**数据结构**

# 1、抽象数据类型（ADT: abstract data type）

可以用数组实现动态列表，但对于内存的利用率和消耗来说它不是最有效的（根据时间复杂度得出的结论）。因为在申明数组是就固定了数组的大小，如果后续需要更大的数组，就需要重新创建数组，并且把钱一个数组的内容全部复制过去，然后再把之前的那一块内存释放。因此需要一种更加有效的方式---链表

# 2、linked-list

内存块的地址是第一个字节的地址，类似的函数指令的地址是第一句指令的地址。对于数组来说是一个连续的内存块存储在内存中。

#include <stdio.h>

int main(int arge,const char\* argv)

{

    int a = 5;

    printf("%p\n",&a);

    printf("%p\n",&a+1);

    char\* p = (char\*)&a;

    printf("%p\n",p+1);

    printf("hello world!\n");

    printf("%d\n",sizeof(a));

    return 0;

}

对于这段代码就可以看出来，一个整型的地址是第一个字节的地址。并且对于&a+1代表&a+1\*sizeof(type(a))。

链表：

struct node

{

    int data;

    struct node\* next;

};

以上整体称为一个节点。next为一个指针，指向下一节点。链表有一个头指针head指向第一个节点，最后的一个指针指向NULL。

以下代码用于验证，一个结构体所占字节，以及结构体内部是怎样实现内存分配，字节填充。

#include <stdio.h>

typedef struct \_\_attribute\_\_((packed)) node

{

    int data;

    struct node\* next;

}node1;

node1 a;

int main(int arge,const char\* argv)

{

    // int a = 5;

    printf("%d\n",sizeof(node1));

    printf("%d\n",sizeof(a.data));

    printf("%d\n",sizeof(a.next));

    printf("%d\n",sizeof(a));

    printf("%d\n",&a);

    printf("%d\n",&a+1);

    char\* p = (char\*)&a;

    printf("%p\n",p+1);

    printf("hello world!\n");

    printf("%d\n",sizeof(a));

    return 0;

}

\_\_attribute\_\_((packed))，这个函数用于设置禁止字节填充，在某些情况下，编译器会为了对齐数据而在结构体成员之间插入一些填充字节，以保证结构体的每个成员都按照某种对齐规则放置。

# 3、linked-list compared with array

对于数组和链表各有各的优势：

1）对于直接访问某个元素，数组显然要更加快捷，时间复杂度为O(1)，而对于linked-list来说时间复杂度为O(n)。

2）对于插入/删除某个特定的元素来说，时间复杂度数组和链表在不同地方时间复杂度不一样。

3）对于内存利用率来说，内存的使用情况链表更高，但也对应有额外数值用于存储下个数据的地址。但链表可变长短，操作性更高。

4）一个数据加一个指向下一个数据的指针称为节点，而只向下一个数据的指针称为链接点。

1.4、linked-list implementation in C

#include <stdio.h>

#include <stdlib.h>

typedef struct \_\_attribute\_\_((packed)) node

{

    int data;

    struct node\* next;

}node1;

int main(int arge,const char\*\* argv)

{

    node1\* a = (node1\*)calloc(sizeof(node1),1);

    a->data = 2;

    a->next = (node1\*)calloc(sizeof(node1),1);

    a->next->data = 4;

    a->next->next = (node1\*)calloc(sizeof(node1),1);

    a->next->next->data = 6;

    a->next->next->next = NULL;

    printf("%d\n",a->data);

    printf("%d\n",(\*a).data);

    printf("%d\n",a->next);

    printf("%d\n",a->next->data);

    printf("%d\n",a->next->next);

    printf("%d\n",a->next->next->data);

    printf("%d\n",a->next->next->next);

    printf("%d\n",sizeof(node1));

    free(a->next->next);

    free(a->next);

    free(a);

}

通过不断的指向下一个结构体，实现数据的存储。对于printf(“%d\n”,a->data)这里a->data是指一个数值，表示通过指针a访问结构体成员data的值，这个值的类型是int，而不是指针，等价于(\*a).data。记得释放内存，和多维数组一样，从里到外依次释放。

用while进行遍历

#include <stdio.h>

#include <stdlib.h>

typedef struct \_\_attribute\_\_((packed)) node

{

    int data;

    struct node\* next;

}node1;

int main(int arge,const char\*\* argv)

{

    node1\* a = (node1\*)calloc(sizeof(node1),1);

    a->data = 2;

    a->next = (node1\*)calloc(sizeof(node1),1);

    a->next->data = 4;

    a->next->next = (node1\*)calloc(sizeof(node1),1);

    a->next->next->data = 6;

    a->next->next->next = (node1\*)calloc(sizeof(node1),1);

    a->next->next->next->data = 0;

    a->next->next->next->next = NULL;

    node1\* temp = a;

    do

    {

        printf("%d\n",temp->data);

        // printf("%d\n",(\*temp).data);

        printf("%d\n",temp->next);

        temp = temp->next;

    }while(temp->next);

    printf("%d\n",sizeof(node1));

    free(a->next->next);

    free(a->next);

    free(a);

}

注意不要移动a指针，他是一个头指针，不能丢，否则就找不到他移动后之前的信息了。头指针永远不能变。

# 4、在链表的开头插入数据：

#include <stdio.h>

#include <stdlib.h>

#include "main.h"

typedef struct node

{

    int data;

    struct node\* next;

}node1;

node1\* head = NULL;

void insert(int x);

void print();

int main(int arge,const char\*\* argv)

{

    printf("please input number about your wanner create linked-list\n");

    int n,x;

    scanf("%d",&n);

    for(int i = 0;i<n;i++)

    {

        printf("enter the number\n");

        scanf("%d",&x);

        insert(x);

        print();

    }

    return 0;

}

void insert(int x)

{

    node1\* temp = (node1\*)malloc(sizeof(node1));

    temp->data = x;

    temp->next = head;

    head = temp;

}

void print()

{

    node1\* temp = head;

    printf("the list is: ");

    while(temp)

    {

        printf("%d ",temp->data);

        // printf("%d\n",temp->next);

        temp = temp->next;

    }

    printf("\n");

}

注意其中对于链表开始插入数据的逻辑关系。防止数据丢失。此时的head为全局变量。

修改，此时head为局部变量，但每个里面都有一个局部变量head。并且注意此时的insert函数的返回值为指针。

#include <stdio.h>

#include <stdlib.h>

#include "main.h"

typedef struct node

{

    int data;

    struct node\* next;

}node1;

node1\* insert(node1\* head,int x);

void print(node1\* head);

int main(int arge,const char\*\* argv)

{

    node1\* head = NULL;

    printf("please input number about your wanner create linked-list\n");

    int n,x;

    scanf("%d",&n);

    for(int i = 0;i<n;i++)

    {

        printf("enter the number\n");

        scanf("%d",&x);

        head = insert(head,x);

        print(head);

    }

    return 0;

}

node1\* insert(node1\* head,int x)

{

    node1\* temp = (node1\*)malloc(sizeof(node1));

    temp->data = x;

    temp->next = head;

    head = temp;

    return head;

}

void print(node1\* head)

{

    printf("the list is: ");

    while(head)

    {

        printf("%d ",head->data);

        // printf("%d\n",temp->next);

        head = head->next;

    }

    printf("\n");

}

对上面算法进一步修改，使得head为main函数中的局部变量，其他都是在这个地址上进行操作。

#include <stdio.h>

#include <stdlib.h>

#include "main.h"

typedef struct node

{

    int data;

    struct node\* next;

}node1;

void insert(node1\*\* head,int x);

void print(node1\* head);

void freememory(node1\* head);

int main(int arge,const char\*\* argv)

{

    node1\* head = NULL;

    printf("please input number about your wanner create linked-list\n");

    int n,x;

    scanf("%d",&n);

    for(int i = 0;i<n;i++)

    {

        printf("enter the number\n");

        scanf("%d",&x);

        insert(&head,x);

        print(head);

    }

    freememory(head);

    return 0;

}

void insert(node1\*\* head,int x)

{

    node1\* temp = (node1\*)malloc(sizeof(node1));

    temp->data = x;

    temp->next = \*head;

    \*head = temp;

}

void print(node1\* head)

{

    printf("the list is: ");

    while(head)

    {

        printf("%d ",head->data);

        // printf("%d\n",temp->next);

        head = head->next;

    }

    printf("\n");

}

void freememory(node1\* head)

{

    node1\* current = head;

    node1\* next;

    while(current)

    {

        next = current->next;

        free(current->next);

        current = next;

    }

}

此时head为main函数中的局部变量，在insert函数中时，传递了指向main函数局部变量head的地址，来进行处理。而对于print函数来说，不需要用head的地址来进一步处理了。当然也可以那么做，但直接传递head给print函数更加省事。最后一段为内存释放语句，采用逐级释放。

# 5、数组转链表、链表转数组输出

#include <stdio.h>

#include <stdlib.h>

typedef struct listed

{

    int val;

    struct listed \*next;

} node;

node\* mergeTwoLists(node\* list1, node\* list2)

{

    if (list1 == NULL)

        return list2;

    else if (list2 == NULL)

        return list1;

    else if (list1->val < list2->val) {

        list1->next = mergeTwoLists(list1->next, list2);

        return list1;

    } else {

        list2->next = mergeTwoLists(list2->next, list1);

        return list2;

    }

}

node\* arraytolisted(int\* a, int size)

{

    node\* list = NULL;

    node\* temp = NULL;

    for (int i = 0; i < size; i++) {

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

        newnode->next = NULL;

        newnode->val = a[i];

        if (list == NULL) {

            list = newnode;

            temp = newnode;

        } else {

            temp->next = newnode;

            temp = temp->next;

        }

    }

    return list;

}

char\* listToArray(node\* list, int\* size)

{

    int count = 0;

    node\* temp = list;

    while (temp != NULL) {

        count++;

        temp = temp->next;

    }

    char\* array = (char\*)malloc(count \* sizeof(char));

    temp = list;

    for (int i = 0; i < count; i++) {

        array[i] = temp->val + '0'; //要将一个整数转换为对应的数字字符，您只需将整数值加上字符 '0' 的ASCII码。

        temp = temp->next;

    }

    \*size = count;

    return array;

}

int main(int argc, char\*\* argv) {

    int a[] = {3, 4, 5, 6};

    int b[] = {4, 5, 6, 7};

    node\* mergedList = mergeTwoLists(arraytolisted(a, sizeof(a) / sizeof(a[0])), arraytolisted(b, sizeof(b) / sizeof(b[0])));

    int size;

    char\* array = listToArray(mergedList, &size);

    printf("%.\*s\n", size, array); // Print the array as string

    // Free dynamically allocated memory

    free(array);

    return 0;

}

# 6、最差、最佳、平均时间复杂度

#include <stdio.h>

#include <stdlib.h>

#include "main.h"

int main(int argc, char\*\* argv)

{

    printf("please input an integer number: \n");

    int n, number;

    scanf("%d", &n);

    printf("please input the number you want to find: \n");

    scanf("%d", &number);

    int\* array = randomnum(n); // 保存返回的指针

    int index = find(number, array, n); // 存储查找结果

    if (index != -1) {

        printf("The number %d is found at index %d.\n", number, index);

    } else {

        printf("The number %d is not found in the array.\n", number);

    }

    free(array); // 在不再需要数组时释放内存

    return 0;

}

int\* randomnum(int n)

{

    int\* array = (int\*)malloc(sizeof(int) \* n);

    if (array == NULL) {

        printf("Memory allocation failed!\n");

        exit(1);

    }

    for (int i = 0; i < n; i++)

    {

        array[i] = i;

    }

    for (int i = n - 1; i > 0; i--)

    {

        int j = rand() % (i + 1);

        int temp = array[i];

        array[i] = array[j];

        array[j] = temp;

    }

    return array;

}

int find(int number, int\* array, int n)

{

    int i = 0;

    while (i < n && array[i] != number)

    {

        i++;

    }

    if (i < n)

    {

        return i; // 找到数字，返回索引

    }

    else

    {

        return -1; // 未找到数字，返回特殊值

    }

}

# 7、列表

main.c

#include <stdio.h>

#include <stdlib.h>

#include <assert.h>

#include "main.h"

#include "happy.h"

int main(int arge, char \*\*argv)

{

    mylist\* list = newMyList();

    arraytolist(list);

    printHELLO\_WORLD();

}

//数组数据存入

void arraytolist(mylist\* nums)

{

    printf("please input your wanner store data!\n");

    int data;

    while(scanf("%d",&data)==1)

    {

        AddMyListhead(nums,data);

    }

    printf("Data stored successfully!\n");

}

//构建列表

mylist\* newMyList()

{

    mylist\* nums = malloc(sizeof(mylist));

    nums->capacity = 10;

    nums->size = 0;

    nums->array = (int\*)malloc(sizeof(int)\*nums->capacity);

    nums->extendRatio = 2;

    return nums;

}

//清空列表

void delMyList(mylist\* nums)

{

    free(nums->array);

    free(nums);

}

//获取列表长度

int MyListSzie(mylist\* nums)

{

    return nums->size;

}

//获取列表容量

int MyListCapacity(mylist\* nums)

{

    return nums->capacity;

}

//访问元素

int GetMyList(mylist\* nums,int index)

{

    assert(index>=0);

    assert(index<nums->size);

    return nums->array[index];

}

//更新元素

void UpdateMyList(mylist\* nums,int index,int data)

{

    assert(index>=0);

    assert(index<nums->size);

    nums->array[index] = data;

}

//列表扩容

void extenCapacity(mylist\* nums)

{

    int newcapacity = nums->capacity \* nums->extendRatio;

    nums->array = (int\*)realloc(nums->array,sizeof(int)\*newcapacity);

}

//在尾部添加元素

void AddMyListtail(mylist\* nums,int data)

{

    if(nums->size == nums->capacity)

    {

        extenCapacity(nums);

    }

    nums->array[nums->size] = data;

    nums->size++;

}

//在中间插入元素

void insert(mylist\* nums,int index,int data)

{

    assert(index>=0);

    assert(index<nums->size);

    if(nums->size == nums->capacity)

    {

        extenCapacity(nums);

    }

    for(int i=nums->size;i>index;--i)

    {

        nums->array[i] = nums->array[i-1];

    }

    nums->array[index] = data;

    nums->size++;

}

//在头部添加元素

void AddMyListhead(mylist\* nums,int data)

{

    if(nums->size == nums->capacity)

    {

        extenCapacity(nums);

    }

    for(int i=nums->size-1;i>0;--i)

    {

        nums->array[i] = nums->array[i-1];

    }

    nums->array[0] = data;

    nums->size++;

}

//删除元素

int removeMyList(mylist\* nums, int index)

{

    assert(index>=0);

    assert(index < nums->size);

    int num = nums->array[index];

    for(int i = index;i<nums->size;i++)

    {

        nums->array[i] = nums->array[i+1];

    }

    nums->size--;

    return num;

}

//列表转数组打印

int \*listtoarray(mylist\* nums)

{

    return nums->array;

}

printHELLO\_WORLD.c

#include <stdio.h>

#include "happy.h"

void printHELLO\_WORLD()

{

    printf("hello world!\n");

}

注意对于结构体可以应用nums.array[i]，也可以用nums->array[i]，对于结构体中的数组成员的内部进行访问。

## 7.1、for循环

for循环的表达式一般如下：

for(表达式1;表达式2;表达式3)

{

表达式4;

}

执行的顺序为：

第一次循环：

首先执行表达式1（一般为初始化语句，只执行一次），再执行表达式2（条件判断语句），判断表达式2的条件，如果符合，则执行表达式4，最后执行3。 如果判断2不符合，则停止执行，且不会执行表达式3.

之后的循环：

首先执行表达式2，如果符合，继续执行表达式4，然后是3。否则停止执行，且不执行表达式3

如此往复，直到不再满足表达式2的条件。执行顺序为：1243，243，243

# 8、VScode+Cmake+gbd

对于Cmakelist.txt

cmake\_minimum\_required(VERSION 3.29.1)

project(xixi VERSION 0.1.0 LANGUAGES C CXX)

include(CTest)

enable\_testing()

#file(GLOB\_RECURSE SOURCES "src/\*.cpp")  # 自动从 src 目录读取所有 .cpp 文件

file(GLOB\_RECURSE SOURCES "src/\*.c")  # 自动从 src 目录读取所有 .c 文件

include\_directories("lib")              # 设置头文件搜索路径为 lib 目录

set(EXECUTABLE\_OUTPUT\_PATH ${CMAKE\_SOURCE\_DIR}/bin)

add\_executable(xixi ${SOURCES})         # 将所有 .cpp 文件或者 .c 文件添加到可执行文件中

set(CPACK\_PROJECT\_NAME ${PROJECT\_NAME})

set(CPACK\_PROJECT\_VERSION ${PROJECT\_VERSION})

include(CPack)

采用这个配置，C++/C通用。

注意此时的.exe文件生成位置在当前目录下的bin文件夹下。

配置文件位src存储.c文件，lib存储头文件。对于C++的使用时仅对于编译，调试文件夹进行更改。

# 9、链表实现栈

main.c

#include <stdio.h>

#include <stdlib.h>

#include <assert.h>

#include <stdbool.h>

#include "main.h"

#include "happy.h"

int main(int arge, char \*\*argv)

{

   LinkedListStack\* node = newLinkedListStack();

   int data;

   while(scanf("%d",&data)==1)//用于检测输入数据是否为整数

   {

    push(node,data);

   }

   printf("Data stored successfully!\n");

   printf("the length of stack is %d\n",node->size);

}

//构造初始栈空间

LinkedListStack\* newLinkedListStack()

{

    LinkedListStack\* nodes = (LinkedListStack\*)malloc(sizeof(LinkedListStack));

    nodes->top = NULL;

    nodes->size = 0;

    return nodes;

}

//出栈-消除栈空间

void DelLinkedListStack(LinkedListStack\* nodes)

{

    while(nodes->top)

    {

        ListNode\* n = nodes->top->next;

        free(nodes->top);

        nodes->top = n;

    }

    free(nodes);

}

//获取栈的长度

int size(LinkedListStack\* nodes)

{

    return nodes->size;

}

//判断栈是否为空

bool isEmpty(LinkedListStack\* nodes)

{

    return nodes->size == 0;

}

//入栈

void push(LinkedListStack\* nodes, int num)

{

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

    node->val = num;

    node->next = nodes->top;

    nodes->top = node;

    nodes->size++;

}

//访问栈顶元素

int peek(LinkedListStack\* nodes)

{

    if(nodes->size == 0)

    printf("the stack is empty!\n");

    return nodes->top->val;

}

//出栈

int pop(LinkedListStack\* nodes)

{

    int val = nodes->top->val;

    ListNode\* node = nodes->top->next;

    nodes->top = nodes->top->next;

    free(node);

    nodes->size--;

    return val;

}

main.h

#ifndef MIAN\_H

#define MAIN\_H

typedef struct main

{

    struct main\* next;

    int val;

}ListNode;

typedef struct node1

{

    ListNode\* top;

    int size;

}LinkedListStack;

LinkedListStack\* newLinkedListStack();

void DelLinkedListStack(LinkedListStack\* nodes);

int size(LinkedListStack\* nodes);

bool isEmpty(LinkedListStack\* nodes);

void push(LinkedListStack\* nodes, int num);

int peek(LinkedListStack\* nodes);

int pop(LinkedListStack\* nodes);

#endif

栈的思想是先进后出，单链表即可实现。