V.1 Data structures

A data structures is a way of organizing and storing information on a computer. Modern computer languages often have a selection of in-built data structures, usually as a library. Here we will look at some of the most common structures and implement them, by hand, in c. We will see that different data structures are suited to different tasks. For example, so far we have been using arrays in which the elements are accessed by index, if you want the ith entry of a you write a[i] and so access is $O(1)$. However, adding an element to the middle of an array is hard since lots of elements have to be moved to make room. This is illustrated in Table 1, adding an entry is an $O(n)$ operation.

Often, in fact, you don't need to access elements by index; you might just need to go through them one-by-one in order but want to be able to add extra elements in different locations. For this you can use a linked list, which will be describe below: accessing a specified element is an $O(n)$ operation, but accessing each element one-by-one in turn, is also $O(n)$ and adding an extra element is $O(1)$.

Linked list

A linked list is a data structure in which each entry is stored in a node which also stores the location of the next node. There is no index, so it isn't possible to access the entry in the ith node directly, in the way you can access the ith entry in an array. However, since each node knows the location of the next one, you can go through the nodes in turn.

For example, lets use an arrow to illustrate one node knowing the location of the next, a linked list storing the entries $[3, 1, 9, 10, 4]$ would look like

$$3 \rightarrow 1 \rightarrow 9 \rightarrow 10 \rightarrow \times \tag{1}$$

where the last node points to nothing, here illustrated with a $\times$ and, in practice in c it is implemented as a NULL pointer. Now, assuming the location of the first node, often called the *head*, is known, it can be accessed to find its entry, 3, and the location of the next entry, indicated here by the arrow, this in turn can be accessed to give the entry, 1, and the location of the next node. This continues until the last node is identified by its NULL pointer.

A struct for a node in a linked list is give in Table 2. When making a linked list you need first of all a head, a node whose address you know. There is a function for making the head at Table 3. Now, after that the locations of the subsequent nodes are the business of the linked list, the person using the linked list only has to know the location of head, the location of the other nodes is stored in the nodes themselves.

0	1	2	3	4	5	6	7
a	b	c	e	f	g	h	
a	b	c	e	f	g		h
a	b	c	e	f		g	h
a	b	c	e		f	g	h
a	b	c		e	f	g	h
a	b	c	d	e	f	g	h

Table 1: A schematic intended to illustrate the difficulty of adding the entry d to position 3 of an array, e-h first have to be moved to make room.

```c
struct node
{
  int entry;
  struct node *next;
};
```

Table 2: A node. This is a node of a linked list for storing ints. It contains two variables, entry which stores the actual entry and next, which is a pointer to a node, this is the link.

```c
1 struct node * make_head(int new_entry)
2 {
3   struct node * head = (struct node *)malloc(sizeof(struct node));
4   head->entry=new_entry;
5   return head;
6 }
```

Table 3: A function for making the head. In line three a new node is created on the heap with a pointer to it called head. The entry for head is given the correct value, note that head is pointer, so its entry is head->entry.

```c
void print_list(struct node * iterator)
{
  while(iterator->next!=NULL)
    {
      printf("%d\n",iterator->entry);
      iterator=iterator->next;
    }

  printf("%d\n",iterator->entry);
}
```

Table 4: A function to print out the entries. Notice that when iterator->next=NULL it doesn't print out the entry in iterator, that's why there are two printf statements.

```c
struct node * locate(struct node * iterator, int target)
{
  while(iterator->next!=NULL)
    {
      if(iterator->entry==target)
        return iterator;
      iterator=iterator->next;
    }
  if(iterator->entry==target)
    return iterator;

  return NULL;
}
```

Table 5: A function to return the location of the node containing target. If target isn't found it returns NULL.

```c
void add_node(struct node * here, int new_entry)
{
  struct node * here_next=here->next;
  here->next = (struct node *)malloc(sizeof(struct node));
  here->next->next=here_next;
  here->next->entry=new_entry;
}
```

Table 6: Add a node after here. here_next stores the location of here->next, the new node is added at here->next and this new nodes next is set to here_next.

Now, to iterate through the list is simply a matter of moving along it. In Table 4 there is a function for printing out all the element. It starts at the head and then goes to head->next and keeps going until head->next==NULL. Another iterator is given in Table 5, this locates an entry and returns a pointer to its node.

Adding an entry is mostly a matter of moving around the links. Say we had the address of 9 in

$$3 \to 1 \to 9 \to 10 \to \times \tag{2}$$

and wanted to add a node containing 12 between 9 and 10. First we would store the address of 10 somewhere and add a new node at the end of the arrow from 9

$$3 \to 1 \to 9 \to \_(\to)10 \to \times \tag{3}$$

where the $\to$ in brackets is to indicate that the location of the node with 10 has been stored. Then, in the new node the entry is set equal to 12 and the link is set pointing to the node with 10, giving

$$3 \to 1 \to 9 \to 12 \to 10 \to \times \tag{4}$$

```c
void append_node(struct node * head, int new_entry)
{
  while(head->next!=NULL)
    head=head->next;
  head->next = (struct node *)malloc(sizeof(struct node));
  head->next->entry=new_entry;
}
```

Table 7: Append a node. This goes to the end of the list and adds the new node there.

Code to do this is in Table 6, in Table 7 is code for the related operation of adding a new node at the end. Deleting the node next to a node is done in a similar way, consider

$$3 \to 1 \to 9 \to 10 \to \times \tag{5}$$

and say you wanted to delete the node after the one which stores 1. First you store the link to the node holding 10

$$3 \to 1 \to 9(\to)10 \to \times \tag{6}$$

Then the node holding 9 is deleted

$$3 \to 1 \quad \not 9(\to)10 \to \times \tag{7}$$

and then you set the node with 1 to point to the one with 10

$$3 \to 1 \to 10 \to \times \tag{8}$$