Chapter 8

SORTING

1. Introduction and Notation
2. Insertion Sort
3. Selection Sort
4. Shell Sort
5. Lower Bounds
6. Divide-and-Conquer Sorting
7. Mergesort for Linked Lists
8. Quicksort for Contiguous Lists
9. Heaps and Heapsort
10. Review: Comparison of Methods

Requirements

- In an ***external*** sort, there are so many records to be sorted that they must be kept in external files on disks, tapes, or the like. In an ***internal*** sort the records can all be kept internally in high-speed memory. We consider only internal sorting.

- We use the notation and classes set up in Chapters 6 and 7. Thus we shall sort lists of records into the order determined by keys associated with the records. The declarations for a list and the names assigned to various types and operations will be the same as in previous chapters.

- If two or more of the entries in a list have the same key, we must exercise special care.

- To analyze sorting algorithms, we consider both the number of comparisons of keys and the number of times entries must be moved inside a list, particularly in a contiguous list.

- Both the worst-case and the average performance of a sorting algorithm are of interest. To find the average, we shall consider what would happen if the algorithm were run on all possible orderings of the list (with n entries, there are $n!$ such orderings altogether) and take the average of the results.

Sortable Lists

- To write efficient sorting programs, we must access the private data members of the lists being sorted. Therefore, we shall add sorting functions as methods of our basic List data structures. The augmented list structure forms a new ADT that we shall call a Sortable_List, with the following form.

```
template <class Record>
class Sortable_list: public List<Record> {
public:                    //  Add prototypes for sorting methods here.
private:                   //  Add prototypes for auxiliary functions here.
};
```

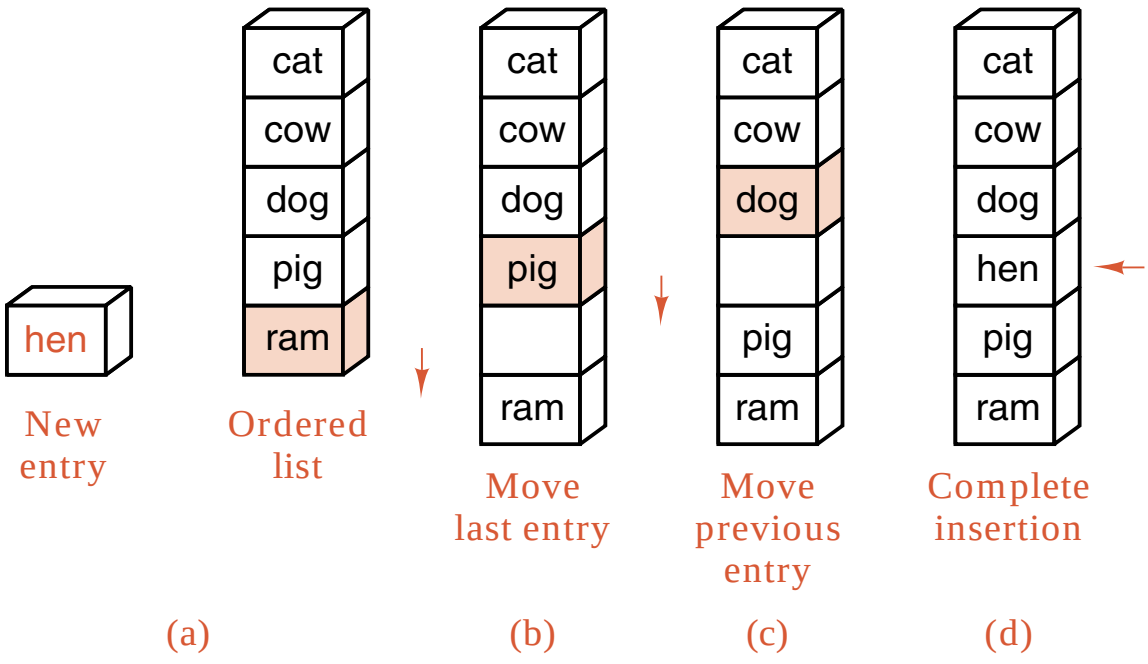
- Hence a Sortable_list is a List with extra sorting methods.
- The base list class can be any of the List implementations of Chapter 6.
- We use a template parameter class called Record to stand for entries of the Sortable_list.

Every Record has an associated key of type Key. A Record can be implicitly converted to the corresponding Key. The keys (hence also the records) can be compared under the operations ‘ < ,’ ‘ > ,’ ‘ >= ,’ ‘ <= ,’ ‘ == ,’ and ‘ != .’

- Any of the Record implementations discussed in Chapter 7 can be supplied, by a client, as the template parameter of a Sortable_list.

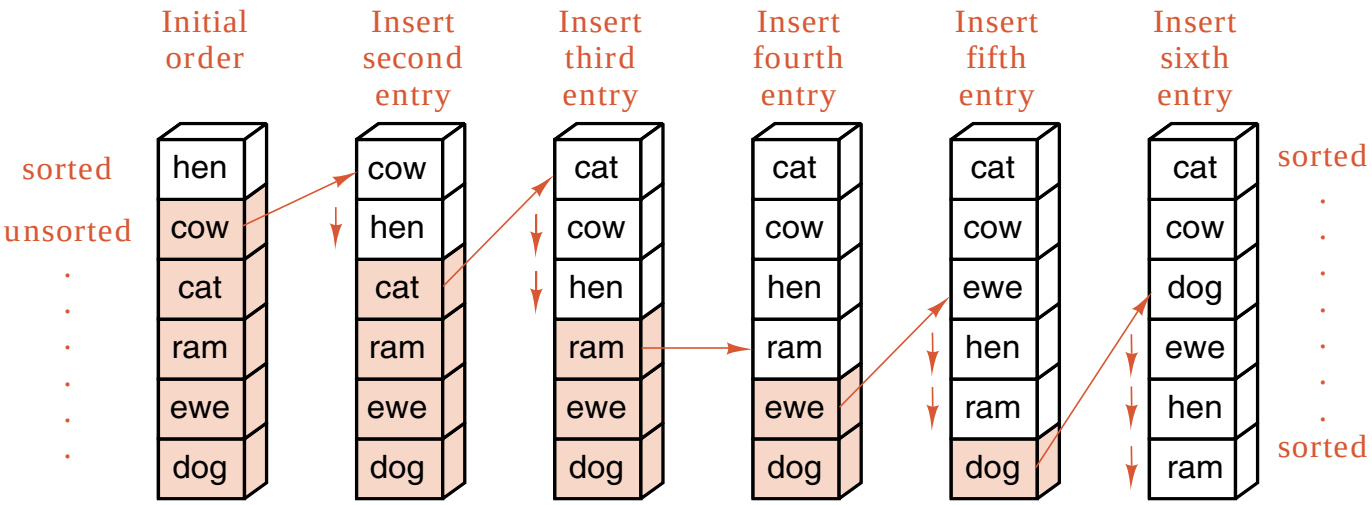
Ordered Insertion

- An ***ordered list*** is an abstract data type, defined as a list in which each entry has a key, and such that the keys are in order; that is, if entry i comes before entry j in the list, then the key of entry i is less than or equal to the key of entry j.
- For ordered lists, we shall often use two new operations that have no counterparts for other lists, since they use keys rather than positions to locate the entry.
  - One operation *retrieves* an entry with a specified key from the ordered list. Retrieval by key from an ordered list is exactly the same as searching.
  - The second operation, ***ordered insertion***, inserts a new entry into an ordered list by using the key in the new entry to determine where in the list to insert it.
- Note that ordered insertion is not uniquely specified if the list already contains an entry with the same key as the new entry, since the new entry could go into more than one position.

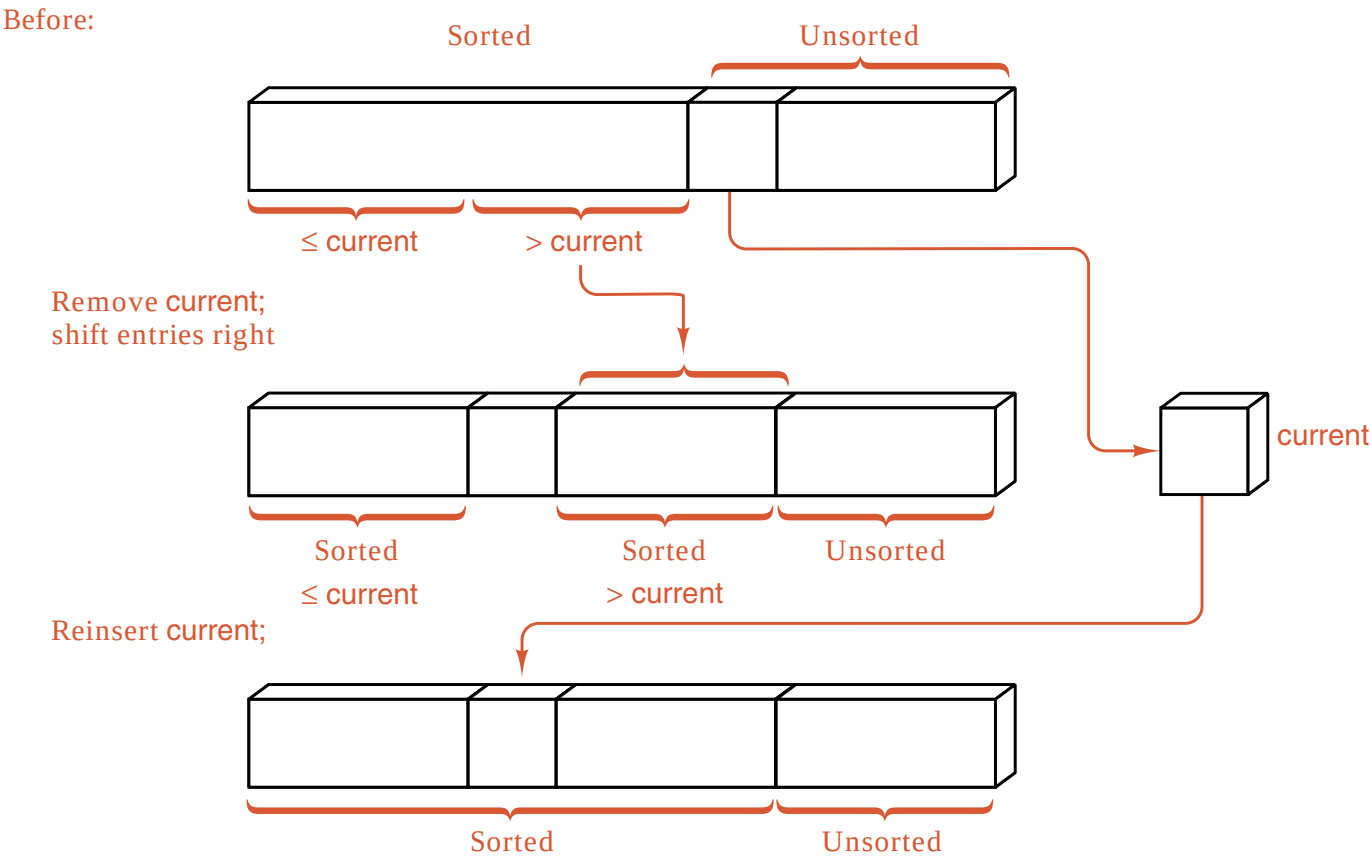


Insertion Sort

Example:



Main step, contiguous case:



Contiguous Insertion Sort

```
template <class Record>
void Sortable_list<Record> :: insertion_sort( )
/* Post:  The entries of the Sortable_list have been rearranged so that the keys in all
          the entries are sorted into nondecreasing order.
   Uses:  Methods for the class Record; the contiguous List implementation of Chapter
          6 */

{
   int first_unsorted;              //   position of first unsorted entry
   int position;                    //   searches sorted part of list
   Record current;                  //   holds the entry temporarily removed from list

   for (first_unsorted = 1; first_unsorted < count; first_unsorted++)
      if (entry[first_unsorted] < entry[first_unsorted - 1]) {
         position = first_unsorted;
         current = entry[first_unsorted];   //   Pull unsorted entry out of the list.
         do {                               //   Shift all entries until the proper position is found.
            entry[position] = entry[position - 1];
            position--;                     //   position is empty.
         } while (position > 0 && entry[position - 1] > current);
         entry[position] = current;
      }
}
```

Linked Insertion Sort

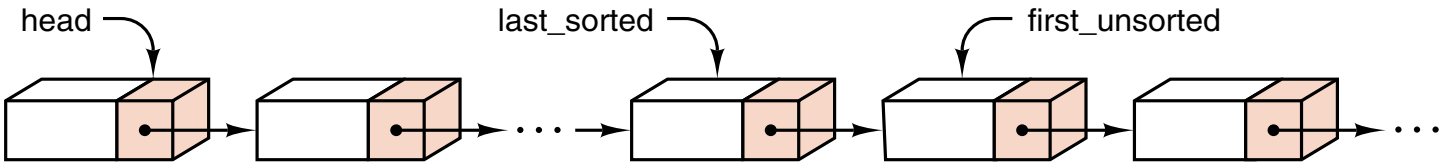
- With no movement of data, there is no need to search from the *end* of the sorted sublist, as for the contiguous case.
- Traverse the original list, taking one entry at a time and inserting it in the proper position in the sorted list.
- Pointer last_sorted references the end of the sorted part of the list.
- Pointer first_unsorted == last_sorted->next references the first entry that has not yet been inserted into the sorted sublist.
- Pointer current searches the sorted part of the list to find where to insert *first_unsorted.
- If *first_unsorted belongs before the head of the list, then insert it there.
- Otherwise, move current down the list until

```
first_unsorted->entry <= current->entry
```

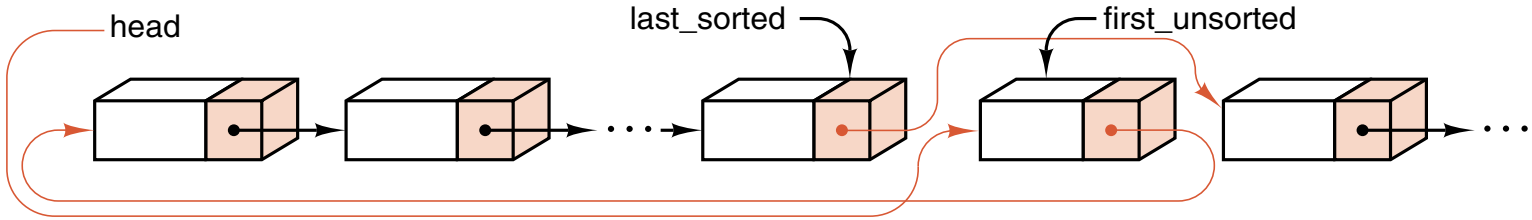
  and then insert *first_unsorted before *current. To enable insertion before *current, keep a second pointer trailing in lock step one position closer to the head than current.
- A ***sentinel*** is an extra entry added to one end of a list to ensure that a loop will terminate without having to include a separate check. Since last_sorted->next == first_unsorted, the node *first_unsorted is already in position to serve as a sentinel for the search, and the loop moving current is simplified.
- A list with 0 or 1 entry is already sorted, so by checking these cases separately we avoid trivialities elsewhere.

```
template <class Record>
void Sortable_list<Record> :: insertion_sort( )
/* Post:  The entries of the Sortable_list have been rearranged so that the keys in all
          the entries are sorted into nondecreasing order.
   Uses:  Methods for the class Record, linked List implementation of Chapter 6.  */
{ Node <Record> *first_unsorted,  //  the first unsorted node to be inserted
                *last_sorted,     //  tail of the sorted sublist
                *current,         //  used to traverse the sorted sublist
                *trailing;        //  one position behind current
  if (head != NULL) {             //  Otherwise, the empty list is already sorted.
    last_sorted = head;           //  The first node alone makes a sorted sublist.
    while (last_sorted->next != NULL) {
      first_unsorted = last_sorted->next;
      if (first_unsorted->entry < head->entry) {
        //  Insert *first_unsorted at the head of the sorted list:
        last_sorted->next = first_unsorted->next;
        first_unsorted->next = head;
        head = first_unsorted;  }
      else {                      //  Search the sorted sublist to insert *first_unsorted:
        trailing = head;
        current = trailing->next;
        while (first_unsorted->entry > current->entry) {
          trailing = current;
          current = trailing->next;
        }
        //  *first_unsorted now belongs between *trailing and *current.
        if (first_unsorted ==  current)
          last_sorted = first_unsorted;   //  already in right position
        else {
          last_sorted->next = first_unsorted->next;
          first_unsorted->next = current;
          trailing->next = first_unsorted;
}   }   }   }   }
```

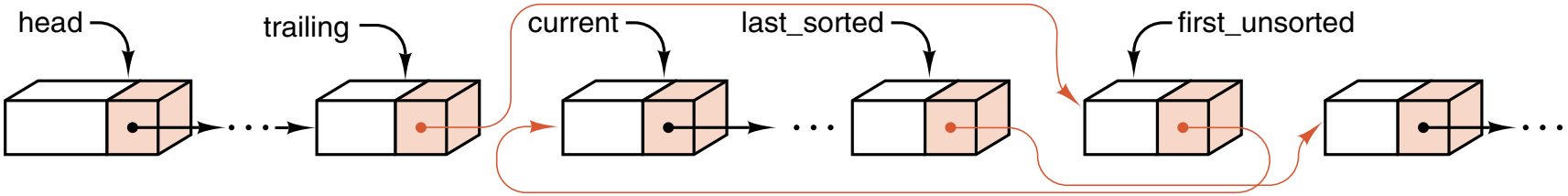
Partially sorted:



Case 1: *first_unsorted belongs at head of list



Case 2: *first_unsorted belongs between *trailing and *current



Analysis of Insertion Sort

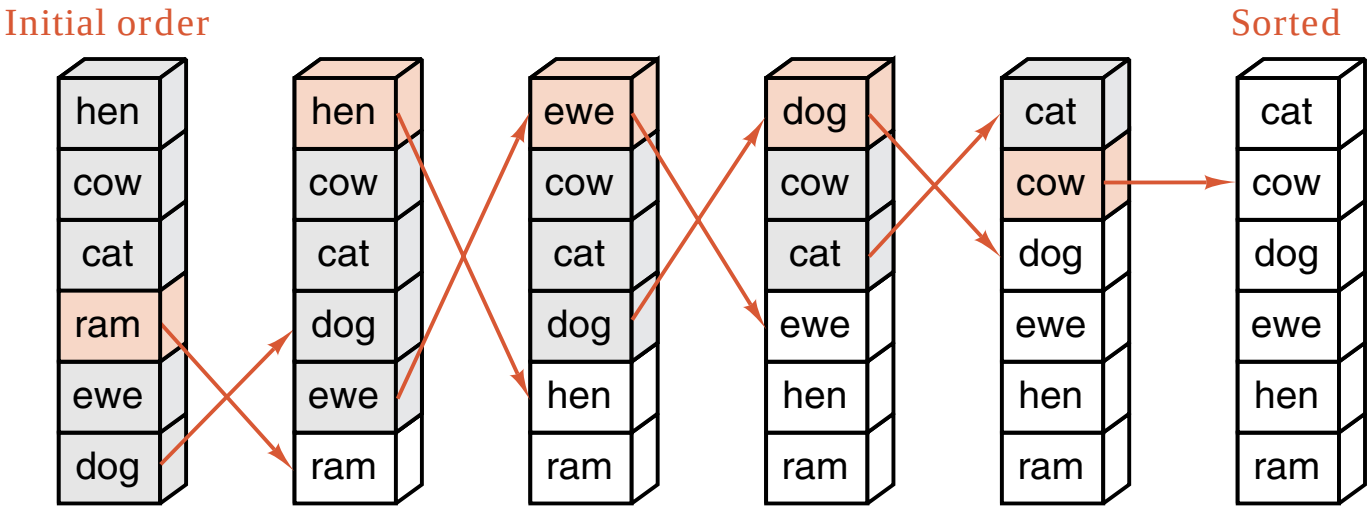
- The average number of comparisons for insertion sort applied to a list of n items in random order is $\frac{1}{4}n^2 + O(n)$.
- The average number of assignments of items for insertion sort applied to a list of n items in random order is also $\frac{1}{4}n^2 + O(n)$.
- The best case for contiguous insertion sort occurs when the list is already in order, when insertion sort does nothing except $n - 1$ comparisons of keys.
- The worst case for contiguous insertion sort occurs when the list is in reverse order.

THEOREM 8.1. Verifying that a list of n entries is in the correct order requires at least $n - 1$ comparisons of keys.

- Insertion sort verifies that a list is correctly sorted as quickly as any method can.

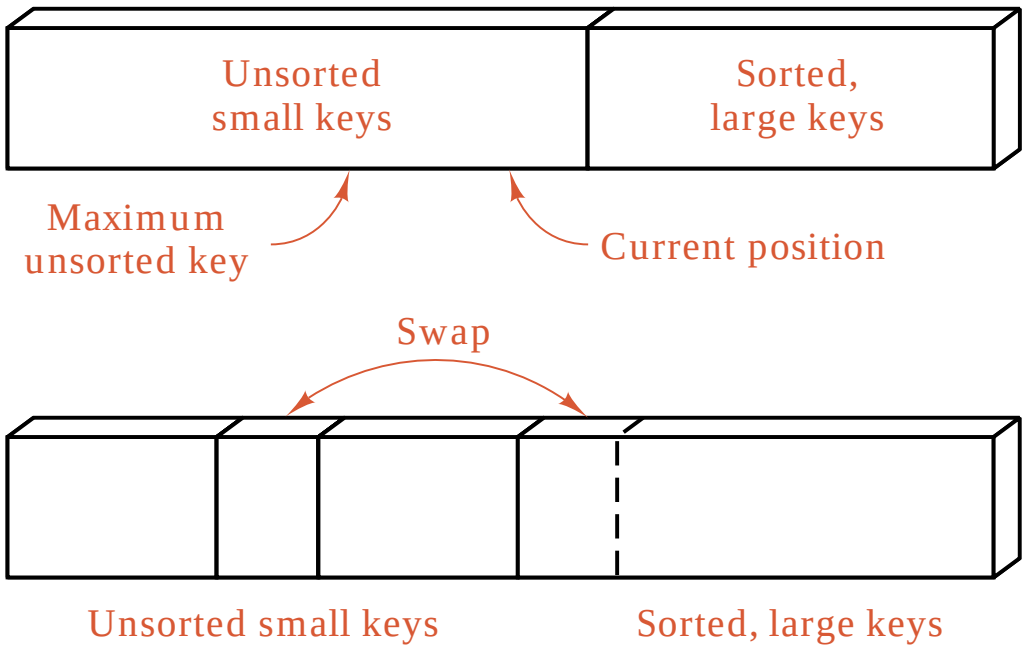
Selection Sort

Example:



Colored box denotes largest unsorted key.
Gray boxes denote other unsorted keys.

General step:



Selection sort:

```
template <class Record>
void Sortable_list<Record> :: selection_sort( )
/* Post:  The entries of the Sortable_list have been rearranged so that the keys
          in all the entries are sorted into nondecreasing order.
   Uses: max_key, swap.  */
{
   for (int position = count − 1; position > 0; position−−) {
      int max = max_key(0, position);
      swap(max, position);
   }
}
```

Find maximum key:

```
template <class Record>
int Sortable_list<Record> :: max_key(int low, int high)
/* Pre:   low and high are valid positions in the Sortable_list and low <= high.
   Post:  The position of the entry between low and high with the largest key is
          returned.
   Uses:  The class Record.  The contiguous List implementation of Chapter 6.
          */
{
   int largest, current;
   largest = low;
   for (current = low + 1; current <= high; current++)
      if (entry[largest] < entry[current])
         largest = current;
   return largest;
}
```

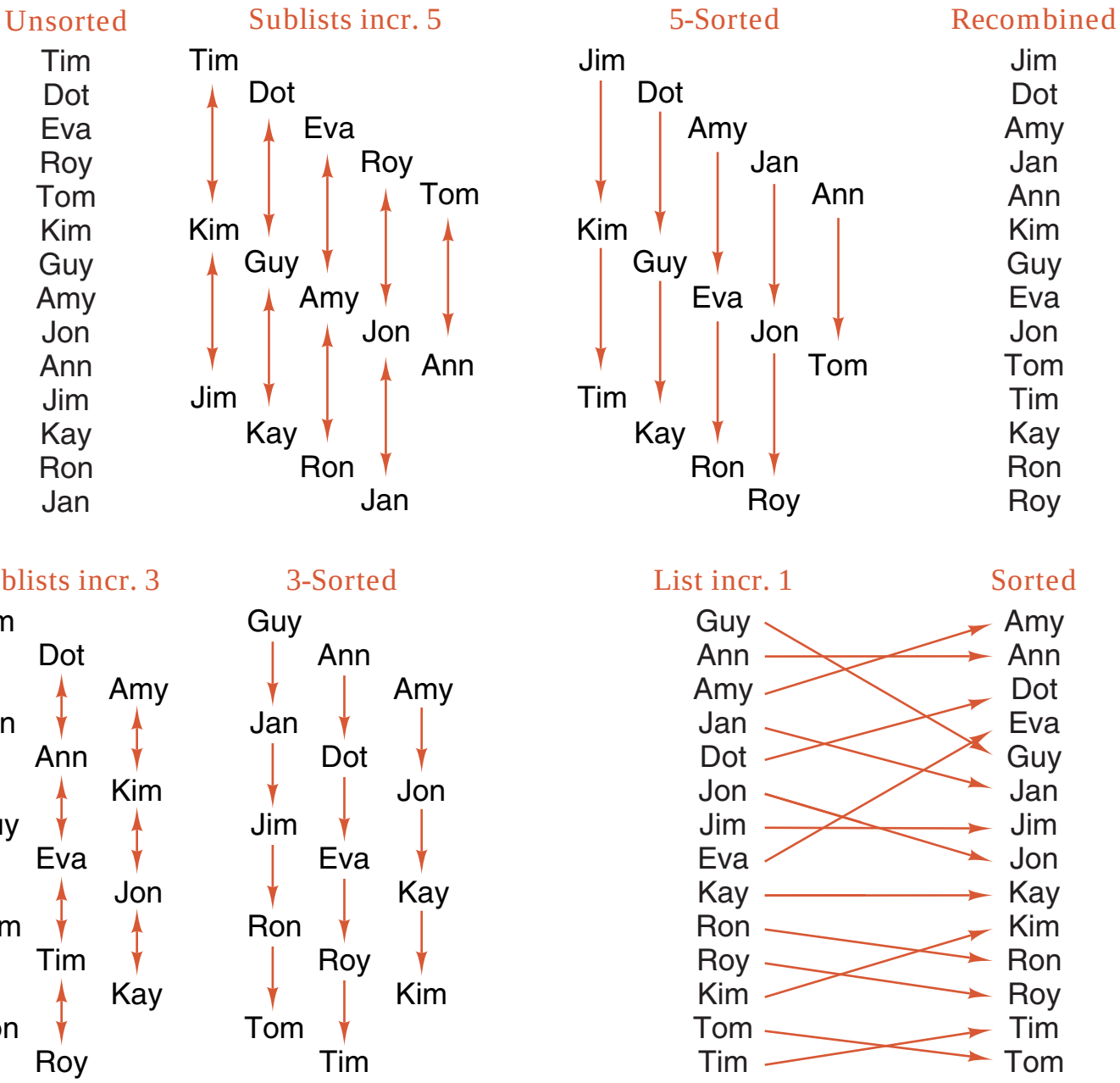
Swap entries:

```
template <class Record>
void Sortable_list<Record> :: swap(int low, int high)
/* Pre:   low and high are valid positions in the Sortable_list.
   Post:  The entry at position low is swapped with the entry at position high.
   Uses:  The contiguous List implementation of Chapter 6.  */
{
   Record temp;
   temp = entry[low];
   entry[low] = entry[high];
   entry[high] = temp;
}
```

Analysis and comparison:

	Selection	*Insertion (average)*
Assignments of entries	$3.0n + O(1)$	$0.25n^2 + O(n)$
Comparisons of keys	$0.5n^2 + O(n)$	$0.25n^2 + O(n)$

Example of Shell Sort



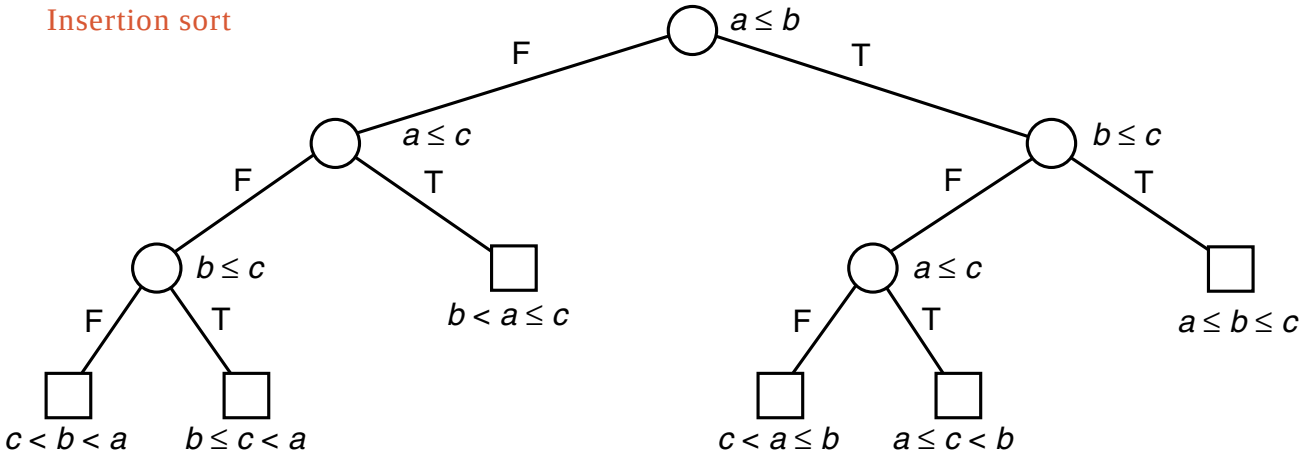
Shell Sort

```
template <class Record>
void Sortable_list<Record> :: shell_sort( )
/* Post:  The entries of the Sortable_list have been rearranged so that the keys in all
          the entries are sorted into nondecreasing order.
   Uses: sort_interval  */


{
   int increment,                    //    spacing of entries in sublist
       start;                        //    starting point of sublist
   increment = count;


   do {
      increment = increment/3 + 1;
      for (start = 0;  start < increment;  start++)
         sort_interval(start, increment);   //    modified insertion sort
   } while (increment > 1);
}
```

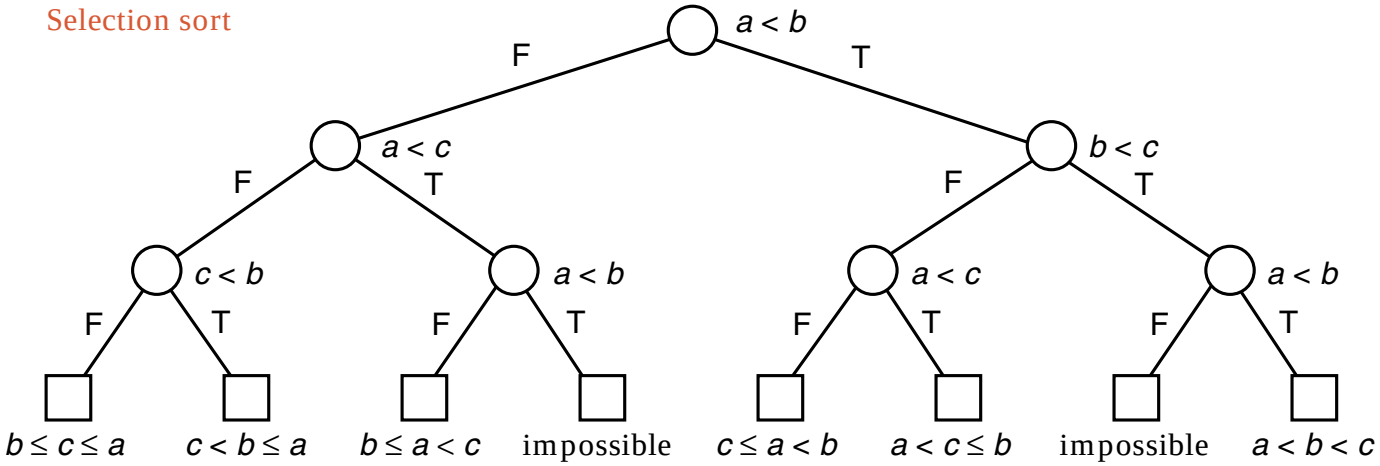
Lower Bounds for Sorting

How fast is it possible to sort?

Insertion sort



Selection sort



THEOREM 8.2 Any algorithm that sorts a list of n entries by use of key comparisons must, in its worst case, perform at least $\lceil \lg n! \rceil$ comparisons of keys, and, in the average case, it must perform at least $\lg n!$ comparisons of keys.

Divide and Conquer Sorting

Outline:

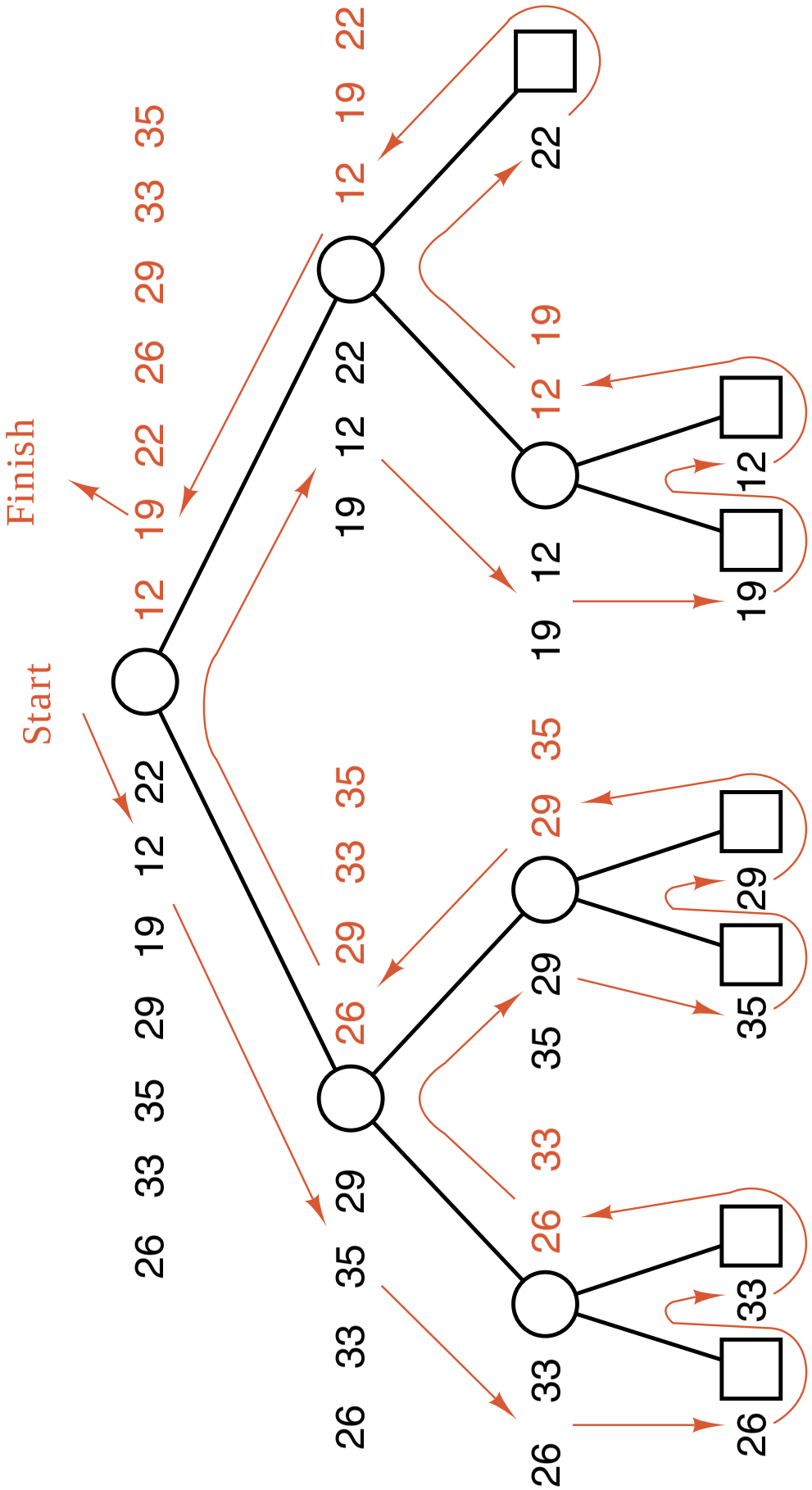
```
void Sortable_list :: sort( )
{
   if the list has length greater than 1 {
      partition the list into lowlist, highlist;
      lowlist.sort( );
      highlist.sort( );
      combine(lowlist, highlist);
   }
}
```

Mergesort:

We chop the list into two sublists of sizes as nearly equal as possible and then sort them separately. Afterward, we carefully merge the two sorted sublists into a single sorted list.

Quicksort:

We first choose some key from the list for which, we hope, about half the keys will come before and half after. Call this key the ***pivot***. Then we partition the items so that all those with keys less than the pivot come in one sublist, and all those with greater keys come in another. Then we sort the two reduced lists separately, put the sublists together, and the whole list will be in order.



Quicksort of 7 Numbers

Sort (26, 33, 35, 29, 12, 22)

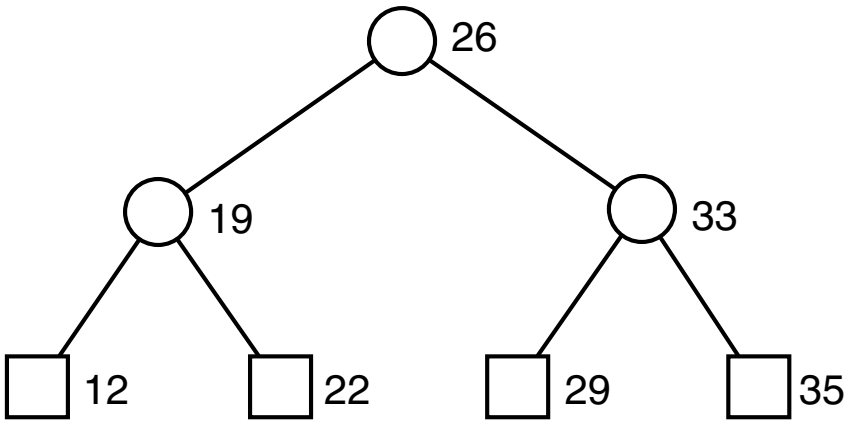
Partition into (19, 12, 22) and (33, 35, 29); pivot = 26
Sort (19, 12, 22)

Partition into (12) and (22); pivot = 19
Sort (12)
Sort (22)
Combine into (12, 19, 22)

Sort (33, 35, 29)

Partition into (29) and (35); pivot = 33
Sort (29)
Sort (35)
Combine into (29, 33, 35)

Combine into (12, 19, 22, 26, 29, 33 35)



Mergesort for Linked Lists

Interface method:

```
template <class Record>
void Sortable_list<Record> :: merge_sort( )
/* Post:  The entries of the sortable list have been rearranged so that their keys are
          sorted into nondecreasing order.
   Uses:  Linked List implementation of Chapter 6 and recursive_merge_sort.
          */
{
   recursive_merge_sort(head);
}
```

Recursive function:

```
template <class Record>
void Sortable_list<Record> ::
      recursive_merge_sort(Node<Record> * &sub_list)
/* Post:  The nodes referenced by sub_list have been rearranged so that their keys
          are sorted into nondecreasing order.  The pointer parameter sub_list is
          reset to point at the node containing the smallest key.
   Uses:  The linked List implementation of Chapter 6; the functions divide_from,
          merge, and recursive_merge_sort.  */
{
   if (sub_list != NULL && sub_list->next != NULL) {
      Node<Record> *second_half = divide_from(sub_list);
      recursive_merge_sort(sub_list);
      recursive_merge_sort(second_half);
      sub_list = merge(sub_list, second_half);
   }
}
```

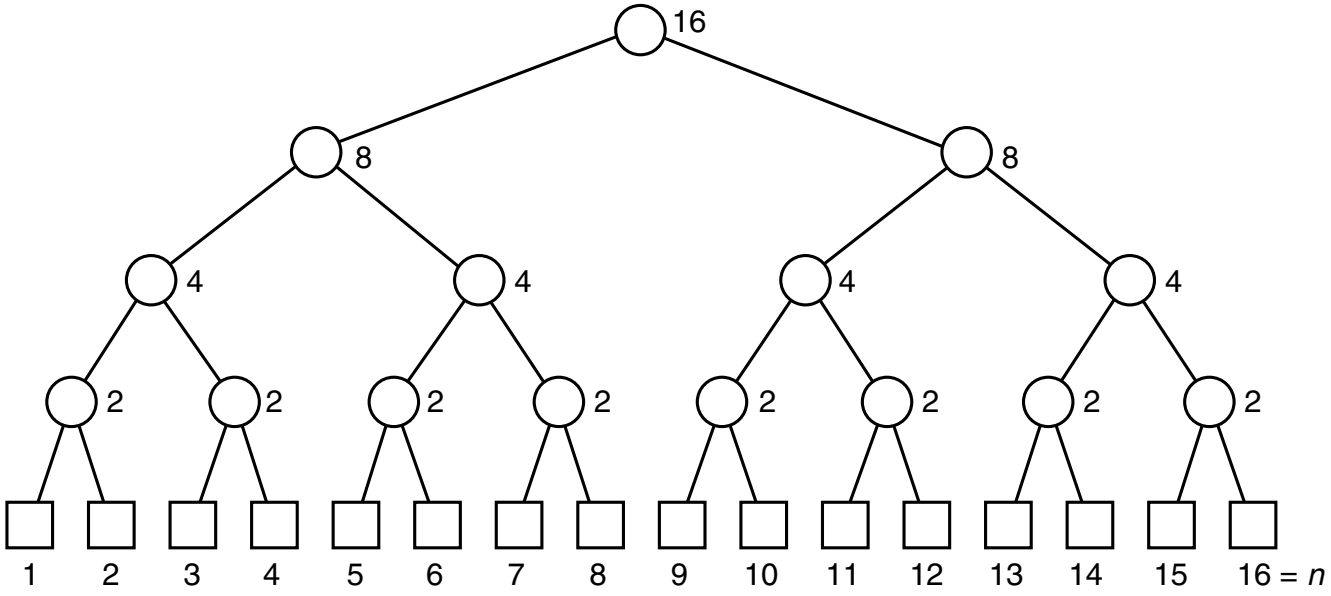
Divide Linked List in Half

```
template <class Record>
Node<Record> *Sortable_list<Record>::
                    divide_from(Node<Record> *sub_list)
/* Post:  The list of nodes referenced by sub_list has been reduced to its first half, and
          a pointer to the first node in the second half of the sublist is returned.  If the
          sublist has an odd number of entries, then its first half will be one entry larger
          than its second.
   Uses:  The linked List implementation of Chapter 6.  */

{
   Node<Record> *position,     //  traverses the entire list
                *midpoint,     //  moves at half speed of position to midpoint
                *second_half;

   if ((midpoint = sub_list) ==  NULL) return NULL;   //  List is empty.
   position = midpoint->next;
   while (position != NULL) {   //  Move position twice for midpoint's one move.
      position = position->next;
      if (position != NULL) {
         midpoint = midpoint->next;
         position = position->next;
      }
   }
   second_half = midpoint->next;
   midpoint->next = NULL;
   return second_half;
}
```

Merge Function

```
template <class Record>
Node<Record> *Sortable_list<Record> ::
        merge(Node<Record> *first, Node<Record> *second)
/* Pre:   first and second point to ordered lists of nodes.
   Post:  A pointer to an ordered list of nodes is returned.  The ordered list contains all
          entries that were referenced by first and second.  The original lists of nodes
          referenced by first and second are no longer available.
   Uses:  Methods for Record class; the linked List implementation of Chapter 6.  */
{
   Node<Record> *last_sorted;   //     points to the last node of sorted list
   Node<Record> combined;       //     dummy first node, points to merged list
   last_sorted =  &combined;
   while (first != NULL && second != NULL) {   //     Attach node with smaller key
      if (first->entry <= second->entry) {
         last_sorted->next = first;
         last_sorted = first;
         first = first->next;            //     Advance to the next unmerged node.
      }
      else {
         last_sorted->next = second;
         last_sorted = second;
         second = second->next;
      }
   }
//    After one list ends, attach the remainder of the other.
   if (first ==  NULL)
      last_sorted->next = second;
   else
      last_sorted->next = first;
   return combined.next;
}
```

Analysis of Mergesort



Lower bound:	$\lg n! \approx n \lg n - 1.44n + O(\log n)$
Mergesort:	$n \lg n - 1.1583n + 1,$
Insertion sort:	$\frac{1}{4}n^2 + O(n)$

Quicksort for Contiguous Lists

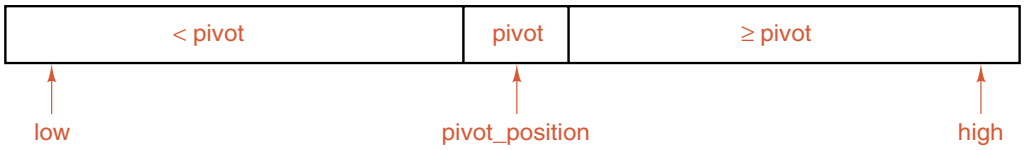
Interface method:

```
template <class Record>
void Sortable_list<Record> :: quick_sort( )
/* Post:  The entries of the Sortable_list have been rearranged so that their keys
          are sorted into nondecreasing order.
   Uses:  Contiguous List implementation of Chapter 6, recursive_quick_sort.
          */
{
   recursive_quick_sort(0, count − 1);
}
```

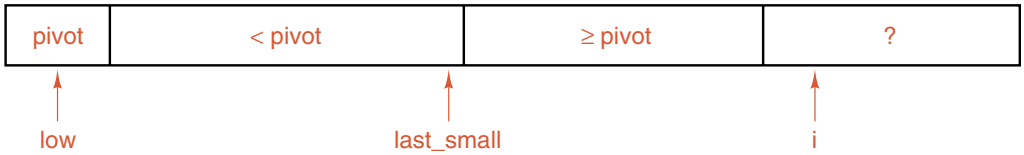
Recursive function:

```
template <class Record>
void Sortable_list<Record> :: recursive_quick_sort(int low, int high)
/* Pre:   low and high are valid positions in the Sortable_list.
   Post:  The entries of the Sortable_list have been rearranged so that their keys
          are sorted into nondecreasing order.
   Uses:  Contiguous List implementation of Chapter 6, recursive_quick_sort,
          and partition.  */
{
   int pivot_position;
   if (low < high) {                    //    Otherwise, no sorting is needed.
      pivot_position = partition(low, high);
      recursive_quick_sort(low, pivot_position − 1);
      recursive_quick_sort(pivot_position + 1, high);
   }
}
```

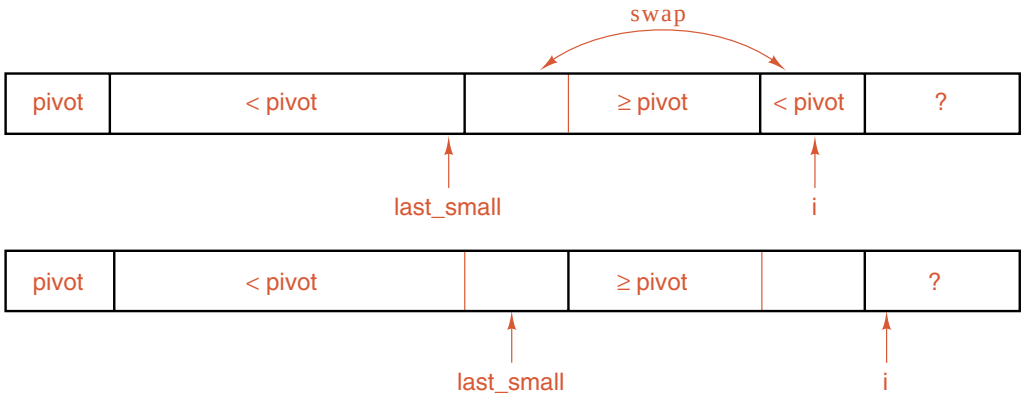
Goal (postcondition):



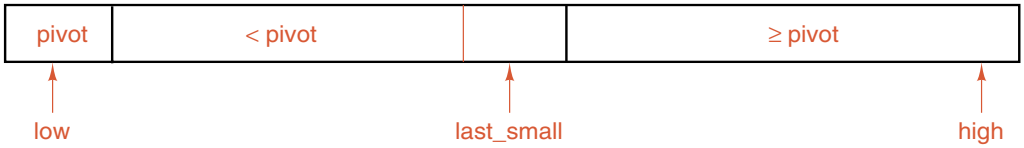
Loop invariant:



Restore the invariant:



Final position:



Partitioning the List

```
template <class Record>
int Sortable_list<Record> :: partition(int low, int high)
/* Pre:   low and high are valid positions of the Sortable_list, with low <= high.
   Post:  The center (or left center) entry in the range between indices low and high of
          the Sortable_list has been chosen as a pivot. All entries of the Sortable_list
          between indices low and high, inclusive, have been rearranged so that those
          with keys less than the pivot come before the pivot and the remaining entries
          come after the pivot. The final position of the pivot is returned.
   Uses: swap(int i, int j) (interchanges entries in positions i and j of a Sortable_list),
          the contiguous List implementation of Chapter 6, and methods for the class
          Record.  */
{
   Record pivot;
   int i,                              //  used to scan through the list
       last_small;                     //  position of the last key less than pivot
   swap(low, (low + high)/2);
   pivot = entry[low];                 //  First entry is now pivot.
   last_small = low;
   for (i = low + 1;  i <= high;  i++)

/* At the beginning of each iteration of this loop, we have the following conditions:
                If low < j <= last_small then entry[j].key < pivot.
                If last_small < j < i then entry[j].key >= pivot. */

      if (entry[i] < pivot) {
         last_small = last_small + 1;
         swap(last_small, i);          //  Move large entry to right and small to left.
      }
   swap(low, last_small);              //  Put the pivot into its proper position.
   return last_small;
}
```

Analysis of Quicksort

Worst-case analysis:

If the pivot is chosen poorly, one of the partitioned sublists may be empty and the other reduced by only one entry. In this case, quicksort is slower than either insertion sort or selection sort.

Choice of pivot:

- *First or last entry*: Worst case appears for a list already sorted or in reverse order.
- *Central entry*: Poor cases appear only for unusual orders.
- *Random entry*: Poor cases are very unlikely to occur.

Average-case analysis:

In its average case, quicksort performs

$$C(n) = 2n \ln n + O(n) \approx 1.39n \lg n + O(n)$$

comparisons of keys in sorting a list of n entries in random order.

Mathematical Analysis

- Take a list of n distinct keys in random order.
- Let $C(n)$ be the average number of comparisons of keys required to sort the list.
- If the pivot is the p^{th} smallest key, let $C(n, p)$ be the average number of key comparisons done.
- The partition function does $n - 1$ key comparisons, and the recursive calls do $C(p - 1)$ and $C(n - p)$ comparisons.
- For $n \geq 2$, we have $C(n, p) = n - 1 + C(p - 1) + C(n - p)$.
- Averaging from $p = 1$ to $p = n$ gives, for $n \geq 2$,

$$C(n) = n - 1 + \frac{2}{n}\big(C(0) + C(1) + \cdots + C(n - 1)\big).$$

An equation of this form is called a ***recurrence relation*** because it expresses the answer to a problem in terms of earlier, smaller cases of the same problem.

- Write down the same recurrence relation for case $n - 1$; multiply the first by n and the second by $n - 1$; and subtract:

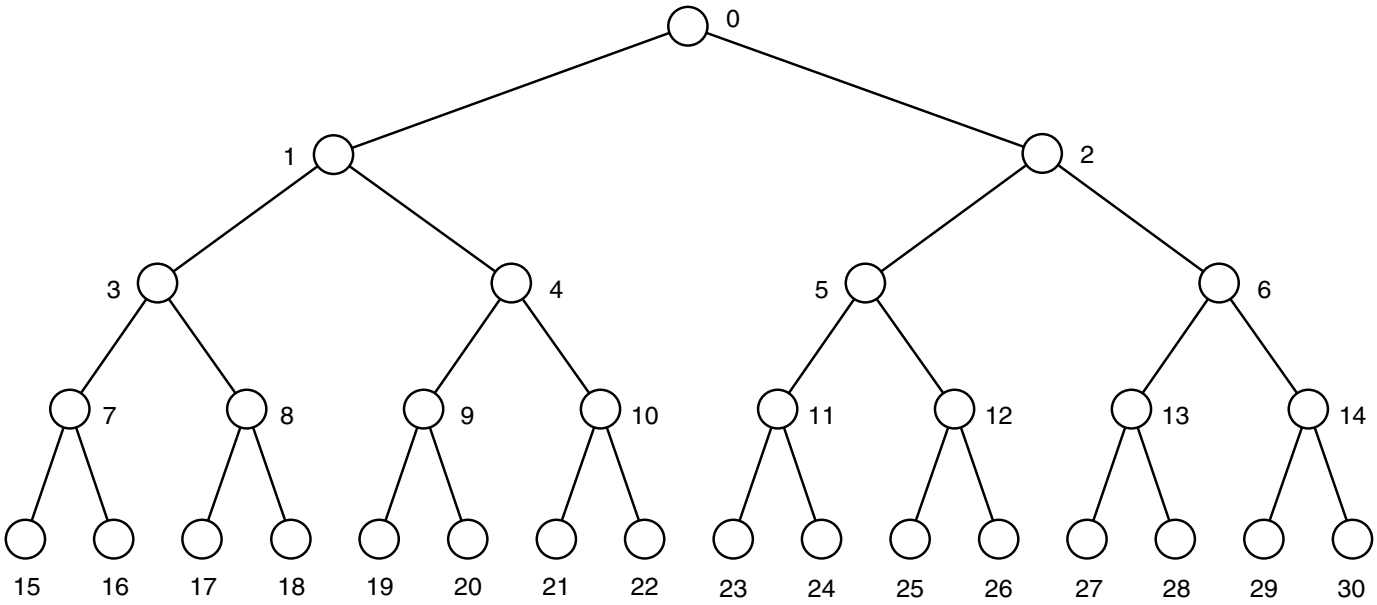
$$\frac{C(n) - 2}{n + 1} = \frac{C(n - 1) - 2}{n} + \frac{2}{n + 1}.$$

- Use the equation over and over to reduce to the case $C(1) = 0$:

$$\frac{C(n) - 2}{n + 1} = -1 + 2\left(\frac{1}{n + 1} + \frac{1}{n} + \frac{1}{n - 1} + \cdots + \frac{1}{4} + \frac{1}{3}\right).$$

- Evaluate the *harmonic number* in the sum: See Appendix A.

Trees, Lists, and Heaps

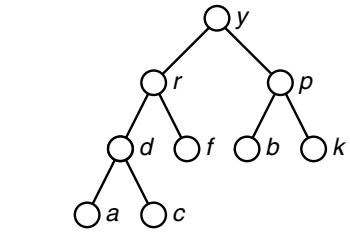


If the root of the tree is in position 0 of the list, then the left and right children of the node in position k are in positions $2k + 1$ and $2k + 2$ of the list, respectively. If these positions are beyond the end of the list, then these children do not exist.

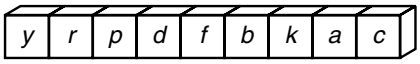
DEFINITION A ***heap*** is a list in which each entry contains a key, and, for all positions k in the list, the key at position k is at least as large as the keys in positions $2k$ and $2k + 1$, provided these positions exist in the list.

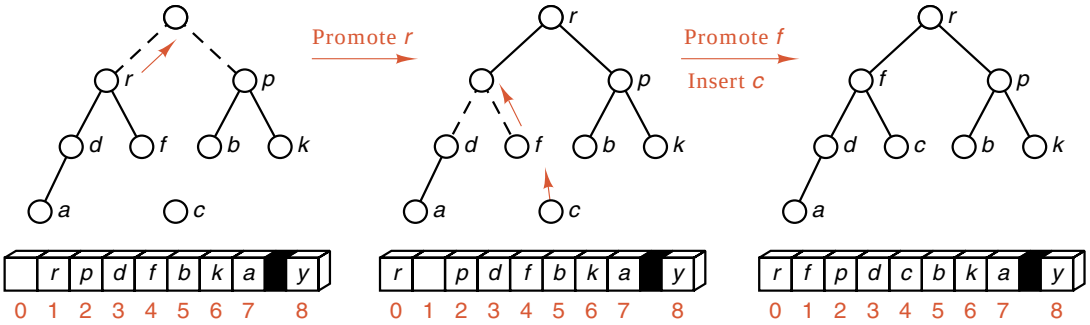
REMARK Many C++ manuals and textbooks refer to the area used for dynamic memory as the "heap"; this use of the word "heap" has nothing to do with the present definition.

Heapsort is suitable only for contiguous lists.

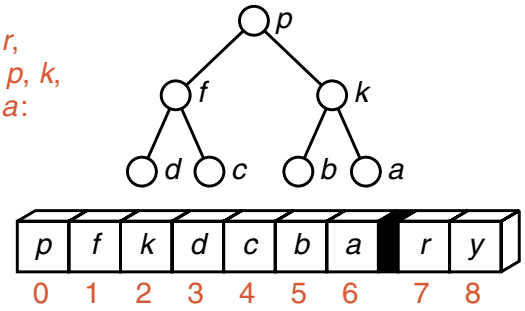


Heap

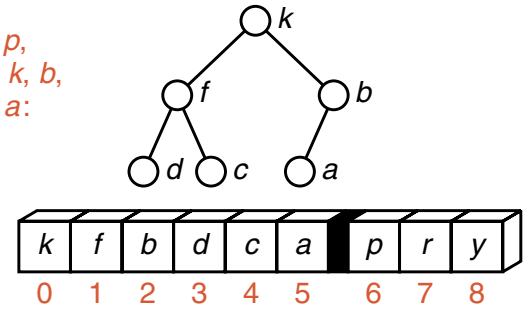




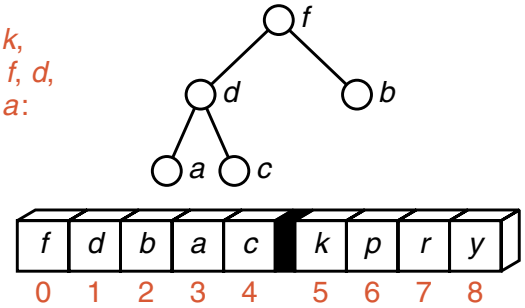
Remove *r*,
Promote *p*, *k*,
Reinsert *a*:



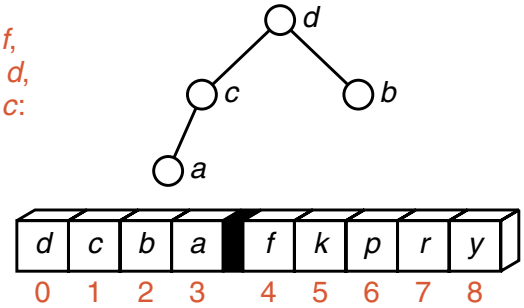
Remove *p*,
Promote *k*, *b*,
Reinsert *a*:



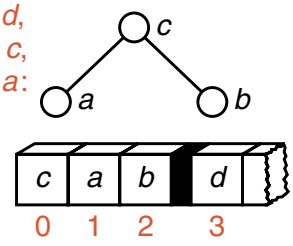
Remove *k*,
Promote *f*, *d*,
Reinsert *a*:



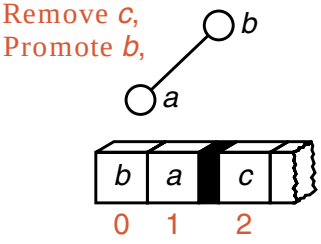
Remove *f*,
Promote *d*,
Reinsert *c*:



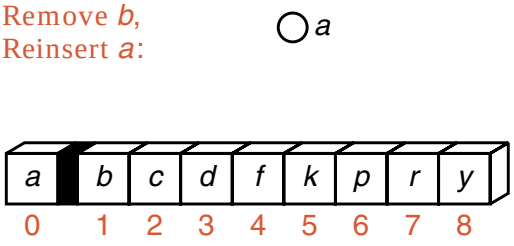
Remove *d*,
Promote *c*,
Reinsert *a*:



Remove *c*,
Promote *b*,



Remove *b*,
Reinsert *a*:



Main function for heapsort:

```
template <class Record>
void Sortable_list<Record> :: heap_sort( )
/* Post:  The entries of the Sortable_list have been rearranged so that their keys
          are sorted into nondecreasing order.
   Uses:  The contiguous List implementation of Chapter 6, build_heap, and in-
          sert_heap.  */
{
   Record current;              //   temporary storage for moving entries
   int last_unsorted;           //   Entries beyond last_unsorted have been sorted.
   build_heap( );               //   First phase:  Turn the list into a heap.
   for (last_unsorted = count - 1;  last_unsorted > 0;
                                    last_unsorted--) {
      current = entry[last_unsorted];   //   Extract last entry from list.
      entry[last_unsorted] = entry[0];  //   Move top of heap to the end
      insert_heap(current, 0, last_unsorted - 1);   //   Restore the heap
   }
}
```

Building the initial heap:

```
template <class Record>
void Sortable_list<Record> :: build_heap( )
/* Post:  The entries of the Sortable_list have been rearranged so that it becomes
          a heap.
   Uses:  The contiguous List implementation of Chapter 6, and insert_heap.  */
{
   int low;                     //   All entries beyond the position low form a heap.
   for (low = count/2 - 1;  low >= 0;  low--) {
      Record current = entry[low];
      insert_heap(current, low, count - 1);
   }
}
```

Insertion into a heap:

```
template <class Record>
void Sortable_list<Record> ::
      insert_heap(const Record &current, int low, int high)
/* Pre:   The entries of the Sortable_list between indices low + 1 and high,
          inclusive, form a heap. The entry in position low will be discarded.
   Post:  The entry current has been inserted into the Sortable_list and the
          entries rearranged so that the entries between indices low and high,
          inclusive, form a heap.
   Uses:  The class Record, and the contiguous List implementation of Chapter 6.
          */
{
   int large;                  //    position of child of entry[low] with the larger key
   large = 2 * low + 1;        //    large is now the left child of low.
   while (large <= high) {
      if (large < high && entry[large] < entry[large + 1])
         large++;              //    large is now the child of low with the largest key.
      if (current >= entry[large])
         break;                //    current belongs in position low.
      else {                   //    Promote entry[large] and move down the tree.
         entry[low] = entry[large];
         low = large;
         large = 2 * low + 1;
      }
   }
   entry[low] = current;
}
```

Analysis of Heapsort

In its worst case for sorting a list of length n, heapsort performs

$$2n \lg n + O(n)$$

comparisons of keys, and it performs

$$n \lg n + O(n)$$

assignments of entries.

Priority Queues

DEFINITION A ***priority queue*** consists of entries, such that each entry contains a key called the ***priority*** of the entry. A priority queue has only two operations other than the usual creation, clearing, size, full, and empty operations:

- Insert an entry.
- Remove the entry having the largest (or smallest) key.

If entries have equal keys, then any entry with the largest key may be removed first.

Comparison of Sorting Methods

- Use of space
- Use of computer time
- Programming effort
- Statistical analysis
- Empirical testing

Pointers and Pitfalls

1. Many computer systems have a general-purpose sorting utility. If you can access this utility and it proves adequate for your application, then use it rather than writing a sorting program from scratch.
2. In choosing a sorting method, take into account the ways in which the keys will usually be arranged before sorting, the size of the application, the amount of time available for programming, the need to save computer time and space, the way in which the data structures are implemented, the cost of moving data, and the cost of comparing keys.
3. Divide-and-conquer is one of the most widely applicable and most powerful methods for designing algorithms. When faced with a programming problem, see if its solution can be obtained by first solving the problem for two (or more) problems of the same general form but of a smaller size. If so, you may be able to formulate an algorithm that uses the divide-and-conquer method and program it using recursion.
4. Mergesort, quicksort, and heapsort are powerful sorting methods, more difficult to program than the simpler methods, but much more efficient when applied to large lists. Consider the application carefully to determine whether the extra effort needed to implement one of these sophisticated algorithms will be justified.
5. Priority queues are important for many applications. Heaps provide an excellent implementation of priority queues.
6. Heapsort is like an insurance policy: It is usually slower than quicksort, but it guarantees that sorting will be completed in $O(n \log n)$ comparisons of keys, as quicksort cannot always do.