CS360 Lecture notes -- Red-Black Trees (JRB)

- James S. Plank
- Directory: **~plank/plank/classes/cs360/360/www-home/notes/JRB**
- Lecture notes: **http://www.cs.utk.edu/~plank/plank/classes/cs360/360/notes/JRB**
- Original notes written August, 1999.
- Latest revision: January, 2015.

Compiling

In order to use the red-black tree library, you should include the file "jrb.h", which can be found in **/home/plank/cs360/include**. Instead of including the full path name in your C file, just do:
`#include "jrb.h",`
and then compile the program with:
`gcc -I/home/plank/cs360/include`
When you link your object files to make an executable, follow the directions in the libfdr lecture notes.

The makefile in this directory does both of these things for you.

Red-Black Trees

Rb-trees are data structures based on balanced binary trees. You don't need to know how they work -- just that they do work, and all operations are in *O(log(n))* time, where *n* is the number of elements in the tree. (If you really want to know more about red-black trees, let me know and I can point you to some texts on them).

The main struct for rb-trees is the **JRB**. Like dllists, all rb-trees have a header node. You create a rb-tree by calling **make_jrb()**, which returns a pointer to the header node of an empty rb-tree. This header points to the main body of the rb-tree, which you don't need to care about, and to the first and last external nodes of the tree. These external nodes are hooked together with flink and blink pointers, so that you can view rb-trees as being dlists with the property that they are sorted, and you can find any node in the tree in *O(log(n))* time.

Like dllists, each node in the tree has a **val** field, which is a Jval. Additionally, each node has a **key** field, which is also a Jval. The rb-tree tree makes sure that the keys are sorted. How they are sorted depends on the tree.

_str, _int, _dbl, _gen

The jrb tree routines in **jrb.h/jrb.c** implement four types of insertion/searching routines. The insertion routines are:

- **JRB jrb_insert_str(JRB tree, char *key, Jval val)**: insert a new node into the tree using a standard character string as the key. **Strcmp()** is used as the comparison function. See **strsort.c** for a simple example of sorting standard input lexicographically with **jrb_insert_str()**.

  Note that it returns a pointer to the new jrb tree node. Also note that if the key is already in the tree, then it still creates a new node and puts it into the tree. No guarantees are made concerning the relative ordering of duplicate keys.

  Even though the key is a string, it will be converted into a **Jval** in the new node. Thus, if you want to get at the key of node **s**, you should either use **jval_s(s->key)** or **s->key.s**.

- **JRB jrb_insert_int(JRB tree, int key, Jval val)**: insert into the tree using an integer as the key. See **nsort.c** for an example of this.
- **JRB jrb_insert_dbl(JRB tree, double key, Jval val)**: insert into the tree using a **double** as the key.
- **JRB jrb_insert_gen(JRB tree, Jval key, Jval val, int (*func)(Jval, Jval))**: Now, your key is a **jval**. You provide a comparison function **func()**, which takes two **Jval**'s as arguments, and returns:
  - a negative integer if the first key is less than the second.
  - a positive integer if the first key is greater than the second.
  - zero if the keys are equal.

  This lets you do more sophisticated things than simply sorting with integers and strings. For example, **strisort.c** sorts strings but ignores case. **strrsort2.c** sorts strings in reverse order. Read these over.

You can't mix and match comparison functions within the same tree. In other words, you shouldn't insert some keys with **jrb_insert_str()** and some with **jrb_insert_int()**. To do so will be begging for a core dump.

To find keys, you use one of **jrb_find_str()**, **jrb_find_int()**, **jrb_find_dbl()** or **jrb_find_gen()**. Obviously, if you inserted keys with **jrb_insert_str()**, then you should use **jrb_find_str()** to find them. If the key that you're looking for is not in the tree, then **jrb_find_*xxx*()** returns NULL.

Finally, there are also: **jrb_find_gte_str()**, **jrb_find_gte_int()**, **jrb_find_gte_dbl()** and **jrb_find_gte_gen()**. These return the jrb tree node whose key is either equal to the specified key, or whose key is the smallest one greater than the specified key. If the specified key is greater than any in the tree, it will return a pointer to the sentinel node. It has an argument **found** that is set to tell you if the key was found or not.

You may use the macros **jrb_first()**, **jrb_last()**, **jrb_prev()** and **jrb_next()**, just like their counterparts in the dllist library.

Also, to delete a tree, you use **jrb_free_tree()**. This does not call **free()** on keys and vals -- it simply deletes all memory associated with the nodes on the tree.

Example programs:

- **strsort.c**: Uses red-black trees to sort standard input lexicographically.
- **strrsort1.c**: Uses red-black trees to sort standard input lexicographically in reverse order. It does this by traversing the tree in reverse order.
- **strrsort2.c**: Uses red-black trees to sort standard input lexicographically in reverse order. It does this by creating a new comparison function **revcomp**, which simply returns -**strcmp()**. Now the tree sorts in reverse order, so it is traversed in the forward direction.
- **strusort.c**: Uses red-black trees to sort standard input lexicographically, and it removes duplicate lines. It does this by checking for a line before inserting it into the tree.
- **strisort.c**: Uses red-black trees to sort standard input lexicographically, ignoring upper and lower case. It does this by creating a new comparison function **ucomp**, which duplicates **strcmp()**'s functionality but ignores case.
- **nsort.c**: Uses red-black trees to sort like **sort -n** -- i.e. it treats each line as an integer, and sorts it that way. If the lines are not integers, or there are duplicate lines, anything goes.
- **nsort2.c**: Uses red-black trees to sort like **sort -n** only now if two lines have the same **atoi()** value, then they are sorted lexicographically. This uses **jrb_insert_gen()**.
- **nsort3.c**: Same as **nsort2**, but instead it uses a two-level red-black tree. If this is confusing to you, please read the next example, which does a very similar thing.

A two-level tree example

Suppose we are reading input composed of names and scores. Names can be any number of words, and scores are integers. Each line contains a name followed by a score, and there can be any amount of whitespace between words in the input file. An example is **input-nn.txt**. As you can see from the first 10 lines, it's kind of messy, but it conforms to the format:

```
UNIX> head -n 10 input-nn.txt
     Molly Skyward                                60
Taylor   Becloud                                47
   Brody   Hysteresis                         56
Tristan   Covenant                             75
Adam   Dyeing                              38
Brianna   Domain                           54
     Jonathan   Value                                5
        Max Head                                    48
Adam   Bobbie                             68
        Jack Indescribable                      99
```

```
UNIX>
```

Suppose we want to process this input file by creating a **Person** struct for each line that has the person's name and score:

```
typedef struct {
  char *name;
  int score;
} Person;
```

And then suppose we want to print the people, sorted first by score, and then by name. We want the format of our output to be the name, left justified and padded to 40 characters, followed by the score padded to two characters. I'm going to write this program three times. I believe the last of the three is the best, but it's a good exercise to go over all three.

ni_sort1.c -- creating a sorting key

The first program is *ni_sort1.c*. It reads each person into a struct and then creates a string that it uses as a comparison string. That key contains the score, right justified and padded to ten characters, and then the name. Thus, when you use the key to insert people into a red-black tree, the tree is sorted in the order that you want.

Let's look at the program:

```
typedef struct {
  char *name;
  int score;
  char *key;
} Person;

main()
{
  JRB t, tmp;
  IS is;
  Person *p;
  int nsize, i;

  is = new_inputstruct(NULL);
  t = make_jrb();

  while (get_line(is) >= 0) {
    if (is->NF > 1) {

      /* Each line is name followed by score.  The score is easy to get. */

      p = malloc(sizeof(Person));
      p->score = atoi(is->fields[is->NF-1]);

      /* The name is a different matter, because names may be composed of any
         number of words with any amount of whitespace.  We want to create a
         name string that has each word of the name separated by one space.
```

```
         Our first task is to calculate the size of our name. */

      nsize = strlen(is->fields[0]);
      for (i = 1; i < is->NF-1; i++) nsize += (strlen(is->fields[i])+1);

      /* We then allocate the string and copy the first word into the string.
*/

      p->name = (char *) malloc(sizeof(char)*(nsize+1));
      strcpy(p->name, is->fields[0]);

      /* We copy in the remaining words, but note how we do so by calling
strcpy
         into the exact location of where the name goes, rather than, say,
repeatedly
         calling strcat() as we would do in a C++-like solution.  This is
much more
         efficient (not to mention inconvenient) than using strcat(). */

      nsize = strlen(is->fields[0]);
      for (i = 1; i < is->NF-1; i++) {
        p->name[nsize] = ' ';
        strcpy(p->name+nsize+1, is->fields[i]);
        nsize += strlen(p->name+nsize);
      }

      /* We create a key for inserting into the red-black tree.  That is
going
         to be the score, padded to 10 characters, followed by the name.  We
         allocate (nsize+12) characters: nsize for the name, 10 for the
score,
         one for the space, and one for the null character. */

      p->key = (char *) malloc(sizeof(char) * (nsize + 12));
      sprintf(p->key, "%10d %s", p->score, p->name);

      jrb_insert_str(t, p->key, new_jval_v((void *) p));
    }
  }

  jrb_traverse(tmp, t) {
    p = (Person *) tmp->val.v;
    printf("%-40s %2d\n", p->name, p->score);
  }
  exit(0);
}
```

In a rare fit of niceness, I have commented this program. You should pay attention to how I created the name, as it how you do such a thing efficiently in C. You'll be tempted to simply allocate a giant string and then use **strcat()** to create the name. That's the paradigm that you'd use in C++. However, that's inefficient because of **strcat()** (see the commentary on **strcat()** in these lecture notes).

Instead, you make one pass over the name to calculate the size of the string, and then you allocate the string. You then use **strcpy()** to copy each word of the name into its proper place. Yes, the code is ugly, but it is the most efficient way to do it.

After creating the name, we create the comparison key, and note how we have to calculate its size and allocate it. We insert the key into the tree with the person struct as a val, and when we traverse it, we get the order that we want. I used "%10d" for the score, because I know the maxiumum integer is 2^{31}, which is roughly 2,000,000,000. I want the scores all aligned and right justified in the keys, because that way it will sort the integers properly. This is because space has a lower ASCII value than numbers.

It's always good to sanity-check your programs to make sure that the output makes sense, and that you have no bugs or typos. Below, I do the following. First, I make sure that the input and output files have the same number of lines and words:

```
UNIX> ni_sort1 < input-nn.txt > output-1.txt
UNIX> wc input-nn.txt output-1.txt
  500  1583 24446 input-nn.txt
  500  1583 22000 output-1.txt
 1000  3166 46446 total
UNIX>
```

They differ in the number of characters, because they format the words and scores differently. All is good so far. Next, I sanity-check the beginning and the ending to make sure that they look right:

```
UNIX> head output-1.txt
Addison Paige Chain                        0
Eli Gneiss                                 0
Ella Craftsperson                          0
Lilly Gianna Zen                           0
Matthew Stiffen                            0
Evan Boorish                               1
Isaiah Metabolism                          1
Mason Fourier                              1
Xavier Agave                               1
Daniel Berman                              2
UNIX> tail output-1.txt
Layla Option                              96
Lucas Fay Jr                              96
Madeline Task                             96
Sofia Nitrous                             96
Gianna Sinh                               97
Lucy Quaternary                           97
Sophia Contrariety                        97
Charlie Lucas Vine                        98
Jack Indescribable                        99
Lily Span                                 99
UNIX>
```

Then I do a sampling, to make sure that one of the score values has the right output. Here, I do that with 96:

```
UNIX> grep 96 output-1.txt
Alexander Bstj                            96
Grace Globulin                            96
```

```
Jonathan Blanket Esq                      96
Kaitlyn Thwack                            96
Layla Option                              96
Lucas Fay Jr                              96
Madeline Task                             96
Sofia Nitrous                             96
UNIX> grep 96 input-nn.txt
   Grace Globulin                          96
        Jonathan   Blanket Esq                96
Alexander   Bstj                         96
     Madeline Task                                96
     Layla Option                                 96
     Kaitlyn Thwack                               96
     Sofia Nitrous                                96
        Lucas Fay Jr                           96
UNIX> grep 96 input-nn.txt | sed 's/^ *//' | sort
Alexander   Bstj                         96
Grace Globulin                          96
Jonathan   Blanket Esq                   96
Kaitlyn Thwack                                96
Layla Option                                  96
Lucas Fay Jr                            96
Madeline Task                                 96
Sofia Nitrous                                 96
UNIX>
```

Ok, I'm good. Note, that's not a conclusive test. Here's a more conclusive test (only look at this if you're really interested. I do this stuff all the time, so I'm good at it. However, I will urge you to learn sed and awk, because they are super-powerful) (Oh, and I'm not testing you on this stuff. I'm just trying to help you become more effective with Unix tools):

```
UNIX> sed 's/^ *//' input-nn.txt | head
Molly Skyward                                 60
Taylor   Becloud                                47
Brody   Hysteresis                        56
Tristan   Covenant                              75
Adam   Dyeing                             38
Brianna   Domain                           54
Jonathan   Value                                 5
Max Head                                      48
Adam   Bobbie                             68
Jack Indescribable                      99
UNIX> sed 's/^ *//' input-nn.txt | sed 's/  */-/g' | head
Molly-Skyward-60-
Taylor-Becloud-47
Brody-Hysteresis-56
Tristan-Covenant-75
Adam-Dyeing-38
Brianna-Domain-54
Jonathan-Value-5
Max-Head-48
Adam-Bobbie-68
Jack-Indescribable-99
UNIX> sed 's/^ *//' input-nn.txt | sed 's/  */-/g' | sed 's/-\([0-9]\)/ \1/'
| head
Molly-Skyward 60-
Taylor-Becloud 47
Brody-Hysteresis 56
```

```
Tristan-Covenant 75
Adam-Dyeing 38
Brianna-Domain 54
Jonathan-Value 5
Max-Head 48
Adam-Bobbie 68
Jack-Indescribable 99
UNIX> sed 's/^ *//' input-nn.txt | sed 's/  */-/g' | sed 's/-\([0-9]\)/ \1/'
| sed 's/-$//' | head
Molly-Skyward 60
Taylor-Becloud 47
Brody-Hysteresis 56
Tristan-Covenant 75
Adam-Dyeing 38
Brianna-Domain 54
Jonathan-Value 5
Max-Head 48
Adam-Bobbie 68
Jack-Indescribable 99
UNIX> sed 's/^ *//' input-nn.txt | sed 's/  */-/g' | sed 's/-\([0-9]\)/ \1/'
| sed 's/-$//' | awk '{ print $2, $1 }' | head
60 Molly-Skyward
47 Taylor-Becloud
56 Brody-Hysteresis
75 Tristan-Covenant
38 Adam-Dyeing
54 Brianna-Domain
5 Jonathan-Value
48 Max-Head
68 Adam-Bobbie
99 Jack-Indescribable
UNIX> sed 's/^ *//' input-nn.txt | sed 's/  */-/g' | sed 's/-\([0-9]\)/ \1/'
| sed 's/-$//' | awk '{ print $2, $1 }' | sort -n | head
0 Addison-Paige-Chain
0 Eli-Gneiss
0 Ella-Craftsperson
0 Lilly-Gianna-Zen
0 Matthew-Stiffen
1 Evan-Boorish
1 Isaiah-Metabolism
1 Mason-Fourier
1 Xavier-Agave
2 Daniel-Berman
UNIX> sed 's/^ *//' input-nn.txt | sed 's/  */-/g' | sed 's/-\([0-9]\)/ \1/'
| sed 's/-$//' | awk '{ print $2, $1 }' | sort -n | awk '{ printf "%-40s
%2d\n", $2, $1 }' | head
Addison-Paige-Chain                      0
Eli-Gneiss                               0
Ella-Craftsperson                        0
Lilly-Gianna-Zen                         0
Matthew-Stiffen                          0
Evan-Boorish                             1
Isaiah-Metabolism                        1
Mason-Fourier                            1
Xavier-Agave                             1
Daniel-Berman                            2
```

```
UNIX> sed 's/^ *//' input-nn.txt | sed 's/  */-/g' | sed 's/-\([0-9]\)/ \1/'
| sed 's/-$//' | awk '{ print $2, $1 }' | sort -n | awk '{ printf "%-40s
%2d\n", $2, $1 }' | sed 's/-/ /g' | head
Addison Paige Chain                      0
Eli Gneiss                               0
Ella Craftsperson                        0
Lilly Gianna Zen                         0
Matthew Stiffen                          0
Evan Boorish                             1
Isaiah Metabolism                        1
Mason Fourier                            1
Xavier Agave                             1
Daniel Berman                            2
UNIX> sed 's/^ *//' input-nn.txt | sed 's/  */-/g' | sed 's/-\([0-9]\)/ \1/'
| sed 's/-$//' | awk '{ print $2, $1 }' | sort -n | awk '{ printf "%-40s
%2d\n", $2, $1 }' | sed 's/-/ /g' > junk.txt
UNIX> openssl md5 junk.txt output-1.txt
MD5(junk.txt)= 4eee1503231b23c0052d9b3c57b1cd50
MD5(output-1.txt)= 4eee1503231b23c0052d9b3c57b1cd50
UNIX>
```

Now, let's write the program a second time, only this time we simply insert the **Person** struct as a key, and write a comparison function to compare keys. The program is in **ni_sort2.c**, and here are the relevant parts:

```
typedef struct {
  char *name;
  int score;
} Person;

int compare(Jval j1, Jval j2)
{
  Person *p1, *p2;

  p1 = (Person *) j1.v;
  p2 = (Person *) j2.v;

  if (p1->score > p2->score) return 1;
  if (p1->score < p2->score) return -1;
  return strcmp(p1->name, p2->name);
}

main()
{
  JRB t, tmp;
  IS is;
  Person *p;
  int nsize, i;

  is = new_inputstruct(NULL);
  t = make_jrb();

  while (get_line(is) >= 0) {

      ....  do the reading and the creation of the person

      /* We now insert using jrb_insert_gen, with the person struct as a key.
```

```
*/

      jrb_insert_gen(t, new_jval_v((void *) p), new_jval_v(NULL), compare);
    }
  }

  jrb_traverse(tmp, t) {
    p = (Person *) tmp->key.v;
    printf("%-40s %2d\n", p->name, p->score);
  }
  exit(0);
}
```

This doesn't require much comment, except to remember that they keys are jvals, so you must typecast them to the type that you want in the comparison function. It's easy to affirm that the output of this program is the same as the last:

```
UNIX> ni_sort2 < input-nn.txt | head
Addison Paige Chain                      0
Eli Gneiss                               0
Ella Craftsperson                        0
Lilly Gianna Zen                         0
Matthew Stiffen                          0
Evan Boorish                             1
Isaiah Metabolism                        1
Mason Fourier                            1
Xavier Agave                             1
Daniel Berman                            2
UNIX> ni_sort2 < input-nn.txt > output-2.txt
UNIX> openssl md5 output-*.txt
MD5(output-1.txt)= 4eee1503231b23c0052d9b3c57b1cd50
MD5(output-2.txt)= 4eee1503231b23c0052d9b3c57b1cd50
UNIX>
```

The final program is **ni_sort3.c**, and it uses a two-level tree. The first level is keyed on scores, and there is only one tree node per score. The val of each node is a red-black tree keyed on name, with the person struct in the val. Take a look at how we create the tree and print it out. I personally like this solution the best, and it's good practice for you, because you will be creating data structures like these all the time:

```
main()
{
  JRB t, tmp, person_tree, t2;
  IS is;
  Person *p;
  int nsize, i;

  is = new_inputstruct(NULL);
  t = make_jrb();

  while (get_line(is) >= 0) {

    if (is->NF > 1) {
```

```
      ... Read and create the person

      /* To insert the person, we first test to see if the score is in the
         tree.  If it is not, we create it with an empty red-black tree as a
val.
         In either case, we insert the name into the second-level tree. */

      tmp = jrb_find_int(t, p->score);
      if (tmp == NULL) {
        person_tree = make_jrb();
        jrb_insert_int(t, p->score, new_jval_v((void *) person_tree));
      } else {
        person_tree = (JRB) tmp->val.v;
      }

      jrb_insert_str(person_tree, p->name, new_jval_v((void *) p));
    }
  }

  /* To print the people, we need to do a nested, two-level recursion */

  jrb_traverse(tmp, t) {
    person_tree = (JRB) tmp->val.v;
    jrb_traverse(t2, person_tree) {
      p = (Person *) t2->val.v;
      printf("%-40s %2d\n", p->name, p->score);
    }
  }
  exit(0);
}
```

Again, we can double-check to make sure it's correct:

```
UNIX> ni_sort3 < input-nn.txt | head
Addison Paige Chain                      0
Eli Gneiss                               0
Ella Craftsperson                        0
Lilly Gianna Zen                         0
Matthew Stiffen                          0
Evan Boorish                             1
Isaiah Metabolism                        1
Mason Fourier                            1
Xavier Agave                             1
Daniel Berman                            2
UNIX> ni_sort3 < input-nn.txt > output-3.txt
UNIX> openssl md5 output-*.txt
MD5(output-1.txt)= 4eee1503231b23c0052d9b3c57b1cd50
MD5(output-2.txt)= 4eee1503231b23c0052d9b3c57b1cd50
MD5(output-3.txt)= 4eee1503231b23c0052d9b3c57b1cd50
UNIX>
```

Another Example: ``Golf''

Here's another typical example of using a red-black tree. It doesn't do much beyond the last example, but I include it for practice. Suppose we have a bunch of files with golf scores. Examples are in**1998_Majors** and **1999_Majors**. The format of these files is:

```
Name     sunday-score F total-score
```

For example, the first few lines of **1999_Majors/Masters** are:

```
Jose Maria Olazabal                -1 F -8
Davis Love III                     -1 F -6
Greg Norman                        +1 F -5
Bob Estes                          +0 F -4
Steve Pate                         +1 F -4
David Duval                        -2 F -3
Phil Mickelson                     -1 F -3
...
```

Note that the name can have any number of words.

Now, suppose that we want to do some data processing on these files. For example, suppose we'd like to sort each player so that we first print out the players that have played the most tournaments, and then within that, we sort by the player with the lowest average score.

This is what **golf.c** does. It takes score files on the command line, then reads in all the players and scores. Then it sorts them by number of tournaments/average score, and prints them out in that order, along with their score for each tournament. For example, look at **score1**:

```
Jose Maria Olazabal                -1 F -8
Davis Love III                     -1 F -6
Greg Norman                        +1 F -5
```

and **score2**:

```
Greg Norman                         +1  F +9
David Frost                         +3  F +10
Davis Love III                      -2  F +11
```

The **golf** program reads in these two files, and ranks the four players by number of tournaments, and then average score:

```
UNIX> golf score1 score2
Greg Norman                             :   2 tournaments :    2.00
   -5 : score1
    9 : score2
Davis Love III                          :   2 tournaments :    2.50
   -6 : score1
   11 : score2
Jose Maria Olazabal                     :   1 tournament  :   -8.00
   -8 : score1
David Frost                             :   1 tournament  :   10.00
   10 : score2
```

Ok, now how does **golf** work? Well it works in three phases. In the first phase, it reads the input files to create a struct for each golfer. The data structure for this is a red-black tree keyed on the golfer's name, and whose val fields are **Golfer** structs that have the following defintion:

```
typedef struct {
  char *name;
  int ntourn;
  int tscore;
```

```
  Dllist scores;
} Golfer;
```
The first three fields are obvious. The last field is a list of the golfer's scores. Each element of the list points to a **Score** struct with the following definition:
```
typedef struct {
  char *tname;             /* File name */
  int score;               /* Total score */
} Score;
```
Note, in each file, we are going to ignore the ``sunday score."

So, to read in the golfers, we create the jrb tree **golfers**, and then read in each line of each input file. For each line, we construct the golfer's name, and then we look to see if the golfer has an entry in the**golfers** tree. If there is no such entry, then one is created. Once the entry is found/created, the score for that file is added. When all the files have been read, phase 1 is completed:

```
  Golfer *g;
  Score *s;
  JRB golfers, rnode;
  int i, fn;
  int tmp;
  IS is;
  char name[1000];
  Dllist dnode;

  golfers = make_jrb();

  for (fn = 1; fn < argc; fn++) {
    is = new_inputstruct(argv[fn]);
    if (is == NULL) { perror(argv[fn]); exit(1); }

    while(get_line(is) >= 0) {

      /* Error check each line */

      if (is->NF < 4 || strcmp(is->fields[is->NF-2], "F") != 0 ||
          sscanf(is->fields[is->NF-1], "%d", &tmp) != 1 ||
          sscanf(is->fields[is->NF-3], "%d", &tmp) != 1) {
        fprintf(stderr, "File %s, Line %d: Not the proper format\n",
          is->name, is->line);
        exit(1);
      }

      /* Construct the golfer's name. This is lazy code that is inefficient,
by the way */

      strcpy(name, is->fields[0]);
      for (i = 1; i < is->NF-3; i++) {
        strcat(name, " ");
        strcat(name, is->fields[i]);
      }

      /* Search for the name */
```

```
      rnode = jrb_find_str(golfers, name);

      /* Create an entry if none exists. */

      if (rnode == NULL) {
        g = (Golfer *) malloc(sizeof(Golfer));
        g->name = strdup(name);
        g->ntourn = 0;
        g->tscore = 0;
        g->scores = new_dllist();
        jrb_insert_str(golfers, g->name, new_jval_v(g));
      } else {
        g = (Golfer *) rnode->val.v;
      }

      /* Add the information to the golfer's struct */

      s = (Score *) malloc(sizeof(Score));
      s->tname = argv[fn];
      s->score = atoi(is->fields[is->NF-1]);
      g->ntourn++;
      g->tscore += s->score;
      dll_append(g->scores, new_jval_v(s));
    }

    /* Go on to the next file */

    jettison_inputstruct(is);
  }
```

Now, this gives us all the information on the golfers, but they are sorted by the golfers' names, not by number of tournaments / average score. Thus, in phase 2, we construct a second red-black tree which will sort the golfers correctly. To do this, we need to construct our own comparison function that compares golfers by number of tournaments / average score. Here is the comparison function:

```
int golfercomp(Jval j1, Jval j2)
{
  Golfer *g1, *g2;

  g1 = (Golfer *) j1.v;
  g2 = (Golfer *) j2.v;

  if (g1->ntourn > g2->ntourn) return 1;
  if (g1->ntourn < g2->ntourn) return -1;
  if (g1->tscore < g2->tscore) return 1;
  if (g1->tscore > g2->tscore) return -1;
  return 0;
}
```

And here is the part of **main** where the second red-black tree is built:

```
  sorted_golfers = make_jrb();

  jrb_traverse(rnode, golfers) {
    jrb_insert_gen(sorted_golfers, rnode->val, JNULL, golfercomp);
  }
```

Note, you pass a **Jval** to **jrb_insert_gen**.

Finally, the third phase is to traverse the **sorted_golfers** tree, printing out the correct information for each golfer. This is straightforward, and done below:

```
  jrb_rtraverse(rnode, sorted_golfers) {
    g = (Golfer *) rnode->key.v;
    printf("%-40s : %3d tournament%1s : %7.2f\n", g->name, g->ntourn,
           (g->ntourn == 1) ? "" : "s",
           (float) g->tscore / (float) g->ntourn);
    dll_traverse(dnode, g->scores) {
      s = (Score *) dnode->val.v;
      printf("  %3d : %s\n", s->score, s->tname);
    }
  }
```

Try it out. You'll see that Tiger Woods did the best in all four majors this year:

```
UNIX> golf 1999_Majors/*
Tiger Woods                              :   4 tournaments :    0.25
   10 : 1999_Majors/British_Open
    1 : 1999_Majors/Masters
  -11 : 1999_Majors/PGA_Champ
    1 : 1999_Majors/US_Open
Colin Montgomerie                        :   4 tournaments :    3.75
   12 : 1999_Majors/British_Open
   -1 : 1999_Majors/Masters
   -6 : 1999_Majors/PGA_Champ
   10 : 1999_Majors/US_Open
Davis Love III                           :   4 tournaments :    4.50
   10 : 1999_Majors/British_Open
   -6 : 1999_Majors/Masters
    5 : 1999_Majors/PGA_Champ
    9 : 1999_Majors/US_Open
Jim Furyk                                :   4 tournaments :    4.50
   11 : 1999_Majors/British_Open
    0 : 1999_Majors/Masters
   -4 : 1999_Majors/PGA_Champ
   11 : 1999_Majors/US_Open
Nick Price                               :   4 tournaments :    4.75
   17 : 1999_Majors/British_Open
   -3 : 1999_Majors/Masters
   -7 : 1999_Majors/PGA_Champ
   12 : 1999_Majors/US_Open
...
```