

# §12.7-12.11, 14.7-14.8: Linked Lists and Binary Search Trees

---

26 Nov 2008

CMPT14x

Dr. Sean Ho

Trinity Western University

# Linked lists: creating

- A **linked list** is a dynamic ADT where each item in the list contains a **pointer** to the next item:

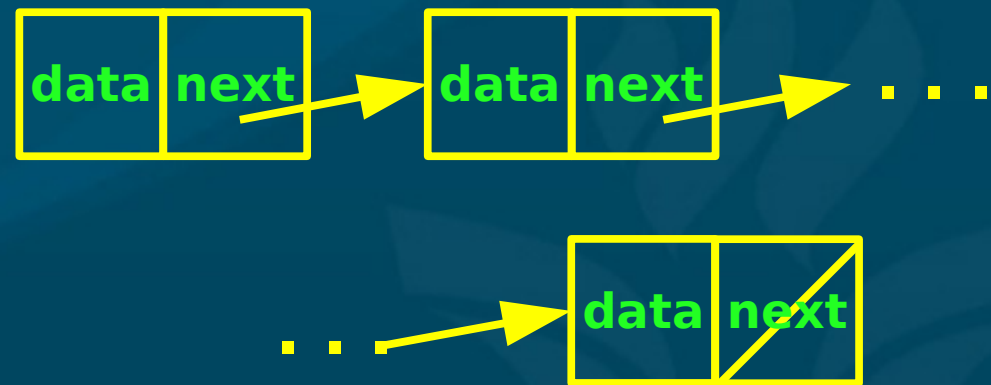
```
class Node:
```

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

```
n1 = Node()
```

```
n2 = Node()
```

```
n1.next = n2
```



# Operations on linked lists

- Index into list (get a reference to  $n^{\text{th}}$  node)
- Print out the list
- Search list for given data (cargo/payload)
- Insert a new node into a linked list
- Delete a node from a linked list
  - By index (0, 1, 2, ...) or by cargo



# Inserting a node into a linked list

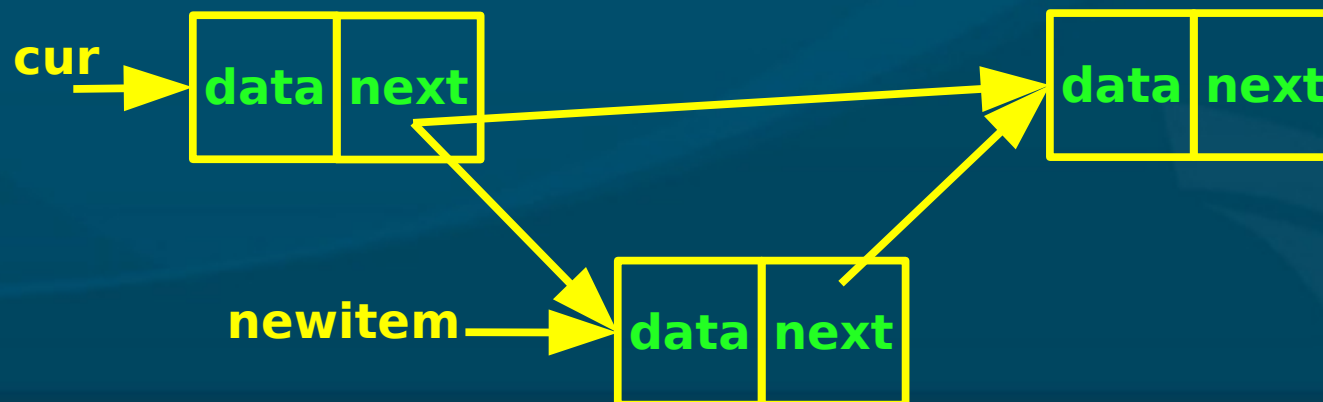
- Follow pointers to get to the right **spot**
  - **Create** a new node with the given cargo
  - **Thread** new node into the list

**newitem = Node(data)**

**newitem.next = cur.next**

**cur.next = newitem**

- What about inserting at **head** of list?



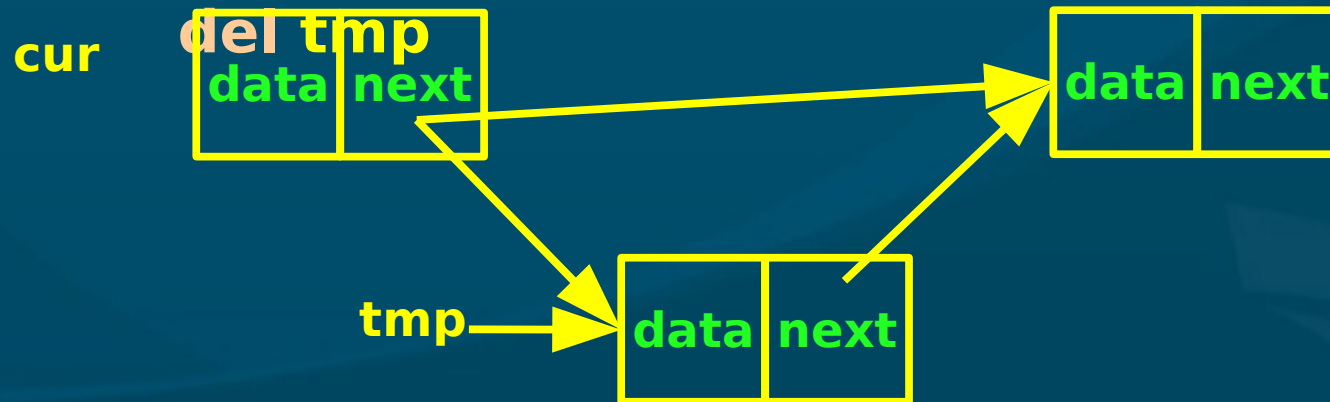
# Insert() method: code

```
def insert (self, n, data=None):  
    """Insert a new node into linked list at position n."""  
    newitem = Node(data)  
    if n == 0: # new head: modify self  
        newitem.next = self  
        self = newitem  
    else:  
        cur = self  
        for idx in range(n-1): # get to proper position  
            cur = cur.next  
        newitem.next = cur.next  
        cur.next = newitem
```

# Deleting from a linked list

- Follow pointers to find the item we want to delete
  - Sew up links to skip over the item
  - Deallocate the item from memory

```
tmp = cur.next  
cur.next = tmp.next
```



# Linked lists: algorithmic efficiency

- Big-O notation:  $O(n)$  means # operations varies linearly with  $n$
- For a linked list with  $n$  items:
  - Insert at **head**: don't have to traverse list:  $O(1)$
  - Append to **tail**: must walk list:  $O(n)$
  - General **insert**:
    - ◆ Worst-case:  $O(n)$
    - ◆ Average-case:  $O(n/2)$ , which is also  $O(n)$
  - **Deleting**: also  $O(n)$
- Double-headed list (keep a **tail pointer**):
  - Speeds up **append to tail** to  $O(1)$

# Variants of linked lists

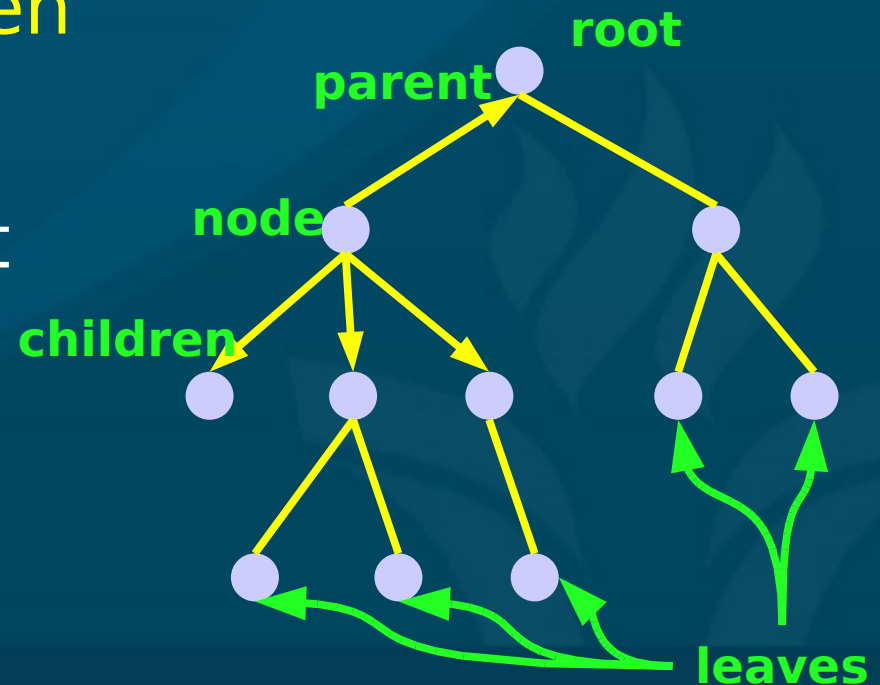
- Circularly linked list:
  - ◆ `tail.next = head`
  - How to keep from infinite loop?
- Bidirectional linked list:

**class Node:**

```
def __init__(self, data=None, prev=None,
             next=None):
    self.data = data
    self.prev = prev
    self.next = next
```

# Trees

- Another kind of dynamic ADT is the **tree**:
  - **Root**: starting node (one per tree)
    - ◆ Could also have a **forest** of several trees
  - Each node has at most one **parent**, and zero or more **children**
  - **Leaves**: no children
  - **Depth**: length of longest path from root
  - **Degree**: max # of children per node



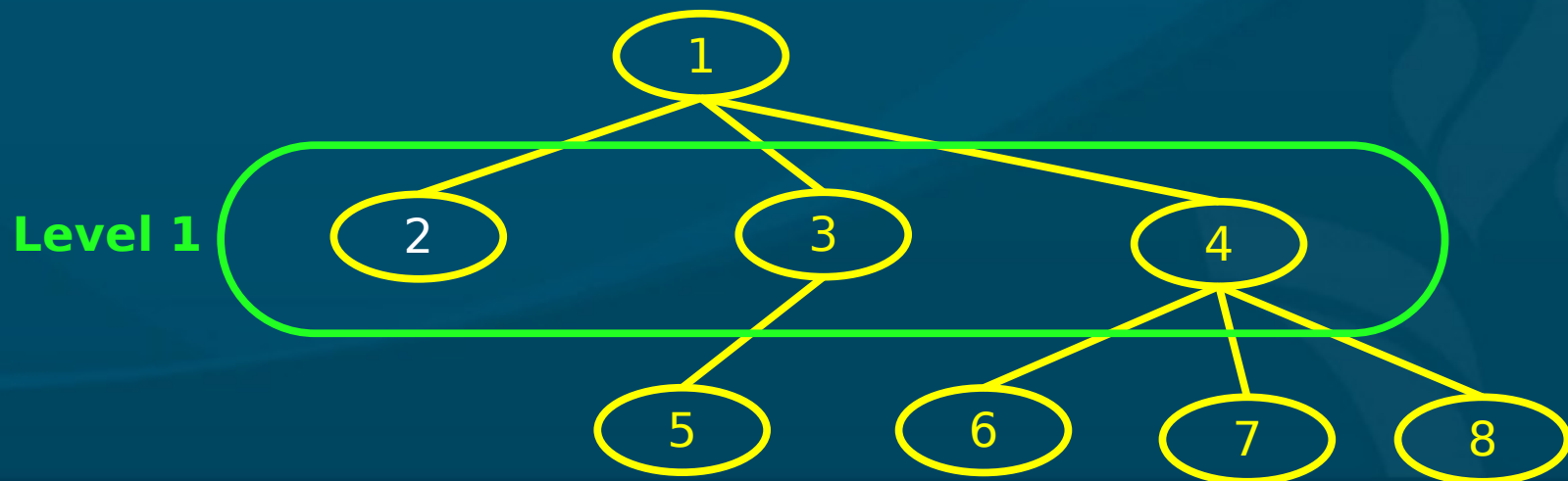
# Searching trees

- A **depth-first** search of a tree pursues each path down to a leaf, then **backtracks** to the next path

◆ 1-2            1-3-5        1-4-6        4-7        4-8

- A **breadth-first** search finishes each **level** before moving on to the next:

◆ 1   2-3-4        5-6-7-8



# 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
```

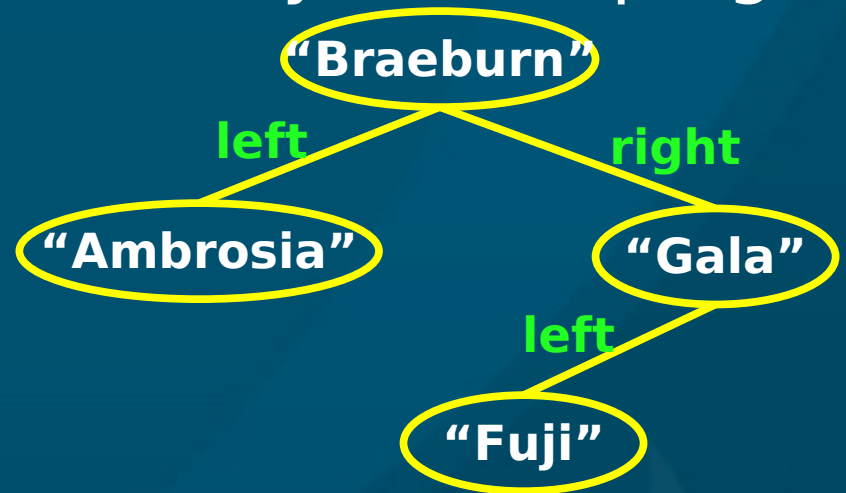
```
        (* could also have a parent ptr *)
```

```
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)

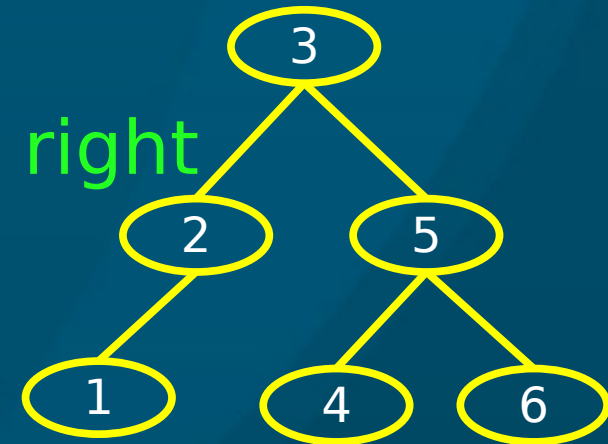
- ◆ e.g. expressions: "12 + (2 \* 5)"

- 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

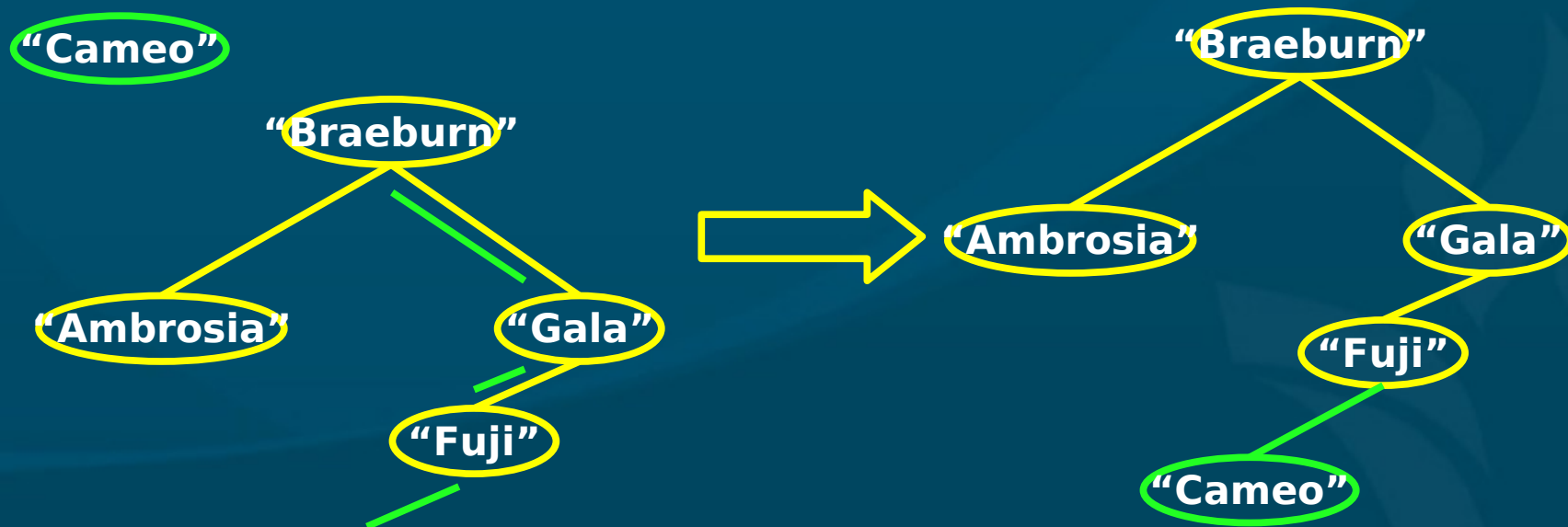
- Recursive algorithm:

```
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
```



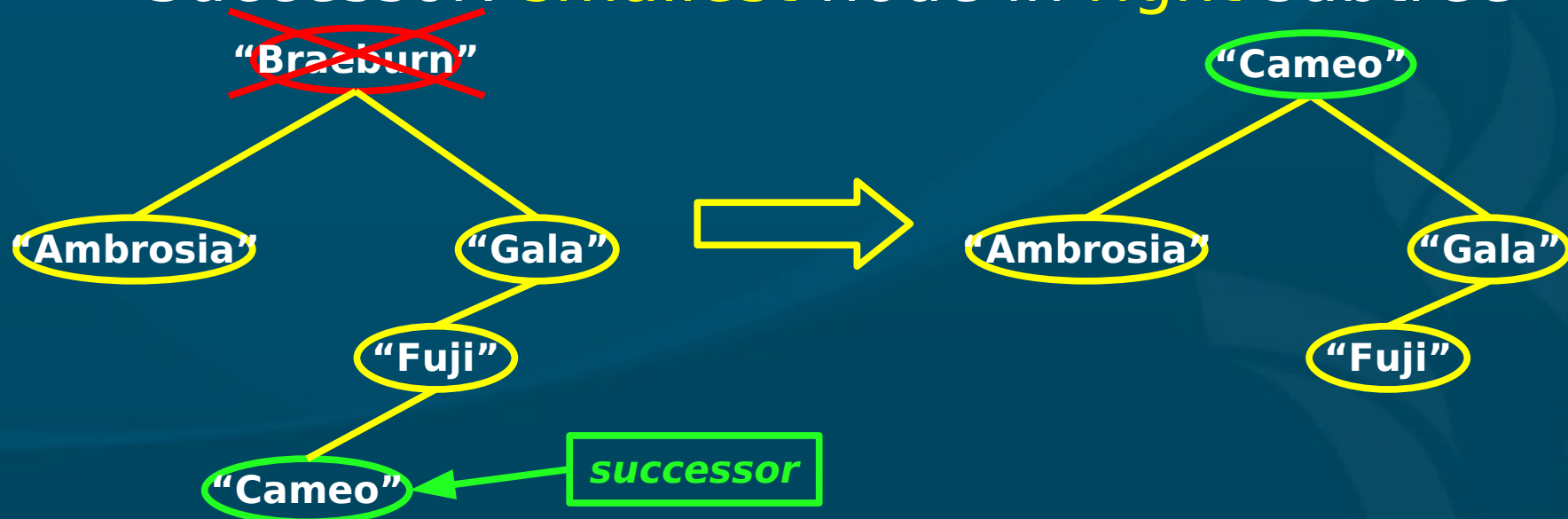
# Inserting into a BST

- 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**