

M2 ch14: Binary Search Trees, Queues and Stacks

1 Dec 2008

CMPT14x

Dr. Sean Ho

Trinity Western University

Binary search trees

- Binary trees (degree=2) are handy for keeping things in sorted order:

```
class BST:
```

```
    def __init__(self, data=None):
```

```
        self.data = data
```

```
        self.left = None
```

```
        self.right = None
```

```
        (* could also have a parent ptr *)
```

```
root = BST( 'Braeburn' )
```

```
root.left = BST( 'Ambrosia' )
```

```
root.right = BST( 'Gala' )
```

```
root.right.left = BST( 'Fuji' )
```



- Everything in **left** subtree is **smaller**
- Everything in **right** subtree is **bigger**

Binary tree traversals

- Pre-order traversal of binary tree:

- Do **self** first, then **left** child, then **right**

- ◆ 3 - 2 - 1 - 5 - 4 - 6

- In-order traversal:

- Do **left** child, then **self**, then **right** child

- ◆ 1 - 2 - 3 - 4 - 5 - 6 (**sorted** order in BST)

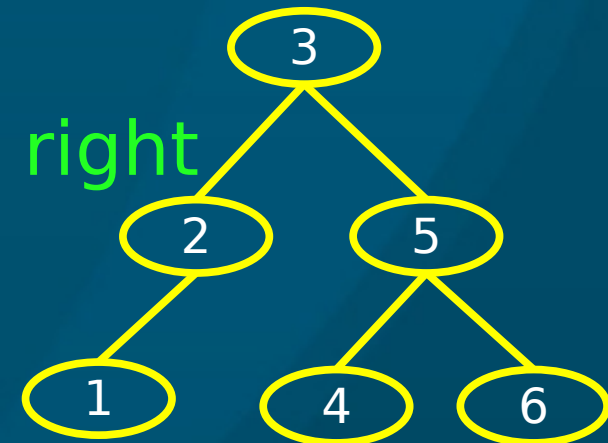
- ◆ e.g. expressions: "12 + (2 * 5)"

- Post-order traversal:

- Do **both** children first before **self**

- ◆ 1 - 2 - 4 - 6 - 5 - 3

- ◆ e.g. Reverse Polish Notation: 12, 2, 5, *, +



Searching a BST

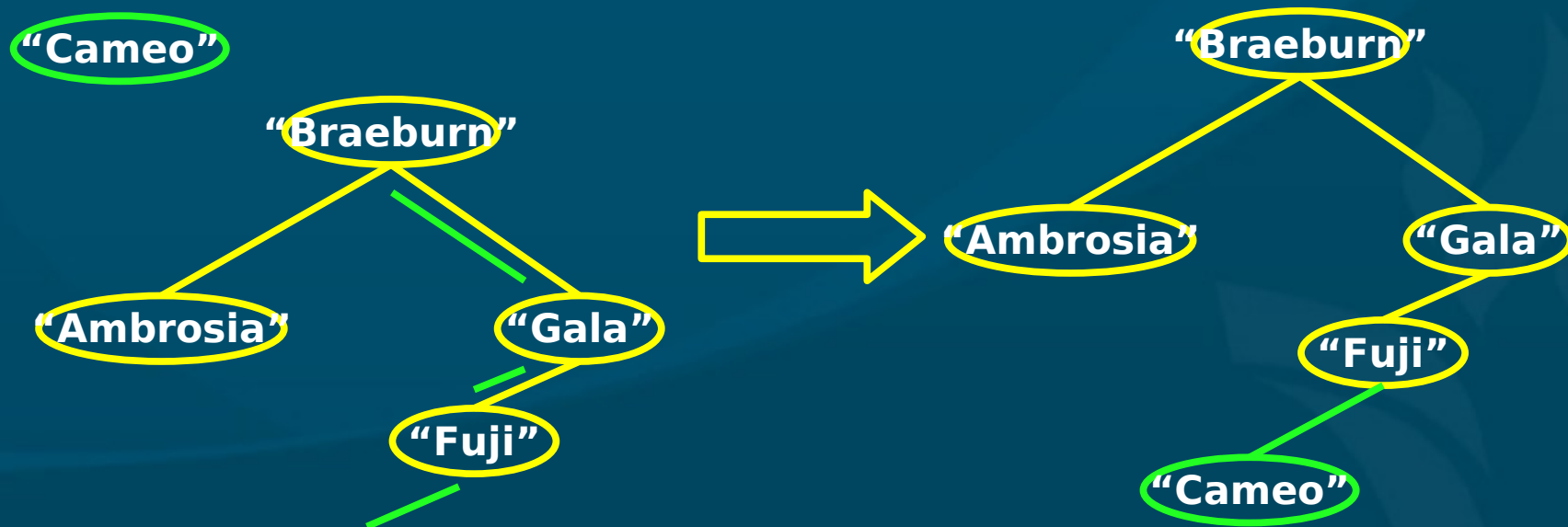
- Recursive algorithm:

```
def search (self, key):  
    if key == self.data:  
        return self  
    elif key < self.data and self.left !=  
        None:  
        return self.left.search(key)  
    elif key > self.data and self.right !=  
        None:  
        return self.right.search(key)  
    else:  
        return None
```



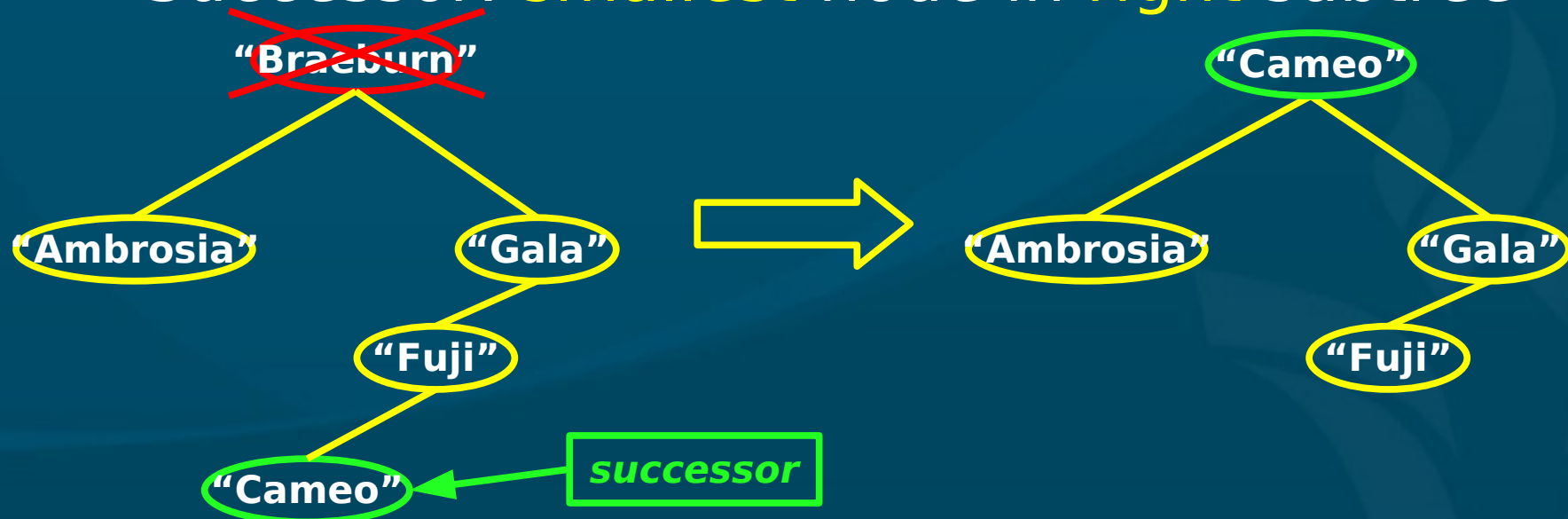
Inserting into a BST

- Keep it sorted: insert in a **proper** place
- One choice: always insert as a **leaf**
 - Use `search()` algorithm to hunt for where the node ought to be if it were already in the tree



Deleting from a BST

- Need to **maintain** sorted structure of BST
- Replace node with **predecessor** or **successor** leaf
 - Predecessor: **largest** node in **left** subtree
 - Successor: **smallest** node in **right** subtree



BSTs and algorithmic efficiency

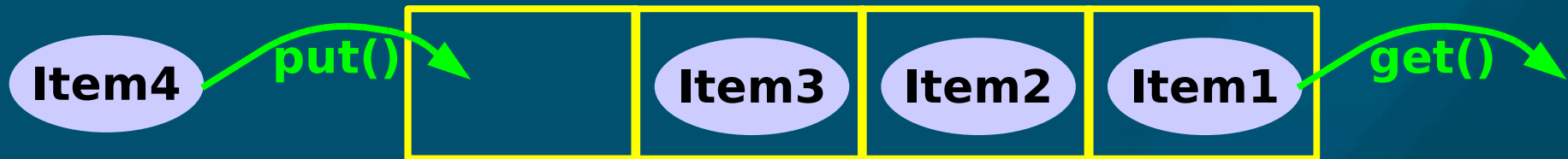
- Searching in a **balanced** binary search tree takes worst-case $O(\log n)$ running time:
 - **Depth** of balanced tree is $\log_2 n$
 - Compare with **arrays/linked lists**: $O(n)$
- But depending on order of inserts, tree may be **unbalanced**:
 - ◆ Insert in **order**: Ambrosia, Braeburn, Fuji, Gala:
 - ◆ Tree **degenerates** to linked-list
 - ◆ Searching becomes $O(n)$
- Keeping a BST **balanced** is a larger topic



◆ e.g., **Splay-trees**

Queues

- A **queue** is a list-like data structure where items **added** first to the queue are **withdrawn** first



- First-in / first-out: **FIFO**
- e.g., waiting in line for a bank teller
- Operations:
 - **put()**: **add** an item to the **end** of the queue
 - **get()**: **withdraw** item at the **head** of the queue
 - **empty()**, **full()**, **size()**: check **number** of items

Implementing queues

- Use a subclass of linked-lists (inheritance)

```
class Queue(LinkedList):
```

- Implement `put()/get()` using linked-list operations:

```
def put(self, data):
```

```
    self.insert(self.size, data)           # insert at tail
```

```
def get(self):
```

```
    data = self.head.data                 # save the  
    payload
```

```
    self.delete(0)                        # delete first  
    node
```

```
    return data
```

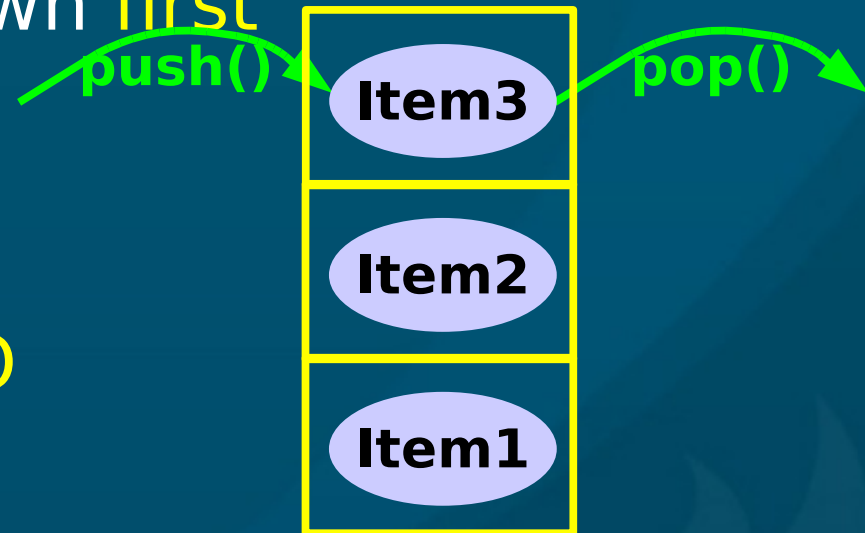
- M2 book gives a different implementation using

dynamic arrays



Stacks

- A **stack** is like a queue, but items added last to the stack are withdrawn **first**



- Last-in / first-out: **LIFO**

- e.g., RPN calculator

- Operations:

- **push()**: **add** an item to the **top** of the stack
- **pop()**: **withdraw** item from the **top** of the stack

- **empty()**, **full()**, **size()**: check **number** of items

Implementing stacks

- Could use either linked-lists or arrays

class Stack:

```
def __init__( self, maxsize=1 ):
```

```
    self.stack = range( maxsize )    # allocate  
    new array
```

```
    self.top = -1    # index of top of  
    stack
```

- push()/pop() from the array:

```
def push( self, data ):    # what if  
    array is full?
```

```
    self.top += 1
```

```
    self.stack[ self.top ] = data    # push onto top
```

```
def pop( self ):
```

```
    self.top -= 1
```

```
    return self.stack[ top+1 ]
```

Python lists as queues/stacks

- Most languages only have **arrays** and **pointers**
 - Use pointers to build a **linked-list** ADT
 - Use either **arrays** or **linked-lists** to make **queue** or **stack** ADT
- **Python lists** are special
 - Many of the advantages of linked-lists
 - Can use Python lists naturally as queues/stacks
 - **Stack**: `.append()`, `.pop()` (pops from **tail**)
 - **Queue**: `.append()`, `.pop(0)` (pops from **head**)