Ch10, 12: Data Structures using Pointers

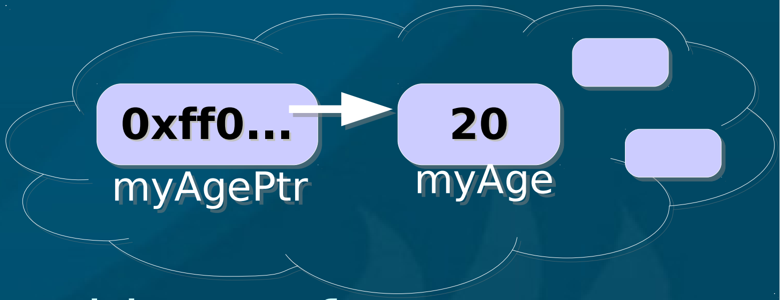
16 Oct 2012
CMPT231
Dr. Sean Ho
Trinity Western University

Outline for today

- Dynamic data structures: using pointers
- Linked lists
  - Variants: doubly-linked, circular
- Stacks and queues
- Trees
- Binary search trees (BSTs)
  - Tree traversals
  - Searching
  - Min/max and successor/predecessor
  - Insert and delete

Pointers

- Local variables created as a program runs are stored in a region of memory called the heap
  - Static variables & formal parameters are stored in the stack frame (size known at compile time)
- A pointer is a variable whose value refers to a memory location in the heap




```
int myAge = 20;
int* myAgePtr;
myAgePtr = &myAge;        // get address of myAge
cout << *myAgePtr;        // dereference the pointer
```

  - Pointer arithmetic can be dangerous

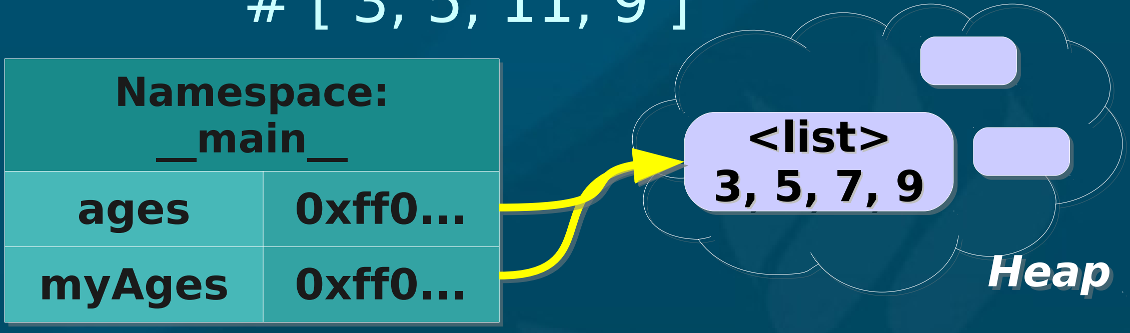
```
*(myAgePtr+1);            // segfault!
*(103883);                // ref random spot in mem
```

Pointer-less languages

- To prevent segfaults, most languages (besides C/C++) do not have explicit pointers
- Instead, you can create references ("aliases"):

```
ages = [ 3, 5, 7, 9]        # Python list (mutable obj)
myAges = ages               # create an alias
myAges[2] = 11              # overwrites "7"
ages                        # [ 3, 5, 11, 9 ]
```

- Variables are entries in a namespace, mapping to locations in the heap



- Be aware of when a reference is made vs a copy!

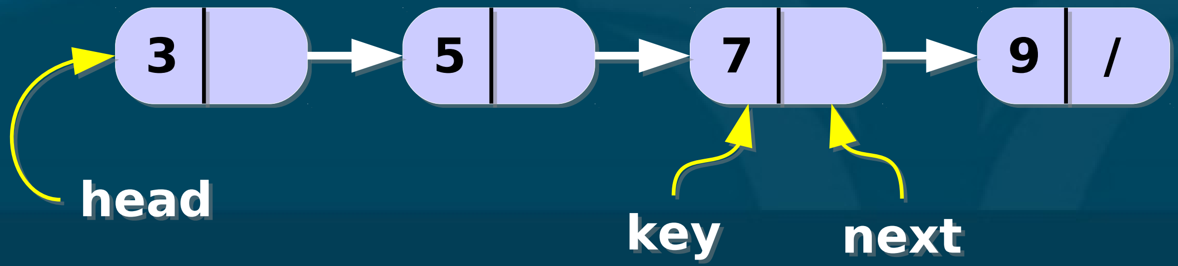
Outline for today

- Dynamic data structures: using pointers
- Linked lists
  - Variants: doubly-linked, circular
- Stacks and queues
- Trees
- Binary search trees (BSTs)
  - Tree traversals
  - Searching
  - Min/max and successor/predecessor
  - Insert and delete

Linked lists

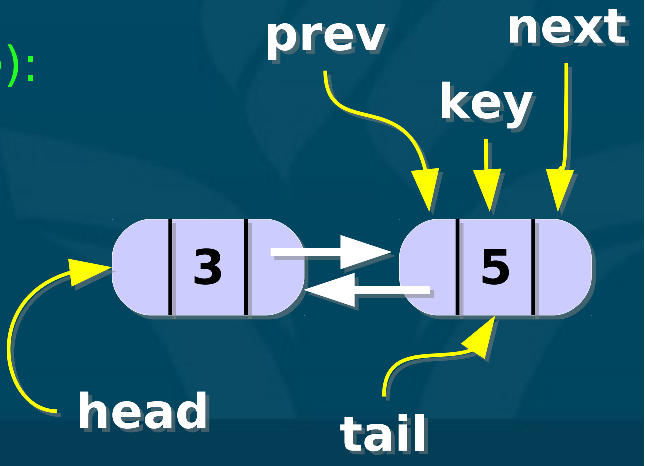
- Linear, array-like data structure, but
  - Dynamic (can change length)
    - Length does not need to be known at compile time)
  - Mid-list insertion/deletion is fast
    - Don't need to shift by copying
  - But random access is slower than array
    - Need to walk down list from head

```
class Node:
    def __init__(self, key=None, next=None):
        self.key = key
        self.next = next
head = Node(9)
head = Node(7, head)
head = Node(5, head)
head = Node(3, head)
```



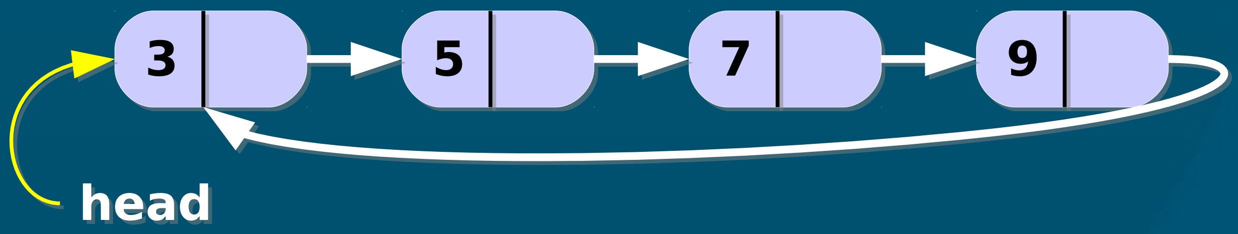
Linked list variants

- The basic list is a singly-linked list
- Doubly-linked lists have both .prev and .next pointers in each node:
  - class Node:
    - def __init__(self, key=None, prev=None, next=None):
      - (self.key, self.prev, self.next) = (key, prev, next)
- Also good to have both head and tail pointers
  - Better to have separate datatype for the overall list:
    - class LinkedList:
      - def __init__(self, head=None, tail=None):
        - (self.head, self.tail) = (head, tail)
    - x = LinkedList(Node(3), Node(5))
    - x.head.next = x.tail
    - x.tail.prev = x.head



Circularly-linked lists

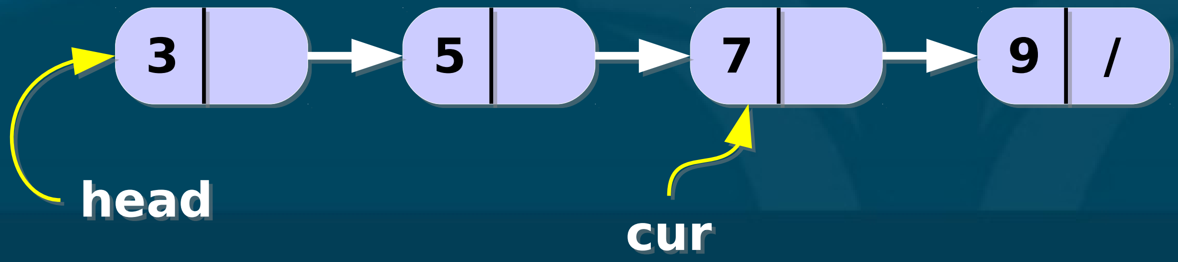
- In a circular singly-linked list, the next pointer of the last node points back to the first node:



  - When traversing the list, make sure we don't just keep circling endlessly!
    - e.g., store length of list, and track how many nodes we've traversed
    - or: add a sentinel node with a special key
- In a circular doubly-linked list, both next and prev links wrap around

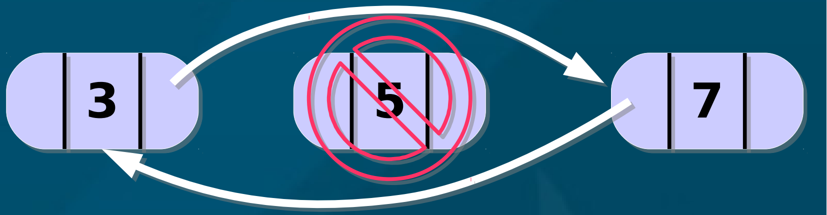
Operations on linked lists

- Insert(key): create a new node with given key, and insert it at the head of the list
  - def insert(self, key=None):
    - self = Node(key, self)
- Search(key): return a reference to node with given key, or return None if it doesn't exist
  - def search(self, key):
    - cur = self
    - while cur != None:
      - if cur.key == key:
        - return cur
      - cur = cur.next
    - return None



Splicing nodes

- Delete(node): splice given node out of list
  - Can be given a reference directly to the node
    - Or given a key (for which we first search)
  - Update prev/next links in neighbouring nodes to skip over the deleted node
    - node.prev.next = node.next
    - node.next.prev = node.prev



  - Free the unused memory so it can be reused
    - del node
    - Otherwise it becomes garbage: allocated memory in the heap that is unused and unreachable
    - Source of memory leaks: heap grows and grows as program runs

Outline for today

- Dynamic data structures: using pointers
- Linked lists
  - Variants: doubly-linked, circular
- Stacks and queues
- Trees
- Binary search trees (BSTs)
  - Tree traversals
  - Searching
  - Min/max and successor/predecessor
  - Insert and delete

Stacks and queues

- Stack: LIFO: last-in-first-out
  - like papers on a memo spike
- Queue: FIFO: first-in-first-out
  - like a pipeline, or a queue at the bank

- Interface:
  - length(), isempty(): # items
  - push(x): add x to stack/queue
  - peek(): get item without deleting
  - pop(): peek and remove item
- Underflow: peek/pop on an empty stack/queue
- Overflow: push on a full stack/queue

Implementing stacks/queues

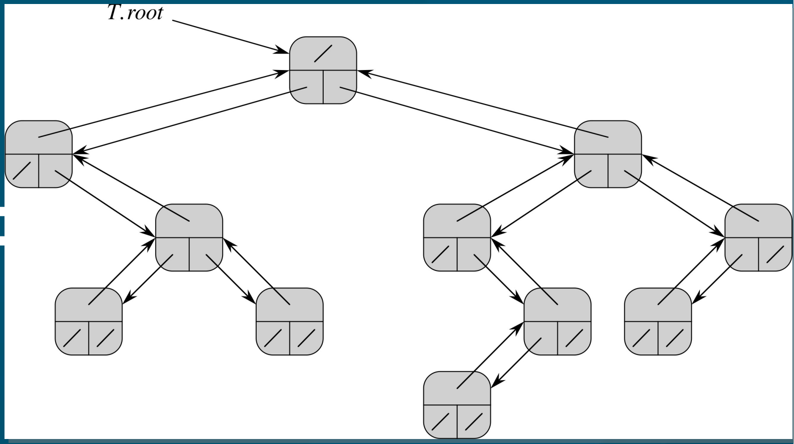
- Stacks/queues are abstract data types (ADTs):
  - Defined in terms of their operations / interface
  - Various implementations have different memory usage, computational complexity, etc.
- Can use either arrays or linked-lists to implement
  - e.g., stack with a singly-linked list:

```
class Stack:
    def __init__(self):
        self.head = None
    def push(self, key):            # overflow not a concern
        self.head = Node(key, self.head)
    def pop(self):                  # watch for underflow!
        item = self.head
        self.head = self.head.next
```

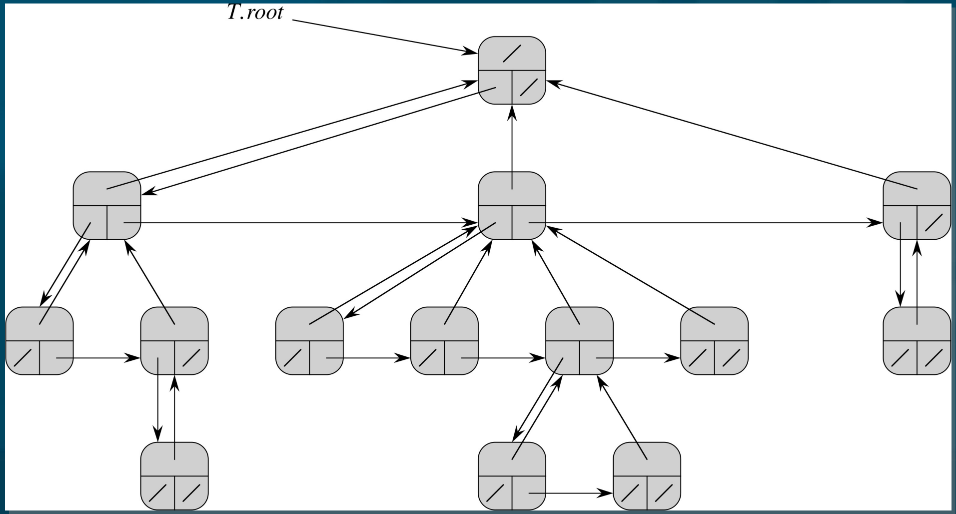
Outline for today

- Dynamic data structures: using pointers
- Linked lists
  - Variants: doubly-linked, circular
- Stacks and queues
- Trees
- Binary search trees (BSTs)
  - Tree traversals
  - Searching
  - Min/max and successor/predecessor
  - Insert and delete

Trees

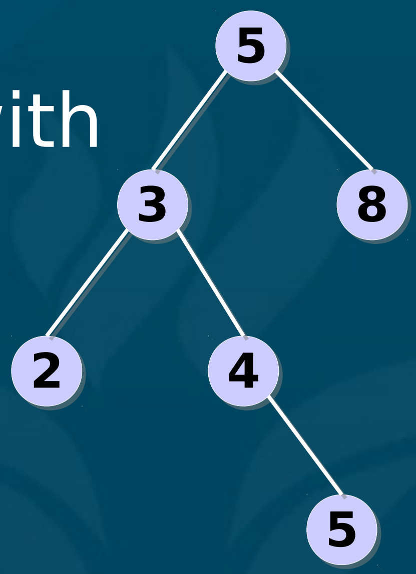


- For binary trees, use 3 pointers:
  - Parent, left child, right child
    - class TreeNode:
      - def __init__(self, par=None, left=None, right=None):
        - (self.par, self.left, self.right) = (par, left, right)
    - class Tree:
      - def __init__(self, root=None):
        - self.root = root
- For d-way trees, with unknown degree d:
  - Pointers: parent, first child, next sibling

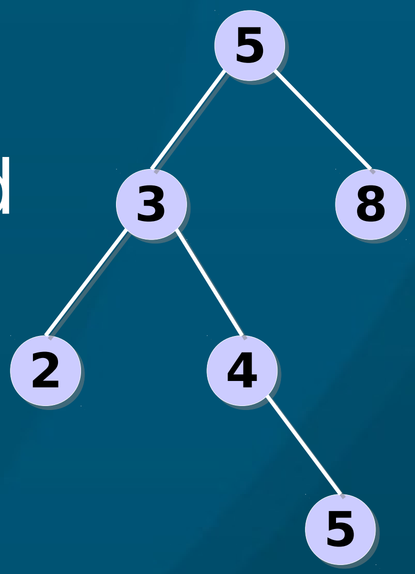


Search trees

- Trees for fast searching
- Operations: insert, delete, search
  - Θ(height of tree): for full tree, Θ(lg n)
  - Can implement a dictionary or priority queue
- Kinds of trees include binary search trees (ch12), red-black trees (ch13), B-trees (ch18)
- Binary search tree (BST): a binary tree with
  - BST property: at any node x,
    - Every node y in left sub-tree has $y \le x$
    - Every node y in right sub-tree has $y \ge x$

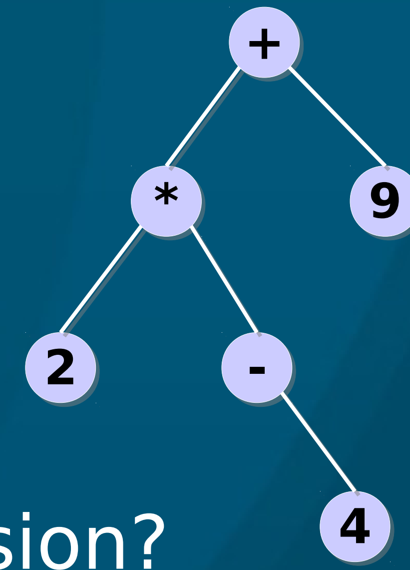


Tree traversals

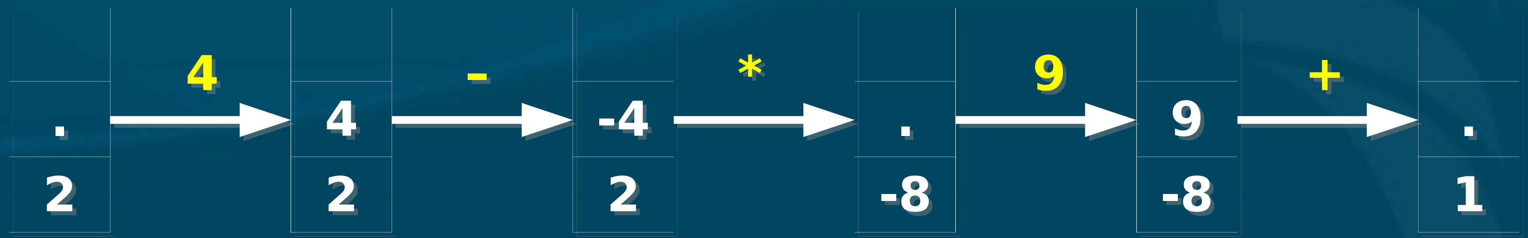


- Traversals/walks print out all nodes
- Preorder: print self before either child
  - preorder(node):
    - print node.key
    - preorder(node.left)
    - preorder(node.right)
  - Output: 5, 3, 2, 4, 5, 8
- Postorder: print both children first before self
  - Output? Pseudocode?
- Inorder: print left child, then self, then right child
  - Output? Pseudocode?
- Which is useful on a tree with the BST property?

Expression trees



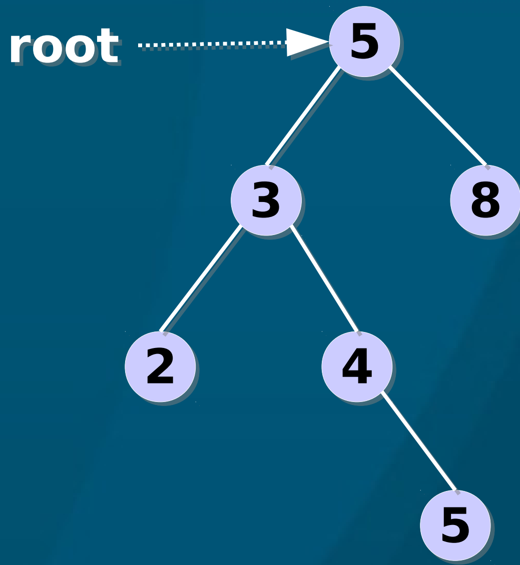
- Trees are also used to parse & evaluate expressions:
  - e.g., (2 * (-4)) + 9
  - Which traversal produces this expression?
  - What tree would represent 2 * (-4 + 9)?
- Reverse Polish Notation (RPN):
  - e.g.: 2, 4, -, *, 9, +
  - Which traversal produces RPN?
  - Make an RPN calculator using a stack:



Outline for today

- Dynamic data structures: using pointers
- Linked lists
  - Variants: doubly-linked, circular
- Stacks and queues
- Trees
- Binary search trees (BSTs)
  - Tree traversals
  - Searching
  - Min/max and successor/predecessor
  - Insert and delete

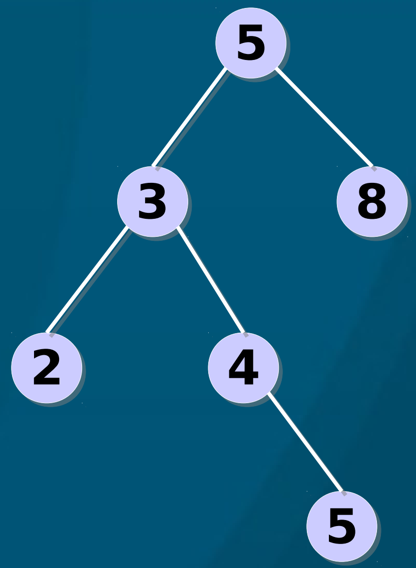
Searching a BST



- Comparison with node's key tells us which subtree to recurse down:
  - search(node, key):
    - if node is NULL or node.key == key:
      - return node
    - if key < node.key:
      - return search(node.left, key)
    - else:
      - return search(node.right, key)
  - e.g., search(root, 4)

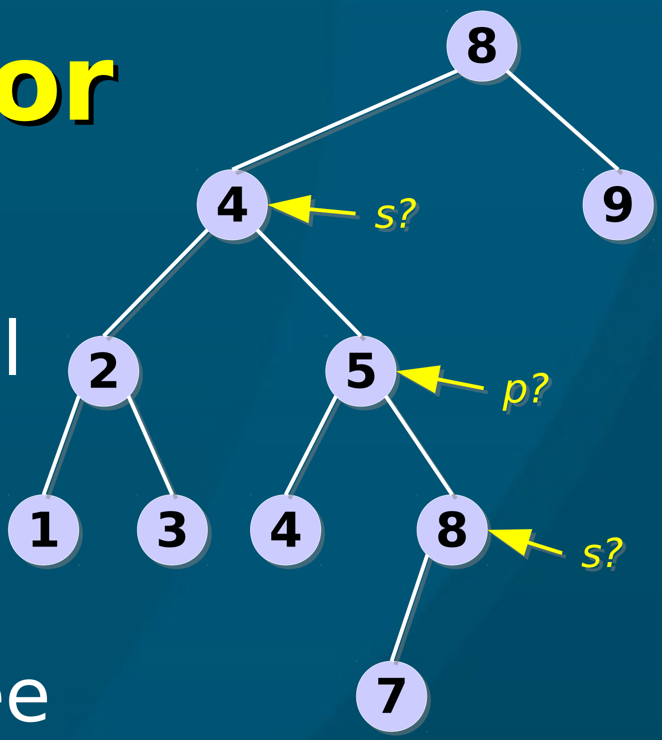
- Complexity is O(height of tree)
  - If tree is full, this is $\Theta(\lg n)$
  - But in worst-case: linked-list is also a tree!
  - $\Rightarrow$ want to keep tree balanced

Min/max of BST



- Find the smallest/largest keys in a BST:
- Smallest:
  - Keep taking left child as far as we can
    - min(node):
      - while node.left is not NULL:
        - node = node.left
      - return node.key
- Largest: keep taking right child as far as we can
    - max(node):
      - while node.right is not NULL:
        - node = node.right
      - return node.key
- Could also implement recursively, but iterative solution is faster, less memory

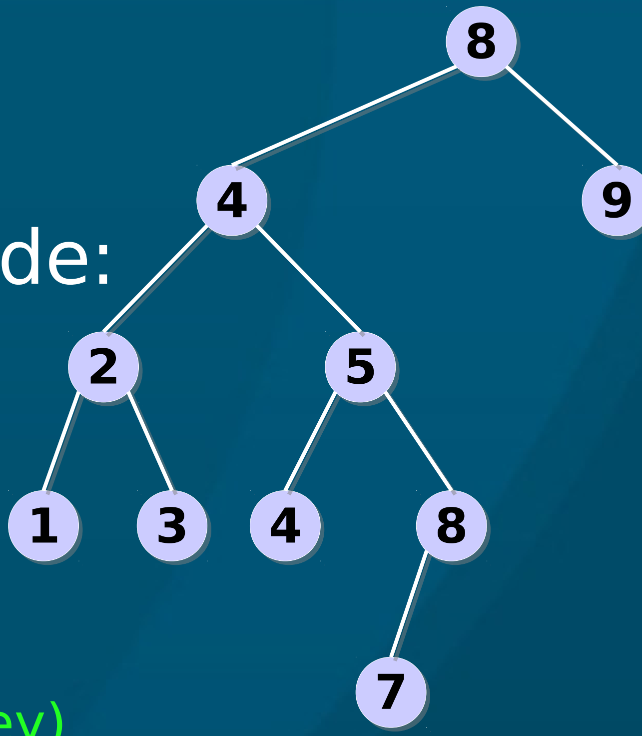
Successor / predecessor



- The successor of a node is next in line in an in-order traversal
  - Predecessor is previous in line
- If right subtree is not NULL:
  - Successor = min of right subtree
- If right subtree is NULL:
  - Walk up the tree until a parent link turns right

```
successor(node):
  if node.right is not NULL:
    return min( node.right )
  (cur, par) = (node, node.parent)
  while par is not NULL and cur == par.right:
    (cur, par) = (par, par.parent)
  return par
```

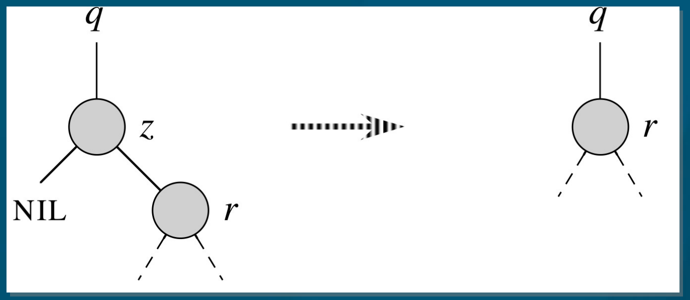
Inserting into a BST



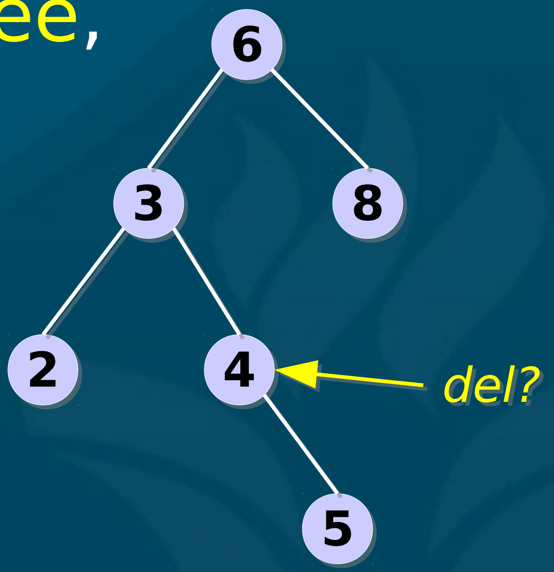
- Do a search to find spot to add node:
  - insert(root, key):
    - cur = root
    - while cur is not NULL:
      - if key < cur.key:
        - (go left)
        - if cur.left is NULL:
          - cur.left = new Node(key)
          - cur.left.parent = cur
          - return
        - cur = cur.left
      - else:
        - (go right)
        - if cur.right is NULL:
          - cur.right = new Node(key)
          - cur.right.parent = cur
          - return
        - cur = cur.right

insert(6)?

Deleting from a BST

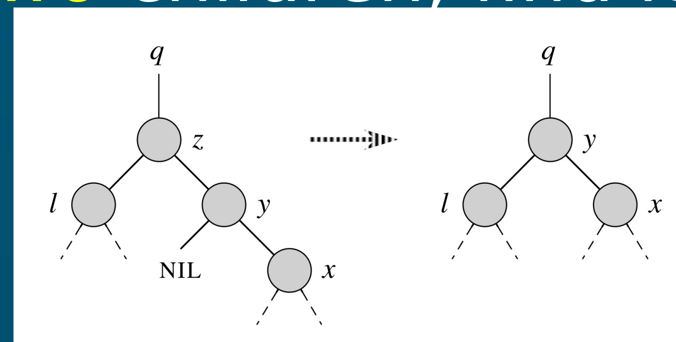


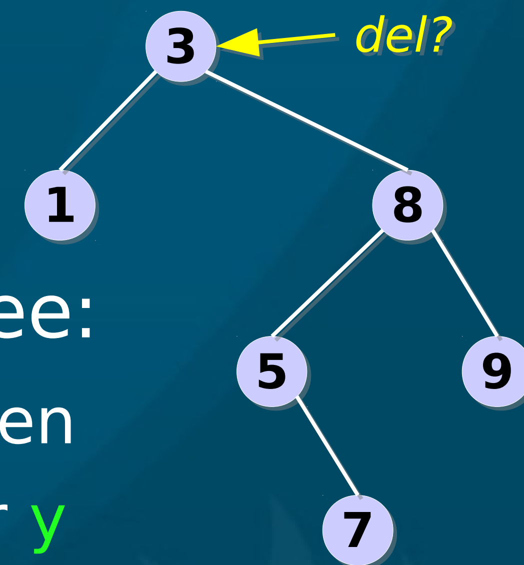
- If node z is a leaf, just delete it (and update links)
- If node has one child, promote it to node's place
  - Child brings its subtrees along with it
- If node has two children, find its successor y:
  - Successor must be in right subtree, with no left child *(why?)*
  - Need to do a bit more splicing



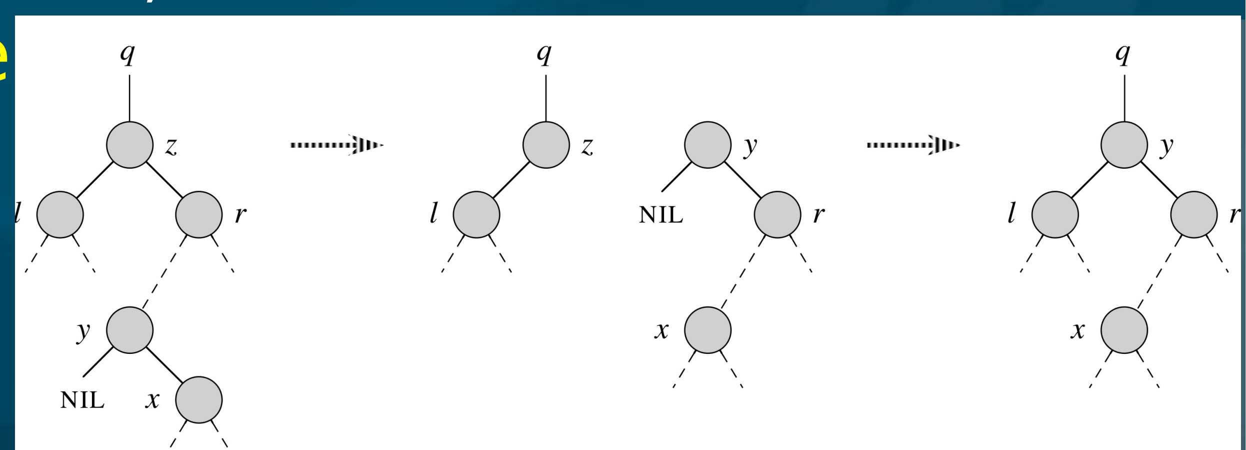
Deletion, continued

- If the node z has two children, find its successor y:
  - if successor is a direct child, just promote it:





  - If successor is elsewhere in right tree:
    - Replace it with its own right child r, then
    - Replace the node z with the successor y
- End result: y replaces z, and the rest of z's old right subtree becomes y's right subtree



Outline for today

- Dynamic data structures: using pointers
- Linked lists
  - Variants: doubly-linked, circular
- Stacks and queues
- Trees
- Binary search trees (BSTs)
  - Tree traversals
  - Searching
  - Min/max and successor/predecessor
  - Insert and delete