National Institute of Technology,Trichy[TN]

Date - 11/4/2022

Logo

Description automatically generated A picture containing text, tree, sky, outdoor

Description automatically generated

# GROUP ASSIGNMENT 1

Course - Advanced DSA (CSPE43)

Faculty - Dr. Nithya B.

Topic - Rapid Processing of Range-Query

Group No - 5

Members - Shubham Kaushal (106120116)

(shubham271306@gmail.com)

Penikalapati Pavan (106120080)

(pavan9093536@gmail.com)

Branch - CSE (4th Semester)

Abstract

Searching & Updating are the most fundamental operations in many complex systems. However, the complexity of the search process would increase dramatically in high-dimensional space & for long size memory. Regular implementation of query processing is performing the operations using current 1Dimensional/multidimensional memory itself, which involves so many operations leading to highly slow process. K-dimensional (KD) tree, as a classical data structure, has been widely used in high-dimensional vital data search. Segment tree & Fenwick tree are so efficient in rapid range query processing for long size memory due to their logarithmic behavior in complexities, However, at present, common methods proposed for larger range query processing are either unstable or time-consuming. Here, we proposed implementation of advanced data structures to construct a balanced KD tree based on presorted results for multidimensional data & Segment tree-Fenwick tree building for unidimensional data. Compared with previous similar method, the new algorithm could reduce the complexity of the processing process (excluding the presorting-pre preparation process) from O (KNlog2N) level to O (Nlog2N) level in KD tree & O(n) to O(logn) in Segment tree-Fenwick tree, (K is the number of dimensions and N is the number of data). Hence, these advanced containers/data structures usage highly optimizes our regular solution, we can observe it by analyzing our time complexity graphs plotted for these operations, corresponding to different-different implementations.

1. Introduction :

1.1 Problem statement:

Processing search query & update query operation on large 1Dimentional and 2Dimentional data.

Processing subarray sum query & update query operation on given big list of data.

1.2 Why the problem needs to be solved:

Regular implementation of query processing is performing the operations using current unidimensional/multidimensional memory itself, which involves so many operations leadings to highly slow process. We need efficient implementation to process these queries in very less time.

1.3 Existing solutions:

Generally, we solve search query & update query operation on 1Dimentional and 2Dimentional data by doing operations using current unidimensional/multidimensional memory itself or using prefix sum approach, which is works properly for small size memory.

1.4 Limitations in these solutions:

These solutions cannot lead to logarithmic /constant complexity. These solutions are so time taking for large size memory. Prefix sum approach causes rapid subarray sum query, but update query is still slow in that implementation. Naive approach i.e. using current unidimensional/multidimensional memory itself also has linear complexity. That’s why there is a need of efficient implementation or advanced data structure for rapid processing of these operations.

1.5 Our Contribution

We came up with advanced implementations which are highly efficient in rapid processing of unidimensional/multidimensional data. By taking help of advanced data structures, we can highly reduce no of operations, just by using some extra memory for these implementations. We can use KD tree for implementing the algorithm, for rapid search query & update query operation on 1Dimentional and 2Dimentional data. We can use Segment-Fenwick tree for implementing the algorithm, for rapid processing of subarray sum query & update query operation on given big list of data.

Data Structure 1 : BST

2.1 Description :

A binary search tree, also known as ordered binary search tree, is a variation of rooted binary tree in which the nodes are arranged in an order. The nodes of the tree store a key (and optionally, an associated value), and each has two distinguished sub-trees, commonly denoted left and right. The tree additionally satisfies the binary search properties :

* The left subtree of a node contains only nodes with keys lesser than the node’s key.
* The right subtree of a node contains only nodes with keys greater than the node’s key.
* The left and right subtree each must also be a binary search tree.



Diagram

Description automatically generated

BST requires an order relation by which every node of the tree is comparable with every other node in the sense of total order. Binary search trees are also efficacious in sorting algorithms and search algorithms. However, the search complexity of a BST depends upon the order in which the nodes are inserted and deleted; since in worst case, successive operations in the binary search tree may lead to degeneracy and form a singly linked list (or "unbalanced tree") like structure, thus has the same worst-case complexity as a linked list. Binary search trees are also a fundamental data structure used in construction of abstract data structures such as sets, multisets, and associative arrays.

2.2 Real time applications :

1. Routing Tables

A routing table is used to link routers in a network. It is usually implemented with a tree data structure, which is a variation of a binary tree. The tree data structure will store the location of routers based on their IP addresses. Routers with similar addresses are grouped under a single subtree.

To find a router to which a packet must be forwarded, we need to traverse the tree using the prefix of the network address to which a packet must be sent. Afterward, the packet is forwarded to the router with the longest matching prefix of the destination address.

2. Decision Trees

Binary trees can also be used for classification purposes. A decision tree is a supervised machine learning algorithm. The binary tree data structure is used here to emulate the decision-making process.

A decision tree usually begins with a root node. The internal nodes are conditions or dataset features. Branches are decision rules while the leave nodes are the outcomes of the decision.

3. Expression Evaluation

Another useful application of binary trees is in expression evaluation. In mathematics, expressions are statements with operators and operands that evaluate a value. The leaves of the binary tree are the operands while the internal nodes are the operators.

The expression is evaluated by applying the operator(s) in the internal node to the operands in the leaves.

4. Sorting

Binary search trees, a variant of binary trees are used in the implementation of sorting algorithms to order items. A binary search tree is simply an ordered or sorted binary tree such that the value in the left child is less than the value in the parent node. At the same time, the values in the right node are greater than the value in the parent node.

To complete a sorting procedure, the items to be sorted are first inserted into a binary search tree. To retrieve the sorted items, the tree is traversed using in-order traversal.

5. Indices for Databases

In database indexing, B-trees are used to sort data for simplified searching, insertion, and deletion. It is important to note that a B-tree is not a binary tree, but can become one when it takes on the properties of a binary tree.

The database creates indices for each given record in the database. The B-tree then stores in its internal nodes, references to data records with the actual data records in its leaf nodes. This provides sequential access to data in the databases.

6. Data Compression

In data compression, Huffman coding is used to create a binary tree capable of compressing data. Data compression is the processing of encoding data to use fewer bits. Given a text to compress, Huffman coding builds a binary tree and inserts the encodings of characters in the nodes based on their frequency in the text.

The encoding for a character is obtained by traversing the tree from its root to the node. Frequently occurring characters will have a shorter path as compared to less occurring characters. This is done to reduce the number of bits for frequent characters and ensure maximum data compression.

2.3 Requirements :

The main disadvantage is that we should always implement a balanced binary search tree - AVL tree, Red-Black tree, Splay tree. Otherwise the cost of operations may not be logarithmic and degenerate into a linear search on an array.

2.4 How it is used to address the chosen Problem :

2.4.1 Explanation:

### Search Operation

Always initiate analyzing tree at the root node and then move further to either the right or left subtree of the root node depending upon the element to be located is either less or greater than the root.

### Insert Operation

This is a very straight forward operation. First, the root node is inserted, then the next value is compared with the root node. If the value is greater than root, it is added to the right subtree, and if it is lesser than the root, it is added to the left subtree.

### Delete Operations

For deleting a node from a BST, there are some cases, i.e. deleting a root or deleting a leaf node. Also, after deleting a root, we need to think about the root node.

2.4.2 Algorithm:

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

*#include* <sys/time.h>

using namespace std;

long time() *// time in micro seconds*

{

   struct timeval start;

   gettimeofday(&start,NULL);

*return* (start.tv\_sec\*1000000 + start.tv\_usec);

}

*// BST(Binary Search Tree) =  sorted Binary Tree, inspired by Binary Search*

template <typename T>

class BinaryTreeNode                                       *// class for creating a node of BST*

{      public:

    T data;

    BinaryTreeNode \*left;

    BinaryTreeNode \*right;

    BinaryTreeNode(T d)

    {  data = d;

       left = NULL;

       right = NULL;   }

};

class BST                                                  *// class for creating a BST*

{

    BinaryTreeNode<int> \*root;

       public:

    BST()

    {  root = NULL;  }

       private:

    BinaryTreeNode<int>\* remove(int x, BinaryTreeNode<int> \*pp)

    {

*if*(pp==NULL)

*return* pp;

*if*(x < pp->data)

       pp->left = remove(x, pp->left);

*else* *if*(x > pp->data)

       pp->right = remove(x, pp->right);

*else                                                // removing/deleting root node of current BST*

       {  *if*(pp->left==NULL && pp->right==NULL)

          {  delete pp;

             pp = NULL;  }

*else* *if*(pp->left==NULL)

          {  BinaryTreeNode<int> \*temp = pp->right;

             pp->right = NULL;

             delete pp;

             pp = temp;  }

*else* *if*(pp->right==NULL)

          {  BinaryTreeNode<int> \*temp = pp->left;

             pp->left = NULL;

             delete pp;

             pp = temp;  }

*else*

          {  BinaryTreeNode<int> \*temp = pp->right;

*while*(temp->left != NULL)

             temp = temp->left;

             int rightBST\_min = temp->data;

             pp->data = rightBST\_min; *// making min element of right child BST as new root*

             pp->right = remove(rightBST\_min, pp->right);  }   }

*return* pp;

    }

    bool search(int x, BinaryTreeNode<int> \*pp)

    {

*if*(pp==NULL)

*return* 0;

*if*(pp->data == x)

*return* 1;

*else* *if*(x > pp->data)

*return* search(x, pp->right);

*if*(x < pp->data)

*return* search(x, pp->left);

*return* 0;

    }

    BinaryTreeNode<int>\* insert(int x, BinaryTreeNode<int> \*pp)

    {

*if*(pp==NULL)

       {  BinaryTreeNode<int> \*np = new BinaryTreeNode<int>(x);

*return* np;  }

*if*(x < pp->data)

       pp->left = insert(x, pp->left);

*else* *if*(x >= pp->data)

       pp->right = insert(x, pp->right);

*return* pp;

    }

        public: *// pp is not a input in these function, so create private helper functions, to apply recursion*

    void remove(int x)

    {  root = remove(x, root);  }

    bool search(int x)

    {  *return* search(x, root);  }

    void insert(int x)

    {  root = insert(x, root);  }

};

int main()

{

  int arr[10000];

*for*(int i=0; i<10000; i++)

  arr[i] = i;

*for*(int n=2000; n<=10000; n+=2000)

  {

    cout<<"\n\tWhen n="<<n<<":"<<endl;

    BST b;

*for*(int i=0; i<200; i++)

    b.insert(arr[i]); *// building BST*

    double startTime;

    double endTime;

    startTime = time();

*for*(int i=0; i<10000; i++) *// update function*

    {   b.insert(10000);

        b.remove(10000);   }

    endTime = time();

    cout<<"Time taken by update function: "<< (endTime - startTime)/10000 <<" microseconds\n";

    startTime = time();

*for*(int i=0; i<10000; i++)

    bool t = b.search(10000); *// search query function*

    endTime = time();

    cout<<"Time taken by search query function: "<< (endTime - startTime)/10000 <<" microseconds\n";

  }

}

2.4.3 Example:

### Search Operation

Always initiate analyzing tree at the root node and then move further to either the right or left subtree of the root node depending upon the element to be located is either less or greater than the root.

Diagram

Description automatically generated

1. The element to be searched is 10
2. Compare the element with the root node 12, 10 < 12, hence you move to the left subtree. No need to analyze the right-subtree
3. Now compare 10 with node 7, 10 > 7, so move to the right-subtree
4. Then compare 10 with the next node, which is 9, 10 > 9, look in the right subtree child
5. 10 matches with the value in the node, 10 = 10, return the value to the user.

### Insert Operation

This is a very straight forward operation. First, the root node is inserted, then the next value is compared with the root node. If the value is greater than root, it is added to the right subtree, and if it is lesser than the root, it is added to the left subtree.

Diagram

Description automatically generated

1. There is a list of 6 elements that need to be inserted in a BST in order from left to right
2. Insert 12 as the root node and compare next values 7 and 9 for inserting accordingly into the right and left subtree
3. Compare the remaining values 19, 5, and 10 with the root node 12 and place them accordingly. 19 > 12 place it as the right child of 12, 5 < 12 & 5 < 7, hence place it as left child of 7. Now compare 10, 10 is < 12 & 10 is > 7 & 10 is > 9, place 10 as right subtree of 9.

### Delete Operations

For deleting a node from a BST, there are some cases, i.e. deleting a root or deleting a leaf node. Also, after deleting a root, we need to think about the root node.

Say we want to delete a leaf node, we can just delete it, but if we want to delete a root, we need to replace the root’s value with another node. Let’s take the following example:

* Case 1- Node with zero children: this is the easiest situation, you just need to delete the node which has no further children on the right or left.
* Case 2 – Node with one child: once you delete the node, simply connect its child node with the parent node of the deleted value.
* Case 3 Node with two children: this is the most difficult situation, and it works on the following two rules
* 3a – In Order Predecessor: you need to delete the node with two children and replace it with the largest value on the left-subtree of the deleted node
* 3b – In Order Successor: you need to delete the node with two children and replace it with the largest value on the right-subtree of the deleted node

Diagram

Description automatically generated

1. This is the first case of deletion in which you delete a node that has no children. As you can see in the diagram that 19, 10 and 5 have no children. But we will delete 19.
2. Delete the value 19 and remove the link from the node.
3. View the new structure of the BST without 19

A screenshot of a computer

Description automatically generated with low confidence

1. This is the second case of deletion in which you delete a node that has 1 child, as you can see in the diagram that 9 has one child.
2. Delete the node 9 and replace it with its child 10 and add a link from 7 to 10
3. View the new structure of the BST without 9

Diagram

Description automatically generated

1. Here you will be deleting the node 12 that has two children
2. The deletion of the node will occur based upon the in order predecessor rule, which means that the largest element on the left subtree of 12 will replace it.
3. Delete the node 12 and replace it with 10 as it is the largest value on the left subtree
4. View the new structure of the BST after deleting 12

Diagram

Description automatically generated

1. 1 Delete a node 12 that has two children
2. 2 The deletion of the node will occur based upon the In Order Successor rule, which means that the largest element on the right subtree of 12 will replace it
3. 3 Delete the node 12 and replace it with 19 as it is the largest value on the right subtree

Data structure 2 : KD tree

3.1 Description :

A k-d tree (short for k-dimensional tree) is a space-partitioning data structure for organizing points in a k-dimensional space.

The k-d tree is a binary tree in which every node is a k-dimensional point. Every non-leaf node can be thought of as implicitly generating a splitting hyperplane that divides the space into two parts, known as half-spaces. Points to the left of this hyperplane are represented by the left subtree of that node and points to the right of the hyperplane are represented by the right subtree. The hyperplane direction is chosen in the following way: every node in the tree is associated with one of the k dimensions, with the hyperplane perpendicular to that dimension's axis. So, for example, if for a particular split the "x" axis is chosen, all points in the subtree with a smaller "x" value than the node will appear in the left subtree and all points with a larger "x" value will be in the right subtree. In such a case, the hyperplane would be set by the x value of the point, and its normal would be the unit x-axis.

Diagram, engineering drawing

Description automatically generated

Diagram

Description automatically generated

3.2 Real time applications:

**1. Nearest Neighbor Search**

k-d trees can help you find the nearest neighbor to a point on a two dimensional map of your city. All you have to do is construct a 2 dimensional k-d tree from the locations of all the police stations in your city, and then query the k-d tree to find the nearest police station to any given location in the city.

k-d trees help in partitioning space just as binary search trees help in partitioning the real line. k-d trees recursively partition a region of space, creating a binary space partition at each level of the tree.

**2. Database queries involving a multidimensional search key**

A query asking for all the employees in the age-group of (40, 50) and earning a salary in the range of (15000, 20000) per month can be transformed into a geometrical problem where the age is plotted along the x-axis and the salary is plotted along the y-axis

Chart, scatter chart

Description automatically generated

The x-axis denotes the age of the employee in *years*, and the y-axis denotes the monthly salary in *thousand rupees*.

A 2-dimensional k-d tree on the composite index of ***(age, salary)*** could help you efficiently search for all the employees that fall in the rectangular region of space defined by the query described above.

**3. n-body Problem**    
The naive method would involve computing the gravitational force between an object due to every other object in order to simulate its motion under gravitational attraction. Moreover, we would have to do it for every object which would take O(n^2) time.

Using k-d trees, however, we can partition the space and for each subdivision of space, figure out its total effect on the rest of space. Below is the pseudo code [6] of the algorithm.

**4. Colour Reduction**    
The naive method could be to pick up the colors which are used most often.

A more efficient method, however, could represent colors in terms of their ***RGB*** values and construct a 3 dimensional k-d tree in order to divide the space containing all the colors of the image. The construction of the k-d tree would stop when the count of the leaf nodes becomes equal to 256. The average of the RGB value of each of the 256 partitions could then be used to get a 256 color palette for the full color image.

K-D Trees are capable of guaranteeing a Log2(n) depth, where n is the number of points in the set.

Since this data structure takes place in a multi-dimensional space, this data structure is incredibly useful right now. Some modern applications of a K-D Tree could range from astrophysical simulation to computer graphics to even data compression. Thanks to being similar in performance to a Binary Search Tree, this data structure also works exceedingly fast.

3.3 Requirements :

*k*-d tree construction has the following constraints:

\*As one moves down the tree, one cycles through the axes used to select the splitting planes. (For example, in a 3-dimensional tree, the root would have an *x*-aligned plane, the root's children would both have *y*-aligned planes, the root's grandchildren would all have *z*-aligned planes, the root's great-grandchildren would all have *x*-aligned planes, the root's great-great-grandchildren would all have *y*-aligned planes, and so on.)

\*Points are inserted by selecting the median of the points being put into the subtree with respect to their coordinates in the axis being used to create the splitting plane. (Note the assumption that we feed the entire set of *n* points into the algorithm up-front.)

This method leads to a balanced  *k*-d tree, in which each leaf node is approximately the same distance from the root. However, balanced trees are not necessarily optimal for all applications.

3.4.1 Explanation :

The algorithms below consider the space to be 2 dimensional but can be applied to any space.

* **Search(x, y):** This function checks if the point exists in space. Start with root node as current node.
  1. If the current node represents the point *(x, y)*, return true.
  2. If current node is not a leaf node, goto step 3, otherwise return false.
  3. Let current node be the point *(X, Y)*. If the node divides space along x-axis, compare *x* with *X*. If *x < X*, set current node as left child, otherwise set current node as right child. If the node divided the space along y-axis, compare *y* and *Y*.  
     Goto step 1.
* **Insert(x, y):** Every insert operation divides the space. The algorithm here considers space to be 2-dimensional but is applicable in all dimensions:
  1. Search the tree for *(x, y)* until a leaf node is reached.
  2. If the tree is empty, add a new node as root representing the point *(x, y)*. Here, the space can be divided along any axis. Indicate the axis along which the space is divided and end insertion.
  3. Insert a new node where the point *(x, y)* should have existed and have it store *(x, y)*. If the parent divided the space along x-axis, have the point divide the space along y-axis, otherwise have it divide space along x-axis.

In case the tree is to be built from a given set of points, the strategy to follow is to find the median point with respect to space to be divided. Insert that point using above method and repeat to find children nodes.

{\displaystyle t\_{\text{worst}}=O\left(k\cdot n^{1-{\frac {1}{k}}}\right)}

3.4.2 Algorithm:

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

*#include* <sys/time.h>

using namespace std;

long time() *// time in micro seconds*

{

   struct timeval start;

   gettimeofday(&start,NULL);

*return* (start.tv\_sec\*1000000 + start.tv\_usec);

}

const int k = 2; *// 2D KD tree*

struct Node

{

    int point[k];

    Node \*left, \*right;

};

struct Node\* newNode(int arr[])

{

    struct Node\* temp = new Node;

*for* (int i=0; i<k; i++)

    temp->point[i] = arr[i];

    temp->left = temp->right = NULL;

*return* temp;

}

Node \*insertRec(Node \*root, int point[], unsigned depth)

{

*if* (root == NULL)

*return* newNode(point);

    unsigned cd = depth % k; *//   current dimension (cd) of comparison*

*if*(point[cd] < (root->point[cd]))

    root->left = insertRec(root->left, point, depth + 1);

*else*

    root->right = insertRec(root->right, point, depth + 1);

*return* root;

}

Node\* insert(Node \*root, int point[])

{

*return* insertRec(root, point, 0);

}

Node \*minNode(Node \*x, Node \*y, Node \*z, int d)

{

    Node \*res = x;

*if* (y != NULL && y->point[d] < res->point[d])

       res = y;

*if* (z != NULL && z->point[d] < res->point[d])

       res = z;

*return* res;

}

Node \*findMinRec(Node\* root, int d, unsigned depth)

{

*if*(root == NULL)

*return* NULL;

    unsigned cd = depth % k;

*if*(cd == d)

    {

*if*(root->left == NULL)

*return* root;

*else*

*return* findMinRec(root->left, d, depth+1);

    }

*return* minNode(root, findMinRec(root->left,d,depth+1), findMinRec(root->right, d, depth+1), d);

}

Node \*findMin(Node\* root, int d)

{

*return* findMinRec(root, d, 0);

}

bool arePointsSame(int point1[], int point2[])

{

*for*(int i=0; i<k; ++i)

    {

*if*(point1[i] != point2[i])

*return* false;

    }

*return* true;

}

void copyPoint(int p1[], int p2[])

{

*for*(int i=0; i<k; i++)

    p1[i] = p2[i];

}

Node \*deleteNodeRec(Node \*root, int point[], int depth)

{

*if*(root == NULL)

*return* NULL;

    int cd = depth % k;

*if* (arePointsSame(root->point, point))

    {

*if*(root->right != NULL)

        {

            Node \*min = findMin(root->right, cd);

            copyPoint(root->point, min->point);

            root->right = deleteNodeRec(root->right, min->point, depth+1);

        }

*else* *if*(root->left != NULL)

        {

            Node \*min = findMin(root->left, cd);

            copyPoint(root->point, min->point);

            root->right = deleteNodeRec(root->left, min->point, depth+1);

        }

*else*

        {

            delete root;

*return* NULL;

        }

*return* root;

    }

*if*(point[cd] < root->point[cd])

    root->left = deleteNodeRec(root->left, point, depth+1);

*else*

    root->right = deleteNodeRec(root->right, point, depth+1);

*return* root;

}

 Node\* deleteNode(Node \*root, int point[])

{

*return* deleteNodeRec(root, point, 0);

}

bool searchRec(Node\* root, int point[], unsigned depth) *//  depth is used to determine current axis*

{

*if*(root == NULL)

*return* false;

*else* *if*(arePointsSame(root->point, point))

*return* true;

    unsigned cd = depth % k;

*if*(point[cd] < root->point[cd])

*return* searchRec(root->left, point, depth + 1);

*else*

*return* searchRec(root->right, point, depth + 1);

}

bool search(Node\* root, int point[])

{

*return* searchRec(root, point, 0);

}

int main()

{

  int arr[10000][2];

*for*(int i=0; i<10000; i++)

  { arr[i][0] = i;

    arr[i][1] = i+1; }

*for*(int n=2000; n<=10000; n+=2000)

  {

    cout<<"\n\tWhen n="<<n<<":"<<endl;

    struct Node \*root = NULL;

*for*(int i=0; i<n; i++)

    root = insert(root, arr[i]); *// building Kd Tree*

    double startTime;

    double endTime;

    startTime = time();

*for*(int i=0; i<10000; i++) *// update function*

    {   root = insert(root, arr[500]);

        root = deleteNode(root, arr[500]);   }

    endTime = time();

    cout<<"Time taken by update function: "<< (endTime - startTime)/10000 <<" microseconds\n";

    startTime = time();

*for*(int i=0; i<10000; i++)

    bool t = search(root, arr[500]); *// search query function*

    endTime = time();

    cout<<"Time taken by search query function: "<< (endTime - startTime)/10000 <<" microseconds\n";

  }

}

3.4.3 Example:

**Search (14,1)**

Since this data structure is based on a Binary Search Tree, the search function acts similarly.

Diagram

Description automatically generated

Based on the above example, lets say we are searching for the point (14,1). We begin at the base node and compare the X values in the sets. Since 14 is greater than 7 we move to the next level and compare the Y values at the given data set. Since 1 is less than 3, we can move to the left child node. Since the X values are the same then we are able to compare Y values. As every element in the set is equal, we can confirm that this point is the correct point we have been searching for.

In the case that the value comparing is the same (lets say the X) and the other value is different (lets say the Y), we will move to the right node as that is the node we send duplicates.

Data structure 3: Segment tree

4.1 Description:

A Segment Tree is a data structure that allows answering range queries over an array effectively, while still being flexible enough to allow modifying the array. This includes finding the sum of consecutive array elements a[l…r], or finding the minimum element in a such a range in O(logn) time. Between answering such queries, the Segment Tree allows modifying the array by replacing one element, or even changing the elements of a whole subsegment (e.g. assigning all elements a[l…r] to any value, or adding a value to all element in the subsegment).

In general, a Segment Tree is a very flexible data structure, and a huge number of problems can be solved with it. Additionally, it is also possible to apply more complex operations and answer more complex queries. In particular the Segment Tree can be easily generalized to larger dimensions. For instance, with a two-dimensional Segment Tree you can answer sum or minimum queries over some sub rectangle of a given matrix in only O(logn\*logn) time.

One important property of Segment Trees is that they require only a linear amount of memory. The standard Segment Tree requires 4n vertices for working on an array of size n.

Here is a visual representation of such a Segment Tree over the array a=[1,3,−2,8,−7]:

Diagram

Description automatically generated

4.2 Real time Application:

Segment Tree is an important data structure for range query processing

* In its early days, the Segment Tree was used to efficiently list all pairs of intersecting rectangles from a list of rectangles in the plane.

We can use this method to report the list of all rectilinear line segments in the plane which intersect a query line segment.

We use this technique to report the perimeter of a set of rectangles in the plane.

* More recently, the segment tree has become popular for use in pattern recognition and image processing.
* Finding range sum/product, range max/min, prefix sum/product, etc
* Computational geometry
* Geographic information systems
* Static and Dynamic RMQ (Range Minimum Query)
* Storing segments in an arbitrary manner

4.3 Requirements:

\*Building a Segment tree requires O(N) time because there are N leaf nodes

in a segment tree and the remaining nodes are equal to N – 1. Therefore, the

total number of nodes = ( N ) + ( N – 1 ) = 2N – 1.

4.4.1 Explanation for Segment tree using Lazy Propagation:

\*Update Query:

First we find mid, then decide either to go left or right, that is, if the index where we have to update the value is greater than mid then we go right, else we go left. Even if the values at other nodes get affected by updating a single node, we only travel the whole height of the tree in the worst case scenario, hence, the time complexity for update operation is O(log(N)).

If the current segment tree node has any pending update, then first add that pending update to the current node. 2. If the current node’s range lies completely in the update query range: a. Update current node b. Postpone updates to children by setting lazy values for children nodes. 3. If the current node’s range overlaps with the update range, follow the same approach as above. a. Recur for left and right children. b. Update current node using results of left and right calls.

\*Sum Query:

The getSum() now first checks if there is a pending update and if there is, then updates the node. Once it makes sure that pending update is done, it works same as the previous getSum().

4.4.2 Code:

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

*#include* <sys/time.h>

using namespace std;

long time() *// time in micro seconds*

{

   struct timeval start;

   gettimeofday(&start,NULL);

*return* (start.tv\_sec\*1000000 + start.tv\_usec);

}

void buildLazySegmentTree(int \*arr, int \*ST, int rStart, int rEnd, int iCurrNode)

{

*if*(rStart == rEnd)

    {   ST[iCurrNode] = arr[rStart];

*return*;   }

    int rMid = (rStart+rEnd)/2;

    buildLazySegmentTree(arr, ST, rStart, rMid, 2\*iCurrNode);

    buildLazySegmentTree(arr, ST, rMid+1, rEnd, 2\*iCurrNode+1);

    ST[iCurrNode] = ST[2\*iCurrNode] + ST[2\*iCurrNode+1];

}

void rangeUpdate\_inLazySegmentTree(int \*ST, int \*PU, int rStart, int rEnd, int rLeft, int rRight, int x, int iCurrNode)

{

*if*(PU[iCurrNode] !=0) *// handlingPendingUpdates*

    {

        ST[iCurrNode] += (rEnd-rStart+1)\*PU[iCurrNode];

*if*(rStart != rEnd)

        {   PU[2\*iCurrNode] += PU[iCurrNode];

            PU[2\*iCurrNode+1] += PU[iCurrNode];   }

        PU[iCurrNode] = 0;

    }

*if*(rRight<rStart || rEnd<rLeft)

*return*;

*else* *if*(rLeft<=rStart  && rEnd<=rRight)

    {   ST[iCurrNode] += (rEnd-rStart+1)\*x; *// handlingCurrentUpdates*

*if*(rStart != rEnd) *// PU[iCurrNode] is already handled*

        {   PU[2\*iCurrNode] += x;

            PU[2\*iCurrNode+1] += x;   }

*return*;   }

    int rMid = (rStart+rEnd)/2;

    rangeUpdate\_inLazySegmentTree(ST, PU, rStart, rMid, rLeft, rRight, x, 2\*iCurrNode);

    rangeUpdate\_inLazySegmentTree(ST, PU, rMid+1, rEnd, rLeft, rRight, x, 2\*iCurrNode+1);

    ST[iCurrNode] = ST[2\*iCurrNode] + ST[2\*iCurrNode+1];

}

int rangeQuery\_inLazySegmentTree(int \*ST, int \*PU, int rStart, int rEnd, int rLeft, int rRight, int iCurrNode)

{

*if*(PU[iCurrNode] != 0) *// handlingPendingUpdates*

    {

        ST[iCurrNode] += (rEnd-rStart+1)\*PU[iCurrNode];

*if*(rStart != rEnd)

        {   PU[2\*iCurrNode] += PU[iCurrNode];

            PU[2\*iCurrNode+1] += PU[iCurrNode];   }

        PU[iCurrNode] = 0;

    }

*if*(rRight<rStart || rEnd<rLeft)

*return* 0;

*else* *if*(rLeft<=rStart  && rEnd<=rRight)

*return* ST[iCurrNode];

    int rMid=(rStart+rEnd)/2;

    int ans1 = rangeQuery\_inLazySegmentTree(ST, PU, rStart, rMid, rLeft, rRight, 2\*iCurrNode);

    int ans2 = rangeQuery\_inLazySegmentTree(ST, PU, rMid+1, rEnd, rLeft, rRight, 2\*iCurrNode+1);

    ST[iCurrNode] = ST[2\*iCurrNode] + ST[2\*iCurrNode+1]; *// weAreHandlingPendingUpdatesAlso soWeNeedToDoTheUpdation*

*return* (ans1 + ans2);

}

int main()

{

  int arr[50000];

*for*(int i=0; i<50000; i++)

  arr[i] = i;

*for*(int n=10000; n<=50000; n+=10000)

  {

    cout<<"\n\tWhen n="<<n<<":"<<endl;

    int ST[4\*n] = {0}; *// Segment Tree*

    int PU[4\*n] = {0}; *// Lazy Tree*

    buildLazySegmentTree(arr, ST, 0, n-1, 1); *// building Segment Tree*

    double startTime;

    double endTime;

    startTime = time();

*for*(int i=0; i<1000000; i++)

    rangeUpdate\_inLazySegmentTree(ST, PU, 0, n-1, 0, 0, 0, 1); *// increases all elements in [2,5] by 0*

    endTime = time();

    cout<<"Time taken by update function: "<< (endTime - startTime)/1000000 <<" microseconds\n";

    startTime = time();

*for*(int i=0; i<1000000; i++)

    int t = rangeQuery\_inLazySegmentTree(ST, PU, 0, n-1, 0, 0, 1) ; *// query to find sum in [2,6]*

    endTime = time();

    cout<<"Time taken by sum query function: "<< (endTime - startTime)/1000000 <<" microseconds\n";

  }

}

4.4.3 Example:

Example : A = [ 1, 2, 3, 4, 5, 6 ] , Query: Sum(2 to 4)

● At A[0, 5] : the value of this node doesn’t give us the answer to our query. Hence, we have to call recursion, now our query requires a sum 6 from 2 to 4 th index so it is partially inside A[0, 2] and A[3, 5]. Therefore, a recursion call is made to both the left and right subtree. ● At A[0, 2] : recursion call to A[0, 1] is not made because 0 and 1 are completely outside the range of our query and the answer of A[2, 2] which is completely inside the range of our query is returned using a recursive call. ● At A[3, 5] : recursion call to A[5, 5] is not made because 5 lies completely outside the range of our query while the answer to A[3, 4] which is completely inside the range of our query is returned using a recursive call.

A whiteboard with blue writing

Description automatically generated with low confidence

Example: A = [ 1, 3,-2,4 ] , Update (0,3) by 3

For the update function in [0,3] or 0th index of the segment tree, only the values at first nodes of lazy tree & segment tree need to be changed.

Diagram

Description automatically generated

Data structure 4: Fenwick tree

5.1 Description:

The idea is based on the fact that all positive integers can be represented as the sum of powers of 2. For example 19 can be represented as 16 + 2 + 1. Every node of the Fenwick tree (BIT tree) stores the sum of n elements where n is a power of 2. For example, in the first diagram above (the diagram for getSum()), the sum of the first 12 elements can be obtained by the sum of the last 4 elements (from 9 to 12) plus the sum of 8 elements (from 1 to 8). The number of set bits in the binary representation of a number n is O(Logn). Therefore, we traverse at-most O(Logn) nodes in both getSum() and update() operations. The time complexity of the construction is O(nLogn) as it calls update() for all n elements

Let, f be some *reversible* function and A be an array of integers of length N.

Fenwick tree is a data structure which:

* calculates the value of function f in the given range [l,r] (i.e. f(Al,Al+1,…,Ar)) in O(logn) time;
* updates the value of an element of A in O(logn) time;
* requires O(N) memory, or in other words, exactly the same memory required for A;
* is easy to use and code, especially, in the case of multidimensional arrays.

Fenwick tree is also called **Binary Indexed Tree**, or just **BIT** abbreviated.

The most common application of Fenwick tree is *calculating the sum of a range* (i.e. f(A1,A2,…,Ak)=A1+A2+⋯+Ak).

Diagram

Description automatically generated

5.2 Real time application:

\*It is used in rapid processing of Range Query problems in programming contest questions.

\*Binary Indexed trees are used to implement the arithmetic coding algorithm. Development of operations it supports were primarily motivated by use in that case.

\*Binary Indexed Tree can be used to count inversions in an array in time.

5.3 Requirements:

\* The first node is a dummy node which contains nothing. Thus, indexing actually

starts from 1. if an array has N elements, then the corresponding

binary indexed tree will have N + 1 nodes

\* The normal Fenwick tree can only answer sum queries of the type using sum(int r), however we can also answer other queries of the type by computing two sums and and subtract them. This is handled in the sum(int l, int r) method.

\*Its normal implementation supports two constructors. We can create a Fenwick tree initialized with zeros, or you can convert an existing array into the Fenwick form.

5.4.1 Explanation:

\*Sum Query

Example: Suppose we have to find the result of the query (0, 6). Since we have started our indexing from 1 we go to index 6 + 1 = 7 (22 + 21 + 20 ). If we subtract 20 from the above we get 22 + 21 (Index 6), which is the parent of index 7. Similarly, if we subtract 21 from the above result, we get 22 (Index 4), which is the parent of index 6. Thus, we can conclude that, by removing / subtracting the smallest power of 2 (or the rightmost set bit), we reach the parent of the current node.

\*Update Query

Suppose we have to update the value at index 2 in the array Arr = [1, 3, 5, 11, 7, 4, 6, 9]. This essentially means that index 3 of the fenwick tree will be updated. But there will also be updates at index 4 and index 8 because, since 4 (22 ) and 8 (23 ) are powers of 2, they will be the direct children of the 0th node and will contain results to the queries with start index 0 thus affected by a change in the index 2 of the array .

Let’s take a look at the code for building, updating and query on a fenwick tree.

5.4.2 Code:

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

*#include* <sys/time.h>

using namespace std;

long time() *// time in micro seconds*

{

   struct timeval start;

   gettimeofday(&start,NULL);

*return* (start.tv\_sec\*1000000 + start.tv\_usec);

}

void update(int \*BIT, int n, int i, int x) *// increase arr[i] by x*

{

*for*(int iCN=i+1 ; iCN<n+1; iCN+=(iCN&(-iCN))) *// leftShiftRightmostSetBitOf'iCN'(indexNoOfCurrentNode) toReachIt's/Ancestor'sRightSiblings*

    BIT[iCN] += x;

}

int query(int\* BIT, int i) *// sum of elements in [0,i]*

{

    int sum = 0;

*for*(int iCN=i+1;  iCN>0;  iCN-=(iCN&(-iCN))) *// removeRightmostSetBitOf'iCN'(indexNoOfCurrentNode) toReachIt'sParent/Ancestor*

    sum += BIT[iCN];

*return* sum;

}

int main() *// FenwickTree/BinaryIndexedTree/BIT (noNeedToImplementBuildTreeFunction butInitializeBITarrayByZero)*

{

  int arr[50000];

*for*(int i=0; i<50000; i++)

  arr[i] = i;

*for*(int n=10000; n<=50000; n+=10000)

  {

    cout<<"\n\tWhen n="<<n<<":"<<endl;

    int BIT[n+1]={0}; *// Fenwick Tree (BIT)*

*for*(int i=0; i<n; i++)

    update(BIT, n, i, (arr[i]-0)); *// building Fenwick Tree*

    double startTime;

    double endTime;

    startTime = time();

*for*(int i=0; i<1000000; i++)

    update(BIT, n, n, 0); *// update arr[2] by 11*

    endTime = time();

    cout<<"Time taken by update function: "<< (endTime - startTime)/1000000 <<" microseconds\n";

    startTime = time();

*for*(int i=0; i<1000000; i++)

    int t =  query(BIT,n) - query(BIT, n/2); *// query to find sum in [2,6]*

    endTime = time();

    cout<<"Time taken by sum query function: "<< (endTime - startTime)/1000000 <<" microseconds\n";

  }

}

5.4.3 Example:

Suppose we have an array, Arr = [1, 3, 5, 11, 7, 4, 6, 9] and we need to make a binary indexed tree from this.

Suppose we have to find the result of the query (0, 5). Since we have started our indexing from 1, we go to index 5 + 1 = 6 (22 + 21 ). Thus, the 6th node stores the answer to the query (4, 5), and its parent node which is at index 4 stores the answer to the query (0, 3). Hence our final answer / result will be the sum of results at index 6 and its parent index 4. So the final answer of the query (0, 5) = 11 + 20 = 31.

Diagram, schematic

Description automatically generated

6. Performance Comparison

6.1 Complexity analysis for Search query in 1D search

Chart, line chart

Description automatically generated

6.2 Complexity analysis for Update query in 1D search

Graphical user interface, chart, line chart

Description automatically generated

6.3 Complexity analysis for Search query in 2D search

Graphical user interface, chart, line chart

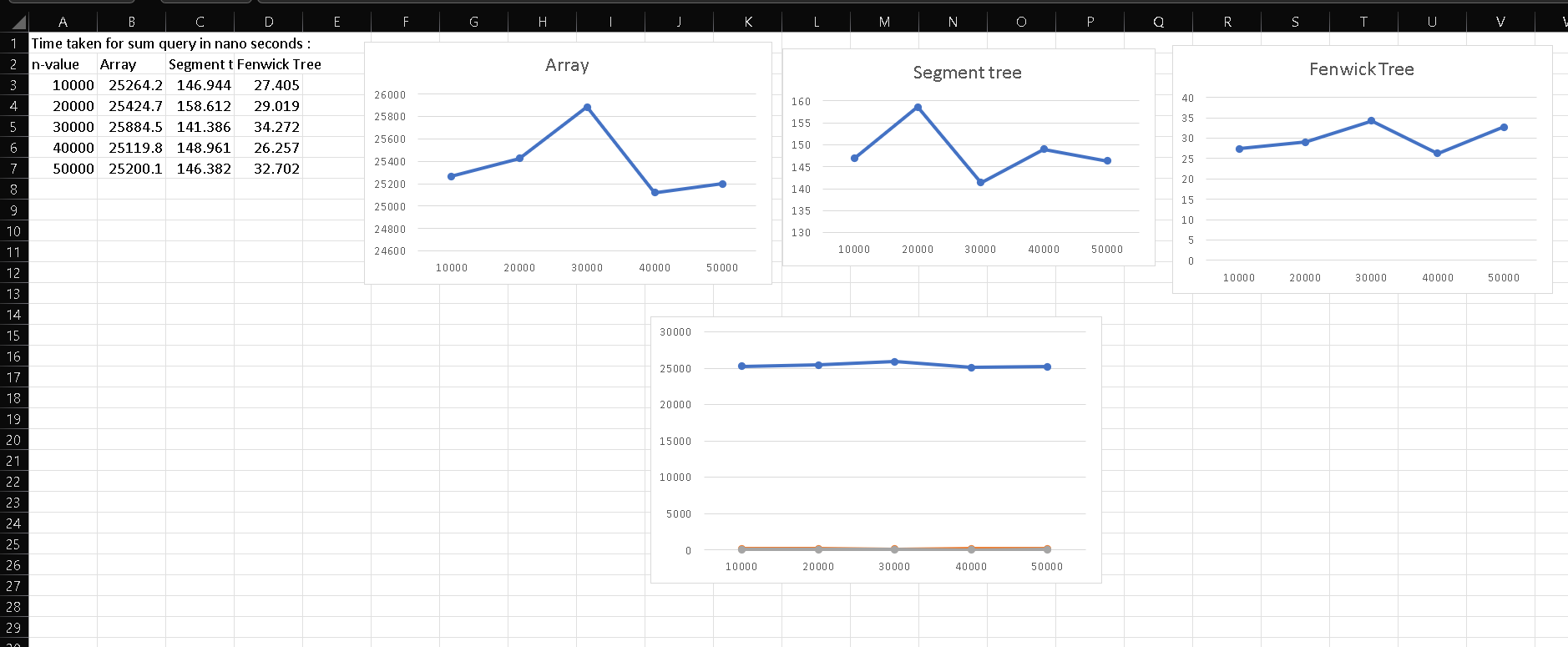
Description automatically generated

6.4 Complexity analysis for Update query in 2D search

Graphical user interface, chart, line chart

Description automatically generated

6.5 Complexity analysis for Sum query in Range sum



6.6 Complexity analysis for Update query in Range sum

Graphical user interface, chart, line chart

Description automatically generated

6.7 Summary table

\*Time Complexity analysis of 1D search & update query

|  |  |  |
| --- | --- | --- |
| Operation | 1D array | BST |
| Update | O(n) | O(logn) |
| Search | O(n) | O(logn) |

\*Time Complexity analysis of 2D search & update query

|  |  |  |
| --- | --- | --- |
| Operation | 2D array | KD Tree |
| Update | O(n) | O(logn) |
| Search | O(n) | O(logn) |

\*Time Complexity analysis of Range sum update & sum query

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Array | Segment Tree | Fenwick Tree |
| Update | O(n) | O(logn) | O(logn) |
| Sum | O(n) | O(logn) | O(logn) |

7. Conclusion:

For processing multiple range query operation on 1Dimentional or 2Dimentional data, it will be so time & resource consuming way to process it on the same memory. We can highly optimize our operations by using advanced data structures smartly in our program. Data structures like KD tree are so much effective in multidimensional (including 2Dimentional) search-update queries. Segment tree & Fenwick tree (Binary Indexed Tree) are capable to optimize our 1D range queries & updates in very less O(logn) time, while naïve approach takes O(n) time. BIT is proved to be 2–4 times faster in practice than Segment tree due to its simple iterative implementation. Due to the simple structure of the Binary Indexed Tree, it can be extended in multithreading and distributed environments obtaining O(log(logN)) time complexities per operations. KD tree is flexible enough to allow any intersection query. A noteworthy advantage of k-d trees is the fact that a single data structure facilitates many different and seemingly unrelated query types. Random insertion in an n node file is, on the average, an O(logn) task. Empirical tests show nearest neighbor searches in KD tree have average running time of O(logn). Deletion of a random node is O(logn) & optimization algorithm of speed O(nlogn) guarantees logarithmic i.e. optimized behavior of the tree.