CSC10004: Data Structure and Algorithms

Lecture 5: Linked lists

Lecturer: Bùi Văn Thạch
TA: Ngô Đình Hy/Lê Thị Thu Hiền
{bvthach,ndhy}@fit.hcmus.edu.vn, lthien@hcmus.edu.vn

Goals

1. To help students understand why linked lists are used rather than arrays.
2. Build knowledge on linked lists which form the basis for more complex structures like stacks, queues, hash tables, and graphs.
3. To be a master of pointers.

Course topics

0. Introduction
1. Algorithm complexity analysis
2. Recurrences
3. Search
4. Sorting
5. Linked list
6. Stack, and Queue, Priority queue
7. Tree
    1. Binary search tree (BST)
    2. AVL tree
8. Graph
    1. Graph representation
    2. Graph search
9. Hashing
10. Algorithm designs
    1. Greedy algorithm
    2. Divide-and-Conquer
    3. Dynamic programming

Summary (1/2)

		Design	Run time				Space	
			Run time	Search	Insert	Delete	In-place	Stable
Unsorted	Array	Det.	$O(1)$	$O(n)$	$O(1)$	$O(n)$		
Comparison-based	Selection sort	Det.	$O(n^2)$	$O(\log n)$	$O(1)$ if position known with list		Yes	No
	Insertion sort	Det.	$O(n^2)$				Yes	Yes
	Heap sort	Det.	$O(n \log n)$				Yes	No
	Merge sort	Det.	$\Theta(n \log n)$				No	Yes
	Quick sort	Rnd.	$O(n \log n)$				Yes	No
Non-comparison-based	Radix sort	Det.	$O\big(d(n+k)\big)$	$O(\log n)$	$O(n)$ for array		No	Yes

d is the number of digits and k is the range of digits.

Summary (2/2)

Criteria	Sorting with Array	Sorting with Linked List
Access Time	$O(1)$ (Direct access via index)	$O(n)$ (Sequential access, must traverse the list)
Sorting Approach	Uses swapping of elements (operates on indices)	Uses pointer manipulation (modifies links)
Memory Usage	Requires contiguous memory allocation	Uses dynamic memory allocation, no need for contiguous space
Efficiency	Fast for small datasets, but resizing may be expensive $O(n)$	Efficient for large datasets with frequent insertions/deletions $O(1)$
Best Sorting Algorithm	QuickSort, MergeSort (due to random access)	MergeSort, QuickSort (preferred for linked lists due to ease of pointer updates)
Extra Space Required	No extra space for in-place sorting	Extra space may be needed for recursive sorting (MergeSort)
Stability	Sorting algorithms like MergeSort and SelectionSort are stable	MergeSort is stable, but some implementations of QuickSort may not be
Ease of Implementation	Easier due to direct indexing	Harder due to pointer manipulation
Use Case	Best for static data where size is known	Best for dynamic data that frequently changes

- Linked List Overview
- Types of Linked Lists
- Operations on Singly Linked Lists
- Sorting using Singly Linked Lists

- Linked List Overview
- Types of Linked Lists
- Operations on Singly Linked Lists
- Sorting using Singly Linked Lists

Implicit Relationships

- Each element has an index, and it is implicitly understood that x_{i+1} follows x_i. Therefore, the elements must be located next to each other in memory.
- Fast random access to elements.
- Elements must be contiguous in memory.
- Fixed number of elements; insertion and deletion require shifting elements.
  - No direct insertion or deletion (only element movement).
- Memory overhead due to unknown future size (pre-allocation).
- Example: One-dimensional array.

0	1	...	i	$i+1$	...	6	$n-1$
x_0	x_1	...	x_i	x_{i+1}	...	19	x_{n-1}

Explicit Relationships

- Each element stores its own data and a pointer to the next element.
- Elements do not need to be contiguous in memory.
- Accessing an element requires traversing from another element.
- Different types of linked lists: singly, doubly, circular.

Linked list

Advantages

- No fixed size: Unlike arrays, linked lists can grow or shrink dynamically as needed. This avoids wasting memory on unused space.
- Efficient insertion and deletion: Inserting or deleting elements in a linked list is generally faster than in an array, as it doesn't require shifting existing elements.
- Flexibility in memory allocation: Linked lists can be scattered throughout memory, unlike arrays which need contiguous blocks of memory.

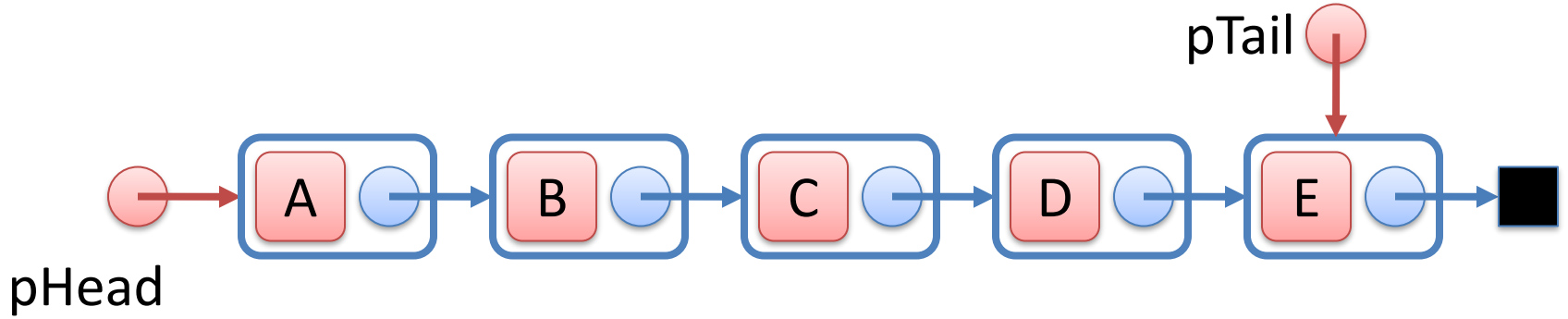
Disadvantages:

- Memory overhead: Each node requires additional space for the link pointer(s).
- Slower random access: Accessing a specific element in a linked list requires traversing the list, which can be slower than accessing an element in an array by index.
- More complex implementation: Linked lists require more complex pointer manipulation compared to arrays.

- Linked List Overview
- Types of Linked Lists
- Operations on Singly Linked Lists
- Sorting using Singly Linked Lists

- Singly Linked List
- Doubly Linked List
- Circular Singly Linked List
- Circular Doubly Linked List

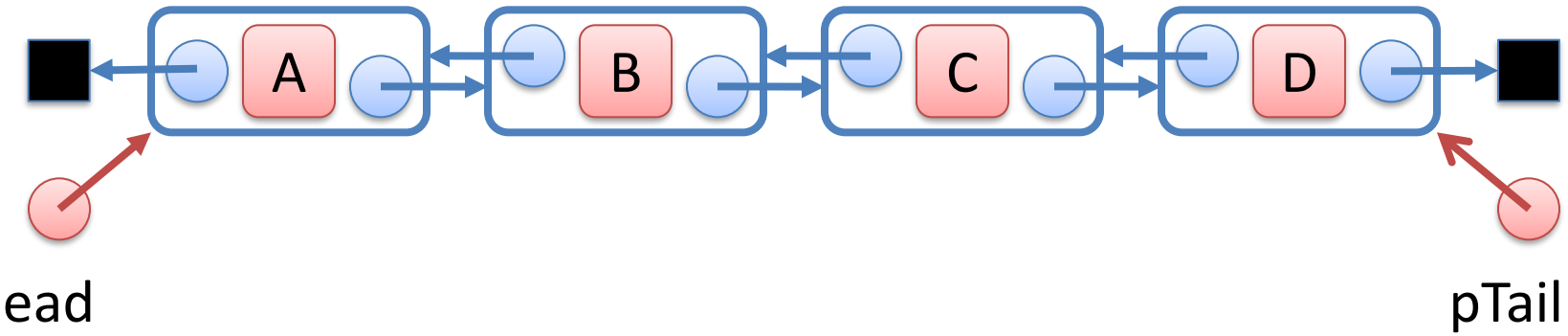
Singly Linked List




```
typedef struct tagNode
{
    Data Info; // predefined data
    struct tagNode *pNext; // pointer to the NEXT tagNode
} NODE;
```

```
typedef struct tagList
{
    NODE *pHead; // FIRST element of the list
    NODE *pTail; // LAST element of the list
} LIST;
```

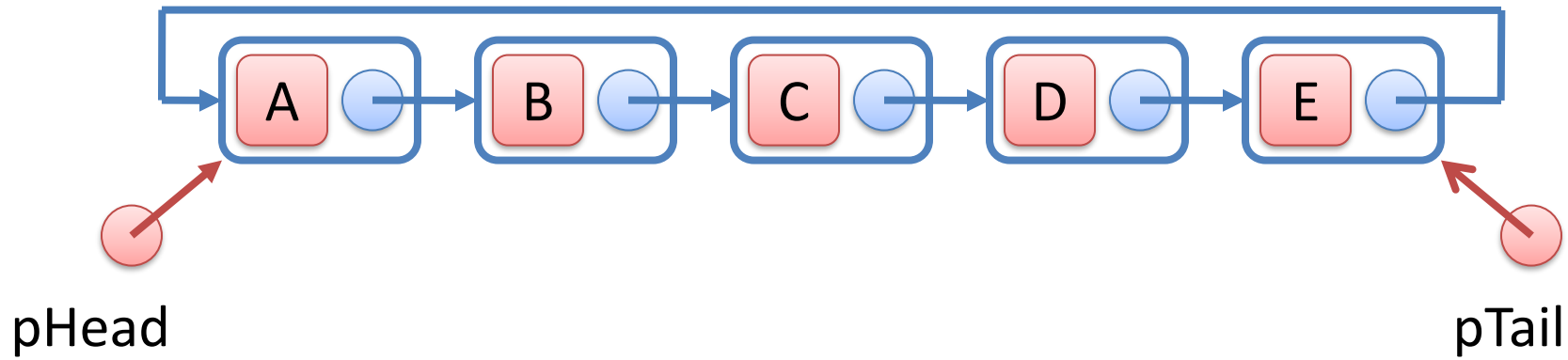
Doubly Linked List




```
typedef struct tagDNode
{
    Data Info;  // predefined data
    struct tagDNode *pNext, *pPrev; // pointer to the PREVIOUS and NEXT tagNodes
} DNODE;
```

```
typedef struct tagDList
{
    NODE *pHead; // FIRST element of the list
    NODE *pTail; // LAST element of the list
} DLIST;
```

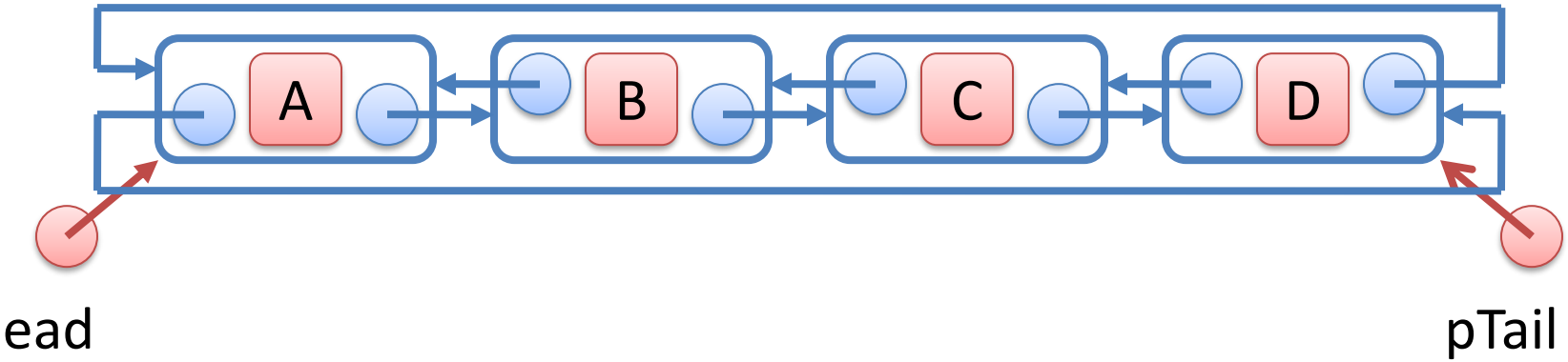
Circular Singly Linked List




```
typedef struct tagCNode
{
    Data Info; // predefined data
    struct tagCNode *pNext; // pointer to the NEXT tagNodes
} CNODE;
```

```
typedef struct tagCList
{
    NODE *pHead; // FIRST element of the list
    NODE *pTail; // LAST element of the list
} CLIST;
pTail->pNext = pHead;
```

Circular Doubly Linked List

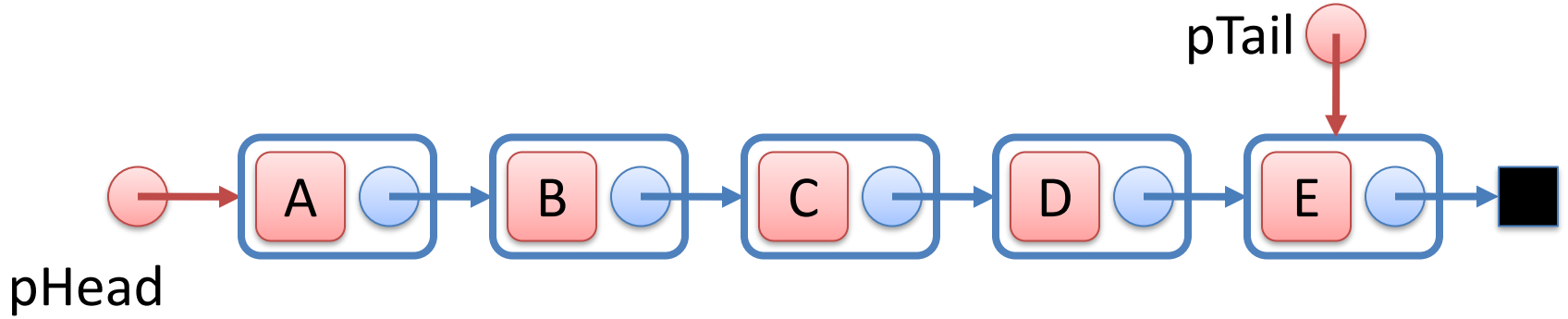



```
typedef struct tagCNode
{
    Data Info;
    struct tagCNode *pNext, *pPrev;
} CNODE;
```

```
typedef struct tagCList
{
    NODE *pHead;
    NODE *pTail;
} CLIST;
pTail->pNext = pHead;
pHead->pPrev = pTail;
```

- Linked List Overview
- Types of Linked Lists
- Operations on Singly Linked Lists
  - Create an empty list
  - Add an element to the list
  - Search for a value in the list
  - Extract an element from the list
  - Traverse the list
  - Delete the entire list
- Sorting using Singly Linked Lists

Structure




```
typedef struct tagNode
{
    Data Info;
    struct tagNode *pNext;
} NODE;
```

```
typedef struct tagList
{
    NODE *pHead;
    NODE *pTail;
} LIST;
```

Example

- Defining an element in a Singly Linked List to store student records

```
typedef struct Student {
    char Name[30];
    int StudentID;
} STUDENT;
```

```
typedef struct StudentNode {
    STUDENT Info;
    struct StudentNode *pNext;
} StudentNODE;
```

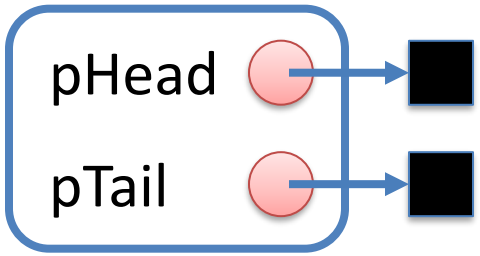
```
typedef struct studentList
{
    StudentNODE *pHead;
    StudentNODE *pTail;
} LIST;
```

- Linked List Overview
- Types of Linked Lists
- Operations on Singly Linked Lists
  - Create an empty list
  - Add an element to the list
  - Search for a value in the list
  - Extract an element from the list
  - Traverse the list
  - Delete the entire list
- Applications of Singly Linked Lists

Create an empty list

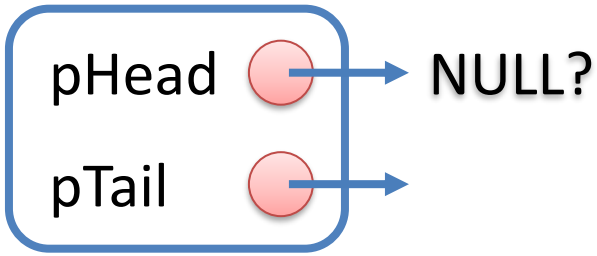
- Initialize a list

```
void Init(LIST &l)
{
   l.pHead = l.pTail = NULL;
}
```



- Check if the list is empty

```
bool isEmpty(LIST &l)
{
   return l.pHead;
}
```



- Linked List Overview
- Types of Linked Lists
- Operations on Singly Linked Lists
  - Create an empty list
  - Add an element to the list
  - Search for a value in the list
  - Extract an element from the list
  - Traverse the list
  - Delete the entire list
- Sorting using Singly Linked Lists

Create an element

- Initialize an element

```
NODE* GetNode(Data x)  {
    NODE *p;
    // Allocate memory for the node
    p = new NODE;
    if (p == NULL) {
        cout << "Not enough memory“ << endl;
        return NULL;
    }
    p->Info = x; // Assign data to node p
    p->pNext = NULL;
    return p;
}
```

- To create a new node for the list, execute the following command:

```
new_e = GetNode(x);
```

- → element will manage the address of the newly created node.

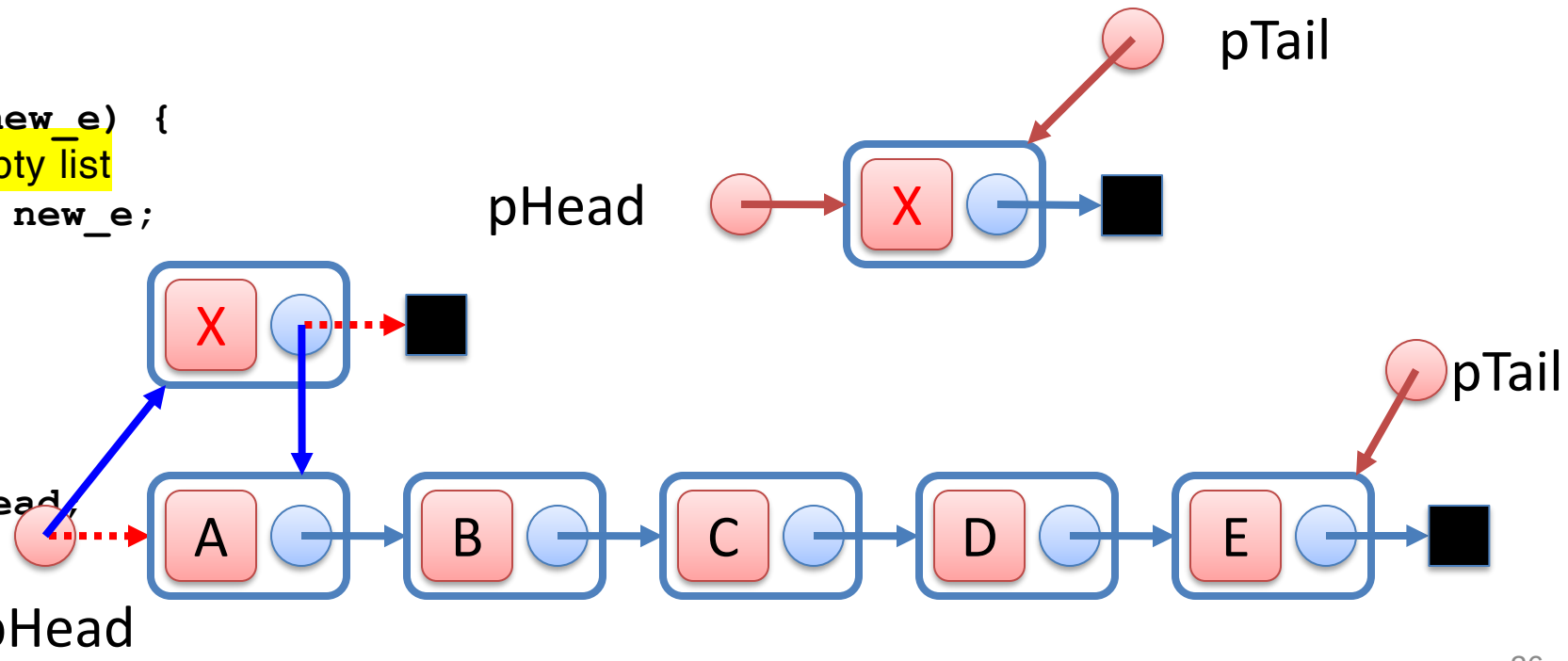
Three positions to add a new element to the list:

- Add at the beginning of the list.
- Append to the end of the list.
- Insert into the list AFTER a node q.
- Insert into the list BEFORE a node q.

Add at the beginning of the list

```
NODE* InsertHead(LIST &l, Data x){
        NODE* new_e = GetNode(x);
        if (new_e == NULL)
                return NULL;
        AddFirst(l, new_e);
        return new_e;
}

void AddFirst(LIST &l, NODE* new_e) {
     if (l.pHead == NULL) // Empty list
             l.pHead = l.pTail = new_e;
     else {
             new_e->pNext = l.pHead;
             l.pHead = new_e;
     }
}
```

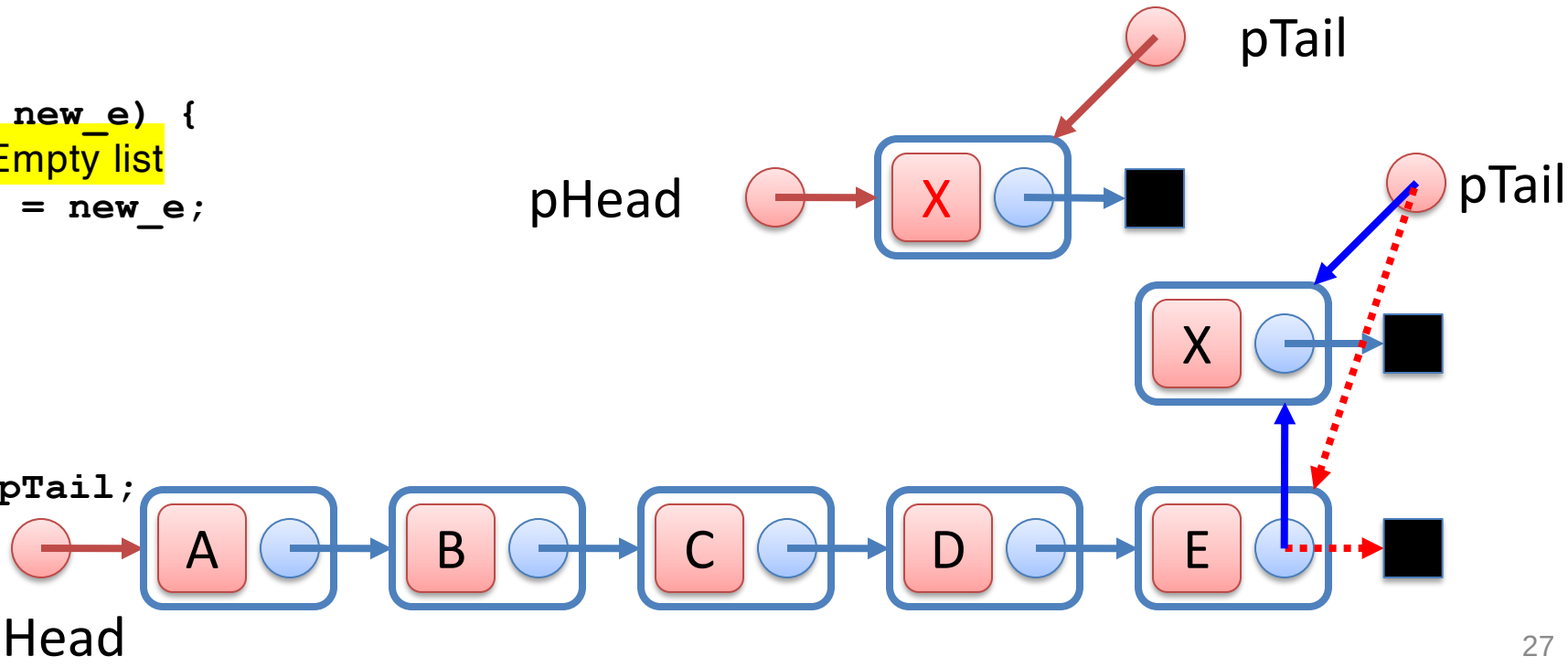
Create a new node new_e to store data x
If the node is not successfully created:
Report an error
else: Add new_e to the beginning of the list



Append to the end of the list

```
NODE* InsertTail(LIST &l, Data x){
        NODE* new_e = GetNode(x);
        if (new_e == NULL)
                return NULL;
        AddTail(l, new_e);
        return new_e;
}

void AddTail(LIST &l, NODE* new_e) {
     if (l.pHead == NULL) // Empty list
             l.pHead = l.pTail = new_e;
     else {
             new_e->pNext = l.pTail;
             l.pTail = new_e;
     }
}
```

Create a new node new_e to store data x
If the node is not successfully created:
Report an error
else: Add new_e to the end of the list

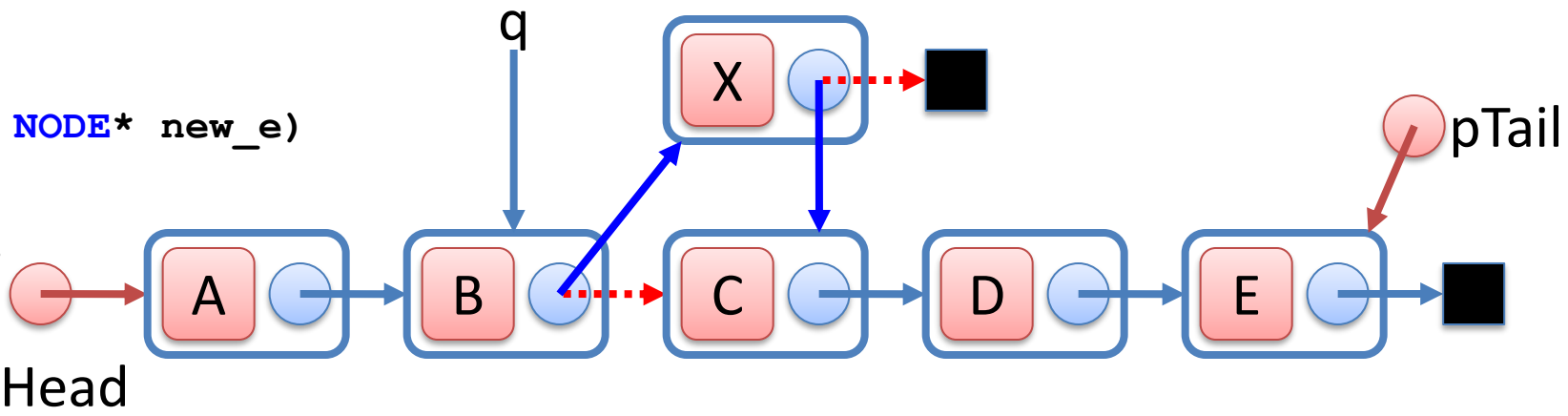


Insert into the list AFTER a node q

```
NODE* InsertAfter(LIST &l, Data x){
      NODE* new_e = GetNode(x);
      if (new_e == NULL)
            return NULL;
      AddAfter(l, q, new_e);
      return new_e;
}
```

Create a new node new_e to store data x
If the node is not successfully created:
Report an error
else: Add new_e to the end of the list

```
void AddAfter(LIST &l,NODE *q, NODE* new_e)
{
     if (q!=NULL) {
       new_e->pNext = q->pNext;
       q->pNext = new_e;
       if(q == l.pTail)
         l.pTail = new_e;
     }
     else // Add at the beginning of the list
       AddFirst(l, new_e);
}
```

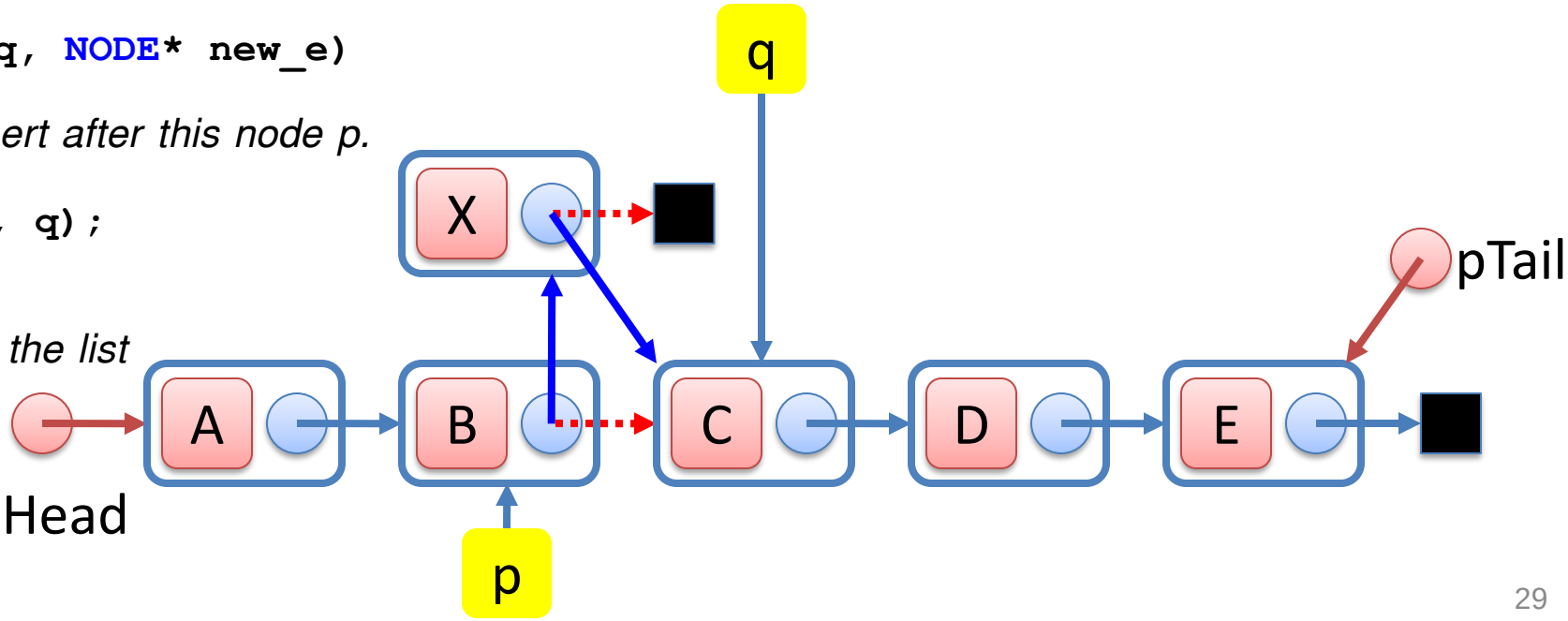


Insert into the list BEFORE a node q

```
NODE* InsertBefore(LIST &l, Data x){
      NODE* new_e = GetNode(x);
      if (new_e == NULL)
        return NULL;
      AddBefore(l, q, new_e);
      return new_e;
}
```

Create a new node new_e to store data x
If the node is not successfully created:
Report an error
else: Add new_e to the end of the list

```
void AddBefore(LIST &l,NODE *q, NODE* new_e)
{
     // Find node p before q, then insert after this node p.
     if (q != NULL) {
       NODE* p = findBefore(l, q);
       AddAfter(l, p, new_e);
     }
     else // Add at the beginning of the list
       AddFirst(l, new_e);
}
```

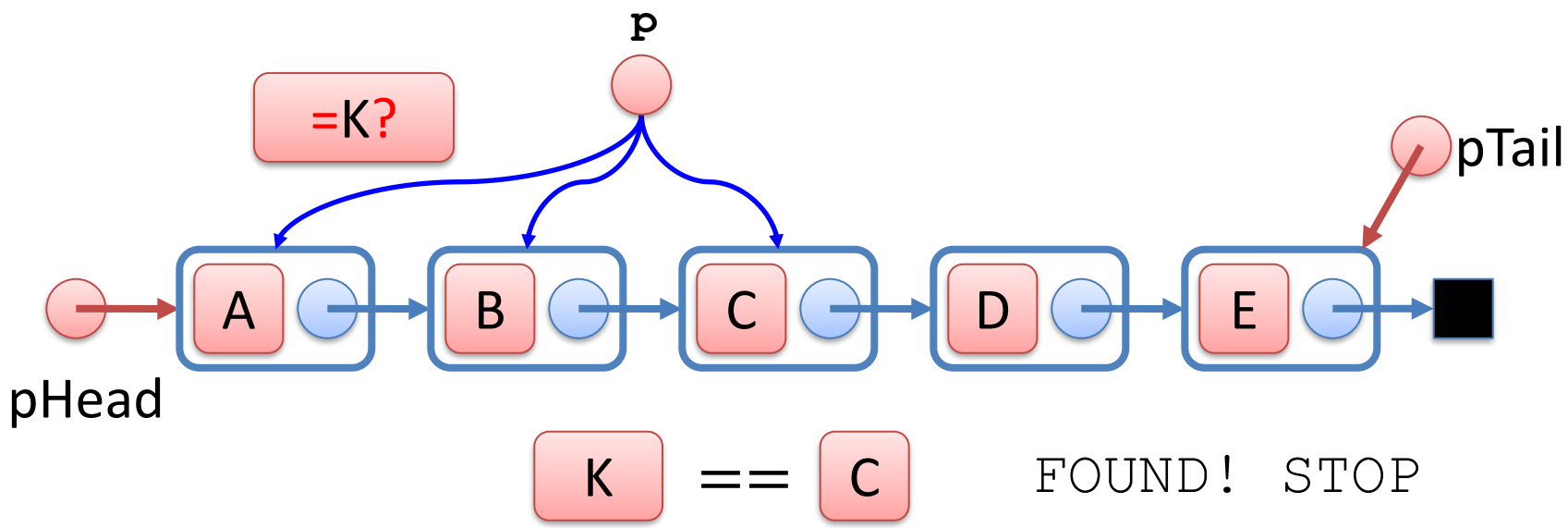


- Linked List Overview
- Types of Linked Lists
- Operations on Singly Linked Lists
  - Create an empty list
  - Add an element to the list
  - Search for a value in the list
  - Extract an element from the list
  - Traverse the list
  - Delete the entire list
- Sorting using Singly Linked Lists

```
NODE* Search(LIST l, Data K) {
        NODE *p;
    1.  p = l.pHead;
    2.  while((p!= NULL)&&(p->Info!=K))
    3.          p = p->pNext;
    4.  return p;
}
```

// Input: List (head, tail), data value K
// Output: Address of the node containing K

1. p = Head; // Set p to the first node in the list
2. While (p != NULL) and (p->Info != Info), do:
3. Move p to the next node: p = p->Next;
4. If p != NULL, then p points to the desired node. Otherwise, the element is not found in the list.



- Linked List Overview
- Types of Linked Lists
- Operations on Singly Linked Lists
  - Create an empty list
  - Add an element to the list
  - Search for a value in the list
  - Extract an element from the list
  - Traverse the list
  - Delete the entire list
- Sorting using Singly Linked Lists

Overview

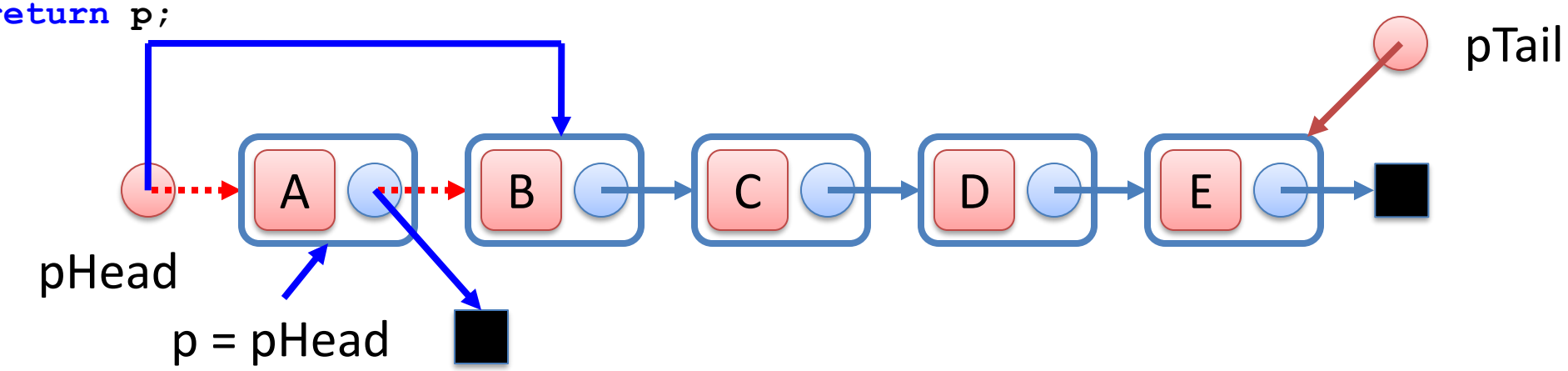
Three cases of extracting a node from the list:

- Extract the head node.
- Extract the node after a given node q.
- Extract the node with value K.

- **Note:** If you want to delete the extracted node, you MUST call the delete operator to free memory.

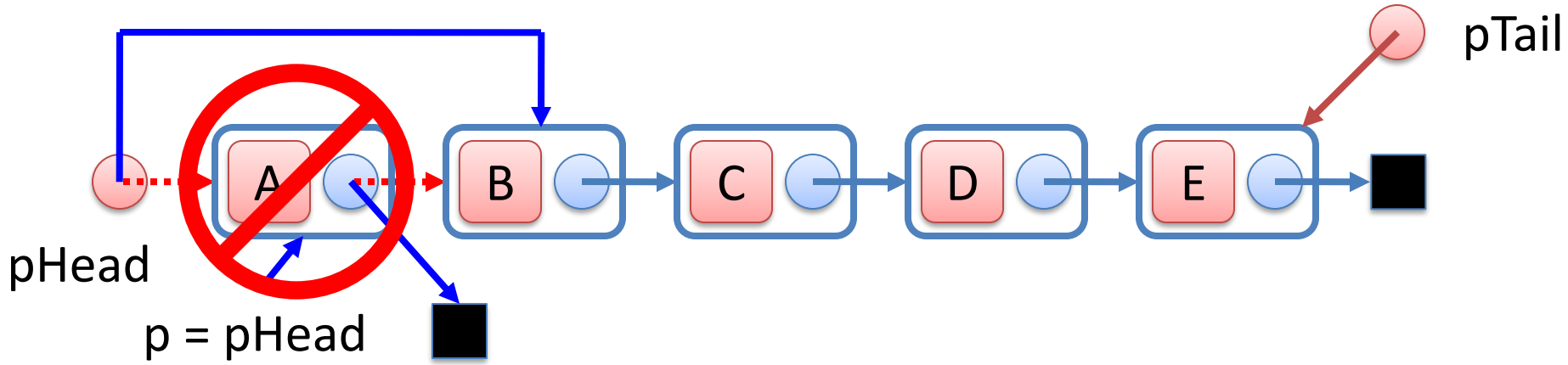
Extract the head node

```
// Input: List (head, tail)
// Output: List after removing the first node
NODE* PickHead(LIST &l) {
      NODE   *p = NULL;
      if (l.pHead != NULL) {
            p = l.pHead; // p is the node to be extracted
            l.pHead = l.pHead->pNext; // Detach p from the list
            p->pNext = NULL;
            if(l.pHead == NULL)
                  l.pTail =  NULL; // The list is now empty
      }
      return p;
}
```



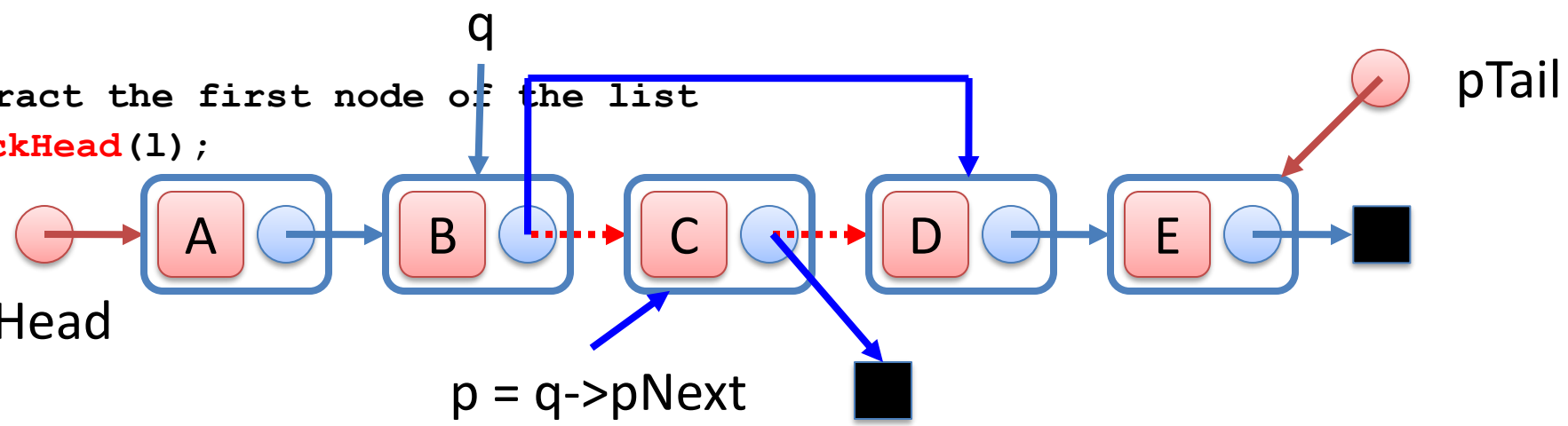
Extract and delete the head node

```
Data RemoveHead(LIST &l)
{
        if (l.pHead == NULL)
                return NULLDATA;
        NODE* p = PickHead(l);
        Data x = p->Info;
        delete p;
        return x;
}
```



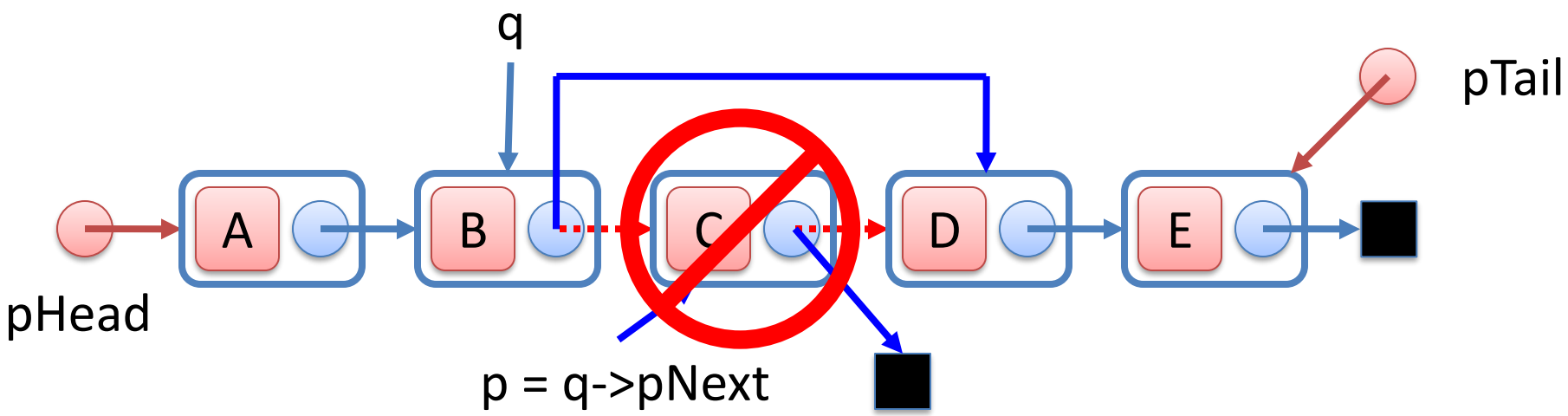
Extract the node after a given node q

```
// Input: List (head, tail), pointer q  // Output: List after removing q
NODE* PickAfter(LIST &l, NODE *q) {
   NODE *p;
   if (q != NULL) {
        p = q ->pNext; // p is the node to be removed
        if (p != NULL) { // Ensure q is not the last node
          if (p == l.pTail)
            l.pTail = q; // Update tail if p was the last node
          q->pNext = p->pNext; // Detach p from the list
          p->pNext = NULL;
        }
   }
   else // Extract the first node of the list
        p = PickHead(l);
   return p;
}
```



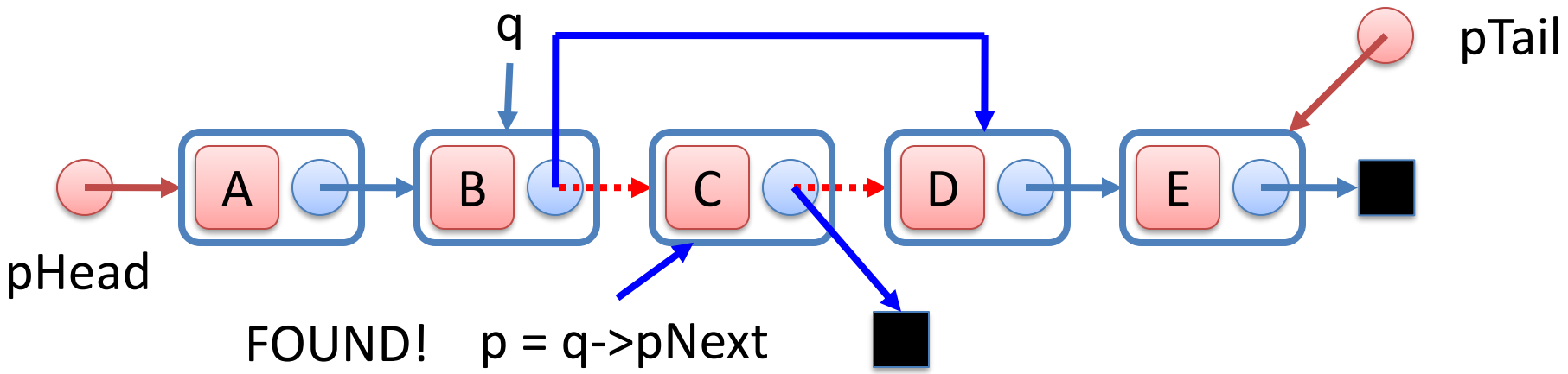
Extract and Delete the node after a given node q

```
// Input: List (head, tail), pointer q
// Output: List after removing q
Data RemoveAfter(LIST &l, NODE *q) {
        NODE *p = PickAfter(l, q);
        if (p == NULL)
                return NULLDATA;
        Data x = p->Info;
        delete p;
        return x;
}
```



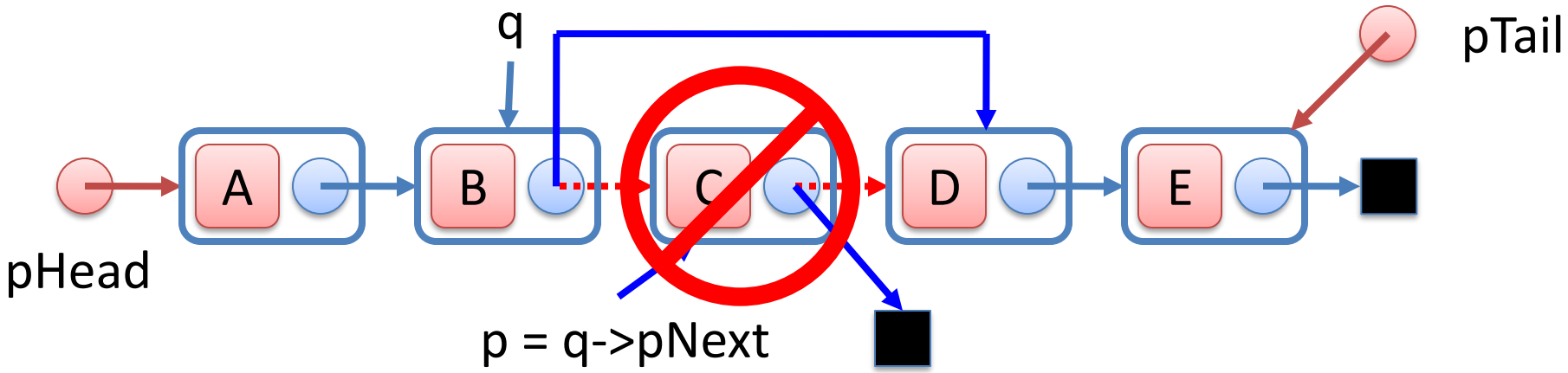
Extract the node with value K

```
// Input: List (head, tail), key K   // Output: List after removing node with key K
NODE* PickNode(LIST &l, Data K) {
      NODE *p = l.pHead, *q = NULL;
      // Find node p with key K and the node q before it.
      while ((p != NULL) && (p->Info != K)) {
         q = p;
         p = p->pNext;
      }
      if (p == NULL) // Report that key K is not found.
         return NULL;
      return PickAfter(l, q); // Key K is found
}
```



Extract and Delete the node with value K

```
// Input: List (head, tail), key K
// Output: List after removing node with key K
NODE* PickNode(LIST &l, Data K) {
        NODE *p = PickNode(l, K);
        if (p == NULL)
                return NULLDATA;
        Data x = p->Info;
        delete p;
        return x;
}
```



- Linked List Overview
- Types of Linked Lists
- Operations on Singly Linked Lists
  - Create an empty list
  - Add an element to the list
  - Search for a value in the list
  - Extract an element from the list
  - Traverse the list
  - Delete the entire list
- Applications of Singly Linked Lists

Overview

Traversing a linked list is a common operation used when there is a need to process all elements in the list in the same manner or to retrieve aggregated information from the elements, such as:

- Counting the number of elements in the list.
- Finding all elements that satisfy a given condition.
- Deleting the entire list (and freeing memory).

Source code

- Step 1: p = pHead; // Initialize p to point to the head of the list.
- Step 2: While the list is not fully traversed:
  - Process the current node p.
  - p=p->pNext; // Move p to the next node.

```
void ProcessList(LIST &l)
{
   NODE *p;
   p = l.pHead;
   while (p!= NULL)
   {
      ProcessNode(p);  // Specific processing depends on the application
      p = p->pNext;
   }
}
```

- Linked List Overview
- Types of Linked Lists
- Operations on Singly Linked Lists
  - Create an empty list
  - Add an element to the list
  - Search for a value in the list
  - Extract an element from the list
  - Traverse the list
  - Delete the entire list
- Sorting using linked lists

Deleting the Entire Linked List

```
void RemoveList(LIST &l)
{
    NODE *p;
    while (l.pHead!= NULL) {
         p = l.pHead;
         l.pHead = p->pNext;
       delete p;
    }
    l.pTail = NULL;
}
```

To delete the entire linked list, the process involves deallocating each node's memory while updating the relevant links:

- Step 1: While the list is not empty:
  - Store the head node in p.
  - Move the head pointer to the next node.
  - Delete p to free memory.
- Step 2: Set pTail = NULL to ensure consistency when the list is empty.

- Linked List Overview
- Types of Linked Lists
- Operations on Singly Linked Lists
- Sorting using Singly Linked Lists

Approach

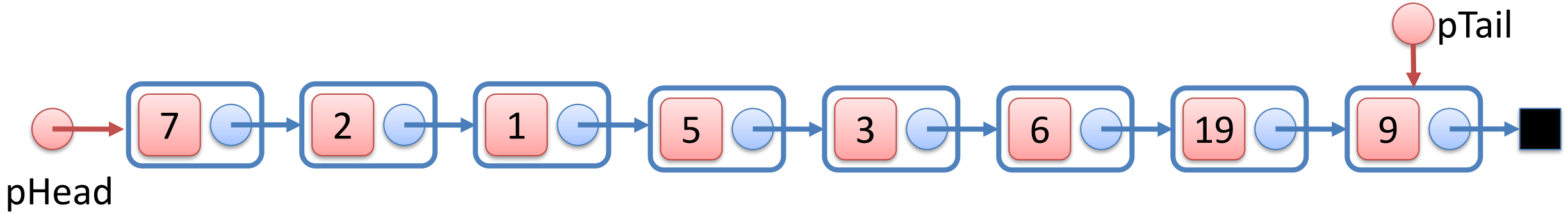
Criteria	Option 1: Swap Contents (Info Field)	Option 2: Change Links (pNext Field)
Sorting Approach	Exchanges values inside nodes	Modifies node pointers to rearrange nodes
Memory Usage	No extra memory required	May require extra pointers for re-linking
Data Movement	Moves only data values	Moves entire node references
Efficiency	Faster since only data is swapped	Slightly slower due to pointer operations
Complexity	Similar to array sorting, works well with BubbleSort and QuickSort	Works better with MergeSort due to easy pointer updates
Stability	Can be stable if implemented properly	Naturally stable since nodes are not reordered incorrectly
Implementation Complexity	Simpler as it follows array-like swapping logic	More complex due to pointer manipulation
Use Case	Suitable for lists with simple data types (integers, small strings)	Suitable for lists with large objects or structures to avoid costly data movement

We only focus on Option 2 here

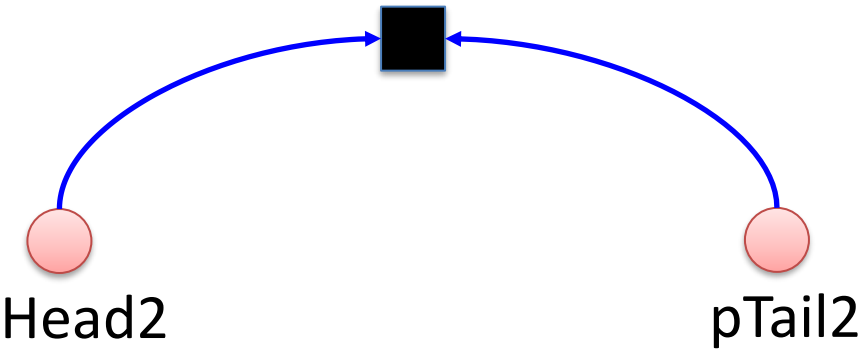
List-based selection sort

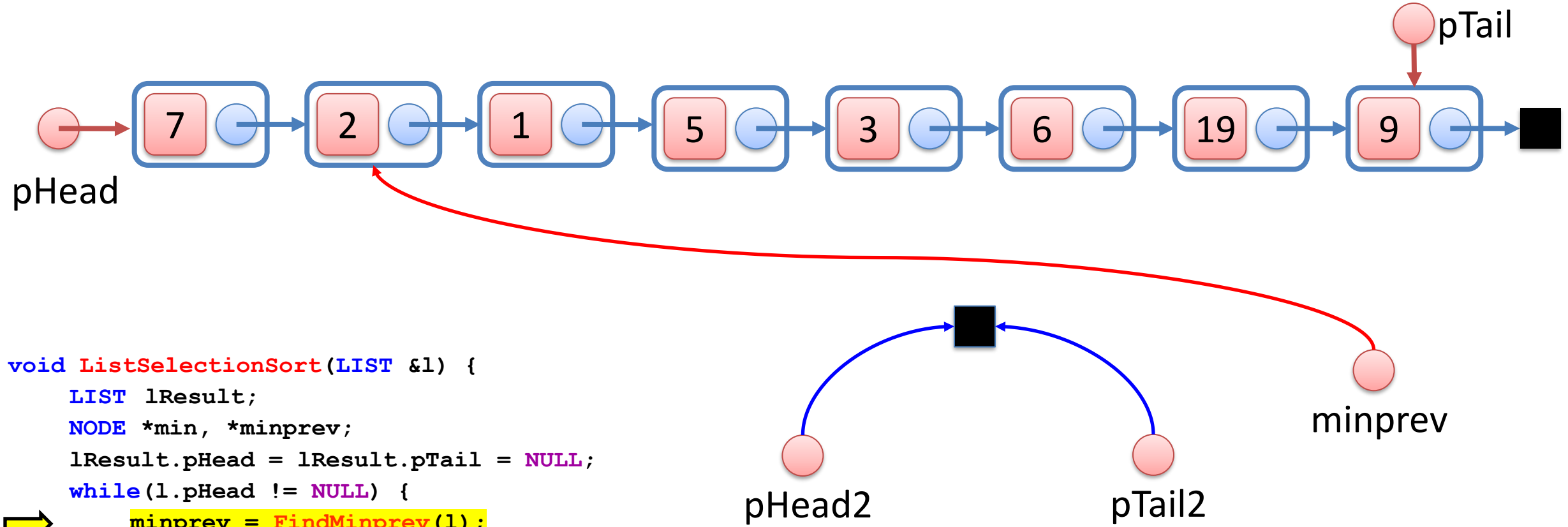
```
void ListSelectionSort(LIST &l) {
     LIST lResult;
     NODE *min, *minprev;
     lResult.pHead = lResult.pTail = NULL;
     while(l.pHead != NULL) {
          minprev = FindMinprev(l);
          min = PickAfter(l, minprev);
          AddTail(lResult, min);
     }
     l = lResult;
}
```

```
NODE* FindMinprev(LIST l) {
     NODE *min, *minprev, *p, *q;
     minprev = q = NULL;
     min= p = l.pHead;
     while(p != NULL)       {
          if (p->Info < min->Info)   {
               min = p;
               minprev = q;
          }
          q = p;
          p = p->pNext;
     }
     return minprev;
}
```

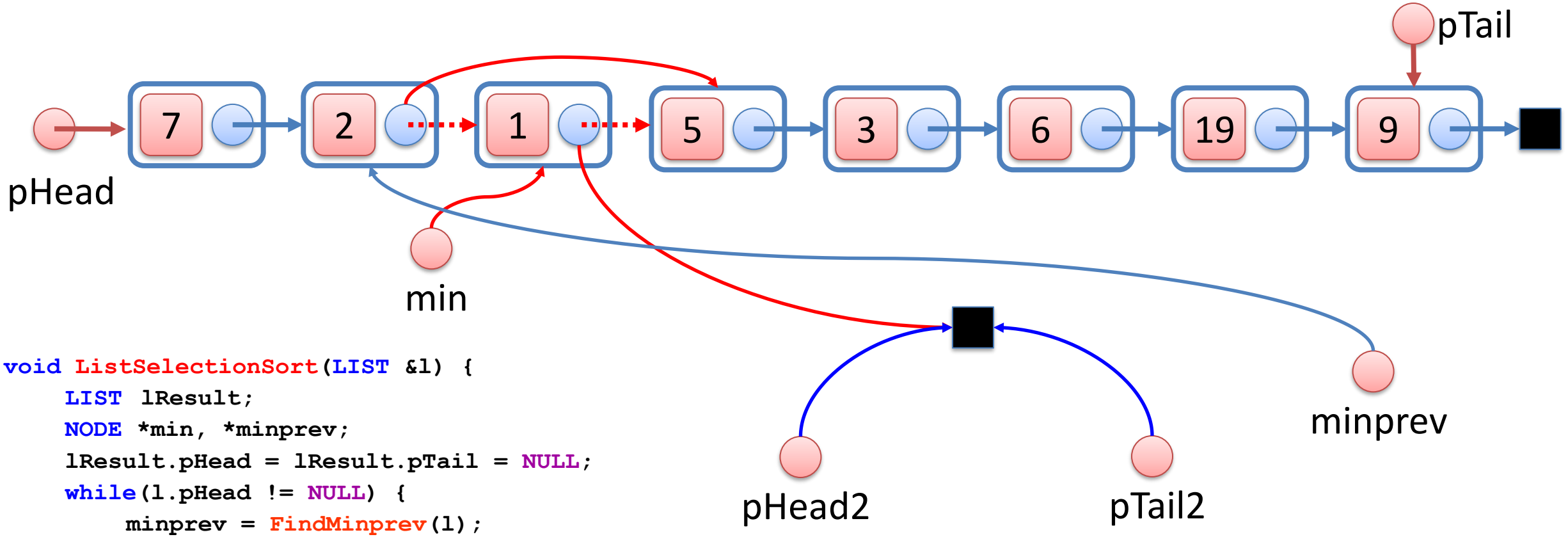



```
void ListSelectionSort(LIST &l) {
     LIST lResult;
     NODE *min, *minprev;
     lResult.pHead = lResult.pTail = NULL;
     while(l.pHead != NULL) {
          minprev = FindMinprev(l);
          min = PickAfter(l, minprev);
          AddTail(lResult, min);
     }
     l = lResult;
}
```

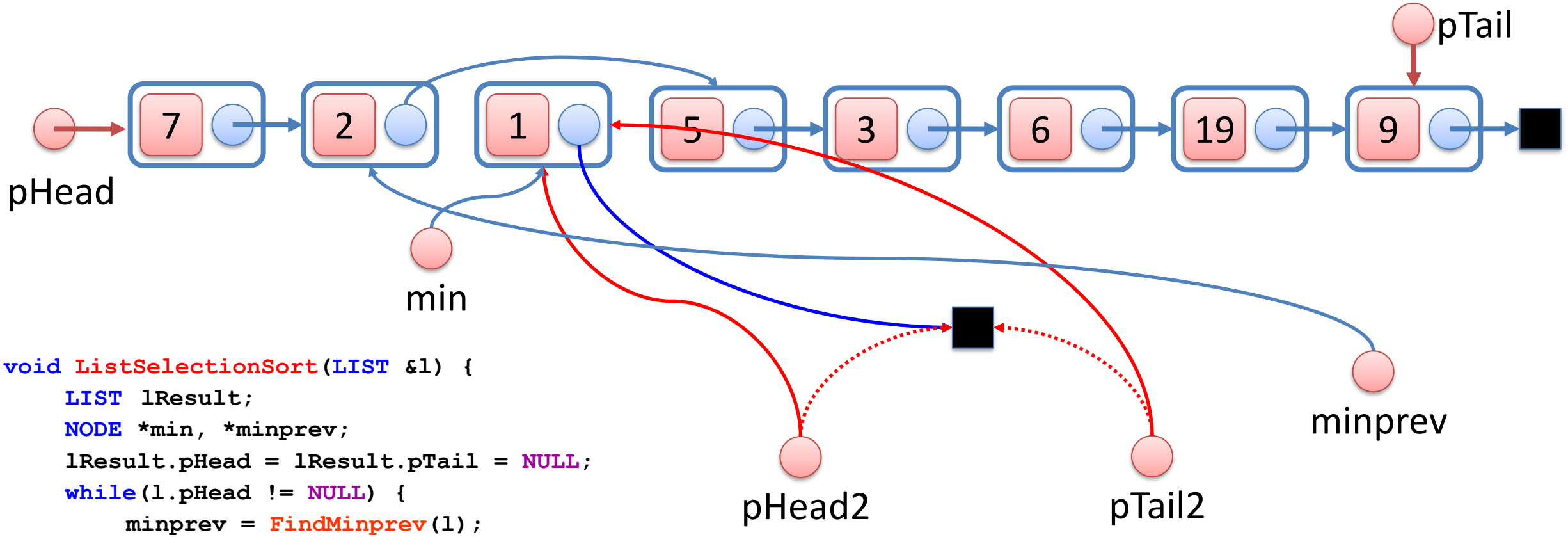




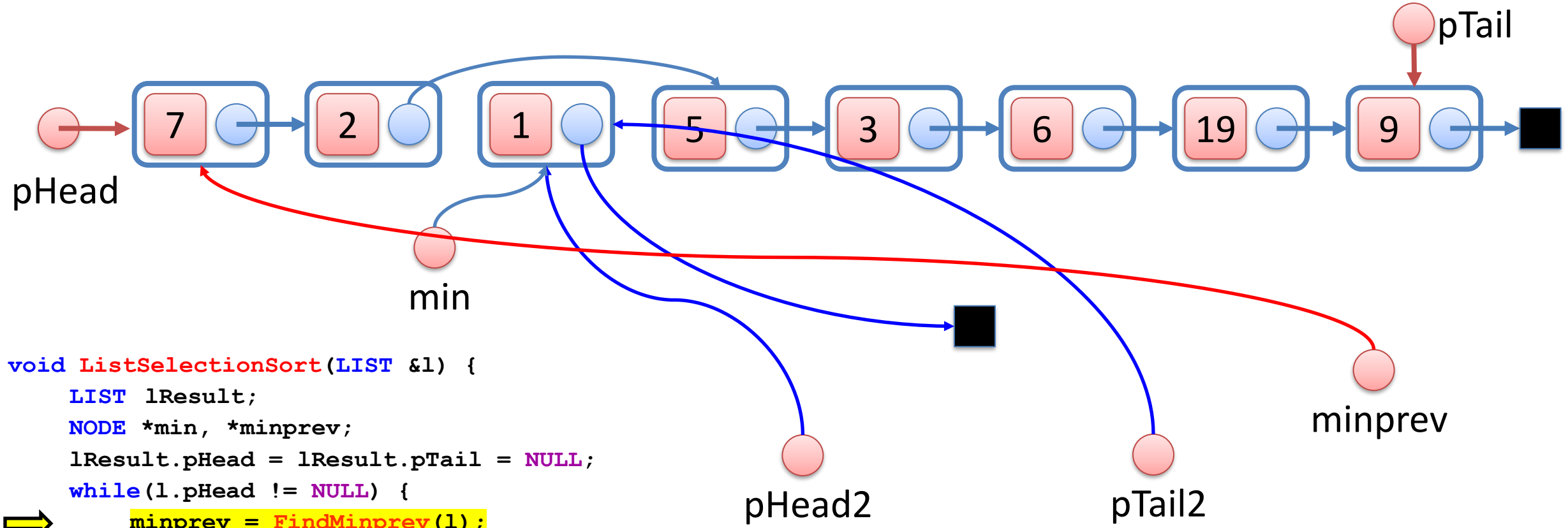

```
void ListSelectionSort(LIST &l) {
     LIST lResult;
     NODE *min, *minprev;
     lResult.pHead = lResult.pTail = NULL;
     while(l.pHead != NULL) {
          minprev = FindMinprev(l);
          min = PickAfter(l, minprev);
          AddTail(lResult, min);
     }
     l = lResult;
}
```



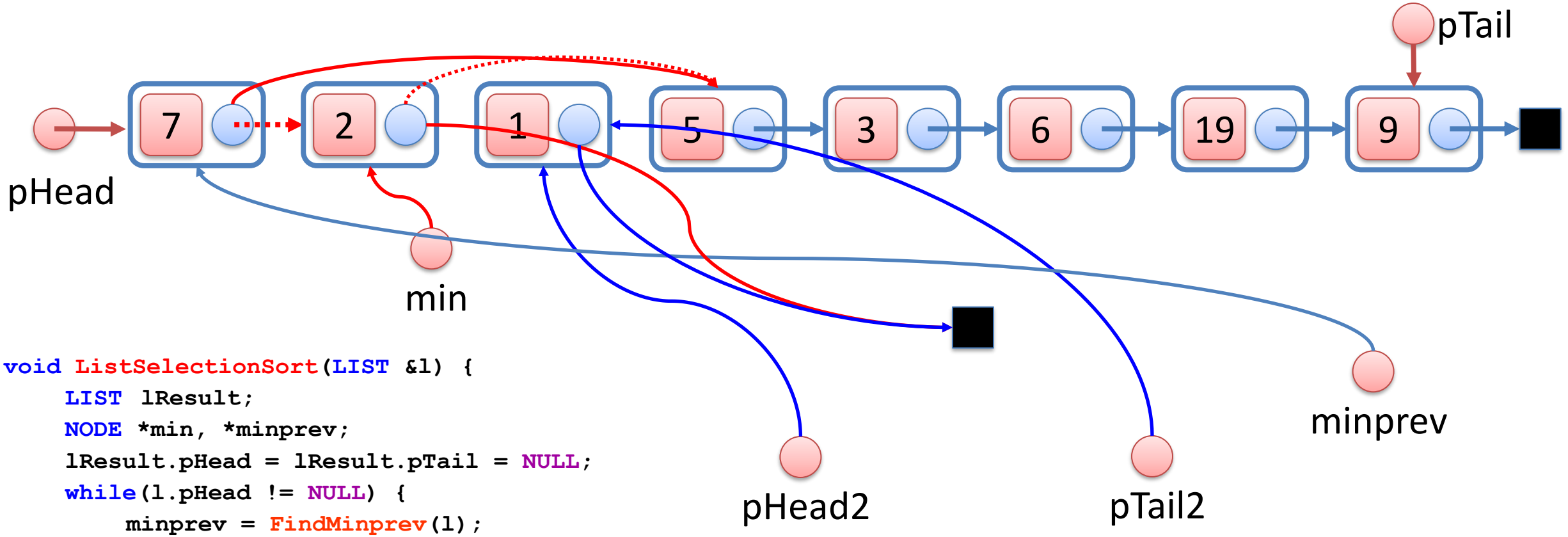

```
void ListSelectionSort(LIST &l) {
     LIST lResult;
     NODE *min, *minprev;
     lResult.pHead = lResult.pTail = NULL;
     while(l.pHead != NULL) {
          minprev = FindMinprev(l);
          min = PickAfter(l, minprev);
          AddTail(lResult, min);
     }
     l = lResult;
}
```



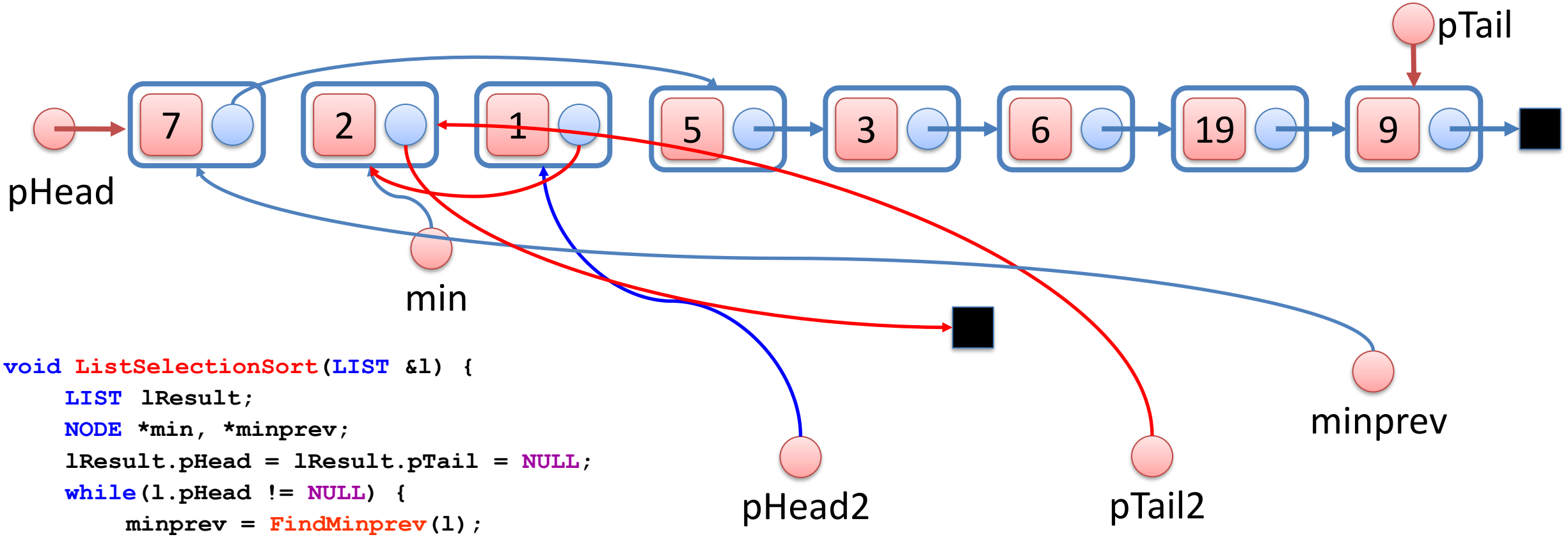

```
void ListSelectionSort(LIST &l) {
     LIST lResult;
     NODE *min, *minprev;
     lResult.pHead = lResult.pTail = NULL;
     while(l.pHead != NULL) {
          minprev = FindMinprev(l);
          min = PickAfter(l, minprev);
          AddTail(lResult, min);
     }
     l = lResult;
}
```



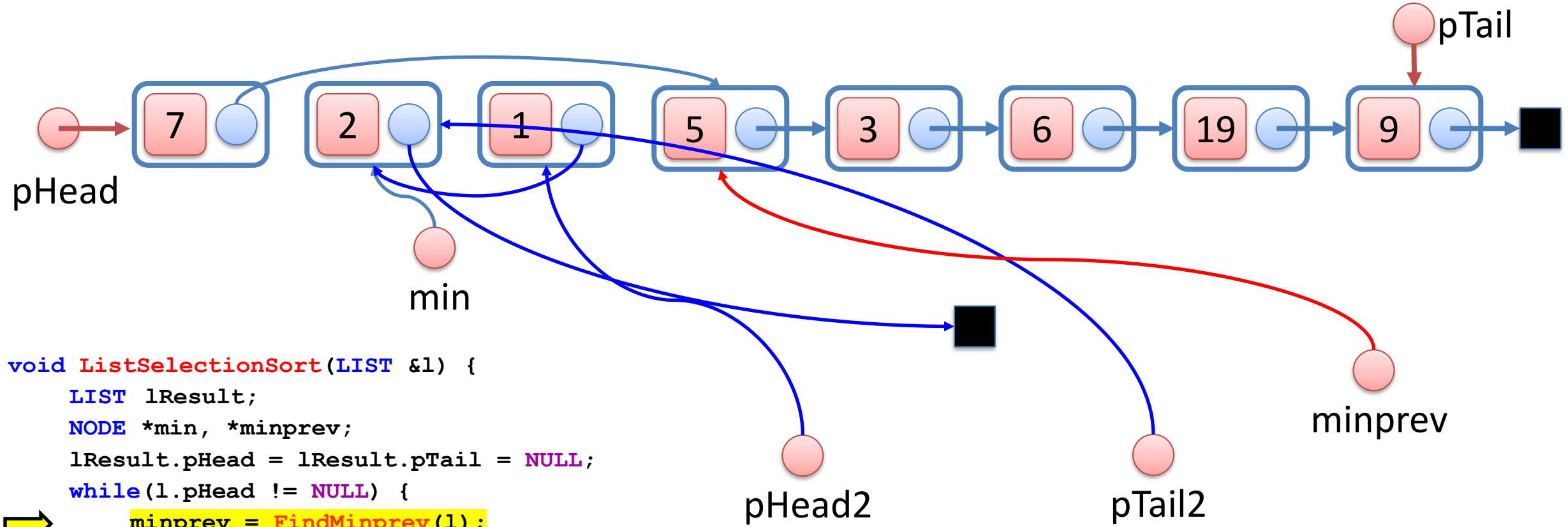

```
void ListSelectionSort(LIST &l) {
     LIST lResult;
     NODE *min, *minprev;
     lResult.pHead = lResult.pTail = NULL;
     while(l.pHead != NULL) {
          minprev = FindMinprev(l);
          min = PickAfter(l, minprev);
          AddTail(lResult, min);
     }
     l = lResult;
}
```



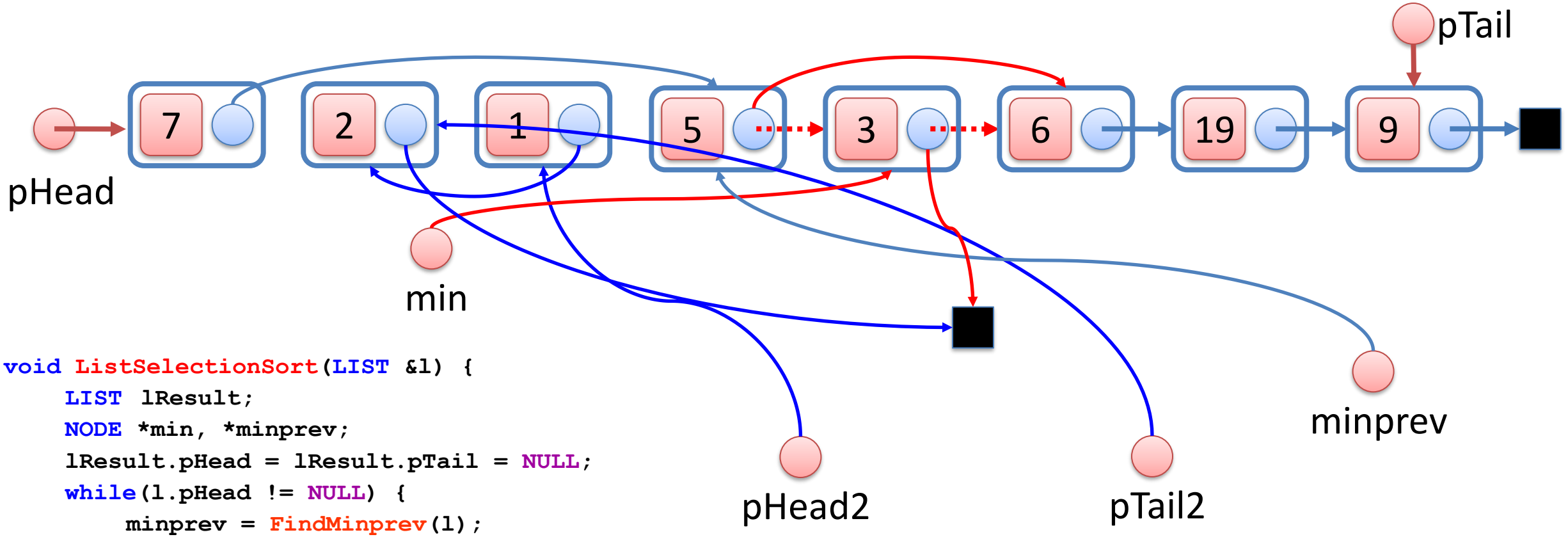

```
void ListSelectionSort(LIST &l) {
     LIST lResult;
     NODE *min, *minprev;
     lResult.pHead = lResult.pTail = NULL;
     while(l.pHead != NULL) {
          minprev = FindMinprev(l);
          min = PickAfter(l, minprev);
          AddTail(lResult, min);
     }
     l = lResult;
}
```



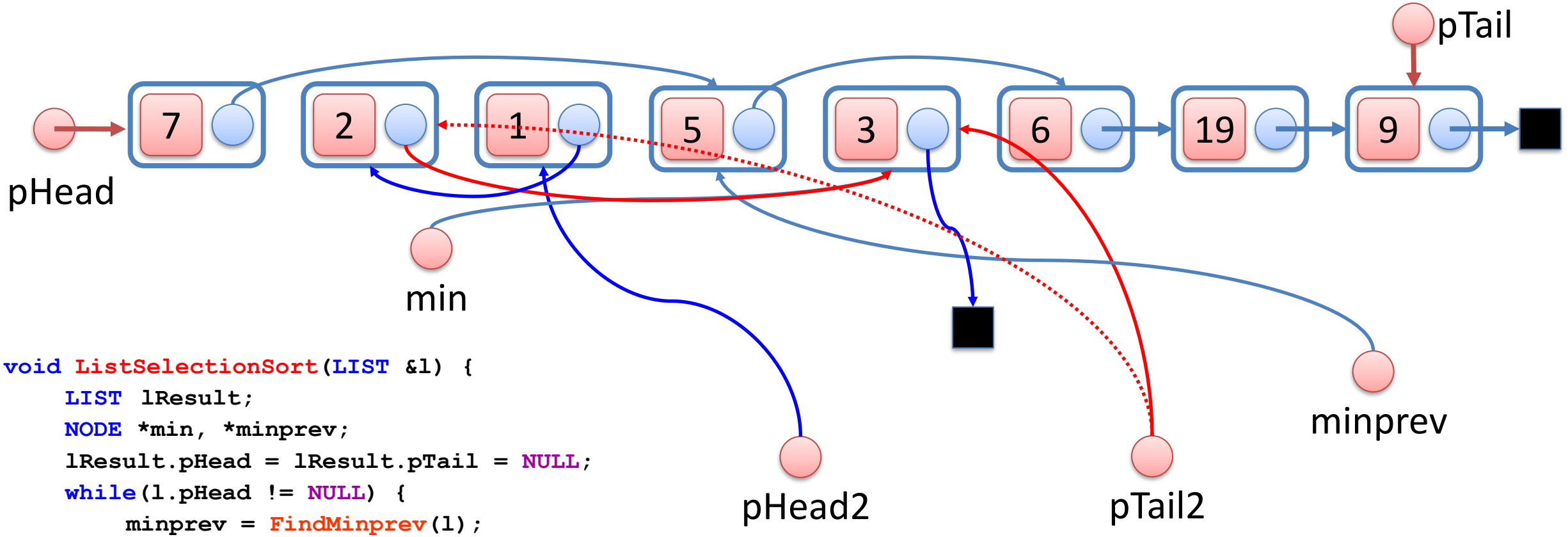

```
void ListSelectionSort(LIST &l) {
     LIST lResult;
     NODE *min, *minprev;
     lResult.pHead = lResult.pTail = NULL;
     while(l.pHead != NULL) {
          minprev = FindMinprev(l);
          min = PickAfter(l, minprev);
          AddTail(lResult, min);
     }
     l = lResult;
}
```



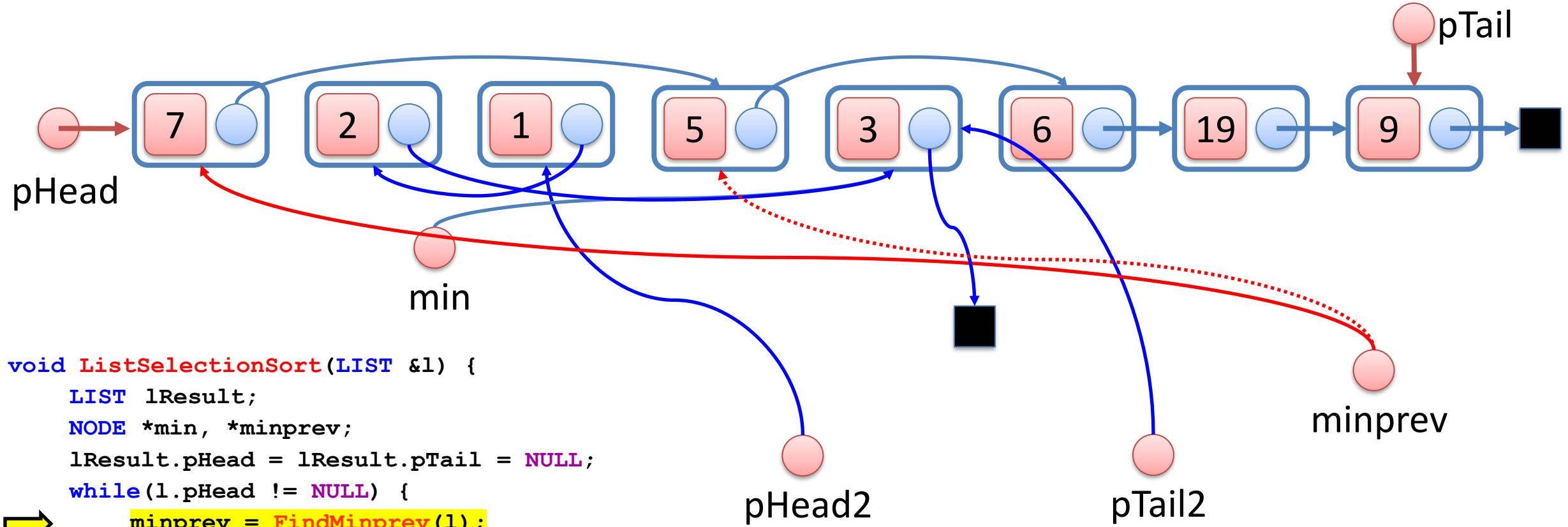

```
void ListSelectionSort(LIST &l) {
     LIST lResult;
     NODE *min, *minprev;
     lResult.pHead = lResult.pTail = NULL;
     while(l.pHead != NULL) {
          minprev = FindMinprev(l);
          min = PickAfter(l, minprev);
          AddTail(lResult, min);
     }
     l = lResult;
}
```



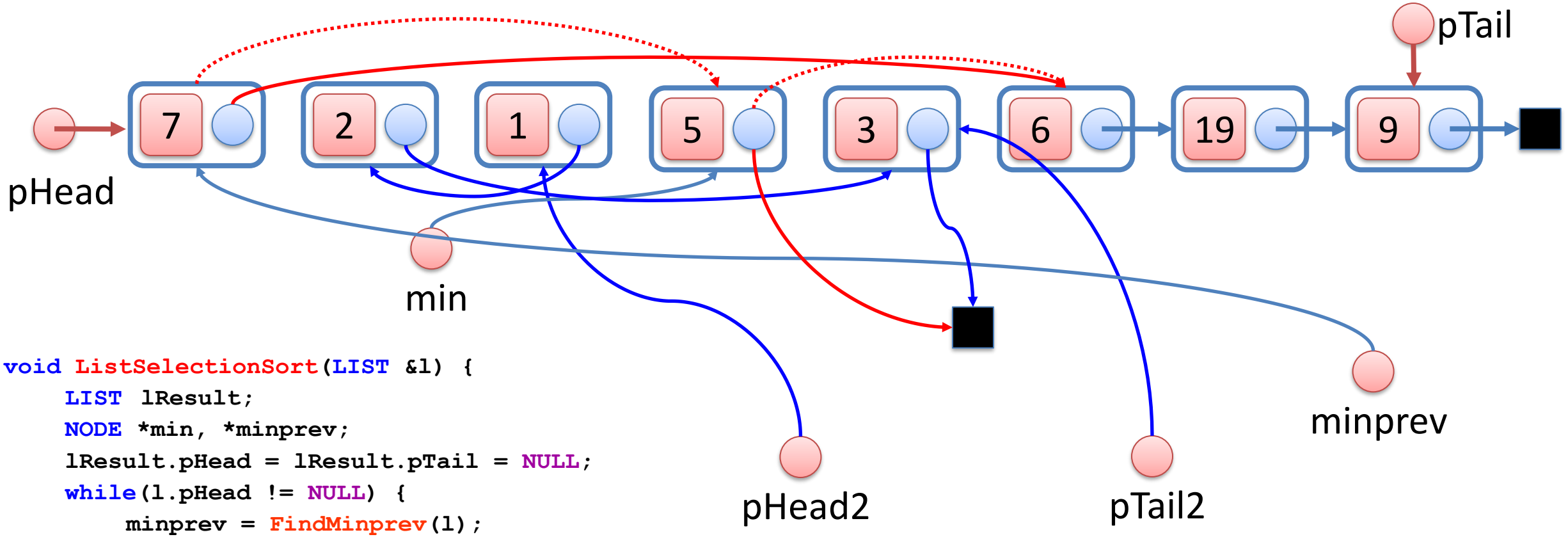

```
void ListSelectionSort(LIST &l) {
     LIST lResult;
     NODE *min, *minprev;
     lResult.pHead = lResult.pTail = NULL;
     while(l.pHead != NULL) {
          minprev = FindMinprev(l);
          min = PickAfter(l, minprev);
          AddTail(lResult, min);
     }
     l = lResult;
}
```



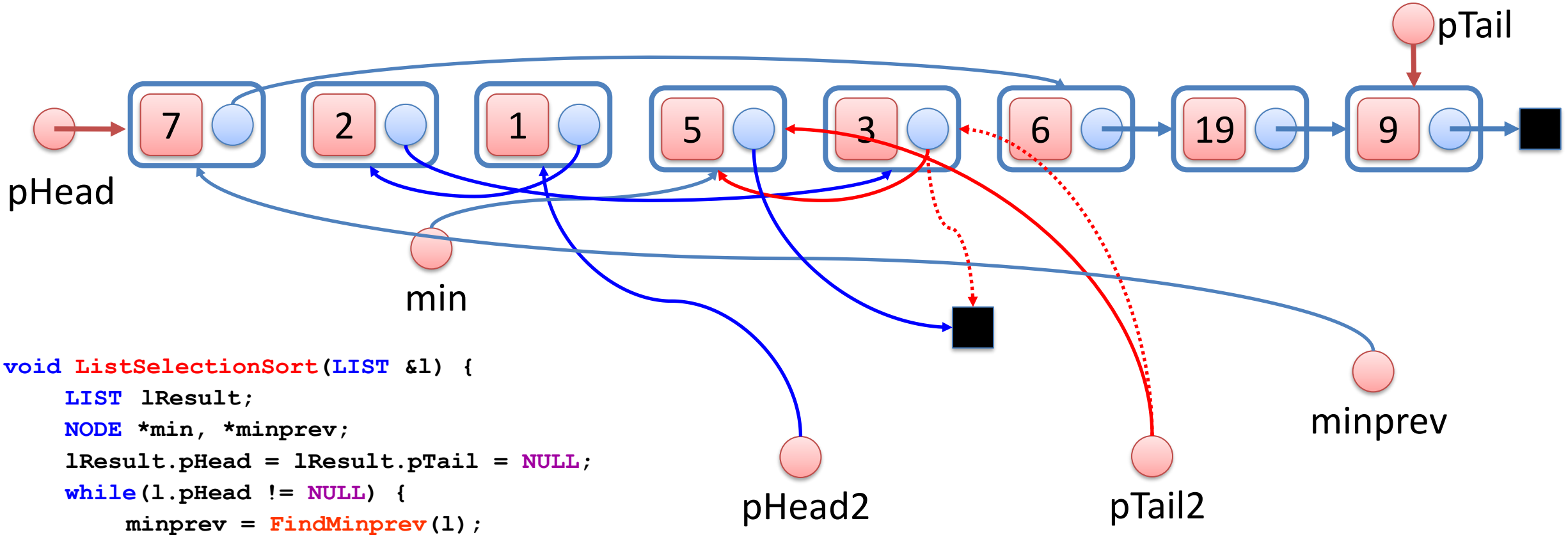

```
void ListSelectionSort(LIST &l) {
     LIST lResult;
     NODE *min, *minprev;
     lResult.pHead = lResult.pTail = NULL;
     while(l.pHead != NULL) {
          minprev = FindMinprev(l);
          min = PickAfter(l, minprev);
          AddTail(lResult, min);
     }
     l = lResult;
}
```



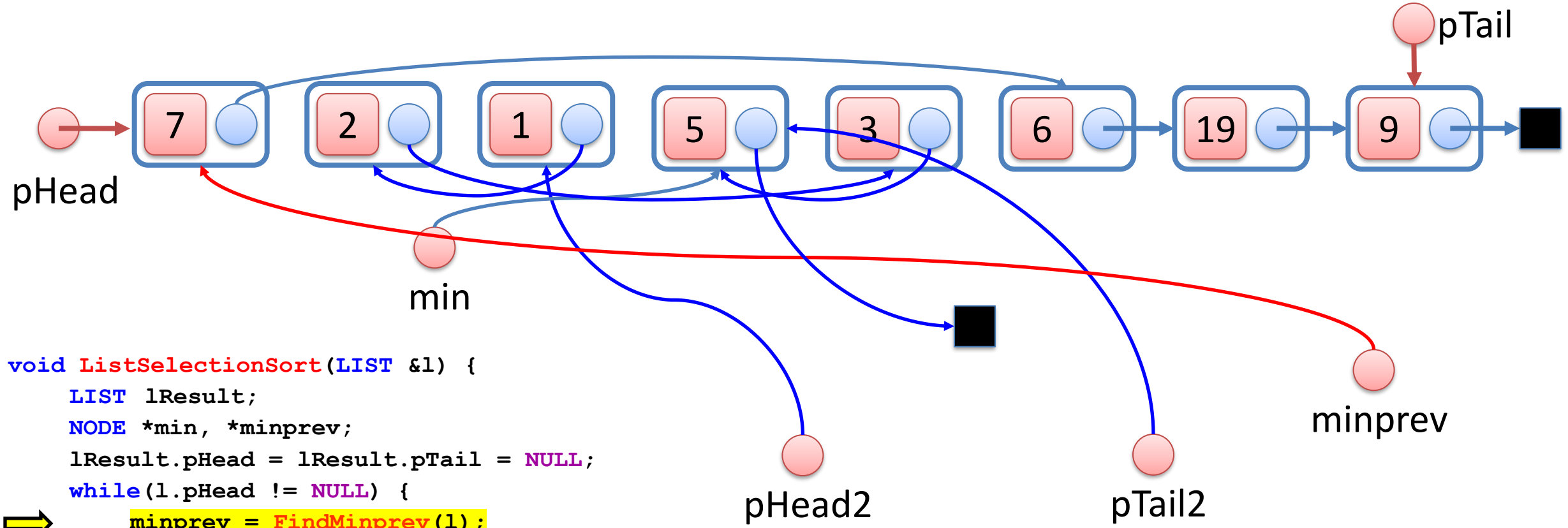

```
void ListSelectionSort(LIST &l) {
     LIST lResult;
     NODE *min, *minprev;
     lResult.pHead = lResult.pTail = NULL;
     while(l.pHead != NULL) {
          minprev = FindMinprev(l);
          min = PickAfter(l, minprev);
          AddTail(lResult, min);
     }
     l = lResult;
}
```



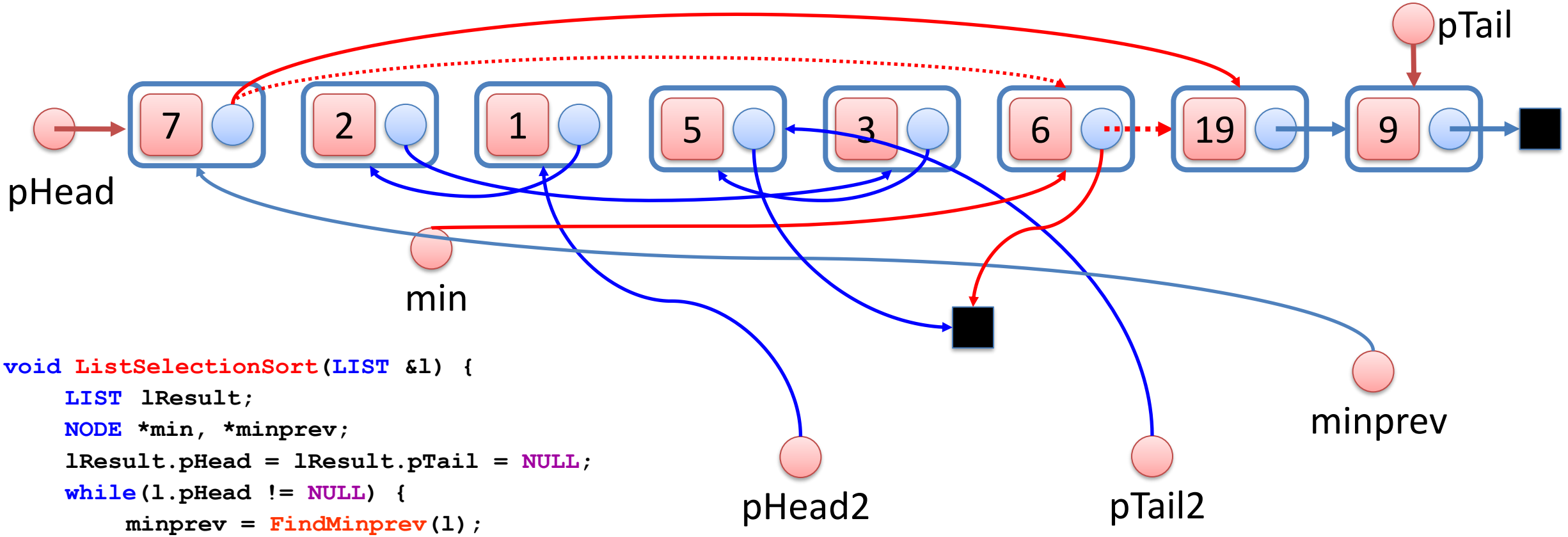

```
void ListSelectionSort(LIST &l) {
     LIST lResult;
     NODE *min, *minprev;
     lResult.pHead = lResult.pTail = NULL;
     while(l.pHead != NULL) {
          minprev = FindMinprev(l);
          min = PickAfter(l, minprev);
          AddTail(lResult, min);
     }
     l = lResult;
}
```



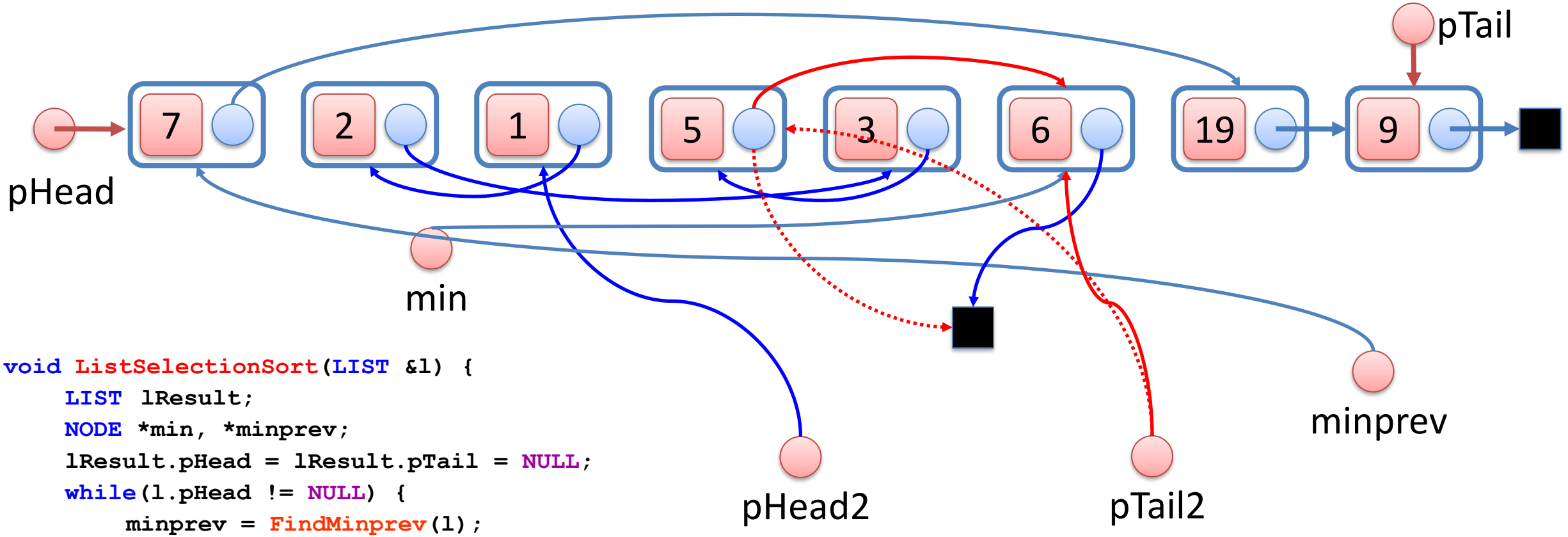

```
void ListSelectionSort(LIST &l) {
     LIST lResult;
     NODE *min, *minprev;
     lResult.pHead = lResult.pTail = NULL;
     while(l.pHead != NULL) {
          minprev = FindMinprev(l);
          min = PickAfter(l, minprev);
          AddTail(lResult, min);
     }
     l = lResult;
}
```



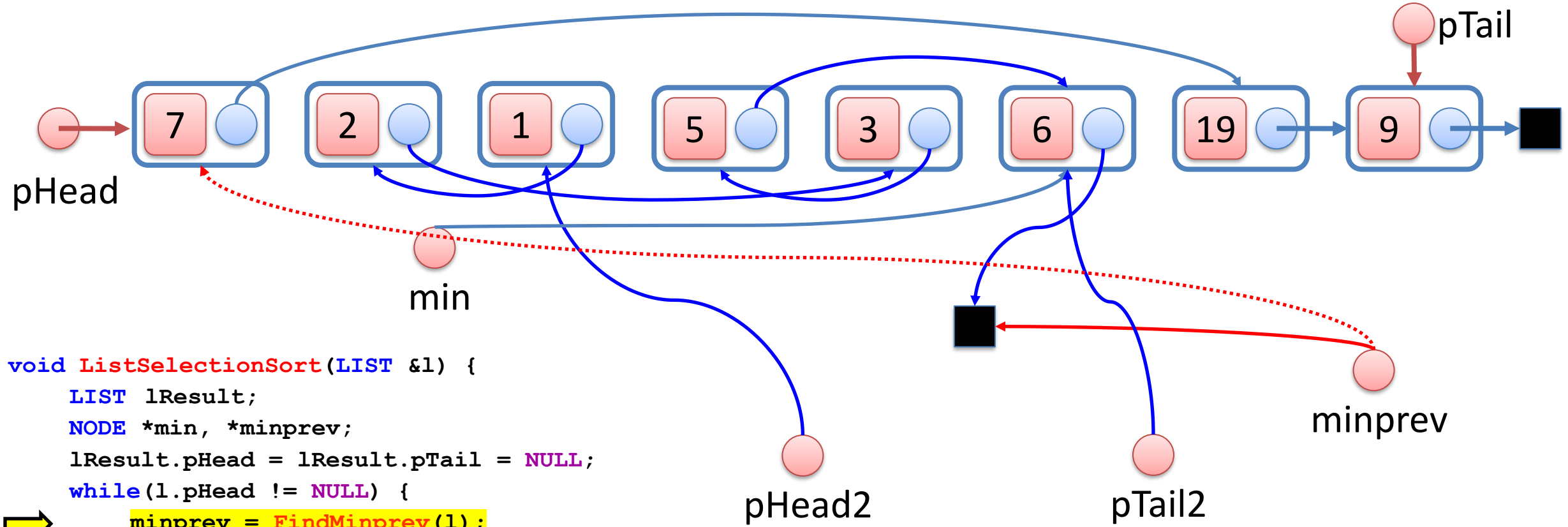

```
void ListSelectionSort(LIST &l) {
     LIST lResult;
     NODE *min, *minprev;
     lResult.pHead = lResult.pTail = NULL;
     while(l.pHead != NULL) {
          minprev = FindMinprev(l);
          min = PickAfter(l, minprev);
          AddTail(lResult, min);
     }
     l = lResult;
}
```



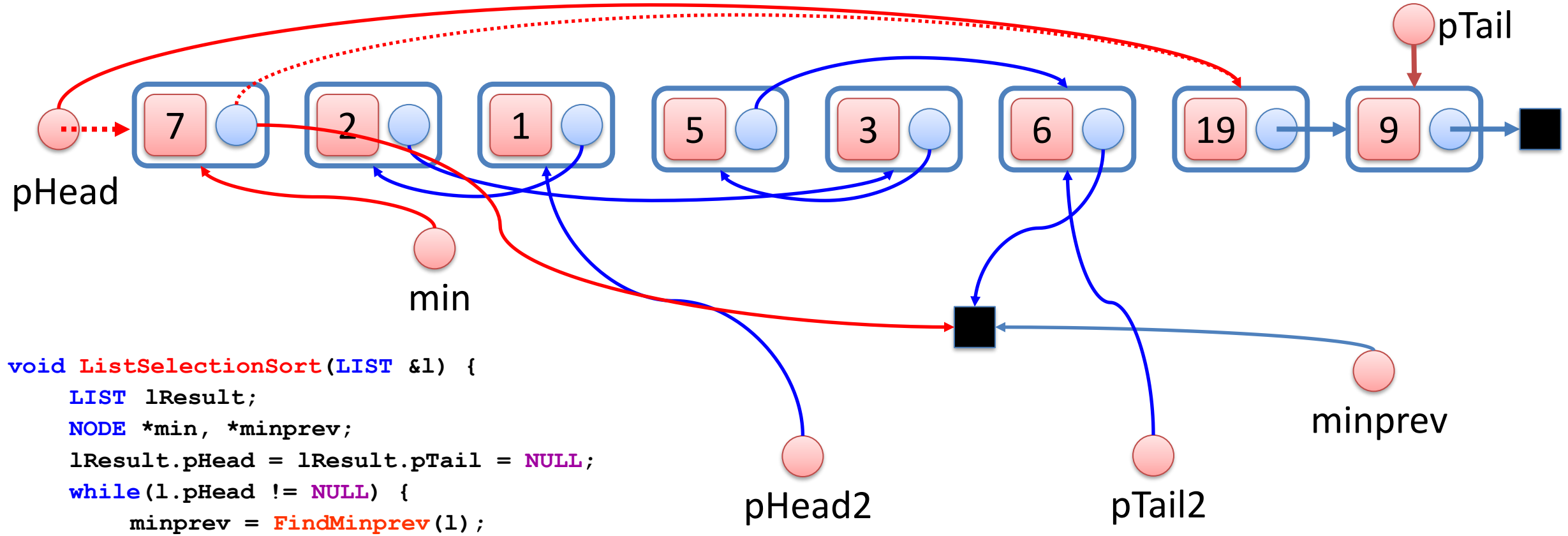

```
void ListSelectionSort(LIST &l) {
     LIST lResult;
     NODE *min, *minprev;
     lResult.pHead = lResult.pTail = NULL;
     while(l.pHead != NULL) {
          minprev = FindMinprev(l);
          min = PickAfter(l, minprev);
          AddTail(lResult, min);
     }
     l = lResult;
}
```



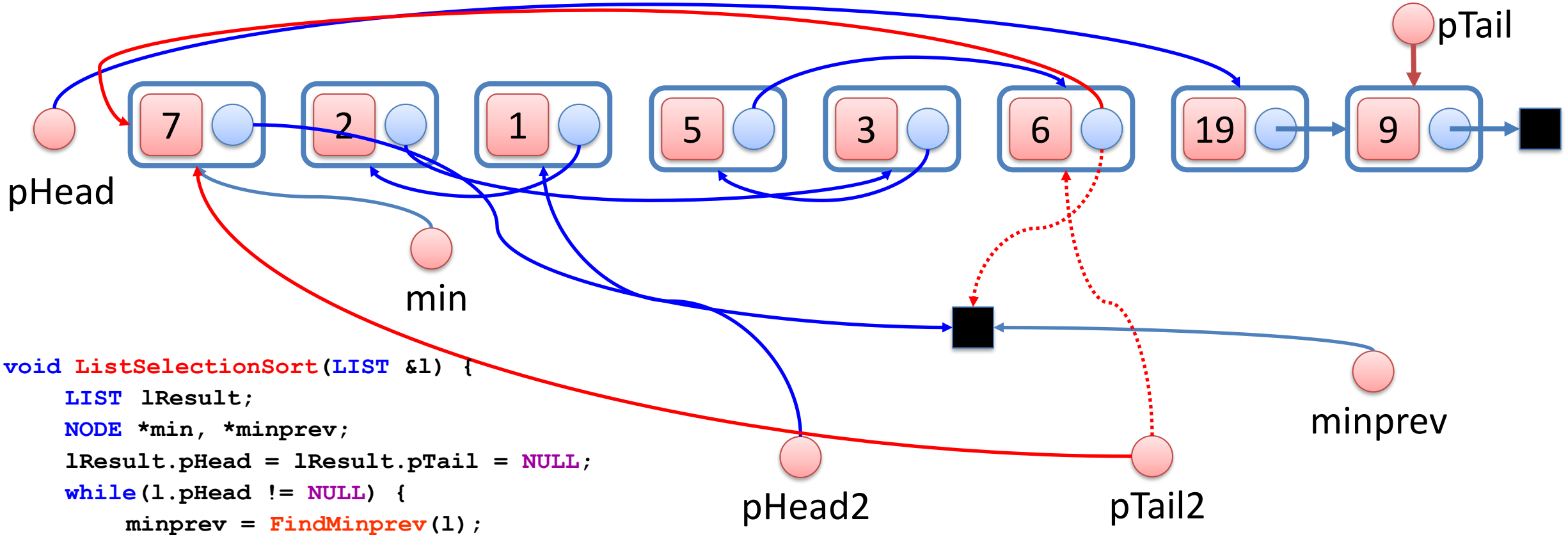

```
void ListSelectionSort(LIST &l) {
     LIST lResult;
     NODE *min, *minprev;
     lResult.pHead = lResult.pTail = NULL;
     while(l.pHead != NULL) {
          minprev = FindMinprev(l);
          min = PickAfter(l, minprev);
          AddTail(lResult, min);
     }
     l = lResult;
}
```



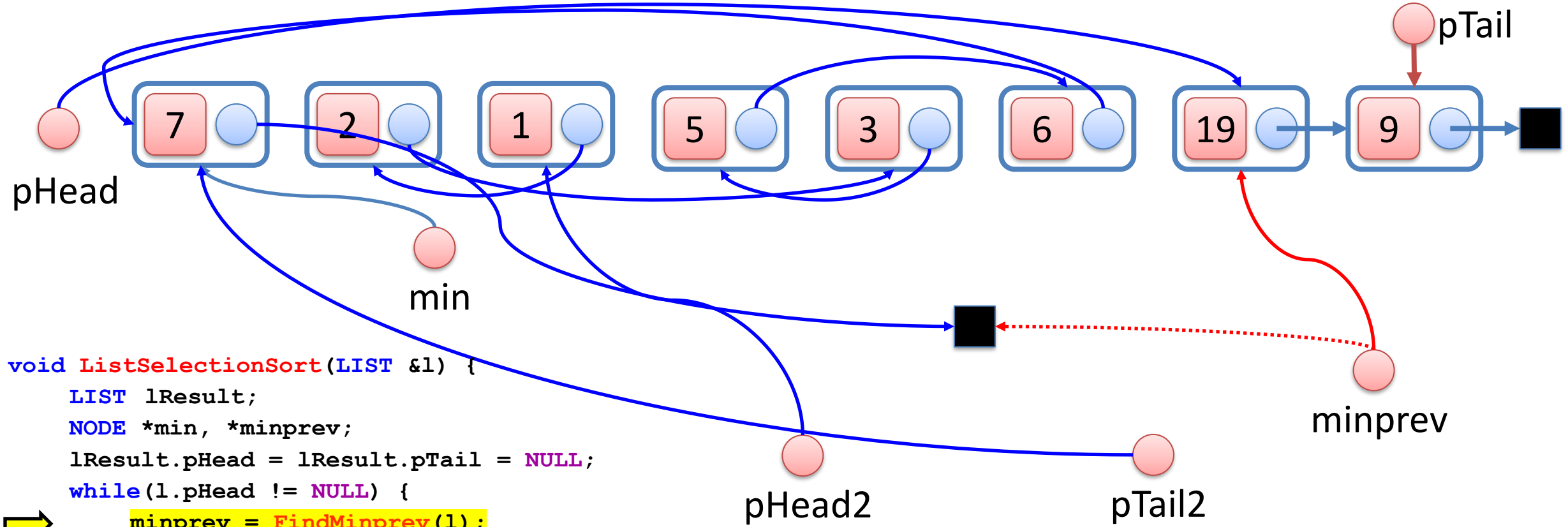

```
void ListSelectionSort(LIST &l) {
     LIST lResult;
     NODE *min, *minprev;
     lResult.pHead = lResult.pTail = NULL;
     while(l.pHead != NULL) {
          minprev = FindMinprev(l);
          min = PickAfter(l, minprev);
          AddTail(lResult, min);
     }
     l = lResult;
}
```



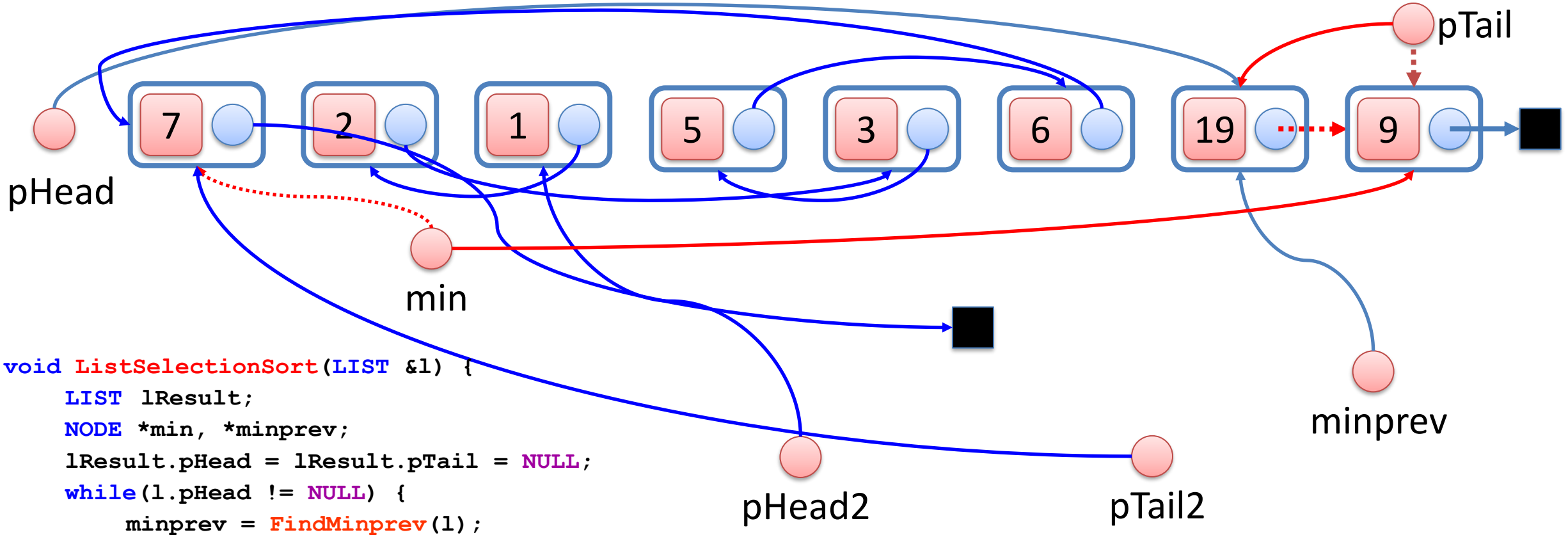

```
void ListSelectionSort(LIST &l) {
     LIST lResult;
     NODE *min, *minprev;
     lResult.pHead = lResult.pTail = NULL;
     while(l.pHead != NULL) {
          minprev = FindMinprev(l);
          min = PickAfter(l, minprev);
          AddTail(lResult, min);
     }
     l = lResult;
}
```



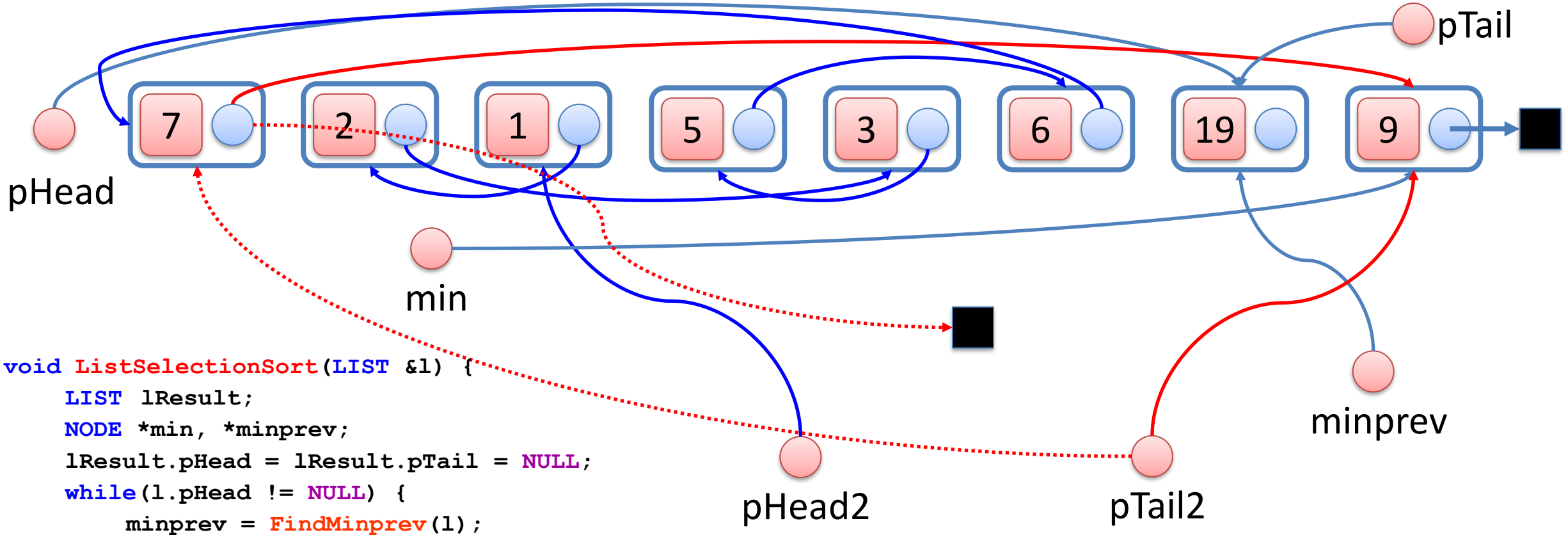

```
void ListSelectionSort(LIST &l) {
     LIST lResult;
     NODE *min, *minprev;
     lResult.pHead = lResult.pTail = NULL;
     while(l.pHead != NULL) {
          minprev = FindMinprev(l);
          min = PickAfter(l, minprev);
          AddTail(lResult, min);
     }
     l = lResult;
}
```



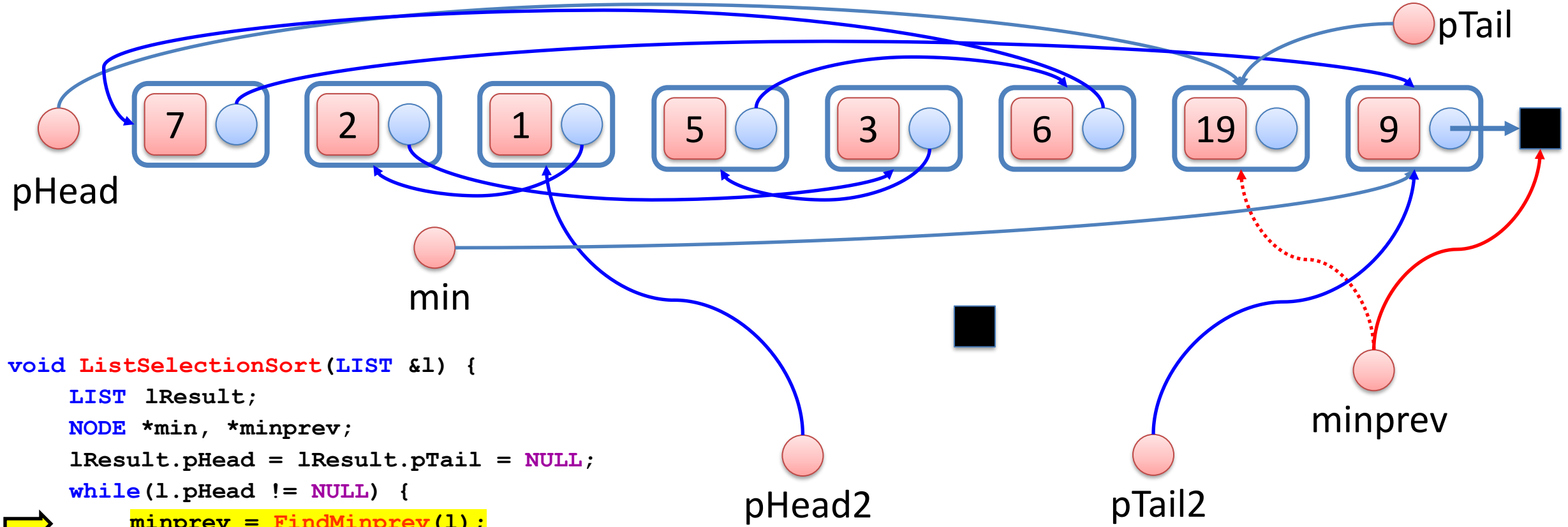

```
void ListSelectionSort(LIST &l) {
     LIST lResult;
     NODE *min, *minprev;
     lResult.pHead = lResult.pTail = NULL;
     while(l.pHead != NULL) {
          minprev = FindMinprev(l);
          min = PickAfter(l, minprev);
          AddTail(lResult, min);
     }
     l = lResult;
}
```



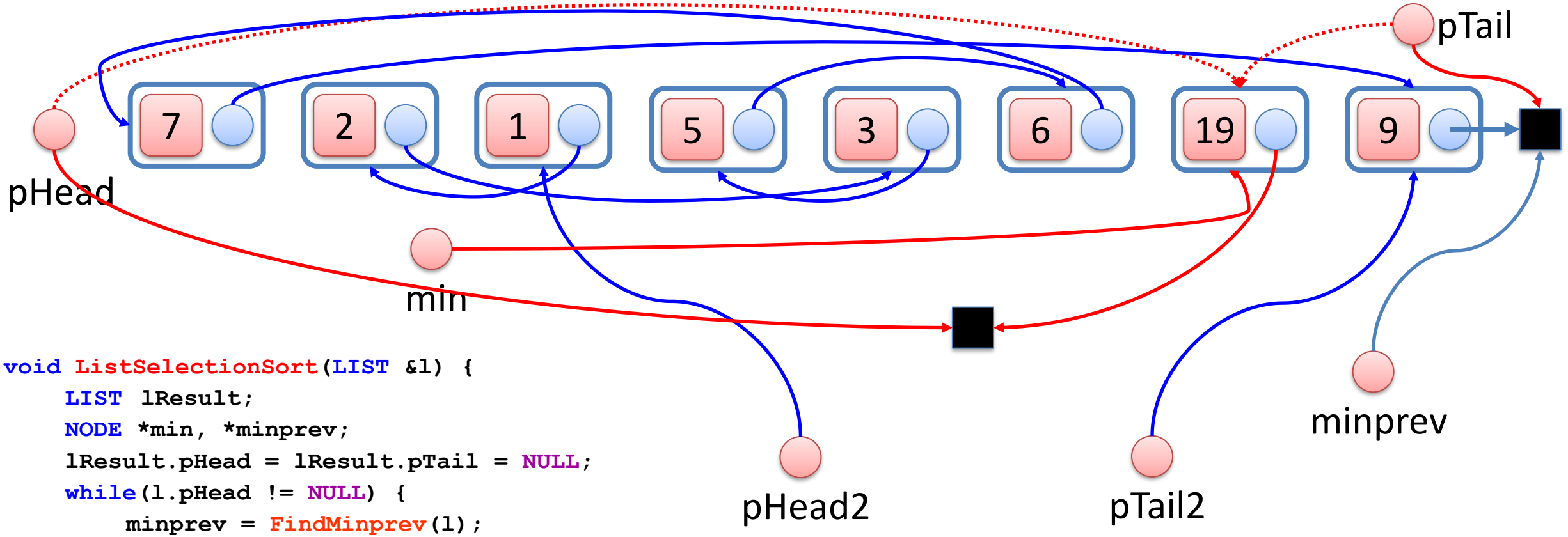

```
void ListSelectionSort(LIST &l) {
     LIST lResult;
     NODE *min, *minprev;
     lResult.pHead = lResult.pTail = NULL;
     while(l.pHead != NULL) {
          minprev = FindMinprev(l);
          min = PickAfter(l, minprev);
          AddTail(lResult, min);
     }
     l = lResult;
}
```



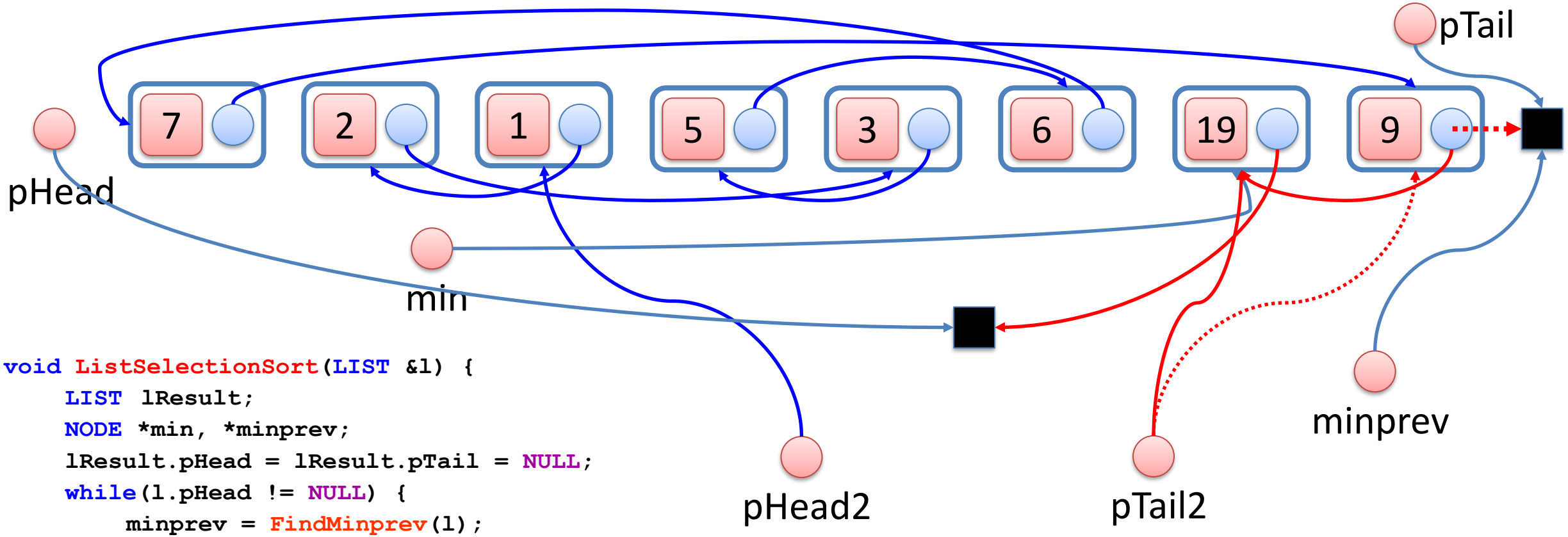

```
void ListSelectionSort(LIST &l) {
     LIST lResult;
     NODE *min, *minprev;
     lResult.pHead = lResult.pTail = NULL;
     while(l.pHead != NULL) {
          minprev = FindMinprev(l);
          min = PickAfter(l, minprev);
          AddTail(lResult, min);
     }
     l = lResult;
}
```



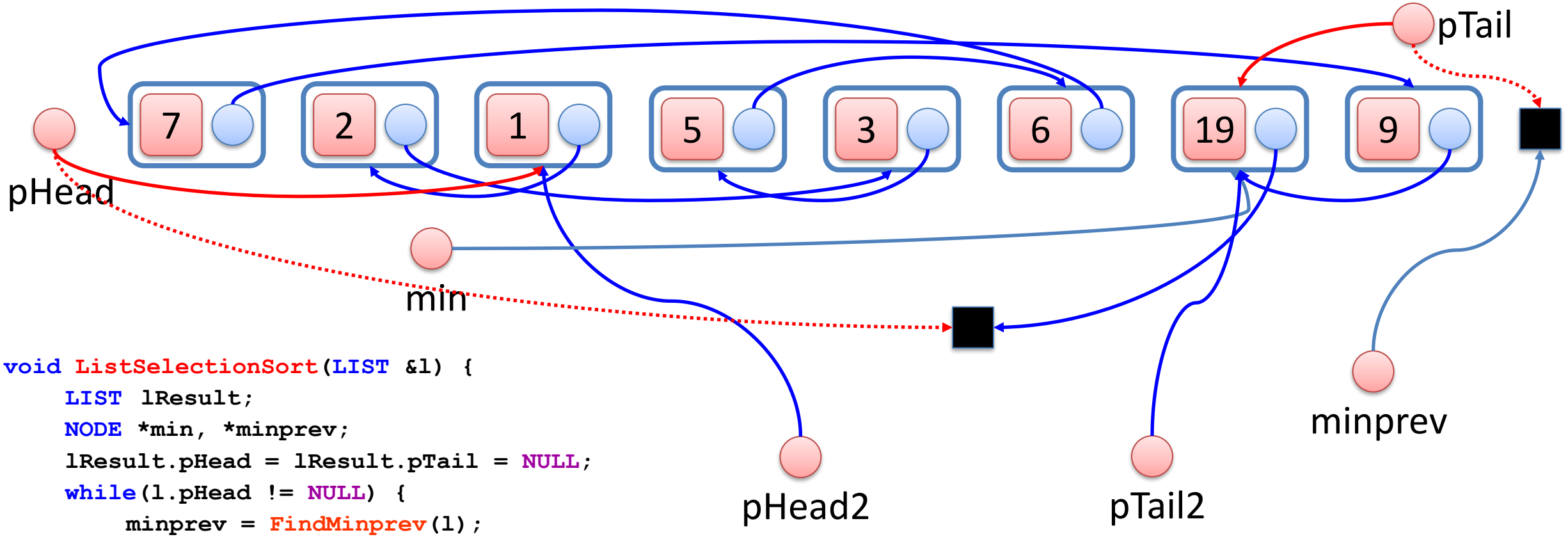

```
void ListSelectionSort(LIST &l) {
     LIST lResult;
     NODE *min, *minprev;
     lResult.pHead = lResult.pTail = NULL;
     while(l.pHead != NULL) {
          minprev = FindMinprev(l);
          min = PickAfter(l, minprev);
          AddTail(lResult, min);
     }
     l = lResult;
}
```



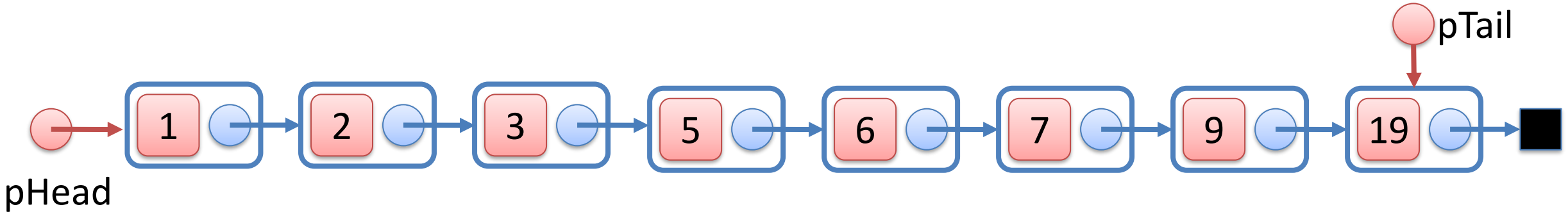

```
void ListSelectionSort(LIST &l) {
     LIST lResult;
     NODE *min, *minprev;
     lResult.pHead = lResult.pTail = NULL;
     while(l.pHead != NULL) {
          minprev = FindMinprev(l);
          min = PickAfter(l, minprev);
          AddTail(lResult, min);
     }
     l = lResult;
}
```




```
void ListSelectionSort(LIST &l) {
     LIST lResult;
     NODE *min, *minprev;
     lResult.pHead = lResult.pTail = NULL;
     while(l.pHead != NULL) {
          minprev = FindMinprev(l);
          min = PickAfter(l, minprev);
          AddTail(lResult, min);
     }
     l = lResult;
}
```



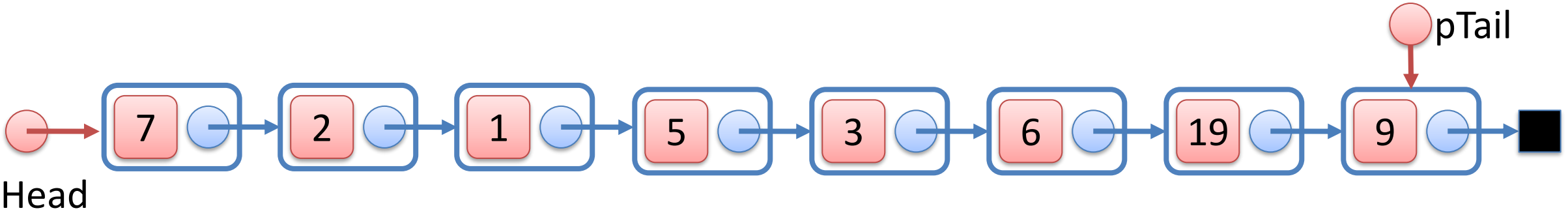

```
void ListSelectionSort(LIST &l) {
     LIST lResult;
     NODE *min, *minprev;
     lResult.pHead = lResult.pTail = NULL;
     while(l.pHead != NULL) {
          minprev = FindMinprev(l);
          min = PickAfter(l, minprev);
          AddTail(lResult, min);
     }
     l = lResult;
}
```


pTail
1
2
3
5
6
7
9
19
pHead

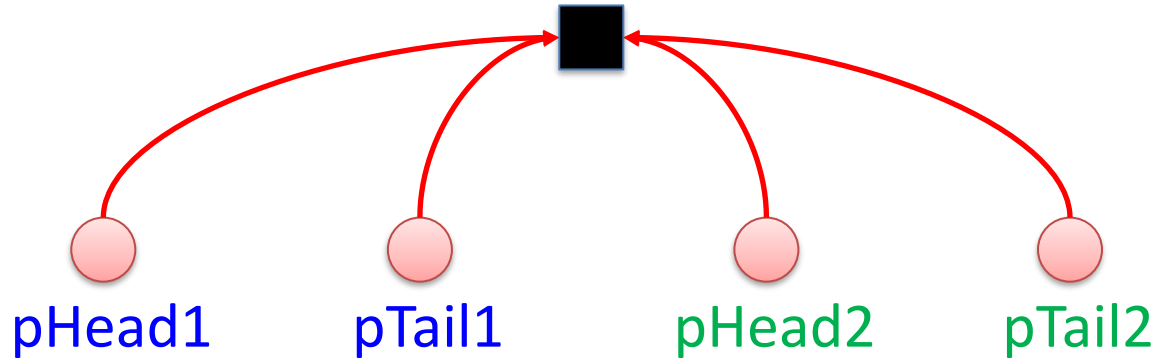
List-based quick sort

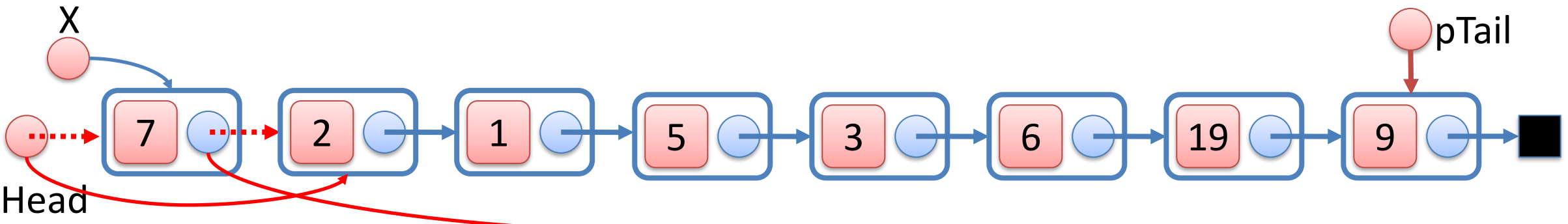
```
void ListQuickSort(LIST &list) {
    NODE *X, *p;
    LIST list1, list2;

    if (list.pHead == list.pTail)
        return;

    Init(list1);
    Init(list2);
    X = PickHead(list);
    while (list.pHead != NULL) {
        p = PickHead(list);
        if (p->Info <= X->Info)
            AddTail(list1, p);
        else
            AddTail(list2, p);
    }
    ListQuickSort(list1);
    ListQuickSort(list2);
    LISTAppend(list, list1);
    AddTail(list, X);
    LISTAppend(list, list2);
}
```

```
void LISTAppend(LIST &list, LIST &list2)
{
    if (list2.pHead == NULL) return;
    if (list.pHead == NULL)
        list = list2;
    else {
        list.pTail->pNext = list2.pHead;
        list.pTail = list2.pTail;
    }
    Init(list2);
}
```

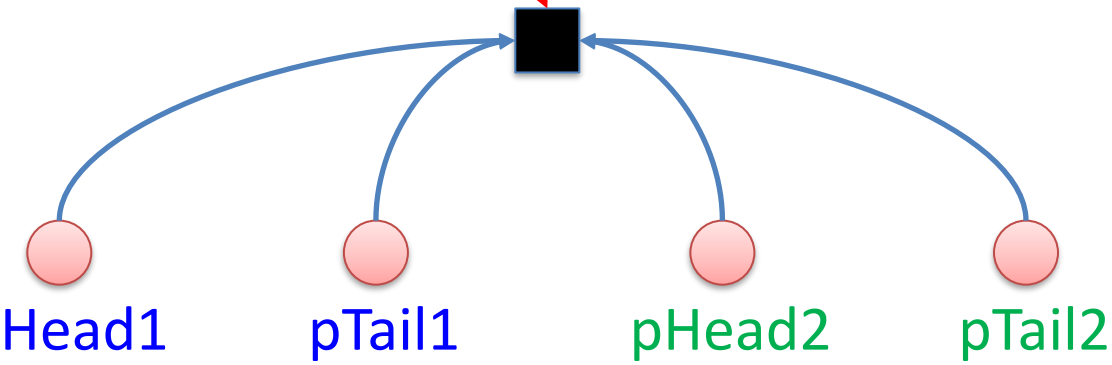


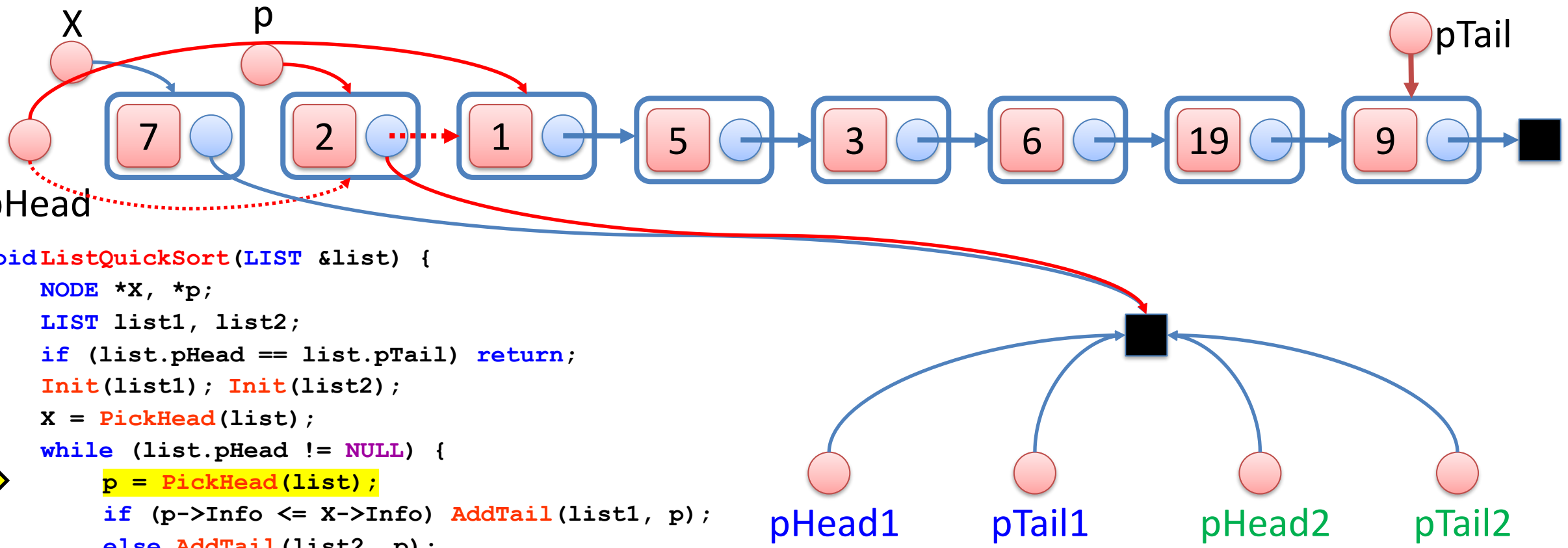

```
void ListQuickSort(LIST &list) {
    NODE *X, *p;
    LIST list1, list2;
    if (list.pHead == list.pTail) return;
    Init(list1); Init(list2);
    X = PickHead(list);
    while (list.pHead != NULL) {
        p = PickHead(list);
        if (p->Info <= X->Info) AddTail(list1, p);
        else AddTail(list2, p);
    }
    ListQuickSort(list1); ListQuickSort(list2);
    LISTAppend(list, list1);
    AddTail(list, X);
    LISTAppend(list, list2);
}
```



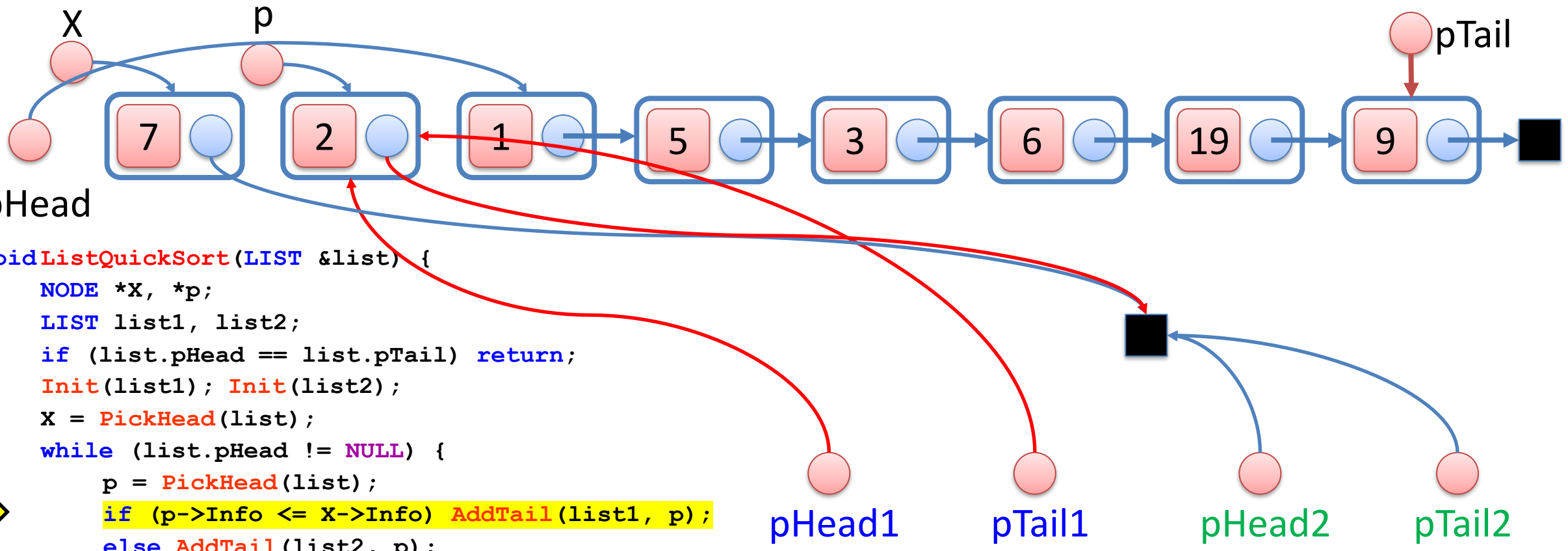



```
void ListQuickSort(LIST &list) {
    NODE *X, *p;
    LIST list1, list2;
    if (list.pHead == list.pTail) return;
    Init(list1); Init(list2);
    X = PickHead(list);
    while (list.pHead != NULL) {
        p = PickHead(list);
        if (p->Info <= X->Info) AddTail(list1, p);
        else AddTail(list2, p);
    }
    ListQuickSort(list1); ListQuickSort(list2);
    LISTAppend(list, list1);
    AddTail(list, X);
    LISTAppend(list, list2);
}
```

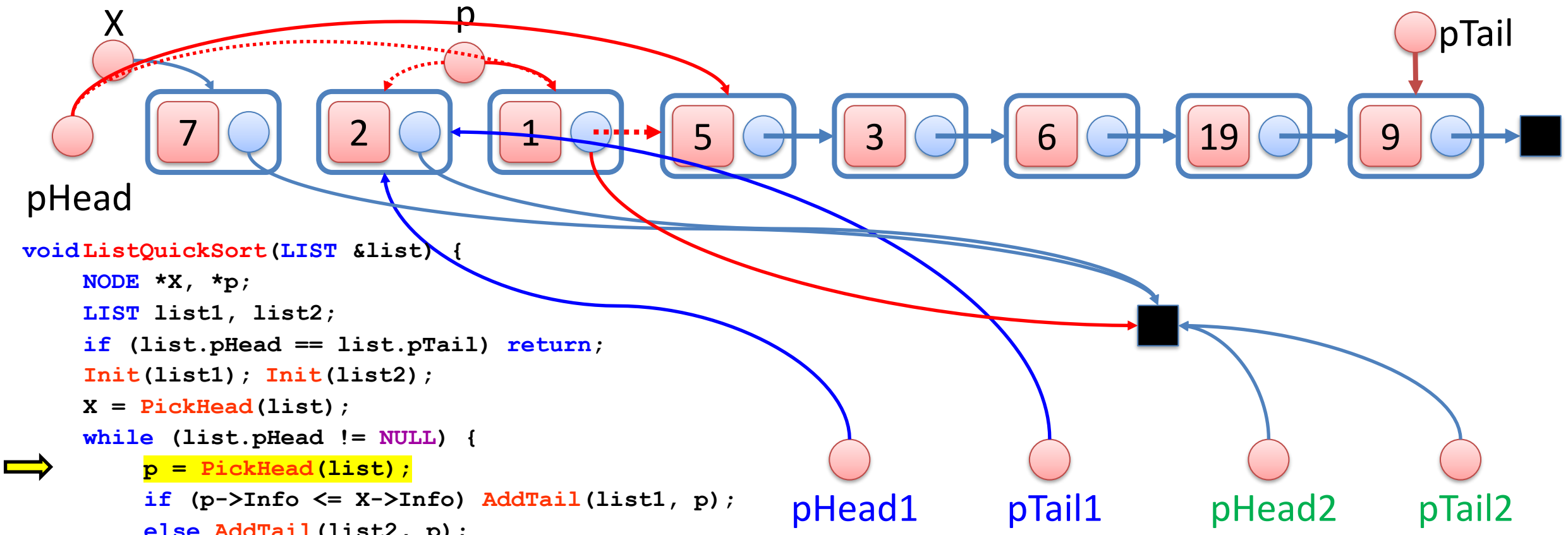




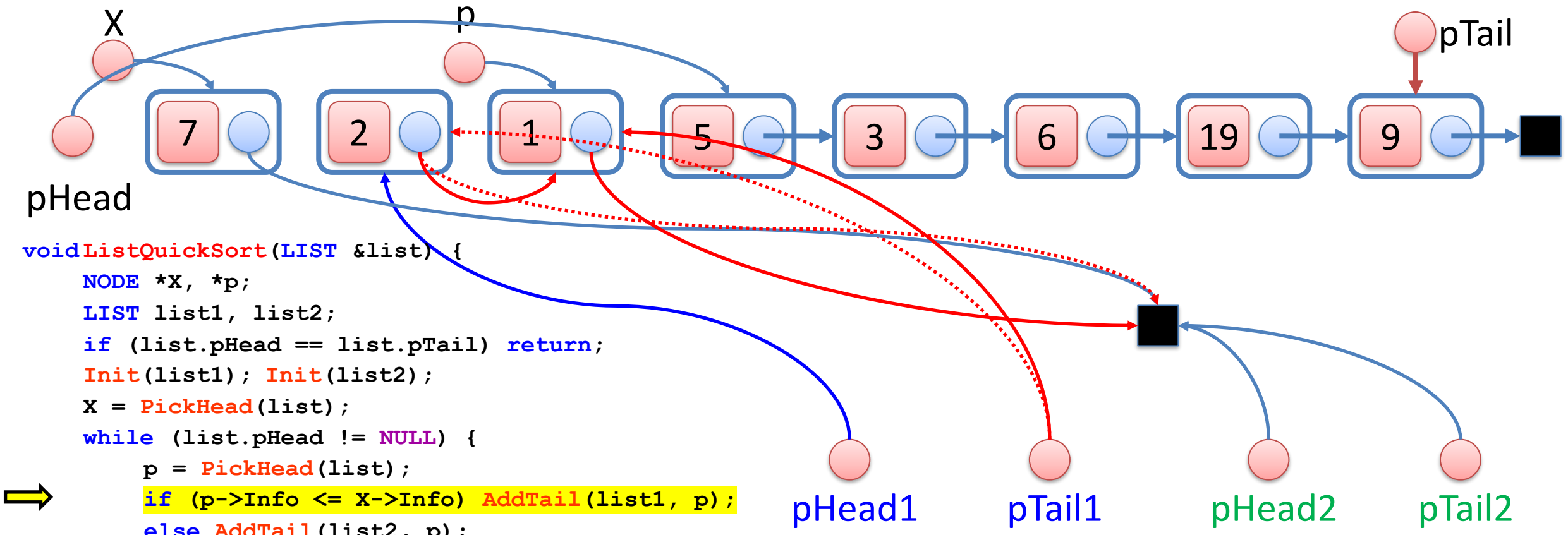

```
void ListQuickSort(LIST &list) {
    NODE *X, *p;
    LIST list1, list2;
    if (list.pHead == list.pTail) return;
    Init(list1); Init(list2);
    X = PickHead(list);
    while (list.pHead != NULL) {
        p = PickHead(list);
        if (p->Info <= X->Info) AddTail(list1, p);
        else AddTail(list2, p);
    }
    ListQuickSort(list1); ListQuickSort(list2);
    LISTAppend(list, list1);
    AddTail(list, X);
    LISTAppend(list, list2);
}
```



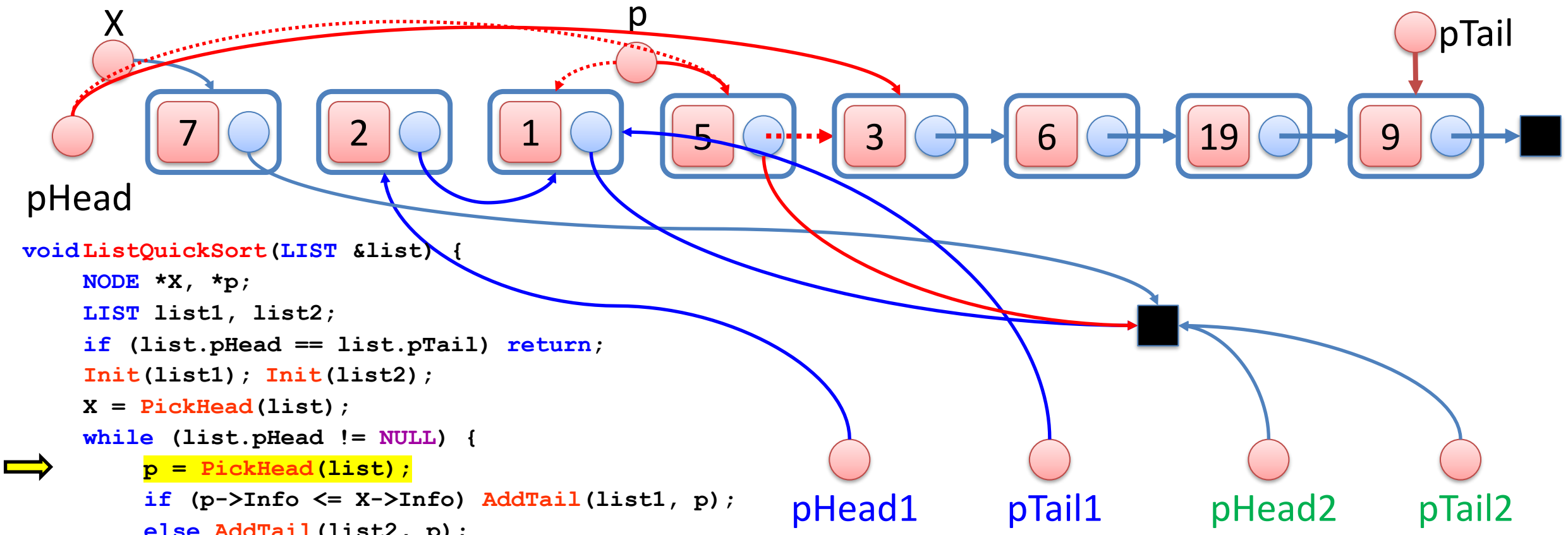

```
void ListQuickSort(LIST &list) {
    NODE *X, *p;
    LIST list1, list2;
    if (list.pHead == list.pTail) return;
    Init(list1); Init(list2);
    X = PickHead(list);
    while (list.pHead != NULL) {
        p = PickHead(list);
        if (p->Info <= X->Info) AddTail(list1, p);
        else AddTail(list2, p);
    }
    ListQuickSort(list1); ListQuickSort(list2);
    LISTAppend(list, list1);
    AddTail(list, X);
    LISTAppend(list, list2);
}
```



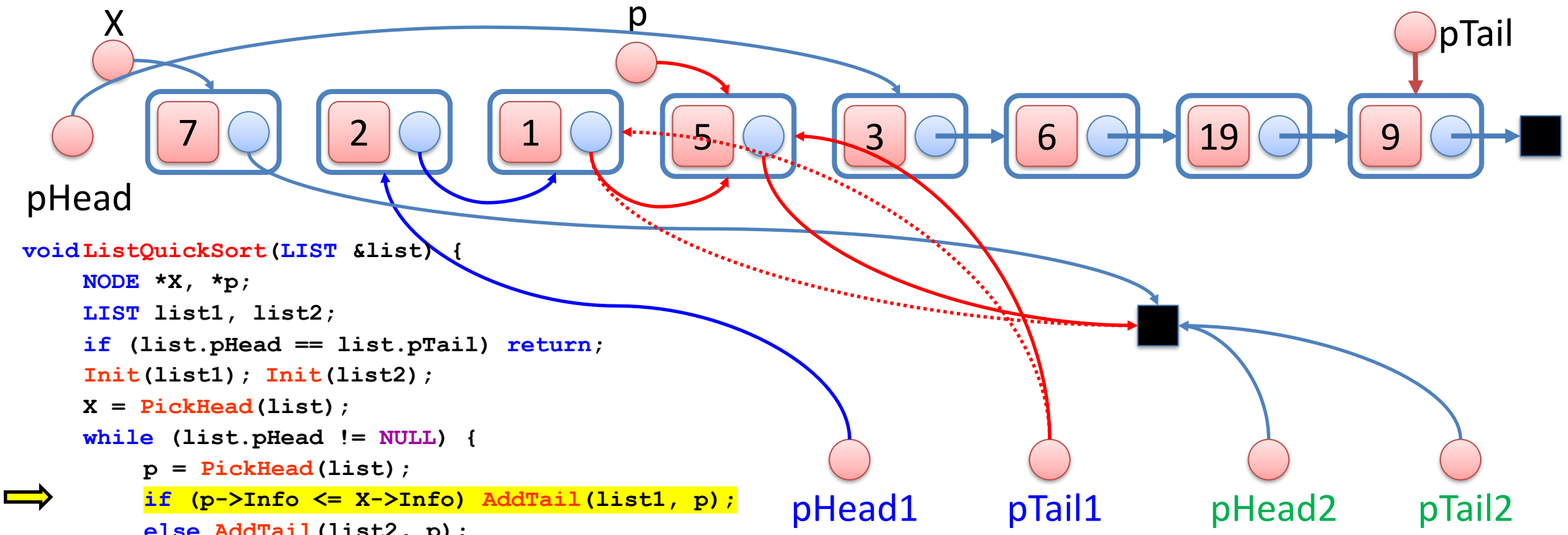

```
void ListQuickSort(LIST &list) {
    NODE *X, *p;
    LIST list1, list2;
    if (list.pHead == list.pTail) return;
    Init(list1); Init(list2);
    X = PickHead(list);
    while (list.pHead != NULL) {
        p = PickHead(list);
        if (p->Info <= X->Info) AddTail(list1, p);
        else AddTail(list2, p);
    }
    ListQuickSort(list1); ListQuickSort(list2);
    LISTAppend(list, list1);
    AddTail(list, X);
    LISTAppend(list, list2);
}
```



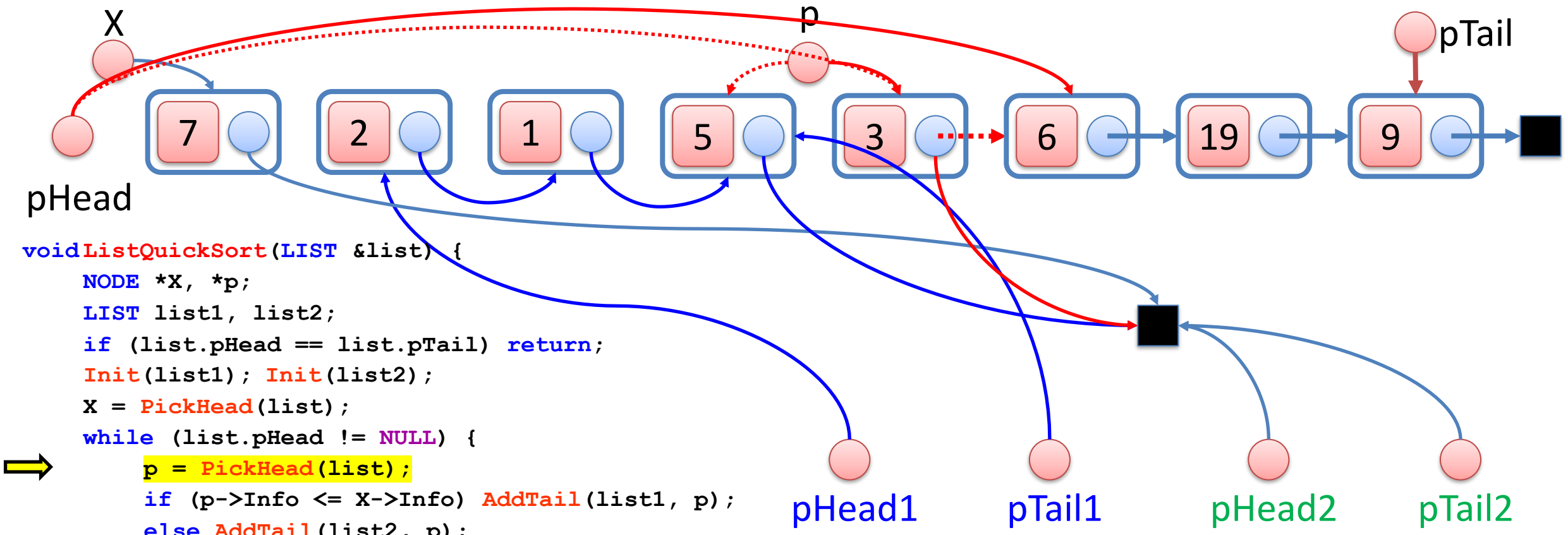

```
void ListQuickSort(LIST &list) {
    NODE *X, *p;
    LIST list1, list2;
    if (list.pHead == list.pTail) return;
    Init(list1); Init(list2);
    X = PickHead(list);
    while (list.pHead != NULL) {
        p = PickHead(list);
        if (p->Info <= X->Info) AddTail(list1, p);
        else AddTail(list2, p);
    }
    ListQuickSort(list1); ListQuickSort(list2);
    LISTAppend(list, list1);
    AddTail(list, X);
    LISTAppend(list, list2);
}
```



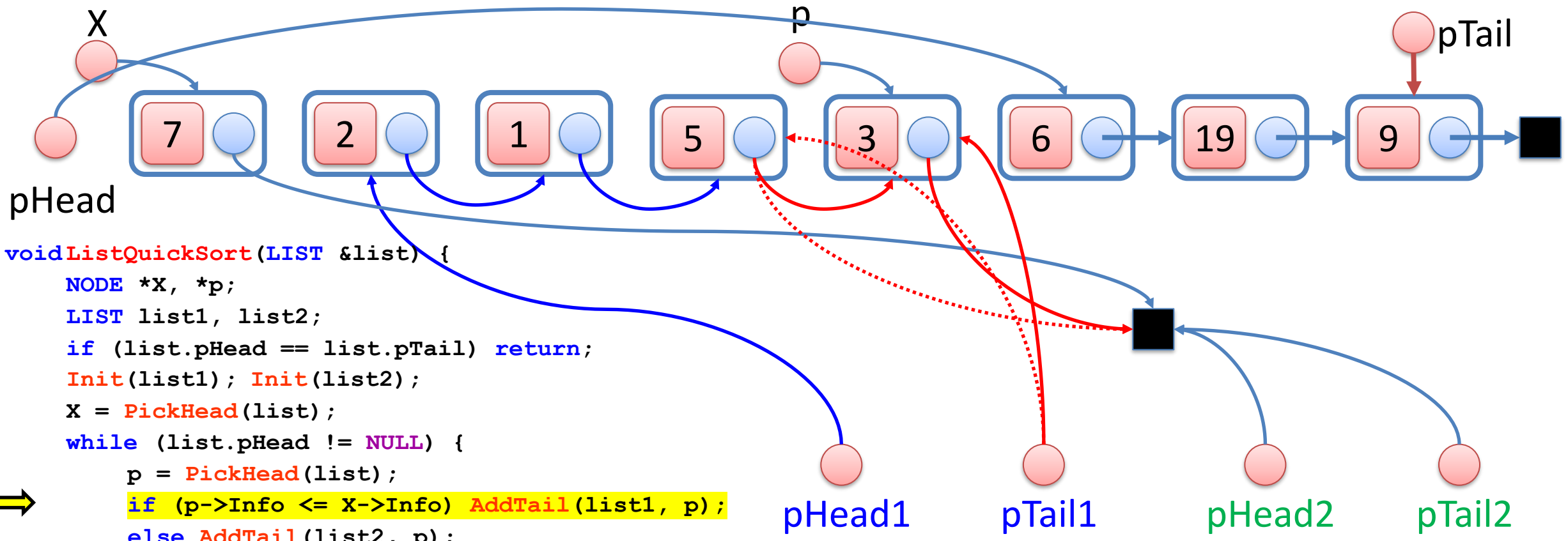

```
void ListQuickSort(LIST &list) {
    NODE *X, *p;
    LIST list1, list2;
    if (list.pHead == list.pTail) return;
    Init(list1); Init(list2);
    X = PickHead(list);
    while (list.pHead != NULL) {
        p = PickHead(list);
        if (p->Info <= X->Info) AddTail(list1, p);
        else AddTail(list2, p);
    }
    ListQuickSort(list1); ListQuickSort(list2);
    LISTAppend(list, list1);
    AddTail(list, X);
    LISTAppend(list, list2);
}
```



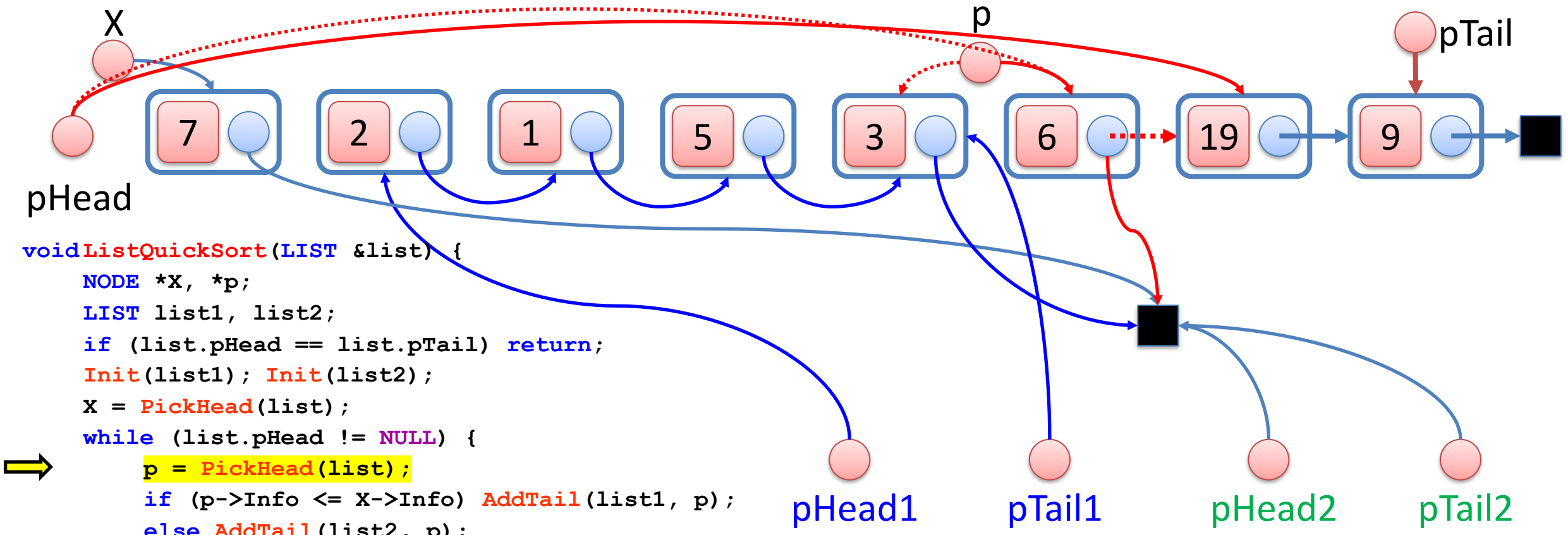

```
void ListQuickSort(LIST &list) {
    NODE *X, *p;
    LIST list1, list2;
    if (list.pHead == list.pTail) return;
    Init(list1); Init(list2);
    X = PickHead(list);
    while (list.pHead != NULL) {
        p = PickHead(list);
        if (p->Info <= X->Info) AddTail(list1, p);
        else AddTail(list2, p);
    }
    ListQuickSort(list1); ListQuickSort(list2);
    LISTAppend(list, list1);
    AddTail(list, X);
    LISTAppend(list, list2);
}
```



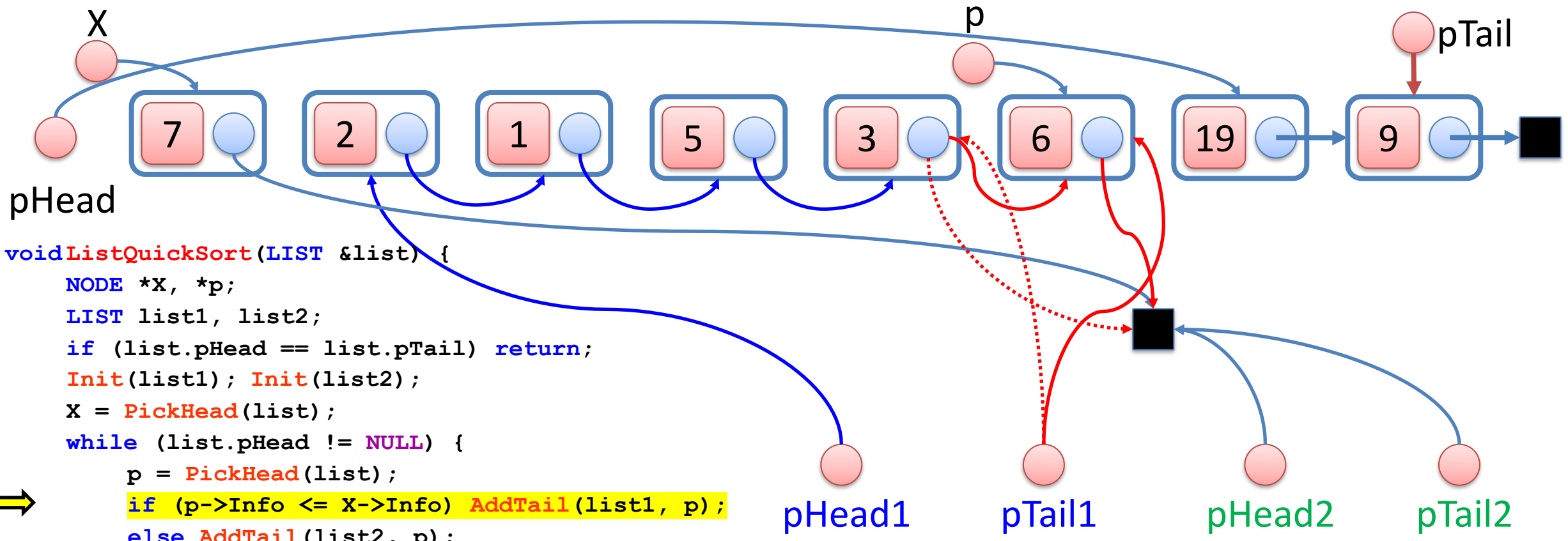

```
void ListQuickSort(LIST &list) {
    NODE *X, *p;
    LIST list1, list2;
    if (list.pHead == list.pTail) return;
    Init(list1); Init(list2);
    X = PickHead(list);
    while (list.pHead != NULL) {
        p = PickHead(list);
        if (p->Info <= X->Info) AddTail(list1, p);
        else AddTail(list2, p);
    }
    ListQuickSort(list1); ListQuickSort(list2);
    LISTAppend(list, list1);
    AddTail(list, X);
    LISTAppend(list, list2);
}
```



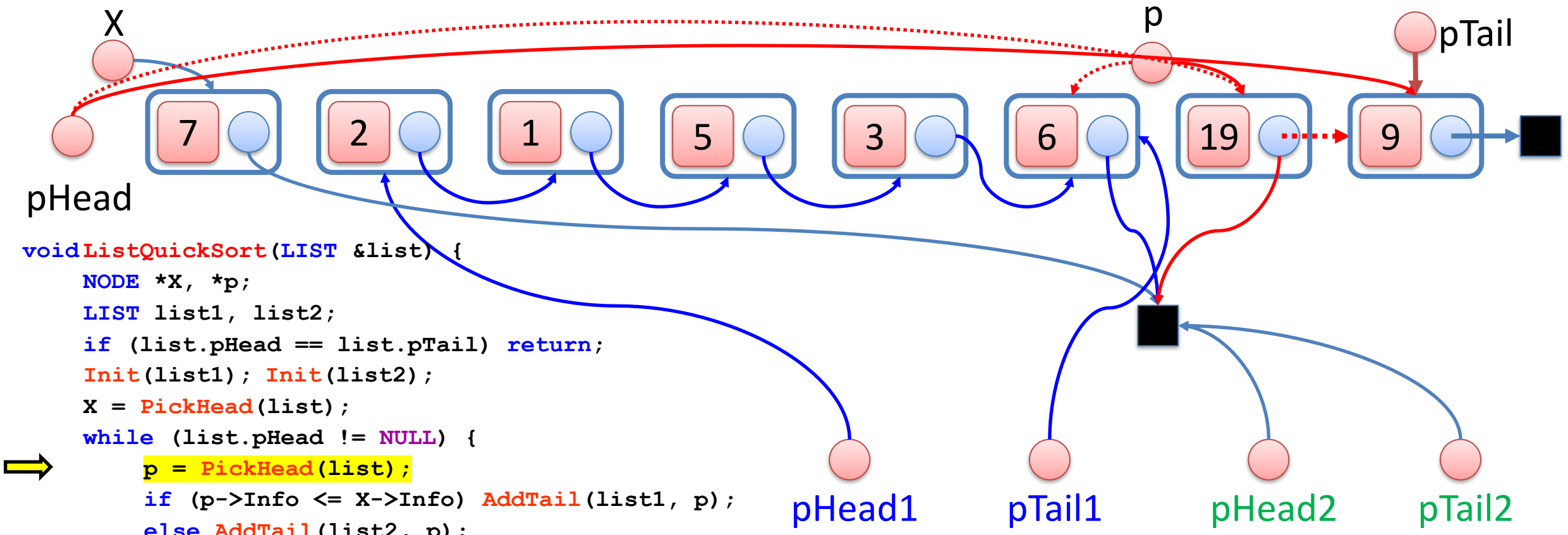

```
void ListQuickSort(LIST &list) {
    NODE *X, *p;
    LIST list1, list2;
    if (list.pHead == list.pTail) return;
    Init(list1); Init(list2);
    X = PickHead(list);
    while (list.pHead != NULL) {
        p = PickHead(list);
        if (p->Info <= X->Info) AddTail(list1, p);
        else AddTail(list2, p);
    }
    ListQuickSort(list1); ListQuickSort(list2);
    LISTAppend(list, list1);
    AddTail(list, X);
    LISTAppend(list, list2);
}
```



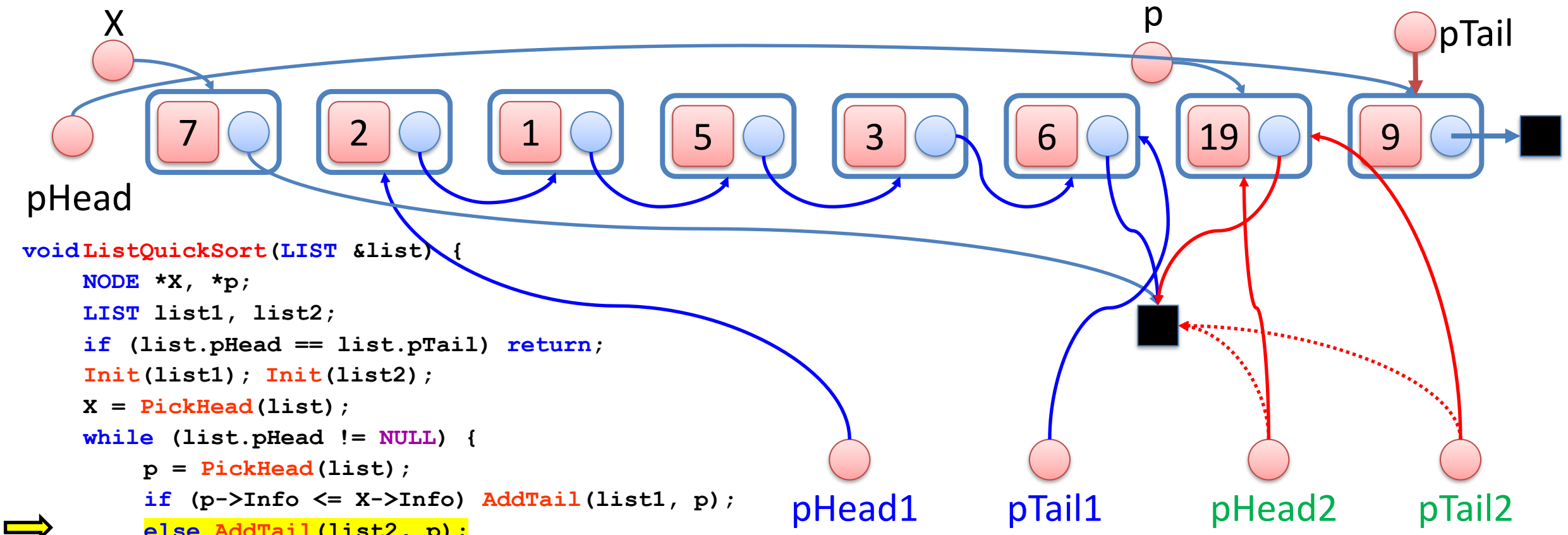

```
void ListQuickSort(LIST &list) {
    NODE *X, *p;
    LIST list1, list2;
    if (list.pHead == list.pTail) return;
    Init(list1); Init(list2);
    X = PickHead(list);
    while (list.pHead != NULL) {
        p = PickHead(list);
        if (p->Info <= X->Info) AddTail(list1, p);
        else AddTail(list2, p);
    }
    ListQuickSort(list1); ListQuickSort(list2);
    LISTAppend(list, list1);
    AddTail(list, X);
    LISTAppend(list, list2);
}
```



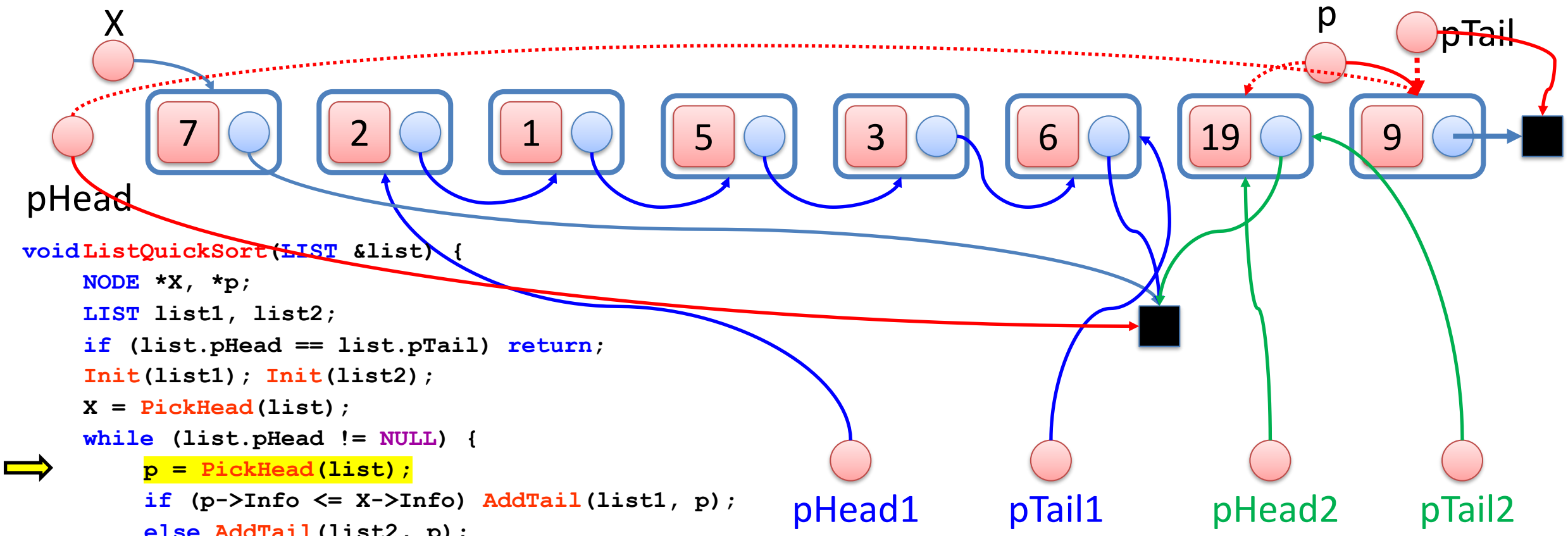

```
void ListQuickSort(LIST &list) {
    NODE *X, *p;
    LIST list1, list2;
    if (list.pHead == list.pTail) return;
    Init(list1); Init(list2);
    X = PickHead(list);
    while (list.pHead != NULL) {
        p = PickHead(list);
        if (p->Info <= X->Info) AddTail(list1, p);
        else AddTail(list2, p);
    }
    ListQuickSort(list1); ListQuickSort(list2);
    LISTAppend(list, list1);
    AddTail(list, X);
    LISTAppend(list, list2);
}
```



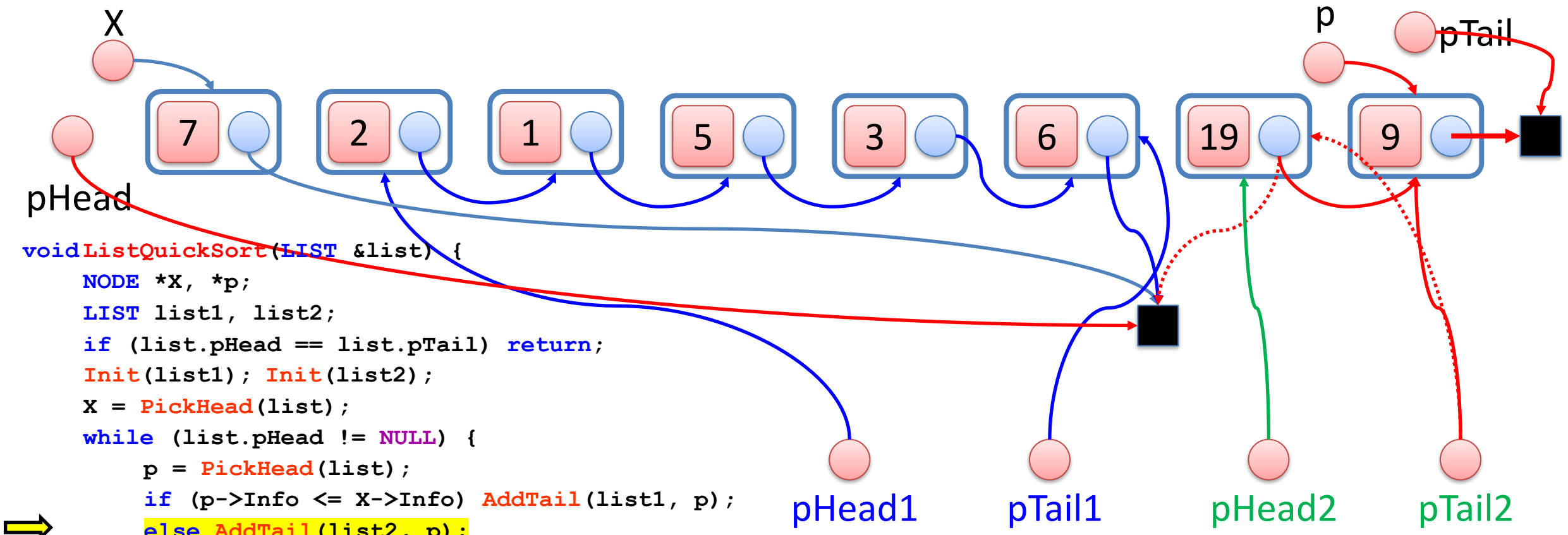

```
void ListQuickSort(LIST &list) {
    NODE *X, *p;
    LIST list1, list2;
    if (list.pHead == list.pTail) return;
    Init(list1); Init(list2);
    X = PickHead(list);
    while (list.pHead != NULL) {
        p = PickHead(list);
        if (p->Info <= X->Info) AddTail(list1, p);
        else AddTail(list2, p);
    }
    ListQuickSort(list1); ListQuickSort(list2);
    LISTAppend(list, list1);
    AddTail(list, X);
    LISTAppend(list, list2);
}
```



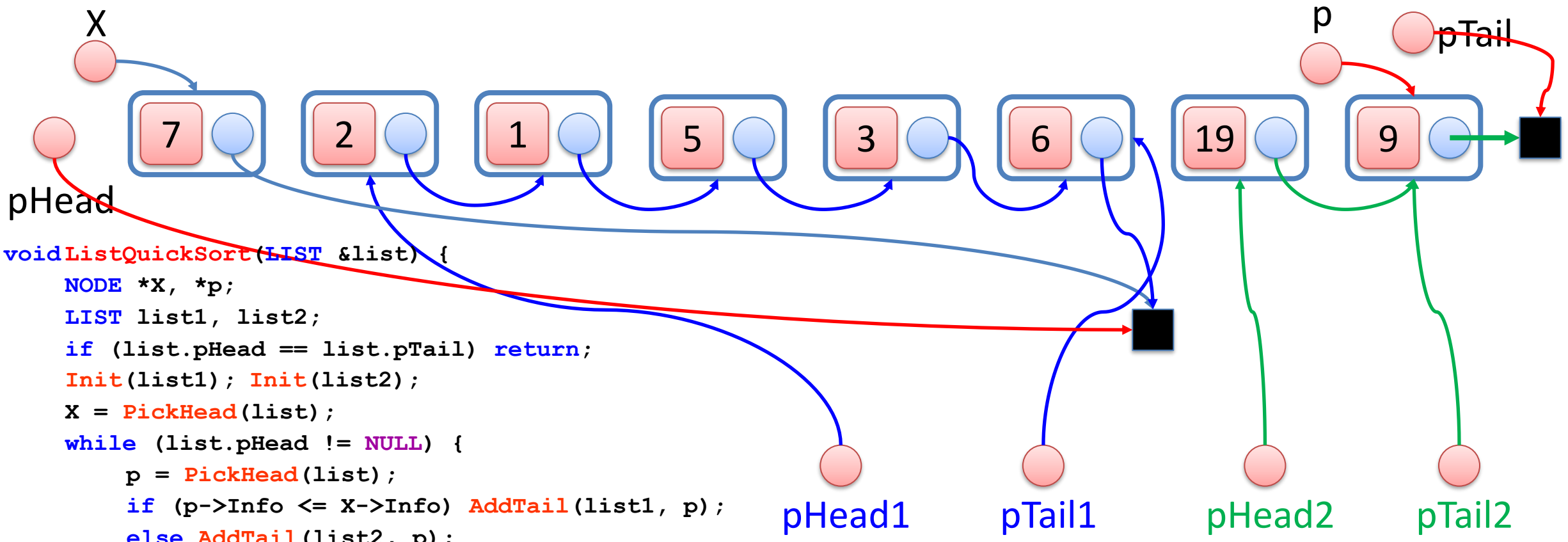

```
void ListQuickSort(LIST &list) {
    NODE *X, *p;
    LIST list1, list2;
    if (list.pHead == list.pTail) return;
    Init(list1); Init(list2);
    X = PickHead(list);
    while (list.pHead != NULL) {
        p = PickHead(list);
        if (p->Info <= X->Info) AddTail(list1, p);
        else AddTail(list2, p);
    }
    ListQuickSort(list1); ListQuickSort(list2);
    LISTAppend(list, list1);
    AddTail(list, X);
    LISTAppend(list, list2);
}
```




```
void ListQuickSort(LIST &list) {
    NODE *X, *p;
    LIST list1, list2;
    if (list.pHead == list.pTail) return;
    Init(list1); Init(list2);
    X = PickHead(list);
    while (list.pHead != NULL) {
        p = PickHead(list);
        if (p->Info <= X->Info) AddTail(list1, p);
        else AddTail(list2, p);
    }
    ListQuickSort(list1); ListQuickSort(list2);
    LISTAppend(list, list1);
    AddTail(list, X);
    LISTAppend(list, list2);
}
```




```
void ListQuickSort(LIST &list) {
    NODE *X, *p;
    LIST list1, list2;
    if (list.pHead == list.pTail) return;
    Init(list1); Init(list2);
    X = PickHead(list);
    while (list.pHead != NULL) {
        p = PickHead(list);
        if (p->Info <= X->Info) AddTail(list1, p);
        else AddTail(list2, p);
    }
    ListQuickSort(list1); ListQuickSort(list2);
    LISTAppend(list, list1);
    AddTail(list, X);
    LISTAppend(list, list2);
}
```

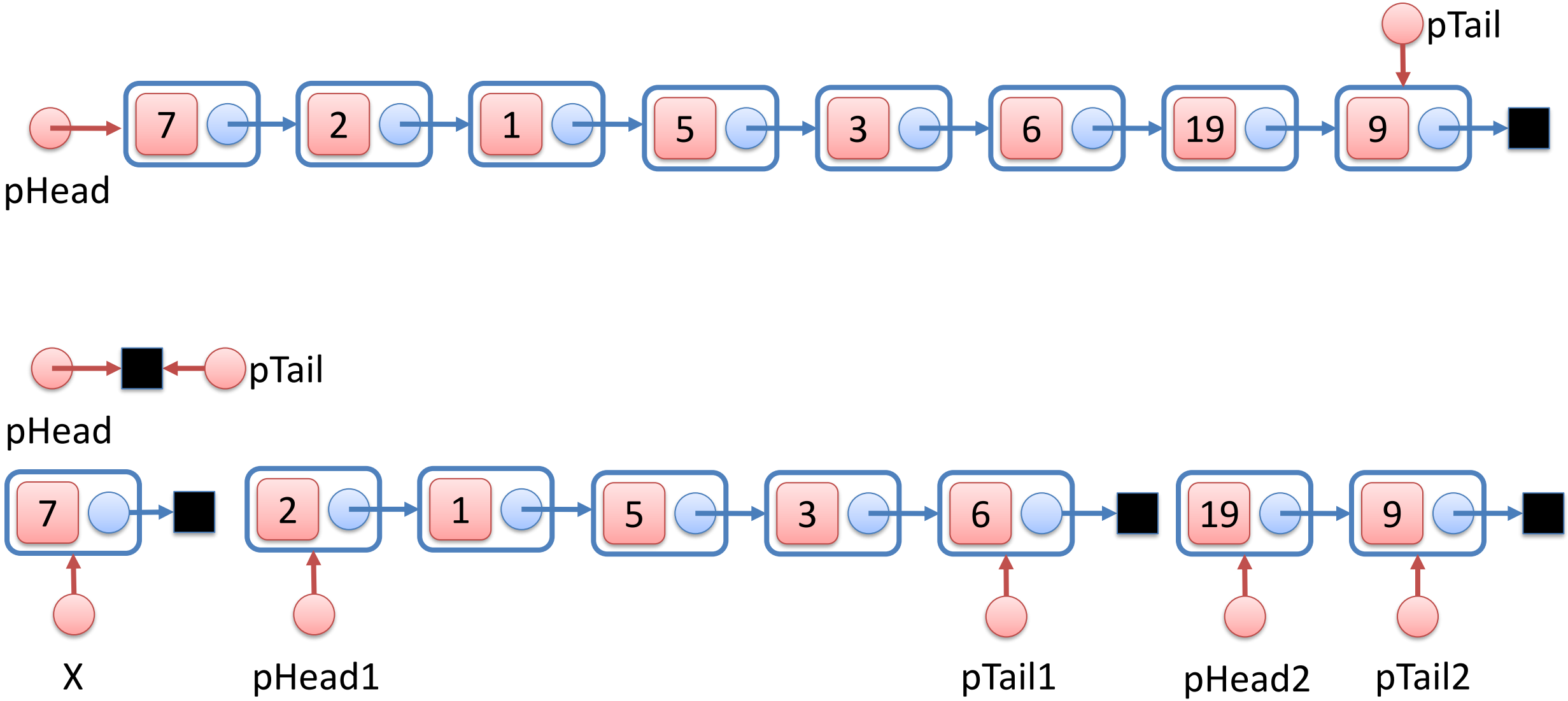



```
void ListQuickSort(LIST &list) {
    NODE *X, *p;
    LIST list1, list2;
    if (list.pHead == list.pTail) return;
    Init(list1); Init(list2);
    X = PickHead(list);
    while (list.pHead != NULL) {
        p = PickHead(list);
        if (p->Info <= X->Info) AddTail(list1, p);
        else AddTail(list2, p);
    }
    ListQuickSort(list1); ListQuickSort(list2);
    LISTAppend(list, list1);
    AddTail(list, X);
    LISTAppend(list, list2);
}
```

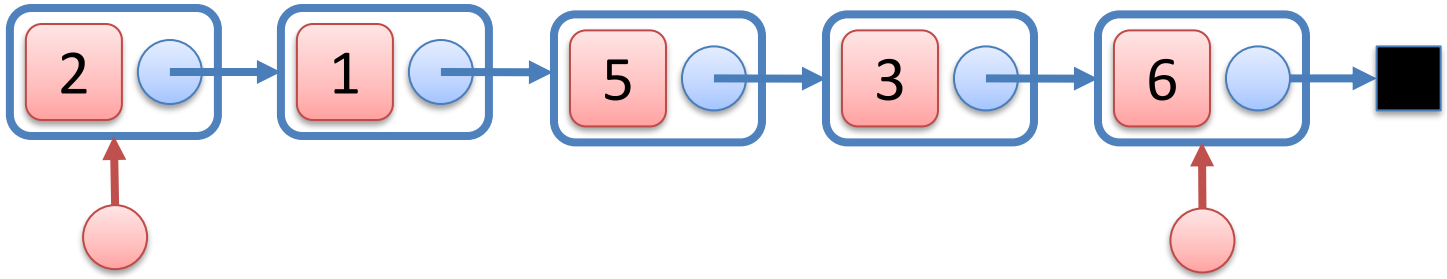
X: 7, pHead = pTail = NULL

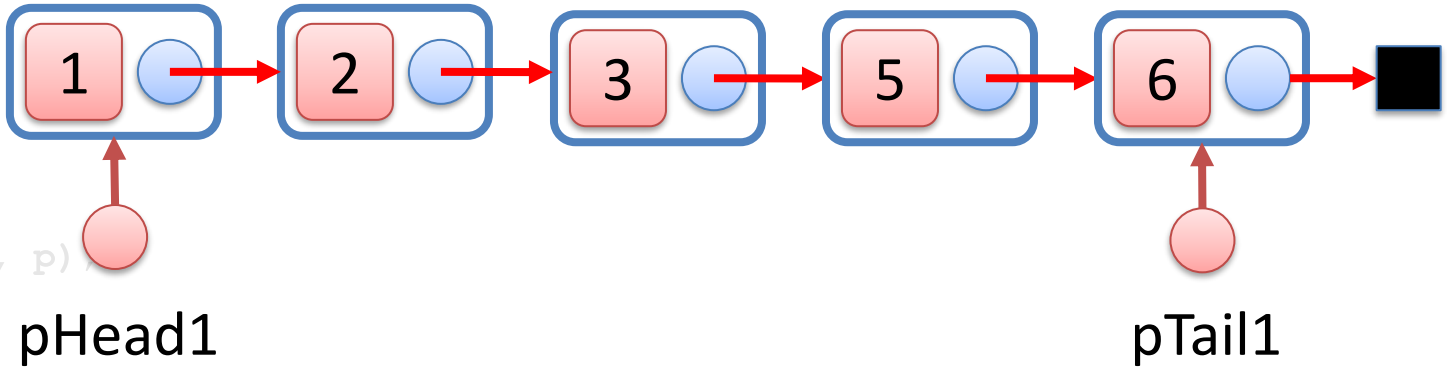
list1: pHead1 → 2 → 1 → 5 → 3 → 6 ← pTail1

list2: pHead2 → 19 → 9 ← pTail2


pTail
7
2
1
5
3
6
19
9
pHead
pTail
pHead
7
2
1
5
3
6
19
9
X
pHead1
pTail1
pHead2
pTail2

Recursive programming

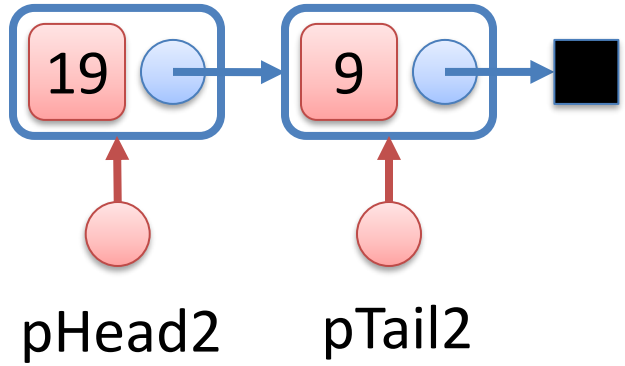


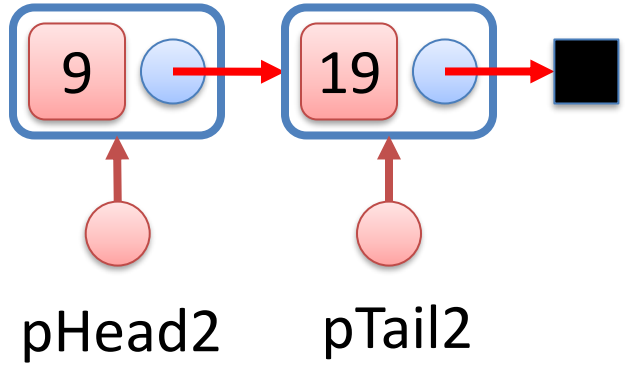



```
void ListQuickSort(LIST &list) {
    NODE *X, *p;
    LIST list1, list2;
    if (list.pHead == list.pTail) return;
    Init(list1); Init(list2);
    X = PickHead(list);
    while (list.pHead != NULL) {
        p = PickHead(list);
        if (p->Info <= X->Info) AddTail(list1, p);
        else AddTail(list2, p);
    }
    ListQuickSort(list1); ListQuickSort(list2);
    LISTAppend(list, list1);
    AddTail(list, X);
    LISTAppend(list, list2);
}
```

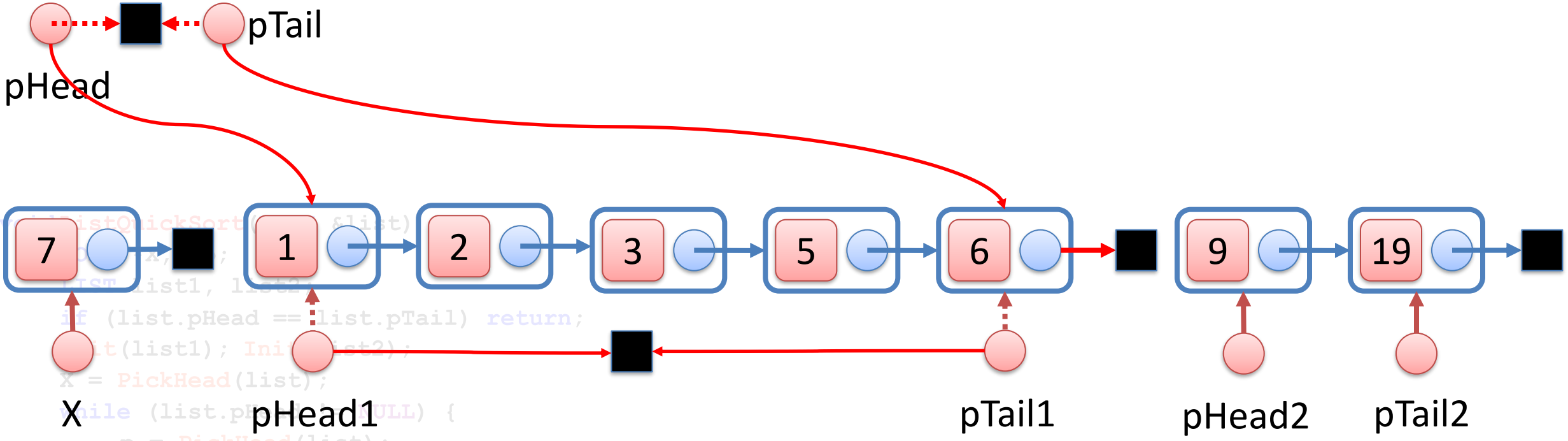
Recursive programming

```
void ListQuickSort(LIST &list) {
    NODE *X, *p;
    LIST list1, list2;
    if (list.pHead == list.pTail) return;
    Init(list1); Init(list2);
    X = PickHead(list);
    while (list.pHead != NULL) {
        p = PickHead(list);
        if (p->Info <= X->Info) AddTail(list1, p);
        else AddTail(list2, p);
    }
    ListQuickSort(list1); ListQuickSort(list2);
    LISTAppend(list, list1);
    AddTail(list, X);
    LISTAppend(list, list2);
}
```



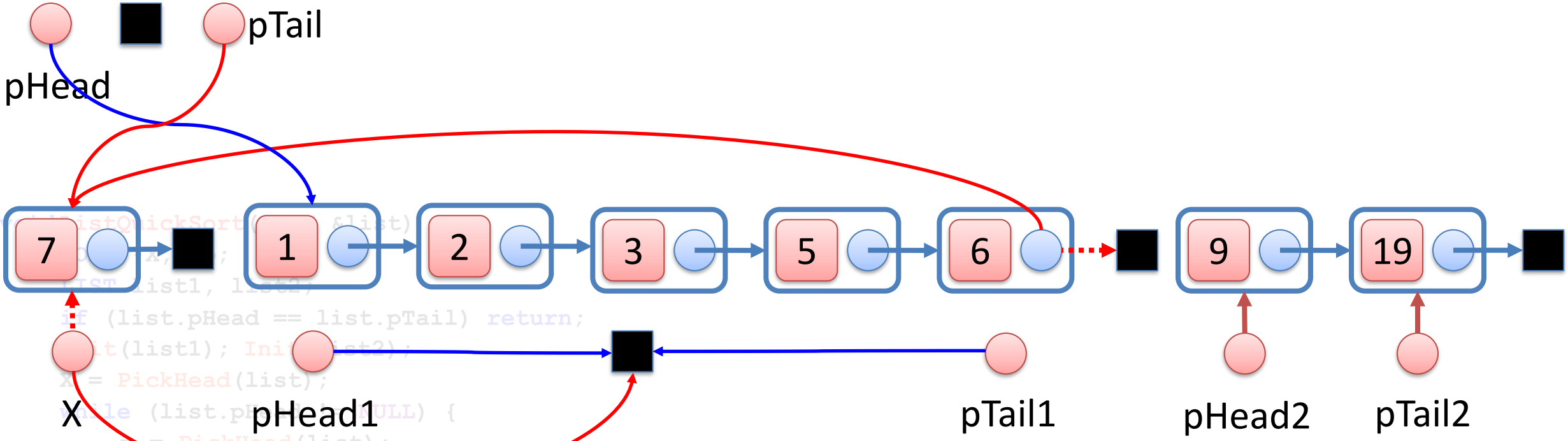


Recursive programming



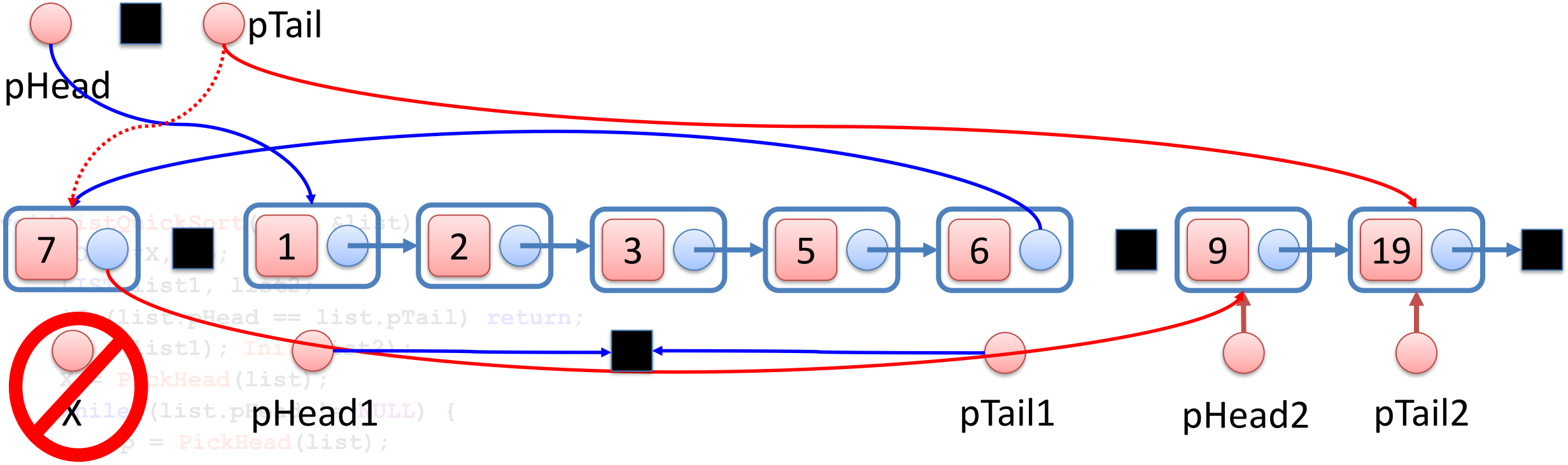

```
    ListQuickSort(list1);  ListQuickSort(list2);
    LISTAppend(list, list1);
    AddTail(list, X);
    LISTAppend(list, list2);
}
```

Recursive programming



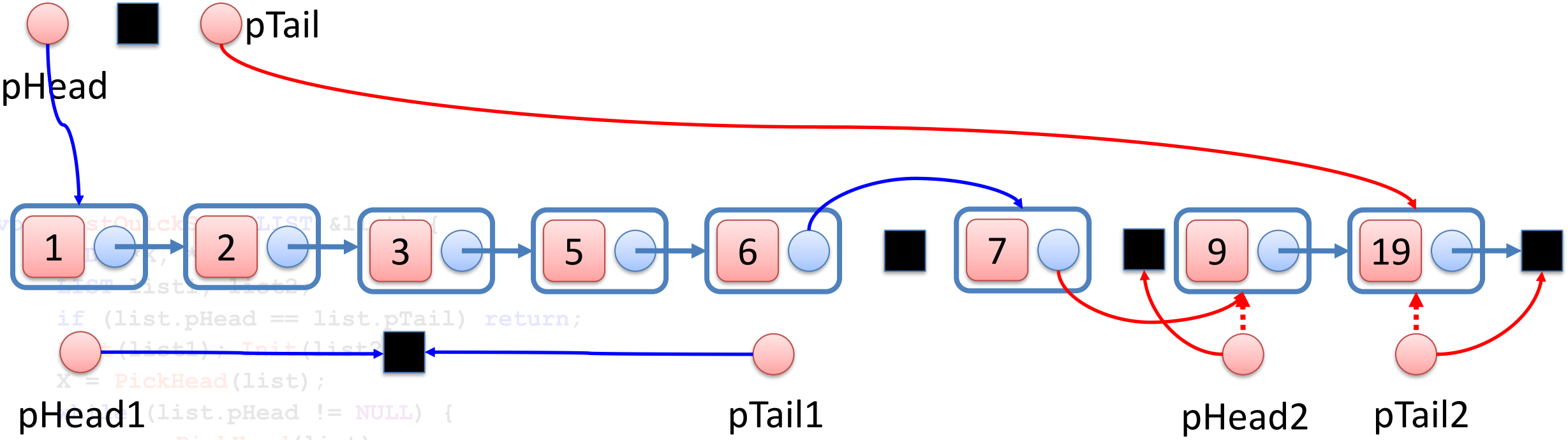

```
    ListQuickSort(list1);  ListQuickSort(list2);
    LISTAppend(list, list1);
⇨   AddTail(list, X);
    LISTAppend(list, list2);
}
```

Recursive programming




```
    ListQuickSort(list1);  ListQuickSort(list2);
    LISTAppend(list, list1);
    AddTail(list, X);
    LISTAppend(list, list2);
}
```

Recursive programming




```
    ListQuickSort(list1);  ListQuickSort(list2);
    LISTAppend(list, list1);
    AddTail(list, X);
⇨   LISTAppend(list, list2);
}
```

Q & A