15-415/615
Database Applications
Spring 2015

HW3: B+ Tree (Recitation)

Vinaykumar Bhat

Jiayu Liu

Carnegie Mellon University

Overview

- You are given a basic B+ tree implementation
- **Task**: extend the B+ tree implementation for new operations
- **Goal**: get familiar with B+ tree and recursive code that manipulates the tree & pages

Why B+ tree?

- large fan out
- balanced and shallow
- efficient for slow I/O devices (i.e. risks)
- for more detail - check the slides (http://www.cs.cmu.edu/~christos/courses/dbms.S15/slides/09TreeStructureIndices.pdf)

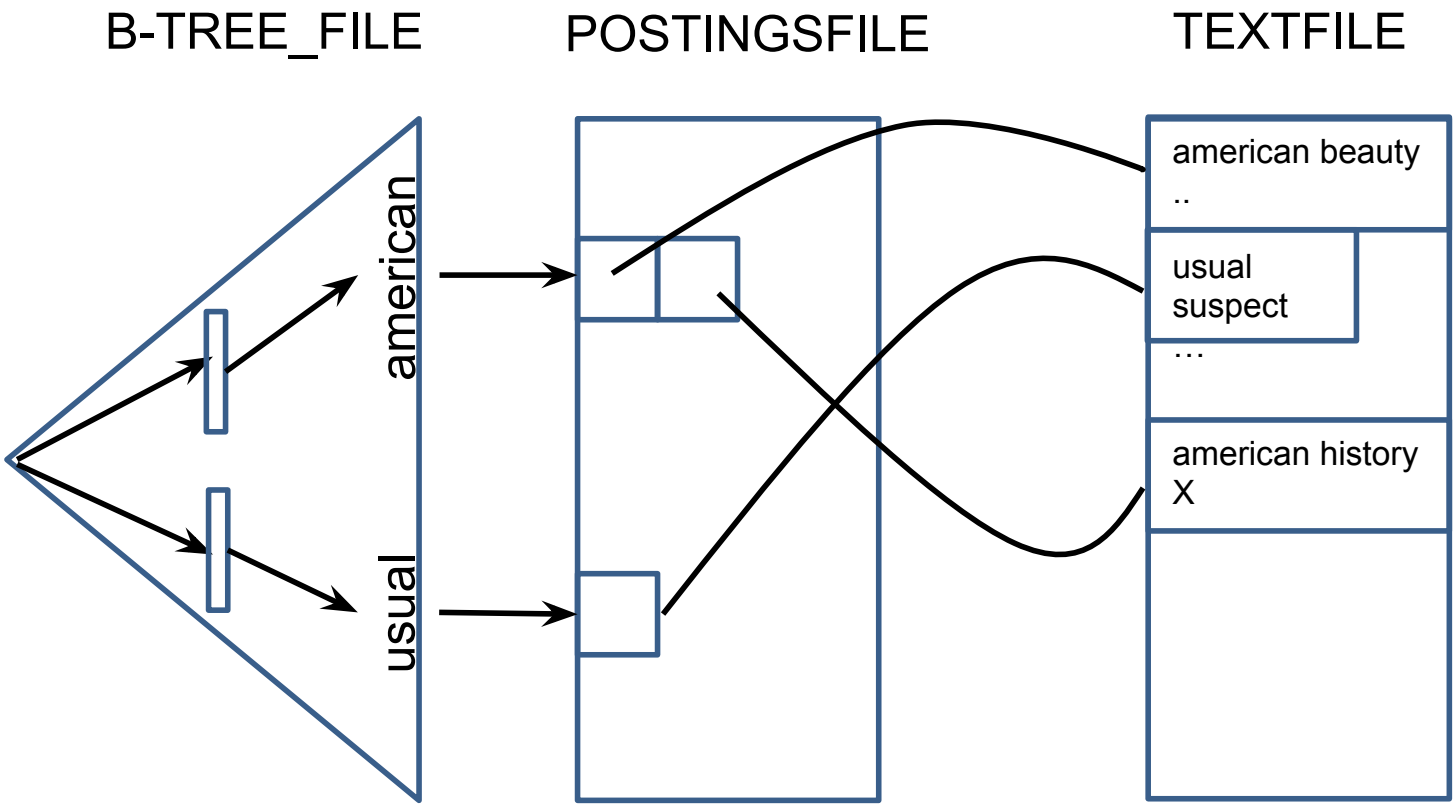
Basic B+ Tree Implementation

- Creates an “inverted index” in the form of a B+ tree
  - key: *word*, value: *document name*
- Supports: insert, scan, search, print
- No duplicate keys are allowed
- No support for deletion
- The tree is stored on disk

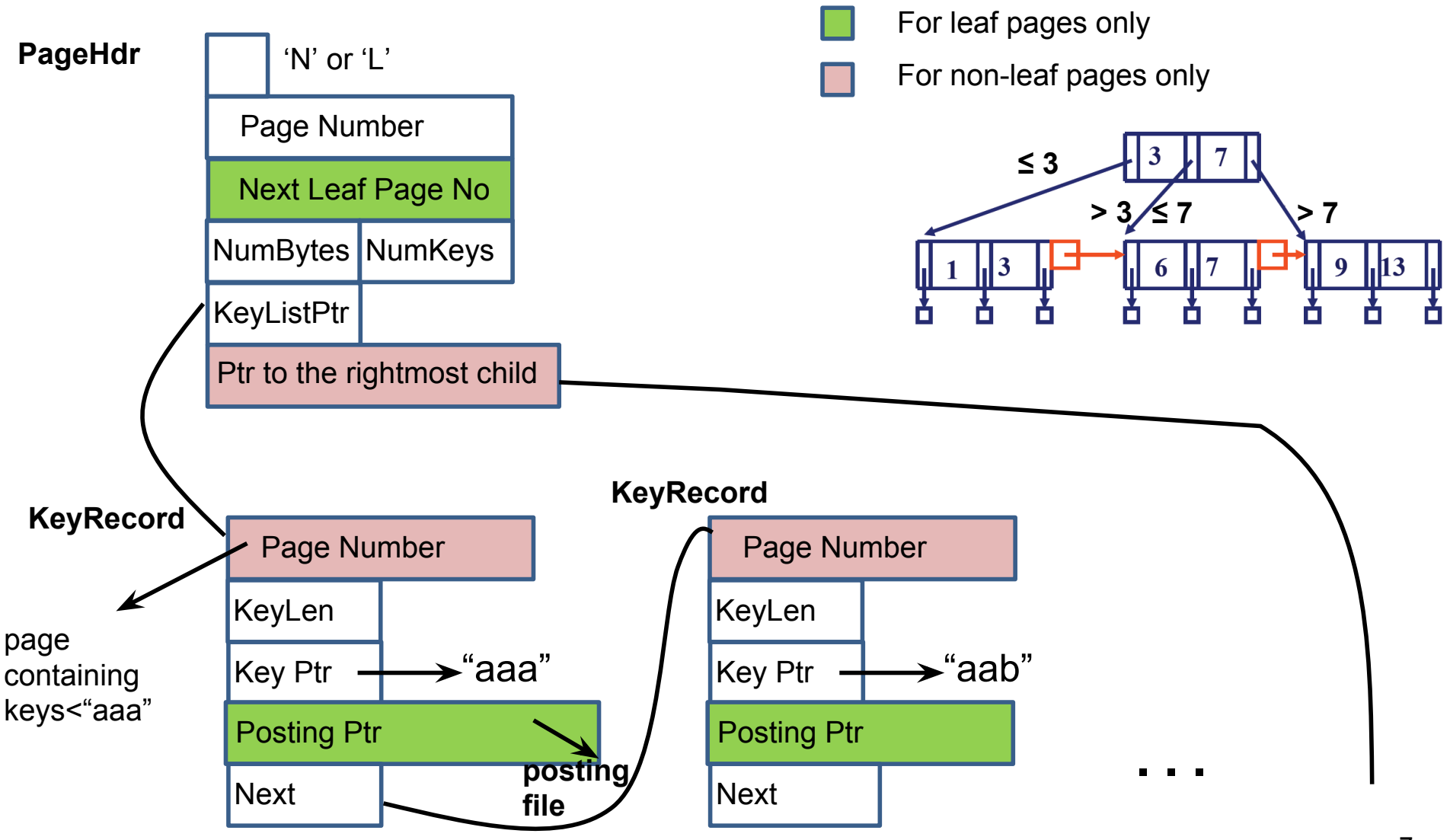
B+ Tree Package

- Folders
  - **DOC**: documentation
  - **SRC**: source code
  - **Datafiles** : sample documents data
  - **Tests**: test files
- B-TREE_FILE, POSTINGSFILE, TEXTFILE, parms are created by the b+ tree.
  - Want a new tree? Delete them

B+ Tree Structure



Structure of a Page (def.h)



Existing Functions

- **C** : print all the keys
- **i <document_name>** : insert the document
  - key: word, value: document_name
- **p <page_no>** : print the info on the page
- **s <key>** : search the key
- **S <key>** : search the key, and print the documents
- **T** : print the tree

Demo

Example code, for searching

- search.c
  - search function entrance, used in main.c
  - calls *treesearch* to locate the page to which the key belongs
- treesearch.c
  - recursive call to locate the page for the key
  - calls *FindPageNumOfChild* to find the correct children (looks down)
- FindPageNumOfChild.c
  - traverse a non-leaf page

To be implemented

- **#**: Display the number of pages fetched (in any operation prior to this)
- **> key n**: Fetch and display ‘n’ keys which are successor to ‘key’
- **< key n**: Fetch and display ‘n’ keys which are predecessor to ‘key’

To be implemented: count (#)

- Should display the number of pages fetched by the command prior to this
- Print and reset
- Hint: Try to leverage existing functionalities

To be implemented: successor (>)

- `> key n`
- Display the ‘n’ successors key-strings of ‘key’
- Use the count command (#) to check your efficiency
- Hint: Try to leverage existing functionalities

To be implemented: predecessor(<)

- `< key n`
- Display the ‘n’ predecessor key-strings of ‘key’
- Use the count command (#) to check your efficiency
- Hint 1: Try to leverage existing functionalities (though may not be straightforward)
- Hint 2: Think of recursion / backtracking

Keep in Mind

- The ‘count’ is not a hard limit.
  - May not exactly match the reference count
  - Should be reasonable
- A fully sequential scan is a strict no-no
  - Will show up as a very large count
- Will be graded to check for efficiency later.
  - Again ‘sequential’ scans will be penalized
- Understand the provided infrastructure before starting!

Build Infra (makefile)

- make load
  - Initialize the tree
  - Insert all datafiles
- make test_sanity
  - Runs load
  - Tests the very minimal functionality. No diffs = test pass!
  - make sure the **output is formatted**. This is **absolutely** necessary for autograding
- make test_successor/make test_predecessor
  - The questions asked in the handout

Testing Mechanism

- Correctness
  - output the correct list of words (don't forget to check all the corner cases!)
- Format
  - Make sure the output follows the **same format as the sanity test** solutions.

Hand-in

- Create a **tar file** of your source code, as well as the makefile. (make handin)
- **Hard-copy** of a document with the functions that you modified/added.
- Please make sure that the “make” command compiles all the source code without any errors
- Submit **your code** on blackboard.

Questions?

- Come to office hours (5 TAs + instructor)
- Read the handout before starting
- Post your questions on blackboard