B+ Tree

Data structure implementation Details:

```
struct node {
        int size; //number of key + 1 or think as #child pointers present = degree
        vector<node*> child; //child pointers
        vector<pair<double, vector<string> > > keys;
        node* next;
        node* prev;
        node* par; //parent pointer
        bool leaf;
};
```

pair<**double**, vector<string> > p → this is used to store key value pair where key holds a vector of values.
vector<pair<**double**, vector<string> > > keys → vector<pair p> which is used to multiple keys.

I have used this node structure for representing both the leaf and internal nodes.

1. Leaf node

- size = #keys + 1
- Child vector will be empty
- keys vector will hold the vector(key, value)
- next pointer and prev pointer will form a doubly linked list
- parent pointer will point to corresponding parent (internal node)
- leaf will be set to true

2. Internal node

- size = #keys + 1
- child vector will point to internal or leaf node
- keys vector will only hold the key (pair< double, vector<string>> → vector<string> will be empty)
- next and prev pointer will be set to NULL
- parent pointer will pointer to parent (internal node)
- leaf will be set to false

Program Description

```
int main(int argc, char** argv) {
        creates root node *head;
        input file f = argv[1];
        parse(f, head, outfile); //function for parsing the input file, outfile is where the output is written
        recdel(head); //cleanup the memory
}


void parse(ifstream &f, node* &head, ofstream &outfile) {
        while(newline) {
                checks whether new line has ‘,’ in it so that we know its search(key) or search(key1, key2)
                ‘,’ is present par2 is set to true other wise fault
```

```
            if new line has cmd == “Insert”
                parseIns(newline, head)
            else if new line has cmd == “Search”
                parsesearch(newline, par2, head, outfile)
        }
}

void parseIns(string x, node* &head) {
        parses the string to get key and value;
        insert(head, key, value);
}

void insert(node* &head, double key, string value) {
        checks whether the key is duplicate,
        if(duplicate )
            it inserts in the value in to corresponding key vector (it    inserts in last position push_back())
        else: // new key
            insertleaf(x, i, key, value, head);
            // x → node where the (key,value) to be inserted, i → position where it should be inserted
}

void insertLeaf(node * x, int i, double key, string value, node* &head) {
        inserts the (key,value) pair in node x at index i
        if(full node) {
            given node x is full its split into x and newnode, gets the value to be inserted in parent → p
            // newnode is the leaf node, and the fields are set correspondingly
            // p is pair<double, <vector<string> > whose vector<string> is empty
            sets the doubly linked list accordingly
            if(x has parent)
                insertPar(x->par, p, newnode, head);
            else: // x is root node
                create new root, whose key is p and sets the other field
                x→par = new root; newnode→ par = newroot
                head = newroot
        }
}

void insertPar(node* x, pair<double, vector<string> > p, node* childnode, node* & head) {
        inserts the pair p in node x (x.keys) at corresponding position
        inserts the childnode in node x (x.child) at corresponding position
        if(full node) { // split is different than that of leaf split
            given node node x is split into two x and newnode, p is the value to be inserted in parent
            // newnode is internal node and the fields are set correspondingly
            // p is pair<double, <vector<string> > whose vector<string> is empty
            if(x has parent)
                recursive call insertPar(x->par, p, newnode, head);
            else: // x is root node
                create new root, whose key is p and sets the other field
                x→par = new root; newnode→ par = newroot
                head = newroot
        }
}
```

```
void parseSearch(string x, bool par2, node*head, ofstream &outfile) {
        if(par2)
                parses the input to get key1, key2; calls search(head, key1, key2, outfile)
        else
                parses the input to get key; calls search(head, key, outfile)
}

void search_range(node *head, double key1, double key2, ofstream &outfile) {
        finds the node where key k1 is present, let it be node x;
        while(x) {
                prints the key, value pair to outfile such that key1 <= key <= key2
                if(key > key2)
                        break;
                x = x->next;
        }
        if(no key found)
                prints “Null”
}

void search(node *head, double key, ofstream &outfile) {
        finds the node where key is present
        prints the values for the key to the outfile
        if(key not found)
                prints “Null”
}
```

Functions where insert, search and search_range happens

- **void insert**(node* &head, **double** key, string value)
- **void search_range**(node *head, **double** key1, **double** key2, ofstream &outfile)
- **void search**(node *head, **double** key, ofstream &outfile)