What is a data structure?

- A table of data including structural relationships
  - Donald Knuth (Turing award ’74)

- Algorithms + data structures = programs
  - Niklaus Wirth (Turing award ’84)

Data structures

- Programs use data
- Data needs to be stored and accessed
- Efficiently
- Different applications have different requirements

Course contents (short)

- List structures, stacks, queues
- Trees
- Sorting
- Heaps, priority queues
- Hashing
- Graphs
- Introduction to algorithms

Course contents (long)

- List structures
- Stacks
- Queues
- Implementation issues
  - Arrays, linked lists
  - Doubly linked lists
  - Dynamic or static
- Recursion
Course contents (long)

- Trees
- Binary trees
- Balanced trees
  - 2-3 trees, red-black
- Binary search trees
- Recursion

Course contents (long)

- Tables
  - Items with a search key
- Priority queues
- Heaps and heapsort
- Hashing
- Hash functions and hash tables

Course contents (long)

- Sorting
- Selection and insertion sort
- Bubblesort
- Mergesort
- Quicksort
- Heapsort
- Comparison of sorting algorithms

Course contents (long)

- Graphs
- Representations
- Graph traversals
- Topological sorting
- Spanning trees
- Shortest paths

Course contents (long)

- Advanced Data structures
- Binomial queues
- Treaps
- Disjoint sets/union find
- Fibonacci heaps
- Analysis of data structures

Abstract Data Types

- The isolation of modules is not total
  - Methods' specifications, or contracts, govern how they interact with each other

Figure 3.2
A slit in the wall
Abstract Data Types

- Typical operations on data
  - Add data to a data collection
  - Remove data from a data collection
  - Ask questions about the data in a data collection
- Data abstraction
  - Asks you to think what you can do to a collection of data independently of how you do it
  - Allows you to develop each data structure in relative isolation from the rest of the solution
  - A natural extension of procedural abstraction

Abstract Data Types

- Abstract data type (ADT)
  - An ADT is composed of
    - A collection of data
    - A set of operations on that data
  - Specifications of an ADT indicate
    - What the ADT operations do, not how to implement them
  - Implementation of an ADT
    - Includes choosing a particular data structure

Abstract Data Types

- Data structure
  - A construct that is defined within a programming language to store a collection of data
  - Example: arrays
- ADTs and data structures are not the same
- Data abstraction
  - Results in a wall of ADT operations between data structures and the program that accesses the data within these data structures

Specifying ADTs

- In a list
  - Except for the first and last items, each item has
    - A unique predecessor
    - A unique successor
    - Head or front
    - Tail or end
    - Does not have a predecessor
    - Does not have a successor

The ADT List

- ADT List operations
  - Create an empty list
  - Determine whether a list is empty
  - Determine the number of items in a list
  - Add an item at a given position in the list
  - Remove the item at a given position in the list
  - Remove all the items from the list
  - Retrieve (get) the item at a given position in the list
- Items are referenced by their position within the list
The ADT Sorted List

- The ADT sorted list
  - Maintains items in sorted order
  - Inserts and deletes items by their values, not their positions

An Array-Based Implementation of the ADT List

- An array-based implementation
  - A list's items are stored in an array \( \text{items} \)
  - A natural choice
    - Both an array and a list identify their items by number
    - A list's \( k \)-th item will be stored in \( \text{items}[k-1] \)

An Array-Based Implementation of the ADT List

- An array-based implementation
  - A list's items are stored in an array \( \text{items} \)
  - A natural choice
    - Both an array and a list identify their items by number
    - A list's \( k \)-th item will be stored in \( \text{items}[k-1] \)

List implementations

- Options for implementing an ADT
  - Array
    - Has a fixed size
    - Data must be shifted during insertions and deletions
  - Linked list
    - Is able to grow in size as needed
    - Does not require the shifting of items during insertions and deletions

Preliminaries

- Options for implementing an ADT
  - Array
    - Has a fixed size
    - Data must be shifted during insertions and deletions
  - Linked list
    - Is able to grow in size as needed
    - Does not require the shifting of items during insertions and deletions
ADT Stack

- ADT stack operations
  - Create an empty stack
  - Determine whether a stack is empty
  - Add a new item to the stack
  - Remove from the stack the item that was added most recently
  - Remove all the items from the stack
  - Retrieve from the stack the item that was added most recently

Stacks

- A stack
  - Last-in, first-out (LIFO) property
    - The last item placed on the stack will be the first item removed

Implementations of the ADT Stack

- The ADT stack can be implemented using
  - An array
  - A linked list
  - The ADT list
- StackInterface
  - Provides a common specification for the three implementations
- StackException
  - Used by StackInterface
  - Extends java.lang.RuntimeException

Stack Interface

```java
public interface StackInterface {
    public boolean isEmpty();
    public void popAll();
    public void push(Object newItem) throws StackException;
    public Object pop() throws StackException;
    public Object peek() throws StackException;
}
```  // end StackInterface

Implementations of the ADT Stack

Figure 6.3

An Array-Based Implementation of the ADT Stack

- StackArrayBased class
  - Implements StackInterface
  - Instances
    - Stacks
      - Private data fields
        - An array of Objects called items
        - The index top
  - an ADT list
An Implementation That Uses the ADT List

- The ADT list can be used to represent the items in a stack
- If the item in position 1 of a list represents the top of the stack
  - `push(newItem)` operation is implemented as `add(1, newItem)`
  - `pop()` operation is implemented as `get(1)`
  - `remove(1)`
  - `peek()` operation is implemented as `get(1)`

Simple Applications of the ADT Stack: Checking for Balanced Braces

- A stack can be used to verify whether a program contains balanced braces
  - An example of balanced braces: `abc(defgijklmn)opqr`
  - An example of unbalanced braces: `abc(def))ghijklm`

The Relationship Between Stacks and Recursion

- The ADT stack has a hidden presence in the concept of recursion
- Typically, stacks are used by compilers to implement recursive methods
  - During execution, each recursive call generates an activation record that is pushed onto a stack
- Stacks can be used to implement a nonrecursive version of a recursive algorithm

The Abstract Data Type Queue

- A queue
  - New items enter at the back, or rear, of the queue
  - Items leave from the front of the queue
  - First-in, first-out (FIFO) property
    - The first item inserted into a queue is the first item to leave
The Abstract Data Type Queue

- ADT queue operations
  - Create an empty queue
  - Determine whether a queue is empty
  - Add a new item to the queue
  - Remove from the queue the item that was added earliest
  - Remove all the items from the queue
  - Retrieve from the queue the item that was added earliest

Queues
- Are appropriate for many real-world situations
  - Example: A line to buy a movie ticket
  - Have many applications in computer science
  - Example: A request to print a document
- A simulation
  - A study to see how to reduce the wait involved in an application

Pseudocode for the ADT queue operations

```java
createQueue()
// Creates an empty queue.

isEmpty()
// Determines whether a queue is empty

enqueue(newItem) throws QueueException
// Adds newItem at the back of a queue. Throws
// QueueException if the operation is not
// successful
```

Pseudocode for the ADT queue operations (Continued)

```java
dequeue() throws QueueException
// Retrieves and removes the front of a queue.
// Throws QueueException if the operation is
// not successful.

dequeueAll()
// Removes all items from a queue

peek() throws QueueException
// Retrieves the front of a queue. Throws
// QueueException if the retrieval is not
// successful
```

Possible implementations of a queue
- A linear linked list with two external references
  - A reference to the front
  - A reference to the back

Possible implementations of a queue (Continued)
- A circular linked list with one external reference
  - A reference to the back
A Reference-Based Implementation

An Array-Based Implementation

Figure 7.4
Inserting an item into a nonempty queue

An Array-Based Implementation

Figure 7.7
a) A naive array-based implementation of a queue; b) rightward drift can cause the queue to appear full

An Array-Based Implementation

Figure 7.8
A circular implementation of a queue

An Array-Based Implementation

Figure 7.9
The effect of some operations of the queue in Figure 7-8

Determining the Efficiency of Algorithms

• Analysis of algorithms
  – Provides tools for contrasting the efficiency of different methods of solution
• A comparison of algorithms
  – Should focus of significant differences in efficiency
  – Should not consider reductions in computing costs due to clever coding tricks

Determining the Efficiency of Algorithms

• Three difficulties with comparing programs instead of algorithms
  – How are the algorithms coded?
  – What computer should you use?
  – What data should the programs use?
• Algorithm analysis should be independent of
  – Specific implementations
  – Computers
  – Data
Algorithm Growth Rates

- An algorithm’s time requirements can be measured as a function of the problem size
- An algorithm’s growth rate
  - Enables the comparison of one algorithm with another
  - Examples
    - Algorithm A requires time proportional to $n^2$
    - Algorithm B requires time proportional to $n$
- Algorithm efficiency is typically a concern for large problems only

Order-of-Magnitude Analysis and Big O Notation

- Definition of the order of an algorithm
  - Algorithm A is order $f(n)$ – denoted $O(f(n))$ – if constants $k$ and $n_0$ exist such that $A$ requires no more than $k \cdot f(n)$ time units to solve a problem of size $n \geq n_0$
- Growth-rate function
  - A mathematical function used to specify an algorithm’s order in terms of the size of the problem
  - Big O notation
    - Example: $O(n^2)$
    - A notation that uses the capital letter O to specify an algorithm’s order

Order-of-Magnitude Analysis and Big O Notation

<table>
<thead>
<tr>
<th>$n$</th>
<th>10</th>
<th>100</th>
<th>1,000</th>
<th>10,000</th>
<th>100,000</th>
<th>1,000,000</th>
<th>10,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^n$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$n$</td>
<td>2</td>
<td>6</td>
<td>9</td>
<td>13</td>
<td>16</td>
<td>19</td>
<td>19</td>
</tr>
<tr>
<td>$n \cdot \log n$</td>
<td>20</td>
<td>644</td>
<td>9,915</td>
<td>10^13</td>
<td>10^16</td>
<td>10^19</td>
<td>10^19</td>
</tr>
<tr>
<td>$n^2$</td>
<td>10^4</td>
<td>10^9</td>
<td>10^14</td>
<td>10^19</td>
<td>10^24</td>
<td>10^29</td>
<td>10^29</td>
</tr>
<tr>
<td>$n^3$</td>
<td>10^7</td>
<td>10^15</td>
<td>10^21</td>
<td>10^27</td>
<td>10^33</td>
<td>10^39</td>
<td>10^39</td>
</tr>
<tr>
<td>$2^n$</td>
<td>10^10</td>
<td>10^19</td>
<td>10^29</td>
<td>10^39</td>
<td>10^49</td>
<td>10^59</td>
<td>10^59</td>
</tr>
</tbody>
</table>

Figure 9.3a
A comparison of growth-rate functions: a) in tabular form

The Efficiency of Searching Algorithms

- Binary search
  - Strategy
    - To search a sorted array for a particular item
    - Repeatedly divide the array in half
    - Determine which half the item must be in, if it is indeed present, and discard the other half
  - Efficiency
    - Worst case: $O(\log n)$
    - For large arrays, the binary search has an enormous advantage over a sequential search

Sorting Algorithms and Their Efficiency

- Sorting
  - A process that organizes a collection of data into either ascending or descending order
- Categories of sorting algorithms
  - An internal sort
    - Requires that the collection of data fit entirely in the computer’s main memory
  - An external sort
    - The collection of data will not fit in the computer’s main memory all at once but must reside in secondary storage

Sorting Algorithms and Their Efficiency

- Data items to be sorted can be
  - Integers
  - Character strings
  - Objects
- Sort key
  - The part of a record that determines the sorted order of the entire record within a collection of records
Selection Sort

• Analysis
  – Selection sort is $O(n^2)$
• Advantage of selection sort
  – It does not depend on the initial arrangement of the data
• Disadvantage of selection sort
  – It is only appropriate for small $n$

Terminology

• A binary search tree
  – A binary tree that has the following properties for each node $n$:
    • $n$’s value is greater than all values in its left subtree $T_L$
    • $n$’s value is less than all values in its right subtree $T_R$
    • Both $T_L$ and $T_R$ are binary search trees

• The height of trees
  – Level of a node $n$ in a tree $T$:
    • If $n$ is the root of $T$, it is at level 1
    • If $n$ is not the root of $T$, its level is 1 greater than the level of its parent
  – Height of a tree $T$ defined in terms of the levels of its nodes:
    • If $T$ is empty, its height is 0
    • If $T$ is not empty, its height is equal to the maximum level of its nodes

• Complete binary trees
  – A binary tree $T$ of height $h$ is complete if:
    • All nodes at level $h-2$ and above have two children each, and
    • When a node at level $h-1$ has children, all nodes to its left at the same level have two children each, and
    • When a node at level $h-1$ has one child, it is a left child

• Balanced binary trees
  – A binary tree is balanced if the height of any node’s right subtree differs from the height of the node’s left subtree by no more than 1
  • Full binary trees are complete
  • Complete binary trees are balanced
Properties of trees

- No cycles
  - a tree is a connected acyclic graph
- Every two nodes are connected by exactly one path
  - simplest way to connect given nodes
- Connected graph with n nodes and n-1 edges

General Operations of the ADT Binary Tree

- General operations of the ADT binary tree
  - createBinaryTree (rootItem, leftTree, rightTree)
  - setRootItem(newItem)
  - attachLeft(newItem) throws TreeException
  - attachRight(newItem) throws TreeException
  - attachLeftSubtree(leftTree) throws TreeException
  - attachRightSubtree(rightTree) throws TreeException
  - detachLeftSubtree() throws TreeException
  - detachRightSubtree() throws TreeException

Traversals of a Binary Tree

Figure 10.9
Traversals of a binary tree: a) preorder; b) inorder; c) postorder

Possible Representations of a Binary Tree

- An array-based representation
  - A Java class is used to define a node in the tree
  - A binary tree is represented by using an array of tree nodes
  - Each tree node contains a data portion and two indexes
    (one for each of the node's children)
  - Requires the creation of a tree list which keeps track of available nodes

Possible Representations of a Binary Tree

Figure 10.10a
a) A binary tree of names

Figure 10.10b
b) An array-based implementation

Possible Representations of a Binary Tree

Figure 10.11
Level-by-level numbering of a complete binary tree

Figure 10.12
An array-based implementation of the complete binary tree in Figure 10.11
A Reference-Based Binary Tree

```java
public class TreeNode {
    private Object item;
    private TreeNode leftChild;
    private TreeNode rightChild;
    // constructors, getLeft() // getRight() getItem() etc
}
```

Tree Traversals

- Basic tree traversals
  - In-order, pre-order, post-order
- Recursive implementations
- Tree iterator implements these tree traversals

The ADT Binary Search Tree

- ADT binary search tree
  - Searching for a particular item
- Each node in a binary search tree satisfies the following properties
  - Its value is greater than all values in its left subtree \( T_L \)
  - Its value is less than all values in its right subtree \( T_R \)
  - Both \( T_L \) and \( T_R \) are binary search trees

The ADT Binary Search Tree

- Record
  - A group of related items, called fields, that are not necessarily of the same data type
- Field
  - A data element within a record
- A data item in a binary search tree has a specially designated search key
  - A search key is the part of a record that identifies it within a collection of records
- KeyedItem class
  - Contains the search key as a data field and a method for accessing the search key
  - Must be extended by classes for items that are in a binary search tree

The ADT Binary Search Tree

- Operations of the ADT binary search tree
  - Insert a new item into a binary search tree
  - Delete the item with a given search key from a binary search tree
  - Retrieve the item with a given search key from a binary search tree
  - Traverse the items in a binary search tree in preorder, inorder, or postorder

The ADT Binary Search Tree

- Figure 10.17
  - A binary search tree

Binary Search Tree: Insertion

- Figure 10.21c
  - Insertion at a leaf
  - TreeNode
  - Insert
  - Keynode
  - Value
  - LeftChild
  - RightChild
  - Item

Figure 10.21c
- Insertion at a leaf
  - TreeNode
  - Insert
  - Keynode
  - Value
  - LeftChild
  - RightChild
  - Item
The Efficiency of Binary Search Tree Operations

• Theorem 10.2
  A full binary tree of height $h \geq 0$ has $2^h - 1$ nodes

• Theorem 10.3
  The maximum number of nodes that a binary tree of height $h$ can have is $2^h - 1$

General Trees

• An $n$-ary tree
  – A generalization of a binary tree whose nodes can have no more than $n$ children

2-3 Trees

• A 2-3 tree
  – Has 2-nodes and 3-nodes
  – A 2-node
    – A node with one data item and two children
  – A 3-node
    – A node with two data items and three children
  – Is not a binary tree
  – Is never taller than a minimum height binary tree
  – A 2-3 tree with $n$ nodes never has height greater than $\lceil \log_2(n + 1) \rceil$

2-3 Trees: The Insertion Algorithm

• To insert an item $I$ into a 2-3 tree
  – Locate the leaf at which the search for $I$ would terminate
  – Insert the new item $I$ into the leaf
  – If the leaf now contains only two items, you are done
  – If the leaf now contains three items, split the leaf into two nodes, $n_1$ and $n_2$

2-3 Trees: The Deletion Algorithm

• When analyzing the efficiency of the insertItem and deleteItem algorithms, it is sufficient to consider only the time required to locate the item
• A 2-3 operation is $O(\log_2 n)$
• A 2-3 tree is a compromise
  – Searching a 2-3 tree is not quite as efficient as searching a binary search tree of minimum height
  – A 2-3 tree is relatively simple to maintain
2-3-4 Trees

• Rules for placing data items in the nodes of a 2-3-4 tree
  – A 2-node must contain a single data item whose search keys satisfy the relationships pictured in Figure 12.3a
  – A 3-node must contain two data items whose search keys satisfy the relationships pictured in Figure 12.3b
  – A 4-node must contain three data items whose search keys S, M, and L satisfy the relationship pictured in Figure 12.21
  – A leaf may contain either one, two, or three data items

Figure 12.21 A 4-node in a 2-3-4 tree

2-3-4 Trees: Inserting into a 2-3-4 Tree

• The insertion algorithm for a 2-3-4 tree
  – Splits a node by moving one of its items up to its parent node
  – Splits 4-nodes as soon as it encounters them on the way down the tree from the root to a leaf
  • Result: when a 4-node is split and an item is moved up to the node’s parent, the parent cannot possibly be a 4-node and can accommodate another item

2-3-4 Trees: Splitting 4-nodes During Insertion

• A 4-node is split as soon as it is encountered during a search from the root to a leaf
• The 4-node that is split will
  – Be the root, or
  – Have a 2-node parent, or
  – Have a 3-node parent

Figure 12.28 Splitting a 4-node root during insertion

2-3-4 Trees: Deleting from a 2-3-4 Tree

• The deletion algorithm for a 2-3-4 tree
  – Locate the node n that contains the item theItem
  – Find theItem’s inorder successor and swap it with theItem (deletion will always be at a leaf)
  – If that leaf is a 3-node or a 4-node, remove theItem
  – To ensure that theItem does not occur in a 2-node
    • Transform each 2-node encountered into a 3-node or a 4-node

2-3-4 Trees: Remarks

• Advantage of 2-3 and 2-3-4 trees
  – Easy-to-maintain balance
  – Trees grow or shrink at the root
• Insertion and deletion algorithms for a 2-3-4 tree require fewer steps that those for a 2-3 tree
  – 2-3-4 tree operations are one-pass
• Allowing nodes with more than four children is counterproductive

Figure 12.21 A 4-node in a 2-3-4 tree
Red-Black Trees

- A 2-3-4 tree
  - Advantages
    - It is balanced
    - Its insertion and deletion operations use only one pass from root to leaf
  - Disadvantage
    - Requires more storage than a binary search tree
- A red-black tree
  - A special binary search tree
  - Used to represent a 2-3-4 tree
  - Has the advantages of a 2-3-4 tree, without the storage overhead

Red-Black Trees

- Basic idea
  - Represent each 3-node and 4-node in a 2-3-4 tree as an equivalent binary tree
- Red and black children references
  - Used to distinguish between 2-nodes that appeared in the original 2-3-4 tree and 2-nodes that are generated from 3-nodes and 4-nodes
  - Black references are used for child references in the original 2-3-4 tree
  - Red references are used to link the 2-nodes that result from the split 3-nodes and 4-nodes

Red-Black Trees

- Figure 12.31: Red-black representation of a 4-node
- Figure 12.32: Red-black representation of a 3-node

Red-Black Trees: Inserting and Deleting From a Red-Black Tree

- Figure 12.34: Splitting a red-black representation of a 4-node that is the root

Red-Black Trees: Inserting and Deleting From a Red-Black Tree

- Figure 12.35: Splitting a red-black representation of a 4-node whose parent is a 2-node
Sorting algorithms

- Bubblesort
- Insertionsort
- Selectionsort
- Mergesort
- Quicksort
- Shellsort
- Heapsort
- Bucketsort

Bubble Sort

- Analysis
  - Worst case: $O(n^2)$
  - Best case: $O(n)$

Insertion Sort

- Analysis
  - Worst case: $O(n^2)$
  - For small arrays
    - Insertion sort is appropriate due to its simplicity
  - For large arrays
    - Insertion sort is prohibitively inefficient

Shellsort

- Extension to insertion sort
- Exchange items far apart
- H-sorted sequence
  - taking every h-th element yields a sorted subsequence
  - every h-th element, starting anywhere
  - H-sorted sequence is h independent sorted sequences put together
Shellsort

- Main idea:
  - Sort for large values of \( h \)
  - This allows long-distance swaps
  - Proceed with smaller \( h \) values
  - Until \( h = 1 \)
- For every \( h \)-pass, use insertion sort
  - On the \( h \) subsequence

Mergesort

- Important divide-and-conquer sorting algorithms
  - Mergesort
  - Quicksort
- Mergesort
  - A recursive sorting algorithm
  - Gives the same performance, regardless of the initial order of the array items
  - Strategy
    - Divide an array into halves
    - Sort each half
    - Merge the sorted halves into one sorted array

Mergesort analysis

- Mergesort running time

\[
T(n) = \begin{cases} 
0 & \text{if } n = 1 \\
2T(n/2) + T(n/2) + n & \text{otherwise}
\end{cases}
\]

Mergesort

- Analysis
  - Worst case: \( O(n \times \log_2 n) \)
  - Average case: \( O(n \times \log_2 n) \)
- Advantage
  - It is an extremely efficient algorithm with respect to time
- Drawback
  - It requires a second array as large as the original array
QuickSort

- **QuickSort**
  - A divide-and-conquer algorithm
  - **Strategy**
    - Partition an array into items that are less than the pivot and those that are greater than or equal to the pivot
    - Sort the left section
    - Sort the right section

![A partition about a pivot](image)

**Figure 9.12**
A partition about a pivot

- Using an invariant to develop a partition algorithm
  - **Invariant for the partition algorithm**
    - The items in region $S_1$ are all less than the pivot, and those in $S_2$ are all greater than or equal to the pivot

![Invariant for the partition algorithm](image)

**Figure 9.14**
Invariant for the partition algorithm

- **Analysis**
  - **Worst case**
    - QuickSort is $O(n^2)$ when the array is already sorted and the smallest item is chosen as the pivot

![A word-case partitioning with quicksort](image)

**Figure 9.19**
A word-case partitioning with quicksort

Radix Sort

- **Radix sort**
  - Treats each data element as a character string
  - **Strategy**
    - Repeatedly organize the data into groups according to the $i^{th}$ character in each element
  - **Analysis**
    - Radix sort is $O(n)$

![A radix sort of eight integers](image)

**Figure 9.21**
A radix sort of eight integers

**A Comparison of Sorting Algorithms**

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Worst case</th>
<th>Average case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection sort</td>
<td>$n^2$</td>
<td>$n^2$</td>
</tr>
<tr>
<td>Bubble sort</td>
<td>$n^2$</td>
<td>$n^2$</td>
</tr>
<tr>
<td>Insertion sort</td>
<td>$n^2$</td>
<td>$n^2$</td>
</tr>
<tr>
<td>Mergesort</td>
<td>$n \cdot \log n$</td>
<td>$n \cdot \log n$</td>
</tr>
<tr>
<td>QuickSort</td>
<td>$n^2$</td>
<td>$n \cdot \log n$</td>
</tr>
<tr>
<td>Radix sort</td>
<td>$n$</td>
<td>$n \cdot \log n$</td>
</tr>
<tr>
<td>Timsort</td>
<td>$n^2$</td>
<td>$n \cdot \log n$</td>
</tr>
<tr>
<td>Heapsort</td>
<td>$n \cdot \log n$</td>
<td>$n \cdot \log n$</td>
</tr>
</tbody>
</table>

![Approximate growth rates of time required for eight sorting algorithms](image)

**Figure 9.22**
Approximate growth rates of time required for eight sorting algorithms
The ADT Table

- The ADT table, or dictionary
  - Uses a search key to identify its items
  - Its items are records that contain several pieces of data

<table>
<thead>
<tr>
<th>City</th>
<th>Country</th>
<th>Population</th>
</tr>
</thead>
<tbody>
<tr>
<td>Athens</td>
<td>Greece</td>
<td>2,700,000</td>
</tr>
<tr>
<td>Barcelona</td>
<td>Spain</td>
<td>1,600,000</td>
</tr>
<tr>
<td>Cairo</td>
<td>Egypt</td>
<td>9,000,000</td>
</tr>
<tr>
<td>London</td>
<td>England</td>
<td>7,800,000</td>
</tr>
<tr>
<td>New York</td>
<td>U.S.A.</td>
<td>7,300,000</td>
</tr>
<tr>
<td>Paris</td>
<td>France</td>
<td>2,200,000</td>
</tr>
<tr>
<td>Rome</td>
<td>Italy</td>
<td>2,600,000</td>
</tr>
<tr>
<td>Toronto</td>
<td>Canada</td>
<td>3,200,000</td>
</tr>
<tr>
<td>Venice</td>
<td>Italy</td>
<td>300,000</td>
</tr>
</tbody>
</table>

Selecting an Implementation

- Categories of linear implementations
  - Unsorted, array based
  - Unsorted, reference based
  - Sorted (by search key), array based
  - Sorted (by search key), reference based

<table>
<thead>
<tr>
<th>size</th>
<th>0</th>
<th>1</th>
<th>size + 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>item</td>
<td>0</td>
<td>1</td>
<td>Min_value</td>
</tr>
</tbody>
</table>

Scenario A: Insertion and Traversal in No Particular Order

- An unsorted order is efficient
  - Both array based and reference based tableInsert operation is $O(1)$
- Array based versus reference based
  - If a good estimate of the maximum possible size of the table is not available
  - Reference based implementation is preferred
  - If a good estimate of the maximum possible size of the table is available
  - The choice is mostly a matter of style

Scenario B: Retrieval

- Binary search
  - An array-based implementation
    - Binary search can be used if the array is sorted
    - A reference-based implementation
    - Binary search can be performed, but is too inefficient to be practical
  - A binary search of an array is more efficient than a sequential search of a linked list
    - Binary search of an array
      - Worst case: $O(n)$
    - Sequential search of a linked list
      - $O(n)$
- For frequent retrievals
  - If the table’s maximum size is known
    - A sorted array-based implementation is appropriate
  - If the table’s maximum size is not known
    - A binary search tree implementation is appropriate
Scenario C: Insertion, Deletion, Retrieval, and Traversal in Sorted Order

- Steps performed by both insertion and deletion
  - Step 1: Find the appropriate position in the table
  - Step 2: Insert into (or delete from) this position
- Step 1
  - An array-based implementation is superior than a reference-based implementation
- Step 2
  - A reference-based implementation is superior than an array-based implementation
  - A sorted array-based implementation shifts data during insertions and deletions

Summary of Tables

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Insertion</th>
<th>Deletion</th>
<th>Retrieval</th>
<th>Traversal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted array based</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Unsorted pointer based</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Sorted array based</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Sorted pointer based</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Binary search tree</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(n)</td>
</tr>
</tbody>
</table>

The ADT Priority Queue: A Variation of the ADT Table

- The ADT priority queue
  - Orders its items by a priority value
  - The first item removed is the one having the highest priority value
- Operations of the ADT priority queue
  - Create an empty priority queue
  - Determine whether a priority queue is empty
  - Insert a new item into a priority queue
  - Retrieve and then delete the item in a priority queue with the highest priority value

Possible implementations (Continued)

- Binary search tree implementation
  - Appropriate for any priority queue

Some implementations of the ADT priority queue: a) array based; b) reference based; c) binary search tree
Heaps

• A heap is a complete binary tree
  – That is empty
  or
  – Whose root contains a search key greater
    than or equal to the search key in each of
    its children, and
  – Whose root has heaps as its subtrees

Heaps: An Array-based Implementation of a Heap

• Data fields
  – items: an array of heap items
  – size: an integer equal to the number of items in the heap

Heaps: heapDelete

• Step 1: Return the item in the root
  – Results in disjoint heaps

Heaps: heapDelete

• Step 2: Copy the item from the last node into the root
  – Results in a semiheap

Heaps: heapDelete

• Step 3: Transform the semiheap back into a heap
  – Performed by the recursive algorithm heapRebuild

Heaps: heapDelete

• Efficiency
  – heapDelete is \( O(\log n) \)
Heaps: heapInsert

- **Strategy**
  - Insert new item into the bottom of the tree
  - Trickle new item up to appropriate spot in the tree
- **Efficiency**: \(O(\log n)\)
- **Heap class**
  - Represents an array-based implementation of the ADT heap

Heapsort

- **Strategy**
  - Transforms the array into a heap
  - Removes the heap's root (the largest element) by exchanging it with the heap's last element
  - Transforms the resulting semiheap back into a heap
- **Efficiency**
  - Compared to mergesort
    - Both heapsort and mergesort are \(O(n \log n)\) in both the worst and average cases
    - Advantage over mergesort
    - Heapsort does not require a second array
  - Compared to quicksort
    - Quicksort is the preferred sorting method

Summary

- A heap that uses an array-based representation of a complete binary tree is a good implementation of a priority queue when you know the maximum number of items that will be stored at any one time
- **Efficiency**
  - Heapsort, like mergesort, has good worst-case and average-case behaviors, but neither algorithms is as good in the average case as quicksort
  - Heapsort has an advantage over mergesort in that it does not require a second array

Red-Black Trees

- **A 2-3-4 tree**
  - **Advantages**
    - It is balanced
    - Its insertion and deletion operations use only one pass from root to leaf
  - **Disadvantage**
    - Requires more storage than a binary search tree
- **A red-black tree**
  - A special binary search tree
  - Can be used to represent a 2-3-4 tree
  - Has the advantages of a 2-3-4 tree, without the storage overhead
Red-Black Trees

- Basic idea
  - Represent each 3-node and 4-node in a 2-3-4 tree as an equivalent binary tree
- Red and black children references
  - Used to distinguish between 2-nodes that appeared in the original 2-3-4 tree and 2-nodes that are generated from 3-nodes and 4-nodes
  - Black references are used for child references in the original 2-3-4 tree
  - Red references are used to link the 2-nodes that result from the split 3-nodes and 4-nodes

Red-Black equivalent

split nodes => red references;
original nodes => black references

Red-Black Trees: Inserting and Deleting From a Red-Black Tree

Other definitions

- Alternatively if you think of red-black trees without deriving them from 2-3-4 trees, there are corresponding definitions.
AVL Trees

- Invented by G.M. Adel'son-Vel'skii and E.M. Landis, "An algorithm for the organization of information" [1962]
- It is self-balancing: as such it is a self-organizing data structure
  - (can you think of any others?)
- It can be thought of as "nearly balanced" or in the context of AVL trees just "balanced".

AVL Trees

- Advantage
  - Height of an AVL tree with n nodes is always very close to the theoretical minimum (so it's good when you need lots of look-up)
- Disadvantage
  - An AVL tree implementation of a table is more difficult than other implementations (so it's a pain to program)

The ADT Priority Queue: A Variation of the ADT Table

- The ADT priority queue
  - Orders its items by a priority value
  - The first item removed is the one having the highest priority value
- Operations of the ADT priority queue
  - Create an empty priority queue
  - Determine whether a priority queue is empty
  - Insert a new item into a priority queue
  - Retrieve and then delete the item in a priority queue with the highest priority value

Summary

- A 2-3 tree and a 2-3-4 tree are variants of a binary search tree in which the balanced is easily maintained (hah!)
- The insertion and deletion algorithms for a 2-3-4 tree are more efficient than the corresponding algorithms for a 2-3 tree
- A red-black tree is a binary tree representation of a 2-3-4 tree that requires less storage than a 2-3-4 tree
- An AVL tree is a binary search tree that is guaranteed to remain balanced
Hashing

HashMap diagram

Key Partitions

Resolving Collisions

Resolving Collisions (Continued)

The Efficiency of Hashing

Figure 12.50
Summary

• A hash function should be extremely easy to compute and should scatter the search keys evenly throughout the hash table.
• A collision occurs when two different search keys hash into the same array location.
• Hashing does not efficiently support operations that require the table items to be ordered.
• Hashing as a table implementation is simpler and faster than balanced search tree implementations when table operations such as traversal are not important to a particular application.

Graphs

Terminology

• G = (V, E)
• A graph G consists of two sets
  – A set V of vertices, or nodes
  – A set E of edges
• A subgraph
  – Consists of a subset of a graph’s vertices and a subset of its edges
• Adjacent vertices
  – Two vertices that are joined by an edge

Graphs as ADTs

• Operations of the ADT graph (Continued)
  – Determine whether an edge exists between two given vertices; for weighted graphs, return weight value
  – Insert a vertex in a graph whose vertices have distinct search keys that differ from the new vertex’s search key
  – Insert an edge between two given vertices in a graph
  – Delete a particular vertex from a graph and any edges between the vertex and other vertices
  – Delete the edge between two given vertices in a graph
  – Retrieve from a graph the vertex that contains a given search key

Implementing Graphs

Figure 14-6

a) A directed graph and b) its adjacency matrix
Implementing Graphs

- Adjacency list for an undirected graph
  - Treats each edge as if it were two directed edges in opposite directions

![Figure 14-9](https://example.com/figure14-9)

a) A weighted undirected graph and b) its adjacency list

Graph Traversals

- A graph-traversal algorithm
  - Visits all the vertices that it can reach
  - Visits all vertices of the graph if and only if the graph is connected
    - A connected component
    - The subset of vertices visited during a traversal that begins at a given vertex
    - Can loop indefinitely if a graph contains a loop
    - To prevent this, the algorithm must
      - Mark each vertex during a visit, and
      - Never visit a vertex more than once

![Figure 13.10](https://example.com/figure13.10)

Visitation order for a) a depth-first search; b) a breadth-first search

Graph Traversals

- A graph-traversal algorithm
  - Visits all the vertices that it can reach
  - Visits all vertices of the graph if and only if the graph is connected
    - A connected component
    - The subset of vertices visited during a traversal that begins at a given vertex
    - Can loop indefinitely if a graph contains a loop
    - To prevent this, the algorithm must
      - Mark each vertex during a visit, and
      - Never visit a vertex more than once

Depth-First Search

- Depth-first search (DFS) traversal
  - Visit vertex then explore all adjacent vertices recursively
  - A first visited, first explored strategy
  - An iterative form uses a queue
  - A recursive form is also possible

```
function DFS(u)
    if not marked(u) then
        marked(u) = true;
        for all s ∈ succs(u)
            DFS(s)
        endfor
    endif
endfunction
```

Algorithm

- Set of successors
  - succ(u) = {v | (u, v) ∈ E}
- Initialization
  - for all nodes u
    - marked(u) = false;
    - DFS(start)

Breath-First Search

- Breath-First Search (BFS) traversal
  - Visits neighbors first before exploring depth.
  - A first visited, first explored strategy
  - An iterative form uses a queue
  - A recursive form is possible, but not simple
Breath-First Search
Algorithm
```java
void BFS(Node u) {
    q = new Queue();
    q.enq(u); marked[u] = true;
    while(!q.isEmpty()) {
        u = q.deq();
        for all v in succ(u)
            if (!marked[v]) {
                q.enq(v);
                marked[v] = true;
            }
    }
}
```

Applications of Graphs: Topological Sorting
- **Topological order**
  - A list of vertices in a directed graph without cycles such that vertex \(x\) precedes vertex \(y\) if there is a directed edge from \(x\) to \(y\) in the graph
  - There may be several topological orders in a given graph
- **Topological sorting**
  - Arranging the vertices into a topological order

Algorithm for Top-Sort
```
function TopSort(u)
    if not marked[u] then
        marked[u] = true;
        for all s in succe(s) TopSort(s)
        order[i] = u;
        i = i - 1;
    endif
endfunction
```

Spanning Trees
- A tree
  - An undirected connected graph without cycles
- A spanning tree of a connected undirected graph \(G\)
  - A subgraph of \(G\) that contains all of \(G\)'s vertices and enough of its edges to form a tree
- To obtain a spanning tree from a connected undirected graph with cycles
  - Remove edges until there are no cycles

Spanning Trees
- You can determine whether a connected graph contains a cycle by counting its vertices and edges
  - A connected undirected graph that has \(n\) vertices must have at least \(n - 1\) edges
  - A connected undirected graph that has \(n\) vertices and exactly \(n - 1\) edges cannot contain a cycle
  - A connected undirected graph that has \(n\) vertices and more than \(n - 1\) edges must contain at least one cycle
The DFS Spanning Tree

- To create a depth-first search (DFS) spanning tree
  - Traverse the graph using a depth-first search and mark the edges that you follow
  - After the traversal is complete, the graph's vertices and marked edges form the spanning tree

The BFS Spanning Tree

- To create a breadth-first search (BFS) spanning tree
  - Traverse the graph using a breadth-first search and mark the edges that you follow
  - When the traversal is complete, the graph's vertices and marked edges form the spanning tree

Minimum Spanning Tree

- Minimum spanning tree
  - A spanning tree for which the sum of its edge weights is minimal
- Prim's algorithm
  - Finds a minimal spanning tree that begins at any vertex
  - Strategy
    - Find the least-cost edge \((v, u)\) from a visited vertex \(v\) to some unvisited vertex \(u\)
    - Mark \(u\) as visited
    - Add the vertex \(u\) and the edge \((v, u)\) to the minimum spanning tree
    - Repeat the above steps until there are no more unvisited vertices

Summary

Graph searching
  - Depth-first search goes as deep into the graph as it can before backtracking
  - Breadth-first search visits all possible adjacent vertices before traversing further into the graph
  - Topological sorting produces a linear order of the vertices in a directed graph without cycles
  - Trees are connected undirected graphs without cycles
    - A spanning tree of a connected undirected graph is a subgraph that contains all the graph's vertices and enough of its edges to form a tree
    - A minimum spanning tree for a weighted undirected graph is a spanning tree whose edge-weight sum is minimal

Geometric problems

- Example: Given \(n\) points on a plane, compute their convex hull
  - Points are given by their \((x, y)\) coordinates

Convex hulls

- A set \(S\) is called convex iff for any \(p, q\) in \(S\) the line \(pq\) is completely contained in \(S\)
- The convex hull of a set \(S\) is the smallest convex set that contains \(S\)
  - The convex hull of \(S\) is the intersection of all convex sets that contain \(S\)
  - The convex hull of \(S\) is denoted \(CH(S)\)
Computing the convex hull

- Given a set of points \( P = \{p_1, p_2, \ldots, p_n\} \), the convex hull is usually given as a list of points in \( P \) that are vertices of \( P \) in clockwise order.
- Testing if points \( p, q \) are on the convex hull: all of \( P \) except \( p \) and \( q \) must be on the same side of \( pq \).

Checking for left turns

- If there is a left turn remove the middle point, and check again the “new” last three points.

Convex hull algorithm

1. sort points by x coordinate in sequence \( p_1, p_2, \ldots, p_n \)
2. add \( p_1 \) and \( p_2 \) to the upper hull list \( L_{upper} \)
3. for \( i := 3 \) to \( n \)
   4. append \( p_i \) to \( L_{upper} \)
   5. while last 3 points in \( L_{upper} \) do not make a right turn
      6. delete the middle of the last 3 points from \( L_{upper} \)
7. Add the points \( p_n \) and \( p_{n-1} \) to the lower hull list \( L_{lower} \)
8. for \( i := n-2 \) down to \( 1 \)
   9. append \( p_i \) to \( L_{lower} \)
10. while last 3 points in \( L_{lower} \) do not make a right turn
11. delete the middle of the last 3 points from \( L_{lower} \)
12. To avoid duplication, remove first and last points of \( L_{upper} \)
13. return the two lists \( (L_{lower}, \text{appended to } L_{upper}) \)

Convex hulls

- Graham’s scan works in time \( O(n \log n) \) (sorting)
- Convex hull computation requires \( n \log n \) steps (lower bound)
- However, the lower bound corresponds to cases where almost all the points are on the hull.
- In practice very few points may appear on the hull.

1-dimensional query

- \( P = \{p_1, p_2, \ldots, p_n\} \) points on a line in other words this is a set of numbers (x-coordinates only)
- locate point query: find the number \( x \).
- use a binary search tree

kd-trees

- alternate splitting on \( x \) and \( y \) coordinates
- pick the point with the median coordinate (to split evenly “balanced” tree)
- kd-trees may also be defined to store points only at the leaves.
kd-trees

- alternate splitting on x and y coordinates
- pick the point with the median coordinate (to split evenly - "balanced" tree)
- left subtree is the "low" subtree, right subtree is the "high" subtree

**Example query**

- find point with coordinates (7,3)

**Performance of kd-trees**

- a kd-tree
  - requires \(O(n)\) space
  - can be built in \(O(n \log n)\)
  - supports \(O(\sqrt{n+k})\) range queries
    where \(k\) is the number of reported points

**Better query time?**

- can we beat the square root of \(n\) query time?
- the data structure must use more space to achieve a better query time
  (lower bound by Chazelle)
- range trees: use \(O(n \log n)\) space and achieve \(O(\log^2 n+k)\) query time
- with some additional techniques, query time can be improved to \(O(\log n+k)\)

**Kd-trees and range trees**

- kd-trees: search alternates between x and y coordinates
- range trees: search on x-coordinates then y-coordinates
range trees

- the main tree is a balanced binary search tree on the x coordinates
- for every internal or leaf node v, there is an associated balanced binary search tree $T_v$ on the y-coordinate for all nodes that appear in the subtree rooted at v.

medians in linear time

- break up in groups of 5
- sort each group
- find the median of the medians
- partition around the median of the medians
- if not done, continue recursively on one side of the partition

1d range query

- use a balanced search tree
- do a modified binary search

1d range query (list, tree, low, high) {
  1. if (tree is empty) return list
  2. if (high < tree.value) return rangeQuery(tree.getLeft);
  3. if (low > tree.value) return rangeQuery(tree.getRight);
  4. if (low <= tree.value <= high) {
     5. add tree.value to the list
     6. list = rangeQuery(list, tree.getLeft(), low, high);
     7. return rangeQuery(list, tree.getRight(), low, high);
  8. }
  9. }