Algorithms and Data Structures (CSci 115 – Spring 2019)

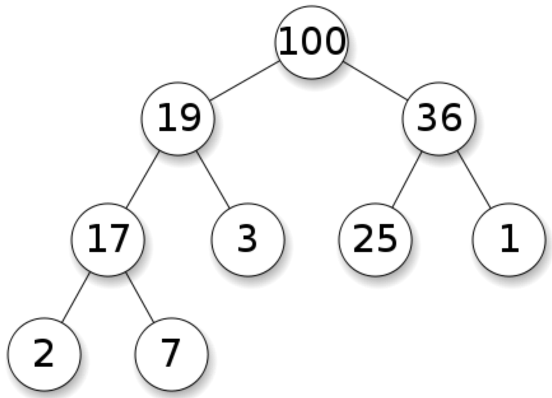
California State University Fresno
College of Science and Mathematics
Department of Computer Science
H. Cecotti

Learning outcomes

- Heaps
  - Definitions
  - Binary
  - Heapsort algorithm

Heap

- Definition
  - A specialized tree-based data structure
    - Satisfying the ***heap property***:
      - if (P is a parent node of C)
      - then the *key* (the *value*) of P is either
        - ≥ to the key of C (*in a max heap*) **or**
        - ≤ to the key of C (*in a min heap*)
    - The node at the top of the heap = *root* node.

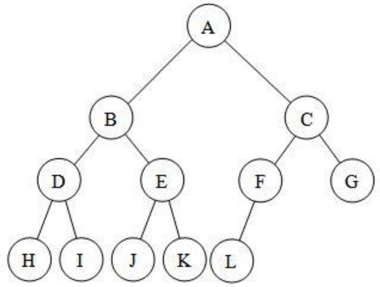


Example: max heap

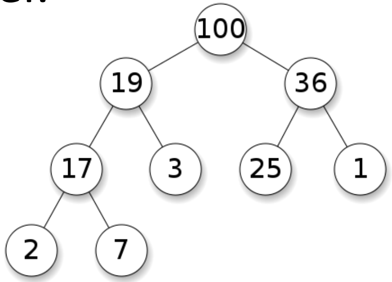
Binary heaps

- Binary tree with two constraints:

1. Shape property
   - A binary heap is a **complete binary tree**
     - All the levels of the tree (except possibly the last one) are fully filled
     - if the last level is not complete → level is filled from left to right

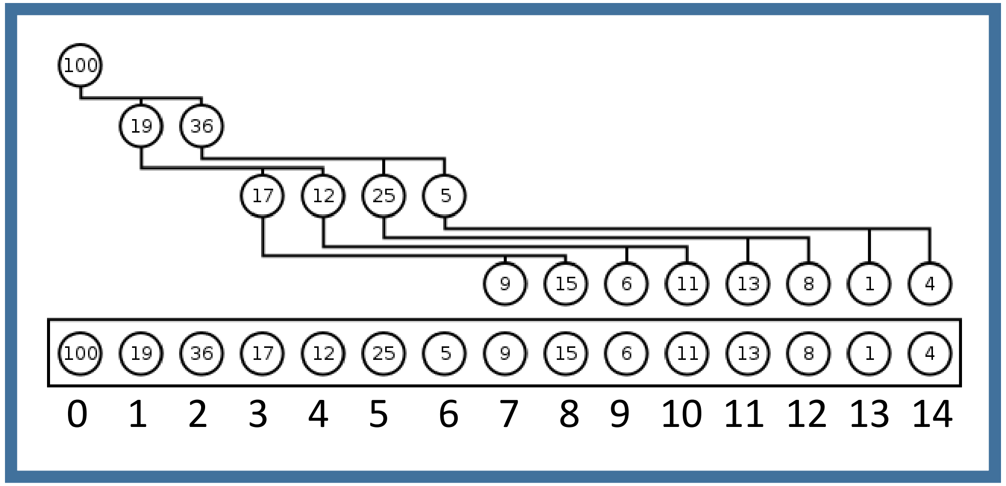


2. Heap property
   - The key stored in each node is either
     - ≥ or ≤ to the keys in the node's children - according to some total order.



Binary heaps

- Implementation
  - Represented as an array
  - The first (or last) element
    - contain the root.
  - The next 2 items contain
    - its children
  - The next 4 items contain
    - the 4 children of the 2 child nodes...
  - → the children of the node at position n would be at positions **2n + 1** and **2n + 2** in a 0-based array (C/C++).
  - Up or Down the tree
    - doing simple index computations



Binary heaps

- Balancing a heap by:
  - ➢ Sift-up/Sift-down operations
    - **Swapping** elements which are out of order
  - ➢ **Sift-Down**
    - Swaps a node that is **too small** with its **largest child** (→ moving it down)
      - until it is at least as large as both nodes **below** it.
  - ➢ **Sift-Up**
    - Swaps a node that is **too large** with its parent (→ moving it up)
      - until it is no larger than the node **above** it
  - ➢ BuildHeap function
    - Array of **unsorted** items and moves them until it they all satisfy the heap property

Binary heap

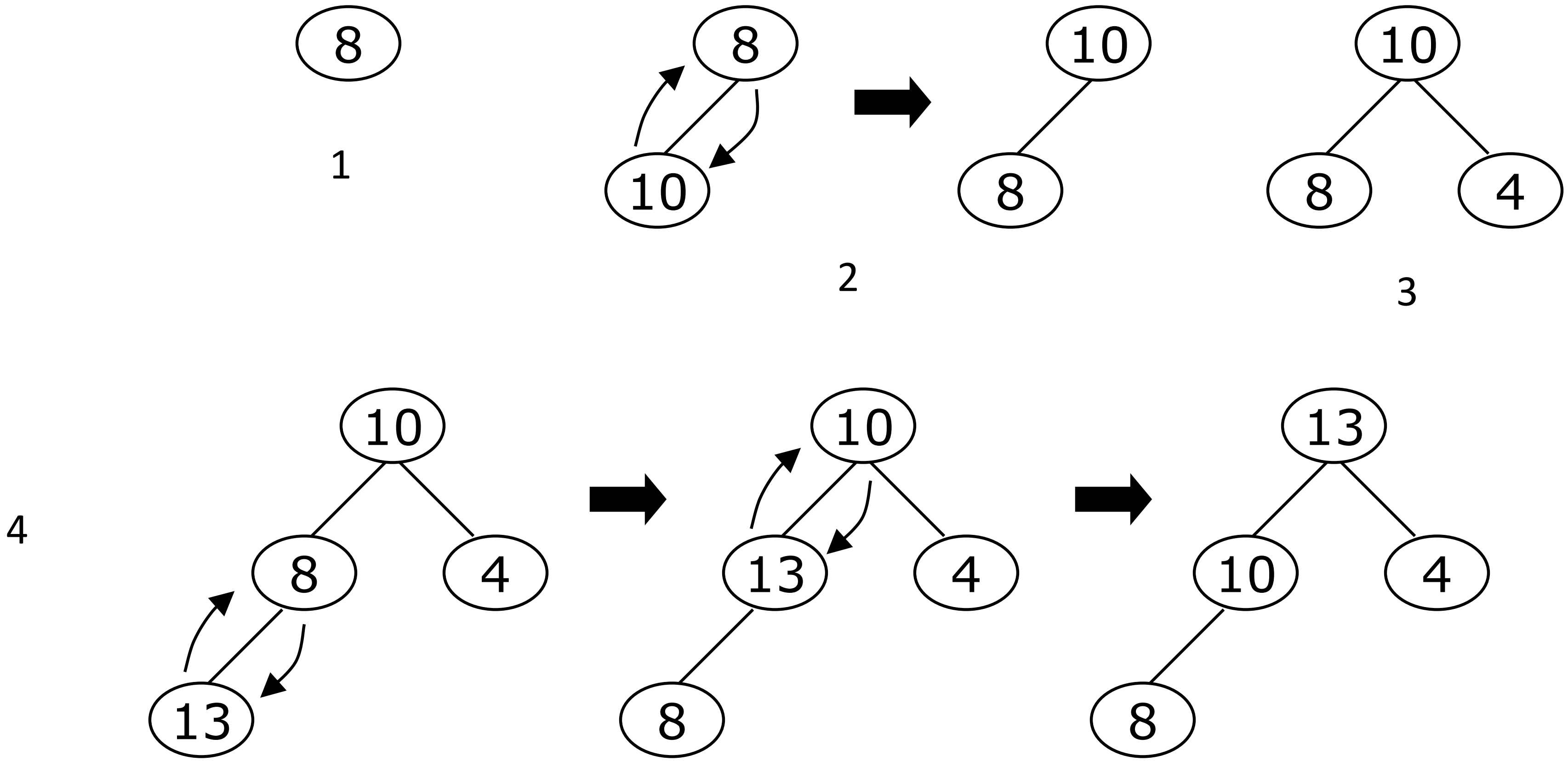
- **Sift-up**
  - A node that does not have the heap property
  - →give it the heap property by swapping its value with the value of the larger child
  - Warning
    - A child may have lost the heap property

Binary heap

- Heap creation
  - By adding nodes one at a time:
    - Add the node just to the **right** of the rightmost node in the deepest level
    - If (the deepest level is full) then start a new level
  - At each insertion
    - **Problem**: we may destroy the heap property of its parent node!
    - **Solution**: sift-up
      - But each time we sift up, the value of the **topmost** node in the sift may increase
      - → It may destroy the heap property of its parent node !!
      - Therefore
      - Repetition (sifting up process)
        - moving up in the tree
      - Until either
        - **Reach nodes whose values don't need to be swapped**
          - as the parent is still larger than both children
        - **Reach the root**

Binary heap

- Example for the construction of a heap



Binary heaps

- Complexity

Algorithm	Average	Worst case
Space	O(n)	O(n)
Search	O(n)	O(n)
Insert	O(1)	O(log n)
Delete	O(log n)	O(log n)
Peek	O(1)	O(1)

Heap sort

- Heapsort (Williams – 1964)
- Comparison-based sorting algorithm
  - “**Better selection sort**”
    - Heap data structure instead of linear-search to find the maximum value
- Algorithm
  1. Build a heap with the sorting array
     - using recursive insertion.
  2. Iterate to extract n times
     - the maximum or minimum element in heap and heapify the heap.
  3. The extracted elements form a sorted sub-sequence.

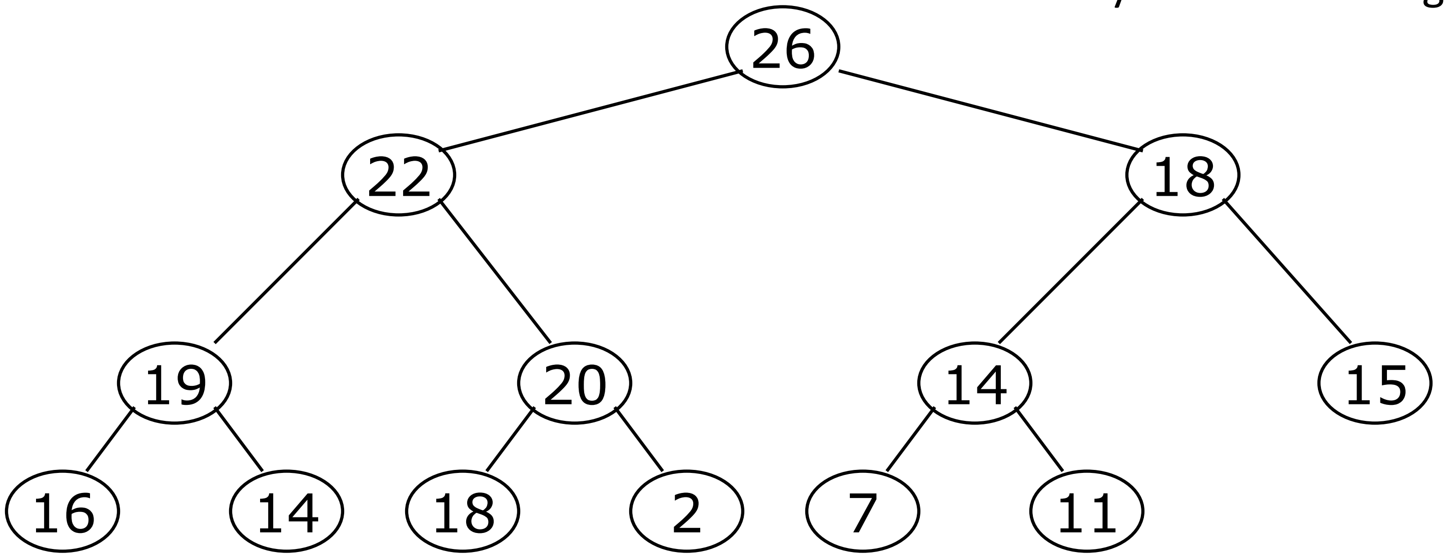
Heap sort

- Main idea
  - Heaps must always follow a specific order (min or max)
  - Leverage that property to find the **largest** (maximum value element)
    - Sequentially sort elements by
      - Selecting the root node of a heap
      - Adding the root to the end of the array
  - Build max heap
    - Using all the data with the build max heap function
  - Get the largest value item
    - At the root node of the heap
    - → Every parent node is larger than the children → swap largest value with the item at the end of the heap
  - Heapify function
    - Move down the root node item to its **correct** place

Heap sort

- Warning
  - **Heapify != Sorting**

Example:
Binary tree after being heapified



Heap sort

- Remove the root
  - Method
    - Remove the rightmost leaf at the deepest level and use it for the new root
    - → tree is balanced and left-justified but no longer a heap
    - However, only the root lacks the heap property ☹
    - → We can siftUp() the root
    - Then 1 and only 1 of its children may have lost the heap property
    - … continue until we reach the bottom of the tree = each node is a heap

Heap sort

- Algorithm
  - Pseudo-code:

```
heapify the array;
while the array is not empty {
    remove and replace the root;
    reheap the new root node;
}
```

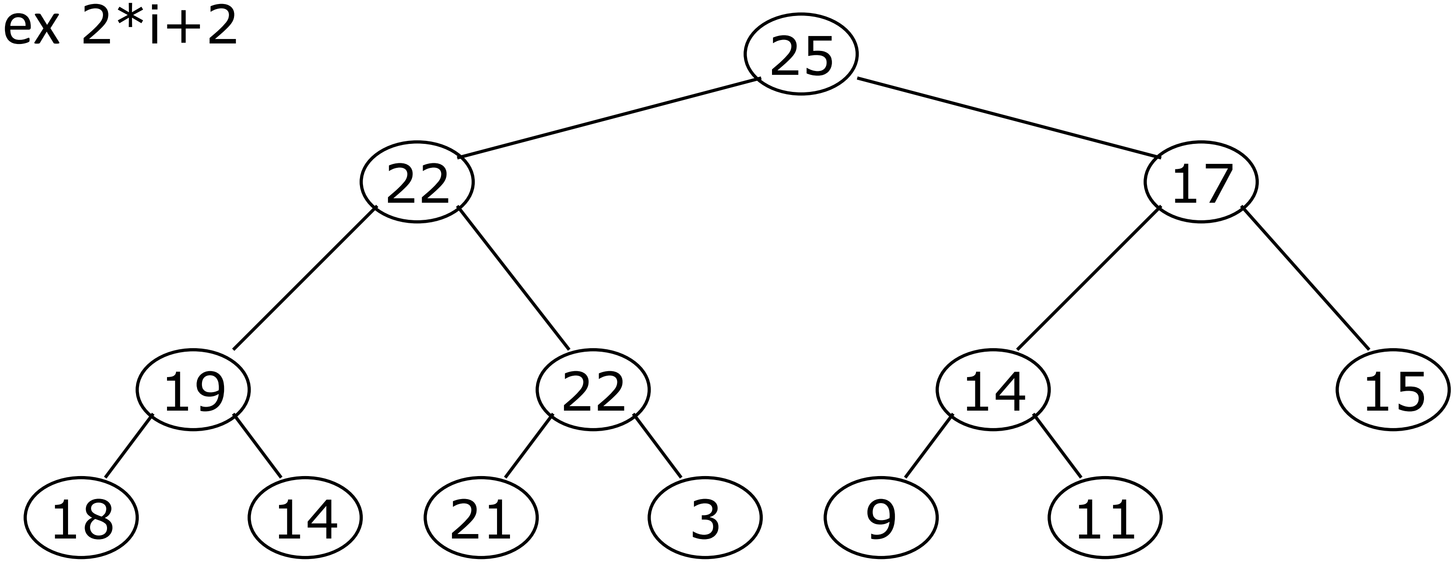
```
Heapsort(int* A) {
    BuildHeap(A)
    for i = n to 1
        swap(A[1], A[i])
        n = n - 1
        Heapify(A, 1)
}

BuildHeap(int *A) {
    n = elements_in(A)
    for i = floor(n/2) to 1
        Heapify(A,i,n)
}

Heapify(int* A,int i,int n) {
    left = 2i
    right = 2i+1
    if ((left <= n) && (A[left] > A[i]))
        max = left
    else
        max = i
    if ((right<=n) && (A[right] > A[max]))
        max = right
    if (max != i) {
        swap(A[i], A[max])
        Heapify(A, max)
    }
}
```

Heap sort

- Example:
  - ➢ The **left** child of index i : index 2*i+1
  - ➢ The **right** child of index i : index 2*i+2
    - the children of node 3 (19)
    - are 7 (18) and 8 (14)



0	1	2	3	4	5	6	7	8	9	10	11	12
25	22	17	19	22	14	15	18	14	21	3	9	11

Heap sort

- Remove/replace the root
  - The “root” is the first element in the array
  - The “rightmost node at the deepest level” is the last element
  - Swap them...

0	1	2	3	4	5	6	7	8	9	10	11	12
25	22	17	19	22	14	15	18	14	21	3	9	11

0	1	2	3	4	5	6	7	8	9	10	11	12
11	22	17	19	22	14	15	18	14	21	3	9	25

  - ...And pretend that the last element in the array no longer exists—that is, the “last index” is 11 (9)

Heap sort

- Reheap, …
  - ➢ Reheap the root node (index 0, containing 11)…
  - ➢ …and continue…, remove and replace the root node
  - ➢ Remember that the **last** array index is changed
  - ➢ Repeat until the **last** becomes **first**
    - →the array is sorted!

0	1	2	3	4	5	6	7	8	9	10	11	12
11	22	17	19	22	14	15	18	14	21	3	9	25

0	1	2	3	4	5	6	7	8	9	10	11	12
22	22	17	19	21	14	15	18	14	11	3	9	25

0	1	2	3	4	5	6	7	8	9	10	11	12
9	22	17	19	22	14	15	18	14	21	3	22	25

Heap sort

- Algorithm
  - C++
    - Example of implementation

```cpp
void swap(int *x, int *y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}

void MaxHeapfiy(int* a,int i,int n) {
    int left, right, largest;
    left=2*i+1;
    right=2*i+2;
    largest = i;
    if ((left<n) && a[left]>a[i])
        largest=left;
    if ((right<n) && (a[right]>a[largest]))
        largest=right;
    if (largest!=i) {
        swap(&(a[i]),&(a[largest]));
        MaxHeapfiy(a,largest,n);
    }
}

void BuildMaxHeap(int* a, int n) {
    for (int k=n/2-1;k>=0;k--) {
        MaxHeapfiy(a,k,n);
    }
}

void HeapSort(int* a, int n) {
    BuildMaxHeap(a,n);
    int i;
    for (i=n-1;i>0;i--) {
        swap(&(a[0]),&(a[i]));
        MaxHeapfiy(a,0,i);
    }
}
```

Warning!
Don't be confused with
Indices (left,right, largest,i)
and
Values (a[left],a[right],a[largest],a[i])

Heap sort

- Conclusion
  - Heapsort is *always* O(n log n)
    - Comparison with Quicksort:
      - O(n log n) but in the worst case slows to $O(n^2)$
      - generally faster,
        - **but** Heapsort is better in time-critical applications
- Heapifying
  - O(n log n) time
- while loop
  - O(n log n) time
- Total time → O(n log n) + O(n log n) → same as O(n log n) time

Questions ?

- Reading & Acknowledgement
  - Chapter 19, Fibonacci heaps, Introduction to Algorithms, 3rd Edition