Assignment 05/08/2024

Author: ThanhTH10

Date: 06/08/2024

1. Real time application for dynamic memory allocation. Minimum 3 examples.

**Game Development:**

- Dynamically create and destroy game objects (enemies, projectiles, etc.) as needed.

- Efficiently allocate and deallocate memory for game objects to handle a large number of them.

**Multimedia Processing:**

- Dynamically allocate memory to store variable-sized data (video frames, audio samples).

- Manage memory required for multimedia processing to ensure smooth playback or editing.

**Web Servers and Network Applications:**

- Dynamically allocate memory to handle incoming client requests and connections.

- Efficiently manage memory for each client connection to support a large number of concurrent connections.

1. What is hash data structure. How it will be implemented with linked list.in C

A hash data structure, also known as a hash table, is a data storage and retrieval system that uses a hash function to map keys to specific indices of a backing array. The hash function takes a key as input and generates a hash code, which is used to determine the index at which the corresponding value is stored.

***Implementation:***

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// Define the structure for a linked list node

typedef struct Node

{

    char \*key;

    int value;

    struct Node \*next;

} Node;

// Define the structure for the hash table

typedef struct HashTable

{

    int size;

    Node \*\*buckets;

} HashTable;

// Function to create a new linked list node

Node \*createNode(char \*key, int value)

{

    Node \*node = (Node \*)malloc(sizeof(Node));

    node->key = strdup(key);

    node->value = value;

    node->next = NULL;

    return node;

}

// Function to create a new hash table

HashTable \*createHashTable(int size)

{

    HashTable \*table = (HashTable \*)malloc(sizeof(HashTable));

    table->size = size;

    table->buckets = (Node \*\*)calloc(size, sizeof(Node \*));

    return table;

}

// Hash function to map keys to indices

int hashFunction(char \*key, int size)

{

    int hash = 0;

    for (int i = 0; i < strlen(key); i++)

    {

        hash += key[i];

    }

    return hash % size;

}

// Function to insert a key-value pair into the hash table

void insert(HashTable \*table, char \*key, int value)

{

    int index = hashFunction(key, table->size);

    Node \*node = createNode(key, value);

    if (table->buckets[index] == NULL)

    {

        table->buckets[index] = node;

    }

    else

    {

        Node \*current = table->buckets[index];

        while (current->next != NULL)

        {

            current = current->next;

        }

        current->next = node;

    }

}

// Function to search for a value in the hash table

int search(HashTable \*table, char \*key)

{

    int index = hashFunction(key, table->size);

    Node \*current = table->buckets[index];

    while (current != NULL)

    {

        if (strcmp(current->key, key) == 0)

        {

            return current->value;

        }

        current = current->next;

    }

    return -1; // Key not found

}

// Function to delete a key-value pair from the hash table

void delete(HashTable \*table, char \*key)

{

    int index = hashFunction(key, table->size);

    Node \*current = table->buckets[index];

    Node \*previous = NULL;

    while (current != NULL)

    {

        if (strcmp(current->key, key) == 0)

        {

            if (previous == NULL)

            {

                table->buckets[index] = current->next;

            }

            else

            {

                previous->next = current->next;

            }

            free(current->key);

            free(current);

            return;

        }

        previous = current;

        current = current->next;

    }

}

int main()

{

    HashTable \*table = createHashTable(10);

    insert(table, "apple", 5);

    insert(table, "banana", 7);

    printf("%d\n", search(table, "apple")); // Output: 5

    delete (table, "apple");

    printf("%d\n", search(table, "apple")); // Output: -1

    return 0;

}

1. what is sprintf function in C. Application of sprintf

**sprintf** allows to print formatted output to a string. It is similar to **printf**, but instead of printing to the console, it stores the formatted output in a character array.

***Example***:

#include <stdio.h>

int main()

{

    char buffer[50];

    int num = 10;

    sprintf(buffer, "The value of num is %d", num);

    printf("buffer: %s\n", buffer);

    return 0;

}

/\*

In this example, the sprintf function is

used to format the string "The value of num is 10"

and store it in the buffer array. The formatted

string is then printed to the console using printf.

\*/

***Applications***:

1. **String formatting**: sprintf can be used to format strings with variables, making it easier to create dynamic strings.
2. **Buffering output:** sprintf can be used to buffer output, allowing you to store formatted output in a string before printing it to the console.
3. **Error messages:** sprintf can be used to create error messages with dynamic information, such as error codes or variable values.
4. **Logging:** sprintf can be used to create log messages with formatted output.
5. **String manipulation:** sprintf can be used to perform string manipulation tasks, such as concatenating strings or inserting variables into strings.

4. Another way to avoid structure padding instead of #pragma pack(1). When application wants to avoid strcture padding.

1. **Use \_\_attribute\_\_((packed)) (GCC, Clang)**

 use the \_\_attribute\_\_((packed)) attribute to specify that a structure should be packed tightly, without any padding bytes.

struct \_\_attribute\_\_((packed)) my\_struct {

    char a;

    int b;

    short c;

};

1. **Use alignas and alignof (C++11)**

use the alignas and alignof keywords to specify the alignment of a structure.

struct alignas(1) my\_struct {

    char a;

    int b;

    short c;

};

1. **Rearrange structure members**

By rearranging the members of the structure, you can minimize the padding bytes. For example, placing the largest members first can help reduce padding.

struct my\_struct {

    int b;

    short c;

    char a;

};