Searching Linked Lists

Assume the list is sorted, but is stored in a linked list.

Can we use binary search?
- Comparisons?
- "Work?"

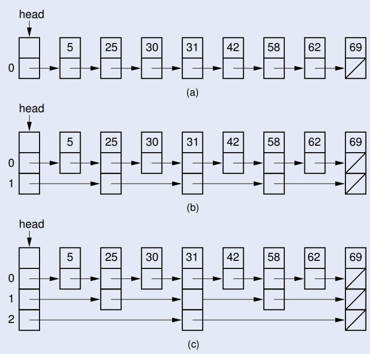
What if we add additional pointers?

Same. Is this a good model? No.

Much higher since we must move around a lot (without comparisons) to get to the same position.

Might get to desired position faster.

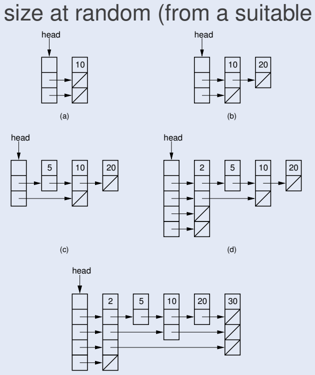
"Perfect" Skip List



What is the access time? log n.
We can insert/delete in log n time as well.

Building a Skip List

Pick the node size at random (from a suitable probability distribution).



no notes

Skip List Analysis (1)

What distribution do we want for the node depths?

```
int randomLevel(void) { // Exponential distrib
  for (int level=0; Random(2) == 0; level++);
  return level;
}
```

What is the worst cost to search in the "perfect" Skip List?

What is the average cost to search in the "perfect" Skip List?

What is the cost to insert?

What is the average cost in the "typical" Skip List?

Exponential decay. 1 link half of the time, 2 links one quarter, 3 links one eighth, and so on.

log n.

Close to log n.

log n.

log n.

Skip List Analysis (2)

How does this differ from a BST?

- Simpler or more complex?
- More or less efficient?
- Which relies on data distribution, which on basic laws of probability?

About the same.

On average, about the same *if* data are well distributed.

BST relies on data distribution, while skiplist merely relies on chance.

Probabilistic Quicksort

Quicksort runs into trouble on highly structured input.

Solution: Randomize input order.

- Chance of worst case is then $2/n!$.

This principle is why, for example, the Skip List data structure has much more reliable performance than a BST. The BST's performance depends on the input data. The Skip List's performance depends entirely on chance. For random data, the two are essentially identical. But you can't trust data to be random.

Random Number Generators

- Most computers systems use a deterministic algorithm to select **pseudorandom** numbers.
- **Linear congruential method**:
  - Pick a **seed** $r(1)$. Then,

$$r(i) = (r(i-1) \times b) \bmod t.$$

- Must pick good values for b and t.
- Resulting numbers must be in the range:
- What happens if $r(i) = r(j)$?

Lots of "commercial" random number generators have poor performance because they don't get the numbers right.
Must be in range 0 to $t-1$.

They generate the same number, which leads to a cycle of length $|j-i|$.

Random Number Generators (cont)

Some examples:

$$\begin{aligned}
r(i) &= 6r(i-1) \bmod 13 = \\
&\quad \cdots 1, 6, 10, 8, 9, 2, 12, 7, 3, 5, 4, 11, 1 \cdots \\
r(i) &= 7r(i-1) \bmod 13 = \\
&\quad \cdots 1, 7, 10, 5, 9, 11, 12, 6, 3, 8, 4, 2, 1 \cdots \\
r(i) &= 5r(i-1) \bmod 13 = \\
&\quad \cdots 1, 5, 12, 8, 1 \cdots \\
&\quad \cdots 2, 10, 11, 3, 2 \cdots \\
&\quad \cdots 4, 7, 9, 6, 4 \cdots \\
&\quad \cdots 0, 0 \cdots
\end{aligned}$$

The last one depends on the start value of the seed.
Suggested generator: $r(i) = 16807r(i-1) \bmod 2^{31} - 1$

no notes

Graph Algorithms

Graphs are useful for representing a variety of concepts:

- Data Structures
- Relationships
- Families
- Communication Networks
- Road Maps

- A **graph** $G = (V, E)$ consists of a set of **vertices** V, and a set of **edges** E, such that each edge in E is a connection between a pair of vertices in V.
- Directed vs. Undirected
- Labeled graph, weighted graph
- Labels for edges vs. weights for edges
- Multiple edges, loops
- Cycle, Circuit, path, simple path, tours
- Bipartite, acyclic, connected
- Rooted tree, unrooted tree, free tree

A Tree Proof

- **Definition**: A **free tree** is a connected, undirected graph that has no cycles.
- **Theorem**: If T is a free tree having n vertices, then T has exactly $n - 1$ edges.
- **Proof**: By induction on n.
- **Base Case**: $n = 1$. T consists of 1 vertex and 0 edges.
- **Inductive Hypothesis**: The theorem is true for a tree having $n - 1$ vertices.
- **Inductive Step**:
  - If T has n vertices, then T contains a vertex of degree 1.
  - Remove that vertex and its incident edge to obtain T', a free tree with $n - 1$ vertices.
  - By IH, T' has $n - 2$ edges.
  - Thus, T has $n - 1$ edges.

This is close to a satisfactory definition for free tree. There are several equivalent definitions for free trees, with similar proofs to relate them.

Why do we know that some vertex has degree 1? Because the definition says that the Free Tree has no cycles.

Graph Traversals

Various problems require a way to **traverse** a graph – that is, visit each vertex and edge in a systematic way.

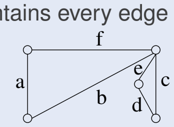
Three common traversals:

1. Eulerian tours
   Traverse each edge exactly once
2. Depth-first search
   Keeps vertices on a stack
3. Breadth-first search
   Keeps vertices on a queue

a vertex may be visited multiple times

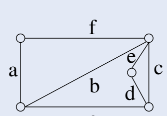
Eulerian Tours

A circuit that contains every edge exactly once.
Example:



Tour: b a f c d e.

Example:



No Eulerian tour. How can you tell for sure?

Why no tour? Because some vertices have odd degree.

All even nodes is a necessary condition. Is it sufficient?

Eulerian Tour Proof

- **Theorem**: A connected, undirected graph with m edges that has no vertices of odd degree has an Eulerian tour.
- **Proof**: By induction on m.
- **Base Case**:
- **Inductive Hypothesis**:
- **Inductive Step**:
  - Start with an arbitrary vertex and follow a path until you return to the vertex.
  - Remove this circuit. What remains are connected components G_1, G_2, ..., G_k each with nodes of even degree and $< m$ edges.
  - By IH, each connected component has an Eulerian tour.
  - Combine the tours to get a tour of the entire graph.

Base case: 0 edges and 1 vertex fits the theorem.
IH: The theorem is true for $< m$ edges.
Always possible to find a circuit starting at any arbitrary vertex, since each vertex has even degree.

Depth First Search

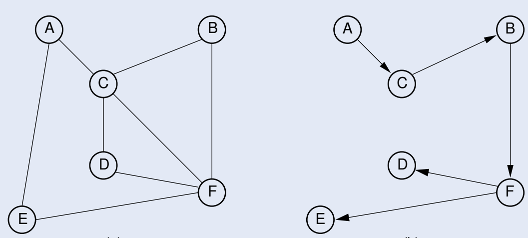
```
void DFS(Graph G, int v) { // Depth first search
  PreVisit(G, v);      // Take appropriate action
  G.setMark(v, VISITED);
  for (Edge w = each neighbor of v)
    if (G.getMark(G.v2(w)) == UNVISITED)
      DFS(G, G.v2(w));
  PostVisit(G, v);     // Take appropriate action
}
```

Initial call: `DFS(G, r)` where r is the **root** of the DFS.

Cost: $\Theta(|V| + |E|)$.

no notes

Depth First Search Example



(a) (b)

The directions are imposed by the traversal. This is the Depth First Search Tree.

DFS Tree

If we number the vertices in the order that they are marked, we get **DFS numbers**.

Lemma 7.2: Every edge $e \in E$ is either in the DFS tree T, or connects two vertices of G, one of which is an ancestor of the other in T.

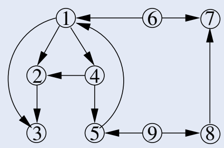
Proof: Consider the first time an edge (v, w) is examined, with v the current vertex.

- If w is unmarked, then (v, w) is in T.
- If w is marked, then w has a smaller DFS number than v AND (v, w) is an unexamined edge of w.
- Thus, w is still on the stack. That is, w is on a path from v.

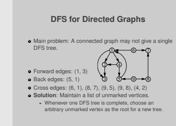
Results: No "cross edges." That is, no edges connecting vertices sideways in the tree.

DFS for Directed Graphs

- Main problem: A connected graph may not give a single DFS tree.



- Forward edges: (1, 3)
- Back edges: (5, 1)
- Cross edges: (6, 1), (8, 7), (9, 5), (9, 8), (4, 2)
- **Solution**: Maintain a list of unmarked vertices.
  - Whenever one DFS tree is complete, choose an arbitrary unmarked vertex as the root for a new tree.



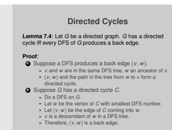
no notes

Directed Cycles

Lemma 7.4: Let G be a directed graph. G has a directed cycle iff every DFS of G produces a back edge.

Proof:

1. Suppose a DFS produces a back edge (v, w).
   - v and w are in the same DFS tree, w an ancestor of v.
   - (v, w) and the path in the tree from w to v form a directed cycle.
2. Suppose G has a directed cycle C.
   - Do a DFS on G.
   - Let w be the vertex of C with smallest DFS number.
   - Let (v, w) be the edge of C coming into w.
   - v is a descendant of w in a DFS tree.
   - Therefore, (v, w) is a back edge.



See earlier lemma.