Btsort - a sort program using a binary tree

Introduction

Why write a sort program? Well if you need to do a linear search through UTF-8 data which includes ordinary ASCII data, the sort command available at the shell prompt by default sorts into dictionary order based on the current locale. That means that if we are using eg a form of English then the default ordering will be like 'Agnes' 'agriculture' ... 'Bronwyn' 'brown' and so on. Moreover words containing apostrophes are required to collate after similar words without. Now this ordering is perfectly fine for many purposes but when doing a linear search through data sorted in such an order the search will fail because the natural bitwise order of ASCII characters is A..Z then a..z. The situation gets far worse when you need to search through strings of Thai characters, or for that matter Burmese, Lao or Cambodian. In these languages some of the vowel symbols are placed before their attached consonants as well as over, under or after. Compound vowel sounds may be formed from symbols in combination also. It's not so weird; look at 'hat' and 'hate'. In the second word the vowel sound is formed by symbols wrapped around the terminal consonant. In the SE Asian languages the wrapping happens around the initial consonant never the terminal. So I wrote this progam so as to be able to sort bitwise left to right ascending or optionally descending regardless of any locale setting.

Why a binary tree? Binary trees are a fascinating data structure, at least to me. Insertion is of the order of log(n) where n is the number of items being sorted. They work very well on random ordered data but do degenerate to n^2 behaviour if the data arrives already in order or in reverse order. To overcome that problem you can use instead a self balancing tree, such as the AVL tree, or another rather well balanced tree called a red-black tree. This program uses an AVL tree. Google on AVL Tree and follow the links to Wikipedia; this will unearth more than enough information on these particular data structures. A binary tree sort is easily made to produce a stable sort, ie one where equal keys are sorted in order of arrival. Btsort is almost as fast as qsort when it is unfettered from any requirement to produce a dictionary order. Of course you can force qsort to make a stable sort by appending an ASCII formatted record number but once you do that any speed advantage of qsort is well and truly lost.

What the program does.

The program sorts a list of lines eg the dictionary used to solve Jumbles, and sorts it in characterset order. The program reads from a text file and writes to stdout. I use a dictionary called 'mydict' which is derived from some source that used Websters spelling. That is fine for puzzles like Jumble which originates in USA. But I also run a program called 'xword' to cheat on crossword puzzles which are mostly based on Oxford or Macqaurie spelling. Consequently I need to add words to 'mydict' from time to time. I do it like this:

```
user> cp /usr/local/etc/mydict .
user> echo new_word >> mydict
user> btsort mydict > newdict
user> sudo mv newdict /usr/local/etc/mydict
```

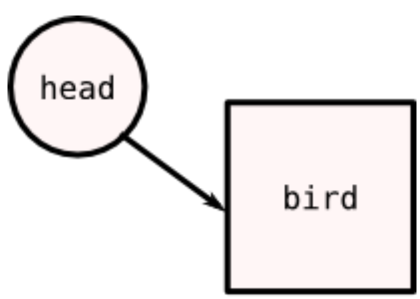
The program has one option, '-d' for a reverse charset ordered sort.

How the program works.

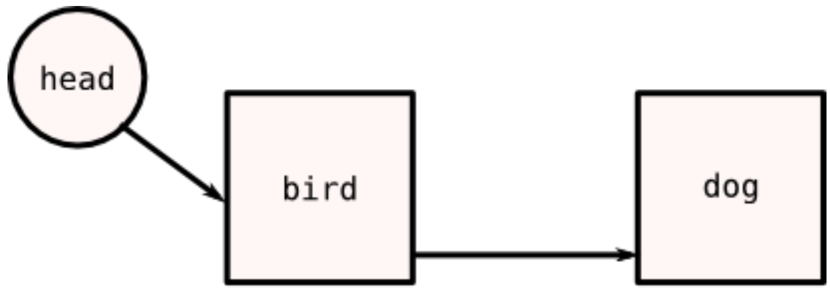
Before getting down to describing binary trees I will start by examining what they are not, the simple linked list. For each node of a linked list we have a structure like this:

```
typedef struct ln {
    char *data;
    char struct ln *next;
} LNODE;
```

When inserting into a linked list, the first node is appended to a head node and then after that any insert goes after the nodes that collate earlier and before those that collate later if any. If there is no bigger item the new node is appended to the list. In diagram form it is like the following:

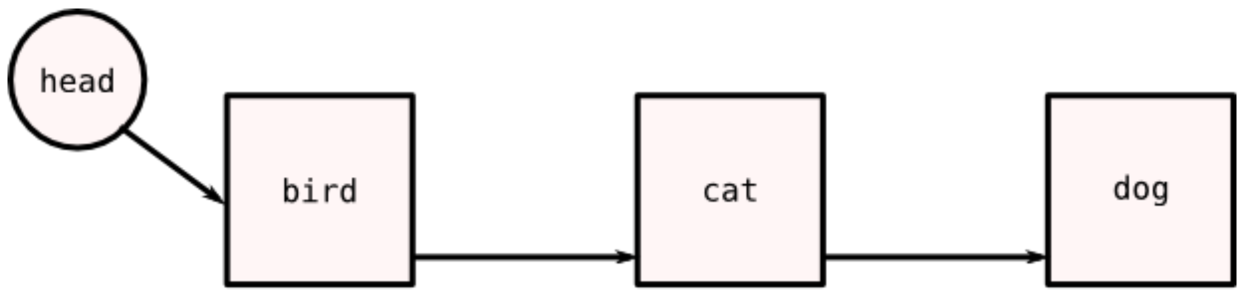


Linked List With One Item



Linked List With Two Items

So when inserting the item "cat", the link from "bird" to "dog" must be broken so that "bird" points to "cat" which in turn points to "dog". The list looks like this:



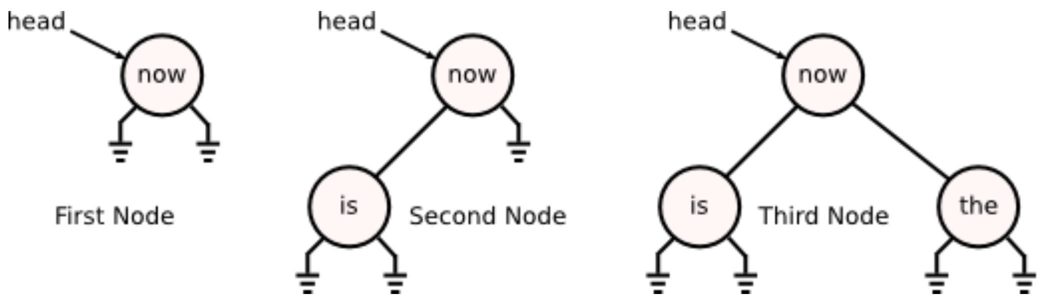
Linked List After Insertion

Well a binary tree is quite unlike the above. First the the data structure:

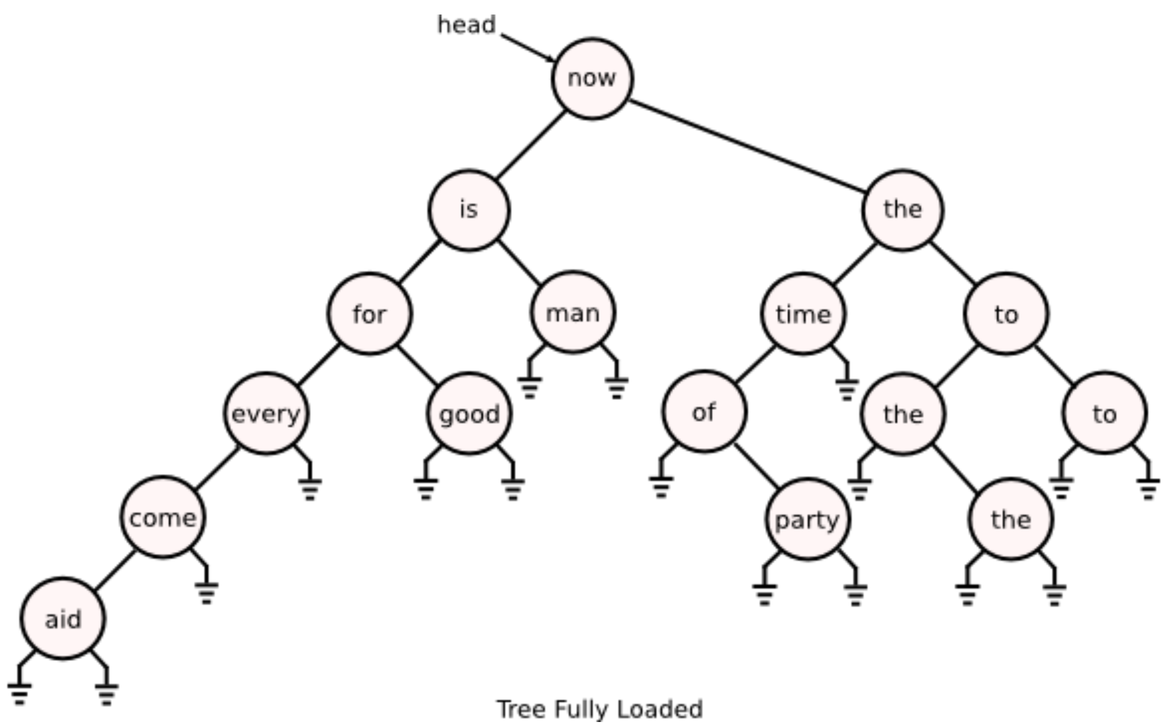
```
typedef struct tn {
        char *key;
        struct tn *left;
        struct tn *right;
}TNODE;
```

The binary tree node has, in the simplest form at least, a minimum of two pointers, one I'll call left, the other right. Now the usual convention is if an item is less than any existing item the program examines the path pointed to by left, otherwise it follows the path to the right. But unlike the linked list above, no linkage is ever broken and reassigned, the program simply follows the sorting rule until it finds an empty path, ie a NULL pointer, and assigns that pointer to the address of the new node. In other words all new insertions are leaf nodes.

Lets examine what happens when we insert that favourite sentence of the tty guys of yore: "now is the time for every good man to come to the aid of the party". The diagram below shows the tree with the first words inserted:



And here is the tree complete:



Since the purpose of the tree in this program to sort data in characterset order the next process is an 'inorder' traversal of the tree.

Inorder traversal

The inorder traversal requires a visit to the leftmost node of the tree and then as we return up the tree.

1. Output this node.
2. Up to the parent node. Output this node.
3. Traverse the right subtree following the same rules as above.

There is also the 'post order' traversal of the tree wherein we travel as far right as posible in a mirror of the inorder traversal and thus obtain a descending order sort.

How it works (The program listing).

```
/* btsort.c - binary tree sort program for lines of chars eg as in a
 * dictionary. Sorting is done using an AVL tree so it's well behaved
 * on pre sorted data. The sort is stable, ie identical elements are
 * output in order of receipt. The program reads from argv[1] or
 * argv[2] and writes to stdout.
*/
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<ctype.h>

// #define DEBUG    1
#ifdef DEBUG
typedef struct tn {
    char *key;
    struct tn *left;
    struct tn *right;
    int balance;
    int number;
}TNODE;
#endif
#ifndef DEBUG
typedef struct tn {
    char *key;
    struct tn *left;
    struct tn *right;
    int balance;
}TNODE;
#endif
TNODE *newnode(char *line);
char *getlin(FILE *fp);    // so named because getline() exists;
            // it returns a '\n' I don't want.
TNODE *tinsert(TNODE *parent, TNODE *node, char *line, int dir);
void tprint(TNODE *node);
void rprint(TNODE *node);
void inprint(TNODE *node);
#ifdef DEBUG
int node_count = 0;
void node_print(TNODE *parent, TNODE *node, char *text);
```

```
#endif
void do_error(char *msg);

enum{Left = -1, Right = 1, None = 0};

int main (int argc, char** argv) {
    TNODE *head = NULL;
    char *line;
    int sortdir = 0;
    FILE *fpi;
    char *infile;
#ifdef DEBUG
    if (system("ls track > /dev/null") == 0)
        system("rm track");
#endif
    if (argc == 3){
        if (strcmp("-d", argv[1]) == 0)
            sortdir = 1;
        else
            do_error("invalid option");
        infile = argv[2];
    } else {
        infile = argv[1];
    }
    fpi = fopen(infile, "r");
    if (!(fpi)) {
        fprintf(stderr, "Failed to open %s\n", infile);
        exit(1);
    }
    while ((line = getlin(fpi))) {
        head = tinsert(NULL, head, line, None);
    }// while()
    if (sortdir == 1)
        rprint(head);
    else
        tprint(head);
    //puts("");
/*    inprint(head);
    puts("");*/
    return 0;
}// main()
TNODE *newnode(char *line){
    char *p;
    TNODE *tmp = (TNODE *)malloc(sizeof(TNODE));
```

```
    if (tmp && (p = strdup(line))) {
        tmp->left = NULL;
        tmp->right = NULL;
        tmp->key = p;
        tmp->balance = 0;
#ifdef DEBUG
        node_count++;
        tmp->number = node_count;
#endif
    } else {
        fprintf(stderr, "Could not get memory\n");
        exit(1);
    }
    return tmp;
}// newnode()
TNODE *tinsert(TNODE *parent, TNODE *node, char *line, int dir){
    int result;
    if (node == NULL) {
        node = newnode(line);
    } else if ((result = strcmp(line, node->key)) >= 0) {
        node->balance++;
        node->right = tinsert(node, node->right, line, Right);
    } else {
        node->balance--;
        node->left = tinsert(node, node->left, line, Left);
    }
    if (node->balance == -2 ){ /* rotate right
        have to make the left child the parent of
        the node we are looking at */
        TNODE *np, *op, *ll; // new parent, old parent, left link
#ifdef DEBUG
        node_print(parent, node, "Before right rotation\n");
#endif
    /* Terminology:
       Old parent, the node we are looking at
       New parent, the left child of the old parent
    What changes:
    1. Left link of old parent to become the right link
       of the new parent.
    2. Right link of new parent -> old parent.
    3. Balance of both new and old becomes 0
    What stays the same:
    1. Left link of new parent remains as is.
    2. Right link of old parent remains as is.
```

```
   */
   /* preserve existing states before we destroy any
      existing linkage
   */
   np    = node->left;
   op    = node;
   ll    = np->right;  // What changes 1.
   np->right   = op;   // What changes 2.
   op->left    = ll;   // What changes 1.
   np->balance = op->balance = 0;  // What changes 3.
       node    = np;   // New parent
#ifdef DEBUG
       node_print(parent, node, "After right rotation\n");
#endif
   }// if (node->bal...
   if (node->balance == 2 ){ /* rotate left
       have to make the right child the parent of
       the node we are looking at  */
       TNODE *np, *op, *rl; // new parent, old parent, right link
#ifdef DEBUG
   node_print(parent, node, "Before left rotation\n");
#endif
   /* Terminology:
      Old parent, the node we are looking at
      New parent, the right child of the old parent
   What changes:
   1. Right link of old parent to become the left link
      of the new parent.
   2. Left link of new parent -> old parent.
   3. Balance of both new and old becomes 0
   What stays the same:
   1. Right link of new parent remains as is.
   2. Left link of old parent remains as is.
   */
   /* preserve existing states before we destroy any
      existing linkage
   */
   np    = node->right;
   op    = node;
   rl    = np->left;   // What changes 1.
   np->left    = op;   // What changes 2.
   op->right    = rl;  // What changes 1.
   np->balance    = op->balance = 0;   // What changes 3.
   node       = np;   // New parent
```

```
#ifdef DEBUG
    node_print(parent, node, "After left rotation\n");
#endif
    }// if (node->bal...
    return node;
}// tinsert()
#define MAX 1000
char *getlin(FILE *fp){
    static char buf[MAX];
    int ch, count;
    count = 0;
    while ((ch = fgetc(fp)) != EOF && (ch != '\n'))
        buf[count++] = ch;

    buf[count] = '\0';
    if (ch == EOF)
        return NULL;
    else
        return buf;
 } // getlin()
void tprint(TNODE *node){
    // pre-order traversal
    if (node->left)
        tprint(node->left);
    printf("%s\n", node->key);
    if (node->right)
        tprint(node->right);
    return;
}// tprint()
void rprint(TNODE *node){
    // post-order traversal
    if (node->right)
        rprint(node->right);
    printf("%s\n", node->key);
    if (node->left)
        rprint(node->left);
    return;
}// rprint()
void inprint(TNODE *node){
    // inorder traversal
    printf("%3d  %s\n", node->balance, node->key);
    if (node->left)
        inprint(node->left);
    if (node->right)
```

```
        inprint(node->right);
    return;
}// inprint()
#ifdef DEBUG
void node_print(TNODE *parent, TNODE *node, char *text) {
    FILE *fp;
    fp = fopen("track", "a");
    fputs(text, fp);
    fprintf(fp, "node count %d\n", node_count);
    if(parent)
        fprintf(fp, "parent->key %s ..->number %d\n", parent->key,
            parent->number);
    fprintf(fp, "node->key %s ..->number %d\n", node->key, node->number);
    if (node->left) {
        fprintf(fp, "node->left->key %s ..->number %d\n",
            node->left->key,
            node->left->number);
        if (node->left->left)
            fprintf(fp, "node->left->left->key %s ..->number %d\n",
                node->left->left->key,
                node->left->left->number);
        if (node->left->right)
            fprintf(fp,"node->left->right->key %s ..->number %d\n",
                node->left->right->key,
                node->left->right->number);
    }// if (node->left)
    if (node->right) {
        fprintf(fp, "node->right->key %s ..->number %d\n",
            node->right->key,
            node->right->number);
        if (node->right->right)
            fprintf(fp,"node->right->right->key %s ..->number %d\n"
                    ,node->right->right->key,
                node->right->right->number);
        if (node->right->left)
            fprintf(fp,"node->right->left->key %s ..->number %d\n",
                node->right->left->key,
                node->right->left->number);
    }// if (node->right)
    fputs("\n", fp);
    fflush(fp);
    fclose(fp);
    return;
```

```
}// node_print()
#endif
void do_error(char *msg) {
    fputs(msg, stderr);
    fputs("\n", stderr);
    exit(1);
} // do_error()
```

Afterwords

Some improvements are strongly needed:

1. It needs a help function so the initial processing should be replaced with standard options processing, maybe allowing the output file to be specified but default to standard out.
2. The sorted file ends up entirely in memory along with the necessary tnode structures which will number slightly more than 50% of the input line count. Each line read is strdup() on read. Possibly I can gain some speed advantage by reading the entire file into memory after opening and then preallocate the space for the tnodes. There would be some waste of memory because I'd allocate one tnode per line.
3. Implementing the above would allow me to have memory allocation failures followed by sorting the file in two halves, four quarters etc and then merge the smaller parts of the file. Whether I'd go that far depends on the how much use this program gets in the wild.
4. That leads me to the next necessity. Put it up somewhere! Sourceforge or Ubuntu One. Suggestions are welcome.