

```

// BtSample.h - BALANCED TREE SAMPLE DEFINITION
//
// MAINTENANCE HISTORY
// DATE          PROGRAMMER AND DETAILS
// 27-09-04      BYG          Original
//
//-----

// GLOBAL DEFINITIONS

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>

#define NAM_LEN      40           // Customer name length
#define N_PURCHASES  10000       // Number of purchases
#define N_CUST       1000        // Number of customers
#define MIN_CUST_NO  100000      // Base customer identifier number
#define MIN_PUR_NO   1000000     // Base purchase number
#define OP_RESET_TIME 0.3        // Reset operation (ms)
#define OP_FIND_TIME  0.7        // Find operation (ms)
#define OP_READ_TIME  0.8        // Read operation (ms)
//-----

// DATA STRUCTURES

typedef struct {
    long    custId;           // Customer identifier
    char    custName[NAM_LEN+1]; // Customer name
} Cust_t;

typedef struct {
    long    purId;           // Purchase customer identifier
    long    purNo;          // Purchase number
    long    purVal;         // Purchase value
} Pur_t;
//-----

// GENERATE CUSTOMER NAME

void
GenName (
    char    *name)           // Name buffer
{
    char    nam[41];        // Name
    int     namLen;        // Name length
    int     i, j;          // General purpose indices

    // Generate last name
    namLen = (int)(4 + rand() % 3);
    for (i=0; i<namLen; i++)
        nam[i] = (char)('A' + rand() % ('Z'-'A'+1));
    nam[i++] = ' ';

    // Generate middle name
    namLen = (int)(4 + rand() % 3);
    for (j=i; j<i+namLen; j++)
        nam[j] = (char)('A' + rand() % ('Z'-'A'+1));
    nam[j++] = ' ';

    // Generate last name
    namLen = (int)(4 + rand() % 3);
    for (i=j; i<j+namLen; i++)
        nam[i] = (char)('A' + rand() % ('Z'-'A'+1));
    nam[i] = '\0';

    // Load the name buffer
    sprintf (name, "%s", nam);
}
//-----

// CUSTOMER COMPARISON FUNCTION

int CustCmp (
    const Cust_t    *cust1,           // Customer 1

```

```

        const Cust_t    *cust2)          // Customer 2
    {
        if (cust1->custId < cust2->custId) return (-1);
        if (cust1->custId > cust2->custId) return (1);
        return (0);
    }
}

//-----
// BALANCED TREE FOR CUSTOMER TABLE

class BtCust_c {
public:
    int      recCnt;                // Record row count
    Cust_t   custTbl[N_CUST];       // Customer table
    int      curInd;                // Current row index
    double   totTime;               // Total theoretical time

    // CONSTRUCTOR

    BtCust_c ()
    {
        memset (custTbl, 0, sizeof(Cust_t)*N_CUST);
        for (recCnt=0; recCnt<N_CUST; recCnt++) {
            custTbl[recCnt].custId = MIN_CUST_NO + recCnt;
            GenName (custTbl[recCnt].custName);
        }
        qsort ((char*)custTbl, recCnt, sizeof(Cust_t),
            (int*)(const void*, const void*)CustCmp);
        BtReset ();
        BtClrCntr ();
    }

    // RESET

    void BtReset ()
    {
        totTime += OP_RESET_TIME;
        curInd = 0;
    }

    // FIND

    bool BtFind (
        const Cust_t    *cust) // Customer record
    {
        int      first, last;      // First and last binary search element
        int      cmpResult;        // Comparison result

        totTime += OP_FIND_TIME;

        first = 0;
        last = recCnt - 1;
        for (;;) {
            cmpResult = CustCmp (&custTbl[curInd], cust);
            if (cmpResult == 0) return 1;
            if (cmpResult < 0) {
                first = curInd + 1;
            } else {
                last = curInd - 1;
            }
            if (first > last) break;
            curInd = (int)((last+first)/2);
        }

        // Realignment

        if (CustCmp (&custTbl[curInd], cust) < 0) curInd++;
        return 0;
    }

    // READ

    bool BtRead (
        Cust_t    *cust) // Customer row
    {
        totTime += OP_READ_TIME;

        if (curInd >= recCnt) return 0;
        cust->custId = custTbl[curInd].custId;
        strcpy (cust->custName, custTbl[curInd].custName);
        curInd++;
    }
}

```

```

        return 1;
    }

    // CLEAR COUNTER
    void BtClrCntr ()
    {
        totTime = 0.0;
    }

    // PRINT CUSTOMER TABLE
    void BtPrint ()
    {
        int i; // General purpose index
        printf ("Customer table: %d records\n", recCnt);
        printf ("=====\n");
        for (i=0; i<recCnt; i++) {
            printf ("%d %s\n",
                custTbl[i].custId, custTbl[i].custName);
        }
    }
};

//-----
// PURCHASE COMPARISON FUNCTION
int PurCmp (
    const Pur_t *pur1, // Purchase 1
    const Pur_t *pur2) // Purchase 2
{
    if (pur1->purId < pur2->purId) return (-1);
    if (pur1->purId > pur2->purId) return (1);
    if (pur1->purNo < pur2->purNo) return (-1);
    if (pur1->purNo > pur2->purNo) return (1);
    return (0);
}

//-----
// BALANCED TREE FOR PURCHASE TABLE
class BtPur_c {
public:
    int recCnt; // Record row count
    Pur_t purTbl[N_PURCHASES]; // Purchase table
    int curInd; // Current row index
    double totTime; // Total theoretical time

    // CONSTRUCTOR
    BtPur_c ()
    {
        printf ("Error: customer table required\n");
        exit (1);
    }

    BtPur_c (
        BtCust_c *btCust) // Customer table
    {
        memset (purTbl, 0, sizeof(Pur_t)*N_PURCHASES);
        for (recCnt=0; recCnt<N_PURCHASES; recCnt++) {
            purTbl[recCnt].purNo = MIN_PUR_NO + recCnt;
            purTbl[recCnt].purId =
                btCust->custTbl[(long) (rand() % N_CUST)].custId;
            purTbl[recCnt].purVal = (long) (rand ());
        }
        qsort ((char*)purTbl, recCnt, sizeof(Pur_t),
            (int (*)(const void*, const void*))PurCmp);
        BtReset ();
        BtClrCntr ();
    }

    // RESET
    void BtReset () {
        totTime += OP_RESET_TIME;
        curInd = 0;
    }
};

```

```

// FIND
bool BtFind (
    const Pur_t      *pur)    // Purchase record
{
    int      first, last;    // First and last binary search element
    int      cmpResult;     // Comparison result

    totTime += OP_FIND_TIME;

    first = 0;
    last = recCnt - 1;
    for (;;) {
        cmpResult = PurCmp (&purTbl[curInd], pur);
        if (cmpResult == 0) return 1;
        if (cmpResult < 0) {
            first = curInd + 1;
        } else {
            last = curInd - 1;
        }
        if (first > last) break;
        curInd = (int)((last+first)/2);
    }

    // Realignment

    if (PurCmp (&purTbl[curInd], pur) < 0) curInd++;
    return 0;
}

// READ
bool BtRead (
    Pur_t      *pur)          // Purchase row
{
    totTime += OP_READ_TIME;

    if (curInd >= recCnt) return 0;
    pur->purId = purTbl[curInd].purId;
    pur->purNo = purTbl[curInd].purNo;
    pur->purVal = purTbl[curInd].purVal;
    curInd++;
    return 1;
}

// CLEAR COUNTER
void BtClrCntr ()
{
    totTime = 0.0;
}

// PRINT PURCHASES TABLE
void BtPrint ()
{
    int      i;              // General purpose index

    printf ("Purchases table: %d records\n", recCnt);
    printf ("=====\n");
    for (i=0; i<recCnt; i++) {
        printf ("%d %d %d\n", purTbl[i].purId,
                purTbl[i].purNo, purTbl[i].purVal);
    }
}
};

```