<table>
<thead>
<tr>
<th><strong>Today</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>• Quiz 4</td>
<td></td>
</tr>
<tr>
<td>• Merge Sort</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Assignments</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>• HW-3 due fri</td>
<td></td>
</tr>
<tr>
<td>• Exercise 5</td>
<td></td>
</tr>
<tr>
<td>• Quiz 5 on Tuesday</td>
<td></td>
</tr>
<tr>
<td>• Exam 1 next Thurs</td>
<td></td>
</tr>
</tbody>
</table>
Merge Sort

The basic idea: A classic recursive “divide and conquer” approach

- divide list into two halves
- sort each half (recursive step)
- merge the sorted halves into sorted list
The merge sort algorithm (psuedocode):

Note: start, mid, and end are array indexes

MergeSort(T array[], int start, int end)
1. if start < end
2. mid = (start + end) / 2
3. mergesort(array, start, mid)  # recursive step
4. mergesort(array, mid + 1, end)  # recursive step
5. merge(array, start, mid, end)  # merge sorted sublists

Merge(T array[], int start, int mid, int end)
1. T tmp[(end - start) + 1]  # tmp array size n
2. first1 := start
3. first2 := mid + 1
4. i := 0
5. while first1 <= mid and first2 <= end do  # merge into tmp
6. if array[first1] < array[first2] then
7. tmp[i++] = array[first1++]
8. else
9. tmp[i++] = array[first2++]
10. while first1 <= mid do  # copy rest
11. tmp[i++] = array[first1++]
12. while first2 <= mid do  # copy rest
13. tmp[i++] = array[first2++]
14. for i = 0 to (end - start) do  # copy to array
15. array[start + i] = tmp[i]

For HW-4, combine both steps into one function
Merge Sort for HW-4:

- Both array (`ArraySeq`) and linked list (`LinkedSeq`) implementations
- The `ArraySeq` implementation via pseudocode above

For `LinkedSeq` ...

- We’ll do merge sort “in place” by splicing and reattaching nodes
- Linked lists are nice for this ...

Mergesort over linked lists:

We’ll use a helper function for the merge sort implementation:

```c
Node* merge_sort(Node* left, int len)
```

This function will do **both** the splitting and merging

- **left** is the left-most node in the list to consider
- **len** is the number of nodes to consider

The general (high-level) algorithm

```c
Node* merge_sort(Node* left, int len)
1. if len <= 1 return left
2. compute mid length and set up right pointer
3. left := merge_sort(left, mid)
4. right := merge_sort(right, len - mid)
5. merge the two lists left and right
6. return pointer to first node of merged list
```

```c
void merge_sort()
1. head := merge_sort(head, node_count)
2. update tail ptr (by looping to last node)
```
Quick Sort

The basic idea (also a recursive “divide and conquer” approach)

• pick a “pivot” element (e.g., first element in list)
• put values smaller than pivot on left
• put values larger than pivot on right
• put pivot value in the middle
• sort the left and right halves (using quick sort)
The quick sort algorithm (psuedocode):

```c
quick_sort(T array[], int start, int end) # start, end indexes
1. if start < end then
2. pivot_val := array[start] # assumes pivot is first elem
3. end_p1 := start # end index, start part
4. for i = start + 1 to end do # partition
5. if array[i] < pivot_val then
6.   end_p1 := end_p1 + 1
7.   swap(array[i], array[end_p1])
8. swap(array[start], array[end_p1]) # move pivot
9. quick_sort(array, start, end_p1 - 1) # recursive step
10. quick_sort(array, end_p1 + 1, end) # recursive step
```

Quick sort for HW-4

- Both array (via `ArraySeq`) and linked list (via `LinkedSeq`) implementation
- Two variants: first element as pivot and random pivot
- For random pivot: setup step to choose random index, swap value with first

For `LinkedSeq` ...

- We’ll again do sort “in place” by splicing and reattaching

Quick Sort over Linked Lists

- Again, implement quick sort by splicing and reattaching nodes

We’ll use a helper function:

```c
Node* quick_sort(Node* start, int len)
```

Function does both partitioning and recursive step
General High-Level Algorithm

Node* quick_sort(Node* start, int len)
1. if len <= 1 then return start
3. separate first node from rest of list (pivot node)
4. move nodes into a smaller and larger list (partition)
5. smaller := quick_sort(smaller, smaller_len)
6. larger := quick_sort(larger, larger_len)
7. attach smaller, pivot, and larger together
8. return head node of resulting list

void quick_sort()
1. head := quick_sort(head, node_count)
2. update tail pointer (by looping to last node)

For random pivot, modify step 3:

- select an index at random between 0 and len - 1
- separate node at random index on each pass