Walking data structures

Terence Parr
MSDS program
University of San Francisco

UNIVERSITY OF SAN FRANCISCO

Most algorithms walk data structures

- That means we need to know how to walk arrays, linked lists, trees, and graphs; and combinations of those
- Think of walking an entire data structure as the foundational alg
- The algorithm then typically computes something during the walk and often avoids part of the data structure to reduce computation time

Walking arrays

- Arrays provide superfast *random-access* to the ith element
- Node+pointer based data structures are typically not random access; we need to walk through the structure to access items; e.g., linked lists don't have random access to ith element
- Incrementing/decrementing a pointer or index is most common walk
- Walking the entire array is our base functionality
- But, often we hope to access fewer items; e.g., binary search bounces around depending on item values (more on this later)
- Arrays are great for holding rows or columns of data
- Matrices are 2D arrays, random-access to i,j

Matrix-walking pattern

- When you see this pattern, think of walking elements of matrix

```
def walk(A,nrows,ncols):
    for i in range(nrows):
        for j in range(ncols):
            # process A[i][j]
```

```
[[1, 1, 1, 0, 0],
 [0, 0, 1, 1, 1],
 [0, 1, 1, 1, 0],
 [1, 1, 0, 0, 1],
 [0, 1, 1, 1, 1]]
```

```
def walk_lower_triangle(A,nrows,ncols):
    for i in range(nrows):
        for j in range(i):
            # process A[i][j]
```

Exercise: visualize walking linked list

- Link https://goo.gl/i68EzJ uses pythontutor.com to visualize a pointer walking through a linked list
- You can step forward and backward
- Now, write a while-loop to walk from head to tail using pointer p, printing the value field at each node
- Write that code until you can do it easily and quickly (and correctly) without looking

```
p = head
while p is not None:
    p = p.next
```

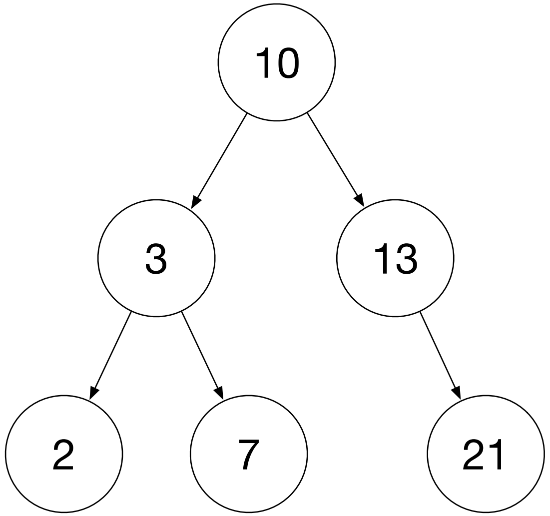
Building binary trees

```
class TreeNode:
  def __init__(self, value, left, right):
    self.value = value
    self.left = left
    self.right = right
```

- Manual construction is a simple matter of creating nodes and setting left/right child pointers
- Construction of decision trees from data requires an algorithm that decides what nodes to create and hook up but it's important to learn manual construction
- **Exercise**: go to notebook linked below and step through "Constructing binary tree" section;
  try modifying the node addition sequence to get different trees

https://github.com/parrt/msds689/blob/master/notes/walking.ipynb

Recursive walk is the most natural



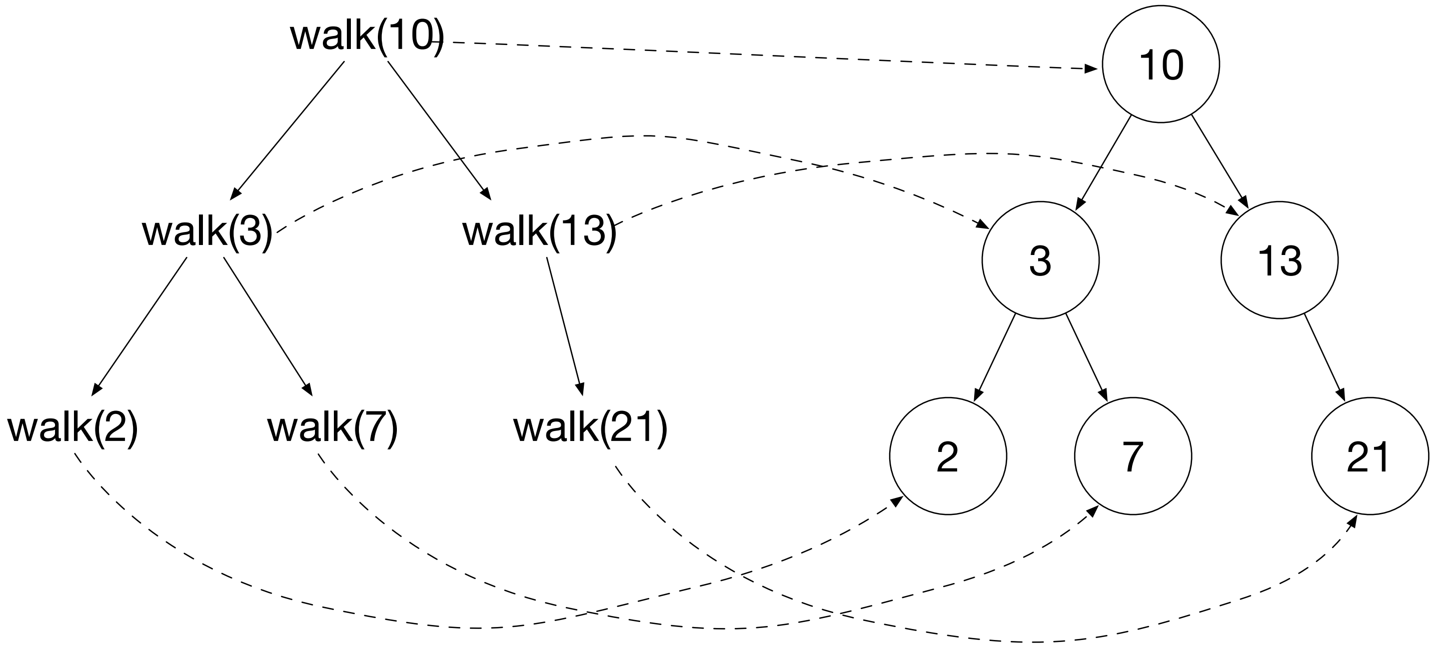
- "Depth-first search" is how we walk every node
- The visitation order (discover, finish nodes) always same
- Traversal (pre-, in-, post-) order depends on action location

```
def walk_tree(p:TreeNode):
    if p is None: return
    print(p.value) # preorder
    walk_tree(p.left)
    walk_tree(p.right)
```

```
def walk_tree(p:TreeNode):
    if p is None: return
    walk_tree(p.left)
    walk_tree(p.right)
    print(p.value) # postorder
```

Recursion tree vs tree

```
def walk(p:TreeNode):
    if p is None: return
    print(p.value)
    walk(p.left)
    walk(p.right)
```



Exercise: Searching in binary tree

- See if you modify the tree walker to search for an element
  `def search_tree(p:TreeNode, x:object) -> TreeNode:…`

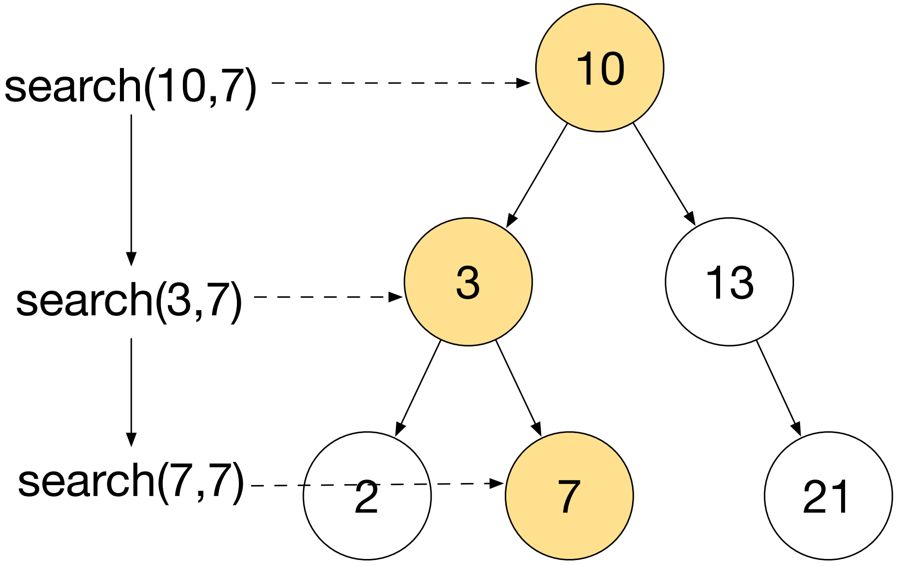
```
def search_tree(p:TreeNode, x:object) -> TreeNode:
    if p is None: return None
    if x==p.value: return p
    q = search_tree(p.left, x)
    if q is not None: return q
    return search_tree(p.right, x)
```

What is T(n) for search_tree?

Now restrict to *binary search tree* structure

- **Exercise**: go to “Constructing Binary Search Tree” section of notebook linked at bottom; try creating different trees
- Restricted walk: search using node values

```
def search(p:TreeNode, x:object):
    if p is None: return None
    if x<p.value:
        return search(p.left, x)
    if x>p.value:
        return search(p.right, x)
    return p
```



https://github.com/parrt/msds689/blob/master/notes/walking.ipynb

Compare BST search to tree walk

- Conditional recursion; we only recurse to ONE child not both

```
def walk_tree(p:TreeNode):
    if p is None: return
    print(p.value)
    walk_tree(p.left)
    walk_tree(p.right)
```

```
def search(p:TreeNode, x:object):
    if p is None: return None
    if x<p.value:
        return search(p.left, x)
    if x>p.value:
        return search(p.right, x)
    return p
```

$T(n) = k + 2T(n/2)$

$T(n) = k + T(n/2)$

Manual graph construction

- **Exercise**: Go to "Constructing graphs" section of link below, play with graph construction code to build different graphs.
- (You need "pip install lolviz" to visualize.)

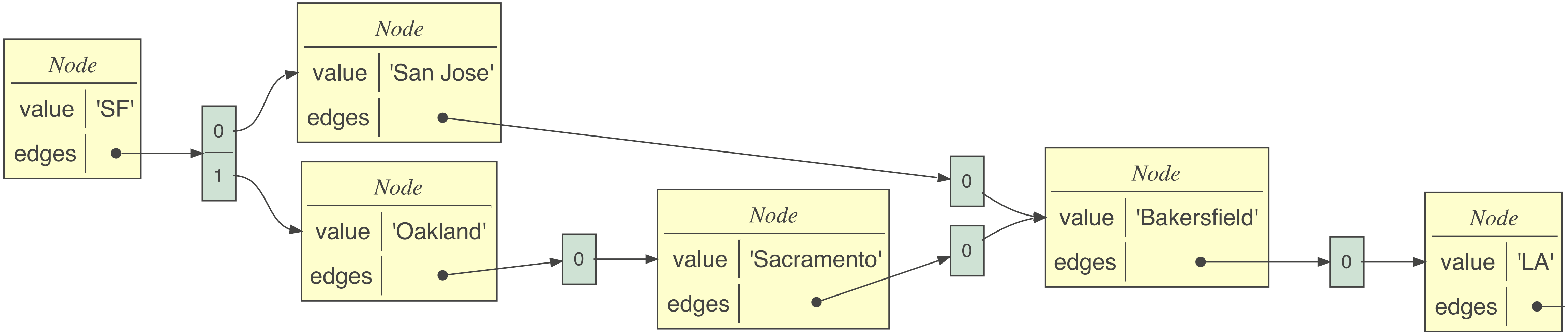
```
class Node:
  def __init__(self, value):
    self.value = value
    self.edges = []
  def add(self, target:Node):
    self.edges.append(target)
```

https://github.com/parrt/msds689/blob/master/notes/walking.ipynb

Depth-first graph walk*, compare to tree

```
def walk_graph(p:Node):
    if p is None: return
    print(p.value)
    for q in p.edges:
        walk_graph(q)
```

```
def walk_tree(p:TreeNode):
    if p is None: return
    print(p.value
    walk_tree(p.left)
    walk_tree(p.right)
```



*This function is missing a key bit

Depth-first graph walk avoiding cycles

- Maintain a set of already seen nodes; mark nodes as we encounter them and add “gate” at start of function

```
seen=set() # naughty but simple
def walk_graph2(p:Node) -> None:
    if p is None: return
    if p in seen: return
    seen.add(p)
    print(p.value)
    for q in p.edges:
        walk_graph2(q)
```

- walk_graph2() should take **seen** as parameter

Summary

- Walking data structures is fundamental to most algorithms
- You should be able to walk arrays, link lists, trees, and graphs
- Algorithms tend to be restricted or even repeated walks
- In the context of walking data structures, dynamic programming or memoization means recording partial results to avoid parts of the structure
- Binary tree and graph walks are very similar in code, but have to transition to more children and must deal with cycles
- Use recursion to walk trees and graphs