

Fundamental Data Structures

Smart data structures and dumb code works a lot better than the other way around.

Eric S. Raymond, *The Cathedral and the Bazaar*

We recall from the previous Chapters that data viewed on the lowest level of abstraction is nothing more as a sequence of bits and bytes. In fact, data in main memory and on the hard disc of a computer are organized on the “operating system level” as a sequential storage space. Searching for information sequentially (byte-by-byte) is not efficient and usually data should be viewed and accessed on a higher level of abstraction. Examples for nonlinear structures are the omnipresent tables or matrices, hierarchies, graphs, and many others.

CHAPTER OBJECTIVES

- To describe fundamental data structures conceptually.
- Provide example use for these structures.
- Show how to implement some of the structures and its operations.

Let’s take as example a person’s record that shall contain the person’s first and last name, street, zip, city and up to 4 phone numbers. The data representing these information will be binary coded (see Chapters 1 and 2) and should appear in the same sequence as listed above. For the moment we do not know how to access a particular information other than reading it sequentially.

If we want to know a person’s phone numbers it is inefficient to access the record sequentially from the beginning since we are not interested in the address of the person. We need a mean to access the phone numbers directly. We learned from Chapter 2 that we can store the 4 phone numbers in an array and access them *logically* by it’s index position. By *logically* we mean a programmer’s view using a high level language like C or Java. On the *operating*

system level (our lowest abstraction level) a data item is located by a byte offset within the storage media.

At the moment, we do not know how to find the operating system level addresses of the phone numbers within that record, but we will find a solution later in this chapter. We only know that the sequence of bits and bytes have a structure imposed which groups the information represented.

In general, any grouping or arrangement of data elements following certain rules is called a **data structure**.

An important classification for data structures is whether the structure is of fixed size (*static*) or if its size can change (*dynamic*). Another criterion for distinction is whether the data elements stored in the structure need to be all of the same type (*homogeneous*) or if its type may vary (*inhomogeneous*). Structures which can accommodate different data types are called *polymorphic* (Greek: $\pi\omega\lambda\iota$ (poli)= many, $\mu\omega\rho\phi\epsilon$ (morphe)= shape).

4.1 Arrays

One omnipresent structure in programming is the array. An (one-dimensional) **array** is a sequence of n data elements of the same type under a single name. The elements of the array are identified by its position using an index. Arrays are very useful for storing a fixed number of values of the same type and accessing the elements by its index. As element any elementary or compound data type can be chosen as long as the size of the element is fixed.

Because an array has a fixed number of elements and all elements are of the same type the structure is classified as static and homogenous.

We have seen an array named “A” of 8 integer numbers in Figure 2.2 of Chapter 2. The index ranged from 0 to 7, where $A[0]$ points to the first and $A[7]$ points to the last element. For instance, we can access the 5th element by the name $A[4]$. The physical storage model is sequential, but as the element size is fixed, we can locate $A[4]$ as 4 times the element size plus the starting address for the array A . If our array A of 8 integers starts at byte location 100 and each integer occupies 2 bytes then the address of $A[4]$ will be at byte location $108 = 100 + 4 \cdot 2$ as shown in 4.1. This means that element $A[4]$ can be located directly by reading byte 108 and 109 (two bytes for the integer) on the storage media.

In general if we have an array A with indices 0 to $n - 1$ the address location for $A[i]$ ($i = 0, 1, \dots, n - 1$) is:

$$address(A[i]) = address(A) + i \cdot element_size$$

This concept of addressing can be extended to arrays with more than one index. The number of indices used to reference an array element is called its **dimension**. If an array has a dimension greater than 1, it is called *multidimensional*.

A two-dimensional array with index ranges n and m is called a **$n \times m$ matrix** or **table**. The first index (n) is referred to as *rows* and the second index (m) is referred to as *columns* of the table.

As example take a 5×4 matrix of integers, called “Temp” where the numbers in each column could be the temperature at 4 different hours of a day and each

Index no	Array A	Address
0	6	100
1	101	102
2	2007	104
3	3	106
A[4] → 4 →	79	108
5	1025	110
6	2	112
7	28	114

A[4] = 79 at address 108, array cell size = 2 bytes

Figure 4.1 Example layout and addressing of an array with 8 elements

Addresses	0	2	4	6
	M ₀₀	M01	M02	M03
8	M10	M11	M12	M13
16	M20	M21	M22	M23
24	M30	M31	M32	M33
32	M40	M41	M42	M43

Cell size = 2 bytes

Figure 4.2 Example storage of a matrix with 5 rows and 4 columns

row represents a different day. The result is a table like in Figure 4.2 where the second temperature measured on day 3 is denoted by $Temp[3, 2]$. The relative address $addr(M(n, m))$ in the general case of a $n \times m$ matrix is calculated as:

$$addr(M[n, m]) = (n - 1) \cdot m \cdot s + (m - 1) \cdot s, \text{ (assuming cell size = } s \text{)}$$

The concept of an (multidimensional) array is not limited to the data type integer. The same formulae holds for any data type of a fixed size. You only need to replace the variable s (= cell size in bytes) by the actual size of the data type.

4.2 Records

Often the data elements are not of the same type. As example consider the properties of a person. Each property (i.e. name, date of birth, skills, SSN, salary, etc) has a different data type and size. But, as all belong to the same person we want to list these attributes together in an inhomogeneous structure. Conceptually a record defines data elements which are all related to an object that the record describes.

Person record

Name		DOB	SSN	Salary	Skills			
Last	First				0	1	2	3
Smith	Sam	1977/02/06	364-26-4738	48 000	op sys	progr	proj.	

Figure 4.3 Example record of a person

A grouping of a fixed number of (inhomogeneous) data elements is called a **record**. Each data element of the record is *logically* referenced by its name.

The definition does not assume that the elements are ordered, that is the reason why elements are addressed by name and not by position. In Mathematics a similar structure is the *tuple*. But here the elements are ordered and not named. That is why the elements of a tuple are referred by its position.

The conceptual structure of a record defines a list of elements, but each element itself may be a record or another compound type which allows constructing complex record structures as in figure 4.3. A specific element of the record is referenced on the logical level by a “dot notation” like:

- person.name.first
- person.name.last
- person.salary
- person.skills[2]

On the operating system level for each data item we need to know its relative byte address within the record. Therefore a mapping table is used that maps each record element identified by its name to an address like in the following table:

person.name	1000
person.name.first	1000
person.name.last	1010
person.salary	1030

If a record element, say *person.name.last* is used in the program, the compiler will translate the name into a relative byte address using the above table.

A record may contain a references to other records thus forming a network of records. Such a network represents a higher level data structure itself building structures like trees, indexes, etc. This structures will be discussed later in this chapter.

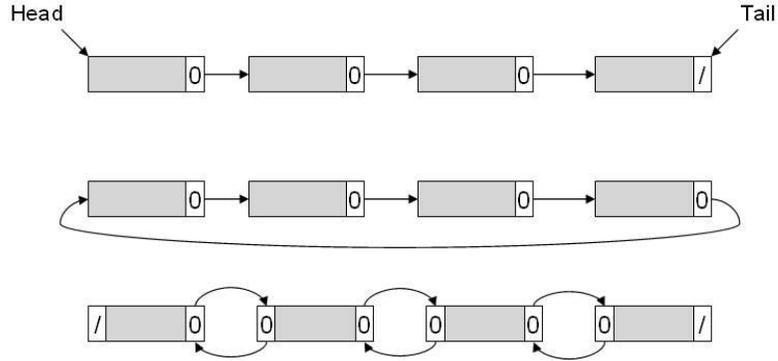


Figure 4.4 Link structures for lists

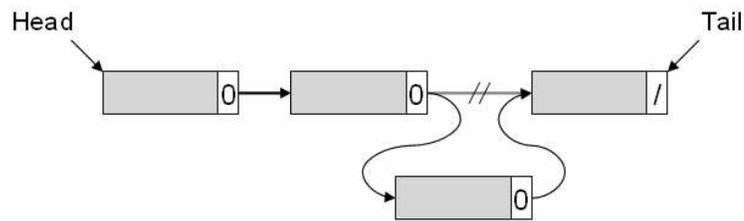


Figure 4.5 Inserting an element into a linked list

4.3 Lists

Sometimes we do not know the number of elements in advance, or an additional element has to be inserted in the middle of a structure. For these cases a dynamic, linear structure is appropriate.

A sequence of data elements where elements can be inserted or removed at any position is called a **(linked) list**. Each data element except the last (called *tail*) has a successor and each element except the first (called *head*) has a predecessor.

The conceptual model assumes that the elements are linked together by pointers. The pointer of the last element should indicate the end of the list by a special value (*nil*) which cannot be a legal address. The linking can be forward only, circular, or back- and forward (see figure 4.4).

List elements are accessed sequentially by moving to the next or previous element. It is not possible to jump directly to a particular element because there is no way to calculate the position in advance due to the dynamic nature of the list. This is a clear limitation of this structure.

An element can be inserted before or after the current position. Figure 4.5 shows an example of inserting a new element into a list. To remove an element it is sufficient to “unlink” the element. This can be done by skipping the element subject to be removed by changing the link from the predecessor to point to the following element. A mechanism is needed to mark the “unlinked” element as free space (see figure 4.6).

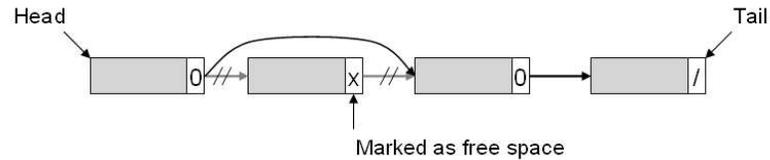


Figure 4.6 Removing an element from a linked list

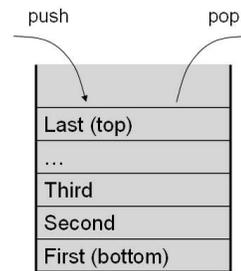


Figure 4.7 Stack and its operations

4.3.1 Stack

In many cases the full functionality of a list is not required. Computer languages and calculators sometimes use the *reverse Polish notation* to hold values and retrieve them later in reverse order. This behavior can be easily simulated by a list.

A **stack** is a list where data elements are only inserted or removed at the head. A synonym for stack is *LIFO* (Last In First Out).

A stack is like a pile of plates where you only add or remove a plate on the top. The top element is always the last added and first to be removed.

The main operations on a stack are: (see figure 4.7)

- *push* to add a new element
- *pop* to retrieve the top element
- *peek* to check to top element without removing it

A nice application for a stack is the “back” button on your browser. It helps you to find your way back in a “surf session”. More examples are the “undo” button of an editor or the command history.

4.3.2 Queue

Another special list is the *queue* which is used for *buffering* or delaying things without changing their order. Items are added only at one end, and removed only from the other end of the list. This ensures that the elements are kept in order and the last element is removed next.

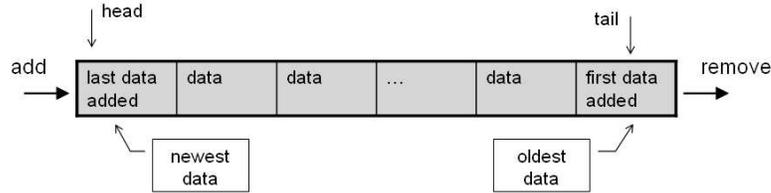


Figure 4.8 Queue and its operations

Typical examples for a queue are the letter in-box and the to-do list. Normally these lists are processed in the order of its entry.

A **queue** is a list where data elements are only inserted at the head and only removed at the tail. A synonym for queue is *FIFO* (First In First Out).

The main operations on a queue are:

- *add* to add a new element at the head
- *remove* to retrieve the element at the tail
- *empty* to check if there are no more elements
- *size* returns the number of elements in the queue

The queue mechanism is depicted in figure 4.8.

In reality the average insertion rate of elements should not exceed the average removal rate since the queue would grow ad infinitum and blast the available memory. Usually a fixed number of queue elements will suffice to balance possible insertion bursts or removal delays.

Here is a possible realization of a queue with a limited size of n elements. This structure is called *ring buffer of size n* . We need an array $q[n]$ and three variables *head*, *tail*, and *state*. The *head* contains the index of the position where the next element has to be added and *tail* contains the index to the position where the next element will be removed. The state indicates whether the queue is *empty*, *full*, or none of both (has elements, but is not full). Figure 4.9 shows a ring buffer of size $n = 10$.

First, we describe its mechanism verbally, and second, we provide a flow chart as introduced in Chapter 2.

The ring buffer is initialized with *head* and *tail* pointing to index 0, i.e. the queue is empty. When an element is added at position 0 the value of *head* is advanced by one. After having inserted 4 elements, *head* reads 4. The first element to remove is the first element added which is found at index 0. When this element is removed, we have to advance the value of *tail* by one (see Figure 4.9). We can remove elements as long as *tail* is “behind” *head*. But what happens when we reach the end of the array? Assuming that meanwhile the tail has advanced, e.g. to index 5 we begin inserting at position 0 again. We can do so as long as we do not run into the elements not yet removed. When we hit the tail of the queue we have to stop inserting and rise a “queue full” condition.

There is one technical detail with this ring buffer: we cannot use the comparison $head > tail$ for the “behind” condition since we wrap around

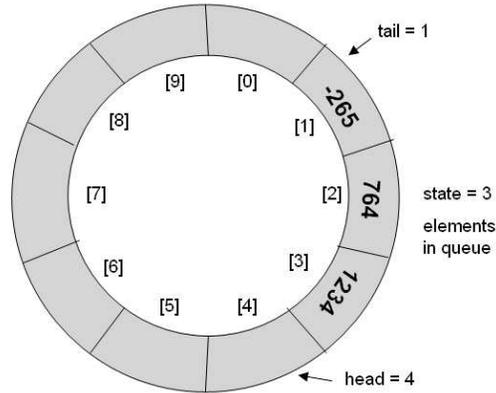


Figure 4.9 Ring buffer

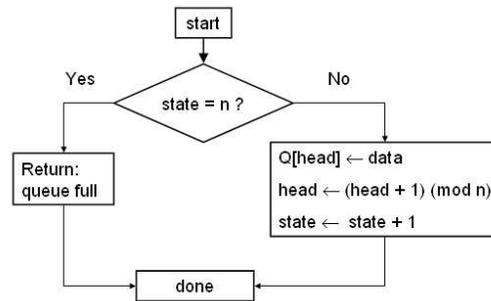


Figure 4.10 Algorithm to **add** an element into the ring buffer

when we reach the end of the array. Luckily it is enough to check the *state* to know whether the queue is empty or full. In our flow chart in Figure 4.10 we use an integer variable *state* to hold the number of actual elements present in the queue. The states *empty* and *full* are then indicated by the value 0 and *n* respectively. The algorithm shows clearly that we cannot insert an element into a full queue. Likewise it is not allowed to remove an element from an empty queue.

It is left as an exercise to design an algorithms for the **remove** operation.

4.4 Trees

Hierarchical structures are very prominent. Organizations like companies, army, government, project teams, and etc form tree like personnel structures. Computer and telephone networks are other examples of hierarchical structures. One of the main benefits of this structure is that the cost for reaching any person in the organization (or any device in the network) is growing only with the logarithm of the number of the persons (or devices). The other benefit

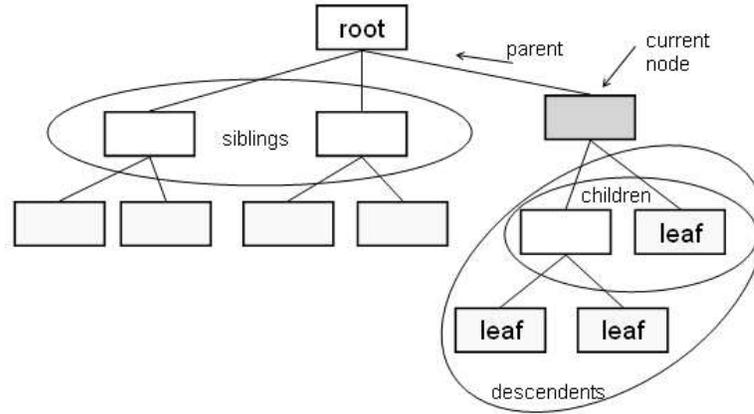


Figure 4.11 Tree terminology

is that each element has at most one parent. These are the reasons why this structure is so popular.

A **tree** is hierarchical structure (see Figure 4.11) of elements (called *node*). Unlike in nature, in computer science trees grow upside down. The trunk is on the top (top level node) and the leaves (terminal nodes) are at the bottom. Each element except the top level node (called *root*) has exactly one immediate ancestor (*parent*) and each element except the terminal (of *leaf*) nodes have immediate descendant (*child*) nodes. All nodes with the same parent are called *siblings*.

All descendants of a node form a *subtree* with the current node as its root.

The longest path from root to leaf is called the *depth* of a tree. If all paths from leaf to root are of the same length then the tree is called *balanced*.

If each node in a tree has at most two children it is called *binary tree*.

A tree has to support many operations:

- *node manipulation* to add, modify, or remove a node
- *node query* to search for and retrieve a certain node, its parent, siblings, or children
- *node information* determine the position (leaf, root) and type of a node
- *tree information* to determine the *size* (number of nodes) and *depth* of a tree

A tree starts initially with a root node that is inserted when a new tree is created. Then child nodes may be added to the root. In fact you can add a child to any node. Adding a child node from a leaf node position introduces a new node level and makes the inserted node a leaf node and the current node will become its parent. This way we can build up a hierarchy of nodes of any depth.

It is never possible to add a parent node because the constraint that each node has at most one parent would be violated.

The node manipulation is usually relative to the current node. We can modify or remove the current node. Deleting the current node will result in removing all child nodes as well and in sequence the whole subtree.

4.4.1 Ordered Trees

In an **ordered tree** the children of each node appear in a certain sequence. Relational properties like *first*, *last*, *previous*, and *next* apply.

To insert a node we need should specify where a node should be inserted or deleted. This can be as previous (resp. following) sibling, or as (first, resp. last) child of the current node.

In the case of an ordered tree, an appropriate navigation strategy makes it unnecessary to traverse the whole tree to find a specific node. If we know the order of the nodes, the navigation to the wanted node is always determined. If n is the number of steps from the root to a node and d is the depth of the tree, then $n \leq d$.

As example take the lexically ordered tree of person names from Figure 4.13. We note that for each node the “left” descendants are all smaller and the “right” descendants are all larger than its parent node. This makes it easy to navigate to a certain node. If you look for a name that is less (resp. bigger) than the current node’s name than navigate to the left (resp. right).

4.4.2 Tree navigation

As there is always one and only one root node, the first access to a tree is its root node. If we are already at a certain node we can either go to its siblings, its children or its parent. Finding the direction in a tree is called *navigation*.

In a tree there exist exactly one way (called *path*) from the root node to any other node.

For querying a tree each node must be visited. This movement through the whole tree called **traversal**.

Starting from the root node we have two possibilities:

- *pre-order* traversal: visit each node before its children
- *post-order* traversal: visit each node after its children

Figure 4.12 demonstrates the differences of tree traversal.

In the post-order mode the navigation process looks for the children of a node first, i.e. moves to the first child if it exists. This is repeated until a leaf node is reached. Then its siblings are navigated to. If there are no more siblings the parent is visited. We are finished when the root is read.

The pre-order mode starts with the root. Only after the current node is read its children are visited. If there are no more children we move to the next sibling repeating the same procedure again.

4.4.3 Search Trees

If the nodes of a tree represent persons, we could sort them by birthday, alphabetical by name, by weight, by height, or any other criteria. In general, we give the sort order criteria a name or unique symbol (e.g. \angle) to distinguish different orderings. Take the numbers as example, they can be sorted in ascending ($<$) or descending ($>$) order.

An ordered tree whose nodes appear during a predefined traversal in the order of \angle is called a \angle -**sorted tree**.

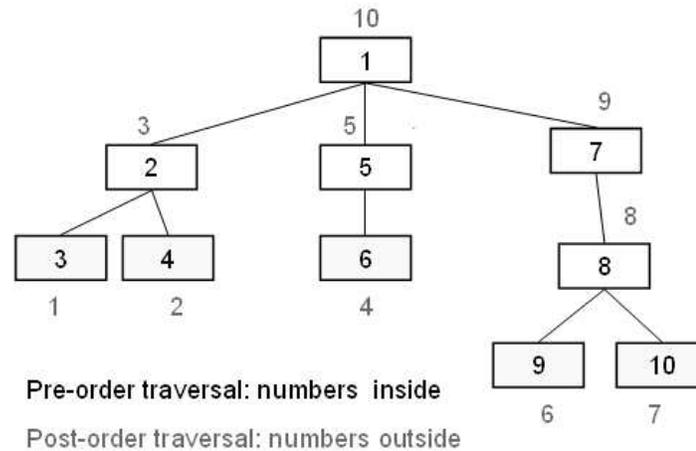


Figure 4.12 Pre- and post-order traversal through a tree

If the sort criteria allows equality and two or more nodes are equal according to the criteria, these nodes are placed next to each other with regard to a predefined traversal method.

A **binary search tree** is a <("less than")-sorted binary tree where every node's "left" subtree contains only nodes less than the current node.

Lets assume a binary tree where all leaf nodes have a path length that is equal to the depth of the tree as in figure 4.13. We call this tree *balanced*. On level 0 we have only the root node, on level 1 we have 2 nodes (the children of the root). Each level further down the number of nodes is doubled. In general on the n^{th} level we have 2^n nodes. A fully balanced binary tree of depth n contains $2^{n+1} - 1$ nodes in total.

To demonstrate how a binary search tree is used and how many nodes have to be visited before a certain node is found, let there be 64000 persons. We build a binary search tree for these people where we choose the alphabetic order of the names as sort criteria. The resulting tree has a depth of 15 because $2^{15+1} - 1 > 64000 > 2^{15+1} - 1$. Searching for the person named "Turing" is done stepwise by beginning with the root node (person "Knuth"). As "Knuth" is alphabetically less than "Turing" the right child node "Schickard" is chosen. As "Schickard" is less than "Turing" the next node is again the right child. Finally we hit the node "Zuse" which is alphabetically bigger then "Turing". Now we have to choose the left child branch which is the desired person. If we reach a leaf node and the person is still not found, it is not present in the tree.

Finally lets have a look on the costs of searching for a particular entry using a binary search tree. The depth of a search tree corresponds to the maximum reading steps necessary to find any entry. A binary tree of depth n holds between 2^n and $2^{n+1} - 1$ nodes. Taking the 2-logarithm leads to the formula:

Let B be a binary search tree with b nodes. Then the number k of operation steps to find a given data element is limited by the 2-logarithm of b :

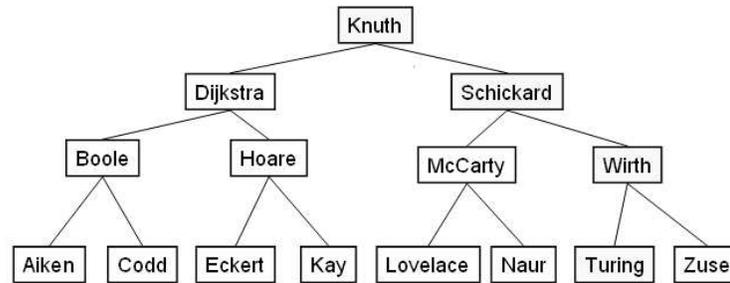


Figure 4.13 Balanced binary tree

$$k \leq \log_2(b)$$

Each step consists of constant costs: reading a value and comparing it to another. The total cost for finding a node in a binary search tree is $< konst \log_2(\text{no. of nodes})$. This means that the costs are only growing only with the logarithm of the size of the tree.

4.4.4 Heap Tree

The **heap tree** is a special form of a binary search tree with two additional properties:

- The leaf level of the tree is filled from left to right
- The value of a node is larger than the values of its children (called *partial order*)

Heap structures are useful to store a partially ordered collection of nodes. To find the largest node it is sufficient to read the root node.

If the root element is removed, the tree structure is broken. To fill the hole, move the far right leaf node on the lowest level, say e , to the top of the tree as in 4.14. The tree structure is now restored but the root element e violates the second property of a heap. To correct the situation rotate the nodes along the path with the largest values of the descendants until the maximum value is in the root position. In other words, the element e sinks down the tree until its descendants have smaller values than e . This “heapify” mechanism is illustrated in Figure 4.14.

The heap can be used to produce a sorted collection. Just remove the root element repeatedly and add it to a list until the heap is empty. This feature will be used for the heap sort algorithm described in the next chapter.

Other applications of the heap are priority queues, spanning tree, and shortest path problems.

A heap could be implemented as an array if the children of a node with index i are located at the double indexes $(2i, 2i + 1)$. The parent-child relation follows from the array indices: node $a(i)$ has children $a(2i)$ and $a(2i + 1)$ if $(2i + 1) < n$ and node $a(i)$ parent is $a(i/2)$ rounded to the next lower integer $(a \lfloor i/2 \rfloor)$. For an array implemented heap the heap conditions read as

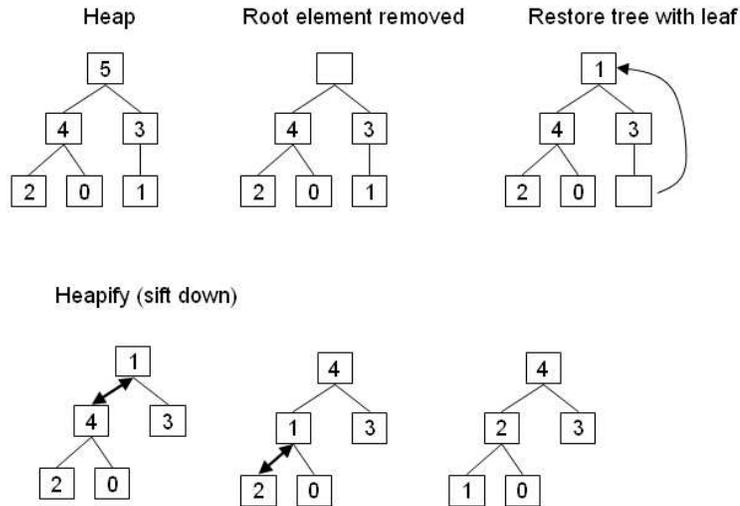


Figure 4.14 Example heap with an heapifying process

$$a(i) \geq a(2i) \text{ and } a(i) \geq a(2i + 1).$$

Using an array for implementing a heap has the benefit to directly jump to a certain node by just addressing it by its index.

4.5 Graphs

Graphs are a generalization of the previously discussed data structures:

- array
- record
- list
- tree

A **graph** is a collection of interrelated data elements (called *nodes*). The direct connection between two nodes n_i and n_j is called *edge*.

If all connections are directed, we call this a **directed graph**. An **acyclic graph** is a graph where the edges do not form cycles.

The most important example for a *Directed Acyclic Graph* (DAG) is a tree. But there are DAGs which are not trees.

There are many graphs structures found in nature, e.g. the spider webs, trees, rivers, the human blood circulation, the nerves system, and more. Communication structures like the railroad net, highways, telephone network, and the internet form graphs. But also many concepts form abstract graphs. Examples are classification schemes in biology, semantic description of a knowledge domain (ontology), class hierarchy, etc.

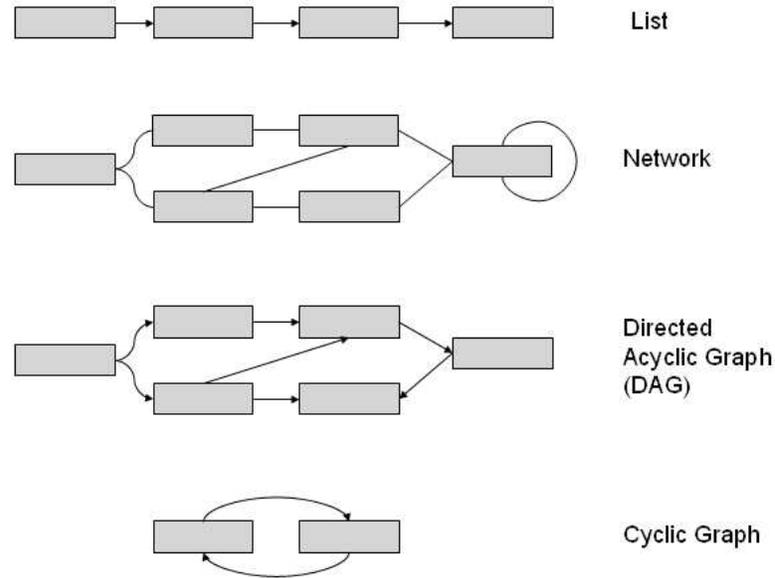


Figure 4.15 Graph examples

These examples show that the graph concept is so fundamental that it can be used in virtually any situation to describe objects (the nodes) and their relationships (the edges) to other objects.

4.6 Sets

In Mathematics the concept of a set belongs to the most fundamental topics. In computer programs they are less used because its implementation is not very efficient unless using other structures for its implementation.

A **set** is an unordered collection of data elements without any duplicates. The number of elements a set contains is called its *cardinality*.

Given two sets A , B , then the following operations are the well known:

- union ($A \cup B$)
- intersection ($A \cap B$)
- difference ($A \setminus B$)

Normally the *complement* of A (this is $U \setminus A$) does not exist in a computer because there is no universal base set U in a computer. Fortunately all sets considered here are finite so we don't have to fear anomalies due to the fact that we cannot build a complement.

Sets are useful if one does not care about the ordering but needs to assure identifiable elements without duplicates.

For route planning the usage of sets is quite beneficial. Say, we want to know how many cities can be reached by one stop from New York. This is

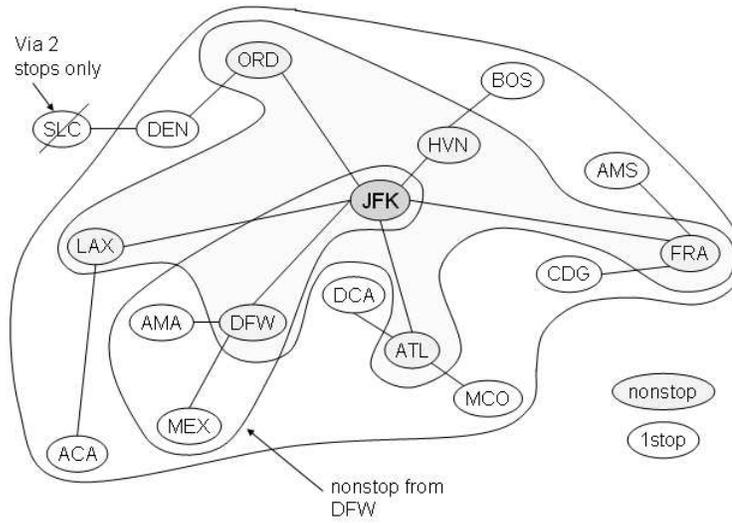


Figure 4.16 Cities that can be reached with one stop from NY

easily done if we put all cities in a set that can be reached non-stop. Then, for all these cities we union the sets of cities that can be reached non-stop from there. The cardinality N of the result is the answer to our question. As formula for N we have:

$$N = \left| \bigcup_{c \in nsCities(NY)} nsCities(c) \right|$$

where $nsCities(x)$ is the set of all cities that can be reached non-stop from city x . All possible duplicates are removed when doing the *union*.

4.7 Indexing

Sequential access to large data structures like files with many records is time consuming. Let S be a data structure with n elements, then an average of $n/2$ data elements have to be read before the wanted element is found.

If we can limit the range of elements to search to a fixed number k , then we have to read only $k/2$ on average, no matter how large the structure is.

To motivate the use of an index let's assume that a super market sells 10000 different articles. The data for each article is maintained as record in a file. If we look for a certain product reading 5000 records on average is not acceptable. So we sort the articles in the file by product groups and make a product group list along with the position (*index*) for each group within the file as shown in Figure 4.17.

If a group contains about 100 articles, and if we know the product group of our article we search, then on an average it is sufficient to read 50 group plus 50 article records. This is 50 times better than before.

An **index sequential structure** is a sequential structure with an index associated to it. IBM's *Index Sequential Access Method (ISAM)* was one of the most famous commercially used file structure using indexes.

Beside the benefit of access acceleration there is a drawback when adding new elements to the data structure because a new index entry has to be made and when the index is sorted, this may lead to a reorganization of the index.

As the number of index entries grows with the number of data elements, the index eventually needs an index itself. As the index can be organized as a search tree the access will still be very fast.

To reduce the number of indices we may use only one index for a fixed number (block) of data elements as in the previous example. In practice the index is often small enough to fit into main memory. We can make sure that the block size does not exceed the physical data transfer unit from the disc file. This reduces the access time for a data element (record) to one physical disc read operation.

4.8 Hashing

As we have seen an index grows linearly with the number of data items. It would be great to overcome the lookup in the index for the position of the data. As an identifier defines an injective mapping between a data object and its identifier we could try the other way round. If we succeed we can compute the location of a data element in a data structure from its identifier.

The difficulty in this approach is that normally there are much more possible identifiers (the set I) than data elements to store. To be economical we have to give up the one-to-one mapping. The challenge is to find a function which maps the really used identifiers with as less conflicts as possible.

Let K be a set of possible keys (identifiers) and m the maximum number of data elements. A **hash function** h is a mapping from K onto (surjective) \mathbf{N}_m , where \mathbf{N}_m are the natural numbers less or equal m .

$$\begin{aligned} h : K &\rightarrow \mathbf{N}_m \\ h(k) &\mapsto n \end{aligned}$$

The natural number n represents the location of a data element (an address or relative position). The space reserved or needed for a data element is called a *bucket* or a *slot*.

A hash function which guaranties an injective mapping is called *perfect*.

As the hash function can be written as a table consisting of pairs (key, address) it is called *hash table*.

Keep in mind that we do not need the hash table to do hashing. The hash function is sufficient to compute the address from the key. The hashing requires a predefined address range \mathbf{N}_m . That means that the data structure and the maximum number of elements are fixed.

The hash algorithm works with little modification for non-numeric keys as well if we use any binary code the characters and interpret the bits as number.

To demonstrate the power and use of hashing suppose IBM wants to store its US based 160 000 employees and use the SSN (Social Security Number) as key for each person's record. We have to define a hash function that maps the

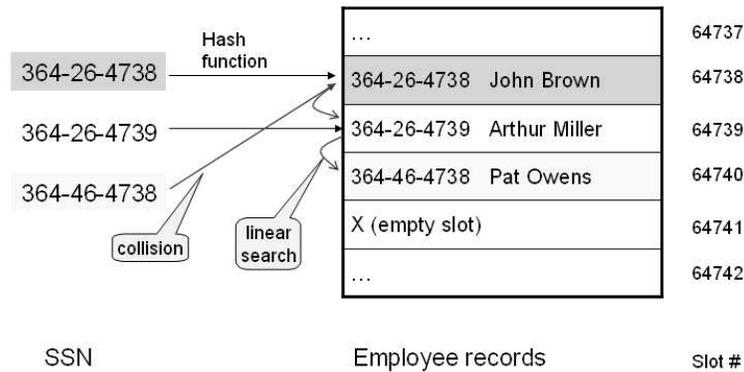


Figure 4.18 Hash addressing example with SSN as key

9 digits SSN to one of the possible 200 000 buckets. We will use the modulo division This can be done by the following function h :

$$h : \{SSN\} \rightarrow \mathbf{N}_{200\,000}$$

$$h(SSN) \equiv SSN \pmod{200\,000}$$

The function $h(SSN)$ returns the remainder of $SSN/200\,000$ which is less than 200 000. Let's take the employee with the $SSN = 364 - 26 - 4738$. Then, the remainder is computed with the following formula:

$$h(SSN) \equiv SSN \pmod{200\,000} = 364\,264\,738/200\,000 - 1821 \cdot 200\,000 = 64\,738$$

The person's record will be found at record number 64 738 if no collision occurs (see Figure 4.18). Only one single access to the file is necessary to access the record no matter how many employees are stored. This is the most important advantage of the hash mechanism. Storing the record of employee with $SSN = 364 - 26 - 4738$ will occupy the next slot number 64 739. But, it is possible that we find an employee that maps to an already stored data record.

When two keys map to the same address, we have an address collision. For that case we need an alternative address by:

- *double hashing*: using a different hash function
- *probing*: searching for a free location.

With a different hash function we can again collide with another data element. In fact, a list of function is required until an empty address is found. Searching for a free address can be time consuming and a chaining mechanism is needed to find all colliding entries. To ease probing a special *overflow area* can be assigned to store colliding data and maintaining a pointer to the beginning of the empty area.

However, if a good hash function is chosen and the load factor is not exceeding 70% then the risk is very low to have many collision as figure 4.19

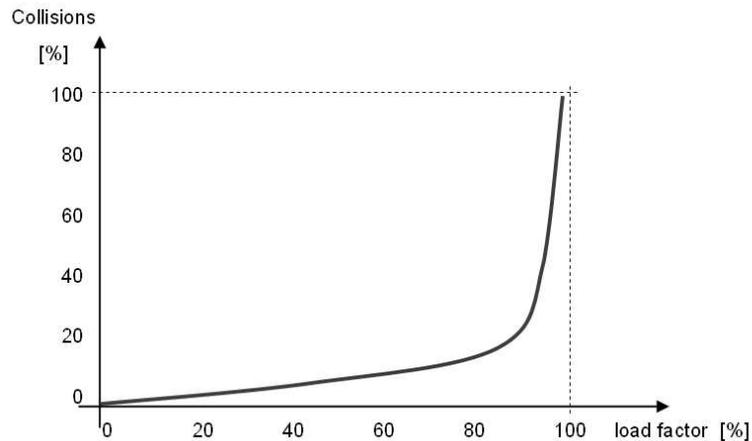


Figure 4.19 Hash collision percentage over the load factor

shows. But, even with very low fill grade the probability for at least one collision is nearly $1/2$.

We continue our previous example of employee records and store another employee, say with $SSN = 364 - 46 - 4738$ then the reader may verify that this SSN yields the same remainder when dividing by 200 000 as employee with $SSN = 364 - 26 - 4738$. In this case an alternative record address needs to be found. In our example we use a linear search checking the locations 64 739, 64 740, 64 741, ... and so on until we find an empty slot. We assume that the next empty slot is 64 740 as in Figure 4.18 where we store the employee's data with $SSN = 364 - 26 - 4738$. Now, how can we find this entry again? When we use our hash modulo formula the result will be address 64 738 where already the data of the person with $SSN = 364 - 26 - 4738$ is stored. In this case we proceed as before and check the next slots if the desired data is there. The search ends when either the data is found or we hit an empty slot. An empty slot obviously means that the searched person's data is not in the file because we proceeded in the same way as when we stored a record.

This example shows that when data is found at the calculated address we need to compare the key of the stored data with the key used for the hashing to make sure we have found the desired data.

4.9 Summary

Computer Science is ...

Exercises

- 4.1 Program a queue as ring buffer for 10 integer elements analog to the explanations and algorithms in Subsection 4.3.2. With each **add** or **remove** operation print out the state variable to show proper operation.

- 4.2 Design a hash algorithm that maps the nine-digit SSN onto 200000 buckets. In case of a collision use a linear probing for finding an empty bucket. Make sure, that you distinguish between values that are in the “right place” and values that were placed in a bucket because of a collision.
- 4.3 Construct a DAG (Directed Acyclic Graph) with 4 nodes that is not a tree.

Bibliographical Notes

Time-sharing systems were proposed first by Silberschatz et al. [2005].