Data Structures
Spring 2009

Kishore Kothapalli

Chapter 3

The Array as a Data Structure

3.1 A Gentle Introduction

In this chaper, we study the notion of a data structure more formally. While the informal idea is that it is a way to organize data efficiently, a formal framework is essential to argue about these ideas. We think of a data structure as a collection of data along with methods/operations to access/operate on the data. This is called as an *abstract data type*, as an extended notion of a data type. An abstract data type is a set of objects together with a set of opertaions on the objects. The semantics of the operations are specified but the actual way the operations are implemented is not specified. There are no formal rules as to what are the right operations for a given abstract data type. This choice depends also on the application the designer of an abstract data type has in mind. Further, the user of the abstract data type is completely oblivious to the exact way the operations are implemented.

In this chapter, we will study the array as an abstract data type. While array can be thought of as a regular data type as it is commonly available in most high level programming languages, our interest in this chapter is to introduce the idea of abstract data types and also introduce efficient parallel operations on the array.

3.2 The Array Data Structure

In this framework, we now see the array as a data structure. The typical operations on this data structure are as follows:

- to create an array data structure, called `create(name, size)`,
- to access the element at a given index i, called `ElementAt(index, name)`,
- to find the size of the array, called `size(name)`, and
- to print the contents of the array, called `print(name)`.

While we are used to writing $A[i]$ to access element i of an array A, it can be seen as the way to access the ith element in a non-object oriented programming language. The other operations are also not explicity specified when one uses languages such as C. But, when one considers purely object oriented languages, such operations are required to be supported. Similarly, in C, array creation is done by `malloc()` or a compile time declaration as `int A[10];`. However, when one looks at a data structure in terms of its operations, a `create()` operation is required.

Thus, let us consider one possible implementation of the above operations in a C style programming language.

```
Algorithm Create(int size, string name)
begin
    name = malloc(size*sizeof(int));
end

Algorithm ElementAt(int index, string name)
begin
return name[i];
end

Algorithm Print(string name)
begin
    for i = 1 to n do
        printf("%d t", name[i]);
    end-for;
end;

Algorithm size(string name)
begin
return size;
end;
```

3.3 Further Operations

While the above operations let us access the elements of the array, we need further operations such as finding the minimum or maximum element in an array, sorting an array, and the like. The focus of this section is to find efficient algorithms for these operations.

3.3.1 Minimum and Maximum

The standard approach is to access each element of the array and keep track of the minimum (maximum). For finding the minimum, the following procedure can be used.

```
Algorithm FindMinimum(name)
begin
    int min,i;
    min = name[1];
    for i = 2 to n do
        if min > name[i] then
            min = name[i];
    end-for;
end;
```

The algorithm is self-explanatory. One can also notice that the algorithm makes $n - 1$ iterations of the loop and hence makes $n - 1$ comparisons. However, if one wants to find both the minimum and the maximum at the maximum at the same time, it is interesting to see that there is a slightly better algorithm.

We start by processing elements in pairs. For each pair, we compare them and see which is the larger and the smaller of the two. The larger is compared with the current maximum and the smaller with the current minimum. This approach requires us to make only $3n/2$ comparisons.

Algorithm MinimumMaximum(name)
begin
int max, min;
max = name[1], min = name[1];
for $i = 1$ to $n/2$ do
    element1 = name[$2i - 1$], element2 = name[$2i$];
    if element1 $<$ element2 then x = element1, y = element2;
    else x = element2, y = element1;
    if $x <$ min then min = x;
        if max $< y$ name[i] then max = y;
end-for
end;

We will see that the above algorithm makes only $3n/2$ comparisons where n is the size of the array.

3.4 Prefix

Consider any associative binary operator o and an array A of elements over which o is applicable. The prefix operation requires us to compute the array S so that $S[i] = A[1]oA[2] \cdots oA[i]$. The prefix operation is very easy to perform in the standard sequential setting.

However, the world is moving towards parallel computing. This is necessiated by the fact that the present sequential comptuers cannot meet the computing needs of the current applications. Already, parallel computers are available with the name of multi-core architectures. Programming and software has to wake up to this reality and have a rethink on the programming solutions. The parallel algorithms community has fortunately given a lot of parallel algorithm design techniques and also studied the limits of parallelizability.

One of the fundamental operations that is important to parallel computing is the prefix operation. The sequential solution of computing the previous index output $S[i-1]$ and then applying the o operator to $s[i-1]$ and $A[i]$ is not acceptable as it does not use any parallelism.

The parallel solution works as follows. Consider the array A and produce the array B of size $n/2$ where $B[i] = A[2i-1]oA[2i]$. Imagine that we recursively compute the prefix output wrt B and call the output array as C. Thus, $C[i] = B[1]oB[2]o \cdots B[i]$. Let us now build the array S using the array C. For this, notice that for even indices i, $C[i] = B[1]oB[2]o \cdots B[i] = A[1]oA[2]o \cdots A[2i]$, which is what $S[2i]$ is to be. Therefore, for even indices of S, we can simply use the values in array C. The above also suggests that for odd indices of S, we can apply the o operation to a value in C and a value in A. The complete pseudocode is given below.

Algorithm Prefix(A)
begin
    Phase I: Set up a recursive problem
    for $i = 1$ to $n/2$ do
        $B[i] = A[2i-1]oA[2i]$;
    end-for
    Phase II: Solve the recursive problem
    Solve Prefix(B) into C;

Phase III: Solve the original problem
for $i = 1$ to n do
if $i = 1$ then $S[1] = A[1]$;
else if i is even then $S[i] = C[i/2]$;
else if i is odd then $S[i] = C[(i-1)/2]oA[i]$;
end-for
end

Analysis

Phase I and Phase III take n operations. Phase II is recursive. So, the runtime of the algorithm can be expressed as $T(n) = T(n/2) + 2n$. Using master's theorem, the solution can be seen as $T(n) = O(n)$.

While the standard sequential algorithm also has an $O(n)$ runtime, the advantage of this algorithm is its suitability for parallel computation. The operations in phase I and Phase III can be done in parallel for each index.

3.5 Sorting

Sorting is a fundamental operation and we shall see two sorting algorithms in this section.

3.5.1 Merge Sort

The idea of merge sort is to shift the onus to the combine step. To this end, we come up with the following design.

- Divide the n–element input into two $n/2$–element sequences.
- Sort the $n/2$–element sequences recursively.
- Merge the two $n/2$–element sequences into a sorted n–element sequence.

The only non-straightforward operation in the above is the last step. So let us first concentrate on this step. The job is to merge two sorted sequences. For this, we can think of comparing elements of the two sorted sequences and iteratively constructively the required sorted array.

For simplicity let us assume that elements in arrays L and R each of size $\ell + 1$ and $r + 1$ respectively are to be merged into an array A of size $\ell + r$. The elements indexed $\ell + 1$ and $r + 1$ are set to ∞ in L and R respectively. The following performs the merge:

1. Procedure Merge(L, R, A) 1. $i = 1, j = 1$
2. for $k = 1$ to $\ell + r$ do
3. if $L[i] < R[j]$ then
4. $A[k] = L[i]; i = i + 1$
5. else
6. $A[k] = R[j]; j = j + 1;$
7. End

⟦***show with an example***⟧

3.5.2 Correctness Argument for Merge

We can argue that the algorithm Merge is correct by using the following loop invariant.

At the beginning of every iteration

- $L[i]$ and $R[j]$ are the smallest elements of L and R respectively that are not copied to A.
- $A[1..k-1]$ is in sorted order and contains the smallest $i-1$ and $j-1$ elements of L and R respectively.

Let us show that the above invariant is valid.

- Initialization At the start we have $i = j = 1$ and A is empty. So both the statements of the LI are valid.
- Maintenance Let us look at any typical iteration k. Let $L[i] \leq R[j]$. By induction, these are the smallest elements of L and R respectively and are not put into A. Since we add $L[i]$ to A at position k and do not advance j the two statements of the LI stay valid after the completion of this iteration.
- Termination At termination $k = \ell + r + 1$ and by the second statement we have that A contains $k - 1 = \ell + r$ elements of L and R in sorted order.

3.5.3 Performance of Merge Sort

It is easy to write the recurrence relation for merge sort as $T(n) = 2T(n/2) + O(n)$. This can be explained by the $O(n)$ time for merge and 2 subproblems obtained during the divide step each take $T(n/2)$ time. Solving this recurrence realtion is done using say the substitution method giving us $T(n) = O(n \log n)$.

3.5.4 Analyzing divide and conquer algorithms

This is where we can use recurrence realtions: in analyze divide and conquer based algorithms. The time taken by the algorithm on an input of size n will have three parts. One is the time taken to divide the problem into subproblems. Let us denote this by $D(n)$. Then, one has to also take into account the time taken to solve the subproblems. If there are p subproblems each of size $1/p$, then the time taken to solve all of them is $pT(n/p)$. Note that in a general setting it is not required that each subproblem has the same size nor do the total size of all the problems add up to n. Let us finally assume that combining the solutions takes $C(n)$ time. Then the required recurrence realtion is:

$$T(n) = D(n) + pT(n/p) + C(n)$$

To specify boundary conditions, we can say that solving a problem of a constant size c takes a constant amount of time.

Applying the above to the case of our merge sort, we have that $D(n) = O(1)$, $p = 2$ and $C(n) = O(n)$. Hence, the recurrence relation for merge sort is:

$$T(n) = 2T(n/2) + O(n)$$

To solve the above recurrence, we can use the substituion method and obtain that $T(n) = O(n \log n)$.

3.5.5 Quick Sort

Let us look at one final example of sorting algorithms along with a short proof of correctness. While merge sort can be said to be optimal in terms of its time requiement, it does use some extra space. So one question to prusue is to design a sorting algorithm that can sort in-place, i.e., without using any extra space.

C. A. R. Hoare gave an algorithm based on the divide and conquer strategy called the quick sort that can sort in place. The 3 steps of the algorithm in the framework of divide and conquer are:

- **Divide**: Divide the input into 3 parts L, E, and R where $L < E < R$ based on a pivot.
- **Conquer**: Solve the sorting problem recursively on L and R. Assuming that all the items are distinct, we have that $|E| = 1$ hence already sorted.
- **Combine**: Produce the sorted L, E, and R in that order.

One can notice that the third step is quite trivial here. In fact, this is an example of the so called partition based algorithms. Let us look at the only non-trivial step, that of partitioning the input.

The key to the partition step is to select a pivot and rearrange the elements of the array. For this the following approach is presented.

1. Procedure Parition(A, ℓ, h)
2. pivot = $A[h]$;
3. $i = \ell - 1$; 4. for $j = p$ to $h - 1$ do
5. if $A[j] \leq$ pivot
6. $i = i + 1$;
7. swap $A[i]$ with $A[j]$
8. swap $A[i + 1]$ with $A[h]$
9. End Procedure

⟦***Show an example***⟧

Let us try to understand why this procedure accomplishes the partition required. We will again use the loop invariant based proof. A loop invariant for the above procedure can be stated as follows observing the example.

At the beginning of each iteration for any array index k:

- If $\ell \leq k \leq i$ then $A[k] \leq$pivot.
- If $k = h$ then $A[k] =$pivot.
- If $i + 1 \leq k \leq j - 1$ then $A[k] \geq$pivot.

The following picture summarizes the invariant. There is a portion to the left of the array which contains items less than the pivot followed by a portion containing items more than the pivot. There is also a portion of elements that are not compared to the pivot yet. Any of these could be empty.

To show that above LI is correct we perform the 3 following steps.

- **Initialization**: The LI is true at the start of the loop where $i = \ell - 1$ and $j = \ell$. Since we chose the pivot as the element at index h, the second condition of the LI also holds.
- **Maintenance**: There are two cases to consider depending on whether $A[j] \leq$pivot or $A[j] >$pivot. In the latter, we simply increment j and it satisfies the LI. In the former, due to the swap operation performed, the LI holds true.

- **Termination**: The loop terminates at $j = h$. Applying the conditions of the LI at $j = h$, we get the requored partition of A into 3 portions.

Thus, the algorithm Partition is correct and hence quicksort too is correct.

3.5.6 Performance of Quicksort

To analyze the performance of quick sort, we can start by writing the following recurrence relation.

$$T(n) = T(|L|) + T(|R|) + O(n)$$

How do we estimate the size of L and R? One can quickly observe that L (or R) can have a very large size if the input and the choice of pivot is particularly bad. In that case, it is possible that $|L| = n - 1$ and $|R| = 0$ for every recursive call to procedure Partition. This suggests that the worst case time for quick sort is $O(n^2)$.

However, it was noticed by extensive experimentation that quick sort performs much better on average. For the notion of an average input we can assume that all permutations of elements are equally likely. Then it is likely that not all recursive calls to the procedure Partition give a skewed partition. This intuition can be formalized and can be used to argue that the average runtime of quick sort is $O(n \log n)$.

To this end, instead of picking the pivot to be the element at index h, one can choose a pivot uniformly at random from the array. Even this variant can be shown to give good performance.

What is the best case for quick sort? It is when the portions produced by the produced Partition have equal size at all times. Then the recurrence is $T(n) = 2T(n/2) + O(n)$ whose solution is $T(n) = O(n \log n)$.

3.6 Median and kth Smallest

The kth smallest element of an array is the element of the array that it is larger than exactly $k - 1$ other elements of the array. In other words, in the sorted list of elements of the array, the kth smallest has rank k. The median of a set of n elements is the $n/2$th smallest element.

3.6.1 Solution 1

Let us now consider how to find the kth smallest element of an array A. One solution would be to extend the minimum algorithm and let the first k indexed elements of the array to contain the kth smallest. Then every other element is compared to this set of k elements and the k smaller elements are retained. At the end, the maximum of these k items is the kth smallest of the array.

⟦***LAB problem: Implement this algorithm. Another variation would be to sort the*** k ***elements and binary search instead of linear search.***⟧

The pseudocode for this operations can be given as below. CHECK.

Algorithm FindKthSmallest(A, k)
begin
//Phase I : get the first K indexed elements into S;
for $i = 1$ to k do
    $S[i] = A[i]$;
end-for
//Phase II : Compare every other element with S;
for $i = k + 1$ to n do

```
    for j = 1 to k do
        if A[i] < S[j] then
            S[j] = A[k];
        end-for
    end-for
//Phase III : Find the maximum in S
end.
```

To analyze the algorithm notice that Phase I and Phase III are $O(k)$ time. Phase II requires $O(k)$ time in each iteration. So, Phase II takes $O(k(n-k))$ time. The total time taken is $O(k + k(n-k))$.

A Small Discussion The above runtime could be as high as $O(n^2)$ for $k = \Theta(n)$. For this reason, the above algorithm is not very efficient. For large values of k, it would instead be better if the input array is sorted and the kth ranked element in the sorted order can be reported. This runs in $O(n \log n)$, independent of k.

⟦***LAB: See which solution is better till what value of*** k**.**⟧

We will also see other approaches to solve this problem during the end of the semester. That solution works in $O(n)$ time, independent of k.

3.7 Advanced Topic: Streaming Model