Question:

Let A be an adjacency list. What is the total number of keys stored in A?

A. $\Theta(|V|)$  
B. $\Theta(|V|^2)$  
C. $\Theta(|E|)$  
D. $\Theta(\max\{\deg(v)\})$  
E. $\Theta(|V| + |E|)$
Trees

A cycle in a graph is a path, such that the first and last vertex is the same.

An undirected (connected) graph is a tree if it doesn’t contain any cycles.

We can designate a node in the tree to be the root.
  • we use the term parent, child, ancestor in this context. (e.g. parent = the last node on the path from the root to the child.)
BFS tree

• Trees define a path between nodes.
• What is the relationship between the paths in the BFS tree(s) and the distance of vertices from s?
Algorithm 1: BFS(G,s)

1. parents ← {} /* empty hash table, parents[v] = v’s parent. */
2. dist ← {} /* empty hash table, dist[v] = distance from s. */
3. Q ← empty FIFO queue/* keep track of active nodes */
4. Q.enqueue(s), parents[s] = None, dist[s] = 0 /* initialization */
5. while Q is not empty do
6.  u ← Q.dequeue();
7.  for v in G[u] do /* explore neighbors of active node u */
8.      if v not in parents then /* v was so far undiscovered */
9.          parents[v] = u;
10.         dist[v] = dist[u] + 1;
11.         Q.enqueue(v);
12. return parents, dist

notation:
• G[u] returns the neighbor list of node u
• parents[v], dist[v] returns the value (node ID, number) stored at index v
BFS - running time analysis

Algorithm 1: BFS(G,s)

/* G is hash table, the adjacency list of a graph */
/* s is a source vertex in G */

1. parents ← {}          /* empty hash table, parents[v] = v’s parent. */
2. dist ← {}             /* empty hash table, dist[v] = distance from s. */
3. Q ← empty FIFO queue   /* keep track of active nodes */
4. Q.enqueue(s), parents[s] = None, dist[s] = 0 /* initialization */
5. while Q is not empty do
6.     u ← Q.dequeue();
7.     for v in G[u] do  /* explore neighbors of active node u */
7.         if v not in parents then /* v was so far undiscovered */
8.             parents[v] = u;
9.             dist[v] = dist[u] + 1;
10.            Q.enqueue(v);
11.       end if
12.     end for
13. return parents, dist

Θ(n) + \sum_{i=1}^{E} \delta(u) \Rightarrow \Theta(|V| + |E|)
BFS

- BFS: a single source shortest paths algorithm, e.g. returns the distance from a node \( s \) to each other node \( v \).
Select all that are true when running BFS on graph G from source s.

A. The BFS search tree is unique.
B. When running BFS on G twice with different processing order the values in dist[ ] (i.e. the computed distance values) are identical in the two runs.
C. The shortest path from s to a node v is always unique.
D. If there is an edge connecting nodes u and v then their distances from s cannot be the same.
E. The path from s to v in the BFS tree (i.e. following the edges of the tree from s to v) is a shortest path.
Theorem. For a node $v$ the length of the shortest path connecting $s$ to $v$ is equal to the layer of $v$ in BFS.

In consequence BFS should return
- the layer of each node, so that we get the distances
- the BFS tree, since they encode the shortest paths themselves
Theorem. For a node $v$ the length of the shortest path connecting $s$ to $v$ is equal to the layer of $v$ in BFS.

Proof.

This is the BFS tree associated with $G$. 
Breadth first search - correctness (multiple slides)

**Theorem.** For a node $v$ the length of the shortest path connecting $s$ to $v$ is equal to the layer of $v$ in BFS.

**Proof.**
Breadth first search - correctness (multiple slides)

**Theorem.** For a node $v$ the length of the shortest path connecting $s$ to $v$ is equal to the layer of $v$ in BFS.

**Proof.**

**Proposition:** any two connected nodes are either in the same or consecutive layers.

Note that the proposition applies to every edge in $G$, both those in the BFS tree and those that are not.
Breadth first search - correctness (multiple slides)

Proposition: any two connected nodes are either in the same or consecutive layers.

Proof:
- For an edge \((u,v)\) without loss of generality we may assume that \(u\) was discovered first.
- If \(v\) is in the same layer as \(u\), the proposition is true
- if \(v\) is not in the same layer, then by the assumption it hasn’t been discovered yet
- by the design of BFS \(v\) is an undiscovered neighbor of \(u\), hence is assigned to the next layer.
Theorem. For a node $v$ the length of the shortest path connecting $s$ to $v$ is equal to the layer of $v$ in BFS.

Proof.

Clearly, there is a path from $s$ to $v$ of the same length as the layer of $v$.

- the one implied by the BFS tree

Is it possible that there is a path from $s$ to $v$ with fewer edges?
**Theorem.** For a node \( v \) the length of the shortest path connecting \( s \) to \( v \) is equal to the layer of \( v \) in BFS.

**Proof.**

For purpose of contradiction, suppose there is a shorter path. That path would have to bypass at least one layer.

This is in contradiction with the proposition that neighbors are at most one layer apart. QED
Algorithm 1: BFS(G,s)

/* G is hash table, the adjacency list of a graph */
/* s is a source vertex in G */
1 parents ← {} /* empty hash table, parents[v] = v’s parent. */
2 dist ← {} /* empty hash table, dist[v] = distance from s. */
3 Q ← empty FIFO queue /* keep track of active nodes */
4 Q.enqueue(s), parents[s] = None, dist[s] = 0 /* initialization */
5 while Q is not empty do
6    u ← Q.dequeue();
7    for v in G[u] do /* explore neighbors of active node u */
8       if v not in parents then /* v was so far undiscovered */
9          parents[v] = u;
10         dist[v] = dist[u] + 1;
11         Q.enqueue(v);
12    return parents, dist

Exercise.
• Given the parents table reconstruct the BFS tree
• For a node v find the path from v back to s. (This is called backtracking)
• do both in O(n) time
**Connected component**

**Def.** An undirected graph is **connected** if for every pair of nodes $u$ and $v$, there is a path between $u$ and $v$.

The **connected components** of a graph are its connected subgraphs. (An analogous concept for directed graphs are the strongly connected components.)

circle 1

```
connected component graph with 3 connected components:
{1,2,3,4,5,6,7,8}
{9,10}
{11,12,13}
```
Design an algorithm to find the connected components of an undirected graph.

1. Pick a random node \( v \) and run BFS from \( v \). The nodes that are discovered will be part of \( v \)'s component.
2. Pick another \( w \) at random that is not discovered yet.
3. Go back to step 1, using \( w \) instead of \( v \).
Connected component

Find an algorithm to find all the connected components in G.

Algorithm: repeatedly run BFS from a yet undiscovered node to find the next connected component.

Running time: still O(n+m)
Connected component - TopHat

Design an algorithm to find the connected components of an undirected graph.

What is the running time of this algorithm on a graph with n nodes, m edges and k connected components?

A. $\Theta(n(n + m))$

B. $\Theta(k(n + m))$

C. $\Theta(n + m)$
Ariadne’s thread in logic

Theseus and the Minotaur
- Greek mythology

Ariadne’s thread
- principle in logic
- solving a problem through exhaustive application of logic through all available routes
- key element: maintain a record with all available and all exhausted options (the record is sometimes referred to as the ‘thread’)
- record is kept for the purpose of backtracking - reverse earlier decisions and try alternatives
Depth First Search (DFS) — high level description

Let $G(V,E)$ be a directed graph.

**Depth First Search (DFS):** Graph traversal algorithm. It explores the entire graph

- visits all reachable nodes in a graph
- “depth first” — traverse the furthest away from the source first
- recursive in nature

- output:
  - DFS tree
  - timestamps (discovery time, finish time). Used for applications.

(Recommended reading on DFS CLRS chapter 22.3)
source: a

node states: unexplored, discovered, finished
timestamps:
  • discovery time - time of first visit
  • finish time - time of last visit
DFS vs. BFS example

source: a-b-d-e-i-f

a-f
DFS recursive pseudocode

Algorithm 1: DFSwrapper($G, s$)

1 /* $G$ is the adjacency list of a graph */
2 /* $s$ is a source node */
3 discovered $\leftarrow$ empty set /* ids of discovered nodes */
4 parents $\leftarrow$ hash table /* ids of parents in DFS tree */
5 times $\leftarrow$ hash table /* times[u] = tuple <discovery, finish> */
6 $t \leftarrow -1$ /* counter */
7 discovered.add($s$), parents[$s$] = None;
8 Return DFS($G, s$)

Algorithm 2: DFS($G, u$)

1 discovered.add($u$);
2 $t = t + 1$;
3 times[$u$][0] = $t$;
4 for $v$ in $G[u]$ do
5 /* recursively explore $u$’s neighbors */
6 if $v$ not in discovered then
7 parents[$v$] = $u$;
8 DFS($G, v$);
9 $t = t + 1$;
10 times[$u$][1] = $t$;

Exercise. write and iterative implementation of DFS using stacks.
DFS recursive — runtime of recursive algorithm

**Algorithm 1: DFSwrapper**(G, s)

1 /* G is the adjacency list of a graph */
2 /* s is a source node */
3 discovered ← empty set /* ids of discovered nodes */
4 parents ← hash table /* ids of parents in DFS tree */
5 times ← hash table/* times[u] = tuple <discovery, finish> */
6 t ← − 1 /* counter */
7 discovered.add(s), parents[s] = None;
8 Return DFS(G, s)

**Algorithm 2: DFS(G, u)**

1 discovered.add(u);
2 t = t + 1;
3 times[u][0] = t;
4 for v in G[u] do /* recursively explore u’s neighbors */
5     if v not in discovered then
6         parents[v] = u;
7         DFS(G, v);
8     t = t + 1;
9     times[u][1] = t;
DFS recursive — runtime of recursive algorithm

How many recursive calls to DFS total?

One call of DFS for every node in \( G \)

\[ \Theta(n) \] calls

Runtime of operations done within the current DFS call?

\[ \Theta(\delta(u)) \]

Algorithm 2: DFS\((G, u)\)

1. discovered.add\((u)\);
2. \( t = t + 1 \);
3. times\([u][0] = t; \)
4. for \( v \) in \( G[u] \) do
   /* recursively explore \( u \)'s neighbors */
   5. if \( v \) not in discovered then
   6.      parents\([v] = u; \)
   7.      DFS\((G, v); \)
   8. \( t = t + 1; \)
   9. times\([u][1] = t; \)

\[ \Theta(n) + \sum_{i=1}^{n} \Theta(\delta(u)) = \Theta(|V| + |E|) \]