Course Agenda

- Module 1 – Personal Software Process ✓
- Module 2 – Data Driven Programming
  - Data Abstraction ✓
  - Linear Collections – Lists and Sets ✓
  - Tuple Data ✓
  - **Searching and Sorting**

Ordered Search

- Recall Library list - Implemented as ordered list.
- Search is efficient i.e. O(logN) time using ordered list.
- compare function is used to abstract key information

int compare(Member a, Member b)

{ if (a.y == b.y) { if (a.i < b.i) return …}
… }

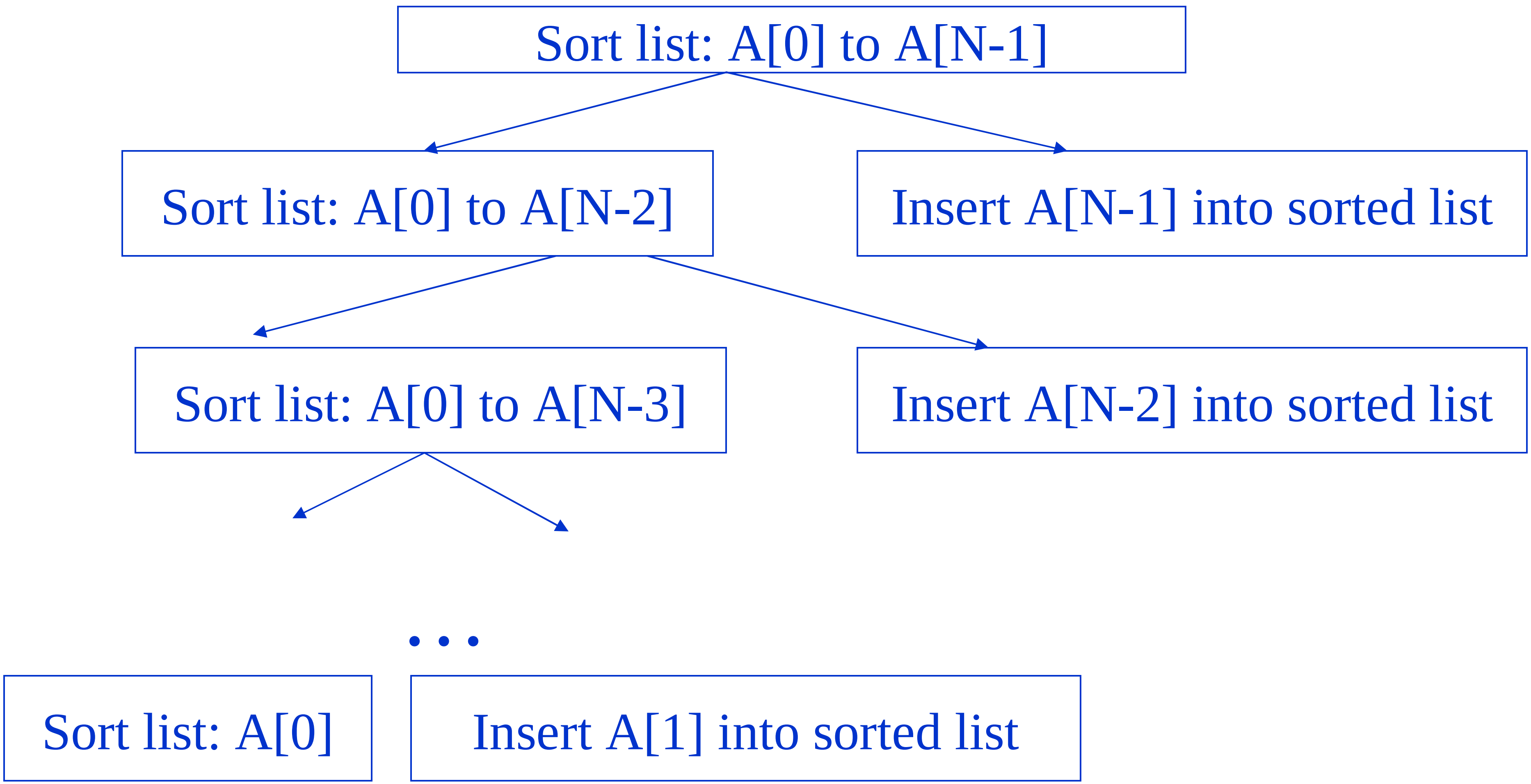
Ordered Search

- How do you get ordered lists?
- E.g. Librarian wants list ordered by Year and then ID today.
  - And tomorrow the requirement changes: he/she wants it ordered by Group and then Name.
- We need an algorithm to be designed
  - Ordering also known as Sorting!

Sorting

- Big idea:
  - Inserting an element into a sorted list in the appropriate position retains the order.
  - So what?
    - Start with a singleton list – sorted trivially.
    - Repeatedly insert elements – one at a time – while keeping it sorted.
  - Leads to sorting technique known as “Insertion Sort”

Insertion Sort – Top Down Design



Insertion Sort – Top Down Design

- Termination:
  - Sort list A[0] : Nothing to be done!(Atomic Soln.)
- Problem decomposed into 2 modules:
  - function for insertion:
    - void insert(Member m, Member ms[], unsigned int size)
  - function for repeated insertion:
    - void insertSort(Member ms[], unsinged int size)

Algorithm for Insertion

```
void insert(Member m, Member ms[], unsigned int size)
{
    j = 0;
        //Loop Invariant: 0 <= i < j  implies ms[i] <= m
    for each j from 0 to size-1
      compare ms[j] with m;
        if (compare(ms[j],m)==GREATER) break;

      //Loop Invariant: ms[k+1] is empty

    for each k from size-1 to j
      ms[k+1] = ms[k];
     ms[j] = m;
}
```

Algorithm for Insertion Sort

```
void insertionSort(Member ms[], unsigned int size)
{
    //Loop Invariant: ms[0] to ms[j-1] is ordered
                j = 1;
                while  j <= size-1 {
                  insert(ms[j], ms, j)
                     j = j + 1;
                   }

// Correctness Assertion: ms[0] to ms[size-1] is ordered.
}
```

Insertion Sort

- Complexity:
  - Insertion of a single element: $O(j)$ where j is size of list.
  - Insertion Sorting: Insert single element into lists from size 1 to size-1:

$$1 + 2 + \ldots + (\text{size} - 1) = O(\text{size}^2)$$

Implementation of Insertion Sort

/* file insort.h/

// Interface for insertionSort function only

/* file insort.c */

// Implementations for insertionSort and insert

/* file compare.h */

//Interface for compare function

/* file compare.c */

// Implementation for compare function

Insertion Sort - Implementation

```
/* insort.h */

void insertSort(Member ms[], int size);
```

Insertion Sort - Implementation

```
/* insort.c */
void insert(Member m, Member ms[], int size)
{
    int j,k;
    for (j=0; j<size; j++)   // Find the right position
        if (compare(m, ms[j]==GREATER) break;
    // Assertion: j is the right position for m
         for (k=size-1; k>=j; k--)  // Right shift values>m
        ms[k+1] = ms[k];
    ms[j] = m;  // Insert m
}
```

Insertion Sort - Implementation

```
/* insort.c */
void insertSort(Member ms[], int size) {
    int j;
    // Loop Invariant:
    //  The sublist from ms[0] to ms[j-1] is sorted
    for (j=1; j<size; j++) {
        insert(ms[j], ms, j);
    }
}
```

Implementation of Insertion Sort

- After creating insort.c
  - gcc –c insort.c
- After creating each compareX.c (with same compare function)
  - gcc –c compareX.c
- After creating a main in testSort.c
  - gcc testSort.c insort.o compareX.o –o sortX

// for each compareX.o separately

Exercises

- Provide two different implementations for compare.
- Identify test cases for compare, insert, and insertionSort.
- Implement insert
- Implement insertionSort
- Link with different compare implementations
- Execute and test!

Course Agenda

- Module 1 – Personal Software Process ✓
- Module 2 – Data Driven Programming
  - – Data Abstraction ✓
  - – Linear Collections – Lists and Sets ✓
  - – Tuple Data ✓
  - – Searching and Sorting
    - Insertion Sorting ✓
    - Additional Sorting Techniques

Insertion Sorting

- Concept:
  - Inserting an element into a sorted list in the appropriate position retains the (sorted) order.
  - Leads to a sorting algorithm that uses repeated insertion in-place.
- Complexity:
  - $O(N^2)$ time – worst case and average case.
  - $O(1)$ space – worst case and average case.

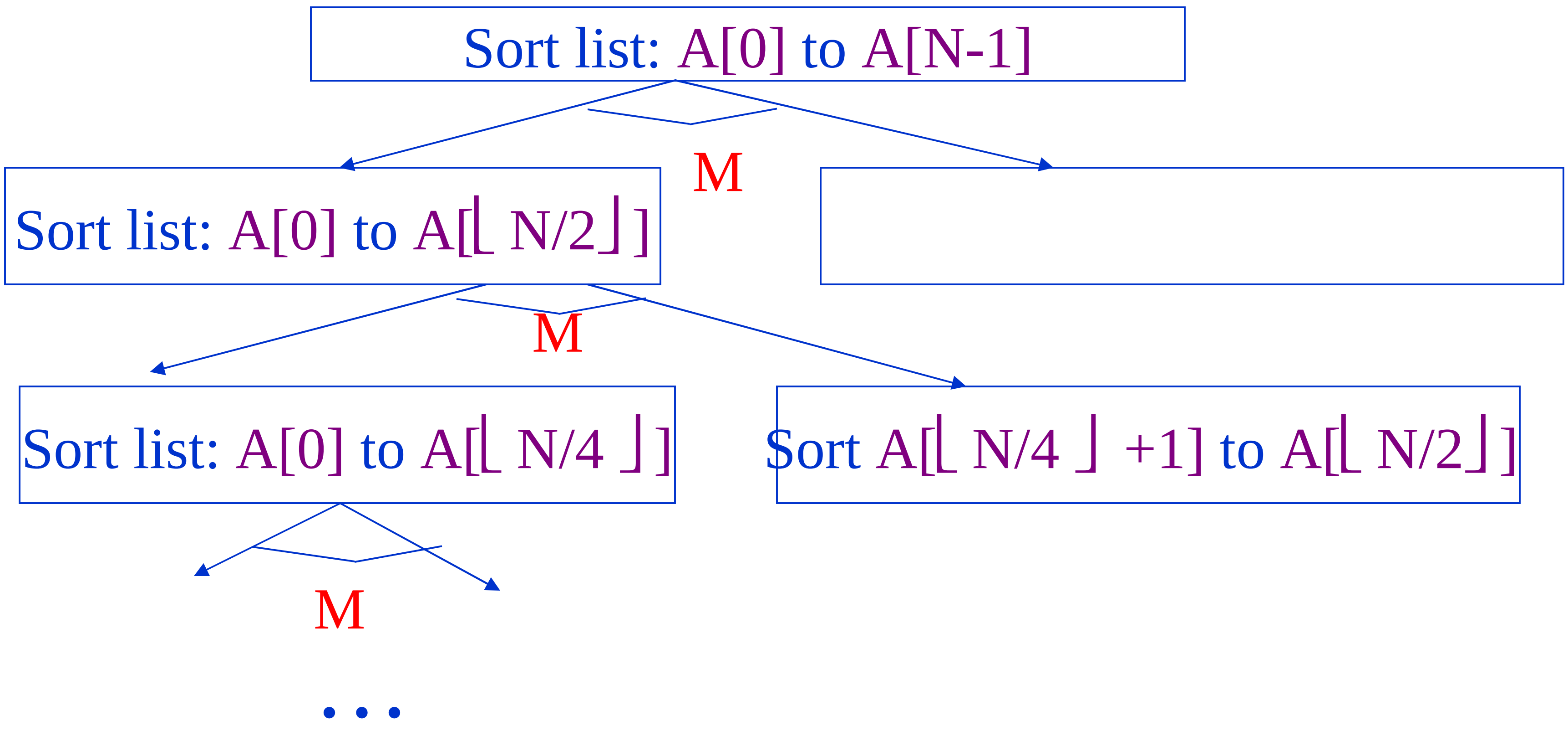
Sorting (in general)

- Insertion Sorting
  - Reduces sorting list of size N to sorting list of size N-1.
  - Relies on : “insertion” as order-preserving operation.
- Are there other
  - reductions and
  - associated order-preserving operations?

Merging

- Recall “merge” operation:
  - Merge two ordered lists such that they result in a single ordered list (containing all elements from either list).
  - Can we find a reduction / division to exploit this operation?

Sorting – Top Down Design



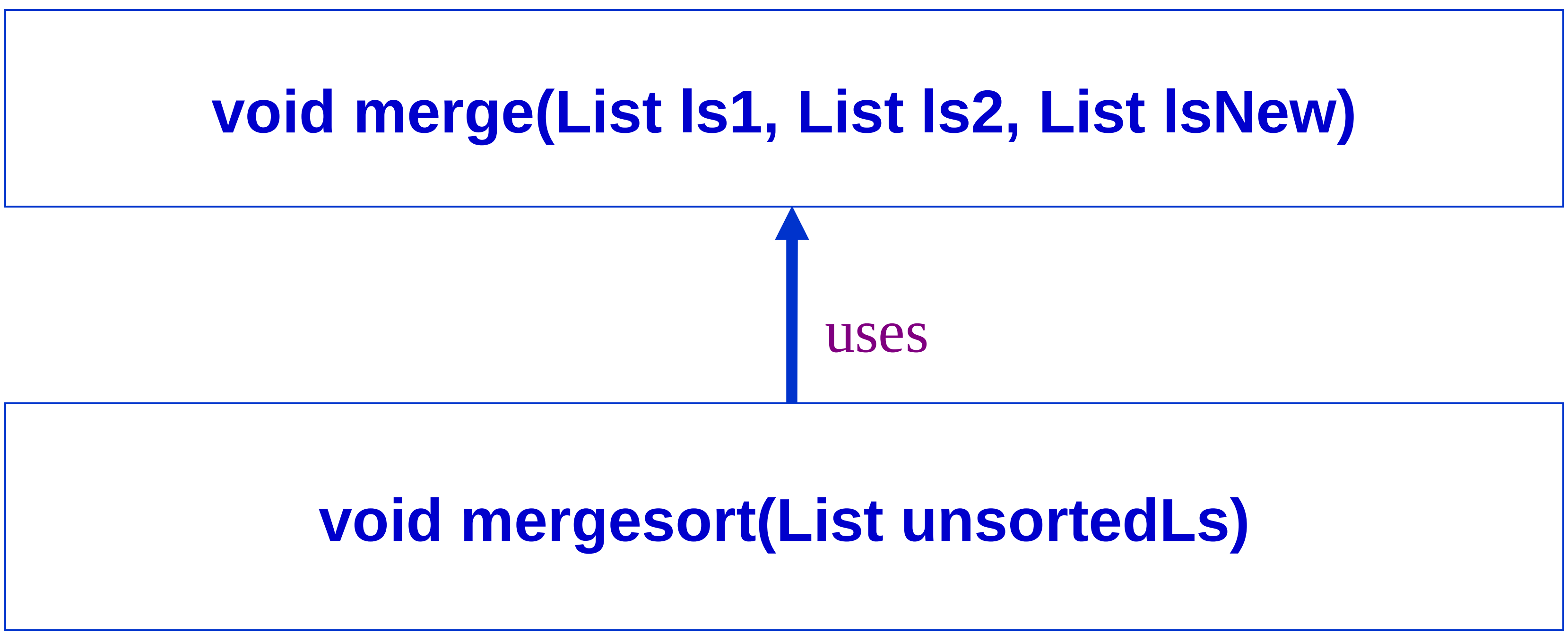
Merge Sorting – Top Down Design

- How do we combine results?
  - Order-preserving operation needed!
  - Requirement: Combine two sorted lists to one sorted list of all elements from either list.
  - Merge fits the bill.
- When do we terminate (the division)?
  - Singleton lists are sorted (trivially).

Merge Sorting

- Repeated merging of 2 sorted lists at a time to form a combined sorted list of elements.
- Can we merge in-place?
  - Not if we are using fixed position lists (i.e. arrays).
- How much extra space needed?
  - Merging two lists of size m and n requires m+n new space.

Design Modules



Algorithm Merge

```
while (ls1 and ls2 are not empty) {

      get next elements e1 and e2;
      if (e1 <= e2) move e1 to newlist;
      else move e2 to newlist;

}

  if (either List is still not empty) {
          move all its elements to newlist;
  }
```

Algorithm MergeSort

for each j from 0 to $\lceil N/2 \rceil - 1$

merge(ulist[2j to 2j], ulist[2j+1 to 2j+1])

for each j from 0 to $\lceil N/4 \rceil - 1$

merge(ulist[4j to 4j+1], ulist[4j+2 to 4j+3]

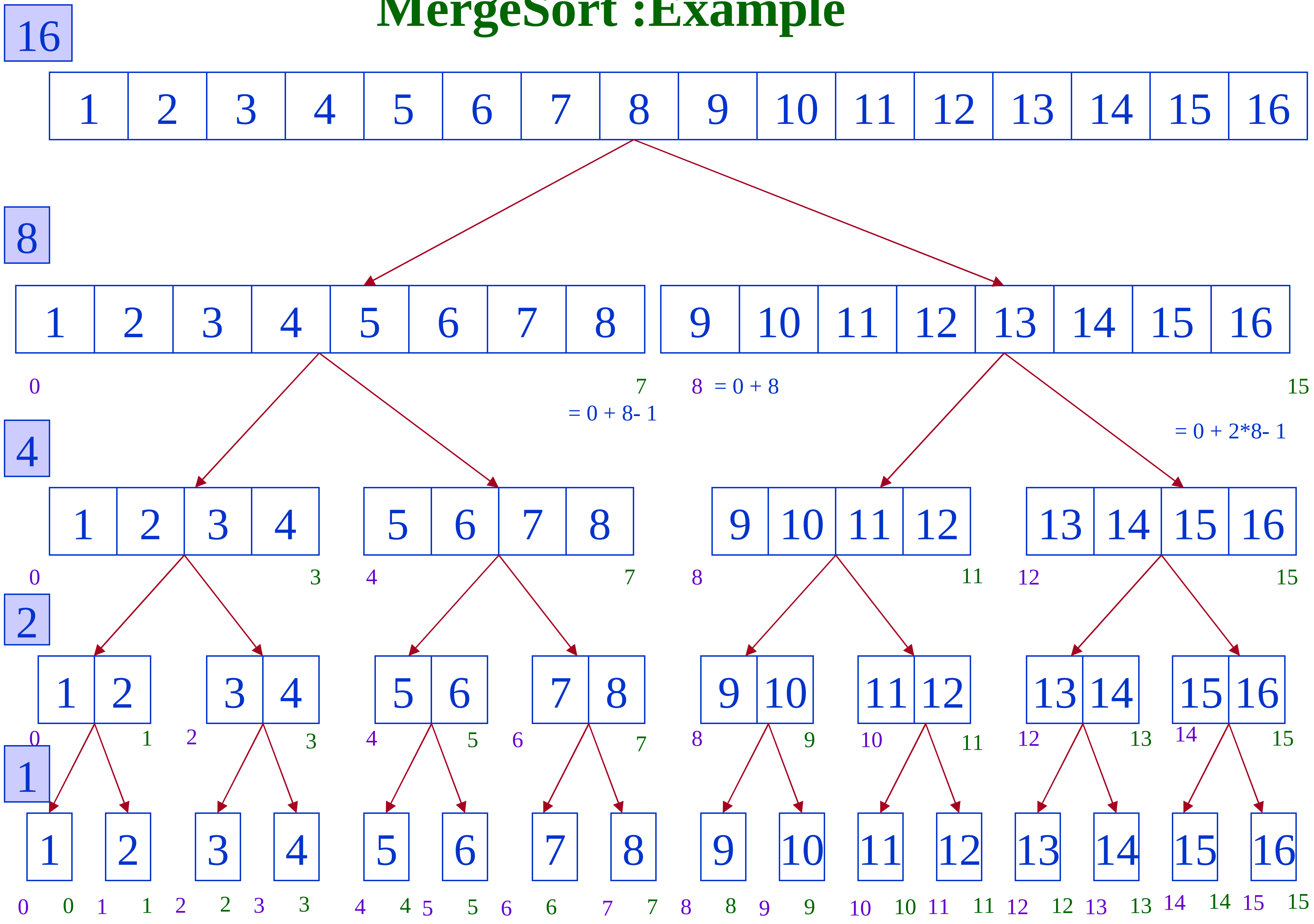
…

// until only 1 pair of lists are left.

Add slide here for example

- Give a list of numbers and show how we sort in mergesort operation
- Either use black board or use additional slide here.
- Students usually face problem understanding how the top-down design we say is converted to code here.

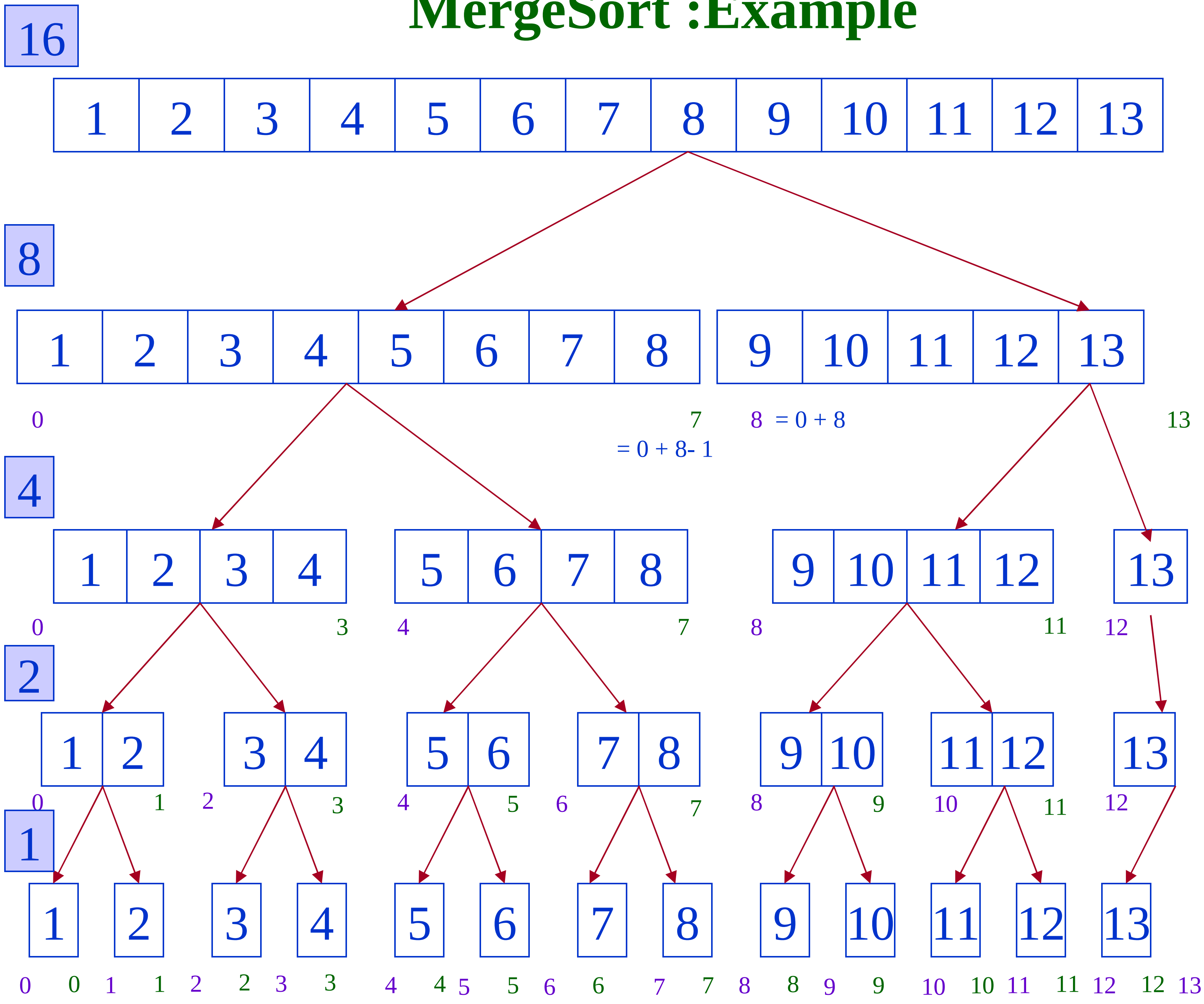
MergeSort :Example



Algorithm MergeSort : ($N = 2^P$)

```
k=1;
while k <= ⌈N/2⌉   {
   for each j from 0 to ⌈N/(2*k – 1)⌉ {
     start = 2*k*j;
    merge(list[start to start+k-1],
                 list[start+k to start+2k-1],
                  newlist)
   }
    copy newlist back into list
    k=2*k;  // double the list size
}
```

MergeSort :Example



Algorithm MergeSort - Analysis

- List size is doubled each iteration.
  - Outer loop: k = 1, 2, 4, … N leads to O(logN) iterations
  - If N is not a power of 2, some of the lists may be empty.
  - Problem to fix: Use N as the limit and rewrite inner loop.
  - Contract: Merge accepts lists of different size including empty lists!

Algorithm MergeSort – Inner Loop

```
if (start <= N) {
    if (start + k – 1  >  N) {
      e1 = (N – 1);
     copy(ulist[start, e1], newList[start, e1]);
  } else {
    e2 = (start + 2k – 1  >  N) ? (N – 1) : (start + 2k – 1);
     merge(ulist[start   to   e1],
                ulist[start +k   to   e2],
          newList[start   to   start+e2])
  }
  }
```

Algortithm MergeSort - Complexity

- One series of merge operations for the list take O(N)
  - N/2 pairs of 1 element lists
  - N/4 pairs of 2 element lists
- Outer loop executed O(logN) times
- Total Time Complexity O(N*logN)
- Total Space Complexity O(N)
- Compare with InsertionSort:
  - $O(N^2)$ time complexity and O(1) space complexity.

MergeSort vs. InsertionSort

- Time vs. Space Tradeoff
- Online vs Offline:
  - Insertion does not need all elements at one to start sorting (Online Sorting)
- Off-memory Sorting:
  - Merge operation does not use random access – it selects elements one-by-one in left-to-right order.
  - Sequential Access Lists:
    - Files (Abstract)
    - Disks, Tape (Concrete)

Questions / Exercises:

- Exercise: Code the algorithm in C.
- Question:
  - How do you pass ulist[start, start+k-1] ?
  - Use the same array ulist, and separate start and end indices (for now.)
- Exercise: Identify Test Cases for merge and mergesort operations.

Back to Sorting Questions

- Is NlogN the best time we can do for sorting?
- Consider Drama Club list scenario:
  – (Year, ID) is known to be the key I.e. a pair (y1, i1) is unique among all the records of the list.
  – Can we directly drop the element (y1, i1) in the position corresponding to the key?

Back to Sorting Questions [2]

- More generic question: Can sorting be done without comparison?
  - Yes, if we know the set (or range of keys) ahead of time!
  - Consider drama club list example: at most 6 different years and at most 1000 ID values each year.
  - Use 1 bucket for each year. Use smaller bucket for each ID within each year!

Bucket Sorting

// newList[ys][ids]

// ys – number of years

// ids – number of ids per year

```
for j from 0 to N-1 {
 e = ulist[j];
newList[e.y][e.ID] = e;
}
```

// Assumption: e.y ranges from 0 to ys-1

// Use e.y – firstyear as index otherwise

// e.g if e.y ranges from 1998 to 2003 use e.y-1998

Bucket Sorting - Complexity

- Time Complexity – O(N)
- Space Complexity – O(N)
- Applies only when range of keys is known ahead of time and can be mapped to a fixed range of integers.

Sorting Comparisons

- Make a comparison chart :

Complexity (Time, Space),

Online vs Offline,

In-memory vs Off-memory,

Known key range vs Unknown key range.

Back to Sorting Questions

- When do we have to sort at all?
  - Sorting takes $O(N)$ to $O(N^2)$ time and upto $O(N)$ space.
  - Result: Better searching – Time reduced from $O(N)$ to $O(\log N)$
  - Useful only if more searches than additions (which will need sorting)!
    - If N searches are done per insertion then insertion sorting is not worthwhile!