505 22240 / ESOE 2012 Data Structures: Lecture 7
Hash Tables and Rooted Trees

§ Dictionaries

· Suppose you have a set of two-letter words and their definitions.

· Word is a key that addresses the definition.

· There are 26 × 26 = 676 words.

★Insert a *Definition* into dictionary:

⇨ We use function hashCode( ): maps each word (key) to integers 0 … 675, which are index to array.

⇨ Suppose the class *Definition* is already defined.

· Code:

```
class Word {
private:
     string word;
public:
     const int LETTERS = 26;
     const int WORDS = LETTERS * LETTERS;
     int hashCode( ) {
          return LETTERS * (word.substr(0, 1) - 'a') + (word.substr(1, 1) -
          'a');
     }
};
class WordDictionary {
private:
     Definition* defTable = new Definition[Word::WORDS];
public:
     void insert(Word* w, const Definition& d) {
          defTable[w->hashCode( )] = d;
     }
     Definition& find(Word* w) {
          return defTable[w->hashCode( )];
     }
};
```

· What if we store every English word?

· 26^{45} for all English words → impossible & impractical.

§ Hash Tables (the most common implementation of dictionaries)

· Parameter definitions:

n: number of keys (words) stored [several hundred thousands].

Table of N buckets. N a bit larger than n [same order].

· A hash table maps huge set of possible keys into N buckets by applying a compression function to each hash code, e.g.,

`h(hashCode) = hashCode mod N`

★Collision: Several keys hash to same bucket, if h(hashCode1) = h(hashCode2).

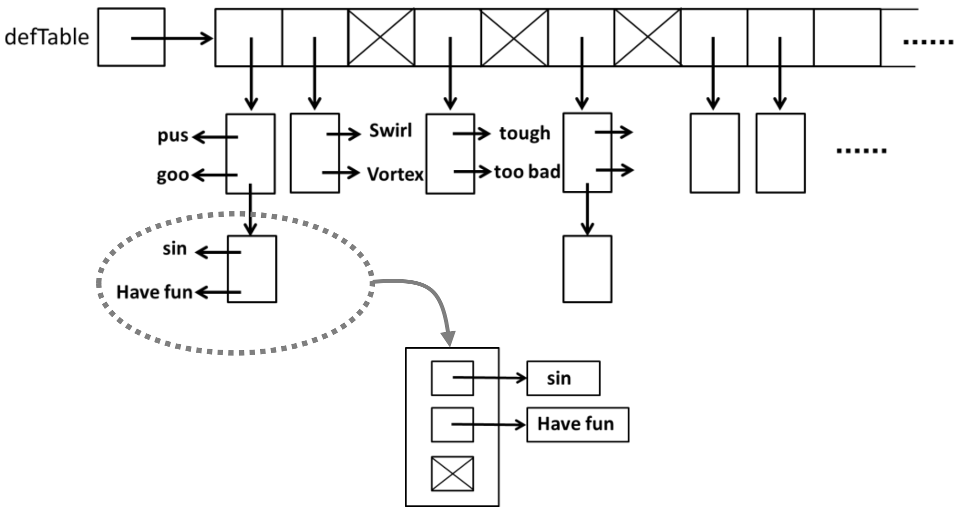
⇩

★Chaining: Each bucket references a linked list of entries, called a chain.

⇩

· Store each key in table with definition.

`Entry = (key, value)`, where key is English word and value is definition.



◎ Three Operations:

Entry insert(key, value):

- Compute the key's hash code.
- Compress it to determine bucket.
- Insert the entry into bucket's chain.

Entry find(key):

- Hash the key to determine bucket.
- Search chain for entry with given key.
- If found, return it; otherwise null.

Entry remove(key):

- Hash key.
- Search chain.
- Remove from chain if found.
- Return entry (found) or null.

★Two entries with same key: Two approaches

① We can insert both; find( ) arbitrarily return one, remove( ) delete all entries.

② Replace old value with new. Only one entry has given key.

★Load factor of a hash table: **n/N**

- If load factor stays low, and hash code & compression function are "good", and no duplicate keys, THEN the chains are short, and each operation takes $O(1)$ time.
- If load factor get BIG (n >> N), $O(n)$ time.

◎ Hash Codes & Compression Functions

Key ⟶ hashcode ⟶ [0, N-1]

- Ideal: Map each key to a random bucket.
- Bad compression function:

  Suppose keys are ints.

  hashcode(i) = i.

  Compression function: h(hashCode) = hashCode mod N

  N = 10,000 buckets.

- Suppose keys are divisible by 4. → h( ) is divisible by 4 too.

  → Three quarters of buckets are never used!

- Same compression function is better if N is prime.
- Better: h(hashCode) = ((a * hashCode + b) mod p) mod N

  a, b, p: positive integers;

  p is large prime, p >> N.

Now, N (buckets) doesn't need to be prime.

- Good hash code for Strings:

```
int hashCode(const string& key) {
```

```
    int hashVal = 0;
    for (int i = 0; i < key.size( ); i++) {
        hashVal = (127 * hashVal + key.substr(i,1)) % 16908799;
    }
    return hashVal;
}
```

◎ Bad hash codes on Words

① Sum ASCII values of characters.

- Most words rarely exceed 500.
- Bunched up in 500 buckets
- Anagrams like "pat", "apt", "tap" collide.

② First 3 letters of a word, with 26^3 buckets.

- Lots of "pre…" words. → collide
- No "xzq" … words

③ Suppose prime modulus to be 127.

(127 * hashVal) % 127 = 0

common factor   increase collision probability

⇨ Final hashVal has form ax + b, where b depends only on last character.

◎ Resizing Hash Tables

① If load factor n / N too large, we lose O(1) time.

- Enlarge hash table when load factor > c, typically 0.75.
- Allocate new array (at least twice as large).
- Walk through old array, rehash entries into the new array.
- CANNOT just copy the linked lists to the same buckets in the new array, because the compression functions of the two arrays will certainly be incompatible.

→ need to change compression function.

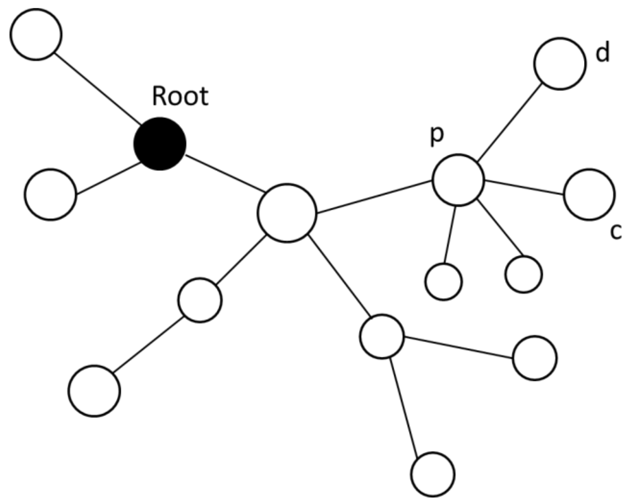
② Shrink hash tables (e.g., when n / N < 0.25) to free memory. (In practice, it's only sometimes worth the effort.)

- Hash table operations: Usually O(1) time (on average)
- When resizing happens, one operation that causes a hash table to resize itself can take O(n) time (more than O(1) time).
- Operations still take O(1) time on average.

§ Rooted Trees

◎Tree: consists of a set of nodes & edges that connect them.

- Exactly one path between any two nodes.
- Path: connected sequence of edges.
- e.g.



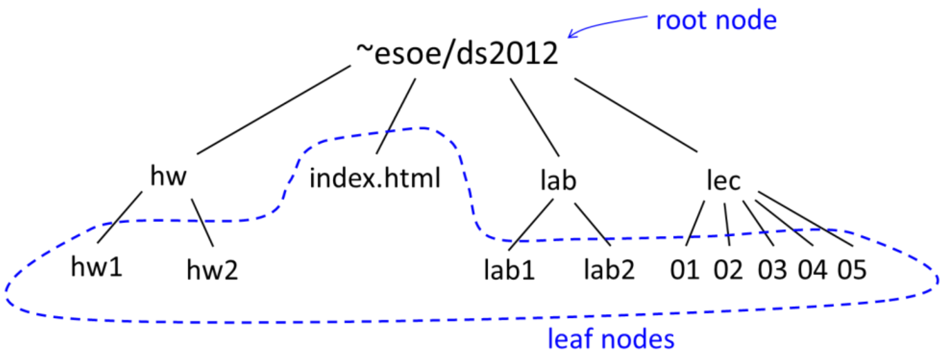
◎Rooted tree : one distinguished node is called the <u>root</u>.

・Every node *c*, except root, has one <u>parent</u> *p*, which is the first node on the path from *c* to the root.

・*c* is *p*'s <u>child</u>.

・Root has no parent.

・A node can have any # of children.



★More definitions:

・<u>Leaf</u>: node with no children, e.g., *c*.

・<u>Siblings</u>: nodes with same parent.

・<u>Ancestors</u> of a node *d*: nodes on the path from *d* to root, including *d*, *d*'s parent, *d*'s grandparent, …, root. Root is the ancestor of each node.

・If *a* is an ancestor of *d*, then *d* is <u>descendant</u> of *a*.

・<u>Length</u> of path: # of edges in path.

・<u>Depth</u> of node *n*: length of path from *n* to root (depth of root is zero), e.g., depth of *c* is 3.

・<u>Height</u> of node *n*: length of path from *n* to its deepest descendant (height of any leaf is zero).

・Height of a tree (is the depth of its deepest node) = height of the <u>root</u>.

・<u>Subtree</u> rooted at *n*: tree formed by n & its descendants.

・A <u>binary tree</u>: no node has > 2 children, every child is either a <u>left</u> child or a <u>right</u> child, even if it's the only child.

・e.g.

◎Representing Rooted Trees

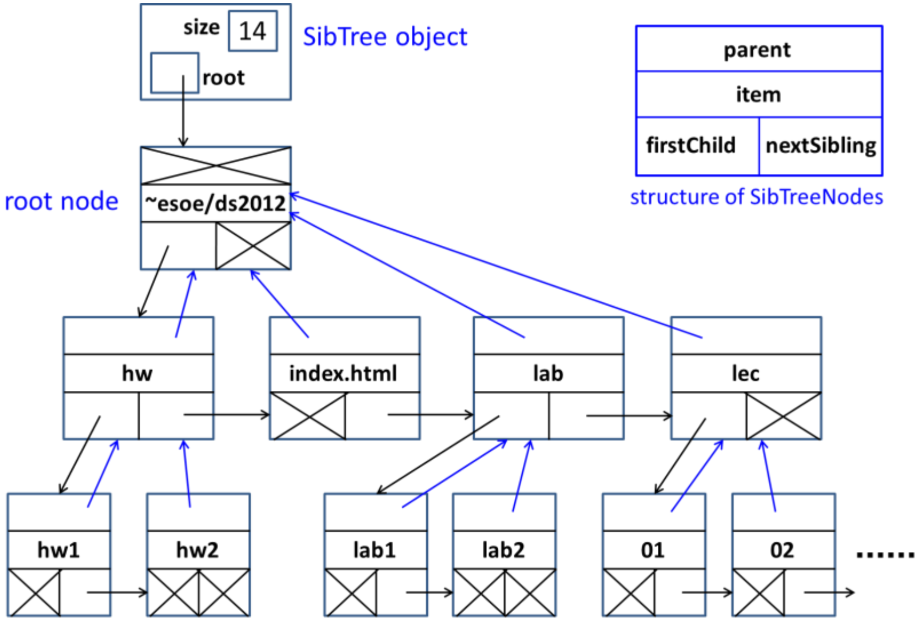
・Each node has 3 references: item, parent, children stored in a list.

・Another option: siblings are directly linked.

```
template <typename E>
class SibTreeNode {
public:
    E item;
    SibTreeNode<E>* parent;
    SibTreeNode<E>* firstChild;
    SibTreeNode<E>* nextSibling;
};
template <typename E>
class SibTree {
public:
    SibTreeNode<E>* root;
    int size;
};
```

int size → # of nodes in tree.

• e.g.



size
14
root
SibTree object
root node
~esoe/ds2012
hw
index.html
lab
lec
hw1
hw2
lab1
lab2
01
02
......
parent
item
firstChild
nextSibling
structure of SibTreeNodes