Trees and Graphs

Basic Definitions

- Tree: Any connected, acyclic graph \( G = (V,E) \)
  - \( |E| = |V| - 1 \)
- \( n \)-ary Tree: Tree s/t all vertices of degree \( \leq n+1 \)
  - A “root” has degree \( \leq n \)
- Binary Search Tree: A binary tree such that
  - If node \( y \) is the left child of node \( x \), \( \text{key}[y] \leq \text{key}[x] \)
  - If node \( z \) is the right child of node \( x \), \( \text{key}[x] \leq \text{key}[z] \)
- Inorder Traversal (for Binary Trees): Recursively process left child, process root, recursively process right child
  - Takes \( \Theta(|V|) \) time
  - Used with a BST to print values in sorted order
  - Other traversal methods include preorder and postorder

Searching a BST

Tree-Search(x, k)
if \( x = \text{NIL} \) or \( \text{key}[x] = k \) then return \( x \)
if \( k < \text{key}[x] \)
  return Tree-Search(left[x], k)
return Tree-Search(right[x], k)

- Eliminating tail recursion:

Tree-Search(x, k)
do
  if \( x = \text{NIL} \) or \( \text{key}[x] = k \) then return \( x \)
  if \( k < \text{key}[x] \) then \( x = \text{left}[x] \) else \( x = \text{right}[x] \)
loop
- What is the runtime?

Other operations on BSTs

- Some other easy operations to perform include:
  - Minimum
    - Left most node
  - Maximum
    - Symmetric to minimum
  - Successor
  - Minimum of the key’s right subtree if it exists, otherwise first ancestor such that the key lies in it’s left subtree
  - Predecessor
    - Symmetric to successor
  - What are the runtimes?

Inserting / Deleting BST Nodes

- Insert(z): Search down the tree from the root… from a node \( x \), move left if \( \text{key}[z] < \text{key}[x] \), else move right. Insert \( z \) in the first empty location encountered.
- Delete(z): If \( z \) has less than 2 children, splice it out. If \( z \) has 2 children, find node \( y = \text{Successor}(z) \) and splice \( z \) out, then replace \( z \) with \( y \).
  - Note that \( y \) cannot have 2 children, so it can always be spliced (this is because \( y \) will be the minimum element in \( z \)’s right subtree by definition of Successor, thus \( y \) has no left child).

Expected Case for a BST

- Assume \( n \) nodes are inserted in random order. How many comparisons are performed?
- Consider the first arrival (the root). How many subsequent nodes go to the left/right of it?
  - Root is the \( k \)-th smallest with probability \( 1/n \)
  - The root acts like the “pivot” in QuickSort, all elements are compared against with it, then sent left or right
- How many total comparisons?
  \[ T(n) = n-1 + \left( \frac{1}{n} \right) \sum_{i=1}^{n} \left[ T(i-1) + T(n-i) \right] \]
- We have solved this before: \( T(n) = O(n \log n) \)
2-3 Trees

- A “balanced” search tree, with special properties:
  - All internal nodes have 2 or 3 children
  - All leaves are at the same depth
  - Plus standard search tree property (ordered subtrees)
  - “Guides” can be used to improve efficiency
  - Each node remembers the maximum key in its subtree
  - \( O(\log n) \) Search/Insert/Delete operations
  - See additional notes in handout for details
  - I will demonstrate in class
  - Useful as a “Dictionary” data structure

General Directed Graphs

- May contain cycles, which often need to be handled as a special case
- Adjacency Matrix vs Adjacency List representation
  - Matrix requires \( \Theta(|V|^2) \) space
    - Constant time to search for existence of an edge
  - List requires \( \Theta(|V| + |E|) \) space
    - Needs \( O(|E|) \) time in the worst case to search an edge
- Traversal techniques:
  - Breadth First Search (BFS)
  - Depth First Search (DFS)

BFS

- Given graph \( G = (V,E) \) and a source vertex \( s \in V \):
  - \( \text{BFS}(G, s) \)
    - for all \( x \in V \) \( \text{color}[x] = \text{WHITE}, \ d[x] = \infty, \ n[x] = \text{NIL} \)
    - \( Q = \emptyset \) \( \text{// Initialize an empty FIFO queue Q} \)
    - \( d[s] = 0, \ Enqueue(Q, s) \)
    - \( Q = \emptyset \)
    - for all \( v \in \text{Adj}[u] \)
      - if \( \text{color}[v] = \text{WHITE} \)
        - \( \text{color}[v] = \text{GRAY} \)
      - \( d[v] = d[u] + 1, \ n[v] = u, \ Enqueue(Q, v) \)
    - \( \text{color}[u] = \text{BLACK} \)

- Runtime? \( O(|V| + |E|) \)
- Useful shortest path property

DFS

- Given graph \( G = (V,E) \) and a source vertex \( s \in V \):
  - \( \text{DFS-VISIT}(u) \) \( \text{// Not the full DFS procedure (see CLRS p. 541) \}
  - \( \text{// Assume nodes } x \text{ were originally initialized with} \)
  - \( \text{// color}[x] = \text{WHITE}, \ n[x] = \text{NIL}, \text{ and that global variable time = 0} \)
  - \( \text{color}[u] = \text{GRAY} \)
  - \( \text{time} = \text{time} + 1 \)
  - for all \( v \in \text{Adj}[u] \)
    - if \( \text{color}[v] = \text{WHITE} \)
      - \( \text{color}[v] = u \)
    - \( \text{DFS-VISIT}(v) \)
  - \( \text{color}[u] = \text{BLACK}, \text{ time} = \text{time} + 1 \)
  - \( f[u] = \text{time} \)

- Runtime? \( O(|V| + |E|) \)

Observations about DFS

- DFS can also be implemented by replacing the FIFO queue in BFS with a LIFO stack
- “Parenthesis structure” (see CLRS p. 543)
  - Classification of edges. Edge \( (u,v) \) in a tree \( T \) is a…
    - Tree edge if \( (u,v) \in T \)
      - In a DFS tree, this means \( v \) is first visited coming from \( u \)
    - Forward edge if \( u \) is an ancestor of \( v \) but \( (u,v) \not\in T \)
      - In a DFS tree, means \( v \) was first visited by a descendent of \( u \)
    - Back edge if \( v \) is an ancestor of \( u \) in \( T \)
      - Exists if and only if there is a cycle in \( G \)
    - Cross edge if it is none of the above

Topological Sort

- A topological sort of a directed acyclic graph \( G = (V,E) \) is an ordering of \( V \) such that if \( (u,v) \in E \)
  then \( u \) appears before \( v \) in the sort
- DFS the graph \( G \), and sort \( V \) in order of decreasing finishing time (i.e. last to be “blackened” is first)
- Runtime is \( \Theta(|V|+|E|) \) for DFS + \( O(|V|) \) time to maintain a stack containing blackened vertices
**Strongly Connected Components**

- A strongly connected component is a *maximal* set of vertices \( C \subseteq V \) such that for any pair of distinct vertices \( u \) and \( v \) in \( C \), there is a path from \( u \) to \( v \) and a path from \( v \) to \( u \) in \( G \).
- Find SCCs by doing a complete DFS of \( G \) to get a topological sort of \( V \), and DFSing the *transpose* graph of \( G \) in the topologically sorted order.
  - Be sure to read details in CLRS, I will only sketch the procedure and proof in class.