22C:21 Lecture Notes

Jan 18th, 2006

Consider a program that manages the payroll of some organization. The three simplest operations performed by this program include (i) adding a new employee to the payroll, (ii) searching for an employee, given some information about her, and (iii) deleting an employee who has left the organization. To keep things simple, let us assume that each employee record consists of three fields: (i) name, (ii) social security number (ssn), and (iii) salary. Since ssn's are unique, the payroll program uses these to access employee records. The three operations mentioned above, can be stated more precisely as follows.

- `INSERT`(record). This operation inserts the given employee record into the collection of employee records.
- `DELETE`(ssn). This operation deletes the employee whose ssn is given, from the collection.
- `SEARCH`(ssn). This operation searches the collection for the employee whose ssn is given.

Note that at this point I have specified what operations I would like to perform on the collection of records, but I have not said anything about how the collection of records is actually stored in memory and neither have I said anything about how the above operations will be implemented. This kind of specification defines an *abstract data type* (ADT). Typically, an ADT can be implemented using one of several different data structures. A useful first step in deciding what data structure to use in a program is to specify an ADT for the program.

Now I consider two alternate data structures for the above ADT: (i) an unordered array of records and (ii) an ordered array of records, ordered by ssn. As you know well, items in an array can be accessed by providing an index. A key feature of an array (in any high level language) is that it takes the same amount of time to access any item in an array - whether it is the 10th item or the 1000th item. Furthermore, this access time is independent of the size of the array. Due to this feature, an array is said to provide *random access* to its elements.

Below I describe how each of the three operations can be implemented, first using an unordered array and then using an ordered array. I also discuss the running time of each operation.

Unordered array. I assume that the employee records are stored in an array, in no particular order. I also assume that there is a variable, say n, that keeps track of the number of employees currently on the payroll.

`INSERT` Simply take the record and put it in slot n of the array (I assume that the array is indexed $0, 1, 2, \ldots$) and increment n. This takes constant amount of time (independent of n).

`SEARCH` Since the array is not ordered in any particular way, this simply involves scanning through the entire array in some systematic way until the record is found or I am sure that the record does not exist. This takes time proportional to n, *in the worst case.* Of course, the record I am looking for might be the very first record that the algorithm examines, but in determining the running time of an algorithm, we are usually interested in "worst case" analysis.

`DELETE` The `DELETE` operation requires that we search the array first to find the record with the given ssn. Once the record is found, the algorithm simply replaces it by the last record (in slot $n-1$) and decrements n. Once the record has been found, it just takes a constant amount of extra time to delete it, but finding the record takes time proportional to n, in the worst case (as we see from above). Therefore, the `DELETE` operation also takes time proportional to n, in the worst case.

Ordered array. I assume that the employee records are stored in an array, in increasing order of ssn. Again, I assume that there is a variable, say n, that keeps track of the number of employees currently on the payroll.

`INSERT` Since the records are sorted, I first need to find where the given record should go in the given array. This can be done by scanning the array in increasing order of ssn until an index i is found such that the ssn of the record in slot i is smaller than the given ssn and the ssn of the record in slot $(i+1)$ is larger than the given ssn. Then the new record should be placed in slot $(i+1)$. But, before this is done all the records in slots $i+1, i+2, \ldots, n-1$ need to be moved down one slot. This operation takes time proportional to n.

`SEARCH` Since the array of records is sorted by ssn, I can do a *binary search* to find the given ssn. We will discuss this algorithm in the next class and analyze its running time next week. We will be able to show that in the worst case, the running time of binary search is proportional to $\log_2 n$. We will also discuss how as n grows, $\log_2 n$ grows extremely slowly relative to n.

`DELETE` The `DELETE` operation requires that we search the array first to find the record with the given ssn. This can be done in time proportional to $\log_2 n$, in the worst case, by using the `SEARCH` operation. Suppose the record we are looking for has been found in slot i. To delete this record, we need to move the records in slots $i+1, i+2, \ldots, n-1$ up by one slot each. This takes $n-i-1$ steps, which is proportional to n in the worst case. Therefore, the running time of the entire operation is $\log_2 n + n$, in the worst case. Later we will see that since n is so much larger than $\log_2 n$, especially for large values of n, this expression can be approximated by n.
