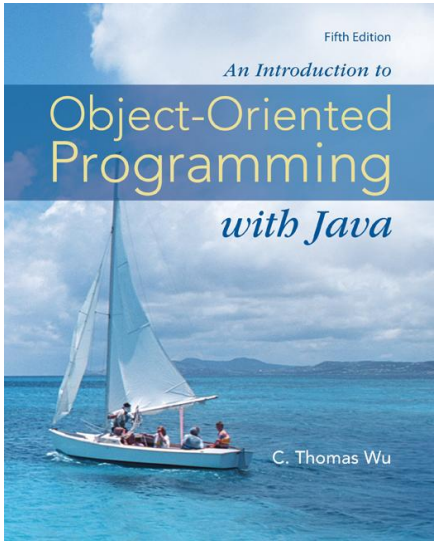


Chapter 11

Sorting and Searching

Objectives

After you have read and studied this chapter, you should be able to

- Perform linear and binary search algorithms on small arrays.
- Determine whether a linear or binary search is more effective for a given situation.
- Perform selection and bubble sort algorithms.
- Describe the heapsort algorithm and show how its performance is superior to the other two algorithms.
- Apply basic sorting algorithms to sort an array of objects.

Searching

- When we maintain a collection of data, one of the operations we need is a search routine to locate desired data quickly.
- Here's the problem statement:

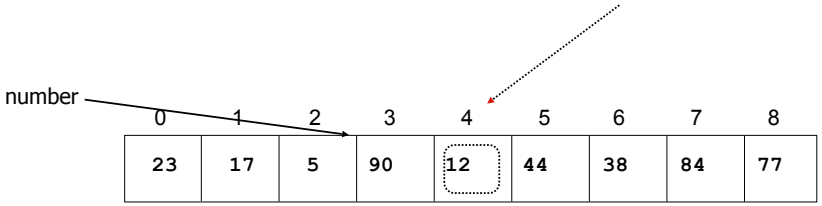
  Given a value X, return the index of X in the array, if such X exists. Otherwise, return NOT_FOUND (-1). We assume there are no duplicate entries in the array.

- We will count the number of comparisons the algorithms make to analyze their performance.
  - The ideal searching algorithm will make the least possible number of comparisons to locate the desired data.
  - Two separate performance analyses are normally done, one for successful search and another for unsuccessful search.

Search Result

Unsuccessful Search: `search( 45 )` → **NOT_FOUND**

Successful Search: `search( 12 )` → **4**



Linear Search

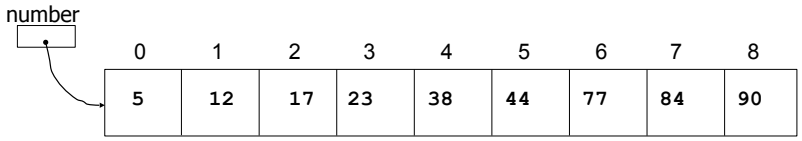
- Search the array from the first to the last position in linear progression.

```
public int linearSearch ( int[] number, int searchValue ) {
    int loc = 0;
    while (loc < number.length    &&  number[loc] != searchValue) {
        loc++;
    }
    if (loc == number.length) { //Not found
        return NOT_FOUND;
    } else {
        return loc;   //Found, return the position
    }
}
```


Linear Search Performance

- We analyze the successful and unsuccessful searches separately.
- We count how many times the search value is compared against the array elements.
- Successful Search
  - Best Case – 1 comparison
  - Worst Case – N comparisons (N – array size)
- Unsuccessful Search
  - Best Case =
    Worst Case – N comparisons

Binary Search



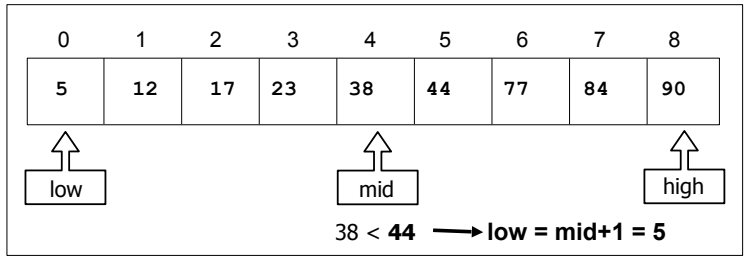
- If the array is sorted, then we can apply the binary search technique.
- The basic idea is straightforward. First search the value in the middle position. If X is less than this value, then search the middle of the left half next. If X is greater than this value, then search the middle of the right half next. Continue in this manner.

Sequence of Successful Search - 1

search(44)

	low	high	mid
#1	0	8	4

$$mid = \left\lfloor \frac{low + high}{2} \right\rfloor$$

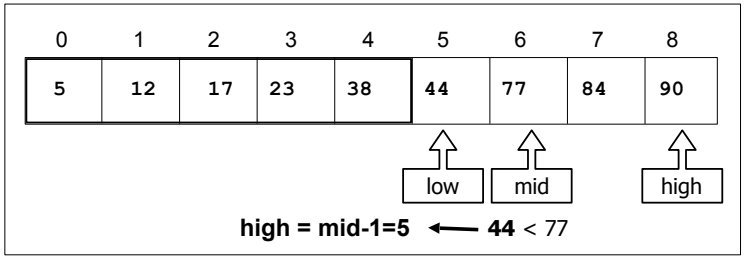


Sequence of Successful Search - 2

	low	high	mid
#1	0	8	4
#2	5	8	6

search(44)

$$mid = \left\lfloor \frac{low + high}{2} \right\rfloor$$

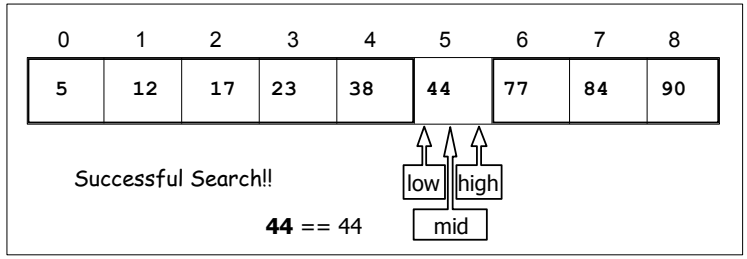


Sequence of Successful Search - 3

	low	high	mid
#1	0	8	4
#2	5	8	6
#3	5	5	5

search(44)

$$mid = \left\lfloor \frac{low + high}{2} \right\rfloor$$

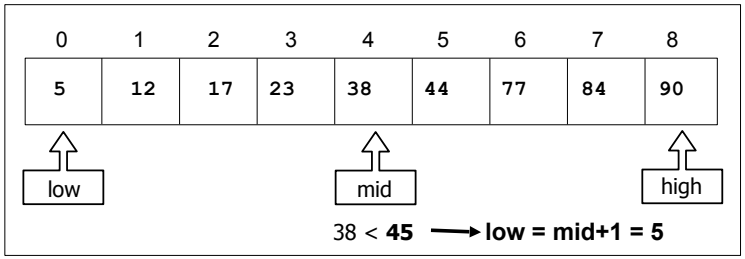


Sequence of Unsuccessful Search - 1

	low	high	mid
#1	0	8	4

search(45)

$$mid = \left\lfloor \frac{low + high}{2} \right\rfloor$$

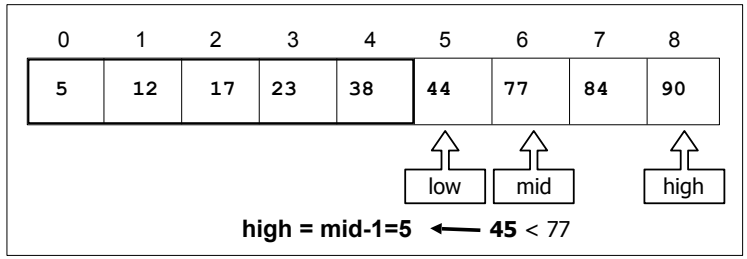


Sequence of Unsuccessful Search - 2

	low	high	mid
#1	0	8	4
#2	5	8	6

search(45)

$$mid = \left\lfloor \frac{low + high}{2} \right\rfloor$$

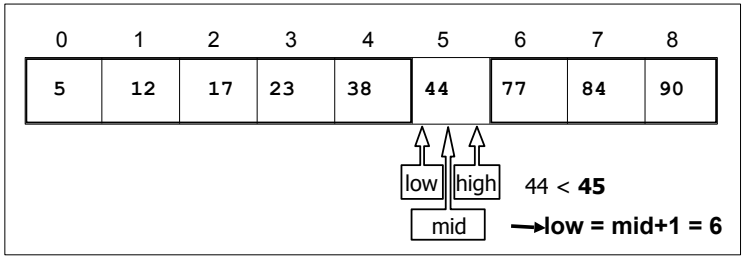


Sequence of Unsuccessful Search - 3

	low	high	mid
#1	0	8	4
#2	5	8	6
#3	5	5	5

search(45)

$$mid = \left\lfloor \frac{low + high}{2} \right\rfloor$$

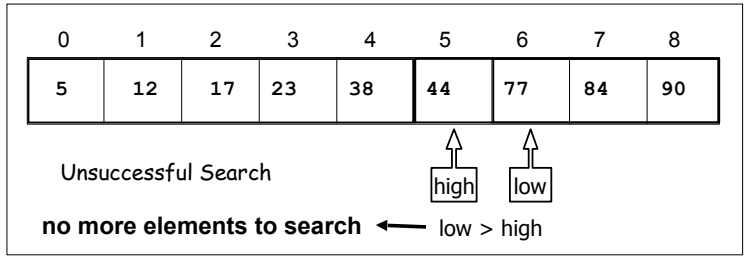


Sequence of Unsuccessful Search - 4

	low	high	mid
#1	0	8	4
#2	5	8	6
#3	5	5	5
#4	6	5	

search(45)

$$mid = \left\lfloor \frac{low + high}{2} \right\rfloor$$



Binary Search Routine

```
public int binarySearch (int[] number, int searchValue) {

    int     low     = 0,
            high  = number.length - 1,
            mid     = (low + high) / 2;
    while (low <= high && number[mid] != searchValue) {
        if (number[mid] < searchValue) {
            low = mid + 1;
        } else  { //number[mid] > searchValue
            high = mid - 1;
        }
        mid = (low + high) / 2;  //integer division will truncate
    }
    if (low > high) {
        mid = NOT_FOUND;
    }
    return mid;
}
```


Binary Search Performance

- Successful Search
  - Best Case – 1 comparison
  - Worst Case – $\log_2 N$ comparisons
- Unsuccessful Search
  - Best Case =
    Worst Case – $\log_2 N$ comparisons
- Since the portion of an array to search is cut into half after every comparison, we compute how many times the array can be divided into halves.
- After K comparisons, there will be $N/2^K$ elements in the list. We solve for K when $N/2^K = 1$, deriving $K = \log_2 N$.

Comparing N and $\log_2 N$ Performance

Array Size	Linear – N	Binary – $\log_2 N$
10	10	4
50	50	6
100	100	7
500	500	9
1000	1000	10
2000	2000	11
3000	3000	12
4000	4000	12
5000	5000	13
6000	6000	13
7000	7000	13
8000	8000	13
9000	9000	14
10000	10000	14

Sorting

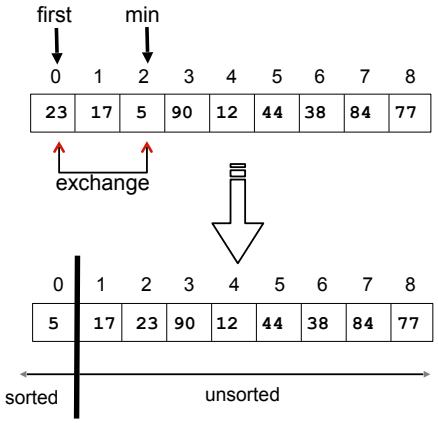
- When we maintain a collection of data, many applications call for rearranging the data in certain order, e.g. arranging Person information in ascending order of age.
- Here's the problem statement:

  Given an array of N integer values, arrange the values into ascending order.

- We will count the number of comparisons the algorithms make to analyze their performance.
  - The ideal sorting algorithm will make the least possible number of comparisons to arrange data in a designated order.
- We will compare different sorting algorithms by analyzing their worst-case performance.

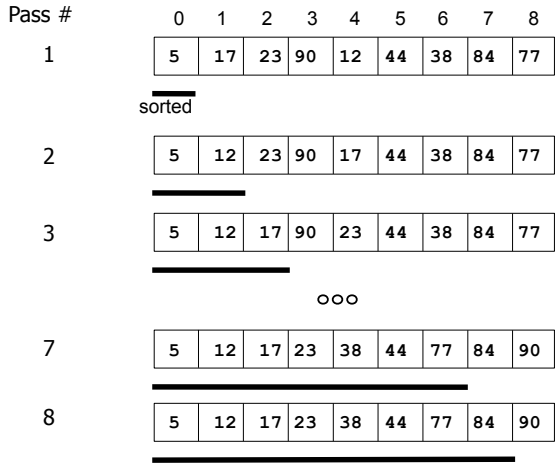
Selection Sort

1. Find the smallest element in the list.
2. Exchange the element in the first position and the smallest element. Now the smallest element is in the first position.
3. Repeat Step 1 and 2 with the list having one less element (i.e., the smallest element is discarded from further processing).



This is the result of one pass.

Selection Sort Passes



Result AFTER one pass is completed.

Selection Sort Routine

```
public void selectionSort( int[] number )
{
    int startIndex, minIndex, length, temp;
    length = number.length;

    for (startIndex = 0; startIndex <= length-2; startIndex++){
        //each iteration of the for loop is one pass
        minIndex = startIndex;

        //find the smallest in this pass at position minIndex
        for (i = startIndex+1; i <= length-1; i++) {
           if (number[i] < number[minIndex]) minIndex = i;
        }
        //exchange number[startIndex] and number[minIndex]
        temp               = number[startIndex];
        number[startIndex] = number[minIndex];
        number[minIndex]   = temp;
    }
}
```


Selection Sort Performance

- We derive the total number of comparisons by counting the number of times the inner loop is executed.
- For each execution of the outer loop, the inner loop is executed length – start times.

Start	Number of Comparisons (Length – Start)
0	length
1	length – 1
2	length – 2
. . .	. . .
length – 2	2

$$N + (N-1) + (N-2) + \cdots + 2$$

The variable length is the size of the array. Replacing length with N, the array size, the sum is derived as…

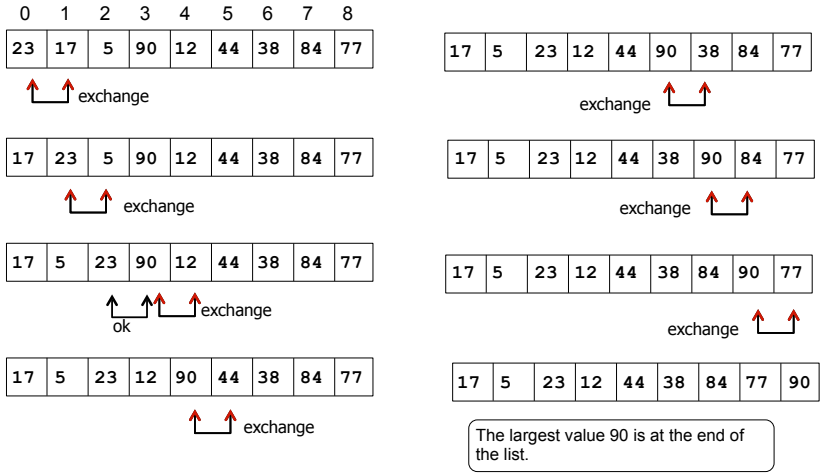
$$= \sum_{i=2}^{N} i = \left(\sum_{i=1}^{N} i\right) - 1 = \frac{N(N+1)}{2} - 1$$

$$= \frac{N^2 + N - 2}{2} \cong N^2$$

Bubble Sort

- With the selection sort, we make one exchange at the end of one pass.
- The bubble sort improves the performance by making more than one exchange during its pass.
- By making multiple exchanges, we will be able to move more elements toward their correct positions using the same number of comparisons as the selection sort makes.
- The key idea of the bubble sort is to make pairwise comparisons and exchange the positions of the pair if they are out of order.

One Pass of Bubble Sort



Bubble Sort Routine

```
public void bubbleSort(int[] number) {

    int      temp, bottom, i;
    boolean  exchanged = true;
    bottom = number.length - 2;

    while (exchanged)  {

        exchanged = false;
        for (i = 0; i <= bottom; i++) {

            if (number[i] > number[i+1]) {
                temp        = number[i];    //exchange
                number[i]   = number[i+1];
                number[i+1] = temp;
                exchanged = true; //exchange is made

            }
        }
        bottom--;
    }
}
```


Bubble Sort Performance

- In the worst case, the outer while loop is executed N-1 times for carrying out N-1 passes.
- For each execution of the outer loop, the inner loop is executed bottom+1 times. The number of comparisons in each successive pass is N-1, N-2, … , 1. Summing these will result in the total number of comparisons.
- So the performances of the bubble sort and the selection sort are approximately equivalent. However, on the average, the bubble sort performs much better than the selection sort because it can finish the sorting without doing all N-1 passes.

$$(N-1)+(N-2)+\cdots+1$$

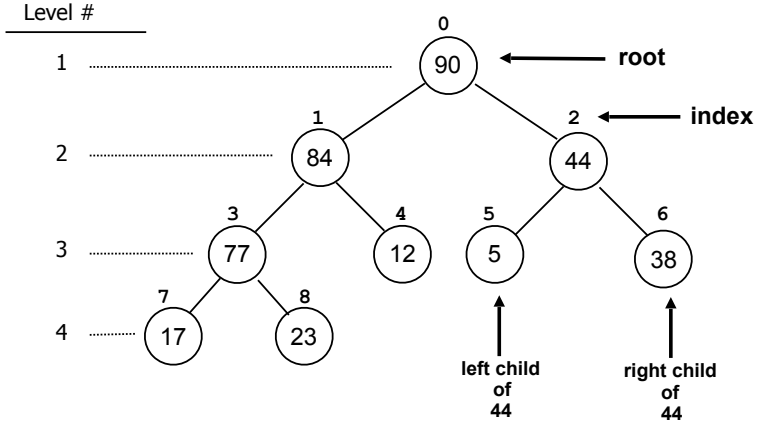
$$=\sum_{i=1}^{N-1} i = \frac{N(N-1)}{2}$$

$$=\frac{N^2-N}{2} \cong N^2$$

Heapsort

- Selection and bubble sorts are two fundamental sorting algorithms that take approximately N^2 comparisons to sort an array of N elements.
- One sorting algorithm that improves the performance to approximately $1.5N\log_2 N$ is called heapsort.
- The heapsort algorithm uses a special data structure called a heap.
- A heap structure can be implemented very effectively by using an array.

Sample Heap

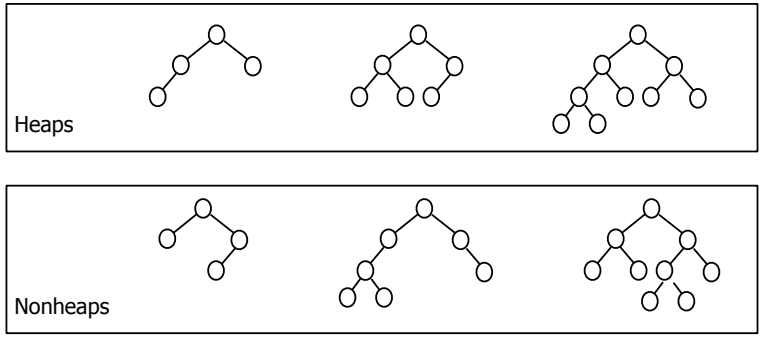


Heap Constraints

- A heap must satisfy the following two constraints:
  - Structural Constraint: Nodes in a heap with N nodes must occupy the positions numbered 0, 1, ..., N-1.
  - Value Relationship Constraint: A value stored in a node must be larger than the maximum of the values stored in its left and right children.

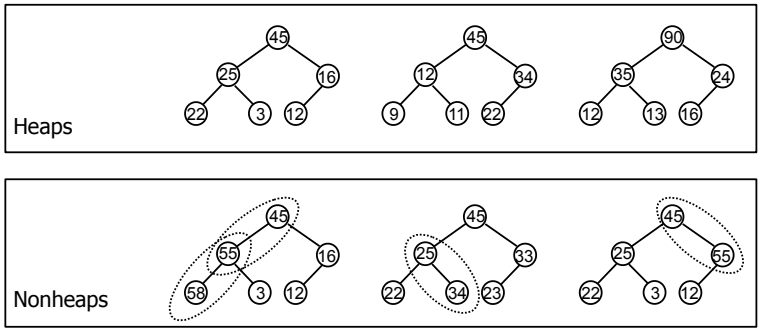
Structural Constraints

- Sample heaps and nonheaps that violate the structural constraints:



Value Relationship Constraints

- Sample heaps and nonheaps that violate the value relationship constraints:

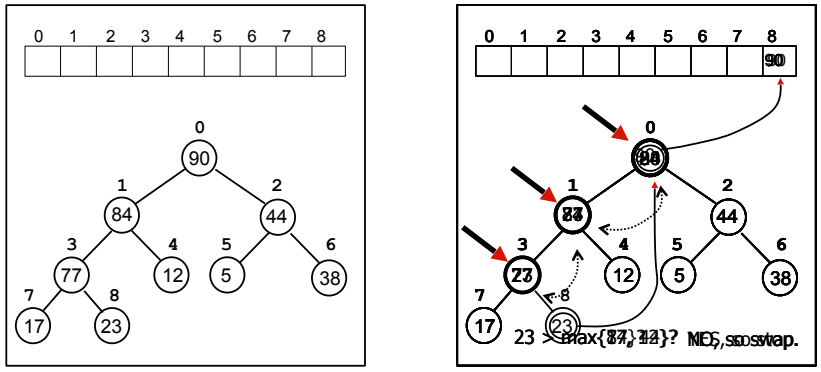


Heapsort Algorithm

- How can we use the heap structure to sort N elements?
- Heapsort is carried out in two phases:
  - Construction Phase: Construct a heap with given N elements.
  - Extraction Phase: Pull out the value in the root successively, creating a new heap with one less element after each extraction.

Extraction Phase

- After each extraction, a heap must be rebuilt with one less element. One sample extraction step:

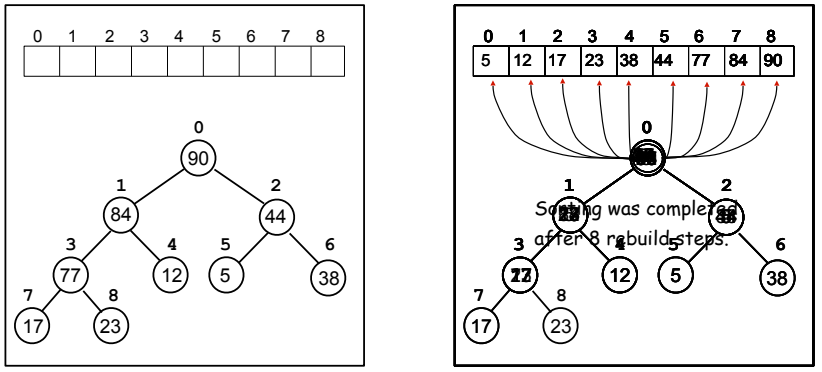


Before

After

Rebuild Steps

- A sequence of rebuild steps to sort the list.

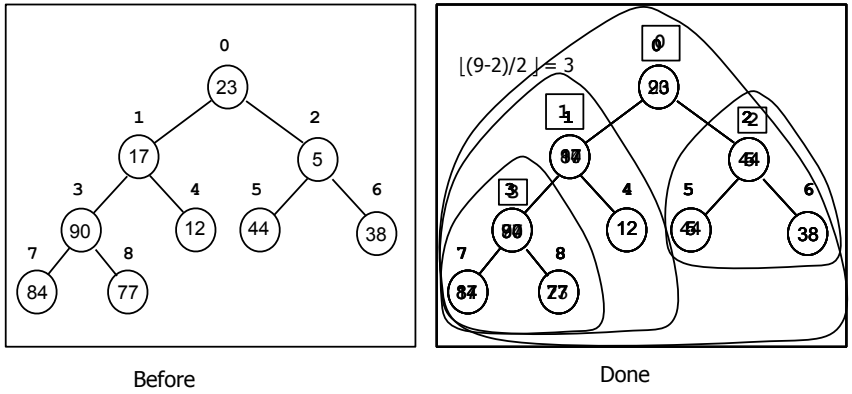


Before

Done

Construction Phase

- Rebuild steps are applied from position ⌊(N-2)/2 ⌋ to position 0.

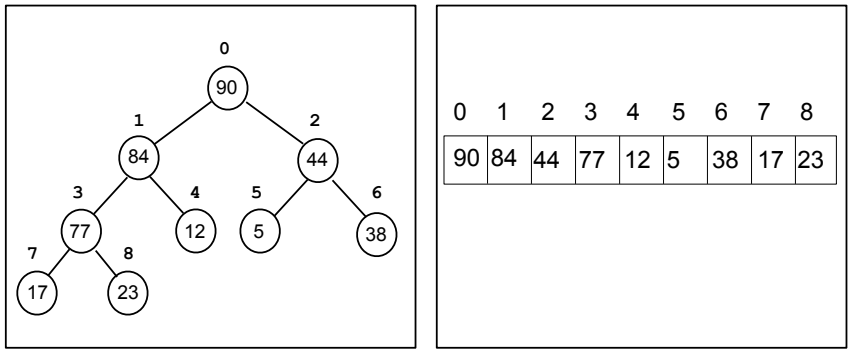


Before

Done

Heap Implementation

- We need to implement the abstract data structure heap into a concrete data structure.



Abstract Heap Structure

Concrete Implementation

The construct Method

```
private void construct( ) {
    for (int i = (heap.length-2) / 2; i >= 0; i--) {
        current = i;
        done    = false;
        while ( !done ) {
           if ( current has no children ) {
              done = true;
           } else {
              if (current node < larger child) {
                 swap the two nodes;
                 set current points to the larger child;
              } else {
                 done = true;
              }
           }
        }
    }
}
```


The extract Method

```
private void extract( ) {
    for (int size = heap.length-1; size >= 0; size--) {
        remove the root node data;
        move the last node to the root;

        done    = false; //rebuild the heap with one less element
        while (!done) {
           if (current has no children) {
              done = true;
           } else {
              if (current node < larger child) {
                 swap the two nodes;
                 set current points to the larger child;
              } else {
                 done = true;
              }
           }
        }
    }
}
```


Heapsort Performance

- The total number of comparisons in the extraction phase is (N-1)×K where K is the depth of a heap.
- We solve for K using the fact that the heap must satisfy the structural constraint:

$$\sum_{i=1}^{K} 2^{i-1} = 2^K - 1$$ ← **total # nodes in a heap with depth K**

$$2^{K-1} - 1 < N \le 2^K - 1$$ ← **this holds because of the structural constraint**

$$K = \lceil \log_2 (N+1) \rceil$$

Therefore,

$$(N-1) \times K \cong N \log_2 N$$

Heapsort Performance (cont'd)

- There are N/2 rebuild steps in the construction phase.
- Each rebuild step will take no more than K comparisons.
- The total number of comparisons in the construction phase is approximately

$$\frac{N}{2} \log_2 N$$

Therefore, the total number of comparisons for both phases is

$$\frac{N}{2} \log_2 N + N \log_2 N = \boxed{1.5 N \log_2 N}$$

Problem Statement

Add the sorting capability to the AddressBook class from Chapter 10. The new AddressBook class will include a method that sort Person objects in alphabetical order of their names or in ascending order of their ages.

Overall Plan

- Instead of going through the development steps, we will present three different implementations.
- The three versions are named AddressBookVer1, AddressBookVer2, and AddressBookVer3.
- These classes will implement the AddressBook interface.

The AddressBook Interface

```
interface AddressBook {
   public void add(Person newPerson);
   public boolean delete(String searchName);
   public Person search(String searchName);
   public Person[ ] sort (int attribute);
}
```


AddressBookVer1 Design

- We use an array of Person objects
- We provide our own sorting method in the AddressBookVer1 class
- We define the Person class so we can compare Person objects by name or by age

Comparing Person Objects

- First, we need to determine how to compare Person objects
- The following does not make sense:

```
Person p1 = …;
Person p2 = …;

    if ( p1 < p2 ) { … }
```


Modify the Person Class

- Modify the Person class to include a variable that tells which attribute to compare.

```
class Person {
    …
    public static final int NAME = 0;
    public static final int AGE  = 1;
    public static int compareAttribute = NAME;
    …
    public static void setCompareAttribute(int attribute) {
        compareAttribute = attribute;
    }
    …
}
```


The compareTo Method

- To compare Person objects, first set the comparison attribute and then call the compareTo Method.

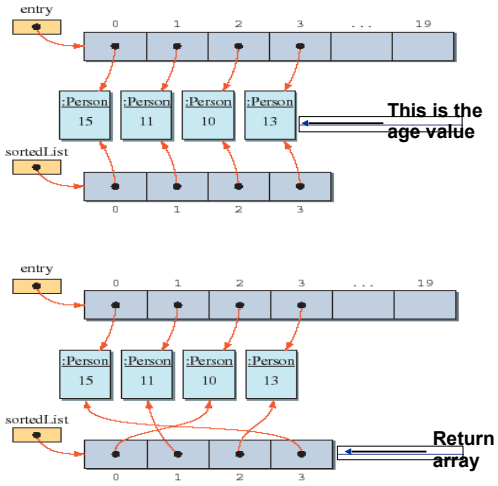
```
Person.setCompareAttribute
          (Person.NAME);

int compResult = p1.compareTo(p2);

if (compResult < 0) {
    //p1 "less than" p2

} else if (compResult == 0) {
    //p1 "equals" pw

} else {
    //p2 "greater than" p2
}
```

```
public int compareTo(Person p) {

   int compResult;
   if ( comparisonAttribute == AGE ) {
      int p2age = p.getAge();
      if ( this.age < p2age ) {
           compResult = LESS;
      }
      …
   } else { //compare Name
      String p2Name = p.getName();
      compResult = this.name.
                compareTo(p2Name);
   }
   return compResult;
}
```


The sort Method

```
public Person[ ] sort (int attribute) {

   Person[ ] sortedList = new Person[count];
   Person p1, p2, temp;

   //copy references to sortedList
   for (int i = 0; i < count; i++) {
      sortedList[i] = entry[i];
   }

   //Set the comparison attribute
   Person.setCompareAttribute(attribute);

   //begin the bubble sort on sortedList
   …

   }
   return sortedList;
}
```

The Use of sortedList in the sort Method



AddressBookVer1 Code

Program source file is too big to list here. From now on, we ask you to view the source files using your Java IDE.

Directory: Chapter11/

Source Files: AddressBookVer1.java

AddressBookVer2 Design

- We use the java.util.Arrays class to sort an array of Person objects
- The Person class does not include any comparison methods. Instead, we implement the Comparator interface. We can define the implementation class so we can compare Person objects using any combination of their attributes
  - For example, we can define a comparator that compares Person objects by using both name and age

The AgeComparator Class

- This class compares the age attributes of Person objects

```
class AgeComparator implements Comparator<Person> {
   private final int LESS = -1;
   private final int EQUAL = 0;
   private final int MORE = 1;

   public int compare(Person p1, Person p2) {
      int comparisonResult;
      int plage = p1.getAge( );
      int p2age = p2.getAge( );
      if (plage < p2age) {
           comparisonResult = LESS;
      } else if (plage == p2age) {
           comparisonResult = EQUAL;
      } else {
           assert plage > p2age;
           comparisonResult = MORE;
      }
      return comparisonResult;
   }
}
```

The NameComparator Class

- This class compares the age attributes of Person objects
- Because the name attribute is a string we can use the compareTo method of the String class

```
class NameComparator implements Comparator<Person> {

   public int compare(Person p1, Person p2) {

      String p1name = p1.getName( );
      String p2name = p2.getName( );

      return p1name.compareTo(p2name);
   }
}
```


The sort Method

- We use the sort method of the java.util.Arrays class

```
public Person[ ] sort ( int attribute ) {
   if (!(attribute == Person.NAME ||
         attribute == Person.AGE) ) {
      throw new IllegalArgumentException( );
   }

   Person[ ] sortedList = new Person[ count ];

   //copy references to sortedList
   for (int i = 0; i < count; i++) {
      sortedList[i] = entry[i];
   }

   Arrays.sort(sortedList, getComparator(attribute));

   return sortedList;
}
```


AddressBookVer2 Code

Program source file is too big to list here. From now on, we ask you to view the source files using your Java IDE.

Directory: Chapter11/

Source Files: AddressBookVer2.java

AddressBookVer3 Design

- In the previous two versions, we used an array data structure to maintain a collection of Person objects
- In the third version, we don't use an array at all.
- Instead, we use a Map from the Java Collection Framework to maintain a collection of Person objects.
- We use the person's name as the key of a Map entry and the person object as the value of a Map entry.

The sort Method

- We retrieve a collection of values in the map, converts it to an array, and use the sort method of the java.util.Arrays class to sort this array.

```
public Person[ ] sort ( int attribute ) {

   if (!(attribute == Person.NAME ||
         attribute == Person.AGE) ) {
      throw new IllegalArgumentException( );
   }

   Person[ ] sortedList = new Person[entry.size()];

   entry.values().toArray(sortedList);

   Arrays.sort(sortedList, getComparator(attribute));

   return sortedList;
}
```


AddressBookVer3 Code

Program source file is too big to list here. From now on, we ask you to view the source files using your Java IDE.

Directory: Chapter11/

Source Files: AddressBookVer3.java