Algorithm 1: Binomial Link

```
Input: node: BinNode, child: BinNode
Output: The root of the binomial heap after linking
Function BinomialLink(node, child):
    if node is NULL then
        return child;
    if child is NULL then
        return node;
    if node's degree < child's degree then
        return BinomialLink(child, node);
    child's parent ← node;
    child's sibling ← node's child;
    node's child ← child;
    node's degree++;
    return node;
```

Algorithm 2: Binomial Merge

```
Input: node1: BinNode, node2: BinNode
Output: The merged binomial heap
Function BinomialMerge(node1, node2):
    // First step, merge them in increasing order;
    BinNode pointer1 ← node1, pointer2 ← node2;
    BinNode begin ← NULL;
    BinNode node ← &begin;
    while pointer1 and pointer2 do
        if pointer1's degree $\le$ pointer2's degree then
            *node ← pointer1;
            pointer1 ← pointer1's sibling;
        else
            *node ← pointer2;
            pointer2 ← pointer2's sibling;
        node ← &((*node)→ sibling)
    if pointer1 == NULL then
        *node ← pointer2;
    else
        *node ← pointer1;
    if !begin then
        return begin;
    // Second step, combine subtrees with same degrees;
    BinNode prev ← NULL;
    BinNode current ← begin;
    BinNode next ← current's sibling;
    while next do
        if current's degree $\neq$ next's degree or (next's sibling and next's
          sibling's degree == current's degree) then
            prev ← current;
            current ← next;
        else
            if current's degree == next's degree then
                if current's value $\le$ next's value then
                    current's sibling ← next's sibling;
                    BinNode temp ← BinomialLink(current, next);
                else
                    if prev == NULL then
                        begin ← next;
                    else
                        prev's sibling ← next;
                    BinomialLink(next, current);
                    current ← next;
        next ← current's sibling;
    return begin;
```

Algorithm 3: BinGetMin2

```
Input: node: BinNode
Output: An array containing the minimum node and its previous node
Function BinGetMin2(node):
    if node == NULL then
        return NULL;
    BinNode* min2 ← (BinNode*)malloc(2 * sizeof(BinNode));
    if min2 == NULL then
        // Handle memory allocation failure return NULL;
    min2[0] ← NULL;
    min2[1] ← node;
    BinNode prev ← NULL;
    BinNode current ← node;
    while current do
        if current→ value≤ min2[1]→ value then
            // Update the min node and its previous node min2[0] ← prev;
            min2[1] ← current;
        prev ← current;
        current ← current→ sibling
        return min2;
```

Algorithm 4: BinDeleteMin

```
Input: node: BinNode
Output: The root node after deleting the minimum node
Function BinDeleteMin(node):
    if node == NULL then
        return NULL;
    // Get the min node and its previous node. BinNode* min2 ←
      BinGetMin2(node);
    BinNode prev ← min2[0];
    BinNode min ← min2[1];
    // Remove the min node from the list. if prev == NULL then
        node ← min→ sibling
    else
        prev-¿sibling ← min→ sibling
    // Reverse the child list of min node. BinNode childlist ← NULL;
    BinNode child ← min→ child while child do
        BinNode temp ← child→ sibling if !childlist then
            // If childlist is empty, create a new list. childlist ← child;
            childlist→ sibling← NULL;
            childlist→ parent← NULL;
        else
            // If childlist is not empty, insert the new node to the head
              of the list. childlist→ parent← NULL;
            child→ sibling← childlist;
            childlist ← child;
        child ← temp;
    free(min2);
    return BinomialMerge(node, childlist);
```

Algorithm 5: BinDeleteMin

```
Input: node: BinNode
Output: The root node after deleting the minimum node
Function BinDeleteMin(node):
    if node == NULL then
        return NULL;
    // Get the min node and its previous node. BinNode* min2 ←
      BinGetMin2(node);
    BinNode prev ← min2[0];
    BinNode min ← min2[1];
    // Remove the min node from the list. if prev == NULL then
        node ← min→ sibling
    else
        prev→ sibling← min→ sibling
    // Reverse the child list of min node. BinNode childlist ← NULL;
    BinNode child ← min→ child while child do
        BinNode temp ← child→ sibling if !childlist then
            // If childlist is empty, create a new list. childlist ← child;
            childlist→ sibling← NULL;
            childlist→ parent← NULL;
        else
            // If childlist is not empty, insert the new node to the head
              of the list. childlist→ parent← NULL;
            child→ sibling← childlist;
            childlist ← child;
        child ← temp;
    free(min2);
    return BinomialMerge(node, childlist);
```

Algorithm 6: BinDecrease

```
Input: binnode: BinNode, value: int, NodeArray: BinNode[]
Function BinDecrease(binnode, value, NodeArray):
    binnode→ value← value;
    BinNode parent, child;
    parent ← binnode→ parent child← binnode;
    while parent ≠ NULL and parent→ value > child → value do
        // Exchange the position between child and parent.
        int temp_value, temp_vertex;
        BinNode temp ← NodeArray[child → vertex]
        NodeArray[child→ vertex]← NodeArray[parent→ vertex]
        NodeArray[parent→ vertex]← temp;
        temp_value ← child→ value
        temp_vertex ← child→ vertex
        child→ value← parent→ value
        child→ vertex← parent→ vertex
        parent→ value← temp_value;
        parent→ vertex← temp_vertex;
        child ← parent;
        parent ← parent→ parent
```

Algorithm 7: IsBinEmpty

```
Input: node: BinNode
Output: 1 if the binomial heap is empty, 0 otherwise
Function IsBinEmpty(node):
    return node == NULL;
```