Concurrent Algorithm to Globally Balance a Binary Search Tree

Mohammed-Ali Khan

Department of Computer Science and Engineering, York University,
4700 Keele Street, Toronto, M3J 1P3
Ontario, Canada

m23khan@cse.yorku.ca

Presented on: February 09, 2011

Agenda

- Brief Introduction to Binary Search tree.
- Introduction to article chosen.
- Motivation behind choosing the article.
- Presenting the Algorithms S and P1.
- Sample trace for algorithms.
- Related Work.
- References.
- Questions Period.

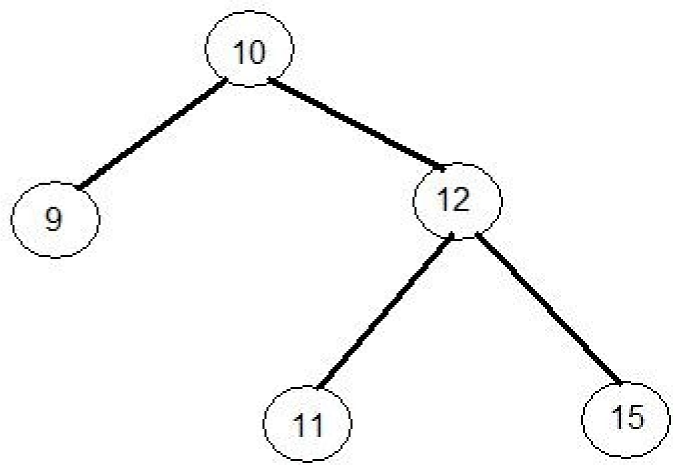
Binary Search Tree 101

. What are search trees?
Search trees are data structures which support operations such as SEARCH, MIN, MAX, PREDECESSOR, SUCCESSOR, INSERT, DELETE. It can be used as a priority queue and as a dictionary. [1]

. What is a binary search tree?
A search tree organized in a binary tree. In this tree, each node is an object [1]. *For our case, each node will consist of: pointer ROOT (pointing to ROOT), KEY (the key value of the node), pointer LSON (pointing to the left subtree), and pointer RSON (pointing to right subtree)* [2].

. Why use binary search tree?
Operations on binary search tree have a worst case running time of O(n), in other words, linear search time [1]. *This is a desirable property especially as the search tree grows large.*

A Binary Search Tree



Node	
ROOT	10
KEY	9
LSON	null
RSON	null

Node	
ROOT	10
KEY	10
LSON	9
RSON	12

Node	
ROOT	10
KEY	11
LSON	null
RSON	null

Node	
ROOT	10
KEY	12
LSON	11
RSON	15

Node	
ROOT	10
KEY	15
LSON	null
RSON	null

Article Chosen

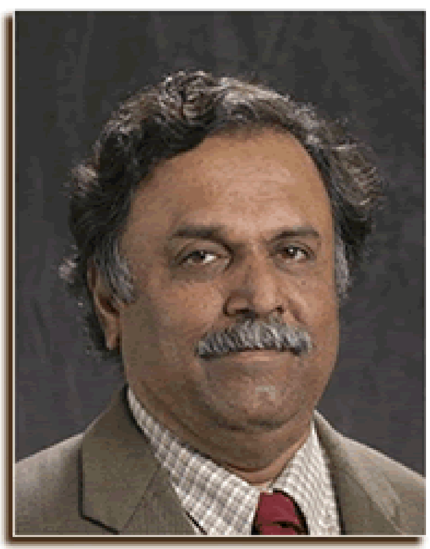
For this Assignment, I chose the following article by Dr. Sitharama Iyengar and Hsi Chang:

S. Sitharama Iyengar and Hsi Chang. Efficient algorithms to globally balance a binary search tree.
Communications of the ACM, 27(7):695–702, July 1984.

Dr. S.S. Iyengar

(AAAS Fellow, IEEE Fellow, ACM Fellow)

Roy Paul Daniels Professor
&
Chairman
Department of Computer Science
Director, Robotics Research Laboratory
298, Coates Hall,
Louisiana State University
Baton Rouge, LA-70802
Phone: (225)578-1495
Fax: (225)578-1465
E-mail: iyengar@bit.csc.lsu.edu



SOURCE: http://www.csc.lsu.edu/~iyengar/

Highlights of Article chosen

. The article starts off with the authors stating a known fact that "by [...] maintaining a perfectly balanced tree [...] we could [...] expect an average improvement of approx. 27.85% in search path length" [2].

. In the article, 3 algorithms are presented (S,P1,P2) to balance an unbalanced binary search tree. Algorithm S is sequential while P1 & P2 are parallel. I am focusing on S and P1.

. All algorithms presented - S,P1,P2 have a worst case run time of O(n) - they run in linear time.

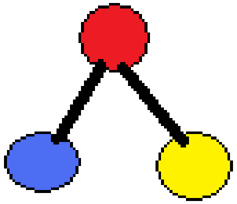
Why this article?

There are several reasons as to why I chose this article:

. Binary search is among the most popular and well known algorithms - thus it is easy to picture real-world applications.

. Binary search is used throughout different Computer Science disciplines - operating systems (file system organization), web engineering, theory of parsing, etc.

. I am a web programmer and I have implemented binary search routines utilizing hash-maps to implement solutions.



Algorithm S

Algorithm S

```
COMMENT: globally balance a binary search tree through folding
PROCEDURE BALANCE(ROOT, LSON, RSON, n)
  integer ROOT, n, N, M, ANSL, ANSR;
  integer array LSON, RSON, LINK(1:n);

  COMMENT: The following procedure traverses the original tree and sets up LINK
  PROCEDURE TRAVBIND(T)
    COMMENT: The pointer T points to the next node to be visited.
     integer T;

    IF T = null THEN RETURN;

    TRAVBIND(LSON(T));
    COMMENT: count the sequence of visit
    N = N + 1;
    COMMENT: store the pointer to the Nth node in the Nth element of LINK
    LINK(N) = T;
    TRAVBIND(RSON(T));
  END PROCEDURE
```

```
COMMENT: The following procedure reorganizes a tree by partitioning and folding
PROCEDURE GROW(LOW, HIGH)
  integer LOW, HIGH;

  COMMENT: MID is the median of a subset bound by LOW and HIGH
  TL is the subtree root in the balanced left half tree,
  TR is the counterpart of TL in the right half tree.
  TL, TR are returned via ANSL, ANSR, respectively.

  integer MID, TL, TR;
  IF LOW > HIGH COMMENT: If null branch THEN
    ANSL, ANSR = null;
  ELSEIF LOW = HIGH COMMENT: Leaf THEN
    ANSL = LINK(LOW), ANSR = LINK(LOW + M);
    LSON(ANSL), RSON(ANSL) = null;
    LSON(ANSR), RSON(ANSR) = null;
  ELSEIF LOW < HIGH COMMENT: Divisible subset THEN
    MID = ⌊(LOW+HIGH)/2 ⌋;
    TL = LINK(MID), TR = LINK(MID+M);
    COMMENT: Form left subtree
    GROW(LOW, MID-1);
    LSON(TL) = ANSL, LSON(TR) = ANSR;
    COMMENT: Form right subtree
    GROW(MID+1, HIGH);
    RSON(TL) = ANSL, RSON(TR) = ANSR;
    ANSL = TL, ANSR = TR;
END PROCEDURE
```

```
 IF n ≤ 2 THEN RETURN;
 COMMENT: Initialize counter
 N = 0;
 TRAVBIND(ROOT);
 COMMENT: Folding value
 M = ⌊(N+1)/2 ⌋;

 COMMENT: new root
 ROOT = LINK(M);

 IF N=2*M THEN COMMENT: N is even
   COMMENT: adjust folding value
   M = M + 1;
   GROW(1, M-2);

   COMMENT: put the node associated with M as a terminal node
   left to its immediate successor
   LSON(LINK(M)), RSON(LINK(M)) = null;
   LSON(LINK(M+1)) = LINK(M);
 ELSE COMMENT: N is odd GROW(1, M-1);

 LSON(ROOT) = ANSL;
 LSON(ROOT) = ANSR;
END PROCEDURE
```

Algorithm P1

Algorithm P1

COMMENT: build balanced left half- and right half-tree in parallel.

```
PROCEDURE BALANCE(ROOT,LSON,RSON,n)
{
  integer ROOT, n, N;
  integer array LSON, RSON, LINK[1:n];

  COMMENT: TRAVBIND(T) identical to TRAVBIND(T) in Algorithm S.
  Therefore it is not reproduced here.
  PROCEDURE TRAVBIND(T)
  {...}
```

```
COMMENT: Reorganize a tree by partitioning.
PROCEDURE GROW(LOW,HIGH){
  integer LOW, HIGH;
  COMMENT: MID is the median of a subset bound by LOW and HIGH,
  T is the root of the balanced subtree reflected by the subset.
  integer MID, T;
  COMMENT: If null branch
  if (LOW > HIGH) {RETURN(null);}
  else if (LOW = HIGH){
    COMMENT: If Leaf.
    T = LINK[LOW];
    LSON[T], RSON[T] = null;
   RETURN(T);}
  else if (LOW < HIGH){
    COMMENT: Divisible set.
    MID = ⌊(LOW+HIGH)/2 ⌋;
    T = LINK[MID];
    COMMENT: Form left and right subtrees in parallel.
    COBEGIN
      LSON[T] = GROW(LOW,MID-1);
      RSON[T] = GROW(MID+1,HIGH);
    COEND;
   RETURN(T);}
}
```

```
  if (n ≤ 2) then return;

  COMMENT: initialize counter.
  N = 0;

  TRAVBIND(ROOT);

  M = ⌊(N+1)/2 ⌋;

  COMMENT: New Root.
  ROOT = LINK[M];

  COMMENT: Balance the left and right halt-trees in parallel.
  COBEGIN
    LSON[ROOT] = GROW(1,M-1);
    RSON[ROOT] = GROW(M+1,N);
  COEND;
}
```

Related Work

. A quick search on ACM Digital Library website reveals that balancing binary search trees were a popular area of research starting in 1970's.

. Even before the publication of Dr. Iyengar and Hsi Chang's article [2], ACM had already published a few articles which dealt with balancing random/unbalanced binary search trees to gain better search performance.

. Before the publication of their article [2], Dr. Iyengar and Hsi Chang had already published another article regarding algorithms to create and maintain balanced and threaded binary search trees [3] (published in 1982 but appears as 1985 on Wiley Online Library website).

. Since the publication of their article regarding balancing binary search trees in 1984 [2], various articles have been published dealing with the issue of balancing binary search trees.

References

[1] Cormen et al. Introduction To Algorithms. Cambridge: McGraw-Hill Book Company, 2004, pp. 253-254.

[2] S. Sitharama Iyengar and Hsi Chang. Efficient algorithms to globally balance a binary search tree. *Communications of the ACM*, 27(7):695–702, July 1984.

[3] S. Sitharama Iyengar and Hsi Chang. Efficient algorithms to create and maintain balanced and threaded binary search trees. *Software: Practice and Experience*, 15(10):925–941, 1985.

QUESTIONS?