FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ
SLOVENSKÁ TECHNICKÁ UNIVERZITA
Ilkovičova 2, 842 16 Bratislava 4

2021/2022
Data structures and algorithms
First assignment

Cvičiaci: MSc. Mirwais Ahmadzai
Čas cvičení: Štvrtok 11:00 – 12:50

Vypracoval: Peter Bartoš
AIS ID: 116143

Contents

Binary Search Tree

1. Introduction

BST is a rooted binary tree data structure whose internal nodes each store a key greater than all the keys in the node's left subtree and less than those in its right subtree.
The time complexity of operations on the binary search tree is directly proportional to the height of the tree. The complexity analysis of BST shows that, on average, the insert, delete and search takes $O(\log n)$ for n nodes. In the worst case, they degrade to that of a singly linked list: $O(n)$. (Wikipedia, 2022)

2. Own implementation – AVL

2.1 Insertion

Inserting elements into a binary tree I have done by recursion. At the beginning program checks if the node where the new data is to be inserted is empty. If not, the function gets called again and goes into left or right child depending on the size of the input. If input is bigger than node, then it looks at right child. If input is smaller than node, then it looks at left child. It runs until it reaches an empty node (NULL node), where a new one with the desired value is created. If such data already exists in the tree, it will not change.

```c
void newnodeAVL(NODEAVL **root, int input, char *name)
{
    if (*root == NULL)  {//if root equals to NULL, then the space for new
node is created and is put into the tree
        NODEAVL *newnode = (NODEAVL*) malloc (sizeof(NODEAVL));
        newnode->id = input;
        strncpy(newnode->name, name, 100);
        newnode->height = 0; //every new node is always a leaf, therefore
height is always 0 for newly added node
        newnode->left = NULL;
        newnode->right = NULL;
        *root = newnode;
        return;     }
    if (input > (*root)->id)    //if input is greater than roots
information, then function is called again and is set to look at right
subtree of root
        newnodeAVL(&((*root)->right), input, name);
    else if (input < (*root)->id)   //if input is smaller than roots
information, then function is called again and is set to look at left
subtree of root
        newnodeAVL(&((*root)->left), input, name);
    else
        return;
```

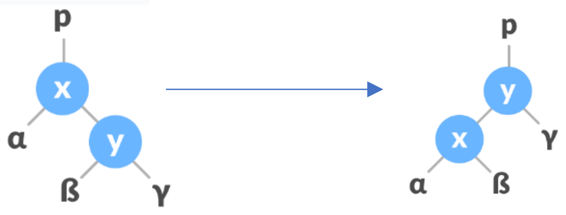
2.2 Balancing

AVL tree is a self-balancing binary search tree in which each node maintains extra information called a balance factor whose value is either -1, 0 or +1.
Balance factor of a node in an AVL tree is the difference between the height of the left subtree and that of the right subtree of that node.

In order to keep this balance factor in range <-1, 1> program needs to perform rotations on nodes to keep the tree balanced. In rotation operation, the positions of the nodes of a subtree are interchanged.

There are two types of rotations:

Left rotation



(Programiz, 2022)

So based on this rotation I integrated it like this:

```
void AVLleftR(NODEAVL **x)  {//variables for 3 nodes x < y < z that the function is supposed to rotate
NODEAVL *y = (*x)->right;          //program needs to perform rotation to left, so it needs to have defined right child of the unbalanced root
NODEAVL *temporary = NULL;
if (y != NULL && y->left != NULL)
      temporary = y->left;    //program also needs to move the left child of y to right child of x, so program saves it to temporary pointer
if (y != NULL)
      y->left = *x;            //program sets x to y's left child
      (*x)->right = temporary;    //and program sets x's right child to
      previous y's left child
temporary = y;  //then program set y as *x so it gets outside of this function
*x = y;
y = temporary;
if ((*x) != NULL)
      (*x)->height = height(*x);  //height of rotated nodes also needs to be updated
 if (y != NULL)
      y->height = height(y);    }
```

Right rotation works the same way, it is just mirrored.

After inserting a node, we need to check if the tree is balanced. 4 cases arise after this happens and they are LL, LR, RR, RL. When it comes to LL case, we only use left rotation once, but in LR program needs to use left rotation and right rotation. Same with the other side when it comes to RR and RL cases. Program needs to know what balance factor has current node after insertion, so after inserting a node when function is returning to node before current node, program calculates the height of the node and uses some form of scales that are going to determine which case it needs to use. Therefore I implemented it like this right after insertion while program is returning from these functions, so it is revisiting previous nodes:

```
int scales(NODEAVL *root)   {//simple function to determine the balance
of the node
      if (root == NULL)
            return 0;
      return (height(root->left) - height(root->right));      }

int height(NODEAVL *node)   //recursive function to get the height of
the node
{
if (node == NULL)   //if the node is empty, then it has zero height and
function knows when to end
      return 0;
else
      return 1 + (max(height(node->right), height(node->left)));
}//otherwise, we add 1 and focus on the node that actually goes
further,
//so we use the max function to know which node goes further


(*root)->height = height(*root);   //program needs to update the height
of the node after insertion
int scale = scales(*root);      //program needs to check for balance in
the subtree and if it is unbalanced, then there are 4 cases
if (scale > 1 && input < (*root)->left->id) {//LL Case
      AVLrightR(&(*root));
      return;      }
if (scale < -1 && input > (*root)->right->id)   {//RR Case
      AVLleftR(&(*root));
      return;      }
if (scale < -1 && input < (*root)->right->id) {//RL Case
      AVLrightR(&((*root)->right));
      AVLleftR(&(*root));
      return;      }
if (scale > 1 && input > (*root)->left->id) {//LR Case
      AVLleftR(&((*root)->left));
      AVLrightR(&(*root));
      return;      }
```

This ensures that every node is balanced in the binary search tree.

2.3 Search

In BST search is very similar to insertion function right until it finds the empty node. Program looks for a value that is the same as the input. If it is bigger than current node then it goes to right child. If it is smaller than current node then it goes to left child. If it finds the node that has same value as input, it gives away some type of information that it found it. If it doesn't find it, then it just simply returns.

```
int searchAVL(NODEAVL *root, int input) {//simple search recursive
function
  if (root == NULL)   //if root is empty, then there is nothing to
search for
      return 0;
  if (input == root->id) {
      printf("AVL.BST -> Found out that id %d belongs to %s\n", root-
      >id, root->name);    //if the input and root-info are equals,
      then the node that was searched for was found
      return 1;}
  else if (input > root->id) {  //if input is bigger than the current
  node
      if (root->left == NULL && root->right == NULL && input != root-
      >id) //if program found an empty node that is not equaled to
      input, it returns
            return 0;
      else if (root->right != NULL)
            searchAVL(root->right, input);}  //function gets called
            again, but it searches through right child of the current
            node
  else if (input < root->id) { //same as elseif before, but it searches
  the left child of the current node
      if (root->left == NULL && root->right == NULL && input != root-
      >id)
            return 0;
  else if (root->left != NULL)
      searchAVL(root->left, input);}        }
```

2.4 Deletion

There are 3 cases in deleting the node. If node has 0 children, 1 child or 2 children. If node has 0 children, then program simply just removes it. If node has 1 child, then program finds that one child, moves information from the child node to parent node(current node) and deletes the child. If node has 2 children program needs to find in order successor or predecessor. I used in order successor, which program finds by finding the smallest node in the right subtree of the current node. Then it copies all the information from successor to root of the subtree and program removes the successor. Then after the deletion we apply the same balancing strategy as in the insertion as the program is returning and revisiting nodes.

```
void deletionAVL(NODEAVL **node, int input) {
  if (*node == NULL) return;
  if (input < (*node)->id)    //through recursion it gets to desired
  node
      deletionAVL(&(*node)->left, input);
  else if (input > (*node)->id)
      deletionAVL(&(*node)->right, input);
  else {   //it got to desired note and now determines if there are no
  children, 1 child or 2 children
      if (*node == NULL) return;
      if ((*node)->left == NULL)  {//if there is 1 child or 0 children
            if ((*node)->right != NULL) { //deletion of node that has 1
            child
                  (*node)->id = (*node)->right->id;
                  strncpy((*node)->name, (*node)->right->name, 100);
                  free((*node)->right);
                  (*node)->right = NULL;}
            else {   //deletion of node that has 0 children
                  free(*node);
                  *node = NULL;}
      return; }
      else if ((*node)->right == NULL) {    //if there is 1 child or 0
      children
            if ((*node)->left != NULL) {  //deletion of node that has
      1child
                  (*node)->id = (*node)->left->id;
                  strncpy((*node)->name, (*node)->left->name, 100);
                  free((*node)->left);
                  (*node)->left = NULL;}
            else {   //deletion of node that has 0 children
                  free(*node);
                  *node = NULL;}
            return;}
    }
    //node has 2 children, so program needs to find inorder successor
  NODEAVL *successor = min_nodeAVL((*node)->right);  //node has 2
children, so program needs to find inorder successor
  (*node)->id = successor->id;    //program overwrites the information
that node is holding to the information that the successor is holding
  strncpy((*node)->name, successor->name, 100);
  deletionAVL(&(*node)->right, successor->id);    //then program
deletes the successor node     }
```

Right after this code program uses 4 balancing cases, calculates balance factor, and uses scales, if there is any balancing needed.

3. Own implementation – REDBLACK

3.1 Introduction

Red-Black tree is a self-balancing binary search tree in which each node contains an extra bit for denoting the colour of the node, either red or black. There is one more colour that represents that balancing is needed and it is double black.

A red-black tree satisfies the following properties:

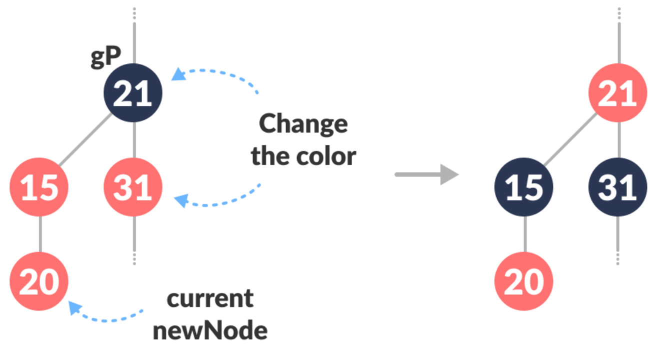
1. *Red/Black Property:* Every node is coloured, either red or black.
2. *Root Property:* The root is black.
3. *Leaf Property:* Every leaf (NIL) is black.
4. *Red Property:* If a red node has children, then the children are always black.
5. *Depth Property:* For each node, any simple path from this node to any of its descendant leaf has the same black depth (the number of black nodes).

The limitations put on the node colours ensure that any simple path from the root to a leaf is not more than twice if any other such path. It helps in maintaining the self-balancing property of the red-black tree. There are also same rotations as in AVL that redblack inherited. Therefore, REDBLACK possesses LL, LR, RR, RL rotations, but in different scenarios with same algorithm. (Programiz, 2022)

3.2 Insertion & balancing

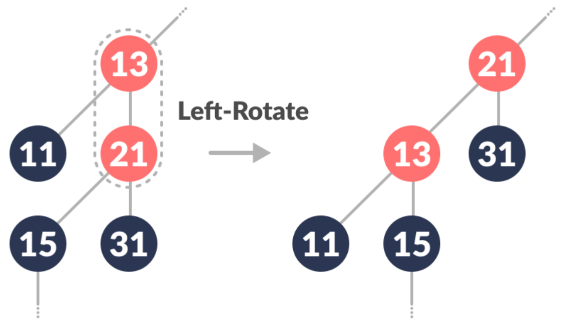
There are 2 parts to insertion. Program can recolour if conditions are met, or rotate.

Case 1: Recoloring - Program simply just recolors, if current node is red, parent of the current node is red and also uncle is red. Grandparent of the current node changes color with uncle and parent of current node is painted to black.



Case 2: Rotations – Program rotates the node and recolors, if uncle is black. Four cases rise as in AVL.

Case 1 is RR rotation, when parent is left child of grandparent and current node is left child of parent. After rotation, program swaps colors between parent and grandparent from before rotation.



Case 2, case 3 and case 4 work the same as this, but rotations are different, as in case 2 it is RL rotation and then recolouring and case 3 and case 4 are mirrored versions of case 1 and case2. (Programiz, 2022)

```
void recrot(NODE **root, NODE *node, int input)   {
   while ((node->id) != input) { //geting to the node that was put in
        if (input > node->id)
            node = node->right;
        if (input < node->id)
            node = node->left;}
   while ( (*root != node) && (node->color != black) && (node->parent->color
== red) )    {
        if ( node->parent != NULL && node->parent->color == red ) {
        //if node has a parent that is red
            NODE *parent = node->parent;
            if (parent->parent != NULL) {
             //if parent has a parent, so node has a grandparent
                NODE *gparent = parent->parent;
                if (parent == gparent->left) {
                   //if grandparent is not a root and parent is the left child
                   of grandparent
                    NODE *uncle = gparent->right;
                    //then gparents right child is an uncle
                    if (uncle != NULL && uncle->color == red) {
                         //if uncle exists, then program recolors in attempt
                         to solve it without rotations
                        uncle->color = black;   //recoloring
                        parent->color = black;  //recoloring
                        gparent->color = red;   //recoloring
                        if (!gparent->parent)
                         //in case of root of tree just to be sure
                            gparent->color = black;
                        if (gparent->color == red &&
                                     gparent->parent->color == red)
                         //in case recoloring leaves 2 red nodes
                            recrot(&(*root), *root, gparent->parent->id); }
                    else {
                        if (parent->right == node) {
                         //if node is right child of its parent, then we need
                         to rotate to the left
                            RBleftR(&(*root), parent);  //left rotation
                            node = parent;
                            parent = node->parent; } //and update the parent
                        RBrightR(&(*root), gparent);    //right rotation
                        colour nodecolor = parent->color;
                        parent->color = gparent->color;
                        gparent->color = nodecolor;
                        node = parent; }
                   //sets pointer on the root of subtree that was just now
                   rotated      }
                else  {  //---same as before, but mirrored---//…
```

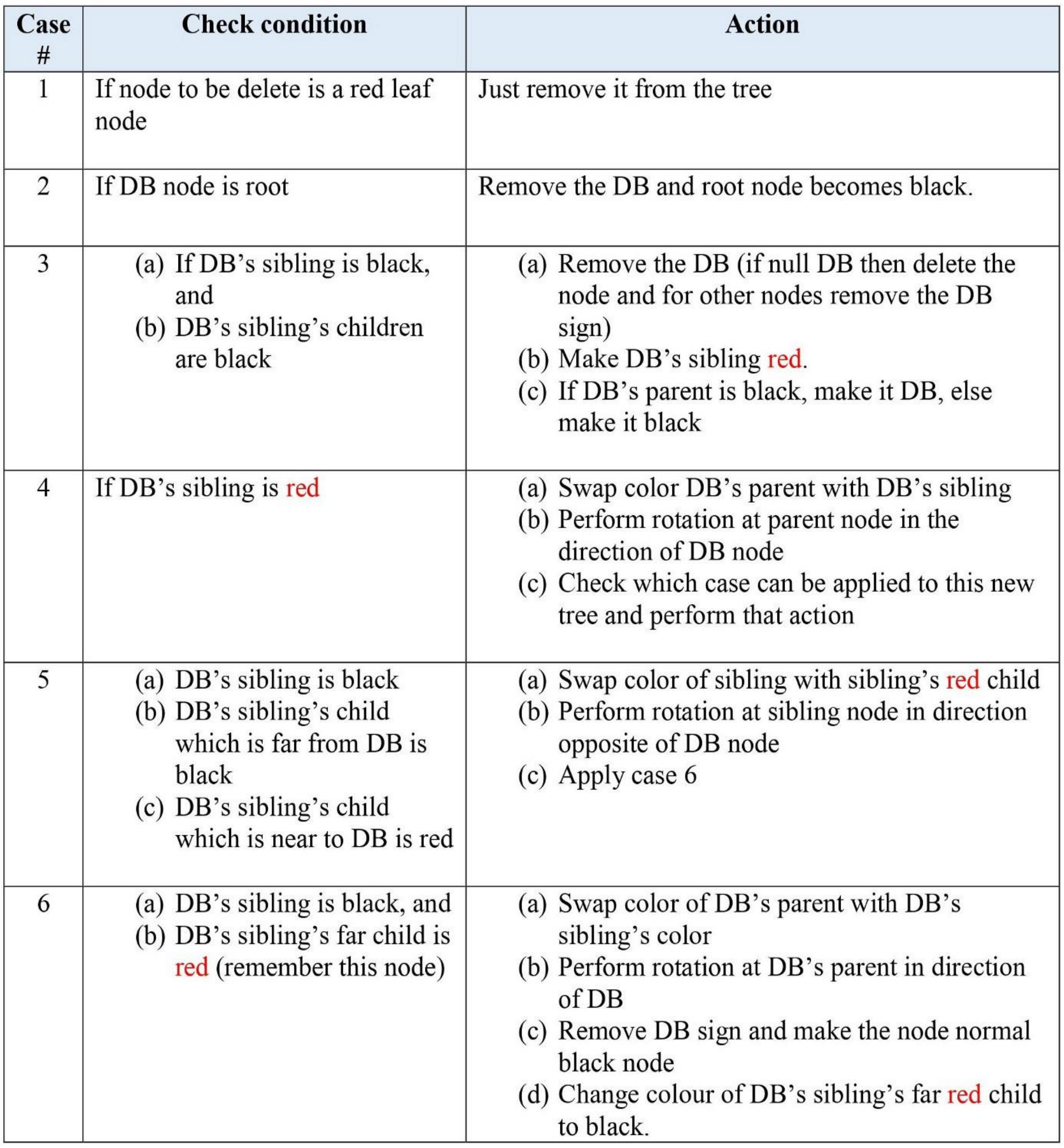
My own implementation of this in one function. I named it recrot, because it recolours and rotates.

3.3 Search

In redblack trees search algorithm stays the same as in standard binary search tree. In my implementation I used the same search method as in AVL.

3.4 Deletion & balancing

For this program uses same algorithms as in AVL, but instead of using balance factors, program again uses cases to keep the integrity of red-black tree. Here are the cases that I used in my implementation (Deletion in RB tree, 2021):



Case #	Check condition	Action
1	If node to be delete is a red leaf node	Just remove it from the tree
2	If DB node is root	Remove the DB and root node becomes black.
3	(a) If DB's sibling is black, and (b) DB's sibling's children are black	(a) Remove the DB (if null DB then delete the node and for other nodes remove the DB sign) (b) Make DB's sibling red. (c) If DB's parent is black, make it DB, else make it black
4	If DB's sibling is red	(a) Swap color DB's parent with DB's sibling (b) Perform rotation at parent node in the direction of DB node (c) Check which case can be applied to this new tree and perform that action
5	(a) DB's sibling is black (b) DB's sibling's child which is far from DB is black (c) DB's sibling's child which is near to DB is red	(a) Swap color of sibling with sibling's red child (b) Perform rotation at sibling node in direction opposite of DB node (c) Apply case 6
6	(a) DB's sibling is black, and (b) DB's sibling's far child is red (remember this node)	(a) Swap color of DB's parent with DB's sibling's color (b) Perform rotation at DB's parent in direction of DB (c) Remove DB sign and make the node normal black node (d) Change colour of DB's sibling's far red child to black.

I integrated these cases to standard BST deletion and after every deletion program paints the node to be deleted double black and runs `void casecheck(NODE **root, NODE *node)` to keep red-black properties in BST. Also I implemented 7th case to make the program run better. It basically checks if node is double black and if it is, then it paints the red parent black, paints the sibling red and removes current node's double black. If parent is black, then it makes him double black, colours sibling red and runs the case 3 of parent to keep the redblack tree integrity. Case 7 looks like this:

```
void case7(NODE *node) { //if node is dblack, then program gives black to
parent
    if (node->color == dblack && node->parent != NULL) {
        if (node->parent->color == red) {
            node->color = black;
            node->parent->color = black;
            if (node->parent->left == node && node->parent->right != NULL) {
                if (node->parent->right->color == black)
                    node->parent->right->color = red; }
            else if (node->parent->right == node && node->parent->left !=
NULL) {
                if (node->parent->left->color == black)
                    node->parent->left->color = red; }
            case3(node->parent);
            case7(node->parent); }
    }
    else if (node->color == dblack && node->parent != NULL) …
```

Hash Table

1. Introduction

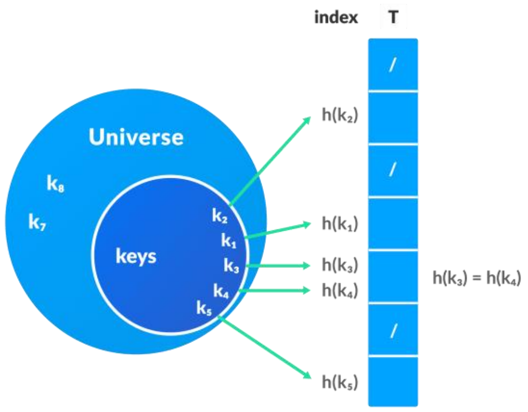
The Hash table data structure stores elements in key-value pairs where

- *Key* - unique integer that is used for indexing the values
- *Value* - data that are associated with keys.

In a hash table, a new index is processed using the keys. And, the element corresponding to that key is stored in the index. This process is called *hashing*.

Let k be a key and h(x) be a hash function.

Here, h(k) will give us a new index to store the element linked with k.



When hashing a key collisions are inevitable to occur. There are many ways to deal with collisions, but I decided to implement hash table that deals with collisions by chaining or open addressing. (Programiz, 2022)

I implemented the hash function like this:

```
unsigned int hashCH(int tablesize, char *name) { //simple hash function
    int length = strnlen(name, 100);    //gets the length of the string
    that stores the name
    unsigned int hint = 0;
    int i = 0;
    for (i = 0; i < length; i++) {
         hint = (hint * 33) + name[i];   //for every character in string
         program adds its integer to hash integer
         hint = (hint * 101 * name[i]) % tablesize; } //then program
         multiplies it by the same character again and divides by % so
         that it gets num in range of indexes
   return hint; }
```

Program calculates the hash number by adding every character's integer to its previous character's integer multiplied by 33. Then its gets multiplied by 101 and the same character and it gets modulo by the size of the table, so It actually fits in the array of elements. I chose 33 and 101 because they are prime numbers and keep it more random.

2. Resizing

I discovered that it is good to have always 50% of all the items in table free so table stays efficient. Therefore, I implemented that hash table always needs to have 50% of its slots free. Program always resizes when it loses the possibility to has it or gains the possibility to downsize and still has it. It counts every item inserted and saves them in stack. Then when it comes to resizing, it copies all those items from stack to new table and allocates the table bigger. If is something deleted from hash table, then it is also deleted from stack. When enough items are deleted, program checks if it can have 50% of slots free after downsizing and if condition is met, then it allocates a new smaller table and puts all the elements from stack into the table.

3. Own implementation – Chaining

3.1 Insertion

In chaining, if a hash function produces the same index for multiple elements, these elements are stored in the same index by using a linked list. (Programiz, 2022) Insertion of an element into a hash table in my version looks like this:

```
unsigned int ihash = hashCH(*tablesize, name);  //calls hash function
to get a number that goes with the name
ITEMCH *data = (ITEMCH*) malloc (sizeof(ITEMCH));   //mallocs memory
for new data
strcpy(data->name, name);   //copies string to new data
data->age = age;
if ((*table)[ihash] != NULL) {
//if collision happens, then it gets chained to the first data
        data->next = (*table)[ihash];
        (*table)[ihash] = data; }
else if ((*table)[ihash] == NULL) {
        data->next = NULL;
        (*table)[ihash] = data; }
return;  }
```

If collision happens, then the whole item in table on hash position is going to be pointed to by the new item and the new item gets set to the hash position in the table. If it doesn't happen, it just gets assigned to the empty slot and its next pointer is set to NULL

3.2 Search

Hash tables make it very efficient when it comes to searching. The hash function calculates an integer that it gets from input that you are trying to find and the same hash integer is going to be used as index in array to find that item. If slot isn't NULL, input and item info is the same, then it returns that item was found. If slot isn't NULL and input and item info aren't the same, then it searches the next node in loop until it doesn't find the item info searched for or ends up on NULL node. My implementation of it looks like this:

```
void searchCH(int tablesize, ITEMCH** table, char *name) {
        int ihash = hashCH(tablesize, name);
        ITEMCH *scan = table[ihash];
        while (scan != NULL && strcmp(scan->name, name) != 0) {
                scan = scan->next; }
        if (scan != NULL && strcmp(scan->name, name) == 0) {
                printf("HM.CH -> Found out that %s is %d years old\n",
                name, scan->age);
                return;  }
        else
                printf("HM.CH -> %s was not found\n", name);
        return;  }
```

3.3 Deletion

Program uses search to locate the item that needs to be deleted. If item isn't chained, then program just deletes it. If item is chained, then program just does standard linked list deletion. 3 cases rise. If item is the head, tail or between two items. It gets deleted and it can check for resizing. This is how it looks in implementation:

```
void deletionCH(int *insertcount, int *tablesize, ITEMCH** *table,
char** *stack, char *name) {
	int ihash = hashCH(*tablesize, name);	//finds the item through
	hash function
	ITEMCH *head = (*table)[ihash];
	ITEMCH *scan = head;
	ITEMCH *prev = NULL;
	while (scan != NULL && strcmp(scan->name, name) != 0)	{
	//if its chained and strings arent the same, so it searches
	through chain
		prev = scan;
		scan = scan->next; }
	if (scan == NULL) {
		return;		//if item was not found }
	else if (prev == NULL && scan != NULL
	&& strcmp(scan->name, name) == 0) { //if item to be deleted is
	the head of the chain
		ITEMCH *head = (*table)[ihash]->next;
		free((*table)[ihash]);
		(*table)[ihash] = NULL;
		(*table)[ihash] = head; }
	else if (prev != NULL && scan != NULL &&
	strcmp(scan->name, name) == 0) //if the item to be deleted is
	between 2 items
		prev->next = scan->next;
```

4. Own implementation – Open Addressing

4.1 Insertion

Open addressing works not by adding a linked list to an item, but rather searching for an empty slot after finding a collision. What happens is that hash function calculates index for an array, program checks if the slot is taken and if slot really is taken, then it looks at the next slot. If the next slot is taken, then it looks at the slot after the next slot and so on. This can also create clusters and make hash table inefficient. My implementation of insertion looks like this:

```
void insert(int *insertcount, int *tablesize, ITEM** *table, char**
*stack, char* name, int age) {
	unsigned int ihash = hash(*tablesize, name);	//calls hash
	function to get a number that goes with the name
	ITEM *item = (ITEM*) malloc (sizeof(ITEM)); //mallocs memory for
	new data
	strcpy(item->name, name);	//copies string to new data
	item->age = age;
	int i;
	for (i = 0; i < *tablesize; i++)  {  //if collission happens,
	then it finds next empty slot
		int num = ihash + i;
		if (num > *tablesize)
			num = num % (*tablesize);
		if ((*table)[num] == NULL) {
			(*table)[num] = item;
		return; }
	}
	return; }
```

4.2 Search

The problem happens in searching. Hash function calculates integer for input and matches it with something that is in hash table, but the input and item info might not match, because of open addressing's insertion. If they don't match, program needs to try next slot in the hash table in loop. If they finally match, loop break and it returns the item that was searched for. This is rather inefficient, because it could need to check a huge amount of slots for that item. This is how I tackled with such a problem:

```
void search(int tablesize, ITEM** table, char *name) {
	int i, ihash = hash(tablesize, name);
	for (i = 0; i < tablesize; i++) { //if collission happens, then
	it finds the item through iteration
	if (table[(ihash + i) % tablesize] != NULL
	&& strcmp(table[(ihash + i) % tablesize]->name, name) == 0) {
       printf("HM.OA -> Found out that %s is %d years old\n",
	name, table[(ihash + i) % tablesize]->age);
```

```
            return; }
        }
        printf("HM.OA -> %s was not found\n", name);
        return; }
```

4.3 Deletion

The same thing happens in deletion as happened in search. If collision happens, program needs to go through maybe a cluster of slots that has been already assigned to that hashed index. So it goes through that cluster again, finds an item to delete and simply deletes it. If it doesn't find the item, then it just returns. If collision doesn't happen, then it simply deletes the item. My implementation of it looks like this:

```
void deletion(int *insertcount, int *tablesize, ITEM** *table, char**
*stack, char* name) {
        int j = -1, i, ihash = hash(*tablesize, name);  //finds the item
        through hash function
        for (i = 0; i < *tablesize; i++)  {  //if collission happens,
        then it finds the item through iteration
                if ((*table)[(ihash + i) % (*tablesize)] != NULL
                && strcmp((*table)[(ihash + i) % (*tablesize)]->name,
                 name) == 0) {
                        j = 1;
                        (*table)[(ihash + i) % (*tablesize)] = NULL; }
        }
        if (j == -1)  {  //if it item was not found
                //printf("|%s not found|", name);
                return; }
}
```

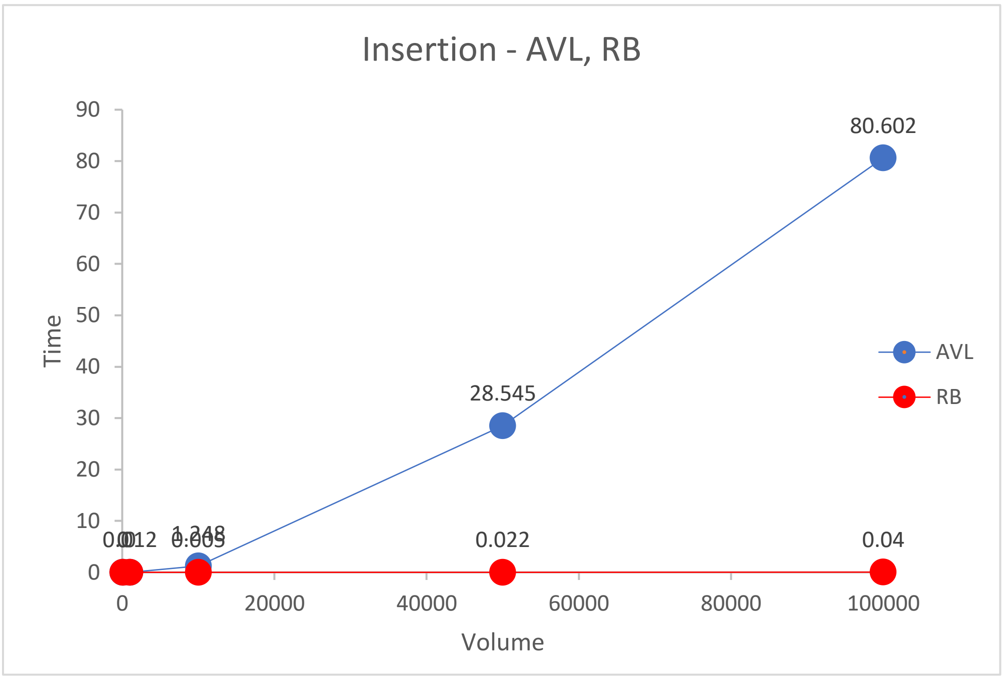
Testing

1. Binary search trees

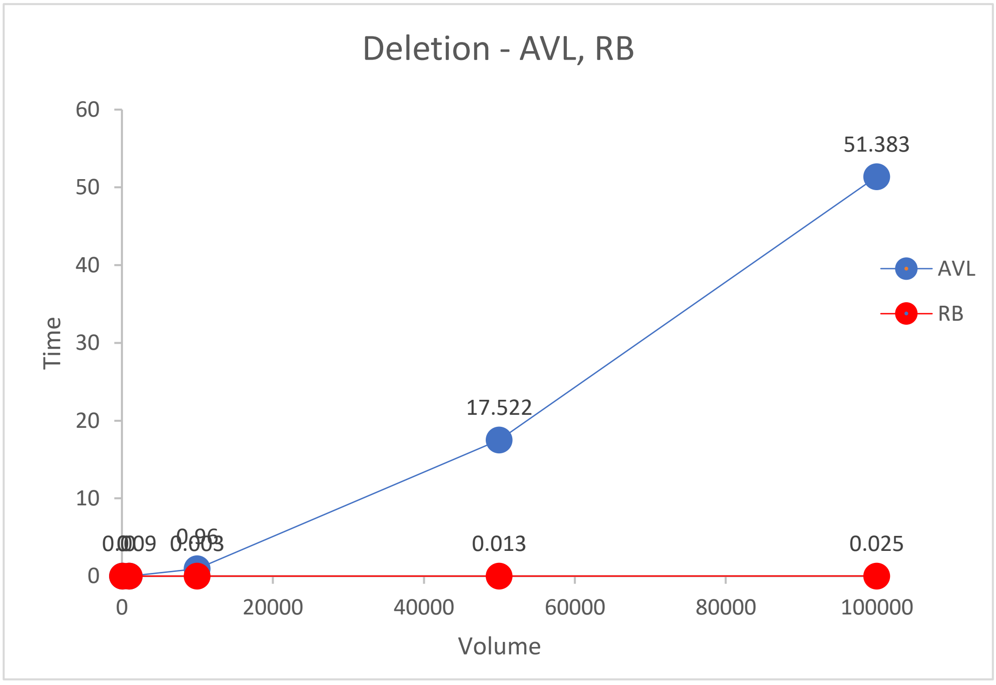
Both algorithms have measured time of inserting, deleting and searching items. Items are randomly generated and BSTs are precisely measured thanks to time.h library. After inserting some items, deletion is next and after deleting items, searching is done.

Results:

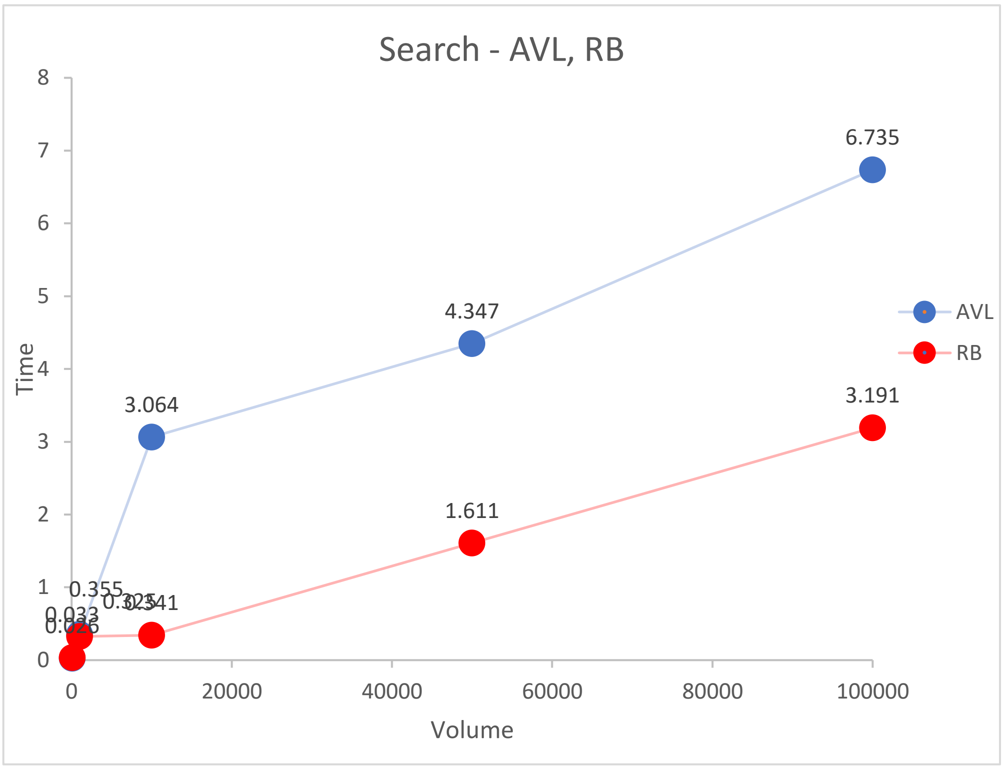
Insert	Volume	Time
AVL	100	0
AVL	1000	0.012
AVL	10000	1.248
AVL	50000	28.545
AVL	100000	80.602
RB	100	0
RB	1000	0
RB	10000	0.005
RB	50000	0.022
RB	100000	0.04



Delete	Volume	Time
AVL	100	0
AVL	1000	0.009
AVL	10000	0.96
AVL	50000	17.522
AVL	100000	51.383
RB	100	0
RB	1000	0
RB	10000	0.003
RB	50000	0.013
RB	100000	0.025



Search	Volume	Time
AVL	100	0.033
AVL	1000	0.355
AVL	10000	3.064
AVL	50000	4.347
AVL	100000	6.735
RB	100	0.026
RB	1000	0.325
RB	10000	0.341
RB	50000	1.611
RB	100000	3.191

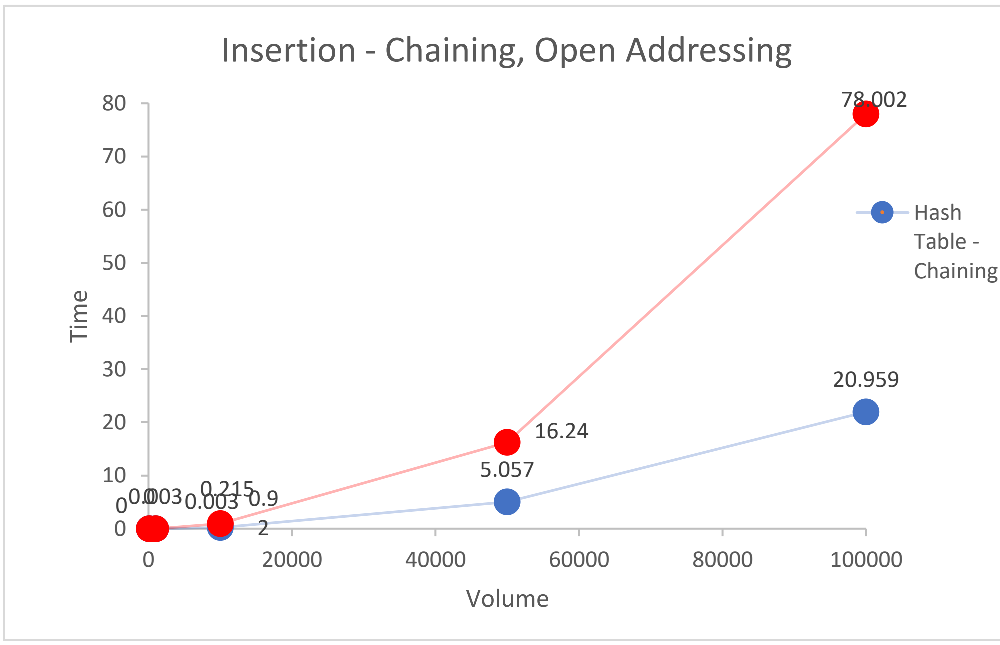


Red-black are tremendously more efficient than AVL trees. That makes sense, because red-black trees are more optimized and don't require that many operations and sometimes just a simple recolouring helps instead of rotations.

2. Hash tables

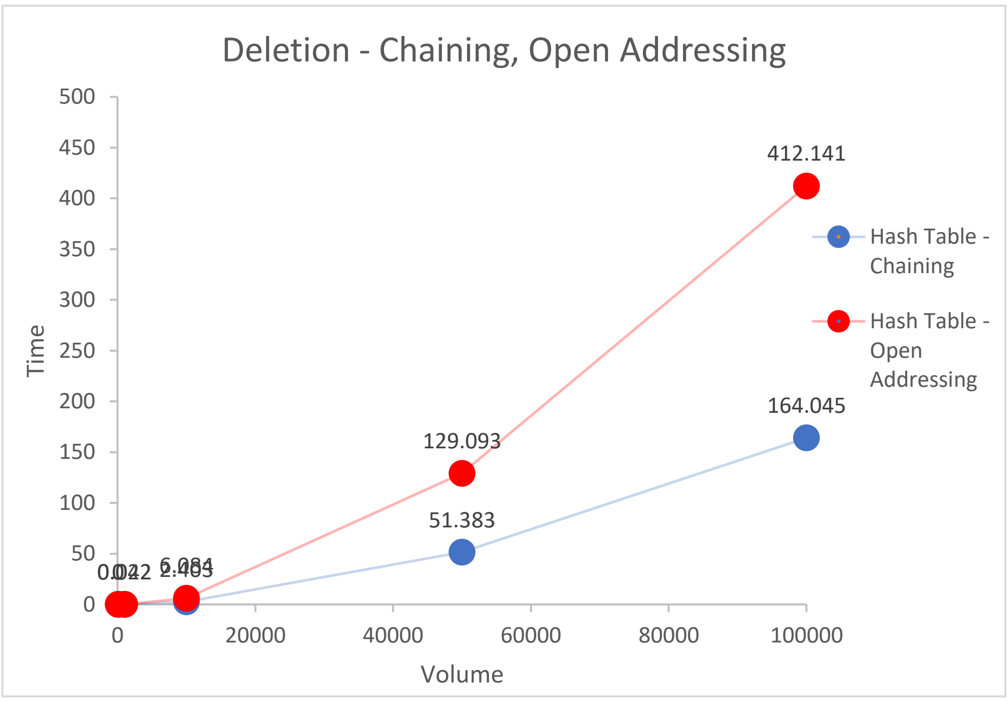
Both algorithms have measured time of inserting, deleting and searching items. Items are randomly generated and hash tables are precisely measured thanks to time.h library. After inserting some items, deletion is next and after deleting items, searching is done. *Results:*

Insert	Volume	Time
CH	100	0
CH	1000	0.003
CH	10000	0.215
CH	50000	5.057
CH	100000	20.959
OA	100	0
OA	1000	0.003
OA	10000	0.92
OA	50000	16.24
OA	100000	78.002

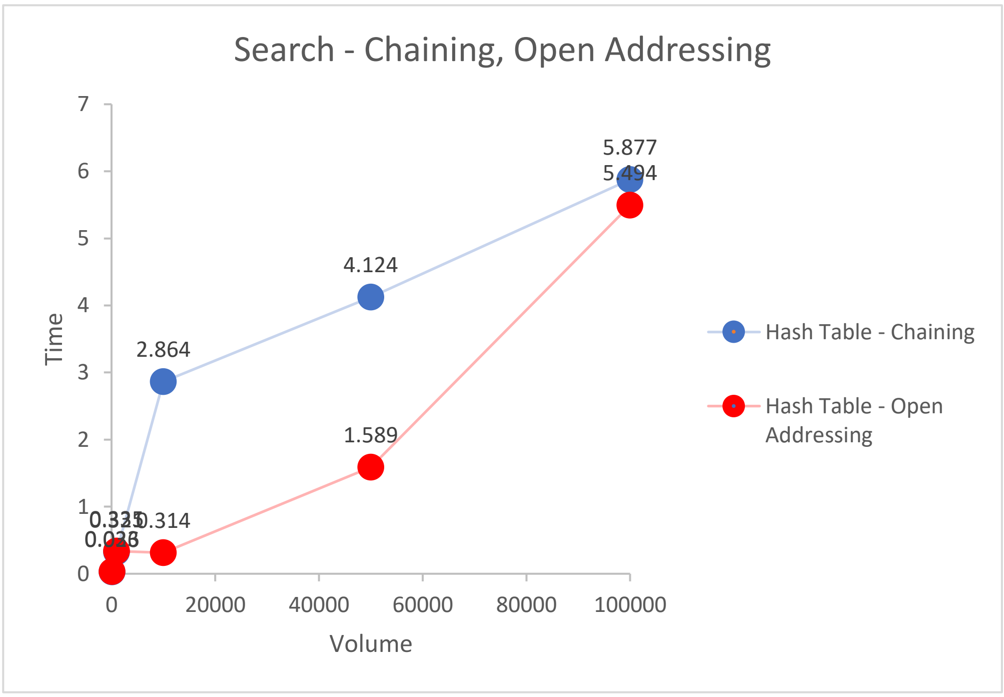


Peter Bartoš AIS ID: 116143

Delete	Volume	Time
CH	100	0
CH	1000	0.022
CH	10000	2.403
CH	50000	51.383
CH	100000	164.045
OA	100	0
OA	1000	0.042
OA	10000	6.084
OA	50000	129.093
OA	100000	412.141



Search	Volume	Time
CH	100	0.026
CH	1000	0.321
CH	10000	2.864
CH	50000	4.124
CH	100000	5.877
OA	100	0.033
OA	1000	0.335
OA	10000	0.314
OA	50000	1.589
OA	100000	5.494



Open-addressing is way less efficient than chaining. That makes sense, because in huge tables open addressing in hash table can become a disadvantage. Program needs to find an empty space and may have created a cluster, which takes time to get through. Somehow search is slower in chaining, which is bizarre to me. Chaining is more efficient because of its collision rules. It doesn’t create clusters and can become even more efficient, if instead of linked list is used BST.