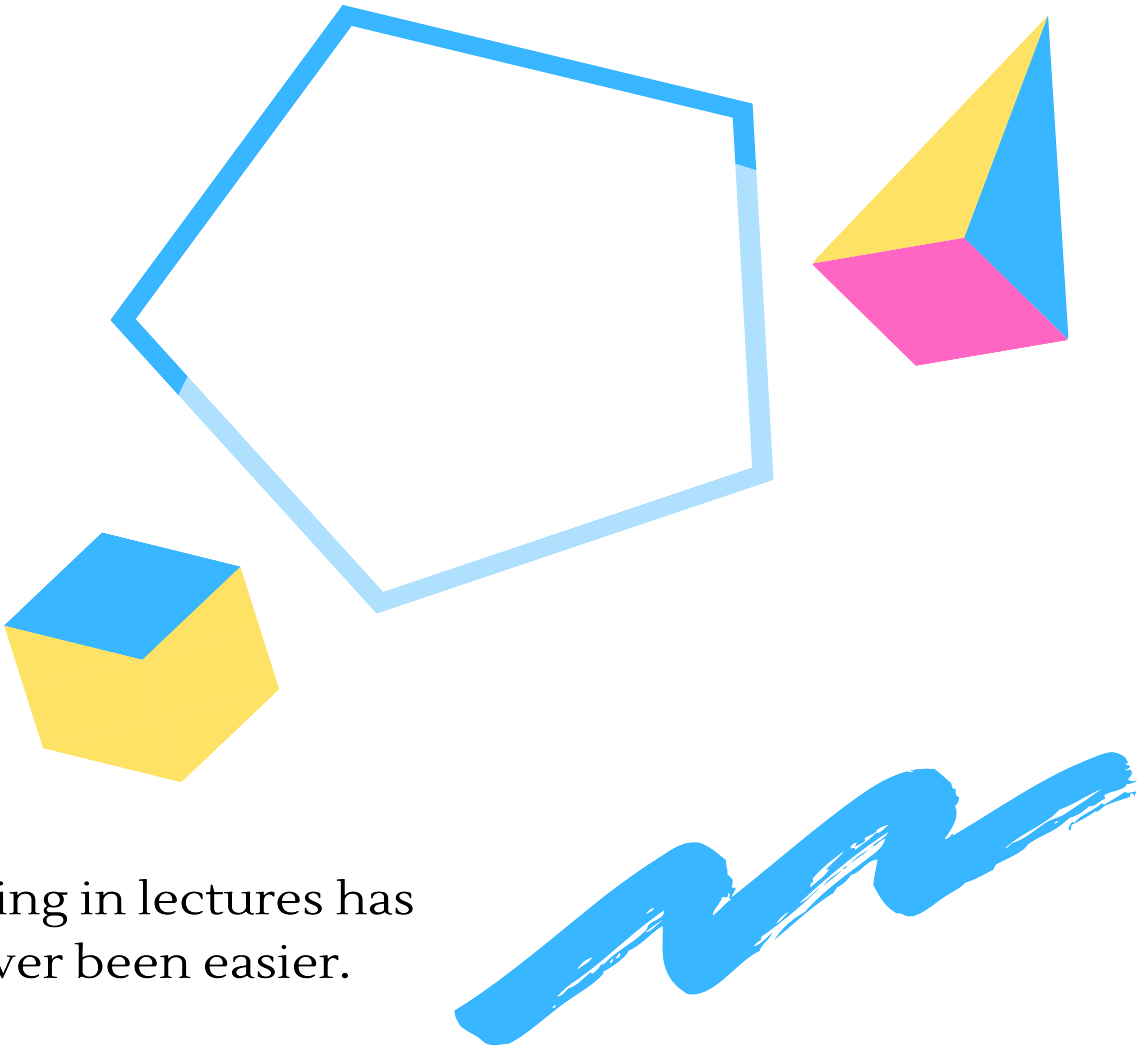
ALGORITHM DESGIN & ANALYSIS

PROBLEM BASED LEARNING



Sleeping in lectures has never been easier.

COMPUTER SCIENCE STUDENT

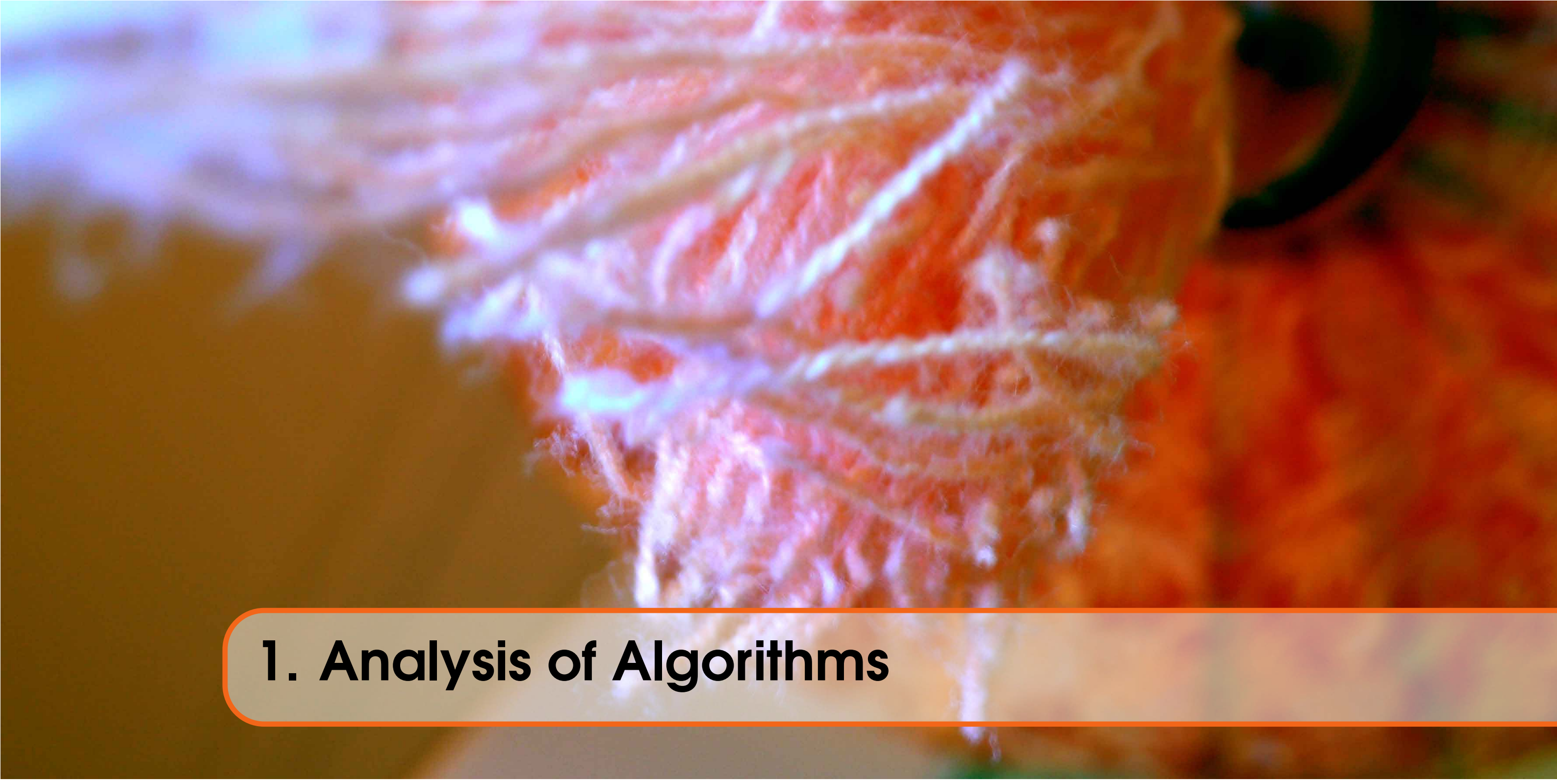
A collection of solved problems

RAMIRO GONZALEZ

First printing, March 2019

Contents

Part One



1. Analysis of Algorithms

1.1 Basic

1.1.1 Insertion Sort

■ **Example 1.1** Given an input array $A = \langle 5,8,4,2,3,1 \rangle$ use insertion sort algorithm to show how the array looks before each iteration of the for loop in line 1. ■

Listing 1.1: Insertion-Sort(A)

```
1  for $j = 2$ to A.length
2      key = $A[j]$
3      //Insert A[j] into the sorted sequence A[1..j − 1]
4      i = $j - 1$
5      while $i > 0$ and $A[i] >$ key
6          $A[i+1] = A[i]$
7          $i = i - 1$
8      $A[i+1] = key$
```

Definition 1.1.1 **Iteration:** A process wherein a set of instructions or structures are repeated in a sequence a specified number of times or until a condition is met.

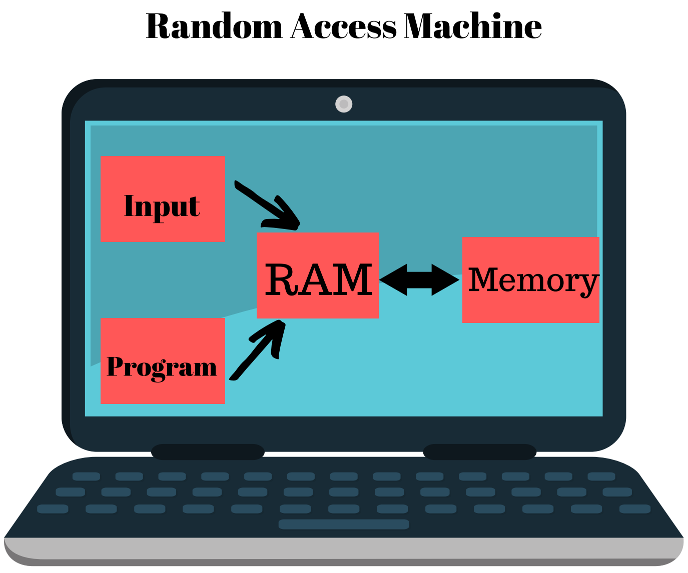
Definition 1.1.2 **Insertion Sort** An efficient sorting algorithm for small number of elements that builds a final sorted list one item at a time, with the movement of higher-ranked elements.

Solution:

1	j = 2	$A =< 5,8,4,2,3,1 >$
2	j = 3	$A =< 4,5,8,2,3,1 >$
3	j = 4	$A =< 2,4,5,8,3,1 >$
4	j = 5	$A =< 2,3,4,5,8,1 >$
5	j = 6	$A =< 1,2,3,4,5,8 >$

1.1.2 RAM - Random Machine Model

■ **Example 1.2** Briefly explain the RAM (Random Access Machine) computational model ■



Definition 1.1.3 Basic operations such as addition, multiplication, load, store, copy, control, initialization are assumed to take a constant amount of time each (if numbers are big, this may not be true, but we assume that this is the case unless stated otherwise).

R Not to be confused with **Random Access Memory**

Solution:

- The RAM (Random Access Machine) model is used for analyzing algorithms where we assume that instructions are executed one after the other. It is used to model how real world programs would run.
- Machine independent and theoretical.
- Simple operations such as $(+, \times, -, =)$ take $O(1)$ time.
- Loops and subroutines are not considered simple operations. Instead, they are the composition of many single-step operations

1.1.3 Merge Sort

■ **Example 1.3** Show the operation of Merge sort on the array $A =< 7,4,2,8,3,1,5,6,9 >$ ■

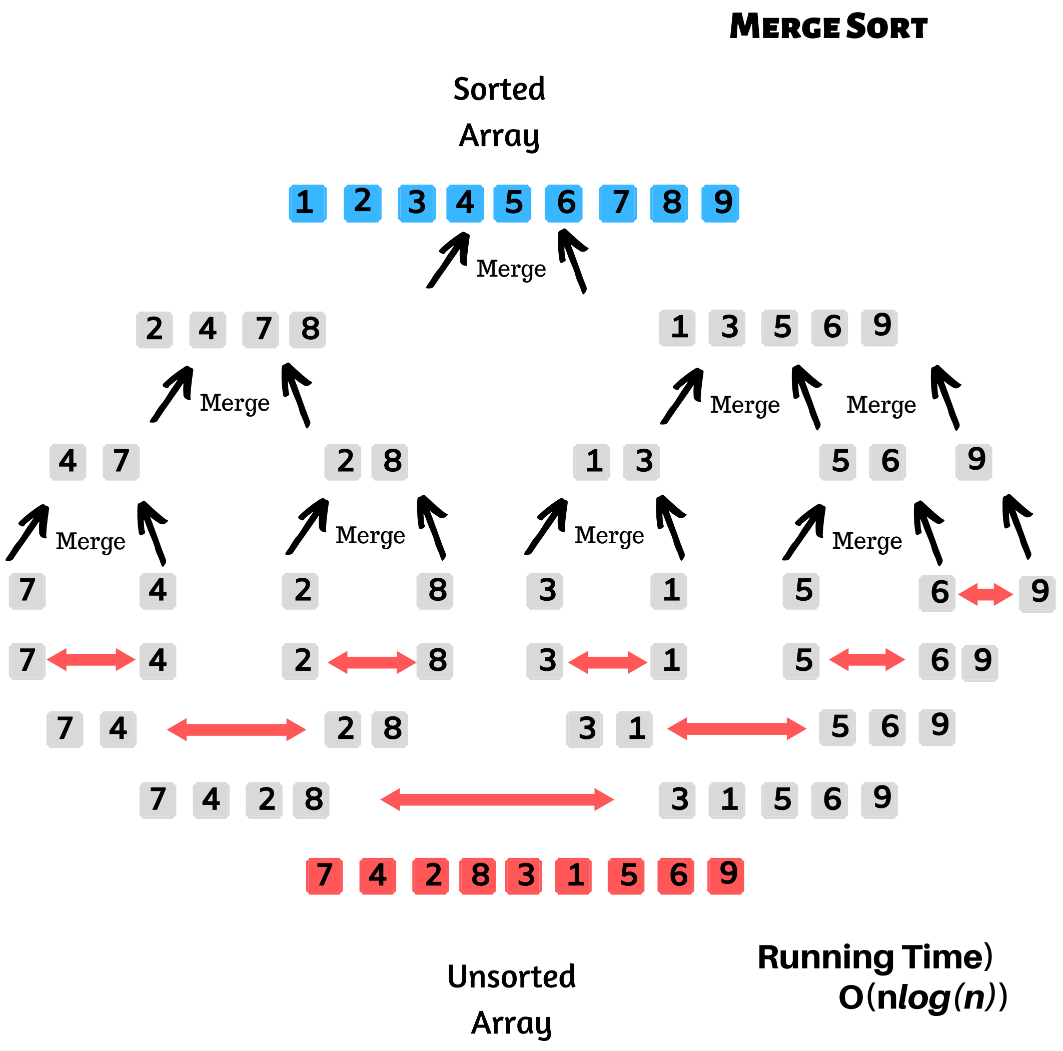
Definition 1.1.4 **Merge Sort** A divide and conquer algorithm. Takes an input array and divides it into two halves, calls itself for the two halves, and merges the two sorted halves.

- **merge()** function merges two halves.
- **MergeSort()** function recursively calls itself to divide the array until its size becomes one.

Listing 1.2: Merge-Sort(A,p,r)

```
if p < r
    q = $\lfloor (p+r)/2 \rfloor$
    Merge-Sort(A,p,q)
    Merge-Sort(A,q+1,r)
    Merge(A,p,q,r)
```

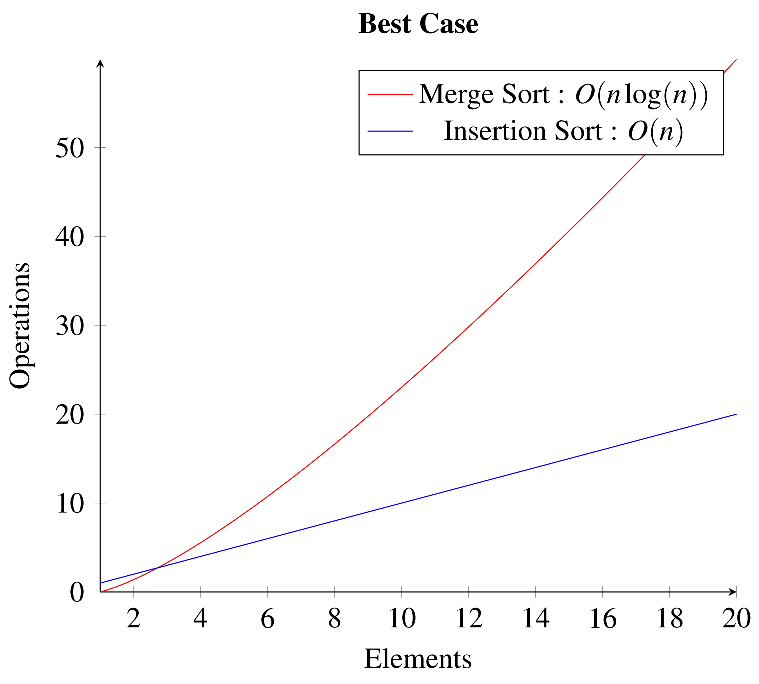
Solution:

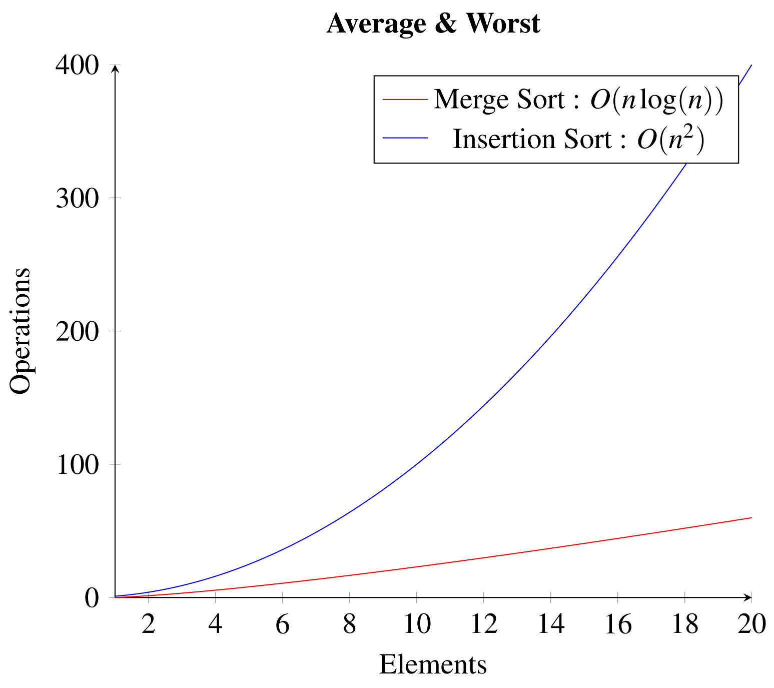


■ **Example 1.4** *Merge Sort* is always faster than *Insertion Sort*. True or false?. Justify your answer ■

Definition 1.1.5 Time Complexity is the computational complexity that describes the amount of time it takes to run an algorithm. Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm, supposing that each elementary operation takes a fixed amount of time to perform.

Big-O Time Complexity Graph





R When asked for running time of an algorithm this refers to $O(\cdot)$. Runtime time means worst case running time.

Solution: False: Merge sort is not always faster than insertion sort. In the **Best Case** Merge Sort's running time is $O(n\log(n)$ and Insertion Sort is $O(n)$. For **Average Case** and **Worst Case** Merge Sort is faster.

1.1.4 Insertion Sort vs Merge Sort

■ **Example 1.5** Suppose the input sequence is already sorted. For this specific input, what is the running time of Insertion-sort and Merge-sort? Assume that they are implemented as in the textbook. ■

Solution: For insertion sort the running time is $\Theta(n)$, for merge sort it is $\Theta(n\log(n))$. Insertion has to iterate through the entire array, the outer loop runs, but not the inner loop.

1.1.5 Running Time

■ **Example 1.6** Suppose there are 2^n inputs to a certain problem. The running time of an algorithm A is exactly 2^n for exactly one of the inputs, and 1 for any other input. Then the running time is O(1) since $2^n \cdot \frac{1}{2^n} + 1 \cdot (1 - \frac{1}{2^n}) \leq 2$. Is this statement correct? ■

Solution: False: The statement is in correct. O notation refers to worst case running time, this means the running time for the algorithm is $O(2^n)$.

1.2 Intermediate

1.2.1 Selection-Sort

■ **Example 1.7** The following is a pseudo-code of Selection-Sort. Describe Selection-Sort in plain English. ■

Solution:

```
n = A.length
for j = 1 to n - 1
    smallest = j
    for i = j + 1 to n
        if A[i] < A[smallest]
            smallest = i
    exchange A[j] with A[smallest]
```

Selection sort finds the smallest element in the array and swaps it with the first element, then the second smallest with the second element, third smallest swaps with third element and so on. For

selection sort the array of size n is iterated from the first index in the array to the one before the last index. It is assumed that the first index is the smallest, then the next elements are compared to the first element and if it is smaller then that index becomes the smallest, and this continues until every element after is tested. Then the element at index j is swapped with the smallest element.

■ **Example 1.8** State the loop invariant and prove that the algorithm is correct. What is the running time? ■

1. We want to show that the loop invariant is true prior to the first iteration of the loop, that each iteration maintains the loop invariant, and that the loop invariant gives us a useful property at loop termination.

   Loop Invariant: $A[\text{smallest}] \leq A[1..j-1]$
   - **Initialization** Prior to the first iteration A[1] is sorted. When j = 1 there is only one element and is therefore sorted. In the second for loop, i = j + 1, $A[1..j-1]$, it is only $A[i]$, one element only.

- **Maintenance** It remains true as j increases and i is the element after j. The outer loop from j = 1 to n -1 and i = j + 1 to n, the inner loop finds smallest element from j + 1 and the outer loop exchanges the jth element with the smallest element. The sub-array A[1..j] is sorted.
- **Termination** The sub-array A[1..n] is sorted, the outer loop iterates from j = 1 to n -1

2. The worst case running time is $O(n^2)$ and best case is $\Omega(n^2)$. The average case is $\Theta(n^2)$

1.2.2 Merge

■ **Example 1.9** Consider the pseudo-code of Merge(A, p, q, r) in the textbook. Given that A[p...q] and A[q+ 1...r] are both sorted, the function call merges the two sorted sub arraysinto a sorted subarray A[p...r]. Prove the correctness of Merge. For simplicity, you can assume that $L[1...n_1] = A[p...q]$ and $R[1...n_2] = A[q+1...r]$., and $L[n_1+1] = R[n_2+1] = \infty$. (So you only need to consider from lines 10). You can assume that all elements stored in the array have distinct values. ■

Solution: MERGE(A,p,q,r)

```
1      n_1 = q − p + 1
2      n_2 = r − q
3      let L[1..n_1 + 1] and R[1..n_2 + 1] be new arrays
4      for i = 1 to n_1
5          L[i] = A[p + i − 1]
6      for j = 1 to n_2
7          R[j] = A[q + j]
8      L[n_1 + 1] = \infinity
9      R[n_2 + 1] = \infinity
10     i = 1
11     j = 1
12     for k = p to r
13         if L[i] <= R[j]
14             A[k] = L[i]
15             i = i + 1
16         else A[k] = R[j]
17             j = j + 1
```

Loop Invariant: $k \in [p,r]$ elements in index k are sorted, this elements come from Left and Right subarrays, also $i <= n_1+1, j \leq n_2+1$ where $i \in [p,q], j \in [q+1,r]$ this means there are no more elements to be copied to A[p..r].

1. Initialization : Our merged array $A[p..r]$ before line 10 contains no elements from our sorted sub arrays $A[i],A[j]$ recall that elements were stored in new sub arrays $L[1..n_1]$ or $R[1..n_2]$ and therefore $A[k]$ is "empty", therefore sorted.
2. Maintenance : The for loop iterates from $k \in ptor$, and elements are added to A[k], we must show this loop invariant is in sorted order. Check by Iterating through each element in L[i] and R[j] and finding the smallest element after the first iteration A[p] holds one single element, this element is sorted since it belongs to the smallest element of subarray L[1..n_1] or subarray R[1..n_2].
3. Termination : The loop iterate through (r - p + 1) element's and compares each element in the subarrays $L[1..n_1$ and $R[1..n_2$ to find the smallest, then it is stored in the array A[p..r], at the end of

1.3 Advanced

1.3.1 Searching Problem

■ **Example 1.10** Consider the searching problem The input is an array A[1..n] of n numbers and a value v. You're asked to find an index i such that A[i] = v. If there's no such index, return NIL. The following is a pseudocode of linear search, which scans through the sequence, looking for v. Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulfills the there necessary properties. What is the asymptotic worst case running time? ■

Solution:

```
For i = 1 to n
    If A[i] == v then return i
return NIL
```

1. Loop Invariant: At the start of each *ith* iteration of the for loop $j \in [1, i-1]$ and A[j] is not v.
   (a) Initialization : Before the loop iteration the **variable answer** contains NIL,that is since $j = 0 \to A[0]$ does not exist. A[0] is therefore not v.
   (b) Maintenance : Assume the loop invariant holds then $A[j]$is not v. If $A[j+1]$ is v then the termination would have followed. Since termination has not followed (loop is still running) this implies that $A[j+1]$ is not v.
   (c) Termination : When the for loop terminates the variable answer contains v or NIL.It returns NIL if at index j + 1 the element is v, or returns NIL if at index j + 2 which is larger than the size of array and therefore A[j+2] is not v.
2. The asymptotic worst case is $O(n)$, that is the element is at the index n, it must examine every element.

1.4 Exam Problems

1.4.1 Insertion Sort

■ **Example 1.11** The following is a pseudocode of Insertion-Sort for instance A[1..8] = $\langle 7,4,2,9,4,3,1,6 \rangle$ what is $A[1..8]$ just before the for loop starts for j = 5?

INSERTION-SORT(A)

```
for j = 2 to A.length
    key = A[j]
    //insert A[j] into the sorted
        sequence A[1..j-1]
    i = j - 1
    while i > 0 and A[i] > key
        A[i+1] = A[i]
        i = i - 1
    A[i+1] = key
```

■

Solution:

j	After
j = 2	$\langle 4,7,2,9,4,3,1,6 \rangle$
j = 3	$\langle 2,4,7,9,4,3,1,6 \rangle$
j = 4	$\langle 2,4,7,9,4,3,1,6 \rangle$
j = 5	$\langle 2,4,4,7,9,3,1,6 \rangle$

Before j = 5 the array $A[1..8]$ is $\langle 2,4,7,9,4,3,1,6 \rangle$

■ **Example 1.12** Give an instance of size n for which the insertion-sort terminates in $\Omega(n^2)$ ■

Solution: The input should be n numbers in decreasing order. The while loop executes. Example: $\langle 6,5,4,3,2,1\rangle$

■ **Example 1.13** Give an instance of size n for which the Insertion-sort terminates in $O(n)$ time. ■

Solution: The input should be n number in increasing order. The while loop does execute. Example: $\langle 1,2,3,4,5,6\rangle$

■ **Example 1.14** The following is a pseudocode of Insertion-sort. Prove its correctness via loop invariant. In other words, state the loop invariant and prove it using *Initialization, Maintenance, and Termination*
INSERTION-SORT(A)

```
1 for j = 2 to A.length
2     key = A[j]
3     //Insert A[j] into the sorted
4         sequence A[1..j-1]
5     i = j - 1
6     while i > 0 and A[i] > key
7         A[i + 1] = A[i]
8         i = i - 1
9     A[i + 1] = key
```

■

Solution: Loop Invariant: At the start of each of the **for** loop of lines 1 -8, the subarray A[1..j-1] consists of the elements originally in A[1..j-1], but in sorted order.

1. **Initialization** Just at the start of the first iteration (j=2), A[1] is ordered and the number was originally in A[1].
2. **Maintenance** Say the invariant is true for iteration $j \geq 2$. At the end of the iteration. The algorithm has $A[1..i]$, A[j], and A[i+1..j-1] in this order in the prefix of A. We know that $A[1..i]$ and $A[i+1..j-1]$ are sorted by the invariant. Further, $A[i] \leq A[j] \leq A[i+1]$ by Algo's definition, meaning A[1..j] is sorted at the end of iteration. Clearly, all elements in $A[1..j]$ originate form the same subarray. Thus, the loop invariant holds before the next iteration j + 1/
3. **Termination** The for loop ends when j = n + 1, and the loop invariant implies that the array is sorted as desired.

1.4.2 RAM - Random Machine Model

■ **Example 1.15** Briefly explain the Random Access Model (RAM) ■

Solution: The RAM model is used for analyzing algorithms where we assume that instructions are executed one after the other. This model is theoretical.It is used to model how real world programs would run.
Basic operations such as addition, multiplication, load, store, copy, control, initialization are assumed to take a constant amount of time each (if numbers are big, this may not be true, but we assume that this is the case unless stated otherwise).

1.4.3 Merge Sort

■ **Example 1.16** Let T(n) denote the running time of Merge-sort on input size n. In the following you can omit floor or ceeling. MERGE-SORT(A,p,r)

```
if p < r
    q = floor((p+r)/2)
    MERGE-SORT(A,p,q)
    MERGE-SORT(A,q+1,r)
    MERGE(A,p,q,r)
```

■

R **Running time of Merge Sort**

$$f(a,b) = \begin{cases} O(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n \geq 2 \end{cases}$$

1. What is the running time of Line 3 (of Merge-Sort)?
   Solution: A call to merge sort is $T(n/2)$
2. What is the running time of Line 4 (of Merge-Sort)?
   Solution: A call to merge sort is $T(n/2)$
3. What is the running time of LIne 5 (of Merge-Sort)?
   Solution: A call to merge is $\Theta(n)$

1.5 Solutions To Exercises

2. Growth of Functions

2.1 Basic

2.1.1 Ranking Functions

■ **Example 2.1** Rank the following functions by order of growth; that is, find an ordering $g_1, g_2, \cdots, g_k$ (here k is the number of functions given) such that $g_1 = \mathcal{O}(g_2), g_2 = \mathcal{O}(g_3), \cdots, g_{k-1} = \mathcal{O}(g_k)$ (For example, if you are given functions, $n^2, n, 2n$, your solution should be either $n, 2n, n^2$ or $2n, n, n^2$.)

$n^2 + 2^n$	$n\log(n)$	$n^2(\log(n))^2$	$n(\log(n))^2$	n^2	$log^{100}(n)$
$\log(\log(n))$	n^3	1	$\log(n)$	$\frac{\log(n)}{\log\log(n)}$	$\frac{n^2}{log(n)}$
$n^{10}3^n$	4^n				

Solution:

1. **Asymptotic upper bound**. $\mathcal{O} - notation$

   $$\mathcal{O}(g(n)) = \{f(n) : \text{there exist positive constants c and } n_o \text{ such that } 0 \leq f(n) \leq c(g(n) \text{for all} n \geq n_o\}$$

2. We consider the following. (Algorithm Design Manual, Page 40)

   $$1 \ll \log(n) \ll n \ll n\log(n) \ll n^2 \ll n^3 \ll 2^n \ll n!$$

3. Find g_1 the smallest function. Since 1 is a constant it is the smallest function. Therefore

   $$g_1 = 1$$

4. Find g_2. We know that $\log(n)$ is smaller than n, so $\log(\log(n))$ is smaller than $\log(n)$ Therefore

   $$g_2 = \log(\log(n))$$

5. Find g_3, g_4, g_5

- Candidates are $\log(n)$, $\log^{100}(n)$ and $\frac{\log(n)}{\log(\log(n))}$
- $\log(n) < (\log(n))^{100}$.
- We find $\frac{\log(n)}{1} > \frac{\log(n)}{\log(\log(n))}$. Recall if that the larger the denominator the smaller the value.
- Therefore we know that $\frac{\log(n)}{\log(\log(n))} < \log(n) < (\log(n))^{100}$

$$g_3 = \frac{\log(n)}{\log(\log(n))}, g_4 = \log(n), g_5 = (\log(n))^{100}$$

6. Find g_6, g_7
   - Find all functions that have n. There are no values with n.
   - Possible candidates $n\log(n), n(\log(n))^2$.

$$n\log(n) < n(\log(n))^2$$

$$g_6 = n\log(n), g_7 = n(\log(n))^2$$

7. Find g_8, g_9, g_10
   - Find functions that contain n^2. Potentially $n^2, n^2(\log(n))^2, \frac{n^2}{\log(n)}$
   - Since $\frac{n^2}{\log(n)} < n^2$
   - And $n^2 < n^2(\log(n))^2$
   - Therefore $g_8 = \frac{n^2}{\log(n)}, g_9 = n^2, g_{10} = n^2(\log(n))^2$
8. Find $g_{11}, g_{12}, g_{13}, g_{14}$
   - Find functions with n^3. Only n^3 is left.

$$g_{11} = n^3$$

   - Find functions with 2^n. Functions $n^2 + 2^n, 2^n$. Therefore

$$g_{12} = 2^n, g_{13} = n^2 + 2^n$$

   - Finally $g_{14} = n^{10}3^n$

Solution:

$$1, \log(\log(n)), \frac{\log(n)}{\log(\log(n))}, \log(n), (\log(n))^{100}, n\log(n), n(\log(n))^2, \frac{n^2}{\log(n)}, n^2, n^2(\log(n))^2, n^3, 2^n, n^2+2^n$$

■

2.1.2 Asymptotically no smaller

■ **Example 2.2** What is $\lim_{n\to\infty} \frac{n^{10}\cdot 3^n}{4^n}$? Which one is asymptotically no smaller between the two functions? ■

Solution:

1. Consider the following

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0, \text{implies} f(n) = O(g(n))$$

2.

$$\lim_{n\to\infty} \frac{n^{10}3^n}{4^n} = 0$$

3. Since $4^n > n^{10}3^n$
4. $g = \Omega(n^{10}3^n)$
5. $f = O(4^n)$

The limit is 0, and the function that is asymptotically no smaller between the two functions is $f = O(4^n)$ since 4^n is the upper bound and $n^{10}3^n$ is the lower bound.

2.1.3 Using asymptotic definitions

■ **Example 2.3** Formally prove that $n^2+100 = \mathcal{O}(n^2)$ using the definition of $\mathcal{O}(\cdot)$ ■

Solution:

1. Consider the following.

> **Definition 2.1.1** $\mathcal{O}(g(n)) = \{f(n)$: there exist positive constants c and n_o such that $0 \leq f(n) \leq c(g(n)$for all$n \geq n_o\}$

2. We want to show that $0 \leq n^2+100 \leq c \cdot n^2$

$$n^2+100 \leq c \cdot n^2$$

$$100 \leq n^2(c-1)$$

$$0 < \frac{100}{c-1} \leq n^2$$

3. A c was chosen by inspection, then you may use c to solve the inequality and solve for n.
4. If $c \geq 101$ then $1 \leq n^2$ this holds true for $n \geq 1$, therefore we have $n^2+100 \leq 101n^2$

Solution: Since c , and n_0 exist where $n \geq n_0$ that is $n^2+100 \leq 101n^2$

■ **Example 2.4** Formally prove that $n^2 = \Omega(n^2+100)$ Using the definition of $\Omega(\cdot)$. ■

Solution:

1. Consider the following.

> **Definition 2.1.2** $\Omega(g(n) = \{f(n)$: there exist positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$

2. We want to show that $0 \leq c(n^2+100) \leq n^2$ by finding a c and n_0 such that $n \geq n_0$

$$0 \leq c(n^2+100) \leq n^2$$

$$cn^2+100c \leq n^2$$

3. Choose a c by inspection.If c is 1, it does not hold. Less than one, we want $100c$ to head to zero. Therefore, choose $c = \frac{1}{101}$ meaning $\frac{n^2}{100}+\frac{100}{101} \leq n^2$
4. $c = \frac{1}{101}$ means

$$\frac{1}{101}(n^2+100) \leq n^2$$

$$n^2+100 \leq 101n^2$$

$$100 \leq 100n^2$$

$$1 \leq n^2 \rightarrow \pm 1 \leq n$$

5. for $c = \frac{1}{101}, n \geq 1$

Solution: There exists a c and a n, this means the definition holds.In turn $\frac{1}{101}(n^2+100) \leq n^2$

■ **Example 2.5** Formally prove that $n^2 = \Omega(n\log_2 n+100)$ using the definition of $\Omega(\cdot)$ ■

1. Consider the following.

Definition 2.1.3 $\Omega(g(n) = \{f(n)$: there exist positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$

2. We want to show that $0 \leq c(n\log_2(n) + 100) \leq n^2$

$$c(n\log_2(n) + 100) \leq n^2$$

3. Algebra: Distribute c, then Raise every-ting to the power of 2, to cancel the log_2. Recall $2^{a+b} = 2^a 2^b$. Recall properties of log. $a\log_2 b \to \log_2 b^a$

$$2^{\log_2(n^{cn})} 2^{100c}) \leq 2^{n^2}$$

$$n^{cn} 2^{100c} \leq 2^{n^2}$$

4. Choose a c by inspection.
5. If $c = 1$ then

$$n^n \leq 2^{n^2 - 100}$$

$$n\log_2 n \leq n^2 - 100$$

$$0 \leq n^2 - n\log_2(n) - 100$$

6. It holds for some n, unable to find. Come back later.

■ **Example 2.6** Formally prove that $n + 10 = \Theta(50n + 1)$ using the definition of $\Theta(\cdot)$ ■

Solution:

1. Consider the following.

   Definition 2.1.4 $\Theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2 and n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$

   $f(n) = \Theta(g(n))$ iff $f(n) = \Omega(g(n))$ and $f(n) = \mathcal{O}g(n)$

2. We want to show that $c_1(50n + 1) \leq n + 10 \leq c_2(50n + 1)$
3. Proving Ω and $\mathcal{O}$ makes it easier.
4. First prove the upper bound $\mathcal{O}$.

$$n + 10 \leq c_2(50n + 1)$$

   If $c_2 = 1$ and $n \geq 1 \to n + 9 \leq 50n$ holds.
5. Second prove the lower bound Ω.

$$c_1(50n + 1) \leq n + 10$$

6. If $c_1 = \frac{1}{50}$ then for $n + \frac{1}{50} \leq n + 10$

Solution: Since we have found a c_1, c_2, n the definition is met. Therefore

$$\frac{1}{50}(50n + 10) \leq n + 10 \leq 50n + 1$$

2.2 Intermediate

2.2.1 Using big-O notation

Example 2.7 Prove that $f = O(g)$ implies $g = \Omega(f)$

1. $f(n) = O(g(n)), g(n) = \Omega(f(n)$
   Meaning that if g(n) is the upper bound then this implies $f(n)$ and $f(n)$ is the lower bound of $g(n)$
2. Solution : $f(n) \leq c_1 g(n)$ Definition of O $\rightarrow \frac{1}{c_1} f(n) \leq g(n) \rightarrow c_2 f(n) \leq g(n)$ Definition of Ω (note $c_2 = \frac{1}{c_1}$)

Example 2.8 Prove that $f = \Omega(g)$ and $g = \Omega(h)$ implies $f = \Omega(h)$

1. $g = \Omega(h)$ means $0 \leq ch(n) \leq g(n)$ for $n_1 > n_0$
2. $f = \Omega(g)$ means $0 \leq cg(n) \leq f(n)$ for $n_2 > n_0$
3. $f = \Omega(h)$ means $0 \leq ch(n) \leq f(n)$ for $n_2 > n_1$

2.3 Exam Problems

2.3.1 True or False

For each of the following claims, decide if it is true or false. No explanation is needed.
To solve consider the following

1. **Dominance Pecking Order** *Page 56, Algo Design Manual*

$$1 < \alpha(n) < \log(\log(n)) < \frac{\log(n)}{\log(\log(n))} < \log(n) < \log^2(n)$$

$$< \sqrt{n} < n < n\log(n) < n^{1+\varepsilon} < n^2 < n^3 < c^n < n!$$

2. **Asymptotic Upper Bound**

   Definition 2.3.1 $f(n) = O(g(n))$means $c \cdot g(n)$ is an *upper bound* on f(n). Thus there exists some constant c such that f(n) is always $\leq c \cdot (n)$, for large enough n (i.e.., $n \geq n_o$ for some constant n_0).

3. **Asymptotic Lower Bound**

   Definition 2.3.2 $\Omega(g(n)) = \{f(n)$: there exist positive constant c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$

4. **Asymptotically tight bound**

   Definition 2.3.3 $\Theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2 and n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$

Example 2.9 $n\log(n) = O(n^2)$

Solution: True, since $n\log(n) < c \cdot n^2$

Example 2.10 $\log(\log(n)) = O(\log(n))$

Solution: True, since $\log(\log(n)) < c \cdot \log(n)$

Example 2.11 $\log^{50} n = O(n^{0.1})$

Solution: True, since $\log^{50} n \leq c(n^{0.1}))$

Example 2.12 $4^n = O(2^n)$

Solution: False, since $4^n \not\leq c \cdot (2^n)$

■ **Example 2.13** $100^{100} = \Theta(1)$ ■

Solution: True, since $100^{100} \leq c \cdot 1$

■ **Example 2.14** If $f = O(g)$, then $g = \Omega(f)$ ■

Solution: True. If $f \leq cg(n)$ then $g(n)$ is the upper bound, this implies that $f(n)$ is the lower bound, therefore $g \geq cf(n)$

■ **Example 2.15** If $n^3 = \Omega(n^2)$ ■

Solution: True, since $n^3 \geq c \cdot (n^2)$

■ **Example 2.16** $100 + 200n + 300n^2 = \Theta(n^2)$ ■

Solution: True, since $100 + 200n + 300n^2 < c \cdot (n^2)$ and $c \cdot (100 + 200n + 300n^2) > (n^2)$

■ **Example 2.17** $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 5$ then $f(n) = O(g(n))$ ■

Solution: True.

Definition 2.3.4 If there exists a constant $c \geq 0$ such that $\lim_{n\to\infty} \frac{f(n)}{f(n)} \leq c$, then $f = O(g)$

■ **Example 2.18** $\sum_{i=1}^{n} \Theta(i) = \Omega(n^2)$ ■

Solution: True. $\Theta(1) + \Theta(2) + \cdots + \Theta(n) > c \cdot n^2$

2.3.2 Proving using definition of O

■ **Example 2.19** Formally prove that $50n + 15 = O(n^2)$ using the definition of $O(\cdot)$ ■

Definition 2.3.5 $\mathscr{O}(g(n)) = \{f(n)$: there exist positive constants c and n_o such that $0 \leq f(n) \leq c(g(n))$for all$n \geq n_o\}$

Solution:

$$0 \leq 50n + 15 \leq c \cdot n^2$$

$$\frac{50n + 15}{n^2} \leq c$$

Part 2

Part Three

IV Final

3. Binary Search Trees (BST)

In this chapter inorder, preorder, and postorder **traversal** techniques will be introduced.

3.1 Basic

3.1.1 Inorder, Pre- order, Post-order

■ **Example 3.1** Consider the BST in Fig 12.1.(b). Print out all the keys in the BST in preorder,then postorder ■

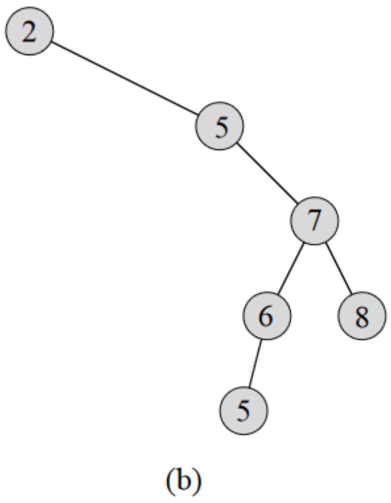


Figure 3.1: Fig 12.1(b)

R Nodes with no children are called leaves. Nodes which are not leaves are called internal nodes. Nodes with the same parent are called siblings.

Solution: Pre-order $\langle 2,5,5,6,7,8\rangle$

Definition 3.1.1 Preorder-Tree-Walk(x) x (root), x's left subtree, and x's right subtree. (Visit the root, traverse the left subtree, traverse the right subtree)

```
if x ≠ NIL
    print(x.key)
    Preorder-Tree-Walk(x.left) //Left subtree
    Preorder-Tree-Walk(x.right)
```

Post-order: $\langle 5,6,8,7,5,2\rangle$

Definition 3.1.2 Postorder-Tree-Walk: x's left subtree, x's right subtree, and x (root). (Traverse the left subtree, traverse the right subtree, visit the root)

```
if x ≠ NIL
    Postorder-Tree-Walk(x.left) //Left subtree
    Postorder-Tree-Walk(x.right) //Right subtree
    print(x.key)
```

R Keys in a BST may or may not be distinct. Figure 4.1 has duplicate values.

■ **Example 3.2** A full binary tree of 7 nodes, find Inorder, Pre-order, Post-order ■

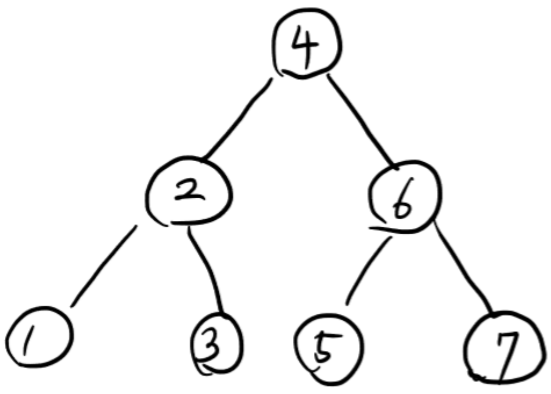


Figure 3.2: Full Binary Tree

R A full binary tree is a binary tree in which each node has exactly zero or two children. A complete binary tree is $\Theta(\log(n))$ in worst case.

Solution:

- **Inorder** ⟨1,2,3,4,5,6,7⟩

Definition 3.1.3 Inorder-Tree-Walk: x's left subtree, x (root), and x's right subtree. (Traverse the left subtree, visit the root, traverse the right subtree)

```
if x ≠ NIL
    Inorder-Tree-Walk(x.left)
    Inorder-Tree-Walk(x.right)
    print(x.key)
```

- **Pre-order** ⟨4,2,1,3,6,5,7⟩
- **Post-order** ⟨1,3,2,5,7,6,4⟩

R The worst-case running time for most search-tree operations is proportional to the height of the tree.

3.1.2 Sequence of nodes

Example 3.3 Suppose that we have numbers between 1 and 1000 in a binary search tree, and we want to search for the number 363. Which of the following sequences could not be a sequence of nodes examined? ∎

Consider the following
Binary-Search-Tree Property

Definition 3.1.4 For any node x in BST.
- If y is in left subtree of x, then y.key ≤ x.key
- If y is in the right subtree of x, then y.key ≥ x.key

(a) 2,252,401,398,330,344,397,363
(b) 924,220,911,244,898,258,362,363
(c) 925,202,911,240,912,245,363
Does not meet BST property Since 911 < 912. BST in Figure 4.3
(d) 2,399,387,219,266,382,381,278,363
(e) 935,278,347,621,299,392,358,363

Construct Binary Search Tree
1. Make the first element the root (x)
2. For the next element (y)
   - If value (y.key) is ≤ node.value (x.key)
     Place left
   - If value (y.key) is > node.value (x.key)
     Place right
   - If the place is empty
     Place the node

R The sequence of nodes are valid if you can easily traverse through them to find a value. If the BST has a single path, then it is valid.

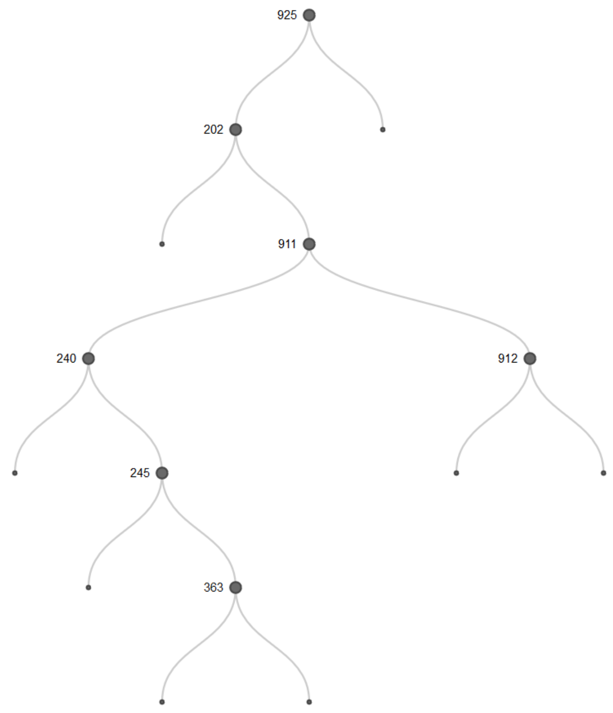


Figure 3.3: Part (c)

3.1.3 Binary Search Tree Problem

■ **Example 3.4** Consider the following binary search tree where nodes are labeled by alphabets. Here the keys are not shown in the picture; *a,b,c,...* are node labels, not keys. Assume that all keys have distinct values.

■

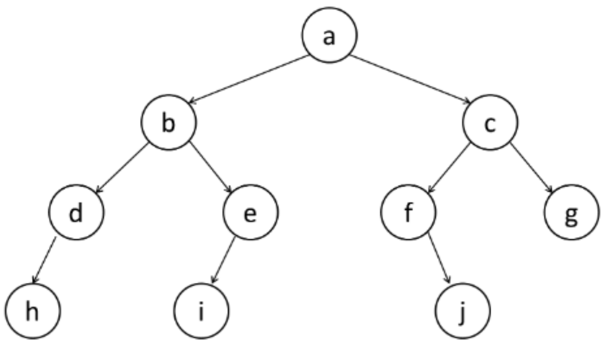


Figure 3.4: Example 3.4

(A) What is the node with the min key value?

Definition 3.1.5 TREE-MINIMUM(x): The min key is at the leftmost node.

```
    while x.left != NIL
```

```
        x = x.left
    return x
```

Solution : Node *h* has the min key value.

(B) What is the node with the max key value?

Definition 3.1.6 TREE-MAXIMUM(x): The max key is at rightmost node.

```
    while x.right != NIL
        x = x.right
    return x
```

Solution : Node *g* has the max key.

R Running time for TREE-MINIMUM and Tree-MAXIMUM is O(*h*) where *h* is the subtree's height.

(C) What is the node with the upper median key value?

Definition 3.1.7

(D) What is the node with the lower median key value?

(E) What is *e*'s successor?

Definition 3.1.8 Successor: The successor of a node x is the node y such that y.key is the smallest key greater than x.key

```
1    if x.right != NIL
2        return TREE–MINIMUM(x.right)
3    y = x.p //(Parent of x)
4    while y != NIL and x == y.right
5        x = y
6        y = y.p //(Parent of y)
7    return(y)
```

Solution : The successor is *a*.

(a) **Lines 1-2:** Start at node label *e* (x), the right subtree of *e* is NIL (x.right) therefore **line 2** does not execute.
(b) **Line 3:** y is set to the parent of x (x.p) which is node label *b*
(c) **Lines 4: While Loop** Since y is *b* it is not NIL and (y.right) is x which is *e*.
(d) **Lines 5-6: Inside While** Set x equal to y which is *b*, and y equal to parent of y which is *a*.
(e) **Lines 4: While Loop** y is b and not NIL and (y.right) is x which is *a*
(f) **Lines 5-6: Inside While** Set x equal to y which is *b*, and y equal to the parent of y which is NIL (y.p).
(g) **Lines 7** While loop terminates. Returns y which node label **a**

(F) What is *i*'s successor?

Solution: The successor is *e*

(a) **Lines 1-2:** Start at node label *i*, the right subtree of *i* is NIL (x.right) therefore **line 2** does not execute.
(b) **Line 3:** y is set to the parent of x (x.p) which is node label *e*

(c) **Line 4: While Loop** Since y is *e* it is not NIL and (y.right) is NIL not x, the while loop does not execute **Lines 5-6**.
(d) **Line 7:** Returns y which is node label **e**.

(G) What is *g*'s successor?
Solution: The successor is *a*
(a) **Line 1-2:** Start at node label *g* (x), the right subtree is NIL (x.right) therefore **line 2** does not execute.
(b) **Line 3:** y is set to the parent x (x.p) which is node label *c*
(c) **Line 4: While Loop** Since y is *c* it is not NIL and (y.right) is x which is *g*
(d) **Line 5-6: Inside While** Set x equal to y which is **c**, and y equal to the parent of y which is *a*.
(e) **Line 4: While Loop** Since y is *a* and (y.right) is x which is *c*.
(f) **Line 5-6: Inside While** Set x equal to y which is **a** and y equal to the parent of y which is **NIL**
(g) **Line 4: While Loop** Since y is NIL **Lines 5-6** do not execute.
(h) **Line 7:** Returns y which is node label **a**.

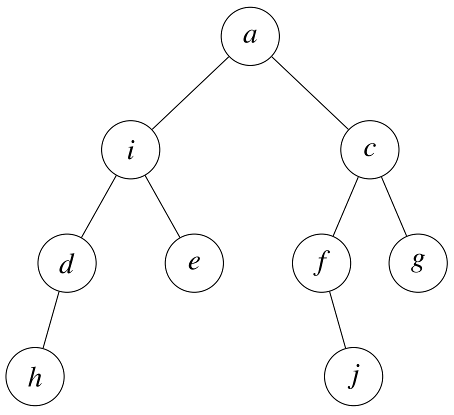
(H) What is *j*'s predecessor?
Solution: The predecessor is *c*

R The running time for **successor** and **predecessor** is $O(h)$

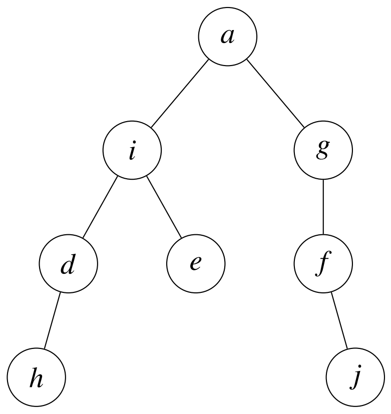
■ **Example 3.5** Consider the BST in the above problem. ■

Definition 3.1.9 Find the minimum of right subtree, copy that value to the node that will be deleted, and delete the duplicate from the right subtree.
Find maximum in left subtree, copy that value to the node that will be deleted, and then delete the duplicate from the right subtree.

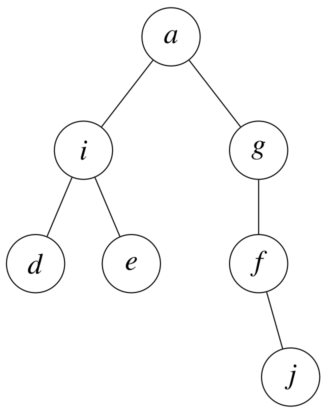
(A) Delete node b and show the resulting BST
Solution: Remove *b* and replace with minimum in rights subtree. We find *e* to be the minimum.



(B) Continue to delete another node c and show the resulting BST.



(C) Continue to delete another node h and show the resulting BST.

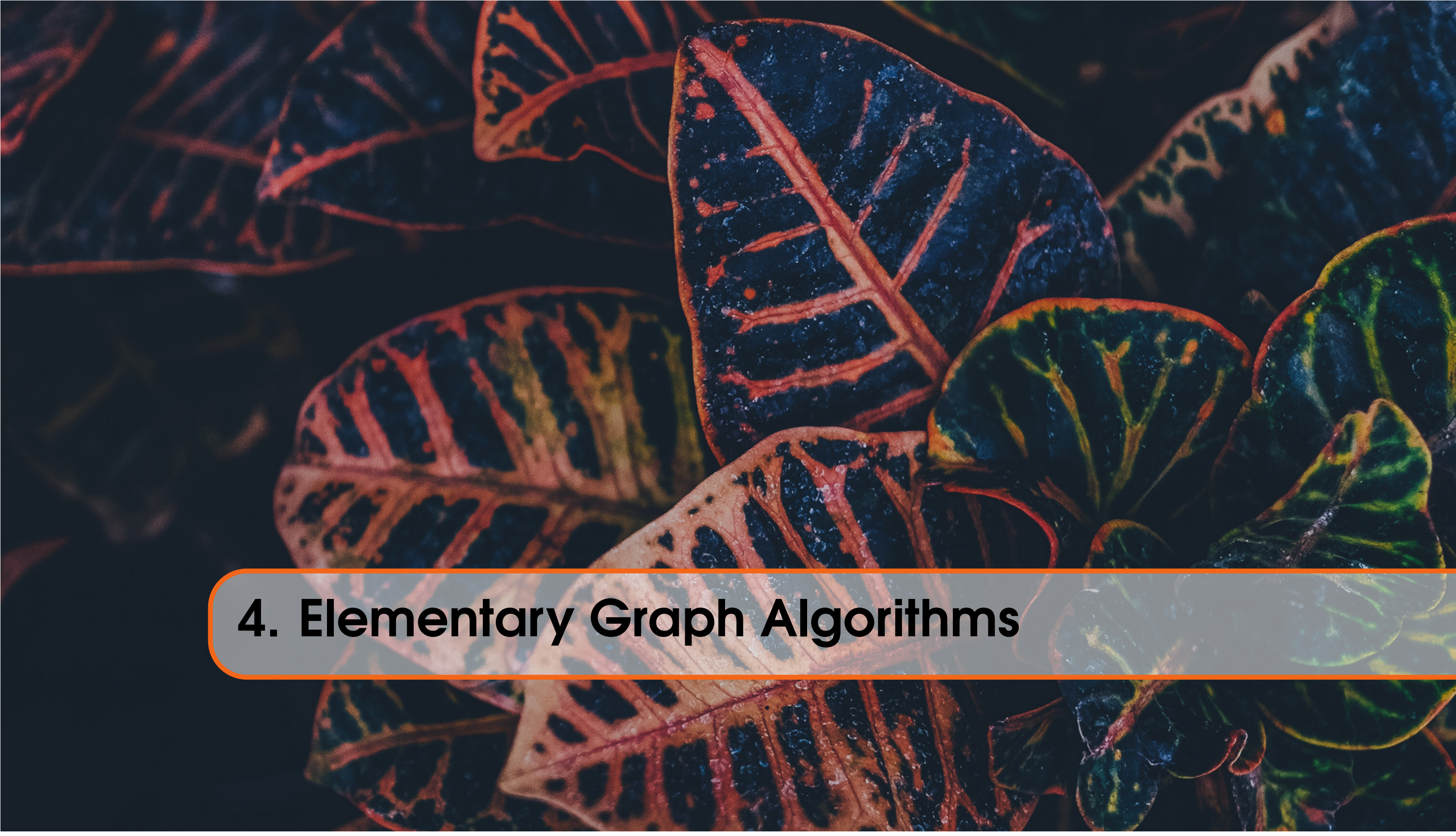


R When leaf nodes are deleted the BST property is maintained. Leaf nodes do not have right or left children. When deleting an internal node that has only one child remove internal node and link its parent to the child.

3.2 Advanced

3.3 Sources and Resources

In order to fully understand the concepts here. Consider the following items.
https://www.cs.rochester.edu/ gildea/csc282/slides/C12-bst.pdf



4. Elementary Graph Algorithms

4.1 Basic

4.1.1 Properties of BFS and DFS

■ **Example 4.1** To implement BFS, what data structure do you use? What about DFS (if we use no recursion)? ■

Definition 4.1.1 Breadth First Search (BFS):

1. **INPUT:** Graph $G = (V, E)$ and a source s.
2. **OUTPUT:** A tree (Breadth First Tree) consisting of vertices reachable from s encoding distance from source vertex s
   - The vertices that are reachable from s
   - The shortest distance from s to each reachable vertex
   - A shortest paths tree that allows reporting the shortest path from s to a reachable vertex.
3. Works on Direct and Undirected graphs
4. Time Complexity is $\Theta(V+E)$
5. Uses a queue (FIFO) data structure.

Definition 4.1.2 Depth First Search (DFS):

1. **INPUT:** Graph $G = (V, E)$ and no source.
2. **OUTPUT:**
   - π to record predecessors (to encode the resulting DFF)
   - two timestamps on each vertex v
3. Time complexity is $\Theta(V+E)$ if g is represented using **adjency lists**
4. Information about structure of graph.

Solution:

1. To implement Breadth First Search (BFS) we use a queue data structure. First In First Out

(FIFO)
2. For Depth First Search (DFS) we use a stack. Last In First Out (LILO)

4.1.2 Undirected and Directed Graph

■ **Example 4.2** We're given a directed graph G, along with a pair of vertices, u and v. We would like to know if there is a path from u to v. What algorithm would you like to use? ■

Definition 4.1.3 Graph: $G = (V, E)$. V is set of vertices. E is set of edges.

Definition 4.1.4 Undirected Graph: Edges are unordered pairs.
- Edges (u, v) and (v, u) are the same.
- Edges go from one vertex to another.
- No self loop. A *self-loop* is an edge that connects a vertex to itself. (v,v) $\notin E$

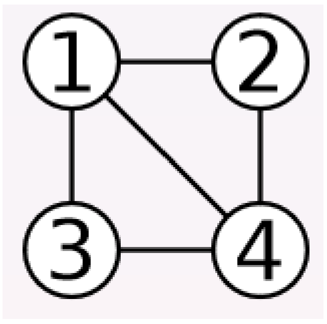


Figure 4.1: Undirected Graph

Definition 4.1.5 Directed Graph: Edges are ordered pairs.
- Edges (u, v) and (v, u) are different.
- u is called tail, and v is the head.

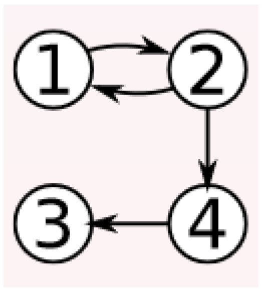


Figure 4.2: Directed Graph

Solution: The algorithm Breadth First Search (BFS).

4.1.3 Definition of Breadth First Tree (BFT)

■ **Example 4.3** What is the definition of BFT (breadth first tree)?
Suppose the graph consideration is undirected. Could there be an edge between two vertices whose depths differ by more than one? ■

Definition 4.1.6 Breadth First Tree

4.1.4 BFS and DFS problem

■ **Example 4.4** Consider the following directed graph.

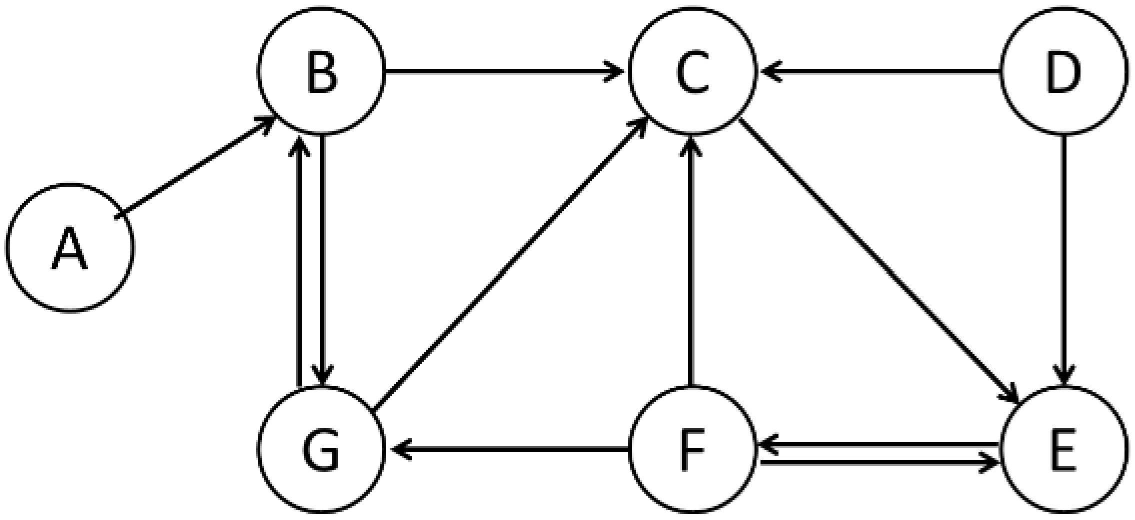


Figure 4.3: (a) through (h)

■

(a) Draw the adjacency-list representation of G, with each list sorted in increasing alphabetical order.

Definition 4.1.7 Adjacency-list Represents the graph $G = (V, E)$ as an array of linked list. The index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex.

(1) Each vertex in the graph is placed in a box in the left hand column
(2) Each adjacent vertex is shown as a node in the list to its right.
(3) The arrows indicate the next node in the list.

R Generally the order of the vertices in an adjacency list do not matter unless specified.

Solution:

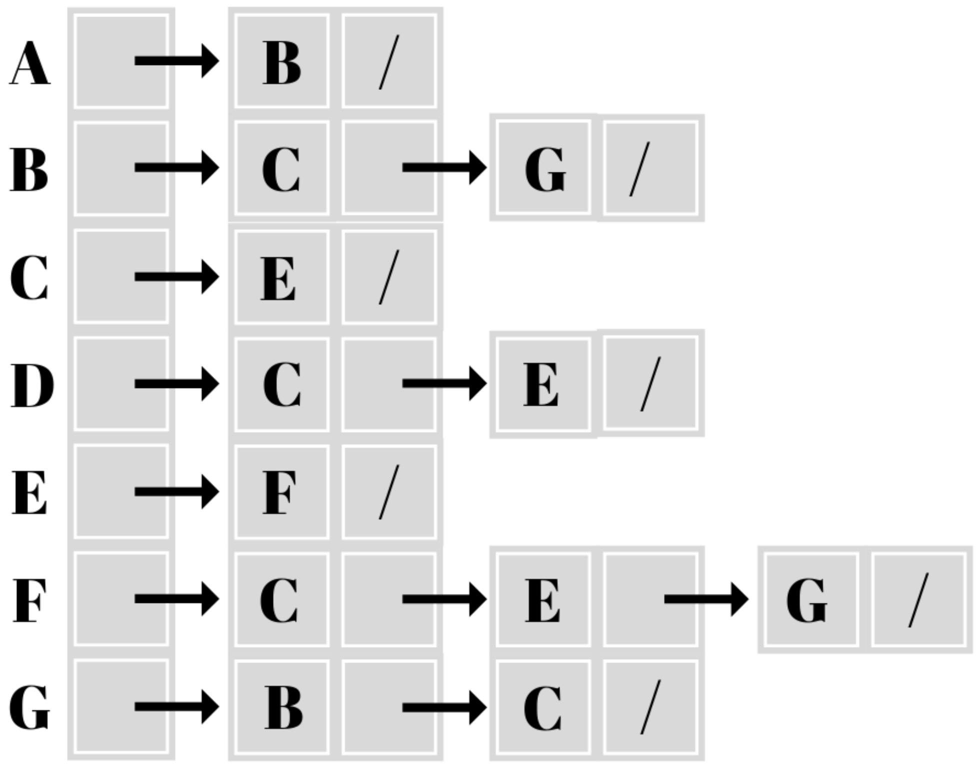


Figure 4.4: (a)

(b) Give the adjacency matrix of G.

> **Definition 4.1.8 Adjacency Matrix** of a graph $G = (V, E)$ is represented by a $|V| \times |V|$ matrix, $A = (a_{ij})$ where $a_{i,j} = 1$ if $(i, j) \in E$ and 0 otherwise.

(1) There are 7 vertices (|V|). The matrix should be 7×7
(2) If there exists and edge between two vertices then a_{ij} is *True (1)* otherwise *False (0)*

$$\mathbf{A} = \begin{array}{c|ccccccc}
 & A & B & C & D & E & F & G \\ \hline
A & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
B & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
C & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
D & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
E & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
F & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\
G & 0 & 1 & 1 & 0 & 0 & 0 & 0
\end{array}$$

(c) Draw the graph, the adjacency-list representation (with each list sorted in increasing alphabetical order), and the adjacency matrix for the transpose graph G^T.

> **Definition 4.1.9 Transpose of Graph:** Reverse the order of edges.
> $(u, v) \to (v, u)$ and $(v, u) \to (u, v)$ where $(u, v) \in E$.

- Find the transpose G^T

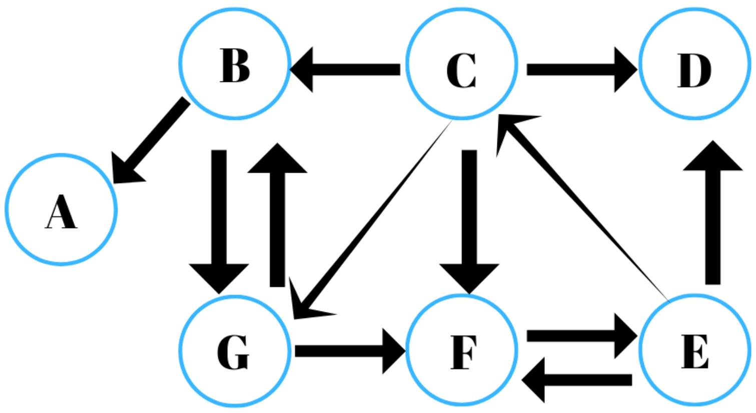


Figure 4.5: (c) G^T

- Draw adjacency-list representation.

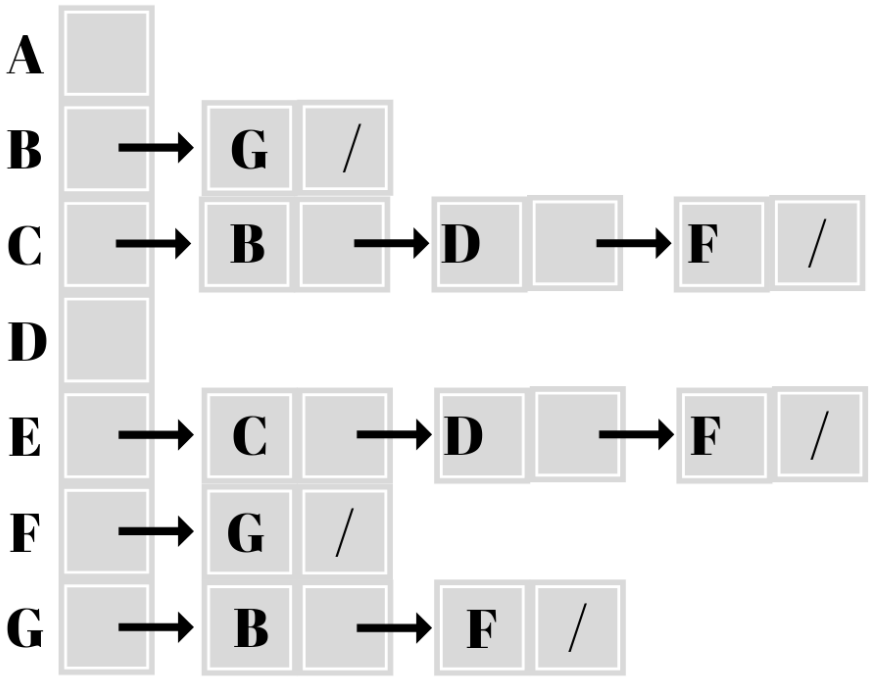


Figure 4.6: (c) Adjacency-list

- Give adjacency matrix.

(d) Do depth-first search in G, considering vertices in increasing alphabetical order. Show the final result, with vertices labeled with their starting and finishing times, and edges labeled with their type (T/B/F/C).

(e) Based on your results, proceed to to run the algorithm to find the strongly connected components of G (show the result of the DFS in G^T, with vertices labeled with their starting and finishing times.

(f) Draw the component graph G^{SCC} of G

(g) Find a topological sort of G^{SCC} using the following algorithms. (label each vertex with its DFS finishing time)

(h) Run BFS with B as a starting vertex. Show the tree edges produced by the BFS along with v.d of each vertex v. You must draw the current tree edges in each itration together with the queue status. More precisely, run BFS with B starting point assuming that each adjacency list is sorted in increasing alphabetical order.

4.1.5 DFF

■ **Example 4.5** If u is reachable from v, u muse be a descendant of v in any DFF. ■

4.2 Intermediate

4.3 Advanced

5. Minimum Spanning Trees

5.1 Basic

5.1.1 Terminology

■ **Example 5.1** What is the definition of a Tree. ■

Solution: A connected graph with no cycles.

■ **Example 5.2** If a tree **T** has n vertices, then **T** must have $n-1$ edges? ■

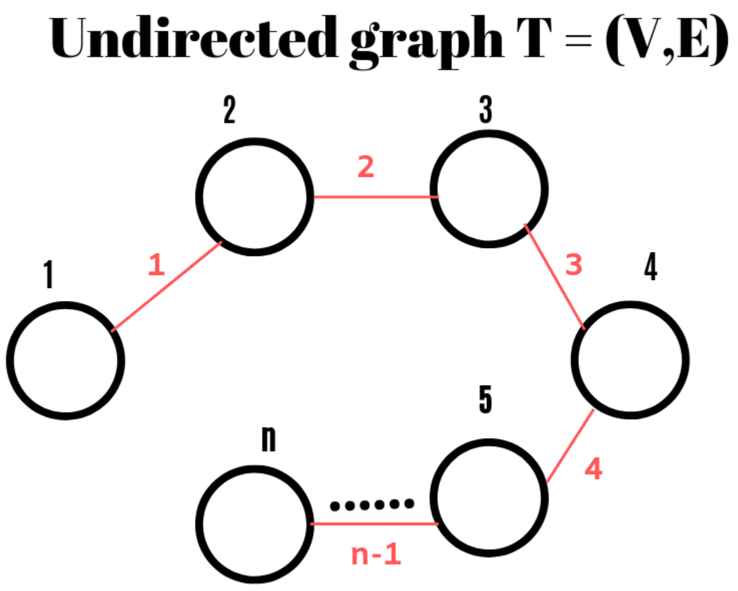


Figure 5.1: Conneced **T**

Solution: *True*. A tree does not have cycles. A tree is an undirected graph in which any two vertices are connected by exactly one path.

R **TRUE:** If a graph **G** = (V,E) has |V| edges or more, then it must have a cycle.

■ **Example 5.3** Consider the pseudocode of **GENERIC-MST(G,w)**. Where the **Input:** Undirected graph $G = (V,E)$ and weight/cost $w(u,v)$ for every edge $(u,v) \in E$. What is the output? ■

GENERIC-MST(G,w)

```
A = ∅
While A does not form a spanning tree
    find and edge $(u,v)$ that is safe for A
    $A = A \cup (u,v)$
return(A)
```

Definition 5.1.1 Spanning Tree: A tree that connects all vertices of $G = (V,E)$

Definition 5.1.2 Minimum Spanning Tree: A spanning tree denoted **T** whose total edge weight is minimized.

Solution: A minimum spanning tree $T \subseteq E$

R To find the **Spanning Tree** of a graph $G = (V,E)$ the vertices must be connected.

■ **Example 5.4** What is the definition of safe edges (assuming that edges have all distinct weights)? The lecture slides use a definition that is different from the textbook. Use the definition in the lecture slides, which is simpler. ■

Solution:

6. Single-Source Shortest Paths

Basic

6.0.1 Optimality of subpaths

■ **Example 6.1** Explain why the following lemma is true.

Lemma 6.0.1 **Optimality of subpaths:** If $P = \langle v_0, v_1, \dots, v_k \rangle$ is a shortest path from v_0 to v_k, then for any $0 \le i \le j \le k$, $\langle v_i, v_{i+1}, \dots, v_j$ is a shortest path from v_i to v_j. In other words, subpaths of shortest subproblems are also shortest paths.

■

Solution: One could get a "better" shortest path from v_0 to v_k by replacing $\langle v_i, v_{i+1}, \dots, v_j$ with a better path from v_i to v_j

6.0.2 Bellman-Ford Algorithm

■ **Example 6.2** The Bellman Ford algorithm repeats relaxing the entire set of edges $|V| - 1$ times. Each iteration edges can be relaxed in an arbitrary order. Consider the following chain graph. The source vertex is s.

$$\begin{tikzcd}
s \arrow[r, "2"] & a \arrow[r, "1"] & b \arrow[r, "3"] & c
\end{tikzcd}$$

Find the values of $s.d, a.d, b.d$ and $c.d$ after exactly one iteration, i.e. just after you relax all edges for $i = 1$, when edges are considered in each of the following orders?

(a) (s,a), (a,b), (b,c).
(b) (b,c), (a,b), (s,a).

■

Bibliography

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rives, and Clifford Stein. *Introdudction to Algorithms*. MIT, 2009.

[2] Sunjin: CSE 100 Algorithm Design and Analysis, `http://faculty.ucmerced.edu/sim3/`

[3] Carreira-Perpiñan: CSE 100 Algorithm Design and Analysis, `http://faculty.ucmerced.edu/mcarreira-perpinan/`