# 二叉树的四种遍历方式

## 算法

二叉树结构如下：

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  | 20 |  |  |  |  |  |  |  |
|  |  |  | 19 |  |  |  |  |  |  |  | 17 |  |  |  |
|  | 16 |  |  |  | 18 |  |  |  | 10 |  |  |  | 15 |  |
| 14 |  | 12 |  | 13 |  |  |  | 9 |  | 7 |  | 6 |  | 8 |

存储结构如下：

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 位置 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| 数值 | 20 | 19 | 17 | 16 | 18 | 10 | 15 | 14 | 12 | 13 | 0 | 9 | 7 | 6 | 8 |

递归实现的思路：

1>先序遍历：先输出当前节点；再递归访问左节点；最后递归访问右节点。

2>中序遍历：先递归访问左节点；无左节点时再输出当前节点；最后递归访问右节点。

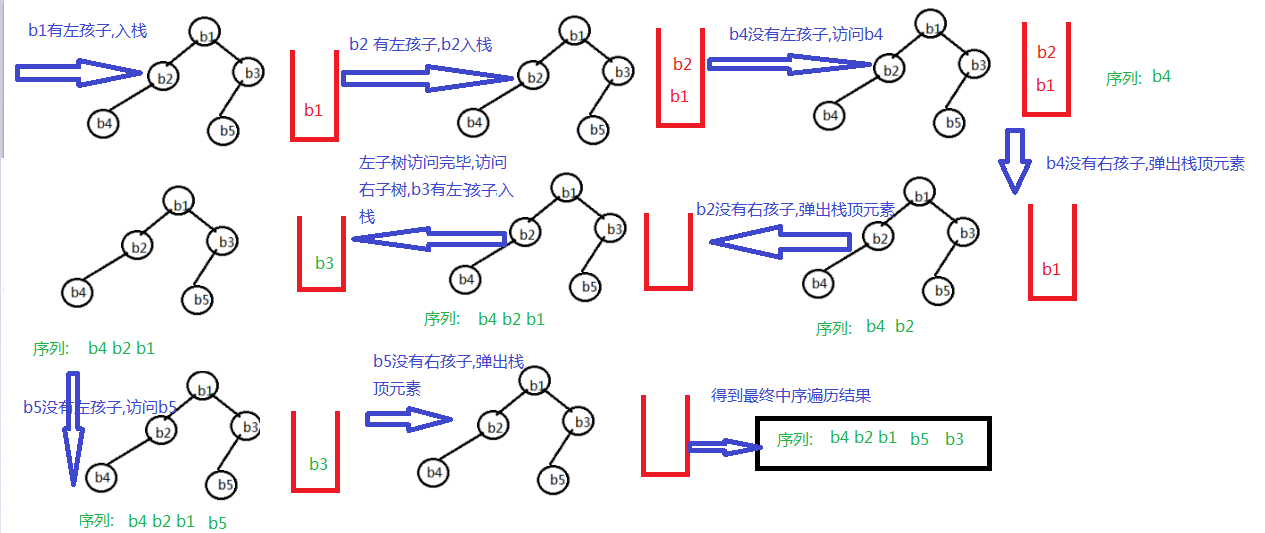
3>后序遍历：先递归访问左节点；再递归访问右节点；最后左右节点均没有时再输出当前节点。

非递归实现的思路：

1>层序遍历：从前往后依次遍历即可。

2>先序遍历：回顾先序遍历递归实现过程：输出当前节点→递归访问左子树→递归访问右子树。非递归实现时，输出当前节点不变，用堆栈缓存左右叶子节点坐标，以此来访问左右子树。但是考虑到堆栈先入后出的结构，所以非递调用的实现过程为：输出当前节点→右叶子节点入栈→左叶子节点入栈→出栈继续遍历（左叶子节点优先）。

3>中序遍历：

结点的所有路径情况： step1： 如果当前结点有左子树，则该结点入栈；如果没有左子树（最左叶子节点），则访问该结点 → step2：如果结点有右子树，则重复step1；如果没有右子树，则回退，让此时的栈顶元素出栈，访问栈顶元素，并访问栈顶元素的右子树，重复step1，2 → strp3：如果栈为空，则表示遍历结束。举个小例子：

4>后序遍历：后序即为"左右根",和先序"根左右"相比可知。先序改为先入栈左叶子节点，即：输出当前节点→左叶子节点入栈→右叶子节点入栈→出栈继续遍历（右叶子节点优先），遍历为"根右左"。再将输出整体反转一下变为"左右根"，即为后序。

## C语言设计

|  |
| --- |
| #include<stdio.h>  #define len 15  int tree[len]={20, 19, 17, 16, 18, 10, 15, 14, 12, 13, 0, 9, 7, 6, 8};  int stack[len]={0}; // 坐标缓存堆栈  int print[len]={0}; // 后序输出缓存堆栈  int stack\_label=0; // 坐标缓存堆栈指针  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* 层序遍历（非递归）  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  int print\_lev\_1(int j)  {  int i=j;  for(;i<len;i++)  {  if(tree[i]!=0)  printf("%d ",tree[i]);  }  return 0;  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* 先序遍历（递 归）  \* 思想："根左右"遍历法。  \* 先输出当前节点；  \* 再递归访问左节点；  \* 最后递归访问右节点。  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  int print\_pre(int j)  {  int i=j;  if(i>=len)  return 0;  printf("%d ",tree[i]); // 先输出当前节点  if(tree[2\*i+1]!=0)  print\_pre(2\*i+1); // 访问左叶子节点  if(tree[2\*i+2]!=0)  print\_pre(2\*i+2); // 访问右叶子节点  return 0;  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* 先序遍历（非递归）  \* 思想：由于堆栈的先入后出性质，所以与递归实现相比，  \* 右叶子节点先入栈，左叶子节点后入栈 。  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  int print\_pre\_1()  {  int i;  while(stack\_label>0) // 缓存区堆栈指针小于等于0结束遍历  {  i=stack[--stack\_label]; // 加载缓存区中坐标  if(i>len) // 当前节点超出范围，换下一个节点重新遍历  continue;  printf("%d ",tree[i]); // 输出当前节点值  if(tree[i\*2+2]!=0) // 右叶子节点非空就入栈  stack[stack\_label++]=i\*2+2;  if(tree[i\*2+1]!=0) // 左叶子节点非空就入栈  stack[stack\_label++]=i\*2+1;  }  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* 中序遍历（递 归）  \* 思想："左根右"遍历法。  \* 先递归访问左节点；  \* 无左节点时再输出当前节点；  \* 最后递归访问右节点。  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  int print\_mid(int j)  {  int i=j;  if(i>=len)  return 0;  if(tree[2\*i+1]!=0)  print\_mid(2\*i+1); // 访问左叶子节点  printf("%d ",tree[i]); // 直到遇到没左叶子节点的节点，就输出当前节点  if(tree[2\*i+2]!=0)  print\_mid(2\*i+2); // 访问右叶子节点  return 0;  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* 中序遍历（非递归）  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  int go\_left(int i)  {  while((2\*i+1<len) && (tree[2\*i+1]!=0))  {  stack[stack\_label]=i;  stack\_label++;  i = 2\*i+1;  }  return i;  }  int print\_mid\_1(int i)  {  //i = go\_left(i);  while((2\*i+1<len) && (tree[2\*i+1]!=0))  {  stack[stack\_label]=i;  stack\_label++;  i = 2\*i+1;  }  while(1)  {  printf("%d ",tree[i]);  if((2\*i+2<len) && (tree[2\*i+2]!=0))  {  // i = go\_left(i\*2+2);  i = i\*2+2;  while((2\*i+1<len) && (tree[2\*i+1]!=0))  {  stack[stack\_label]=i;  stack\_label++;  i = 2\*i+1;  }  }  else if(stack\_label!=0)  {  stack\_label--;  i = stack[stack\_label];  }  else  {  return 0;  }  }  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* 后序遍历（递 归）  \* 思想："左右根"遍历法。  \* 先递归访问左节点；  \* 再递归访问右节点；  \* 最后左右节点均没有时再输出当前节点。  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  int print\_aft(int j)  {  int i=j;  if(i>=len)  return 0;  if(tree[2\*i+1]!=0)  print\_aft(2\*i+1); // 访问左叶子节点  if(tree[2\*i+2]!=0)  print\_aft(2\*i+2); // 访问右叶子节点  printf("%d ",tree[i]); // 左右叶子节点都没有才输出当前节点  return 0;  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* 后序遍历（非递归）  \* 思想：后序即为"左右根",和先序"根左右"相比可知。  \* 先序改为先入栈左叶子节点，遍历为"根右左"，  \* 再将整体反转一下变为"左右根"，即为后序。  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  int print\_aft\_1()  {  int i;  int print\_label=len-1; // 输出缓存堆栈的指针初始化  while(stack\_label>0) // 缓存区堆栈指针小于等于0结束遍历  {  i=stack[--stack\_label]; // 加载缓存区中坐标  if(i>len) // 当前节点超出范围，换下一个节点重新遍历  continue;  print[print\_label--]=tree[i]; // 将当前节点放入缓存数组（反序）  if(tree[i\*2+1]!=0) // 左叶子节点非空就入栈  stack[stack\_label++]=i\*2+1;  if(tree[i\*2+2]!=0) // 右叶子节点非空就入栈  stack[stack\_label++]=i\*2+2;  }  for(print\_label=0;print\_label<len;print\_label++)  {  if(print[print\_label]!=0) // 顺序输出非空节点  printf("%d ",print[print\_label]);  }  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \*  \* 主程序  \*  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  int main()  {  printf("层序遍历（非递归）：");  print\_lev\_1(0);  printf("\n \n");  printf("先序遍历（递 归）：");  print\_pre(0);  printf("\n");    printf("先序遍历（非递归）：");  stack\_label=0; // 堆栈指针初始化  stack[stack\_label++]=0; // 先将根节点坐标入栈  print\_pre\_1();  printf("\n \n");    printf("中序遍历（递 归）：");  print\_mid(0); // 堆栈指针初始化  printf("\n");  printf("中序遍历（非递归）：");  stack\_label=0;  print\_mid\_1(0);  printf("\n \n");  printf("后序遍历（递 归）：");  print\_aft(0);  printf("\n");  printf("后序遍历（非递归）：");  stack\_label=0; // 堆栈指针初始化  stack[stack\_label++]=0; // 先将根节点坐标入栈  print\_aft\_1();  printf("\n \n");    return 0;  } |

## 汇编语言设计

用LoongIDE新建一个工程。

移除core文件夹下的bsp\_start.c源文件。

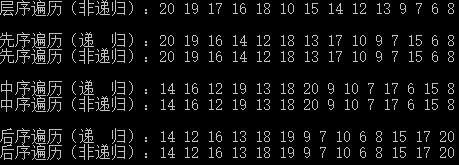
移除main.c源文件。

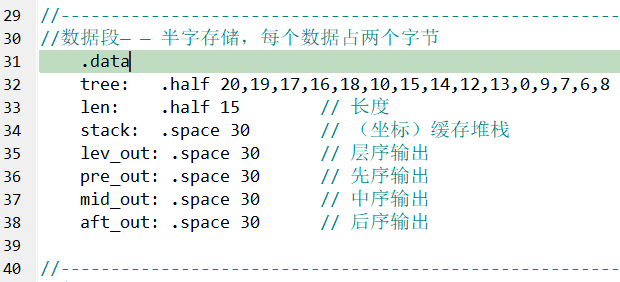
添加bsp\_start.S源文件。

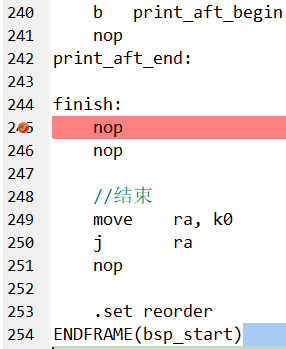
在bsp\_start.S下添加如下代码：

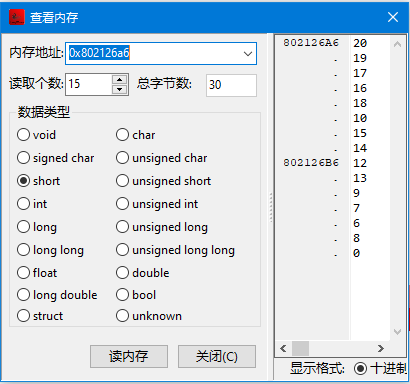
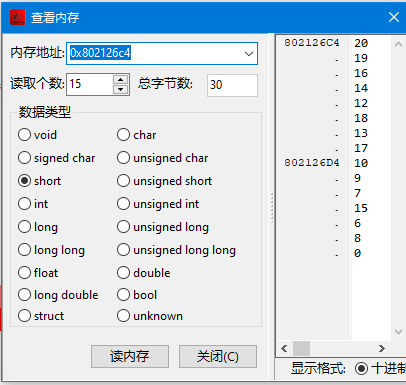
|  |
| --- |
| /\*  \* bsp\_start.S  \*  \* created: 2022/2/14  \* author: Li TianLing  \*/  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \*  \* v0 存储器寻址辅助寄存器(偏移地址)  \*  \* a0->a3 数据区基地址  \*  \* s0 当前节点i  \* s1 左叶子节点2\*1+1  \* s2 右叶子节点2\*i+2  \*  \* t0 长度len （常数）  \* t1 （坐标）缓存区指针  \* t2 输出指针  \*  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  #include "regdef.h"  #include "cpu.h"  #include "asm.h"  //-----------------------------------------------------------------------------  //数据段——半字存储，每个数据占两个字节  .data  tree: .half 20,19,17,16,18,10,15,14,12,13,0,9,7,6,8  len: .half 15 // 长度  stack: .space 30 // （坐标）缓存堆栈  lev\_out: .space 30 // 层序输出  pre\_out: .space 30 // 先序输出  mid\_out: .space 30 // 中序输出  aft\_out: .space 30 // 后序输出  //-----------------------------------------------------------------------------  //代码段  .text  FRAME(bsp\_start,sp,0,ra)  .set noreorder  move k0, ra /\* 返回地址 \*/  la a0,lev\_out  la a1,pre\_out  la a2,mid\_out  la a3,aft\_out  la t0,len // 将长度len加载到t0  lh t0,(t0)  mul t0,2  /\* 层序遍历 \*/  la t2,0 // 输出指针初始化  print\_lev\_begin:  la s0,0 // 先从根节点开始遍历  for\_lev\_begin:  bge s0,t0,for\_lev\_end // 节点超出范围结束  nop  if\_pre\_begin:  lh s3,tree(s0) // 将tree[i]加载到s3  beq s3,zero,if\_pre\_end // 剔除空元素  nop  sh s3,lev\_out(t2) // 否则将其导出到lev\_out  add t2,2  if\_pre\_end:  add s0,2 // 遍历下一个节点  b for\_lev\_begin  nop  for\_lev\_end:  print\_lev\_end:  /\* 前序遍历 \*/  la t1,0 // （坐标）缓存区指针初始化  sh zero,stack(t1) // 先将根节点坐标入栈  add t1,2  la t2,0 // 输出指针初始化  print\_pre\_begin:  blez t1,print\_pre\_end // 缓存区堆栈指针小于等于0结束遍历  nop  sub t1,2 // 加载缓存区中坐标  lh s0,stack(t1)  bge s0,t0,print\_pre\_begin // 当前节点超出范围，换下一个节点重新遍历  nop  lh s3,tree(s0) // 将tree[i]加载到s3  sh s3,pre\_out(t2) // 将其导出到pre\_out  add t2,2  mul s2,s0,2 // 右叶子节点坐标2\*i+4（half类型）  add s2,4  lh s3,tree(s2) // 将tree[右]加载到s3  beq s3,zero,pre\_jump\_right // 为空不入栈  nop  sh s2,stack(t1) // 将右叶子节点坐标入栈  add t1,2  pre\_jump\_right:  mul s1,s0,2 // 左叶子节点坐标2\*i+2（half类型）  add s1,2  lh s3,tree(s1) // 将tree[左]加载到s3  beq s3,zero,pre\_jump\_left // 为空不入栈  nop  sh s1,stack(t1) // 将左叶子节点坐标入栈  add t1,2  pre\_jump\_left:  b print\_pre\_begin  nop  print\_pre\_end:  /\* 中序遍历 \*/  la t1,0 // （坐标）缓存区指针初始化  la t2,0 // 输出指针初始化  la s0,0 // 当前节点从根节点开始  print\_mid\_begin:  while\_go\_left\_1\_begin:  mul s1,s0,2 // 左节点坐标2\*i+2  add s1,2  bge s1,t0,while\_go\_left\_1\_end  nop // 当前节点超出范围，结束  lh s3,tree(s1) // 将tree[左]加载到s3  beq s3,zero,while\_go\_left\_1\_end  nop // 为空结束  sh s0,stack(t1) // 节点坐标i入栈  add t1,2  mul s0,2 // i=i\*2+2访问左节点  add s0,2  b while\_go\_left\_1\_begin  nop  while\_go\_left\_1\_end:  while\_1\_begin:  lh s3,tree(s0) // 将tree[i]加载到s3  sh s3,mid\_out(t2) // 输出  add t2,2  if\_mid\_begin:  mul s2,s0,2 // 右节点坐标2\*i+4  add s2,4  bge s2,t0,else\_if\_mid // 当前节点超出范围跳转  nop  lh s3,tree(s2) // 将tree[右]加载到s3  beq s3,zero,else\_if\_mid  nop // 为空结束  mul s0,2 // i=i\*2+4访问右节点  add s0,4  while\_go\_left\_2\_begin:  mul s1,s0,2 // 左节点坐标2\*i+2  add s1,2  bge s1,t0,while\_go\_left\_2\_end  nop // 当前节点超出范围，结束  lh s3,tree(s1) // 将tree[左]加载到s3  beq s3,zero,while\_go\_left\_2\_end  nop // 为空结束  sh s0,stack(t1) // 节点坐标i入栈  add t1,2  mul s0,2 // i=i\*2+2访问左节点  add s0,2  b while\_go\_left\_2\_begin  nop  while\_go\_left\_2\_end:  b if\_mid\_end // 退出if  nop  else\_if\_mid:  beq t1,zero,else\_mid // 堆栈为空跳转  nop  sub t1,2  lh s0,stack(t1)  b if\_mid\_end // 退出if  nop  else\_mid:  b while\_1\_end  nop  if\_mid\_end:  b while\_1\_begin  nop  while\_1\_end:  print\_mid\_end:  /\* 后序遍历 \*/  la t1,0 // （坐标）缓存区指针初始化  sh zero,stack(t1) // 先将根节点坐标入栈  add t1,2  move t2,t0 // 输出指针初始化到最后一格  sub t2,2  print\_aft\_begin:  blez t1,print\_aft\_end // 缓存区堆栈指针小于0结束遍历  nop  sub t1,2 // 加载缓存区中坐标  lh s0,stack(t1)  bge s0,t0,print\_aft\_begin // 当前节点超出范围重新遍历  nop  lh s3,tree(s0) // 将tree[i]加载到s3  sh s3,aft\_out(t2) // 将其导出到pre\_out  sub t2,2  mul s1,s0,2 // 左叶子节点坐标2\*i+2（half类型）  add s1,2  lh s3,tree(s1) // 将tree[左]加载到s3  beq s3,zero,aft\_jump\_left // 为空不入栈  nop  sh s1,stack(t1) // 将左叶子节点坐标入栈  add t1,2  aft\_jump\_left:  mul s2,s0,2 // 右叶子节点坐标2\*i+4（half类型）  add s2,4  lh s3,tree(s2) // 将tree[右]加载到s3  beq s3,zero,aft\_jump\_right // 为空不入栈  nop  sh s2,stack(t1) // 将右叶子节点坐标入栈  add t1,2  aft\_jump\_right:  b print\_aft\_begin  nop  print\_aft\_end:  finish:  nop  nop  //结束  move ra, k0  j ra  nop  .set reorder  ENDFRAME(bsp\_start) |

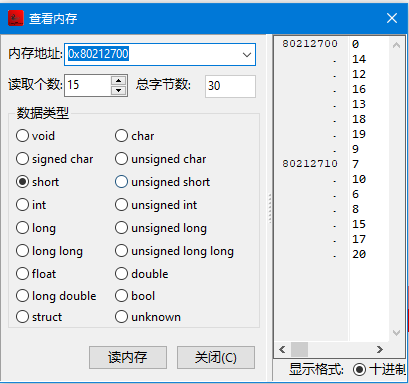
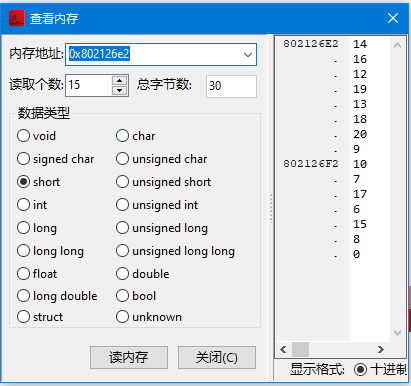
## 调试运行

C语言程序运行结果如下：

初始数据如下所示：

如下图所示，在finish后打上断点：

联合调试后，双击a0寄存器，查看层序遍历结果，双击a1寄存器，查看先序遍历结果：

双击a2寄存器，查看中序遍历结果，双击a3寄存器，查看后序遍历结果：