Advantages of BST over Hash Table

Hash Table supports following operations in Θ(1) time.

1) Search

2) Insert

3) Delete

The time complexity of above operations in a self-balancing Binary Search Tree (BST) (like Red-Black Tree, AVL Tree, Splay Tree, etc) is O(Logn). more

So Hash Table seems to beating BST in all common operations. When should we prefer BST over Hash Tables, what are advantages. Following are some important points in favor of BSTs.

1 We can get all keys in sorted order by just doing Inorder Traversal of BST. This is not a natural operation in Hash Tables and requires extra efforts.

2 Doing order statistics, finding closest lower and greater elements, doing range queries are easy to do with BSTs. Like sorting, these operations are not a natural operation with Hash Tables.

3 BSTs are easy to implement compared to hashing, we can easily implement our own customized BST. To implement Hashing, we generally rely on libraries provided by programming languages.

With BSTs, all operations are guaranteed to work in O(Logn) time. But with Hashing, Θ(1) is average time and some particular operations may be costly, especially when table resizing happens.

Volatile Keyword in C:

The declaration of a variable as volatile tells the compiler that the variable can be modified at any time externally to the implementation, for example, by the operating system, by another thread of execution such as an interrupt routine or signal handler, or by hardware.

In practice, you must declare a variable as volatile whenever you are:

1. accessing memory mapped peripherals
2. sharing global variables between multiple threads

3. accessing global variables in an interrupt routine or signal handler.

4. a volatile object is accessed through a non-volatile-qualified reference, resulting in [undefined behavior](https://www.securecoding.cert.org/confluence/display/c/BB.+Definitions#BB.Definitions-undefinedbehavior)

Ref:

<http://priyaranjan-technicalzone.blogspot.com/2014/02/c-language-volatile-keyword.html>

SpinLocks:

Whenever you write kernel code, you should ask yourself these questions:

1. Is the data global? Can a thread of execution other than the current one access it?

2. Is the data shared between process context and interrupt context? Is it shared between two different interrupt handlers?

3. If a process is preempted while accessing this data, can the newly scheduled process access the same data?

4. Can the current process sleep (block) on anything? If it does, in what state does that leave any shared data?

5. What prevents the data from being freed out from under me?

6. What happens if this function is called again on another processor?

7. Given the proceeding points, how am I going to ensure that my code is safe from concurrency?