DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 3 | Part 1

Abstract Data Types

Python's `list`

- You can go a long time without ever knowing how `list` is **implemented**.
- But you knew its **interface**.
    - supports `.append`, random access, is ordered, etc.

Abstract vs. Concrete

- An **abstract data type** (ADT) is a formal description of a type's **interface**.

- A **data structure** is a concrete strategy for implementing an abstract data type.
  - Describes how data is stored in memory.
  - How to access the data.

Example: Stacks

- A **stack** is an ADT which supports two operations:
  - push: put a new object on to the "top"
  - pop: remove and return item at the "top"
- Most often implemented using **linked lists**.
- But can also be implement with **(dynamic) arrays**.

Main Idea

A given abstract data type can be implemented in several ways, but some data structures are more natural choices than others.

Main Idea

The data structure (not the abstract data type) determines the time complexity of operations.

Building Blocks

- Data structures are used to implement ADTs.
- But they are also used to implement more advanced data structures.
  - Example: arrays used to implement dynamic arrays.
- Arrays, linked lists are basic building blocks.

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 3 | Part 2

Priority Queues

Priority Queues

- A **priority queue** is an abstract data type representing a collection.
- Each element has a **priority**.
- Supports operations[1]:
  - `.pop_highest_priority()`
  - `.insert(value, priority)`
  - `.is_empty()`

[1] and possibly more, like `.increase_priority`

Example

```
>>> er = PriorityQueue()
>>> er.insert('flu', priority=1)
>>> er.insert('heart attack', priority=20)
>>> er.insert('broken hand', priority=10)
>>> er.pop_highest_priority()
'heart attack'
>>> er.pop_highest_priority()
'broken hand'
```

Applications

- Scheduling.
- Simulations of future events.
- Useful in algorithms.
  - Example: Prim’s MST algorithm

Array Implementation

- We *can* implement a priority queue with a **(dynamic) array**.

- `.insert(k, p)`
  - append (value, priority) pair: $\Theta(1)$ time

- `.pop_highest_priority()`
  - find entry with highest priority: $\Theta(n)$ time
  - remove it: $O(n)$ time

Array Implementation (Variant)

- Alternatively, maintain dynamic array in sorted order of priority.

- `.insert(k, p)`
  - find place in sorted order: $\Theta(\log n)$ time worst case
  - actually insert: $\Theta(n)$ time worst case

- `.pop_highest_priority()`
  - remove/return last entry: $\Theta(1)$ time

Main Idea

If we made no insertions/deletions, a sorted array would be great. But we want a data structure with quick remove/return even after being modified.

DSC 190

Data Structures & Algorithms

Lecture 3 | Part 3

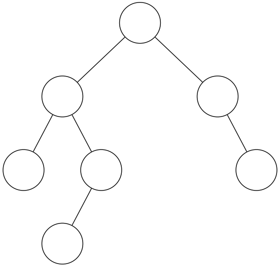
Binary Heaps

Binary Heaps

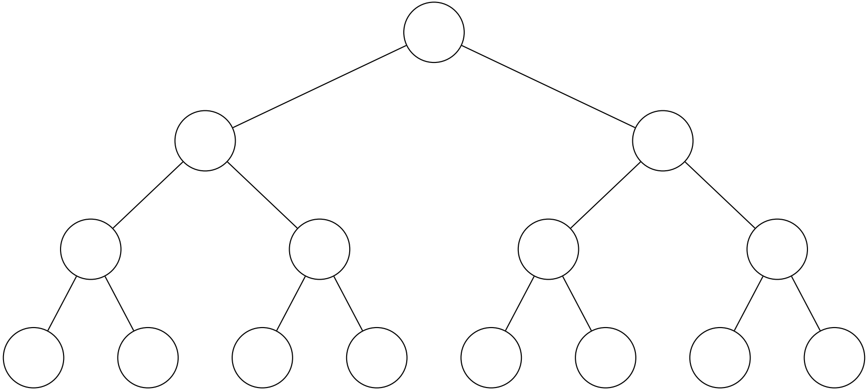
- A **binary heap** is a **binary tree** data structure often used to implement **priority queues**.

Binary Trees

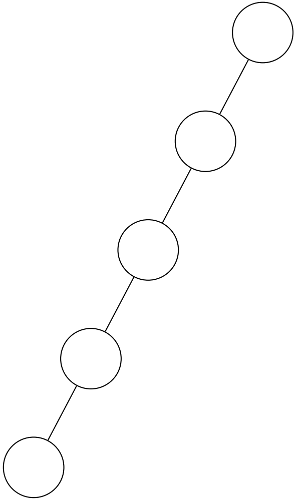
- Each node has **at most** two children (left, right).



Example

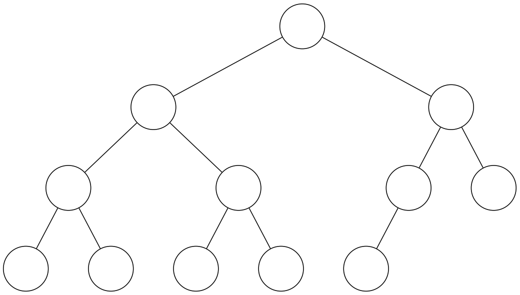


Example



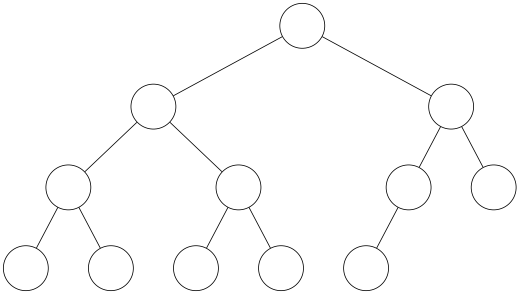
Complete Binary Trees

- A binary tree is **complete** if every level is filled, except for possibly the last (which fills from left to right).



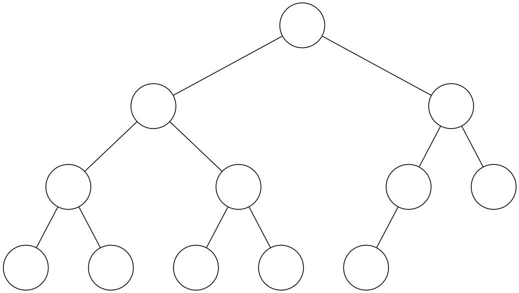
Node Height

- The **height** of node in a tree is the largest number of edges along any path to a leaf.
- The **height** of a tree is the height of the root.



Complete Tree Height

- The height of a complete binary tree with n nodes is $\Theta(\log n)$.



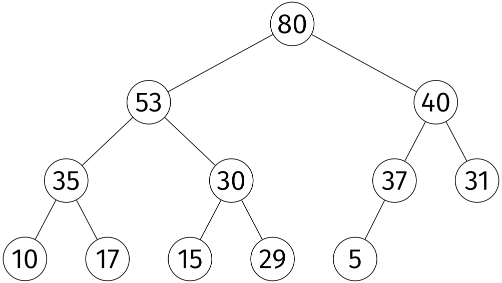
Binary Heap Properties

- A **binary max heap**[2] is a binary tree with three additional properties:
  1. Each node has a **key**.
  2. **Shape**: the tree is complete.
  3. **Max-Heap**: the key of a node is ≥ the key of each of its children.

[2] There's also a min heap, of course.

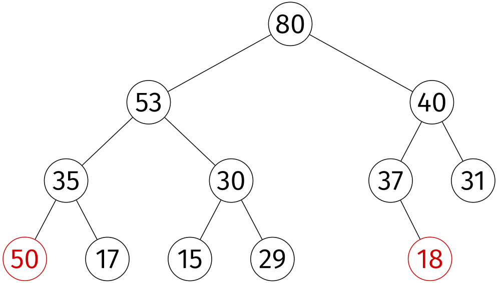
Example

- This is a binary max-heap.



Example

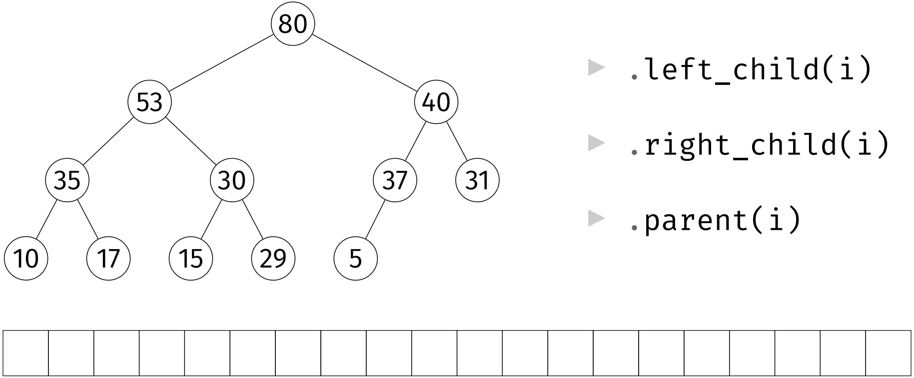
- This is **not** a binary max-heap.



Representation

- One representation: nodes are objects with pointers to children.
- But due to completeness property, we can store a binary heap in a (dynamic) array.

Array Representation

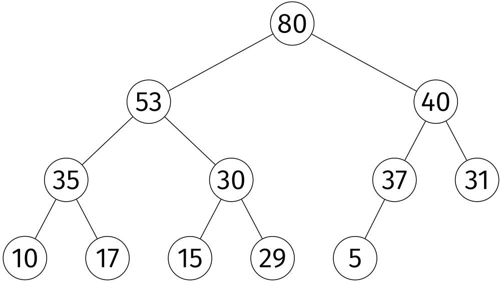


- `.left_child(i)`
- `.right_child(i)`
- `.parent(i)`

Operations

- `.max()`
  - Return (but do not remove) the max key
- `.increase_key(i, new_key)`
  - Increase key of node *i*, maintaining heap
- `.insert(key)`
  - Insert new node, maintaining heap
- `.pop_max()`
  - Remove max-key node, return key

.max



80	53	40	35	30	37	31	10	17	15	29	5
0	1	2	3	4	5	6	7	8	9	10	11

.max

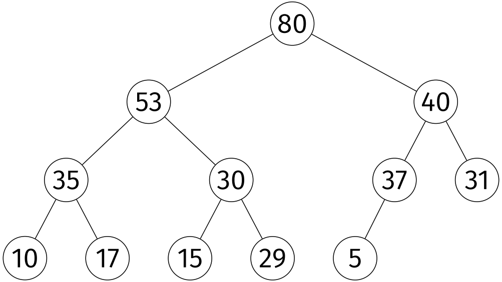
```python
class MaxHeap:

    def __init__(self, keys=None):
        if keys is None:
            keys = []
        self.keys = keys

    def max(self):
        return self.keys[0]
```

`.max`

- Takes $\Theta(1)$ time.

.increase_key

`.increase_key(9, key=60)`



80	53	40	35	30	37	31	10	17	15	29	5
0	1	2	3	4	5	6	7	8	9	10	11

.increase_key

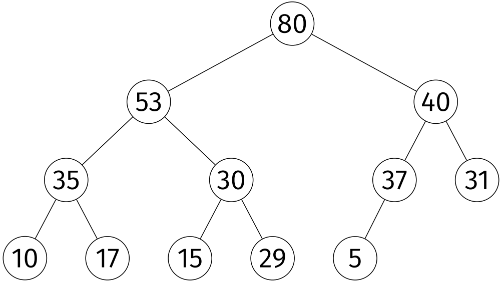
```python
def increase_key(self, ix, key):
    if key < self.keys[ix]:
        raise ValueError('New key is smaller.')

    self.keys[ix] = key
    while (
            parent(ix) >= 0
            and
            self.keys[parent(ix)] < key
        ):
        self._swap(ix, parent(ix))
        ix = parent(ix)
```

`.increase_key`

- Takes $O(\log n)$ time.

.insert

```
.insert(key=60)
```



0	1	2	3	4	5	6	7	8	9	10	11
80	53	40	35	30	37	31	10	17	15	29	5

.insert

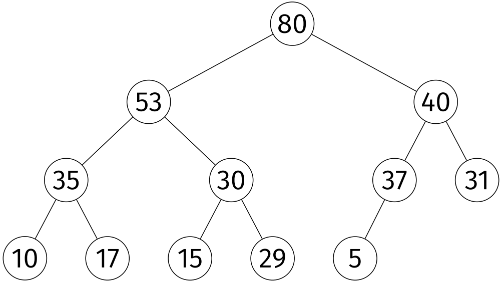
```
def insert(self, key):
    self.keys.append(key)
    self.increase_key(
        len(self.keys)-1, key
    )
```

`.insert`

- Takes $O(\log n)$ time (amortized)[3].

[3] If we use a static array the worst case is $\Theta(\log n)$

.pop_max_key



80	53	40	35	30	37	31	10	17	15	29	5
0	1	2	3	4	5	6	7	8	9	10	11

.pop_max_key

```python
def pop_max_key(self):
    if len(self.keys) == 0:
        raise IndexError('Heap is empty.')
    highest = self.max()
    self.keys[0] = self.keys[-1]
    self.keys.pop()
    self._push_down(0)
    return highest
```

`._push_down(i)`

- Assume that left and right subtrees of node *i* are max heaps, but key of *i* is possibly too small.

- Push it down until heap property satisfied.
  - Recursively swap with largest of left and right child.

._push_down()

```python
def _push_down(self, i):
    left = left_child(i)
    right = right_child(i)
    if (
            left < len(self.keys)
            and
            self.keys[left] > self.keys[i]
    ):
        largest = left
    else:
        largest = i

    if (
        right < len(self.keys)
        and
        self.keys[right] > self.keys[largest]
    ):
        largest = right

    if largest != i:
        self._swap(i, largest)
        self._push_down(largest)
```

`.pop_max_key`

- `._push_down(i)` takes $O(h)$ where h is i's height
- Since $h = O(\log n)$, `.pop_max_key` takes $O(\log n)$ time.

Summary

For a binary heap[4]:

`.max`	$\Theta(1)$
`.increase_key`	$O(\log n)$
`.insert`	$O(\log n)$
`.pop_max_key`	$O(h) = O(\log n)$

[4] There are other heap data structures. Fibonacci heaps have $\Theta(1)$ insert and increase key, but slower for small n.

Implementing Priority Queues

- Can use max heaps to implement priority queues.
- But a priority queue has values *and* keys.

```
pq.insert('heart attack', priority=20)
```

Trick

- Heap keys need not be integers.
- Need only be comparable.
- Can store key and value with a `tuple`.

Tuple Comparison

- In Python, tuple comparison is lexicographical.
  - Compare first entry; if tie, compare second, etc.

```
»> (10, 'test') > (5, 'zzz')
True
»> (10, 'test') > (10, 'zzz')
False
```

Trick

- Use 2-tuples: priority in 1st spot, value in 2nd.

```python
class PriorityQueue:

    def __init__(self):
        self._heap = MaxHeap()

    def insert(self, value, priority):
        self._heap.insert((priority, value))

    def pop_highest_priority(self):
        return self._heap.pop_max()

    def max(self):
        return self._heap.max()

    def is_empty(self):
        return not bool(self._heap.keys)
```

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 3 | Part 4

Example: Online Median

Online Median

- **Given**: a stream of numbers, one at a time.
- **Compute**: the median of all numbers seen so far.
- **Design**: a data structure with the following operations:
  - `.insert(number)`: in $\Theta(\log n)$ time
  - `.median()`: in $\Theta(1)$ time

Review

- Given an array, we can compute the median in:
  - $\Theta(n \log n)$ time by sorting
  - $\Theta(n)$ (expected) time with quickselect

- But modifying the array and repeating is costly.

Idea

- Median is the:
  - **maximum** of the smallest $\approx n/2$ numbers.
  - **minimum** of the largest $\approx n/2$ numbers.
- Keep a max heap for the smallest half.
- Keep a min heap for the largest half.
- May become unbalanced.
  - Move elements between them to balance.

Example

- Given 5, 1, 9, 8, 10, 7, 3, 6, 2, 4

Analysis

- Given a stream of n numbers, compute median, insert another, compute median

quickselect (dyn. arr.)

- $\Theta(n)$ time for n appends
- $\Theta(n)$ time for quickselect
- $\Theta(1)$ time for 1 append
- $\Theta(n)$ time for quickselect

now (double heap)

- $\Theta(n \log n)$ time for n inserts
- $\Theta(1)$ time for median
- $\Theta(\log n)$ time for 1 insert
- $\Theta(1)$ time for quickselect