CS 5200 Spring 2018 Analysis of Algorithms-Homework6

Xiongming Dai

April 28,2018

1 Red-black tree-insert

Inserting the keys 41,38,31,12,19,8, we can find the results shown as following figure 1 to 6.

2 Red-black tree-delete

Successive deletion of the keys in the order 8,12,19,31,38,41, which is shown in the figure 7-12.

3 Recursive algorithm

The proposed method that is not recursive can be used to solve this problem

```
##the common algorithm
def dynamic_fib(n)
  i=1
  j=1
  if n<=1 then
     return 1
  end if
  for i=2 upto n d
     tmp=i+j
     j=i
     i=tmp
  end for
return i
```

For a recursive algorithm Matrix-chain-multiply(A,S,i,j) that performs the optimal matrix-chain multiplication. we can do as the follows:

```
Matrix_chain_multiply(A,s,i,j)
if i==j then
   Return Ai
  end if
```

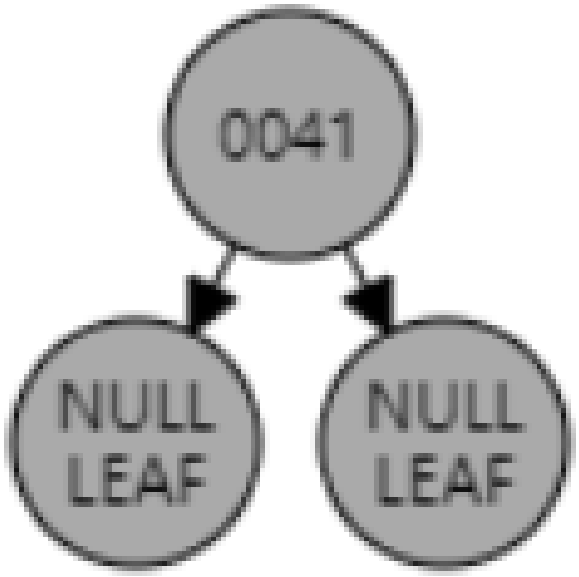

Figure 1: after inserting 41

```
return Matrix_chain_multiply(A,s,i,s[i][j])
          *matrix_chain_multiply(A,s,s[i][[j]+1,j)
```

The sample written with python is shown as follows:

```
p = []
i = 0
j = 0
temp = 0
k = 0
pas = 2
A = 0
def disp(l):
        for i in l:
                print i
        return
A= int(raw_input("Enter no. of Matrices "))
for i in range(0, A+1):
        p.append(int(raw_input()))
mink = [ [] for _ in range(A)]
m = [ [] for _ in range(A)]
#Step 0:
for i in range(0, A):
```

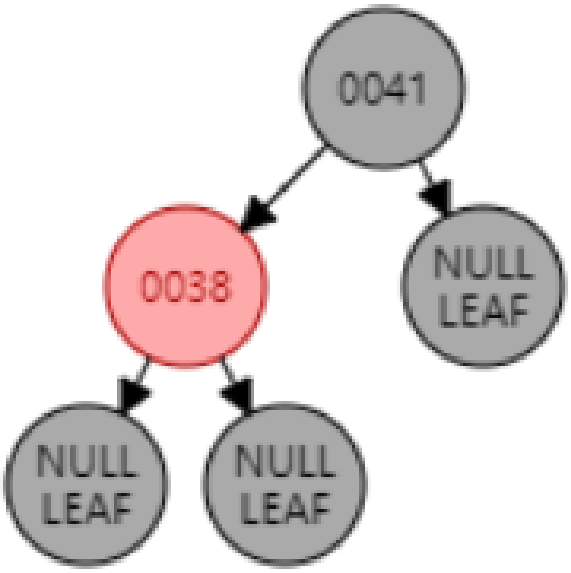

Figure 2: after inserting 38

```
        for j in range(i+1):
                m[i].append(0)
                mink[i].append(0)
print "m in step 0:\n", disp(m)
#Step 1:
for i in range(A-1):
      m[i].append(p[i]*p[i+1]*p[i+2])
print "m in step 1:\n", disp(m)
#Step 3:
for r in range(A-2, 0, -1):
        for i in range(0,r):
                minn = 99999
                j = i+pas
                for z in range(j-i):
                        #when k = i
                        k = i+z
                        temp = m[i][k] + m[k+1][j] + p[i]*p[k+1]*p[j+1]
                        if temp < minn:
                                minn = temp
                                mik = k + 1
                m[i].append(minn)
                mink[i].append(mik)
        pas = pas + 1
disp(m)
```

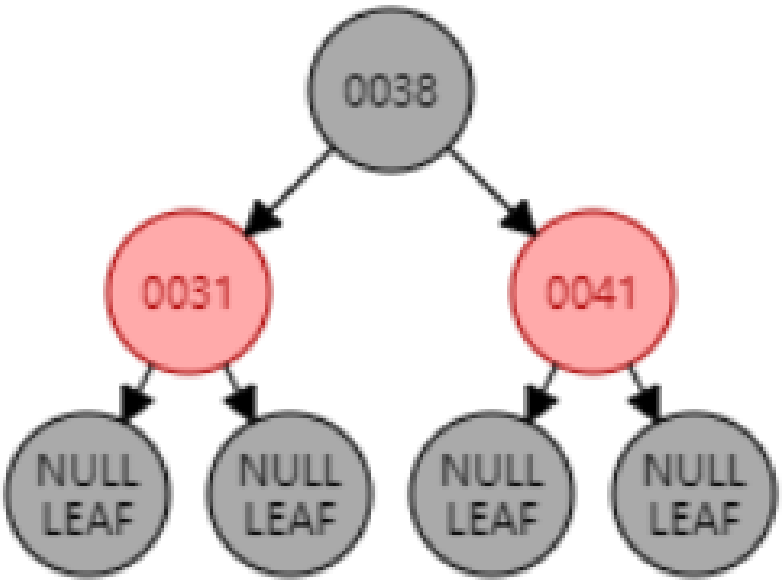

Figure 3: after inserting 31

```
disp(mink)
```

4 LCS

As we can see, we can find that the first list contain a ”00”,but the second contains none, from the second list, we also can find it contains two copies of ”11”, this is not inculded in the first list.For the sake of reconciling this, any LCS will have to skip at least three elements.Therefore, we know the common subsequence was maximal, from what have been discussed above, we can find easily an LCS :(1,0,1,0,1,0).

5 Longest monotinically increasing subsequence

For a given sequence S, we make a copy of S called S’, then we try to sort S’, we can run the LCS algorithm on these two sequence. The longest common subsequence must be monotone increasing because it is a subsequence of S’ which is sorted. It is also the longest monotone increasing subsequence ,since being a subsequence of S’ only adds the limitation of monotone increasing subsequence. As we can see $|S| = |S'| = n$, and sorting S can be done in $O(n^2)$ time, the final running time will be $O(|S||S'|) = O(n^2)$.

6 Least duration

(a) Let us consider a concrete case: suppose the activity times are $([2, 10], [7, 12], [11, 19])$, then, picking the shortest first, we must remove the other two, where if we picked the other two instead,

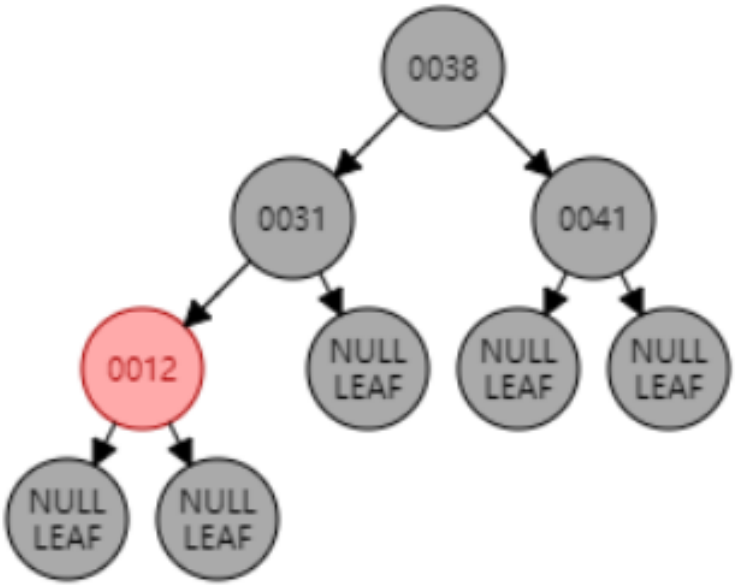

Figure 4: after inserting 12

before we have two tasks not just only one.
(b)The case that conflicts with the fewest remaining activities. Assume that the activity times are $([1,2],[3,6],[1,5],[1,5],[6,10],[9,15],[11,17],[11,17],[16,20])$. Then, by the method of greedy strategy, we would first pick $[6,10]$ since it only has a two confilicts. However, if we do like this, it means that we could not pick the only optimal solution of other times of the activities.
(c)The case that conflicts the optimality of greedily selecting the earlist start times, assume our activities times are $([1,8],[2,6],[7,8])$. If we pick the earliest start time, we will only have a single activity $[1,8]$, whereas the optimal solution would be to pick the other two activities.

7 Efficient algorithm

Let us consider the rightmost interval. It will be worse if it extends any further right than the rightmost point, however, it is well-known to us all that it must contain the rightmost point. Therefore,we know that it's right hand side is exactly the rightmost point. So, we just delete any point that is within a unit shortest distance of the right most point since they are restricted in this single interval. We repeat until all points are covered. Since at each step there is a clearly optimal choice for where to put the rightmost interval, so the final solution is optimal.Therefore, It is easy for us to find that for a given set of points on the real line, the smallest set of unit-length closed intervals that contains all of the given points $x_1, x_2, ..., x_n$, which have been proved.

8 Optimal Huffman code

The detail step for huffman code is shown as the following figure 12 and 13.

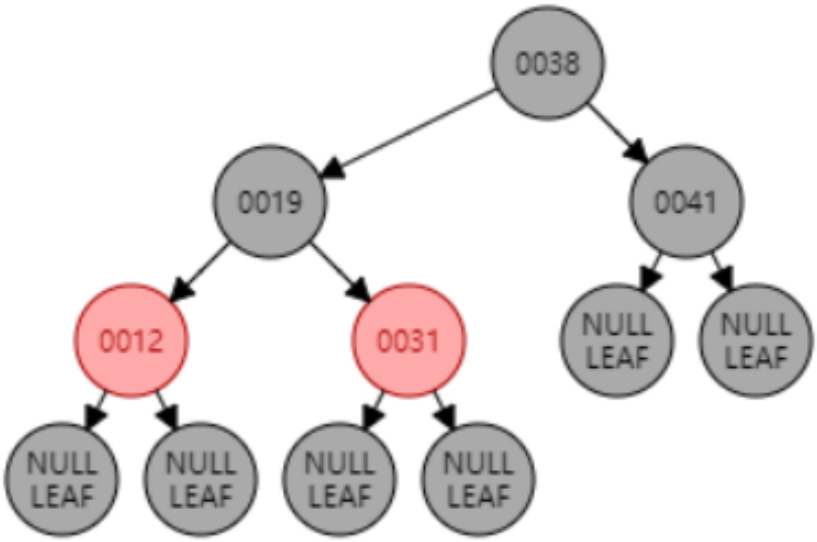

Figure 5: after inserting 19

9 Amortized cost

The detailed step for the demonstration is shown as following figure 14.

10 Empty B-tree

Inserting the keys in order into an empty B-tree with minimum degree 2, the detailed steps for 21 elements are shown in the following figure 15.

11 Tree delete

The results of deleting C,P, and V, in order, from the tree are shown in the following figure 16.

12 Fibonacci heap

Initially, we can consider from the subtrees rooted at 24,17 and 23, respectively, and then we add them to the root list. We set H.min to 18, and run consolidate. It has its degree 2 set to the subtree rooted at 18. Then the degree 1 is the subtree rooted at 38. Secondly, we get a repeated subtree of degree 2 when we consider the one rooted at 24. So, we make it a subheap through locating the 24 node under 18. Then, we consider the heap rooted at 17. This is a repeat for heaps of degree 1, so we place the heap rooted at 38 below 17. Finally we consider the heap rooted at 23, and then we have that all the different heaps have distinct degrees and are finished, setting H.min to the smallest, that is ,the one rooted at 17.

These heaps that we finish in our root list are shown as following figure 17.

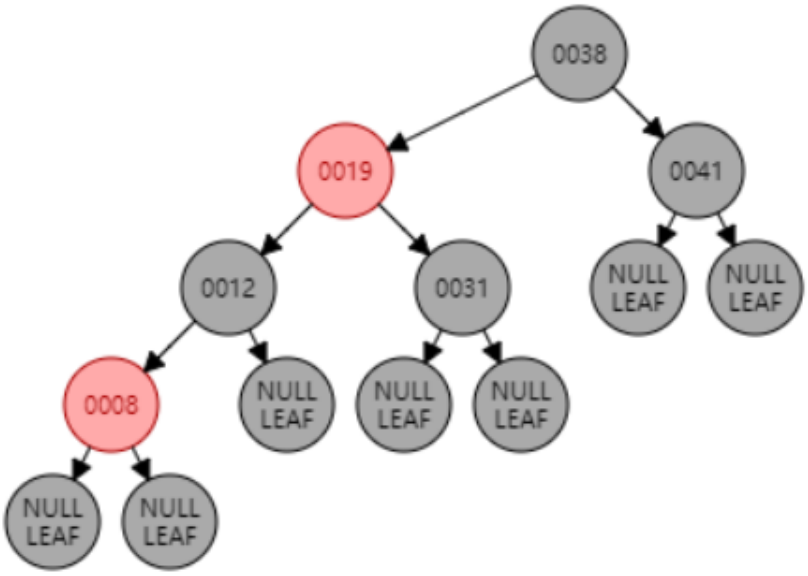

Figure 6: after inserting 8

13 Fibonacci heap creation

Firstly, we can do some operations for the Fibonacci-heap to demonstate the mistaken postitution. We try to add some nodes into the tree and then delete one node, if we pick 3, which will lead to a chain of length 1 obtained. Then add three nodes, all with the smaller values that the first three, and delete one of them. Then delete the leaf that is only at depth 1. After that, we can obtain a chain of length 2. Therefore, make a chain of length two using this process except with all smaller keys. Then, upon a consolidate being forced, we will have that the remaining heap will have one path of length 3 and one of length 2, with a root that is unmarked. So, just run decrease key on all of the children along the shorter path, starting with those of shorter depth. Then, extract min the appropriate number of times. Then what is left over will be just a path of length 3. We can continue this step. It will result in a chain of arbitrarily long length where all but the leaf is marked. It will take time exponential in n.

Secondly, we can make the procedure $linear(n, b)$ that makes heap that is a linear chain of n nodes and has all of its keys between b and $b + 2^n$. Also, as a precondition of running $linear(n, b)$, we have all the keys currently in the heap are less than b. As a base case, define $linear(1, b)$ to be the command $insert(b)$.

Assume we define $linear(1, b)$ to be the command $insert(b)$. Define $linear(n+1, b)$ as follows, where the return value list of nodes that lie on the chain but aren't the root:

```
S1=linear(n,b)
S2=linear(n,b+2^n)
x.key=-infinity
insert(x)
extractmin()
for each entry in S1, delete that key
The heap now has the desired structure, return S2
```

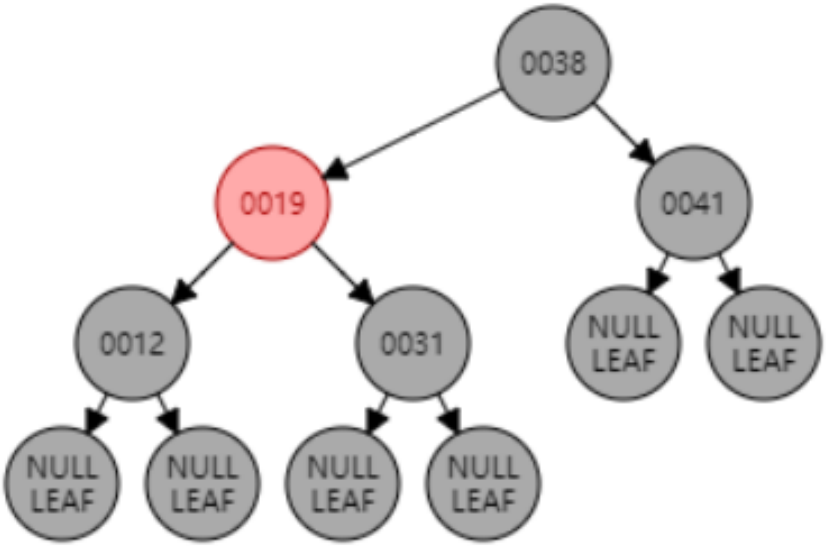

Figure 7: after inserting 41

14 Find set

To write a nonrecursive version of $Find - Set$ with path compression, let x be the element we call the function on. Create a linked list A which contains a pointer to x. Each time we most one element up the tree, insert a pointer to that elemetn into A. Once the root r has been found, use the linked list to find each node on the path from the root to x and update its parent to r.

```
# 2to3 sanity
from __future__ import (
    absolute_import, division, print_function, unicode_literals,
)

# Third-party libraries
import numpy as np


class UnionFind(object):
    def __init__(self, elements=None):
        self.n_elts = 0  # current num of elements
        self.n_comps = 0  # the number of disjoint sets or components
        self._next = 0  # next available id
        self._elts = []  # the elements
        self._indx = {}  #  dict mapping elt -> index in _elts
        self._par = []  # parent: for the internal tree structure
        self._siz = []  # size of the component - correct only for roots
```

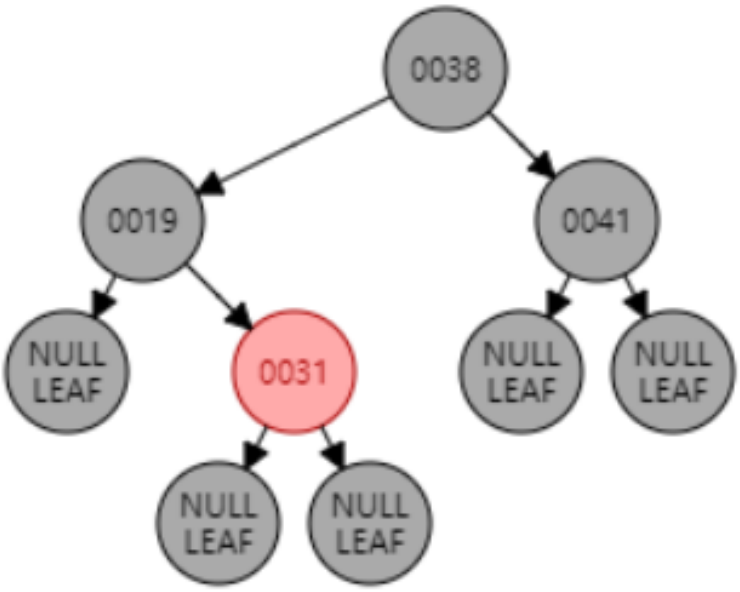

Figure 8: after inserting 38

```
    if elements is None:
        elements = []
    for elt in elements:
        self.add(elt)


def __repr__(self):
    return (
        '<UnionFind:\n\telts={},\n\tsiz={},\n\tpar={}>'
        .format(
            self._elts,
            self._siz,
            self._par,
            self.n_elts,
            self.n_comps,
        ))

def __len__(self):
    return self.n_elts

def __contains__(self, x):
    return x in self._indx

def __getitem__(self, index):
    if index < 0 or index >= self._next:
        raise IndexError('index {} is out of bound'.format(index))
```

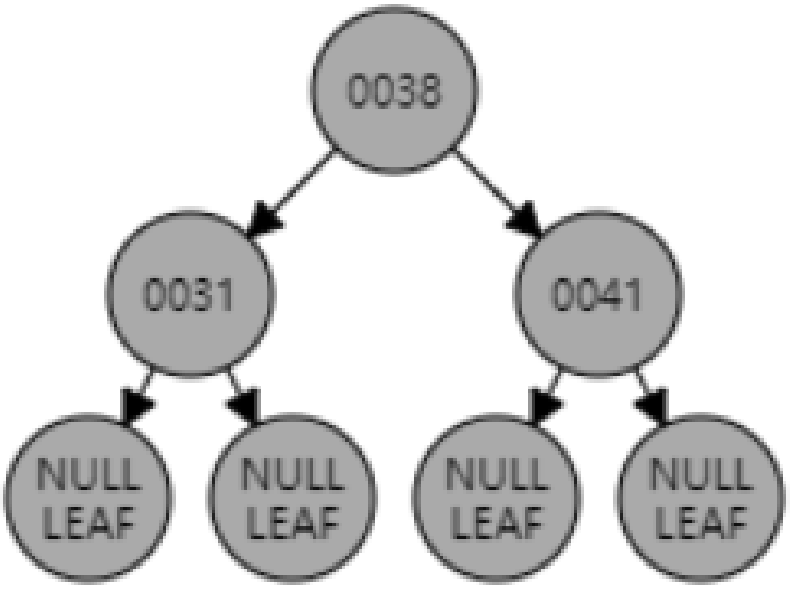

Figure 9: after inserting 31

```
    return self._elts[index]

def __setitem__(self, index, x):
    if index < 0 or index >= self._next:
        raise IndexError('index {} is out of bound'.format(index))
    self._elts[index] = x

def add(self, x):
    """Add a single disjoint element.
    Parameters
    ----------
    x : immutable object
    Returns
    -------
    None
    """
    if x in self:
        return
    self._elts.append(x)
    self._indx[x] = self._next
    self._par.append(self._next)
    self._siz.append(1)
    self._next += 1
    self.n_elts += 1
```

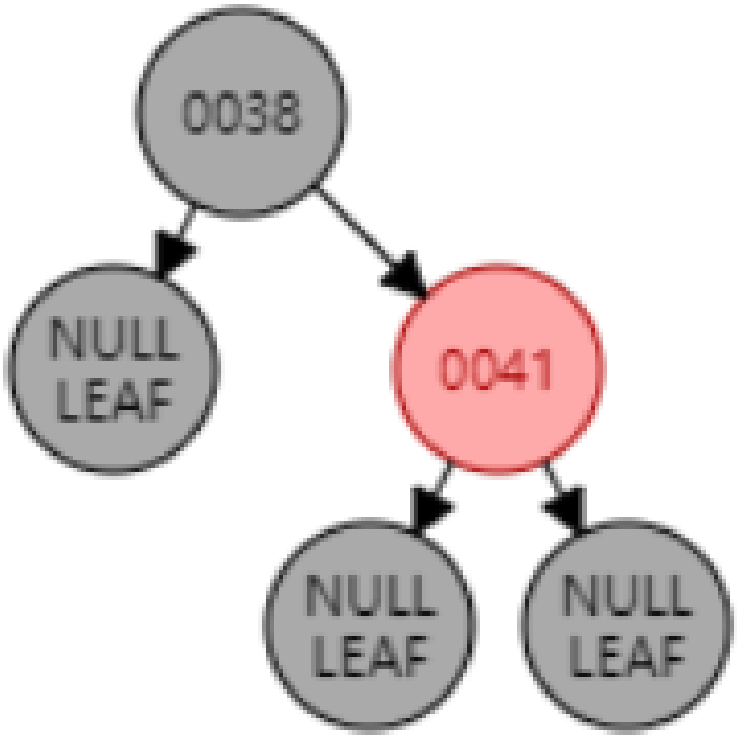

Figure 10: after inserting 12

```
        self.n_comps += 1

    def find(self, x):
        """Find the root of the disjoint set containing the given element.
        Parameters
        ----------
        x : immutable object
        Returns
        -------
        int
            The (index of the) root.
        Raises
        ------
        ValueError
            If the given element is not found.
        """
        if x not in self._indx:
            raise ValueError('{} is not an element'.format(x))
```

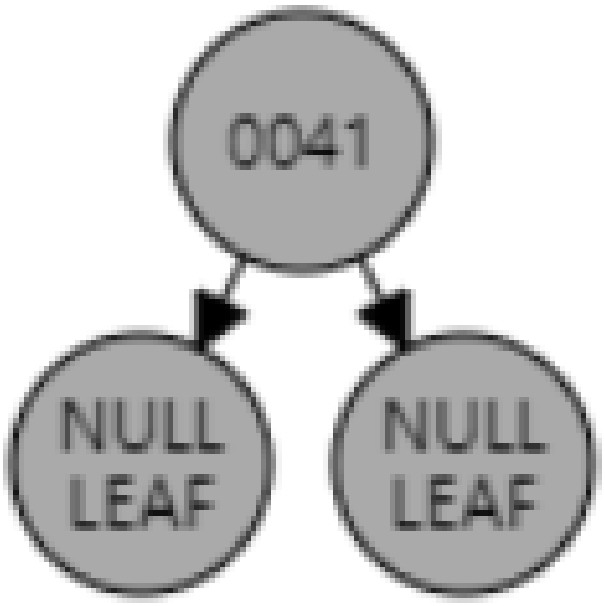

Figure 11: after inserting 19

```
    p = self._indx[x]
    while p != self._par[p]:
        # path compression
        q = self._par[p]
        self._par[p] = self._par[q]
        p = q
    return p

def connected(self, x, y):
    """Return whether the two given elements belong to the same compon
    Parameters
    ----------
    x : immutable object
    y : immutable object
    Returns
    -------
    bool
        True if x and y are connected, false otherwise.
    """
    return self.find(x) == self.find(y)

def union(self, x, y):
    """Merge the components of the two given elements into one.
    Parameters
    ----------
```

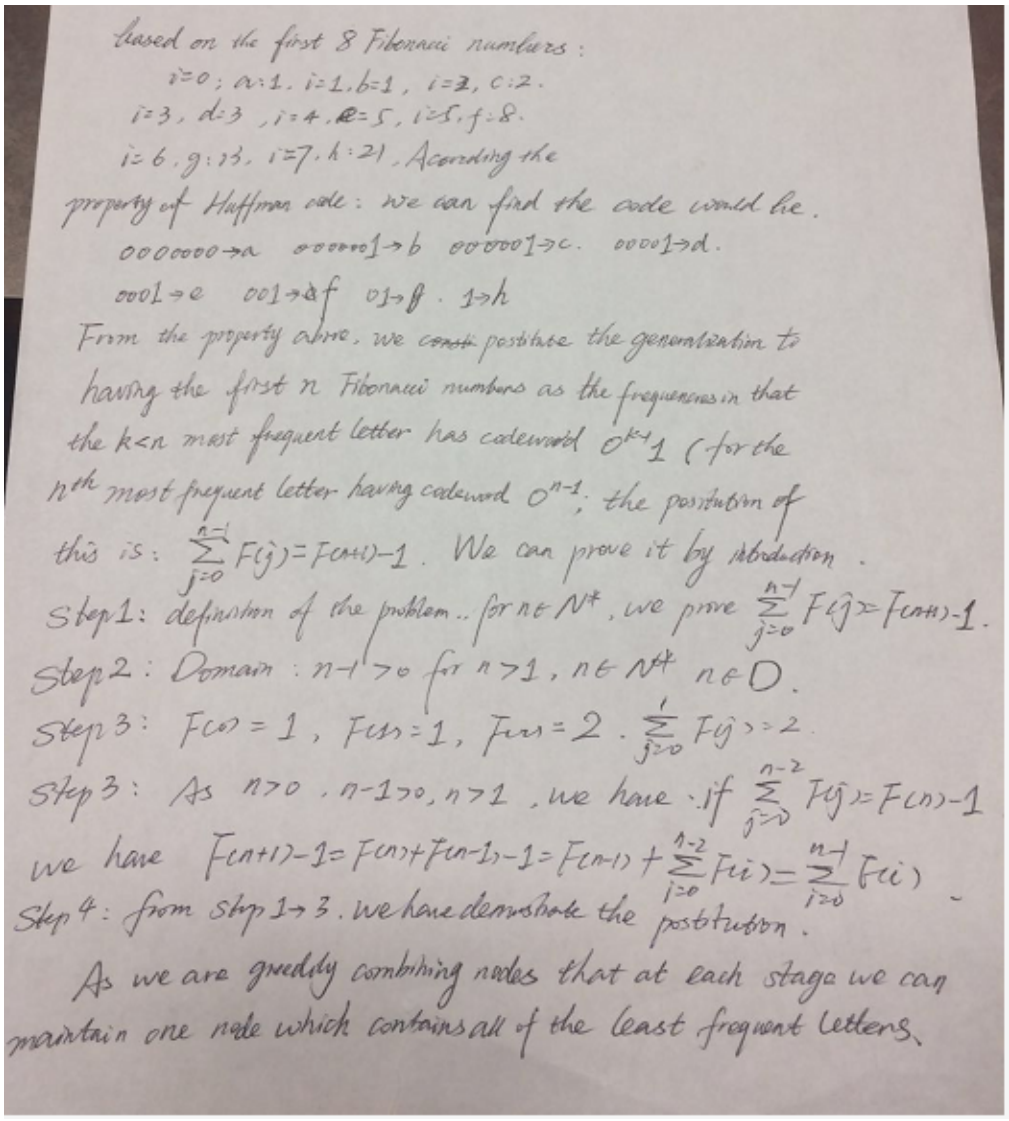

Figure 12: Hoffman code1

```
x : immutable object
y : immutable object
Returns
_______
None
"""
# Initialize if they are not already in the collection
for elt in [x, y]:
    if elt not in self:
        self.add(elt)

xroot = self.find(x)
yroot = self.find(y)
if xroot == yroot:
    return
```

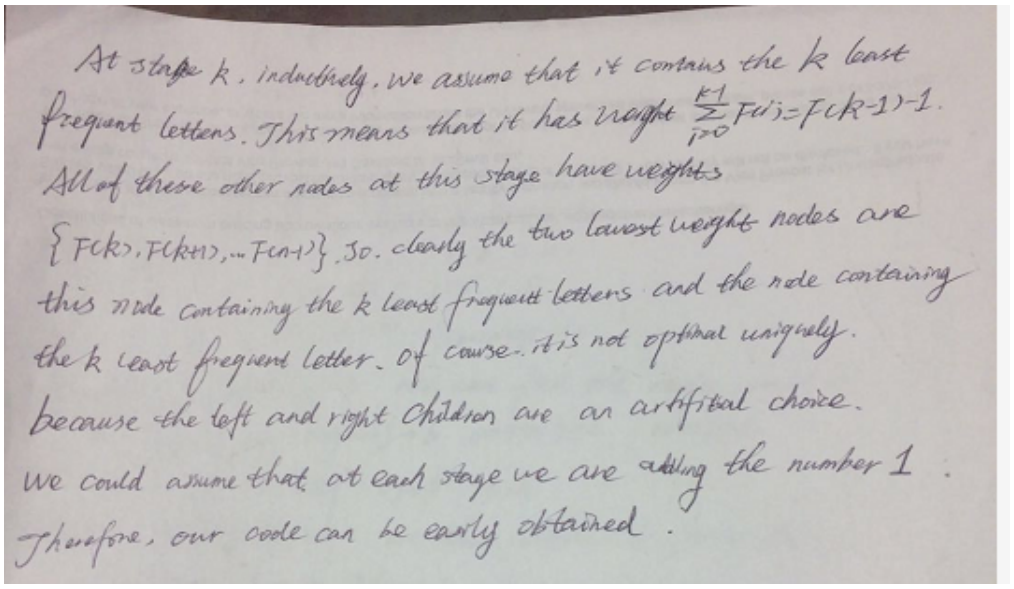

Figure 13: Hoffman code2

```
        if self._siz[xroot] < self._siz[yroot]:
            self._par[xroot] = yroot
            self._siz[yroot] += self._siz[xroot]
        else:
            self._par[yroot] = xroot
            self._siz[xroot] += self._siz[yroot]
        self.n_comps -= 1

    def component(self, x):
        """Find the connected component containing the given element.
        Parameters
        _____
        x : immutable object
        Returns
        _____
        set
        Raises
        _____
        ValueError
            If the given element is not found.
        """
        if x not in self:
            raise ValueError('{} is not an element'.format(x))
        elts = np.array(self._elts)
        vfind = np.vectorize(self.find)
        roots = vfind(elts)
        return set(elts[roots == self.find(x)])

    def components(self):
```

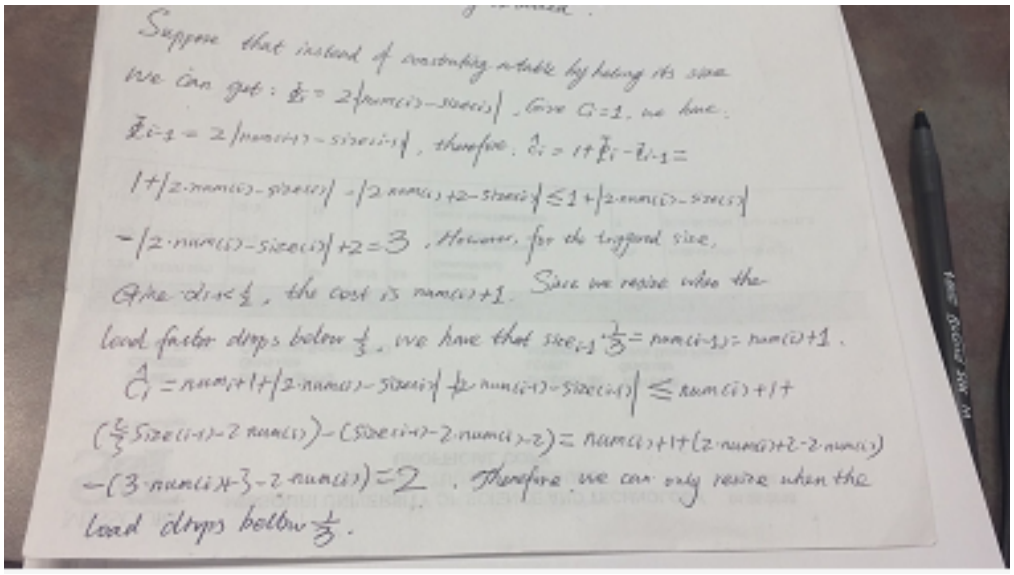

Figure 14: amortized cost

```
        """Return the list of connected components.
        Returns
        -------
        list
            A list of sets.
        """
        elts = np.array(self._elts)
        vfind = np.vectorize(self.find)
        roots = vfind(elts)
        distinct_roots = set(roots)
        return [set(elts[roots == root]) for root in distinct_roots]
        # comps = []
        # for root in distinct_roots:
        #     mask = (roots == root)
        #     comp = set(elts[mask])
        #     comps.append(comp)
        # return comps

    def component_mapping(self):
        elts = np.array(self._elts)
        vfind = np.vectorize(self.find)
        roots = vfind(elts)
        distinct_roots = set(roots)
        comps = {}
        for root in distinct_roots:
            mask = (roots == root)
            comp = set(elts[mask])
            comps.update({x: comp for x in comp})
            # Change ^this^, if you want a different behaviour:
```

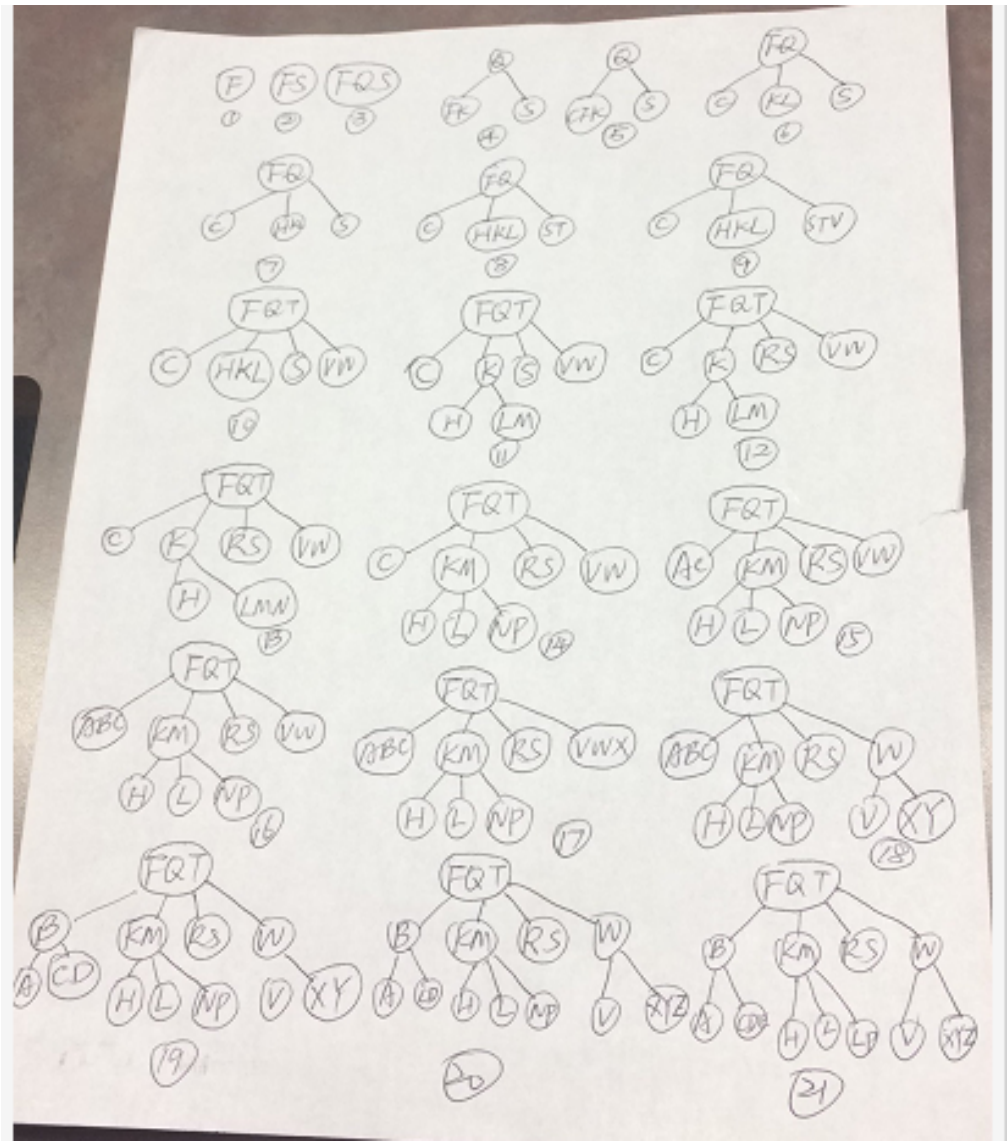

Figure 15: B-tree inserting

```
    # If you don't want to share the same set to different keys:
    # comps.update({x: set(comp) for x in comp})
return comps
```

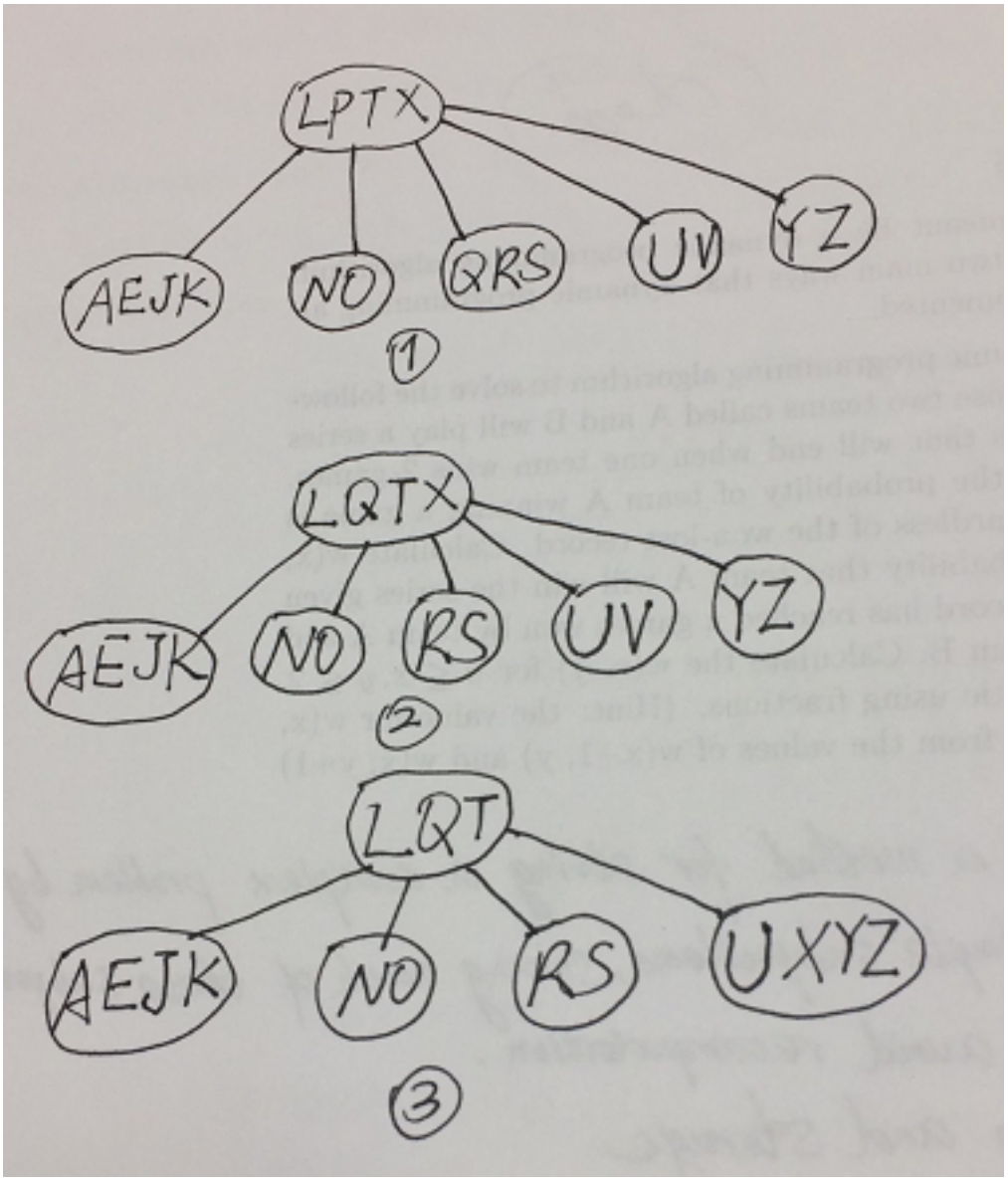

Figure 16: deleting operation

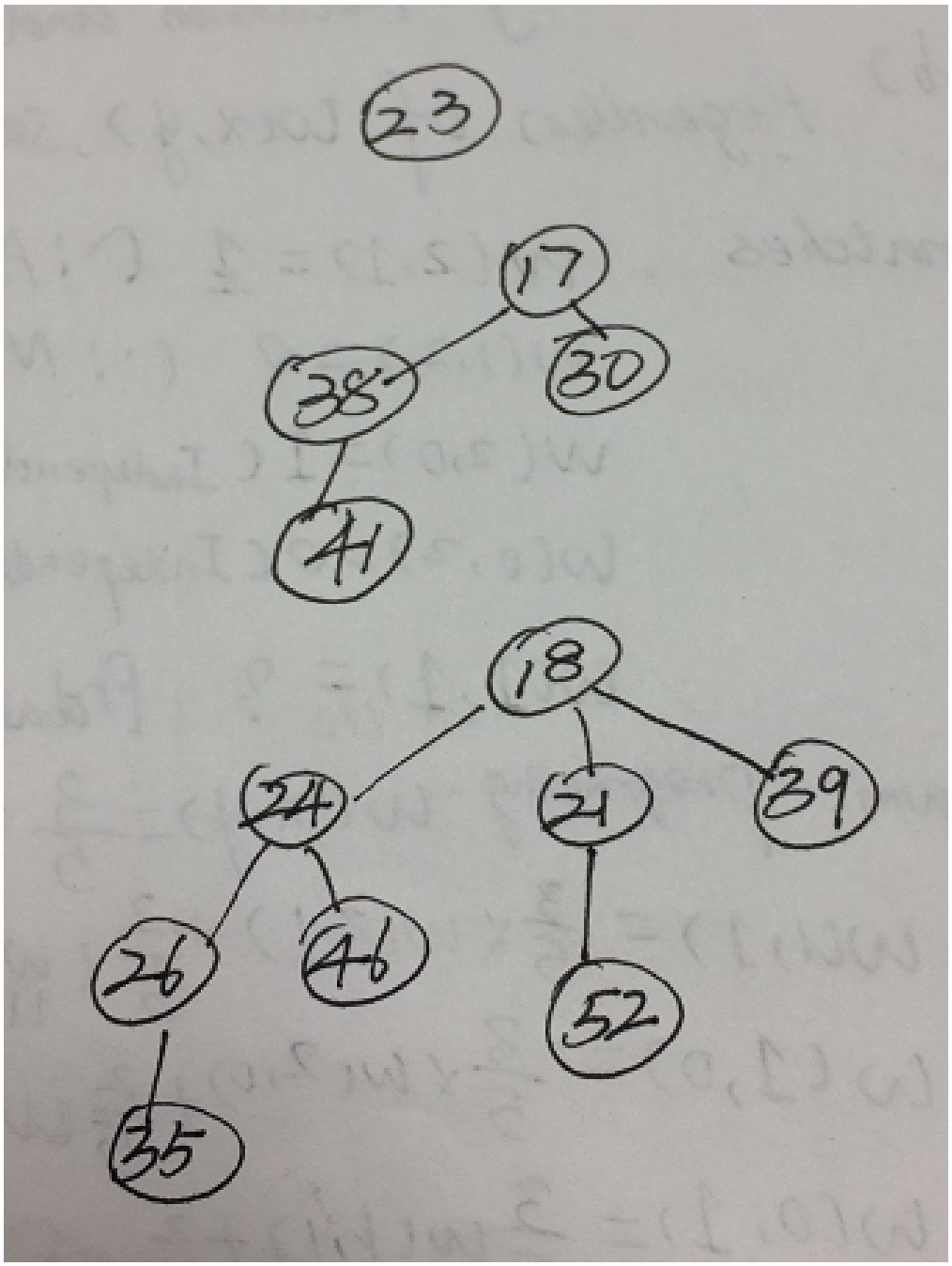

Figure 17: heaps