# CS469 Assignment 4

Tianye Zhao, Wenda Yang

April 20, 2019

## 1

### 1.1

BinaryTree(T info):
  $newNode = createNode($info$)$
  $root = newNode$

### 1.2

∼BinaryTree():
  $clear()$

### 1.3

T& top():
  **if** $root \neq$ NULL **then**
    **return** $root \rightarrow value$
  **end if**

### 1.4

T pop_front():
  $tmpNode = root$
  $tmpVal = root \rightarrow value$
  $nodePtr = root \rightarrow left$
  **while** $nodePtr \rightarrow right \neq$ NULL **do**
    $nodePtr = nodePtr \rightarrow right$
  **end while**
  $nodePtr \rightarrow right = root \rightarrow right \rightarrow left$
  $root \rightarrow right \rightarrow left = root \rightarrow left$
  $root = root \rightarrow right$
  delete $tmpNode$
  **return** $tmpVal$

## 1.5

bool empty():

**return** $root$ == NULL

## 1.6

void insertNode(T info):

$newNode = createNode($info$)$
**if** $root ==$ NULL **then**
$root = newNode$
**return**
**end if**
$nodePtr = root$
**while** $nodePtr \neq$ NULL **do**
$parent = nodePtr$
**if** $nodePtr \rightarrow value >$ info **then**
$nodePtr = nodePtr \rightarrow left$
**else**
$nodePtr = nodePtr \rightarrow right$
**end if**
**end while**
**if** $parent \rightarrow value >$ info **then**
$parent \rightarrow left = newNode$
**else**
$parent \rightarrow right = newNode$
**end if**

## 1.7

void deleteNode(T info, bool removeAll):

**if** $root ==$ NULL **then**
**return**
**end if**
**if** $root \rightarrow value ==$ info **then**
**if** $root \rightarrow left \neq$ NULL **and** $root \rightarrow right \neq$ NULL **then**
$tmpNode = root$
$nodePtr = root \rightarrow left$
**while** $nodePtr \rightarrow right \neq$ NULL **do**
$nodePtr = nodePtr \rightarrow right$
**end while**
$nodePtr \rightarrow right = root \rightarrow right \rightarrow left$
$root \rightarrow right \rightarrow left = root \rightarrow left$
$root = root \rightarrow right$
delete $tmpNode$
**else if** $root \rightarrow left ==$ NULL **and** $root \rightarrow right ==$ NULL **then**
delete $root$
**else if** $root \rightarrow left \neq$ NULL **and** $root \rightarrow right ==$ NULL **then**

$tmpNode = root$
$root = root \rightarrow left$
delete $tmpNode$
**else**
$tmpNode = root$
$root = root \rightarrow right$
delete $tmpNode$
**end if**
**if** removeAll $==$ **false then**
**return**
**end if**
**else if** $root \rightarrow value >$ info **then**
Create subtree $L$ rooted on $root \rightarrow left$
$L.deleteNode($info, removeAll$)$
**else**
Create subtree $R$ rooted on $root \rightarrow right$
$R.deleteNode($info, removeAll$)$
**end if**

## 1.8

void preOrderTraversal():
**if** $root ==$ NULL **then**
**return**
**else**
Create subtree $L$ rooted on $root \rightarrow left$
Create subtree $R$ rooted on $root \rightarrow right$
**print** $root \rightarrow value$
$L.preOrderTraversal()$
$R.preOrderTraversal()$
**end if**

## 1.9

void inOrderTraversal():
**if** $root ==$ NULL **then**
**return**
**else**
Create subtree $L$ rooted on $root \rightarrow left$
Create subtree $R$ rooted on $root \rightarrow right$
$L.inOrderTraversal()$
**print** $root \rightarrow value$
$R.inOrderTraversal()$
**end if**

## 1.10

void postOrderTraversal():

**if** $root ==$ NULL **then**
  **return**
**else**
  Create subtree $L$ rooted on $root \rightarrow left$
  Create subtree $R$ rooted on $root \rightarrow right$
  $L.postOrderTraversal()$
  $R.postOrderTraversal()$
  **print** $root \rightarrow value$
**end if**

## 1.11

int countLesserThan(T val):
  **if** $root ==$ NULL **then**
    **return** $0$
  **else if** $root \rightarrow value <$ info **then**
    Create subtree $L$ rooted on $root \rightarrow left$
    Create subtree $R$ rooted on $root \rightarrow right$
    **return** $1 + L.countLesserThan($val$) + R.countLesserThan($val$)$
  **else**
    **return** $L.countLesserThan($val$) + R.countLesserThan($val$)$
  **end if**

## 1.12

int countGreaterThan(T val):
  **if** $root ==$ NULL **then**
    **return** $0$
  **else if** $root \rightarrow value >$ info **then**
    Create subtree $L$ rooted on $root \rightarrow left$
    Create subtree $R$ rooted on $root \rightarrow right$
    **return** $1 + L.countGreaterThan($val$) + R.countGreaterThan($val$)$
  **else**
    **return** $L.countGreaterThan($val$) + R.countGreaterThan($val$)$
  **end if**

## 1.13

int length():
  **if** $root ==$ NULL **then**
    **return** $0$
  **else**
    Create subtree $L$ rooted on $root \rightarrow left$
    Create subtree $R$ rooted on $root \rightarrow right$
    **return** $1 + L.length() + R.length()$
  **end if**

## 1.14

int height():

  **if** $root ==$ NULL **then**
    **return** $0$
  **else**
    Create subtree $L$ rooted on $root \rightarrow left$
    Create subtree $R$ rooted on $root \rightarrow right$
    **return** $1 + max(L.height(), R.height())$
  **end if**

## 1.15

void clear():

  **if** $root \neq$ NULL **then**
    Create subtree $L$ rooted on $root \rightarrow left$
    Create subtree $R$ rooted on $root \rightarrow right$
    $L.clear()$
    $R.clear()$
    delete $root$
  **end if**

## 1.16

void mirror():

  **if** $root \neq$ NULL **then**
    $tmpNode = root \rightarrow left$
    $root \rightarrow left = root \rightarrow right$
    $root \rightarrow right = tmpNode$
    Create subtree $L$ rooted on $root \rightarrow left$
    Create subtree $R$ rooted on $root \rightarrow right$
    $L.mirror()$
    $R.mirror()$
  **end if**

## 1.17

bool isIdenticalTo(BinaryTree B):

  **if** $root ==$ NULL **and** $B.empty()$ **then**
    **return true**
  **else if** $root \neq$ NULL **and not** $B.empty()$ **then**
    Create subtree $L1$ rooted on $root \rightarrow left$
    Create subtree $R1$ rooted on $root \rightarrow right$
    Create subtree $L2$ rooted on $B.getRoot() \rightarrow left$
    Create subtree $R2$ rooted on $B.getRoot() \rightarrow right$
    **return** $root \rightarrow value == B.top()$ **and** $L1.isIdenticalTo(L2)$ **and** $R1.isIdenticalTo(R2)$
  **else**
    **return false**

**end if**

## 1.18

bool isIsomorphicWith(BinaryTree B):

$B.mirror()$
**return** $isIdenticalTo(B)$

## 1.19

int countLeafNodes():

**if** $root ==$ NULL **then**
**return** $0$
**end if**
**if** $root \rightarrow left ==$ NULL **and** $root \rightarrow right ==$ NULL **then**
**return** $1$
**else**
Create subtree $L$ rooted on $root \rightarrow left$
Create subtree $R$ rooted on $root \rightarrow right$
**return** $L.countLeafNodes() + R.countLeafNodes()$
**end if**

## 1.20

int countSemiLeafNodes():

**if** $root ==$ NULL **then**
**return** $0$
**end if**
**if** $root \rightarrow left \neq$ NULL **and** $root \rightarrow right ==$ NULL **or** $root \rightarrow left ==$ NULL **and** $root \rightarrow right \neq$ NULL **then**
**return** $1$
**else**
Create subtree $L$ rooted on $root \rightarrow left$
Create subtree $R$ rooted on $root \rightarrow right$
**return** $L.countSemiLeafNodes() + R.countSemiLeafNodes()$
**end if**

# 2

## 2.1

BinaryMinHeap(int Capacity):

$array = new$ T[Capacity]
$capacity =$ Capacity
$size = 0$

## 2.2

~BinaryMinHeap():
  delete[] $array$

## 2.3

void percolate(int nodeI):
  **if** $nodeI * 2 + 1 \leq size$ **then**
    $minIdx =$ index of $min(array[nodeI * 2], array[nodeI * 2 + 1]))$
  **else if** $nodeI * 2 == size$ **then**
    $minIdx = size$
  **else**
    **return**
  **end if**
  **if** $array[nodeI] > array[minIdx]$ **then**
    $swap(array[nodeI], array[minIdx])$
    $percolate(minIdx)$
  **end if**

## 2.4

T getMin(int nodeI):
  **return** $array[1]$

## 2.5

bool empty():
  **return** $size == 0$

## 2.6

T extractMin(int nodeI):
  $tmp = array[1]$
  $array[1] = array[size]$
  $size = size - 1$
  $percolate(1)$
  **return** $tmp$

## 2.7

void deleteNode(int nodeI):
  $array[nodeI] = array[size]$
  $size = size - 1$
  $percolate(nodeI)$

## 2.8

void insertKey(T val):

$size = size + 1$
$parent = floor(size/2)$
$idx = size$
**while** $parent > 0$ **do**
  **if** $array[parent] >$ val **then**
    $swap(array[parent], array[idx])$
    $idx = parent$
    $parent = floor(parent/2)$
  **else**
    **return**
  **end if**
**end while**

## 2.9

T getLeftChild(int nodeI):

**if** $nodeI * 2 <= size$ **then**
  **return** $array[nodeI * 2]$
**end if**

## 2.10

T getRightChild(int nodeI):

**if** $nodeI * 2 + 1 <= size$ **then**
  **return** $array[nodeI * 2 + 1]$
**end if**