**B-tree\_with\_cache**

The goal of this project is to implement a B-tree and to measure an enhanced structure where an amount of cache can be used.

The tree is as described during the lecture. I.e. a tree of order t, has to store at least t-1 elements in each node (except the root) and at most 2t-1.

First, to implement B-tree - the basic operations are:

I x - Construct a tree of order x.

A x - Add the value x to the tree, x is a signed integer. The values added are unique, i.e. there will be no two insertions of the same value (except the case when it was deleted in between).

? x - Check if the value x is in the tree. If it is present print "x +", otherwise print "x -".

P - Print all values that are in the tree in increasing order.

L t - Load a tree of order t. In the next line there is a tree in a format given as follows. A pair of brackets ( ) signify a node. A value signify a record (key in the tree).

( ) Hence ( ( ( 1 2 ) 3 ( 4 ) 5 ( 6 7 ) ) 8 ( ( 9 ) 10 ( 11 ) ) can be interpreted as the following tree:

( . 8 . )

( . 3 . 5 . ) ( . 10 . )

( 1 2 ) ( 4 ) ( 6 7 ) ( 9 ) ( 11 )

S - Save the tree in the format described above.

R x - Remove the element x from the tree. The element may be present or not. Hence the operation might not change the elements in the tree, but may force reorganization.

# - Ignore the line.

X - End the program.

C x q1, q2, ... - Calculate an impact of a hypothetical cache.

Assume that you are given a set of quarries q1, ..., qn (the list of quarries ends by a new line character).

The program can store the result of the quarry in a cache of size x - measured in the number of entries (keys) that can be stored.

Hence, if the next quarried entry was already visited, the program does not have to visit B-tree, but may simply read the data from cache.

Due to the fact that the size of the cache is much smaller than the size of B-tree, only chosen entries can be stored.

The policy that has to be implemented is First-In-First-Out.

I.e. the first entry that was read to the cache is removed from it when some other entry is read from B-tree and the result has to be added to the cache.

Keep in mind, that a result "Not found" is a proper result to store in the cache.

The measure of the impact of the cache is given by the number of accesses to the disk (measured as depth in the B-tree).

As an example consider the tree from point L and series of quarries given by C 2 1 2 1 1 3 1.

The measure without cache is 17.

The measure with cache is 3 + 3 ( to fill the cache ) + 0 + 0 ( free loads due to the cache) + 2 + 3 (sadly, the entry 1 was removed from the cache, hence it has to be re-loaded) = 11.

A bonus can be earned, if other reasonable cache policy is proposed and it's impact (in disk accesses) is measured.

Grades:

- I A ? - together gives 25%

- P - 5%

- L S - 15%

- R - 40%

- C - 15% (+10% bonus, if other cache policy than discussed is proposed and measured (calculating the impact on the number of disk accesses is enough))

A penalty may be given for hard-coding the program for tests, i.e., the program shall work for any reasonable input.

Bounding the order of tree by 2/10/10000 (by for example allocating always an array of such a size for each node) is not proper.

Please allocate the memory dynamically.

As a reminder, no algorithms/data structures from STL, like vectors, lists, maps, etc. can be used. All general rules still apply.

Tests:

1 : sanity check (failing = 0 from this project)

2 - 5 : tests for I A ?

6 - 7 : tests for P

8 - 11: tests for L

12 - 13: tests for S

14 - 17: tests for R

18 - 22: tests for C

The tests assume that ? / C - operation does not change the structure of the tree. Otherwise, they should be not dependent on the details of the implementation.

Examples:

1) Input:

I 2

A 2

A 4

A 5

A 1

? 2

? 3

X

1) Output:

2 +

3 -

2) Input:

I 2

A 2

A 4

A 5

A 1

P

X

2) Output:

1 2 4 5

3) Input:

I 2

A 2

A 4

A 5

A 1

S

X

3) Output:

2

( ( 1 2 ) 4 ( 5 ) )

4) Input:

L 2

( ( 1 2 ) 4 ( 5 ) )

? 1

? 10

X

4) Output:

1 +

10 -

5) Input:

I 2

A 2

A 4

A 5

A 1

? 1

R 1

? 1

A 1

? 1

X

5) Output:

1 +

1 -

1 +

6) Input:

I 2

A 2

A 4

A 5

C 2 2 10 2 2 4 2

X

6) Output:

NO CACHE: 6 CACHE: 4

Tips:

Implement first I A ? - to see that the program works at all.

Then implement P and S before starting with D.

These operations (together with "health checks" that you should implement) should allow you to implement D operations more easily.

Also, try to write a few functions that will measure a "health" of the program, if all the pointers are valid, nodes have good sizes etc.

Do not optimize the code before checking that all the operations are working as required, it might be not even necessary.

Precisely, you can assume that the order of tree is small.

Hence, the gain from implementing a more clever check for a key in a given node (using bisection, balanced tree, etc.) is not necessary.

For delete operation decouple merging and deleting.

Notice that if a tree has structure and we would like to remove Y

( ... X . Y . Z ... )

( a b ) ( c d )

first we can change the structure to

( ... X . Z ... )

( a b Y c d )

And recursively call for removal of Y in the new node ( a b Y c d).

If the root was merged it might be the case that the old root is composed of a single pointer.

The height of the tree has to be decreased.

Similarly, the splitting and adding can be decoupled.

Keep in mind that when "shifting" the pointers from node to node the parent of the pointered nodes changes.

Introduction to Algorithms by T. H. Cormen et al. is a good reference.