i117: Basic of Programming

5. User-defined data structures (2)

Kazuhiro Ogata, Canh Minh Do

Roadmap

- Binary trees
- Enumeration types
- User-defined exceptions

Binary trees

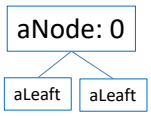
Inductively defined as follows:

(1) Leaf is a binary tree.

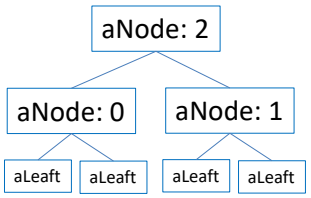
(2) If *v* is a value and *lt* & *rt* are binary trees, then Node(*v*,*lt*,*rt*) is a binary tree.

Leaf

Node(0,Leaf,Leaf)

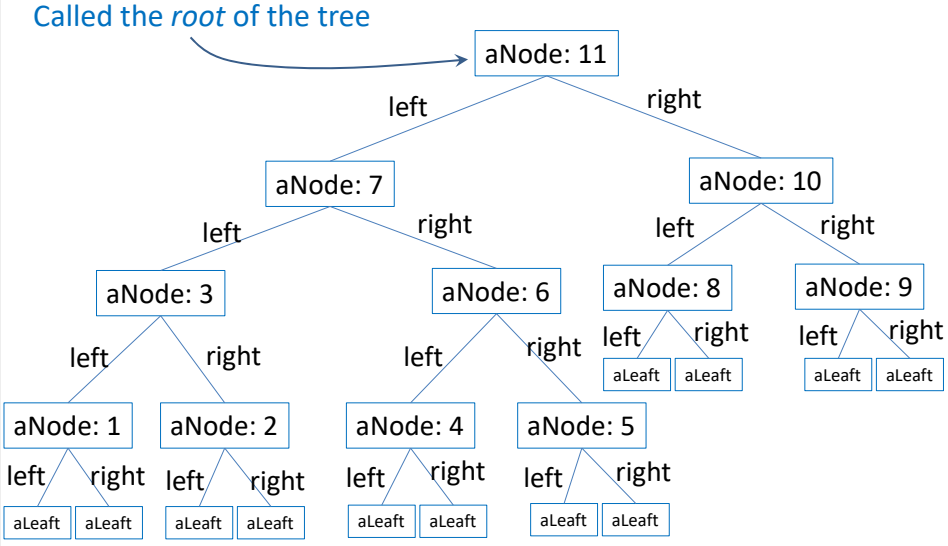


Node(2,
Node(0,Leaf,Leaf),
Node(1,Leaf,Leaf))



Binary trees

Called the *root* of the tree



Binary trees

```
class Tree(object):
    def isLeaf(self):
        pass
    def __str__(self):
        pass
```

```
class Leaf(Tree):
    def isLeaf(self):
        return True
    def __str__(self):
        return 'leaf'
```


Binary trees

```
class Node(Tree):
    def isLeaf(self):
        return False
    def __str__(self):
        return '(val: ' + str(self.val) + ') (left: ' + str(self.left) + ') (right: ' + str(self.right) + ')'
    def __init__(self,v,lt,rt):
        self.val = v
        self.left = lt
        self.right = rt
```

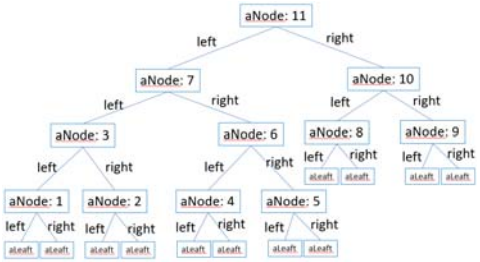
Binary trees

```
n1 = Node(1,Leaf(),Leaf())
n2 = Node(2,Leaf(),Leaf())
n3 = Node(3,n1,n2)
n4 = Node(4,Leaf(),Leaf())
n5 = Node(5,Leaf(),Leaf())
n6 = Node(6,n4,n5)
n7 = Node(7,n3,n6)
n8 = Node(8,Leaf(),Leaf())
n9 = Node(9,Leaf(),Leaf())
n10 = Node(10,n8,n9)
n11 = Node(11,n7,n10)
print(str(n11))
```

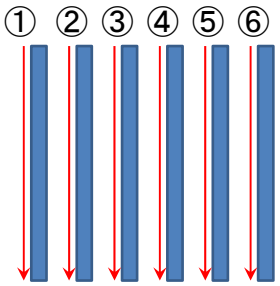

Binary trees

- How to search a binary tree for something (or value) can be classified into two ways:
  - *Depth-first search*
  - *Breadth-first search*
- Depth-first search: For each path from the root to each leaf, search is carried out.
- Breadth-first search: Search starts with the shallowest depth that only consists of the root; all nodes located at the same depth are checked, and if nothing is found, the search moves to the next depth.

Binary trees



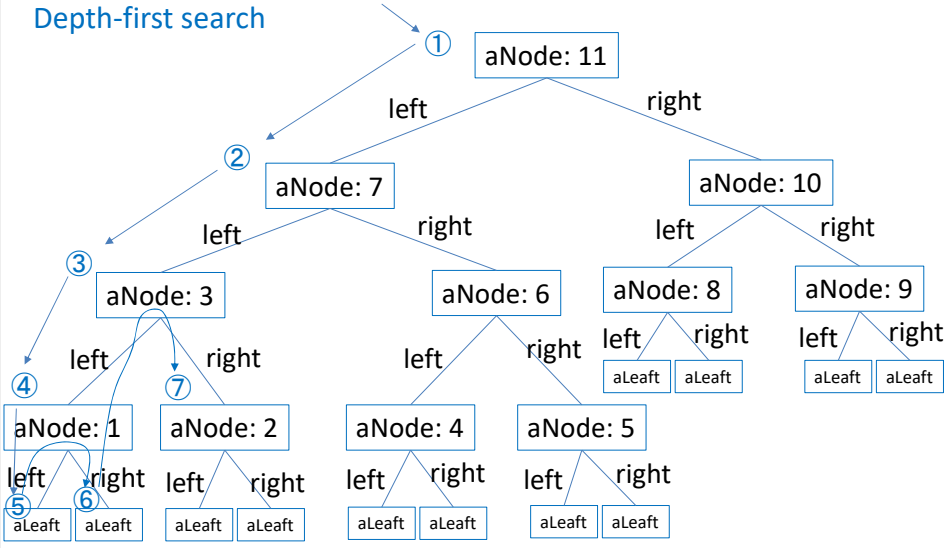
Depth-first search



Breadth-first search

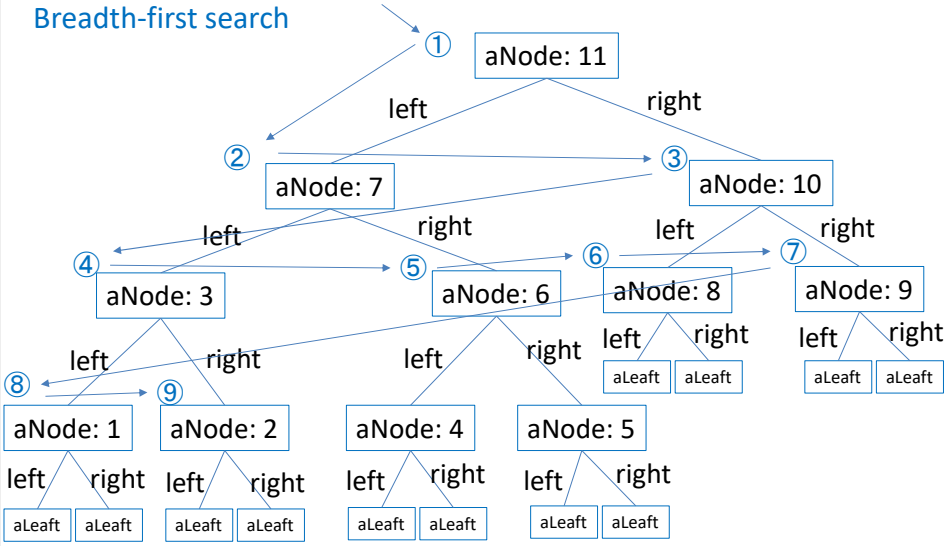
Binary trees

Depth-first search



Binary trees

Breadth-first search



Binary trees

```
def dfSearch(tree,x):
    if tree.isLeaf():
        return tree
    if x == tree.val:
        return tree
    tmp = dfSearch(tree.left,x)
    if not tmp.isLeaf():
        return tmp
    return dfSearch(tree.right,x)
```

```
n1 = Node(1,Leaf(),Leaf())
n2 = Node(2,Leaf(),Leaf())
n3 = Node(3,n1,n2)
n4 = Node(4,Leaf(),Leaf())
n5 = Node(5,Leaf(),Leaf())
n6 = Node(6,n4,n5)
n7 = Node(7,n3,n6)
n8 = Node(8,Leaf(),Leaf())
n9 = Node(9,Leaf(),Leaf())
n10 = Node(10,n8,n9)
n11 = Node(11,n7,n10)
print(n11.str())
r = dfSearch(n11,2)
print(str(r))
r = dfSearch(n11,0)
print(str(r))
```

Binary trees

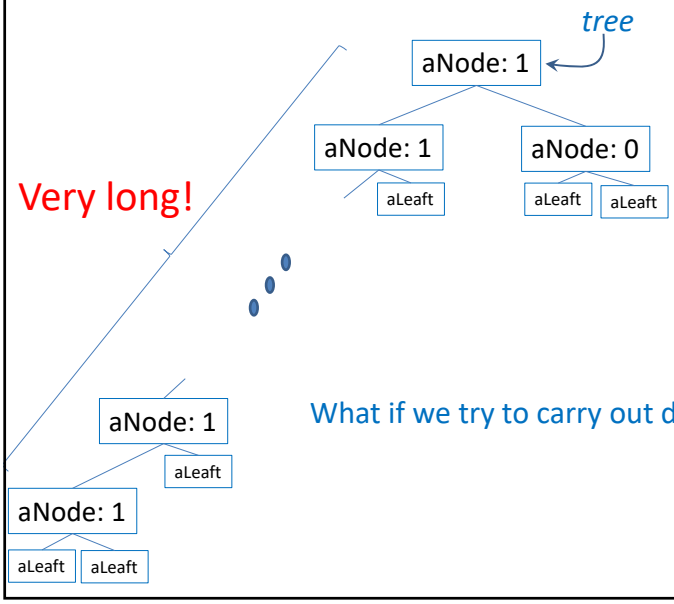
```
def bfSearch(tree,x):
    qu = Queue()
    qu.enqueue(tree)
    while not qu.isEmpty():
        node = qu.top()
        qu.dequeue()
        if node.isLeaf():
            continue
        if x == node.val:
            return node
        qu.enqueue(node.left)
        qu.enqueue(node.right)
    return Leaf()
```

```
n1 = Node(1,Leaf(),Leaf())
n2 = Node(2,Leaf(),Leaf())
n3 = Node(3,n1,n2)
n4 = Node(4,Leaf(),Leaf())
n5 = Node(5,Leaf(),Leaf())
n6 = Node(6,n4,n5)
n7 = Node(7,n3,n6)
n8 = Node(8,Leaf(),Leaf())
n9 = Node(9,Leaf(),Leaf())
n10 = Node(10,n8,n9)
n11 = Node(11,n7,n10)
print(n11.str())
r = bfSearch(n11,2)
print(str(r))
r = bfSearch(n11,0)
print(str(r))
```


Binary trees

- Why do we need to have both depth-first search and breadth-first search for (binary) trees?
- In most cases, depth-first search is more efficient than breadth-first search for conventional computers.
- This is because we need to use a queue to implement breadth-first search for conventional computers.
- When should we use breadth-first search?

Binary trees



What if we try to carry out dfSearch(*tree*,0)?

Enumeration types

- You could use integers, strings, etc., to distinguish several different cases.
- We may want to name such cases and then integers are not very adequate.
- Strings may be OK but need more spaces than integers.
- Enumeration types should be used to distinguish several different cases.

Enumeration types

Enum is a class.
An enumeration type, such as Thing, is defined as a sub-class of Enum.

```
from enum import *

class Thing(Enum):
    Gold = auto()
    Silver = auto()
    Stone = auto()
    Poison = auto()
    Nothing = auto()
```

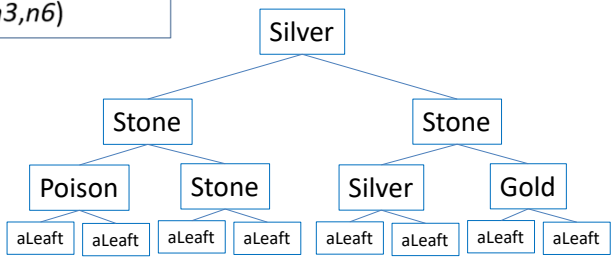
Different values are automatically generated by auto().

```
print(Thing.Gold)
print(Thing.Silver)
print(Thing.Stone)
print(Thing.Poison)
print(Thing.Nothing)
```

```
def __str__(self):
    if self == Thing.Gold:
        return 'Gold'
    elif self == Thing.Silver:
        return 'Silver'
    elif self == Thing.Stone:
        return 'Stone'
    elif self == Thing.Poison:
        return 'Poison'
    elif self == Thing.Nothing:
        return 'Nothing'
    else:
        return 'Error'
```


Enumeration types

```
n1 = Node(Thing.Poison,Leaf(),Leaf())
n2 = Node(Thing.Stone,Leaf(),Leaf())
n3 = Node(Thing.Stone,n1,n2)
n4 = Node(Thing.Silver,Leaf(),Leaf())
n5 = Node(Thing.Gold,Leaf(),Leaf())
n6 = Node(Thing.Stone,n4,n5)
n8 = Node(Thing.Silver,n3,n6)
```



Enumeration types

- Let's make a game as follows:
  - A tree that has Gold, Silver, Stone and Poison is searched in a random way;
  - If you find Gold, you win;
  - If you find Poison, you lose; etc.
- How do we do "search in a random way?"
- It is based on Breadth-first search for trees.
  - The queue used is shuffled somehow from time to time.

Enumeration types

The module *random* is imported.
Note that
import random
is basically the same as
from random **import** *

An integer x such that $0 \geq x > size$ is randomly generated.

```
import random
class Queue(object):
  elements = []
  ...
def shuffle(self):
    size = len(self.elements)
    t = size // 2
    while t > 0:
      i = random.randrange(size)
      j = random.randrange(size)
      self.swap(i,j)
      t = t – 1
```

Enumeration types

```
def swap(self,i,j):
    if i >= 0 and j >= 0 and i < len(self.elements) and j < len(self.elements) and i != j:
        tmp = self.elements[i]
        self.elements[i] = self.elements[j]
        self.elements[j] = tmp
...
```

```
q = Queue()
q.enqueue(1)
q.enqueue(2)
q.enqueue(3)
q.enqueue(4)
q.enqueue(5)
q.enqueue(6)
```

```
q.enqueue(7)
q.enqueue(8)
q.enqueue(9)
q.enqueue(10)
print(q.str())
q.shuffle()
print(q.str())
```

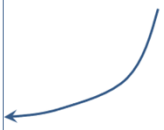

Enumeration types

```
from queue import *
from thing import *

def rSearch(tree,x):
    qu = Queue()
    qu.enqueue(tree)
    while not qu.isEmpty():
        node = qu.top()
        qu.dequeue()
        if isinstance(node, Node):
            print(node.val)
        if node.isLeaf():
            continue
        if x == node.val:
            return node
        qu.enqueue(node.left)
        qu.enqueue(node.right)
        qu.shuffle()
    return Leaf()
```

Enumeration types

```
n1 = Node(Thing.Stone,Leaf(),Leaf())
n2 = Node(Thing.Poison,Leaf(),Leaf())
n3 = Node(Thing.Stone,n1,n2)
n4 = Node(Thing.Silver,Leaf(),Leaf())
n5 = Node(Thing.Silver,Leaf(),Leaf())
n6 = Node(Thing.Stone,n4,n5)
n7 = Node(Thing.Stone,n3,n6)
...
n11 = Node(Thing.Stone,n7,n8)
...
n17 = Node(Thing.Stone,n15,n16)

r = rSearch(n17,Thing.Gold)
print(str(r))
```

This piece of code is available from the course website.



Enumeration types

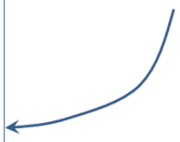
Search bound

```
def gameSearch(tree,x,times):
    qu = Queue()
    qu.enqueue(tree)
    while not qu.isEmpty():
        if times <= 0:
            print('Failure!')
            print('You are exhausted.')
            return
        times = times - 1
        node = qu.top()
        qu.dequeue()
        if node.isLeaf():
            continue
        if node.val == Thing.Poison:
            print('Failure!')
            print('You found Poison.')
            return
        if x == node.val:
            print('Success!')
            print('You found ', node.val, '.')
            return
        qu.enqueue(node.left)
        qu.enqueue(node.right)
        qu.shuffle()
    print('Failure!')
    print('Nothing was found.')
```

Enumeration types

```
n1 = Node(Thing.Stone,Leaf(),Leaf())
n2 = Node(Thing.Poison,Leaf(),Leaf())
n3 = Node(Thing.Stone,n1,n2)
n4 = Node(Thing.Silver,Leaf(),Leaf())
n5 = Node(Thing.Silver,Leaf(),Leaf())
n6 = Node(Thing.Stone,n4,n5)
n7 = Node(Thing.Stone,n3,n6)
...
n11 = Node(Thing.Stone,n7,n8)
...
n17 = Node(Thing.Stone,n15,n16)

gameSearch(n17,Thing.Gold,25)
```

This piece of code is available from the course website.



User-defined exceptions

- You may want to distinguish several different exceptions so that you can handle each exception adequately.
- You can define your own exceptions to make it doable.

User-defined exceptions

Three exceptions Poison, Exhausted, and NotFound, are defined as follows:

```
class Poison(Exception):
    pass

class Exhausted(Exception):
    pass

class NotFound(Exception):
    pass
```

Exception is a class, and so are user-defined exceptions.
User-defined exceptions are sub-classes of Exception.

User-defined exceptions

- gameSearch will be revised such that when you find Poison, an exception “Poison” is raised, when you exceed the search bound, an exception “Exhausted” is raised, and when you do not find Gold, an exception “NotFound” is raised.

User-defined exceptions

```
def revGameSearch(tree,x,times):
    qu = Queue()
    qu.enqueue(tree)
    while not qu.isEmpty():
        if times <= 0:
            raise Exhausted('You are exhausted.')
        times = times - 1
        node = qu.top()
        qu.dequeue()
        if node.isLeaf():
            continue
```


User-defined exceptions

```
        if node.val == Thing.Poison:
            raise Poison('You found Poison.')
        if x == node.val:
            print('Success!')
            print('You found ', node.val, '.')
            return
        qu.enqueue(node.left)
        qu.enqueue(node.right)
        qu.shuffle()
    raise NotFound('Nothing was found.')
```

User-defined exceptions

```
def game(tree,x,times):
    try:
        revGameSearch(tree,x,times)
    except Exhausted as em:
        print('Failure!')
        print(em)
    except Poison as em:
        print('Failure!')
        print(em)
    except NotFound as em:
        print('Failure!')
        print(em)
```

Multiple exceptions can be handled in each way.

Enumeration types

```
n1 = Node(Thing.Stone,Leaf(),Leaf())
n2 = Node(Thing.Poison,Leaf(),Leaf())
n3 = Node(Thing.Stone,n1,n2)
n4 = Node(Thing.Silver,Leaf(),Leaf())
n5 = Node(Thing.Silver,Leaf(),Leaf())
n6 = Node(Thing.Stone,n4,n5)
n7 = Node(Thing.Stone,n3,n6)
...
n11 = Node(Thing.Stone,n7,n8)
...
n17 = Node(Thing.Stone,n15,n16)

game(n17,Thing.Gold,25)
```

This piece of code is available from the course website.