CSCM12: Efficency software and solving problems
Trees

Cécilia Pradic

March 9th (and probably 16th tbqh) 2026

Now

Introduce tree-like datastructure

- What are they?
- How to encode them in python?
- Motivating examples & applications

Now

Introduce tree-like datastructure

- What are they?
- How to encode them in python?
- Motivating examples & applications

Also an opportunity to **recap material on sorting algorithms** with heapsort.
(c.f. challenge task of last lab)

Motivations

What do you mean by a tree-like datastructure?

Recursive definition of a list

A list of As is either

- An empty list
- Or an A and (a reference to) another list of As

What do you mean by a tree-like datastructure?

Recursive definition of a list

A list of As is either

- An empty list
- Or an A and (a reference to) another list of As ← **one** subobject of the same kind

What do you mean by a tree-like datastructure?

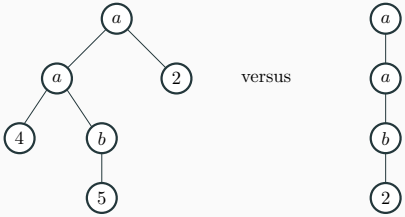
Recursive definition of a list

A list of As is either

- An empty list
- Or an A and (a reference to) another list of As ← **one** subobject of the same kind

For the most part

Recursive datatypes with possibly **multiple** subobjects of the same kind



What do you mean by a tree-like datastructure?

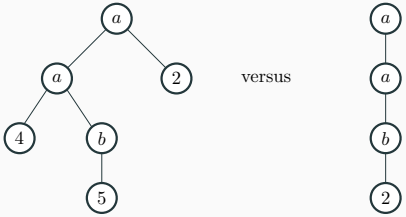
Recursive definition of a list

A list of As is either

- An empty list
- Or an A and (a reference to) another list of As ← **one** subobject of the same kind

For the most part

Recursive datatypes with possibly **multiple** subobjects of the same kind

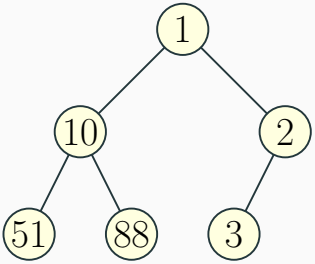


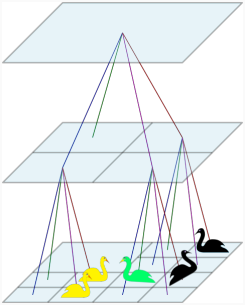
(list-like datatypes are degenerate tree-like datatypes)

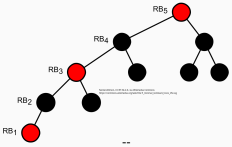
Why should we care? (1/2)

Tree-like structures come up in a variety of contexts:

- Efficient datastructures: sets/priority queues with $\mathcal{O}(\log(n))$ operations, random access lists, quad/octtrees...





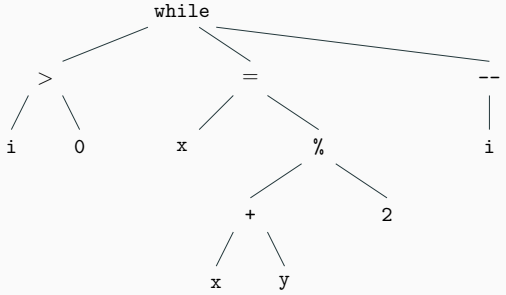


Why should we care? (2/2)

Trees can also come up as natual objects we'd like to manipulate

- E.g., anything hierarchical, abstract syntax trees, directory trees

```
while(i > 0) {
  x = x + y % 2;
  i--;
}
```



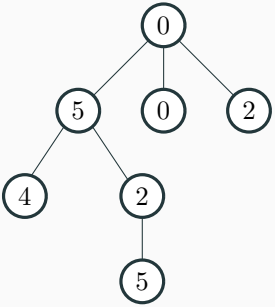
Generalities

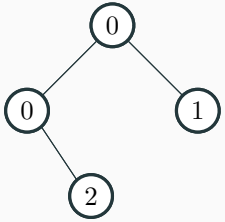
Recursive mathsy definition of a tree

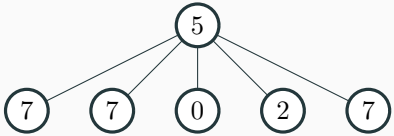
Formal definition

A tree with labels in L is a pair $(\text{label}, \langle c_1, \ldots, c_n \rangle)$ where:

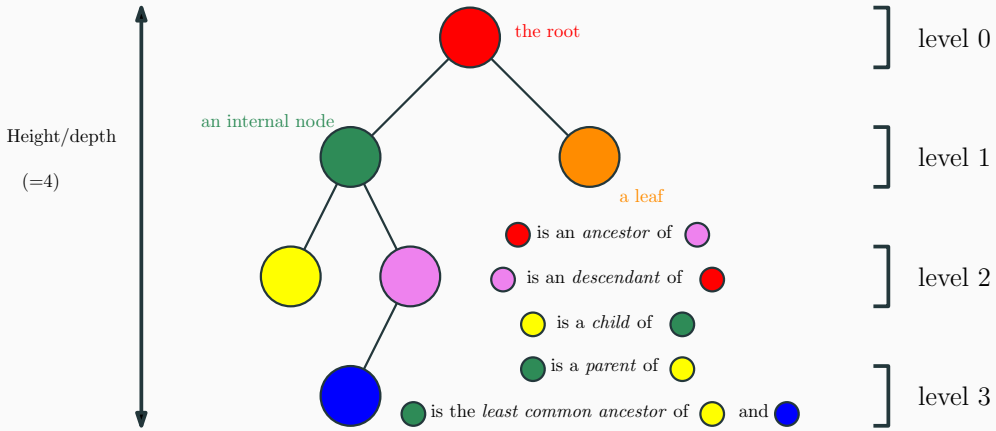
- label $\in L$
- $\langle c_1, \ldots, c_n \rangle$ is a list of trees with labels in L (possibly an empty list)







Vocabulary/basic notions



$\text{depth} \leq \text{size} \leq \max(\textit{arity})^{\text{depth}}$

- **breadth-first enumeration:**
- depth-first prefix enumeration:
- depth-first postfix enumeration:
- depth-first infix enumeration:

(← only makes sense for *binary* trees)

In java

- Typically encoded via a recursive class

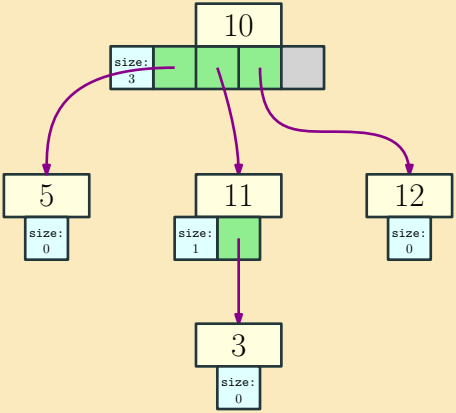
```
class tree:
  def __init__(self, x, c = []):
    self.label = x
    self.children = []
    for i in c:
      self.children.append(i)
```

```
root = tree(9, [tree(8)])
root.children.append(tree(1))
root.children.append(tree(2))
someNode = root.children[1]
someNode.children.append(tree(6))
someNode.children[0].children.append(
```

Tree example with memory representation

```
root = tree(10, [5, 11, 12])
root.children[1].children.append(leaf(3))
```

A somewhat honest of the representation in memory

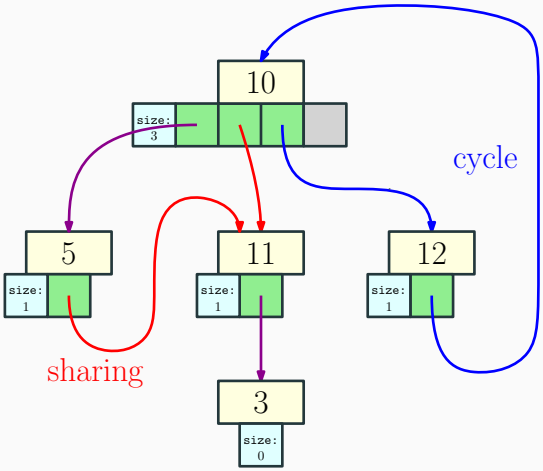


Non-trees?

- With linked lists, possible to create **cycles**

(and worse pathologies in the case of doubly-linked lists)

- Same here + an additional pitfall: **sharing**



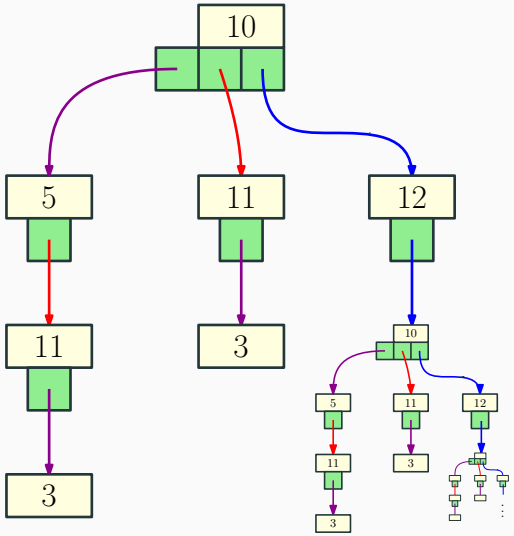
Are those illegitimate?

Not necessarily:

- They can be seen as **graphs**

  (Topic of next lecture)

- Can represent (potentially infinite) trees



A time and place for sharing and cycles

Cons

- **Cycles**: no longer a finite well-defined notion of **depth**

  ⇒ a lot of tree algorithm no longer terminates like e.g. traversal

- **Sharing**: a single update modifies **several spots** in the unravelling

  (⇒ not an issue for **immutable** datastructures)

A time and place for sharing and cycles

Cons

- **Cycles**: no longer a finite well-defined notion of **depth**

  $\Rightarrow$ a lot of tree algorithm no longer terminates like e.g. traversal

- **Sharing**: a single update modifies **several spots** in the unravelling

  ($\Rightarrow$ not an issue for **immutable** datastructures)

Pros

- **Cycles**: can represent infinite trees in finite space

  only regular trees, which may arguably admit more convenient representations

- **Sharing**: saves memory/can represent *directed acyclic graphs* (DAGs)

A time and place for sharing and cycles

Cons

- **Cycles**: no longer a finite well-defined notion of **depth**

  ⇒ a lot of tree algorithm no longer terminates like e.g. traversal

- **Sharing**: a single update modifies **several spots** in the unravelling

  (⇒ not an issue for **immutable** datastructures)

Pros

- **Cycles**: can represent infinite trees in finite space

  only regular trees, which may arguably admit more convenient representations

- **Sharing**: saves memory/can represent *directed acyclic graphs* (DAGs)

- Commonplace tacit assumption: No sharing/cycles in tree-like datastructure
- One just has to be extra clear about what they consider legal inputs/outputs

A time and place for sharing and cycles

Cons

- **Cycles**: no longer a finite well-defined notion of **depth**

  $\Rightarrow$ a lot of tree algorithm no longer terminates like e.g. traversal

- **Sharing**: a single update modifies **several spots** in the unravelling

  ($\Rightarrow$ not an issue for **immutable** datastructures)

Pros

- **Cycles**: can represent infinite trees in finite space

  only regular trees, which may arguably admit more convenient representations

- **Sharing**: saves memory/can represent *directed acyclic graphs* (DAGs)

- Commonplace tacit assumption: No sharing/cycles in tree-like datastructure
- One just has to be extra clear about what they consider legal inputs/outputs
- **For this lecture:** no more sharing/cycles

Tree-like datastructures

Often, you may want more/less flexibility than the generic tree datastructure

- Do you want to bound the arity of internal nodes?
- Do you care about the ordering of children?
- Do you care about empty spots for future children?
- Do you want more labels?
- Do you want different type of labels for e.g. leaves?
- ...

```
class ArithExpr:
  def __init__(self, s, l = None, r = None):
    self.op = s
    self.left = l
    self.right = r
```

→ for most situations, similar issues/resolutions

One last restriction for today

Binary trees are those trees whose nodes have at most two children.

```
class BTree:
  def __init__(self, label, l = None, r = None):
    self.label = label
    self.leftChild = l
    self.rightChild = r
```

Conventions:

- `leftChild` and `rightChild` may be set to `None`

  (for a leaf: both are `None`)

- it is possible that `leftChild = null` and `rightChild != null`

  (we care about the order and "empty spots")

Binary search trees

Motivation: set with $\mathcal{O}(\log(n))$ lookup and delete

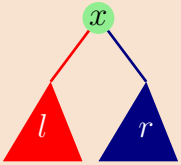
```
Set(); // creates an empty set
void remove(T e); // removes one element
boolean contains(T e); // do I contain the element?
void add(T e); // add one element
Set union(Set s2); // adds all elements of s2
...
```

Op \Data	`Array`	`List`	`ArrayList`	`TreeSet`
`Set(T)`	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`remove`	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(\log(n))$
`contains`	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(\log(n))$
`add`	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$ $\mathcal{O}(1)$ amortized	$\mathcal{O}(\log(n))$
`union`	$\mathcal{O}(n+m)$	$\mathcal{O}(1)$	$\mathcal{O}(n+m)$ $\mathcal{O}(m)$ amortized	$\mathcal{O}(m\log(n))$

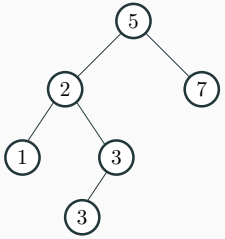
A datastructure to represent set of numbers

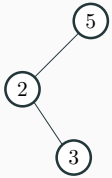
Definition

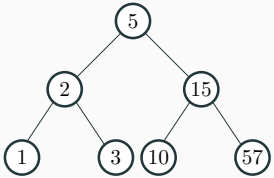
A **Binary Search Tree** is a binary tree labeled by integers such that



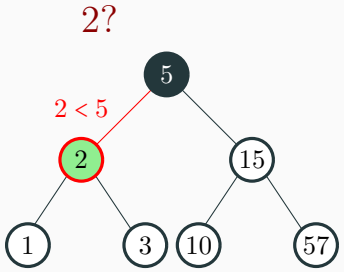
$$\Rightarrow \qquad l \leq x \leq r$$

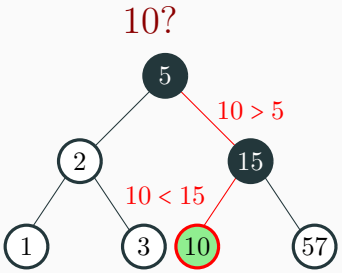


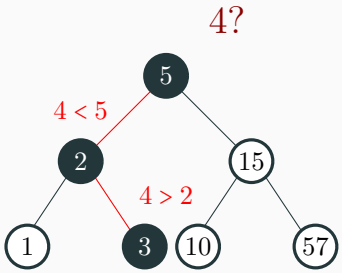




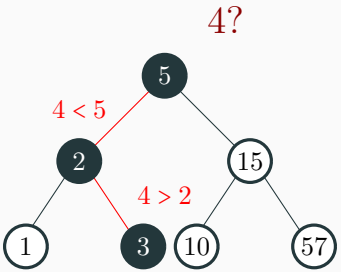
Look up an element `boolean contains(int e)`



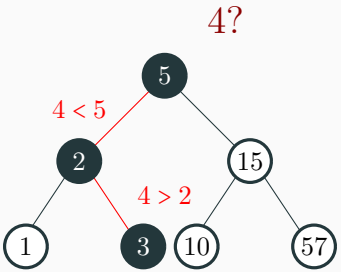




Complexity of `boolean contains(int e)`?

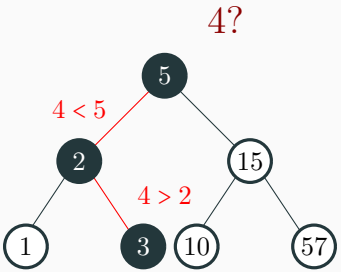


Complexity of `boolean contains(int e)`?



$\rightarrow \mathcal{O}(\text{depth})$

Complexity of `boolean contains(int e)`?



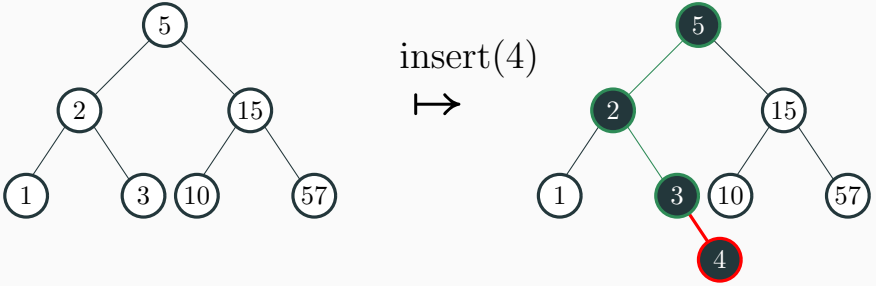
$\rightarrow \mathcal{O}(\text{depth})$

Relation to the size of the set

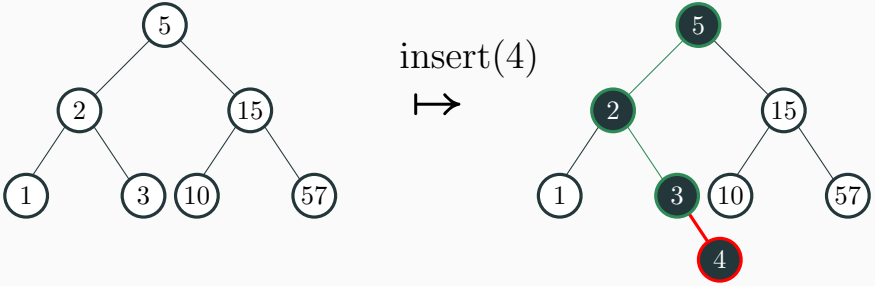
- Best case: the tree is balanced $\rightarrow$ depth $= \mathcal{O}(\log(\text{size}))$
- Worst case: one child everywhere $\rightarrow$ depth $= \Omega(\text{size})$

$\rightarrow$ **Important concern:** work on **balanced trees**

Insert an element `void add(int e)`



Insert an element `void add(int e)`

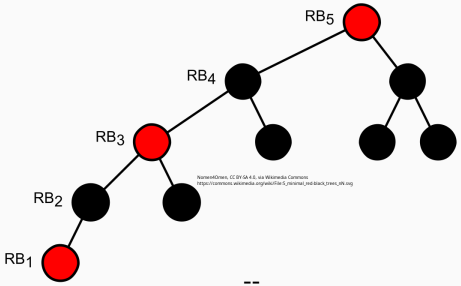


- Complexity: still $\mathcal{O}(\text{depth})$
- **Issue:** repeatedly inserting bigger and bigger elements can unbalance a tree

Try inserting $1, 2, 3, \ldots$ to `Leaf(0)`

Solutions (not covered in-depth here)

- Either try to do some probabilistic analysis and try to prove things are not that bad on average for a given use-case ...
- ... or use fancier invariants to have classes of trees with depth $= \mathcal{O}(\log(\text{size}))$
- Paradigmatic examples: red-black trees and AVLs



--

- Involved "repair" procedures to maintain the invariants after an insertion/deletion running in $\mathcal{O}(\text{depth})$
- Something like this is implemented for `TreeSet`

So now, you should be able to tell why this table is like this

Op \Data	`Array`	`List`	`ArrayList`	`TreeSet`
`Set(T)`	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`remove`	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(\log(n))$
`contains`	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(\log(n))$
`add`	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$ $\mathcal{O}(1)$ amortized	$\mathcal{O}(\log(n))$
`union`	$\mathcal{O}(n+m)$	$\mathcal{O}(1)$	$\mathcal{O}(n+m)$ $\mathcal{O}(m)$ amortized	$\mathcal{O}(m\log(n))$

Priority queues, heaps and heapsort

Quick note

Motivation

Implement a priority queue with $\mathcal{O}(\log(n))$ operations

$+ \rightarrow$ a new in-place sorting algorithm in $\mathcal{O}(n \log(n))$

Ordines prioritatis memento

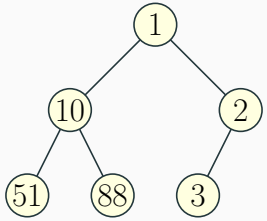
```
class priorityQueue:
  def __init__(self):
    # empty queue
  def enqueue(self, x, k):
    # enqueue x with priority k
  def dequeue(self):
    # remove and return an element with minimal priority
  def min(self):
    # return the pair (x, k) of an element x with priority k
    # that would be dequeued next, or None if empty
```

What's a heap?

Definition

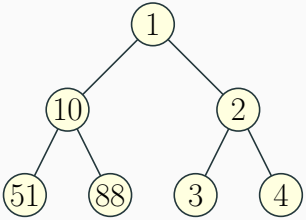
A min-heap is a binary tree such that

- The label of every node is smaller than its children's
- All of its levels are full, except possibly the last
- The last level is completely filled left-to-right

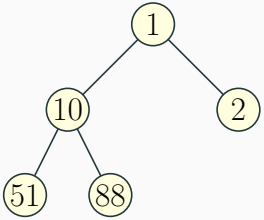


(for priority queues: numbers are priorities + extra label type `T` in nodes)

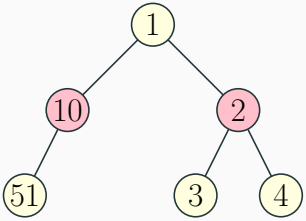
Examples/counter-examples



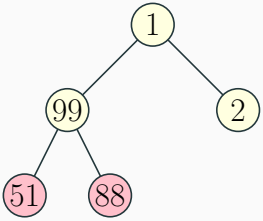
↑ valid heap



↑ valid heap

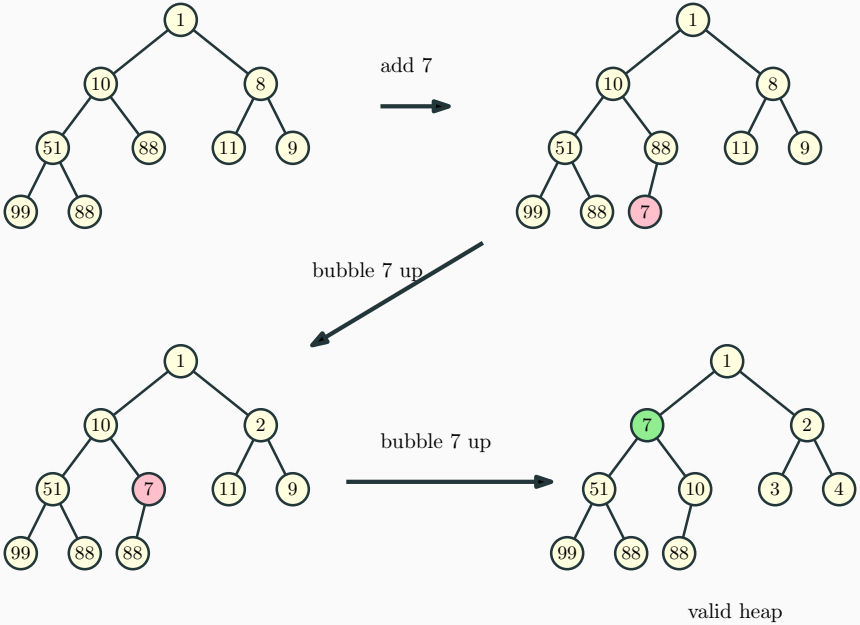


↑ not a heap (unbalanced)

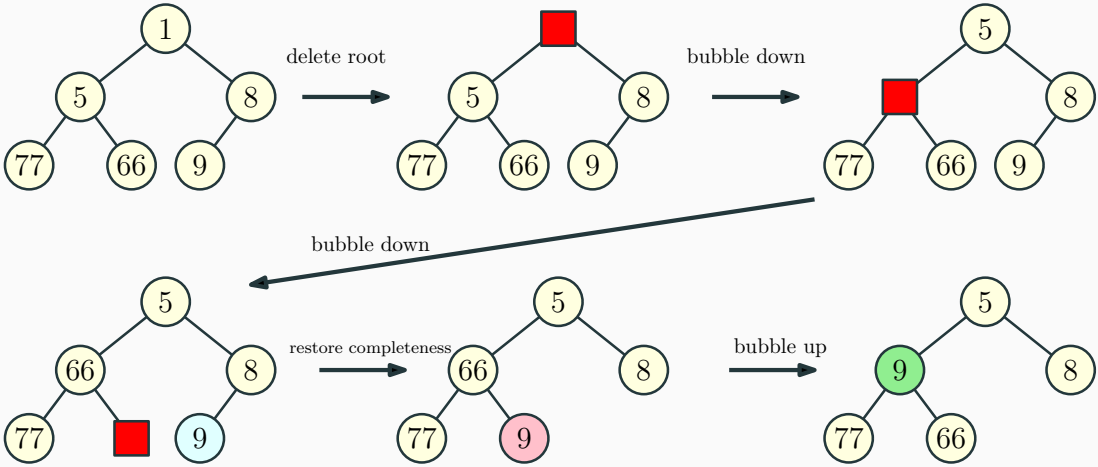


↑ not a heap ($99 > 88$)

Inserting a new element and repairing in $\mathcal{O}(\log(n))$

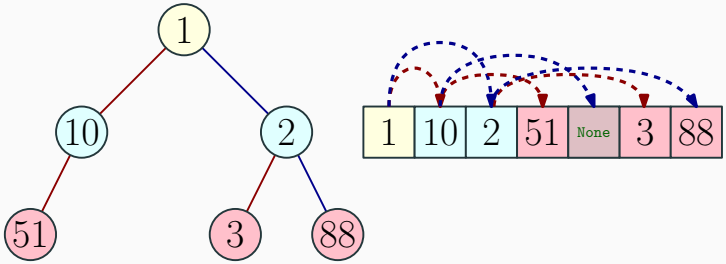


Deleting the root and repairing in $\mathcal{O}(\log(n))$



Representing trees as arrays

While the shape of a tree is good ot keep in mind, when they are of bounded arity and close to complete, it might be better to represent them as arrays



- Fast access due to $\mathcal{O}(1)$ lookup in arrays
- Downsides: *potentially* **wasting memory** and bounding a priori arities

(absent nodes = cells filled with `None`)

For heaps: that's a good representation!

Heap sort

The algorithm

- start with an empty heap
- insert all the elements in the collection you want sorted

$$\sum_{i=1}^{n} K \log(i) + K' = \mathcal{O}(n \log(n))$$

- insert the value of the root at the back of your output and delete the root

$$\sum_{i=1}^{n} K'' \log(i) + K''' = \mathcal{O}(n \log(n))$$

Heap sort

The algorithm

- start with an empty heap
- insert all the elements in the collection you want sorted

$$\sum_{i=1}^{n} K \log(i) + K' = \mathcal{O}(n \log(n))$$

- insert the value of the root at the back of your output and delete the root

$$\sum_{i=1}^{n} K'' \log(i) + K''' = \mathcal{O}(n \log(n))$$

- Optimal asymptotic complexity for a comparison-based sort!
- Can be done *in-place* in an array wiht minor adjustement

$\mathcal{O}(n)$ space complexity

Quick recap on sorting algorithms

Bubble sort, insertion sort

- $\mathcal{O}(n^2)$
- In place for arrays and linked lists

Merge sort

- $\mathcal{O}(n \log(n))$, good for parallelization
- Not in-place for arrays, but in place for linked lists
- A *stable* sort (does not disturb elements that are "equal")
- Python's `sort`: an elaboration on merge sort.

Today

Heap sort

- $\mathcal{O}(n \log(n))$
- In-place for arrays! (not linked lists)

If you want to go further

Quick sort

- Idea: pick some element (a pivot), put apart elements below and above it, sort them recursively, and put things back together.
- $\mathcal{O}(n^2)$, $\mathcal{O}(n \log(n))$ on average with randomized pivot
- Easily done in-place for both arrays and linked lists
- $\mathcal{O}(n \log(n))$ with a smart pivot, but this is complicated
- Java's `sort` for arrays: quicksort, but with a three-way division in the recursion.

CountSort

- *Not a comparison-based sort*, so can run in linear time **if working with numbers in a restricted range**.
- Idea: list the range of your collection, count the occurrences, and then reconstruct your collection.

That's all for trees

See you in the lab to practice working with trees!

Next we shall discuss **hashing**.