CSE213L: DSA Lab
LNMIIT, Rupa Ki Nangal
Training Set 09

A code implementing ADT interface for 2-3 trees is provided. The code is large but is not difficult for the best students in a DSA class to understand. To support the reading of the code, ADT implementation has been divided into three files: `tree23Display.c`, `tree23Insert.c` and `tree23Delete.c`. File `tree23.h` lists the interface functions and other declarations.

File tree23.h

```
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>

struct node23 *toRoot;

struct node23 {
      int data[2];
      struct node23 *subT[3];
      int countKeys;
};
typedef struct node23 *node23p;

int search(int val);
void insert(int val);
void rid(int val);
void display();
```

First please familiarise yourself with the display format used for 2-3 trees. This format is like the nesting and indenting we use to indent and print C programs. A brace-pair encloses information about a 2-3 node and its subtrees. The first line within a brace-pair reveals the number of subtrees of the node. The key values and the subtree details are included in the brace-pair and are indented for a clearer reading.

It is suggested that you refresh your classroom learnings related to the 2-3 trees and B-tree before reading this document any further.

A tree constructed from a test run of the program is provided here. Function `main()` code can be found at the end of this document.

Sample display of created 2-3 tree (two columns):

```
{ count of keys = 1
 <12
     { count of keys = 1
      <7
          { count of keys = 2
           <3
               { NULL }
           <5 && >3
               { NULL }
           >5
               { NULL }
          }
      >7
          { count of keys = 2
           <9
               { NULL }
           <10 && >9
               { NULL }
           >10
               { NULL }
                                        }
                                    }
                               >12
                                    { count of keys = 1
                                     <30
                                          { count of keys = 1
                                           <20
                                               { NULL }
                                           >20
                                               { NULL }
                                          }
                                      >30
                                          { count of keys = 1
                                           <37
                                               { NULL }
                                           >37
                                               { NULL }
                                          }
                                    }
                               }
```

Training Set 09: Task 01

As demonstration of your understanding of the code provided to you, augment the program provide a new ADT interface function `int search(int key)`. To demonstrate the function implemented by you, please change function `main()` as follows:

1. Define two functions: int `nextAction()` and int `nextValue()`. Both these functions use standard C function `rand()` to return a random value. Function nextAction() returns int values specified by three macros with probabilities as shown in the parentheses: SEARCH (5%), INSERT (70%), and DELETE (25%). Function `nextValue()` returns a randomly generated integer value between 3001 and 5000.
2. Add an ADT interface function to compute height of the tree. Feel free to include additional member in `struct node23` to track the tree height.
3. The idea is to create a 2-3 tree with about 100 values for demonstration and more importantly as a preparation for the remaining tasks in the training set.

Training Set 09: Task 02

Alter the data structure declarations to make the 2-3 tree, a B-tree up to 7-way branching. This can be done by making necessary changes to the array sizes in `struct node23`.

Modify implementations of B-tree ADT interface functions related to `search()`, `display()` and `insert()`.

Interface function `rid()` could immediately return without any real change in the B-tree.

What is the height of the tree with following number of entries in the B-tree?

1. 100
2. 1000
3. 10000
4. 20000

Training Set 09: Task 03

It should already be obvious to the trainee that this task requires them to complete the implementation of function `rid()`.

File `main.c` defines function `main()`:

```
#include "tree23.h"

int main(void) {
      insert(10); insert(5); insert(7); insert(12);
      insert(3); insert(20); insert(9); insert(30);
      insert(54); insert(36); insert(63);
      insert(72); insert(90); insert(81); insert(37);
      rid(37); rid(36);
      insert(18); insert(27); insert(45);
      insert(39); insert(99); insert(42);
      rid(63);
      insert(37); insert(47);
      rid(99); rid(54); rid(42); rid(27);
      rid(81);
      rid(72); rid(39); rid(45);
      rid(90);
      rid(47); rid(18);
      display();

      return 0;
}
```

The code from files implementing 2-3 tree is placed in the appendices to this training document.

Vishu Malhotra
30 April 2020
The LNMIIT, Rupa ki Nangal
Jaipur, Rajasthan 302031

Appendices

File tree23Display.c

```
#include "tree23.h"

static void display2(node23p node, int offset);
static void printSpaces(int offset);

void display() {
	if (toRoot == NULL) {
		printf("NULL TREE\n");
		return;
	}
	display2(toRoot, 0);
}

static void printSpaces(int offset) {
	int i;
	for (i=0; i<offset; i++)
		printf(" ");
}

void display2(node23p node, int offset) {
	int kys;

	if (node == NULL) {
		printSpaces(offset);
		printf("{ NULL }\n");
		return;
	}

	printSpaces(offset);
	printf("{ count of keys = %d\n", node->countKeys);
	for (kys = 0; kys <= node->countKeys; kys++)
		switch(kys) {
			case 0:
				printSpaces(offset);
				printf(" <%d\n", node->data[0]);
				display2(node->subT[0], offset+5);
				break;
			case 1:
				if (kys < node->countKeys) {
					printSpaces(offset);
					printf(" <%d && >%d\n", node->data[1],
									node->data[0]);
					display2(node->subT[1], offset+5);
					break;
				} else {
					printSpaces(offset);
					printf(" >%d\n", node->data[0]);
					display2(node->subT[1], offset+5);
					break;
			}
			case 2:
				printSpaces(offset);
				printf(" >%d\n", node->data[1]);
				display2(node->subT[2], offset+5);
				break;
		}
	printSpaces(offset);
	printf("}\n");
}
```

File tree23Insert.c

```
#include "tree23.h"

struct node23 *toRoot = NULL;

/* Calls to insert2() and remove2() will use
       these variables to tell their parents if
       the parent node needs to accommodate a new
       key after the call */
struct node23 *newRight = NULL;
int newKey = 0;

static void insert2(node23p node, int data);
static void splitLeaf(node23p node, int data);
static insertInternal(node23p node, int data);

static node23p mkNode(int val) {
       node23p tmp = malloc(sizeof(struct node23));
       assert(tmp != NULL);
       tmp->data[0] = val;
       tmp->data[1] = 0;
       tmp->countKeys = 1;
       tmp->subT[0] = tmp->subT[1] =
              tmp->subT[2] = NULL;
       return tmp;
}

void insert(int data) {
       node23p oldRoot = toRoot;

       if (toRoot == NULL) {
              toRoot = mkNode(data);
              return;
       }
       newRight = NULL;
       insert2(toRoot, data);
       /* check if tree height needs increase */
       if (newRight != NULL) {
              toRoot = mkNode(newKey);
              toRoot->subT[0] = oldRoot;
              toRoot->subT[1] = newRight;
       }
}

static void insert2(node23p node, int data) {
       assert(node != NULL);
       int kys, temp;
       // Primarily to insert in a LEAF
       if (node->subT[0] == NULL) {// This is a leaf
              if (node->data[0] == data)
                     return; // Duplicate data
              if (node->countKeys == 2) { // Full leaf node
                     if (node->data[1] == data)
                            return; // Duplication of data
                     // Tell parent to care for new right sibling.
                     splitLeaf(node, data);
                     return;
              }
              // Add data in this leaf node
              node->data[node->countKeys] = data;
              node->countKeys += 1;
              kys = node->countKeys-1;
              while (kys>0) {
                     if (node->data[kys-1] <= node->data[kys])
                            return; // Node set up
                     // Swap
```

```
                        temp = node->data[kys-1];
                        node->data[kys-1] = node->data[kys];
                        node->data[kys] = temp;
                }
                return;
        }
        // Insertion into an internal node
        assert(node->subT[0] != NULL && node->subT[1] != NULL);
        assert(node->countKeys == 1 || node->subT[2] != NULL);
        /* When inserting data at an internal node there are
                two phases of importance. So we separately code it */
        insertInternal(node, data);
}

static void splitLeaf(node23p node, int data) {
        // Use newRight to tell the parent to update.
        assert(node->countKeys == 2);

        int temp = node->data[1];

        node->countKeys -= 1;
        if (temp < data) {
                newRight = mkNode(data);
                newKey = temp;
                return;
        }

        newRight = mkNode(node->data[1]);
        if (node->data[0] < data) {
                newKey = data;
                return;
        }

        newKey = node->data[0];
        node->data[0] = data;
        return;
}



static insertInternal(node23p node, int data) {
        // Not a leaf node

        int kys;
        int median;
        int who; // Subtree index
        node23p newSibling;

        assert(node != NULL);
        assert(node->subT[0] != NULL);
        newRight = NULL; // Perhaps superflous
        /* If duplicate data do not insert */
        if (node->data[0] == data) return;
        if (node->countKeys == 2 && node->data[1] == data) return;

        if (node->data[0] > data) {
                who = 0;
                insert2(node->subT[0], data);
        } else if (node->countKeys == 2 && node->data[1] < data) {
                who = 2;
                insert2(node->subT[2], data);
        } else {
                who = 1;
                insert2(node->subT[1], data);
        }
        /* On return determine if this node is obliged to accommodate
                a new key? */
        if (newRight == NULL) // All done
```

```
			return;
		if (node->countKeys == 1) {// There is space for key 2
			node->countKeys += 1;
			switch (who) {
				case 0:
					node->subT[2] = node->subT[1];
					node->data[1] = node->data[0];
					node->data[0] = newKey;
					node->subT[1] = newRight;
					newRight = NULL;
					return;
				case 1:
					node->subT[2] = newRight;
					newRight = NULL;
					node->data[1] = newKey;
					return;
			}
		}

		switch (who) {
			case 2: // newKey is largest key
				newSibling = mkNode(newKey);
				newSibling->subT[1] = newRight;
				newSibling->subT[0] = node->subT[2];
				node->subT[2] = NULL;
				node->countKeys -= 1;
				newKey = node->data[1];
				newRight = newSibling;
				return;
			case 1: // Right key in node is largest
				newSibling = mkNode(node->data[1]);
				newSibling->subT[1] = node->subT[2];
				newSibling->subT[0] = newRight;
				node->subT[2] = NULL;
				node->countKeys -= 1;
				newRight = newSibling;
				return;
			case 0: // newKey is smallest
				newSibling = mkNode(node->data[1]);
				newSibling->subT[1] = node->subT[2];
				newSibling->subT[0] = node->subT[1];
				node->subT[2] = NULL;
				median = node->data[0]; // Median key
				node->data[0] = newKey;
				node->subT[1] = newRight;
				node->countKeys -= 1;
				newKey = median;
				newRight = newSibling;
				return;
		}
}
```

File tree23Delete.c

```
#include "tree23.h"

struct node23 *toRoot = NULL;

/* Calls to insert2() and remove2() will use
       these variables to tell their parents if
       the parent node needs to accommodate a new
       key after the call */
struct node23 *newRight = NULL;
int newKey = 0;

static void insert2(node23p node, int data);
static void splitLeaf(node23p node, int data);
static insertInternal(node23p node, int data);

static node23p mkNode(int val) {
       node23p tmp = malloc(sizeof(struct node23));
       assert(tmp != NULL);
       tmp->data[0] = val;
       tmp->data[1] = 0;
       tmp->countKeys = 1;
       tmp->subT[0] = tmp->subT[1] =
              tmp->subT[2] = NULL;
       return tmp;
}

void insert(int data) {
       node23p oldRoot = toRoot;

       if (toRoot == NULL) {
              toRoot = mkNode(data);
              return;
       }
       newRight = NULL;
       insert2(toRoot, data);
       /* check if tree height needs increase */
       if (newRight != NULL) {
              toRoot = mkNode(newKey);
              toRoot->subT[0] = oldRoot;
              toRoot->subT[1] = newRight;
       }
}

static void insert2(node23p node, int data) {
       assert(node != NULL);
       int kys, temp;
       // Primarily to insert in a LEAF
       if (node->subT[0] == NULL) {// This is a leaf
              if (node->data[0] == data)
                     return; // Duplicate data
              if (node->countKeys == 2) { // Full leaf node
                     if (node->data[1] == data)
                            return; // Duplication of data
                     // Tell parent to care for new right sibling.
                     splitLeaf(node, data);
                     return;
              }
              // Add data in this leaf node
              node->data[node->countKeys] = data;
              node->countKeys += 1;
              kys = node->countKeys-1;
              while (kys>0) {
                     if (node->data[kys-1] <= node->data[kys])
                            return; // Node set up
                     // Swap
```

```
                        temp = node->data[kys-1];
                        node->data[kys-1] = node->data[kys];
                        node->data[kys] = temp;
                }
                return;
        }
        // Insertion into an internal node
        assert(node->subT[0] != NULL && node->subT[1] != NULL);
        assert(node->countKeys == 1 || node->subT[2] != NULL);
        /* When inserting data at an internal node there are
                two phases of importance. So we separately code it */
        insertInternal(node, data);
}

static void splitLeaf(node23p node, int data) {
        // Use newRight to tell the parent to update.
        assert(node->countKeys == 2);

        int temp = node->data[1];

        node->countKeys -= 1;
        if (temp < data) {
                newRight = mkNode(data);
                newKey = temp;
                return;
        }

        newRight = mkNode(node->data[1]);
        if (node->data[0] < data) {
                newKey = data;
                return;
        }

        newKey = node->data[0];
        node->data[0] = data;
        return;
}

static insertInternal(node23p node, int data) {
        // Not a leaf node

        int kys;
        int median;
        int who; // Subtree index
        node23p newSibling;

        assert(node != NULL);
        assert(node->subT[0] != NULL);
        newRight = NULL; // Perhaps superflous
        /* If duplicate data do not insert */
        if (node->data[0] == data) return;
        if (node->countKeys == 2 && node->data[1] == data) return;

        if (node->data[0] > data) {
                who = 0;
                insert2(node->subT[0], data);
        } else if (node->countKeys == 2 && node->data[1] < data) {
                who = 2;
                insert2(node->subT[2], data);
        } else {
                who = 1;
                insert2(node->subT[1], data);
        }
        /* On return determine if this node is obliged to accommodate
                a new key? */
        if (newRight == NULL) // All done
                return;
        if (node->countKeys == 1) {// There is space for key 2
```

```
                node->countKeys += 1;
                switch (who) {
                        case 0:
                                node->subT[2] = node->subT[1];
                                node->data[1] = node->data[0];
                                node->data[0] = newKey;
                                node->subT[1] = newRight;
                                newRight = NULL;
                                return;
                        case 1:
                                node->subT[2] = newRight;
                                newRight = NULL;
                                node->data[1] = newKey;
                                return;
                }
        }

        switch (who) {
                case 2: // newKey is largest key
                        newSibling = mkNode(newKey);
                        newSibling->subT[1] = newRight;
                        newSibling->subT[0] = node->subT[2];
                        node->subT[2] = NULL;
                        node->countKeys -= 1;
                        newKey = node->data[1];
                        newRight = newSibling;
                        return;
                case 1: // Right key in node is largest
                        newSibling = mkNode(node->data[1]);
                        newSibling->subT[1] = node->subT[2];
                        newSibling->subT[0] = newRight;
                        node->subT[2] = NULL;
                        node->countKeys -= 1;
                        newRight = newSibling;
                        return;
                case 0: // newKey is smallest
                        newSibling = mkNode(node->data[1]);
                        newSibling->subT[1] = node->subT[2];
                        newSibling->subT[0] = node->subT[1];
                        node->subT[2] = NULL;
                        median = node->data[0]; // Median key
                        node->data[0] = newKey;
                        node->subT[1] = newRight;
                        node->countKeys -= 1;
                        newKey = median;
                        newRight = newSibling;
                        return;
        }
}
```