CSCI 470: Priority Queues, Partition Procedure, Quick Sort

Vijay Chaudhary

September 18, 2023

Department of Electrical Engineering and Computer Science
Howard University

Overview

1. A quick update
2. Reviewing Substitution Method
3. Priority Queue
4. Partition
5. Quicksort

A quick update

Quick Update

- There was a typo in Homework 1, question 4
- Quiz 02 score has been released
- Any question from Homework 1 that I can quickly address?

Reviewing Substitution Method

Reviewing Substitution Method

- We make an inductive hypothesis.

Reviewing Substitution Method

- We make an inductive hypothesis.
- First, we have to show this inductive hypothesis holds for $m < n$

Reviewing Substitution Method

- We make an inductive hypothesis.
- First, we have to show this inductive hypothesis holds for $m < n$
- Then we do the "substitution" into the original equation.

Reviewing Substitution Method

- We make an inductive hypothesis.
- First, we have to show this inductive hypothesis holds for $m < n$
- Then we do the “substitution” into the original equation.
- This substitution must lead to the *exact form* of the inductive hypothesis expressed in terms of n. This step also involves fixing the value of c that supports the exact form of inductive hypothesis.

Example

Using substitution method, let's show that if $T(n) = 2T(\lfloor n/2 \rfloor) + n^2$ and $T(1) = 1$, then $T(n) = O(n^2)$

- What will be our guess to show $T(n) = O(n^2)$?

Example

Using substitution method, let's show that if $T(n) = 2T(\lfloor n/2 \rfloor) + n^2$ and $T(1) = 1$, then $T(n) = O(n^2)$

- What will be our guess to show $T(n) = O(n^2)$?
- What will be our inductive hypothesis here?

Example

Using substitution method, let's show that if $T(n) = 2T(\lfloor n/2 \rfloor) + n^2$ and $T(1) = 1$, then $T(n) = O(n^2)$

- What will be our guess to show $T(n) = O(n^2)$?
- What will be our inductive hypothesis here?
- Does the inductive hypothesis hold for the base case?

Example

Using substitution method, let's show that if $T(n) = 2T(\lfloor n/2 \rfloor) + n^2$ and $T(1) = 1$, then $T(n) = O(n^2)$

- What will be our guess to show $T(n) = O(n^2)$?
- What will be our inductive hypothesis here?
- Does the inductive hypothesis hold for the base case?
- Using the inductive hypothesis, we calculate $T(n)$ by substituting $T(m)$ for an appropriate value of $m < n$ that fits in the recurrence relation.

Example

Using substitution method, let's show that if $T(n) = 2T(\lfloor n/2 \rfloor) + n^2$ and $T(1) = 1$, then $T(n) = O(n^2)$

- What will be our guess to show $T(n) = O(n^2)$?
- What will be our inductive hypothesis here?
- Does the inductive hypothesis hold for the base case?
- Using the inductive hypothesis, we calculate $T(n)$ by substituting $T(m)$ for an appropriate value of $m < n$ that fits in the recurrence relation.
- As we derive, our goal is to find $T(n) \leq cn^2$

Example

Using substitution method, let's show that if $T(n) = 2T(\lfloor n/2 \rfloor) + n^2$ and $T(1) = 1$, then $T(n) = O(n^2)$

- What will be our guess to show $T(n) = O(n^2)$?
- What will be our inductive hypothesis here?
- Does the inductive hypothesis hold for the base case?
- Using the inductive hypothesis, we calculate $T(n)$ by substituting $T(m)$ for an appropriate value of $m < n$ that fits in the recurrence relation.
- As we derive, our goal is to find $T(n) \leq cn^2$
- We will have to specify the value of c that can satisfy $T \leq cn^2$, which is the exact form of inductive hypothesis expressed in terms of n.

Subtleties

- Can we always get to the ***exact form*** of the inductive hypothesis?

Subtleties

- Can we always get to the ***exact form*** of the inductive hypothesis?
- Even if we make the right guess, sometimes, we do not quite get to the exact form of the inductive hypothesis.

Subtleties

- Can we always get to the ***exact form*** of the inductive hypothesis?
- Even if we make the right guess, sometimes, we do not quite get to the exact form of the inductive hypothesis.
- Let's work with this recurrence $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1$, $T(1) = 1$ to show $T(n) = O(n)$

Subtleties

With our guess $T(n) \leq cn$

$$\begin{aligned} T(n) &\leq c(\lfloor n/2 \rfloor) + c(\lceil n/2 \rceil) + 1 \\ &= c(n/2 - 1) + c(n/2 + 1) + 1 \\ &= cn + 1 \end{aligned}$$

For any $c > 0$, $cn + 1 > cn$, therefore, this does not imply $T(n) \leq cn$ for any choice of c. :(

Subtleties

- Can we make a stronger inductive hypothesis?

$$
\begin{aligned}
T(n) &\leq c(\lfloor n/2 \rfloor - d) + c(\lceil n/2 \rceil - d) + 1 \\
&= c(n/2 - 1 - d) + c(n/2 + 1 - d) + 1 \\
&= cn - 2d + 1 \\
&\leq cn - d,
\end{aligned}
$$

for $d \geq 1$. This time, we ended up with the exact form of inductive hypothesis, which holds for some c that satisfies the boundary condition.

Subtleties

- Can we make a stronger inductive hypothesis?
- How about $T(n) \leq cn - d$ where $d \geq 0$?

$$
\begin{aligned}
T(n) &\leq c(\lfloor n/2 \rfloor - d) + c(\lceil n/2 \rceil - d) + 1 \\
&= c(n/2 - 1 - d) + c(n/2 + 1 - d) + 1 \\
&= cn - 2d + 1 \\
&\leq cn - d,
\end{aligned}
$$

for $d \geq 1$. This time, we ended up with the exact form of inductive hypothesis, which holds for some c that satisfies the boundary condition.

Boundary condition

We also need to check on the boundary condition such that choosing c large enough holds the base case(s).

Going back to the stronger inductive hypothesis

- What would be a stronger inductive hypothesis for a $T(n) = O(n \lg n)$, if $T(n) \leq cn \lg n$ does not suffice?

Going back to the stronger inductive hypothesis

- What would be a stronger inductive hypothesis for a $T(n) = O(n \lg n)$, if $T(n) \leq cn \lg n$ does not suffice?
- $T(n) \leq cn \lg n - d$ or $T(n) \leq c(n - d) \lg(n - d)$ where $d \geq 0$

Going back to the stronger inductive hypothesis

- What would be a stronger inductive hypothesis for a $T(n) = O(n \lg n)$, if $T(n) \leq cn \lg n$ does not suffice?
- $T(n) \leq cn \lg n - d$ or $T(n) \leq c(n-d) \lg(n-d)$ where $d \geq 0$
- Depending on the offset with the simpler guess, we can improve our guess with a stronger hypothesis which is enough to imply asymptotic notation we want to show.

Priority Queue

Priority Queue

Priority queue is a data structure for maintaining a set S of elements, each with an associated value called **key**.

A **max-priority** queue supports following operations:

- INSERT(S, x) insert the element x into the set S.

Priority Queue

Priority queue is a data structure for maintaining a set S of elements, each with an associated value called **key**.

A **max-priority** queue supports following operations:

- $\textsc{Insert}(S, x)$ insert the element x into the set S.
- $\textsc{Maximum}(S)$ returns the element of S with the largest key.

Priority Queue

Priority queue is a data structure for maintaining a set S of elements, each with an associated value called **key**.

A **max-priority** queue supports following operations:

- $\textsc{Insert}(S, x)$ insert the element x into the set S.
- $\textsc{Maximum}(S)$ returns the element of S with the largest key.
- $\textsc{Extract-Max}(S)$ removes and returns the element with the largest key.

Priority Queue

Priority queue is a data structure for maintaining a set S of elements, each with an associated value called **key**.

A **max-priority** queue supports following operations:

- $\textsc{Insert}(S, x)$ insert the element x into the set S.
- $\textsc{Maximum}(S)$ returns the element of S with the largest key.
- $\textsc{Extract-Max}(S)$ removes and returns the element with the largest key.
- $\textsc{Increase-Key}(S, x, k)$ increases the value of element x's key to the new value k, which is assumed to be at least as x's current key value.

One application of priority queue

We can use max-priority queues to schedule jobs on a shared computer. The max-priority queue keeps track of the jobs to be performed and their relative priorities. When a job is finished or interrupted, the scheduler selects the highest-priority job from among those pending by calling EXTRACT-MAX. The scheduler can add a new job to the queue at any time by calling INSERT.

Implementation of priority queue using heaps

HEAP-MAXIMUM(A)

return $A[1]$

HEAP-MAXIMUM implements the MAXIMUM operation in $\Theta(1)$ time.

Implementation of priority queue using heaps

HEAP-EXTRACT-MAX(A)

```
if A.heapsize < 1
    error "heap underflow"
max = A[1]
A[1] = A[A.heapsize]
A.heapsize = A.heapsize − 1
MAX-HEAPIFY(A, 1)
return max
```

The procedure HEAP-EXTRACT-MAX implements the EXTRACT-MAX operation, which is similar to body of **for** loop in HEAPSORT procedure.

Heap-Increase-Key

HEAP-INCREASE-KEY(A, i, key)

if $key < A[i]$
    error "new key is smaller than current key"
$A[i] = key$
while $i > 1$ and $A[\textsc{Parent}(i)] < A[i]$
    exchange $A[i]$ with $A[\textsc{Parent}(i)]$
    $i = \textsc{Parent}(i)$

The procedure HEAP-INCREASE-KEY implements INCREASE-KEY operation.

Runtime??

Heap-Increase-Key: Illustration

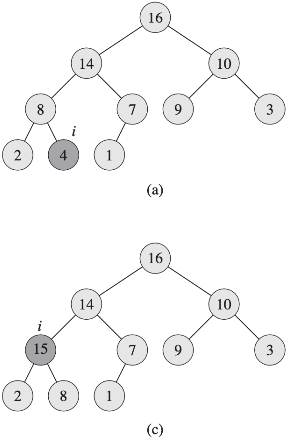


Figure 6.5 The operation of HEAP-INCREASE-KEY. **(a)** The max-heap of Figure 6.4(a) with a node whose index is i heavily shaded. **(b)** This node has its key increased to 15. **(c)** After one iteration of the **while** loop of lines 4–6, the node and its parent have exchanged keys, and the index i moves up to the parent. **(d)** The max-heap after one more iteration of the **while** loop. At this point, $A[\text{PARENT}(i)] \geq A[i]$. The max-heap property now holds and the procedure terminates.

Max-Heap-Insert

MAX-HEAP-INSERT(A, key)

$A.heapsize = A.heapsize - 1$
$A[A.heapsize] = -\infty$
HEAP-INCREASE-KEY$(A, A.heapsize, key)$

The procedure MAX-HEAP-INSERT implements the INSERT operation.

Runtime?

Partition

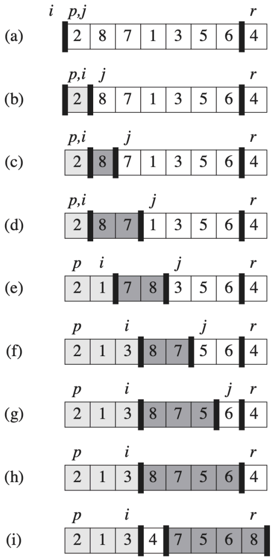
Quick-Sort

Before we discuss quick sort, we will go over PARTITION procedure, used in this algorithm.

Partition Procedure

```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

Partition Illustration



Classwork

- Using the illustration as a model, illustrate the operation of PARTITION on the array $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 21, 2, 6, 11 \rangle$.
- Give a brief argument that the running time of PARTITION on a subarray of size n is $\Theta(n)$.

Proof of Correctness

The following properties will be our loop invariant:

1. If $p \leq k \leq i$, then $A[k] \leq x$.
2. If $i + 1 \leq k \leq j - 1$, then $A[k] > x$.
3. If $k = r$, then $A[k] = x$.

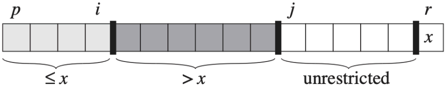


Figure 7.2 The four regions maintained by the procedure PARTITION on a subarray $A[p \,..\, r]$. The values in $A[p \,..\, i]$ are all less than or equal to x, the values in $A[i + 1 \,..\, j - 1]$ are all greater than x, and $A[r] = x$. The subarray $A[j \,..\, r - 1]$ can take on any values.

Initialization

Initialization: Prior to the first iteration of the loop, $i = p - 1$ and $j = p$. Because no values lie between p and i and no values lie between $i + 1$ and $j - 1$, the first two conditions of the loop invariant are trivially satisfied. The assignment in line 1 satisfies the third condition.

Maintenance

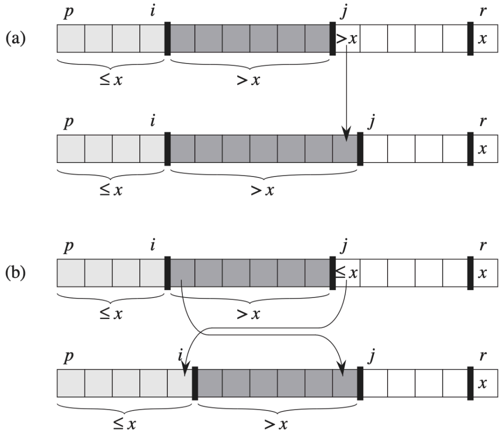


Figure 7.3 The two cases for one iteration of procedure PARTITION. **(a)** If $A[j] > x$, the only action is to increment j, which maintains the loop invariant. **(b)** If $A[j] \leq x$, index i is incremented, $A[i]$ and $A[j]$ are swapped, and then j is incremented. Again, the loop invariant is maintained.

Maintenance

Maintenance: As Figure 7.3 shows, we consider two cases, depending on the outcome of the test in line 4. Figure 7.3(a) shows what happens when $A[j] > x$; the only action in the loop is to increment j. After j is incremented, condition 2 holds for $A[j-1]$ and all other entries remain unchanged. Figure 7.3(b) shows what happens when $A[j] \leq x$; the loop increments i, swaps $A[i]$ and $A[j]$, and then increments j. Because of the swap, we now that $A[i] \leq x$, and condition 1 is satisfied. Similarly, we also have that $A[j-1] > x$, since the item that was swapped into $A[j-1]$ is, by the loop invariant, greater than x.

Termination

Termination: At termination, $j = r$. Therefore, every entry in the array is in one of the three sets described by the invariant, and we have partitioned the values in the array into three sets: those less than or equal to x, those greater than x, and a singleton set containing x.

Quicksort

Quicksort

Divide: Partition (rearrange) the array $A[p \mathinner{..} r]$ into two subarrays $A[p \mathinner{..} q-1]$ and $A[q+1 \mathinner{..} r]$ such that element of $A[p \mathinner{..} q-1]$ is less than or equal to $A[q]$, which is, in turn, less than or equal to each element of $A[q+1 \mathinner{..} r]$. Compute the index q as part of this partitioning procedure.

Quicksort

Conquer: Sort the two subarrays $A[p \mathinner{..} q-1]$ and $A[q+1 \mathinner{..} r]$ by recursive calls to quicksort.

Quicksort

Combine: Because the subarrays are already sorted, no work is needed to combine them: the entire array $A[p \mathinner{..} r]$ is now sorted.

Quicksort

$\textsc{Quicksort}(A, p, r)$

if $p < r$
    $q = \textsc{Partition}(A, p, r)$
    $\textsc{Quicksort}(A, p, q - 1)$
    $\textsc{Quicksort}(A, q + 1, r)$

To sort an entire array A, the initial call is $\textsc{Quicksort}(A, 1, A.length)$.