Institute of
Pervasive Computing

Randomized Treaps

Algorithms and Data Structures 2, 340300
Lecture – 2023W
Univ.-Prof. Dr. Alois Ferscha, teaching@pervasive.jku.at

**JOHANNES KEPLER
UNIVERSITY LINZ**
Altenberger Straße 69
4040 Linz, Austria
jku.at

Remember :: Heaps

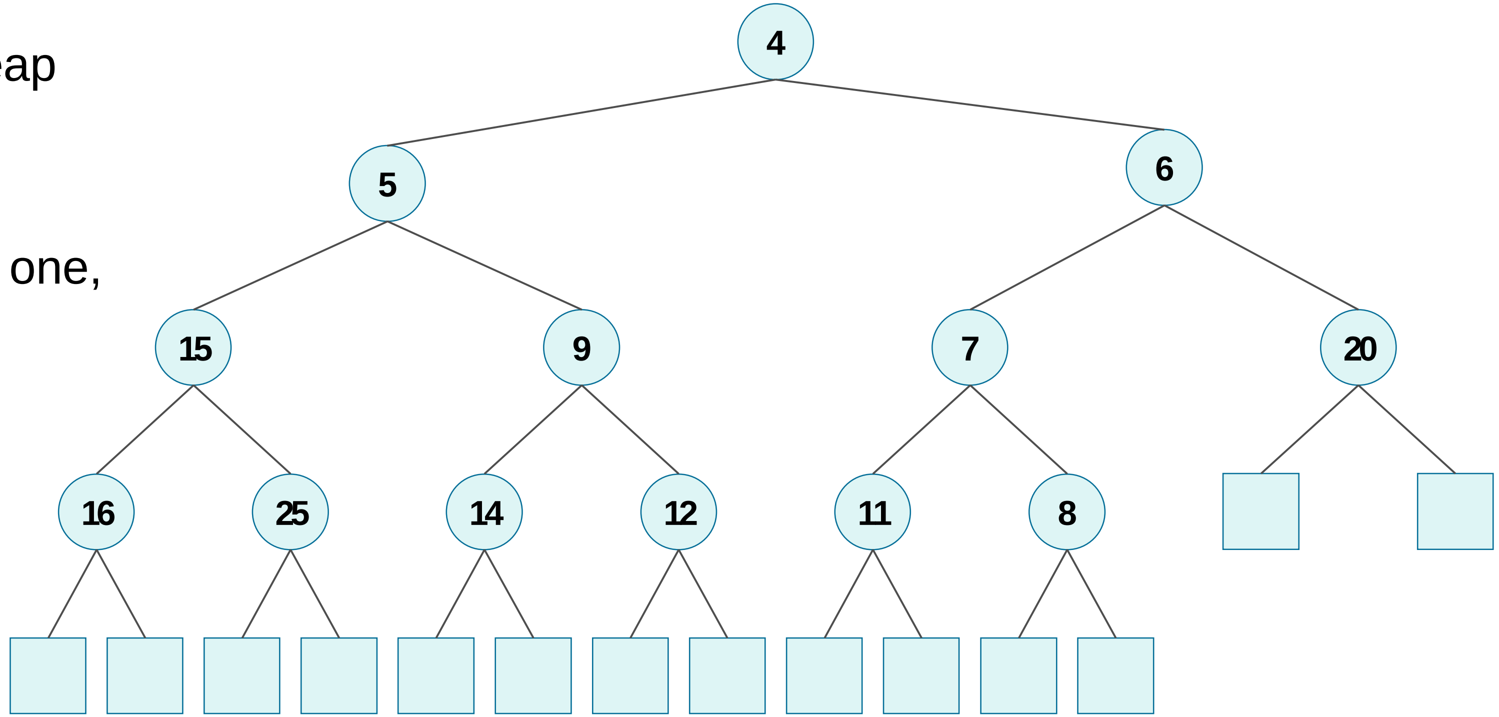
A **heap** is a binary tree *T* whose internal nodes contain a set of keys (or key-value pairs) stored with the following properties

order property:

key(parent) ≤ key(child) for Min-Heap

structural property:

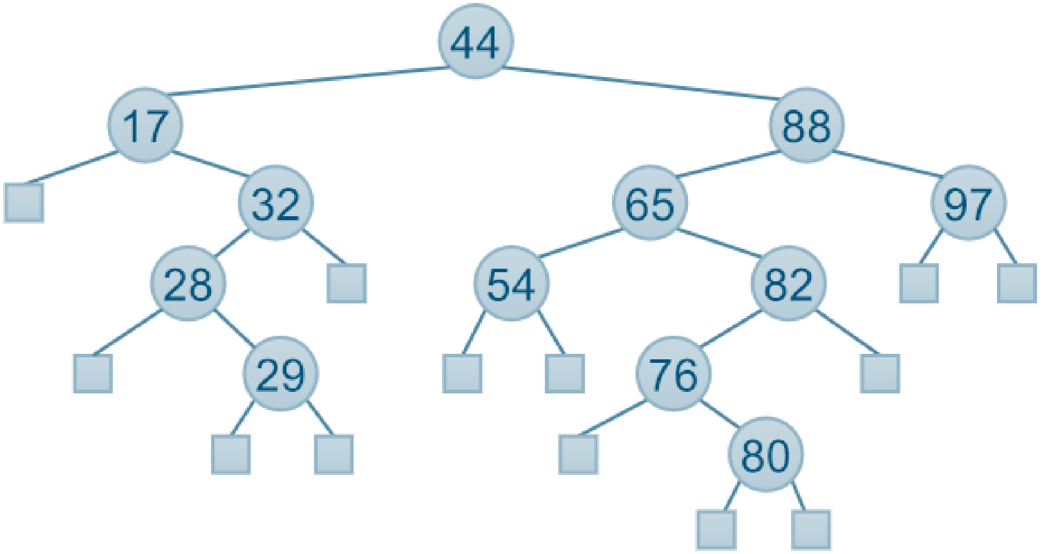
All levels are full except the lowest one, which is filled up from the left (almost complete binary tree)



Remember :: Binary Search Trees

A **binary search tree** is a **binary tree** ***T*** where:

- Each internal node stores a **key-value pair** of a dictionary
- Keys which are stored in nodes of the **left** subtree of a node *v*, are **less than** or **equal** to the key stored in *v*
- Keys which are stored in nodes of the **right** subtree of a node *v*, are **greater** than the key stored in *v*
- **External nodes** serve only as **placeholders** and do not store elements



Treaps

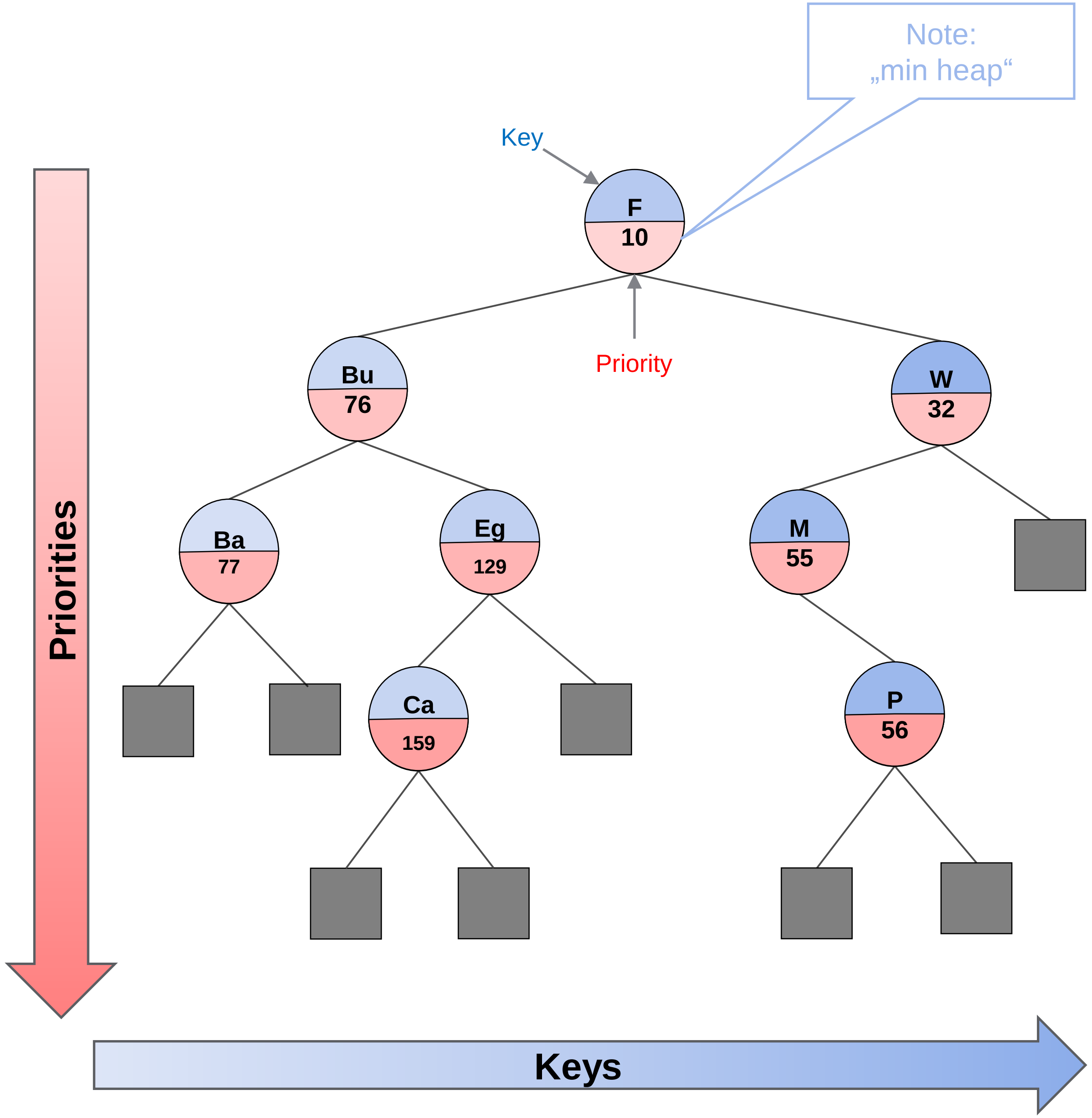
A Treap is a combination of a binary search tree and a heap

Keys of the treap are treated as **keys of a BST**

- keys define an **ordering** in the treap **from left to right**
- keys(left subtree) < key(parent)
- keys(right subtree) > key(parent)

Priorities of the treap are treated as **priorities of a min heap**

- priorities define an **ordering from top to bottom**
- priority(parent) > priority(child)



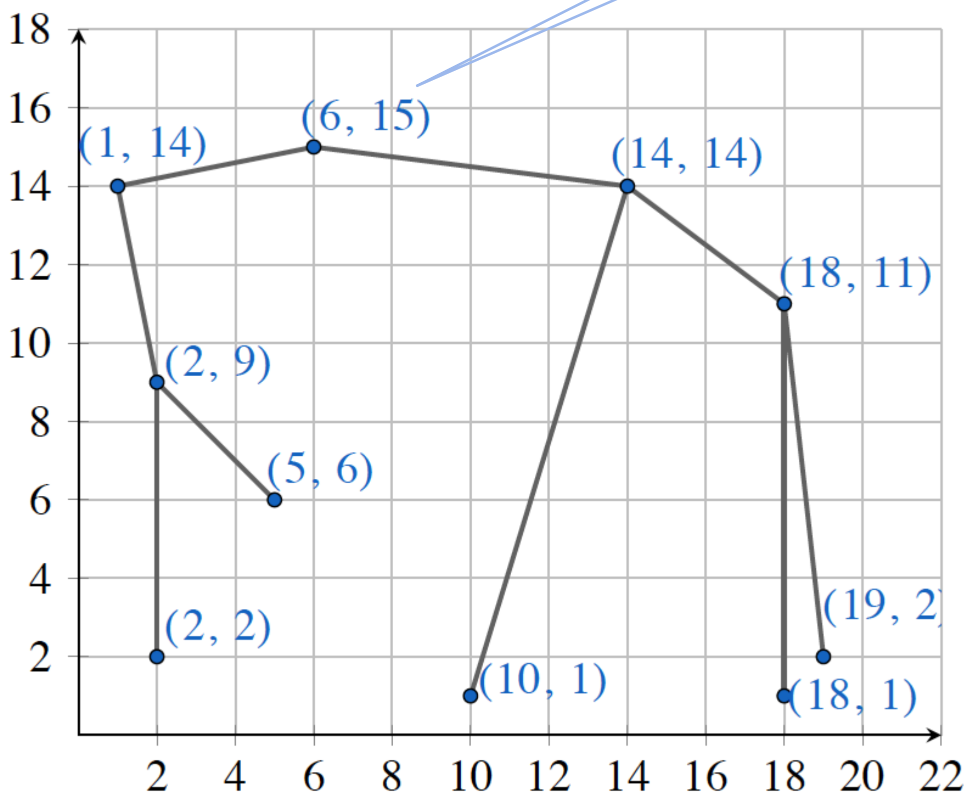
Treaps

often referred to as a "**cartesian tree**",
as it is easy to embed it in a Cartesian plane

Note:
„max heap“

A Treap is a data structure that stores pairs **(X,Y)** in a **binary tree** in such a way that it is

- a **binary search tree** by **X**, and
- a **binary heap** by **Y**.



If some node of the tree contains values **(X_0,Y_0)**,

- all nodes in the **left** subtree have **X** ≤ **X_0**,
- all nodes in the **right** subtree have **X_0** ≤ **X**,
- and all nodes in both left and right subtrees have **Y** ≤ **Y_0**.

Treaps

Advantages

X values are the **keys** (and at same time the values stored in the treap),
and
Y values are called **priorities**.

Without priorities, the treap would be a regular binary search tree by **X**, and one set of **X** values could correspond to a lot of different trees, some of them degenerate (for example, in the form of a linked list), and therefore extremely slow (the main operations would have **O(N)** complexity).

At the same time, **priorities** (when they're unique) **allow to uniquely specify** the tree that will be constructed (of course, it does not depend on the order in which values are added). Obviously, if **the priorities are chosen randomly**, non-degenerate trees will emerge on average, which will ensure **O(log N)** complexity for the main operations.

Hence another name for this data structure: **randomized binary search tree.**

Treaps :: Operations

A treap provides the following operations:

- **Insert (X,Y)** in $O(\log N)$.
  Adds a new node to the tree. One possible variant is to pass only *X* and **generate *Y* randomly** inside the operation.
- **Search (X)** in $O(\log N)$.
  Looks for a node with the specifed key value *X*. The implementation is the same as for an ordinary binary search tree.
- **Erase (X)** in $O(\log N)$.
  Looks for a node with the specifed key value *X* and removes it from the tree.
- **Build ($X_1, \ldots, X_N$)** in $O(N)$.
  Builds a tree from a list of values. This can be done in linear time (assuming that $X_1, \ldots, X_N$ are sorted).
- **Union (T_1,T_2)** in $O(M \log(N / M))$.
  Merges two trees, assuming that all the elements are different.
  It is possible to achieve the same complexity if duplicate elements should be removed during merge.
- **Intersect (T_1,T_2)** in $O(M \log(N / M))$.
  Finds the intersection of two trees (i.e. their common elements). We will not consider the implementation of this operation here.

In addition, due to the fact that a treap is a binary search tree, it can implement other operations,
such as finding the ***K***-th largest element or finding the **index** of an element.

Treaps :: Implementation :: Split

Split (*T* , *X*) separates tree ***T*** in 2 subtrees ***L*** and ***R*** trees (which are the return values of split) so that
L contains all elements with key $X_L \leq X$, and
R contains all elements with key $X_R > X$.

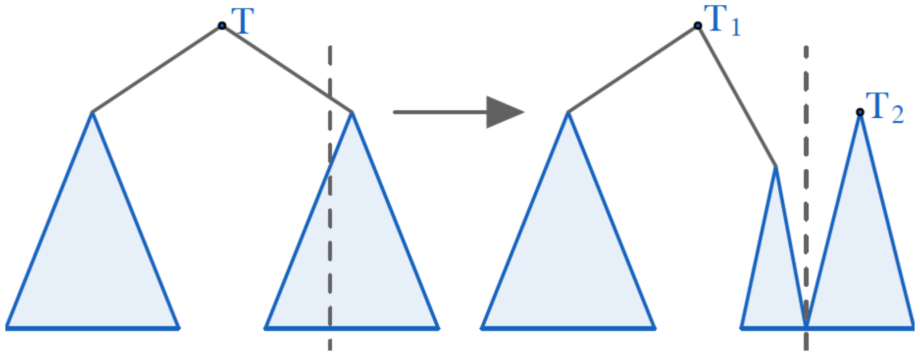
This operation has $O(\log N)$ complexity and is implemented using a clean **recursion**:

1. If the value of the root node (**R**) is ≤ ***X***, then **L** would at least consist of **R->L** and **R**.
We then call split on **R->R** , and note its split result as **L'** and **R'**. Finally, **L** would also contain **L'**, whereas **R =R'**.

2. If the value of the root node (**R**) is > ***X***, then **R** would at least consist of **R** and **R->R**. We then call split on **R->L**, and note its split result as **L'** and **R'**. Finally, **L=L'**, whereas **R** would also contain **R'**.

Thus, the split algorithm is:

1. decide **which subtree** the **root** node would belong to (left or right)
2. **recursively call split** on one of its children
3. create the **final result** by **reusing** the **recursive split call**.



https://cp-algorithms.com/data_structures/treap.html

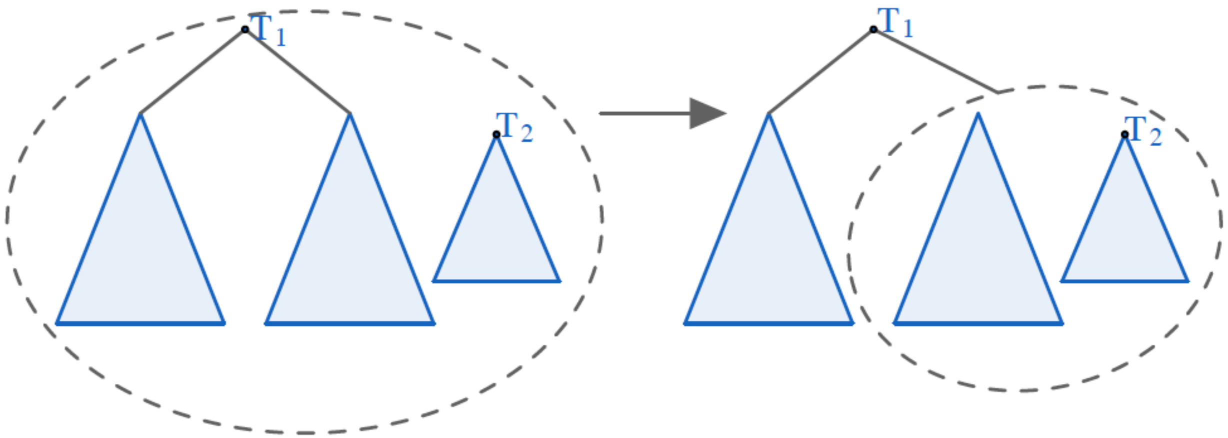
Treaps :: Implementation :: Merge

Merge (T_1,T_2) combines **two subtrees T_1** and **T_2** and returns the **new tree**.

This operation also has $O(\log N)$ complexity.
It works under the assumption that T_1 and T_2 are ordered (all keys X in T_1 are smaller than keys in T_2).

Thus, we need to combine these trees without violating the order of priorities Y.

To do this, we **choose** as the **root** the tree which has higher priority Y in the root node,
and **recursively** call **Merge** for the other tree and the corresponding subtree of the selected root node.



https://cp-algorithms.com/data_structures/treap.html

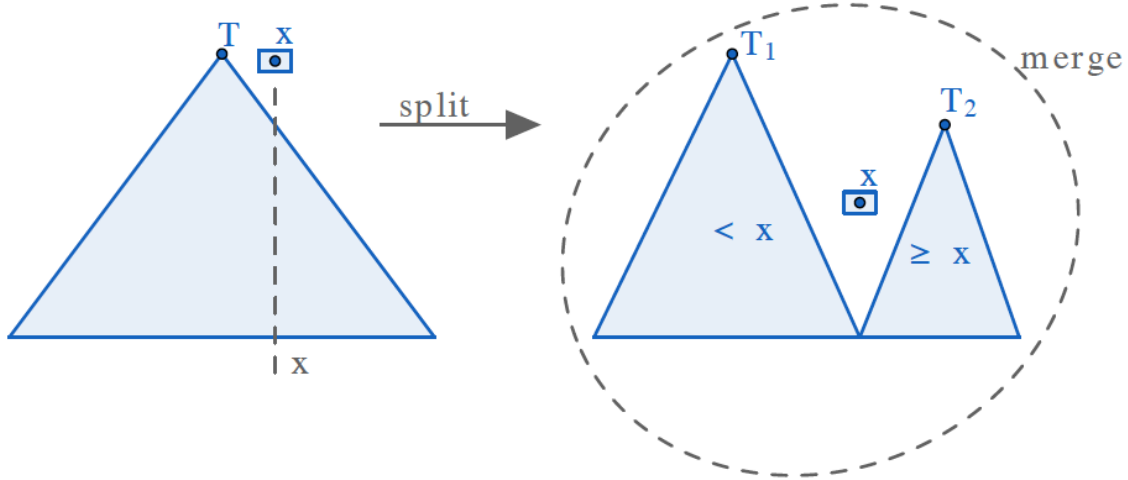
Treaps :: Implementation :: Insert

Insert (*X* , *Y*)
First we **descend** in the tree (as in a regular binary search tree by X),
and stop at the first node in which the **priority** value is less than ***Y***.

We have found the **place where we will insert** the new element.

Next, we call **Split (T, X)** on the subtree starting at the found node,
and use returned subtrees ***L*** and ***R*** as left and right children of the new node.

Alternatively, insert can be done by
splitting the initial treap on ***X***
and doing **2** merges with the new node.



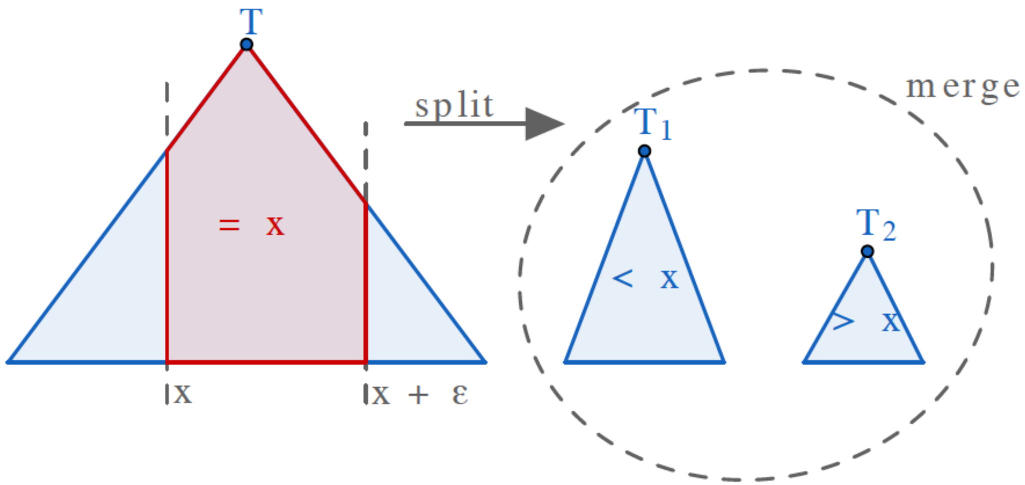
https://cp-algorithms.com/data_structures/treap.html

Treaps :: Implementation :: Erase

Erase (*X*)

First we descend in the tree (as in a regular binary search tree by),
looking for the element we want to delete. Once the node is found, we call **Merge** on its children and put the return value of the operation in the place of the element we're deleting.

Alternatively, we can factor out the subtree holding ***X*** with **2** split operations and merge the remaining treaps.



Treaps :: Implementation :: Build / Union

Build

We implement **Build** operation with $O(N \log N)$ complexity using ***N* Insert** calls.

Union

Union (T_1,T_2) has theoretical complexity $O(M \log(N/M))$, but in practice it works very well, probably with a very small hidden constant.

Let's assume without loss of generality that $T_1 \rightarrow Y > T_2 \rightarrow Y$, i. e. root of T_1 will be the root of the result.

To get the result, we need to merge trees $T_1 \rightarrow L$, $T_1 \rightarrow R$ and T_2 in two trees which could be children of T_1 root.

To do this, we call Split (T_2, $T_1 \rightarrow X$), thus splitting T_2 in two parts L and R, which we then recursively combine with children of T_1 : Union ($T_1 \rightarrow L$, L) and Union ($T_1 \rightarrow R$, R), thus getting left and right subtrees of the result.

Treaps :: Implementation

```
class item:
    int key, prio
    item left, right
```

```
void split(t, key, l, r):
```

split treap **t** by value **key** into two treaps, and store the left treap in **l** and right treap in **r**

```
    if t is null:
        global left = global right = null
```

if there are no further splits, we clear the subtrees which were found so far and stop

```
    else if t.key <= key:
        split(t.right, key, t.right, r)
        global left = t
    else:
        split(t.left, key, l, t.left)
        global right = t
```

Recursively call split function based on two cases:

- Root node value is **≤** key: split treap **t.right** (right subtree of **t**) by value **key** and store the left subtree in **t.right** and right subtree in **r;** set **global left = t** . so that the **global left** result value contains **t.left,** , **t** as well as **t.right** (which is the result of the recursive call we made)
- Vice versa for the case that the node value > key with the left subtree

```
global left = global right = null
split(t=root, key=5, l=null, r=null)
```

call split with the treap defined by the root node, the key to split by as 5 and the left and right subtrees initially empty

The resulting subtrees are stored in **global left** and **global right**

Treaps :: Implementation

This **split** function can be tricky to understand,

Let us understand in words what the function call **split(t, key, l, r)** intends:

1. When the root node value is **≤** key, we call split **(t->r, key, t->r, r)** , which means: "split treap **t->r** (right subtree of **t**) by value **key** and store the left subtree in **t->r** and right subtree in **r** ". After that, we set **l = t** . Note now that the **l** result value contains **t->l** , **t** as well as **t->r** (which is the result of the recursive call we made) all already merged in the correct order!

2. When the root node value is greater than key, we call **split (t->l, key, l, t->l)** , which means: "split treap **t->l** (left subtree of **t**) by value **key** and store the left subtree in **l** and right subtree in **t->l** ". After that, we set **r = t** . Note now that the **r** result value contains **t->l** (which is the result of the recursive call we made), **t** as well as **t->r** , all already merged in the correct order!

Treaps :: Implementation

```
void insert(item t, item it):
    if t is null:
        t = it
```

If **t** is empty (meaning the tree is fully traversed until the insert position), make the new node the (sub) root

```
    else if it.prio > t.prio:
        split(t, it.key, it.left, it.right)
        it.left = global left
        it.right = global right
        t = it
```

Fulfill the priority requirements by splitting the tree and inserting the resulting subtrees

```
    else
        insert(t.right if t.key <= it.key else t.left, it)
```

Fulfill the BST requirements by finding the correct insert position

Treaps :: Implementation

```
void merge(item t, item l, item r):
    if left is empty or right is empty:
         t = global left if left not null
         else global right

    else if l.prio > r.prio
         merge(l.right, l.right, r)
         t = global left

    else
         merge(r.left, l, r.left)
         t = global right
```

Set the root to the appropriate resulting tree

Choose as (sub) root the subtree with the higher priority

Merge with the remainder of the tree

Treaps :: Implementation

```
void erase(t, key):
    if t.key == key:
         item helper = t
         t = merge(t, t.l, t.r)
         delete th
         return t

    else
         return erase(t.left if key < r.key else t.right, key)
```

Merge left and right subtrees of node which should be removed and remove the node itself

Find the node to remove recursively

```
void unite(item l, item r):
    if l is null or r is null:
         return l if l is not null else r

    if l.prio < r.prio:
         swap(l, r)
    item lt, rt = split (r, l.key, null, null)
    l.left = unite(l.left, lt)
    l.right = unite(l.right, rt)
    return l
```

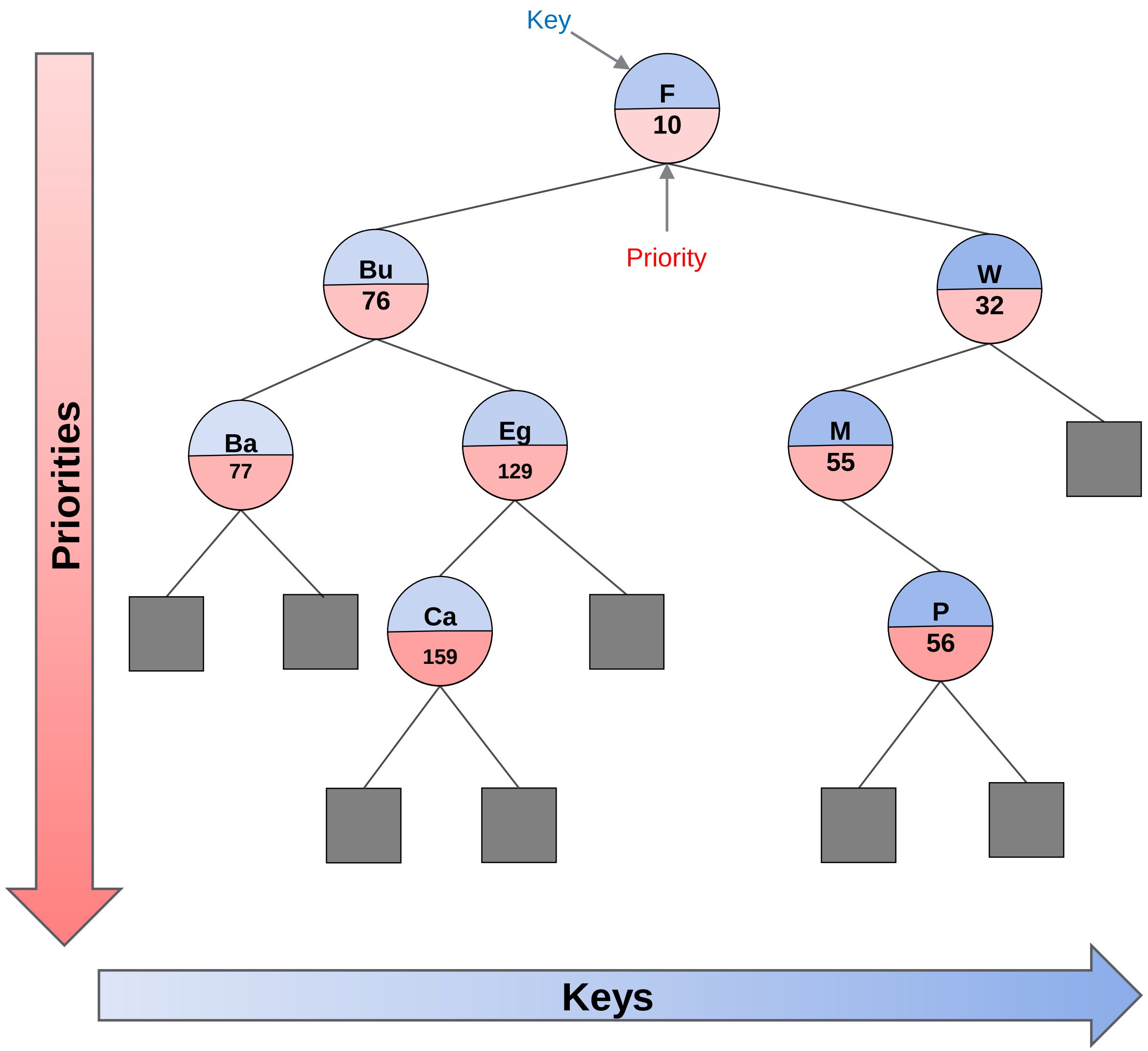
In-place sort of both trees
split accordingly and connect them back together
in correct order

Treaps :: Operations :: Search

Search for **key** just like in BST

Search for **value (priority)** can only return highest priority, which is the root

```
search(node, targetKey)
    if node == null
        return null

    if node.key == targetKey
        return node
    elsif targetKey < node.key
        return search(node.left, targetKey)
    else
        return search(node.right, targetKey)
```

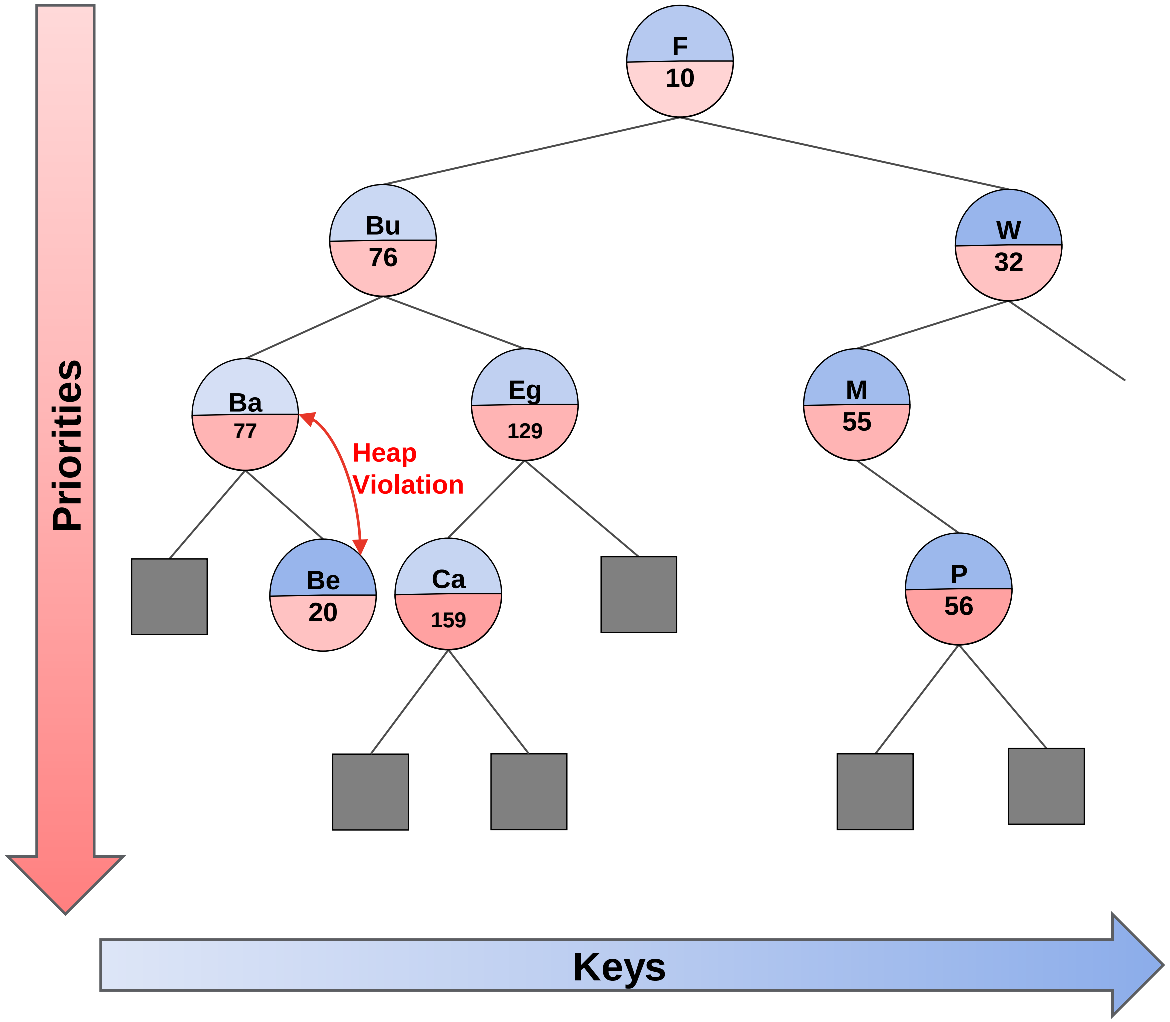


Treaps :: Operations :: Insert

Instead of recursive insert, also (AVL) rotations can be used

Both **BST** and **heap properties** have to be **adhered**

- e.g. *insert*(Be, 20) will violate **heap property**
- performing upheap will violate **BST property**



Treaps :: Operations :: Insert :: Rotations

Rotations

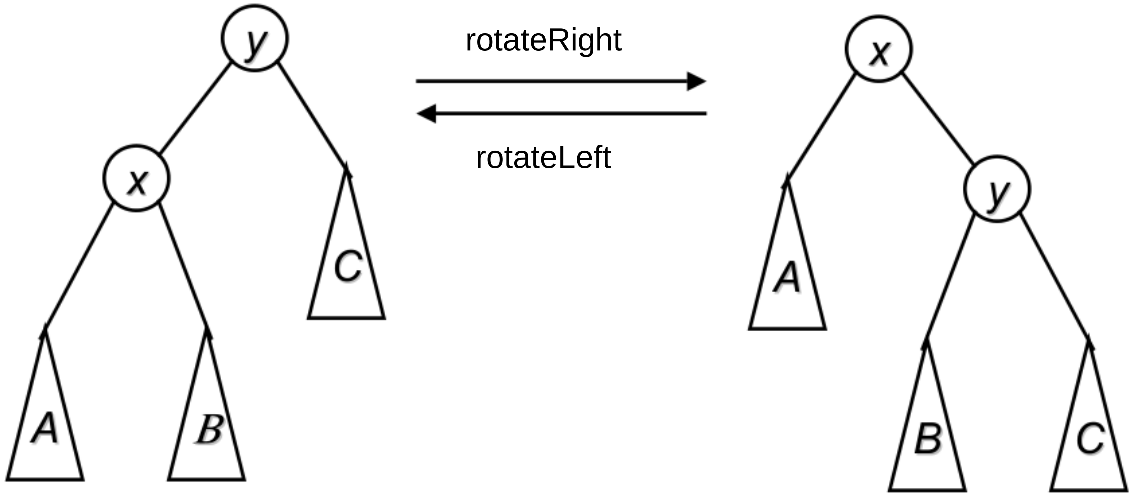
A rotation in a BST is a transformation to **invert** the **parent-child relation** of **two nodes**

We cannot just swap the two nodes without violating the restrictions

Treat entire subtrees like nodes

Insert with rotations

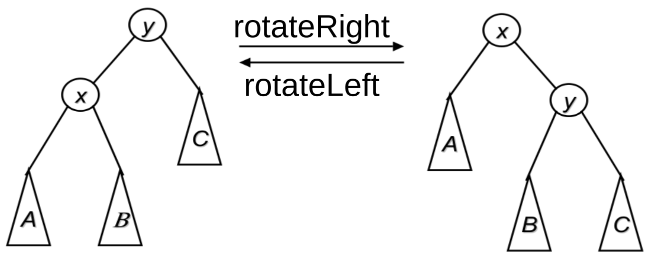
```
insert(x)

insert x according to BST position

while priority(parent(x)) > priority(x)
    if x is left child
        rotateRight(parent(x))
    else
        rotateLeft(parent(x))
```



Treaps :: Operations :: Insert :: Rotations

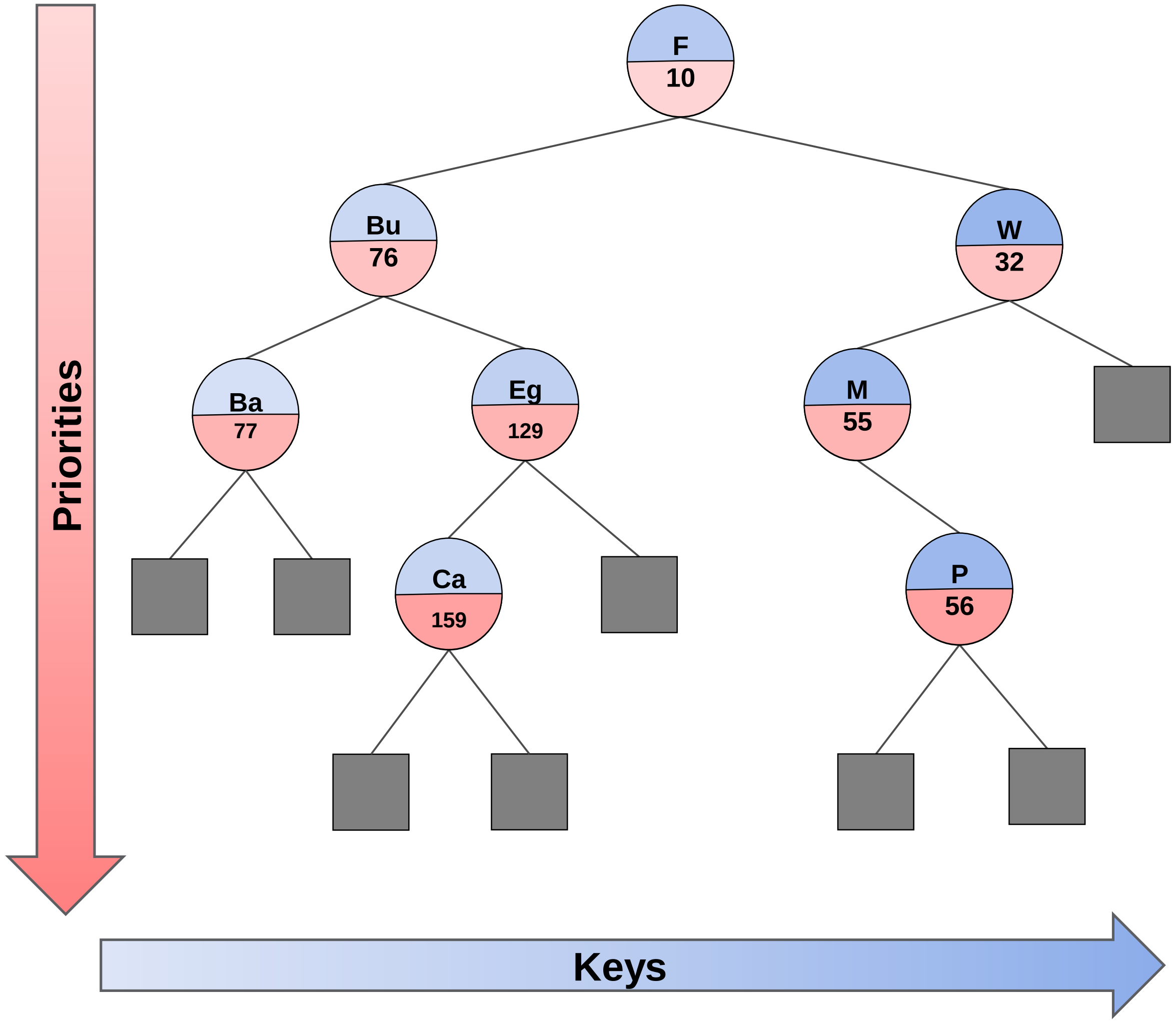
```
leftRotate(x)

if x is null or isRoot(x) then
    throw exception

y ← x.parent
    throw exception if y.right != x

p ← y.parent

if p != null then
    if p.left == y then
        p.setLeft(x)
    else
        p.setRight(x)

else
    treap.root ← x

y.setRight(x.left)
x.setLeft(y)
```

```
rightRotate(x)

if x is null or isRoot(x) then
    throw exception

y ← x.parent
    throw exception if y.left != x

p ← y.parent

if p != null then
    if p.left == y then
        p.setLeft(x)
    else
        p.setRight(x)

else
    treap.root ← x

y.setLeft(x.right)
x.setRight(y)
```



Treaps :: Operations :: Insert :: Example

insert(Be, 20)

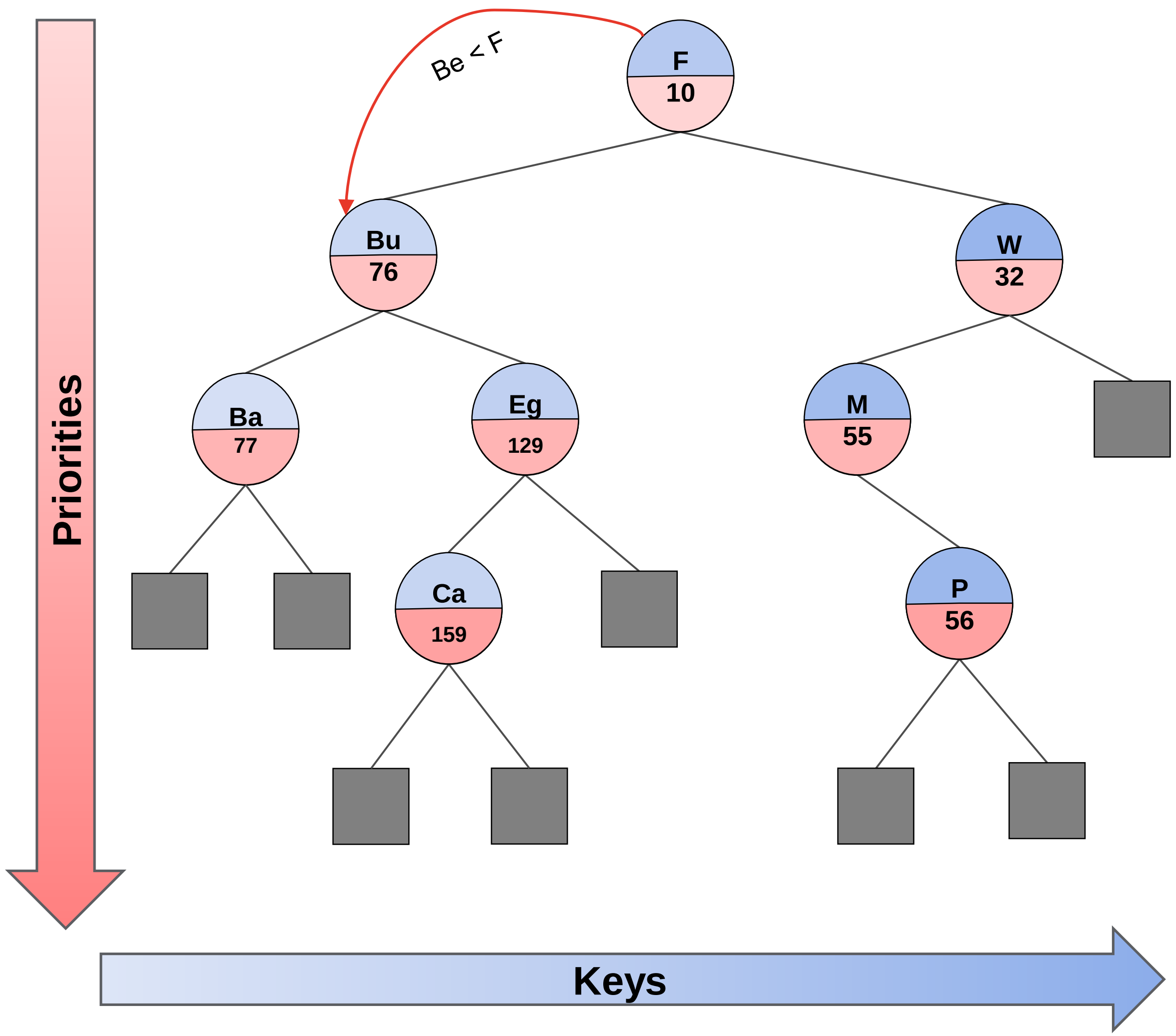


Treaps :: Operations :: Insert :: Example

insert(Be, 20)

insert according to BST insert

Be < F → follow left path



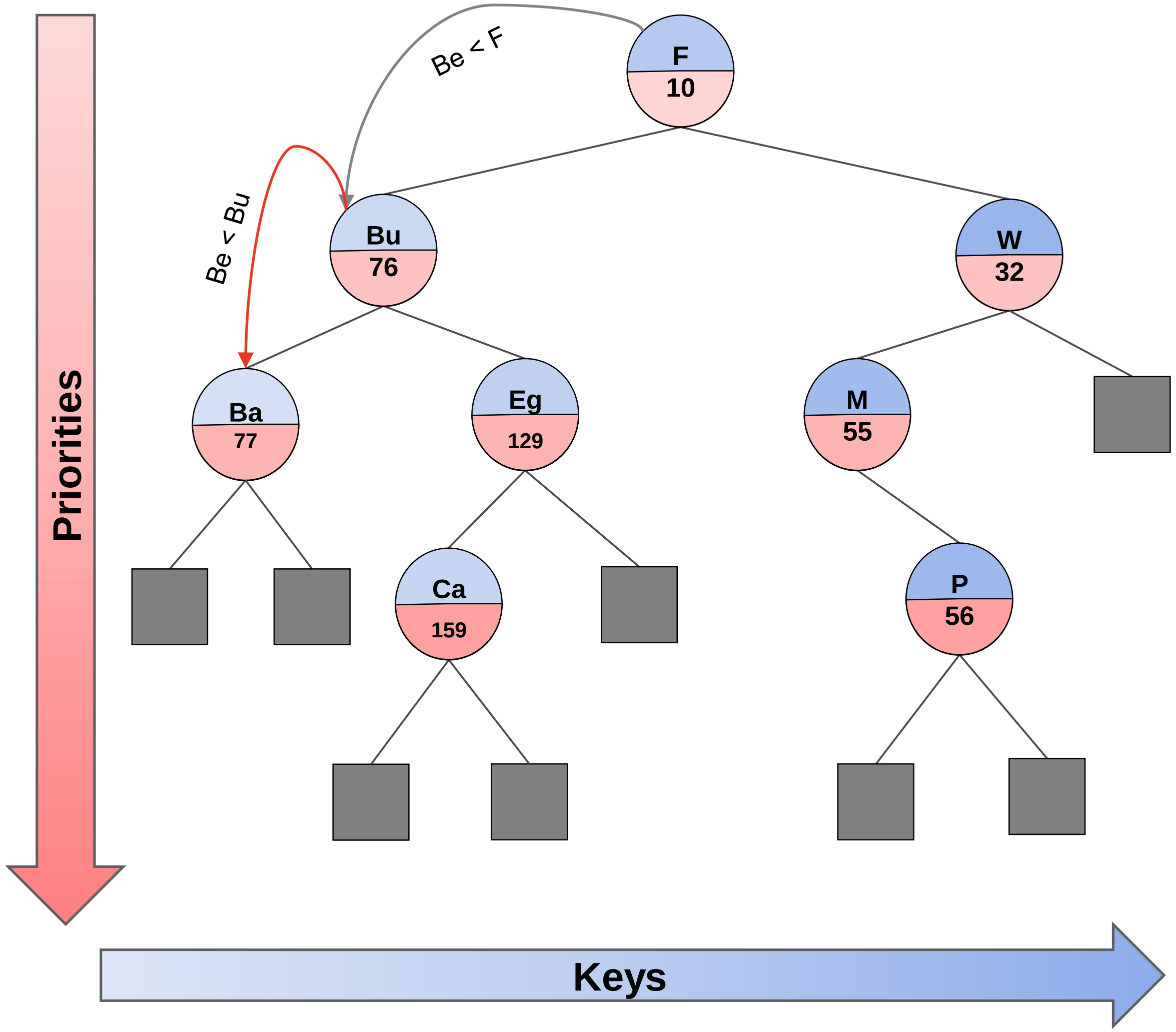
Treaps :: Operations :: Insert :: Example

insert(Be, 20)

insert according to BST insert

Be < F → follow left path

Be < Bu → follow left path



Treaps :: Operations :: Insert :: Example

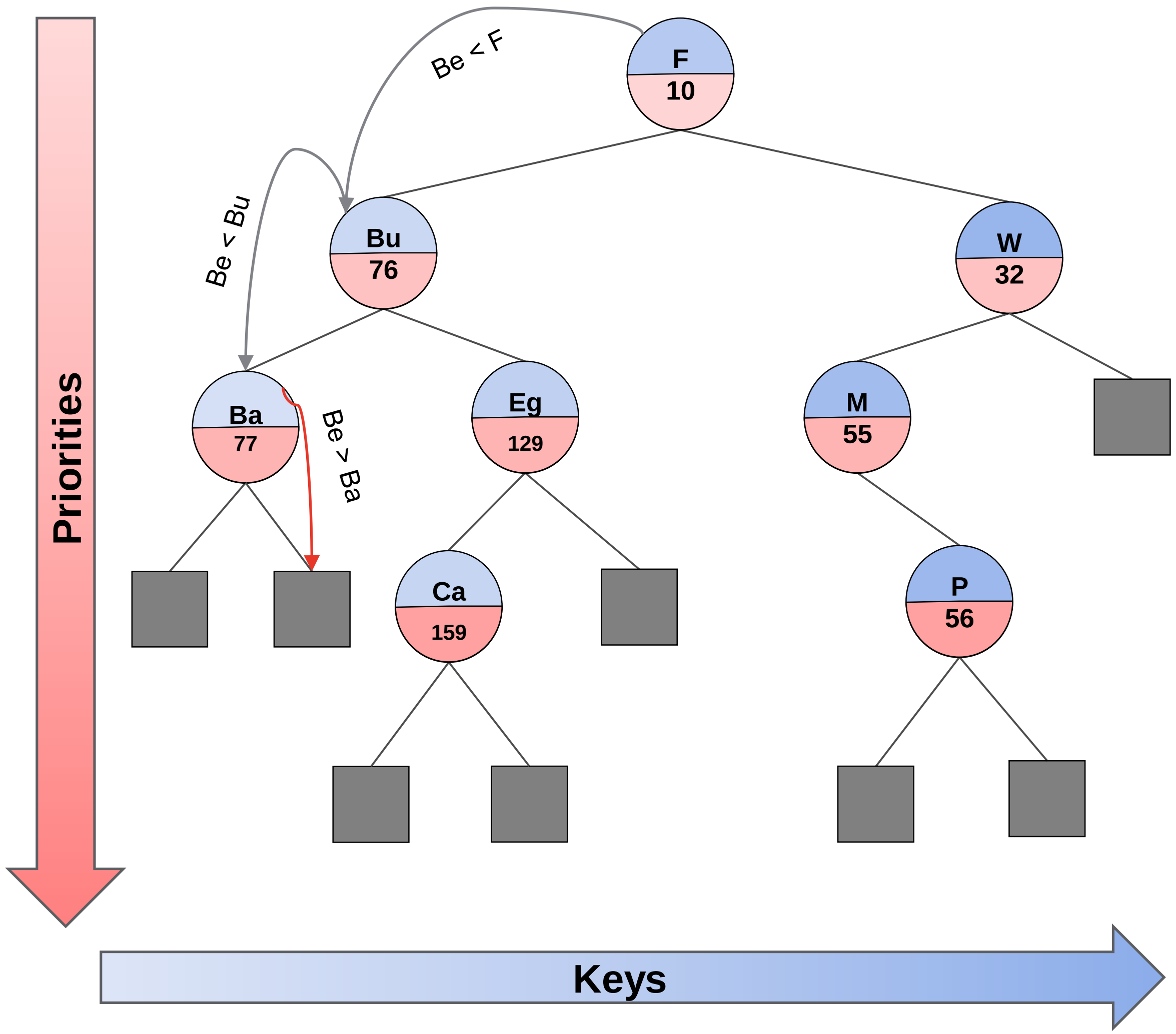
insert(Be, 20)

insert according to BST insert

Be < *F* → follow left path

Be < *Bu* → follow left path

Be > *Ba* → follow right path



Treaps :: Operations :: Insert :: Example

insert(Be, 20)

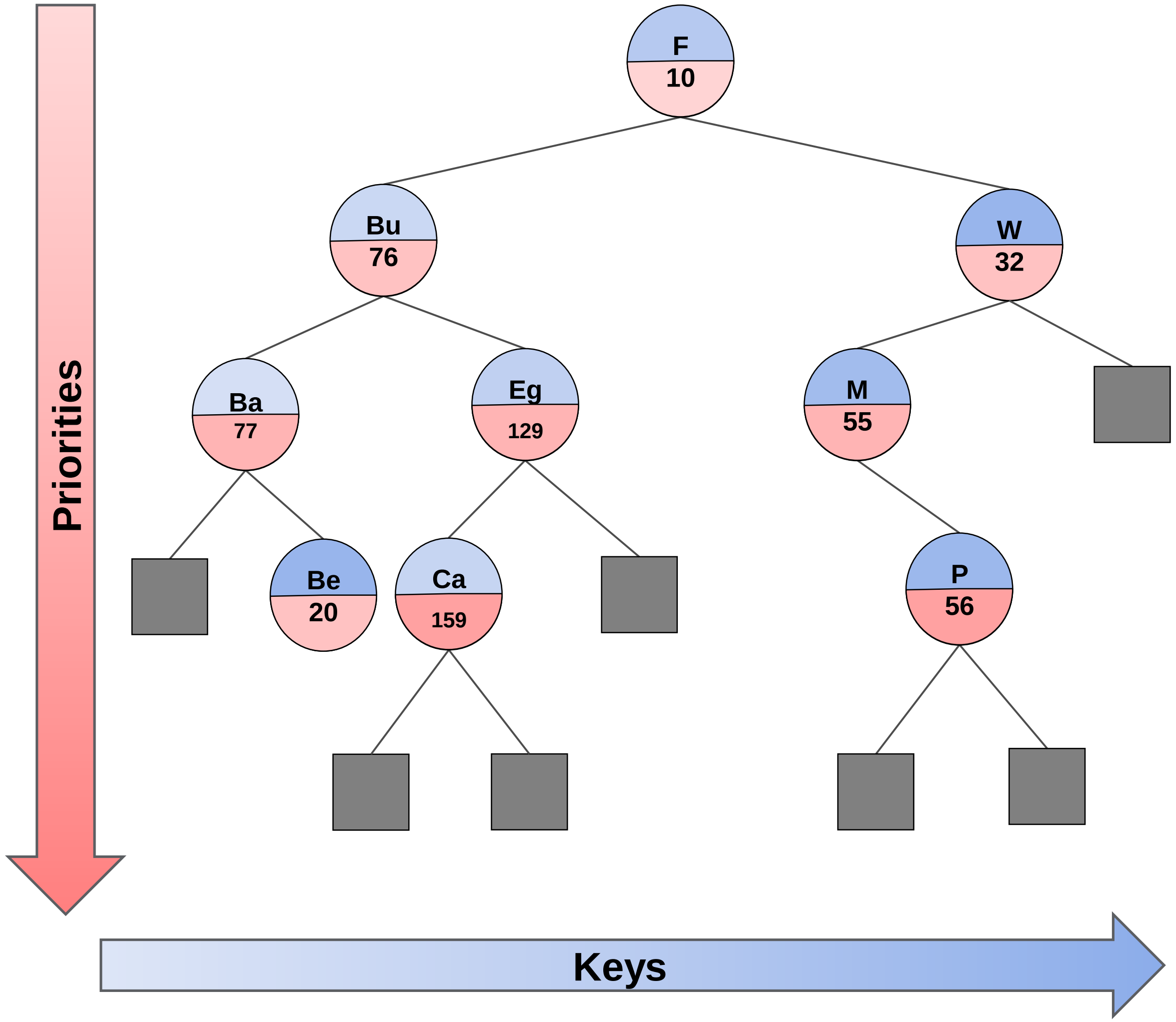
insert according to BST insert

Be < *F* → follow left path

Be < *Bu* → follow left path

Be > *Ba* → follow right path

Insert (*Be, 20)* in free leaf node



Treaps :: Operations :: Insert :: Example

insert(Be, 20)

insert according to BST insert

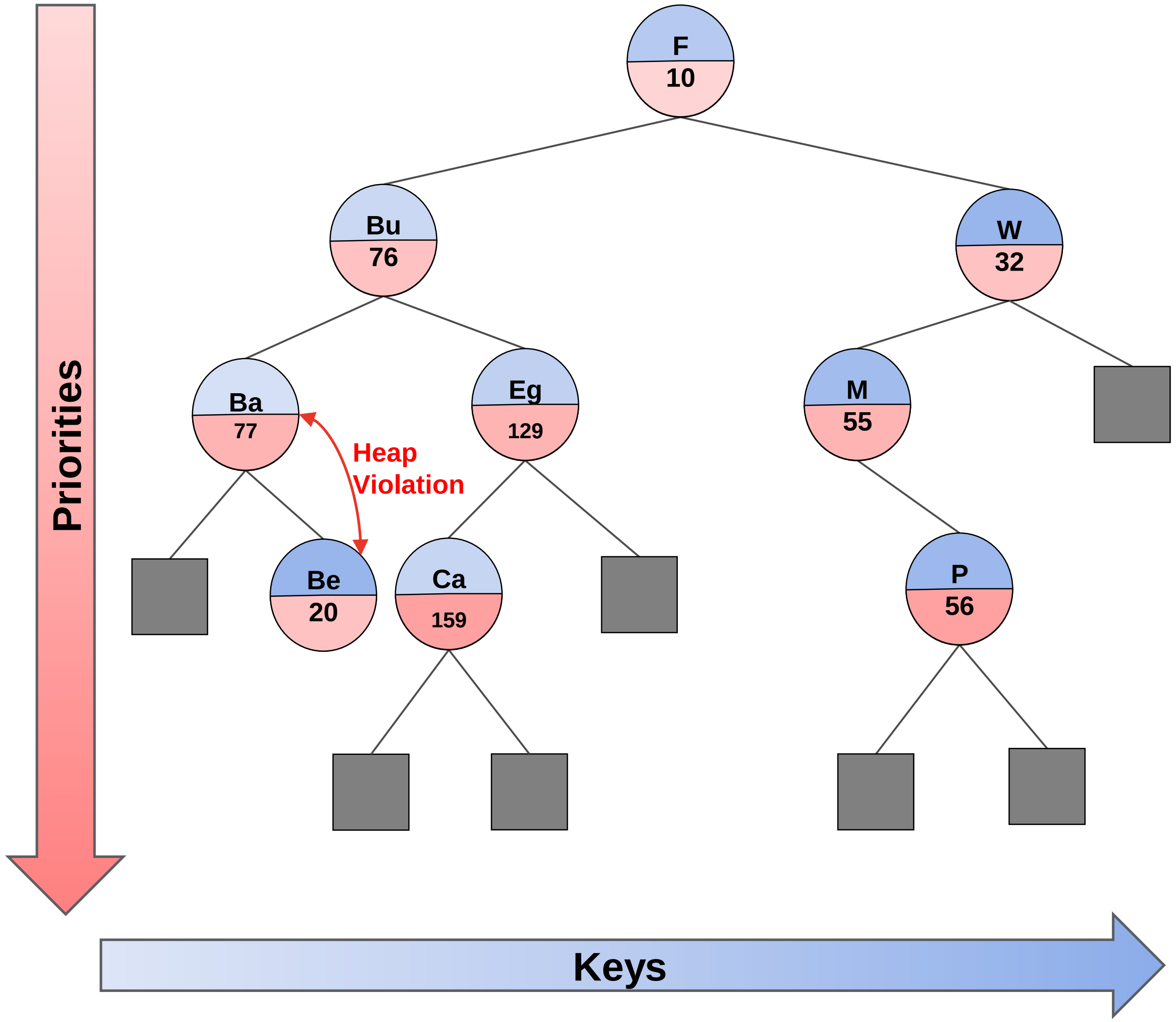
Be < F → follow left path

Be < Bu → follow left path

Be > Ba → follow right path

Insert (*Be, 20)* in free leaf node

Insert position violates heap property

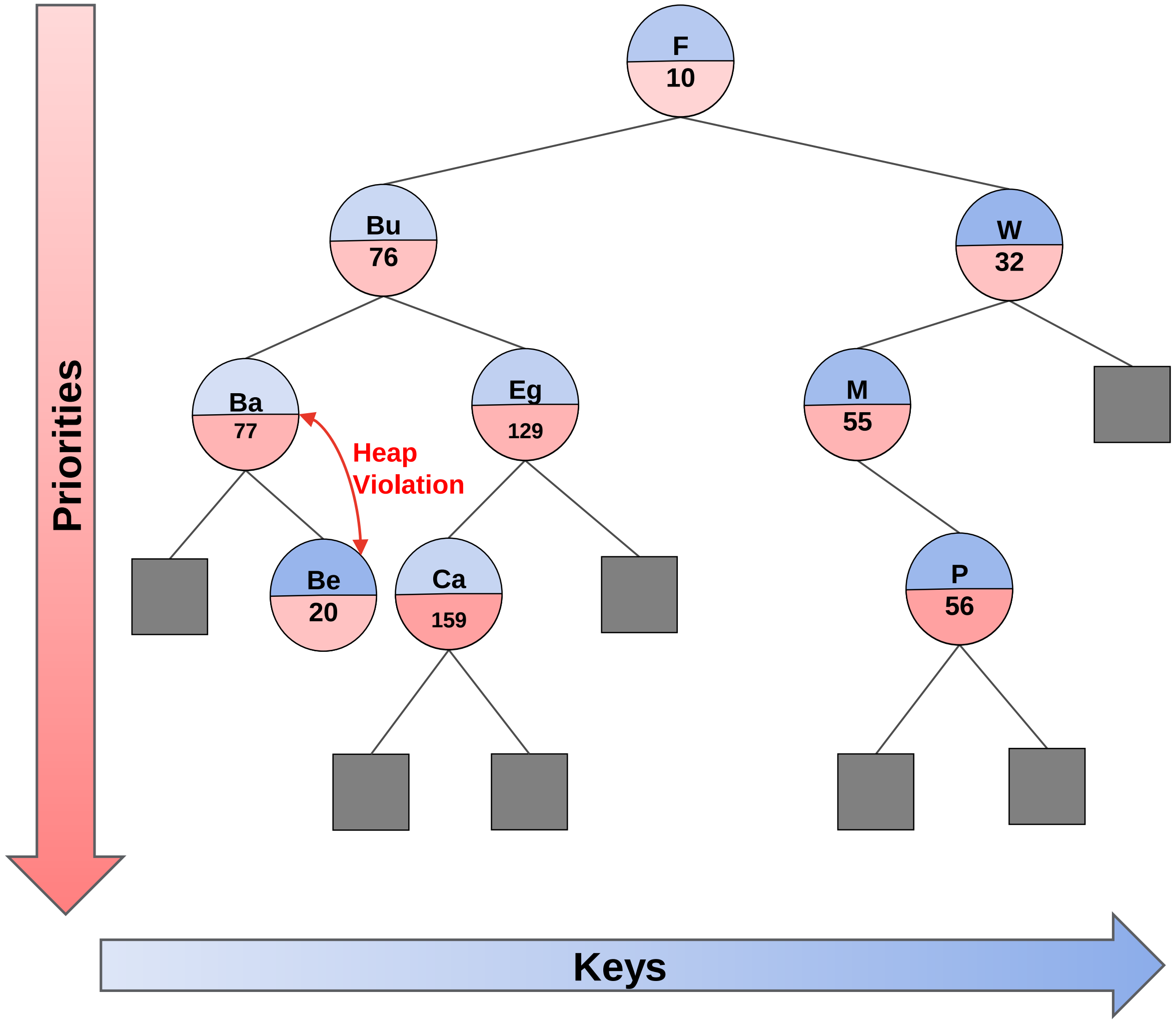


Treaps :: Operations :: Insert :: Example

insert(Be, 20)

Insert position violates heap property

(Be, 20) is right child → rotateLeft



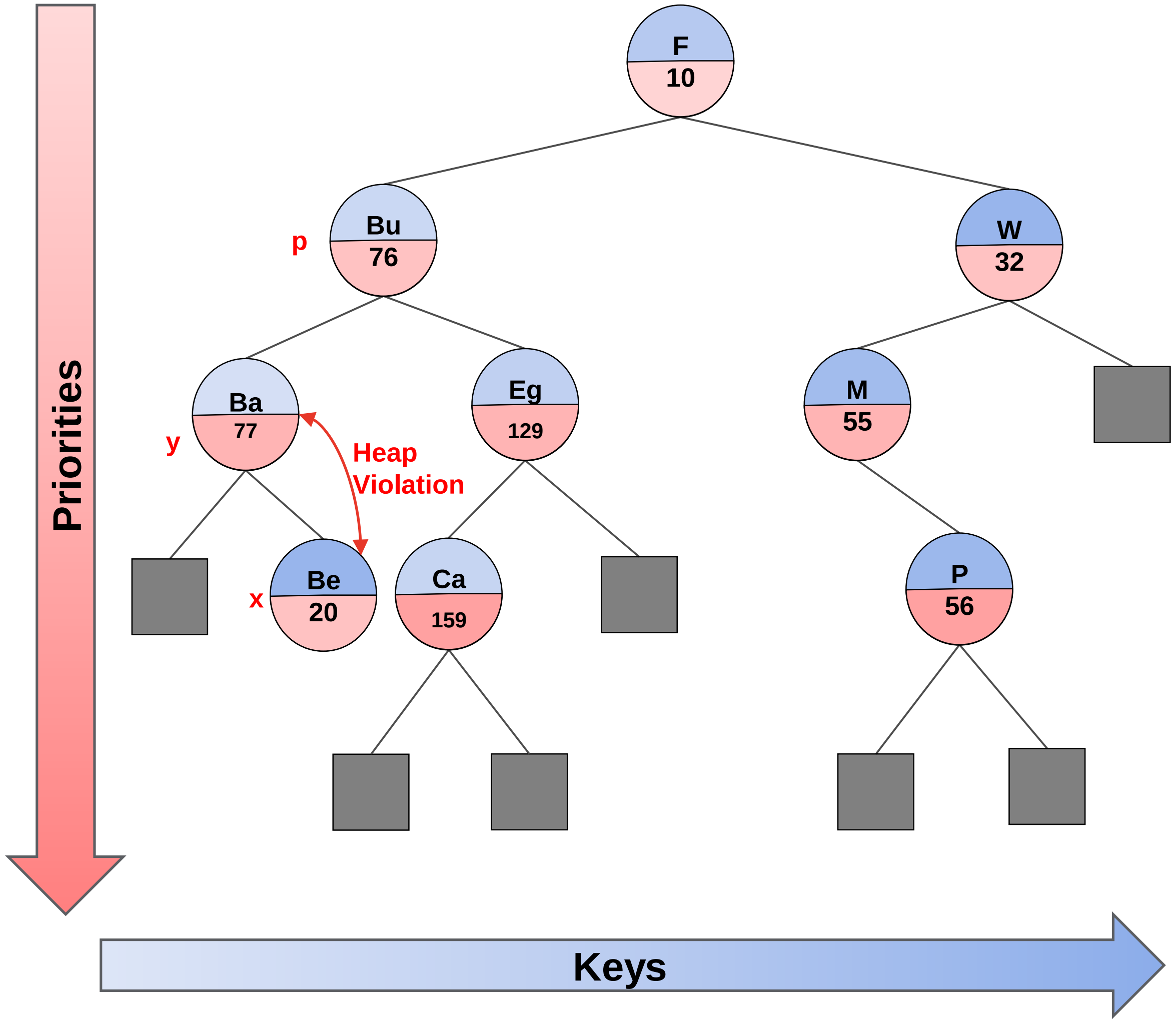
Treaps :: Operations :: Insert :: Example

insert(Be, 20)

Insert position violates heap property

(Be, 20) is right child → rotateLeft

set x = *(Be, 20)*, y = *(Ba, 77)*, p = *(Bu, 76)*



Treaps :: Operations :: Insert :: Example

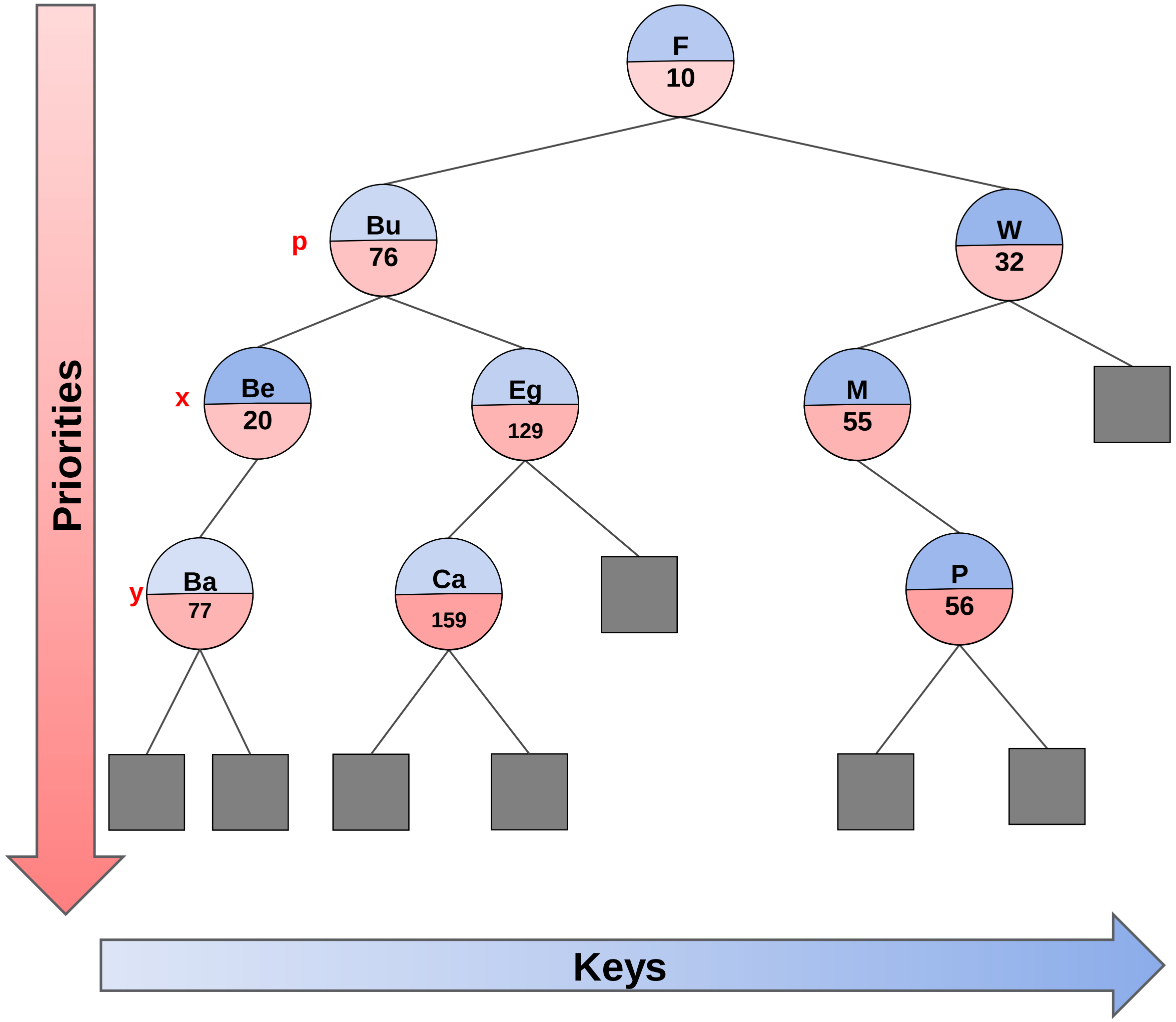
insert(Be, 20)

Insert position violates heap property

(Be, 20) is right child → rotateLeft

set x = *(Be, 20)*, y = *(Ba, 77)*, p = *(Bu, 76)*

set p.left = x, x.left = y, y.right=x.left

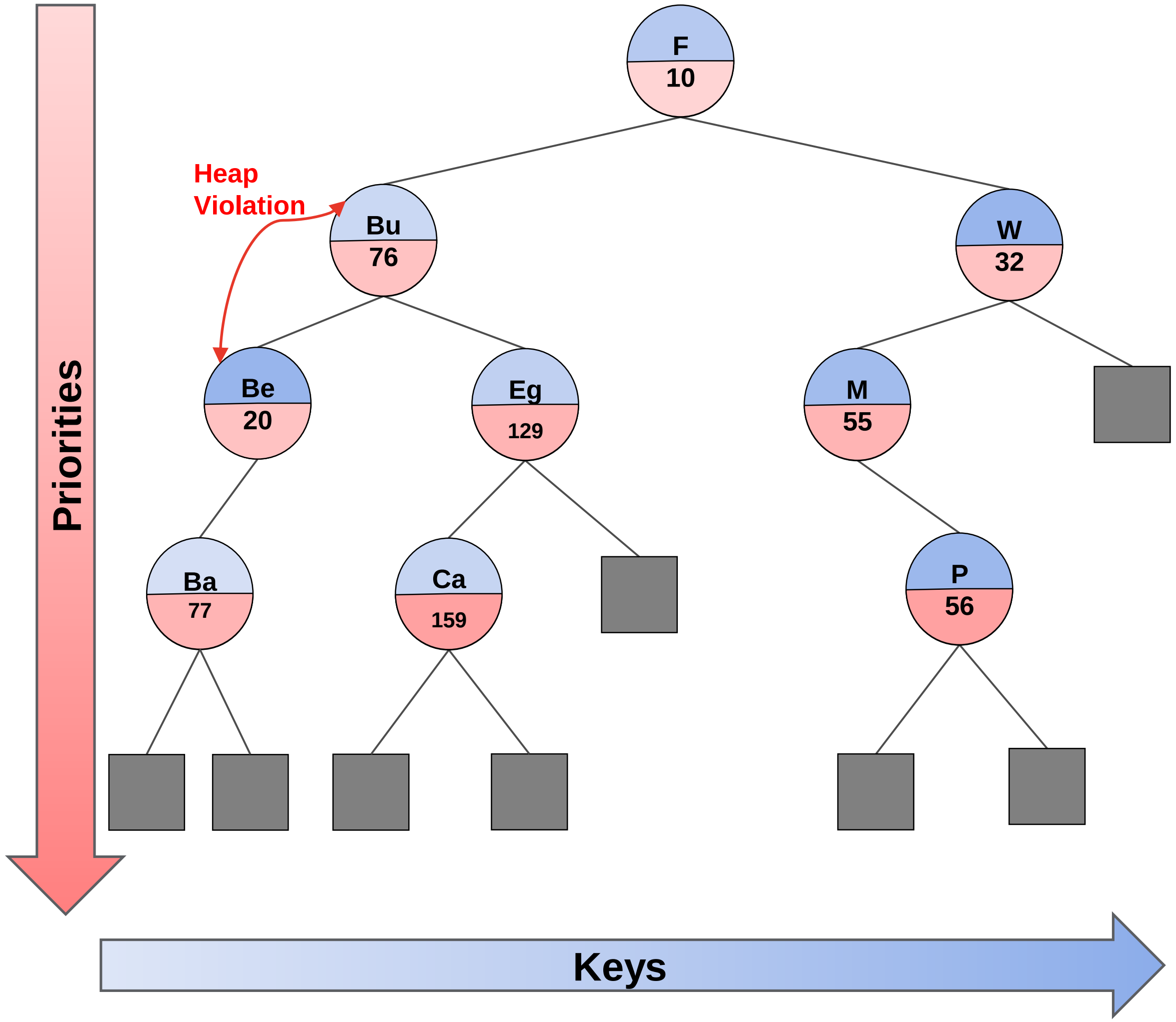


Treaps :: Operations :: Insert :: Example

insert(Be, 20)

After rotate heap property is still violated

(Be, 20) is left child → rotateRight



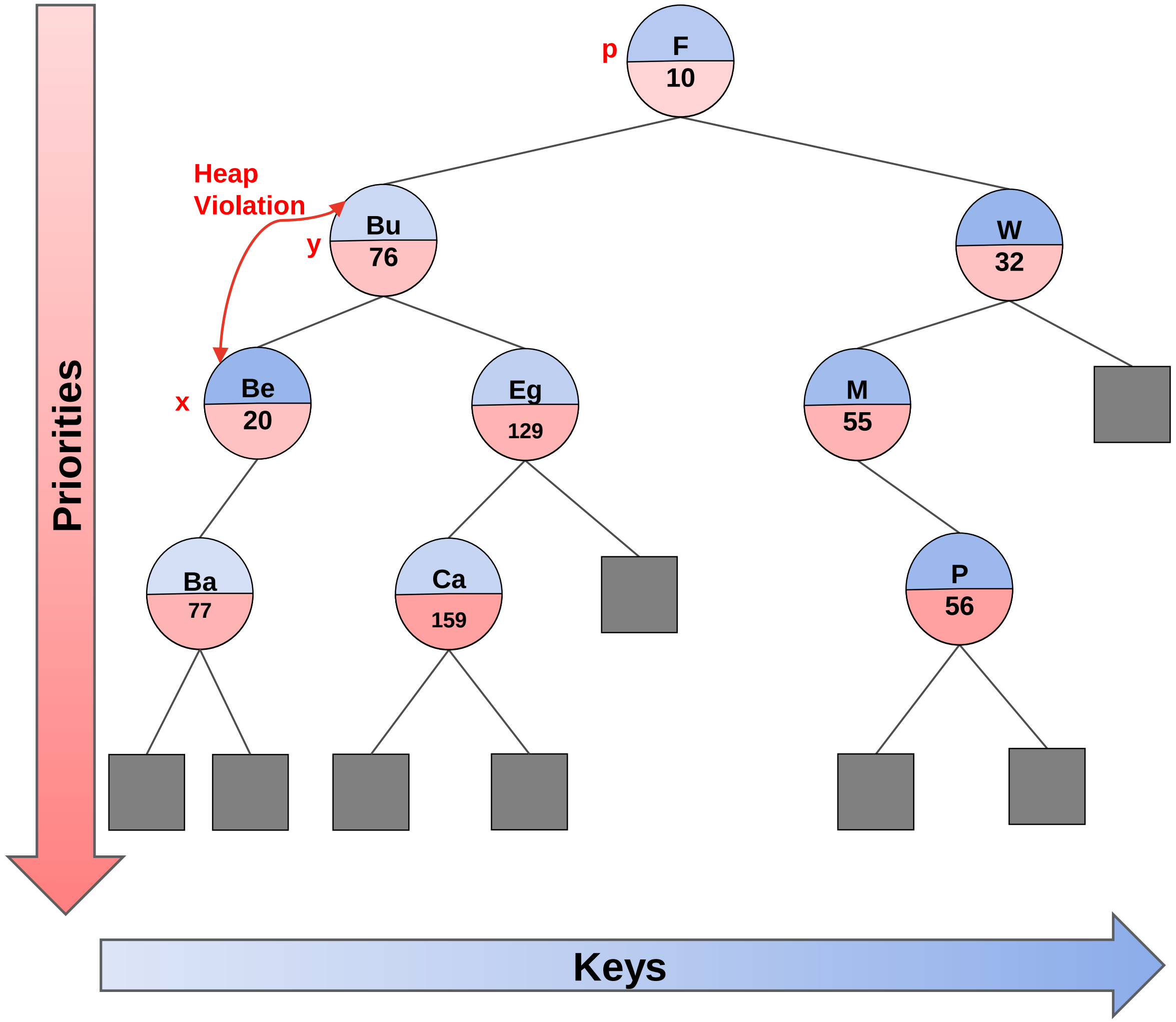
Treaps :: Operations :: Insert :: Example

insert(Be, 20)

After rotate heap property is still violated

(Be, 20) is left child → rotateRight

set x = *(Be, 20)*, y = *(Bu, 76)*, p = *(F, 10)*



Treaps :: Operations :: Insert :: Example

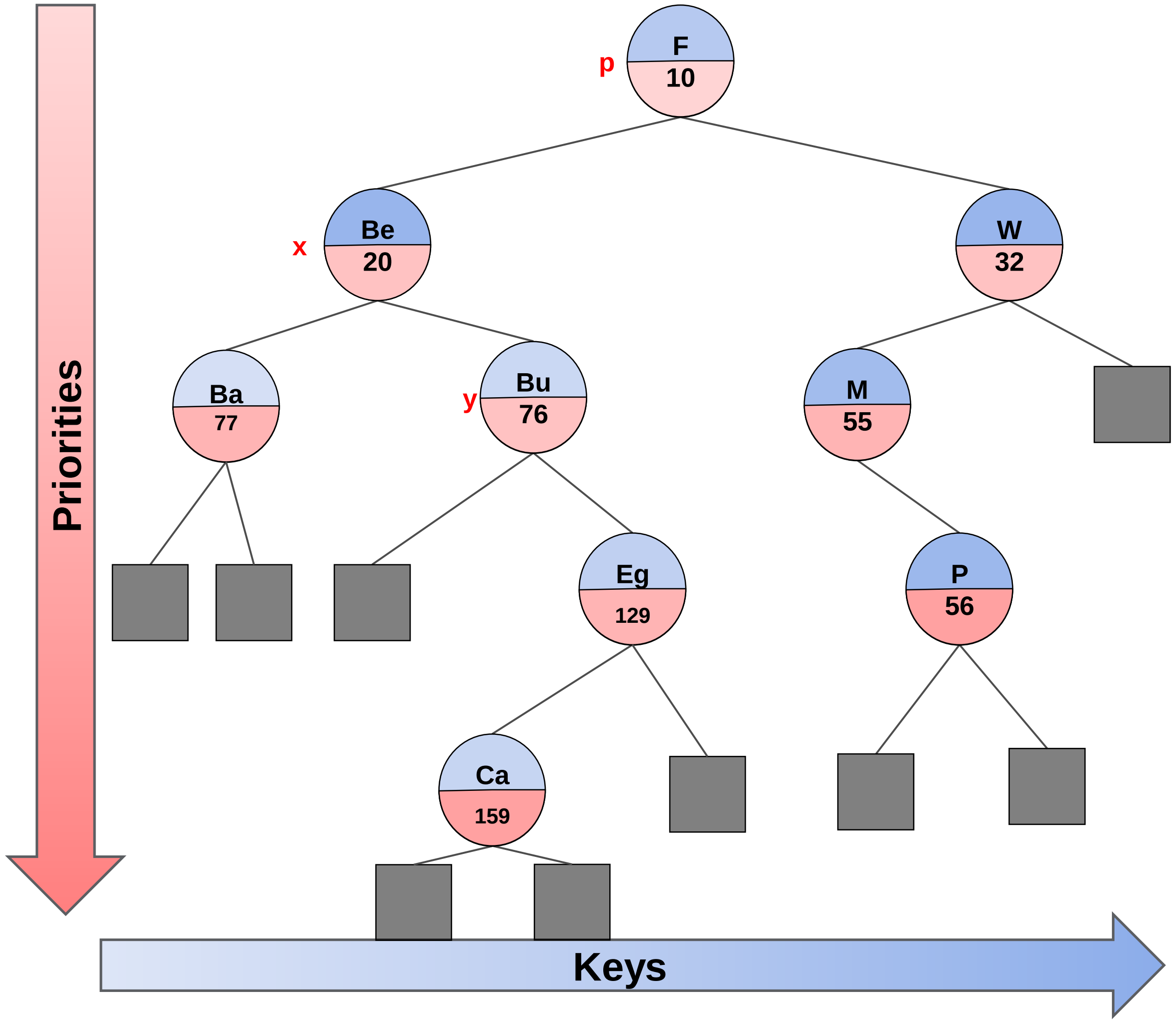
insert(Be, 20)

After rotate heap property is still violated

(Be, 20) is left child → rotateRight

set x = *(Be, 20)*, y = *(Bu, 76)*, p = *(F, 10)*

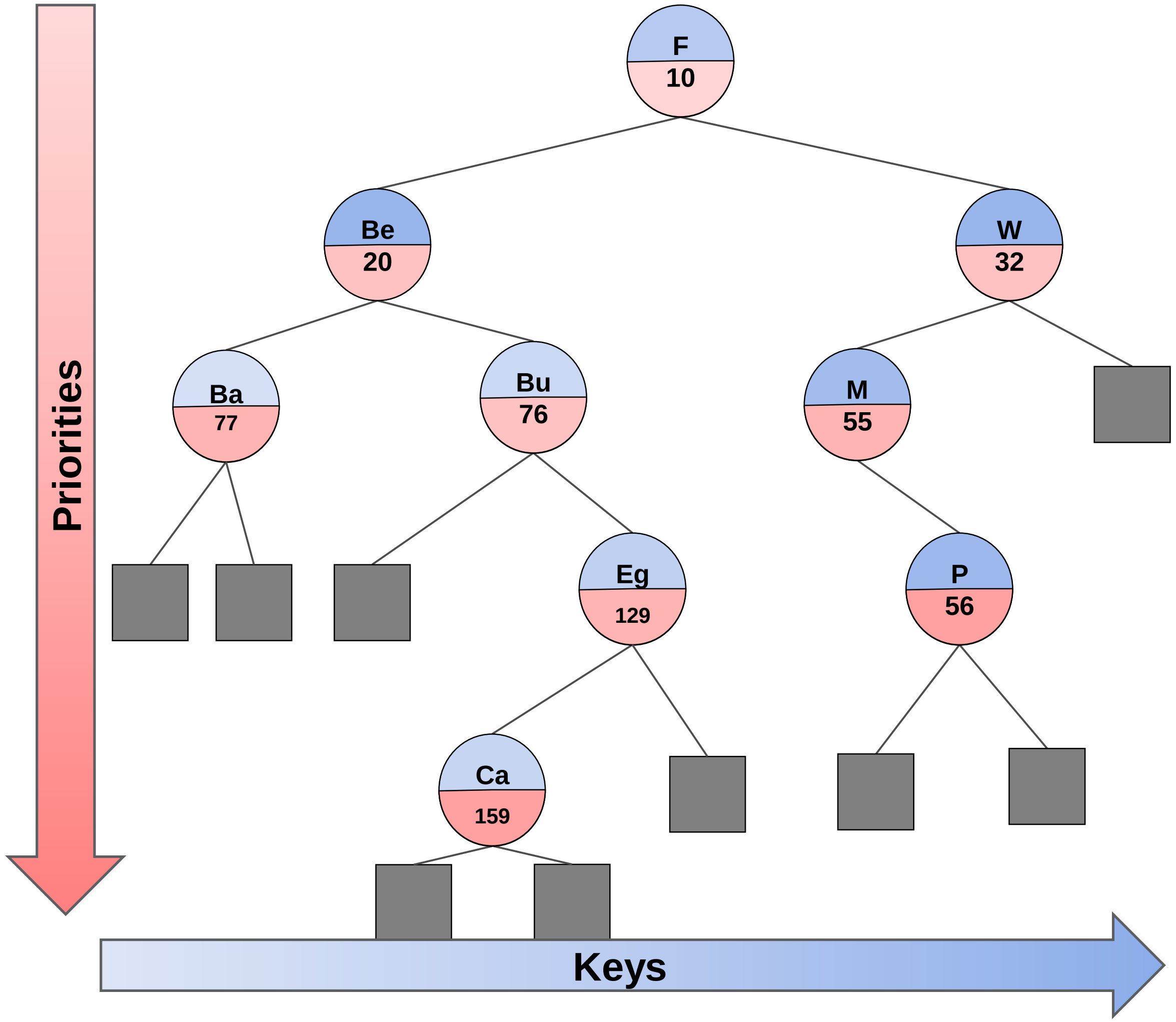
set p.left = x, x.right = y, y.left = x.right



Treaps :: Operations :: Insert :: Example

insert(Be, 20)

BST and heap property are satisfied



Treaps :: Limitations

Treap can be indexed by **keys** and **values**

Advantage of BST operations like **search, insert and remove**

Advantage of a heap to sort by priority and **quick access** to the node with the **highest priority**

But the treap can – just like a BST – degenerate into a linear list with a worst case complexity O(n)

Solution for imbalanced treaps:

Do not use the priority to manually assign importance to nodes but to give a **weight** to nodes **in order to balance**

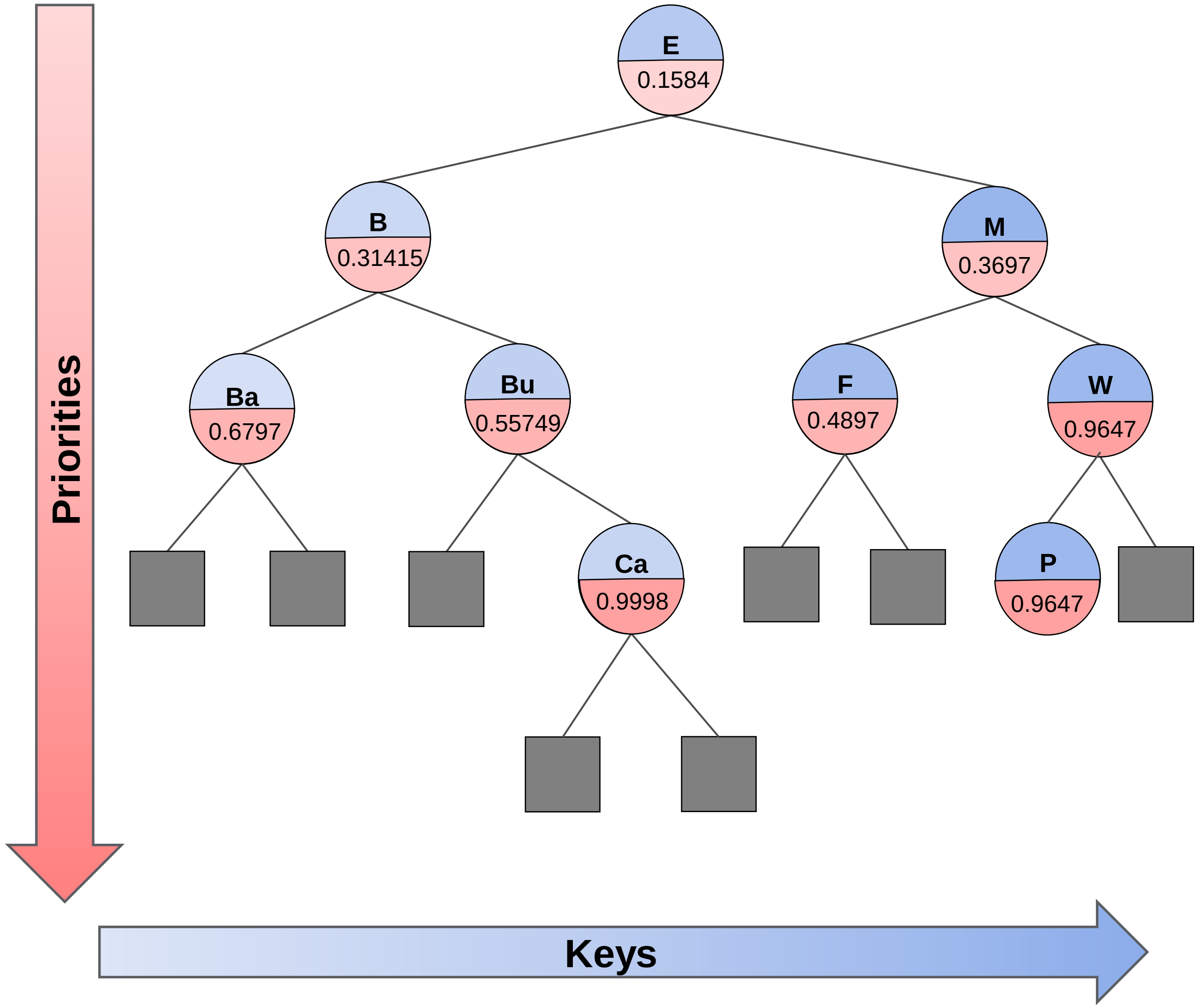
Randomized Treaps

Updating priorities in such a way that the tree is **balanced** is **not efficient**

=> Randomized priorities

Instead assign **random priorities** to each node

If these random priorities are drawn from a **uniform distribution,** then the expected height of the treap is **logarithmic**



Randomized Treaps :: Height

Proof of logarithmic height

First, define an **expected value** for the **random variable** *V* as:

$$E[V] = \sum_{i=1}^{M} v_i * pi$$

where *v* is a finite, countable set $\{v_1 ... v_M\}$ each with a **probability** $\{p_1 ... p_M\}$

Further, define a random variable $\boldsymbol{D_k}$ for the **depth of a given node** N_k as:

$$D_k = \sum_{i=0}^{n-1} N_i \, is \; an \; ancestor \; of \; Nk$$

where the index $k \in \{0,...n\text{-}1\}$ denotes the index of the node's key in the sorted set

In other words: D_k counts **how many ancestors** there are for the node holding the **k-th smallest key**

Randomized Treaps :: Height

Proof of logarithmic height

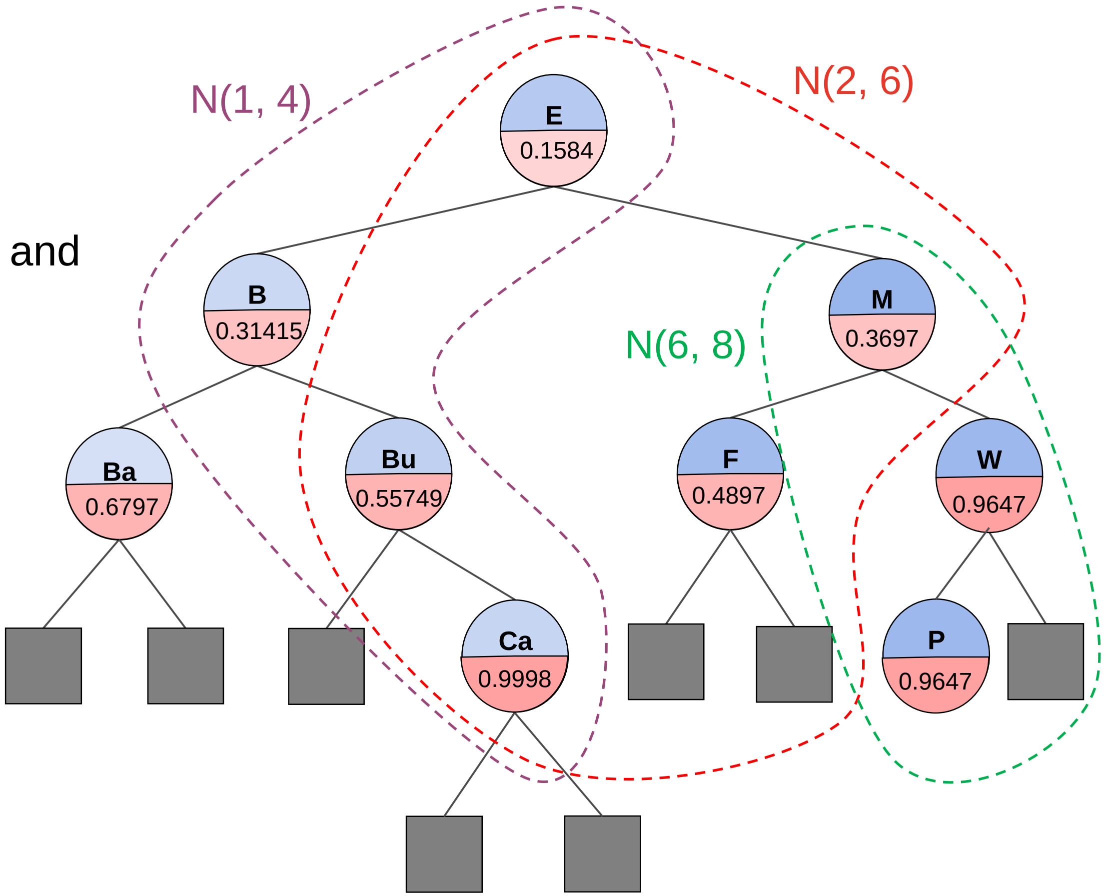
Denote the event '**N_i is an ancestor of N_k**' as the **binary variable** A_k^i and formulate the expected value for D_k as:

$$E[D_k] = \sum_{i=0}^{n-1} P(A_k^i)$$

In order to calculate the **probability** $P(A_k^i)$ define

$$N(i, k) = N(k, i) = \{N_i, N_{i+1}, \ldots N_{k-1}, N_k\}$$

as the **subset of treap nodes** whose keys are **between the i-th and k-th smallest** of the whole tree



Randomized Treaps :: Height

Proof of logarithmic height

It can be proved that the following lemma holds:

For all i ≠ k, 0≤i,k≤n-1, N_i is an ancestor of N_k if and only if N_i has the smallest priority among all nodes in N(i, k)

With this lemma probability that the node with the *i*-th smallest key becomes an ancestor of the node with the *k*-th smallest key can be calculated:

$$P\left(A_k^i\right)_{i \neq k} = \frac{1}{N(i,K)} = \frac{1}{|k-i|+1}$$

Now substitute the probability into the expected value for the depth of a node

$$E[D_k] = \sum_{i=0}^{n-1} P(A_k^i) = \sum_{i=0}^{k-1} \frac{1}{k-i+1} + \sum_{i=k}^{k} 0 + \sum_{i=k+1}^{n-1} \frac{1}{i-k+1}$$

Randomized Treaps :: Height

Proof of logarithmic height

$$E[D_k] = \sum_{i=0}^{n-1} P(A_k^i) = \sum_{i=0}^{k-1} \frac{1}{k-i+1} + \sum_{i=k}^{k} 0 + \sum_{i=k+1}^{n-1} \frac{1}{i-k+1} = \sum_{j=2}^{k-1} \frac{1}{j} + \sum_{j=2}^{n-k} \frac{1}{j} = \sum_{j=1}^{k-1} \frac{1}{j} - 1 + \sum_{j=1}^{n-k} \frac{1}{j} - 1$$

Evaluates to 0

when i = 0 denominator becomes equal to k-1 and diminishes of 1 unit as *i* increases until *i=k-1* *to* become equal to 2

The two summations in the previous formula are both partial sums of the harmonic series and can be reformulated to:

$$E[D_k] = H_{k-1} + H_{n-k} - 1$$

with $H_n < \ln(n)$ the result is:

$$E[D_k] = H_{k-1} + H_{n-k} - 2 < \ln(k-1) + \ln(n-k) - 2 < 2 * \ln(n) - 2$$

which guarantees over a large number of attempts that the mean value of the height is O(log(n))

Institute of
Pervasive Computing

Randomized Treaps

Algorithms and Data Structures 2, 340300
Lecture – 2023W
Univ.-Prof. Dr. Alois Ferscha, teaching@pervasive.jku.at

**JOHANNES KEPLER
UNIVERSITY LINZ**
Altenberger Straße 69
4040 Linz, Austria
jku.at