Implementing General Balanced trees

Arne Andersson
Computing Science Department
Information Technology
Uppsala University
Box 311, SE - 751 05 Uppsala, Sweden
www.csd.uu.se/~arnea

NOTE: This document is a combined LaTeXreport and C program. Remove the first character (%) in the LaTeXfile, and it will compile as a C program. You can run it and try it out yourself.

Abstract

We argue that the simplest possible balance criterion—let the tree take any shape as long as its height is logarithmic—can be used to produce very efficient code. A *general balanced tree* is a plain binary search tree with no balance information in the nodes. Balance is maintained by partial rebuilding. This data structure can be implemented very efficiently. (In experiments, it performs better than the skip list and the plain unbalanced tree.)

1 Introduction

Which is the fastest way to maintain a comparison-based dictionary? If we know that the keys are inserted and deleted in random (or random-like) order, a good alternative is the plain, unbalanced binary search tree. But what if conditions are such that skew updates occur often enough to prevent us from using a plain binary search tree? Then we can use the general balanced tree [3], which is nothing but a plain binary search tree but which has the feature of repairing its shape when needed.

Following the notation in [3], we do not distinguish between nodes and subtrees; the subtree v is the subtree rooted at the node v and T is the entire tree (or the root of the tree). Each internal node contains one element and all leaves are empty. The number of edges on the path from the root T to the node v is the *depth* of v. The largest number of edges from a node v to a leaf is the *height* of v, denoted $h(v)$. The *weight* of v equals the number of leaves in v and is denoted $|v|$. Thus, a tree containing n elements has a weight of $n + 1$.

A general balanced tree, or GB-tree, or GB(c)-tree, can take any shape as long as its height is logarithmic. For some constant c, we require that

$$h(T) \leq \lceil c \log |T| \rceil$$

The basic philosophy for maintaining GB-trees is

Do nothing until the tree becomes too high.

During insertion only the path down to the inserted node may become too long. Balance will be restored by rebuilding an appropriate subtree, located by a depth-first search along the insertion path. This strategy does not hold for deletion, but the skewness caused by deletions can be handled by rebuilding the entire tree periodically. There is no need to store any balance information in the nodes; only two global integers, telling the weight of T and the number of deletions made since the last time T was globally rebuilt, are needed.

Updates are performed in the following way:

Insertion: If the depth of the new leaf exceeds $\lceil c \log |T| \rceil$, we traverse the path bottom-up, counting the weights of subtrees. At the lowest node v, $h(v) > \lceil c \log |v| \rceil$, we make a partial rebuilding. (There will always be a node v satisfying $h(v) > \lceil c \log |v| \rceil$, since $h(T) > c \log |T|$.)

Deletion: When the number of deletions made since the last global rebuilding is lagrer than $d|T|$, d is a constant, we make a global rebuilding.

2 Discussion

The general balanced tree is a good candidate for implementing comparison-based dictionaries.

It is worth noting that the skip list is not very much slower than the GB-tree although it uses significantly more comparisons. This indicates that the skip list is very well-coded. When comparisons are expensive, the skip list is the slowest.

An interesting fact is that GB-trees actually run faster than unbalanced trees even for random data. This can be explained by the fact that rebalancing occurs very rarely. Fringe heuristics [4] probably gives a similar effect but without the guaranteed good behaviour for bad inputs.

If updates are skew, the GB-tree will run a bit slower than some alternative data structures, like skip lists, although still with high efficiency. For skew updates, the splay tree [5] is probably the fastest. Therefore, as an informal recommendation for a fast practical implementation, we state:

Use a GB-tree in the normal case and switch to a splay tree when updates turn out to be skew, i.e. when partial rebuildings occur often.

The two structures are both based on plain binary trees and therefore swithching between them is easy.

References

[1] A. Andersson. A note on the expected behaviour of binary tree traversals. *Computer Journal*, 33(5):471–472, 1990.

[2] A. Andersson. A note on searching in a binary search tree. *Software-Practice and Experience*, 21(10):1125–1128, 1991.

[3] A. Andersson. General balanced trees. *Journal of Algorithms*, 30:1–28, 1999. Conference version presented at WADS'89.

[4] P. V. Poblete and J. I. Munro. The analysis of a fringe heuristics for binary search trees. *Journal of Algorithms*, 6:335–350, 1985.

[5] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. In *Proc. ACM Symp. Theory of Computing*, 1983.

[6] Q. F. Stout and B. L. Warren. Tree rebalancing in optimal time and space. *Communications of the ACM*, 29(9):902–908, 1986.

Implementation

The code contains a dictionary implementation, where each dictionary contains a GB-tree plus the two counters neede for maintaining balance. The use of the dictionary should define the following

`ky_type`: The keys used for lookup.

`data_type`: The stored data.

`KY_ASSIGN(a,b)`: Give variable `a` the key value `b`.

`KY_LESS(a,b)`: Compare keys —tt a and —tt b. Returns poitive, negative, or zero. (For integers it is implemented as —tt (a-b).

`InitGlobal()`

Initializes the global array `minweight`, used in `insert` and `FixBalance` to check the balance criterion.

`dictptr construct_dict()`

Allocates a record of type `dict` containing an empty tree and the two integers needed for bookkeeping.

`leftrot, rightrot`

Standard procedures.

`static void Skew(noderef *t)`

Produce a right-skewed (sub-)tree by sliding along the rightmost path, making repeated right rotations.

`static void Split(noderef *t, long p1, long p2)`

Compress a skewed path of `p1` nodes into a path of `p2` nodes. In order to do this, we traverse the rightmost path making left rotations. At each right rotation, the path length is decreased by 1. Hence, we should make `(p1 - p2)` evenly spaced right rotations. To get the rotations evely spaced we use a counter to step from `p1-p2` to `p2*(p1-p2)` with increment equal to `p1 - p2`. Every time this counter reaches or exceeds a multiple of `p2` a rotation is performed.

`static void PerfectBalance(noderef *t, long w)`

A simple procedure for rebalancing a binary search tree. In fact, it is simpler than the Stout-Warren algorithm [6], the main improvement being the procedure `Split` above.

First, we `Skew` the tree, then we `Split` until the tree is balanced.

`static long TreeWeight(noderef t)`

We compute the weight of a binary search during a stack-based preorder traversal [1].

`void FixBalance (dictptr D, ky_type key, long d1)`

The procedure FixBalance reduces the length of a path in the tree.

(a) Traverse the path, putting all nodes in a stack `p`.

(b) Then, the path is traversed bottom-up and the weight of the subtrees are counted, until a subtree is found, which in itself does not satisfy the balance criterion of a GB(c)-tree.

(c) This subtree is rebuilt to perfect balance.

`void CreateNode(ky_type x, data_type in, noderef *t)`

Creates a new node during insertion.

```
noderef insert (dictptr D, ky_type key,
    data_type in)
```

When inserting, we proceed as in a plain binary search tree; the only difference is that we keep track of the depth of the inserted node. If this depth is too large, we call the procedure `FixBalance`. The search path is traversed with two-valued comparisons, for more discussion about the use of two-valued comparisons, see [2].

(a) Search down the tree, keep track of depth (`d1`).

(b) If the keys was found (pointed at by `candidate`), return without inserting.

(c) Create new node.

(d) If depth too large, call `FixBalance`.

`noderef lookup (dictptr D, ky_type key)`

Search an element in the tree. A pointer to the node containing the element is returned. If the element is not present, a pointer to the successor is returned. If there is no successor, a null pointer is returned. The algorithm uses two-valued comparisons. For more discussion of the algorithm, see [2].

(a) `candidate` will point at the last node where we turned right.

(b) Rigth turn, set `candidate`.

(c) If the key was there, it is pointed at by `candidate`. Otherwise, `candidate` will point at the successor, if any.

`void delete (dictptr D, ky_type key)`

Also here, two-valued comparisons are used. After a large number of deletions have been made, the entire tree is rebalanced.

(a) We have now traversed a search path. `last` points at the last node on the path and `candidate` points at the last right turn, we test if the element is there.

(b) Case 1: The deleted element is at the bottom of the tree.

(c) Case 2: The deleted element is internal. Since the element reference pointer given to the user should not change, we can not move elements between nodes. Instead, nodes must be swapped. In effect, the node `last` should be moved and replace the node `candidate`.

(d) Check if a global rebuilding should be made.

```
 ky_type keyval (dictptr D, noderef item)

 data_type *infoval (dictptr D, noderef item)

 int size (dictptr D)

 static void ClearTree(t)

 void clear (dictptr D)

 dictptr destruct_dict (dictptr D}
```

These functions are implemented the obvious way. (`Clear` is a simple recursive procedure that removes the nodes of a tree.)

```
C code: */
/*--------------- Standard things -----------------*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <math.h>


/*-------------- Tuning parameters ----------------*/

#define C 1.35 /* Other values could be used.      */
              /* as long as C > 1.                 */
#define maxdel 10 /* The number of deletions       */
                  /* since last global rebalancing */
                  /* is at most than 10 times the  */
                  /* tree weight.                  */
                  /* (Other constant possible.)    */

#define maxheight 40 /* We assume C * log n < 40.  */
                     /* Keep an eye on this one!   */

#define screenwidth 40 /* For displaying tree.     */

/*---------- User defined data types --------------*/

typedef long data_type;       /* User defined */
typedef int  ky_type;         /* User defined */
#define KY_ASSIGN(a,b) a=b    /* User defined */
#define KY_LESS(a,b) (a<b)    /* User defined */
#define KY_EQUAL(a,b) (a==b)  /* User defined */
#define IN_ASSIGN(a,b) a=b    /* User defined */

/*----- Procedures for external use -----------------

The following procedures are for exernal use:

dictptr construct_dict()
   Generates a new dictionary.

noderef insert (dictptr D, ky_type key,
                data_type in)
   Insert key and data, returns a reference.

noderef lookup (dictptr D, ky_type key)
   Returns a reference.

void delete (dictptr D, ky_type key)
   Delete key (and data)

ky_type keyval (dictptr D, noderef item)
   Get key via reference.

data_type *infoval (dictptr D, noderef item)
   Get a pointer to data.

int size (dictptr D)
   Number of stored items (= tree weight - 1)

void clear (dictptr D)
   Remove everything from dictionary.

dictptr destruct_dict (dictptr D)
  Destruct the dictionary.

Furthermore, the data type noderef is used by
the external user as reference to stored items.
(It is also a reference to tree nodes.)

---------------------------------------------------*/

typedef struct node {
  ky_type key;
  data_type data;
  struct node *left, *right;
} node;
typedef node *noderef;
typedef struct dict {
  noderef t;
  long weight, numofdeletions;
} dict;
typedef dict *dictptr;

static long minweight[maxheight + 1];


/*---------------------------*/
/* The tree is shown on the  */
/* screen in a simple way.   */
/* (We only display the      */
/*  higest levels.)          */
/*---------------------------*/

static void Display(t, depth)
noderef t;
long depth;
{
  if ((t == NULL) || (depth > 8))
    return;
  Display(t->left, depth + 1);
  printf("%*ld\n",
     (int)(screenwidth - depth * 4 - 4), t->key);
  Display(t->right, depth + 1);
}

/*-------------- construction -------------------*/

static void InitGlobal()
{ long h;

  for (h = 1; h <= maxheight; h++)
   minweight[h]
     = (long)(exp((h-1)/C*log(2.0)) + 0.5) + 1;
}

dictptr construct_dict()
{ dictptr p;

  InitGlobal();
  p = (dictptr)malloc(sizeof(dict));
  p->t = NULL;
  p->weight = 1;
  p->numofdeletions = 0;
  return p;
}

/*---------------- Rebalancing ------------------*/

static void leftrot(noderef *t)
{ node *tmp;

  tmp = *t;
  *t = (*t)->right;
  tmp->right = (*t)->left;
  (*t)->left = tmp;
}

static void rightrot(noderef *t)
{ node *tmp;

  tmp = *t;
  *t = (*t)->left;
  tmp->left = (*t)->right;
  (*t)->right = tmp;
}

static void Skew(noderef *t)
{
  do {
    while ((*t)->left)
      rightrot(t);
    t = &(*t)->right;
  } while (*t);
}

static void oldSplit(noderef *t, long p1, long p2)
{ float tmp; long d;

  tmp = (float)(p1 - p2)/p2;
  for (d = 1; d <= p2; d++) {
   if ((long)(d*tmp) > (long)((d-1)*tmp))
      leftrot(t);
    t = &(*t)->right;
  }
}

static void Split(noderef *t, long p1, long p2)
{ long incr = p1 - p2;
  long count = 0;
  long i;

  for (i = p2; i > 0; i--) {
    count += incr;
    if (count >= p2) {
      leftrot(t);
      count -= p2; /* incr <= p2 */
    }
    t = &(*t)->right;
  }
}

static void PerfectBalance(noderef *t, long w)
{ long b;

  Skew(t);
  b = 1;
  while (b <= w)
    b *= 2;
  b /= 2;
  if (b != w)
    Split(t, w - 1, b - 1);
  while (b > 2) {
    Split(t, b - 1, b / 2 - 1);
    b /= 2;
  }
}

#define NULLSTACK   {top = 0; stack[0]=NULL;}
#define PUSH(t)     {top++; stack[top]=t;}
#define POP(t)      {t=stack[top]; top--;}
static long TreeWeight(noderef t)
{ node *stack[maxheight]; long top, w;

  w = 1;
  NULLSTACK;
  while (t) {
    while (t->left) {
      w++;
      if (t->right)
        PUSH(t->right);
      t = t->left;
    }
    w++;
    if (!t->right)
      POP(t)
    else
      t = t->right;
  }
  return w;
}

void FixBalance (dictptr D, ky_type key, long d1)
{ long d2;
  noderef *p[maxheight + 1];
  long w;

  p[1] = &(D->t);                             /* a */
  for (d2 = 1; d2 < d1; d2++) {
    if (KY_LESS(key,(*p[d2])->key))
      p[d2 + 1] = &(*p[d2])->left;
    else
      p[d2 + 1] = &(*p[d2])->right;
  }
  w = 2;                                    /* b */
  do {
    d2--;
    if (&(*p[d2])->left == p[d2 + 1])
      w = w + TreeWeight((*p[d2])->right);
    else
      w = w + TreeWeight((*p[d2])->left);
  } while (w >= minweight[d1 - d2 + 1]);
  PerfectBalance(p[d2], w);                 /* c */
}

void CreateNode(ky_type x, data_type in, noderef *t)
{
  *t = (node *)malloc(sizeof(node));
  KY_ASSIGN((*t)->key, x);
  IN_ASSIGN((*t)->data, in);
  (*t)->left = NULL;
  (*t)->right = NULL;
}

noderef insert (dictptr D, ky_type key,
                data_type in)
{ long d1;
  noderef *candidate, *p, newnode;

  d1 = 1;
  p = &(D->t);
  candidate = NULL;
  while (*p) {                                /* a */
    if (KY_LESS(key,(*p)->key))
      p = &(*p)->left;
    else {
      candidate = p;
      p = &(*p)->right;
    }
    d1++;
  }
  if (candidate &&                            /* b */
  (KY_EQUAL((*candidate)->key, key)))
    return *candidate;
  CreateNode(key, in, p);                     /* c */
  newnode = *p;
  D->weight++;
  if (D->weight < minweight[d1])             /* d */
    FixBalance(D, key, d1);
  return newnode;
}

noderef lookup (dictptr D, ky_type key)
{ register noderef t, candidate;

  t = D->t;
  candidate = NULL;                           /* a */
  while (t) {
    if (KY_LESS (key, t->key))
      t = t->left;
    else {
      candidate = t;                          /* b */
      t = t->right;
    }
  }
  return candidate;                           /* c */
}

void delete (dictptr D, ky_type key)
{ noderef *candidate, *last, tmp, *t;

  t = &(D->t);
  candidate = NULL;
  while (*t) {
    last = t;
    if (KY_LESS(key,(*t)->key))
      t = &(*t)->left;
    else {
      candidate = t;
      t = &(*t)->right;
    }
  }
  if (candidate &&                             /* a */
  (KY_EQUAL((*candidate)->key, key))) {
    D->numofdeletions++;
    D->weight--;
    tmp = *last;
    if (candidate == last) {                   /* b */
      *last = (*last)->left;
      free(tmp);
    }
    else {                                     /* c */
      *last = (*last)->right;
      tmp->right = (*candidate)->right;
      tmp->left = (*candidate)->left;
      free(*candidate);
      *candidate = tmp;
    }
  }
  if (D->numofdeletions > maxdel*D->weight    /* d */
  && D->weight > 3) {
    PerfectBalance(&(D->t),D->weight);
    D->numofdeletions = 0;
  }
}

ky_type keyval (dictptr D, noderef item)
{
  return((noderef)item)->key;
 }/*keyval*/

data_type *infoval (dictptr D, noderef item)
{
  return &((noderef)item)->data;
}/*infoval*/

int size (dictptr D)
{
  return D->weight-1;
} /*size*/

static void ClearTree(t) noderef *t;
{
  if (!*t)
    return;
  ClearTree(&(*t)->right);
  ClearTree(&(*t)->left);
  free(*t);
  *t = NULL;
}

void clear (dictptr D)
{
  ClearTree (&(D->t));
  D->weight = 1;
  D->numofdeletions = 0;
}/*clear*/

dictptr destruct_dict (dictptr D)
{
  clear(D);
  free(D);
} /*destruct_dict */


/*-----------------------------*/



void Help()
{
  printf("Write\n");
  printf(" i x to insert integer x,\n");
  printf(" d x to delete x,\n");
  printf(" c to clear the tree,\n");
  printf(" b to balance the entire tree,\n");
  printf(" q to quit, \n");
  printf(" ? to see this message. \n");
}


/*----------------------*/
/* A small test program. */
/*----------------------*/

static void Go()
{
  dictptr thedict;
  char command;
  ky_type x;
  noderef temp;
  int ok;

  thedict = construct_dict();
  Help();
  do {
    printf ("treeweight %ld \n",thedict->weight);
    printf("Enter command, write ? for help.\n");
    command = getchar();
    if (command == '\n')
      command = ' ';
    switch (command) {

    case 'i':
      scanf("%ld%*[^\n]", &x);
      getchar();
      temp = lookup (thedict, x);
      if (temp)
        { if (temp->key == x)
          { printf("No insertion, %ld is already present.\n", x);
            break;
         }}
        { temp = insert (thedict, x, 0);
          Display(thedict->t, 0L);
         if (temp->key != x)
            printf("something is wrong.\n");
          temp = lookup (thedict, x);
          if (temp->key != x)
    printf("something is wrong.\n");
        }
      break;

    case 'd':
      scanf("%ld%*[^\n]", &x);
      getchar();
      temp = lookup (thedict, x);
      if (!temp)
        { printf("Sorry, could not find %ld\n", x); }
      else if (temp->key != x)
        { printf("Sorry, could not find %ld\n", x); }
      else
        { delete (thedict, x);
        Display(thedict->t, 0L);
      }
    break;

  case 'b':
    scanf("%*[^\n]");
    getchar();
    PerfectBalance(&thedict->t, thedict->weight);
    Display(thedict->t, 0L);
    break;

  case 'c':
    scanf("%*[^\n]");
    getchar();
    clear(thedict);
    Display(thedict->t, 0L);
    break;

  case 'q':
    printf("  Bye\n");
    break;

  case '?':
    scanf("%*[^\n]");
    getchar();
    Help();
    break;

  default:
    scanf("%*[^\n]");
    getchar();
    printf("I don''t understand.\n");
    printf("Write ? for help. \n");
    break;
  }
} while (command != 'q');
}


main(argc, argv)
int argc;
char *argv[];
{
  Go();
  exit(1);
}


/*
```