Lecture #19

Tables: Searching & Sorting

Overview

- Given: a collection of <tag,value> pairs
  - e.g., symbol table
- Searching =
  given a tag, return corresponding value
- Q: is this spec ok?

Robustness

- What do we do if key not in table?
  - a) return an arbitrary value
  - b) crash, halt, explode
  - c) return a special value (NULL, error, …)
- Two solutions:
  - explicit precondition: tag must be present
    - client must ensure this condition before search
  - change postcondition to guarantee (c)
    - “defensive” programming

Relevance for Assemblers

- Every line (almost) has an instruction or a pseudo op
  - requires table search
- Can have lots of symbols and literals
  - need to create and search tables
- Assemblers can spend 50% of their time searching tables!!
  - so, it is important that this be efficient

Linear Search

- Algorithm:
  compare target with 1st key
  if match, then done (return value)
  else, compare target with 2nd key
  if match, then done (return value)
  …
- Advantages:

Linear Search - Complexity

- How long to insert a new <tag,value> pair?
  -
- How long to find a target?
  - best case:
  - worst case:
  - "average" case:

Linear Search - Complexity II

- Average case analysis requires an assumption of the distribution of targets

$$T_{avg} = \sum_i p(i)t(i)$$

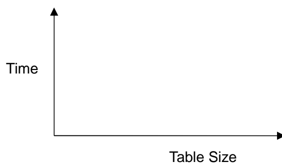
- E.g., for a uniform distribution,

$T_{avg} =$

- One way to lower average case complexity:
  - – ________________________________
- All this analysis assumes ______________
  - if not present, time is ____________

Linear Search - Complexity III

- Overall search complexity is O( )
  - i.e. double table size ⇒



Binary Search

- Algorithm

  compare target with middle key

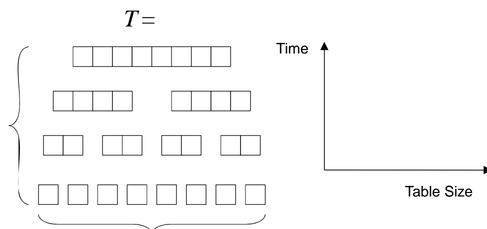
  if target ≤ middle key, then search first half

  if target > middle key, then search second half
- This algorithm requires:
  - a)
  - b)
  - c)

Binary Search - Complexity

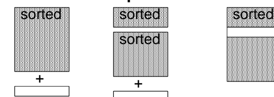
- Each iteration divides the problem in half

$T =$



Binary Search - Complexity II

- Example:
  - table with 1,000,000 entries: _________
  - table with 2,000,000 entries: _________
- Drawback: simple insertion is linear



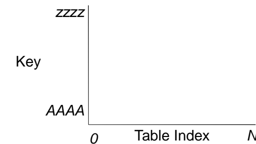
  - time to build =

Building Sorted Tables

- Solution #1: use a heap
  - insertion is faster, O(lg N)
  - but more complicated algorithm
- Solution #2: build, then sort
  - in assembler, pass #1 does mostly insertions, then pass #2 does mostly searches
  - create as a linear table O( )
  - sort O( )
  - use binary search on sorted table

Estimated Search

- How do *we* search for things?
  - e.g., finding "Brutus" in phone book
  - binary search? No! Make a guess…
- For table search, must know ___________



Lecture #20

Hash Tables - Overview

- Combines:
  - strength of binary search (fast _________)
  - strength of linear search (fast _________)
- Hash function: converts keys to integers

  $h: K \longrightarrow Z_N$
  - where: $Z_N = \{ 1, 2, 3, \ldots, N \}$

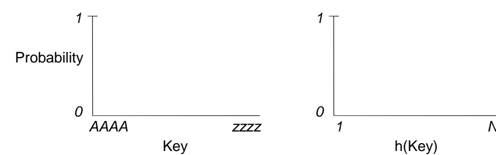
    N =

    K =

Hash Functions

- Used to insert and to search
  - insert(key) --> into h(key)
  - search(key) --> look in h(key)
- Ideal: generate a unique integer for each key
  - but this ideal does not (usually) exist
  - because:
- So what we really look for in h:
  - returns a number in [1..N] with *uniform* distribution

Designing Hash Functions



- Want: *small* differences in key result in *big* differences in h(key)
  - like a "deterministic random number generator"

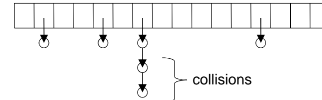
Example Hash Function

h(key) = (∑ letter values) mod 50 + 1

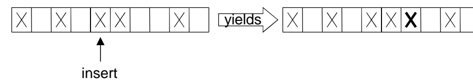
- For example:
  h(magenta) = (13+1+7+5+14+20+1) mod 50 + 1
- But two different keys could be mapped to same integer
  h(rub) = 18 + 21 + 2 + 1 =
  h(madre) = 13 + 1 + 4 + 18 + 5 + 1 =
- This is called a "collision"

Dealing with Collisions

a) Keep a linked list

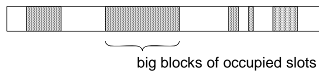


b) Go to next open cell



Dealing with Collisions II

– this approach can lead to "clustering"



– alternative: exponential back-off

c) Rehash with another function

– cost: ____________________

Complexity - Insertion

- How long does an insertion take?
- #attempts depends on __________________
  - e.g., first entry never collides (1 attempt)
- Let r be the fraction of the table that is full
  - probability of a collision =
  - probability of success =
- (Aside: if clustering occurs, everything breaks down… search becomes linear)

Insertion II

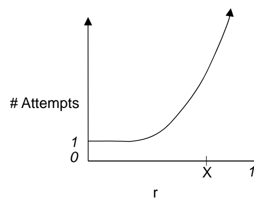
- Let $p(i)$ = prob. insertion requires i attempts
  = $p(i\text{-}1 \text{ failures}) \times p(1 \text{ success})$
  =
- So the *expected* number of attempts is given by: $\sum_{i=1}^{\infty} p(i) \times i =$
- Example: if a table is three quarters full, how many attempts does the next insertion take?

$r =$
$\#\text{attempts} =$

Complexity - Building a Table

- Example:
  – table size = 1000, wish to insert 900 elements
  – how long does this take?
  – first item: _____ (so a lower bound: ________)
  – last item: _____ (so an upper bound: _______)
- Problem: each insertion takes more time

Building a Table II



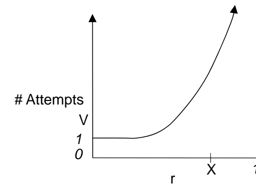
$$A = \int_0^X \frac{1}{1-r}\,dr$$

$$= \ldots$$

$$= ________$$

Building a Table III

- On average, how many attempts needed?



$$V \times X = \ln\left(\frac{1}{1-X}\right)$$

$$V = \frac{1}{X}\ln\left(\frac{1}{1-X}\right)$$

- In our example: X = .9
  - ∴ V = 2.558
  - ∴ time = 2302 (ie V attempts/insert x 900 inserts)

Lecture #21

Midterm

Lecture #22

CVS

To Ponder

- Why do Computer Scientists often confuse Halloween and Christmas?

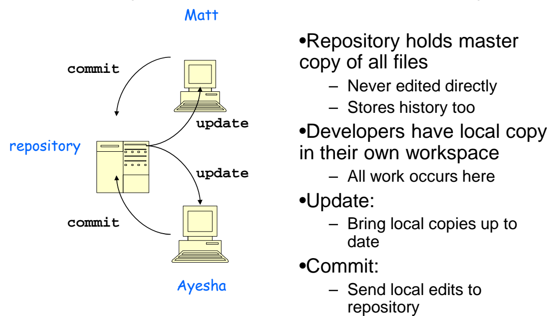
CVS: Concurrent Version System

- Widely used, especially in the opensource community, to track all changes to a project and allow access
  - Can work across networks
- Key Idea: *Repository*
  - The place where the originals and all the modifications to them are kept.
  - Each person "checkout"s their own, private copy
  - Changes are "commit"ed by each person
  - Everyone else's changes are "update"d into your own copy.

Motivation

- Team-based development
  - Developers share and extend common code base
  - Team members comply with standards (coding conventions, comment templates,…)
  - Bug fixes applied to deployed version 1.0 while development continues, in parallel on version 2.0
- Especially important for Agile methods
  - Popular for open source development
  - But *every* team project needs some kind of code management and versioning system

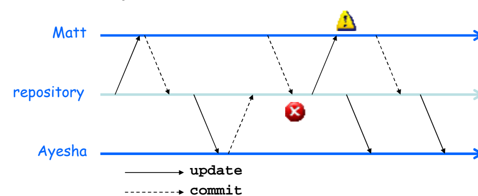
Key Idea: The Repository



- Repository holds master copy of all files
  - Never edited directly
  - Stores history too
- Developers have local copy in their own workspace
  - All work occurs here
- Update:
  - Bring local copies up to date
- Commit:
  - Send local edits to repository

Conflicts and Merging

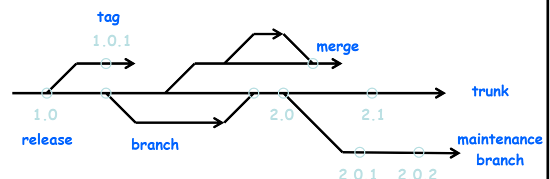
- Optimistic team model
  - Anyone can modify any file any time (no locking)
  - Most edits can be safely merged automatically
  - Assumption: real conflicts are rare



Tagging, Branching, and Merging

- Repository is a *tree* of versions
  - Development of main product occurs as a series of revisions along trunk
- A *tag* names a particular revision
- *Branches* off of trunk or other branches
  - Bug fixes of a particular release
  - Exploring different development paths
- Branches can be *merged* back to trunk
  - Speculative direction pans out

A History of Revisions



CVS: Examples

- The following examples show an existing project being put under CVS
- How to start using the repository
- Then two different people making changes:
  - Putting modified file into repository
  - Getting each other's changes
  - Finding out how things have changed
- You can read more:
  - man cvs, or in emacs: `^H ^i /usr/local/info/cvs.info`
  - Pointer to manual on web page

The Repository

- Two ways to set the "root" of the repository
  - Environment variable
    - `setenv CVSROOT "/n/silver/2/c560aa/CVSREP"`
  - Command line flag (-d)
    - `-d /n/silver/2/c560aa/CVSREP`
- Repository may contain several modules

Creating the Repository

- Once per project, by one person.
- Command:
  - `cvs init`
- Creates repository root, administrative files
- Make sure group permissions properly set

Creating the Repository (Example)

```
% cd /n/silver/2/c560aa/
% ls
Lab1/   Lab2/
% cvs -d /n/silver/2/c560aa/CVSREP init
% ls
CVSREP/    Lab1/    Lab2/
```

Adding an Existing Project

- Once per project, by one person
- Command:
  - `cvs import <module> <vendor> <release>`
- Copies current directory contents to module
- Afterwards, original source can be removed
  - But be *careful*!!

Adding an Existing Project (Example)

```
% cd Lab1
% ls
loader.c   loader.h   simulator.c  simulator.h
% cvs -d /n/silver/2/c560aa/CVSREP import sim A start
N sim/loader.c
N sim/loader.h
N sim/simulator.c
N sim/simulator.h

No conflicts created by this import
% (carefully remove contents of c560aa/Lab1)
```

Creating Local, Working Directory

- Once per person
- Command:
  - `cvs checkout <module>`
- Copies repository files to local, working directory
  - Local directory contains CVS subdirectory for administrative book-keeping

Creating Local, Working Directory (Example)

```
% cd ~person1/mycode/
% cvs -d /n/silver/2/c560aa/CVSREP checkout sim
cvs checkout: updating sim
U sim/loader.c
U sim/loader.h
U sim/simulator.c
U sim/simulator.h
% ls
sim/
% ls sim/
CVS/   loader.c   loader.h  simulator.c simulator.h
```

Committing Changes to Repository

- Each person, with appropriate frequency
- Command:
  - `cvs commit`
- Commit (copy) changes made on local (working) files to the repository
- New files created in local working directory must be *explicitly* added (before commit)
  - `cvs add <new-file>`

Committing Changes to Repository (Example)

```
% cd ~person1/mycode
% (modify loader.c and create memory.h)
% cvs add memory.h
cvs add: scheduling file 'memory.h' for addition
cvs add: use 'cvs commit' to add this file permanently
% cvs commit
cvs commit: Examining .
RCS file: /n/silver/2/c560aa/CVSREP//sim/memory.h,v
Checking in memory.h
/n/silver/2/c560aa/CVSREP//sim/memory.h,v ← memory.h
Initial revision: 1.1
done
Checking in loader.c;
/n/silver/2/c560aa/CVSREP//sim/loader.c,v ← loader.c
New revision: 1.2; previous revision: 1.1
done
```

Updating the Local Directory

- Each person, with appropriate frequency
- Command:
  - `cvs update`
- Brings your local working directory up-to-date with repository (merging differences if possible)
  - U : local file was updated
  - A/R: local file added/removed
  - M: local file is a modification of repository
  - C: conflict detected between local file and repository

Updating the Local Directory (Example)

```
% cd ~person2/mycode/sim
% ls
CVS/  loader.h  loader.c  simulator.h simulator.c
% cvs update
cvs update: Updating .
U loader.c
A memory.h
% ls
CVS/        loader.c   simulator.h
loader.h    memory.h   simulator.c
```

Working on Project

- *Multiple* people can *simultaneously* checkout the *same* module
- Person1 and Person2 are both working away on their local copies
  - If working on different files, no problem
    - Not quite true!
  - If they are working on the same file, there could be a conflict

Conflict Resolution

- Person1 checks out code
  - modifies loader.c
- Person2 also checks out code
  - also modifies loader.c
- Person1 commits: no problem
- Person2 commits:

```
% cvs commit
cvs commit Examining .
cvs commit: Up-to-date check failed for 'loader.c'
cvs [commit aborted]: correct above errors first!
```

- Person2 must update their local directory first.

Resolving Conflicts

- During an update, CVS tries to merge differences between repository and local directory
- Sometimes these changes clash

```
% cd ~person1/mycode
% cvs update
Cvs update: Updating .
RCS file: /n/silver/2/c560aa/CVSREP//sim/loader.c,v
Retrieving revision 1.5
Retrieving revision 1.6
Merging differences between 1.5 and 1.6 into loader.c
Rcsmerge: warning: conflicts during merge
Cvs update: conflicts found in loader.c
C loader.c
```

Resolving Conflicts: Human

```
cvs diff
Index: elf.C
===================================================================
RCS file: /n/silver/3/c560aa/CVSROOT/example/elf.C,v
retrieving revision 1.1.1.1
diff -r1.1.1.1 elf.C
23a24,28
>   /*
>    * Every ELF file starts by doing an elf_begin() and
>    * is finished by doing an elf_end ().
>    */
>
26a32,37
>   /*
>    * We need to describe the object file (what we are creating)
>    * type by supplying an ELF header.  Here, we are creating
>    * a 32-bit (ELFCLASS32), big-endian (ELFDATA2MSB), Sparc
>    * (EM_SPARC), relocatable (ET_REL) object file.
>    */
42a54,57
>   /*
>    * A program header is only needed for executables as it
>    * describes how to bring the program into memory
>    */
```

When to Update/Commit

- Commit when confident that your work can be used by others
  - *do not* wait until perfection!
  - *do* make sure your new version compiles!
- Update before committing
  - integrates everyone else's changes
- Update when you are ready for someone else's work
  - availability of new modules that may affect your code
- More files, the better

Best Practices: Golden Rule

- Never break the build
  - Applies (primarily) to trunk, although breaking a multi-developer branch is almost as bad
  - Frequent commits are a good thing, but partial code should not prevent another developer from building and testing their modifications
- (Almost) Never break a test case
  - Other developers may think their (local) changes are responsible for new errors when they next update

Best Practices: What *Not* to Include in Repository

- Generated code
  - eg Java byte code, javadoc html
- FIXME comments *in trunk*
  - OK for developer branches, but should be resolved before merging into trunk
- TODO comments *in trunk* (?)
  - Team convention whether or not these should be allowed
  - Good reasons on each side of argument:
    - Useful for book marking tasks needing attention (by self or others!)
    - Lazy cruft that will accumulate over project lifespan
  - Advice: the more agile the process, the more permissible TODO comments are in the trunk
  - Always OK for developer branches

Best Practices: Process

- Daily build schedule
  - The “heartbeat” of the project
- Release means: tag + create branch for maintenance
- Always tag before a merge
  - Simplifies roll-back if merge goes horribly wrong
- Adopt team standard style:
  - tag names (versions, major, minor, bug fixes…)
  - light comment template (brief 1-liners are best)
- Use locks for non-mergeable (eg binary) files

Pitfalls

- Incomplete commits
  - Common problem: forgetting to add a new file
  - Incentive-based approach may help
- Binary vs ASCII files
  - Binary files must be explicitly marked as such to prevent end-of-line mangling
- Iteration 1 takes forever
- System clock synchronization
  - UTC, so no time zone issues

Alternative: SVN

- “Subversion” (subversion.tigris.org)
- Increasingly popular in open source community
- Repository stored as a series of diffs
  - Faster update and commits
- Support isn’t native in Eclipse, but good plugins do exist
- Advantages:
  - File attributes are part of stored properties
  - Transactional commits
  - Versions refer to entire project (eg directories, not file by file)
  - No need to explicitly mark binaries
  - Support for renaming resources (vs delete and re-add)
  - Better authentication management for remote access
  - Faster, especially for large binaries