Week 13

Avni Arora_20103153_B6_week#13

1)

```
#include<iostream>
using namespace std;
int mrg=0,spl=0;
class node
{
    int *keys;
    int t;
    node **C;
    int n;
    bool leaf;

public:
    node(int _t, bool _leaf);
    void inorder();
    node *search_(int k);
    int Find_key(int k);
    void insert_non_full(int k);
    void split_the_child(int i, node *y);
    void delete_(int k);
    void Remove_leaf(int x_id);
    void Remove_nonleaf(int x_id);
    int predecessor_(int x_id);
    int successor_(int x_id);
    void fill(int x_id);
    void prev_borrow(int x_id);
    void next_borrow(int x_id);
    void merge(int x_id);
```

```
    friend class b_tree;
};

class b_tree
{
    node *root;
    int t;
public:
    b_tree(int _t)
    {
        root = NULL;
        t = _t;
    }

    void inorder()
    {
        if (root != NULL)
            root->inorder();
    }
    node* search_(int k)
    {
        return (root == NULL)? NULL : root->search_(k);
    }
    void insert(int k);
    void delete_(int k);
};

node::node(int t1, bool leaf1)
{
    t = t1;
```

```
    leaf = leaf1;
    keys = new int[2*t-1];
    C = new node *[2*t];
    n = 0;
}

int node::Find_key(int k)
{
    int x_id=0;
    while (x_id<n && keys[x_id] < k)
        ++x_id;
    return x_id;
}
void node::delete_(int k)
{
    int x_id = Find_key(k);
    if (x_id < n && keys[x_id] == k)
    {
        if (leaf)
            Remove_leaf(x_id);
        else
            Remove_nonleaf(x_id);
    }
    else
    {
        if (leaf)
        {
            cout << "The key "<< k <<" is does not exist in the tree\n";
            return;
        }
```

```
        bool flag = ( (x_id==n)? true : false );
        if (C[x_id]->n < t)
            fill(x_id);
        if (flag && x_id > n)
            C[x_id-1]->delete_(k);
        else
            C[x_id]->delete_(k);
    }
    return;
}

void node::Remove_leaf (int x_id)
{

    for (int i=x_id+1; i<n; ++i)
        keys[i-1] = keys[i];

    n--;
    return;
}
void node::Remove_nonleaf(int x_id)
{

    int k = keys[x_id];
    if (C[x_id]->n >= t)
    {
        int pred = predecessor_(x_id);
        keys[x_id] = pred;
        C[x_id]->delete_(pred);
    }
```

```
    else if  (C[x_id+1]->n >= t)
    {
        int succ = successor_(x_id);
        keys[x_id] = succ;
        C[x_id+1]->delete_(succ);
    }
    else
    {
        merge(x_id);
        C[x_id]->delete_(k);
    }
    return;
}

int node::predecessor_(int x_id)
{
    node *cur=C[x_id];
    while (!cur->leaf)
        cur = cur->C[cur->n];
    return cur->keys[cur->n-1];
}

int node::successor_(int x_id)
{

    node *cur = C[x_id+1];
    while (!cur->leaf)
        cur = cur->C[0];
    return cur->keys[0];
}
```

```
void node::fill(int x_id)
{

   if (x_id!=0 && C[x_id-1]->n>=t)
      prev_borrow(x_id);

   else if (x_id!=n && C[x_id+1]->n>=t)
      next_borrow(x_id);
   else
   {
      if (x_id != n)
         merge(x_id);
      else
         merge(x_id-1);
   }
   return;
}
void node::prev_borrow(int x_id)
{

   node *child=C[x_id];
   node *sibling=C[x_id-1];
   for (int i=child->n-1; i>=0; --i)
      child->keys[i+1] = child->keys[i];
   if (!child->leaf)
   {
      for(int i=child->n; i>=0; --i)
         child->C[i+1] = child->C[i];
   }
```

```
    child->keys[0] = keys[x_id-1];
    if(!child->leaf)
        child->C[0] = sibling->C[sibling->n];
    keys[x_id-1] = sibling->keys[sibling->n-1];

    child->n += 1;
    sibling->n -= 1;

    return;
}

void node::next_borrow(int x_id)
{

    node *child=C[x_id];
    node *sibling=C[x_id+1];
    child->keys[(child->n)] = keys[x_id];
    if (!(child->leaf))
        child->C[(child->n)+1] = sibling->C[0];

    keys[x_id] = sibling->keys[0];

    for (int i=1; i<sibling->n; ++i)
        sibling->keys[i-1] = sibling->keys[i];

    if (!sibling->leaf)
    {
        for(int i=1; i<=sibling->n; ++i)
            sibling->C[i-1] = sibling->C[i];
```

```
    }

    child->n += 1;
    sibling->n -= 1;

    return;
}

void node::merge(int x_id)
{
    node *child = C[x_id];
    node *sibling = C[x_id+1];
    child->keys[t-1] = keys[x_id];
    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=x_id+1; i<n; ++i)
        keys[i-1] = keys[i];
    for (int i=x_id+2; i<=n; ++i)
        C[i-1] = C[i];
    child->n += sibling->n+1;
    n--;
    mrg++;
    delete(sibling);
    return;
}
```

```
void b_tree::insert(int k)
{
   if (root == NULL)
   {
      root = new node(t, true);
      root->keys[0] = k;
      root->n = 1;
   }
   else
   {
      if (root->n == 2*t-1)
      {
         node *s = new node(t, false);
         s->C[0] = root;
         s->split_the_child(0, root);
         int i = 0;
         if (s->keys[0] < k)
            i++;
         s->C[i]->insert_non_full(k);
         root = s;
      }
      else
         root->insert_non_full(k);
   }
}

void node::insert_non_full(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)
        {
            split_the_child(i+1, C[i+1]);
            if (keys[i+1] < k)
                i++;
        }
        C[i+1]->insert_non_full(k);
    }
}

void node::split_the_child(int i, node *y)
{
    node *z = new node(y->t, y->leaf);
    z->n = t - 1;
    for (int j = 0; j < t-1; j++)
        z->keys[j] = y->keys[j+t];
```

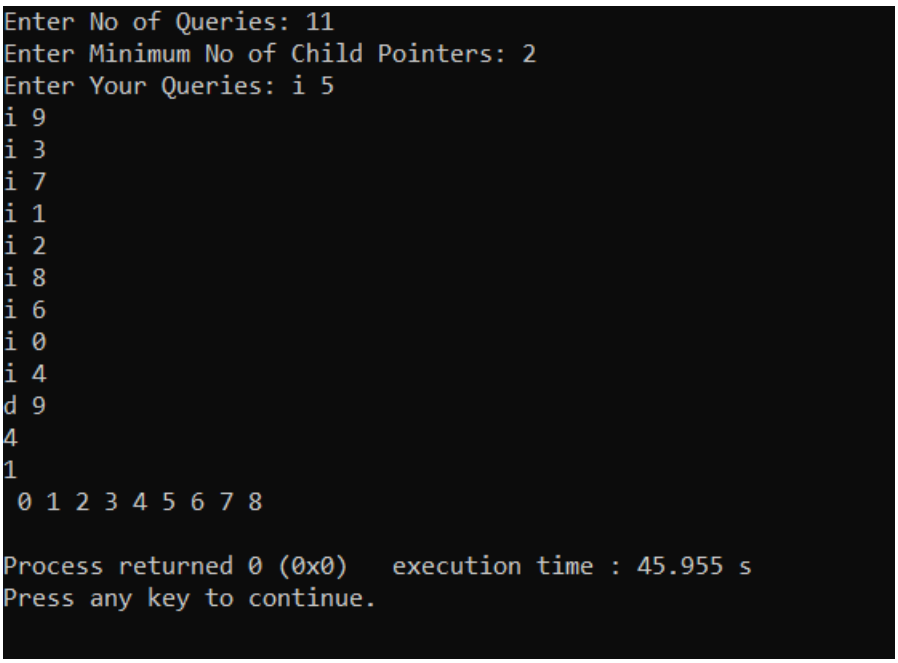
```
    if (y->leaf == false)
    {
        for (int j = 0; j < t; j++)
            z->C[j] = y->C[j+t];
    }
    y->n = t - 1;
    for (int j = n; j >= i+1; j--)
        C[j+1] = C[j];
    C[i+1] = z;
    for (int j = n-1; j >= i; j--)
        keys[j+1] = keys[j];
    keys[i] = y->keys[t-1];
    n = n + 1;
    spl++;
}

void node::inorder()
{
    int i;
    for (i = 0; i < n; i++)
    {
        if (leaf == false)
            C[i]->inorder();
        cout << " " << keys[i];
    }
    if (leaf == false)
        C[i]->inorder();
}

node *node::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);
}

void b_tree::delete_(int k)
{
    if (!root)
    {
        cout << "The B tree is empty"<<endl;
        return;
    }
    root->delete_(k);
    if (root->n==0)
    {
        node *t = root;
        if (root->leaf)
            root = NULL;
        else
            root = root->C[0];
        delete t;
    }
    return;
}
```

```
int main()
{
   int NQ,j=0,a,cp;
        char ch;
        cout<<"Enter No of Queries: ";
        cin>>NQ;
        cout<<"Enter Minimum No of Child Pointers: ";
        cin>>cp;
        b_tree bt(cp);
        node* root=NULL;
        cout<<"Enter Your Queries: ";
        for(j=0;j<NQ;j++)
        {
           cin>>ch;
           cin>>a;
           if(ch=='i')
           {
              bt.insert(a);
           }
           else
           {
              bt.delete_(a);
           }

        }
        cout<<spl<<endl;
        cout<<mrg<<endl;
   bt.inorder();
   cout<<endl;
```

```
    return 0;
}
```

Output:



2)

```
#include <bits/stdc++.h>
using namespace std;
struct Node{
    int value;
    Node *left, *right;
    bool rightThread;
};
Node *convert(Node *root)
{

    if (root == NULL)
        return NULL;
    if (root->left == NULL &&
        root->right == NULL)
        return root;
```

```
    if (root->left != NULL)
    {

        Node* a = convert(root->left);

        a->right = root;
        a->rightThread = true;
    }

    if (root->right == NULL)
        return root;

    return convert(root->right);
}

Node *leftmost(Node *root)
{
    while (root != NULL && root->left != NULL)
        root = root->left;
    return root;
}
void inorder(Node *root)
{
    if (root == NULL)
    return;
    Node *current = leftmost(root);

    while (current != NULL)
    {
        cout << current->value << " ";
```

```
        if (current->rightThread)
            current = current->right;
        else
            current = leftmost(current->right);
    }
}
Node *newNode(int value)
{
    Node *temp = new Node;
    temp->left = temp->right = NULL;
    temp->value = value;
    return temp;
}

int main()
{
    Node* root = newNode(15);
    root->left = newNode(25);
    root->right = newNode(35);
    root->left->left = newNode(45);
    root->left->right = newNode(55);
    root->right->left = newNode(65);
    root->right->right = newNode(75);

    convert(root);

    cout << "Inorder traversal of created threaded binary tree is \n";
    inorder(root);
    return 0;
}
```

Output:

```
Inorder traversal of created threaded binary tree is
45 25 55 15 35 75
Process returned 0 (0x0)   execution time : 4.472 s
Press any key to continue.
```

3)

```
#include <bits/stdc++.h>

using namespace std;
int MAX = 3;
class BPTree;
class Node {
  bool IS_LEAF;
  int *key, size;
  Node **ptr;
  friend class BPTree;

   public:
  Node();
};
class BPTree {
  Node *root;
  void insertInternal(int, Node *, Node *);
  void removeInternal(int, Node *, Node *);
  Node *findParent(Node *, Node *);
  public:
  BPTree();
  void search(int);
  void insert(int);
  void remove(int);
  void display(Node *);
```

```
  Node *getRoot();
};
Node::Node() {
  key = new int[MAX];
  ptr = new Node *[MAX + 1];
}
BPTree::BPTree() {
  root = NULL;
}
void BPTree::insert(int x) {
  if (root == NULL) {
    root = new Node;
    root->key[0] = x;
    root->IS_LEAF = true;
    root->size = 1;
  } else {
    Node *cursor = root;
    Node *parent;
    while (cursor->IS_LEAF == false) {
      parent = cursor;
      for (int i = 0; i < cursor->size; i++) {
        if (x < cursor->key[i]) {
          cursor = cursor->ptr[i];
          break;
        }
        if (i == cursor->size - 1) {
          cursor = cursor->ptr[i + 1];
          break;
        }
      }
```

```
}
if (cursor->size < MAX) {
  int i = 0;
  while (x > cursor->key[i] && i < cursor->size)
    i++;
  for (int j = cursor->size; j > i; j--) {
    cursor->key[j] = cursor->key[j - 1];
  }
  cursor->key[i] = x;
  cursor->size++;
  cursor->ptr[cursor->size] = cursor->ptr[cursor->size - 1];
  cursor->ptr[cursor->size - 1] = NULL;
} else {
  Node *newLeaf = new Node;
  int virtualNode[MAX + 1];
  for (int i = 0; i < MAX; i++) {
    virtualNode[i] = cursor->key[i];
  }
  int i = 0, j;
  while (x > virtualNode[i] && i < MAX)
    i++;
  for (int j = MAX + 1; j > i; j--) {
    virtualNode[j] = virtualNode[j - 1];
  }
  virtualNode[i] = x;
  newLeaf->IS_LEAF = true;
  cursor->size = (MAX + 1) / 2;
  newLeaf->size = MAX + 1 - (MAX + 1) / 2;
  cursor->ptr[cursor->size] = newLeaf;
  newLeaf->ptr[newLeaf->size] = cursor->ptr[MAX];
```

```
      cursor->ptr[MAX] = NULL;
      for (i = 0; i < cursor->size; i++) {
        cursor->key[i] = virtualNode[i];
      }
      for (i = 0, j = cursor->size; i < newLeaf->size; i++, j++) {
        newLeaf->key[i] = virtualNode[j];
      }
      if (cursor == root) {
        Node *newRoot = new Node;
        newRoot->key[0] = newLeaf->key[0];
        newRoot->ptr[0] = cursor;
        newRoot->ptr[1] = newLeaf;
        newRoot->IS_LEAF = false;
        newRoot->size = 1;
        root = newRoot;
      } else {
        insertInternal(newLeaf->key[0], parent, newLeaf);
      }
    }
  }
}
void BPTree::insertInternal(int x, Node *cursor, Node *child) {
  if (cursor->size < MAX) {
    int i = 0;
    while (x > cursor->key[i] && i < cursor->size)
      i++;
    for (int j = cursor->size; j > i; j--) {
      cursor->key[j] = cursor->key[j - 1];
    }
    for (int j = cursor->size + 1; j > i + 1; j--) {
```

```
    cursor->ptr[j] = cursor->ptr[j - 1];
  }
  cursor->key[i] = x;
  cursor->size++;
  cursor->ptr[i + 1] = child;
} else {
  Node *newInternal = new Node;
  int virtualKey[MAX + 1];
  Node *virtualPtr[MAX + 2];
  for (int i = 0; i < MAX; i++) {
    virtualKey[i] = cursor->key[i];
  }
  for (int i = 0; i < MAX + 1; i++) {
    virtualPtr[i] = cursor->ptr[i];
  }
  int i = 0, j;
  while (x > virtualKey[i] && i < MAX)
    i++;
  for (int j = MAX + 1; j > i; j--) {
    virtualKey[j] = virtualKey[j - 1];
  }
  virtualKey[i] = x;
  for (int j = MAX + 2; j > i + 1; j--) {
    virtualPtr[j] = virtualPtr[j - 1];
  }
  virtualPtr[i + 1] = child;
  newInternal->IS_LEAF = false;
  cursor->size = (MAX + 1) / 2;
  newInternal->size = MAX - (MAX + 1) / 2;
  for (i = 0, j = cursor->size + 1; i < newInternal->size; i++, j++) {
```

```
      newInternal->key[i] = virtualKey[j];
    }
    for (i = 0, j = cursor->size + 1; i < newInternal->size + 1; i++, j++) {
      newInternal->ptr[i] = virtualPtr[j];
    }
    if (cursor == root) {
      Node *newRoot = new Node;
      newRoot->key[0] = cursor->key[cursor->size];
      newRoot->ptr[0] = cursor;
      newRoot->ptr[1] = newInternal;
      newRoot->IS_LEAF = false;
      newRoot->size = 1;
      root = newRoot;
    } else {
      insertInternal(cursor->key[cursor->size], findParent(root, cursor), newInternal);
    }
  }
}
Node *BPTree::findParent(Node *cursor, Node *child) {
  Node *parent;
  if (cursor->IS_LEAF || (cursor->ptr[0])->IS_LEAF) {
    return NULL;
  }
  for (int i = 0; i < cursor->size + 1; i++) {
    if (cursor->ptr[i] == child) {
      parent = cursor;
      return parent;
    } else {
      parent = findParent(cursor->ptr[i], child);
      if (parent != NULL)
```

```
      return parent;
    }
  }
  return parent;
}
void BPTree::remove(int x) {
  if (root == NULL) {
    cout << "Tree empty\n";
  } else {
    Node *cursor = root;
    Node *parent;
    int leftSibling, rightSibling;
    while (cursor->IS_LEAF == false) {
      for (int i = 0; i < cursor->size; i++) {
        parent = cursor;
        leftSibling = i - 1;
        rightSibling = i + 1;
        if (x < cursor->key[i]) {
          cursor = cursor->ptr[i];
          break;
        }
        if (i == cursor->size - 1) {
          leftSibling = i;
          rightSibling = i + 2;
          cursor = cursor->ptr[i + 1];
          break;
        }
      }
    }
    bool found = false;
```

```
int pos;
for (pos = 0; pos < cursor->size; pos++) {
  if (cursor->key[pos] == x) {
    found = true;
    break;
  }
}
if (!found) {
  cout << "Not found\n";
  return;
}
for (int i = pos; i < cursor->size; i++) {
  cursor->key[i] = cursor->key[i + 1];
}
cursor->size--;
if (cursor == root) {
  for (int i = 0; i < MAX + 1; i++) {
    cursor->ptr[i] = NULL;
  }
  if (cursor->size == 0) {
    cout << "Tree died\n";
    delete[] cursor->key;
    delete[] cursor->ptr;
    delete cursor;
    root = NULL;
  }
  return;
}
cursor->ptr[cursor->size] = cursor->ptr[cursor->size + 1];
cursor->ptr[cursor->size + 1] = NULL;
```

```
if (cursor->size >= (MAX + 1) / 2) {
  return;
}
if (leftSibling >= 0) {
  Node *leftNode = parent->ptr[leftSibling];
  if (leftNode->size >= (MAX + 1) / 2 + 1) {
    for (int i = cursor->size; i > 0; i--) {
      cursor->key[i] = cursor->key[i - 1];
    }
    cursor->size++;
    cursor->ptr[cursor->size] = cursor->ptr[cursor->size - 1];
    cursor->ptr[cursor->size - 1] = NULL;
    cursor->key[0] = leftNode->key[leftNode->size - 1];
    leftNode->size--;
    leftNode->ptr[leftNode->size] = cursor;
    leftNode->ptr[leftNode->size + 1] = NULL;
    parent->key[leftSibling] = cursor->key[0];
    return;
  }
}
if (rightSibling <= parent->size) {
  Node *rightNode = parent->ptr[rightSibling];
  if (rightNode->size >= (MAX + 1) / 2 + 1) {
    cursor->size++;
    cursor->ptr[cursor->size] = cursor->ptr[cursor->size - 1];
    cursor->ptr[cursor->size - 1] = NULL;
    cursor->key[cursor->size - 1] = rightNode->key[0];
    rightNode->size--;
    rightNode->ptr[rightNode->size] = rightNode->ptr[rightNode->size + 1];
    rightNode->ptr[rightNode->size + 1] = NULL;
```

```
    for (int i = 0; i < rightNode->size; i++) {
      rightNode->key[i] = rightNode->key[i + 1];
    }
    parent->key[rightSibling - 1] = rightNode->key[0];
    return;
  }
}
if (leftSibling >= 0) {
  Node *leftNode = parent->ptr[leftSibling];
  for (int i = leftNode->size, j = 0; j < cursor->size; i++, j++) {
    leftNode->key[i] = cursor->key[j];
  }
  leftNode->ptr[leftNode->size] = NULL;
  leftNode->size += cursor->size;
  leftNode->ptr[leftNode->size] = cursor->ptr[cursor->size];
  removeInternal(parent->key[leftSibling], parent, cursor);
  delete[] cursor->key;
  delete[] cursor->ptr;
  delete cursor;
} else if (rightSibling <= parent->size) {
  Node *rightNode = parent->ptr[rightSibling];
  for (int i = cursor->size, j = 0; j < rightNode->size; i++, j++) {
    cursor->key[i] = rightNode->key[j];
  }
  cursor->ptr[cursor->size] = NULL;
  cursor->size += rightNode->size;
  cursor->ptr[cursor->size] = rightNode->ptr[rightNode->size];
  cout << "Merging two leaf nodes\n";
  removeInternal(parent->key[rightSibling - 1], parent, rightNode);
  delete[] rightNode->key;
```

```
      delete[] rightNode->ptr;
      delete rightNode;
    }
  }
}
void BPTree::removeInternal(int x, Node *cursor, Node *child) {
  if (cursor == root) {
    if (cursor->size == 1) {
      if (cursor->ptr[1] == child) {
        delete[] child->key;
        delete[] child->ptr;
        delete child;
        root = cursor->ptr[0];
        delete[] cursor->key;
        delete[] cursor->ptr;
        delete cursor;
        cout << "Changed root node\n";
        return;
      } else if (cursor->ptr[0] == child) {
        delete[] child->key;
        delete[] child->ptr;
        delete child;
        root = cursor->ptr[1];
        delete[] cursor->key;
        delete[] cursor->ptr;
        delete cursor;
        cout << "Changed root node\n";
        return;
      }
    }
```

```
}
int pos;
for (pos = 0; pos < cursor->size; pos++) {
  if (cursor->key[pos] == x) {
    break;
  }
}
for (int i = pos; i < cursor->size; i++) {
  cursor->key[i] = cursor->key[i + 1];
}
for (pos = 0; pos < cursor->size + 1; pos++) {
  if (cursor->ptr[pos] == child) {
    break;
  }
}
for (int i = pos; i < cursor->size + 1; i++) {
  cursor->ptr[i] = cursor->ptr[i + 1];
}
cursor->size--;
if (cursor->size >= (MAX + 1) / 2 - 1) {
  return;
}
if (cursor == root)
  return;
Node *parent = findParent(root, cursor);
int leftSibling, rightSibling;
for (pos = 0; pos < parent->size + 1; pos++) {
  if (parent->ptr[pos] == cursor) {
    leftSibling = pos - 1;
    rightSibling = pos + 1;
```

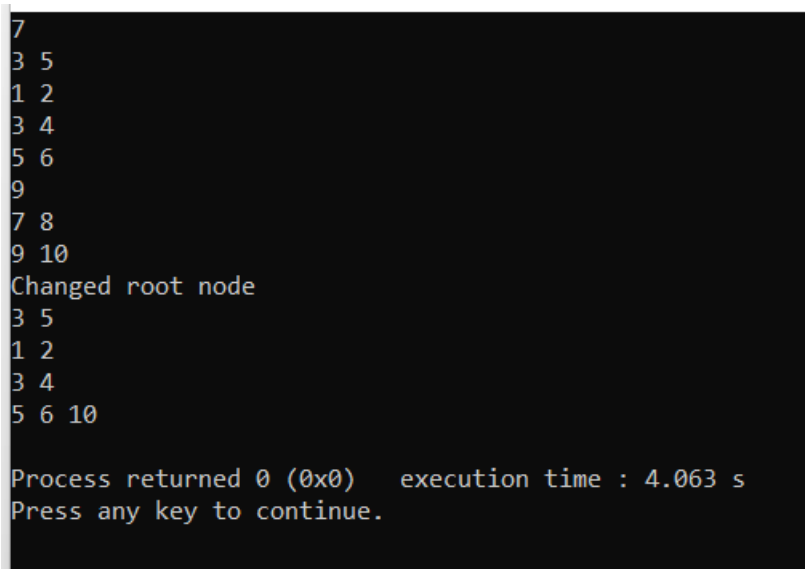
```
    break;
  }
}
if (leftSibling >= 0) {
  Node *leftNode = parent->ptr[leftSibling];
  if (leftNode->size >= (MAX + 1) / 2) {
    for (int i = cursor->size; i > 0; i--) {
      cursor->key[i] = cursor->key[i - 1];
    }
    cursor->key[0] = parent->key[leftSibling];
    parent->key[leftSibling] = leftNode->key[leftNode->size - 1];
    for (int i = cursor->size + 1; i > 0; i--) {
      cursor->ptr[i] = cursor->ptr[i - 1];
    }
    cursor->ptr[0] = leftNode->ptr[leftNode->size];
    cursor->size++;
    leftNode->size--;
    return;
  }
}
if (rightSibling <= parent->size) {
  Node *rightNode = parent->ptr[rightSibling];
  if (rightNode->size >= (MAX + 1) / 2) {
    cursor->key[cursor->size] = parent->key[pos];
    parent->key[pos] = rightNode->key[0];
    for (int i = 0; i < rightNode->size - 1; i++) {
      rightNode->key[i] = rightNode->key[i + 1];
    }
    cursor->ptr[cursor->size + 1] = rightNode->ptr[0];
    for (int i = 0; i < rightNode->size; ++i) {
```

```
      rightNode->ptr[i] = rightNode->ptr[i + 1];
    }
    cursor->size++;
    rightNode->size--;
    return;
  }
}
if (leftSibling >= 0) {
  Node *leftNode = parent->ptr[leftSibling];
  leftNode->key[leftNode->size] = parent->key[leftSibling];
  for (int i = leftNode->size + 1, j = 0; j < cursor->size; j++) {
    leftNode->key[i] = cursor->key[j];
  }
  for (int i = leftNode->size + 1, j = 0; j < cursor->size + 1; j++) {
    leftNode->ptr[i] = cursor->ptr[j];
    cursor->ptr[j] = NULL;
  }
  leftNode->size += cursor->size + 1;
  cursor->size = 0;
  removeInternal(parent->key[leftSibling], parent, cursor);
} else if (rightSibling <= parent->size) {
  Node *rightNode = parent->ptr[rightSibling];
  cursor->key[cursor->size] = parent->key[rightSibling - 1];
  for (int i = cursor->size + 1, j = 0; j < rightNode->size; j++) {
    cursor->key[i] = rightNode->key[j];
  }
  for (int i = cursor->size + 1, j = 0; j < rightNode->size + 1; j++) {
    cursor->ptr[i] = rightNode->ptr[j];
    rightNode->ptr[j] = NULL;
  }
```

```
    cursor->size += rightNode->size + 1;
    rightNode->size = 0;
    removeInternal(parent->key[rightSibling - 1], parent, rightNode);
  }
}
void BPTree::display(Node *cursor) {
  if (cursor != NULL) {
    for (int i = 0; i < cursor->size; i++) {
      cout << cursor->key[i] << " ";
    }
    cout << "\n";
    if (cursor->IS_LEAF != true) {
      for (int i = 0; i < cursor->size + 1; i++) {
        display(cursor->ptr[i]);
      }
    }
  }
}
Node *BPTree::getRoot() {
  return root;
}

int main() {
  BPTree node;
  node.insert(1);
  node.insert(3);
  node.insert(5);
  node.insert(7);
  node.insert(9);
  node.insert(2);
```

```
  node.insert(4);
  node.insert(6);
  node.insert(8);
  node.insert(10);
  node.display(node.getRoot());
  node.remove(9);
  node.remove(7);
  node.remove(8);

  node.display(node.getRoot());
}
```

Output:



4)

```
#include <bits/stdc++.h>

using namespace std;
int MAX = 3;
class BPTree;
class Node {
```

```
  bool IS_LEAF;
  int *key, size;
  Node **ptr;
  friend class BPTree;

   public:
  Node();
};
class BPTree {
  Node *root;
  void insertInternal(int, Node *, Node *);
  void removeInternal(int, Node *, Node *);
  Node *findParent(Node *, Node *);
  public:
  BPTree();
  void search(int);
  void insert(int);
  void remove(int);
  void display(Node *);
  Node *getRoot();
};
Node::Node() {
  key = new int[MAX];
  ptr = new Node *[MAX + 1];
}
BPTree::BPTree() {
  root = NULL;
}
void BPTree::search(int x) {
  if (root == NULL) {
```

```
    cout << "Tree is empty\n";
  } else {
    Node *cursor = root;
    while (cursor->IS_LEAF == false) {
      for (int i = 0; i < cursor->size; i++) {
        if (x < cursor->key[i]) {
          cursor = cursor->ptr[i];
          break;
        }
        if (i == cursor->size - 1) {
          cursor = cursor->ptr[i + 1];
          break;
        }
      }
    }
    for (int i = 0; i < cursor->size; i++) {
      if (cursor->key[i] == x) {
        cout << "Found\n";
        return;
      }
    }
    cout << "Not found\n";
  }
}
void BPTree::insert(int x) {
  if (root == NULL) {
    root = new Node;
    root->key[0] = x;
    root->IS_LEAF = true;
    root->size = 1;
```

```
} else {
  Node *cursor = root;
  Node *parent;
  while (cursor->IS_LEAF == false) {
    parent = cursor;
    for (int i = 0; i < cursor->size; i++) {
      if (x < cursor->key[i]) {
        cursor = cursor->ptr[i];
        break;
      }
      if (i == cursor->size - 1) {
        cursor = cursor->ptr[i + 1];
        break;
      }
    }
  }
  if (cursor->size < MAX) {
    int i = 0;
    while (x > cursor->key[i] && i < cursor->size)
      i++;
    for (int j = cursor->size; j > i; j--) {
      cursor->key[j] = cursor->key[j - 1];
    }
    cursor->key[i] = x;
    cursor->size++;
    cursor->ptr[cursor->size] = cursor->ptr[cursor->size - 1];
    cursor->ptr[cursor->size - 1] = NULL;
  } else {
    Node *newLeaf = new Node;
    int virtualNode[MAX + 1];
```

```
for (int i = 0; i < MAX; i++) {
  virtualNode[i] = cursor->key[i];
}
int i = 0, j;
while (x > virtualNode[i] && i < MAX)
  i++;
for (int j = MAX + 1; j > i; j--) {
  virtualNode[j] = virtualNode[j - 1];
}
virtualNode[i] = x;
newLeaf->IS_LEAF = true;
cursor->size = (MAX + 1) / 2;
newLeaf->size = MAX + 1 - (MAX + 1) / 2;
cursor->ptr[cursor->size] = newLeaf;
newLeaf->ptr[newLeaf->size] = cursor->ptr[MAX];
cursor->ptr[MAX] = NULL;
for (i = 0; i < cursor->size; i++) {
  cursor->key[i] = virtualNode[i];
}
for (i = 0, j = cursor->size; i < newLeaf->size; i++, j++) {
  newLeaf->key[i] = virtualNode[j];
}
if (cursor == root) {
  Node *newRoot = new Node;
  newRoot->key[0] = newLeaf->key[0];
  newRoot->ptr[0] = cursor;
  newRoot->ptr[1] = newLeaf;
  newRoot->IS_LEAF = false;
  newRoot->size = 1;
  root = newRoot;
```

```
      } else {
        insertInternal(newLeaf->key[0], parent, newLeaf);
      }
    }
  }
}
void BPTree::insertInternal(int x, Node *cursor, Node *child) {
  if (cursor->size < MAX) {
    int i = 0;
    while (x > cursor->key[i] && i < cursor->size)
      i++;
    for (int j = cursor->size; j > i; j--) {
      cursor->key[j] = cursor->key[j - 1];
    }
    for (int j = cursor->size + 1; j > i + 1; j--) {
      cursor->ptr[j] = cursor->ptr[j - 1];
    }
    cursor->key[i] = x;
    cursor->size++;
    cursor->ptr[i + 1] = child;
  } else {
    Node *newInternal = new Node;
    int virtualKey[MAX + 1];
    Node *virtualPtr[MAX + 2];
    for (int i = 0; i < MAX; i++) {
      virtualKey[i] = cursor->key[i];
    }
    for (int i = 0; i < MAX + 1; i++) {
      virtualPtr[i] = cursor->ptr[i];
    }
```

```
int i = 0, j;
while (x > virtualKey[i] && i < MAX)
  i++;
for (int j = MAX + 1; j > i; j--) {
  virtualKey[j] = virtualKey[j - 1];
}
virtualKey[i] = x;
for (int j = MAX + 2; j > i + 1; j--) {
  virtualPtr[j] = virtualPtr[j - 1];
}
virtualPtr[i + 1] = child;
newInternal->IS_LEAF = false;
cursor->size = (MAX + 1) / 2;
newInternal->size = MAX - (MAX + 1) / 2;
for (i = 0, j = cursor->size + 1; i < newInternal->size; i++, j++) {
  newInternal->key[i] = virtualKey[j];
}
for (i = 0, j = cursor->size + 1; i < newInternal->size + 1; i++, j++) {
  newInternal->ptr[i] = virtualPtr[j];
}
if (cursor == root) {
  Node *newRoot = new Node;
  newRoot->key[0] = cursor->key[cursor->size];
  newRoot->ptr[0] = cursor;
  newRoot->ptr[1] = newInternal;
  newRoot->IS_LEAF = false;
  newRoot->size = 1;
  root = newRoot;
} else {
  insertInternal(cursor->key[cursor->size], findParent(root, cursor), newInternal);
```

```
    }
  }
}
Node *BPTree::findParent(Node *cursor, Node *child) {
  Node *parent;
  if (cursor->IS_LEAF || (cursor->ptr[0])->IS_LEAF) {
    return NULL;
  }
  for (int i = 0; i < cursor->size + 1; i++) {
    if (cursor->ptr[i] == child) {
      parent = cursor;
      return parent;
    } else {
      parent = findParent(cursor->ptr[i], child);
      if (parent != NULL)
        return parent;
    }
  }
  return parent;
}
void BPTree::remove(int x) {
  if (root == NULL) {
    cout << "Tree empty\n";
  } else {
    Node *cursor = root;
    Node *parent;
    int leftSibling, rightSibling;
    while (cursor->IS_LEAF == false) {
      for (int i = 0; i < cursor->size; i++) {
        parent = cursor;
```

```
    leftSibling = i - 1;
    rightSibling = i + 1;
    if (x < cursor->key[i]) {
      cursor = cursor->ptr[i];
      break;
    }
    if (i == cursor->size - 1) {
      leftSibling = i;
      rightSibling = i + 2;
      cursor = cursor->ptr[i + 1];
      break;
    }
  }
}
bool found = false;
int pos;
for (pos = 0; pos < cursor->size; pos++) {
  if (cursor->key[pos] == x) {
    found = true;
    break;
  }
}
if (!found) {
  cout << "Not found\n";
  return;
}
for (int i = pos; i < cursor->size; i++) {
  cursor->key[i] = cursor->key[i + 1];
}
cursor->size--;
```

```
if (cursor == root) {
  for (int i = 0; i < MAX + 1; i++) {
    cursor->ptr[i] = NULL;
  }
  if (cursor->size == 0) {
    cout << "Tree died\n";
    delete[] cursor->key;
    delete[] cursor->ptr;
    delete cursor;
    root = NULL;
  }
  return;
}
cursor->ptr[cursor->size] = cursor->ptr[cursor->size + 1];
cursor->ptr[cursor->size + 1] = NULL;
if (cursor->size >= (MAX + 1) / 2) {
  return;
}
if (leftSibling >= 0) {
  Node *leftNode = parent->ptr[leftSibling];
  if (leftNode->size >= (MAX + 1) / 2 + 1) {
    for (int i = cursor->size; i > 0; i--) {
      cursor->key[i] = cursor->key[i - 1];
    }
    cursor->size++;
    cursor->ptr[cursor->size] = cursor->ptr[cursor->size - 1];
    cursor->ptr[cursor->size - 1] = NULL;
    cursor->key[0] = leftNode->key[leftNode->size - 1];
    leftNode->size--;
    leftNode->ptr[leftNode->size] = cursor;
```

```
    leftNode->ptr[leftNode->size + 1] = NULL;
    parent->key[leftSibling] = cursor->key[0];
    return;
   }
  }
  if (rightSibling <= parent->size) {
   Node *rightNode = parent->ptr[rightSibling];
   if (rightNode->size >= (MAX + 1) / 2 + 1) {
    cursor->size++;
    cursor->ptr[cursor->size] = cursor->ptr[cursor->size - 1];
    cursor->ptr[cursor->size - 1] = NULL;
    cursor->key[cursor->size - 1] = rightNode->key[0];
    rightNode->size--;
    rightNode->ptr[rightNode->size] = rightNode->ptr[rightNode->size + 1];
    rightNode->ptr[rightNode->size + 1] = NULL;
    for (int i = 0; i < rightNode->size; i++) {
     rightNode->key[i] = rightNode->key[i + 1];
    }
    parent->key[rightSibling - 1] = rightNode->key[0];
    return;
   }
  }
  if (leftSibling >= 0) {
   Node *leftNode = parent->ptr[leftSibling];
   for (int i = leftNode->size, j = 0; j < cursor->size; i++, j++) {
    leftNode->key[i] = cursor->key[j];
   }
   leftNode->ptr[leftNode->size] = NULL;
   leftNode->size += cursor->size;
   leftNode->ptr[leftNode->size] = cursor->ptr[cursor->size];
```

```
      removeInternal(parent->key[leftSibling], parent, cursor);
      delete[] cursor->key;
      delete[] cursor->ptr;
      delete cursor;
    } else if (rightSibling <= parent->size) {
      Node *rightNode = parent->ptr[rightSibling];
      for (int i = cursor->size, j = 0; j < rightNode->size; i++, j++) {
        cursor->key[i] = rightNode->key[j];
      }
      cursor->ptr[cursor->size] = NULL;
      cursor->size += rightNode->size;
      cursor->ptr[cursor->size] = rightNode->ptr[rightNode->size];
      cout << "Merging two leaf nodes\n";
      removeInternal(parent->key[rightSibling - 1], parent, rightNode);
      delete[] rightNode->key;
      delete[] rightNode->ptr;
      delete rightNode;
    }
  }
}
void BPTree::removeInternal(int x, Node *cursor, Node *child) {
  if (cursor == root) {
    if (cursor->size == 1) {
      if (cursor->ptr[1] == child) {
        delete[] child->key;
        delete[] child->ptr;
        delete child;
        root = cursor->ptr[0];
        delete[] cursor->key;
        delete[] cursor->ptr;
```

```
      delete cursor;
      cout << "Changed root node\n";
      return;
    } else if (cursor->ptr[0] == child) {
      delete[] child->key;
      delete[] child->ptr;
      delete child;
      root = cursor->ptr[1];
      delete[] cursor->key;
      delete[] cursor->ptr;
      delete cursor;
      cout << "Changed root node\n";
      return;
    }
  }
}
int pos;
for (pos = 0; pos < cursor->size; pos++) {
  if (cursor->key[pos] == x) {
    break;
  }
}
for (int i = pos; i < cursor->size; i++) {
  cursor->key[i] = cursor->key[i + 1];
}
for (pos = 0; pos < cursor->size + 1; pos++) {
  if (cursor->ptr[pos] == child) {
    break;
  }
}
```

```
for (int i = pos; i < cursor->size + 1; i++) {
  cursor->ptr[i] = cursor->ptr[i + 1];
}
cursor->size--;
if (cursor->size >= (MAX + 1) / 2 - 1) {
  return;
}
if (cursor == root)
  return;
Node *parent = findParent(root, cursor);
int leftSibling, rightSibling;
for (pos = 0; pos < parent->size + 1; pos++) {
  if (parent->ptr[pos] == cursor) {
    leftSibling = pos - 1;
    rightSibling = pos + 1;
    break;
  }
}
if (leftSibling >= 0) {
  Node *leftNode = parent->ptr[leftSibling];
  if (leftNode->size >= (MAX + 1) / 2) {
    for (int i = cursor->size; i > 0; i--) {
      cursor->key[i] = cursor->key[i - 1];
    }
    cursor->key[0] = parent->key[leftSibling];
    parent->key[leftSibling] = leftNode->key[leftNode->size - 1];
    for (int i = cursor->size + 1; i > 0; i--) {
      cursor->ptr[i] = cursor->ptr[i - 1];
    }
    cursor->ptr[0] = leftNode->ptr[leftNode->size];
```

```
    cursor->size++;
    leftNode->size--;
    return;
  }
}
if (rightSibling <= parent->size) {
  Node *rightNode = parent->ptr[rightSibling];
  if (rightNode->size >= (MAX + 1) / 2) {
    cursor->key[cursor->size] = parent->key[pos];
    parent->key[pos] = rightNode->key[0];
    for (int i = 0; i < rightNode->size - 1; i++) {
      rightNode->key[i] = rightNode->key[i + 1];
    }
    cursor->ptr[cursor->size + 1] = rightNode->ptr[0];
    for (int i = 0; i < rightNode->size; ++i) {
      rightNode->ptr[i] = rightNode->ptr[i + 1];
    }
    cursor->size++;
    rightNode->size--;
    return;
  }
}
if (leftSibling >= 0) {
  Node *leftNode = parent->ptr[leftSibling];
  leftNode->key[leftNode->size] = parent->key[leftSibling];
  for (int i = leftNode->size + 1, j = 0; j < cursor->size; j++) {
    leftNode->key[i] = cursor->key[j];
  }
  for (int i = leftNode->size + 1, j = 0; j < cursor->size + 1; j++) {
    leftNode->ptr[i] = cursor->ptr[j];
```

```
      cursor->ptr[j] = NULL;
    }
    leftNode->size += cursor->size + 1;
    cursor->size = 0;
    removeInternal(parent->key[leftSibling], parent, cursor);
  } else if (rightSibling <= parent->size) {
    Node *rightNode = parent->ptr[rightSibling];
    cursor->key[cursor->size] = parent->key[rightSibling - 1];
    for (int i = cursor->size + 1, j = 0; j < rightNode->size; j++) {
      cursor->key[i] = rightNode->key[j];
    }
    for (int i = cursor->size + 1, j = 0; j < rightNode->size + 1; j++) {
      cursor->ptr[i] = rightNode->ptr[j];
      rightNode->ptr[j] = NULL;
    }
    cursor->size += rightNode->size + 1;
    rightNode->size = 0;
    removeInternal(parent->key[rightSibling - 1], parent, rightNode);
  }
}
void BPTree::display(Node *cursor) {
  if (cursor != NULL) {
    for (int i = 0; i < cursor->size; i++) {
      cout << cursor->key[i] << " ";
    }
    cout << "\n";
    if (cursor->IS_LEAF != true) {
      for (int i = 0; i < cursor->size + 1; i++) {
        display(cursor->ptr[i]);
      }
```

```
    }
  }
}
Node *BPTree::getRoot() {
  return root;
}

int main() {
  BPTree node;
  node.insert(25);
  node.insert(15);
  node.insert(35);
  node.insert(45);
  node.insert(5);
  node.insert(15);
  node.insert(20);
  node.insert(25);
  node.insert(30);
  node.insert(35);
  node.insert(40);
  node.insert(45);
  node.insert(55);
  cout<<"searching for key =45: ";
  node.search(45);
}
```

Output:

```
searching for key =45: Found

Process returned 0 (0x0)   execution time : 0.842 s
Press any key to continue.
```