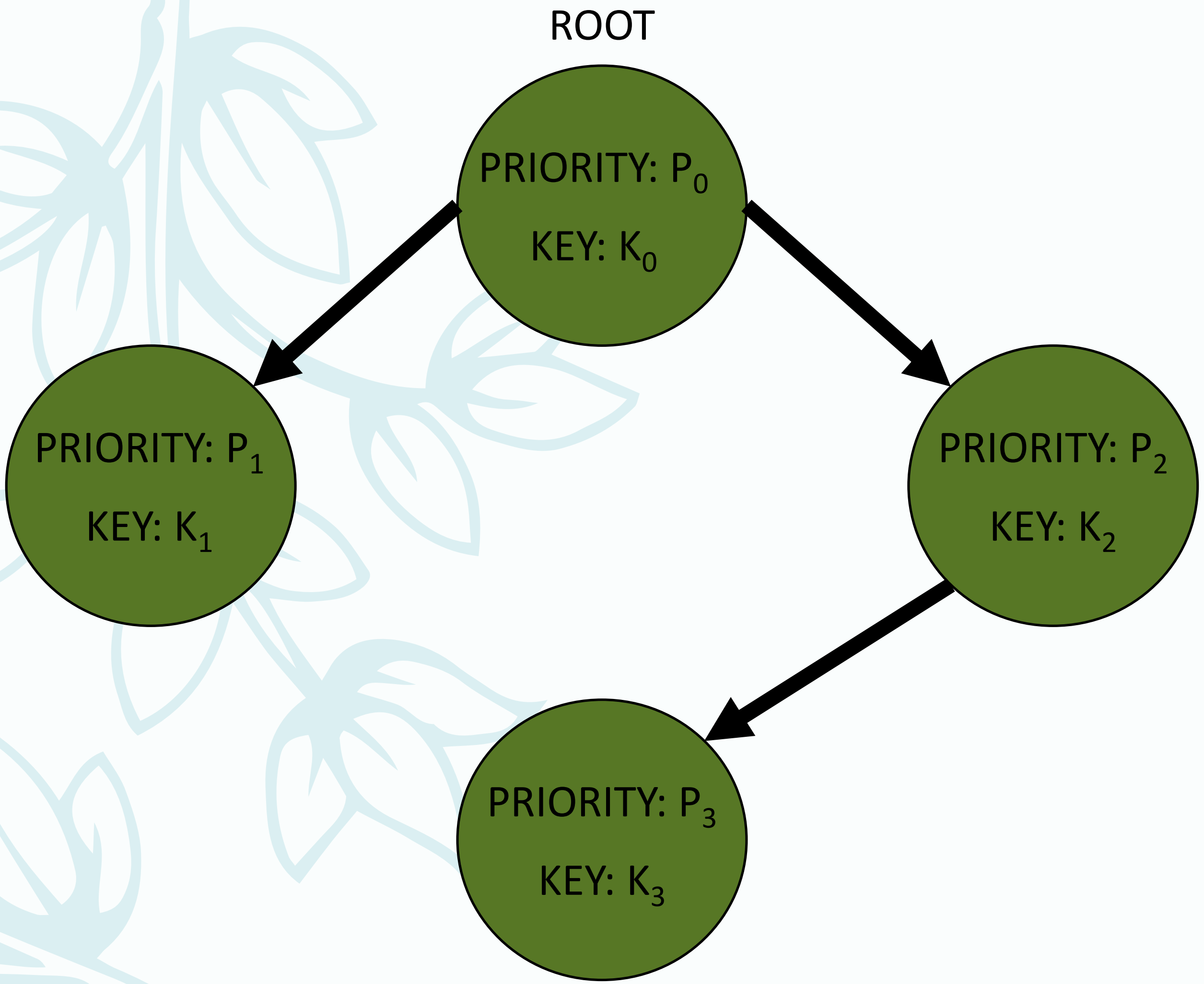
TREAPS

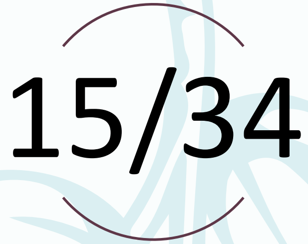
Brenda Escoto
CPSC 406-01

What are treaps?

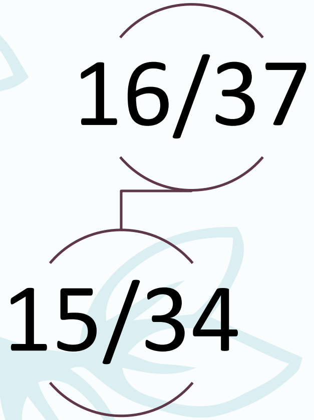


- In terms of priorities:
  - $P_0 > P_1$ & $P_2 > P_3$
- In terms of keys:
  - $K_1 < K_0 < K_3 < K_2$
- Combines the ordered structure of a binary search tree and the rules of a max heap (greatest value on top)

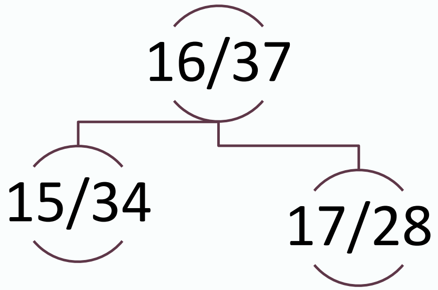
```
        **ROOT**
CURRENT NODE KEY: 15
CURRENT NODE PRIORITY: 34
```



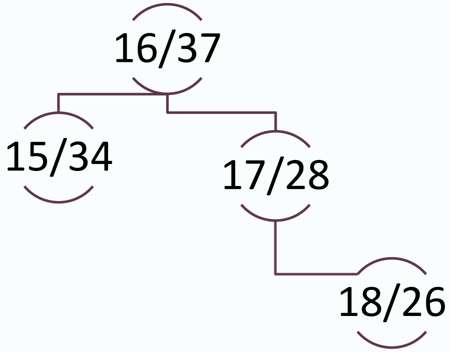

```
CURRENT NODE KEY: 15
CURRENT NODE PRIORITY: 34
PREVIOUS NODE: 16

        **ROOT**
CURRENT NODE KEY: 16
CURRENT NODE PRIORITY: 37
LEFT CHILD: 15
```



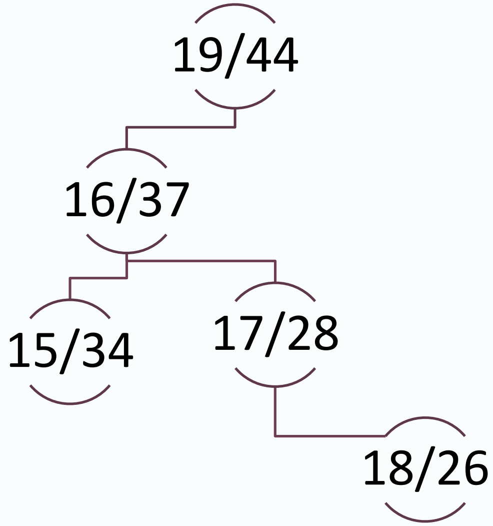

```
--
CURRENT NODE KEY: 15
CURRENT NODE PRIORITY: 34
PREVIOUS NODE: 16

        **ROOT**
CURRENT NODE KEY: 16
CURRENT NODE PRIORITY: 37
LEFT CHILD: 15
RIGHT CHILD: 17

CURRENT NODE KEY: 17
CURRENT NODE PRIORITY: 28
PREVIOUS NODE: 16
```




```
CURRENT NODE KEY: 15
CURRENT NODE PRIORITY: 34
PREVIOUS NODE: 16

        **ROOT**
CURRENT NODE KEY: 16
CURRENT NODE PRIORITY: 37
LEFT CHILD: 15
RIGHT CHILD: 17

CURRENT NODE KEY: 17
CURRENT NODE PRIORITY: 28
PREVIOUS NODE: 16
RIGHT CHILD: 18

CURRENT NODE KEY: 18
CURRENT NODE PRIORITY: 16
PREVIOUS NODE: 17
```




```
CURRENT NODE KEY: 15
CURRENT NODE PRIORITY: 34
PREVIOUS NODE: 16

CURRENT NODE KEY: 16
CURRENT NODE PRIORITY: 37
PREVIOUS NODE: 19
LEFT CHILD: 15
RIGHT CHILD: 17

CURRENT NODE KEY: 17
CURRENT NODE PRIORITY: 28
PREVIOUS NODE: 16
RIGHT CHILD: 18

CURRENT NODE KEY: 18
CURRENT NODE PRIORITY: 16
PREVIOUS NODE: 17

        **ROOT**
CURRENT NODE KEY: 19
CURRENT NODE PRIORITY: 44
LEFT CHILD: 16
```



Uses for treaps

- In a regular BST, the root is a fixed value, so there is the possibility of degradation with time, and the tree essentially becoming a linked list
- With treaps, there is always a possibility that a new node could become the new root, or that a new node could cause a rotation in the tree, since priorities are random
- Keeps the structure balanced

History

Abstract

We present a randomized strategy for maintaining balance in dynamically changing search trees that has optimal *expected* behavior. In particular, in the expected case a search or an update takes logarithmic time, with the update requiring fewer than two rotations. Moreover, the update time remains logarithmic, even if the cost of a rotation is taken to be proportional to the size of the rotated subtree. Finger searches and splits and joins can be performed in optimal expected time also. We show that these results continue to hold even if very little true randomness is available, i.e. if only a logarithmic number of truely random bits are available. Our approach generalizes naturally to weighted trees, where the expected time bounds for accesses and updates again match the worst case time bounds of the best deterministic methods.

We also discuss ways of implementing our randomized strategy so that no explicit balance information is maintained. Our balancing strategy and our algorithms are exceedingly simple and should be fast in practice.

- 🌲 Treaps were first proposed in Raimund Seidel and Cecilia R. Aragon's paper, "Randomized Search Trees" (1989)
- 🌲 Implementations vary based on programming language, basic functions are outlined in the paper
  - 🌲 Includes insert, delete, search, rotations, ect.

Implementation in C++

Each node has five variables:

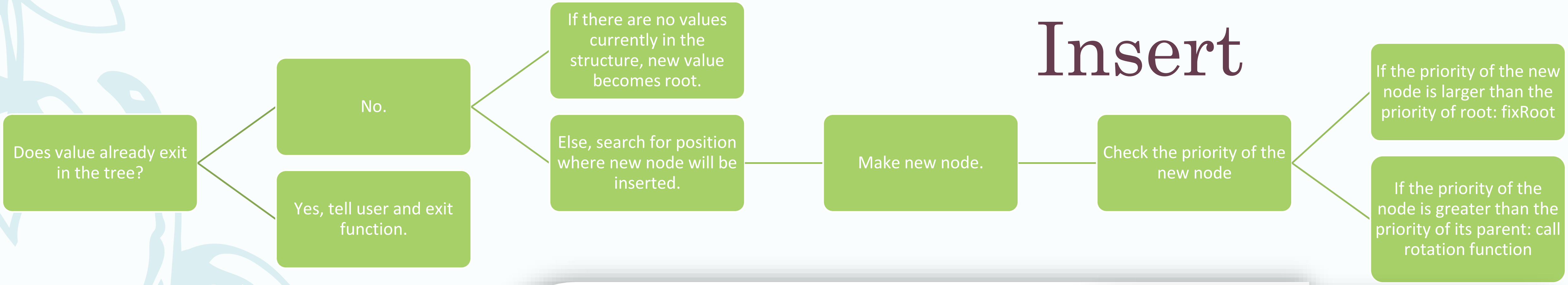
- Its **key** value, determined by the user
- Its **priority**, randomly determined when a new node is made
- A pointer to its **left** child and **right** child, if they exist
- A pointer to its **previous** node, the parent

```
struct Node
{
  int key, priority;
  Node *left;
  Node *right;
  Node *prev;
};
```

- The insert function calls the newNode helper function, passing along the value of the key
- For a treap that only accepts 50 values:

```
Node* Treap::newNode(int k)
{
  Node *tempNode = new Node;
  int p;
  tempNode->key=k;

  while(true)
  {
    p=rand()%50+1;//1-50
    if(priorities[p-1]==false)//find a priority that's empty
    {
      priorities[p-1]=true;
      break;
    }
  }

  tempNode->priority=p;
  tempNode->left=NULL;
  tempNode->right=NULL;

  return tempNode;
}
```

Insert




```
Node* Treap::insert(int k, Node* curr)
{
  if(curr==NULL)
  {
    curr = newNode(k);
    if(root==NULL)
    {
      root=curr;
    }
    recent=curr;
  }

  if(k < curr->key)//k smaller than current key
  {
```

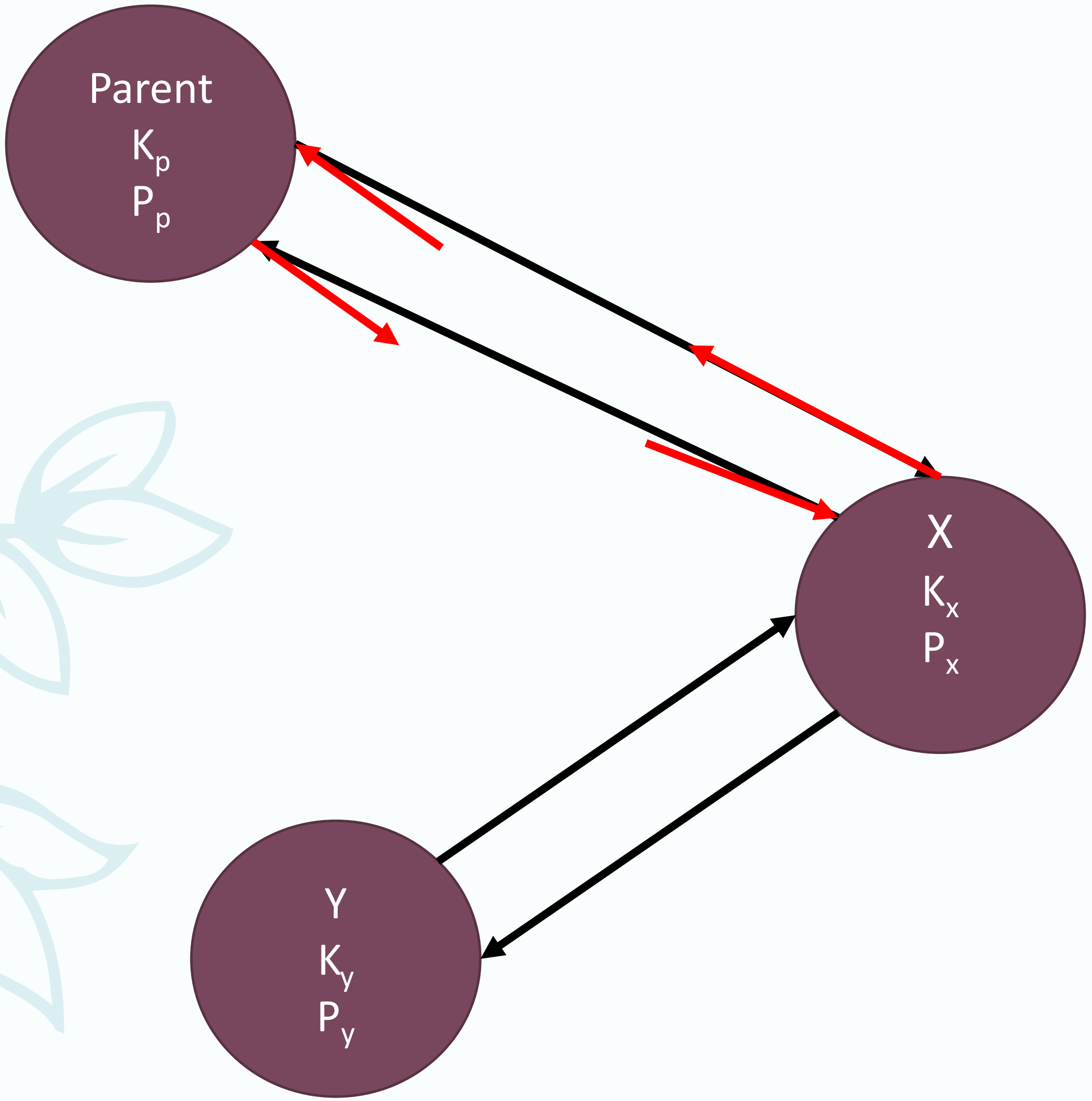
```
oid Treap::fixRoot(Node* curr)
{

  Node* temp=root;

  //changing pointers to new root so that we don't
  if(curr->prev->left==curr)
  {
    curr->prev->left=NULL;
  }
  if(curr->prev->right==curr)
  {
```

```
if(temp->priority > root->priority)
{

  fixRoot(temp);
  return;
}

//not root and rotation needed
if(temp->priority > temp->prev->priority)
{
  //cout << "mess detected" << endl;
  leafRotation(temp);
}
```

```
void Treap::leafRotation(Node* y)
{
  Node *x = y->prev;
  Node* parent = x->prev;
```

procedure TREAP-INSERT((k,p) : item, T : treap)
  if $T = tnull$ **then** $T \leftarrow$ NEWNODE()
      $T \rightarrow [key, priority, lchild, rchild] \leftarrow [k, p, tnull, tnull]$
  else if $k < T \rightarrow key$ **then** TREAP-INSERT((k,p) , $T \rightarrow lchild$)
      if $T \rightarrow lchild \rightarrow priority > T \rightarrow priority$ **then** ROTATE-RIGHT(T)
  else if $k > T \rightarrow key$ **then** TREAP-INSERT((k,p) , $T \rightarrow rchild$)
      if $T \rightarrow rchild \rightarrow priority > T \rightarrow priority$ **then** ROTATE-LEFT(T)
  else (* key k already in treap T *)

Endless Rotations!

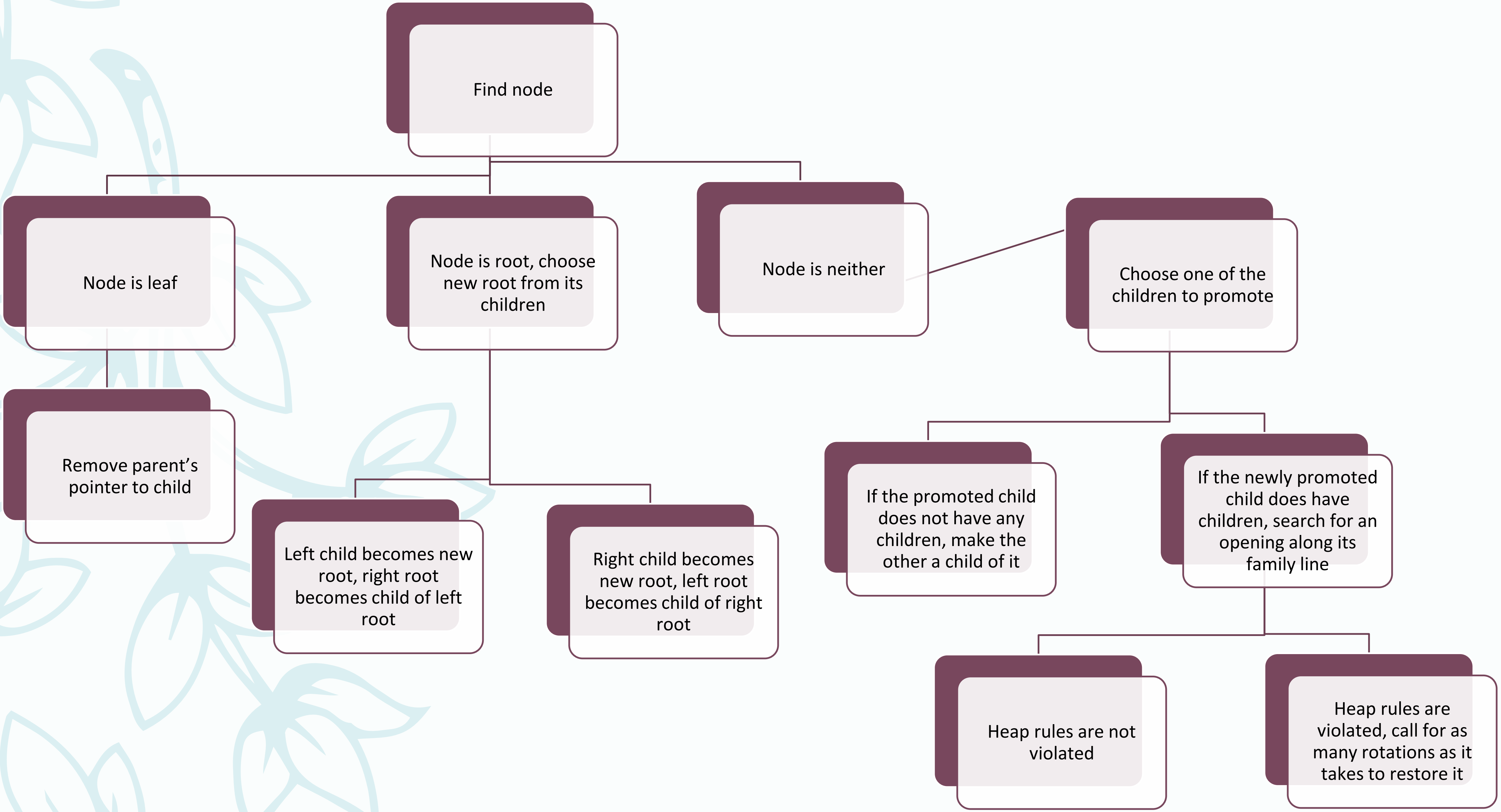


Problem:

$P_p > P_y > P_x$

$K_p < K_y < K_x$

- Best case scenario: y is a leaf and x doesn't have any other children besides y
- Rotation functions leafRotation & loadedRotation are adirectional


Find node
Node is leaf
Node is root, choose new root from its children
Node is neither
Choose one of the children to promote
Remove parent's pointer to child
Left child becomes new root, right root becomes child of left root
Right child becomes new root, left root becomes child of right root
If the promoted child does not have any children, make the other a child of it
If the newly promoted child does have children, search for an opening along its family line
Heap rules are not violated
Heap rules are violated, call for as many rotations as it takes to restore it

Remove

Other functions

search

```
bool Treap::search(int k, Node* curr)
{
  if(curr==NULL)//since root is passed first, this means that there's
  nothing in the treap
  {
    return false;
  }
  if(curr->key==k)//found it
  {
    return true;
  }
  else //search next node
  {
    if(curr->key > k)
    {
      search(k, curr->left);
    }
    else if(curr->key < k)
    {
      search(k, curr->right);
    }
  }

}
```

- Returns true if value is in treap
- Insert & remove call search before running to ensure no extra work is done

Print

- Provides all information about a node and all the nodes in the treap
- Prints in order

```
Treap s;

s.insert(3);
s.insert(4);
s.insert(2);
s.printTreap();
```

```
CURRENT NODE KEY: 2
CURRENT NODE PRIORITY: 28
PREVIOUS NODE: 3

CURRENT NODE KEY: 3
CURRENT NODE PRIORITY: 34
PREVIOUS NODE: 4
LEFT CHILD: 2

        **ROOT**
CURRENT NODE KEY: 4
CURRENT NODE PRIORITY: 37
LEFT CHILD: 3
```

Runtimes for functions

Theorem 3.1 *A randomized search tree storing n items has the expected performance character istics listed in the table below:*

Performance measure	**Bound on expectation**
access time	$O(\log n)$
insertion time	$O(\log n)$
*insertion time for element with handle on predecessor or successor	$O(1)$
deletion time	$O(\log n)$
*deletion time for element with handle	$O(1)$

- Search is dependent on the number of ancestors the searched value has, and if it does not exist, then it is equal to the depth of the treap
- Insert requires a search and then as many rotations as it takes to ensure the heap rule is not violated
- Delete is the inverse of insert, at best no rotations are required, at worse several rotations could be required
- These runtimes can be faster without having to start at the root node each time, having a “handle” on the parent node where the new child will be inserted/deleted
  - In a treap where it costs the same to travel from one node to another, runtime is constant