Sorting

Internal sorting

Sorting

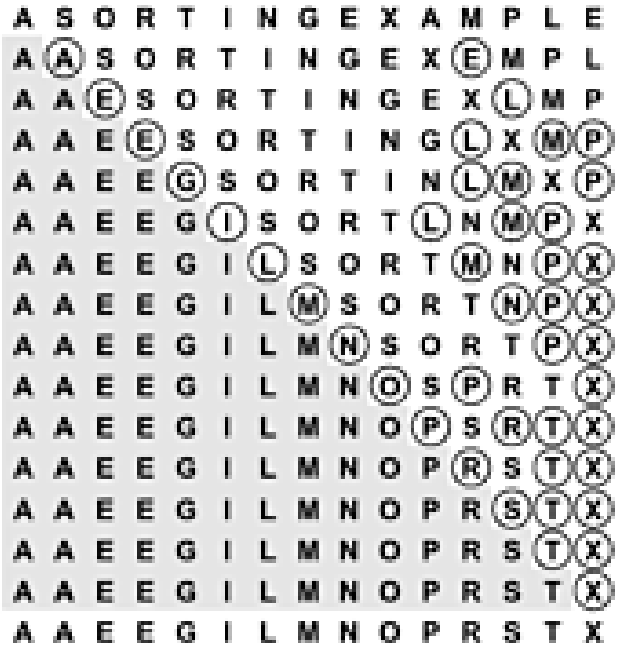
- We assume that objects to be sorted are records consisting of one or more fields. One of the fields, called the *key* is of a type for which a linear ordering relationship '<' is defined.
- The sorting problem: to arrange a sequence of records so that the values of their key fields form a nondecreasing sequence, i.e. for records $r_1, r_2, \ldots, r_n$ with key values $k_1, k_2, \ldots, k_n$ respectively, we must produce the same records in an order $r_{i_1}, r_{i_2}, \ldots, r_{i_n}$ such that $k_{i_1} \leq k_{i_2} \leq \ldots \leq k_{i_n}$

Sorting

- Criteria used to evaluate the running time of an internal sorting algorithm are:
  - The number of steps required to sort n records;
  - The number of comparisons between keys needed to sort n records (if the comparison is expensive);
  - The number of times the records must be moved
- Simple sorting schemes worth our attention because they:
  - Are easy to code
  - Are fastest for small input
  - Form a context for developing ground rules
  - Are fastest in some special situations

Bubblesort

```
void bubble(ITEM[] a, int l, int r)
{
   for (int i = l; i < r; i++)
      for (int j = r; j > i; j--)
         compExch(a, j-1, j);
}
```



For each i from l to r-1, the inner (j) loop puts the minimum element among the elements in a[i], ..., a[r] into a[i] by passing from right to left through the elements, compare–exchanging successive elements. The smallest one moves on all such comparisons, so it "bubbles" to the beginning.

Complexity of Bubblesort

- For an array of size N, in the worst case:
  - 1^{st} passage through the inner loop: $N-1$ comparisons and $N-1$ swaps
  - ...
  - $(N-1)^{st}$ passage through the inner loop: 1 comparison and 1 swap
  - All together: $c\ ((N\text{-}1) + (N\text{-}2) + \ldots + 1) + k$

where c is the time required to do one comparison and one swap

Complexity of Bubblesort

$c\,((N-1)+(N-2)+\ldots+1)+k$

$(N-1)+(N-2)+\ldots+1$

$+$

$1+2+\ldots+(N-1)=$

$=N+N+\ldots+N=N\times(N-1)$

- so our function equals

  $c\,N\times(N-1)/2+k=1/2c\,(N^2-N)+k$

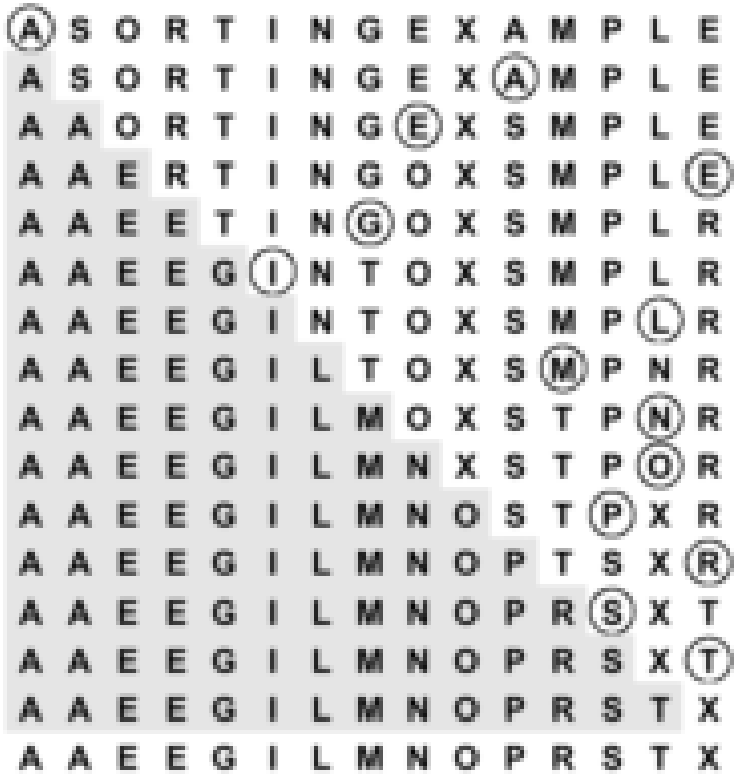
- complexity $O(N^2)$.

  $1/2c\,(N^2-N)+k\leq KN^2$

- for which values of K and N is this inequality true?
- For example, $K=c+k$ and $N>0$ (provided N can only take integer values).

Selection Sort

```
void selection(ITEM[] a, int l, int r)
{
  for (int i = l; i < r; i++)
  {
     int min = i;
     for (int j = i+1; j <= r; j++)
              if  (less(a[j], a[min]))
          min = j;
     exch(a, i, min);
  }
}
```



For each i from l to r-1, exchange a[i] with the minimum element in a[i], ..., a[r]. As the index i travels from left to right, the elements to its left are in their final position in the array (and will not be touched again), so the array is fully sorted when i reaches the right end.

Complexity of Selection Sort

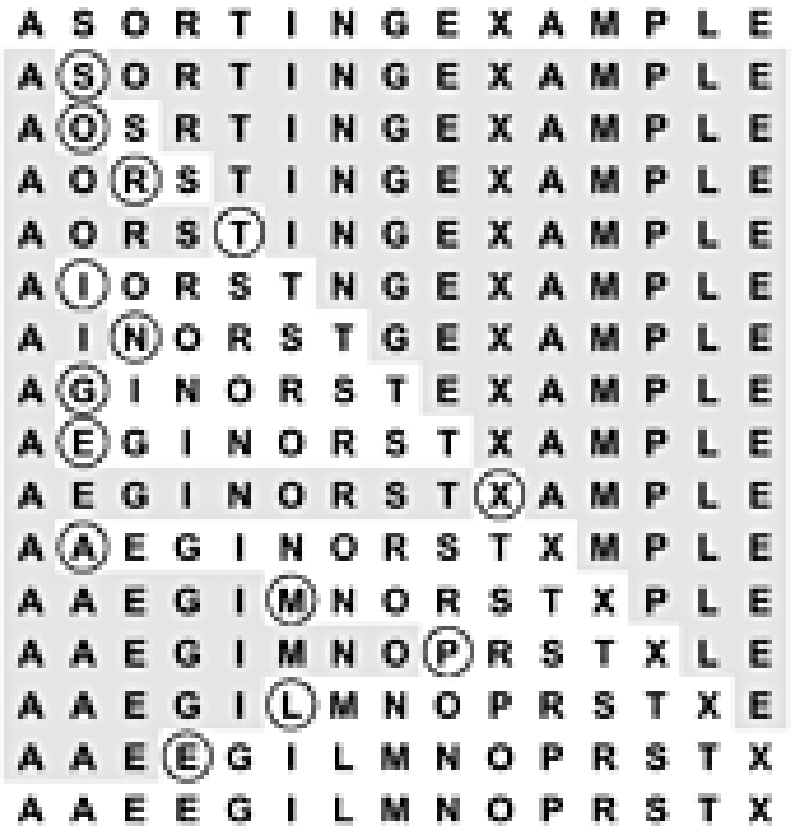
- Complexity of selection sort:
  - Same number of iterations as for bubblesort in the worst case
  - for some different constant c'

    $$c'(N^2 - N) + k$$

  - also $O(N^2)$

Insertion Sort

```
void insertion(ITEM[] a, int l, int r)
{
  int i;
  for (i = r; i > l; i--)
     compExch(a, i-1, i);
  for (i = l+2; i <= r; i++)
    {
      int j = i; ITEM v = a[i];
      while (less(v, a[j-1]))
        {
          a[j] = a[j-1];
          j--;
        }
      a[j] = v;
    }
}
```



For each i, it sorts the elements a[l], ..., a[i] by moving one position to the right elements in the sorted list a[l], ..., a[i-1] that are larger than a[i], then putting a[i] into its proper position.

Complexity of Insertion Sort

- Worst case: has to make $N \times (N - 1)/2$ comparisons and shifts to the right
  - $O(N^2)$ worst case complexity
- Best case: array already sorted, no shifts.

Shellsort

- Insertion sort
  - slow because the only exchanges it does involve adjacent items
  - items can move through the array only one place at a time
- Shellsort
  - simple extension of insertion sort that gains speed by allowing exchanges of elements that are far apart.

Shellsort Example

insertion sorting the subfile at positions 0, 4, 8, 12

insertion sorting the subfile at positions 1, 5, 9, 13

insertion sorting the subfile at positions 2, 6, 10, 14

insertion sorting the subfile at positions 3, 7, 11

we can achieve the same result by inserting each element into position into its subfile, going back four at a time

Shellsort

```
void shell(ITEM[] a, int l, int r)
{
  int h;
  for (h = 1; h <= (r-l)/9; h = 3*h+1);
  for ( ; h > 0; h /= 3)
    for (int i = l+h; i <= r; i++)
    {
      int j = i;
      ITEM v = a[i];
      while (j >= l+h && less(v, a[j-h]))
      {
        a[j] = a[j-h];
        j -= h;
      }
      a[j] = v;
    }
}
```

The increment sequence 1 4 13 40 121 364 1093 3280 9841 ... with a ratio between increments of about one-third, was recommended by Knuth in 1969

Mergesort

Divide-and-conquer.

To sort $A[p \,..\, r]$:

1. **Divide Step**

   - If a given array A has zero or one element, simply return; it is already sorted. Otherwise, split $A[p \,..\, r]$ into two sub-arrays $A[p \,..\, q]$ and $A[q + 1 \,..\, r]$, each containing about half of the elements of $A[p \,..\, r]$. That is, q is the halfway point of $A[p \,..\, r]$.

Mergesort

2. **Conquer Step**
   - Conquer by recursively sorting the two sub-arrays $A[p \,..\, q]$ and $A[q + 1 \,..\, r]$.
3. **Combine Step**
   - Combine the elements back in $A[p \,..\, r]$ by merging the two sorted subarrays $A[p \,..\, q]$ and $A[q + 1 \,..\, r]$ into a sorted sequence. To accomplish this step, we will define a procedure MERGE (A, p, q, r).

Note that the recursion bottoms out when the sub-array has just one element, so that it is trivially sorted.

Merge Sort Algorithm

```
MERGE-SORT(A, p, r)
1  if p < r
2     then q ← ⌊(p + r)/2⌋
3          MERGE-SORT(A, p, q)
4          MERGE-SORT(A, q + 1, r)
5          MERGE(A, p, q, r)
```

```
MERGE(A, p, q, r)
 1  n1 ← q − p + 1
 2  n2 ← r − q
 3  create arrays L[1 .. n1 + 1] and R[1 .. n2 + 1]
 4  for i ← 1 to n1
 5       do L[i] ← A[p + i − 1]
 6  for j ← 1 to n2
 7       do R[j] ← A[q + j]
 8  L[n1 + 1] ← ∞
 9  R[n2 + 1] ← ∞
10  i ← 1
11  j ← 1
12  for k ← p to r
13       do if L[i] ≤ R[j]
14            then A[k] ← L[i]
15                 i ← i + 1
16            else A[k] ← R[j]
17                 j ← j + 1
```

Heapsort

- A sorting algorithm sorts in place (in situ) if additional space for only a constant number of elements is needed.
  - Typically the data to be sorted is a record containing a key, which determines the sort order, and further satellite data.
  - A sorting algorithm is called ***stable*** if the order of elements with equal keys is preserved. Stability is only relevant if satellite data exists. In practice, this is typically the case. For our purposes, we assume that the elements consist only of the keys.
  - Heapsort sorts *in place*, is *not stable*, and has a running time of $O(n \lg n)$ in the average and worst case.

Heapsort

- n insert operations for a heap result in $O(n \log n)$ time
- Can we build a heap for n given elements faster? ($O(n)$)
- Insert elements from a given sequence S in a priority queue/heap
- After this process the sequence is empty
- Remove the *minimum* and insert element in S
- Repeat this step until every element is inserted back in S

Heapsort – heap in array

```
boolean less(int i, int j) {
  return pq[i].less(pq[j]);
}
void exch(int i, int j) {
  ITEM t = pq[i]; pq[i] = pq[j]; pq[j] = t;
}
private void swim(int k) {
// bottom up heapify
  while (k > 1 && less(k/2, k))
    { exch(k, k/2); k = k/2; }
}
private void sink(int k, int N) {
// top down heapify
  while (2*k <= N) {
    int j = 2*k;
    if (j < N && less(j, j+1)) j++;
    if (!less(k, j)) break;
      exch(k, j); k = j;
  }
}
```

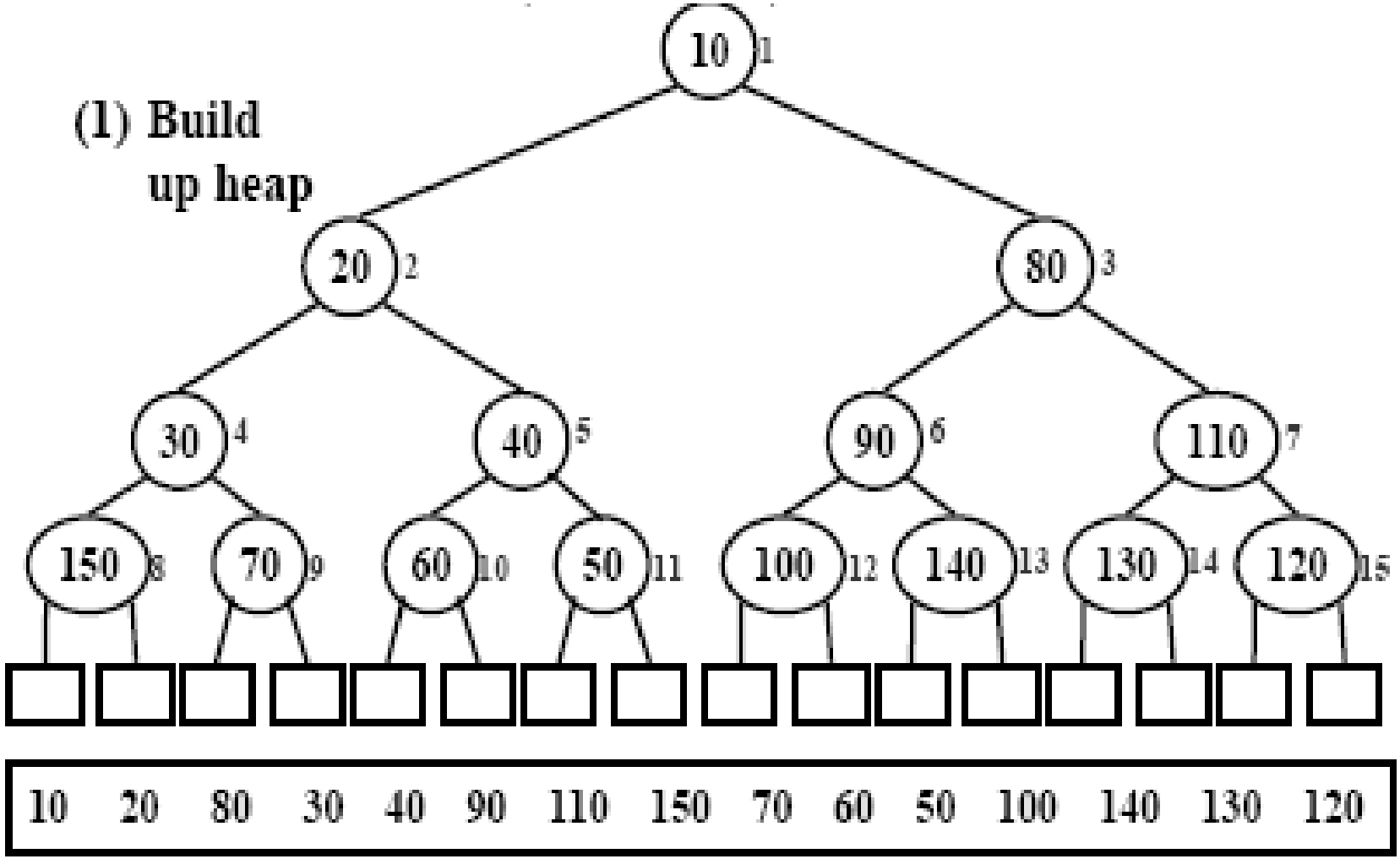
```
ITEM[] pq;
int N;
void makePQ(int maxN) {
  pq = new ITEM[maxN+1];
  N = 0;
}
boolean isEmpty() {
  return N == 0;
}
void insert(ITEM v) {
  pq[++N] = v; swim(N);
}
ITEM getmax() {
   exch(1, N);
   sink(1, N-1);
   return pq[N--];
}
```

Heapsort

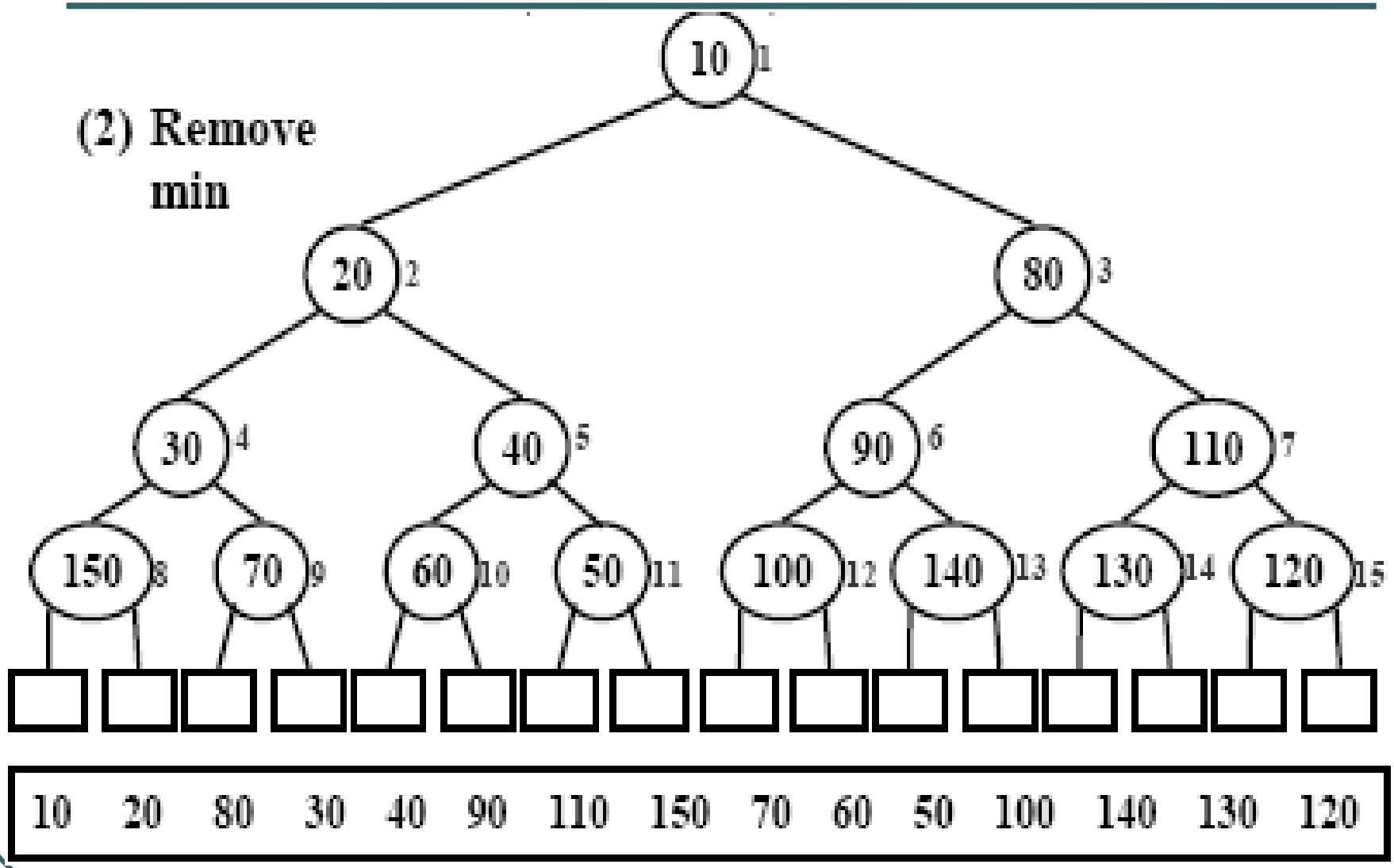
```
void sort(ITEM[] a, int l, int r) {
   PQsort(a, l, r);
}
void PQsort(ITEM[] a, int l, int r) {
  int k;
  PQ pq = makePQ(r-l+1);
  for (k = l; k <= r; k++)
      pq.insert(a[k]);
  for (k = r; k >= l; k--)
      a[k] = pq.getmax();
}
```

Heapsort Example

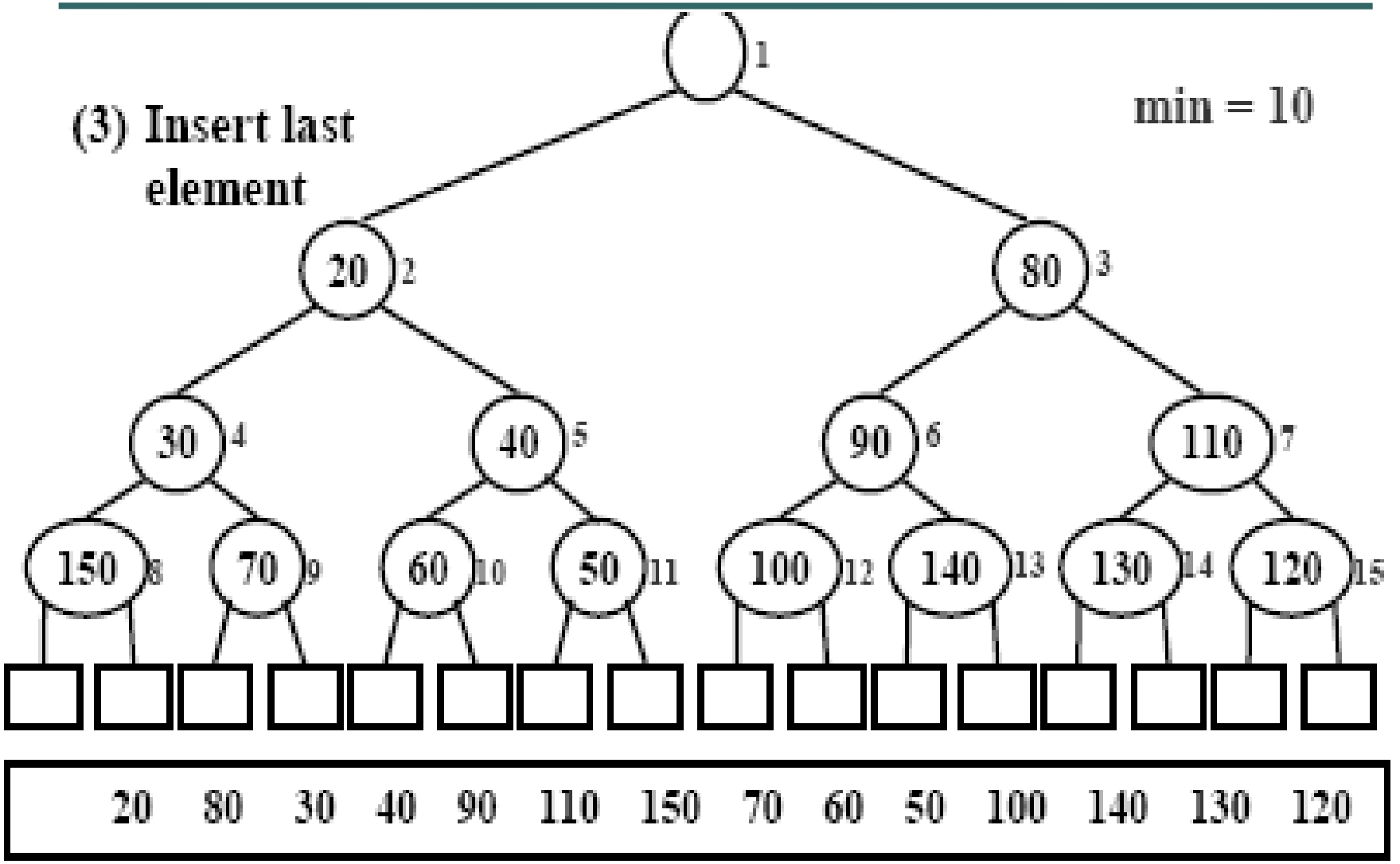
Data set: 150 70 30 10 20 60 40 50 140 100 80 90 130 110



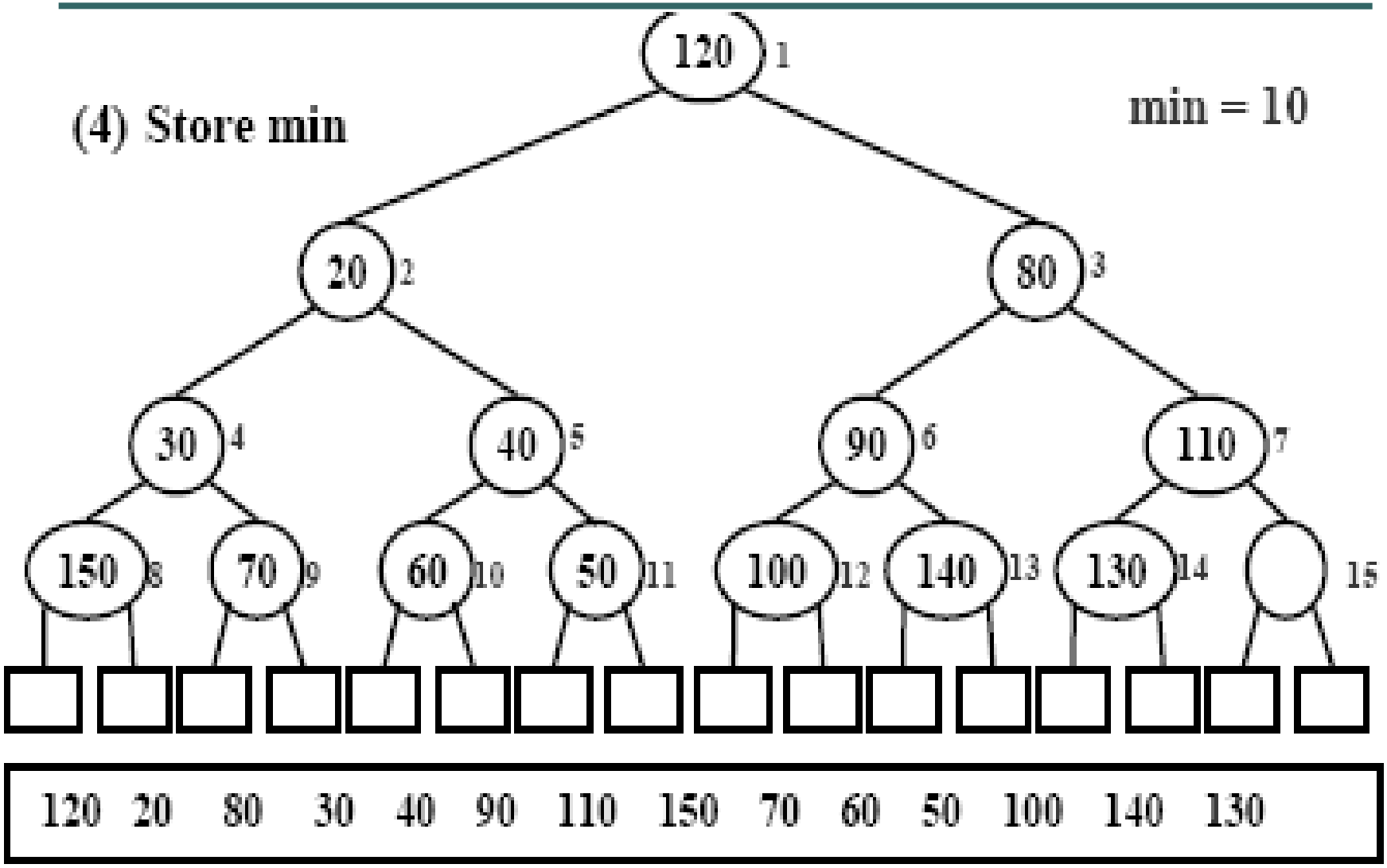
Heapsort Example



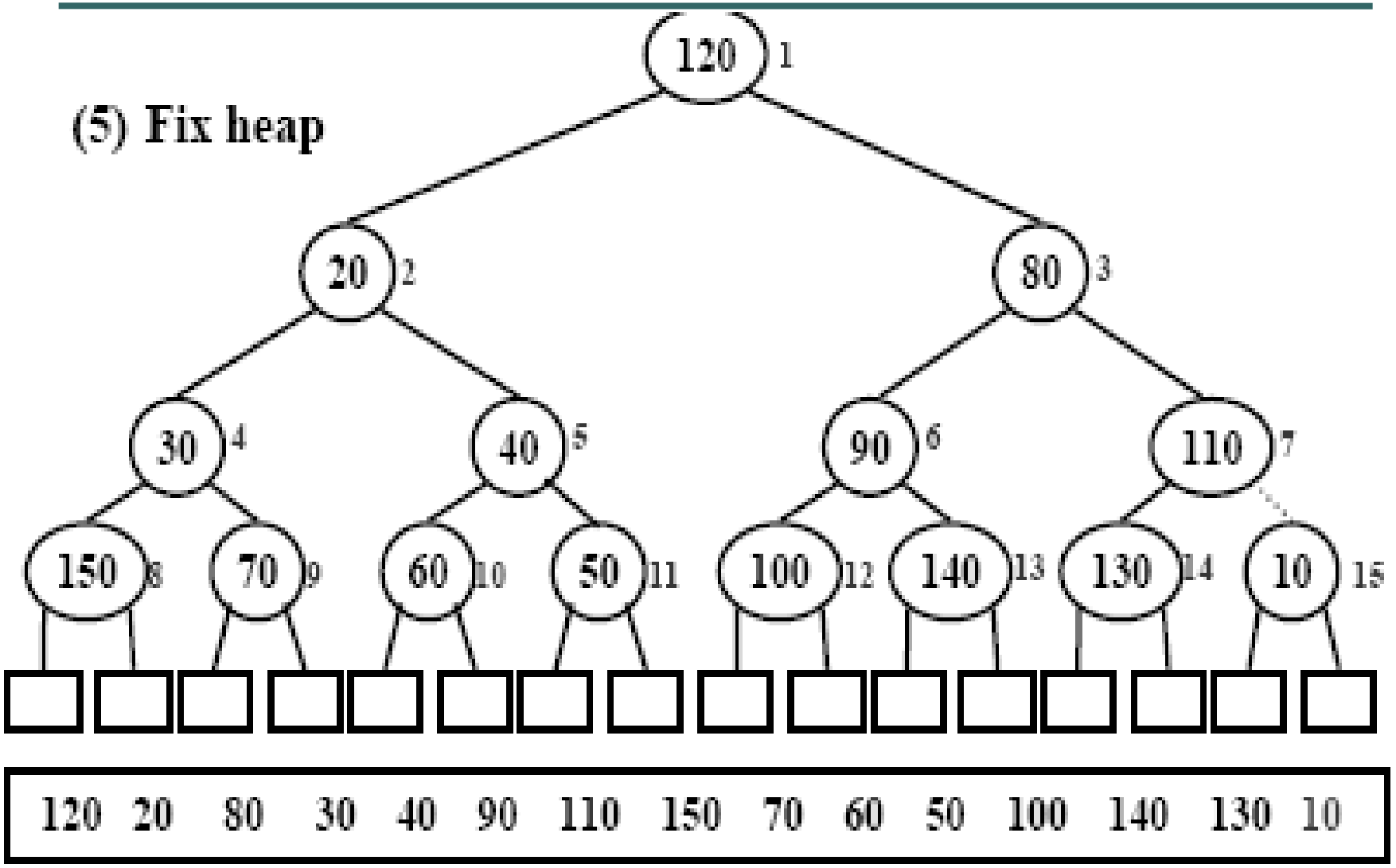
Heapsort Example



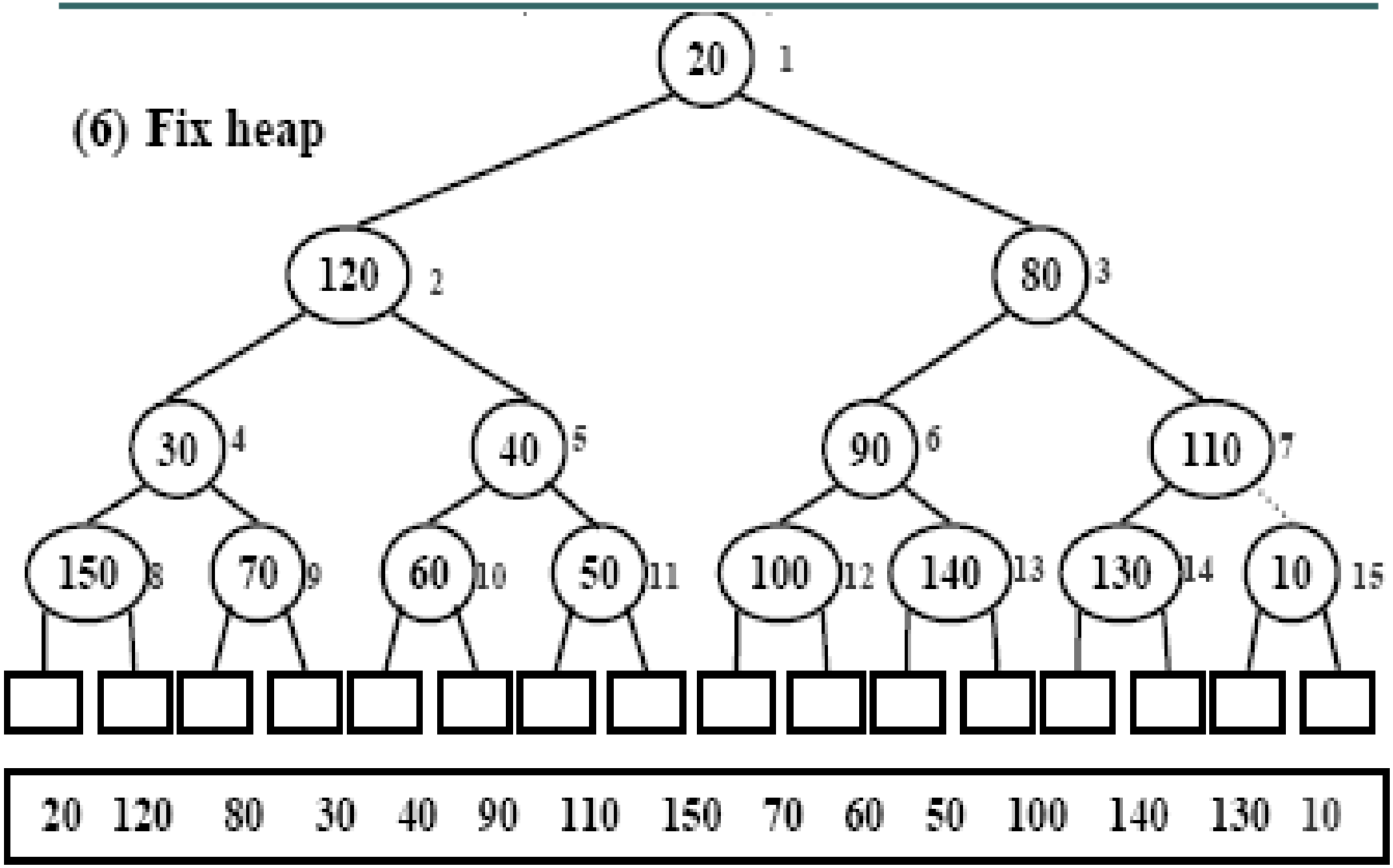
Heapsort Example



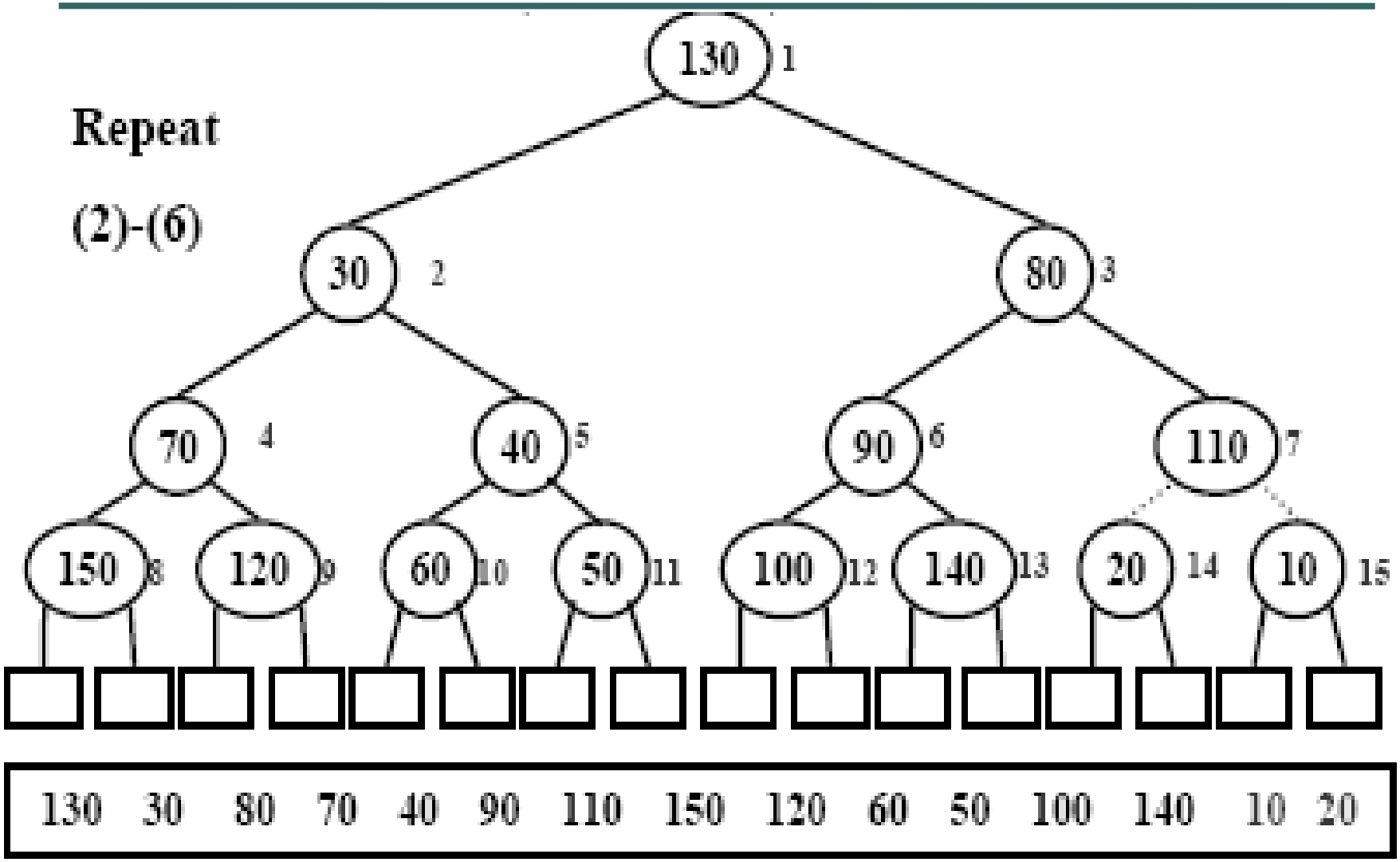
Heapsort Example



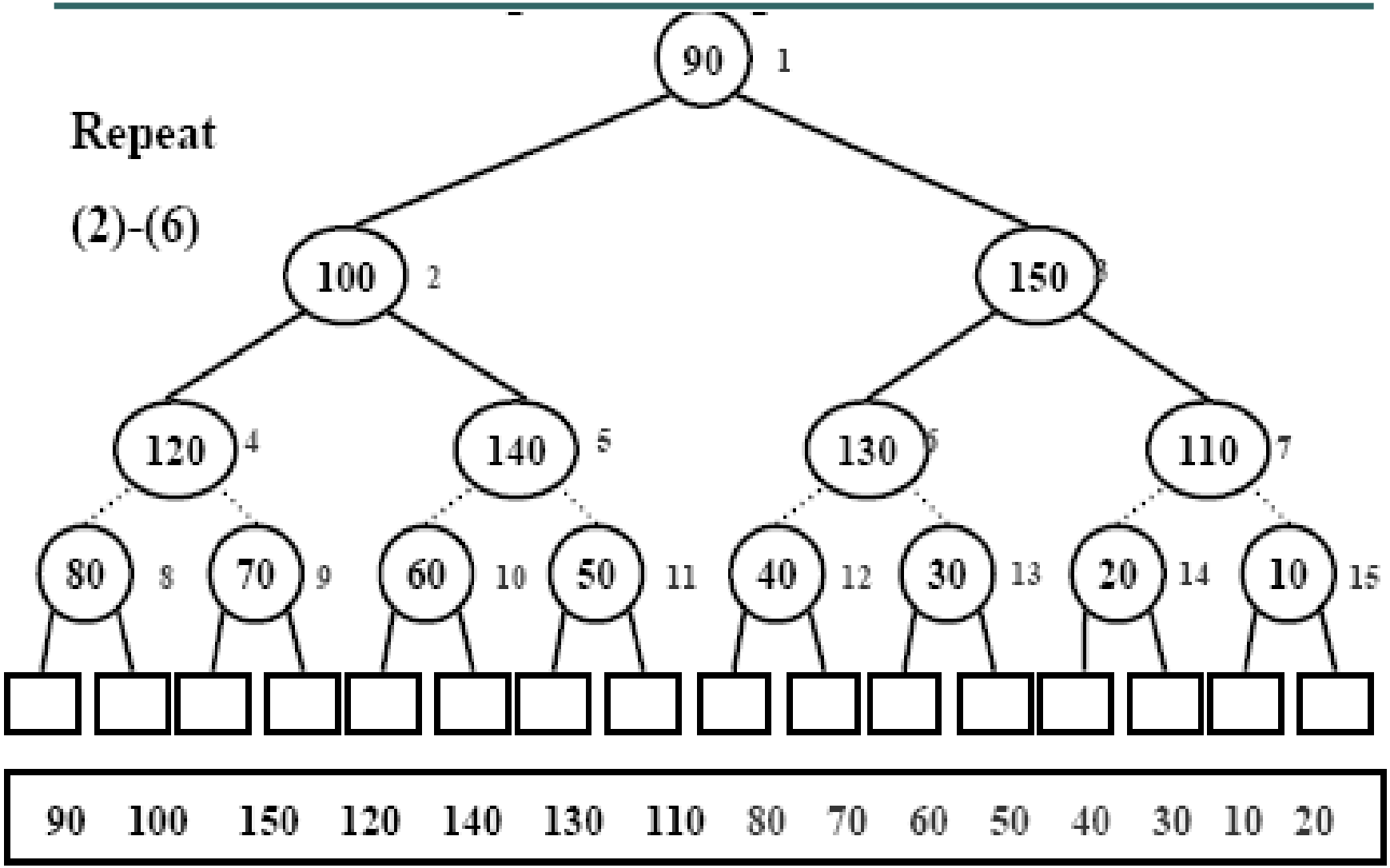
Heapsort Example



Heapsort Example



... Heapsort Example



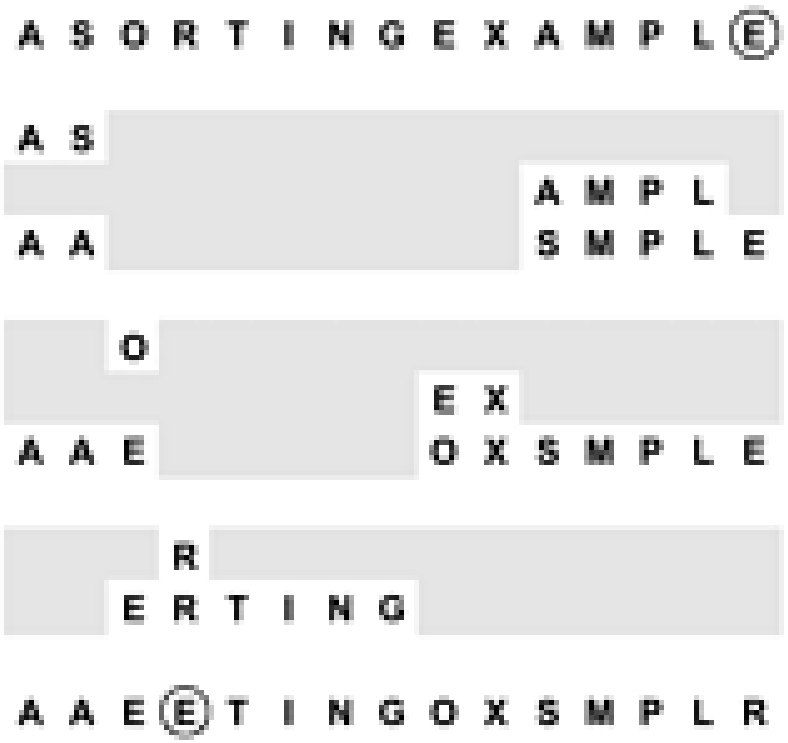
Quicksort

- Quicksort sorts *in place* and is *not stable*
- It has a running time of $\Theta(n \lg n)$ in the average case and beats Heapsort by a factor of 2 to 3.
- It has a worst case running time of $\Theta(n^2)$.
- It follows the divide-and-conquer approach:
  - **Divide**: rearranges elements of $A[p \ldots r]$ into $A[p \ldots q]$ and $A[q+1 \ldots r]$ such that all elements of $A[p \ldots q]$ are less than or equal to the elements of $A[q+1 \ldots r]$, for some q.
  - **Conquer**: sort $A[p \ldots q]$ and A$[q+1 \ldots r]$ recursively.
  - **Combine**: since sorting is done in place, no work needed.

Quicksort Algorithm

```
QUICKSORT(A, p, r)
1  if p < r
2     then q ← PARTITION(A, p, r)
3          QUICKSORT(A, p, q − 1)
4          QUICKSORT(A, q + 1, r)
PARTITION(A, p, r)
1  x ← A[r]
2  i ← p − 1
3  for j ← p to r − 1
4       do if A[j] ≤ x
5             then i ← i + 1
6                  exchange A[i] ↔ A[j]
7  exchange A[i + 1] ↔ A[r]
8  return i + 1
```

Quicksort partitioning

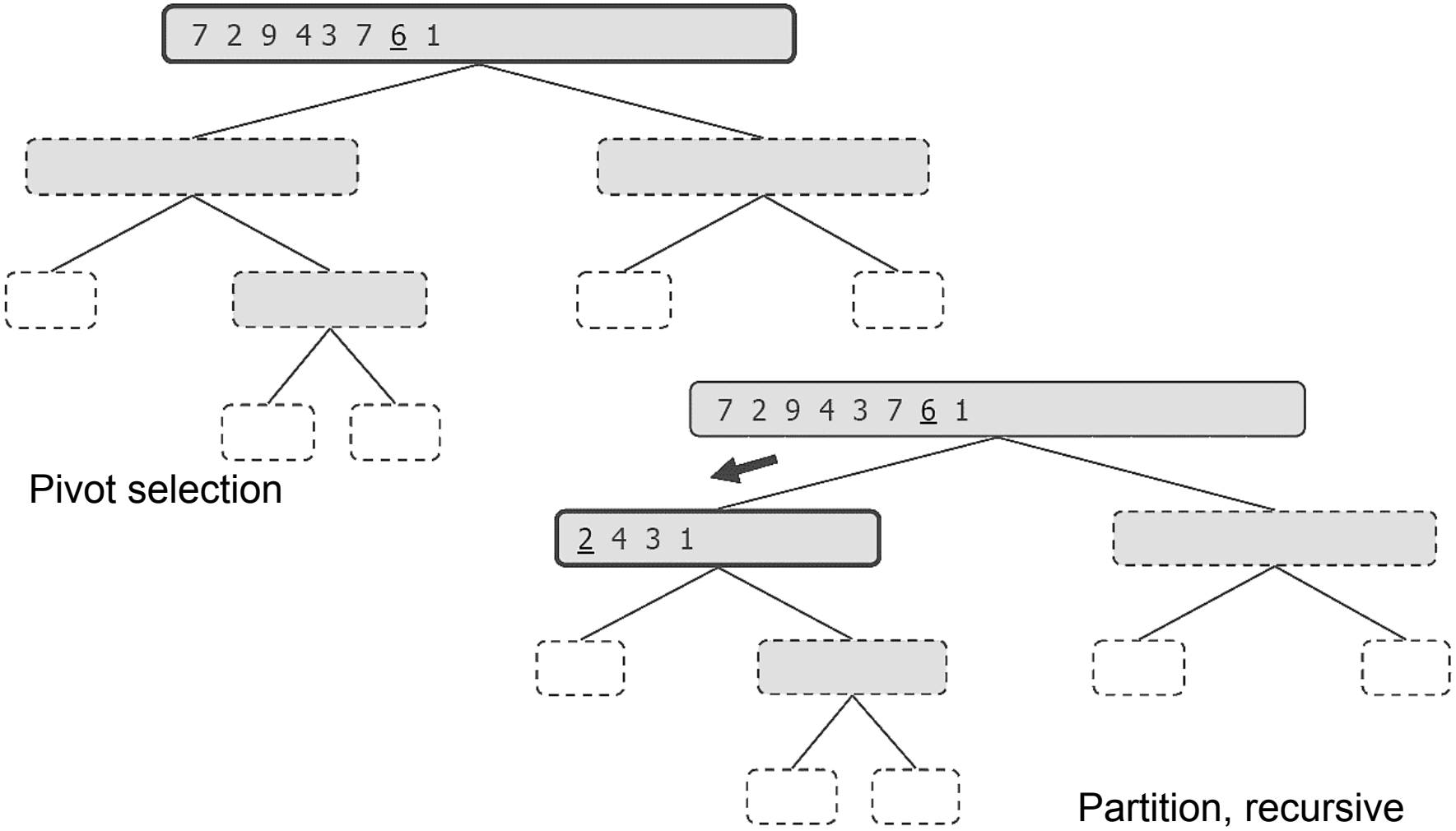


Choose pivot – here is E

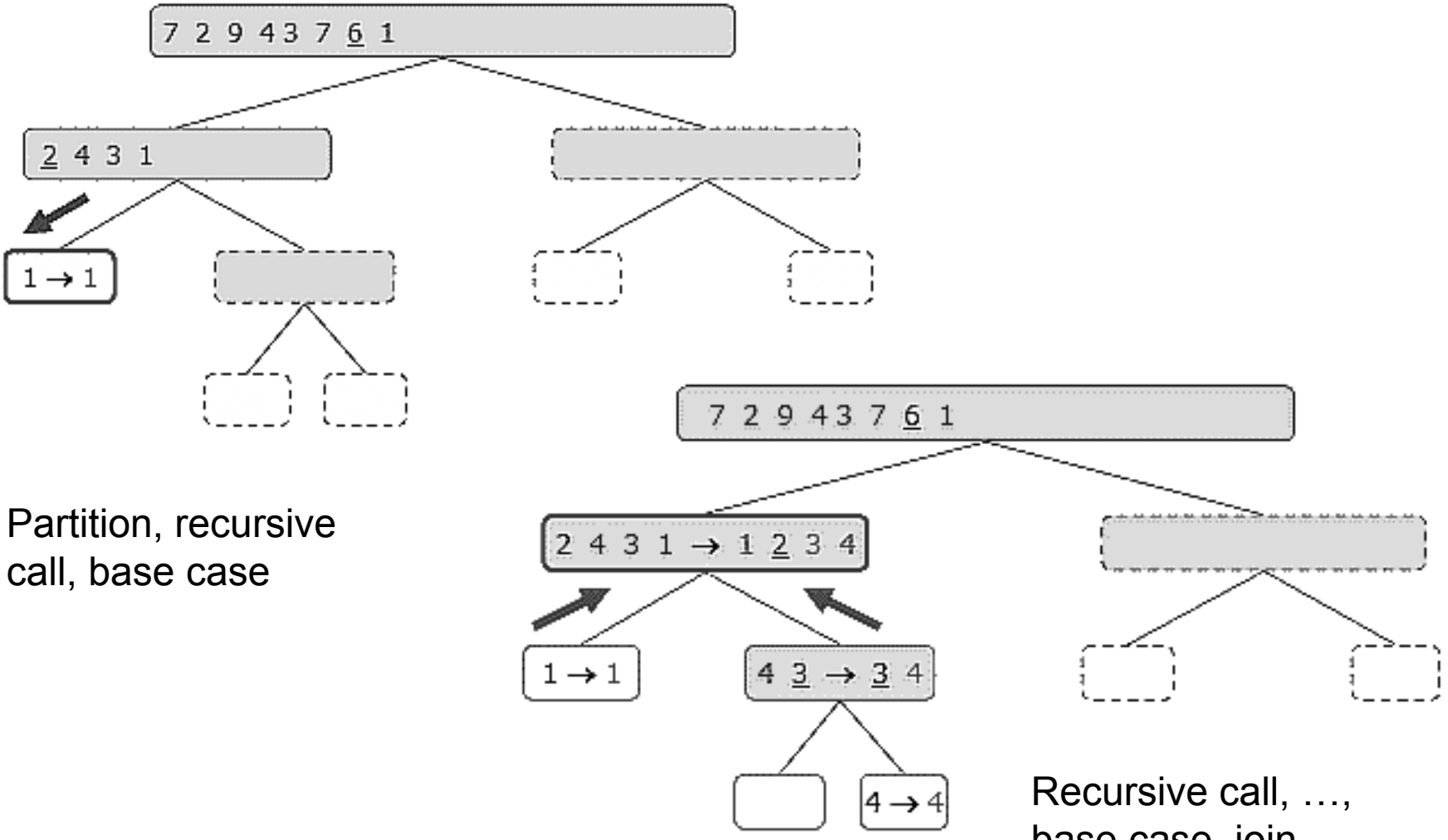
scan from the left and stop at the S,
scan from the right and stop at the A
exchange the S and the A

scan from the left; stop at the O,
scan from the right;stop at the E,
exchange the O and the E.
scanning indices cross: continue the scan from the left until we stop at the R, then continue the scan from the right (past the R) until we stop at the E.
To finish the process, we exchange the partitioning element (the E at the right) with the R.

Quicksort Operation



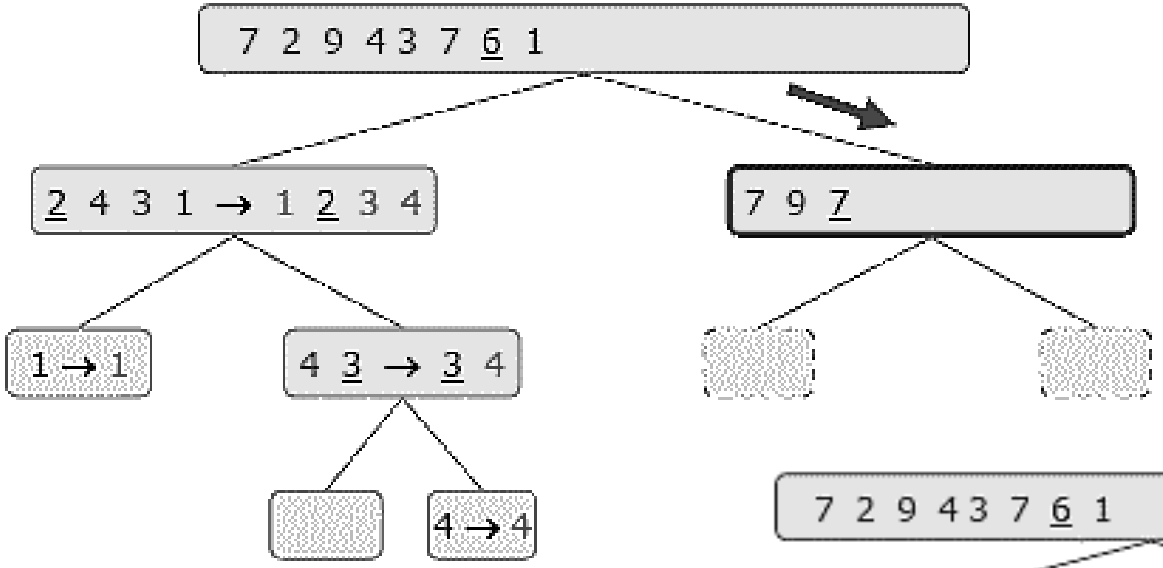
Quicksort operation



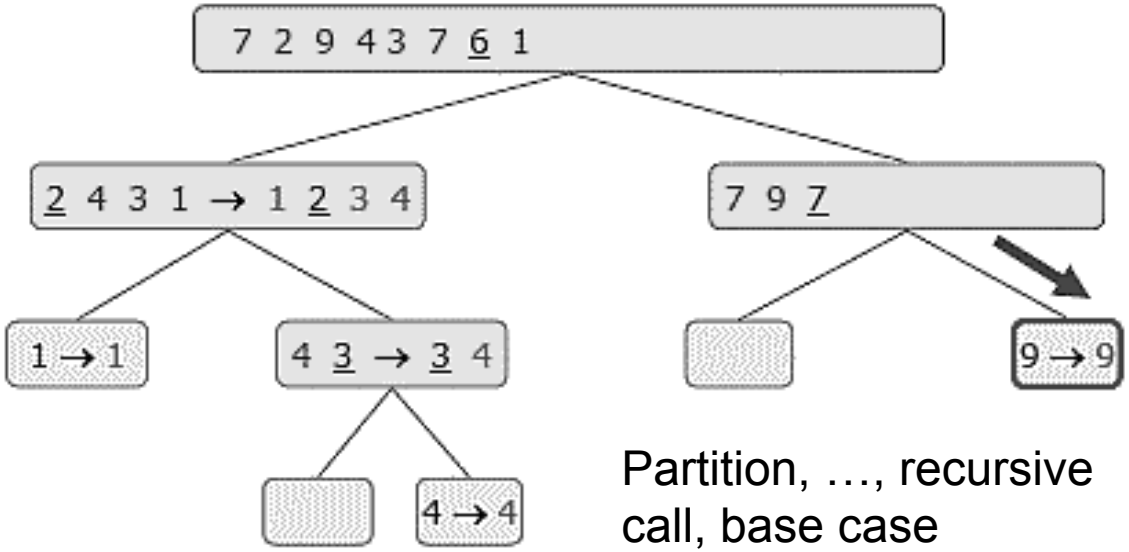
Partition, recursive call, base case

Recursive call, …, base case, join

Quicksort operation

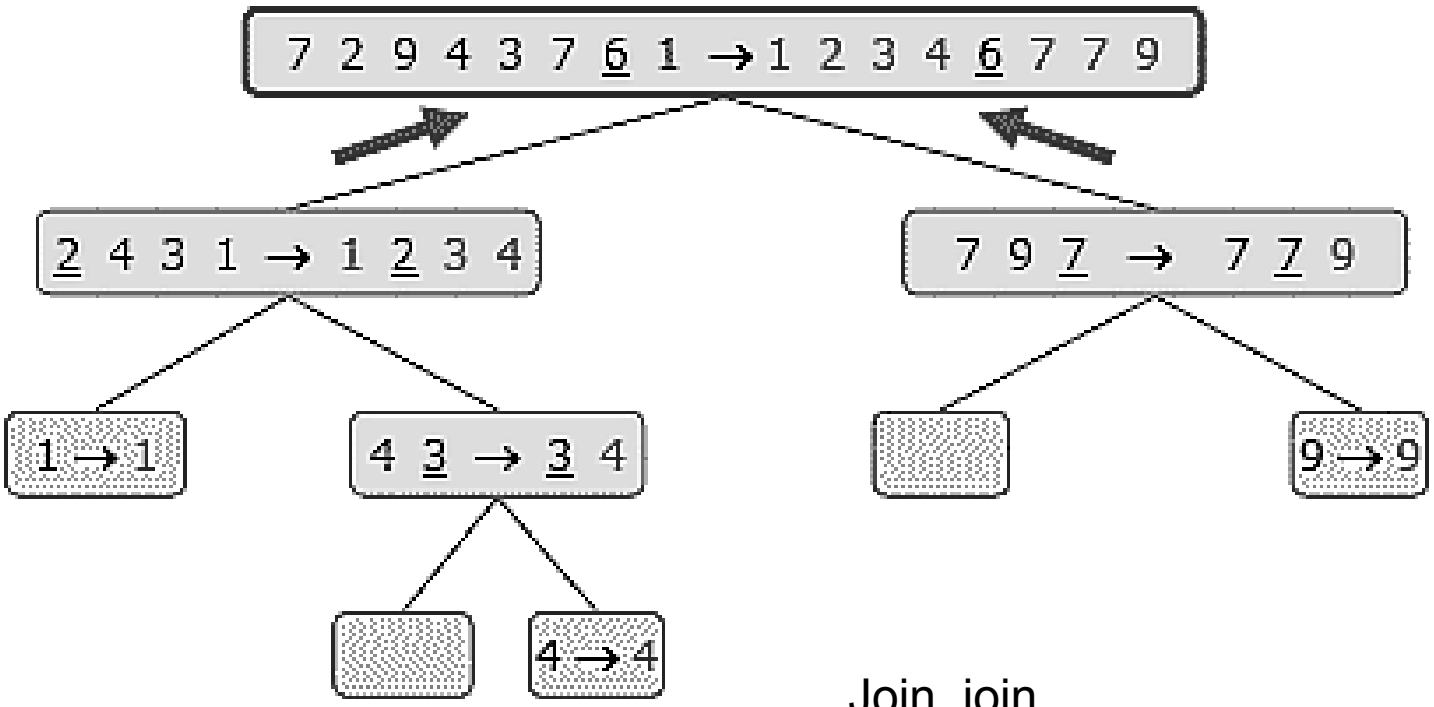


Recursive call, pivot selection



Partition, …, recursive call, base case

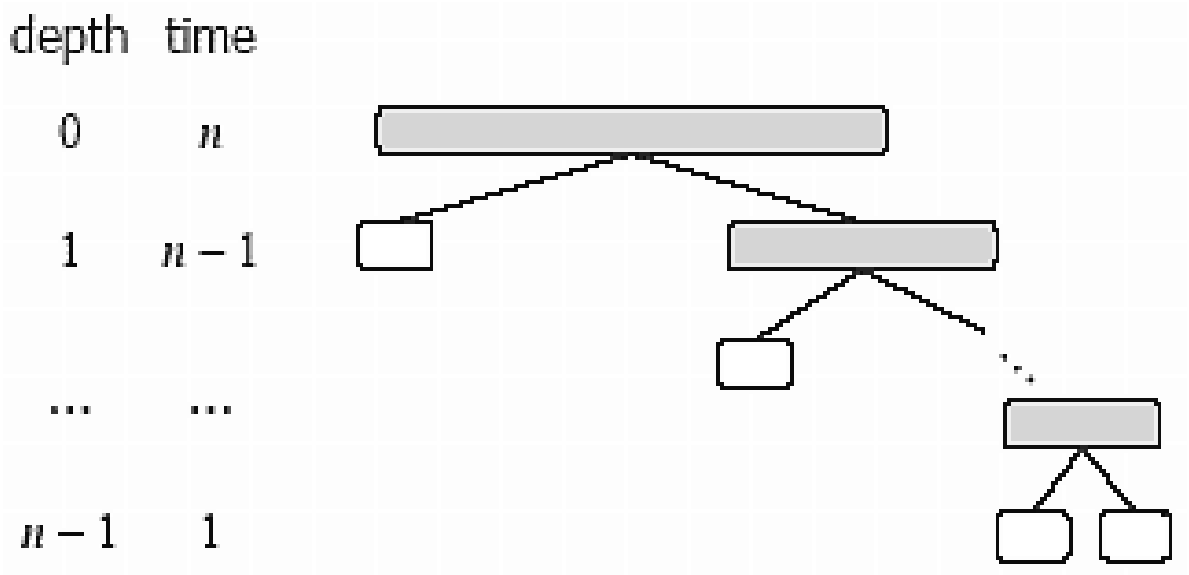
Quicksort operation



Join, join

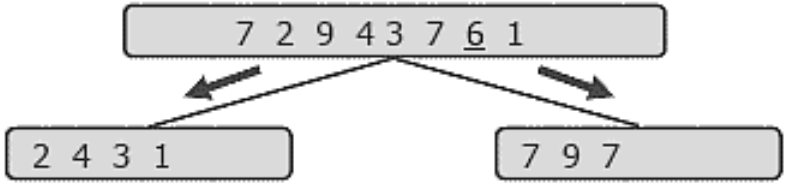
Running Time of Quicksort

- The worst case for quick-sort occurs when the pivot is the unique minimum or maximum element
- One of ***L*** and ***G*** has size $n - 1$ and the other has size 0
- The running time is proportional to the sum $n + (n - 1) + \ldots + 2 + 1$
- Thus, the worst-case running time of quick-sort is $O(n^2)$

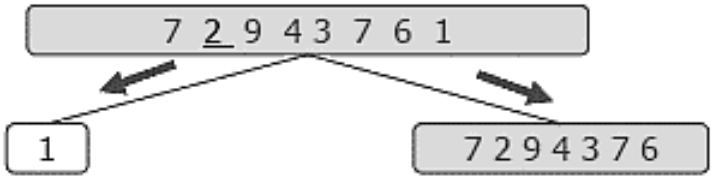


Expected Running Time

- Consider a recursive call of quick-sort on a sequence of size ***s***
  - **Good call:** the sizes of ***L*** and ***G*** are each less than 3***s***/4
  - **Bad call:** one of ***L*** and ***G*** has size greater than 3***s***/4

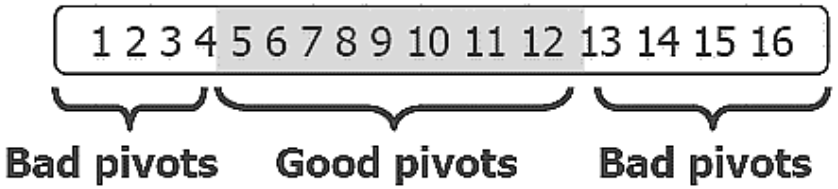


Good call



Bad call

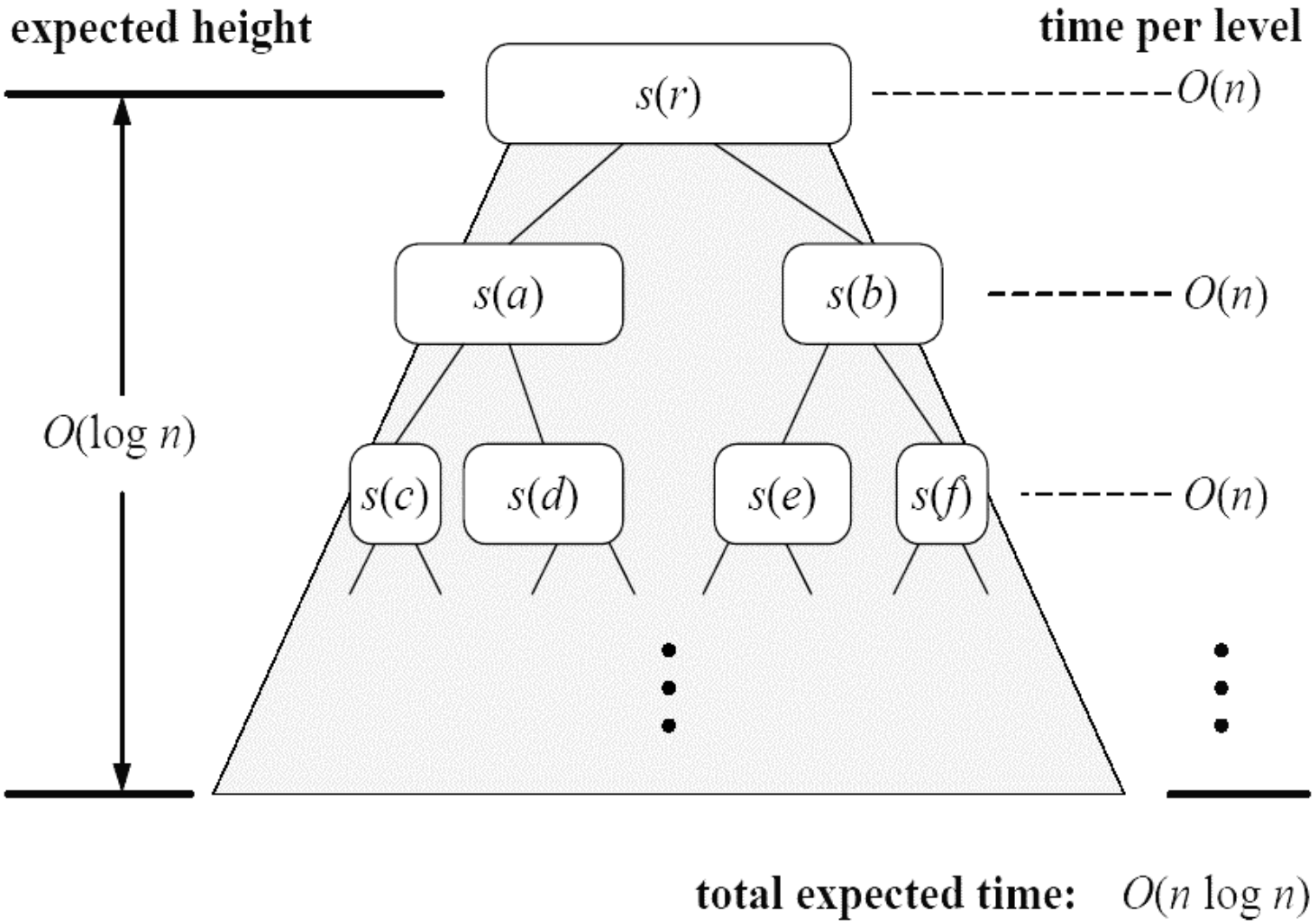
- A call is good with probability 1/2
  - 1/2 of the possible pivots cause good calls:



Expected Running Time

- Probabilistic Fact: The expected number of coin tosses required in order to get k heads is $2k$
- For a node of depth i, we expect
  - $i/2$ ancestors are good calls
  - The size of the input sequence for the current call is at most $(3/4)i/2n$
- Therefore, we have
  - For a node of depth $2\log_{4/3} n$, the expected input size is one
  - The expected height of the quick-sort tree is $O(\log n)$
- The amount or work done at the nodes of the same depth is $O(n)$
- Thus, the expected running time of quick-sort is $O(n \log n)$

Expected Running Time

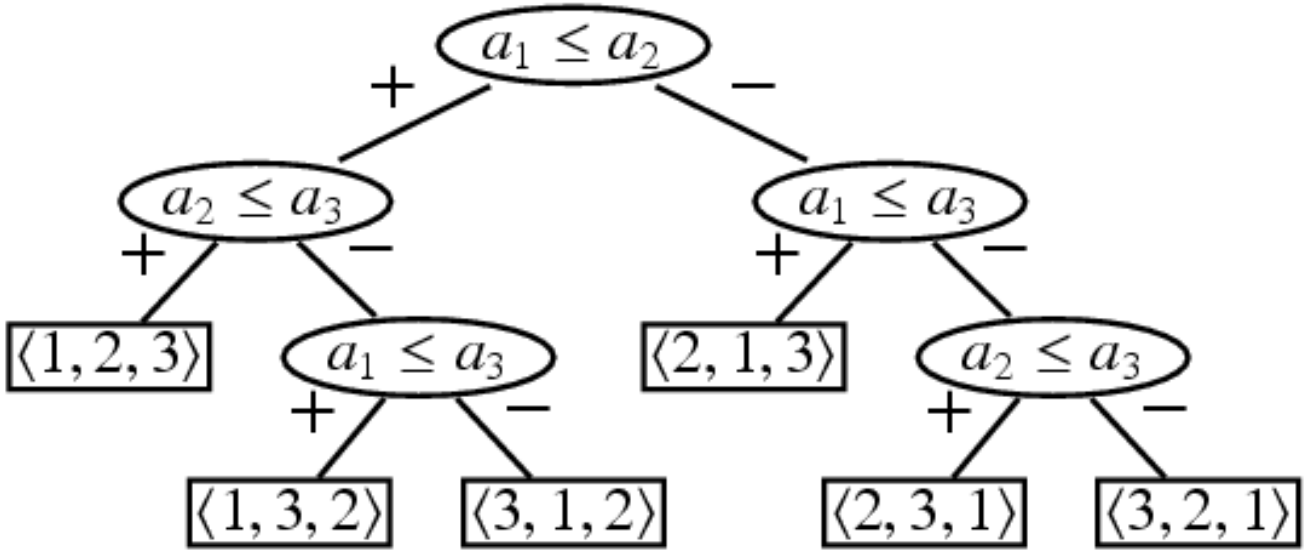


Lower Bound for Sorting, Sorting in Linear Time

- ***MergeSort*** and ***HeapSort*** have a worst case running time of $O(n \log n)$, ***Quicksort*** has that at least as average case running time.
  - Can we sort asymptotically even faster?
  - It turns out that all sorting algorithms which are based on comparison must have $O(n \log n)$ worst case running.
  - All the sorting algorithms introduced so far are comparison sorts and thus must have $O(n \log n)$ worst case running.
  - If we allow other operations than comparison we can sort in $O(n)$.
  - ***CountingSort***, ***RadixSort*** and ***BucketSort*** are examples of such algorithms.

The Decision-Tree Model (1)

- Assume the input sequence is $\langle a_1, a_2, \ldots, a_n \rangle$ and assume that we are – without loss of generality – only allowed to compare two elements a_i and a_j by $a_i \leq a_j$.
- The decision tree represents all the possible comparisons which have to be performed for establishing the order of the elements.



The Decision-Tree Model (2)

- The leaves indicate the permutation of elements for a sorted sequence.
- $n!$ different permutations for ordering n elements $\Rightarrow$ there must be $n!$ leaves in the tree.
- A path from the root to a leaf determines the comparisons which have to be performed to establish the order indicated in the leaf. Hence, the height of this path = number of comparisons needed.
- The minimum height of leaves represents the best case number of comparisons needed to establish the order of elements. For n elements, this is $n - 1$ = $\Theta(n)$.

A Lower Bound for the Worst Case

- The maximal height of leaves represents the worst case number of comparisons needed to establish the order of elements.
- We get a lower bound for the worst case by observing that a binary tree of height h as no more than 2^h elements, hence we can conclude:

$$n! \leq 2^h$$

- or equivalently $h \geq \lg(n!)$. From Stirling's approximation we know that $n! > (n/e)^n$, hence

$$\begin{aligned} h &\geq \lg (n/e)^n \\ &= n \lg n - n \lg e \\ &= \Omega(n \lg n) \end{aligned}$$

Interpretation of the Lower Bound

- As a consequence, ***HeapSort*** and ***MergeSort*** are asymptotically optimal.
- However, algorithms may still differ in their constant factors. No practical algorithm can achieve the minimal number of comparisons since this would imply that the results of all previous decisions are memorized.
- Furthermore, this is a lower bound for the number of element comparisons, and thus for the number of steps. However, algorithms may still differ in the number of element moves they require.

Counting Sort

- ***CountingSort*** is an algorithm which requires that the keys are in the range of 1 to k and exploits the fact that keys can be used for indexing an array, hence it is not based on comparison.
- First we count the occurrences of each key of the sequence A and store the count in array C:

	1	2	3	4	5	6
A =	3	4	1	4	1	1
C =	3	0	1	2		

- Then we determine the number of elements which are less than or equal to each of the keys by calculating the prefix sums:

C =	3	0	4	6

Counting Sort

- We produce the output in sequence B by going through A and determining for each element to which position in B goes by looking that up in C. Then we decrease the entry for that element in C.

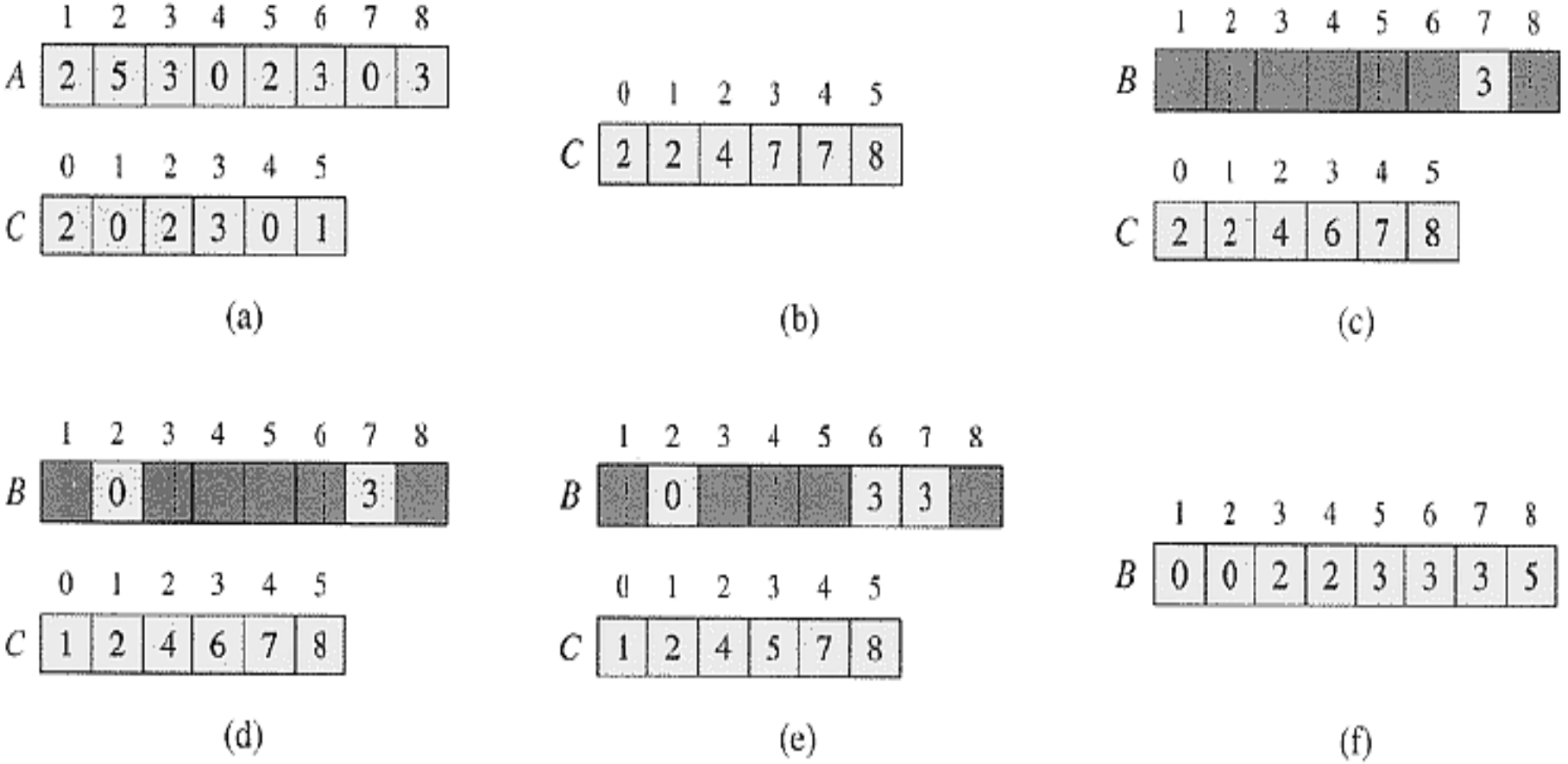
	1	*2*	*3*	*4*	*5*	*6*
A **=**	3	4	1	4	1	1
B **=**			1			
C **=**	2	3	4	6		
B **=**	1	1	1	3		
C **=**	1	3	4	6		
B **=**	1	1	1	3		4
C **=**	1	3	4	5		

Analisys of Counting Sort

1.	**for** $i \leftarrow 0$ **to** k	$\Theta(k)$
2.	**do** $C[i] \leftarrow 0$	$\Theta(1)$
3.	**for** $j \leftarrow 1$ **to** *length*$[A]$	$\Theta(n)$
4.	**do** $C[A[j]] \leftarrow C[A[j]]+1$	$\Theta(1)$ $(\Theta(1)$ $\Theta(n)= \Theta(n))$
5.	// $C[i]$ contains number of elements equal to i	
6.	**for** $i \leftarrow 1$ **to** k	$\Theta(k)$
7.	**do** $C[i] = C[i] + C[i-1]$	$\Theta(1)$ $(\Theta(1)$ $\Theta(n)= \Theta(n))$
8.	// C$[i]$ contains number of elements $\leq i$.	
9.	**for** $j \leftarrow$ *length*$[A]$ **downto** 1	$\Theta(n)$
10.	**do** $B[C[A[j]]] \leftarrow A[j]$	$\Theta(1)$ $(\Theta(1)$ $\Theta(n)= \Theta(n))$
11.	$C[A[j]] \leftarrow C[A[j]]-1$	$\Theta(1)$ $(\Theta(1)$ $\Theta(n)= \Theta(n))$

- Total cost is $\Theta(k+n)$, suppose $k=O(n)$, then total cost is $\Theta(n)$. **Beat $\Omega(n \lg n)$.**

Example of Counting Sort



Radix Sort

- Suppose you have a sequence of integers to sort. Consider their decimal representation and suppose it requires two digits. Then we sort them by first sorting them by to their last digit and then by their first digit.

21		21		14
17	$\Rightarrow$	14	$\Rightarrow$	17
75		75		21
14		17		75

RADIX-SORT(A, d)

1 **for** $i \leftarrow 1$ **to** d

2 **do** use a stable sort to sort array A on digit i

Radix Sort

- Suppose each digit is in the range of 1 to k and k is not too large. Then we can use counting sort to sort on the i-th digit. Thus each pass over n digits takes $\Theta(n + k)$
- Since there are d passes, the total running time is $\Theta(d\,(n + k))$.
- This can also be applied to other sequences of keys, like year-month-date.

Bucket Sort

- Assume that our keys are real numbers in the interval $[0,1)$. We create say 10 buckets for each of the intervals $[i / 10, (i+1) / 10)$ and store each element in its appropriate bucket.
- Finally we sort the buckets with e.g. insertion sort.
- This takes $\Theta(n)$ on average, assuming that the number are equally distributed and we have chosen the intervals sufficiently small.
- Again, this is not based on comparisons, rather we assume that we can multiply the keys by 10 and take the integer part to select the bucket.

Bucket Sort (Bin Sort) Algorithm

```
BUCKET-SORT(A)
1  n ← length[A]
2  for i ← 1 to n
3        do insert A[i] into list B[⌊n A[i]⌋]
4  for i ← 0 to n − 1
5        do sort list B[i] with insertion sort
6  concatenate the lists B[0], B[1], . . . , B[n − 1] together in order
```

Analysis of Bucket Sort

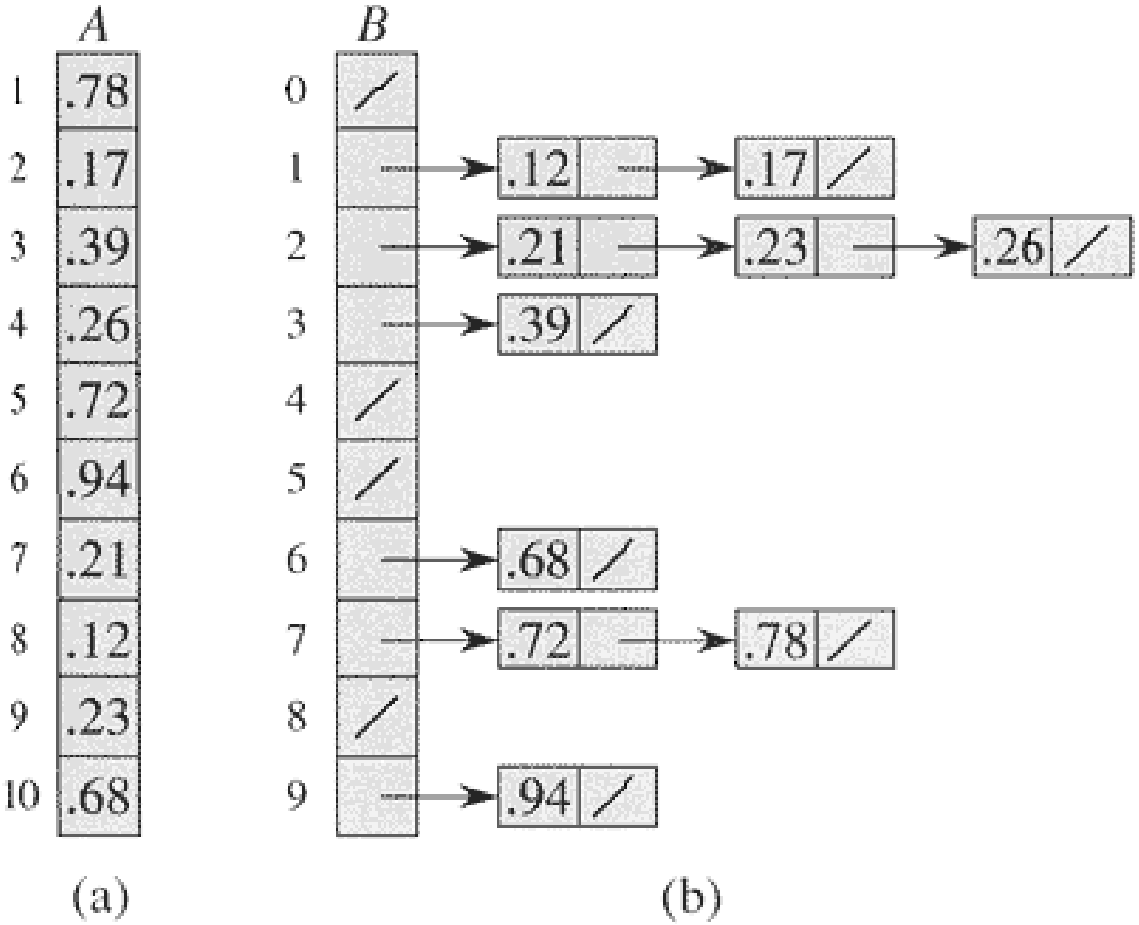
$n \leftarrow$length[A]	$\Omega(1)$
for $i \leftarrow 1$ to n	$O(n)$
do insert $A[i]$ into bucket B[$\lfloor nA[i] \rfloor$]	$\Omega(1)$ (**i.e. total** $O(n)$)
for $i \leftarrow 0$ to $n-1$	$O(n)$
do sort bucket $B[i]$ with insertion sort	$O(n_i^2)$ ($\sum_{i=0}^{n-1} O(n_i^2)$)
Concatenate bucket $B[0], B[1], \ldots, B[n-1]$	$O(n)$

Where n_i is the size of bucket $B[i]$.

Thus $T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$

$= \Theta(n) + nO(2-1/n) = \Theta(n)$. **Beat $\Omega(n \lg n)$**

Operation of Bucket Sort



Selecting a Sorting Algorithm (1)

- Even for a simple problem as sorting, there are many algorithms. Which one should we use? It depends on the situation:
- Is the size of the input large (e.g. all courses) or small (e.g. courses a student is taking)?
- For a small input size $O(n^2)$ algorithms like *InsertionSort* perform better, for a large size $O(n \lg n)$ algorithms.
- Do the elements to be sorted require lots of memory?
  - If so, we can avoid moving them around during sorting by using an auxiliary sequence with pointers to the elements instead and moving only the pointers.
  - If they are small, we better sort them directly.

Selecting a Sorting Algorithm (2)

- Can elements have the same keys? If so, do we require a stable sort?
  - $O(n^2)$ algorithms tend to be stable, $O(n \log n)$ in place algorithms not.
  - However, we can make any unstable algorithm stable by adding a key with the position of the elements in the original array. This costs extra space and extra time for the comparisons.
  - If we decided anyway to sort the sequence of pointers rather than the elements, we can use the position of the elements in the unsorted sequence in comparisons. In this case, no additional space is required.
- Do we require guarantees on the sorting time, e.g. in a hard realtime environment (e.g. in control systems, networks)?
  - This rules out Quicksort because of its $\Theta(n^2)$ worst case behavior

Selecting a Sorting Algorithm (3)

- Do we have a limited amount of space available, like in embedded processor?
  - This ***rules out MergeSort*** since it requires in the order of n extra space and it makes ***Quicksort*** questionable since it requires also in the order of n extra space in the worst case. However, we can improve ***Quicksort*** to require only in the order of $\lg n$ extra space.
- Can the sequence be so large that it does not completely fit into main memory and virtual memory is going to be used?
  - If so, sorting algorithms with good local behavior are to be preferred.
    - If we are at element $A[i]$ in the ***Heapify*** procedure of ***HeapSort***, then the next element accessed with be $A[2i]$ or $A[2i+1]$, and so forth, so elements are accessed all over the array in quick succession.
    - The Partition procedure of Quicksort accesses $A[i]$, then $A[i+1]$, etc., as well as $A[j]$, then $A[j-1]$, etc., so has a good local behavior. Most $O(n^2)$ algorithms have good local behavior.

Selecting a Sorting Algorithm (4)

- Is the input so big that is cannot fit into main memory and too big for virtual memory?
  - Then we have to use external sorting algorithms anyway.
- Do we know more about the input which we can exploit for sorting in $\Theta(n)$?
  - If the keys are in a small range of integers (e.g. the age of a person, year of printing), we can use ***CountingSort***.
  - If each key is a sequence of keys which can be compared on their own we can use ***RadixSort***.
  - If the keys are real number over an interval and are distributed evenly, we can use ***BucketSort***.

Reading

- AHU, chapter 8
- Preiss, chapter: Sorting Algorithms and Sorters
- CLR, chapters 7 section 7.4, 8 sections 8.1, 8.2, 9, CLRS chapter 2, sect. 1, chapters 6, 7, 8
- Notes