**ADS Experiment List Practical Exam**

1. Implementation of Multi-Pop Stack and perform amortized analysis on it.

#include <iostream>

#include <stack>

#include <cstdlib>

#include <ctime>

using namespace std;

int main()

{

    int n;

    cout << "Enter the number of elements in stack : ";

    cin >> n;

    int arr[n];

    srand(time(0));

    for (int i = 0; i < n; i++)

    {

        arr[i] = rand() % 5 + 2;

        printf("%d ", arr[i]);

    }

    printf("--------------------\n");

    int cost = 0;

    int accountingcost = 0;

    int potentialcost = 0;

    stack<int> stk;

    for (int i = 0; i < n; i++)

    {

        // printf("rand: %d, k: ", srand());

        int k = rand() % 3;

        printf("%d ", k);

        if (k == 0)

        {

            stk.push(arr[i]);

            cost += 1;

            accountingcost += 1;

            cout << stk.top() << " push accounting cost : "

                 << "1" << endl;

            potentialcost += 1;

        }

        else if (k == 1)

        {

            if (stk.size() >= 1)

            {

                cout << stk.top() << " pop accounting cost : "

                     << "-1" << endl;

                stk.pop();

                cost += 1;

                accountingcost -= 1;

                potentialcost -= 1;

            }

        }

        else

        {

            int k = rand() % 5;

            int s = stk.size();

            int minimum = min(k, s);

            for (int i = 0; i < minimum; i++)

            {

                cout << stk.top() << " multipop accounting cost : "

                     << "-1" << endl;

                stk.pop();

                accountingcost -= 1;

                cost += 1;

                potentialcost -= 1;

            }

        }

    }

    cout << "Cost: " << cost << endl;

    cout << "Aggregate cost: " << (float)cost / n << endl;

    cout << "Accounting: " << accountingcost << endl;

    cout << "Potential or Excess: " << potentialcost << endl;

    cout << "Final stack size: " << stk.size();

    return 0;

}

1. Implementation of Dynamic Table and perform amortized analysis on it.

#include <bits/stdc++.h>

using namespace std;

class DynamicArray {

  private: int currSize;

  int capacity;

  int \* a;

  int cost;

  void resizeArray() {

    if (currSize == capacity) {

      cout << "DOUBLING SIZEEEEE" << endl;

      int \* temp = new int(2 \* capacity);

      for (int i = 0; i < capacity; i++) {

        temp[i] = a[i];

      }

      free(a);

      a = temp;

      cost = capacity + 1;

      capacity = 2 \* capacity;

    }

  }

  public: DynamicArray() {

    a = new int(1);

    cost = 1;

    currSize = 0;

    capacity = 1;

  }

  void insertAtEnd(int x) {

    if (currSize == capacity) {

      resizeArray();

    }

    else cost = 1;

    a[currSize] = x;

    currSize++;

  }

  int getArraySize() {

    return currSize;

  }

  int getCost(){

      return cost;

  }

  int getCapacity() {

    return capacity;

  }

};

int main() {

  DynamicArray a;

  int n;

  cout << "Enter the number of elements ";

  cin >> n;

  for (int i = 0; i < n; i++) {

    a.insertAtEnd(i);

    cout << "Curr Size: " << a.getArraySize() << endl;

    cout << "Capacity: " << a.getCapacity() << endl;

    cout << "Cost: "<< a.getCost()<<endl;

  }

}

1. Implementation of Binary Counter and perform amortized analysis on it.

#include <bits/stdc++.h>

using namespace std;

void display(vector<int> v)

{

    for (int i = 3 ; i >= 0; i--)

    {

        cout << v[i] << " ";

    }

}

int increment(vector<int> &binary)

{

    int i = 0;

    int no\_of\_flips = 0;

    while (i < binary.size() && binary[i] == 1)

    {

        binary[i] = 0;

        no\_of\_flips++;

        i++;

    }

    if (i < binary.size())

    {

        binary[i] = 1;

        no\_of\_flips++;

    }

    return no\_of\_flips;

}

vector<int> convert(int n)

{

    vector<int> binary(4, 0);

    int rem = 0;

    int i = 0;

    while (n > 0)

    {

        rem = n % 2;

        binary[i] = rem;

        i++;

        n /= 2;

    }

    return binary;

}

void binary\_counter(int n, int k)

{

    vector<int> binary = convert(n);

    cout << "Original Number : ";

    display(binary);

    cout << endl;

    int count = 0;

    while (count < k)

    {

        int tc = increment(binary);

        display(binary);

        cout << "Total Cost : " << tc;

        cout << endl;

        count++;

    }

}

int main()

{

    cout << "Enter starting number : ";

    int n;

    cin >> n;

    cout << "Enter number of times to increment : ";

    int a;

    cin >> a;

    binary\_counter(n, a);

    return 0;

}

1. Implementation of standard tries and perform insert operation, search and delete operation.

// C++ implementation of delete

// operations on Trie

#include <bits/stdc++.h>

using namespace std;

const int ALPHABET\_SIZE = 26;

// trie node

struct TrieNode

{

    struct TrieNode \*children[ALPHABET\_SIZE];

    // isEndOfWord is true if the node represents

    // end of a word

    bool isEndOfWord;

};

// Returns new trie node (initialized to NULLs)

struct TrieNode \*getNode(void)

{

    struct TrieNode \*pNode = new TrieNode;

    pNode->isEndOfWord = false;

    for (int i = 0; i < ALPHABET\_SIZE; i++)

        pNode->children[i] = NULL;

    return pNode;

}

// If not present, inserts key into trie

// If the key is prefix of trie node, just

// marks leaf node

void insert(struct TrieNode \*root, string key)

{

    struct TrieNode \*pCrawl = root;

    for (int i = 0; i < key.length(); i++)

    {

        int index = key[i] - 'a';

        if (!pCrawl->children[index])

            pCrawl->children[index] = getNode();

        pCrawl = pCrawl->children[index];

    }

    // mark last node as leaf

    pCrawl->isEndOfWord = true;

}

// Returns true if key presents in trie, else

// false

bool search(struct TrieNode \*root, string key)

{

    struct TrieNode \*pCrawl = root;

    for (int i = 0; i < key.length(); i++)

    {

        int index = key[i] - 'a';

        if (!pCrawl->children[index])

            return false;

        pCrawl = pCrawl->children[index];

    }

    return (pCrawl != NULL && pCrawl->isEndOfWord);

}

// Returns true if root has no children, else false

bool isEmpty(TrieNode \*root)

{

    for (int i = 0; i < ALPHABET\_SIZE; i++)

        if (root->children[i])

            return false;

    return true;

}

// Recursive function to delete a key from given Trie

TrieNode \*remove(TrieNode \*root, string key, int depth = 0)

{

    // If tree is empty

    if (!root)

        return NULL;

    // If last character of key is being processed

    if (depth == key.size())

    {

        // This node is no more end of word after

        // removal of given key

        if (root->isEndOfWord)

            root->isEndOfWord = false;

        // If given is not prefix of any other word

        if (isEmpty(root))

        {

            delete (root);

            root = NULL;

        }

        return root;

    }

    // If not last character, recur for the child

    // obtained using ASCII value

    int index = key[depth] - 'a';

    root->children[index] =

        remove(root->children[index], key, depth + 1);

    // If root does not have any child (its only child got

    // deleted), and it is not end of another word.

    if (isEmpty(root) && root->isEndOfWord == false)

    {

        delete (root);

        root = NULL;

    }

    return root;

}

// Driver

int main()

{

    // Input keys (use only 'a' through 'z'

    // and lower case)

    string keys[10] ;

    cout<<"Enter the no of strings";

    cin>>n;

    for (int i=0;i<n;i++){

        cout<<"Enter the strings ";

        cin>>keys[i];

    }

    struct TrieNode \*root = getNode();

    // Construct trie

    for (int i = 0; i < n; i++)

        insert(root, keys[i]);

    // Search for different keys

    search(root, "the") ? cout << "Yes\n" : cout << "No\n";

    search(root, "these") ? cout << "Yes\n" : cout << "No\n";

    remove(root, "heroplane");

    search(root, "hero") ? cout << "Yes\n" : cout << "No\n";

    return 0;

}

1. Implement the code to merge two binomial Heap.

#include <iostream>

#include <cstdlib>

using namespace std;

// Node Declaration

struct node

{

    int n;

    int degree;

    node \*parent;

    node \*child;

    node \*sibling;

};

// Class Declaration

class BinomialHeap

{

private:

    node \*H;

    node \*Hr;

    int count;

public:

    node \*Initializeheap();

    int Binomial\_link(node \*, node \*);

    node \*Create\_node(int);

    node \*Union(node \*, node \*);

    node \*Insert(node \*, node \*);

    node \*Merge(node \*, node \*);

    int Display(node \*);

    node \*Search(node \*, int);

    BinomialHeap()

    {

        H = Initializeheap();

        Hr = Initializeheap();

        int count = 1;

    }

};

// Initialize Heap

node \*BinomialHeap::Initializeheap()

{

    node \*np;

    np = NULL;

    return np;

}

// Linking nodes in Binomial Heap

int BinomialHeap::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(int k)

{

    node \*p = new node;

    p->n = k;

    return p;

}

// 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);

    return H;

}

// Union Nodes in Binomial Heap

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

{

    node \*H = Initializeheap();

    H = Merge(H1, H2);

    if (H == NULL)

        return H;

    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;

    }

    return H;

}

// 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;

            else if (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;

        }

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

        {

            a = y->sibling;

            y->sibling = z;

            y = a;

        }

        else

        {

            b = z->sibling;

            z->sibling = y;

            z = b;

        }

    }

    return H;

}

int BinomialHeap::Display(node \*H)

{

    if (H == NULL)

    {

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

        return 0;

    }

    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;

}

int main()

{

    int n, m, l, i;

    BinomialHeap bh;

    node \*p;

    node \*H;

    H = bh.Initializeheap();

    char ch;

    while (1)

    {

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

        cout << " BINOMIAL HEAP" << endl;

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

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

        cout << "2. Display Heap" << endl;

        cout << "3. Exit" << endl;

        cout << "Enter Your Choice: ";

        cin >> l;

        switch (l)

        {

        case 1:

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

            cin >> m;

            p = bh.Create\_node(m);

            H = bh.Insert(H, p);

            break;

        case 2:

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

            bh.Display(H);

            break;

        case 3:

            exit(1);

        default:

            cout << "Wrong Choice";

        }

    }

    return 0;

}

1. Write down the code to insert an element in Red Black Tree.

#include <iostream>

 using namespace std;

 struct node

 {

     int key;

     node \*parent;

     char color;

     node \*left;

     node \*right;

 };

 class RBtree

 {

     node \*root;

     node \*q;

 public:

     RBtree()

     {

         q = NULL;

         root = NULL;

     }

     void insert();

     void insertfix(node \*);

     void leftrotate(node \*);

     void rightrotate(node \*);

     void del();

     node \*successor(node \*);

     void disp();

     void display(node \*);

 };

 void RBtree::insert()

 {

     int z, i = 0;

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

     cin >> z;

     node \*p, \*q;

     node \*t = new node;

     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);

 }

 void RBtree::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';

     }

 }

 void RBtree::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;

     }

 }

 void RBtree::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;

     }

     return y;

 }

 void RBtree::disp()

 {

     display(root);

 }

 void RBtree::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"\*/

     }

 }

 int main()

 {

     int ch, y = 0;

     RBtree obj;

     do

     {

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

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

         cout << "\n 2. Display the tree ";

         cout << "\n 3. Exit ";

         cout << "\nEnter Your Choice: ";

         cin >> ch;

         switch (ch)

         {

         case 1:

             obj.insert();

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

             break;

         case 2:

             obj.disp();

             break;

         case 3:

             y = 1;

             break;

         default:

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

         }

         cout << endl;

     } while (y != 1);

     return 1;

 }

1. Write down the code to insert an element in B Tree.

#include <iostream>

using namespace std;

class BTreeNode

{

  int \*keys;

  int t;

  BTreeNode \*\*C;

  int n;

  bool leaf;

public:

  BTreeNode(int \_t, bool \_leaf);

  void traverse();

  BTreeNode \*search(int k);

  int findKey(int k);

  void insertNonFull(int k);

  void splitChild(int i, BTreeNode \*y);

  void merge(int idx);

  friend class BTree;

};

class BTree

{

  BTreeNode \*root;

  int t;

public:

  BTree(int \_t)

  {

    root = NULL;

    t = \_t;

  }

  void traverse()

  {

    if (root != NULL)

      root->traverse();

  }

  BTreeNode \*search(int k)

  {

    return (root == NULL) ? NULL : root->search(k);

  }

  void insert(int k);

};

BTreeNode::BTreeNode(int t1, bool leaf1)

{

  t = t1;

  leaf = leaf1;

  keys = new int[2 \* t - 1];

  C = new BTreeNode \*[2 \* t];

  n = 0;

}

int BTreeNode::findKey(int k)

{

  int idx = 0;

  while (idx < n && keys[idx] < k)

    ++idx;

  return idx;

}

void BTreeNode::merge(int idx)

{

  BTreeNode \*child = C[idx];

  BTreeNode \*sibling = C[idx + 1];

  child->keys[t - 1] = keys[idx];

  for (int i = 0; i < sibling->n; ++i)

    child->keys[i + t] = sibling->keys[i];

  if (!child->leaf)

  {

    for (int i = 0; i <= sibling->n; ++i)

      child->C[i + t] = sibling->C[i];

  }

  for (int i = idx + 1; i < n; ++i)

    keys[i - 1] = keys[i];

  for (int i = idx + 2; i <= n; ++i)

    C[i - 1] = C[i];

  child->n += sibling->n + 1;

  n--;

  delete (sibling);

  return;

}

void BTree::insert(int k)

{

  if (root == NULL)

  {

    root = new BTreeNode(t, true);

    root->keys[0] = k;

    root->n = 1;

  }

  else

  {

    if (root->n == 2 \* t - 1)

    {

      BTreeNode \*s = new BTreeNode(t, false);

      s->C[0] = root;

      s->splitChild(0, root);

      int i = 0;

      if (s->keys[0] < k)

        i++;

      s->C[i]->insertNonFull(k);

      root = s;

    }

    else

      root->insertNonFull(k);

  }

}

void BTreeNode::insertNonFull(int k)

{

  int i = n - 1;

  if (leaf == true)

  {

    while (i >= 0 && keys[i] > k)

    {

      keys[i + 1] = keys[i];

      i--;

    }

    keys[i + 1] = k;

    n = n + 1;

  }

  else

  {

    while (i >= 0 && keys[i] > k)

      i--;

    if (C[i + 1]->n == 2 \* t - 1)

    {

      splitChild(i + 1, C[i + 1]);

      if (keys[i + 1] < k)

        i++;

    }

    C[i + 1]->insertNonFull(k);

  }

}

void BTreeNode::splitChild(int i, BTreeNode \*y)

{

  BTreeNode \*z = new BTreeNode(y->t, y->leaf);

  z->n = t - 1;

  for (int j = 0; j < t - 1; j++)

    z->keys[j] = y->keys[j + t];

  if (y->leaf == false)

  {

    for (int j = 0; j < t; j++)

      z->C[j] = y->C[j + t];

  }

  y->n = t - 1;

  for (int j = n; j >= i + 1; j--)

    C[j + 1] = C[j];

  C[i + 1] = z;

  for (int j = n - 1; j >= i; j--)

    keys[j + 1] = keys[j];

  keys[i] = y->keys[t - 1];

  n = n + 1;

}

void BTreeNode::traverse()

{

  int i;

  for (i = 0; i < n; i++)

  {

    if (leaf == false)

      C[i]->traverse();

    cout << " " << keys[i];

  }

  if (leaf == false)

    C[i]->traverse();

}

BTreeNode \*BTreeNode::search(int k)

{

  int i = 0;

  while (i < n && k > keys[i])

    i++;

  if (keys[i] == k)

    return this;

  if (leaf == true)

    return NULL;

  return C[i]->search(k);

}

int main()

{

  BTree t(3);

  t.insert(1);

  t.insert(3);

  t.insert(7);

  t.insert(10);

  t.insert(11);

  t.insert(13);

  t.insert(14);

  t.insert(15);

  t.insert(18);

  t.insert(16);

  t.insert(19);

  t.insert(24);

  t.insert(25);

  t.insert(26);

  t.insert(21);

  t.insert(4);

  t.insert(5);

  t.insert(20);

  t.insert(22);

  t.insert(2);

  t.insert(17);

  t.insert(12);

  t.insert(6);

  cout << "Traversal of tree constructed is\n";

  t.traverse();

  cout << endl;

  return 0;

}

1. Write down the code to find overlapping interval in interval tree.

#include <bits/stdc++.h>

using namespace std;

// class representing the node of interval tree

class Node

{

public:

    int l, r, max;

    Node \*left;

    Node \*right;

    Node(int lVal, int rVal)

    {

        l = lVal;

        r = rVal;

        max = rVal;

        left = right = NULL;

    }

};

// Function to create a new Node

Node \*newNode(int l, int r)

{

    Node \*node = new Node(l, r);

    return node;

}

// Function to insert an interval in interval tree

Node \*insert(Node \*root, int l, int r)

{

    if (root == NULL)

    {

        return newNode(l, r);

    }

    if (l < root->l)

    {

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

    }

    else if (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;

    }

    return root;

}

bool checkOverlap(Node \*root, int l, int r)

{

    // If current node is null, return false

    if (root == NULL)

    {

        return false;

    }

    // If overlaps return true

    if (root->l <= r && l <= root->r)

    {

        return true;

    }

    // 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)

    {

        return checkOverlap(root->left, l, r);

    }

    // Else search in right subtree

    return checkOverlap(root->right, l, r);

}

int main()

{

    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;

    return 0;

}

1. Write down the code to find sum of given interval in Segment Tree.

#include <bits/stdc++.h>

using namespace std;

struct Node

{

    // store the sum of the interval

    int sum;

    // store the interval in a pair of integers

    pair<int, int> interval; /\* L=interval.first and R=interval.second\*/

    Node \*left;              // points to left child

    Node \*right;             // points to right child

};

void build(vector<int> array, Node \*cur\_node, int L, int R)

{

    cur\_node->interval = make\_pair(L, R);

    if (L == R)

    {

        // if current node is a leaf node

        cur\_node->sum = array[L];

        cur\_node->left = NULL;

        cur\_node->right = NULL;

        return;

    }

    cur\_node->left = new Node;

    cur\_node->right = new Node;

    build(array, cur\_node->left, L, (L + R) / 2);

    build(array, cur\_node->right, (L + R) / 2 + 1, R);

    cur\_node->sum = cur\_node->left->sum + cur\_node->right->sum;

    return;

}

// returns the sum in the range [start, end]

int query(vector<int> array, Node \*cur\_node, int start, int end)

{

    int L = cur\_node->interval.first;

    int R = cur\_node->interval.second;

    if (R < start || L > end)

    {

        return 0;

    }

    if (start <= L && end >= R)

    {

        return cur\_node->sum;

    }

    int left\_index = query(array, cur\_node->left, start, end);

    int right\_index = query(array, cur\_node->right, start, end);

    return left\_index + right\_index;

}

int main()

{

    // define n and array

    int n = 5, a, b;

    vector<int> array = {1, 2, 1, 8, 7};

    Node \*root = new Node();

    build(array, root, 0, n - 1);

    cout << "Enter the range";

    cin >> a >> b;

    cout << "The sum in the interval is "

         << query(array, root, a, b) << '\n';

    return 0;

}

1. Implementation of Count-Min Sketch using two hash functions.

#include <bits/stdc++.h>

using namespace std;

int hash1(string s, int arrsize)

{

    int hash = 6;

    for (int i = 0; i < s.size(); i++)

    {

        hash = (hash \* 34 + s[i]) % arrsize;

    }

    return hash;

}

int hash2(string s, int arrsize)

{

    int hash = 19;

    for (int i = 0; i < s.size(); i++)

    {

        hash = (hash \* 21 + s[i]) % arrsize;

    }

    return hash;

}

void check(int \*arr1, int \*arr2, int size, string s)

{

    int h1 = hash1(s, size);

    int h2 = hash2(s, size);

    int a = arr1[h1];

    int b = arr2[h2];

    if (a && b)

        cout << "Count of " << s << " is: " << (min(a, b)) << endl;

    else

        cout << s << " doesnt exist" << endl;

}

void insert(int \*arr1, int \*arr2, int size, string s)

{

    int h1 = hash1(s, size);

    int h2 = hash2(s, size);

    arr1[h1] += 1;

    arr2[h2] += 1;

    cout << s << " is inserted" << endl;

}

int main()

{

    int size = 10;

    int arr1[10] = {0};

    int arr2[10] = {0};

    string s[] = {"jack", "jack", "jack", "lane", "lane", "lane", "lane", "lane", "bob", "bob", "bob", "bob", "bob", "bob", "alice", "zuck", "eve", "max"};

    for (int i = 0; i < sizeof(s) / sizeof(s[0]); i++)

    {

        insert(arr1, arr2, size, s[i]);

    }

    for (int i = 0; i < size; i++)

    {

        cout << arr1[i] << " ";

    }

    cout << endl;

    for (int i = 0; i < size; i++)

    {

        cout << arr2[i] << " ";

    }

    cout << endl;

    cout << "Enter name to lookup: ";

    string name;

    cin >> name;

    check(arr1, arr2, size, name);

    return 0;

}

1. Implementation of Bloom Filter Sketch using two hash functions

#include <bits/stdc++.h>

#define ll long long

using namespace std;

// hash 1

int h1(string s, int arrSize)

{

    ll int hash = 0;

    for (int i = 0; i < s.size(); i++)

    {

        hash = (hash + ((int)s[i]));

        hash = hash % arrSize;

    }

    return hash;

}

// hash 2

int h2(string s, int arrSize)

{

    ll int hash = 1;

    for (int i = 0; i < s.size(); i++)

    {

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

        hash = hash % arrSize;

    }

    return hash % arrSize;

}

// hash 3

int h3(string s, int arrSize)

{

    ll int hash = 7;

    for (int i = 0; i < s.size(); i++)

    {

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

    }

    return hash % arrSize;

}

// hash 4

int h4(string s, int arrSize)

{

    ll int hash = 3;

    int p = 7;

    for (int i = 0; i < s.size(); i++) {

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

        hash = hash % arrSize;

    }

    return hash;

}

// lookup operation

bool lookup(bool\* bitarray, int arrSize, string s)

{

    int a = h1(s, arrSize);

    int b = h2(s, arrSize);

    int c = h3(s, arrSize);

    int d = h4(s, arrSize);

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

        && bitarray[d])

        return true;

    else

        return false;

}

// insert operation

void insert(bool\* bitarray, int arrSize, string s)

{

    // check if the element in already present or not

    if (lookup(bitarray, arrSize, s))

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

    else

    {

        int a = h1(s, arrSize);

        int b = h2(s, arrSize);

        int c = h3(s, arrSize);

        int d = h4(s, arrSize);

        bitarray[a] = true;

        bitarray[b] = true;

        bitarray[c] = true;

        bitarray[d] = true;

       cout << s << " inserted" << endl;

    }

}

// Driver Code

int main()

{

    bool bitarray[100] = { false };

    int arrSize = 100;

    string sarray[10];

    for (int i =0;i<10;i++){

        cout<<"Enter the strings ";

        cin>>sarray[i];

    }

    for (int i = 0; i < 10; i++) {

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

    }

    return 0;

}