**ADS Experiment List Practical Exam**

**//https://pastebin.pl/view/908e6ebd**

1. Implementation of Multi-Pop Stack and perform amortized analysis on it.
2. Implementation of Dynamic Table and perform amortized analysis on it.
3. Implementation of Binary Counter and perform amortized analysis on it.
4. Implementation of standard tries and perform insert operation, search and delete operation.
5. Implement the code to merge two binomial Heap.
6. Write down the code to insert an element in Red Black Tree.
7. Write down the code to insert an element in B Tree.
8. Write down the code to delete an element in B Tree.
9. Write down the code to find overlapping interval in interval tree.
10. Write down the code to find sum of given interval in Segment Tree.
11. Implementation of Count-Min Sketch using two hash functions.
12. Implementation of Bloom Filter Sketch using two hash functions

Implementation of RB Tree.

#include<iostream>

usingnamespacestd;

structnode

{

    intkey;

    node \*parent;

    charcolor;

    node \*left;

    node \*right;

};

classRBtree

{

    node \*root;

    node \*q;

public:

    RBtree()

    {

        q = NULL;

        root = NULL;

    }

    voidinsert();

    voidinsertfix(node\*);

    voidleftrotate(node\*);

    voidrightrotate(node\*);

    voiddel();

    node\*successor(node\*);

    voiddelfix(node\*);

    voiddisp();

    voiddisplay(node\*);

    voidsearch();

};

voidRBtree::insert()

{

    intz, i = 0;

    cout <<"\nEnter key of the node to be inserted: ";

    cin >>z;

    node \*p, \*q;

    node \*t = newnode;

    t->key = z;

    t->left = NULL;

    t->right = NULL;

    t->color = 'r';

    p = root;

    q = NULL;

    if (root == NULL)

    {

        root = t;

        t->parent = NULL;

    }

    else

    {

        while (p != NULL)

        {

            q = p;

            if (p->key<t->key)

                p = p->right;

            else

                p = p->left;

        }

        t->parent = q;

        if (q->key<t->key)

            q->right = t;

        else

            q->left = t;

    }

    insertfix(t);

}

voidRBtree::insertfix(node\*t)

{

    node \*u;

    if (root == t)

    {

        t->color = 'b';

        return;

    }

    while (t->parent != NULL&&t->parent->color == 'r')

    {

        node \*g = t->parent->parent;

        if (g->left == t->parent)

        {

            if (g->right != NULL)

            {

                u = g->right;

                if (u->color == 'r')

                {

                    t->parent->color = 'b';

                    u->color = 'b';

                    g->color = 'r';

                    t = g;

                }

            }

            else

            {

                if (t->parent->right == t)

                {

                    t = t->parent;

                    leftrotate(t);

                }

                t->parent->color = 'b';

                g->color = 'r';

                rightrotate(g);

            }

        }

        else

        {

            if (g->left != NULL)

            {

                u = g->left;

                if (u->color == 'r')

                {

                    t->parent->color = 'b';

                    u->color = 'b';

                    g->color = 'r';

                    t = g;

                }

            }

            else

            {

                if (t->parent->left == t)

                {

                    t = t->parent;

                    rightrotate(t);

                }

                t->parent->color = 'b';

                g->color = 'r';

                leftrotate(g);

            }

        }

        root->color = 'b';

    }

}

voidRBtree::del()

{

    if (root == NULL)

    {

        cout <<"\nEmpty Tree.";

        return;

    }

    intx;

    cout <<"\nEnter the key of the node to be deleted: ";

    cin >>x;

    node \*p;

    p = root;

    node \*y = NULL;

    node \*q = NULL;

    intfound = 0;

    while (p != NULL&&found == 0)

    {

        if (p->key == x)

            found = 1;

        if (found == 0)

        {

            if (p->key<x)

                p = p->right;

            else

                p = p->left;

        }

    }

    if (found == 0)

    {

        cout <<"\nElement Not Found.";

        return;

    }

    else

    {

        cout <<"\nDeleted Element: "<<p->key;

        cout <<"\nColour: ";

        if (p->color == 'b')

            cout <<"Black\n";

        else

            cout <<"Red\n";

        if (p->parent != NULL)

            cout <<"\nParent: "<<p->parent->key;

        else

            cout <<"\nThere is no parent of the node. ";

        if (p->right != NULL)

            cout <<"\nRight Child: "<<p->right->key;

        else

            cout <<"\nThere is no right child of the node. ";

        if (p->left != NULL)

            cout <<"\nLeft Child: "<<p->left->key;

        else

            cout <<"\nThere is no left child of the node.  ";

        cout <<"\nNode Deleted.";

        if (p->left == NULL || p->right == NULL)

            y = p;

        else

            y = successor(p);

        if (y->left != NULL)

            q = y->left;

        else

        {

            if (y->right != NULL)

                q = y->right;

            else

                q = NULL;

        }

        if (q != NULL)

            q->parent = y->parent;

        if (y->parent == NULL)

            root = q;

        else

        {

            if (y == y->parent->left)

                y->parent->left = q;

            else

                y->parent->right = q;

        }

        if (y != p)

        {

            p->color = y->color;

            p->key = y->key;

        }

        if (y->color == 'b')

            delfix(q);

    }

}

voidRBtree::delfix(node\*p)

{

    node \*s;

    while (p != root&&p->color == 'b')

    {

        if (p->parent->left == p)

        {

            s = p->parent->right;

            if (s->color == 'r')

            {

                s->color = 'b';

                p->parent->color = 'r';

                leftrotate(p->parent);

                s = p->parent->right;

            }

            if (s->right->color == 'b'&&s->left->color == 'b')

            {

                s->color = 'r';

                p = p->parent;

            }

            else

            {

                if (s->right->color == 'b')

                {

                    s->left->color == 'b';

                    s->color = 'r';

                    rightrotate(s);

                    s = p->parent->right;

                }

                s->color = p->parent->color;

                p->parent->color = 'b';

                s->right->color = 'b';

                leftrotate(p->parent);

                p = root;

            }

        }

        else

        {

            s = p->parent->left;

            if (s->color == 'r')

            {

                s->color = 'b';

                p->parent->color = 'r';

                rightrotate(p->parent);

                s = p->parent->left;

            }

            if (s->left->color == 'b'&&s->right->color == 'b')

            {

                s->color = 'r';

                p = p->parent;

            }

            else

            {

                if (s->left->color == 'b')

                {

                    s->right->color = 'b';

                    s->color = 'r';

                    leftrotate(s);

                    s = p->parent->left;

                }

                s->color = p->parent->color;

                p->parent->color = 'b';

                s->left->color = 'b';

                rightrotate(p->parent);

                p = root;

            }

        }

        p->color = 'b';

        root->color = 'b';

    }

}

voidRBtree::leftrotate(node\*p)

{

    if (p->right == NULL)

        return;

    else

    {

        node \*y = p->right;

        if (y->left != NULL)

        {

            p->right = y->left;

            y->left->parent = p;

        }

        else

            p->right = NULL;

        if (p->parent != NULL)

            y->parent = p->parent;

        if (p->parent == NULL)

            root = y;

        else

        {

            if (p == p->parent->left)

                p->parent->left = y;

            else

                p->parent->right = y;

        }

        y->left = p;

        p->parent = y;

    }

}

voidRBtree::rightrotate(node\*p)

{

    if (p->left == NULL)

        return;

    else

    {

        node \*y = p->left;

        if (y->right != NULL)

        {

            p->left = y->right;

            y->right->parent = p;

        }

        else

            p->left = NULL;

        if (p->parent != NULL)

            y->parent = p->parent;

        if (p->parent == NULL)

            root = y;

        else

        {

            if (p == p->parent->left)

                p->parent->left = y;

            else

                p->parent->right = y;

        }

        y->right = p;

        p->parent = y;

    }

}

node\*RBtree::successor(node\*p)

{

    node \*y = NULL;

    if (p->left != NULL)

    {

        y = p->left;

        while (y->right != NULL)

            y = y->right;

    }

    else

    {

        y = p->right;

        while (y->left != NULL)

            y = y->left;

    }

    returny;

}

voidRBtree::disp()

{

    display(root);

}

voidRBtree::display(node\*p)

{

    if (root == NULL)

    {

        cout <<"\nEmpty Tree.";

        return;

    }

    if (p != NULL)

    {

        cout <<"\n\t NODE: ";

        cout <<"\n Key: "<<p->key;

        cout <<"\n Colour: ";

        if (p->color == 'b')

            cout <<"Black";

        else

            cout <<"Red";

        if (p->parent != NULL)

            cout <<"\n Parent: "<<p->parent->key;

        else

            cout <<"\n There is no parent of the node. ";

        if (p->right != NULL)

            cout <<"\n Right Child: "<<p->right->key;

        else

            cout <<"\n There is no right child of the node. ";

        if (p->left != NULL)

            cout <<"\n Left Child: "<<p->left->key;

        else

            cout <<"\n There is no left child of the node.  ";

        cout << endl;

        if (p->left)

        {

            cout <<"\n\nLeft:\n";

            display(p->left);

        }

        /\*else

         cout<<"\nNo Left Child.\n";\*/

        if (p->right)

        {

            cout <<"\n\nRight:\n";

            display(p->right);

        }

        /\*else

         cout<<"\nNo Right Child.\n"\*/

    }

}

voidRBtree::search()

{

    if (root == NULL)

    {

        cout <<"\nEmpty Tree\n";

        return;

    }

    intx;

    cout <<"\n Enter key of the node to be searched: ";

    cin >>x;

    node \*p = root;

    intfound = 0;

    while (p != NULL&&found == 0)

    {

        if (p->key == x)

            found = 1;

        if (found == 0)

        {

            if (p->key<x)

                p = p->right;

            else

                p = p->left;

        }

    }

    if (found == 0)

        cout <<"\nElement Not Found.";

    else

    {

        cout <<"\n\t FOUND NODE: ";

        cout <<"\n Key: "<<p->key;

        cout <<"\n Colour: ";

        if (p->color == 'b')

            cout <<"Black";

        else

            cout <<"Red";

        if (p->parent != NULL)

            cout <<"\n Parent: "<<p->parent->key;

        else

            cout <<"\n There is no parent of the node. ";

        if (p->right != NULL)

            cout <<"\n Right Child: "<<p->right->key;

        else

            cout <<"\n There is no right child of the node. ";

        if (p->left != NULL)

            cout <<"\n Left Child: "<<p->left->key;

        else

            cout <<"\n There is no left child of the node.  ";

        cout << endl;

    }

}

intmain()

{

    intch, y = 0;

    RBtreeobj;

    do

    {

        cout <<"\n\t RED BLACK TREE ";

        cout <<"\n 1. Insert in the tree ";

        cout <<"\n 2. Delete a node from the tree";

        cout <<"\n 3. Search for an element in the tree";

        cout <<"\n 4. Display the tree ";

        cout <<"\n 5. Exit ";

        cout <<"\nEnter Your Choice: ";

        cin >>ch;

        switch (ch)

        {

        case1:

            obj.insert();

            cout <<"\nNode Inserted.\n";

            break;

        case2:

            obj.del();

            break;

        case3:

            obj.search();

            break;

        case4:

            obj.disp();

            break;

        case5:

            y = 1;

            break;

        default:

            cout <<"\nEnter a Valid Choice.";

        }

        cout << endl;

    } while (y != 1);

    return1;

}

**Implementation of Binomial Heap.**

#include<iostream>

#include<cstdlib>

usingnamespacestd;

// Node Declaration

structnode

{

    intn;

    intdegree;

    node \*parent;

    node \*child;

    node \*sibling;

};

// Class Declaration

classBinomialHeap

{

private:

    node \*H;

    node \*Hr;

    intcount;

public:

    node\*Initializeheap();

    intBinomial\_link(node\*, node\*);

    node\*Create\_node(int);

    node\*Union(node\*, node\*);

    node\*Insert(node\*, node\*);

    node\*Merge(node\*, node\*);

    node\*Extract\_Min(node\*);

    intRevert\_list(node\*);

    intDisplay(node\*);

    node\*Search(node\*, int);

    intDecrease\_key(node\*, int, int);

    intDelete(node\*, int);

    BinomialHeap()

    {

        H = Initializeheap();

        Hr = Initializeheap();

        intcount = 1;

    }

};

// Initialize Heap

node\*BinomialHeap::Initializeheap()

{

    node \*np;

    np = NULL;

    returnnp;

}

// Linking nodes in Binomial Heap

intBinomialHeap::Binomial\_link(node\*y, node\*z)

{

    y->parent = z;

    y->sibling = z->child;

    z->child = y;

    z->degree = z->degree + 1;

}

// Create Nodes in Binomial Heap

node\*BinomialHeap::Create\_node(intk)

{

    node \*p = newnode;

    p->n = k;

    returnp;

}

// Insert Nodes in Binomial Heap

node\*BinomialHeap::Insert(node\*H, node\*x)

{

    node \*H1 = Initializeheap();

    x->parent = NULL;

    x->child = NULL;

    x->sibling = NULL;

    x->degree = 0;

    H1 = x;

    H = Union(H, H1);

    returnH;

}

// Union Nodes in Binomial Heap

node\*BinomialHeap::Union(node\*H1, node\*H2)

{

    node \*H = Initializeheap();

    H = Merge(H1, H2);

    if (H == NULL)

        returnH;

    node \*prev\_x;

    node \*next\_x;

    node \*x;

    prev\_x = NULL;

    x = H;

    next\_x = x->sibling;

    while (next\_x != NULL)

    {

        if ((x->degree != next\_x->degree) || ((next\_x->sibling != NULL) && (next\_x->sibling)->degree == x->degree))

        {

            prev\_x = x;

            x = next\_x;

        }

        else

        {

            if (x->n<= next\_x->n)

            {

                x->sibling = next\_x->sibling;

                Binomial\_link(next\_x, x);

            }

            else

            {

                if (prev\_x == NULL)

                    H = next\_x;

                else

                    prev\_x->sibling = next\_x;

                Binomial\_link(x, next\_x);

                x = next\_x;

            }

        }

        next\_x = x->sibling;

    }

    returnH;

}

// Merge Nodes in Binomial Heap

node\*BinomialHeap::Merge(node\*H1, node\*H2)

{

    node \*H = Initializeheap();

    node \*y;

    node \*z;

    node \*a;

    node \*b;

    y = H1;

    z = H2;

    if (y != NULL)

    {

        if (z != NULL)

        {

            if (y->degree<= z->degree)

                H = y;

            elseif (y->degree>z->degree)

                H = z;

        }

        else

            H = y;

    }

    else

        H = z;

    while (y != NULL&&z != NULL)

    {

        if (y->degree<z->degree)

        {

            y = y->sibling;

        }

        elseif (y->degree == z->degree)

        {

            a = y->sibling;

            y->sibling = z;

            y = a;

        }

        else

        {

            b = z->sibling;

            z->sibling = y;

            z = b;

        }

    }

    returnH;

}

// Display Binomial Heap

intBinomialHeap::Display(node\*H)

{

    if (H == NULL)

    {

        cout <<"The Heap is empty"<< endl;

        return0;

    }

    cout <<"The root nodes are: "<< endl;

    node \*p;

    p = H;

    while (p != NULL)

    {

        cout <<p->n;

        if (p->sibling != NULL)

            cout <<"-->";

        p = p->sibling;

    }

    cout << endl;

}

// Extract Minimum

node\*BinomialHeap::Extract\_Min(node\*H1)

{

    Hr = NULL;

    node \*t = NULL;

    node \*x = H1;

    if (x == NULL)

    {

        cout <<"Nothing to Extract"<< endl;

        returnx;

    }

    intmin = x->n;

    node \*p = x;

    while (p->sibling != NULL)

    {

        if ((p->sibling)->n<min)

        {

            min = (p->sibling)->n;

            t = p;

            x = p->sibling;

        }

        p = p->sibling;

    }

    if (t == NULL&&x->sibling == NULL)

        H1 = NULL;

    elseif (t == NULL)

        H1 = x->sibling;

    elseif (t->sibling == NULL)

        t = NULL;

    else

        t->sibling = x->sibling;

    if (x->child != NULL)

    {

        Revert\_list(x->child);

        (x->child)->sibling = NULL;

    }

    H = Union(H1, Hr);

    returnx;

}

// Reverse List

intBinomialHeap::Revert\_list(node\*y)

{

    if (y->sibling != NULL)

    {

        Revert\_list(y->sibling);

        (y->sibling)->sibling = y;

    }

    else

    {

        Hr = y;

    }

}

// Search Nodes in Binomial Heap

node\*BinomialHeap::Search(node\*H, intk)

{

    node \*x = H;

    node \*p = NULL;

    if (x->n == k)

    {

        p = x;

        returnp;

    }

    if (x->child != NULL&&p == NULL)

        p = Search(x->child, k);

    if (x->sibling != NULL&&p == NULL)

        p = Search(x->sibling, k);

    returnp;

}

// Decrease key of a node

intBinomialHeap::Decrease\_key(node\*H, inti, intk)

{

    inttemp;

    node \*p;

    node \*y;

    node \*z;

    p = Search(H, i);

    if (p == NULL)

    {

        cout <<"Invalid choice of key"<< endl;

        return0;

    }

    if (k>p->n)

    {

        cout <<"Error!! New key is greater than current key"<< endl;

        return0;

    }

    p->n = k;

    y = p;

    z = p->parent;

    while (z != NULL&&y->n<z->n)

    {

        temp = y->n;

        y->n = z->n;

        z->n = temp;

        y = z;

        z = z->parent;

    }

    cout <<"Key reduced successfully"<< endl;

}

// Delete Nodes in Binomial Heap

intBinomialHeap::Delete(node\*H, intk)

{

    node \*np;

    if (H == NULL)

    {

        cout <<"\nHEAP EMPTY!!!!!";

        return0;

    }

    Decrease\_key(H, k, -1000);

    np = Extract\_Min(H);

    if (np != NULL)

        cout <<"Node Deleted Successfully"<< endl;

}

// Driver Code

intmain()

{

    intn, m, l, i;

    BinomialHeapbh;

    node \*p;

    node \*H;

    H = bh.Initializeheap();

    charch;

    while (1)

    {

        cout <<"----------------------------"<< endl;

        cout <<" BINOMIAL HEAP"<< endl;

        cout <<"----------------------------"<< endl;

        cout <<"1. Insert Element in the heap"<< endl;

        cout <<"2. Extract Minimum key node"<< endl;

        cout <<"3. Decrease key of a node"<< endl;

        cout <<"4. Delete a node"<< endl;

        cout <<"5. Display Heap"<< endl;

        cout <<"6. Exit"<< endl;

        cout <<"Enter Your Choice: ";

        cin >>l;

        switch (l)

        {

        case1:

            cout <<"Enter the element to be inserted: ";

            cin >>m;

            p = bh.Create\_node(m);

            H = bh.Insert(H, p);

            break;

        case2:

            p = bh.Extract\_Min(H);

            if (p != NULL)

                cout <<"The node with minimum key: "<<p->n<< endl;

            else

                cout <<"Heap is empty"<< endl;

            break;

        case3:

            cout <<"Enter the key to be decreased: ";

            cin >>m;

            cout <<"Enter new key value: ";

            cin >>l;

            bh.Decrease\_key(H, m, l);

            break;

        case4:

            cout <<"Enter the key to be deleted: ";

            cin >>m;

            bh.Delete(H, m);

            break;

        case5:

            cout <<"The Heap is: "<< endl;

            bh.Display(H);

            break;

        case6:

            exit(1);

        default:

            cout <<"Wrong Choice";

        }

    }

    return0;

}

**Implementation of Interval Trees.**

#include<bits/stdc++.h>

usingnamespacestd;

// class representing the node of interval tree

classNode {

    public:

    intl, r, max;

    Node\* left;

    Node\* right;

    Node(intlVal, intrVal) {

        l = lVal;

        r = rVal;

        max = rVal;

        left = right = NULL;

    }

};

// Function to create a new Node

Node\*newNode(intl, intr) {

    Node\* node = newNode(l, r);

    returnnode;

}

// Function to insert an interval in interval tree

Node\*insert(Node\*root, intl, intr) {

    if (root == NULL) {

        returnnewNode(l, r);

    }

    if (l<root->l) {

        root->left = insert(root->left, l , r);

    } elseif (l>root->l) {

        root->right = insert(root->right, l, r);

    } else {

        if (r<root->r) {

            root->left = insert(root->left, l, r);

        } else {

            root->right = insert(root->right, l, r);

        }

    }

    // If current node's max is less than r, then update max

    if (root->max<r) {

        root->max = r;

    }

    returnroot;

}

boolcheckOverlap(Node\*root, intl, intr) {

    // If current node is null, return false

    if (root == NULL) {

        returnfalse;

    }

    // If overlaps return true

    if (root->l<= r&&l<= root->r) {

        returntrue;

    }

    // If max value of current is greater than starting point of I(l)

    // search in left subtree

    if (root->left != NULL&&root->left->max>= l) {

        returncheckOverlap(root->left, l, r);

    }

    // Else search in right subtree

    returncheckOverlap(root->right, l, r);

}

// Function to delete a binary tree, same as normal delete in a BST

Node\*deleteInterval(Node\*root, intl, intr) {

    if (root == NULL) {

        returnNULL;

    }

    if (l<root->l) {

        root->left = deleteInterval(root->left, l, r);

    } elseif (l>root->l) {

        root->right = deleteInterval(root->right, l, r);

    } else {

        if (r<root->r) {

            root->left = deleteInterval(root->left, l, r);

        } elseif (r>root->r) {

            root->right = deleteInterval(root->right, l, r);

        } else {

            // This is the interval to be deleted

            if (root->left == NULL)

                returnroot->right;

            elseif (root->right == NULL)

                returnroot->left;

            // Find the minimum value in the right subtree of root

            Node\* curr = root->right;

            while (curr->left != NULL) {

                curr = curr->left;

            }

            root->l = curr->l;

            root->r = curr->r;

            root->right = deleteInterval(root->right, root->l, root->r);

        }

    }

    returnroot;

}

intmain() {

    Node \*root = NULL;

    root = insert(root, 5, 10);

    root = insert(root, 3, 8);

    root = insert(root, 10, 15);

    root = insert(root, 16, 18);

    root = insert(root, 9, 11);

    root = insert(root, 1, 1);

    if (checkOverlap(root, 1, 2))

        cout<<"true"<<endl;

    else

        cout<<"false"<<endl;

    if (checkOverlap(root, 7, 11))

        cout<<"true"<<endl;

    else

        cout<<"false"<<endl;

    if (checkOverlap(root, 20, 25))

        cout<<"true"<<endl;

    else

        cout<<"false"<<endl;

    root = deleteInterval(root, 1, 1);

    root = deleteInterval(root,10, 15);

    if (checkOverlap(root, 12, 14))

        cout<<"true"<<endl;

    else

        cout<<"false"<<endl;

    if (checkOverlap(root, 8, 15))

        cout<<"true"<<endl;

    else

        cout<<"false"<<endl;

    if (checkOverlap(root, 1, 2))

        cout<<"true"<<endl;

    else

        cout<<"false"<<endl;

    return0;

}

**Implementation of Segment Tree.**

#include<bits/stdc++.h>

usingnamespacestd;

// A utility function to get the middle index from corner indexes.

intgetMid(ints, inte) { returns + (e - s) / 2; }

intgetSumUtil(int\*st, intss, intse, intqs, intqe, intsi)

{

    // If segment of this node is a part of given range, then return

    // the sum of the segment

    if (qs<= ss&&qe>= se)

        returnst[si];

    // If segment of this node is outside the given range

    if (se<qs || ss>qe)

        return0;

    // If a part of this segment overlaps with the given range

    intmid = getMid(ss, se);

    returngetSumUtil(st, ss, mid, qs, qe, 2 \* si + 1) +

           getSumUtil(st, mid + 1, se, qs, qe, 2 \* si + 2);

}

voidupdateValueUtil(int\*st, intss, intse, inti, intdiff, intsi)

{

    // Base Case: If the input index lies outside the range of

    // this segment

    if (i<ss || i>se)

        return;

    // If the input index is in range of this node, then update

    // the value of the node and its children

    st[si] = st[si] + diff;

    if (se != ss)

    {

        intmid = getMid(ss, se);

        updateValueUtil(st, ss, mid, i, diff, 2 \* si + 1);

        updateValueUtil(st, mid + 1, se, i, diff, 2 \* si + 2);

    }

}

// The function to update a value in input array and segment tree.

// It uses updateValueUtil() to update the value in segment tree

voidupdateValue(intarr[], int\*st, intn, inti, intnew\_val)

{

    // Check for erroneous input index

    if (i<0 || i>n - 1)

    {

        cout <<"Invalid Input";

        return;

    }

    // Get the difference between new value and old value

    intdiff = new\_val - arr[i];

    // Update the value in array

    arr[i] = new\_val;

    // Update the values of nodes in segment tree

    updateValueUtil(st, 0, n - 1, i, diff, 0);

}

// Return sum of elements in range from index qs (query start)

// to qe (query end). It mainly uses getSumUtil()

intgetSum(int\*st, intn, intqs, intqe)

{

    // Check for erroneous input values

    if (qs<0 || qe>n - 1 || qs>qe)

    {

        cout <<"Invalid Input";

        return -1;

    }

    returngetSumUtil(st, 0, n - 1, qs, qe, 0);

}

// A recursive function that constructs Segment Tree for array[ss..se].

// si is index of current node in segment tree st

intconstructSTUtil(intarr[], intss, intse, int\*st, intsi)

{

    // If there is one element in array, store it in current node of

    // segment tree and return

    if (ss == se)

    {

        st[si] = arr[ss];

        returnarr[ss];

    }

    // If there are more than one elements, then recur for left and

    // right subtrees and store the sum of values in this node

    intmid = getMid(ss, se);

    st[si] = constructSTUtil(arr, ss, mid, st, si \* 2 + 1) +

             constructSTUtil(arr, mid + 1, se, st, si \* 2 + 2);

    returnst[si];

}

/\* Function to construct segment tree from given array. This function

allocates memory for segment tree and calls constructSTUtil() to

fill the allocated memory \*/

int\*constructST(intarr[], intn)

{

    // Allocate memory for the segment tree

    // Height of segment tree

    intx = (int)(ceil(log2(n)));

    // Maximum size of segment tree

    intmax\_size = 2 \* (int)pow(2, x) - 1;

    // Allocate memory

    int \*st = newint[max\_size];

    // Fill the allocated memory st

    constructSTUtil(arr, 0, n - 1, st, 0);

    // Return the constructed segment tree

    returnst;

}

// Driver program to test above functions

intmain()

{

    intarr[] = {1, 3, 5, 7, 9, 11};

    intn = sizeof(arr) / sizeof(arr[0]);

    // Build segment tree from given array

    int \*st = constructST(arr, n);

    // Print sum of values in array from index 1 to 3

    cout <<"Sum of values in given range = "<<getSum(st, n, 1, 3) << endl;

    // Update: set arr[1] = 10 and update corresponding

    // segment tree nodes

    updateValue(arr, st, n, 1, 10);

    // Find sum after the value is updated

    cout <<"Updated sum of values in given range = "

         <<getSum(st, n, 1, 3) << endl;

    return0;

}

**Implementation of B Tree.**

// Searching a key on a B-tree in C

#include<stdio.h>

#include<stdlib.h>

#define MAX 3

#define MIN 2

structBTreeNode

{

    intval[MAX + 1], count;

    structBTreeNode\*link[MAX + 1];

};

structBTreeNode\*root;

// Create a node

structBTreeNode\*createNode(intval, structBTreeNode\*child)

{

    structBTreeNode\*newNode;

    newNode = (structBTreeNode\*)malloc(sizeof(struct BTreeNode));

    newNode->val[1] = val;

    newNode->count = 1;

    newNode->link[0] = root;

    newNode->link[1] = child;

    returnnewNode;

}

// Insert node

voidinsertNode(intval, intpos, structBTreeNode\*node,

                structBTreeNode\*child)

{

    intj = node->count;

    while (j>pos)

    {

        node->val[j + 1] = node->val[j];

        node->link[j + 1] = node->link[j];

        j--;

    }

    node->val[j + 1] = val;

    node->link[j + 1] = child;

    node->count++;

}

// Split node

voidsplitNode(intval, int\*pval, intpos, structBTreeNode\*node,

               structBTreeNode\*child, structBTreeNode\*\*newNode)

{

    intmedian, j;

    if (pos>MIN)

        median = MIN + 1;

    else

        median = MIN;

    \*newNode = (structBTreeNode\*)malloc(sizeof(struct BTreeNode));

    j = median + 1;

    while (j<= MAX)

    {

        (\*newNode)->val[j - median] = node->val[j];

        (\*newNode)->link[j - median] = node->link[j];

        j++;

    }

    node->count = median;

    (\*newNode)->count = MAX - median;

    if (pos<= MIN)

    {

        insertNode(val, pos, node, child);

    }

    else

    {

        insertNode(val, pos - median, \*newNode, child);

    }

    \*pval = node->val[node->count];

    (\*newNode)->link[0] = node->link[node->count];

    node->count--;

}

// Set the value

intsetValue(intval, int\*pval,

             structBTreeNode\*node, structBTreeNode\*\*child)

{

    intpos;

    if (!node)

    {

        \*pval = val;

        \*child = NULL;

        return1;

    }

    if (val<node->val[1])

    {

        pos = 0;

    }

    else

    {

        for (pos = node->count;

             (val<node->val[pos] &&pos>1); pos--)

            ;

        if (val == node->val[pos])

        {

            printf("Duplicates are not permitted\n");

            return0;

        }

    }

    if (setValue(val, pval, node->link[pos], child))

    {

        if (node->count<MAX)

        {

            insertNode(\*pval, pos, node, \*child);

        }

        else

        {

            splitNode(\*pval, pval, pos, node, \*child, child);

            return1;

        }

    }

    return0;

}

// Insert the value

voidinsert(intval)

{

    intflag, i;

    structBTreeNode\*child;

    flag = setValue(val, &i, root, &child);

    if (flag)

        root = createNode(i, child);

}

// Search node

voidsearch(intval, int\*pos, structBTreeNode\*myNode)

{

    if (!myNode)

    {

        return;

    }

    if (val<myNode->val[1])

    {

        \*pos = 0;

    }

    else

    {

        for (\*pos = myNode->count;

             (val<myNode->val[\*pos] && \*pos>1); (\*pos)--)

            ;

        if (val == myNode->val[\*pos])

        {

            printf("%d is found", val);

            return;

        }

    }

    search(val, pos, myNode->link[\*pos]);

    return;

}

// Traverse then nodes

voidtraversal(structBTreeNode\*myNode)

{

    inti;

    if (myNode)

    {

        for (i = 0; i<myNode->count; i++)

        {

            traversal(myNode->link[i]);

            printf("%d ", myNode->val[i + 1]);

        }

        traversal(myNode->link[i]);

    }

}

intmain()

{

    intval, ch;

    insert(8);

    insert(9);

    insert(10);

    insert(11);

    insert(15);

    insert(16);

    insert(17);

    insert(18);

    insert(20);

    insert(23);

    traversal(root);

    printf("\n");

    search(11, &ch, root);

}

**Implementation of Perfect Hashing.**

#include<time.h>

#include<algorithm>

#include<iostream>

#include<vector>

#include<list>

#include<stdio.h>

usingstd::cin;

usingstd::cout;

usingstd::endl;

usingstd::list;

usingstd::vector;

typedeflonglongintlong\_int;

constintmax\_int = 1000000001;

inlineinthash(long\_inta\_prime, long\_intb\_prime, intp\_prime, inttable\_size, intkey)

{

    return (((a\_prime \* key + b\_prime) % p\_prime) % table\_size);

}

classBucket

{

    vector<int>\_cells;

    intsize;

    long\_inthash\_a;

    long\_inthash\_b;

    intprime;

public:

    Bucket() {}

    voidInitialize()

    {

        prime = 17;

        hash\_a = std::rand() % prime;

        hash\_b = 1 + std::rand() % (prime - 1);

    }

    voidConstruct(list<int>&input)

    {

        if (input.empty())

        {

            size = 0;

            return;

        }

        size = input.size() \* input.size();

        boolflag = true;

        while (flag)

        {

            \_cells.assign(size, max\_int);

            Initialize();

            list<int>::iterator elem = input.begin();

            while (elem != input.end() &&flag)

            {

                inthashKey = hash(hash\_a, hash\_b, prime, size, \*elem);

                if (hashKey<0)

                    hashKey = -hashKey;

                if (\_cells[hashKey] != max\_int)

                {

                    flag = false;

                    break;

                }

                \_cells[hashKey] = \*elem;

                ++elem;

            }

            if (!flag)

                flag = true;

            else

                flag = false;

        }

    }

    boolContains(intelem)

    {

        if (size == 0)

            returnfalse;

        inthashKey = hash(hash\_a, hash\_b, prime, size, elem);

        if (hashKey<0)

            hashKey = -hashKey;

        if (\_cells[hashKey] == elem)

            returntrue;

        returnfalse;

    }

};

classFixedSet

{

    int\_tableSize;

    long\_int\_hashFuncA;

    long\_int\_hashFuncB;

    int\_primeNumber;

    vector<list<int>> \_elementsInCells;

    vector<Bucket>\_buckets;

public:

    FixedSet()

    {

        \_primeNumber = 100013;

        \_hashFuncA = std::rand() % \_primeNumber;

        \_hashFuncB = 1 + std::rand() % (\_primeNumber - 1);

    }

    voidsetTableSize(intsize)

    {

        \_tableSize = size;

        \_buckets.resize(size);

    }

    voidInitialize(constvector<int>&numbers)

    {

        \_tableSize = numbers.size();

        \_buckets.resize(numbers.size());

        \_elementsInCells.resize(numbers.size());

        for (inti = 0; i<numbers.size(); ++i)

        {

            inthashKey = hash(\_hashFuncA, \_hashFuncB, \_primeNumber,

                               \_tableSize, numbers[i]);

            if (hashKey<0)

                hashKey = -hashKey;

            \_elementsInCells[hashKey].push\_back(numbers[i]);

        }

        for (inti = 0; i<numbers.size(); ++i)

        {

            \_buckets[i].Construct(\_elementsInCells[i]);

        }

    }

    boolContains(intnumber)

    {

        inthashKey = hash(\_hashFuncA, \_hashFuncB, \_primeNumber,

                           \_tableSize, number);

        if (hashKey<0)

            hashKey = -hashKey;

        return\_buckets[hashKey].Contains(number);

    }

};

intmain(intargc, char\*argv[])

{

    clock\_tbegin, end;

    doubletime\_spent;

    std::srand(time(NULL));

    intnumberOfElements;

    printf("Enter number of elements : ");

    scanf("%i", &numberOfElements);

    FixedSetfs;

    begin = clock();

    vector<int> inputVector;

    fs.setTableSize(numberOfElements);

    printf("\nEnter Elements : ");

    for (inti = 0; i<numberOfElements; ++i)

    {

        intelemValue;

        scanf("%d", &elemValue);

        inputVector.push\_back(elemValue);

    }

    fs.Initialize(inputVector);

    end = clock();

    intnumberOfElementsForSearch;

    printf("\nNumber of elements to search : ");

    scanf("%i", &numberOfElementsForSearch);

    for (inti = 0; i<numberOfElementsForSearch; ++i)

    {

        intelem;

        printf("\nEnter element : ");

        scanf("%d", &elem);

        if (fs.Contains(elem))

        {

            cout <<"Yes , element exists"<< endl;

        }

        else

        {

            cout <<"No , element doesn't exists"<< endl;

        }

    }

    time\_spent = (double)(end - begin) / CLOCKS\_PER\_SEC;

    cout <<"Time spent : "<<time\_spent<< endl;

    return0;

}

Implementation of Bloom Filter using two hash functions.

#include<bits/stdc++.h>

#define ll long long

usingnamespacestd;

// hash 1

inth1(strings, intarrSize)

{

    llinthash = 0;

    for (inti = 0; i<s.size(); i++)

    {

        hash = (hash + ((int)s[i]));

        hash = hash % arrSize;

    }

    returnhash;

}

// hash 2

inth2(strings, intarrSize)

{

    llinthash = 1;

    for (inti = 0; i<s.size(); i++)

    {

        hash = hash + pow(19, i) \* s[i];

        hash = hash % arrSize;

    }

    returnhash % arrSize;

}

// hash 3

inth3(strings, intarrSize)

{

    llinthash = 7;

    for (inti = 0; i<s.size(); i++)

    {

        hash = (hash \* 31 + s[i]) % arrSize;

    }

    returnhash % arrSize;

}

// hash 4

inth4(strings, intarrSize)

{

    llinthash = 3;

    intp = 7;

    for (inti = 0; i<s.size(); i++) {

        hash += hash \* 7 + s[0] \* pow(p, i);

        hash = hash % arrSize;

    }

    returnhash;

}

// lookup operation

boollookup(bool\*bitarray, intarrSize, strings)

{

    inta = h1(s, arrSize);

    intb = h2(s, arrSize);

    intc = h3(s, arrSize);

    intd = h4(s, arrSize);

   if (bitarray[a] &&bitarray[b] &&bitarray

        &&bitarray[d])

        returntrue;

    else

        returnfalse;

}

// insert operation

voidinsert(bool\*bitarray, intarrSize, strings)

{

    // check if the element in already present or not

    if (lookup(bitarray, arrSize, s))

        cout <<s<<" is Probably already present"<< endl;

    else

    {

        inta = h1(s, arrSize);

        intb = h2(s, arrSize);

        intc = h3(s, arrSize);

        intd = h4(s, arrSize);

       bitarray[a] = true;

        bitarray[b] = true;

        bitarray[c] = true;

        bitarray[d] = true;

       cout <<s<<" inserted"<< endl;

    }

}

// Driver Code

intmain()

{

    boolbitarray[100] = { false };

    intarrSize = 100;

    string sarray[33]

        = { "abound", "abounds",     "abundance",

            "abundant", "accessible", "bloom",

            "blossom", "bolster",     "bonny",

            "bonus", "bonuses",     "coherent",

            "cohesive", "colorful",     "comely",

            "comfort", "gems",         "generosity",

            "generous", "generously", "genial",

            "bluff", "cheater",     "hate",

            "war",     "humanity",     "racism",

            "hurt",     "nuke",         "gloomy",

            "facebook", "geeksforgeeks", "twitter" };

    for (inti = 0; i<33; i++) {

        insert(bitarray, arrSize, sarray[i]);

    }

    return0;

}

I