Walking data structures

Terence Parr
MSDS program
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
- An algorithm then typically computes something during the walk and often avoids part of the data structure to reduce computation time
- With dynamic programming (caching), algorithms can often avoid repeated, redundant computations to dramatically improve complexity

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
- 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

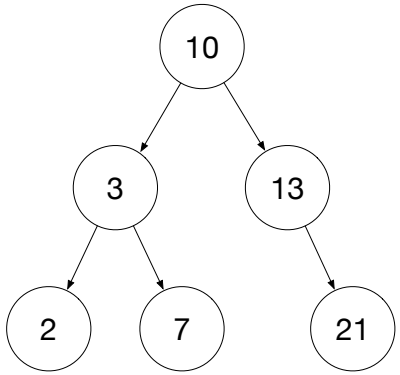
- Link https://goo.gl/i68EzJ uses pythontutor.com to visualize a pointer walking through a linked list
- You can step forward and backward with pythontutor.com
- Now, write a while-loop to walk from head to tail using pointer p, printing the value field at each node
- Write that code until you can do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

treverse

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

Trees: Recursive walk is most natural

reach evry node in the tree in one direction



- "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

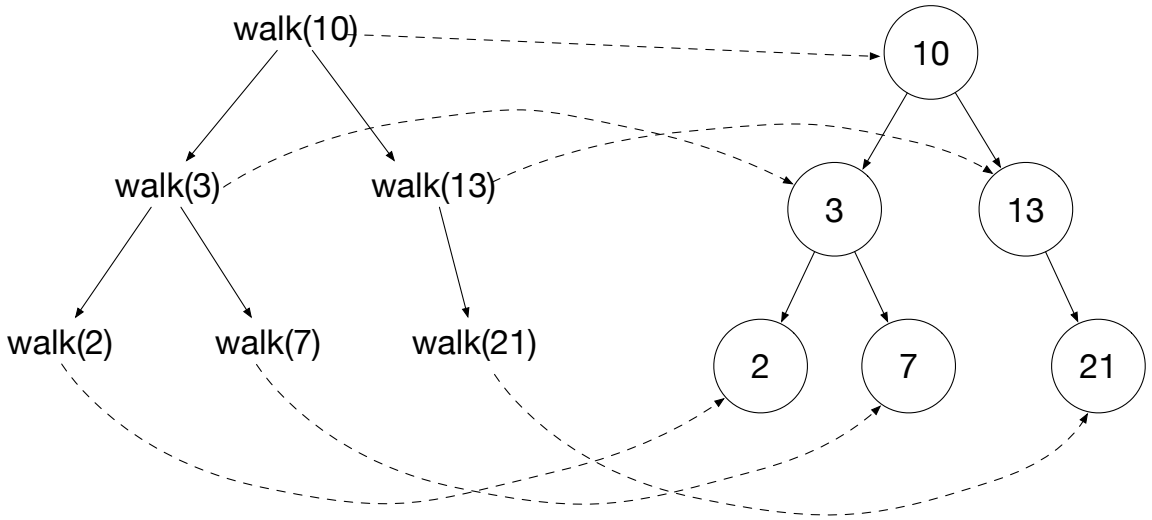
```
def walk_tree(p:TreeNode):
    if p is None: return
    print(p.value) # preorder
    walk_tree(p.left)
    walk_tree(p.right)
```

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

preorder: 10, 3, 2, 7, 13, 21
postorder: 2, 7, 3, 21, 13, 10

Recursion tree vs binary tree

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



Exhaustive search of all nodes

Search in binary tree

Exhaustive walk

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

Restricted walk for search

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

Ex: What is T(n) for search?

Compare binary tree walk with BST search

- 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)
```

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

```
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
```

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

Graphs: Manual construction

(we'll have full lecture on graphs later)

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

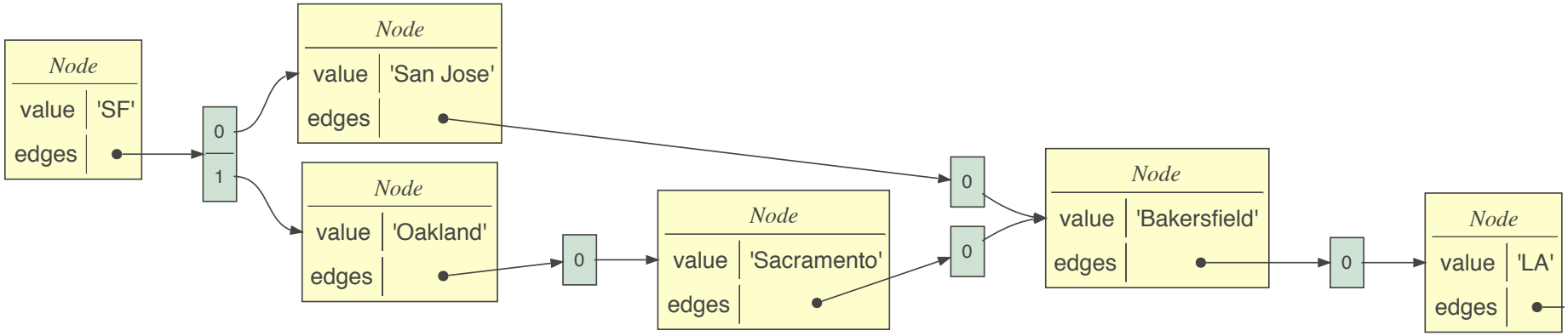
```
class Node:
  def __init__(self, value):
    self.value = value
    self.edges = []
  def add(self, target:Node):
    self.edges.append(target)
```

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

```
visited=set() # naughty but simple
def walk_graph2(p:Node) -> None:
    if p is None or p in visited: return
    visited.add(p) # must be before recursion step
    print(p.value)
    for q in p.edges:
        walk_graph2(q)
```

- (`walk_graph2()` should take **visited** as parameter)
- A form of dynamic programming where we record partial result "seen"

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 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 (can be slow in python)