Trees

Tree - A connected graph without cycles
Parent - The node above a node
Child - The node below a node
Leaf - Child-less nodes

0.1 Using trees to store data

In a binary tree, each node has:

- Pointer to parent (or NIL, or to itself, if root)
- Pointer to left child (or NIL, or to itself, if there isn't one)
- Pointer to right child (or NIL, or to itself, if there isn't one)
- Payload

Binary trees are useful to store data, giving fast insert, lookup and delete operations.

1 BST

A binary search tree (BST) is a tree in which no node has more than two children

You must build and maintain the tree such that it's true for every node v of the tree that

- all elements in its left sub tree are ”smaller” than v
- all elements in its right sub tree are ”bigger” than v

Smaller and bigger refer to the payload. The left/right sub-tree refers to the tree rooted in a node's left/right child

1.1 Traversals

Starting with root:

- In order: Recurse into left sub-tree, print payload, recurse into right sub-tree
- Pre-order: Print payload, recurse into subtrees
- Post-order: recurse into sub-trees, print payload

1.2 Code for a BST

Initialising a node:

```
class Node:
    """
    Tree node: left and right child + data which can be any object
    """
    def __init__(self, data):
    """
    Node constructor
    @param data node data object
    """
    self.left = None
    self.right = None
    self.data = data
```

Inserting a node:

```
class Node:
    ...
    def insert(self, data):
        """
        Insert new node with data
        @param data node data object to insert
        """
        if data < self.data:
            if self.left is None:
                self.left = Node(data)
            else:
                self.left.insert(data)
        else:
            if self.right is None:
                self.right = Node(data)
            else:
                self.right.insert(data)
```

Looking up a value:

```
class Node:
    ...
    def lookup(self, data, parent=None):
    """
    Lookup node containing data
    param data node data object to look up
    param parent node's parent
    returns node and node's parent if found or None, None
    """
    if data < self.data:
        if self.left is None:
            return None, None
        return self.left.lookup(data, self)
    elif data > self.data:
        if self.right is None:
            return None, None
        return self.right.lookup(data, self)
    else:
        return self, parent
```

1.2.1 Deleting

Deleting is the most difficult operation on a BST.

We first do a lookup on the element that we wish to remove. If that returns "not found" then we're done.

Otherwise, three cases need to be considered:

- If a node is a leaf then simply remove it
- If a node has one child then remove and replace it with that one child - list the sub tree up
- If a node has two children

Counting children:

```
class Node:
    ...
    def children_count(self):
        """
        Returns the number of children
```

```
        @returns number of children: 0, 1, 2
        """
        if node is None:
            return None
        cnt = 0
        if self.left:
            cnt += 1
        if self.right:
            cnt += 1
        return cnt
```

Deleting a node

```
class Node:
    ...
    def delete(self, data):
    """
    Delete node containing data
    @param data node's content to delete
    """
    # get node containing data
    node, parent = self.lookup(data)
    if node is not None:
        children_count = node.children_count()

    if children_count == 0:
        # if node has no children, just remove it
        if parent.left is node:
            parent.left = None
        else:
            parent.right = None
        del node

    elif children_count == 1:
        # if node has 1 child
        # replace node by its child
        if node.left:
            n = node.left
        else:
            n = node.right
        if parent:
            if parent.left is node:
                parent.left = n
            else:
                parent.right = n
        del node

    else:
        # if node has 2 children
        # find its successor
        parent = node
        successor = node.right
        while successor.left:
            parent = successor
            successor = successor.left
        # replace node data by its successor data
        node.data = successor.data
        # fix successor's parent's child
        if parent.left == successor:
```

```
            parent.left = successor.right
        else:
            parent.right = successor.right
```

2 RedBlack trees

BST property: for all nodes v with key k:

- nodes in left subtree has keys $< k$
- nodes in right subtree have keys $> k$

Have seen simple BST operations:

- Lookup
- Minimum, maximum
- Predecessor, successor

All take $O(h)$ time, h height of tree

This is OK if the BST is **balanced**, then $h = O(\log n)$
Bad if the BST is **degenerated**, then $h = \Omega(n)$

2.1 The idea

A red black tree is an ordinary BST with one extra bit of storage per node, it's colour: red or black

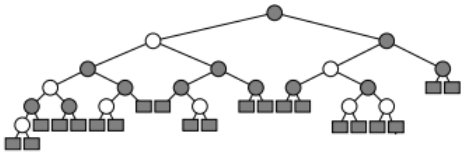
Constraining how nodes can be coloured results in (approximate) balancedness of tree

Assumption: if either parent, left, or right does not exist, pointer to NULL

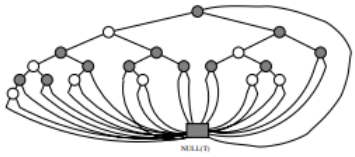
Regard NULLs as leaves of tree, thus normal nodes are internal.

A BST is a red-black tree if:

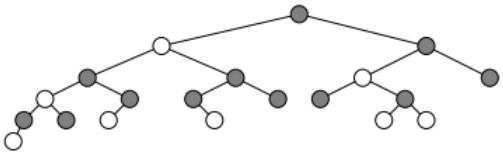
1. Every node is either red or black
2. the root is black
3. every leaf (NULL) is black
4. Red nodes have black children
5. for all nodes, all paths from node to descendant leaves contain the same number of black nodes'



red-black tree, according to definition



replace NULLs with single "sentinel" NULL(T)



normal drawing style (no NULLleaves or sentinel), but sentinel **is** there

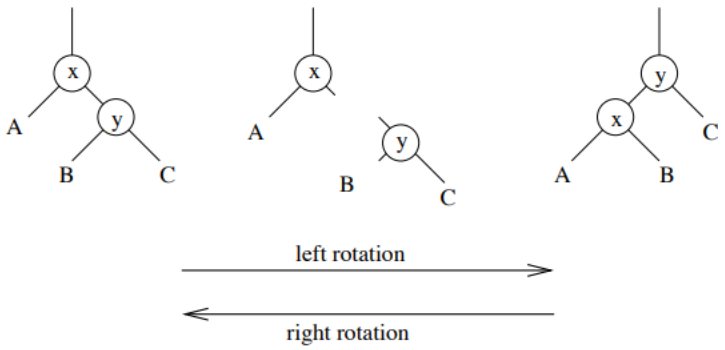
A redblack tree with n internal nodes has height at most $2\log(n+1)$
This means that all Lookup, min, max, successor, predecessor all have worst case $O(\log n)$ time

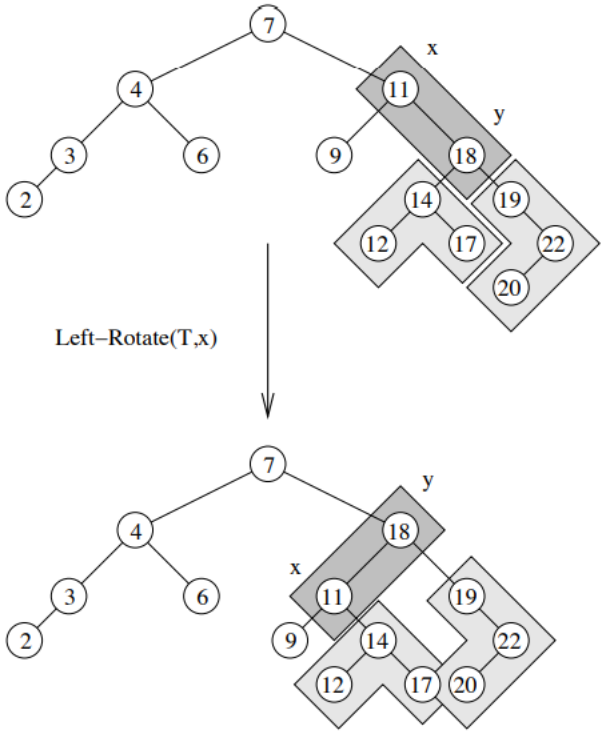
2.2 Rotations

In order to maintain the red-black property, rotations must be used.

There is the option of a left or right rotation, under the assumptions:

- left rotation on x: right child not NULL
- right rotation on x: left child not NULL





This can be done in $O(\log n)$ time

Suppose we insert a new node z

First, ordinary BST insertion of z according to key value (walk down from root, open new leaf)

The, colour z red

We may have violated the red-black property

Then call the fixup procedure to restore the red black property

3 Heaps

Heaps are also trees, but typically assumed to be stored in a flat array

- each tree node corresponds to an element of the array
- the tree is complete except perhaps the lowest level, filled left to right
- heap property: for all nodes v in the tree, `v.parent.data >= v.data`
- This is a max heap (for min heaps, `v.parent.data =< v.data`)

Heap represented as an array A has two attributes:

- Length(A) - The size of the array
- HeapSize(A) - The size of the heap stored within the array

Clearly, Length(A)≥HeapSize(A) at all times

3.1 Array indices

Assume we start counting at position 1, then:

- The root is in A[1]

- parent(i)=A[i/1] (integer division, rounds down)
- left(i)=A[2i]
- right(i)=A[2i+1]

3.2 Why use heaps?

Heaps are very good data structures for priority queues as sorting

```
HeapExtractMax(A)
ret = A[1] // biggest element (highest priority)
A[1] = A[HeapSize(A)]
HeapSize(A) = HeapSize(A)-1
Heapify(A,1,HeapSize(A))
return ret
```

3.3 Heapify

Heapify changes a tree into a heap

Idea:

- Starting at the root, identify largest of current node v and its children
- Suppose largest element is in w
- If $w \neq v$
  - Swap A[w] and A[v]
  - Recurse into w (contains now what root contained previously)

```
Heapify(A,v,n)
// n is heap size
// find largest among v and 2v (left child)
largest = v
if 2v <= n and A[2v]>A[v] then largest = 2v

// find largest among winner above and
// 2v+1 (right child)
if 2v+1 <= n and A[2v+1]>A[largest] then
    largest = 2v+1

if largest != v then
    swap A[v], A[largest]
    Heapify (A, largest, n)
```

3.4 BuildHeap

Task: given array A with n arbitrary numbers in it, convert A into a heap

```
BuildHeap(A,n)
for i=n downto n/2 do
Heapify(A,i,n)
endfor
```

3.4.1 Runtime

Loop with n iterations, each call to Heapify takes $O(\log n)$ time
This implies an overall bound of $O(n \log n)$
However big O gives an upper bound, so can we reduce this any further?

- If *height* of a node is counting upwards from lowest leaves with height of leaf on lowest level-1 and height(root)=log n then in n-element heap there are $\leqslant \frac{n}{2^h}$ nodes of height h
- time for Heapify when called on a node of height h is $O(h)$
- Cost:

$$T(n) = \sum_{h=1}^{\log n} (\#\text{ nodes at height } h) \cdot O(h) \leq \sum_{h=1}^{\log n} \frac{n}{2^h} \cdot O(h) = O\left(\sum_{h=1}^{\log n} \frac{n}{2^h} \cdot h\right) = O\left(n \cdot \sum_{h=1}^{\log n} \frac{h}{2^h}\right)$$

  It is well known that for $x \in (0,1)$,

$$\sum_{i=0}^{\infty} i \cdot x^i = \frac{x}{(1-x)^2}$$

  Use this (x=1/2)

$$\sum_{h=1}^{\log n} \frac{h}{2^h} \leq \sum_{h=0}^{\infty} h \cdot \left(\frac{1}{2}\right)^h = \frac{1/2}{(1-1/2)^2} = \frac{1/2}{1/4} = 2$$

  And hence:

$$T(n) = O\left(n \cdot \sum_{h=1}^{\log n} \frac{h}{2^h}\right) = O(n)$$

  That is, can turn any array into heap in time $O(n)$

3.5 HeapSort

The method for this is:

- Call BuildHeap on unsorted data, and
- repeatedly call HeapExtractMin until empty

This has running time: $O(n) + n \cdot O(\log n) = O(n \log n)$

```
HeapSort(A)
    BuildHeap(A, Length(A))
    for i = Length(A) downto 2 do
        swap A[i] and A[1]
        HeapSize(A) = HeapSize(A)-1
        Heapify(A, 1, HeapSize(A))
    endfor
```

4 Lower bounds

4.1 Sorting and decision trees

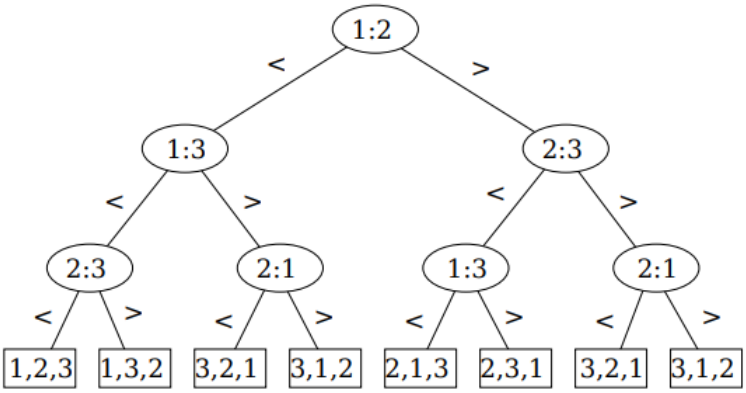
4.1.1 Theorem

Any comparison based sorting algorithm requires $\Omega(n \log n)$

4.1.2 Decision trees

A **decision tree** is a **full binary tree** (every node has either 0 or 2 children)
It represents comparisons between elements performed by a particular algorithm run on a particular size of input
Only **comparisons** are relevant, everything else is ignored
Internal nodes are labelled i:j for $1 \leqslant i, j \leqslant n$, meaning elements i and j are compared (indicies w.r.t original order)
Downward edges are labelled $\leqslant$ or $>$, depending on the outcome of the comparisons
Leaves are labelled with some permutation of $\{1, ..., n\}$
A **branch** from root to leaf describes sequence of comparisons (nodes and edges) and ends in some resulting sequence (leaf)
We have at least n! leaves - at least one for each outcome

4.1.3 Selection sort



An correct sorting algorithm must be able to produce each permutation of input (or: must sort any permutation of e.g. $\{1, ..., n\}$)
A necessary condition is that each of the n! permutations must appear as leaf of decision tree
Lower bound for worst case
Length of longest path from root of decision tree to any leaf (height) represents worst case number of comparisons for given value of n.
For given n, lower bound on heights of all decision trees where each permutation appears as leaf is thus lower bound on running time of **any comparison based sorting algorithm**

4.1.4 Theorem

Any comparison based sorting algorithm requires $\Omega(n \log n)$

4.1.5 Proof

- Sufficient to determine minimum height of a decision tree in which each permutation appears as a leaf
- Consider decision tree of height h with ℓ leaves corresponding to a comparison sort on n elements
- Each of the n! permutations of the input appears as some leaf: $\ell \geqslant n!$
- Binary tree of height h has at most 2^h leaves: $\ell \leqslant 2^h$
- Together $n! \leqslant \ell \leqslant 2^h$ and therefore $2^h \geqslant n!$
- Take logs: $h \geqslant \log(n!) = \Omega(n \log n)$

4.2 Selection and adversaries

4.2.1 Lower bound via adversaries

Adversary:

- Is second algorithm intercepting access to input
- Gives answers so that there's always a consistent input
- Tries to make original algorithm delay a decision by dynamically constructing a bad input for it
- doesn't know what original algorithm will do in the future, must work for **any original algorithm**

To get a good lower bound, design a good adversary

4.2.2 Finding max

Goal: find max element in unsorted array
In an array of size n, can do this with n-1 comparisons
Same set-up as before: only comparisons are relevant

Theorem
Any comparison based algorithm for this problem requires at least n-1 comparisons in the worst case

Proof
After $\leqslant n-2$ comparisons, $\geqslant 2$ elements never lost (a comp)

- Adversary can make any of them max and be consistent
- Not enough info for algorithm to make a decision

Hence algorithm needs to make at least n-1 comparisons

4.2.3 Representing Adversary's strategy

1. As a digraph:
   - the elements of array are the nodes
   - if i loses to j, draw edge $i \to j$
   - The adversary is consistent, so there cannot be any cycles, as that would lead to a contradiction
   - The algorithm can stop when there is an element which can be reached from any other element following the path of directed edges, then you know the element is maximum. The smallest number of edges to make the graph connected is n-1
2. As a status update table with bits of info revealed:
   Each index i has status NL (never lost) or L (lost)
   When comparing i and j:

Status for i, j	Adv Reply	newStatus	Info
N;N	$a_i > a_j$	N;L	1
N;L	$a_i > a_j$	No Change	0
L,N	$a_i < a_j$	No Change	0
L;L	Consist	No Change	0

All the elements but one must lose at least once

4.2.4 Finding 2nd largest

Goal: find the 2nd largest element in an unsorted array

Theorem
Any comparison based algorithm for this problem requires at least $n + \lceil \log_2 n \rceil - 2$ comparisons in the worst case

Observations

1. Any correct algorithm must also determine the max. $n-1$ elements most lose $\geqslant 1$ comparisons
2. The 2nd largest must lose to max directly (there is no other element that it could lose to)
3. If max had direct comparisons with m elements, then m-1 of these elements must lose $\geqslant 2$ comparisons
4. Hence, at least $n-1+m-1 = n+m-2$ comparisons are required

4.2.5 Adversary's strategy

Lemma Adversary can force $\lceil \log_2 n \rceil$ comparisons involving max

Proof
Adversary assigns weight w_i to each input element a_i
Initially all $w_i = 1$, then update using these rules

Weights	Adv Reply	Update
$w_i > w_j$	$a_i > a_j$	$w_i = w_i + w_j, w_j = 0$
$w_i < w_j$	$a_i < a_j$	$w_j = w_i + w_j, w_i = 0$
$w_i = w_j > 0$	$a_i > a_j$	$w_i = w_i + w_j, w_j = 0$
$w_i = w_j = 0$	Consistent	No change

If the weights are equal and positive then the adversary makes a random decision

4.2.6 Analysis

- $w_i = 0 \Leftrightarrow$ i lost $\geqslant 1$ comparison
- Weight of an element at most doubles after a comparison (because you are shifting the smaller weight to the larger one)
- If max is involved in m comparisons, its weight is $\leqslant 2^m$
- In the end, max accumulates all the weight, so $n \leqslant 2^m$ (all elements feeding into max)
- Taking logs, $\log_2 n \leqslant m$, as required

To conclude, Adversary can force $n + \lceil \log_2 n \rceil - 2$ comparisons

4.2.7 Finding 2nd largest: Algorithm

Here's an algorithm that matches out lower bound.
Consider a knock out tournament: players= array elements
Think a balanced tree, leaves-array elements. With array of size n, the height of the tree is $\lceil \log_2 n \rceil$
Play the tournament to find max: this takes n-1 comparisons.
Consider the $\lceil \log_2 n \rceil$ elements who lost directly to max.
2nd largest overall=largest among those.
Find it with $\lceil \log_2 n \rceil - 1$ comparisons.
Altogether, this used $n + \lceil \log_2 n \rceil - 2$ comparisons

4.2.8 Finding kth largest

Let C_k be the number of comparisons necessary and sufficient to find the kth largest element in an array of size n
The exact value of C_k is known for

$$k = 1, 2, n-1, n$$

For the other values of k, this is open.
Though there are known bounds on C_k