Analysis of Algorithm

Mostafa M. Aref

University of Bridgeport
Department of Computer Science & Engineering
Analysis of Algorithms

- Data structures is concerned with the representation and manipulation of data.
- All programs manipulate data.
- So, all programs represent data in some way.
- Data manipulation requires an algorithm.
- Algorithm design methods needed to develop programs that do the data manipulation.
- The study of data structures and algorithms is fundamental to Computer Science.

Sorting

- Rearrange a[0], a[1], ..., a[n-1] into ascending order. When done, a[0] <= a[1] <= ... <= a[n-1]
- 8, 6, 9, 4, 3 => 3, 4, 6, 8, 9
- Sort Methods: Insertion Sort, Bubble Sort, Selection Sort, Count Sort, Shaker Sort, Shell Sort, Heap Sort, Merge Sort, Quick Sort
- Insert An Element:
  - Given a sorted list/sequence, insert a new element
  - Given 3, 6, 9, 14, Insert 5, Result 3, 5, 6, 9, 14
  - 3, 6, 9, 14 insert 5
  - Compare new element (5) and last one (14)
  - Shift 14 right to get 3, 6, 9, , 14
  - Shift 9 right to get 3, 6, , 9, 14
  - Shift 6 right to get 3, , 6, 9, 14
  - Insert 5 to get 3, 5, 6, 9, 14

```c
// insert t into a[0:i-1]
int j;
for (j = i - 1; j >= 0 && t < a[j]; j--)
  a[j + 1] = a[j];
a[j + 1] = t;
```
Insertion Sort

- Start with a sequence of size 1
- Repeatedly insert remaining elements
- Sort 7, 3, 5, 6, 1
- Start with 7 and insert 3 => 3, 7, Insert 5 => 3, 5, 7
- Insert 6 => 3, 5, 6, 7, Insert 1 => 1, 3, 5, 6, 7

```java
for (int i = 1; i < a.length; i++)
{// insert a[i] into a[0:i-1]
    int t = a[i];
    int j;
    for (j = i - 1; j >= 0 && t < a[j]; j--)
        a[j + 1] = a[j];
    a[j + 1] = t; }
```

Complexity

- Space/Memory
- Time
  - Count a particular operation
  - Count number of steps
  - Asymptotic complexity
- Comparison Count
  - Pick an instance characteristic … n, n = a.length for insertion sort
  - Determine count as a function of this instance characteristic.
  - Worst case count = maximum count
    ```java
    for (int i = 1; i < n; i++)
        for (j = i - 1; j >= 0 && t < a[j]; j--)
            a[j + 1] = a[j];
    total compares = 1 + 2 + 3 + … + (n-1) = (n-1)n/2
    ```
  - Best case count = minimum count
  - Average count

- Complexity (continued)

  - For insertion sort:
    ```java
    for (int i = 1; i < a.length; i++)
        for (j = i - 1; j >= 0 && t < a[j]; j--)
            a[j + 1] = a[j];
    total compares = 1 + 2 + 3 + … + (n-1) = (n-1)n/2
    ```
  - Complexity (continued):
Step Count

- A step is an amount of computing that does not depend on the instance characteristic n
- 10 adds, 100 subtracts, 1000 multiplies, can all be counted as a single step
- n adds cannot be counted as 1 step

<table>
<thead>
<tr>
<th></th>
<th>s/e</th>
<th>step</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (int i = 1; i &lt; a.length; i++)</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>{ // insert a[i] into a[0:i-1]</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>int t = a[i];</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>int j;</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>for (j = i - 1; j &gt;= 0 &amp;&amp; t &lt; a[j]; j--)</td>
<td>1</td>
<td>i + 1</td>
</tr>
<tr>
<td>a[j + 1] = a[j];</td>
<td>1</td>
<td>i</td>
</tr>
<tr>
<td>a[j + 1] = t</td>
<td></td>
<td></td>
</tr>
<tr>
<td>}</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

- s/e isn’t always 0 or 1
- x = MyMath.sum(a, n); where n is the instance characteristic has a s/e count of n
- step count for for (int i = 1; i < a.length; i++) is n
- step count for body of for loop is
  \[2(1+2+3+\ldots+n-1) + 3(n-1) + 1= (n-1)n + 3(n-1) = (n-1)(n+3) + 1\]
- Asymptotic Complexity of Insertion Sort O(n²)
- Time or number of operations does not exceed c. n² on any input of size n
- Actually, the worst-case time is Theta(n²) and the best-case is Theta(n)
- So, the worst-case time is expected to quadruple each time n is doubled
- Is O(n²) too much time?
- Is the algorithm practical?
Big Oh, Theta, and Omega notations

- $\Theta(g(n)) = \{f(n):$ there exist positive constants $c_1, c_2$ and $n_0$ such that $0 \leq c_1g(n) \leq f(n) \leq c_2g(n)$ for all $n \geq n_0\}$
- $O(g(n)) = \{f(n):$ there exist positive constant $c$ and $n_0$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0\}$
- $\Omega(g(n)) = \{f(n):$ there exist positive constants $c$ and $n_0$ such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0\}$
- $o(g(n)) = \{f(n):$ for any positive constant $c > 0$ there exists a constant $n_0 > 0$ such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0\}$
- $\omega(g(n)) = \{f(n):$ for any positive constant $c > 0$ there exists a constant $n_0 > 0$ such that $0 \leq cg(n) < f(n)$ for all $n \geq n_0\}$

**Practical Complexities**

- $10^9$ instructions/second

<table>
<thead>
<tr>
<th>$n$</th>
<th>$n$</th>
<th>$n \log n$</th>
<th>$n^2$</th>
<th>$n^3$</th>
<th>$n^4$</th>
<th>$n^{10}$</th>
<th>$2^n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^3$</td>
<td>1 mic</td>
<td>10 mic</td>
<td>1 mic</td>
<td>1 sec</td>
<td>17 min</td>
<td>$3.2 \times 10^{13}$ years</td>
<td>$3.2 \times 10^{283}$ years</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$10^4$</td>
<td>10 mic</td>
<td>130 mic</td>
<td>100 milli</td>
<td>17 min</td>
<td>116 days</td>
<td>???</td>
<td>???</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$10^6$</td>
<td>1 milli</td>
<td>20 milli</td>
<td>17 min</td>
<td>32 years</td>
<td>$3 \times 10^7$ years</td>
<td>???</td>
<td>???</td>
</tr>
</tbody>
</table>

- Faster Computer Vs Better Algorithm
- Algorithmic improvement more useful than hardware improvement. E.g. $2n$ to $n^3$
Performance Measurement

- Paper and pencil: Don’t need a working computer program or even a computer.

- **Some Uses Of Performance Analysis**
  - determine practicality of algorithm
  - predict run time on large instance
  - compare 2 algorithms that have different asymptotic complexity e.g., \( O(n) \) and \( O(n^2) \)

- **Limitations of Analysis**
  - Doesn’t account for constant factors, but constant factor may dominate \( 1000n \) vs \( n^2 \) and we are interested only in \( n < 1000 \)
  - Modern computers have a hierarchical memory organization with different access time for memory at different levels of the hierarchy.
  - Memory Hierarchy

- **Measure actual time on an actual computer. What do we need?**
  - programming language, working program
  - Computer, compiler and options to use `javac -o`
  - data to use for measurement, worst-case data, best-case data, average case data, timing mechanism --- clock
Timing In Java

```java
long startTime = System.currentTimeMillis();
// gives time in milliseconds since 1/1/1970 GMT
// code to be timed comes here
long elapsedTime = System.currentTimeMillis() - startTime;
```

- **Short Comings**
  - Clock accuracy assume 100 milliseconds
  - Repeat work many times to bring total time to be >= 1 second

- **Accurate Timing**

```java
long startTime = System.currentTimeMillis();
long counter;
do {
    counter++;
    doSomething();
} while (System.currentTimeMillis() - startTime < 1000)
long elapsedTime = (System.currentTimeMillis() - startTime) / counter;
```

- Now accuracy is 10%.
- first reading may be just about to change to startTime + 100
- second reading may have just changed to finishTime
- so finishTime - startTime is off by 100ms
- first reading may have just changed to startTime
- second reading may be about to change to finishTime + 100
- so finishTime - startTime is off by 100ms
- Examining remaining cases, we get
- trueElapsedTime = finishTime - startTime +/- 100ms
- To ensure 10% accuracy, require
- elapsedTime = finishTime – startTime >= 1sec
What May go Wrong?

```java
long startTime = System.currentTimeMillis();
long counter;
do {
    counter++;
    // put code to initialize a[] here
    InsertionSort.insertionSort(a);
} while (System.currentTimeMillis() - startTime < 1000)
long elapsedTime = (System.currentTimeMillis() - startTime) / counter;

- Time Shared System
- Bad Way To Time
```

Data Structures

- Data object: Set or collection of instances
  - integer = {0, +1, -1, +2, -2, +3, -3, …}
  - daysOfWeek = {S,M,T,W,Th,F,Sa}
- Instances may or may not be related
  - myDataObject = {apple, chair, 2, 5.2, red, green, Jack}
- Relationships that exist among instances and elements that comprise an instance. Among instances of integer
  - 369 < 370 or 280 + 4 = 284
- Among elements that comprise an instance 369
  - 3 is more significant than 6
  - 3 is immediately to the left of 6
  - 9 is immediately to the right of 6
- The relationships are usually specified by specifying operations on one or more instances. add, subtract, predecessor, multiply
Linear (or Ordered) Lists

- Instances are of the form: \((e_0, e_1, e_2, \ldots, e_{n-1})\), where \(e_i\) denotes a list element \(n \geq 0\) is finite list size is \(n\)

- \(L = (e_0, e_1, e_2, e_3, \ldots, e_{n-1})\) relationships
  - \(e_0\) is the zero’th (or front) element
  - \(e_{n-1}\) is the last element
  - \(e_i\) immediately precedes \(e_{i+1}\)

- Linear List Examples
  - Students in CS502 = (Jack, Jill, Abe, Henry, Mary, \ldots, Judy)
  - Quizes in CS502 = (Quiz1, Quiz2, Quiz3)
  - Days of Week = (S, M, T, W, Th, F, Sa)
  - Months = (Jan, Feb, Mar, Apr, \ldots, Nov, Dec)

- Linear List Operations
  - \(\text{size}(\ ): \) determine list size \(L = (a,b,c,d,e)\) \(\text{size} = 5\)
  - \(\text{get}(\text{theIndex})\): get element with given index
    - \(L = (a,b,c,d,e)\) \(\text{get}(0) = a, \text{get}(2) = c, \text{get}(4) = e\)
    - \(\text{get}(-1) = \text{error}, \ \text{get}(9) = \text{error}\)
  - \(\text{indexOf}(\text{theElement})\): determine the index of an element
    - \(L = (a,b,d,b,a)\) \(\text{indexOf}(d) = 2, \text{indexOf}(a) = 0, \text{indexOf}(z) = -1\)
  - \(\text{remove}(\text{theIndex})\): remove and return element with given index
    - \(L = (a,b,c,d,e,f,g)\) \(\text{remove}(2)\) returns \(c\) and \(L\) becomes \((a,b,d,e,f,g)\)
    - index of \(d, e, f,\) and \(g\) decrease by 1
    - \(\text{remove}(-1) \Rightarrow \text{error}\)
    - \(\text{remove}(20) \Rightarrow \text{error}\)
  - \(\text{add}(\text{theIndex}, \text{theElement})\): add an element so that the new element has a specified index
    - \(L = (a,b,c,d,e,f,g)\)
    - \(\text{add}(0,h) \Rightarrow L = (h,a,b,c,d,e,f,g)\)
    - index of \(a, b, c, d, e, f,\) and \(g\) increase by \(1\)
    - \(\text{add}(2,h) \Rightarrow L = (a,b,h,c,d,e,f,g)\)
    - index of \(c, d, e, f,\) and \(g\) increase by \(1\)
    - \(\text{add}(10,h) \Rightarrow \text{error}\)
    - \(\text{add}(-6,h) \Rightarrow \text{error}\)
Data Structure Specification

- Language independent: Abstract Data Type
- Java
  - Interface
  - Abstract Class
- Linear List Abstract Data Type

AbstractDataType LinearList
{instances
  ordered finite collections of zero or more elements
operations
  isEmpty(): return true iff the list is empty, false otherwise
  size(): return the list size (number of elements in the list)
  get(index): return the indexth element of the list
  indexOf(x): return the index of the first occurrence of x in the list, return -1 if x is not in the list
  remove(index): remove and return the indexth element, elements with higher index have their index reduced by 1
  add(theIndex, x): insert x as the indexth element, elements with theIndex >= index have their index increased by 1
  output(): output the list elements from left to right

- Linear List as Java Interface
  - An interface may include constants and abstract methods (i.e., methods for which no implementation is provided)

public interface LinearList
{  public boolean isEmpty();
  public int size();
  public Object get(int index);
  public int indexOf(Object elem);
  public Object remove(int index);
  public void add(int index, Object obj);
  public String toString();  }
Java: Interface & Abstract Class

- Implementing An Interface

  ```java
  public class ArrayLinearList implements LinearList
  {
    // code for all LinearList methods must be provided here
  }
  ```

- Linear List As An Abstract Class
  - An abstract class may include constants, variables, abstract methods, and nonabstract methods.

- Linear List As Java Abstract Class

  ```java
  public abstract class LinearListAsAbstractClass
  {
    public abstract boolean isEmpty();
    public abstract int size();
    public abstract Object get(int index);
    public abstract int indexOf(Object theElement);
    public abstract Object remove(int index);
    public abstract void add(int index, Object theElement);
    public abstract String toString();
  }
  ```

- Extending A Java Class

  ```java
  public class ArrayLinearList
  extends LinearListAsAbstractClass
  {
    // code for all abstract classes must come here
  }
  ```

- Implementing Many Interfaces

  ```java
  public class MyInteger implements Operable, Zero,
  CloneableObject
  {
    // code for all methods of Operable, Zero,
    // and CloneableObject must be provided
  }
  ```

- Extending Many Classes
  - A Java class may implement as many interfaces as it wants but can extend at most 1 class.

- Java specifies all of its data structures as interfaces.
  - java.util.List
Differences between Java & C++

- Every class should be defined in a separate file has the same name of the class.
- Beside public, protected and private access modifier, there is the default one (without specifying any modifier).
- The default access modifier allows friend classes to access the class data and methods.
- The friend classes are whose defined within the same package.
- Several classes can be defined in one file if they are within one package.
- As a class is mapped to a file, a package is mapped to a directory. The hierarchy of classes is mapped to hierarchy of directories.
- Java has four level of abstractions: Objects, Classes, Abstract classes and Interfaces.
- Naming convention:
  - A class start with Capital letter
  - A data or a method starts with small letter
  - Whenever several words are used in naming, capitalize the word initials.
- The same identifier can be used for a class name, a method (constructor) and a data (variable).
- Signatures include the type of return value.
- The main program is a test program in every class (appears once per class). It is a static method.
- Objects are only created as reference, so always we use new (there is no pointers and you can not have object).
Linear List Array Representation

- use a one-dimensional array element[]

| a | b | c | d | e |

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |

- $L = (a, b, c, d, e)$, Store element $i$ of list in element[$i$].
- Right To Left Mapping

| e | d | c | b | a |

- Mapping That Skips Every Other Position

| a | b | c | d | e |

- Wrap Around Mapping

| d | e | a | b | c |

- Add/Remove An Element

| a | b | c | d | e |

- add(1, g) size = 6

| a | g | b | c | d | e |

- Data Type Of Array element[]
  - Data type of list elements is unknown.
  - Define element[] to be of data type Object.
  - Cannot put elements of primitive data types (int, float, double, char, etc.) into our linear lists.

- Length of Array element[]
  - Don’t know how many elements will be in list.
  - Must pick an initial length and dynamically increase as needed
Array Representation

- **Increasing Array Length**
  - Length of array element[] is 6.
  - First create a new and larger array
  - Now copy elements from old array to new one.
  - Finally, rename new array.

```
// create a new array of proper length and data type
Object [] newArray = new Object [newLength];
// copy all elements from old array into new one
System.arraycopy(element, 0, newArray, 0, element.length);
// rename array
element = newArray;
```

```
public static Object [] changeLength(Object [] a, int newLength)
{
  Object [] newArray = Object [newLength];
  System.arraycopy(a, 0, newArray, 0, a.length);
  return newArray;
}
```

- **How Big Should The New Array Be?**
  - At least 1 more than current array length.
  - Cost of increasing array length is $\Theta(new\ length)$
  - Cost of $n$ add operations done on an initially empty linear list increases by $\Theta(n^2)$

- **Space Complexity**
  - `element[6]` `newArray = new char[7];`
  - space needed = \[2 * newLength - 1\]
    \[= 2 * oldLength + 1\]
  - Array Doubling
  - Time for $n$ adds goes up by $\Theta(n)$.
  - Space needed = 1.5*newLength.
  - Space needed $\leq 3\ times\ oldLength$

- **How Big Should The New Array Be?**
  - Resizing by any constant factor $new\ length = c * old\ length$
  - increases the cost of $n$ adds by $\Theta(n)$.
  - Resizing by an additive constant increases the cost of $n$ add operations by $\Theta(n^2)$.
How Big Should The New Array Be?

- Resizing by any constant factor \( \text{new length} = c \times \text{old length} \)
- requires at most \((1+c) \times \text{oldLength}\) space.
- Resizing by an additive constant \(c\) requires at most
  \[(\text{oldLength}) + (\text{oldLength} + c)\]
  \[= 2 \times (\text{oldLength}) + c\] space.

How Does Java Do?

- java.util.Vector … array doubling
- java.util.ArrayList … \(c = 1.5\)

The Class ArrayLinearList

- General purpose implementation of linear lists.
- Unknown number of lists.

Create An Empty List

ArrayLinearList \(a = \text{new ArrayLinearList}(100),\)
\[b = \text{new ArrayLinearList}(),\]
\(c;\)
LinearList \(d = \text{new ArrayLinearList}(1000),\)
\(e = \text{new ArrayLinearList}(),\)
\(f;\)

Using A Linear List

System.out.println(a.size());
a.add(0, new Integer(2));
b.add(0, new Integer(4));
System.out.println(a);
b.remove(0);
if (a.isEmpty())
  a.add(0, new Integer(5));

Array Of Linear Lists

LinearList [] \(x = \text{new LinearList}[4];\)
x[0] = new ArrayLinearList(20);
x[1] = new Chain();
x[2] = new Chain();
x[3] = new ArrayLinearList();
for (int \(i = 0; \ i < 4; \ i++\))
  x[i].add(0, new Integer(i));
### The Class ArrayLinearList

```java
to implement LinearList */
import java.util.*;    // has exception classes
import utilities.*;     // has array resizing class

public class ArrayLinearList implements LinearList

{   protected Object[] element;  // array of elements
    protected int size;    // number of elements in array
}

// constructors and other methods come here

/** create a list with initial capacity initialCapacity
 throws IllegalArgumentException when initialCapacity < 1 */
public ArrayLinearList(int initialCapacity)
{  if (initialCapacity < 1)
    throw new IllegalArgumentException
        ("initialCapacity must be >= 1");
    // size has the default initial value of 0
    element = new Object[initialCapacity];
}

public ArrayLinearList() /** create a list with initial capacity 10 */
{  this(10);      } // use default capacity of 10

public boolean isEmpty() /** return true iff list is empty */
{return size == 0;}  

public int size() /** return current number of elements in list */
{return size;}

/** throws IndexOutOfBoundsException when index is not between 0 and size - 1 */
void checkIndex(int index)
{  if (index < 0 || index >= size)
    throw new IndexOutOfBoundsException
        ("index = " + index + " size = " + size);
}

/** return element with specified index, throws IndexOutOfBoundsException when index is not between 0 and size - 1 */
public Object get(int index)
{    checkIndex(index);
    return element[index];      }
```
The Class ArrayLinearList (2)

/** return index of first occurrence of theElement,
   return -1 if theElement not in list */
public int indexOf(Object theElement)  
{ for (int i = 0; i < size; i++) // search element[] for theElement
    if (element[i].equals(theElement))
        return i;
    return -1; }    // theElement not found

public Object remove(int index)
{ checkIndex(index);
  // valid index, shift elements with higher index
  Object removedElement = element[index];
  for (int i = index + 1; i < size; i++)
      element[i-1] = element[i];
  element[--size] = null;   // enable garbage collection
  return removedElement; }

public void add(int index, Object theElement)
{ if (index < 0 || index > size) // invalid list position
    throw new IndexOutOfBoundsException
        ("index = " + index + "  size = " + size);
  if (size == element.length)      // valid index, make sure we have space
  // no space, double capacity
      element = ChangeArrayLength.changeLength1D(element, 2 * size);
  for (int i = size - 1; i >= index; i--) // shift elements right one position
      element[i + 1] = element[i];
  element[index] = theElement;
  size++;      }

● Faster Way To Shift Elements 1 Right
System.arraycopy(element, index, element, index + 1, size - index);
● Convert To A String

public String toString()
{ StringBuffer s = new StringBuffer("[");
  for (int i = 0; i < size; i++) // put elements into the buffer
      if (element[i] == null) s.append("null, ");
    else s.append(element[i].toString() + ", ");
    if (size > 0) s.delete(s.length() - 2, s.length());        // remove last ", 
      s.append("]");
  return new String(s); } // create equivalent String
Iterators

- An iterator permits you to examine the elements of a data structure one at a time.

- **Iterator Methods**
  - `Iterator ix = x.iterator();`
    - constructs and initializes an iterator to examine the elements of `x`; constructed iterator is assigned to `ix`
    - you must define the method iterator in the class for `x`
  - `ix.hasNext( )`: returns true iff `x` has a next element
  - `ix.next( )`: throws `NoSuchElementException` if there is no next element, returns next element otherwise
  - `ix.remove( )`: removes last element returned by `ix.next( )`
    - throws `UnsupportedMethodException` if method not implemented
    - throws `IllegalStateException` if `ix.next()` not yet called or did not return an element

- **Using An Iterator**

  ```java
  Iterator ix = x.iterator();
  while (ix.hasNext()) examine(get(i));
  vs
  for (int i = 0; i < x.size(); i++) examine(get(i));
  ```

- **Merits Of An Iterator**
  - It is often possible to implement the method `next` so that its complexity is less than that of `get`
  - Many data structures do not have a `get` by index method
  - Iterators provide a uniform way to sequence through the elements of a data structure

- **Linked Representation**
  - lists elements are stored, in memory, in an arbitrary order
  - explicit information (called a link) is used to go from one element to the next
  - Layout of `L = (a,b,c,d,e)` using an array representation.

  ```
  a b c d e
  ```
Linked Representation

- A linked representation uses an arbitrary layout.

- pointer (or link) in e is null
- use a variable firstNode to get to the first element a

- Normal Way To Draw A Linked List

```
firstNode
```

- Chain
  - A chain is a linked list, each node represents one element.
  - There is a link or pointer from one element to the next.
  - The last node has a null pointer.

- Node Representation

```java
package dataStructures;

class ChainNode // package visible data members
{
  Object element;
  ChainNode next;

  ChainNode() {} // constructors come here
  ChainNode(Object element) {this.element = element;}
  ChainNode(Object element, ChainNode next)
  {this.element = element; this.next = next;}
}
```

- get(0) desiredNode = firstNode; // gets you to first node
- get(1) desiredNode = firstNode.next; // gets the second node
- get(2) desiredNode = firstNode.next.next; // gets the third node
- get(5) desiredNode = firstNode.next.next.next.next.next;
Remove and Add

- **NullPointerException**
  - desiredNode = firstNode.next.next.next.next.next.next;
  - gets the computer mad, you get a NullPointerException

- **Remove An Element**
  - remove(0) firstNode = firstNode.next;
  - remove(2)
    - first get to node just before node to be removed,
      beforeNode = firstNode.next;
    - now change pointer in beforeNode
      beforeNode.next = beforeNode.next.next;

- **Add an Element**
  - add(0,’f’)
    - get a node, set its data and link fields
      ChainNode newNode = new ChainNode(new Character(‘f’), firstNode);
    - update firstNode
      firstNode = newNode;
Add element at the middle

- **add(3,'f')**
  - first find node whose index is 2
  - next create a node and set its data and link fields
  - finally link beforeNode to newNode

```java
ChainNode newNode = new ChainNode(new Character('f'), beforeNode.next);
beforeNode.next = newNode;
beforeNode = firstNode.next.next;
beforeNode.next = new ChainNode(new Character('f'), beforeNode.next);
```

- **The Class Chain**
  /** linked implementation of LinearList */
  package dataStructures;
  import java.util.*; // has exception classes
  public class Chain implements LinearList
  {
      protected ChainNode firstNode; // data members
      protected int size;
      // methods of Chain come here
      public Chain(int initialCapacity) /** create a list that is empty */
      {
          // the default initial values of firstNode and size
          } // are null and 0, respectively
      public Chain( ) {this(0);}
      public boolean isEmpty( ) /** @return true iff list is empty */
      {return size == 0;}
      public int size( ) /** @return current number of elements in list */
      {return size;}
      // other methods
  }
**checkIndex, get, indexOf**

/** @throws IndexOutOfBoundsException when
 * index is not between 0 and size - 1 */

void checkIndex(int index)
{ if (index < 0 || index >= size)
    throw new IndexOutOfBoundsException
        ("index = " + index + "  size = " + size); }

public Object get(int index)
{ checkIndex(index);
    ChainNode currentNode = firstNode; // move to desired node
    for (int i = 0; i < index; i++)
        currentNode = currentNode.next;
    return currentNode.element; }

public int indexOf(Object theElement)
{ // search the chain for theElement
    ChainNode currentNode = firstNode; // index of currentNode
    int index = 0;
    while (currentNode != null &&
        !currentNode.element.equals(theElement))
    { // move to next node
        currentNode = currentNode.next;
        index++; }
    // make sure we found matching element
    if (currentNode == null)
        return -1;
    else
        return index;
}
public Object remove(int index) {
    checkIndex(index);
    Object removedElement;
    if (index == 0) // remove first node
    { removedElement = firstNode.element;
      firstNode = firstNode.next;
    }
    else
    { // use q to get to predecessor of desired node
      ChainNode q = firstNode;
      for (int i = 0; i < index - 1; i++)
        q = q.next;
      removedElement = q.next.element;
      q.next = q.next.next; // remove desired node
    }
    size--;
    return removedElement;
}
public void add(int index, Object theElement) {
  if (index < 0 || index > size)
    // invalid list position
    throw new IndexOutOfBoundsException("index = " + index + " size = " + size);
  if (index == 0)
    // insert at front
    firstNode = new ChainNode(theElement, firstNode);
  else
  { // find predecessor of new element
    ChainNode p = firstNode;
    for (int i = 0; i < index - 1; i++)
      p = p.next;
    // insert after p
    p.next = new ChainNode(theElement, p.next);
  }
size++; }
### Performance

- 40,000 operations of each type

<table>
<thead>
<tr>
<th>Operation</th>
<th>FastArrayLinearList</th>
<th>Chain</th>
</tr>
</thead>
<tbody>
<tr>
<td>get</td>
<td>5.6ms</td>
<td>157sec</td>
</tr>
<tr>
<td>best-case adds</td>
<td>31.2ms</td>
<td>304ms</td>
</tr>
<tr>
<td>average adds</td>
<td>5.8sec</td>
<td>115sec</td>
</tr>
<tr>
<td>worst-case adds</td>
<td>11.8sec</td>
<td>157sec</td>
</tr>
<tr>
<td>best-case removes</td>
<td>8.6ms</td>
<td>13.2ms</td>
</tr>
<tr>
<td>average removes</td>
<td>5.8sec</td>
<td>149sec</td>
</tr>
<tr>
<td>worst-case removes</td>
<td>11.7sec</td>
<td>157sec</td>
</tr>
</tbody>
</table>

- **Chain With Header Node**

- **Circular List**

- **Doubly Linked Circular List With Header Node**

- Linked implementation of a linear list.
- Doubly linked circular list with header node.
- Has all methods of LinearList plus many more.
Stacks

- Linear list.
- One end is called top.
- Other end is called bottom.
- Additions to and removals from the top end only.
  - Add a cup to the stack.
  - Remove a cup from new stack.
  - A stack is a LIFO list.
- The Interface Stack

```java
public interface Stack {
  public boolean empty();
  public Object peek();
  public void push(Object theObject);
  public Object pop();
}
```

- Parentheses Matching
  - $(((a+b)*c+d-e)/(f+g)-(h+j)*(k-l))/(m-n)$
  - Output pairs $(u,v)$ such that the left parenthesis at position $u$ is matched with the right parenthesis at $v$.
    - $(2,6)$ $(1,13)$ $(15,19)$ $(21,25)$ $(27,31)$ $(0,32)$ $(34,38)$
  - $(a+b))^((c+d)$
    - $(0,4)$
      - right parenthesis at 5 has no matching left parenthesis
      - $(8,12)$
        - left parenthesis at 7 has no matching right parenthesis
  - scan expression from left to right
  - when a left parenthesis is encountered, add its position to the stack
  - when a right parenthesis is encountered, remove matching position from stack
  - Example: $(((a+b)*c+d-e)/(f+g)-(h+j)*(k-l))/(m-n)$
  - $(2,6)$, $(1,13)$ $(15,19)$ $(21,25)$ $(27,31)$ $(0,32)$
Towers Of Hanoi

- 64 gold disks to be moved from tower A to tower C
- each tower operates as a stack
- cannot place big disk on top of a smaller one
- 3-disk Towers Of Hanoi
- 3-disk Towers Of Hanoi
  - 7 disk moves

Recursive Solution
- \( n > 0 \) gold disks to be moved from A to C using B
- move top \( n-1 \) disks from A to B using C
- move top disk from A to C
- move top \( n-1 \) disks from B to C using A
- \( \text{moves}(n) = 0 \) when \( n = 0 \)
- \( \text{moves}(n) = 2 \times \text{moves}(n-1) + 1 = 2n-1 \) when \( n > 0 \)
- \( \text{moves}(64) = 1.8 \times 10^{19} \) (approximately)
- Performing \( 10^9 \) moves/second, a computer would take about 570 years to complete.

Chess Story
- One 1 grain of rice on the first square, 2 for next, 4 for next, 8 for next, and so on.
- Surface area needed exceeds surface area of earth.

Switch Box Routing

Method Invocation And Return
- public void a() { ...; b(); ...}  return address in e()
- public void b() { ...; c(); ...}  return address in d()
- public void c() { ...; d(); ...}  return address in c()
- public void d() { ...; e(); ...}  return address in b()
- public void e() { ...; c(); ...}  return address in a()

Try-Throw-Catch
- When you enter a try block, push the address of this block on a stack.
- When an exception is thrown, pop the try block that is at the top of the stack (if the stack is empty, terminate).
- If the popped try block has no matching catch block, go back to the preceding step.
- If the popped try block has a matching catch block, execute the matching catch block.
Derive From A Linear List Class

- Derive From ArrayLinearList

| a | b | c | d | e | null |

- stack top is either left end or right end of linear list
- empty() => isEmpty() \(O(1)\) time
- peek() => get(0) or get(size() - 1) \(O(1)\) time
- when top is left end of linear list
  - push(theObject) => add(0, theObject) \(O(size)\) time
  - pop() => remove(0) \(O(size)\) time
- when top is right end of linear list
  - push(theObject) => add(size(), theObject) \(O(1)\) time
  - pop() => remove(size()-1) \(O(1)\) time
- use right end of list as top of stack

- Derive From Chain

- stack top is either left end or right end of linear list
- empty() => isEmpty() \(O(1)\) time
- when top is left end of linear list
  - peek() => get(0) \(O(1)\) time
  - push(theObject) => add(0, theObject) \(O(1)\) time
  - pop() => remove(0) \(O(1)\) time
- when top is right end of linear list
  - peek() => get(size() - 1) \(O(size)\) time
  - push(theObject) => add(size(), theObject) \(O(size)\) time
  - pop() => remove(size()-1) \(O(size)\) time
- use left end of list as top of stack
package dataStructures;
import java.util.*; // has stack exception
public class DerivedArrayStack
    extends ArrayLinearList
    implements Stack
{
    // constructors come here
    // create a stack with the given initial capacity
    public DerivedArrayStack(int initialCapacity)
    {super(initialCapacity);}
    // create a stack with initial capacity 10
    public DerivedArrayStack() {this(10);}
    // Stack interface methods come here
    public boolean empty() {return isEmpty();}
    public Object peek()
    {  if (empty()) throw new EmptyStackException();
        return get(size() - 1) }
    public void push(Object theElement)
    {add(size(), theElement);}
    public Object pop()
    {  if (empty()) throw new EmptyStackException();
        return remove(size() - 1); } }

● Evaluation: Merits of deriving from ArrayLinearList
  – Code for derived class is quite simple and easy to develop.
  – Code is expected to require little debugging.
  – Code for other stack implementations such as a linked implementation are easily obtained.
  – Just replace extends ArrayLinearList with extends Chain
  – For efficiency reasons we must also make changes to use the left end of the list as the stack top rather than the right end.
Demerits

- All public methods of ArrayLinearList may be performed on a stack.
  - get(0) … get bottom element
  - remove(5)
  - add(3, x)
  - So we do not have a true stack implementation.
  - Must override undesired methods.

```java
public Object get(int theIndex)
{throw new UnsupportedOperationException();}
```
- Change earlier use of get(i) to super.get(i).

- Unnecessary work is done by the code.
  - peek() verifies that the stack is not empty before get is invoked. The index check done by get is, therefore, not needed.
  - add(size(), theElement) does an index check and a for loop that is not entered. Neither is needed.
  - pop() verifies that the stack is not empty before remove is invoked. remove does an index check and a for loop that is not entered. Neither is needed.
  - So the derived code runs slower than necessary.

- Evaluation
  - Code developed from scratch will run faster but will take more time (cost) to develop.
  - Tradeoff between software development cost and performance.
  - Tradeoff between time to market and performance.
  - Could develop easy code first and later refine it to improve performance.

- A Faster `pop()`
  ```java
  if (empty())
  try {return remove(size() - 1);} 
  throw new EmptyStackException();
  return remove(size() - 1);
  catch(OutOfBoundsException e) 
  {throw new EmptyStackException();}
  ```

- Use a 1D array stack whose data type is Object.
  - same as using array element in ArrayLinearList
Code From Scratch

- Use an int variable top.
  - Stack elements are in stack[0:top].
  - Top element is in stack[top].
  - Bottom element is in stack[0].
  - Stack is empty iff top = -1.
  - Number of elements in stack is top+1.

```java
package dataStructures;
import java.util.EmptyStackException;
import utilities.*;  // ChangeArrayLength
public class ArrayStack implements Stack
{
    int top;               // current top of stack
    Object [] stack;  // element array
    // constructors come here
    public ArrayStack(int initialCapacity)
    {  if (initialCapacity < 1)     throw new IllegalArgumentException
                                       ("initialCapacity must be >= 1");
        stack = new Object [initialCapacity]; top = -1;}
    public ArrayStack( ) {this(10);}
    // Stack interface methods come here
    public Object pop()
    {  if (empty()) throw new EmptyStackException();
        Object topElement = stack[top];
        stack[top--] = null;   // enable garbage collection
        return topElement; }
}
```

- java.util.Stack
  - Derives from java.util.Vector.
  - java.util.Vector is an array implementation of a linear list.

- Performance: 500,000 pop, push, and peek operations

<table>
<thead>
<tr>
<th>Class</th>
<th>Initial capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
</tr>
<tr>
<td>ArrayStack</td>
<td>0.44s</td>
</tr>
<tr>
<td>DerivedArrayStack</td>
<td>0.60s</td>
</tr>
<tr>
<td>DerivedArrayStackWithCatch</td>
<td>0.55s</td>
</tr>
<tr>
<td>java.util.Stack</td>
<td>1.15s</td>
</tr>
<tr>
<td>DerivedLinkedStack</td>
<td>3.20s</td>
</tr>
<tr>
<td>LinkedStack</td>
<td>2.96s</td>
</tr>
</tbody>
</table>
Queues

- Linear list.
- One end is called front.
- Other end is called rear.
- Additions are done at the rear only.
- Removals are made from the front only.

The Interface Queue

```java
public interface Queue {
    public boolean isEmpty();
    public Object getFrontElement();
    public Object getRearElement();
    public void put(Object theObject);
    public Object remove();
}
```

Applications in which the stack cannot be replaced with a queue.
- Parentheses matching.  
  Towers of Hanoi.
- Switchbox routing.  
  Method invocation and return.
- Try-catch-throw implementation.

Derive From ArrayLinearList
- when front is left end of list and rear is right end
- `Queue.isEmpty() => ArrayLinearList.isEmpty()`  \(O(1)\) time
- `getFrontElement() => get(0)`  \(O(1)\) time
- `getRearElement() => get(size() - 1)`  \(O(1)\) time
- `put(theObject) => add(size(), theObject)`  \(O(1)\) time
- `remove() => remove(0)`  \(O(size)\) time
Derive From ArrayLinearList

- when rear is left end of list and front is right end
  - Queue.isEmpty() => ArrayLinearList.isEmpty() \( \text{O}(1) \) time
  - getFrontElement() => get(size() - 1) \( \text{O}(1) \) time
  - getRearElement() => get(0) \( \text{O}(1) \) time
  - put(theObject) => add(0, theObject) \( \text{O}(\text{size}) \) time
  - remove() => remove(size() - 1) \( \text{O}(1) \) time

- to perform each operation in \( \text{O}(1) \) time (excluding array doubling), we need a customized array representation.

Derive From ExtendedChain

- when front is left end of list and rear is right end
  - Queue.isEmpty() => ExtendedChain.isEmpty() \( \text{O}(1) \) time
  - getFrontElement() => get(0) \( \text{O}(1) \) time
  - getRearElement() => getLast() \( \text{O}(1) \) time
  - put(theObject) => append(theObject) \( \text{O}(1) \) time
  - remove() => remove(0) \( \text{O}(1) \) time

- when front is right end of list and rear is left end
  - Queue.isEmpty() => ExtendedChain.isEmpty() \( \text{O}(1) \) time
  - getFrontElement() => getLast() \( \text{O}(1) \) time
  - getRearElement() => get(0) \( \text{O}(1) \) time
  - put(theObject) => add(0, theObject) \( \text{O}(1) \) time
  - remove() => remove(size-1) \( \text{O}(\text{size}) \) time

Custom Linked Code

- Develop a linked class for Queue from scratch to get better performance than obtainable by deriving from ExtendedChain.

Custom Array Queue

- Use a 1D array queue. queue[]
Custom Array Queue

- Circular view of array.
- Use integer variables front and rear.
  - front is one position counterclockwise from first
  - rear gives position of last element

- Add An Element
  - Move rear one clockwise.
  - Then put into queue[rear].

- Remove An Element
  - Move front one clockwise.
  - Then extract from queue[front].

- Moving Clockwise
  - rear++;
  - if (rear == queue.length) rear = 0;
  - rear = (rear + 1) % queue.length;

- Empty That Queue
  - When a series of removals causes the queue to become empty, front = rear.
  - When a queue is constructed, it is empty.
  - So initialize front = rear = 0.

- A Full Tank Please
  - When a series of adds causes the queue to become full, front = rear.
  - So we cannot distinguish between a full queue and an empty queue!

- Remedies.
  - Don’t let the queue get full.
    - When the addition of an element will cause the queue to be full, increase array size.
    - This is what the text does.
  - Define a boolean variable lastOperationIsPut.
    - Following each put set this variable to true.
    - Following each remove set to false.
    - Queue is empty iff (front == rear) & & !lastOperationIsPut
    - Queue is full iff (front == rear) & & lastOperationIsPut
  - Performance is slightly better when first strategy is used.
Dictionaries

- Collection of items.
- Each item is a pair.
  - (key, element)
  - Pairs have different keys.

Operations.
- get(theKey)
- put(theKey, theElement)
- remove(theKey)

Application
- Collection of student records in this class.
  - (key, element) = (student name, linear list of assignment and exam scores)
  - All keys are distinct.
- Get the element whose key is John Adams.
- Update the element whose key is Diana Ross.
  - put() implemented as update when there is already a pair with the given key.
  - remove() followed by put().

Dictionary With Duplicates
- Keys are not required to be distinct.
- Word dictionary.
  - Pairs are of the form (word, meaning).
  - May have two or more entries for the same word.
    - (bolt, a threaded pin)
    - (bolt, a crash of thunder)
    - (bolt, to shoot forth suddenly)
    - (bolt, a gulp)
    - (bolt, a standard roll of cloth), etc.

Represent As A Linear List
- L = (e0, e1, e2, e3, ..., en-1)
- Each ei is a pair (key, element).
- 5-element dictionary D = (a, b, c, d, e).
  - a = (aKey, aElement), b = (bKey, bElement), etc.
- Array or linked representation.
Array Representation

- get(theKey) O(size) time
- put(theKey, theElement)
  - O(size) time to verify duplicate, O(1) to add at right end.
- remove(theKey) O(size) time.

Sorted Array
- elements are in ascending order of key.
- get(theKey) O(log size) time
- put(theKey, theElement)
  - O(log size) time to verify duplicate, O(size) to add.
- remove(theKey) O(size) time.

Unsorted Chain
- get(theKey) O(size) time
- put(theKey, theElement)
  - O(size) time to verify duplicate, O(1) to add at left end.
- remove(theKey) O(size) time.

Sorted Chain
- Elements are in ascending order of Key.
- get(theKey) O(size) time
- put(theKey, theElement)
  - O(size) time to verify duplicate, O(1) to add at left end.
- remove(theKey) O(size) time.

Hash Tables
- Worst-case time for get, put, and remove is O(size).
- Expected time is O(1).

Ideal Hashing
- Uses a 1D array (or table) table[0:b-1].
  - Each position of this array is a bucket.
  - A bucket can normally hold only one dictionary pair.
- Uses a hash function \( f \) that converts each key \( k \) into an index in the range \([0, b-1]\).
  - \( f(k) \) is the home bucket for key \( k \).
- Every dictionary pair (key, element) is stored in its home bucket table[\( f(key) \)].
Ideal Hashing Example

- Pairs are: (22,a), (33,c), (3,d), (73,e), (85,f).
- Hash table is table[0:7], b = 8.
- Hash function is key/11.
- Pairs are stored in table as below:

<table>
<thead>
<tr>
<th></th>
<th>(3,d)</th>
<th>(22,a)</th>
<th>(33,c)</th>
<th></th>
<th>(73,e)</th>
<th>(85,f)</th>
</tr>
</thead>
</table>

- get, put, and remove take O(1) time.
- What Can Go Wrong?
  - Where does (26,g) go?
  - Keys that have the same home bucket are synonyms.
    - 22 and 26 are synonyms with respect to the hash function that is in use.
  - The home bucket for (26,g) is already occupied.
- A collision occurs when the home bucket for a new pair is occupied by a pair with a different key.
- An overflow occurs when there is no space in the home bucket for the new pair.
- When a bucket can hold only one pair, collisions and overflows occur together.
- Need a method to handle overflows.

- Hash Table Issues
  - Choice of hash function.
  - Overflow handling method.
  - Size (number of buckets) of hash table.

- Hash Functions
  - Two parts:
    - Convert key into an integer in case the key is not an integer.
      - Done by the method hashCode().
    - Map an integer into a home bucket.
      - f(k) is an integer in the range [0, b-1], where b is the number of buckets in the table.

- Map Into A Home Bucket
  - Most common method is by division.
    - homeBucket = Math.abs(theKey.hashCode()) % divisor;
    - divisor equals number of buckets b.
    - 0 <= homeBucket < divisor = b
Uniform Hash Function

- **Uniform Hash Function**
  - Let keySpace be the set of all possible keys.
  - A uniform hash function maps the keys in keySpace into buckets such that approximately the same number of keys get mapped into each bucket.
  - Equivalently, the probability that a randomly selected key has bucket $i$ as its home bucket is $1/b$, $0 \leq i < b$.
  - A uniform hash function minimizes the likelihood of an overflow when keys are selected at random.

- **Hashing By Division**
  - keySpace = all ints.
  - For every $b$, the number of ints that get mapped (hashed) into bucket $i$ is approximately $2^{32}/b$.
  - Therefore, the division method results in a uniform hash function when keySpace = all ints.
  - In practice, keys tend to be correlated.
  - So, the choice of the divisor $b$ affects the distribution of home buckets.

- **Selecting The Divisor**
  - Because of this correlation, applications tend to have a bias towards keys that map into odd integers (or into even ones).
  - When the divisor is an even number, odd integers hash into odd home buckets and even integers into even home buckets.
  - $20\%14 = 6$, $30\%14 = 2$, $8\%14 = 8$
  - $15\%14 = 1$, $3\%14 = 3$, $23\%14 = 9$
  - The bias in the keys results in a bias toward either the odd or even home buckets.
  - When the divisor is an odd number, odd (even) integers may hash into any home.
    - $20\%15 = 5$, $30\%15 = 0$, $8\%15 = 8$
    - $15\%15 = 0$, $3\%15 = 3$, $23\%15 = 8$
  - The bias in the keys does not result in a bias toward either the odd or even home buckets.
  - Better chance of uniformly distributed home buckets.
  - So do not use an even divisor.
Selecting The Divisor

- Similar biased distribution of home buckets is seen, in practice, when the divisor is a multiple of prime numbers such as 3, 5, 7, ...
- The effect of each prime divisor $p$ of $b$ decreases as $p$ gets larger.
- Ideally, choose $b$ so that it is a prime number.
- Alternatively, choose $b$ so that it has no prime factor smaller than 20.

- **Java.util.HashTable**
  - Simply uses a divisor that is an odd number.
  - This simplifies implementation because we must be able to resize the hash table as more pairs are put into the dictionary.
    - Array doubling, for example, requires you to go from a 1D array table whose length is $b$ (which is odd) to an array whose length is $2b+1$ (which is also odd).

- **Overflow Handling**
  - An overflow occurs when the home bucket for a new pair (key, element) is full.
  - We may handle overflows by:
    - Search the hash table in some systematic fashion for a bucket that is not full.
      - Linear probing (linear open addressing).
      - Quadratic probing.
      - Random probing.
    - Eliminate overflows by permitting each bucket to keep a list of all pairs for which it is the home bucket.
      - Array linear list.
      - Chain.

- **Linear Probing – Get And Put**
  - $\text{divisor} = b$ (number of buckets) = 17.
  - $\text{Home bucket} = \text{key} \% 17.$
**Linear Probing**

- Put in pairs whose keys are 6, 12, 34, 29, 28, 11, 23, 7, 0, 33, 30, 45

- **Remove: remove(0)**

  - Search cluster for pair (if any) to fill vacated bucket.

- **remove(34)**

  - Search cluster for pair (if any) to fill vacated bucket.

- **remove(29)**

  - Search cluster for pair (if any) to fill vacated bucket.

- **Performance Of Linear Probing**
  - Worst-case get/put/remove time is Theta(n), where n is the number of pairs in the table.
  - This happens when all pairs are in the same cluster.
Expected Performance

- \( \alpha = \text{loading density} = \frac{\text{(number of pairs)}}{b} \)
  - \( \alpha = \frac{12}{17} \).
- \( S_n = \text{expected number of buckets examined in a successful search when } n \text{ is large} \)
- \( U_n = \text{expected number of buckets examined in a unsuccessful search when } n \text{ is large} \)
- Time to put and remove governed by \( U_n \).
- \( S_n \sim \frac{1}{2}(1 + 1/(1 – \alpha)) \)
- \( U_n \sim \frac{1}{2}(1 + 1/(1 – \alpha)^2) \)
- Note that \( 0 \leq \alpha \leq 1 \).
- Alpha \( \leq 0.75 \) is recommended.

Hash Table Design

- Performance requirements are given, determine maximum permissible loading density.
- We want a successful search to make no more than 10 compares (expected).
  - \( S_n \sim \frac{1}{2}(1 + 1/(1 – \alpha)) \)
  - \( \alpha \leq \frac{18}{19} \)
- We want an unsuccessful search to make no more than 13 compares (expected).
  - \( U_n \sim \frac{1}{2}(1 + 1/(1 – \alpha)^2) \)
  - \( \alpha \leq \frac{4}{5} \)
  - So \( \alpha \leq \min\{\frac{18}{19}, \frac{4}{5}\} = \frac{4}{5} \).
- Dynamic resizing of table.
  - Whenever loading density exceeds threshold (\( \frac{4}{5} \) in our example), rehash into a table of approximately twice the current size.
- Fixed table size.
  - Know maximum number of pairs.
  - No more than 1000 pairs.
  - Loading density \( \leq \frac{4}{5} \Rightarrow b \geq \frac{5}{4} * 1000 = 1250 \).
  - Pick \( b \) (equal to divisor) to be a prime number or an odd number with no prime divisors smaller than 20.
Hashing With Chains

– Each bucket keeps a chain of all pairs for which it is the home bucket.
– The chain may or may not be sorted by key.

- **Sorted Chains**
  – Put in pairs whose keys are 6, 12, 34, 29, 28, 11, 23, 7, 0, 33, 30, 45
  – Home bucket = key % 17.

- **Expected Performance**
  – Note that alpha >= 0.
  – Expected chain length is alpha.
  – S_n <= max{1, alpha}.
  – U_n <= alpha.

- **Java.util.Hashtable**
  – Unsorted chains.
  – Default initial b = divisor = 101
  – Default alpha <= 0.75
  – When loading density exceeds max permissible density, rehash with newB = 2b+1.
**Trees**

- **Computer Scientist’s View**
- **Linear Lists And Trees**
  - Linear lists are useful for serially ordered data.
    - (e0, e1, e2, ..., en-1)
    - Days of week.
    - Months in a year.
    - Students in this class.
  - Trees are useful for hierarchically ordered data.
    - Employees of a corporation.
      - President, vice presidents, managers, and so on.
    - Java’s classes.
      - Object is at the top of the hierarchy.
      - Subclasses of Object are next, and so on.
- **Hierarchical Data And Trees**
  - The element at the top of the hierarchy is the root.
  - Elements next in the hierarchy are the children of the root.
  - Elements next in the hierarchy are the grandchildren of the root, and so on.
  - Elements at the lowest level of the hierarchy are the leaves.
  - Java’s Classes
Tree Definition

- A tree $t$ is a finite nonempty set of elements.
- One of these elements is called the root.
- The remaining elements, if any, are partitioned into trees, which are called the subtrees of $t$.

- **Subtrees**
  - Object
  - Number
    - Integer
    - Double
  - Throwable
    - Exception
    - RuntimeException
  - OutputStream
    - FileOutputStream

- **Leaves**

- **Parent, Grandparent, Siblings, Ancestors, Descendents**

- **Levels – Caution**
  - Some texts start level numbers at 0 rather than at 1.
  - Root is at level 0. Its children are at level 1.
  - The grand children of the root are at level 2. And so on.
  - We shall number levels with the root at level 1.

- **height = depth = number of levels**

- **Node Degree = Number Of Children**

- **Tree Degree = Max Node Degree - Degree of the above tree = 3**

- **Binary Tree**
  - Finite (possibly empty) collection of elements.
  - A nonempty binary tree has a root element.
  - The remaining elements (if any) are partitioned into two binary trees.
  - These are called the left and right subtrees of the binary tree.

- **Differences Between A Tree & A Binary Tree**
  - No node in a binary tree may have a degree more than 2, whereas there is no limit on the degree of a node in a tree.
  - A binary tree may be empty; a tree cannot be empty.
  - The subtrees of a binary tree are ordered; those of a tree are not ordered.
Arithmetic Expressions

- Differences Between A Tree & A Binary Tree
  - The subtrees of a binary tree are ordered; those of a tree are not ordered.
    - Are different when viewed as binary trees.
    - Are the same when viewed as trees.

- Arithmetic Expressions
  - \((a + b) \times (c + d) + e - f/g \times h + 3.25\)
  - Expressions comprise three kinds of entities.
    - Operators (+, -, /, *).
    - Operands (a, b, c, d, e, f, g, h, 3.25, (a + b), (c + d), etc.).
    - Delimiters ((, )).
  - Operator Degree
    - Number of operands that the operator requires.
    - Binary operator requires two operands.
      - \(a + b\)
      - \(c / d\)
      - \(e - f\)
    - Unary operator requires one operand.
      - \(+ g\)
      - \(- h\)
  - Infix Form
    - Normal way to write an expression.
    - Binary operators come in between their left and right operands.
      - \(a * b\)
      - \(a + b * c\)
      - \(a * b / c\)
      - \((a + b) * (c + d) + e - f/g \times h + 3.25\)
  - Operator Priorities
    - How do you figure out the operands of an operator?
      - \(a + b * c\)
      - \(a * b + c / d\)
    - This is done by assigning operator priorities.
      - \(\text{priority}(\times) = \text{priority}(\div) > \text{priority}(+) = \text{priority}(-)\)
    - When an operand lies between two operators, the operand associates with the operator that has higher priority.
Arithmetic Expressions (2)

- **Tie Breaker**
  - When an operand lies between two operators that have the same priority, the operand associates with the operator on the left.  
    \[ a + b - c \quad a * b / c / d \]

- **Delimiters**
  - Subexpression within delimiters is treated as a single operand, independent from the remainder of the expression.
    \[ (a + b) * (c - d) / (e - f) \]

- **Infix Expression Is Hard To Parse**
  - Need operator priorities, tie breaker, and delimiters.
  - This makes computer evaluation more difficult than is necessary.
  - Postfix and prefix expression forms do not rely on operator priorities, a tie breaker, or delimiters.
  - So it is easier for a computer to evaluate expressions that are in these forms.

- **Postfix Form**
  - The postfix form of a variable or constant is the same as its infix form.  
    \[ a, b, 3.25 \]
  - The relative order of operands is the same in infix and postfix forms.
  - Operators come immediately after the postfix form of their operands.
    \[ \text{Infix} = a + b \quad \text{Postfix} = ab+\]

- **Postfix Examples**
  - \[ \text{Infix} = a + b * c \quad \text{Postfix} = a b c * +\]
  - \[ \text{Infix} = a * b + c \quad \text{Postfix} = a b c * +\]
  - \[ \text{Infix} = (a + b) * (c - d) / (e + f) \quad \text{Postfix} = a b + c d - * e f + /\]

- **Unary Operators**
  - Replace with new symbols.
    \[ + a \Rightarrow a @ \quad + a + b \Rightarrow a @ b +\]
    \[ - a \Rightarrow a ? \quad - a - b \Rightarrow a ? b -\]
Postfix Evaluation

– Scan postfix expression from left to right pushing operands on to a stack.
– When an operator is encountered, pop as many operands as this operator needs; evaluate the operator; push the result on to the stack.
– This works because, in postfix, operators come immediately after their operands.
– Example: \((a + b) \times (c - d) / (e + f)\)

● Prefix Form
– The prefix form of a variable or constant is the same as its infix form. \(a, b, 3.25\)
– The relative order of operands is the same in infix and prefix forms.
– Operators come immediately before the prefix form of their operands. Infix = \(a + b\)  Postfix = \(ab+\)  Prefix = \(+ab\)

● Binary Tree Form
– \(a + b\)

\[
\begin{array}{c}
+ \\
/ \\
\end{array}
\]

\[a\]  \[b\]  \[-a\]

– \((a + b) \times (c - d) / (e + f)\)

\[
\begin{array}{c}
* \\
/ \\
\end{array}
\]

\[+\]  \[-\]  \[+\]

\[a\]  \[b\]  \[c\]  \[d\]  \[e\]  \[f\]

● Merits Of Binary Tree Form
– Left and right operands are easy to visualize.
– Code optimization algorithms work with the binary tree form of an expression.
– Simple recursive evaluation of expression.
Binary Tree Properties & Representation

- **Minimum Number Of Nodes**
  - Minimum number of nodes in a binary tree whose height is h. At least one node at each of first h levels.
  - Minimum number of nodes is h

- **Maximum Number Of Nodes**
  - All possible nodes at first h levels are present.
  - Maximum number of nodes = \(1 + 2 + 4 + 8 + \ldots + 2^{h-1} = 2^h - 1\)

- **Number Of Nodes & Height**
  - Let n be the number of nodes in a binary tree whose height is h.
  - \(h \leq n \leq 2^h - 1\)
  - \(\log_2(n+1) \leq h \leq n\)

- **Full Binary Tree**
  - A full binary tree of a given height h has \(2^h - 1\) nodes.

- **Numbering Nodes In A Full Binary Tree**
  - Number the nodes 1 through \(2^h - 1\).
  - Number by levels from top to bottom.
  - Within a level number from left to right.
  - **Node Number Properties**
    - Parent of node i is node \(i / 2\), unless i = 1.
    - Node 1 is the root and has no parent.
    - Left child of node i is node \(2i\), unless \(2i > n\), n is the number of nodes.
    - If \(2i > n\), node i has no left child.
    - Right child of node i is node \(2i+1\), unless \(2i+1 > n\), where n is the number of nodes.
    - If \(2i+1 > n\), node i has no right child.

- **Binary Tree Representation**
  - **Array Representation**
    - Number the nodes using the numbering scheme for a full binary tree. The node that is numbered i is stored in tree[i].
**Right-Skewed Binary Tree**
- An n node binary tree needs an array whose length is between n+1 and 2^n.

**Linked Representation**
- Each binary tree node is represented as an object whose data type is BinaryTreeNode.
- The space required by an n node binary tree is n * (space required by one node).

**The Class BinaryTreeNode**
```java
package dataStructures;
public class BinaryTreeNode{
    Object element;
    BinaryTreeNode leftChild; // left subtree
    BinaryTreeNode rightChild;// right subtree
    // constructors and any other methods come here
}
```

**Some Binary Tree Operations**
- Determine the height.
- Determine the number of nodes.
- Make a clone.
- Determine if two binary trees are clones.
- Display the binary tree.
- Evaluate the arithmetic expression represented by a binary tree.
- Obtain the infix form of an expression.
- Obtain the prefix form of an expression.
- Obtain the postfix form of an expression.
Binary Tree Traversal

- Many binary tree operations are done by performing a traversal of the binary tree.
- In a traversal, each element of the binary tree is visited exactly once.
- During the visit of an element, all action (make a clone, display, evaluate the operator, etc.) with respect to this element is taken.
- Preorder, Inorder, Postorder or Level order

**Preorder Traversal**

```java
public static void preOrder(BinaryTreeNode t)
{
    if (t != null)
    {
        visit(t);
        preOrder(t.leftChild);
        preOrder(t.rightChild);
    }
}
```

- Preorder Example (visit = print)
  a b d g h e i c f j
- Preorder Of Expression Tree
  `/ * + a b - c d + e f`
- Gives prefix form of expression!

**Inorder Traversal**

```java
public static void inOrder(BinaryTreeNode t)
{
    if (t != null)
    {
        inOrder(t.leftChild);
        visit(t);
        inOrder(t.rightChild);
    }
}
```

- Inorder Example (visit = print)
  g d h b e i a f j c
- Inorder By Projection (Squishing)
- Inorder Of Expression Tree
  `a + b * c - d / e + f`
- Gives infix form of expression (sans parentheses)!
public static void postOrder(BinaryTreeNode t)
{
    if (t != null)
    {
        postOrder(t.leftChild);
        postOrder(t.rightChild);
        visit(t);
    }
}

– Postorder Example (visit = print)      g h d i e b j f c a
– Postorder Of Expression Tree          a b + c d - * e f + /
– Gives postfix form of expression!

● Traversal Applications
  – Make a clone.      Determine height.      Determine # of nodes.

● Level Order
Let t be the tree root.
while (t != null)
{ visit t and put its children on a FIFO queue;
  remove a node from the FIFO queue and call it t;
} // remove returns null when queue is empty

– Level-Order Example (visit = print)     a b c d e f g h i j

● Binary Tree Construction
  – Suppose that the elements in a binary tree are distinct.
  – Can you construct the binary tree from which a given
    traversal sequence came?
    ● When a traversal sequence has more than one element, the
      binary tree is not uniquely defined.
    ● Therefore, the tree from which the sequence was obtained
      cannot be reconstructed uniquely.
  – Can you construct the binary tree, given two traversal
    sequences?
    ● Depends on which two sequences are given.
  – Preorder And Postorder
    ● preorder = ab           postorder = ba
    ● Preorder and postorder do not uniquely define a binary tree.
    ● Nor do preorder and level order (same example).
    ● Nor do postorder and level order (same example).
Binary Tree Construction

- **Inorder And Preorder**
  - inorder = g d h b e i a f j c
  - preorder = a b d g h e i c f j
  - Scan the preorder left to right using the inorder to separate left and right subtrees.
  - a is the root of the tree; gdhbei are in the left subtree; fjc are in the right subtree.
  - b is the next root; gdh are in the left subtree; ei are in the right subtree.
  - d is the next root; g is in the left subtree; h is in the right subtree.

- **Inorder And Postorder**
  - Scan postorder from right to left using inorder to separate left and right subtrees.
  - inorder = g d h b e i a f j c
  - postorder = g h d i e b j f c a
  - Tree root is a; gdhbei are in left subtree; fjc are in right subtree.

- **Inorder And Level Order**
  - Scan level order from left to right using inorder to separate left and right subtrees.
  - inorder = g d h b e i a f j c
  - level order = a b c d e f g h i j
  - Tree root is a; gdhbei are in left subtree; fjc are in right subtree.

- **Priority Queues: Two kinds of priority queues:**
  - **Min priority queue.**
    - Collection of elements.
    - Each element has a priority or key.
    - Supports following operations:
      - isEmpty
      - size
      - add/put an element into the priority queue
      - get element with min/max priority
      - remove element with min/max priority
Priority Queues

- **Applications**
  - **Sorting**
    - use element key as priority
    - put elements to be sorted into a priority queue
    - extract elements in priority order
      - if a min priority queue is used, elements are extracted in ascending order of priority (or key)
      - if a max priority queue is used, elements are extracted in descending order of priority (or key)
  - **Machine Scheduling**
    - m identical machines (drill press, cutter, sander, etc.)
    - n jobs/tasks to be performed
    - assign jobs to machines so that the time at which the last job completes is minimum
    - **Machine Scheduling Example**
      - 3 machines and 7 jobs
      - job times are [6, 2, 3, 5, 10, 7, 14]
      - possible schedule
      - Finish time = 21
      - Objective: Find schedules with minimum finish time.
    - **LPT Schedules**
      - Longest Processing Time first.
      - Jobs are scheduled in the order 14, 10, 7, 6, 5, 3, 2
      - Each job is scheduled on the machine on which it finishes earliest.
      - Finish time is 16!
NP-hard Problems

- LPT Schedule
  - LPT rule does not guarantee minimum finish time schedules.
  - \((\text{LPT Finish Time})/(\text{Minimum Finish Time}) \leq 4/3 - 1/(3m)\)
    where \(m\) is number of machines.
  - Usually LPT finish time is much closer to minimum finish time.
  - Minimum finish time scheduling is NP-hard.

- NP-hard Problems
  - Infamous class of problems for which no one has developed a polynomial time algorithm.
  - That is, no algorithm whose complexity is \(O(n^k)\) for any constant \(k\) is known for any NP-hard problem.
  - The class includes thousands of real-world problems.
  - Highly unlikely that any NP-hard problem can be solved by a polynomial time algorithm.
  - Since even polynomial time algorithms with degree \(k > 3\) (say) are not practical for large \(n\), we must change our expectations of the algorithm that is used.
  - Usually develop fast heuristics for NP-hard problems.
    - Algorithm that gives a solution close to best.
    - Runs in acceptable amount of time.
  - LPT rule is good heuristic for minimum finish time scheduling.

- Complexity Of LPT Scheduling
  - Sort jobs into decreasing order of task time.
    - \(O(n \log n)\) time (\(n\) is number of jobs)
  - Schedule jobs in this order.
    - assign job to machine that becomes available first
    - must find minimum of \(m\) (\(m\) is number of machines) finish times
    - takes \(O(m)\) time using simple strategy
    - so need \(O(mn)\) time to schedule all \(n\) jobs.

- Using A Min Priority Queue
  - Min priority queue has the finish times of the \(m\) machines.
  - Initial finish times are all 0.
  - To schedule a job remove machine with minimum finish time from the priority queue.
  - Update the finish time of the selected machine and put the machine back into the priority queue.
  - \(m\) put operations to initialize priority queue
  - 1 remove min and 1 put to schedule each job
  - each put and remove min operation takes \(O(\log m)\) time
  - time to schedule is \(O(n \log m)\)
  - overall time is \(O(n \log n + n \log m) = O(n \log (mn))\)
Tournament Trees

- **Winner Trees**
  - Complete binary tree with n external nodes and n - 1 internal nodes.
  - External nodes represent tournament players.
  - Each internal node represents a match played between its two children; the winner of the match is stored at the internal node.
  - Root has overall winner.
  - **Winner Tree For 16 Players**

- Smaller element wins => min winner tree.
- height is \( \log_2 n \) (excludes player level)
- **Complexity Of Initialize**
  - O(1) time to play match at each match node.
  - n - 1 match nodes.
  - O(n) time to initialize n player winner tree.
- **Applications Sorting.**
  - Put elements to be sorted into a winner tree.
  - Repeatedly extract the winner and replace by a large value.
  - Time To Sort
    - Initialize winner tree. O(n) time
    - Remove winner and replay. O(log n) time
    - Remove winner and replay n times. O(n log n) time
    - Total sort time is O(n log n).
    - Actually Theta(n log n).

- **Loser Tree:** Each match node stores the match loser rather than the match winner.
More Winner Tree Applications

- **Truck loading**
  - $n$ packages to be loaded into trucks
  - each package has a weight
  - each truck has a capacity of $c$ tons
  - minimize number of trucks
    - $n = 5$ packages, weights $[2, 5, 6, 3, 4]$, truck capacity $c = 10$
    - Load packages from left to right. If a package doesn’t fit into current truck, start loading a new truck.
    - uses 3 trucks when 2 trucks suffice
    - truck1 = $[2, 5, 3]$, truck2 = $[6, 4]$

- **Bin Packing**
  - $n$ items to be packed into bins, each item has a size
  - each bin has a capacity of $c$ tons, minimize number of bins
  - Truck loading is same as bin packing.
    - Truck is a bin that is to be packed (loaded).
    - Package is an item/element.
    - Bin packing to minimize number of bins is NP-hard.
    - Several fast heuristics have been proposed.

- **Bin Packing Heuristics**
  - **First Fit.**
    - Bins are arranged in left to right order.
    - Items are packed one at a time in given order.
    - Current item is packed into leftmost bin into which it fits.
    - If there is no bin into which current item fits, start a new bin.
      - $n = 4$ weights $=[4, 7, 3, 6]$ capacity $= 10$

  - **First Fit Decreasing.**
    - Items are sorted into decreasing order.
    - Then first fit is applied.

  - **Best Fit.**
    - Items are packed one at a time in given order.
    - To determine the bin for an item, first determine set $S$ of bins into which the item fits.
    - If $S$ is empty, then start a new bin and put item into this new bin.
    - Otherwise, pack into bin of $S$ that has least available capacity.
    - Use a max tournament tree in which the players are $n$ bins and the value of a player is the available capacity in the bin.
Binary Search Trees

- Complexity Of Dictionary Operations, get(), put() and remove(): n is the number of elements

<table>
<thead>
<tr>
<th>Data Structures</th>
<th>Worst Case</th>
<th>Expected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash Table</td>
<td>O(n)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Binary Search Tree</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Balanced Binary Search Tree</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
</tbody>
</table>

- Definition Of Binary Search Tree
  - A binary tree.
  - Each node has a (key, value) pair.
  - For every node x, all keys in the left subtree of x are smaller than that in x.
  - For every node x, all keys in the right subtree of x are greater than that in x.

- The Operation remove( ): Three cases:
  - Element is in a leaf.
  - Element is in a degree 1 node.
  - Element is in a degree 2 node.
    - Replace with largest key in left subtree (or smallest in right subtree).
    - Largest key must be in a leaf or degree 1 node.
Balanced Binary Search Trees

- height is $O(\log n)$, where $n$ is the number of elements in the tree
- AVL (Adelson-Velsky and Landis) trees
- get, put, and remove take $O(\log n)$ time
- Indexed AVL trees
- Indexed operations also take $O(\log n)$ time

- **AVL Tree**
  - binary tree, for every node $x$, define its balance factor
  - balance factor of $x = \text{height of left subtree of } x - \text{height of right subtree of } x$
  - balance factor of every node $x$ is -1, 0, or 1
  - The height of an AVL tree that has $n$ nodes is at most $1.44 \log_2 (n+2)$.
  - The height of every $n$ node binary tree is at least $\log_2 (n+1)$.
  - put (9)
  - put (29)
    - RR imbalance => new node is in right subtree of right subtree of “20” node (node with bf = -2)
    - RR rotation.

- **AVL Rotations**
  - RR
  - LL
  - RL
  - LR
AVL Rotations

- **LL**
  - ![LL Diagram]

- **RR**
  - ![RR Diagram]

- **LR**
  - ![LR Diagram]

- **RL**
  - ![RL Diagram]

- **LL & LR**
  - ![LL & LR Diagram]
Graphs

- $G = (V,E)$
- $V$ is the vertex set.
- Vertices are also called nodes and points.
- $E$ is the edge set.
- Each edge connects two different vertices.
- Edges are also called arcs and lines.
- Directed edge has an orientation $(u,v)$.
- Undirected edge has no orientation $(u,v)$.
- Undirected graph $\Rightarrow$ no oriented edge.
- Directed graph $\Rightarrow$ every edge has an orientation.

Applications

- **Communication Network:** Vertex = city, edge = communication link.
- **Driving Distance/Time Map**
  - Vertex = city, edge weight = driving distance/time.
- **Street Map:** Some streets are one way.

Complete Undirected Graph

- Has all possible edges.
- **Number Of Edges Undirected Graph**
  - Each edge is of the form $(u,v)$, $u \neq v$.
  - Number of such pairs in an $n$ vertex graph is $n(n-1)$.
  - Since edge $(u,v)$ is the same as edge $(v,u)$, the number of edges in a complete undirected graph is $n(n-1)/2$.
  - Number of edges in an undirected graph is $\leq n(n-1)/2$. 
Graph Operations And Representation

- **Number Of Edges--Directed Graph**
  - Each edge is of the form \((u,v), u != v\).
  - Number of such pairs in an \(n\) vertex graph is \(n(n-1)\).
  - Since edge \((u,v)\) is not the same as edge \((v,u)\), the number of edges in a complete directed graph is \(n(n-1)\).
  - Number of edges in a directed graph is \(\leq n(n-1)\).

- **Vertex Degree**
  - Number of edges incident to vertex.
  - Sum of degrees = \(2e\) (\(e\) is number of edges)
  - in-degree is number of incoming edges
  - out-degree is number of outbound edges
  - each edge contributes 1 to the in-degree of some vertex and 1 to the out-degree of some other vertex
  - sum of in-degrees = sum of out-degrees = \(e\), where \(e\) is the number of edges in the digraph

- **Sample Graph Problems**
  - Path problems.
  - Connectedness problems.
  - Spanning tree problems.
  - Path Finding: Path between 1 and 8. Path length is 20.
  - Another Path

Between 1 and 8:
Path length is 24.

- **Connected Graph**
  - Undirected graph.
  - There is a path between every pair of vertices.
  - Connected Components
Connected Component

- A maximal subgraph that is connected.
  - Cannot add vertices and edges from original graph and retain connectedness.
  - A connected graph has exactly 1 component.

- Communication Network Problems
  - Is the network connected?
    - Can we communicate between every pair of cities?
  - Find the components.
  - Want to construct smallest number of feasible links so that resulting network is connected.

- Cycles And Connectedness
  - Removal of an edge that is on a cycle does not affect connectedness.
  - Connected subgraph with all vertices and minimum number of edges has no cycles.
  - Tree
    - Connected graph that has no cycles.
    - n vertex connected graph with n-1 edges.
  - Spanning Tree
    - Subgraph that includes all vertices of the original graph.
    - Subgraph is a tree.
      - If original graph has n vertices, the spanning tree has n vertices and n-1 edges.
    - Minimum Cost Spanning Tree
      - Tree cost is sum of edge weights/costs.
A Spanning Tree

- Spanning tree cost = 46.
- Minimum Cost Spanning Tree: Spanning tree cost = 41.

Graph Representation

- Adjacency Matrix
  - 0/1 n x n matrix, where n = # of vertices
  - \( A(i,j) = 1 \) iff \((i,j)\) is an edge
  - Adjacency Matrix Properties
    - Diagonal entries are zero.
    - Adjacency matrix of an undirected graph is symmetric.
    - \( A(i,j) = A(j,i) \) for all \( i \) and \( j \).
  - Adjacency Matrix (Digraph)
    - Diagonal entries are zero.
    - Adjacency matrix of a digraph need not be symmetric.
  - \( n^2 \) bits of space
  - For an undirected graph, may store only lower or upper triangle (exclude diagonal). \((n-1)n/2\) bits
  - \( O(n) \) time to find vertex degree and/or vertices adjacent to a given vertex.

- Adjacency Lists
  - Linked Adjacency Lists
  - Array Adjacency Lists

\[
\begin{array}{cccccc}
1 & 0 & 1 & 0 & 1 & 0 \\
2 & 1 & 0 & 0 & 0 & 1 \\
3 & 0 & 0 & 0 & 0 & 1 \\
4 & 1 & 0 & 0 & 0 & 1 \\
5 & 0 & 1 & 1 & 1 & 0 \\
\end{array}
\]
Adjacency Lists

- Adjacency list for vertex i is a linear list of vertices adjacent from vertex i.
- An array of n adjacency lists.

**Linked Adjacency Lists**
- Each adjacency list is a chain.
- Each adjacency list is an array list.
- Array Length = n
  - # of chain nodes = 2e (undirected graph)
  - # of chain nodes = e (digraph)

**Weighted Graphs**
- Cost adjacency matrix. \( C(i,j) = \text{cost of edge } (i,j) \)
- Adjacency lists => each list element is a pair (adjacent vertex, edge weight)

**Graph Search Methods**
- A vertex u is reachable from vertex v iff there is a path from v to u.
- A search method starts at a given vertex v and visits/labels/marks every vertex that is reachable from v.
- Many graph problems solved using a search method.
  - Path from one vertex to another.
  - Is the graph connected?
  - Find a spanning tree., Etc.
- Commonly used search methods: BFS, DFS
- Breadth-first search.
  - Visit start vertex and put into a FIFO queue.
  - Repeatedly remove a vertex from the queue, visit its unvisited adjacent vertices, put newly visited vertices into the queue.
  - Breadth-First Search Property
    - All vertices reachable from the start vertex (including the start vertex) are visited.
  - Time Complexity
    - Each visited vertex is put on (and so removed from) the queue exactly once.
    - When a vertex is removed from the queue, we examine its adjacent vertices.
      - \( O(n) \) if adjacency matrix used
      - \( O(\text{vertex degree}) \) if adjacency lists used
Commonly used search methods: BFS, DFS

- **Time Complexity**
  - Total time: \( O(mn) \), where \( m \) is number of vertices in the component that is searched (adjacency matrix)
  - \( O(n + \text{sum of component vertex degrees}) \) (adj. lists)
    \[ = O(n + \text{number of edges in component}) \]

- **Path From Vertex \( u \) To Vertex \( v \)**
  - Start a breadth-first search at vertex \( u \).
  - Terminate when vertex \( v \) is visited or when \( Q \) becomes empty (whichever occurs first).

- **Is The Graph Connected?**
  - Start a breadth-first search at any vertex of the graph.
  - Graph is connected iff all \( n \) vertices get visited.

- **Connected Components**
  - Start a breadth-first search at any as yet unvisited vertex of the graph.
  - Newly visited vertices (plus edges between them) define a component.
  - Repeat until all vertices are visited.

- **Spanning Tree**
  - Start a breadth-first search at any vertex of the graph.
  - If graph is connected, the \( n-1 \) edges used to get to unvisited vertices define a spanning tree (breadth-first spanning tree).

- **Time:** \( O(n^2) \) when adjacency matrix used
- \( O(n+e) \) when adjacency lists used (\( e \) is number of edges)

### Depth-first search.

```java
depthFirstSearch(v)
{
    Label vertex \( v \) as reached.
    for (each unreached vertex \( u \) adjacent from \( v \))
    {
        depthFirstSearch(u);
    }
}
```

- **Depth-First Search Properties**
  - Same complexity as BFS.
  - Same properties with respect to path finding, connected components, and spanning trees.
  - Edges used to reach unlabeled vertices define a depth-first spanning tree when the graph is connected.
  - There are problems for which bfs is better than dfs and vice versa.
Algorithm Design Methods

– Greedy method.
– Divide and conquer.
– Dynamic Programming.
– Backtracking.
– Branch and bound.

• Others
  – Linear Programming.
  – Integer Programming.
  – Simulated Annealing.
  – Neural Networks.
  – Genetic Algorithms.
  – Tabu Search.

• Optimization Problem
  – A problem in which some function (called the optimization or objective function) is to be optimized (usually minimized or maximized) subject to some constraints.
    – Machine Scheduling
      • Find a schedule that minimizes the finish time.
      • optimization function … finish time
      • constraints
        – each job is scheduled continuously on a single machine for an amount of time equal to its processing requirement
        – no machine processes more than one job at a time
    – Bin Packing
      • Pack items into bins using the fewest number of bins.
      • optimization function … number of bins
      • constraints
        – each item is packed into a single bin
        – the capacity of no bin is exceeded
    – Min Cost Spanning Tree
      • Find a spanning tree that has minimum cost.
      • optimization function … sum of edge costs
      • constraints
        – must select n-1 edges of the given n vertex graph
        – the selected edges must form a tree
Feasible And Optimal Solutions

– A feasible solution is a solution that satisfies the constraints.
– An optimal solution is a feasible solution that optimizes the objective/optimization function.

● Greedy Method
– Solve problem by making a sequence of decisions.
– Decisions are made one by one in some order.
– Each decision is made using a greedy criterion.
– A decision, once made, is (usually) not changed later.
– Machine Scheduling: LPT Scheduling.
  ● Schedule jobs one by one and in decreasing order of processing time.
  ● Each job is scheduled on the machine on which it finishes earliest.
  ● Scheduling decisions are made serially using a greedy criterion (minimize finish time of this job).
  ● LPT scheduling is an application of the greedy method.
  ● LPT rule does not guarantee minimum finish time schedules.
  ● \((\text{LPT Finish Time})/(\text{Minimum Finish Time}) \leq 4/3 - 1/(3m)\)
    where \(m\) is number of machines
  ● Minimum finish time scheduling is NP-hard.
  ● In this case, the greedy method does not work.
  ● Greedy method does, however, give us a good heuristic for machine scheduling.

– Container Loading
  ● Ship has capacity \(c\).
  ● \(m\) containers are available for loading.
  ● Weight of container \(i\) is \(w_i\).
  ● Each weight is a positive number.
  ● Sum of container weights \(> c\).
  ● Load as many containers as is possible without sinking the ship.
  ● Greedy Solution
    – Load containers in increasing order of weight until we get to a container that doesn’t fit.
    – Does this greedy algorithm always load the maximum number of containers? Yes.
Divide And Conquer

– A large instance is solved as follows:
  • Divide the large instance into smaller instances.
  • Solve the smaller instances somehow.
  • Combine the results of the smaller instances to obtain the result for the original large instance.
– A small instance is solved in some other way.
– Small instance.
  • Sort a list that has \( n \leq 10 \) elements.
  • Find the minimum of \( n \leq 2 \) elements.
– Large instance.
  • Sort a list that has \( n > 10 \) elements.
  • Find the minimum of \( n > 2 \) elements.

● Sort A Large List
  – Sort a list that has \( n > 10 \) elements.
    • Sort 15 elements by dividing them into 2 smaller lists.
      – One list has 7 elements and the other has 8.
    • Sort these two lists using the method for small lists.
    • Merge the two sorted lists into a single sorted list.

● Find The Min Of A Large List
  – Find the minimum of \( 20 \) elements.
    • Divide into two groups of 10 elements each.
    • Find the minimum element in each group somehow.
    • Compare the minimums of each group to determine the overall minimum.

● Recursion In Divide And Conquer
  – Often the smaller instances that result from the divide step are instances of the original problem (true for our sort and min problems). In this case,
    • If the new instance is a small instance, it is solved using the method for small instances.
    • If the new instance is a large instance, it is solved using the divide-and-conquer method recursively.
  – Generally, performance is best when the smaller instances that result from the divide step are of approximately the same size.
Dynamic Programming

- Sequence of decisions.
  - As in the greedy method, the solution to a problem is viewed as the result of a sequence of decisions.
  - Unlike the greedy method, decisions are not made in a greedy and binding manner.

- Problem state.

- Principle of optimality.
  - An optimal solution satisfies the following property:
  - No matter what the first decision, the remaining decisions are optimal with respect to the state that results from this decision.

- Dynamic programming may be used only when the principle of optimality holds.

- Dynamic Programming Recurrence

- Steps.
  - View the problem solution as the result of a sequence of decisions.
  - Obtain a formulation for the problem state.
  - Verify that the principle of optimality holds.
  - Set up the dynamic programming recurrence equations.
  - Solve these equations for the value of the optimal solution.
  - Perform a traceback to determine the optimal solution.

- When solving the dynamic programming recurrence recursively, be sure to avoid the recomputation of the optimal value for the same problem state.

- To minimize run time overheads, and hence to reduce actual run time, dynamic programming recurrences are almost always solved iteratively (no recursion).

- Matrix Multiplication Chains
  - Multiply an m x n matrix A and an n x p matrix B to get an m x p matrix C.
    \[
    C(i,j) = \sum_{k=1}^{n} A(i,k) \times B(k,j)
    \]

  - We shall use the number of multiplications as our complexity measure.
  - n multiplications are needed to compute one \(C(i,j)\).
Matrix Multiplication Chains

- mnp multiplicatons are needed to compute all mp terms of C.
- Suppose that we are to compute the product X*Y*Z of three matrices X, Y and Z.

The matrix dimensions are: X:(100 x 1), Y:(1 x 100), Z:(100 x 1)
- Multiply X and Y to get a 100 x 100 matrix T.
  - 100 * 1 * 100 = 10,000 multiplications.
- Multiply T and Z to get the 100 x 1 answer.
  - 100 * 100 * 1 = 10,000 multiplications.
- Total cost is 20,000 multiplications.
  - 10,000 units of space are needed for T.

The matrix dimensions are: X:(100 x 1), Y:(1 x 100), Z:(100 x 1)
- Multiply Y and Z to get a 1 x 1 matrix T.
  - 1 * 100 * 1 = 100 multiplications.
- Multiply X and T to get the 100 x 1 answer.
  - 100 * 1 * 1 = 100 multiplications.
- Total cost is 200 multiplications.
  - 1 unit of space is needed for T.

Product Of 5 Matrices
- Some of the ways in which the product of 5 matrices may be computed.
  - A*(B*(C*(D*E)))  right to left
  - (((A*B)*C)*D)*E  left to right
  - (A*B)*((C*D)*E)  (A*B)*(C*(D*E))
  - (A*(B*C))*(D*E)  ((A*B)*C)*(D*E)

Find Best Multiplication Order
- Number of ways to compute the product of q matrices is $O(4^q/q^{1.5})$.
- Evaluating all ways to compute the product takes $O(4^q/q^{0.5})$ time.
- Determine the best way to compute the matrix product $M_1 x M_2 x M_3 x \ldots x M_q$.
- Let the dimensions of $M_i$ be $r_i x r_{i+1}$.
- q-1 matrix multiplications are to be done.
- Decide the matrices involved in each of these multiplications.
Matrix Multiplication Chains (2)

- **Decision Sequence**
  - Determine the q-1 matrix products in reverse order.
    - What is the last multiplication?
    - What is the next to last multiplication? And so on.

- **Problem State**
  - The matrices involved in each multiplication are a contiguous subset of the given q matrices.
  - The problem state is given by a set of pairs of the form (i, j), i <= j.
    - The pair (i,j) denotes a problem in which the matrix product $M_i \times M_{i+1} \times \ldots \times M_j$ is to be computed.
    - The initial state is (1,q).
    - If the last matrix product is $(M_1 \times M_2 \times \ldots \times M_k) \times (M_{k+1} \times M_{k+2} \times \ldots \times M_q)$, the state becomes $\{(1,k), (k+1,q)\}$.

- **Verify Principle Of Optimality**
  - Let $M_{ij} = M_i \times M_{i+1} \times \ldots \times M_j$, i <= j.
  - Suppose that the last multiplication in the best way to compute $M_{ij}$ is $M_{ik} \times M_{k+1,j}$ for some k, i <= k <= j.
  - Irrespective of what k is, a best computation of $M_{ij}$ in which the last product is $M_{ik} \times M_{k+1,j}$ has the property that $M_{ik}$ and $M_{k+1,j}$ are computed in the best possible way.
  - So the principle of optimality holds and dynamic programming may be applied.

- **Recurrence Equations**
  - Let $c(i,j)$ be the cost of an optimal (best) way to compute $M_{ij}$, i <= j.
  - $c(1,q)$ is the cost of the best way to multiply the given q matrices.
  - Let $kay(i,j) = k$ be such that the last product in the optimal computation of $M_{ij}$ is $M_{ik} \times M_{k+1,j}$.
  - $c(i,i) = 0$, 1 <= i <= q. ($M_{ii} = M_i$)
  - $c(i,i+1) = r_ir_{i+1}r_{i+2}$, 1 <= i < q. ($M_{ii+1} = M_i \times M_{i+1}$)
  - $kay(i,i+1) = i$.

$$c(i, i+s), 1 < s < q$$
Recurrence Equations

- The last multiplication in the best way to compute $M_{i,i+s}$ is $M_{ik} \times M_{k+1,i+s}$ for some $k$, $i \leq k < i+s$.
- If we knew $k$, we could claim:
  - $c(i,i+s) = c(i,k) + c(k+1,i+s) + r_i r_{k+1} r_{i+s+1}$
  - Since $i \leq k < i+s$, we can claim
  - $c(i,i+s) = \min\{c(i,k) + c(k+1,i+s) + r_i r_{k+1} r_{i+s+1}\}$, where the min is taken over $i \leq k < i+s$.
  - $kay(i,i+s)$ is the $k$ that yields above min.
  - $c(i,i+s) = \min\{c(i,k) + c(k+1,i+s) + r_i r_{k+1} r_{i+s+1}\}$, where $i \leq k < i+s$.
- $c(\ast,\ast)$ terms on right side involve fewer matrices than does the $c(\ast,\ast)$ term on the left side.
- So compute in the order $s = 2, 3, \ldots, q-1$.

Example
- $q = 4$, $(10 \times 1) \times (1 \times 10) \times (10 \times 1) \times (1 \times 1)$
- $r = [r_1, r_2, r_3, r_4, r_5] = [10, 1, 10, 1, 10]$
- $s = 0$, $c(i,i)$ and $kay(i,i)$, $1 \leq i \leq 4$ are to be computed.
- $s = 1$ $c(i,i+1)$ and $kay(i,i+1)$, $1 \leq i \leq 3$ are to be computed.
  - $c(i,i+1) = r_i r_{i+1} r_{i+2}$, $1 < i < q$. ($M_{i+1} = M_i \times M_{i+1}$)
  - $kay(i,i+1) = i$.
- $s = 2$ $c(i,i+2) = \min\{c(i,i) + c(i+1,i+2) + r_i r_{i+1} r_{i+3}$,
  - $c(i,i+1) + c(i+2,i+2) + r_i r_{i+2} r_{i+3}\}$
  - $c(1,3) = \min\{c(1,1) + c(2,3) + r_1 r_2 r_4, c(1,2) + c(3,3) + r_1 r_3 r_4\}$
  - $c(1,3) = \min\{0 + 10 + 100, 0 + 100\}$
  - $c(2,4) = \min\{c(2,2) + c(3,4) + r_2 r_3 r_5, c(2,3) + c(4,4) + r_2 r_4 r_5\}$
  - $c(2,4) = \min\{0 + 100 + 100, 10 + 0 + 10\}$
- $s = 3$ $c(1,4) = \min\{c(1,1) + c(2,4) + r_1 r_2 r_5, c(1,2) + c(3,4) + r_1 r_3 r_5, c(1,3) + c(4,4) + r_1 r_4 r_5\}$
  - $c(1,4) = \min\{0+20+100, 100+100+1000, 20+0+100\}$
Example

\[
c(i,j), \ i \leq j
\]

- **Determine The Best Way To Compute** \(M_{14}\)
  - \(kay(1,4) = 1\).
  - So the last multiplication is \(M_{14} = M_{11} \times M_{24}\).
  - \(M_{11}\) involves a single matrix and no multiply.
  - Find best way to compute \(M_{24}\).

- **Determine The Best Way To Compute** \(M_{24}\)
  - \(kay(2,4) = 3\).
  - So the last multiplication is \(M_{24} = M_{23} \times M_{44}\).
  - \(M_{23}\) = \(M_{22} \times M_{33}\).
  - \(M_{44}\) involves a single matrix and no multiply.

- **The Best Way To Compute** \(M_{14}\)
  - The multiplications (in reverse order) are:
    - \(M_{14} = M_{11} \times M_{24}\)
    - \(M_{24} = M_{23} \times M_{44}\)
    - \(M_{23} = M_{22} \times M_{33}\)

- **Time Complexity**
  - \(O(q^2)\) \(c(i,j)\) values are to be computed, where \(q\) is the number of matrices.
  - \(c(i,i+s) = \min_{i \leq k < i+s} \{c(i,k) + c(k+1,i+s) + r_i r_{k+1} r_{i+s+1}\}\.)
  - Each \(c(i,j)\) is the min of \(O(q)\) terms.
  - Each of these terms is computed in \(O(1)\) time.
  - So all \(c(i,j)\) are computed in \(O(q^3)\) time.
  - The traceback takes \(O(1)\) time to determine each matrix product that is to be done.
  - \(q-1\) products are to be done.
  - Traceback time is \(O(q)\).
All-Pairs Shortest Paths

- Given an n vertex directed weighted graph, find a shortest path from vertex i to vertex j for each for the n^2 vertex pairs (i,j).

- Dijkstra’s Single Source Algorithm
  - Use Dijkstra’s algorithm n times, once with each of the n vertices as the source vertex.
  - Performance
    - Time complexity is O(n^3) time.
    - Works only when no edge has a cost < 0.

- Dynamic Programming Solution
  - Time complexity is O(n^3) time.
  - When there is a cycle whose length is < 0, some shortest paths aren’t finite.
  - If vertex 1 is on a cycle whose length is -2, each time you go around this cycle once you get a 1 to 1 path that is 2 units shorter than the previous one.
  - Simpler to code, smaller overheads.
  - Known as Floyd’s shortest paths algorithm.

- Decision Sequence
  - First decide the highest intermediate vertex (i.e., largest vertex number) on the shortest path from i to j.
Dynamic Programming Solution
– If the shortest path is 1, 2, 6, 3, 8, 5, 7, j the first decision is that vertex 8 is an intermediate vertex on the shortest path and no intermediate vertex is larger than 8.
– Then decide the highest intermediate vertex on the path from i to 8, and so on.

Problem State
– (i,j,k) denotes the problem of finding the shortest path from vertex i to vertex j that has no intermediate vertex larger than k.
– (i,j,n) denotes the problem of finding the shortest path from vertex i to vertex j (with no restrictions on intermediate vertices).

Cost Function
– Let c(i,j,k) be the length of a shortest path from vertex i to vertex j that has no intermediate vertex larger than k.
– c(i,j,n)
  – c(i,j,n) is the length of a shortest path from vertex i to vertex j that has no intermediate vertex larger than n.
  – No vertex is larger than n.
  – Therefore, c(i,j,n) is the length of a shortest path from vertex i to vertex j.
– c(i,j,0)
  – c(i,j,0) is the length of a shortest path from vertex i to vertex j that has no intermediate vertex larger than 0.
    – Every vertex is larger than 0.
    – Therefore, c(i,j,0) is the length of a single-edge path from vertex i to vertex j.

Recurrence For c(i,j,k), k > 0
– The shortest path from vertex i to vertex j that has no intermediate vertex larger than k may or may not go through vertex k.
– If this shortest path does not go through vertex k, the largest permissible intermediate vertex is k-1. So the path length is c(i,j,k-1).
Recurrence Equations

- Shortest path goes through vertex \( k \).
- We may assume that vertex \( k \) is not repeated because no cycle has negative length.
- Largest permissible intermediate vertex on \( i \) to \( k \) and \( k \) to \( j \) paths is \( k-1 \).
- \( i \) to \( k \) path must be a shortest \( i \) to \( k \) path that goes through no vertex larger than \( k-1 \).
- If not, replace current \( i \) to \( k \) path with a shorter \( i \) to \( k \) path to get an even shorter \( i \) to \( j \) path.
- Similarly, \( k \) to \( j \) path must be a shortest \( k \) to \( j \) path that goes through no vertex larger than \( k-1 \).
- Therefore, length of \( i \) to \( k \) path is \( c(i,k,k-1) \), and length of \( k \) to \( j \) path is \( c(k,j,k-1) \).
- So, \( c(i,j,k) = c(i,k,k-1) + c(k,j,k-1) \).
- Combining the two equations for \( c(i,j,k) \), we get \( c(i,j,k) = \min\{c(i,j,k-1), c(i,k,k-1) + c(k,j,k-1)\} \).
- We may compute the \( c(i,j,k) \)s in the order \( k = 1, 2, 3, \ldots, n \).

Floyd’s Shortest Paths Algorithm

- for (int \( k = 1; k <= n; k++ \))
  - for (int \( i = 1; i <= n; i++ \))
    - for (int \( j = 1; j <= n; j++ \))
      - \( c(i,j,k) = \min\{c(i,j,k-1), c(i,k,k-1) + c(k,j,k-1)\} \);
- Time complexity is \( O(n^3) \).
- More precisely \( \Theta(n^3) \).
- \( \Theta(n^3) \) space is needed for \( c(*,*,*). \)
Space Reduction

- \( c(i,j,k) = \min\{c(i,j,k-1), c(i,k,k-1) + c(k,j,k-1)\} \)
- When neither \( i \) nor \( j \) equals \( k \), \( c(i,j,k-1) \) is used only in the computation of \( c(i,j,k) \).

- So \( c(i,j,k) \) can overwrite \( c(i,j,k-1) \).
- \( c(i,j,k) = \min\{c(i,j,k-1), c(i,k,k-1) + c(k,j,k-1)\} \)
- When \( i \) equals \( k \), \( c(i,j,k-1) \) equals \( c(i,j,k) \).
  - \( c(k,j,k) = \min\{c(k,j,k-1), c(k,k,k-1) + c(k,j,k-1)\} = \min\{c(k,j,k-1), 0 + c(k,j,k-1)\} = c(k,j,k-1) \)
  - So, when \( i \) equals \( k \), \( c(i,j,k) \) can overwrite \( c(i,j,k-1) \).
  - Similarly when \( j \) equals \( k \), \( c(i,j,k) \) can overwrite \( c(i,j,k-1) \).
  - So, in all cases \( c(i,j,k) \) can overwrite \( c(i,j,k-1) \).

**Floyd’s Shortest Paths Algorithm**

- for (int \( k = 1; k <= n; k++ \))
  - for (int \( i = 1; i <= n; i++ \))
    - for (int \( j = 1; j <= n; j++ \))
      - \( c(i,j) = \min\{c(i,j), c(i,k) + c(k,j)\}; \)
  - Initially, \( c(i,j) = c(i,j,0) \).
  - Upon termination, \( c(i,j) = c(i,j,n) \).
  - Time complexity is \( \Theta(n^3) \).
  - \( \Theta(n^2) \) space is needed for \( c(\cdot,\cdot) \).

**Building The Shortest Paths**

- Let \( \kay(i,j) \) be the largest vertex on the shortest path from \( i \) to \( j \).
- Initially, \( \kay(i,j) = 0 \) (shortest path has no intermediate vertex).

for (int \( k = 1; k <= n; k++ \))
  - for (int \( i = 1; i <= n; i++ \))
    - for (int \( j = 1; j <= n; j++ \))
      - if \( (c(i,j) > c(i,k) + c(k,j)) \)
        - \( \{\kay(i,j) = k; c(i,j) = c(i,k) + c(k,j);\} \)
Example

-  7  5  1  -  -  -  -  
-  -  -  -  4  -  -  -  
-  7  -  -  9  9  -  -  
-  5  -  -  -  16  -  
-  -  -  4  -  -  -  1  
-  -  -  -  -  1  -  
2  -  -  -  -  -  -  4  
-  -  -  -  -  2  4  -  

--- Initial Cost Matrix ---
c(\*,\*) = c(\*,\*,0)

--- Final Cost Matrix c(\*,\*) = c(\*,\*,n) === kay Matrix ---
0  6  5  1  10  13  14  11  0 4 0 0 4 8 8 5
10  0  15  8  4  7  8  5  8 0 8 5 0 8 8 5
12  7  0  13  9  9  10  10  7 0 5 0 0 6 5
15  5  20  0  9  12  13  10  8 0 8 0 2 8 8 5
6  9  11  4  0  3  4  1  8 4 8 0 0 8 8 0
3  9  8  4  13  0  1  5  7 7 7 7 7 0 0 7
2  8  7  3  12  6  0  4  0 4 1 1 4 8 0 0
5  11 10  6  15  2  3  0  7 7 7 7 7 0 6 0

- Shortest Path
  - Shortest path from 1 to 7.
  - Path length is 14.
Build A Shortest Path

The path is 1 → 4 → 2 → 5 → 8 → 6 → 7.

- \( \kappa(1,7) = 8 \)
- \( \kappa(1,8) = 5 \)
- \( \kappa(1,5) = 4 \)
- \( \kappa(1,4) = 0 \)
- \( \kappa(4,5) = 2 \)
- \( \kappa(4,2) = 0 \)
- \( \kappa(2,5) = 0 \)
- \( \kappa(5,8) = 0 \)
- \( \kappa(8,7) = 6 \)
- \( \kappa(8,6) = 0 \)
- \( \kappa(6,7) = 0 \)
Backtracking & Branch And Bound