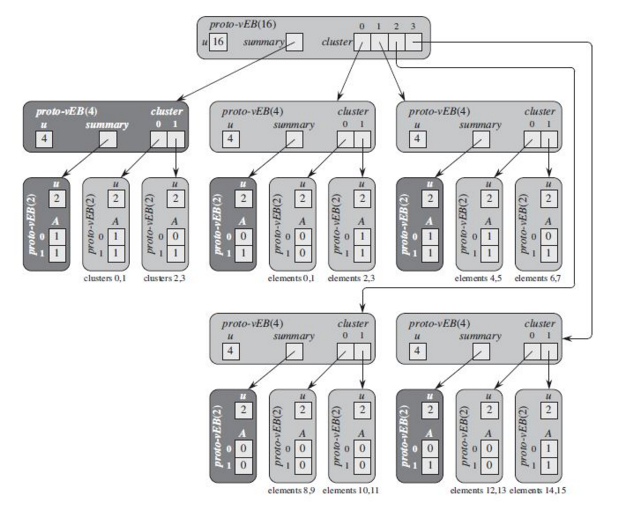
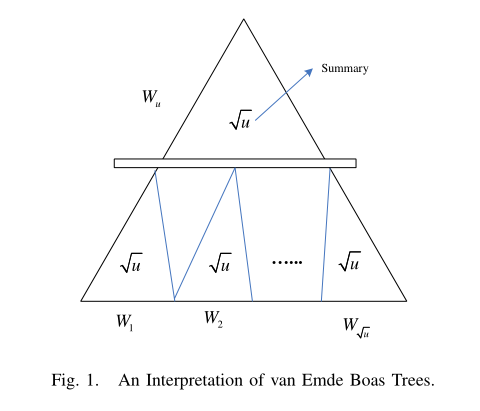
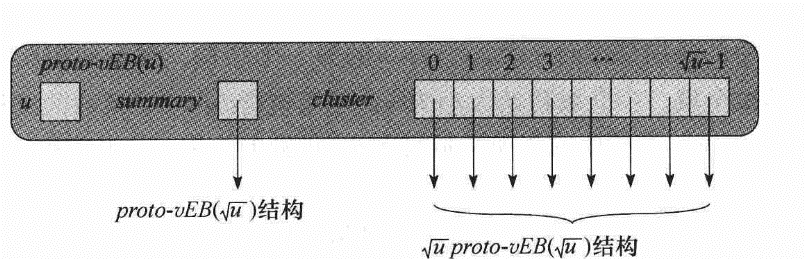
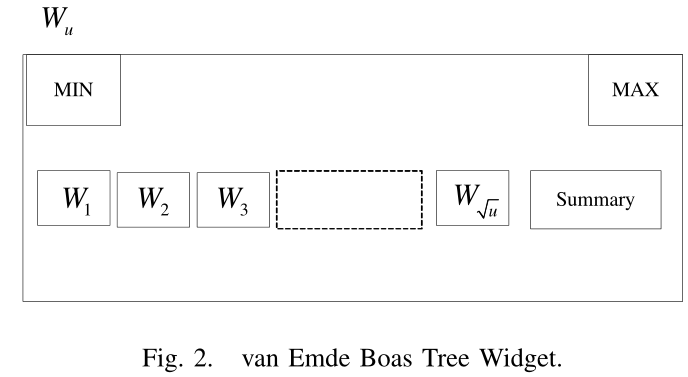
1. vEB树结构图及其说明如下：

处理大小为 u 的全域，创建 √u + 1 个子结构。它们以相同的方式递归构造（并且它们本身就是 van Emde Boas 结构）。其中右图是算法导论中以u = 16为例时得到的proto-vEB(16)结构图，两者结合分析。

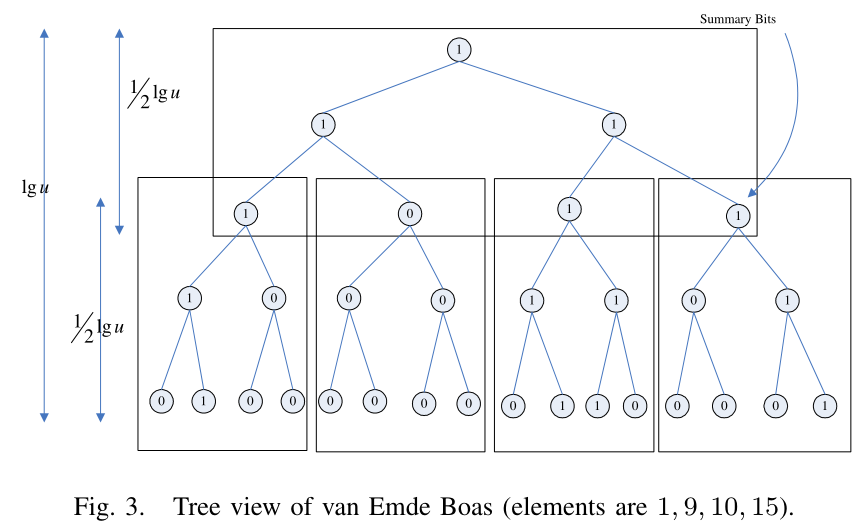


van Emde Boas (vEB) 树可以描述为“控件”树，一个控件的结构图如下，也即是上图中右图的子控件。



1. vEB树的解释：

Van Emde Boas树可以用完全二叉树直观地解释，如图3所示。考虑一棵高h的完全二叉树，那么树中有2h −1个节点。把整棵树看作是一个控件，则Summary是上面的子树，而下面的子树是它的子控件。这时，只需要访问the summary或子树之一就可以找到后继者或前驱者，遵循亚对数搜索时间。



1. vEB树的核心代码如下：

class vEBTree

{

private:

//min不出现在簇和summary中，插入空树和从只有一个元素的树中删除就只需要 O(1) 时间

int max;//最大值

int min;//最小值

int u;//全域

//当 u 为2的奇数次幂 根号下u不为整数

vEBTree\* summary;// v=2^( [log2(u)/2] ) log2(u)/2向下取整 例：u=8 得到 2 summary指向一个全域大小 u/v的vBETree

vEBTree\*\* cluster;//簇指针， 有 u/v个簇，每个簇全域为v 储存[i\*v , (i+1)\*v-1]元素信息

public:

int MIN();//最小值

int MAX();//最大值

int extractmin();//删除并返回VEB树中的the minimum priority key value

vEBTree(int u\_);//递归构造vEB树

int high(int x);//x在第几个簇中

int low(int x);//x在所在簇中下标是几

int index(int x, int y);//x为第几个簇，y为簇中的下标，返回this vEBTree中x的位置

bool Member(int x); //是否包含元素x

int successor(int x); //x不一定在集合中 求后继

int extractsucc(int x); //删除并返回VEB树中大于x的下一个最小priority key value

int predecessor(int x); //前驱

void Insert(int x); //将该the priority key x插入到VEB树中

void Insert\_Empty(int x); //min值不出现在簇中 可在O（1）时间完成

void Delete(int x);

void Out();

};

其中，要测试的操作为，详细代码见程序：

void Insert(int x)//将该the priority key x插入到VEB树中

int extractmin();//删除并返回VEB树中的the minimum priority key value

int successor(int x); //x不一定在集合中 求后继

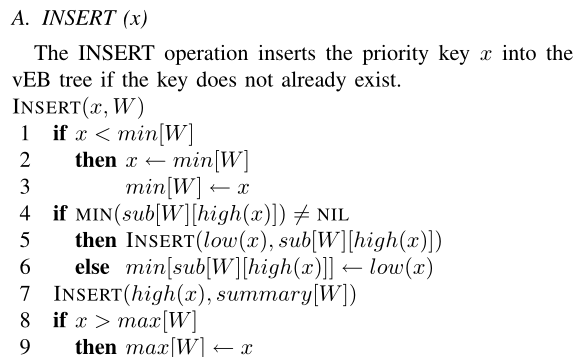
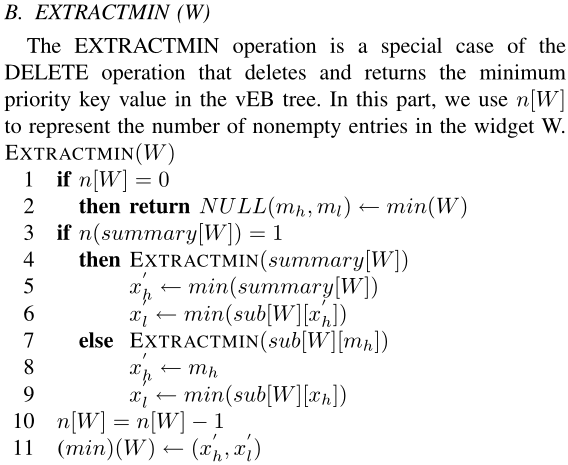
int extractsucc(int x); //删除并返回VEB树中大于x的下一个最小priority key value

1. INSERT(x):

在INSERT操作中，每个二进制数据平均分为两部分，较高部分和较低部分。有两个子结构，分别对应于较高部分和较低部分。对于每个循环，确定新数据属于哪个子结构。按照此步骤操作，直到每个部分都只包含一位。

1. EXTRACTMIN():

因为我们总是跟踪van Emde Boas树的最小值，所以这个操作可以在固定的时间内完成。

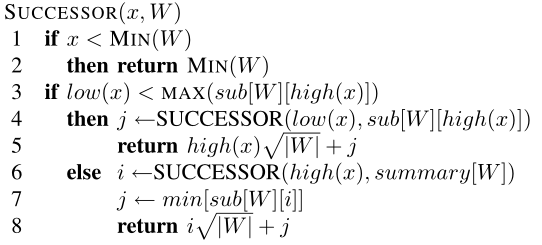
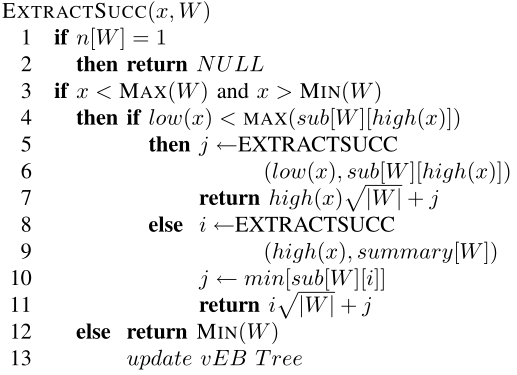
1. SUCCESSOR (x)：

VEB树自然支持后继操作，这在二进制堆或流水线堆上很难实现。这可以按照与查找当前条目相同的规则查找下一个操作条目。如果没有这样的条目，则我们访问的数据是整个VEB树结构中的最后一个。

1. EXTRACTSUCC (x, W)：

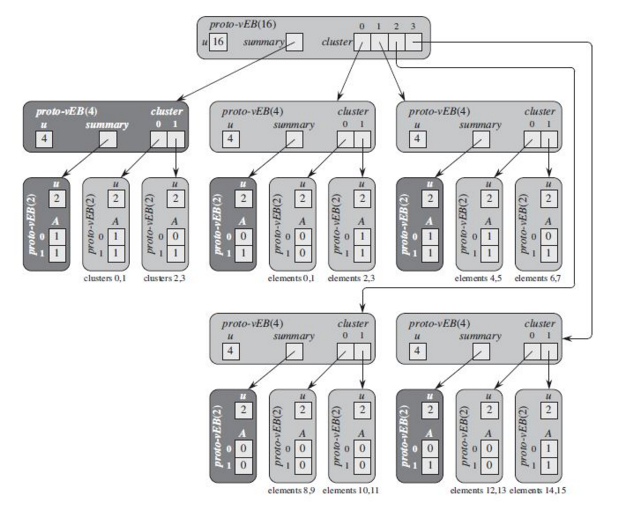
找到并移除具有大于*x*的最小值的条目。如果没有这样的条目，并且VEB树除了我们正在使用的当前数据之外不为空，则提取该树的最小数据。

对于high(x) 和 low(x)，可以假设 x 有 w 位。然后将 x 分成两部分：high(x) 和 low(x)，每部分有 w/2 位。例如设 x = 11001001，则拆分为 high(x) = 1100 and low(x) = 1001。

1. vEB树测试：

直接采用类似算法导论311页的proto-vEB(16)结构进行测试，然后采用随机数生成0或1初始化vEB树。



创建vEB树：改变universe 的值即可改变u，其余部分均用随机数

int universe = 16;

vEBTree vEB(universe);//创建u = universe的vEB树

测试INSERT():

srand((unsigned)time(NULL));

std::cout << "vEB树初始化的输入如下：\n";

for (int i = 0; i < universe; i++)

{

int j = rand() % 2;

if (j)

{

vEB.Insert(i);//elements i

cout << "A[" << i << "] = " << j << "\t";

}

else

{

cout << "A[" << i << "] = " << j << "\t";

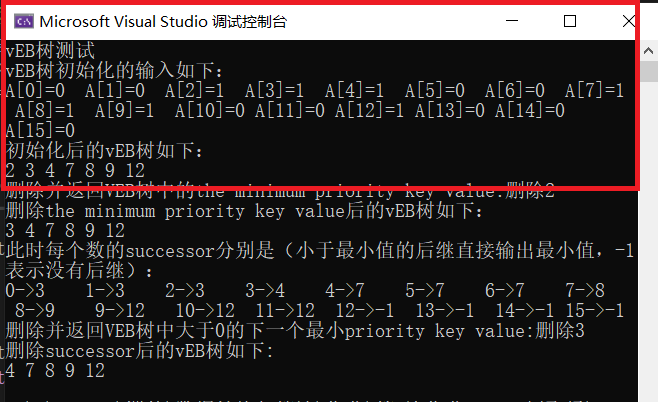
continue;

}

}

cout << endl << "初始化后的vEB树如下：" << endl;

vEB.Out();

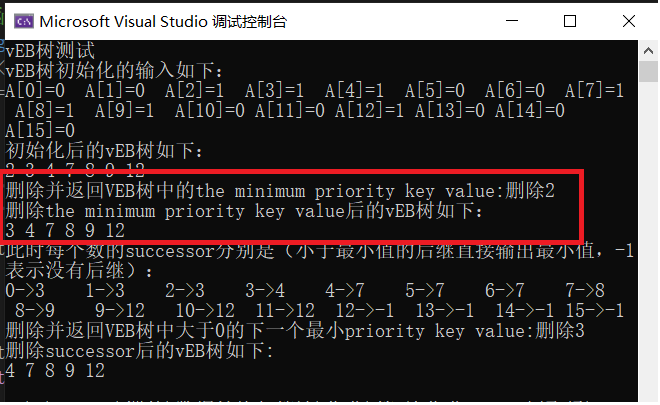


测试EXTRACTMIN():

cout << "删除并返回VEB树中的the minimum priority key value:删除" << vEB.extractmin() << endl;

cout << "删除the minimum priority key value后的vEB树如下：" << endl;

vEB.Out();



测试SUCCESSOR (x):

cout << "此时每个数的successor分别是（小于最小值的后继直接输出最小值，-1表示没有后继）：" << endl;

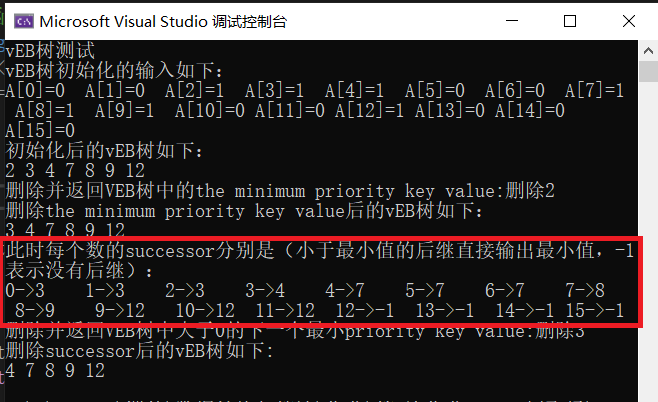
for (int i = 0; i < universe; i++)

{

cout << i << "->" << vEB.successor(i) << "\t";

}

cout << endl;



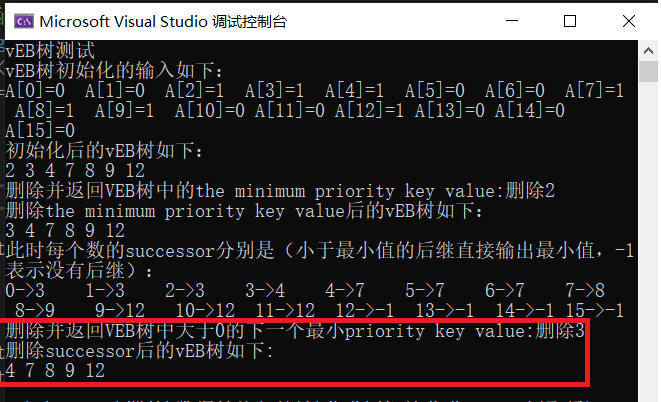
测试EXTRACTSUCC (x, W):

int r = rand() % 16;

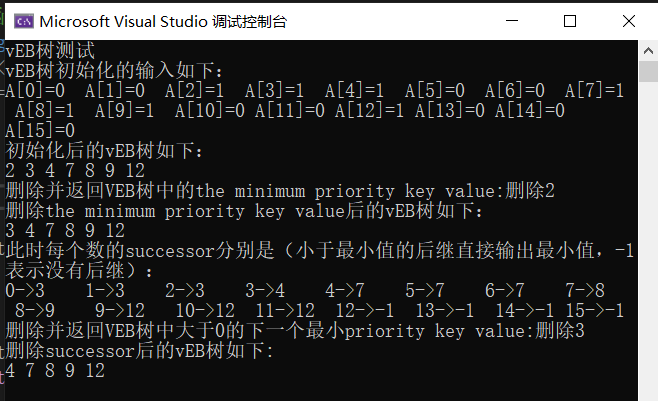
cout << "删除并返回VEB树中大于" << r << "的下一个最小priority key value:删除" << vEB.extractsucc(r) << endl;

cout << "删除successor后的vEB树如下:" << endl;

vEB.Out();

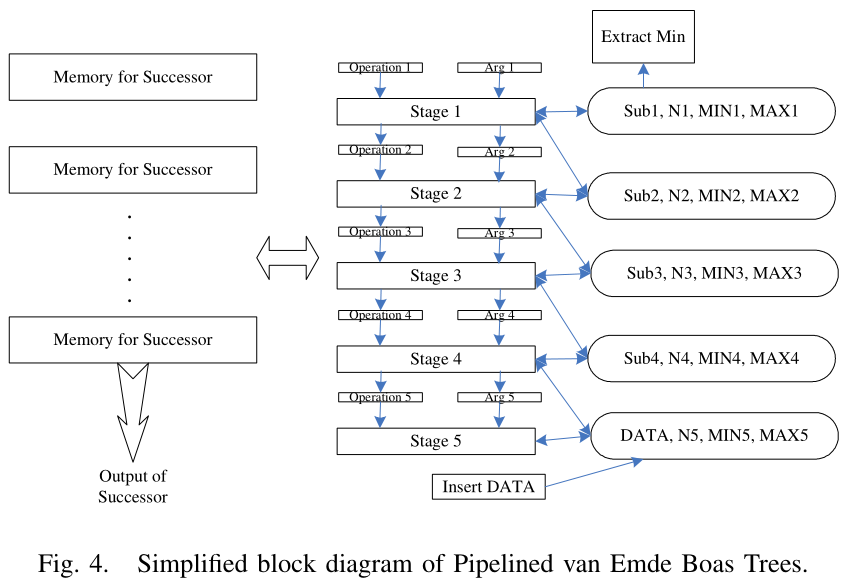


总结果如下：



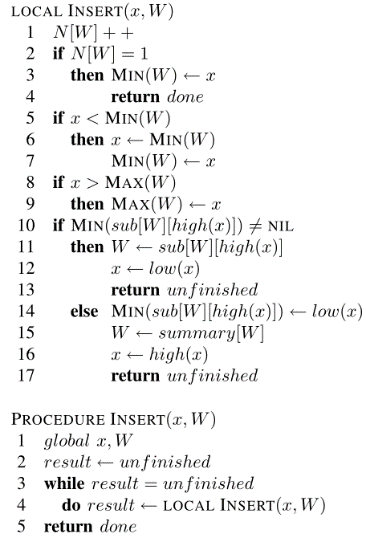
1. pipelined van Emde Boas Tree：

为了获得流水线算法，必须确保没有两个操作同时在同一操作区内，操作区即操作可以访问的内存区。通过使用MIN和MAX来跟踪每个控件的最小值和最大值，使用N来表示每个控件中的元素数量，然后为每一种操作的LOCAL版本对应流水线阶段，流水线阶段从第一级到最后一级的遍历如图：

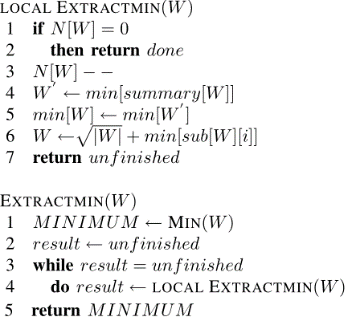


这样就可以让所有操作的每一步都划分为不同的操作区，在开始第一个操作之后，可能会在第一次操作结束之后再过一段时间开始另一个操作。从而构造出一个流水线的van Emde Boas树。流水线版本的各个操作如下：

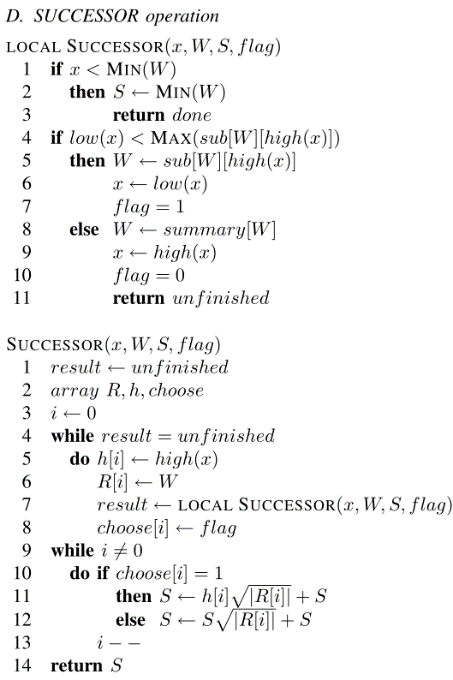
1. INSERT(x):

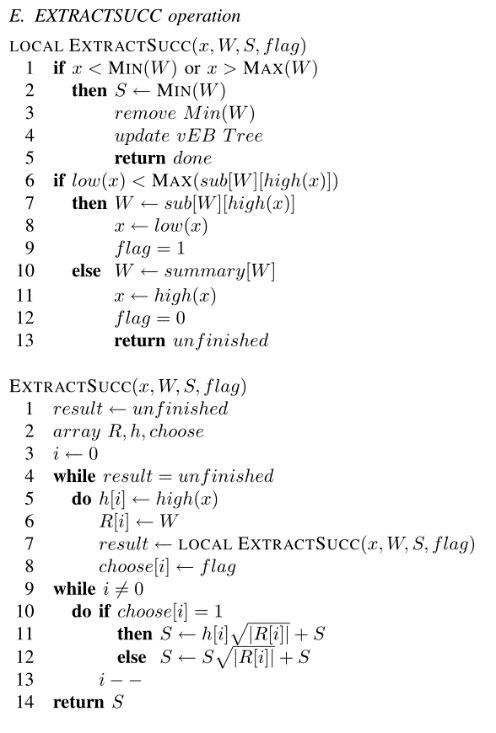
 INSERT操作的流水线版本的关键思想是：没有任何递归调用，只是使用内存来跟踪每个操作的位置及其结果。因此，一次插入操作将仅限于局部区域，即一个操作区，这使得流水线的想法成为可能。

1. EXTRACTMIN():

流水线EXTRACTMIN操作中，没有递归调用。内存被用来记录操作的位置。如果在此操作开始时树为空，则算法将立即终止。

1. SUCCESSOR (x)：

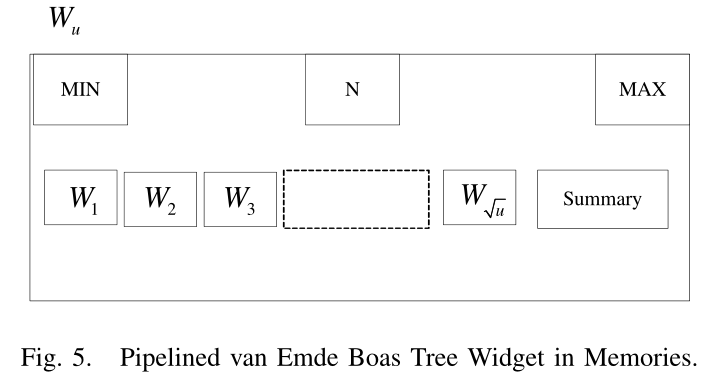
the pipelined SUCCESSOR operation中没有递归调用。内存被用来记录操作的位置。为了输出我们找到的后继的值，我们还必须跟踪我们在寻找当前值的继任者时所经历的路径。

1. EXTRACTSUCC (x, W)：

流水线EXTRACTSUCC操作中也没有递归调用，这一点也是正确的。内存用于跟踪操作的内存位置。如果没有这样的后继条目，并且VEB树不是空的，则除了我们当前正在使用的数据，提取该树中最小的数据。

1. pipelined van Emde Boas Tree的内存管理：

必须使用两个time slots来完成一个阶段，因为 EXTRACTMIN 操作可能需要访问下一阶段的内存。当一个控件的条目被提取时，相应的子控件的条目将需要替换它。给定两个时隙（time slots）可以保证没有两个连续的 EXTRACTMIN 操作同时访问同一个时隙。每个内存的结构如图：



可以看出和普通的vEB树相比，多出了表示每个控件中的元素数量的N，在除最后一个之外的所有存储器中，都存储了MIN、MAX和N。

阶段1的内存大小为 O(u1/2)。第 2 阶段为 O(u3/4)，第 3 阶段为 O(u7/8)，第 4 阶段为 O(u15/16)。Fig.4.示例中的最后阶段，即第 5 阶段，是大小为 O(*u*)。

Fig.4.的左侧是用于后继的内存，该结构中同时允许有（log log u）个操作，（l​​og log u）个阶段。所以内存矩阵的大小是 O(log log u2) 的数量级。实际上，由于在某些应用程序中不需要查找后继的操作，因此我们可以在没有此内存的情况下实现更简单的流水线结构。

最后是Fig.4.的operation部分。首先，输入operation 1。经过一个time slot后，所有参数将被传送到下一个处理器。为了避免 EXTRACTMIN 操作可能发生的冲突，stage 1 中不会输入新的操作。再经过一个time slot，当操作1移动到 stage 3 时，可以将新的操作 2 输入到 stage 1。这样，我们可以确保在这个流水线结构中不会发生碰撞。此结构所需的处理器数量为 log log u。给定 1 GB 的universe，我们将需要 10 个并行工作的处理器。此结构中支持的操作量为 log log u/2。从而实现了恒定时间操作。

更好的方法是将每个操作分成三个时钟周期。它们是： a) 从memory中读取； b) 比较next stage memory中的两个值以找到最小值； c) 将此最小值写入当前阶段的memory。从而操作可以以每三个周期进行一次或多次操作的速率进行。不是每两个周期一个操作，是因为可能有连续的 EXTRACTMIN 操作。但是，我们需要memory来支持每个周期两个条目的读取吞吐量，以及每个周期一个条目的额外写入吞吐量。