Searching, Sorting

part 1

Week 3 Objectives

- Searching: binary search
- Comparison-based search: running time bound
- Sorting: bubble, selection, insertion, merge
- Sorting: Heapsort
- Comparison-based sorting time bound

Brute force/linear search

- Linear search: look through all values of the array until the desired value/event/condition found
- Running Time: linear in the number of elements, call it $O(n)$
- Advantage: in most situations, array does not have to be sorted

Binary Search

- Array must be sorted
- Search array A from index b to index e for value V
- Look for value V in the middle index m = (b+e)/2
  - That is compare V with A[m]; if equal return index m
  - If V<A[m] search the first half of the array
  - If V>A[m] search the second half of the array

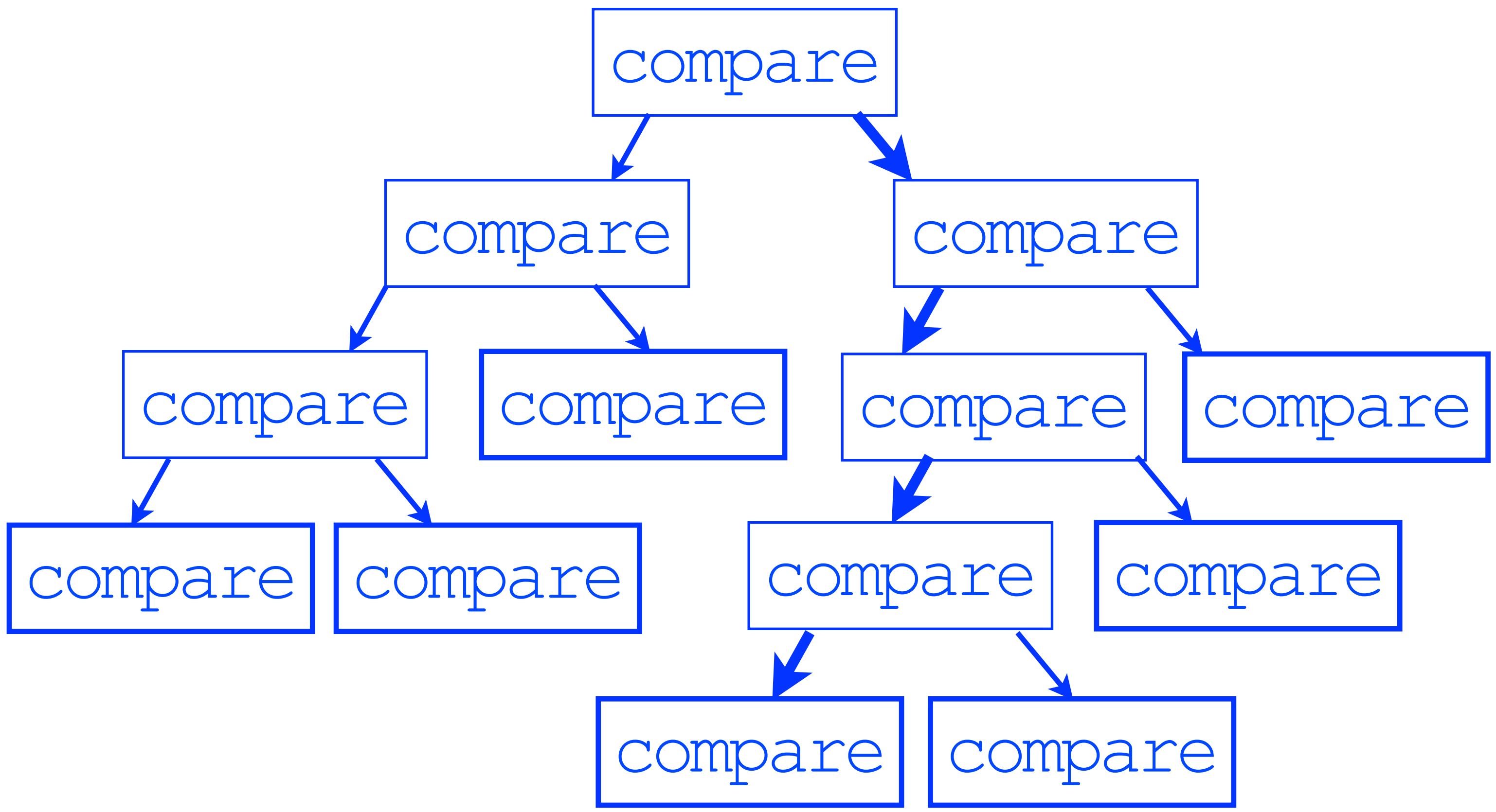
	b				m					e
V=3	-4	-1	0	0	1	1	3	19	29	47

A[m]=1 < V=3 => search moves to the right half

Binary Search Efficiency

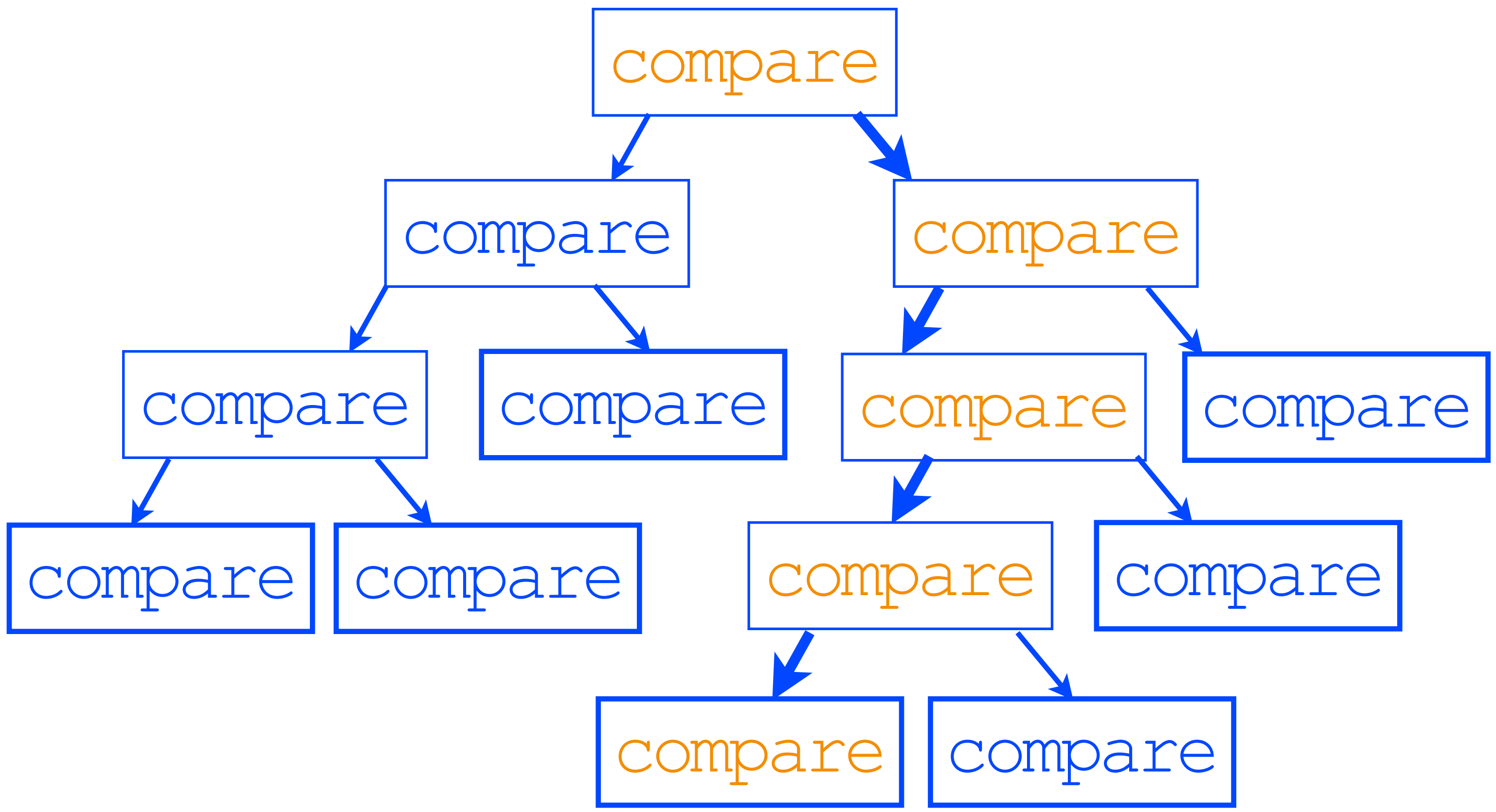
- every iteration/recursion
  - ends the procedure if value is found
  - if not, reduces the problem size (search space) by half
- worst case : value is not found until problem size=1
  - how many reductions have been done?
  - n / 2 / 2 / 2 / . . . . / 2 = 1. How many 2-s do I need ?
  - if k 2-s, then $n= 2^k$, so k is about log(n)
  - worst running time is O(log n)

Search: tree of comparisons



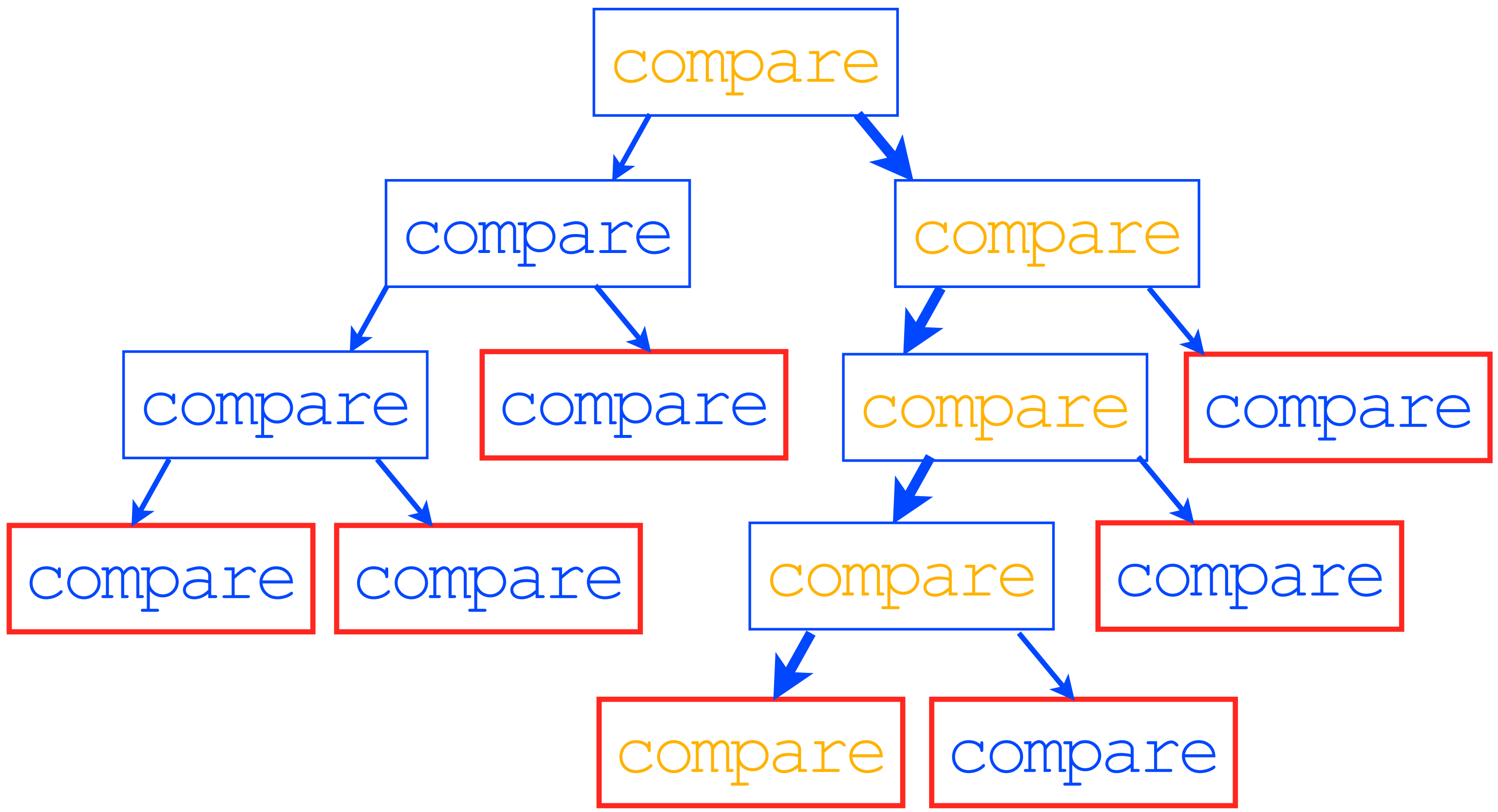
- tree of comparisons : essentially what the algorithm does

Search: tree of comparisons



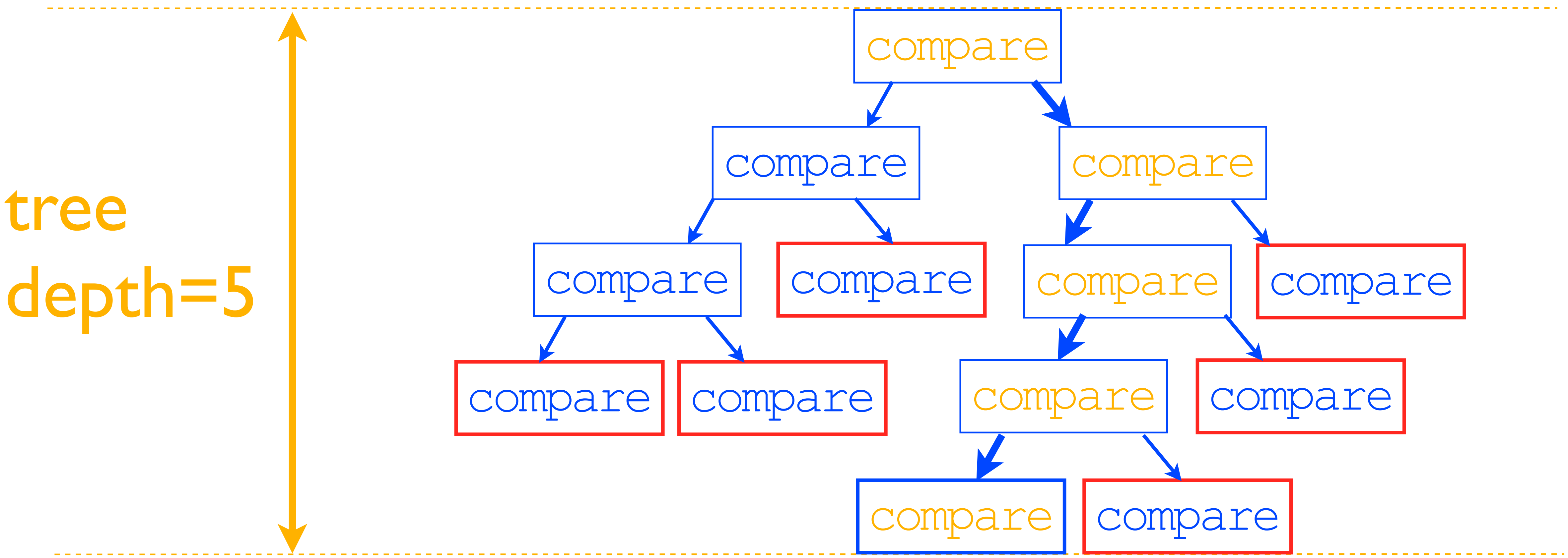
- tree of comparisons : essentially what the algorithm does
  - each program execution follows a certain path

Search: tree of comparisons



- tree of comparisons : essentially what the algorithm does
  - each program execution follows a certain path
  - red nodes are terminal / output
  - the algorithm has to have at least n output nodes... why ?

Search: tree of comparisons



- tree of comparisons : essentially what the algorithm does
  - each program execution follows a certain path
  - red nodes are terminal / output
  - the algorithm has to have n output nodes... why ?
  - if tree is balanced, longest path = tree depth = log(n)

Bubble Sort

- Simple idea: as long as there is an inversion, swap the bubble
  - inversion = a pair of indices $i<j$ with $A[i]>A[j]$
  - swap A[i]<->A[j]
    - directly `swap (A[i], A[j]);`
    - code it yourself: `aux = A[i]; A[i]=A[j];A[j]=aux;`
- how long does it take?
  - worst case : how many inversions have to be swapped?
  - $O(n^2)$

Insertion Sort

- partial array is sorted

1	5	8	20	49					

- get a new element V=9

Insertion Sort

- partial array is sorted

1	5	8	20	49					

- get a new element V=9
- find correct position with binary search i=3

Insertion Sort

- partial array is sorted

1	5	8	20	49					

- get a new element V=9
- find correct position with binary search i=3
- move elements to make space for the new element

1	5	8		20	49				

Insertion Sort

- partial array is sorted

1	5	8	20	49					

- get a new element V=9
- find correct position with binary search i=3
- move elements to make space for the new element

1	5	8		20	49				

- insert into the existing array at correct position

1	5	8	9	20	49				

Insertion Sort - variant

- partial array is sorted

1	5	8	20	49					

Insertion Sort - variant

- partial array is sorted

1	5	8	20	49					

Insertion Sort - variant

- partial array is sorted

1	5	8	20	49					

- get a new element V=9; put it at the end of the array

1	5	8	20	49	9				

Insertion Sort - variant

- partial array is sorted

1	5	8	20	49					

- get a new element V=9; put it at the end of the array

1	5	8	20	49	9				

- Move in V=9 from the back until reaches correct position

1	5	8	20	9	49				

Insertion Sort - variant

- partial array is sorted

1	5	8	20	49					

- get a new element V=9; put it at the end of the array

1	5	8	20	49	9				

- Move in V=9 from the back until reaches correct position

1	5	8	20	9	49				

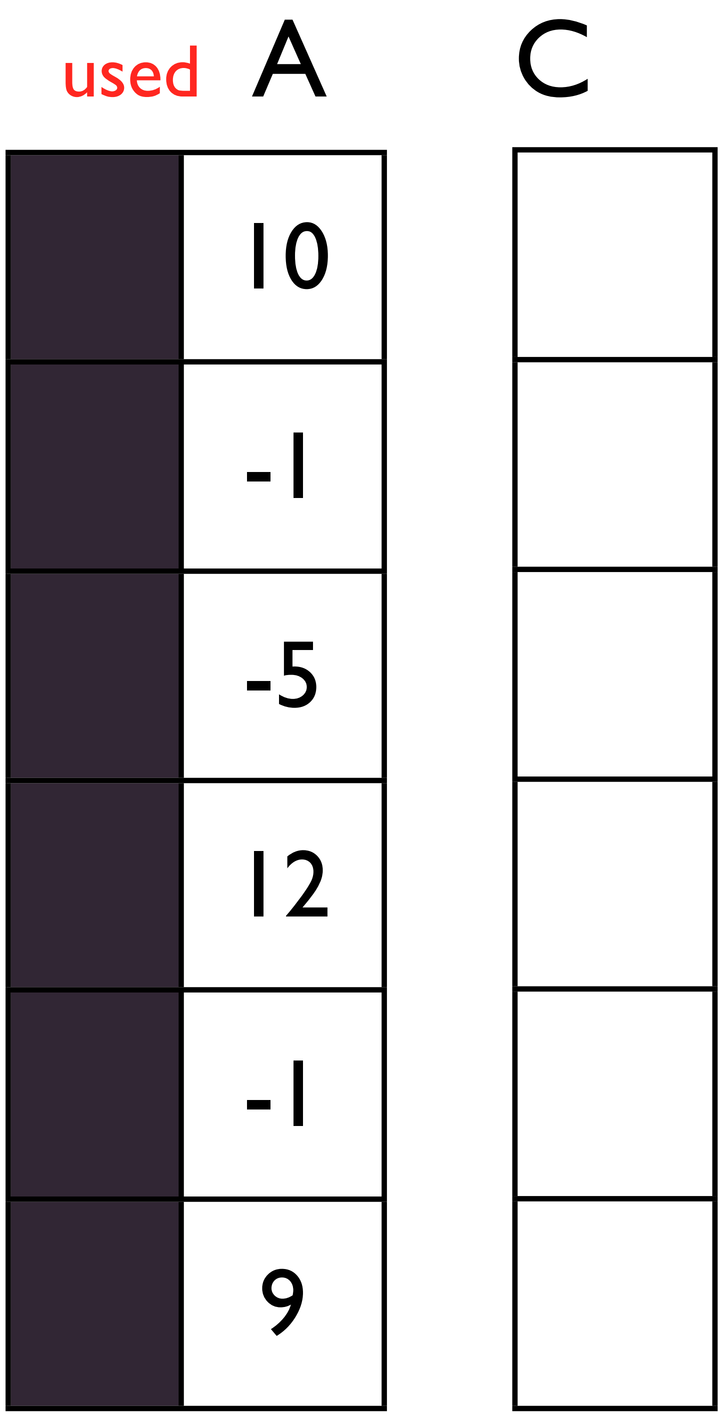
1	5	8	9	20	49				

Insertion Sort Running Time

- For one element, there might be required to move $O(n)$ elements (worst case $\Theta(n)$)
  - $O(n)$ insertion time
- Repeat insertion for each element of the n elements gives $n*O(n) = O(n^2)$ running time

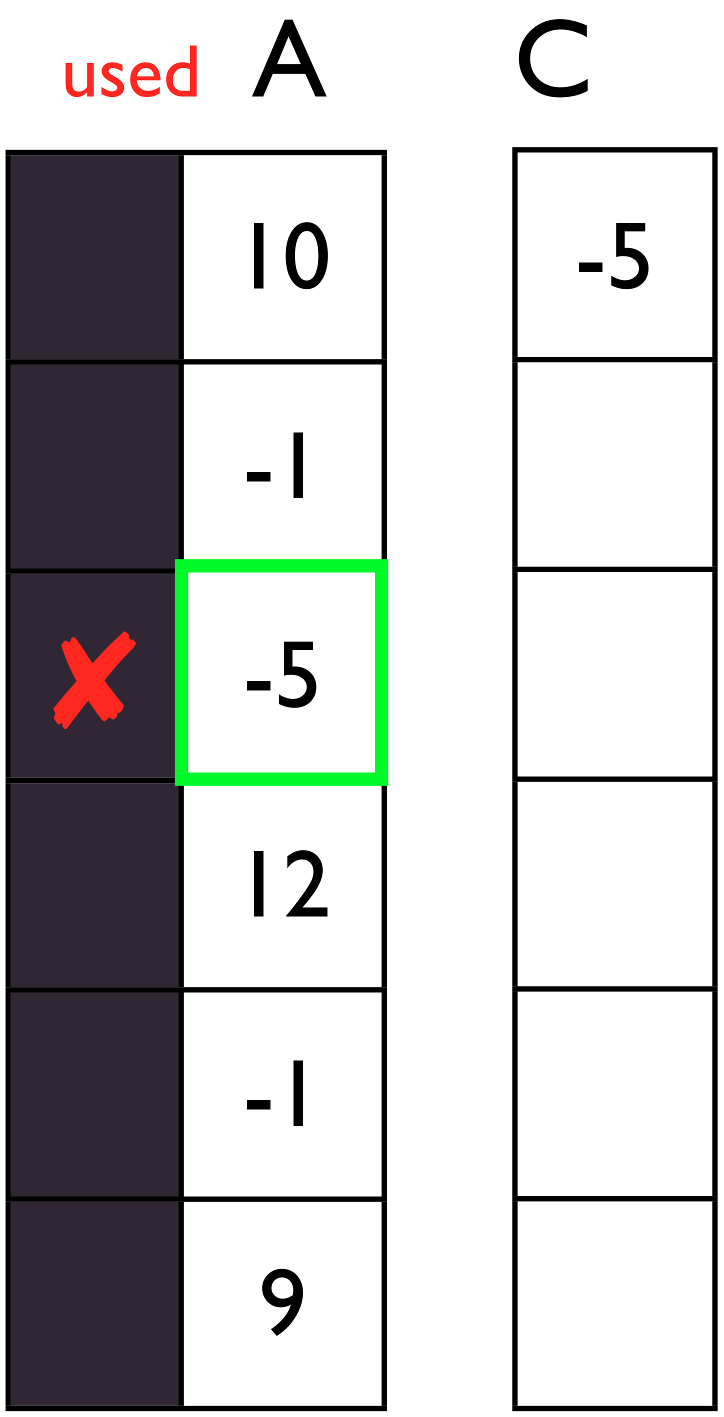
Selection Sort

- sort array A[] into a new array C[]
- while (condition)
  - find minimum element x in A at index i, ignore "used" elements
  - write x in next available position in C
  - mark index i in A as "used" so it doesn't get picked up again
- Insertion/Selection Running Time = $O(n^2)$



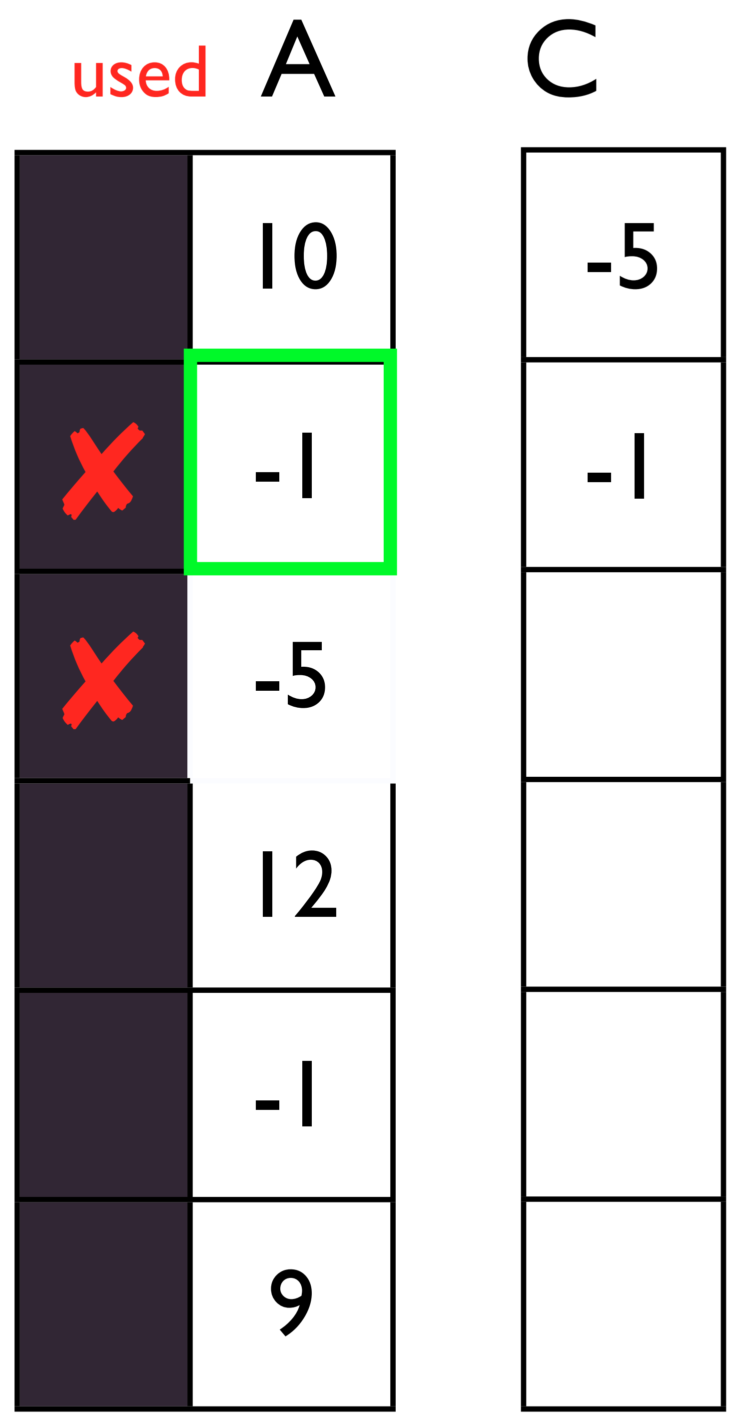
Selection Sort

- sort array A[] into a new array C[]
- while (condition)
  - find minimum element x in A at index i, ignore "used" elements
  - write x in next available position in C
  - mark index i in A as "used" so it doesn't get picked up again
- Running Time = $O(n^2)$



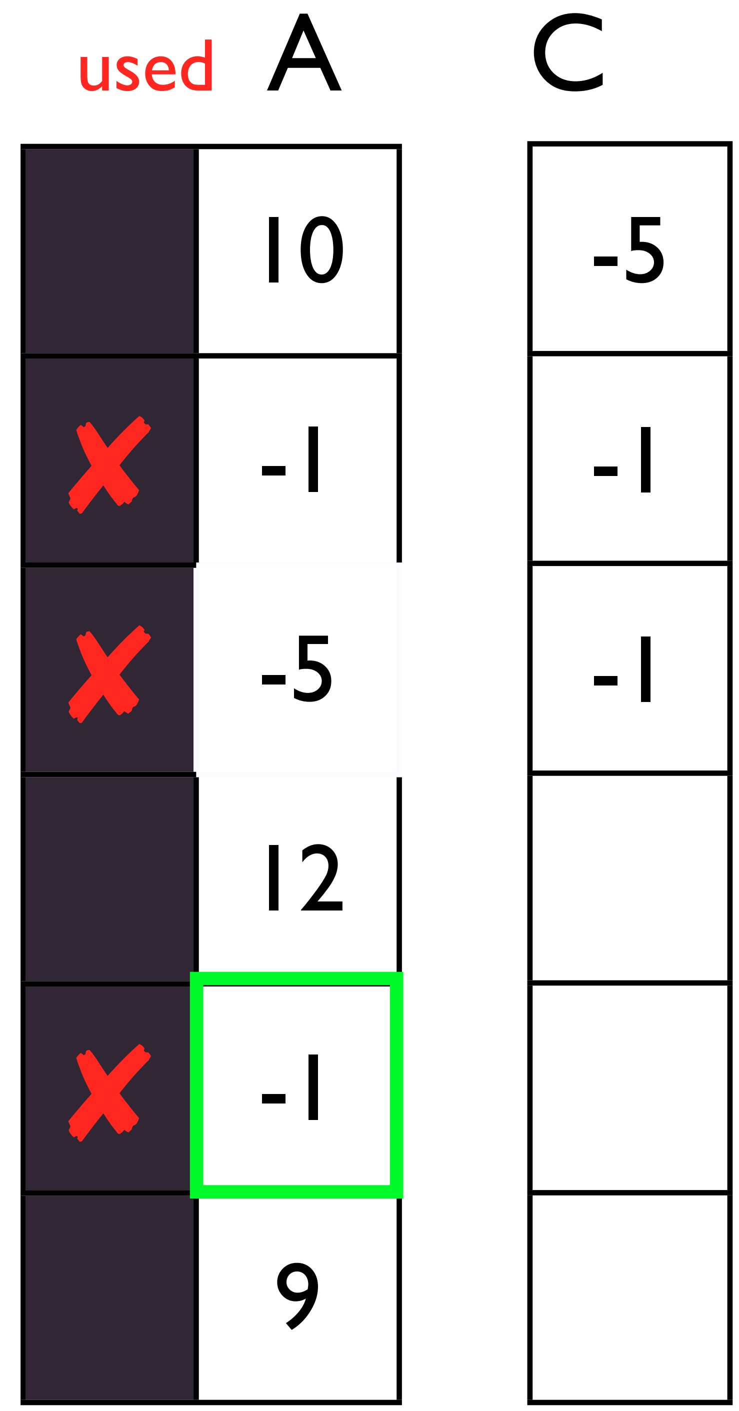
Selection Sort

- sort array A[] into a new array C[]
- while (condition)
  - find minimum element x in A at index i, ignore "used" elements
  - write x in next available position in C
  - mark index i in A as "used" so it doesn't get picked up again
- Running Time = $O(n^2)$



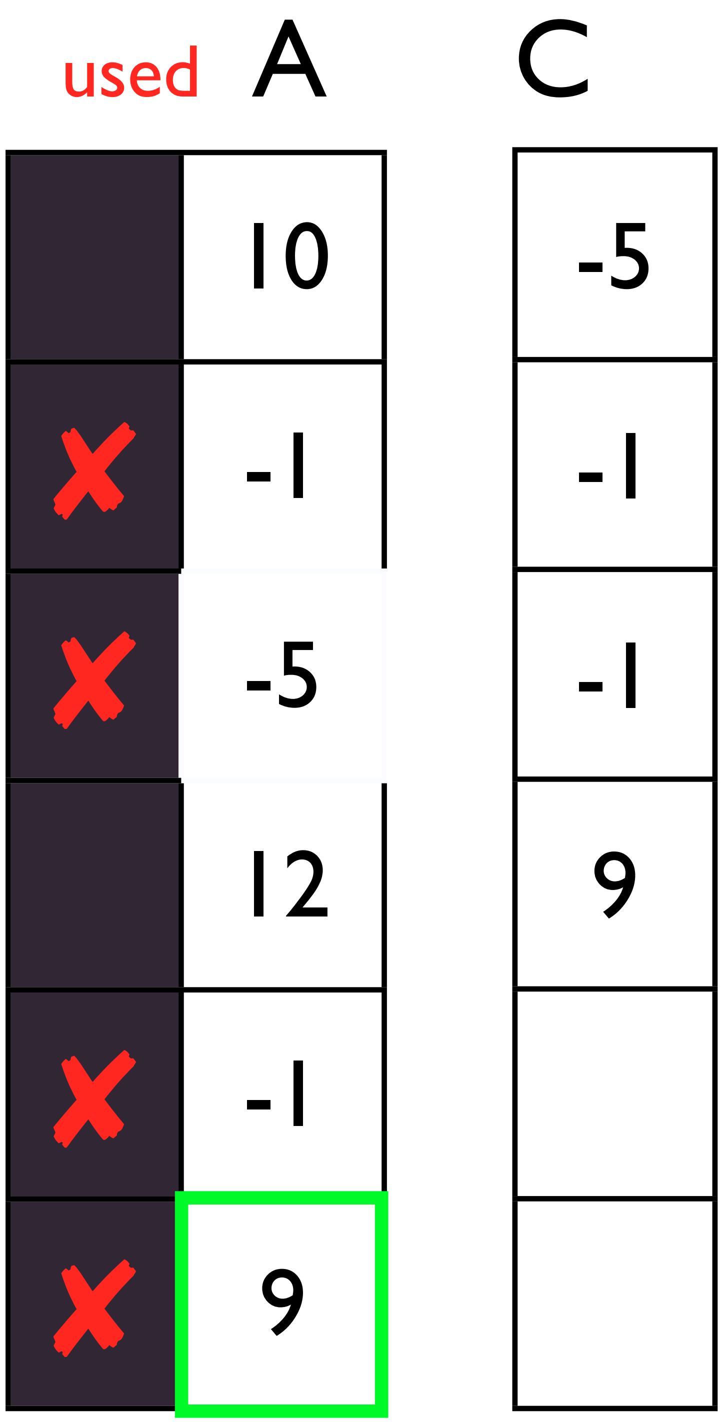
Selection Sort

- sort array A[] into a new array C[]
- while (condition)
  - find minimum element x in A at index i, ignore "used" elements
  - write x in next available position in C
  - mark index i in A as "used" so it doesn't get picked up again
- Running Time = $O(n^2)$



Selection Sort

- sort array A[] into a new array C[]
- while (condition)
  - find minimum element x in A at index i, ignore "used" elements
  - write x in next available position in C
  - mark index i in A as "used" so it doesn't get picked up again
- Running Time = $O(n^2)$



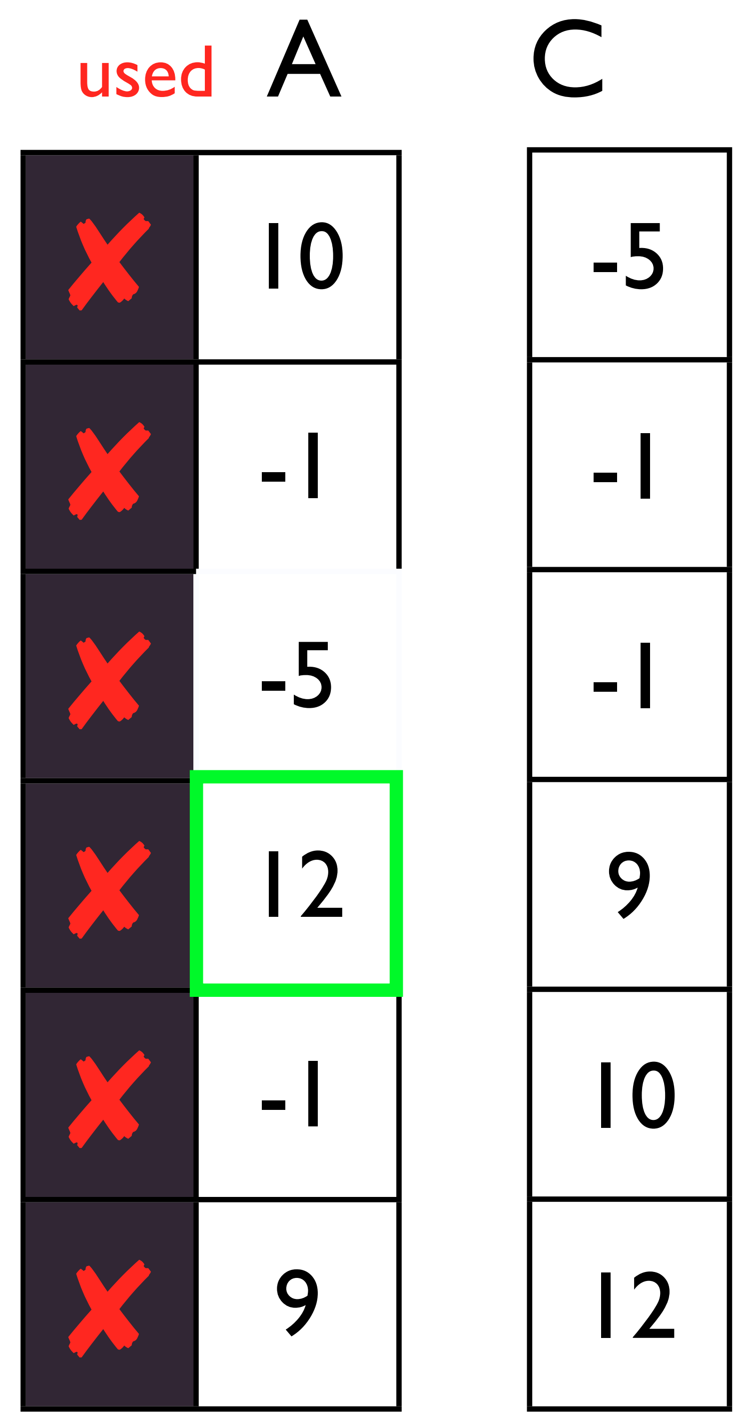
Selection Sort

- sort array A[] into a new array C[]
- while (condition)
  - find minimum element x in A at index i, ignore "used" elements
  - write x in next available position in C
  - mark index i in A as "used" so it doesn't get picked up again
- Running Time = $O(n^2)$

used	A	C
✗	10	-5
✗	-1	-1
✗	-5	-1
	12	9
✗	-1	10
✗	9	

Selection Sort

- sort array A[] into a new array C[]
- while (condition)
  - find minimum element x in A at index i, ignore "used" elements
  - write x in next available position in C
  - mark index i in A as "used" so it doesn't get picked up again
- Running Time = $O(n^2)$



Merge two sorted arrays

- two sorted arrays
  - A[] = { 1, 5, 10, 100, 200, 300}; B[] = {2, 5, 6, 10};

- merge them into a new array C
  - index i for array A[], j for B[], k for C[]
  - init i=j=k=0;
  - while (what_condition_?)
    - if (A[i] <= B[j]) { C[k]=A[i], i++ } //advance i in A
    - else {C[k]=B[j], j++} // advance j in B
    - advance k
  - end_while

Merge two sorted arrays

- complete pseudocode
  - index i for array A[], j for B[], k for C[]
  - init i=j=k=0;
  - while (k < size(A)+size(B)+1)
    - if(i>size(A) {C[k]=B[j], j++} // copy elem from B
    - else if (j>size(B) {C[k]=A[i], i++} // copy elem from A
    - else if (A[i] <= B[j]) { C[k]=A[i], i++ } //advance i
    - else {C[k]=B[j], j++} // advance j
    - k++ //advance k
  - end_while

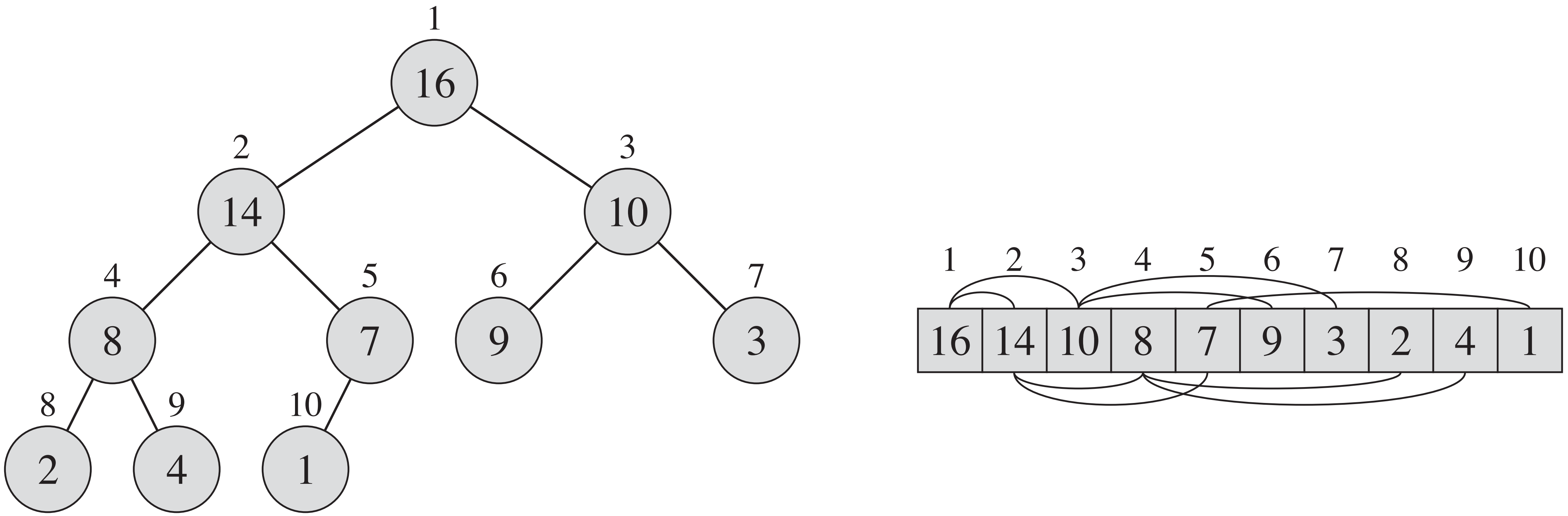
MergeSort

- divide and conquer strategy
- MergeSort array A
  - divide array A into two halves A-left, A-right
  - MergeSort A-left (recursive call)
  - MergeSort A-right (recursive call)
  - Merge (A-left, A-right) into a fully sorted array
- running time : O(nlog(n))

MergeSort running time

- $T(n) = 2T(n/2) + \Theta(n)$
  - 2 sub-problems of size n/2 each, and a linear time to combine results
  - Master Theorem case 2 ($a=2$, $b=2$, $c=1$)
  - Running time $T(n) = \Theta(n \log n)$

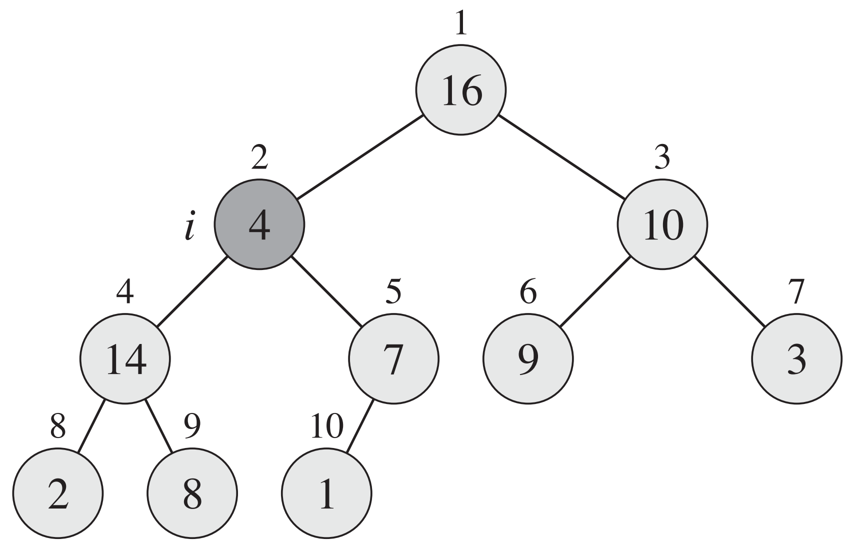
Heap DataStructure



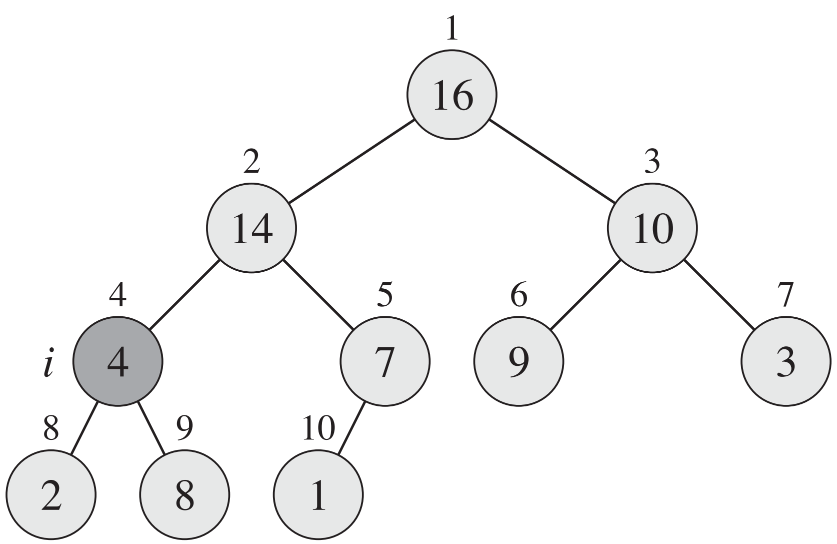
(a) (b)

- binary tree
- max-heap property : parent > children

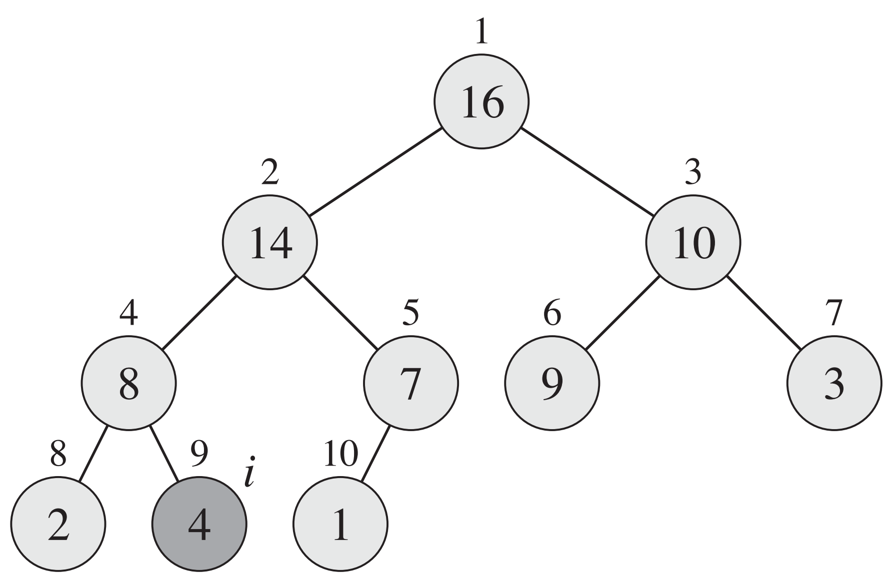
Max Heap property



(a)



(b)



(c)

- Assume the Left and Right subtrees satisfy the Max-Heap property, but the top node does not
- Float down the node by consecutively swapping it with higher nodes below it.

Building a heap

- Representing the heap as array datastructure
  - Parent(i) = i/2
  - Left_child(i)=2i
  - Right_child(i) = 2i+1
- A = input array has the last half elements leafs
- MAX-HEAPIFY the first half of A, reverse order

```
for i=size(A)/2 downto 1
    MAX-HEAPIFY (A,i)
```

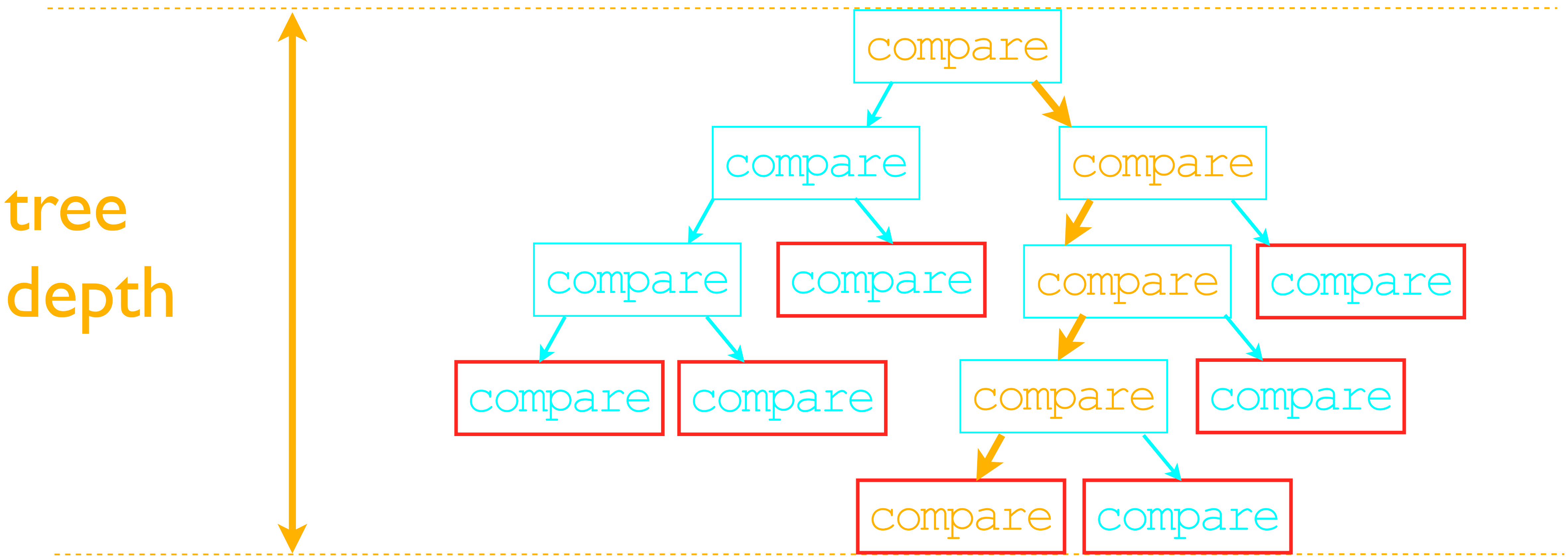
Heapsort

- Build a Max-Heap from input array
- LOOP
  - swap heap_root (max) with a leaf
  - output (take out) the max element; reduce size
  - MAX-HEAPIFY from the root to maintain the heap property
- END LOOP
- the output is in order

HeapSort running time

- Max-Heapify procedure time is given by recurrence
  - $T(n) \leq T(2n/3) + \Theta(1)$
  - master Theorem $T(n)=O(\log n)$
- Build Max-Heap : running n times the Max-Heapify procedure gives the running time $O(n\log n)$
- Extracting values: again run n times the Max-Heapify procedure gives the running time $O(n\log n)$
- Total $O(n\log n)$

Sorting : tree of comparisons



- tree of comparisons : essentially what the algorithm does
  - each program execution follows a certain path
  - red nodes are terminal / output
  - the algorithm has to have n! output nodes... why ?
  - if tree is balanced, longest path = tree depth = n log(n)