EECS 2011M: Fundamentals of Data Structures

Instructor: Suprakash Datta

Office : LAS 3043

Course page: http://www.eecs.yorku.ca/course/2011M

Also on Moodle

Note: Some slides in this lecture are adopted from James Elder' slides.

Sorting: Motivation

1. We have seen that sorted data allow faster operations than unsorted data
2. Fundamental step in algorithm design
3. There are many available sorting algorithms, and we need to understand them to select the best one for an application

Sorting: Problem

1. We want to be able to sort any data as long as we can compare them (i.e., an appropriate comparator is available)

2. For this lecture we restrict ourselves only to comparison-based sorts

3. We assume that the data are stored in arrays

Comparison-based Sorting

1. Very general, makes no assumptions about the type of data. The data may not be only keys. They may be records with a key that is used to sort

2. Work by comparing elements and moving data around based on comparison results

3. Algorithms we will look at are: selection sort, bubble sort, insertion sort, merge sort, quick sort, heap sort

(Sub) Classes of Sorting Algorithms

1. In-place sort: Use only O(1) extra space (in addition to the data)

2. Stable sort: the ordering of identical keys in the input is preserved in the output.

3. Algorithms we will look at are: selection sort, bubble sort, insertion sort, merge sort, quick sort, heap sort

Selection sort

Algorithm: Given an array A of n integers, sort them by repetitively selecting the smallest among the yet unselected integers.

Is this precise enough?

Not-in-place version:

1. find the next smallest value, mark it "deleted" and copy it to an output array.
2. Continue in this way until all the input elements have been selected and placed in the output list in the correct order.

Selection sort

Algorithm: Given an array A of n integers, sort them by repetitively selecting the smallest among the yet unselected integers.

Is this precise enough?

In-place version:

1. Swap the smallest integer with the integer currently in the place where the smallest integer should go.
2. Continue in this way until all the input elements have been selected and placed in the output list in the correct order.

Selection sort

```
for i = 0 to n-1
    //Loop Invariant = ?
    j_min = i
    for j = i+1 to n-1
        if A[ j ] < A[j_min]
            j_min = j
    swap A[i] with A[j_min]
```

Selection sort

for i = 0 to n-1

//Loop Invariant: A[0…i-1] contains the i smallest keys in sorted order.

Is this precise enough?

j_{min} = i

for j = i+1 to n-1

if A[j] < A[j_{min}]

j_{min} = j

swap A[i] with A[j_{min}]

Selection sort

```
for i = 0 to n-1
    //Loop Invariant: A[0…i-1] contains the i smallest keys
    in sorted order.
    // A[i…n-1] contains the remaining keys

    j_min = i
    for j = i+1 to n-1
        if A[ j ] < A[j_min]
            j_min = j
    swap A[i] with A[j_min]
```

Exercise: complete the proof of correctness

Selection sort

for i = 0 to n-1

//Loop Invariant: A[0…i-1] contains the i smallest keys in sorted order.

// A[i…n-1] contains the remaining keys

j_{min} = i

for j = i+1 to n-1

if A[j] < A[j_{min}]

j_{min} = j

swap A[i] with A[j_{min}]

Running time?

Θ(n-i-1)

Running time: $\Theta(n^2)$

Bubble sort

```
for i = n-1 downto 1
    //Loop Invariant : A[i+1…n-1] contains the n-i-1 largest
    keys in sorted order.
    //A[0…i] contains the remaining keys

    for j = 0 to i-1
        if A[ j ] > A[ j + 1 ]
            swap A[ j ] and A[ j + 1 ]
```

Running time?

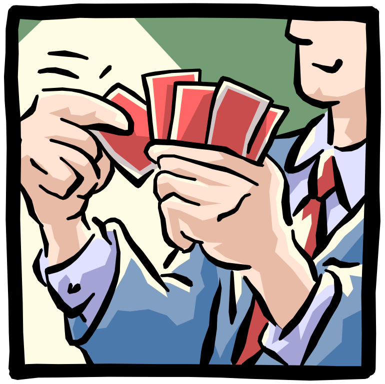
Θ(i)

Running time: $\Theta(n^2)$

Useful for practice in proving correctness, and little else

Analysis example: Insertion sort

"We maintain a subset of elements sorted within a list. The remaining elements are off to the side somewhere. Initially, think of the first element in the array as a sorted list of length one. One at a time, we take one of the elements that is off to the side and we insert it into the sorted list where it belongs. This gives a sorted list that is one element longer than it was before. When the last element has been inserted, the array is completely sorted."



English descriptions:

- Easy, intuitive.
- Often imprecise, may leave out critical details.

Insertion sort: pseudocode

```
for i=1 to length(A)-1
   do key=A[i]
      j=i
      while j>0 and A[j-1]>key
        do A[j]=A[j-1]
           j--
      A[j]:=key
```

Can you understand
The algorithm?
I would not know
this is insertion sort!

Moral: document code!

What is a good loop invariant?

Correctness of Insertion sort

```
for i=1 to length(A)-1
   do key=A[i]
 //Insert A[i] into the sorted
//sequence A[0..i-1]
      j=i
      while j>0 and A[j-
1]>key
        do A[j]=A[j-1]
           j--
      A[j]:=key
```

Invariant: at the start of for loop iteration i, A[0…i-1] consists of elements originally in A[0…i-1] but in sorted order,

A[i…n-1] contains the remaining keys

Initialization: *i= 1,* the invariant trivially holds because *A*[0] is a sorted array ☺

Correctness of Insertion sort – contd.

```
for i=1 to length(A)-1
   do key=A[i]
      j=i
      while j>0 and A[j-1]>key
        do A[j]=A[j-1]
           j--
      A[j]:=key
```

Invariant: at the start of for loop iteration i, A[0…i-1] consists of elements originally in A[0…i-1] but in sorted order, A[i…n-1] contains the remaining keys

Maintenance: the inner **while** loop moves elements $A[k]$, $A[k+1]$, …, $A[i-1]$ one position right without changing their order. Then the former $A[j]$ element is inserted into k^{th} position so that $A[k-1] \le A[k] \le A[k+1]$.

$A[0\ldots i-1]$ sorted + $A[i] \rightarrow A[0\ldots i]$ sorted

Correctness of Insertion sort – contd.

```
for i=1 to length(A)-1
   do key=A[i]
      j=i
      while j>0 and A[j-1]>key
        do A[j]=A[j-1]
           j--
      A[j]:=key
```

Invariant: at the start of for loop iteration i, A[0…i-1] consists of elements originally in A[0…i-1] but in sorted order, A[i…n-1] contains the remaining keys

Termination: the loop terminates, when *i=n*.

Then the invariant states: *"A[0...n-1] consists of elements originally in A[0...n-1] but in sorted order"* ☺

Divide and conquer

Divide-and conquer is a general algorithm design paradigm:

1. Divide: divide the input data $\boldsymbol{S}$ in two disjoint subsets $\boldsymbol{S}_1$ and $\boldsymbol{S}_2$
2. Conquer: solve the subproblems associated with $\boldsymbol{S}_1$ and $\boldsymbol{S}_2$ (often done recursively)
3. Combine: combine the solutions for $\boldsymbol{S}_1$ and $\boldsymbol{S}_2$ into a solution for $\boldsymbol{S}$

The base case for the recursion is a subproblem of size 0 or 1

More divide and conquer : Merge Sort

- **Divide**: If *S* has at least two elements (nothing needs to be done if *S* has zero or one elements), remove all the elements from *S* and put them into two sequences, S_1 and S_2 , each containing about half of the elements of S. (i.e. S_1 contains the first $\lceil n/2 \rceil$ elements and S_2 contains the remaining $\lfloor n/2 \rfloor$ elements).
- **Conquer**: Sort sequences S_1 and S_2 using Merge Sort.
- **Combine**: Put back the elements into *S* by merging the sorted sequences S_1 and S_2 into one sorted sequence

Merge Sort: Algorithm

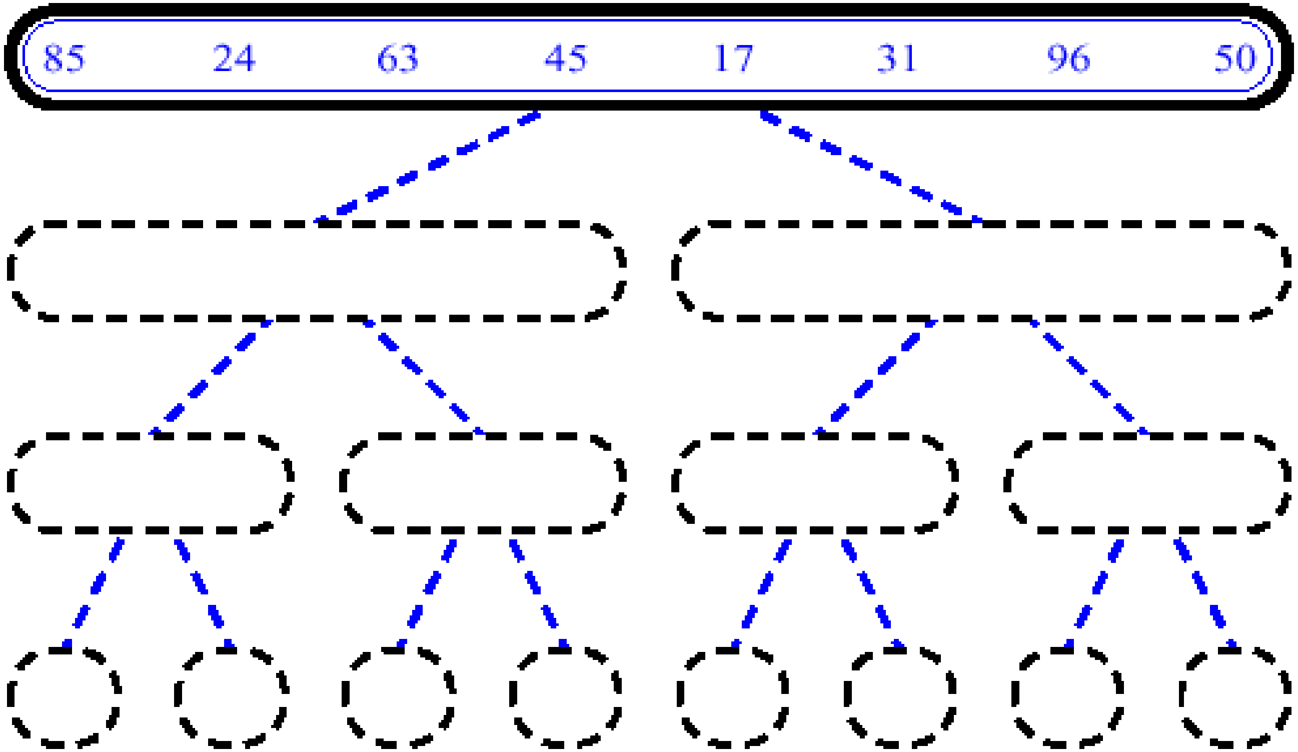
```
Merge-Sort(A, p, r)
   if p < r then
      q←(p+r)/2
      Merge-Sort(A, p, q)
      Merge-Sort(A, q+1, r)
      Merge(A, p, q, r)
```

```
Merge(A, p, q, r)
   Take the smallest of the two topmost elements of
sequences A[p..q] and A[q+1..r] and put into the
resulting sequence. Repeat this, until both sequences
are empty. Copy the resulting sequence into A[p..r].
```

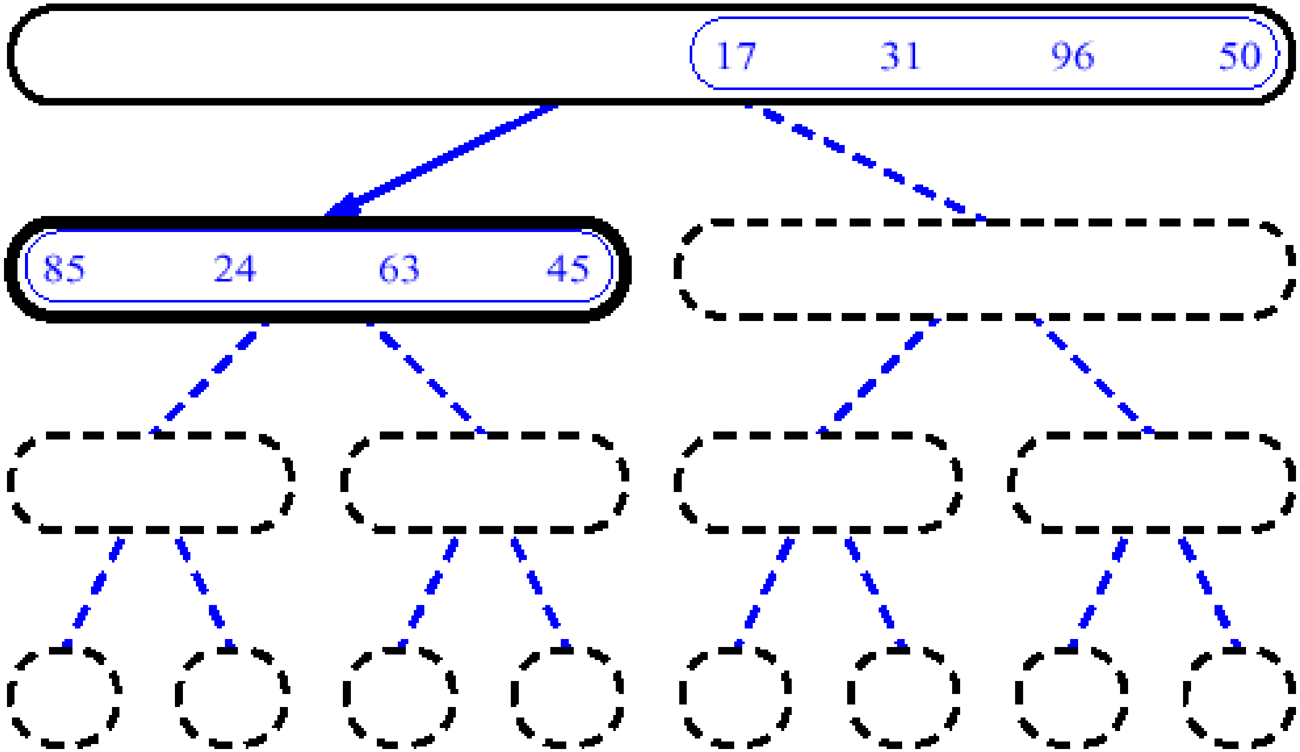
The Merge Step

1. Merging two sorted sequences, each with $n/2$ elements takes $O(n)$ time
2. Straightforward to make the sort stable.
3. Normally, merging is not in-place: new memory must be allocated to hold S.
4. It **is** possible to do in-place merging using linked lists.

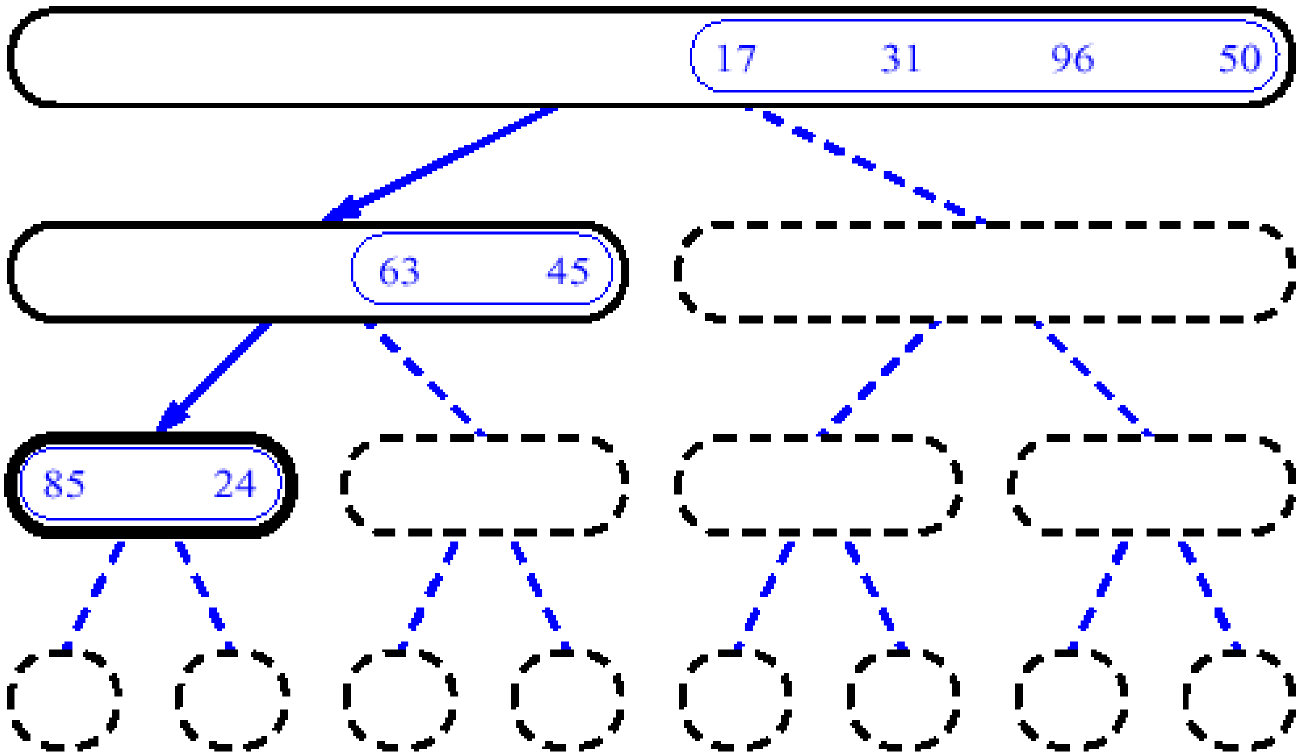
Merge Sort: example



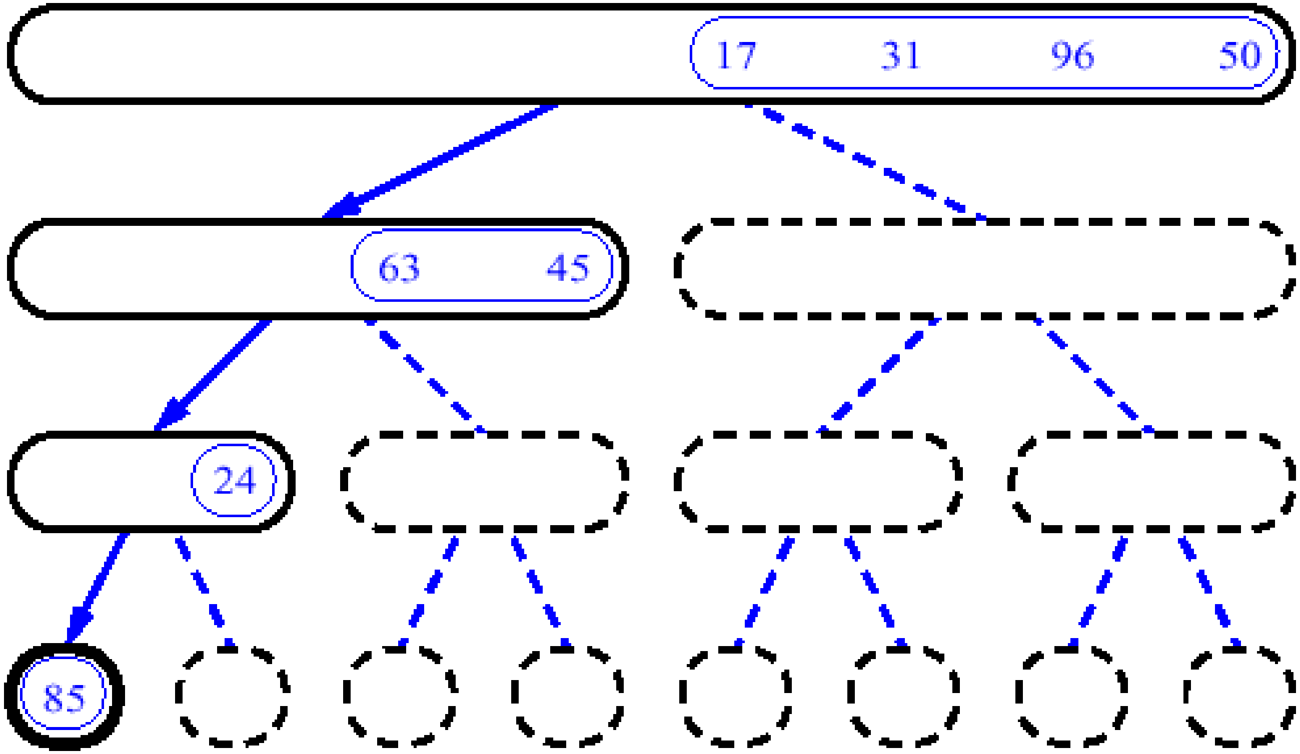
Merge Sort: example



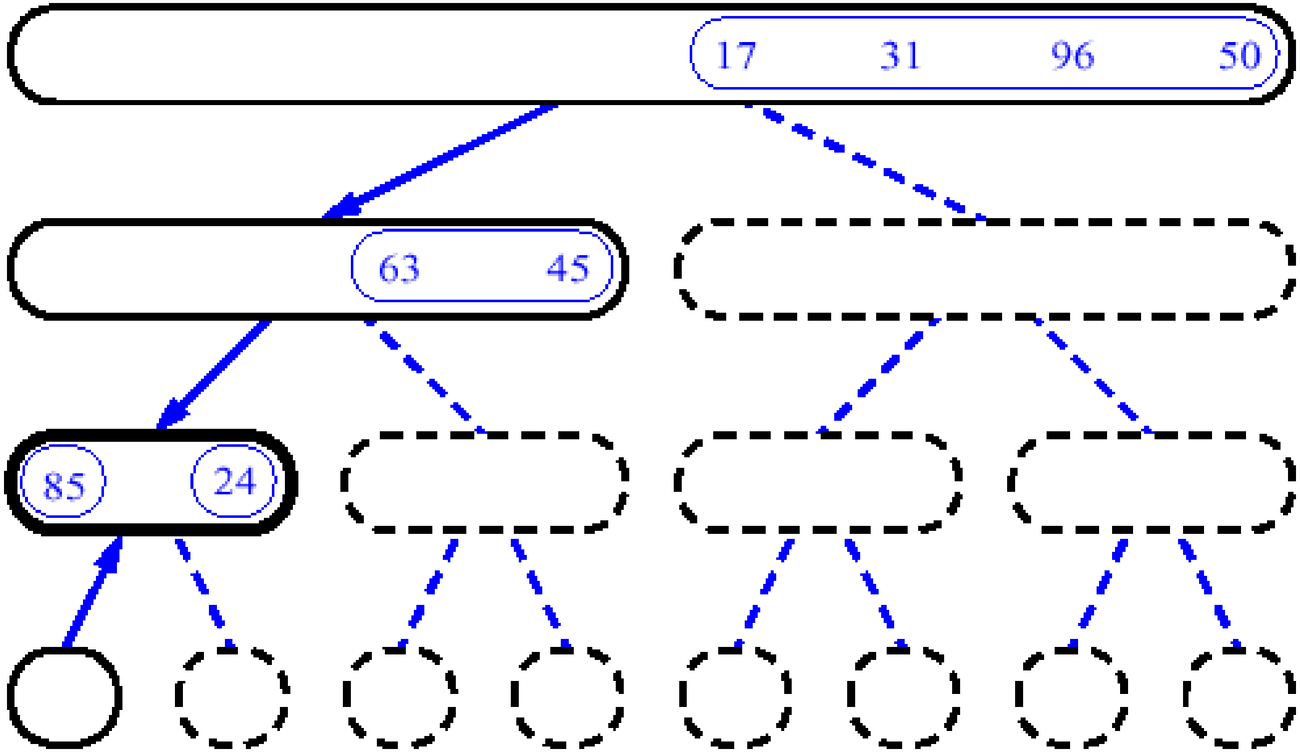
Merge Sort: example



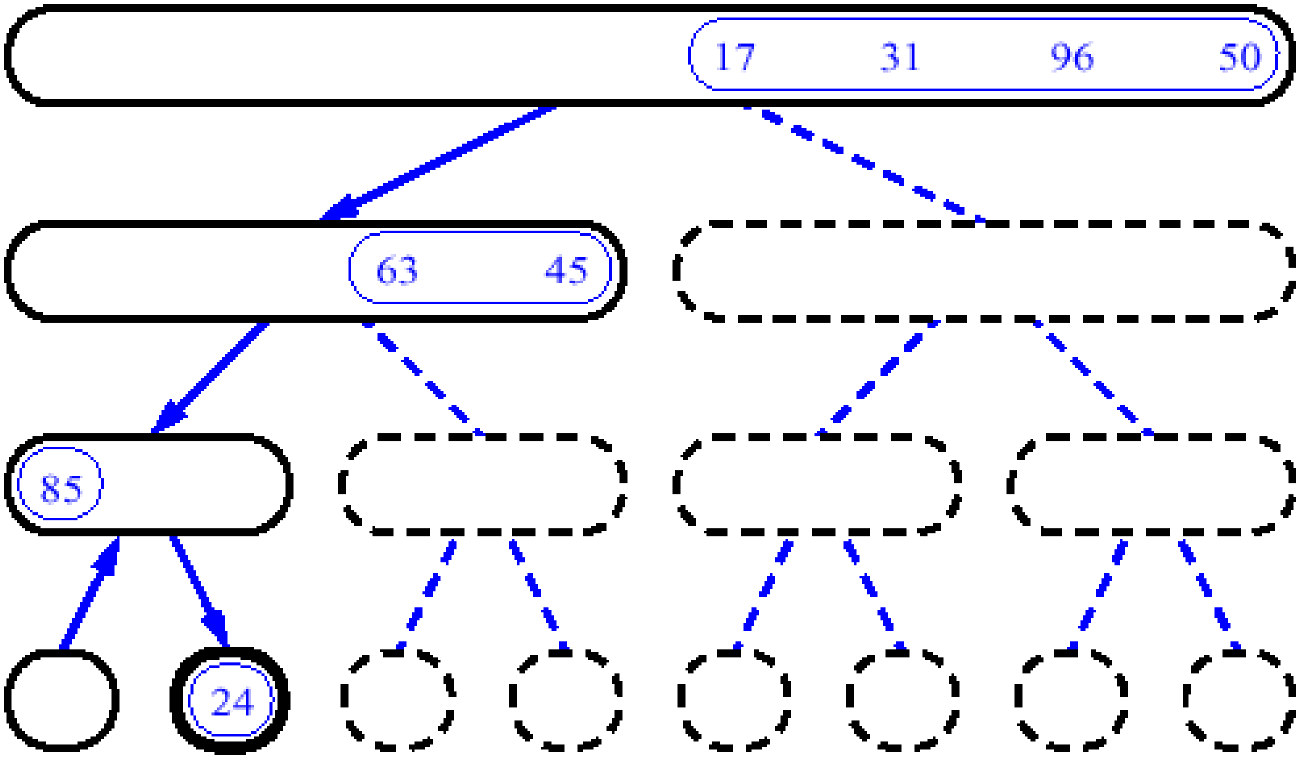
Merge Sort: example



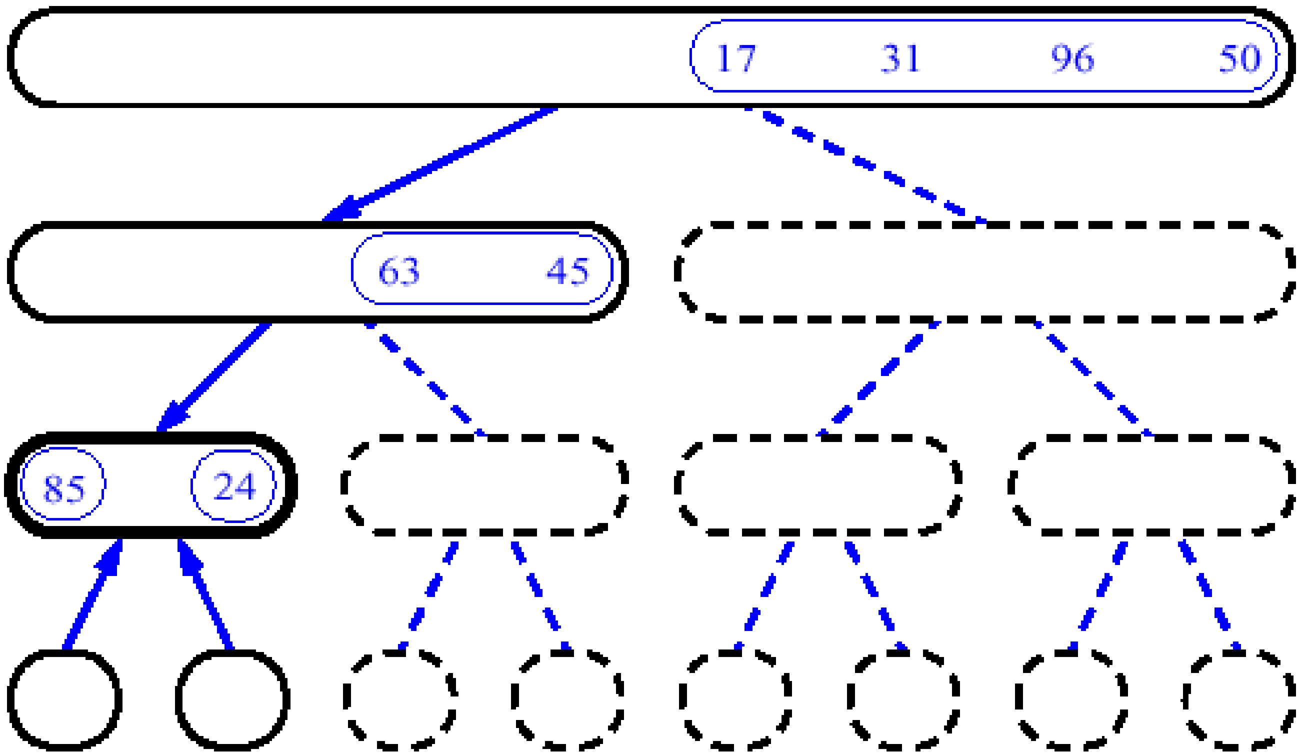
Merge Sort: example



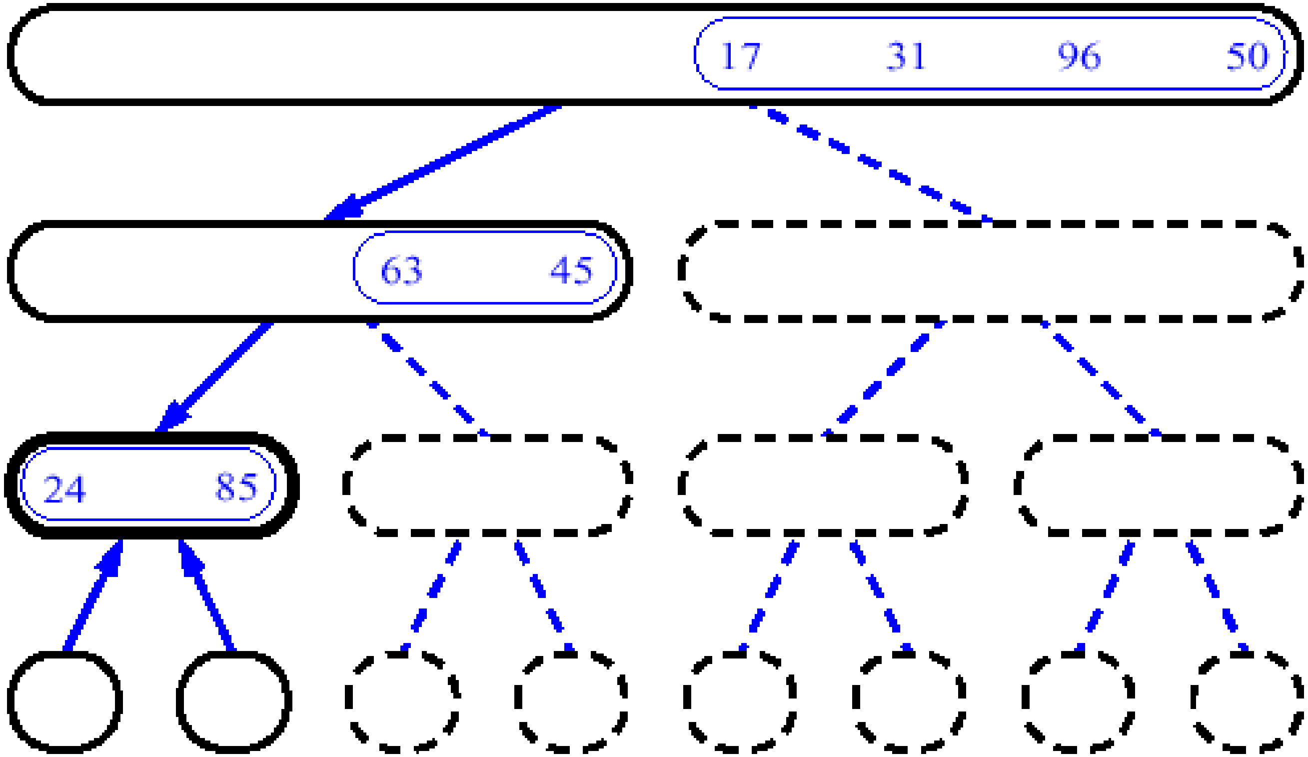
Merge Sort: example



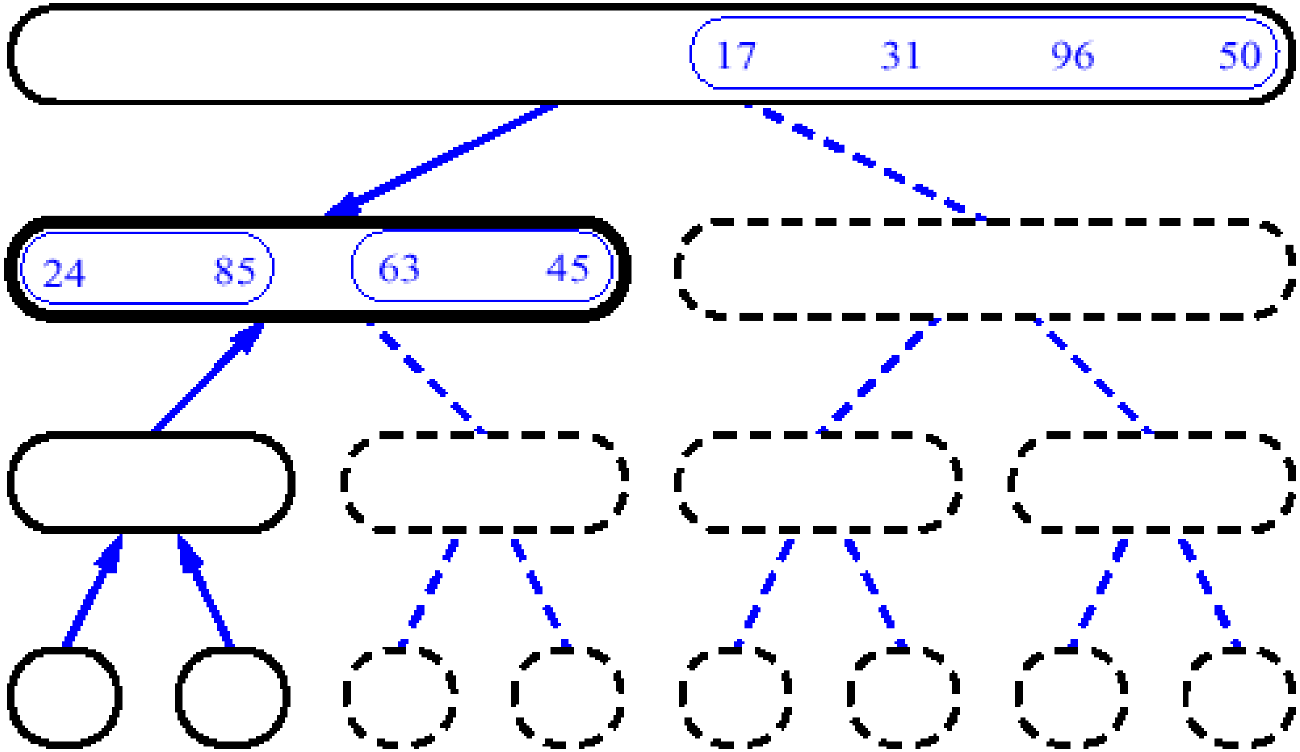
Merge Sort: example



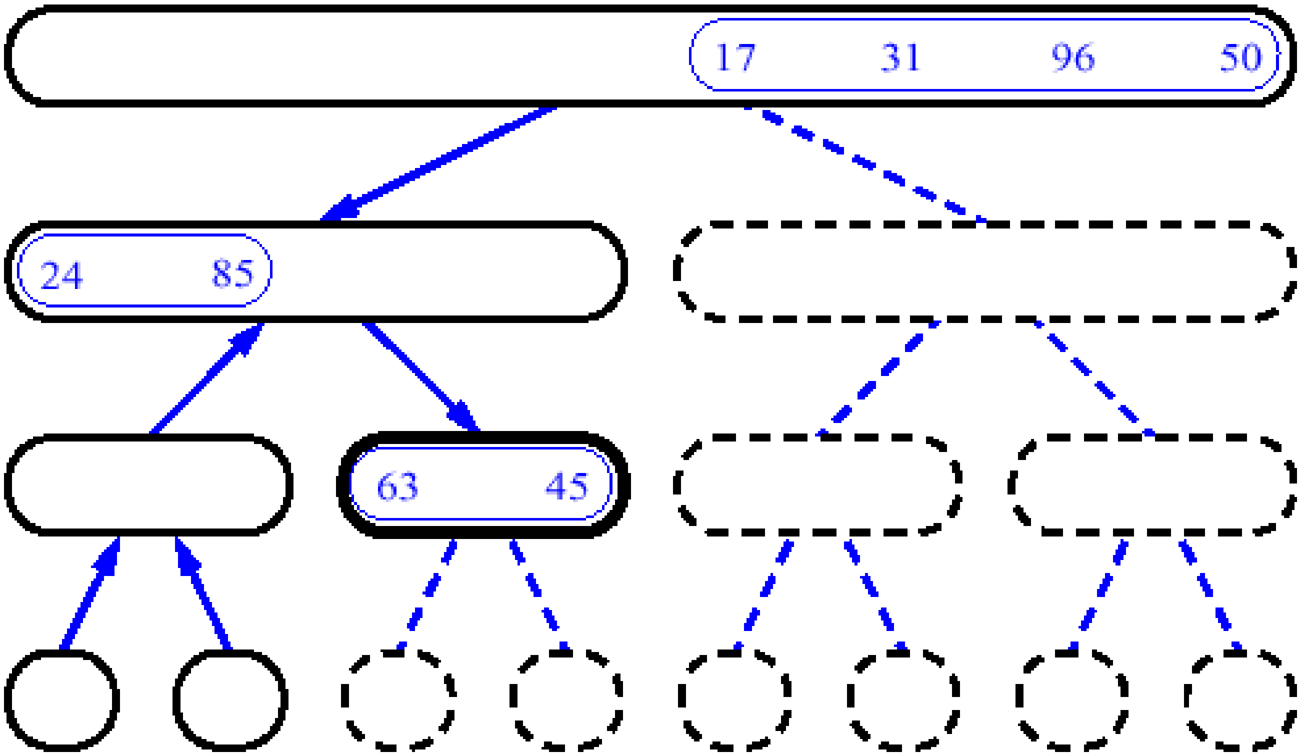
Merge Sort: example



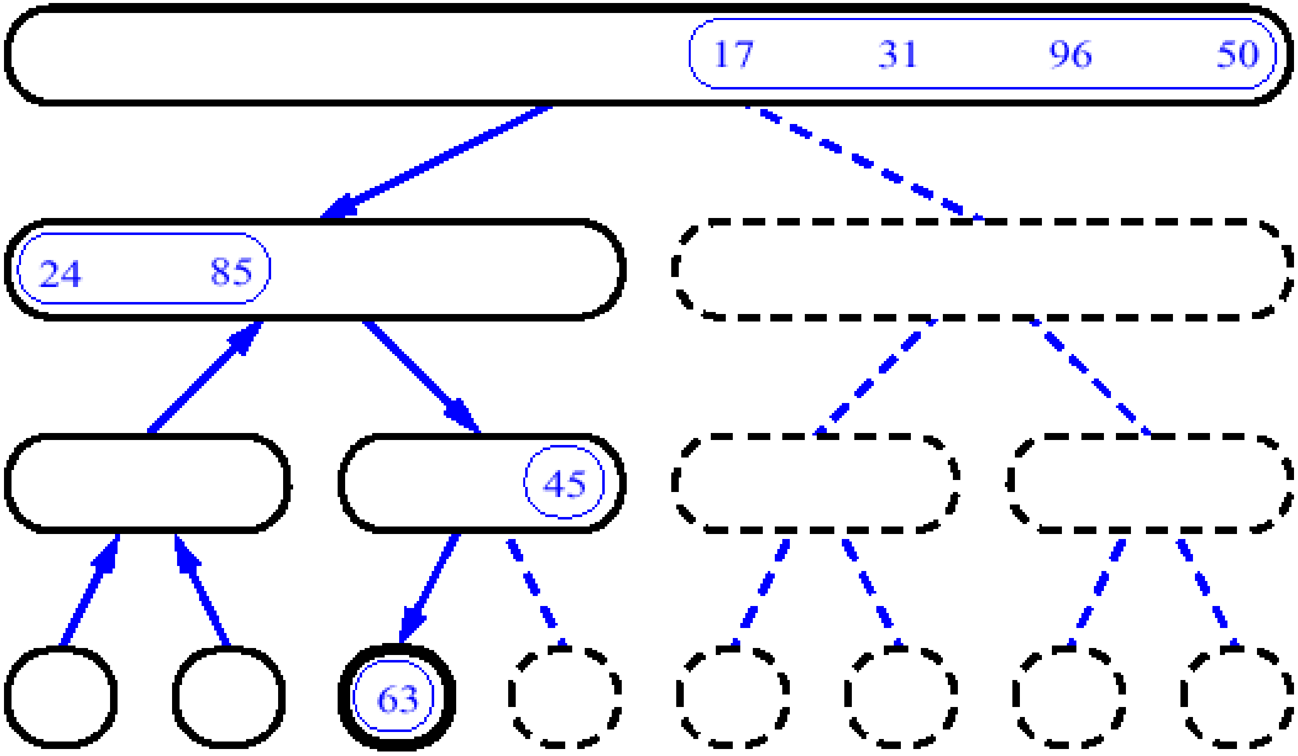
Merge Sort: example



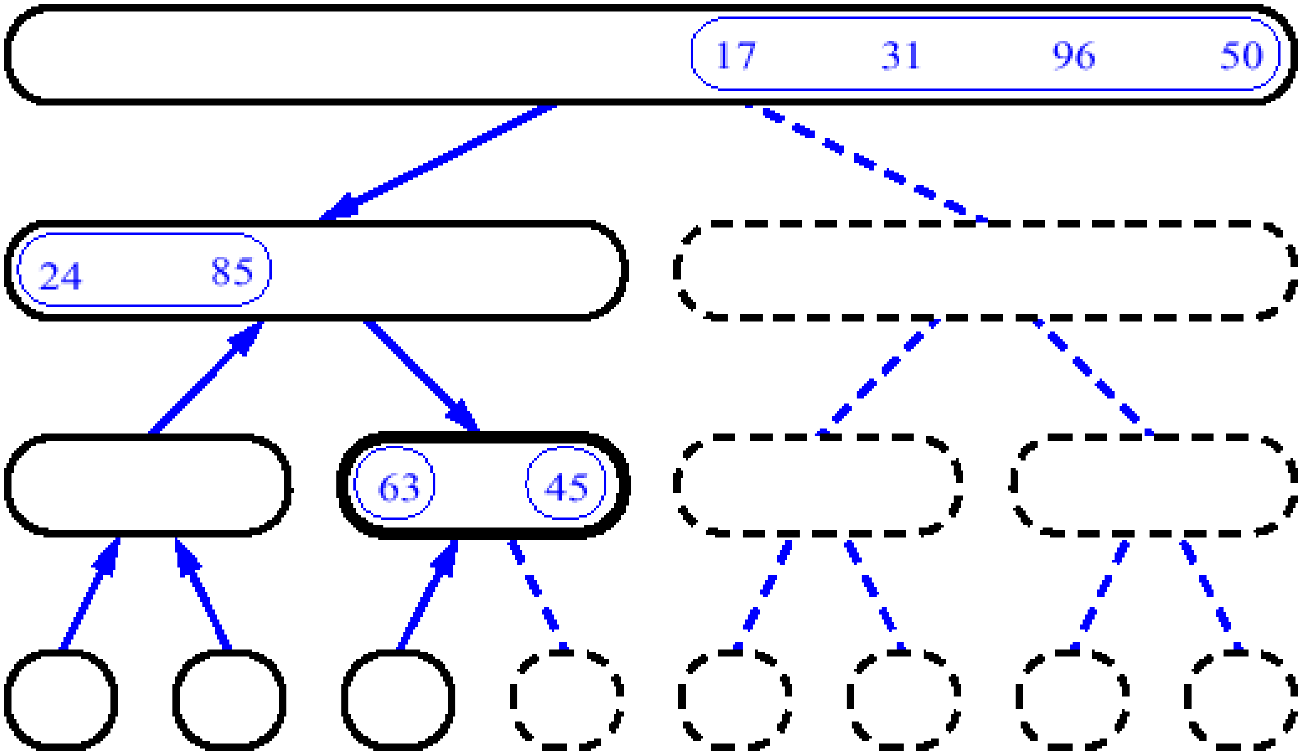
Merge Sort: example



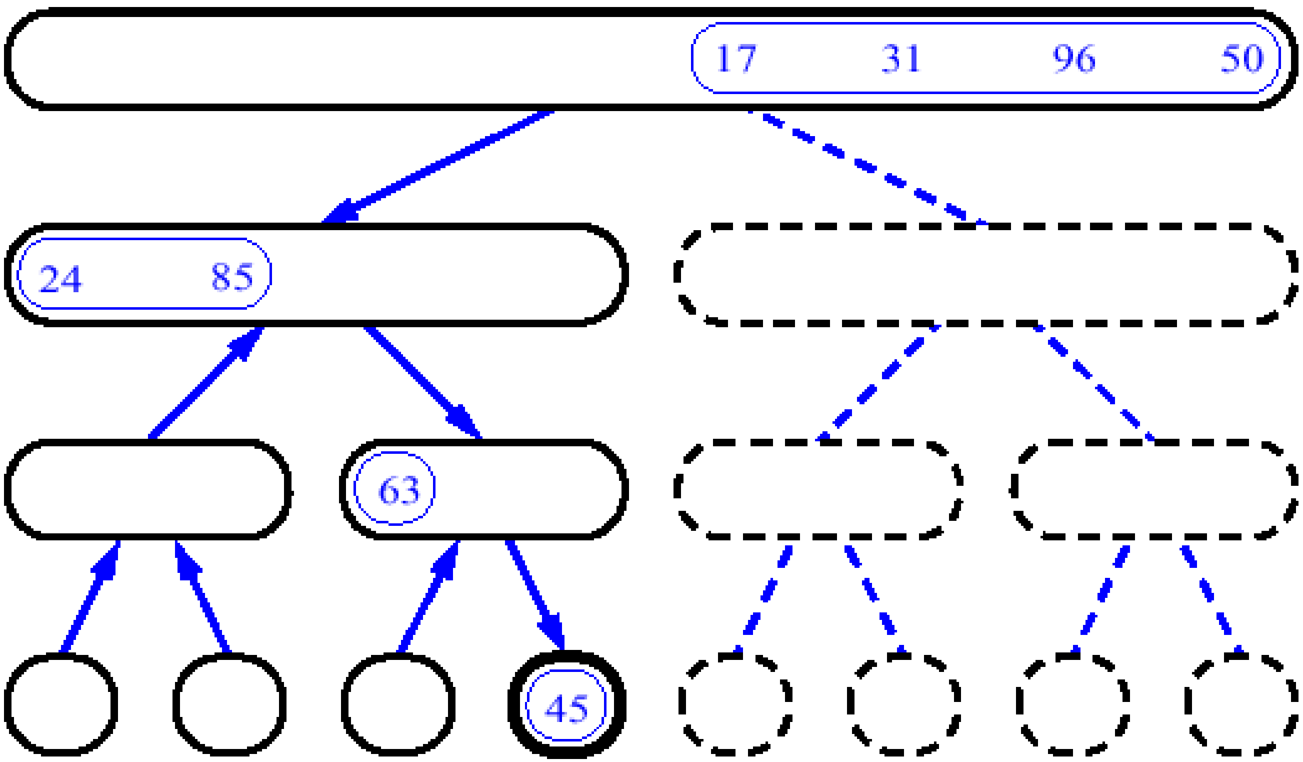
Merge Sort: example



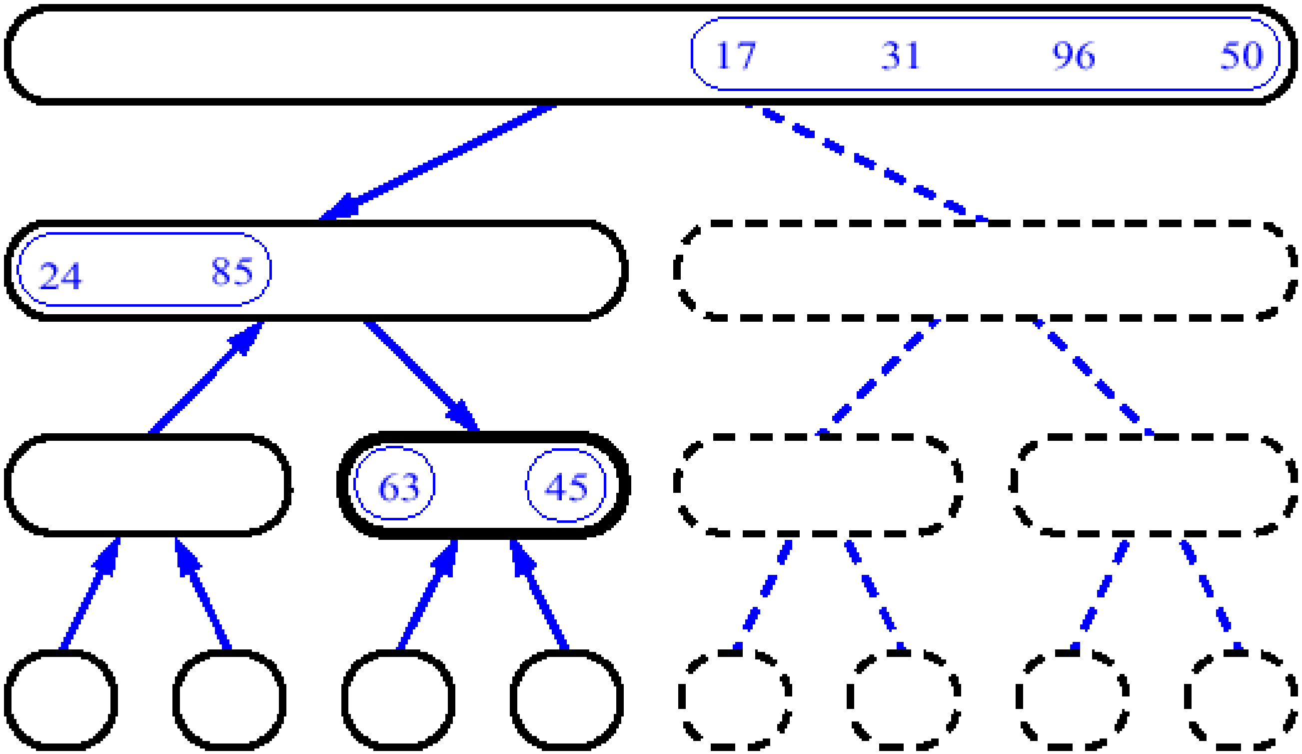
Merge Sort: example



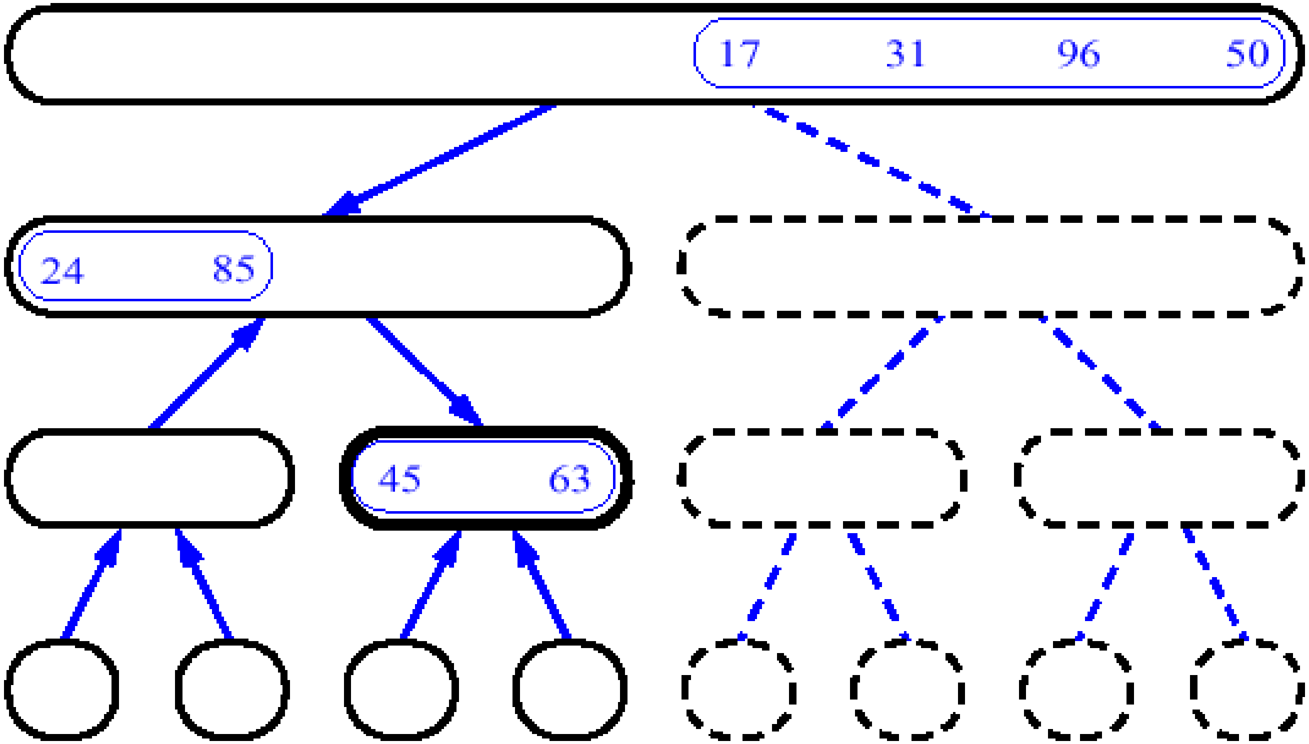
Merge Sort: example



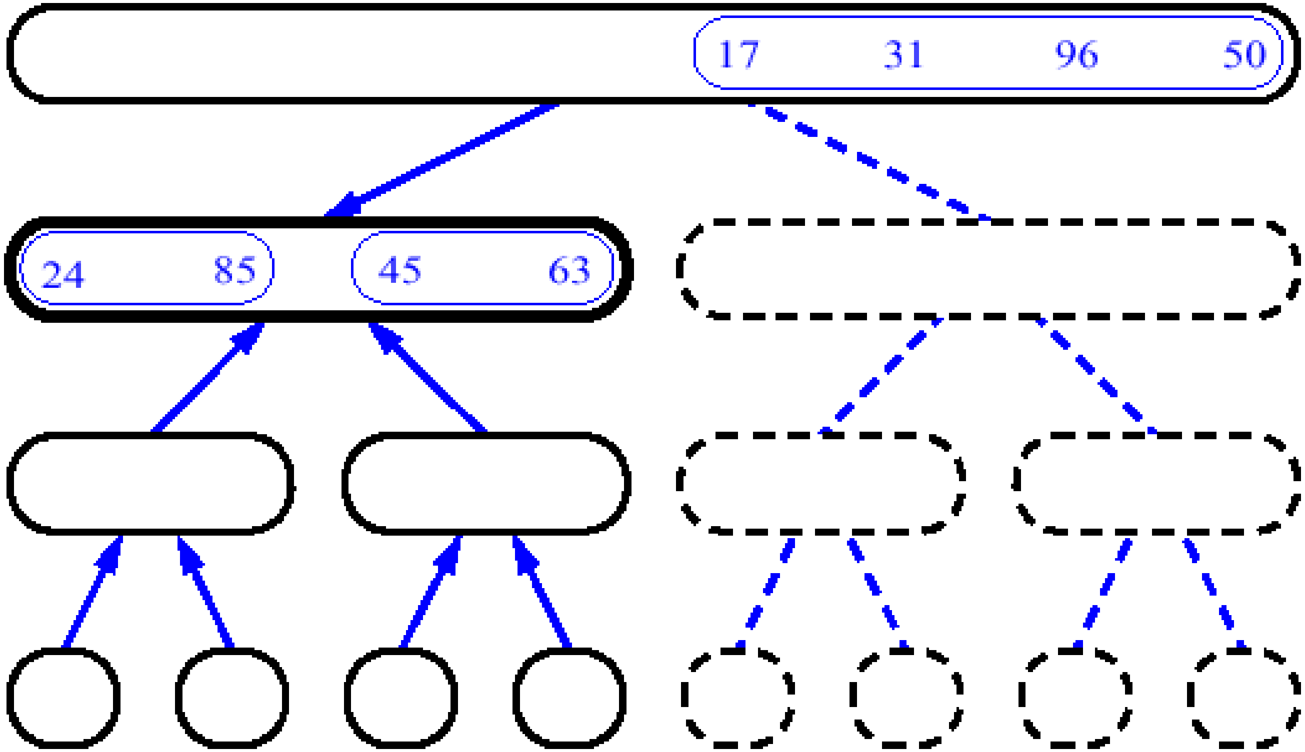
Merge Sort: example



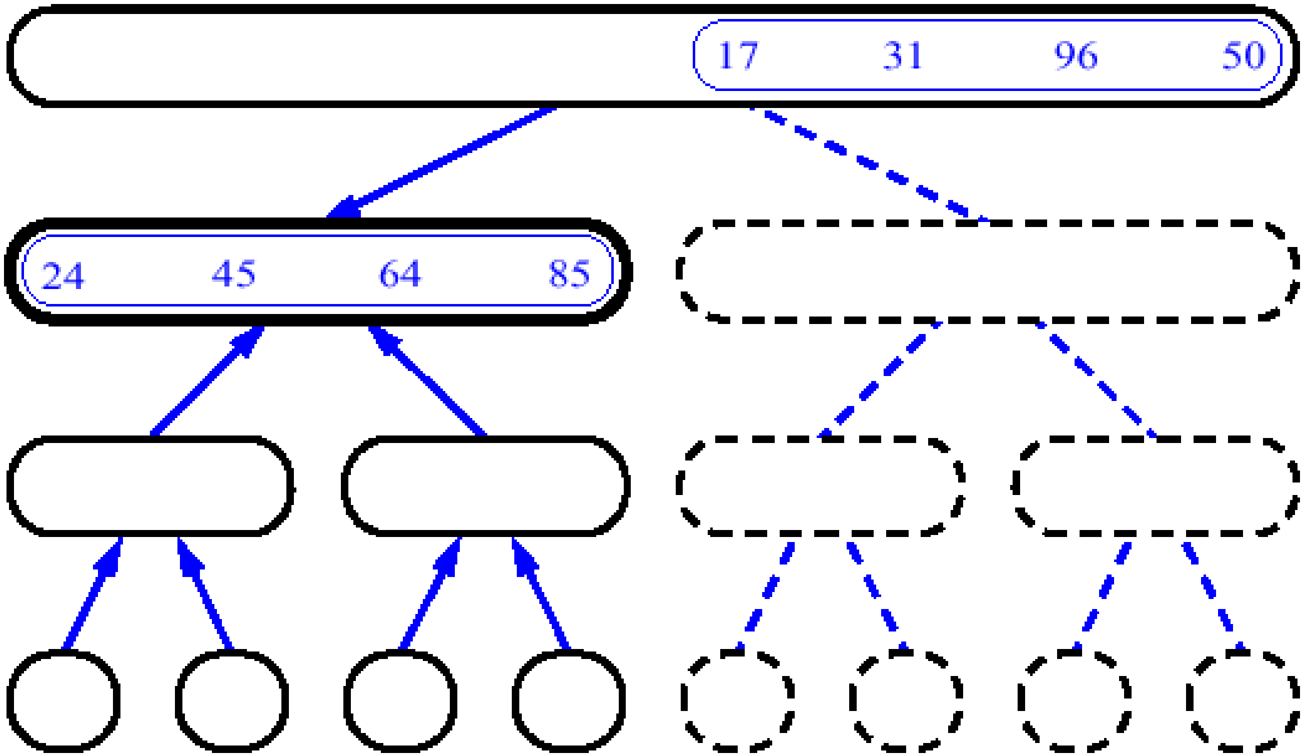
Merge Sort: example



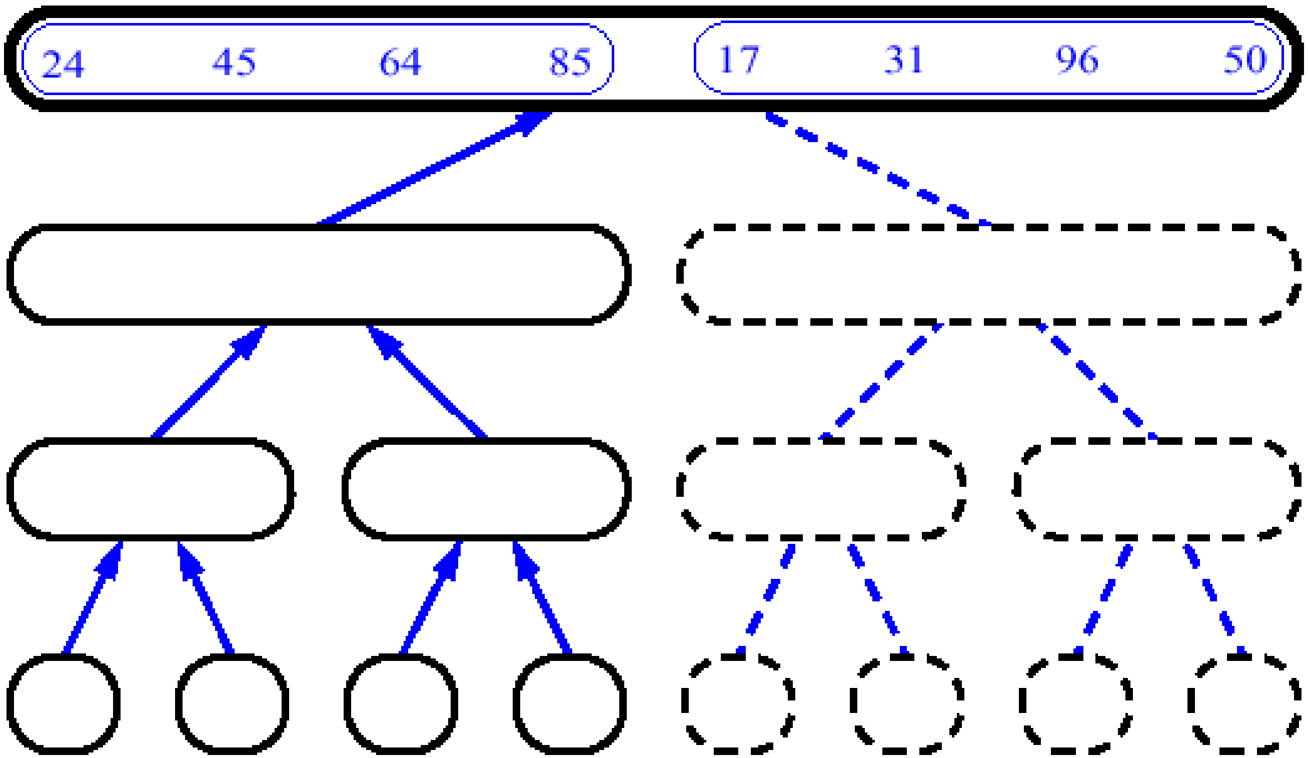
Merge Sort: example



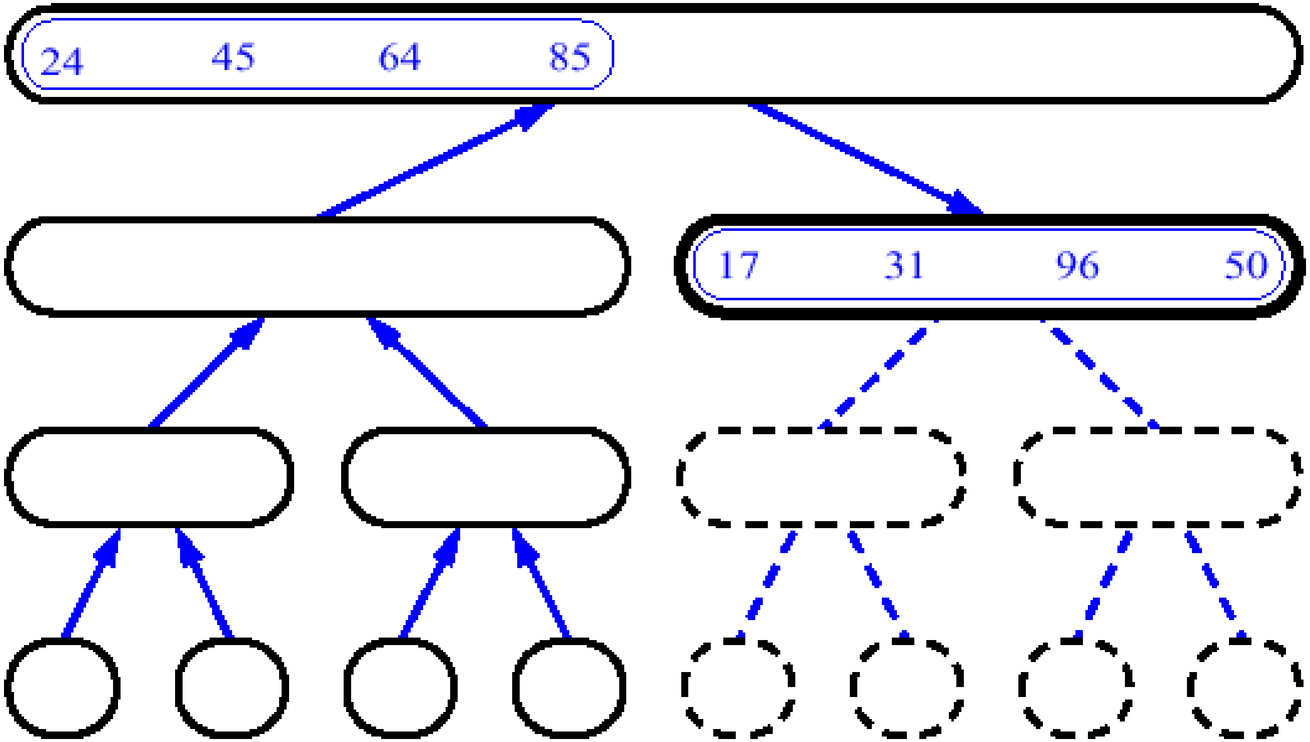
Merge Sort: example



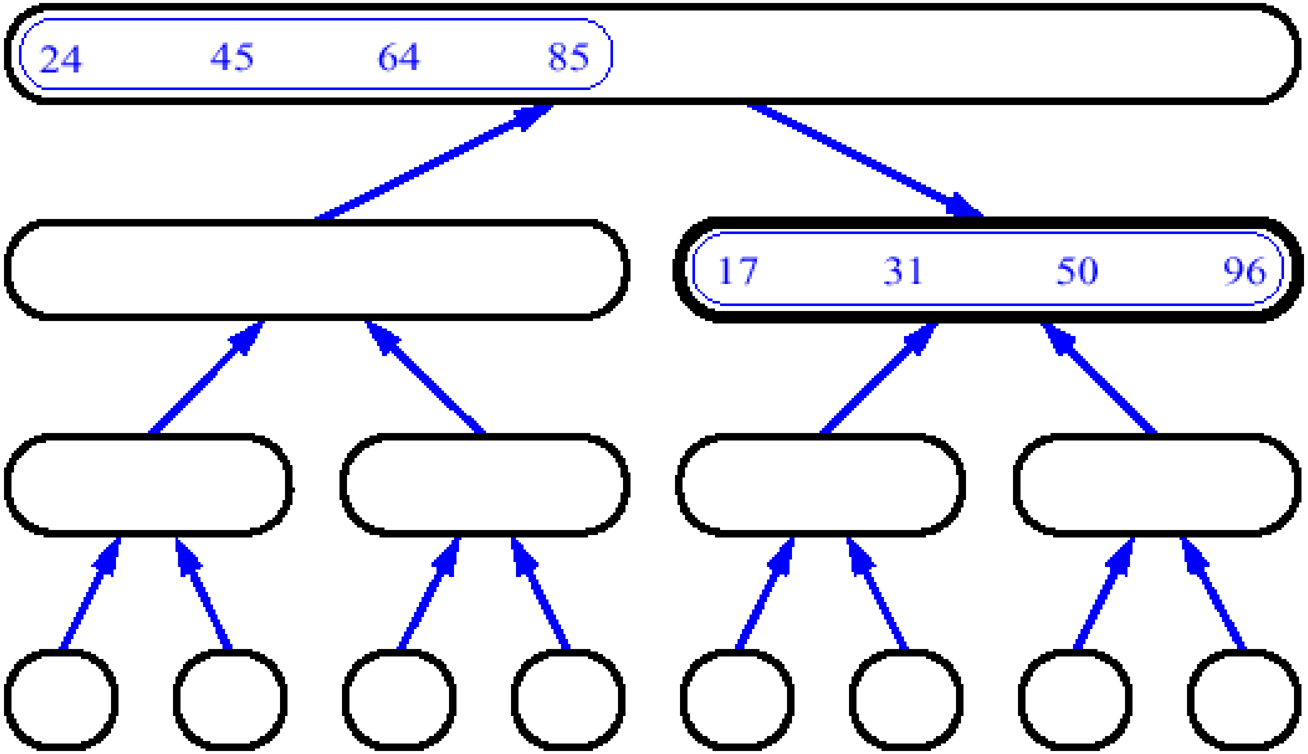
Merge Sort: example



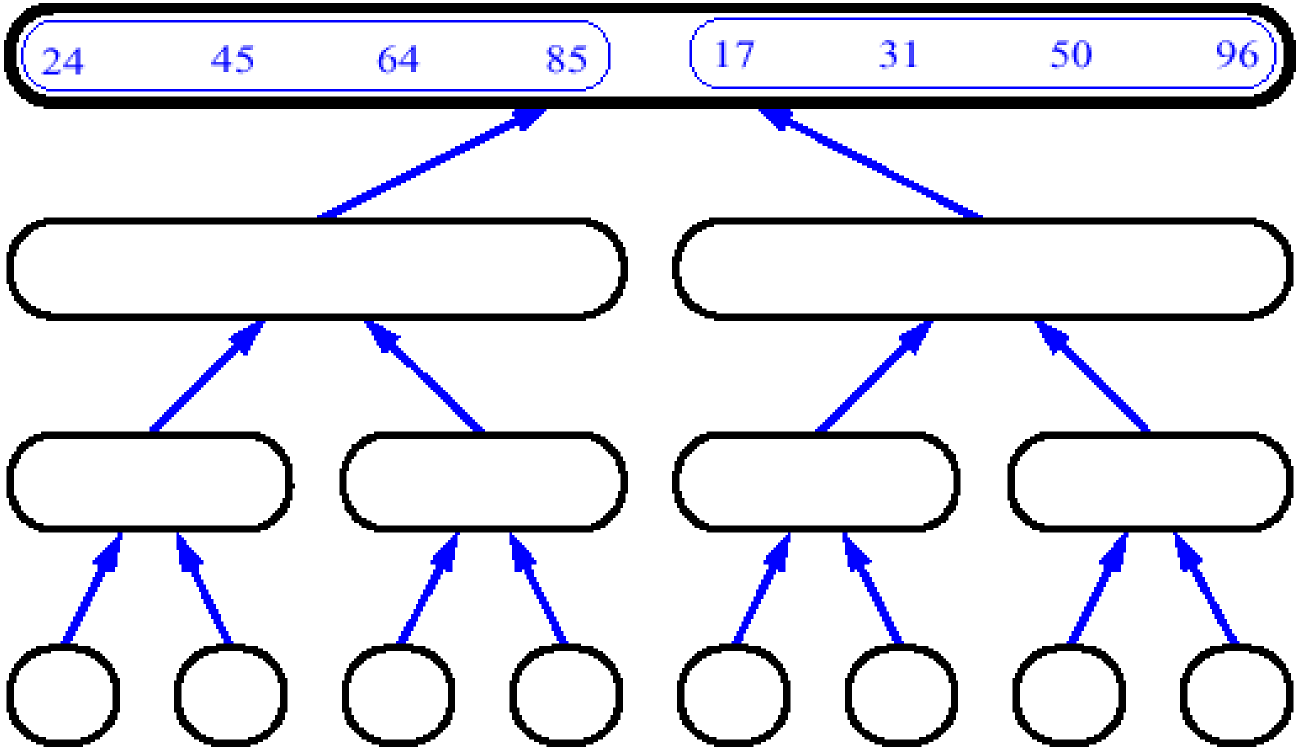
Merge Sort: example



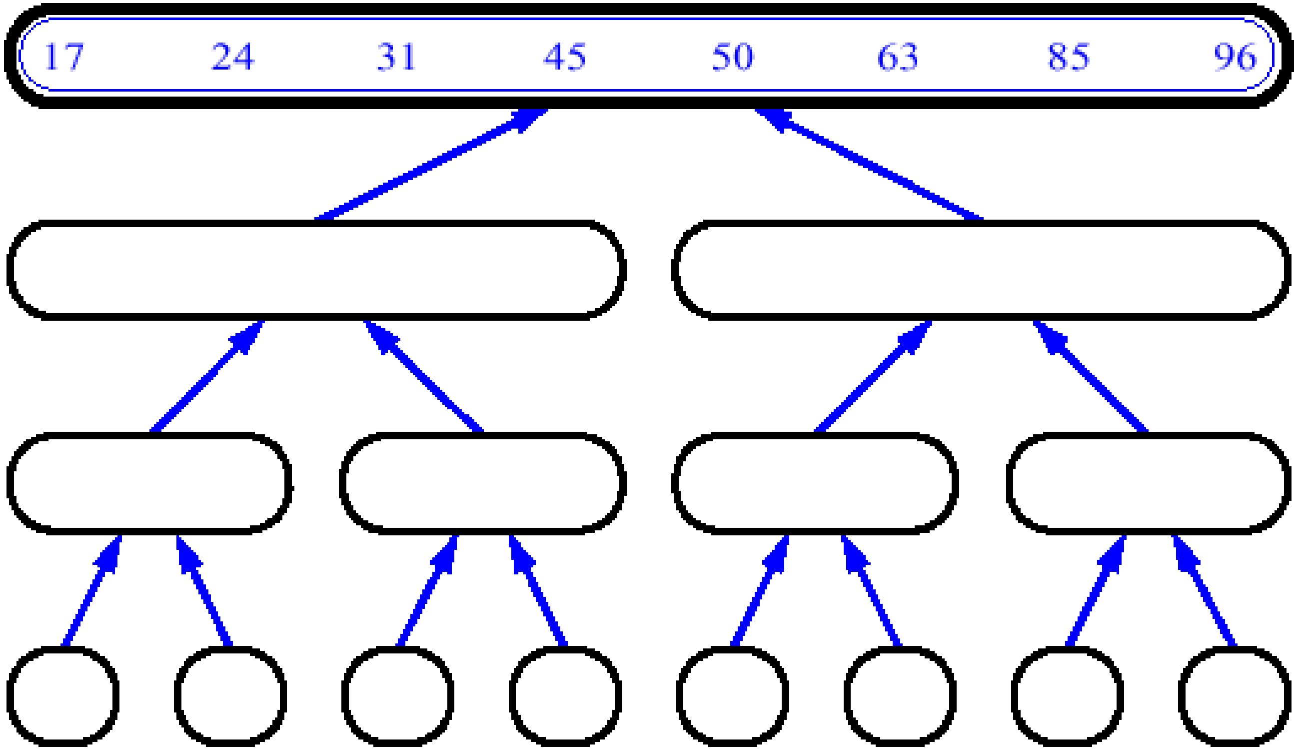
Merge Sort: example



Merge Sort: example

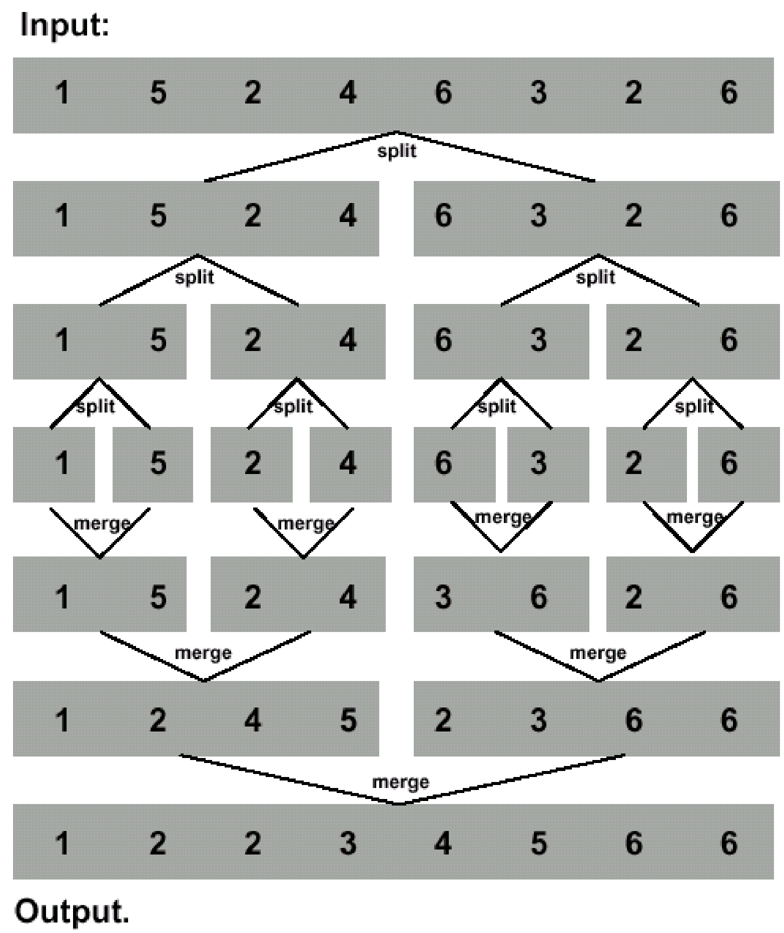


Merge Sort: example



Merge Sort: summary

- To sort *n* numbers
  - if n=1 done!
  - recursively sort 2 lists of numbers $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$ elements
  - merge 2 sorted lists in $\Theta(n)$ time
- Strategy
  - break problem into similar (smaller) subproblems
  - recursively solve subproblems
  - combine solutions to answer



Recurrences

- Running times of algorithms with **Recursive calls** can be described using recurrences
- A **recurrence** is an equation or inequality that describes a function in terms of its value on smaller inputs

Example: Merge Sort

$$T(n) = \begin{cases} \text{solving\_trivial\_problem} & \text{if } n = 1 \\ \text{num\_pieces } T(n/\text{subproblem\_size\_factor}) + \text{dividing} + \text{combining} & \text{if } n > 1 \end{cases}$$

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

Solving recurrences

- Repeated substitution method
  - Expanding the recurrence by substitution and noticing patterns
- Substitution method
  - guessing the solutions
  - verifying the solution by the mathematical induction
- Recursion-trees
- Master method
  - templates for different classes of recurrences

Repeated Substitution Method

- The procedure is straightforward:
  - Substitute
  - Expand
  - Substitute
  - Expand
  - …
  - Observe a pattern and write how your expression looks after the i-th substitution
  - Find out what the value of i (e.g., lg n) should be to get the base case of the recurrence (say $T(1)$)
  - Insert the value of $T(1)$ and the expression of i into your expression

Master Method

- The idea is to solve a class of recurrences that have the form

$$T(n) = aT(n/b) + f(n)$$

- a $\geq$ 1 and b > 1, and f is asymptotically positive!
- Abstractly speaking, T(n) is the runtime for an algorithm and we know that
  - a subproblems of size n/b are solved recursively, each in time T(n/b)
  - f(n) is the cost of dividing the problem and combining the results. In merge-sort

$$T(n) = 2T(n/2) + \Theta(n)$$

Master Theorem Summarized

- Given a recurrence of the form $T(n) = aT(n/b) + f(n)$

1. $f(n) = O\left(n^{\log_b a - \varepsilon}\right)$

   $\Rightarrow T(n) = \Theta\left(n^{\log_b a}\right)$

2. $f(n) = \Theta\left(n^{\log_b a}\right)$

   $\Rightarrow T(n) = \Theta\left(n^{\log_b a} \lg n\right)$

3. $f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right)$ and $af(n/b) \le cf(n)$, for some $c < 1, n > n_0$

   $\Rightarrow T(n) = \Theta\left(f(n)\right)$

- The master method cannot solve every recurrence of this form; there is a gap between cases 1 and 2, as well as cases 2 and 3

Using the Master Theorem

- Extract a, b, and f(n) from a given recurrence
- Determine $n^{\log_b a}$
- Compare f(n) and $n^{\log_b a}$ asymptotically
- Determine appropriate MT case, and apply
- Example merge sort

$$T(n) = 2T(n/2) + \Theta(n)$$

$$a = 2,\ b = 2;\ n^{\log_b a} = n^{\log_2 2} = n = \Theta(n)$$

$$\text{Also } f(n) = \Theta(n)$$

$$\Rightarrow \text{Case 2: } T(n) = \Theta\left(n^{\log_b a} \lg n\right) = \Theta\left(n \lg n\right)$$

Examples

$$T(n) = T(n/2) + 1$$

$$a = 1, b = 2;\ n^{\log_2 1} = 1$$

also $f(n) = 1,\ f(n) = \Theta(1)$

$\Rightarrow$ Case 2: $T(n) = \Theta(\lg n)$

```
Binary-search(A, p, r, s):
   q←(p+r)/2
   if A[q]=s then return q
   else if A[q]>s then
        Binary-search(A, p, q-1, s)
   else Binary-search(A, q+1, r, s)
```

$$T(n) = 9T(n/3) + n$$

$$a = 9, b = 3;$$

$f(n) = n,\ f(n) = O(n^{\log_3 9-\varepsilon})$ with $\varepsilon = 1$

$\Rightarrow$ Case 1: $T(n) = \Theta\left(n^2\right)$

Examples

$$T(n) = 4T(n/2) + n^3$$

$$a = 4, b = 2;\ n^{\log_2 4} = n^2$$

$$f(n) = n^3;\ f(n) = \Omega(n^2)$$

$\Rightarrow$ Case 3: $T(n) = \Theta\left(n^3\right)$

Checking the regularity condition

$$4f(n/2) \le cf(n)$$

$$4n^3/8 \le cn^3$$

$$n^3/2 \le cn^3$$

$$c = 3/4 < 1$$

Sorting algorithms so far

- Insertion sort, selection sort
  - Worst-case running time $\Theta(n^2)$; in-place
- Merge sort
  - Worst-case running time $\Theta(n \log n)$, but requires additional memory $\Theta(n)$;

Improving Selection sort

```
Selection-Sort(A[1..n]):
   For i → n downto 2
A:    Find the largest element among A[1..i]
B:    Exchange it with A[i]
```

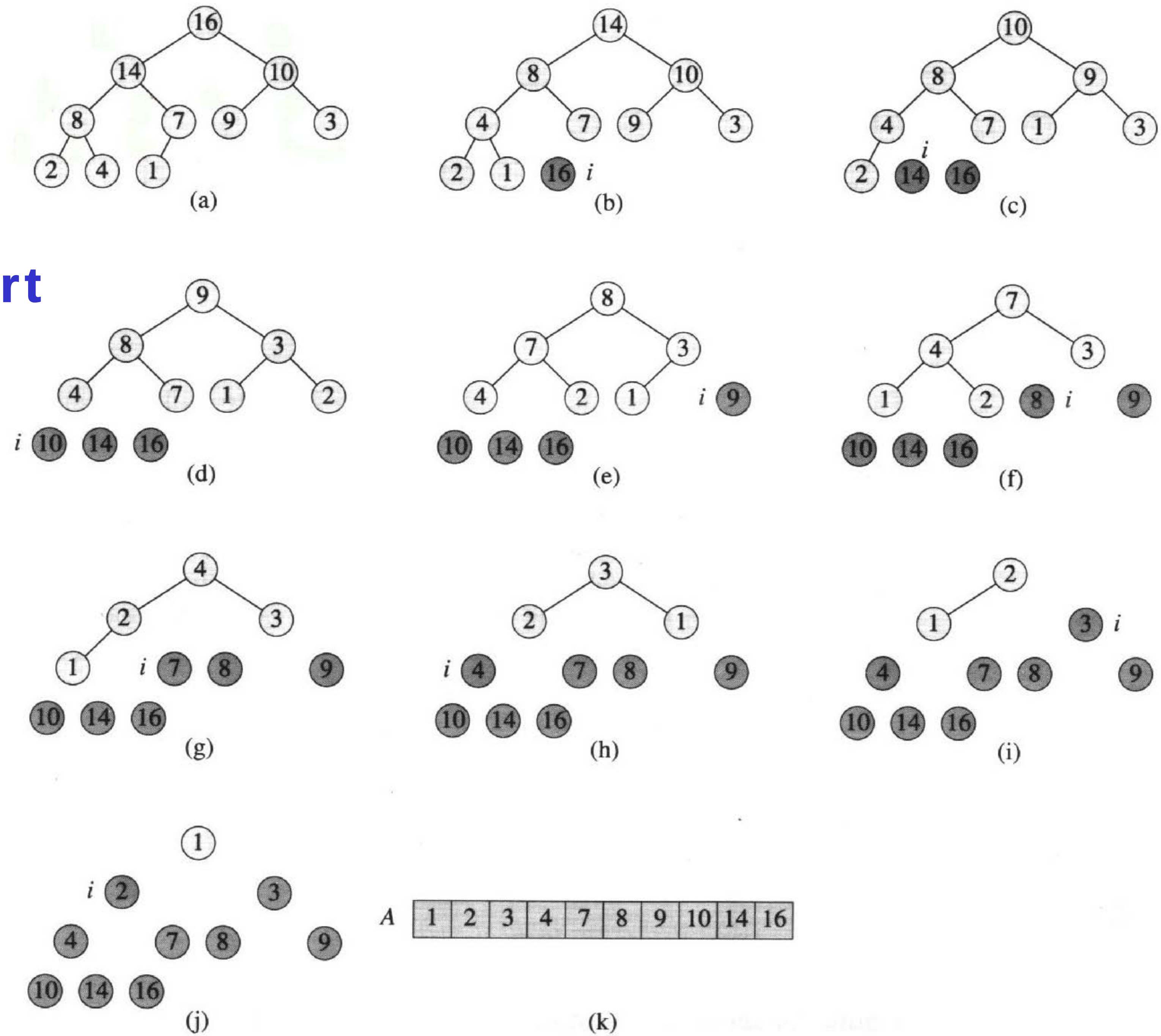
- A takes $\Theta(n)$ and B takes $\Theta(1)$: $\Theta(n^2)$ in total
- One idea for improvement: use a *data structure (heap)*, to do both A and B in *O(lg n)* time, balancing the work, achieving a better trade-off, and a total running time *O(n log n)*.
- We have already seen how heap-sort does this.

Heap sort

$\text{HEAPSORT}(A)$	**Analysis**
1 $\text{BUILD-HEAP}(A)$	?? O(n)
2 **for** $i \leftarrow n$ **downto** 2	n times
3 **do** exchange $A[1] \leftrightarrow A[i]$	O(1)
4 $n \leftarrow n - 1$	O(1)
5 $\text{HEAPIFY}(A, 1)$	O($\lg n$)

The total running time of heap sort is O(n lg n)
+ Build-Heap(A) time, which is O(n)

Heap sort

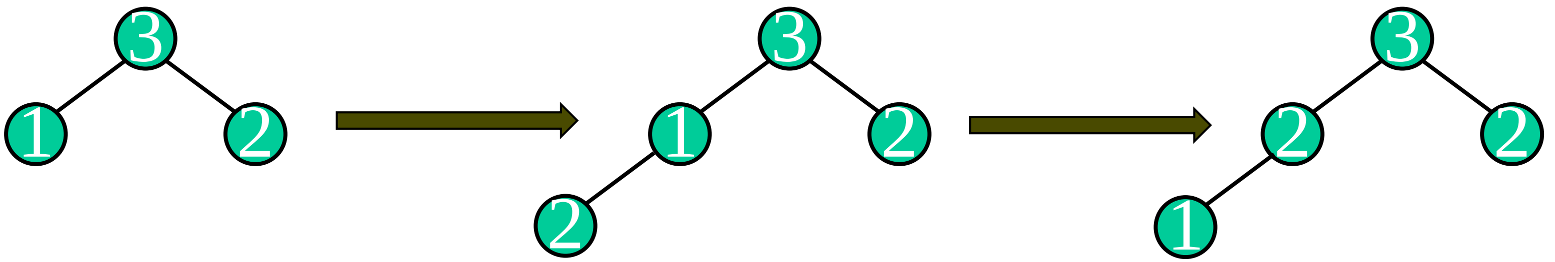


Heap Sort: Summary

- Heap sort uses a heap data structure to improve selection sort and make the running time asymptotically optimal
- Running time is O(n log n) – like merge sort, but unlike selection, insertion, or bubble sorts
- Sorts in place – like insertion, selection or bubble sorts, but unlike merge sort

Heap Sort is Not Stable

Example (MaxHeap)



Quick Sort

- Characteristics
  - sorts in place, i.e., does not require an additional array, like insertion sort
  - Divide-and-conquer, like merge sort
  - very practical, average sort performance O(n log n) (with small constant factors), but worst case $O(n^2)$
  - May be randomized (GTG gives this version)
  - The first GTG version is not in-place

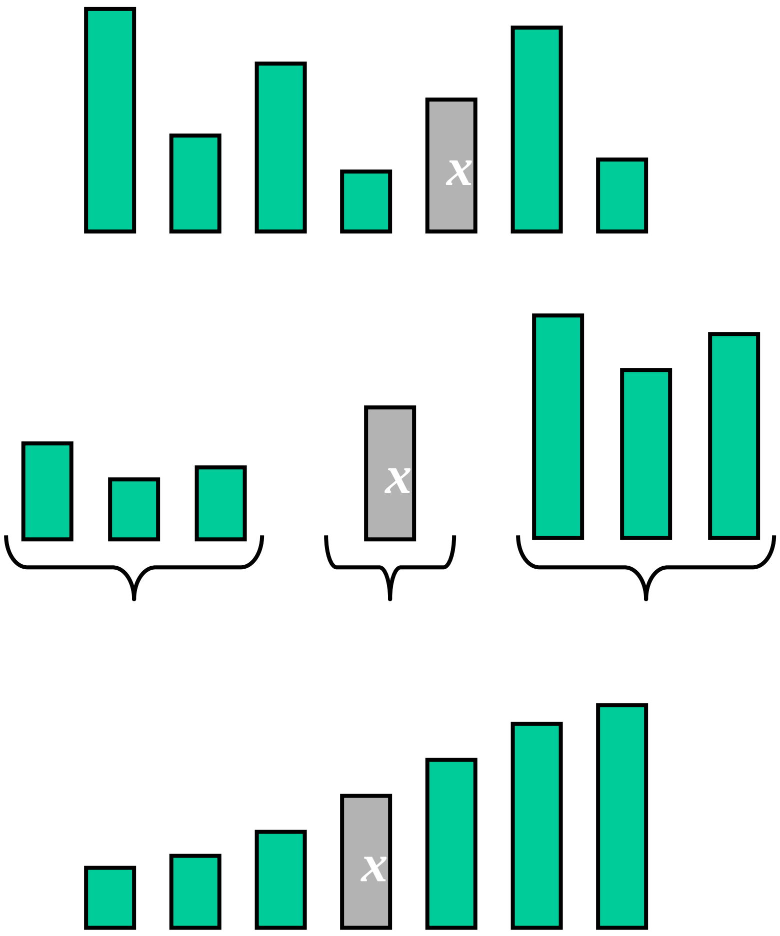
Quick Sort – the main idea

- To understand quick-sort, let’s look at a high-level description of the algorithm
- A divide-and-conquer algorithm
  - Divide: partition array into 2 subarrays such that elements in the lower part <= elements in the higher part
  - Conquer: recursively sort the 2 subarrays
  - Combine: trivial since sorting is done in place

Quick-Sort (GTG)

Quick-sort is a randomized sorting algorithm based on the divide-and-conquer paradigm:

1. Divide: pick a random element x (called pivot) and partition S into
   - L elements less than x
   - E elements equal x
   - G elements greater than x
2. Recur: sort L and G
3. Conquer: join L, E and G



Partition

- We partition an input sequence as follows:
  - We remove, in turn, each element y from S and
  - We insert y into L, E or G, depending on the result of the comparison with the pivot x
- Each insertion and removal is at the beginning or at the end of a sequence, and hence takes $O(1)$ time
- Thus, the partition step of quick-sort takes $O(n)$ time

Algorithm *partition*(S, p)
  Input sequence S, position p of pivot
  Output subsequences L, E, G of the elements of S less than, equal to, or greater than the pivot, resp.
  $L, E, G \leftarrow$ empty sequences
  $x \leftarrow S.remove(p)$
  while $\neg S.isEmpty()$
    $y \leftarrow S.remove(S.first())$
    if $y < x$
      $L.addLast(y)$
    else if $y = x$
      $E.addLast(y)$
    else $\{ y > x \}$
      $G.addLast(y)$
  return L, E, G

Java Implementation

```
/** Quick-sort contents of a queue. */
public static <K> void quickSort(Queue<K> S, Comparator<K> comp) {
  int n = S.size();
  if (n < 2) return;                          // queue is trivially sorted
  // divide
  K pivot = S.first();                        // using first as arbitrary pivot
  Queue<K> L = new LinkedQueue<>();
  Queue<K> E = new LinkedQueue<>();
  Queue<K> G = new LinkedQueue<>();
  while (!S.isEmpty()) {                      // divide original into L, E, and G
    K element = S.dequeue();
    int c = comp.compare(element, pivot);
    if (c < 0)                                // element is less than pivot
      L.enqueue(element);
    else if (c == 0)                          // element is equal to pivot
      E.enqueue(element);
    else                                      // element is greater than pivot
      G.enqueue(element);
  }
  // conquer
  quickSort(L, comp);                         // sort elements less than pivot
  quickSort(G, comp);                         // sort elements greater than pivot
  // concatenate results
  while (!L.isEmpty())
    S.enqueue(L.dequeue());
  while (!E.isEmpty())
    S.enqueue(E.dequeue());
  while (!G.isEmpty())
    S.enqueue(G.dequeue());
}
```

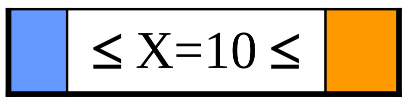
In place, deterministic (non-randomized) Quicksort

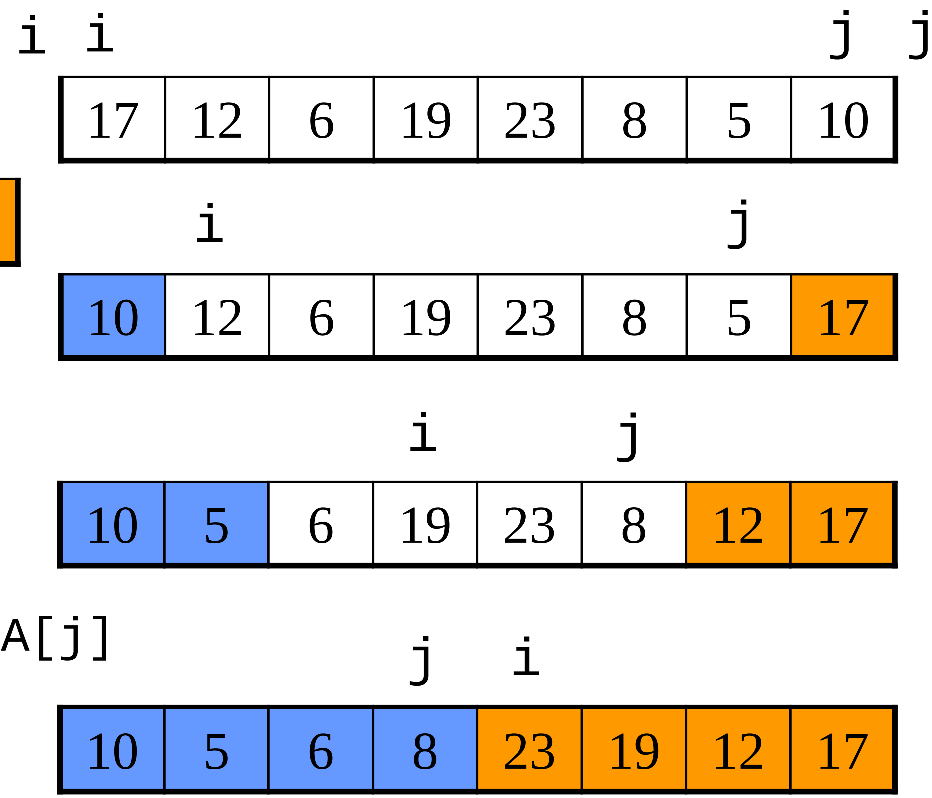
- Choose a pivot deterministically: say the last element of the array
- Define partition to be in place

Partitioning

- Linear time partitioning procedure

```
Partition(A,p,r)
x←A[r]
i←p-1
j←r+1
while TRUE
   repeat j←j-1
      until A[j] ≤x
   repeat i←i+1
      until A[i] ≥x
   if i<j
      then exchange A[i]↔A[j]
      else return j
```





Quick Sort Algorithm

- Initial call Quicksort(A, 1, length[A])

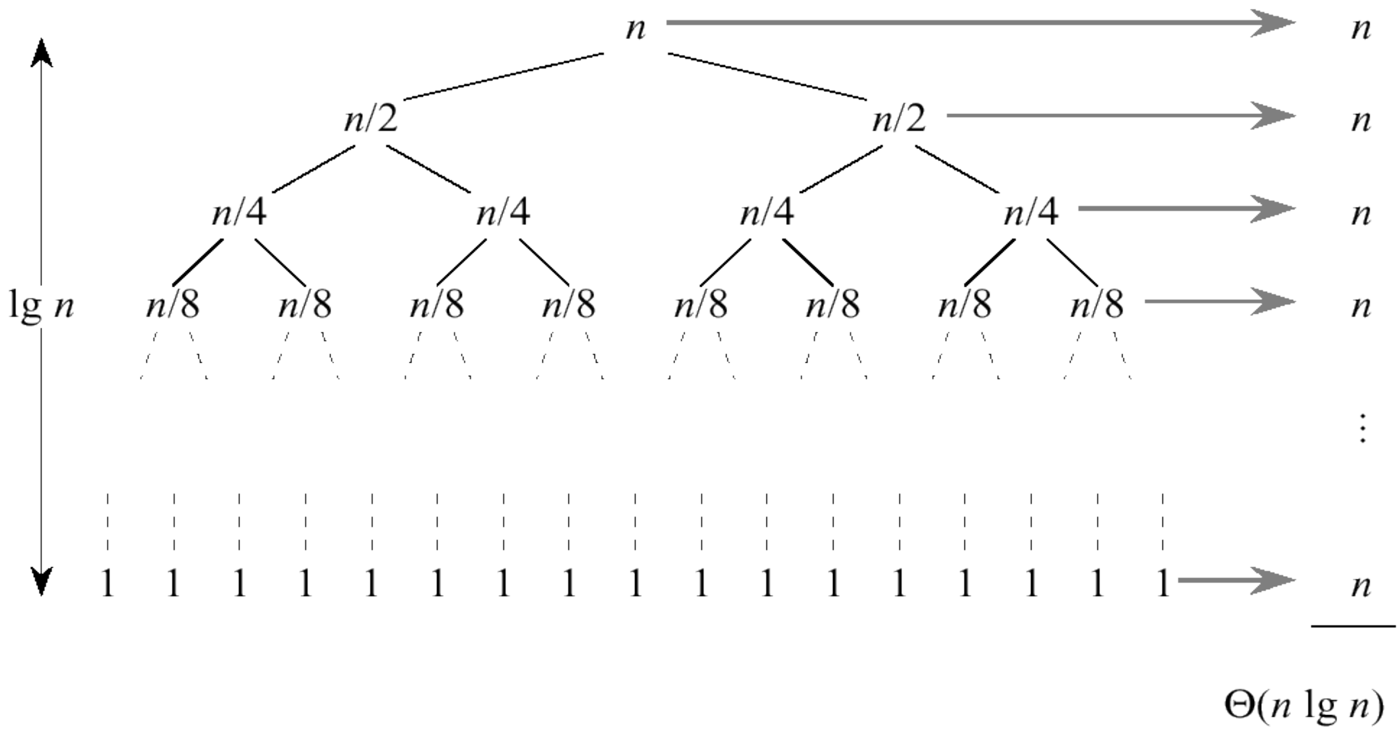
```
Quicksort(A,p,r)
if p<r
   then q←Partition(A,p,r)
        Quicksort(A,p,q)
        Quicksort(A,q+1,r)
```

Analysis of Quicksort

- Assume that all input elements are distinct
- The running time depends on the distribution of splits

Best Case

- If we are lucky, Partition splits the array evenly $T(n) = 2T(n/2) + \Theta(n)$



Using the median as a pivot

- The recurrence in the previous slide works out, BUT……

Q: Can we find the median in linear-time?

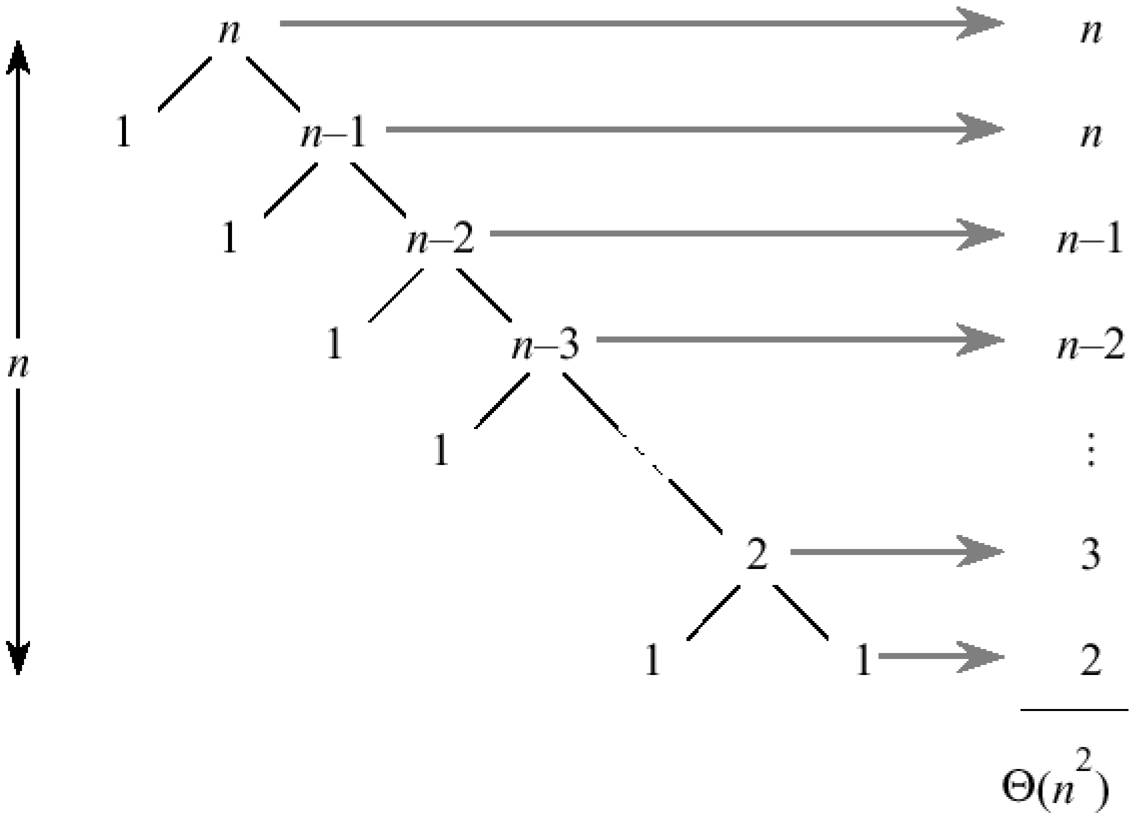
A: YES! But the algorithm is complex and has large constants, and is of limited use in practice

Worst Case

- What is the worst case?
- One side of the parition has only one element

$$
\begin{aligned}
T(n) = T(1) + T(n-1) + \Theta(n) \\
&= T(n-1) + \Theta(n) \\
&= \sum_{k=1}^{n} \Theta(k) \\
&= \Theta\left(\sum_{k=1}^{n} k\right) \\
&= \Theta(n^2)
\end{aligned}
$$

Worst Case (2)



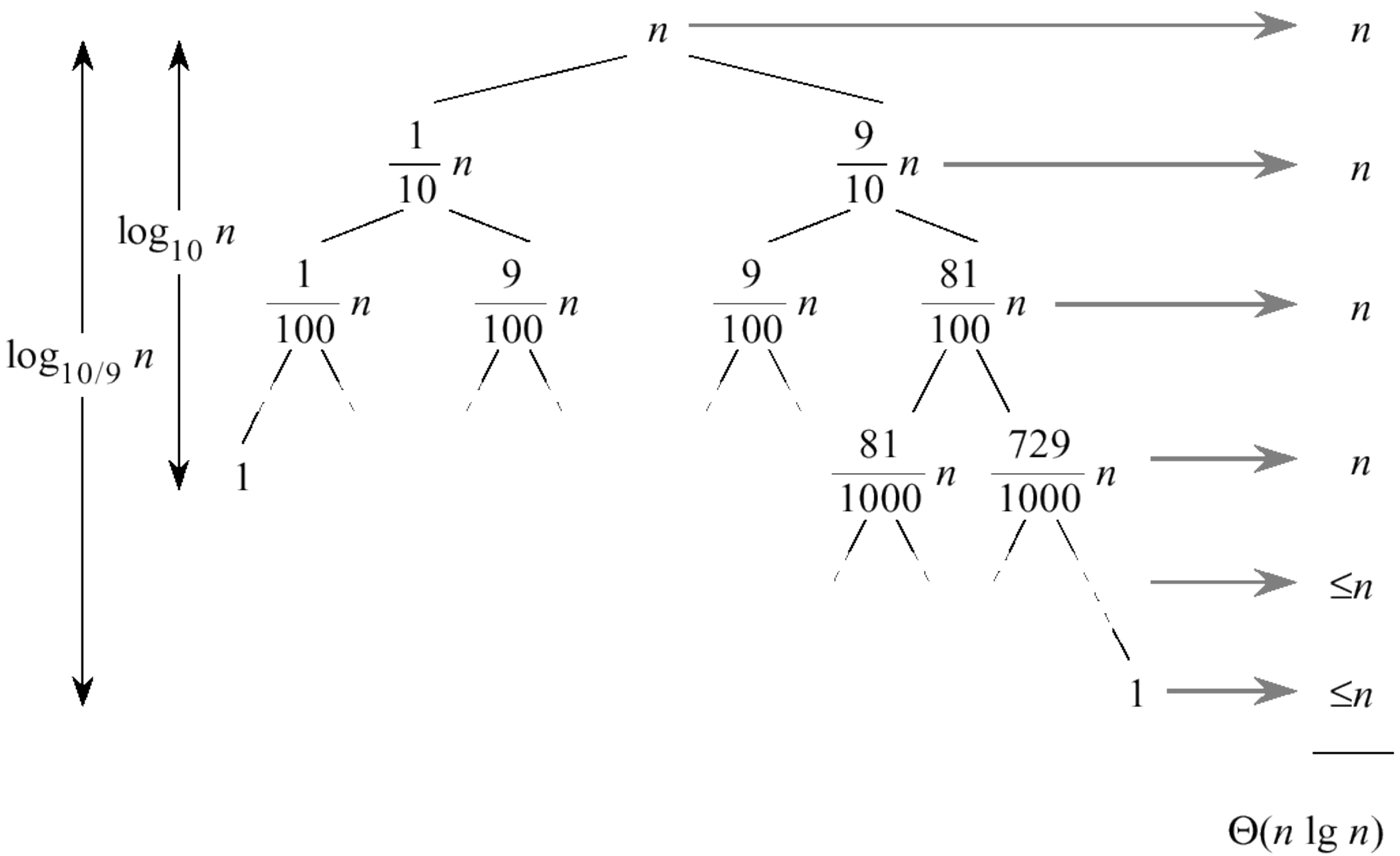
Worst Case (3)

- When does the worst case appear?
  - input is sorted
  - input reverse sorted
- Same recurrence for the worst case of insertion sort
- However, sorted input yields the best case for insertion sort!

Analysis of Quicksort

- Suppose the split is 1/10 : 9/10

$$T(n) = T(n/10) + T(9n/10) + \Theta(n) = \Theta(n \log n)!$$



An Average Case Scenario

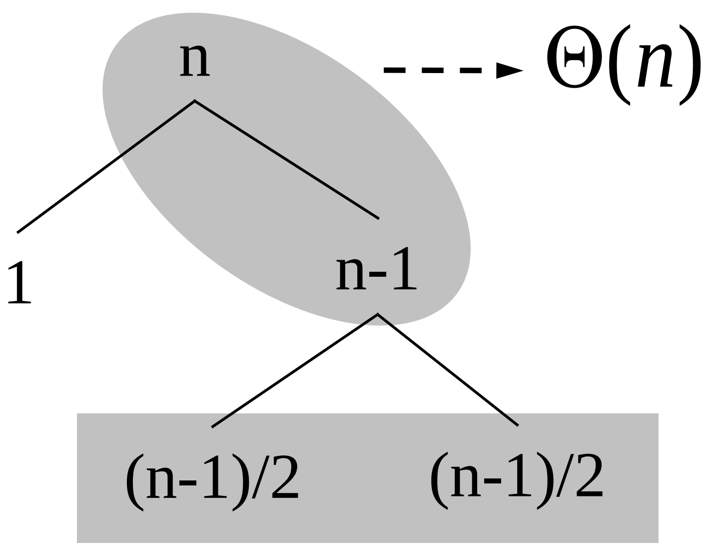
- Suppose, we alternate lucky and unlucky cases to get an average behavior

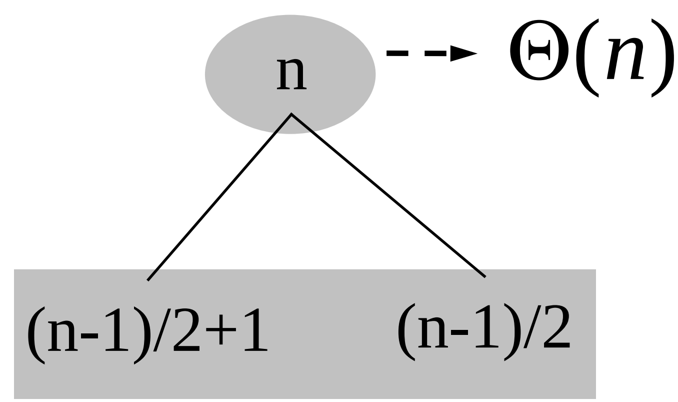
$$L(n) = 2U(n/2) + \Theta(n) \quad \text{lucky}$$

$$U(n) = L(n-1) + \Theta(n) \quad \text{unlucky}$$

we consequently get

$$\begin{aligned} L(n) &= 2(L(n/2-1) + \Theta(n/2)) + \Theta(n) \\ &= 2L(n/2-1) + \Theta(n) \\ &= \Theta(n \log n) \end{aligned}$$





An Average Case Scenario (2)

- How can we make sure that we are usually lucky?
  - Partition around a random element (works well in practice)
- Randomized algorithm
  - running time is independent of the input ordering
  - no specific input triggers worst-case behavior
  - the worst-case is only determined by the output of the random-number generator

Summary of Comparison Sorts

Algorithm	Best Case	Worst Case	Average Case	In Place	Stable	Comments
Selection	n^2	n^2		Yes	Yes	
Bubble	n	n^2		Yes	Yes	Must count swaps for linear best case running time.
Insertion	n	n^2		Yes	Yes	Good if often almost sorted
Merge	n log n	n log n		No	Yes	Good for very large datasets that require swapping to disk
Heap	n log n	n log n		Yes	No	Best if guaranteed n log n required
Quick	n log n	n^2	n log n	Yes	Yes	Usually fastest in practice

But not both!

Next: How fast can we Sort?

We have seen algorithms with worst case $\Theta(n^2)$ and $\Theta(n \log n)$ times.

Can we improve to O(n)?

We cannot do better because we have to read and write *n* elements

Lower bounds: a provable minimum worst case time for a <u>problem</u>

Comparison-based algorithms

- The algorithm only uses the results of comparisons, not values of elements (*).
- Very general – does not assume much about what type of data is being sorted.
- However, other kinds of algorithms are possible!
- In this model, it is reasonable to count #comparisons.
- Note that the #comparisons is a **lower bound** on the running time of an algorithm.

(*) If values are used, lower bounds proved in this model are not lower bounds on the running time.

Points to note

Crucial observations: We must prove our claim about ANY algorithm that only uses comparisons to find the minimum.

Specifically, we <u>made no assumptions about</u>

1. Nature of algorithm.
2. Order or number of comparisons.
3. Optimality of algorithm
4. Whether the algorithm is reasonable – e.g. it could be a very wasteful algorithm, repeating the same comparisons.

Lower bounds for Comparison-based Sorting

Claim: Any comparison-based sorting algorithm must have a worst-case rime of Ω(n log n)

Unfortunate facts:

Lower bounds are usually hard to prove.

Virtually no known general techniques – must try ad hoc methods for each problem.

Lower bounds for comparison-based sorting

- Trivial: $\Omega(n)$ – every element must take part in a comparison.
- Best possible result – $\Omega(n \log n)$ comparisons, since we already know several O(n log n) sorting algorithms.
- Proof is non-trivial: how do we reason about all possible comparison-based sorting algorithms?

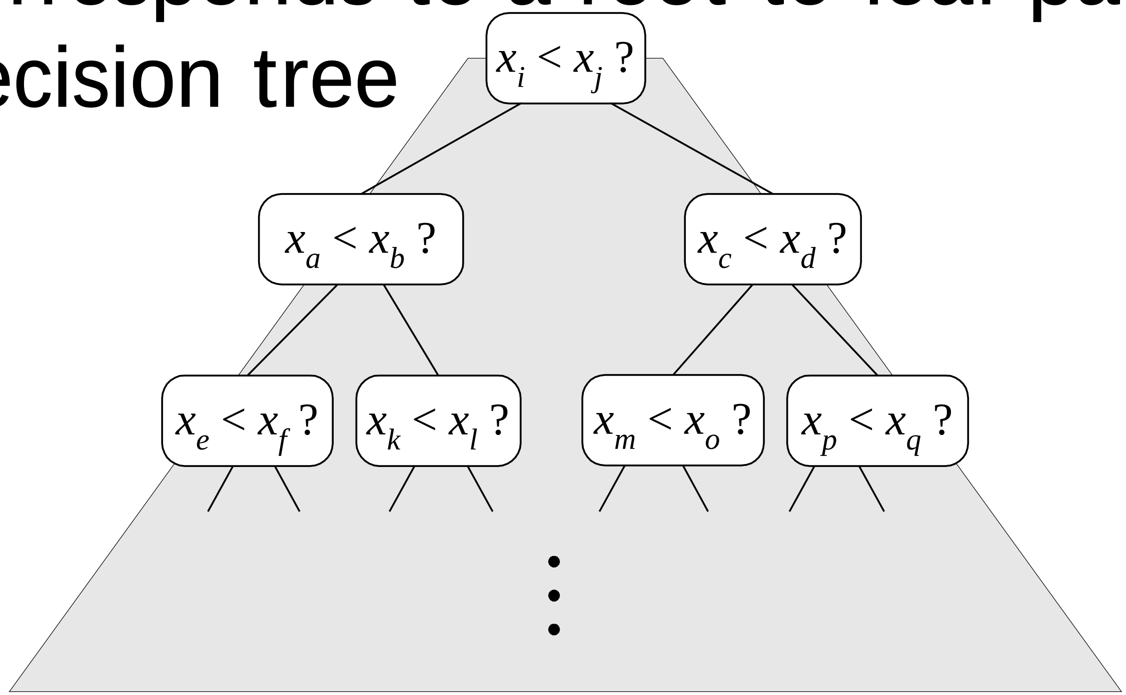
The Decision Tree Model

- Assumptions:
  - All numbers are distinct (so no use for $a_i = a_j$)
  - All comparisons have form $a_i \leq a_j$ (since $a_i \leq a_j$, $a_i \geq a_j$, $a_i < a_j$, $a_i > a_j$ are equivalent).
- Decision tree model
  - Full binary tree
  - Ignore control, movement, and all other operations, just use comparisons.
  - suppose three elements $< a_1, a_2, a_3 >$ with instance $<6,8,5>$.

Counting Comparisons

Let us just count comparisons then.

Each possible run of the algorithm corresponds to a root-to-leaf path in a decision tree

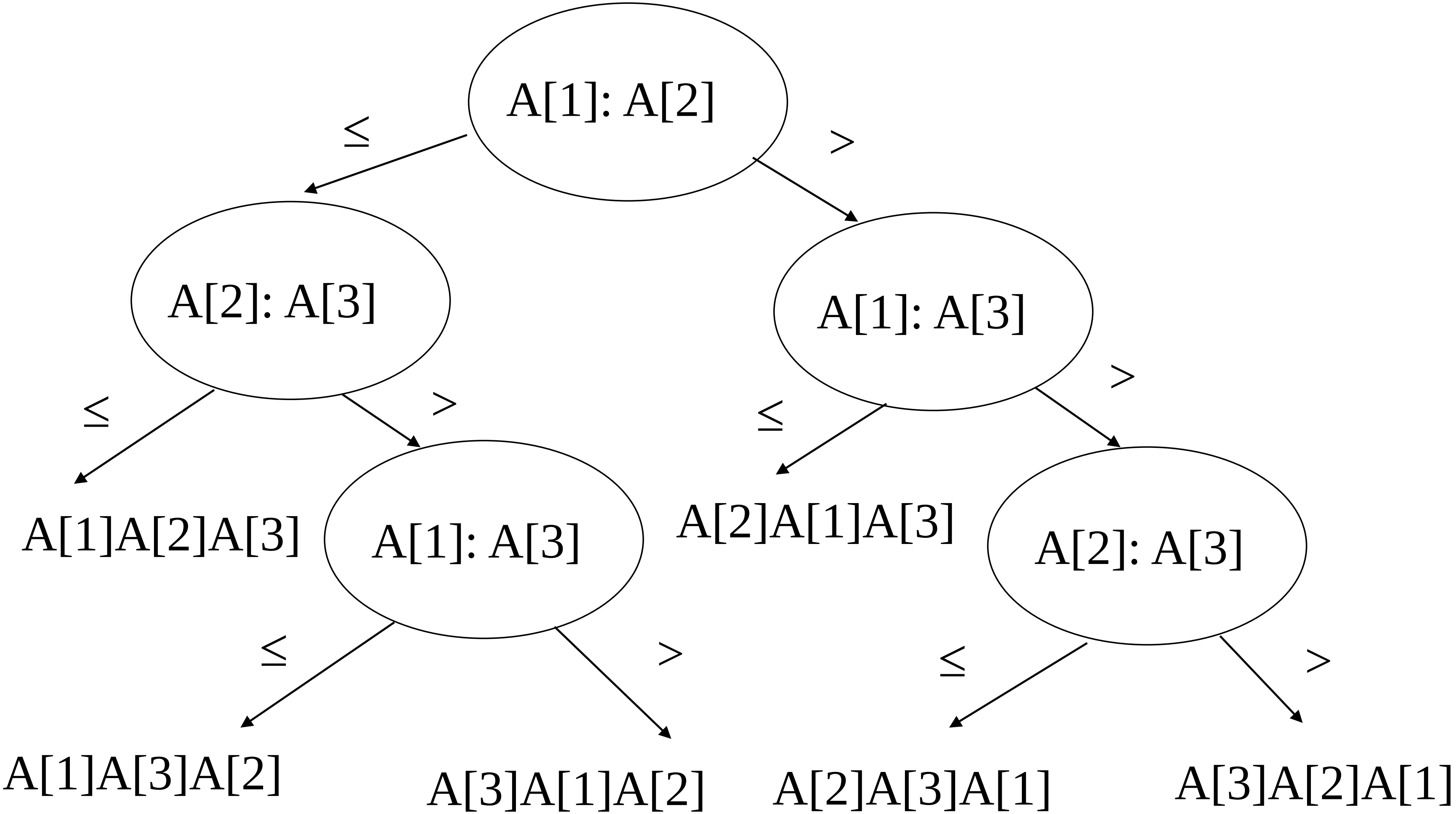


The Decision Tree Model - contd

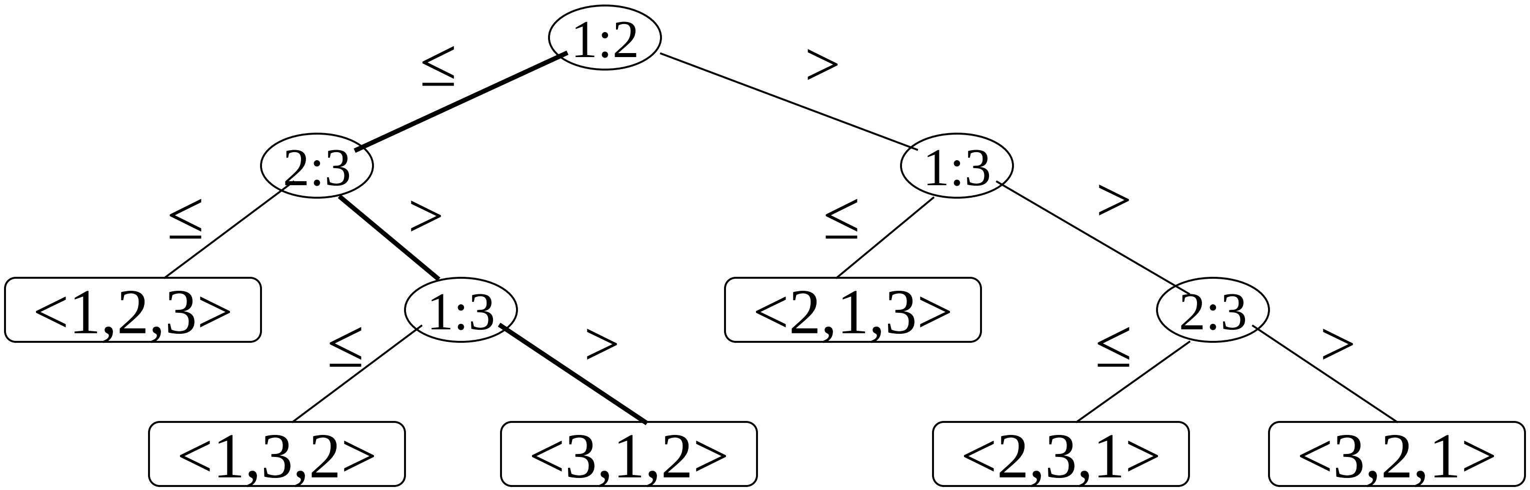
- Consider Insertion sort again

```
for j=2 to length(A)
   do key=A[j]
      i=j-1
      while i>0 and A[i]>key
        do A[i+1]=A[i]
           i--
      A[i+1]:=key
```

Example: insertion sort (n=3)



The Decision Tree Model



Internal node i:j indicates comparison between a_i and a_j.
Leave node $<\pi(1), \pi(2), \pi(3)>$ indicates ordering $a_{\pi(1)} \le a_{\pi(2)} \le a_{\pi(3)}$.
Path of bold lines indicates sorting path for <6,8,5>.
There are total 3!=6 possible permutations (paths).

Summary

- Only consider comparisons
- Each internal node = 1 comparison
- Start at root, make the first comparison

  - if the outcome is $\leq$ take the LEFT branch

  - if the outcome is > - take the RIGHT branch
- Repeat at each internal node
- Each LEAF represents ONE correct ordering

Lower bound for the worst case

- Claim: The decision tree must have at least n! leaves. WHY?
- worst case number of comparisons= the height of the decision tree.
- Claim: Any comparison sort in the worst case needs Ω(n log n) comparisons.
- Suppose height of a decision tree is h, number of paths (i,e,, permutations) is n!.
- Since a binary tree of height h has at most 2^h leaves,

$$n! \leq 2^h \text{, so } h \geq \lg(n!) \geq \Omega(n \lg n)$$

Lower bound: limitations

Q: Can we beat the lower bound for sorting?

A: In general no, but in some special cases YES!