one comparison for each pair, i.e., $n/2$ comparisons.

- *Merge phase*

  We set *lowest1* and *lowest2* equal to the first pair. We then compare these lowest values with each successive pair, and update them as necessary. In comparing (*lowest1, lowest2*) with $(a(i), a(i+1))$ there are 4 possible outcomes. These are shown in Table 1.

In the merge phase, each comparison of pairs requires two comparisons, so the merge phase requires about n comparisons. Combining this with the $n/2$ comparisons of the merge phase gives a total of $3n/2$ comparisons for the whole algorithm.

Table of Outcomes

When comparing pairs the following outcomes are possible.

Case	New value of lowest1	New value lowest2	Comment
1	*lowest1*	*lowest2*	no change, $a(i)$ not less than *lowest2*
2	*lowest1*	$a(i)$	$a(i)$ less than *lowest2*, but not *lowest1*
3	$a(i)$	*lowest1*	$a(i)$ less than both *lowest*, but $a(i+1)$ less than neither
4	$a(i)$	$a(i+1)$	$a(i+1)$, and hence $a(i)$, less than both *lowest*

Table 1: Possible outcome of comparing pairs.

Example of Pair-Based Algorithm

Consider the pair-based algorithm using the following initial data

2, 6, 7, 3, 5, 6, 9, 2, 4, 1

- Divide the data into pairs

  (2, 6), (7, 3), (5, 6), (9, 2), (4, 1)

- Sort each pair

  (2, 6), (3, 7), (5, 6), (2, 9), (1, 4)

- Initialise the merge phase by setting

  $$lowest1 = 2, \quad lowest2 = 6$$

  – Compare $(2, 6)$ with the second pair $(3, 7)$, to give

  $$lowest1 = 2, \quad lowest2 = 3$$

  – Compare $(2, 3)$ with the third pair $(5, 6)$. This gives no change.

  – Compare $(2, 3)$ with the fourth pair $(2, 9)$, to give

  $$lowest1 = 2, \quad lowest2 = 2$$

  – Compare $(2, 2)$ with the last pair $(1, 4)$, to give

  $$lowest1 = 1, \quad lowest2 = 2$$

This gives the correct answer,

$$lowest1 = 1, \quad lowest2 = 2$$

Code for Pair-Based Algorithm

```
procedure lowest_two (n, a)
   declare a(1:n)
   //sort the pairs //
   i ← 1
   while i < n do
      if a(i) > a(i + 1) then [
         temp ← a(i); a(i) ← a(i + 1); a(i + 1) ← temp ]
      i ← i + 2
   end
   //merge the pairs //
   lowest1 ← a(1); lowest2 ← a(2)
   i ← 3
   while i < n do
      if a(i) < lowest1 then [
         if a(i+1) < lowest1 then [
            lowest1 ← a(i)
            lowest2 ← a(i + 1) ]
         else [
            lowest2 ← lowest1
            lowest1 ← a(i) ]
      ]
      else [
         if a(i) < lowest2 then
            lowest2 ← a(i)
      ]
      i ← i + 2
   end
   print ("Lowest values are ", lowest1, lowest2)
end
```

Branch and Bound

The *branch and bound* technique can improve the performance of a brute strength method by looking no further for a solution once we know it cannot be down a particular branch of the tree.

We have already applied the branch and bound technique in the brute strength approach to the coin change problem. There is no point in continuing down a branch of the tree once the sum of change so far exceeds the desired amount, n.

Branch and bound is a technique for *tree pruning* since it lops off these parts of the tree which cannot contain the solution.

Travelling Salesman Problem

The Travelling Salesman Problem (TSP) is a famous example of a problem to which a brute strength method might be applied for sufficiently small problems.

Travelling salesman problem: given a set of n cities, find the shortest route connecting all cities returning to the original city, such that no city is visited twice.

- Only solution to TSP appears to be to examine every possible tour and choose the one that's shortest. There are $n!$ tours.
- An algorithm that checks every possibility like this is called an *exhaustive search* algorithm. This a form of brute strength method.

TSP and Permutations

- In the Travelling Salesman Problem we can label the cities A, B, C, D, etc.
- If we start at city A, then any permutation of the letters, followed by A is a valid tour.
- There are $n!$ such tours for n cities.
- The permutations can be represented in a tree, as shown in Figure 16

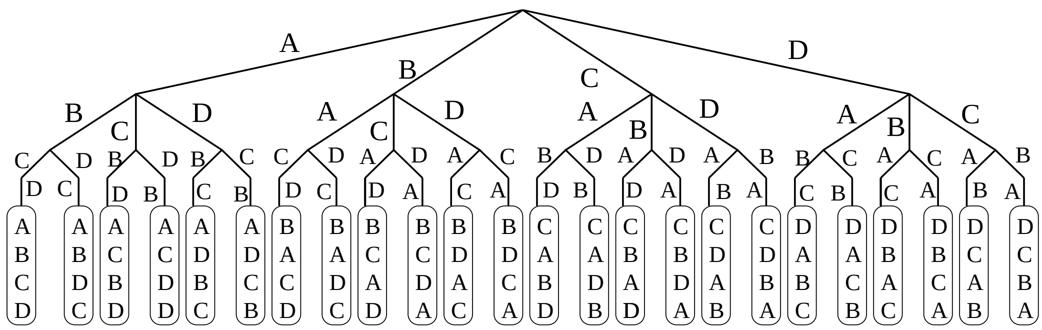


Figure 16: Permutation tree

In an exhaustive search of the tree for the shortest tour of all cities we store the shortest tour so far, and use this to prune the tree. There is no point in continuing down any branch once the length exceeds the current best.

Datatypes

We make a distinction between simple data types and data structures. Simple data types include

- integers
- real numbers, e.g., 12.87975
- characters

.

A data structure is composed of multiple simple data types. For example,

- an *array*, which is composed of data types that are all the same;
- a *record*, which is composed of different data types. For example, the following record might be used to hold personal information:

Name	string
Address	string
Age	integer
Height	real

Abstract Datatypes

An *abstract data type* (ADT) is

- a data structure,
- together with the operations defined on it. These operations can be specified using a set of rules, or axioms.

We shall look at several ADT widely used in computer science algorithms.

- arrays
- lists
- stacks
- queues

Array ADT

- An array is a set of pairs, index and value.
- Two operations are defined for an array
  - *retrieve (array, index)* which takes as input an array and an index and returns the item from the array at that index.
  - *store (array, index, value)* which stores a new index/value pair in an array.
- In addition, we must also provide a *create* function that returns a new, empty array.

.

Most high level programming languages provide explicit support for arrays, so retrieve can be expressed as

$$\mathrm{x = a(i)}$$

and store can be expressed as

$$\mathrm{a(i) = x}$$

Lists

An ordered *list* is a sequence of items (which in general may be records) together with

- a rule that implies an order for the items,
- a length — the number of items.

.

The values in a deck of cards form an ordered list:

(2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King, Ace)

The following operations are (usually) defined on lists

1. store a new item at position i
2. retrieve the item at position i
3. insert a new item at position i causing items numbered $i, i+1, \ldots, n$ to be renumbered $i+i, i+2, \ldots, n+1$.
4. delete item at position i causing items numbered $i+1, \ldots, n$ to be renumbered $i, i+1, \ldots, n-1$.

together with a function for creating a new, empty list. Other functions, such as for finding the length of the list, may also be defined.

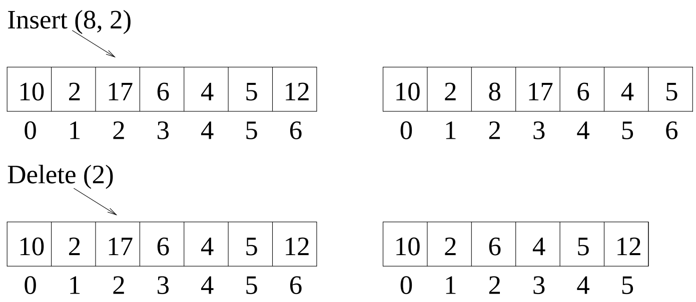
Sequential Mapping of Lists

One obvious way to represent an ordered list is with an array where we associate list item a_i with array index i. This is refered to as a *sequential mapping* because we store a_i and a_{i+1} into consecutive locations i and $i+1$ of the array.

The sequential mapping makes it easy to retrieve and store data efficiently. In fact these operations take constant time.

BUT

Insertion and deletion requires us to move some of the remaining elements to preserve the sequential mapping. These operations can be expensive, which is why other ways of managing ordered lists are often used, such as linked lists which we shall be discussing later in the course.



Pseudocode for Insertion

With a sequential mapping an new item could be inserted as follows.

procedure *insert (list, value, index)*
  if *index* < 0 **or** *index* $>$ *length(list)* -1 **then** [
    print *("Invalid position")*
    return]
  if *length(list)* $+1 >$ *maxlength(list)* **then** [
    print *("Storage limit exceeded")*
    return]
  incrementlength(list, 1)
  for $i \leftarrow$ *length(list)* **to** *index*$+1$ **by** -1 **do**
    list(i) $\leftarrow$ *list(i-1)*
  end
  list(index) $\leftarrow$ *value*
  return
end *insert*

Pseudocode for Deletion

With a sequential mapping a new item could be deleted as follows.

procedure *delete (list, index)*
  if *index* < 0 **or** *index* $>$ *length(list)* -1 **then** [
    print *("Invalid position")*
    return]
  if *length(list)* ≤ 0 **then** [
    print *("Empty list!")*
    return]
  for $i \leftarrow$ *index* **to** *length(list)* -2 **do**
    list(i) $\leftarrow$ *list(i+1)*
  end
  decrementlength(list, 1)
  return
end *delete*

Linear Search

Another operation that we might define for a list is to search for the first occurrence of a given value. We can do this by starting at the start of the list and examining each item. This is called a *linear search*.

```
procedure linear_search (list, value)
   if length(list) ≤ 0 then [
      print ("Empty list!")
      return ]
   found = 0; k = 0
   while found = 0 and k ≤ length(list)-1 do
      if list(k) = value then
         found ← 1
      else
         k ← k + 1
   end
   if found = 1 then
      print ("Found ",value," at ",k)
   else
      print ("Value not found")
   return
end linear_search
```

Binary Search

Suppose the list is sorted into ascending order. In this case we can search for a particular value by searching sequentially through the list until we find the value we are looking for, or until we come to a value that is greater. Once we reach a value that is greater than the one we are looking for we know that value is not in the list.

This is faster than sequentially searching through an unordered list, but is still has computing time $O(n)$. A faster method is the *binary search* algorithm. If x is the value we are searching for the binary search algorithm works as follows.

We look at middle item of the list. Denote this by L_m.

1. If $x < L_m$ then if x is in the list it must be in the lower half of the list.
2. If $x = L_m$ then the middle record is the one being searched for.
3. If $x > L_m$ then if x is in the list it must be in the upper half of the list.

Assuming $x \neq L_m$, we repeat the search on either the lower or upper half of the list. The algorithm continues recursively until we either find the value, or until all the sublists are of length 1.

Binary Search Example

Suppose we want to search for the value 35 in the following list

$$(3, 19, 20, 34, 42, 50, 52, 56, 57, 80)$$

1. The middle of the list is 42. Since $35 < 42$ we must continue searching the lower half of the list.
2. The middle item of (3, 19, 20, 34) is 19. Since $35 > 19$ we must continue searching the upper part of this list.
3. The middle term of (20,34) is 20. Since $35 > 20$ we must continue searching the upper part of this list.
4. The middle term of (34) is 34. Since this is not the value we are searching for and the list is of length 1 we know the item is not in the original list.

Binary Search Pseudocode

We can express the binary search algorithm with the following iterative procedure.

procedure *binary_search (list, value)*
  low $\leftarrow 0$
  high $\leftarrow$ *length(list)* -1
  while *low* $\leq$ *high* **do**
    mid $\leftarrow$ (*low*+*high*)/2
    if *list(mid)* $<$ *value* **then**
      low $\leftarrow$ *mid* $+1$
    else if *list(mid)* $>$ *value* **then**
      high $\leftarrow$ *mid* -1
    else
      return *(mid) //found //*
  end
  return *(-1) //not found //*
end *binary_search*

Points To Note

In each pass through the loop the length of the list searched is approximately halved. If the value is not found, then the while loop terminates after p iterations at which point the length of each sublist is 1, i.e.,

$$n/2^p = 1$$

This means $2^p = n$, or, $p = \log_2 n$. Thus, in the worst case in which the value sought is not in the list, the binary search algorithm has computing time $O(\log n)$ so it is a logarithm algorithm.
The binary search method is the quickest way of searching a list implemented as an array. However it requires us to maintain the list in sorted order, so the list would have to be re-sorted after each insertion or deletion. Thus, the binary search algorithm should only be used when these types of changes are infrequent.

Tree Representation

A binary search can be represented with a tree.

- The root of the tree is labelled with the middle index of the list .
- If the value associated with this index is greater than the value we are searching for we next examine the mid-point of the lower half of the list. This is represented in the tree as the left-hand child of the root
- If the value is smaller than the value we are searching for we examine the mid-point of the upper half of the list. This is represented in the tree as the right-hand child of the root
- This process continues until either find the value we are searching for or until we get to a leaf node.

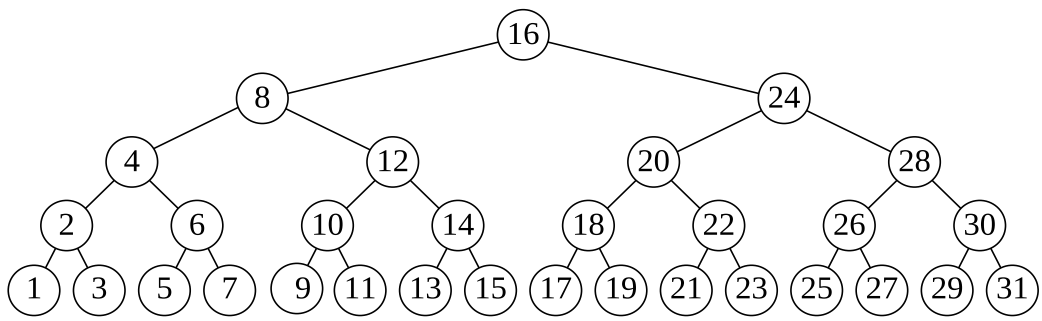


Figure 17: Tree representing a binary search of list of length 31.

Linked Lists

Using an array and a sequential mapping to represent a list has some limitations.

- When inserting or deleting an item the remaining items have to be moved in order to maintain the sequential mapping. This can be expensive, especially if the item is inserted or deleted near the start of the list.
- An array is a static data structure – once its size has been determined it cannot change. This can be a disadvantage because we might not know the maximum size of the list.

A *linked list* is a dynamic data structure that can grow and shrink, and for which insertions and deletions are more efficient.
In a singly linked list, each item is represented by a node which contains the data item itself and a link to the next node in the list, i.e., each item "knows" what the location of the next item in the list.

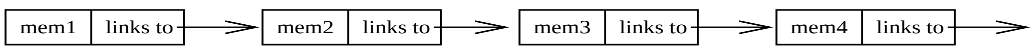


Figure 18: Schematic representation of a linked list.

Insertion Into A Linked List

A new item, Z, can be inserted into a linked list between items X and Y as follows

- Set the link for item X to point to item Z.
- Set the link for item Z to point to item Y.

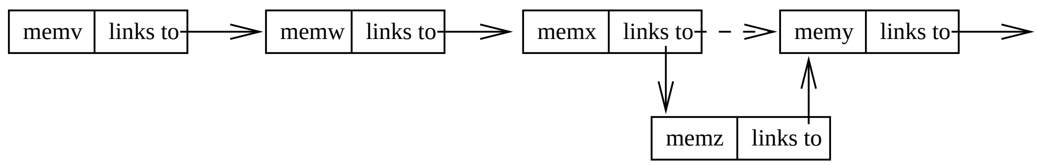


Figure 19: Insertion into a linked list.

When the new item is inserted into the list we do not have to move any other elements that are already in the list.

Deletion From A Linked List

An item, X, lying between items W and Y can be deleted from linked list as follows

- Set the link for item W to point to item Y.

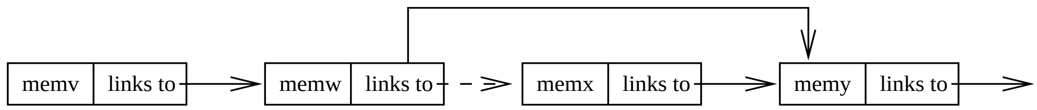


Figure 20: Deletion from a linked list.

When the item is deleted from the list we do not have to move any other items in the list.

Representing A Linked List

A linked list can be represented in a number of ways. One way uses a pair of arrays. One array contains the data in any order, and the second array contains the links. We shall call these arrays `DATA` and `LINK`. Suppose the first item in the list is at index `I` of the `DATA` array. Then the index of the next item is at `LINK(I)`. In general, the index of the next item in the list is given by the corresponding entry in the `LINK` array. An example is given in Figure 21.

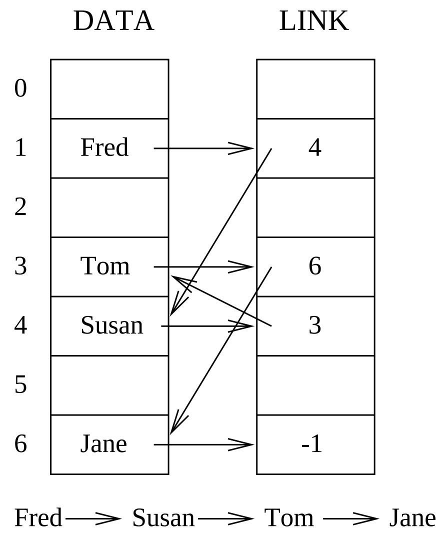


Figure 21: Representation of a linked list using arrays.

The Storage Pool

From our discussion of linked list we see that the following capabilities are needed to make linked representations possible

1. A means of dividing memory into nodes having at least one link field.
2. A mechanism to determine which nodes are in use and which are free.
3. A mechanism to transfer nodes from the reserved pool to the free pool, and vice versa.

Using the `DATA` and `LINK` arrays is one way of implementing linked lists, but not the only one. For now we assume that all free nodes are kept in an abstract space called *the storage pool*, and that there exist subalgorithms

- *getnode(x)* which provides in x a pointer to a free node, but if no free node is available prints an error message and stops.
- *ret(x)* which returns node x to the storage pool.

Creation and Insertion

We can create an empty linked list as follows. This return returns a link, or pointer.

```
procedure create_linked_list ()
   call getnode(i)
   link(i) = end_of_list
   return (i)
end create_linked_list
```

We can insert an item in a linked list after the node pointed to by x as follows.

```
procedure insert_linked_list (item, x)
   call getnode(i)
   data(i) ← item
   if link(x) = end_of_list then [
      link(i) ← end_of_list
      link(x) ← i ]
   else [
      link(i) ← link(x)
      link(x) ← i ]
end insert_linked_list
```

Deletion

Suppose we want to delete the node pointed to by x from a linked list. This is rather tricky as we need to know which node preceeds the one we want to delete so we can change its link. This would require us to search through the whole list to find the node whose link points to x. This is an $O(n)$ operation, where n is the number of items in the list.

Instead let's look at the easier task of deleting the item that follows that pointed to by x.

```
procedure delete_linked_list (x)
   if link(x) = end_of_list then [
      print ("Invalid pointer")
      stop ]
   i ← link(x)
   link(x) ← link(i)
   call ret (i)
end delete_linked_list
```

Doubly Linked Lists

Using a singly linked list it is difficult to determine which node precedes a given node. This creates difficulty, for example, when wanting to delete a given node. In problems in which it is necessary to travel in either direction along the list it is useful to have doubly linked lists.

Each node in a doubly linked list has at least three components, or fields.

1. A data field
2. A left link that points to the preceding node in the list
3. A right link that points to the next node in the list

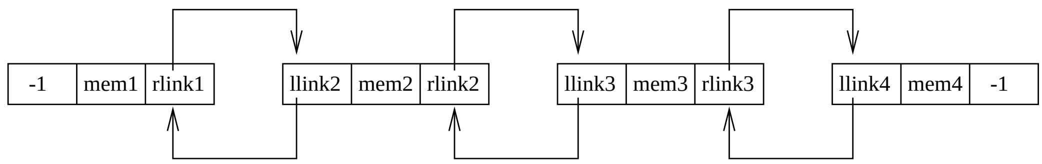


Figure 22: Schematic representation of a doubly linked list.

Creation Of A Doubly Linked List

We can create an empty doubly linked list as follows.

procedure *create_dll ()*
  call *getnode(i)*
  llink(i) = *end_of_list*
  rlink(i) = *end_of_list*
  return *(i)*
end *create_dll*

We can insert an item in a doubly linked list after the node pointed to by x as follows.

procedure *insert_dll (item, x)*
  call *getnode(i)*
  data(i) $\leftarrow$ *item*
  if *rlink(x)* = *end_of_list* **then** [
    rlink(i) $\leftarrow$ *end_of_list*
    llink(i) $\leftarrow$ x
    rlink(x) $\leftarrow$ i]
  else [
    $j \leftarrow$ *rlink(x)*
    rlink(i) $\leftarrow$ j
    llink(j) $\leftarrow$ i
    rlink(x) $\leftarrow$ i
    llink(i) $\leftarrow$ x]
end *insert_dll*

Deletion From A Doubly Linked List

We can delete the node pointed to by x from a doubly linked list as follows.

procedure *delete_dll (x)*
    $j \leftarrow$ *llink(x)*
    $k \leftarrow$ *rlink(x)*
    if $j =$ *end_of_list* **then**
        if $k \neq$ *end_of_list* **then**
            llink(k) $\leftarrow$ *end_of_list*
    else if $k =$ *end_of_list*
        rlink(j) $\leftarrow$ *end_of_list*
    else [
        rlink(j) $\leftarrow k$
        llink(k) $\leftarrow j$]
end *delete_dll*

Dynamic Storage Management

See Section 4.8 of Horowitz and Sahni.

We now consider an example of the use of doubly linked lists. On a multiprocessing computer, such as a workstation, several programs may reside in memory at the same time. These programs will, in general, require different amounts of memory, and will start and end at different times.

Whenever the operating system needs to request memory it must be allocated a block of contiguous memory of the correct size.

When a program completes execution the operating system must free the memory block allocated to it, and this memory can then be used by other programs.

There is no way of knowing ahead of time what size programs will run, or when they will start and end. We need some way of knowing which blocks of memory are free and which are allocated to some program. We can use a doubly linked list to do this.

Dynamic Storage Management

Suppose the computer memory is of size M words. Initially all of this is available to be allocated to programs.

Suppose five programs, P1, P2, P3, P4, and P5 are submitted and make requests for memory as follows.

Program	Size
P1	10000
P2	15000
P3	6000
P4	8000
P5	20000

If the total size of memory is $M = 100000$ then the status of memory after these 5 programs have been allocated is as shown in Figure 23.

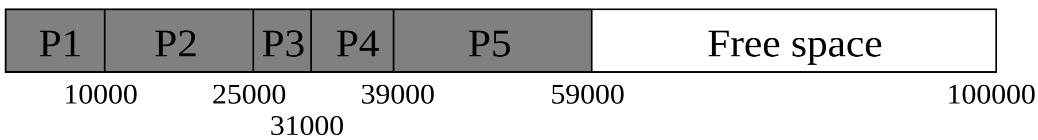


Figure 23: Memory after allocating 5 programs.

Dynamic Storage Management

Suppose programs P2 and P4 complete execution and free the memory allocated to them. Then there will be three blocks of free memory and three blocks of allocated memory, as shown in Figure 24.

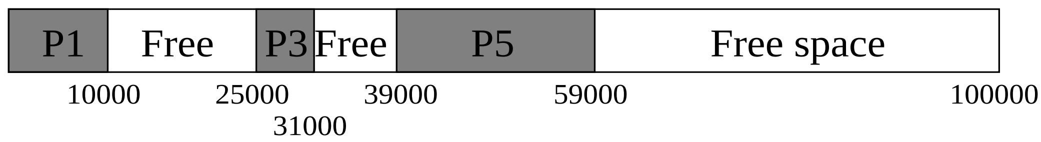


Figure 24: Memory after completing P2 and P4.

When a new program is run the operating system must select a block of contiguous memory for it from one of the free blocks.

One the the functions of an operating system is to maintain a list of all the blocks of storage currently not in use, and then to allocate storage from this unused pool as required.

This can be done by keeping a list of all free blocks.

The Free Block List

Consider the singly-linked list in which each node contains two fields, `SIZE` and `LINK`, giving the size of each free block and a link to the next one.

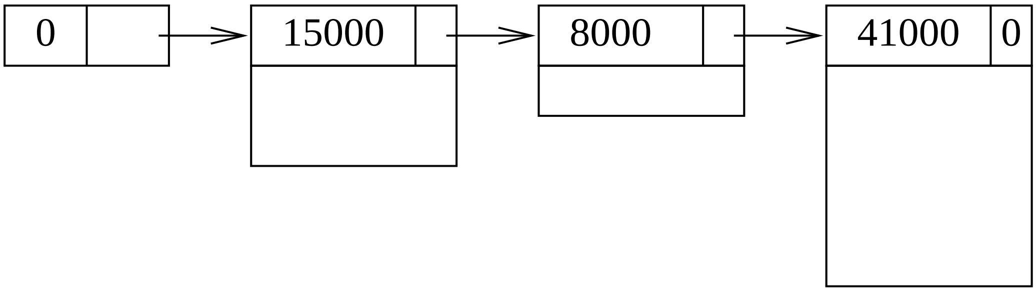


Figure 25: Simple representation of list of free blocks.

In this scheme the first two fields of each free block give its size and a link to the next free block. The list is started with an empty *head node* of size zero.

Search Algorithms

When the operating system receives a request for N words of memory the *first fit* allocation strategy searches through the list until it finds the first block with at least N words, and allocates the requested memory out of this block.

```
procedure firstfit (n, p)
   p ← link(av); q ← av
   while p ≠ end_of_list do
      if size(p) ≥ n then [
         size(p) ← size(p) − n
            if size(p) = 0 then
               link(q) ← link(p)
            else
               p ← p + size(p)
            return ]
      q ← p; p ← link(p)
   end
   //Unable to allocate block //
   p ← 0
end firstfit
```

- *av* is the start of the list.
- On exit p points to the block to be allocated, or is 0 if no block could be found.
- Memory is allocated from the end of a block to avoid changing any links (except when all of a block is allocated).

First Fit Allocation

There are two major problems with the first fit algorithm.

1. After several requests the list tends to contain many small nodes, which may be smaller than any requests that the system might make. These small nodes slow down the allocation process since the time taken is proportional to the number of nodes. We can get round this by allocating all of the block if the portion that would otherwise be left behind is smaller than some specified size, ϵ.

2. Since the search for space starts at the beginning of the list, small nodes tend to collect near the start of the list, so several nodes must be examined before we come to a big node. We can distribute the small nodes evenly through the list by beginning each search at a different location. This can be done using a *circular list*, and maintaining a pointer av, to the last node from which an allocation was made.

Freeing Blocks

When freeing a block we also want to recognise if its neighbors are also free so they can be coalasced into a single block. Returning to our earlier example, if P3 were the next program to terminate it would be better to combine it with the adjacent free blocks to give a single free block of size 29000, as shown in Figure 26.

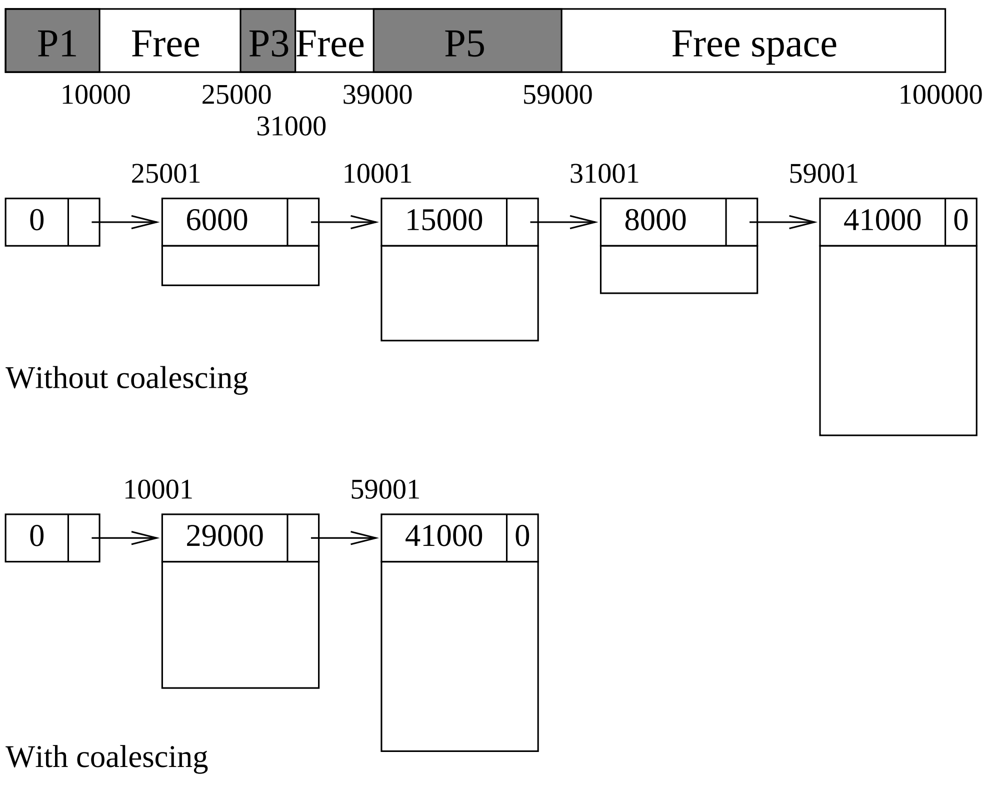


Figure 26: Coalascing adjacent free blocks.

Coalescing Free Blocks

With a singly linked list it is difficult to tell whether blocks adjacent to the one being freed are free. To do this we would have to examine all the nodes in the list and determine whether

1. the left adjacent block is free, i.e., the one ending at $p-1$,
2. the right adjacent block is free, i.e., the one beginning at $p+n$,

where the block to be freed starts at p and has size n.

The *boundary tag method* solves this problem by using a doubly linked circular list for the free blocks, and by tagging each block to indicate if it is allocated or not. The *tag* field is at the start of the first word of each block, and is repeated at the start of the last word in the block.
The boundary tag method makes it possible to return blocks to the free list and to coalesce adjacent free blocks. There are 4 possibilities

1. Neither adjacent block is free,
2. The left block is free, but the right is not,
3. The right block is free, but the left is not,
4. Both adjacent blocks are free.

Structure of Blocks

The structure of free and allocated blocks is as shown in Figure 27.

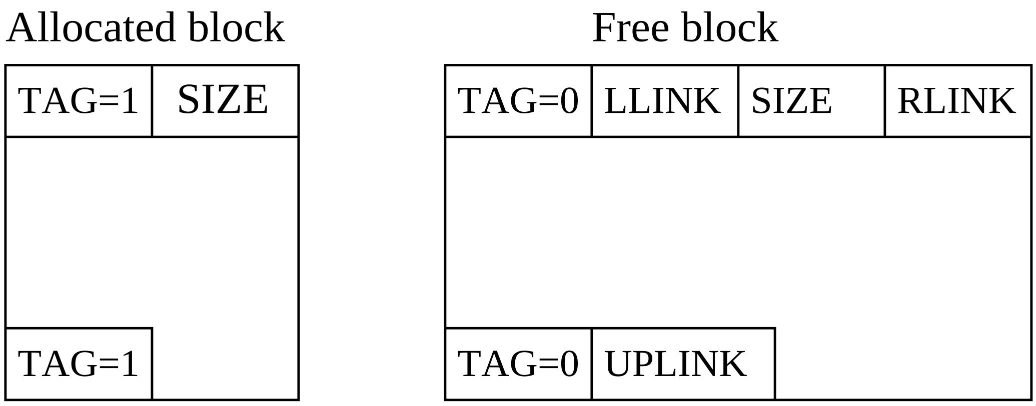


Figure 27: Structure of free and allocated blocks.

- $tag = 0$ for a free block, and $tag = 1$ for an allocated block.
- If we are examining the block at p with size n, then *tag(p-1)* will tell us if the previous block is free, and *tag(p+n)* will tell us if the next block is free.

Allocation

- The *allocate* routine is based on a doubly linked circular list.
- No blocks of size smaller than ϵ are retained.
- p is returned as the address of the start of the allocated block, or zero is no block is found large enough.

The basic structure of the *allocate* routine is as follows.

procedure *allocate (n, p)*
    $p \leftarrow rlink(av)$
    repeat
        $\vdots$
    until $p = rlink(av)$
    $p \leftarrow 0$
end *allocate*

We iterate through the loop until we either find a block large enough, in which case the routine returns from within the loop after setting p to the start of the allocated block, or until we get back to where we started in the circular list, in which case we return p as zero to indicate that no block was found.

Complete Allocation Code

```
procedure allocate (n, p)
   p ← rlink(av)
   repeat
      if size(p) ≥ n then [
         diff ← size(p) − n
         if diff < ϵ then [
            rlink(llink(p)) ← rlink(p)
            llink(rlink(p)) ← llink(p)
            tag(p) ← 1
            tag(p+size(p)−1) ← 1
            av ← llink(p)
            return ]
         else [
            size(p) ← diff
            uplink(p+diff−1) ← p
            tag(p+diff−1) ← 0
            av ← p
            p ← p + diff
            size(p) ← n
            tag(p) ← 1
            tag(p+n−1) ← 1
            return ]
      ]
      p ← rlink(p)
   until p = rlink(av)
   p ← 0
end allocate
```

Freeing A Block

In the *free* routine we handle the 4 possible cases with a **case** statement.

procedure *free (p)*
  $n \leftarrow$ *size(p)*
  case
    : *tag(p-1)* = *1* **and** *tag(p+n)* = *1* :
      ⋮   *//both adjacent blocks in use //*
    : *tag(p-1)* = *0* **and** *tag(p+n)* = *1* :
      ⋮   *//only left block free //*
    : *tag(p-1)* = *1* **and** *tag(p+n)* = *0* :
      ⋮   *//only right block free //*
    : **else** :
      ⋮   *//both adjacent blocks free //*
  end
end *free*

Both Adjacent Blocks In Use

If both adjacent blocks are in use then no coalescing is necessary, and we set up a free block as follows.

$$
\begin{array}{l}
\mathit{tag(p)} \leftarrow \mathit{0} \\
\mathit{tag(p+n\text{-}1)} \leftarrow \mathit{0} \\
\mathit{uplink(p+n\text{-}1)} \leftarrow p \\
\mathit{llink(p)} \leftarrow \mathit{av} \\
\mathit{rlink(p)} \leftarrow \mathit{rlink(av)} \\
\mathit{llink(rlink(p))} \leftarrow p \\
\mathit{rlink(av)} \leftarrow p
\end{array}
$$

Only Left Block Free

If only the left block is free then we have to coalesce with it as follows.

$$q \leftarrow \textit{uplink(p-1)}$$
$$\textit{size(q)} \leftarrow \textit{size(q)} + n$$
$$\textit{uplink(p+n-1)} \leftarrow q$$
$$\textit{tag(p+n-1)} \leftarrow 0$$

Only Right Block Free

If only the right block is free then we have to coalesce with it as follows.

$$
\begin{array}{l}
\mathit{rlink(llink(p{+}n))} \leftarrow p \\
\mathit{llink(rlink(p{+}n))} \leftarrow p \\
\mathit{llink(p)} \leftarrow \mathit{llink(p{+}n)} \\
\mathit{rlink(p)} \leftarrow \mathit{rlink(p{+}n)} \\
\mathit{size(p)} \leftarrow n + \mathit{size(p{+}n)} \\
\mathit{uplink(p{+}size(p){-}1)} \leftarrow p \\
\mathit{tag(p)} \leftarrow \mathit{0} \\
\textbf{if } av = p + n \textbf{ then } av \leftarrow p
\end{array}
$$

Both Adjacent Blocks Free

If both adjacent blocks are free we must coalesce with both of them.

$$
\begin{array}{l}
\textit{rlink(llink(p+n))} \leftarrow \textit{rlink(p+n)} \\
\textit{llink(rlink(p+n))} \leftarrow \textit{llink(p+n)} \\
q \leftarrow \textit{uplink(p-1)} \\
\textit{size(q)} \leftarrow \textit{size(q)} + n + \textit{size(p+n)} \\
\textit{uplink(q+size(q)-1)} \leftarrow q \\
\textbf{if } av = p + n \textbf{ then } av \leftarrow \textit{llink(p+n)}
\end{array}
$$

Complete Free Code

```
procedure free (p)
  n ← size(p)
  case
    : tag(p-1) = 1 and tag(p+n) = 1 :
      tag(p) ← 0; tag(p+n-1) ← 0
      uplink(p+n-1) ← p
      llink(p) ← av; rlink(p) ← rlink(av)
      llink(rlink(p)) ← p; rlink(av) ← p
    : tag(p-1) = 0 and tag(p+n) = 1 :
      q ← uplink(p-1)
      size(q) ← size(q) + n
      uplink(p+n-1) ← q; tag(p+n-1) ← 0
    : tag(p-1) = 1 and tag(p+n) = 0 :
      rlink(llink(p+n)) ← p
      llink(rlink(p+n)) ← p
      llink(p) ← llink(p+n)
      rlink(p) ← rlink(p+n)
      size(p) ← n + size(p+n)
      uplink(p+size(p)-1) ← p
      tag(p) ← 0
      if av = p + n then av ← p
    : else :
      rlink(llink(p+n)) ← rlink(p+n)
      llink(rlink(p+n)) ← llink(p+n)
      q ← uplink(p-1)
      size(q) ← size(q) + n + size(p+n)
      uplink(q+size(q)-1) ← q
      if av = p + n then av ← llink(p+n)
  end
end free
```

Stacks and Queues

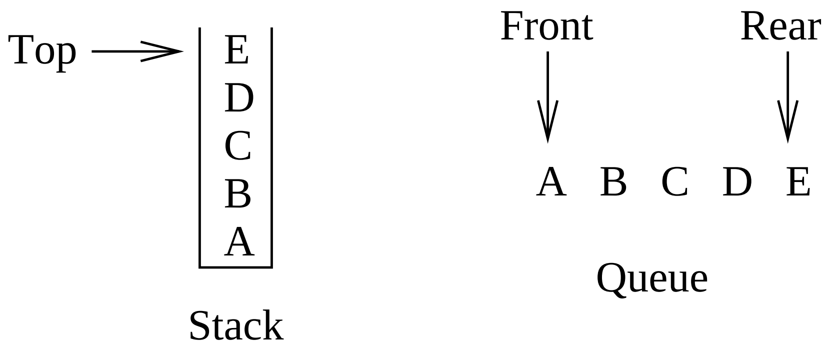
Stacks

A *stack* is an ordered list in which all insertions and deletions are made at one end, called the top of the stack.

Queues

A *queue* is an ordered list in which all insertions take place at one end, called the rear of the queue, and all deletions take place at the other end, called the front of the queue.

Given a stack $S = (a_1, \ldots, a_n)$ we say that a_1 is the bottom element and a_i is on top of a_{i-1}.



- A stack is often referred to as a Last In First Out (LIFO) list.
- A queue is often referred to as a First In First Out (FIFO) list.

Processing Subroutine Calls

One way in which a stack may be used is in the processing of subroutine calls. Suppose some main procedure calls routine A1, which in turn calls A2, which in turn calls A3.

- The operating system from which the main routine was invoked will resume execution on completion of main at some address, q.
- The main procedure will resume execution on completion of A1 at r.
- Routine A1 will resume execution on completion of A2 at s.
- Routine A2 will resume execution on completion of A3 at t.

We can imagine these return addresses being placed on a stack as the calls are made. After calling A3 the stack would look like:

$$(\mathrm{q},\ \mathrm{r},\ \mathrm{s},\ \mathrm{t})$$

Every time a return is made, it is made to the top item on the stack, which is then removed (or *consumed*).

Stack Operations

A stack has the following operations associated with it.

1. CREATE(S) which creates an empty stack S.
2. ADD(i,S) which inserts element i onto the stack S and returns the new stack.
3. DELETE(S) which removes the top element of stack S and returns the new stack.
4. TOP(S) which returns the top element of stack S.
5. ISEMTS(S) which returns true if S is empty and false otherwise.

Examples Of Stacks

Examples of the use of stacks include

1. "Reverse Polish" (or postfix) notation calculators
2. The PostScript language

Consider the evaluation of

$$(3 + 4) \times 2 + 6$$

on a RPN calculator. This is calculated by placing 6, 2, 3, and 4 on the stack and then applying the "+", "×" and "+" operators to the stack in that order. Each of these operators takes the top two items off the stack, applies the operator, and puts the result back on the stack. When something is taken off the stack we say it is "popped" off the stack. When something is put onto the stack we say it is "pushed" onto the stack. The key strokes that would be used in this example are

$$6\ 2\ 3\ 4 + \times +$$

and the stack would undergo the following transformations.

- (6 2 3 4) initially
- (6 2 7) after applying +
- (6 14) after applying × (20) finally result after applying second +.

PostScript

PostScript is a stack-based, interpreted language for graphics. Commands remove operands from the stack, perform some work (such as drawing a line), and return results, if any, to the stack.

For example

```
100 100 moveto
```

applying the **moveto** command to the operands 100 and 100, and has the effect of moving the cursor (called the *current point* in PostScript) to position (100,100) in the coordinate system. **moveto** does not push anything onto the stack.

The following PostScript program draws a box

```
100 100 moveto
300 100 lineto
300 300 lineto
100 300 lineto
closepath
stroke
```

Add and Delete Operations

The pseudocode for the *add* and *delete* operations is very simple.

procedure *add (item, stack, n, top)*
  if *top* $\geq$ *n* **then call** *stack_full*
  top $\leftarrow$ *top* + *1*
  stack(top) $\leftarrow$ *item*
end *add*

procedure *delete (item, stack, top)*
  if *top* $\leq$ *0* **then call** *stack_empty*
  item $\leftarrow$ *stack(top)*
  top $\leftarrow$ *top* $-$ *1*
end *delete*

Here the stack is represented by a one-dimensional array indexed from 1 to n where n is the maximum number of entries. The variable *top* always points to the top of the stack, and is initialised as 0.

Queue Operations

A queue has the following operations associated with it.

1. CREATEQ(Q) which creates an empty queue Q.
2. ADDQ(i,Q) which adds element i to the rear of queue Q and returns the new queue.
3. DELETEQ(Q) which removes the front element from the queue Q and returns the resulting queue.
4. FRONT(Q) which returns the front element of queue Q.
5. ISEMTQ(Q) which returns true if Q is empty and false otherwise.

Add and Delete Operations

Suppose we represent the queue by a one-dimensional array indexed from 1 to n. We need two variables, *front*, and *rear*, to keep track of the front and rear of the queue.

- *front* is always one less than the actual front of the queue.
- *rear* always points to the last item in the queue.

When a new queue is created we set *front* and *rear* to 0. Whenever *front* and *rear* are equal then the queue is empty.

The pseudocode for the *addq* and *deleteq* operations is very simple.

```
procedure addq (item, queue, n, rear)
   if rear = n then call queue_full
   rear ← rear + 1
   queue(rear) ← item
end addq

procedure deleteq (item, queue, front, rear)
   if front = rear then call queue_empty
   front ← front + 1
   item ← queue(front)
end deleteq
```

Job Queue Example

Suppose we use a queue represented by a linear array to hold a job queue. Initially the queue is empty. Jobs join and leave the queue as shown in Table 2

front	rear	Q(1)	Q(2)	Q(3)	Q(4)	Q(5)	Remarks
0	0						Queue empty
0	1	J1					Job 1 joins queue
0	2	J1	J2				Job 2 joins queue
0	3	J1	J2	J3			Job 3 joins queue
1	3		J2	J3			Job 1 leaves queue
1	4		J2	J3	J4		Job 4 joins queue
2	4			J3	J4		Job 2 leaves queue

Table 2: Example of use of queue for managing a job queue.

Note that the queue gradually moves to the right of the array. We could move the whole queue to the left so it starts again at index 1 every time it becomes full.

Using A Circular Array

Using a linear array to represent a queue results is an inefficient use of memory which can only be solved by extra work, i.e., moving the queue back to the start of the array.

A better approach is to regard the array as circular.

In this case it is more convenient when designing the *addq* and *deleteq* procedures to index the array from 0 to $n-1$, instead of from 1 to n.

When $rear = n-1$ the next element added to the queue will be added at index 0 (if it is free). As before *rear* always points to the last item in the queue. *front* always points one position anticlockwise from the first item. $front = rear$ if, and only if, the queue is empty.

Add and Delete Operations With A Circular Array

When a circular array is used pseudocode for the *addq* and *deleteq* operations is as follows.

procedure *addq (item, queue, n, rear)*
  rear ← (*rear* + *1*)**mod** n
  if *front* = *rear* **then call** *queue_full*
  queue(rear) ← *item*
end *addq*

procedure *deleteq (item, queue, front, rear)*
  if *front* = *rear* **then call** *queue_empty*
  front ← (*front* + *1*)**mod** n
  item ← *queue(front)*
end *deleteq*

Here x **mod** y is the remainder when x is divided by y, where x and y are integers.

Sorting

Sorting a set of items into some order is important in many areas. Often sorting is a necessary first step in some other process, such as searching for a particular item, as we saw in the binary search algorithm.

We shall examine several sorting algorithms, and analyse their performance. We shall look at

- Selection sort
- Bubble sort
- Insertion sort
- Merge sort
- Heap sort
- Quicksort

Stability

In many of the examples we shall discuss we shall be sorting arrays of integers. In many applications, however, we need to sort records according to the value of a particular field or *key*. For example, if we want to sort records of personal information the key might be a person's name, and other fields might include data such as address, age, etc.

Another example involves sorting a set of exam results into order. We might initially have a set of records each consisting of a person's name and their mark in the exam. Suppose we want to sort these records according to the exam mark. In this case the exam mark would be the key.

Consider what happens to records with equal keys (i.e., equal exam results). The sorting algorithm may, or may not, preserve the alphabetical order for equal keys, depending on what algorithm is used.

A sorting method is called *stable* if it preserves the relative order of equal keys in the input data.

Sorting Records

Most sorting algorithms can be described in terms of two basic operations,

1. Compare two records.
2. Exchange two records.

Comparison of two records involves examining their respective keys. Exchanging two records can involve a lot of work if the records are very long.

Thus, sorting algorithms often sort records by manipulating an array of indices, rather than moving records. On completion of the sort algorithm the auxiliary index array contains the order of the sorted records.

Records to be sorted		Index array: Before	Index array: After
45	Adams, John	1	4
53	Barrett, Jane	2	5
49	Dodgeson, Peter	3	2
72	Singleton, Valerie	4	3
66	Vicks, David	5	1

Selection Sort

Suppose we want to sort a set of n integers into ascending order. A simple way to do this is as follows.

1. Look through the list to find the smallest item
2. Exchange this item with the first item in the list. This moves the smallest item to the start of the list, as required.
3. Look through the list to find the second smallest item. Exchange this with the second item in the list. We start examining the list at the second item since we know the first item in the list is the smallest.
4. Repeat the same basic steps for the third, fourth, fifth, etc., smallest item in the array. In step i of the algorithm we find the ith smallest item and move it to the ith location in the list.

This algorithm is how most people would sort a list of integers, and is known as the *selection sort* algorithm.

Selection Sort Example

Consider how selection sort would work for the following list

(10, 4, 14, -3, 12, 6)

1. Examine items starting at location 1, find smallest item, and exchange with first item

   (-3, 4, 14, 10, 12, 6)

2. Examine items starting at location 2, find second smallest item, and exchange with second item

   (-3, 4, 14, 10, 12, 6)

3. Examine items starting at location 3, find third smallest item, and exchange with third item

   (-3, 4, 6, 10, 12, 14)

4. Examine items starting at location 4, find fourth smallest item, and exchange with fourth item

   (-3, 4, 6, 10, 12, 14)

5. Examine items starting at location 5, find fifth smallest item, and exchange with fifth item

   (-3, 4, 6, 10, 12, 14)

Note that to sort n items requires $n-1$ passes through the list. The last pass compares the last two items in the list, and places them in the correct order.

Pseudocode For Selection Sort

The pseudocode for the selection sort algorithm is as follows.

procedure *selection_sort (list)*
  $n \leftarrow$ *length(list)*
  for $i \leftarrow 1$ **to** $n - 1$ **do**
    $min \leftarrow i$
    for $j \leftarrow i + 1$ **to** n **do**
      if *list(j)* $<$ *list(min)* **then** $min \leftarrow j$
    end
    if $(i \neq min)$ **then** [
      $temp \leftarrow$ *list(i)*
      list(i) $\leftarrow$ *list(min)*
      list(min) $\leftarrow temp$]
  end
end *selection_sort*

The number of passes through the inner loop is

$$(n-1)+(n-2)+\cdots+2+1=n(n-1)/2$$

The number of passes through the outer loop is $n-1$, so the total number of comparisons is

$$(n+2)(n-1)/2$$

The computing time for selection sort is $O(n^2)$, so it is a quadratic algorithm.

Bubble Sort

Bubble sort sorts a list of n integers as follows.

1. Compare items 1 and 2 and exchange them, if necessary. Then compare items 2 and 3 and exchange them, if necessary. Continue in this way, comparing pairs of items, until reaching the end of the list

2. After competing $n - 1$ passes as described in step 1 the list will be sorted.

Note that it doesn't matter whether each pass starts at the beginning and works forward, or whether it starts at the end of the list and works back.
We can sort into ascending or descending order.

Bubble Sort Example

Consider how bubble sort would work for the following list

(10, 4, 14, -3, 12, 6)

1. The first pass would transform the list in the following steps.
   (a) (4, 10, 14, -3, 12, 6). Compare items 1 and 2, and exchange.
   (b) (4, 10, 14, -3, 12, 6). Compare items 2 and 3, and leave.
   (c) (4, 10, -3, 14, 12, 6). Compare items 3 and 4, and exchange.
   (d) (4, 10, -3, 12, 14, 6). Compare items 4 and 5, and exchange.
   (e) (4, 10, -3, 12, 6, 14). Compare items 5 and 6, and exchange.

Bubble Sort Example

2. The second pass would transform the list in the following steps.
   (a) (4, 10, -3, 12, 6, 14). Compare items 1 and 2, and leave.
   (b) (4, -3, 10, 12, 6, 14). Compare items 2 and 3, and exchange.
   (c) (4, -3, 10, 12, 6, 14). Compare items 3 and 4, and leave.
   (d) (4, -3, 10, 6, 12, 14). Compare items 4 and 5, and exchange.
   (e) (4, -3, 10, 6, 12, 14). Compare items 5 and 6, and leave.

Bubble Sort Example (Continued)

3. The third pass would transform the list in the following steps.
   (a) (-3, 4, 10, 6, 12, 14). Compare items 1 and 2, and exchange.
   (b) (-3, 4, 10, 6, 12, 14). Compare items 2 and 3, and leave.
   (c) (-3, 4, 6, 10, 12, 14). Compare items 3 and 4, and exchange.
   (d) (-3, 4, 6, 10, 12, 14). Compare items 4 and 5, and leave.
   (e) (-3, 4, 6, 10, 12, 14). Compare items 5 and 6, and leave.
4. The fourth and fifth passes would leave the list unchanged as it is sorted after the third pass.

Pseudocode For Bubble Sort

The pseudocode for the bubble sort algorithm is as follows.

```
procedure bubble_sort (list)
  n ← length(list)
  for i ← 1 to n − 1 do
    for j ← 1 to n − 1 do
      if list(j+1) < list(j) then [
        temp ← list(j)
        list(j) ← list(j+1)
        list(j+1) ← temp ]
    end
  end
end bubble_sort
```

The number of comparisons in each pass is $n - 1$ so the total number of comparisons is $(n-1)^2$. So the computing time for bubble sort is $O(n^2)$, and this is also a quadratic sorting algorithm.

Improved Bubble Sort

The bubble sort algorithm just described is one of the least efficient ways of sorting data and is rarely used. One source of inefficiency is that data that has already been sorted as the algorithm progresses is re-examined in each pass. For example, in our example, after the first pass we have

(4, 10, -3, 12, 6, 14)

The 14 is now in the correct position, but it is examined in every subsequent pass.

It is generally true that if we work from left to right then at the end of the first pass the largest item will be in the correct position. And at the end of the second pass the second largest item will be in the correct position, and so on. We can, therefore, reduce the upper loop bound of the inner loop by 1 for each pass.

Improved Bubble Sort

The pseudocode of this improved bubble sort algorithm then becomes as follows.

```
procedure bubble_sort2 (list)
   n ← length(list)
   for i ← 1 to n − 1 do
      for j ← 1 to n − i do
         if list(j+1) < list(j) then [
            temp ← list(j)
            list(j) ← list(j+1)
            list(j+1) ← temp ]
      end
   end
end bubble_sort2
```

This reduces the number of comparisons to $n(n-1)/2$.

Another Improvement To Bubble Sort

Another point to note about the bubble sort algorithm is that it continues making passes through the data, even if it has been completely sorted already. Thus, even if the data is initially in sorted order the algorithm will still make $(n-1)^2$ comparisons although no values will ever be exchanged.

We can avoid this problem by setting a marker in the inner loop which records whether any exchanges take place on a particular pass. If this marker is left unset then no changes took place and the algorithm can terminate.

Another Improvement To Bubble Sort

The amended pseudocode to terminate execution once the list is sorted is as follows.

```
procedure bubble_sort3 (list)
  n ← length(list)
  limit ← n − 1
  done ← 0
  while done = 0 do
    done ← 1
    for j ← 1 to limit do
      if list(j+1) < list(j) then [
        temp ← list(j)
        list(j) ← list(j+1)
        list(j+1) ← temp
        done ← 0 ]
    end
    limit ← limit − 1
  end
end bubble_sort3
```

Insertion Sort Algorithm

Insertion sort works the way you might sort a hand of cards.

- Start with an empty hand and the cards face down on the table
- Remove one card at a time from the deck and insert it in correct position in the hand.
- To find the correct position for a card, compare it with each of the cards already in the hand from right to left.

Suppose the numbers to be sorted are as follows

$$10, 4, 3, 5, 7, 2$$

- Take second number (4) and place in in correct position amid numbers to left. Gives 4, 10, 3, 5, 7, 2
- Take third number (3) and place in in correct position amid numbers to left. Gives 3, 4, 10, 5, 7, 2
- Take 5, gives 3, 4, 5, 10, 7, 2
- Take 7, gives 3, 4, 5, 7, 10, 2
- Take 2, gives 2, 3, 4, 5, 7, 10

Insertion Sort Algorithm

Given a list of integer we can express the insertion sort algorithm with the following pseudocode.

procedure *insertion_sort (list)*
    for $j \leftarrow 2$ **to** *length(list)* **do**
        lj $\leftarrow$ *list(j)*
        $i \leftarrow j - 1$
        while $i > 0$ **and** *list(i)* $>$ *lj* **do**
            list(i+1) $\leftarrow$ *list(i)*
            $i \leftarrow i - 1$
        end
    list(i+1) $\leftarrow$ *lj*
    end
end *insertion_sort*

Analysis of Insertion Sort

We want to analyse the insertion sort algorithm to get the running time in terms of the input size.

- The input size is the number of items to be sorted
- The running time is the number of primitive steps executed

```
    procedure insertion_sort (list)
1.    for j ← 2 to length(list) do
2.      lj ← list(j)
3.      i ← j − 1
4.      while i > 0 and list(i) > lj do
5.        list(i+1) ← list(i)
6.        i ← i − 1
7.      end
8.    list(i+1) ← lj
9.    end
    end insertion_sort
```

- Let c_k be the cost for executing line k once
- Let t_j be the number of times the while loop test is executed for a particular value of j.

Running time is

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + \\
&\quad c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)
\end{aligned}
$$

Best Case Analysis

In analysing algorithms we are usually interested in the running time for

- the best case
- the worst case
- the average case

For insertion sort the best case occurs when the input data are already sorted since the while test is executed only once, i.e., $t_j = 1$.

$$\begin{aligned}T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_8)n - (c_2 + c_3 + c_4 + c_8)\end{aligned}$$

So the best case running time is a linear function of n.

Worst Case Analysis

For insertion sort the worst case occurs when the input data is originally in reverse order, i.e., in decreasing order.

- Each item must be compared with all those to the left
- $t_j = j$

Recall that

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 \text{ and } \sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

then we have

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_5\left(\frac{n(n-1)}{2}\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2}\right) n^2 + \\ &\quad \left(c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_8\right) n \\ &\quad -(c_2 + c_3 + c_4 + c_8) \end{aligned}$$

So the worst case running time is a quadratic function of n.

Average Case Analysis

For insertion sort we have to check *list(j)* against the already-sorted list *list(1)* up to *list(j-1)*. On average we would expect half of these numbers to be greater than *list(j)* and half less. So $t_j = j/2$.

If we do the arithmetic with $t_j = j/2$, we find the average case running time is also quadratic in n.

Worst case running time is often most important because

- It is an upper bound on the running time for any input
- For some algorithms the worst case occurs fairly often, e.g., searching a database for an item of information that is not in the database.
- The average case is often almost as bad as the worst case.
- It is often difficult or impossible to analyse the average case analytically.

Shellsort

One problem with insertion sort is that it can be slow if some items in the list are initially far from their final position. This is because it exchanges only adjacent items.

Shellsort gains speed by allowing exchanges of items that are far apart.

A list is said to be *h-sorted* if every hth item yields a sorted list. Shellsort first produces an h-sorted list using a fairly large value of h, reduces h and repeats the process. We continue in this way, using successively smaller values of h, until we finally sort with $h = 1$ which produces the final sorted list.

	1	20	17	19	21	9	14	7	5	24	1	13	15	12	5
$h = 13$	1	20	17	19	21	9	14	7	5	24	1	13	15	12	5
$h = 13$	1	5	17	19	21	9	14	7	5	24	1	13	15	12	20
$h = 4$	1	5	17	19	5	9	14	7	15	24	1	13	21	12	20
$h = 4$	1	5	17	19	5	9	14	7	15	12	1	13	21	24	20
$h = 4$	1	5	1	19	5	9	14	7	15	12	17	13	21	24	20
$h = 4$	1	5	1	7	5	9	14	13	15	12	17	19	21	24	20
$h = 1$	1	1	5	5	7	9	12	13	14	15	17	19	21	20	24

Shellsort Algorithm

Given a list of integers we can express the Shellsort algorithm with the following pseudocode.

procedure *shellsort (list)*
    $n \leftarrow$ *length(list)*
    $h \leftarrow$ *1*
    while $h \leq n/9$ **do**
        $h = 3 * h + 1$
    end
    while $h > 0$ **do**
        for $j \leftarrow h + 1$ **to** n **do**
            lj $\leftarrow$ *list(j)*
            $i \leftarrow j$
            while $i > h$ **and** *list(i-h)* $>$ *lj* **do**
                list(i) $\leftarrow$ *list(i-h)*
                $i \leftarrow i - h$
            end
            list(i) $\leftarrow$ *lj*
        end
        $h = h/3$
    end
end *shellsort*

Explanation Of The Shellsort Code

The Shellsort code interleaves the insertion sort steps for different h-sorted sublists in the **for** loop over j. Suppose $n = 15$ and initially the list to be sorted is

$$(15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1).$$

In this case the sequence of h values is just 4, 1.
For $h = 4$ we now examine how the list is transformed as the algorithm loops over j in the **for** loop.

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Initial	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
$j = 5$	11				15										
$j = 6$		10				14									
$j = 7$			9				13								
$j = 8$				8				12							
$j = 9$	7				11				15						
$j = 10$		6				10				14					
$j = 11$			5				9				13				
$j = 12$				4				8				12			
$j = 13$	3				7				11				15		
$j = 14$		2				6				10				14	
$j = 15$			1				5				9				13

So after h-sorting the list with $h = 4$ the list is as follows,

$$(3, 2, 1, 4, 7, 6, 5, 8, 11, 10, 9, 12, 15, 14, 13)$$

Notes on the Shellsort Algorithm

Here we have used $\ldots, 1093, 364, 121, 40, 13, 4, 1$ as the sequence of h values. Other sequences could be used, but this one has been found to be efficient in practice. Some sequences will lead to bad performance. For example, the sequence, $\ldots, 64, 32, 16, 8, 4, 2, 1$ is likely to lead to bad performance because items at odd positions are not compared with items at even positions until we get to $h = 1$.

For the sequence of h values used here it can be shown that Shellsort never does more than $n^{3/2}$ comparisons.

Shellsort is the method of choice for many sorting applications because it is fairly fast, and can be implemented with a small amount of code.

Quicksort

Quicksort is a very popular, general-purpose sorting algorithm. Its attractive features are

1. It is easy to implement
2. On average it requires only $n \log n$ operations to sort a list of n items.
3. It does not require much additional memory.

The main drawbacks for Quicksort are

1. It is recursive so is difficult to implement in programming languages that do not support recursion.
2. It takes about n^2 operations in the worst case.

The Basic Quicksort Algorithm

Quicksort is a divide-and-conquer algorithm. It works by partitioning a list into two parts, then sorts the parts independently.
The basic Quicksort algorithm is as follows.

procedure *quicksort (list, lower, upper)*
  if *upper* > *lower* **then** [
    j ← *partition (list, lower, upper)*
    call *quicksort (list, lower, j − 1)*
    call *quicksort (list, j + 1, upper)*]
end *quicksort*

- The variables *lower* and *upper* delimit the sublist within the original list to be sorted. So *quicksort (list, 1, n)* sorts the whole list.
- The procedure *partition* is crucial to the algorithm.

Quicksort Example

The partition procedure can be simply implemented as follows.

1. Arbitrarily chosen *list(r)* to be the one to be placed in the correct position.
2. Scan the list from the left until an element greater than or equal to *list(r)* is found.
3. Scan the list from the right until an element less than or equal to *list(r)* is found.
4. Since the items found in steps 2 and 3 are in the wrong positions exchange them.
5. Continue in this way until the scan from the left overlaps the scan from the right.
6. Exchange *list(r)* with the item at the location where the scan from the right finished.

Quicksort Example

Suppose we chose the first item in the list as the one to be moved to the correct position, i.e., r is set to *lower*. Then an example of how the partition procedure works is given in Figure 28

10	11	7	12	5	6	9	11	4	$\boxed{8}$
10	8	7	12	5	6	9	11	$\boxed{4}$	11
10	8	7	4	5	6	$\boxed{9}$	11	12	11
9	8	7	4	5	6	10	11	12	11

Figure 28: Example of partition procedure

The Partition Procedure

The partition procedure performs the following tasks.

1. It places *list(i)* in its final place, for some i.
2. It rearranges *list* so that all items from *list(1)* up to *list(i-1)* are less than or equal to *list(i)*.
3. It rearranges *list* so that all items from *list(i+1)* up to *list(n)* are greater than or equal to *list(i)*.

Quicksort Pseudocode

The code for the complete Quicksort algorithm can now be written as follows.

procedure *quicksort (list, lower, upper)*
  if *upper* $>$ *lower* **then** [
    $i \leftarrow$ *lower;* $j \leftarrow$ *upper* $+$ *1*
    $v \leftarrow$ *list(lower)*
    loop
      repeat $i \leftarrow i +$ *1* **until** *list(i)* $\geq v$
      repeat $j \leftarrow j -$ *1* **until** *list(j)* $\leq v$
      if $i < j$ **then**
        call *exchange (list, i, j)*
      else
        exit
    forever
    call *exchange (list, lower, j)*
    call *quicksort (list, lower,* $j -$ *1)*
    call *quicksort (list,* $j +$ *1, upper)*]
end *quicksort*

Here the procedure *exchange(list,i,j)* exchanges the items of *list* indexed by i and j.
To make this algorithm work correctly when *list(lower)* is the largest item in the list we must add a *sentinel* item to the end of the list , i.e., at *list(n+1)* that will terminate the leftward scan.

Analysis of Quicksort

The Quicksort algorithm we have discussed has two major drawbacks when applied to files that are already in order

1. The number of comparisons is about $n^2/2$.
2. The procedure calls itself n times, requiring a lot of space to handle the recursion.

In the best case the the partition procedure would partition the sublists into equal halves. If C_n is the number of comparisons needed to sort a list of length n then we have

$$C_n = 2C_{n/2} + n$$

since n comparisons are needed in each partitioning. We can write

$$\begin{aligned} C_n &= n + 2(n/2 + 2(n/4 + 2(n/8 + \cdots))) \\ &= n + n + n + n + \cdots = Ln \end{aligned}$$

where L satisfies $n/2^L = 1$, i.e., $L = \log_2 n$. Thus, in the best case the number of comparisons is approximately $n \log_2 n$
A more detailed analysis shows that on average the number of comparisons is $2n \ln n$. So in the best and average cases Quicksort is an $O(n \log n)$ algorithm.

Non-Recursive Quicksort

For the recursive version of the Quicksort algorithm the computer has to "remember" where to resume execution on returning from each call to the recursive procedure. A stack is used to do this. In the worst case in which the list is already sorted a stack of size n would be required since n recursive calls would be made.

If we re-write the procedure in non-recursive form we had avoid the need for such a large stack. We can do this by using an explicit stack.

The basic idea is as follows. After each partitioning put the indices delimiting the larger of the two sub-lists on the stack and then partition the smaller subfile. When we get to a sub-list of length 1, we pop a new sub-list of the stack and process that. When the stack is empty then we are finished.

Non-Recursive Quicksort Pseudocode

The code for the non-recursive Quicksort algorithm can be written as follows.

```
procedure quicksort2 (list, n)
  lower ← 1; upper ← n
  call creates (stack)
  loop
    while upper > lower then
      j ← partition (list, lower, upper)
      if j − lower > upper − j then [
        call adds (lower, stack)
        call adds (j-1, stack)
        lower ← j + 1 ]
      else [
        call adds (j+1, stack)
        call adds (upper, stack)
        upper ← j − 1 ]
    end
    if isemts (stack) then exit
    upper ← deletes (stack)
    lower ← deletes (stack)
  forever
end quicksort2
```

Since each sub-list on the stack represents a sub-list less than half the length of the entry below it then the stack needs to have room for only about $\log n$ entries.

Tree Representation of Quicksort

The Quicksort algorithm can be represented using a binary tree. Suppose we have 20 integers to sort. We can represent this as “1:20”, meaning everything in the list from index 1 to index 20. Suppose in the first partitioning we put place the correct item in position 9 of the list. We then call the *quicksort* procedure again for 1:8 and 10:20. This partitioning can be represented in a tree in which the initial list is the root node and the two sub-lists generated by the partitioning are its children. As we continue the process each sub-list gets partitioned further until we get down to sub-lists of just one item which cannot be partitioned further.

Tree Representation of Quicksort

Suppose the partitioning generated by the Quicksort algorithm for some list is as shown in Figure 29. Each node shows the values of *lower* and *upper* on entry to *quicksort*. For each interior node we show the index at which the partitioning was performed.

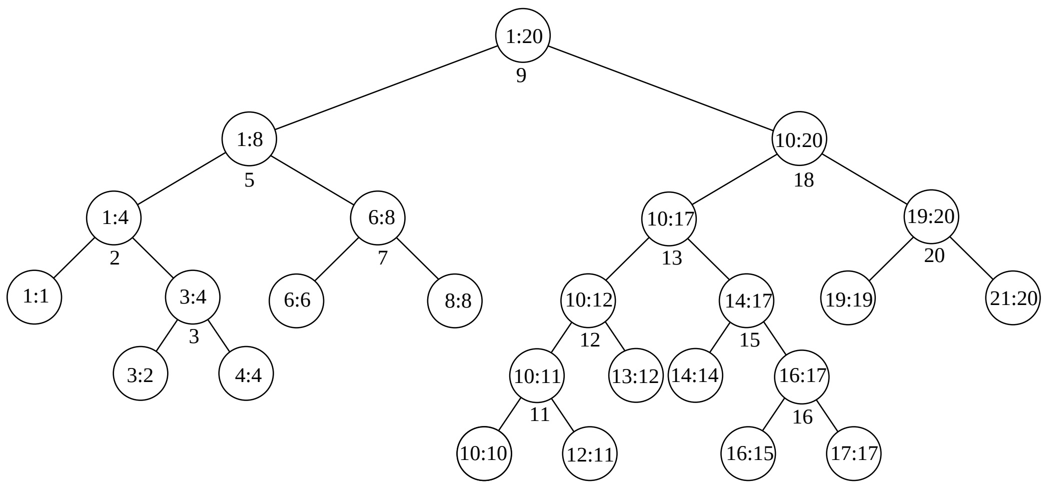


Figure 29: Tree representation of Quicksort.

- The recursive Quicksort algorithm processes the sub-lists in a depth-first (or *preorder*) way. That is, it always takes the lefthand node until it gets to a leaf node. Then it backtracks to the most recent unprocessed sub-list and processes that in the same way.
- The non-recursive algorithm processes the same sub-lists but in a different order. It processes the smallest sub-lists first until it gets to a leaf node. Then it backtracks to the most recent unprocessed sub-list and processes that in the same way.

Improving Quicksort Further

In addition to the removal of recursion, Quicksort can also be improved as follows.

- Once the sub-lists are shorter than a specified length use insertion sort to sort them, instead of continuing the recursion. This avoids the overhead of manipulating the stack for short sub-lists.
- Use a better partitioning element. If we are unlucky in our choice of partitioning elements (i.e., partitioning element is always the smallest item in the sub-list) the computing time of the Quicksort algorithm is $O(n^2)$. This will be the case is the list is already sorted. We could select a partitioning element at random. A more common method is the *median-of-threes* partitioning method.

The median-of-threes method works as follows

1. Take the first, the last and the middle items in the list and sort them.
2. Select the median of these sorted items and exchange it with the element first element.
3. Apply the standard partitioning algorithm to the sub-list starting at *lower+1* and ending at *upper-1*.

This method makes the worst case much less likely.

Merging Sorted Lists.

Before looking at the merge sort algorithm we shall first look at the problem of merging two lists, each of which is already sorted. The two lists are,

$$(x_\ell, x_{\ell+1}, \ldots, x_m)$$
$$(x_{m+1}, x_{m+2}, \ldots, x_n)$$

The merged list is

$$Z = (z_\ell, z_{\ell+1}, \ldots, z_n)$$

The pseudocode for the *merge* procedure is as follows.

procedure *merge (x, l, m, n, z)*
  $i \leftarrow \ell$; $j \leftarrow m + 1$
  $k \leftarrow \ell$
  while $i \leq m$ **and** $j \leq n$ **do**
    if $x(i) \leq x(j)$ **then** [
      $z(k) \leftarrow x(i)$
      $i \leftarrow i + 1$]
    else [
      $z(k) \leftarrow x(j)$
      $j \leftarrow j + 1$]
    $k \leftarrow k + 1$
  end
  if $i > m$ **then**
    $(z(k), z(k+1), \ldots, z(n)) \leftarrow (x(j), x(j+1), \ldots, x(n))$
  else
    $(z(k), z(k+1), \ldots, z(n)) \leftarrow (x(i), x(i+1), \ldots, x(m))$
end *merge*

Merge Sort Algorithm

The merge sort algorithm begins by treating the n items to be sorted as n sorted lists of length 1. These are merged pairwise to give $n/2$ sub-lists of length $n/2$. These $n/2$ sub-lists are then merged pairwise, and so on until we are left with only one sorted file. An example is shown in Figure 30.

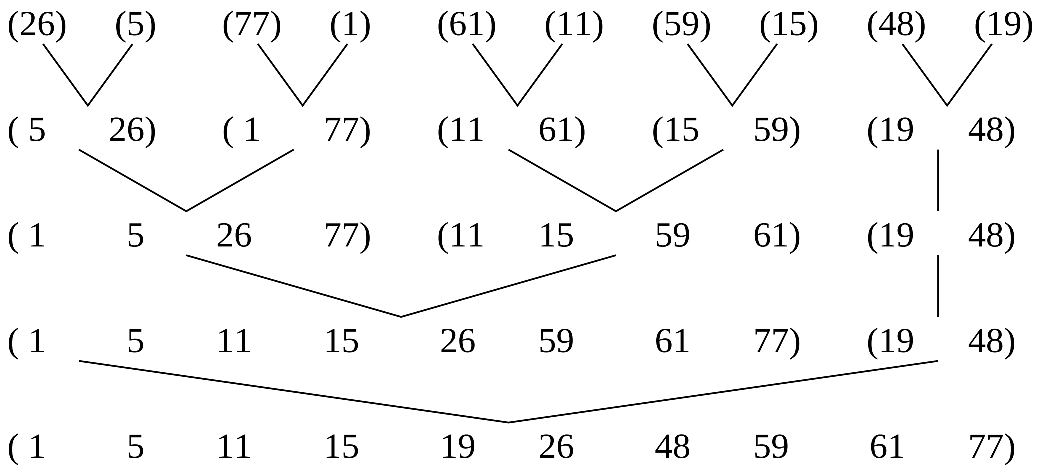


Figure 30: Example of merge sort for (26, 5, 77, 1, 61, 11, 59, 15, 48, 19).

Analysis of Merge Sort

Merge sort consists of several phases. In the first we merge files of size 1; in the second we merge files of size 2; and so on. In phase i we merge files of size 2^{i-1}. So all together there are

$$\lceil \log_2 n \rceil$$

phases. Here $\lceil x \rceil$ is the smallest integer greater than or equal to x.

Since two lists can be merged in linear time, each phase takes $O(n)$. So the total computing time is $O(n \log n)$.

Pseudocode For Merge Sort

We first look at the pseudocode for merging adjacent sub-lists of length ℓ from list x into list y. n is the number of items in x.

procedure *merge_phase (x, y, n, ℓ)*
    $i \leftarrow 1$
    while $i \leq n - 2\ell + 1$ **do**
        call *merge (x, i, i+ℓ-1,i+2ℓ-1, y)*
        $i \leftarrow i + 2\ell$
    end
    if $i + \ell - 1 < n$ **then**
        call *merge (x, i, i+ℓ-1, n, y)*
    else
        $(y(i), y(i+1), \ldots, y(n)) \leftarrow (x(i), x(i+1), \ldots, x(n))$
end *merge_phase*

Pseudocode For Merge Sort

Now we can implement the merge sort algorithm using procedure *merge_phase*. Here *auxlist* is an auxiliary list of the same length as *list*, the list to be sorted.

```
procedure mergesort (list, auxlist)
   n ← length(list)
   ℓ ← 1
   while ℓ < n do
      call merge_phase (list, auxlist, n, ℓ)
      ℓ ← 2*ℓ
      call merge_phase (auxlist, list, n, ℓ)
      ℓ ← 2*ℓ
   end
end mergesort
```

The above algorithm is stable.

Heap Sort

The merge sort algorithm has a computing time of $O(n \log n)$ in the worst and average cases. However, a disadvantage is that it requires additional storage proportional to the number of items in the list.

The heap sort algorithm also has computing time $O(n \log n)$ in the worst and average cases, but requires only a constant amount of additional storage.

In heap sort we first represent the items to be sorted with a binary tree. We place the first item at the root of the tree, and then build the tree as follows. The node containing the ith item has item $2i$ at its left child and item $2i + 1$ at its right child. With this scheme the parent of the node containing item i is item $\lceil i \rceil$. See Figure 31.

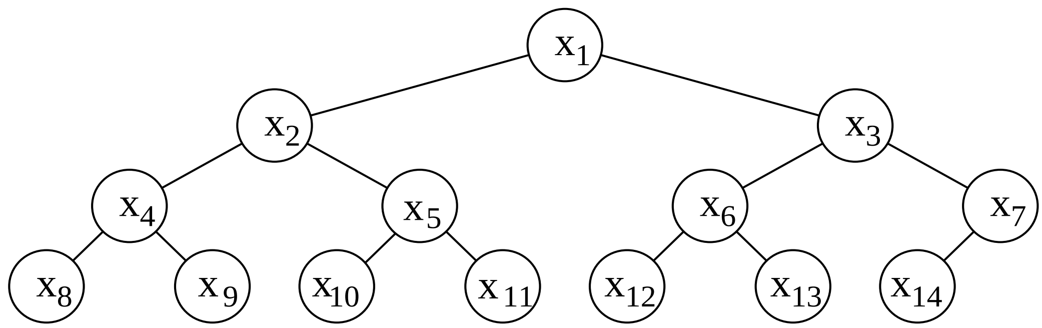


Figure 31: Representation of list by a tree.

The Heap Sort Algorithm

Heap sort has two main phase.

1. The tree representing the list is converted into a *heap*. A heap is a complete binary tree with the property that the value of each node is at least as large as the value of its child nodes (if any). This implies that the root of the heap has the largest value in the tree.

2. In the second phase the output sequence is generated in decreasing order by successively outputting the root, and restructuring the remaining tree into a heap.

An important component of the heap sort algorithm is the procedure *adjust* which takes a binary tree T whose left and right subtrees satisfy the heap property, but whose root may not, and adjusts T so that the entire binary tree satisfies the heap property. This is shown in Figure 32.

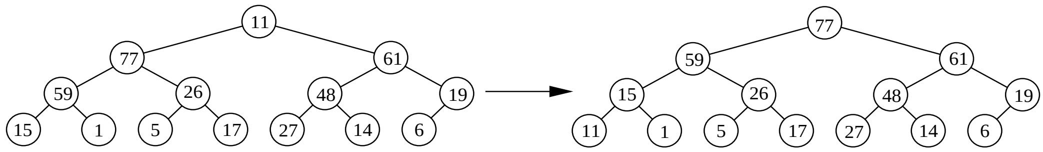


Figure 32: Example of use of procedure *adjust*.

Pseudocode For Adjust Procedure

The *adjust* procedure works as follows

- We determine the larger of the two children of the root.
- If root value is larger than the larger of its two children then we are done. Otherwise we exchange the root with the larger of its two children.
- We repeat this procedure for the subtree of the root with which a value was exchanged.

The pseudocode for the *adjust* procedure is as follows.

```
procedure adjust (list, root_index, n)
   k ← list(root_index)
   j ← 2 * root_index
   while j ≤ n do
      if j < n and list(j) < list(j+1) then j ← j + 1
      if k ≥ list(j) then exit
      list(j/2) ← list(j)
      j ← 2 * j
   end
   list(j/2) ← k
end adjust
```

Pseudocode For Heap Sort

The pseudocode for heap sort is as follows.

procedure *heapsort (list)*
  $n \leftarrow$ *length (list)*
  for $i \leftarrow n/2$ **to** *1* **by** -1 **do**
    call *adjust (list, i, n)*
  end
  for $i \leftarrow n - 1$ **to** *1* **by** -1 **do**
    call *exchange (list, i+1, 1)*
    call *adjust (list, 1, i)*
  end
end *heapsort*

- The first **for** loop re-arranges the list into a heap. It does this by examining each node with at least one child and applying the *adjust* procedure to it. Node $n/2$ is the last node with at least one child so we process this node first and then the preceding nodes.
- The second **for** loop produces the final sorted list. On the first pass we exchange the root (the largest item) with the last node, and then call *adjust* to re-arrange the first $n - 1$ nodes into a heap again. On the second pass we exchange the root (now the second largest item) with node $n - 1$ and the re-arrange the first $n - 2$ nodes into a heap again. Continuing in this ways produces the sorted list.

First Phase of Heap Sort

In the first phase of heap sort the initial list, represented with a tree is converted to a heap, as shown in Figure 33.

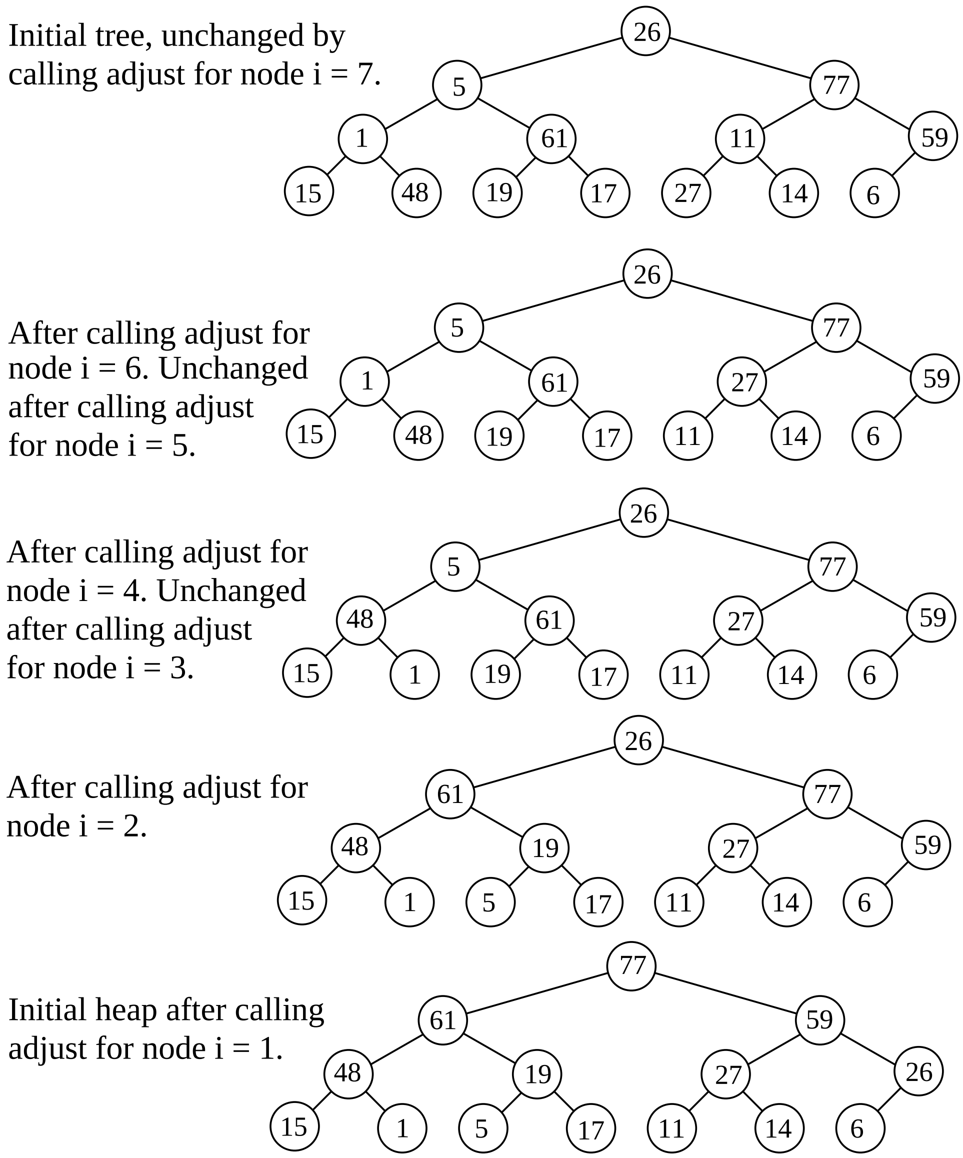


Figure 33: Transformation of initial list to a heap.

Second Phase of Heap Sort

In the second phase of heap sort the final sorted list is generated in a series of steps. In the ith of these steps, the ith largest item is placed in the correct position in the list by exchanging the root item with the item at node i. The first $n - i$ nodes of the tree are then re-arranged as a heap. Several steps in the second phase are shown in Figure 34.

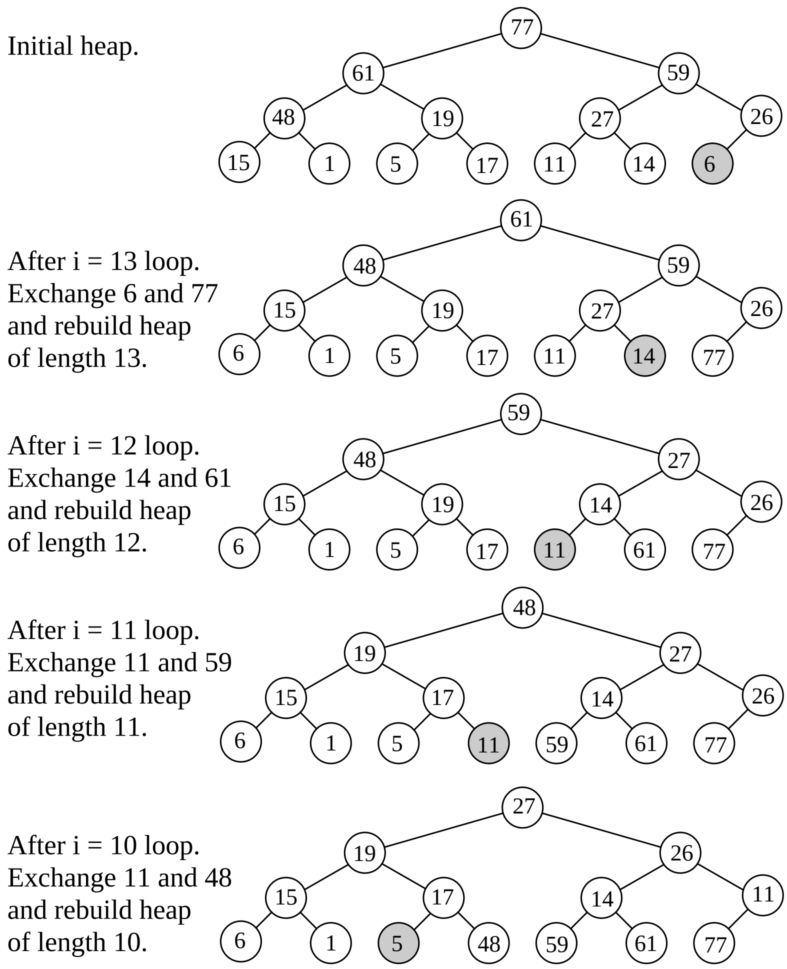


Figure 34: First few steps of second phase.

Second Phase of Heap Sort (Continued)

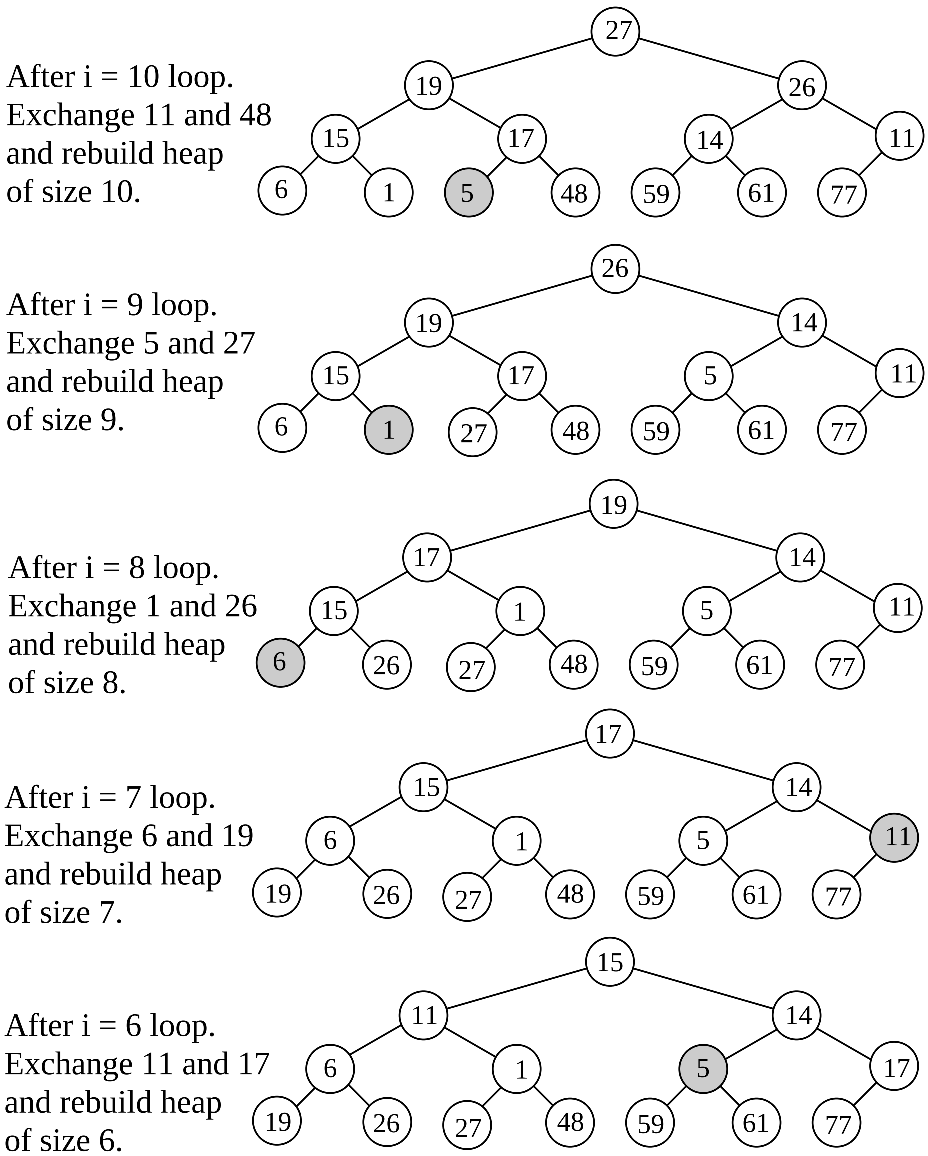


Figure 35: A few more steps of second phase.

Second Phase of Heap Sort (Continued)

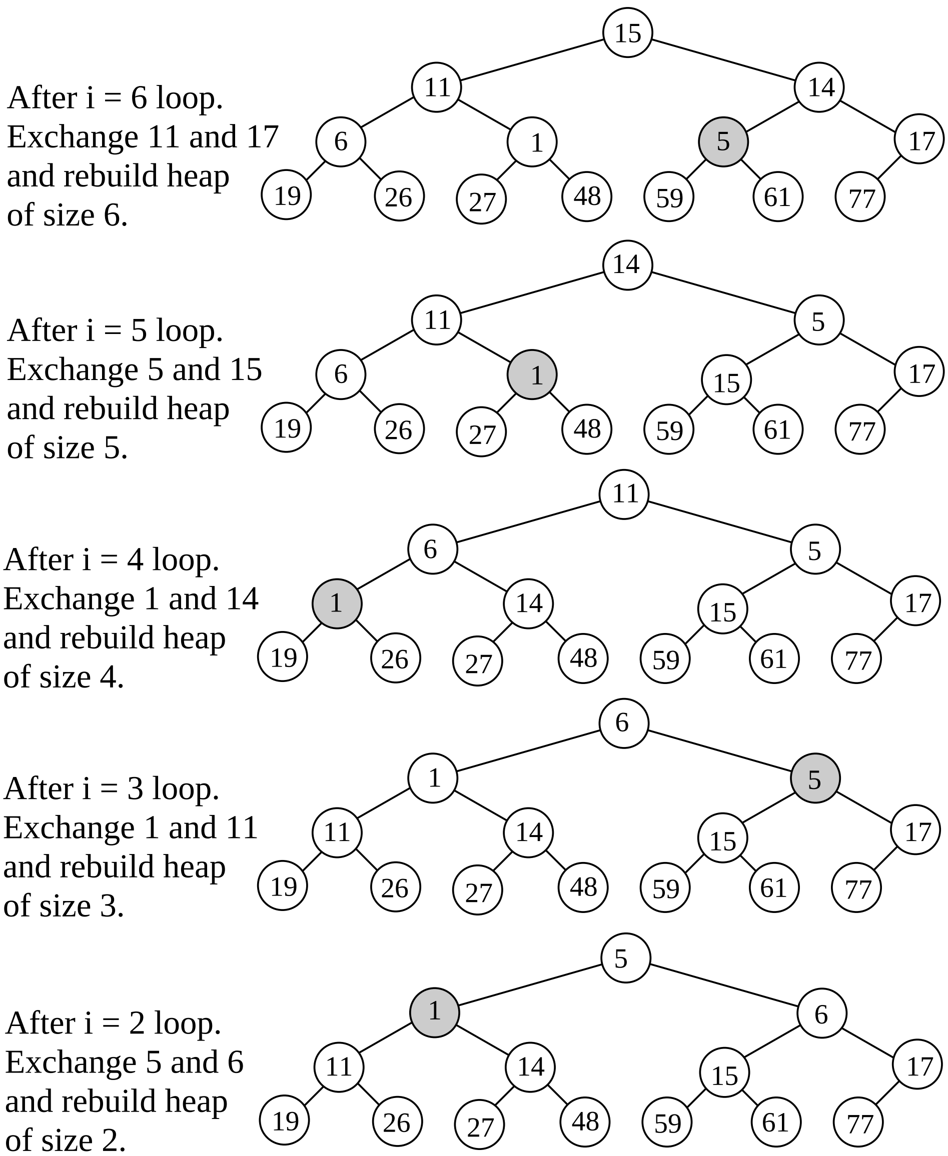


Figure 36: Yet more steps of second phase.

Second Phase of Heap Sort (Continued)

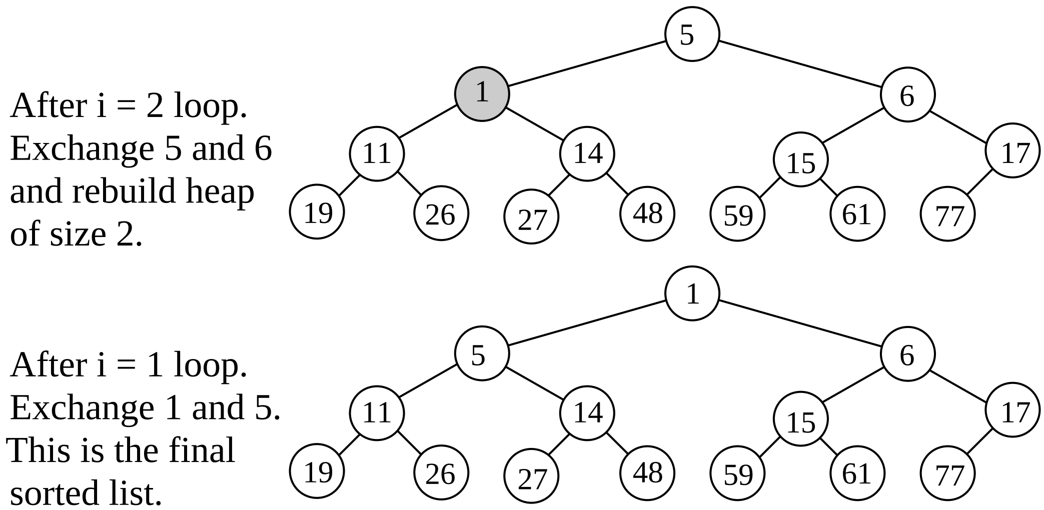


Figure 37: The final steps of second phase.

Search Methods

Searching is the retrieval of some particular piece of information from a large amount of previously stored information. It is important in a great many computational tasks, especially database applications.

The information may be thought of as being made up of records. Each record has a particular field, called the *key*, that is used in the search. We wish to find the record(s) that have a key matching some specified value. Then we usually want to examine or process the data in the complete record(s) found by the search. In many of our examples we shall search for integer key values, but it should be remembered that the key may be of any data type.

If we know information on the relative frequency with which keys are accessed we could speed up searches by placing frequently-accessed records near the start. If frequency information is not initially available then a "self-organising" search could be performed in which a record is moved to the start of the list each time it is accessed.

Search Algorithms

We have already looked at two algorithms for searching a list (see Lecture 6).

- Linear search (or sequential search) in which we examine each successive key in turn, starting with the first. This has computing time of $O(n)$.
- Binary search applied to a sorted list in which we examine the key at the middle of the list and if it is not the one being searched for we then repeat the procedure for either the sub-list to the left or right. This has computing time of $O(\log n)$.

It should be recalled that because it is expensive to insert a new record into a sorted list, the binary search algorithm is best for situations in which few insertions are done.

Binary Search Example

Suppose we want to search for the value 35 in the following list

$$(3, 19, 20, 34, 42, 50, 52, 56, 57, 80)$$

1. The middle of the list is 42. Since $35 < 42$ we must continue searching the lower half of the list.
2. The middle item of (3, 19, 20, 34) is 19. Since $35 > 19$ we must continue searching the upper part of this list.
3. The middle term of (20,34) is 20. Since $35 > 20$ we must continue searching the upper part of this list.
4. The middle term of (34) is 34. Since this is not the value we are searching for and the list is of length 1 we know the item is not in the original list.

Binary Search Pseudocode

We can express the binary search algorithm with the following iterative procedure.

procedure *binary_search (list, value)*
    low $\leftarrow 0$
    high $\leftarrow$ *length(list)* -1
    while *low* $\leq$ *high* **do**
        mid $\leftarrow$ (*low*+*high*)/2
        if *list(mid)* $<$ *value* **then**
            low $\leftarrow$ *mid* $+1$
        else if *list(mid)* $>$ *value* **then**
            high $\leftarrow$ *mid* -1
        else
            return *(mid) //found //*
    end
    return *(-1) //not found //*
end *binary_search*

Interpolation Search

When we look up somebody in the phone directory we know roughly where to look based on the first letter of their name. For example, if we are looking for John Young we would open the book near the end, and if we were looking for Jane Bell we open the book near the beginning. We might expect this to be better than the binary search algorithm.

This general idea forms the basis of the *interpolation search* method. This seeks to improve the binary search algorithm by guessing more accurately where the key being sought is in the current search interval. Binary search divides the interval at index

$$mid = lower + \frac{1}{2}(upper - lower)$$

If the keys are numerical and evenly distributed the interpolative search algorithm uses

$$ind = lower + \left(\frac{value - list(lower)}{list(upper) - list(lower)} \right) (upper - lower)$$

instead of *mid*.

The interpolation search algorithm is $O(\log \log n)$ for both successful and unsuccessful searches on lists with random keys.

Trees

We have introduced trees previously, but have not given a definition so far.

A *tree* is a collection of nodes that have the property that every node is pointed to by only one other node, called its *parent*. The other nodes that a node points to are called its *children*. A node that has no children is called a *leaf node*.

A *binary tree* is a tree in which each node has no more than two children

The length of the path from the root to a given node is called the *depth* of that node.

A *complete binary tree* is a binary tree in which all leaf nodes have the same depth, and all other nodes have exactly two children. A complete binary tree has $2^{h+1} - 1$ nodes for some positive integer $h = 0, 1, \ldots$. h is called the *height* of the tree. A complete binary tree of height $h = 3$ is shown in Figure 38.

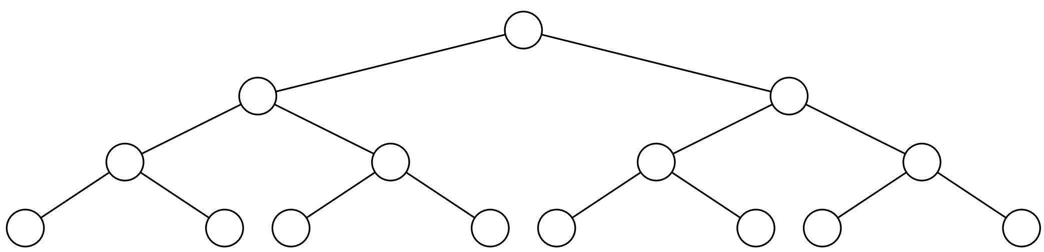


Figure 38: Example of a complete binary tree.

Binary Search Trees

In a binary tree each node may have no more than two children. If present these are called the left and right children of the node. The sub-tree of which the left child of a node is the root is called the *left sub-tree*, and the right sub-tree is similarly defined.

A *balanced tree* (or B-tree) is a binary tree in which, as nearly as possible each node has the same number of nodes in its left and right sub-trees.

With each node we associate a record with a key value. A *binary search tree* has the defining property that all records with smaller keys are in the left sub-tree and all records with larger (or equal) key values are in the right sub-tree.

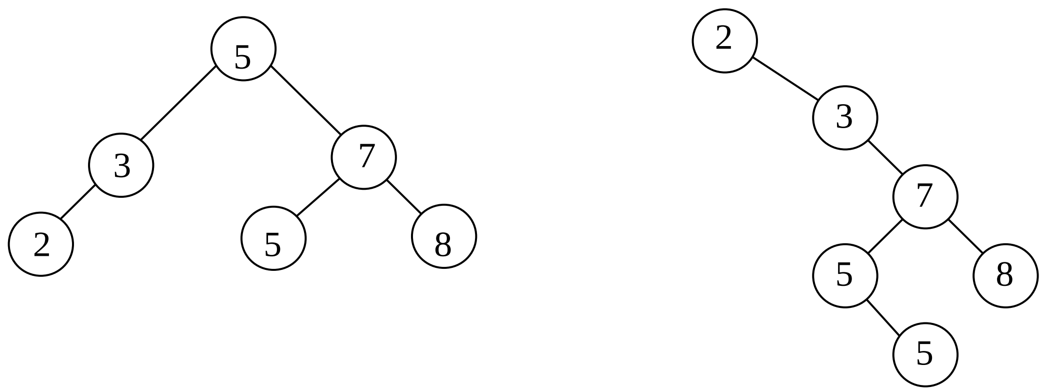


Figure 39: Examples of binary search trees for the keys (5, 3, 5, 7, 2, 8).

Structure of a Binary Search Tree

Each node, x, of a binary search tree has the following components

- The key value, *key(x)*, of the record associated with node x.
- The information, *info(x)*, contained in the record. This is the record minus the key in most cases, though in same cases it may be the entire record.
- A link, *left(x)*, to the left child of node x.
- A link, *right(x)*, to the right child of node x.
- A link, *parent(x)*, to the parent of node x.

The parent of the root node is the *head* of the binary search tree data structure. The right child of the head is the root of the tree. The left child of the head is a special node called the *tail* of the tree. For nodes with no left child the left link points to the tail node. Similarly for nodes with no right child the right link points to the tail node. The parent of the head and tail nodes is not defined.

Example of BST Data Structure

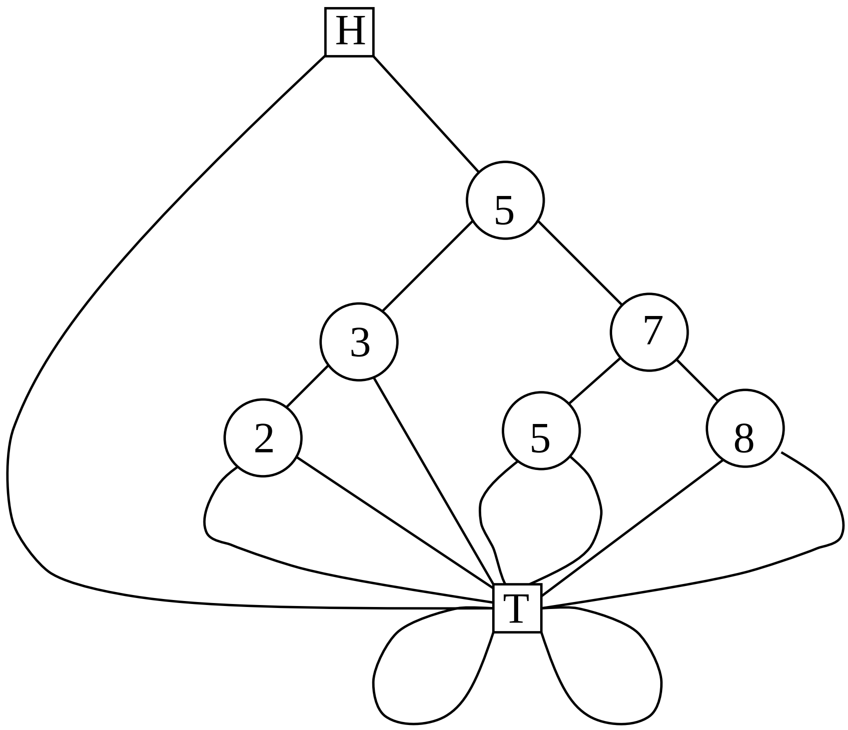


Figure 40: Example of a BST data structure.

Code for Binary Tree Search

The *binary tree search* algorithm uses a binary tree to search for a record with a given key, v.

The algorithm starts at the root and compares its key value with v. If they are equal then the search is done. If the key is less than v we go to the left child; otherwise we go to the right child. This method is applied recursively. At each stage of the algorithm the record(s) being searched for is either in the subtree of which the current node is the root, or there are no records with key v. So we either find a record with key v or we reach the tail node.

```
procedure tree_search (head, tail, v)
   x ← right(head)
   tail(key) ← v
   while v ≠ key(x) do
      if v < key(x) then
         x ← left(x)
      else
         x ← right(x)
   end
   return info(x)
end tree_search
```

Note how the tail node is used to insure the search terminates when the key is not present. The value of *info(tail)* always equals the special value *no_info*, so if this value is returned we know the search was unsuccessful.

Example of Binary Tree Search

Suppose we want to search for the key 35 from a set of records with keys 3, 19, 20, 34, 42, 50, 52, 57, 80. We construct a binary search tree and label each node by the corresponding key, as shown in Figure 41. The binary tree search algorithm follows the thick line through the tree to the tail node, indicating that the key 35 is not present.

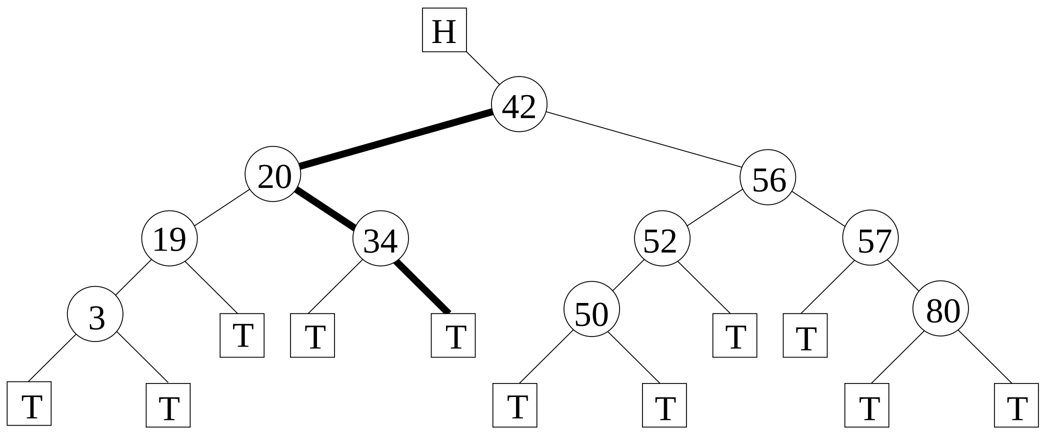


Figure 41: Example of use of binary tree search algorithm.

Important Distinctions

It is important to distinguish between

- The binary search algorithm, first described in Lecture 6. For this algorithm we can use a binary tree to describe the sequence of comparisons made.
- The binary tree search algorithm described in this lecture. In this algorithm we actually construct a data structure of records and use it in the search.

Initialising a Binary Search Tree.

In order to use the binary tree search algorithm we must maintain the binary search tree representation of the record data as records are inserted and deleted. We shall now examine how to initialise a binary search tree,

procedure *tree_initialise (head, tail)*
  tail ← *getnode ()*
  left(tail) ← *tail*
  right(tail) ← *tail*
  info(tail) ← *no_info*
  head ← *getnode ()*
  left(head) ← *tail*
  right(head) ← *tail*
  key(head) ← *nil*
end *tree_initialise*

Initially the left and right links of the head node both point to the tail node and its key has the special value *nil*. The left and right links for the tail node both point to itself, and the value of info is set to the special value *no_info*.

Insertion

Suppose we want to insert a new record into the binary search tree data structure. We have to do this in such a way that the structure of the tree is maintained.

To insert a node corresponding to the new record into the binary search tree we do an unsuccessful search for the key of the record, keeping track of the last node visited as we go. Eventually we reach the tail node. We then insert the new node to the left or right of the last node visited.

In Figure 42 we shown how a record with a key value 77 is inserted into a binary search tree.

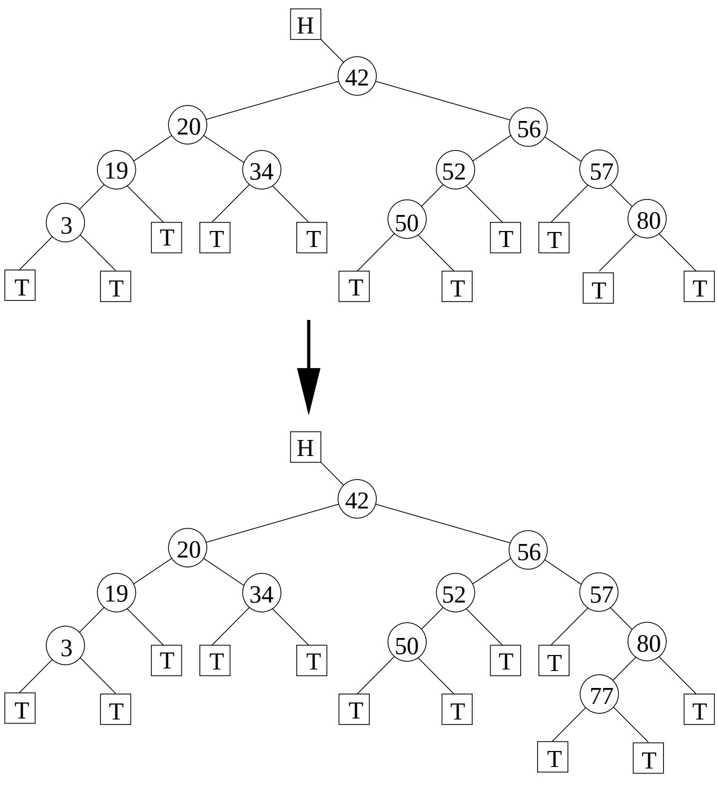


Figure 42: Insert of new node with a unique key.

Example of Insertion

In Figure 43 we show how a record with a key value 20 is inserted into a binary search tree. Note that there is already a record with the same key in the tree.

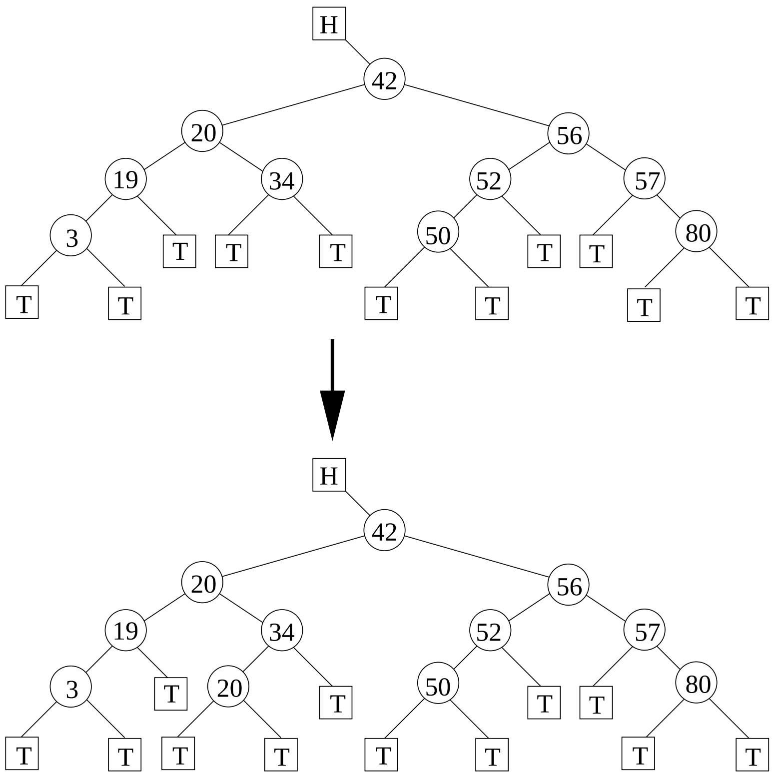


Figure 43: Insert of new node with a duplicate key.

Pseudocode for Insertion

The pseudocode for inserting a new record with key v and information i into a binary search tree is as follows.

```
procedure tree_insert (head, tail, v, i)
   p ← head
   x ← right(head)
   while x ≠ tail do
      p ← x
      if v < key(x) then
         x ← left(x)
      else
         x ← right(x)
   end
   x ← getnode()
   key(x) ← v
   info(x) ← i
   left(x) ← tail
   right(x) ← tail
   if v < key(p) then
      left(p) ← x
   else
      right(p) ← x
end tree_insert
```

Sorting Records in a BST

Sorting almost comes "for free" if the records are already arranged in a binary search tree. The procedure *tree_sort*, sorts the nodes into the array a in order of increasing key value.

procedure *tree_sort (head, tail, a)*
  $i \leftarrow 0$
  call *tree_sortr (right(head), tail, a, i)*
end *tree_sort*

procedure *tree_sortr (x, tail, a, i)*
  if $x \neq tail$ **then** [
    call *tree_sortr (left(x), tail, a, i)*
    $i \leftarrow i + 1$
    $a(i) \leftarrow x$
    call *tree_sortr (right(x), tail, a, i)*]
end *tree_sortr*

Analysis of BST Algorithms

The distance from the root to a given node of a BST is the number of links that must be traversed to get from one to the other. The sum of these distances over all nodes is known as the *internal path length* of the tree.

The time taken to search for a given key in a BST, or to insert a new node into a BST, is proportional to the pathlength from the root to the node at which the algorithm is terminated. The average case computing time for these algorithms is therefore proportional to the internal path length, I, divided by the number of nodes, n.

The value of I/n depends on how well-balanced the tree is. If random keys are inserted into the tree then the tree will be nicely balanced. In this case it can be shown that a search or insertion in a binary search tree requires about $2 \ln n$ comparisons, on the average.

However, if keys are inserted in order (or in reverse order) the binary tree search method is a linear algorithm and is no better than a linear search.

Deletion From A BST

Deletion from a binary search tree is rather more complicated than searching, sorting, or inserting. Consider the following three cases.

1. Deletion of a node with no children, such 12 or 16 in Figure 44. This is straightforward – just make the corresponding link in the parent point to the tail.

2. Deletion of a node with only one child, such as 1, 8, or 18. In this case we replace the deleted node by its child.

3. Deletion of a node with two children, at least one of which has no children, such as 14. In this case we replace the deleted node with the child that has no children.

Deletion of nodes higher up the tree, such as 5, requires more thought.

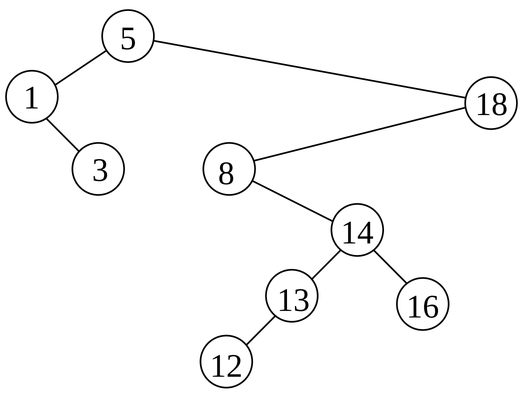


Figure 44: Deletion from a binary search tree.

Algorithm for Deletion

To delete a node, such as 5 in Figure 44, from a binary search tree we replace it with the node with the next highest key, as shown in Figure 45.

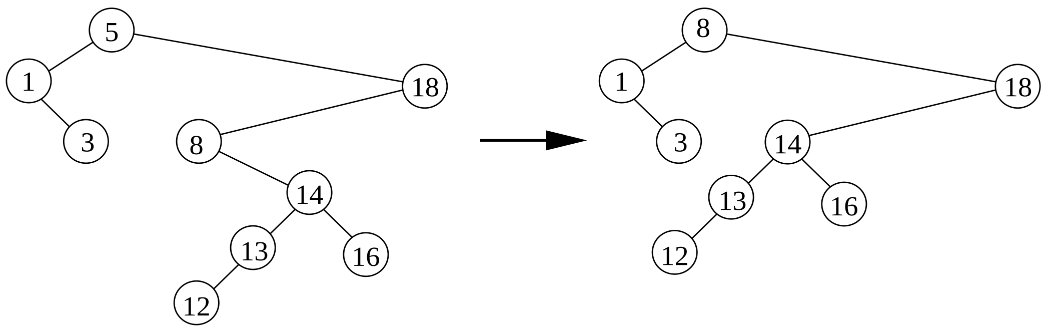


Figure 45: Deletion of node 5 from a binary search tree.

Pseudocode for Deletion

The pseudocode for the deletion algorithm is as follows.

```
procedure tree_delete (head, tail, v)
   key(tail) ← v
   p ← head
   x ← right(head)
   while v ≠ key(x) do
      p ← x
      if v < key(x) then
         x ← left(x)
      else
         x ← right(x)
   end
   t ← x
   if right(t) = tail then
      x ← left(x)
   else if left(right(t)) = tail then [
      x ← right(x); left(x) ← left(t) ]
   else [
      c ← right(x)
      while left(left(c)) ≠ tail do
         c ← left(c)
      end
      x ← left(c); left(c) ← right(x)
      left(x) ← left(t); right(x) ← right(t) ]
   call ret(t)
   if v < key(p) then
      left(p) ← x
   else
      right(p) ← x
end tree_delete
```

Explanation of Delete Pseudocode

The deletion algorithm first searches for the node to be deleted and sets its parent variable p.

A conditional then deals with the following three cases.

1. If the node to be deleted has no right child it is replaced by its left child.
2. If the right child of the node to be deleted has no left child then this left child becomes the left child of the deleted node, and the deleted node is replaced by its right child.
3. In all other cases we look for the successor node with the next largest key . This is stored in variable $left(c)$. The node to be deleted, x, is replaced by its successor.

Searching for Strings

The binary tree search algorithm is useful for searching for information that is identifiable by a key. A key is a small, clearly-defined chunk of information and its use is well-suited to record processing and the manipulation of databases.

We will now consider how to search for information in the form of a *string*. A string is a linear sequence of characters. Strings are particularly important in word-processing systems. A *text string* consists of letters, numbers, and certain special characters such as punctuation marks. A *binary string* is a sequence of 0's and 1's.

In some sense text and binary strings are equivalent since we can convert a text string to a binary string by representing each character by 8 bits, and we can convert a binary string to a text string by chopping it into groups of 8 bits and replacing each group of bits by the corresponding character.

We shall consider pattern matching algorithms. We want to take a given text string of length N and a pattern of length M and look for occurrences of the pattern in the text.

Brute Force Algorithm

The brute force approach examines every possible position in the text at which a match could occur and checks whether the pattern does match.

If s is a string we shall denote character number i in the string by $s(i)$, and will start numbering the characters from 1. Thus, a string of length N contains the characters $s(1)$, $s(2)$,..., $s(N)$. We shall also assume the existence of a function *strlen(s)* that returns the length of the string s.

For each position i in the string we line up the pattern so that *text(i)* lines up with *pattern(1)*, *text(i+1)* lines up with *pattern(2)*, and so on. We then check whether corresponding characters in the text and pattern are equal. As soon as we come to a case of inequality we know that the pattern cannot match the text starting at position i. We can then increment i by 1, and compare the text and pattern starting at the new value of i.

In the worst case, the first $M - 1$ characters in the pattern match the portion of the text being examined. So for each of the N positions in the text we make M comparisons (except near the end of the text where the pattern "sticks out" beyond the text). Thus, the number of comparisons for the brute force algorithm is about MN.

Example of Brute Force Algorithm

In Figure 46 we show an example of the brute force algorithm applied to a binary string. The pattern in this case is 10100111. The first character in the pattern that fails to match the portion of the string being examined is shown shaded.

Figure 46: Example of brute force algorithm for pattern matching.

KMP Algorithm

Knuth, Morris, and Pratt devised an algorithm for pattern matching in strings which has a computing time of $O(N + M)$. This is a significant improvement over the $O(NM)$ brute force algorithm.

The KMP algorithm avoids having to rescan the text string after an attempted match has failed. Consider the following pattern,

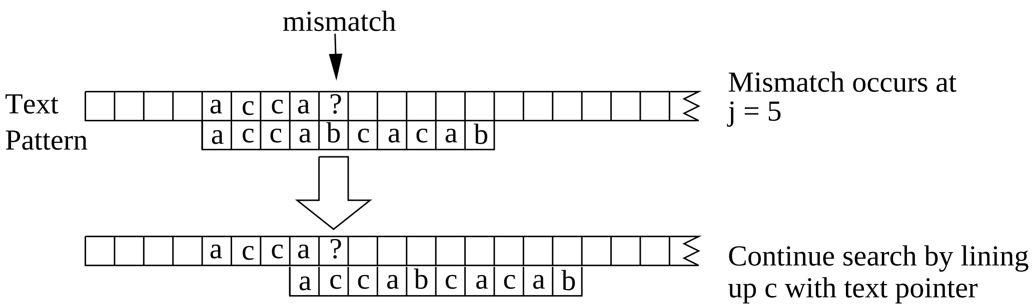
$$a \; c \; c \; a \; b \; c \; a \; c \; a \; b$$

Let $S = s_1 s_2 s_3 \ldots s_N$ be the text string and suppose we are looking for a match starting at s_i.

- If $s_i \neq a$ the match fails and we continue by next comparing s_{i+1} with a.
- If $s_i = a$ but $s_{i+1} \neq c$, the match fails and we continue by next comparing s_{i+1} with a.
- If $s_i s_{i+1} = ac$ and $s_{i+2} \neq c$ the match fails. The brute force algorithm would then compare s_{i+1} with a. However, we already know that this will produce a failed match since we know that $s_{i+1} = c$. Thus, the KMP algorithm avoids backing up in the text string and next compares s_{i+2} with a.

KMP Algorithm (Continued)

- If we have a match for the first 4 characters of the pattern and a non-match for the fifth, then $s_i s_{i+1} s_{i+2} s_{i+3} = acca$ and $s_{i+4} \neq b$. The KMP algorithm next compares s_{i+4} with the second character of the pattern, i.e., c. We do this because we already know that s_{i+3} matches the first letter a of the pattern, and that s_{i+4} must be either a or c.



Failure Function

The key point of the KMP algorithm is that by knowing the characters in the pattern and the point where a mis-match occurred it is possible to determine where to continue the search from without backing up in the text string.

Determining where to continue the search from depends only on the pattern. Suppose we are comparing the pattern with the portion of the text string starting at character i. For a given pattern, if a mis-match occurs between character j in the pattern and character $i + j - 1$ in the text (with the first $j - 1$ characters all matching) then it is possible to compute the location, *next(j)*, in the pattern that we should next compare with character $i + j - 1$ to continue the search for a match. The array *next* can be pre-computed for a given pattern and used in a subsequent search.

To help compute the *next array* we introduce the failure function, f. If the pattern is $P = p_1 p_2 \ldots p_M$ then f is defined as

$$f(j) = \text{largest } i < j \text{ such that } p_1 p_2 \ldots p_i = p_{j-i+1} p_{j-i+2} \ldots p_j$$

if such an $i \geq 1$ exists, and 0 otherwise.

Example of Failure Function

The failure function, $f(j)$, is the number of characters from the start of the pattern that match that part of the pattern ending at character j. $f(1)$ is always zero. Thus we seek the largest i (with $1 \leq i < j$) such that

$$p_1 p_2 \ldots p_i = p_{j-i+1} p_{j-i+2} \ldots p_j$$

for a given value of j.

Consider the pattern

a b c a b c a c a b

$j = 1$: $f(1) = 0$.

$j = 2$: $f(2) = 0$ because $p_1 \neq p_2$.

$j = 3$: $f(3) = 0$ because $p_1 \neq p_3$.

$j = 4$: $f(4) = 1$ because $p_1 = p_4$.

$j = 5$: $f(5) = 2$ because $p_1 p_2 = p_4 p_5$.

$j = 6$: $f(6) = 3$ because $p_1 p_2 p_3 = p_4 p_5 p_6$.

$j = 7$: $f(7) = 4$ because $p_1 p_2 p_3 p_4 = p_4 p_5 p_6 p_7$.

$j = 8$: $f(8) = 0$ because $p_1 \neq p_8$.

$j = 9$: $f(9) = 1$ because $p_1 = p_9$.

$j = 10$: $f(10) = 2$ because $p_1 p_2 = p_9 p_{10}$.

The *next* Array

Suppose we are comparing the pattern with the part of the text string starting at character number i. Assume the first $j-1$ characters of the pattern match *text(i),...,text(i+j-2)* but there is a mismatch between *text(i+j-1)* and *pattern(j)*.

This situation is illustrated in Figure 47, where we also show shaded the $f(j-1)$ characters at the start of the pattern that match the $f(j-1)$ characters that precede the jth character of the pattern.

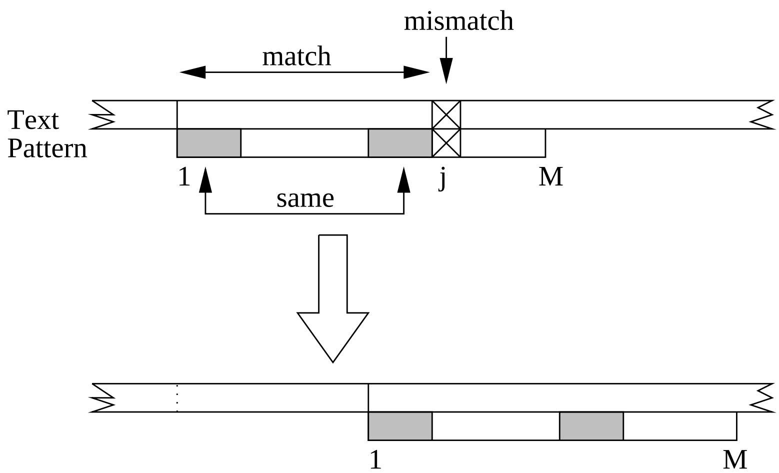


Figure 47: How the *next* array is computed.

Figure 47 also shows how we continue processing after the mismatch, by shifting the pattern along so that character number $f(j-1)+1$ of the pattern lines up with *text(i+j-1)*.

The *next* Array

The *next* array is defined as follows.

$$next(j) = \begin{cases} 0 & \text{if } j = 1 \\ f(j-1)+1 & \text{if } j > 1 \end{cases}$$

In the KMP algorithm, when a mismatch occurs and *next(j)* is not zero we continue processing by replacing j with *next(j)*, leaving i, the index into the text string, unchanged. If *next(j)* is zero we increment i by 1 and set j to 1, i.e., the text and pattern are aligned at the character after where the mismatch occurred.

KMP Pseudocode

The pseudocode for the KMP algorithm is as follows.

procedure *kmp_string_search (pattern, text)*
  $i \leftarrow 1$
  $j \leftarrow 1$
  $M \leftarrow$ *strlen(pattern)*
  $N \leftarrow$ *strlen(text)*
  while $i \leq N$ **and** $j \leq M$ **do**
    if *text(i)* $=$ *pattern(j)* **then** [
      $i \leftarrow i + 1$
      $j \leftarrow j + 1$]
    else [
      $j \leftarrow$ *next(j)*
      if $j = 0$ **then** [
        $j \leftarrow 1$
        $i \leftarrow i + 1$]
    ]
  end
  if $j > M$ **then return** *(i-M-1)*
  else return *(not_found)*
end *kmp_string_search*

Example of the KMP Algorithm

Figure 48 shows an example of the KMP algorithm for the pattern

a b c a b c a c a b

f	=	0	0	0	1	2	3	4	0	1	2
next	=	0	1	1	1	2	3	4	5	1	2

	b	a	b	b	a	b	c	a	b	c	c	a	b	c	a	b	c	a	c	a	b
0	$\boxed{a}$	b	c	a	b	c	a	c	a	b											
1		a	b	$\boxed{c}$	a	b	c	a	c	a	b										
0				$\boxed{a}$	b	c	a	b	c	a	c	a	b								
4					a	b	c	a	b	c	$\boxed{a}$	c	a	b							
1								a	b	c	$\boxed{a}$	b	c	a	c	a	b				
0											$\boxed{a}$	b	c	a	b	c	a	c	a	b	
												a	b	c	a	b	c	a	c	a	b

Figure 48: Example of KMP algorithm.

Alternative View of KMP Algorithm

If the pattern is fixed we can hard-code it into the KMP algorithm. For example, for the pattern *abcabcacab* we have

procedure *kmp_fixed (text)*
        $i \leftarrow 0$
sm: $i \leftarrow i + 1$
s1: **if** *text(i)* $\neq$ *'a'* **then goto** *sm* **else** $i \leftarrow i + 1$
s2: **if** *text(i)* $\neq$ *'b'* **then goto** *s1* **else** $i \leftarrow i + 1$
s3: **if** *text(i)* $\neq$ *'c'* **then goto** *s1* **else** $i \leftarrow i + 1$
s4: **if** *text(i)* $\neq$ *'a'* **then goto** *s1* **else** $i \leftarrow i + 1$
s5: **if** *text(i)* $\neq$ *'b'* **then goto** *s2* **else** $i \leftarrow i + 1$
s6: **if** *text(i)* $\neq$ *'c'* **then goto** *s3* **else** $i \leftarrow i + 1$
s7: **if** *text(i)* $\neq$ *'a'* **then goto** *s4* **else** $i \leftarrow i + 1$
s8: **if** *text(i)* $\neq$ *'c'* **then goto** *s5* **else** $i \leftarrow i + 1$
s9: **if** *text(i)* $\neq$ *'a'* **then goto** *s1* **else** $i \leftarrow i + 1$
s10: **if** *text(i)* $\neq$ *'b'* **then goto** *s2* **else** $i \leftarrow i + 1$
        return *(i-10)*
end *kmp_fixed*

We assume we have added the pattern onto the end of the text string to ensure it will terminate successfully. This avoids having to check if $i = N$ every time i is incremented. If the value returned is greater than the number of characters in the text string then we know there was no match. The **goto** labels correspond exactly with the entries in the *next* array. Given a pattern we can produce a very efficient program to scan for it in a text string.

Finite State Machine

The procedure *kmp_fixed* can be described in terms of a simple machine model known as a *finite state machine* (FSM). The FSM is shown in Figure 49.

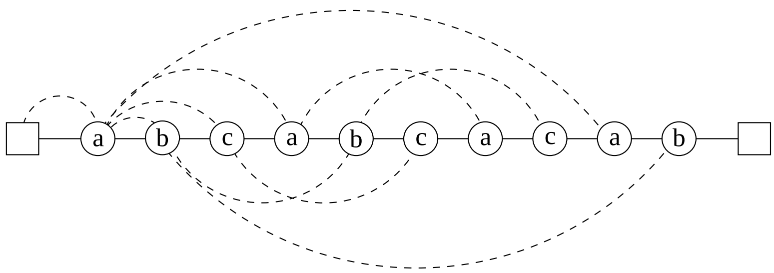


Figure 49: A finite state machine for string searching.

A FSM consists of

- states (circles in Figure 49) and
- transitions (lines in Figure 49). Each state has two transitions associated with it
  - A match transition represented by a solid line going right
  - A non-match transition represented by a dashed line going left.

When in the state labelled "x" the FSM can execute only the following one instruction.

> *If the current character is x take the match transition, otherwise take the non-match transition.*

An Improved FSM

When a non-match transition leads from one state to another state with the same label, this corresponds to searching for a match starting at the character labelling the state transitioned to. This obviously cannot produce a match since we already know that that character is a non-match. So the FSM for a particular pattern can be improved by checking if the character of the state moved to is the same as that of the state moved from. In this case we just take the non-match transition of the state moved to.

This would produce the improved FSM shown in Figure 50.

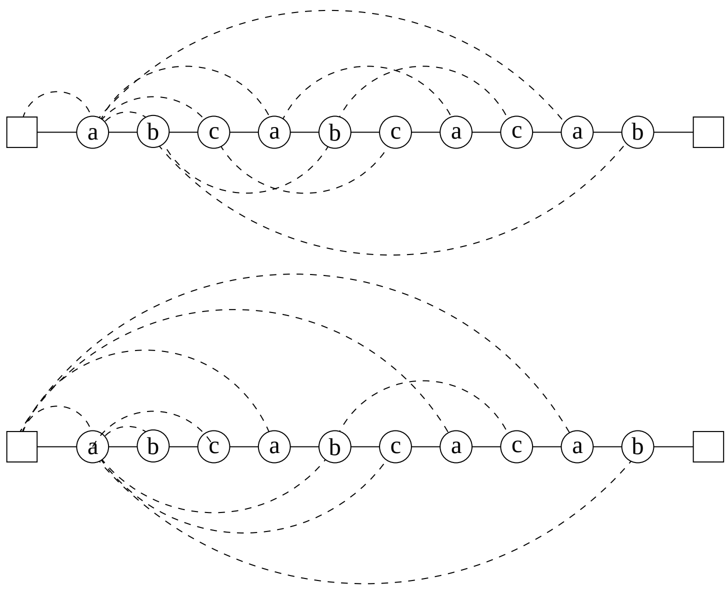


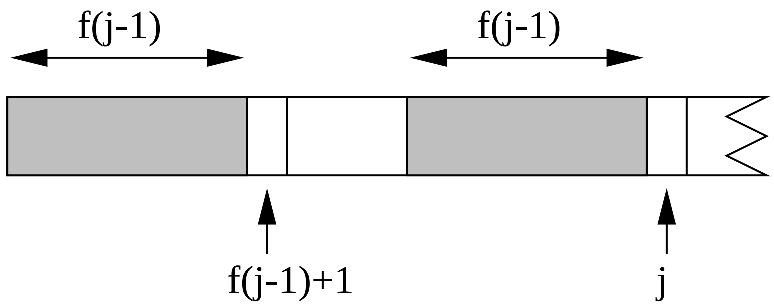
Figure 50: Improved finite state machine for string searching.

The improved FSM corresponds to the following *next* array:

0 1 1 0 1 1 0 5 0 1

Evaluation of the Failure Function

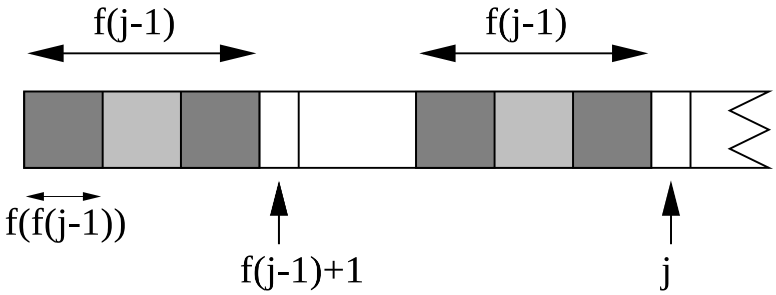
Suppose we have a pattern whose ith character is denoted by $p(i)$, $i = 1, 2, \ldots, M$. Suppose also we have already found $f(1), f(2), \ldots, f(j-1)$ for some j and want to next find $f(j)$. Remember that $f(1) = 0$.



Clearly, if $p(j) = p(f(j-1)+1)$ then $f(j) = f(j-1)+1$.

Evaluation of the Failure Function

If $p(j) \neq p(f(j-1)+1)$ we must examine the first $f(j-1)$ characters of the pattern. We know that the first $f(f(j-1))$ characters of this part of the pattern are the same as the last $f(f(j-1))$ characters. If we also have $p(j) = p(f(f(j-1)))$ then $f(j) = f(f(j-1)) + 1$.



Let's write $f(f(j-1))$ as $f^2(j-1)$, $f(f(f(j-1)))$ as $f^3(j-1)$, and so on.
Continuing in this way we get the following definition of the failure function,

$$f(j) = \begin{cases} 0 & \text{if } j = 1 \\ f^m(j-1)+1 & \text{where } m \text{ is least } k \text{ such that} \\ & p(j) = p(f^k(j-1)+1) \\ 0 & \text{if there is no such } k \end{cases}$$

Pseudocode for Failure Function

The pseudocode for evaluating the failure function can now be written as follows.

procedure *failure_function (pattern, f)*
  $f(1) \leftarrow 0$
  for $j \leftarrow 2$ **to** *strlen(pattern)* **do**
    $i \leftarrow f(j-1)$
    while *pattern(j)* $\neq$ *pattern(i+1)* **and** $i > 0$ **do**
      $i \leftarrow f(i)$
    end
    if *pattern(j)* $=$ *pattern(i+1)* **then** $f(j) \leftarrow i + 1$
    else $f(j) \leftarrow 0$
  end
end *failure_function*

Once the failure function is known it is simple to construct the *next* array.

Relaxed Searches

In some situations we may want to perform a search on the set of records in a database when there may be errors in the keys. This is often applicable when the keys are text strings such as peoples' names. Errors may arise through spelling mistakes, "typos", or through mis-hearing information given verbally.

In such cases we want to relax the criteria for the search so that instead of returning only exact matches for a given search, records with similar keys in some sense are returned.

One way to do this is through pattern matching techniques. Usually a question mark ('?') means accept any single character in this position,and an asterisk ('*') means accept any zero or more characters at this position.

So for example if the search key is "B?ggs" records with keys Baggs, Beggs, Biggs, Boggs, and Buggs might be returned. For a search key of "B*ggs" the same records might be returned, plus those with keys Briggs, Braggs, Boriggs, etc.

Soundex Codes

Soundex is intended to transform all words that sound similar into the same code. When searching for a specified character-based key (such as a surname) the key is reduced to its Soundex code and all records whose keys have the same Soundex code are returned. Typically a person then examines the records extracted to determine which are the desired ones.

The Soundex encoding algorithm works as follows.

1. Retain the first character of the name and drop all subsequent occurrences of a, e, h, i, o, u, w, y.
2. Assign the following numbers to the remaining letters after the first.

b, f, p, v, $\rightarrow$ 1	l $\rightarrow$ 4
c, g, j, k, q, s, x, z $\rightarrow$ 2	m, n $\rightarrow$ 5
d, t, $\rightarrow$ 3	r $\rightarrow$ 6

3. If two or more letters with the same code were adjacent in the original name drop all but the first.
4. Convert to the form "letter, digit, digit, digit" by adding trailing zeros (if there are less than 3 digits), or by dropping rightmost digits (if there are more than three).

Examples of Soundex Codes

1. Consider the names *Smith* and *Smythe*. After the first step of the Soundex algorithm these both reduce to *Smt*, hence the Soundex code is *S530*.

2. Consider the names *Welch* and *Walsh*. After the first step of the Soundex algorithm these reduce to *Wlc* and *Wls*, respectively, hence the Soundex code for both is *W420*.

3. Consider the names *Bocksman* and *Boxman*. After the first step of the Soundex algorithm the first reduces to *Bcksmn* which encode as *B22255*. Because the "cks" are adjacent in the original name and encode to the same number (2) we keep only one 2. The "m" and "n" also encode to the same number (5), but were not adjacent in the original name, hence we keep both fives. Hence the Soundex encoding of *Bocksman* is *B255*. For *Boxman* we first reduce it to *Bxmn* and hence got the Soundex code *B255*. Thus the encodings for *Bocksman* and *Boxman* are the same.

4. Consider *Cyncoed* and *Cinkoid*. After the first step of the Soundex algorithm these reduce to *Cncd* and *Cnkd*, respectively. Hence the Soundex code for both is *C523*.

Shortcomings of Soundex

Soundex codes are not foolproof. Sometimes names that sound distinct get the same code. For example, *Santa* and *Smythe* both have Soundex code *S530*.

In addition, related names might have different Soundex codes. For example, *Rogers* (*R262*) and *Rodgers* (*R326*).

To check a Soundex code visit the following WWW site:

`http://orwant.www.media.mit.edu/soundex.html`

The Recall-Precision Curve

We can characterise a search algorithm in terms of its *recall-precision curve.* Here *recall* means the number of items returned by a search. This is often called the number of "hits." The *precision* is the ratio of the number of useful hits to the total number of hits.

The recall-precision curve is a plot of the relationship between recall (on the y axis) and precision (on the x axis) for a search on a given key. For an algorithm that matches exactly we would expect a high precision and a fairly low recall. As we relax the search criteria, we expect the precision to go down and the recall to go up. A example recall-precision curve is shown in Figure 51.

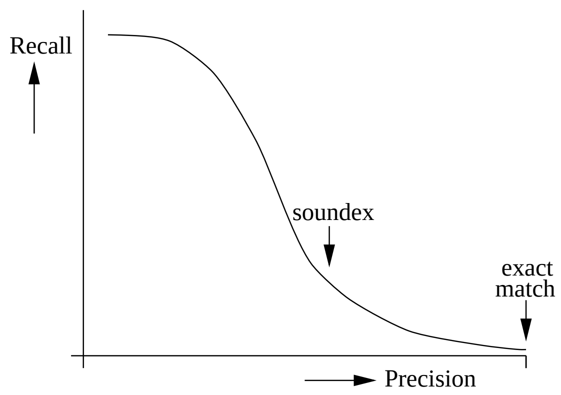


Figure 51: Example recall-precision curve.

If we select an algorithm that takes us to far up the recall-precision curve we get swamped by useless hits. If we choose an algorithm that takes us too far down it we may not retrieve the information we want. The best search algorithm to use depends on how critical it is to retrieve the data we want (and on how patient we are).