CSI 402 – Lecture 6
(Table Organization Methods)
A variety of (large) tables used by assemblers (and compilers).

Efficiently maintaining tables is important.

**Dictionary:** A data structure that efficiently supports insertion, deletion and search operations.

Many ways of implementing dictionaries are known.

1. **Linked Lists:**
   - Each node stores one row of table. (List is not necessarily in order.)
   - Simple method – easy to insert a new table entry.
   - Search is slow. (For a list with $n$ nodes, on the average, $n/2$ nodes will be encountered during search.)
   - Suitable for small tables.
II. Self-Organizing Lists:

- Small modifications to sequential search to improve average search time.
- Rely on “program locality”. (Each phase of a program tends to reference a small collection of symbols.)
- **Idea:** Keep the more frequently accessed entries at the beginning of the list.
- Several heuristics are known.

(a) **Move-to-Front heuristic:**

- When a symbol is referenced, move the corresponding node to the front of the list.
- Average number of nodes during a search is $\approx \frac{2n}{\log_2 n}$.
- Better than sequential search as $n$ becomes larger.
(b) **Transpose heuristic:**

- When a symbol is referenced, exchange the corresponding node with its predecessor (if one exists) in the list.
- In practice, performance is similar to that of Move-to-Front.

(c) **Move-Ahead-k heuristic:**

- When a symbol is referenced, it is moved ahead in the list by $k$ positions.
- Generalizes Move-to-Front and Transpose heuristics.
- Difficult to choose an appropriate value of $k$.
- Value of $k$ is generally chosen as a percentage of the number of nodes in the list.
(d) **Count heuristic:**

- Add a count field to each node.
- When a node is inserted into the list, set the count to 1.
- When a symbol is referenced, increment the count by 1.
- Maintain list in **non-increasing order of count**.
- Needs additional space and time overhead compared to the other heuristics.
- In practice, search performance is not significantly better than that of the other heuristics.

**Examples for all the heuristics:** To be presented in class.
III. Ordered Tables:

- Search time can be improved (to $O(\log_2 n)$) using a sorted table.
- Good method for static tables; such tables need to be sorted just once.
- Symbol Tables used by two-pass assemblers can be sorted at the beginning of Pass 2.

IV. Binary Search Trees:

- Suitable for dynamic tables.
- Each node of the tree has
  - Data (symbol, LC value, etc.).
  - Pointer to left child.
  - Pointer to right child.
For each node containing symbol X:

- Symbols of all the nodes in the left subtree precede X in sorted order.
- Symbols of all the nodes in the right subtree follow X in sorted order.

Tree is “balanced”; for each node, the heights of the left and right subtrees are equal or almost equal.

For a balanced tree with $n$ nodes, the height is $O(\log_2 n)$.

Insert, delete and search operations can be done in $O(\log_2 n)$ time.
Sorted order of symbols can be obtained by an **inorder** traversal of the tree.

Rebalancing needed after insertions and deletions: time overhead.

Two pointers per node: space overhead.

**Examples:** To be presented in class.
IV. Hash Tables

- Most commonly used method in assemblers and compilers.
- Simple to implement; good performance in practice.
- **Idea:** Reduce search time by performing a small amount of computation on the key value (i.e., the string corresponding to the symbol).
- Hash Table (HT): Array of pointers.
- Hash Function \((h)\): Given a key, computes an index into HT.

**Example:** To be presented in class. (Examples of hash functions are given in Handout 6.1.)
IV. Hash Tables (continued)

- **Collision:** Hash function produces the same index value for two different keys. (Collisions can’t be avoided in practice.)

- **Chaining:** For each index \( i \), keys that hash to \( i \) are kept in a linked list pointed to by \( HT[i] \).

**Inserting a symbol \( X \) into \( HT \):**

**Note:** Assume that symbol \( X \) is not in \( HT \).

1. Let \( t = h(X) \).
2. Insert the node for \( X \) in the list pointed to by \( HT[t] \).

**Searching for a symbol \( X \) in \( HT \):**

1. Let \( t = h(X) \).
2. Search for \( X \) in the list pointed to by \( HT[t] \).
Performance of Hashing:

- If HT has $k$ pointers, a good hash function should distribute the $n$ keys so that each list has $\approx \frac{n}{k}$ keys.
- After computing the hash function, the sequential search will only examine $\frac{n}{k}$ nodes.
- Faster than sequential search by a factor of $k$.
- Studies have shown that $k$ should be a prime number for good performance.
Suggested Exercises

1. Make sure that you understand the different self-organizing search heuristics by doing additional examples.

2. Write C functions to implement the self-organizing search heuristics discussed in this lecture.

3. Construct examples of balanced binary search trees; make sure that you understand the three forms of binary tree traversals (namely, pre-order, in-order and post-order).

4. Write C functions to implement Insert and Search operations in a Hash Table. (In implementing the Insert operation, make sure to check whether the symbol is already in HT.)