Quicksort

Note Title 9/19/2007

First let's review two things from last class.

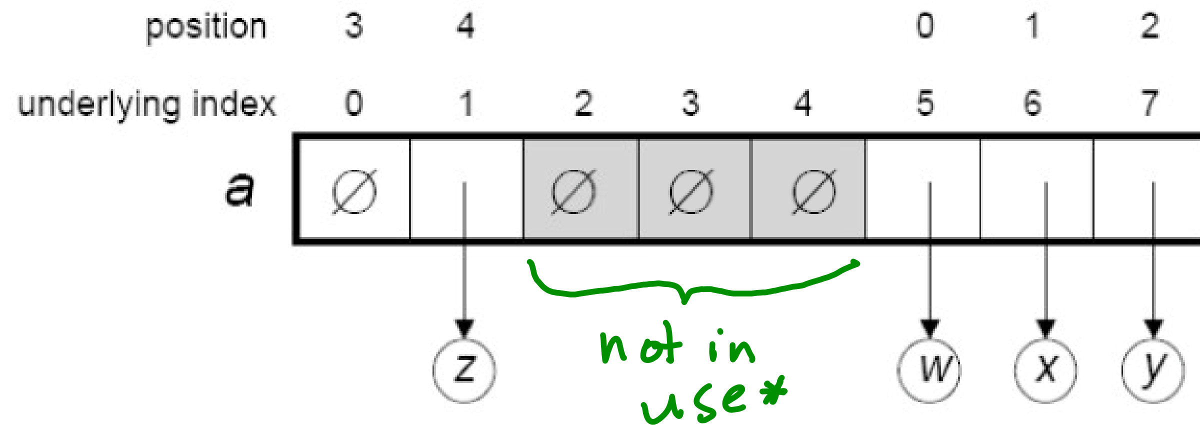


Figure 12.1

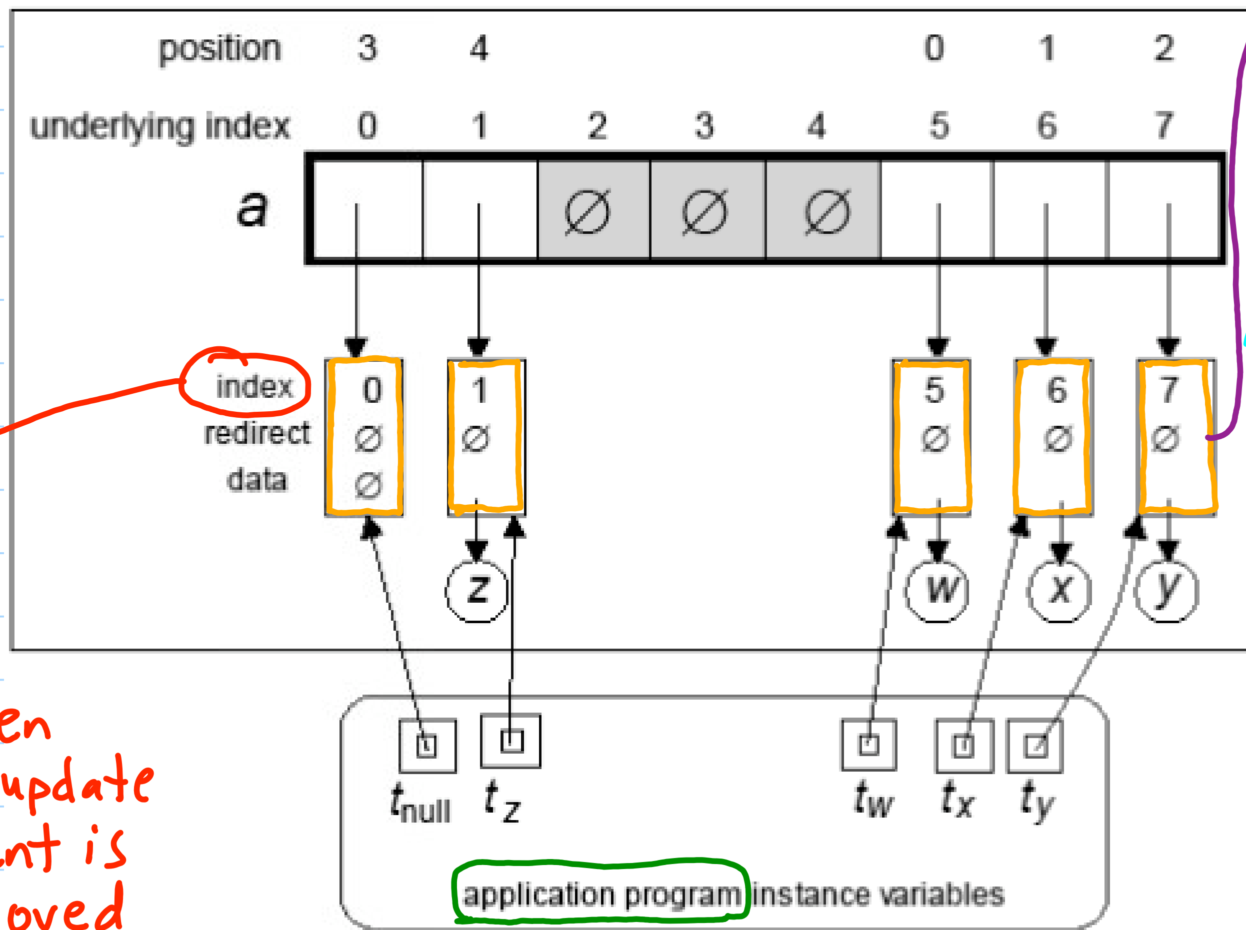
A circular array representing the positional collection $\langle w, x, y, null, z \rangle$ where *start*=5.

* implementation in text guarantees that array slots not in use are null.

I called this offset last class.

Tracked Array holding <w, x, y, ∅, z>

redirect used to navigate to next element in iteration order from a tracked element even when it has been deleted



underlying index (not position) stored. Then only need to update when element is physically moved

Sorting Algorithms

Seen insertion sort – good for nearly sorted data but worst-case $\Theta(n^2)$ time

Seen merge sort – worst-case $\Theta(n \log n)$ time $\left(T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)\right)$

Today we'll study quicksort

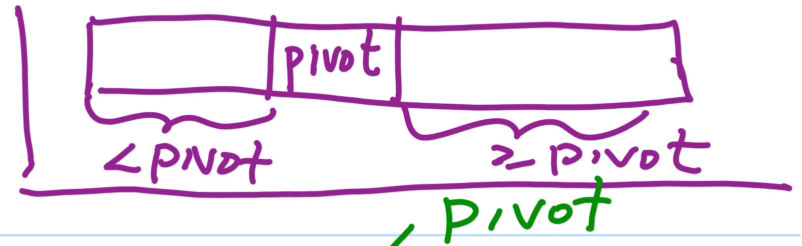
Quicksort

Divide-and-Conquer Alg

Do all the hard work in splitting
(+ recursive call). No combine.

Divide: Partition array into two subarrays where all elements in left portion are less than all elements in right portion

Example of Partition



pivot

… 11, 4, 9, 7, 3, 10, 2, 6, 8 …

i j

… 6, 4, 9, 7, 3, 10, 2, 11, 8

i j

≥ pivot

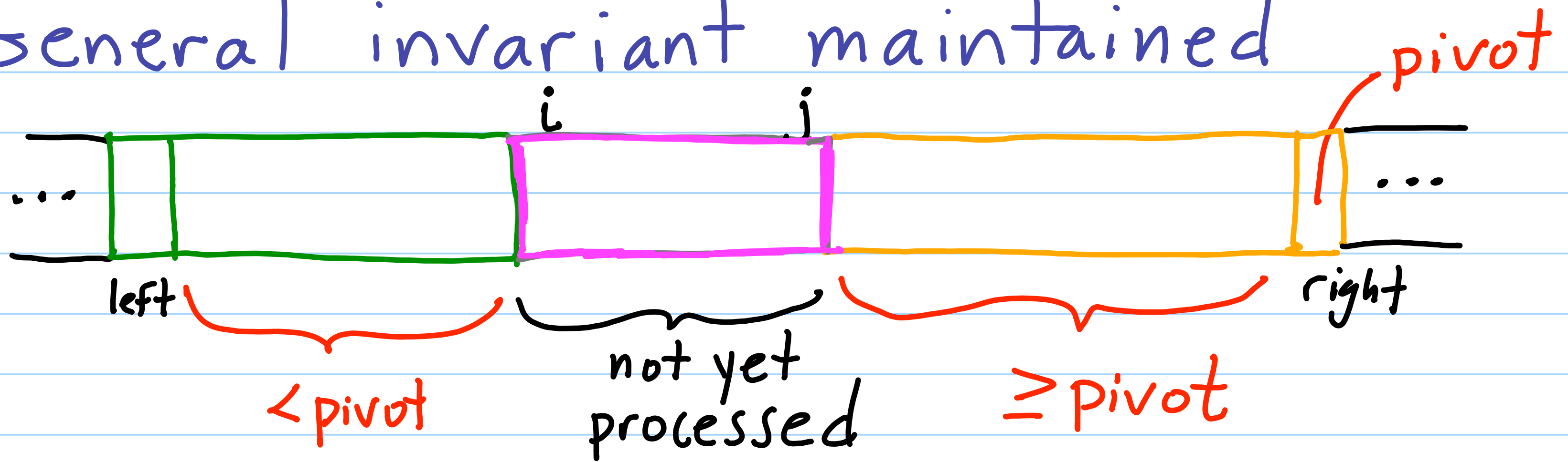
… 6, 4, 2, 7, 3, 10, 9, 11, 8

j i

… 6, 4, 2, 7, 3, 8, 9, 11, 10

i

General invariant maintained



Java Code For Partition

final position of pivot element

comparator that defines the sorted order

```
int partition(int left, int right, Comparator<? super E> sorter){
    E pivot = read(right);      a[right]         //pivot around the right element
    int i = left;                                 //positions left...i-1 hold elements < pivot
    int j = right;                                //positions j...right-1 hold elements >= pivot
    while (i < j){
        while (i < j && sorter.compare(read(i), pivot) < 0)   //pos. i element < pivot
            i++;
        while (j > i && sorter.compare(read(j), pivot) ≥ 0)   //pos i element ≥ pivot
            j--;
        if (i < j)
            swapImpl(i, j);   // swaps pos i and pos j elements
    }
    swapImpl(i, right);
    return i;
}
```

```
Quicksort(Comparator sorter) {
    quicksortImpl(0, n-1, sorter)
}

            int    int    comp.
quicksortImpl(left, right, sorter) {
    if (left<right) {
        swap what's in pos right using median-of-three or by random
        mid = partition(left, right, sorter);
        quicksortImpl(left, mid-1, sorter);
        quickSortImpl(mid+1, right, sorter);
    }
}
```

← method of positional collection interface

Time Complexity

Asymptotic time complexity for Partition is $\Theta(n)$

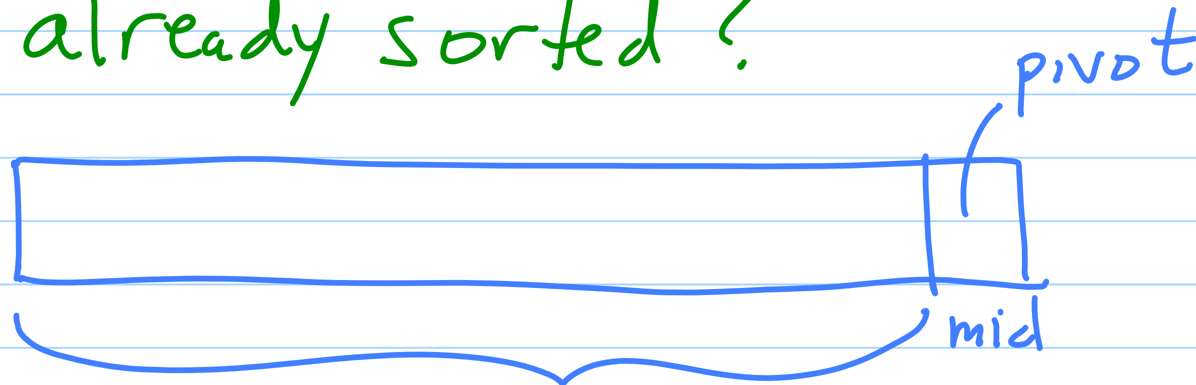
Best Case: Pivot is in middle (median)

$$T(n) = 2T\left(\frac{n-1}{2}\right) + \Theta(n) = \Theta(n \log n)$$

What's the worst-case?

What if pivot is min or max?

What if positional collection was already sorted?



Leads to recurrence

$$T(n) = T(n-1) + \Theta(n)$$

($\Theta(n) \leftarrow c \cdot n$)

n	cn
$n-1$	$c(n-1)$
$n-2$	$c(n-2)$
$\vdots$	
2	$c \cdot 2$
1	c

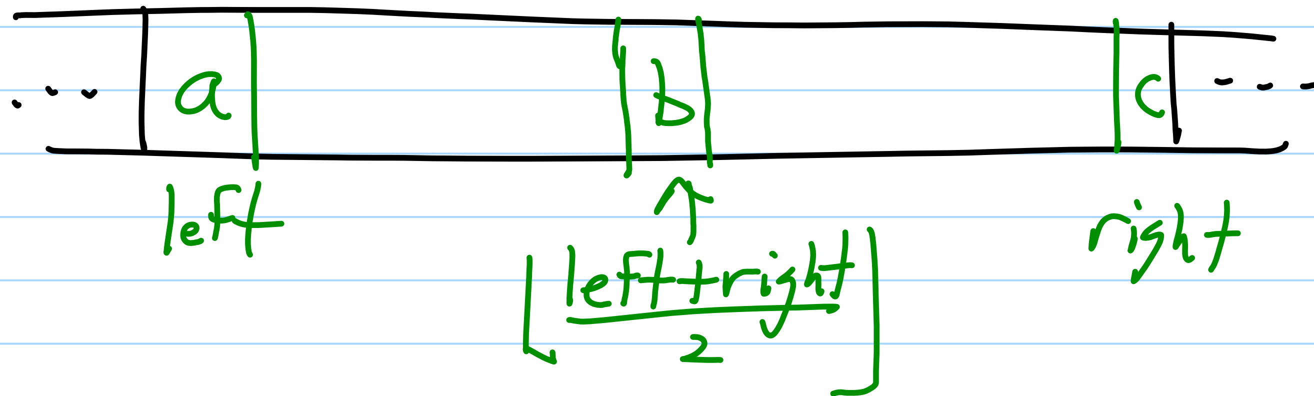
sum

$$c(1+2+\cdots+n)$$

$$\frac{c \cdot n(n+1)}{2} = \Theta(n^2)$$

How can we pick the partition to try to avoid bad behavior.

Median-of-three partitioning



Find median of a, b & c & then if this is not c, swap c with median

The other common solution is

Randomized Quicksort

Pick a random element (uniformly) from subarray + swap that with element in rightmost position of the subarray

if x items in subarray pick each with prob $1/x$

Show highlights that expected (average) time is $\Theta(n \log n)$

Expected Time Complexity

$T_A(x)$ time for alg A on input x

~~Expected Value~~

$$E[T_A(x)] = \sum_{\substack{\text{time } t \\ \text{alg can} \\ \text{take}}} t \cdot \text{Prob}[T_A(x) = t]$$

random var (X) → $T_A(x)$

Die with prob ½ of 1, 1/10 for each of 2, ..., 6

$$\frac{1}{2} \cdot 1 + \frac{1}{10} \cdot 2 + \cdots + \frac{1}{10} \cdot 6 = 2.5$$

Overview of Analysis

- Dominant cost is comparison performed by partition
- Only aspect that affects time complexity is # elements in subarrays for recursive calls (at each level of recursion)