CSE331 Introduction to Algorithms
Lecture 20: Review of Graph Algorithms and Data Structures I

Antoine Vigneron
antoine@unist.ac.kr

Ulsan National Institute of Science and Technology

July 23, 2021

Introduction

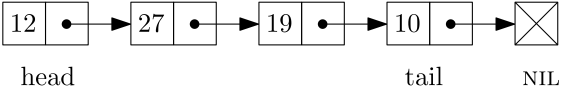
- This lecture and the next one will be a review of data structures and algorithms that were presented in the data structures course.
- Topics: linked lists, stacks, queues, graph traversals (BFS, DFS).
- **Reference**: Section 10 and 22 of the textbook

  Introduction to Algorithms by Cormen, Leiserson, Rivest and Stein.
- I will not be following this textbook closely in this lecture.

Arrays

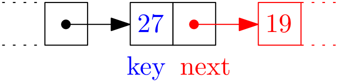
- Array $A[1 \ldots n]$ is created in $O(n)$ time.
- We can access element $A[i]$ at any index i in $O(1)$ time
  - This is called *random access*
- 2-dimensional array: $B[1 \ldots m, 1 \ldots n]$
- Access $B[i,j]$ in $O(1)$ time, create array in $O(mn)$ time
- Generalizes to any dimension
- Remark: sometimes arrays are considered to be created in $O(1)$ time at compilation. There is no definite answer to this, but in any case our bounds $O(n)$ and $O(mn)$ are sufficient in most applications as they do not dominate the running time.

Linked Lists



Node

- next — *reference to next node*
- key — *for searching*
- (data) — *satellite data*



- A list L is given by its first node L.head
- The data field records data that does not play a role in the data structure operations. We will not mention it in the rest of this lecture.

Linked Lists: Insertion and Deletion

Insertion at the head of a list

procedure $\textsc{InsertHead}$(list L, node ν)
    ν.next $\leftarrow$ L.head
    L.head $\leftarrow$ ν

Deletion from the head of a list

procedure $\textsc{DeleteHead}$(list L)
    $\nu \leftarrow$ L.head
    L.head $\leftarrow$ ν.next
    return ν

- Both operations take time $O(1)$.

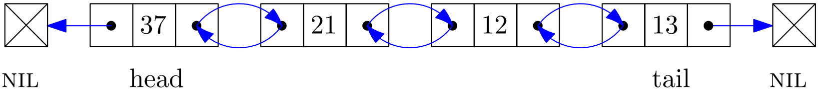
Linked Lists: Search

Searching a linked list

procedure $\textsc{Search}$(list L, key k)
    $\nu \leftarrow L$.head
    while $\nu \neq \text{NIL}$ and ν.key $\neq k$ **do**
        $\nu \leftarrow \nu$.next
    return ν

- Finding an element in a list of size n takes $O(n)$ time.
- No random access: accessing/inserting/deleting an element in the middle of the list takes $\Theta(n)$ time.

Doubly Linked Lists



Node

- next *reference to next node*
- prev *reference to previous node*
- key
- (data) *satellite data*

List

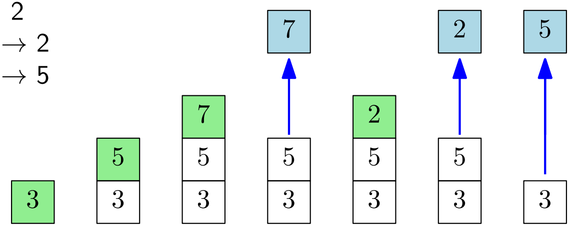
- head *reference to the head node*
- tail *reference to the tail node*

Doubly Linked Lists

- Operations:
  - Insert/delete element at the head or tail: $O(1)$ time.
  - Search for an element in a list of size n in $O(n)$ time.
  - Delete/insert element at any location in $O(n)$ time.
- Drawback: compared with singly linked lists, space usage increases by a constant factor.

Stacks

- A *stack* is an *abstract data type* with two operations:
  - push: insert an element
  - pop: remove from the stack the most recently inserted element
- Example:
  - start with empty stack
  - push 3
  - push 5
  - push 7
  - pop $\rightarrow$ 7
  - push 2
  - pop $\rightarrow$ 2
  - pop $\rightarrow$ 5



Stacks

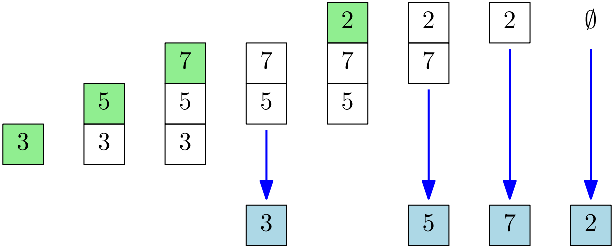
- This is called *LIFO*: last in, first out.

- A stack can be implemented with a linked list.
- Then each operation takes $O(1)$ time.
  - push: insert at the head
  - pop: delete from the head

- We can also use an array, where the last element is the top of the stack, and we keep track of its index. Operations still take $O(1)$ time.

Queues

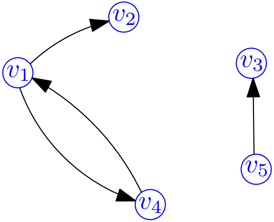
- A *queue* is an abstract data type with two operations:
  - enqueue: insert an element
  - dequeue: remove from the queue the earliest inserted element
- Example:
  - start with empty queue
  - enqueue 3
  - enqueue 5
  - enqueue 7
  - dequeue $\rightarrow$ 3
  - enqueue 2
  - dequeue $\rightarrow$ 5
  - dequeue $\rightarrow$ 7
  - dequeue $\rightarrow$ 2



Queue

- This is called *FIFO*: first in, first out.
- A queue can be implemented with a doubly linked list.
- Then each operation takes $O(1)$ time.
- Can also be implemented with a singly linked list, by keeping a pointer to the tail of the list.
- We can also use an array, seen as a circular list, and keep track of the index of the head and tail.

Directed Graphs



$V = \{v_1, v_2, v_3, v_4, v_5\}$

$n = 5$

$E = \{(v_1, v_2), (v_1, v_4), (v_4, v_1), (v_5, v_3)\}$

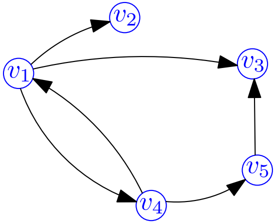
$m = 4$

Directed graphs

A *directed graph* $G(V, E)$ consists of a set V of *vertices* and a set $E \subset V \times V$ of *edges*.

- So an edge is an *ordered pair* of vertices.
- A vertex may also be called a *node*.
- Usually, the number of vertices is denoted $n = |V|$ and the number of edges is denoted $m = |E|$.

Adjacency Lists



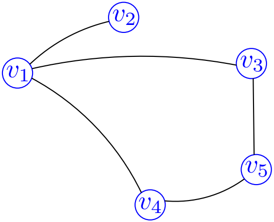
$L(v_1) = \{v_2, v_3, v_4\}$
$L(v_2) = \emptyset$
$L(v_3) = \emptyset$
$L(v_4) = \{v_1, v_5\}$
$L(v_5) = \{v_3\}$

Adjacency lists

The *adjacency list* $L(v_i)$ of v_i is the set of vertices v_j such that $(v_i, v_j) \in E$. These vertices v_j are called the *neighbors* of v_i, and are said to be *adjacent* to v_i.

- So a directed graph can be represented by a list of vertices, and an adjacency list for each vertex.

Undirected Graphs



$V = \{v_1, v_2, v_3, v_4, v_5\}$

$E = \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_1, v_4\}, \{v_3, v_5\}, \{v_4, v_5\}\}$

$L(v_1) = \{v_2, v_3, v_4\}$

$L(v_2) = \{v_1\}$ $\qquad$ $L(v_4) = \{v_1, v_5\}$

$L(v_3) = \{v_1, v_5\}$ $\qquad$ $L(v_5) = \{v_3, v_4\}$

Directed graphs

An *undirected graph* $G(V, E)$ consists of a set V of *vertices* and a set E of *edges*. Each edge is an *unordered* pair of vertices.

- Two vertices v_i, v_j are said to be adjacent, or neighbors, if $\{v_i, v_j\}$ is an edge.
- We can also represent an undirected graph using adjacency lists.

Depth-First Search (DFS)

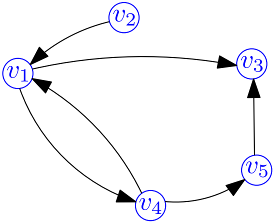
- *Depth-first search* (DFS) is an algorithm that, starting from a node s, finds all the nodes v such that there is a path from s to v in the graph.
- Initially, all nodes are *unmarked*.
- Then we call DFS(s).

Pseudocode

procedure DFS(node u)
    mark u
    for each $v \in L(u)$ **do**
        if v is unmarked **then**
            DFS(v)

- It applies to directed and undirected graphs.

Example



$$L(v_1) = \{v_3, v_4\}$$
$$L(v_2) = \{v_1\}$$
$$L(v_3) = \emptyset$$
$$L(v_4) = \{v_1, v_5\}$$
$$L(v_5) = \{v_3\}$$

- Suppose we run DFS from v_4.
- Then nodes v_1, v_3, v_5 are visited in this order.
- v_2 remains unmarked.

Analysis

Proposition

DFS runs in $O(n + m)$ time.

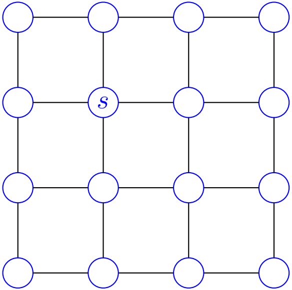
Proof.

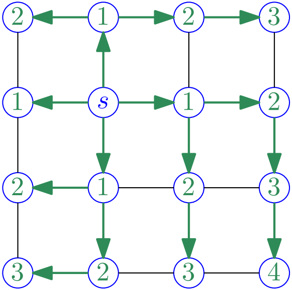
We need $O(n)$ time to unmark all vertices. Then DFS is called at most once for each edge (twice for undirected graphs). □

Applications of DFS

- DFS(s) as we presented it marks all vertices that are reachable from s.
- It can be used for other purposes if we perform other operations at line 2 or 5.
- For instance, we can return the set of nodes reachable from s, or their number, or the whole subgraph reachable from s.
- or given s and t, we can decide whether there is a path from s to t.

Breadth-First Search (BFS)





- *Breadth-first search* (BFS) visits the same set of nodes as DFS, but in a different order.
- In addition, it computes:
  - The distance from s to all visited nodes.
  - A tree T rooted at s, such that the shortest path from s to all nodes within T is also a shortest path in G.

Breadth-First Search (BFS)

Pseudocode

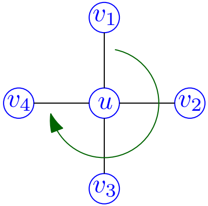
procedure $\mathrm{BFS}(G(V,E),\ s \in V)$
    $Q \leftarrow$ new queue containing only s
    $T \leftarrow$ empty tree $T(V,\emptyset)$
    $d \leftarrow$ array of integers
    unmark all nodes
    mark s
    $d(s) = 0$
    while Q is nonempty **do**
        $u \leftarrow Q.\mathrm{dequeue}$
        for each $v \in L(u)$ **do**
            if v is unmarked **then**
                mark v
                enqueue v
                add edge (u,v) to T
                $d(v) \leftarrow d(u) + 1$ ▷ distance from s to u

Breadth-First Search (BFS)

Remark

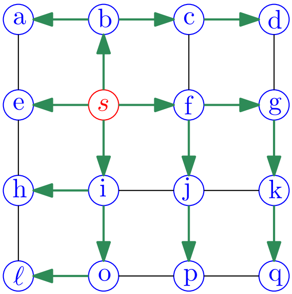
The order in which the vertices are traversed, and the tree T, depend on the ordering of the adjacency lists. The figure in Slide 21 corresponds to adjacency lists being in clockwise order for each vertex.

$$L(u) = \{v_1, v_2, v_3, v_4\}$$



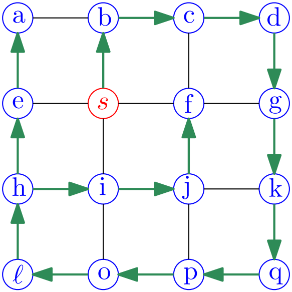
Comparison BFS vs DFS

BFS



Nodes are visited in the order:
s, b, f, i, e, c, g, j,
o, h, a, d, k, p, ℓ, q

DFS



Nodes are visited in the order:
s, b, c, d, g, k, q, p,
o, ℓ, h, e, a, i, j, f

Breadth-First Search (BFS)

- Proof of correctness (sketch): The queue ensures that nodes are visited by nondecreasing distance from s.
- Analysis: Each node and edge is visited once, so

Proposition

BFS runs in $O(m+n)$ time.

- DFS was implemented recursively and BFS iteratively.
- How can we implement DFS iteratively? (See exercise set.)