```cpp
#include<bits/stdc++.h>
using namespace std;
int MAX; //size of each node
class BPTree; //self explanatory classes
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();
	void cleanUp(Node*);
	~BPTree();
};
//give command line argument to load a tree from log
//to create a fresh tree, do not give any command line argument
int main(int argc, char* argv[])
{
	BPTree bpt;//B+ tree object that carries out all the operations
	string command;
	int x;
	bool close = false;
	string logBuffer;//used to save into log
	ifstream fin;
	ofstream fout;
	//create tree from log file from command line input
	if(argc > 1)
	{
		fin.open(argv[1]);//open file
		if(!fin.is_open())
		{
			cout<<"File not found\n";
			return 0;
		}
		int i = 1;
		getline(fin, logBuffer, '\0');//copy log from file to logBuffer for saving purpose
		fin.close();
```

```
        fin.open(argv[1]);//reopening file
        getline(fin,command);
        stringstream max(command);//first line of log contains the max degree
        max>>MAX;
        while(getline(fin,command))//iterating over every line ie command
        {
            if(!command.substr(0,6).compare("insert"))
            {
                stringstream argument(command.substr(7));
                argument>>x;
                bpt.insert(x);
            }
            else if(!command.substr(0,6).compare("delete"))
            {
                stringstream argument(command.substr(7));
                argument>>x;
                bpt.remove(x);
            }
            else
            {
                cout<<"Unknown command: "<<command<<" at line #"<<i<<"\n";
                return 0;
            }
            i++;
        }
        cout<<"Tree loaded successfully from: \""<<argv[1]<<"\"\n";
        fin.close();
    }
    else//create fresh tree
    {
        cout<<"Enter the max degree\n";
        cin>>command;
        stringstream max(command);
        max>>MAX;
        logBuffer.append(command);
        logBuffer.append("\n");
        cin.clear();
        cin.ignore(1);
    }
    //command line menu
    cout<<"Commands:\nsearch <value> to search\n";
    cout<<"insert <value> to insert\n";
    cout<<"delete <value> to delete\n";
    cout<<"display to display\n";
    cout<<"save to save log\n";
    cout<<"exit to exit\n";
    do
    {
        cout<<"Enter command: ";
        getline(cin,command);
        if(!command.substr(0,6).compare("search"))
        {
                stringstream argument(command.substr(7));
                argument>>x;
                bpt.search(x);
            }
            else if(!command.substr(0,6).compare("insert"))
            {
                stringstream argument(command.substr(7));
                argument>>x;
                bpt.insert(x);
                logBuffer.append(command);
                logBuffer.append("\n");
            }
            else if(!command.substr(0,6).compare("delete"))
            {
                stringstream argument(command.substr(7));
                argument>>x;
                bpt.remove(x);
                logBuffer.append(command);
                logBuffer.append("\n");
            }
            else if(!command.compare("display"))
            {
                bpt.display(bpt.getRoot());
            }
            else if(!command.compare("save"))
            {
                cout<<"Enter file name: ";
                string filename;
                cin>>filename;
                fout.open(filename);
                fout<<logBuffer;
                fout.close();
                cout<<"Saved successfully into file: \""<<filename<<"\"\n";
                cin.clear();
                cin.ignore(1);
            }
            else if(!command.compare("exit"))
            {
                close = true;
            }
            else
            {
                cout<<"Invalid command\n";
            }
        }while(!close);
        return 0;
}
Node::Node()
{
        //dynamic memory allocation
        key = new int[MAX];
        ptr = new Node*[MAX+1];
```

```
}
BPTree::BPTree()
{
    root = NULL;
}
void BPTree::search(int x)
{
    //search logic
    if(root==NULL)
    {
        //empty
        cout<<"Tree empty\n";
    }
    else
    {
        Node* cursor = root;
        //in the following while loop, cursor will travel to the leaf node possibly consisting the key
        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;
                }
            }
        }
        //in the following for loop, we search for the key if it exists
        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)
{
    //insert logic
    if(root==NULL)
    {
        root = new Node;
        root->key[0] = x;
        root->IS_LEAF = true;
        root->size = 1;
        cout<<"Created root\nInserted "<<x<<" successfully\n";
    }
    else
    {
        Node* cursor = root;
        Node* parent;
        //in the following while loop, cursor will travel to the leaf node possibly consisting
the key
        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;
                }
            }
        }
        //now cursor is the leaf node in which we'll insert the new key
        if(cursor->size < MAX)
        {
            //if cursor is not full
            //find the correct position for new key
            int i = 0;
            while(x > cursor->key[i] && i < cursor->size) i++;
            //make space for new key
            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;
            cout<<"Inserted "<<x<<" successfully\n";
        }
        else
        {
            cout<<"Inserted "<<x<<" successfully\n";
            cout<<"Overflow in leaf node!\nSplitting leaf node\n";
            //overflow condition
            //create new leaf node
```

```
		Node* newLeaf = new Node;
		//create a virtual node and insert x into it
		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++;
		//make space for new key
		for(int j = MAX+1;j > i; j--)
		{
			virtualNode[j] = virtualNode[j-1];
		}
		virtualNode[i] = x;
		newLeaf->IS_LEAF = true;
		//split the cursor into two leaf nodes
		cursor->size = (MAX+1)/2;
		newLeaf->size = MAX+1-(MAX+1)/2;
		//make cursor point to new leaf node
		cursor->ptr[cursor->size] = newLeaf;
		//make new leaf node point to the next leaf node
		newLeaf->ptr[newLeaf->size] = cursor->ptr[MAX];
		cursor->ptr[MAX] = NULL;
		//now give elements to new leaf nodes
		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];
		}
		//modify the parent
		if(cursor == root)
		{
			//if cursor is a root node, we create a new 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;
			cout<<"Created new root\n";
		}
		else
		{
			//insert new key in parent node
			insertInternal(newLeaf->key[0],parent,newLeaf);
		}
	}
	}
}
void BPTree::insertInternal(int x, Node* cursor, Node* child)
{
	if(cursor->size < MAX)
	{
		//if cursor is not full
		//find the correct position for new key
		int i = 0;
		while(x > cursor->key[i] && i < cursor->size) i++;
		//make space for new key
		for(int j = cursor->size;j > i; j--)
		{
			cursor->key[j] = cursor->key[j-1];
		}//make space for new pointer
		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;
		cout<<"Inserted key in an Internal node successfully\n";
	}
	else
	{
		cout<<"Inserted key in an Internal node successfully\n";
		cout<<"Overflow in internal node!\nSplitting internal node\n";
		//if overflow in internal node
		//create new internal node
		Node* newInternal = new Node;
		//create virtual Internal 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++;
		//make space for new key
		for(int j = MAX+1;j > i; j--)
		{
			virtualKey[j] = virtualKey[j-1];
		}
		virtualKey[i] = x;
		//make space for new ptr
		for(int j = MAX+2;j > i+1; j--)
		{
			virtualPtr[j] = virtualPtr[j-1];
		}
		virtualPtr[i+1] = child;
		newInternal->IS_LEAF = false;
		//split cursor into two nodes
		cursor->size = (MAX+1)/2;
		newInternal->size = MAX-(MAX+1)/2;
		//give elements and pointers to the new node
		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];
		}
		// m = cursor->key[cursor->size]
		if(cursor == root)
		{
			//if cursor is a root node, we create a new 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;
			cout<<"Created new root\n";
		}
		else
		{
			//recursion
			//find depth first search to find parent of cursor
			insertInternal(cursor->key[cursor-
>size] ,findParent(root,cursor) ,newInternal);
		}
	}
}
Node* BPTree::findParent(Node* cursor, Node* child)
{
	//finds parent using depth first traversal and ignores leaf nodes as they cannot be parents
	//also ignores second last level because we will never find parent of a leaf node during
insertion using this function
	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)
{
    //delete logic
    if(root==NULL)
    {
        cout<<"Tree empty\n";
    }
    else
    {
        Node* cursor = root;
        Node* parent;
        int leftSibling, rightSibling;
        //in the following while loop, cursor will will travel to the leaf node possibly consisting the key
        while(cursor->IS_LEAF == false)
        {
            for(int i = 0; i < cursor->size; i++)
            {
                parent = cursor;
                leftSibling = i-1; //leftSibling is the index of left sibling in the parent node
                rightSibling =  i+1; //rightSibling is the index of right sibling in the parent node
                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;
                }
            }
        }
        //in the following for loop, we search for the key if it exists
        bool found = false;
        int pos;
```

```
for(pos = 0; pos < cursor->size; pos++)
{
    if(cursor->key[pos] == x)
    {
        found = true;
        break;
    }
}
if(!found)//if key does not exist in that leaf node
{
    cout<<"Not found\n";
    return;
}
//deleting the key
for(int i = pos; i < cursor->size; i++)
{
    cursor->key[i] = cursor->key[i+1];
}
cursor->size--;
if(cursor == root)//if it is root node, then make all pointers NULL
{
    cout<<"Deleted "<<x<<" from leaf node successfully\n";
    for(int i = 0; i < MAX+1; i++)
    {
        cursor->ptr[i] = NULL;
    }
    if(cursor->size == 0)//if all keys are deleted
    {
        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;
cout<<"Deleted "<<x<<" from leaf node successfully\n";
if(cursor->size >= (MAX+1)/2)//no underflow
{
    return;
}
cout<<"Underflow in leaf node!\n";
//underflow condition
//first we try to transfer a key from sibling node
//check if left sibling exists
if(leftSibling >= 0)
{
    Node *leftNode = parent->ptr[leftSibling];
    //check if it is possible to transfer
    if(leftNode->size >= (MAX+1)/2+1)
```

```
            {
                //make space for transfer
                for(int i = cursor->size; i > 0; i--)
                {
                    cursor->key[i] = cursor->key[i-1];
                }
                //shift pointer to next leaf
                cursor->size++;
                cursor->ptr[cursor->size] = cursor->ptr[cursor->size-1];
                cursor->ptr[cursor->size-1] = NULL;
                //transfer
                cursor->key[0] = leftNode->key[leftNode->size-1];
                //shift pointer of leftsibling
                leftNode->size--;
                leftNode->ptr[leftNode->size] = cursor;
                leftNode->ptr[leftNode->size+1] = NULL;
                //update parent
                parent->key[leftSibling] = cursor->key[0];
                cout<<"Transferred "<<cursor->key[0]<<" from left sibling of leaf
node\n";
                return;
            }
        }
        if(rightSibling <= parent->size)//check if right sibling exist
        {
            Node *rightNode = parent->ptr[rightSibling];
            //check if it is possible to transfer
            if(rightNode->size >= (MAX+1)/2+1)
            {
                //shift pointer to next leaf
                cursor->size++;
                cursor->ptr[cursor->size] = cursor->ptr[cursor->size-1];
                cursor->ptr[cursor->size-1] = NULL;
                //transfer
                cursor->key[cursor->size-1] = rightNode->key[0];
                //shift pointer of rightsibling
                rightNode->size--;
                rightNode->ptr[rightNode->size] = rightNode->ptr[rightNode-
>size+1];
                rightNode->ptr[rightNode->size+1] = NULL;
                //shift conent of right sibling
                for(int i = 0; i < rightNode->size; i++)
                {
                    rightNode->key[i] = rightNode->key[i+1];
                }
                //update parent
                parent->key[rightSibling-1] = rightNode->key[0];
                cout<<"Transferred "<<cursor->key[cursor->size-1]<<" from right
sibling of leaf node\n";
                return;
            }
        }
        //must merge and delete a node
        if(leftSibling >= 0)//if left sibling exist
        {
            Node* leftNode = parent->ptr[leftSibling];
            // transfer all keys to leftsibling and then transfer pointer to next leaf node
            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];
            cout<<"Merging two leaf nodes\n";
            removeInternal(parent->key[leftSibling],parent,cursor);// delete parent node key
            delete[] cursor->key;
            delete[] cursor->ptr;
            delete cursor;
        }
        else if(rightSibling <= parent->size)//if right sibling exist
        {
            Node* rightNode = parent->ptr[rightSibling];
            // transfer all keys to cursor and then transfer pointer to next leaf node
            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 parent node key
            delete[] rightNode->key;
            delete[] rightNode->ptr;
            delete rightNode;
        }
    }
}
void BPTree::removeInternal(int x, Node* cursor, Node* child)
{
    //deleting the key x first
    //checking if key from root is to be deleted
    if(cursor == root)
    {
        if(cursor->size == 1)//if only one key is left, change root
        {
            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];
}
//now deleting the pointer child
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)//no underflow
{
        cout<<"Deleted "<<x<<" from internal node successfully\n";
        return;
}
cout<<"Underflow in internal node!\n";
```

```
    //underflow, try to transfer first
    if(cursor==root)return;
    Node* parent = findParent(root, cursor);
    int leftSibling, rightSibling;
    //finding left n right sibling of cursor
    for(pos = 0; pos < parent->size+1; pos++)
    {
        if(parent->ptr[pos] == cursor)
        {
            leftSibling = pos - 1;
            rightSibling = pos + 1;
            break;
        }
    }
    //try to transfer
    if(leftSibling >= 0)//if left sibling exists
    {
        Node *leftNode = parent->ptr[leftSibling];
        //check if it is possible to transfer
        if(leftNode->size >= (MAX+1)/2)
        {
            //make space for transfer of key
            for(int i = cursor->size; i > 0; i--)
            {
                cursor->key[i] = cursor->key[i-1];
            }
            //transfer key from left sibling through parent
            cursor->key[0] = parent->key[leftSibling];
            parent->key[leftSibling] = leftNode->key[leftNode->size-1];
            //transfer last pointer from leftnode to cursor
            //make space for transfer of ptr
            for (int i = cursor->size+1; i > 0; i--)
            {
                cursor->ptr[i] = cursor->ptr[i-1];
            }
            //transfer ptr
            cursor->ptr[0] = leftNode->ptr[leftNode->size];
            cursor->size++;
            leftNode->size--;
            cout<<"Transferred "<<cursor->key[0]<<" from left sibling of internal node\
n";
            return;
        }
    }
    if(rightSibling <= parent->size)//check if right sibling exist
    {
        Node *rightNode = parent->ptr[rightSibling];
        //check if it is possible to transfer
        if(rightNode->size >= (MAX+1)/2)
        {
            //transfer key from right sibling through parent
            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];
            }
            //transfer first pointer from rightnode to cursor
            //transfer ptr
            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--;
            cout<<"Transferred "<<cursor->key[0]<<" from right sibling of internal
node\n";
            return;
        }
    }
    //transfer wasnt posssible hence do merging
    if(leftSibling >= 0)
    {
        //leftnode + parent key + cursor
        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;
        //delete cursor
        removeInternal(parent->key[leftSibling], parent, cursor);
        cout<<"Merged with left sibling\n";

    }
    else if(rightSibling <= parent->size)
    {
        //cursor + parent key + rightnode
        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;
        //delete cursor
        removeInternal(parent->key[rightSibling-1], parent, rightNode);
        cout<<"Merged with right sibling\n";
    }
}
void BPTree::display(Node* cursor)
{
    //depth first display
    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;
}
void BPTree::cleanUp(Node* cursor)
{
    //clean up logic
    if(cursor!=NULL)
    {
        if(cursor->IS_LEAF != true)
        {
            for(int i = 0; i < cursor->size+1; i++)
            {
                cleanUp(cursor->ptr[i]);
            }
        }
        for(int i = 0; i < cursor->size; i++)
        {
            cout<<"Deleted key from memory: "<<cursor->key[i]<<"\n";
        }
        delete[] cursor->key;
        delete[] cursor->ptr;
        delete cursor;
```

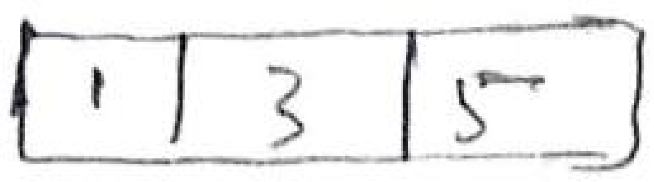
```
	}
}
BPTree::~BPTree()
{
	//calling cleanUp routine
	cleanUp(root);
}
```

Panasa Teja B191143CS

Insertion of 15 elements in B+ trees of order 4

1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29

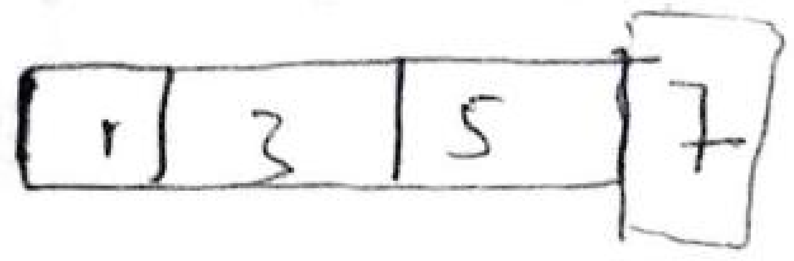
1, 3, 5 –



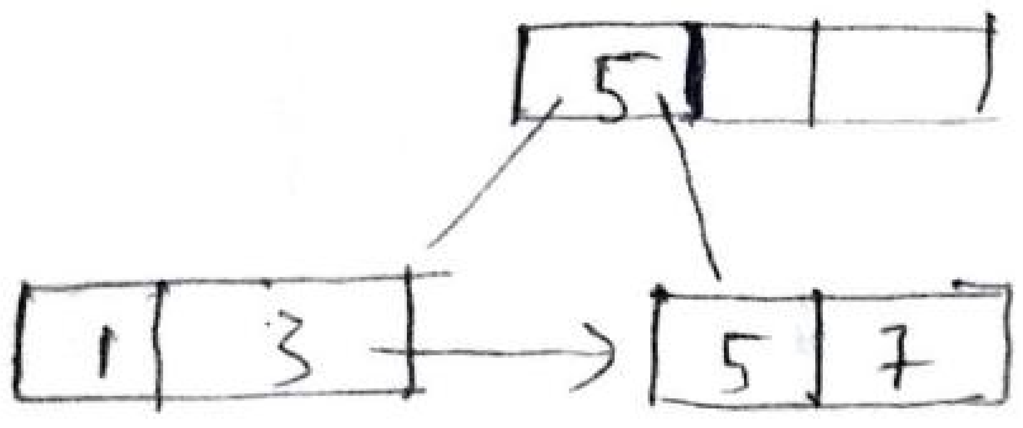
max (4 – 1) keys = 3

min $\lceil \frac{4}{2} \rceil - 1$ keys = 1

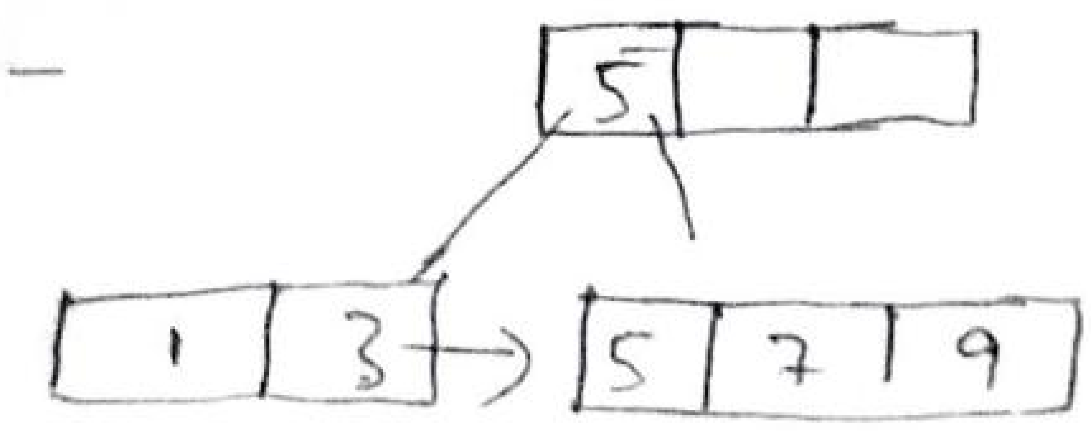
7 –



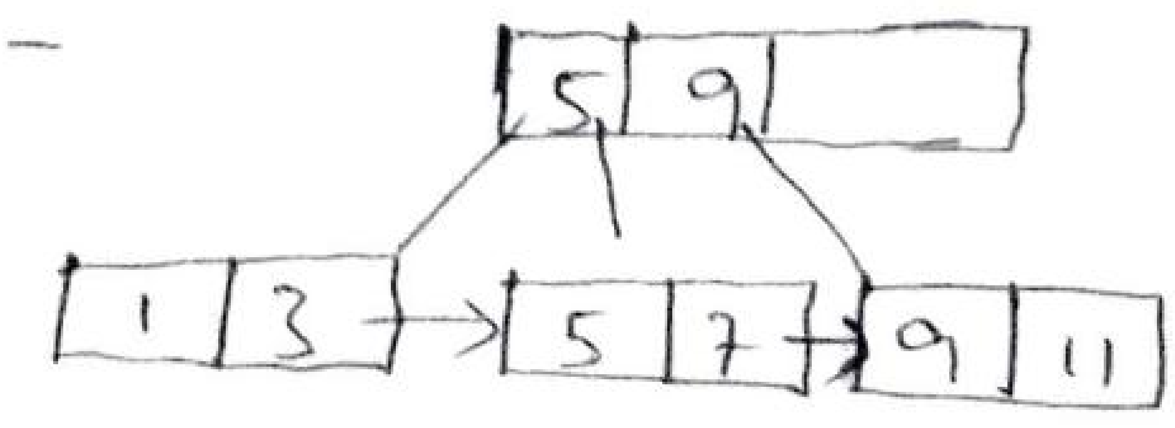
- Spilting will Occur



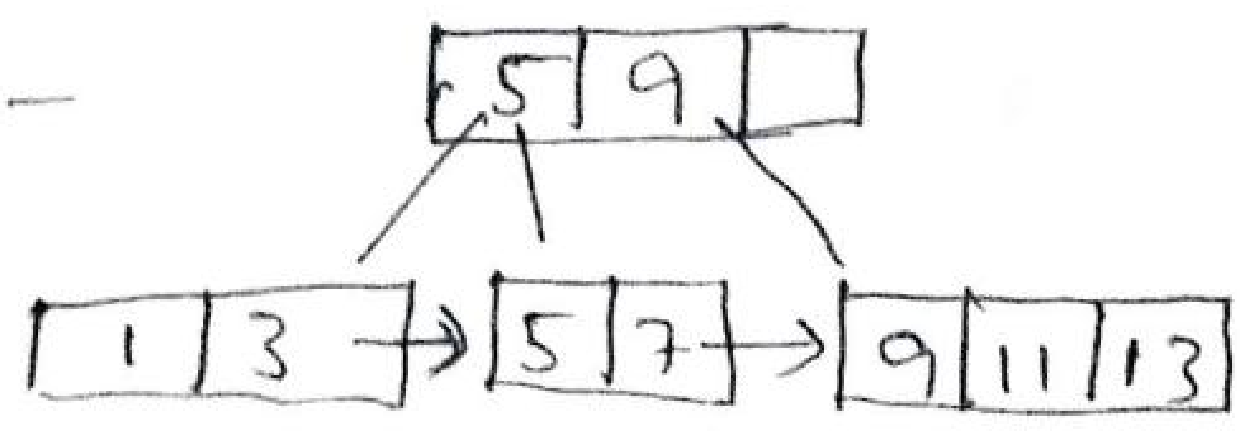
9 –



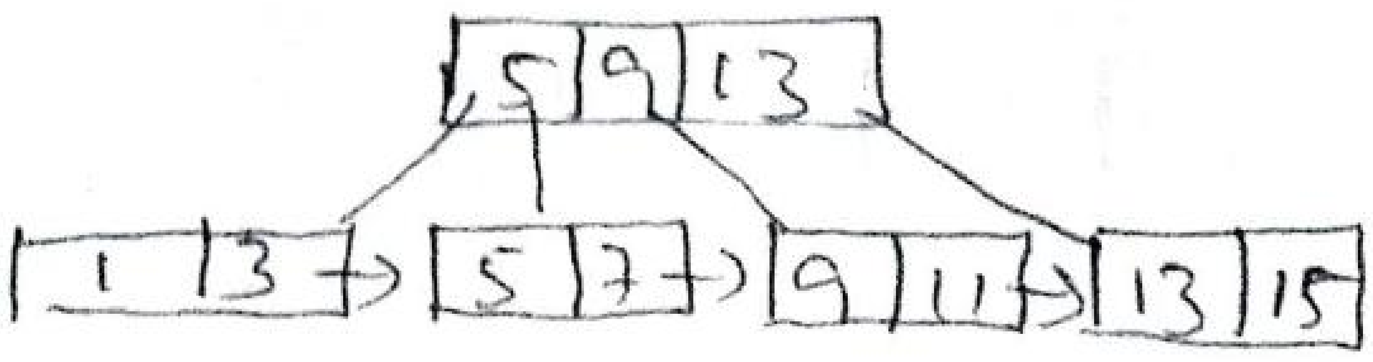
11 –



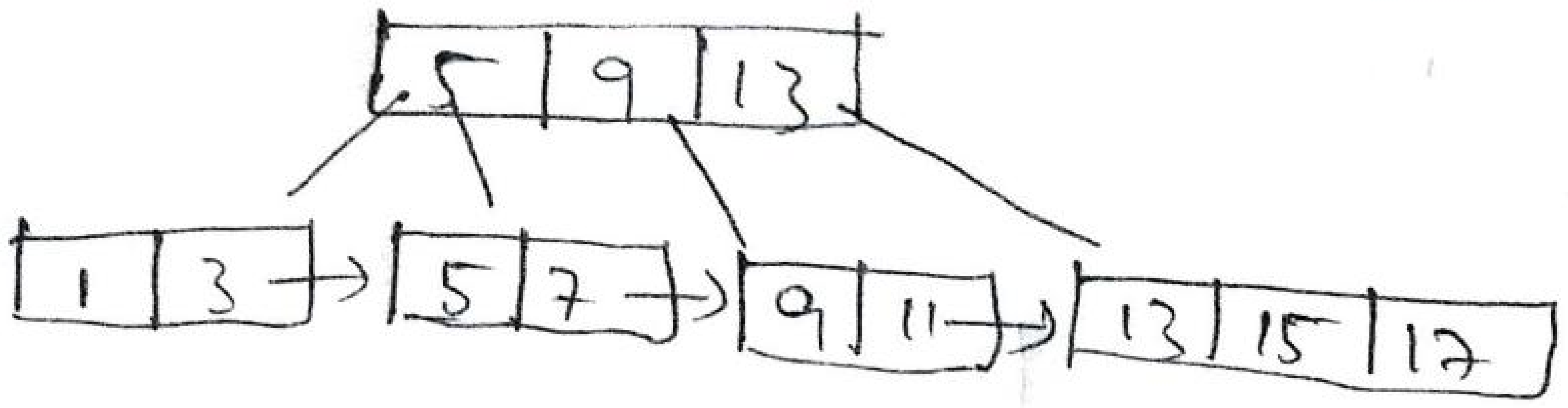
13 –



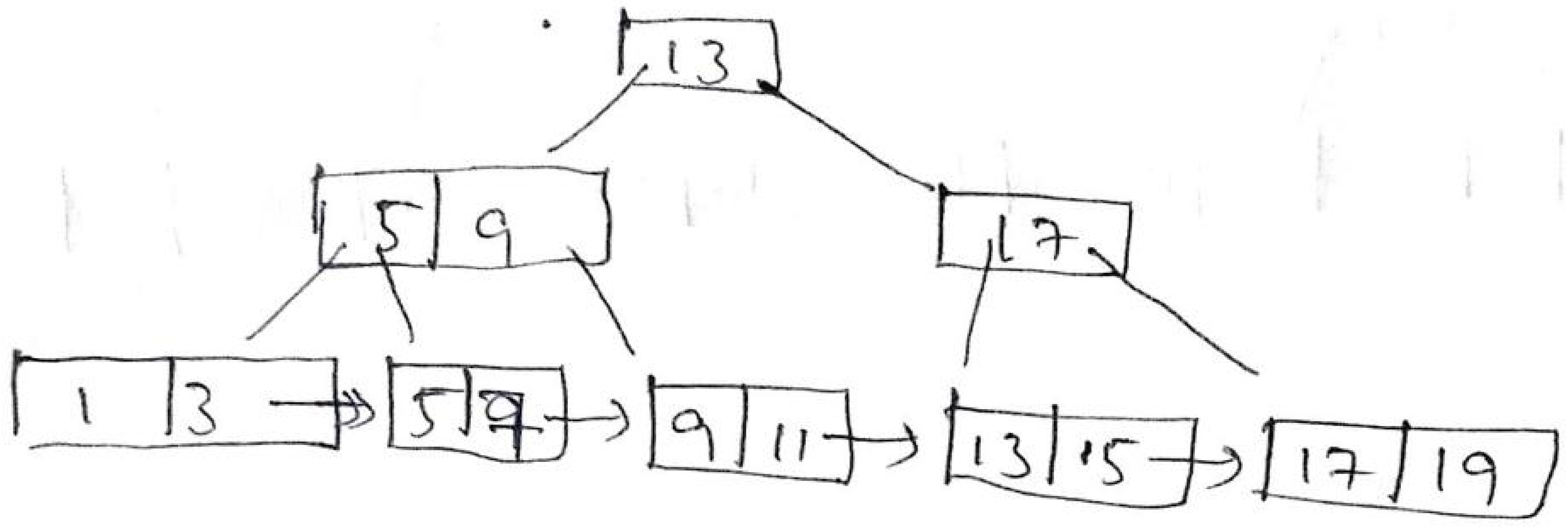
15 –



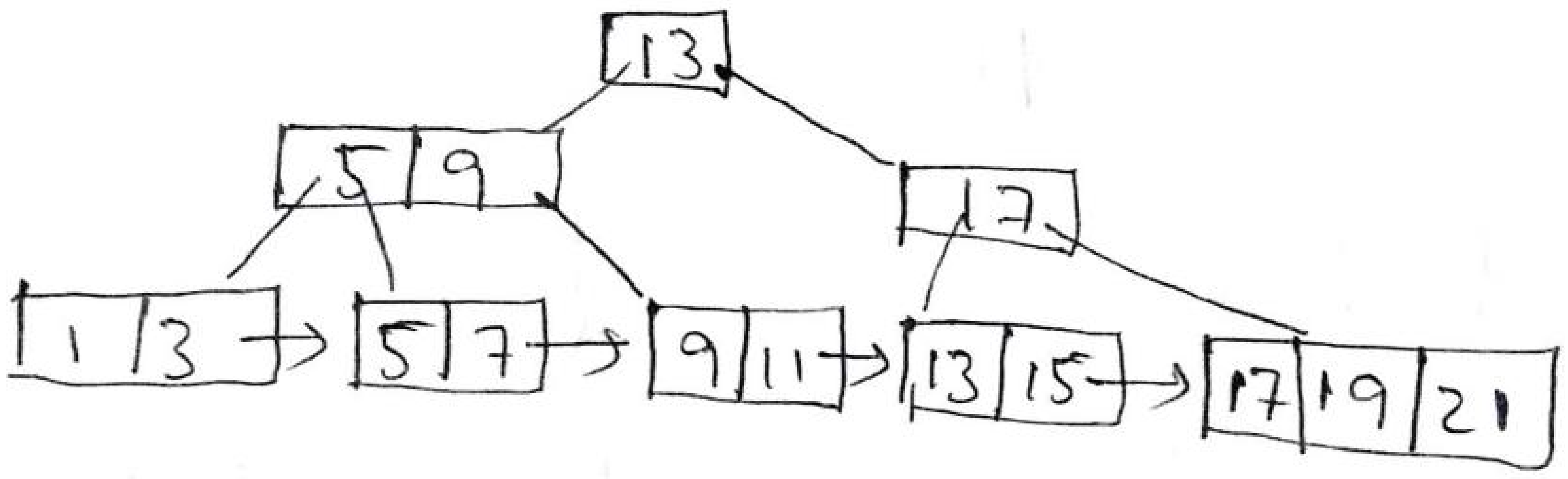
17 –



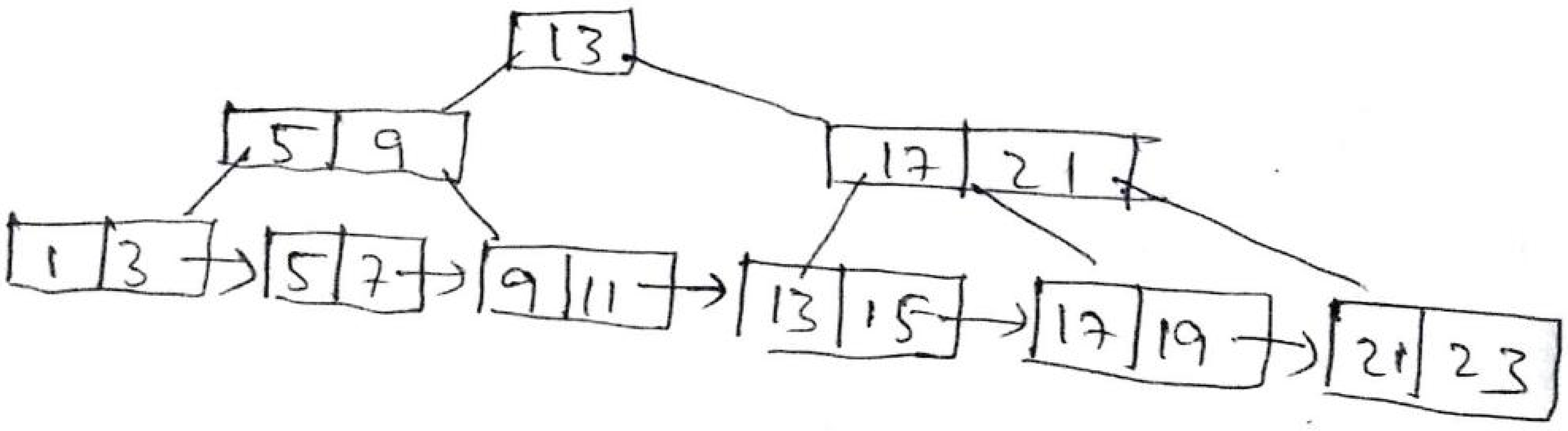
19 –



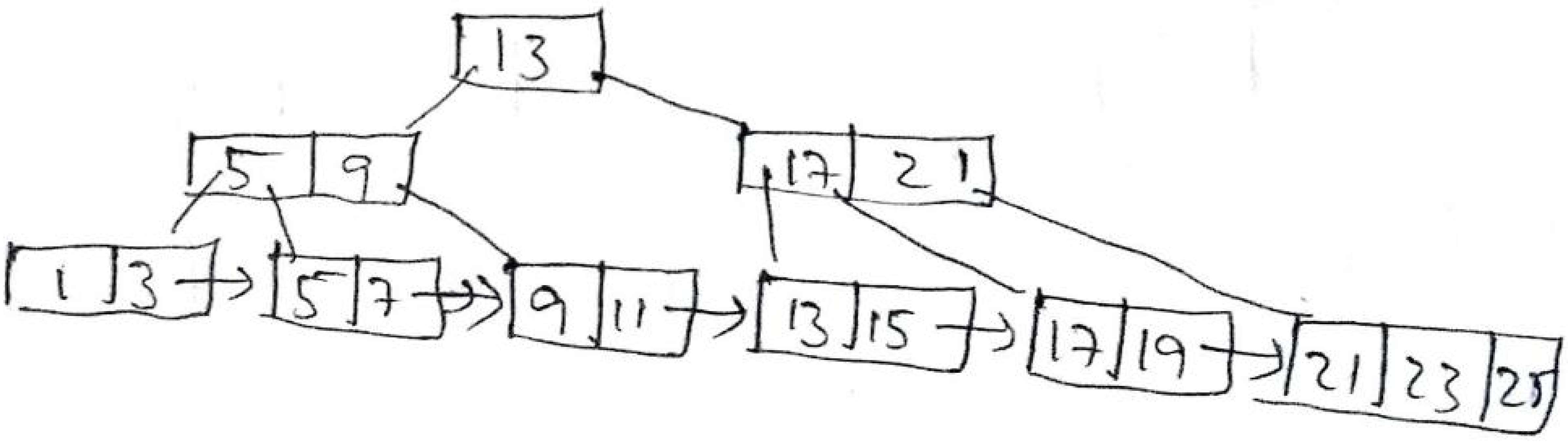
21 –



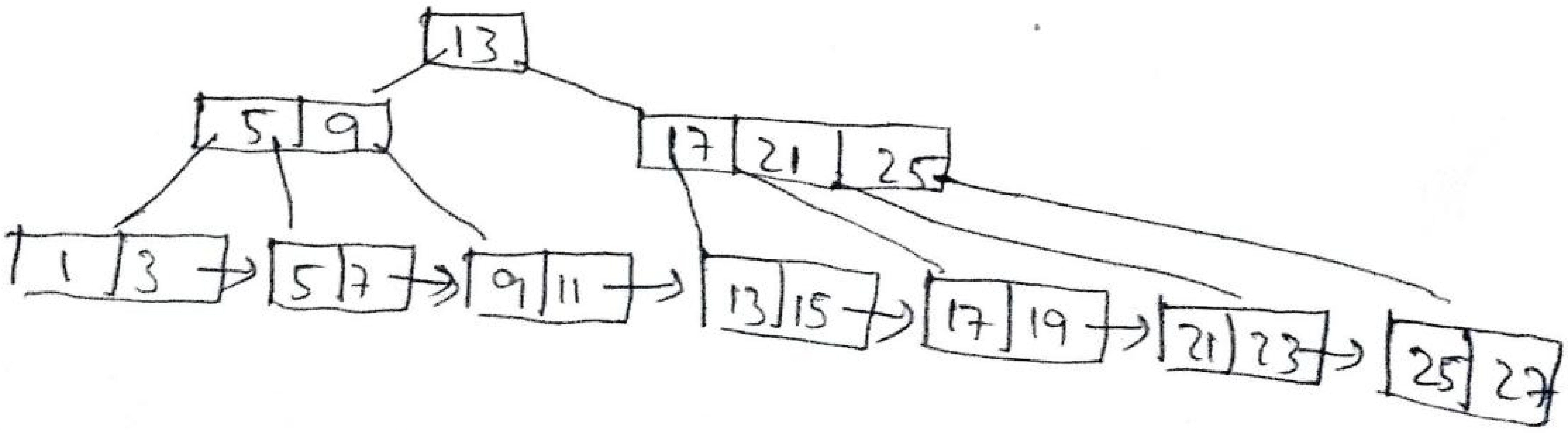
23 –



25 –

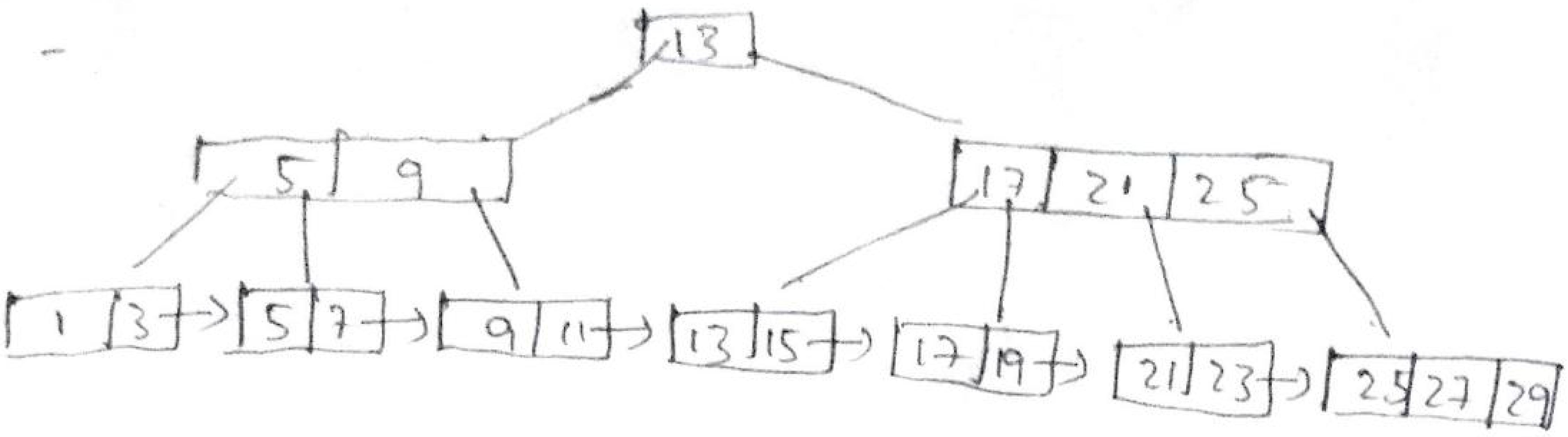


27 –

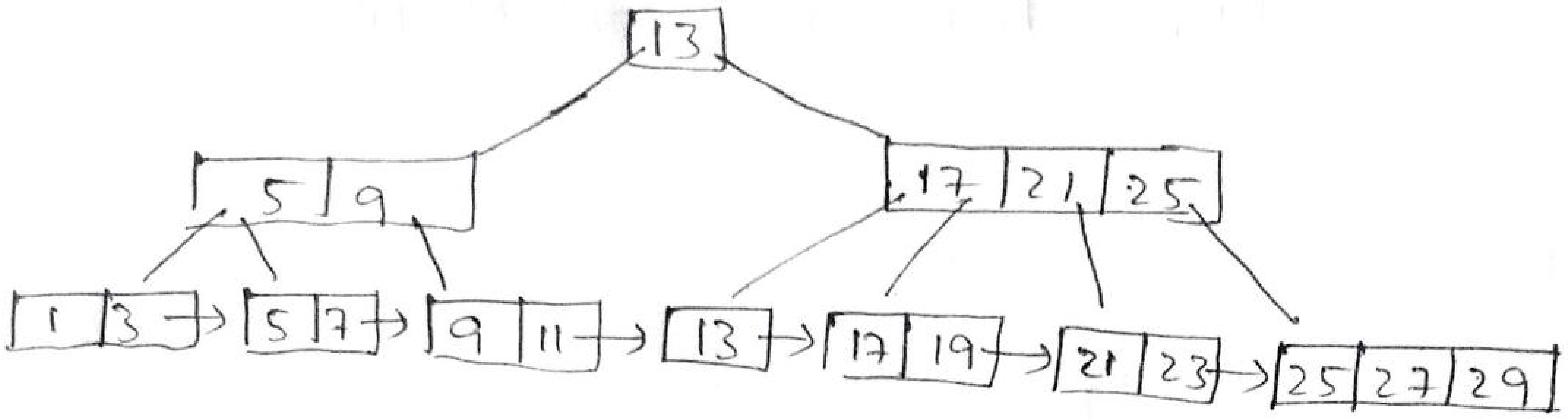


Scanned by TapScanner

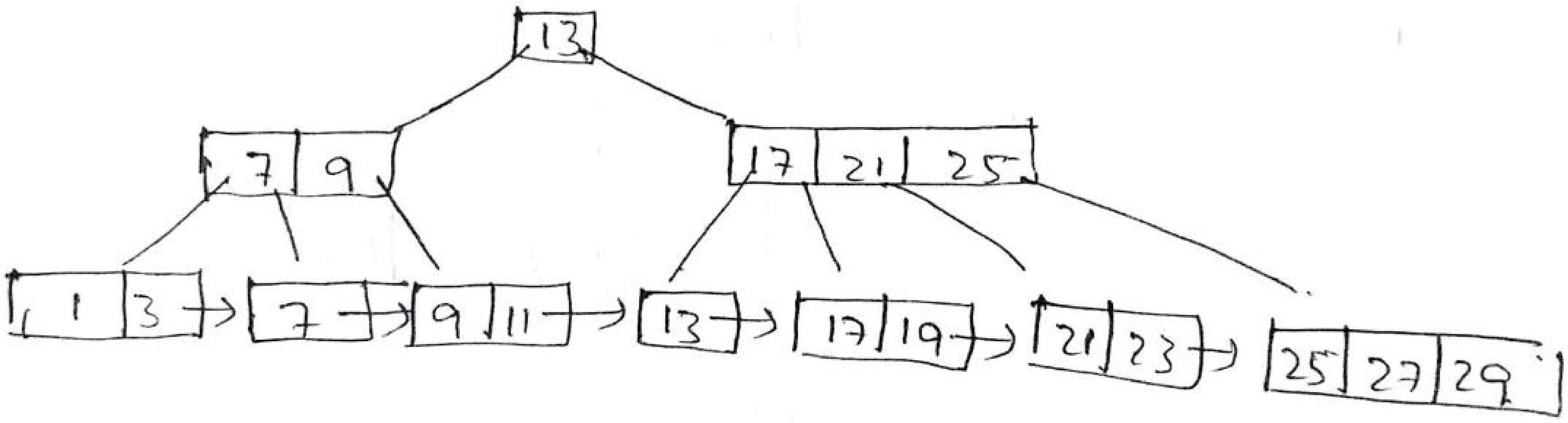
29 -



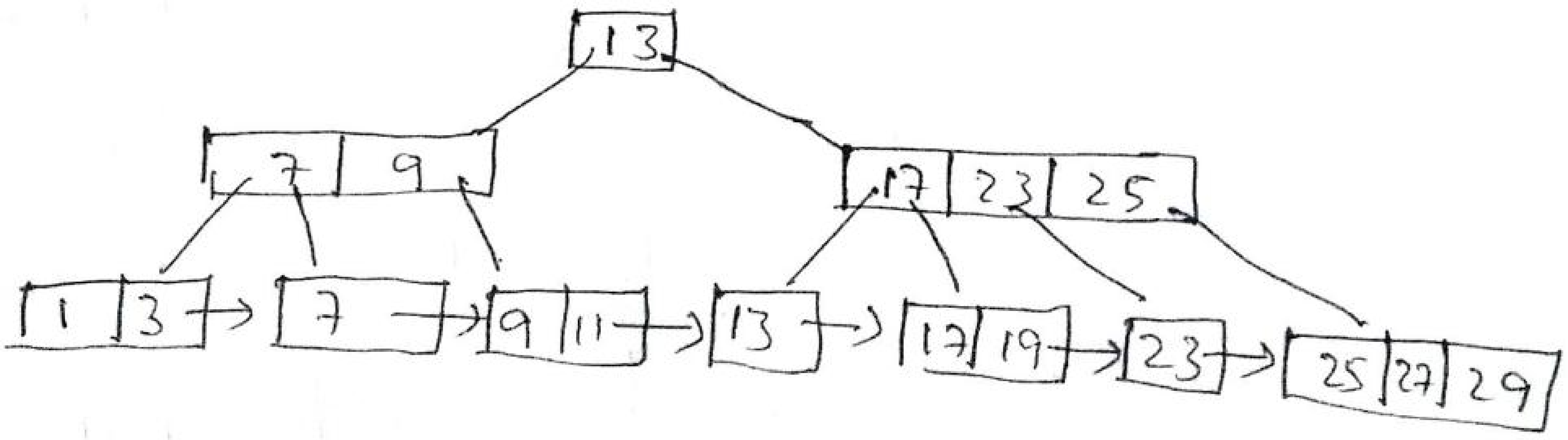
Deletion of •5

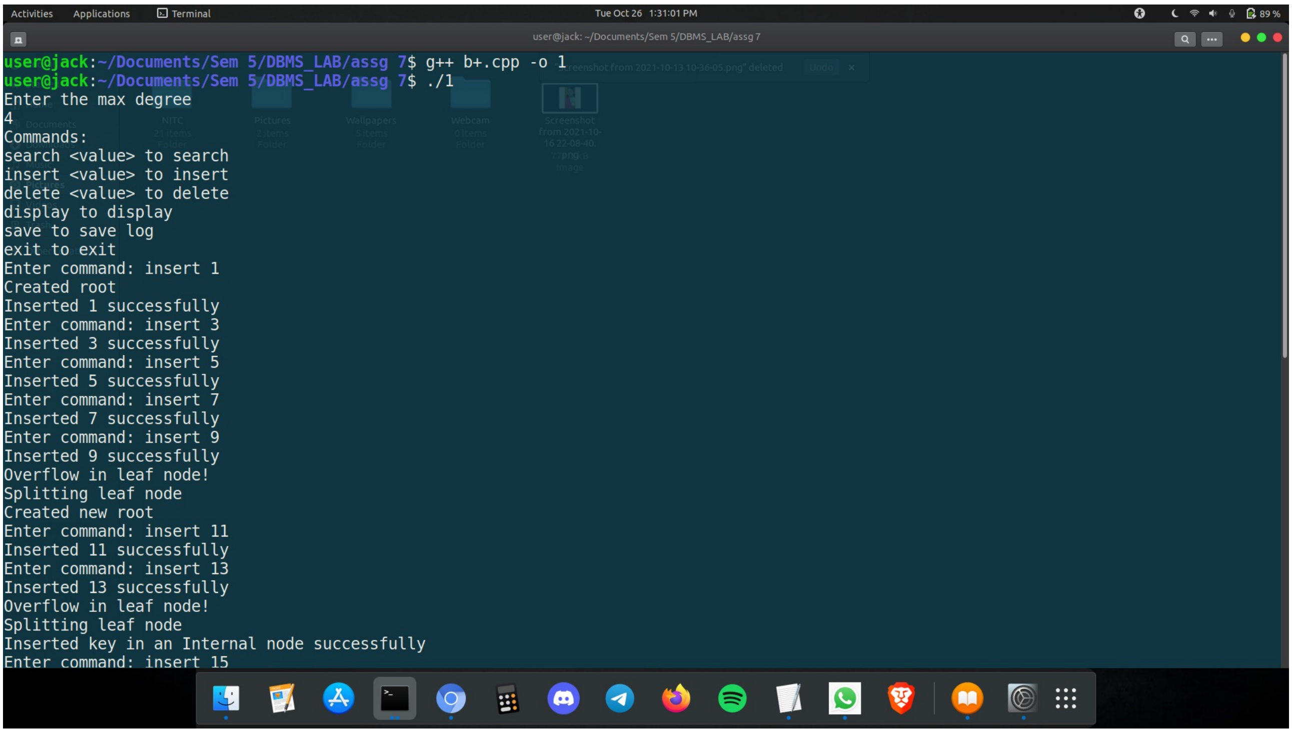


Deletion of 5

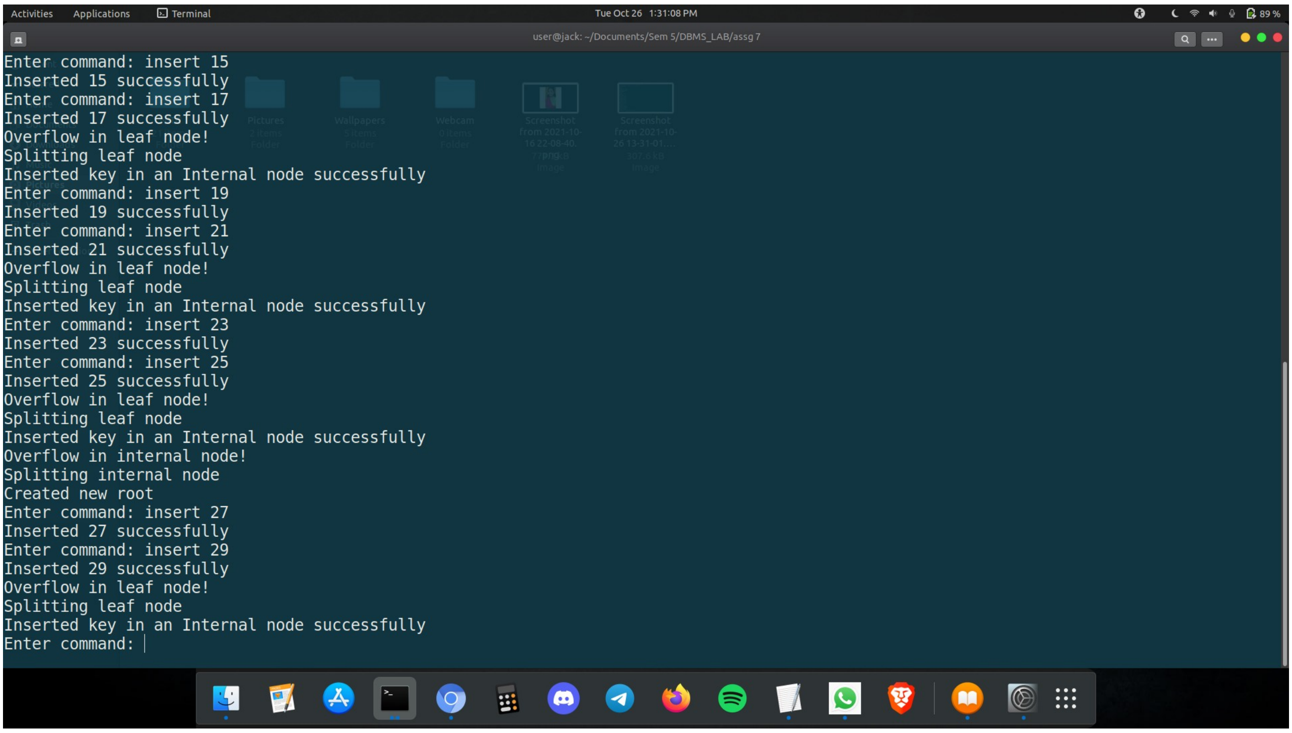


Deletion of 21

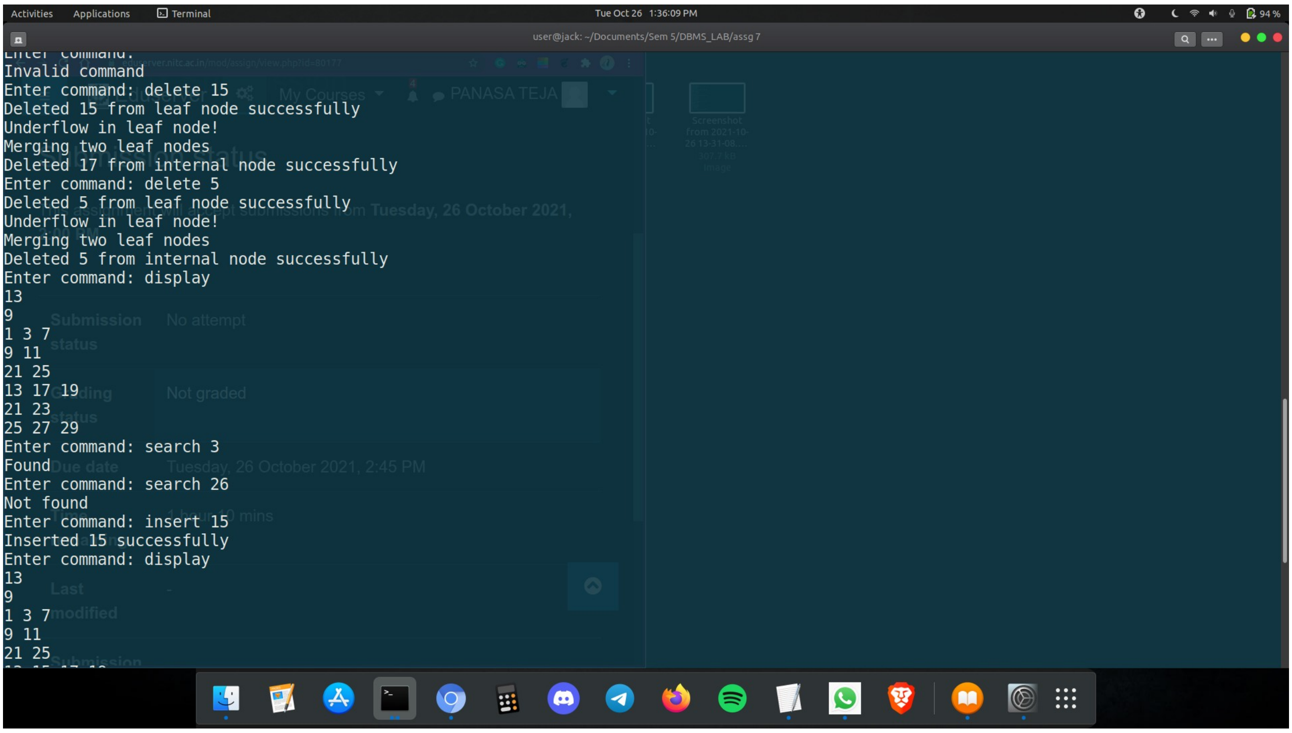




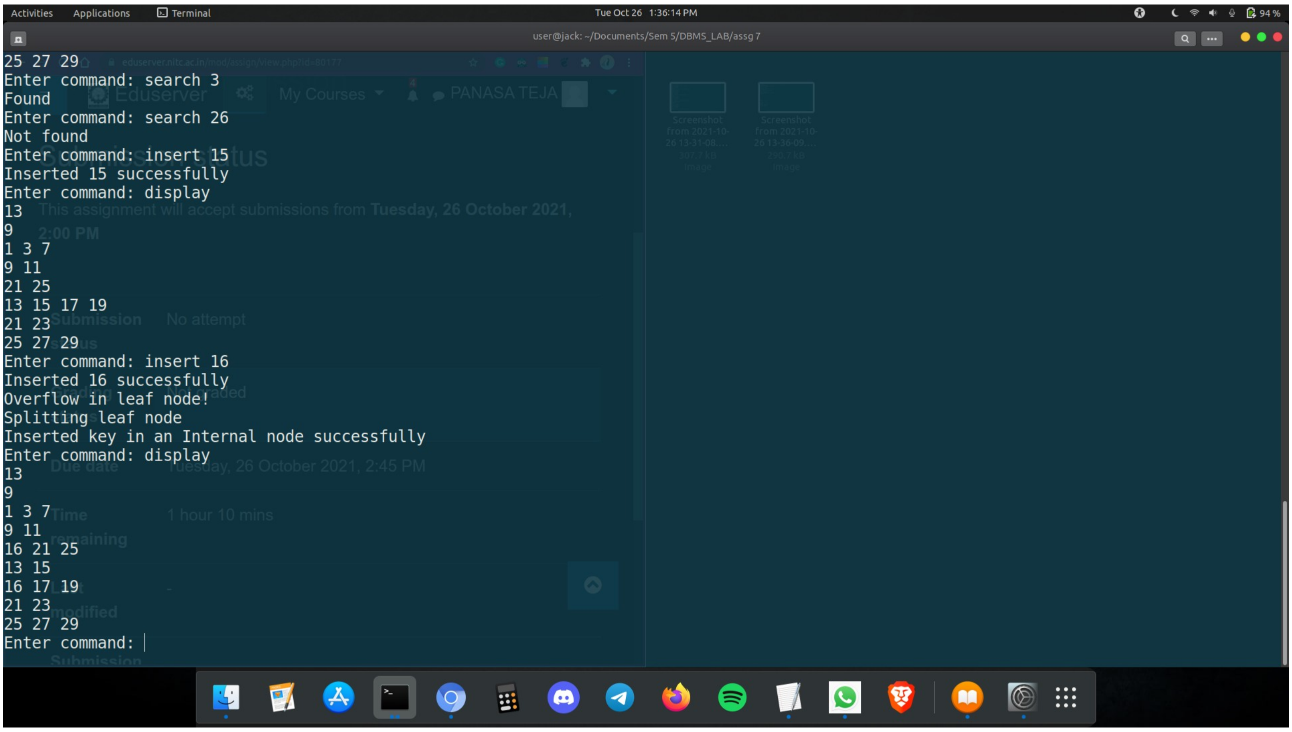
```
user@jack:~/Documents/Sem 5/DBMS_LAB/assg 7$ g++ b+.cpp -o 1
user@jack:~/Documents/Sem 5/DBMS_LAB/assg 7$ ./1
Enter the max degree
4
Commands:
search <value> to search
insert <value> to insert
delete <value> to delete
display to display
save to save log
exit to exit
Enter command: insert 1
Created root
Inserted 1 successfully
Enter command: insert 3
Inserted 3 successfully
Enter command: insert 5
Inserted 5 successfully
Enter command: insert 7
Inserted 7 successfully
Enter command: insert 9
Inserted 9 successfully
Overflow in leaf node!
Splitting leaf node
Created new root
Enter command: insert 11
Inserted 11 successfully
Enter command: insert 13
Inserted 13 successfully
Overflow in leaf node!
Splitting leaf node
Inserted key in an Internal node successfully
Enter command: insert 15
```



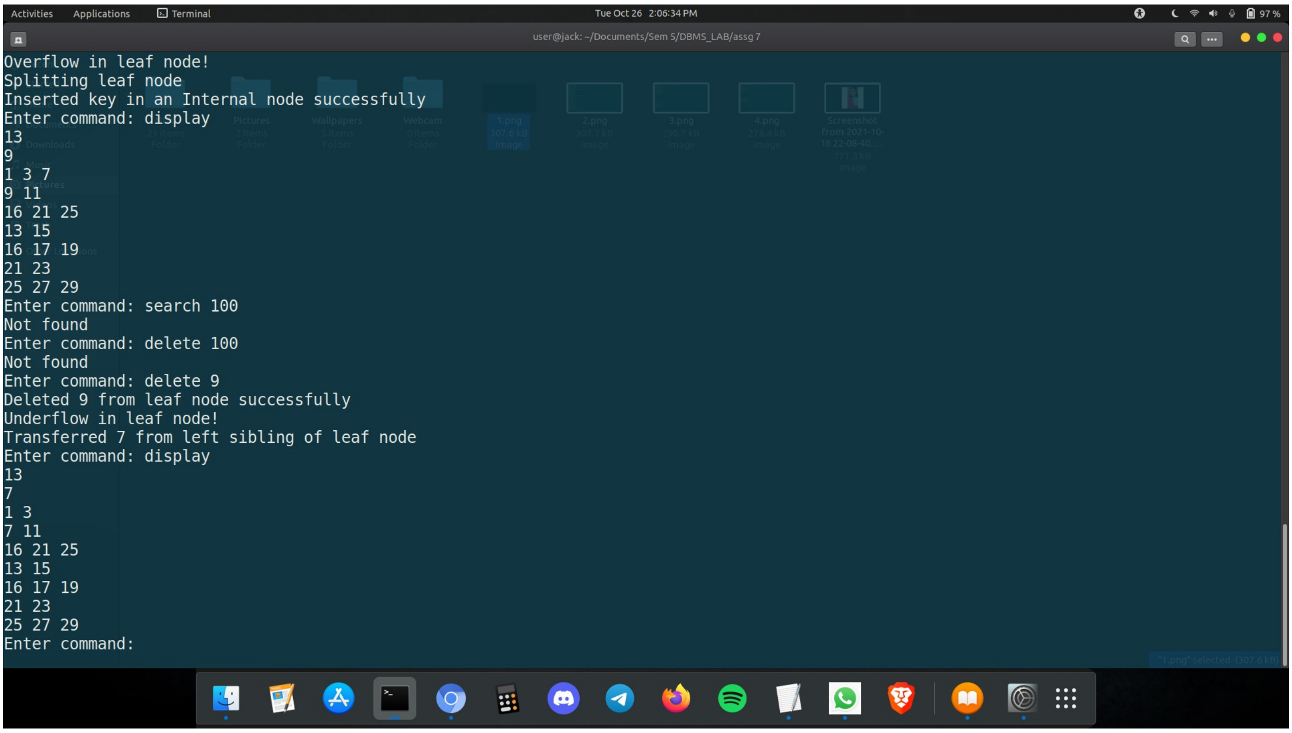
```
Enter command: insert 15
Inserted 15 successfully
Enter command: insert 17
Inserted 17 successfully
Overflow in leaf node!
Splitting leaf node
Inserted key in an Internal node successfully
Enter command: insert 19
Inserted 19 successfully
Enter command: insert 21
Inserted 21 successfully
Overflow in leaf node!
Splitting leaf node
Inserted key in an Internal node successfully
Enter command: insert 23
Inserted 23 successfully
Enter command: insert 25
Inserted 25 successfully
Overflow in leaf node!
Splitting leaf node
Inserted key in an Internal node successfully
Overflow in internal node!
Splitting internal node
Created new root
Enter command: insert 27
Inserted 27 successfully
Enter command: insert 29
Inserted 29 successfully
Overflow in leaf node!
Splitting leaf node
Inserted key in an Internal node successfully
Enter command: 
```



```
Enter command:
Invalid command
Enter command: delete 15
Deleted 15 from leaf node successfully
Underflow in leaf node!
Merging two leaf nodes
Deleted 17 from internal node successfully
Enter command: delete 5
Deleted 5 from leaf node successfully
Underflow in leaf node!
Merging two leaf nodes
Deleted 5 from internal node successfully
Enter command: display
13
9
1 3 7
9 11
21 25
13 17 19
21 23
25 27 29
Enter command: search 3
Found
Enter command: search 26
Not found
Enter command: insert 15
Inserted 15 successfully
Enter command: display
13
9
1 3 7
9 11
21 25
```



```
25 27 29
Enter command: search 3
Found
Enter command: search 26
Not found
Enter command: insert 15
Inserted 15 successfully
Enter command: display
13
9
1 3 7
9 11
21 25
13 15 17 19
21 23
25 27 29
Enter command: insert 16
Inserted 16 successfully
Overflow in leaf node!
Splitting leaf node
Inserted key in an Internal node successfully
Enter command: display
13
9
1 3 7
9 11
16 21 25
13 15
16 17 19
21 23
25 27 29
Enter command:
```



```
Overflow in leaf node!
Splitting leaf node
Inserted key in an Internal node successfully
Enter command: display
13
9
1 3 7
9 11
16 21 25
13 15
16 17 19
21 23
25 27 29
Enter command: search 100
Not found
Enter command: delete 100
Not found
Enter command: delete 9
Deleted 9 from leaf node successfully
Underflow in leaf node!
Transferred 7 from left sibling of leaf node
Enter command: display
13
7
1 3
7 11
16 21 25
13 15
16 17 19
21 23
25 27 29
Enter command:
```