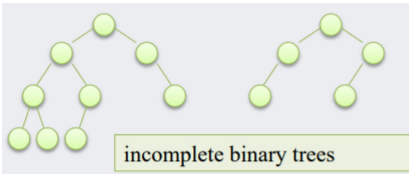
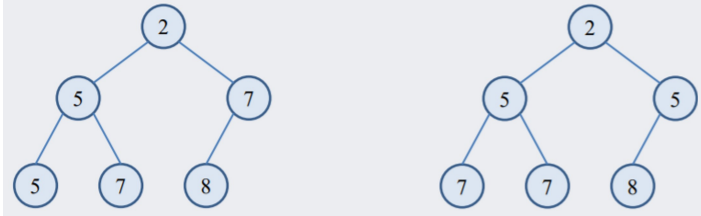
Insertion Removal Heapsort

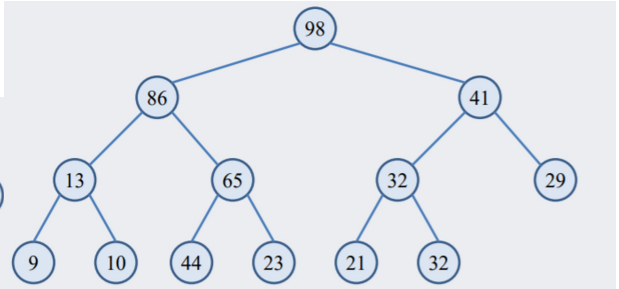
November 13, 2019 8:08 AM

Binary heap



- heap is binary tree with 2 properties
  - Complete
    - all levels except bottom must be completely filled in
    - Leaves on the bottom level are as far to left as possible
  - Partially Ordered
    - For max heap, value of node at least as large as its children's values
    - For min heap, value of node is no greater than its children's values
- **Duplicate priority values**
  - two binary heaps can contain same data, but items may appear in different positions

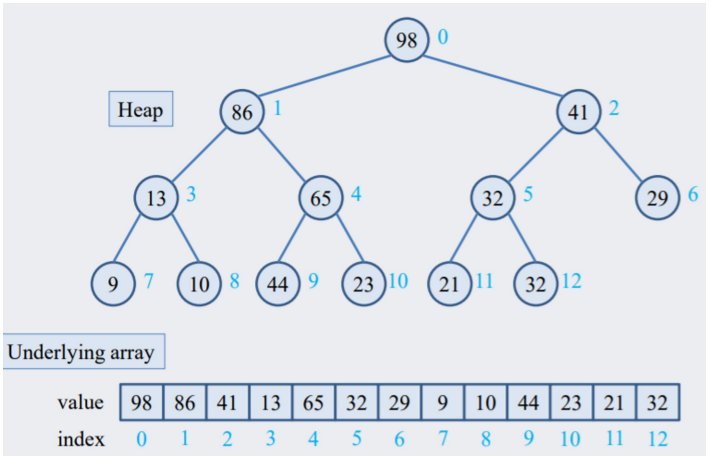




Heaps are *not* fully ordered – an in-order traversal would result in:

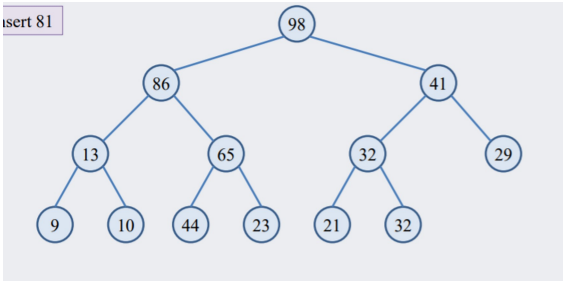
9, 13, 10, 86, 44, 65, 23, 98, 21, 32, 32, 41, 29

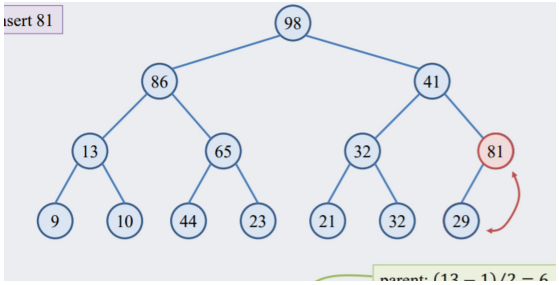
- **Heap implementation**
  - can be implemented using arrays
  - natural method of indexing tree nodes
  - Index nodes from top to bottom and left to right
  - heaps are complete binary trees, no gaps in array
- **Referencing nodes**
  - to move, find index of parent of a node or children
  - array indexed from 0 to n-1
  - Each level's nodes indexed from 2^{level}-1 to $2^{level+1}$-2 (root = 0)
  - children of node i are the array elements indexed at 2i+1 and 2i+2
  - parent of node is array element at (i-1)/2

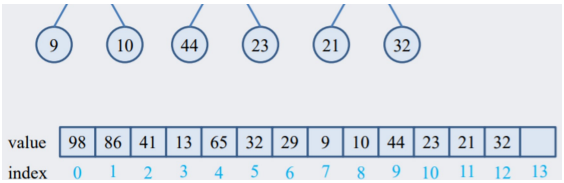


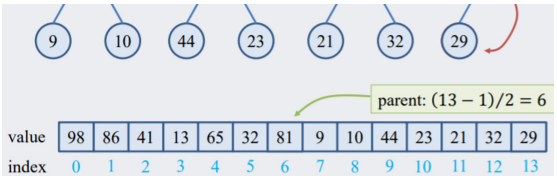

```
typedef struct MinHeap {
  int size;     // number of stored elements
  int capacity; // maximum capacity of array
  int* arr;     // array in dynamic memory
} MinHeap;
void initializeMinHeap(MinHeap* h, int initcapacity) {
  h->size = 0;
  h->capacity = initcapacity;
  h->arr = (int*) malloc(h->capacity * sizeof(int));
}
```

- **Heap insertion**
  - heap properties have to be maintained
  - 2 general strategies that could be used to maintain heap properties
    - make sure tree is complete and fix ordering
    - make sure ordering is correct first
  - insertion algorithm ensures tree is complete
    - make new item in first available (leftmost) leaf on bottom level
    - fix partial ordering, compare new value to parent, swap if new value greater than parent

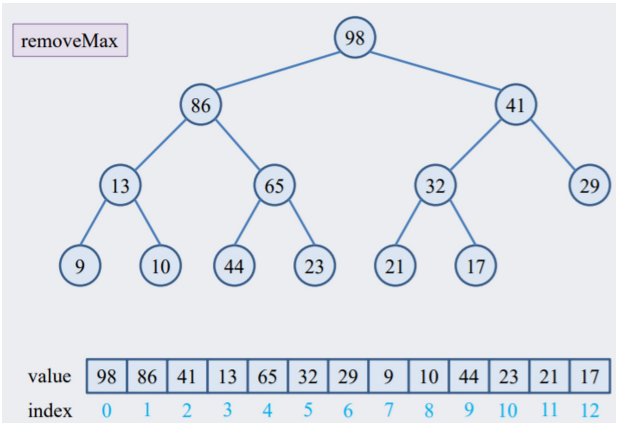


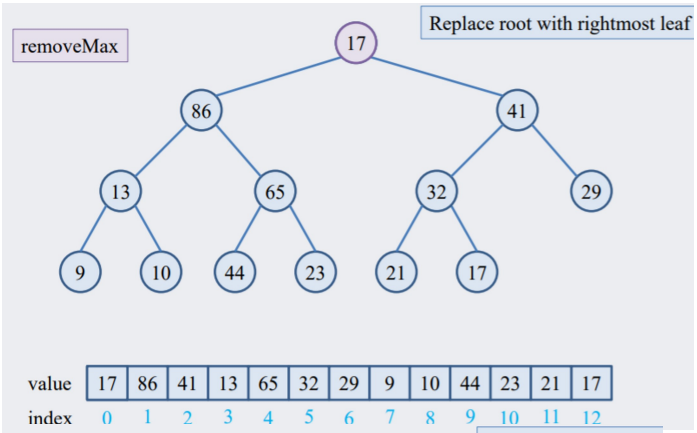


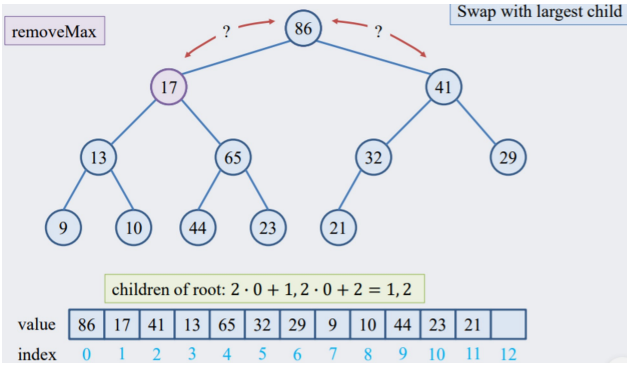


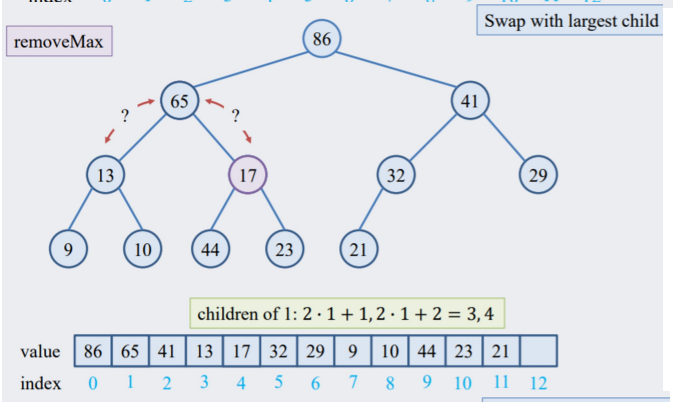


- **Complexity**
  - Item inserted at bottom level in first available space, tracked using size attribute
  - O(1) access using array index
  - Repeated heapify-up operation, moves inserted value up on level in tree
  - Upper limit on number of levels in complete tree O(logn)
  - Heap insertion has worst case performance of O(logn)
- **Removing priority item**
  - properties ust be satisfied after removal
  - make temp copy of root's data
  - heap remains complete, replace root node with right most leaf
  - swap new root with largest valued child until partially ordered property holds, return copied root's data

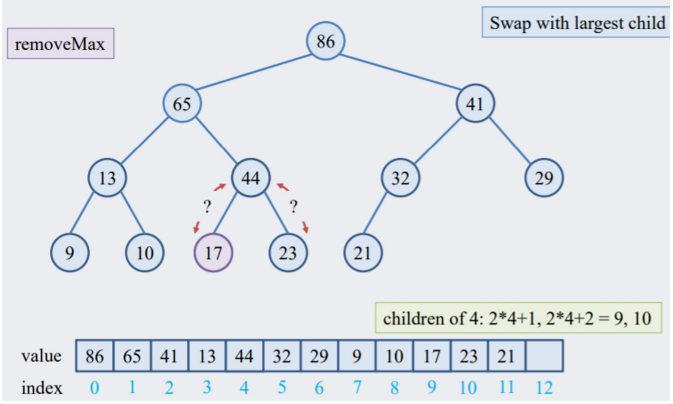








- **Complexity**
  - Similar to insertion
  - replace root with last element - O(1)
  - repeated heapify-down operations from root
    move 1 level closer to bottom
    how many levels? O(logn)
  - worst case O(logn)



Array implementation and instertion

- expand when array is full
- copy array contents into new array same indices

Sorting with heaps

- Removal of node from heap performedin O(logn) time
- Nodes removed in order
- Removing all nodes one by one result in sorted output

- Removal of all nodes from heap is a O(nlogn) operation
- heap is an array, can be used to return sorted data in O(nlogn) time
- cant assume that data to be sorted happens to be a heap
    - can put it in a heap
    - inserting in a heap is a O(logn) operation, so inserting n items is O(nlogn)
- buildHeap
    - to create a feap from unordered array, repeatedly call heapifyDown