Abstract Data Types

Abstract Data Types

- An ***abstract data type*** is a set of values (represented by a ***model***) and a set of operations (***methods***) that can be performed on those values
- Abstract because it separates the ***specification*** (*what* you can do with the objects) from the ***implementation*** (*how* the objects are represented with ***state*** variables and how the operations realize the desired behavior)
  - Access to the data is exclusively through an ***interface*** that prescribes *what* the methods are and how they are invoked and what the parameters are.
  - Specification includes an unambiguous description of the behavior of the operations, without specifying **how** this behavior is implemented.
- Advantages
  - Code is easier to understand ⇒ more likely to be correct
  - Implementation can change (e.g., for efficiency) without requiring changes to client code (code that uses the ADT)
  - Promotes reusability

Abstract Data Types

- Each abstract data type consists of two components:
  1. The *public* or *external* portion, which consists of:
     - A conceptual or user's view of what the objects look like
     - The *methods* or conceptual operations available to the users of the type
  2. The *private* or *internal* portion, which consists of:
     - The object representation or *state* (how each object is actually stored)
     - The *implementation* of the public methods
     - The implementation of some internal methods (not available directly to users of the ADT)

Abstract

Abstract Data Types

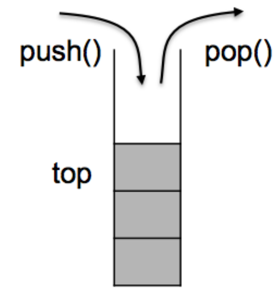
- Methods usually fall into one of the following categories:
  - Initialization—to be used when an object is created
  - State changing (e.g., adding or removing data to/from the object)
  - Access—to query different portions of the data
  - Destruction—to eliminate an object

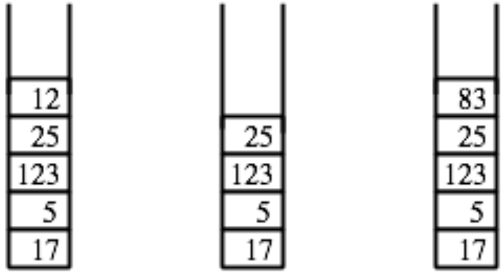
Data Structures

- A ***data structure*** is a policy for storing a collection of data values in computer memory with the goal of supporting a specific set of operations efficiently
  - *Example:* given a list of values $\langle x_1, x_2, \ldots, x_n \rangle$, determine if a query value x appears in the list, find the smallest value in the list, find the median, and so on
- Used to implement an abstract data type (ADT)
  - The ADT defines the *logical form* of the data
  - The data structure implements the *physical form* via *state* variables
  - *Example:* dictionaries, adjacency matrix, adjacency list

Stack ADT

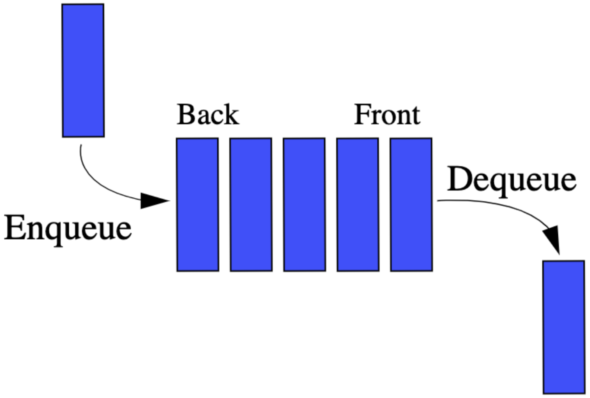
- A **stack** is a dynamic set (i.e., supports inserts and deletes); a delete (*pop*) removes the most recently inserted element
- Enforces a last-in, first-out (LIFO) policy
- Operations
  - stack(*Type*) → *Stack*
    - Constructor, creates an empty set of elements of type *Type*
  - empty(*Stack*) → *Boolean*
    - True if set has no elements
  - top(*Stack*) → *Type*
    - The most recently inserted element
  - push(*Stack*, *Type*) → *Stack*
    - Adds an element to the set
  - pop(*Stack*) → *Stack*
    - Removes the most recent element in the set (error if empty)





Original stack. After pop(). After push(83).

Queue ADT



- A *queue* is a dynamic set that supports inserts and deletes; a delete operation (*dequeue*) removes the element that has been in the set for the longest time
- Enforces a first-in, first-out (FIFO) policy
- Operations
  - queue(*Type*) → *Queue*
    - Constructor, creates an empty set of elements of type *Type*
  - empty(*Queue*) → *Boolean*
    - True if set has no elements
  - enqueue(*Queue*, *Type*) → *Queue*
    - Adds an element to the set
  - dequeue(*Queue*) → *Type*
    - Removes and returns the oldest element in the set (error if empty)

Implementing ADTs

- In Python, ADTs are implemented using the **class** *type.*
- A class definition creates an object of type *type* and associates with it a structure consisting of state and methods for that class.
- Some "special" methods start and end with two underscores.
    - Such methods can be invoked using simpler syntax, compatible with that of built-in types.

Implementing ADTs

- Special methods include:
  - ***__init__(self)*** is a constructor—when the interpreter creates a new instance of the class (e.g., myDie = Die() calls it to initialize data members)
  - ***__str__(self)*** is invoked when the print() command is executed on an instance of the class; all it needs to do is create a string representation of an object
- Attributes may be *private* or *public.*
  - Data attributes are private, while method attributes are public
  - Method interfaces should *never* refer to data attributes

Example: A Stack of Integers

```
class intStack(object):
    def __init__(self):
        self.state = []
    def push(self, elem):
        """Adds an element to the top of a stack"""
        self.state.append(elem)
    def empty(self):
        """True iff stack is empty"""
        return len(self.state) == 0
    def pop(self):
        """ Removes the top of a nonempty stack"""
        if not self.empty():
            self.state.pop()
    def top(self):
        """ Returns the top of a nonempty stack"""
        if self.empty():
            raise ValueError("Requested top of an empty stack")
        else:
            return self.state[-1]
```

A Parameterized Stack Class

Parameter →

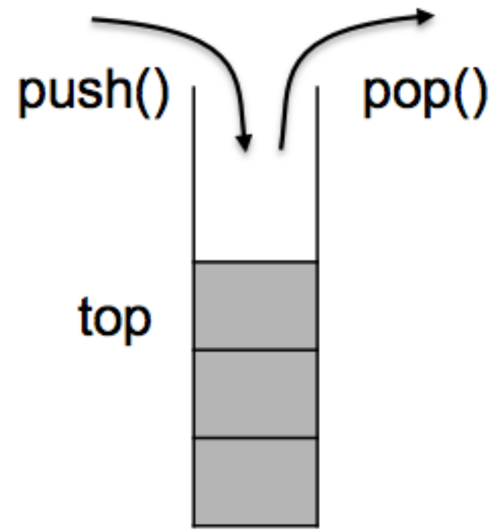
```
class Stack(object):
     def __init__(self, type):
         self.elemType = type
         self.state = []
    def push(self, elem):
         """Adds an element to the top of a stack"""
         assert type(elem) == self.elemType
         self.state.append(elem)
    def empty(self):
          return len(self.state) == 0
    def pop(self):
         """ Removes the top of a nonempty stack"""
         if not self.empty():
              self.state.pop()
    def top(self):
         """ Returns the top of a nonempty stack"""
         if self.empty():
              raise ValueError("Requested top of an empty stack")
         else:
              return self.state[−1]
```

END

Stacks

Stacks

- Recall that stacks follow a LIFO policy.
- There are several ways to implement a stack.
- The most popular is using a list.
  - But do you push/pop from the front or end of the list?
  - Does it even matter?



Stack: List Implementation, Push/Pop Front

Create empty

0	1	2	3	4	5	6	7

Push A

A							
0	1	2	3	4	5	6	7

Push B

B	A						
0	1	2	3	4	5	6	7

Push C

C	B	A					
0	1	2	3	4	5	6	7

Stack: List Implementation, Push/Pop Front

C	B	A					
0	1	2	3	4	5	6	7

Pop

B	A						
0	1	2	3	4	5	6	7

Pop

A							
0	1	2	3	4	5	6	7

Stack: List Implementation, Push/Pop Rear

Create empty

0	1	2	3	4	5	6	7

Push A

A							
0	1	2	3	4	5	6	7

Push B

A	**B**						
0	1	2	3	4	5	6	7

Push C

A	**B**	**C**					
0	1	2	3	4	5	6	7

Stack: List Implementation, Push/Pop Rear

A	B	C					
0	1	2	3	4	5	6	7

Pop

A	B						
0	1	2	3	4	5	6	7

Pop

A							
0	1	2	3	4	5	6	7

Stack: List Implementation

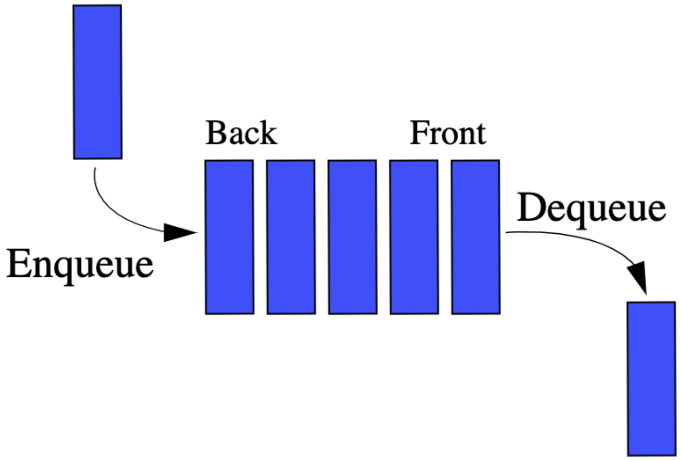
- O(n) to push/pop from the front
- O(1) to push/pop from the end
  - stlist = [ ]
  - stlist.append(element)
  - stlist.pop()

END

Queues

Queues

- Recall that queues follow a FIFO policy.
- There are several ways to implement a queue.
  - As a basic list
  - As a circular list
  - As a doubly linked list



Queue: List Implementation

Create empty

0	1	2	3	4	5	6	7

Enqueue A

A							
0	1	2	3	4	5	6	7

Enqueue B

B	A						
0	1	2	3	4	5	6	7

Enqueue C

C	B	A					
0	1	2	3	4	5	6	7

Queue: List Implementation

C	B	A					
0	1	2	3	4	5	6	7

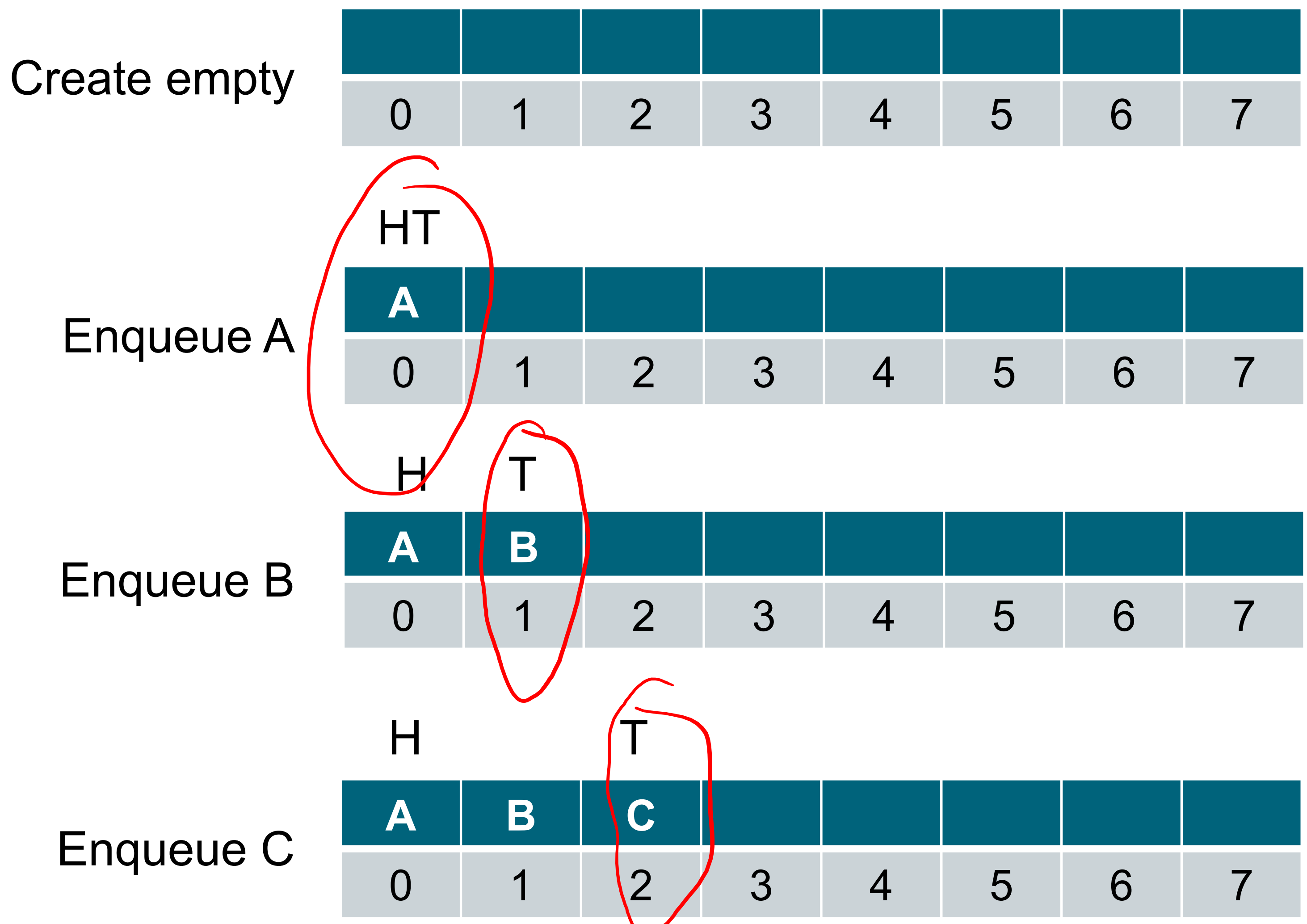
Dequeue

C	B						
0	1	2	3	4	5	6	7

Dequeue

C							
0	1	2	3	4	5	6	7

Queue: Circular List Implementation



Queue: Circular List Implementation

H		T					
A	**B**	**C**					
0	1	2	3	4	5	6	7

Dequeue

	H	T					
	B	**C**					
0	1	2	3	4	5	6	7

Dequeue

		HT					
		C					
0	1	2	3	4	5	6	7

Enqueue D

		H	T				
		C	**D**				
0	1	2	3	4	5	6	7

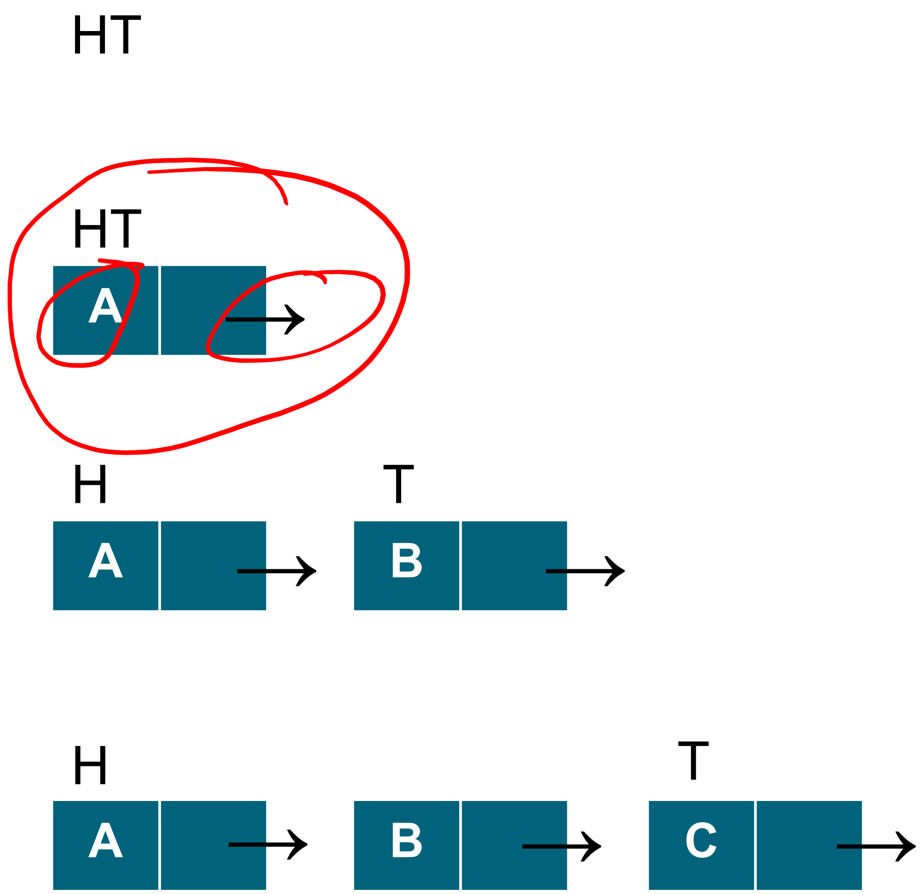
Queue: Linked List

Create empty

Enqueue A

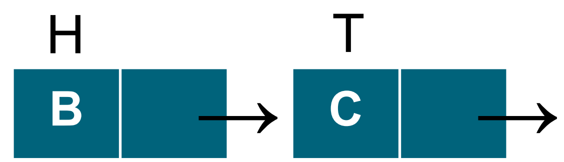
Enqueue B

Enqueue C



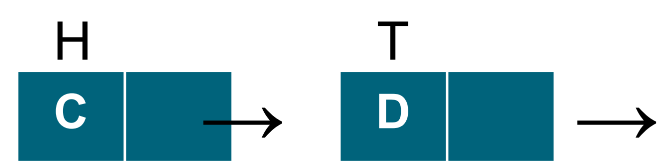
Queue: Linked List

Dequeue



Dequeue

Enqueue D



Queue: Implementations

- List: O(n) to enqueue, O(1) to dequeue
- Circular list: O(1) to enqueue/dequeue
- Doubly linked list: O(1) to enqueue/dequeue
  - q = dequeue() # from collections (“deck”)
  - q.append(elementt)
  - q.popleft()

END

Priority Queue

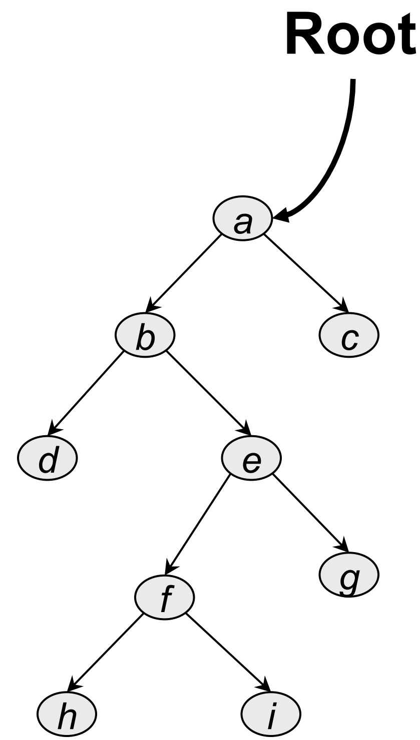
Priority Queues

- ADT to keep track a dynamic set of elements with support for the following operations:
  - Insert(S, x): add element x to S
  - Max(S): return the maximum of S
  - ExtractMax(S): remove maximum from S

  $\boldsymbol{O(log\, n)}$ **time with heaps**
- Like a queue were one can cut in line
- *Applications*: task scheduling, simulation, greedy algorithms, Huffman coding, and so on
- Data structure: *binary heap*

Rooted Trees

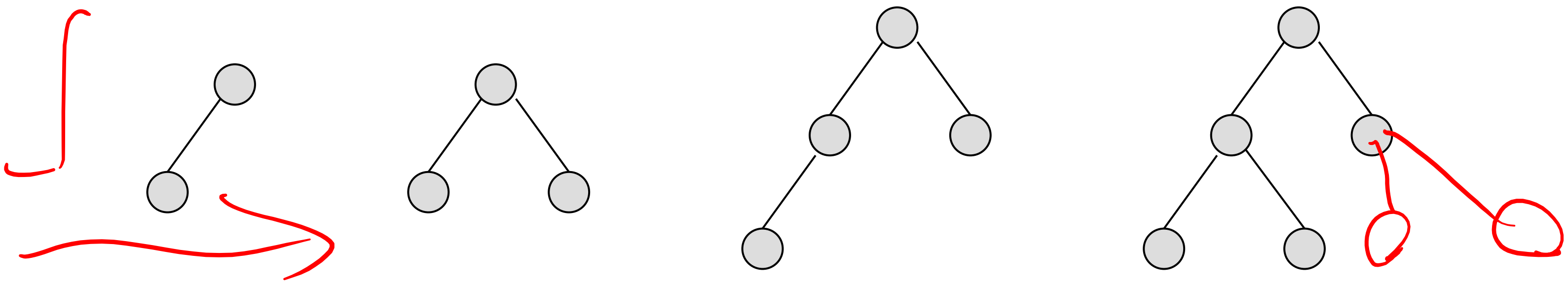
- A rooted tree is a directed graph with no cycles.
  - There is a designated root node with indegree 0
  - Every other node has indegree = 1
- We consider binary trees .
  - All nodes have outdegree ≤ 2
- Do you know what the following are?
  - Leaf, internal node, sibling, parent, child, ancestor, descendant, degree, full tree, complete tree, height, depth



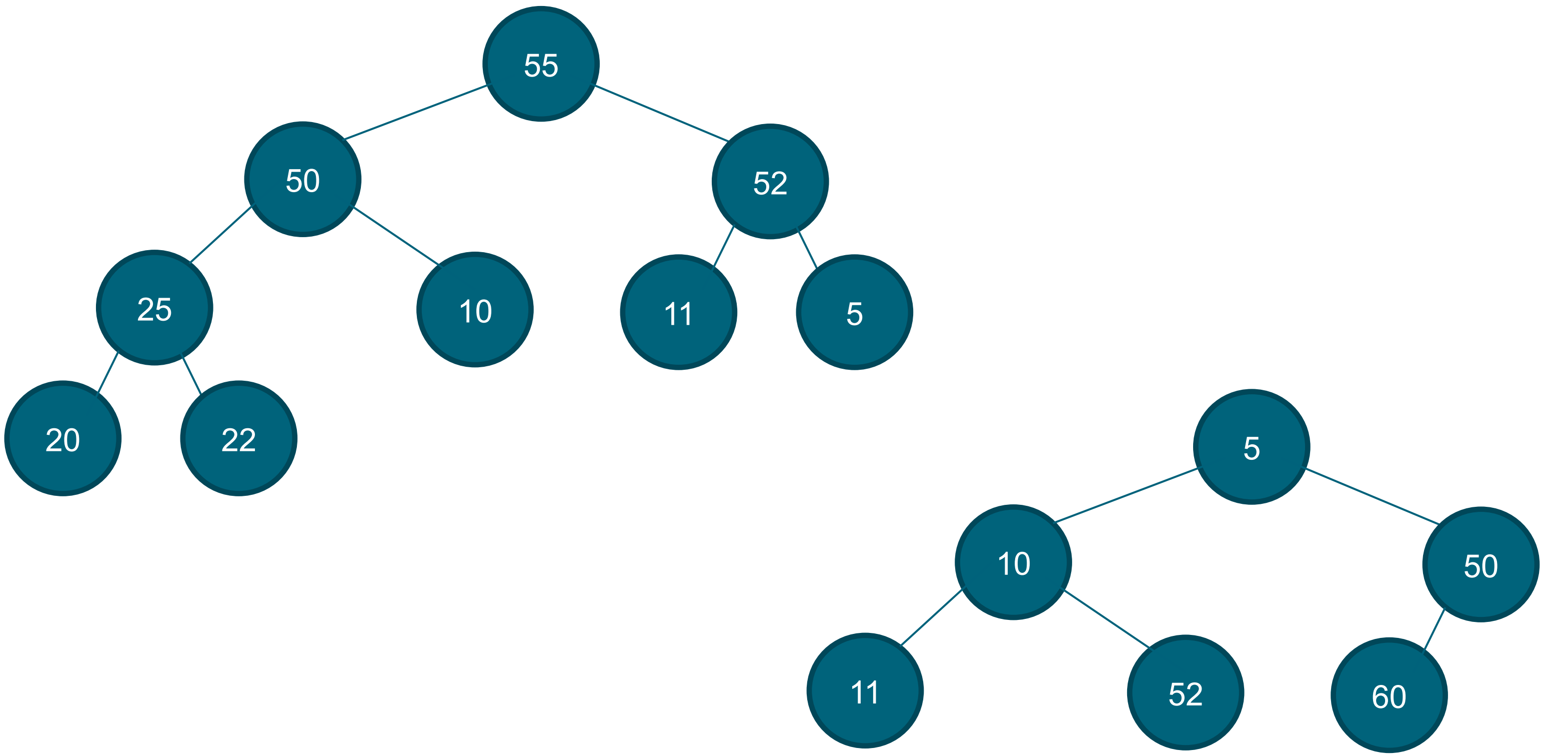
Heaps

A heap is a rooted binary tree H that satisfies two properties:

1. *Structural property*
   - (Almost) complete binary tree (it fills from top to bottom and, at each level, from left to right)
2. *Order or heap property* (for maximum heaps)
   - $H(\text{parent}(v)) \geq H(v)$, for all nodes v

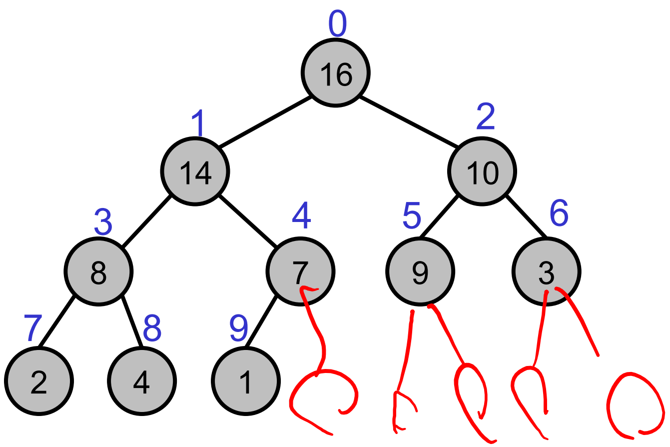


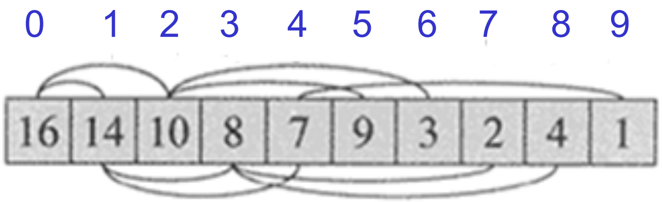
Heap Examples



List Representation of Heaps

- The tree is (almost) complete.
  - All levels are full, except possibly the last.





- Where are the children of *H*[*i*]? The parent?

```
def Left(i):
    return 2*i+1
```

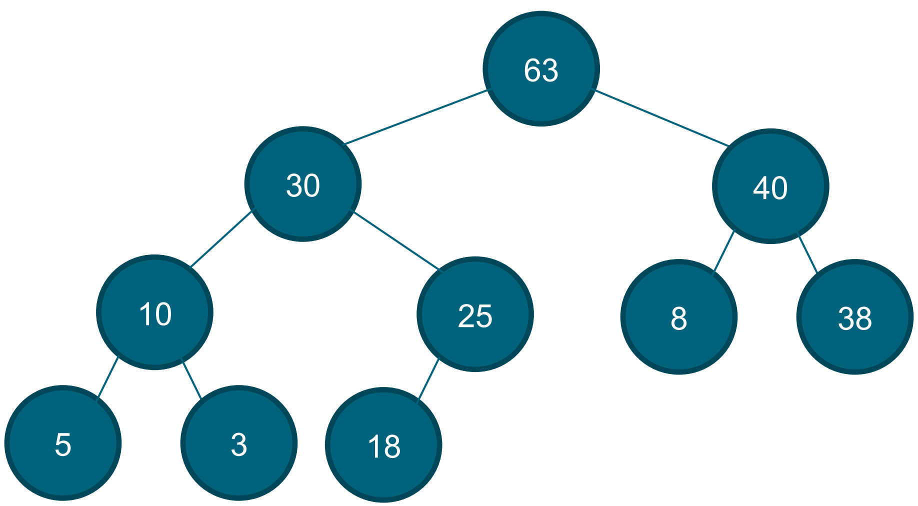
```
def Right(i):
    return 2*i+2
```

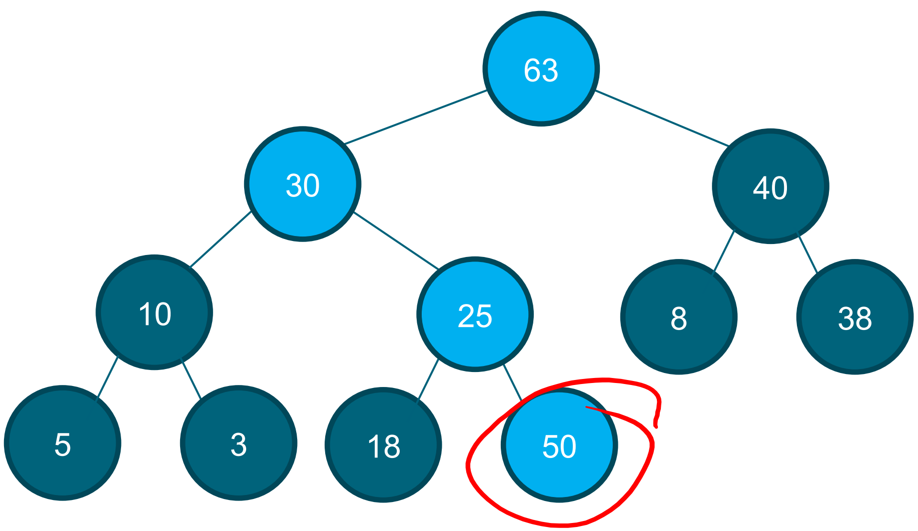
END

Heap Insertion and Removal

Inserting Into a Heap

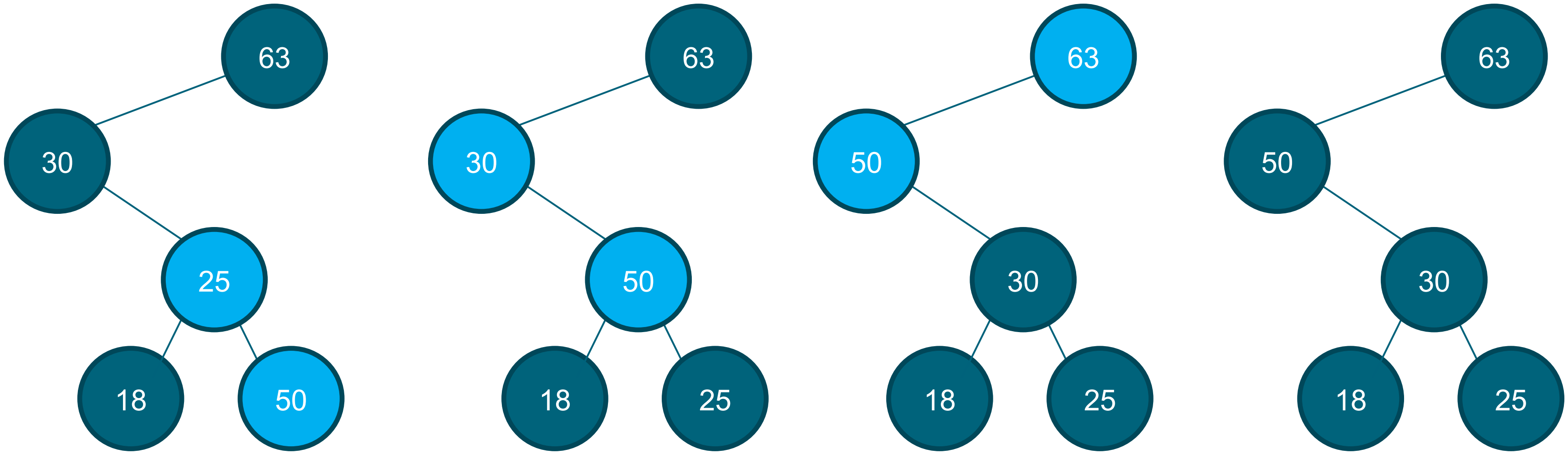
- Add the new element (50) in the next structural position in the heap.
  - Note that this is the rightmost position in the list.
  - Values from root to new node may need to be adjusted to maintain the heap property.





Inserting Into a Heap

- Compare the new element with its parent.
  - If bigger, then trade places and repeat.



Inserting Into a Heap Analysis

- The new node goes into the known position
  - Takes fixed time
- At most, we walk up one full branch of the tree to the root
  - Full binary tree of n nodes has height of log *n*
- Insert into heap has time $O(\log_2 n)$

$\log_{10}$ $\ln_e$

Insert

```
def HeapInsert(H,key):
    H.append(key)
    H[len(H)-1] = -math.inf
    IncreaseKey(H,len(H)-1,key)
```

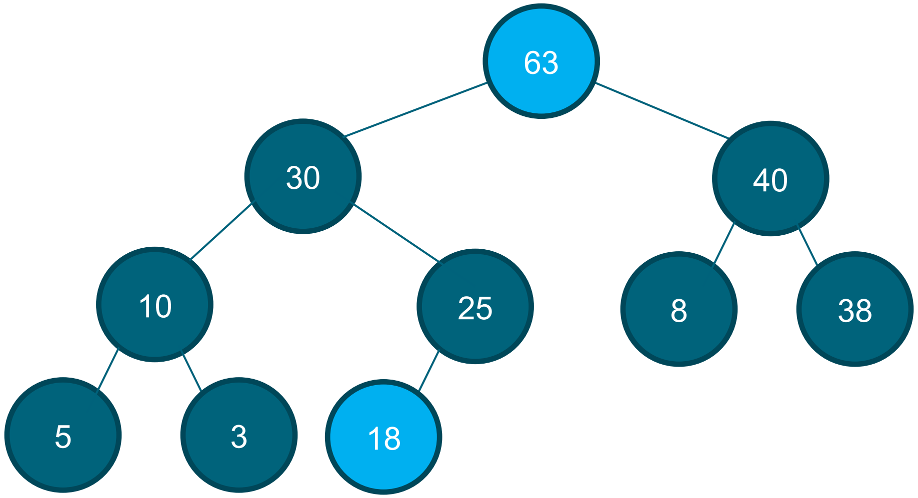
IncreaseKey

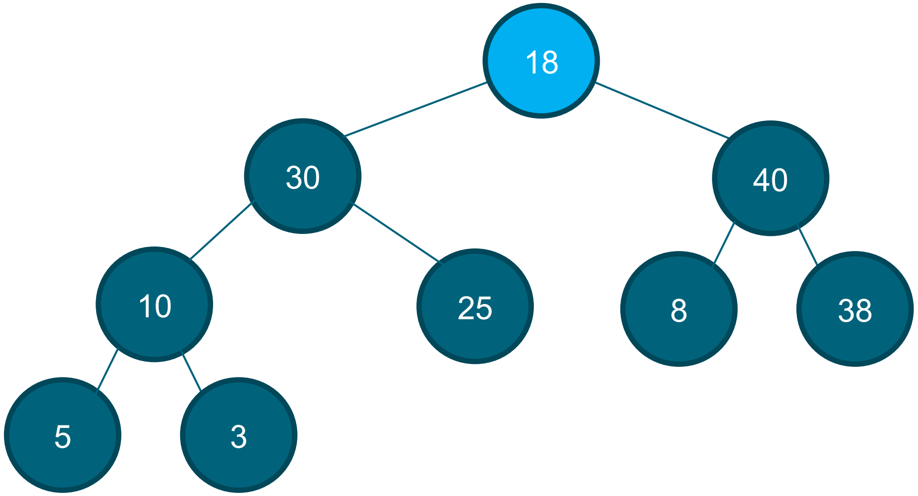
```
def IncreaseKey(H,i,key):
    assert key >= H[i]
    H[i] = key
    while i>0 and H[Parent(i)]<H[i]:
        H[i],H[Parent(i)] = H[Parent(i)],H[i]
        i = Parent(i)


def Parent(i):
    return (i-1)//2
```

Remove From a Heap

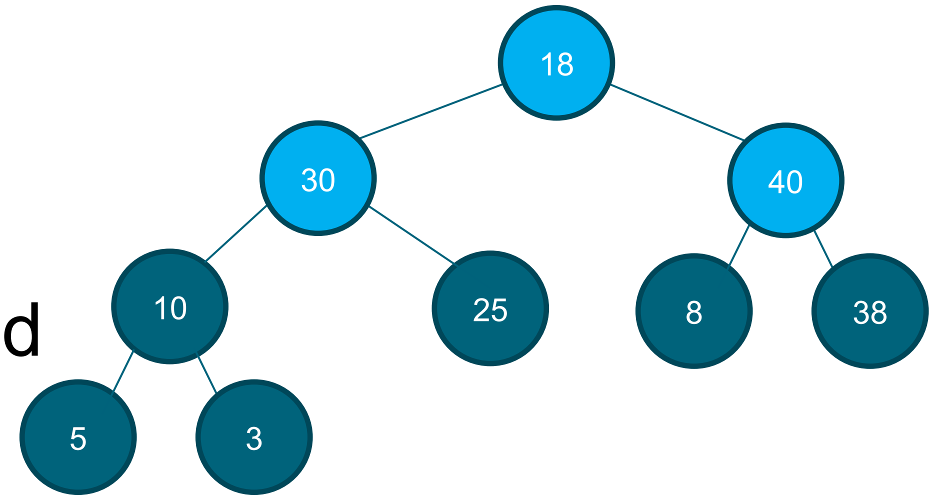
- Removal is restricted to the root node only
  - The root always contains the maximum element
- Can't leave an open hole at the root
  - Structure dictates that it must be the last position
  - Swap root and last values

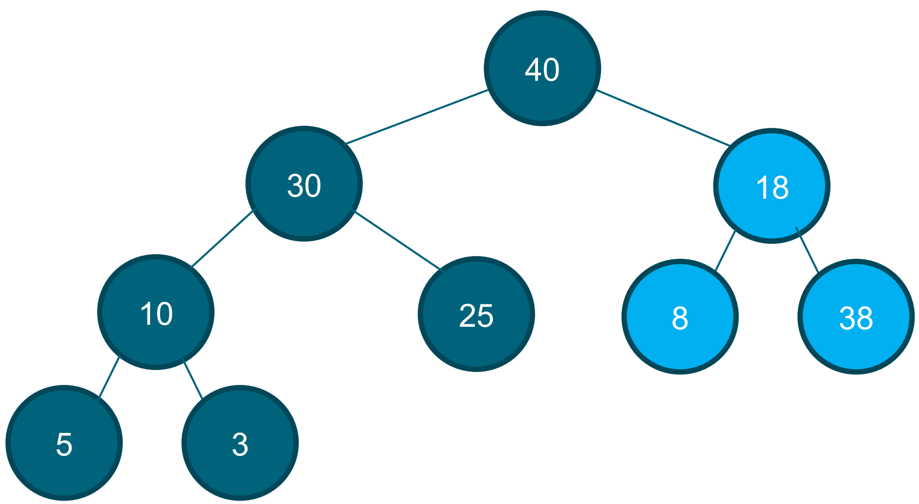


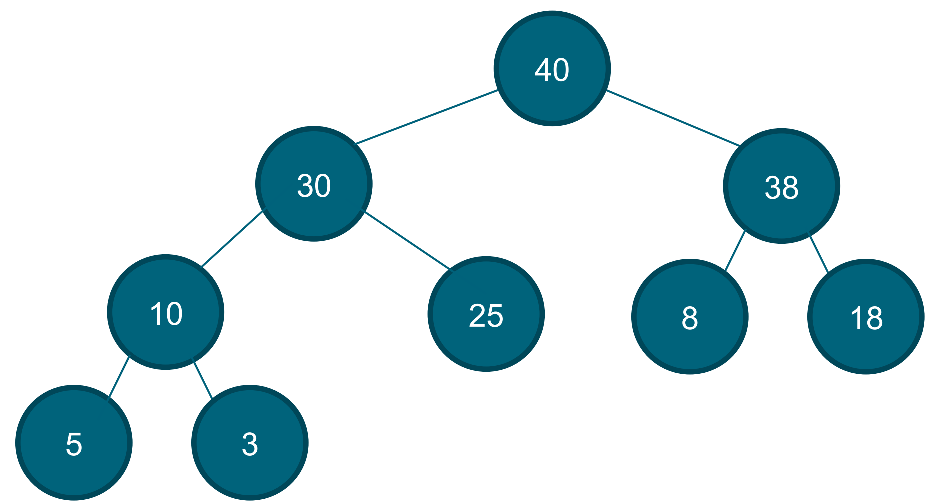


Remove From a Heap

- Need to fix the heap property (heapify)
  - Swap with larger child if out of order







Remove From a Heap Analysis

- Root value gets replaced with the last value
  - Takes fixed time
- At most, we walk down one full branch of the tree from root to leaf
  - Full binary tree of n nodes has height of log *n*
- Remove from heap has time $O(\log n)$

ExtractMax

```
def HeapExtractMax(H):
    assert not Empty(H)
    maximum = H[0]
    H[0] = H[len(H)-1]
    H.pop()
    MaxHeapify(H, len(H), 0)
    return maximum
```

Heapify

```
def MaxHeapify(H,n,i):
    left = Left(i)
    right = Right(i)
    if left < n and H[left]>H[i]:
        largest = left
    else:
        largest = i
    if right < n and H[right]>H[largest]:
        largest = right
    if largest != i:
        H[i],H[largest] = H[largest],H[i]
        MaxHeapify(H,n,largest)
```

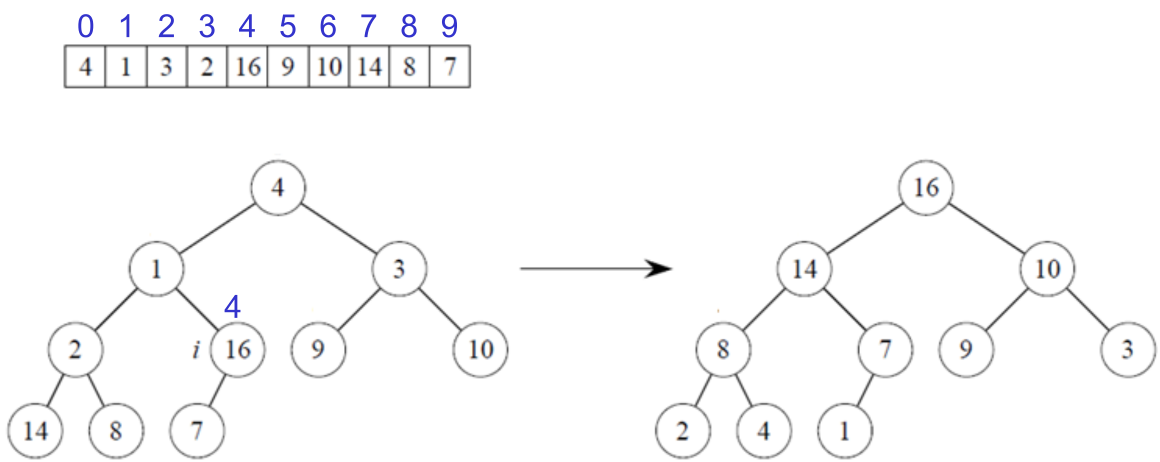
END

Heap Sort

Building a Heap From a Plain List

Start with an arbitrary unsorted list *H*.

```
def BuildMaxHeap(H):
    for i in range(len(H)//2-1,-1,-1):
        MaxHeapify(H,len(H),i)
```



Analysis

- A simple bound
  - $O(n)$ calls to heapify, each of which takes $O(\log n)$
  - Time $\Rightarrow O(n \lg n)$

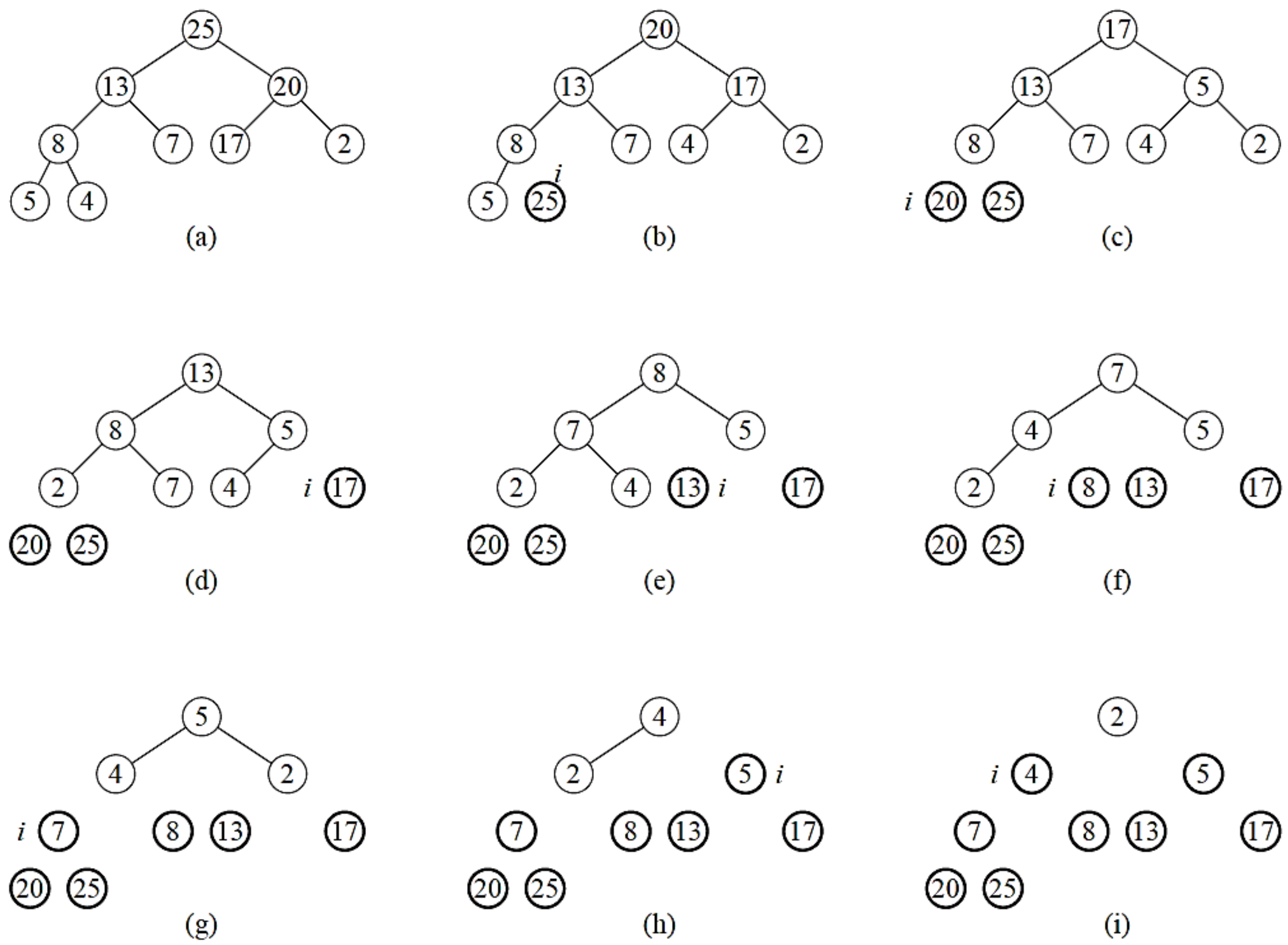
Heap Sort

- **Idea:** after creating a max-heap, output the elements in descending order, one at a time
- Analysis
  - Build-heap: $O(n \log n)$
  - n removals
    - Remove maximum elements: $O(\log n)$
  - Total time: $O(n \lg n)$
- Though heapsort is a fast algorithm, a well-implemented quicksort **usually** runs faster

n log n +

n log n

Example: Heapsort



END

Acknowledgements

- Introduction to Algorithms, 3rd edition, by T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein; MIT Press, 2009