Data Structures and Algorithms

Eliezer A. Albacea

University of the Philippines
OPEN UNIVERSITY

Data Structures and Algorithms
By Eliezer A. Albacea

Published in the Philippines by the UP Open University
Office of Academic Support and Instructional Services
2/F ICTO/UPOU Building
C.P. Garcia Avenue, Diliman, Quezon City 1101
Telephone (632) 426-1515
Email oasis@upou.edu.ph

ISBN 971-767-158-3

First Printing, 1997
Second Printing, 2005

Layout by Cecilia G. Santiago

Printed in the Philippines

About the Author

Dr. ELIEZER A. ALBACEA finished his PhD in Computer Science from the Australian National University, his MSc (Honours) in Computing Science from the University of Wollongong, and his BS in Statistics from the University of the Philippines Los Baños.

He is a Professor of Computer Science, currently the Director of the Institute of Computer Science, UPLB and the Director of the University of the Philippines System Computer Science Program.

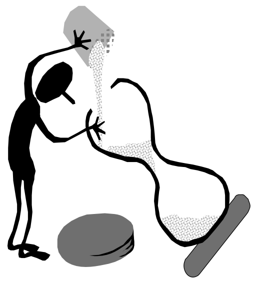
He is the author of at least 40 technical papers published in international and national journals and in proceedings of international and national conferences.

He is a member of the Technical Panel for Information Technology Education of the Commission on Higher Education (CHED), a member of the Technical Panel on Information Technology of the Department of Science and Technology (DOST), and a member of the Science and Technology Strategic Thinkers of DOST.

Among the awards he received are: NAST Outstanding Young Scientist Award, NRCP Achievement Award for Mathematical Sciences, DOST Eduardo A. Quisumbing Outstanding Research Development Work (Basic Research Category), UP System C.B. Garcia and C.C. Garcia Annual Award for Excellence, and UPLB Outstanding Teacher Award.

💻	Email:	eaa@ics.uplb.edu.ph
☎	Phone:	(63-49) 536-2313
✆	Fax:	(63-49) 536-2302
💻	URL:	http://www.uplb.edu.ph/~eaa

Suggested Schedule



Week	Module: Topic
1	Module 1: Arrays
2	Module 2: Linked Lists
3	Module 3: Stacks
4	Module 4: Queues
	Study Session
5	Module 5-6: Binary Trees, Binary Search Trees and AVL Trees
6	Module 7: Heaps
7	Module 8: Hashing
8	Module 9: Data Structures for Graphs
	Answer Sample Examination 1
	Study Session
9-10	Module 10: Complexity Analysis
11-12	Module 11: Simplifying Summations
	Study Session
13-14	Module 12: Simplifying Recurrences
15-16	Module 13: Analysis of Sorting Algorithms
	Answer Sample Examination 2
	Study Session

Table of Contents

Preface

This text is intended to introduce data structures and algorithms to students who have already taken an introductory course in programming. Hence, it assumes that the student has knowledge of at least one high-level programming language like Pascal or C. In order to avoid being programming language specific in presenting the algorithms, the pseudocode format was used. However, the said pseudocodes can easily be implemented in Pascal or in C. Also, the students, tutors, and faculty incharge of this course should note that the algorithms presented in this manual cater only to the general cases. Hence, special cases are not handled by the algorithms. The tutors and faculty incharge are encourage to give as exercises the special cases not covered by the algorithms.

The topics can be classified into two broad topics, namely: introduction to data structures and introduction to analysis of algorithms. First, concepts in data structures are introduced and these are used in the design of efficient algorithms. Second, the algorithms formulated using existing data structures are analyzed in terms of running time. Only the basic techniques of doing analysis of algorithms are introduced. The students are expected to take another course on Design and Analysis of Algorithms where instead of concentrating on how the algorithms work, the concentration is on analyzing the running times and memory requirements of algorithms.

The Self-Assessment Questions (SAQ's) are designed to test your understanding of the topics discussed. Aside from the SAQ's which are provided with answers in Appendix 1, a set of Supplementary Self-Assessment Questions (SSAQ) are provided in Appendix 2. This set, however, is not provided with solutions.

Several people have to be thanked for allowing me to write this book. First is the duo John Patrick and Zita for allowing me some time at night to scribble some of the pages in this book. Many thanks to the Diploma in Computer Science students of CMSC D: Data Structures and Algorithms, 1st Semester 1996-97 and 2nd Semester 1996-97 for being the experimental classes where the topics in this text were tested. Margarita Carmen S. Paterno for acting as reader of this manual. Ana Minela M. Rada for spending a lot of time trying to reformat the original version of this manual to the UPOU format.

If you spot any error (typographical, on content, or grammar) or you may want to make a comment on this material, it would be appreciated if you can report this to the author through his email address eaa@ics.uplb.edu.ph.

Unit I
Introduction to Data Structures

The study of data structures involves the identification and develop ment of useful mathematical entities and operations and the determination of classes of problems that can be solved by these entities and operations. Another area of interest is the determination of representation of these abstract entities and the implementation of the operations on these representations.

A **data structure** is defined as a collection of related data and a set of rules for organizing and accessing it. For example, the list of members of a family is a simple data structure. The choice of data structure obviously affects the operations that can be done on the data. In the list of family members, for instance, one can:

a) sort the names alphabetically;
b) determine whether or not a given name is in the family;
c) find the members with the longest and shortest names.

However, determining the relationship of any two members of the family is impossible unless one defines another list containing the position of each member in the family. If the operation of determining relationships between two family members is important, then a different data structure may be needed. A family tree data structure might be appropriate.

In this unit, we cover the different data structures used in the design of algorithms. The objective is to arm you with options in designing efficient algorithms for some computing problems. At the end, you should learn the basic characteristics of problems where a particular data structure is most suited.

The area of data structures is quite wide. It is not the intention of this unit to introduce all of them. What will be covered here are only those that are used quite often in popular computing problems. Studying on your own the remaining data structures will be easy once you learned those that are covered in this course.

You are expected to spend half of the semester in this unit.

Module 1
Arrays

In this module, we will review you on the array structure. I am sure that this topic has already been introduced in CMSC B (Principles of Programming). Our concentration, however, will be on the operations that can be done on an array structure. In particular, we will look at how to search for a value in this structure. As we know, arrays together with some simple abstract data types like character, integer, real, are usually incorporated in most programming languages. We shall not discuss these other data types anymore as they have already been discussed in detail in CMSC B.

Since arrays are used heavily in the implementations of other data structures in later modules, I suggest that you do not leave this topic unless you are convinced that you can properly use array structures.

Objectives

At the end of this module, you should be able to:

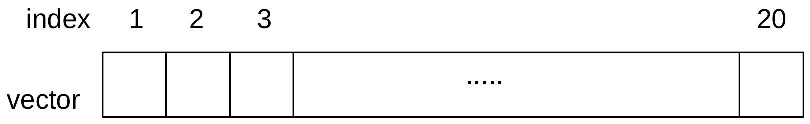
1. Design algorithms for searching, deleting and inserting new elements into the basic data structure arrays; and
2. Solve programming problems involving the use of arrays.

Arrays

An **array** is a structure consisting of a fixed number of components with each component of the same type. The primary characteristics of an array are its name, component type, and indices of the first and last components. There are two types of arrays that are often used in real problems. The simplest is **one-dimensional** array and the generalization of this is the **multidimensional array**.

One-dimensional Array

One-dimensional arrays are referenced by the **array name** and an **index value**. In memory, a declared array vector with 20 components will look like the following:



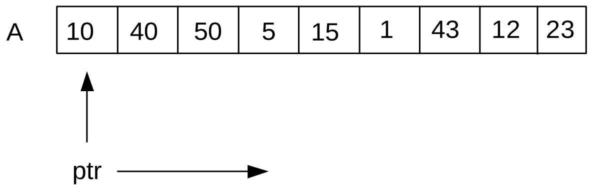
Memory map of array vector

The type of the component may be any of the basic data types defined in the language, e.g., character, integer, real, and boolean in Pascal. An **element** of the array found in position *p* where *p* is any value from 1 to 20 can be directly accessed by using the identifier *vector[p]*. For example, the second element of the array can be referenced by the identifier *vector[2]*.

One advantage of the array is that each element found in the array can be accessed directly. To illustrate this, we consider the problem of searching the array for an element. Search algorithms determine whether or not any given element x exists in a set of data which is not strictly arranged in any order. Two methods will be discussed herein, the sequential search and the binary search. The algorithms are strictly boolean functions, because we are only interested in determining the existence of the element to be searched in the data set.

Searching for an Element in an Array

One obvious solution to searching an element in an array is to use a **pointer** and set the value of this initially to the first element. The value of this pointer is increased by 1 until the element is found or the end of the array is reached. This is called **sequential search**. To illustrate, consider the array in the diagram next page.



To search for x = 43, *ptr* (the pointer used) is initially 1. Since 43 is in the array, the search stops when the value of *ptr* is 7.

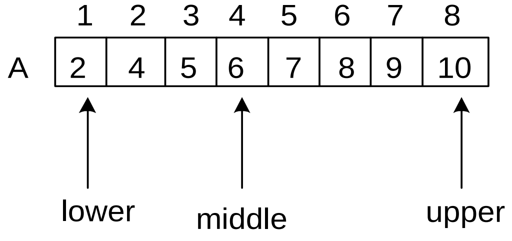
```
SequentialSearch(A,x)
begin
   ptr = 0;
   repeat
      ptr = ptr + 1;
   until (A[ptr] = x) or (ptr = n);
   if (A[ptr] = x) return(TRUE)
   else return(FALSE)
end
```

The procedure just outlined works for both sorted and unsorted data. However, you can search an element in the array more efficiently when the data is already sorted.

One algorithm for doing a search on a sorted data set is **binary searching**. In binary searching, the data is "halved" at each step of the execution — meaning, the midway observation of the section of the array to be searched (or the whole array, initially) is used to compare with the value of the element being searched. If the element being searched is greater than the midway element, then you can be sure that the element (if ever it is in the array) may exist at the right of the midway element - never to the left, if the array is sorted in increasing order. Searching is to be done again, but this time, on the section (which is already "halved") where the observation may possibly exist.

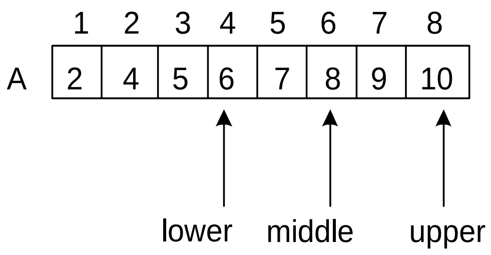
The process of locating the midway element in the constantly "halved" section continues until the observation is found or the index of the leftmost element of the subarray exceeds the index of the rightmost element of the subarray.

As an example, consider the array A and the problem of searching for x = 7 in A.



Set the pointers initially at the beginning, end, and middle of the array. With array size n = 8, the middle element index is computed as (lower + upper) div 2, which is (1 + 8) div 2 = 4. So, middle = 4. Since 7 (the element x to be searched) is greater than A[4] (which is 6), x does not exist to the left of A[4]. The section to be searched is therefore reduced to that part starting from (middle + 1) to upper. Hence, we can set lower to (middle+1).

Repeat the process, i.e., recompute the middle and then compare the new middle element with the value of x.



The new middle value would be (5 + 8) div 2 = 6. Since, 7 < A[6], we modify our upper value to that of the middle value. The upper value is now equal to 5.

Upon computing middle (middle = (5 + 5)div 2 = 5), we find that A[5] = 7, and the element has been found. If the element searched is nowhere to be found in the array, the lower value would get to exceed the upper value.

```
BinarySearch(A,x)
begin
  lower = 1;
  upper = n;
  repeat
    middle = (lower + upper) div 2;
    if x > A[middle] lower = middle + 1;
     else upper = middle - 1;
  until (A[middle]=x) or (lower > upper);
  if (A[middle] = x) return(TRUE)
  else return(FALSE)
end
```

All we need to be able to use arrays in solving problems is an understanding of how the search is carried out. To test your understanding of the topic, I am providing several problems specific to one-dimensional arrays. When the problem is silent on the size of the array, we assume a general value ***n*** for it. Also, the problems require you to write a procedure or formulate an algorithm. The two, procedure and algorithm, are similar (you will learn later that the two are not really exactly the same). Anyway, write your solution in pseudocode form.

If the problems seem difficult, do not despair. Maybe you are not yet used to the style of how the questions are formulated. However, try to understand the question and learn the style of how questions in data structures are formulated. After doing this, try to answer the question. Please try SAQ 1-1 now.

You may be at a loss as to how and where to start. Okay, I will allow you to look at the ASAQ in Appendix 1 for SAQ 1-1, which is reproduced here after the SAQ's. Do this only for SAQ 1-1 and not for the others. I will allow you to do this so you will have an idea of how to proceed.

SAQ 1-1

1. Write a procedure for reversing the elements of an array.
2. Write a procedure for merging two sorted arrays into one.
3. Formulate an algorithm for inserting a new element x into a sorted array A of size n.
4. Write a procedure for finding the minimum in an array and deleting it from the array.

Answers to Self-Assessment Questions (ASAQ) are found in Appendix 1.

Multidimensional Arrays

Although the memory of a computer is one-dimensional, a user can define and manipulate directly a **multi-dimensional array**. The idea is that most programming languages provide this facility directly and the mapping of multi-dimensional arrays to one-dimensional arrays is implemented transparently from the user.

For example, consider the following 3 x 2 matrix M,

$$M = \begin{array}{c|c|c|c} & 1 & 2 & \\ & 11 & 22 & 1 \\ & 33 & 44 & 2 \\ & 55 & 66 & 3 \end{array}$$

where M[1,1] = 11, M[1,2] = 22, M[2,1] = 33, etc. The first index indicates the row position and second index indicates the column component. As in one-dimensional arrays, the elements of a multi-dimensional array can be accessed directly.

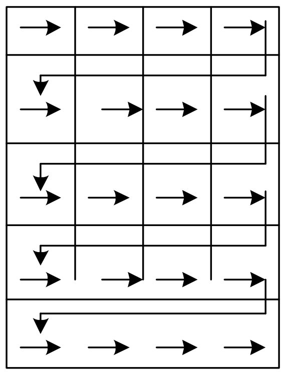
Searching A Two-Dimensional Array

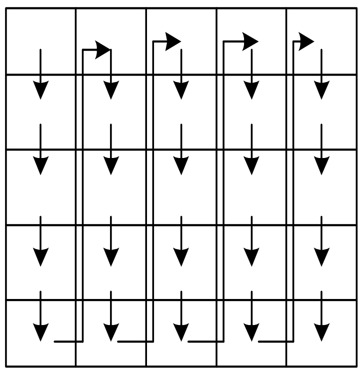
There are several ways of searching two-dimensional arrays. One is to increase the row number faster than the column number. Another is to reverse the process, i.e., increase the column number faster than the row number.

Consider a square matrix M with n rows and n columns. This can be searched using the following algorithm:

```
SearchA2DArray(M,x)
begin
    found = FALSE
    for i = 1 to n do
       for j = 1 to n do
          if A[i,j] = x then found = TRUE
    return(found)
end
```

If *i* is the index for the row, then the algorithm searches the matrix horizontally going from top to bottom. But if **i** is the index for the column, then the algorithm searches the matrix vertically going from left to right.


Row index is *i*


Column index is *i*

When the rows and/or columns are sorted, binary search could be employed to speed up the search.

As in the one-dimensional arrays, a set of SAQ's are provided to test your understanding of multi-dimensional arrays. Answer all of them. In data structures, each problem you solved will add to your experience which can later be recalled to form part of a solution to a much larger problem. Hence, we are trying to build this experience through the SAQ's.

From among the SAQ's, SAQ 1-2 number 1 requires a more extensive solution. Hence, I suggest you do this after the others.

SAQ 1-2

1. Represent the moves of a tic-tac-toe game using multidimensional arrays. Formulate an algorithm for checking whether a move by one player will result to a win, a draw, or a situation where the other player is required to make his move.

2. Given a square matrix which is stored in a two-dimensional array. Check if the matrix is symmetrical about the main diagonal.

3. Given a two-dimensional nxn matrix M. Check if M is triangular, i.e., every element below the main diagonal is zero.

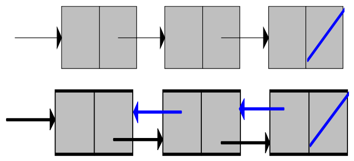
4. Given an nxn matrix A and a nxn matrix B. The matrix product AxB is an nxn matrix C, where the element C_{ij} of C is given by:

$$C_{ij} = \sum_{1 <= k <= p} A_{ik} B_{kj}$$

Write a procedure for multiplying any two matrices A and B.

ASAQ's are found in Appendix 1. Supplementary Self-Assessment Questions (SSAQ) are given in Appendix 2. SSAQ's, however, are not provided with answers.

Module 2
Linked Lists



Objectives

At the end of this module, you should be able to:

1. Design algorithms for searching, deleting and inserting new elements into the linked list structure; and
2. Solve programming problems involving the use of linked lists.

One major drawback of arrays is that a fixed amount of storage remains allocated even when only a small fraction of this is being used. Moreover, the amount of storage is fixed by the declaration, thus the possibility of an overflow (a case of more data need to be stored in a fixed number of storage in the array). An alternative data structure that specifically addresses these two problems is the **linked list**.

As in arrays, our discussion will focused on the operations that can be done on linked lists. We assume that the student already knows how to create a linked list.

Linear Linked Lists

The simplest among the different implementation of linked lists is the **linear linked list**. Each item in the list is called a **node** and contains two fields: **information field** and **pointer to the next node field**. The information field contains the relevant information being stored in the structure, while the pointer to the next node contains the address of the next node. The entire linked list is accessed from an external pointer (usually called the **list pointer**), which points to the first node of the linked list. The next node field of the last node contains a special value called nil. The **nil pointer** signals the end of the linked list. A linked list with no element is usually called a **nil list** (can be created by the statement **list = nil**). The first element of the list is called the **head** and last node is called the **tail**.

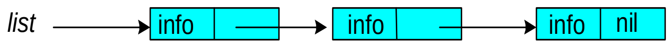


Figure 2-1. A linear inked list

To simplify the discussions of the operations on linked list, we consider some notations. Let

ptr	- the pointer to a node,
node(ptr)	- node pointed to by *ptr,*
info(ptr)	- information in the node pointed to by *ptr*,
next(ptr)	- address of the next node.

Therefore, if *next(ptr)* is not *nil*, *info(next(ptr))* refers to the information part of the next node that follows *node(ptr)* in the list.

To create a node, a special operation **getnode** is usually defined in most programming languages. Hence, to create a node pointed by *ptr* the statement

ptr = getnode

is sufficient. However, if no memory is available then getnode will return a nil pointer. To set the information field of the new node, the statement

info(ptr) = value

may be issued. To free the space being occupied by a node pointed by *ptr*, a **freenode** operation is usually provided together with the **getnode** operation. Hence, to free the space pointed by *ptr*, the statement *freenode(ptr)* may be issued.

Searching for an Element in a Linear Linked List

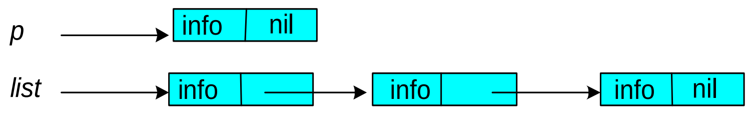
Unlike in an array where every element can be referenced directly, the elements of a linked list can be accessed only by following the link starting from the first node to the node being searched if this exists and to the end of list if the node being searched is not in the list. Hence, searching for an element can only be achieved using an algorithm similar to the sequential search algorithm. Binary searching a linked list will not result in a more efficient algorithm. In fact, searching a list using the binary search method will be slower than searching it sequentially.

```
SearchAList(list,x)
begin
   ptr = list;
   while ptr ¹ nil and info(ptr) ¹ x do
      ptr = next(ptr);
   if ptr ¹ nil return(TRUE)
   else return(FALSE)
end
```

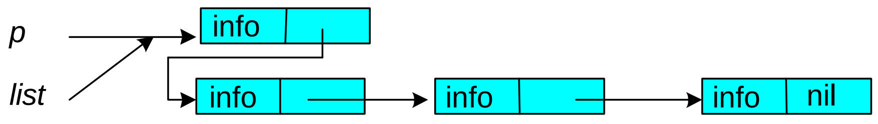
Inserting a Node into the List

There are three places where a node can be inserted into a list. These places are the front, the end, and in between two existing nodes in the list. Consider a list pointed to by *list* and a new node pointed to by *p*.

To insert a node in front of the list, this means making the new node the head of the list. Here, we have to consider two cases: case where the list is initially empty and the case where it is not initially empty. For the case where it is initially empty we simply make the list pointer point to the new node to be inserted. For the case where the list is not initially empty, it is illustrated in the diagram below.



Before

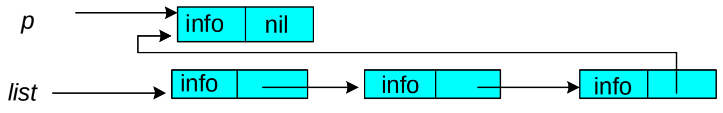


After Inserting at the Front of the List

Though we have two cases, the implementation of the problem of inserting a node at the front of the list can be done cleanly as follows:

```
InsertatFront(list,p)
begin
   next(p) = list;
       list = p;
end
```

Next, we consider inserting a node at the end of the list. Again, we will be encountering two cases: a case where the list is initially empty and a case where it is not initially empty. If the list is initially empty, we simply make it point to the node to be inserted. Otherwise, do it as shown in the diagram below.



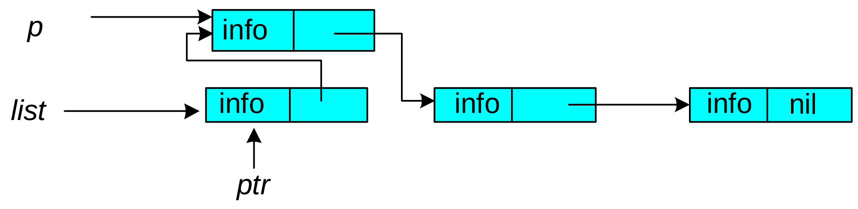
After Inserting at the End of the List

The implementation of this is not as clean as when inserting at the front of the list. In the implementation, you must have noticed the clear handling of the two special cases.

```
InsertatEnd(list,p)
begin
   if list = nil list = p
   else
      ptr = list;
      while next(ptr) ¹ nil do
         ptr = next(ptr);
      next(ptr) = p;
end
```

Finally, we consider inserting in between two nodes. In this situation, we will assume that the two nodes where we want to insert in between them do exist. This is because if there is only one node, then this is similar to either inserting at the head or tail of the list. The case where the list is empty can still be handled as when inserting at the head or tail.

An implementation of this is shown in the diagram below.



After Inserting at any Two Nodes in the List

Let *ptr* be the address of the node after which the node pointed by *p* is to be inserted. The algorithm for doing the insertion is given below.

```
InsertinBetween(list,p,ptr)
begin
    next(p) = next(ptr);
    next(ptr) = p;
end
```

After considering the special cases in the insert operation, let us combine these special cases into one. In some problems, insert is preceded by a search. This means we are provided with a position in the list where we will be inserting the new element. With a particular position where to make the insert already known, there are still two choices: insert before the given position or after the given position. The easier case is to insert after a given position. If the given position is nil, we insert just before the head of the list. Otherwise, we insert it right after the node in the given position.

Let the list be pointed by *list*, the position where we insert be pointed by *ptr*, and the node to be inserted be pointed by *p*. The algorithm for a generalized insert is given below.

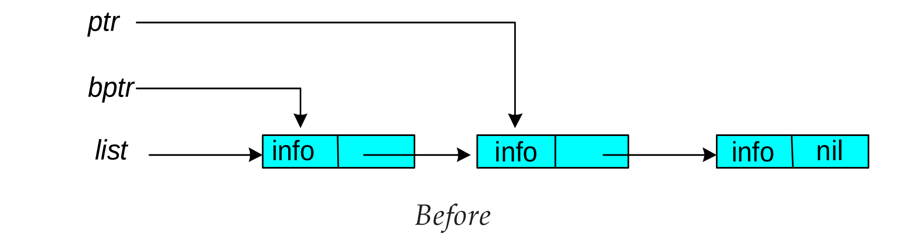
```
Insert (list,p,ptr)
begin
    if ptr = nil
        next(p) = list
        list = p
    else
        next(p) = next(ptr)
        next(ptr) = p
end
```

Deleting a Node from the List

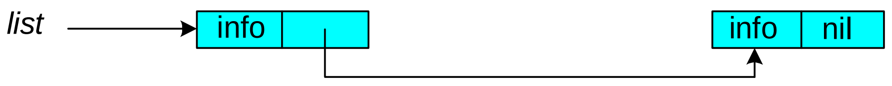
The delete operation is usually preceded by a search operation. Hence, when doing a delete we already have a pointer to the node to be deleted. This node to be deleted will again be found in several places in the list - thus defining again several cases. One case is when the node to be deleted happens to be at the head of the list and another when it is not the head of the list. What is crucial to the delete operation is actually the node positioned before the node to be deleted. Hence, a pointer to this node must be available to the delete operation.

Let the list be pointed by *list,* the node to be deleted be pointed by *ptr*, and node before the node to be deleted be pointed by *bptr*. When *bptr* is nil this means that the node to be deleted is the head of the list.

```
DeleteNode(list,bptr,ptr)
begin
   if bptr = nil
      list = next(ptr)
      freenode(ptr)
  else
      next(bptr) = next(ptr);
      freenode(ptr);
end
```

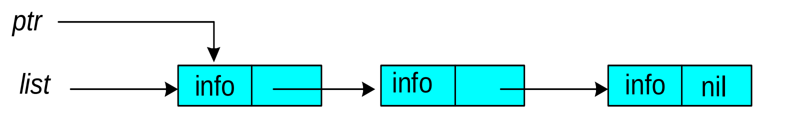


Before

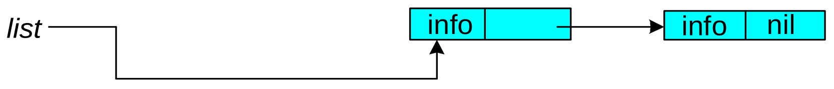


After

Case: bptr ≠ nil



Before



After

Case: bptr = nil

Now, you are ready for some SAQ's on linear linked lists. The ASAQ's for these SAQ's are given in Appendix 1. I suggest you try answering some more questions on linear linked list, found in Appendix 2. I should warn you though that the additional problems are not provided with answers.

SAQ 2-1

1. Let L1 and L2 be two sorted linear linked lists. Write the procedure Union(L1, L2). The procedure produces the union of the elements in L1 and L2, also in sorted order. The definition of union in this case is similar to the definition of union of sets.

2. Given a list and a pointer (ptr) to one of the elements of the list. Write a procedure Swap(ptr, next(ptr)) that swaps the nodes pointed by ptr and next(ptr). The swap should be done without touching the information stored in the node, i.e., only the pointers will be manipulated. Do this for a linear linked list.

ASAQ's are found in Appendix 1. SSAQ's are given in Appendix 2.

Circular Lists

One of the main problems of linear linked lists is that nodes preceding a particular node are not accessible anymore unless a pointer to the first node is maintained throughout. This problem is solved when you provide a pointer from the last node back to the first node. This defines the **circular list** structure. In this arrangement, the external pointer can be pointing at any node with each node still accessible (through a search operation). A circular list structure therefore has no explicit first and last nodes. However, we can view the node pointed to by the external node as the first node and the node preceding it as the last node.

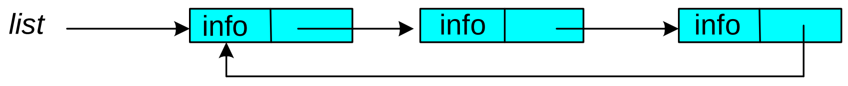


Figure 2-2. A circular linked list

Primitive Operations on Circular Lists

As in linear linked lists, three basic operations are defined for circular linked list. These operations are: search, insert and delete. We discuss how these operations are implemented below.

Searching for an element in a circular list is almost similar to that of linear linked list except for the termination condition. In circular list, the search stops when the element being searched is found or when the pointer has gone one full circle in the circular list. This is illustrated in the code for search below.

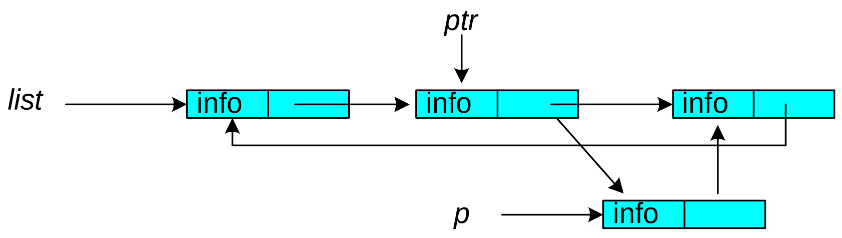
```
Search(list,x)
begin
  if list = nil return(FALSE)
  else
    ptr = list
    while next(ptr) ¹ list and infor(ptr) ¹ x do
       ptr = next(ptr)
    if info(ptr) = x return(TRUE)
    else return(FALSE)
end
```

From the implementation, note that when the list is initially empty then it immediately return FALSE (i.e., the value being search is not found in the list).

Next, let us consider how the insert operation is done. This operation, as mentioned earlier, is usually done after a search is done. The output of the search is the position where the node to be inserted is to be placed. If the list is not empty, then we usually insert the node after the node returned by the search operation. Otherwise, the node to be inserted simply becomes the first node of the list. An implementation of this is given below.

```
Insert(list,ptr,p)
begin
   if list = nil
      list = p
      next(p) = list
   else
      next(p) = next(ptr)
      next(ptr) = p
end
```

An illustration of this algorithm for the case where the list is not empty is given in the diagram below.



Finally, let us look at how the delete operation is implemented. Similarly, this is usually done after a search is made to determine the node to be deleted. Aside from the node to be deleted, we will require the search operation to return a pointer to the node preceding the node to be deleted. We assume if the list has only one node the pointer to the node preceding the node to be deleted is nil.

```
Delete(list,ptr,bptr)
begin
   if list ¹ nil
      if bptr = nil list = nil
       else next(bptr) = next(ptr)
       freenode(ptr)
end
```

Now, try and answer one SAQ for circular linked list. As in the previous SAQ's, the problem assumes a linked list of size *n*. There is another more challenging problem given in the SSAQ (Appendix 2). Try it!

SAQ 2-2

Given two circular linked lists L1 nd L2. Formulate an algorithm for merging the two circular linked lists.

ASAQ's are found in Appendix 1. SSAQ's are given in Appendix 2.

Doubly Linked Lists

Although a circular linked list can solve some problems that cannot be solved by linear linked lists, there are still problems that cannot be solved by both structures. Two of these problems are:

1. traversing the list backwards; and
2. deleting a node given only a pointer to that node.

The appropriate data structure for solving these problems is the **doubly linked list**. In a doubly linked list, each node has two pointers: one pointing to the next node **(right(ptr))** and the other pointing to the preceding node **(left(ptr))**.

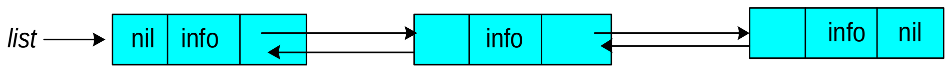


Figure 2-3. A doubly linked list.

Primitive Operations on Doubly Linked Lists

As in the previous structures, we will consider the operations search, insert and delete.

The search in a doubly linked list is actually similar to searching a linear linked list. The pointer to the left node, however, will remain unused during the search, i.e., only the pointer to the right node will be involved in the search.

For the insert operation, there are actually two places where the node to be inserted can be inserted. It can either be inserted at the right or left of a node identified as the point of insertion. Let us tackle the problem of inserting at the right. Inserting at the left is similar. Again, we will consider two cases: a case where the list is initially empty and a case where it has at least one element. The node identified as the point of insertion will again be supplied by a search operation that has to be called before the insert is done.

```
InsertRight(list,ptr,p)
begin
   if list = nil list = p
   else
      if right(ptr) $^1$ nil left(right(ptr)) = p
      right(p) = right(ptr)
      right(ptr) = p
      left(p) = ptr
end
```

An illustration for the case where there is at least one node in the list is given in the diagram below.

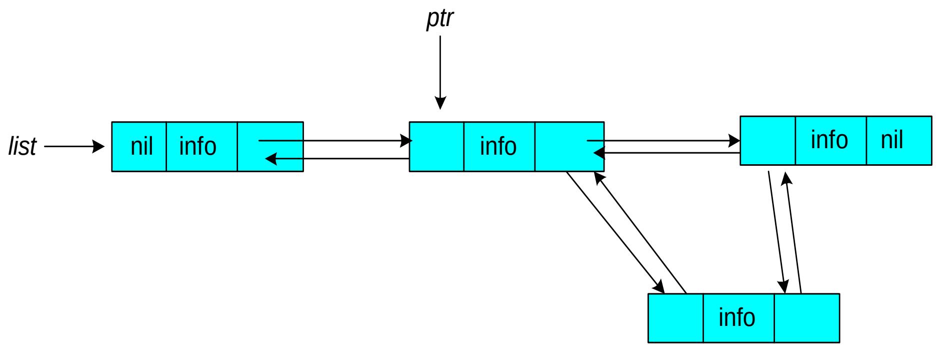


Figure 2-4. Insert right

Finally, let us consider the delete operation. As mentioned earlier, all we need to delete a node from a doubly linked list is a pointer to the node to be deleted. Let *ptr* points to the node to be deleted. In the implementation below, we note that there are several cases:

1. the list is empty
2. there is only one node in the list
3. the node to be deleted is the tail of the list
4. the node to be deleted is the head of the list
5. the node to be deleted is in between two legitimate nodes.

All these cases are handled by the code below.

```
Delete(list,ptr)
begin
  if list ≠ nil
     if left(ptr) = nil and right(ptr) = nil list = nil
     if left(ptr) ≠ nil and right(ptr) = nil
   right(left(ptr)) = nil
if left(ptr) = nil and right(ptr) ≠ nil
   left(right(ptr)) = nil
   list = right(ptr)
if left(ptr) ≠ nil and right(ptr) ≠ nil
   right(left(ptr)) = right(ptr)
   left(right(ptr)) = left(ptr)
     freenode(ptr)
end
```

To test your understanding of the operations involved in doubly linked lists, try the SAQ. Again, there are some questions related to doubly linked list in Appendix 2.

SAQ 2-3

Given a list and a pointer (ptr) to one of the elements of the list. Write a procedure Swap(ptr, next(ptr)) that swaps the nodes pointed by ptr and next(ptr). The swap should be done without touching the information stored in the node, i.e., only the pointers will be manipulated. Do this for a doubly linked list.

ASAQ's are found in Appendix 1. SSAQ's are given in Appendix 2.

Circular Doubly Linked Lists

This is just a doubly linked list with the last node having a forward pointer to the first node and the first node having a back pointer to the last node. This is to take advantage of the properties of circular linked lists. As in the circular linear linked list, in this structure there is no more explicit first and last nodes.

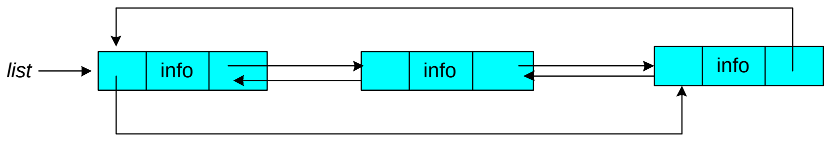
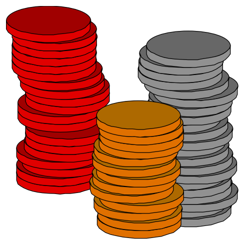


Figure 2-5. A circular doubly linked list

Circular doubly linked list simply combines the advantages of circular lists and doubly linked lists. Hence, the operations for this data structure are similar to those in the circular lists and doubly linked lists.

Module 3
Stacks



One of the data structures commonly used in simulation studies is the stack. A **stack** is a one-dimensional data structure in which values are entered and removed one item at a time at one end, called the **top** of stack. A stack operates on a **last-in, first-out** basis, and is therefore sometimes called a **LIFO** data structure. In this module, we introduce the basic operations defined on a stack. Unlike arrays which is readily available in almost all programming languages, stacks have to be implemented using arrays or linked lists. As in the modules for arrays and linked lists, the discussion will concentrate on the operations on the stack structure.

Stack Structure

A **stack** consists of a block of memory and a variable denoted by **top** which basically points to the top of stack. The top of stack pointer may point to the next available space on the stack or to the element found at the top of stack. As to which of this is adopted depends on the implementation. Just like in the data structures discussed earlier, several operations are defined on the stack data structure. In the case of a stack, two operations are defined: the **pop** an element and **push** an element operations.

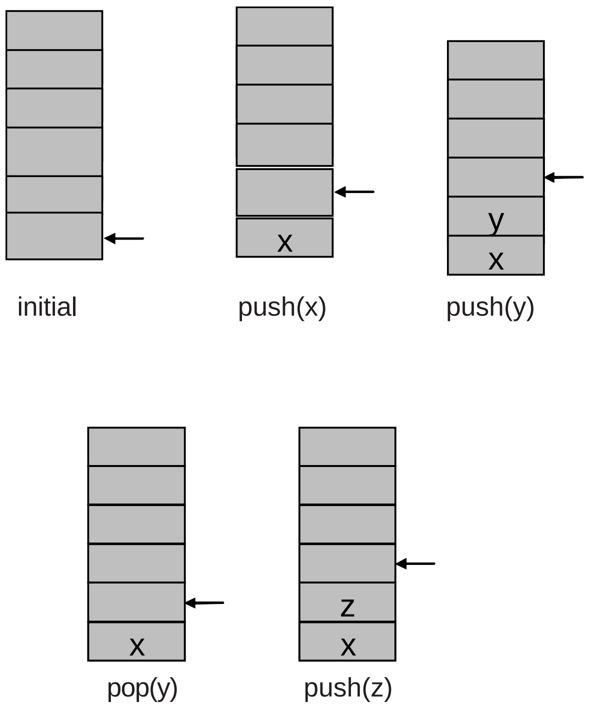
Objectives

At the end of this module, you should be able to:

1. Design algorithms for pushing into and popping elements in a stack; and
2. Solve programming problems involving the use of stacks.

The push operation stores the datum at the top of the stack and updates the stack pointer, while the pop operation backs up the top of stack pointer by one memory location and removes the datum stored there. To illustrate the push and pop operation, we show the execution of the following:

push(x); push(y); pop(y); push(z)



Some errors may occur while doing some operations on the stack. These errors are the stack underflows and overflows. An **overflow** occurs when one pushes so many items and the size of the stack is not enough. An **underflow** occurs when one tries to pop something from an empty stack.

The stack data structure may be implemented using an **array** or using a **linked list**.

Array Implementation of a Stack

When implementing the stack data structure using arrays, we can interpret the top of stack pointer *top* as pointing to the next available space in the array. The declaration on the stack may be given by the following statements:

```
const
    stacksize = 100;
var
    Stack : array [1..stacksize] of value-type;
    top    : integer;
```

The beginning of any program using the stack must have the stack initially empty and this can be done by initializing *top* to 1. The operations push and pop can be implemented as follows:

```
Push(Stack,x);
begin
    if top £ stacksize
        Stack [top] = x;
        top   =  top + 1;
        else
        error("Stack overflow");
end;
```

```
Pop(Stack);
begin
    if top £ 1 error("Stack underflow");
   else
       top = top - 1;
       return(Stack[top]);
end;
```

Linked List Implementation of a Stack

In a linked list implementation, the most appropriate interpretation of the pointer *top* is to make it point to the last node pushed. With this interpretation and the fact that the pop operation deletes the node being pointed to by top and the push operation inserts a new node on top of the stack, the pop and push operations are equivalent to the following operations in the linear and doubly linked lists.

Stack		Linear Linked List		Doubly Linked List
Push	≡	InsertatFront	≡	InsertLeft
Pop	≡	DeleteFirstNode	≡	Delete.

Let us show how to implement a stack structure using a linear linked list. As mentioned above, a push is equivalent to inserting a node at the head of the list and a pop operation is similar to deleting and returning the head of the list. Unlike in the array implementation where the size of the stack is fixed, in the linked list implementation, the size of the stack depends on the availability of memory. Hence, once *getnode* returns nil this indicates that we run out of memory for the stack and thus an overflow message will come out. The underflow on the other hand will occur when popping from an empty list.

Let *top* be the pointer to the top of stack.

```
Push(top,x)
begin
   ptr = getnode
   if ptr $^1$ nil
      info(ptr) = x
      next(ptr) = top
      top = ptr
   else error("Stack overflow")
end
```

```
Pop(top)
begin
   if top $^1$ nil then
      x = info(top)
      ptr = top
      top = next(top)
      freenode(ptr)
      return(x)
   else error("Stack underflow")
end
```

To check your understanding of the implementation and applications of stacks, try the SAQ's below. SAQ 3-1 is in the category of a difficult question. Try it anyway! It is a good exercise. Compare your answers with the ASAQ's in Appendix 1.

SAQ 3-1

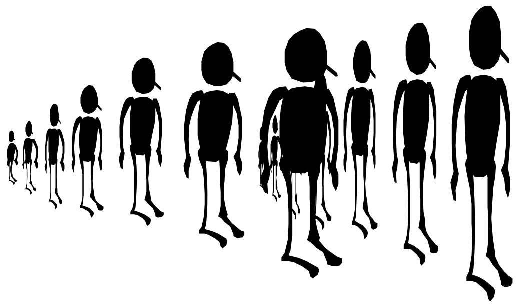
1. Two stacks S1 and S2 can be implemented using one array. Write procedures: PopS1(Stack), PushS1(Stack,x), PopS2 (Stack), PushS2(Stack,x). The procedures should not declare an overflow unless every slot in the array is used.

2. Formulate an algorithm for printing the elements of a stack in reverse order.

3. Formulate an algorithm for simulating a parking lot that operates on a last-in, first-out basis (LIFO). The data structure used is the stack due to the nature of the operations described.

ASAQ's are found in Appendix 1. SSAQ's are given in Appendix 2.

Module 4
Queues



Objectives

At the end of this module, you should be able to:

1. Design algorithms for inserting and deleting elements in a queue and a dequeue; and
2. Solve programming problems involving the use of queues.

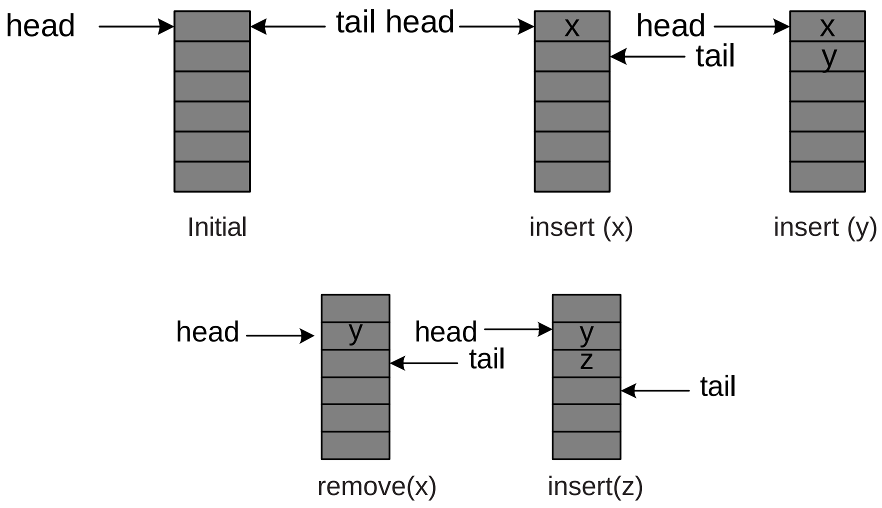
The stack operates on a last-in first-out manner. The reverse of this structure is one that operates on a **first-in, first-out** basis. Hence, corresponding to the **LIFO** data structure is the **FIFO** data structure. One such FIFO data structure is the **queue**. As in stacks, queues are heavily used in simulation applications where the server attends to those who arrive at the system first.

Queue Structure

In a queue structure the values are entered in one end called the **tail** and are removed from the other end called the **head**.

In implementing a queue structure, two pointers are needed, one pointing at the head and the other pointing at the tail of the queue. Using **head** and **tail** as pointers, a series of operations on a queue can be defined. The most basic operations on queue structures are the **insert** and **remove** operations. The insert operation stores the information at the tail of the queue, while the remove operation removes the information found at the head of the queue. To illustrate these two operations, we show how the following operations are executed.

insert(x); insert(y); remove(x); insert(z)



As in the stack data structure, this structure can be implemented using **arrays** or using **linked lists**.

Array Implementation of Queues

In order to implement the operation on queues using arrays as illustrated above, we have to modify the definitions of head and tail. Let head be pointing to the available space before the head item and let tail be pointing to the tail item. With this, the queue structure can be declared as follows:

```
const
    QUEUESIZE = 100;
  var
     Queue : array[1..QUEUESIZE] of value-type;
     head, tail  : integer;
```

and the beginning of the program should initialize the head and the tail to its maximum value (actually it does not matter what the initial values of head and tail as long as they are equal), i.e.,

```
begin
        tail = QUEUESIZE
        head = QUEUESIZE
             . . . .
```

We have to mention that in using arrays, it is possible that as we continue inserting or deleting elements we might reach the end of the array. In this case, we simply wrap around and start inserting or deleting at the other end. Also, we have to note that when the head and tail are equal, this could either mean the queue is empty or the queue is full. You will notice this in the code below.

```
Insert(Queue, x)
begin
   if tail = QUEUESIZE tail = 1
   else tail = tail + 1;
   if tail ≠ head Queue[tail] = x
   else error("Queue overflow")
end
```

```
Remove(Queue)
begin
   if head = tail error("Queue underflow")
   else
      if head = QUEUESIZE head = 1
      else head = head + 1
      return(Queue[head])
end
```

Linked Lists Implementation of Queues

As in the stack data structure, queues can be implemented using the **linear linked lists** or using the **doubly linked lists**. The equivalent operations are as follows:

Queue		Linear Linked Lists		Doubly Linked List
Insert	≡	InsertatEnd	≡	InsertRight
Remove	≡	DeleteFirstNode	≡	Delete

Let *head* and *tail* be pointing to the head and tail of the list implementing a queue. Initially, these two will have values nil.

```
Insert(tail,head,x)
begin
   ptr = getnode
   if ptr = nil error("Queue overflow")
   else
      if tail ≠ nil
         next(tail) = ptr
         info(ptr) = x
         tail = ptr
      else
         tail = ptr
         head = ptr
end
```

```
Remove(head,tail)
begin
   if head = nil error("Queue underflow")
   else
      ptr = head
      x = info(head)
      if next(head) ≠ nil head = next(head)
      else
         head = nil
         tail = nil
      freenode(ptr)
      return(x)
end
```

Now, try to apply the concepts of queues on some applications. The SAQ's given below are designed to enable you to apply the concepts. SAQ 4-1 is relatively easy. However, SAQ 4-2 is quite large and is designed to integrate all the operations of queues.

SAQ 4-1

1. Formulate an algorithm for printing the elements of a queue in reverse order.

2. Formulate an algorithm for simulating a parking lot that operates on a first-in, first-out (FIFO) basis. The data structure used is the queue due to the nature of the operations described.

ASAQ's are found in Appendix 1. SSAQ's are given in Appendix 2.

Double-Ended Queue (Dequeue)

A natural extension of the queue structure is the **double-ended queue** structure. If in a queue structure we are allowed to insert at the tail and delete at the head, in a double ended queue delete and insert are allowed both at the head and at the tail. Hence, the following operations are allowed:

InsertatHead(x)
DeleteatHead
InsertatTail(x)
DeleteatTail.

These operations are similar to those in the queue structure.

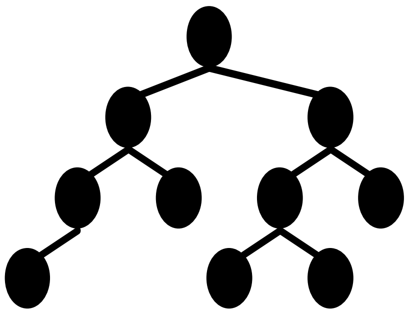
By this time, you should already be familiar with the queue data structure. Try and see if you can apply your knowledge of queues in formulating the operations for double-ended queues. SAQ 4-2 below is designed with this objective in mind.

SAQ 4-2

Formulate the procedures for the dequeue operation using the array implementation: InsertatTai(Dequeue,x), RemoveatHead(Dqueue), InsertatHead(Dequeue,x), RemoveatTail (Dequeue).

ASAQ's are found in Appendix 1.

Module 5
Binary Trees



Objectives

At the end of this module, you should be able to:

1. Use binary trees in solving some tree-based problems; and
2. Formulate recursive solutions to some computing problems.

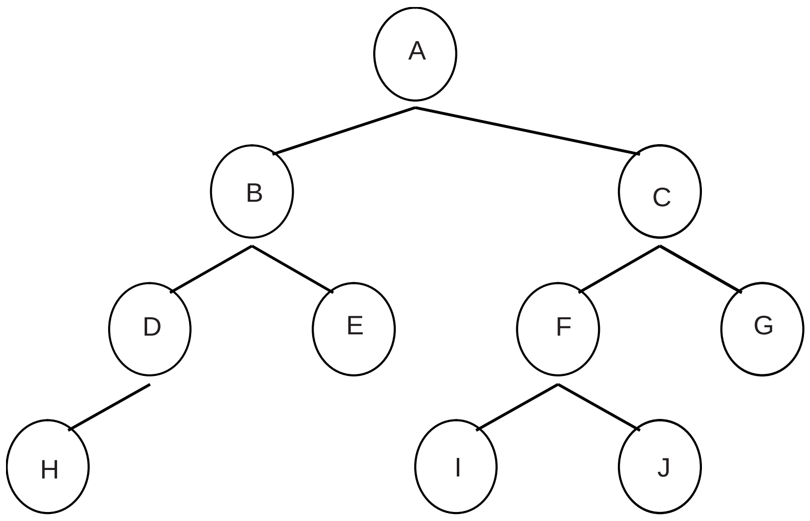
A natural extension of linked lists is to allow a node to have more than one pointer to the next node. The simplest of these extensions is to allow each node to have two pointers. This defines a new data structure called a **binary tree** which has many applications specially in tree-based problems.

In this module, we introduce the different methods of traversing a binary tree. It is important that you understand this module as these topics will be used in other binary-tree based data structures in the succeeding modules. Since most algorithms for traversing a tree are formulated recursively, we also provide as supplementary topic a discussion of recursion.

Binary Trees

A **binary tree** is a finite set of elements that is either empty, or contains a single element called the **root**, or a root with at most two descendants each descendant being a root of binary tree. Each element in the binary tree is called the **node** of the tree. If each node of the tree has either no child or two children, then the binary tree is called a **complete binary**

tree.

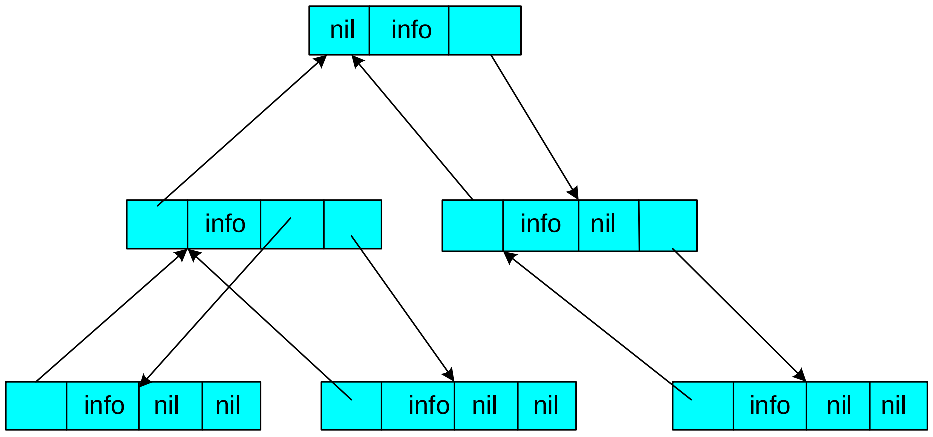


In the example binary tree, A is a special node called the **root**. Each node has at most two children, one of them may be the **left child** and the other the **right child**. For example, D and E are the left child and right child, respectively, of B. B is said to be the father of D and E. Those nodes with no children are called the **leaves** of the tree and those with at least one child are called **internal nodes**. The **level** of a node in a binary tree is equal to 1 plus the level of its father. The root has level 0. This means that the level of B is 1 and that of D is 2. The **height** of a binary tree is the greatest level assignment given to a node in the tree - or, it is the length of the longest path from the root to a leaf.

Level ***i*** of a binary tree is **full** if there are exactly 2^i nodes at this level. A binary tree of height ***k*** is full if level ***k*** in this binary tree is full - level ***k*** being full implies that levels k-1, k-2, …, 2, 1, 0 are all full. A binary tree of height ***k*** is **balanced** if level ***k-1*** in this binary tree is full. Consider node *v* in a binary tree. The binary tree rooted at the left child of *v* is the **left subtree** of *v*, and that rooted at the right child is the **right subtree**. A binary tree is **height balanced** if, at every node in the tree, its left and right subtrees differ by no more than 1 in height.

Each node of the binary tree contains, at least, the **information field**, **left child pointer**, **right child pointer**, and **pointer to its parent**. If a parent or a child is not present, then the value of the pointer is nil.

If a binary tree is complete, then it can be implemented using arrays, otherwise the linked list representation is necessary. In the array representation of a binary tree, the pointers (left child, right child, and parent pointers) can be computed given the index of a node. Specifically, if a node is indexed i, then its left child is indexed 2i, its right child 2i+1, and its parent by $\lfloor i/2 \rfloor$. The root occupying index 1.



In the succeeding discussions on binary trees, we refer to: left(ptr) as the left child of node ptr, right(ptr) as the right child of ptr, and parent(ptr) as the parent of ptr. The info component may be composed of several fields. When it is composed of one filed, we use info itself. However, if it is composed of several fields, then we sometimes use the key field of info to represent whole info.

Binary Tree Traversals

In order to search the binary tree for a certain value, there should be a way of visiting each node in the tree at least once. There are three popular ways of traversing binary trees, namely: **preorder**, **inorder**, and **postorder**.

Inorder:
1. Traverse the left subtree in inorder;
2. Visit the root;
3. Traverse the right subtree in inorder.

Preorder:
1. Visit the root;
2. Traverse the left subtree in preorder;
3. Traverse the right subtree in preorder.

Postorder:
1. Traverse the left subtree in postorder;
2. Traverse the right subtree in postorder;
3. Visit the root.

In the example binary tree above, the following is the result of traversing it in different orders.

Inorder:	H D B E A I F J C H
Preorder:	A B D H E C F I J G
Postorder:	H D E B I J F G C A

These tree traversals can best be implemented using recursion.

```
Inorder(tree)
begin
   if tree ≠ nil
      Inorder(left(tree))
      process(info(tree))
      Inorder(right(tree))
end
```

```
Preorder(tree)
begin
   if tree ≠ nil
      process(info(tree))
      Preorder(left(tree))
      Preorder(right(tree))
end
```

```
Postorder(tree)
begin
   if tree ≠ nil
      Postorder(left(tree))
      Postorder(right(tree))
      process(info(tree))
end
```

The recursive algorithms above can be simulated using a stack implementation of recursive. This is illustrated in the code below.

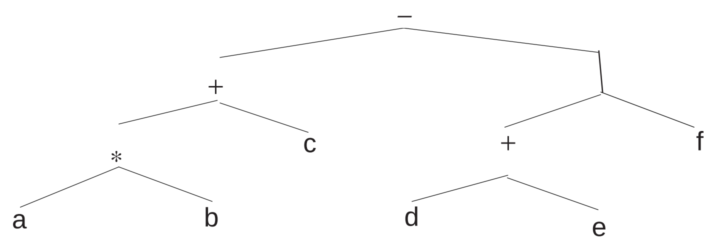
```
NorecursiveInorder(tree)
begin
    p = tree
    repeat
        while p ≠ nil do
            Push(stack,p)
            p = left(p)
        if stack ≠ empty
            p = Pop(stack)
            process(info(p))
            p = right(p)
    until (stack = empty) and (p = nil)
end
```

Applications of Binary Trees

Consider the problem of checking if a list contains any duplicate or not. The obvious way of solving this problem is to maintain a pointer that moves from the first element to the last element. While the pointer is pointing to an element, the value of that element is compared with the values of elements preceding it in the list. If one of them is equal to the value of the element currently being pointed to, the algorithm stops declaring that a duplicate exists. Otherwise, it moves to the next element. The algorithm stops declaring that no duplicate exists when the pointer falls after the last element.

The obvious solution described above is very inefficient. An alternative is to use a binary tree. The first element of the list is made the root of a binary tree. Then, one by one the remaining elements are compared to the root. If the value of the element is less than the root, it is compared with the root of the left subtree. If the left subtree does not exist the element becomes the left child of the root. However, if a left subtree exists, then the element is compared to the root of the left subtree repeating the process. Along the way, if the element is equal to one of the nodes, then a duplicate is declared existing.

Another application of binary trees is on the evaluation of mathematical expressions. Mathematical expressions are normally written in infix form (i.e., the operator is found in between the operands like a+b) and can be evaluated using a stack when it is translated to its postfix form (i.e., the operator is found after the operands like a b +). The same expressions can be evaluated when they are represented by expression trees. An expression tree is a representation of mathematical expressions where the operands are found at the leaves and the operators are found in the internal nodes. For example, the expression (a*b+b) - ((d+e)/f) can be represented by the following expression tree:



Let the node v in an expression tree contains the fields

value(v) - contains the value of the operand when v is a leaf
operator(v) - contains the operator type when v is an internal node, NULL otherwise.

Any expression tree pointed to by E can be evaluated using the procedure below.

```
EvaluateExpression(E)
begin
   if E ≠ nil then
      if operator(E) ≠ NULL then
         lvalue = EvaluateExpression(left(E))
         rvalue = EvaluateExpression(right(E))
         return Compute(operator(E), lvalue, rvalue)
      else return value(E)
end
```

The procedure Compute(operator(E), lvalue, rvalue) simply returns the value of the expression

lvalue operator(E) rvalue.

If the objective is to compute all subexpressions in the expression, we can do this by replacing return Compute(operator(E), lvalue, rvalue) with

```
value(E) = Compute(operator(E), lvalue, rvalue)
return value(E).
```

Now, try to answer the SAQ designed to apply binary trees on one specific problem. There are several other questions found in SAQ in Appendix 2.

SAQ 5-1

Write in pseudocode form the algorithm for checking duplicates in a list of n elements which is found in an array A.

ASAQ's are found in Appendix 1. SSAQ's are given in Appendix 2.

Supplementary Topic: Recursion

Something is **recursive** if part of it is defined by itself. **Recursion** is actually a powerful means of defining mathematical functions and is also considered a powerful programming tool. Although recursion is quite simple to apply, it is the most least understood by beginning programmers. To illustrate how this tool can be used to solve problems, we illustrate several problems and their recursive solutions.

The n! Problem

The factorial of numbers from 0 to n is defined by the following:

	Iterative Definition	Recursive Definition
0!	= 1	= 1
1!	= 1	= 1*0!
2!	= 2*1	= 2*1!
3!	= 3*2*1	= 3*2!
4!	= 4*3*2*1	= 4*3!
5!	=5*4*3*2*1	= 5*4!
…		
n!	= n*(n-1)* … *3*2*1	= n*(n-1)!

For any $n \geq 0$, an iterative solution is quite simple and can be implemented as follows:

```
fact(n)
begin
   j = n
   fact = 1
   while j ¹ 0 do
      fact = j*fact
      j = j-1
end
```

Using the recursive definition, the same problem can be solved by the following:

```
fact(n)
begin
   if n = 0 then return 1
   else return n*fact(n-1)
end
```

Fibonacci Sequence Problem

Next, we consider the problem where its mathematical recursive definition is directly implemented. The first few elements of the Fibonacci sequence are the following:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ….

The nth element in the sequence is computed as follows:

$$fib(n) = \begin{cases} n, & n=0, n=1 \\ fib(n-2) + fib(n-1), & n > 1 \end{cases}$$

which can be implemented as:

```
fib(n)
begin
   if n £ 1 then return n
   else return fib(n-2+fib(n-1)
end
```

Binomial Coefficients

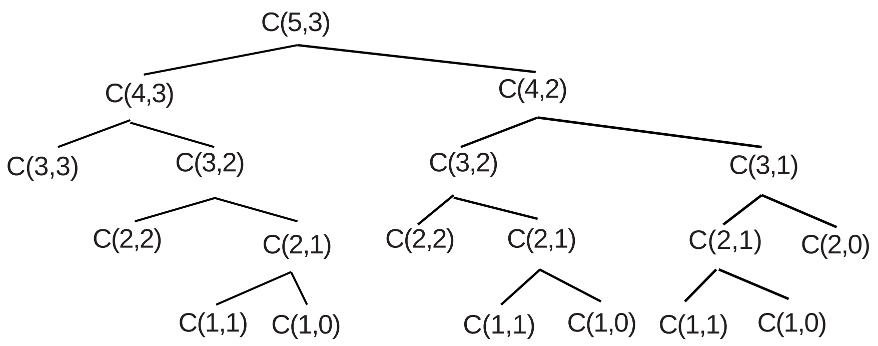
The binomial coefficients $\binom{n}{k}$ can be specified recursively. Letting C(n,k) $= \binom{n}{k}$, we can use the following relationships:

C(n,0) = 1 or C(n,n) = 1, for n ³ 0
C(n,k) = C(n-1,k)+C(n-1,k-1), for n > k > 0.

Using recursion, C(n,k) can be computed as follows:

```
C(n,k)
begin
   if k = 0 return 1
   elseif n = k return 1
   else return C(n-1,k) + C(n-1,k-1)
end
```

To illustrate the execution of this recursive procedure, we present the recursion tree for C(5,3).



The order in which the procedures complete their execution is postorder (similar to postorder traversal of binary trees). Hence, the trace of the execution above is the following:

C(5,3) - C(4,3) - C(3,3) - $C(4,3)_{\text{C(3,3) already computed}}$ - C(3,2) - C(2,2) - $C(3,2)_{\text{C(2,2) already computed}}$ - C(2,1) - C(1,1) - $C(2,1)_{\text{C(1,1) already computed}}$ - C(1,0) - $C(2,1)_{\text{C(1,1)+C(1,0) already computed}}$ - $C(3,2)_{\text{C(2,2)+C(2,1) already computed}}$ - $C(4,3)_{\text{C(3,3)+C(3,2) already computed}}$ - $C(5,3)_{\text{C(4,3) already computed}}$ - C(4,2) - C(3,2) - C(2,2) - $C(3,2)_{\text{C(2,2) already computed}}$ - C(2,1) - C(1,1) - $C(2,1)_{\text{C(1,1) already computed}}$ - C(1,0) - $C(2,1)_{\text{C(1,1)+C(1,0) already computed}}$ - $C(3,2)_{\text{C(2,2)+C(2,1) already computed}}$ - $C(4,2)_{\text{C(3,2) already computed}}$ - C(3,1) - C(2,1) - C(1,1) - $C(2,1)_{\text{C(1,1) already computed}}$ - C(1,0) - $C(2,1)_{\text{C(1,1)+C(1,0) already computed}}$ - $C(3,1)_{\text{C(2,1) already computed}}$ - C(2,0) - $C(3,1)_{\text{C(2,1)+C(2,0) already computed}}$ - $C(4,2)_{\text{C(3,2)+C(3,1) already computed}}$ - $C(5,3)_{\text{C(4,3)+C(3,3) already computed}}$.

Although recursion is just a supplementary topic, I need to check if you understand how it works. So please, answer the SAQ's below. The SAQ's are designed to test if you can formulate recursive solutions to some computing problems.

SAQ 5-2

1. The power(x,n) = x^n is defined by:

   Power(x,n) = 1, n = 0
   Power(x,n) = x * Power(x,n-1), n ^{3} 1.

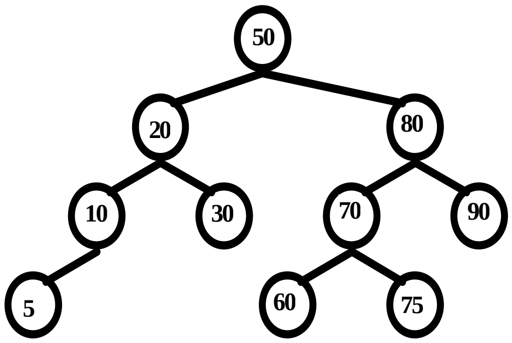
   Formulate a recursive procedure for computing power(x,n).

2. Write an improved recursive version of Power(x,n) that works by breaking n down into halves (half of n is n div 2), squaring Power(x,n div 2), and multiplying x again if n is odd. For example, $x^7 = x^3 * x^3 * x$, whereas $x^8 = x^4 * x^4$.

3. Write a recursive algorithm for reversing a linear linked list.

ASAQ's are found in Appendix 1.

Module 6
Binary Search Trees and AVL Trees



A binary tree has no restriction on the values and heights of its subtrees. Since a value may be found in any node in the tree, searching for a value will require at worst examining every node in the tree. One solution to this is the special type of binary tree called the **binary search tree**. Also, with no restriction on the height of the subtrees, the height of a binary tree may range from log n to n, where n is the number of nodes. In this case the solution is another type of binary tree called the **AVL tree**.

In this module, we will discuss the different operations that are defined in binary search trees and AVL trees.

Objectives

At the end of this module, you should be able to:

1. Construct a binary search tree and an AVL tree from a set of keys; and
2. Apply these data structures in solving some tree-based

Binary Search Trees

Search trees are data structures that support many dynamic-set operations, including search, minimum, maximum, predecessor, successor, insert, delete. Hence, a search tree can be used both as a **dictionary** (insert, search, and delete) and as a **priority queue** (insert, maximum, delete maximum). A **binary search tree (BST)** is a binary tree where the keys satisfy the binary-search-tree property given below.

Let ***u*** be a node in a binary search tree and let info(u) contain a field key(u) which is unique to ***u***. If v is a node in the left subtree of ***u***, then key(v) ≤ key(u). If ***v*** is a node in the right subtree of ***u***, then key(u) ≤ key(v).

An example of binary search tree is given below.

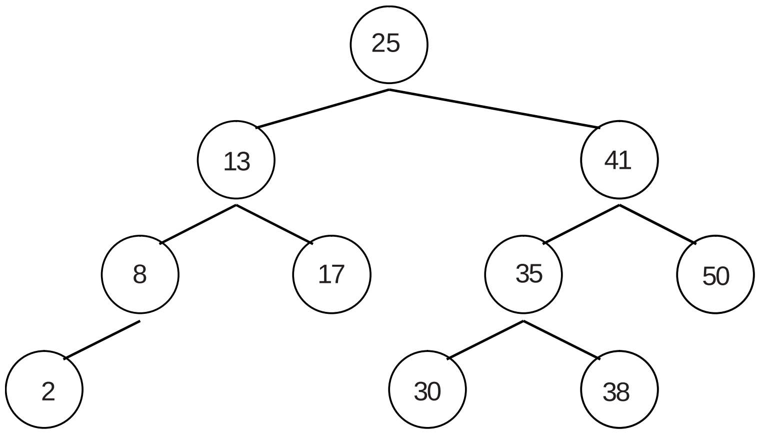


Figure 6-1. An example binary search tree

With a binary search tree, we can easily print out all the keys in sorted order by using a simple recursive procedure, called an **inorder tree traversal**.

```
 procedure Inorder-Tree-Traversal(r)
begin
   if r ≠ nil then
       Inorder-Tree-Traversal(left(r))
       print(key(r))
       Inorder-Tree-Traversal(right(r))
end
```

Searching a BST

Given a pointer to the root r of a BST T and a value k to be searched, a procedure for searching T is as follows:

```
procedure BST-Search(r,k)
begin
   if r = nil or k = key(r) then
      return r
   if k < key(r) then
      return BST-Search(left(r),k)
   else
      return BST-Search(right(r),k)
end
```

The nodes encountered during the recursion form a path downward from the root of the tree to one of the leaves.

Alternatively, we can unroll the recursion to form an iterative solution to the problem (in some computers this is faster).

```
procedure Iterative-BST-Search(r,k)
begin
   while r ≠ nil or k ≠ key(r) do
      if k < key(r) then
         r = left(r)
      else
         r = right(r)
   end
   return r
end
```

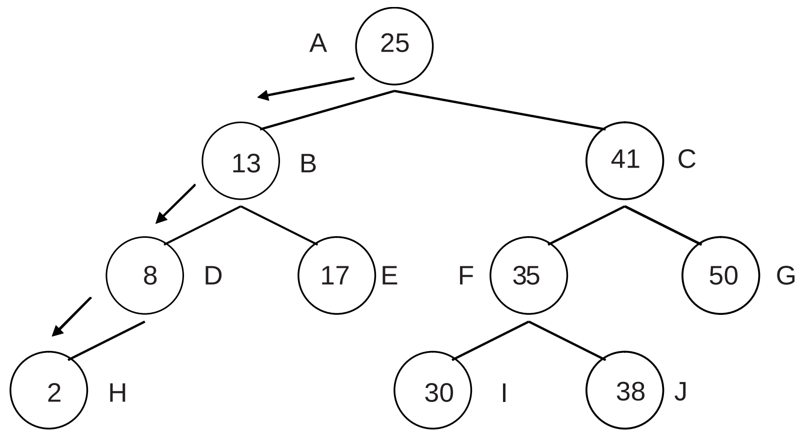


Figure 6-2. Searching for the minimum: r = A, B, D, & H.

Minimum and Maximum

The **minimum** in a BST can be found by following the left child pointer until a nil left child pointer is encountered.

```
procedure BST-Minimum(r)
begin
    while left(r) ≠ nil do
        r = left(r)
    end
    return r
end
```

Similarly, the **maximum** can be found by following the right child pointer.

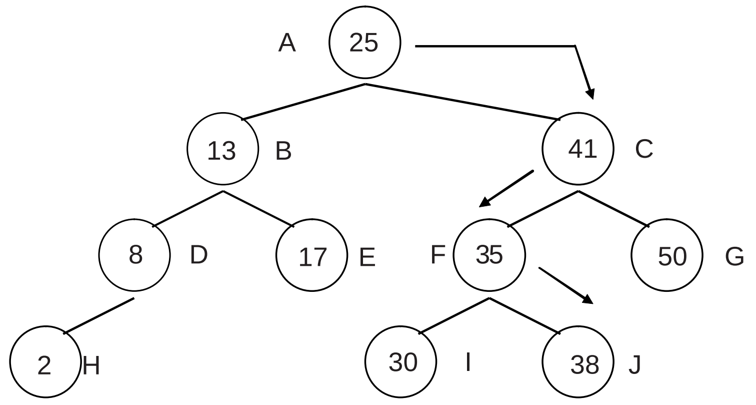


Figure 6-3. Searching for k=38: r =A, C, F, & J.

Successor and Predecessor

Given a set of values, say 7 3 4 2 1 8 6 5, the **successor** of an element is the value that follows the element when the values are written in sorted order. That is, the successor of 4 in 1 2 3 4 5 6 7 8 is 5. The **predecessor**, on the other hand, is the value before the element in the sorted order of the elements. That is, the predecessor of 7 is 6.

Given a node in a binary search tree, it is sometimes important to be able to find the successor or predecessor of the node. If all the keys are distinct, the successor of a node x is the node with the smallest key value in the right subtree rooted at x. The structure of the BST allows us to determine the successor of a node without even comparing keys. The following returns the successor of the node x in a binary search tree if it exists, and nil if x has the largest key in the tree.

```
procedure BST-Successor(x)
begin
    if right(x) ≠ nil then
        return BST-Minimum(right(x))
    else
        y = parent(x)
        while y ≠ nil and x = right(y) do
            x = y
            y = parent(y)
        end
        return y
end
```

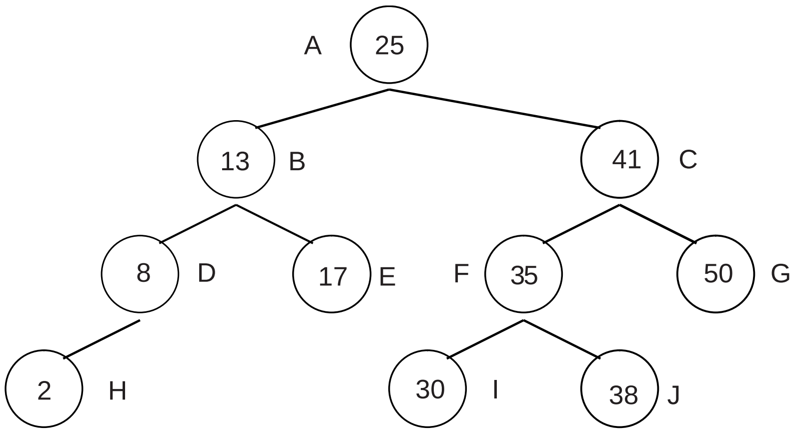


Figure 6-4. Finding the successor: (a) x = A → Successor is I, (b) x = J →Successor is C.

Insertion

Here, we pass a node z to the procedure where key(z) = v, left(z) = nil, and right(z) = nil. The procedure modifies T and some of the fields of z in such a way that z is inserted into an appropriate position in the tree.

```
procedure BST-Insert(T,z)
begin
    y = nil
    x = root(T)
    while x ≠ nil do
        y = x
        if key(z) < key(x) x = left(x)
        else x = right(x)
    end
    parent(z) = y
    if y = nil root(T) = z
    else
        if key(z) < key(y) left(y) = z
        else right(y) = z
end
```

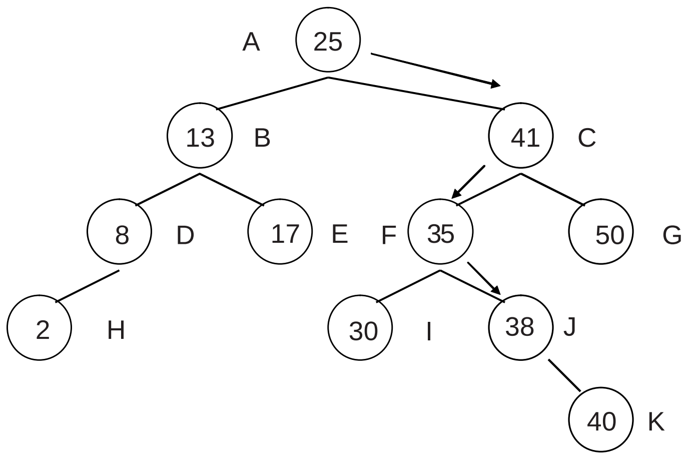


Figure 6-5. Inserting z = K into the tree.

Deletion

The procedure for deleting a given node z from a BST takes as an argument the pointer to z. The procedure considers three cases:

1. z has no children—we just remove z, i.e., we modify its parent parent(z) to replace z with NIL as its child.
2. z has only one child—we **"splice out"** z by making a new link between its child and its parent, i.e., replace z with its only child.
3. z has two children—we splice out z's successor y, which has no left child and replace the contents of z with the contents of y.

```
procedure BST-Delete(T,z)
begin
   if left(z) = nil or right(z) = nil y = z
   else y = BST-Successor(z)
   if left(y) ≠ nil x = left(y)
   else x = right(y)
   if x ≠ nil then parent(x) = parent(y)
   if parent(y) = nil root(T) = x
   else
      if y = left(parent(y)) left (parent(y)) = x
      else right(parent(y)) = x
   if y ≠ z
      key(z) = key(y)
      {copy other fields of y}
   return y
end
```

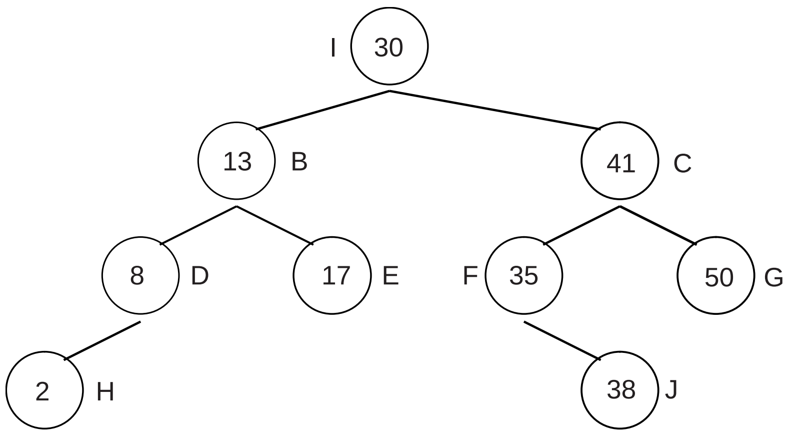


Figure 6-6. Deleting z = A.

Although a number of operations are defined for binary search trees, most of these operations are quite simple. Most of these operations are covered in the SAQ's below. Try to answer the SAQ's and identify the operations that were involved.

SAQ 6-1

1. Show the result of inserting 5, 4, 2, 1, 7, 3, 8, 6, 9 into an initially empty binary search tree. Show the result after deleting the root.

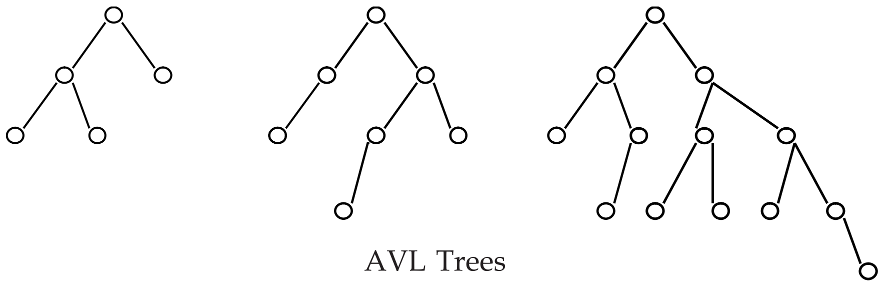
2. Define a reverse binary search tree as a tree where the values of nodes at the left subtree of the node is greater than or equal to the value of the node and the values of nodes at the right subtree is less than the node. Write a procedure for converting a binary search tree into a reverse binary search.

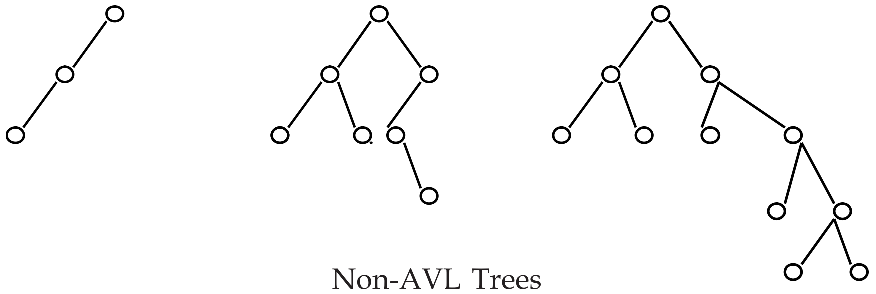
ASAQ's are found in Appendix 1. SSAQ's are given in Appendix 2.

AVL Trees

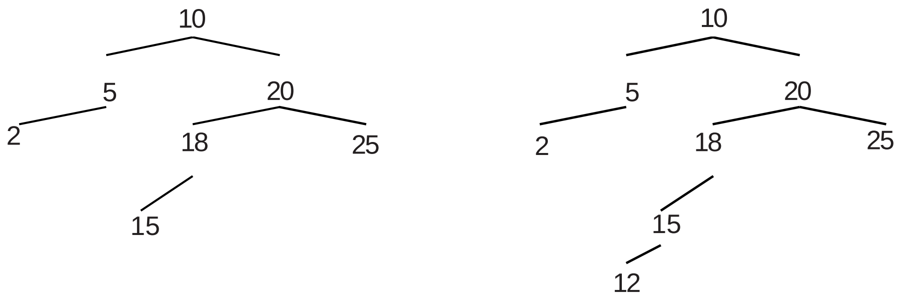
Binary search trees produce good search times when they are balanced. However, when constructing a binary search tree by repeated insertion it may produce a tree that is not balanced and hence the search operation deteriorates. Re-balancing a binary search tree may take time.

One solution to this problem was introduced by two Russian mathematicians Adelson-Velskii and Landis. Their solution is what is commonly known as the AVL trees. An **AVL tree** is a binary search tree where every node of the tree satisfies the AVL property: **the heights of the left and right subtrees differ by at most one.**

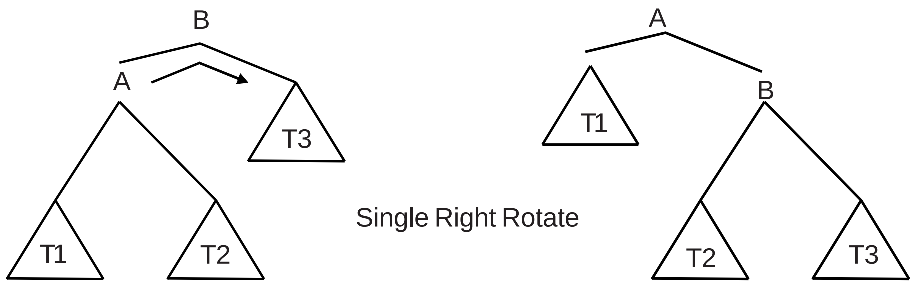

AVL Trees


Non-AVL Trees

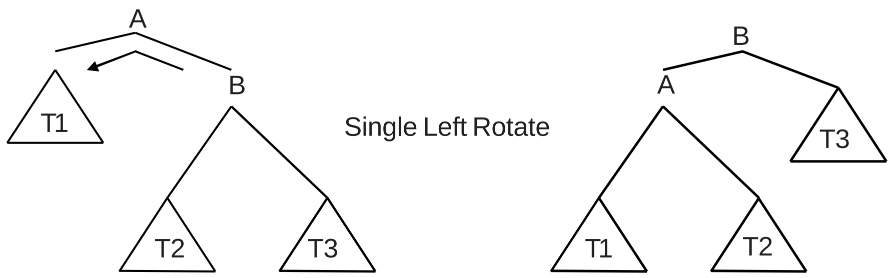
When building or inserting new nodes into an AVL tree, it is possible that the AVL property will be lost at some point. For example, inserting the new key 12 into the AVL tree below will produce a non-AVL tree.



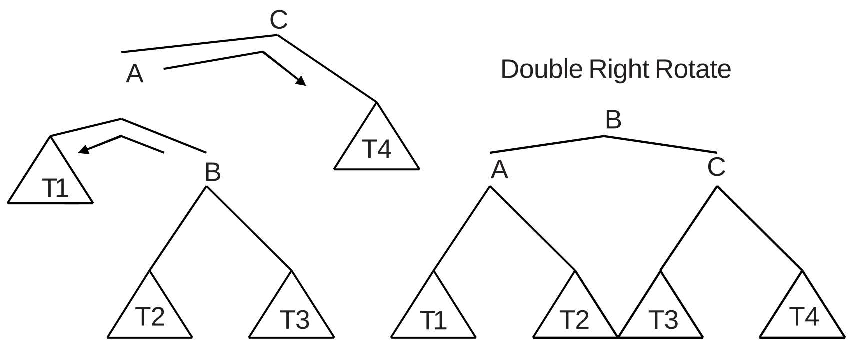
When the AVL property is lost at a node, we can apply some shape-changing tree transformations to restore the AVL property. To restore the AVL property, we may apply four different transformations all of which involve **rotations**, namely: **single right**, **single left**, **double right** and **double left**.



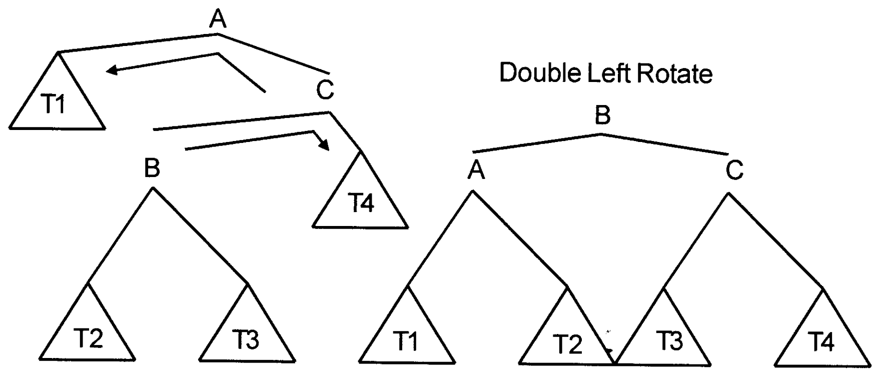
Single Right Rotate



Single Left Rotate

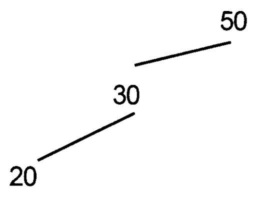


Double Right Rotate

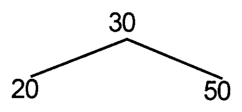


Creating an AVL Tree Using Insertions and Rotations

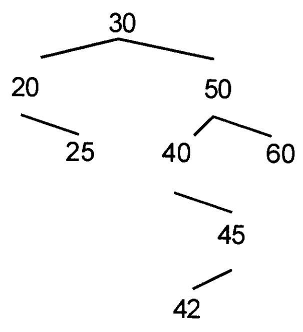
Consider a sequence of keys 50, 30, 20, 25, 40, 60, 45, and 42. Creating an AVL tree from this set of keys would be just like creating a binary search tree. The process will first create the following binary search tree:



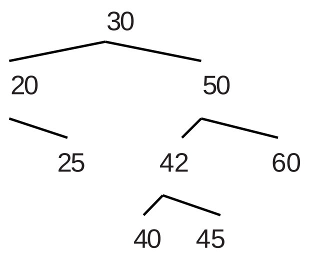
which will be rotated (right rotate) to make it an AVL into the following:



Continuing the same procedure as in a binary search tree, we will come to the following configuration of the tree:



Again, this is not an AVL tree. To restore the conditions of an AVL tree, we need to do a double left rotate at node 40. This produces the following AVL tree.



AVL trees are basically binary search trees with additional properties. Hence, the operations on binary search trees are almost similar to those in AVL trees. Try the SAQ's to see if you can use your knowledge of binary search trees to do operations on AVL trees.

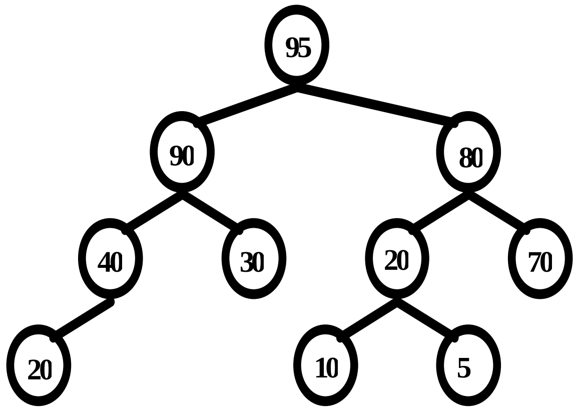
SAQ 6-2

1. Construct an AVL tree by repeated insertion from the following keys: 50, 20, 35, 75, 42, 61, 10, 5, 45, 12.

2. Write the procedures for the following: Single-Left-Rotate, Single-Right-Rotate, Double-Leaf-Rotate, and Double-Right-Rotate.

ASAQ's are found in Appendix 1. SSAQ's are given in Appendix 2.

Module 7
Heaps



Unlike binary search trees, AVL trees, and binary trees which are implemented using linked list structures, heaps are implemented using arrays. In this module, we will discuss the different operations on heaps. Also included is a discussion on the use of heaps as priority queues.

Objectives

At the end of this module, you should be able to:

1. Apply the heap structure in sorting and finding the kth smallest element in a set.

Heaps

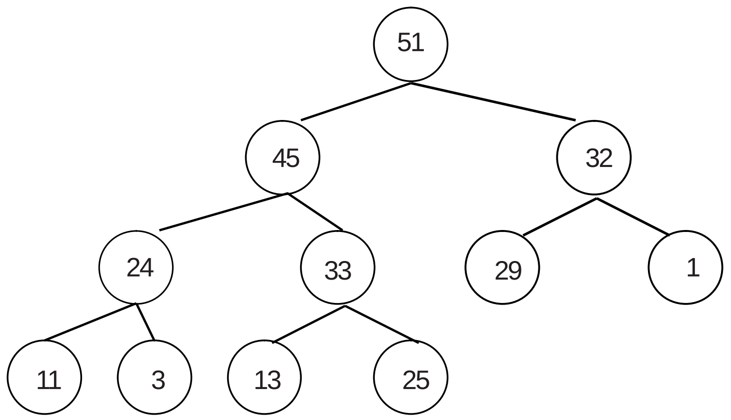
A **(binary) heap** data structure is an array object that can be viewed as a complete binary tree. The elements in the heap are indexed in level order and from left to right. The root of the tree is indexed 1. If a node has index i, then the indices of its parent parent(i), left child left(i), and right child right(i) can be computed as follows:

$$\text{parent}(i) = \lfloor i/2 \rfloor$$
$$\text{left}(i) = 2i$$
$$\text{right}(i) = 2i + 1$$

Moreover, a heap satisfies the heap property: **For every node i other than the root**, the following condition holds:

key(parent(i)) $\geq$ key(i),

i.e., the value of a node is ^{3} the values of its children.



1	2	3	4	5	6	7	8	9	10	11
51	45	32	24	33	29	1	11	3	13	25

Figure 7-1. A heap and its array equivalent.

Basic Procedures on Heaps

1. Heapify—procedure for maintaining the heap property when one of the elements of the heap is changed.

2. Build-Heap—procedure for producing a heap from an unordered input.

Maintaining the Heap Property

There are two common situations where the Heapify procedure is used. These are:

1. The value of one of the elements of the heap changes.

2. Given two heaps of the same sizes, we merge these by introducing a new node as the parent of the roots of the two heaps. For instance,

```
procedure Heapify(key,i,n)
begin
    l = left(i)
    r = right(i)
    if l ≤ n and key(l) > key(i) largest = l
    else largest = i
    if r ≤ n and key(r)> key(largest) largest = r
    if largest ≠ i
        swap(key(i), key(largest))
        Heapify(key, largest, n)
end
```

In the algorithm, n is the number of elements in the heap, i is the root of the subtree where the heap property is not satisfied. At each step, the largest of the elements key(i), key(left(i)). and key(right(i)) is determined and its index i is stored in largest. If key(i) is largest, then the subtree rooted at i is a heap and the procedure terminates. Otherwise, one of the two children has the largest element, and key(i) is swapped with key(largest) which cause node i and its children to satisfy the heap property. Consequently, Heapify must be called recursively on that subtree.

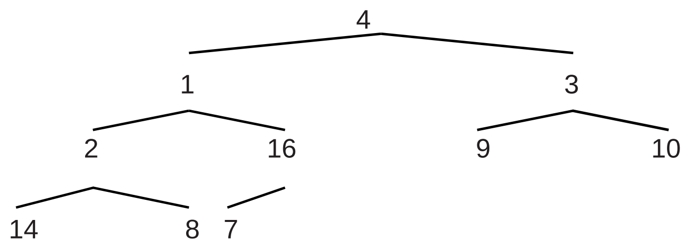
Building a Heap

Assuming the size of the heap is n, then building a heap can be done as follows:

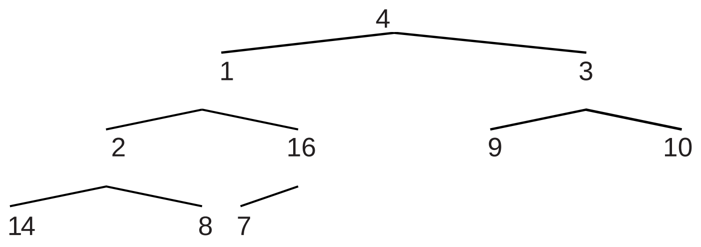
```
Build-Heap(key)
begin
    for i = $\lfloor v/2 \rfloor$ downto 1 do
        Heapify(key,i,n)
end
```

The operator $\lfloor\,\rfloor$ is the floor or truncate function which simply drops fractional part of the real number.

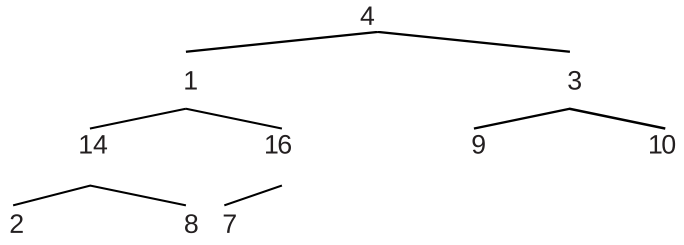
To illustrate, consider the following example. In this example $\lfloor n/2 \rfloor = 5$.



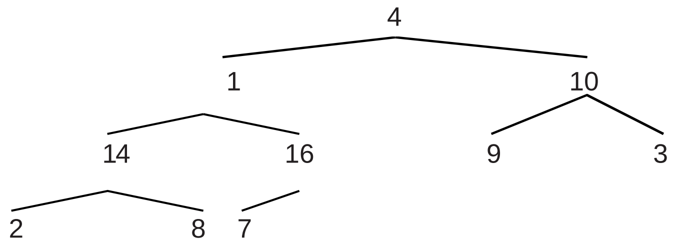
Original Input



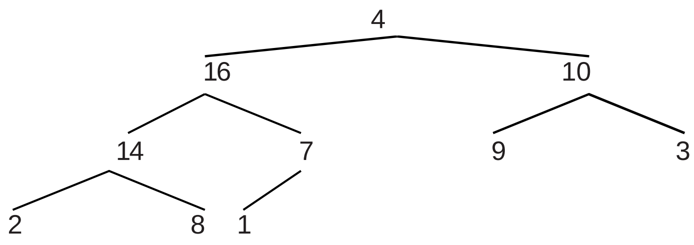
i = 5



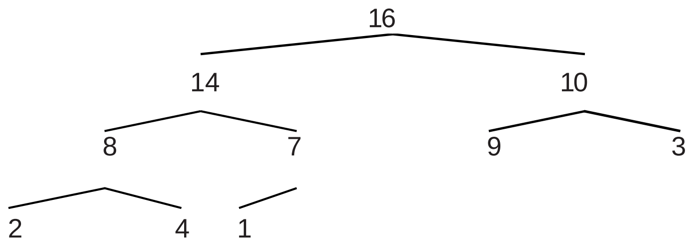
i = 4



i = 3



i = 2



i = 1

Priority Queues: Another Application of Heaps

A **priority queue** is a data structure for maintaining a set of ***s*** elements, each with an associated value called a **key**. It is called a priority queue mainly because the element with the highest priority can be accessed immediately just like the head of a queue. For a data structure to be considered a priority queue, it should at least support the following operations:

Insert(S,x)—inserts the element x into the set S. This could be written as S = S ∪ {x}. Let S be represented by the array key. With key forming a heap priority queue, the code for insert can be written as follows:

```
Heap-Insert(key,n)
begin
   i = n + 1
   while i > 1 and key(parent(i)) < x do
   begin
       key(i) = key(parent(i))
       i = parent(i)
   end
   key(i) = x
end
```

Clearly, the algorithm inserts the new values x at the end of the heap and compares this with its ancestors until it reaches its correct position.

Maximum(S)—returns the element of S with the largest key value, i.e., for a heap stored in array key:

```
Heap-Maximum(key)
begin
    return key(1)
end
```

Extract-Max(S)—removes and returns the element of S with the largest key value, i.e., for a heap stored in array key

```
Heap-Extract-Max(H)
begin
    max = key(1)
    key(1) = key(n)
    Heapify(key,1,n-1)
    return max
end
```

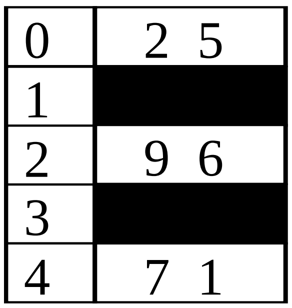
At this point, I think it is time for you to answer some SAQ's in order to test our understanding of heaps.

SAQ 7-1

1. Formulate an algorithm for locating the kth largest element in a heap with n elements.
2. Extend or modify the kth largest algorithm to produce an algorithm that produces a sorted sequence of the array.

ASAQ's are found in Appendix 1. SSAQ's are given in Appendix 2.

Module 8
Hashing



Many applications require a dynamic set that supports only the dictionary operation insert, search, and delete. For example, a compiler for a computer language maintains a symbol table, in which the keys of elements are arbitrary character strings that correspond to identifiers in the language. To support this type of applications, a hash table is usually used. A **hash table** is an effective data structure for implementing dictionaries. The implementation of hash tables is called **hashing**. Hashing is a technique used for performing insertion, deletion, and finds in constant average time.

In this module, we will introduce several hashing techniques. Unlike the previous modules which concentrated on the operations defined in the data structure, here we will be interested more on the functions used to identify locations in the data structure where data are to be stored.

Objectives

At the end of the module, you should be able to:

1. Select an appropriate hashing method for an existing application; and
2. Implement the different hashing methods.

Direct-Address Table

A simple technique that works well when the universe n of keys is reasonably small is the **direct address table**. Suppose that an application needs a dynamic set in which each element has a key drawn from the universe U = {0, 1, 2, ..., m-1}, where m is not too large. Assume that no two elements have the same key. To represent a dynamic set, we use an array, or *direct-address-table*, T[0, .. m-1], in which each position, or **slot** corresponds to a key in the universe U. The dictionary operations for this type of tables can be implemented easily. In fact, all the operations can be implemented in constant time. The implementations of these operations are:

```
 Direct-Address-Search(T,k)
begin
    return (T[k])
end
```

```
 Direct-Address-Insert(T,x)
begin
    T[key[x]] = x
end
```

```
 Direct-Address-Delete(T,x)
begin
    T[key[x]] = NIL
end
```

Direct-addressing has several disadvantages. Some of them are as follows:

1. If the universe is too large, then storing a table T of size $|U|$ may be impractical;

2. The set k keys actually stored may be small relative to U that most of the space allocated for T would be wasted.

Hash Tables

With direct addressing, an element with key k is stored in slot k. While in hashing, this element is stored in slot h(k), i.e., a **hash function** h is used to compute the slot from the key. Here, h maps the universe U of keys into the slots of the hash table T[0,..., m-1]:

$$H: U \rightarrow \{0, 1, ..., m-1\}.$$

We say that an element with key k hashes to slot h(k), or h(k) is the hash value of k. Instead of $|U|$ values, we need to handle only m values. So the storage requirements are correspondingly reduced. The problem with this idea is that two keys may hash to the same slot—a **collision**. Fortunately, there are effective ways of resolving the conflict created by collisions.

Collision Resolution by Chaining

In **chaining**, we put all elements that hash to the same slot in a linked list, i.e., slot contains a pointer to the head of the list of all stored elements that hash to j; if there are no such elements, slot j contains nil.

```
Chained-Hash-Insert(T,x)
begin
    insert x at the end of the list T[h(key[x])]
end
```

```
Chained-Hash-Search(T,x)
begin
  search for an element with key k in list T[h(key[x]]
end
```

```
Chained-Hash-Delete(T,x)
begin
    Delete x from the list T[h(key[x])]
end
```

Hash Functions

What makes a good hash function? A good hash function satisfies (approximately) the assumption of **simple uniform hashing (SUH)**: each key is equally likely to hash to any of the m slots. Normally, however, we do not know the distribution of the set of keys. In cases where the distribution is known, the keys are known to be random real numbers k independently and uniformly distributed in the range [0,1). In this case, we can use the hash function

$$h(k) = \lfloor km \rfloor$$

because this satisfies the assumption of SUH.

Formally, let us assume that each key is drawn independently from U according to a probability distribution of the set of keys; i.e., P(k) is the probability that k is drawn. Then, the assumption of SUH is that

$$\sum_{k:h(k)=j} P(k) = 1/m \text{ , for } j = 0, 1, ..., m-1$$

Another issue is the interpretation of keys as natural numbers. Most hash functions assume that the universe of keys is the set N = {0, 1, 2,...} of natural numbers. Thus, if the keys are not natural numbers, a way must be found to interpret them as natural numbers.

Examples are:

1. Real numbers can be multiplied by the number of decimal places.

2. Character strings can be interpreted as integers expressed in a suitable radix notation. For instance, the string *"pt"* might be interpreted as (112, 116) where 112 and 116 are the ASCII values of *"p"* and *"t,"* respectively. Then, expressed as a radix-128 integer, *"pt"* becomes (112x128) + 116 = 14452.

Division Method

We map a key k into one of m slots by taking the remainder of k divided by m, i.e., the hash function is

$$h(k) = k \bmod m.$$

When using the method, we usually avoid certain values of m. Some values which we must avoid are:

1. The value m should not be a power of 2. If $m = 2^p$, then, h(k) is just the p lowest-order bits of k. It is always better to make the hash function depend on all the bits of the key rather than on a few bits.

2. The values which are powers of 10 should be avoided if the application deals with decimal numbers as keys, since then the hash function does not depend on all the decimal digits of k.

3. When $m = 2^p - 1$ and k is a character string interpreted in radix 2^p, two strings, that are identical except for n transpositions of two adjacent characters, will hash to the same value.

Good values for m are primes not too close to exact powers of 2.

Multiplication Method

We multiply the key k by a constant A in the range of $0 < A < 1$ and extract the fractional part of kA. Then, we multiply this value by m and take the floor of the result. In short, the hash value is

$$h(k) = \lfloor m(kA \bmod 1) \rfloor$$

where *"kA mod 1"* is the fractional part of kA. Its advantage is that it does not depend on the value of m. Typically we choose it to be a power of 2, i.e., $m = 2^p$ for some integer p.

The procedure just described can be implemented in most computers as follows: Suppose the word size of the machine is w bits and k fits into a single word. We multiply k by the w-bit integer $\lfloor A \cdot 2^w \rfloor$. The result is a 2w-bit value $r_1 2^w + r_0$, where r_1 is the high-order word of the product and r_0 is the low-order word of the product. The desired p-bit hash value consists of the p most significant bits of r_0. Although this works for any value of A, Knuth, D.E. suggested the value $A \approx (\sqrt{5} - 1)/2 = 0.6180339887...$

Universal Hashing

One problem with a fixed hash function is that if a malicious adversary chooses the keys to be hashed, then he can choose n keys that all hash to the same slot, yielding an average retrieval time of O(n). The solution to this is to choose the hash function randomly in a way that is independent of the keys that are actually going to be stored. This approach, called **universal hashing**, yields good performance on the average, no matter what keys are chosen by the adversary.

Let H be a finite collection of hash functions that map a given universe U of keys into the range {0, 1, ..., m-1}. Such a collection is said to be **universal** if for each pair of distinct keys x, y $\in$ U, the number of hash functions h $\in$ H for which h(x) = h(y) is precisely $|H|/m$. This implies that with a hash function randomly chosen from H, the chance of a collision between x and y when x $\neq$ y is exactly 1/m, which is exactly the chance of a collision if h(x) and h(y) are randomly chosen from the set {0, 1, ..., m-1}.

Open Addressing

In **open addressing**, all elements are stored in the hash table itself, i.e., each table entry contains either an element of the set or NIL. To perform insertion using open addressing, we successively examine, or **probe**, the hash table until we find an empty slot in which to put the key. Instead of being fixed in the order 0, 1, ..., m-1, the sequence of positions probed depends upon the key being inserted. To determine which slots to probe, we extend the hash function to include the probe number as second input. Thus, the hash function becomes

$$h: U \times \{0, 1, 2, ..., m-1\} \rightarrow \{0, 1, 2, ..., m-1\}.$$

With open addressing, we require that for every key k, the probe sequence

$$<h(k, 0), h(k, 1) ..., h(k, m-1)>$$

be a permutation of <0, 1, 2, ..., m-1>, so that every hash table position is eventually considered as a slot for a new key as the table fills up.

```
Hash-Insert (T, k)
   begin
      i = 0
      repeat
         j = h(k, i)
         if T[j] = NIL
            T[j] = k
            return j
         else i = i + 1
      until i = m
      error ("Hash table overflow")
   end
```

The algorithm for searching for key k probes the same sequence of slots that the insertion algorithm examined when key k was inserted. Therefore, the search can terminate (unsuccessfully) when it finds an empty slot, since k would have been inserted there and not later in its probe sequence.

```
Hash-Search (T, k)
begin
   i = 0
   repeat
      j = h(k, i)
      if T[j] = k return j
      i = i + 1
   until T[j] = NIL or i = m
   return NIL
end
```

Deletion from an open address table is difficult. When we delete a key from slot i, we can not simply mark that slot by storing NIL in it. Doing so might make it impossible to retrieve any key during whose insertion we have probed slot i and found it occupied. One solution is to mark the slot by storing in it a special value DELETED instead of a NIL. We should then modify procedures Hash-Search so that it keeps on looking when it sees the value DELETED, while Hash-Insert would treat such a slot as if it were empty so that a new key can be inserted.

Linear Probing

Given an ordinary hash function

$$h': U \rightarrow \{0, 1, ..., m-1\}$$

the method of **linear probing** uses the hash function

$$h(k, i) = (h'(k) + i) \bmod m$$

for $i = 0, 1, ..., m-1$. Given k, the first slot probed is T[h′(k)]. We next probe position slot T[h′(k) + 1], and so on up to slot T[m-1]. Then, we wrap around to slots T[0], T[1], ..., until we finally probe slot T[h′(k)-1]. Since the initial probe position determines the entire probe sequence, only m distinct probe sequences are used with linear probing.

The main advantage of linear probing is that it is easy to implement. One disadvantage is that it suffers from a problem known as **primary clustering**. Long runs of occupied slots build up, increasing the average search time. For example, let n = m/2 keys in the table, where every even-indexed slot is occupied and every odd-indexed slot is empty, then the average unsuccessful search takes 1.5 probes. If the first n = m/2 locations are the ones occupied, however, the average number of probes successful increases to n/4 = m/8.

To illustrate how this works, we consider inserting the keys

25,10,32,17,42,43,21

into the slots. Using linear probing with m = 11, we obtain the following:

k	i	h(k,i) = (h′(k) + i) mod m
25	0	(25+0) mod 11 = 3
10	0	(10+0) mod 11 = 10
32	0	(32+0) mod 11 = 10 (collision)
	1	(32+1) mod 11 = 0
17	0	(17+0) mod 11 = 6
42	0	(42+0) mod 11 = 9

43	0	(43+0) mod 11 = 10 (collision)
	1	(43+1) mod 11 = 0 (collision)
	2	(43+2) mod 11 = 1
21	0	(21+0) mod 11 = 10 (collision)
	1	(21+1) mod 11 = 0 (collision)
	2	(21+2) mod 11 = 1 (collision)
	3	(21+3) mod 11 = 2

To summarize,

slot	k
0	32
1	43
2	21
3	25
4	
5	
6	17
7	
8	
9	42
10	10

Quadratic Probing

Quadratic probing uses a hash function of the form

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$$

where h′ is an auxiliary hash function, c_1 and $c_2 \neq 0$ are auxiliary constants, and i = 0, 1, ..., m-1. The initial position probed is T[h′(k)] later positions probed are offset by amounts that depend on a quadratic manner on the probe number i. This works much better than linear probing, but to make full use of the hash table, the values of c_1, c_2 and m are constrained, i.e., they must be selected properly to make full use of this technique.

If two keys have the same initial probe position, then their probe sequence are the same, since $h(k_1,0) = h(k_2,0)$ implies $h(k_1,i) = h(k_2,i)$. This leads to a milder form of clustering, called **secondary clustering**. As in linear probing, the initial probe determines the entire sequence, so only m distinct probe sequences are used.

To illustrate the operation of this technique, we again consider the example of inserting the keys

25, 10, 32, 17, 42, 43, 21

into a table with 11 slots, i.e., m = 11 and h′(k) = k mod 11. Further, we let c_1= 2 and c_2 = 3. We obtain the following:

k	i	h(k,i) = (h′(k) + c i + c i) mod m
25	0	(3+0+0) mod 11 = 3
10	0	(10+0+0) mod 11 = 10
32	0	(10+0+0) mod 11 = 10 (collision)
	1	(10+2+3) mod 11 = 4
17	0	(6+0+0) mod 11 = 6
42	0	(9+0+0) mod 11 = 9
43	0	(10+0+0) mod 11 = 10 (collision)
	1	(10+2+3) mod 11 = 4 (collision)
	2	(10+4+12) mod 11 = 4 (collision)
	3	(10+6+27) mod 11 = 10 (collision)
	4	(10+8+48) mod 11 = 0
21	0	(10+0+0) mod 11 = 10 (collision)
	1	(10+2+3) mod 11 = 4 (collision)
	2	(10+4+12) mod 11 = 4 (collision)
	3	(10+6+27) mod 11 = 10 (collision)
	4	(10+8+48) mod 11 = 0 (collision)
	5	(10+10+75) mod 11 = 7

To summarize,

slot	k
0	43
1	
2	
3	25
4	32
5	
6	17
7	21
8	
9	42
10	10

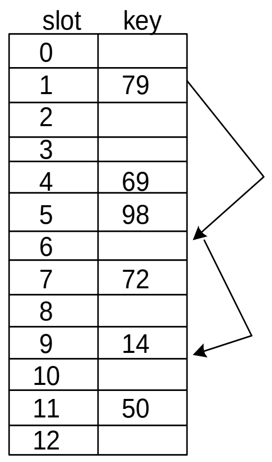
Double Hashing

One of the best methods available for open addressing because the permutations produced have many of the characteristics of the randomly chosen permutations. **Double hashing** uses a hash function of the form

$$h(k, i) = (h_1 (k) + i\, h_2 (k)) \bmod m$$

where h_1 and h_2 are auxiliary hash functions. The initial position probed is $T[h_1 (k)]$, successive probed positions are offset from previous positions by the amount h_2 (k) modulo m. Thus, unlike the case of linear and quadratic probing, the probe sequence here depends in two ways upon the key k, since the initial probe position, the offset, or both may vary.

For example, consider the figure below. Here, we have a hash table of size 13 with $h_1 (k) = k \bmod 13$ and $h_2 (k) = 1 + (k \bmod 11)$ since $k = 14 \equiv 1 \bmod 13$ and $14 \equiv 3 \bmod 11$, the key 14 will be inserted into slot 9, after slots 1 and 5 have been examined and found to be already occupied.



slot	key
0	
1	79
2	
3	
4	69
5	98
6	
7	72
8	
9	14
10	
11	50
12	

The value h_2 (k) must be relatively prime to the hash table size m for the entire hash table to be searched. Otherwise, if m and h_2 (k) have greatest common divisor d > 1 for some key k, then a search for key k would examine only (1/d)th of the hash table. A convenient way to ensure this condition is to let m be a power of 2 and to design h_2 so that it always produces an odd number. Another way is to let m be prime and to design h_2 so that it always return a positive integer less than m.

Example: Let m be prime and

h_1 (k) = k mod m
h_2 (k) = 1 + (k mod m′)

where m′ is chosen to be slightly less than m (say, m-1 or m-2). Suppose

k = 123956 and m = 701,

we have

h_1 (k) = 80 and h_2 (k) = 257.

Therefore, the first probe is in position 80, and then every 257th slot (modulo m) is examined while the key is found or every slot is examined.

To illustrate further, we consider inserting the keys

25, 10, 32, 17, 42, 43, 21

using m = 11, h_1 (k) = k mod m, h_2 (k) = k mod (m-1).

k	i	h(k,i) = (h (k) + i h (k)) mod m
25	0	(3+0) mod 11 = 3
10	0	(10+0) mod 11 = 10
32	0	10+0) mod 11 = 10 (collision)
	1	(10+2) mod 11 = 1
17	0	(6+0) mod 11 = 6
42	0	(9+0) mod 11 = 9
43	0	(10+0) mod 11 = 10 (collision)
	1	(10+3) mod 11 = 2
21	0	(10+0) mod 11 = 10 (collision)
	1	(10+1) mod 11 = 0

To summarize,

slot	k
0	21
1	32
2	43
3	25
4	
5	
6	17
7	
8	
9	42
10	10

After that very long topic on hashing, I think it is time for SAQ's. Compared to other modules, this module is relatively easy to understand. Hence, you will find the SAQ's much easier to answer.

SAQ 8-1

1. Professor *X* hypothesizes that substantial performance gains can be obtained if we modify the chaining scheme so that each list is kept in sorted order. How does the professor's modification affect the running time for successful searches, unsuccessful searches, insertions, and deletions?

2. Consider inserting the keys 10,22,31,4,15,28,17,88,59 into a table of length 11 using open addressing with the primary hash function h'(k) = k mod m. Illustrate the result of inserting these keys using

   a. linear probing;
   b. quadratic probing with $c_1 = 1$ and $c_2 = 3$;
   c. double hashing with h_2 (k) = 1 + (k mod (m-1)).

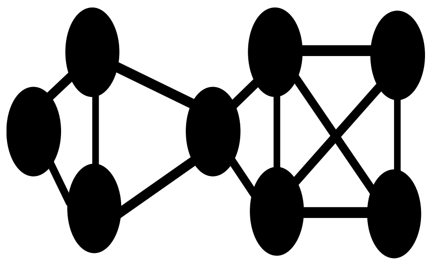
3. Write the pseudocode for Hash-Delete incorporating the method of using the special value *DELETED* instead of completely deleting the element. Modify the appropriate operations affected by the use of this special value *DELETED*.

ASAQ's are found in Appendix 1. SSAQ's are given in Appendix 2.

Module 9

Data Structures for Graphs

Trees are graphs with no cycles and there are problems where cycles are intrinsic to the problem. Hence, the binary-tree based data structures are unable to handle this type of problems. In this module, we will introduce data structures and algorithms for manipulating graphs.



Specifically, this module concentrates on two of the most basic graph searching techniques: the **breadth-first** search and the **depth-first** search. Almost all problems involving graphs use one of these search techniques for visiting all the nodes.

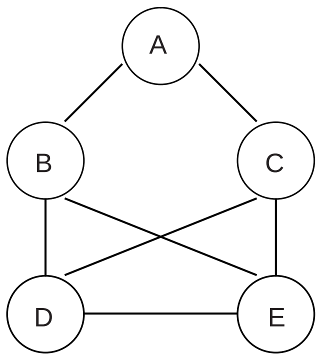
Objectives

At the end of the module, you should be able to:

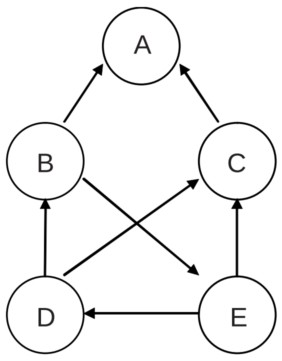
1. Evaluate which representation of graphs is appropriate for a given application; and
2. Apply the appropriate graph searching algorithm for a given

Basic Concepts on Graphs

A **graph** G = (V,E) consists of a set of vertices, V, and a set of edges, E. Each edge is a pair {u,v}, where u, v ∈ V. If the pair is ordered, i.e., (u,v), then the graph is called a **directed graph** (or **digraph)**, otherwise it is an **undirected graph**. From here onwards, however, when we say graph, we mean undirected graph.



Undirected Graph G



Directed Graph H

A **path** in a graph is a sequence of vertices $v_1, v_2, ..., v_n$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq n-1$. The **length** of such a path is the number of edges on the path, which is equal to n-1. A **simple path** is such that all vertices are distinct, except that the first and last could be the same. In graph G: A, C; A, B, D, C and A, B E, C, and A, B, D, E, C are all the simple paths from A to C.

A **cycle** is a path of length at least 1 such that $v_1 = v_n$. In graph G: A, B, D, C, A and A, B, D, E, C, A are examples of cycles in the graph.

An undirected graph is **connected** if there is a path from every vertex to every other vertex. Graph G above is an example of a connected graph. A directed graph with this property is said to be **strongly connected**. If a directed graph is not strongly connected, but the underlying graph (directions of the edges are ignored) is connected, then the graph is said to be **weakly connected**. Graph H above is weakly connected. A **complete graph** is a graph in which there is an edge between every pair of vertices.

Representations of Graphs

There are standard ways of representing graphs for manipulation in a computer. Two of these are:

1. Adjacency lists
2. Adjacency matrix

In both representations, we assume that the vertices in the graph are numbered 1, 2, ..., |V| in some arbitrary manner. Let e be the number of edges and n be the number of vertices in graph G, i.e., e = |E| and n = |V|. With the number of vertices fixed to n, a graph whose number of edges e that ranges from 0 to n(n-1) may be defined. When e is close to 0, the adjacency lists representation is usually used, while when e is close to n(n-1), the adjacency matrix is appropriate.

Adjacency Lists Representation

The **adjacency lists** representation of a graph G = (V, E) consists of an array Adj[1..n] of |V| lists, one for each vertex in V. For each u ∈ V, the adjacency list Adj[u] contains (pointers to) all the vertices v such that there is an edge (u, v) ∈ E. To illustrate, we present the adjacency representation of graphs G and digraph H above.

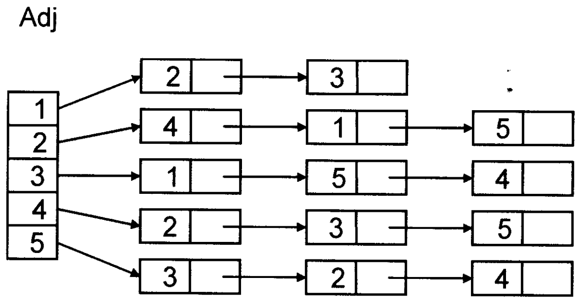


Figure 9-1 Adjacency lists of graph G.

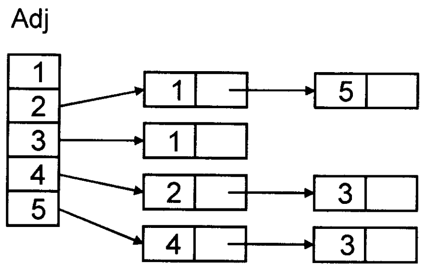


Figure 9-2. Adjacency lists of digraph H.

Adjacency Matrix Representation

The **adjacency matrix** representation of a graph G = (V, E) consists of a |V|x|V| matrix A = (a_{ij}) such that

$$a_{ij} = \begin{cases} 1, & \text{if } (i,j) \in E \\ 0, & \text{else} \end{cases}$$

To illustrate, consider the adjacency matrix representation of graph G and digraph H.

$$M \quad \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 1 & 0 & 0 \\ 2 & 1 & 0 & 0 & 1 & 1 \\ 3 & 1 & 0 & 0 & 1 & 1 \\ 4 & 0 & 1 & 1 & 0 & 1 \\ 5 & 0 & 1 & 1 & 1 & 0 \end{array}$$

Adjacency matrix of graph G

$$M \quad \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 0 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 & 0 & 1 \\ 3 & 1 & 0 & 0 & 0 & 0 \\ 4 & 0 & 1 & 1 & 0 & 0 \\ 5 & 0 & 0 & 1 & 1 & 0 \end{array}$$

Adjacency matrix of digraph H

The higher storage requirement of adjacency matrix compared to adjacency lists representation is balanced by faster methods of solving certain problems on an adjacency matrix. For example, checking if an edge exists between vertex 1 and 5 can be done by simply checking if M[1,5] = 1. While in the adjacency lists representation, there is a need to traverse the adjacency list pointed to by Adj[1].

Graph Searching

As in tree traversal, **searching a graph** is basically a method of visiting all the vertices of a graph in some systematic order. This is done by identifying a vertex s in G as a starting point. The vertices are systematically enumerated from s along paths in G. One such algorithm for enumerating the vertices is the following:

```
GraphSearch(G, s)
begin
 for each vertex v ∈ V that is accessible from s do
    visit(v)
end
```

Let each vertex v be composed of several fields, one important field is the field visited which is set to true when the information in the vertex is processed. The algorithm above can be refined to the following:

```
GraphSearch(G, s)
begin
   for each vertex v ∈ V do
      visited(v) = false
   put s into storage structure D
   repeat
      remove vertex v from D
      if not visited(v) then
         visit(v)    /* or process info(v) */
         visited(v) = true
         for each vertex w adjacent to v
            if not visited(w) then put w into D
   until D is empty
end
```

It should be noted that putting an element into D and removing an element from D will depend on what data structure is used in place of D. In fact, we shall show that by using different data structures in place of D. We will be producing different methods of searching the graph.

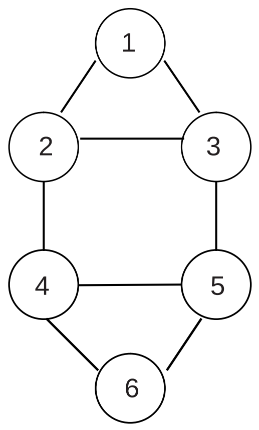
Depth-First Search (DFS)

When the storage structure D in the algorithm GraphSearch(G,s) is a stack, i.e., by replacing the **put** and **remove** operations in GraphSearch with the **push** and **pop** operations of the stack, respectively,

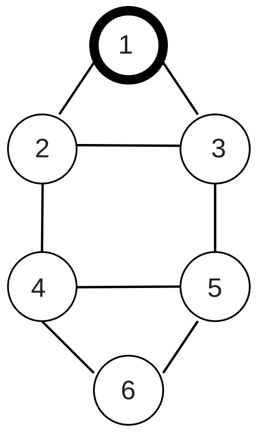
```
put w into D                  ≡ Push(Stack,w)
remove vertex v from D        ≡ v = Pop(Stack)
```

we produce the DFS algorithm. The searching algorithm defined is a **depth-first search (DFS)** algorithm. DFS is a generalization of **preorder traversal** for trees. As its name implies, DFS search "deeper" in the graph whenever possible. The idea is that edges are explored out of the most recently discovered vertex v that still has unexplored edges leaving it. When all the v's edges have been explored, the search **"backtracks"** to explore edges leaving the vertex from where v was discovered. This process continues until we have discovered all the vertices that are reachable from the original source vertex s. If any undiscovered vertices remain, then one of them is selected as a new source and the search is repeated from that source. This process is repeated until all vertices are discovered.

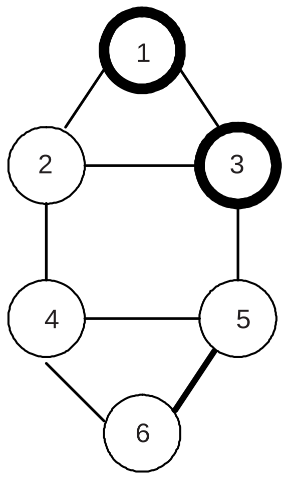
Let s = 1. An example execution of the algorithm above is given below.



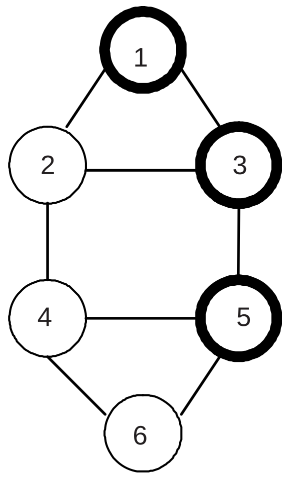
Stack: 1



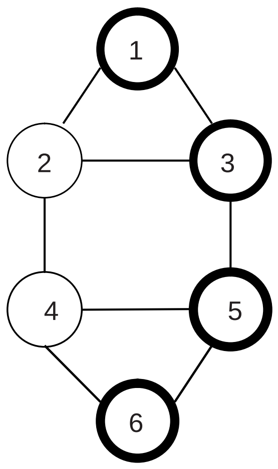
Stack: 2, 3



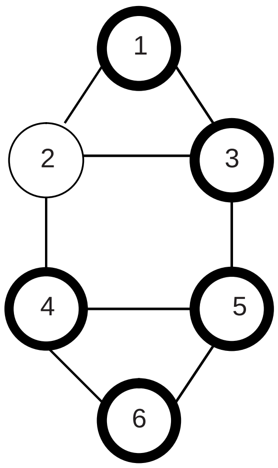
Stack: 2, 2, 5



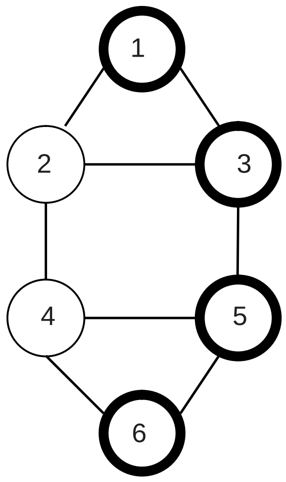
Stack: 2, 2, 4, 6



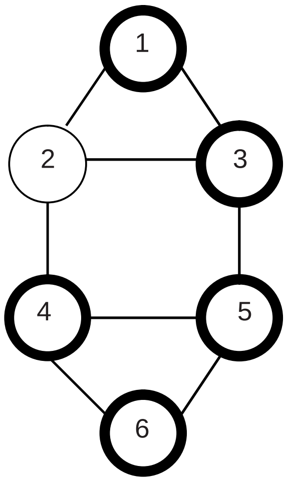
Stack: 2, 2, 4, 4



Stack: 2, 2, 4, 2



Stack: 2, 2, 4



Stack: 2, 2

Since, every vertex has already been visited, the succeeding execution will simply pop the contents of the stack one by one.

Predecessor Subgraph

Let us define the **predecessor subgraph** of a graph G(V,E) with source s as $G_{searchtree} = (V_{searchtree}, E_{searchtree})$, where

$$V_{searchtree} = \{v \in V \mid searchtree[v] \neq nil\} \cup \{s\}$$

and

$$E_{searchtree} = \{(searctree[v],v) \in E \mid v \in V_{searchtree} - \{s\}\}.$$

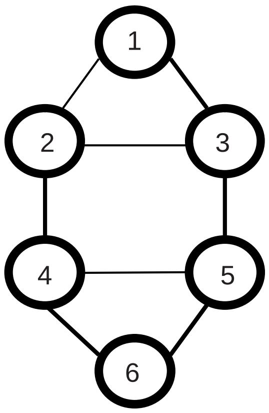
For a breadth-first search and depth-first search methods we define the G_{BFS} and G_{DFS}.

Depth-First Search Tree

The algorithm for DFS can easily be extended to compute the DFS tree of a graph. The general outline of the algorithm is similar to that of the DFS. The difference is that we push not only the vertex but also its predecessor into the stack. Once the vertex is visited its predecessor becomes part of the DFS tree. To illustrate this, consider the algorithm for finding the DFS tree below.

```
DFSTree(G, s)
begin
    for each vertex v ∈ V do
        visited(v) = false
    put nil into D
    put s into storage structure D
    repeat
        remove possible predecessor u from D
        remove vertex v from D
        if not visited(v) then
            visit(v)    /* or process info(v) */
            visited(v) = true
            DFStree[v] = u
            for each vertex w adjacent to v
                if not visited(w) then
                    put v into D
                put w into D
    until D is empty
end
```

Using the graph in the example for DFS, the depth-first search tree (bold edges and nodes) will be the result of the DFSTree algorithm above.



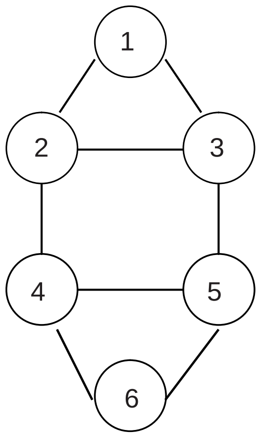
Breadth-First Search (BFS)

When the storage structure is a queue, i.e., by replacing the put and remove operations in GraphSearch with the insert and remove operations of the queue, respectively,

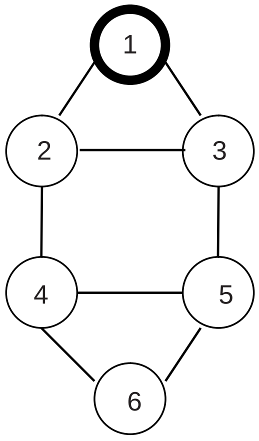
put w into D	≡ Insert(Queue, w)
remove vertex v from D	≡ v = Remove(Queue)

the searching algorithm defined is a **breadth-first search** algorithm. The search operates by processing vertices in layers: the vertices closest to the start are evaluated first, and the most distant vertices are evaluated last. This is much the same as the **level-order traversal** for trees.

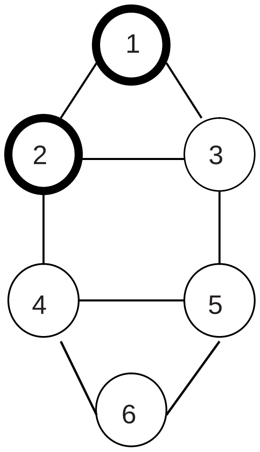
To illustrate the execution of the algorithm on a queue storage structure, we present the state of execution on a specific graph below. In the graph, let s = 1.



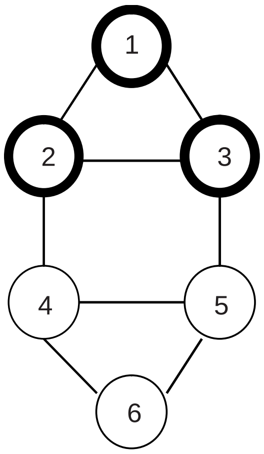
Queue: 1



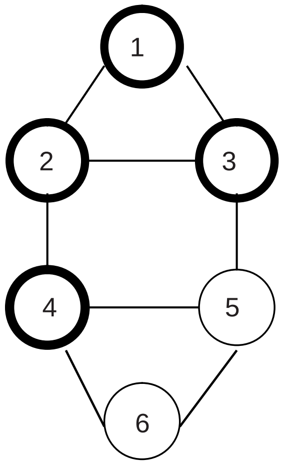
Queue: 2, 3



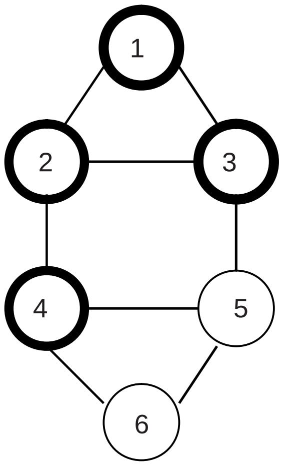
Queue: 3, 4, 3



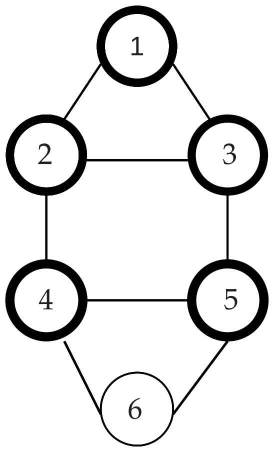
Queue: 4, 3, 5



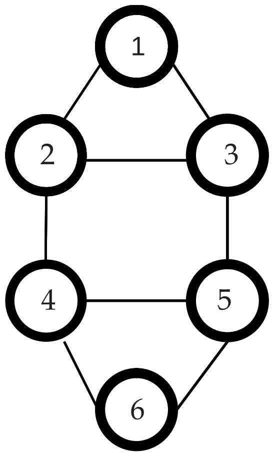
Queue: 3, 5, 6, 5



Queue: 5, 6, 5



Queue: 6, 5, 6



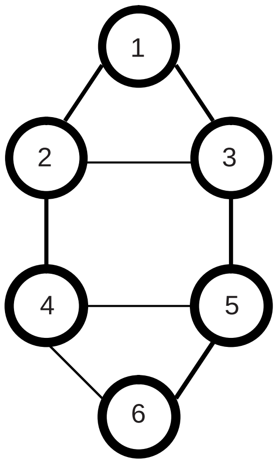
Queue: 5, 6

Breadth-First Search Tree

The BFS tree can be computed by making a small modification to the GraphSearch algorithm. The modification is obvious in the algorithm for BFSTree below.

```
BFSTree(G, s)
begin
    for each vertex v ∈ V do
        visited(v) = false
    put s into storage structure D
    seen(s) = true
    repeat
        remove vertex v from D
        if not visited(v) then
            visit(v)    /* or process info(v) */
            visited(v) = true
            for each vertex w adjacent to v
                if not seen(w) then
                    put w into D
                    seen(w) = true
                    BFStree[w] = v
    until D is empty
end
```

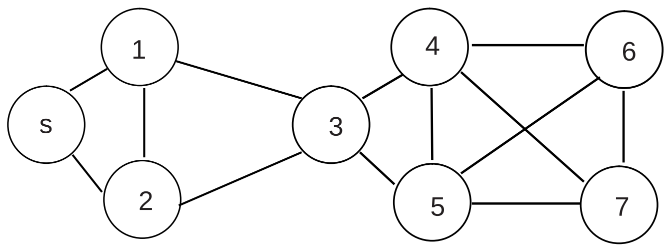
Similarly, the queue will be emptied one by one after all the vertices have been visited. After the search, the breadth-first search tree will have been formed. In the example, the BFS tree (bold edges and nodes) is the following:



To test your understanding of BFS and DFS, try the SAQ below. Please show your solution in detail. Compare your final result with the ASAQ in Appendix 1.

SAQ 9-1

Given the graph,



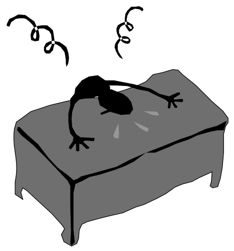
Find the (a) DFS tree and the (b) BFS tree of the graph.

ASAQ's are found in Appendix 1. SSAQ's are given in Appendix 2.

Congratulations! You have just finished Unit 1 of this course. Now, try the unit exam provided in the next page. The unit exam is intended as a practice exam. I suggest that you simulate the examination by answering the sample exam in one hour. Do it by yourself and compare your answers to those given in the ASAQ's. To pass the exam you need to get correct answers to at least 18 out of the 30 questions in the examination.

The graphics in the sample exam is intended to tell you that it is not easy to take examinations so you must take it seriously. It is not intended to discourage you from taking the sample examination. Good luck!

After the exam, compare your answers with those given in Appendix 1. I suggest that you re-take the sample examination until you get a correct answer to each item in the examination.



Unit 1 Sample Examination

Instruction. Multiple choice. Simply encircle the letter corresponding to your answer to the question.

1. The number of nodes in a complete binary tree of height h is equal to ____. Let height be equal to the number of levels.

   a) h
   b) 2^h
   c) 2^{h-1}
   d) 2^h-1
   e) 2^{h-1} - 1

2. The height of the left and right subtrees in an AVL tree differs by ____.

   a) 0
   b) at most 1
   c) at least 1
   d) at most 2
   e) NOTA (None of the Above)

   For numbers 3-5, let v be a node in a heap indexed by j. Give the indices of the following:

3. Parent of v:

   a) 2j
   b) 2j+1
   c) j div 2
   d) 2j-1
   e) NOTA

4. Left child of v:

   a) 2j
   b) 2j+1
   c) j div 2
   d) 2j-1
   e) NOTA

5. Right child of v:

   a) 2j
   b) 2j+1
   c) j div 2
   d) 2j-1
   e) NOTA

6. The hash function employing the division method is given by _____.

   a) h(k) = k div m
   b) h(k) = k mod m
   c) h(k) = k / m
   d) h(k) = k div (m+1)
   e) NOTA

7. Let the pointer 'list' be pointing to an existing list and pointer 'p' be pointing to a node not in the list. The code below insert node pointed by p into the list pointed by list.

   ```
   insert(list,p)
   begin
      ptr := list
      while ptr ¹ nil do
          ptr = next(ptr)
      ptr := p
   end.
   ```

   In what part of the list will the new node be inserted?

   a) head
   b) tail
   c) after p
   d) before p
   e) NOTA

8. The variant of linked lists that allows delete given only the pointer to the node to be deleted is the ____ linked list.

   a) linear linked list
   b) circular linked list
   c) doubly linked list
   d) All of the above
   e) NOTA

   For numbers 9-10, suppose a stack is implemented using linear linked list.

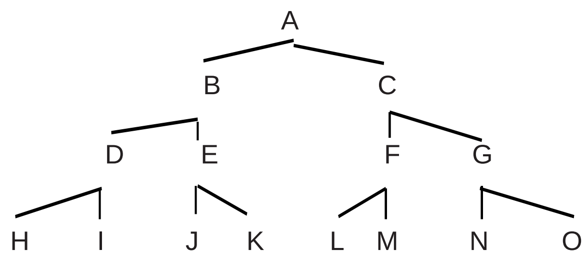
9. Pushing an element on top of stack is similar to inserting an element at the ______ of the list.

   a) head
   b) tail
   c) middle
   d) All of the above
   e) NOTA

10. Popping an element is similar to deleting the _______ node of the list.

    a) head
    b) tail
    c) middle
    d) All of the above
    e) NOTA

    For number 11-13, given the tree below.



Give the order by which the nodes will be visited in:

11. Postorder:

    a) A B C D E F G H I J K L M N O
    b) A B D H I E J K C F L M G N O
    c) H I D J K E B L M F N O G C A
    d) H D I B J E K A L F M C N G O
    e) NOTA

12. Preorder:

    a) A B C D E F G H I J K L M N O
    b) A B D H I E J K C F L M G N O
    c) H I D J K E B L M F N O G C A
    d) H D I B J E K A L F M C N G O
    e) NOTA

13. Inorder:

    a) A B C D E F G H I J K L M N O
    b) A B D H I E J K C F L M G N O
    c) H I D J K E B L M F N O G C A
    d) H D I B J E K A L F M C N G O
    e) NOTA

14. One can print the elements in a binary search tree in increasing order by traversing the nodes in _____ order.

    a) pre
    b) post
    c) in
    d) level
    e) NOTA

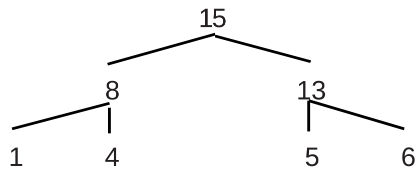
15. Accessing a node preceding the node where the pointer is currently pointing is best implemented in:

    a) linear linked list
    b) circular linked list
    c) doubly linked list
    d) All of the above
    e) NOTA

16. The First-In-First-Out data structure is the:

    a) linked list
    b) stack
    c) queue
    d) array
    e) NOTA

17. The array representation of the heap below is:



    a) 1 4 5 6 8 13 15
    b) 15 8 13 1 4 5 6
    c) 1 4 8 15 5 6 13
    d) 1 4 8 5 6 13 15
    e) NOTA

18. Let h1(k) = k mod m and h2(k) = k mod (m-1), where m = 10. In double hashing, the key value 13 will hash to slot:

    a) 0
    b) 1
    c) 2
    d) 3
    e) 13

19. After number 18 is done, the new key 23 will hash to:

    a) 3
    b) 6
    c) 7
    d) 8
    e) 9

20. Graphs with very few edges can best be represented by:

    a) adjacency matrix
    b) adjacency lists
    c) stack
    d) a and b
    e) NOTA

21. In a heap, the 2nd largest element is found in the array indexed:

    a) 1-3
    b) 2-3
    c) 3-7
    d) 4-10
    e) 5-15

22. Deleting an internal node following the node pointed to by p in a linked list can be carried out using the code:

    a) ptr = next(p); next(p) = next(next(p)); freenode(ptr);
    b) ptr = p; next(p) = next(next(p)); freenode(ptr);
    c) ptr = next(p); next(p) = next(next(next(p))); freenode(ptr);
    d) ptr = p; p = next(p); freenode(ptr);
    e) NOTA

23. To insert a node pointed to by p into a list pointed by list can be carried out using the code:

    a) ptr = list; while ptr ≠ nil do ptr = next(ptr) end; p = ptr;
    b) ptr = list; while ptr ≠ nil do next(ptr) = ptr end; p = ptr;
    c) ptr = list; while ptr ≠ nil do ptr = next(ptr) end; ptr = p;
    d) list = ptr; while ptr ≠ nil do ptr = next(ptr) end; p = ptr;
    e) NOTA

24. Counting the number of element in a stack can be carried out using the code:

    a) count = 0; while top>1 do Pop(Stack); count = count+1 end; return(count);
    b) count = 0; do Pop(Stack); count = count+1 until count=0 end; return(count);
    c) return(top);
    d) All of the above
    e) NOTA

25. Given n values stored in array A[1..n]. Finding the sum of n values recursively can be achieved by:

    a) sum(n)

```
    if  n = 1 return(A[1])
    else return sum(n+1)+A[n]
```

b) sum(n)

```
if  n = 1 return(A[1])
else return sum(n-1)+A[n]
```

c) sum(n)

```
if  n < 1 return(0)
else return sum(n-1)+A[n]
```

d) sum(n)

```
if  n < 1 return(0)
else return sum(n-1)+A[n]
```

e) NOTA

26. The third largest element in a heap can take the positions:

   a) 1-3
   b) 2-7
   c) 4-7
   d) 8-15
   e) NOTA

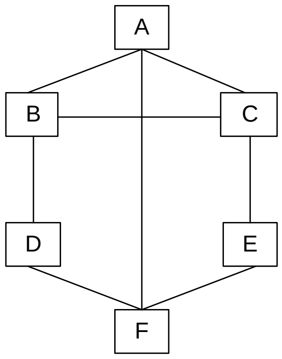
27. The smallest element in a binary search tree may be positioned at the:

   a) root
   b) rightmost leaf
   c) leftmost leaf
   d) internal node
   e) All of the above

28. The graph searching method that finds the shortest path from the source to every other node in the graph is:

   a) depth-first search
   b) breadth-first search
   c) preorder search
   d) inorder search
   e) postorder search

For numbers 29-30, given the graph:



29. Using Breadth-First Search with A as the source, which nodes (encircle all) will possibly be the third node to be visited:

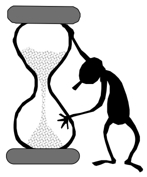
    a) B
    b) C
    c) D
    d) E
    e) F

30. Using Depth-First Search with A as the source, which nodes (encircle all) will possibly be the third node to be visited:

    a) B
    b) C
    c) D
    d) E
    e) F

Unit 2

Introduction to Analysis of Algorithms



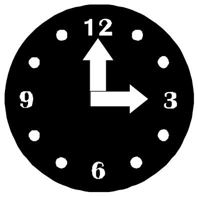
After an algorithm has been designed using an appropriate data structure, the next step is to verify whether or not the algorithm designed is efficient. **Efficiency** is measured in terms of resources used by the algorithm. Two of the most expensive resources required in the execution of an algorithm are the **CPU cycles** (or **time**) and **memory**. In order to estimate the amount of resources needed by a solution to a problem (**algorithm**), we normally go through the process of analyzing the requirements of an algorithm. This is usually in terms of its CPU time and memory requirements. The memory requirement of an algorithm is usually fixed regardless of the arrangement and values of the input, while the CPU time changes with the configuration of the input. Hence, the time requirement of an algorithm is usually examined for different configurations of the input. Related to this, we define the **best-case (worst-case) complexity** of an algorithm as the case where the execution time of the algorithm is the **minimum (maximum) execution** over all possible inputs. In some cases, we may want to consider every possible arrangement and value of the input. In this case, we consider the **average-case complexity** of the algorithm.

In this unit, we will introduce most of the tools needed in order to analyze an algorithm. Here, you will have to recall some of your mathematical skills on summations and mathematical series.

You are expected to spend the second half of the semester in this unit.

Module 10
Complexity Analysis

This is the first module in the analysis of algorithms portion of this course.. In this module, we introduce the mathematical notations used to describe the running times and memory requirements of algorithms. Using examples, we introduce how to compute the best-, worst-, and average-case complexities.



You should try and spend time understanding this module because this forms the basis of the succeeding modules. In fact, the concepts introduced here are used in the topics of the succeeding modules. Therefore, I suggest that you do not move to the next module unless you are convinced that you understood this one.

Objectives

At the end of this module, you should be able to:

1. Analyze the best-, worst-, and average-case complexities of simple problems; and
2. Correctly use the O, Ω and θ notations in the analysis.

Notations

First let us introduce some definitions which will be used in the analysis.

O-Notation

Definition: O(g(n)) is a set of functions f(n) such that there exists positive constants c and n_0 such that $0 \le f(n) \le c\, g(n)$ for all $n \ge n_0$.

Example: Show that $n^2/2 - 3n \in O(n^2)$
Solution: From the definition of O-notation,

$$n^2/2 - 3n \le c\, n^2$$
$$1/2 - 3/n \le c$$

This inequality holds for any value of $n \geq 6$ by choosing $c = 1/2$. Hence, with $c = 1/2$ and $n_0 = 6$ we can verify that $n^2/2 - 3n \in O(n^2)$. Actually, any value > 0 can be the choice for c.

When an algorithm runs in O(f(n)) time, this actually mean that the algorithm will run in time no greater than f(n) times some constant c.

Ω -Notation

Definition: Ω(g(n)) is a set of functions f(n) such that there exists positive constants c and n_0 such that $c\, g(n) \leq f(n)$ for all $n \geq n_0$.

Example: Show that $n^2/2 -3n \in \Omega(n^2)$.
Solution: From the definition

$$cn^2 \leq n^2/2 -3n$$
$$c \leq 1/2 -3/n$$

This inequality holds for any value of $n \geq 7$ by choosing $c = 1/14$. Hence, $n^2/2 -3n \in \Omega(n^2)$. Note however that any positive value $\leq 1/14$ can be used for c.

When an algorithm runs in Ω(f(n)) time, this actually mean that the algorithm will run in time no less than f(n) times some constant c.

θ -Notation

Definition: θ(g(n)) is a set of functions f(n) such that there exists positive constants c_1, c_2 and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$.

Example: Show that $n^2/2 -3n \in \theta(n^2)$.
Solution: From the O- and Ω-notations, we can show that $n^2/2 -3n \in \theta(n^2)$ by choosing $c_1 = 1/14$, $c_2 = 1/2$, and $n_0 = 7$.

Example: Show that $6n^3 \notin \theta(n^2)$.
Solution: Suppose c_1 and n_0 exist such that $6n^3 \leq c_2 n^2$ for all $n \geq n_0$. But $n \leq c_2/6$ which can not possibly hold for arbitrarily large n, since c_2 is a constant. Hence, $6n^3 \notin \theta(n^2)$.

When an algorithm runs in θ(f(n)) time, this actually mean that the algorithm will run in time no greater than f(n) times some constant c_1 and in time no less than f(n) times some constant c_2.

Example: Sequential Search

Sequential Search: Given an array A[1..n] that contains distinct keys, k_i , ($1 \leq i \leq n$), where each key, k_i , is stored in the array item, A[i], and a specific search key, k. The problem is to find the position of the key k in A[1..n].

An algorithm for this problem is the following:

```
procedure SequentialSearch (k, A)
begin
    i = 1
    found = false
    while (i ≤ n) and (not found) do
    begin
        if k = A[i] return(i
        else i = i + 1
    end
    return(0)
end
```

Memory Complexity Analysis

Memory complexity or the **storage complexity** is the amount of memory needed in the execution of an algorithm. Usually, this can easily be determined by examining the data structures used by the algorithm. In our example, the storage complexity is analyzed as follows:

Storage Complexity: The data structure is an array A[1...n] plus some constant number of variables which are used to store temporary values. Hence, the amount of storage used is $n + b = O(n)$, where b is a constant.

Best- and Worst-Case Time Complexity

For the sequential search problem, we can instantly determine the best case and the worst case for successful searches (i.e., the key k is equal to A[i], $1 \leq i \leq n$).

Best case: This occurs when k is found at A[1]. This obviously takes O(1) running time.

Worst case: This occurs when k is found at the last position of the array. Since n comparisons were required, this takes amount of work proportional (constant of proportionality is a) to the length of the array, n, plus possibly some overhead for setting up the while loop (given by b). The amount of work is therefore an + b = O(n).

Average-Case Time Complexity

To carry this out, we assume that each key in A[1..n] is equally likely to be used in the search. The average can then be computed by taking the total of all the work done for finding all of the different positions of keys and dividing by n. The work needed to find the i^{th} key, k_i, is of the form ai + b (a and b are similar to those in the worst-case analysis). Now, we have to add the terms ai + b for all i in the range ($1 \le i \le n$), and then divide by n.

$$\text{Average-case complexity} = \sum_{i=1}^{n} (ai + b) / n$$

$$= (a \sum_{i=1}^{n} i + \sum_{i=1}^{n} b) / n$$

$$= (a\ (n(n+1)/2) + bn) / n$$

$$= an/2 + a/2 + b$$

$$= O(n)$$

Example: Selection Sort

The sequential search example is quite simple such that its worst- and best-cases can be determined directly. Here, we consider a more complicated example so as to illustrate further how analysis of algorithms is carried out. We shall be using the selection sort algorithm which is given as follows:

```
procedure SelectionSort(A)
begin
   for i = n downto 2 do
    begin
      MaxPosition = i;
     for j = 1 to (i - 1) do
        if A[j] > A[MaxPosition]  MaxPosition = j;
      Swap(A[i],A[MaxPosition])
   end
end
```

Worst-Case Analysis

We note that the algorithm is composed of a nested for-loop. It is usually advisable to start analyzing the inner loop, i.e.,

```
        for j = 1 to (i-1) do
    if ...                          (Cost: constant, say a)
```

Hence, the total cost of the for loop is a(i-1). Then, we consider the next outer loop, i.e.,

```
for i = n downto 2 do
begin
  MaxPosition = i;                (Cost: constant, say $b_1$)
  for ...                         (Cost: a(i-1))
   Swap(A[i], A[MaxPosition])     (Cost: constant, say $b_2$)
end
```

In the algorithm, the assignment statement is assigned a constant cost of b_1 and the swap statement by a cost of b_2. A different constant symbol is usually used to indicate the cost of another statement of constant cost mainly to show that although both statements are of constant costs, they are actually different constant values.

Let $b = b_1 + b_2$, we can simplify the loop to

```
for i = n downto 2 do
   begin
      ...         (Cost: a(i-1) + b)
   end
```

we can replace a(i-1) + b by ai - a + b = ai + d, where d = b - a. From this, we can compute the cost of the loop, i.e.,

$$\text{Worst-case complexity} = \sum_{i=2}^{n} (ai + d)$$

$$= \sum_{i=2}^{n} (ai + d) - (a + d)$$

$$= a \sum_{i=2}^{n} i + \sum_{i=2}^{n} d - (a + d)$$

$$= an^2 /2 + (a/2 + d)n - (a+d)$$

$$= O(n^2)$$

Example: Recursive Selection Sort

We have presented a nonrecursive version of selection sort. To illustrate a different method of carrying out the analysis, we consider the following recursive selection sort algorithm:

```
procedure SelectionSort(A,n)
begin
  if n > 1 then
  begin
    MaxPosition = FindMax(A,n)
    Swap(A[n], A[MaxPosition])
    SelectionSort(A,n-1)
  end
end
```

```
FindMax(A,n)
begin
  j = n
  for i = 1 to (n-1) do
     if A[i] > A[j] then j = i
  return(j)
end
```

Worst-Case Analysis

In this example, we have a procedure call inside a loop. Hence, the logical thing to do is to analyze the procedure first. The analysis of FindMax is as follows:

```
j = n                        (Cost: constant, say $b_1$ )
for i = 1 to (n-1) do
begin
    if ...                   (Cost: constant, say $b_2$ )
end
return(j)                    (Cost: constant, say $b_3$ )
```

Hence, the total cost for one execution of the procedure is

$$b_1 + b_2 (n-1) + b_3 = an + b = O(n),$$

where $b = b_1 - b_2 + b_3$ and $a = b_2$.

The cost of the procedure is

```
if n > 1 then
begin
    MaxPosition = FindMax(A,n)        (Cost: an+b)
    Swap(A[n], A[MaxPosition])        (Cost: $c_1$ )
    SelectSort(A,n-1)                 (Cost: T(n-1))
end
```

Letting $d = b + c_1$, we get the total cost

$$T(n) = an + d + T(n-1) = O(n^2).$$

Memory Complexity

When a recursive procedure calls itself, the storage allocated to the caller is preserved, i.e., it is just deactivated until the recursive procedure called **inside the concerned procedure returns**. Hence, the depth of the recursion dictates the storage requirement of a recursive procedure. Suppose in the selection sort algorithm the array is declared globally. Thus, the storage requirement of the procedure is constant (O(1)). Since the depth of the recursion is n, the total amount needed by the procedure is

$$\begin{aligned} S(n) &= O(n) && \text{(Array A[1...n)} \\ &+ O(n) && \text{(worst stack storage required)} \\ &= O(n) \end{aligned}$$

The complexity is different when the array is copied in the called procedure. In this case

$$\begin{aligned} S(n) &= O(n) \times n && \text{(worst stack storage)} \\ &= O(n^2) \end{aligned}$$

Complexity Classes

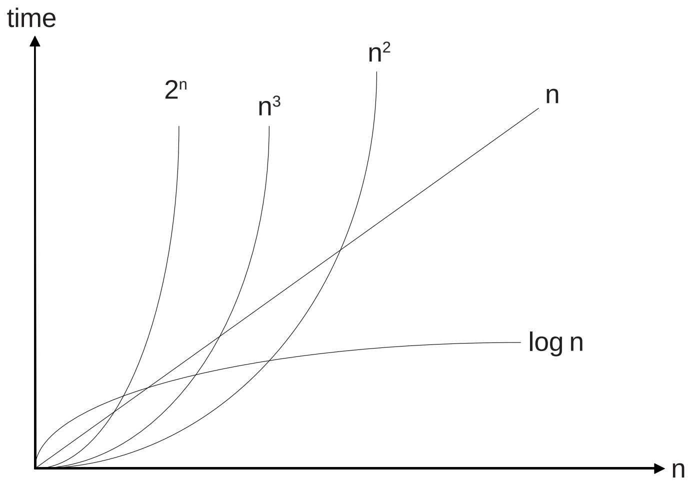
We learned earlier that when we say *"An algorithm runs in O(n^2) time,"* what we actually mean is that the algorithm runs in time no greater than n^2 times some constant c, provided the problem size (size of the input in most cases) is big enough. Hence, we classify all algorithms with this running time as belonging to the class of "quadratic" algorithms. Some of the common complexity cases that we shall be encountering in this unit are:

Description	O-notation
constant	O(1)
logarithmic	O(log n)
linear	O(n)
n log n	O(n log n)
quadratic	$O(n^2)$
cubic	$O(n^3)$
exponential	$O(2^n)$
exponential	$O(10^n)$

To give us an idea of the growth rates of these complexity classes, we consider a hypothetical computer that can execute one step of the algorithm in one microsecond.We might be interested in the number of microseconds it takes the computer to execute the algorithm for different sizes of the problem.

Class	n = 2	n = 16	n = 256	n = 1024
1	1	1	1	1
log n	1	4	8	1.00x10
n	2	1.6x10	$2.56x10^2$	$1.02x10^3$
n log n	2	6.4x10	$4.48x10^2$	$1.02x10^4$
n^2	4	$2.56x10^2$	$6.55x10^4$	$1.05x10^6$
n^3	8	$4.10x10^3$	$1.68x10^7$	$1.07x10^9$
2^n	4	$6.55x10^4$	$1.16x10^{77}$	$1.8x10^{308}$

In graphical form, this is given as follows:



I guess, it's time for some SAQ's. Turn to the next page for the SAQ's. The number of SAQ's are indication of the importance of this module. You need to understand this module thoroughly since it plays an important role in the succeeding topics.

SAQ 10-2

For SAQ's 1-3, show that the following statements are true.

1. 12 is O(1) and 12 is Ω(1)

2. $n^2/2 + n + 2$ is $O(n^2)$ and $n^2/2 + n + 2$ is $\Omega(n^2)$

3. $\max(n^2, n^3+n^2+n)$ is $O(n^3)$ and $\max(n^2, n^3+n^2+n)$ is $\Omega(n^3)$

4. What is the average-case complexity of (a) non-recursive selection sort, (b) recursive selection sort and (c) FindMax algorithm.

5. Analyze the two searching algorithm below. Compare the two results.

```
BinarySearch(A,x)
begin
      lower = 1;
      upper = n;
      repeat
          middle = (lower + upper) div 2;
          if x > A[middle] lower = middle + 1;
           else upper = middle - 1;
      until (A[middle]=x) or (lower > upper);
      return(A[middle] = x);
end

BinarySearch(A,x, lower, upper)
begin
     middle = (lower + upper) div 2
     if x > A[middle]
        BinarySearch(A,middle + 1, upper)
     elseif x < A[middle]
        BinarySearch(A,lower,middle-1)
     else
        found = true;
end
```

ASAQ's are found in Appendix 1. SSAQ's are given in Appendix 2.

Module 11
Simplifying Summations

$$\sum_{i=0}^{\infty} i$$

The usual way of describing the running time of an algorithm is through the use of the **summation notation**. However, for the summation to be meaningful, it must be simplified. In this module we will introduce some of the tools for facilitating the simplification of summations.

To do well in this module, you have to recall previous knowledge of **summation** and **arithmetic series**. It will help you understand this module if you review these two concepts first.

Objectives

At the end of this module, you should be able to:

1. Explain some tools needed to simplify a summation to its simplest form; and
2. Apply these tools in actually simplifying a summation to its simplest form.

Telescoping a Series

When no existing identity is available to simplify a summation, one alternative method is what we call **telescoping a series**. To illustrate this method, we show that

$$\sum_{k=1}^{n} (a_k - a_{k-1}) = a_n - a_{0.}$$

This is obvious from the fact that a_1, a_2, ..., a_{n-1} are each added once and subtracted out once. As an application, consider the series: $\sum_{k=1}^{n-1} 1/(k(k+1))$.

Since 1/ k(k+1) = 1/k - 1/(k+1) we get

$$\sum_{k=1}^{n-1} 1/(k(k+1)) = \sum_{k=1}^{n-1} (1/k - 1/(k+1))$$
$$= 1 - 1/n.$$

Shifting a Series

Another method is **shifting a series**. This usually works hand in hand with the method telescoping a series. Consider the example series,

$$S(n) = \sum_{k=0}^{n} 2^i = 1 + 2 + 4 + ... + 2^n$$

The expression is shifted to the right by multiplying both sides by 2, resulting in:

$$2S(n) = 2 + 4 + ... + 2^n + 2^{n+1}$$

Since S(n) = 2S(n) - S(n), by telescoping the series, we get

$$2S(n) - S(n) = S(n) = 2^{n+1} - 1.$$

Bounding Summations by Mathematical Induction

Sometimes the summations may be too difficult to simplify. An alternative is to simply bound the summation by finding a term that is clearly less than or equal to the actual value of the summation. The objective in this case is to find a term that is as near as possible to the actual simplified value of the summation.

One way of doing this is the use of mathematical induction. We illustrate this using the following example:

Prove: $\sum_{k=1}^{n} k = n(n+1)/2$

Proof: By mathematical induction

Base Case: For n = 1, $\sum_{k=1}^{1} k = 1$. Hence, it holds.

Inductive Step: Assume that it holds for n and verify that it also holds for n + 1.

$$\sum_{k=1}^{n+1} k = \sum_{k=1}^{n} k + (n+1)$$
$$= n(n+1)/2 + (n+1)$$
$$= (n+1)(n+2)/2$$

One need not guess the exact value of the summation in order to use mathematical induction. Induction can be used to show a bound as well.

Prove: $\sum_{k=1}^{n} 3^k = O(3^n)$

Proof: More specifically, we want to prove that: $\sum_{k=0}^{n} 3^k = O(3^n) \leq c3^n$ for some c. This follows from the definition of the O-notation.

Base Case: For n = 0, we have $\sum_{k=0}^{0} 3^k = 1 \leq c\,3^0 = c\,(1)$ as long as $c \geq 1$.

Inductive Step: Assume that the bound holds for n and show that it holds for n+1 as well.

$$\sum_{k=0}^{n+1} 3^k = \sum_{k=0}^{n} 3^k + 3^{n+1}$$
$$\leq c3^n + 3^{n+1}$$
$$\leq (1/3 + 1/c)c3^{n+1}$$
$$\leq c3^{n+1}$$

provided $(1/3 + 1/c) \leq 1$ or $c \geq 3/2$.

Bounding the Terms

If the whole summation is difficult to bound, one can also bound each term of the summation. For example,

$$\sum_{k=1}^{n} k \leq \sum_{k=1}^{n} n$$
$$= n^2$$

In general, for a series $\sum_{k=1}^{n} a_k$, if we let $a_{max} = \max\{a_k \mid 1 \leq k \leq n\}$, then

$\sum_{k=1}^{n} a_k \leq na_{max}$.

A better bound, however, is obtained if the summation can be bounded by a geometric series. Given $\sum_{k=0}^{n} a_k$ and suppose $a_{k+1}/ a_k \leq r$ for some k ≥ 0, where $r < 1$ is a constant. The sum can be bounded by an infinite decreasing geometric series, since $a_k \leq a_0 r^k$, and thus,

$$\sum_{k=0}^{n} a_k \leq \sum_{k=0}^{\infty} a_0 r^k$$

$$= a_0 \sum_{k=0}^{\infty} r^k$$

$$= a_0 (1/(1-r)).$$

Splitting Summations

Finally, another way of obtaining bounds on a difficult summation is to express the series as the sum of two or more series by partitioning the range of the index and then to bound each of the resulting series. For example, in the summation $\sum_{k=0}^{\infty} k^2 / 2^k$ we observe that the ratio of the consecutive terms is:

$$((k+1)^2 /2^{k+1}) / (k^2 /2^k) = (k+1)^2 / 2k^2,$$
$$\leq 8/9$$

if $k \geq 3$. Thus, $\sum_{k=0}^{\infty} k^2 / 2^k$ can be split into two summations

$$\sum_{k=0}^{\infty} k^2 / 2^k = \sum_{k=0}^{2} k^2 / 2^k + \sum_{k=3}^{\infty} k^2 / 2^k$$

$$\leq O(1) + (9/8)\sum_{k=3}^{\infty} (8/9)^k$$

$$= O(1)$$

Summation Formulas and Identities

The usual way of describing the running time of an algorithm is through the use of the summation notation. However, the summation needs to be simplified and this usually is facilitated by the use of existing identities. Some of the popular identities that will be useful later are:

1. $\sum_{k=1}^{\infty} a_k = \lim_{n\to\infty} \sum_{k=1}^{n} a_k$

2. $\sum_{k=1}^{n} (ca_k + b_k) = c\sum_{k=1}^{n} a_k + \sum_{k=1}^{n} b_k$

3. $\sum_{k=1}^{n} O(f(k)) = O(\sum_{k=1}^{n} f(k))$

4. Arithmetic Series: $\sum_{k=1}^{n} k = 1 + 2 + 3 + \ldots + n = n(n+1)/2 = O(n^2)$

5. Geometric Series: $\sum_{k=0}^{n} x^k = 1 + x + x^2 + \ldots + x^n = (x^{n+1} - 1)/(x-1)$, provided x ^{1} 1 and x is real.

6. Infinite Decreasing Series: $\sum_{k=0}^{\infty} x^k = 1/(1-x)$, when $|x| < 1$.

7. Harmonic Series: $H_n = 1 + 1/2 + 1/3 + \ldots + 1/n$

   $= \sum_{k=1}^{n} 1/k,$

   $= \ln(n) + O(1)$

8. The product $\prod_{k=1}^{n} a_k$ can be converted to a formula with a summation using the identity $\log(\prod_{k=1}^{n} a_k) = \sum_{k=1}^{n} \log a_k$

9. Additional formulas can be obtained by integrating or differentiating the formulas given above. For example, differentiating both sides of $\sum_{k=0}^{\infty} x^k = 1/(1-x)$, and multiplying by x, we will get the new identity $\sum_{k=0}^{\infty} kx^k = x/(1-x)^2$.

After introducing some of the mathematical tools for simplifying a summation, let us test you if you can use these tools in actual problems. Try to answer the SAQ's and I encourage you to also work on the SSAQ's in Appendix 2.

SAQ 11-1

Find the closed form expression of the following sums:

1. $\sum_{k=1}^{n}$ (2k -1)

2. $\sum_{k=1}^{n}$ $n/2^k$

3. $\sum_{i=0}^{\infty}$ $i/4^i$

ASAQ's are found in Appendix 1. SSAQ's are given in Appendix 2.

Module 12
Simplifying Recurrences

A **recurrence** is an equation or an inequality that describes a function in terms of its value on smaller inputs. There are two common methods of solving recurrences: **substitution** and **iteration**. However, if the form of the recurrence is

$$T(n) = aT(n/b) + f(n),$$

then this can be solved using a theorem which will be introduced later.

In this module, we provide you with the tools for simplifying recurrence relations. Examples are also provided to illustrate the methods. On your own, you have to discover which method to use when presented with a new recurrence relation. Recurrence relations are usually quite similar to each other. Hence, you can start by comparing them with the examples in this module and adopt the method used in the example almost similar to your new problem.

Objectives

At the end of this module, you should be able to:

1. Identify the most appropriate method to use in simplifying recurrences; and
2. Simplify recurrence relations.

Substitution Method

The idea in this method is to guess a bound and then use a mathematical induction to prove that the guess is right. To illustrate this method, we solve the recurrence

$$T(n) = \begin{cases} 2T(n/2) + n, & n > 1 \\ 1 & , n = 1 \end{cases}$$

Solution: We guess a bound for T(n), say O(n log n). With $T(n) \le O(n \log n)$ it means that we have to show by mathematical induction that T(n) £ cn log n for an appropriate choice of the constant $c > 0$. Again, this follows from the definition of the O-notation.

Base Case: $n = 2$.

$$\begin{aligned} T(2) = 2T(1) + 2 &\le c(2 \log 2) \\ = 4 &\le 2c \end{aligned}$$

which holds provided $c \ge 2$.

Inductive Step: Assume that $T(n/2) \le c(n/2) \log (n/2)$. Substituting this into the recurrence yields

$$\begin{aligned} T(n) &\le 2 (c(n/2) \log (n/2)) + n \\ &\le cn \log (n/2) + n \\ &\le cn \log n - cn + n \\ &\le cn \log n - n(c-1) \\ &\le cn \log n \end{aligned}$$

provided $c > 1$. Hence, T(n) £ O(n log n).

Making a Good Guess

If the recurrence is similar in form to one which you have seen before, guessing a similar solution is reasonable. For example, the recurrence $T(n) = 2T(n/2 + 100) + n$ is intuitively similar to $T(n) = 2T(n/2) + n$. This is because T(n/2) and T(n/2 + 100) is not so different when n is very large. Therefore, our guess for the recurrence $T(n) = 2T(n/2 + 100) + n$ is the solution to $T(n) = 2T(n/2) + n$, which is O(n log n).

Another is to prove a loose upper bound, i.e., give as a first guess a reasonably large value. Then, gradually reduce this initial guess.

Example: $T(n) = 2T(n/2) + n$

Suppose our guess is $T(n) = O(n^2)$. We need to show that $T(n) \le cn^2$ for an appropriate choice of constant $c > 0$. Again assume that this holds for n/2, i.e., $T(n) \le cn^2 /4$. Substituting this into the original recurrence yields,

$$\begin{aligned} T(n) &\le cn^2 /4 + n \\ &\le cn^2 /2 + n \end{aligned}$$

Provided $c \ge 2$, $T(n) \le cn^2$.

Suppose our guess is T(n) = O(n). We therefore have to show that T(n) ≤ cn. Substituting this in the recurrence, we obtain

$$\begin{aligned} T(n) &\le 2(cn/2) + n \\ &\le cn + n \\ &\le 2(c+1)n \end{aligned}$$

Note that T(n) ≤ cn only when c ≤ 0. This means T(n) ∉ O(n). We can therefore select as our next guess for T(n) a function f(n) such that O(n) < f(n) < O(n^2).

Subtleties

Sometimes the mathematics does not seem to work even though you have the right guess. This problem is usually found in the inductive assumption.

Example: T(n) = 2T(n/2) + 1

We guess T(n) = O(n) and try to show that T(n) ≤ cn for some constant c. Substituting our guess into the recurrence, we obtain

$$\begin{aligned} T(n) &\le 2c(n/2) + 1 \\ &= cn + 1 \end{aligned}$$

which does not imply T(n) ≤ cn for any choice of c. The usual approach is to try a larger guess, O(n log n) or O(n^2), but we were only off by 1 in our original guess. We can overcome this by subtracting a lower-order term from our previous guess. Our new guess now is T(n) ≤ cn — b where b is a constant

$$\begin{aligned} T(n) &\le (c\,(n/2) - b\,) + (\,c\,(n/2) - b\,) + 1 \\ &= cn - 2b + 1 \\ &\le cn - b \end{aligned}$$

as long as b ≥ 1.

Changing Variables

Sometimes a little algebraic manipulation of the recurrence can reduce the form of the recurrence to one that is easy to simplify.

Example: $T(n) = 2T(\sqrt{n}) + \log n$

Let $m = \log n$. This yields $T(2^m) = 2T(2^{m/2}) + m$ where we can rename $S(m) = T(2^m)$ to produce $S(m) = 2S(m/2) + m$ which can be simplified to $S(m) = O(m \log m)$. Changing back,

$T(n) = T(2^m) = S(m) = O(m \log m) = O(\log n \log \log n)$

Iteration Method

The **iteration method** is carried out by expanding the recurrence and expressing it as a summation of terms dependent on n and the initial conditions.

Example: $T(n) = 3T(n/4) + n$,

We iterate this recurrence as follows:

$$\begin{aligned} T(n) &= n + 3T(n/4) \\ &= n + 3(n/4 + 3T(n/16)) \\ &= n + 3(n/4 + 3((n/16) + 3T(n/64))) \\ &= n + 3(n/4) + 9(n/16) + 27T(n/64) \end{aligned}$$

How far must we iterate the recurrence before we reach a boundary condition?

The <u>ith</u> term in the series is $3^i (n/4^i)$. The iteration hits $n = 1$ when $(n/4^i) = 1$ or i exceeds $\log_4 n$. By counting the iteration until this point and using the bound $\lfloor n/4^i \rfloor \leq n/4^i$, we discover that the summation contains a decreasing geometric series:

$$\begin{aligned} T(n) &\leq n + 3n/4 + 9n/16 + 27n/64 + \ldots 3^{\log_4 n} O(1) \\ &\leq n \sum_{i=0}^{\infty} (3/4)^i + O(n^{\log_4 3}) \qquad ;; \text{note } 3^{\log_4 n} = n^{\log_4 3} \\ &= 4n + O(n) \\ &= O(n) \end{aligned}$$

Recursion Trees

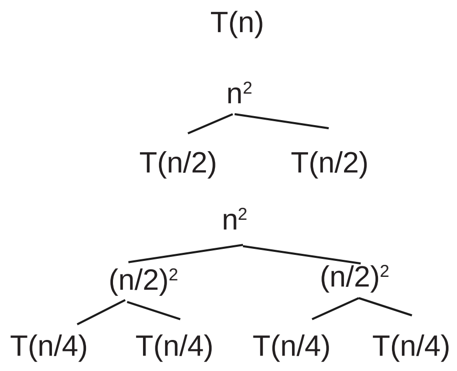
A convenient way of visualizing the iteration method is the use of recursion trees. We illustrate several examples below.

Example: T(n) = 2T(n/2) + n^2

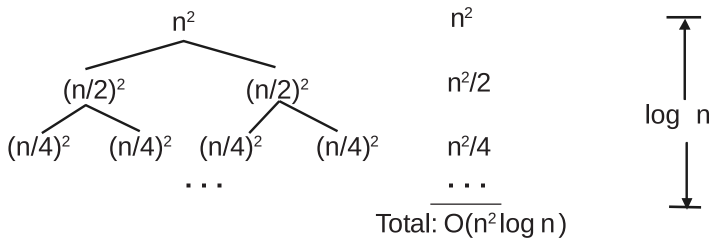
a)

b)

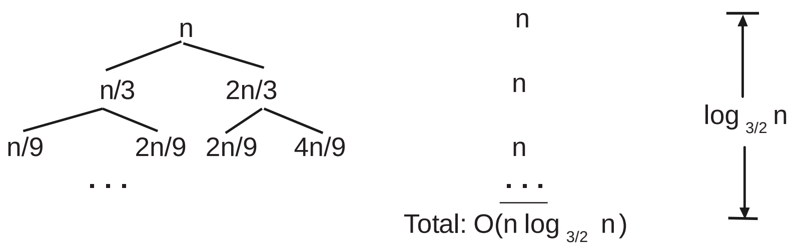
c)



and so on until we produce the following tree:



Example: T(n) = T(n/3) + T(2n/3) + n



Solutions to Recurrences of the Form $T(n) = aT(n/b) + f(n)$

Theorem: The solution to the equation $T(n) = aT(n/b) + \theta(n^k)$ where $a \geq 1$ and $b > 1$, is

$$T(n) = \begin{cases} O(n^{\log_b a}), & a > b^k \\ O(n^k \log n), & a = b^k \\ O(n^k), & a < b^k \end{cases}$$

Proof: Without loss of generality, we assume that n is a power of b, i.e., $n = b^m$. Then $n/b = b^{m-1}$ and $n^k = (b^m)^k = b^{mk} = (b^k)^m$ Assume that $T(1) = 1$, and ignore the constant factor in $\theta(n^k)$. Then, we have

$$T(b^m) = aT(b^{m-1}) + (b^k)^m$$
$$T(b^m)/a^m = T(b^{m-1})/a^{m-1} + (b^k/a)^m$$

we can apply this equation for other values of m, obtaining

$$T(b^{m-1})/a^{m-1} = T(b^{m-2})/a^{m-2} + (b^k/a)^{m-1}$$
$$T(b^{m-2})/a^{m-2} = T(b^{m-3})/a^{m-3} + (b^k/a)^{m-2}$$
$$\ldots$$
$$T(b^1)/a^1 = T(b^0)/a^0 + (b^k/a)^1$$

All the terms on the left cancel the leading term in the right yielding,

$$T(b^m)/a^m = 1 + \sum_{i=0}^{m} (b^k/a)^i$$

Thus,

$$T(n) = T(b^m) = \sum_{i=0}^{m} a^m (b^k/a)^i$$

Case 1. $a > b^k$. The sum is a geometric series with ratio smaller than 1. This would converge to a constant, thus

$$T(n) = O(a^m) = O(a^{\log_b m}) = O(n^{\log_b a}).$$

Case 2. $a = b^k$. Each term in the sum is 1. Since the sum contains $1 + \log_b n$ terms and $a = b^k$ implies that $\log_b a = k$,

$$T(n) = O(a^m \log_b n) = O(n^{\log_b a} \log_b n) = O(n^k \log_b n)$$

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

Case 3. $a < b^k$. The terms in the series are larger than 1, we obtain

$$T(n) = a^m ((b^k/a)^{m+1} - 1) / ((b^k/a) - 1) = O(a^m (b^k/a)^m)$$
$$= O((b^k)^m) = O(n^k).$$

Theorem: The solution to the equation $T(n) = aT(n/b) + \theta(n^k \log^p n)$, where $a \geq 1$, $b > 1$, and $p \geq 0$ is

$$T(n) = \begin{cases} O(n^{\log_b a}), & a > b^k \\ O(n^k \log^{p+1} n), & a = b^k \\ O(n^k \log^p n), & a < b^k \end{cases}$$

Proof: Observe that $\log^p n = \log^p (b^m) = m^p \log^p b$. Working through as in the previous theorem, we will arrive at

$$T(n) = T(b^m) = a^m \sum_{i=0}^{m} (b^k/a)^i \; i^p \log^p b$$

If $a = b^k$, then

$$T(n) = a^m \log^p b \sum_{i=0}^{p} i^p$$
$$= O(a^m m^{p+1} \log^p b).$$

Since $m = \log n / \log b$ and $a^m = n^k$, and b is a constant, we obtain $T(n) = O(n^k \log^{p+1} n)$.

The case $a > b^k$ and $a < b^k$ is similar to the previous theorem. €

Theorem: If $\sum_{i=1}^{k} \alpha_i < 1$, then the solution to the equation $T(n) = \sum_{i=1}^{k} T\alpha_i n) + O(n)$ is $T(n) = O(n)$.

Proof: This can be proven using substitution. That is, we find a constant $c > 0$ such that $T(n) \leq cn$.

$$T(n) = \sum_{i=1}^{k} T(\alpha_i n) + O(n)$$

Let a be the constant in the O(n) (second term).

$$T(n) \le \sum_{i=1}^{k} c(a_i n) + an$$

$$\le cn \sum_{i=1}^{k} \alpha_i + an$$

$$\le cn$$

provided c^{3} $a / (1 - \sum_{i=1}^{k} \alpha_i)$. Note that $c \ge 0$ since $\sum_{i=1}^{k} \alpha_i < 1$.

At this point, it is time for some SAQ's. The SAQ's are designed to check your understanding of how the tools for simplifying recurrences can be applied to specific problems.

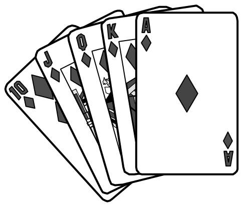
SAQ 12-1

Simplify the recurrence relation T(n) = 3T(n/2) + n, where T(1) = 1.

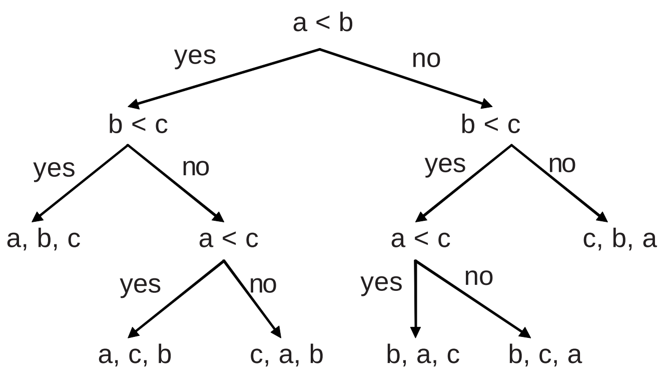
1. Using the substitution method.
2. Using recursion tree.
3. Using the Theorem on T(n) = aT(n/b) + f(n).

ASAQ's are found in Appendix 1. SSAQ's are given in Appendix 2.

Module 13
Analysis of Sorting Algorithms



Sorting is a process of rearranging an initially unordered to produce an ordered collection of keys. For our purposes, we assume that the order is **ascending**, i.e., from lowest to highest. A sorting algorithm sorts **in-place** if only a constant number of elements of the input array are ever stored outside the array. A sorting algorithm is **stable** if the elements with the same value appear in the output array in the same order as they do in the input array. A sorting algorithm is **internal** if no external storage device is used during the sorting process; otherwise, it is considered an **external sorting algorithm**. A sorting algorithm is **comparison-based** if it does the sort by comparing the items until a sorted order is attained. In comparison-based sorting, the idea is to produce the sorted order by comparing the values of the keys. For example, to sort the keys a, b, c depending on the values of a, b, and c, the comparison may follow any path in the following comparison tree.



Comparison tree for sorting a, b, c.

Objectives

At the end of this module, you should be able to:

1. Analyze the best-, worst-, and average-case complexities of some popular sorting algorithm; and
2. Differentiate one sorting algorithm from another.

There are several sorting algorithms which have been developed. In this module, we will discuss only those which are commonly used for sorting small data sets.

Armed with the basic tools for analysis (given in the last three modules), you can now proceed and use these to specific computing problems, such as the topic in this module, the sorting problem. I suggest that you concentrate on how each sorting algorithm is analyzed since you will basically be following the same procedure for analyzing other problems.

Bubble Sort

Bubble sort is the simplest sorting algorithm ever devised. The idea behind its operations can be illustrated by imagining the values of the observations to be objects which are proportional to their weights. These objects are kept in a tube (with water in it) and held vertically upward. Several passes are made and for each pass the lightest is bubbled to the top—hence the name bubble sort. For each pass a pair is exchanged if the pair is not in order, i.e., the lighter one is under. Passes are repeated until no exchange happened during a pass.

```
BubbleSort(n);
begin
   for i = 1 to n-1 do
      for j = n downto i+1 do
          if A[j] < A[j-1]
             swap(A[j], A [j-1]);
end;
```

The following is an example of how the bubble sort works. Only those swaps that occur are shown.

Index:	1 2 3 4 5 6
Original sequence:	4 2 3 1 6 5
First pass (i=1):	4 2 3 1 **5** **6**
	4 2 **1** **3** 5 6
	4 **1** **2** 3 5 6
	1 **4** 2 3 5 6
Second pass (i=2):	1 **2** **4** 3 5 6
Third pass (i=3):	1 2 **3** **4** 5 6
Fourth pass (i=4):	no swapping occurs
Fifth pass (i=5):	no swapping occurs
Final sequence:	1 2 3 4 5 6

To simplify the analysis of the algorithm, we assign a cost of 1 to a statement that cost O(1).

```
BubbleSort(n);
begin

   for i = 1 to n-1 do                         Cost: $\sum_{i=1}^{n-1} 2(n-i)$

      for j = n downto i+1 do                  Cost: 2(n-i)
          if A[j] < A[j-1] then                Cost: 1
              swap(A[j], A [j-1]);             Cost: 1
end;
```

The for j = n downto i+1 loop will be repeated n - (i+1) + 1 times, hence the total cost of the whole loop is 2(n-i). The outer loop, which is the cost of the whole procedure is

$$\text{Cost of BubbleSort} = \sum_{i=1}^{n-1} 2(n-i)$$
$$= 2\sum_{i=1}^{n-1} n - 2\sum_{i=1}^{n-1} i$$
$$= 2n(n-1) - 2(n-1)n/2$$
$$= 2n^2 - 2n - n^2 + n$$
$$= n^2 - n$$
$$= O(n^2)$$

Insertion Sort

Insertion sort is basically how most of us logically sort objects. The fundamental principle of the method is to initially sort two observations of the data set, then take another observation and insert it in its correct position. With three elements already sorted, we take another element and similarly insert it in its correct position. Repeat this process until the last element has been inserted.

```
for  i= 2 to n do
        insert the ith observation in its correct position
            between the first
        and the ith observation
```

```
InsertionSort(n);
begin
   for i= 2 to n do
   begin
       tobs = A[i];
       j= i - 1;
       while (tobs < A[j]) and (j > 0) do
       begin
           A[j+1] = A[j];
           j= j - 1;
       end;
       A[j+1] = tobs;
   end;
end;
```

To illustrate how it works, consider the following example:

Index:	1	2	3	4	5	6
Original sequence :	3	7	8	4	9	6
i = 2	3	**7**	8	4	9	6
i = 3	3	7	**8**	4	9	6
i = 4	3	**4**	7	8	9	6
i = 5	3	4	7	8	**9**	6
i = 6	3	4	**6**	7	8	9
Final sequence :	3	4	6	7	8	9

```
InsertionSort(n);
begin
  for i= 2 to n do
  begin
     tobs = A[i];                              Cost: 1
     j= i - 1;                                 Cost: 1
     while (tobs < A[j]) and (j > 0) do        Cost: 2(i-1)
     begin
        A[j+1] = A[j];                         Cost: 1
        j= j - 1;                              Cost: 1
     end;
     A[j+1] = tobs;                            Cost: 1
  end;
end;
```

The while loop will at worst cost 2(i-1) since j can go all the way from i-1 to 0. The total cost of InsertionSort therefore is:

$$\text{Cost of InsertionSort} = \sum_{i=2}^{n} (3 + 2(i-1))$$

$$= \sum_{i=2}^{n} (1 + 2i)$$

$$= \sum_{i=2}^{n} 1 + 2\sum_{i=2}^{n} i$$

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

$$= n - 1 + n^2 + n - 2$$

$$= n^2 + 2n - 3$$

$$= O(n^2)$$

QuickSort

Among all sorting algorithms, it is widely accepted that **quicksort** is one of the most efficient. The method selects a **pivot** element which ideally is an element whose correct position is near the middle of the data set when it is already sorted. Elements on either side are moved so that the items on one side of the pivot element are smaller than the said observation, whereas those on the other side are larger. This means that the pivot element is already in its correct position. The same process is then applied recursively to the two parts of the data set, i.e., on both sides of the pivot element, until the data set is sorted. This can be summarized as follows:

```
procedure QuickSort
   select a pivot element
   transfer those elements which are on the wrong
         side of the pivot element
   apply QuickSort on both sides of the pivot element
```

```
QuickQort(A,f,l)
begin
   if f ³ l then return
   pivot = A[f]
   i = f+1
   while A[i] < pivot do i = i+1
   j = l
   while A[j] > pivot do j = j-1
   while i < j do
   begin
      swap(A[i], A[j])
      i = i+1
      while A[i] < pivot do i = i+1
      j = j-1
      while A[j] > pivot do j = j-1
   end
   swap(A[f],A[j])
   QuickSort(A,f,j-1)
   QuickSort(A,j+1,l)
end
```

An example of how QuickSort works is the following:

Index:	1	2	3	4	5	6	7	8	9
Original sequence:	5	4	1	7	3	8	2	9	6
f = 1 & l = 9	**5**	4	1	7	3	8	2	9	6
	5	4	1	2	3	8	7	9	6
	3	4	1	2	**5**	8	7	9	6
f = 1 & l = 4	**3**	4	1	2	**5**	8	7	9	6
	3	2	1	4	**5**	8	7	9	6
	1	2	**3**	4	**5**	8	7	9	6
f = 1 & f = 2	**1**	2	**3**	4	**5**	8	7	9	6
	1	2	**3**	4	**5**	8	7	9	6
	1	2	**3**	4	**5**	8	7	9	6
f = 2 & f = 2	**1**	**2**	**3**	4	**5**	8	7	9	6
	1	**2**	**3**	4	**5**	8	7	9	6
	1	**2**	**3**	4	**5**	8	7	9	6
f = 4 & f = 4	**1**	**2**	**3**	**4**	**5**	8	7	9	6
	1	**2**	**3**	**4**	**5**	8	7	9	6
	1	**2**	**3**	**4**	**5**	8	7	9	6
f = 6 & f = 9	**1**	**2**	**3**	**4**	**5**	**8**	7	9	6
	1	**2**	**3**	**4**	**5**	**8**	7	6	9
	1	**2**	**3**	**4**	**5**	6	7	**8**	9
f = 6 & f = 7	**1**	**2**	**3**	**4**	**5**	**6**	7	**8**	9
	1	**2**	**3**	**4**	**5**	**6**	7	**8**	9
	1	**2**	**3**	**4**	**5**	**6**	7	**8**	9
f = 7 & f = 7	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	9
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	9
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	9
f = 9 & f = 9	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

When QuickSort is called, it **partitions** the original array into two. When this is done, it may produce a partition whose size ranges from 0 to n-1. Since the worst case is dictated by the larger partition, we obtain the following worst case performance:

$$T(n) = \max_{i \leq q \leq n-1} (T(q)) + O(n),$$

where O(n) is the cost of partitioning the array and $\max_{i \leq q \leq n-1} (T(q))$ is the cost of applying Quicksort on the larger partition.

To solve this recurrence relation, we use the substitution method. We guess that $T(n) \leq cn$ for some constant c. Substituting this guess, we obtain

$$\begin{aligned} T(n) &= \max_{i \leq q \leq n-1} (T(q)) + O(n), \\ &= c \max_{i \leq q \leq n-1} (cq^2 + c(n-q)^2) + O(n) \end{aligned}$$

The expression $q^2 + (n-q)^2$ achieves a maximum over the range $1 \leq q \leq n-1$ at one of the endpoints, i.e., when q = 1 or q = n-1. This gives us a bound

$$\max_{i \leq q \leq n-1} (q^2 + (n - q)^2 \leq 1^2 + (n-1)^2 = n^2 - 2(n-1)$$

Continuing with our T(n), we obtain

$$\begin{aligned} T(n) &\leq cn^2 - 2c(n-1) + O(n) \\ &\leq cn^2 \end{aligned}$$

provided we select c such that 2c(n-1) is greater than the O(n) of the partition procedure. This shows that

$$T(n) = O(n^2).$$

Although the worst-case complexity of QuickSort is $O(n^2)$, its average-case complexity is O(nlog n). In fact, QuickSort is considered the fastest sorting algorithm in terms of average running time.

MergeSort

Given an array of n elements, the problem is to sort the elements in ascending order using the **MergeSort** strategy. The three steps in this strategy are:

1. Divide the n-element sequence into two sub-sequences of size n/2 elements each (to be specific, one sub-sequence is of size $\lfloor n/2 \rfloor$ and the other is of size $\lfloor n/2 \rfloor$ or $\lceil n/2 \rceil$ depending on whether n is even or odd.
2. Sort the two sub-sequences recursively using MergeSort.
3. Merge the two sub-sequences to form one long sorted sequence.

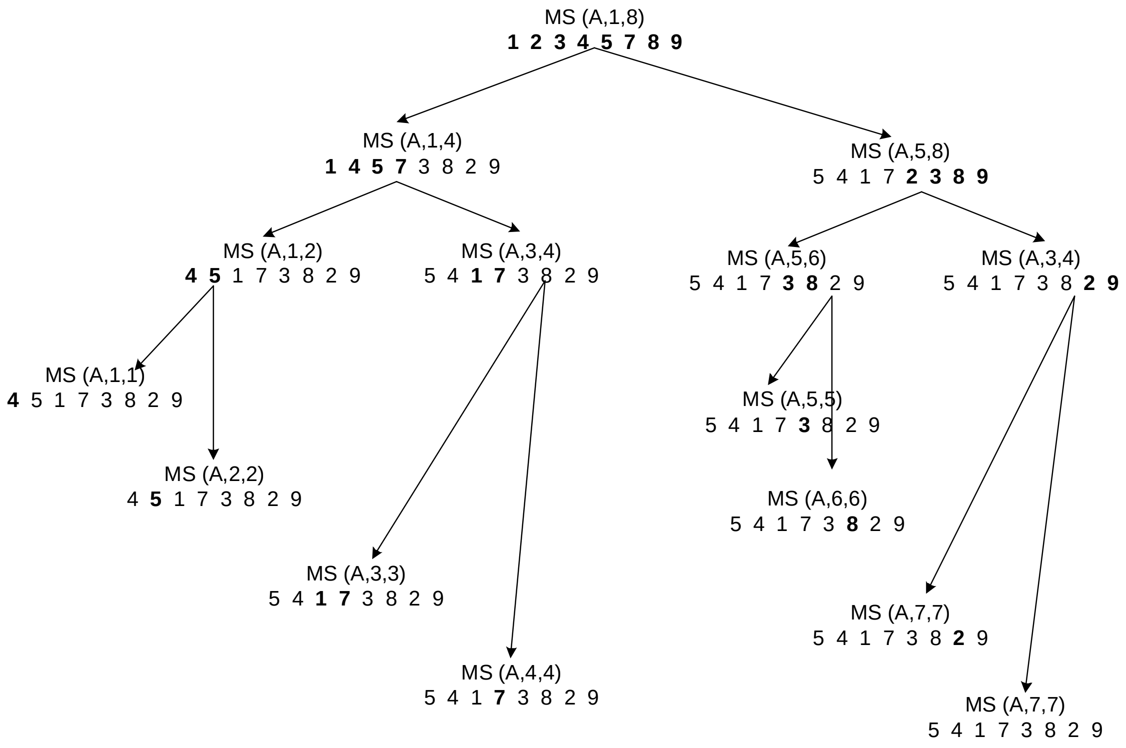
To illustrate this more clearly, we present the MergeSort algorithm below.

```
MergeSort(A,f,l)
begin
    if f < l then
        q = (f+l) div 2
        MergeSort(A,f,q)
        MergeSort(A,q+1,l)
        Merge(A,f,q,l)
end
```

To illustrate the execution of the algorithm, we consider the following example.

Index:	1 2 3 4 5 6 7 8
Original sequence:	5 4 1 7 3 8 2 9

Let MergeSort(A,f,l) be represented by MS(A,f,l). In the sequence shown below, the call to MS() is the array produced after completely executing the call to MS(). Hence, after the first call to MS() is completed, we end up with a sorted sequence of the original sequence.



The procedure Merge(A,f,q,l) merges two sorted sub-sequences into one sorted sequence. As illustrated in the execution tree, the subsequences:

$$1\ 2\ 3\ 4\ 5\ 7\ 8\ 9 \quad \leftarrow \quad \begin{array}{c} 1\ 4\ 5\ 7 \\ \\ 2\ 3\ 8\ 9 \end{array}$$

This can be done by comparing the first elements of the two sub-sequences. The smaller of the two is placed in an output array. The smaller element which was transferred to the output array is replaced by the next element in the sub-sequence where it came from. This is repeated until one of the sub-sequences runs out of elements, in which case the elements in the sub-sequence with remaining elements are copied to the output array.

```
Merge(A,f,q,l)
begin
   i = f
   j = q+1
   k = f
   while (i ≤ q) and (j ≤ l) do
      if A[i] < A[j] then
         B[k] = A[i]
         i = i+1
      else
         B[k] = A[j]
         j = j+1
      k = k+1
   end
   if i ≤ q then
      for h = i to q do
         B[k] = A[h]
         k = k+1
   elseif j ≤ l do
      for h = j to l do
         B[k] = A[h]
         k = k+1
   for k = l to l do
      A[k] = B[k]
end
```

From the algorithm, we note that each element of array A is transferred to array B once. At the end of the algorithm, these elements are returned to array A from B. Hence, a call to Merge(A,1,(n+1) div 2, n) will cost 2n. Let T(n) be the cost of a call to MergeSort(A,1,n).

```
MergeSort(A,f,l)
begin
   if f < l then
      q = (f+l) div 2             Cost: 1
      MergeSort(A,f,q)            Cost: T(n/2)
      MergeSort(A,q+1,l)          Cost: T(n/2)
      Merge(A,f,q,l)              Cost: 2n
end
```

Combining the cost of the recursive calls and the cost of Merge, we get

T(n) = 1 + 2n + 2T(n/2)

We have earlier solved this recurrence to be

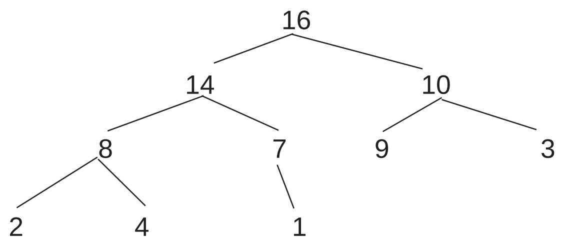
T(n) = O(n log n).

HeapSort

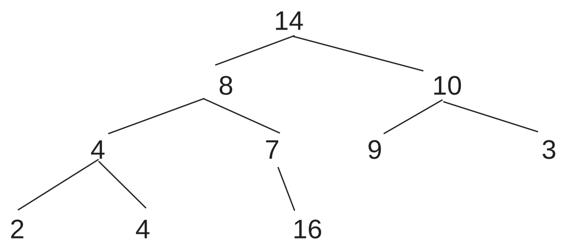
Given a procedure for building a heap **(Build-Heap)** and a procedure for restoring the heap condition **(Heapify)**, we can formulate the algorithm for **HeapSort** as follows:

```
HeapSort(A,n)
begin
    Build-Heap (A)
    for i = n downto 2 do
    begin
        swap(A[1],A[i])
        Heapify(A,1,i-1)
    end
end
```

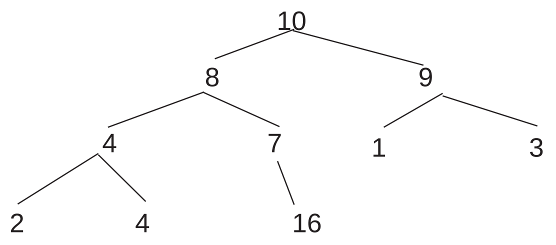
To illustrate the execution of this algorithm, consider the following example:



We swap A[1] with A[10] and apply Heapify on A[1] excluding A[10], i.e.,



Then we swap A[1] with A[9] and apply Heapify on A[1] excluding A[9], i.e.,



and so on until we get the sequence sorted.

To analyze HeapSort, we have to compute the costs of building a heap and restoring the heap property.

```
procedure Heapify(key,i,n)
begin
    l = left(i)
    r = right(i)
    if l ≤ n and key(l) > key(i) largest = l
    else largest = i
    if r ≤ n and key(r)> key(largest) largest = r
    if largest ≠ i
        swap(key(i), key(largest))
        Heapify(key, largest,n)
end
```

Calling the procedure Heapify(key,1), excluding the recursive call to Heapify(key, largest), the cost of the other statements is O(1). If we let T(n) as the cost of a call to Heapify(key,1), the call to Heapify(key,largest) is approximately T(n/2). Hence,

$$T(n) = O(1) + T(n/2)$$

which simplifies to

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

Next, we look at the cost of building a heap.

```
Build-Heap(key)
begin
    for i = $\lfloor n/2 \rfloor$ downto 1 do
        Heapify(key,i,n)
end
```

Intuitively, the total cost can be computed by considering that each call to Heapify cost O(log n) time, and there are O(n) such calls. Thus, the running time is at most O(n log n). This cost though correct, is not accurate.

We can get a more accurate cost by observing that the time for Heapify varies with the height of the node in the tree, and the height of most nodes are small. In fact, in an n-element heap, there are at most $\lceil n/2^{h+1} \rceil$ nodes at height h. The time required by Heapify when called on a node of height h is O(h), this implies a total cost for Build-Heap of

$$\sum_{h=0}^{floor(\log n)} \lceil n/2^{(h+1)} \rceil \, O(h) = O\left(n \sum_{h=0}^{floor(\log n)} h/2^h\right)$$

By substituting x = 1/2 in the identity

$$\sum_{k=0}^{\infty} kx = x/(1-x)^2 ,$$

we obtain

$$\sum_{h=0}^{\infty} n/2^h = (1/2)/(1-1/2)^2 = 2.$$

Thus, the running time of Build-Heap can be bounded as

$$O(n \sum_{h=0}^{floor(\log n)} h/2^h) \leq O(n \sum_{h=0}^{\infty} h/2^h) = O(n).$$

Heapsort calls Heapify n-1 times and each time it is called the cost is O(log i), where i is 1 less than the size of the previous call. Hence, the total cost is

$$\sum_{i=2}^{n} O(\log i) = O(n \log n)$$

plus the cost of Build-Heap which is O(n). Hence, a total of O(n log n).

Counting Sort

This is one of the **non-comparison-based** sorting algorithms. The algorithm assumes that each of the n input elements is an integer in the range 1 to k, for some integer k, where k = O(n). The sorting process runs in O(n) time. The idea is to count the number of elements less than a particular element. This number plus 1 is the position of the said element in the sorted order of the elements. Let n be the size of array (input).

```
Counting-Sort(A,B,k)
begin
  for i = 1 to k do c[i] = 0                        Cost: O(k)
  for j = 1 to n do c[A[j]] = c[A[j]] + 1           Cost: O(n)

  {c[i] now contains the number of
    elements equal to i}

  for i = 2 to k do c[i] = c[i] + c[i-1]            Cost: O(k)

  {c[i] now contains the number of
    elements less than or equal to i}

  for j = n downto 1 do                              Cost: O(n)
  begin
     B[c[A[j]]] = A[j]
     c[A[j]] = c[A[j]] - 1
  end

  {B now contains the sorted elements}

end
```

Hence, the cost will depend on whether the value of k is greater or less than n. In O-notation, we can write this cost of counting sort as O(max(n,k)) = O(n+k) time. When k = O(n), counting sort requires O(n) time.

Radix Sort

Radix sort assumes that each element in the n-element array has d digits, where the rightmost digit is the **lowest-order digit** and dth digit is the **highest-order digit**.

```
 Radix-Sort(A,d)
begin
    for i = 1 to d
      do use a stable sort to sort
         array A on digit i.
end
```

To illustrate how the algorithm works, we consider the following example:

(a)	(b)	(c)	(d)
	↓	↓	↓
329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

where (a) is the original input and (b-d) is the sorting of the digits from lowest-order to the highest order digits.

When each digit in the range 1 to k, k is not too large, counting sort is the obvious choice for the stable sorting algorithm in the description. Each pass over n d-digit number then takes time O(max(n,k)) = O(n+k). There are d passes, so the total time for the radix sort is O(dn+dk). When d is constant and k is O(n), radix sort runs in linear time.

Bucket Sort

Bucket sort runs in linear time on the average. Whereas counting sort assumes that the input consists of integers in a small range, bucket sort assumes that the input is generated by a random process that distributes elements uniformly over the interval [0, 1).

The idea of bucket sort is to divide the interval [0,1) into n equal-sized subintervals, or **buckets**, and then distribute the n input numbers into the buckets. Since the inputs are uniformly distributed over [0,1), we do not expect many numbers to fall into each bucket. To produce a sorted sequence, we simply sort the numbers in each bucket and then go through the buckets in order, listing the elements in each.

```
Bucket-Sort(A)
begin
    for i = 1 to n do
        Insert A[i] into list B[ë n A[i] û ]
    for i = 0 to n-1 do
        Sort list B[i] with insertion sort
    Concatenate the lists B[0], B[1], ...,
        B[n-1] together in order
end
```

The worst-case cost is O(n^2) and this happens when all the elements happens to fall in exactly one bucket. In which case, the cost is the same as that of InsertionSort.

Now, we are ready for the last set of SAQ's. Relax and try to answer the SAQ's. This set will basically reinforce your skills in analyzing algorithms.

SAQ 13-1

What is the running time for the input sequence 1, 2, 3, …, n (already sorted) of the sorting algorithms?

1. BubbleSort
2. InsertionSort
3. QuickSort
4. MergeSort
5. HeapSort.

ASAQ's are found in Appendix 1. SSAQ's are given in Appendix 2.

Congratulations! You have reached the end of the unit, also the end of the course. Now, try the sample examination in the next page. As most UP courses require you to get 60% to pass the course, you need to get 21 correct answers out of the 36 items in the sample examination.

As in the first sample examination, try to answer the exam in one hour. Do it yourself and compare your answers with those given in the ASAQ's.

Good luck!

Unit 2 Sample Examination

Instruction. Multiple choice. Simply encircle the letter corresponding to your answer to the question.

1. In an nxn two dimensional array, searching for an element will require ___ worst-case time.

   a) O(n)
   b) O(nlog n)
   c) O(n^2)
   d) O(n^3)
   e) NOTA (None of the Above)

2. When the n elements in a linear linked list are already sorted, searching for an element will require ___ worst-case time.

   a) O(1)
   b) O(log n)
   c) O(n)
   d) O(n^2)
   e) NOTA

3. In a balanced binary search tree, finding the minimum will require ______ worst-case time.

   a) O(1)
   b) O(log n)
   c) O(n)
   d) O(n log n)
   e) NOTA

   For numbers 4-5, let V and E be the number of vertices and edges, respectively, in a graph.

4. The memory required to represent this graph using adjacency lists is: ___.

   a) O(V)
   b) O(E)
   c) O(V+E)
   d) O(V^2)
   e) NOTA

5. The memory required to represent this graph using adjacency matrix is: ___.

   a) O(V)
   b) O(E)
   c) O(V+E)
   d) O(V^2)
   e) NOTA

6. Inserting a new element into a BST of height h, will require ______ worst-case time.

   a) O(h)
   b) O(1)
   c) O(log h)
   d) O(h^2)
   e) NOTA

7. In a graph G=(V,E), BFS will require _____ worst-case time.

   a) O(V)
   b) O(E)
   c) O(V+E)
   d) O(V^2)
   e) NOTA

8. In a graph G=(V,E), DFS will require _____ worst-case time.

   a) O(V)
   b) O(E)
   c) O(V+E)
   d) O(V^2)
   e) NOTA

9. $\sum_{i=1}^{n}$ (i-1) simplifies to:

   a) n(n+1)/2
   b) n(n-1)/2 + 1
   c) n(n-1)/2
   d) n(n+1)/2 + 1
   e) NOTA

10. $100n^2 + n^3 + 1000n + 2000$ is:

   a) O(1)
   b) O(n)
   c) $O(n^2)$
   d) $O(n^3)$
   e) NOTA

11. T(n) = 2T(n/2) + n simplifies to:

   a) O(n)
   b) O(nlog n)
   c) $O(n^2)$
   d) $O(n^3)$
   e) NOTA

12. Consider the code:

```
x := 0
for j := 1 to n do
    for k := 1 to j do
        x := x + j*k
```

   The cost of the algorithm is

   a) O(n)
   b) O(log n)
   c) O(n log n)
   d) $O(n^2)$
   e) NOTA

13. When the data is sorted and stored in an array of size n, the cost of searching an element will be:

    a) O(1)
    b) O(log n)
    c) O(n)
    d) O(n log n)
    e) NOTA

For numbers 14-17, consider the code below. Let the pointer 'list' be pointing to an existing list and let n be the number of nodes in the list. The code below checks if x is in the list or not.

```
search(list,x)
begin
   ptr := list
   while ptr ≠ nil and info(ptr) ≠ x do
      ptr = next(ptr)
   if ptr ≠ nil then foundx := true
   else foundx := false
end.
```

14. The minimum number of times the condition in the while loop is executed is:

    a) O(1)
    b) O(log n)
    c) O(n)
    d) 0
    e) NOTA

15. If found true at the end of the execution, the number of times the condition in the while loop executed is:

    a) O(1)
    b) O(log n)
    c) O(n)
    d) 0
    e) NOTA

16. If the elements in the list sorted and found true at the end of the execution, the number of times the condition in the while loop executed is:

    a) O(1)
    b) O(log n)
    c) O(n)
    d) 0
    e) NOTA

17. If found false at the end of execution, the number of times the condition in the while loop is executed is:

    a) O(1)
    b) O(log n)
    c) O(n)
    d) 0
    e) NOTA

18. Sorting an already sorted array using Heapsort will require:

    a) O(1)
    b) O(n)
    c) O(n^2)
    d) O(n^3)
    e) NOTA

19. Which of the following requires O(n) worst-case time to compute:

    a) searching for a value in an array of size n
    b) searching for a value in a linked list of size n
    c) searching for a value in a sorted linked list of size n
    d) All of the above (AOTA)
    e) None of the above (NOTA)

20. Inserting a new element into its correct position in a given sorted linked list will require ____ time.

    a) O(1)
    b) O(log n)
    c) O(n)
    d) O(n^2)
    e) NOTA

21. The best-case cost for merging two sorted lists of sizes m and n is:

    a) O(m)
    b) O(n)
    c) O(m+n)
    d) O(min(m,n))
    e) O(max(m,n))

22. The term n log n - nln n + n log log n + $n\log^2 n$ is an element of:

    a) O(nlog n)
    b) O(nln n)
    c) O(nlog log n)
    d) $O(n\log^2 n)$
    e) NOTA

23. The term $2^n - e^n + 10^n + n1.5^n$ is an element of:

    a) 2^n
    b) e^n
    c) 10^n
    d) $n1.5^n$
    e) NOTA

24. Which of the following is always true for any problem P?

    a) worst-case time ≥ average-case time ≥ best-case time
    b) average-case time ≥ worst-case time ≥best-case time
    c) best-case time ≥ average-case time ≥ worst-case time
    d) worst-case time ≥ best-case time ≥average-case time
    e) average-case time ≥ best-case time ≥ worst-case time

25. The recurrence relation T(n) = 2T(n/2) + 1 is:

    a) O(n)
    b) O(1)
    c) O(log n)
    d) O(nlog n)
    e) NOTA

26. The recurrence T(n) = T(n-1) + n simplifies to:

    a) O(n)
    b) O(n^2)
    c) O(log n)
    d) O(nlog n)
    e) NOTA

27. Given an input that is already sorted. Quicksort will require ____ time.

    a) O(n)
    b) O(1)
    c) O(n^2)
    d) O(nlog n)
    e) NOTA

28. Which sorting algorithm involved at most O(n) data move?

    a) Quicksort
    b) Mergesort
    c) Insertion sort
    d) selection sort
    e) bubble sort

29. Which sorting algorithm has O(n log n) worst-case?

    a) Quicksort
    b) Mergesort
    c) Insertion sort
    d) selection sort
    e) bubble sort

30. Given the code:

    ```
    sum(n)
      if n=1 return(A[1])
      else return(A[n]+sum(n-1))
    ```

    The cost of this recursive procedure is:

    a) O(1)
    b) O(log n)
    c) O(n)
    d) O(n^2)
    e) NOTA

31. Given the code:

```
for j = 1 to n do
  for k = 1 to j do
    sum = sum + k;
```

The statement sum = sum + k will be executed _____ times.

a) $(n^2+n)/2$
b) n^2
c) $(n^2-n)/2$
d) nj
e) NOTA

32. In Heapsort, the maximum element of the set to be sorted is found after ____ time.

a) O(1)
b) O(log n)
c) O(n)
d) O(n log n)
e) NOTA

33. In an adjacency matrix representation of a graph, checking if the edge (i,j) exists or not can be done in:

a) O(1)
b) O(log n)
c) O(n)
d) $O(n^2)$
e) NOTA

34. In an adjacency lists representation of a graph, checking if the edge (i,j) exists or not can be done in:

a) O(1)
b) O(log n)
c) O(n)
d) $O(n^2)$
e) NOTA

35. The class of algorithms in O(m+n) is the same as the class:

a) O(m)
b) O(n)
c) O(max(m,n))
d) O(min(m,n))
e) NOTA

References

Aho, A.V., Hopcroft, J.E., and Ullman, J.D. (1983). Data Structures and Algorithms. Addison-Wesley.

Albacea, E.A. (1996). Algorithm Analysis and Design, ICS Lecture Notes Series, ICS.

Albacea, E.A. and Samaniego, J.M. (1997). Lecture Notes in Analysis of Algorithms, Symposium on Mathematical Aspects of Computer Science, Baguio City, May 28-June 1, 1997.

Cormen, T.H., Leiserson, C.E., and Rivest, R.L. (1990). Introduction to Algorithms. MIT Press.

Gonnet, G.H., and Baeza-Yates, R. (1991). Handbook of Algorithms. Addison-Wesley.

Knuth, D.E. (1969). The Art of Computer Programming: Fundamental Algorithms (Vol 1). Addison-Wesley.

______________. (1963). The Art of Computer Programming: Sorting and Searching (Vol 3). Addison-Wesley.

Sedgewick, R. Algorithms. (1988). Addison-Wesley.

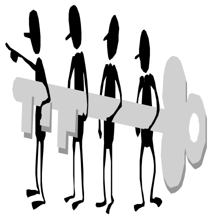
Standish, T.A. (1994). Data Structures, Algorithms, and Software Principles. Addison-Wesley.

Tenenbaum, A.M., and Augenstein, M.J. (1981). Data Structures Using Pascal. Prentice-Hall.

Weiss, M.A. (1992). Data Structures and Algorithm Analysis. Benjamin/ Cummings.

Wirth, N. (1976). Algorithms + Data Structures = Programs. Prentice-Hall.

Appendix 1
Answers to Self-Assessment Questions



If your solutions are different from what are given here, it does not mean that your solutions are wrong. For each of the questions given, there are actually many possible solutions. The ASAQ's presented here were formulated by a person who has at least 15 years experience in solving problems like those given in the SAQ's. So do not get discouraged if your solution is different. Your are not expected to produce solutions exactly the same as those in the ASAQ's.

What is important is for you to try the SAQ's and slowly improve your answers. I am sure, through practice, you will be able to produce answers better than those given in the ASAQ's.

To check if your answers are right, try to hand execute your solutions. Once you are convinced that these are correct, have them double checked by your tutor. He/She will be more than willing to assist you in checking if the solutions you formulated are correct or not.

Basic Data Structures

ASAQ 1-1

1. **Write a procedure for reversing the elements of an array.**

```
Reverse(A,n)
begin
    i = 1
    j = n
    while i < j do
        swap(A[i],A[j])
        i = i + 1
        j = j - 1
end
```

2. **Write a procedure for merging two sorted arrays into one.**

```
Merge(A,B,C)
begin
    a = size(A)
    b = size(B)
    i = j = k = 1
    while i £ a and j £ b do
        if A[i] < B[j] then
            C[k] = A[i]
            i = i + 1
        else
            C[k] = B[j]
            j = j + 1
        k = k + 1
    end
    if i > a then
        for l = j to b do
            C[k] = B[l]
            k = k + 1
    if j > b then
        for l = i to a do
            C[k] = A[l]
            k = k + 1
  end
```

3. **Formulate two procedures for inserting a new element into a sorted array. Compare the two procedures you formulated.**

```
LinearInsert(A,n,x)
begin
   i = 1
   while x < A[i] do
      i = i + 1
   t1 = x
   while i ≤ n do
      t2 = A[i]
      A[i] = t1
      t1 = t2
   A[n+1] = t1
end
```

4. **Write a procedure for finding the minimum in an array and deleting it from the array.**

```
DeleteMinimum(A,n)
begin
   for i = n-1 downto 1 do
      if A[n] > A[i] then swap(A[n],A[i])
end
```

Basic Data Structures

ASAQ 1-2

1. **Represent the moves of a tic-tac-toe game using multi-dimensional arrays. Formulate an algorithm for checking whether a move by one player will result to a win, a draw, or a situation where the other player is required to make his move.**

   Initialize each cell in the array to 0. Let 1 be a move by player 1 and -1 be a move by player 2.

```
    EvaluateMove(A)
    begin
       for i = 1 to 3 do
          sum = 0
          for j = 1 to 3 do
             sum = sum + A[i,j]
             if A[i,j] = 0 then GameOver = false
          if sum = 3 then
             write("Player one wins")
             return
          if sum = -3 then
             write("Player two wins")
             return
    if GameOver then
         write("It is a draw")
         return
    else
    write("Next player must make a move");
    return
end
```

2. **Given a square matrix which is stored in a two-dimensional array. Check if the matrix is symmetrical about the main diagonal.**

```
Symmetric(A,n)
begin
   Symmetric = true
   for i = 2 to n do
      for j = 1 to i-1 do
    if A[i,j] ¹ A[j,i] then Symmetric = false
end
```

3. **Given a two-dimensional nxn matrix M. Check if M is triangular, i.e. every element below the main diagonal is zero.**

```
LowerTriangular(A,n)
begin
   LowerTraingular = true
   for i = 2 to n do
      for j = 1 to i-1 do
    if A[i,j] ¹ 0 then LowerTriangular = false
end
```

4. **Given an nxn matrix A and a nxn matrix B. The matrix product AxB is an nxn matrix C, where the element C_{ij} of C is given by:**

$$C_{ij} = \sum_{i<=k<=p} A_{ik}B_{kj}$$

Write a procedure for multiplying any two matrices A and B.

```
MatrixMultiply(A,B,C)
begin
   for i = 1 to n do
      for j = 1 to n do
    t = 0
    for k = 1 to n do
       t = t + A[i,k] * B[k,j]
    C[i,j] = t
end
```

Linked Lists

ASAQ 2-1

1. **Let L1 and L2 be two sorted linear linked lists. Write the procedure Union(L1, L2). The procedure produces the union of the elements in L1 and L2.**

```
Union(L1,L2)
begin
   /* make L12 point to its first element if either L1
      or L2 or both are not empty */
   if L1 ¹ nil and L2 ≠ nil then
      if info(L1) > info(L2) then
         L12 = ptr = L2
     else
         L12 = ptr = L1
   elseif L1 ≠ nil then
      L12 = ptr = L1
   elseif L2 ≠ nil then
      L12 = ptr = L2
```

```
        /* similar to merge, except that if both leading elements
           are equal one in L2 is skipped. */

    while L1 ≠ nil and L2 ≠ nil do
           if info(L1) < info(L2) then
              next(ptr) = L1
              L1 = next(L1)
           elseif info(L1) > info(L2) then
              next(ptr) = L2
              L2 = next(L2)
           else
              tptr = L2
              L2 = next(L2)
              freenode(tptr)
        end
        /* one of the list may possibly have some elements left */
        if L1 ≠ nil then next(ptr) = L1
        if L2 ≠ nil then next(ptr) = L2
     end
```

2. **Given a list and a pointer (ptr) to one of the elements of the list. Write a procedure Swap(ptr, next(ptr)) that swaps the nodes pointed by ptr and next(ptr). The swap should be done without touching the information stored in the node, i.e., only the pointers will be manipulated. Do this for a linear linked list.**

```
    SwapLinearLL(ptr, next(ptr))
    begin
        nextptr = next(ptr)
        next(ptr) = next(nextptr)
        next(nextptr) = ptr
        /* if ptr = list then okay, else locate the node just
           before the one pointed by ptr */
    if ptr = list then
          list = nextptr
    else
      leftptr = list
      while next(leftptr) ¹ ptr do
           leftptr = next(leftptr)
           next(leftptr) = nextptr
    end
```

Linked Lists

ASAQ 2-2

Given two circular linked lists L1 nd L2. Formulate an algorithm for merging the two circular linked lists.

```
Merge2Clists(L1,L2)
begin
    ptr = next(L1)
    next(L1) = next(L2)
    next(L2) = ptr
end
```

Which Lists

ASAQ 2-3

Given a list and a pointer (ptr) to one of the elements of the list. Write a procedure Swap(ptr, next(ptr)) that swaps the nodes pointed by ptr and next(ptr). The swap should be done without touching the information stored in the node, i.e., only the pointers will be manipulated. Do this for a doubly linked list.

```
SwapDoublyLL(ptr, right(ptr))
begin
    rightptr = right(ptr)
    right(ptr) = right(rightptr)
    left(rightptr) = left(ptr)
    if right(rightptr) ¹ nil then
        left(right(rightptr)) = ptr
        right(rightptr) = ptr
    else
        right(rightptr) = ptr
    if list ¹ ptr then
        right(left(ptr)) = rightptr
        left(ptr) = rightptr
    else
        left(ptr) = rightptr
end
```

Stacks

ASAQ 3-1

1. **Two stacks S1 and S2 can be implemented using one array. Write procedures: PopS1(Stack), PushS1(Stack, x), PopS2(Stack), PushS2(Stack,x). The procedures should not declare an overflow unless every slot in the array is used.**

   Let the array Stack[1..n] be used for storing the stacks S1 and S2, where S1 is growing upward from index 1 and S2 is growing downward from index n. The top of stacks are pointed to by top1 and top2, respectively.

```
PopS1(Stack)
begin
    if top1 < 1 then error("Stack underflow")
   else
        top1 = top1 - 1;
        return(Stack[top1])
end

PushS1(S1,x)
begin
    if top1 ≤ top2 then
        Stack[top1] = x
        top1 = top1 + 1
   else
        error("Stack overflow")
end

PopS2(Stack)
begin
    if top2 > n then error("Stack underflow")
   else
        top2 = top2 + 1;
        return(Stack[top2])
end
```

```
PushS2(S1,x)
begin
    if top2 ≥ top1 then
        Stack[top2] = x
        top2 = top2 - 1
    else
        error("Stack overflow")
end
```

2. **Formulate an algorithm for printing the elements of a stack in reverse order.**

```
ReverseStackElements
begin
    do while top > 0
        x = Pop(Stack)
        Push(AuxStack,x)
    end
    do while Auxtop > 0
        print(Pop(AuxStack))
    end
end
```

3. **Formulate an algorithm for simulating a parking lot that operates on a last-in, first-out basis (LIFO). The data structure used is the stack due to the nature of the operations described.**

```
/* identifiers used */

  parksize = 10
  rate = 0.1 /* rate of request/hour */

  space : array [1..parksize]
  sp
  dtime
  currtime
  earliest
  nrequest, ndenied,naccept
```

```
LIFOParkSimulate
```

/* The parking area has a single-lane space enough for 10 cars. Before a car is admitted, the owner is asked for his departure time, if the said departure time is later than the last car admitted, that he will be denied access to the space. Otherwise he will be admitted. The first car is always admitted.

```
  This algorithm computes the number of cars who made request for
  access, number of those denied access and number of those admitted
  until the space is filled.  */
  begin
    print('The parking space opens at 8 AM and closes');
    print('at 10 PM.  Requests therefore should be made');
    print('within this time period, i.e., 8.0-22.0');
    sp= 1;
    currtime = 8;
    print('Time at the moment is: ',currtime:4:2);
    nrequest = 1;
    ndenied = 0;
    /* admit the first car */
    print('Departure Time: ');
    /* Ask driver for departure time */
    read(dtime);
    Park(dtime);
    print('*** Access Allowed ***');
    /* set time departure of outermost car */
    earliest = dtime;
    while (sp <= parksize) and (sp > 0) do
      /* increment time */
      currtime = currtime + rate;
      print('Time at the Moment is: ',currtime);
      while (currtime > earliest) and (sp > 0) do
           Depart;
           earliest = space[sp];
     /* request for access */
     print('Departure Time : ');
     /* Ask driver for departure time */
     read(dtime);
     nrequest = nrquest + 1;
     if dtime > earliest then
           ndenied = ndenied + 1;
           print('*** Access Denied ***');
     else
           Park(dtime);
           naccept = naccept + 1;
           earliest = dtime;
           print('*** Access Allowed ***');
 print('Number of Car Requests : ',nrequest);
 print('Number of Cars  Denied :  ',ndenied);
 print('Number of Cars Accepted : ',naccept);
end
```

```
Park(s)     /* Push */
begin
     if sp £ parksize then
     space[sp] = s;
     sp = sp + 1;
    else print('XXX No more Space left');
end

Depart  /* Pop */
begin
    if sp < 1 then print('XXX No more car in the parking space XXX')
    else
     print('### A Car Departed ###');
     sp = sp - 1;
end;
```

Queues

ASAQ 4-1

1. **Formulate an algorithm for printing the elements of a queue in reverse order.**

```
ReverseQElements
begin
   do while head ≠ tail
      x = Remove(Queue)
      Push(Stack,x)
   end
   do while top > 0
      print(Pop(Stack))
   end
end
```

2. **Formulate an algorithm for simulating a parking lot that operates on a first-in, first-out (FIFO) basis. The data structure used is the stack due to the nature of the operations described.**

```
/* identifiers used */
   parksize = 10;
   rate  = 0.1;    /* rate of requests/hour */
```

```
    space: array[1..parksize]
    front, rear
    dtime
    currtime
    earliest
    nrequest, ndenied, naccept
```

FIFOParkSimulate
/* The parking area has a single-lane space enough for 10 cars. Before a car is admitted, the owner is asked for his departure time, if the said departure time is earlier than the last car admitted then he will be denied access to the space. Otherwise he will be admitted. The first car is always admitted.

This algorithm computes the number of cars who made request for access, number of those denied access, and number of those admitted until the space is filled. */

```
begin
   print('The parking space opens at 8 AM and close');
   print('at 10 PM.  Requests therefore should be made');
   print('within this time period, i.e., 8.0-22.0');
   print(' (8:00 AM to 10:00 PM)');
   rear = parksize;
   front = parksize;
   currtime= 8;
   print('Time at the moment is: ',currtime:4:2);
   nrequest = 1;
   ndenied = 0;
   naccept = 1;
   /* admit the first car */
   print('Departure Time: ');
   /* Ask driver for departure time */
   read(dtime);
   Park(dtime);
   print('*** Access Allowed ***');
   /* set time departure of outermost car */
   earliest = dtime;
   while <front <> rear) do
     /* increment time */
     currtime = currtime + rate;
     writeln('Time at the Moment is: ',currtime:4:2);
     while (currtime >= space[front]) and (front <>rear) do
             Depart;
             earliest = space[rear];
     /* request for access */
     print('Departure Time : ');
          /* Ask driver for departure time */
     read(dtime);
     nrequest = nrquest + 1;
     if dtime <earliest then
             ndenied = ndenied + 1;
             print('*** Access Denied ***');
     else
            Park(dtime);
             naccept = naccept + 1;
            earliest = dtime;
             writeln('*** Access Allowed ***');
   writeln('Number of Car Requests : ',nrequest);
   writeln('Number of Cars  Denied :  ',ndenied);
   writeln('Number of Cars Accepted : ',naccpet);
end

Park(s)      /* Insert */
begin
    if rear= parksize then
       rear = 1
   else
       rear = rear + 1;
    if rear<>front then
       space[rear] = 5
   else
       print('XXX No more space left XXX');
end;

Depart     /* Remove */
begin
     if front = rear then
         print('XXX No more Car in the Parking Space XXX")
   else
        if front = parksize then
     front = 1
        else
          print('### A Car Departed ###");
          front = front + 1;
end;
```

Queues

ASAQ 4-2

Formulate the procedures for the dequeue operation: InsertatTail (Dequeue,x), RemoveatHead(Dqueue), InsertatHead (Dequeue,x), RemoveatTail(Dequeue).

```
/* Similar to Insert(Queue,x) */

InsertatTail(Dequeue,x)
begin
    if tail = queuesize then tail = 1
    else tail = tail + 1
    if tail $^1$ head then Dequeue[tail] = x
    else error("Dequeue overflow")
end

/* Similar to Remove(Queue) */
RemoveatHead(Dequeue)
begin
    if head = tail then error("Queue underflow")
    x = Dequeue[head]
    if head = queuesize then head = 1
    else head = head + 1
    return(x)
end
InsertatHead(Dequeue,x)
begin
    if head = 1 then head = queuesize
    else head = head - 1
    if head ≠ tail then Dequeue[head] = x
    else error("Dequeue overflow")
end

RemoveatTail(Dequeue)
begin
    if tail = head then error("Queue underflow")
    x = Dequeue[tail]
    if tail = 1 then tail = queuesize
    else tail = tail - 1
    return(x)
end
```

Binary Trees

ASAQ 5-1

Write in pseudocode form the algorithm for checking duplicates in a list of n elements which is found in an array A.

```
CheckforDuplicates(tree)
begin
   tree = getnode
   info(tree) = A[1]
   Duplicate = false
   j = 2
   while Duplicate = false and j £ n do
      Duplicate = InsertBT(tree,A[j])
      j = j + 1
   if Duplicate = true then print("Duplicates detected")
   else print("Duplicates not detected")
end

InsertBT(tree,x)
begin
   found = false
   ptr = tree
   while ptr ≠ nil do
      parent = ptr
      if info(ptr) < x then ptr = right(ptr)
      else
         ptr = left(ptr)
         if info(ptr) = x then found = tree
   newnode = getnode
   info(newnode) = x
   parent(newnode) = parent
   if info(parent) < x then right(parent) = newnode
   else left(parent) = newnode
   return(found)
end
```

Binary Trees

ASAQ 5-2

1. **The power(x,n) = x^n is defined by:**

Power(x,n) = 1, n = 0
Power(x,n) = x * Power(x,n-1), n ³ 1.

Write a recursive procedure for computing x^n, where n ³ 0.

```
Power(x,n)
begin
    if n = 0 then return(1)
     else return(x*Power(x,n-1))
end
```

2. **Write an improved recursive version of Power(x,n) that works by breaking n down into halves (half of n is n div 2), squaring Power(x,n div 2), and multiplying x again if n is odd. For example, $x^7 = x^3 * x^3 * x$, whereas $x^8 = x^4 * x^4$.**

```
        Power(x,n)
        begin
            if n = 0 then return(1)
            else if odd(n) then
      return(Power(x, n div 2) * Power(x, n div 2) * x)
  else
      return(Power(x, n div 2) * Power(x, n div 2)
        end
```

3. **Write a recursive algorithm for reversing a linear linked list.**

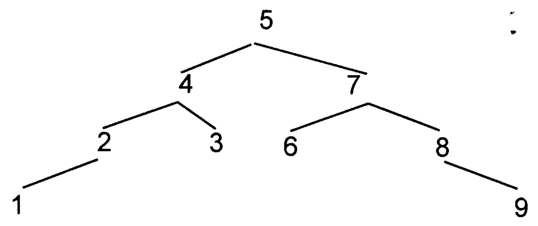
```
Reverse(list)
begin
    if next(list) ¹ nil then Reverse(next(list))
    InsertatEnd(revlist,list)
end
```

Binary Search Trees and AVL Trees

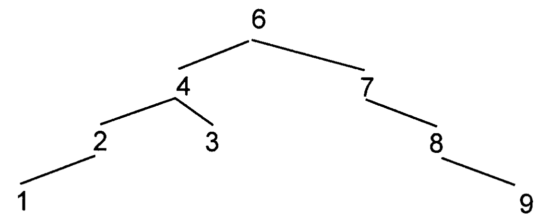
ASAQ 6-1

1. **Show the result of inserting 5, 4, 2, 1, 7, 3, 8, 6, 9 into an initially empty binary search tree. Show the result after deleting the root.**

   Inserting the elements repeatedly produces the BST



   After deleting the root, we produce the following BST



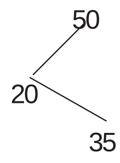
2. **Define a reverse binary search tree as a tree where the values of nodes at the left subtree of node is greater than or equal to the value of the node and the values of nodes at the right subtree is less than the node. Write a procedure for converting a binary search tree into a reverse binary search tree.**

```
ReverseBST(tree)
begin
   if tree ≠ nil
      ptr = left(tree)
      left(tree) = right(tree)
      right(tree) = ptr
      ReverseBST(left(tree))
      ReverseBST(right(tree))
end
```

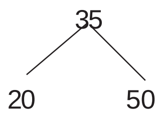
Binary Search Trees and AVL Tress

ASAQ 6-1

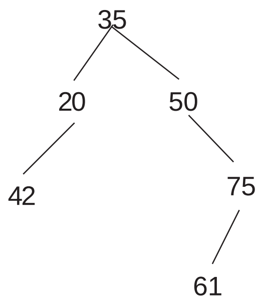
1. **Construct an AVL tree by repeated insertion from the following keys: 50, 20, 35, 75, 42, 61, 10, 5, 45, 12.**



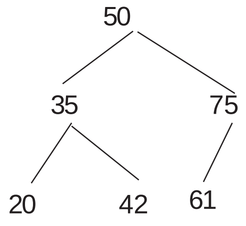
Insert 50, 20, 35



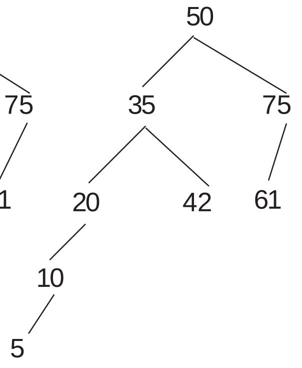
Double right rotate



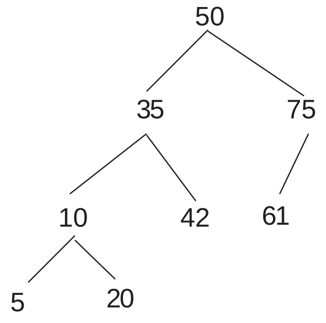
Insert 75, 42, 61



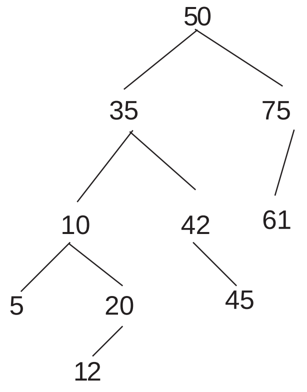
Single left rotate



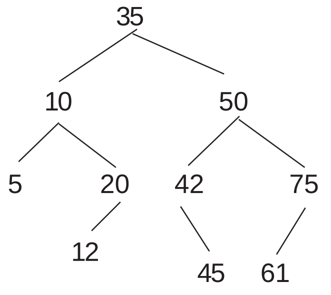
Insert 10, 5



Single right rotate



Insert 45,12



Single right rotate

2. **Write the procedures for the following: Single-Left-Rotate, Single-Right-Rotate, Double-Leaf-Rotate, and Double-Right-Rotate.**

```
Single-Right-Rotate(pivotnd)
begin
   b = pivotnd
   a = left(pivotnd)
   left(b) = right(a)
   right(a) = b
   right(parent(pivotnd)) = a /* if pivot node is a right child    */
end                           /*  change right to left otherwise */

Single-Left-Rotate(pivotnd)
begin
   a = pivotnd
   b = right(pivotnd)
   right(a) = left(b)
   left(b) = a
   right(parent(pivotnd)) = b /* if pivot node is a right child    */
end                           /*  change right to left otherwise */

Double-Right-Rotate(pivotnd)
begin
   c = pivotnd
   a = left(c)
   b = right(a)
   right(a) = left(b)
   left(c) = right(b)
   left(b) = a
   right(b) = c
   right(parent(pivotnd)) = b /* if pivot node is a right child    */
end                           /*  change right to left otherwise */

Double-Left-Rotate(pivotnd)
begin
   a = pivotnd
   c = right(a)
   b = left(c)
   right(a) = left(b)
   left(c) = right(b)
   left(b) = a
   right(b) = c
   right(parent(pivotnd)) = b /* if pivot node is a right child    */
end                           /*  change right to left otherwise */
```

Heaps

ASAQ 7-1

1. **Formulate an algorithm for locating the kth largest element in a heap with n elements.**

```
KthLargest(A,n,k)
begin
    for j = n downto n-k+1 do
        swap(A[1],a[j])
        Heapify(A,1,j-1)
    return(A[n-k+1])
end
```

2. **Extend or modify the kth largest algorithm to produce an algorithm that produces a sorted sequence of the array.**

```
Sort(A,n)
begin
    for j = n downto 2 do
        swap(A[1],a[j])
        Heapify(A,1,j-1)
end
```

Hashing

ASAQ 8-1

1. **Professor *X* hypothesizes that substantial performance gains can be obtained if we modify the chaining scheme so that each list is kept in sorted order. How does the professor's modification affect the running time for successful searches, unsuccessful searches, insertions, and deletions?**

   In terms of worst case, there is actually no difference. For all the operations mentioned the worst case is similar for chaining where the elements are sorted and when the elements are not sorted.

2. **Consider inserting the keys 10,22,31,4,15,28,17,88,59 into a table of length 11 using open addressing with the primary hash function h′(k) = k mod m. Illustrate the result of inserting these keys using**

   a. linear probing;
   b. quadratic probing with $c_1 = 1$ and $c_2 = 3$;
   c. double hashing with $h_2(k) = 1 + (k \bmod (m-1))$.

slot	key
0	22
1	88
2	
3	
4	4
5	15
6	28
7	17
8	59
9	31
10	10

Linear Probing

slot	key
0	22
1	59
2	88
3	17
4	4
5	
6	28
7	
8	15
9	31
10	10

Quadratic Probing

slot	key
0	22
1	
2	59
3	17
4	4
5	15
6	28
7	88
8	
9	31
10	10

Double Hasting

3. **Write the pseudocode for Hash-Delete incorporating the method of using the special value *DELETED* instead of completely deleting the element. Modify the appropriate operations affected by the use of this special value *DELETED*.**

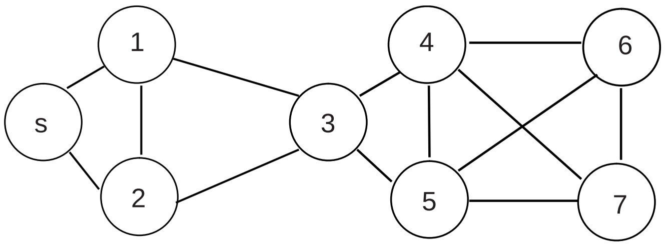
```
Hash-Search(T,k)
begin
   i = 0
   repeat
      j = h(k,i)
      if T[j] = k then return(j)
      i = i + 1
   until T[j] = nil or i = m
   return(nil)
end

Hash-Delete(T,k)
begin
   j = Hash-Search(T,k)
   if j ≠ nil then T[j] = DELETED
end
```

```
        Hash-Insert(T,k)
        begin
            i = 0
            repeat
                j = h(k,i)
                 if T[j] = nil or T[j] = DELETED then
        T[j] = k
   return(j)
     else i = i + 1
 until I = m
  error("Hash table overflow")
        end
```

Data Structures for Graphs

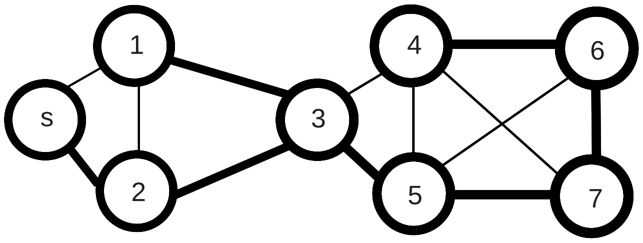
ASAQ 9-1

1. **Given the graph,**

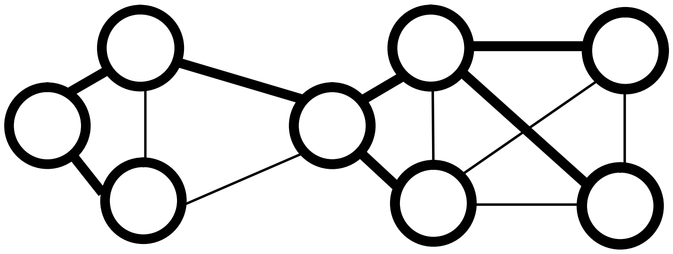


Find the (a) DFS tree and the (b) BFS tree of the graph.

We assume that the algorithm pushes or inserts the lower numbered vertex in cases where two or more vertices are to be pushed into the data structure.



DFS Tree



BFS Tree

Answers to Unit 1 Sample Examination

1. d
2. b
3. c
4. a
5. b
6. b
7. b
8. c
9. a
10. a
11. c
12. b
13. d
14. c
15. c
16. c
17. b
18. d
19. d
20. b
21. b
22. a
23. e
24. a
25. b
26. b
27. c
28. b
29. c,d
30. a,b,c,d

Grading Scale

26-30	Excellent
23-25	Very Satisfactory
21-24	Satisfactory
18-20	Pass
< 18	Fail

How well did you do? Check with the grading scale above on where your score belongs. If you failed, do not worry. Practice as they say makes perfect. Go through the lesson again. Try to give the correct answers to the other numbers you made mistake on the first try.

Introduction to Analysis of Algorithms

SAQ 10-1

For 1-3, show that the following statements are true.

1. **12 is O(1) and 12 is Ω(1)**

   From the definition of O-notation,

   $12 \leq c\,(1)$
   $12 \leq c$

   This inequality holds for any value of $n \geq 0$ by choosing $c = 12$. Hence, with $c = 12$ and $n_0 = 1$ we can verify that $12 \in O(1)$.

   From the definition of Ω-notation,

   $12 \geq c(1)$
   $12 \geq c$

   This inequality holds for any value of $n \geq 0$ by choosing $c = 12$. Hence, with $c = 12$ and $n_0 = 1$ we can verify that $12 \in \Omega(1)$.

   12 is O(1) and 12 is Ω(1) implies 12 is θ(1).

2. **$n^2/2 + n + 2$ is $O(n^2)$ and $n^2/2 + n + 2$ is $\Omega(n^2)$**

   From the definition of O-notation,

   $n^2/2 + n + 2 \leq c\,n^2$
   $1/2 + 1/n + 2/n^2 \leq c$

   This inequality holds for any value of $n \geq 4$ by choosing $c = 1$. Hence, with $c = 1$ and $n_0 = 4$ we can verify that $n^2/2 + n + 2 \in O(n^2)$. Actually, any value ≥ 1 can be the choice for c.

From the definition of Ω-notation,

$$n^2/2 + n + 2 \;\geq\; c\,n^2$$
$$1/2 + 1/n + 2/n^2 \;\geq\; c$$

This inequality holds when $n = 2$ by choosing $c = 1$. Hence, with $c = 1$ and $n_0 = 2$ we can verify that $n^2/2 + n + 2 \in \Omega(n^2)$.

$n^2/2 + n + 2$ is $O(n^2)$ and $n^2/2 + n + 2$ is $\Omega(n^2)$ implies $n^2/2 + n + 2$ is $\theta(n^2)$.

3. **max(n^2, n^3+n^2+n) is O(n^3) and max(n^2, n^3+n^2+n) is Ω(n^3)**

From the definition of O-notation,

$$\max(n^2, n^3+n^2+n) \leq c\,n^2$$
$$\max(1/n, 1+1/n+1/n^2) \leq c$$

This inequality holds for any value of $n \geq 1$ by choosing $c = 3$. Hence, with $c = 3$ and $n_0 = 1$ we can verify that $\max(n^2, n^3+n^2+n)$ is $O(n^3)$. Actually, any value ≥ 3 can be the choice for c.

From the definition of Ω-notation,

$$\max(n^2, n^3+n^2+n)\ {}^{3}\ c\,n^2$$
$$\max(1/n, 1+1/n+1/n^2)\ {}^{3}\ c$$

This inequality holds for $n = 1$ by choosing $c = 1$. Hence, with $c = 1$ and $n_0 = 1$ we can verify that $\max(n^2, n^3+n^2+n)$ is $\Omega(n^3)$.

$\max(n^2, n^3+n^2+n)$ is $O(n^3)$ and $\max(n^2, n^3+n^2+n)$ is $\Omega(n^3)$ implies $\max(n^2, n^3+n^2+n)$ is $\theta(n^3)$

4. **What is the average-case complexity of (a) non-recursive selection sort, (b) recursive selection sort and (c) FindMax algorithm.**

The average-case complexities of (a-c) is similar to their worst-case complexities since whatever the arrangement of the input in the case of (a-b) and wherever the maximum is found in the case of (c), the running time is equivalent to the worst-case running time.

5. **Analyze the two searching algorithm below. Compare the two results.**

```
BinarySearch(A,x)
begin
    lower = 1;
    upper = n;
    repeat
        middle = (lower + upper) div 2;
        if x > A[middle]
    then lower = middle + 1;
    else upper = middle - 1;
    until (A[middle]=x) or (lower > upper);
    return(A[middle] = x);
end
```

The running time for this is $T(n) = \sum_{i=1}^{\log n} O(1) = O(\log n)$.

```
BinarySearch(A,x, lower, upper)
begin
    middle = (lower + upper) div 2
    if x > A[middle]
        BinarySearch(A,middle + 1, upper)
    elseif x < A[middle]
        BinarySearch(A,lower,middle-1)
    else
        found = true;
end
```

The running time for this is $T(n) = O(1) + T(n/2) = O(\log n)$.

Simplifying Summations

ASAQ 11-1

Find the closed form expression of the following sums:

1. $\sum_{k=1}^{n} (2k - 1) = 2 \sum_{k=1}^{n} k - \sum_{k=1}^{n} 1$
   $= 2 (n(n+1)/2) - n$
   $= n^2$

2. $\sum_{k=1}^{n} n/2^k = n \sum_{k=1}^{n} 1/2^k$
   $= n (1 - 1/2^n)$

3. $\sum_{i=0}^{\infty} i/4^i = \sum_{i=0}^{\infty} i(1/4)^i$
   $= (1/4) / (1-1/4)^2$
   $= (1/4) / (9/16)$
   $= 4/9$

Simplifying Recurrences

ASAQ 12-1

Simplify the recurrence relation $T(n) = 3T(n/2) + n$, where $T(1) = 1$.

1. **Using the substitution method:**

   Suppose we assume that $T(n) = O(n^2)$. By this assumption, we show that $T(n) \leq cn^2$ for an appropriate choice of $c > 0$.

   Base case: Let $n = 2$.

   $$\begin{aligned} T(2) = 3T(2/2) + 2 &\leq c\,(2^2) \\ 3 + 2 &\leq 4c \\ 5/4 &\leq c \end{aligned}$$

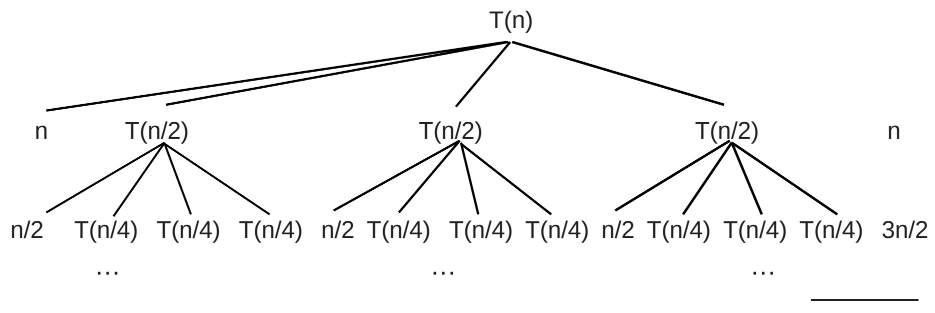
With $c = 5/4$ and $n = 2$, $T(n) \le cn^2$.
Inductive step: We assume that $T(n/2) \le c(n/2)^2$. Substituting this on $T(n)$,

$$\begin{aligned} T(n) &= 3(c(n/2)^2) + n \\ &= (3/4)\, cn^2 + n \\ &< cn^2 \end{aligned}$$

provided $c \ge 1$. We select $c = 5/4$ to be consistent with the case $n = 2$.

Hence, $T(n) \le cn^2$ which implies $T(n) = O(n^2)$.

2. **Using recursion tree:**



$$\sum_{i=0}^{\log n - 1} (3/2)^i\, n$$

$$\begin{aligned} \sum_{i=0}^{\log n - 1} (3/2)^i\, n &= n \sum_{i=0}^{\log n - 1} (3/2)^i \\ &= n\, ((3/2)^{\log n} - 1)\, /(3/2 - 1) \\ &= 2n\, (3/2)^{\log n} - 2n \\ &= 2(3^{\log n}) - 2n \\ &= 2\, n^{\log 3} - 2n \\ &= O(n^{\log 3}) \end{aligned}$$

3. **Using the Theorem on T(n) = aT(n/b) + f(n):**

$a = 3$
$b = 2$
$f(n) = n$, which means $k = 1$

This is a case where $a > b^k$, i.e., $3 > 2^1$. Hence, $T(n) = O(n^{\log_a b}) = O(n^{\log 3})$.

Analysis of Sorting Algorithms

ASAQ 13-1

What is the running time for the input sequence 1, 2, 3, ..., n (already sorted) of the sorting algorithms BubbleSort, InsertionSort, QuickSort, MergeSort, HeapSort.

1. **BubbleSort**

   On a sorted input, BubbleSort will have a cost similar to the worst case except for the following:

   Line 5 as in the worst-case analysis will be executed regardless of the arrangement of the input. However, Line 6 will never be executed on an input that is already sorted.

```
1   BubbleSort(n);
2   begin
3      for i = 1 to n-1 do                    Cost: $\sum_{i=1}^{n-1}$ (n-i)
4         for j = n downto i+1 do             Cost: (n-i)
5            if A[j] < A[j-1] then            Cost: 1
6               swap(A[j], A [j-1]);          Cost: 0
7   end;
```

$$
\begin{aligned}
\text{Cost} &= \sum_{i=1}^{n-1} (n-i) \\
&= \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i \\
&= n(n-1) - (n-1)n/2 \\
&= n^2 - n - (n^2 + n)/2 \\
&= n^2/2 - n/2 \\
&= O(n^2)
\end{aligned}
$$

2. **InsertionSort**

```
1    InsertionSort(n);
2    begin
3       for i= 2 to n do
4          begin
5             tobs = A[i];                          Cost: 1
6             j= i - 1;                             Cost: 1
7             while (tobs < A[j]) and (j > 0) do    Cost: 1
8             begin
9                A[j+1] = A[j];                     Cost: 0
10               j= j - 1;                          Cost: 0
11            end;
12            A[j+1] = tobs;                        Cost: 1
13         end;
14   end;
```

On a sorted input, Line 7 will be executed once and those inside the loop (Lines 9-10) will never be executed. Hence, we have a total cost of:

$$\begin{aligned} \text{Cost} &= \sum_{i=2}^{n} 4 \\ &= 4(n-1) \\ &= 4n - 4 \\ &= O(n) \end{aligned}$$

3. **Quicksort**

On a sorted input, the partition step on n elements will produce two partitions one of size n-1 and the other of size 0. The pivot element being positioned at the leftmost part of the sequence being partition. Hence, the cost on an input that is already sorted is:

$$\begin{aligned} T(n) &= T(n-1) + O(n) \\ &= O(n^2) \end{aligned}$$

4. **MergeSort**

   On a sorted input, the cost of MergeSort will still be dictated by the cost of the Merge algorithm. Since the transfer elements from array B to A will still be done, Merge will cost O(n). Hence, the total cost is:

$$\begin{aligned} T(n) &= 2T(n/2) + O(n) \\ &= O(n \log n) \end{aligned}$$

   In fact, regardless of the arrangement of the input MergeSort will cost O(n log n).

5. **HeapSort**

   As with MegreSort, HeapSort will have the same cost regardless of the arrangement of the input. The reason is Heapify will be executed n-1 times and everytime it is executed, its cost is equal to the height of the heap being considered. Hence, a cost of O(n log n) for an input that is already sorted.

Answers to Unit 2 Sample Examination

1. c
2. c
3. b
4. c
5. d
6. a
7. c
8. c
9. c
10. d
11. b
12. d
13. b
14. a
15. c
16. c
17. c
18. e
19. d
20. c
21. d
22. d
23. c
24. a
25. a
26. b
27. c
28. d
29. b
30. c
31. a
32. a
33. a
34. c
35. c

Grading Scale

30-35	Excellent
27-29	Very Satisfactory
24-26	Satisfactory
21-23	Pass
< 21	Fail

Again, do not worry if you fail this sample exam. What is important is you go through the lessons again until you can correctly answer the items where you made mistakes.

Appendix 2
Supplementary Self-Assessment Questions

WARNING: The questions given here are not provided with answers. They are provided for you to try if you have completely answered the SAQ's and would like some more practice. It is not absolutely necessary for you to answer these questions.

Multidimensional Arrays

1. A 10x10 array of 0's and 1's represents a maze in which a traveler must find a path from maze[1,1] to maze[10,10]. The traveler may move from a square into any adjacent square in the same row or column, but may not skip over any squares or move diagonally. In addition, the traveler may not move into any square that contains a 1. The positions maze[1,1] and maze[10,10] contains 0's. Write a routine that accepts such a maze and either prints a message that no path through the maze exists or a path exists.

Linear Linked Lists

1. Let L1 and L2 be two linear linked lists where the elements are sorted in ascending order from the head to the tail. Formulate an algorithm for merging L1 and L2 to form L with the elements also sorted. Generalize the procedure so it merges n sorted linked lists into one.

2. Formulate an algorithm for rearranging the elements of a linear linked list L1 to produce the list L2 where in L2 all nodes whose values are odd precede those nodes whose values are even.

3. Write a procedure for producing one sorted linked list L from out of one sorted linked list L1 and one unsorted linked list L2.

4. A self-adjusting list is like a regular list, except that all insertions are performed at the front, and when an element is accessed by find, it is moved to the front of the list without changing the relative order of the other items. Formulate the procedure Self-Adjusting-Insert(v) and Self-Adjusting-Find(x).

Circular Linked Lists

1. The Josephus problem is the following mass suicide game: n people, numbered 1 to n, are sitting in a circle. Starting with person 1, a handgun is passed. After m passes, the person holding the gun commits suicide, the body is removed, the circle closes ranks, and the game continues with the person who was sitting after the corpse. The last survivor is tried for n-1 counts of manslaughter. Thus, if m = 0 and n = 5, players are killed in order and player 5 stands trial. If m = 1 and n = 5, the order of death is 2, 4, 1, 5. Write a procedure to solve the Josephus problem for general values of m and n.

Doubly Linked Lists

1. Given a doubly linked list L, divide said list into two: L1 containing elements with values less than x and L2 containing elements with values greater than or equal to x.

2. Formulate an algorithm for checking if one doubly linked list L1 is a sublist of list L2.

Stacks

1. Given a mathematical expression which may contain three different types of parentheses, e.g., {{(a+b)*c}-[[4+d]]+2}. Formulate an algorithm that checks if the parentheses are balanced.

2. Suppose each element is stamped with the time when it is pushed into the stack. Merge two stacks S1 and S2 with this kind of elements into one stack S with the elements still stamped with the time it is pushed.

Queues

1. Arithmetic expressions are usually written in infix notation, e.g., 1 + 2 and a * (b + c) - d. Postfix notation is another form in which one can write expressions. In postfix, each operator occurs immediately to the right of its operands. For example, the two expression given earlier are written in postfix as 1 2 +, a b c + * d -. An advantage of postfix notation is that parentheses are not necessary. Postfix notations can be evaluated using a stack. Write a procedure that uses a stack to evaluate postfix expressions. Before evaluating an expression, however, read the postfix expression into a queue where each element of the queue is either an integer or an operator. The operators are just (+, -, *, /). Use the stack and queue operations given earlier.

Binary Trees

1. Write the procedure for converting an infix expression into an expression tree.

2. Formulate the nonrecursive procedures for doing a preorder and postorder traversal of trees.

3. Show that the maximum number of nodes in a binary tree of height h is 2^{h+1}-1.

4. A full node is a node with two children. Prove that the number of full nodes plus 1 is equal to the number of leaves in a binary tree.

5. The level-order listing of the nodes of a binary tree first lists the root, then all nodes of depth 1, then all nodes of depth 2, and so on. Nodes at the same depth are listed in left-to-right order. Write a procedure to list the nodes of a tree in level order.

Binary Search Trees

1. Draw binary search trees of height 2, 3, 4, 5, 6 using the keys {1, 5, 10, 14, 19, 23, 27}.

2. Formulate an inorder tree traversal algorithm that uses the procedures BST-Minimum and BST-Successor.

3. Formulate a procedure for finding the kth smallest element in a binary search tree.

4. Write a procedure for extracting the minimum from a binary search tree.

AVL Trees

1. Let T be an AVL tree of height h. What is the maximum number of nodes in T? What is the minimum number of nodes in T?

2. Formulate the procedure for inserting and deleting an element into an AVL tree.

Heaps

1. Formulate a procedure for building a heap by repeated insertion.

2. A d-heap is a heap where each node has at most d children. If a d-heap is stored in an array, for an entry located in position i, where are the parents and children.?

Hashing

1. Consider a dynamic set *S* that is represented by a direct-address table *T* of length *m*. Describe a procedure that finds the maximum element of *S*.

2. Consider a version of the division method in which $h(k) = k \bmod m$, where $m = 2^p - 1$ and k is a character string interpreted in radix 2^p. Show that if string *x* can be derived from string *y* by permuting its characters, then *x* and *y* hash to the same value.

3. Consider a hash table of size m = 1000 and the hash function h(k) = $\lfloor$m (kA mod 1)$\rfloor$ for A = (sqrt(5)-1)/2. Compute the locations to which the keys 61,62,63,64, and 65 are mapped.

Data Structures for Graphs

1. Formulate a procedure for checking if a cycle exists in a graph where the representation used is the (a) adjacency lists and (b) adjacency matrix.

2. Formulate a procedure to enumerate all simple cycles of a graph in a (a) adjacency lists and (b) adjacency matrix representation.

3. Formulate an algorithm to insert and delete edges in the adjacency list representation of a graph.

Simplifying Recurrences

1. $\prod_{k=1}^{n} 2(4^k)$

2. Prove $\sum_{i=1}^{n} (2i-1) = n^2$

3. Prove $\sum_{i=1}^{n} i^3 = (\sum_{i=1}^{n} i)^2$

4. Simplify the following recurrence relations:

   (a) $T(n) = 3T(n/2) + n^2$ (c) $T(n) = 2T(n/2) + \log n$
   (b) $T(n) = T(n) + n$ (d) $T(n) = 2T(n-1) + n$

5. Prove or disprove the following:
   (a) $2n + 5 \in O(n^2)$ (d) $5n^3 + n^2 \in O(n^2)$
   (b) $f(n) + g(n) \in O(\max(f(n),g(n)))$ (e) $2n + 5 \in \Omega(n^2)$
   (c) $5n^3 + n^2 \in \Omega(n^2)$

6. Prove or disprove the following:
   (a) f(n) = O(g(n)) implies g(n) = O(f(n))
   (c) f(n) + g(n) = θmin(f(n),g(n)))
   (d) f(n) = O(g(n)) implies g(n) = W(f(n))

7. Suppose S(n) = O(f(n)) and T(n) = O(f(n)). Prove or disprove the following:

   (a) S(n) + T(n) = O(f(n)) (b) S(n) / T(n) = O(1)

8. Find two functions f(n) and g(n) such that neither f(n) = O(g(n)) nor g(n) = O(f(n)).

Analysis of Sorting Algorithms

1. A sorting algorithm is stable if it preserves the original order of records with equal keys. Which of the eight sorting algorithms are stable?

2. What is the running time of the eight sorting algorithms when the keys in the input are all equal.

3. Formulate an algorithm for finding the kth smallest element in a set of n elements? What is the cost of your algorithm? Is it possible to solve the problem in O(n) time? Support your answer.