The Merge Sort

Textbook Authors:
Ken Lambert & Doug Nance
PowerPoint Lecture
by Dave Clausen

Merge Sort Description

- The essential idea behind merge sort is to make repeated use of a function that merges two lists, each already in ascending order, into a third list, also arranged in ascending order.
- The merge function itself only requires sequential access to the lists.
- Its logic is similar to the method you would use if you were merging two sorted piles of index cards into a third pile.
  - That is, start with the first card from each pile.
  - Compare them to see which one comes first, transfer that one over to the third pile, and advance to the next card in that pile.
  - Repeat that comparison, transfer, and advance operations until one of the piles runs out of cards.
  - At that point, move what is left of the remaining pile over the third merged pile.

Merge Sort Recursive Outline

- Merge sort employs a divide-and-conquer strategy to break the $O(n^2)$ barrier.
- Here is an outline of the algorithm:
  - Compute the middle position of a vector and recursively sort its left and right subvectors (divide and conquer).
  - Merge the two sorted subvectors back into a single sorted vector.
  - Stop the process when subvectors can no longer be subdivided.
  - This top-level design strategy can be implemented as three C++ functions:
  - **`Merge_Sort:`** the function called by clients (int main or control menu execution).
  - **`MergeSortHelper:`** a helper function that hides the extra parameter required by recursive calls.
  - **`Merge:`** a helper function that implements the merging process.

Merge Sort Parameters

- The merging process uses an extra vector, which we call `copyBuffer` (you can name this list_copy2).
- To avoid the overhead of allocating and deallocating the `copyBuffer` each time `merge` is called, the buffer is allocated once in `mergeSort` and subsequently passed to `mergeSortHelper` and `merge`.
- Each time `mergeSortHelper` is called, it needs to know the bounds of the subvector with which it is working. These bounds are provided by two parameters, `low` and `high`.

The Merge Sort Code C ++

- **Here is the code for Merge_Sort:**

```
void Merge_Sort(vector<int> &v)
{
   vector<int> copyBuffer(v.size());
   MergeSortHelper(v,copyBuffer,0,v.size() - 1);
}
```

- **Remember to use logical_size instead of v.size(), and logical_size-1 instead of v.size()-1.**
- **You will also have to pass logical_size to Merge_Sort.**
- **After checking that it has been passed a subvector of at least two items, mergeSortHelper computes the midpoint of the subvector, recursively sorts the portions below and above the midpoint, and calls the merge function to merge the results.**

MergeSortHelper Code C++

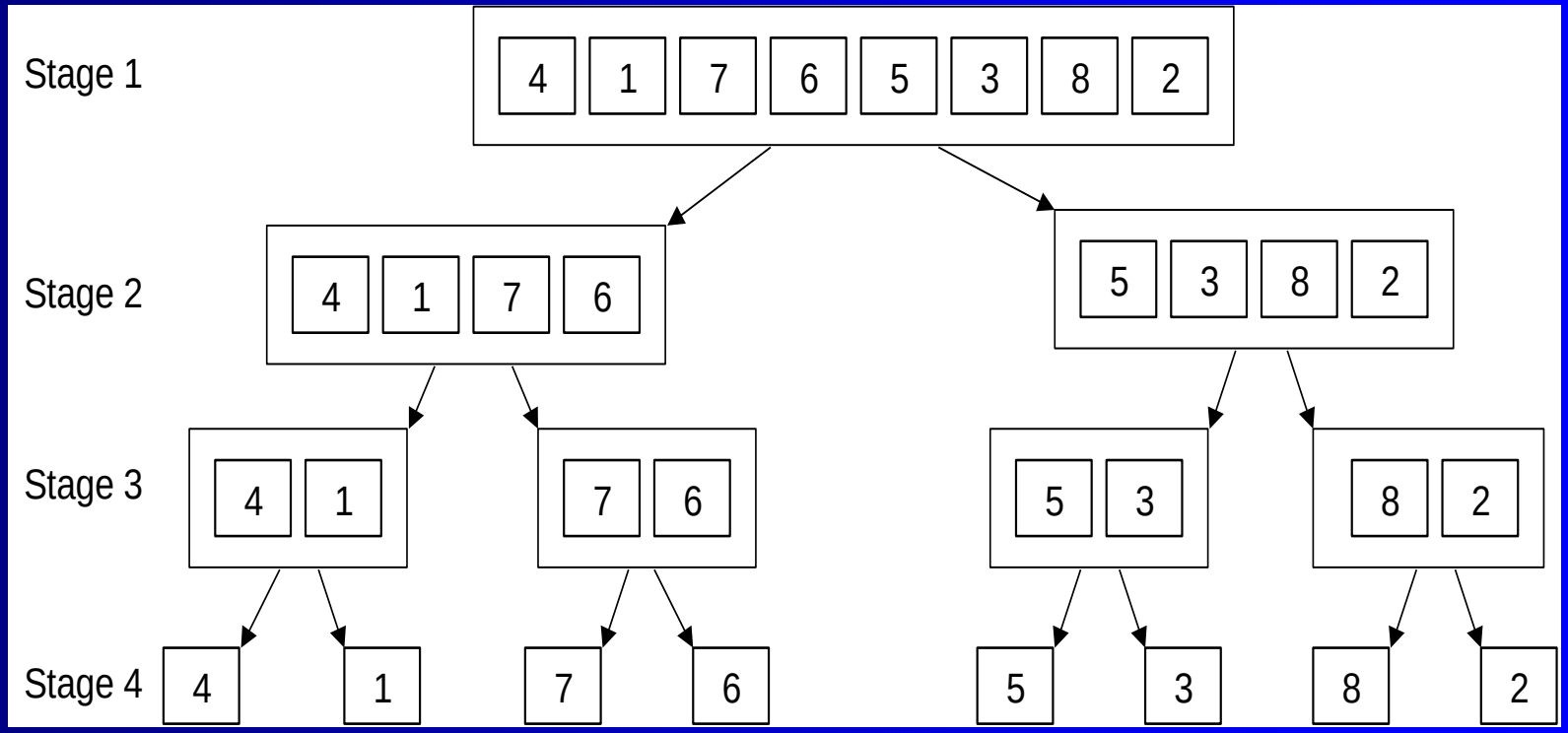
```
void MergeSortHelper(vector<int> &v, vector<int> &copyBuffer,
                                        int low, int high)
{
   // v is the vector being sorted
   // copyBuffer  is temp space needed during merge
   // low, high are bounds of subvector
   // middle is the midpoint of subvector
    if (low < high)
{
      int middle = (low + high) / 2;
     MergeSortHelper(v, copyBuffer, low, middle);
     MergeSortHelper(v, copyBuffer, middle + 1, high);
      Merge(v, copyBuffer, low, middle, high);
   }
}
```

Trace through Merge Sort

- The figure on the next slide shows the subvectors generated during recursive calls to `mergeSortHelper`, starting from a vector of eight items.
- Note that in this example the subvectors are evenly subdivided at each stage and there are 2^{k-1} subvectors to be merged at stage k.
- Had the length of the initial vector not been a power of two, then an exactly even subdivision would not have been achieved at each stage and the last stage would not have contained a full complement of subvectors.

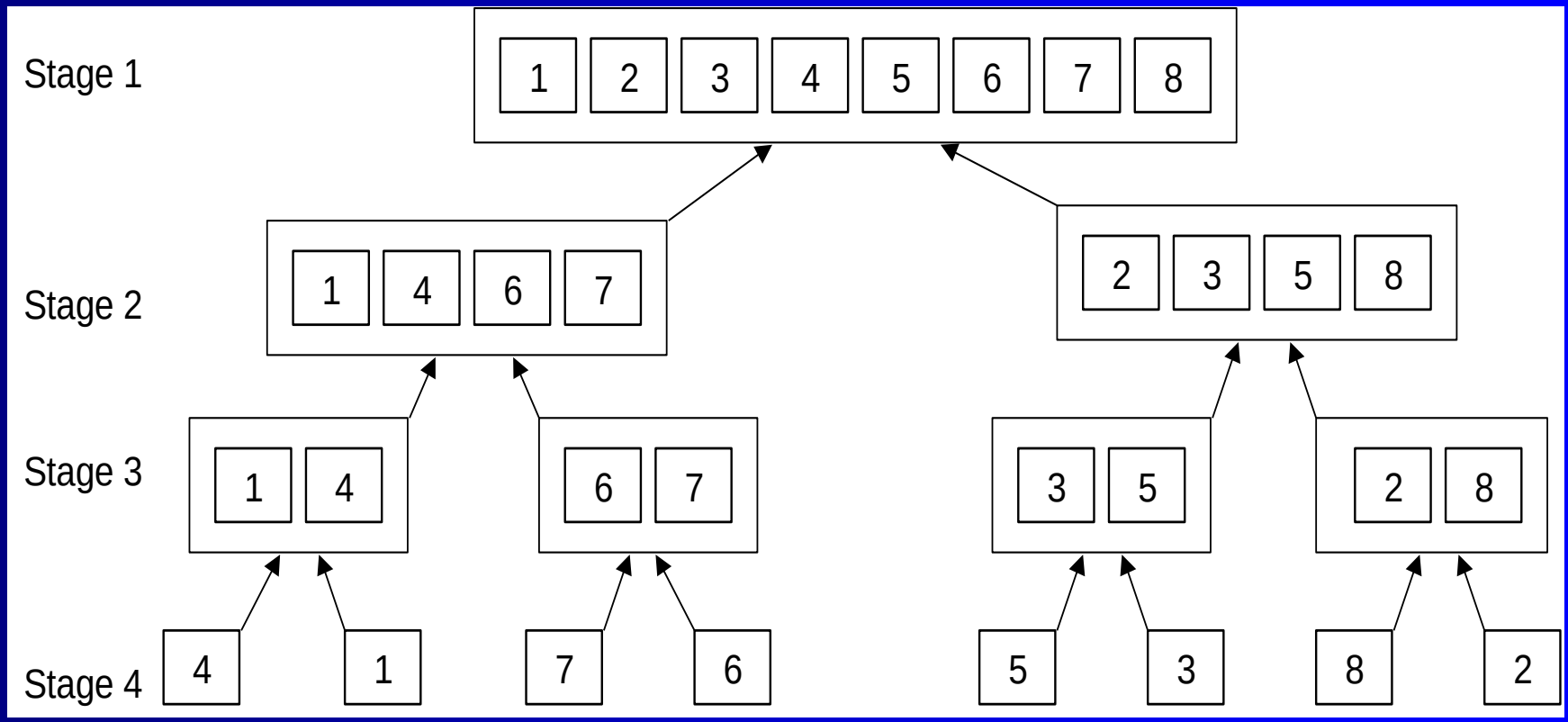
Merge Sort Walk Through

- The figure on this slide shows the subvectors generated during recursive calls to `mergeSortHelper`, starting from a vector of eight items.



Walk Through 2

- Merging the sub vectors generated during a merge sort in order as they are merged.



Merge Code C++

```
void Merge(vector<int> &v, vector<int> &copyBuffer, int low, int middle, int high)
{
   int i1 = low, i2 = middle + 1, i;
   for (i = low; i <= high; i++)
   {
      if (i1 > middle)
         copyBuffer[i] = v[i2++];        //First subvector exhausted
      else if (i2 > high)
         copyBuffer[i] = v[i1++];        //Second subvector exhausted
      else if (v[i1] < v[i2])
         copyBuffer[i] = v[i1++];        //Item in first subvector is less
      else
         copyBuffer[i] = v[i2++];        //Item in second subvector is less
   }
    for (i = low; i <= high; i++)            //Copy sorted items back into
      v[i] = copyBuffer[i];                    //proper position in a
}
```

Merge Description

- The `merge` function combines two sorted subvectors into a larger sorted subvector. The first subvector lies between `low` and `middle` and the second between `middle + 1` and `high`. The process consists of three steps:
  - Set up index pointers to the first items in each subvector. These are at positions `low` and `middle+1`.
  - Starting with the first item in each subvector, repeatedly compare items. Copy the smaller item from its subvector to the copy buffer and advance to the next item in the subvector. Repeat until all items have been copied from both subvectors. If the end of one subvector is reached before the other's, finish by copying the remaining items from the other subvector.
  - Copy the portion of `copyBuffer` between `low` and `high` back to the corresponding positions in the vector v.

Big - O Notation

Big - O notation is used to describe the efficiency of a search or sort. The actual time necessary to complete the sort varies according to the speed of your system. Big - O notation is an approximate mathematical formula to determine how many operations are necessary to perform the search or sort. The Big - O notation for the Merge Sort is O(**$nlog_2n$**), because it takes approximately **$nlog_2n$** passes to find the target element.