Lists

Manoj Gupta

August 7, 2016

Assume that you are owner of a company that contains 5 employee. Each employee is represented by a tuple { name, salary}. A natural way to represent an employee is using a structure and five employees via an array. In course of time, new employees join the company or old employees leave the company. We should find a way to process these changes. Consider the case when the sixth employee joins the company. Since our initial array size was 5, we now need to allocate space for this sixth employee. A trivial way is then to construct a new array of size 6 and copy the first five employee from empRecord to this new array.

```
struct record {
                char name[100];
                int salary;
              };

struct record empRecord[5];
```

This process seems non-optimal as it involves creation of a new array. One needs to ask why arrays fail: it is because **the size of arrays cannot increase**. It is fixed at its definition. If the number of employee overshoots this size, then the only way to recover is to define a new array of bigger size.

It may seem that deleting an employee is not a problem as it does not increase the size of the array. However, if many employee leave the company, the size of the array is now more than the number of employees. Not only this is a wastage of space, it increases the time to search for employees. In all, we want a data-structure that has the following property:

The space taken by the data-structure should be proportional to the number of employees in the company.

To this end, we will tweak the above code.

```
struct record {
                char name[100];
                int salary;
                struct record* next;
              };

struct record *head = NULL;
```

Our new definition contains a pointer to the structure of the same type. One can construct a data-structure for 5 employees using the above definition as follows:

1. head points to the first employee record.

2. Each ith record $(1 \leq i \leq 4)$, the next pointer points to the $(i+1)$th record.

3. The next pointer of the fifth record points to $null$.

Pictorially the five employees are represented as follows:

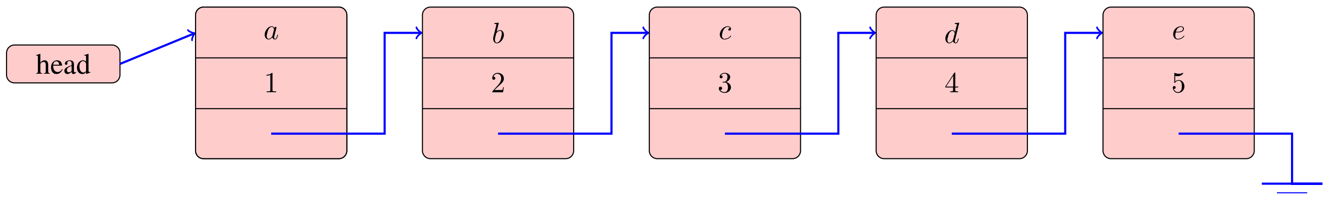

Figure 1: Pictorial representation of five employees

In the ensuing discussion, we will use the following syntax to represent the employee record. If e is an employee record, $e.name$ and $e.sal$ denotes the name and salary of the employee respectively. Also, $e.next$ either points to the next record or is $null$.

Consider the procedure ADDEMPLOYEE($name, salary$) that adds an employee at end of the list.

```
e ← allocate space for a new record;
e.name ← name ;
e.sal ← salary ;
e.next ← null ;
if head is null then
    /* The list is empty */
    head ← e ;
end
else
    /* The list is not empty */
    while head.next is not equal to null do
        head ← head.next ;
    end
    head.next ← e ;
end
```

Figure 2: ADDEMPLOYEE($name, salary$)

This procedure first allocate space, e, for a new employee and populates its name and salary. It's $next$ pointer is set to null as this employee will be the last employee in the list. If there are no employees in the list (**if condition**), then e is the first employee in the list. So, $head$ points to e. Else, $head$ already points to some employee record. So, we use the $next$ pointer to reach the last record in this list. Then, we set the $next$ pointer of this last record to point to e. Thus, e becomes the last record in this list.

We now describe a procedure DELETE($name$) which deletes the employee record e with $e.name = name$. If the list is empty then there is nothing to be done. Else, we check if the first record matches with $name$. If yes, then we let $head$ point to $head.next$ and then deallocate the space allocated to the first record. Else, the first record does not match with $name$. In that case, we move through the list using two pointers p and q with the property that $q.next = p$. If at any stage, we find that $p.name = name$, the $q.next$ is set to $p.next$. Then, the memory allocated to record p is deallocated.

One can check that the space taken by the list is proportional to the number of employees in the company. This was the main property that we wanted from our data-structure. As far as this property is concerned , the

```
if $head$ is null then
    /* The list is empty                                   */
    return;
end
else
    /* The list is not empty                               */
    if $head.name = name$ then
        /* The first record matches                        */
        $e \leftarrow head$;
        $head \leftarrow head.next$;
        deallocate the memory allocated to record $e$;
    end
    else
        $q \leftarrow head$;
        $p \leftarrow head.next$;
        while $p$ is not null do
            if $p.name = name$ then
                $q.next \leftarrow p.next$;
                deallocate the memory allocated to record $p$;
            end
            else
                $q \leftarrow p$;
                $p \leftarrow p.next$;
            end
        end
    end
end
```

Figure 3: DELETE($name$)

list data-structure is better than array. However, one needs to be aware of the disadvantages of lists. If we want to find the ith employee in the list, then we can use the following procedure FIND(i). This procedure moves to ith record using the $next$ pointers. So, the time taken to find the ith record is proportional to i. However, if we had implemented our data-structure as an array, then the ith record can be retrieved in constant time.

```
$count \leftarrow 1$;
while $head$ is not equal to $null$ and $count \leq i$ do
    $head \leftarrow head.next$;
    $count \leftarrow count + 1$;
end
return $head$
```

Figure 4: FIND(i)