

CS4618: Prerequisite Knowledge of Data Structures

1 Introduction

Suppose you have a collection of data, e.g. rainfall figures for last week. It is inconvenient to store each in a separate variable. Instead, you need a data structure: a systematic way of storing and manipulating the collection of data. Arrays are examples of data structures.

But the study of data structures separates the interface from the implementation. The interface refers to the set of operations that the data structure supports (e.g. insertion, deletion, etc.). The implementation refers to how the data is stored and how the operations in the interface are performed (the code, if you like).

Many data structures will share a common interface. In other words, they offer the same set of operations but implement them differently. These shared interfaces are called Abstract Data Types (ADTs). The rest of this document describes some common ADTs. These are all we need for CS4618; we need not concern ourselves with the implementations.

2 Set

Set is our first ADT. You know what a set is from CS1105: an unordered collection of objects, without duplicates. For example, \{a, b, c, d\} is a set and, because order does not matter, \{b, a, c, d\} is the same set. As described above, to define an ADT, we must give its interface: operations.

If \( S, S_1 \), and \( S_2 \) are sets, we expect operations such as: \(|S|\) (the cardinality or size of \( S \)); \( x \in S \) (true iff \( x \) is an element of set \( S \)); \( S_1 \cup S_2 \) (the union of \( S_1 \) and \( S_2 \)); \( S_1 \cap S_2 \) (their intersection); \( S_1 \setminus S_2 \) (the difference between \( S_1 \) and \( S_2 \)); and \( \mathcal{P}(S) \) (the powerset of \( S \)).

3 Pairs, Triples and \( n \)-tuples

A pair comprises two objects, where order matters. So, for example, \( \langle a, b \rangle \) is a pair and \( \langle b, a \rangle \) is a different pair. If \( x \) is a pair, then we access the individual objects using subscripts, \( x_1 \) and \( x_2 \).

A triple comprises three objects, where order matters, e.g. \( \langle ann, ben, dan \rangle \).

An \( n \)-tuple comprises \( n \) objects, where order matters.
4 Sequences or lists

A sequence is a collection of objects, arranged in some linear order, where duplicates are allowed. We might write them in square brackets. For example, here are three different lists: \([a, b, c]\), \([b, a, c]\) and \([a, b, c, a]\). Elements in the list are indexed by position, with positions starting from zero. So an \(n\)-element list has position \(0 \ldots n - 1\). For example, \(a\) is in position 0 of \([a, b, c]\).

Different treatments of this material will give different sets of operations. But we might expect something along the following lines:

- **get\((i, xs)\):** returns the object that is at position \(i\) in list \(xs\) (assumes \(0 \leq i < n\), where \(n\) is the length of the list). We will allow ourselves to also write this operation as \(xs[i]\).

- **insert\((x, i, xs)\):** inserts object \(x\) into position \(i\) in list \(xs\) (assumes \(0 \leq i < n\), where \(n\) is the length of the list). Objects in the list at positions greater than \(i\) are now at indexes that are one greater than previously.

- **delete\((i, xs)\):** deletes the object that is at position \(i\) in \(xs\) (assumes \(0 \leq i < n\), where \(n\) is the length of the list). Objects in the list at positions greater than \(i\) are now at indexes that are one less than previously.

- **replace\((i, x, xs)\):** replaces the object that is at position \(i\) by a new object, \(x\) (assumes \(0 \leq i < n\), where \(n\) is the length of the list).

The above is the ADT, the interface. Those of you who have studied data structures will realise that there are several ways to implement this ADT. One is to use the kind of arrays used in Java and C.\(^1\) With this implementation, the first and fourth operations are quite straightforward. But the second and third are not: they change the size of the array. Since changing the size of the array is not allowed in Java or C, our implementation of **insert\((x, i, xs)\)** would create a new larger array and then copy all elements from the original array plus the new element into the correct positions of the new larger array. **delete\((i, xs)\)** will be similar but copying to a new but smaller array.

An alternative implementation is a linked list (a node-and-pointer structure). In fact, there are choices here between using a singly-linked list (pointers in only one direction) or a doubly-linked list (pointers in both directions). It now becomes easier to insert and delete elements from the list: you follow the pointers until you find the position at which you wish to make changes and you adjust the pointers. But now the first and fourth operations are slower: they too require you to follow pointers in order to get the position you need.

This shows the kind of trade-off one makes when choosing concrete implementations. The nice thing is that their interface is the same. Even if, at some later

\(^1\)But not PHP. Confusingly, what PHP calls indexed arrays are really more like linked lists.
point, we switch implementations (e.g. from using arrays to using linked-lists), any client code that makes use of the data structure using only the operations in the interface will not need changing.

We won’t discuss implementations in quite this detail again. I mentioned them here to relate these notes to the data structures modules that you may or may not have studied previously. Our focus, however, remains on the ADTs.

5 Stacks

A stack is a collection of objects where insertion and deletion follow a last-in-first-out principle. In other words, when you delete from a stack, you always delete the most recently-inserted of its objects. If \( S \) is a stack, then the operations of its interface are:

- \( \text{push}(x, S) \): inserts object \( x \) onto the top of the stack.
- \( \text{pop}(S) \): deletes the object on the top of the stack (assumes the stack is not empty).
- \( \text{top}(S) \): returns the object on the top of the stack without deleting it (assumes the stack is not empty).

These are the three main operations, although you might also have \( \text{size}(S) \) (returns the size of the stack, i.e. how many objects it contains) and \( \text{isEmpty}(S) \) (returns true iff the stack contains no objects).

Stacks can be implemented using arrays and have a very efficient implementation using singly-linked lists.

6 Queues

A queue is a collection of objects where insertion and deletion follow a first-in-first-out principle. In other words, when you delete from a queue, you always delete the element that has been in the queue the longest. We think of insertion taking place at the rear of the queue and deletion taking place at the front — just like with real-world queues. The operations are:

- \( \text{insert}(x, Q) \): inserts object \( x \) at the rear of queue \( Q \). (This operation is sometimes called \( \text{enqueue} \).)
- \( \text{delete}(Q) \): deletes the object at the front of the queue (assumes the queue is not empty). (This operation is sometimes called \( \text{dequeue} \).)
Again you might also have \(\text{size}(Q)\) and \(\text{isEmpty}(Q)\).

Queues can be implemented using arrays but, like stacks, they have a very efficient implementation using singly-linked lists (however, to be really efficient, requires pointers to both front and rear of the queue).

## 7 Priority-ordered queues

A **priority-ordered queue** is a collection of objects, each associated with a priority (which we can think of as a number). When you delete from a priority-ordered queue, you always delete the element that has highest priority. To confuse matters, usually smaller numbers mean higher priority! The operations are as follows:

\[
\text{insert}(k, x, Q): \text{inserts object } x \text{ with priority } k \text{ into priority-ordered queue } Q.
\]

\[
\text{delete}(Q): \text{deletes the object with highest priority (smallest } k\text{) from } Q \text{ (assumes the queue is not empty).}
\]

Again you might also have \(\text{size}(Q)\) and \(\text{isEmpty}(Q)\).

Most implementations store the objects in order of priority (from smallest to largest). This makes finding the highest priority element straightforward: it’s the one at the front. But it makes insertion more complicated because the code must make sure that it inserts into the right place to preserve the ordering. Linked list implementations are more convenient and faster than ones that use arrays.

## 8 Dictionaries or maps

A **dictionary** (also called a **map**) is a collection of objects, each associated with a unique key. You can think of it as a collection of pairs: the first item in each pair is the key; the second is the object that is associated with that key. The keys are unique (no duplicates); the objects need not be. An example might be \(\{\langle CS\#4618, DB\rangle, \langle CS\#4619, DB\rangle, \langle CS\#614, SF\rangle\}\). In some treatments of this material, a distinction is drawn between unordered dictionaries and ordered dictionaries (where the keys are sorted in some way). We’ll just look at unordered dictionaries, in which case the operations might be as follows:

\[
\text{exists}(k, D): \text{returns true iff dictionary } D \text{ contains a pair whose key is } k.
\]

\[
\text{get}(k, D): \text{returns the object whose key is } k \text{ in } D \text{ (assumes } k \text{ is one of the keys in } D).}
\]
**insert**(*k, x, D*): inserts a pair comprising key *k* and object *x* into *D* (assumes *k* is not already one of the keys in *D*).

**delete**(*k, D*): deletes the pair whose key is *k* from *D* (assumes *k* is one of the keys in *D*).

Again you might also have **size**(*D*) and **isEmpty**(*D*).

There are numerous implementations of this ADT including arrays, linked lists, binary search trees, AVL trees, skip lists and hash tables.²

## 9 Trees

A *tree* is a collection of nodes with a hierarchical organization. With the exception of the *root* node, all nodes in the tree have a single parent node. All nodes have zero or more *children* nodes. It is normal for each node to be labelled with some object (maybe a string or some other data). There is lots of other terminology associated with trees, including: leaves (nodes with no children), parent, child, ancestor, descendant, sibling, and so on. There are special kinds of trees including ordered trees (where the ordering of children is significant) and unordered trees (where their ordering is not significant), and binary trees (where parents never have more than two children), and many others. Here we give some of the operations for an unordered tree:

**root**(*T*): returns the node that is the root of tree *T* (assumes *T* is not empty).

**parent**(*n, T*): returns the node that is the parent of node *n* in tree *T* (assumes *n* is not the root of *T*).

**children**(*n, T*): returns a sequence (list) of the nodes that are the children of node *n* in *T*. The list will be empty if *n* is a leaf in *T*.

**label**(*n, T*): returns the object that labels node *n* in *T*.

**createNode**(*x*): creates a node (with no children and no parent) whose label is *x*.

**insertChild**(*m, n, T*): inserts node *m* into the children of node *n* in *T* (assumes that the result is well-formed, e.g. it is not allowed for *m* to be a child of some other node already, or for *n* to be a child or other descendant of *m*, and so on).

We can imagine lots more operations including **size**(*D*), **isEmpty**(*D*), **isRoot**(*n, T*), and so on.

Trees are almost always implemented using node-and-pointer structures, although there are quite neat ways of flattening them for storage in arrays and lists.

²PHP’s associative arrays are, in fact, dictionaries or maps, implemented using linked lists.
10  Graphs

A graph is a set of nodes connected by edges.\textsuperscript{3} For CS4618, we consider only ordered graphs, in which the edges have a direction. Here is an example of a graph which has three nodes and three edges: \( G = \{\langle n_1, n_2, n_3 \rangle, \langle n_1, n_3 \rangle, \langle n_2, n_3 \rangle, \langle n_3, n_2 \rangle \} \). In this example, there are three nodes and there is an edge from \( n_1 \) to \( n_3 \), another from \( n_2 \) to \( n_3 \), and a third back from \( n_3 \) to \( n_2 \). In a weighted graph, the edges have costs (usually numbers) associated with them.

There are numerous operations that might define an ADT for unordered graphs, including:

- \( \texttt{numNodes}(G) \): returns the number of nodes in graph \( G \).
- \( \texttt{numEdges}(G) \): returns the number of edges in graph \( G \).
- \( \texttt{nodes}(G) \): returns the set of all nodes in \( G \).
- \( \texttt{edges}(G) \): returns the set of edges in \( G \).
- \( \texttt{inDegree}(n, G) \): returns the in-degree of node \( n \), i.e. the number of edges coming into \( n \).
- \( \texttt{outDegree}(n, G) \): returns the out-degree of node \( n \), i.e. the number of edges coming out of \( n \).
- \( \texttt{inEdges}(n, G) \): returns the set of edges that come into \( n \).
- \( \texttt{outEdges}(n, G) \): returns the set of edges that come out of \( n \).
- \( \texttt{origin}(e, G) \): returns the origin of directed edge \( e \).
- \( \texttt{destination}(e, G) \): returns the destination of directed edge \( e \).
- \( \texttt{insertNode}(x, G) \): inserts a new node into graph \( G \) and labels the node with object \( x \).
- \( \texttt{insertEdge}(n, n', c, G) \): inserts an edge as above but assigns cost \( c \) to the edge.

There are many more, but this suffices for our purposes.

There are several ways of implementing ordered graphs. One, called the adjacency matrix representation, is to use a two-dimensional array \( a \) of booleans: there is an edge from node \( n_i \) to node \( n_j \) iff \( a[i, j] \) is true. A weighted graph can be stored this way too: the matrix contains either null (where there isn’t an edge) or the cost (when there is an edge). Another implementation, called the adjacency list representation, is to store a list of the nodes and then, for each, you store a list of the destinations of each edge that comes out that node.

\textsuperscript{3}This notion of ‘graph’ is distinct from the plots that you drew on graph paper in school maths lessons. They are two essentially unrelated uses of the word ‘graph’.

6