|  |
| --- |
| #include <stdio.h> |
| #include <stdlib.h> |
| #include <string.h> |
| #include "Fibo.h" |
|  |
|  |
| FiboHeap::FiboHeap() |
| { |
| m\_pRootHeap = NULL; |
| m\_pMinHeap = NULL; |
| } |
|  |
| FiboHeap::~FiboHeap() |
| { |
| stHeapNode\* pNode = m\_pRootHeap; |
|  |
| while (pNode != NULL) |
| { |
| stHeapNode\* pNextNode = pNode->pRight; |
|  |
| m\_fnDeleteTree(pNode); |
| pNode = pNextNode; |
| } |
| } |
|  |
| void FiboHeap::Insert(stHeapInfo hpInfo) |
| { |
| stHeapNode\* pNewNode = m\_fnNewNode(hpInfo); |
|  |
| m\_fnAddToRootHeap(pNewNode); |
| } |
|  |
| void FiboHeap::ExtractMin(stHeapInfo\* hpInfo) |
| { |
| if (m\_pMinHeap == NULL) return; |
|  |
| \*hpInfo = m\_pMinHeap->hpInfo; |
|  |
| // Move the children into root |
| m\_fnLevelUp(m\_pMinHeap); |
|  |
| // Remove the minimum node |
| m\_fnRemoveNode(m\_pMinHeap); |
| m\_pMinHeap = NULL; |
|  |
| // Re-assign a new one for m\_pMinHeap |
| m\_fnAssignMinNode(); |
|  |
| // Consolidate the heap trees |
| m\_fnConsolidate(); |
| } |
|  |
| void FiboHeap::DumpHeap() |
| { |
| stHeapNode\* pNode = m\_pRootHeap; |
|  |
| while (pNode != NULL) |
| { |
| printf("------------------------\n"); |
| if (pNode == m\_pMinHeap) |
| { |
| printf("\*"); |
| } |
| m\_fnDumpTree(pNode); |
| pNode = pNode->pRight; |
| } |
| } |
|  |
| void FiboHeap::m\_fnAssignMinNode() |
| { |
| stHeapNode\* pCheckNode = m\_pRootHeap; |
|  |
| while (pCheckNode != NULL) |
| { |
| if (m\_pMinHeap == NULL || m\_pMinHeap->hpInfo.nKey > pCheckNode->hpInfo.nKey) |
| { |
| m\_pMinHeap = pCheckNode; |
| } |
|  |
| pCheckNode = pCheckNode->pRight; |
| } |
| } |
|  |
| void FiboHeap::m\_fnRemoveNode(stHeapNode\* pNode) |
| { |
| if (pNode == NULL) return; |
|  |
| if (pNode == m\_pRootHeap) |
| { |
| m\_pRootHeap = m\_pRootHeap->pRight; |
| } |
| if (pNode->pLeft)  pNode->pLeft->pRight = pNode->pRight; |
| if (pNode->pRight) pNode->pRight->pLeft = pNode->pLeft; |
| delete pNode; |
| } |
|  |
| void FiboHeap::m\_fnLevelUp(stHeapNode\* pNode) |
| { |
| for (int i = 0; i < pNode->nChild; i++) |
| { |
| stHeapNode\* pChildNode = pNode->ppChild[i]; |
|  |
| m\_fnAddToRootHeap(pChildNode); |
| pChildNode->pParent = NULL; |
| pNode->ppChild[i] = NULL; |
| } |
|  |
| pNode->nChild = 0; |
| } |
|  |
| void FiboHeap::m\_fnAddToRootHeap(stHeapNode\* pNode) |
| { |
| // First time |
| if (m\_pRootHeap == NULL) |
| { |
| m\_pRootHeap = pNode; |
| m\_pMinHeap = pNode; |
| return; |
| } |
|  |
| // Add to the most left of the root heap |
| pNode->pRight = m\_pRootHeap; |
| m\_pRootHeap->pLeft = pNode; |
| m\_pRootHeap = pNode; |
|  |
| // Check the minimum heap |
| if (m\_pMinHeap->hpInfo.nKey > pNode->hpInfo.nKey) |
| { |
| m\_pMinHeap = pNode; |
| } |
| } |
|  |
| stHeapNode\* FiboHeap::m\_fnNewNode(stHeapInfo hpInfo) |
| { |
| stHeapNode\* pNode = new stHeapNode; |
|  |
| pNode->pLeft    = NULL; |
| pNode->pRight   = NULL; |
| pNode->pParent  = NULL; |
| pNode->hpInfo   = hpInfo; |
| pNode->nDegree  = 0; |
| pNode->nChild   = 0; |
|  |
| memset(pNode->ppChild, 0, sizeof(pNode->ppChild)); |
|  |
| return pNode; |
| } |
|  |
| void FiboHeap::m\_fnConsolidate() |
| { |
| // Make sure the degree of each tree is unique |
| stHeapNode\* ppDegreeNode[MAX\_DEGREES]; |
| stHeapNode\* pNode = m\_pRootHeap; |
|  |
| memset(ppDegreeNode, 0, sizeof(ppDegreeNode)); |
| while (pNode != NULL) |
| { |
| if (ppDegreeNode[pNode->nDegree] == NULL) |
| { |
| ppDegreeNode[pNode->nDegree] = pNode; |
| pNode = pNode->pRight; |
| } |
| else    // merge the two trees |
| { |
| stHeapNode\* pPreNode = ppDegreeNode[pNode->nDegree]; |
|  |
| if (pPreNode->hpInfo.nKey > pNode->hpInfo.nKey) |
| { |
| m\_fnAttachTree(pPreNode, pNode); |
| } |
| else |
| { |
| m\_fnAttachTree(pNode, pPreNode); |
| } |
|  |
| // Reset the search |
| memset(ppDegreeNode, 0, sizeof(ppDegreeNode)); |
| pNode = m\_pRootHeap; |
| } |
| } |
| } |
|  |
| void FiboHeap::m\_fnAttachTree(stHeapNode\* pFromNode, stHeapNode\* pToNode) |
| { |
| if (pFromNode == m\_pRootHeap) |
| { |
| m\_pRootHeap = m\_pRootHeap->pRight; |
| } |
|  |
| // Break the link of pFromNode |
| if (pFromNode->pLeft)  pFromNode->pLeft->pRight = pFromNode->pRight; |
| if (pFromNode->pRight) pFromNode->pRight->pLeft = pFromNode->pLeft; |
| pFromNode->pLeft = NULL; |
| pFromNode->pRight = NULL; |
|  |
| // Move the pFromNode under the pToNode |
| pToNode->ppChild[ pToNode->nChild ] = pFromNode; |
| pFromNode->pParent = pToNode; |
| pToNode->nChild++; |
| pToNode->nDegree++; |
| } |
|  |
| void FiboHeap::m\_fnDumpTree(stHeapNode\* pNode) |
| { |
| printf("(%d)\n", pNode->hpInfo.nKey); |
| for (int i = 0; i < pNode->nChild; i++) |
| { |
| stHeapNode\* pChildNode = pNode->ppChild[i]; |
|  |
| printf("  %d", pChildNode->hpInfo.nKey); |
| } |
| printf("\n"); |
|  |
| for (i = 0; i < pNode->nChild; i++) |
| { |
| stHeapNode\* pChildNode = pNode->ppChild[i]; |
| m\_fnDumpTree(pChildNode); |
| } |
| } |
|  |
| void FiboHeap::m\_fnDeleteTree(stHeapNode\* pNode) |
| { |
| if (pNode == NULL) return; |
|  |
| for (int i = 0; i < pNode->nChild; i++) |
| { |
| stHeapNode\* pChildNode = pNode->ppChild[i]; |
| m\_fnDeleteTree(pChildNode); |
| } |
|  |
| delete pNode; |
| } |