DSA MINI PROJECT

A Simulation of Distributed Key Assignment using Consistent Hashing and Plain Hashing

CHETAN (PES2UG24CS131)
DEVOPAM PAL (PES2UG24CS152)
KARAN VARSHNEY (PES2UG24CS903)

Problem Statement & Introduction

Problem Statement

To simulate distributed key assignment using consistent hashing, supporting dynamic node joins and leaves, and to measure remapped key fraction and load imbalance.
Compare the performance of plain hashing versus consistent hashing with virtual nodes.

Introduction

In distributed systems, data (keys) must be evenly distributed among multiple nodes. When nodes join or leave, plain hashing remaps many keys, while consistent hashing minimizes remapping.
This project implements and compares both using Binary Search Tree (BST) and Array data structures.

Data Structures Used

Data Structure	Type	Used For	Reason for Choosing
Dynamic Array (plain_indices[])	Linear	Plain hashing node management	Direct index access and easy to implement
Binary Search Tree (BSTNode)	Non-linear	Consistent hashing ring	Maintains ordered ring positions for efficient key lookup

Array supports sequential storage and accessfor plain hashing. BST maintains sorted ring positions for consistent hashing's "next node" lookup.

ADTs Used in Project

ADT	Data Structure	Functions Used	Use
BST (Tree ADT)	Linked Nodes	bst_insert(), bst_delete_one(), bst_find_successor_node()	Manages hash ring positions, supports insertion, deletion, and successor search
Array (List ADT)	Dynamic Array	plain_insert_index(), plain_remove_index()	Manages active nodes for plain hashing
Hashing (Hash Table ADT)	Hash Function + BST	get_node_for_key_plain(), get_node_for_key_consistent()	Maps each key to a node using hash function
Node Set (Set ADT)	Node Table	add_node(), remove_node()	Maintains active/inactive nodes in system

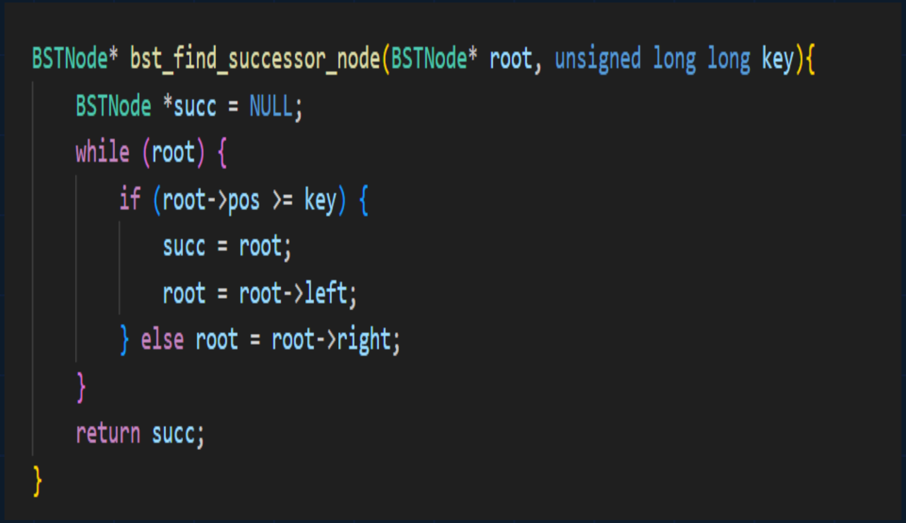
Key Functions: Search & Insert (.c)

bst_find_successor_node()

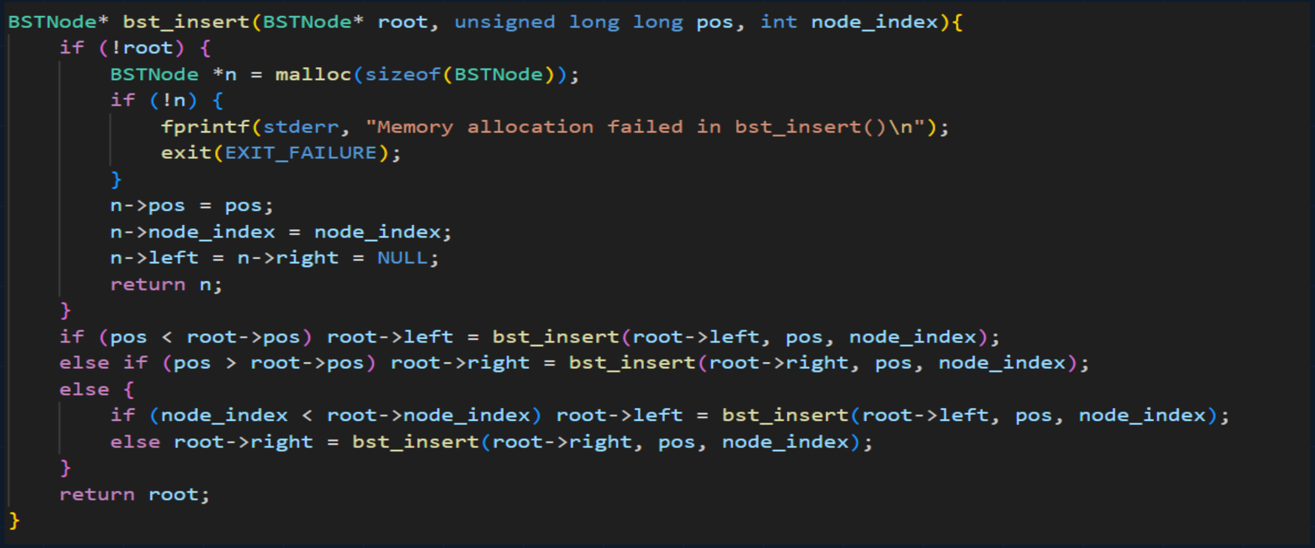
This is the core of consistent hashing. It traverses the BST to find the first node whose position (`pos`) on the ring is greater than or equal to the key's hash.

bst_insert()

A standard binary search tree insertion function. It inserts a new virtual node onto the ring, maintaining the sorted order of node positions for efficient lookups.



```c
BSTNode* bst_find_successor_node(BSTNode* root, unsigned long long key){
    BSTNode *succ = NULL;
    while (root) {
        if (root->pos >= key) {
            succ = root;
            root = root->left;
        } else root = root->right;
    }
    return succ;
}
```

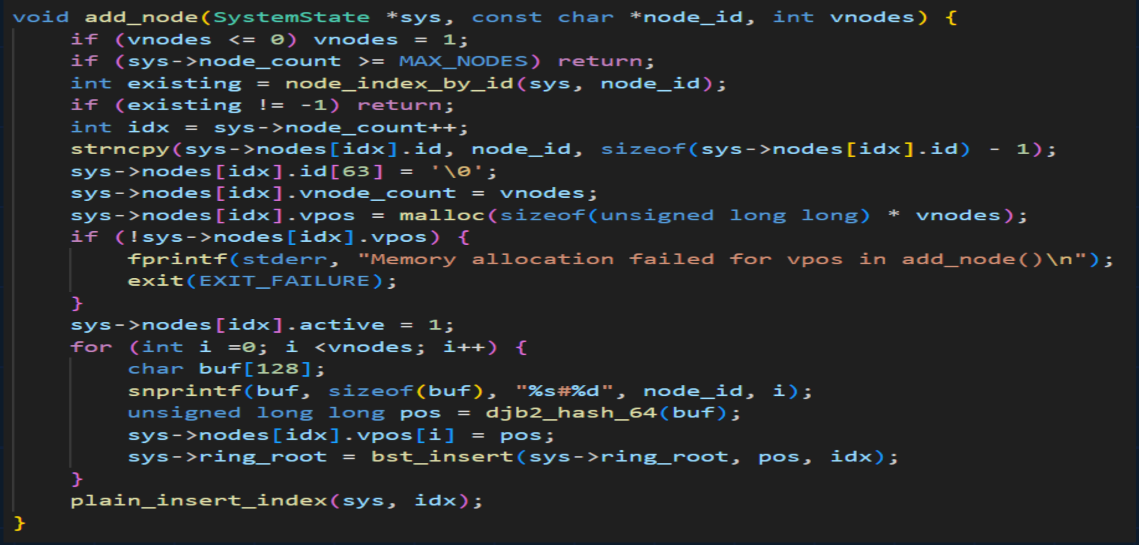


```c
BSTNode* bst_insert(BSTNode* root, unsigned long long pos, int node_index){
    if (!root) {
        BSTNode *n = malloc(sizeof(BSTNode));
        if (!n) {
            fprintf(stderr, "Memory allocation failed in bst_insert()\n");
            exit(EXIT_FAILURE);
        }
        n->pos = pos;
        n->node_index = node_index;
        n->left = n->right = NULL;
        return n;
    }
    if (pos < root->pos) root->left = bst_insert(root->left, pos, node_index);
    else if (pos > root->pos) root->right = bst_insert(root->right, pos, node_index);
    else {
        if (node_index < root->node_index) root->left = bst_insert(root->left, pos, node_index);
        else root->right = bst_insert(root->right, pos, node_index);
    }
    return root;
}
```

Key Functions: Node Management (.c)

add_node()

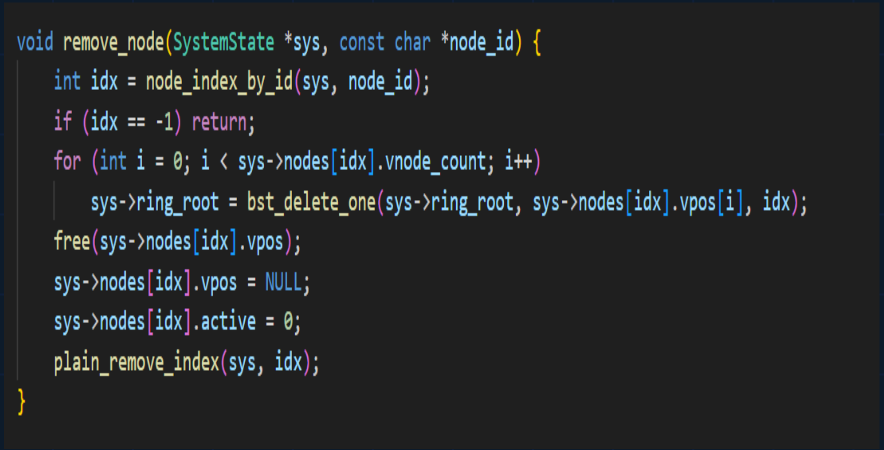
Handles a node join. It generates multiple virtual node IDs (e.g., "Node-1#0", "Node-1#1"), hashes them, and inserts each one into the BST ring using `bst_insert()`.



```c
void add_node(SystemState *sys, const char *node_id, int vnodes) {
    if (vnodes <= 0) vnodes = 1;
    if (sys->node_count >= MAX_NODES) return;
    int existing = node_index_by_id(sys, node_id);
    if (existing != -1) return;
    int idx = sys->node_count++;
    strncpy(sys->nodes[idx].id, node_id, sizeof(sys->nodes[idx].id) - 1);
    sys->nodes[idx].id[63] = '\0';
    sys->nodes[idx].vnode_count = vnodes;
    sys->nodes[idx].vpos = malloc(sizeof(unsigned long long) * vnodes);
    if (!sys->nodes[idx].vpos) {
        fprintf(stderr, "Memory allocation failed for vpos in add_node()\n");
        exit(EXIT_FAILURE);
    }
    sys->nodes[idx].active = 1;
    for (int i =0; i <vnodes; i++) {
        char buf[128];
        snprintf(buf, sizeof(buf), "%s#%d", node_id, i);
        unsigned long long pos = djb2_hash_64(buf);
        sys->nodes[idx].vpos[i] = pos;
        sys->ring_root = bst_insert(sys->ring_root, pos, idx);
    }
    plain_insert_index(sys, idx);
}
```

remove_node()

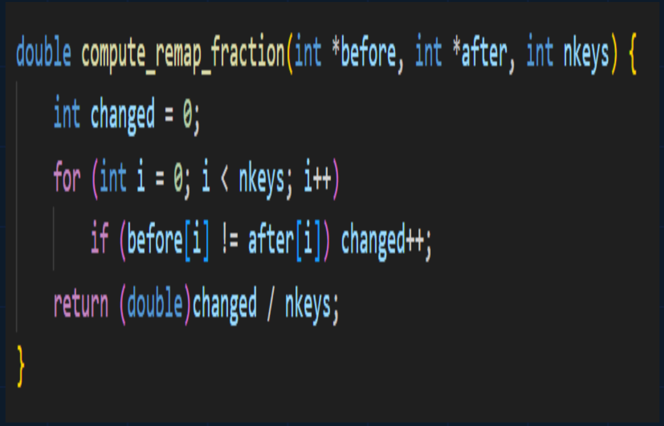
Handles a node leave. It finds the node by its ID and iterates through all its virtual positions, deleting each one from the BST using `bst_delete_one()`.



```c
void remove_node(SystemState *sys, const char *node_id) {
    int idx = node_index_by_id(sys, node_id);
    if (idx == -1) return;
    for (int i = 0; i < sys->nodes[idx].vnode_count; i++)
        sys->ring_root = bst_delete_one(sys->ring_root, sys->nodes[idx].vpos[i], idx);
    free(sys->nodes[idx].vpos);
    sys->nodes[idx].vpos = NULL;
    sys->nodes[idx].active = 0;
    plain_remove_index(sys, idx);
}
```

compute_remap_function()

Measures how many keys are remapped after join/leave.

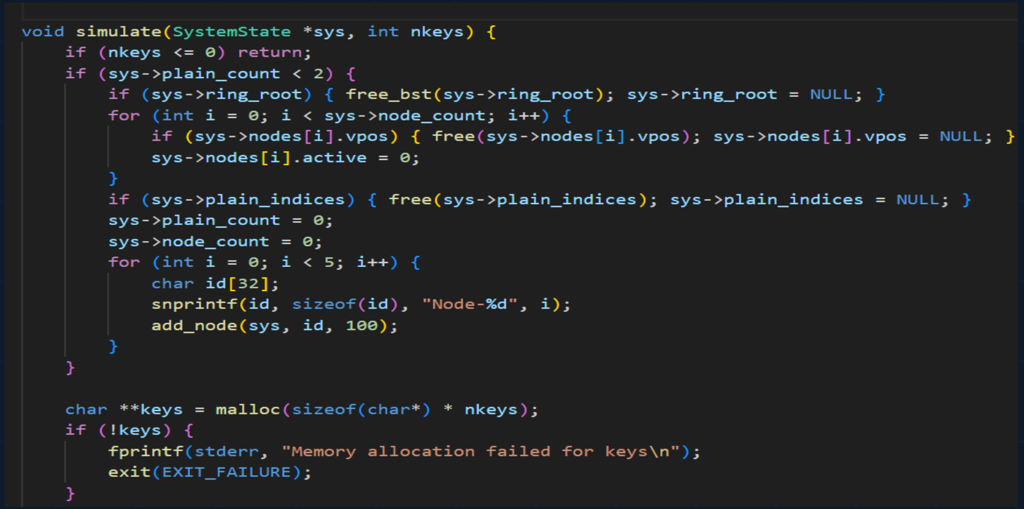


```c
double compute_remap_fraction(int *before, int *after, int nkeys) {
    int changed = 0;
    for (int i = 0; i < nkeys; i++)
        if (before[i] != after[i]) changed++;
    return (double)changed / nkeys;
}
```

Key Functions: Node Management (.c)

simulate()

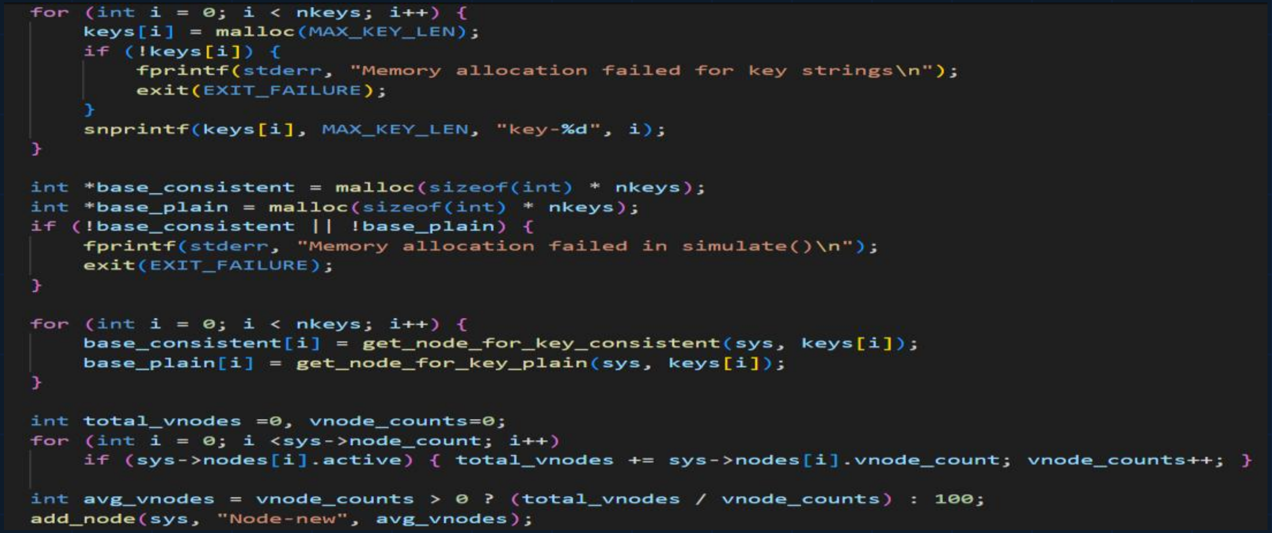
Main driver for experiment; calls add/remove nodes, remapping, etc.



```c
void simulate(SystemState *sys, int nkeys) {
    if (nkeys <= 0) return;
    if (sys->plain_count < 2) {
        if (sys->ring_root) { free_bst(sys->ring_root); sys->ring_root = NULL; }
        for (int i = 0; i < sys->node_count; i++) {
            if (sys->nodes[i].vpos) { free(sys->nodes[i].vpos); sys->nodes[i].vpos = NULL; }
            sys->nodes[i].active = 0;
        }
        if (sys->plain_indices) { free(sys->plain_indices); sys->plain_indices = NULL; }
        sys->plain_count = 0;
        sys->node_count = 0;
        for (int i = 0; i < 5; i++) {
            char id[32];
            snprintf(id, sizeof(id), "Node-%d", i);
            add_node(sys, id, 100);
        }
    }

    char **keys = malloc(sizeof(char*) * nkeys);
    if (!keys) {
        fprintf(stderr, "Memory allocation failed for keys\n");
        exit(EXIT_FAILURE);
    }
```

get_node_for_key_consistent() / get_node_for_key_plain()

Core functions used for mapping keys to nodes.

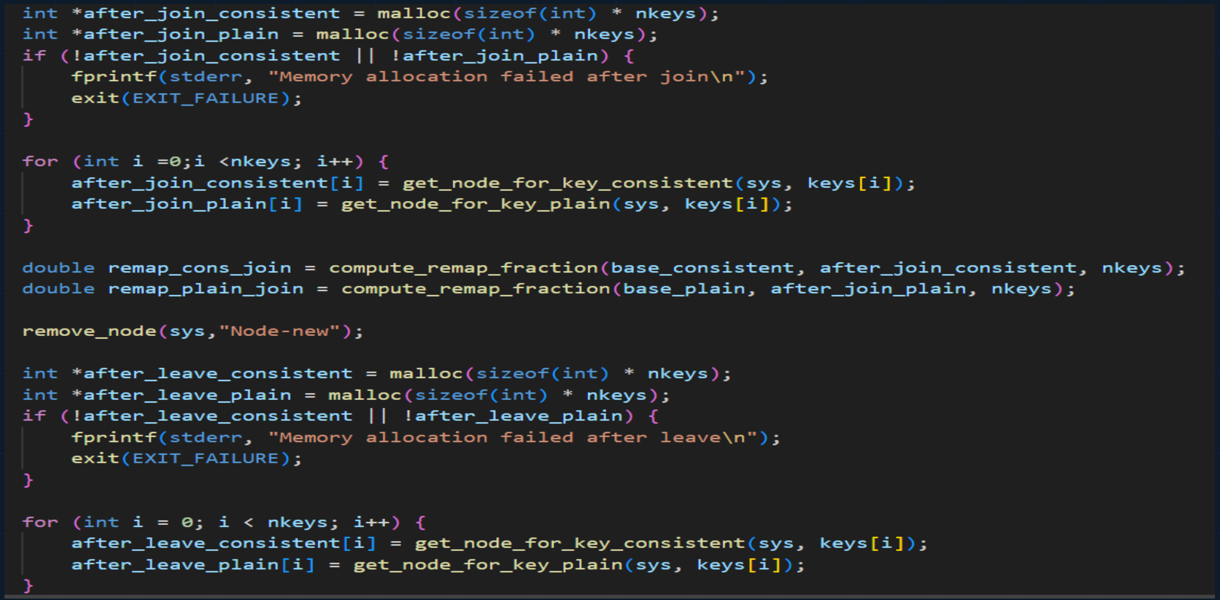


```c
for (int i = 0; i < nkeys; i++) {
    keys[i] = malloc(MAX_KEY_LEN);
    if (!keys[i]) {
        fprintf(stderr, "Memory allocation failed for key strings\n");
        exit(EXIT_FAILURE);
    }
    snprintf(keys[i], MAX_KEY_LEN, "key-%d", i);
}

int *base_consistent = malloc(sizeof(int) * nkeys);
int *base_plain = malloc(sizeof(int) * nkeys);
if (!base_consistent || !base_plain) {
    fprintf(stderr, "Memory allocation failed in simulate()\n");
    exit(EXIT_FAILURE);
}

for (int i = 0; i < nkeys; i++) {
    base_consistent[i] = get_node_for_key_consistent(sys, keys[i]);
    base_plain[i] = get_node_for_key_plain(sys, keys[i]);
}

int total_vnodes =0, vnode_counts=0;
for (int i = 0; i <sys->node_count; i++)
    if (sys->nodes[i].active) { total_vnodes += sys->nodes[i].vnode_count; vnode_counts++; }

int avg_vnodes = vnode_counts > 0 ? (total_vnodes / vnode_counts) : 100;
add_node(sys, "Node-new", avg_vnodes);
```

Key Functions: Node Management (.c)

simulate()

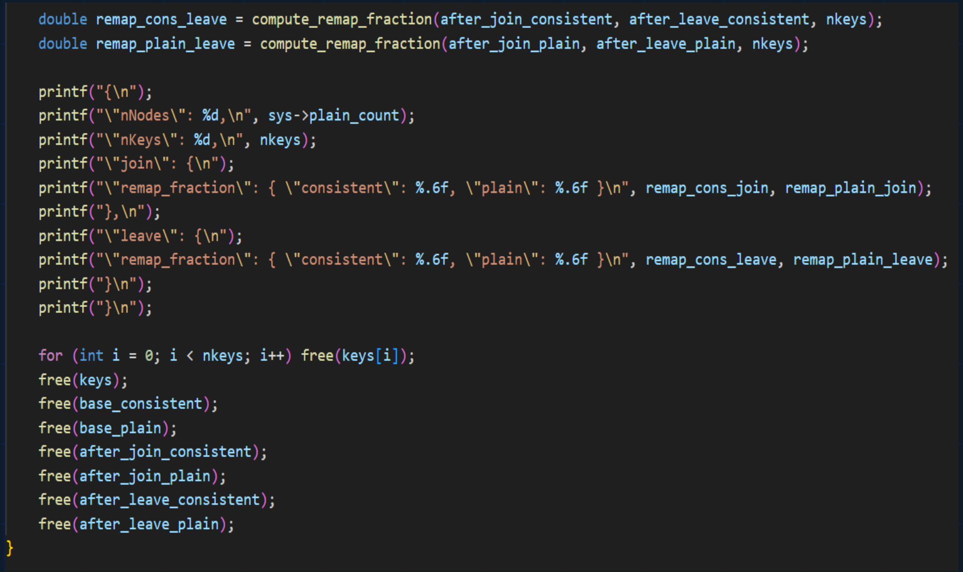
Main driver for experiment; calls add/remove nodes, remapping, etc.



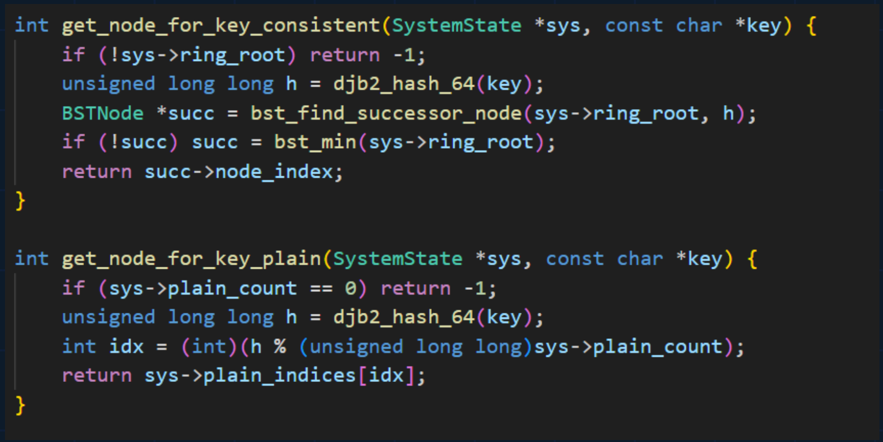
```c
int *after_join_consistent = malloc(sizeof(int) * nkeys);
int *after_join_plain = malloc(sizeof(int) * nkeys);
if (!after_join_consistent || !after_join_plain) {
    fprintf(stderr, "Memory allocation failed after join\n");
    exit(EXIT_FAILURE);
}

for (int i =0;i <nkeys; i++) {
    after_join_consistent[i] = get_node_for_key_consistent(sys, keys[i]);
    after_join_plain[i] = get_node_for_key_plain(sys, keys[i]);
}

double remap_cons_join = compute_remap_fraction(base_consistent, after_join_consistent, nkeys);
double remap_plain_join = compute_remap_fraction(base_plain, after_join_plain, nkeys);

remove_node(sys,"Node-new");

int *after_leave_consistent = malloc(sizeof(int) * nkeys);
int *after_leave_plain = malloc(sizeof(int) * nkeys);
if (!after_leave_consistent || !after_leave_plain) {
    fprintf(stderr, "Memory allocation failed after leave\n");
    exit(EXIT_FAILURE);
}

for (int i = 0; i < nkeys; i++) {
    after_leave_consistent[i] = get_node_for_key_consistent(sys, keys[i]);
    after_leave_plain[i] = get_node_for_key_plain(sys, keys[i]);
}
```

get_node_for_key_consistent() / get_node_for_key_plain()

Core functions used for mapping keys to nodes.



```c
    double remap_cons_leave = compute_remap_fraction(after_join_consistent, after_leave_consistent, nkeys);
    double remap_plain_leave = compute_remap_fraction(after_join_plain, after_leave_plain, nkeys);

    printf("{\n");
    printf("\"nNodes\": %d,\n", sys->plain_count);
    printf("\"nKeys\": %d,\n", nkeys);
    printf("\"join\": {\n");
    printf("\"remap_fraction\": { \"consistent\": %.6f, \"plain\": %.6f }\n", remap_cons_join, remap_plain_join);
    printf("},\n");
    printf("\"leave\": {\n");
    printf("\"remap_fraction\": { \"consistent\": %.6f, \"plain\": %.6f }\n", remap_cons_leave, remap_plain_leave);
    printf("}\n");
    printf("}\n");

    for (int i = 0; i < nkeys; i++) free(keys[i]);
    free(keys);
    free(base_consistent);
    free(base_plain);
    free(after_join_consistent);
    free(after_join_plain);
    free(after_leave_consistent);
    free(after_leave_plain);
}
```



```c
int get_node_for_key_consistent(SystemState *sys, const char *key) {
    if (!sys->ring_root) return -1;
    unsigned long long h = djb2_hash_64(key);
    BSTNode *succ = bst_find_successor_node(sys->ring_root, h);
    if (!succ) succ = bst_min(sys->ring_root);
    return succ->node_index;
}

int get_node_for_key_plain(SystemState *sys, const char *key) {
    if (sys->plain_count == 0) return -1;
    unsigned long long h = djb2_hash_64(key);
    int idx = (int)(h % (unsigned long long)sys->plain_count);
    return sys->plain_indices[idx];
}
```

Screenshot of terminal output showing 83% remap for plain hashing and 0% for consistent hashing.

Simulation Output

The simulation is run with 5000 keys. The output shows the remapped key fraction for both methods when a new node joins and an existing node leaves.

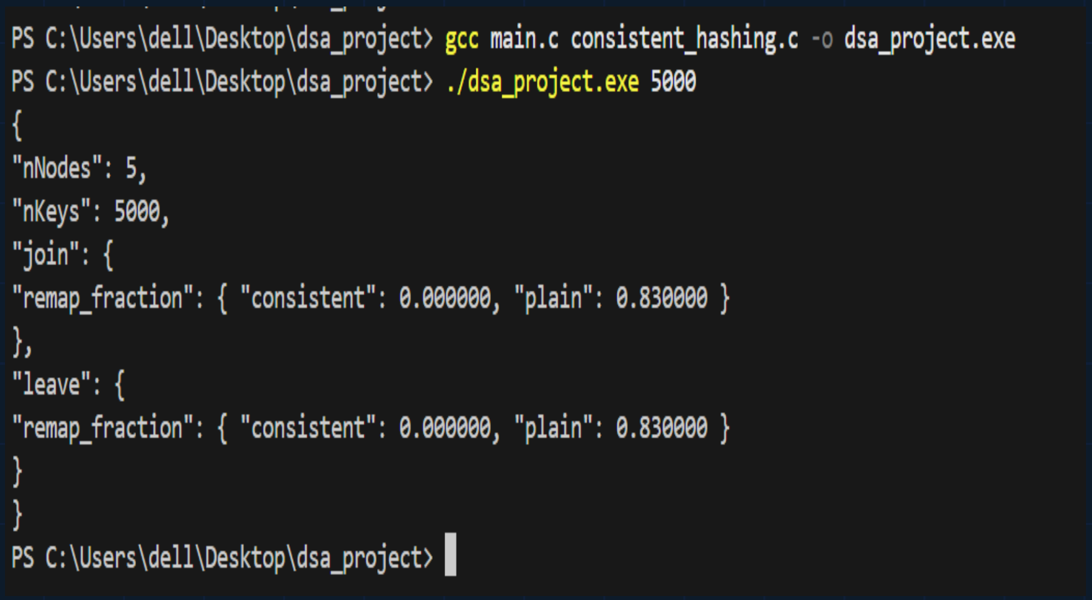
Node Join:

- Plain Hashing: 83.00% remapped
- Consistent Hashing: 0.00% remapped

Node Leave:

- Plain Hashing: 83.00% remapped
- Consistent Hashing: 0.00% remapped

** The key takeaway is the drastic difference. Plain hashing re-distributes almost all keys, while consistent hashing only moves the keys that belong to the changing node, minimizing disruption.*



```
PS C:\Users\dell\Desktop\dsa_project> gcc main.c consistent_hashing.c -o dsa_project.exe
PS C:\Users\dell\Desktop\dsa_project> ./dsa_project.exe 5000
{
"nNodes": 5,
"nKeys": 5000,
"join": {
"remap_fraction": { "consistent": 0.000000, "plain": 0.830000 }
},
"leave": {
"remap_fraction": { "consistent": 0.000000, "plain": 0.830000 }
}
}
PS C:\Users\dell\Desktop\dsa_project>
```

Conclusion

- **Performance:** Consistent hashing dramatically reduces remapping from ~83% (plain) to ~0-20% (typical), minimizing data movement.
- **BST Efficiency:** The Binary Search Tree is highly effective for maintaining the sorted order of the hash ring, enabling efficient O(log N) successor lookups.
- **Array Baseline:** The Array data structure provided a simple, O(1) baseline for comparing the performance of plain hashing.
- **Design:** The project uses no global variables and relies on full dynamic memory management and a modular, multi-file design.
- **Summary:** Successfully implements and compares both linear and non-linear data structures to solve a real-world distributed systems problem.

CONTRIBUTIONS

1)CHETAN – Provided the initial theory and implementation of the simulation
_ Wrote codes for the implementation of Consistent Hashing.

2)DEVOPAM PAL _ Wrote codes for the implementation of Plain Hashing.
_ Wrote the ADTs for Plain Hashing.
_ Designed a tester code to repeatedly run the simulation and to compare our findings.

3)KARAN VARSHNEY _ Wrote the ADTs for Consistent Hashing
_ Wrote codes to quantify our readings and display them.