Lecture 9: Sorting & lists

1. Sorting
2. Operations on (ordered) lists

Introduction

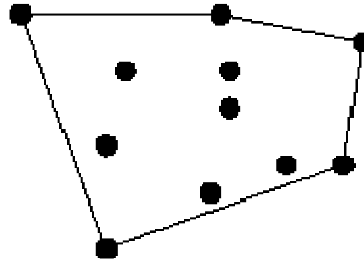
- Sorting: Sort elements of an array in ascending or descending order
- Implementations:
  - in-place sorting
  - indexing
  - ranking
- Why should you know about sorting algorithms?
  - Handling of experimental and numerical data
  - Ordering of basis states for sparse matrix problems
  - Search in unordered data: cost is $O(N)$ for N data points
  - Search in ordered data: cost is $O(\log N) \rightarrow$ bisection

- Sorting algorithms: $O(N \log N)$
  - Quick sort (some memory overhead)
  - Heap sort (in-place sorting)

  Not recommended for N exceeding 10…20:
  - Naïve ordering: $O(N^2)$. Select largest element, then second largest, etc

Applications of Sorting

1. Searching in O(log N) time opposed to O(N)
2. Closest pair: given N elements, which two are closest to each other?
3. Element uniqueness: Given N elements, are there duplicates?
4. Frequency distribution: Given N elements, which occurs most often (most frequent)?
5. What's the kth largest element?
6. Convex hull: Given N points in D dimensions, which is the smallest convex polygon that encloses them all?



7. Huffman codes: Minimize the space a text file is taking by assigning a short binary code to the most frequent letters.

Character	Frequency	Code
f	5	1100
e	9	1101
c	12	100
b	13	101
d	16	111
a	45	0

Quicksort

Quicksort is a divide and conquer algorithm.

- First linear sweep through array puts pivot into place and everything left (right) from pivot is smaller (larger) than pivot. [divide]
- Repeat on left and right subarrays recursively, until done. [conquer]

- Choose pivot
- Starting from the left, move to right until element a(i) > pivot
- Starting from right, move to left until element a(j) < pivot
- If (i < j) exchange a(i) with a(j). Else exchange a(i) or a(i) with pivot, depending on pivot's position.

Example:

17	12	6	19	23	8	**5**	**10**
5	**12**	6	19	23	**8**	17	**10**
5	8	**6**	**19**	23	12	17	**10**
5	8	6	**10**	23	12	17	19

Repeat on left and right sub-arrays.

Worst case: array is already ordered. No exchanges are made. Cost $O(N^2)$.

Quicksort needs some additional memory O(log N) to keep track of partitions.

Standard case: Quciksort is fastest available sorting routine.

Heapsort

- Heapsort is an in-place sorting routine. Constructs heap (partially ordered tree) from array.

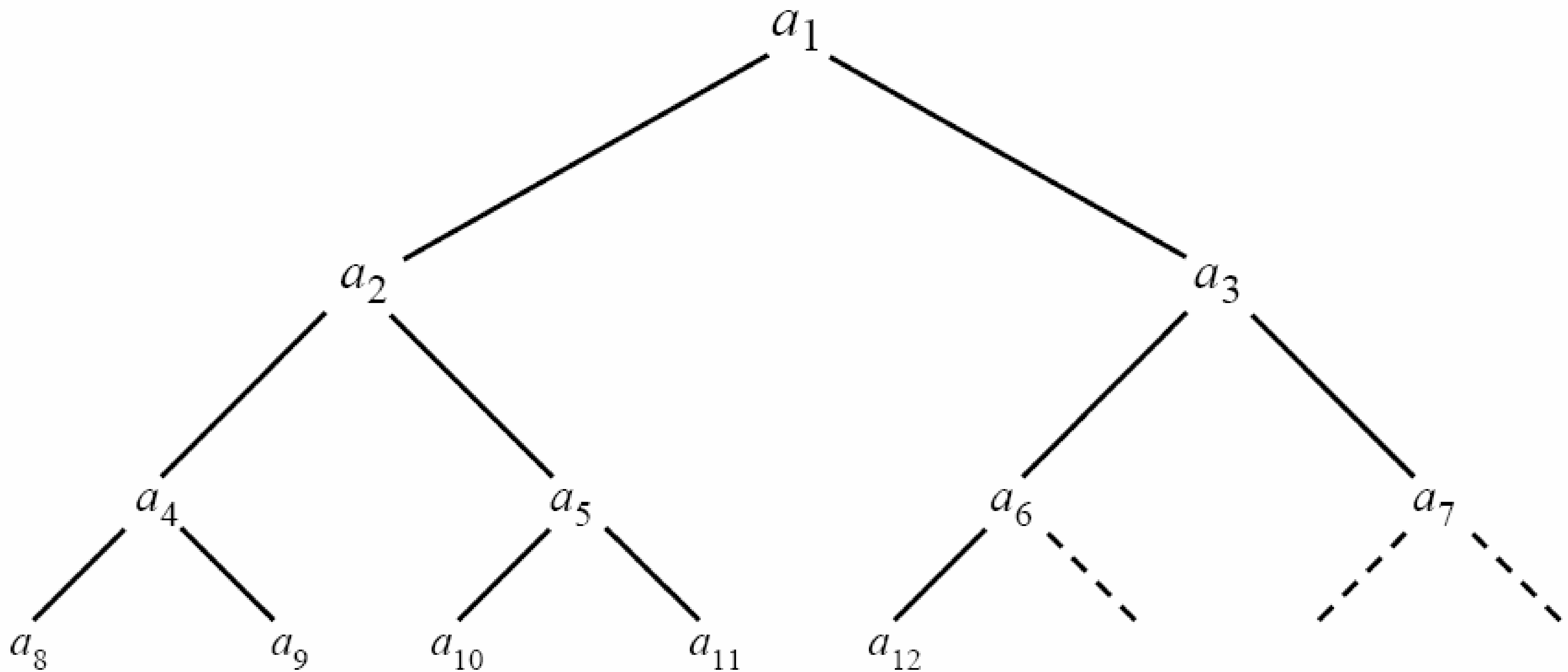
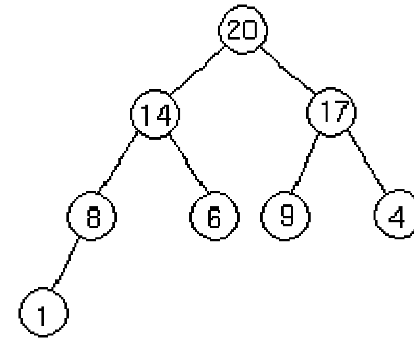


Figure 8.3.1. Ordering implied by a “heap,” here of 12 elements. Elements connected by an upward path are sorted with respect to one another, but there is not necessarily any ordering among elements related only “laterally.”

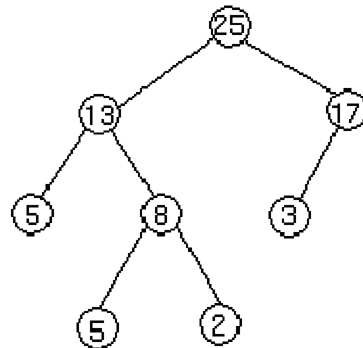
Heaps

Array (/ 20, 14, 17, 8, 6, 9, 4, 1 /) is a heap:

- Each parent >= child
- partially ordered binary tree
- complete tree (balanced)



Not a heap:



Operations for heap sort

Four basic procedures on heap are

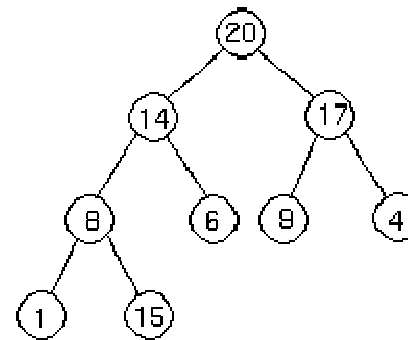
1. Build-Heap, which runs in linear time.
2. Heapify, which runs in O(log *N*) time, for each element.

Basic operations for heap sort

1. Extract-Max = Heapify, which runs in O(log *N*) time.
2. Heap Sort thus runs in O(*N* log *N*) time.

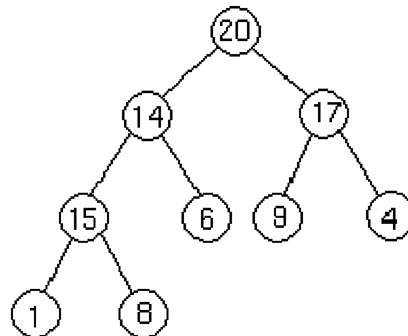
Build heap from array: cost O(N)

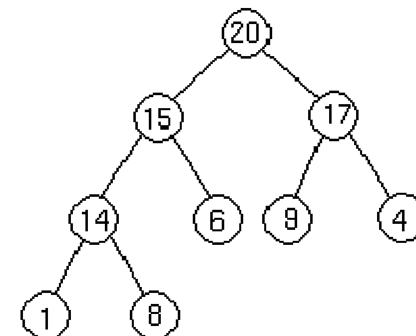
(/ 20, 14, 17, 8, 6, 9, 4, 1, 15 /)



Heapify by exchanges:

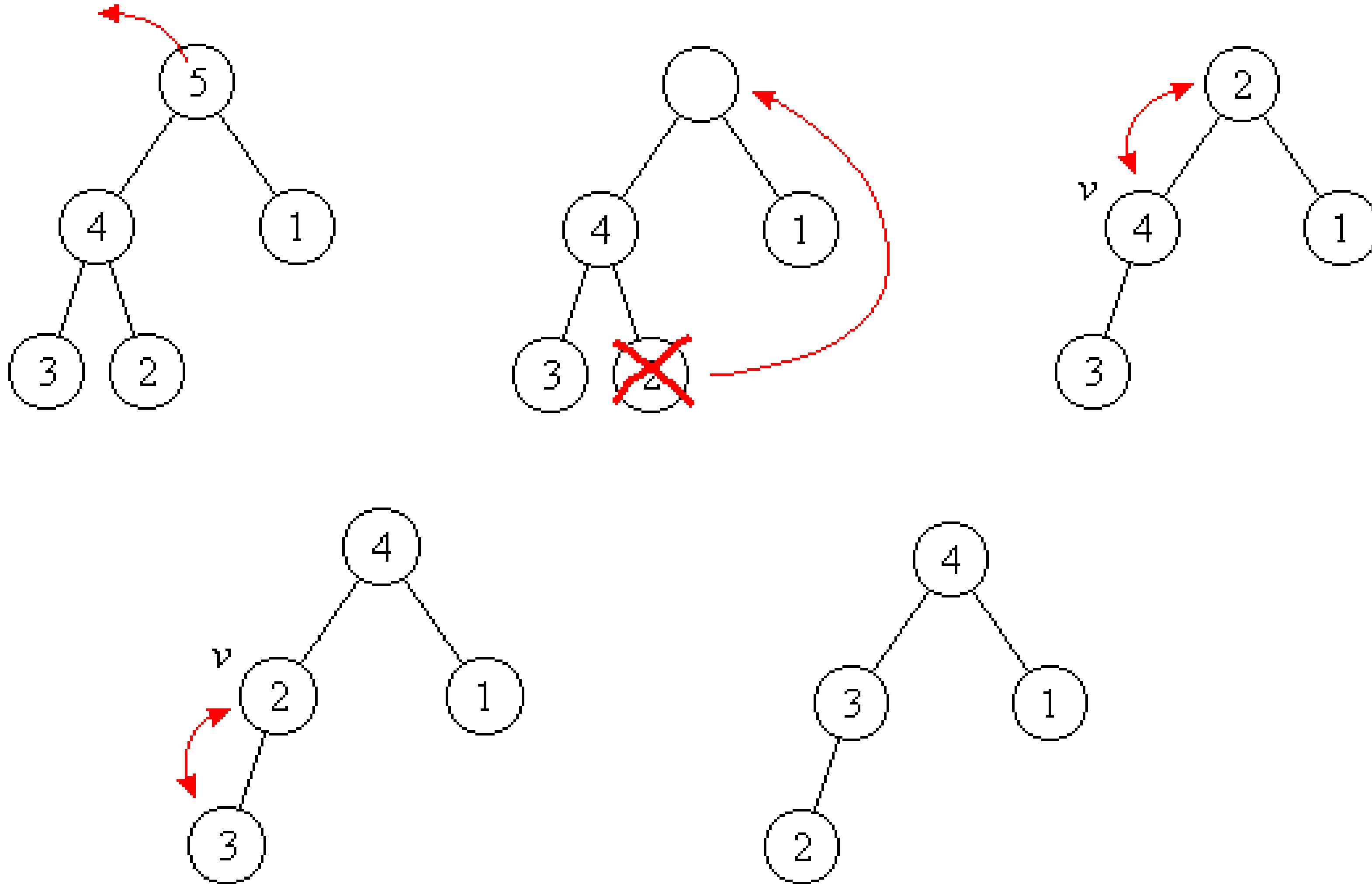
Cost O($\log_2$N)





Heap sort operations (cont'd)

Extraction of largest element and subsequent heapify



Lists

```
program LinkedList
type node
        real data
        type( node ), pointer :: next
end type node

type( node ), pointer :: list, current, previous
integer, parameter :: N = 10
nullify( list ) ! Initialize list to point to no target
! Add the 1st element as a special case.
if ( N > 0 ) then
        allocate( list )
        nullify( list%next )
        call random_number( list%data )
end if

current => list ! Add N random numbers to the list.

do i = 2, N
        allocate( current%next )
        nullify( current%next%next )
        call random_number( current%next%data )
         current => current%next
end do
 ! Output the list, deallocating them after use.
print *, 'List elements are:' current => list

do while ( associated( current ) )
        print *, current%data
        previous => current
        current => current%next
        deallocate( previous )
 end do
end program LinkedList
```