Data Structures

In a rudimentary sense of a computer as Turing machine, it was imagined that the storage space for data consists of a (potentially) infinite tape with one unit of space for each "word". At the start, the tape contains the input (which is finite) and when it halts the tape contains the output. Hence, most of the tape consists of the "blank" symbol. The utilisation of the space resource by the machine can be interpreted as how much of the tape is actually utilised in the process of completing this computation. In other words, we not only examine the size of the input data and the output data, but also the data used in interim stages of the computation.

So far we have (largely) taken the viewpoint that our data is stored as a sequence (or tuple) of numbers of a *fixed* size and the operations that can be performed are basic arithmetic operations (within which we include the comparison between such numbers). In the process of making "new" tuples out of existing ones, we have not worried about the "cost" of making such partial copies. We must now think about this and try to optimise it.

As computers are put to more complex tasks, we wish to work at a "high level". This means we would like to *interpret* this storage as more complicated mathematical objects that plain numbers and perform more complex operations on it than basic arithmetic. In other words, we must work with data and algorithms in place of numbers and operations.

In order to analyse this usage of space (or storage) by an algorithm we must understand:

- How more complex data is to be stored.
- What basic algorithms need to be implemented.

The study of these two is intricately linked and is called the study of *data structures*.

This view of data structures as *defined by* the combination of stored values (or "objects") and algorithms for transforming them (or "methods" or "interfaces") is sometimes called "Object Oriented Programming". Another name you come across is "Abstract Data Types". In both these names, the emphasis is shifted from how the data structure is *implemented* to the ways we can *utilise it*, This helps us to conceptualise data mathematically and *reason* about it.

Imposing data structures on "flat" data stored in a "Turing tape" imposes restrictions on its use. While some would argue that this reduced flexibility in algorithms can reduce efficiency, others would argue that this removes the danger of unforeseen side-effects.

In the following discussion, we will discuss some important and useful data structures as follows. First we will present the mathematical notion, then the fundamental operations that we need to perform and finally, we will discuss various ways in which this data structure can be implemented.

Arrays

An array is a tuple of elements from a certain fixed set. Mathematically, this is an element $\underline{a} = (a_0, \ldots, a_{n-1})$ of A^n where A is a fixed set and n is a chosen positive integer called the *size* of the array. It may seem to be no different from what we have so far been calling a "tuple". However, an important difference is that we can *replace* any individual a_i by another value from the set A and this replacement can be done *without* making a copy of the whole array.

The operations available are:

- Extraction of elements $\underline{a}[i]$ which returns the element a_i of A when i is an index between 0 and $n-1$.
- Sub-arrays $\underline{a}[i:j]$ which returns the element $(a_i, \ldots, a_{j-1})$ of A^{j-i} when $0 \leq i < j \leq n-1$.
- Assignments of elements $\underline{a}[i] := b$ which modifies $\underline{a}$ to be

  $$(a_0, \ldots, a_{i-1}, b, a_{i+1}, \ldots . a_{n-1})$$

  when i is an index between 0 and $n-1$, and b is an element of A.
- Multi-assignment $\underline{a}[i:j] := \underline{b}$ which modifies $\underline{a}$ to be

  $$(a_0, \ldots, a_{i-1}, b_0, \ldots, b_{j-i-1}, a_j, \ldots, a_{n-1})$$

  where $0 \leq i < j \leq n-1$ and $\underline{b}$ is an element of A^{j-i}, i.e. an array.

Note that we can *modify* an array *in place* while tuples (at least in Python) are *immutable*.

Since A is a fixed set, its elements take up a fixed number α of words in memory. Thus, an array is usually represented by $n \cdot \alpha$ *continuous* words in memory. Note that this contiguity improves efficiency of computation in some types of computers which is why it is done. However, there is no *a priori* guarantee in the *definition* of an array (in most languages) that the memory locations will be contiguous. (An exception is the programming language `C` and its descendents where this is part of the specification.)

Note that by treating A^n as the base set B of an array type B^m, we can talk about multi-dimensional arrays $(A^n)^m$.

Since the algorithms required to implement an array are basic operations (such as copying and indexing) of a standard processor, we will not discuss them further.

Records

Records are also called `structs` in `C` and its descendents. Mathematically, we have a fixed finite collection of sets $A_1, \ldots, A_k$ and an element $\underline{a} = (a_1, \ldots, a_k)$ of $A_1 \times \cdots \times A_k$ is a *record* that is made out of the base types $A_1, \ldots, A_k$.

The operations available are:

- Extraction of elements $\underline{a}.i$ which returns the element a_i of A when i is an integer between 1 and k. Often, the *name* A_i of the i-th set is itself used to perform extraction so $\underline{a}.A_i$ is also a_i.
- Assignments of elements $\underline{a}.i := b$ which modifies $\underline{a}$ to be

  $$(a_1, \ldots, a_{i-1}, b, a_{i+1}, \ldots . a_k)$$

  when i is an index between 1 and k, and b is an element of A_i. The same effect can be achieved with $\underline{a}.A_i := b$.

There may appear to be some overlap between this notion and that of an array. However, in most cases, A_i are different sets, or at least have different interpretations. Secondly, while specifying a record type, each of the sets $A_1, \ldots, A_k$ often need to be *individually* spelt out.

Note that Python tuples can serve as a record type *except* that they are *immutable*.

A record is usually stored as a $\alpha_1 + \cdots + \alpha_k$ contiguous memory locations, where α_i are the storage sizes of A_i for $i = 1, \ldots, k$.

Since the operations above are basic read and copy operations from memory, we will not discuss the implementation further.

References

Reference data types are sometimes called "indirect" or "pointer" data types. The reference p is a label that can be used in the algorithm and it points to a location in memory where the actual data $*p$ of type A is stored. Typically, memory is thought of as an indexed array of words, in which case the reference stores the index at the start of the the collection of words that stores $*p$.

The basic operations are:

- De-referencing a pointer p by using $*p$ as an element of A.
- Extracting a label $\&a$ via an assignment like $p = \&a$ to create a pointer to data of type A.

Since this is a data type that allows us to talk about storage *within* the algorithm, there is a self-referential aspect to these types.

Secondly, it is *up to the implementation* of the algorithm to ensure that the data that is pointed to is *not* used in any way *except* the operations that apply to the data structure A.

Both of these points make pointer types somewhat tricky to deal with. The language Fortran does not (typically) use pointers. The language Python has a unique way of dealing with pointers — *all* variable names in Python are pointers! In other words, the labels that we use to implement the algorithm are treated as the labels that refer to memory locations. As a result, the pointer type is not required for most programming in Python. This comes at the cost that the label "does not know" what type of data it is pointing at! Python tries to suggest this

is an advantage by calling it "Dynamic typing". However, it can lead to some programming errors.

Lists

A list $\underline{a} = (a_0, \ldots, a_{n-1})$ is, as the name suggests, a list of data. There is, *a priori*, no requirement that a_i have the same data types or that n be some fixed integer. Often we can even allow the list to be *empty*. Secondly, a_i and a_j *can* refer to data that is *identical in storage*.

The basic operations are:

- Create an empty list $\underline{a} := []$.
- Get the length $\texttt{len}(\underline{a})$ of a list.
- Extract an element $\underline{a}[i]$ which returns the element a_i of A when i is an index between 0 and $n-1$.
- Insertion $\underline{a}.\texttt{insert}(i, b)$ of data b into the list at the i-th position for $0 < i \leq n-1$. This will make $\underline{a}$ into the *longer* list

$$(a_0, \ldots, a_{i-1}, b, a_i, \ldots, a_{n-1})$$

- Append the data b using $\underline{a}.\texttt{append}(b)$ to make $\underline{a}$ into the *longer* list

$$(a_0, \ldots, a_{n-1}, b)$$

- Delete the i-th element of the list using $\underline{a}.\texttt{delete}(i)$ to make $\underline{a}$ into the *shorter* list

$$(a_0, \ldots, a_{i-1}, a_{i+1}, \ldots, a_{n-1})$$

- Find the first occurrence of b in the list by $\underline{a}.\texttt{find}(b)$.

Given lists $\underline{a}$ and $\underline{b}$, implementations often provide $\underline{a}.\texttt{extend}(\underline{b})$ that makes $\underline{a}$ into the combined list

$$(a_0, \ldots, a_{n-1}, b_0, \ldots, b_{m-1})$$

Note that there is no provided way to insert at the *top* of a list! Of course, one can *create* a *new* list that contains a new 0-th element and then extend it by the earlier list.

One implementation of a list is as an array of a large size of (un-typed) pointers and an index that indicates where the list ends. For example, we have an array element $\underline{p}$ of R^{1024} where R is the set of *all* labels and an index $\underline{p}.\texttt{len}$ that is an integer between 0 and 1024 which indicates the end of the list. We think of the data pointed to by the labels at $0, \ldots, \underline{p}.\texttt{len} - 1$ as the elements of the list.

- Creation of an empty list amounts to creating a new array $\underline{p}$ and setting $\underline{p}.\texttt{len} = 0$.
- Extraction of an element is done in the obvious way.
- Insertion and deletion require *shifting* the entries of the array to the right and left respectively and changing the value of $\underline{p}.\texttt{len}$ appropriately.

- We can implement `find` by checking elements of the list one-by-one.

Note that this implementation of lists does not have restriction of insertion at the 0-th position. After all, we can right shift all the existing points and insert a pointer to new first element at the starting position.

However, note that this implementation has the limitation that the list has a bounded number of elements. One can modify this by always using the last entry $\underline{p}[1023]$ to point to another array, when the size exceeds 1023.

An alternate implementation of lists uses a pair-record for each entry. The first element of the pair is pointer to the element and the second element of the pair is a pointer to the next list entry (or a "mull pointer" to indicate that there are no more entries). The list is itself stored as a pointer to the first element. This is called a *linked list* implementation.

Stacks

A stack is a data structure that stores objects in a "last-in-first-out" way. Conceptually, this is like a physical stack of dishes where the last dish placed on the top of the stack is the first one we can take.

The main operations on a stack $\underline{s}$ are:

- Put an object a on the stack with `push`$(\underline{s}, a)$.
- Extract the object on the top of the stack with $a :=$ `pop`$(\underline{s})$. This also *deletes* the top of the stack.
- Peek at the object at the top of the stack with $a :=$ `peek`$(\underline{s})$. Note that this can be implemented as $a :=$ `pop`$(\underline{s})$ followed by `push`$(\underline{s}, a)$.
- Get the size `depth`$(\underline{s})$ of a stack.
- Check if the stack is empty: `isempty`$(\underline{s})$.

The last three operations are only available in *some* implementations of stacks.

A bounded stack is easily implemented using an array of references. A Linked list can be used to implement a stack which is not bounded. In terms of a list one can see `push` as `append` and `pop` in terms of `delete`.

Queues

A queue is a data structure that stores objects in a "first-in-first-out" way. Conceptually, this exactly like a queue of people waiting to get some service; the first one to arrive is processed while the others "wait in queue".

The main operations in a queue $\underline{q}$ are:

- Enter an object a in the queue with `enqueue`$(\underline{q}, a)$. The object becomes the end (or tail) of the queue.
- Process the object at the head of the queue by extracting it with $a :=$ `dequeue`$(\underline{q})$. This also deletes the head of the queue from it.

- Peek at the object at the head of the queue with $a :=$ `peek`($\underline{q}$). Note that this *cannot* be implemented with `dequeue` and `enqueue`, since the latter function would put the object at the queue.
- Get the size `depth`($\underline{q}$) of a queue.
- Check if the queue is empty: `isempty`($\underline{q}$).

A queue is easily implemented using an array of references. A double-linked list (where each entry has a pointer in both directions!) can be used to implement an unbounded queue. We will later also see another implementation using priority queues.

The array $\underline{q}$ of pointers that represents a bounded queue carries *two* indices $\underline{q}.head$ and $\underline{q}.tail$. We treat the array *circularly* by wrapping round to the front of the array to enqueue when we reach the end.

Trees

Mathematically, a tree is defined as connected graph without a cycle.

However, as a data structure is called a tree if it is a collection of nodes. There are three types of nodes: root node (which, if it exists, is unique), intermediate nodes and leaf nodes.

An intermediate node has a parent node and a collection of child nodes.

A root node note has no parent node it only has child nodes. A leaf node has no child nodes it only has a parent node.

Note that the parent node of a non-root node exists and is unique.

Moreover, a and b are distinct nodes then no child node of a can be a child node of b. (This, together with the uniqueness of the parent node ensures that there are no cycles.)

The (child) *valency* of a node is the number of of child nodes of that node. We will often work with *binary trees* where the every node has at most two child nodes.

A tree can be implemented by using a triple-record for each node. The first element of such a record is a pointer to the data stored in the node, the second element is a pointer to the parent node and the third element is a list of pointers to the child nodes. However, one should be careful since it may not be easy to *verify* the lack of cycles very easily while adding child nodes. (However, hashing functions discussed in the next section can help.)

A (bounded) rooted binary tree can be implemented as an array $\underline{t}$ of pointers. The convention is that $\underline{t}[0]$ is the root of the tree. The child nodes of the node represented by the i-th element of the array are at positions $2 \cdot i + 1$ and $2 \cdot i + 2$ in the array. Moreover, the parent of the node represented by the i-th element of the array of the node represented by the $\lfloor (i-1)/2 \rfloor$.

Sets

The most general notion of a mathematical object that can be represented as data is a (finite) set of data objects.

As such, we would think of this as an array of pointers to the data objects. However, we need to ensure that distinct elements are indeed distinct. Thus, while inserting an element, we need to ensure that the same object does not already exist in the set.

To do this, we need to have a relatively quick algorithm to calculate a "hash value" with the following properties:

- When the hash value of the data objects is different, the objects are definitely distinct.
- It is rare (in some precise sense) that distinct objects give the same hash value. (This is called a hash collision.)

The basic idea is that computations that work with data of size n *typically* produce $f(n)$ data objects of size $f(n)$ is bounded by a polynomial in n. On the other hand, *possible* hash values of size m are numbers between 0 and $2^m - 1$. Thus, if we choose m large enough, we have enough hash values to avoid collisions. We will examine some ideas to create hash functions in later lectures.

Note that data objects that can change ("mutable" objects) cannot be assigned a hash value. Thus, hash values are typically only used for constants, tuples of constants and other things like that.