Chapter 4

Abstract data types II: trees, graphs and heaps

4.1 Essential reading

Duane A. Bailey *Java Structures: Data Structures in Java for the principled programmer*. (McGraw-Hill Companies, Inc. 1999, McGraw-Hill International editions) [ISBN 0-13-489428-6]. Chapter 10, 11

Michael T. Goodrich and Roberto Tamassia, *Data Structures and Algorithms in Java*. (John Wiley & Sons, Inc., 2001, fourth edition) [ISBN 0-201-35744-5]. Chapter 7, 8, 13

Anany Levintin *Introduction to the design and analysis of algorithm*. (Addison-Wesley Professional, 2003) [ISBN 0-201-743957]. Chapter 6.4

Michael Main, *Data Structures and Other Projects Using Java*. (Addison Wesley Longman Inc., 1999) [ISBN 0-201-35744-5]. Chapter 9

Russell L Shachelford *Introduction to Computing and Algorithms*. (Addison Wesley Longman, Inc., 1998) [ISBN 0-201-31451-7]. Chapter 5

Mark Allen Weiss *Data Structures and Problem Solving Using Java*. (Addison Wesley Longman Inc., 1998) [ISBN 0-201-54991-3]. Chapter 6, 17, 18, 20

4.2 Learning outcomes

This chapter introduces one of the most important abstract data structures which is called a *binary tree*.

Having read this chapter and consulted the relevant material you should be able to:

- explain the importance of binary trees, graphs and heaps
- implement the binary tree data structure in Java or other languages
- describe some applications of binary trees.
- demonstrate how to represent a graph in computers
- describe the two main algorithms of graph traversal
- outline the algorithms for solving some classical graph problems.

4.3 Trees

A tree (also called a free tree) is defined as a set of vertices (also called nodes) connected by their edges so that there is exactly one way to traverse from any vertex to any other vertex.

Trees are a very important and widely used abstract data structure in computer science. Recall *arrays, linked lists, stacks and queues*. Each of these data structures represents a *relationship* between data. A *tree* represents a hierarchical relationship. Each node in a tree spawns one or more branches that each leads to the top node of a *subtree*. Almost all operating systems store sets of files in trees or tree-like structures.

Example 4.1 *The file system in a user account (Figure 4.1).*

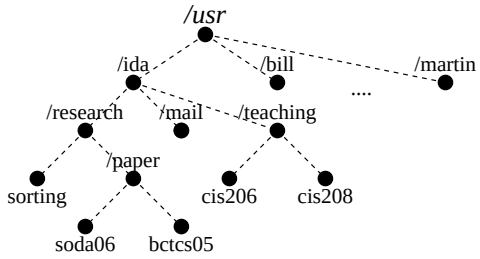


Figure 4.1: A tree structure in a file system

Trees are very rich in properties. We shall find out later that there are actually many ways to define a tree.

4.3.1 Terms and concepts

To ease the discussion, we first define some terms and concepts about trees and look at an example in Figure 4.1.

root The top most node is called the '*root*'. For example, '`usr`' is the root of the (free) tree.

leaf A node that has no children is called a *leaf*. For example, '`soda06`' is a leaf.

parent The predecessor node that every node has (except the root). For example, '`ida`' is the parent of '`research`' and '`research`' is the parent of '`sorting`'.

child A successor node that each node has (except leaves). For example, '`sorting`' is a child of '`research`'.

siblings Successor nodes that share a common parent. For example, '`sorting`' and '`paper`' are siblings.

subtrees A subtree is a substructure of a tree. Each node in a tree may be thought of as the root of a *subtree*. For example, '`paper`' is the root of a three-nodes subtree consisting of '`paper`', '`soda06`' and '`bctcs05`'.

degree of a tree node The number of children (or subtrees) of a node. For example, the node '`ida`' has a degree of 3.

degree of the tree The maximum of the degrees of the nodes of the tree, which is 3 in the example.

ancestors of a node All the nodes along the path from the root to that node. For example, the ancestors of node '`research`' are '`usr`' and '`ida`' (or '`usr/ida`').

path from node n_1 to n_k A sequence of nodes from n_1 to n_k. The *length* of the path is the number of edges on the path. For example, the length from '`research`' to '`cis208`' is 3.

depth of a node The length of the unique path from the root to the node. For example, the depth of node '`cis208`' is 3.

In addition,

forest A collection of trees is called a *forest*.

binary trees The trees in which every node has at most two subtrees, although either or both subtrees could be empty.

Example 4.2 *An arithmetic expression: $A * B + C$ can be represented by a binary tree.*

Recall we represented an arithmetic expression by a list to emphasise the hierarchy of operators, i.e. $*$, $/$ prior to $+$, $-$. An arithmetic expression tree more naturally shows the hierarchy (Figure 4.2).

An expression tree is a binary tree. The operators, such as $*$, $/$, $+$, $-$, are stored on the internal nodes and the operands, such as A, B, C, are on the leaves. The value of the subtree rooted at an internal node can be derived by applying the operator at the node to the operands at its children recursively.

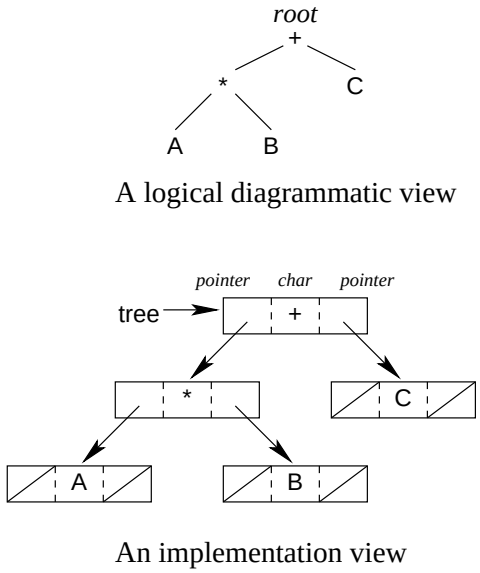


Figure 4.2: An expression tree

For example, consider the node '$*$' in Figure 4.2. Let $A = 2$, $B = 3$. The value of the subtree at '$*$' is $2 \times 3 = 6$.

4.3.2 Implementation of a binary tree

There are actually many kinds of trees. We are only concerned with *rooted, labelled* and *binary trees* here.

First, we need to define a tree node structure. A simple way to achieve this is to use a similar approach to the node definition for linked lists in section 2.7.2.

Example 4.3 *We need to define three fields, namely* `left, da, right`*, where* `left` *and* `right` *are the links to the left child and right child respectively, and* `da` *is the data field. In this way, similar to a linked list represented by its head, a tree can be referenced by its root.*

```
import java.io.*;

// A class of treeNode which allows us to construct
// a tree node

// This defines a 'simple tree node' (type).
public class treeNode {
  private Object da;
  private treeNode left;
  private treeNode right;

// This defines a new tree node.
  public treeNode(Object newItem) {
    da = newItem;
    left = null;
    right = null;
  } // Constructor

// This is to create a 'normal' node.
  public treeNode(Object newItem, treeNode leftNode,
    treeNode rightNode) {
      da = newItem;
      left = leftNode;
      right = rightNode;
  }

// This is to update the da field of a node.
  public void setItem(Object newItem) {
    da = newItem;
  }

// This is to read the item field of a node.
 public Object getItem() {
    return da;
  }

// This is to update the left field of a node.
 public void setLeft(treeNode leftNode) {
    left = leftNode;
  }

// This is to read the left field of a node
 public treeNode getLeft() {
    return left;
  }

// This is to update the right field of a node.
 public void setRight(treeNode rightNode) {
    right = rightNode;
```

```
  }

// This is to read the right field of a node
 public treeNode getRight() {
    return right;
  }

}
```

With the tree structure, it is easy to access a tree node.

Example 4.4 *In this example, we first display the data field value of the root, then move to the left child of the root and display the data field value, finally move to the right child of the left child of the root, and display the value.*

1: *print*($tree.da$)
2: $tree \leftarrow tree.left$
3: *print*($tree.da$)
4: $tree \leftarrow tree.right$
5: *print*($tree.da$)

Let us look at an arithmetic expression tree, for $(a + b)^2/(a - b)$, in Figure 4.3.

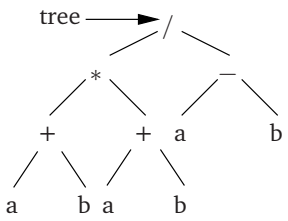


Figure 4.3: Another expression tree

The expression $(a + b)^2/(a - b)$ can be thought of as the operator '/' applies to two sub-expressions $(a + b)^2$ and $(a - b)$. This corresponds to the fact that the root of the expression tree has two subtrees, the left subtree and the right with their roots '$*$' and '$-$' respectively. If we move our view from the root '/' to '$*$', we see a similar picture where '$*$' applies to two subtrees $(a + b)$ and $(a + b)$ respectively. This actually applies to every non-leaf node of a tree. In fact, the most natural and easy way to define a tree is to define the tree *recursively*.

4.3.3 Recursive definition of Trees

A tree is a collection of nodes. The collection can be *empty*. Otherwise, a tree consists of a distinguished node r (for *root*) and zero or more subtrees each of whose roots are connected by a directed edge from r.

Each (non-empty) subtree is called a *child* of r, and r is the *parent* of each subtree of r. Nodes with no children are called *leaves*. Nodes with the same parent are called *siblings*.

A *binary tree* (see Figure 4.4) is either empty, or it consists of a node called the *root* together with two binary trees called the *left subtree* and the *right subtree* of the root.

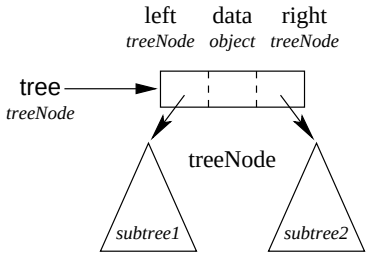


Figure 4.4: Recursive definition

Recall that many problems can be solved by including a subtask that calls itself as part of the solution. A program that calls itself at some stage is called a *recursive program*. An algorithm is called *recursive* if it contains a statement that calls itself.

Trees are recursive in nature and it is natural to write tree algorithms (programs) that use recursion. A recursive algorithm can be simpler and more elegant than a non-recursive equivalent, but one needs to be careful to make sure to implement recursion correctly. The reader is encouraged to undertake some revision of recursive algorithms (Chapter 3) before moving on to the next section.

4.3.4 Basic operations on binary trees

1. `initialise()` - create an empty binary tree.
2. `empty()` - *true* if the binary tree is empty, *false* otherwise.
3. `create(x)` - create a one-node binary tree `T`, where `T.da=x`.
4. `combine(l,r)` - create a tree `T` whose left subtree is `l` and right subtree is `r`.
5. `traverse()` - traverse every node of a tree `T` (see Section 4.3.5).

Algorithm 4.1 object initialise()

1: return null

Algorithm 4.2 boolean empty(treeNode t)

1: return ($t = null$)

Algorithm 4.3 treeNode combine(l,r)

1: $t \leftarrow null$
2: $setLeft(l)$, $setRight(r)$
3: return t

4.3.5 Traversal of a binary tree

Traversal of a data structure means visiting each node exactly once. This is more interesting in trees than in lists because trees offer no natural linear sequence to follow.

There are three particularly important traversals of a binary tree, namely *preorder, inorder* and *postorder traversal*. A binary tree may be empty, in which case there is no node to visit. Otherwise, the tree consists of a root node, left subtree and right subtree which must be visited. The only difference between the three traversals is the order of the steps: `preorder` visits the root first, `postorder` visits it last and `inorder` visits it in between the two subtree traversals. The left subtree is always visited before the right.

The recursive algorithms for the three traversals are as follows (Algorithms 4.4– 4.6):

Algorithm 4.4 preorder(treeNode T)

```
1: if not empty(T) then
2:     print(T.da);
3:     preorder(T.left);
4:     preorder(T.right)
5: end if
```

Algorithm 4.5 inorder(tree T)

```
1: if not empty(T) then
2:     inorder(T.left);
3:     print(T.da);
4:     inorder(T.right)
5: end if
```

Algorithm 4.6 postorder(tree T)

```
1: if not empty(T) then
2:     postorder(T.left);
3:     postorder(T.right);
4:     print(T.da)
5: end if
```

The three traversals, i.e. the *preorder*, *inorder* and *postorder* traversal on an expression tree can result in three forms of arithmetic expression, namely, *prefix*, *infix* and *postfix* form respectively.

Example 4.5 *A*B+C* *(See Figure 4.2)*

Traversal	*Nodes visited*	*Arithmetic expression*
postorder:	*AB*C+*	*postfix form*
preorder:	*+*ABC*	*prefix form*
inorder:	*A*B+C*	*infix form*

4.3.6 Construction of an expression tree

As an example, we show how to construct an arithmetic expression tree given the expression in its *postfix* form. The postfix form is an expression where the operator is placed after both operands.

For example, the postfix form of an arithmetic expression $a + b$ is $ab+$, and the postfix form of $(a + b) * c$ is $ab + c*$.

The normal form of an arithmetic expression such as $a + b$ and $(a + b) * c$ is called the *infix* form. Similarly, an expression can also

be written in *prefix* where the operator is placed before operands. For example, the prefix form of the above expressions are $+ab$ and $*+abc$ respectively. Brackets are not needed in the postfix or prefix form and therefore are economical for computer storage. However, they are not easily recognisable by humans.

Suppose the expression is stored in an array and a stack, initially empty, is used to store the partial results.

Algorithm 4.7 treeNode construct()

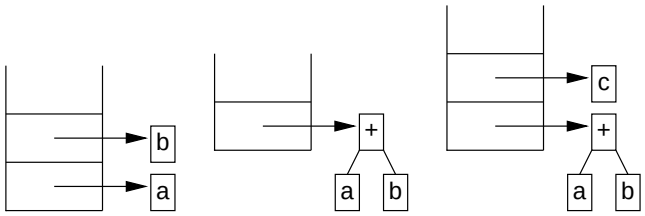
INPUT: An expression in array
RETURN: The root of the expression tree

1: Read the expression one symbol at a time.
2: **if** the symbol is an operand **then**
3: create a one-node tree and push the pointer to it onto a stack.
4: **else if** the symbol is an operator **then**
5: pop pointers to two trees T1 and T2 from the stack and form a new tree whose root is the operator and whose left and right children point to T2 and T1 respectively.
6: A pointer to this new tree is then pushed onto the stack.
7: **end if**
8: return the top of the stack

Example 4.6 *We first write the arithmetic expression* $(a + b) * c * (d + e)$ *in the postfix form (by hand for the moment):* $ab + c * de + *$*, and store it in an array* S.

We use the standard procedures and functions defined earlier for trees (Section 4.3.4), and stacks (Section 2.8.1), where simple variables l, r *point to the left and the right subtree respectively, T is the root of the expression tree to be built.*

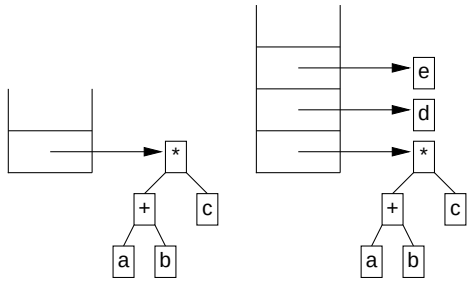
1. *Read a, create('a',T), push(S,T) read b, create('b',T), push(S,T) (Figure 4.5(a))*
2. *Read +, pop(S,r), pop(S,l), combine(l,r) and push(S,T) (Figure 4.5(b))*
3. *Read c, create('c',T), push(S,T) (Figure 4.5(c))*



(a) step 1 (b) step 2 (c) step 3

Figure 4.5: steps 1–3

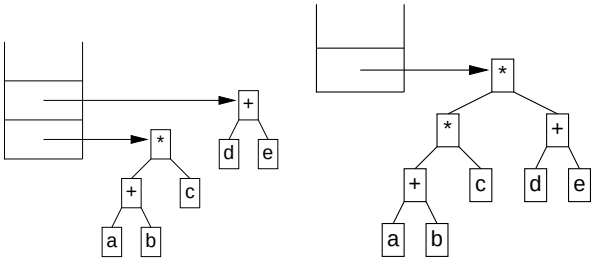
4. *Read *, pop(S,r), pop(S,l), combine(l,r) and push(S,T) (Figure 4.6(a))*
5. *Read d, create('d',T), push(S,T)*
6. *Read e, create('e',T), push(S,T) (Figure 4.6(b))*
7. *Read +, pop(S,r), pop(S,l), combine(l,r) and push(S,T) (Figure 4.7(a))*



(a) step 4 (b) step 5,6

Figure 4.6: steps 4–6

8. *Read *, pop(S,r), pop(S,l), combine(l,r) and push(S,T) (Figure 4.7(b))*



(a) step 7 (b) step 8

Figure 4.7: steps 7–8

Applications

There are too many tree applications to list completely here. We will look at some of them in later chapters. The reader is encouraged to study more examples in the text books.

Activity 4.3

TREES

1. What is the difference between a tree and a binary tree?
2. Draw an expression tree for each of the following expressions:
   (a) 5
   (b) (5+6*4)/2
   (c) (5+6*4)/2-3/7
   (d) 1+9*((5+6*4)/2-3/7)
   (e) $A \times B - (C + D) \times (P/Q)$
3. Hand draw binary expression trees that correspond to the expressions for which

(a) The infix representation is $P/(Q+R)*X-Y$

(b) The postfix representation is $XYZPQR*+/-*$

(c) The prefix representation is $+*-MNP/RS$

4. For each expression tree drawn in the above question, list the sequence of elements encountered in inorder, preorder and postorder traversals.
5. Describe, with an example of 5 nodes, the topological characteristics that distinguish a tree from a linked list.
6. Define a treeNode class for tree nodes that each consists of following 4 fields:

parent	leftChild	rightChild	data

7. Following the above node definition, define and implement a binary tree class with necessary access methods for the 4-field treeNodes, for example, `setXXX`, `getXXX`, `isEmpty`, etc..
8. Write a method that takes two binary trees `t1`, `t2` and a binary tree node `v` as the arguments. It constructs and returns a new binary tree that has `v` as its root and whose left subtree is `t1` and whose right subtree is `t2`. Both `t1` and `t2` should be empty on completion of the execution.

4.4 Priority queues and heaps

A priority queue is a queue with a conditional *dequeue* operation in addition to the FIFO principle. The elements in the queue have certain orderable characteristics which can be used to decide a *priority*.

For example, given a queue of integers, we want to

1. remove the smallest number from the queue
2. add a new integer to the queue according to the pre-defined priority.

The priority means the *smallest* (or biggest) in some comparable value. This task is such a common feature of many algorithms that the generic name of a *Priority Queue* has been given to such a queue.

A priority queue can be realised by a partially ordered data structure called *heap*.

4.4.1 Binary heaps

A binary heap is a partially ordered *complete* binary tree. Similar to a binary search tree, a heap[1] has an *order* property as well as a *structural* property. We first look at the structural property.

[1] In this module, we use 'heap' to mean a binary heap unless stated otherwise.

4.4.1.1 Structural property

The structure of a heap is as a complete binary tree. A complete binary tree is a binary tree in which every level is full except possibly the last (bottom) level where only the rightmost leaves may

be missing. A complete binary tree is referred as a *full* binary tree if all the leaves are at the same level.

Example 4.7 *The binary tree in Figure 4.8 is a complete binary tree, and a full binary tree in which all its leaves are at the same level.*

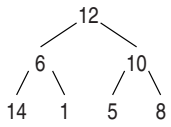


Figure 4.8: A full binary tree

A complete binary tree can be derived from a full binary tree by deleting some right most leaves (and edges leading to them). For example, if we delete the right most leaf '8' in the full binary tree in Figure 4.8. We get another complete binary tree.[2]

[2] Note the term 'complete binary tree' is sometimes also used as a synonym for full tree in some books. We distinguish the two concepts in this module to avoid confusion.

The structural property of the heap makes an array implementation easy. The nodes in the complete binary tree in Figure 4.8 can be stored in an array level by level contiguously, for example:

```
i      1  2  3  4  5  6  7
A[i]   12 6  10 14 1  5  8
```

The advantage of the array implementation is that each node can be accessed in $O(1)$ time. In addition, the parent or each child of a node can be accessed in $O(1)$ time. For example, node A[i]'s parent is: A[i div 2], its left child is A[$2i$] and the right child is A[$2i + 1$].

Example 4.8 *Figure 4.9 is another complete binary tree in which all the levels are full except the last level where the leaves are stored from the left to the right without gap.*

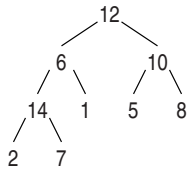


Figure 4.9: A complete binary tree

```
i      1  2  3  4  5  6  7  8  9
A[i]   12 6  10 14 1  5  8  2  7
```

Example 4.9 *Figure 4.10 is a binary tree but not a complete binary tree because the left most node is missing from the last level. An incomplete binary tree leaves gaps in an array structure. The tree structure can, of course, still be stored in an array but dummy elements such as "%" are required.*

```
i      1  2  3  4  5  6  7  8  9
A[i]   12 6  10 14 1  5  %  %  7
```

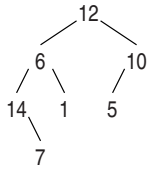


Figure 4.10: A binary tree but not complete

4.4.1.2 The order property

The order property requires, for every node in the structure, the value of both its child nodes must be smaller (or bigger) than the value at the current node.

Example 4.10 *Figure 4.11 shows a heap with the minimum value at the root. The value at each node is smaller than either child.*

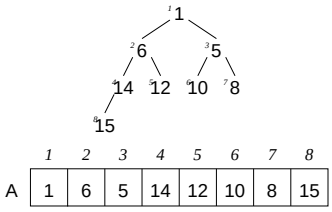


Figure 4.11: A heap with a MinKey root

or

A heap with the maximum value at the root, and the value at each node is smaller than either child (Figure 4.12).

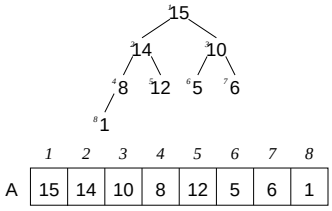


Figure 4.12: A heap with a MaxKey root

Note the complete binary tree in Figure 4.9 is not a heap because it does not have the order property required.

4.4.2 Basic heap operations

In this implementation, the numbers in our set are placed at the nodes of a binary tree in such a way that the numbers stored at the children of any node are smaller (or larger) than the number stored at that node (order property). Moreover, the tree is a left complete binary tree, i.e. a binary tree that is completely filled except possibly

the bottom level which is filled from left to right (structural property).

Typical operations (Binary heap):

buildHeap() construct the initial heap from a list of items (keys) in arbitrary order (Figure 4.16).

deleteMin() remove the smallest element from the root and maintain the heap (Figure 4.13).

deleteMax() remove the largest element from the root and restore the heap (Figure 4.14).

insertOne(x) add one element x to the heap and maintain the heap (Figure 4.15).

4.4.2.1 Deletion

Consider the operation of removing the smallest element from this structure (and reconstituting the tree).

The smallest (or largest) element will always be at the root node.

Example 4.11 *Figure 4.13 shows how the order property is maintained after (a) the smallest element '1' is removed: (b) The root position becomes available. (c)(d) Datum 15 is moved from the leaf to the root. (e) Datum 15 is swapped with its smaller (the right) child 5, and then (e) swapped with its smaller (the left) child 8. (f) Datum 15 is finally settled as a leaf and becomes the left child of node 8.*

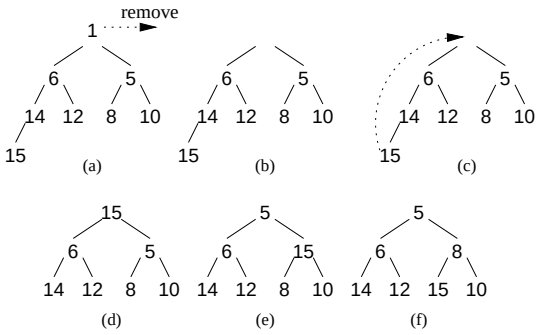


Figure 4.13: DeleteMin

A similar restoring process takes place for a max-heap.

Example 4.12 *Consider the max-heap in Figure 4.12. Figure 4.14 shows how, after the maximum root element 15 is deleted, the order property is restored by an* `insertion` *and two* `swaps` *on the corresponding array (the heap).*

1. *Remove the root element 15, and move element 1, the rightmost leaf at the bottom level to the root. (Figure 4.14(a))*
2. *Restore the order property by repeatedly swapping element 1 with its larger child element 14, e.g. first swap 1 at the root with the*

left child 14, and then swap 1 with its right child 12, until the order property is restored. (Figure 4.14(b)).

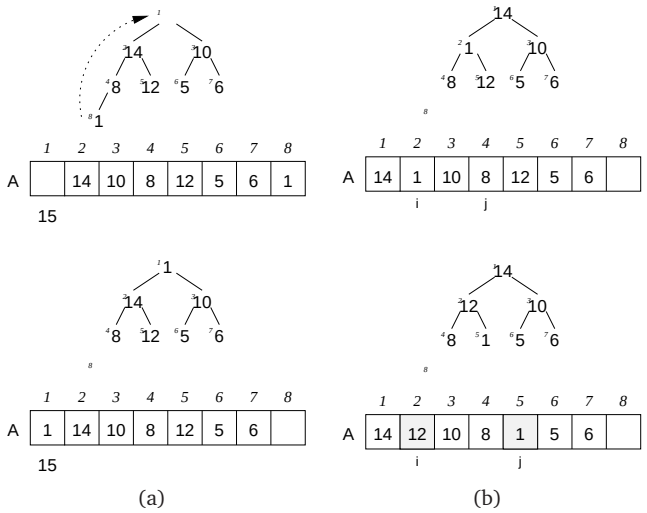


Figure 4.14: addRoot-1,2

This process can be described in Algorithm 4.8 and 4.9.

Algorithm 4.8 addRoot(index i, n)

```
1: j ← 2 * i
2: if j ≤ n then
3:    if j < n and A[j] < A[j + 1] then
4:       j ← j + 1
5:       if A[i] < A[j] then
6:          tmp ← A[i]
7:          A[i] ← A[j]
8:          A[j] ← tmp
9:          addRoot(j, n)
10:      end if
11:   end if
12: end if
```

Algorithm 4.9 buildHeap()

```
1: for k ← (length(A) div 2, k ≤ 1, k ← k + 1 do
2:    addRoot(k,length(A))
3: end for
```

Thus removing the minimum element takes a time proportional to the height of the tree in the worst case, i.e. $O(\log n)$ time, where n is the number of elements in the heap.

4.4.2.2 Insertion

Now consider the operation of adding an element to a heap (and reconstituting a binary heap).

Since heaps are complete trees, a new node can easily be added to the first available location from the left at the bottom level. We then

check the order property and adjust internal nodes. This is done in a 'bottom-up' fashion. We first compare the value of the new node with its father. If it satisfied the order property, the addition process is completed. If not, we swap it with its father. The checking process is then repeated on the new node on the level above. This process continues until the order property is satisfied (or the new element reaches the root position).

Example 4.13 *Figure 4.15 shows (a) a binary heap and how the order property is maintained after (b) a new element 2 is inserted at the bottom level of the heap, where the left most available location. (c) 2 is to be swapped with its father 5 since 2 is smaller than 5, (d) 2 is to be swapped with its father 4 because 2 is smaller than 4. (e) The process ends because 2 is at the root position and the order property is now restored.*

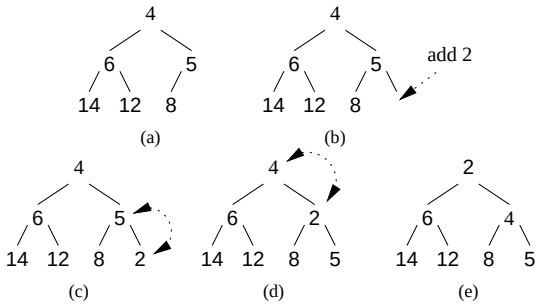


Figure 4.15: InsertOne

Thus adding an element also takes a time proportional to the height of the tree in the worst case, i.e. $O(\log n)$ time, where n is the number of elements in the heap.

Having studied how to add one element to a binary heap, we can construct a binary maximum-heap for a given list using the insertion method repeatedly. We look at this from Example 4.14.

Example 4.14 *Demonstrate, step by step, how to construct a max-heap for the list of integers* $A[1..8] = (10, 8, 14, 15, 12, 5, 6, 1)$. *Assume the heap is empty initially.*

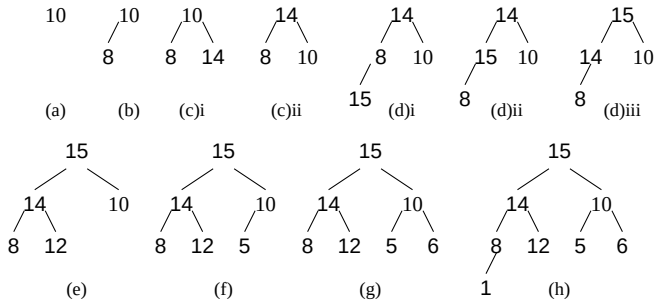


Figure 4.16: Construct a binary max-heap

Figure 4.16 shows the construction process from the initial state. Starting from the root, a new element is inserted, one by one, to the left most available position at the bottom level of the tree. (a) The first element 10 is the root, (b) 8 is added as its left child, (c)i 14 is

added and ii 10 and 14 are swapped to restore the order property, (d)i 15 is added, ii 15 is swapped with its father 8, iii 15 is swapped with its father 14 to restore the order property, (e)—(h) elements 12, 5, 6, and 1 are added respectively into the heap without a position swap.

This process is implemented on an array, where the new element x is marked by a box, e.g. $\boxed{x}$ and two swap elements x and y are underlined, e.g. $\underline{x} \cdots \underline{y}$.

$\boxed{10}$	8	14	15	12	5	6	1
10	$\boxed{8}$	14	15	12	5	6	1
10	8	$\boxed{14}$	15	12	5	6	1
$\underline{10}$	8	$\underline{14}$	15	12	5	6	1
14	8	10	$\boxed{15}$	12	5	6	1
14	$\underline{8}$	10	$\underline{15}$	12	5	6	1
$\underline{14}$	$\underline{15}$	10	8	12	5	6	1
15	14	10	8	$\boxed{12}$	5	6	1
15	14	10	8	12	$\boxed{5}$	6	1
15	14	10	8	12	5	$\boxed{6}$	1
15	14	10	8	12	5	6	$\boxed{1}$

Applications

A heap is a very useful data structure and can be used in many useful applications. For example, heap sort is one of the efficient sorting algorithms using heaps (see Section 6.13).

We look at a simple example.

Example 4.15 *Sort a list of integers* $A[1..8] = (10, 8, 14, 15, 12, 5, 6, 1)$ *using a max-heap.*

The list of $n = 8$ *unsorted integers is first converted to a max-heap, i.e. a partially sorted, left complete binary tree with the largest integer at the root as in Figure 4.12. The first position* $A[1]$ *is the root element.*

During the sorting process, the list is divided into two sections: $A[1..k]$*, the heap and* $A[k+1..n]$*, the sorted part (in shade in Figures 4.17–4.19). We shall each time remove the root element at* $A[1]$*, the largest integer from the current heap and insert it to location* $k+1$*, and* $k \leftarrow k-1$.

This is followed by a restoring of the order property of the heap. When the root r *is removed, we move the larger one of its children to the root position, and similarly, the larger one of the child's children to its position. The process repeats until the order property is restored.*

The following ordered steps show the sorting process:

1. *Figure 4.17(a).*
2. *Figure 4.17(b)*
3. *Figure 4.18(a).*
4. *Figure 4.18(b).*
5. *Figure 4.19(a).*

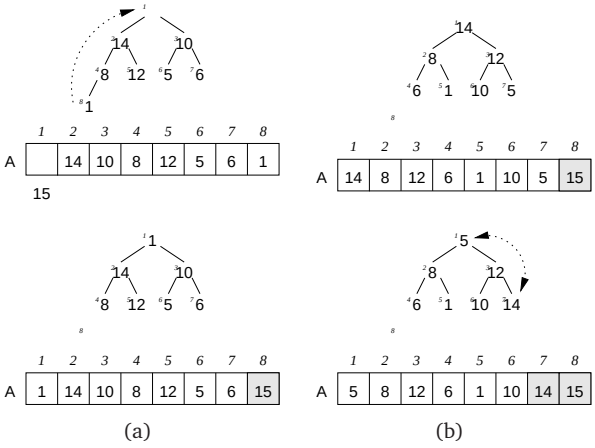


Figure 4.17: steps 1–2: addRoot-1,2

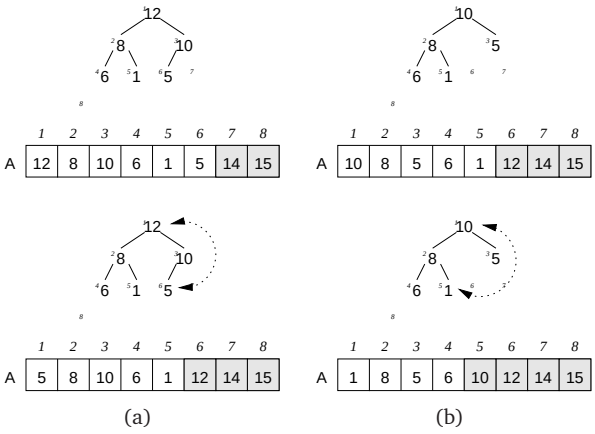


Figure 4.18: steps 3–4: addRoot-3,4

6. *Figure 4.19(b)*.
7. *Figure 4.19(c)*.

The above steps can be summarised in Algorithm 4.10.

Algorithm 4.10 heapSort(heapArray A)

1: buildHeap {Construct a heap}
2: **for** $k \leftarrow length(A)$ down to 2 **do**
3: $tmp \leftarrow A[k]$, $A[k] \leftarrow A[1]$, $A[1] \leftarrow tmp$ {swap A[1] with A[k]}
4: $addRoot(1, k-1)$ {Restore the heap properties for the shortened array}
5: **end for**

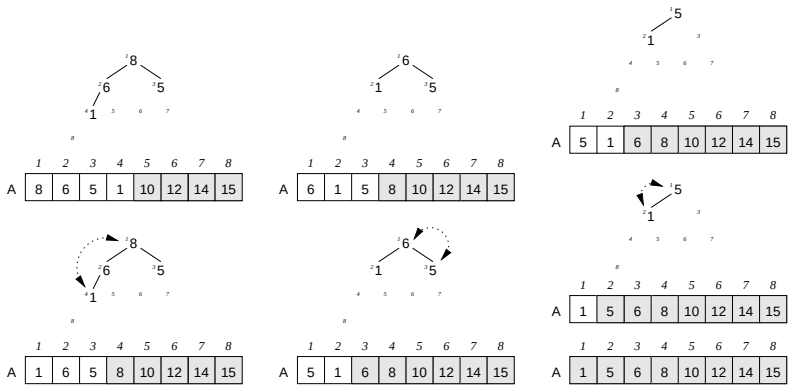


Figure 4.19: steps 5–7: addRoot-5,6,7

Activity 4.4

HEAPS

1. What is the difference between a binary tree and a tree in which each node has at most two children?
2. What is the approximate number of comparisons of keys required to find a target in a complete binary tree of size n?
3. What is a heap?
4. What is a priority queue?
5. Demonstrate, step by step, how to construct a max-heap for the list of integers $A[1..8] = (12, 8, 15, 5, 6, 14, 1, 10)$ (in the given order), following Example 4.14.
6. Demonstrate, step by step, how to sort a list of integers (2, 4, 1, 5, 6, 7) using a max-heap, following Example 4.15.

4.5 Graphs

Many relationships in the real world are not *hierarchical* but *bi-directional* or *multi-directional*. In this section, we study another abstract data structure called a *graph*, which represents such *bi-directional* relationships.

Problems as diverse as minimising the costs of communications networks, generating efficient assembly code for evaluating expressions, measuring the reliability of telephone networks, and many others, can be modeled naturally with graphs.

We first look at a real problem:

Suppose that we want to connect computers in four buildings (Figure 4.20) by cable. The *question* is: Which pairs of buildings should be directly connected so that the total installation cost would be minimum?

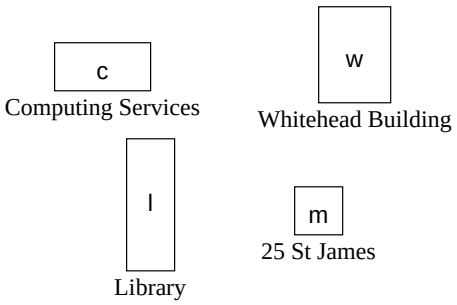


Figure 4.20: Four buildings

Naturally, the four buildings could be represented by four vertices with labels c, w, l, m (Figure 4.21). We then mark the cost between each pair of vertices by lines between vertices.

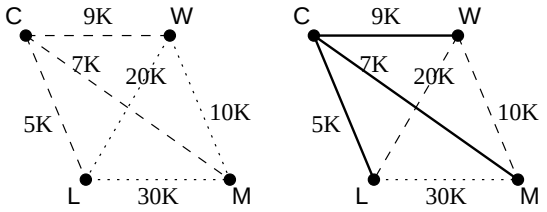


Figure 4.21: A graph

We look at all the possible connections (Figure 4.22):

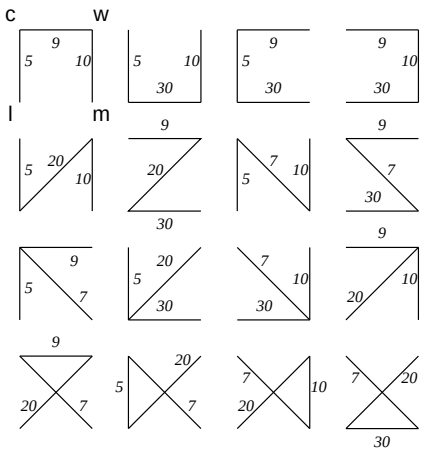


Figure 4.22: Possible connections

We could then make a decision to install the cable for the route that consists of direct connections (c,l), (c,m) and (c,w). The total cost would be: 5 + 7 + 9 = 21K.

Definitions

A Graph $G = (V, E)$ is a finite set of points (vertices) which are interconnected by a finite set of lines (edges) in a space, where V is a set of vertices and E is a set of edges.

In our example (Figure 4.21), $V = \{c, l, w, m\}$ and $E = \{(c, l), (c, m), (c, w)\}$. Normally, the set of vertices are represented by labels of numbers and the edges by letters.

There are two main classes of graphs: *graphs* and *directed graphs*. For graphs, the edge set consists of a *non-ordered* pair of vertices, e.g. (1,2)=(2,1), (2,3)=(3,2) etc. For directed graphs digraphs, the edge set consists of an *ordered* pair of vertices, e.g. (1,2)≠(2,1), (3,2)≠(2,3), etc. (Figure 4.23).

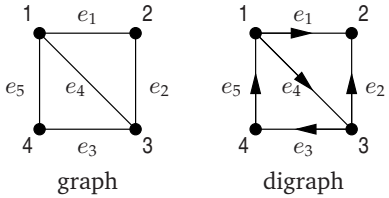


Figure 4.23: A graph and a digraph

As for trees, some terms and concepts are introduced for discussions on graphs:

Typical operations (Graphs):

order The number of vertices in a graph is called the *order* of the graph.

size The number of the edges in a graph is called the *size* of the graph.

path A path is a sequence of vertices $v_1, v_2, \ldots, v_k$, where $k \geq 1$ is a path if $(v_i, v_{i+1}) \in V$ for $1 \leq i \leq k - 1$.

length of a path The length of a path is the number of edges on the path, which equals $k - 1$, where k is the number of vertices on the path.

simple path A simple path is a path such that all vertices are distinct, except possibly the first and last vertices. No vertex on the path appears more than once.

self-loop A self-loop is a path from a vertex v to itself (v, v).

cycle A path $v_1, v_2, \ldots, v_k$ is a cycle if $v_1 = v_k$.

spanning tree A spanning tree of a connected graph is a subgraph and tree that contains all the vertices of the graph.

minimum spanning tree A minimum spanning tree of a weighted graph is a spanning tree of minimum total weight.

connected graph A graph is connected if there is a path between any two vertices of the graph.

complete graph A graph is complete if there is a path between *every* two vertices of the graph.

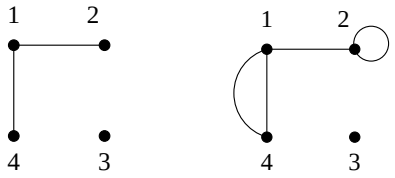
labelled graph A graph is labelled if every vertex has a fixed identity.

unlabelled graph A graph is unlabelled if there is no fixed identity for each vertex.

weighted graph A weighted graph is a graph in which every edge is associated with a real value as its weight (or cost).

simple graph A simple graph is a graph that contains no self-loop nor parallel edges.

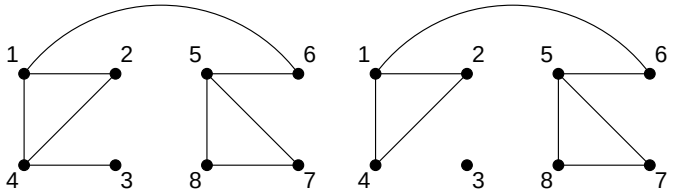
Example 4.16 *Figure 4.24(a) shows a simple graph while Figure 4.24(b) shows a non-simple graph with a parallel edge and self-loop. They are both disconnected.*



(a) simple (b) non-simple

Figure 4.24: A simple and non-simple graph

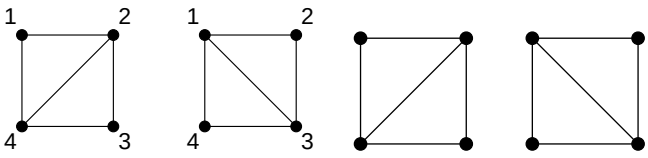
Example 4.17 *Figure 4.25(a) shows a connected graph. Figure 4.25(b) shows a disconnected graph with an isolated vertex. They are both unweighted.*



(a) A connected graph (b) A disconnected graph

Figure 4.25: A connected and disconnected graph

Example 4.18 *Figure 4.26(a) shows two labelled graphs and they are different, but the two unlabelled graphs in Figure 4.26(b) are the same. The graphs are connected and unweighted in both figures.*



(a) Two different labelled graphs (b) Two identical unlabelled graphs

Figure 4.26: Labelled and unlabelled graphs

We consider the simple graph in this course unit unless stated otherwise.

Representation of graphs

We discuss three commonly used data structures to represent graphs. They are *adjacency matrices, incidence matrices* and *adjacency lists*. As we shall see later, the use of different data

structures can sometimes improve (or worsen) the efficiency of algorithms.

1. Adjacency matrices

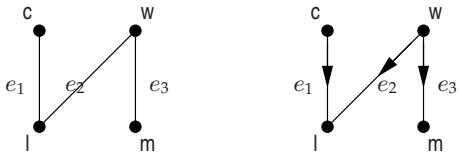
   A graph $G = (V, E)$ can be represented by a 0-1 matrix showing the relationship between each pair of vertices of the graph. We assign 1 or 0 depending on whether the two vertices are connected by an edge or not.

   Given a graph $G = (V, E)$, let n be the number of vertices. The adjacency matrix of the graph is a $n \times n$ matrix.

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{pmatrix}$$

   where

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



a graph a digraph

Figure 4.27: A graph and digraph

Example 4.19 *The Adjacency matrix for the graph in Figure 4.27 is,*

	c	*w*	*m*	*l*
c	*0*	*0*	*0*	*1*
w	*0*	*0*	*1*	*1*
m	*0*	*1*	*0*	*0*
l	*1*	*1*	*0*	*0*

$$A = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix}$$

and for the digraph in Figure 4.27 is,

	c	*w*	*m*	*l*
c	*0*	*0*	*0*	*1*
w	*0*	*0*	*1*	*1*
m	*0*	*0*	*0*	*0*
l	*0*	*0*	*0*	*0*

$$A = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Note The Adjacency matrix for an undirected graph is always symmetric if row and column nodes are listed in the same order.

2. Incidence matrices

   An incidence matrix represents a graph G by showing the relationship between every vertex and every edge. We assign 1 or 0 depending on whether a vertex is incident to the edge.

   Let n be the number of vertices of the graph, and m be the number of edges. The incidence matrix is an $n \times m$ matrix:

$$B = \begin{pmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,m} \\ \vdots & \vdots & \vdots & \vdots \\ b_{n,1} & b_{n,2} & \cdots & b_{n,m} \end{pmatrix}$$

where for a *graph*

$$b_{i,j} = \begin{cases} 1 & \text{if vertex } i \text{ and edge } j \text{ are incident} \\ 0 & \text{otherwise} \end{cases}$$

For a *digraph*,

$$b_{i,j} = \begin{cases} -1 & \text{if edge } j \text{ leaves vertex } i \\ 1 & \text{if edge } j \text{ enters vertex } i \\ 0 & \text{otherwise} \end{cases}$$

Example 4.20 *The incidence matrix for the graph in Figure 4.27 is*

	e_1	e_2	e_3
c	*1*	*0*	*0*
w	*0*	*1*	*1*
m	*0*	*0*	*1*
l	*1*	*1*	*0*

$$B = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

and for the digraph in Figure 4.27 is,

	e_1	e_2	e_3
c	-1	*0*	*0*
w	*0*	-1	-1
m	*0*	*0*	*1*
l	*1*	*1*	*0*

$$B = \begin{pmatrix} -1 & 0 & 0 \\ 0 & -1 & -1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

Note Incidence matrices are *not* suitable for any digraph with a self-loop ('loop' for short).

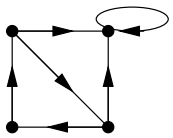


Figure 4.28: A digraph with a self-loop

3. Adjacency lists

   In an adjacency list representation, a graph $G = (V, E)$ is represented by an array of lists, one for each vertex in V. For each vertex u in V, the list contains all the vertices adjacent to u in an arbitrary order, usually in increasing or decreasing order for convenience.

   Why an adjacency list?

   It is space efficient for sparse graphs where the number of edges is much less than the squared power of the number of vertices.

   Example 4.21 *Suppose that we need to store the digraph with few edges in Figure 4.27 . Suggest a data structure for the digraph.*

   Solution *An adjacency list can save space for a sparse graph (Figure 4.29).*

Implementations

Like other abstract data structures, the best implementation of a graph, here by an array of lists, or by an array, depends on the given problem. It is conventional to label the vertices of a graph by

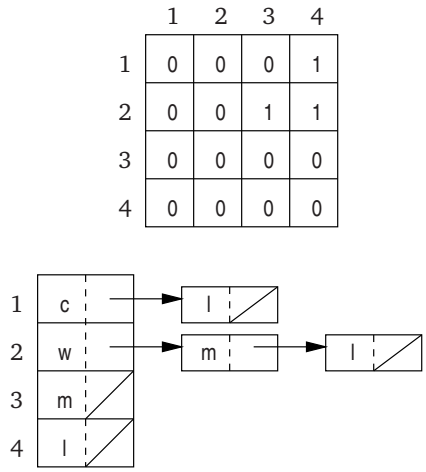


Figure 4.29: An adjacency list

numerals. For our example (see the digraph in Figure 4.27), the labels for c, w, m, l can be replaced by 1, 2, 3, 4 respectively. Hence the data structure may be represented as follows:

Example 4.22 *See Figure 4.30.*

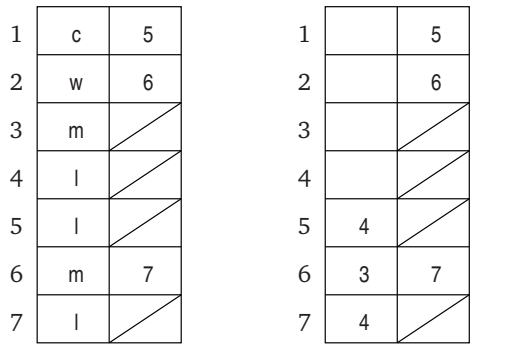


Figure 4.30: Implementation

Graph algorithms

For many important problems about graphs, no efficient algorithms have been found. A lot of effort has been expended on seeking efficient algorithms for graph problems. Interesting algorithms have been found for some graph problems but a comprehensive account of these problems would fill volumes. We will look at a few of these problems here.

One important class of graph problems is about graph traversal. As with binary trees, we would like to investigate all the vertices in a graph in some systematic order. We also want to avoid cycles during the traversal. With many possible orders for visiting the vertices of a graph, two traversals are particularly important, namely *Depth-first traversal* and *Breadth-first traversal*. They are also called *Depth-first*

search and *Breadth-first search*.

See Section 5.3.3 for the algorithms.

Activity 4.5

GRAPHS

1. Consider the adjacency matrix of a graph below:

$$A = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \end{pmatrix}$$

   (a) Draw the graph

   (b) Write the adjacency list for the graph

   (c) Discuss the suitability of using an adjacency matrix and the adjacency list for the graph. Justify your answer.

2. Using the adjacency matrix approach, write a program to store a simple graph and display the graph.

   Example 4.23

```
          Store and Display
           a simple graph
      ------------------------
        1. Store a graph
        2. Display a graph
        0. Quit
   Please input your choice (0-2) >
```

   Hint An easy approach may be:

   (a) define a data structure for the graph

   (b) decide a means to input the graph, for example, you may

      i. type the entries of the adjacency matrix on the keyboard

      ii. read the entries of the adjacency matrix from a text file

      iii. generate a random adjacency matrix by a program.[3]

   (c) write the main program or method with interfaces of the sub-methods or procedures

   (d) develop each part of the program.

3. Implement the graphs in question 1 using the adjacency list approach.

[3] Use a random generator to generate a 0 or 1 uniformly at random for each entry of the matrix.