Lesson 3:
Sorting, Searching

Raymond Mitchell III

3.1

3.1 Sorting Introduction

Sorting

- Arranging set of elements in a prescribed order
- Many algorithms available that perform sorting

Purpose

- Organize data to be more meaningful for display
- Some algorithms require data to be sorted

Example: sorting ints

- Unsorted:
  - 9, 3, 13, 1, 22
- Sorted – ascending order
  - 1, 3, 9, 13, 22
- Sorted – descending order
  - 22, 13, 9, 3, 1

Example: sorting people

- Person struct:

```
typedef struct Person_
{
	char firstName[MAX_NAME_LENGTH];
	char lastName[MAX_NAME_LENGTH];
} Person;
```

- Unsorted:
  1. { “Joe”, “Smith” }
  2. { “Jane”, “Doe” }
  3. { “Sam”, “Jones” }
  4. { “Jane”, “Smith” }
- Sorted – first name ascending
  1. { “**Jane**”, “Doe” }
  2. { “**Jane**”, “Smith” }
  3. { “**Joe**”, “Smith” }
  4. { “**Sam**”, “Jones” }

  …OR…

  1. { “**Jane**”, “*Smith*” }
  2. { “**Jane**”, “*Doe*” }
  3. { “**Joe**”, “Smith” }
  4. { “**Sam**”, “Jones” }

- Sorted – first name ascending, last name descending (subsort)
  - `{ "`**Jane**`", "Smith" }`
  - `{ "`**Jane**`", "Doe" }`
  - `{ "`**Joe**`", "Smith" }`
  - `{ "`**Sam**`", "Jones" }`

3.2 Sorting Properties

Sorting classes

- Comparison sorts
  - Rely on comparing elements to place them in correct order
  - Requires comparison function to compare two elements
    - Comparison functions described in-full later
  - Fastest possible time: O(n lg n)
- Linear-time sorts
  - Rely on certain characteristics in data
  - Requires that data possess characteristics needed by sort algorithm
  - Cannot always be applied
  - Fasted possible time: O(n)

Space requirements

- In-place
  - No additional space required during the sort
- Extra storage
  - Additional space required during the sort

Stability

- Unstable
  - Elements that compare as equal *may be reordered* during the sort
  - Example: unstable sort
    - Employee struct:

```
typedef struct Employee_
{
    int id;
    char name[MAX_NAME_LENGTH];
} Employee;
```

    - Unsorted:
      - { 1, “Ray” }
      - { 3, “Joe” }
      - { 2, “Ray”}
      - { 4, “Joe” }
    - Sorted – descending name (unstable):
      - { 2, **“Ray”** } ← Position swapped with ID 1
      - { 1, **“Ray”** }
      - { 4, **“Joe”** } ← Position swapped with ID 3
      - { 3, **“Joe”** }
- Stable
  - Elements that compare as equal *will not be reordered* during the sort

- Example: stable sort
  - Employee struct:

```
typedef struct Employee_
{
    int id;
    char name[MAX_NAME_LENGTH];
} Employee;
```

  - Unsorted:
    - { 1, “Ray” }
    - { 3, “Joe” }
    - { 2, “Ray” }
    - { 4, “Joe” }
  - Sorted – descending name (unstable):
    - { 1, **“Ray”** } ← Position unchanged with ID 2
    - { 2, **“Ray”** }
    - { 3, **“Joe”** } ← Position unchanged with ID 4
    - { 4, **“Joe”** }

3.3 Sorting Applications

Order statistics

- Finding the *i*th smallest element in a set
- One simplistic approach – sort the set then select *i*th element

Binary search

- Efficient search method
- Requires data to be sorted

Directory listings

- Operating system typically sorts file listing by various criteria before displaying
- Sorted listing makes data easier for person to understand

Database systems

- Large amounts of data must be stored & retrieved quickly
- Storing data in sorted arrangement allows fast reads & writes

Spell checkers

- Programs that check spelling of words in text
- Dictionary stored in sorted arrangement allows efficient searches

Spreadsheets

- Important for business, financial, scientific data
- Data more meaningful when sorted
- Sorting / subsorting can occur on multiple columns

3.4 Comparison Functions

Required for comparison sorts

- Must understand comparison functions before we can look at comparison sorts

Comparison function

- Takes pointers to two elements
- Returns <0 if first element should appear **before** second
- Returns >0 if first element should appear **after** second
- Returns 0 if elements are equal

Example: strcmp

- C library function “strcmp” is a comparison function for null-terminated strings

Example: custom comparison functions

```
/*
 * Demonstration of comparison functions.
 *
 * Program output:
        Compare by ID = -1
        Compare by ID = 0
        Compare by name = 1
        Compare by name = 0
 */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define MAX_NAME_LENGTH 256

/* A type for us to compare */
typedef struct Employee_
{
    int id;
```

```
    char name[MAX_NAME_LENGTH];
} Employee;

/* Comparison function 1 */
int compareEmployeesById(const void *pKey1, const void *pKey2)
{
    Employee *pEmp1;
    Employee *pEmp2;

    pEmp1 = (Employee *)pKey1;
    pEmp2 = (Employee *)pKey2;

    if (pEmp1->id > pEmp2->id) {
        return 1;
    }
    if (pEmp1->id < pEmp2->id) {
        return -1;
    }
    return 0;
}

/* Comparison function 2 */
int compareEmployeesByName(const void *pKey1, const void *pKey2)
{
    Employee *pEmp1;
    Employee *pEmp2;
    int strcmpResult;

    pEmp1 = (Employee *)pKey1;
    pEmp2 = (Employee *)pKey2;

    strcmpResult = strcmp(pEmp1->name, pEmp2->name);

    if (strcmpResult > 0) {
        return 1;
    }
    if (strcmpResult < 0) {
        return -1;
    }
    return 0;
}

```

```
int main()
{
    Employee emp1 = { 1, "Ray" };
    Employee emp2 = { 2, "Joe" };
    int (*compare)(const void *key1, const void *key2);

    /* Compare by ID */
    compare = compareEmployeesById;
    printf("Compare by ID = %d\n", compare(&emp1, &emp2));      /* -1 */
    printf("Compare by ID = %d\n", compare(&emp1, &emp1));      /* 0 */

    /* Compare by name */
    compare = compareEmployeesByName;
    printf("Compare by name = %d\n", compare(&emp1, &emp2));    /* 1 */
    printf("Compare by name = %d\n", compare(&emp1, &emp1));    /* 0 */

    return EXIT_SUCCESS;
}
```

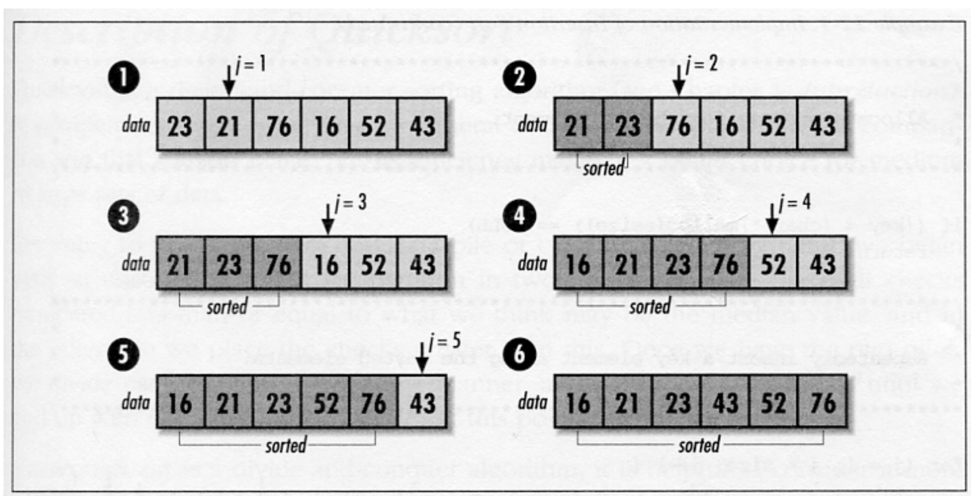
3.5 Insertion Sort

Overview

- Comparison sort
- One of simplest sorting algorithms
- Example: Sorting pile of canceled checks by hand
  - Start with pile of unsorted checks
  - One at a time, remove a check from unsorted pile and put in proper position in sorted pile
- Not the fastest sort
  - Inefficient speed for large sets of data
- At first appears would require additional space, but can represent unsorted and sorted “piles” in-place
- Good at inserting element into already sorted set (see example of use)
  - Inserting one element requires only one scan of sorted elements (incremental sort) as opposed to complete run of the algorithm

Visualization

- j (looks like an i in diagram) is next element to be added to sorted sequence
- Elements to left of j are sorted
- Elements to right of j (including j) are unsorted
- Procedure:
  - Start with j = 1
  - Save value at j
  - Starting with element to left of j, shift each element larger than j one position to the right, stop once position for j is found
  - Assign original value at j to position
  - Repeat until end of sequence reached at which point entire sequence is sorted



Time

- O(n^2)
  - Note: Inserting element into already sorted set is O(n)

Space

- In-place

Stability

- Stable

Data structures supported

- Array
- Doubly-linked list

When to use?

- Small sets of data
- If you must use linked list data structure
- *When solving problems that involve incremental sorting:* Inserting new elements into sorted set
- Example: Hotel reservation system
    - One display in the system lists all guests, sorted by name
    - New guests inserted as they arrive (requiring only single scan for insertion point)

3.6 Insertion Sort Implementation

Interface

```
int issort(void *data, int size, int esize,
      int (*compare)(const void *key1, const void *key2))
```

- Parameters
  - data – Pointer to array of data to be sorted
  - size – Number of elements in data
  - esize – Size of each element
  - compare – Comparison function used to compare elements
- Returns
  - 0 if sorting successful, -1 otherwise

Implementation

```c
/*
 * issort.c
 */
#include <stdlib.h>
#include <string.h>

#include "sort.h"

int issort(void *data, int size, int esize,
int (*compare)(const void *key1, const void *key2))
{
    char *a = data;
    void *key;
    int   i,
          j;

    /* Allocate storage for the key element. */
    if ((key = (char *)malloc(esize)) == NULL)
        return -1;

    /* Repeatedly insert a key element among the sorted elements. */
    for (j = 1; j < size; j++) {
        memcpy(key, &a[j * esize], esize);
        i = j - 1;

        /* Determine the position at which to insert the key element. */
        while (i >= 0 && compare(&a[i * esize], key) > 0) {
            memcpy(&a[(i + 1) * esize], &a[i * esize], esize);
            i--;
        }

        memcpy(&a[(i + 1) * esize], key, esize);
    }

    /* Free the storage allocated for sorting. */
    free(key);

    return 0;
}
```

3.7 Quick Sort

Description

- Comparison sort
- Divide and conquer algorithm
- Widely regarded as best sort for general use
- Example: Sorting pile of canceled checks by hand
  - Start with pile of unsorted checks
  - Partition pile in two
    - In one pile place all checks less than or equal to what we think might be the median value
    - In the other pile place all checks greater than this
  - Once we have two piles, divide each of them in the same manner
  - Repeat until we end up with one check in every pile
  - At this point the checks are sorted
- Worst case performs same as insertion sort (O(n^2))
  - We can make worst case so unlikely that we can count on algorithm performing in average case (O(n lg n))
  - Key to reaching average case performance is selecting partition value that results in balanced sub-groups
- Partitioning example: imbalanced – *worst-case performance*
  - Unsorted: `{ 15, 20,  18, 51, 36, 10, 77, 43 }`
  - Partition: `10`
  - Sub-groups: `{ 10 }, { 15, 20, 18, 51, 36, 77, 43 }`
- Partitioning example: balanced – *average-case performance*
  - Unsorted: `{ 15, 20,  18, 51, 36, 10, 77, 43 }`
  - Partition: `36`
  - Sub-groups: `{ 15, 20, 18, 10 }, { 36, 51, 77, 43 }`
- Selecting good partition
  - Selecting randomly
  - Median-of-three method
    - Randomly select 3 elements
    - Select median value as partition
    - Virtually guarantees average-case performance

- Quick sort is a randomized algorithm (because partition chosen randomly)

Visualization – partition

- Procedure:
  - Pivot value selected from median-of-three random values
  - Move k to left until value found less than pivot value
  - Move i to right until value found greater than pivot value
  - If k & i did not cross paths
    - Swap values at k & i (putting values on correct side of pivot)
    - Repeat procedure

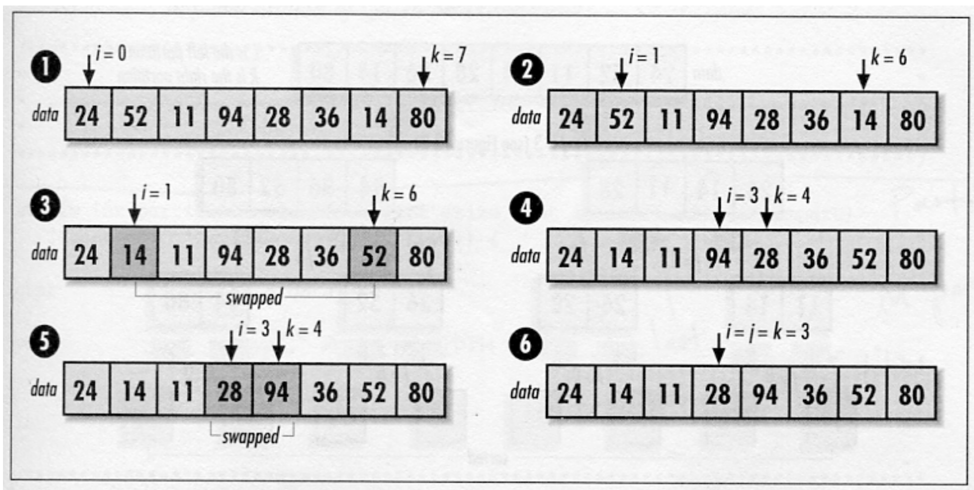


Figure 12-2. Partitioning around 28

Visualization – quicksort

- Procedure:
  - If array has more than one element
    - Partition array
    - quicksort left partition
    - quicksort right partition

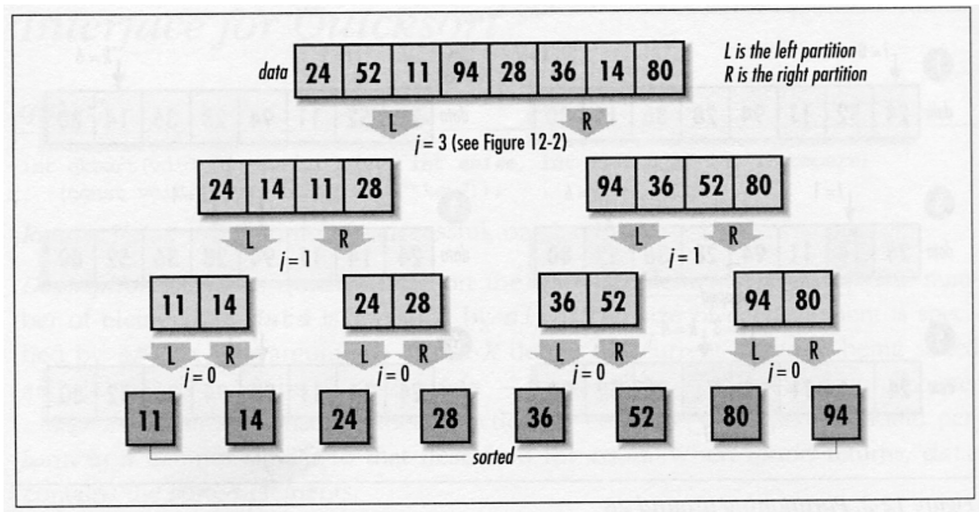


Figure 12-3. Sorting with quicksort assuming optimal partitioning

Time

- O(n lg n) – Average case
  - By selecting random partition value we can virtually guarantee average case performance

Space

- In-place

Stability

- Unstable

Data structures supported

- Array – requires random access

When to use?

- Any general sort
- Medium and large data sets
- Only restriction is that space must be provided to store all elements in memory
  - Unpredictable partition size prevents identifying space required to hold partition
    - Cannot break sort into guaranteed smaller pieces
  - Merge sort can be used when entire sequence to be sorted cannot be held in memory

3.8 Quick Sort Implementation

Interface

```
int qksort(void *data, int size, int esize, int i, int k,
        int (*compare)(const void *key1, const void *key2))
```

- Parameters
  - data – Pointer to array of data to be sorted
  - size – Number of elements in data
  - esize – Size of each element
  - i – left index of range in data to be sorted
  - k – right index of range in data to be sorted
  - compare – Comparison function used to compare elements
- Returns
  - 0 if sorting successful, -1 otherwise

Implementation

```
/*
 * qksort.c
 */
#include <stdlib.h>
#include <string.h>

#include "sort.h"

static int compare_int(const void *int1, const void *int2)
{
    /* Compare two integers (used during median-of-three partitioning). */
    if (*(const int *)int1 > *(const int *)int2)
        return 1;
    else if (*(const int *)int1 < *(const int *)int2)
        return -1;
    else
        return 0;
}

static int partition(void *data, int esize, int i, int k,
        int (*compare)(const void *key1, const void *key2))
{
    char *a = data;
    void *pval,
          *temp;
    int r[3];

    /* Allocate storage for the partition value and swapping. */
    if ((pval = malloc(esize)) == NULL)
        return -1;

    if ((temp = malloc(esize)) == NULL) {
        free(pval);
        return -1;
    }

    /* Use the median-of-three method to find the partition value. */
    r[0] = (rand() % (k - i + 1)) + i;  /* Get random index in range [i, k] */
    r[1] = (rand() % (k - i + 1)) + i;  /* "" */
    r[2] = (rand() % (k - i + 1)) + i;  /* "" */
```

```
    /*
     * BUG:  The next two lines are selecting the median INDEX from the three
     * randomly generated indices.  These lines should instead be selecting
     * the median VALUE from the values at the three randomly generated
     * indices.  This bug effectively causes a random value to be selected
     * for the pivot rather than the median-of-three random values.  This means
     * quicksort will not be virtually guaranteed to perform in average case
     * O(nlgn) and instead may perform in worst case O(n^2).
     */
    issort(r, 3, sizeof(int), compare_int); /* Sort random indices */
    memcpy(pval, &a[r[1] * esize], esize);  /* Set pivot = value at mid index */

    /* Create two partitions around the partition value. */
    i--;
    k++;

    while (1) {
        /* Move left until an element is found in the wrong partition. */
        do {
            k--;
        } while (compare(&a[k * esize], pval) > 0);

        /* Move right until an element is found in the wrong partition. */
        do {
            i++;
        } while (compare(&a[i * esize], pval) < 0);

        if (i >= k) {
            /* Stop partitioning when the left and right counters cross. */
            break;
        }
        else {
            /* Swap the elements now under the left and right counters. */
            memcpy(temp, &a[i * esize], esize);
            memcpy(&a[i * esize], &a[k * esize], esize);
            memcpy(&a[k * esize], temp, esize);
        }
    }

    /* Free the storage allocated for partitioning. */
    free(pval);
    free(temp);
```

```

    /* Return the position dividing the two partitions. */
    return k;
}

int qksort(void *data, int size, int esize, int i, int k,
        int (*compare)(const void *key1, const void *key2))
{
    int j;

    /* Stop the recursion when it is not possible to partition further. */
    if (i < k) {
        /* Determine where to partition the elements. */
        if ((j = partition(data, esize, i, k, compare)) < 0)
            return -1;

        /* Recursively sort the left partition. */
        if (qksort(data, size, esize, i, j, compare) < 0)
            return -1;

        /* Recursively sort the right partition. */
        if (qksort(data, size, esize, j + 1, k, compare) < 0)
            return -1;
    }

    return 0;
}
```

3.9 Merge Sort

Description

- Comparison sort
- Divide and conquer algorithm
- In all cases performs as well as average case of quicksort
- Requires twice the space of the unsorted data
- Valuable for large sets of data
  - Divides data into predictable sizes (unlike quick sort)
  - Does not require all elements to be in memory at one time
- Example: Sorting pile of canceled checks by hand
  - Start with pile of unsorted checks
  - Divide the pile in half
  - Divide each of the resulting piles in half
  - Repeat until all piles contain a single check
  - Merge the piles two-by-two so each new pile is a sorted combination of the two that were merged
  - Repeat until we end up with one big pile again
  - At this point the checks are sorted

Visualization – merge

- Procedure:
  - While ipos hasn't reached j and jpos hasn't reached the end of the list
    - Compare the values at ipos and jpos
    - Place the smaller value at the next position in the merged list and increment that index
  - Add all remaining elements pointed to by ipos or jpos to the end of the merged list (only ipos or jpos will point to elements)

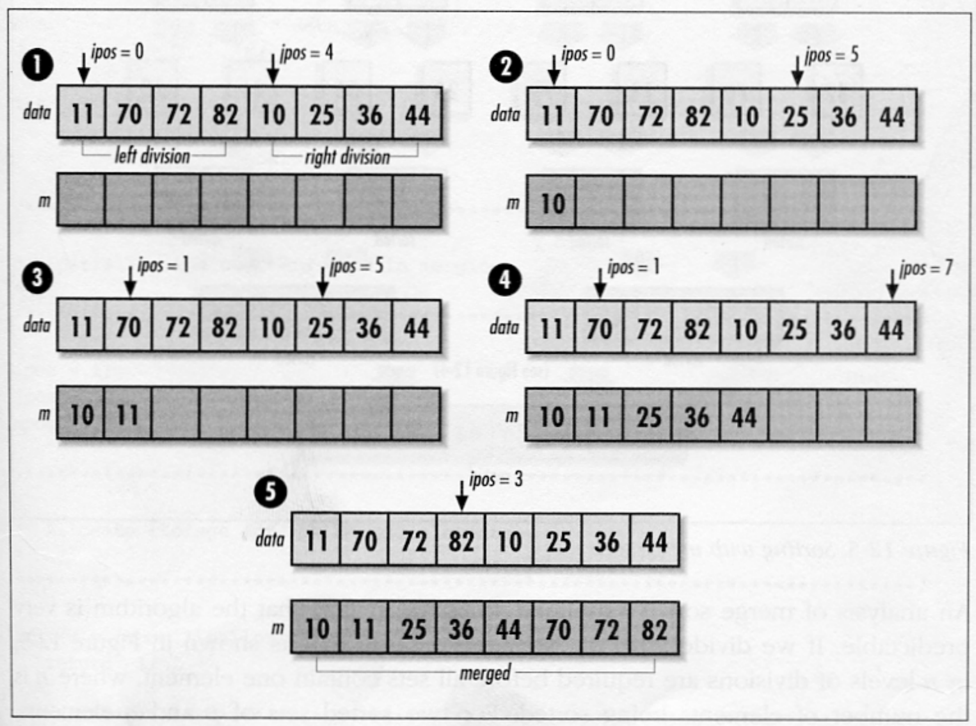


Figure 12-4. Merging two sorted sets

Visualization – mergesort

- Procedure:
  - (light gray) If the list has more than one element
    - Split the list into left and right halves
    - mergesort the left half
    - mergesort the right half
  - (dark gray) merge the left and right halves

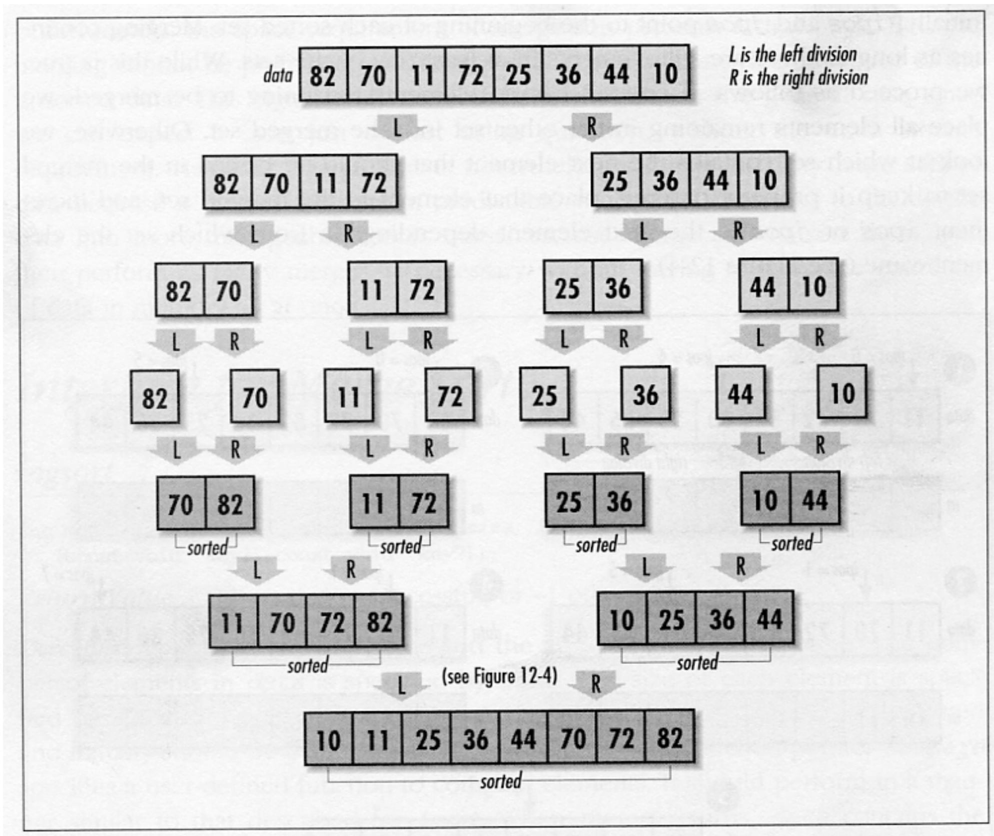


Figure 12-5. Sorting with merge sort

Time

- O(n lg n)
  - lg n levels of divisions are required
  - For each of the lg n levels, we traverse all n elements to merge

Space

- Twice the space of the original list

Stability

- Stable

Data structures supported

- Array
- Singly-linked list
- Doubly-linked list

When to use?

- When dealing with very large sets; when there's not enough memory to hold all elements in memory
- When stable sort is required

3.10 Merge Sort Implementation

Interface

```
int mgsort(void *data, int size, int esize, int i, int k,
        int (*compare)(const void *key1, const void *key2))
```

- Parameters
  - data – Pointer to array of data to be sorted
  - size – Number of elements in data
  - esize – Size of each element
  - i – left index of range in data to be sorted; should always be 0
  - k – right index of range in data to be sorted; should always be size - 1
  - compare – Comparison function used to compare elements
- Returns
  - 0 if sorting successful, -1 otherwise

Implementation

```c
/*
 * mgsort.c
 */
#include <stdlib.h>
#include <string.h>

#include "sort.h"

static int merge(void *data, int esize, int i, int j, int k,
        int (*compare) (const void *key1, const void *key2))
{
    char *a = data,
          *m;

    int ipos,
        jpos,
        mpos;

    /* Initialize the counters used in merging. */
    ipos = i;
    jpos = j + 1;
    mpos = 0;

    /* Allocate storage for the merged elements. */
    if ((m = (char *)malloc(esize * ((k - i) + 1))) == NULL)
        return -1;

    /* Continue while either division has elements to merge. */
    while (ipos <= j || jpos <= k) {
        if (ipos > j) {

            /* The left division has no more elements to merge. */
            while (jpos <= k) {
                memcpy(&m[mpos * esize], &a[jpos * esize], esize);
                jpos++;
                mpos++;
            }

            continue;
        }
```

```
        else if (jpos > k) {
            /* The right division has no more elements to merge. */
            while (ipos <= j) {
                memcpy(&m[mpos * esize], &a[ipos * esize], esize);
                ipos++;
                mpos++;
            }

            continue;
        }

        /* Append the next ordered element to the merged elements. */
        if (compare(&a[ipos * esize], &a[jpos * esize]) < 0) {
            memcpy(&m[mpos * esize], &a[ipos * esize], esize);
            ipos++;
            mpos++;
        }
        else {
            memcpy(&m[mpos * esize], &a[jpos * esize], esize);
            jpos++;
            mpos++;
        }
    }

    /* Prepare to pass back the merged data. */
    memcpy(&a[i * esize], m, esize * ((k - i) + 1));

    /* Free the storage allocated for merging. */
    free(m);

    return 0;
}

int mgsort(void *data, int size, int esize, int i, int k,
        int (*compare)(const void *key1, const void *key2))
{
    int j;

    /* Stop the recursion when no more divisions can be made. */
    if (i < k) {
        /* Determine where to divide the elements. */
        j = (int)(((i + k - 1)) / 2);
```

```c

        /* Recursively sort the two divisions. */
        if (mgsort(data, size, esize, i, j, compare) < 0)
            return -1;

        if (mgsort(data, size, esize, j + 1, k, compare) < 0)
            return -1;

        /* Merge the two sorted divisions into a single sorted set. */
        if (merge(data, esize, i, j, k, compare) < 0)
            return -1;
    }

    return 0;
}
```

3.11 Counting Sort

Description

- Linear-time sort
- Counts how many times each element of a set occurs to determine how the set should be ordered
- Does not compare elements in the set
- Improves on O(n lg n) run-time upper bound of comparison sorts
- Requires elements in set to be integral type
- We must know largest value that can occur in the set so we can allocate space for the counts
- Overview
  - Start with unsorted set of integers (or objects that can be represented as integers)
  - Allocate array of counts initialized to zero; size of array = value of largest integer in unsorted set
  - Loop over unsorted set incrementing count of each integer
  - Place integers in sorted order into set using counts from array

Visualization

- Procedure:
  - Phase I – Count (steps 1a & 1b)
    - Allocate array of counts initialized to zero; size of array = value of largest integer in unsorted set
    - Loop over unsorted set incrementing count of each integer
  - Phase II – Adjust counts so they reflect last index of each value in sorted array (step 1c)
    - Loop over counts and increment each count by all previous counts
  - Phase III – Place sorted values into new set (steps 2a – 2f)
    - Loop over unsorted values in reverse order
    - For each unsorted value
      - Lookup count of that value in array of counts
      - Assign unsorted value into sorted array using the count as the index
      - Decrement count by 1

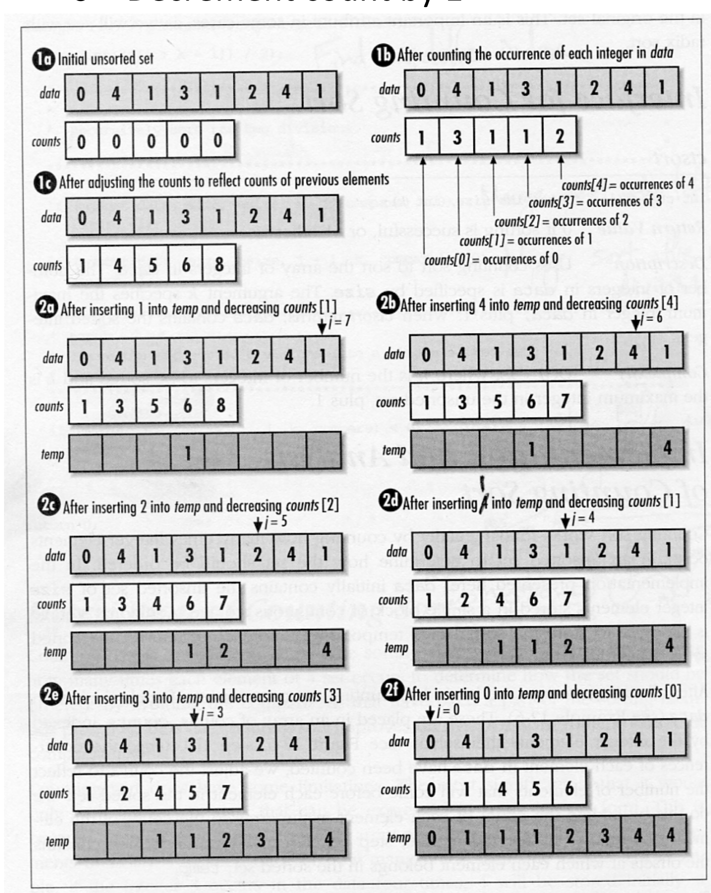


Figure 12-6. Sorting with counting sort

Time

- O(n + k)
  - n is the number of integers to be sorted
  - k is the maximum integer in the unsorted set plus 1

Space

- Twice the space of the original list plus space for the counts

Stability

- Stable

Data structures supported

- Array
- Singly-linked list
- Doubly-linked list

When to use?

- When sorting integral types that have a small maximum value

3.12 Counting Sort Implementation

Interface

```
int ctsort(int *data, int size, int k)
```

- Parameters
  - data – Pointer to array of ints to be sorted
  - size – Number of elements in data
  - k – Maximum value occurring in data
- Returns
  - 0 if sorting successful, -1 otherwise

Implementation

```
/*
 * ctsort.c
 */
#include <stdlib.h>
#include <string.h>

#include "sort.h"

int ctsort(int *data, int size, int k)
{
    int *counts,
        *temp;

    int i,
        j;

    /* Allocate storage for the counts. */
    if ((counts = (int *)malloc(k * sizeof(int))) == NULL)
        return -1;

    /* Allocate storage for the sorted elements. */
    if ((temp = (int *)malloc(size * sizeof(int))) == NULL)
        return -1;

    /* Initialize the counts. */
    for (i = 0; i < k; i++)
        counts[i] = 0;

    /* Count the occurrences of each element. */
    for (j = 0; j < size; j++)
        counts[data[j]] = counts[data[j]] + 1;

    /* Adjust each count to reflect the counts before it. */
    for (i = 1; i < k; i++)
        counts[i] = counts[i] + counts[i - 1];

    /* Use the counts to position each element where it belongs. */
    for (j = size - 1; j >= 0; j--) {
        temp[counts[data[j]] - 1] = data[j];
        counts[data[j]] = counts[data[j]] - 1;
```

```
    }

    /* Prepare to pass back the sorted data. */
    memcpy(data, temp, size * sizeof(int));

    /* Free the storage allocated for sorting. */
    free(counts);
    free(temp);

    return 0;
}
```

3.13 Radix Sort

Description

- Linear-time sort
- Sorts data in pieces called *digit*, one digit at a time, from the digit in least significant position to the most significant
- Example: Sorting radix-10 numbers
  - Unsorted: { 15, 12, 49, 16, 36, 40 }
  - After sorting least significant digit: { 40, 12, 15, 16, 36, 49 }
  - After sorting most significant digit: { 12, 15, 16, 36, 40 49}
- Digit sorting must use a stable sort
  - Important because once less significant digit order determined, more significant digit sorts must not cause reordering for values where more significant digits are the same
  - Example: Using unstable digit sort
    - Unsorted: { 15, 12 }
    - After sorting least significant digit: { 12, 15 }
    - After *unstable* sorting most significant digit : { 15, 12 }
- Uses counting sort to sort digits
  - Stable
  - We know largest integer any digit may be (radix – 1)
- Not limited to sorting data keyed by integers
  - Can sort any data that can be divided into integer pieces
  - Examples:
    - Strings (radix 2^8)
    - 64-bit integers (four-digit, radix 2^16 values)

Visualization

- Procedure:
  - For each digit from least to most significant
    - Apply count sort for the digit

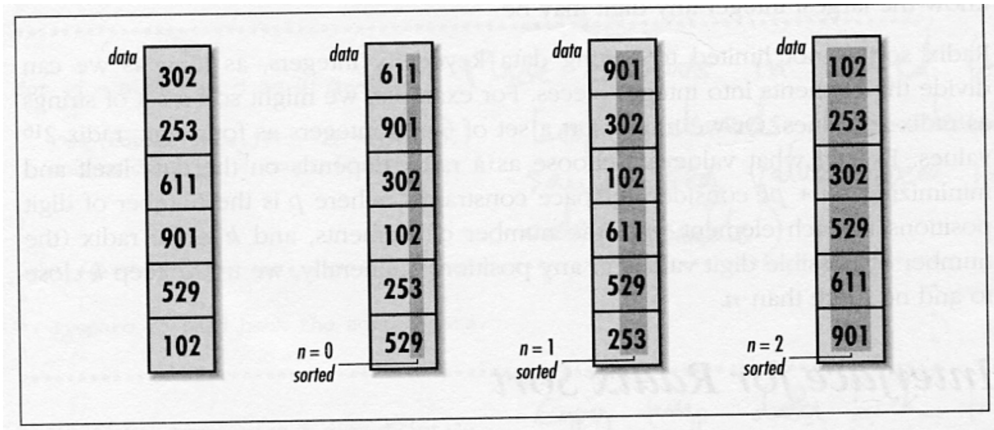


Figure 12-7. Sorting integers as radix-10 numbers with radix sort

Time

- O(pn + pk)
  - p is the number of digit positions in each element
  - n is the number of elements
  - k is the radix
  - Note: We try to keep k close to and no more than n

Space

- Twice the space of the original list plus space for the counts

Stability

- Stable

Data structures supported

- Array

When to use?

- When data can be split into integer pieces

3.14 Radix Sort Implementation

Interface

```
int rxsort(int *data, int size, int p, int k)
```

- Parameters
  - data – Pointer to array of ints to be sorted
  - size – Number of elements in data
  - p – Number of digit positions in each element
  - k – Radix
- Returns
  - 0 if sorting successful, -1 otherwise

Implementation

```
/*
 * rxsort.c
 */
#include <limits.h>
#include <math.h>
#include <stdlib.h>
#include <string.h>

#include "sort.h"

int rxsort(int *data, int size, int p, int k)
{
    int *counts,
        *temp;

    int index,
        pval,
        i,
        j,
        n;

    /* Allocate storage for the counts. */
    if ((counts = (int *)malloc(k * sizeof(int))) == NULL)
        return -1;

```

```c
    /* Allocate storage for the sorted elements. */
    if ((temp = (int *)malloc(size * sizeof(int))) == NULL)
        return -1;

    /* Sort from the least significant position to the most significant. */
    for (n = 0; n < p; n++) {
        /* Initialize the counts. */
        for (i = 0; i < k; i++)
            counts[i] = 0;

        /* Calculate the position value. */
        pval = (int)pow((double)k, (double)n);

        /* Count the occurrences of each digit value. */
        for (j = 0; j < size; j++) {
            index = (int)(data[j] / pval) % k;
            counts[index] = counts[index] + 1;
        }

        /* Adjust each count to reflect the counts before it. */
        for (i = 1; i < k; i++)
            counts[i] = counts[i] + counts[i - 1];

        /* Use the counts to position each element where it belongs. */
        for (j = size - 1; j >= 0; j--) {
            index = (int)(data[j] / pval) % k;
            temp[counts[index] - 1] = data[j];
            counts[index] = counts[index] - 1;
        }

        /* Prepare to pass back the data as sorted thus far. */
        memcpy(data, temp, size * sizeof(int));
    }

    /* Free the storage allocated for sorting. */
    free(counts);
    free(temp);

    return 0;
}
```

3.15 Searching Introduction

Overview

- Looking for a target value in a set

Data structure requirements

- Generally search requires
  - Random access
  - Data sorted
- Specialized searches exist for specific data structures
  - More on this later in class

3.16 Linear Search

Description

- Does not require set to be sorted
- Search from beginning of set to end
- O(n)
- Worst possible search
    - Only acceptable for very small sets

3.17 Binary Search

Description

- Requires set to be sorted
- Divide and conquer
- Procedure
  - Look at middle element
  - Return true if element matches target
  - Repeat search on lower half of elements if middle element > target
  - Repeat search on upper half of elements if middle element < target
  - Return false if no more elements left to search

Visualization

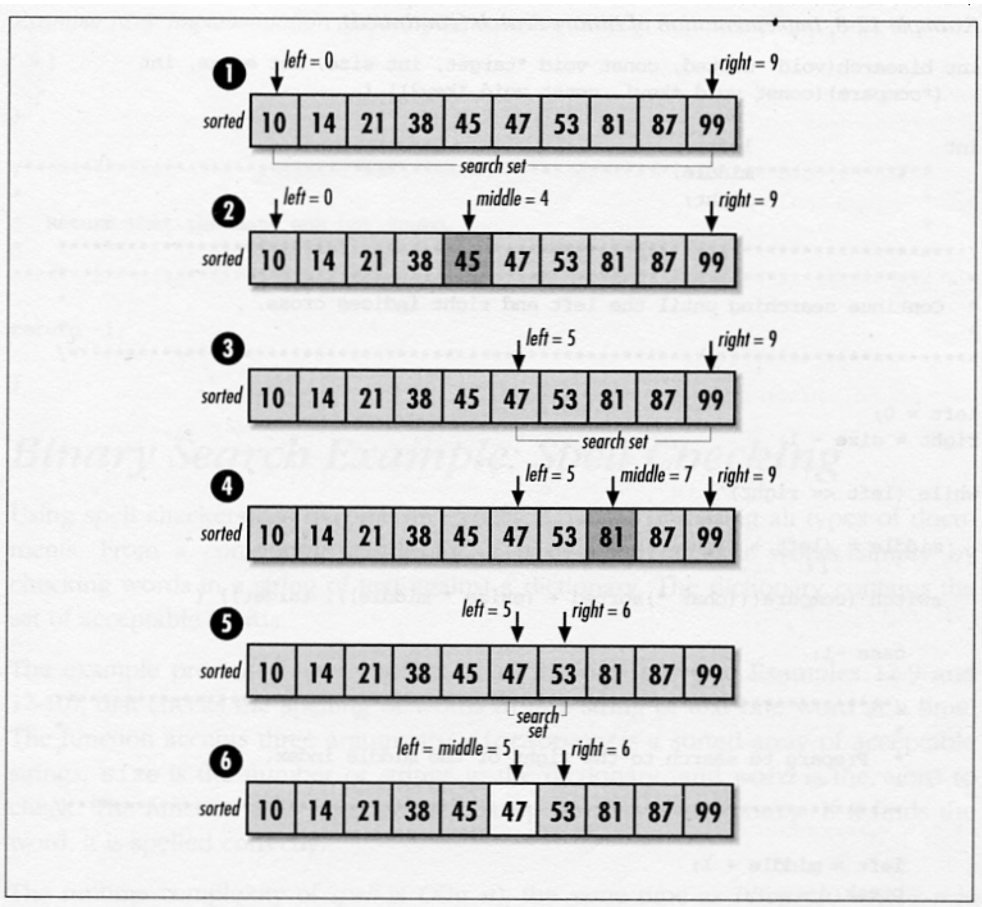


Figure 12-8. Searching for 47 using binary search

Time

- O(lg n)
  - Each division reduces problem size by ½

Space

- In-place

Data structures supported

- Array – requires random access

When to use?

- When fast searches are required

3.18 Binary Search Implementation

Interface

```
int bisearch(void *sorted, const void *target, int size, int esize,
        int (*compare)(const void *key1, const void *key2))
```

- Parameters
  - sorted – Pointer to array of elements to be sorted, will be sorted after call completes
  - target – Value being searched for
  - size – Number of elements in sorted array
  - esize – Size of each element in sorted array
  - compare – Comparison function used to compare elements during search
- Returns
  - index where target was found, -1 if target not found

Implementation

```
/*
 * bisearch.c
 */
#include <stdlib.h>
#include <string.h>

#include "search.h"

int bisearch(void *sorted, const void *target, int size, int esize,
        int (*compare)(const void *key1, const void *key2))
{
    int left,
        middle,
        right;

    /* Continue searching until the left and right indices cross. */
    left = 0;
    right = size - 1;

    while (left <= right) {
        middle = (left + right) / 2;

        switch (compare(((char *)sorted + (esize * middle)), target)) {
        case -1:
            /* Prepare to search to the right of the middle index. */
            left = middle + 1;
            break;

        case 1:
            /* Prepare to search to the left of the middle index. */
            right = middle - 1;
            break;

        case 0:
            /* Return the exact index where the data has been found. */
            return middle;
        }
    }

    /* Return that the data was not found. */
```

```
    return -1;
}
```